Reinforcement Learning via Predictive Coding: Revisiting Perspectives from Neuroscience in the Context of Deep Learning

by
Alex Goodall

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
Advanced Computer Science

at the
University of Oxford
Trinity Term 2022

Reinforcement Learning via Predictive Coding: Revisiting Perspectives from Neuroscience in the Context of Deep Learning

Abstract

Reinforcement learning (RL) is an important paradigm of machine learning (ML) built on strong fundamental theories and perspectives from neuroscience and statistics. Built on recent successes in deep learning (DL), RL has blossomed into an exciting area of research that shows incredible promise in reaching the goal of artificial general intelligence (AGI). While there remain many outstanding problems to address in RL, we aim to shed light on the problem of "catastrophic interference" by revisiting an increasingly popular theory of brain function known as predictive coding. In this thesis, we conduct preliminary research into the application of predictive coding networks (PCNs) for a variety of classic RL tasks. Specifically, we examine the convergence and stability of both value-based and policy-gradient methods trained with predictive coding, in comparison to the typical deep function approximators trained with backpropagation (BP). After sufficient hyperparameter tuning, we demonstrate that in a number cases, PCNs achieve no worse and sometimes better performance than neural networks (NNs) trained with BP. In addition to these results, we first establish the rich theory and intuition behind major topics discussed in this thesis. We conclude with a discussion summarising the insights we have obtained and most immediate challenges associated with predictive coding. Finally, we outline the most important directions for future work.

Declaration

I declare that this thesis is entirely my own work, and except where otherwise stated, describes my own research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Deep learning (DL) has become an incredibly active area of research that stems from two greatly related and important fields, namely, *machine learning* (ML) and *artificial intelligence* (AI). DL has found incredible success in supervised and unsupervised machine learning tasks such as classification [1], regression [2] and latent representation learning [3]. As a result deep learning has been adopted by a variety of disciplines, such as computer vision, signal processing, natural language processing, robotics, computational biology and finance among others [4, p. 9].

Much of the success of DL can be attributed to the adoption of GPU computing and the ever increasing access to large amounts of training data [5]. Most of the computational models used in DL come under the umbrella term of *artificial neural networks* (ANNs). This nomenclature can be traced back to the *perceptron* [6], an early model of learning and one of the fundamental building blocks for modern deep learning architectures that was originally intended to mimic biological learning in the brain [4, p. 13]. While the results obtained by DL in domain specific tasks are impressive, we are still far from developing generalised systems that emulate human-level (or better) intelligence and learning

capabilities [7].

When we think about biologically inspired AI, *convolutional neural networks* (CNNs) [8] are probably one of the first major successes that spring to mind. Results obtained by neurophysiologists and nobel laureates David Hubel and Torsten Wiesel [9–11] helped characterize neural processing in the visual cortex, which lead to the hypothesis that visual processing in the brain occurs in a hierarchical fashion, whereby neurons in the early visual cortex respond to simple patterns such as precisely oriented bars and shapes, and the latter neurons process increasingly complex patterns and textures [4, p. 353]. By studying the learned convolution filters of some of the first deep CNNs such as AlexNet [1], we see that this hypothesis is substantiated. In fact, AlexNet was such a significant breakthrough that CNNs have now been adopted by the computer vision community for almost every visual task. Additionally, the *backpropagation* (BP) algorithm [12] used to train AlexNet along with regularization techniques such as dropout [13] have become fundamental for the successful training of deep neural networks.

In this thesis, our main concern is *reinforcement learning* (RL): an important paradigm of ML, also with strong links to neuroscience. Inspired by the *reward prediction error hypothesis of dopamine neuron activity*, RL aims to minimise temporal difference (TD) errors observed by interacting with an environment to learn how to best pick actions [14, p. 381]. Dopamine in the brain appears to be sensitive to reward signals and how far in the future we expect to see them, which effectively conveys TD errors [15]. By adopting this notion of learning and by using deep function approximators such as ANNs, RL has found remarkable success in tasks such as optimal control and robotics [16–18], game playing [19–23] and financial decision making [24].

This goal-directed view of learning offered by RL has strong links to psychology and neuroscience, and so we can be almost certain RL has a big part to play in the development of artificial general intelligence (AGI) [14, p. 472]. However, this does not mean we can stop listening to neuroscience, as there are many outstanding issues surrounding deep

reinforcement learning that need to be addressed. Modern methods based on DL and ANNs trained with BP are not well suited to online incremental learning and so many of the best algorithms learn offline, on batches of interleaved experience from multiple workers, interacting with different instances of the same environment at the same time [14, p. 472]. This key problem referred to as "catastrophic interference" or "correlated data" is often overlooked, and it stems from the common phenomena: when something new is learnt it often replaces or interferes with past experience [14, p. 472]. The typical work around is to use *replay buffers*, which retain old experience that we may learn from again during training. It should not be hard to realise, based on one's own experience, that biological learning does not follow this procedure, and us humans typically do not need anywhere near as much experience as modern RL algorithms do [25].

There is increasing belief that the problem of "catastrophic interference" is a symptom of the learning algorithm *backpropagation* (BP) used to update most modern neural architectures [26–28]. In addition, BP has historically been viewed as problematic and not biologically sound [29, 30] even though brain function may closely approximate it [31, 32] and so once again we may need to look to neuroscience for an answer. *Predictive coding* (PC) is a theory of brain function first used as a means of explaining hierarchical image processing in the visual cortex [33–35]. While experimental evidence for PC is often varied [36–38], it has had striking success on tasks such as image classification [39], information retrieval [40] and associative memories [41]. Furthermore, PC has promise in addressing the problem of "catastrophic interferenc", as in a recent study Song *et al.* demonstrated superior performance in a number of biologically relevant tasks when compared to BP [28]. This is an exciting direction of research that revisits how we understand biological learning in relation to deep learning and it offers us a new perspective that could likely become vital for the development of AGI.

1.2 Contributions

The goal of this thesis is to understand and empirically validate the advantages of predictive coding over backpropagation in typical deep reinforcement learning settings, such as grid worlds, game playing and continuous control. Our main contributions include the following:

- We provide a detailed introduction to artificial neural networks, energy-based models, predictive coding networks [40–42] and the mathematical formalisms of reinforcement learning, along with a brief dive into some state-of-the-art algorithms.
- We demonstrate that directed networks trained with PC comparable performance to networks trained with BP in a number of toy scenarios when parameter updates to the network are made with *stochastic gradient descent* (SGD). In certain circumstances, we show experimentally, that networks trained with PC boast quicker and more stable convergence than networks trained with BP, when parameter updates are also made with SGD.
- We apply PC to *deep Q-networks* (DQNs) [19] trained with the *Adam* optimiser [43] to learn good policies for two Atari 2600 games in the *Arcade Learning Environment* (ALE) [44]. While the results are disappointing, we are able to draw several insights and establish important directions for future research.
- We apply PC to policy networks trained with Adam and the proximal policy optimisation (PPO) algorithm [45] for learning good policies in two complex continuous control tasks. We demonstrate the PC and BP achieve comparable performance on two different continuous control tasks from the MuJoCo benchmark [46].

Although we believe with additional tuning, PC can maintain better and more stable performance. In addition to our experimental contributions, we aim to provide valuable insight and conclusions drawn from our results. Whether successful or not, we hope that our experiments are clear, informative and provide motivation for further research.

1.3 Organisation

This thesis is composed of five chapters, including this one. Chapter 2, the subsequent chapter, provides the reader with all the prerequisite information and related work needed to understand the remaining two chapters. We start Chapter 2 by formally characterising artificial neural networks and the backpropagation algorithm. Then we characterise both discriminative and generative predictive coding networks under the energy-based model (EBM) framework. Chapter 2, concludes with a far from complete but thorough introduction to reinforcement learning, the fundamental mathematical formalisms, and key algorithms and ideas needed for the remaining chapters of this thesis. Additional material, such as the derivations of the gradient steps for BP and PC can be found in the appendices.

Chapter 3 presents the main algorithms in this thesis, that are used for comparing the performance of BP and PC in a number of RL settings. In addition, we describe the algorithmic modifications that we propose to facilitate better learning with PC. Chapter 3 then concludes with a concise description of the evaluation framework we use to compare the performance of BP and PC in RL settings. Chapter 4 presents the results of several experiments that directly compare the performance of networks trained with BP to networks trained with PC in a number of typical RL settings. Additionally, we provide a brief discussion of the results and we explain any additional setup where necessary. Chapter 5 concludes this thesis, in this chapter we summarise the insights drawn from our experiments and set out the most important directions for future work.

Chapter 2

Background and Preliminaries

In this chapter we will start by introducing some of the most fundamental concepts and results in deep learning (DL) that will be relevant to the work in this thesis. We will then cover predictive coding and energy-based models (EBMs). Specifically, we will introduce the EBM framework by studying Hopfield networks, we will then formalise discriminative and generative predictive coding networks (PCNs) with their corresponding energy-based interpretations. Finally, we will conclude this chapter with a thorough introduction to reinforcement learning (RL), starting from the fundamentals to important algorithms and results.

2.1 Artificial Neural Networks

The term *neural network* broadly captures most popular deep learning models, all of which have been developed as a result of a culmination of centuries of progress in the problem of function approximation [4, p. 217]. Some fundamental successes in history that have contributed to the popularity of neural networks include: the very first mathematical model of the biological neuron [47], the perceptron learning algorithm [6], efficient implementations of the chain rule of calculus [12] and the successful training of deep networks based on the contrastive divergence procedure [48]. Until recently, support vector machines (SVMs)

[49] were the go-to model for classification tasks; they are easier to train and tune since their objective function is convex. However, after over a decade of successes it has become clear that neural networks are here to stay, and with advances in optimization techniques and design of learning algorithms, we expect to see neural networks continue to fulfill their potential [4, p. 220].

The goal of neural networks is to approximate some function $y = f^*(\mathbf{x})$ that maps an input $\mathbf{x}$ to an output y. Neural networks consist of layers of *neurons* or *units* that process information from input to output, typically in one direction. Each neuron has an associated set of *weights* and *biases* meant to model the neuron's firing rate given some input. Together all these weights and biases parameterise the neural network, typically these parameters are denoted by $\boldsymbol{\theta}$. So a neural network parameterised by $\boldsymbol{\theta}$ defines a mapping $y = f(\mathbf{x}; \boldsymbol{\theta})$. The ability for the strength of synaptic connections in the brain to change is referred to as *synaptic plasticity*, and parameter updates to $\boldsymbol{\theta}$ adjusted by learning algorithms correspond to this ability [14, p. 379]. In the end we wish to learn the set of parameters $\boldsymbol{\theta}^*$ that best approximate the function f^*. Networks of this form are typically referred to as feed-forward neural networks since their computation can be represented as a *directed acyclic graph* (DAG). Networks with loops or feedback connections are called *recurrent neural networks* (RNNs) and are beyond the scope of this work [4, Ch. 10].

Many different types of layers exist in the DL literature and the particular choice and ordering of layers is referred to as the *architecture* of a neural network. Most layers are typically followed by a non-linear *activation function* g that models the spiking or firing of a neuron in the brain. In theory, the use of non-linear activation functions allows neural networks to capture complex non-linear relationships; if training is done correctly this makes them powerful models for many machine learning tasks. The choice of which activation function to use is important, for example, the adoption of the *rectified linear unit* (ReLU) activation function over the sigmoid function is a major landmark in the DL literature [4, p. 219]. This choice was actually biologically motivated since the ReLU

function $f(\mathbf{x}) = \mathsf{max}(0, \mathbf{x})$ captures two important properties of biological neurons: most of the time biological neurons are inactive, and for some inputs biological neurons fire at a rate proportional to their input [50]. While it seems much of the success related to neural networks is biologically inspired, it is important to note that neural networks are not intended to be realistic models of the brain, but are rather loosely inspired by results and hypothesis related to neural activity, brain function and learning in the brain.

In this section, we will start by covering the fundamental building blocks that make up neural networks, before describing gradient based optimisation techniques and the backpropagation algorithm. We will then conclude this section by outlining the basic operations of *convolutional neural networks* (CNNs), which will be relevant for some of the experiments in this thesis.

2.1.1 The Perceptron

The perceptron [6] is a learning algorithm for classifying inputs in some vector space into one of two distinct classes $\{-1, +1\}$. The perceptron is based on the McCulloch-Pitts neuron [47] - a simple mathematical model which approximates the neural activity of a neuron by a weighted sum of its inputs passed through a threshold function [51, p. 569],

$$y = \mathbb{1}\left(\sum_{i=1} w_i \cdot x_i > \lambda\right) \tag{2.1}$$

for some threshold λ. Here each x_i corresponds to an input feature and each w_i corresponds to its associated weight. The perceptron extends this idea by adding a bias term b and replacing the threshold function with the sign function. The *pre-activation* value z is computed by taking a linear combination of the inputs $x_1, ..., x_D$ plus the bias term b. The output value or prediction $\hat{y}$ (sometimes called the *activation* and denoted a) is computed by passing z through the sign function, classifying it into one of $\{-1, +1\}$, see Figure 2.1 for a visual representation of this computation.

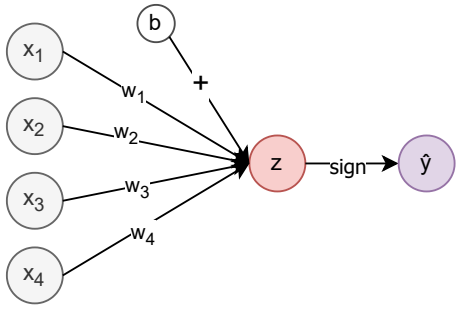

Figure 2.1: Perceptron with input dimension $D = 4$.

In machine learning tasks like classification, we are typically given a dataset of input output-pairs, $\mathcal{D} = \langle(\mathbf{x}_i, y_i)\rangle_{i=1}^{n}$, the goal is to learn the parameters of the model that best match the dataset. Algorithm 1 presents the original perceptron algorithm [6] for classifying some given dataset $\mathcal{D}$. The perceptron algorithm can be applied to a rigid dataset (fixed size) or it can be applied in an online fashion (to infinity and beyond). Either way, the perceptron algorithm has been show to maintain nice convergence guarantees in both these settings [52, 53].

Algorithm 1 The Perceptron Algorithm

Input: Fixed size dataset $\mathcal{D} = \langle(\mathbf{x}_i, y_i)\rangle_{i=1}^{n}$, or Online $\langle(\mathbf{x}_i, y_i)\rangle_{i=1}^{\infty}$
Initialize: $\mathbf{w_1} = \mathbf{0} \in \mathbb{R}^D$
  for t = 1, 2, ... **do**
    Retrieve $\mathbf{x}_t$
    $\hat{y} = \mathsf{sign}(\mathbf{w}_t \cdot \mathbf{x}_t)$
    Retrieve y
    if $\hat{y} \neq y$ **then**
      $\mathbf{w}_{t+1} = \mathbf{w}_t + y_t \mathbf{x}_t$
    end if
  end for

A book titled "Perceptron" [54] published by Minsky & Papert shows that simple linear models such as the perceptron cannot perfectly classify data that is not linearly separable and famously the XOR function,

$$\mathsf{XOR}(x_1, x_2) = -\mathsf{sign}(x_1 \cdot x_2) \quad \text{where, } x_1, x_2 \in \{-1, +1\} \tag{2.2}$$

Fortunately by stacking layers of parallel perceptrons we can construct a parameterisation that precisely captures the XOR function, see [4, Ch. 6.1]. However, for arbitrary functions we will first need to introduce the artificial neuron. The artificial neuron generalises the perception by replacing the sign function with some arbitrary activation function g, also called the *transfer function*. Mathematically the artificial neuron computes the following function,

$$y = g\left(\sum_{i=1}^{D}(w_i \cdot x_i) + b\right) \tag{2.3}$$

where D is the dimension of the input feature vector $\mathbf{x}$. We can stack multiple artificial neurons in parallel over the same inputs. Given that each artificial neuron has its own set of learnable weights and biases, we may expect each neuron to compute different output values, see Figure 2.2.

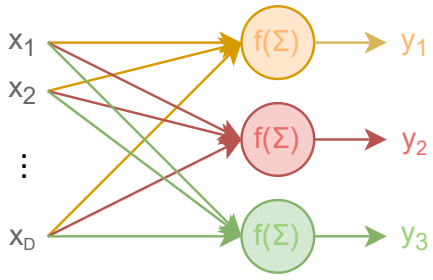

Figure 2.2: 3 stacked artificial neurons forming a layer of a neural network

When we stack artificial neurons in this way we form what is known as a *linear* or a *fully connected layer*. Importantly, the operation applied by a single linear layer can be conveniently expressed in terms of matrix and vector operations,

$$\mathbf{y} = g(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{2.4}$$

where,

- $\mathbf{y}$ denotes the output (column) vector with dimension N, where N corresponds to the number of stacked artificial neurons.

- g denotes the activation function typically applied element-wise to the pre-activation vector $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$.
- $\mathbf{W}$ denotes the $N \times D$ matrix of weights.
- $\mathbf{x}$ denotes the D-dimensional vector of inputs or activations from the previous layer.
- $\mathbf{b}$ denotes the D-dimensional vector of biases.

Linear layers of this form followed by non-linear activation functions are referred to as *hidden layers* in the literature, and the neurons that make up the hidden layers are called *hidden units*. Which activation function to use entirely depends on the task at hand. The *Sigmoid* and *Softmax* activation functions functions are typically used for classification tasks due to their convenient properties. Other activation functions include the, *rectified linear unit* (ReLU) function and the *hyperbolic tangent function* (*tanh*), which are commonly used for internal layers of the network. Table 2.1 lists some of the most common activation functions we expect to encounter and their mathematical definitions.

Activation Function	Equation	Properties
Identity	$g(x) = x$	This reduces a single linear layer to a linear regression model.
Sign	$g(x) = \mathsf{sign}(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ -1, & \text{if } x < 0 \end{cases}$	The perceptron model.
Binary	$g(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$	$\{0, 1\}$ classifier
Sigmoid	$g(x) = \frac{1}{1+e^{-x}}$	"Soft" differentiable version of the binary classifier.
Softmax	$g(x) = \frac{1+e^{x}}{\sum_{i=1}^{D} 1+e^{x_i}}$	Multiclass sigmoid applied at the layer level not element-wise.
ReLU	$g(x) = \mathsf{max}\{0, x\}$	Acts like the identity unless $x < 0$.
Tanh	$g(x) = \frac{2}{1+e^{-2x}} - 1$	Similar to Sigmoid but with range [-1, 1].

Table 2.1: Common activation functions

2.1.2 Multi-layer Perceptrons

Multi-layer perceptrons (MLPs) [4, 51] also called feed-forward neural networks are networks composed of more than one layer of artificial neurons. Layers of artificial neurons are organised in a sequential manner so that the output of one layer is fed as the input into the next layer. Figure 2.3 is provided as a visual example.

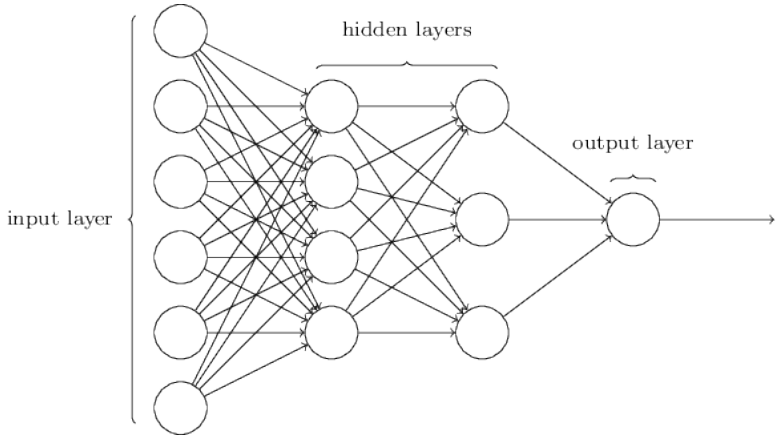

Figure 2.3: Multi-layer perceptron with 2 hidden layers

The network given in figure 2.3 can be viewed as a composition of functions $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$, where each of the functions $f^{(1)}, f^{(2)}, f^{(3)}$, represent the computation at each layer. We can think of each layer as computing a non-linear representation of the features of the previous layer; a good learning algorithm must decide how to best use and learn the internal representations at each hidden layer in order to best match the outputs to the target outputs [4, p. 164]. The *depth* of a network refers to the number of hidden layers used; this is essentially where the name deep learning comes from. The *width* of a network refers to the number of hidden units or artificial neurons in each layer of the network. For example, in figure 2.3 the network consists of 2 hidden layers, the first of which has 4 hidden units and the second has 3.

Input features (measurements, pixel value etc.) are fed to the input layer and subsequently processed by the hidden layers until the output value is computed. While the definition of an MLP is not necessarily very strict, we will assume that all the hidden layers of an MLP

are fully connected. With this configuration in mind, the full computation performed by an MLP can be expressed by the *forward equations*:

$$\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l \tag{2.5}$$

$$\mathbf{a}^l = g^l(\mathbf{z}^l) \tag{2.6}$$

where,

- $\mathbf{z}^l$ denotes the *pre-activations* of the l^{th} layer.
- g^l denotes the activation function used at the l^{th} layer.
- $\mathbf{W}^l$ is the weight matrix for the l^{th} layer.
- $\mathbf{b}^l$ is the vector of biases for the l^{th} layer.
- $\mathbf{a}^{l-1}$ and $\mathbf{a}^l$ denote the *activations* of the $l-1^{\text{th}}$ and l^{th} layer respectively.

By convention we set the first pre-activation and activation to the input vector:

$$\mathbf{z}^1 = \mathbf{a}^1 = \mathbf{x} \tag{2.7}$$

An MLP with L layers has $L-1$ pairs of weights and biases:

$$\boldsymbol{\theta} = \{\mathbf{W}^2, \mathbf{W}^3, ..., \mathbf{W}^L, \mathbf{b}^2, \mathbf{b}^3, ..., \mathbf{b}^L\} \tag{2.8}$$

By repeatedly applying the forward equations we can make a prediction $\hat{y}$ given some input feature vector $\mathbf{x}$. This is called the *forward pass*.

The key benefit of stacking hidden layers is that we can now capture non-linear dependencies and approximate non-linear functions. In other words, MLPs offer us with a much more expressive class of models than traditional linear models like the perceptron. An important result by Hornik *et al.* showed that MLPs can approximate any function on a closed interval to an arbitrary degree of accuracy when the number of hidden units in a

single layer is unconstrained [55]. This result is known as the *universal function approximation* result, which claims in theory, that MLPs are *universal function approximators*; they can approximate any target function f^*. Although in general, finding a parameterisation that closely approximates the function we care about, f^*, is not easy. In machine learning the problem of finding such a parameterisation is known as *training*.

2.1.3 Optimisation and Backpropagation

Training neural networks is typical done via gradient-based optimisation, a common technique used in many more traditional machine learning algorithms as well. In high school, when we are asked to find the minima of a function, our first instinct is to compute its derivative and set this to zero. Modern machine learning algorithms are based on exactly the same principle, although in practice for higher dimensional models solving the gradient equation is infeasible, and instead we rely on general purpose methods such as *gradient descent*. Gradient descent works by iteratively updating the parameters of our model $\boldsymbol{\theta}$ in the direction that minimises some differentiable objective function $J(\boldsymbol{\theta})$ as follows,

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1}) \tag{2.9}$$

where η is a (scalar) learning rate parameter, typically set to a small positive value that scales the gradient update steps. If the objective function is convex, then provided η is small enough, we are guaranteed to converge to the global minima, no matter where we start in the parameter space. Unfortunately, when it comes to training neural networks we are rarely afforded such guarantees. The problem is typically is non-convex [56]. This fact is easy to show, since there exist many parameterisations of the same MLP that compute the same function (simply permute the hidden units). Nevertheless, we still apply convex optimisation techniques similar to gradient descent for training neural networks, but several additional tricks are typically required.

For gradient descent to work we require the objective function $J(\boldsymbol{\theta})$ to be continuous and

differentiable with respect to the parameters $\boldsymbol{\theta}$. For MLPs this means any of the activation functions we use must be differentiable with respect to their inputs. In classification tasks our aim is to minimise the number of miss-classified examples of some given dataset $\mathcal{D} = \langle(\mathbf{x}_i, y_i)\rangle_{i=1}^{n}$. For most useful machine learning models including MLPs, finding the set of parameters that minimises the number of missclassified examples is intractable. Instead we specify a differentiable objective function called the *loss function*, that when minimised should improve the performance of our model on the task at hand.

The choice of loss function tends to be critical to the final performance of the network. The loss function needs to reflect the given task in such a way, that by minimising the loss function we expect perform well at the given task. In most cases we employ the principle of maximum likelihood: in a probabilistic sense MLPs typically model some conditional distribution $p(y\,|\,\mathbf{x}, \boldsymbol{\theta})$, and we wish to recover the parameters $\boldsymbol{\theta}$ that maximise the likelihood of the data under this distribution. This should in theory provide the best explanation of the data at hand and best capture the conditional distribution.

In this thesis we will use *deep Q-networks* (DQNs) [19, 20] which approximate the Q-values of state-action pairs; this task can be viewed as a form of regression. For regression tasks our output features typically approximate the mean of a Gaussian distribution. In this case, minimising the *mean-squared error* (MSE) is equivalent to maximising the likelihood of the data under the conditional distribution $p(y|\mathbf{x}, \theta)$ [4, p. 176]. So for DQNs we typically use the MSE loss function as the objective function we seek to minimise,

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{2.10}$$

In this thesis we also cover the cross-entropy method, a simple but effective method for solving relatively simple RL problems. The details are left for later, but effectively the cross-entropy method is a form of classification, where the underlying distribution from which the examples are drawn from changes slightly each batch iteration. Artificial neural networks (ANNs) used for the cross-entropy method implement the softmax function in

their linear output layer, see Table 2.1. This is because the softmax function is differentiable with respect to its inputs and it gives us a categorical distribution on the output features, which we can sample from. In this scenario minimising the cross entropy loss is equivalent to maximising the likelihood of the conditional distribution $p(y|\mathbf{x}, \theta)$ [4, p. 178]. The cross entropy loss for probability vectors $\mathbf{y}$ and $\hat{\mathbf{y}}$ is given below,

$$\mathcal{L}_{\text{cross entropy}} = -\frac{1}{n}\sum_{i=1}^{n} \mathbf{y} \cdot \log(\hat{\mathbf{y}}) \tag{2.11}$$

Now that we are equipped with some useful differentiable loss functions we can start to think about computing gradients. The backpropagation (BP) algorithm [12] provides us with a means for computing the gradients of the parameters of a neural network $\boldsymbol{\theta}$ with respect some loss function $\mathcal{L}$. Once we have computed these gradients we can begin to optimise the model by applying gradient based optimisation techniques, like gradient descent. Essentially, BP is a straightforward application of the chain rule of differentiation and it forms the basis for almost all gradient based learning algorithms for neural networks. Given some set of input-output targets or dataset $\mathcal{D}$, BP works in four stages stages:

1. **Forward pass:** feed the inputs $\mathbf{x}$ through the MLP, by applying the forward equations (Equations 2.5 and 2.6) to obtain the corresponding output value $\hat{y}$.

2. **Compute the loss:** under the criteria specified by the loss function $\mathcal{L}$, compute the scalar loss by comparing the outputs $\hat{y}$ computed in the forward pass to the output targets y provided in the dataset.

3. **Backward pass:** back-propagate the loss though the network by applying the chain rule of differentiation to compute the parameter gradients.

4. **Parameter update:** update the parameters of the network $\boldsymbol{\theta}$ in the direction that minimises the loss function $\mathcal{L}$ according to some optimisation procedure.

More formally, Algorithm 2 outlines the general procedure for a training on a single data point. We refer the reader to Appendix 2 for a formal derivation of the error back-

propagation procedure outlined by Algorithm 2. It should be clear that the forward pass and parameter update steps outlined in Algorithm 2 are immediately derived from the forward equations (Equations 2.5 and 2.6) and the gradient descent update step (Equation 2.9) respectively. We also note that the procedure outlined by Algorithm 2 can easily be extended to batch data by replacing the matrix multiplications with appropriate tensor operations.

Algorithm 2 Backpropagation

Input: inputs $\mathbf{x}$, targets $\mathbf{y}$, weights $\{\mathbf{W}^2, \mathbf{W}^3, ..., \mathbf{W}^L\}$ and biases $\{\mathbf{b}^2, \mathbf{b}^3, ..., \mathbf{b}^L\}$.
Output updated weights $\{\mathbf{W}^2, \mathbf{W}^3, ..., \mathbf{W}^L\}$ and biases $\{\mathbf{b}^2, \mathbf{b}^3, ..., \mathbf{b}^L\}$.

$\mathbf{a}^1, \mathbf{z}^1 \leftarrow \mathbf{x}^1$
for $l \leftarrow 2$; $l < L + 1$; $l \leftarrow l + 1$ **do** ▷ Forward pass
  $\mathbf{z}^l \leftarrow \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$
  $\mathbf{a}^l \leftarrow g^l(\mathbf{z}^l)$
end for
$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^L} \leftarrow \nabla_{\mathbf{z}^L} \mathcal{L}(\mathbf{y}, \mathbf{a}^\mathbf{L})$ ▷ Compute the loss
for $l \leftarrow L$; $l > 2$; $l \leftarrow l - 1$ **do** ▷ Backward pass
  $\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{l-1}} \leftarrow \frac{\partial \mathcal{L}}{\partial \mathbf{z}^l} \mathbf{W}^l \frac{\partial g^l}{\partial \mathbf{z}^l}$
end for
for $l \leftarrow 2$; $l < L + 1$; $l \leftarrow l + 1$ **do** ▷ Update the parameters
  $\Delta \mathbf{W}^l \leftarrow \eta \left[\mathbf{a}^{l-1} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^l} \right]^T$
  $\mathbf{W}^l \leftarrow \mathbf{W}^l + \Delta \mathbf{W}^l$
  $\Delta \mathbf{b}^l \leftarrow \eta \left[\frac{\partial \mathcal{L}}{\partial \mathbf{z}^l} \right]^T$
  $\mathbf{b}^l \leftarrow \mathbf{b}^l + \Delta \mathbf{b}^l$
end for

An important property of BP is that it can compute gradients locally, each layer only needs l to know about gradients of the next layer $\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{l+1}}$ and the activations at the previous layer $\mathbf{a}^{l-1}$. This property leads to efficient implementations of BP and provides some justification of its biological plausibility [51, p. 572], although this is a widely debated interpretation [29, 30].

Typically in modern deep learning we are given access to large datasets, and so computing gradients over the entire dataset $\mathcal{D}$ is very costly. Instead, we typically employ a batch gradient descent algorithm called *stochastic gradient descent* (SGD). With SGD, we sample a mini-batch of input-output pairs from $\mathcal{D}$ and compute the gradients using

this much smaller amount of data. Because of the randomness associated with sampling mini-batches, the parameter updates can become noisy which may cause divergence during training, to deal with this issue we can either increase the mini-batch size or adjust the learning rate parameter η accordingly.

In the deep learning literature there exists a vast array of mini-batch gradient optimisers, all built upon the core SGD algorithm. SGD is very sensitive to the learning rate parameter η; if η is set too big we will likely overshoot the minima and diverge, if η is set too small we will converge very slowly to the minima. Clever scheduling and annealing of the learning rate can help but this just gives us another thing to tune.

The *Adam* optimiser [43] is an optimisation scheme than has found a lot of success in RL and DL as a whole. The Adam optimiser is fairly sophisticated algorithm that is less sensitive to the user-specified learning rate η and is generally treated as a black box optimiser. Although, we must note that in some settings Adam still appears sensitive to the choice of learning rate, however it is still preferred over SGD since it converges much quicker in practice.

