University of Cape Town

Masters Dissertation

Self-Attention Policy Architectures for Reinforcement Learning Under Partial Observability

Author:
Jeremy du Plessis

Supervisor:
Assoc. Prof. Jonathan Shock
Co-Supervisor:
Prof. Benjamin Rosman

A dissertation submitted in fulfilment of the requirements for the degree of Master of Science
in the
Department of Mathematics and Applied Mathematics

01 February 2024

Declaration of Authorship

I, Jeremy du Plessis, hereby declare that the work on which this dissertation is based is my original work (except where acknowledgements indicate otherwise) and that neither the whole work nor any part of it has been, is being, or is to be submitted for another degree in this or any other university. I authorise the University of Cape Town to reproduce, for the purpose of research, either the whole or any portion of the contents in any manner whatsoever.

Signature: Signed by candidate Date: 01-02-2024.

Abstract

Intermittent unavailability of sensory signals due to sensor failure and/or latency is a problem encountered in production environments such as in large manufacturing plants, for example. Deep reinforcement learning offers a natural solution for process control and optimisation in such environments. However, a shortcoming of conventional agent policy architectures in this instance is an inability to handle variable-sized inputs composed of available sensory signals, thus requiring the imputation of unavailable sensory signals with data which necessarily constitutes noise. We explore self-attention-based policy architectures as a solution to this problem, demonstrating their robustness under conditions of high partial observability on different reinforcement learning benchmark tasks, and explore the advantages and disadvantages offered by our solution over conventional policy architectures. Additionally, we propose a novel hard attention mechanism, used in conjunction with our proposed policy architecture, enabling the agent to attend to the most salient sensory signals and allowing for greater interpretability of the agent's decision-making.

Acknowledgements

A little over 7 years ago, at the ripe old age of 25, I decided to leave my career in advertising to start studying all over again, enrolling in an undergraduate degree program for the second time. In the same year, I met my now-wife, Kathryn, who, having made more responsible choices in her young life than I had, was well into her career. Being at such different stages of life was not easy, but from the very start, Kathryn was my greatest supporter. To say that I could not have completed this dissertation without her is an understatement. I am eternally grateful for her patience, constant encouragement, sacrificing mornings, nights, weekends, and holidays with me, for all the extra housework, and for always believing in me. I love you more than words can express.

The halls of the academy are replete with stories of neglectful supervisors—a truth that becomes more apparent the more postgraduate students you speak to. For this reason, among many others, I am incredibly grateful to both Assoc. Professor Jonathan Shock and Professor Benjamin Rosman for being the most attentive, responsive, and supportive supervisors I could have asked for. To Jonathan especially, who has been my lecturer and/or supervisor since MAM1000W in my first year; thank you for your limitless attention and patience and for giving so much of yourself to your students. For carefully reading every draft of every section (and all the bits of writing which never amounted to anything!), for answering every question on Slack whether during the day or night, for allaying my anxieties, and for believing in me and pushing me to do my best work always. My academic experience and the immense value I have gotten out of it are in very large part thanks to you, and I am forever grateful for everything you have done in support of me and my education.

I would also like to extend my deep and sincere thanks to my wider family, including my mother and father, Corinne and Andre, as well as my Aunt Shelley, and my late grandfather, Reg. Thank you for your endless support and encouragement. I am so incredibly grateful for you all.

Lastly, funding for this research was provided by ETDP-SETA[1] over the course of the 2022 and 2023 academic years, for which I am grateful.

[1]Education, Training and Development Practices Sector Education and Training Authority.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ADRQN	**A**ction-specific **D**eep **R**ecurrent **Q**-**N**etwork
ANN	**A**rtificial **N**eural **N**etwork
CNN	**C**onvolutional **N**eural **N**etwork
DARQN	**D**eep **A**ttention **R**ecurrent **Q**-**N**etwork
DQN	**D**eep **Q**-**N**etwork
DRQN	**D**eep **R**ecurrent **Q**-**N**etwork
DTQN	**D**eep **T**ransformer **Q**-**N**etwork
GPI	**G**eneralised **P**olicy **I**teration
GPT	**G**enerative **P**re-trained **T**ransformer
GRU	**G**ated **R**ecurrent **U**nit
LSTM	**L**ong **S**hort-**T**erm **M**emory
MAE	**M**asked **A**uto**e**ncoder
MDP	**M**arkov **D**ecision **P**rocess
MHA	**M**ulti-**H**ead **A**ttention
ML	**M**achine **L**earning
MLP	**M**ulti-**L**ayered **P**erceptron
MSE	**M**ean **S**quared **E**rror
NLP	**N**atural **L**anguage **P**rocessing
PINN	**P**ermutation **I**nvariant **N**eural **N**etwork
POMDP	**P**artially **O**bservable **M**arkov **D**ecision **P**rocess
PPO	**P**roximal **P**olicy **O**ptimisation

ReLU **R**ectified **L**inear **U**nit

RGB **R**ed **G**reen **B**lue

RL **R**einforcement **L**earning

RMC **R**elational **M**emory **C**ore

RNN **R**ecurrent **N**eural **N**etwork

SGD **S**tochastic **G**radient **D**escent

TD **T**emporal **D**ifference

ViT **V**ision **T**ransformer

Summary of Notation

$\mathcal{S}$	The set of all environment states.
Ω	The set of all environment observations.
$\mathcal{A}$	The set of all environment actions.
$\mathcal{A}(s)$	The set of all environment actions possible in some state, $s \in \mathcal{S}$.
R	The environment reward function.
P	The environment state transition probability function.
$\mathcal{O}$	The environment observation transition probability function.
s_t	The state of the environment at time step t.
o_t	A partial observation derived from the state of the environment at time step t.
r_t	The reward obtained by the agent after taking a given action in a given state at time step $t-1$.
T	The final time step of an episode.
G_t	The return following time step t, computed as the discounted sum of rewards obtained following time step t.
γ	The discount factor (in the range $[0, 1]$) used for computing returns.
π	The agent's decision-making policy which maps states to actions.
π_θ	An ANN modelling π parameterised by $\boldsymbol{\theta}$.
v^π	The true state value function under a given policy, π.
v^*	The optimal state value function.
V	An estimate of the true state value function maintained by an agent
V_θ	An ANN modelling V parameterised by $\boldsymbol{\theta}$.

q^{π}	The true state-action value function under a given policy, π.
q^{*}	The optimal state-action value function.
Q	An estimate of the true state-action value function maintained by an agent.
Q_{θ}	An ANN modelling Q parameterised by $\boldsymbol{\theta}$.
ρ	The behaviour distribution (Q-learning) or the trajectory distribution (Policy Gradient).
φ	An arbitrary activation function.
h_t	Hidden state of an RNN at time step t.
E	A matrix wherein each row is an embedding vector.
$J(\boldsymbol{\theta})$	The objective function for optimizing policy/value function.
$\hat{A}_t$	The advantage estimate at time t.
ψ_t	Bit mask at time step t, indicating availability of sensory signals.
m_t	The 'global latent code' produced by a policy network at time t.
μ^{train}	Masking ratio during training.
μ^{eval}	Masking ratio during evaluation.

Chapter 1

Introduction

1.1 Motivation

Next time you're out for a leisurely stroll (ideally not on a bustling street), try this simple experiment. After taking a good look around, close your eyes and continue walking. You'll find that you can move forward with a certain level of confidence, visualising your surroundings in your mind and how they alter as you progress. As the uncertainty builds, you'll start to feel increasingly uncomfortable. Now, try opening and closing your eyes quickly to recalibrate the scene in your mind. You'll likely find your sense of certainty somewhat restored, allowing you to proceed. As you improve at visualising your surroundings, your confidence will grow, and you'll likely be able to extend the time between glimpses.

This seemingly child-like experiment illustrates a remarkable capability of the human brain: the ability to integrate partial sensory information of variable quantities over time in order to act and accomplish tasks, such as walking. Reflecting on this, it's clear that humans constantly operate under conditions of partial observability, both due to the physical limitations of our sensory apparatus and the fact that not all salient information required for optimal decision-making is known to an individual at any given moment.

In contrast to the animal brain, conventional layers used in artificial neural networks, such as dense or convolutional layers, can only process inputs of a fixed size [1]. This is a peculiar design choice, given that these networks are loosely based on the brain[2] and that the data used for training neural networks often contain missing values, necessitating meticulous pre-processing. The self-attention mechanism central to the recently popularised Transformer architecture, the adoption of which has led to significant performance gains in domains such as Natural Language Processing and Computer Vision, offers an alternative capable of handling variable-sized inputs.

Deep reinforcement learning provides a framework for learning sequential decision-making tasks using artificial neural network policies. In this dissertation, we aim to study the phenomenon outlined above, modelled as a partially observable reinforcement learning task. Specifically, we will compare conventional policy architectures, such as multi-layered perceptions and convolutional neural networks, to attention-based policy architectures. Our goal is to demonstrate that attention-based architectures present a promising, generalised alternative, better suited to tasks of this nature.

1.2 Problem Statement

1.2.1 Sensor Failure & Latency In Control Systems

Consider a sequential decision-making task, as depicted in Figure 1.1, characterised by a remote interaction between a controller and a system. The system maintains an internal state and executes the decisions of the

[1] Note that in animal sensing, the information flow is a complex, multi-directional process [27] [28]. We include the motivating example and the contrast between the animal brain and artificial neural networks not to draw hard parallels between them, but simply to provide an intuition for the problem we're trying to address.

[2] Generally speaking, not just the human brain.

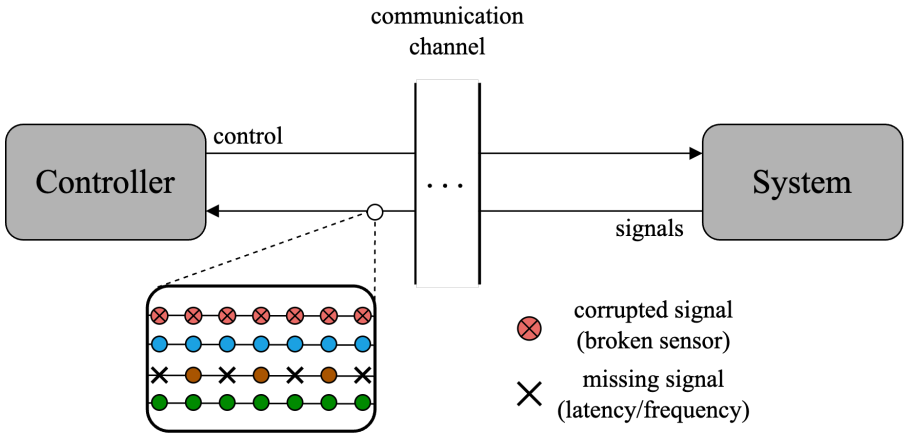


Figure 1.1.: An illustration of the problem of 'sensor failure and latency'. A system and a controller communicate over a remote channel: the system constantly emits signals that collectively encode the internal state of the system and form the basis for the controller's decisions, which are emitted at a fixed frequency in an effort to optimise the state of the system. A challenge arises when sensory signals are unavailable at the time a decision is required from the controller due to either (a) signal latency caused by variable sensor sampling rates, or (b) sensor failure, resulting in either a corrupt or unavailable signal. In such scenarios, the controller should be able to make a decision based on the limited information available.

controller which, in turn, influences how the state evolves. The system and the controller continuously exchange signals over a communication channel. The system emits a set of sensory signals which collectively define the state of the system and, in response, the controller emits decisions, in the form of a control signal, which are required at a fixed frequency by the system. The task's complexity arises from two primary challenges associated with the sensory apparatus:

1. **Asynchrony and Latency**: The sensors may operate at different sampling rates and are subject to intermittent latency variations. As a result, some sensory signals may not be available to the controller when a decision is needed.

2. **Sensor Reliability**: The sensors are susceptible to occasional malfunctions, leading to the transmission of distorted or noisy sensory signals.

These complications necessitate the development of robust control strategies capable of accommodating asynchronous and potentially unreliable sensory inputs while maintaining effective system operation. A real-world manifestation of these challenges can be found in the manufacturing sector. In preparation for this dissertation, we consulted with an expert — a Senior Data Scientist at DataProphet, a company specialising in the development of deep learning solutions for process optimisation in large manufacturing facilities. The expert confirmed that incomplete and compromised data are recurrent obstacles encountered in their operational projects.

Note: for the sake of brevity, we will henceforth refer to this problem scenario as 'sensor failure'.

1.2.2 Low-Bandwidth Communication

A distinct but related problem to the scenario of sensor failure arises when the bandwidth - the capacity in bits per second at which data may be transferred - of the communication channel between the controller and the system is significantly smaller than the total size of the complete set of sensory signals which collectively define the state of the system (assuming the full set of sensory signals are available).

Under such conditions, the system must select a subset of the sensory signals to transmit over the communication channel. This selection may be determined in one of two ways. The first is via (uniform) random sampling, where the controller has no say in which sensory signals are transmitted. In this case, the problem

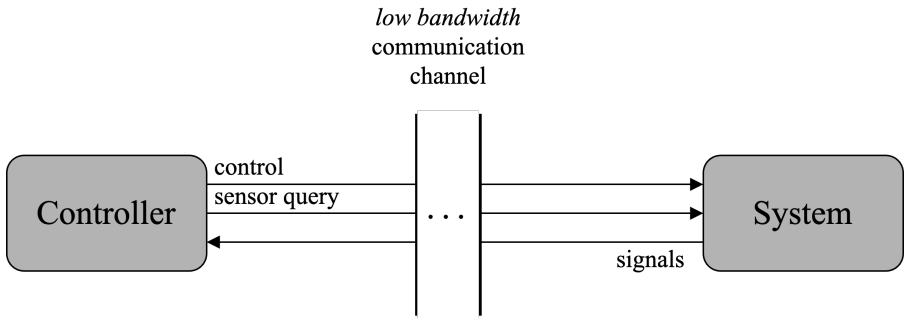


Figure 1.2.: An illustration of the problem of 'low-bandwidth communication'. A system and a controller communicate over a remote channel, continuously exchanging state and control signals at a fixed frequency to optimise the system's state. The challenge in this scenario arises from the channel's limited bandwidth — the rate at which it can transfer information between the system and the controller. As a result, the channel cannot transmit the entire set of sensory signals, which collectively define the system's state, at any one time step. Consequently, the controller must send a sensor query in addition to its control signals at each time step, instructing the system on which signals to transmit. Ultimately, the controller faces the challenge of optimal sensor querying for the purpose of process optimisation.

of low bandwidth communication is materially equivalent to the problem of intermittent sensor failure described in section 1.2.1. In the second scenario, the controller must send, at each time step, a control signal as well as a query that the system may use to select the subset of sensory signals to send back to the controller at the subsequent time step. Given that certain sensory signals will likely be more useful than others in making a decision at any given time, depending on the system's state, the principal challenge in this problem setup is that of optimal querying of the sensory signals under a limited bandwidth budget.

1.3 Research Questions

In this dissertation, we will investigate viable strategies for addressing the problem of partial observability arising from intermittent sensor failure, as well as the problem of low-bandwidth communication, within the domain of deep reinforcement learning (RL). To do so, we frame these scenarios as RL tasks, formalized as partially observable Markov Decision Processes, in which the controller is a decision-making RL agent and the system is a well-defined RL environment.

This research builds upon a few key findings in the literature, one of which comes from a recent paper titled *The Sensory Neuron as a Transformer: Permutation-Invariant Neural Networks for Reinforcement Learning* [25], in which the authors demonstrate the ability of Transformer-like-attention-based RL policy networks to handle arbitrary permutations of sensory signals. To gain an intuition for the problem we study here, it is useful to examine Figure 3.17, which illustrates the authors' experimental methodology. This setup involves both vector-based and vision-based tasks where the observations emitted by the environment are divided into smaller components each of which constitutes an individual "signal" received as input by a single network receptor, termed a "sensory neuron" by the authors. In the case of vector-based tasks, each sensory signal is a single vector component (a real number). In the case of vision-based tasks, an observation, consisting of a pixel grid (e.g. a video game screen), is subdivided into equal-sized patches, each constituting a sensory signal. One finding of this research was that the attention-based policy was robust under conditions of extreme partial observability wherein only a small subset of the sensory signals was made available to the agent during training and inference - this is a key finding in the literature which we seek to build upon in this dissertation through answering the research questions given below.

In the 'sensor failure' scenario, we will examine the dense and convolutional-based deep learning policy network architectures conventionally used to process observations in vector and vision-based reinforcement learning environments, respectively. We will then assess the manner and degree of performance degradation suffered by these conventional policy architectures under the conditions of partial observability induced by

different rates of sensor failure, which we will emulate experimentally with random masking. Specifically, we will seek to understand the impact of three different methods of imputation utilised in such conditions to maintain fixed-size observations, which is required by both dense and convolutional artificial neural network layers:

1. **Zero masking:** a naive method of imputation in which masked signals are imputed with zeros in order to emulate missing sensory signals.
2. **Noise masking:** a method of imputation, wherein masked signals are imputed with Gaussian Noise, designed to emulate noisy (corrupt) signals emitted by a faulty sensor.
3. **Forward-fill masking:** A method of imputation employed for handling missing values, as recommended by the industry expert at DataProphet. It involves imputing masked signals with the most recently observed value from the specific sensor.

We will then propose an alternative policy network architecture for addressing the specified problems, which incorporates elements of the Transformer architecture, such as self-attention, which, by contrast with the conventional policy architectures mentioned, is able to process variable-sized observations. We will seek to ascertain the potential advantages and disadvantages offered by our attention-based policy architecture over the conventional policy architectures across vector and vision-based reinforcement learning tasks, ultimately arguing that, while more research is required, our approach presents a promising alternative which offers the potential for enhanced robustness and efficacy as an architectural solution in such contexts.

Finally, in the 'low-bandwidth communication channel' scenario, where only a fixed proportion of the sensors may be queried at each time step due to a limitation on the amount of data that may be transmitted over the communication channel, we will seek to compare two variants of our attention agent - random vs non-random - which differ according to the method in which they sample signals from the available sensors. The 'random' variant samples signals according to a fixed, uniform probability distribution, whereas the 'non-random' variant samples signals according to an explicitly generated probability distribution over all sensors output from the agent's policy network, theoretically allowing the agent to query the sensors which are most salient for the purposes of optimal control. We seek to ascertain the advantages and disadvantages offered by the non-random generation of sensor queries over the random alternative.

With this in mind, we pose the following research questions which will form the basis for our experimentation and analysis:

1. *In the 'sensor failure' scenario, what is the manner and extent of performance degradation suffered by conventional policy network architectures using the imputation methods of zero masking, noise masking, and forward-fill masking under different rates of sensor failure?*
2. *What advantages and disadvantages might self-attention-based policy network architectures offer over conventional policy network architectures in such conditions?*
3. *In the 'low-bandwidth communication channel' scenario, what advantages and disadvantages are offered by the non-random generation of sensor queries over the random alternative?*

1.4 Hypotheses

Based on the research questions posed, the following hypotheses are proposed.

Hypotheses for Research Question 1:

- **H1.1: Impact of Noise Masking.** Noise masking, due to its non-constant nature and the out-of-distribution observations it causes, is expected to have the most severe impact on performance, especially at high rates of sensor failure/masking.
- **H1.2: Impact of Zero Masking.** Zero masking is hypothesised to have a less severe impact on performance than noise masking due to its constant nature, which may facilitate the policy network's learning to adapt to the noise constituted by the masking.

- **H1.3: Impact of Forward-fill Masking.** Forward-fill masking is expected to be the most effective imputation method at low rates of sensor failure, as it is likely to result in observations that are not statistically novel to the agent.
- **H1.4: Impact of Sensor Failure Rates.** Performance degradation is expected to be more severe at higher rates of sensor failure. However, equipped with a recurrent neural network layer, the policy networks are expected to be able to learn to handle the additional noise to some extent.
- **H1.5: Impact of Imputation Methods on Different Environments.** The impact of different imputation methods is expected to vary between environments. In our vision-based test environment, CarRacing-v2, imputation is expected to have a more severe impact due to the propagation of incorrect pixel values through the network. In contrast, in our vector-based test environment, Acrobot-v1, the policy network may be able to consider high-level features independently, mitigating the impact of imputation.

Hypotheses for Research Question 2:

- **H2.1: Advantage of Attention-based Policy Architecture.** The attention-based policy architecture is expected to have a performance advantage over dense or convolutional-based architectures, especially at high rates of sensor failure/masking, due to its ability to process variable-sized observations and eliminate the need for imputation.
- **H2.2: Disadvantage of Multi-head Attention.** However, based on the literature, the inclusion of Transformer-like multi-head attention in the policy network may result in lower sample efficiency, leading to potentially poorer performance under an equivalent computational budget. This disadvantage is expected to be more evident in the vision-based task due to its increased computational complexity relative to the vector-based task, as well as the fact that multi-head attention layers do not have a spatial inductive bias, as is the case with convolutional layers.

Hypotheses for Research Question 3:

- **H3.1: Advantages of Explicit Querying.** Explicit (non-random) querying is expected to offer advantages in terms of interpretability and the ability to query the most salient sensors for decision-making. This is expected to lead to improved performance across both test environments.
- **H3.2: Disadvantages of Hard Explicit Querying.** However, hard explicit querying over a large number of sensors may make the learning task more challenging and computationally intensive. Combined with the challenges in optimizing attention-based networks, this may negatively impact performance in unexpected ways, especially in complex tasks such as the vision-based environment.

1.5 Contributions of Research

This research is expected to make the following contributions to the field of reinforcement learning and control systems:

Empirical Evaluation of Sensor Failure Impact: This research will provide an empirical evaluation of the impact of sensor failure on the performance of conventional policy architectures in reinforcement learning tasks.

A Novel Attention-Based Policy Architecture: An attention-based policy architecture which represents a novel approach to dealing with sensor failure in reinforcement learning tasks[3]. We hope this will contribute to the development of more effective control strategies in environments characterised by asynchronous and potentially unreliable sensory inputs.

A Novel Hard Attention Mechanism: A novel hard attention mechanism used in conjunction with our attention-based agents for sampling sensory signals which we hope will contribute to the understanding of

[3]Note: we are aware of previously proposed attention-based policy architectures, many of which are examined in this dissertation. Ours is novel with respect to its overall architecture, not with respect to the inclusion of the attention mechanism.

the challenges regarding how such mechanisms might be effectively implemented in reinforcement learning tasks.

Development of a Novel Experimental Framework: We plan to open-source the experimental framework developed in this dissertation, along with the accompanying code used to simulate random sensor failure. This framework is versatile and can be adapted to various environments and we hope that this contribution will encourage further experimentation with different environments, beyond what we have been able to conduct in this dissertation due to resource limitations.

Foundation for Future Research: The proposed attention-based policy architecture and the hard attention mechanism represent promising avenues for further investigation and development. We hope this research will provide a basis for future studies to build upon, contributing to the advancement of the field.

1.6 Dissertation Layout

The remainder of this dissertation will take the following structure:

- **Chapter 2 – Background:** This chapter covers the foundational theories and research necessary to understand the remainder of the dissertation.
- **Chapter 3 – Literature Review:** This chapter reviews previous research related to partial observability and attention-based policy architectures in RL.
- **Chapter 4 – Methodology:** This chapter describes the methodology that underpins our experiments, including the specifics of our proposed attention-based architecture.
- **Chapter 5 – Experiments:** This chapter details the experiments carried out for this dissertation, including an overview of the test environments.
- **Chapter 6 – Results:** This chapter presents the results of our experiments.
- **Chapter 7 – Discussion:** This chapter discusses the implications of our experiments' results, as well as how our work fills gaps in and contributes to the existing body of knowledge in the area.
- **Chapter 8 – Conclusion:** This chapter concludes our research findings by providing answers to our research questions, detailing the limitations of our work, and suggesting possible directions for future research.

Chapter 2

Background

In the Background chapter of this dissertation, we lay the groundwork necessary for understanding the complexities of attention-based policy architectures within the domain of partially observable reinforcement learning. In section 2.1, we begin by establishing foundational reinforcement learning theory, encompassing scenarios of both full and partial observability, and introduce rudimentary, tabular learning algorithms. This foundational knowledge is necessary for grasping the more complex learning algorithms covered thereafter.

In section 2.2 we give an overview of artificial neural networks as function approximators, which is important for understanding deep reinforcement learning. Additionally, we provide a theoretical overview of Multilayer Perceptrons and Convolutional Neural Networks, which underpin the baseline policy architectures against which we measure our attention-based architecture in our experiments. Furthermore, we explore Recurrent Neural Networks, which we utilise to deal with the challenge of partial observability.

We then go on to review deep reinforcement learning algorithms that are fundamental to our research in section 2.3. We give an overview of the Deep Q-Network algorithm, a canonical reinforcement learning algorithm that provides a baseline for understanding subsequent methods discussed in the literature review section. We also cover policy gradient methods, with a particular focus on Proximal Policy Optimisation (PPO), the algorithm employed in our experiments. The contents of this background chapter will equip the reader with the necessary information to understand the remainder of the dissertation.

Finally, in section 2.4, we give a detailed overview of the attention mechanism of the Transformer architecture, which is central to our proposed policy architecture. Additionally, we discuss the adaptation of Transformers and transformer-like attention in the domain of computer vision through Vision Transformers and demonstrate how Masked Autoencoders demonstrate the proficiency of Vision Transformers — and by extension, transformer-like attention — in handling partial observability.

2.1 Reinforcement Learning Theory

2.1.1 The Reinforcement Learning Problem

In the canonical reinforcement learning problem [1] an agent interacts with an environment by making observations and taking actions at each point in a sequence of discrete time steps to accomplish a predetermined goal, or a set of goals, which constitute a task that is either finite in time or ongoing. The agent's success (or failure) in completing the task is quantified by a numerical reward received from the environment in response to taking a specific action in a specific environment state at each time step over the duration of the task. The agent's ultimate goal is to learn, through experience, a decision-making policy - a mapping from states to actions - under which the cumulative reward obtained from the environment over some (finite or infinite) time horizon is maximised. An illustration of the so-called 'agent-environment interface' [1] is shown in Figure 2.1.

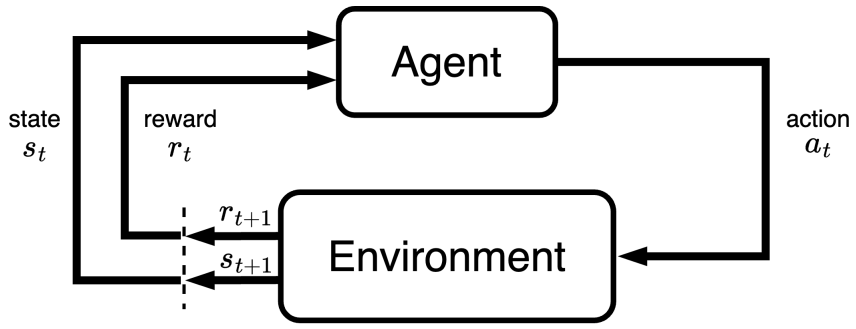


Figure 2.1.: The agent-environment interface [1]. At each time step, t, the agent receives an observation from the environment - this may be a complete representation of the environment state, notated as s_t, but it may also be an incomplete (partial) observation, notated as o_t, that simply depends on the true environment state - and performs an action based on its decision-making policy, receiving a numerical reward in return, and a representation of the updated environment state.

2.1.2 Reinforcement Learning as a Markov Decision Process

Under certain conditions, the reinforcement learning problem may be modelled as a *Markov Decision Process* (MDP) [29]; a mathematical framework for modelling decision-making problems comprising of a stochastic (or deterministic) process wherein the state transition probabilities are determined in part by the present state of the process and in part by actions selected by an external decision maker, or agent.

Formally, an MDP is a five-tuple $(\mathcal{S}, \mathcal{A}, R, P, \gamma)$. Here, $\mathcal{S}$ is the set of all possible environment states. $\mathcal{A}(s)$ is the set of all possible actions available in some state, $s \in \mathcal{S}$[1]. $R : S \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function, which produces a numerical reward given the present state, an action selected by the agent in that state, and the resultant state [2]. $P : S \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the state transition probability function, which gives a probability distribution over resultant states given a current state and action. Lastly, $\gamma \in [0, 1]$ is a scalar discount factor.

In an MDP, the agent interacts with the environment at each point in a sequence of discrete time steps $t = 0, 1, 2, \ldots$. At each time step, t, the agent observes $s_t \in \mathcal{S}$, a representation of the state of the environment, utilising its decision-making policy, $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$, which gives a distribution over possible actions given a state, to sample an action, $a_t \sim \pi(\cdot|s_t)$[3], in response. At the next time step, $t + 1$, the environment then transitions to a new state, $s_{t+1} \sim P(\cdot|s_t, a_t)$, sampled from the probability distribution over $\mathcal{S}$ as defined by the state transition probability function, and the agent receives a numerical reward, $r_{t+1} = R(s_t, a_t, s_{t+1})$. The objective of the agent is to learn a decision-making policy which optimises a quantity called the return; The expected sum of discounted rewards following each time step, t:

$$G_t = \mathbb{E}_{\tau \sim \pi, P} \left[\sum_{k=0}^{T} \gamma^k r_{t+k+1} \right]$$

where the expectation is taken over trajectories, τ, which are sequences of alternating states and actions beginning at some time step t and ending at some final time step, T. The probability of a given trajectory, τ, is determined by the agent's policy (the probability of action selection in each state) and the environment's state transition probability function, P.

Episodes of agent-environment interaction are defined as a sequence of states, actions, and rewards, beginning with some initial state (determined by the environment) and continuing in the manner described above. Depending on the environment in question, the set $\mathcal{S}$ may or may not contain one or several *terminal states*. A terminal state, s, is a state for which $P(s'|a, s) = 0$ for all $s' \in \mathcal{S}$ and for all $a \in \mathcal{A}$, which, if reached by

[1] For the sake of brevity, we will henceforth abbreviate this to $\mathcal{A}$.

[2] Although rewards are technically a function of the present state, selected action, and resultant state, actions must be selected based on the present state alone and so, as far as the agent is concerned, the reward function is a stochastic function of the form $R : S \times \mathcal{A} \to \mathbb{R}$

[3] Here we assume a stochastic policy, but in some cases, the policy may be deterministic, and denoted as $\pi : \mathcal{S} \to \mathcal{A}$.

the agent, constitutes the end of an episode. On the other hand, if $\mathcal{S}$ does not contain any terminal states for a given environment, episodes may continue indefinitely.

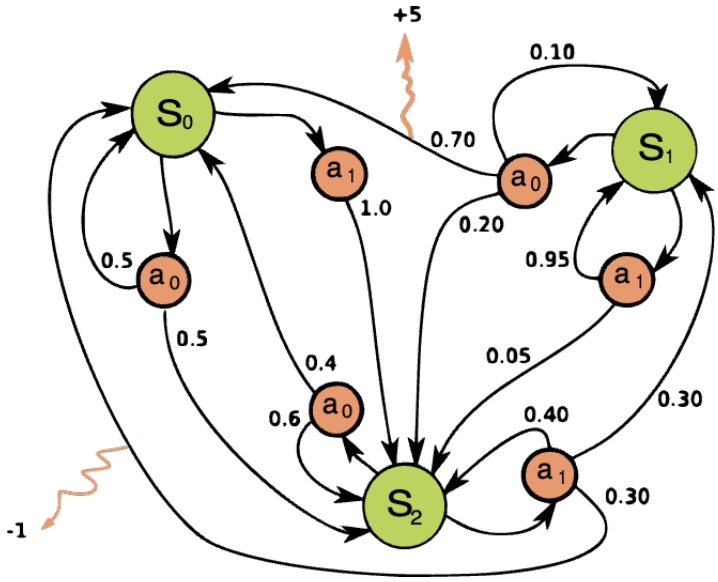


Figure 2.2.: An example of a simple stochastic MDP with three states (green circles), two actions (orange circles), and two non-zero rewards (orange arrows)[2]. State transition probabilities are indicated on the lines connecting actions and states.

2.1.3 The Markov Property

In order for an environment that constitutes a reinforcement learning task to be modelled as an MDP, the environment states must adhere to the Markov property:

$$p(s_t = s'|a_{t-1}, s_{t-1}) = p(s_t = s'|a_{t-1}, s_{t-1}, a_{t-2}, s_{t-2}, \ldots, a_0, s_0) \tag{2.1}$$

for all subsequent states $s' \in \mathcal{S}$ and for all historical trajectories of state-action pairs, $a_{t-1}, s_{t-1}, a_{t-2}, s_{t-2}, \ldots, a_0, s_0$. In words, this property states that the probability distribution over possible resultant states is fully determined given the current state and action, observed and chosen by the agent, respectively. What this property implies is that the current state captures in full the dynamics of the environment and that no additional information is gained by looking backwards at states and actions from preceding time steps - if this were the case, the implication would instead be that the 'state' observed by the agent was instead a partial observation, containing partial information, as opposed to a complete representation of the state of the environment ("complete" in the sense that the Markov property holds).

As an illustrative example, consider the simple 'catch' environment [3] [4] shown in Figure 2.3; a 10x5 grid with a single block falling at constant velocity (tiles per time step) in a given column and a paddle (a second block) on the bottom row of the grid. The agent can move the paddle left and write, with the objective being to 'catch' the ball when it reaches the bottom row. Now, suppose the observation emitted by the environment at some time step, t, consisted only of the x and y positions of the ball and paddle, respectively. In this case, the Markov property would not hold since the probability over possible resultant states at $t+1$ would change given, for example, the y coordinate of the ball at time $t-1$, which indicates the velocity at which the ball is falling. In order to satisfy the Markov property, the environment would need to emit a representation of the state which encodes the velocity at which the ball is falling at a given time step, in which case, observations from previous time steps would not change the probability distribution over possible resultant states.

[4]The 'catch' environment is part of a set of environments proposed in the paper *"Behaviour Suite for Reinforcement Learning"*. published by Google DeepMind in 2019 [3] designed to measure diverse aspects of an agent's problem-solving abilities.

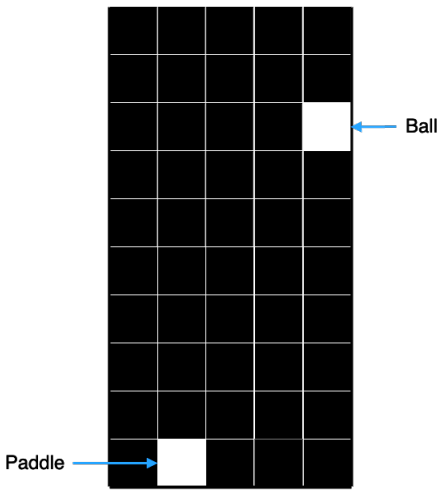


Figure 2.3.: The 'catch' environment [3]; a simple environment consisting of a 10x5 grid where during an episode, the agent must move a paddle left/right along the bottom row to 'catch' a block falling at a fixed velocity (tiles per time step) in a given column.

For a given reinforcement learning task, if the agent's decision-making policy takes the form $\pi : \mathcal{S} \to \mathcal{A}$, it is essential that the Markov property holds, since the policy depends solely on the current state of the environment at each time step, without any additional historical context. The assumption here is that the current state contains all relevant information to determine the transition probabilities to future states, and therefore to future returns, enabling the agent to learn a decision-making policy which can reliably learn to select the optimal action in any given state.

2.1.4 Value Functions

The return, G_t, obtained by the agent over a given trajectory - a sequence of alternating states and actions - following some time step, t, is a function of the actions selected by the agent over the course of the episode, determined by its policy, π, as well as the environment dynamics, determined by the transition probability function, P. So, return maximisation requires optimal action selection in any given state, meaning the agent needs a means to quantify the 'quality' of each possible action in any given state (and therefore the 'quality' of the state itself) with respect to its effect on the future rewards the agent might obtain under a fixed policy. This leads directly to the concept of *state* and *state-action* value functions, which we now formally define. The state-value function - how 'good' it is, on average for the agent to be in a given state - computes the expected return following a state, s, observed at some time, t, under some fixed policy, π:

$$
\begin{aligned}
v^{\pi}(s) &= \mathbb{E}_{\tau\sim\pi,P}\left[G_t | s_t = s\right] && (2.2)\\
&= \sum_a \pi(a|s) \sum_{s'} P(s'|a,s)\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots\right] && (2.3)\\
&= \sum_a \pi(a|s) \sum_{s'} P(s'|a,s)\left[r_t + \gamma\left[r_{t+1} + \gamma r_{t+2} + \dots\right]\right] && (2.4)\\
&= \sum_a \pi(a|s) \sum_{s'} P(s'|a,s)\left[R(s,a,s') + \gamma v^{\pi}(s')\right] && (2.5)
\end{aligned}
$$

The state-action value function - how 'good' it is on average for an agent to be in a given state and take a given action - computes the expected return following an action, a, taken in a state, s, at some time, t, under some fixed policy, π:

$$q^{\pi}(s,a) = \mathbb{E}_{\tau\sim\pi,P}\left[G_t|s_t=s,a_t=a\right] \tag{2.6}$$
$$= \sum_{s'} P(s'|a,s)\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots\right] \tag{2.7}$$
$$= \sum_{s'} P(s'|a,s)\left[r_t + \gamma\left[r_{t+1} + \gamma r_{t+2} + \dots\right]\right] \tag{2.8}$$
$$= \sum_{s'} P(s'|a,s)\left[R(s,a,s') + \gamma\sum_{a'}\pi(a'|s')q^{\pi}(s',a')\right] \tag{2.9}$$

It is important to emphasise here that, as can be seen in the definitions above, both the state value and state-action value functions are dependent on the decision-making policy, π, which makes sense since the action choices determine the trajectory of the agent through the space of all possible states and actions, $\mathcal{S}\times\mathcal{A}$. Additionally, note that equations 2.5 and 2.9 above express the relationship between the value of states/state-action pairs and their successor states/state-action pairs - these self-consistency properties, known as the Bellman equations for the state and state-action value functions respectively, are fundamental properties of value functions in RL and are widely used in RL algorithms, as we will demonstrate.

Since a given value function is defined only with respect to a given (fixed) policy, value functions in general define a partial ordering over policies. A policy, π, is defined to be better than or equal to a policy, π', if its expected return is greater than or equal to that of π' for all states (and hence for all state-action pairs). In other words, if and only if $v^{\pi}(s) \geq v^{\pi'}(s)$ for all $s\in\mathcal{S}$. There is always at least one policy that is better than or equal to all other policies - this is an *optimal policy*. Although there may be more than one, we denote all the optimal policies by π^*. By definition, all optimal policies share the same value functions, called the optimal value functions. The optimal state value function is denoted as v^* and defined as follows, $\forall s\in\mathcal{S}$:

$$v^*(s) = \max_{\pi} v^{\pi}(s) \tag{2.10}$$
$$= \max_{a\in\mathcal{A}} q^*(s,a) \tag{2.11}$$
$$= \max_{a\in\mathcal{A}} \mathbb{E}^{\pi^*}\left[G_t|s_t=s,a_t=a\right] \tag{2.12}$$
$$= \max_{a\in\mathcal{A}} \sum_{s'} P(s'|a,s)\left[R(s,a,s') + \gamma v^*(s')\right] \tag{2.13}$$

where q^* denotes the optimal state-action value function, which is defined as follows, $\forall s\in\mathcal{S}, \forall a\in\mathcal{A}$:

$$q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \tag{2.14}$$
$$= \mathbb{E}\left[r_{t+1} + \gamma v^*(s_{t+1})|s_t=s,a_t=a\right] \tag{2.15}$$
$$= \mathbb{E}\left[r_{t+1} + \gamma \max_{a'\in\mathcal{A}} q^*(s_{t+1},a')|s_t=s,a_t=a\right] \tag{2.16}$$
$$= \sum_{s'} P(s'|a,s)\left[R(s,a,s') + \gamma\max_{a'} q^*(s',a')\right] \tag{2.17}$$

Note that the value of a state under an optimal policy must equal the expected return for the best action from that state, whereas the value of a given state-action pair under an optimal policy is equal to the expectation of the sum of the reward obtained for the given state-action pair and the (discounted) return obtained thereafter under an optimal policy. Additionally, notice that if q^* is known, one obtains an optimal policy, π^*, by simply acting in a greedy manner with respect to q^*.

Related to the Bellman equations 2.5 and 2.9, equations 2.13 and 2.17 are the *Bellman optimality equations* for the state and state-action value functions, respectively. In a finite MDP - an MDP where $\mathcal{S}$ and $\mathcal{A}$ each contain a finite number of elements - the Bellman optimality equations for each state/state-action pair constitute a

system of non-linear equations; n equations with n unknowns corresponding to the values of states/state-action pairs. One can theoretically solve this system of equations for the optimal value function in question, provided the environment dynamics - meaning the transition probability and reward functions, P and R - are known. However, for many problems of interest, the dynamics of the environment are not known, which leads us directly to the idea of learning, via value function estimation, from experience.

2.1.5 Generalised Policy Iteration

If environment dynamics are unknown, it is impossible to compute the optimal value function(s) directly. The question is: how do we begin with a random policy and iteratively improve it in an attempt to converge on an optimal policy? *Generalised Policy Iteration* [1] (GPI), illustrated in Figure 2.4, provides a generalised framework (algorithm) for iteratively improving an agent decision-making policy via two simultaneous, interacting processes: *policy evaluation*, wherein the value function for a fixed policy is estimated, and *policy improvement*, wherein the agent's policy is updated to one which is greedy with respect to the newly estimated value function. While GPI may be employed by learning algorithms which rely on complete knowledge of the environment dynamics, for our purposes we are interested in how GPI may be applied when the environment dynamics are unknown.

Note: in what follows, we use v^π and q^π to denote the true state and state action value functions, respectively, under some policy, π, and we will use V^π and Q^π to denote the respective estimates of these functions (e.g. a lookup table with a value for each state or state-action pair) which may be maintained by an agent in an RL algorithm.

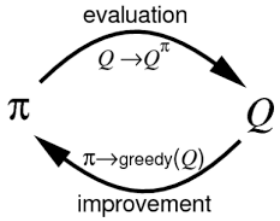


Figure 2.4.: Generalised Policy Iteration [1]. An agent collects experience to estimate the true state-action value function under its present policy - this is policy evaluation. Then, the agent improves its policy by acting in a greedy manner with respect to the newly estimated action-value function - this is policy improvement. The agent continues in this cycle, generating increasingly accurate approximations of the optimal action-value function, q^*. Note: while q^π denotes the true state-action value function under π, Q^π denotes the agent's approximation of q^π.

In this scenario, then, GPI would work in the following way. The agent begins with a random decision-making policy, π_0. Then, during *policy evaluation* the agent collects experience in the form of transition tuples (s, a, r, s') by interacting with the environment in order to compute an estimate, Q^{π_o}, of the true value function, q^{π_0}, under the policy, π_0. Then, during *policy improvement*, the agent updates its decision-making policy to π_1; one under which actions are selected in a greedy or semi-greedy manner with respect to Q^{π_0}. Under a greedy policy, the action with the highest estimated value in a given state is selected:

$$\pi(s) = \arg\max_{a'} Q(s, a')$$

However, in practice, a semi-greedy policy is employed in order to facilitate the exploration of possible actions in any given state, as the best possible action under a given policy may not necessarily be the one with the highest value estimate at any given time. The simplest of such policies is the so-called ϵ-greedy policy, where in a given state the agent randomly (uniformly) selects an action with probability $\epsilon \ll 1$, but acts according to a greedy policy otherwise. The degree to which the agent *explores* possible actions (in this case, the magnitude of ϵ) as opposed to *exploiting* its present estimates of state-action values is the so-called "exploration-exploitation" trade-off.

The policy improvement step then leads to an updated policy, which necessarily means that a second round of *policy evaluation* is required to estimate the new value function, Q_{π_1}, associated with the new policy. The hope, then, is that this cycle of evaluation and improvement produces a sequence of policies and estimated value functions which ultimately converge on an (approximately) optimal policy:

$$\pi_0 \overset{\text{eval}}{\rightarrow} Q^{\pi_0} \overset{\text{improve}}{\rightarrow} \pi_1 \overset{\text{eval}}{\rightarrow} Q^{\pi_1} \overset{\text{improve}}{\rightarrow} \dots \overset{\text{improve}}{\rightarrow} \pi^* \overset{\text{eval}}{\rightarrow} Q^*$$

The question then is: under which conditions, if any, can convergence to an optimal policy be guaranteed? The *policy improvement theorem*, given below, is central to GPI, as it provides the theoretical guarantee that the *policy improvement* step will yield a policy that is at least as good as, if not better than, the previous one. By applying the Policy Improvement Theorem, it is possible to prove that certain algorithms will converge to an optimal policy.

Theorem 1 (Policy Improvement Theorem)**.** Let π and π' be any pair of deterministic policies such that for all $s \in \mathcal{S}$:

$$q^{\pi}(s, \pi'(s)) \geq v^{\pi}(s)$$

Then the policy, π', must be as good, or better than, π. That is, it must obtain a greater or equal expected return from all states $s \in \mathcal{S}$:

$$v^{\pi'}(s) \geq v^{\pi}(s)$$

Moreover, if there is a strict inequality for a given state in the first expression, there must be a strict inequality for at least one state in the second expression.

The proof of the policy improvement theorem is straightforward - we simply expand the left-hand side of the first expression and keep going until we get $v^{\pi'}(s)$:

$$\begin{aligned}
v^{\pi}(s) &\leq q^{\pi}(s, \pi'(s)) \\
&= \mathbb{E}_{\pi'}\left[r_{t+1} + \gamma v^{\pi}(s_{t+1}) | s_t = s\right] \\
&\leq \mathbb{E}_{\pi'}\left[r_{t+1} + \gamma q^{\pi}(s_{t+1}, \pi'(s_{t+1})) | s_t = s\right] \\
&= \mathbb{E}_{\pi'}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 v^{\pi}(s_{t+2}) | s_t = s\right] \\
&\vdots \\
&\leq \mathbb{E}_{\pi'}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots | s_t = s\right] \\
&= v^{\pi'}(s)
\end{aligned}$$

We can show that the policy improvement theorem applies in the case of GPI in the following way. Suppose we have a deterministic policy, π, and that we have computed perfect estimates, V^{π} and Q^{π}, of its associated value functions. Then, we consider a new deterministic policy, π', which is greedy with respect to Q^{π}. Firstly, we can show that π' satisfies the conditions of the policy improvement theorem, since for all $s \in \mathcal{S}$ we have:

$$Q^{\pi}(s, \pi'(s)) = Q^{\pi}(s, \arg\max_a Q^{\pi}(s, a)) \geq V^{\pi}(s)$$

so we know that π' is at least as good, or better than π. Then, note that:

$$\begin{aligned}
\pi'(s) &= \arg\max_a Q^{\pi}(s, a) && (2.18) \\
&= \arg\max_a \mathbb{E}\left[r_{t+1} + \gamma V^{\pi}(s_{t+1})\right] && (2.19) \\
&= \arg\max_a \sum_{s'} P(s'|s, a)\left[R(s, a, s') + \gamma V^{\pi}(s')\right] && (2.20)
\end{aligned}$$

and suppose that the new greedy policy, π', is as good as, but no better than, the old policy, π. Then $V^{\pi} = V^{\pi'}$ and from 2.19 it follows that, for all $s \in \mathcal{S}$:

$$V^{\pi'}(s) = \max_a \mathbb{E}\left[r_{t+1} + \gamma V^{\pi'}(s_{t+1}) | s_t = s\right] \tag{2.21}$$

$$= \max_a \sum_{s'} P(s'|a, s)\left[R(s, a, s') + \gamma V^{\pi'}(s')\right] \tag{2.22}$$

which is equivalent to the Bellman optimality equation 2.13 and therefore $V^{\pi'}$ must be V^* and both π and π' must be optimal policies. Therefore, policy improvement must give us a strictly better policy except when the original policy is already optimal. Note that we have shown the above for deterministic policies, however, it is easy to extend the policy improvement theorem and the ideas conveyed above to the case of stochastic policies. Further details may be found in [1].

The critical assumption underlying the proof of convergence in GPI is that we can compute the state and state-action value functions exactly for any given policy [5]. However, when the environment dynamics are unknown, we must estimate the value functions from the experience collected through agent-environment interactions, which means we can typically only obtain an approximation of the value functions. This task is further complicated if the state space is continuous or very large, or if certain states are rarely visited by the agent. The specifics of how experience is collected and how value functions are approximated based on that experience vary among algorithms. Below, we introduce two families of algorithms—Monte Carlo Methods and Temporal Difference methods—which illustrate two different approaches to this challenge.

2.1.6 Monte Carlo Methods

In this section, we will briefly introduce a class of RL algorithms called Monte Carlo methods. This is especially important for our purposes as the learning algorithm used for our experiments falls into this class of learning algorithms.

Monte Carlo methods [6] are a class of RL algorithms in which an agent learns by gathering experience through interacting with an environment, sampling entire trajectories - sequences of states, actions, and rewards - in order to estimate the values of state and/or state-action pairs by averaging sample returns. To ensure that well-defined returns are available, we assume that the agent interacts with the environment over the course of episodes of finite length, that is, that the agent will always encounter a terminal state at some final time step, $t = T$, regardless of which actions it selects. A single episodic trajectory is denoted as:

$$s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T$$

This means that, at each time step, t, the agent observes s_t, selects a_t according to its policy and obtains a reward, r_{t+1}. Once a trajectory, or batch of trajectories, has been sampled we can compute sample returns for any given state or state-action pair in the sequence (the discounted sum of rewards following the time step in which the state/state-action pair is observed) in order to estimate its value. Monte Carlo methods rely on the fact that increasingly accurate value estimates can be computed the more experience is gathered with which to estimate them. The question of how much experience is required to estimate the values for state or state-action pairs with sufficient accuracy depends on the task (the size of the state space $\mathcal{S}$ and the environment dynamics) and the learning algorithm.

Monte Carlo methods follow GPI in order to improve the agent's decision-making policy. Policy evaluation in the context of Monte Carlo methods is the process of sampling trajectories under a fixed policy, π, in order to estimate the value function(s) associated with it. During policy improvement, the policy is updated to π' which is greedy or semi-greedy with respect to the state/state-action values estimated under π.

Regarding action selection and the exploration-exploitation trade-off in the context of Monte Carlo methods: if we do not make the assumption of "exploring starts" - where the initial state of each trajectory is

[5] Note that, while it is possible to prove theoretical convergence of certain RL algorithms using the policy improvement theorem and GPI, there are algorithms, such as Q-learning covered in section 2.1.7, for which theoretical convergence is provable without perfect value function estimation.

[6] The term 'Monte Carlo' outside of RL refers generally to methods which rely on generating estimates through random sampling.

sampled randomly from $\mathcal{S}$ - then we need to ensure that all actions have a non-zero probability of being selected in a given state in order to facilitate exploration - policies which have this property are called $\epsilon-$soft policies.

We now present an example Monte Carlo (MC) algorithm called *On-policy first-visit MC control* - we will refer to this algorithm as 'MC control' for brevity. 'On-policy' here refers to a class of RL algorithms which use the same policy they are learning to collect experience, as opposed to 'Off-policy' methods in which agents gather experience using a different policy to the one they are learning (i.e. iteratively updating). 'First-visit' methods are ones in which state-action value estimates are computed as the sample returns from the earliest occurrence of a given state-action pair in a given trajectory/episode, which is contrasted against 'every-visit' methods which average the returns computed from every occurrence of a given state-action pair in a given trajectory.

The MC control algorithm, as illustrated in **Algorithm** 1, performs policy optimisation according to GPI as follows. Consider an MDP $(\mathcal{S}, \mathcal{A}, R, P, \gamma)$, where $\mathcal{S}$ and $\mathcal{A}$ are sets of discrete elements (the continuous case will be addressed later). Initially, $\pi(a|s)$ is an arbitrary ϵ-soft policy, wherein all actions in a given state have a non-zero probability of being selected. Subsequently, $Q(s, a)$ is initialized as a lookup table of arbitrary real numbers, and $Returns(s, a)$ as a lookup table of empty lists, one for each state-action pair. Then, during each iteration of an indefinite loop, a trajectory of states, actions, and rewards is sampled. Using the sampled rewards, the sample returns for the first occurrence of (s, a) in each trajectory are computed and appended to the list of sample returns, $Returns(s, a)$. This list is then used to update the approximation of the state-action value in $Q(s, a)$—this constitutes the policy evaluation step. Policy improvement is subsequently performed by incrementally adjusting the action selection probabilities in $\pi(a|s)$ for each s encountered in the sample trajectory. This adjustment increases the probability of selecting the (estimated) optimal action in s, according to $Q(s, a)$, and decreases the probability of selecting all other actions in s.

Notice that in the MC Control algorithm, value estimates for each state-action pair (s, a) are retained in $Returns(s, a)$ even though the agent's policy is updated, meaning the value estimates in $Q(s, a)$ may be computed from trajectories sampled under several different policies. To understand why this works in practice, consider that the probability of a given trajectory of states and actions being generated under some ϵ-soft policy, π, always has a non-zero probability (provided it is possible under the dynamics of the MDP) which can be decomposed into a product of state-transition and action-selection probabilities as:

$$P(s_0, a_0, ..., s_{T-1}, a_{T-1}, s_T) = P(s_0) \prod_{t=0}^{T-1} \pi(a_t|s_t) P(s_{t+1}|s_t, a_t) > 0$$

where $P(s_0)$ is the probability of s_0 being the initial state. So while any given trajectory might be *unlikely* under a given policy, it is never *impossible*. Furthermore, as the policy converges as the cycles of policy evaluation and policy improvement continue, the distribution of the sample returns for each state-action pair in $Returns(s, a)$ will converge since the trajectories with the highest probability under the converging policy will increasingly be generated far more frequently than the low-probability trajectories sampled under previous iterations of the policy.

2.1.7 Temporal Difference Methods

In this section, we will briefly introduce a second major class of RL algorithms called Temporal Difference methods. This is useful for our purposes as much of the related research we will examine utilises algorithms which fall into this class of methods.

Temporal Difference (TD) methods represent a second major class of algorithms in RL, stated by Sutton and Barto in their canonical textbook *Reinforcement Learning: An Introduction* [1] as being an idea "central and novel to reinforcement learning". Like Monte Carlo methods, TD methods estimate value functions based only on experience, but unlike Monte Carlo methods, TD methods iteratively update value function estimates using other estimates – a technique known as *bootstrapping* in the statistical literature. To understand precisely how the value estimates are computed in this way, suppose that we are aiming to learn a state-value function, $V(s)$,

Algorithm 1 On-policy first-visit MC control

Require: $\epsilon \ll 1$
  $Q(s,a) \leftarrow \mathbb{R}$(arbitrary values) $\forall s \in \mathcal{S}$, $\forall a \in \mathcal{A}$
  $Returns(s,a) \leftarrow$ empty list $\forall s \in \mathcal{S}$, $\forall a \in \mathcal{A}$
  while *true* **do** ▷ (continuing indefinitely)
    generate an episode following π: $s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T$
    $G \leftarrow 0$
    for $t = T-1, T-2, \ldots, 0$ **do**
      $G \leftarrow r_{t+1} + \gamma G$
      if (s_t, a_t) not in $s_0, a_0, r_1, \ldots, s_{t-1}, a_{t-1}$ **then**
        append G to $Returns(s_t, a_t)$
        $Q(s,a) \leftarrow$ average$(Returns(s_t, a_t))$
        $a^* \leftarrow \arg\max_a Q(s_t, a)$
        for $s_t \in$ States **do**
          $$\pi(a|s_t) \leftarrow \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s_t)|} & \text{if } a = a^* \\ \frac{\epsilon}{|\mathcal{A}(s_t)|} & \text{otherwise} \end{cases}$$
        end for
      end if
    end for
  end while

under some policy in the context of a Monte Carlo method. After collecting a full trajectory of experience, we can then compute the value estimate, G_t, for a given state, s_t, then, instead of averaging over all previous value estimates for s_t we might adjust our current estimate, $V(s_t)$, resulting in an updated $V'(s_t)$ by:

$$V'(s_t) \leftarrow V(s_t) + \alpha \left[G_t - V(s_t) \right]$$

where α is a constant hyperparameter called the 'learning rate', with a value typically ≤ 1 in the case of TD algorithms [1]. This will result in an increase in our value estimate for s_t if G_t is larger than $V(s_t)$ and a decrease (or no change, if precisely equal) otherwise. Notice then that instead of using the target, G_t, as is the case in the Monte-Carlo update in Algorithm 1, we can create an update target which approximates the true value of s_t using the reward, r_{t+1}, obtained after just one-time step:

$$V'(s_t) \leftarrow V(s_t) + \alpha \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right]$$

This bootstrapped target is the one-step temporal difference denoted $TD(0)$ which is a special case of the n-step $TD(\lambda)$ methods which compute bootstrapped approximations using rewards obtained for n-steps following the target time step. The critical idea here is that we are able to incrementally improve our value function estimate of a state or state-action pair based on experience from a single environment transition.

We now present a well-known TD method called Q-Learning, one of the early breakthroughs in reinforcement learning [1] proposed by Christopher Watkins in 1989 [30]. Q-learning is an off-policy TD algorithm defined by the 1-step TD update:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \right] \tag{2.23}$$

An important aspect to notice in this expression is that the expression in brackets takes the form of the Bellman Optimality equation 2.17, meaning that it results in an approximation of q^*. The form of this update makes it simple to prove convergence, for which the only requirement in the case of Q-learning is the minimal requirement under GPI for any algorithm which learns via policy improvement; that all state-action pairs continue to be updated [1].

Q-learning works in the following way. As with MC control, we consider an MDP $(\mathcal{S}, \mathcal{A}, R, P)$ where $\mathcal{S}$ and $\mathcal{A}$ discrete sets. We set the hyperparameters of the algorithm: $\epsilon \ll 1$, for stochastic action selection, and the

learning rate, $\alpha \in (0, 1]$, for performing the 1-step TD updates on Q. We then initialise $Q(s, a)$ to be a lookup table of arbitrary real numbers[7]. Then, as is required for convergence, we continue to simulate episodes indefinitely in order to collect experience and iteratively update our estimate of q^*. Note that whereas with MC control we required that each episode terminates after a finite number of time steps, we do not require that for Q-learning, provided that the MDP is ergodic, that is, that all states are reachable from all other states given a non-zero probability of selecting each possible action in all states - this ensures that all state-action pairs will continue to be updated indefinitely, as required. For each time step, t, of each episode, s_t is observed and a_t is sampled from an ϵ-soft policy derived from $Q(s, a)$. Finally, we perform the 1-step TD update to $Q(s, a)$, which is repeated for each time step, as opposed to MC control which performs an update once per sampled episodic trajectory.

Recall that Q-learning is an off-policy method; while we are aiming to learn an optimal policy via the approximation of $q*$, the policy used during training to select actions need not be an ϵ-soft policy, as the only condition we require for convergence is, as far as the agent's policy is concerned, that each action in any given state has a non-zero probability of being selected. This means that the policy could be any policy which satisfies this condition, including a policy which selects actions in a uniformly random way in any given state, although this might result in tiny probabilities for reaching large subsets of the state space, and might therefore make training (to a sufficiently accurate approximation of q^*) longer than is practical.

Algorithm 2 Q-Learning

Require: $\epsilon \ll 1$
Require: $\alpha \in (0, 1]$
$Q(s, a) \xleftarrow{R} \mathbb{R}\ \forall s \in \mathcal{S}$, $\forall a \in \mathcal{A}$ except that $Q(s_{terminal}, \cdot) \leftarrow 0$ for all terminal states in $\mathcal{S}$
while true **do** ▷ (continuing indefinitely)
  for each time step, t, in the episode **do**
    Select action $a_t \sim \pi(\cdot|s_t)$ where π is a policy derived from Q (e.g. ϵ-soft)
    Take action a_t and observe r_{t+1}, s_{t+1}
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma \max_q Q(s_{t+1}, a) - Q(s_t, a_t)\right]$
  end for
end while

It is worth noting here that, while Q-learning does not strictly conform to the GPI framework, it is in fact possible to prove that the Q-learning algorithm will converge on the optimal state-action value function based on its own theoretical foundations [31]. If the assumptions that (a) the environment MDP is finite (meaning that there are finitely many states and actions), and (b) that all state-action pairs are visited infinitely often, are satisfied, convergence is theoretically guaranteed in the limit. A detailed proof may be found in [31].