The Adam update procedure is based on two key ideas:

- Adapting the learning rate for each parameter based on the exponentially weighted moving average of its previous gradient norms.
- Using the exponentially weighted average of the gradients momentum for each parameter.

Algorithm 3 below outlines Adam update for a single parameter w.

While the Adam optimiser has 3 hyper-parameters η, β_1 and β_2, the betas are typically left as their default values $\beta_1 = 0.9$ and $\beta_2 = 0.999$. In some of our work we will use Adam, as it simply converges much quicker in many challenging RL scenarios, although we must note that Adam is optimised for backpropagation and not for alternative gradient based learning algorithms such as predictive coding. So in general we will try use SGD

Algorithm 3 Adam Optimiser

Input: η (learning rate), β_1, β_2, w_0 (parameter), $L(w)$ (loss function)
Initialize: $m_0 = 0$, $v_0 = 0$
  for t = 1, 2, ... **do**
    $g_t \leftarrow \nabla_w \mathcal{L}(w_{t-1})$
    $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
    $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
    $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$
    $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$
    $w_t \leftarrow w_{t-1} - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$
  end for

where possible in the interest of fairness and explainability. Furthermore, we note that the development of a more effective gradient based optimiser built on the predictive coding algorithm is an important area of orthogonal research.

2.1.4 Convolutional Neural Networks

We have already alluded to the idea that the hidden units of neural networks learn non-linear representations of the features of the previous layer; this is what makes MLPs more expressive than linear models. However, MLPs have their fair share of shortcomings in practice and commonly suffer from *overfitting* to the dataset. Overfitting in machine learning refers to the phenomena whereby a statistical model fits perfectly or very closely to the training data and achieves poor generalisation on unseen data. A common indicator of overfitting is when we observe excellent accuracy on the training dataset but poor accuracy on the test dataset. The idea of overfitting can be traced back to the famous *Occam's razor* principle, which states: when it comes to comparing theories or explanations, the simplest one, is preferred to the more complex one.

Convolutional neural networks (CNNs) [8] inspired by principles like Occam's razor and signal processing in the visual cortex have achieved much better generalisation capabilities than standard MLPs in tasks such as image classification, signal processing and representation learning. CNNs are well suited to 1d signals such as audio, 2d signals such as images, and even 3d signals such as MRI scans and other health data [51, p. 565]. CNNs use con-

volution filters with learnable weights to process input signals. For 2d signals this leads to *weight sharing* across the input image and as a result the hidden units of a convolutional layer have local 2-dimensional *receptive fields* [51, p. 565] and often *sparse activations*. The intuitive property of this spatial parameter tying is that learned convolution filters can detect the same feature or pattern anywhere in the input image, without having to independently learn the same weights as we would if we used a standard MLP [51, p. 565]. This leads to *translation invariance*, meaning we can detect a specific pattern even if it is translated in the image. Another added benefit of CNNs is that we can apply them to variable size input, whereas the input layer of an MLP must have a rigid structure.

Let's consider some 2d input signal I, for example a grey-scale image consisting of raw pixel data. The mathematical operation applied by a convolution filter K at a single point in the image (i, j) is typically denoted by an asterisk $*$ and is expressed as follows,

$$S(i,j) = (K * I)(i,j) = \sum_{m=1}^{W} \sum_{n=1}^{H} I\left(i + m - \left\lfloor \frac{W}{2} \right\rfloor, j + n - \left\lfloor \frac{H}{2} \right\rfloor\right) \cdot K(m,n) \tag{2.12}$$

where the convolution filter K has width W and height H. Typically (W, H) are equal and take values in $\{(3,3),(5,5),(7,7),...\}$. Figure 2.4 presents a visual interpretation of the convolution operation.

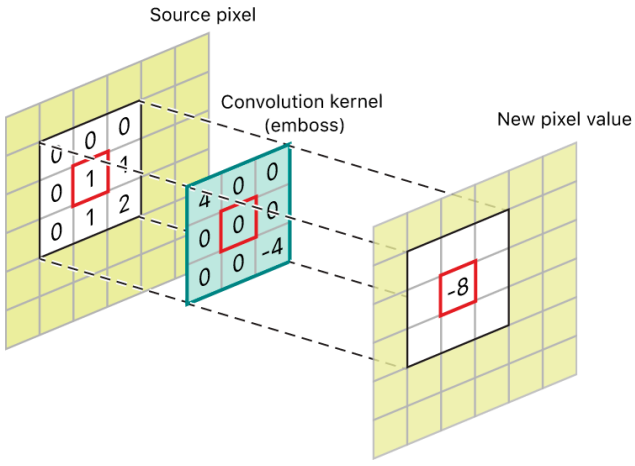

Figure 2.4: The convolution operation (Equation 2.12) applied by a convolution filter to a specific point in the input image

Typically, convolution filters are applied across the entire input in a sliding window fashion. If we want to apply the convolution filter K to every pixel in the image we can systematically shift the filter by one pixel (stride of 1) across the entire image and what we get back is called a *feature map*. In this scenario we will likely need to pad the image with zeros at its boundaries. We can also change the *stride* of the convolutional layer and skip every other pixel as opposed to applying it to every pixel, for more details about stride and zero-padding please refer to [57].

A convolutional layer in a neural network typically consists of many learnable convolutional filters applied in parallel, with each outputting their own feature map. It is also typical to use a *pooling layer* after a convolutional layer, since the same feature may have been picked up in two neighbouring pixels, and this is essentially redundant information. Pooling works by replacing each location in the output feature map with a summary statistic of nearby values [4, p. 330]; essentially we reduce the size of the feature maps by removing redundant or repeated feature detections in small local regions of the feature map, see figure 2.5.

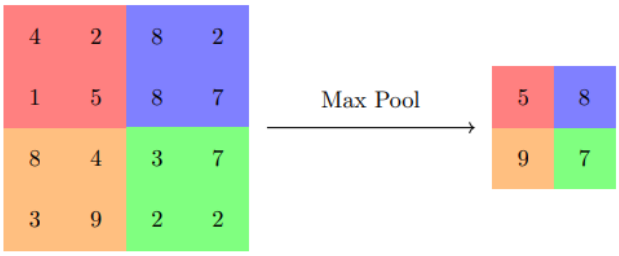

Figure 2.5: Max pooling operation. Pool size is 2x2 and stride is (2, 2).

Typically after several convolutional and pooling layers the output feature maps are flattened and passed through an MLP classifier or regressor. The feature extraction and processing performed by the convolutional and pooling layers is similar to internal representations learned in the hidden layers of a standard MLP, although exploitation of the local geometry of the input is what helps CNNs achieve better generalisation capabilities. Figure 2.6 illustrates the typical architecture of a CNN designed to classify images.

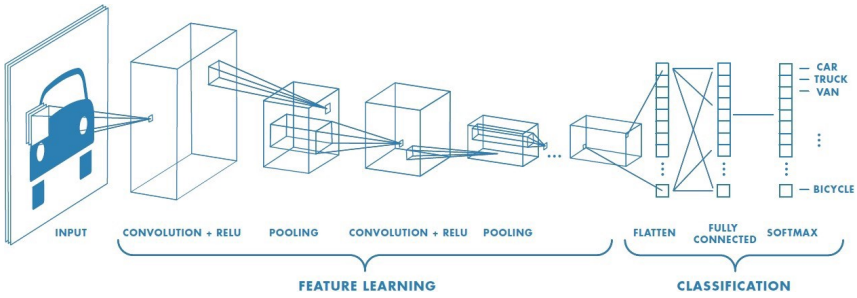

Figure 2.6: Typical architecture of a CNN images classifier.

Since the inception of CNNs, much of the recent research has become more of an engineering and mathematical effort than a neuroscience one. Important challenges that have been overcome include: successfully applying backpropagation to CNNs [1, 8], utilising deeper CNN architectures while avoiding the vanishing gradient problem [58], and adopting tricks such as dropout [13] and batch normalisation layers [59] to achieve better generalisation. Key successes from the past decade include AlexNet [1] for image classification and achieving super-human performance on Atari 2600 games by learning from raw pixel data [19, 20]. The later work will be used as the basis for much of the research in this thesis and its importance will become apparent.

2.2 Predictive Coding and Energy-Based Models

Predictive coding (PC) is a promising theory of learning in the brain first used as a means of explaining neural processing in the retina and visual cortex [33, 34]. More recently PC has evolved into a general purpose Hebbian learning algorithm that can be applied to networks of arbitrary topology [28, 40, 41]. Driven by local updates, PC has strong mathematical foundations and links to theories of Bayesian inference in the brain [35, 60]. In addition, PC posits as a more biologically plausible algorithm compared to BP and could be the key to achieving greater cognitive flexibility for systems aiming to capture human intelligence [42].

The idea behind PC is that brain function can be captured by inference and learning in some deep generative model. The architecture of such a model can be expressed in a

similar way to that of an artificial neural network (ANN), with layers of hidden neurons passing information from one layer to the next. The key distinction between ANNs and these deep generative models is the way in which synaptic connections are updated. In ANNs error computed at the output layer is back-propagated through the network before synaptic plasticity (parameter updates), this is the backpropagation (BP) algorithm, see Section 2.1.3. On the other hand, PC is a Hebbian learning algorithm relying only on local updates, it works by first updating the internal neural activity of the network by minimising some energy potential function, before updating the synaptic connections to minimise local prediction errors.

Despite PC's origins in neuroscience, there has developed a strong drive to understand its applications in deep learning and how networks trained with PC relate to deep neural networks trained with BP. Important work has been done in characterising PC under the energy-based model (EBM) framework [38, 61]. Insights into how PC and Hebbian learning in the brain can approximate BP have also lead to increased understanding of PC [39, 62].

Networks trained with PC have shown promise over BP in a number of recent studies [28, 41, 42]. In addition, the reliance on only local updates allows for greater flexibility, meaning PC can be implemented on specialised architectures and potentially run several orders of magnitude quicker than BP [42]. This could play a significant role in the adoption of PC in the near future as we aim to scale up large predictive and generative models. However, in this thesis we will be concerned with applying the theory of PC to *predictive coding networks* (PCNs) which emulate neural network architectures trained with local Hebbian updates. In this section, before we cover PCNs we will first briefly cover Hopfield networks. We hope this material serves as a useful introduction into the origin of EMBs and their mathematical interpretations.

2.2.1 Hopfield Networks

Hopfield networks [63] are fully connected Ising models with a symmetric weight matrix $\mathbf{W} = \mathbf{W}^T \in \{-1, +1\}^{D \times D}$ [51, p. 669] and biases $\mathbf{b} \in \{-1, +1\}^D$, see Figure 2.7. Ising models themselves are *energy-based models* (EBMs) originally developed in statistical physics for modelling the behaviour of magnets. EMBs are generative models, which means they aim to capture the underlying distribution of a given dataset. In general, EMBs work by associating an unnormalised log probability scalar (energy) value to every configuration of the model. To capture dependencies in the dataset EMBs learn to associate low energy states with correct values for the latent variables and high energy states with incorrect values for the latent variables.

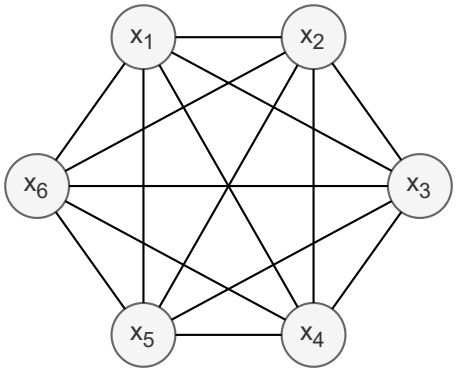

Figure 2.7: Hopfield network with 6 nodes represented as an undirected graph. Each edge represents a symmetric weight $w_{j,i} = w_{i,j}$ denoting the dependency between nodes.

Hopfield networks themselves act as content-addressable memory systems and are used for the problem of *associative memories*; given some set of observed bit vectors we want to "memorise" these patterns, so that when we are presented with a corrupted or partial pattern we can recover the corresponding pattern.

The operations of Hopfield networks and EMBs in general can be split into two phases: *learning* and *inference*. In words, learning corresponds to the process by which we pick the parameters of our model, and inference corresponds to the process of minimising some energy function to recover or infer the values of observed and/or latent variables. The energy function for Hopfield networks is derived from the unnormalised log probability

of the fully connected Ising model, see [51, p. 668] for details. The energy function for Hopfield networks is given below,

$$E(\mathbf{s};\boldsymbol{\theta}) = -\frac{1}{2}\sum_{i,j:i\neq j} w_{i,j}\cdot s_i \cdot s_j + \sum_{i=1}^{N} b_i \cdot s_i = -\frac{1}{2}\mathbf{s}^T\mathbf{W}\mathbf{s} + \mathbf{b}^T\mathbf{s} \tag{2.13}$$

where $\mathbf{s} \in \{-1, +1\}^D$ denotes the values of the nodes (see Figure 2.7) or the state of the Hopfield network. During inference we minimise the energy function by coordinate descent using *iterative conditional modes* (ICM), a deterministic algorithm which iteratively sets each node to its most likely state given its neighbours. This corresponds to following the update rules of the dynamical system defined by the network,

$$s_i \leftarrow \begin{cases} 1 & \text{if } \sum_{j=1}^{N} w_{i,j}s_j > b_i \\ -1 & \text{if } \sum_{j=1}^{N} w_{i,j}s_j < b_i \end{cases} \tag{2.14}$$

By minimising the energy function we expect the dynamical system to converge to a low-energy state. The general idea behind training Hopfield networks is that we lower the energy of the states we wish to "memorise", making them local minima of the energy function, so that when we are provided with a corrupted version of the "memorised" state we can recover it by running inference.

There are several methods for fixing the parameters of a Hopfield network to memorise a given bit vector dataset, for example, Hebb's law of association [64], Storkey's rule [65], and via maximum likelihood and gradient descent [51, Ch. 19.5.1]. The important thing to note is that all of these methods have the same two desirable properties:

- **Local:** updates only rely on local information immediately available from neurons either side of the weight.
- **Incremental:** it can learn new patterns without explicit access to information about older learned patterns.

These two key properties result in a *Hebbian* learning rule [64] making it more biologically

plausible, since the synaptic strengths between neurons in the brain are incrementally updated based on local information. Although, for some update rules like maximum likelihood, inference needs to be run before every parameter update, which makes training these undirected models much slower than directed models like ANNs [51, Ch. 19.5].

Several extensions to the classic binary Hopfield networks exists, these include Boltzmann machines that generalise Hopfield networks with hidden units. In addition, *Modern Hopfield Networks* (MHNs) also called *Dense Associative Memories* (DAMs) [66] generalise Hopfield networks. They break the linear relationship between the number of units and the number of stored "memories" by introducing strong non-linearities into the energy function. MHNs have also been extended to continuous states so that they can be applied to more generic tasks [67].

Deep Boltzmann machines [68] were also among the first deep architectures to be successfully trained for modern machine learning tasks, like image classification, by a famous method known as contrastive divergence [48]. This material is beyond the scope of this thesis but we refer the interested reader to the following resources: [51, Ch. 27.7], [4, Ch. 20].

2.2.2 Predictive Coding Networks

Predictive coding networks (PCNs) [40–42] apply the general theory of predictive coding (PC) to arbitrary graph topologies. Mathematically PCNs can be viewed as variational inference on hierarchical Gaussian generative models [42]. On the other hand, artificial neural networks (ANNs) (Section 2.1) are discriminative models that model conditional distributions of the form $p(y \mid \mathbf{x}, \boldsymbol{\theta})$. Conveniently, we can express hierarchical PCNs and ANNs in the same way: in terms of layers of neurons processing information from input to output. This establishes a framework for the direct comparison of ANNs and PCNs and by extension the two learning algorithms BP and PC. In fact under certain conditions it can be shown that PCNs actually exactly implement backpropagation (BP) and thus generalise ANNs [69].

The key distinction between ANNs and PCNs is the learning rule and objective function. With ANNs the goal is to update the parameters of the network to minimise the scalar loss computed at the output layer. With PCNs the goal is to minimise the local *prediction errors* or local energy functions between layers. Many resources describe PCNs in slightly different ways [28, 40–42, 61], however we will aim to give a description that is consistent with the notation used in Section 2.1.

Let's first note the following conventions we plan on using:

- Let f be the network of L fully connected layers with parameters:

$$\boldsymbol{\theta} = \{\mathbf{W}^2, \mathbf{W}^3, ..., \mathbf{W}^L, \mathbf{b}^2, \mathbf{b}^3, ..., \mathbf{b}^L\} \tag{2.15}$$

- We assume every layer (including the input and output layer) has dimension n.
- We will rewrite the forward equations (Equations 2.6 and 2.5 from Section 2.1.2) of the network f in terms of $\mathbf{x}^l$:

$$\mathbf{x}^l \leftarrow \mathbf{W}^l g^l(\mathbf{x}^{l-1}) + \mathbf{b}^l \tag{2.16}$$

- We assume that $\mathbf{x}^1$ is the input layer and $\mathbf{x}^L$ is the output layer.

In general PCNs incorporate two types of time dependent neurons: value nodes denoted by $x^l_{t,i}$ and error nodes denoted by $\mathcal{E}^l_{t,i}$. The value nodes $x^l_{t,i}$ correspond to the activity of a single neuron; their aim is to predict the signal $\mu^l_{t,i}$ which is the weighted sum of the activations of the neurons feeding into them, see Figure 2.8. If $x^l_{t,i}$ is the i^{th} neuron of the l^{th} fully connected layer in f, then $\mu^l_{t,i}$ is computed as follows,

$$\mu^l_{t,i} = \sum_{j=1}^{n} \left(w^l_{i,j} g^l(x^{l-1}_{t,j}) \right) + b^l_i \tag{2.17}$$

where $w^l_{i,j}$ is the weight connecting the i^{th} neuron of layer l to the j^{th} neuron of layer

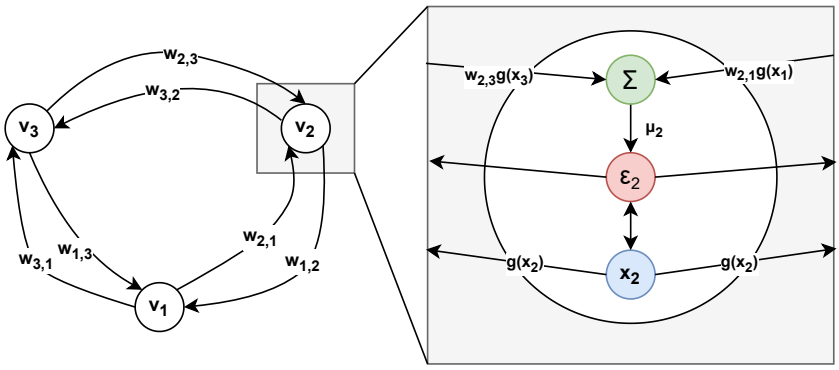

Figure 2.8: Fully connected predictive coding network with 3 nodes. The weighted directed edges represent connections between nodes. The biases have been removed for simplicity.

$l-1$, g^l is some arbitrary activation function and b_i^l is the bias for $x_{t,i}^l$. The prediction error $\mathcal{E}_{t,i}^l$, refers to the mismatch between $x_{t,i}^l$ and $\mu_{t,i}^l$, that is, $\mathcal{E}_{t,i}^l = x_{t,i}^l - \mu_{t,i}^l$. Writing this in vector notation at the layer level gives us the following,

$$\boldsymbol{\mu}_t^l \leftarrow \mathbf{W}^l g^l(\mathbf{x}_t^{l-1}) + \mathbf{b}^l \tag{2.18}$$

$$\boldsymbol{\mathcal{E}}_t^l \leftarrow \mathbf{x}_t^l - \boldsymbol{\mu}_t^l \tag{2.19}$$

During both learning and inference the goal is to minimise the sum of squared prediction errors at each layer, which can equivalently be interpreted as minimising the following global energy function,

$$E(\mathbf{x}^1, ..., \mathbf{x}^L; \boldsymbol{\theta}) = \frac{1}{2}\sum_{l=2}^{L}(\mathbf{x}^l - \mathbf{W}^l g^l(\mathbf{x}^{l-1}) - \mathbf{b}^l)^2 = \frac{1}{2}\sum_{l=2}^{L}\left(\boldsymbol{\mathcal{E}}^l\right)^2 \tag{2.20}$$

Inference: Typically during inference the input and output nodes are clamped, the parameters of the network are fixed, and only the value nodes of internal layers (layers $2, ..., L-1$) of the network are optimised to minimise the prediction errors. Inference is typically run for T steps or until some convergence criteria is met. Typically standard gradient descent with some form of learning rate annealing is used to modify the value nodes in order to minimise the energy function during inference.

Learning: consider the supervised learning setting, where we are given some input stimulus $\mathbf{s}^{\mathbf{in}}$ and output target $\mathbf{s}^{\mathbf{target}}$ that we wish to learn. The internal layers are initialised to $\mathbf{0}$ and the input and output layers are clamped as follows: $\mathbf{x}^1 \leftarrow \mathbf{s}^{\mathbf{in}}$, $\mathbf{x}^L \leftarrow \mathbf{s}^{\mathbf{target}}$. Inference is run for T steps to minimise the energy function. After inference, the weights and biases are updated using a gradient based optimiser to minimise the prediction errors (or equivalently the energy function).

Prediction: in the discriminative setting, predictions with PCNs are made in exactly the same way as with ANNs, by applying the forward equations. By re-framing the forward equations from Section 2.1.2 in terms of value nodes, we derive a simple procedure for making predictions with either an ANN or PCN. Algorithm 4 presents this procedure.

Algorithm 4 Predict with ANN or PCN

Input: inputs $\mathbf{s}^{\mathbf{in}}$, weights $\{\mathbf{W}^2, \mathbf{W}^3, ..., \mathbf{W}^L\}$ and biases $\{\mathbf{b}^2, \mathbf{b}^3, ..., \mathbf{b}^L\}$.
Output: output prediction $\mathbf{s}^{\mathbf{out}}$.

$\mathbf{x}^1 \leftarrow \mathbf{s}^{\mathbf{in}}$ ▷ Clamp input neurons
for $l \leftarrow 1$; $l < L$; $l \leftarrow l + 1$ **do** ▷ Forward pass
  $\mathbf{x}^{l+1} \leftarrow \mathbf{W}^{l+1} g^l(\mathbf{x}^l) + \mathbf{b}^{l+1}$
end for
return $\mathbf{s}^{\mathbf{out}} \leftarrow \mathbf{x}^L$

Algorithm 5 presents the full predictive coding algorithm applied to f, the fully connected network with L layers. For a detailed derivation of the gradient updates please refer to Appendices A.2 and A.3, or refer to the original derivations by Whittington & Bogacz [39].

PCNs set up in this way effectively re-frame learning on ANN architectures as an inference problem [42], where each layer aims to predict the weighted activations of the previous layer. Since we use squared errors, the uncertainty in the network and at the output layer is assumed to be Gaussian, which lends itself immediately to regression tasks. The Hebbian nature of this algorithm should become apparent after studying the weight updates in Algorithm 5; updates to value nodes and parameters are made using only local errors and activations from neighbouring layers.

Algorithm 5 Predictive Coding

Input: inputs $\mathbf{s}^{\mathbf{in}}$, targets $\mathbf{s}^{\mathbf{target}}$, weights $\{\mathbf{W}^2, \mathbf{W}^3, ..., \mathbf{W}^L\}$ and biases $\{\mathbf{b}^2, \mathbf{b}^3, ..., \mathbf{b}^L\}$
Output updated weights $\{\mathbf{W}^2, \mathbf{W}^3, ..., \mathbf{W}^L\}$ and biases $\{\mathbf{b}^2, \mathbf{b}^3, ..., \mathbf{b}^L\}$

$\mathbf{x}^1 \leftarrow \mathbf{s}^{\mathbf{in}}$ ▷ Clamp input neurons
$\mathbf{x}^L \leftarrow \mathbf{s}^{\mathbf{target}}$ ▷ Clamp output neurons
for $l \leftarrow 2 \; ; \; l < L \; ; \; l \leftarrow l + 1$ **do** ▷ Initialization
    $\mathbf{x}^l \leftarrow \mathbf{0}$
end for
for $t = 0 \; ; \; t < T \; ; \; t \leftarrow t + 1$ **do** ▷ Inference
    for $l \leftarrow 1 \; ; \; l < L \; ; \; l \leftarrow l + 1$ **do**
        $\boldsymbol{\mu}^{l+1} \leftarrow \mathbf{W}^{l+1} g^l(\mathbf{x}^l) + \mathbf{b}^{l+1}$
        $\boldsymbol{\mathcal{E}}^{l+1} \leftarrow \mathbf{x}^{l+1} - \boldsymbol{\mu}^{l+1}$
    end for
    for $l \leftarrow 2 \; ; \; l < L \; ; \; l \leftarrow l + 1$ **do**
        $\Delta \mathbf{x}^l \leftarrow \alpha \left(-\boldsymbol{\mathcal{E}}^l + \frac{\partial g^l}{\partial \mathbf{x}^l} \left(\left(\mathbf{W}^{l+1} \right)^T \boldsymbol{\mathcal{E}}^{l+1} \right) \right)$
        $\mathbf{x}^l \leftarrow \mathbf{x}^l + \Delta \mathbf{x}^l$ ▷ Inference step
    end for
end for
for $l \leftarrow 2 \; ; \; l < L + 1 \; ; \; l \leftarrow l + 1$ **do** ▷ Parameter updates
    $\Delta \mathbf{W}^l \leftarrow \eta \boldsymbol{\mathcal{E}}^l \left[g^l(\mathbf{x}^{l-1}) \right]^T$
    $\mathbf{W}^l \leftarrow \mathbf{W}^l + \Delta \mathbf{W}^l$
    $\Delta \mathbf{b}^l \leftarrow \eta \boldsymbol{\mathcal{E}}^l$
    $\mathbf{b}^l \leftarrow \mathbf{b}^l + \Delta \mathbf{b}^l$
end for

Some of the first results showed that small PCNs trained on MNIST were able to achieve comparable performance to ANNs of the same size trained with BP [39]. Similar results were obtained by using a variant of PC that allows for layer-specific loss functions [70], rather than the default squared error loss. This variant achieved similar performance to BP on MNIST and FashionMNIST. Furthermore, deep convolutional PCNs were also able to achieve only slightly worse performance than BP on more challenging datasets such as CIFAR10 and ImageNet [71].

2.2.3 Generative Predictive Coding Networks

Generative PCNs have subtle but important differences compared to those used in the discriminative setting, although the way in which they are trained is almost entirely the same. Rather than trying to emulate ANN architectures generative PCNs are setup in a way that is very reminiscent of classical Hopfield networks, restricted Boltzmann machine (RBMs) and deep Boltzmann machines (DBMs), see figure 2.9.

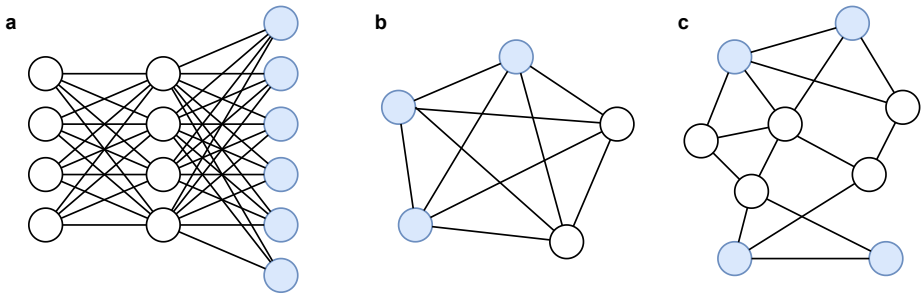

Figure 2.9: Generative PCN architectures, sensory vertices are blue and internal vertices are grey. **a:** generative PCN with two fully connected hidden layers, reminiscent of stacked RBMs or a DBM. **b:** fully connected generative PCN with 5 vertices reminiscent of the standard Boltzmann machine. **c:** chaotic generative PCN shown to illustrate that PCNs can have an arbitrary topology.

The vertices of generative PCNs are classified into two distinct classes: sensory and internal vertices. As before, both types of vertex implement two types of time-dependent neurons: value nodes $x_{t,i}$ and error nodes $\mathcal{E}_{t,i}$. Again, the goal of the value nodes $x_{t,i}$ is to predict the signal $\mu_{t,i}$, which in an arbitrary graph topology can be expressed by,

$$\mu_{t,i} = \sum_j w_{i,j} g_j(x_{t,j}) \tag{2.21}$$

where the sum is over all vertices j that feed into i with a directed edge, each edge may have its own activation function g_j. During inference the local errors $\mathcal{E}_{t,i} = \mu_{t,i} - x_{t,i}$ are computed, and the global energy function, which corresponds to the sum of squared errors, is minimised until convergence using gradient descent as before. We can write the

global energy function as,

$$E_t = \frac{1}{2}\sum_i (\mathcal{E}_{i,t})^2 \tag{2.22}$$

where the sum is over all vertices i. The goal of generative models is to learn the underlying distribution of some dataset so that we can use them in a more flexible way than discriminative models. Learning in generative PCNs occurs in a similar way to learning in discriminative PCNs, although we usually don't have access to target outputs or labels.

Learning: suppose we are presented with a stimulus $\mathbf{s} \in \mathbb{R}^d$ we wish to learn. The sensory vertices of the PCN are clamped to the stimulus $\mathbf{s}$, the weights of the model are fixed, and the value nodes of the internal vertices are optimised with gradient descent to minimise the prediction errors. After T inference steps, the weights of the model are updated using a gradient based optimiser to minimise the prediction errors. Algorithm 6 outlines the procedure at a high level. Please refer to [40] for a detailed outline of the gradient computations during the update steps in Algorithm 6.

Algorithm 6 Learn on generative PCN

Input: stimulus $\mathbf{s}$, weights $\mathbf{W}$
Output: updated weights $\mathbf{W}$

```
x_1, ...x_d ← s_1, ..., s_d                    ▷ Clamp sensory neurons
for t ← 0 ; t < T ; t ← t + 1 do
    for each vertex i ∉ {1, ..., d} do
        update x_i to minimise E_t              ▷ Inference
    end for
end for
update every w_{i,j} to minimise E_T            ▷ weight update
```

Querying: once trained, generative PCNs can be queried in a flexible way. Either by conditioning on some partial clamping of the sensory neurons, or by randomly initialising the sensory neurons and running inference to generate a novel stimulus. As such, generative PCNs can be applied to a variety of machine learning tasks such as: image generation, image completion, image denoising, image classification and associative memories [40, 41].

Salvatori *et al.* showed that PCNs achieved superior performance in high-dimensional associative memory tasks when compared to state-of-the art models [41]. Specifically, PCNs achieve high image retrieval and reconstruction accuracy when compared to auto-encoders (AEs) of the same size; overparameterisation is essential for AEs, whereas PCNs tend to do well even with small architectures. Salvatori *et al.* also showed that deep generative PCNs out perform modern Hopfield networks (MHNs) and deep associative neural networks (DANNs) [72] in image retrieval and reconstruction tasks. Furthermore, associative memory models trained with PC possess a high degree of biological plausibility since it has been hypothesised that the brain stores and retrieves memories using similar mechanisms [73].

In another study Salvatori *et al.* showed that PC can be applied to arbitrary graph topologies and still maintain good performance [40], a feat which Boltzmann machines fail to accomplish. It is important to note that learning on arbitrary network topologies is not possible with BP since we would likely run into infinite loops while trying to back-propagate error. This once again speaks to the biologically plausibility of PC, and in fact in the same study Salvatori *et al.* showed that PC is able to perform image generation and classification on extremely dense networks that resemble regions of the brain [40], networks that look vastly different to typical ANN architectures.

One of the most important properties of PCNs is their ability to generalise to novel tasks. ANNs for example, typically need to be re-trained when applied to a new task, whereas PCNs appear to learn an internal representation of the dataset and generalise well to tasks they were not specifically trained for [42]. This property comes from the inference phase which solves a dynamical system by minimising the internal energy of the network. While inference appears to unlock the potential for cognitive flexibility in PCNs, it also happens to be one of its current major pitfalls. Running inference is slow, particularly for large T, and so in practice we can't set T too big. This of course affects inference and subsequently makes learning less stable, although if we are careful, we can make the most of PCNs in a number of settings.

2.3 Reinforcement Learning

Reinforcement learning (RL) [14] is a paradigm of machine learning lying somewhere in the spectrum between supervised and unsupervised learning. RL has many interesting connections to control theory, optimization, psychology, neuroscience and animal behaviour and learning. This makes it a topic if high interest in biological learning and subsequently this thesis. We will start this section by introducing the general formalisms of the RL paradigm, before continuing on a small adventure that takes us through some of the most fundamental methods.

Put simply RL can be characterised as learning what to do in a given environment so as to maximise the accumulation of some scalar *reward*, often denoted by r_t. The learner, often referred to as the *agent*, interacts with the environment by picking actions and must implicitly learn which actions lead to the most reward. Along with reward signals r_t the agent receives an *observation* o_t at each time step, which is often a partial or full description of the of the underlying *state* s_t of the environment. Figure 2.10 illustrates this information flow at each timestep.

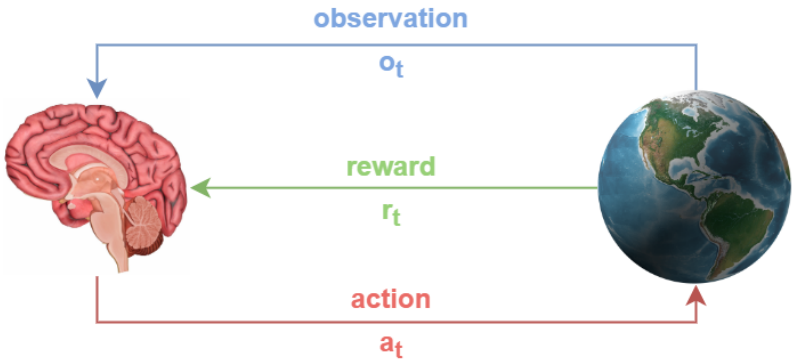

Figure 2.10: An agent interacting with its environment by picking an action a_t, receiving reward signal r_t and observation o_t.

The differences between this setup and more traditional paradigms like supervised and unsupervised learning should now become apparent. While in RL it is common to train on labelled experience (or examples), the experience itself does not necessarily encode

the best actions to take. This is different of course to supervised learning, where the labelled examples encode exactly the desired behaviour of the system. On the other hand, with unsupervised learning the goal is to uncover the underlying structure of a given dataset, and while uncovering the structure of the environment would certainly benefit the agent, the main goal is still to maximise accumulated reward and learn which actions are best to take in which scenarios, and in most cases having a perfect representation of the environment doesn't trivialise this problem.

In some of the most interesting and challenging cases, actions may not affect the most immediate reward, but rather affect the state of the environment such that there is potential for some delayed reward signal in the future. Environments like this, with sparse reward signals, cause some of the best algorithms out there to fail; agents must learn to manipulate their environment and search, sometimes in a trial-and-error type fashion, for reward signals they can learn from. This is closely related to the famous trade-off between exploration and exploitation. Using its experience the agent must exploit what is has learned to ensure it maximises reward, but how can the agent hope to find better actions in the future without trying them, and exploring the environment?

To characterise RL formally, we draw from ideas about dynamical systems modelling and optimal control. We formalise the RL problem as optimal control of an unknown *Markov decision process* (MDP). We will continue by introducing the formal definition of the MDP model, along with important notions, such as, the *policy* and *future return*.

2.3.1 Markov Decision Process

Throughout this entire section we will consider the case where we have full observability, $o_t = s_t$, or in words, the observations we receive from the environment are exactly the current state of the environment. An MDP is essentially an extension of a Markov process augmented with rewards, and where state transitions are conditioned on actions. Formally an MDP it is a 4-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, P \rangle$ where:

- $\mathcal{S}$ is the set of states or the state space.
- $\mathcal{A}$ is the set of actions or the action space.
- $\mathcal{R}$ is the set of all possible rewards, $\mathcal{R} \subseteq \mathbb{R}$
- P denotes the transition dynamics of the MDP. Specifically, $P(s', r, s, a) = \Pr(s_{t+1} = s', r_t = r \,|\, s_t = s, a_t = a)$ is the probability of transitioning from s to s' and receiving reward r by picking action a.

Additionally,

- s and s' denote states, a denotes an action and r denotes a scalar reward.
- S_t is the random variable denoting the state at time t, A_t is the random variable denoting the action taken at time t, R_t is the random variable denoting the reward received at time t.

Note that the transition matrix satisfies the Markov property, it only depends on the current state and action, not the entire history of states. This may seem like an obvious oversimplification and it is, but in practice there are ways of capturing longer temporal dependencies if we need. Figure 2.11 gives an example of a generic Markov process augmented with rewards and actions.

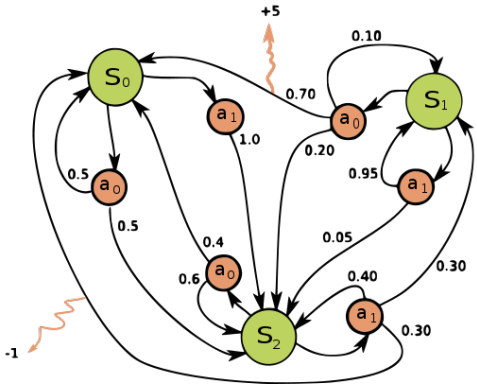

Figure 2.11: A simple MDP with three states (green circles), two actions (orange circles) and reward signals denoted by orange arrows.

In the discrete time model, actions are taken at discrete timesteps $\{0, ..., T\}$. MDPs can be extended to a continuous-time scenario where actions are taken at arbitrary time points, but this is beyond our needs. Instead we will assume the following: at each timestep the agent observes state s_t, picks an action a_t according to its current policy π and receives reward r_{t+1}. When T is finite the MDP is said to have *finite horizon*. Additionally, an MDP is said to be *finite* when both the set of states $\mathcal{S}$ and set of actions $\mathcal{A}$ are finite sets [74, Ch. 14.2].

We note that our definition for MDPs is quite general and assumes the reward received at r_{t+1} is a random variable given s_t, and a_t, but most of the time it is simply a deterministic function $r(s, a)$ depending on s_t, and a_t. Either way, we let the sequence of random variables $R_{t+1}, R_{t+2}, R_{t+3}, ...$ denote the reward at discrete timesteps after t. In the finite horizon case we can write the *accumulated reward* or *return* as,

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{2.23}$$

MDPs with finite horizon typically have a notion of terminal state, where agent-environment interaction ends. These environments are called *episodic*. For episodic tasks, the goal of maximising accumulated reward G_t coincides with picking actions that result longer episodes. Informally, this could be seen as trying to keep your character "alive" in a video game. For MDPs with infinite horizon we introduce a discount factor $\gamma \in [0, 1]$, which favours immediate rewards over rewards far in the future. We now define the return G_t as,

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.24}$$

This way of writing G_t is called the *discounted accumulated reward* or *future return* and it is perhaps the most important quantity in RL. Even in episodic tasks we usually write G_t in this way, for both convenience and practical reasons.

Perhaps one of the most key notions in RL is the idea of a *policy*. The policy, typically denoted by $\pi(a \mid s)$, refers to the way in which the agent picks actions given some state (or observation). Formally the policy is defined as a probability distribution over actions for every possible state,

$$\pi(a \mid s) = \Pr(a_t = a \mid s_t = s)$$

The policy can be expressed in many ways: as a table of probabilities over all state-action pairs $(s, a) \in \mathcal{S} \times \mathcal{A}$, as a linear model or even as a deep neural network. The policy can be and is often deterministic. All we need to understand at this point is that the policy defines how the agents acts in the environment. Since the policy π gives us a (probabilistic) mapping from $\mathcal{S}$ to $\mathcal{A}$ we can in a sense "combine" the policy with the MDP. This reduces the MDP to a Markov process augmented with rewards, or more succinctly a *Markov reward process*. This idea is called *policy evaluation*, which is an important procedure for comparing policies.

The goal of the agent is to determine which policy results in behaviour that maximises the *expected future return*, $\mathbb{E}_\pi[G_t]$. In the next section we will introduce the idea of the optimal policy π^*, which maximises this quantity over all possible states $s \in \mathcal{S}$. We will also cover *value functions* and their relationships to the optimal policy.

2.3.2 Value Functions and the Optimal Policy

Before formally describing the optimal policy π^* we must introduce the notion of a *value function*. The value of a state $s \in \mathcal{S}$ corresponds to how desirable it is to be in that state with respect to the expected future return. Under a fixed policy π the value of a state is given by the state-value function $v_\pi(s)$, which is defined as,

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| \, S_t = s \right] \tag{2.25}$$

where the randomness here is over all state transitions, reward dynamics and action selections if the policy π is stochastic. Another important value function is the *state-action*

value function or *q-function* $q_\pi(s,a)$. $q_\pi(s,a)$ is defined for all pairs $(s,a) \in \mathcal{S} \times \mathcal{A}$ as the expected future return by taking action a from state s and following the fixed policy π thereafter,

$$q_\pi(s,a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right] \tag{2.26}$$

Recall that the goal in RL is to determine the policy π^* that maximises the expected future return for all states $s \in \mathcal{S}$. We are now equipped to give the formal definition of the optimal policy π^* and its properties. A policy π' is said to be better than or equal to a policy π if for all states $v_{\pi'}(s) \geq v_\pi(s)$. The optimal policy π^* is such that $v_{\pi^*}(s) \geq v_\pi(s)$ for all states $s \in \mathcal{S}$ and all policies π. We note that in some cases there may be many optimal policies, although they all share the same state-value function, called the *optimal state-value function*,

$$v_*(s) = \max_\pi v_\pi(s) \qquad \text{for all } s \in \mathcal{S} \tag{2.27}$$

Similarly, the optimal policies all share the same *optimal state-action value function*,

$$q_*(s,a) = \max_\pi q_\pi(s,a) \qquad \text{for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A} \tag{2.28}$$

How we determine the state-value function $v_\pi(s)$ or state-action value function $q_\pi(s,a)$ for some policy π is called *policy evaluation*. Recall the definition of the discounted accumulated reward or future return,

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.29}$$

Observe that we can write G_t by the following recursion,

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + ...) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.30}$$

Using the recursion established by Equation 2.30, we can rewrite the value function as,

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right] \tag{2.31}$$

$$= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \mid S_t = s \right]$$

$$= \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s \right] \tag{2.32}$$

If the MDP is finite we can sum over all randomness, giving us,

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s',r} \Pr(s', r \mid s, a) \left[r + \gamma v_\pi(s') \right] \tag{2.33}$$

where we implicitly assume actions a are taken from the set $\mathcal{A}$, next states s' are taken from the set $\mathcal{S}$ and rewards r are taken from the set of all rewards $\mathcal{R}$. Equation 2.33 is called the *Bellman equation* for the state-value function $v_\pi(s)$, it defines the relationship between the value of a state and its successor states under some fixed policy π.

Similarly, we can write the Bellman equation for $q_\pi(s, a)$,

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi \left[\gamma G_t \mid S_t = s, A_t = a \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a \right] \\ &= \sum_{s',r} \Pr(s', r \mid s, a) \left(r + \gamma \sum_{a'} \pi(a' \mid s') q_\pi(s', a') \right) \end{aligned} \tag{2.34}$$

Now, by applying Equations 2.33 and 2.34 we can write out the *Bellman optimality equations* for both $v_*(s)$ and $q_*(s,a)$,

$$v_*(s) = \max_a \sum_{s',r} \Pr(s', r \mid s, a) \left[r + \gamma v_*(s') \right] \tag{2.35}$$

$$q_*(s,a) = \sum_{s',r} \Pr(s', r \mid s, a) \left(r + \gamma \max_{a'} q_*(s', a') \right) \tag{2.36}$$

In the case of finite MDP with no loops, the Bellman optimality equation for $v_*(s)$ and $q_*(s,a)$ have a unique solution that can be obtained by solving a system of linear equations. For less straightforward MDPs we must rely on alternative methods.

2.3.3 Dynamic Programming Methods

Dynamic programming broadly refers to the technique whereby a complicated problem with a recursive structure is broken into many sub-problems, the problems are then subsequently solved and the result stored in memory for re-use. Dynamic programming methods for RL utilise the recursive nature of the Bellman equations and apply dynamic programming principles to determine or closely approximate the optimal policy π^*. We will look at two methods in this section, namely, *policy iteration* and *value iteration*. Both these methods actually solve the problem of optimal control in a known MDP. However, unfortunately we can only use them under some quite strict assumptions that rarely hold in practice.

Value iteration: [14, Ch. 4.4] is an iterative dynamic programming method for obtaining the optimal policy π^* of a given MDP by closely approximating the optimal state-value function $v_*(s)$. Once we have obtained the optimal state-value function $v_*(s)$ an optimal policy can easily be obtained by,

$$\pi^*(a \mid s) = \begin{cases} 1 & \text{if } a = \text{argmax}_a \sum_a \Pr(s', r \mid s, a)(r + \gamma v_*(s)) \\ 0 & \text{otherwise} \end{cases} \tag{2.37}$$

In fact if the optimal state-action value function $q_*(s, a)$ is known, we can write this more succinctly:

$$\pi^*(a \mid s) = \begin{cases} 1 & \text{if } a = \operatorname{argmax}_a q_*(s, a) \\ 0 & \text{otherwise} \end{cases} \tag{2.38}$$

Value iteration works by constructing a table or list of all state values $V(s)$ and iteratively updating the state values by unrolling the Bellman equation (Equation 2.33) one step at a time. The full procedure is outlined by Algorithm 7.

Algorithm 7 Value iteration

Input: finite set of states $\mathcal{S}$, finite set of actions $\mathcal{A}$, transition and reward dynamics $\Pr(s', r \mid s, a)$.
Initialise: all state values $v(s) \leftarrow 0$.

```
repeat
    for all s ∈ S do
        for all a ∈ A do
            q(s, a) ← Σ_{s',r} Pr(s', r | s, a)[r + γv(s)]      ▷ unroll the Bellman equation
        end for
        v(s) = max_a q(s, a)                                    ▷ update state values
    end for
until convergence
return optimal policy π ≈ π* using v(s) and Equation 2.37.
```

An important property of value iteration and other dynamic programming methods is that they are guaranteed to converge to the optimal state-value function $v_*(s)$, and thus recover the optimal policy π^*, the proof can be found here [74, Ch. 14.4.1]. Additionally, they are very efficient when compared with linear programming methods, and in general they converge must faster than their worst-case guarantees [14, Ch. 4.7].

However, unfortunately these dynamic programming methods suffer from the *curse of dimensionality*. The procedure outlined in Algorithm 7 iterates over all states $s \in \mathcal{S}$ and actions $a \in \mathcal{A}$. For most practical problems the state space $\mathcal{S}$ grows combinatorically large, think about how many possible configurations of a chess board exist. Additionally, for methods like value and policy iteration we require a perfect model of the transition

dynamics of the environment, that is, we need access to $\Pr(s', r \mid s, a)$. This requirement is often never satisfied in practice and in fact there is a whole sub-field of RL called *model-based* RL that aims to learn the transition dynamics of the environment, for planning and other procedures like local policy optimisation.

Policy iteration: [14, Ch. 4.3] is an alternative dynamic programming method that stores and iteratively updates the policy π by running policy evaluation and policy improvement until convergence. The general procedure is to start from a random policy π_0, fix this policy and evaluate it over all states $s \in \mathcal{S}$ computing the value function by,

$$v_{\pi_0}(s) = \mathbb{E}_{\pi_0}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \mid S_t = s] \tag{2.39}$$

Typically we can only unroll the discounted accumulated reward one step at a time (see Equation 2.32), since the number of terms in the expectation may explode quickly when we have stochastic transition and reward dynamics. After we have computed (or approximated) the value function of the current policy π we update the policy 'greedily' (policy improvement) so that is satisfies,

$$\pi'(a \mid s) = \begin{cases} 1 & \text{if } a = \text{argmax}_a \sum_a \Pr(s', r \mid s, a)(r + \gamma v_\pi(s)) \\ 0 & \text{otherwise} \end{cases} \tag{2.40}$$

By iteratively applying policy evaluation and policy improvement, we generate a sequence on monotonically improving policies and value functions [14, p. 80]. Because there are only a finite number of deterministic policies for a finite MDP, this procedure is also guaranteed to converge to the optimal policy π^*. Although it still suffers from the same drawbacks as value iteration. The full procedure is outlined by Algorithm 8.

The key takeaway here, is that both algorithms are efficient and optimal. And although they are rarely practical, ideas like policy evaluation, policy improvement and value iteration have been fundamental for the development and success of much more impressive RL algorithms.

Algorithm 8 Policy iteration

Input: finite set of states $\mathcal{S}$, finite set of actions $\mathcal{A}$, transition and reward dynamics $\Pr(s', r \mid s, a)$.

Initialise: the current policy π randomly. Set $v(s) \leftarrow 0$ for all $s \in \mathcal{S}$.

repeat

  for all $s \in \mathcal{S}$ **do**

    $v(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s',r} \Pr(s', r \mid s, a)[r + \gamma v(s')]$ ▷ one-step policy evaluation

  end for

  Update the current policy π using $v(s)$ and Equation 2.40. ▷ policy improvement

until convergence

return the optimal policy $\pi \approx \pi^*$.

2.3.4 Q-learning

The *Q-learning* algorithm acts as the theoretical basis for most value-based methods utilised in deep RL. Q-learning [75] is an off-policy temporal difference (TD) control algorithm. For those readers that are not familiar with the taxonomy of RL algorithms please refer to Table C.1 in Appendix C.1. The Q-learning algorithm works by maintaining a table of state-action values, $Q(s, a)$, that directly approximate the optimal state-action value function q^*. In contrast to the dynamic programming methods covered in the previous section, Q-learning learns from experience generated by an agent interacting with the environment, rather than iterating over all states $s \in \mathcal{S}$ and actions $a \in \mathcal{A}$. Starting from some state S_t the agent picks an action A_t according to the current policy π derived from $Q(s, a)$. Then using the received reward R_{t+1} and observed state S_{t+1}, the table $Q(s, a)$ is updated according to the following update rule,

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.41}$$

where α is a step-size parameter controlling how big the updates are. We note that the quantity $R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)$ refers to the TD error δ_t. In contrast to Monte Carlo methods [14, Ch. 5] which estimate, $v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$ as a target, TD methods estimate and use $v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s]$ as a target. Batch TD methods typically converge faster than batch Monte Carlo methods as they aim to find es-

timates that would be correct for a maximum likelihood model of the MDP, whereas Monte Carlo methods find estimates that minimise the mean squared error [14, p. 128].

Convergence guarantees for Q-learning require that each state-action pair $(s, a) \in \mathcal{S} \times \mathcal{A}$ is visited infinitely often [74, p. 332]. As such, we can't just blindly follow the "greedy" policy,

$$\pi(a \mid s) = \begin{cases} 1 & \text{if } a = \operatorname{argmax}_a Q(s, a) \\ 0 & \text{otherwise} \end{cases} \tag{2.42}$$

Instead we employ an ε-greedy strategy, whereby with probability ε we pick a random action $a \in \mathcal{A}$ and with remaining probability we follow the "greedy" policy given by Equation 2.42. As a result the Q-learning algorithm can be viewed as a stochastic version of the value iteration algorithm presented in the previous section. Algorithm 9 outlines the general Q-learning algorithm in procedural form.

Algorithm 9 Q-learning

Input: learning rate $\alpha \in (0, 1]$, epsilon exploration parameter $\varepsilon > 0$.
Initialise: table of state-action value $Q(s, a)$ arbitrary for all $s \in \mathcal{S}$, $a \in \mathcal{A}$, except for terminal states, set $Q(s', a) \leftarrow 0$ for all $s' \in \mathcal{S}_{\text{terminal}}$, $a \in \mathcal{A}$.

Loop for each episode
  Observe start state S.
  Loop for each step
    With probability ε pick randomly an action A from $\mathcal{A}$.
    O/w pick A from S according to the "greedy" policy $\pi(s) = \operatorname{argmax}_a Q(s, a)$.
    Play action A and observe reward R and next state S'.
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
    $S \leftarrow S'$
  until S is terminal
return optimal policy $\pi \approx \pi^*$ using Q and Equation 2.38.

The most important property of this algorithm is that the update step is independent of the policy being followed. This property simplifies the analysis of the algorithm which lead to early convergence proofs [76]. Specifically, it can be shown that Q will converge to probability q^* with probability 1, under some reasonable assumptions about the step-size

parameter α and if all state-action pairs $(s, a) \in \mathcal{S} \times \mathcal{A}$ continue to be updated, see [77] for a simple proof of this fact.

Additionally, the Q-learning algorithm is a our first example of a model-free algorithm, it does not require a perfect model of the environment, or more formally, access to $\Pr(s', r \,|\, s, a)$. This makes it a more practical algorithm than those outlined in the previous section, although unfortunately, for large state and action spaces it is infeasible to store Q-values $Q(s, a)$ for all state-action pairs $(s, a) \in \mathcal{S} \times \mathcal{A}$. Instead we can rely on linear models or function approximators, which give us a map $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that approximate the Q function.

This idea encouraged the development of deep Q-learning, one of the first of many success stories in learning good policies from high-dimensional input using neural networks [19, 20]. In 2013 Mnih *et al.* introduced the *deep Q-network* (DQN) that achieved super human performance on several Atari 2600 games, learning directly from raw pixel data. Before this, most of the successes in RL were limited to domains with handcrafted features or with fully observable low-dimensional state spaces. By utilizing recent successes in deep learning at the time, DQNs were trained with backpropagation and a variant of stochastic gradient descent, *root mean squared propagation* (RMSProp) in an end-to-end fashion. While this success was certainly a major engineering effort, the DQN algorithm is grounded in the *Q-learning* update step described in this section. The full DQN algorithm will be presented in the next chapter, along with our proposed modifications to better facilitate learning with PC.

2.3.5 Policy Gradient Methods

Instead of approximating the optimal state-action value function q_*, policy gradient methods directly model the policy and frame the RL problem in terms of policy optimisation. Policy gradient methods can parameterise the policy in anyway, as long as the learned policy $\pi(a \,|\, s, \boldsymbol{\theta})$ is differentiable with respect to its parameters. For some simple problems we can use linear models to represent the policy, but it is now much more common use

neural networks, which are capable of modelling more complex dependencies. Regardless, we denote the parameters of our policy by $\boldsymbol{\theta}$. The general approach of policy gradient methods is to first specify some scalar objective function $J(\boldsymbol{\theta})$. The goal is then to maximise the objective function with respect to the parameters, this gives us an update step reminiscent of gradient ascent,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\widehat{\nabla J(\boldsymbol{\theta})} \tag{2.43}$$

where $\widehat{\nabla J(\boldsymbol{\theta})}$ is a stochastic estimate of the gradient objective function, whose expectation approximates the true gradient [78, p. 321], and α is a learning rate parameter. There are several advantages to modelling the policy directly, the main one being that we can represent stochastic policies. In some environments with partial observability, the optimal policy may be a stochastic one, and by using deterministic policies we fail to capture this property. Additionally, by picking actions according to a stochastic policy we encourage exploration in a more natural way, as opposed to the artificial ε-greedy strategy used for value-based methods like DQNs. Furthermore, policy-gradient methods can be very easily extended to continuous action spaces, making them much more practical for continuous control tasks.

In addition to these practical advantages, policy gradient methods also have important theoretical advantages over value-based methods. By representing the policy in terms of continuous probabilities, the action probabilities change more smoothly with small updates to the model parameters, whereas small changes to the model parameters of a value-based Q-network can dramatically change the action probabilities from 0 to 1 or vice versa [78, Ch. 13.2]. Because of this advantage, policy gradient methods tend to come with stronger convergence guarantees than value-based methods. In particular, it is the continuity of the model parameters that allows the update step to approximate gradient ascent (equation 2.43) [78, p. 324].

In this section, we will continue by describing how policy gradient methods can be framed

in terms of both discrete and continuous action spaces. We will then conclude this section by briefly describing two important policy gradient algorithms, which form the theoretical basis for the proximal policy optimisation (PPO) algorithm [45] described in the next chapter.

Discrete action space: if the action space is not too large then it is common to parameterise the model in such a way that it outputs a set of scalar preferences $h(s, a, \boldsymbol{\theta}) \in \mathbb{R}$ for all pairs $(s, a) \in \mathcal{S} \times \mathcal{A}$. The probability of an action being selected is then given by the softmax probabilities,

$$\pi(a \mid s, \boldsymbol{\theta}) = \frac{e^{(h(s,a,\boldsymbol{\theta}))}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}} \tag{2.44}$$

These action preferences can be the output of a deep network Φ, that is $h(s, a, \boldsymbol{\theta}) = \Phi_{\boldsymbol{\theta}}(s, a)$. Alternatively, they could be computed by a linear model $h(s, a, \boldsymbol{\theta}) = \boldsymbol{\theta}^T \mathbf{x}(s, a)$, where $\mathbf{x}(s, a)$ is some state-action feature vector. Consider some fixed start state s_0, we then define the objective function to be,

$$J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(s_0) \tag{2.45}$$

where $v_{\pi_{\boldsymbol{\theta}}}(s_0)$ is the value function for the policy $\pi_{\boldsymbol{\theta}}$ parameterised by $\boldsymbol{\theta}$. It should be clear that the policy that maximises this value function, maximises the expected return, which is what it means to be the optimal policy. By applying the *policy gradient theorem* (see [14, p. 325]), we can derive an analytic expression for the gradient of this objective function with respect to the parameters $\boldsymbol{\theta}$,

$$J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_{\pi_{\boldsymbol{\theta}}}(s, a) \nabla \pi(a \mid s, \boldsymbol{\theta}) \tag{2.46}$$

where $\pi_{\boldsymbol{\theta}}$ denotes the policy corresponding to parameters $\boldsymbol{\theta}$. The distribution $\mu(s)$ is the on-policy distribution of states under $\pi_{\boldsymbol{\theta}}$ [14, p. 326]. By iteratively maximising this objective function we expect to converge to the optimal policy π^*.

Continuous action space: for infinitely large action spaces we instead learn the parameters of probability distributions over a subset of the reals. For example the action space might be all the reals, $\mathbb{R}$, and in this case we typically model the policy as a normal (Gaussian) distribution. Writing the probability density function of the normal distribution we see it has only two parameters μ and σ,

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{2.47}$$

Letting $\boldsymbol{\theta} = [\boldsymbol{\theta}_\mu, \boldsymbol{\theta}_\sigma]^T$, we can write the policy as,

$$\pi(a \mid s, \boldsymbol{\theta}) = \frac{1}{\sigma(s, \boldsymbol{\theta}_\sigma)\sqrt{2\pi}} \exp\left(-\frac{(x-\mu(s, \boldsymbol{\theta}_\mu))^2}{2\sigma(s, \boldsymbol{\theta}_\sigma)^2}\right) \tag{2.48}$$

where $\mu : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$ and $\sigma : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}^+$ are function approximators of the form,

$$\mu(s, \boldsymbol{\theta}_\mu) = \Phi_{\boldsymbol{\theta}_\mu}(s) \quad \text{and} \quad \sigma(s, \boldsymbol{\theta}_\sigma) = \exp\left(\Phi_{\boldsymbol{\theta}_\sigma}(s)\right) \qquad \text{(deep network)} \tag{2.49}$$

$$\mu(s, \boldsymbol{\theta}_\mu) = \boldsymbol{\theta}_\mu^T \mathbf{x}_\mu(s) \quad \text{and} \quad \sigma(s, \boldsymbol{\theta}_\sigma) = \exp\left(\boldsymbol{\theta}_\sigma^T \mathbf{x}_\sigma(s)\right) \qquad \text{(linear model)} \tag{2.50}$$

Here $\mathbf{x}_\mu(s)$, $\mathbf{x}_\sigma(s)$ are state feature vectors, $\Phi_{\boldsymbol{\theta}_\mu}$ and $\Phi_{\boldsymbol{\theta}_\sigma}$ are deep networks parameterised by $\boldsymbol{\theta}_\mu$ and $\boldsymbol{\theta}_\sigma$ respectively. In this setting it is common to only model the mean $\mu(s, \boldsymbol{\theta}_\mu)$ as a state conditional distribution. The variance σ^2 is either fixed or modelled as a learnable parameter independent of the state.

This formulation can easily be extended to the multi-variate normal distribution when we need to choose several continuous actions at the same time. In this case the variance is typically modelled as a diagonal covariance matrix $\boldsymbol{\Lambda}$, that is either fixed or learnable.

REINFORCE: the gradient update step of the REINFORCE algorithm is derived from the policy gradient theorem (Equation 2.46), by taking expectations under the current policy π,

$$J(\boldsymbol{\theta}) \propto \mathbb{E}_{\pi}\left[G_t \frac{\nabla \pi(A_t \mid S_t, \boldsymbol{\theta})}{\pi(A_t \mid S_t, \boldsymbol{\theta})}\right] \tag{2.51}$$

This yields the REINFORCE update:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla \pi(A_t \mid S_t, \boldsymbol{\theta})}{\pi(A_t \mid S_t, \boldsymbol{\theta})} \tag{2.52}$$

The full derivation of this update step is given in Appendix A.4. This update has several intuitive properties, for example, the gradient is scaled by the total return G_t so that actions resulting in high return have higher gradient than those that do not. Additionally, the gradient is divided by the probability of an action, so that more probable actions do not have a higher gradient than other better actions simply because they have a higher probability. To simplify the update step, the gradient of the logarithm of $\pi(A_t \mid S_t, \boldsymbol{\theta})$ is used instead, since $\nabla \ln x = \frac{\nabla x}{x}$. The full procedure is outlined by Algorithm 10.

Algorithm 10 REINFORCE

Input: differentiable policy $\pi(a \mid s, \boldsymbol{\theta})$, step size $\alpha > 0$.
Initialise: the model parameters $\boldsymbol{\theta}$ randomly.

Loop forever
  Play an episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$ by following $\pi(\cdot \mid \cdot, \boldsymbol{\theta})$.
  for $t \leftarrow 1$; $t < T$; $t \leftarrow t + 1$
    $G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G_t \nabla \ln \pi(A_t \mid S_t, \boldsymbol{\theta})$

REINFORCE has good theoretical guarantees, since by construction, the expected update over an episode is in the direction that increases performance. Although, the gradient update step requires the full return of the episode to be computed. This makes REINFORCE a Monte Carlo method, and as such, the gradient updates can have high variance, which results in slow or unstable convergence.