2.2 Artificial Neural Networks For Function Approximation

Up until now, we have only explicitly considered the case where the observations emitted by the environment are discrete elements, allowing us to represent $Q(s, a)$ in MC control and Q-learning, for example, as a lookup table with a value for each state-action pair, of which there are finitely many. It's worth noting that we have also made the assumption so far that the number of all discrete observation-action pairs in a given environment is small enough such that it is practical to store them in computer memory as a lookup table - the methods described in this chapter may also be applied in cases where the size of the joint observation-action space is so large that it becomes impractical to use tabular learning algorithms.

We now consider the case where the observations emitted by the environment are continuous[8]. One example of such a state space would be $\mathcal{S} \subset \mathbb{R}^N$, meaning each observation (in the case of an MDP) is an

[7] Note: the notation $\xleftarrow{R} \mathbb{R}$ means "sample a random element from the set of all real numbers".

[8] Note: Although these observations are theoretically continuous, it is important to acknowledge that their representation in computer memory necessitates a finite set of values. Consequently, our implementation treats them as continuous for practical purposes. However, for the sake of rigour, it is important to recognise that what we refer to as a 'continuous' observation space is, in reality, a discrete space that approximates the continuous ideal.

$N-$dimensional real-valued vector where each component might represent a quantity like position along some coordinate axis, or velocity, and where the agent seeks to learn a policy, π, with the objective of optimal control of some physical system, based on some well-defined reward function, which maps real-valued vectors of size N to a probability distribution over the action space $\mathcal{A}$:

$$\pi : \mathbb{R}^N \times \mathcal{A} \rightarrow [0, 1]$$

A second example might be the case where $\mathcal{S} \subset [0, 255]^{H \times W \times 3}$, the space of all possible images of height H, width W, and 3 colour channels, and where each pixel in each channel is allowed a value between 0 and 255 - an example of such an environment might be a video game simulator where the agent takes on the role of the player, seeking to learn a policy, π, which maximises the game score (i.e. the reward), and which, as above maps to a probability distribution over actions:

$$\pi : [0, 255]^{H \times W \times 3} \times \mathcal{A} \rightarrow [0, 1]$$

In practice, while the set of all possible observations arising from a theoretically continuous state space is not truly infinite due to the constraints of computer memory, the number of potential observations that could be stored in memory for approximation purposes is still exceedingly vast. This makes it impractical to represent observation-action pairs in a tabular form. Consequently, we turn to Artificial Neural Networks, which offer a viable solution for handling such extensive state spaces. We will now elaborate on the reasons why neural networks are particularly well-suited for this task.

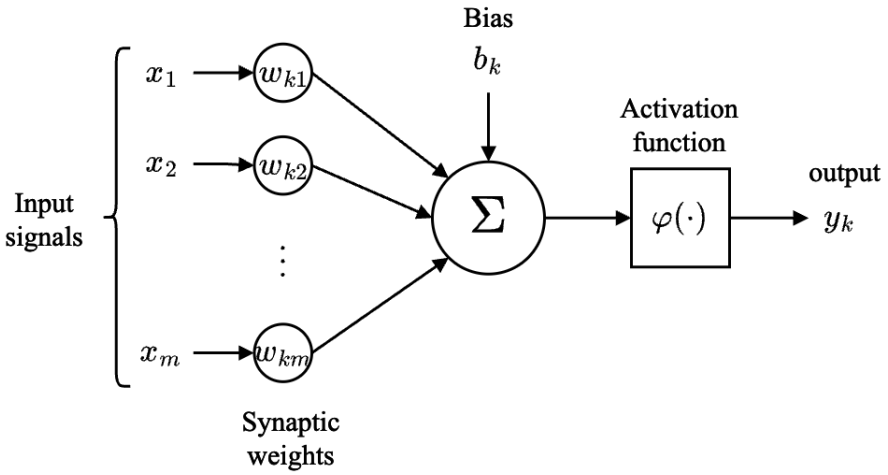


Figure 2.5.: The non-linear model of a neuron [4]; the output activation, y_k, of the k^{th} neuron in a layer is produced by performing a weighted sum of the input vector, $\mathbf{x}$, followed by the addition of a bias term, with the result being passed through a non-linear activation function.

Artificial Neural Networks (ANN) are a broad class of models consisting of hierarchical layers of artificial neurons which transform input tensors (multi-dimensional arrays) of real numbers to some output (tensor or scalar) representation via a sequence of intermediate transformations between layers. Each transformation typically consists of (at least) a matrix multiplication with a matrix of parameters, which are called the weights of the network and which represent the strength of the connections between neurons in adjacent layers [9], followed by a non-linear activation function transformation. The general idea underlying ANNs is that the weighted connections between neurons can be tuned in order to optimise some objective function, perhaps representing a task such as classification or regression, of the network output.

ANNs are inspired by networks of biological neurons in the brain connected by synapses which mediate the interactions between neurons in order to process incoming stimuli (i.e. via the body's nervous system) transforming them into some useful output representation facilitating the functioning of the body and mind [4]. Figure 2.5 illustrates the basic "nonlinear model of a [single] neuron"[4]; each component x_i of the input

[9] It is not strictly the case that connections need only exist between neurons in adjacent layers; skip/residual connections in the ResNet architecture [32] are an example of this.

vector, representing the stimulus which might be external or the output of a preceding layer, is multiplied by its corresponding weight w_{ki} - the i^{th} component of the weight vector belonging to the k^{th} neuron - which models the strength of the synaptic connection between the i^{th} input and the neuron. A bias scalar is added to the output, and the result is passed through some non-linear activation function φ to produce the output, y_k.

Activation functions were designed to scale the value of the weighted sum computed from the stimuli and the neuron's weights and bias value into some permissible, finite range. Two of the earliest examples of activation functions are shown in Figure 2.6. On the left, we have the Heaviside step function, a threshold function which maps all positive values to 1 and all negative values to 0:

$$\varphi(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.24}$$

On the right, we have the sigmoid activation function which, instead of having a step change, has a smooth slope which increases gradually from 0 to 1 about $x = 0$:

$$\varphi(x) = \frac{1}{1 + e^{-x}}$$

Note that, while the Heaviside step function is not differentiable at $x = 0$, the sigmoid function is, which is a requirement for gradient-based optimisation methods which we will see shortly. At the time of writing, the study of activation functions in the context of ANNs is a large and active area of research and there are many different types of activation functions beyond the two mentioned here.

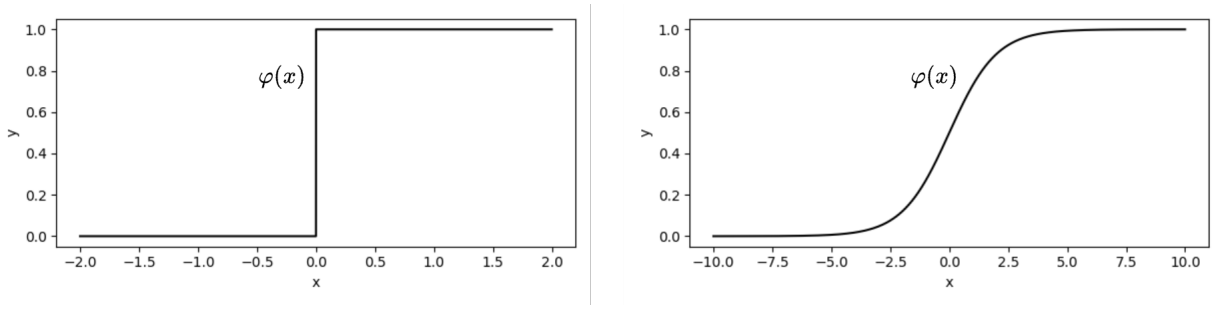


Figure 2.6.: Examples of activation functions [4] **Left**: the Heaviside step function **Right**: the sigmoid activation function

In 1943 Walter Pitts and Warren McCulloch proposed the first artificial neuron model in *A logical calculus of the ideas immanent in nervous activity* [33] and were the first researchers to describe what would later be referred to as an (artificial) neural network. In 1958, Frank Rosenblatt [34] introduced the Perceptron algorithm; a single-layer neural network capable of performing binary classification which is equivalent to the neuron model shown in 2.5 where the activation function is the Heaviside step function:

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \cdot \mathbf{w} + b > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.25}$$

The functional representation of a single neuron shown here in 2.25 is a vector-to-scalar mapping $f : \mathbb{R}^M \to \mathbb{R}$. This can be extended by combining several neurons to form a simple *fully-collected* layer, also called a *dense* or *linear* layer; a mapping $f : \mathbb{R}^M \to \mathbb{R}^N$, constructed by stacking together the weight vectors from each neuron into a weight matrix $\mathbf{W}$ and the bias scalars into a vector $\mathbf{b}$. A dense layer followed by an element-wise activation function, φ, may be notated as:

$$f(\mathbf{x}) = \varphi\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right) \tag{2.26}$$

where φ is applied element-wise to the resulting vector. The layer is referred to as 'fully connected' because all inputs are connected (via weighted connections) to all output neurons - note that it is possible to mute the

connection between input i and output j by fixing the weight $w_{ij} = 0$, meaning that the i^{th} input will not contribute to the weighted sum of output j. The term 'linear' refers to the fact that each neuron computes a linear combination (with a bias) of the input values.

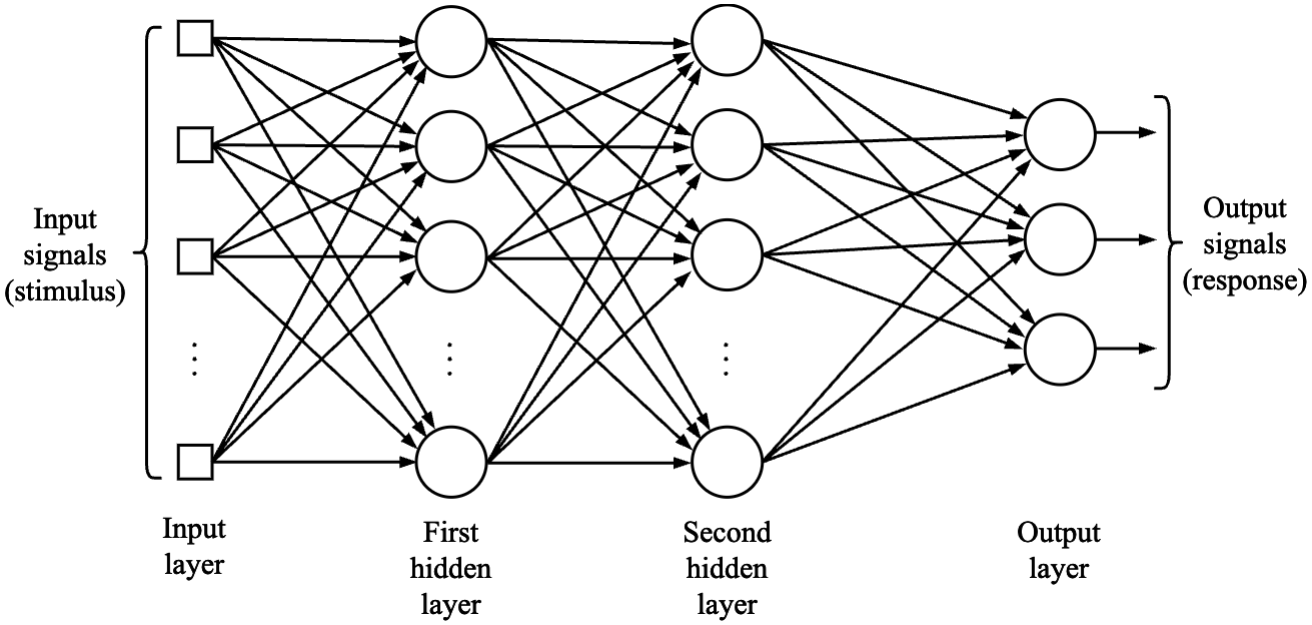


Figure 2.7.: A multi-layered perceptron made up of three fully connected (dense) layers; two hidden layers and an output layer [4]

In 1962, Rosenblatt went on to propose a "multi-layered perceptron" (MLP); a network with 3 layers of neurons; an input layer, a hidden layer, and an output layer, however, it differed from modern MLPs in that the weights belonging to the hidden layer were random and fixed [35]. Modern MLP networks, as illustrated in Figure 2.7 [4], consist of a sequence of fully connected layers which feed into one another, with the output vector from each preceding layer becoming the input for the subsequent layer. It is important to observe that each output neuron in the MLPs final layer is a function of all the parameters - the weights and biases - of the network, save those in the final layer which belong to the other output neurons - the hierarchical nature of the network is such that each subsequent layer computes features (weighted sums of inputs followed by an activation function) of the features computed in the previous layer.

While much research of multi-layered networks such as MLPs was conducted throughout the 1960s and 1970s, a major inflection point came when Paul Webros, in his 1974 PhD thesis, first described in full the *back-propagation* algorithm for optimising multi-layered ANNs; an efficient optimisation method which utilises the Leibniz Chain Rule to efficiently compute the partial derivatives of a so-called 'loss function', which defines the learning objective with respect to each parameter in the network by recording the neural activation values during a forward pass through the network and then propagating the errors backwards through the layers [36]. David Rumelhart, along with Geoffery Hinton and Ronald Williams, would go on to further popularise back-propagation when they published their *Learning representations by back-propagating errors* in which they showed that interesting, meaningful intermediate representations of the input could be learned in the hidden layers of a multi-layered ANN trained with gradient descent via backpropagation [37].

We now describe the optimisation problem for ANNs. Consider a standard regression problem where we have a set of inputs, $X \subset \mathbb{R}^N$, a set of targets, $Y \subset \mathbb{R}$, and an unknown joint probability distribution, $P(X, Y)$, which determines the probability of a given observation, (x, y), occurring as $P(X = x, Y = y)$. Then, suppose we have a sample of observations, (x_i, y_i), for $i = 1 \dots M$ drawn from $P(X, Y)$, where $x_i \in X$ and $y_i \in Y$ are the input vector and target value of the i^{th} sample, respectively. Given our sample observations, we would like to model (approximate) the relationship between X and Y using a parameterised ANN $f_{\boldsymbol{\theta}} : X \to Y$, where $\boldsymbol{\theta}$ represents the vector of all the network parameters. To this end, we seek to minimise the average value of the prediction error over all samples, (x_i, y_i), for $i = 1 \dots M$. The average prediction error - the mean of the magnitude of the difference between $f_{\boldsymbol{\theta}}(x)$ and y over all samples, (x, y)

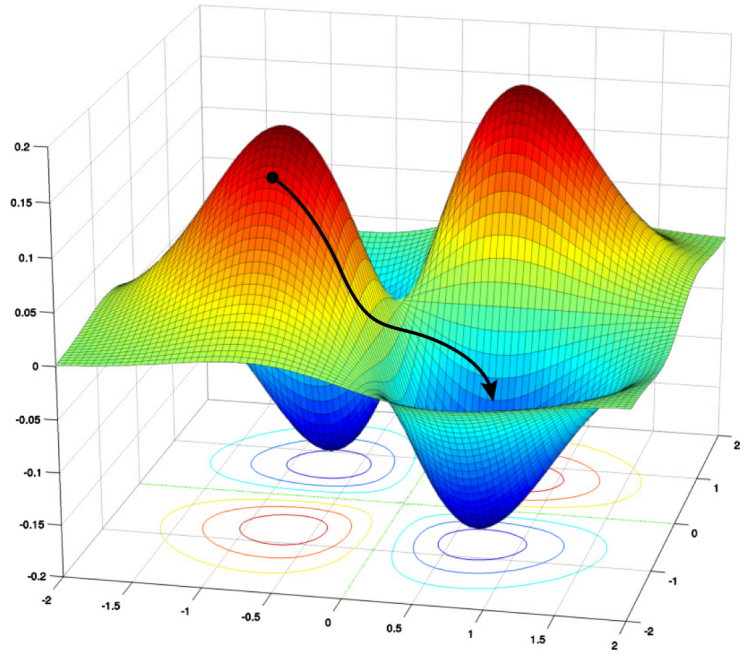


Figure 2.8.: An illustration of a loss surface over two parameters [5]

- is calculated using a so-called loss (or 'cost') function. A simple example of such a loss function is the mean-squared error (MSE):

$$MSE(\boldsymbol{\theta}) = \frac{1}{M}\sum_{i=1}^{M}(f_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i)^2 \tag{2.27}$$

What we have then is an optimisation problem wherein we seek to tune the network parameters $\boldsymbol{\theta}$ to those values which minimise the expected value of a given loss function with respect to the joint probability distribution over all inputs and outputs, $P(X, Y)$. As shown in [38], the full supervised learning problem may be formally expressed as:

$$\min_{\theta \in \boldsymbol{\theta}} J(\boldsymbol{\theta}) \text{ where, } J(\boldsymbol{\theta}) := \int_{X \times Y} l(f_{\boldsymbol{\theta}}(x), y) dP(x, y) = \mathbb{E}\left[l(f_{\boldsymbol{\theta}}(x), y)\right] \tag{2.28}$$

where l computes a real-valued prediction error for a single example, for example, the quantity $(f_{\boldsymbol{\theta}}(\mathbf{x}) - y)^2$ in the MSE loss function 2.27. Notice that the differential, $dP(x, y)$, means that the contribution of the loss for given values of x and y to the integral in 2.28 is scaled by the probability of the observation, meaning that an optimal solution to 2.28 weights the contributions over all pairs (x, y) according to their joint probability. The implication here is that, as in the example of the MSE, since the loss function approximates the expectation of the prediction error over the joint distribution by averaging over all sample observations, the quality of the approximation depends on the sample observations having a distribution which is representative of the population.

How then should we go about optimising an ANN in practice? Notice that given a fixed set of sample observations, a smooth loss function constitutes a so-called loss surface over all possible parameter values. Optimising our network then requires navigating along the loss surface to find the lowest possible minima. An illustration of a loss surface over two parameters is shown in Figure 2.8 - the indicated path along the surface illustrates how one might begin with random parameter values - a point on the surface at which the loss is high (sub-optimal) - moving downwards along the surface to a minima - this is the idea behind gradient descent, which we now describe.

Again, consider the standard regression problem described above and the optimisation objective captured by the MSE loss function 2.27. Supposing that we begin by assigning random values to each of the network parameters; we would then like to iteratively optimise the parameters of our network $f_{\boldsymbol{\theta}}$ with respect to

the MSE loss function by making incremental updates to our parameter values in order to move 'downhill' along the loss surface. To perform a single optimisation step using gradient descent, we first compute the gradient vector of the MSE with respect to $\boldsymbol{\theta}$, which is equivalent to averaging the gradient vectors across all N observations in the sample:

$$\nabla_\theta MSE(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \frac{1}{M} \sum_{i=1}^{M} (f_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i)^2 \tag{2.29}$$

$$= \frac{1}{M} \sum_{i=1}^{M} \nabla_{\boldsymbol{\theta}} (f_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i)^2 \tag{2.30}$$

$$= \frac{2}{M} \sum_{i=1}^{M} (f_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i) \nabla_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(\mathbf{x}_i) \tag{2.31}$$

$$= \frac{2}{M} \sum_{i=1}^{M} (f_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i) \left[\frac{\partial f_{\boldsymbol{\theta}}(\mathbf{x}_i)}{\partial \theta_0}, \frac{\partial f_{\boldsymbol{\theta}}(\mathbf{x}_i)}{\partial \theta_1}, \ldots, \frac{\partial f_{\boldsymbol{\theta}}(\mathbf{x}_i)}{\partial \theta_K} \right] \tag{2.32}$$

Note that the output layer of our MLP, $f_{\boldsymbol{\theta}} : \mathbb{R}^N \to \mathbb{R}$, contains a single neuron, and so we can express the gradient vector in 2.32 as a vector of partial derivatives of the network output with respect to each parameter - however, it is straightforward to generalise to the case of a network with more than one neuron in the final layer, given that the loss function combines the output values of all neurons in the final layer to compute a real-valued prediction error. In practice, the gradient vector denoted in 2.32 may be computed using the backpropagation algorithm [36], wherein each computation is performed in two steps; the forward pass and the backward pass. During the forward pass, each input is passed through the network from layer to layer to produce an output, during which all the values output by the activation functions of each neuron in each layer of the network are recorded, and finally, the value of the loss function - the mean prediction error over all observations - is computed. During the backwards pass, the error computed by the loss function is propagated backwards through the computational graph constituted by the network via the Leibniz Chain Rule, using the activation values recorded during the forward pass to efficiently compute the partial derivatives of the loss function with respect to each network parameter [10]. Once we have computed the mean gradient vector over all our observations, we can perform a single optimisation step as follows:

$$\boldsymbol{\theta}' \leftarrow (1 - \alpha)\boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} MSE(\boldsymbol{\theta}) \tag{2.33}$$

where $\alpha \ll 1$ is the learning rate. This small update to each parameter in the network has the effect of taking a small step along the loss surface in the direction of steepest descent, by definition, and therefore decreasing the mean value of the prediction error over the batch of sample observations (given that α is small enough – indeed, it is possible to take too large a step and end up going uphill!).

In the field of deep learning, 'optimisers' are a category of algorithms used for applying gradient updates to ANNs in the context of gradient descent. Optimisers constitute a large and ongoing body of research and as such there are hundreds of options, with the two most frequently mentioned in the literature being 'Stochastic Gradient Descent' (SGD) and 'Adam' [39]. SGD has its origins in stochastic optimisation research done in the 1950s [40] and grew to be the de facto optimisation method for training ANNs by 2012 [41]; as opposed to computing a gradient vector averaged over a batch of observations, SGD involves computing and applying a gradient vector from each example in the batch independently, making for a sequence of updates which are far more stochastic [11] than batch gradient descent - a feature which can, in fact, help, not harm, the optimisation process as it makes it less likely that the optimiser will get stuck in a sub-optimal minima [41]. Figure 2.9 [6] illustrates the effect of stochastic vs batch gradient descent on parameter updates over the course of optimisation; as is evident, both algorithms converge approximately on the same region in parameter space, but the SGD updates are far more noisy.

[10] A full explanation of the backpropagation algorithm is beyond the scope of this dissertation, but a brilliant explanation may be found here: `http://neuralnetworksanddeeplearning.com/chap2.html`.

[11] As expected.

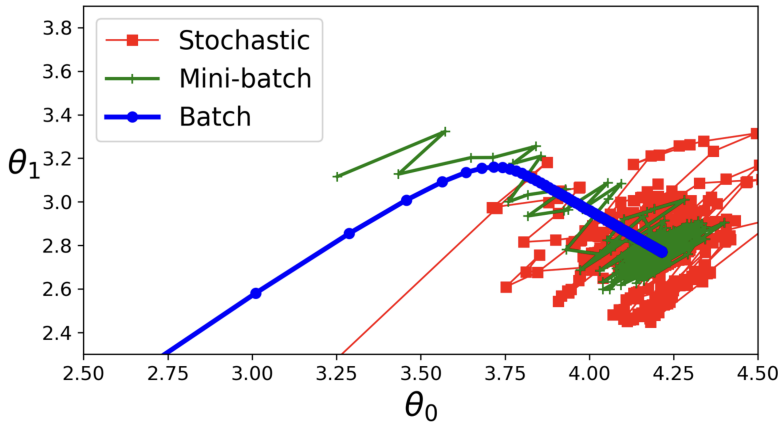


Figure 2.9.: An illustration of paths through parameter space during optimisation arising from batch, mini-batch, and stochastic gradient descent optimisation methods [6]. Batch gradient descent results in more stable updates which converge on the optimal parameter values slowly, whereas stochastic gradient descent results in noisy updates which bounce around the optimal parameter values. In high-dimensional parameter spaces stochastic updates can offer an advantage by preventing the optimiser from getting stuck in local minima.

The Adam algorithm [42], proposed in 2014 and shown to be clearly superior to the state-of-the-art optimisers at the time (see Figure 2.9), is part of a class of optimisers which, instead of computing updates from the gradient vector arising from a single input or batch of inputs in an isolated fashion, uses an exponential moving average of gradients computed over time, meaning that the gradient updates maintain a kind of momentum and are only partially determined by the most recently computed gradient. The Adam algorithm maintains exponentially moving averages of the gradient (the first moment, or mean) and the squared gradient (the second moment, or un-centred variance), computing the gradient update as the ratio between the first vector (mean) and the square root of the second vector (variance), respectively, which has the effect of normalising the values in the update vector and stabilising learning. Despite years of research efforts to make progress, Adam remains widely used and is competitive, with no clearly superior algorithm having been proposed since its invention [41].

At this point, it is worth asking if there are any mathematical guarantees regarding which types of functions multi-layered networks like MLPs are able to approximate, and to what degree. To this end, several universal approximation theorems (see [43]) have been proposed proving that an MLP with a single hidden layer can approximate arbitrarily well any continuous function of n variables, given a sufficient number of neurons in the hidden layer. Early results proved by Kurt Hornik [44] and George Cybenko [45] at a similar time required that the activation function be of a certain form (e.g. a Sigmoid function), however, later proofs [46] [47] were published which relaxed these requirements showing, for example, that having an activation function which is piece-wise continuous and locally bounded is sufficient to prove the result. Below, we give a derivative of the theorem proved in [46].

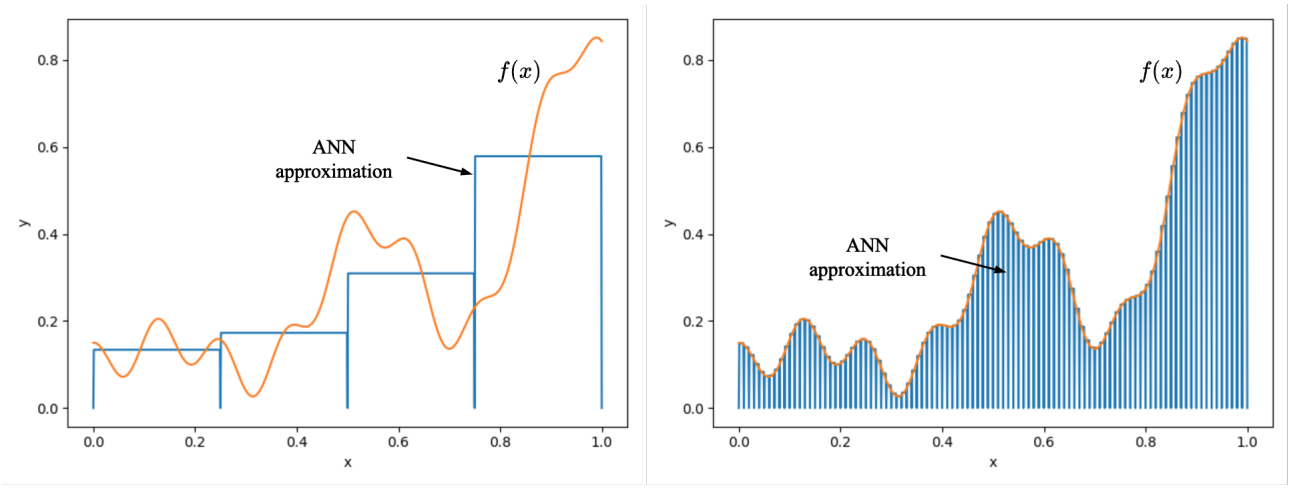
Theorem 2 (Universal Approximation Theorem for MLPs)**.** Let $C(X, \mathbb{R})$ denote the set of continuous functions from a subset, X, of Euclidean space, $\mathbb{R}^n$, to a Euclidean space, $\mathbb{R}^m$. Let $\varphi \in C(\mathbb{R}, \mathbb{R})$ be a continuous, locally bounded activation function.

Then φ is a non-polynomial function $\iff$ for every $n \in \mathbb{N}$, $m \in \mathbb{N}$, compact set $K \in \mathbb{R}^n$, $f \in C(K, \mathbb{R}^m)$, $\epsilon > 0$ there exists $k \in \mathbb{N}$, $W_1 \in \mathbb{R}^{k \times n}$, $b \in \mathbb{R}^k$, $W_2 \in \mathbb{R}^{m \times k}$ such that

$$\sup_{x \in K} \|f(x) - g(x)\| < \epsilon$$

where $g(x) = W_2 \cdot (\varphi(W_1 \cdot x + b))$, and where φ is applied element-wise to each element in the vector.

In words, Theorem 2 says, given a few minor conditions are satisfied, that a continuous function from one Euclidean space of arbitrary dimension to another can be approximated arbitrarily well (within ϵ) provided the number of neurons in the hidden layer is sufficiently large. Without seeing formal proof this is difficult



(a) 4 hidden neurons (b) 100 hidden neurons

Figure 2.10.: An illustration of a 2-layered MLP (an ANN) approximating a non-linear function with 4 (left) and 100 (right) neurons in its hidden layer [7]. In both diagrams the orange curve represents the non-linear function, and the blue rectangles represent the ANN approximation of the function.

to believe, and even harder to intuit - Figure 2.10 [7] illustrates how increasing the number of neurons in the hidden layer of a 2-layer MLP can improve the fidelity of the approximation.

While the universal approximation theorem shown above only applies to MLPs, equivalent results have been proven for other types of ANNs, namely; convolutional neural networks [48] and recurrent neural networks [49]. We briefly describe both network architectures in the following sections.

2.2.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a class of ANNs specifically designed for processing grid-like data, such as images and videos, which can learn hierarchical spatial features directly from raw pixel values. A typical CNN architecture may be viewed as having two sub-networks, as illustrated in Figure 2.11[8]: a 'feature learning' network composed of interleaving convolutional and pooling (aggregation) layers which have the role of extracting spatial features hierarchically, and a 'predictor' network which is essentially an MLP with multiple dense layers which transform the extracted feature information into a scalar or vector output, the exact form of which is task dependent.

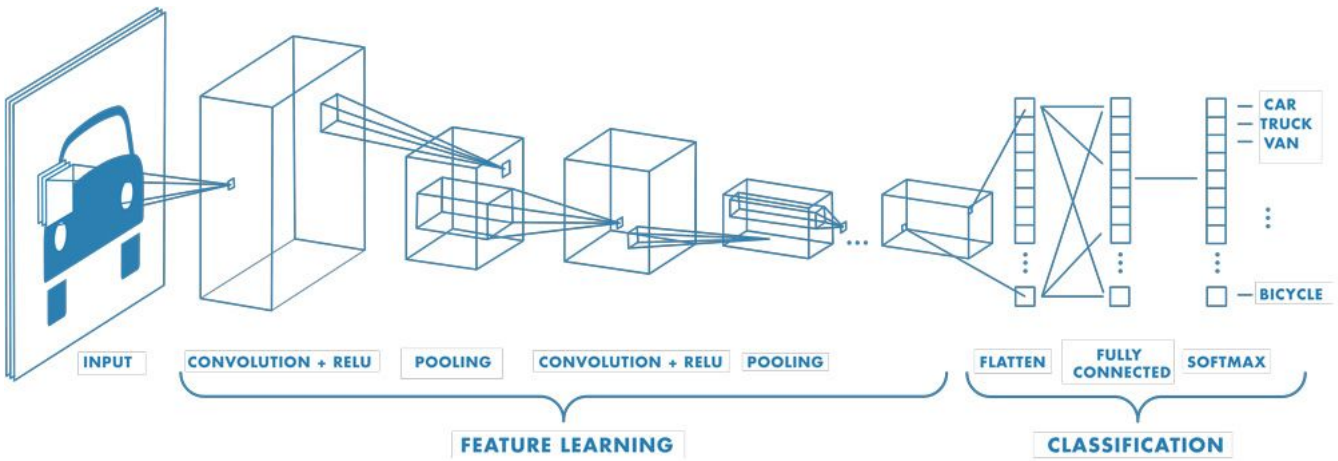


Figure 2.11.: An example of a convolutional neural network [8]. The feature learning section of the network, made up of interleaving convolution and pooling layers, learns to extract visual features from pixels in a hierarchical fashion, while the classification section of the network, made up of dense layers, learns to classify the object in a given image using the extracted visual features.

As with MLPs, the development of CNNs is not attributable to a single researcher or paper, but a few major contributions were made to their development. In 1980, Kunihiko Fukushima, a Japanese scientist, proposed a neural network architecture called Neocognitron [50]. While not a convolutional neural network (CNN) in the modern sense, Neocognitron laid the groundwork for the development of convolutional networks. The Neocognitron was designed to recognize visual patterns in an image and was inspired by the hierarchical organization of cells in the human visual cortex. It consisted of multiple layers of artificial neurons, where each layer had a specific role in processing visual information. The Neocognitron introduced the concept of filters as local receptive fields with weights which were 'shared' amongst all input pixels, which are key components in modern CNNs.

In the 1990s Yan LeCun (et. al) published seminal research proposing one of the pioneering CNN architectures which represent modern CNNs and proved state-of-the-art for handwritten digit recognition tasks in which the network learned directly from pixels via gradient descent [51] [52].

For the following decades, CNNs continued to outperform other methods on vision-based learning tasks, such as image classification, with AlexNet (2012) [53], the first major breakthrough following LeCun's architecture [52], dominating the ImageNet competition on large-scale classification tasks. In the following years, further developments such as VGGNet (2014) [54] and ResNet (2015) [32] made it possible to train deeper and deeper CNNs, each proving state-of-the-art on given tasks in the vision domain. The key feature of the ResNet architecture worth noting is that of residual or skip connections; researchers found that while in theory, deeper networks should lead to better performance, this was not always the case in practice and that a remedy for this problem was to sum the output of a given layer with its input before passing it on to the next layer, allowing the input to 'skip' the present layer - this seemingly minor development enabled performance to continue improving with increasing depth up to hundreds of layers.

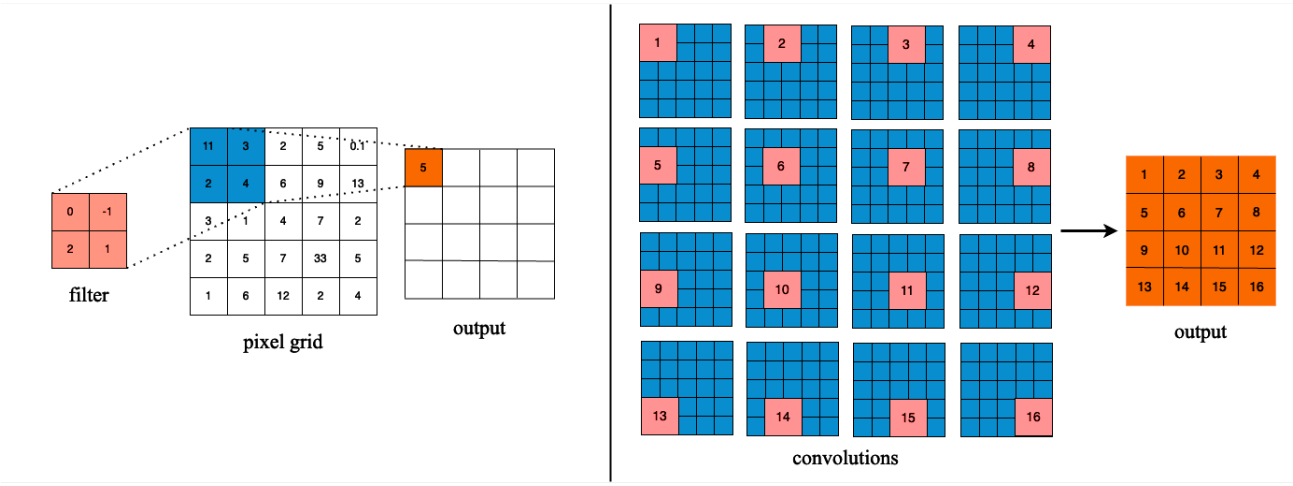


Figure 2.12.: **Left**: A single convolution operation between a 2×2 filter and a 2×2 patch in the input grid **Right**: A sequence of overlapping convolutions between a 2×2 filter and a 5×5 2D input gird, where filter moves horizontally and vertically across the image with a stride of 1.

To understand how CNNs work, we consider an input tensor, $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$; a 3-dimensional array of pixels where the first two dimensions H and W correspond to the height and width of the image/frame and a third dimension C which may correspond to colour channels or, for the purposes of reinforcement learning, it may correspond to discrete time steps.

A single convolutional layer has a set of filters, also called kernels, $F \in \mathbb{R}^{H_F \times W_F \times C}$; matrices of learnable weights which have height and width dimensions which are only a fraction of the size of those belonging to the input tensor, $\mathbf{x}$. The convolutional layer transforms the input by sliding each of its filters over the image (horizontally and vertically) computing a sequence of overlapping convolutions in which the filters' weights are multiplied by the pixel values in an element-wise fashion and then summed together, producing a scalar output which is positioned at the x-y position of the output grid corresponding to the horizontal and vertical position of the given filter over the input. In this way, each subsequent filter then produces a 3-dimensional output tensor of its own.

Pooling layers perform a down-sampling function by aggregating information in order to select important features from across each channel in the output tensor from the preceding convolutional layer. This is also done by sliding a window over the width and breadth of the output tensor selecting, for example, the largest value in the region covered by the window ("Max Pooling"), or computing the average of all the values ("Mean Pooling"). A secondary effect of the pooling operation is a reduction in the size of the tensor along the width and height dimensions.

As the input tensor (e.g. an image) moves through the convolution and pooling layers of the CNN, lower-level spatial features (e.g. edges) are combined to form higher-level spatial features (e.g. whole shapes). This occurs because the convolutions computed between the filter and the pixels of each image in each local region covered by the filters overlap, and so while a convolution in the first layer may cover a 6×6 region of pixels in the input grid, a convolution with a filter of equivalent dimension in a latter layer might include information from a 30×30 region of pixels in the input tensor. This means that the output of the final convolutional layer can encode features corresponding to entire objects, which can then be fed into the predictor portion of the network and used to produce the task-specific output (e.g. classification).

The overlapping of convolutions in a grid structure across a hierarchy of layers as described above, as well as the fact that the same filters are applied across the entire input grid in each layer in order to extract local features, have two important consequences. First, it equips CNNs with a spatial inductive bias, meaning that the learning of spatial information - the positioning of features relative to one another on a 2-dimensional grid and their hierarchical relationships - is facilitated by the network architecture itself. This makes it far easier for CNNs to learn to identify visual features than it would be for MLPs (in the case that the input pixel grid was flattened out into a 1-dimensional array). The spatial inductive bias of CNNs is believed to be a key component contributing to the exceptional performance of CNNs on vision-based tasks [55].

The second consequence of the CNN architecture, in particular the overlapping convolutions, is that an input pixel grid may *not* be divided up into square patches (of equal size) which might be processed individually, as is the case with the Vision Transformer architecture[12], which we will cover later. For smaller images, this might not be a problem, but in the case that the visual field is very large, and where only a limited number of sensors are available with which to capture the visual information such that only a part of the field may be observed at any one time, this requirement is restrictive for CNNs.

2.2.2 Recurrent Neural Networks

Feedforward neural networks is a term used to describe ANNs such as the MLP and CNN architectures discussed above. In a feedforward network data flows only in one direction, from the input to the output with no feedback loops. While suitable for certain tasks such as image classification, they are unsuitable for tasks which involve sequential data, such as time series data, which require an architecture which is able to process each element of the sequence in such a way that it incorporates information from other elements across the sequence.

Recurrent Neural Networks (RNNs) are a class of ANNs designed for processing sequential data. Unlike feedforward networks, RNNs have connections that form cycles within the network, allowing them to maintain a hidden state, represented as h_t for a given time step, t, which aggregates information from preceding elements in the sequence. This ability to capture temporal dependencies makes RNNs suitable for tasks such as language modelling, speech recognition, and time series prediction.

The Elman Network, first proposed by Jeffrey Elman in 1990 [56], was one of the earliest RNN architectures and formed the basis for many variations of the 'vanilla' RNN architecture, an example of which is illustrated in Figure 2.14(a). At each time step, t, the 'vanilla' RNN updates its hidden state by concatenating the input, x_t, and the hidden state from the previous time step, h_{t-1}, and passing them through a dense layer with a tanh activation function, for example, which maps values to within the range $(-1, 1)$ - the hidden state might then be transformed via a second dense layer into an output tensor suitable for a given task, for example, a probability distribution or a real number. The major issue with the vanilla RNN is that, in order to compute gradients over long sequences, the chain rule must be applied to compute partial derivatives going back through the unrolled network (as illustrated in Figure 2.13) resulting in so-called 'exploding' or 'vanishing' gradients which cause numerical instability as well as an inability of the network to capture the effects of

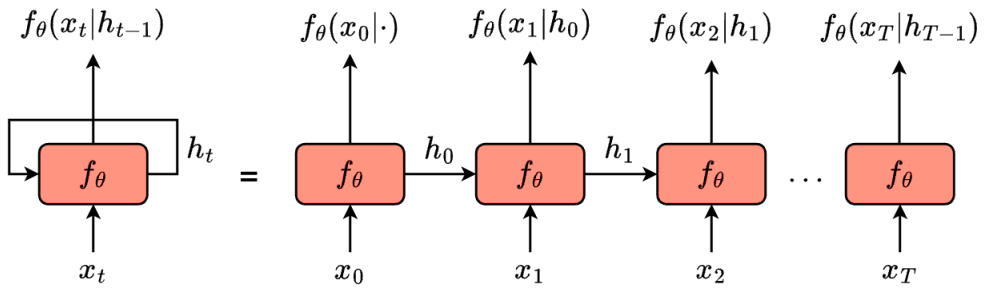
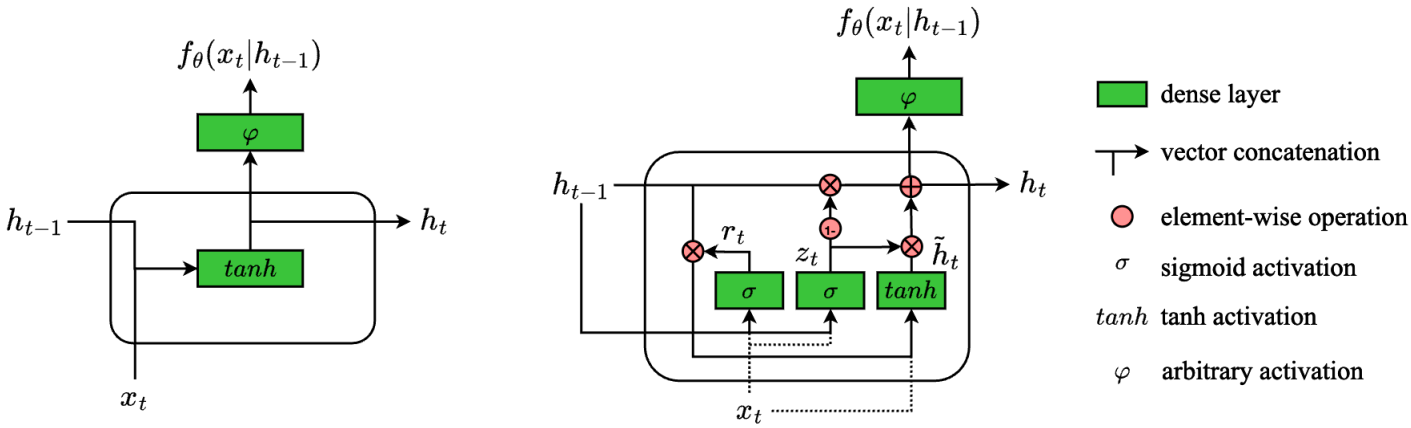


Figure 2.13.: An illustration of a recurrent neural network (RNN) 'unrolled' across time; at each time step, $t = 0, 1, \ldots, T$, the RNN processes each element x_t in the sequence producing an output which is conditioned on the hidden state, h_{t-1}, from the previous time step

long-term dependencies across the sequence which grow exponentially smaller with increasing sequence length and so are masked by short-term dependencies. Altogether, this makes it difficult to train vanilla RNNs effectively with gradient descent. [57] [58].



(a) Vanilla Recurrent Unit **(b) Gated Recurrent Unit**

Figure 2.14.: **(a)** A 'vanilla' recurrent unit comprising of a single dense layer with a tanh activation function which produces a new hidden state, h_t, given an input, x_t, and the hidden state, h_{t-1}, from the previous time step **(b)** A Gated Recurrent Unit which updates its hidden state by passing it through a reset gate, where r_t determines which information to extract from h_{t-1} producing a candidate hidden state, $\tilde{h}_t$, and an update gate which combines $\tilde{h}_t$ and h_{t-1} via linear interpolation using z_t (and $(1 - z_t)$)

The so-called "exploding/vanishing gradient problem" posed by vanilla RNNs was solved by the invention of gated RNNs such as the Long Short-Term Memory (LSTM) RNN [59], proposed in 1997, and the Gated Recurrent Unit (GRU) [60], proposed in 2014, which utilise gating mechanisms comprising dense layers followed by specifically chosen activation functions and element-wise operations allowing the network to ultimately select which information to keep and which to discard during each update of the hidden state. The effect of gating mechanisms in RNNs is to ultimately enable the network to better learn long-term dependencies.

The GRU, illustrated in Figure 2.14 **(b)**, is a simplification of the LSTM which has been shown to be more efficient, but comparable to the LSTM which was previously considered state-of-the-art across various sequence-related tasks (e.g. machine translation) [58] - it works in the following way. In updating the hidden state of the GRU, the input, x_t, and the previous hidden state, h_{t-1}, are concatenated and passed through two dense layers with sigmoid activations functions, $\sigma(x) = 1/(1 - e^{-x})$, called the 'reset gate', which outputs a tensor, r_t[12], and the 'update gate', which outputs a tensor, z_t. A candidate hidden state, $\tilde{h}_t$, is then computed by (i)

[12]Note: r_t here is a tensor, not a scalar reward as denoted in RL - we will only use this meaning here as it is convention.

taking an element-wise product of $r_t \in (0,1)^N$ with h_{t-1}, determining which information is 'remembered' (multiplied by ~ 1) or 'forgotten' (multiplied by ~ 0), and then (ii) concatenating the result with x_t and passing it through a dense layer with a tanh activation function, mapping values into the range $(-1, 1)$. The new hidden state is computed via linear interpolation as $h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$ which determines the extent to which the values in the hidden state vector are updated - note that the effect of the tanh activation function (the mapping of values to be within the range $(-1, 1)$) allows individual scalar components of the hidden state to be increased or decreased, proportional to the scalar values in z_t.

Two additional major developments in RNNs following GRUs were sequence-to-sequence (seq2seq) models [61] and the introduction of an attention mechanism [62]. Seq2seq utilises RNNs in a special encoder-decoder architecture which comprises an encoder RNN to read the input sequence and a decoder RNN to generate the output sequence, enabling the model to handle input and output sequences of different lengths. The introduction of an attention mechanism used in conjunction with RNNs allowed the network to attend to *explicitly* (focus on) different parts of the input sequence when producing each element of the output sequence. This significantly improves the model's performance on tasks requiring the transformation of long sequences. RNNs were, however, ultimately superseded as the primary architecture for sequential tasks by Transformers, a novel sequence-to-sequence modelling architecture proposed in 2017 based on a special self-attention mechanism, that has since yielded significant improvements on sequence modelling tasks across the board [11]

In RL problems wherein the state space is continuous, neural networks are able to serve as function approximators for policies and value functions. In the case of partial observability, discussed in section 2.3.5, gated RNNs present an important tool which allows us to deal with the 'non-Markovness' of the observations by maintaining a hidden state which aggregates information from all preceding observations in a given episode, which may be thought of as a latent representation of the agent's belief over possible states optimised for the purpose of solving the given task - more on this in section 2.3.5.

2.3 Deep Reinforcement Learning Methods

The application of ANNs in reinforcement learning dates back to the late 1900s. In 1989 Gerald Tesauro proposed a method which used ANNs in order to learn to play backgammon [63]. Tesauro's method used an MLP to evaluate board positions, demonstrating the potential of neural networks in reinforcement learning settings. In 1992, Ronald J. Williams introduced the REINFORCE algorithm [64], in which he proposed a new gradient-based method for policy optimisation. In REINFORCE, policies, represented as parameterised distributions over possible actions, are optimised by adjusting the parameters in the direction of the gradient of the expected return, approximated in Monte Carlo fashion. In 1999, Richard Sutton [65] built on top of Williams' work proposing a new class of policy gradient methods in which policies are represented as function approximators, such as ANNs, allowing REINFORCE to be applied to large, continuous state (and action) spaces. In 2005 a method called Neural Fitted Q Iteration [66] was proposed; an algorithm that employed MLPs for approximating the state-action value function by collecting 1-step transitions and optimising the network weights in order to minimise the temporal difference error. This work further demonstrated the feasibility of using deep learning techniques in reinforcement learning tasks and laid the foundation for later research into the intersection of Q-learning methods and deep learning.

2.3.1 Deep Q-learning

While the foundations that had been laid in the preceding decades demonstrated the potential for applying deep learning methods to solve reinforcement learning tasks, the introduction of the DQN (deep Q-Network) algorithm in the seminal paper "Playing Atari With Deep Reinforcement Learning" [9] in 2013 [13] represents a major milestone and (arguably) an inflection point after which scientific interest in reinforcement learning methods greatly increased.

[13] And the subsequent article published in the journal Nature in 2015 [10].

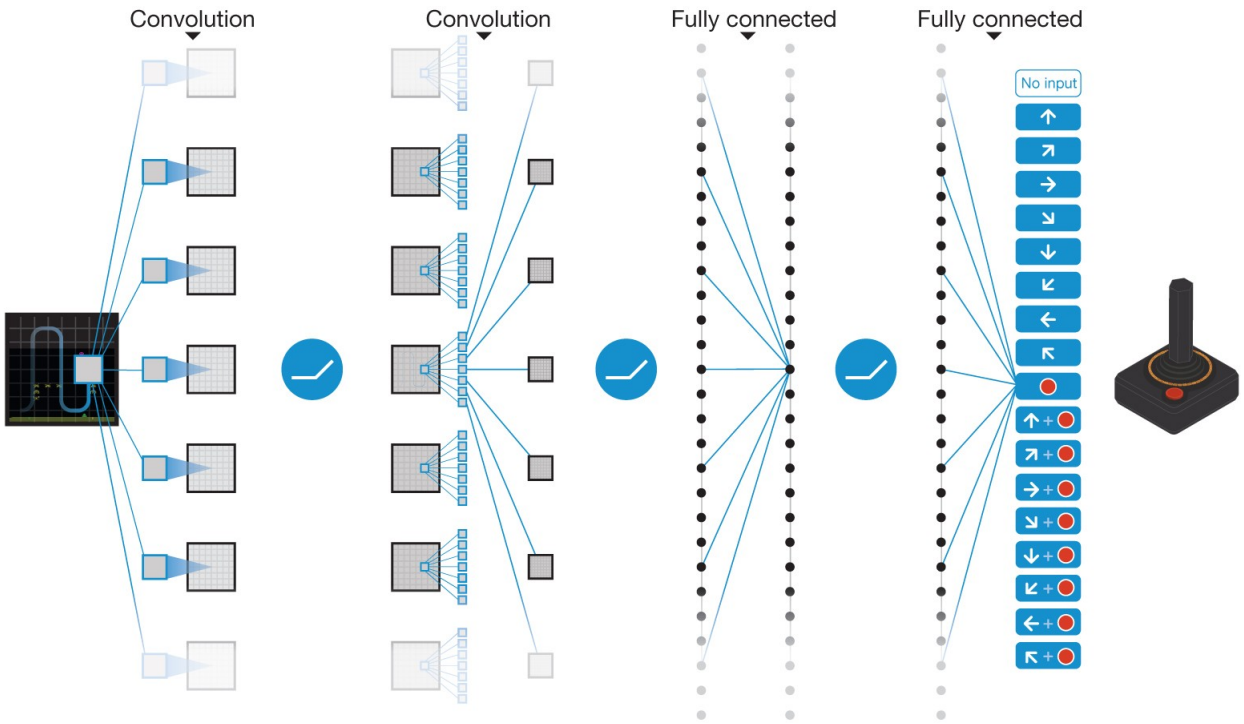


Figure 2.15.: The Q-network architecture proposed in the DQN paper [9] in order to estimate state-action values from pixel frames; two convolutional layers for feature extraction and two dense layers for transforming features into state-action value estimates.

The authors of the DQN paper argue that while the application of RL methods had seen limited success across a number of domains, their applicability had been limited to those domains with fully observable, low dimensional state spaces (e.g. finite, discrete state spaces) or those domains with high-dimensional state spaces in which useful features can be handcrafted. The novel contribution of DQN - which applied deep learning methods in the context of RL in order to teach artificial agents to play Atari 2600 games - was that it could learn successful policies directly from high-dimensional sensory inputs, such as the raw pixel grid comprising a game screen. DQN broke all records set by previous artificial agents on 49 Atari 2600 games and achieved scores comparable with a professional human game tester [10].

Before we formally define the DQN algorithm, we must consider the nature of the state space and the observations belonging to an environment constituted by a vision task such as an Atari 2600 game. In any such environment, the agent may observe only the game screen, selecting actions from a finite set in the usual way. Now, at a single time step, the game screen, represented as a 3D tensor of pixel values consists of an image of size 210×160 with three colour channels (RGB). However, a single image is unable to satisfy the Markov Property since it holds no information about, for example, the velocity of objects on the screen, which is assumed for all the RL algorithms we have considered thus far. For this reason [14], each "state", $s \in \mathcal{R}^{84\times84\times4}$, fed to the agent as input at a given time, t, consists of 4 game screens - each converted to grey-scale, resized, and cropped to 84×84 - spanning the four most recent time steps, from t to $t-3$ - a technique known as frame stacking. Using frame stacking we are able to satisfy the Markov property, as each observation captures the dynamics of the environment as well as its contents, giving us a Markov decision process $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ which represents the learning task posed by any Atari 2600 game to which DQN might be applied.

The DQN algorithm is based on the Q-learning algorithm [30] (Algorithm 2) in which the agent seeks to learn, in an iterative fashion, increasingly better approximations of the optimal state-action value function, q^*. Since the state space, in the case of a vision-based task, is *theoretically* continuous[15] and therefore too large for tabular methods such as tabular Q-learning, the authors propose using an ANN (as opposed to a lookup table) to model the state-action value function, denoted $Q_\theta : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ and having parameters $\boldsymbol{\theta} \in \mathbb{R}^m$.

[14] As well as other engineering-related reasons we won't cover here.
[15] Again, due to how computers represent real numbers it cannot be truly continuous in practice.

$Q_{\boldsymbol{\theta}}$ takes as input a 3D tensor of pixel values[16] representing the state of the environment, producing a vector of value estimates for each possible action. The proposed DQN architecture, shown in Figure 2.15, consists of two convolutional layers which learn to extract visuospatial features, followed by two fully connected layers which learn transformations from the feature vectors to a vector of state-action value estimates. Whereas, in ordinary Q-learning, updates to Q are made by updating individual value estimates of state-action pairs in a lookup table via the update rule 2.23, updates in DQN are performed by updating the parameters of the Q-network, $Q_{\boldsymbol{\theta}}$, according to some optimisation objective, a process which we now detail.

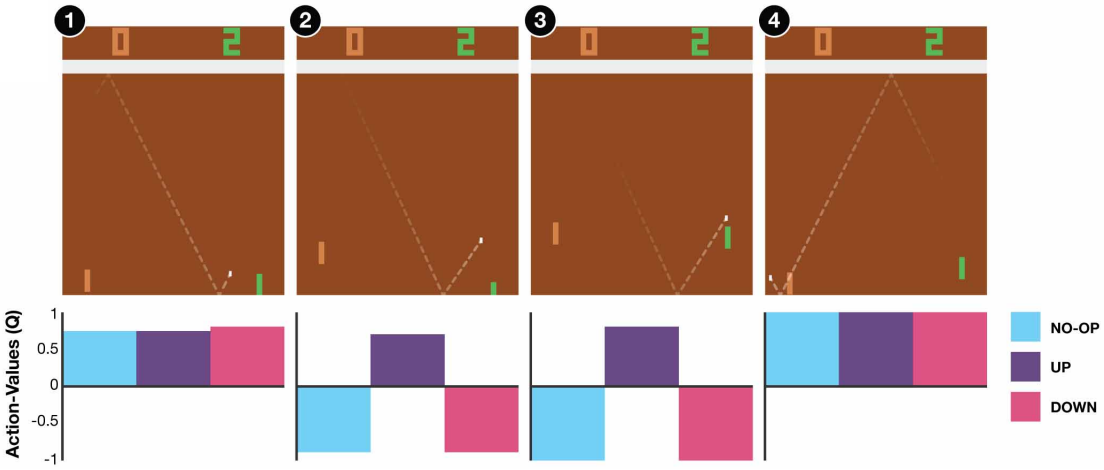


Figure 2.16.: A representation of state-action value estimates made by the Q-network of a DQN agent in Atari Pong [10] where the agent plays against the environment and controls the green paddle. Time goes from left to right. On the left, the agent is agnostic as to which action (up, down, or nothing) is best. In the middle two frames, it estimates the value of the 'up' action to be higher than the other two as the ball is above the agent's paddle. In the final frame the agent is about to win, regardless of which action it chooses, and so estimates a high action value for all 3 actions.

In optimising the Q-network, we begin with a vector of randomly initialised parameters, $\boldsymbol{\theta}^{(0)}$, where $\boldsymbol{\theta}^{(k)}$ represents the parameters following the k^{th} update. Then, the agent iteratively interacts with the environment using an ϵ-greedy policy[17] to collect experiences $(s_t, a_t, r_{t+1}, s_{t+1})$ in a memory buffer $\mathcal{B}$. At each step, we sample a mini-batch B of transition tuples, (s, a, r, s'), from $\mathcal{B}$ to compute the TD-based loss function:

$$J(\boldsymbol{\theta}^{(k)}) = \mathbb{E}_{a,s\sim\rho(\cdot)}\left[(y - Q(s, a; \boldsymbol{\theta}^{(k)}))^2\right] \tag{2.34}$$

which is an expectation over all state-action pairs distributed according to the so-called "behaviour distribution", $\rho(s, a)$; the joint probability distribution determined by the state-transition probability distribution function and current ϵ-soft policy derived from $Q_{\boldsymbol{\theta}^{(k)}}$. Then, the DQN bootstrap target, y, in equation 2.34 is computed for each tuple in the batch, B, as:

$$y = r_{j+1} + \gamma \max_{a'} Q(s_{j+1}, a'; \boldsymbol{\theta}^{(k-1)}) \tag{2.35}$$

where $Q(s, a; \boldsymbol{\theta}^{(k-1)})$, the target network, is a time-delayed copy of the principle Q-network, updated every C steps during training. The reason for using a time-delayed copy of the Q-network to compute the target, y, is because the correlations that would arise naturally when attempting to use value estimates, determined by $\boldsymbol{\theta}^{(k)}$ in the loss function used to update $\boldsymbol{\theta}^{(k)}$, cause instability in the learning (optimisation) process, which is alleviated by the use of the target network. Notice that the objective function 2.34 has the same form as the update rule 2.23, approximating q^* in the same greedy manner. Finally, in order to update the parameters of the Q-network, the gradient of the loss function 2.34 is computed as:

[16] Grey scale game screens stacked along the time dimension.

[17] Actions are selected randomly with probability $1 - \epsilon$ and greedily, according to $\arg\max_a Q(s_t, a; \boldsymbol{\theta}^{(k)})$, otherwise.

$$\nabla_{\boldsymbol{\theta}^{(k)}} J(\boldsymbol{\theta}^{(k)}) = \mathbb{E}_{a,s\sim\rho(\cdot),s'\sim P(\cdot)}\left[\left((r + \gamma \max_{a'} Q(s', a'; \boldsymbol{\theta}^{(k-1)})) - Q(s, a; \boldsymbol{\theta}^{(k)})\right) \nabla_{\boldsymbol{\theta}} Q(s, a; \boldsymbol{\theta}^{(k)})\right] \quad (2.36)$$

where the expectation is typically approximated during training by computing and averaging out the inner expression over the entire batch, B, of experience. The gradients computed over a single mini-batch are typically applied using a gradient descent algorithm like *Stochastic Gradient Descent* (SGD) or the *Adam* algorithm.