Actor-critic methods: actor-critic methods aim to reduce the variance of the policy gradient by introducing bias into the objective function. One-step actor-critic methods are the policy gradient analog of TD methods like Q-learning. The one-step actor-critic algorithm replaces the full return in the REINFORCE gradient update step with the one-step return,

$$\begin{aligned}\boldsymbol{\theta}_{i+1} &= \boldsymbol{\theta}_i + \alpha \left(G_{t:t+1} - \hat{v}(S_t, \mathbf{w})\right) \nabla \ln \pi(A_t \mid S_t, \boldsymbol{\theta}_i) \\ &= \boldsymbol{\theta}_i + \alpha \left(R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})\right) \nabla \ln \pi(A_t \mid S_t, \boldsymbol{\theta}_i) \\ &= \boldsymbol{\theta}_i + \alpha \hat{\delta}_t \nabla \ln \pi(A_t \mid S_t, \boldsymbol{\theta}_i) \end{aligned} \tag{2.53}$$

where $\hat{v}(s, \mathbf{w})$ is a trainable state-value function parameterised by $\mathbf{w}$, typically called the critic. The actor on the other hand, refers to the policy $\pi(a \mid s, \boldsymbol{\theta})$ parameterised by $\boldsymbol{\theta}$. Algorithm 11 outlines the one-step actor-critic method in procedural form.

Algorithm 11 One-step actor-critic

Input: differentiable actor $\pi(a \mid s, \boldsymbol{\theta})$, actor step size $\alpha^{\boldsymbol{\theta}} > 0$, differentiable critic $\hat{v}(s, \mathbf{w})$, critic step size $\alpha^{\mathbf{w}} > 0$.
Initialise: the actor parameters $\boldsymbol{\theta}$ and critic parameters $\mathbf{w}$ randomly.
  Loop for each episode
    Observe start state S.
    $I \leftarrow 1$
    Loop for each step
      $A \sim \pi(\cdot \mid S, \boldsymbol{\theta})$
      Play action A, observe reward R and next state S'.
      $\hat{\delta} \leftarrow R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \hat{\delta} \nabla v(S, \mathbf{w})$
      $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \hat{\delta} \nabla \ln \pi(A \mid S, \boldsymbol{\theta})$
      $I \leftarrow \gamma I$
      $S \leftarrow S'$
    until S is terminal

Modern implementations of actor-critic methods typically extend this procedure to batch computations, where experience is generated from multiple actors interacting with the environment in parallel. It is also common to unroll the Bellman equation more than just one step at a time.

Chapter 3

Methodology

In this chapter we outline the main RL algorithms used to conduct experimental comparisons between BP and PC. Specifically, we cover the cross-entropy method [79], deep Q-learning [19] and proximal policy optimisation (PPO) [45]. In addition, we propose some small algorithmic changes to the deep Q-learning algorithm and the PPO algorithm which are intended to facilitate better learning with PC. All these algorithmic changes are detailed in full and we conclude this chapter by detailing the general methods used for comparing the performance of networks trained with BP and PC in Chapter 4.

3.1 The Cross-Entropy Method

The *cross-entropy method* (CEM) was first proposed as an adaptive importance sampling procedure for estimating the probabilities of rare events [79]. The CEM has been successfully applied to a wide variety of problems, such as, continuous optimal control problems [80], optimal policy search [81], DNA sequence alignment [82] and various practical optimisation tasks [83–85]. While the CEM is certainly not the most famous RL algorithm out there, it stands as a very simple example with nice theoretical backing.

The general idea with the CEM is to model the policy as some trainable non-linear function approximator, for example, by using an ANN or a PCN. Recall that the policy is a

distribution over actions $a \in \mathcal{A}$ given some state $s \in \mathcal{S}$ and so the model must implement the softmax function at the output layer, to give a valid probability distribution that we can sample from. The theoretical basis for the CEM comes from the importance sampling theorem, which states,

$$\mathbb{E}_{x\sim p(x)}[H(x)] = \int_x p(x)H(x) = \int_x q(x)\frac{p(x)}{q(x)}H(x)dx = \mathbb{E}_{x\sim q(x)}\left[\frac{p(x)}{q(x)}H(x)\right] \tag{3.1}$$

where p and q are probability densities and $H(x)$ is some function. In terms of policy optimisation, the CEM can be framed in terms of maximising the future return G_t drawn from an MDP under some policy π. Formally, we are tasked with solving the following optimisation problem,

$$\max_{\pi} \mathbb{E}_{\pi}[G_t] \tag{3.2}$$

The CEM iteratively updates the policy π changing the distribution on G_t in such a way that we hope to maximise the quantity $\mathbb{E}_{\pi}[G_t]$. Let π_i denote the current policy at iteration i. We start by fixing the current policy π_i and drawing a sample of N episodes from the MDP. The (discounted) accumulated reward for each episode is computed and the $(1-\rho_{\text{CE}})$ quantile λ_i of the sample is computed, where $\rho_{\text{CE}} \in (0,1)$ is a user-specified hyperparameter. By applying the importance sampling theorem, we define the policy update to be,

$$\pi_{i+1} = \underset{\pi'}{\text{argmax}} \left\{\mathbb{E}_{\pi_i}[\mathbb{1}[G_t \geq \lambda_i]] \log(\pi'(a \mid s))\right\} \tag{3.3}$$

Since we approximate the expectation with a batch of sampled episodes, it is more correct to write the update step as,

$$\pi_{i+1} = \underset{\pi'}{\text{argmax}} \left\{\frac{1}{N}\sum^{N}[\mathbb{1}[G_t \geq \lambda_i] \log(\pi'(a \mid s))\right\} \tag{3.4}$$

The CEM doesn't directly apply this update, instead this update step is simulated by

training the function approximator only on "elite" episodes that satisfy $\mathbb{1}[G_t \geq \lambda_i]$. The function approximator is trained to minimise the cross entropy loss between the current policy $\pi_i(a|s)$ and the actions picked during the "elite" episodes. At each iteration the idea is that the distribution of episodes will shift to the the one that maximises the quantity, $\max_\pi \mathbb{E}_\pi[G_t]$, namely, the optimal policy π^*. Algorithm 12 provides a very high-level outline of the full procedure.

Algorithm 12 The cross-entropy method

Initialse: differentiable classifier π_0 with random weights and biases.
Output: trained classifier π

repeat
  Play N episodes with the currently policy π_i.
  Construct a batch of experience B by keeping the "elite" episodes.
  Train on the batch B and update π_i to minimise the cross entropy loss $\mathcal{L}_{\text{cross entropy}}$.
until convergence

For readers interested in further theoretical justification of the CEM, please refer to the following resources: [81, 86]. The simplicity of the CEM allows us to establish a fair framework for the comparison of BP and PC on some toy RL environments. During training, to evaluate the performance of BP or PC, we plot the average batch reward every batch iteration. We also use the mean reward as a way of comparing BP and PC. The batch reward for some batch B is computed by,

$$\text{batch reward} = \frac{1}{|B|} \sum_{i \in B} \sum_{t=1}^{T_i} r_{i,t} \tag{3.5}$$

where $r_{i,t}$ denotes the reward received at time step t during episode i and T_i denotes the length of episode i. Additionally, the CEM provides us with an opportunity to show that PCNs can still be effectively trained when the output loss function doesn't match the local energy functions of the internal layers.

3.2 Deep Q-learning

For high dimensional input like raw pixel data, the state space is much too big to store in memory and we would suffer from very slow convergence anyway. Deep Q-learning is a work around that uses non-linear function approximators to represent the Q function, $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. To develop a fair and straightforward framework for the comparison of DQNs trained with BP and PC, we aim to emulate the original DQN algorithm [19] as closely as possible. Some key components of the original DQN algorithm [19] that we use in our implementation include the following:

- **Function approximation:** ANNs are universal function approximators that generalise well to unseen data. Mnih *et al.* processed the raw pixel data by first gray-scaling it and resizing it, before passing it through a deep neural network consisting of 3 convolutional and 2 linear layers, all separated by ReLU activation functions.

- **Epsilon greedy policy:** The exploration-exploitation trade off is a fundamental unsolved problem in RL. Q-learning is guaranteed to converge to q^* provided all states are visited often enough. For large state spaces this requirement cannot feasibly be satisfied. Although, by picking a random action with probability ε we hope to sufficiently explore the state space to learn good policies. The parameter ε is usually set to 1.0 at the start of training and is decayed during training by some user-specified schedule, this is so that the agent starts exploiting what it has learned in order to maximise accumulated reward.

- **Experience replay:** to deal with the problem of "catastrophic interference" or "catastrophic forgetting", past experience sampled from the environment is kept in a *replay buffer*. During training the agent samples a batch experience from the replay buffer to train on rather than training on the most immediate experience which could overwrite past experience.

- **Target network:** in the Q-learning update step (Equation 2.41) the Q function is used to derive the target TD error we wish to minimise: $\delta_t = R_{t+1} +$

$\gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)$. Using the Q-function itself to derive to target δ_t can result in instability during training. Instead a target network $\widehat{Q}(s, a)$ with frozen parameters is used to derive the TD targets. Every N steps the parameters of $Q(s, a)$ are copied to $\widehat{Q}(s, a)$.

- **Reward clipping:** reward signals are clipped to the range $[-1, 1]$ to prevent catastrophic divergence during learning. Divergence occurs when the Q-function predicts unrealistically high state-action values which can destroy the stability of training. By clipping the rewards in the range $[-1, 1]$, the future return is bounded by,

$$\sum_t^T \gamma^{t-1} r_t \leq \sum_t^T \gamma^{t-1} |r_t| \leq \sum_t^T \gamma^{t-1} = \frac{1}{1-\gamma} \tag{3.6}$$

- **Frame stacking and skipping:** the Atari 2600 emulators run at 60 frames per second. There is strong correlation between consecutive frames, so frame skipping with $k = 4$ is used to break this correlation and learn more quickly. Theoretically this can be thought of as unrolling the Bellman equation k steps. In addition, frame stacking helps capture temporal dependencies that aren't captured in the raw pixel data, for example, the direction a ball is moving.

Since the inception of the original DQN algorithm, many extensions have been proposed, such as, N-step DQN [78], double DQN [87], prioritised experience replay [88] and many more [89–91]. Our aim is not to over complicate things, and so we will only make the following two modifications to the original DQN algorithm:

- We modify the loss function at the output layer to be the sum of squared errors divided by 2, as opposed to the normal mean squared error loss.

$$\mathcal{L}_{\text{SSE}} = \frac{1}{2} \cdot \sum_{i=1}^{B} (\hat{y}_i - y_i)^2 \tag{3.7}$$

where B is a batch of indices. We propose this form for the loss function since it has the same form as the local energy functions of the internal layers of the PCN, see

Section 2.2.2. This means we can use SGD for inference without worrying about the norm of the loss at the output layer being different from the norms at the internal layers.

- We will use the double Q-learning targets [87],

$$y = R + \gamma \widehat{Q}(S', \underset{a}{\operatorname{argmax}}\, Q(S', a)) \tag{3.8}$$

instead of the original Q-learning targets,

$$y \leftarrow R + \gamma \underset{a}{\operatorname{argmax}}\, \widehat{Q}(S', a) \tag{3.9}$$

for computing the loss at the output layer. This reason for this choice is that double Q-learning is very simple trick that makes training remarkably more stable for Atari 2600 games and as a result speeds up the convergence of training.

Algorithm 13 presented on the next page, reflects these modifications and provides a more precise description of the deep Q-learning algorithm that we will use.

3.2.1 Clamp Loss

For PCNs used as Q-function approximators we propose a slightly different loss function at the output layer. By adopting this new loss function called the *clamp loss* we show that PCNs converge better in some of the scenarios presented in the next chapter. Recall that the target Q-values for double Q-learning are computed as follows,

$$y = R + \gamma \widehat{Q}(S', \underset{a}{\operatorname{argmax}}\, Q(S', a)) \tag{3.10}$$

and so the gradient step is,

$$\nabla \delta_t \propto \nabla (R + \gamma \widehat{Q}(S', \underset{a}{\operatorname{argmax}}\, Q(S', a)) - Q(S, A)) \tag{3.11}$$

Algorithm 13 Deep Q-learning

Input: learning rate $\alpha \in (0, 1]$, epsilon exploration parameter $\varepsilon > 0$
Initialise: the parameters of the network Q and the target network $\widehat{Q}$ randomly.

Loop for each episode
  Observe start state S.
  Loop for each step
    With probability ε pick randomly an action A from $\mathcal{A}$.
    Otherwise pick A from S according to the 'greedy' policy $\pi(S) = \text{argmax}_a\, Q(S, a)$
    Play action A and observe reward R, next state S' and done flag D.
    Store the tuple $\langle S, A, R, S', D\rangle$ in the replay buffer.
    Sample a random mini-batch of transitions from the replay buffer.
    For every transition in the batch compute the target:

$$y \leftarrow R \quad \text{if } D = \text{True} \quad \text{else}$$
$$y \leftarrow R + \gamma \hat{Q}(S', \underset{a}{\text{argmax}}\, Q(S', a))$$

    Calculate the batch loss $\mathcal{L}_{\text{SSE}} = \frac{1}{2} \cdot \sum^{B} (Q(S, A) - y)^2$.
    Update the parameters of Q in the direction that minimises $\mathcal{L}_{\text{SSE}}$.
    Every N steps copy the parameters of Q to $\widehat{Q}$.
  until done D
return trained network Q

If we use these targets with PC this only clamps one of the value nodes at the output layer: the value node corresponding to action A. This means during inference the other value nodes at the output layer are free to change. Instead we propose the *clamp loss* function for PC Q-networks, which clamps the value nodes at the output layer to their corresponding activations computed in the forward pass, and clamps the action A to the target value computed by Equation 3.10. Let the vector $Q(S, \cdot)$ be the output of the network computed during a forward pass. By applying the following two equations we compute the clamp loss target,

$$\mathbf{y} = Q(S, \cdot) \tag{3.12}$$

$$y_A \leftarrow R + \gamma \widehat{Q}(S', \underset{a}{\text{argmax}}\, Q(S', a)) \tag{3.13}$$

Then the clamp loss function for a single datapoint is written as,

$$\mathcal{L}_{\text{clamp}} = \frac{1}{2} \sum_{a \in \mathcal{A}} (Q(S, a) - y_a)^2 \tag{3.14}$$

This can easily be extended to batch computations by summing over the batch. Also note that if we were to use the clamp loss for BP, this would not be any different from using the original targets. Because by construction $Q(S, a) - y_a = 0$ for all $a \neq A$, and so for BP, the scalar loss computed at the output layer would be exactly the same as before.

3.3 Proximal Policy Optimisation

The proximal policy optimization (PPO) [45] proposed by Schulman *et al.* is a clever improvement over actor-critic methods, that changes the objective function of the policy. Instead of using the gradient of the logarithm of the policy scaled by the advantages, the PPO algorithm uses the ratio between the new and old policy scaled by the advantages. First, we will explain what is meant by the advantages in relation to actor-critic methods.

Recall that the one-step actor-critic objective function corresponds to the gradient of the logarithm of the policy $\nabla \ln \pi(A_t \mid S_t, \boldsymbol{\theta})$ scaled by the TD error $\hat{\delta}_t$, where,

$$\hat{\delta}_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}) \tag{3.15}$$

Asynchronous Actor-Critic (A2C) methods [92] generalise this notion to the T-step advantage $\hat{A}_t$, not to be confused with A_t, the action at time t. Here T is some integer much smaller than the episode length, and t is some timestep in the range $[0, T]$. This style of advantage estimation was introduced by Mnih *et al.* [92] and doesn't look beyond timestep T. The T-step advantage $\hat{A}_t$ at time t is defined as,

$$\hat{A}_t = -\hat{v}(S_t, \mathbf{w}) + R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-t+1} R_T + \gamma^{T-t} \hat{v}(S_T, \mathbf{w}) \tag{3.16}$$

Note that the T-step advantage $\hat{A}_t$ is only an estimate of the true advantage, this is because we use $\hat{v}(S_t, \mathbf{w})$ as an estimate of the state-value function. Observe that when $T = 1$ we have,

$$\hat{A}_t = -\hat{v}(S_t, \mathbf{w}) + R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) = \delta_t \tag{3.17}$$

We can then write the A2C update step as,

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i + \alpha \hat{A}_t \nabla \ln \pi(A_t \mid S_t, \boldsymbol{\theta}_i) \tag{3.18}$$

It should be clear that the one-step actor-critic algorithm is a special case of the A2C algorithm with $T = 1$. We now write the objective function for A2C as follows,

$$J(\boldsymbol{\theta}) = \hat{\mathbb{E}}_t \left[\ln \pi(A_t \mid S_t, \boldsymbol{\theta}) \hat{A}_t \right] \tag{3.19}$$

where $\hat{\mathbb{E}}_t$ is the empirical expectation over policy π. We are now equipped to describe the PPO objective function. The PPO algorithm [45] modifies the A2C algorithm by maximising the "surrogate" objective function,

$$J(\boldsymbol{\theta}) = \hat{\mathbb{E}}_t \left[\frac{\pi(A_t \mid S_t, \boldsymbol{\theta})}{\pi(A_t \mid S_t, \boldsymbol{\theta}_{\text{old}})} \hat{A}_t \right] = \hat{\mathbb{E}}_t \left[r_t(\boldsymbol{\theta}) \hat{A}_t \right] \tag{3.20}$$

where $r_t(\boldsymbol{\theta}) = \frac{\pi(A_t \mid S_t, \boldsymbol{\theta})}{\pi(A_t \mid S_t, \boldsymbol{\theta}_{\text{old}})}$ is the ratio between the new policy parameterised by $\boldsymbol{\theta}$ and the old policy parameterised by $\boldsymbol{\theta}_{\text{old}}$. Blindly maximising this objective function would lead to large excessively large parameter updates [45]. Instead the clipped objective function is used to compute the policy loss,

$$J^{\text{CLIP}}(\boldsymbol{\theta}) = \hat{\mathbb{E}}_t \left[\min \left\{ r_t(\boldsymbol{\theta}) \hat{A}_t, \mathsf{clip}(r_t(\boldsymbol{\theta}), 1 - \epsilon, 1 + \epsilon) \hat{A}_t \right\} \right] \tag{3.21}$$

where ϵ is some user-specified hyperparameter that clips the ratio to a suitable range (usually $\epsilon \leftarrow 0.2$). For PPO the estimated T-step advantages are also computed in a

slightly different way,

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + ... + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{3.22}$$

$$\text{where} \qquad \delta_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}) \tag{3.23}$$

This way of computing the advantages is a bias-variance trade-off proposed by Schulman *et al.* [93]. When $\lambda = 1$ this is the normal advantage estimation used by A2C, which corresponds to high variance. Additionally, to encourage exploration, an entropy bonus term with coefficient β is added to the objective function,

$$J^{\text{ENTROPY}}(\boldsymbol{\theta}) = \hat{\mathbb{E}}_t \left[-\log(\pi(A_t \mid S_t, \boldsymbol{\theta})) \right] \tag{3.24}$$

As with actor-critic methods, we jointly train an actor $\pi(a \mid s, \boldsymbol{\theta})$ parameterised by $\boldsymbol{\theta}$ to maximise the objective function $J(\boldsymbol{\theta})$, along with a critic $\hat{v}(s, \mathbf{w})$ parameterised by $\mathbf{w}$. The critic is trained to minimise the TD error, that is,

$$J^{\text{CRITIC}}(\mathbf{w}) = \hat{\mathbb{E}}_t \left[\hat{(}\hat{v}(S_t, \mathbf{w}) - \hat{V}_t^{\text{target}})^2 \right] \tag{3.25}$$

where the target $\hat{V}_t^{\text{target}} = R_{t+1} + \lambda\hat{v}(S_{t+1}, \mathbf{w}) + (\gamma\lambda)\delta_{t+1} + ... + (\gamma\lambda)^{T-t+1}\delta_{T-1}$. The full PPO objective function can now be written as,

$$\underset{\boldsymbol{\theta}, \mathbf{w}}{\text{minimise}} \left\{ J^{\text{CRITIC}}(\mathbf{w}) - \beta J^{\text{ENTROPY}}(\boldsymbol{\theta}) - J^{\text{CLIP}}(\boldsymbol{\theta}) \right\} \tag{3.26}$$

In practice the PPO algorithm follows a slightly different training procedure than classic actor-critic algorithms: long sequences of samples are obtained from interacting with the environment, the advantage and value estimates, $\hat{A}_t$ and $\hat{V}_t^{\text{target}}$, are then computed by evaluating the whole sequence, before several batch updates are performed. Algorithm 14 on the next page, gives the full PPO algorithm (for continuous action spaces), that we will use for comparing the performance of BP and PC in challenging continuous control tasks.

Algorithm 14 Proximal Policy Optimisation (Continuous Case)

Input: differentiable actor $\pi(a \mid s, \boldsymbol{\theta})$. differentiable critic $\hat{v}(s, \mathbf{w})$,
Initialise: actor parameters $\boldsymbol{\theta}$, diagonal covariance matrix $\boldsymbol{\Lambda}$ and critic parameters $\boldsymbol{w}$.

repeat
observe start state S_1.
empty replay buffer.
for $t \leftarrow 1 \; ; \; t < T + 1 \; ; \; t \leftarrow t + 1$ **do**
Compute policy $\boldsymbol{\pi}_t \leftarrow \pi(\cdot \mid S_t, \boldsymbol{\theta})$.
Compute the value $V_t \leftarrow v(S_t, \mathbf{w})$.
Play action $A_t \sim \boldsymbol{\pi}_t + \mathcal{N}(0, \boldsymbol{\Lambda})$.
Observe reward R_{t+1}, next state S_{t+1} and done flag D_{t+1}.
Store the tuple $\langle S_t, A_t, R_{t+1}, S_{t+1}, \boldsymbol{\pi}_t, V_t, D_{t+1} \rangle$ in the replay buffer.
if done flag $D_{t+1} =$ True **then**
Reset environment and observe start state S_{t+1}.
end if
end for
for $t \leftarrow 1 \; ; \; t < T + 1 \; ; \; t \leftarrow t + 1$ **do**
Compute the τ-step advantage estimate $\hat{A}_t$ and target state value V_t^{target}:

$$\begin{aligned} \hat{A}_t &\leftarrow \delta_t + (\gamma\lambda)\delta_{t+1} + ... + (\gamma\lambda)^{\tau-t+1}\delta_{\tau-1} \\ V_t^{\text{target}} &\leftarrow R_{t+1} + \lambda V_{t+1} + (\gamma\lambda)\delta_{t+1} + ... + (\gamma\lambda)^{\tau-t+1}\delta_{\tau-1} \\ \text{where} \quad & \tau \leftarrow \inf\{\tau' \geq t : D_{\tau'} = \text{True} \vee \tau = T\} \quad \text{and} \\ \text{where} \quad & \delta_t \leftarrow R_{t+1} + \gamma V_{t+1} - V_t \quad \text{if} \quad D_{t+1} = \text{False} \\ \text{else} \quad & \delta_t \leftarrow R_{t+1} - V_t \end{aligned}$$

Compute the current log probabilities LL_t^{old}:

$$\text{LL}_t^{\text{old}} \leftarrow -\frac{1}{2}\left[\ln(\boldsymbol{\Lambda}) + (A_t - \boldsymbol{\pi}_t)^T \boldsymbol{\Lambda}^{-1}(A_t - \boldsymbol{\pi}_t) + |\mathcal{A}| \ln(2\pi)\right] \tag{3.27}$$

end for
for $i \leftarrow 1 \; ; \; i < N + 1 \; ; \; i \leftarrow i + 1$ **do**
Sample a random batch of indices $B \subset \{1, 2, ..., T\}$.
Calculate the value loss:

$$\mathcal{L}_{\text{VALUE}} \leftarrow \frac{1}{|B|} \sum_{j \in B} (\hat{V}_j - V_j^{\text{target}})^2$$

Compute the new log probabilities LL_j^{new} for all $j \in B$ using Equation 3.27.
Compute the ratio $r_j \leftarrow \exp(\text{LL}_j^{\text{new}} - \text{LL}_j^{\text{old}})$ for all $j \in B$.
Calculate the policy loss:

$$\mathcal{L}_{\text{POLICY}} \leftarrow -\frac{1}{|B|} \sum_{j \in B} \min\left\{ r_j \hat{A}_j, \mathsf{clip}(r_j, 1-\epsilon, 1+\epsilon)\hat{A}_j \right\} - \frac{\beta}{|B|} \sum_{j \in B} \frac{D}{2}(1 + \log(2\pi)) + \frac{1}{2} \log |\boldsymbol{\Lambda}|$$

Update $\mathbf{w}$ in the direction that minimises $\mathcal{L}_{\text{VALUE}}$ ▷ with either BP or PC
Update $\boldsymbol{\theta}$, $\boldsymbol{\Lambda}$ in the direction that minimises $\mathcal{L}_{\text{POLICY}}$ ▷ with either BP or PC
end for
until convergence

3.3.1 Different Forms for the Loss Functions

For PCNs used as value and policy networks, we propose slightly different forms for the loss functions used to train these networks. From Algorithm 14 we recall that the value network is trained using the following loss function,

$$\mathcal{L}_{\text{VALUE}} \leftarrow \frac{1}{|B|} \sum_{j \in B} (\hat{V}_j - V_j^{\text{target}})^2 \tag{3.28}$$

Since the value network only outputs one scalar value there is no equivalent clamp loss function for the value network. However, similar to what we did in Section 3.2, we propose modifying the loss function to be the sum of squared errors divided by 2,

$$\mathcal{L}_{\text{SSE}} = \frac{1}{2} \cdot \sum_{j \in B}^{B} (\hat{V}_j - V_j^{\text{target}})^2 \tag{3.29}$$

The justification is the same: this form for the loss function has the same form as the local energy functions of the internal layers of the PCN. Additionally, we propose an alternative form for the loss function of the policy network. Instead of the mean of the policy losses,

$$\mathcal{L}_{\text{MEAN}} \leftarrow -\frac{1}{|B|} \sum_{j \in B} \min \left\{ r_j \hat{A}_j, \mathsf{clip}(r_j, 1-\epsilon, 1+\epsilon) \hat{A}_j \right\} \tag{3.30}$$

we propose the sum of the policy losses scaled by some scalar ρ,

$$\mathcal{L}_{\text{SUM}} \leftarrow -\rho \sum_{j \in B} \min \left\{ r_j \hat{A}_j, \mathsf{clip}(r_j, 1-\epsilon, 1+\epsilon) \hat{A}_j \right\} \tag{3.31}$$

The idea behind doing this is as follows: by scaling the objective function we hope that the norm of the policy loss may be more in line with the norms of the local energy functions of the internal layers of the PCN.

3.4 Comparing Predictive Coding and Backpropagation

For establishing a fair framework for comparing the performance of BP against PC in RL scenarios we need to fix many of the hyperparameters and algorithmic choices outlined in this chapter. Firstly, in all the experiments detailed in Chapter 4 we ensure that the network architecture for the network trained with BP and the network trained with PC are identical. This is so that neither BP or PC has an immediate advantage over the other, because they are using models with exactly the same expressivity.

Additionally, in all our experiments we fix all hyperparameters, other than the learning rate, the optimiser for parameter updates and changes to the form of the loss functions. For example, for both BP and PC, we fix the batch size, discount factor γ and number of steps taken (in the environment), along with all other algorithm specific hyperparameters. This is so that we know that any advantage observed is purely due to the different algorithms, BP and PC, used to compute parameter gradients.

To compare the performance of an algorithm implementing BP for computing parameter gradients versus an algorithm implementing PC for computing parameter gradients, we typically plot the accumulated reward after every episode. The accumulated reward for an episode is calculated simply as,

$$\text{episode reward}(i) = \sum_{t}^{T_i} r_{i,t} \tag{3.32}$$

where $r_{i,t}$ denotes the reward received at time step t during episode i and T_i denotes the length of episode i. We will also record the mean reward calculated after a full run of the algorithm, this is computed simply as the mean episode reward across all episodes,

$$\text{mean reward} = \frac{1}{M} \sum_{i=1}^{M} \text{episode reward}(i) \tag{3.33}$$

where M is the total number of episodes played. For some experiments, those with many episodes played we may instead plot the mean accumulated reward of the last $K \ll M$

episodes. Specifically, we plot the following quantity after every K episodes,

$$\text{mean episode reward}(i) = \frac{1}{K} \sum_{j=i-K}^{i} \sum_{t=1}^{T_j} r_{j,t} \tag{3.34}$$

where $r_{j,t}$ denotes the reward received at time step t during episode j and T_j denotes the length of episode j.

Different runs of the same RL algorithm can have quite high variance since there are many random processes that can affect the learning of the agent, such as, the initial parameters of the network, which are typically sampled from a multivariate Gaussian. The random sampling of experience from the replay buffer and random exploration through the environment also affect an agents ability to learn. As a result, we can't make any concrete conclusions by running BP and PC with the same RL algorithm only once. Since maybe BP got lucky with which seed was chosen, or vice versa. So in many of our experiments we average our results over many seeds. For example, if we ran the same algorithm over N many seeds, then we would plot the episode reward averaged over all seeds,

$$\text{average episode reward}(i) = \frac{1}{N} \sum_{j=1}^{N} \sum_{t}^{T_i^j} r_{i,t}^j \tag{3.35}$$

where $r_{i,t}^j$ denotes the reward received at time step t during episode i on seed j and T_i^j denotes the length of episode i on seed j. The mean reward over all seeds is computed in a similar way,

$$\text{average mean reward} = \frac{1}{N} \sum_{j=1}^{N} \frac{1}{M} \sum_{i=1}^{M} \text{episode reward}^j(i) \tag{3.36}$$

In addition to averaging the plots and results over many seeds, we plot the non-parametric confidence intervals, computed by sampling with replacement. Algorithm 15 outlines this procedure.

Algorithm 15 Computing Bootstrapped Confidence Intervals

Input: episode rewards for all pairs $(i, j) \in M \times N$.
Output: upper confidence interval $\mathbf{u}$, lower confidence interval $\mathbf{l}$.

Initialise vectors $\mathbf{u}, \mathbf{l} \in \mathbb{R}^M$.
for $i \leftarrow 1$; $i < M + 1$; $i \leftarrow i + 1$ **do**
  Initialise vector $\mathbf{b} \in \mathbb{R}^{2000}$.
  for $k \leftarrow 0$; $k < 2000$; $k \leftarrow k + 1$ **do**
    Sample batch B of N indices with replacement from $\{1, ..., N\}$.
    Compute the sample mean:

$$\text{bootstrapped mean} \leftarrow \frac{1}{N} \sum_{j \in B}^{B} \text{episode reward}^j(i)$$

    $b_k \leftarrow$ bootstrapped mean.
  end for
  Let u_i be the 97.5^{th} percentile of $\mathbf{b}$.
  Let l_i be the 2.5^{th} percentile of $\mathbf{b}$.
end for
return upper confidence interval $\mathbf{u}$, lower confidence interval $\mathbf{l}$.

By plotting the confidence intervals we can get a better understanding of how robust and algorithm is, and it provides us with a more sound way of comparing the convergence properties of two different algorithms.

Chapter 4

Experiments and Results

In this chapter we will present the results of several experiments that aim to compare the performance of models trained with backpropagation (BP) and predictive coding (PC) in several common RL tasks. All of our experiments will use the OpenAI `gym` library [46], which provides a comprehensive suite of RL environments and a standard interface for agent-environment interaction. We will start by looking at two simple toy environments: *frozen lake* and *cartpole*. In both these environments we apply the cross entropy method (CEM) and Q-learning with function approximation to obtain promising results which motivate further research.

In the second half of this chapter we scale up to more challenging RL tasks. First we aim to show that deep Q networks (DQNs) trained with PC on Atari 2600 games are able to achieve similar performance to DQNs trained with BP. To do this we reproduced the original DQN algorithm as a standard framework for comparison between the two methods. Unfortunately due to time constraints, we were unable to get promising results on either of the two Atari 2600 games: *Pong* and *Breakout*. Finally, we conclude this chapter by applying PC to policy gradient methods for challenging continuous control tasks. Specifically, we look at two environments from the Multi-Joint dynamics with contact (MuJoCo) test suite provided by OpenAI's `gym` library. In these environments we

compare the proximal policy optimisation (PPO) algorithm modified with the PC learning rule to the standard PPO algorithm that uses BP instead to compute parameter updates. We show that both algorithms maintain similar performance in both these environments which motivates the need for additional research.

4.1 Frozen Lake

Frozen lake is one of the OpenAI `gym` toy text environments, that are designed to be extremely simple, with small discrete state and action spaces. This makes them extremely suitable for obtaining preliminary results and debugging implementations of RL algorithms. In the frozen lake environment the goal is to cross the frozen lake and reach the reward state without falling into any of the holes in the ice. Figure 4.1 illustrates the setup.

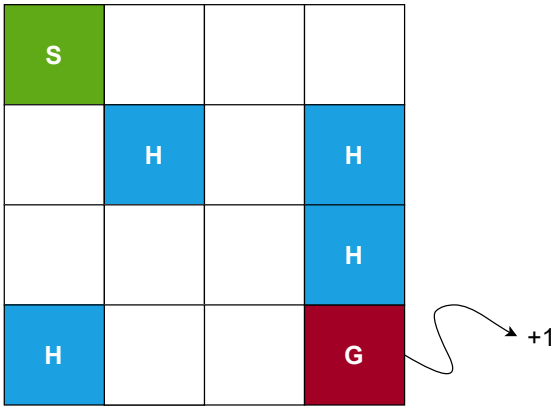

Figure 4.1: Frozen lake environment: S denotes the start state, H denotes the holes in the ice, G denotes the goal state.

Starting from the initial state S the agent may pick one of four actions $\{0, 1, 2, 3\}$, where 0 : move left, 1 : move down, 2 : move right, 3 : move up. Due to the slippery nature of the frozen lake there is some probability p that a random action is taken, rather than the one picked by the agent. In all of our experiments we set $p \leftarrow 0$, so that the transition dynamics of the environment are deterministic, rather than stochastic.

When the agents reaches the goal state G, they receive a reward of $+1$ and the episode

terminates, all other states return a reward of 0, and if the agent falls into a hole in the ice (states denoted H) the episode immediately terminates. At each timestep t the agent receives an observation o_t which corresponds to the agents position in the 4×4 grid, although the agent receives no information about which states are holes and which state is the goal state. The optimal policy π^* is the one that takes the agent from the start state S to the goal state G in as few moves as possible. And so the goal of the agent is to learn the quickest path from S to G that avoids any holes in the ice.

Since the frozen lake environment has a sufficiently small state space $\mathcal{S}$ and action space $\mathcal{A}$, we can easily compute the optimal policy π^* and the optimal state-value function v_* for all states $s \in \mathcal{S}$, using either value iteration or policy iteration. In fact in this very simple scenario we can just use *Dijkstra's shortest path algorithm*, since the optimal value of a state $v_*(s)$ is simply $\gamma^{d^*(s)-1}$, where $d^*(s)$ is the length of the shortest path from s to G.

We will use the optimal policy π^* and the optimal state value function $v_*(s)$ as a means of evaluating the performance of different methods in this section. For value-based methods we will plot the loss of the learned value function $v(s)$ to the true optimal state value function $v_*(s)$. We will also derive a notion of the optimal policy loss, which is the loss of the learned policy π to the optimal policy π^*. By using the optimal value loss and the optimal policy loss we can see exactly how the algorithm converges to the optimal state value function $v_*(s)$ and optimal policy π^*. This helps us diagnose exactly what is going on and establishes a very sound framework for evaluating the convergence of two different RL algorithms.

In the remaining parts of this section we will conduct several experiments comparing the performance of networks trained with BP to networks trained with PC. We will start by applying the cross entropy method (CEM) described in section 3.1 which uses policy networks to represent the learned policy π. We will then describe and present the results of several experiments that utilise Q-networks for approximating the optimal state action

value function $q_*(s, a)$. Along the way we will draw insights and attempt to interpret the results in a way that is consistent with what we know about BP and PC.

4.1.1 Frozen lake with the CEM

In this section we directly apply the CEM (Algorithm 12) to frozen lake. Before we dive into the experiments, first we explain how the optimal policy π^* and optimal state value function $v_*(s)$ are computed. We run value iteration to recover the optimal state action value function $q_*(s, a)$. We then construct the optimal policy π^* as follows,

$$\pi^*(a \mid s) = \begin{cases} \frac{1}{|A(s)^*|} & \text{if } a \in A(s)^* \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

$$\text{where } A(s)^* = \{a : q_*(s, a) = \max_{a'} q_*(s, a')\} \tag{4.2}$$

which gives us the stochastic policy illustrated by Figure 4.2.

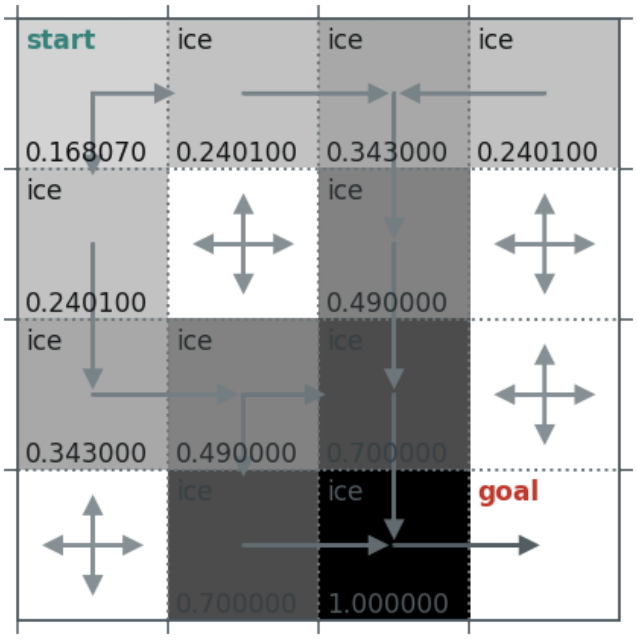

Figure 4.2: Frozen lake environment solved with value iteration, $\gamma = 0.7$. Optimal state values $v_*(s)$ are given, the optimal policy computed using Equation 4.1 is denoted by the arrows.

We are now equipped to run the CEM. To evaluate the performance of a run we plot the average batch reward during training, see Section 3.1. We also plot the cross entropy loss of the current policy π to the optimal policy π^*,

$$\mathcal{L}^*_{policy} = -\frac{1}{|\mathcal{S}|}\sum_{s\in\mathcal{S}}[\pi^*(s)]^T \log \pi(s) \tag{4.3}$$

where $\pi^*(s)$ and $\log \pi(s)$ are vectors over the action space $\mathcal{A}$. We represent the policy π with a network consisting of one hidden layer with dimension $n = 128$. ReLU activations are used after the input layer and the hidden layer and softmax is used at the output layer. In all our experiments we set $\rho_{\text{CE}} \leftarrow 0.7$, so the bottom 30^{th} percentile of episodes are thrown away to retain the "elite" batch B. We also set $N \leftarrow 100$, where N is the number of episodes we play with current policy π before a batch update. Finally, we set the discount parameter $\gamma \leftarrow 0.9$.

We conduct experiments to compare the performance of the policy network trained with BP every batch update or with PC every batch update. For the PC update step we fix the number of inference steps $T \leftarrow 128$, the inference learning rate $\alpha \leftarrow 0.1$. We use SGD for both inference and parameter update steps. We perform a fairly exhaustive search over the SGD learning rate parameter η. Specifically, we tried $\eta \in \{0.6, 0.8, 1.0, 1.2, 1.4, 1.6\}$ for both the network trained with BP and PC. The best learning rate we found for BP was $\eta_{\text{BP}} = 1.0$ and for PC $\eta_{\text{PC}} = 1.2$. Figure 4.3 presents the batch reward and policy loss for the policy network trained with BP and PC.

Both BP and PC have very similar performance in this setup. This is probably because the frozen lake environment is very easy to solve and doesn't provide enough complexity for us to differentiate between the two learning rules. Nevertheless, these results are promising and at the very least, they show that PC can be applied to policy networks that don't utilise squared loss at the output layer.

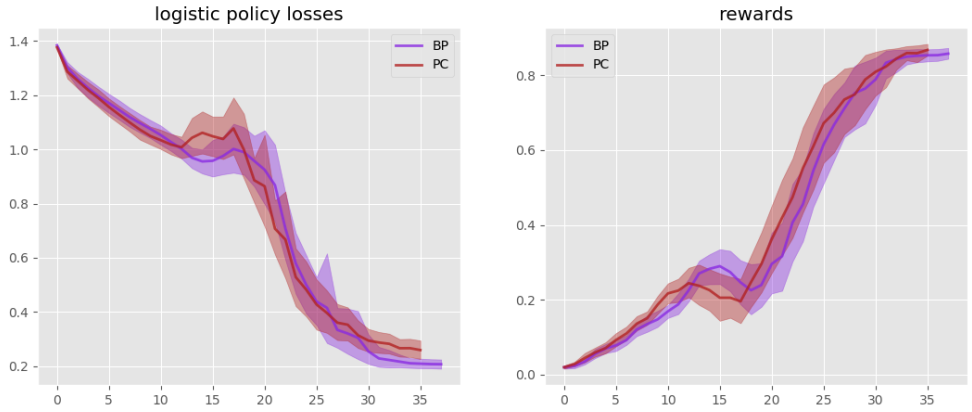

Figure 4.3: Frozen lake environment solved with the CEM. Network trained with BP (**purple**), network trained with PC (**red**). Policy loss computed with equation 4.3, rewards computed with 3.5. x-axis corresponds to the number of batch iterations.

4.1.2 Frozen lake with Q-learning

In this section we directly apply the deep Q-learning algorithm to frozen lake, see Algorithm 13. Because Q-networks give us a deterministic policy we need to modify the evaluation criteria first. As before, we run value iteration to compute the optimal state-action value function $q_*(s, a)$, and we compute optimal policy π^* using Equation 4.1. During training we plot the MSE loss of the current value function $v(s)$ against the true optimal state value function $v_*(s) = \max_a q^*(s, a)$,

$$\mathcal{L}^*_{\text{value}} = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} (v(s) - v_*(s))^2 \tag{4.4}$$

We also plot the 0-1 loss of the learned policy π against the optimal policy π^*,

$$\mathcal{L}^*_{\text{policy}} = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \mathbb{1}[\pi(s)^T \pi^*(s) = 0] \tag{4.5}$$

where $\pi^*(s)$ and $\pi(s)$ are probability vectors. Since π is deterministic the vector $\pi(s)$ will be a one-hot encoding, so $\mathcal{L}^*_{\text{policy}} = 0$ means that for all $s \in \mathcal{S}$ the deterministic policy π is pointing in at least one of the same directions as the optimal policy π^* in Figure 4.2.

In our experiments the Q-function is represented by a network with one hidden layer with dimension $n = 128$. ReLU activations are used after the input layer and the hidden layer, the last layer is linear and it computes a linear combination of the activations of the hidden layer. Let the output of the network be the vector $Q(s, \cdot)$. The policy is an epsilon greedy policy: with probability ε the agent picks an action uniformly at random from $\mathcal{A}$, and with probability $1 - \varepsilon$ the agent pick an action according to $\text{argmax}_a Q(s, a)$. The epsilon parameter ε is decayed over time according to the following schedule,

$$\varepsilon \leftarrow 0.01 + (1.0 - 0.01) * \exp\left(\frac{-k}{500}\right) \tag{4.6}$$

where k is the number of steps taken in the environment (across all episodes). We set the target network sync frequency to $N \leftarrow 10$, the batch size is set to 128, the discount factor $\gamma \leftarrow 0.7$ and the replay buffer size is set to 1000. Again we conduct experiments to compare the performance of the Q-network trained with BP or with PC every batch update. For the PC update step we modify the hyperparameters slightly in line with insights from Song *et al.* [28]: the number of inference steps $T \leftarrow 32$, the inference learning rate $\alpha \leftarrow 0.05$. The clamp loss targets, see Section 3.2.1, are also used for PC in these experiments. Again, we use SGD for both inference and parameter update steps. For both BP and PC we perform a search over the learning rate parameter η: we tried $\eta \in \{0.005, 0.01, 0.02, 0.03, 0.04, 0.05\}$. The best learning rate parameter η for both BP and PC proved to be $\eta_{\text{BP}}, \eta_{\text{PC}} \leftarrow 0.03$. Figure 4.4 presents the 0-1 policy loss and the value loss during training for the Q-network trained with BP and PC.

Once again both BP and PC demonstrate very similar performance. We note that there appears to be an interesting artefact in both the policy and value loss plots. The value loss and policy loss are stationary for the first ≈ 150 frames. This is an artefact of the environment, the agent only starts learning anything useful after it has seen its first reward. This means the agent must first reach the goal state before and information about the other states can be recovered. Once the goal state has been reached the reward signal is

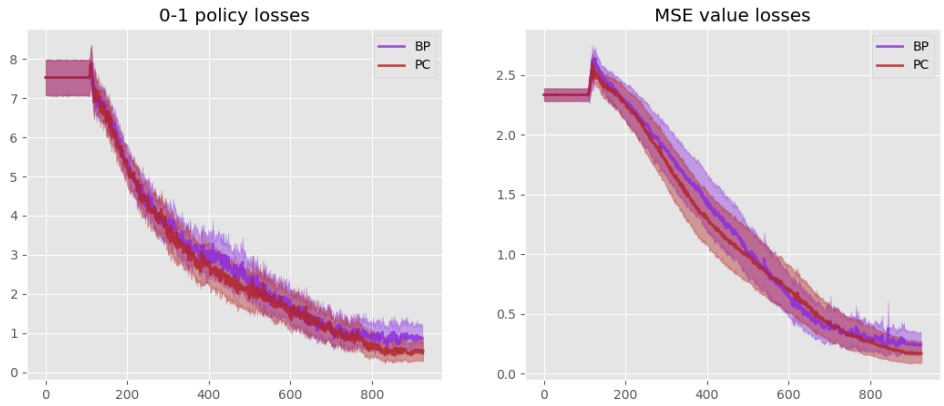

Figure 4.4: Frozen lake environment solved with the Q-network. Network trained with BP (**purple**), network trained with PC (**red**). 0-1 policy loss computed with equation 4.5, value loss computed with 4.4. x-axis corresponds to the number of interactions with the environment.

propagated to all the other visited states one step at a time, by unrolling the Bellman equation during the Q-learning update step. Reaching the goal state for the first time is a random event we cannot control, and if the agent gets unlucky this will affect the agent's ability to learn anything.

Although, this isn't the only problem, we still need to visit every state and try every action to learn the optimal state values $v_*(s)$. If every tuple corresponding to every state-action pair $(s, a) \in \mathcal{S} \times \mathcal{A}$ is in the replay buffer then we will converge to the optimal state value function $v_*(s)$, since we will update every state-action pair infinitely often, this is the main requirement for the convergence guarantee of Q-learning. If we haven't visited every state-action pair and we only follow the greedy policy (since ε has decayed to 0.01), then we may never visit certain states and can't hope to recover the optimal state values for these states. These insights should motivate the experiments we present in following sections.

4.1.3 Q-learning with Hand-crafted Replay Buffer

A recurring problem in RL is the exploration-exploitation trade-off. In the experiments so far, it appears there is no significant advantage of training networks with PC over BP, or vice versa. One reason for this could be because exploration of the environment is the limiting factor holding both these learning algorithms back. So any advantage in using PC over BP (or vice versa) is "absorbed" by the fact that the agent is restricted by how efficiently it is exploring the environment. However, what if we give the agent everything it needs to learn?

Since the frozen lake environment is a sufficiently small finite MDP, we can construct a replay buffer with hand crafted experience for the agent to learn on. We propose the following scenario, we will construct a replay buffer with every possible tuple $\langle s, a, s', r, d \rangle$, for $s, s' \in \mathcal{S}$, $a \in \mathcal{A}$, $r \in \mathcal{R}$, $d \in \{0, 1\}$ and train the networks on batches sampled from this replay buffer. By doing this we give the agent everything it needs to know to recover the optimal state value function q^*, and so it no longer needs to explore and interact with the environment, instead it can simply learn off the replay buffer and propagate the reward signal obtained at the goal state to all other states. Algorithm 16 outlines the procedure for constructing this replay buffer.

Algorithm 16 Constructing the Hand-Crafted Replay Buffer

Initialise: empty replay buffer $B \leftarrow \varnothing$
Output: full replay buffer B

```
for all s ∈ S do
    for all a ∈ A do
        pick action a from state s.
        receive reward r, observe new state s' and done flag d.
        append tuple ⟨s, a, s', r, d⟩ to replay buffer B.
    end for
end for
return the replay buffer B
```

It is important to note that this doesn't actually reduce the problem to a supervised learning task, since we don't actually provide the optimal state values $v_*(s)$ as targets.

Rather, the agent is provided with the outcome of every state action pair $(s, a) \in \mathcal{S} \times \mathcal{A}$ and must still learn to infer the optimal state values $v_*(s)$. Although, since we will update every state-action pair $(s, a) \in \mathcal{S} \times \mathcal{A}$ infinitely often, we are guaranteed to converge to the optimal state value function $v_*(s)$.

For these experiments we use exactly the same network architecture as in Section 4.1.2. The target network sync frequency is set to $N \leftarrow 10$, the batch size is set to the size of the replay buffer (for the 4×4 frozen lake environment this is 64), the discount factor is set to $\gamma \leftarrow 0.7$. For the PC update step we set: the number of inference steps $T \leftarrow 128$, the inference learning rate $\alpha \leftarrow 0.1$. SGD is used for both the inference and parameter update steps. For both networks trained with BP and PC we tried learning rates $\eta \in \{0.02, 0.04, 0.06, 0.08, 0.1\}$, the best learning rate we found for BP was $\eta_{\text{BP}} = 0.06$ and for PC $\eta_{\text{PC}} = 0.08$. To demonstrate that the clamp loss targets (see Section 3.2.1) provide an empirical advantage over the normal double Q-learning targets, we first present the results of these experiments run without the clamp loss targets. Figure 4.5 presents the 0-1 policy loss and the value loss during training for the network trained with both BP and PC (without clamp loss).

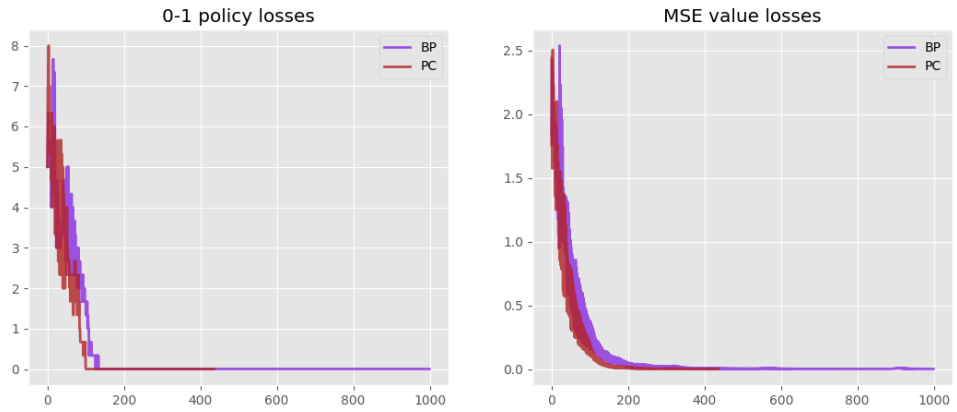

Figure 4.5: Frozen lake environment solved with the hand-crafted replay buffer and Q-network. Network trained with BP (**purple**), network trained with PC (**red**). 0-1 policy loss computed with Equation 4.5, value loss computed with 4.4. x-axis corresponds to the number of batch iterations. The plots are averaged over 3 seeds.

Here we see that the the network converges much faster than the generic Q-learning algorithm with exploration. After the network reaches $\mathcal{L}^*_{\text{policy}} = 0$ and $\mathcal{L}^*_{\text{value}} \leq 10^{-3}$ we consider the problem solved. The average solve time for PC in this setting was ~ 399, whereas the average solve time for BP was ~ 835. Additionally, the mean policy loss and value loss for PC was ~ 0.1 and ~ 0.027 respectively, whereas for BP the mean policy loss and value loss was ~ 0.37 and ~ 0.036 respectively. These results establish a small advantage using PC over BP in this setting. We believe this advantage is attributed to the fact that networks trained with PC suffer less from interference and sporadic weight updates [28]. In the next section we run some additional experiments to try and observe the "interference" in this setting.

Now we give the results of using the clamp loss targets instead of the generic double Q-learning targets. The setup of these experiments is exactly the same as described earlier. We tried learning rates $\eta \in \{0.02, 0.04, 0.06, 0.08, 0.1\}$ and found that the best learning rate for PC is the same, that is, $\eta_{\text{PC}} = 0.08$ (for BP it is also still the same, with $\eta_{\text{BP}} = 0.06$, since the loss computed at the output layer is the same). In addition, we also ran experiments on the 8×8 grid version of frozen lake. For these experiments all other hyperparameters are the same, except we set the batch size to 256 (the size of the replay buffer), and we tried learning rates $\eta \in \{0.008, 0.01, 0.012, 0.014, 0.015, 0.016, 0.018, 0.02\}$. For PC the best learning rate that we found was $\eta_{\text{PC}} = 0.015$, and for BP the best learning rate the we found was $\eta_{\text{PC}} = 0.01$. Notice that the best learning rates for the 8×8 version are roughly four times smaller than for the 4×4 version, this is because the loss function is summing over 4 times more targets than before. Figure 4.6 presents the results of using the clamp loss targets instead of the standard targets.

These results illustrate that PC has a very clear advantage in terms of stability and convergence speed for both the 4×4 and 8×8 versions of frozen lake. Both the policy loss and value loss converge much more smoothly when the network is trained with PC. Table 4.1 gives the corresponding average solve times, mean policy losses and mean value losses for each of the experiments.

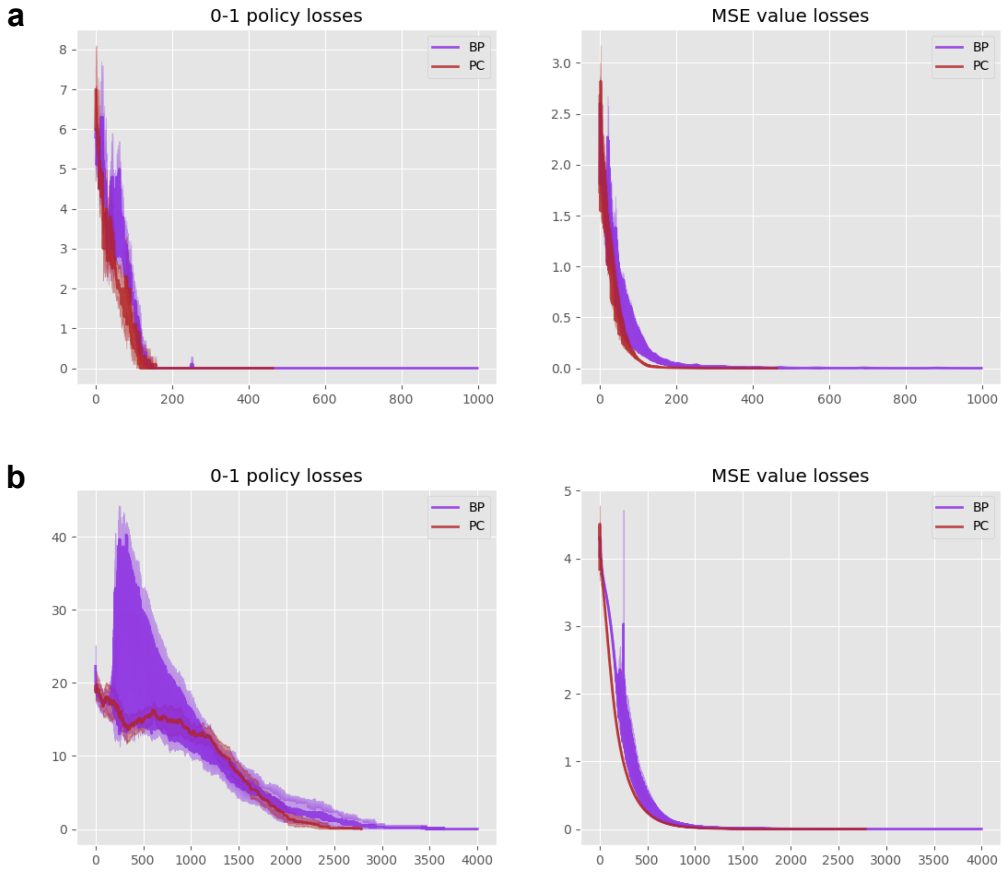

Figure 4.6: Frozen lake environment solved with the hand-crafted replay buffer and Q-network. Network trained with BP (**purple**), network trained with PC (**red**) and clamp loss $\mathcal{L}_{\text{clamp}}$. 0-1 policy loss computed with Equation 4.5, value loss computed with Equation 4.4. x-axis corresponds to the number of batch iterations. The plots are averaged over 10 seeds. **a:** frozen lake 4×4. **b:** frozen lake 8×8.

Algorithm	**Average Solve Time**	**Mean Policy Loss**	**Mean Value Loss**
BP (4×4)	~ 820	~ 0.40	~ 0.011
PC (4×4)	$\sim$ **300**	$\sim$ **0.030**	$\sim$ **0.0080**
BP (8×8)	~ 3100	~ 7.2	~ 0.026
PC (8×8)	$\sim$ **2300**	$\sim$ **0.60**	$\sim$ **0.019**

Table 4.1: Average solve time, mean 0-1 policy loss and mean MSE value loss of BP and PC on frozen lake 4×4 and 8×8.

In the 8×8 experiment the network trained with PC maintains a fairly consistent policy as it converges to the optimal state value function $v_*(s)$, whereas with BP the policy appears to fluctuate rapidly. This is likely caused by the interference associated with BP batch

updates and points to the apparent undesirable property of BP: networks trained with BP suffer more from interference and sporadic weight updates. This effect is more pronounced in the bigger 8×8 scenario because states far away from the goal state have smaller absolute differences between their optimal state values $v_*(s)$. Because of the argmax in the policy, small changes to the parameters of the Q-network can result in vastly different policies, this is likely why the policy loss appears to fluctuate so much even though the value loss converges quite smoothly for BP. Interestingly, the Q-network trained with PC doesn't seem to have the same problem. Further research is needed to verify these insights and the cause of this fluctuating policy loss, but we note that these results provide good motivation for using the clamp loss and scaling up to bigger problems.

4.1.4 Observing Interference (Q-learning with Hand-crafted Replay Buffer)

In this section we conduct some additional experiments to try and uncover the interference associated with weight updates in networks trained with both BP and PC. Let the replay buffer be denoted B. After a batch update with batch $B_{\text{half}} \subset B$ we define absolute interference to be,

$$I(B_{\text{half}}) = \sum_{(s,a)\in B\setminus B_{\text{half}}} |Q^{\text{new}}(s,a) - Q^{\text{old}}(s,a)| \tag{4.7}$$

where $Q^{\text{new}}(s,a)$ is the Q-value of action a from state s of the Q-network after the batch update, and $Q^{\text{old}}(s,a)$ is the Q-value of action a from state s before the batch update. In words, the interference is the absolute difference between Q-values of the of the state action pairs not present in the sampled batch, computed before and after the batch update. The experiments are setup as follows:

- Same network architecture as in Section 4.1.2 and 4.1.3.
- Sync frequency $N \leftarrow 10$.
- Batch size is 32 for frozen lake 4×4 and 128 for frozen lake 8×8 (exactly half the replay buffer size in both cases).

- Discount factor $\gamma \leftarrow 0.7$.
- Number of inference steps $T \leftarrow 128$.
- Inference learning rate $\alpha \leftarrow 0.1$.
- Learning rates set as the best configuration: $\eta_{\text{BP}} = 0.1$ and $\eta_{\text{PC}} = 0.16$ ($\eta_{\text{BP}} = 0.03$ and $\eta_{\text{PC}} = 0.05$ for 8×8 frozen lake).
- Learning rate search for frozen lake 4×4: $\eta \in \{0.1, 0.12, 0.14, 0.18, 0.2\}$.
- Learning rate search for frozen lake 8×8: $\eta \in \{0.01, 0.02, 0.03, 0.04, 0.05\}$.
- Use the clamp loss function (see Section 3.2.1).

Figure 4.8 presents the 0-1 policy loss and the value loss during training for the network trained with both BP and PC. Notice how the plots are more noisy because of the random sampling of experience from the replay buffer.

The plots in Figure 4.8 coincide with our results in the previous section. Since only half of the examples in the replay buffer are used every batch update, we can plot the interferences calculated by Equation 4.7. Figure 4.7 plots the interference calculated every batch update during training.

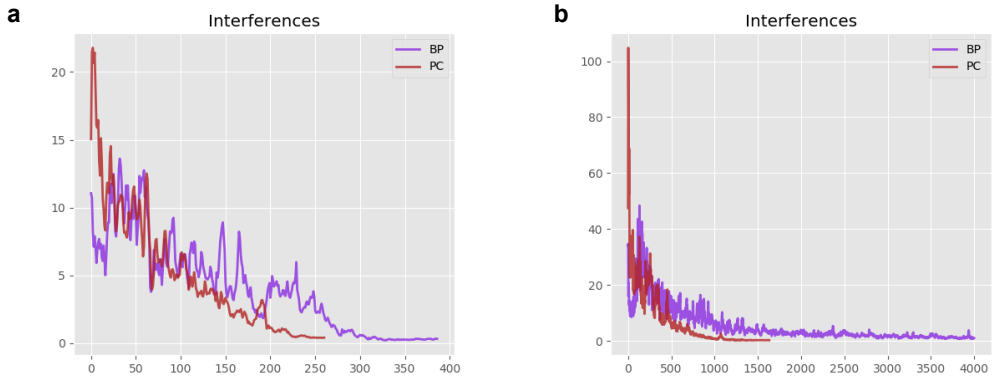

Figure 4.7: Interference plots calculated by Equation 4.7 every batch update. Network trained with BP (**purple**), network trained with PC (**red**). x-axis corresponds to the number of batch iterations. The plots compute the mean interference averaged over 10 seeds. **a:** frozen lake 4×4. **b:** frozen lake 8×8.