Algorithm 3 Deep Q-Learning (DQN) With Experience Replay

Require: $\gamma \in (0, 1]$
Require: $\epsilon \ll 1$
Require: $C \in \mathbb{N}$
  $\mathcal{B} \leftarrow \emptyset$ (memory buffer)
  $k \leftarrow 0$
  Initialise main and target Q-networks with random parameters $\boldsymbol{\theta}^{(0)}$
  for episode=1,M **do**
    Observe s_1
    for t=1,T **do**
      With probability ϵ select a random action a_t
      Else select $a_t = \arg\max_a Q_(s_t, a; \boldsymbol{\theta})$
      Execute a_t and receive r_{t+1}, s_{t+1}
      Store experience $(s_t, a_t, r_{t+1}, s_{t+1})$ in $\mathcal{B}$
      Sample random mini-batch of transitions $(s_j, a_j, r_{j+1}, s_{j+1})$ from $\mathcal{B}$
      Set $y_j = \begin{cases} r_{j+1} & \text{if } s_{j+1} \text{ is terminal} \\ r_{j+1} + \gamma \max_{a'} Q(s_{j+1}, a'; \boldsymbol{\theta}^{(k-1)}) & \text{otherwise} \end{cases}$
      Update $\boldsymbol{\theta}^{(k)}$ using the gradient $\nabla_{\boldsymbol{\theta}^{(k)}} J(\boldsymbol{\theta}^{(k)})$ as per equation 2.36
      Every C steps, update target network weights to be $\boldsymbol{\theta}^{(k)}$ and increment k by 1
    end for
  end for

There are two important points to note regarding the experience replay buffer, $\mathcal{B}$, used in DQN to collect transition tuples, as it is not used in tabular Q-learning. Firstly, the individual transition tuples are policy-agnostic and may therefore be used to compute updates regardless of the policy under which they were sampled. Second, the distribution of state-action pairs in $\mathcal{B}$ is non-stationary, tracking the behaviour distribution, $\rho(s, a)$, with each successive update to the Q-network and allowing accurate approximations of the expectation 2.34.

The advent of the DQN algorithm was a transformative event in the field of reinforcement learning. By successfully integrating deep learning techniques with Q-learning, DQN has demonstrated an unprecedented ability to handle high-dimensional sensory inputs and learn directly from raw data. This breakthrough not only set new benchmarks in the domain of video game playing, where it has performed on par with human experts, but it also ignited a surge of interest and progress in reinforcement learning research.

2.3.2 Policy Gradient Methods

In the seminal REINFORCE paper by Ronald Williams [64], a novel method for reinforcement learning was introduced that uses a policy gradient approach to policy optimisation. The policy, represented as a probability distribution over actions given the current state, is adjusted in the direction of the gradient of the expected return. This gradient is estimated by sampling trajectories using the current policy. A key insight of the paper is that this gradient can be estimated without needing to know the explicit functions (and therefore their derivatives) which govern the environment's dynamics (i.e. the state-transition probability and reward functions), making the algorithm widely applicable, since in many cases these functions are unavailable.

Williams also introduced the concept of using a baseline to reduce the variance of the gradient estimates, thereby improving the efficiency of the learning process - more on this in section 2.3.3.

Richard Sutton extended Williams' REINFORCE algorithm by incorporating function approximation into the policy gradient framework [65]. This significant extension allowed the algorithm to be applied to problems with large or continuous state and action spaces, which were previously intractable with the original REINFORCE algorithm. Sutton replaced the representation of the policy as a state-dependent, parameterised distribution, with a parameterised function approximator, such as an ANN, which is differentiable with respect to its parameters. He also improved on Williams' baseline idea by proposing the use of value function estimates [18] as a baseline to reduce the variance of the gradient estimates, leading to faster and more stable learning. Sutton provided a rigorous proof of the policy gradient theorem, forming the basis for many modern reinforcement learning algorithms. We now cover the basic policy gradient algorithm (REINFORCE with function approximation).

We consider an RL task modelled as an MDP $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$. In a policy gradient method, the objective is to learn an explicit policy which maps states to a probability distribution over actions (as opposed to action values), $\pi_{\boldsymbol{\theta}} : \mathcal{S} \times \mathcal{A} \to [0, 1]$, which we take to be an ANN with parameters $\boldsymbol{\theta} \in \mathbb{R}^m$, where the probability distribution is generated using a *softmax* activation function of the form:

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad for\ i = 1, 2, \ldots, K$$

The primary difference between DQN (and its associated class of value-based methods) and policy gradient methods lies in their approaches to action-value estimation. The Q-network in DQN directly estimates the value of actions for each state, while policy gradient methods utilise a policy network, $\pi_{\boldsymbol{\theta}}(a|s)$, to output action preferences as probabilities within a given state. These probabilities are conditioned directly on rewards, allowing the policy network to implicitly learn the relative 'value' of actions in that state.

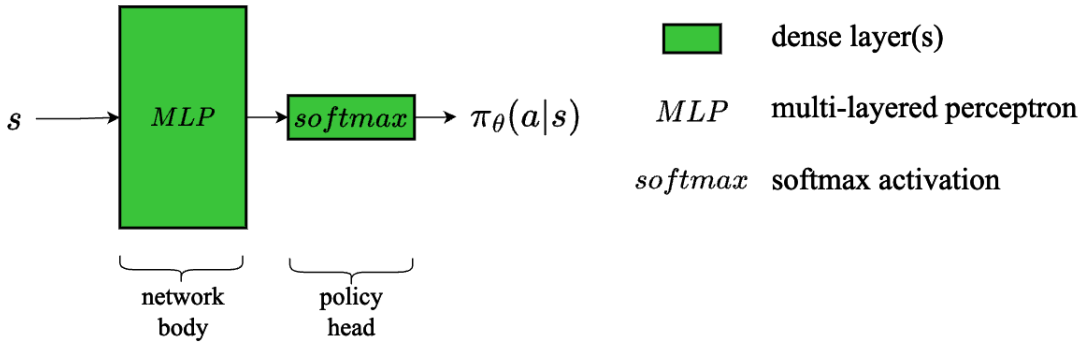


Figure 2.17.: A simple policy gradient network with an MLP forming the network body and a policy head comprising of a dense layer with a softmax activation function to transform the outputs of the final layer into a probability distribution over actions.

As usual, the objective is to learn an optimal policy which maximises the expected return received by the agent. Policy gradient methods utilise Monte Carlo methods in order to sample trajectories of experience from the environment and estimate returns for the purposes of policy optimisation. To this end, as with DQN, we require an objective function in order to evaluate and optimise our policy network, which, for policy gradient methods, takes the form:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim \rho_{\boldsymbol{\theta}}(\cdot)} \left[R(\tau) \right] \tag{2.37}$$

which, in words, is the expected return over all trajectories τ through the state action space, $\mathcal{S} \times \mathcal{A}$, given some fixed policy parameters, $\boldsymbol{\theta}$. Here, a trajectory, τ, is defined as a distinct sequence of states and actions observed and chosen by an agent, respectively, when interacting with an environment:

[18] In practice, the baseline is estimated from observed rewards.

$$s_0, a_0, s_1, a_1, \ldots, s_{T-1}, a_{T-1}, s_T$$

where s_T is taken to be a terminal state. The probability distribution $\rho_{\boldsymbol{\theta}}(\tau)$ determines the probability of a given trajectory through the state-action space under $\boldsymbol{\theta}$ - equivalent to the "behaviour distribution" seen in the DQN objective function above, it is simply a joint probability distribution determined by the agent's policy, $\pi_{\boldsymbol{\theta}}(a|s)$, used to determine action selection probabilities, and the state transition probability function, $P(s'|s,a)$. Now, in order to improve our policy with respect to our objective function, our strategy will be, in a similar manner to the Monte Carlo method detailed above, for the agent to sample a number of trajectories in order to sufficiently approximate $J(\boldsymbol{\theta})$ using sample returns. Then, using our approximation, we will seek to compute its gradient, and perform the following optimization step (or something akin to this using a given optimisation method, such as Adam):

$$\boldsymbol{\theta} \leftarrow (1-\alpha)\boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{2.38}$$

where $\alpha \in (0,1)$ is the learning rate. Note, as opposed to gradient *descent*, the update 2.38 is a gradient *ascent* step; updating the parameters $\boldsymbol{\theta}$ in this way should theoretically lead to more optimal agent behaviour, i.e. action-selection which increases the expected return over all possible trajectories, and better approximation of the optimal policy, π^*.

So, we seek to compute $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$, that is, the gradient of the expectation of the return over all trajectories under a fixed policy, a task which is computationally intractable. To solve this problem, notice that the objective function 2.37, being an expectation, can be expressed as an integral over all possible trajectories where the integrand consists of the product of the return for a given trajectory, $R(\tau)$, and the probability of that trajectory being sampled under a given policy, $\rho_{\boldsymbol{\theta}}(\tau)$. Then, using the properties of the logarithm and the form of its derivative, we can show that the gradient of the expectation of $J(\boldsymbol{\theta})$ is in fact the expectation of the quantity $R(\tau)\nabla_{\boldsymbol{\theta}} log \rho_{\boldsymbol{\theta}}(\tau)$ as follows:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \int_{\tau} R(\tau)\rho_{\boldsymbol{\theta}}(\tau) d\tau && (2.39)\\
&= \int_{\tau} R(\tau)\nabla_{\boldsymbol{\theta}}\rho_{\boldsymbol{\theta}}(\tau) d\tau && (2.40)\\
&= \int_{\tau} R(\tau)\frac{\nabla_{\boldsymbol{\theta}}\rho_{\boldsymbol{\theta}}(\tau)}{\rho_{\boldsymbol{\theta}}(\tau)}\rho_{\boldsymbol{\theta}}(\tau) d\tau && (2.41)\\
&= \int_{\tau} R(\tau)\nabla_{\boldsymbol{\theta}} log\rho_{\boldsymbol{\theta}}(\tau)\rho_{\boldsymbol{\theta}}(\tau) d\tau && (2.42)\\
&= \mathbb{E}_{\tau\sim\rho_{\boldsymbol{\theta}}(\cdot)}\left[R(\tau)\nabla_{\boldsymbol{\theta}} log\rho_{\boldsymbol{\theta}}(\tau)\right] && (2.43)
\end{aligned}$$

Next, consider the expansion of the behaviour distribution, $\rho_{\boldsymbol{\theta}}$, for a given trajectory, τ, which is just the product of probabilities for each state and action in the sequence:

$$\begin{aligned}
\rho_{\boldsymbol{\theta}}(\tau) &= p_0(s_0)\pi_{\boldsymbol{\theta}}(a_0|s_0)p(s_1|s_0,a_0)\pi_{\boldsymbol{\theta}}(a_1|s_1)\ldots\pi_{\boldsymbol{\theta}}(a_{T-1}|s_{T-1})p(s_T|s_{T-1},a_{T-1}) && (2.44)\\
&= p_0(s_0)\prod_{t=0}^{T}\pi_{\boldsymbol{\theta}}(a_t|s_t)p(s_{t+1}|s_t,a_t) && (2.45)
\end{aligned}$$

where p_0 gives the probability distribution over starting states, defined implicitly by the environment. Now, it is not possible to compute the gradient of the state transition probability function, $P(s'|s,a)$, since it is not explicitly defined. However, expanding the gradient term in expression 2.43, we see that since $P(s'|s,a)$ does not depend on $\boldsymbol{\theta}$, all its corresponding terms disappear:

$$\nabla_{\boldsymbol{\theta}} log \rho_{\boldsymbol{\theta}}(\tau) = \nabla_{\boldsymbol{\theta}} log \left[p_0(s_0) \prod_{t=0}^{T} \pi_{\boldsymbol{\theta}}(a_t|s_t) p(s_{t+1}|s_t, a_t) \right] \tag{2.46}$$

$$= \sum_{t=0}^{T} \nabla_{\boldsymbol{\theta}} log \pi_{\boldsymbol{\theta}}(a_t|s_t) \tag{2.47}$$

Finally, since $\mathbb{E}_{x \sim P(\cdot)}[f(x)] = \lim_{N \to \infty} \frac{1}{N} \sum_{j=1}^{N} f(x_j)_{x_j \sim P(\cdot)}$ we can approximate the expectation of the objective function in Monte Carlo fashion and, using our result from 2.47, we get:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim \rho_{\boldsymbol{\theta}}(\cdot)}[R(\tau) \nabla_{\boldsymbol{\theta}} log \rho_{\boldsymbol{\theta}}(\tau)] \tag{2.48}$$

$$\approx \frac{1}{N} \sum_{j=1}^{N} \left[\sum_{t=1}^{T} \left(\nabla_{\boldsymbol{\theta}} log \pi_{\boldsymbol{\theta}}(a_t|s_t) (\sum_{k=t}^{T} \gamma^{k-t} r_{k+1}) \right) \right] \tag{2.49}$$

$$= \frac{1}{N} \sum_{j=1}^{N} \left[\sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} log \pi_{\boldsymbol{\theta}}(a_t|s_t) G_t \right] \tag{2.50}$$

where $N \gg 1$ is the number of trajectories sampled from having the agent interact with the environment, T is the number of time steps in each trajectory, and G_t is the return obtained following time step t. Note that updating our policy parameters using the approximation 2.50 with gradient ascent optimisation will increase or decrease the probability of a given action relative to other actions depending on the relative value of its associated return - this was the observation made by Williams in [64]. An example of a vanilla policy gradient algorithm is given in full in Algorithm 4.

Algorithm 4 Vanilla Policy Gradient

Require: $\gamma \in (0, 1]$
Require: $\alpha \in (0, 1)$
Initialise policy network $\pi_{\boldsymbol{\theta}}$ with random parameters $\boldsymbol{\theta}$.
for $k = 1, 2, 3, \ldots$ **do**
Sample trajectories $\tau_1, \ldots, \tau_N$, consisting of tuples $(s_j, a_j, r_{j+1}, s_{j+1})$ for $j = 1, \ldots, T$, with policy $\pi_{\boldsymbol{\theta}}$
Compute the approximation of the objective function $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \frac{1}{N} \sum_{j=1}^{N} \left[\sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} log \pi(a_t|s_t; \boldsymbol{\theta}) G_t \right]$
Update the policy parameters $\boldsymbol{\theta} \leftarrow (1 - \alpha)\boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
end for

2.3.3 Deep Actor-Critic (A2C) Methods

In the original paper in which REINFORCE was proposed [64], Williams noted that a source of instability when performing policy optimisation with gradient estimates of the form given by 2.50 is the high variance which may be observed in the distribution of returns in a given task, as it is not a given that the rewards are bounded (e.g. in the range (0, 1)) or normalised in any way. This can result in high-variance gradient estimates which in turn translates to high-variance parameter updates to the policy parameters, and hence instability in learning, which is undesirable. To address this problem, Williams proposes the idea of subtracting a real-valued baseline from all empirical returns giving a new objective function of the form:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim \rho(\cdot)}[G(\tau) - b(\tau)] \tag{2.51}$$

where $b(\tau)$ is a real-valued baseline computed for each return, $R(\tau)$. Now, to approximately normalise an empirical distribution, one might subtract the sample mean and divide it by the sample variance which, if the distribution is normal, transforms it into something approximating the standard normal distribution of

mean 0 and variance 1. Thus, we would like the baseline, $b(\tau)$, to be something akin to the expectation of the returns computed during training, and since the mean is likely to change over time (hopefully the agent's performance improves as its policy is updated), we would like the baseline to adjust accordingly. Based on this observation, Richard Sutton [65] extended Williams's idea by proposing that a good candidate for the baseline might be an estimate of the state value, $V(s) \approx \mathbb{E}[G_t|s_t = s]$, the expected return from a given state, s, over all possible actions (see 2.5). Given that the empirical return, the discounted sum of rewards obtained at time t following the taking of an action a_t in some state, s_t, is an estimation of the state-action value, using the state value as a baseline we arrive at a quantity known as the advantage estimate:

$$\hat{A}_t = G_t - V(s_t) \approx q^{\pi}(a_t, s_t) - v^{\pi}(s_t)$$

Notice that the advantage estimate approximates the difference between the expected return for taking a specific action a_t in s_t and the expected return over all possible actions in s_t. As such, the advantage estimate tells us whether the specific action chosen resulted in a better or worse return than what was expected for the given state under a fixed policy.

In order to implement this in practice we require, in addition to a parameterised policy, $\pi_{\boldsymbol{\theta}}$, referred to as the 'actor' network, a parameterised state-value function approximator, $V_\omega : \mathcal{S} \to \mathbb{R}$, known as the 'critic' network which we take to be an ANN with parameters ω. In practice, the actor-network and the critic network may be separate networks, or they may share a network body which has an actor and a critic "head", as illustrated in Figure 2.18. During training, V_ω is conditioned alongside $\pi_{\boldsymbol{\theta}}$ in a supervised manner on the returns estimated across each set of trajectories collected in order to minimise the loss function:

$$\mathcal{L}(\omega) = \mathbb{E}_{s\sim\rho(\cdot)}[(y - V_\omega(s))^2] \tag{2.52}$$

where $y = G_t$ is simply the empirical (sample) return obtained for each observed state, s. As training proceeds, the function approximator is constantly optimised to estimate the expected returns under the non-stationary policy and may be used to compute an approximation of an adapted version of the policy gradient objective function which uses the advantage as opposed to the observed return:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \frac{1}{N} \sum_{j=1}^{N} \left[\sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} log \pi_{\boldsymbol{\theta}}(a_t|s_t) \hat{A}_t \right] \tag{2.53}$$

The class of algorithms which utilise value function approximation in order to condition the agent's policy, known as *Advantage Actor-Critic* (A2C) methods, are the basis for many further developments in policy gradient algorithms such as PPO, which we explore in the following section.

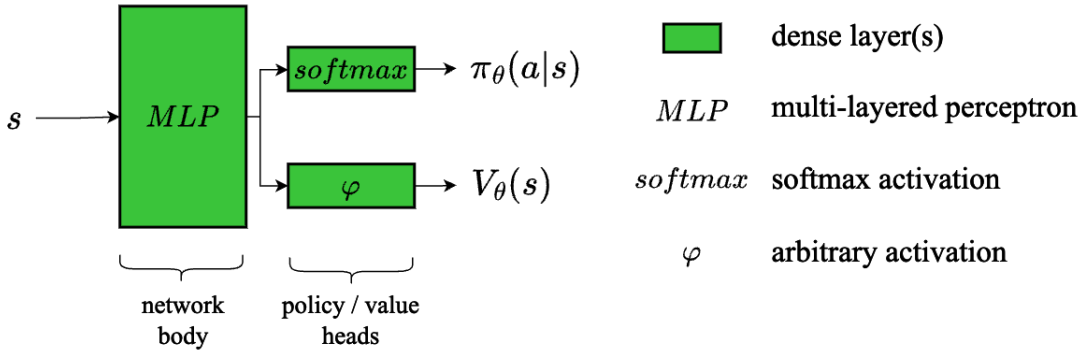


Figure 2.18.: A simple actor-critic network with an MLP forming the network body, a policy head, and a value head which outputs a single number estimating the value for the given state.

2.3.4 Proximal Policy Optimisation

Further to the challenge of instability in deep reinforcement learning, and deep learning in general, is the phenomenon known as "catastrophic forgetting" [67] [68]; when optimising the parameters of an ANN with gradient descent it can sometimes happen that a gradient step may occur too much in one direction in a portion of the parameter space where the gradient is very steep (i.e. where the magnitude of the gradient norm is large) and consequently one may observe a sharp, nearly immediate drop in performance. This phenomenon is called catastrophic forgetting since the network appears to 'forget' - almost instantly - what it 'learned' earlier in the optimisation process. While this is an issue in the field of deep learning at large, it presents a particular challenge in reinforcement learning as the learning task is non-stationary (as the agent learns it explores new parts of the state space) [69] and the learning algorithms are often sensitive to hyperparameter settings like the learning rate α [70].

Proximal Policy Optimization (PPO) [70], a policy gradient method proposed in 2017, seeks to address this issue. PPO is, as with A2C, a simple variation of the classic policy gradient algorithm, which takes inspiration from previously proposed methods seeking to solve the same problem, such as Trust Region Policy Optimisation, which while effective had many shortcomings in the form of constraints (e.g. the value and policy networks may not share weights) [70] [71]. PPO was developed with the objective of striking a balance between ease of implementation, good sample complexity, and ease of hyperparameter tuning whilst ensuring more stable policy optimization by constraining the policy parameter updates at each optimisation step such that the updated policy does not deviate too far from the current policy [70]. PPO achieves this aim by optimising a specially designed objective function:

$$J^{CLIP}(\boldsymbol{\theta}) = \mathbb{E}_t \left[\min \left(r_t(\boldsymbol{\theta}) \hat{A}_t, clip \left(r_t(\boldsymbol{\theta}), 1 - \epsilon, 1 + \epsilon \right) \hat{A}_t \right) \right] \tag{2.54}$$

where $\epsilon \ll 1$ (say 0.2) is a hyperparameter, and the $clip(x, l_{low}, l_{high})$ function clips the value of x to be within l_{low} and l_{high}. Optimisation according to 2.54 takes place over several mini-batch updates - as opposed to a single update step as in vanilla policy gradient or DQN - wherein the policy prior to optimisation is held fixed and the new version of the policy is generated via incremental, constrained parameter updates. The magnitude of change between the updated version of the policy and the previous version is quantified by the ratio of action probabilities, $r(\boldsymbol{\theta})$, between the fixed policy, $\pi_{\boldsymbol{\theta}_{old}}$, and the policy undergoing optimisation, $\pi_{\boldsymbol{\theta}}$, which is defined as:

$$r(\boldsymbol{\theta}) = \frac{\pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}_{old}(a|s)}} \tag{2.55}$$

In words, the goal of PPO optimization is to maximise J^{CLIP} with respect to $\boldsymbol{\theta}$ but to ensure that the magnitude of the gradient with respect to $\boldsymbol{\theta}$ is bounded, both above and below. This is achieved in the following way. Consider a state-action pair (s, a) from a batch of experience collected under $\pi_{\boldsymbol{\theta}_{old}}$ and the ratio, $r(\boldsymbol{\theta})$, from the PPO objective function. Now, if for s and a we have $\hat{A}(s, a) > 0$, then the optimization process for that single observation would result in an increase in $r(\boldsymbol{\theta})$, since we would want the agent to perform a when observing s more often on average. However, to ensure that the change is not too large, an upper bound of $(1+\epsilon)\hat{A}(s, a)$ is placed on the value of J^{CLIP}. The result of this upper bound is that, at a certain point in the optimization, the gradients of J^{CLIP} with respect to $\boldsymbol{\theta}$ for state-action pairs (s, a) (or similar) will become zero, thus preventing any further gradient updates to $\boldsymbol{\theta}$ in the direction in parameter space induced by (s, a). Similarly, if $\hat{A}(s, a) < 0$ for some (s, a), the optimization process would attempt to decrease $r(\boldsymbol{\theta})$ - and so *increase* $r(\boldsymbol{\theta})\hat{A}(s, a)$ - but this time J^{CLIP} is bounded above by $(1 - \epsilon)\hat{A}(s, a)$, that is, the negative quantity would be prevented from getting too close to zero.

Figure 2.19 illustrates the effect of the clipping function used in J^{CLIP}; the red dot indicates $r(\boldsymbol{\theta}) = 1$, which is the starting point for every optimization step during training. The important part to notice is that as $r(\boldsymbol{\theta})$ changes in order to *increase* the expected advantage (for both positive and negative values of the advantage estimate) the value of J^{CLIP} is bounded above and thus the partial derivatives with respect to $\boldsymbol{\theta}$ constituting the gradient will be driven to zero if the magnitude of change in the parameters relative to $\boldsymbol{\theta}_{old}$ is too large. This simple change to the objective function often yields great performance improvement on tasks where simpler policy gradient methods struggle to perform [70].

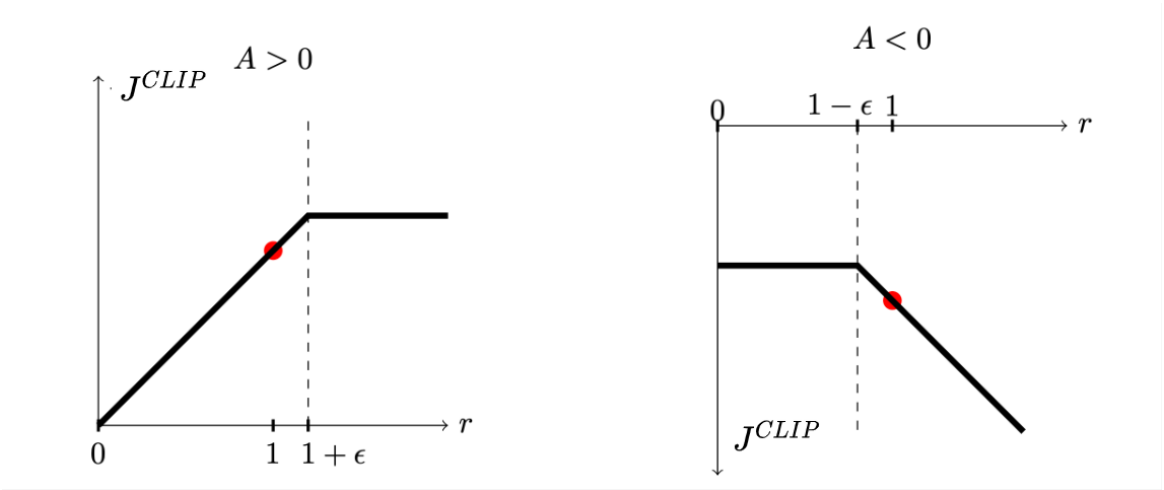


Figure 2.19.: The PPO objective J^{CLIP} as a function of the ratio, $r(\boldsymbol{\theta})$, which defines the magnitude of the change of the agent policy during the optimization process.

2.3.5 Partial Observability in RL

While the ability to observe the full underlying state of an MDP is desirable for the purposes of optimal decision-making, this is often not possible in many real-world problems, wherein only a partial observation is available, which is only probabilistically related to the true underlying state. Under such conditions, the learning task may instead be modelled as a *partially observable Markov decision processes* (POMDP) - a generalisation of the MDP first proposed by Edward Sondik in 1971 [72] where the underlying dynamics of the environment are still determined by an MDP, but in which the agent receives only partial observations probabilistically related to the true state, as opposed to the true state itself.

Formally, a POMDP is a seven-tuple $(\mathcal{S}, \mathcal{A}, R, P, \Omega, O, \gamma)$ having all the components of an MDP with the addition of Ω, the set of all possible partial observations, and $O : \mathcal{S} \times \mathcal{A} \times \Omega \rightarrow [0, 1]$, a probability distribution over observations conditioned on states and actions. In an MDP, upon transition to a state, s', after taking action, a, the agent observes this state directly. In a POMDP the agent instead receives a partial observation, $o \in \Omega$, which depends on the new state, s', and action, a, with probability $\mathcal{O}(o|s', a)$ (or simply $\mathcal{O}(o|s')$ depending on the environment).

In a POMDP setting, the Markov property does not hold for the partial observations received by the agent, that is, the probability distribution over subsequent observations changes depending on whether you consider only the most recent observation-action pair, or whether you incorporate information from the historical sequence of observations and actions. As an illustration, consider the example of an observation comprising a single frame from a video game screen with a moving object (e.g. a ball). As explained in section 2.1.3, this is insufficient to deduce the velocity of the object, therefore making the observation non-Markov. Now, consider how the probability distribution over subsequent observations (e.g. the position of the object at the next time step) changes as you begin to consider historical observations and actions moving backwards through the sequence. The most recent observation only gives you position making it impossible to even deduce the direction of the object, the two most recent observation-action pairs allow for an approximation of both the speed and direction of the object (and how the agent's actions might have affected them), and the three most recent might even allow us to get an approximation of acceleration. In this way, each additional historical observation-action pair lends additional information allowing us to make a better guess at the true underlying state, therefore altering the probability distribution over which subsequent observations we might expect the agent to receive.

To appreciate the challenge presented by partial observability in the context of RL, consider the following. Say we have two distinct partial observations which are highly similar, but which have been drawn from two different states of the underlying MDP. The first state is undesirable, having associated with it a large negative reward, and the second state is desirable, having a large positive reward associated with it. How then is the agent to learn to distinguish between these two observations when actions are selected on the basis of a single

observation alone?

The following theory is thoroughly detailed in [73]. Given that considering the historical sequence of actions and observations allows for a more accurate estimation of the environment dynamics, and so the underlying environment state, the way to overcome the issue of distinguishing between atomic partial observations is for the agent to maintain a belief b - a probability distribution over all possible states conditioned on the entire historical sequence of observation-action pairs - which forms the basis for value estimation and therefore for the agent's policy.

Assuming a discrete state space, and given that both the state-transition function, P, and the observation probability function, O, are known[19], the belief is the probability mass function $b = \{b(s_1), \ldots, b(s_{|\mathcal{S}|})\}$, which may be iteratively updated as follows. Suppose the environment is in some state, s, and that after performing action a, the agent receives an observation, o, with probability $\mathcal{O}(o|s', a)$. Let $b'(s')$ denote the probability that the environment transitions to some new state[20], $s' \in \mathcal{S}$, then the belief update is computed as:

$$b'(s') = Pr(s'|b, a, o) \tag{2.56}$$

$$= \frac{\mathcal{O}(o|s', a) \sum_{s \in \mathcal{S}} P(s'|s, a) b(s)}{Pr(o|b, a)} \tag{2.57}$$

where $Pr(o|b, a) = \sum_{s' \in \mathcal{S}} \mathcal{O}(o|s', a) \sum_{s \in \mathcal{S}} P(s'|s, a) b(s)$. Note that the update to the belief incorporates information from the existing belief, the action taken, and the new observation received - this is illustrated in Figure 2.20.

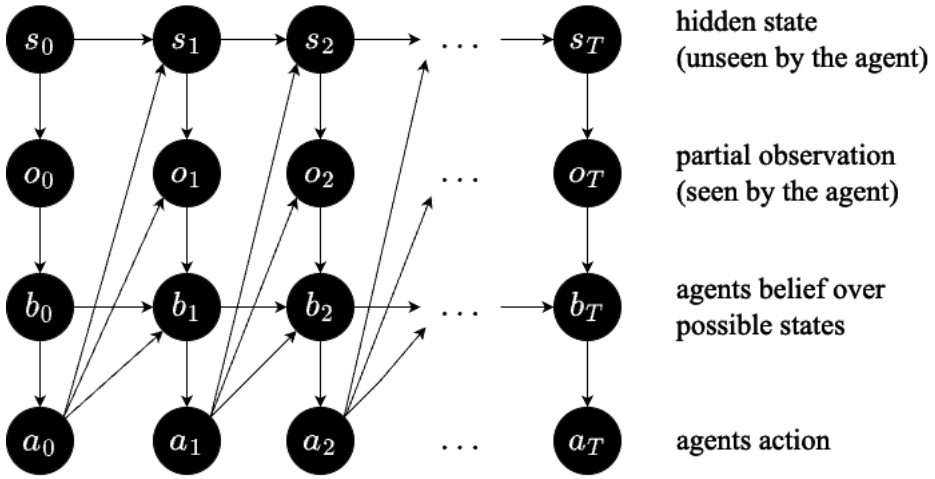


Figure 2.20.: An illustration of a belief-space MDP. At each time step an observation is sampled based on the environment state and the previous action. The agent then updates its belief based on the observation it receives and the previous belief, as well as its previous action, aggregating information across the trajectory.

Formulated this way, we can convert POMDP into a so-called *belief-space MDP* in which the belief itself is a state and which is no longer partially observable as the belief is always fully knowable by the agent [73]. Formally a belief-space MDP is a five-tuple $(\mathcal{B}, \mathcal{A}, \tilde{P}, \tilde{R}, \gamma)$ where $\mathcal{B}$ is the set of all beliefs over states, and $\mathcal{A}$ the set of all actions, belonging to the underlying MDP. Actions are selected by the agent on the basis of beliefs, as opposed to observations, using a policy, $\pi : \mathcal{B} \times \mathcal{A} \to [0, 1]$, and the expected reward for taking an action, a, given a belief, b, is then given by:

$$\tilde{R}(b, a) = \sum_{s_i \in \mathcal{S}} b(s_i) R(s_i, a)$$

[19] As is the case in the field of Dynamic Programming[1].

[20] Note: here we mean the probability as estimated by the agent which forms part of its 'belief', not the true state transition probability as determined by the state-transition probability function $P(s'|s, a)$ of the underlying MDP.

Finally, the transition probability function between beliefs becomes:

$$\tilde{P}(b'|b,a) = \sum_{o \in \Omega} Pr(o|b,a)\mathbb{1}(b' = b^{a,o})$$

that is, we sum the probabilities of all observations which will result in b' being the next belief. Given the transition probability and reward functions over possible beliefs, it is possible for us to define the belief-action value function:

$$q(b,a) = \sum_{b'} \tilde{P}(b'|b,a) \left[\tilde{R}(b,a) + \gamma \sum_{a'} \pi(a'|b') q(b',a') \right] \tag{2.58}$$

It is critical to note that as the value function in a POMDP setting is defined with respect to the belief, it incorporates by definition information from the entire historical sequences of observations and actions observed and taken by the agent, respectively. Now, even though it is often the case that the observation and state-transition probability functions are unknown, making it impossible to compute belief updates explicitly, this provides a strong theoretical basis from which to argue that value estimates, and therefore the agent's policy, should be conditioned on the entire historical sequence of observations and actions in order to take advantage of all the information contained therein regarding the environment dynamics, and therefore the state of the underlying MDP.

2.4 Transformers and Self-Attention

2.4.1 Transformers

In the seminal paper *Attention Is All You Need* (2017) [11], the authors propose a novel model architecture for sequence modelling and transduction (sequence-to-sequence) tasks focusing on natural language processing (NLP) tasks such as machine translation. The authors note that up until that time, RNN-based architectures had dominated the space. Many state-of-the-art methods such as the one proposed in [61], utilise separate multi-layered LSTM-based encoder and decoder networks which encode entire input sequences as latent context vectors which can then be decoded into output sequences without needing to worry about differences in sequence length or structure. Such encoder-decoder models were further improved with the inclusion of, among other things, attention mechanisms, as with Google Neural Machine Translation System [74] which incorporated a special attention mechanism allowing the decoder to 'focus' on different regions of the source sequence when producing each subsequent element in the output sequence. Despite their success, however, the authors [11] note that RNN-based models suffer from the structural constraint of sequential computation; sequences need to be processed in order, which precludes the use of parallelism[21] for speeding up training time, giving rise to poor scaling.

In seeking a solution to these challenges, the authors [11] developed a novel architecture named the *Transformer* which dispenses with recurrence altogether and instead relies on a simple attention-based approach which is able to process entire sequences in one shot, as opposed to sequentially, in a highly parallelisable fashion, making them far more efficient (and far simpler) than the encoder-decoder models dominating the field at the time.

Each architecture we have considered thus far performs specific functional mappings. MLPs map one-dimensional (1D) input vectors to one-dimensional output vectors. In contrast, CNNs (Convolutional Neural Networks) transform grid-like input tensors, such as three-dimensional (3D) image-like data[22], into output tensors of the same dimensionality via a convolutional operation, which typically reduces the size along each axis[23]. Transformers, on the other hand, are functional mappings of the form $f : \mathbb{R}^{N \times L} \to \mathbb{R}^{N \times M}$, designed to process matrices where each row vector is an embedding (a vector representation) of a given feature of

[21] Memory constraints also limit batching across examples.
[22] While they can also process data in arbitrarily high dimensions, such cases are beyond the scope of this dissertation.
[23] Although techniques such as zero-padding can be employed to maintain an output of equivalent size

the input. In the context of NLP the input might be a sentence, in which case the embedding vectors that are processed by the transformer are vector representations of small character sequences, called 'tokens' [24] which together make up the sentence. In the context of computer vision, each embedding vector might be a vector representation of a patch of pixels forming part of an image - more on this in section 2.4.2.

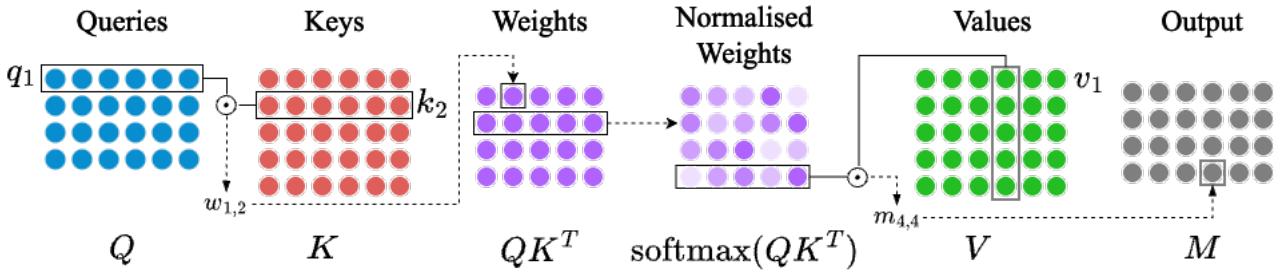


Figure 2.21.: An illustration of dot-product attention [11]. The input consists of Keys, Queries, and Values which are matrices of embedding (row) vectors, each of which represents an element in a sequence or set (e.g. a sequence of tokens - smaller character sequences - which together make up a sentence). A matrix of normalised attention values is computed via matrix multiplication of the Key and Query matrices, followed by a softmax activation function. The final output is produced by multiplying the matrix of attention weights with the Values matrix; ultimately a weighted sum over the components of each embedding vector in the Values matrix.

The Transformer architecture comprises an encoder and a decoder network, each of which is comprised of attention blocks, the core of which is a special multi-head attention layer and which also includes dense and normalisation layers - more on these below. The multi-head attention layer itself comprises multiple attention heads, each of which performs a special attention operation, termed "scaled dot-product attention" by the authors, in order to transform sets of elements encoded as embedding vectors in the form of query, key, and value, matrices into an output representation which captures the global interactions between each pair of elements in the key and query matrices in an attention matrix which is subsequently used to perform a weighted sum over the components of the vectors in the value matrix. Formally, scaled dot product attention takes as input three matrices - a query Q, a key K, and a value V - consisting of an arbitrary number of embedding vectors (one per row). The query and key matrices have embedding dimension d_k and the value matrix has embedding dimension d_v, then the scaled dot-product attention is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.59}$$

where the scaling factor of $1/d_k$ is to counteract the effect of extremely small gradients arising in the case of large d_k due to the softmax function. Breaking it down step by step, as illustrated in Figure 2.21, scaled-dot product attention 2.59 first computes a matrix of attention weights, where the weight in position (i, j) is the dot product between the i^{th} query vector and the j^{th} key vector, which is then normalised by computing a row-wise softmax such that each row in the resulting matrix sums to 1. Finally, the value in each position (i, j) of the output matrix M is computed as a weighted sum over the j^{th} column of V with the weights coming from the i^{th} row of the matrix of normalised attention weights. In this way, the components of the value vectors which are weighted most in each sum correspond to the largest attention values computed between pairs of keys and query vectors. In the context of sequence modelling, scaled dot-product attention has the desirable property of concurrent pair-wise computation over all elements in a single shot, which resolves the issue of sequential computation.

In sequence modelling and translation tasks, a special version of scaled dot-product attention is implemented called *self-attention* wherein the query, key, and value matrices are all linear projections of a single matrix of embedding vectors, $E \in \mathbb{R}^{N \times L}$, where N is the sequence length and L is the embedding dimension. The key, query, and value matrices are produced via matrix multiplication with learnable weight matrices

[24] These may be whole words or commonly occurring parts of words such as "ing".

$$Q = EW^q$$
$$K = EW^k$$
$$V = EW^v$$

where each weight matrix is an element in $\mathbb{R}^{L \times d_{model}}$, projecting the input embeddings into an embedding space with dimension d_{model}.

To provide an intuitive explanation of the attention mechanism in the Transformer architecture, let's first consider the problem it tries to solve. In sequence modelling tasks, such as language translation, it's crucial to understand not just the individual words but also the context in which they appear. Earlier models, like RNNs and LSTMs, process sequences step by step, which can make it difficult to maintain context over long distances within the text. The attention mechanism, specifically the scaled dot-product attention used in Transformers, addresses this by allowing the model to focus on different parts of the sequence when processing each word. Imagine you're reading a complex sentence, and you come across a pronoun like "she." To understand who "she" refers to, you might need to look back at earlier parts of the sentence to find the relevant noun. This is similar to what the attention mechanism does—it lets the model look at the entire sequence and decide which parts are most relevant to understand at each step.

The scaled dot-product attention operates by taking three sets of vectors: queries (Q), keys (K), and values (V). The queries represent the current word the model is trying to understand, the keys correspond to all the words in the sequence, and the values are the actual content of those words. The mechanism computes attention scores by taking the dot product of the query with all the keys. These scores determine how much focus to put on each value when constructing the output representation of the query. The "scaled" part of the scaled dot-product attention comes from dividing the dot products by the square root of the dimension of the key vectors ($\sqrt{d_k}$). This scaling helps prevent the softmax function, which is applied next, from having extremely small gradients when the dimensionality is high. Small gradients can slow down learning and make the model less effective.

After scaling, a softmax function is applied to convert the attention scores into probabilities, ensuring that they all add up to one. This step is like deciding how much of your attention to give to each part of the sentence when figuring out the meaning of "she." The model then uses these probabilities to create a weighted sum of the value vectors, which forms the final output for the current word.

The power of this attention mechanism lies in its ability to capture global dependencies within the sequence, regardless of their distance from each other. Unlike previous models that process sequences in order, the Transformer can handle all the words at once, allowing it to consider the entire context simultaneously. This parallel processing not only makes the Transformer more efficient but also gives it a remarkable ability to understand complex relationships within the data, making it a revolutionary tool for sequence modelling tasks.

One important thing to note regarding self-attention for our purposes is that, due to the properties of matrix multiplication, the dimensions of the weight matrices do not depend on the number of embedding (row) vectors in E, meaning that the same self-attention layer can process, for example, sequences of arbitrary lengths. Crucially, notice that this is not the case for dense layers, which have the form $\mathbf{W}\mathbf{x} + \mathbf{b}$, and wherein each column of the weight matrix corresponds to a single feature (a scalar value) in the input vector, $\mathbf{x}$. In the case of self-attention however, K, Q, and V are produced by taking the dot product between every column in each projection matrix and every embedding vector - every feature - in E. In summary, in the dense layer, there is a 1-1 relationship between the columns in the weight matrix and the features in the input vector, meaning that dense layers require inputs with a fixed number of features which is not the case with dot-product attention.

A second property of dot-product attention worth noting is that of *permutation equivariance*. That is, for a given permutation, permuting the rows in E and then performing self-attention is equivalent to performing self-attention and permuting the rows of the output matrix, which can be illustrated by:

$$\varphi(QK^T)V = \varphi\left(\begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix} \begin{bmatrix} k_1 & k_2 & k_3 \end{bmatrix}\right) \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \tag{2.60}$$

$$= \varphi\left(\begin{bmatrix} q_1k_1 & q_1k_2 & q_1k_3 \\ q_2k_1 & q_2k_2 & q_2k_3 \\ q_3k_1 & q_3k_2 & q_3k_3 \end{bmatrix}\right) \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \tag{2.61}$$

$$= \begin{bmatrix} \varphi(q_1k_1)v_1 + \varphi(q_1k_2)v_2 + \varphi(q_1k_3)v_3 \\ \varphi(q_2k_1)v_1 + \varphi(q_2k_2)v_2 + \varphi(q_2k_3)v_3 \\ \varphi(q_3k_1)v_1 + \varphi(q_3k_2)v_2 + \varphi(q_3k_3)v_3 \end{bmatrix} \tag{2.62}$$

where $q_i/k_i/v_i$ represent the i^{th} row vectors in each respective matrix, and φ represents the row-wise softmax operation. Permuting the rows of each matrix by $[1, 2, 3] \rightarrow [3, 1, 2]$ (by swapping indices) and reordering the summations yields:

$$\begin{bmatrix} \varphi(q_3k_1)v_1 + \varphi(q_3k_2)v_2 + \varphi(q_3k_3)v_3 \\ \varphi(q_1k_1)v_1 + \varphi(q_1k_2)v_2 + \varphi(q_1k_3)v_3 \\ \varphi(q_2k_1)v_1 + \varphi(q_2k_2)v_2 + \varphi(q_2k_3)v_3 \end{bmatrix} \tag{2.63}$$

which is equivalent to the matrix given by 2.62 with the rows permuted in the same way since the order of summation doesn't matter. The fact that the values in each row of the output matrix are not affected by a reordering of the rows of E (as opposed to, for example, the ordering of elements in a vector being passed through a dense layer) arises from the fact that there is no information inherent in the operations comprising scaled dot-product attention regarding the relative positioning of the rows of E. Note that RNNs do not suffer from this issue since, by definition, they process elements sequentially. To overcome this problem in the case of tasks involving sequential data, the authors add fixed positional encoding vectors to the input embeddings in the form of a positional encoding matrix Pos, where each row encodes the relative position of the corresponding element in the input sequence using sine and cosine functions of different frequencies. Formally, the value for the j^{th} component of the positional encoding vector in Pos corresponding to the i^{th} element in the sequence is given by:

$$Pos_{i,j} = \begin{cases} sin(\omega_k \cdot i) & \text{if } j = 2k \\ cos(\omega_k \cdot i) & \text{if } j = 2k + 1 \end{cases}$$

where

$$\omega_k = \frac{1}{10000^{2k/d_{model}}}$$

that is, the values for the components of each positional encoding vector are computed by interleaving sine and cosine functions, the input of which is scaled by the integer value of the position of the element in the sequence. In this way, the wavelengths across all sequence positions form a geometric progression from 2π to $10000 \cdot 2\pi$ such that the inner product of a positional encoding vector at position i and a second at position $i + k$ grows proportionally to the size of k (elements closer together in the sequence have a larger dot product than those further away). It is worth noting that increasing d_{model} allows for higher fidelity positional encoding which is useful for longer sequences.

Returning to the concept of multi-head attention, the authors found that instead of performing a single attention function (equation 2.59) with d_{model}-dimensional key, value, and query vectors it was beneficial to linearly project the queries, keys, and values h times with distinct learnable weight matrices to dimensions d_k, in the case of the queries and keys, and d_v, in the case of the values. Then, performing the attention function in parallel for each of the h query, key, and value matrices, the h matrices produced from all operations are finally concatenated and projected into an output space with a learnable weight matrix, W^O:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(head_1, \ldots, head_h)W^O \tag{2.64}$$

$$\text{where } head_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{2.65}$$

where $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$, and $W_i^O \in \mathbb{R}^{hd_v \times d_{model}}$. In their experiments, the authors use $h = 8$ attention heads with $d_{model} = 512$, where each head has $d_k = d_v = d_{model}/h = 64$, as an example. The authors found that multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions, whereas this was inhibited by the weighted averaging computed by the row-wise softmax operation.

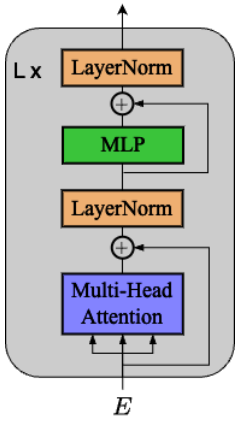


Figure 2.22.: An illustration of the attention block, which is a core component of the Transformer architecture [11]. Multi-head attention transforms an input matrix of embedding vectors, where each vector corresponds to an element in an input sequence or set, by computing pair-wise scaled dot-product attention between all embedding vectors. The multi-head attention operation is followed by a residual connection (an element-wise sum with the input matrix) and a LayerNorm layer. Finally, an MLP performs a row-wise transformation of the output matrix, followed by a final residual connection and a LayerNorm layer.

While many of the details of the complete Transformer architecture are not relevant for our purposes, there are two final components of the Transformer architecture which are worth mentioning which will allow for a complete picture of the attention block, the core component of the Transformer encoder, illustrated in Figure 2.22. First, an MLP follows each multi-head attention layer in order to compute intermediate, independent transformations of the embeddings such that each embedding vector is transformed in isolation. Second, each multi-head attention layer and each MLP is followed by a residual connection and a *Layer Normalisation* layer [75] which performs a normalisation operation over the values of each sample input in a batch and has been shown to stabilise and accelerate the training of ANNs via gradient descent. Suppose we have a batch of N inputs $B = \{x_1, \ldots, x_N\}$ where each $x_i \in B$ has K components (it could be a vector or a tensor of arbitrary shape). We first compute the sample mean μ_i and variance σ_i^2 for each sample in the batch:

$$\mu_i = \frac{1}{K} \sum_{k=1}^{K} x_{i,k}$$

$$\sigma_i^2 = \frac{1}{K} \sum_{k=1}^{K} (x_{i,k} - \mu_i)^2$$

Then, we normalize each sample such that the components in the sample have zero mean and unit variance:

$$\hat{x}_{i,k} = \frac{x_{i,k} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

where $\epsilon \ll 1$ is for numerical stability. To allow the model flexibility, there is a final scaling and shifting step performed using learnable parameters η and β as:

$$\text{LayerNorm}_{\eta,\beta}(\hat{x}_{i,k}) = \hat{x}_{i,k}\eta + \beta$$

Results reported by the authors in [11] demonstrate the superiority of the transformer in sequence modelling and sequence transduction tasks, such as machine translation, over the state-of-the-art RNN-based models at the time. Presently, Transformer-based architectures have largely replaced RNN-based models as the defacto architecture for sequence-related NLP tasks [21] [12] and their superiority has been confirmed independently on multiple benchmarks [76]. It should be noted, however, that due to the nature of Transformer-like attention mechanisms, architectures in the broader Transformer family have been shown to perform poorly on auto-regressive time series modelling tasks [77].

2.4.2 Extending Transformers To Vision-Based Tasks

While *transfer learning* - techniques for improving a model's performance in a target domain by pre-training the model in a related but different source domain - was an active field of research prior to the Transformer [78], its inception sparked a wave of research into so-called *foundation models* [79] - massive[25] generalist models pre-trained on web-scale data sets - including BERT (Bidirectional Encoder Representations from Transformers) [80] and GPT (Generative Pre-Trained Transformer) [81] [82] [83], each of which achieved state-of-the-art performance in the domain of NLP.

The significant success of Transformers in the domain of NLP naturally prompted research into the application of Transformer-like models in computer vision. To this end, the Vision Transformer (ViT) was proposed in *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale* [12] in 2021. In it, the authors aim to address the shortcomings of prior related research, which had applied the Transformer architecture in conjunction with CNNs or only replaced certain components of CNNs while keeping their overall structure in place, in order to improve on performance in computer vision tasks such as object recognition.

The ViT architecture, illustrated in Figure 2.23, is made up of three major components: an embedding layer for transforming images into a matrix of embedding vectors; a network core which consists of a sequence of "attention blocks" nearly identical in design to the encoder of the original Transformer, and a classification head in the form of an MLP which produces a vector of probabilities over possible target classes for the purpose of performing image recognition (classifying images as one of a fixed number of classes).

Image embeddings are produced in the following way. First, the image is segmented along the height and width dimensions into a grid of square patches of equal size. Each patch is then flattened into a single, one-dimensional vector and projected into a common embedding space of dimension d_{model} by a shared dense layer. In addition to the patch embeddings, the authors include a learnable $[class]$ embedding vector which occupies the first row of the embedding matrix, E. As with the original Transformer architecture, positional encoding allows the model to determine the relative position of features represented by the embedding vectors in E; the authors experiment with various methods of positional encoding, with the default across experiments being to maintain a set of learnable positional encoding vectors [26] - one for each patch - which are optimised jointly with the model parameters along with the $[class]$ embedding vector, and which are added element-wise to their corresponding patch embedding vectors before being passed into the network core.

As mentioned above, the network core takes the form of a Transformer-like encoder consisting of several attention blocks, each of which consists of a multi-head attention (MHA) layer and an MLP which maps a set of input vectors to a set of output vectors of the same number identical to the Transformer. Like the Transformer encoder, layer normalisation layers are included before the MHA and MLP components, each followed by a residual connection to facilitate stable gradient-based optimisation for very deep networks with several attention blocks. In the ViT, each MHA layer performs self-attention where the input to the layer is used to produce the keys, queries, and values. In this way, MHA in the ViT facilitates the learning of the global relationships between all patches in the image from the very first layer. This stands in contrast to the

[25] With respect to number of parameters and network depth.

[26] As opposed to the fixed positional encoding vectors used in the original Transformer.

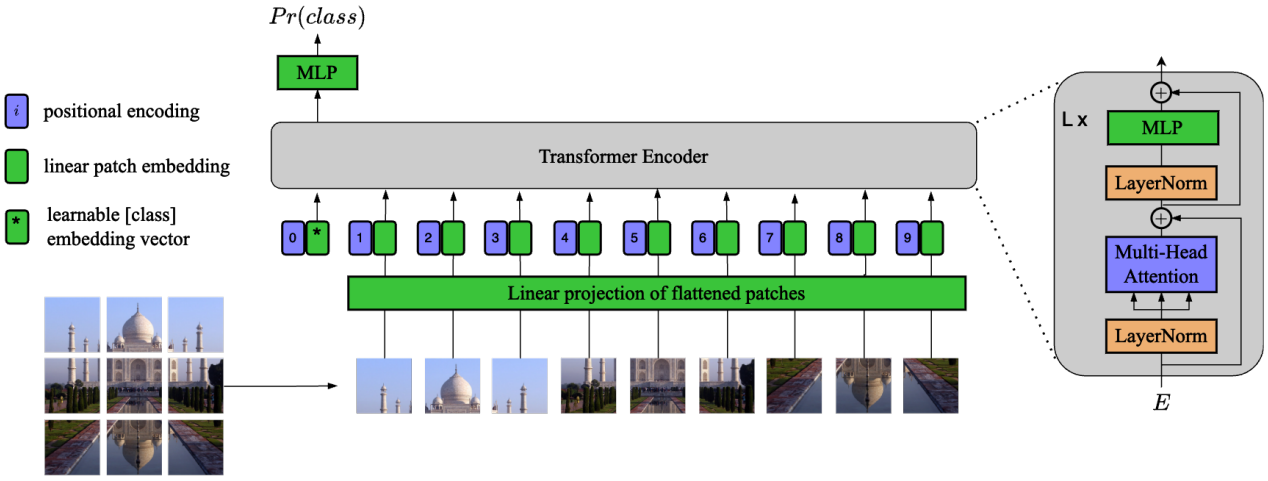


Figure 2.23.: The Vision Transformer architecture for image classification [12]. The input RGB image, a 3-dimensional tensor of pixel values, is segmented into square patches of equal size along the height and width dimensions. The patches are then flattened into 1-dimensional arrays and projected into a common embedding space using a shared dense layer. Positional encoding vectors, which encode the relative horizontal and vertical position of each patch in the grid, are added to each patch embedding vector element-wise. The embedding vectors, along with a special $[class]$ vector of learnable parameters, are stacked row-wise to form a matrix of embedding vectors E which is passed into a sequence of L attention blocks. Finally, the first row of E (the row corresponding to the $[class]$ vector) is passed through a final MLP in order to produce a vector of probabilities over possible classes, with which image classification may be performed.

CNN where interactions between features in different parts of an image might only be modelled by the latter layers in the network due to the nature of convolutional layers.

In order to produce class probabilities, the row vector corresponding to the $[class]$ embedding vector in the matrix produced by the ViT encoder core is passed through the classification head; a small MLP which maps input vectors to vectors with a dimension equal to the number of possible classes.

Convolutions with a shared set of filters, in the case of CNNs, and self-attention using embeddings of flattened patches, in the case of ViTs, constitute very different approaches to the task of extracting meaningful features from images. Crucially, ViTs do not have the same spatial inductive bias inherent in the architecture of the CNN and must rely entirely on the linear embeddings and the positional encodings in order to learn the spatial relationship between pixels and patches, which it must do in order to successfully classify visual objects with inherent spatial structure. The authors hypothesise that as Transformers have been shown to excel when trained at scale (large models trained on very large data sets) in the domain of NLP, the same might be found in the domain of computer vision.

In order to test this hypothesis, the authors perform pre-training with a set of ViT and state-of-the-art ResNet (CNN-based) [32] architectures, with varying numbers of layers and parameters, on data sets of different sizes, evaluating them on well-known image classification benchmark tasks, such as ImageNet. Based on their results, the authors made a number of interesting observations. First, ViTs perform worse than ResNets when trained on smaller data sets ($\mathcal{O}(1M)$ images), but reliably improve, at a faster rate than ResNets, as the size of the data set grows, excelling in the case of very large data sets ($\mathcal{O}(100M)$ images). Second, while the ResNets' performance plateaued above $\sim 30M$ input examples, the larger ViTs continued to improve reliably up until $\sim 100M$ with the top-performing ViT beating the top-performing ResNet. Finally, when trained with a fixed computational budget, ViTs were shown to outperform ResNets across the board, but especially for lower budgets.

What is the theoretical relationship between ViTs and CNNs? Can we be sure that Self-Attention layers are at least as expressive as Convolutional layers with respect to visual feature learning and representation? In [84] the authors examine this question both theoretically and experimentally to ascertain whether self-

attention is a viable alternative to convolutional layers with respect to visual feature learning, ultimately arriving at a proof for the following theorem:

Theorem 3 (The Relationship Between Self-Attention and Convolutional Layers)**.** A multi-head self-attention layer with h heads of dimension d_k, output dimension d_{model}, and a relative positional encoding of a dimension ≥ 3 can express any convolutional layer with filter size $\sqrt{h} \times \sqrt{h}$ and $min(d_k, d_{model})$ output channels.

This implies that in order to have a theoretical guarantee that a given self-attention layer is able to express a given convolutional layer, the number of attention heads must be proportional to the size of the convolutional filter, and the smallest of the embedding dimensions must be at least the same as the number of output channels (the number of convolutional filters).

Experiments conducted by the authors [84] yielded two observations worth noting. First, they observed that fully attentional models appear to learn in a manner which appears to be a generalisation of CNNs where the convolutional pattern is learned at the same time as the filter weights. Secondly, when trained to perform image classification on the CIFAR-10 data set (just 60K images) compared to a ResNet baseline, the authors find that their self-attention-based model (note: not a ViT) takes longer to converge than the ResNet - the authors are inconclusive as to the reason for this, but one possible cause might be that the MHA layers need to learn the spatial inductive bias inherent in convolutional layers, and that this lag is more evident for smaller models trained on smaller data sets than in the case of the full ViT.

An additional finding regarding the comparison between CNNs and ViTs worth noting comes from a paper titled *"ConvNets Match Vision Transformers at Scale"* [85], published by Google DeepMind in October 2023. In the paper, the authors challenge what they say is a widely held belief among researchers that "ConvNets perform well on small or moderately sized data sets, but are not competitive with Vision Transformers when given access to data sets on the web-scale". To this end, the authors compare ViTs to state-of-the-art fully-convolutional architectures on a massive image classification data set (4B images) under comparable computational budgets and conclude that "Although the success of ViTs in computer vision is extremely impressive, in our view there is no strong evidence to suggest that pre-trained ViTs outperform pre-trained ConvNets when evaluated fairly" and that what matters most, given a sensibly designed model, is the compute and training data available.