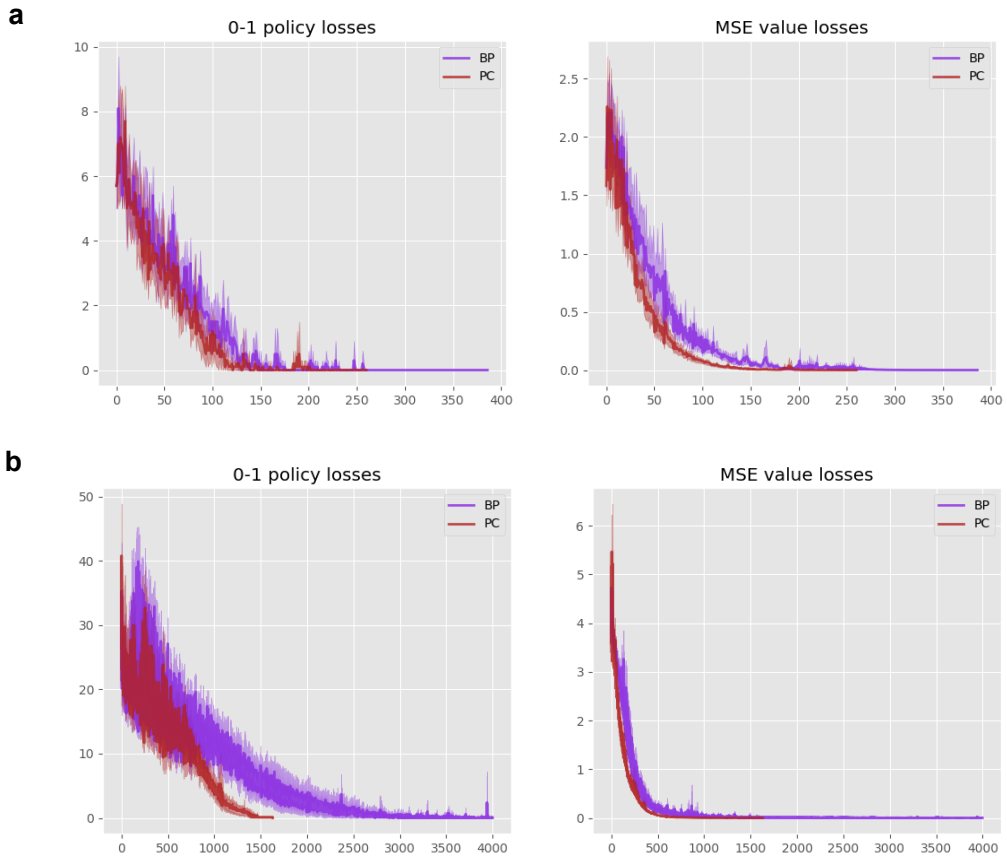

Figure 4.8: Frozen lake environment solved with the hand-crafted replay buffer and Q-network. Network trained with BP (**purple**), network trained with PC (**red**). 0-1 policy loss computed with equation 4.5, value loss computed with 4.4. x-axis corresponds to the number of batch iterations. The plots are averaged over 10 seeds. **a:** frozen lake 4×4 with PC solved in average time ~ 230, with mean policy loss ~ 0.10 and mean value loss ~ 0.021, with BP solved in average time ~ 320, with mean policy loss ~ 0.36 and mean value loss ~ 0.030. **b:** frozen lake 8×8 with PC solved in average time ~ 1400, with mean policy loss ~ 0.39 and mean value loss ~ 0.013, with BP solved in average time ~ 3700, with mean policy loss ~ 6.3 and mean value loss ~ 0.020.

The interference plots (Figure 4.7) give us some insight into what is going on. It appears there is high interference with PC at the start of training when compared with BP. This could correspond to large parameter updates. The interference with PC shrinks as it converges to the optimal state value function $v_*(s)$, whereas with BP, the interference decays more slowly. This could simply be due to the fact PC is converging quicker better. Its hard to interpret these plots, perhaps we could devise a better notion of interference?

Nevertheless, these plots provide some food for thought, but understanding the empirical differences between these two learning algorithms remains an important direction for future research.

4.1.5 Supervised Learning

In this section we present the results of some simple supervised learning experiments on the frozen lake environment. Instead of constructing a replay buffer of tuples, we construct a dataset of labelled examples and frame the frozen lake problem as a purely supervised learning task. By doing this we hope to uncover whether there are any empirical differences between networks trained with BP and PC in this simpler paradigm. We pose the following questions. Are the advantages of PC still apparent in this new paradigm? Or does PC just seem to have smoother convergence when applied to RL problems?

Once again, we approximate the Q-function with a network consisting of one hidden layer with dimension $n = 128$. ReLU activations are used after the input layer and the hidden layer, and the last layer is linear. We don't need a target network anymore since we are provided with labels. We set the batch size to 16 (or 64 for the 8×8 version), the discount factor is set to $\gamma \leftarrow 0.7$. For the PC update step we set: the number of inference steps $T \leftarrow 128$, the inference learning rate $\alpha \leftarrow 0.1$. We use SGD for running inference and we use the clamp loss function at the output layer. For both BP and PC we tried SGD with learning rates $\eta \in \{0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1\}$, and we tried the Adam optimiser with learning rate $\eta = 0.01$. Algorithm 17 outlines how we construct the dataset.

The problem is now a pure regression problem, where the input features are states $s \in \mathcal{S}$ and the output targets $Q(s, \cdot)$ are vectors of state-action values. Once again we plot the optimal value loss (see Equation 4.4) and the 0-1 policy loss (see Equation 4.5) during training. The Adam optimiser with learning rate $\eta = 0.01$ achieved the quickest convergence for both BP and PC, this is expected since Adam is known for very quick convergence. With SGD, the best learning rate for BP was $\eta_{\text{BP}} = 0.05$ and for PC it was

Algorithm 17 Constructing the Supervised Learning Dataset

Initialise: empty dataset $\mathcal{D} \leftarrow \varnothing$
Output: full dataset $\mathcal{D}$

```
Run value iteration (algorithm 7) to recover q_*(s, a).
for all s ∈ S do
    initialise Q-vector: Q ← 0 ∈ ℝ^|A|.
    for all a ∈ A do
        Q_a ← q_*(s, a)
    end for
    append example ⟨s, Q⟩ to the dataset D.
end for
return the replay buffer B.
```

$\eta_{\text{BP}} = 0.07$. We note that the best learning rates we determined by which one achieved the lowest mean value loss and not how smooth the learning curve looks. Figure 4.9 presents the results of these experiments.

The Adam plot is omitted here because it is uninteresting and both networks trained with BP and PC converge very fast since this is a very straightforward problem. For the Adam plot and bigger versions of the plots in Figure 4.9 please refer to Appendix B.1.

Once again we see that the policy loss fluctuates much more for the network trained with BP as opposed to PC, even the value loss appears to be sporadic for BP as well. This shows that the same issues with BP are apparent in both the supervised learning and RL. Although, we must note that the Adam optimiser alleviates these issues entirely in this

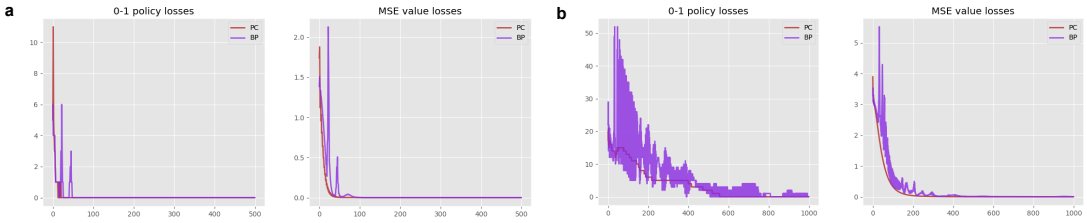

Figure 4.9: Frozen lake as a supervised learning task solved with Q-network. Network trained with BP (**purple**), network trained with PC (**red**) and clamp loss $\mathcal{L}_{\text{clamp}}$. 0-1 policy loss computed with equation 4.5, value loss computed with 4.4. x-axis corresponds to the number of batch iterations. The plots are averaged over 10 seeds. **a:** frozen lake 4×4. **b:** frozen lake 8×8.

setting (see Appendix B.1). Nevertheless, the results in this section are promising and motivate further research on harder RL environments. However, before we scale-up we will continue with another different toy environment to gather more insight into how PC operates in the RL paradigm.

4.2 Cart Pole

Cart pole is part of the classic control suite provided by OpenAI's `gym` library. These environments are considered some of the easiest ones to solve, although we can no longer use dynamic programming methods since the state space of cart pole is not finite. In the previous section we saw that PC and BP achieved comparable performance on the frozen lake environment. This was the case for both the CEM and Q-learning. In this section we show that in the cart pole environment, Q-learning modified with the PC learning rule outperforms the standard Q-learning algorithm with BP.

Before we present these results we will first describe the cart pole environment, which is remarkably similar to a problem outlined by Barto *et al.* in an early RL paper [94]. A pole is attached by a rigid joint to a cart that moves along a frictionless track. The pole starts in a upright position and the goal is to balance the pole upright by moving the cart left or right. Figure 4.10 illustrates the problem.

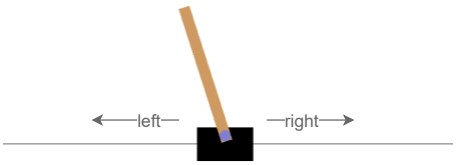

Figure 4.10: Cart pole environment: balance the pole upright by moving the cart left or right.

At each discrete timestep t the agent receives an observation o_t of four real numbers which correspond to the quantities outlined in Table 4.2. The agent then picks an action a_t in $\{0, 1\}$, where 0 corresponds to pushing the cart left with some fixed force, and 1

corresponds to pushing the cart right with the same fixed force. At every timestep the agent receives a reward of +1. The episode terminates if any of the following conditions occurs:

- Pole angle is greater than ± 12.
- Cart position is greater than ± 2.4.
- Episode length exceeds 200 (or 500 for `CartPole-v1`).

Formally the goal of the agent is to pick actions $a \in \{0, 1\}$, such that the quantity $\mathbb{E}[T]$ is maximised, where $T \leq 200$ is a random variable denoting the length of an episode. All randomness is captured by the policy $\pi(a \,|\, s)$ and the initial start state S_0; state transitions are deterministic given the action $a \in \{0, 1\}$ since the dynamics of the environment are Newtonian. Before we present the experiments that train Q-networks to solve the cart pole environment, we will first conduct some experiments that apply the CEM to this problem.

Num	**Observation**	**Min**	**Max**
0	Cart Position	-4.8	4.8
1	Cart Velocity	$-\infty$	∞
2	Pole Angle	$-24°$	$24°$
3	Pole Angular Velocity	$-\infty$	∞

Table 4.2: Observation space shape of cart pole [46]

4.2.1 Cart pole with the CEM

Unlike frozen lake, the cart pole environment has an infinite state space, so we won't consider the optimal policy loss or optimal value loss like we did in Section 4.1. Instead we will plot the average batch reward after every batch iteration and use the mean reward as a means of comparing BP and PC.

The policy π is represented by a network of one hidden layer with dimension $n = 128$. ReLU activations are used after the input layer and the hidden layer and softmax is used at the output layer. In all our experiments we set $\rho_{\text{CE}} \leftarrow 0.3$, so only the top 30^{th} percentile

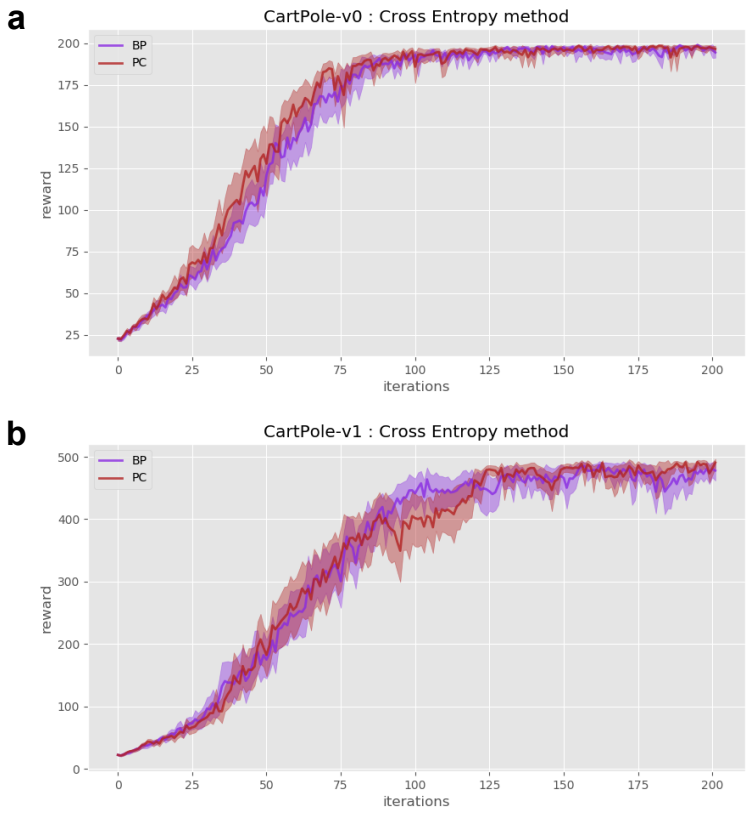

Figure 4.11: Cart pole environment solved with the CEM. Network trained with BP (**purple**), network trained with PC (**red**). Average batch reward computed with Equation 3.5. The plots are averaged over 25 seeds. **a:** `CartPole-v0`, episode ends at step 200. **b:** `CartPole-v1`, episode ends at step 500.

of episodes are retained for the "elite" batch B. We also set $N \leftarrow 16$, where N is the number of episodes we play with current policy π before a batch update. Finally, we set the discount parameter $\gamma \leftarrow 1.0$. For each run we do 200 batch iterations.

With these hyperparameters we ran the CEM on cart pole and compared the performance of the policy network trained with BP every batch update or with PC every batch update. For the PC update steps we fixed: the number of inference steps $T \leftarrow 128$, and the inference learning rate $\alpha \leftarrow 0.1$. SGD was used for both inference and parameter updates. For both BP and PC we tried the following learning rates: $\eta \in \{0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0\}$. We also ran one experiment where both the PC network and the network trained with BP used the Adam optimiser for parameter updates, with learning rate $\eta = 0.01$.

We ran the CEM on `CartPole-v0` where the episodes terminate after 200 steps, and on `CartPole-v1` where the episodes terminate after 500 steps. For `CartPole-v0` the best learning rates we found were: $\eta_{\text{BP}} = 1.0$ for BP and $\eta_{\text{PC}} = 1.2$ for PC. For `CartPole-v1` the best learning rates we found were both $\eta_{\text{BP}} = \eta_{\text{PC}} = 1.8$ for BP and PC. Figure 4.11 presents the batch rewards of the CEM with BP and PC, for both cart pole environments.

Again we see that both BP and PC have very similar performance when they are applied to the CEM. Once again this could be because the cart pole environment is relatively easy to solve. Interestingly, when we use the Adam optimiser for the parameter updates of the network, BP maintains good performance, but PC appears to fail and diverge at a certain point. For the reward plots of these experiments please refer to Figure B.3 in Appendix B.2. Table 4.3 presents the mean rewards of all the experiments.

Algorithm	**Mean Reward**
BP (v0) SGD $\eta = 1.0$	~ 155
PC (v0) SGD $\eta = 1.2$	$\sim$ **159**
BP (v1) SGD $\eta = 1.8$	$\sim$ **336**
PC (v1) SGD $\eta = 1.8$	~ 335
BP (v0) Adam $\eta = 0.01$	$\sim$ **157**
PC (v0) Adam $\eta = 0.01$	~ 143
BP (v1) Adam $\eta = 0.01$	$\sim$ **387**
PC (v1) Adam $\eta = 0.01$	~ 301

Table 4.3: Mean rewards for the CEM experiments on `CartPole-v0` and `CartPole-v1`.

These results give some initial indication that Adam and PC are not well suited for each other. While it is well understood how PC and SGD interact, further research is needed to identify what exactly is going on when we use PC with Adam. Adam is the go to optimiser for BP, perhaps there is a different optimiser better suited to PC?

4.2.2 Cart pole with Q-learning

In this section we apply the deep Q-learning algorithm (Algorithm 13) to the `CartPole-v1` environment. Once again we will not consider the optimal policy loss or optimal value

loss. Instead we will plot the accumulated reward after every episode, and we will use the mean reward over all episodes as a means of comparing the performance of different algorithms.

In all our experiments we set the total number of episodes played to $M \leftarrow 100$. For the Q-network we utilise the same architecture as we did for frozen lake: one hidden layer with dimension $n = 128$, ReLU activations are used after the input layer and the hidden layer, and the last layer is linear. Once again, the policy is epsilon greedy. The epsilon parameter ε is decayed over time according to the same schedule,

$$\varepsilon \leftarrow 0.01 + (1.0 - 0.01) * \exp\left(\frac{-k}{500}\right) \tag{4.8}$$

where k is the number of steps taken in the environment (across all episodes). We set the target network sync frequency to $N \leftarrow 100$, the batch size is set to 64, the discount factor is set to $\gamma \leftarrow 0.98$, and the replay buffer size is set to 5000. The PC inference parameters are set to: $T \leftarrow 32$, $\alpha \leftarrow 0.05$ and SGD is used for inference update steps. We also use the original double Q-learning targets (see Equation 3.10) instead of the clamped targets (see Equation 3.13), because the PC network trained with the clamp loss targets did not achieve quite as good convergence. For both PC and BP we tried Adam with $\eta = 0.001$ and SGD with learning rates: $\eta \in \{0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05\}$. For BP, Adam with $\eta = 0.001$ performed the best, and with SGD the best learning rate was $\eta_{\text{BP}} = 0.0005$. For PC, parameter updates with SGD outperformed Adam with $\eta = 0.001$, the best learning rate for SGD was $\eta_{\text{PC}} = 0.01$, this configuration outperformed both BP with Adam and SGD. The Adam plots are omitted from the main body of text, instead please refer to Figure B.4 in Appendix B.3. Figure 4.12 presents the reward plots of the Q-networks trained with both BP and PC for `CartPole-v1`. Table 4.4 also presents the mean rewards of these experiments.

These results are very promising and it is interesting to see that PC networks trained with SGD outperform networks trained with BP and Adam in this setting. It is important to

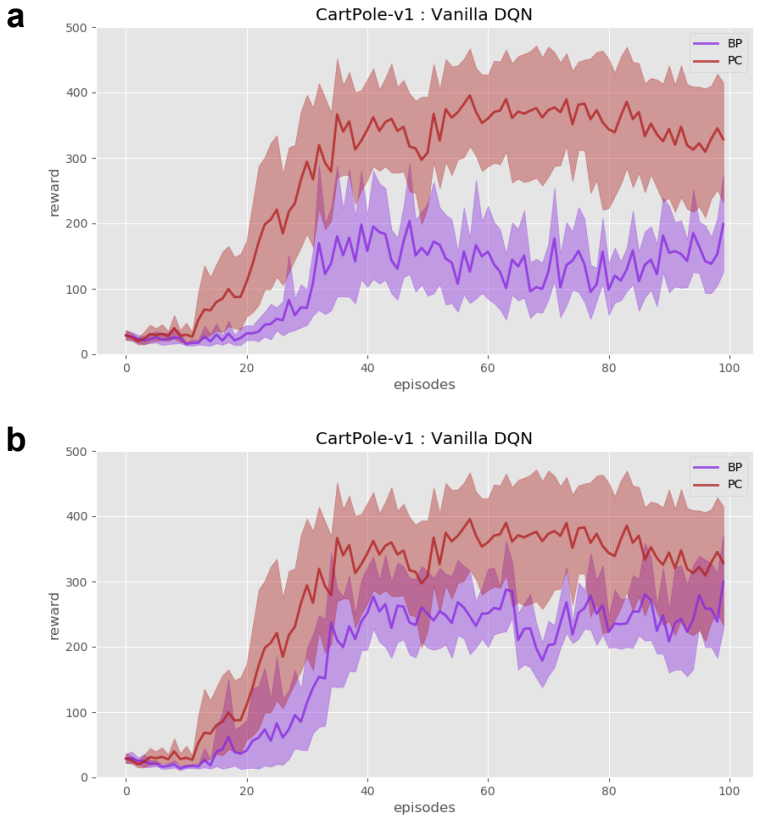

Figure 4.12: `CartPole-v1` environment solved with Q-network. Network trained with BP (**purple**), network trained with PC (**red**). Episode reward computed with Equation 3.32. The plots are averaged over 10 seeds. **a:** both PC network and BP network trained with SGD. **b:** PC network trained with SGD, BP network trained with Adam.

Algorithm	**Mean Reward**
BP (v1) SGD $\eta = 0.0005$	~ 111
PC (v1) SGD $\eta = 0.01$	$\sim$ **272**
BP (v1) Adam $\eta = 0.01$	$\sim$ **178**
PC (v1) Adam $\eta = 0.01$	~ 145

Table 4.4: Mean reward for Q-learning experiments on `CartPole-v1`.

note that PC with Adam still appears to diverge and fail in this setting, see Appendix B.3. This further supports the challenges and important research directions that we remarked upon in the previous sections. Regardless, these results are still quite promising and motivate further research, scaling up, and applying PC networks to harder and different RL problems.

4.3 The Atari Benchmark

OpenAI `gym` [46] provides us with a standard interface for evaluating RL algorithms in the *arcade learning environment* (ALE) [44]. The ALE was developed to act as a benchmark for research domains such as RL, world model learning and planning, imitation learning, representation learning, transfer learning and curiosity driven learning. As such the ALE provides AI researchers with a rigorous test bet for domain independent AI systems. Specifically, the ALE consists of hundreds different Atari 2600 game environments, each one is designed to have its own challenges to overcome.

Due to time constraints we will only look at two different Atari 2600 games, namely, *Pong* and *Breakout*. The easier of the two games Pong, is a bit like tennis; each player moves their paddle to hit the ball back to their opponent, the player who misses the ball and breaks the rally looses a point and their opponent gains a point, the first player to reach a score of 21 wins the game. One of the players is computer controlled and the other is moved by the player or agent interacting with the game. Figure 4.13 shows a game of Pong being played on an Atari 2600 screen.

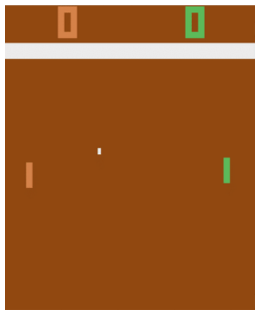

Figure 4.13: Atari 2600 Pong environment [46].

A reward of $+1$ is received when the computer misses the ball and the agent gains a point, and a reward of -1 is received when the agent missed the ball and the computer gains a point. Learning in Pong can be quite sporadic and random, as soon as the agent learns to move the paddle to the location where the ball is expected to land, the episode reward shoots up from around -10 to $+20$ very dramatically. This situation is a bit like frozen

lake, as soon as we see reward we very quickly converge the optimal policy. This makes Pong quite an easy Atari 2600 game to solve, but unfortunately it doesn't provide us with a very sound framework for comparison. Fortunately Breakout is a more challenging Atari 2600 game and typically we see that the learned policy is improved more incrementally than dramatically.

Breakout actually has similar dynamics to Pong: the player controls a paddle that hits a ball into a brick wall. When the ball comes into contact with a brick the brick is destroyed and the goal is to destroy all the bricks. In fact good players will aim to breakthrough the brick wall and let the ball destroy the bricks from the other side as the ball bounces along the edge of the screen. A reward is received if the player destroys a brick, the exact amount of reward received depends on the colour of the brick, however, in all our experiments we clamp the reward signals to the range $[-1, +1]$ (see Section 3.2 or [19] for justification).

Additionally, the player has 5 lives, if the player misses the ball with their paddle they loose one life and after loosing 5 lives the episode terminates. We modify the scenario such that the player only has one life, and episodes correspond to playing with just one life, this modification helps with stability during training [19]. We also terminate the episodes after 10000 frames, the idea of this is to encourage the agent to accumulate reward more quickly by breaking through the wall. Figure 4.14 shows a game of Breakout being played on an Atari 2600 screen.

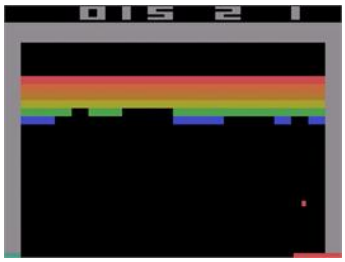

Figure 4.14: Atari 2600 Breakout environment [46].

The reason learning in Breakout happens in a more incremental fashion is because: (1)

the agent must learn to move the paddle where the ball is going to land, (2) the agent must position the paddle in such a way so that it hits a brick of better colour, (3) the agent must learn to breakthrough the wall to the other side to accumulate reward more quickly. Of course breaking through the wall is quite a long term goal and reward signals received for doing so may take a while to propagate to earlier actions which helped break the wall.

In both Pong and Breakout the agent receives observations o_t at each timestep t, which correspond to the 210×160 pixel RGB image displayed on the Atari 2600 screen. In all our experiments the observations are scaled down to an 84×84 grey scale image, every 4 frames are skipped, and the past 4 frames (those not skipped) are concatenated together and passed to the agent. This means the agent learns from high dimensional tensors with shape $4 \times 84 \times 84$ which correspond to 4 stacked grey scale images. This is exactly what was done in the original Atari paper [19]. Futhermore, the pixel values of the grey scale images are scaled down to the range $[0, 1]$, this corresponds to dividing each pixel value by 255.

In general there are 18 distinct actions available when playing Atari 2600 games, however for Pong and Breakout some of these actions are equivalent, due to the nature of these games. Table 4.5 outlines the action spaces for both Pong and Breakout, we should note that the size of the action space must corresponds to the dimension of the output layer of any network used to solve these games.

Pong		**Breakout**	
Num	*Action*	*Num*	*Action*
0	NOOP (no action)	0	NOOP (no action)
1	FIRE	1	FIRE
2	RIGHT	2	RIGHT
3	LEFT	3	LEFT
4	FIRE RIGHT		
5	FIRE LEFT		

Table 4.5: Action space of Atari 2600 Pong [46]

In all of our experiments we also use the `NoFrameskip` and `Deterministic` options when setting up the game environments. This means there is no random frame skipping (2-5 frames are skipped at random every step) and no sticky actions (the previous action is repeated with probability 0.25). For further details of the different options please refer to [46].

We also looked at the equivalent RAM versions of both the Pong and Breakout environments. Instead of a 210×160 pixel RGB image, the agent receives 128 bytes of RAM corresponding to the state of the Atari 2600 console. This observation is converted to a vector of 128 integers (between in the range [0, 255]), once again the values are scaled down to the range $[0, 1]$. We no longer use frame stacking since temporal information should be encoded in the RAM. The architectural details of the networks will be left for later, but note that the RAM observations are lower dimensional and can processed with an MLP rather than a CNN.

In the remaining parts of this section we present the results of various experiments applied to Pong and Breakout. The hyperparameters used in all our experiments reflect those used to obtain early results with DQNs (trained with BP) on Atari 2600 games [19, 20, 87]. While we were able to successfully reproduce these results, unfortunately the same networks trained with PC appeared to diverge at certain points during training. We believe this to due to the Adam optimiser and the much larger replay buffers required to train DQNs for Atari 2600 games. Nevertheless, we are still able to draw insight from these experiments and determine the most important immediate directions for future research.

4.3.1 Results on Pong

In this section we present the results of applying the deep Q-learning algorithm (Algorithm 13) to Pong. To compare the performance of DQNs trained with BP and PC we plot the mean accumulated reward of the last 10 episodes. We also use the mean reward as a means of quantifying the performance of specific algorithmic choices.

In all of our experiments we ran the algorithms for exactly 1M step. For the pixel version of Pong we used a DQN consisting of 3 convolutional layers and 2 linear layers, all with ReLU activations, except the final output layer. This architecture mimics the one used in the original DQN paper [19], for precise details please refer to Appendix C.2. For the RAM version of Pong, we used a Q-network consisting of one hidden layer with dimension $n = 128$, ReLU is used after the input layer and the hidden layer, and the final output layer is linear (this is exactly the same architecture used in the previous sections). Again we use an epsilon greedy exploration strategy, with the following the epsilon decay schedule,

$$\varepsilon \leftarrow 0.01 + (1.0 - 0.01) * \exp\left(\frac{-k}{15000}\right) \tag{4.9}$$

where k is the number of steps taken in the environment (across all episodes). The target network sync frequency is set to $N \leftarrow 1000$, the batch size is set to 32, the discount factor is set to $\gamma \leftarrow 0.99$, and the replay buffer size is set to 100000. The number of PC inference steps is set to $T \leftarrow 32$, the inference learning rate is set to $\alpha \leftarrow 0.05$. SGD is used for inference steps, and in all our experiments we use Adam for parameter updates. We use Adam instead of SGD because SGD is much too slow for these kind of problems and Adam significantly speeds up convergence. However, even with Adam, training is still quite sensitive to learning rates, and so we tried learning rates $\eta \in \{0.0005, 0.0001, 0.00005, 0.00001, 0.000005, 0.000001\}$. For PC we also tried both the clamp loss targets (see Equation 3.14) and the standard double Q-learning targets (see Equation 3.10). Additionally, for both BP and PC we also tried the normal Q-learning targets computed with the target network,

$$y \leftarrow R + \gamma \operatorname*{argmax}_a \widehat{Q}(S', a) \tag{4.10}$$

Table 4.6 presents the best learning rates and mean reward scores for each of these algorithmic choices. Figure 4.15 presents the episode reward plots for the pixel and RAM versions of Pong.

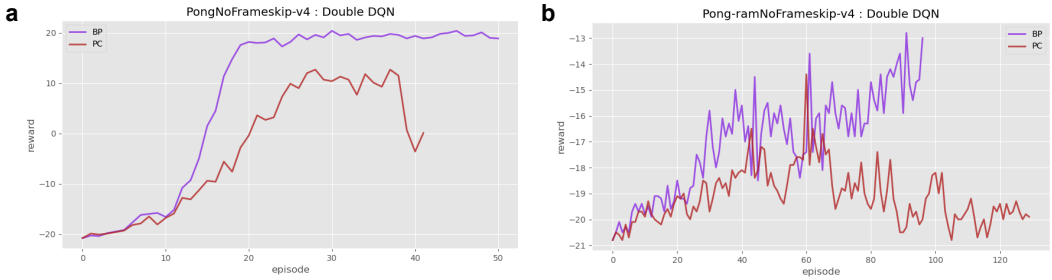

Figure 4.15: Atari 2600 Pong solved with deep Q-network. Network trained with BP (**purple**), network trained with PC (**red**). Episode reward computed with Equation 3.34. Only one seed run. **a:** `PongNoFrameskip-v4` environment solved with convolutional network. **b:** `Pong-ramNoFrameskip-v4` environment solved with MLP-style network.

Environment	Algorithm	Learning Rate	Mean Reward
`PongNoFrameskip-v4`	Double Q-learning BP	0.00001	**7.88**
	Double Q-learning PC (Clamp)	0.00001	−6.71
	Double Q-learning PC (No Clamp)	0.00001	*-3.14*
	Single Q-learning BP	0.00001	1.31
	Single Q-learning PC (No Clamp)	0.00001	−19.64
`Pong-ramNoFrameskip-v4`	Double Q-learning BP	0.0001	**−17.00**
	Double Q-learning PC (Clamp)	0.0001	−19.17
	Double Q-learning PC (No Clamp)	0.0001	−19.11
	Single Q-learning BP	0.0001	−17.00
	Single Q-learning PC (No Clamp)	0.0001	*-18.33*

Table 4.6: Mean reward and optimal learning rate for various algorithmic choices of the deep Q-learning algorithm (Algorithm 13) applied to Atari 2600 Pong. Bold and italic configurations presented in Figure 4.15.

Clearly in both the pixel and RAM versions of Pong the DQNs trained with BP achieve better performance than those trained with PC. In both plots presented in figure 4.15, the PC networks appears to falter, and in the case of the RAM version it diverges after a certain amount of training. The reason for this could be attributed to the mismatch between PC and Adam, that we have seen in previous experiments with Q-networks and policy networks. Additionally, we speculate that this divergence could be caused due to the large replay buffers used to provide experience to learn from. The purpose of these replay buffers is to deal with "catastrophic interference" or "catastrophic forgetting", although PC posits as a remedy for this. Using PC and large replay buffers may be "overkill" and both solutions could be interfering with each other in some way that we

don't understand.

Another thing to consider is the lack of hyperparameter tuning. Other than the learning rate, the hyperparameter choices were made we all based on what works well for DQNs trained with BP. In effect all the other hyperparameters were optimised for BP. Perhaps with a more thorough hyperparameter search we could have obtained better performance with PC. However, the choice of which hyperparameters to tune and change comes with increased understanding of how PC interacts with Adam and other similar optimisers, and what exactly is going wrong to cause divergence during training in the first place.

One final remark is that, perhaps the original deep Q-learning algorithm [19] is not suited for networks trained with PC, maybe we need to revisit the design of the algorithm and the original choices made by Mnih *et al.*, in order come up with a novel algorithm that better facilitates learning with PC in this setting.

4.3.2 Results on Breakout

Now we present the results on Breakout. Once again we plot the mean accumulated reward, except since episodes are much shorter we plot the mean accumulated reward of the last 100 episodes.

In all our experiments we played 5M steps in the environment. For the pixel version of Breakout we used the same convolutional architecture outlined in the previous section and detailed in Appendix C.2. And for the RAM version of Breakout we used the same MLP architecture: one hidden layer with dimension $n = 128$, ReLU activations used after the input and hidden layers. Since Breakout is a harder game than Pong we also modified the algorithm in a few different ways:

- The first 50000 steps are played with a completely random policy.
- No parameter updates are made during the first 50000 steps.
- Parameter updates from batches of experience are only made after every 4 steps (rather than after every step).

- The epsilon decay schedule is modified accordingly,

$$\varepsilon \leftarrow 0.1 + (1.0 - 0.1) * \exp\left(\frac{-k}{1000000}\right) \tag{4.11}$$

Additionally, we set the target network sync frequency to $N \leftarrow 10000$, the batch size is set to 32, the discount factor is set to $\gamma \leftarrow 0.99$, and the replay buffer size is set to 1000000. The PC inference parameters remain the same: number of inference steps $T \leftarrow 32$, inference learning rate $\alpha \leftarrow 0.05$. Again, SGD is used for inference update steps and Adam is used for all parameter update steps. Once again, we tried learning rates $\eta \in \{0.0005, 0.0001, 0.00005, 0.00001, 0.000005, 0.000001\}$, and we tried the clamped double Q-learning targets (see Equation 3.13), the original double Q-learning targets (see Equation 3.10), and the normal Q-learning targets (see Equation 4.10). Table 4.7 presents the best learning rates and mean reward scores for each experiment. Figure 4.16 presents the episode reward plots for the pixel and RAM versions of Breakout.

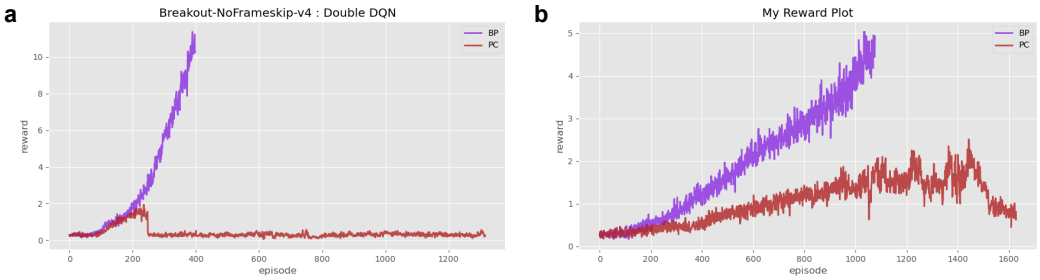

Figure 4.16: Atari 2600 Breakout solved with deep Q-network. Network trained with BP (**purple**), network trained with PC (**red**). Episode reward computed with Equation 3.34. Only one seed run. **a:** `BreakoutNoFrameskip-v4` environment solved with convolutional network. **b:** `Breakout-ramNoFrameskip-v4` environment solved with MLP-style network.

Once again we see that BP outperforms PC in both the pixel and RAM versions of Breakout. In this scenario the divergence of the DQN trained with PC appears to be more spectacular than with Pong. Its likely that some of the same reasons outlined in the previous section have played a part in this: the mismatch between Adam and PC, the larger replay buffer (1M instead of 100K steps are stored) and the lack of an exhaustive hyperparameter search.

Environment	Algorithm	Learning Rate	Mean Reward
`BreakoutNoFrameskip-v4`	Double Q-learning BP	0.00001	3.18
	Double Q-learning PC (Clamp)	0.00001	0.32
	Double Q-learning PC (No Clamp)	0.00001	*0.39*
	Single Q-learning BP	0.00001	**3.33**
	Single Q-learning PC (No Clamp)	0.00001	0.29
`Breakout-ramNoFrameskip-v4`	Double Q-learning BP	0.00005	1.93
	Double Q-learning PC (Clamp)	0.00005	0.82
	Double Q-learning PC (No Clamp)	0.00005	*1.04*
	Single Q-learning BP	0.00005	**2.01**
	Single Q-learning PC (No Clamp)	0.00005	0.90

Table 4.7: Mean reward and optimal learning rate for various algorithmic choices of the deep Q-learning algorithm (Algorithm 13) applied to Atari 2600 Breakout. Bold and italic configurations presented in Figure 4.16.

The reason we see an even more catastrophic failure is unknown, perhaps the fact that Breakout is more challenging results in a more significant failure. Either way, the results presented in this section point us towards important directions for future research. Understanding the interplay between PC and Adam is paramount, and by gaining further insight into what is going wrong here, we can hope to better tune the hyperparameters and modify the algorithm to accommodate better learning with PC.

4.4 Continuous Control

In this section we will look at two continuous control problems that are part of the MuJoCo test suite provided by OpenAI `gym` [46]. In total there are ten MuJoCo environments, and these are thought to be some of the most challenging environments to solve. Specifically, we will look at the `Ant` and `HalfCheetah` environments. Both these environments have stochastic initial states so the agent can't simply remember the best sequence of actions to take, instead the agent must generalise well to the state space.

In all of our experiments we will use a policy network that follows the continuous action space formulation of the policy gradient method extended to the multivariate Gaussian distribution case, for details please refer to Section 2.3.5. The policy network or actor

consists of one hidden layer with dimension $n = 64$, the input, hidden and output layers are each followed by tanh activations. This network models the mean of a multivariate Gaussian with dimension equal to the dimension of the action space. The covariance matrix is modelled by a single learnable diagonal matrix $\boldsymbol{\Lambda}$ (only diagonal elements are learnable parameters) rather than a deep network.

The policy network is trained with the PPO algorithm, see Section 3.3, which means we also train a separate value network or critic. In all our experiments the value network consists of one hidden layer with dimension $n = 64$, ReLU activations are used after the output layer and hidden layer, the output layer is linear, outputting a single scalar state value $\hat{v}(s)$.

In our experiments we compare the performance of the standard PPO algorithm that uses BP to compute parameter gradients versus a modified version of the PPO algorithm that uses PC instead to update both the policy and value networks. Before we present the results of our experiments we wil first give a brief overview of both the `Ant` and `HalfCheetah` environment.

Ant: the `Ant` environment was introduced in the A2C paper by Schulman *et al.* [93]. The ant is a 3D robot with one torso and four moveable legs with two joints. The goal is move the ant forward by applying the right torques to the eight hinges: four connecting the legs to the torso and four connecting the two separate parts of the legs. Figure 4.17 illustrates this setup.

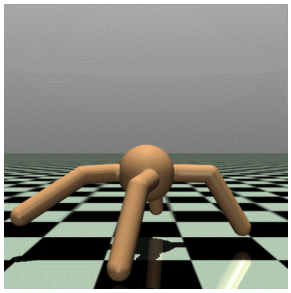

Figure 4.17: MuJoCo `Ant` environment [46]

The action space of the environment is continuous, it corresponds to picking torque val-

ues for each of the eight joints. The observation space is also continuous, it consists of positional and angular values for different parts of the ant's body, along with velocities for each of these parts. Table 4.8 defines the action space and observation space more formally, for further details on what each value corresponds to please refer to [46]. The

Action Space:	$[-1+1]^8$
Observation Space:	$[-\infty, \infty]^{26}$

Table 4.8: Action and observation space of the MuJoCo `Ant` environment [46]

reward dynamics of the environment are as follows:

- Every timestep the agent receives a reward of +1.
- The agent receives a reward for moving forward computed by,

$$R^{\text{forward}} = \frac{x_t - x_{t+1}}{dt} \tag{4.12}$$

  where x_t is the x-coordinate of the ant at time step t and x_{t+1} is the x-coordinate of the ant at time step $t+1$, dt is the time between frames (default $dt \leftarrow 0.05$).

- The agent receives a negative reward that penalises the agent if it takes actions that are too large, calculated by,

$$R^{\text{action}} = 0.5 \cdot \mathbf{a}^T \mathbf{a} \tag{4.13}$$

  where $\mathbf{a}$ is the action vector output by the policy network.

- The agent receives another negative reward that penalises the agent if any external forces are too large, calculated by,

$$R^{\text{external}} = 0.5 \cdot 0.001 \cdot \sum(\mathsf{clip}(\text{external force, -1, +1}))^2 \tag{4.14}$$

  where the sum is over all points of contact with the ground.

- Total reward received is at each timestep is: $R_t = 1 + R_t^{\text{forward}} - R_t^{\text{action}} - R_t^{\text{external}}$.

The episode terminates if any of the following conditions are satisfied:

- The y-orientation (the 2^{nd} element of the observation) is not in the range $[0.2, 1.0]$, or in words, the ant has flipped or fallen over.
- Any of the state space values are no longer finite
- Episode length exceeds 1000.

Half Cheetah: the `HalfCheetah` environment is based on work by Wawrzyński [95]. The half cheetah is a 2D robot with consisting of nine parts with eight joints linking them. The goal is to apply the right torque to the joints such that the cheetah runs as quickly as possible to the right, reward is received for how far the cheetah moves during an episode. Torque can only be applied to 6 of the joints since the torso and head of the cheetah are fixed. Figure 4.18 illustrates the problem.

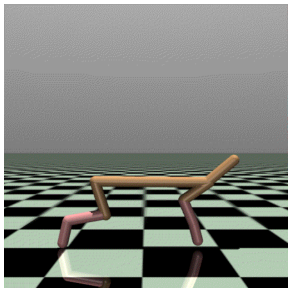

Figure 4.18: MuJoCo `HalfCheetah` environment [46]

The action space corresponds to picking torque values for each of the six moveable joints. The observation space consists of positional and angular values for different parts of the cheetah, along with velocities for each of these parts. Table 4.9 formally describes the action and observation space, once again, for further details on what each value corresponds to please refer to [46].

Action Space:	$[-1 + 1]^6$
Observation Space:	$[-\infty, \infty]^{16}$

Table 4.9: Action and observation space of the MuJoCo `HalfCheetah` environment [46]

The reward dynamics of the environment are as follows:

- The agent receives a reward for moving forward computed by,

$$R^{\text{forward}} = \frac{x_t - x_{t+1}}{dt} \tag{4.15}$$

  where x_t is the x-coordinate of the cheetah at time step t and x_{t+1} is the x-coordinate of the cheetah at time step $t+1$, dt is the time between frames (default $dt \leftarrow 0.05$).

- Negative reward that penalises the agent if it takes actions that are too large, calculated by,

$$R^{\text{action}} = 0.1 \cdot \mathbf{a}^T \mathbf{a} \tag{4.16}$$

  where $\mathbf{a}$ is the action vector output by the policy network.

- Total reward received is at each timestep is: $R_t = R_t^{\text{forward}} - R_t^{\text{action}}$.

The episode terminates if any of the following conditions are satisfied:

- Any of the state space values are no longer finite
- Episode length exceeds 1000.

4.4.1 Results on the MuJoCo Benchmark

Now we present the results of applying the PPO algorithm detailed in Section 3.3 to both the `Ant` and `HalfCheetah` environments. To compare the original algorithm that utilises BP for parameter updates to the variant that uses PC for parameter updates we plot the average episode reward for the last 100 episodes during training. We also compute the mean reward after each run, as a means of comparing specific algorithmic choices.

For each experiment we ran the PPO algorithm for for exactly 5M steps. It is important to note that 5M steps is actually rather short, typically the PPO algorithm with BP converges after anywhere between 20-30M steps. The reason we only run the experiments for 5M

steps is for practical reasons: PPO with PC runs approximately ten times slower than BP (because we run inference every parameter update step), therefore it is not feasible to conduct a thorough analysis while running the PPO algorithm with PC for 20-30M steps since it would take a couple weeks for one run. Therefore, we must run the experiments for only 5M frames before terminating; this should give us an idea of good hyperparameters to use for scaling up to 20-30M frames in the future, and the results will act as a useful proof of concept.

In all of our experiments we fix the following hyperparameters for both BP and PC:

- The discount factor $\gamma \leftarrow 0.99$.
- The generalised advantage estimation [93] discount parameter $\lambda \leftarrow 0.95$.
- The sampled trajectory length $T \leftarrow 2049$.
- The number of batch updates per epoch $N \leftarrow 10$.
- The batch size $|B|$ is set to 64.
- The epsilon clipping parameter for the surrogate objective function (see Equation 3.21 in Section 2.3.5) is set to $\epsilon \leftarrow 0.2$.

For the PC inference parameters we set: the number of inference steps $T \leftarrow 32$, and the inference learning rate $\alpha \leftarrow 0.05$. SGD is used for inference steps, and once again Adam is used for parameter updates in all of our experiments. Again, the reason for this is because Adam dramatically speeds up the convergence of training, so that we may feasibly run experiments and draw conclusions in a suitable amount of time. For the value network we set the learning rate parameter to $\eta_{\text{critic}} \leftarrow 0.0001$, and for the policy network we set the learning rate parameter to $\eta_{\text{actor}} \leftarrow 0.00001$. In preliminary runs it became clear that this configuration was the optimal configuration, since with other learning rates the algorithm either dramatically failed or converged very slowly.

In addition to using the standard loss functions used in Algorithm 14 in Section 3.3. We ran experiments with the different loss functions, $\mathcal{L}_{\text{SSE}}$ and $\mathcal{L}_{\text{SUM}}$, described in Section

Environment	Algorithm	Policy Loss	Value Loss	Mean Reward
`Ant`	BP	Mean of $J(\boldsymbol{\theta})$	MSE	1105.04
	BP	Mean of $J(\boldsymbol{\theta})$ + entropy ($\beta = 0.001$)	MSE	**1145.47**
	PC	Mean of $J(\boldsymbol{\theta})$	MSE	1038.19
	PC	Mean of $J(\boldsymbol{\theta})$ + entropy ($\beta = 0.001$)	MSE	*1060.53*
	PC	Sum of $\rho \cdot J(\boldsymbol{\theta})$ ($\rho = 0.05$)	MSE	671.94
	PC	Sum of $\rho \cdot J(\boldsymbol{\theta})$ ($\rho = 0.05$) + entropy ($\beta = 0.001$)	MSE	640.53
	PC	Sum of $\rho \cdot J(\boldsymbol{\theta})$ ($\rho = 0.05$)	SSE	660.26
	PC	Sum of $\rho \cdot J(\boldsymbol{\theta})$ ($\rho = 0.05$) + entropy ($\beta = 0.001$)	SSE	731.95
`HalfCheetah`	BP	Mean of $J(\boldsymbol{\theta})$	MSE	**989.90**
	BP	Mean of $J(\boldsymbol{\theta})$ + entropy ($\beta = 0.001$)	MSE	862.61
	PC	Mean of $J(\boldsymbol{\theta})$	MSE	740.17
	PC	Mean of $J(\boldsymbol{\theta})$ + entropy ($\beta = 0.001$)	MSE	813.28
	PC	Sum of $\rho \cdot J(\boldsymbol{\theta})$ ($\rho = 0.05$)	MSE	503.03
	PC	Sum of $\rho \cdot J(\boldsymbol{\theta})$ ($\rho = 0.05$) + entropy ($\beta = 0.001$)	MSE	672.91
	PC	Sum of $\rho \cdot J(\boldsymbol{\theta})$ ($\rho = 0.05$)	SSE	783.20
	PC	Sum of $\rho \cdot J(\boldsymbol{\theta})$ ($\rho = 0.05$) + entropy ($\beta = 0.001$)	SSE	*879.51*

Table 4.10: Mean reward for variants of the PPO algorithm (Algorithm 14) applied to `AntBulletEnv-v0` and `HalfCheetahBulletEnv-v0`. Bold and italic configurations presented in figure 4.19.

3.3.1. We tried scaling factors $\rho \in \{100, 10, 5, 2, 1, 0.5, 0.1, 0.05, 0.01, 0.001\}$ for the sum of the policy losses, and in all cases the best configuration we found was $\rho \leftarrow 0.05$. Table 4.10 presents the mean reward scores for each experiment. Figure 4.19 presents the episode reward plots for BP and PC in the `Ant` and `HalfCheetah` environments. For larger versions of the plots presented in Figure 4.19 please refer to appendix B.4.

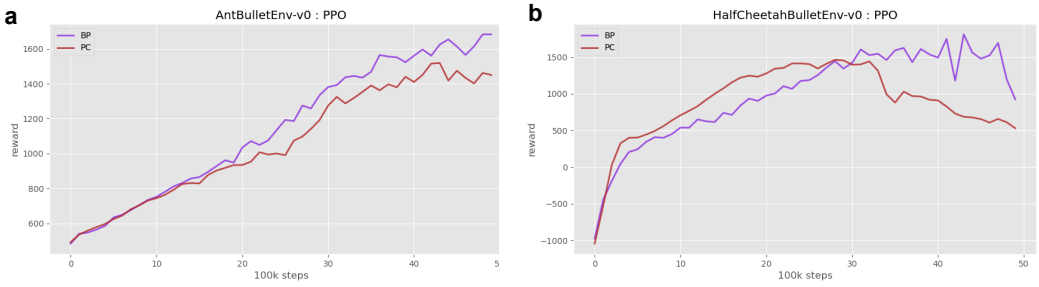

Figure 4.19: **a:** `AntBulletEnv-v0` solved with PPO. **b:** `HalfCheetahtBulletEnv-v0` solved with PPO. Actor and Critic trained with BP (**purple**), Actor and Critic trained with PC (**red**). Episode reward computed with Equation 3.34. Only one seed run.

By studying the plots in Figure 4.19 and the results in table 4.10 its clear that PC and

BP achieve much more comparable performance in this setting, compared to the results on Atari 2600 games. However, these results do initially suggest that BP may be slightly better than PC in this setting, although this small advantage may become less apparent if we ran these experiments over multiple seeds.

The `Ant` plots in Figure 4.19 for BP and PC look remarkably similar. This is likely due to the fact that we used the same loss functions for both BP and PC; the best configuration was to use the mean of $J(\boldsymbol{\theta})$ to train the policy network and the MSE loss to train the value network, along with an entropy bonus with $\beta = 0.001$. When we used the SSE loss and/or the sum of $J(\boldsymbol{\theta})$ with $\rho = 0.05$ (see Section 3.3.1), the reward appeared to shoot up quickly and then diverge quite drastically, which is why we get low mean reward scores for these configurations.

In contrast to the `Ant` environment the best configurations for the `HalfCheetah` environment differed between BP and PC; for BP the best configuration was to use the mean of $J(\boldsymbol{\theta})$ to train the policy networks, the MSE loss to train the value network and no entropy bonus, for PC the best configuration was to use the sum of $J(\boldsymbol{\theta})$ with $\rho = 0.05$ to train the policy networks, the SSE loss to train the value network and entropy bonus with $\beta = 0.001$. Interestingly, PC appears to initially converge faster than BP, although both diverge somewhat towards the end of the 5M frames. This indicates the need for more hyperparameter tuning for both algorithms. Stronger clipping of the ratios, for example setting $\epsilon \leftarrow 0.1$, or a stronger entropy coefficient β may prevent these divergences.

Nevertheless, these results are significantly more promising than the ones obtained on the Atari benchmark. Policy gradient methods offer a much more natural view of learning and policy optimisation in MDPs, than the more artificial deep Q-learning procedures. The PPO algorithm is also on-policy, meaning the agent only ever learns from the most immediate experience generated with the current policy. As such, learning with PPO is done in a more online fashion, without large replay buffers and perhaps this suits PC better.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis, we applied the theory of predictive coding (PC) [33–35] to a number of reinforcement learning (RL) tasks. The main approach in RL is to use function approximators like neural networks to represent learned policies or value functions which are typically optimised with backpropagation (BP) [12] and gradient descent to maximise the accumulated reward in a given environment. Instead, we trained these function approximators with PC and developed a framework for directly comparing the performance of BP and PC in RL tasks. Specifically, we ran the cross-entropy method (CEM) [79], Q-learning [75], and the proximal policy optimisation (PPO) algorithm [45] on a range of different environments, from toy scenarios to Atari 2600 games and challenging continuous control problems. To maintain fairness in all our experiments we fixed almost of all the hyperparameters to the same values for both BP and PC, and we only modified the learning rates and the output loss functions of the networks. Finally, to compare BP and PC we plotted reward curves (or learning curves) of both algorithms against each other and computed the mean reward (or mean loss).

The results we obtained on the toy scenarios, cart pole and frozen lake, are promising

and in all cases the networks trained with PC achieved comparable or better performance than the networks trained with BP. The only caveat is that SGD was used to obtain all of these results and PC with Adam appeared to diverge or fail in some of these scenarios. Even so, in these scenarios we conclude that PC with SGD appears to have an empirical advantage over BP in terms of stability and convergence to the optimal policy or state-value function.

Unfortunately the results we obtained on Atari 2600 games were disappointing. In all cases the networks trained with PC appear to falter and diverge at certain points during training, whereas the networks trained with BP consistently improved during training. Regardless, these results provided us with some good insight, which helped us determine some of the important directions for future work highlighted in the upcoming section.

We obtained much more promising results on two continuous control problems from the Multi-Joint dynamics with contact (MuJoCo) benchmark [46]. Specifically, we demonstrated that policy and value networks trained with PC and PPO obtain comparable performance to policy and value networks trained with the standard PPO algorithm, that uses BP for computing parameter gradients. There are two main takeaways here, PC is still effective when the output loss function doesn't exactly match the local energy functions, and perhaps PC is better suited to online settings without large replay buffers that retain old experience.

All things considered, applying PC to deep learning systems remains a promising area of research. We note that there is no obvious advantage in using PC over BP in many of the scenarios in this thesis, although this is probably a symptom of the many research gaps associated with how PC operates in deep learning systems. Popularising PC is not an easy task, since for the past decade, BP has been the predominant algorithm for training deep networks. However, with further research and the development of specialised hardware implementing PC several orders of magnitude faster than BP [42], we could soon see a shift towards PC and other biologically plausible algorithms.

5.2 Future Work

In this section we note some of the most important areas for further research, not only for the work in this thesis but for the problem of applying PC to deep learning systems as a whole.

5.2.1 Quantifying Interference

Recent results from Song *et al.* [28] provide compelling evidence that PC suffers less from interference and makes parameter updates that result in neural activity more aligned with the training target. In our experiments presented in Section 4.1.4 we tried to quantify this interference by defining a natural notion of interference. The interference plots were not particularly telling and perhaps by using a different notion of interference we can uncover where the advantage in using PC comes from in this setting.

5.2.2 Understanding PC with Adam

In all of our toy experiments we see that PC performs better or no worse than BP when SGD is used for parameter updates. However, when Adam is used for parameter updates the networks trained with PC appear to diverge during training, whereas BP maintains good performance. The reason Adam is so popular is because it works remarkably well with BP. Our results suggest that there is some apparent mismatch between Adam and PC. A key research challenge is to understand the connection between Adam and PC and why they seem to be fighting against each other. It is also important to understand why PC diverges when Adam is used for parameter updates, in the scenarios presented in this thesis. These insights can help us modify existing optimisers or devise an alternative optimiser better suited for PC.

5.2.3 Exploring the Effect of Hyperparameters

Since PC runs approximately ten times slower than BP it is more difficult to do a thorough hyperparameter search for the harder environments, like Atari 2600 games and complex

continuous control tasks. The toy experiments had fewer hyperparameters and we were able to tune everything with preliminary runs before proceeding with a learning rate search. With increased understanding of how PC operates in deep learning systems, we can hope to get better results by choosing the hyperparameters more appropriately. This better understanding comes from both theoretical analyses and well documented trial-and-error style experiments. Both types of research are important for motivating the adoption of PC in deep learning systems.

5.2.4 Replay Buffers and Online Learning

Replay buffers pose as a very artificial fix to the problem of "catastrophic interference" or "catastrophic forgetting". Understanding whether large replay buffers actually affect learning with PC is an interesting direction for future work that can be quite easily verified or not. Once we understand how replay buffers affect PC, we can suitably modify existing algorithms to accommodate better learning with PC. However, perhaps PC is just better suited to online learning scenarios. Our results on the MuJoCo benchmark form some initial evidence for this statement. Evidence provided by Song *et al.* [28] also suggest PC is better suited than BP to online learning scenarios and learning off limited amounts of data. However, it is clear that additional work is needed to verify these ideas.

5.2.5 Testing on Other Benchmarks

In this thesis we have exclusively used environments provided by OpenAI `gym` [46] as it provides a standard interface for agent environment interaction, and so there is less overhead associated with programming scripts for different environments. However, there are many more environments and benchmarks out there provided by OpenAI, DeepMind, Unity and more. Testing on different environments is important so that we gain further insight into what hyperparameters work, what types of environments are hard to solve and why, and how to overcome inherent challenges in RL like the exploration-exploitation trade off.

5.2.6 Training with Different Algorithms

Using different algorithms and network architectures is incredibly important if we are to convince the deep learning community to adopt PC. By showing PC works with different RL algorithms and on more sophisticated architectures we can spread further belief and interest in PC. Of the algorithms we used, deep Q-learning is quite an artificial algorithm that implements many tricks that work well, but without strong theoretical justifications. Policy gradient methods come with stronger theoretical backing, and so perhaps looking at other popular policy gradient algorithms, like advantage actor critic (A2C) [92] or trust region policy optimisation (TRPO) [96], is a promising direction for building further justification for the adoption of PC.

Appendix A

Derivations

A.1 Derivation of the error back-propagation step

The goal of backpropagation (BP) is to compute the derivatives $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^l}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{b}^1}$ for $l = 2, ..., L$. As an intermediate step we compute the derivates $\frac{\partial \mathcal{L}}{\partial \mathbf{z}^l}$ for $l = 2, ..., L$ by back-propagating the error at the output. We note that the output $\hat{\mathbf{y}} = \mathbf{a}^L$, and so we can compute,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}^L}$$

directly from the loss function. Where $\frac{\partial \mathcal{L}}{\partial \mathbf{a}^L}$ is the row vector,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}^L} = \left[\frac{\partial \mathcal{L}}{\partial a_1^L}, ..., \frac{\partial \mathcal{L}}{\partial a_{n^L}^L} \right]$$

We can then compute,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^L} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^L} \frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L}$$

Where $\frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L}$ is an $n^L \times n^L$ Jacobian matrix computed using the final activation function g^L. Once we have $\frac{\partial \mathcal{L}}{\partial \mathbf{z}^L}$ we can compute $\frac{\partial \mathcal{L}}{\partial \mathbf{z}^l}$ for $l = 2, ..., L-1$ by the chain rule of

differentiation,

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \mathbf{z}^l} &= \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{l+1}} \frac{\partial \mathbf{z}^{l+1}}{\partial \mathbf{z}^l} \\ &= \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{l+1}} \frac{\partial \mathbf{z}^{l+1}}{\partial \mathbf{a}^l} \frac{\partial \mathbf{a}^l}{\partial \mathbf{z}^l}\end{aligned}$$

Since $\mathbf{z}^{l+1} = \mathbf{W}^{l+1}\mathbf{a}^l + \mathbf{b}^{l+1}$, then $\frac{\partial \mathbf{z}^{l+1}}{\partial \mathbf{a}^l} = \mathbf{W}^{l+1}$. Once again, $\frac{\partial \mathbf{a}^l}{\partial \mathbf{z}^l}$ is an $n^l \times n^l$ Jacobian matrix computed using the activation function g^l. So,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^l} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{l+1}} \mathbf{W}^{l+1} \frac{\partial g^l}{\partial \mathbf{z}^l}$$

Recall that,

$$\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$$

and so it is relatively easy to see that,

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \mathbf{W}^l} &= \frac{\partial \mathbf{z}^l}{\partial \mathbf{W}^l} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^l} \\ &= \left[\mathbf{a}^{l-1} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^l}\right]^T\end{aligned}$$

and that,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^l} = \frac{\partial \mathbf{z}^l}{\partial \mathbf{b}^l} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^l} = \left[\frac{\partial \mathcal{L}}{\partial \mathbf{z}^l}\right]^T$$

A.2 Derivation of the inference update step in predictive coding

First recall the forward equations,

$$\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$$

$$\mathbf{a}^l = g^l(\mathbf{z}^l)$$

Letting $\mathbf{x}^l$ denote the pre-activations $\mathbf{z}^l$ and framing the forward equations in terms of $\mathbf{x}^l$ gives us,

$$\mathbf{x}^l = \mathbf{W}^l g^l(\mathbf{x}^{l-1}) + \mathbf{b}^l$$

The goal of the relaxation phase in predictive coding networks is to minimise the global energy function,

$$E(\mathbf{x}^1, ..., \mathbf{x}^L; \boldsymbol{\theta}) = \frac{1}{2}\sum_{l=2}^{L}(\mathbf{x}^l - \mathbf{W}^l g^l(\mathbf{x}^{l-1}) - \mathbf{b}^l)^2 = \frac{1}{2}\sum_{l=2}^{L}\left(\boldsymbol{\mathcal{E}}^l\right)^2$$

Let's simply consider the layer l in our calculations and observe that $\mathbf{x}^l$ only appears twice in the energy function. Let E^l and E^{l+1} denote the terms in which $\mathbf{x}^l$ appears, that is,

$$E^l = \frac{1}{2}(\mathbf{x}^l - \mathbf{W}^l g^l(\mathbf{x}^{l-1}) - \mathbf{b}^l)^2$$

$$E^{l+1} = \frac{1}{2}(\mathbf{x}^{l+1} - \mathbf{W}^{l+1} g^l(\mathbf{x}^l) - \mathbf{b}^{l+1})^2$$

Now the partial derivative of E with respect to $\mathbf{x}^l$ can be written as,

$$\frac{\partial E}{\partial \mathbf{x}^l} = \frac{\partial E^l + E^{l+1}}{\partial \mathbf{x}^l}$$

By the chain rule of calculus we compute

$$\frac{\partial E^l}{\partial \mathbf{x}^l} = \mathbf{x}^l - \mathbf{W}^l g^l(\mathbf{x}^{l-1}) - \mathbf{b}^l = \boldsymbol{\mathcal{E}}^l$$

and,

$$\begin{aligned}\frac{\partial E^{l+1}}{\partial \mathbf{x}^l} &= (\mathbf{x}^{l+1} - \mathbf{W}^{l+1} g^l(\mathbf{x}^l) - \mathbf{b}^{l+1})\frac{\partial}{\partial \mathbf{x}^l}(\mathbf{x}^{l+1} - \mathbf{W}^{l+1} g^l(\mathbf{x}^l) - \mathbf{b}^{l+1}) \\ &= -\frac{\partial g^l}{\partial \mathbf{x}^l}\left(\mathbf{W}^{l+1}\right)^T \boldsymbol{\mathcal{E}}^{l+1}\end{aligned}$$

To minimise the energy function we move in the direction of the negative gradient, that is,

$$\Delta \mathbf{x}^l \leftarrow \alpha \left(-\boldsymbol{\mathcal{E}}^l + \frac{\partial g^l}{\partial \mathbf{x}^l} \left(\left(\mathbf{W}^{l+1} \right)^T \boldsymbol{\mathcal{E}}^{l+1} \right) \right)$$

A.3 Derivation of the parameter update step in predictive coding

After inference we update the parameters of the network $\boldsymbol{\theta} = \{\mathbf{W}^2, \mathbf{W}^3, ..., \mathbf{W}^L, \mathbf{b}^2, \mathbf{b}^3, ..., \mathbf{b}^L\}$ to minimise the global energy function,

$$E(\mathbf{x}^1, ..., \mathbf{x}^L; \boldsymbol{\theta}) = \frac{1}{2} \sum_{l=2}^{L} (\mathbf{x}^l - \mathbf{W}^l g^l(\mathbf{x}^{l-1}) - \mathbf{b}^l)^2 = \frac{1}{2} \sum_{l=2}^{L} \left(\boldsymbol{\mathcal{E}}^l \right)^2$$