2.4.3 Masked Auto-Encoders: Transformers & Partial Observability

A major challenge in efforts to make progress in training large models in a supervised manner is the ever-increasing need for more labelled data, with the main issue being that manually labelling data (e.g. a human being classifying images one at a time) is prohibitively expensive and time-consuming when done at scale (i.e. for millions/billions of images). This challenge has been addressed in the NLP space by turning to *self-supervised learning* methods, such as those used to train BERT [80] and the GPT family of models [81] [82] [83], where instead of requiring an input, x, and a ground truth output, y, a target is constructed by masking components of the input x - the model is then trained to predict the missing components, thus dispensing with the need for a ground truth label/value altogether. A simple example might be masking the end of a sentence, such as *"The cat sat on the* $[mask]$*"* and training the model to correctly predict the missing word at the end of the sentence; *"mat"*. This method of learning unlocks vast amounts of text data on the internet, enabling the training of massive generative models like the GPT family, which have achieved state-of-the-art performance across the board on NLP benchmarking tasks.

These ideas may be naturally extended to computer vision by training models to reconstruct partially masked images, but research in the area has lagged behind that of NLP, leading researchers to ask the question "What are the main differences between computer vision and NLP when it comes to the task of self-supervised learning via masking?". In their paper titled *Masked Autoencoders Are Scalable Vision Learners* [13] the authors ask this question and offer three observations in response:

1. CNNs were the dominant architecture in computer vision tasks over the last decade and implementing masking in the manner it's done in the NLP context is not straightforward or easy given the convolutional architecture.

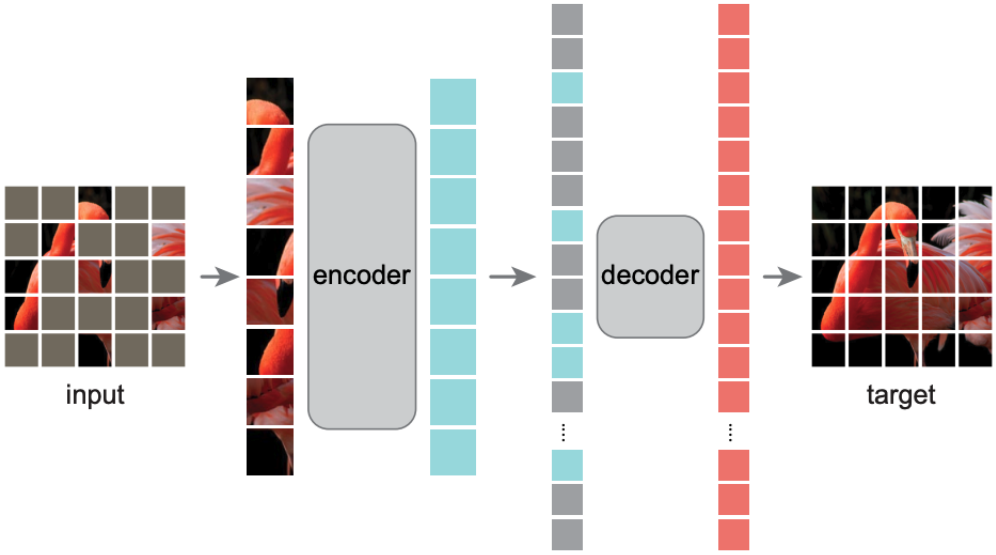


Figure 2.24.: The Masked Autoencoder architecture [13] consists of a large encoder and a lightweight decoder which both take the form of the encoder of the original Transformer architecture, as used in the ViT [12]. The encoder takes as input a matrix of embedding vectors corresponding to the set of unmasked image patches which are sampled uniformly according to a fixed masking ratio (e.g. 75% of patches are masked), producing an intermediate matrix of latent embedding vectors which together encode the visuospatial features in the unmasked patches. Copies of a learnable $[mask]$ vector are added element-wise to the relevant positional encoding vectors and inserted into the matrix output by the decoder at the indices corresponding to the masked patches - the resulting matrix is then fed into the decoder which attempts to predict the values of the pixels in the missing patches. During training, the MAE is able to learn to produce semantically meaningful latent representations of images which may be used for downstream tasks such as image classification.

2. Images are less information-dense than text, that is, a patch of missing pixels might easily be imputed with the values of neighbouring pixels without much loss of meaning, which is not the case with words in sentences.

3. The difference in semantic density between a word and a pixel makes the task of predicting masked elements less straightforward in computer vision than it is in NLP.

In [13] the authors propose a method of self-supervised learning using a novel architecture called a *Masked Auto-Encoder* (MAE) which attempts to address these challenges. At a high level, the MAE, illustrated in Figure 2.24, consists of an encoder and a decoder which both take the form of the Transformer encoder used in the ViT architecture. The encoder takes as input a partially masked image to produce a latent representation of the image and the decoder attempts to reconstruct the image by predicting the values of the missing pixels. In training the MAE to reconstruct the entire image from a partial observation, the model is forced to infer the missing visual features and thereby learn useful latent representations of images which may be used for downstream tasks such as image classification.

We now explore in detail how the MAE is trained. First, as with the ViT, the input image is segmented into a grid of non-overlapping patches of equal size. A large subset of patches is then sampled randomly, in a uniform manner without replacement, and masked (i.e. excluded) - it is worth noting upfront that optimal performance was achieved on downstream tasks at high masking ratios of around 75%. The remaining patches are then embedded via linear projection to produce a matrix of embedding vectors, E. As with the ViT, positional encoding vectors are added to each of the patch embedding vectors in E, however, the positional encoding vectors are fixed throughout training, as opposed to being learnable vectors jointly optimised with the model. The reason for this is that, since the majority of the patches are masked for a given image during each forward

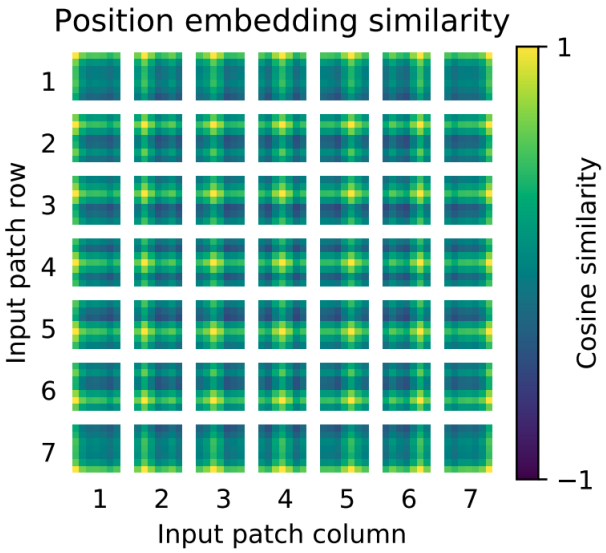


Figure 2.25.: An illustration of the cosine similarities between the fixed positional encoding vectors used for encoding the relative positions of image patches when training the MAE [13]. The heat map at position (i, j) in the grid illustrates the cosine similarity between the positional encoding vector at row i and column j in the grid of image patches and all other positional encoding vectors - the euclidean distance between patches in the grid is (approximately) proportional to the cosine similarity between their associated positional encoding vectors.

pass through the model, the majority of the positional encoding vectors would only be subject to optimisation in a small percentage of the optimisation steps if they were learnable, thus badly impacting the efficiency of the learning process. As in the case of the original Transformer, each positional encoding vector is computed using sinusoids wherein the frequency is a function of a positional index of the given image patch, with the exception that the first half of the vector is dedicated to the vertical position and the second half to the horizontal position of the patch, yielding a relative positional encoding in two dimensions. An illustration of the cosine similarity between the fixed positional encoding vectors on the plane is shown in Figure 2.25.

Following the addition of the positional encoding vectors, the embedding matrix is processed by the MAE encoder in the usual way, producing a matrix with the same number of row vectors as E. Now, the task of the decoder is to use this latent representation of the unmasked patches, which contain information about visual features at specific spatial locations, to predict the values of the missing pixels in each of the masked patches. To this end, copies of a special learnable $[mask]$ vector are added to the positional encoding vectors associated with the positions of the masked patches and inserted into the matrix of latent vectors output by the encoder before being passed through the decoder – this communicates to the model for which specific spatial locations it should predict missing pixel values. It is worth noting that, due to the high masking ratios under which the MAE is trained, the encoder-decoder design is asymmetrical, with the encoder having significantly more attention blocks than the lightweight decoder, as it only has to process a fraction of the total number of embedding vectors when producing the intermediate latent representation of the masked images, whereas the encoder must process the full set of embedding vectors covering all patches in the image. The final MLP of the final attention block of the decoder produces pixel predictions in the form of embedding vectors of the same dimension as the flattened patches in order to compute a mean squared error loss over the pixels in the masked patches, which is used to optimise the entire model via gradient descent.

The results reported by the authors showed (to their surprise) that the MAE, trained in the self-supervised manner described above on the ImageNet-1K data set, was able to reconstruct the input images under high masking ratios with surprising accuracy - the MAE's reconstruction of images under various masking ratios is illustrated in Figure 2.26. Furthermore, the authors found that the model excelled under higher masking ratios, hypothesizing that the reason for this is because - per point 2 above regarding the information density of pixels versus words - a high degree of sparsity forces the model to learn meaningful latent representations of images as it is not able to resort to simply imputing missing pixels with the average of the values of pixels

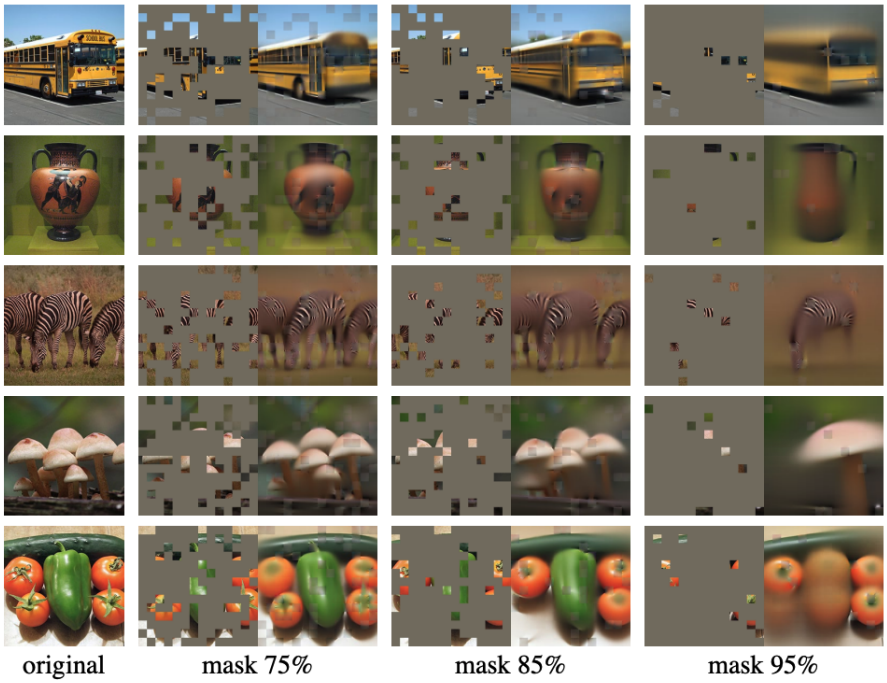


Figure 2.26.: An illustration of the trained MAE predicting missing pixel values belonging to the masked image patches of images in a held-out test set. The original images (the ground truth) are shown in the left-most column and each column to the right shows the unmasked patches alongside an image with the predicted pixel values filled in for different masking ratios. As is evident, the model is able reconstruct the input image to a surprising degree of accuracy even under very high masking ratios.

in neighbouring (unmasked) patches in order to minimise the loss function.

The authors tested the pre-trained MAE on image classification tasks by two different methods; linear probing and fine-tuning. In both cases, only the MAE encoder is used and a dense layer is appended to it, which produces class probabilities. In the case of linear probing, the encoder weights are frozen and only the dense layer is optimised, whereas in fine-tuning the entire encoder is optimised along with the dense layer. The authors optimised the pre-trained MAE using both methods under various masking ratios, the results of which are illustrated in Figure 2.27. The authors found that linear probing performed best under high masking ratios of around 75%, achieving a top accuracy of 73.5%, whereas fine-tuning demonstrated strong performance over a large range of masking ratios from 40% to 80%, achieving a top accuracy of 85%. It is worth noting for our purposes that the authors experimented with two different approaches to producing the input vector which was fed into the dense (classification) layer - appending a learnable $[class]$ embedding vector to the input embedding matrix, as per the ViT approach, and simply averaging over all the latent vectors produced by the encoder - no significant difference in performance was found between the two approaches.

The methods and results reported in [13] are highly relevant for this work, as they demonstrate that transformers not only provide an architecture which is able to process variable-sized inputs (i.e. different numbers of embedding vectors), but excel (in the case of vision-based tasks) under conditions of partial observability in the form of high masking ratios.

2.5 Chapter Conclusion

In conclusion, this Background chapter has laid a comprehensive foundation in the essential topics of RL and ANNs, including foundational RL theory in cases of both full and partial observability, ANNs as function approximators, conventional ANN architectures, and the necessary deep RL algorithms required to appreciate the remainder of this dissertation. Crucially, we have also provided a detailed overview of Transformer-like

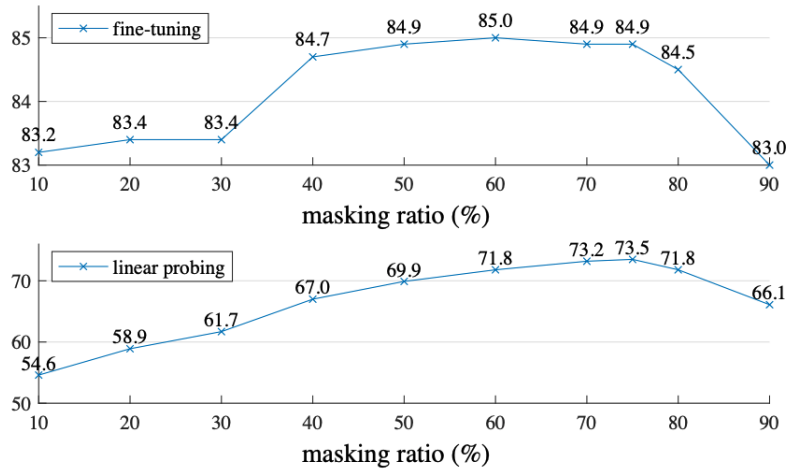


Figure 2.27.: An illustration of the accuracy obtained by the MAE encoder [13] adapted to the task of image classification using fine-tuning (task-specific optimisation of a dense classification layer along with the entire model) and linear probing (task-specific optimisation of a dense classification layer only). Note that high masking ratios yield the best performance, especially in the case of linear probing.

attention, the application of Transformers to vision-based tasks, and MAEs which underscore the proficiency of ViTs — and by extension, Transformer-like attention — in learning to produce semantically meaningful embedding vectors of images under high masking ratios.

In the forthcoming chapter, we will delve into pertinent research concerning the use of RNNs to handle partial observability in RL. We will examine instances where RNNs and alternative attention mechanisms have been employed to enhance interpretability and, critically, investigate the application of Transformer-like attention within RL policy architectures, emphasising research that studies such architectures under conditions of partial observability.

Chapter 3

Literature Review

In this chapter, we discuss the prominent research that informs our work. In section 3.1, we examine the use of RNNs as a memory mechanism to address partial observability in the context of RL, a concept integral to our proposed attention-based architecture. In section 3.2, we explore the use of attention mechanisms, both Transformer-like and non-Transformer-like, to enhance interpretability, which is particularly relevant to the task of optimal sensory querying in the 'low-bandwidth communication channel' problem outlined in Section 1.2.2.

The main focus of this chapter, covered in section 3.3, is the application of Transformer-like attention in RL policy architectures, especially in instances where it has been used to tackle the challenge of partial observability. We cover three research sub-areas where Transformer-like attention has been employed in the context of RL: (a) as a memory mechanism akin to RNNs, (b) for one-shot processing of sequences of partial observations, and (c) for attending to either the input space directly or its abstract representation, often in conjunction with RNNs or methods such as frame stacking to integrate information over time. Our discussion will emphasise how our research not only addresses existing gaps but also advances the current body of literature.

3.1 RNNs As Memory Under Partial Observability

In order to overcome the challenges presented by partial observability, we have shown how an agent may form a Belief MDP by acting on the basis of a belief, $b(s)$ - a probability distribution over possible states - which is updated at each time step in a manner which aggregates observation and action information over time. A restrictive assumption with this strategy (see the belief update rule 2.57) is that the state transition probability function, P, and the observation probability function, $\mathcal{O}$, are known (as they are in the field of dynamic programming). However, in many cases, such as in the case of a video game, such functions are defined implicitly in the environment and as such are not available to the agent. Additionally, if the state space is very high-dimensional, such as a large pixel grid, maintaining a probability distribution over all states becomes computationally and statistically impractical.

In general, the canonical approach to handling POMDP environments in the context of deep RL has been to adapt the agents' policy/value network architectures by incorporating a type of memory mechanism, giving the agent the ability to 'remember', aggregating information from historical partial observations (and actions) on which to condition the agent's policy. The most widely used memory mechanism used to address partial observability in deep RL has been the RNN (in particular, the LSTM) [21] which maintains a hidden state which is akin to a belief in the sense that it aggregates information over time (e.g. observations and actions), except that the RNN, along with the rest of the ANN, is optimised with respect to the expected return obtained by the agent. In this section, we explore key research regarding the use of RNNs for addressing partial observability in deep RL.

3.1.1 Static Environments

Human beings process large amounts of visual information when deciding how to act in the world. However, as we are unable to capture and process all of the visual information in our immediate surroundings at a given time, we must aggregate visual information taken from small subsets of our surroundings over time in order to make decisions. In *Recurrent Models of Visual Attention* (2014) [14] the authors address the issue of efficient information extraction (e.g. image classification) from increasingly large images, making the observation that the size of an ANN (in numbers of parameters) required for processing an entire image in a single forward pass grows proportionally with the size image, and therefore so does the amount of computation. The authors propose a possible solution to this problem, using the case of image classification as an example: instead of classifying the entire image in a single forward pass, they propose converting the problem into a reinforcement learning task where the agent is allowed to 'query' fixed sized patches (where each patch is a square region over the pixel grid) of the image at each time step, using an RNN-based policy in order to aggregate information across time and ultimately make a final classification decision. In this way, the authors argue, the size of the network, and therefore the computation required, can be controlled independently of the size of the input image.

In order to evaluate their proposed model, the authors test it on a number of image classification tasks involving images of handwritten digits from the MNIST dataset: centred digits; translated digits, where the digits were positioned off centre, testing the model's ability to iteratively locate salient information, and; cluttered images where the digits are surrounded by visual clutter (noise) in order to test the model's ability to focus on salient information in the image whilst ignoring noise. During each individual classification, the image in question remains static and the agent is able to 'query' different parts of the image in order to make a classification decision after a fixed number of time steps.

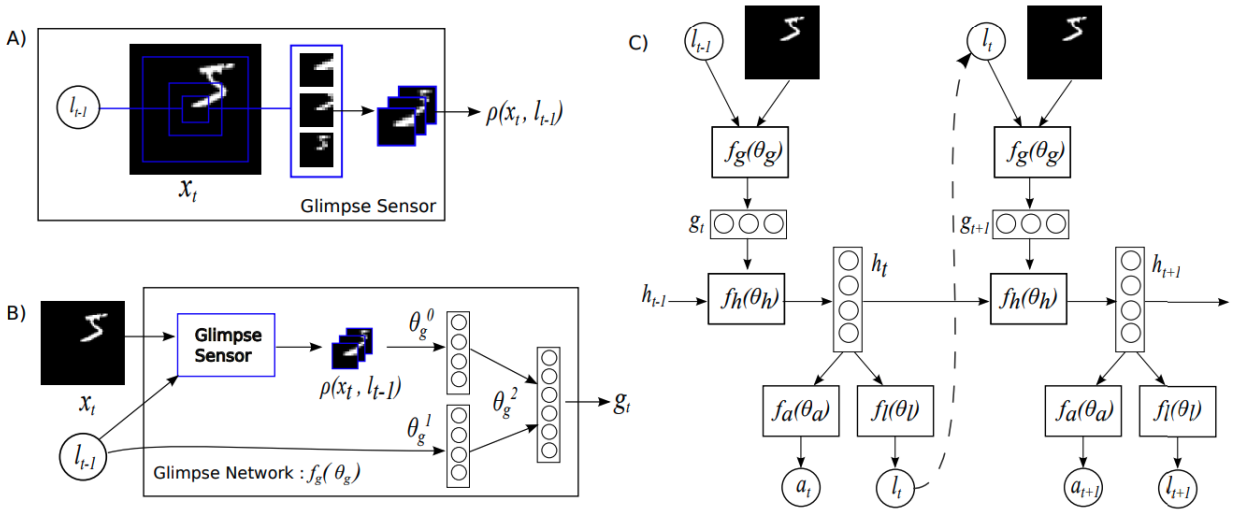


Figure 3.1.: The RNN-based policy architecture from the paper *Recurrent Models of Visual Attention*[14]. **(A)** The so-called 'glimpse sensor' which extracts a retina-like glimpse centred at l_{t-1} comprising of concentric patches, resized to be of the same dimensions, giving a low-fidelity view of the more global region and higher-fidelity views of more local regions **(B)** the glimpse $\rho(x, l_{t-1})$ and location vector l_{t-1} are passed through f_g to produce a joint embedding vector g_t **(C)** A view of the entire policy architecture - the RNN core f_h retains a hidden state h_t which aggregates past information and present information via h_{t-1} and g_t in order to produce a joint action $u_t = \{a_t, l_t\}$ at each time step.

The proposed network architecture - illustrated in Figure 3.1 - has three main components (which the authors treat as sub-networks): A glimpse network, f_g, for performing feature extraction, an RNN core, f_h, for aggregating information across time, and two actor heads: f_a, for making classification decisions, and f_l, for deciding which region of the pixel grid to query.

Given an input image, x, classification decisions are reached over a number of time steps. At each time step, $t = 0, 1, \ldots, T$, the x-y coordinate vector, l_{t-1}[1], emitted from f_l at $t-1$ is used to extract a retina-like 'glimpse', $\rho(x, l_{t-1})$, consisting of a number of concentric patches centred at l_{t-1} - each significantly smaller than the entire image - resized to be of equal dimension so as to give a low-fidelity view of the region covered by the largest patch and a higher fidelity view for regions covered by smaller patches contained therein.

Both the glimpse, $\rho(x, l_{t-1})$, and the coordinate vector, l_{t-1}, are transformed by separate linear layers in the glimpse network f_g, whereafter the resultant vectors are concatenated and passed through a final linear layer in order to combine information from both to produce g_t - in this way the agent has information regarding both 'what' (from the pixel information) and 'where' (from the coordinate information). Then, g_t and h_{t-1}, the hidden state from the previous time step, are passed through f_h which updates its hidden state to produce h_t, which aggregates information from past observations and locations with information encoded in g_t.

Finally, h_t is passed through f_a and f_l to produce a joint action, $u_t = \{a_t, l_t\}$, where $a_t \sim p(\cdot|f_a(h_t))$ and $l_t \sim p(\cdot|f_l(h_t))$ are drawn from distributions parameterised by $f_a(h_t)$ (e.g. a softmax for classification tasks) and $f_l(h_t)$ (e.g. a bivariate normal distribution with constant variance), respectively. After T time steps, the agent makes a classification decision, obtaining a reward, $r_T = 1$, for correct classification and 0 otherwise. The full network, f_θ, constitutes a decision-making policy, $\pi_\theta(u_t|\xi_{1:t}) = \pi(\{a, l\}|\xi_{1:t})$, which the authors state may be optimised via policy gradient methods. The objective function used for optimising the policy network, equivalent to 2.51, has the form:

$$J(\theta) \approx \frac{1}{N} \sum_{j=1}^{N} \left[\sum_{t=1}^{T} log \pi_\theta(u_t|\xi_{1:t}) \left(G_t - b_t\right) \right]$$

where $\xi_{1:t}$ represents the information aggregated, for example via h_t, over the interaction sequence

$$l_0, \rho(x, l_0), l_1, \rho(x, l_1), \ldots, l_{t-1}, \rho(x, l_{t-1})$$

emitted and observed by the agent respectively over the course of a given trajectory up to time t. The quantity $\pi_\theta(u_t|\xi_{1:t})$ in the objective function is taken as the joint probability of sampling action a_t and location l_t, although the authors do not specify the details of how this should be computed.

The authors note that at the time of publication, CNNs were the dominant method for image classification in the field. As such the authors compare their novel RNN-based method to a regular CNN baseline, which was trained to classify images in one shot, in the usual way. The results demonstrated that their RNN-based method performed well across the board, with its ability to focus on specific local regions of an image enabling it to significantly outperform the CNN baseline on the cluttered image task.

In addition to the static digit classification task, the authors additionally tested their method on a basic dynamic task, which is nearly identical to the 'catch' task illustrated in Figure 2.3, where in order to catch the ball the agent must track it over time by querying the associated locations on the game screen [2]. The results demonstrated that the agent was able to successfully learn to perform the task, catching the ball roughly 85% of the time.

To our knowledge, this research was the first to demonstrate agent learning under conditions of partial observability using an RNN-based policy. It is important to note two things regarding this work for the purposes of this dissertation: first, due to the policy network architecture the glimpse observed by the agent must necessarily be of fixed size and may not be expanded or contracted during either training or inference and, second, the agent is only allowed to observe a single glimpse from a single local region at a given time.

A similar method to [14] was employed in *In Show, Attend and Tell: Neural Image Caption Generation with Visual Attention* (2015) [15] where the authors used an RNN-based network and policy-gradient-style optimisation in order to iteratively produce image captions based on partial observations gleaned from a static image. Given an image, x, the model proposed by the authors produces a sequence of vectors, $y = \{y_1, \ldots, y_C\}, y_i \in \mathbb{R}^K$, each of which encodes a word in the sequence making up the predicted image caption.

[1] l for "location"

[2] `http://www.cs.toronto.edu/Ëœvmnih/docs/attention.mov`

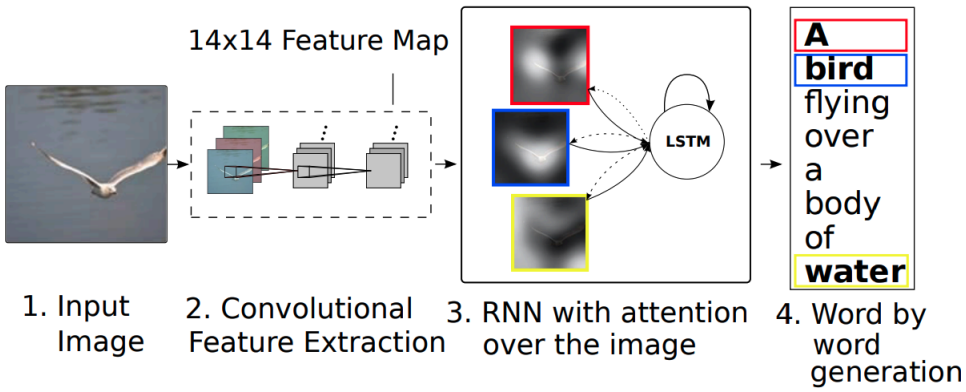


Figure 3.2.: An illustration of the caption generation model proposed in [15]. A CNN extracts a feature map from an input image comprising a set of feature vectors which encode visual features at the corresponding spatial locations in the input image. An LSTM controller then iteratively samples feature vectors, which may be thought of as partial observations of the static image, in order to generate a probability distribution constituting a stochastic policy over possible next words.

The model proposed by the author is composed of two components, as illustrated in Figure 3.2; a CNN feature extractor which produces a feature map which encodes spatial features extracted from the image, followed by an LSTM which iteratively attends to local regions of the image via the feature map in order to generate the word encoding vectors.

The CNN takes as input an RGB image, $x \in \mathbb{R}^{H \times W \times C}$, in order to produce a set of feature maps, $\tilde{x} \in \mathbb{R}^{\tilde{H} \times \tilde{W} \times \tilde{C}}$, with $\tilde{H} < H$ and $\tilde{W} < W$, comprising of vectors $v = \{v_1, \ldots, v_L\}, v_i \in \mathbb{R}^{\tilde{C}}$, each of which encodes spatial features in a local region of x corresponding to its x-y position in $\tilde{x}$, constituting a partial observation of the image, x.

In order to generate each word vector, the LSTM maintains a hidden state which aggregates data from across the interaction sequence of partial observations and actions (word vectors generated). At each time step, t, a distribution over each v_i is computed using a simple MLP attention network, f_{att}, whose output also depends on the hidden state from the previous time step, followed by a softmax distribution in the following way:

$$e_{it} = f_{att}(v_i, h_{t-1}) \tag{3.1}$$

$$\alpha_{it} = \frac{exp(e_{it})}{\Sigma_{k=1}^{L} exp(e_{it})} \tag{3.2}$$

From 3.2 a feature vector, $\mathbf{v_{it}}$, is sampled, which is passed into the LSTM, along with h_{t-1}, followed by a final output layer in order to produce a probability distribution over possible next words, $p(y_t|y_{t-1}, h_t, v_{it})$, conditioned on the previous word, the updated hidden state, and the sampled feature vector. Altogether, the architecture comprises a stochastic policy which is conditioned on returns which are proportional to the log-likelihood of the target caption being sampled under the given model parameters. The authors observed the model's ability to focus on salient local regions of a static image when producing associated portions of the caption - the method achieved state-of-the-art on a number of image captioning benchmarks.

These early works demonstrated the effectiveness of applying RNNs to POMDP RL problems with static environments in order to aggregate information from local regions in the static state over time for the purposes of optimal decision-making. The two limitations of this work are (i) the limited application to dynamic environments and (ii) the fact that the architectures in each case require the partial observations to be of a fixed size - in the case of [14], for example, it may be beneficial to sample a different number of concentric patches in order to expand or contract the size of the region observed by the agent, perhaps starting off observing a wider region, but focusing entirely on a local region at a later time step. We explore the extension of these methods to dynamic environments below.

3.1.2 Dynamic Environments

Early research conducted into applying LSTMs to solving POMDP RL tasks involving dynamic environments includes using an LSTM (RNN) as both an advantage function approximator [86] and as a parameterised policy [87], trained with policy gradient methods. In both cases the POMDP tasks were low-dimensional and relatively simple, for example, the canonical Cart Pole control task. The paper *Deep Recurrent Q-Learning For Partially Observable MDPs* [16] is, as far as we are aware, the earliest example of RNNs being applied to solve high-dimensional, complex RL tasks, such as the Atari 2600 arcade games.

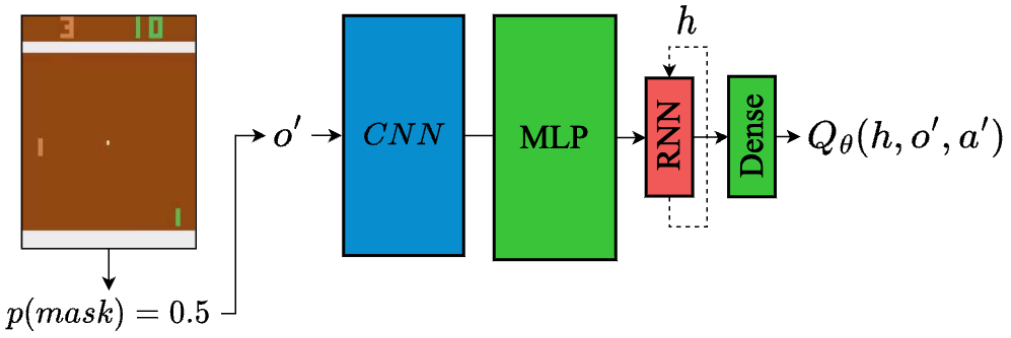


Figure 3.3.: An illustration of the Deep Recurrent Q-Network (DRQN) method being applied to the flickering version of Atari Pong. At each time step the agent receives a single frame of the game screen (an image), o, which is fully obscured (via zero-masking) with $p(mask) = 0.5$. The recurrent Q-network maintains a hidden state, h, in its RNN, occupying the penultimate layer of the network, which aggregates information from the noisy signal across time in order to estimate the value of the underlying state over all possible actions, a. Note that the hidden state takes the role of the agent's belief regarding the true state of the environment and as such the state-action value estimate in each case is a function of the hidden state, as opposed to the observation itself.

In [16] the authors extend the DQN algorithm to solve POMDP versions of 9 Atari 2600 games (e.g. Pong). As explained in section 2.3.1, the authors begin by making the observation that a single frame of a game screen is insufficient to satisfy the Markov property, as an agent is unable to deduce the direction or speed of moving objects on the screen. In order to address this problem, there are two possible approaches. The first approach, as implemented in DQN, is to stack the frames from the preceding K time steps (usually $K = 4$) into a single 3D tensor to be processed in a single forward pass by the network. The second approach is to include, as the penultimate layer in the Q-network architecture, a recurrent layer consisting of an RNN, as is illustrated in Figure 3.3. In their paper [16], the authors choose to use an LSTM, which, they hypothesise, should equip the agent with 'memory', giving it the ability to deduce the underlying dynamics at each time step by integrating the features of the current observation with the information from previous observations encoded in the hidden state of the RNN. Note that the Q-values are generated by a final dense layer which takes as input the hidden state from the RNN layer.

As a baseline evaluation, the authors first compare DQN and their recurrent variant, named Deep Recurrent Q-Network (DRQN), on the test set of Atari games wherein $K = 4$ for both agents (i.e. the observation received by both agents includes the most recent 4 frames). The results obtained demonstrated that DQN and DRQN are comparable, with DRQN outperforming DQN in 2 of the 9 games in which long-term memory is required to make progress in the game. It is worth noting that long-term memory required for acting over long time horizons (e.g. remembering something from several time steps ago in order to perform an action in the present which yields high return), serves a different function than that of making the observing Markov, which also requires aggregating information over time, but possibly over a shorter horizon.

The authors then compare both agents on the so-called 'flickering' variants of the Atari games wherein each frame is fully obscured via zero-masking at each time step with probability 0.5. For a fair comparison, the

observations received by the DQN agent include the most recent $K = 10$ frames, where potentially several adjacent frames are fully masked, and the DRQN agent is trained via backpropagation on sequences of 10 frames. In this way, both agents have access to the same information [3], but the DRQN agent must handle the partial observability arising from the non-Markov nature of single-frame observations, from which it is impossible to deduce object velocity, as well as that arising from the intermittent masking, requiring the agent to 'bridge the gap' between masked frames.

The results of the second evaluation demonstrate that the DQN and DRQN agents perform comparably on the flickering variant of the Atari games, with the only statistically significant differences being in Beam Rider, where DQN outperformed DRQN, and Pong, where DRQN outperformed DQN. Importantly, the authors examine the convolution filters in the 3 convolutional layers of Q-networks belonging to the trained DQN and DRQN agents and observe that both agents, in spite of the DRQN agent observing only a single frame at a time, are able to integrate noisy information over time in order to detect high-level events in Pong such as the agent missing the ball, the ball reflecting off the paddle, and the ball reflecting off the wall. The authors conclude from these results that DRQN, optimised over sequences of 10 frames (time steps), presents a viable alternative to DQN with $K = 10$, demonstrating that recurrent networks can indeed integrate information through time and serve as a viable alternative to frame-stacking.

The final comparison of DQN and DRQN was to evaluate the ability of each algorithm to generalise from an MDP setting to a POMDP setting. In order to achieve this, the authors vary the observation probability in the flickering variants of the 9 Atari 2600 games (inversely, increasing the probability of obscuring the frames via zero masking), beginning with $p = 1.0$ (fully observable) and decreasing to $p = 0.1$ in increments of 0.1. The results, illustrated in Figure 3.4 which shows the mean percentage of the original score achieved by each agent, demonstrate that as the observation probability decreases, the performance of DRQN degrades at a slower rate than that of DQN, with DRQN maintaining it's performance at 50% of it's original score between $p = 0.7$ and $p = 0.4$, compared to the drop from 35% to 25% suffered by the DQN agent over the same interval.

Given that production systems must account for potential software bugs as well as hardware failure, the prospect of an agent which is able to resist performance degradation under an increasingly noisy signal is desirable. The experimental work reported in [16] is the basis for the experiments conducted in this paper, in particular, inspiration is taken from the idea of randomly obscuring the observation received by the agent via masking to induce conditions of partial observability.

DRQN presents a solution to the problem of partial observability in a dynamic RL environment which is to learn to estimate state-action values conditioned on the historical sequence of observations received by the agent where the aggregation of information over time is captured by the hidden state of the RNN layer of the Q-network. Now, as demonstrated in section 2.3.5, value estimates in the context of a belief-space MDP are formed on the basis of a belief maintained by the agent which is conditioned on the historical sequence of both observations *and* actions. This makes sense as state transition probabilities of the underlying MDP depend explicitly on both states and actions giving rise to the intuition that value estimation might be improved by considering the combination of observations and actions.

In the paper *On Improving Deep Reinforcement Learning for POMDPs* [17] the authors make precisely this argument and propose the idea of extending DRQN [16] by conditioning value estimates on sequences of observation-action pairs, as opposed to observations alone. To this end, the authors propose a new Q-network architecture which they name Action-specific Deep Recurrent Q-Network (ADRQN). ADRQN, illustrated in Figure 3.5, extends the DRQN architecture by adding a dense layer for encoding actions (in the form of vectors), the output of which is concatenated with the latent representation of the observation output by the network body (a CNN followed by an MLP) and passed jointly into the LSTM layer in order to update the hidden state from which the action value estimates are produced.

The effectiveness of the ADRQN model is demonstrated by experiments carried out on several partially observable domains, including the flickering version of the Atari games proposed in [16]. Results on the test environments demonstrate improvements over DRQN, both in terms of the rate at which the agent learns

[3] However, (Karpathy, Johnson, and Li 2015) show that LSTMs can learn functions at training time over a limited set of time steps and then generalize them at test time to longer sequences.

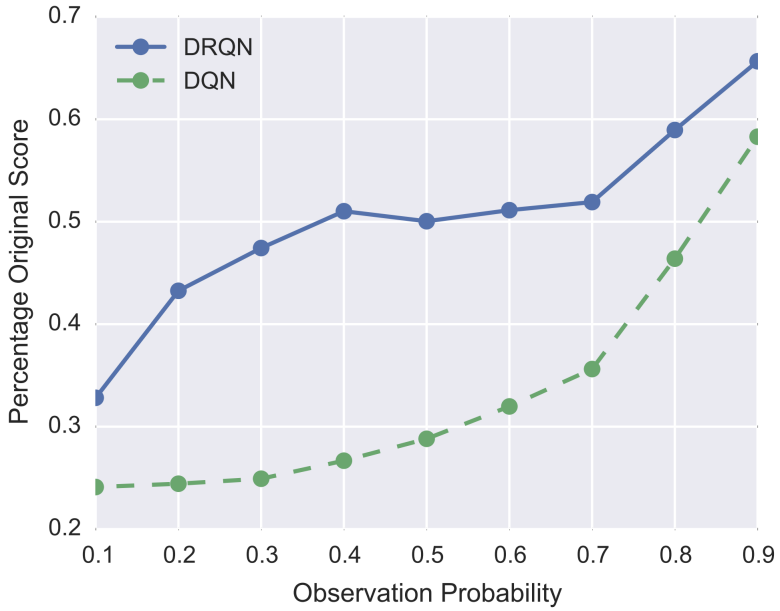


Figure 3.4.: An illustration of the mean percentage of the game score obtained by DQN and DRQN under full observability, trained on the MDP versions of 9 Atari 2600 games and evaluated on flickering (POMDP) versions of the same games with decreasing observation probabilities (increasing masking probabilities) [16]. The results demonstrate that the inclusion of the RNN layer in the Q-network compensates for the lack of information arising from the increasing partial observability due to its ability to aggregate information gleaned from partial observations across time, allowing DRQN to degrade more gracefully as the observation probability is decreased.

tasks and the maximum scores achieved. In addition, the architecture is stable and does not require extensive hyper-parameter tuning. The authors also test the agent's ability to generalise from an MDP setting to a POMDP setting, as in [16], with results showing minor improvements over DRQN with respect to performance degradation as the masking probability is increased. The improvement was especially evident in the game of Frostbite, an Atari 2600 game which requires memory in order to succeed.

Altogether, the results in [17] add evidence in support of the hypothesis that value estimation in the POMDP setting can be improved by conditioning the Q-network on historical sequences of both observations and actions.

While the inclusion of an RNN layer as a memory mechanism has been demonstrated by the above works in the context of the DQN algorithm, it is straightforward to extend these ideas to the Policy Gradient setting. For example, in [88] the authors discuss the technical details regarding the utilisation of RNN-based policies in PPO. It is worth noting that training a Q-network to predict state-action values based on sequences presents technical challenges in DQN owing to the memory buffer - instead of sampling individual transition tuples, one must sample random sequences of length K, which involves several statistical and engineering considerations, often making recurrent policy gradient more straightforward as optimisation occurs directly from sampled trajectories which are discarded after each network update.

As demonstrated in this section, RL agents require memory to operate under conditions of partial observability. While many of the examples above have utilised the LSTM, there are many other possible memory mechanisms, including other RNNs and even rudimentary methods such as frame stacking. In [89] the authors conduct the largest benchmarking comparison of different 'memory models' [4] used in RL policy architectures in POMDP tasks, including frame-stacking, Elman networks (vanilla RNNs), LSTMs, GRUs, and more. They determined that GRUs are the best general-purpose memory model for RL tasks, performing consistently on par, or better than, LSTMs, and with greater efficiency. Beyond RNNs, as noted in section 2.4.1, Transformers have essentially replaced RNNs as the SOTA architecture for many tasks involving sequential data. Given this, a natural avenue of research has emerged investigating the application of Transformer-like self-attention in

[4]The word "models" feels like an odd one here, but it's the one used by the authors.

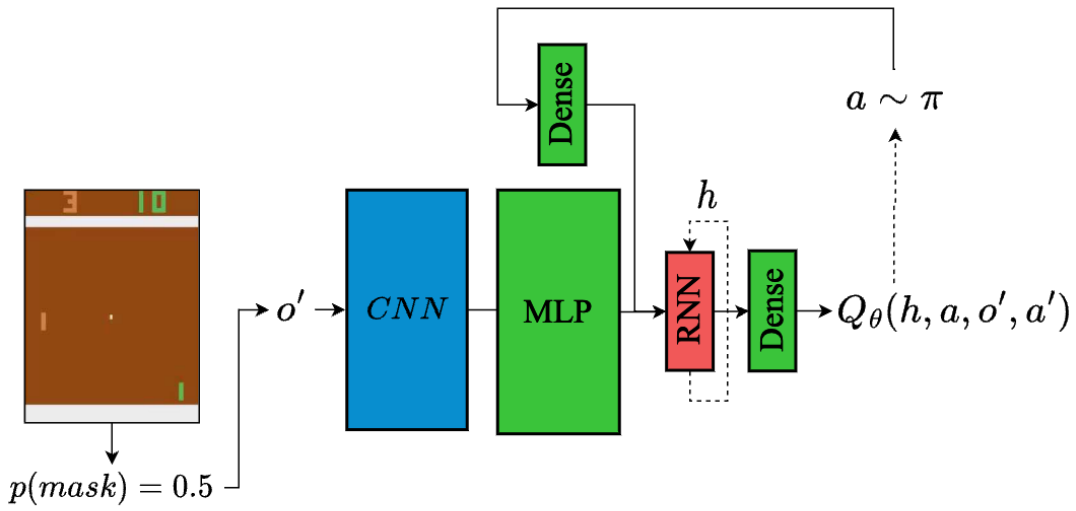


Figure 3.5.: An illustration of the Action-specific Deep Recurrent Q-Network (ADRQN) proposed in [17] performing value estimation on the flickering version of an Atari 2600 game. ADRQN takes as input both the partial observation o' and the previous action a (a vector), which is encoded via a dense layer and concatenated with the latent representation of the observation output by the network body (a CNN followed by an MLP), forming the input for the RNN layer which maintains the hidden state h used ultimately to estimate the values for possible actions a'. In this way, the hidden state encodes information from the sequence of observations received by the agent and the actions which gave rise to them, resulting in improved learning efficiency and higher game scores in the experiments conducted by the authors.

RL, including in the partially observable setting, in the hopes of replicating something like the improvements observed in other domains - more on this in section 3.3.

3.2 Recurrence & Attention For Interpretability In RL

While Deep RL algorithms such as DQN and PPO have shown to be effective approaches to solving complex sequential decision-making tasks, one downside of using an ANN trained via gradient descent as an RL policy is that the 'logic' by which the agent makes decisions is typically completely opaque. It is very challenging to decipher exactly what certain weights in the network represent in a way that is concretely related to the task at hand, and thus interpret how it arrives at certain decisions, which is crucial for many mission-critical applications. For example, in the medical domain or in the case of self-driving cars, being able to explicitly observe and interpret the decision-making process of software systems is critical in case something goes wrong[5]. Much work has been done on this topic [90] and among the possible approaches, attention-based methods present one avenue for improved interpretability, especially in vision-based RL tasks. We cover two such methods in this section.

In [18] (2015) the authors address the issue in the domain of "interpretability in RL" by augmenting the DRQN architecture with an attention mechanism which forces the agent to allocate a limited attention budget to parts of the input space which it deems most important for the task. The authors trained their novel Deep Attention Recurrent Network (DARQN) architecture on the canonical set of Atari 2600 environments. The reason for this choice was that, for any given game, the salient visual features that require the attention of a reinforcement learning (RL) agent are fairly unambiguous. For example, in Atari Pong, the agent needs to attend to the ball and the two paddles, making it easier to assess the efficacy of the architecture. The so-called attention weights are allocated by the agent at each time step and may be projected back onto the input image,

[5] If only for legal reasons.

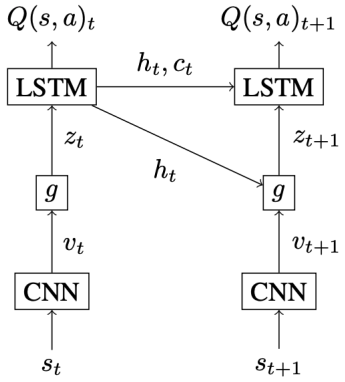


Figure 3.6.: The Deep Attention Recurrent Q-Network architecture [18]. The authors extend the recurrent Q-network by inserting an attention bottleneck g between the CNN core of the network and the LSTM layer, which forces the agent to allocate a fixed attention budget over the input image based on which information is most salient for the purposes of decision-making.

allowing for an interpretation of which visuospatial features are most salient for the purposes of (optimal) action selection.

The DARQN architecture - illustrated in Figure 3.6 - is comprised of a CNN core which, in a manner similar to the recurrent caption generation model [15] described in section 3.1.1, takes as input a single frame (or stack of frames), x_t, in order to produce a set of features maps of shape (H, W, C) where each of the $L = H * W$ feature vectors, $v_t = \{v_t^1, \ldots, v_t^L\}$ (each having dimension C), corresponds to a visual feature located at its corresponding spatial location in x_t. In between the CNN core and the LSTM is an attention mechanism[6] which acts as a bottleneck, forcing the agent to choose which parts of the input space to attend to, and which to ignore. The authors propose two attention mechanisms; a 'hard' attention mechanism which samples only a single feature vector at each time step, and a 'soft' attention mechanism which performs a weighted aggregation of all feature vectors. We first consider the soft attention mechanism, g, which allocates a fixed unit of attention by computing attention weights, α_t^i, for each feature vector such that $\sum_{i=1}^{L} \alpha_t^i = 1$. In order to compute each weight, g takes as input a given feature vector, v_t^i, and the previous hidden state, h_{t-1}, of the LSTM, transforming each by a set of linear layers, followed by a softmax function:

$$\tilde{\alpha}_t^i = f_{dense}(\varphi(f_{dense}(v_t^i) + W h_{t-1})) \tag{3.3}$$

$$\alpha_t^i = \frac{e^{\tilde{\alpha}_t^i}}{\sum_j e^{\tilde{\alpha}_t^j}} \tag{3.4}$$

where f_{dense} represents a dense layer, φ represents the tanh activation function, and W is a weight matrix matching the dimension of h_{t-1} along it's column axis. The attention weights are then used to compute a so-called context vector, z_t, by performing a simple weighted sum over all the feature vectors:

$$z_t = \sum_i \alpha_t^i v_t^i \tag{3.5}$$

The context vector is then fed as input into the LSTM and onto the value head which is used to estimate state-action values in the usual way. Due to the softmax, the agent is forced to decide how to allocate the unit of attention, using the temporal information in combination with the feature information in each feature vector, v^i, in order to decide which features to attend to and to what extent. In this way g forms an attention bottleneck which forces the agent to assign the largest attention weights to the most salient features in order to obtain the information required for decision-making. The attention weights α^i may then be projected back

[6]Note: Both attention mechanisms are distinct from the one used in the Transformer architecture.

onto the input image at each time step allowing for a (partial) visual interpretation of the agent's decision-making process.

Contrast against the soft attention mechanism which performs a weighted sum over all feature vectors in v_t to produce the context vector, z_t, the hard attention mechanism is far more restrictive, allowing the agent to sample only a single feature vector from v_t, which is used as the basis for action selection.

While the results obtained by DARQN relative to the DQN baseline were mixed, with DARQN producing superior results when trained on some Atari 2600 games and inferior results on others, the attention mechanism provides novel insight into the agent's decision-making in the form of the visual attention map described above, an illustration of which is shown below in Figure 3.8.

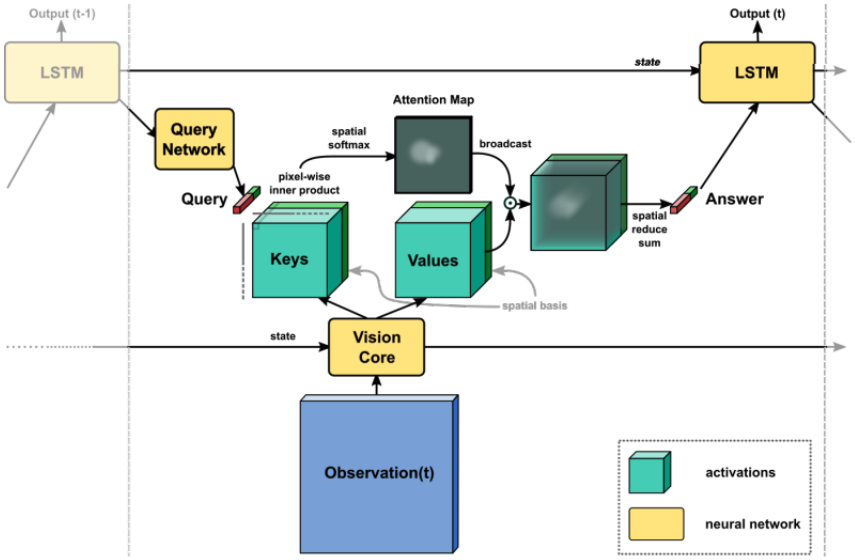


Figure 3.7.: The recurrent attention-based architecture proposed in [19]. The authors propose an attention bottleneck which forces the agent to select which parts of the input image to attend to based on which visuospatial features are most salient for the purposes of decision-making. The attention bottleneck works by deriving matrices of 'key' and 'value' vectors from the tensor output by the CNN network core. A sequence of inner products between a 'query' vector, output by the LSTM layer, and each of the key vectors is performed, followed by a softmax operation, to produce a 2D attention map wherein all values sum to 1. A final inner product and summation operation between the attention map and the values matrix is performed to produce an 'answer' vector which is fed back into the LSTM, the output of which is used to perform action selection and value estimation.

Similar to [18], the authors of *Towards Interpretable Reinforcement Learning Using Attention Augmented Agents* [19] (2019) use the Atari 2600 suite of games as a test bed for a novel attention-augmented architecture, illustrated in Figure 3.7, which is also composed of a CNN vision core, for performing feature extraction, an attention bottleneck[7] which forces the agent to allocate attention over a set of visuospatial feature vectors, and an LSTM layer which maintains a hidden state and produces a set of query vectors at each time step which ultimately determines what the agent attends to.

At each time step, t, The attention agent takes as input an RGB image, x_t, which is processed by the CNN vision core to produce a set of feature maps of shape (H, W, C) which is split along the channel dimension to form a set of keys, $K_t \in \mathbb{R}^{H \times W \times C_k}$, and values, $V_t \in \mathbb{R}^{H \times W \times C_v}$. A set of fixed spatial encoding vectors, $S \in \mathbb{R}^{\tilde{H} \times \tilde{W} \times C_s}$, is concatenated to both K_t and V_t along the channel dimension. The reason for this is to enforce a spatial structure which is necessary because the outcome of the attention operation is permutation invariant with respect to the relative positioning of the feature vectors in K_t and V_t. The hidden state of the LSTM, h_{t-1}, is then fed through an MLP 'query network' in order to produce a query vector, $q_t \in \mathbb{R}^{H \times W \times C_k}$,

[7]Note: once again, this attention mechanism is distinct from the one used in the Transformer.

which is used to compute an inner product with each feature vector in K in order to produce a 2-dimensional attention map, $\tilde{A} \in \mathbb{R}^{H \times W}$, as:

$$\tilde{A}_{i,j} = \sum_k q_l K_{i,j,k}$$

The attention map is then normalised using a spatial softmax function:

$$A_{i,j} = \frac{\exp(\tilde{A}_{i,j})}{\sum_{i',j'} \exp(\tilde{A}_{i',j'})}$$

The normalised attention map is then broadcast along the channel dimension of the values tensor, $V \in \mathbb{R}^{H \times W \times C_v + C_s}$, multiplying point-wise and summing across the spatial dimensions in order to produce an 'answer vector', $\alpha \in \mathbb{R}^{1 \times 1 \times C_v + C_s}$, as:

$$\alpha_k = \sum_{i,j} A_{i,j} V_{i,j,k}$$

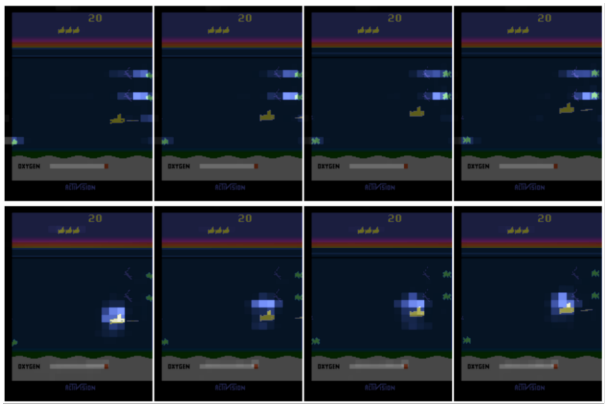
Finally, the query, answer and hidden state vectors, along with the action and reward from the previous time step, are fed into the LSTM, the output of which is fed into the policy and value heads of the network.

One additional implementation detail to note is that the authors implement N attention heads which compute the attention operation between N distinct sets of queries, keys, and values in order to produce N answer vectors which are concatenated and fed back into the LSTM together. In this way, each attention head may allocate attention to different foci in the input space, allowing the agent to more easily focus on many entities/features at once.

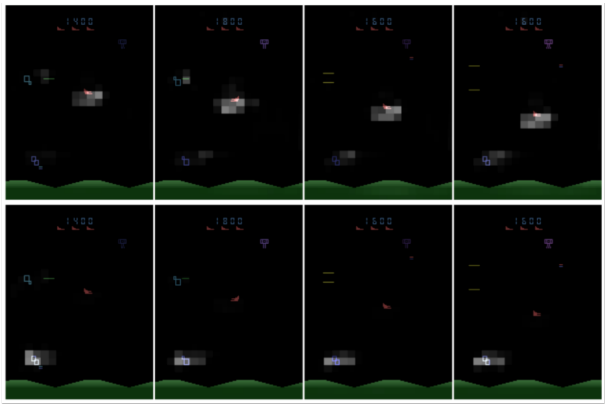
Regarding the concatenation of the spacial encoding vectors; notice that this is different than the approach taken in the Transformer [11] and Vision Transformer [12] case where spatial encoding vectors are summed element-wise with the embedding vectors which are used to produce the queries, keys, and values. Notice too that this stands in contrast to [18], where the authors do not use any spatial encoding in the computation of the context vector. The reason for this choice is to allow the agent to produce separate 'what' vs 'where' queries which allocate more or less attention to the spatial location of a visual feature, as opposed to the feature itself. This adds an extra layer of interpretability as, at inference time, the authors can compute the relative attention allocated by the query vectors to the spatial and/or feature components of the concatenated keys and spatial encoding vectors, projecting each over the input image separately in order to gain further insight into the agent's decision-making.

Examples of the spatial attention maps produced by the attention-augmented agent may be seen in Figure 3.8 - brighter areas mean a greater allocation of attention - for two Atari 2600 games; Seaquest, where the agent pilots a submarine and must shoot enemies as they appear on the screen and rescue divers, and a second game called Star Gunner. In both examples, the two rows show maps from two of the four attention heads and examining each gives information regarding what the agent is attending to. In Seaquest, for example, the attention head represented by the top row is attending to enemies on the screen whilst the attention head represented by the bottom row is attending to the submarine (the player). To test whether the agent was able to generalise, the authors introduced familiar-looking enemies into the game screen at the pixel level, but in unexpected locations and at unexpected times. In doing so, the authors observed that, despite the state being novel to the agent, it was able to attend to the artificially introduced enemies and respond appropriately (i.e. by shooting at them), demonstrating a clear ability to generalise by adapting to previously unseen states. Finally, the authors examined separately the 'what' vs 'where' components of the queries produced by the agent in the racing game Enduro, observing that some attention heads weigh more heavily towards querying 'what' features - focusing on the player car and the score (which is useful for computing value estimates) - and others towards 'where' features - focusing on the region just in front of the player car, acting as a 'tripwire' for upcoming cars in order to avoid a collision.

The results obtained by the attention-augmented agent were found to be competitive with state-of-the-art baseline architectures, although the primary reason for investigating such architectures is not to produce a superior agent, but rather to examine the issue of interpretability using attention mechanisms.



(a) Seaquest



(a) Star Gunner

Figure 3.8.: An illustration of the attention maps produced by the recurrent attention networks in [18] and [19]. The bright areas represent areas of high attention. In this way, the agent's decision-making over time can be associated with objects tracked over the course of each episode by examining its attention maps.

3.3 Transformer-Like Attention & Partial Observability In RL

As noted in section 2.4, Transformers have shown breakthrough success in the fields of both NLP and computer vision, owing to their ability to integrate information over long time horizons and scale to massive amounts of data. Naturally, researchers have since attempted to ascertain whether the application of Transformers in the domain of RL might yield similar gains. In this section, we review the papers relevant to our purposes which explore this question.

One common theme echoed in much of the research cited in this section is that Transformers are difficult to optimise. In one of the earliest research efforts made in applying Transformer-like self-attention in the context of RL [91] (2017), the authors propose a simple meta-learner architecture which aggregates temporal signals in order to better handle dynamically changing tasks - at the end of the paper they recount their frustration in attempting to train a pure Transformer-like version of their proposed model by naively having it attend across the temporal dimension:

> *"This model, which is equivalent to [the original] Transformer architecture, could not solve the bandit or MDP tasks. In both domains, its performance was no better than random. To no avail, we experimented with multiple blocks of attention and multiple heads per block. We hypothesize that this architecture's inadequacy stems from the fact that purely attentive lookups cannot easily process sequential information. Despite their infinite receptive field, they cannot directly compare two adjacent time steps (such as a single state-action-state transition) in the same way as a single convolution can. The TC layers are essential because they allow the agent to locally analyse contiguous parts of a sequence to produce a better contextual representation over which to attend."*

Much of the research covered in this section attempts to address challenges similar to these and attempts to understand how the unique properties of the Transformer might be applied to solving specific challenges in the RL domain.