Again let's simply consider the layer l. We note that $\mathbf{W}^l$ and $\mathbf{b}^l$ only appear in one term of the energy function, specifically,

$$E^l = \frac{1}{2} (\mathbf{x}^l - \mathbf{W}^l g^l(\mathbf{x}^{l-1}) - \mathbf{b}^l)^2$$

and so we can write the partial derivative of E^l with respect to $\mathbf{W}^l$ as,

$$\begin{aligned} \frac{\partial E}{\partial \mathbf{W}^l} = \frac{\partial E^l}{\partial \mathbf{W}^l} &= (\mathbf{x}^l - \mathbf{W}^l g^l(\mathbf{x}^{l-1}) - \mathbf{b}^l) \frac{\partial}{\partial \mathbf{W}^l} (\mathbf{x}^l - \mathbf{W}^l g^l(\mathbf{x}^{l-1}) - \mathbf{b}^l) \\ &= \boldsymbol{\mathcal{E}}^l \frac{\partial}{\partial \mathbf{W}^l} (\mathbf{x}^l - \mathbf{W}^l g^l(\mathbf{x}^{l-1}) - \mathbf{b}^l) = -\boldsymbol{\mathcal{E}}^l \left[g^l(\mathbf{x}^{l-1}) \right]^T \end{aligned}$$

For the biases $\mathbf{b}^l$ this is even easier,

$$\frac{\partial E}{\partial \mathbf{b}^l} = \frac{\partial E^l}{\partial \mathbf{b}^l} = -(\mathbf{x}^l - \mathbf{W}^l g^l(\mathbf{x}^{l-1}) - \mathbf{b}^l) = -\boldsymbol{\mathcal{E}}^l$$

This gives us the following parameter update steps,

$$\Delta \mathbf{W}^l \leftarrow \eta \boldsymbol{\mathcal{E}}^l \left[g^l(\mathbf{x}^{l-1}) \right]^T$$

$$\Delta \mathbf{b}^l \leftarrow \eta \boldsymbol{\mathcal{E}}^l$$

A.4 Derivation of the REINFORCE update step

Recall that the policy gradient theorem [78, Ch. 13.2] states,

$$J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a \mid s, \boldsymbol{\theta})$$

By taking expectations with respect to π we get,

$$\begin{aligned}
J(\boldsymbol{\theta}) &\propto \mathbb{E}_\pi \left[\sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a \mid s, \boldsymbol{\theta}) \right] \\
&= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a \mid S_t, \boldsymbol{\theta}) \right] && \text{(by the definition of expectations)} \\
&= \mathbb{E}_\pi \left[\sum_a \pi(a \mid S_t, \boldsymbol{\theta}) q_\pi(S_t,a) \frac{\nabla \pi(a \mid S_t, \boldsymbol{\theta})}{\pi(a \mid S_t, \boldsymbol{\theta})} \right] \\
&= \mathbb{E}_\pi \left[q_\pi(S_t,A_t) \frac{\nabla \pi(A_t \mid S_t, \boldsymbol{\theta})}{\pi(A_t \mid S_t, \boldsymbol{\theta})} \right] && \text{(by the definition of expectations)} \\
&= \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(A_t \mid S_t, \boldsymbol{\theta})}{\pi(A_t \mid S_t, \boldsymbol{\theta})} \right] && \text{(since } \mathbb{E}_\pi[G_t \mid S_t, A_t] = q_\pi(S_t,A_t)\text{)}
\end{aligned}$$

Appendix B

Additional Plots

B.1 Frozen lake as a supervised learning task

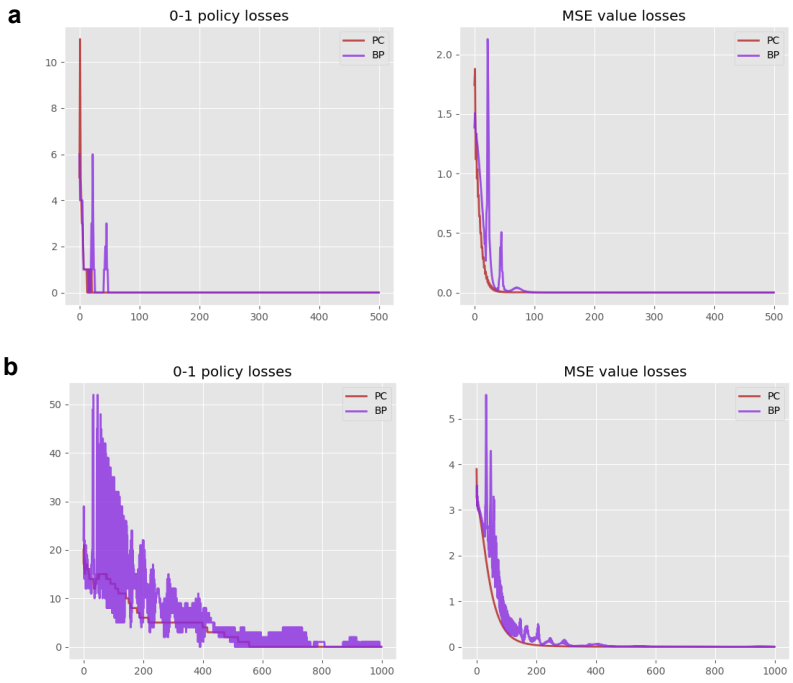

Figure B.1: Frozen lake as a supervised learning task solved with Q-network. Network trained with BP (**purple**), network trained with PC (**red**) and clamp loss $\mathcal{L}_{\text{clamp}}$. 0-1 policy loss computed with Equation 4.5, value loss computed with Equation 4.4. x-axis corresponds to the number of batch iterations. The plots are averaged over 10 seeds. **a:** frozen lake 4×4. **b:** frozen lake 8×8.

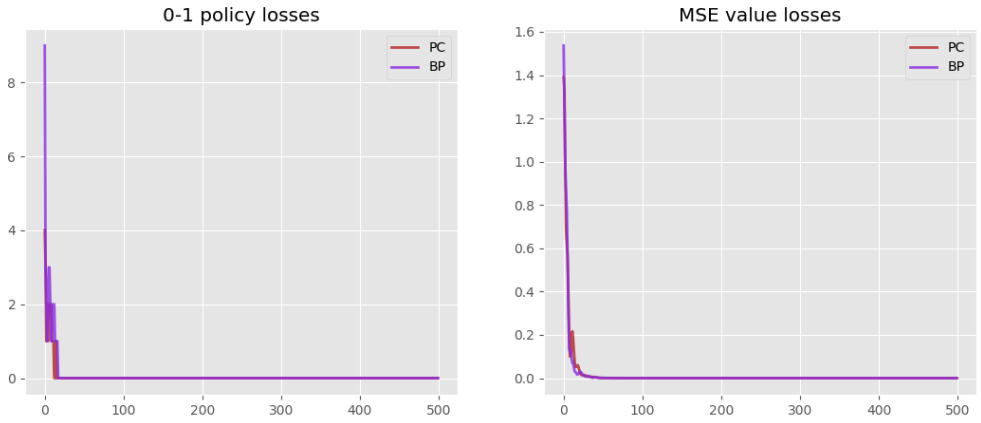

Figure B.2: **ADAM:** Frozen lake as a supervised learning task solved with Q-network. Network trained with BP (**purple**), network trained with PC (**red**) and clamp loss $\mathcal{L}_{\text{clamp}}$. 0-1 policy loss computed with Equation 4.5, value loss computed with 4.4 Equation. x-axis corresponds to the number of batch iterations. The plots are averaged over 10 seeds.

B.2 Cart pole with the CEM (Adam plots)

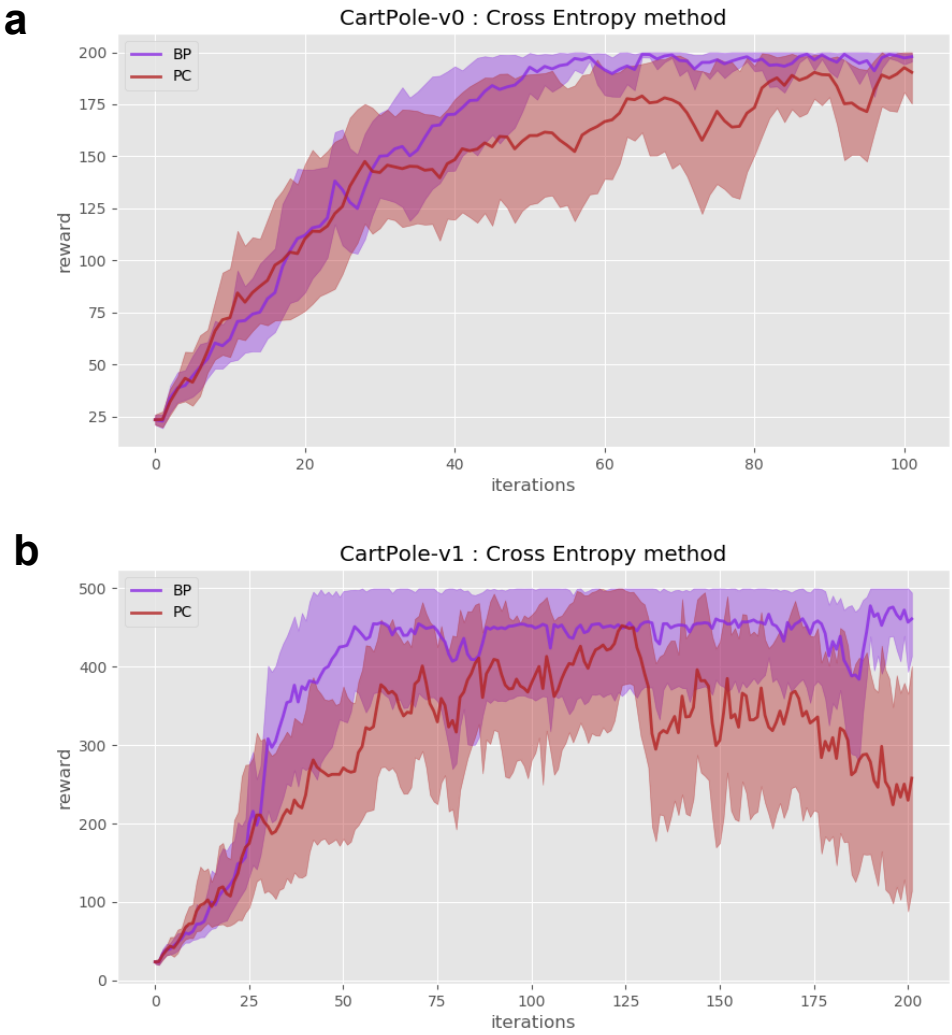

Figure B.3: **ADAM:** Cart pole environment solved with the CEM. Network trained with BP (**purple**), network trained with PC (**red**). Average batch reward computed with Equation 3.5. The plots are averaged over 25 seeds. **a:** cart pole v0, episode ends at step 200. **b:** cart pole v1, episode ends at step 500.

B.3 Cart pole with Q-learning (Adam plot)

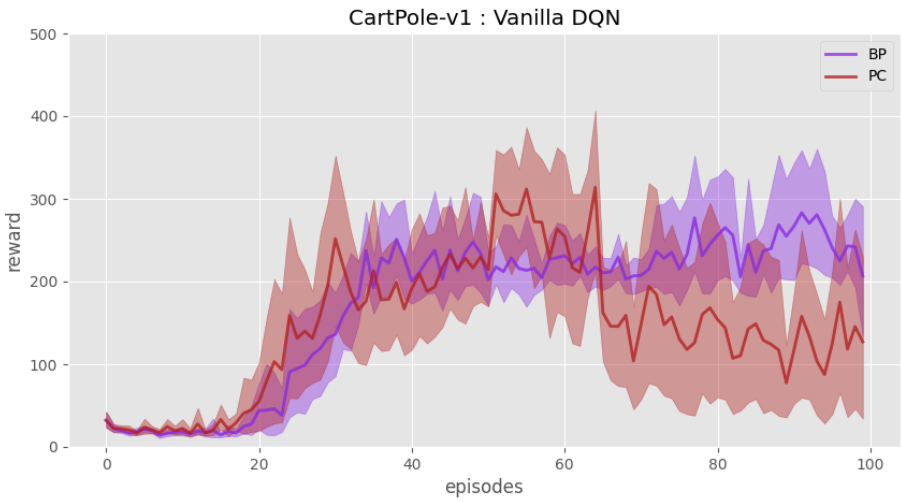

Figure B.4: **ADAM:** Cart pole v1 environment solved with Q-network. Network trained with BP (**purple**), network trained with PC (**red**). Episode reward computed with Equation 3.32. The plots are averaged over 10 seeds.

B.4 Results on the MuJoCo Benchmark (bigger plots)

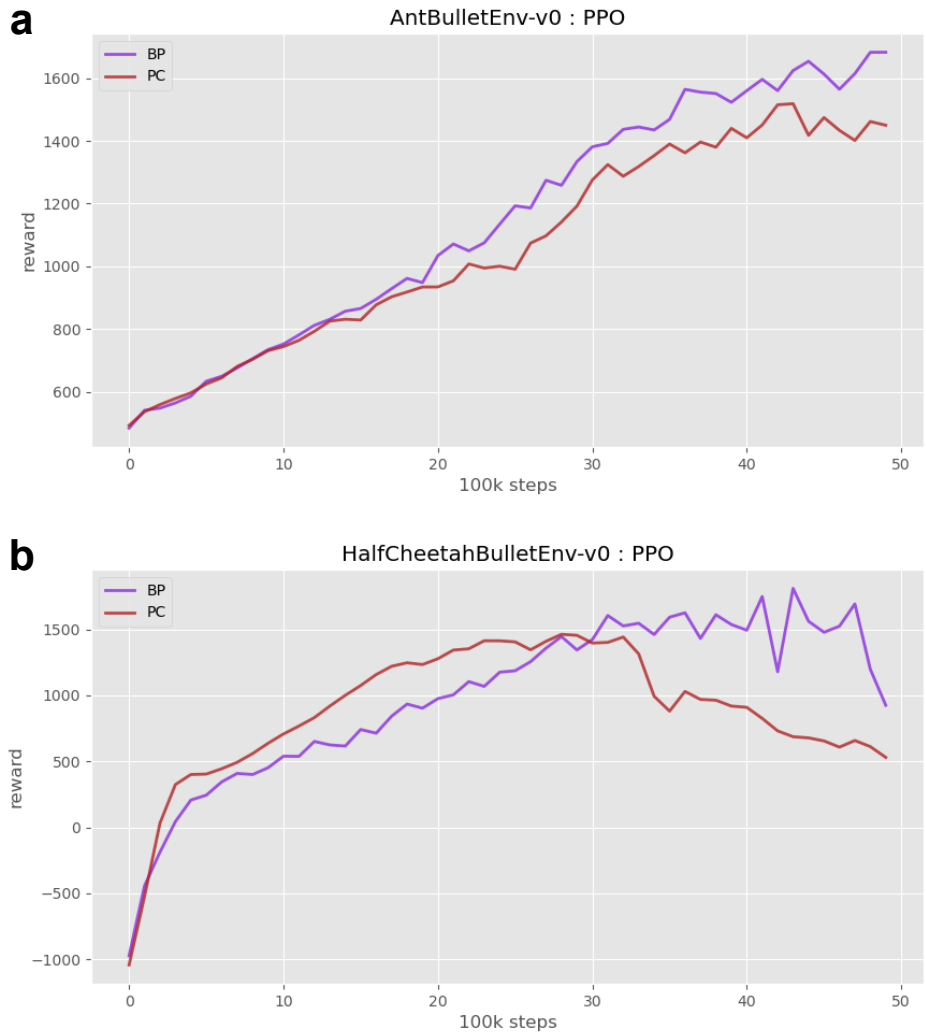

Figure B.5: **a:** `AntBulletEnv-v0` solved with PPO. **b:** `HalfCheetahtBulletEnv-v0` solved with PPO. Actor and Critic trained with BP (**purple**), Actor and Critic trained with PC (**red**). Episode reward computed with Equation 3.34. Only one seed run.

Appendix C

Miscellaneous

C.1 Taxonomy of RL algorithms

Taxonomy	Properties
Model-free v.s Model-based	Model-free algorithms are methods that don't explicitly build or learn a model of the environment and the reward dynamics. Model-free methods map observations directly to actions. In contrast model-based methods use a learned or given model of the environments dynamics and typically plan through the model in order to pick actions.
Policy-based v.s Value-based	Policy based methods explicitly represent the policy as a distribution over actions. Value-based methods numerically approximate state-action values or Q-values and the policy is represented as an argmax over the Q-values.
On-policy v.s Off-Policy	On-policy methods require that the experience collected for the agent to train on was collected by following the current policy. Off-policy methods do not require this restriction and can train on old experience collected by an older policy.
Monte Carlo v.s Temporal Difference (TD)	Monte Carlo methods need to play full episodes before they can compute targets to train on. TD methods instead seek to minimise the TD error and only need to play one step before they can start learning from target state-values.

Table C.1: Taxonomy of RL algorithms.

C.2 Architectural Details of the DQN Used for Atari 2600 Experiments

Layer	Input Dimension	Ouput Dimension
Conv 2D (Input)	$4 \times 84 \times 84$	32 channels, 8×8 kernel, stride 4.
Conv2D	32 channels	64 channels, 4×4 kernel, stride 2.
Conv2D	64 channels	64 channels, 4×3 kernel, stride 1.
Linear	3136	512
Linear (output)	512	6 for Pong, 5 for Breakout

Table C.2: Architectural details of the convolutional DQN used for Atari 2600 pixel environments.

Bibliography

1. Krizhevsky, A., Sutskever, I. & Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* **25** (2012).
2. Qiu, X., Zhang, L., Ren, Y., Suganthan, P. N. & Amaratunga, G. *Ensemble deep learning for regression and time series forecasting* in *2014 IEEE symposium on computational intelligence in ensemble learning (CIEL)* (2014), 1–6.
3. Kingma, D. P. & Welling, M. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).
4. Goodfellow, I., Bengio, Y. & Courville, A. *Deep learning* (MIT press, 2016).
5. Cireşan, D. C., Meier, U., Gambardella, L. M. & Schmidhuber, J. Deep, big, simple neural nets for handwritten digit recognition. *Neural computation* **22,** 3207–3220 (2010).
6. Rosenblatt, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review* **65,** 386 (1958).
7. Marcus, G. Deep learning: A critical appraisal. *arXiv preprint arXiv:1801.00631* (2018).
8. LeCun, Y. *et al.* Handwritten digit recognition with a back-propagation network. *Advances in neural information processing systems* **2** (1989).
9. Hubel, D. H. & Wiesel, T. N. Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology* **148,** 574 (1959).
10. Hubel, D. H. & Wiesel, T. N. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology* **160,** 106 (1962).
11. Hubel, D. H. & Wiesel, T. N. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology* **195,** 215–243 (1968).
12. Rumelhart, D. E., Hinton, G. E. & Williams, R. J. Learning representations by back-propagating errors. *nature* **323,** 533–536 (1986).
13. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* **15,** 1929–1958 (2014).
14. Sutton, R. S. & Barto, A. G. *Reinforcement learning: An introduction* (MIT press, 2018).
15. Montague, P. R., Dayan, P. & Sejnowski, T. J. A framework for mesencephalic dopamine systems based on predictive Hebbian learning. *Journal of neuroscience* **16,** 1936–1947 (1996).

16. Lin, L.-J. *Reinforcement learning for robots using neural networks* tech. rep. (Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993).
17. Finn, C. *et al.* Learning visual feature spaces for robotic manipulation with deep spatial autoencoders. *arXiv preprint arXiv:1509.06113* **25,** 2 (2015).
18. Recht, B. A tour of reinforcement learning: The view from continuous control. *arXiv preprint arXiv:1806.09460* (2018).
19. Mnih, V. *et al.* Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).
20. Mnih, V. *et al.* Human-level control through deep reinforcement learning. *nature* **518,** 529–533 (2015).
21. Silver, D. *et al.* Mastering the game of go without human knowledge. *nature* **550,** 354–359 (2017).
22. Vinyals, O. *et al.* Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* **575,** 350–354 (2019).
23. Brown, N. & Sandholm, T. Superhuman AI for multiplayer poker. *Science* **365,** 885–890 (2019).
24. Hambly, B., Xu, R. & Yang, H. Recent advances in reinforcement learning in finance. *arXiv preprint arXiv:2112.04553* (2021).
25. Tsividis, P. A., Pouncy, T., Xu, J. L., Tenenbaum, J. B. & Gershman, S. J. *Human learning in Atari* in *2017 AAAI spring symposium series* (2017).
26. McCloskey, M. & Cohen, N. J. in *Psychology of learning and motivation* 109–165 (Elsevier, 1989).
27. McClelland, J. L., McNaughton, B. L. & O'Reilly, R. C. Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory. *Psychological review* **102,** 419 (1995).
28. Song, Y. *et al.* Inferring Neural Activity Before Plasticity: A Foundation for Learning Beyond Backpropagation. *bioRxiv* (2022).
29. Crick, F. The recent excitement about neural networks. *Nature* **337,** 129–132 (1989).
30. Stork, D. G. *Is backpropagation biologically plausible* in *International Joint Conference on Neural Networks* **2** (1989), 241–246.
31. Whittington, J. C. & Bogacz, R. Theories of error back-propagation in the brain. *Trends in cognitive sciences* **23,** 235–250 (2019).
32. Lillicrap, T. P., Santoro, A., Marris, L., Akerman, C. J. & Hinton, G. Backpropagation and the brain. *Nature Reviews Neuroscience* **21,** 335–346 (2020).
33. Srinivasan, M. V., Laughlin, S. B. & Dubs, A. Predictive coding: a fresh view of inhibition in the retina. *Proceedings of the Royal Society of London. Series B. Biological Sciences* **216,** 427–459 (1982).
34. Rao, R. P. & Ballard, D. H. Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects. *Nature neuroscience* **2,** 79–87 (1999).
35. Friston, K. A theory of cortical responses. *Philosophical transactions of the Royal Society B: Biological sciences* **360,** 815–836 (2005).
36. Walsh, K. S., McGovern, D. P., Clark, A. & O'Connell, R. G. Evaluating the neurophysiological evidence for predictive processing as a model of perception. *Annals of the new York Academy of Sciences* **1464,** 242–268 (2020).

37. Kell, A. J., Yamins, D. L., Shook, E. N., Norman-Haignere, S. V. & McDermott, J. H. A task-optimized neural network replicates human auditory behavior, predicts brain responses, and reveals a cortical processing hierarchy. *Neuron* **98,** 630–644 (2018).
38. Millidge, B., Seth, A. & Buckley, C. L. Predictive coding: a theoretical and experimental review. *arXiv preprint arXiv:2107.12979* (2021).
39. Whittington, J. C. & Bogacz, R. An approximation of the error backpropagation algorithm in a predictive coding network with local hebbian synaptic plasticity. *Neural computation* **29,** 1229–1262 (2017).
40. Salvatori, T. *et al.* Learning on arbitrary graph topologies via predictive coding. *arXiv preprint arXiv:2201.13180* (2022).
41. Salvatori, T. *et al.* Associative memories via predictive coding. *Advances in Neural Information Processing Systems* **34,** 3874–3886 (2021).
42. Millidge, B., Salvatori, T., Song, Y., Bogacz, R. & Lukasiewicz, T. Predictive Coding: Towards a Future of Deep Learning beyond Backpropagation? *arXiv preprint arXiv:2202.09467* (2022).
43. Kingma, D. P. & Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
44. Bellemare, M. G., Naddaf, Y., Veness, J. & Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research* **47,** 253–279 (2013).
45. Schulman, J., Wolski, F., Dhariwal, P., Radford, A. & Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).
46. Brockman, G. *et al. OpenAI Gym* 2016. eprint: `arXiv:1606.01540`.
47. McCulloch, W. S. & Pitts, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics* **5,** 115–133 (1943).
48. Hinton, G. E. Training products of experts by minimizing contrastive divergence. *Neural computation* **14,** 1771–1800 (2002).
49. Boser, B. E., Guyon, I. M. & Vapnik, V. N. *A training algorithm for optimal margin classifiers* in *Proceedings of the fifth annual workshop on Computational learning theory* (1992), 144–152.
50. Glorot, X., Bordes, A. & Bengio, Y. *Deep sparse rectifier neural networks* in *Proceedings of the fourteenth international conference on artificial intelligence and statistics* (2011), 315–323.
51. Murphy, K. P. *Machine learning: a probabilistic perspective* (MIT press, 2012).
52. Block, H.-D. The perceptron: A model for brain functioning. i. *Reviews of Modern Physics* **34,** 123 (1962).
53. Novikoff, A. B. *On convergence proofs for perceptrons* tech. rep. (STANFORD RESEARCH INST MENLO PARK CA, 1963).
54. Minsky, M. L. & Papert, S. A. *Perceptrons: expanded edition* 1988.
55. Hornik, K., Stinchcombe, M. & White, H. Multilayer feedforward networks are universal approximators. *Neural networks* **2,** 359–366 (1989).
56. Li, H., Xu, Z., Taylor, G., Studer, C. & Goldstein, T. Visualizing the loss landscape of neural nets. *Advances in neural information processing systems* **31** (2018).
57. Martín Abadi *et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems* Software available from tensorflow.org. 2015. `https://www.tensorflow.org/api_docs/python/tf/nn/conv2d`.

58. He, K., Zhang, X., Ren, S. & Sun, J. *Deep residual learning for image recognition* in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), 770–778.
59. Ioffe, S. & Szegedy, C. *Batch normalization: Accelerating deep network training by reducing internal covariate shift* in *International conference on machine learning* (2015), 448–456.
60. Knill, D. C. & Pouget, A. The Bayesian brain: the role of uncertainty in neural coding and computation. *TRENDS in Neurosciences* **27,** 712–719 (2004).
61. Millidge, B., Song, Y., Salvatori, T., Lukasiewicz, T. & Bogacz, R. Backpropagation at the Infinitesimal Inference Limit of Energy-Based Models: Unifying Predictive Coding, Equilibrium Propagation, and Contrastive Hebbian Learning. *arXiv preprint arXiv:2206.02629* (2022).
62. Millidge, B., Tschantz, A. & Buckley, C. L. Predictive Coding Approximates Backprop Along Arbitrary Computation Graphs. *Neural Computation* **34,** 1329–1368. ISSN: 0899-7667. eprint: `https://direct.mit.edu/neco/article-pdf/34/6/1329/2023477/neco\_a\_01497.pdf`. `https://doi.org/10.1162/neco%5C_a%5C_01497` (May 2022).
63. Hopfield, J. J. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences* **79,** 2554–2558 (1982).
64. Hebb, D. O. The first stage of perception: growth of the assembly. *The Organization of Behavior* **4,** 60–78 (1949).
65. Storkey, A. *Increasing the capacity of a Hopfield network without sacrificing functionality* in *International Conference on Artificial Neural Networks* (1997), 451–456.
66. Krotov, D. & Hopfield, J. J. Dense associative memory for pattern recognition. *Advances in neural information processing systems* **29** (2016).
67. Ramsauer, H. *et al.* Hopfield networks is all you need. *arXiv preprint arXiv:2008.02217* (2020).
68. Salakhutdinov, R. & Hinton, G. Semantic hashing. *International Journal of Approximate Reasoning* **50,** 969–978 (2009).
69. Salvatori, T., Song, Y., Lukasiewicz, T., Bogacz, R. & Xu, Z. Predictive coding can do exact backpropagation on convolutional and recurrent neural networks. *arXiv preprint arXiv:2103.03725* (2021).
70. Ororbia, A. G. & Mali, A. *Biologically motivated algorithms for propagating local target representations* in *Proceedings of the aaai conference on artificial intelligence* **33** (2019), 4651–4658.
71. Han, K. *et al.* Deep predictive coding network with local recurrent processing for object recognition. *Advances in neural information processing systems* **31** (2018).
72. Liu, J., Gong, M. & He, H. Deep associative neural network for associative memory based on unsupervised representation learning. *Neural Networks* **113,** 41–53 (2019).
73. Barron, H. C., Auksztulewicz, R. & Friston, K. Prediction and memory: A predictive coding account. *Progress in neurobiology* **192,** 101821 (2020).
74. Mohri, M., Rostamizadeh, A. & Talwalkar, A. *Foundations of machine learning* (MIT press, 2018).
75. Watkins, C. J. C. H. Learning from delayed rewards (1989).
76. Watkins, C. J. & Dayan, P. Q-learning. *Machine learning* **8,** 279–292 (1992).

77. Melo, F. S. Convergence of Q-learning: A simple proof. *Institute Of Systems and Robotics, Tech. Rep,* 1–4 (2001).
78. Sutton, R. S. Learning to predict by the methods of temporal differences. *Machine learning* **3,** 9–44 (1988).
79. Rubinstein, R. Y. Optimization of computer simulation models with rare events. *European Journal of Operational Research* **99,** 89–112 (1997).
80. Sani, A. Stochastic Modelling and Intervention of the Spread of HIV/AIDS (2009).
81. Busoniu, L., Babuska, R., De Schutter, B. & Ernst, D. *Reinforcement learning and dynamic programming using function approximators* (CRC press, 2017).
82. Pihur, V., Datta, S. & Datta, S. Weighted rank aggregation of cluster validation measures: a monte carlo cross-entropy approach. *Bioinformatics* **23,** 1607–1615 (2007).
83. Alon, G., Kroese, D. P., Raviv, T. & Rubinstein, R. Y. Application of the cross-entropy method to the buffer allocation problem in a simulation-based environment. *Annals of Operations Research* **134,** 137–151 (2005).
84. Cohen, I., Golany, B. & Shtub, A. Resource allocation in stochastic, finite-capacity, multi-project systems through the cross entropy methodology. *Journal of Scheduling* **10,** 181–193 (2007).
85. Costa, A., Jones, O. D. & Kroese, D. Convergence properties of the cross-entropy method for discrete optimization. *Operations Research Letters* **35,** 573–580 (2007).
86. De Boer, P.-T., Kroese, D. P., Mannor, S. & Rubinstein, R. Y. A tutorial on the cross-entropy method. *Annals of operations research* **134,** 19–67 (2005).
87. Van Hasselt, H., Guez, A. & Silver, D. *Deep reinforcement learning with double q-learning* in *Proceedings of the AAAI conference on artificial intelligence* **30** (2016).
88. Schaul, T., Quan, J., Antonoglou, I. & Silver, D. Prioritized experience replay. *arXiv preprint arXiv:1511.05952* (2015).
89. Fortunato, M. *et al.* Noisy networks for exploration. *arXiv preprint arXiv:1706.10295* (2017).
90. Wang, Z. *et al. Dueling network architectures for deep reinforcement learning* in *International conference on machine learning* (2016), 1995–2003.
91. Bellemare, M. G., Dabney, W. & Munos, R. *A distributional perspective on reinforcement learning* in *International Conference on Machine Learning* (2017), 449–458.
92. Mnih, V. *et al. Asynchronous methods for deep reinforcement learning* in *International conference on machine learning* (2016), 1928–1937.
93. Schulman, J., Moritz, P., Levine, S., Jordan, M. & Abbeel, P. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438* (2015).
94. Barto, A. G., Sutton, R. S. & Anderson, C. W. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics,* 834–846 (1983).
95. Wawrzyński, P. *A cat-like robot real-time learning to run* in *International Conference on Adaptive and Natural Computing Algorithms* (2009), 380–390.
96. Schulman, J., Levine, S., Abbeel, P., Jordan, M. & Moritz, P. *Trust region policy optimization* in *International conference on machine learning* (2015), 1889–1897.