3.3.1 Attention As Memory

In [20] (2018), the authors consider the role of neural memory mechanisms in partially observable tasks which require reasoning regarding the relationships between entities observed over time. They begin by hypothesising that existing memory-based ANNs, such as GRUs or LSTMS, whilst proficient at storing and retrieving information across time, struggle to perform complex relational reasoning tasks on temporal data. That is,

they struggle to reason about the relationships between entities encountered across time. The authors address this issue by introducing the Relational Memory Core (RMC), which integrates partial observations over time in a manner similar to RNNs, but instead uses a memory module based on Transformer-like multi-head attention, which, the authors argue, allows for improved relational reasoning between entities over the temporal dimension. In this way, the authors bypass the issues encountered in previous research [91] when utilising Transformer-like attention to attend over the temporal dimension directly.

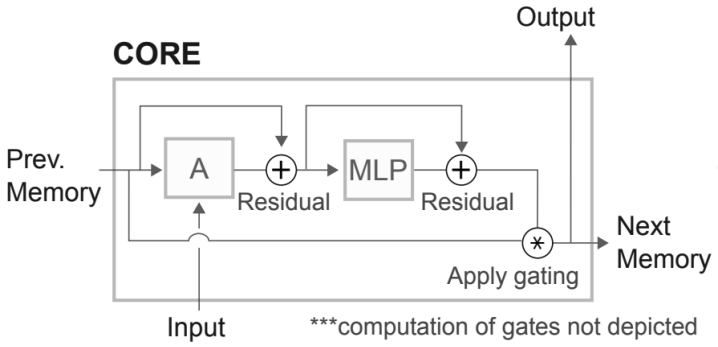


Figure 3.9.: The relational memory core (RMC) proposed in [20]; an attention block comparable to that used in the Transformer and ViT architectures. The RMC performs a memory function, maintaining a memory matrix M which is updated with new information at each time step in the form of a partial observation vector x via a multi-head attention (MHA) layer, followed by an MLP with two interleaving residual connections, to produce a candidate memory matrix $\tilde{M}$. M and $\tilde{M}$ are used to compute an updated memory matrix via a gating mechanism, for example, an LSTM. In this way the RMC leverages MHA in order to aggregate information over time in a manner similar to, but distinct from, classical RNNs such as the LSTM.

The core component of the RMC, illustrated in Figure 3.9, is a memory module which takes the form of an attention block equivalent to that of the Transformer encoder, with some minor modifications. The RMC maintains a memory matrix, M, akin to the hidden state maintained by a GRU or LSTM RNN, which aggregates information across time in a manner optimised for task-specific decision-making. At each time step, the agent receives new information in the form of a partial observation, encoded in the form of a vector, x. The RMC then computes a new candidate memory matrix, $\tilde{M}$, which integrates the new information with the historical information encoded in M via multi-head dot-product attention, in which each head computes:

$$\tilde{M} = \text{softmax}\left(\frac{MW_q\left[M;x\right]W_k}{\sqrt{d_k}}\right)\left[M;x\right]W_v$$

where $[M;x]$ indicates row-wise concatenation, ensuring that $\tilde{M}$ and M are of equivalent dimensions. In this way, x is projected by W_k into the common embedding space, forming a key vector which interacts with the historical information in the query matrix to produce the attention weights in the final column of the matrix to which the softmax is applied. In this way, the authors hypothesise, the agent is equipped with the ability to reason between entities across time encoded in the rows of the memory matrix.

Following the attention operation, the RMC design includes an MLP and two interleaving residual connections (leaving out the LayerNorm layer used in the Transformer and ViT), which process the output of the MHA layer to produce a new candidate memory matrix, $\tilde{M}$. Instead of replacing M directly, the authors propose a gating mechanism in order to integrate the information from $\tilde{M}$. The specific gating mechanism used by the authors in their experiments is that of the LSTM, wherein the memory matrix, M, plays the role of the internal cell state, C, which the LSTM maintains, in addition to the hidden state vector, h. In order to update M, each partial observation, x, is used to compute special input, output and forget gates in the form of vectors and used in combination with sigmoid and tanh activation functions to combine individual row vectors from M and $\tilde{M}$ in order to produce a new memory matrix, as well as a new hidden state vector. The

use of gating in conjunction with self-attention is worth noting here as it is a recurring theme in the literature, also employed in [21].

In sum, the agent maintains a relational memory matrix which it updates via a special attention operation in order to incorporate new information from partial observations. Each new memory matrix is ultimately produced via an LSTM-like gating mechanism which uses each new observation in the computation of gating vectors which are used to blend information from each pair of row vectors from the current and candidate memory matrices, producing a new matrix and new hidden state. The hidden state vector may then be processed by downstream layers in order to compute the policy and/or value estimates.

The authors evaluate an agent with an RMC-based policy architecture on a POMDP version of the Mini Pacman environment, wherein the agent must navigate a maze in order to collect food while being chased by ghosts. Instead of the entire game screen, the observation emitted at each time step consists of a 5x5 pixel patch which centres on the agent's position in the maze. In this way, the task is made partially observable as the agent's view at each time step is insufficient to capture the full state of the environment. Crucially, the agent must predict the dynamics of the ghosts in memory to avoid dying, planning its navigation accordingly, in addition to remembering information regarding the location of food it has already consumed and has yet to consume. The authors report their RMC agent was able to improve on the score obtained by an LSTM-based agent in the POMDP setting by approximately 12%, nearly doubling the LSTM agent's score when the task was made fully observable. These results suggest that transformer-like attention may offer an advantage as a mechanism for aggregating information over time (i.e. memory) over conventional RNNs, especially in partially observable environments.

3.3.2 Attention Over Time

In *Stabilizing Transformers for Reinforcement Learning* [21] (2020), the authors note the limited success reported in applying self-attention architectures in RL, making the observation that a key challenge is the instability of self-attention-based architectures during training, resulting in poor sample efficiency (slow learning) and difficulty converging on optimal policy/value network parameters. The authors reiterate the successes achieved in applying Transformers in the NLP domain, stating that in theory, Transformer-like policy architectures should yield similar gains especially when applied to partially observable RL tasks.

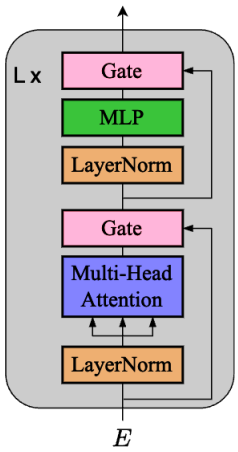


Figure 3.10.: The gated attention block proposed in [21]. The signal from the multi-head attention and MLP components is modulated by the input in order to stabilise learning when incorporated into an RL policy architecture.

In their introduction, the authors note that although superior memory mechanisms to the LSTM have been proposed for dealing with partial observability in RL, they have not seen widespread adoption, likely due to their complexity. The Transformer architecture, they argue, offers a simple alternative which is able to encode the entire history of partial observations received by an agent in one shot, as opposed to maintaining a hidden

state which is updated over time, which may make it harder for LSTMs to reason about relations between entities observed at different points in time, as noted by [22].

However, as reported in [91], the authors find that naively applying self-attention over a sequence of embedding vectors corresponding to discrete time steps is significantly more difficult to optimise, often resulting in performance comparable to a random policy. They hypothesise that the instability which arises in policy architectures which use self-attention in this way is due to the residual connections which occur after the multi-head attention and MLP components. In order to address these challenges and test their hypothesis, the authors propose a simple policy architecture - the Gated Transformer XL (GTrXL), illustrated in Figure 3.10 - which uses a modified attention block, replacing the residual connections with a gating mechanism to modulate the signal produced by the multi-head attention layer.

The authors note that multiplicative interactions, such as the gating mechanisms in the LSTM and GRU RNNs, have been successful at stabilising learning across a wide variety of architectures. Motivated by this, the authors propose and experiment with a number of multiplicative gating mechanisms via which the input signal is allowed to modulate the output of the multi-head attention and MLP layers. Among the gating mechanisms tested by the authors, the top two performers - by a significant margin - were a simple *output* gate, where the output connection is modulated by a dense layer with a sigmoid activation:

$$g(x, y) = x + \sigma(W_g x + b_g) \odot y$$

and a *GRU* gate which has a form equivalent to that illustrated in Figure 2.14 (b) where the GRU's hidden state, h, is replaced with the input signal, x, and the GRU's input is replaced with the output signal, y.

The authors test their gated-attention policy architecture variants - trained using a policy gradient method - on a suite of tasks; some partially observable and requiring memory on the part of the agent, as well as some fully observable tasks. Their principle finding is that the gating mechanisms - the 'output' and GRU gates in particular - enable stable learning on tasks where, without the gating mechanism, the transformer-based policy fails to learn completely. What's more, the authors find a significant improvement over the LSTM baseline policy on the partially observable tasks and comparable performance on the fully observable tasks.

In sum, the gated variant of the attention block presented in [21] demonstrates that self-attention may indeed be effectively utilised in an RL policy architecture in order to encode a temporal sequence of partial observations. For our purposes, it is worth asking the question of whether such gating mechanisms might provide a performance benefit when utilised in a policy architecture wherein self-attention is used to encode a single partial observation.

In *Deep Transformer Q-Networks for Partially Observable Reinforcement Learning* [92] (2022) the authors seek to improve on the challenges in training RNN-based Q-networks on POMDP RL tasks: catastrophic forgetting, a relatively large number of parameters, and a poor ability to generalise leading to the need for large amounts of training data and poor sample efficiency. In seeking to address these challenges, the authors propose a novel architecture called the Deep Transformer Q-Network, which uses a self-attention mechanism to encode historical sequences of partial observations - similar to the work proposed in [23] and [21] - in order to estimate state-action values.

The proposed Deep Transformer Q-Network (DTQN) architecture takes as input the agent's previous k partial observations, linearly projecting each into the common embedding space and adding positional encodings - the authors experiment with both learnable and fixed sinusoidal positional encoding vectors, as well as with not using any at all. The network body consists of several attention blocks applied in sequence, producing k embedding vectors which are used in turn to predict the state-action values for each of the k time steps simultaneously. A crucial detail regarding the attention block is that it implements causal masking, as in the decoder of the original transformer architecture, which masks the attention values computed for observations $0, 1, \ldots, j-1$ in the sequence when estimating the Q-value for the j^{th} observation in the sequence - this makes sense because at inference time, the agent will only be able to rely on historical observations in order to estimate the value of the present state-action pair. The authors note that for this reason, the agent must learn to estimate Q values at the beginning of the sequence with relatively little information, which they claim makes for a more robust policy when trained on partially observable tasks.

In their experiments the authors test their DTQN architecture on a number of partially observable vision-based tasks against a number of baseline architectures including the DQN, to show the importance of memory, and DRQN to show improvement over RNN-based architectures. Additionally, the authors compare DTQN to a baseline attention network which is equivalent to the DTQN architecture, except it uses only a single attention block in which the LayerNorm layers and residual connections are excluded, which is important because (to the best of our knowledge) no prior work has experimentally confirmed the benefits of using LayerNorm layers and residual connections in the attention block in the context of RL.

In the results reported in [92], DTQN outperformed the baseline models on all partially observable tasks. DRQN showed comparable performance on many of the tasks but was slower to learn and suffered from catastrophic forgetting, demonstrating the value of using attention to encode the entire observation history. A pattern observed during training with the baseline attention network was that it learned quickly initially, in a manner comparable to DTQN, but would become unstable and struggle to improve beyond a point, demonstrating the benefit of LayerNorm, residual connections, and the use of multiple attention blocks for more complex tasks. The DQN baseline failed to make any progress learning on nearly all POMDP tasks, as expected since they all required memory.

Finally, the authors performed an ablation study testing several modifications to their architecture. First, they experimented with swapping out the residual connection in the attention block with the GRU gating mechanism proposed in [21], finding that it provided only minor performance improvements on one of the three test environments, but performed comparably overall, which is surprising given the claim made in the source paper. Secondly, the results from experimenting with different positional encoding methods showed that, while all three encoding methods were shown to be comparable on average, an absence of positional encoding altogether only produced significantly worse results in one of the three tasks, whereas the sinusoidal positional encodings demonstrated consistent performance over all tasks.

3.3.3 Attention Over The Input Space

In the paper titled *Relational Deep Reinforcement Learning* [22] (2018), the authors identify two areas of weakness in (then) conventional deep learning reinforcement approaches; the ability of agents to generalise to novel tasks and interpretability. The authors seek to improve on these weaknesses, proposing an attention-based policy architecture to enable the agent to reason directly about relations between visual entities globally across the entire input space. Specifically, the authors draw on decades-old ideas from the field of Relational Reinforcement Learning in which states, actions, and policies are represented using a pre-determined relational language. For example, the predicate *above*(s, A, B) could be used to indicate that in some state, s, an object, A, is positioned above an object, B, in some coordinate plane - this relation could then be applied generally to any two objects encountered by the agent to 'reason' about relative positioning. The authors extend this idea by applying it in the context of deep learning where, instead of equipping the agent with hard-coded relational predicates, the agent is able to leverage the inductive bias of the self-attention mechanism, as applied in the case of the ViT, to learn abstract relations between visuospatial entities in the input space.

As opposed to maintaining a hidden state in the form of a memory matrix and using self-attention to integrate new partial observations [20], or performing attention over the time dimension in order to aggregate information from the entire sequence of historical partial observations in a single shot [91] [21] [92], the authors [22] proposed an attention-based policy architecture which computes self-attention over each observation in isolation in order to model relations between abstract visual entities in the input space and enable better reasoning and planning towards achieving the task-specific objective. The proposed policy architecture, illustrated in Figure 3.11, is structured as follows. First, the visual input (e.g. tensor of pixel values representing a game screen) is processed by a 2-layer CNN in order to extract a set of C feature maps of height H and width W, each of which models visual features at corresponding x-y coordinates in the input, which is subsequently reshaped to produced an embedding matrix E with $H * W$ embedding vectors of dimension C. This is similar to the image captioning method employed in [15] - each of the embedding vectors may be thought of as representing abstract visual entities in local regions located at the associated spatial coordinates in an input frame/image. In order to encode the relative positioning of the feature vectors in E, the authors

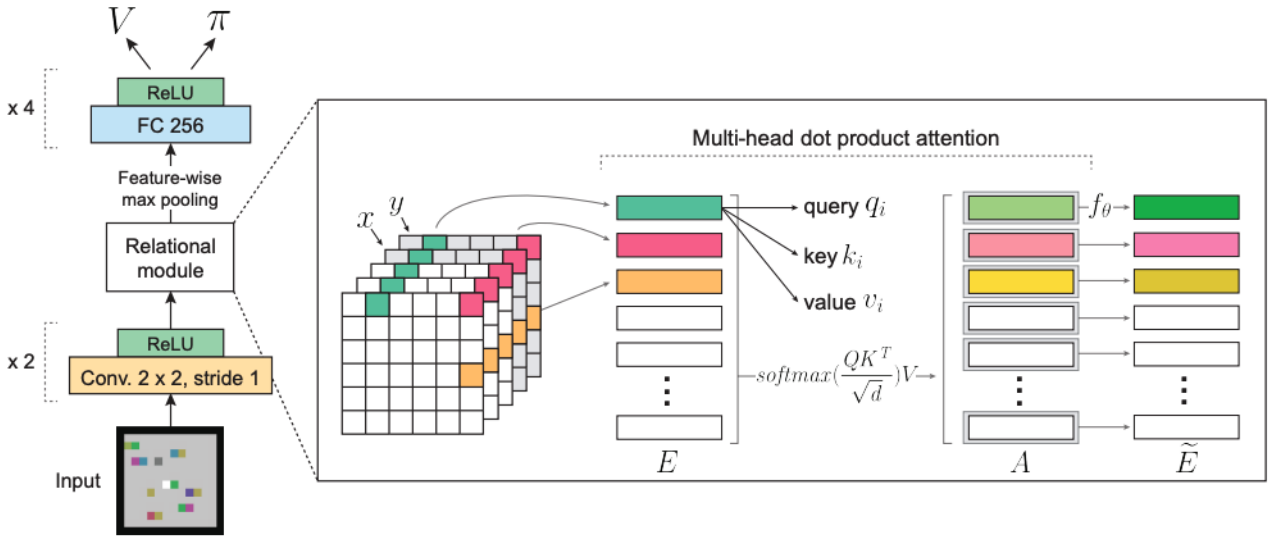


Figure 3.11.: An illustration of the attention-based (relational) policy architecture proposed in [22]. Self-attention is computed directly over feature vectors representing visuospatial entities in the input in order to model abstract relations between them, enabling the agent to better reason and plan towards solving task-specific objectives.

concatenate onto each vector a small positional encoding vector which encodes its relative x-y position. The embedding matrix, E, is then passed through a relational module consisting of one or more Transformer-like encoders, each of which includes an MHA layer and an MLP with interleaving LayerNorm layers and residual connections - crucially, the MHA layer computes self-attention over E, as in the case of the ViT. The authors hypothesise that, unlike CNNs which model interactions between local regions in an input image, the attention blocks are able to model global relations between entities across the entire input space. Following the relational module, the authors include an optional RNN layer (not illustrated) in cases where the environment is partially observable - as is the case in the StarCraft environment used by the authors in their experiments - equipping the agent with memory. The final section of the network consists of an MLP followed by a policy head and a value head - a structure similar to the actor-critic architecture illustrated in Figure 2.18.

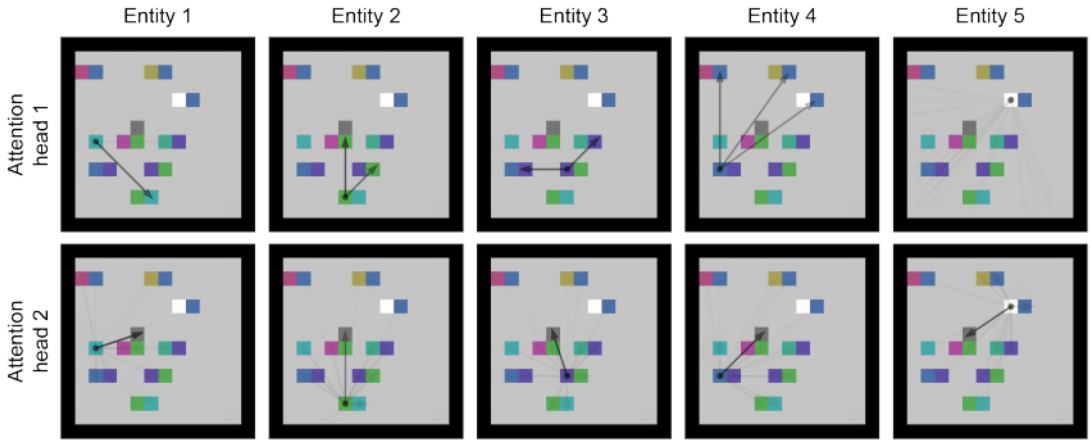


Figure 3.12.: An illustration of the relations modelled by the attention heads in the policy architecture proposed in [22] in the Box-World task. Each attention head appears to model sensible relations between meaningful entities in the input space, such as those between locks and keys.

The authors test their proposed policy architecture against a baseline CNN policy on two environments designed to test the agent's ability to model relations between entities across the input space. The first en-

vironment is a custom Box-World environment; a set of generated pixel grids containing pixels representing keys, locks, gems, and entities specifically designed to distract the agent from the core task, which is to gather as many gems as possible by finding keys and locks, all of which are positioned randomly throughout the grid. It is worth noting that the agent does not require memory to solve this task as collected keys are represented in the corner of the grid when they are collected by the agent, serving as an external memory mechanism and ensuring the Markov property is satisfied. The relational agent was able to significantly outperform the baseline agent on the Box-World tasks, demonstrating improved sample efficiency, interpretability, and generalisability to novel, more complex versions of the task. Upon examination of the attention weights, the authors confirmed their hypothesis; as illustrated in Figure 3.12, each attention head in the multi-head attention layer appears to be modelling relations between specific visuospatial entities in the input scene in order to complete the task. For example, the first attention head might model the relative spatial positioning of the agent to all the important entities in the scene, whereas the second head might model the relations between keys and locks. Furthermore, the relational agent was able to generalise far better to novel variations of the original set of tasks than the baseline agent, suggesting that the relations learned by the agent are akin to the set of hard-coded relational predicates used in traditional Relational RL algorithms, and which may be applied to familiar entities encountered in novel situations.

The second test environment utilised by the authors in their experiments was a set of two miniature Star Craft environments designed specifically to test reinforcement learning agents. In order to succeed, the agent must coordinate the control of multiple entities on the game screen in a cooperative manner to achieve objectives such as gathering resources or defeating enemies. Furthermore, the game is partially observable as the agent might only see a portion of the game screen at any given time, requiring the inclusion of an RNN layer in the relational policy architecture. The relational agent achieved state-of-the-art performance on both tasks, once again demonstrating an ability to generalise to novel variations of the specific scenarios encountered during training.

One noteworthy finding was that the inclusion of additional attention blocks in the relational module, used to enable the modelling of higher-order relations, had a positive impact on performance up to a point on more complex tasks but was shown to degrade the agent's performance and its ability to generalise if too many attention blocks were added. This leads the authors to hypothesise that the attention layers might be overfitting to the vast amount of training data required to train an RL agent, concluding that more work is needed to examine this effect.

In [23] (2019) the authors experiment with a similar self-attention-based architecture to [22], testing it on a set of Atari-like game environments, but with a few minor implementation differences. Instead of processing a single frame per time step and using an RNN to aggregate information from partial observations over time, as in the StarCraft experiments in [22], the authors use frame stacking in the same way as in DQN [9], concatenating frames along the time dimension to form an input which satisfies the Markov property. The tensor of stacked frames is then fed into a convolutional layer in order to produce a set of feature maps, each of which encodes visuospatial features from across the time frame spanned by the stacked frames. The tensor of feature maps is then passed into an isolated self-attention layer (as opposed to multi-head attention or a full attention block) followed by a residual connection, and finally two more convolutional layers, before being passed on to the policy and value heads. The layers preceding the policy and value heads are illustrated in Figure 3.13. In this way, the authors bypass the need to attend over time, which had been shown to be a challenging task for transformer-like attention [91][21]. Instead, the authors choose to have the network perform attention over the abstract input space occupied by the feature maps output by the first convolutional layer, in a manner similar to [22].

Additionally, the authors experiment with including additional self-attention layers in between the second and third convolutional layers. The hypothesis of the authors is that performing self-attention over a set of embedding vectors which encode both spatial and temporal information will yield a performance boost. Testing their attention agent variations against a regular CNN agent on a suite 10 of Atari-like environments, including Ms Pacman and Breakout, the authors find that the top-performing variants of their attention-based architecture perform comparably to the baseline agent in 50% of the environments, and significantly outperform the baseline agent in the remaining environments. As in the StarCraft experiments conducted in [22], the authors analyse the areas attended to in the input space by the attention agent, which demonstrated an ability to reason about relations between entities globally across the input space, as well as the ability

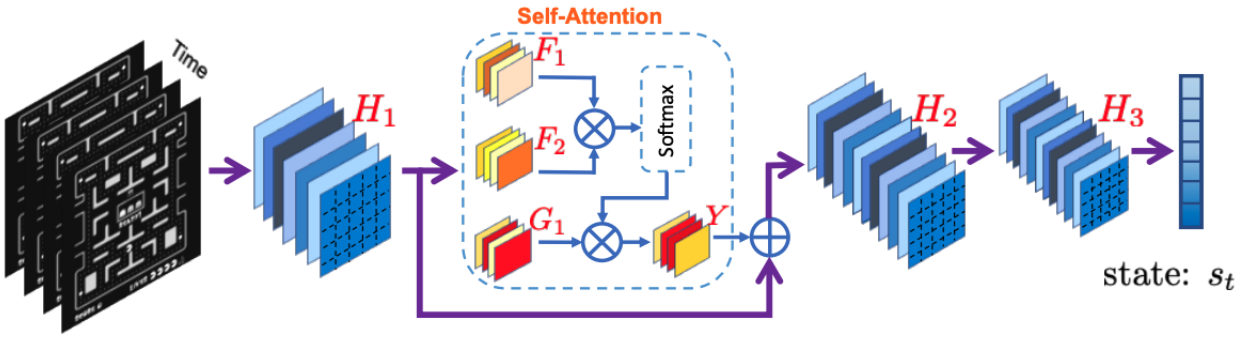


Figure 3.13.: The attention-based architecture proposed in [23] - H1, H2, and H3 are convolutional layers. In order to handle partial observability, the authors utilise frame stacking to allow the attention layer to attend over both spatial and temporal information.

to track and attend to multiple entities (hostile enemies) over time in both fully and partially observable environments, which the authors claim was not equally demonstrated by the baseline CNN agent. In general, the authors note that conventional neural network layers, such as CNNs, have typically been used by default to extract features from observations in RL policy architectures and that further research into alternative methods may yield promising results.

In the following two papers published by Google Brain, [24] (2020) and [26] (2021), the authors study problem scenarios wherein partial observability arises due to only a given fraction of the full observation being available to the agent at any given time step. Furthermore, both utilise Transformer-like attention in the novel policy architectures they propose to solve the specific problems posed in each paper. In this way, these papers are especially relevant to the problem we are studying in this dissertation and have greatly informed the design of our experiments as well as our methodology.

Figure 3.14.: An illustration of the attention agent from [24] playing the CarRacing game wherein it is constrained to only be able to sample a fraction of **overlapping** patches from the game screen at each time step. In each of the screens we can see the agent attending to critical visual features; the edges and corners of the race track.

In *Neuroevolution of Self-Interpretable Agents* [24] the authors seek to address the challenges of interpretability in RL. In order to address this, the authors design an attention-based agent which is constrained to only be allowed to selectively sample a fraction of each input observation in order to perform action selection.

The policy architecture proposed in [24] - illustrated in Figure 3.15 - is remarkably simple. First, the input image is segmented into N overlapping patches - this is accomplished using, for example, a square sliding window of size 7 (pixels) and a stride of 4. The overlapping pixels ensure a high correlation between adjacent patches, which supports the computation of semantically meaningful attention values, but ensures that the patches are not mutually exclusive. It is important to note that this differs from problems in which sensory signals are mutually exclusive (i.e. non-overlapping patches of pixels in this case), which is the focus of this dissertation. The patches are then flattened and linearly projected to produce key and query matrices, K and Q, respectively. The authors do not make use of a value matrix but instead compute an attention matrix,

A, which is then summed along the row dimension to produce a so-called *patch importance vector* of size N which represents the relative importance of each given patch for decision-making in the given state. The patch importance vector is then sorted and the indices of the $k < N$ patches with the largest mean attention values are mapped to feature vectors, using a lookup table, which may be hard coded or vectors of learnable parameters. The feature vectors are then concatenated and passed into an RNN controller which is used to perform action selection. Two important things to note regarding the proposed architecture are as follows. First, the index mapping operation is non-differentiable and so the network parameters are optimised using evolutionary methods as opposed to gradient descent. Secondly, as the LSTM requires a fixed-sized input, the agent is not able to process an arbitrary number of patches, as is the requirement for the problem laid out in this dissertation.

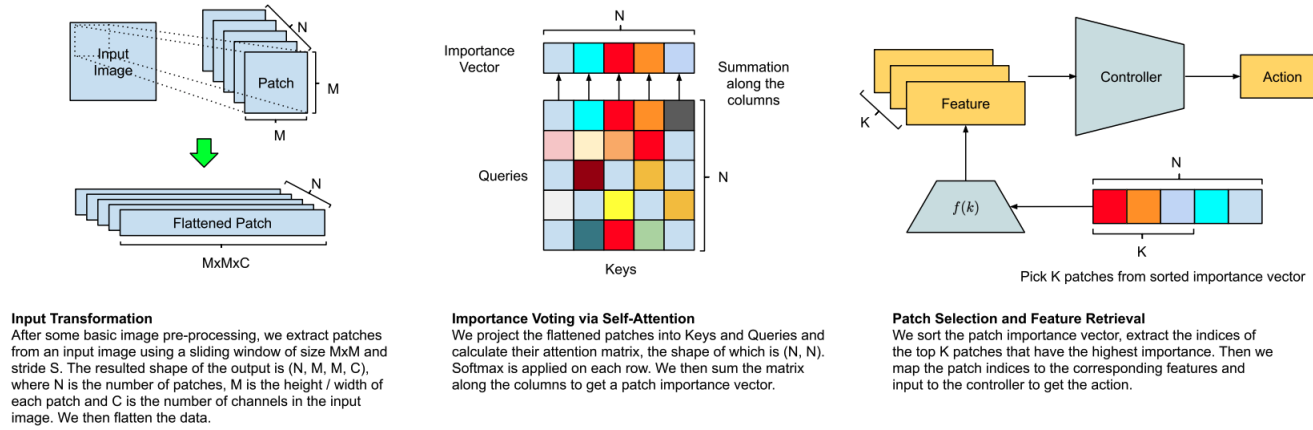


Figure 3.15.: The attention agent architecture from [24]. The flattened patches of the input image are projected into matrices of keys and queries in order to perform self-attention and produce an importance vector over all patches. The indices of the top k patches are then mapped to a set of feature vectors - the authors simply use x-y coordinate vectors - which are processed by a controller network in order to produce an action.

Taking inspiration from the field of indirect coding [93] [94], the authors hypothesise that self-attention presents a powerful mechanism for encoding a large input space into a far smaller representation and that it should, in combination with the evolutionary learning approach, allow for the learning of an optimal control policy with far fewer parameters as would be required when optimising a policy network via gradient descent. Indeed, the authors demonstrate that their attention agent is able to achieve state-of-the-art performance on the CarRacing task with 3,667 parameters, outperforming the baseline PPO agent which had 445,955 parameters. Furthermore, the authors constrain the agent by allowing it to select only $k = 10$ patches at each time step. Remarkably, the feature vectors used by the authors consisted of only the x-y indices of each patch on the grid of overlapping patches, meaning that the communication of "where", but not "what", from the self-attention mechanism to the controller was sufficient for the agent to learn a near-optimal policy, which it did after just 500 iterations of the evolutionary learning algorithm.

The main takeaways from [24] for our purposes are twofold. First, a single self-attention layer is sufficient for computing semantically rich features which may be used to solve a complex pixel-based task like CarRacing. Second, it is possible for the controller to perform optimal action selection with very limited information - in this case, simply the coordinates of the patches deemed most important by the self-attention operation. It is worth noting however, that the self-attention operation receives the full observation which it uses to compute the patch importance vector and, as such, it is unsuitable for the problem we consider in this dissertation.

In the paper *The Sensory Neuron as a Transformer: Permutation-Invariant Neural Networks for Reinforcement Learning* [25] (2021), the authors build on the Set Transformer [95], exploring systems wherein emergent complex global behaviour arises from the collective local interactions of multiple identical, but independent, agents. The study of such phenomena has inspired the development of artificial intelligence algorithms in areas such as swarm optimisation and cellular automata. Additionally, the authors take inspiration from the

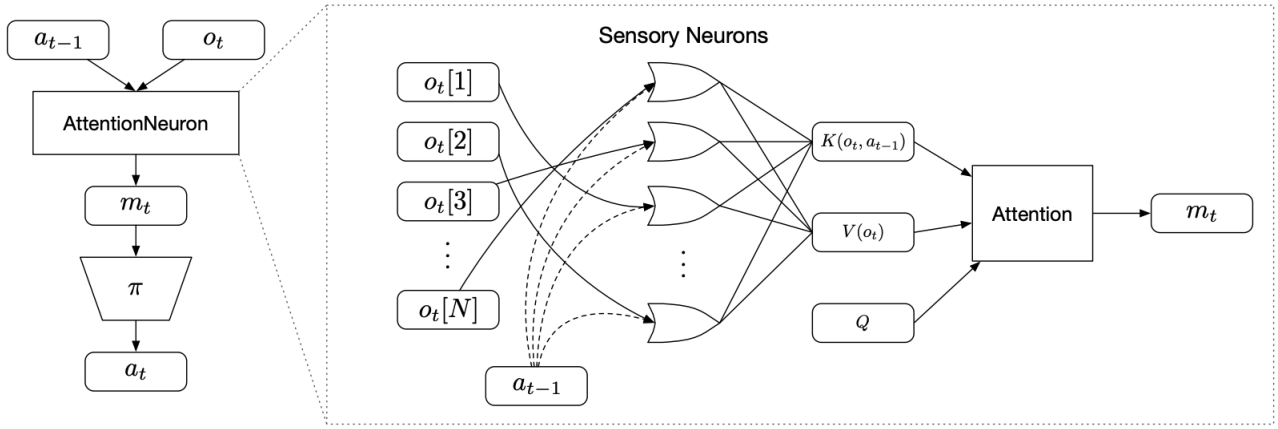


Figure 3.16.: The Permunation-Invariant Neural Network architecture proposed in [25] as an RL policy. Sensory signals, together making up o_t, along with previous actions, are processed independently by a shared network to produce embedding vectors which are fed into a self-attention mechanism in order to produce a global latent code m_t, used as input to the policy controller. The sensory embeddings are not encoded in any way, and the network architecture is agnostic to their ordering.

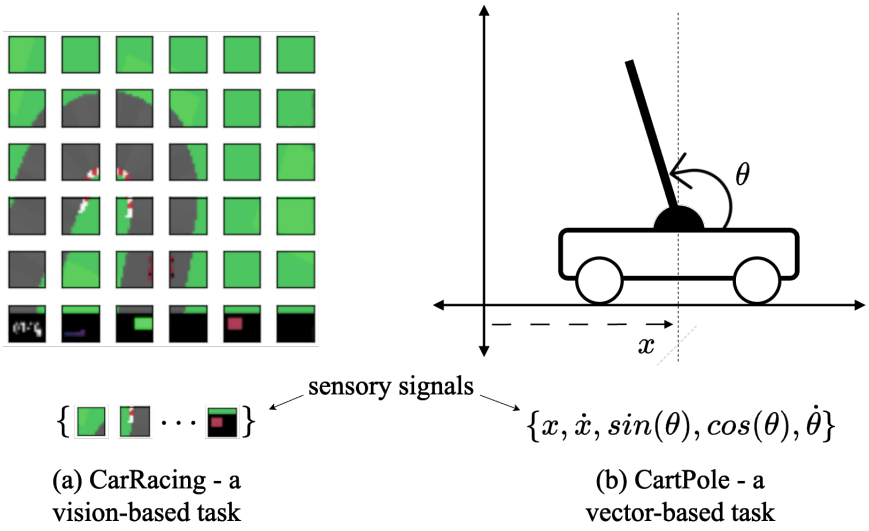
phenomenon of sensory substitution - the ability of the brain to interpret sensory signals delivered through the 'wrong' channels, such as being able to 'see' the patterns on a surface through touch, as opposed to sight. Based on this, the authors seek to develop an RL policy architecture in which arbitrary sensory signals may be processed independently and then combined in a manner agnostic to ordering (i.e. invariant with respect to the mapping between sensory channels and signals) in order to produce a coherent RL policy.

In the experiments conducted in [25], the authors consider two categories of RL tasks: vision-based and vector-based, as illustrated in Figure 3.17. In the vision-based tasks, each sensory signal corresponds to a patch of pixels in a grid which together constitute the full game screen. In the vector-based tasks, each sensory signal is a real number that may, for example, correspond to a physical quantity such as velocity or horizontal position. In both tasks, the sensory signals are ordered arbitrarily and not explicitly identifiable (e.g. the agent receives no information regarding the relative positioning of patches).

In seeking to design a policy architecture to solve such tasks, the authors propose Permutation Invariant Neural Networks (PINNs)[8], illustrated in Figure 3.16; a novel self-attention-based architecture which leverages a modified version of self-attention in the following way. The authors begin by noting that the self-attention mechanism, as illustrated in equation 2.63, is *permutation equivariant*, that is, a reordering of the rows of the input embedding matrix results in an equivalent reordering in the rows of the output matrix. As established, all that is required to make self-attention *permutation invariant* is to fix the rows of Q, as any arbitrary row ordering applied jointly to K and V only results in a reordering of the summations in each row which is by definition invariant to ordering. Having identified this, the authors opt to fix Q as a learnable parameter matrix, yielding a modified, *permutation invariant* version of self-attention which forms the core of the PINN architecture, which we now cover in detail.

The policy input as usual is an observation, o_t, which is made up of a set of sensory signals - flattened pixel patches for vision tasks and scalar values for vector tasks, denoted $o[i]$ for $i = 1, \ldots, N$. In order to produce the key matrix and value matrices, K and V, each sensory signal is processed independently by shared embedding layers, f_k and f_v. As the observations are not encoded with identifiers (e.g. positional encoding) the authors find it beneficial to generate the embedding vectors in K (1) based on the historical sequence of sensory signals, as opposed to a single signal from a single time step, and (2) by concatenating the previous action with each sensory signal before feeding it into f_k, which they find assists the agent in identifying and encoding each signal effectively. The embedding function, f_k, in the case of the vector task is an LSTM which processes signals independently as a batch to produce distinct embedding vectors. In the

[8]Not to be confused with so-called 'Physics-Informed Neural Networks' [96].



(a) CarRacing - a vision-based task

(b) CartPole - a vector-based task

Figure 3.17.: Two different RL tasks studied in [25]. (a) CarRacing - a vision-based environment in which the agent must drive a car around a variety of tracks as quickly as possible - each sensory signal corresponds to a patch of pixels in a grid which together make up the game screen, and which have no positional encoding. (b) CartPole - a vector-based environment in which the agent must apply horizontal forces to a cart in order to balance a pole vertically on its top - each sensor corresponds to some physical quantity, such as horizontal position, or angular velocity, which together describe the state of the physical system. The permutation-invariant neural network policy proposed in [25] is able to process each set of sensory signals in a manner which is agnostic to their ordering, that is, none of the signals are identified explicitly and the agent must learn to act on their combination with no additional knowledge.

vision task, f_k takes as input a number of stacked frames from preceding k time steps and computes the first-order temporal difference between them by simply subtracting the pixel values point-wise in each frame at time t with those at time $t-1$ - the authors find this gives the agent information about the environment dynamics in an efficient but effective manner. In producing V the authors simply use $f_v(x) = x$ for the vector-based tasks, whereas in vision-based tasks, f_v simply takes the stacked frames as input and returns a matrix of flattened patch vectors. The latent global latent code, m_t, is then produced via the modified self-attention layer in the following way:

$$K(o_t, a_{t-1}) = \begin{bmatrix} f_k(o[1], a_{t-1}) \\ \dots \\ f_k(o[N], a_{t-1}) \end{bmatrix} \in \mathbb{R}^{N \times d_{f_k}}$$

$$V(o_t) = \begin{bmatrix} f_v(o[1]) \\ \dots \\ f_k(o[N]) \end{bmatrix} \in \mathbb{R}^{N \times d_{f_v}}$$

$$m_t = \sigma\left(\frac{[QW_q]K(o_t, a_{t-1})W_k}{\sqrt{d_q}}\right)[V(o_t)W_v]$$

In the vector-based tasks, m_t is a single column vector ($d_v = 1$) which is then fed into a simple MLP policy. In the vision-based tasks, m_t is an embedding matrix ($d_v > 1$) which is reshaped into a 3-dimensional tensor and fed into a convolutional policy in order to force a spatial structure on the output of the modified self-attention layer.

A secondary effect of fixing Q in the modified self-attention layer, in addition to being permutation invariant, is that the network is also able to process an arbitrary number of signals in any forward pass through the layer without affecting the shape of m_t - this is relevant for our purposes since, as outlined above, we are seeking an architecture which is able to process a set of signals under various masking ratios, although we take a slightly different approach - more on this in section 4.

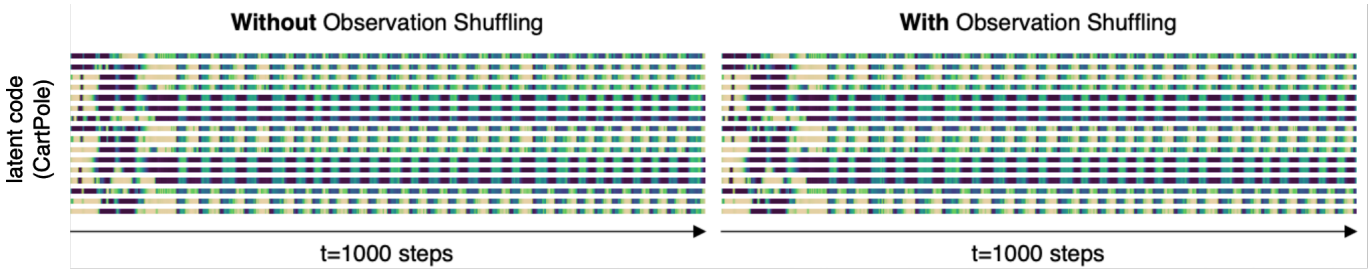


Figure 3.18.: *Permutation invariant outputs*: an illustration of the 16-dimensional global latent code m_t output from a PINN RL policy trained to play the CartPole task. Yellow represents higher values and blue for lower values. The values in m_t are (nearly) identical when sensory signals are randomly shuffled (right) vs when they are not (left).

The results of the experiments conducted by the authors revealed a number of important findings for our purposes. Firstly, the authors confirm that, as expected, the PINN policy architecture is able to produce an output unaffected by ordering and is even robust to reordering mid-episode[9]. Figure 3.18 illustrates the output of the latent code produced by a PINN policy after being trained on the CartPole (vector) environment - the latent code produced is (nearly) identical with and without shuffling of the sensory signals.

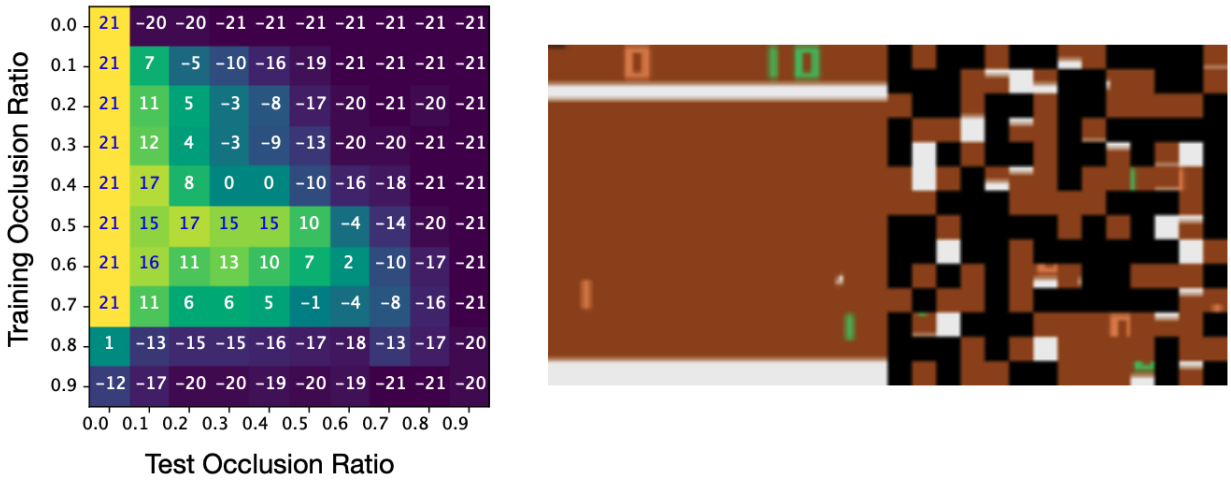


Figure 3.19.: An illustration of the PINN agent [25] trained to play Atari Pong in a modified version of the task where a fixed random subset of the patches are masked during training (right). The authors also evaluate the PINN agents under the same range of masking ratios and illustrate their returns in a heatmap (left), finding that, when evaluated under lower masking ratios than those under which they were trained, the agents are able to use the additional information in order to improve their performance.

Secondly, the authors find that PINN policies demonstrate robustness, both under conditions of high masking and when additional random vectors, constituting noise, are included with the set of sensory embedding vectors as part of the input to the policy network, which are findings particularly relevant for our purposes. As illustrated in Figure 3.19, the authors trained PINN agents to play a variant of Atari Pong wherein a fixed subset of patches are masked during training - the authors experiment with training under different masking ratios, ranging from 0% to 90%. During evaluation, the agents are evaluated over the same range of masking ratios, finding that (a) the PINN agent was able to win the game (a score greater than 0) when trained under masking ratios of up to 60%, and (b) when trained on masking ratios between 10% and 90%, the PINN agent was able to improve at evaluation time when evaluated under lower masking ratios, showing the agent more of the game screen than what it had during training - we will show later on that this is not a property that CNN policies have when trained under similar conditions.

[9]For an interactive demonstration see `https://attentionneuron.github.io/`

In all experiments, the authors compared their PINN agents to baseline agents with MLP (for vector tasks) and CNN (for vision tasks) policies. While they found that the PINN agents did *not* outperform the baseline policies under normal conditions, they far exceeded the baseline policies when the task was modified by shuffling or masking of the sensory signals, or when additional noise was added in the form of random embedding vectors. In this way, the PINN agents were shown to be far more robust and able to generalise better to novel states not encountered during training.

An important technical point to note is that the authors did not train the PINN agents using conventional gradient-based learning methods, such as DQN or PPO, but rather with either (a) evolutionary methods, using generated returns as a fitness function, or (b) behaviour cloning methods, wherein the PINN agent was trained in a supervised manner in order to 'clone' the policy of a second agent[10] which has learned to play the game proficiently. This is likely due to the fact that, as noted above, Transformer-like attention can cause instability during training and is also slower to converge than an equivalently sized CNN-based policy, an effect which would be amplified by the extreme conditions of partial observability imposed by the high masking ratios. A second point to note is that, due to the lack of positional encoding, PINN agents must be trained on a *fixed* subset of patches, and would likely not be able to handle a random sampling of patches during training – this more general version of the problem is what we aim to address in this dissertation.

3.4 An Aside: Decision Transformers

A promising avenue of research into the application of transformers to RL is that of Decision Transformers [26]. While the RL algorithms we have considered above seek to optimise policy networks via the fitting of value functions or the computation of policy gradients, the Decision Transformer, illustrated in Figure 3.20, abstracts the entire RL problem, casting it as a sequence modelling task, conditioning a self-attention-based network on entire sequences of rewards, states, and actions. Action selection is performed via causal masking, where the action for each given step is masked and its associated *desired* reward is fed as *input* into the network - the agent then selects an action in an attempt to achieve the desired reward, a strategy reminiscent of 'Upside Down RL' [97]. The Decision Transformer has been shown to be highly effective when compared to other state-of-the-art methods at both single-task RL and multi-task generalisation [98]. Additionally, adjacent to the interests of this dissertation, Decision Transformers have recently been augmented with techniques from Masked Auto-Encoders [13] demonstrating the effectiveness of training transformer-like policies using masked token prediction in the RL context [99].

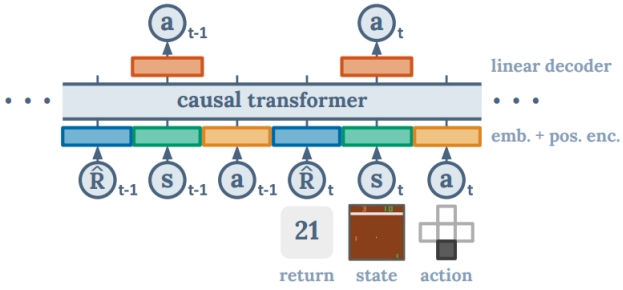


Figure 3.20.: An illustration of the Decision Transformer [26]. The Transformer policy is conditioned on entire sequences (trajectories) of states, actions, and rewards. Action selection is performed by masking the next action token in the sequence and providing its associated *desired* reward as *input* - the agent then attempts to select an action which will produce the desired reward

[10]The policy network of the 'teacher' agent in each case was always a standard feed-forward ANN (e.g. an MLP) which did not incorporate attention mechanisms in any way.

3.5 A Note Regarding Classical (Non-RL) Methods

In the classical statistical literature, there is a large body of research dating back to the mid-twentieth century regarding solutions to the so-called 'filtering problem,' which involves the estimation of true, hidden states of a dynamic system based on noisy, partial observations or measurements [100]. The class of methods designed to solve the filtering problem is called state estimation methods, the most notable of which is arguably the Kalman Filter [101].

Since the filtering problem closely resembles the problem studied in this dissertation, it is worth acknowledging the existence of this work and briefly explaining why these algorithms were not incorporated, either as baseline methods or as part of the broader theoretical foundation. Our main motivation for this decision was that classical statistical methods of this nature, developed before the era of modern deep learning, are not designed to solve the reinforcement learning problem—learning an optimal control policy from reward alone—posed by environments with high-dimensional state spaces and complex dynamics where the reward and state transition probability functions are unknown.

For example, consider the Kalman Filter, designed to estimate the hidden states of an underlying dynamical system based on partial observations and random noise [101]. This problem formulation is nearly identical to the one studied here, and it has even been studied in variations of the problem where either part [102] or all [103] of each partial observation is randomly omitted or lost. Firstly, the Kalman Filter in its original form may only be used to estimate a Markov process's true hidden state. Utilising the Kalman Filter to learn an optimal control policy would therefore require its incorporation in a larger reinforcement learning-based system that can learn policies from rewards alone, such as a larger ANN architecture. Additionally, the traditional Kalman Filter method requires a state transition model—a model of the environment dynamics—to be known in advance. Models of this nature must typically either be derived from physical laws or estimated from observation data, a task that becomes extremely challenging or even intractable when the dynamics of the system are complex and the state space is very high-dimensional, such as in the case of a video game environment. Indeed, there is an entire sub-area of research in RL, called 'model-based reinforcement learning', which is concerned with the problem of learning so-called world models (state transition models) from data, which is beyond the scope of this dissertation.

For these reasons, while it is important to acknowledge this adjacent area of research, incorporating these methods in a meaningful way would require adapting them significantly to be able to solve the task at hand.

3.6 Chapter Conclusion

In conclusion, this chapter has explored the integration of RNNs and attention mechanisms within RL policy architectures to address the challenges of partial observability. We began by examining the use of RNNs as a memory mechanism, aggregating historical information over sequences of partial observations. We also explored the application of attention mechanisms, both Transformer-like and non-Transformer-like, to enhance interpretability and facilitate optimal decision-making in the context of partial observability.

In particular, we highlighted the use of Transformer-like attention in RL policy architectures for handling partial observability across three sub-areas of research: (a) as a memory mechanism akin to an RNN, (b) for attending over entire sequences of partial observations and specific strategies used to stabilise learning in this instance, and (c) for attending over the input space to perform relational reasoning over abstract entities or to determine patch importance, often in conjunction with techniques to integrate information over time such as frame stacking or RNNs. Importantly, we identified a gap in the research: the use of Transformer-like attention to allow for variable-sized observations consisting of arbitrary subsets of the set of sensory signals that make up the full observation, which we aim to address in this dissertation.

In the next chapter, we cover our methodology, wherein we will build upon the insights gained from this review to inform the design of our experiments and the development of our attention-based RL policy architecture.

Chapter 4

Methodology

In this chapter, we discuss the methodology that underpins our experiments. In section 4.1, we start by providing a detailed overview of the problems introduced in the Introduction (section 1). Initially, we address the issue of intermittent sensory failure and latency, explaining how this scenario can be simulated in the context of reinforcement learning through the random masking of sensory signals. This approach is applied in our 'Random Masking' experiments. Subsequently, we explore the challenge of low-bandwidth communication, demonstrating how this situation can similarly be emulated in reinforcement learning by requiring our attention agent to explicitly request sensory signals from the environment at each time step. This method is employed in our 'Generated Masking' experiments.

In section 4.2, we present an overview of our proposed attention-based policy architecture, detailing its key components. This includes the distinct embedding methods we use for generating embedding vectors from sensory signals in the case of vector and vision tasks. Additionally, we explain how our architecture can be modified to generate sensor queries in the form of bit masks.

Lastly, in section 4.3, we describe the baseline policy architectures against which we will compare our proposed attention-based policy architecture. This includes a dense baseline policy for vector-based tasks and a convolutional-based policy for vision tasks. We also clearly define the three methods of imputation—zero masking, noise masking, and forward-fill masking—that we will employ in our experiments to ensure observations remain a fixed size, as required by the baseline policy architectures.

4.1 Problem Overview

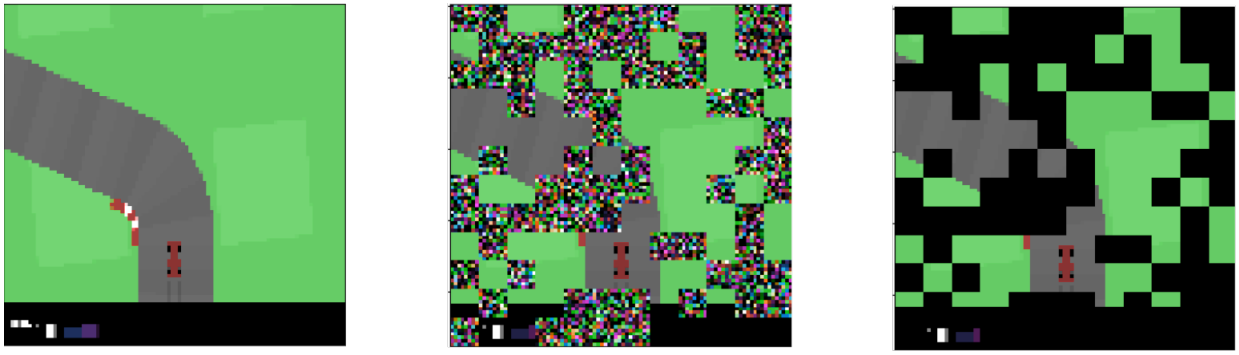
4.1.1 Sensory Failure & Latency: Random Masking

We seek to study viable strategies in the RL domain for addressing the control problem outlined in section 1.2, wherein a controller and a remote system continuously exchange control and sensory signals, respectively, in order to optimise a given process within the system, and wherein a subset of the sensory signals, sent from the system to the controller at a given time, may be:

1. **Noisy**: due to the intermittent malfunctioning of individual sensors.
2. **Unavailable**: due to intermittent variations in latency.

In order to study this problem in the context of RL, we consider the agent to be the controller and the environment to be the system. We follow [25] and define 'sensory signals', as illustrated in Figure 3.17, to be the components of a given observation emitted by the environment at a given time step. In the case of vector-based tasks, such as CartPole, each scalar value in the observation vector represents a sensory signal. In the case of vision-based tasks, such as CarRacing, the image comprising the observation is partitioned into a grid of equal-sized patches, with each patch constituting a sensory signal.

In order to emulate the phenomenon of intermittent noisy and unavailable sensory signals, we consider modified, *partially observable* versions of a set of vector and vision-based RL tasks in which a random subset



(a) full observation

(a) noisy sensory signals

(a) unavailable sensory signals

Figure 4.1.: An illustration of random masking to emulate noisy or unavailable sensory signals (i.e. image patches of pixels) in a vision-based RL environment.

of the sensory signals making up the observation is masked at each time step, according to a fixed masking ratio. This approach is similar to, and inspired by, the modified flickering environments studied in [16] as well as the partially observable version of Atari Pong, studied in [25] and illustrated in Figure 3.19, wherein a fixed subset of the image patches were masked during training.

As illustrated in Figure 4.1, the masked sensory signals may be imputed with Gaussian noise, in order to emulate noisy sensory signals, or zeros to emulate unavailable sensory signals. We will also study the impact of the forward-fill method utilised by DataProphet, a South African company that builds machine learning solutions for industrial manufacturing businesses, in which unavailable sensory signals are imputed with their most recently observed values - pixel patches in the case of images, scalars in the case of vectors - more on this below.

As we have discussed, ANN layers conventionally used for processing observations in policy networks require fixed-sized observations. Dense layers (see Figure 2.7), typically used for processing observations in vector-based tasks, perform the affine transformation $Wx + b$ and therefore require the input vectors to be of fixed size since there is a 1-1 mapping between the columns of the weight matrix W, as well as the components of the bias vector b, and the components of the input vector x. Convolutional layers (see Figure 2.12), typically used for processing observations in vision-based tasks, are characterised by sequences of *overlapping* convolutions between the input image (a grid of pixels) and a shared set of kernels. Furthermore, CNNs are typically composed of a sequence of convolutional layers, producing a feature map which is reshaped into a one-dimensional vector and fed into a Dense layer which, again, requires a fixed-sized input. Together, these two facts mean that convolutional layers are unable to process disjoint subsets of unmasked patches and require all inputs (images) to be of fixed size.

The requirement for fixed-size observations presents a distinct challenge for both dense and convolutional layers in the context of the problem outlined above. An arbitrary RL agent trained on a task in which sensory signals are subject to intermittent random masking must attempt to learn a coherent policy under significant partial observability, which presents a challenge in itself. Imposing the additional condition that all observations must be fixed-size means that, in addition to the challenge of learning under partial observability, the agent must learn to ignore the noise introduced by the imputed values.

In the case of Dense layers processing vector observations, consider a scenario wherein a missing sensory signal, $x \in [-1, 1]$, is a scalar value representing the horizontal position of an object, and where the objective of the RL task is to keep the object as close to $x = 0$ as possible (this is similar to the CartPole task). Now suppose that at a given time step the true value of the signal is $x = -0.5$, but that the sensory signal is unavailable and consequently masked such that the agent receives $x = 0$. In this scenario, the sensory value of 0 holds significant meaning in the context of the task and as such will likely have a non-trivial effect on the agent's decision-making process and a potentially negative effect on the return obtained by the agent.

In the case of Convolutional layers processing image observations, a similar problem arises when considering the scenario wherein a subset of patches is randomly masked at a given time step, as this might cause

an important object on the screen to be obscured. For example, masking the ball in Atari Pong suggests to the agent that the ball is in fact *not present* on the screen (especially considering the background is black and therefore the pixel values are zero), as opposed to merely being *un-observable* which is an important distinction. As in the case of Dense layers above, such masking would also likely have a non-trivial impact on the agent's decision-masking.

The dot product self-attention operation used in the Transformer (see equation 2.59) can process matrices with arbitrary numbers of row (embedding) vectors, presenting a promising alternative to dense and convolutional layers for addressing the problem at hand. When subject to intermittent, random masking of sensory signals, as illustrated in Figure 3.19, self-attention-based policies, while having to handle the same degree of partial observability, are not subject to the noise presented by imputation for the purposes of retaining a fixed-size input. Instead, self-attention layers are able to process *only the available information* at any given time step - this is the difference between incorrect information (e.g. masking the ball in Atari Pong) and partial information (e.g. excluding the patch with the ball altogether). Indeed, the results reported in [13] and [25] provide compelling evidence that dot product self-attention is well-suited to the task of learning to extract meaningful latent representations from images under high masking ratios.

In the experiments conducted in [25] the authors train their self-attention-based agent on vision-based tasks where a *fixed* subset of sensory signals is made available to the agent throughout training. We seek to study a generalisation of this problem, wherein, via a process of random masking, the subset of available sensory signals *changes from one time step to the next*, but where the *proportion* of masked sensory signals remains fixed. Specifically, we would like to conduct a series of experiments to study the impact of random masking, as well as various imputation strategies on conventional policy network architectures using dense and convolutional layers for processing observations, and to compare their performance to that of a novel self-attention-based policy architecture which, we will argue, presents a more general, robust alternative for solving partially-observable RL tasks of this nature.

4.1.2 Low-Bandwidth Communication: Generated Masking

An adjacent problem to that which arises from intermittent sensory failure or latency is the scenario wherein the communication channel between the controller and the system - the agent and the environment in the RL setting - has low bandwidth. That is, the rate at which it can transfer data (bits per second) is limited to the degree that the system is only able to transmit a fraction of the sensory signals at each time step.

In framing this problem in the RL context, we assume that all sensory information is available at each time step and that instead of randomly masking sensory signals to emulate intermittent sensor failure/latency, the agent must instead *generate* a query, in the form of a bit mask, to send to the environment, along with the selected action, indicating which sensory signals should be sent back to the agent in response. This scenario is similar to the partially observable task of digit classification studied in [14] in which the agent, constrained to only being able to sample local regions of the input image at any given time step, must decide which parts of the image to attend to, and in what order, in order to make a correct classification decision. In order to learn coherent policy and successfully solve the task, the agent must, in addition to learning how to map partial observations to optimal actions, learn which sensory signals are most important to sample at any given time based on the historical sequence of partial observations, and how the action of sampling sensory signals relates to the return obtained at each time step.

In our experiments, we seek to study the performance of self-attention-based policy architectures in the context of this problem. Specifically, we will examine the effectiveness of different methods of attention via mask generation, where the agent must decide at each time step which sensory signals should be included in each subsequent partial observation according to a fixed masking ratio. We will compare our proposed methods against a random baseline, which is materially equivalent to the problem of random, intermittent sensory failure/latency described above, and demonstrate that generated masking presents an improvement over random masking for our proposed self-attention-based agent.

4.2 Attention Policy Architecture

4.2.1 Overview

To address the problems outlined above, we propose the novel self-attention-based policy architecture illustrated in Figure 4.2. At a high level, the architecture has a similar form to those proposed in [14], [15], [16], [18], and [19] in that it broadly consists of three components:

1. A 'feature extraction core', which processes partial observations o_t, derived from the true environment state s_t, and produces a latent representation m_t.
2. An RNN which integrates information over time, combining m_t and h_{t-1} in order to produce an updated hidden state, h_t.
3. A policy head and a value head, which take h_t as input and produce (i) a probability distribution over possible actions, and (ii) an estimate of the value of the underlying state, respectively.

The key difference between our proposed architecture and those proposed in the aforementioned research is the utilisation of an attention block for the feature extraction core, which has a structure similar to those used in [12] (Vision Transformers), [13] (Masked Auto-Encoders), [22], and [21]; an MHA layer followed by an MLP, with interleaving LayerNorm [75] layers and residual connections [32]. The overall architecture is most similar to the architecture proposed in [22] - the variant used in the StarCraft experiments: a feature extraction core comprised of two convolutional layers and an MHA layer, followed by an RNN and a pair of MLP heads (policy and value). However, as we seek to embed each sensory signal independently, we do not include the convolutional layers for generating a set of abstract feature vectors (as in [15] and [22]). Instead, we follow [25], and process each sensory signal independently, via a shared embedding layer, in order to produce a set of mutually exclusive sensory embedding vectors, to which positional encoding vectors may be added before being fed into the attention block. A final detail is the use of the mean pooling operation to combine the set of vectors produced by the attention block; the decision to use mean pooling follows the work in [13], where instead of using a learnable $[class]$ token to aggregate information across embedding vectors, as proposed in [12], the authors found that aggregation via mean-pooling worked just as well when conducting their linear probing and fine-tuning image classification experiments. In this way, in addition to being *permutation invariant* in a manner equivalent to the architecture proposed in [25], our attention architecture can process *an arbitrary number of sensory embedding vectors*. In the experiments outlined below, we leverage this property and apply a distinct sensor mask ψ_t at each time step, t, in order to mask a fixed percentage of sensory signals by excluding their associated embedding vectors entirely from the forward pass through the network (this is not the case for the baseline architectures due to the requirement of fixed-sized observations - more on this below). We cover each aspect of the attention policy in detail below, including details regarding how the architecture may be adjusted so that it might be applied to either vector or vision-based tasks.

4.2.2 Embedding, Positional Encoding, & Masking

In this section, we detail the methods used for embedding sensory signals derived from vector and image observations in vector and vision-based tasks, respectively. We will also detail the positional encoding used in the vision case, as well as the process of masking.

Vector Embedding

In vector tasks, each observation is a vector of N scalar values, $o = (o^{(1)}, o^{(2)}, \dots, o^{(N)})$, each representing a sensory signal. We seek to transform each vector observation into an embedding matrix, $E \in \mathbb{R}^{N \times L}$, where each row vector is an embedding vector of dimension L corresponding to a single component in o. To do so, we construct a diagonal matrix, O, from our observation as:

$$O = \text{diag}(o^{(1)}, o^{(2)}, \dots, o^{(N)}) = \begin{bmatrix} o^{(1)} & & \\ & \ddots & \\ & & o^{(N)} \end{bmatrix} \tag{4.1}$$

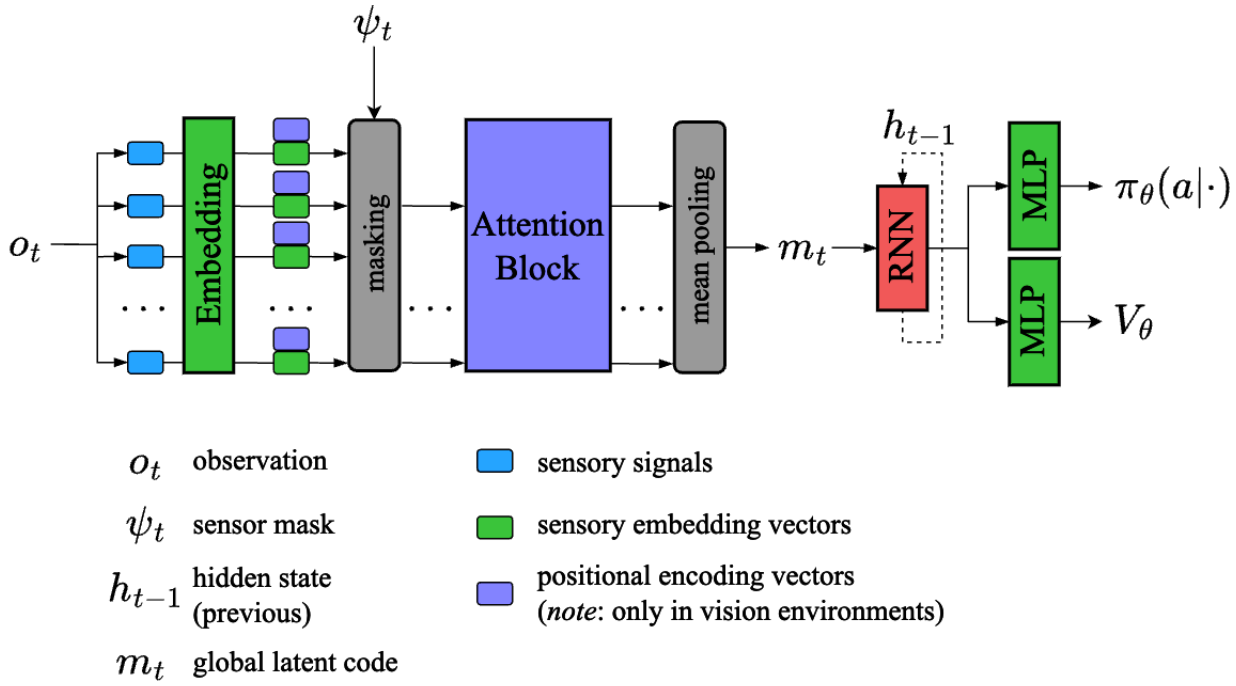


Figure 4.2.: The proposed self-attention-based policy architecture. The four main components include (i) an embedding layer which transforms individual sensory signals in o_t into a matrix of embedding vectors to which masking may be applied to exclude the embedding vectors corresponding to masked signals, (ii) an attention block which consists of a multi-head attention layer followed by an MLP, and a final mean pooling operation which produces a global latent code vector m_t capturing the salient features of the available sensory signals, (iii) an RNN layer which aggregates information over time allowing the agent to handle the partial observability induced by the masking, and (iv) a policy head and a value head for performing action selection and state value estimation.

In this way, O may be viewed as a batch of row vectors, where the i^{th} row vector has the scalar value $o^{(i)}$ in the i^{th} position and zeros everywhere else. We then compute the embedding matrix, E, by passing the row vectors of O through a dense embedding layer, f_{embed}, such that the embedding vector for the sensory signal $o^{(i)}$ - the i^{th} row vector of E - is computed as:

$$E_{i,1:L} = f_{embed}(o^{(i)}) = WO_{i,1:N} + b$$

where $W \in \mathbb{R}^{L \times N}$ and $b \in \mathbb{R}^L$ are the parameters of f_{embed}. In this way, the embedding vector corresponding to the i^{th} sensory signal is the i^{th} column of W, scaled by $o^{(i)}$, with the bias vector, b, being shared by all embedding vectors. The parameters of the embedding layer are optimised jointly with the parameters of the policy network via backpropagation.

Image Embedding

In vision tasks, we take each observation to be a pixel grid, $o \in \mathbb{R}^{H \times W \times C}$, which may consist of one or many frames stacked along the channel dimension. In what follows we make two assumptions about o:

1. We assume the image is square (i.e. $H = W$).
2. We assume the image is a single grayscale frame ($C = 1$).

Both of these assumptions are only for the sake of simplicity and may be relaxed with some minor adaptations if necessary, that is, they are not strict conditions for the successful implementation of the method

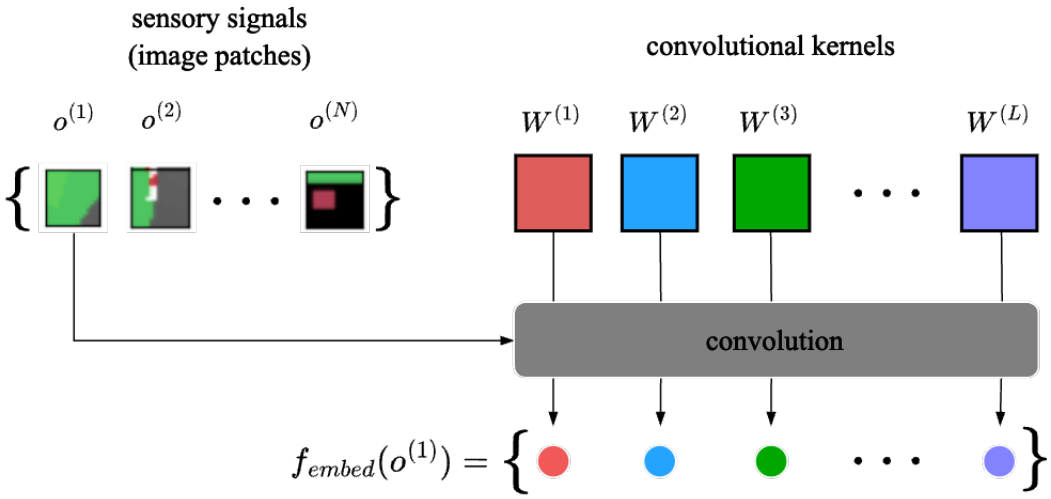


Figure 4.3.: An illustration of linear patch embedding of an image using a single convolutional layer in which the horizontal and vertical stride of the convolution operation is equal to the patch size, such that the convolutions are non-overlapping. For each of the N patches, the set of convolutions with each of the L convolutional kernels produces an embedding vector of dimension L.

described here. Following [25], we divide o into a grid of N non-overlapping image patches, $o^{(i)} \in \mathbb{R}^{k \times k}$, such that $N = (H/k)^2$, where each patch is considered a distinct sensory signal. To generate an embedding vector for each image patch, we follow [13], utilising a single convolutional layer, f_{embed}, with kernels $W^{(j)}$, $j = 1, 2, \ldots, L$, and perform a sequence of non-overlapping convolutions over o by setting the horizontal and vertical stride to be equal to the patch width/height k - see Figure 4.3 for an illustration. In this way, the i^{th} embedding (row) vector in the embedding matrix, $\tilde{E} \in \mathbb{R}^{N \times L}$, is computed by convolving the i^{th} image patch with each of the L kernels:

$$\tilde{E}_{i,1:L} = f_{embed}(o^{(i)}) = (o^{(i)} * W^{(1)}, \ldots, o^{(i)} * W^{(L)})$$

Thus, we avoid the computational overhead of having to reshape the image into a batch of image patches followed by a flattening of each patch in order to pass the flattened patches through a dense layer, which would be materially equivalent to the convolutional embedding method described above, but less efficient.

Positional Encoding

Unlike convolutional layers, Transformer self-attention does not have a spatial inductive bias, which accounts for the property of permutation equivariance discussed above. In the case of vector-based tasks, our embedding method assigns each sensory signal a corresponding column vector of learnable weights in the weight matrix of the dense f_{embed} layer which serves to identify the sensor itself and to encode the value of the signal at each time step. Moreover, there is no notion of ordering or spatial structure in vector observations. As such, there is no need to add positional encoding vectors to the matrix of embedding vectors in the case of vector-based tasks before feeding it into the attention block. In the case of vision-based tasks, however, the inherent spatial structure over the image patches must be encoded explicitly by adding to $\tilde{E}$ a matrix of positional encoding vectors, $U \in \mathbb{R}^{N \times L}$, where the i^{th} row encodes the position of the image patch which has its embedding vector in the i^{th} row of $\tilde{E}$.

We employ the 2D positional encoding method used in [13]. In this method, each image patch is assigned coordinates (row, col), where row and col are the indices of the vertical and horizontal positions of the image patch in the grid, respectively. Each positional encoding vector, $u^{row,col} \in \mathbb{R}^L$, is a concatenation of two equal-sized vectors, $u^{row} \in \mathbb{R}^{L/2}$ and $u^{col} \in \mathbb{R}^{L/2}$. In a similar manner to the 1D positional encoding employed in [11], u^{row} is a vector of sinusoids, evaluated at different frequencies as:

$$u^{row}_{1:L/4} = \big(sin(\omega_1 \cdot row), \ldots, sin(\omega_{L/4} \cdot row)\big)$$
$$u^{row}_{L/4:L/2} = \big(cos(\omega_1 \cdot row), \ldots, cos(\omega_{L/4} \cdot row)\big)$$
$$u^{row} = \text{concatenate}(u^{row}_{1:L/4}, u^{row}_{L/4:L/2})$$

where

$$\omega_k = \frac{1}{1000^{k/(L/4)}}$$

The vector, u^{col}, encoding the horizontal position of the image patch is computed in the same way, and the final positional encoding vector for an image patch positioned at (row, col) in the grid is constructed as $u^{row,col} = \text{concatenate}(u^{row}, u^{col})$. When computed in this way, the cosine similarity between each given positional encoding vector and all others when plotted in a grid as a heat map, produces the illustration shown in Figure 2.25, where adjacent and surrounding image patches have high similarity which diminishes in a manner approximately proportional to the euclidean distance between patches on the grid.

The i^{th} row of U is then the vector $u^{row,col}$ which encodes the 2D position of the image patch, $o^{(i)}$, at (row, col) in the grid corresponding to the embedding vector in the i^{th} row of $\tilde{E}$. As per [13], each positional encoding vector is added element-wise to its corresponding embedding vector in order to produce a final embedding matrix ready to be processed by the attention block:

$$E = \tilde{E} + U$$

In this way, the columns of the projection matrices W_q, W_k, and W_v belonging to each attention head are able to extract both 'what' (visual) and 'where' (spatial) features from the rows of E since the inner products between the projection and embedding matrices will be determined partly by visual features in $\tilde{E}$ extracted by the convolutional embedding layer, and partly by the spatial encoding in U.

Masking

In all experiments outlined below, we utilise masking to emulate (a) the intermittent failure or latency of sensory signals, and (b) sensory queries emitted by the agent to the controller in the 'low-bandwidth channel' scenario. In practice, this is implemented as follows. At each time step, t, we have a bit mask, $\psi_t \in [0, 1]^L$, which is generated either by the environment (random) or by the agent (non-random), where the i^{th} bit in ψ_t corresponds to the i^{th} row of E and hence the i^{th} sensory signal with 0 indicating that the sensory signal should be masked. In each experiment conducted, we set a masking ratio, $\mu^{train} \in [0, 1)$, which determines the percentage of sensory signals which are masked at each time step throughout training, and hence the percentage of 0s in each ψ_t. Through our experiments, we seek to examine the effects of different masking ratios on the performance of the various policy architectures studied.

Regarding practical implementation, in the case of the proposed attention agent, the process of masking simply involves excluding the rows of E for which the corresponding bits in ψ_t are zero, which is possible since the self-attention layer is able to process embedding matrices with arbitrary numbers of rows. We will cover the practical details of masking in the case of the baseline agents below.

4.2.3 Attention Block

The form of the attention block used in the proposed attention policy is identical to that used in the ViT [12] - see Figure 2.23 - and in the MAE [13]; multi-head (self) attention (MHA) [11], per equation 2.64, followed by a shared 2-layered MLP which transforms each row vector in the matrix produced by the self-attention layer independently (i.e. as a batch). A LayerNorm layer [75] proceeds both the MHA and MLP components and each is followed by a residual connection [32]. We also experiment with replacing the residual connection with a GRU gate as proposed in [21] - more on this below.

Two important hyperparameters which must be preset for the MHA layer, and which determine the capacity of the model, are the parameters d_{model}, the embedding dimension of the model, and h, the number of attention heads. As usual, the i^{th} attention head performs the dot-product attention operation:

$$\text{head}_i = \text{Attention}(E_t W_i^Q, E_t W_i^K, E_t W_i^V) \tag{4.2}$$

$$= \text{softmax}\left(\frac{E_t W_i^Q (E_t W_i^K)^T}{\sqrt{d_{model}}}\right) E_t W_i^V \tag{4.3}$$

where E_t is assumed to be the matrix of all *unmasked* embedding vectors at a given time step, t, and where $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{L \times M}$ are the query, key, and value projection matrices for the i^{th} attention head, each of which projects the rows of E_t into an embedding space with dimension $M = \frac{d_{model}}{h}$.

As in equation 2.64, the outputs of the attention heads are finally concatenated column-wise and projected into an output embedding space by a weight matrix, $W^O \in \mathbb{R}^{d_{model} \times d_{model}}$, in order to produce a matrix with $\lceil (1 - \mu^{train}) \cdot N \rceil$ rows, where N is the total number of sensory signals (masked and unmasked), and d_{model} columns.

The 2-layered MLP performs the transformation $f : \mathbb{R}^{d_{model}} \rightarrow \mathbb{R}^{d_{model}}$ on each row vector in the matrix produced by the MHA layer, with the hidden layer having a number of neurons proportional to d_{model} (usually a multiple of 2), which is an additional hyperparameter set per experiment. We apply a ReLU activation function to the activation values output by the hidden layer. Finally, we follow [13], and perform a mean-pooling operation over the batch of feature vectors produced by the MLP to produce the *global latent code*, $m_t \in \mathbb{R}^{d_{model}}$.

In this way, the attention block takes as input the embedding vectors corresponding to the available (unmasked) sensory signals derived from each environment observation o_t emitted at each time step t, performing the attention operation outlined above and producing a global latent code m_t which encodes the salient features gleaned from the available sensory information. The hope here is that the attention operation over *only available sensory signals* will be able to produce a latent code which allows for increased robustness to performance degradation under increasingly high masking ratios when compared with the baseline methods described below, each of which requires imputation to satisfy the requirement of fixed-sized observations.

4.2.4 Recurrent Layer

The masking of sensory signals induces partial observability which may be extreme in the case of high masking ratios (i.e. $\mu^{train} > 0.7$). For this reason, we follow previous work in which RL agents were trained under partial observability [14][16] and include an RNN layer with hidden dimensionality d_{model}, which takes as input at each time step, t, the global latent code, m_t, containing salient features gleaned from the unmasked sensory signals by the attention block, along with the hidden state from the previous time step, h_{t-1}, which contains feature information aggregated across the agent's historical trajectory of partial observations, to produce an updated hidden state, $h_t \in \mathbb{R}^{d_{model}}$. In this way, the hidden state of the RNN is akin to the belief the agent holds regarding the true state of the underlying MDP.

In our experiments, we choose the Gated Recurrent Unit (GRU) [60] as the RNN layer, as it has been shown empirically to be comparable in performance to the LSTM, but more efficient [58]. We acknowledge that much of the related work on the application of self-attention in POMDP RL examined above cite as motivation for their work shortcomings in RNNs as a memory mechanism for dealing with partial observability [20][23][21][92], however, (a) our work is not directly concerned with this problem, but rather with the problem of variable sized partial observations arising from sensory masking, and (b) the use or RNNs for handling partial observability in RL is well-established and remains widely used in the literature, as pointed out by [21]. For these reasons, while future work might consider using an attention mechanism in place of an RNN in order to aggregate information over time, as in [20], we will utilise RNNs for experiments conducted in this dissertation.

4.2.5 Policy & Value Heads

The proposed attention policy architecture is an actor-critic network similar to that proposed in [20] and, as such, it has both an actor and a critic head. Both the actor and the critic head take as input the updated

hidden state, h_t, from the RNN layer and produce a probability distribution over possible actions, $\pi_\theta(a|h_t)$, and a value estimate of the underlying state of the MDP, $V_\theta(h_t) \sim v^\pi(s_t)$. Both the actor and critic head take the form of a 2-layered MLP where the number of neurons in the hidden layer is proportional to d_{model} (in all our experiments we use a multiplier of 2). Again, we apply a ReLU activation function to the activations in the hidden layer.

4.2.6 Mask Head

In the low-bandwidth scenario described above, we would like to allow the agent to query the environment for specific sensory signals by generating, at each time step, the bit mask, ψ_t, as opposed to having it be randomly generated by the environment. To this end, we append an additional masking head onto the attention policy which, as with the policy and value heads, takes the form of a 2-layered MLP with a number of neurons in the hidden layer proportional to d_{model} (in our experiments we use a multiplier of 2) and a ReLU action applied to the values output from the hidden layer.

Mask generation happens in the following way. At each time step the agent generates, from h_t, a multinomial distribution $(p_1, \dots, p_N)$ over the N sensors [1]. Then, we sample **without replacement** according to μ^{train} (μ^{eval} during evaluation). In this way, we generate the bitmask, ψ_t, with $M = N - \lceil \mu^{train} * N \rceil$ 1s and compute the probability:

$$
\begin{aligned}
Pr(\psi_t|h_t) &= f(x_1, \dots, x_N; M, \beta_1, \dots, \beta_N) && (4.4)\\
&= \frac{M!}{x_1! \dots x_N!} p_1^{x_1} \times \dots \times p_N^{x_N} && (4.5)\\
&= M! \times p_1^{x_1} \times \dots \times p_N^{x_N} && (4.6)
\end{aligned}
$$

where f represents the probability mass function for the multinomial distribution, p_i is the probability of the i^{th} sensory signal being sampled, and $x_i \in \{0, 1\}$ is the number of times the i^{th} sensory signal is drawn, which may be either 0 or 1 since we are sampling without replacement.

The parameters of the mask head are optimised jointly with the rest of the policy network. This is achieved by viewing mask generation and action selection as a joint 'action' and computing their joint probability in the following way. Note that since both $\pi(a_t|h_t)$ and $Pr(\psi_t|h_t)$ depend directly on h but are computed independently of one another via separate heads, we have conditional independence [104] meaning we can compute the probability of the joint action as:

$$
Pr(a_t, \psi_t|h_t) = \pi(a_t|h_t) \cdot Pr(\psi_t|h_t) \tag{4.7}
$$

Note that in practice, for numerical stability, we work with log probabilities wherever possible, and the joint probability is computed using a sum of log probabilities instead. In this way, as we use PPO to train the agents in all experiments, $Pr(a_t, \psi_t)$ may be used instead of $\pi(a_t|h_t)$ for computing the gradient of the PPO objective 2.54.

4.3 Baseline Policy Architectures

We seek to study the effects of sensory masking and various imputation methods on a set of baseline policy architectures using conventional dense and convolutional layers, which require fixed-sized observations, comparing their performance to that of our proposed attention agent. To this end, we modify the policy architecture shown in 4.2 by replacing the embedding layer and the attention block in order to construct two baseline policy architectures of comparable size (with respect to the number of network parameters): a dense agent, for vector-based tasks, and a convolutional agent, for vision-based tasks.

[1] We use the *distrax* package which has convenient implementations of probability distributions in Jax.

4.3.1 Dense Agent

In constructing the dense agent, we replace the embedding layer and attention block with a 2-layered MLP which performs the transformation $f : \mathbb{R}^N \rightarrow \mathbb{R}^{d_{model}}$ (in our experiments we set $d_{model} = 128$), and where the number of neurons in the hidden layer is proportional to d_{model} (in all our experiments we use a multiplier of 2 and apply a ReLU activation function to the activation values output by the hidden layer). In this way, the dense agent produces a latent code, $m_t \in \mathbb{R}^{d_{model}}$, at each time step, t, equal to the dimension of the latent code produced by the attention agent, which may be passed into the RNN layer and on to the network heads in the manner described above.

4.3.2 Convolutional Agent

In constructing the convolutional agent, we replace the embedding layer and attention block with a convolutional network consisting of two convolutional layers followed by a dense layer, which is structurally equivalent to the DQN network proposed in [9], used to train agents to play Atari 2600 games. In all our experiments, we set the number of kernels in the first convolutional layer to 32, and the horizontal and vertical stride to 4. Each kernel in the first convolutional layer is in $\mathbb{R}^{k \times k}$ and where k is equal to the patch size (height/width) used for segmenting input images in the case of the attention agent (in our experiments we set $k = 8$). We set the number of kernels in the first convolutional layer to 64, and the horizontal and vertical stride to 2. Each kernel in the second convolutional layer is in $\mathbb{R}^{4 \times 4}$. The output of the second convolutional layer is reshaped to form a 1-dimensional feature vector which is transformed by a dense layer to produce the latent code, $m_t \in \mathbb{R}^{d_{model}}$ (in all our experiments we set $d_{model} = 128$). We apply a ReLU activation function to the output activations of each convolutional layer, as well as to the output of the dense layer.

4.3.3 Masking & Imputation

As established above, both dense and convolutional layers require fixed-sized inputs. Consequently, masked sensory signals may not be removed entirely from the input to either the dense or convolutional agent's policy networks. Instead, we are required to impute - that is, substitute - the masked sensory signal with some other value(s). We experiment with three methods of imputation:

1. **Noise Masking**: the values making up each masked sensory signal - scalars in the case of vector-based tasks and each individual pixel in the image patch in the case of vision-based tasks - are imputed with Gaussian noise with mean 0 and variance 1. The intention here is to emulate a faulty sensor. The mean and variance selected for generating the noise produce values which are the range of the values which make up the observations in both the Acrobot environment, where the majority of the values are in the range $[-1, 1]$, and the CarRacing environment, where we normalise the pixel values to be within the range $[0, 1]$.

2. **Zero Masking**: the values making up each masked sensory signal are imputed with zeros. This is a naive method of imputation for dealing with missing values.

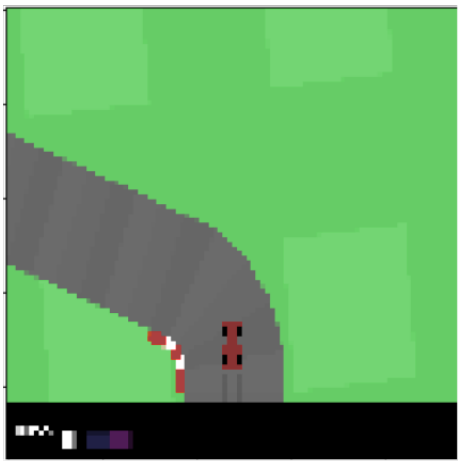
3. **Forward-fill Masking**: the values making up each masked sensory signal are imputed with the values of the most recently observed sensory signals. This is the method employed by the industry experts we consulted, DataProphet, when dealing with missing sensory data.

The practical implementation details of masking and imputation are as follows. At each time step, t, the environment emits an observation, o_t, and a randomly generated bit mask, ψ_t. Additionally, we have an imputation tensor, $\tilde{o}_t$, which is the same shape as, o_t, and which is filled with imputation values; Gaussian noise, zeros, or values belonging to the the most recently observed sensory signal. In the case of vector-based environments, the bit mask and the observation are one-dimensional vectors of equal size, where the i^{th} bit in ψ_t corresponds to the i^{th} sensory signal (scalar) in o_t. In the case of vision-based tasks, we broadcast the bit mask to fill a 2-dimensional tensor of the same shape as o_t - we will denote this broadcast tensor as ψ_t for consistency, we assume the meaning will be evident from the context in what follows - where the i^{th} bit fills

the values in the 2D mask which correspond to the image patch which is the i^{th} sensory signal. We can then compute the masked and imputed observation as:

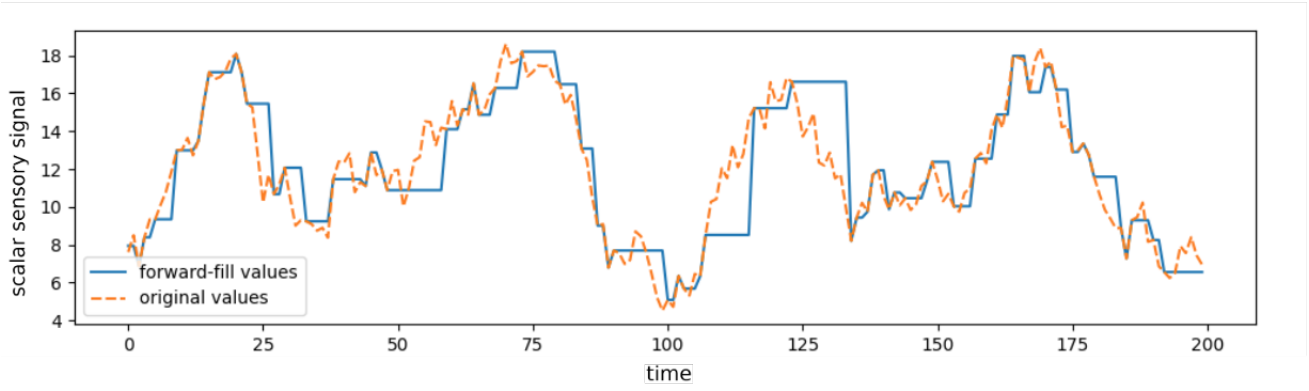
$$\bar{o}_t = o_t \odot \psi_t + \tilde{o}_t \odot (1 - \psi_t) \tag{4.8}$$

where $\odot$ denotes an element-wise multiplication. The agent then observes $\bar{o}_t$ as opposed to o_t. The effect of this operation in the case of a vision-based task is illustrated in Figure 4.1 (b) and (c), which illustrate noise and zero masking, respectively. The forward fill imputation method is illustrated below in Figure 4.4 (a) for vision-based tasks and in 4.4 (b) for vector-based tasks - both produced based on a masking ratio of 50%. In both cases, the forward-fill method provides a fair approximation of the original observation with some distortion.



(a.1) full observation
(vision-based task)

(a.2) forward-fill
(vision-based task)



(b) actual signal vs forward fill
(vector-based task)

Figure 4.4.: An illustration of forward-fill masking with a masking ratio of 50% in (a) a vision-based task wherein each image patch constitutes a sensory signal and, (b) a vector-based task wherein each scalar value in the vector constitutes a sensory signal. In both cases, the values of masked sensory signals are imputed with their most recently observed values. In the vision case, this results in a somewhat distorted, but still comprehensible, version of the original image, while in the vector case, the time series associated with each sensory signal tends towards having constant values for periods of time, followed by a step change to a new value, and so on.

Chapter 5

Experiments

In this section, we will detail all experiments conducted in an attempt to answer our research questions which, as a reminder, are as follows:

1. *In the 'sensor failure and latency' scenario, what is the manner and extent of performance degradation suffered by conventional policy network architectures using the imputation methods of zero masking, noise masking, and forward-fill masking under different rates of sensor failure and latency?*
2. *What advantages and disadvantages might self-attention-based policy network architectures offer over conventional policy network architectures in such conditions?*
3. *In the 'low-bandwidth communication channel' scenario, what advantages and disadvantages are offered by the non-random generation of sensor queries over the random alternative?*

With this in mind, we have chosen to conduct two major sets of experiments. In our first set of experiments - 'Random Masking' - we train both our attention agent and the baseline agents, detailed in sections 4.3.1 and 4.3.2, on adapted versions of two test environments wherein a fixed proportion of the sensory signals are randomly masked at each time step, to emulate signal failure or latency. In these experiments, we will seek to study the effect of different degrees of partial observability induced by different masking ratios, as well as the various imputation methods, on the agents' performance.

In our second set of experiments - 'Generated Masking' - we utilise a similar experimental framework, but instead of having the mask be randomly generated by the environment, we equip the agent with the ability to generate the mask and, in so doing, query specific sensory signals at each time step.

The chapter is structured as follows. Section 5.1 gives an overview of the test environments used in our experiments. Then, sections 5.2 and 5.3 detail the design of our Random Masking and Generated Masking experiments, respectively. Finally, section 5.4 covers some of the implementation details including hyperparameter values and an overview of the software and hardware used.

5.1 Test Environments

In this section, we detail the test environments used in the experiments that follow. We have opted to experiment using one vector-based task as well as one vision-based task in order to study the problem outlined above since pixel patches and scalar values corresponding to physical quantities represent distinct categories of sensory signals, and there will likely be distinct insights to be drawn from each category regarding the research questions posed above.

5.1.1 Acrobot - Vector Task

The Acrobot environment models a physical system consisting of two links connected to form a chain, with one end of the chain fixed to a wall, but free to rotate. The joint between the two links is actuated, meaning that it has within it a component, such as a motor which is able to exert force and cause movement. The goal

of the agent is to apply torque on the actuated joint at each time step, either in the clockwise or anti-clockwise direction, in order to swing the free end of the linear chain above a given height while starting from the initial state of hanging downwards. In order to achieve the objective, the agent must explore several possible action sequences in order to swing the end of the chain up to the goal height. There are several canonical vector-based environments which have been implemented in a unified framework called Gym (or Gymnasium) by OpenAI - the reason we chose Acrobot is that it has the highest number of components of all the 'Classic Control' tasks which is useful for experimenting with a range of masking ratios. We describe each aspect of the environment in detail below.

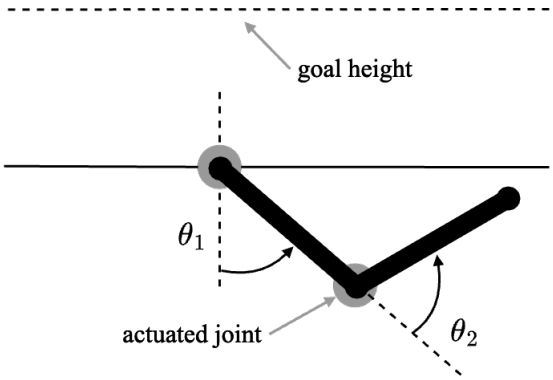


Figure 5.1.: An illustration of the Acrobot task.

Observation space The observation is a vector of dimension 6 which encodes information about the two rotational joint angles as well as their angular velocities. θ_1 is the angle of the first joint, where an angle of 0 indicates the first link is pointing directly downwards. θ_2 is the angle relative to the angle of the first link. Each angle is encoded by taking the sine and cosine of each to avoid the discontinuity between 2π and 0 radians. The angular velocities of θ_1 and θ_2 are bounded at $\pm 4\pi$, and $\pm 9\pi$ radians per second, respectively, for reasons to do with numerical stability[1]. An illustration of the system may be seen in Figure 5.1, and the details of the observation vector are tabulated in Table 5.1.

Index	Quantity	Min	Max
1	$\cos(\theta_1)$	-1	1
2	$\sin(\theta_1)$	-1	1
3	$\cos(\theta_2)$	-1	1
4	$\sin(\theta_2)$	-1	1
5	$\dot{\theta}_1$	-4π	4π
6	$\dot{\theta}_2$	-9π	9π

Table 5.1.: The observation space of the Acrobot-v1 (vector) environment. Note: $\dot{\theta}$ denotes the angular velocity.

Action space The action space consists of three discrete actions, each representing the torque applied on the actuated joint between the two links. The details of each action are tabulated in Table 5.2

Rewards The goal is to have the free end of the chain reach a designated target height in as few steps as possible, and as such all steps that do not reach the goal incur a reward of -1. Achieving the target height results in termination with a reward of 0. The reward threshold - the episodic return above which the task is considered solved - is -100.

Starting states Each parameter in the underlying state (θ_1, θ_2, and the two angular velocities) is initialized uniformly between -0.1 and 0.1. This means both links are pointing approximately downwards with some initial stochasticity.

[1]The official documentation for the environment may be found here: `https://gymnasium.farama.org/environments/classic_control/acrobot/`.

Index	Description
1	apply -1 ($N \cdot m$) torque to the actuated joint
2	apply 0 ($N \cdot m$) torque to the actuated joint
3	apply 1 ($N \cdot m$) torque to the actuated joint

Table 5.2.: The action space of the Acrobot-v1 (vector) environment.

Episodic termination The episode ends if: the free end reaches the target height (a terminal state), which is constructed as $-cos(\theta_1) - cos(\theta_2 + \theta_1) > 1.0$, or if the episode continues for more than 500 time steps.

5.1.2 CarRacing - Vision Task

CarRacing, illustrated in Figure 4.1 (a), is a video game environment in which the agent's task is to drive a racing car around a winding track as quickly as possible without going off course. The track is randomly generated in each episode.

Observation space The observation space is simply an RGB pixel grid of shape $(96, 96, 3)$ which constitutes the game screen depicting the racing car, the track, and the surrounding grass. However, in our experiments, we convert the screen to grayscale for efficiency, meaning that each observation has shape $(96, 96)$.

Action space The action space consists of five discrete actions giving the agent the ability to accelerate, brake, and steer the car. The details of each action are tabulated in Table 5.3

Index	Description
1	do nothing
2	steer left
3	steer right
4	accelerate
5	brake

Table 5.3.: The action space of the CarRacing-v2 (vision) environment.

Rewards The racing track is divided into tiles, with the division between each tile cutting across the width of the track, meaning that driving forward along the track one would move from one tile to the next. The agent's objective is to visit all the tiles on the track as quickly as possible without going off course. The reward received by the agent is -0.1 for every episodic time step (frame) and $\frac{1000}{N}$ for every track tile visited, where N is the total number of tiles in the track. For example, if the agent completes a track with 732 tiles in 732 time steps (frames), the total episodic reward received by the agent is $1000 - 0.1 * 732 = 926.8$.

Starting states At the beginning of each episode, the car starts at rest in the centre of the road on a randomly generated track.

Episodic termination The episode ends if: the agent visits all tiles on the track, or if the car goes too far off the track, in which case the agent will receive a large negative reward of -100 and the episode will terminate.

5.2 Random Masking Experiments

Table 5.4 below details the Random Masking experiments run for all agents across both test environments. In our Random Masking experiments, we train our attention agent and the baseline agents on a range of masking ratios μ^{train} spanning from 0% (0/6 sensors) to 83% (5/6 sensors) in the case of the Acrobot (vector) task, and from 0% to 98% in the case of CarRacing (vision). Note that we ran the Random Masking experiment for the CarRacing environment under a 98% masking ratio to see whether the performance of the baseline methods would break down under conditions of extreme noise and partial observability - we stopped at 90% in the case

of the Generated Making experiments detailed in section 5.3 below. Regarding the baseline agents, we train both the dense and convolutional baseline agents under the three methods of imputation outlined above: zero masking, noise masking, and forward-fill masking.

We train all agents using the PPO algorithm for a fixed number of time steps - see section 5.4.1 for details - which we selected based on trial runs of the experiments, selecting a number of training steps where convergence was more or less observed for agents trained under full observability (a 0% masking ratio). In our view, this and the fact that the number of training steps is held constant across the board (for all agents, and for all masking ratios) is sufficient for the purposes of our comparative study.

During evaluation time, we evaluate all agents, each trained under a fixed μ^{train}, for a fixed number of time steps across the full range of masking ratios, denoted μ^{eval}. This method of evaluation follows [16] where the authors studied MDP to POMDP generalisation (and vice versa) of their DRQN agent, to assess the agents' abilities to generalise to varying degrees of partial observability induced by μ^{eval} which differ from the degree of partial observability induced by μ^{train} during training. The ability of an agent to generalise to scenarios with greater or lesser degrees of partial observability is important to study in the context of the problem we are examining since it's not possible to fully predict which scenarios exactly the agent might encounter at inference time.

Importantly, all results reported are averaged across 5 random seeds to ensure statistical validity (as far as possible)[2].

Agent	#	Environment	Imputation method
Attention Agent	1 (a)	Acrobot	-
	1 (b)	CarRacing	-
Baseline Agent (Dense)	1 (c)	Acrobot	Gaussian noise
	1 (d)	Acrobot	zeros
	1 (e)	Acrobot	forward-fill
Baseline Agent (Conv)	1 (f)	CarRacing	Gaussian noise
	1 (g)	CarRacing	zeros
	1 (h)	CarRacing	forward-fill

Table 5.4.: Random Masking Experiments

Environment	Training Steps	Evaluations Steps	μ^{train}/μ^{eval}
Acrobot	8×10^6	4×10^4	$\{0/6, 1/6, 2/6, 3/6, 4/6, 5/6\}$
CarRacing	12×10^6	8×10^4	$\{0.0, 0.1, 0.3, 0.5, 0.7, 0.9, 0.98\}$

Table 5.5.: Random Masking Experiments: the total number of training and evaluation steps across all environments, and the set of masking ratios used for both training and evaluation.

5.3 Generated Masking Experiments

In our Generated Masking experiments, detailed in Table 5.6 below, we perform experiments 1(a) and 1(b), detailed in Table 5.4 above, again but instead of having ψ_t be generated randomly by the environment, it is generated by the agent at each time step based on h_t in the manner described above.

5.4 Implementation Details

[2]We would have liked to perform additional runs, however, due to constraints on computation resources and budget this was not possible.

Agent	#	Environment	Masking
Attention Agent	2 (a)	Acrobot	agent-generated
	2 (b)	CarRacing	agent-generated

Table 5.6.: Generated Masking Experiments

Environment	Training Steps	Evaluations Steps	μ^{train}/μ^{eval}
Acrobot	8×10^6	4×10^4	$\{0/6, 1/6, 2/6, 3/6, 4/6, 5/6\}$
CarRacing	12×10^6	8×10^4	$\{0.0, 0.1, 0.3, 0.5, 0.7, 0.9\}$

Table 5.7.: Generated Masking Experiments: training steps, evaluation steps, and masking ratios per environment.

5.4.1 Training Algorithm: PPO

Both our attention agent and the baseline agents have actor-critic policy architectures. PPO is designed to be a stable actor-critic learning algorithm which works well out of the box. We, therefore, opt to use PPO for all our experiments. However, we note that results obtained might not necessarily generalise under value-based methods like DQN - this should be noted as a limitation of this research and an item for future work.

We adapt state-of-the-art PPO implementations from a set of third-party sources, detailed in section 5.4.3. All hyperparameter values are detailed in Table 5.8 below. While we performed test runs and conducted some minor hyperparameter tuning (e.g. for the number of training steps), we found the hyperparameters used in the baseline implementation of PPO on which ours was (indirectly) based [105] worked well, likely because the authors have taken care to re-implement PPO according to the original implementation by the authors [106], against which they have rigorously benchmarked their implementation.

5.4.2 Agent Hyperparameters

Table 5.9 below details the hyperparameters used for the attention and baseline agent architectures for both test environments. The hyperparameter d_{model} refers to the dimension of the embeddings in the attention agent, as well as the hidden dimension of the RNN in both the attention and baseline agents. For the baseline agents d_{model} also represents the dimension of the vector produced by the network core - the MLP in the case of the dense agent, and the CNN in the case of the convolutional agent.

5.4.3 Software & Hardware Details

Language and Libraries All code is written in Python 3.9. RL agent and PPO implemented in Jax 0.4.20, Flax 0.7.5. Environments using Envpool 0.8.4 (CarRacing), Gymnax 0.0.1 (Acrbobot).

Hardware Experiments run in Google Colab on Nvidia V100 and on Puzl.cloud on an Nvidia A100.

Third-party code PPO implementation adapted from PureJaxRL [107] and CleanRL [105], each of which contain state of the art implementations based on the literature, and were originally based on OpenAIs Baselines [106] implementation following their publication of the PPO paper [70]. Aspects of the attention agent implementation were adapted to Jax from the original PyTorch implementation [108] of the MAE from Facebook AI Research, including the convolutional embedding layer, the 2D positional encoding method, and the structure of the attention block.

Hyperparameter	Type	Acrobot	CarRacing	Description
Training steps	integer	8×10^6	12×10^6	Total number of environment steps during training
Number of environments	integer	4	8	Number of parallel environments with which to collect experience
Steps per environment	integer	2×10^6	1.5×10^6	The number of training steps per parallel environment
Batch size	integer	128	128	The number of time steps to run per gradient update
Number of mini-batches	integer	4	4	The number of mini-batches into which the collected experience is divided and used for a single PPO update
α	float	2.5×10^{-4}	2.5×10^{-4}	The (initial) learning rate which determines the magnitude of the Gradient update
Anneal learning rate	boolean	true	true	Whether or not to decay the learning rate throughout training according to a linear schedule
γ	float	0.99	0.99	The discount parameter for computing returns
λ^{GAE}	float	0.95	0.95	The coefficient for computing the generalised advantage estimate
Clip ϵ	float	0.1	0.2	Used to clip r_θ in the PPO objective 2.54
Entropy coefficient	float	0.01	0.01	The coefficient for an additional *entropy* term added to the total PPO loss which rewards the agent for maintaining a level of entropy in $\pi_\theta(a\|\cdot)$ for the purposes of exploration.
Value loss coefficient	float	0.5	0.5	The coefficient for the value loss term in the total PPO loss, which takes the form 2.52
Maximum gradient norm	float	0.5	0.5	A threshold for the norm of gradients computed from the PPO objective function, above which gradients are scaled by multiplying them by the threshold value divided by the norm of the gradient

Table 5.8.: PPO Hyperparameters

Agent	Hyperparameter	Type	Acrobot	CarRacing
Attention Agent	d_{model}	integer	128	128
	Patch size	integer	-	8
	number of attention heads	integer	8	4
	MLP hidden dim ratio (attention block)	integer	2	2
	MLP hidden dim ratio (policy heads)	integer	1	1
	activation	function	ReLU	ReLU
Dense Agent	d_{model}	integer	128	-
	MLP hidden dim ratio (network core)	integer	2	-
	MLP hidden dim ratio (policy heads)	integer	1	-
	activation	function	ReLU	-
Convolutional Agent	d_{model}	integer	-	128
	Patch size (for random masking)	integer	-	8
	number of kernels (conv layer 1)	integer	-	32
	kernel dimension (conv layer 1)	integer	-	8
	stride (conv layer 1)	integer	-	4
	number of kernels (conv layer 2)	integer	-	64
	kernel dimension (conv layer 2)	integer	-	4
	stride (conv layer 2)	integer	-	2
	MLP hidden dim ratio (policy heads)	integer	-	1
	activation	function	-	ReLU

Table 5.9.: Agent Hyperparameters

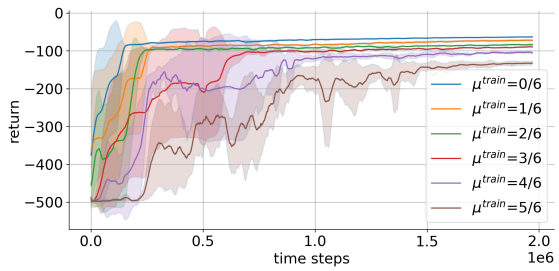
Chapter 6

Results

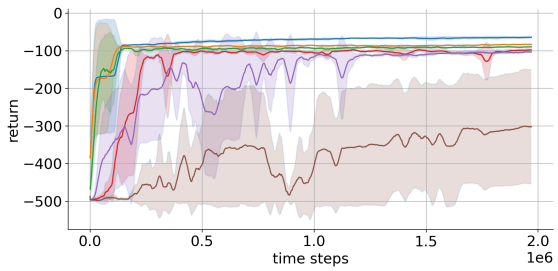
In this chapter, we provide a detailed overview of the results from our experiments. In sections 6.1 and 6.2, we discuss the outcomes of the Random Masking experiments, as detailed in section 5.2, and the Generated Masking experiments, as detailed in section 5.3, respectively. In both cases, we examine the data recorded during both training and evaluation by all agents across both environments and conduct additional analyses on data generated during the evaluation by each agent.

6.1 Random Masking

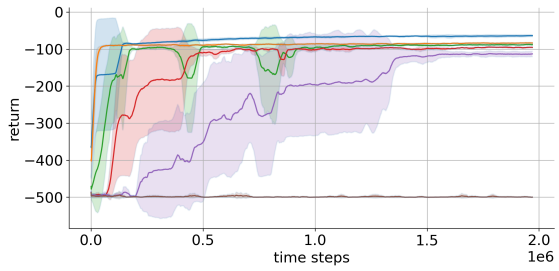
6.1.1 Acrobot - Vector Task



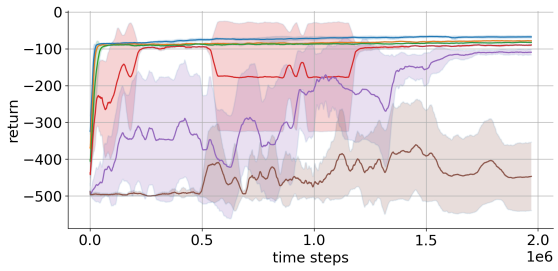
(a) Attention Agent



(b) Dense Agent - Zero Masking



(c) Dense Agent - Noise Masking



(d) Dense Agent - Forward-Fill Masking

Figure 6.1.: Episodic returns obtained during training in the Acrobot-v1 environment during our Random Masking experiments by our attention agent and the baseline dense agents trained under fixed masking ratios $\mu^{train} = 0/6, 1/6, \ldots, 5/6$.

Figure 6.1 illustrates the episodic returns obtained during training on the Acrobot-v1 task for our attention agent and the dense baseline agent under zero masking, noise masking, and forward-fill masking, trained under fixed masking ratios $\mu^{train} = 0/6, 1/6, \ldots, 5/6$. Note that tabulated results are given in the Appendix. When examining these plots, we are looking to assess how each of the following aspects of the agents' learning are affected under increasing μ^{train}:

1. Mean episodic return: the mean episodic return obtained by the agent towards the end of training.
2. Sample efficiency: the rate (return per time step) at which the agent learns
3. Stability: the degree to which the mean return generated by the agent throughout training gradually increases and then converges, as opposed to fluctuating up and down.

Examining the returns generated by our attention agent (Figure 6.1 (a)) during training we observe the following. The attention agent is able to learn to perform the task successfully under all masking ratios, converging to a maximum mean return of -64 under $\mu^{train} = 0/6$ [1]and -122 under $\mu^{train} = 5/6$ towards the end of training. Note that each increase in μ^{train} corresponds to a lower maximum mean return as might be expected, as the agent has access to less information at each time step and must rely more and more on the recurrent layer to aggregate information obtained from partial observations over time. Sample efficiency also degrades with each increase in μ^{train}. Convergence occurs in $\approx 0.25e6$ steps for $\mu^{train} = 0/6$ to $\mu^{train} = 2/6$, increasing to $\approx 0.7e6$ for $\mu^{train} = 3/6$, $\approx 1e6$ for $\mu^{train} = 4/6$, and $\approx 1.5e6$ for $\mu^{train} = 5/6$. Learning stability also appears to degrade reliably during training with each increase in μ^{train}, with instability being most evident for $\mu^{train} = 5/6$ where the mean return fluctuates throughout training, producing sharp increases and decreases intermittently until convergence.

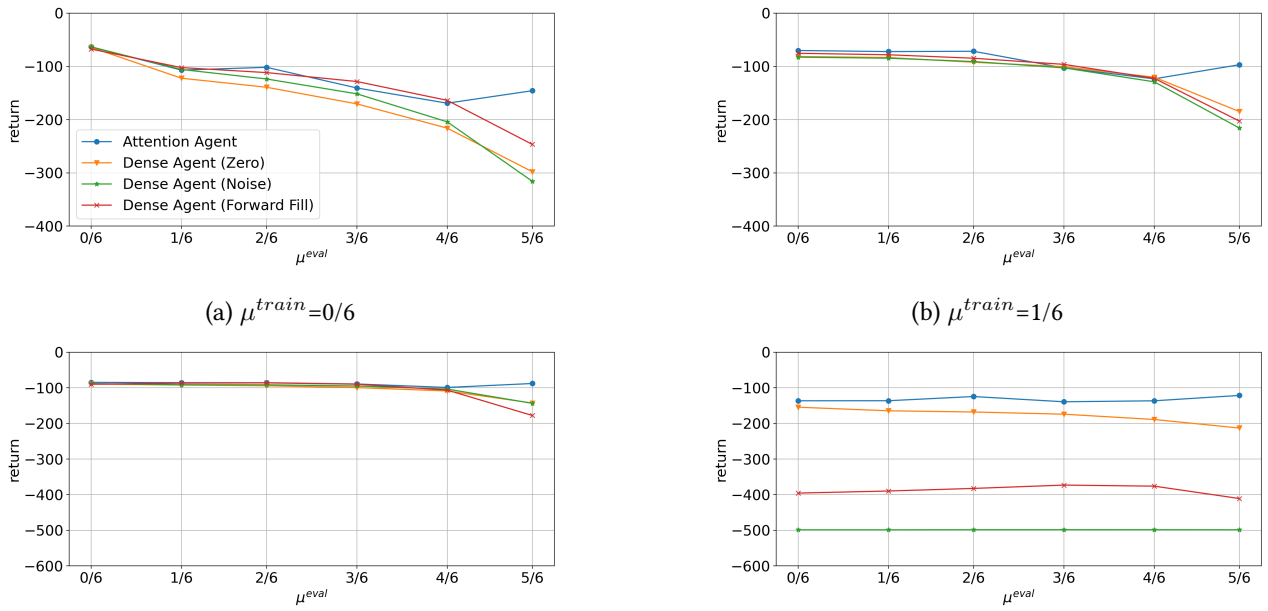
Examining the returns generated by the baseline dense agent under zero masking, noise masking, and forward-fill masking (Figures 6.1 (b), (c), and (d), respectively) during training we observe the following. For $\mu^{train} = 0/6$ the baseline dense agent variants are essentially equivalent (as the agent receives the full observation at each time step) and each converges to a mean return between -63 and -67. From $\mu^{train} = 1/6$ to $\mu^{train} = 4/6$ the dense agents are ultimately able to learn to perform the task successfully by the end of training, achieving scores comparable to that of our attention agent. However, each of the dense agent variants struggles to make progress learning under $\mu^{train} = 5/6$, with the zero masking variant scoring a mean return between -200 and -400 by the end of training, the forward-fill variant scoring between -400 and -500, and the noise masking agent scoring -500, failing to make any progress learning whatsoever. The dense agent variants exhibit similar sample efficiency to our attention agent up until $\mu^{train} = 3/6$. For $\mu^{train} = 4/6$, convergence is only obtained at $\approx 1.2e6$, $\approx 1.4e6$, and $\approx 1.6e6$ time steps by the zero masking, noise masking, and forward-fill masking agents, respectively. Instability also clearly increases with increasing μ^{train}. The noise masking agent exhibits the most stability up until $\mu^{train} = 4/6$. The zero masking variant exhibits stable learning up until $\mu^{train} = 3/6$, whereafter we observe significant fluctuations in the mean returns obtained under both $\mu^{train} = 4/6$ and $\mu^{train} = 5/6$ throughout training. Finally, the forward-fill variant appears to exhibit the least stability with significant fluctuations observed even for $\mu^{train} = 3/6$, and severe fluctuations for $\mu^{train} = 4/6$ and $\mu^{train} = 5/6$.

In the case of the Acrobot-v1 task, zero masking appears to degrade the performance of the dense agent the least of all the masking (imputation) variants of the baseline dense agent. Interestingly, noise masking appears preferable to forward-fill masking with respect to stability and sample efficiency, but fails to learn entirely for $\mu^{train} = 5/6$. This is likely because, in the case of forward-fill masking under $\mu^{train} > 3/6$, having in-distribution values which are roughly correlated with the true values frustrates the learning process as the values may be easily misinterpreted by the agent (statistically speaking) as true when they are false/delayed. Gaussian noise on the other hand is entirely uncorrelated with the true values of each observation, which the agent appears to be able to learn to ignore, up until $\mu^{train} = 5/6$ at which point the noise becomes overwhelming.

In general, the ability of all agents to make progress learning at all under high masking ratios is further evidence of the effectiveness of RNNs - especially those with gating mechanisms such as the GRU - in dealing with partial observability. It is clear, however, that the ability to process only available sensory signals enabled by the self-attention layer offers a distinct advantage over having to process noise constituted by imputed values, as in the case of the baseline dense agents, in the Acrobot-v1 (vector) task.

As outlined in the Methodology - section 4 - we would like to evaluate the ability of our attention and the baseline agent variants to generalise to masking ratios other than the one under which they were trained.

[1]According to the OpenAI leader board (`https://github.com/openai/gym/wiki/Leaderboard`) the highest score reported for which evidence is provided is a mean return of -61.8 (4th place).



(a) μ^{train}=0/6

(b) μ^{train}=1/6

(c) μ^{train}=3/6

(d) μ^{train}=5/6

Figure 6.2.: Generalisation to novel masking ratios on Acrobot-v1 (vector): episodic returns generated by our attention agent and dense baseline agents (zero, noise, and forward-fill masking), each trained under a fixed masking ratio μ^{train}, when evaluated across a set of increasing masking ratios μ^{eval}.

Given the problem we are studying, this is an especially important ability to asses, as it's unlikely that the probability of sensor failure will be uniform across all sensors, nor is it likely that the proportion of missing/-corrupted sensory values will remain consistent across time, meaning that the conditions during training are likely to be different than those at inference time. As such it is useful to asses how varying the masking ratio during evaluation affects the episodic returns generated by each agent.

Figure 6.2 illustrates the mean episodic returns generated by our attention agent and the baseline dense agent variants, trained under fixed μ^{train}, when evaluated under a set of increasing evaluation masking ratios $\mu^{eval} = 0/6, 1/6, \ldots, 5/6$. Considering $\mu^{train} = 0/6$ (Figure 6.2 (a)) we see that, while the mean return of all agents reliably decreases as μ^{eval} increases, the attention agent is the most resistant to the degradation at the extreme, obtaining a mean return of -148 under $\mu^{eval} = 5/6$. The forward-fill masking variant of the baseline dense agent performs comparably to the attention agent up until $\mu^{eval} = 4/6$, but obtains a mean return of just -251 for $\mu^{eval} = 5/6$. The mean returns of the zero masking and noise masking variants of the baseline dense agents each appear to degrade significantly faster than the forward-fill variant as μ^{eval} is increased, obtaining mean returns of -290 and -315 for $\mu^{eval} = 5/6$, respectively.

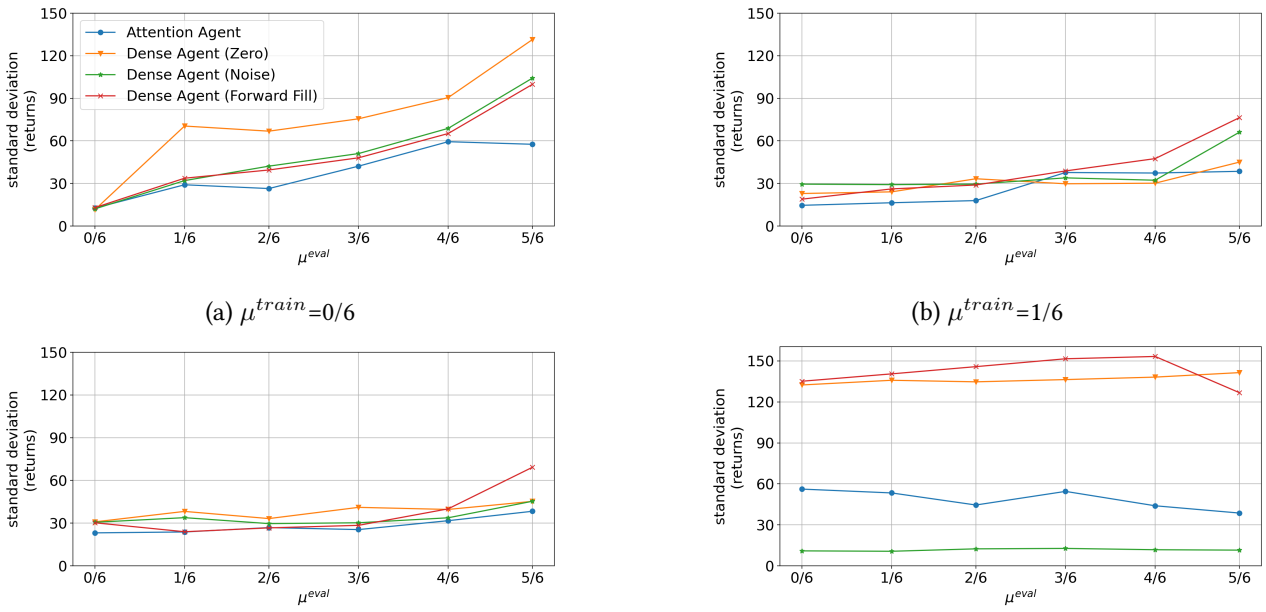
The difference in the ability to generalise between the baseline agents under the different masking variants for $\mu^{train} = 0/6$ is likely accounted for by the fact that forward-fill masking ensures the imputed values are in-distribution with respect to the data on which the agent was trained, as opposed to zero and noise masking, each of which produces noisy observation vectors which are increasingly out-of-distribution with respect to the training data as μ^{eval} is increased.

Next, considering the returns generated during evaluation by each of the agents trained under $\mu^{train} = 1/6$ and $\mu^{train} = 3/6$ (Figure 6.2 (b) and (c), respectively), we observe the following. Exposure to a moderate degree of partial observability and noise (in the case of the baseline dense agents) during training greatly improves the ability of all agents to generalise at evaluation time, allowing each to handle greater degrees of partial observability and noise than those encountered during training ($\mu^{eval} > \mu^{train}$). In both cases, however, the attention agent outperforms all baseline dense agent variants when evaluated under $\mu^{eval} = 5/6$, suffering little-to-no degradation in mean episodic return generated relative to $\mu^{eval} = 0/6$.

Finally, we consider the evaluation returns obtained by all agents when trained under $\mu^{train} = 5/6$. As might be expected from analysing the training curves in Figure 6.1, our attention agent performs the best,

obtaining a mean episodic return of ≈ -130 for all values of μ^{eval}. The zero masking variant of the baseline dense agent produces results which are better than expected, obtaining a mean episodic return of -155 under $\mu^{eval} = 0/6$ and -217 under $\mu^{eval} = 5/6$, demonstrating an ability to benefit from the additional information when evaluated under lower masking ratios than that under which it was trained ($\mu^{eval} < \mu^{train}$). The forward-fill baseline dense agent suffers a massive drop in mean returns generated across all μ^{eval} (relative to those observed for $\mu^{train} < 5/6$), which is likely the result of extreme delays in sensory signals confounding the agent during action selection. Consistent with results observed during training, the noise masking variant of the baseline dense agent appears unable to complete the task successfully in under 500 time steps.

A key observation we make from analysing the data in Figure 6.2 is that it appears as though masking unavailable sensory signals with zeros allows the baseline agent to effectively treat zero-valued signals as absent. A plausible reason for this is because the true value of any given signal is almost certainly never exactly 0, a fact which may be exploited in the transformation of the masked observation by the policy network in order to achieve an effect similar to that which arises naturally the attention agents policy architecture when missing sensory signals are excluded entirely from the input.



(a) μ^{train}=0/6

(b) μ^{train}=1/6

(c) μ^{train}=3/6

(d) μ^{train}=5/6

Figure 6.3.: Standard deviations of the distributions of the returns - illustrated in Figure 6.2 - obtained during evaluation on Acrobot-v1 by our attention agent and dense baseline agents (zero, noise, and forward-fill masking), each trained under a fixed masking ratio μ^{train}, and evaluated across a set of increasing masking ratios μ^{eval}.

For the purpose of fair evaluation and comparison, as well as to assess robustness and reliability, it is useful to consider the standard deviation of the distribution of returns generated by each of the agents during evaluation[2]. Figure 6.3 illustrates the standard deviations of the distribution of returns produced by agents trained under fixed μ^{train} and evaluated across $\mu^{eval} = 0/6, 1/6, \ldots, 5/6$ [3]. In the plot, lower standard deviations are generally more desirable, as this indicates the agent's performance is consistent for fixed μ^{eval}, even if the mean of the distribution of episodic returns decreases for increasing μ^{eval}.

The highest standard deviations were produced by each agent respectively when trained under $\mu^{train} = 0/6$, with the values increasing reliably across the board from $\mu^{eval} = 0/6$ to $\mu^{train} = 5/6$. The most extreme standard deviations are produced by the baseline dense agents: ≈ 130 for the zero masking agent and ≈ 100 for both the noise masking and forward-fill masking agents under $\mu^{eval} = 5/6$. The attention agent proved to

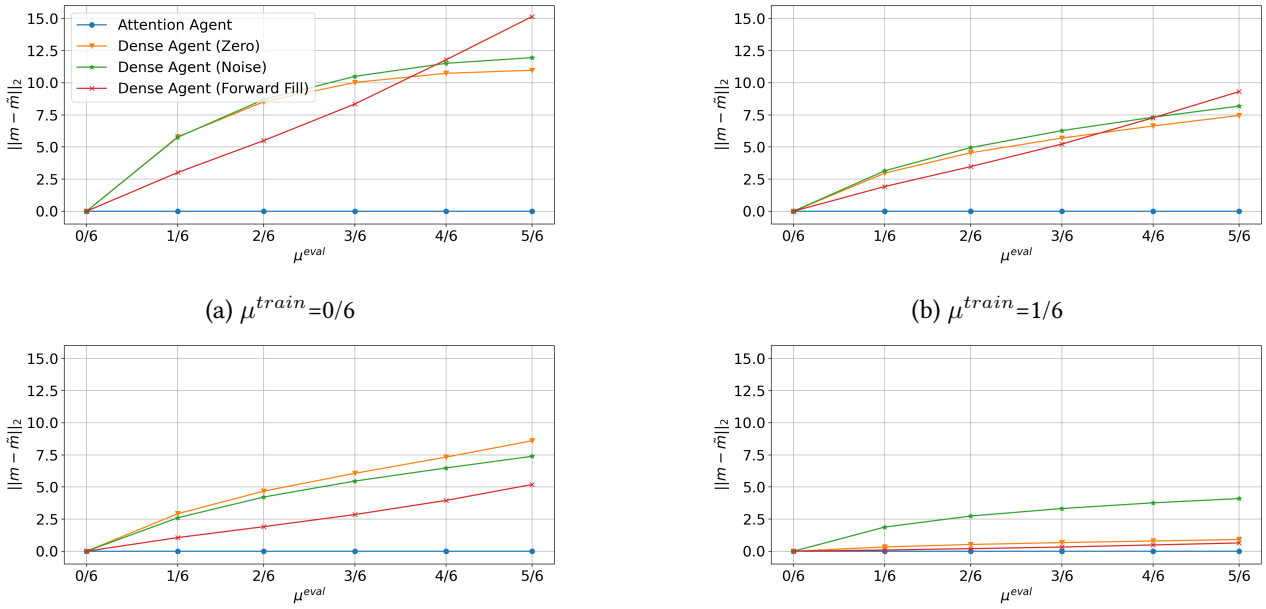
[2]Indicating the standard deviations with vertical error bars in Figure 6.2 made the plot very difficult to ready, so we split the mean and standard deviation into two plots.

[3]Note: the standard deviations are tabulated with the evaluation results in the Appendix.

be more consistent across the board, producing the lowest standard deviation across all μ^{eval} and a maximum of ≈ 60 under $\mu^{eval} = 5/6$.

When trained under $\mu^{train} = 1/6$ and $\mu^{train} = 3/6$ the standard deviations produced by all agents are far lower in general. This is consistent with the data illustrated in Figure 6.2 which shows all agents generalise far better when exposed to moderate degrees of partial observability and noise (in the case of the baseline dense agents) during training, which in turn appears to translate to more consistent, less variable episodic returns. In both cases when evaluated under under $\mu^{eval} = 5/6$ our attention agent produces a lower standard deviation than the baseline dense agents, while the forward-fill baseline agent produces the largest standard deviation, once again demonstrating that our attention agent is more consistent with respect to returns generated under high, novel rates of sensor failure.

Finally, the standard deviations produced by the baseline dense agents trained under $\mu^{train} = 5/6$ are the most extreme of all. Maximum standard deviations of 144 (under $\mu^{eval} = 4/6$) and 150 (under $\mu^{eval} = 5/6$) were produced by the zero masking and forward-fill masking variants of the baseline dense agent, respectively, which is consistent with high variance in the returns generated by these agents during training as illustrated in Figure 6.1. Meanwhile, our attention agent produced standard deviations of <60 across the board, demonstrating significantly higher levels of consistency and robustness relative to the baseline dense agents. The noise masking variant of the baseline agent has the lowest standard deviation of all, but this is only because it consistently produces returns of ≈ -500 for all μ^{eval}.



(a) μ^{train}=0/6

(b) μ^{train}=1/6

(c) μ^{train}=3/6

(d) μ^{train}=5/6

Figure 6.4.: An illustration of the mean L2 norm of the vectors computed by taking the difference between the latent code vectors produced from unmasked and masked observations, m_t and $\tilde{m}_t$, respectively, from the Acrobot-v1 environment. The curves are plotted for our attention agent and all baseline dense agents, each trained under fixed masking ratios μ^{train}, across a set of increasing masking ratios μ^{eval}. Since the underlying observation is the same, it is desirable to see as small a change in the L2 norm as possible as μ^{eval} is increased.

To handle increasing degrees of partial observability, each agent, broadly speaking, has two tools at its disposal. First, the RNN can be optimised to filter and aggregate noisy signals from across time to form a belief about the true current state of the underlying MDP - this phenomenon has been studied (e.g. in the case of DRQN [16]) and is a feature we hold constant in our experiments between our attention and baseline agents. A second tool the agent might use to handle partial observability is the network core which processes the masked observations to produce the latent code m_t. In the case of the baseline dense agent, the network core can potentially be optimised to extract meaningful features from the observation whilst 'ignoring' the

noise contributed by the imputed values. Since the focus of our experiments is the comparison between our attention-based network core and the baseline dense (MLP) network core, it makes sense to examine the question: *given a fixed, unmasked observation o_t and its associated latent code m_t, to what degree does the latent code $\tilde{m}_t$, produced from o_t following masking, differ from m_t under masking ratios $\mu^{eval} = 0/6, 0.1, \ldots, 5/6$?*

In answering this question, we examine the data illustrated in 6.4 wherein we plot the mean L2 norm $||m_2 - \tilde{m}_t||_2$ of the difference between latent codes produced for an unmasked batch of observations (m_t), and the latent codes produced from the same batch of observations with masks applied ($\tilde{m}_t$) across all μ^{eval}, for all agents trained under fixed masking ratios $\mu^{train} = 0/6, 1/6, 3/6, 5/6$.

Considering the baseline dense agent, we make two key observations. First, the norms increase reliably as μ^{eval} increases for all masking variants, across all μ^{train}. One interesting point here is that the mean of the norms appears to plateau for the zero and noise masking variants of the baseline dense agent, trained under $\mu^{train} = 0/6$, at around $\mu^{eval} = 3/6$, while the mean of the norm generated by the forward-fill agent appears to increase linearly. This might be explained by the fact that under forward-fill masking, increasingly novel (out of distribution) masked observation vectors are produced as μ^{eval} is increased due to the increase in latency, resulting in increasingly novel combinations of sensory signal values. By contrast, in the case of noise and zero masking, there is a point of saturation whereafter imputing additional zeros/values drawn from the standard normal distribution has only a marginal effect on m_t. The second key observation is that the mean norm of the difference between the masked and unmasked latent codes decreases reliably for agents trained under larger μ^{train}, across *all* μ^{eval} - this suggests that the MLP network core in the policy network of the baseline dense agents is indeed able to learn to filter out noise and produce less variable latent codes for the same underlying observation under different masking ratios.

Considering the attention agent, we observe that the norms produced are consistently small (< 1) across all μ^{train} and μ^{eval}[4]. This might be accounted for by (i) the absence of noise as suffered by the baseline agents (ii) the LayerNorm layers in the attention block, which preceded the MHA layer and the MLP, and which ensure the activation values are small and centred about 0, and (iii) the mean pooling operation which averages over all embedding vectors produced by the attention block. Regardless of the reason, the subtle change in the latent code vector across masking ratios likely contributes to the robustness and consistency of the attention agent when evaluated under novel degrees of partial observability.

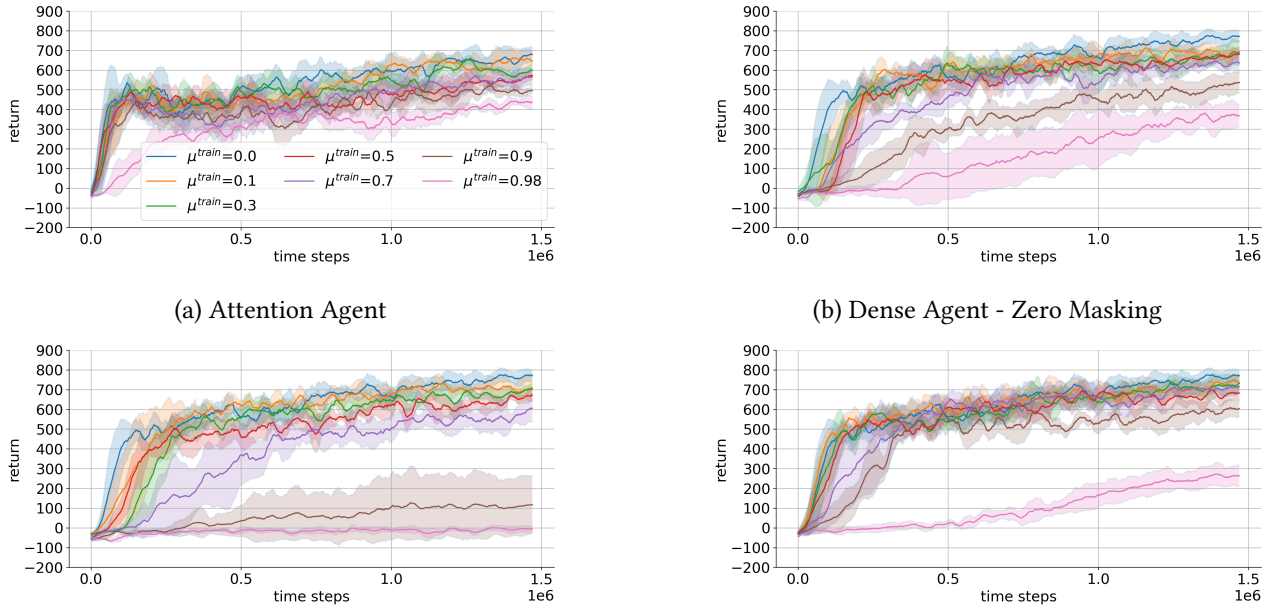
6.1.2 CarRacing - Vision Task

Figure 6.5 illustrates the mean returns obtained during training by our attention agent and the baseline convolutional agent variants on the CarRacing-v2 environment for $\mu^{train} = 0.0, 0.1, 0.3, \ldots, 0.9, 0.98$, where $\mu^{train} = 0.98$ corresponds to just 3 unmasked sensory signals (pixel patches) per time step. Again, we are looking to examine the agents' maximum return obtained, as well as their sample efficiency and stability during training.

Considering the results from our attention agent illustrated in Figure 6.5 (a) we make the following observations. First, the attention agent is able to make progress in learning the task under all masking ratios and, as with the vector task, we see a reliable decrease in mean return with each incremental increase in μ^{train}. In the final time steps the agent obtains a mean return of 734 under $\mu^{train} = 0.0$ and 476 under $\mu^{train} = 0.98$. Regarding sample efficiency, one key feature of the return curves across $\mu^{train} < 0.98$ is that they all spike sharply at the start of training, peaking at around $0.1e6$ time steps, but suffer an immediate depression thereafter, decreasing until around $0.5e6$ time steps and increasing from there on. Note that this is not something observed in the return curves in the training returns produced by the baseline convolutional agent variants, which might suggest a degree of overfitting on the part of the attention agent. Regarding learning stability, while there are some fluctuations in the return curves, especially for $\mu^{train} \geq 0.9$, the data suggests gradual, stable improvement over the course of training.

Considering the results from the baseline convolutional agent variants illustrated in Figures 6.5 (b), (c), and (d), we make the following observations. As with the attention agent, the baseline convolutional agent

[4]Note: When plotting these norms on a different scale, they do indeed produce a set of curves similar in shape to those produced by the zero/noise variants of the baseline dense agents.



(a) Attention Agent

(b) Dense Agent - Zero Masking

(c) Dense Agent - Noise Masking

(d) Dense Agent - Forward-Fill Masking

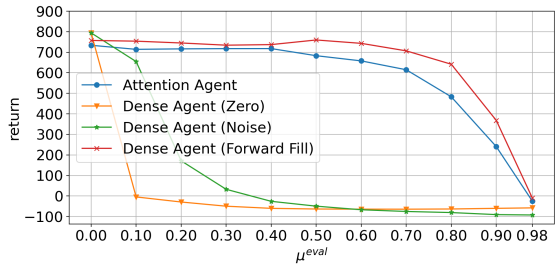
Figure 6.5.: Episodic returns obtained during training in the CarRacing-v2 environment during our Random Masking experiments by our attention agent and the baseline convolutional agents trained under fixed masking ratios $\mu^{train} = 0.0, 0.1, 0.3, \ldots, 0.9, 0.98$.

variants are generally able to make progress learning the task to a greater or lesser extent under all masking ratios with two exceptions. First, the forward-fill agent, while demonstrating efficient and stable learning across lower masking ratios, struggles to make progress learning under $\mu^{train} = 0.98$. Second, the noise masking variant fails to learn almost entirely under $\mu^{train} \geq 0.9$. The mean returns obtained in the final time steps of training range from 757-794 under $\mu^{train} = 0.0$ to 398, 278, and -3 under $\mu^{train} = 0.9$, for the zero masking, noise masking, and forward-fill masking variants, respectively.

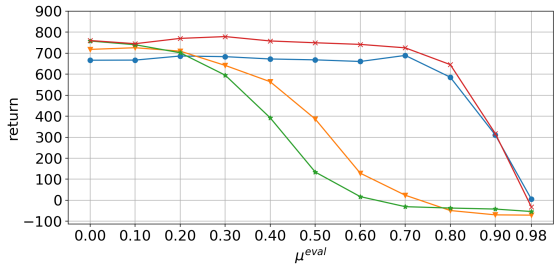
It is evident from examining the plots that noise masking causes the most significant degradation to learning with increasing μ^{train}. Interestingly, the forward-fill agent demonstrates greater robustness with respect to performance degradation than the zero masking agent when training under $\mu^{train} < 0.98$. However, $\mu^{train} = 0.98$ the mean return produced across training by the forward-fill agent drops dramatically in a manner similar to the dense forward-fill agent trained under $\mu^{train} = 5/6$ on the Acrobot-v1 task (Figure 6.2 (d)). In the case of $\mu^{train} < 0.98$, one major contributing factor to the robustness of the convolutional forward-fill agent to performance degradation under increasing partial observability and noise (imputed values) is that the vast majority of the game screen is made up of (i) road and (ii) grass. Due to this fact, imputing missing patches of pixel values with those from previous time steps is likely to cause only minor meaningful distortions to the observation, which is not the case for zero or noise masking.

Regarding sample efficiency, the mean return obtained by all baseline convolutional agent variants under $\mu^{train} \leq 0.7$ generally increases sharply between 0 and $0.5e6$ time steps, although not as sharply as the attention agent, and continues to gradually increase until the end of training. Additionally, the sample efficiency degrades with increasing μ^{train}, affecting the noise masking variant most severely for $\mu^{train} \geq 0.9$. Regarding stability, as with the attention agent, there are minor fluctuations which increase with increasing μ^{train}, but in general, the return curves suggest gradual, stable improvement throughout training.

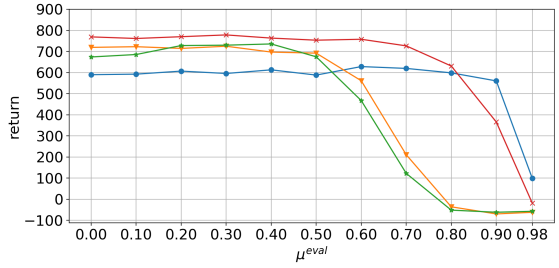
Figure 6.6 illustrates the same generalisation evaluation conducted with the Acrobot-v1 task; agents trained under a fixed μ^{train} are evaluated across $\mu^{eval} = 0.0, 0.1, 0.3, \ldots, 0.9, 0.98$ to determine how they generalise to novel degrees of partial observability and noise. Considering Figure 6.6 (a), which illustrates the results of all agents trained under $\mu^{train} = 0.0$ we observe the following. The attention agent is able to generalise, maintaining a return of ≈ 700 until $\mu^{eval} = 0.5$, whereafter it experiences a drop-off, decreasing to a mean return of ≈ 0 by $\mu^{eval} = 0.98$, as might be expected. Considering the baseline agents we see two distinct outcomes. First, the forward-fill masking variant of the baseline agent demonstrates the strongest generalisation,



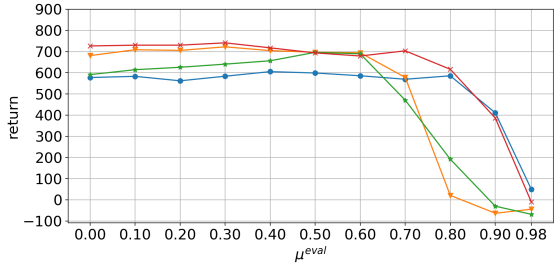
(a) $\mu^{train} = 0.0$



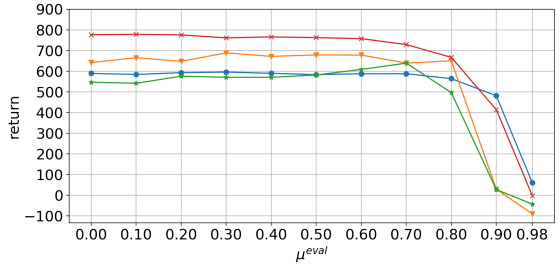
(b) $\mu^{train} = 0.1$



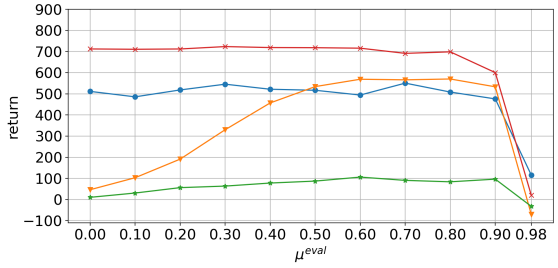
(c) $\mu^{train} = 0.3$



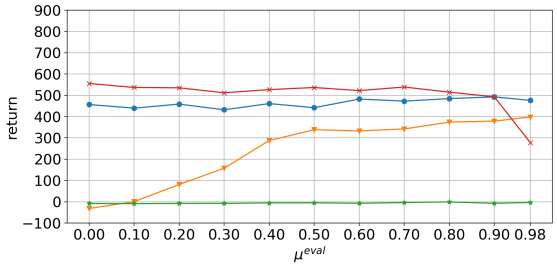
(d) $\mu^{train} = 0.5$



(e) $\mu^{train} = 0.7$



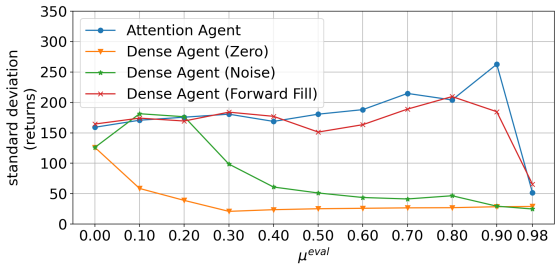
(f) $\mu^{train} = 0.9$



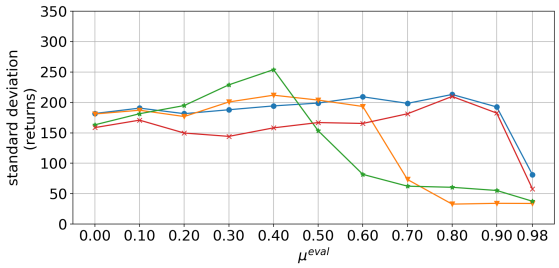
(g) $\mu^{train} = 0.98$

Figure 6.6.: Generalisation to novel masking ratios on CarRacing-v2 (vision): episodic returns generated by our attention and dense agents (zero, noise, and forward-fill masking), each trained on a fixed masking ratio μ^{train}, when evaluated across a set of increasing masking ratios μ^{eval}.

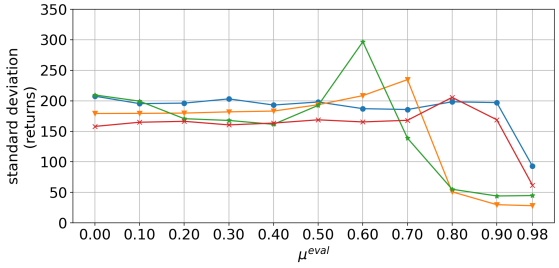
obtaining a mean return of ≈ 750 until $\mu^{eval} = 0.6$ and thereafter decreasing to ≈ 0 by $\mu^{eval} = 0.98$, as with the attention agent. Second, both the zero masking and noise masking variants of the baseline agent suffer a near-immediate drop in mean return, from 794 at $\mu^{eval} = 0.0$ to -5 at $\mu^{eval} = 0.1$ for the zero masking variant and from 794 at $\mu^{eval} = 0.0$ to 33 at $\mu^{eval} = 0.3$ (dropping below zero thereafter) for the noise masking variant. The dramatic drop-off in performance observed for the zero and noise masking variants of the baseline agents is because the imputed values are out-of-distribution, that is, entirely novel to the agent. Due to this fact, and the structure of the CNN network core, even a single patch of zeros/Gaussian noise propagates through the network layers and significantly alters the latent code, as we will show. The forward-fill variant of the baseline agent, on the other hand, does not experience the same drop-off in performance because pixel values imputed to the missing patches, while potentially causing minor distortions to the image such as those seen in Figure 4.4 (a.2), are in-distribution and do not affect the latent code in the same way.



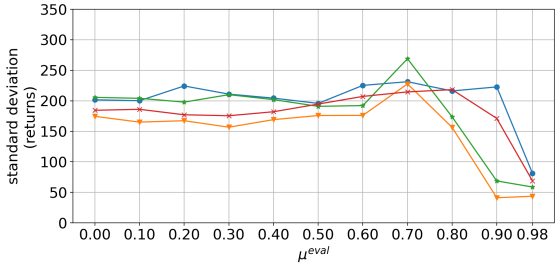
(a) $\mu^{train} = 0.0$



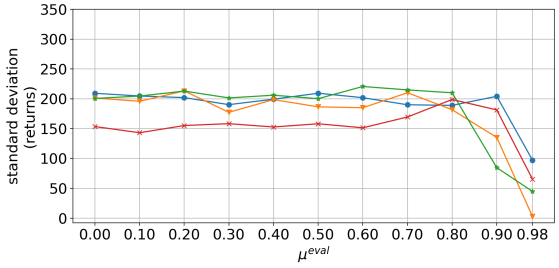
(b) $\mu^{train} = 0.1$



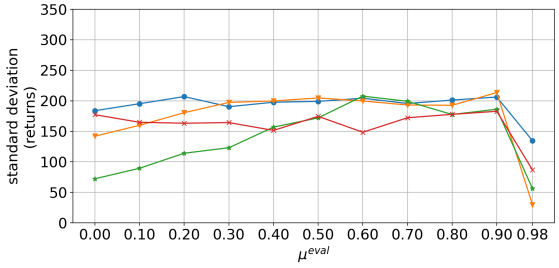
(c) $\mu^{train} = 0.3$



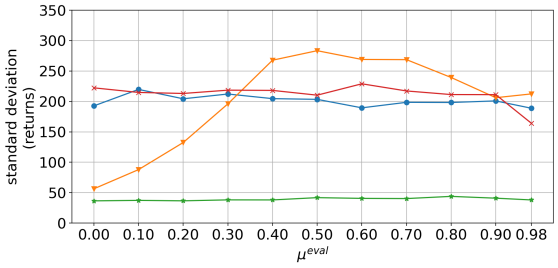
(d) $\mu^{train} = 0.5$



(e) $\mu^{train} = 0.7$



(f) $\mu^{train} = 0.9$



(g) $\mu^{train} = 0.98$

Figure 6.7.: Standard deviations of the distributions of the returns - illustrated in Figure 6.2 - obtained during evaluation on CarRacing-v2 by our attention agent and the baseline convolutional agents.

Then, considering the results for agents trained under $\mu^{train} = 0.1, 0.3, 0.5, 0.7$ illustrated in Figure 6.6 (b), (c), (d), and (e) we see a broadly similar pattern among all agents. In the case of the attention agent, increasing μ^{train} leads to a marginal decrease in mean return obtained across $\mu^{eval} < 0.7$ from ≈ 700 under $\mu^{train} = 0.0$ to ≈ 600 under $\mu^{train} = 0.7$. For $\mu^{eval} \geq 0.7$ the attention agent's mean return increases as it learns to generalise to higher degrees of partial observability. Additionally, although our attention agent generally obtains lower returns when compared to the forward-fill variant of the baseline agent, it performs comparably or better in the case of $\mu^{eval} \geq 0.9$ for the values of μ^{train} being considered, likely because it does not suffer the additional burden of extreme noise suffered by the baseline agents at high masking ratios. In the case of both the zero masking and noise masking variants of the baseline agent, we see significant improvement in the agent's ability to generalise to greater degrees of partial observability as μ^{train} increases, however, in each case there is a dropping-off point whereafter the agent appears unable to handle the noise introduced by the imputed values. Worth noting is the fact that zero masking and noise masking agents also perform

comparably up until $\mu^{train} = 0.3$, whereafter the zero masking variant appears to exceed the performance of the noise masking variant in many cases. Likely, this is simply due to the higher variance of the Gaussian noise values which confounds the agent. Lastly, examining the forward-fill variant, we see significant improvement in its ability to generalise, far exceeding the performance of the other baseline variants, likely due once again to the in-distribution nature of the imputed values. A possible conclusion we might draw from these results is that it is not the absence of information which causes the degradation of performance under increasing partial observability, but the noise introduced by the imputed values, and in particular, the statistical properties of the noise.

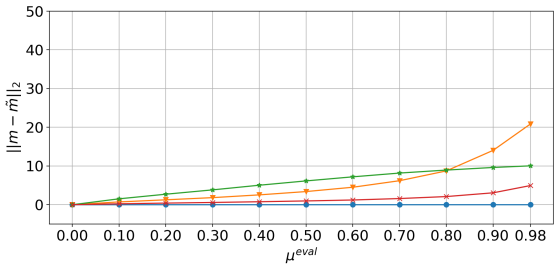
Finally, examining the results for agents trained under $\mu^{train} \geq 0.9$, illustrated in Figure 6.6 (f) and (g), respectively, we make the following observations. Considering our attention agent, we see that although there is some variance in the mean returns generated over all μ^{eval}, the agent's performance is largely consistent, obtaining a return of ≈ 500, except for the agent trained under $\mu^{train} = 0.9$ and evaluated under $\mu^{eval} = 0.98$ in which case a return of ≈ 100 was obtained, exceeding the performance of all variants of the baseline agent. Furthermore, for our attention agent trained and evaluated under extreme rates of sensor failure ($\mu^{train} = 0.98$ and $\mu^{eval} = 0.98$) and therefore of partial observability, we observe returns of ≈ 500, nearly 100 and 200 points higher than the zero masking and forward-fill masking variants of the baseline agent, respectively. This result clearly demonstrates one advantage of our attention-based architecture under such conditions.

Considering the baseline agents, we see that as with the Acrobot-v1 task, the noise masking variant performs very poorly in general, obtaining mean returns between 0 and 100 for $\mu^{train} = 0.9$ and ≈ 0 for $\mu^{train} = 0.98$ for all μ^{eval}. In the case of the zero masking variant of the baseline agent trained under $\mu^{train} = 0.9$, we observe returns comparable to, or exceeding, our attention agent for $\mu^{eval} = 0.5, \ldots, 0.9$ while scoring a lower return than our attention agent when evaluated under $\mu^{eval} = 0.98$ and suffering a near total decline in mean return as μ^{eval} is decreased towards 0.0, scoring a mean return of just 46 for $\mu^{eval} = 0.0$. We see a similar pattern in the returns obtained by the zero masking variant trained under $\mu^{train} = 0.98$, only with lower returns across the board with the exception of $\mu^{eval} = 0.98$, which it obtains the second highest return of ≈ 400. The conclusion we might draw from these observations is that the CNN network core is able to learn to handle noise (in the form of Gaussian noise or zeros) provided the distribution remains close to that observed during training, but suffers outside of that distribution, even when a greater proportion of 'useful' information (the true pixel values) is included at evaluation time than during training. Finally, the forward-fill variant performs the best overall in both cases for $\mu^{eval} < 0.98$, obtaining a mean return of 533 for $\mu^{eval} = 0.9$, and, in contrast to the other baseline agents, experiences an *increase* in the mean return it obtains to around 700 when μ^{eval} is decreased, meaning that it benefits from the additional information. However, for $\mu^{eval} = 0.98$ under conditions of extreme partial observability, the forward-fill variant suffers a drop in mean return to ≈ 300, likely overwhelmed by the noise arising from the imputed values.

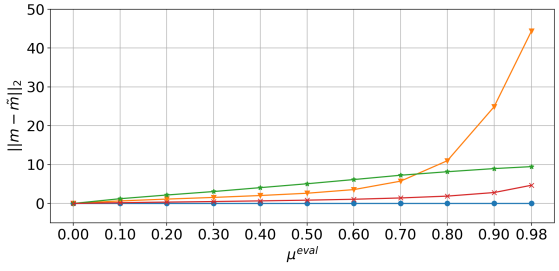
As above, Figure 6.7 illustrates the standard deviations of the returns obtained by each agent, trained under a fixed μ^{train}, and evaluated across all μ^{eval}. In general, the standard deviations are more consistent across agents, falling within the window of 150-200 in most cases, with the exception of the lower standard deviations observed for $\mu^{eval} = 0.98$ for all agents trained under $\mu^{train} \leq 0.9$ which is explained by the low returns obtained in most cases. The low standard deviations obtained by the zero and noise masking variants of the baseline agent, especially those trained under $\mu^{train} = 0.0$ and $\mu^{train} = 0.1$ are explained by the consistently near-or-below-zero returns obtained by each agent when evaluated under high μ^{eval}.

Finally, we examine the results in Figure 6.8, which illustrate the mean norm of the difference between latent codes produced by masked and unmasked versions of the same observations. Examining the results for agents trained under $\mu^{train} = 0.0$ in 6.8 (a) we observe that in the case of the baseline CNN agent, zero and noise masking cause large, significant changes to the latent code, with zero masking appearing to have the largest impact. An interesting question to ask here is whether the interpretation of zero masking as simulating agent 'blindness' - as is the interpretation in such masking experiments as those conducted in [16] - is correct, or whether the correct view to take is that zeros are more akin to noise as opposed to simply the absence of information. A second observation is that forward-fill masking appears to have a far smaller impact on the latent code vector produced by the baseline CNN agent for all μ^{eval}. In the case of the attention agent, we once again see that the norm of the difference between latent codes produced by masked and unmasked observations is far smaller (<1) than those produced by the baseline agent, possibly for the same architectural

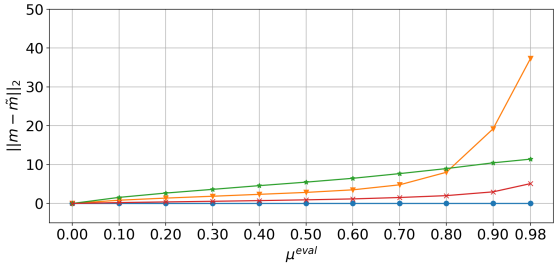
(a) $\mu^{train} = 0.0$



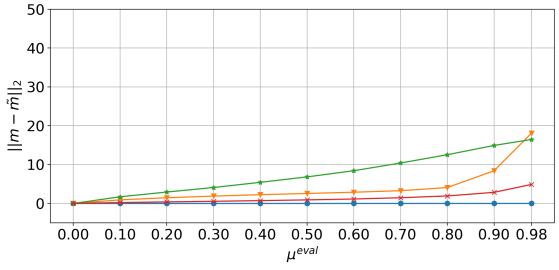
(b) $\mu^{train} = 0.1$



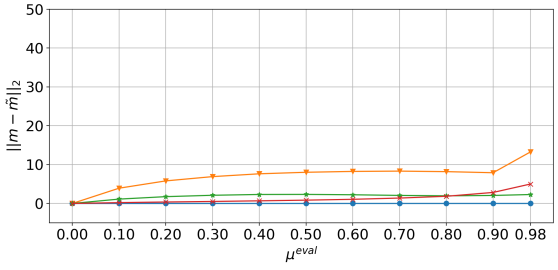
(c) $\mu^{train} = 0.3$



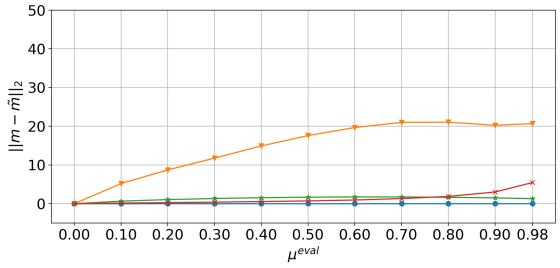
(d) $\mu^{train} = 0.5$



(e) $\mu^{train} = 0.7$



(f) $\mu^{train} = 0.9$



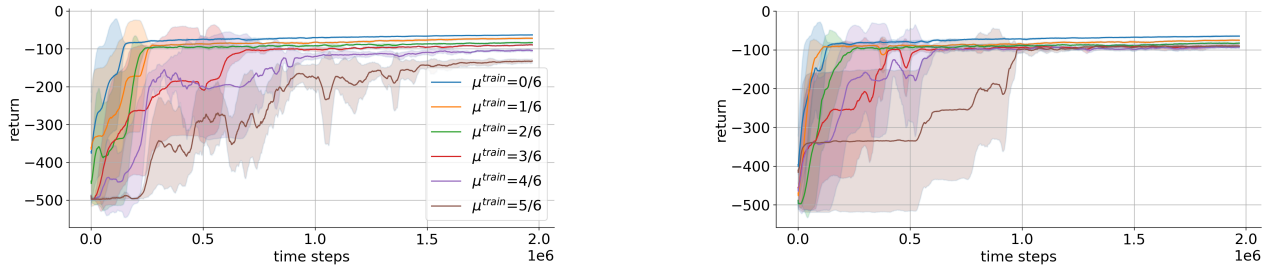
(g) $\mu^{train} = 0.98$

Figure 6.8.: Distance between latent codes produced by unmasked and masked observations in CarRacing-v2

reasons we posited in the case of the results observed from Acrobot-v1 environment. Examining the results for agents trained under $\mu^{train} > 0.0$, illustrated in Figure 6.8 (b) through (f), we observe a similar pattern which corresponds with the results illustrated in Figure 6.6 - the mean returns generated by the agents - that is; as μ^{train} increases, the difference between masked and unmasked latent codes decreases across all μ^{eval}. For $\mu^{train} = 0.9$, it is worth noting that the small norms recorded for the noise masking variant of the baseline agent are because the agent fails to learn to extract any meaningful features from the image, therefore struggling to make any progress learning the task, and as such the latent code vectors produced for masked an unmasked observations may largely be considered random.

6.2 Generated Masking

6.2.1 Acrobot - Vector Task



(a) Random Masking Policy

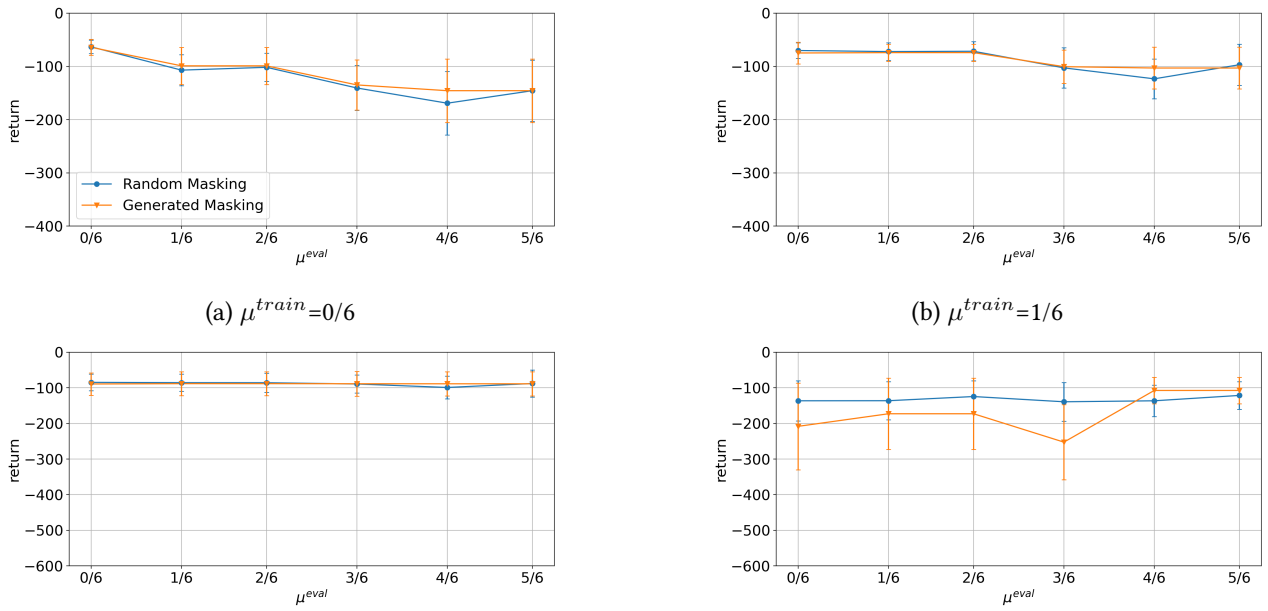
(b) Non-random (Generated) Masking Policy

Figure 6.9.: Training Returns from the generated masking experiments in the Acrobot-v1 (vector) environment. We emulate a low-bandwidth communication channel between the agent and the environment, allowing only a fixed percentage of the observation, μ^{train}, to be queried by the agent at each time step, where each query takes the form of a bit mask. We compare two mask generation policies: (a) a random policy under which each bit mask is generated by randomly sampling bits (without replacement) according to a uniform distribution and (b) a generated (non-random) policy under which each bit mask is sampled (without replacement) from a multinomial distribution output from a separate 'masking head' appended to the policy network - in this way, the agent can choose which sensory signals to attend to, and which to ignore.

Figure 6.9 illustrates the returns generated by our attention agent during training in our generated masking experiments in the Acrobot-v1 environment. As a reminder, in these experiments we emulated a limited-bandwidth communication channel between the agent and the environment, requiring the agent to query a fixed proportion, μ^{train}, of the sensory signals by generating a bit mask at each time step according to some mask generation policy. We ran the experiment under the masking ratios $\mu^{train} = 0/6, 1/6, \ldots, 5/6$. Figure 6.9 (a) illustrates the returns obtained under a random mask-generation policy, where bit masks are generated by randomly sampling bits (without replacement) from a uniform distribution over all sensory signals. Figure 6.9 (b) illustrates the returns generated under a non-random mask-generation policy under which each bit mask is sampled (without replacement) from a multinomial distribution output from a separate 'masking head' appended to the policy network - in this way, the agent can choose which sensory signals to attend to, and which to ignore. Note that our attention agent with a random mask-generation policy in the 'low-bandwidth communication channel' scenario is materially equivalent to our attention agent trained under conditions of random sensor failure and, as such, we use the same results as a baseline against which to compare our attention agent with a non-random mask generation policy.

We make the following observations by comparing the difference between the returns generated by our attention agent with random and non-random mask-generation policies, referred to henceforth as the 'random' and 'non-random' agents. Returns generated by the non-random agent demonstrate better sample efficiency and stability during training, especially for $\mu^{train} > 2/6$. In the case of the random agent trained under both $\mu^{train} = 3/6$ and $\mu^{train} = 4/6$, convergence to a mean return or ≈ -100 takes around $1e6$ time steps, whereas in the case of the non-random agent convergence to an equivalent (or slightly higher) mean return takes just over $0.5e6$ time steps. In the case of $\mu^{train} = 5/6$, we see the non-random agent is able to obtain a mean return exceeding -100 in $1e6$, whereas the random agent converges to a mean return of less than -100.

Allowing the agent to explicitly query the sensors at each time step via mask generation confers an advantage in this case likely because it allows the agent to attend to the most salient sensory signals for the purposes of optimal decision-making with respect to return maximisation. However, it is important to keep in mind these results are particular to this environment in which the observation space consists of only six real numbers (and hence six sensory signals). As we will discuss, there are likely advantages and disadvantages to this sort of hard attention mechanism which depends on the environment in question.



(a) μ^{train}=0/6

(b) μ^{train}=1/6

(c) μ^{train}=3/6

(d) μ^{train}=5/6

Figure 6.10.: Generalisation to novel communication bandwidth limits in the Acrobot-v1 (vector) environment: the plots indicate returns generated by our attention agent with random ("Random Masking") and non-random ("Generated Masking") mask generation policies, trained under fixed μ^{train}, during evaluation under $\mu^{eval} = 0/6, 1/6, 3/6, 5/6$. The vertical bars indicate the standard deviation.

As with the random masking experiments, the results of which are detailed above in section 6.1, we would like to evaluate our non-random attention agent under novel masking ratios to asses how the agent performs under different communication bandwidth limits than that under which it was trained, μ^{train}. Figure 6.10 illustrates the returns generated by both the random ("Random Masking") and non-random ("Generated Masking"), each trained under fixed μ^{train}, when evaluated under $\mu^{eval} = 0/6, 1/6, 3/6, 5/6$. For the agents trained under $\mu^{train} = 0/6$, we observe that the performance of both the non-random and random agents are approximately equivalent across all μ^{eval}, which makes sense since the non-random agent would have received the full observation vector during training and as such would not have learned to attend in any meaningful way to salient sensory signals via mask generation, likely leading to a more or less random mask generation policy. For the agents trained under $\mu^{train} = 1/6$ and $\mu^{train} = 3/6$ we once again see approximately equivalent performance over all μ^{eval}, which is consistent with the results in Figure 6.9 which show the random and non-random agents converge, albeit with different sample efficiency, to the same returns during training. This is likely due, at least in part, to the fact that the observation space is so small and as such a random mask generation policy is sufficient to obtain the information required for optimal decision-making under lower μ^{train}. Finally, for agents trained under $\mu^{train} = 5/6$, we see that for $\mu^{eval} \geq 4/6$ the non-random agent outperforms the random agent, which is consistent with the returns obtained by each during training. However, we see that for $\mu^{eval} < 4/6$ the non-random agent performs worse than the random agent, indicating that the additional information is in some way confounding the non-random agent in order to understand this, it is useful to examine the masks generated by the non-random agent under high μ^{eval}, which we now examine.

Figure 6.11 illustrates the bit masks generated by the non-random agent, trained under $\mu^{train} = 5/6$, under $\mu^{eval} = 5/6$ in the Acrobot-v1 environment over 50 steps during an episode in which the agent scored a return of >-100. The unmasked sensory signals, indicated by the yellow squares, give a sense of the relative importance of the sensory signals for optimal decision-making under the agent's policy. The masks illustrate a pattern of attention over time in which the agent periodically attends to signals encoding the angular position and velocity of the links. However, the agent appears to allocate the majority of its attention to the angular velocities, $\dot{\theta}_1$ and $\dot{\theta}_2$, as this is likely the most important information in assessing whether the links are rotating

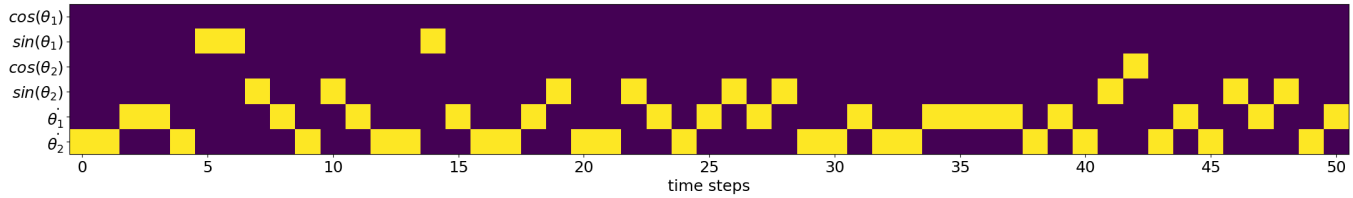


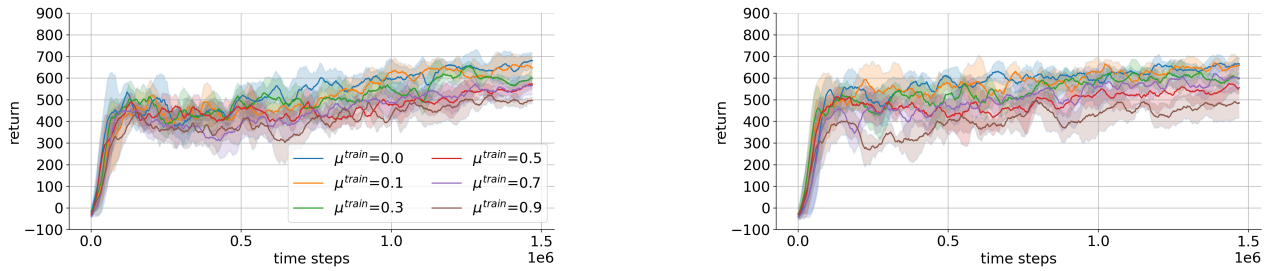
Figure 6.11.: Bit masks generated by our attention agent with a non-random mask generation policy, trained under $\mu^{train} = 5/6$, on the Acrobot-v1 task under $\mu^{eval} = 5/6$ for 50 steps of an episode in which the agent obtained a return of > -100. Yellow denotes unmasked sensory signals. The agent attends most frequently to the sensory signals encoding the angular velocities of the two links, $\dot{\theta}_1$ and $\dot{\theta}_2$, alternating between them. The agent appears to periodically sample positional information encoded in the sensory signals $sin(\theta_2)$ - the relative angle between the links - and, to a lesser extent, $sin(\theta_1)$. Curiously, the agent allocates nearly none of its attention over time to $cos(\theta_1)$ and $cos(\theta_2)$, indicating that the information encoded by these sensory signals may be redundant.

in such a way that will result in the goal state being reached. Interestingly, the agent obtains information regarding angular position nearly exclusively from the sensory signals $sin(\theta_2)$, the sine of the relative angle between the two links, and $sin(\theta_1)$, the sine of the absolute angle of the link attached to the wall. The fact that the agent does not attend to the sensory signals $cos(\theta_1)$ and $cos(\theta_2)$ offers two insights. Firstly, the information regarding angular position encoded in these signals is likely largely redundant given $sin(\theta_1)$ and $sin(\theta_2)$. Secondly, it gives us a clue as to why the non-random agent trained under $\mu^{train} = 5/6$ might suffer when evaluated under $\mu^{eval} \leq 3/6$; the agent most likely learned a mask generation policy which largely ignored $cos(\theta_1)$ and $cos(\theta_2)$, meaning that the embedding vectors associated with these sensory signals in f_{embed} may have been poorly optimised. Therefore, evaluating the agent under a low masking ratio likely leads to $cos(\theta_1)$ and $cos(\theta_2)$ being attended to more frequently, which would confound the agent's decision-making. If this is indeed the case, it offers a cautionary lesson regarding the decision between non-random and random mask generation: allowing an agent to attend to a subset of sensory signals during training may be beneficial, but the bias it leans towards certain sensory signals may lead to performance degradation if it encounters other sensory signals at inference time.

6.2.2 CarRacing - Vision Task

Figure 6.9 illustrates the returns generated by our attention agent during training in our generated masking experiments in the CarRacing-v2 environment. Figure 6.9 (a) and (b) correspond to the random and non-random agents, equivalent to the Acrobot-v1 results illustrated above in section 6.2.1. In contrast with the gains in sample efficiency, stability, and maximum returns demonstrated on the Acrobot-v1 task by the non-random agent over the random agent, the non-random agent appears to offer little advantage whatsoever in the case of CarRacing-v2. The one exception to this is that the returns produced by the non-random agent during the initial 0.5e6 training steps appear not to suffer the same depression after the initial spike as in the case of the random agent - it is unclear what the reason for this is, but it may be that overfitting is harder to do when the agent is required to learn a mask generation policy in addition to an action selection policy. One final observation to make here is that the maximum returns obtained towards the end of training appear to be slightly lower in the case of the non-random agent, possibly because learning an effective mask generation policy negatively impacts sample efficiency.

Equivalent to the Acrobot-v1 results in Figure 6.10, Figure 6.13 illustrates the returns obtained by the random and non-random agents, each trained under fixed μ^{train}, when evaluated on novel masking ratios $\mu^{eval} = 0.0, 0.1, 0.3, \ldots, 0.9$ in order to emulate novel communication bandwidth limits. For $\mu^{train} \leq 0.3$ we observe that the the returns generated by the non-random agent under novel masking ratios are at most equal to, but often less than, those generated by the random agent. This is especially true for the non-random agent trained under $\mu^{train} = 0.0$, which is likely because, as discussed in section 6.2.1, the agent's mask generation policy was immaterial during training as it received the full observation regardless of the output



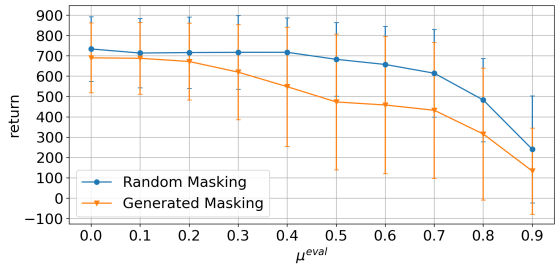
(a) Random Masking

(b) Generated Masking

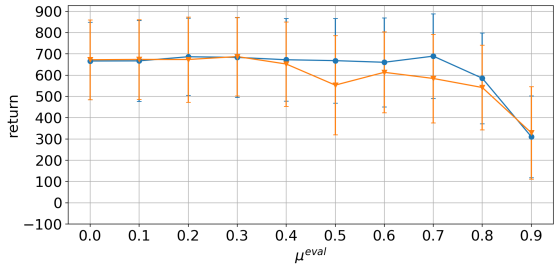
Figure 6.12.: Training Returns from the generated masking experiments in the CarRacing-v2 (vision) environment. We emulate a low-bandwidth communication channel between the agent and the environment, allowing only a fixed percentage of the observation, μ^{train}, to be queried by the agent at each time step, where each query takes the form of a bit mask. We compare two mask generation policies: (a) a random policy under which each bit mask is generated by randomly sampling bits (without replacement) according to a uniform distribution and (b) a generated (non-random) policy under which each bit mask is sampled (without replacement) from a multinomial distribution output from a separate 'masking head' appended to the policy network - in this way, the agent can choose which sensory signals to attend to, and which to ignore.

from the masking head, and so the mask generation policy produced at the end of training was likely random, but not uniformly random, a fact which, coupled with the large number of sensory signals (144), resulted in a worse performance when evaluated on novel masking ratios. In the case of $\mu^{train} = 0.1$ and $\mu^{train} = 0.3$, the lower returns obtained by the non-random agent relative to the random agent may be explained by the fact that learning an effective mask generation policy at lower masking ratios which generalises to higher masking ratios might simply require either (a) more training steps (b) exposure to a range of masking ratios during training, or both, which we did not experiment with due to budget/computational constraints. For $\mu^{train} \geq 0.5$ the returns generated by the random and non-random agents are approximately equivalent across all μ^{eval}, suggesting that allowing the agent to query to sensory signals explicitly via mask generation offers no meaningful advantage with respect to performance.

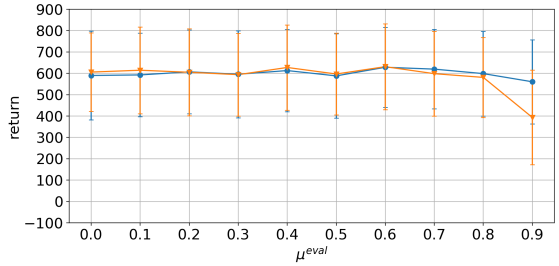
One advantage offered by the non-random agent over the random agent is that given the agent can learn an effective policy, generated masks allow for interpretability by allowing us to see explicitly which visual features on the game screen the agent finds most salient for the purposes of action selection. Figure 6.14 illustrates masks generated by our non-random attention agent, trained under $\mu^{train} = 0.9$, at evaluation time under $\mu^{eval} = 0.9$. The high masking ratio should in theory force the agent to learn to attend to salient visual features dynamically as the game screen changes. However, since the policy ultimately learned by the agent was sub-optimal, converging on a mean return just shy of 400, some of the masks appear to be coherent, while others appear to be less coherent and more random. Figure 6.14 (a) illustrates examples of masks, illustrated as white translucent squares over the attended patches, which appear to be coherent, showing the agent attending to salient features such as the edges and bends/corners of the racing track. Figure 6.14 (b) illustrates examples of masks which appear to be less coherent and more random, with many of the attended patches being located off the track on the grass, containing little meaningful information about the underlying state of the MDP. The important takeaway here is that the advantage of interpretability is only realised if the agent can learn an effective policy - an outcome which might be negatively impacted by the use of a hard attention mechanism such as generated masking.



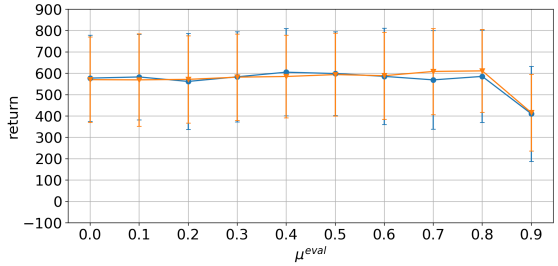
(a) $\mu^{train} = 0.0$



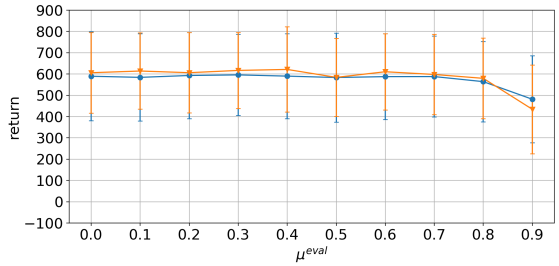
(b) $\mu^{train} = 0.1$



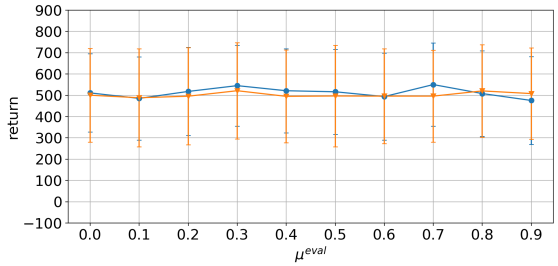
(c) $\mu^{train} = 0.3$



(d) $\mu^{train} = 0.5$

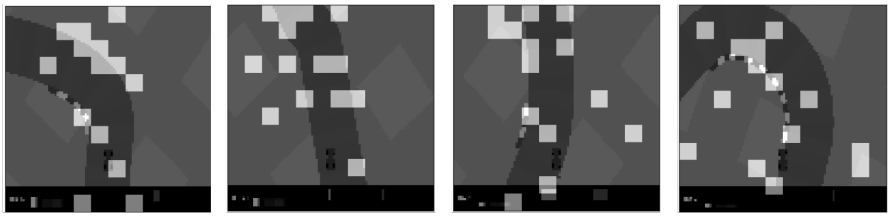


(e) $\mu^{train} = 0.7$

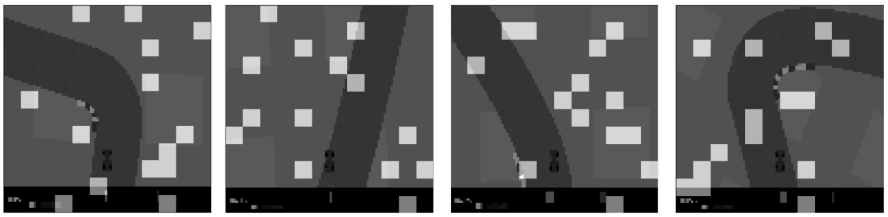


(f) $\mu^{train} = 0.9$

Figure 6.13.: Generalisation to novel communication bandwidth limits in the CarRacing-v2 (vision) environment: the plots indicate returns generated by our attention agent with random ("Random Masking") and non-random ("Generated Masking") mask generation policies, trained under fixed μ^{train}, during evaluation under $\mu^{eval} = 0.0, 0.1, 0.3, \ldots, 0.9$. The vertical bars indicate the standard deviation.



(a) Seemingly *coherent* generated masks



(b) Seemingly *incoherent* generated masks

Figure 6.14.: Masks, illustrated by white translucent squares over the attended patches, generated by our attention agent with a non-random mask generation policy, trained under $\mu^{train} = 0.9$, on the CarRacing-v3 task under $\mu^{eval} = 0.9$: (a) Masks which appear to be coherent - the agent appears to be focusing on salient features such as the edges and bends in the track, and (b) Masks which appear to be incoherent - the agent does not appear to be focusing on salient features.

Chapter 7

Discussion

In this section, we discuss the results of our experiments and how the findings both build on and fill gaps in the related research presented in section 3. In sections 7.1 and 7.2, we discuss the results from the Random Masking experiments, presented in section 6.1, and the Generated Masking experiments, presented in 6.2, respectively.

7.1 Random Masking

In the Acrobot-v1 (vector) task, it is clear that the absence of sensory information negatively affects the ability of both our attention agent and the baseline dense agent to learn efficiently from experience during training. It is also clear in the case of the baseline dense agent, however, that the type and proportion of imputed values further degrade the agent's ability to learn, as evidenced by the training curves in Figure 6.1.

Zero masking appears to degrade the agent's performance the least - this is likely because, although imputing zeros may give rise to out-of-distribution observation vectors, the imputed value is constant, which is not the case for noise masking or forward-fill masking. Consequently, given that the true scalar values in an unmasked observation vector are likely never to be precisely zero, and that ANNs are excellent non-linear function approximators, it is possible that the policy network was able to learn to treat values of precisely zero distinctly from values about zero.

By contrast, noise masking appears to have the most detrimental effect on the learning ability of the baseline agent. This is most likely because the imputed values, being Gaussian noise, give rise to out-of-distribution observations and are also, by definition, random/highly variable. This means that the agent must attempt to learn an optimal mapping from observations to actions, but in a scenario where the observations are increasingly random (i.e. as μ^{train} is increased) - a task which proves impossible in the Acrobot-v1 environment by $\mu^{train} = 5/6$.

The impact of forward-fill on the ability of the baseline dense agent to learn efficiently appears to depend on the value of μ^{train}. Imputing the most recently observed sensory values under low rates of sensor failure appears to have little effect on the agent's ability to learn for $\mu^{train} \leq 2/6$ since the imputed values provide a sufficient approximation to the true observation vector. For $\mu^{train} > 2/6$, however, it appears that the increase in latency of the imputed values results in a worsening of the approximation which begins to severely impact the agent's ability to learn. The imputed values in the case of forward-fill masking are both non-constant and give rise to observation vectors which are approximately in-distribution, but are yet incorrect due to the latency. Based on this, the baseline agent's policy network is unable to learn to discriminate, even approximately, between imputed (false) values and true values, as might be the case under zero masking, since the observations under forward-fill masking are in-distribution and yet false (i.e. the observation doesn't match the underlying state of the MDP). Note that while the forward-fill method of imputation might give rise to approximately in-distribution observation vectors in a simple environment such as Acrobot-v1 where the state space is relatively small, even for relatively high masking ratios, this may not necessarily be the case for more complex environments with larger state spaces.

While our attention agent does not appear to learn as efficiently as the baseline agents for lower rates of

sensor failure - for $\mu^{train} \leq 2/6$ - its ability to learn is more robust under higher masking ratios during training. It is clear from the data that the absence of noise arising from imputed values offers a distinct advantage, allowing for successful optimisation of the attention-based policy under conditions of extreme partial observability - for $\mu^{train} = 5/6$ - and resulting in a maximum mean return which is only marginally less than that obtained by the attention agent under lower masking ratios with relatively efficient convergence and a low standard deviation across all 5 seeds.

Concerning generalisation to novel degrees of partial observability in the Acrbobot task, it is clear that, in the case of the baseline dense agent, forward-fill masking offers a distinct advantage over both noise and zero masking across all μ^{eval} for agents trained under $\mu^{train} = 0/6$. However, training under moderate degrees of partial observability ($1/6 < \mu^{train} < 4/6$) greatly improves the ability of the baseline agents to handle novel degrees of partial observability across the board - examining the norms of the differences between latent codes produced from masked and unmasked observations in Figure 6.4 it is clear that this is due to, at least in part, the ability of the MLP core of the policy network to learn to be more robust to noise when trained under higher degrees of partial observability. The poor returns produced across all μ^{eval} by the baseline dense agents trained under $\mu^{train} = 5/6$ are consistent with the returns obtained during training, except for the zero masking which performs better than expected and is even able to take advantage of the additional information and generate higher returns when evaluated under lower degrees of partial observability ($\mu^{eval} < \mu^{train}$). These findings further support the hypothesis that the policy network has learned to discriminate between imputed values (zeros) and true values.

The attention agent appears to be the most robust when evaluated under novel degrees of partial observability, suffering the least performance degradation with increasing μ^{eval} across the board. One feature of the evaluation returns obtained by the attention agent, illustrated in Figure 6.2, is that the mean return appears to increase slightly from $\mu^{eval} = 4/6$ to $\mu^{eval} = 5/6$ - a pattern most pronounced for agents trained under $\mu^{train} \leq 2/6$. It is unclear why this might be the case, but perhaps the availability of only a single sensory signal forces the agent to rely entirely on the RNN to aggregate information over time, as opposed to relying more on the attention block to approximate the true state based on just two sensory signals, which might lead to incorrect inference on the part of the agent in the case that it has not been trained under a sufficiently high degree of partial observability.

Before discussing the results of the vision-based experiments, it is helpful to establish the key differences between the vector-based and vision-based environments used in our experiments regarding their observation spaces, and therefore the nature of the sensory signals constructed from their observations. In the case of Acrobot-v1, the observation space is a vector in $\mathbb{R}^6$ wherein each scalar component represents a physical quantity (e.g. angular velocity) which holds a lot of information, relatively speaking, about the state of the underlying MDP, even independently. By contrast, in the case of CarRacing-v2, the observation space is a 2D grid of $96 \times 96 = 9216$ pixels, each of which holds a relatively small amount of information regarding the state of the underlying MDP. In this way, the scalar values in the vector example are high-level features whereas the scalar values in the vision example are low-level features which might be transformed into a high-level feature vector by, for example, a sequence of convolutional layers in an ANN. These important differences have several distinct consequences for each agent in the context of the problem studied herein, which we will highlight in what follows.

In the CarRacing-v2 (vision) environment, it is once again clear that partial observability arising from the absence of sensory information, in the case of all agents, and noise in the form of imputed values, in the case of the baseline agents, negatively impact the ability of the agents concerned to learn efficiently. However, the effects of both partial observability and noise each have a distinct effect which appears to be related to the nature of the environment, and therefore the sensory values, as described above, which is evidenced by the difference in training and evaluation results observed for each variant (zero, noise, and forward-fill) of the baseline convolutional agent.

In the case of zero and noise masking, we see comparable results during training, with respect to maximum mean return obtained and sample efficiency, across both agent variants for lower rates of sensor failure ($\mu^{train} < 0.7$). However, for higher rates of sensor failure ($\mu^{train} \geq 0.7$) we see once again that the effect of noise masking appears to be more detrimental than zero masking, likely due to similar reasons as those discussed in the case of the Acrbobot tasks - zeros are constant, allowing the policy network to adapt, while

Gaussian noise is, by definition, not. Note that the delta between the maximum mean return obtained between $\mu^{train} = 0.0$ and $\mu^{train} = 0.9$ is 300 in the case of zero masking and 700 in the case of noise masking, which shows that noise masking causes learning to completely break down when the rate of sensor failure rate is extreme.

By contrast, the forward-fill variant of the baseline convolutional agent appears to be (somewhat surprisingly) robust, with respect to its ability to learn efficiently under high degrees of partial observability, that is, for $\mu^{train} \leq 0.9$, experiencing a breakdown in its ability to learn efficiently only under $\mu^{train} = 0.98$. To better understand this, consider that the observations emitted by the CarRacing-v2 environment are in some sense relatively simple, with the vast majority of the game screen consisting of either road or grass (two colours). Given that the car is always in the same position in the middle of the screen, an agent need only pay attention to the boundary edge between the road and the grass in order to successfully learn to 'drive' the car around the track - this is supported by the findings of [24], illustrated in Figure 3.14, which show that the attention agent attends to the patches which lie on the edge separating the road and the grass. The consequences of this are two-fold: first, an agent can learn to play CarRacing-v2 with access to only a small proportion of sensory information (it only needs to be able to approximate the shape of the road) and second, imputing the pixel values of missing patches with their most recently observed patches will, in addition to being in-distribution, very likely make little difference to the observation received by the agent - this may be seen in the illustration of a forward-filled observation from the CarRacing-v2 environment in Figure 4.4. Furthermore, it may be the case that when trained under higher masking ratios, the RNN in the agents policy network - specifically the gating mechanisms in the GRU cell - are able to update the hidden state based only on the patches which change from time step to time step. Altogether, these hypotheses might explain why the baseline convolutional agent appears more robust with respect to its ability to learn efficiently than the other baseline agents, having a delta of just 200 between the maximum returns obtained under $\mu^{train} = 0.0$ and $\mu^{train} = 0.9$. Increasing the masking ratio during training to $\mu^{train} = 0.98$ does, however, ultimately cause a breakdown in the agent's ability to learn as the latency causes the gap between the true observation and the masked observation received by the forward-fill agent to widen to a point where learning becomes impossible - this is where we see a clear advantage enjoyed by our attention agent.

In the case of our attention agent, we see a distinct pattern in the mean returns obtained during training, illustrated in Figure 6.5 (a). For each μ^{train}, the agent learns at a significantly faster rate than the baseline agents (including the forward-fill agent, especially in the case of $\mu^{train} >= 0.7$) up until approximately 0.1×10^6 time steps, followed by a depression and gradual recovery until approximately 0.5×10^6 time steps, increasing gradually thereafter. This pattern suggests that while the attention agent is not struggling to learn the task in general, some combination of (a) properties of the multi-head attention and (b) the learning algorithm might be combining to frustrate the ability of the attention agent to learn in a stable, monotonically increasing manner. Part of the explanation for this pattern might be the fact that the spatial inductive bias inherent in CNNs must be learned in the case of pure self-attention - this is likely the reason that the attention-based policy architecture proposed in [20], which is most similar to ours, utilises a 2-layered CNN to first extract the visual features from the input image before reshaping the output and passing it on to the attention block. Based on this, one idea for future work which might improve the stability and sample efficiency of our architecture across the board might be to embed patches independently via a small 2-layered CNN as a batch, instead of linearly embedding the patches as we have done. On the other hand, the fact that our attention agent is able to learn at such a high rate initially suggests this might not be the primary reason, and that this instability is likely similar to that encountered in previous research - as described in [21] - meaning that improvement might be achieved by other means such as by swapping the residual connections in the attention block with gating mechanisms [21] or increasing the number of attention blocks [20]. Possibly the most plausible theory is that the MHA layer is especially sensitive to the learning algorithm; we have used PPO in our experiments, but successful attempts to train smaller attention-based policies on CarRacing-v2 previously have utilised behaviour cloning [25] and evolutionary methods [24], choices which may have been motivated by similar observations when attempting to train their attention-based policy architectures with conventional deep RL algorithms.

It is worth noting here that several modifications to the attention-based architecture, which we suspect might improve the efficiency and stability of learning, are outlined in section 8.3 and may form the basis for an ablation study.

Nevertheless, we observe that our attention agent is able to make progress in learning the task, obtaining a mean return comparable to the baseline convolutional agent under $\mu^{train} = 0.0$, and is robust, with respect to its ability to learn, to increasing rates of sensor failure, having a delta of approximately 200 between the maximum return obtained between $\mu^{train} = 0.0$ and $\mu^{train} = 0.9$, as with the forward-fill variant the baseline agent, but surpassing all agents with respect to sample efficiency and maximum return when trained under $\mu^{train} = 0.98$, converging to a mean return of approximately 100 and 200 points higher than the zero and forward-fill variants of the baseline convolutional agent, respectively.

In analysing the results concerning the ability of the baseline convolutional agent to generalise to higher degrees of partial observability, illustrated in Figure 6.6, we observe that the effects of different imputation methods are distinct from those observed in the case of the Acrobot-v1 environment, resulting from the difference in nature between vector and vision-based environments outlined above. When evaluating the zero and noise masking variants of the baseline agents trained under $\mu^{train} = 0.0$ on higher rates of sensor failure (Figure 6.6 (a)), the near-complete failure of the agents to perform the task does not arise from the absence of sensory information, but from the imputation of sensory (pixel) values which are statistically novel to the agent with respect to the data on which it was trained, which causes the observations received by the agent to be out-of-distribution. Considering the case of $\mu^{eval} = 0.1$, the imputed pixel values (zeros or Gaussian noise) may make barely any difference to the game screen as it is perceived by a human being, but, in the case of the agent, these values propagate through the CNN and end up having a very significant impact on the latent code vector produced, as illustrated in Figure 6.8. This finding is consistent with previous findings from the literature wherein it has been established that perturbing even a single pixel can significantly alter the output of a convolutional neural network [109] in the case of image classification. On the other hand, [25] have shown that attention-based policies are robust to out-of-distribution visual features, such as changing the entire background (everything except the road) in the CarRacing-v2 environment.

As with the Acrobot-v1 task, we find that training all variants of the baseline convolutional agent under intermediate rates of sensor failure ($0.1 < \mu^{train} < 0.7$) appears to positively impact the agents' ability to generalise to novel conditions of greater partial observability up to a point. However, it appears that there is a saturation point (a μ^{eval} threshold) for each μ^{train} in the case of both the zero and noise masking variants beyond which the agent's return drops to near zero. Under extreme rates of sensor failure, that is for $\mu^{train} \geq 0.9$, a most noteworthy finding is the fact that the zero masking variant of the baseline convolutional agent is unable to generalise to *lower* rates of sensor failure ($\mu^{eval} < 0.6$). This finding reveals a critically important feature of convolutional neural networks; they struggle to generalise to novel degrees of partial observability when the imputed values cause the input to differ significantly from the data encountered during training data, statistically speaking.

Consistent with the training results, when evaluated under novel degrees of partial observability in the CarRacing-v2 environment, the forward-fill variant of the baseline convolutional agent, while performing comparably to the attention agent when trained under lower μ^{train}, appears to be the most robust to the novel, higher rates of sensor failure, up until $\mu^{eval} = 0.9$. However, for $\mu^{eval} = 0.98$, our attention agent appears to perform as well, in the case of $\mu^{train} \leq 0.1$, or better, in the case of $\mu^{train} \geq 0.3$, than all the variants of the baseline convolutional agent, including the forward-fill variant. It's worth noting that, given the failure of the baseline agent to generalise under zero/noise-masking, there remains an open question as to whether results produced by the forward-fill agent would replicate on another, more complex visual environment where signal latency might produce visual features which are foreign enough to cause a breakdown in learning at similar rates of sensor failure to the other two variants.

While our attention agent demonstrates robustness to novel degrees of partial observability in general, it appears to suffer from a drop in mean return for $\mu^{eval} \leq 0.9$ relative to the forward-fill variant of the baseline convolutional agent, especially for $0.1 \leq \mu^{train} \leq 0.9$. From the literature, we have both theoretical guarantees that self-attention layers are as expressive as convolutional layers [84] and empirical evidence that self-attention-based policy architectures can excel at CarRacing-v2 [24] and other vision-based tasks [20][23][21], so future research and experimentation into architectural adjustments and alternative training (optimisation) algorithms may well be able to close the gap, along with simply increasing the number of training steps.

When considering the problem of random sensor failure, the performance of the baseline agents under novel

degrees of partial observability depends on the combination of the environment's nature, the type of imputed values, and the degree of partial observability encountered by the agent during training versus at evaluation time. The results of the vision-based task, in particular, highlight a fundamental weakness of convolutional neural networks in the context of random sensor failure: they struggle to generalize to significantly higher or lower degrees of partial observability than those under which they were trained. Their performance appears to be highly dependent on the statistical proximity of observations encountered at evaluation time to the observations encountered during training. Consequently, due to the sequential dependencies that arise in Reinforcement Learning (RL) across time, such as trajectories of actions and observations, poor decision-making in response to a one-off outlier or out-of-distribution observation can have a knock-on effect. It is crucial to note that such a weakness could be dire in a production setting.

It is also important to note that any discussion regarding the comparative effectiveness of the imputation methods studied here is limited, as the relative effectiveness of each method will likely vary depending on the nature of the environment. For example, while zero masking may yield better results than forward-fill masking in a video game environment such as Atari Pong, where the background consists mostly of zero-valued pixels, the results might be inverted in a game where salient visual entities occupy the majority of the screen most of the time. Therefore, since we have only conducted experiments in two environments, wider experimentation is needed to validate our findings.

By contrast, the attention policy architecture demonstrates robustness to missing sensory information in a manner which appears more independent of these factors, especially at high levels of partial observability, with evident superiority on the vector-based task. In general, we submit that attention-based policies present a promising alternative to the dense and convolutional-based policies typically used in RL, especially in the case of missing data arising from, for example, random sensor failure, which is a common occurrence in some production settings, as confirmed by the expert we consulted from DataProphet.

Experimentally speaking, our research primarily builds on the DRQN paper's random masking experiments [16], which introduced partial observability by masking Atari game screens with zeros, as well as the 'Sensory Neuron' paper [25] wherein the authors train an attention-based PINN policy on a *fixed* random subset of patches from the game screen. We expand on these works in three ways. Firstly, we test the ability of RNN-augmented policy networks to handle partially masked observations, viewing an observation as a composition of multiple sensors. This approach is based on the PINN/Sensory Neuron paper and is applicable to real-world scenarios such as large-scale manufacturing. Secondly, we experiment with noise masking to simulate a corrupted signal from a faulty sensor, and forward fill masking, an imputation method for handling missing sensory signals. Lastly, we challenge the DRQN paper's concept of zero masking as "blindness". We argue that zero masking, along with other masking variants, should be seen as noise rather than an absence of information. This is because a novel masking ratio fundamentally alters the statistical relationship between the observations received by the agent during evaluation and those observed during training.

Additionally, our proposed attention-based policy architecture, which incorporates Transformer-like attention in a recurrent RL policy network, addresses gaps in the literature and builds on related work. Previous research has focused on using Transformer-like attention in policy networks to address the issue of partial observability in a manner which allows only for fixed-sized partial observations. These proposed methods utilise attention to: integrate information from partial observations into a memory matrix in a recurrent manner [20]; aggregate temporal information by performing attention over entire sequences of fixed-sized partial observations in a single shot [21] [92]; perform relational reasoning over abstract entities [22] [23], using RNNs or frame stacking to aggregate temporal information, or; determine patch/sensor importance for decision-making [24].

Our proposed architecture, which allows for variable-sized partial observations, builds on the PINN network proposed in the 'Sensory Neuron' paper [25] in the following way. The authors ([25]) trained an attention-based policy on fixed random subsets of patches of various arcade games, including CarRacing-v2, across various masking ratios, and evaluated them across the same set of masking ratios to test the agent's ability to generalise to novel degrees of partial observability, as we have done. Our work expands on this in two key ways. Firstly, our architecture is slightly simplified and more general, applicable to both vector and vision-based tasks by merely changing the embedding layer. Secondly, we train our attention network by randomly sampling different subsets of sensory signals at each time step during training, emulating the problem

of random sensor failure we aim to address, as opposed to training on a fixed subset of sensory signals. Lastly, whereas the PINN networks were trained with either evolutionary methods or behaviour cloning, presumably due to the difficulty of training attention-based policy architectures[1], we demonstrate that an attention agent can be successfully trained with the conventional gradient-based approach.

[1] Although the authors do not state their explicit reasoning for this choice.

7.2 Generated Masking

Generated masking seems to offer two potential benefits: (a) higher returns and improved sample efficiency under certain conditions due to the agent's ability to focus on the most salient sensory signals in the observation space for decision-making rather than having the sensory signals randomly sampled, and (b) interpretability, as we can explicitly see what the agent is focusing on in order to make decisions.

However, generated masking may also adversely affect the agent's performance under specific conditions. As noted in section 7.1, the number and nature of the sensory signals differ significantly between environments: the Acrobot-v1 environment admits only 6 sensory signals, each containing substantial information about the state of the underlying MDP, whereas the CarRacing-v2 environment admits 144 sensory signals[2], each containing only a small amount of information relative to the Acrbot-v1 task regarding the state of the underlying MDP. It is likely that, for these reasons, the agent was able to learn an effective mask generation policy, leading to improved sample efficiency and returns in the case of Acrobot-v1, but not in the case of CarRacing-v2.

Additionally, as seen in the Acrobot-v1 results, training under a high fixed masking ratio (communication bandwidth limit) and attempting to generalize to lower novel masking ratios at inference time can also lead to unexpected performance degradation. This is because the agent might learn to attend to only a subset of the sensory signals during training. This can result in poorly optimized embedding vectors for certain sensory signals (in the case of the dense attention agent). Consequently, when the agent samples the lesser-encountered sensory signals at inference time, these embedding vectors negatively impact the agent's action selection, leading to a drop in returns. One possible solution to this might be to train the attention agent not under a fixed masking ratio, but to vary the masking ratio during training. This would force the agent to encounter all sensory signals more often during training, while still allowing it to learn to focus on sensory signals which are most salient for decision-making.

Lastly, it is important to note that generated masking introduces additional complexity to the learning task, as well as additional policy parameters, which generally leads to slower learning (poorer sample efficiency). For this reason alone, it is not surprising that we observe a decrease in performance relative to an attention agent trained under random masking in some cases.

Overall, generated masking seems to offer advantages such as higher returns, improved sample efficiency, and interpretability, but only under specific conditions. These conditions are dependent on the environment, the training regime, and likely the method of implementation, of which we have only considered one in this dissertation. Therefore, further research is necessary to better understand generated masking in the context of attention-based policy architectures.

Our Generated Masking experiments build on previous research and fill a gap in the areas of hard attention and interpretability in RL. Our method of masking, a form of hard attention, allows our agent to explicitly select which sensors to attend to. This builds on research into architectures that employ hard attention mechanisms, such as that proposed in "Recurrent Models of Visual Attention" [14] and the Deep Attention Recurrent Q-Network proposed in [18]. However, in these architectures, the agent could only attend to a fixed-sized subset of the observation. Our attention mechanism gives our agent the ability to sample an arbitrary number of sensors without replacement. This property allows for more flexibility and could theoretically allow attention to be scaled up or down depending on the need and availability of computational resources. To our knowledge, this is the first example of a hard attention mechanism of this kind being used in conjunction with Transformer-like self-attention in the context of RL.

[2]To be clear: each frame is 96×96 pixels, which can be subdivided up into a grid of $12 \times 12 = 144$ patches, each of which constitutes a sensory signal.

Chapter 8

Conclusion

In this section, we provide a conclusion to the dissertation. In section 8.1, we provide answers to our research questions and an evaluation of our hypotheses in light of these answers. In section 8.2, we state the limitations of our work as we see them, followed by a discussion of possible avenues for future work in section 8.3. Finally, in section 8.4, we give our concluding remarks.

8.1 Answers to Research Questions

Research Question 1: *In the 'sensor failure and latency' scenario, what is the manner and extent of performance degradation suffered by conventional policy network architectures using the imputation methods of zero masking, noise masking, and forward-fill masking under different rates of sensor failure/latency (masking ratios)?*

Answer The performance degradation of conventional policy network architectures under different rates of sensor failure and latency, using zero masking, noise masking, and forward-fill masking, varied significantly. Noise masking had the most detrimental effect on performance during both training and evaluation, especially at high rates of sensor failure, across both environments, confirming **H1.1**. Zero masking proved to be less detrimental than forward-fill masking in the vector-based task, partially supporting **H1.2**. While its performance degradation during training was comparable to forward-fill masking in the vision-based task, its ability to generalize when evaluated under novel masking ratios was very poor. Forward-fill masking was arguably the most robust to sensor failure when considering the results across both environments, supporting **H1.3**. However, it also suffered performance degradation during both training and evaluation under high rates of sensor failure. In agreement with **H1.4**, higher rates of sensor failure consistently resulted in more severe performance degradation. However, the baseline agents exceeded expectations with respect to their robustness under high and moderate rates of sensor failure, which is almost certainly attributable to the RNN layer in the baseline policy architectures. Regarding **H1.5**, while high rates of sensor failure (and therefore imputation) resulted in performance degradation during training and evaluation across both environments, the impact of zero and noise masking on the ability of the baseline agent to generalize to novel rates of sensor failure in the vision-based task confirmed this hypothesis, that is, the performance degradation suffered by the baseline agents under the various imputation methods appeared to be more severe in general in the vision-based environment.

Research Question 2: *What advantages and disadvantages might self-attention-based policy network architectures offer over conventional policy network architectures in such conditions?*

Answer The attention-based policy architecture demonstrated a performance advantage over conventional architectures, especially at high rates of sensor failure, confirming **H2.1**. However, the disadvantage of multi-head attention in terms of sample efficiency and slower learning rates (**H2.2**) was also observed, particularly in the vision-based task and especially when compared with the performance of the forward-fill agent during evaluation under novel rates of sensor failure in this case.

Research Question 3: *In the 'low-bandwidth communication channel' scenario, what advantages and disadvantages are offered by the non-random generation of sensor queries over the random alternative?*

Answer The non-random generation of sensor queries seems to offer a slight advantage in terms of interpretability and the ability to focus on relevant sensors in the vector-based environment, leading to a modest performance improvement in some instances in the case of the vector-based task. However, we observed that when trained under low communication bandwidth (high masking ratio) and evaluated at a higher communication bandwidth (lower masking ratio) in the vector-based environment, the less frequently encountered sensory signals seemed to confuse the agent's decision-making. This confusion could be due to the inadequate optimization of the associated embedding vectors, highlighting an unexpected and significant potential drawback of using such a hard attention mechanism for explicit sensor querying. Moreover, the disadvantages of explicit (non-random) querying became particularly noticeable in the vision-based task, where the increased complexity of the task presumably led to performance during both training and evaluation that was, at best, equivalent to the random variant, if not inferior, supporting **H3.2**. Based on the overall results, **H3.1** is largely proven incorrect, as our method of explicit sensor querying appears to offer little to no advantage, generally speaking.

8.2 Limitations

In this section, we state the limitations of our work identified across both sets of experiments.

Our Random Masking experiments have a few limitations worth noting. First, we have only considered two types of environments, Acrobot-v1 and CarRacing-v2, which may limit the generalisability of our findings. Additionally, we have assumed (a) that the masking ratios remain constant during both training and evaluation and (b) that the probability of sensor failure is uniform across all sensors. This may not reflect real-world scenarios where sensor failure rates can vary over time and the probability of failure may differ between sensors. Lastly, we have used PPO as our learning algorithm and it is therefore unclear how the attention-based policy would perform when trained with other reinforcement learning algorithms, such as DQN.

In addition to those stated above, the main limitation of our Generated Masking experiments is that we have only explored one possible mechanism for querying sensors - sampling individual sensors without replacement. A possible direction for future work might be to implement an attention mechanism that more closely resembles that proposed in [14]. In this case, for vision-based tasks, the agent would be able to move a foveal window around the input space to sample a local subset of sensors at specific coordinates on the grid.

8.3 Future Work

In this section, we outline several potential avenues for further exploration and experimentation that could expand upon the research presented in this dissertation. Unfortunately, due to limitations in computational resources, budget, and time, as well as some aspects being beyond the scope of this study, we were unable to pursue these avenues. However, we believe they offer valuable directions for future research in this area:

1. **Experimentation With Additional, More Complex Environments:** Testing the model in more complex vector and vision-based environments could provide deeper insights into its capabilities and limitations. This could help in refining the model further and in understanding how it can be adapted to various types of tasks and challenges.

2. **Experiment With Non-Uniform Sensor Failure:** In our experiments we assumed the probability of failure was uniform across all sensors by sampling a fixed proportion of sensors with equal probability in our random masking experiments. In many real-world scenarios, it is likely that the probability of failure will vary between sensors - this experimental setup may be worth exploring in future in order to test the attention-based policy architectures in a more realistic setting.

3. **Exploring Different Training Algorithms:** Investigating neuro-evolution [24] and behaviour cloning [25] as optimisation algorithms could be fruitful. These methods have demonstrated exceptional results in terms of maximum returns and sample efficiency when applied to training attention-based policy architectures similar to the one we have developed.

4. **Experiment With Additional Querying Mechanisms:** In our generated masking experiments we only experimented with a single hard-attention mechanism for querying sensory signals. A possible direction for future work might include exploring alternatives, such as location-based querying in which the agent attends to local, connected regions of the input space as opposed to independent, disconnected sensors.

5. **Experimenting With the Attention Architecture:** We suggest exploring the addition of more attention blocks, inspired by the theoretical equivalence between CNNs and Transformer-like self-attention. Another avenue could involve replacing the residual connections in the attention block with the top-performing gating layers, as proposed in recent studies [21]. These modifications could potentially enhance the model's stability and sample efficiency during training.

6. **Augmentation of the Query Matrix:** Incorporating elements such as the previous action and reward into the query matrix - by concatenating along the column dimension of Q - could be beneficial, as demonstrated in the theory and literature on POMDPs and RL [17] [19]. Additionally, incorporating the hidden state from the previous time step into the query matrix might yield a performance improvement since incorporating information encoded by the historical sequence of partial observations may help the network determine how to better query the present partial observation. By directly informing the attention mechanism with these elements, the model might better focus on the most relevant information, potentially improving decision-making and learning efficiency.

7. **Improving Pixel Patch Embedding:** Enhancing the pixel patch embedding by increasing the patch size to 16×16 and replacing the linear patch embedding layer with a convolutional one could leverage the spatial bias inherent in convolutional layers. This approach would maintain independent processing of sensory signals while potentially improving the model's ability to interpret visual information.

These suggested areas for future work aim to build upon the foundation laid by this dissertation, offering pathways to enhance and extend the research in meaningful ways. In particular, we feel that items 3-7 might form a good basis for an ablation study of our proposed architecture.

8.4 Concluding Remarks

The central problem studied in this dissertation was the challenge of partial observability arising from intermittent sensor failure, latency, and low-bandwidth communication in control systems. In particular, we investigated the impact of sensor failure and latency, as well as various methods of imputation, on conventional policy architectures in two distinct reinforcement learning tasks and proposed an attention-based policy architecture as a robust alternative. We set out to demonstrate that attention-based architectures offer a promising, generalised alternative better suited to tasks of this nature, capable of handling variable-sized observations and eliminating the need for imputation. Additionally, we proposed a novel hard attention mechanism in the form of generated masking for the explicit (non-random) querying of sensory signals and sought to study its advantages and disadvantages over the random alternative.

Our findings confirmed that attention-based policy architectures present a significant advantage over conventional architectures, particularly under conditions of high partial observability. This advantage was most pronounced in the vector-based task, where the attention agent demonstrated robustness to missing sensory information resulting in superior performance during both training and evaluation under novel degrees of partial observability.

However, the research also highlighted the challenges associated with implementing attention-based architectures, such as the potential for poorer sample efficiency and slower learning rates due to the inclusion of multi-head attention, a finding which was highlighted in the vision-based task. Additionally, while generated masking offered potential benefits in terms of higher returns and improved sample efficiency under certain conditions, it also introduced additional complexity and did not always lead to improved performance.

In conclusion, while more research is required, our attention-based policy architecture offers a promising alternative for addressing the challenges of partial observability in reinforcement learning tasks. Future

work should explore architectural adjustments and alternative training algorithms to further enhance the performance and efficiency of such attention-based policies.

Appendix A

Supplementary Tables

A.1 Evaluation Returns - Random Masking - Acrobot

μ^{eval}	0/6	1/6	2/6	3/6	4/6	5/6
Attention Agent	-64 ± 13	-106 ± 29	-101 ± 30	-143 ± 45	-166 ± 55	-148 ± 63
Dense Agent (Zero Masking)	-64 ± 8	-120 ± 67	-144 ± 75	-169 ± 76	-214 ± 95	-290 ± 130
Dense Agent (Forward-Fill Masking)	-67 ± 10	-101 ± 33	-112 ± 38	-127 ± 47	-165 ± 69	-251 ± 95
Dense Agent (Noise Masking)	-63 ± 10	-106 ± 33	-123 ± 39	-153 ± 52	-203 ± 67	-315 ± 106

Table A.1.: Random masking experiments: Acrobot-V1 evaluation returns, μ^{train}=0/6.

μ^{eval}	0/6	1/6	2/6	3/6	4/6	5/6
Attention Agent	-70 ± 16	-72 ± 18	-71 ± 16	-102 ± 34	-124 ± 37	-97 ± 36
Dense Agent (Zero Masking)	-82 ± 23	-82 ± 23	-90 ± 29	-99 ± 28	-122 ± 35	-182 ± 45
Dense Agent (Forward-Fill Masking)	-76 ± 25	-78 ± 24	-85 ± 30	-97 ± 36	-124 ± 50	-207 ± 76
Dense Agent (Noise Masking)	-83 ± 31	-85 ± 28	-91 ± 31	-102 ± 32	-128 ± 36	-213 ± 67

Table A.2.: Random masking experiments: Acrobot-V1 evaluation returns, μ^{train}=1/6.

μ^{eval}	0/6	1/6	2/6	3/6	4/6	5/6
Attention Agent	-79 ± 19	-81 ± 23	-81 ± 22	-90 ± 26	-106 ± 33	-92 ± 31
Dense Agent (Zero Masking)	-86 ± 22	$-88 + 30$	-89 ± 28	-95 ± 31	-109 ± 37	-142 ± 51
Dense Agent (Forward-Fill Masking)	-81 ± 18	-81 ± 20	-82 ± 27	-88 ± 28	-109 ± 41	-185 ± 73
Dense Agent (Noise Masking)	-85 ± 23	-86 ± 24	-87 ± 26	-93 ± 31	-110 ± 34	-168 ± 67

Table A.3.: Random masking experiments: Acrobot-V1 evaluation returns, μ^{train}=2/6.

μ^{eval}	0/6	1/6	2/6	3/6	4/6	5/6
Attention Agent	-86 ± 23	-85 ± 23	-84 ± 22	-88 ± 24	-99 ± 31	-88 ± 34
Dense Agent (Zero Masking)	-92 ± 37	$-92 + 33$	-95 ± 39	-99 ± 41	-107 ± 37	-139 ± 41
Dense Agent (Forward-Fill Masking)	-89 ± 28	-88 ± 30	-86 ± 24	-90 ± 30	-107 ± 44	-176 ± 70
Dense Agent (Noise Masking)	-88 ± 28	-88 ± 27	-92 ± 34	-94 ± 29	-104 ± 35	-145 ± 50

Table A.4.: Random masking experiments: Acrobot-V1 evaluation returns, μ^{train}=3/6.

μ^{eval}	0/6	1/6	2/6	3/6	4/6	5/6
Attention Agent	-92 ± 26	-93 ± 28	-92 ± 30	-97 ± 30	-103 ± 32	-93 ± 32
Dense Agent (Zero Masking)	-91 ± 28	-92 ± 30	-94 ± 30	-98 ± 31	-103 ± 28	-127 ± 35
Dense Agent (Forward-Fill Masking)	-105 ± 40	-100 ± 35	-96 ± 33	-107 ± 41	-168 ± 70	-154 ± 72
Dense Agent (Noise Masking)	-89 ± 29	-91 ± 25	-97 ± 34	-100 ± 30	-111 ± 33	-157 ± 50

Table A.5.: Random masking experiments: Acrobot-V1 evaluation returns, μ^{train}=4/6.

μ^{eval}	0/6	1/6	2/6	3/6	4/6	5/6
Attention Agent	$-137 + 57$	-138 ± 56	-122 ± 41	$-138 + 52$	$-136 + 38$	-122 ± 43
Dense Agent (Zero Masking)	-155 ± 133	-160 ± 134	-169 ± 136	-174 ± 135	-189 ± 139	-217 ± 144
Dense Agent (Forward-Fill Masking)	-394 ± 135	-386 ± 142	-381 ± 147	-372 ± 150	-377 ± 150	-409 ± 129
Dense Agent (Noise Masking)	-499 ± 9	-499 ± 5	-500 ± 6	-500 ± 3	-499 ± 6	-499 ± 8

Table A.6.: Random masking experiments: Acrobot-V1 evaluation returns, μ^{train}=5/6.

A.2 Evaluation Returns - Random Masking - CarRacing

μ^{eval}	0.0	0.1	0.3	0.5	0.7	0.9	0.98
Attention Agent	734 ± 159	714 ± 171	718 ± 181	683 ± 181	615 ± 215	240 ± 263	-25 ± 52
Convolutional Agent (Zero Masking)	794 ± 126	-5 ± 59	-50 ± 21	-63 ± 25	-64 ± 27	-60 ± 29	-58 ± 29
Convolutional Agent (Forward-Fill Masking)	757 ± 164	754 ± 174	734 ± 184	760 ± 151	707 ± 189	368 ± 185	-10 ± 65
Convolutional Agent (Noise Masking)	794 ± 126	655 ± 181	33 ± 99	-49 ± 51	-75 ± 41	-91 ± 29	-92 ± 25

Table A.7.: Random masking experiments: CarRacing-v2 evaluation returns, μ^{train}=0.0.

μ^{eval}	0.0	0.1	0.3	0.5	0.7	0.9	0.98
Attention Agent	666 ± 182	667 ± 191	683 ± 188	668 ± 199	689 ± 198	311 ± 193	6 ± 81
Convolutional Agent (Zero Masking)	718 ± 181	726 ± 187	642 ± 201	387 ± 204	24 ± 73	-70 ± 34	-71 ± 34
Convolutional Agent (Forward-Fill Masking)	760 ± 159	745 ± 171	779 ± 144	750 ± 167	725 ± 181	317 ± 182	-31 ± 58
Convolutional Agent (Noise Masking)	758 ± 163	740 ± 181	595 ± 229	134 ± 153	-31 ± 62	-42 ± 55	-54 ± 38

Table A.8.: Random masking experiments: CarRacing-v2 evaluation returns, μ^{train}=0.1.

μ^{eval}	0.0	0.1	0.3	0.5	0.7	0.9	0.98
Attention Agent	590 ± 208	593 ± 195	596 ± 203	588 ± 198	620 ± 186	561 ± 197	99 ± 93
Convolutional Agent (Zero Masking)	719 ± 179	723 ± 179	725 ± 182	692 ± 194	211 ± 235	-70 ± 30	-62 ± 28
Convolutional Agent (Forward-Fill Masking)	769 ± 158	762 ± 165	778 ± 161	753 ± 169	727 ± 168	366 ± 169	-19 ± 62
Convolutional Agent (Noise Masking)	674 ± 210	686 ± 199	730 ± 168	675 ± 193	122 ± 139	-62 ± 44	-57 ± 45

Table A.9.: Random masking experiments: CarRacing-v2 evaluation returns, μ^{train}=0.3.

μ^{eval}	0.0	0.1	0.3	0.5	0.7	0.9	0.98
Attention Agent	578 ± 202	583 ± 200	584 ± 211	599 ± 196	569 ± 231	411 ± 223	50 ± 81
Convolutional Agent (Zero Masking)	681 ± 175	709 ± 165	723 ± 157	697 ± 176	578 ± 228	-63 ± 41	-44 ± 44
Convolutional Agent (Forward-Fill Masking)	727 ± 184	730 ± 186	741 ± 176	694 ± 195	704 ± 215	386 ± 171	-10 ± 69
Convolutional Agent (Noise Masking)	591 ± 206	614 ± 204	641 ± 210	696 ± 191	470 ± 269	-30 ± 69	-68 ± 59

Table A.10.: Random masking experiments: CarRacing-v2 evaluation returns, μ^{train}=0.5.

μ^{eval}	0.0	0.1	0.3	0.5	0.7	0.9	0.98
Attention Agent	590 ± 209	584 ± 205	596 ± 190	584 ± 209	588 ± 190	482 ± 204	61 ± 97
Convolutional Agent (Zero Masking)	642 ± 201	665 ± 196	689 ± 177	679 ± 186	640 ± 210	29 ± 135	-90 ± 3
Convolutional Agent (Forward-Fill Masking)	777 ± 154	779 ± 143	762 ± 158	763 ± 158	729 ± 170	414 ± 181	-2 ± 65
Convolutional Agent (Noise Masking)	547 ± 201	542 ± 205	571 ± 202	582 ± 200	639 ± 215	28 ± 85	-42 ± 45

Table A.11.: Random masking experiments: CarRacing-v2 evaluation returns, μ^{train}=0.7.

μ^{eval}	0.0	0.1	0.3	0.5	0.7	0.9	0.98
Attention Agent	511 ± 184	486 ± 195	545 ± 190	517 ± 199	550 ± 196	476 ± 206	115 ± 135
Convolutional Agent (Zero Masking)	46 ± 142	103 ± 160	330 ± 197	534 ± 205	566 ± 193	533 ± 213	-71 ± 30
Convolutional Agent (Forward-Fill Masking)	712 ± 177	711 ± 165	723 ± 164	718 ± 175	691 ± 172	600 ± 183	20 ± 87
Convolutional Agent (Noise Masking)	10 ± 72	31 ± 89	63 ± 123	87 ± 172	90 ± 199	96 ± 186	-32 ± 56

Table A.12.: Random masking experiments: CarRacing-v2 evaluation returns, μ^{train}=0.9.

μ^{eval}	0.0	0.1	0.3	0.5	0.7	0.9	0.98
Attention Agent	457 ± 193	440 ± 220	432 ± 212	442 ± 203	473 ± 198	493 ± 201	476 ± 189
Convolutional Agent (Zero Masking)	-32 ± 56	1 ± 88	158 ± 196	338 ± 283	342 ± 269	379 ± 206	398 ± 212
Convolutional Agent (Forward-Fill Masking)	556 ± 222	537 ± 215	512 ± 219	537 ± 210	539 ± 217	493 ± 211	278 ± 164
Convolutional Agent (Noise Masking)	-8 ± 37	-9 ± 37	-7 ± 38	-5 ± 42	-4 ± 40	-7 ± 41	-3 ± 38

Table A.13.: Random masking experiments: CarRacing-v2 evaluation returns, μ^{train}=0.98

A.3 Evaluation Returns - Generated Masking - Acrobot

μ^{eval}	0/6	1/6	2/6	3/6	4/6	5/6
Random Masking	-63 ± 13	-107 ± 29	-102 ± 26	-141 ± 42	-169 ± 59	-146 ± 58
Generated Masking	-64 ± 15	-99 ± 35	-99 ± 35	-135 ± 47	-146 ± 60	-146 ± 60

Table A.14.: Generated masking experiments: Acrobot-v1 evaluation returns, μ^{train}=0/6

μ^{eval}	0/6	1/6	2/6	3/6	4/6	5/6
Random Masking	-70 ± 15	-72 ± 16	-72 ± 18	-103 ± 38	-123 ± 37	-97 ± 39
Generated Masking	-75 ± 20	-74 ± 16	-74 ± 16	-100 ± 31	-103 ± 39	-103 ± 39

Table A.15.: Generated masking experiments: Acrobot-v1 evaluation returns, μ^{train}=1/6

μ^{eval}	0/6	1/6	2/6	3/6	4/6	5/6
Random Masking	-70 ± 15	-72 ± 16	-72 ± 18	-103 ± 38	-123 ± 37	-97 ± 39
Generated Masking	-75 ± 20	-74 ± 16	-74 ± 16	-100 ± 31	-103 ± 39	-103 ± 39

Table A.16.: Generated masking experiments: Acrobot-v1 evaluation returns, μ^{train}=2/6

μ^{eval}	0/6	1/6	2/6	3/6	4/6	5/6
Random Masking	-81 ± 26	-81 ± 24	-81 ± 22	-90 ± 27	-105 ± 28	-93 ± 35
Generated Masking	-82 ± 26	-82 ± 23	-82 ± 23	-88 ± 27	-88 ± 30	-88 ± 30

Table A.17.: Generated masking experiments: Acrobot-v1 evaluation returns, μ^{train}=3/6

μ^{eval}	0/6	1/6	2/6	3/6	4/6	5/6
Random Masking	-92 ± 31	-93 ± 25	-92 ± 28	-97 ± 29	-103 ± 26	-93 ± 29
Generated Masking	-101 ± 48	-92 ± 33	-92 ± 33	-90 ± 34	-89 ± 32	-89 ± 32

Table A.18.: Generated masking experiments: Acrobot-v1 evaluation returns, μ^{train}=4/6

μ^{eval}	0/6	1/6	2/6	3/6	4/6	5/6
Random Masking	-136 ± 56	-136 ± 53	-124 ± 45	-139 ± 55	-136 ± 44	-121 ± 39
Generated Masking	-208 ± 122	-173 ± 100	-173 ± 100	-252 ± 105	-107 ± 37	-107 ± 37

Table A.19.: Generated masking experiments: Acrobot-v1 evaluation returns, μ^{train}=5/6

A.4 Evaluation Returns - Generated Masking - CarRacing

μ^{eval}	0.0	0.1	0.3	0.5	0.7	0.9
Random Masking	734 ± 159	714 ± 171	718 ± 181	683 ± 181	615 ± 215	240 ± 263
Generated Masking	690 ± 172	688 ± 176	620 ± 234	474 ± 333	433 ± 334	134 ± 212

Table A.20.: Generated masking experiments: CarRacing-v2 evaluation returns, μ^{train}=0.0

μ^{eval}	0.0	0.1	0.3	0.5	0.7	0.9
Random Masking	666 ± 182	667 ± 191	683 ± 188	668 ± 199	689 ± 198	311 ± 193
Generated Masking	672 ± 187	674 ± 187	687 ± 184	553 ± 233	584 ± 208	329 ± 217

Table A.21.: Generated masking experiments: CarRacing-v2 evaluation returns, μ^{train}=0.1

μ^{eval}	0.0	0.1	0.3	0.5	0.7	0.9
Random Masking	590 ± 208	593 ± 195	596 ± 203	588 ± 198	620 ± 186	561 ± 197
Generated Masking	606 ± 184	615 ± 203	594 ± 194	597 ± 191	598 ± 198	394 ± 221

Table A.22.: Generated masking experiments: CarRacing-v2 evaluation returns, μ^{train}=0.3

μ^{eval}	0.0	0.1	0.3	0.5	0.7	0.9
Random Masking	578 ± 202	583 ± 200	584 ± 211	599 ± 196	569 ± 231	411 ± 223
Generated Masking	570 ± 200	569 ± 216	582 ± 202	594 ± 194	609 ± 202	416 ± 179

Table A.23.: Generated masking experiments: CarRacing-v2 evaluation returns, μ^{train}=0.5

μ^{eval}	0.0	0.1	0.3	0.5	0.7	0.9
Random Masking	590 ± 209	584 ± 205	596 ± 190	584 ± 209	588 ± 190	482 ± 204
Generated Masking	606 ± 189	614 ± 179	617 ± 178	584 ± 183	598 ± 188	434 ± 208

Table A.24.: Generated masking experiments: CarRacing-v2 evaluation returns, μ^{train}=0.7

μ^{eval}	0.0	0.1	0.3	0.5	0.7	0.9
Random Masking	511 ± 184	486 ± 195	545 ± 190	517 ± 199	550 ± 196	476 ± 206
Generated Masking	501 ± 220	489 ± 230	522 ± 226	497 ± 239	496 ± 216	508 ± 214

Table A.25.: Generated masking experiments: CarRacing-v2 evaluation returns, μ^{train}=0.9

Appendix B

Experiment Code

The code used to run the experiments may be found here:
`https://github.com/JeremyDouglas91/msc-dissertation-code`.

Bibliography

[1] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[2] Jan 2024.

[3] Ian Osband, Yotam Doron, Matteo Hessel, John Aslanides, Eren Sezener, Andre Saraiva, Katrina McKinney, Tor Lattimore, Csaba Szepesvari, Satinder Singh, et al. Behaviour suite for reinforcement learning. *arXiv preprint arXiv:1908.03568*, 2019.

[4] Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1998.

[5] Anil Ananthaswamy. Researchers build ai that builds ai, Jan 2022.

[6] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. " O'Reilly Media, Inc.", 2022.

[7] Andrea Lörke. Cybenko's theorem and the capability of a neural network as function approximator, 2019.

[8] MathWorks. How cnns work.

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[12] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[13] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 16000–16009, 2022.

[14] Volodymyr Mnih, Nicolas Heess, Alex Graves, et al. Recurrent models of visual attention. *Advances in neural information processing systems*, 27, 2014.

[15] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057. PMLR, 2015.

[16] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *2015 aaai fall symposium series*, 2015.

[17] Pengfei Zhu, Xin Li, Pascal Poupart, and Guanghui Miao. On improving deep reinforcement learning for pomdps. *arXiv preprint arXiv:1704.07978*, 2017.

[18] Ivan Sorokin, Alexey Seleznev, Mikhail Pavlov, Aleksandr Fedorov, and Anastasiia Ignateva. Deep attention recurrent q-network. *arXiv preprint arXiv:1512.01693*, 2015.

[19] Alexander Mott, Daniel Zoran, Mike Chrzanowski, Daan Wierstra, and Danilo Jimenez Rezende. Towards interpretable reinforcement learning using attention augmented agents. *Advances in neural information processing systems*, 32, 2019.

[20] Adam Santoro, Ryan Faulkner, David Raposo, Jack Rae, Mike Chrzanowski, Theophane Weber, Daan Wierstra, Oriol Vinyals, Razvan Pascanu, and Timothy Lillicrap. Relational recurrent neural networks. *Advances in neural information processing systems*, 31, 2018.

[21] Emilio Parisotto, Francis Song, Jack Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant Jayakumar, Max Jaderberg, Raphael Lopez Kaufman, Aidan Clark, Seb Noury, et al. Stabilizing transformers for reinforcement learning. In *International conference on machine learning*, pages 7487–7498. PMLR, 2020.

[22] Vinicius Zambaldi, David Raposo, Adam Santoro, Victor Bapst, Yujia Li, Igor Babuschkin, Karl Tuyls, David Reichert, Timothy Lillicrap, Edward Lockhart, et al. Deep reinforcement learning with relational inductive biases. In *International conference on learning representations*, 2018.

[23] Anthony Manchin, Ehsan Abbasnejad, and Anton Van Den Hengel. Reinforcement learning with attention that works: A self-supervised approach. In *Neural Information Processing: 26th International Conference, ICONIP 2019, Sydney, NSW, Australia, December 12–15, 2019, Proceedings, Part V 26*, pages 223–230. Springer, 2019.

[24] Yujin Tang, Duong Nguyen, and David Ha. Neuroevolution of self-interpretable agents. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, pages 414–424, 2020.

[25] Yujin Tang and David Ha. The sensory neuron as a transformer: Permutation-invariant neural networks for reinforcement learning. *Advances in Neural Information Processing Systems*, 34:22574–22587, 2021.

[26] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems*, 34:15084–15097, 2021.

[27] Helga Kolb, Eduardo Fernandez, and Ralph Nelson. Webvision: the organization of the retina and visual system [internet]. 1995.

[28] Robert Desimone and John Duncan. Neural mechanisms of selective visual attention. *Annual review of neuroscience*, 18(1):193–222, 1995.

[29] Richard Bellman. Dynamic programming and stochastic control processes. *Information and control*, 1(3):228–239, 1958.

[30] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[31] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[33] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.

[34] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[35] Frank Rosenblatt et al. *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms*, volume 55. Spartan books Washington, DC, 1962.

[36] Paul Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *PhD thesis, Committee on Applied Mathematics, Harvard University, Cambridge, MA*, 1974.

[37] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[38] Frank E Curtis and Katya Scheinberg. Optimization methods for supervised machine learning: From linear models to deep learning. In *Leading Developments from INFORMS Communities*, pages 89–114. INFORMS, 2017.

[39] Robin M Schmidt, Frank Schneider, and Philipp Hennig. Descending through a crowded valley-benchmarking deep learning optimizers. In *International Conference on Machine Learning*, pages 9367–9376. PMLR, 2021.

[40] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[41] Yann LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–50. Springer, 2002.

[42] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[43] Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta numerica*, 8:143–195, 1999.

[44] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[45] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[46] Moshe Leshno, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993.

[47] Balázs Csanád Csáji et al. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, 24(48):7, 2001.

[48] Ding-Xuan Zhou. Universality of deep convolutional neural networks. *Applied and computational harmonic analysis*, 48(2):787–794, 2020.

[49] Anton Maximilian Schäfer and Hans Georg Zimmermann. Recurrent neural networks are universal approximators. In *Artificial Neural Networks–ICANN 2006: 16th International Conference, Athens, Greece, September 10-14, 2006. Proceedings, Part I 16*, pages 632–640. Springer, 2006.

[50] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[51] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[52] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[53] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[54] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[55] Zihao Wang and Lei Wu. Theoretical analysis of inductive biases in deep convolutional networks. *arXiv preprint arXiv:2305.08404*, 2023.

[56] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[57] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[58] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[59] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[60] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[61] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[62] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[63] Gerald Tesauro et al. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[64] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

[65] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

[66] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *Machine Learning: ECML 2005: 16th European Conference on Machine Learning, Porto, Portugal, October 3-7, 2005. Proceedings 16*, pages 317–328. Springer, 2005.

[67] Michael McCloskey and Neal J Cohen. Catastrophic interference in connectionist networks: The sequential learning problem, 1989.

[68] Robert M French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999.

[69] Benjamin Frederick Goodrich. Neuron clustering for mitigating catastrophic forgetting in supervised and reinforcement learning. 2015.

[70] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[71] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[72] Edward Jay Sondik. *The optimal control of partially observable Markov processes*. Stanford University, 1971.

[73] Guy Shani, Joelle Pineau, and Robert Kaplow. A survey of point-based pomdp solvers. *Autonomous Agents and Multi-Agent Systems*, 27:1–51, 2013.

[74] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[75] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[76] Shigeki Karita, Nanxin Chen, Tomoki Hayashi, Takaaki Hori, Hirofumi Inaguma, Ziyan Jiang, Masao Someki, Nelson Enrique Yalta Soplin, Ryuichi Yamamoto, Xiaofei Wang, et al. A comparative study on transformer vs rnn in speech applications. In *2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pages 449–456. IEEE, 2019.

[77] Ailing Zeng, Muxi Chen, Lei Zhang, and Qiang Xu. Are transformers effective for time series forecasting? In *Proceedings of the AAAI conference on artificial intelligence*, volume 37, pages 11121–11128, 2023.

[78] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. A comprehensive survey on transfer learning. *Proceedings of the IEEE*, 109(1):43–76, 2020.

[79] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[80] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[81] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[82] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[83] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[84] Jean-Baptiste Cordonnier, Andreas Loukas, and Martin Jaggi. On the relationship between self-attention and convolutional layers. *arXiv preprint arXiv:1911.03584*, 2019.

[85] Samuel L Smith, Andrew Brock, Leonard Berrada, and Soham De. Convnets match vision transformers at scale. *arXiv preprint arXiv:2310.16764*, 2023.

[86] Bram Bakker. Reinforcement learning with long short-term memory. *Advances in neural information processing systems*, 14, 2001.

[87] Daan Wierstra, Alexander Foerster, Jan Peters, and Juergen Schmidhuber. Solving deep memory pomdps with recurrent policy gradients. In *Artificial Neural Networks–ICANN 2007: 17th International Conference, Porto, Portugal, September 9-13, 2007, Proceedings, Part I 17*, pages 697–706. Springer, 2007.

[88] Marco Pleines, Matthias Pallasch, Frank Zimmer, and Mike Preuss. Generalization, mayhems and limits in recurrent proximal policy optimization. *arXiv preprint arXiv:2205.11104*, 2022.

[89] Steven Morad, Ryan Kortvelesy, Matteo Bettini, Stephan Liwicki, and Amanda Prorok. Popgym: Benchmarking partially observable reinforcement learning. *arXiv preprint arXiv:2303.01859*, 2023.

[90] Claire Glanois, Paul Weng, Matthieu Zimmer, Dong Li, Tianpei Yang, Jianye Hao, and Wulong Liu. A survey on interpretable reinforcement learning. *arXiv preprint arXiv:2112.13112*, 2021.

[91] Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. A simple neural attentive meta-learner. *arXiv preprint arXiv:1707.03141*, 2017.

[92] Kevin Esslinger, Robert Platt, and Christopher Amato. Deep transformer q-networks for partially observable reinforcement learning. *arXiv preprint arXiv:2206.01078*, 2022.

[93] Jürgen Schmidhuber. Discovering neural nets with low kolmogorov complexity and high generalization capability. *Neural Networks*, 10(5):857–873, 1997.

[94] Kenneth O Stanley and Risto Miikkulainen. A taxonomy for artificial embryogeny. *Artificial life*, 9(2):93–130, 2003.

[95] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International conference on machine learning*, pages 3744–3753. PMLR, 2019.

[96] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10561*, 2017.

[97] Juergen Schmidhuber. Reinforcement learning upside down: Don't predict rewards–just map them to actions. *arXiv preprint arXiv:1912.02875*, 2019.

[98] Micah Carroll, Orr Paradise, Jessy Lin, Raluca Georgescu, Mingfei Sun, David Bignell, Stephanie Milani, Katja Hofmann, Matthew Hausknecht, Anca Dragan, et al. Uni [mask]: Unified inference in sequential decision problems. *Advances in neural information processing systems*, 35:35365–35378, 2022.

[99] Fangchen Liu, Hao Liu, Aditya Grover, and Pieter Abbeel. Masked autoencoding for scalable and generalizable decision making. *Advances in Neural Information Processing Systems*, 35:12608–12618, 2022.

[100] Dan Crisan. The stochastic filtering problem: a brief historical account. *Journal of Applied Probability*, 51(A):13–22, 2014.

[101] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *ASME–Journal of Basic Engineering, 82 (Series D)*, pages 35–45, 1960.

[102] Xiangheng Liu and Andrea Goldsmith. Kalman filtering with partial observation losses. In *2004 43rd IEEE Conference on Decision and Control (CDC)(IEEE Cat. No. 04CH37601)*, volume 4, pages 4180–4186. IEEE, 2004.

[103] Bruno Sinopoli, Luca Schenato, Massimo Franceschetti, Kameshwar Poolla, Michael I Jordan, and Shankar S Sastry. Kalman filtering with intermittent observations. *IEEE transactions on Automatic Control*, 49(9):1453–1464, 2004.

[104] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[105] Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022.

[106] OpenAI. Openai baselines. `https://github.com/openai/baselines`, 2017.

[107] Chris Lu. Purejaxrl. `https://github.com/luchris429/purejaxrl`, 2023.

[108] Facebook Research. Masked autoencoders: A pytorch implementation. `https://github.com/openai/baselines`, 2021.

[109] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. One pixel attack for fooling deep neural networks. *IEEE Transactions on Evolutionary Computation*, 23(5):828–841, 2019.