Deep Reinforcement Learning of Video Games

Jos van de Wolfshaar
s2098407
September 29, 2017

MSc. Project
Artificial Intelligence
University of Groningen, The Netherlands

Supervisors
Dr. M.A. (Marco) Wiering
Prof. dr. L.R.B. (Lambert) Schomaker
ALICE Institute
University of Groningen
Nijenborgh 9, 9747 AG, Groningen, The Netherlands

university of groningen / faculty of science and engineering / artificial intelligence

Contents

Abstract

The ability to learn is arguably the most crucial aspect of human intelligence. In reinforcement learning, we attempt to formalize a certain type of learning that is based on rewards and penalties. These supervisory signals should guide an agent to learn optimal behavior. In particular, this research focuses on deep reinforcement learning, where the agent should learn to play video games solely from pixel input.

This thesis contributes to deep reinforcement learning research by assessing several variations to an existing state-of-the-art algorithm. First, we provide an extensive analysis on how the design decisions of the agent's deep neural network affect its performance. Second, we introduce a novel neural layer that allows for local specializations in the visual input of the agents, as opposed to the global weight sharing that occurs in convolutional layers. Third, we introduce a 'what' and 'where' neural network architecture, inspired by the information flow of the visual cortical areas in the human brain. Finally, we explore prototype based deep reinforcement learning by introducing a novel output layer that is largely inspired by learning vector quantization. In a subset of our experiments, we show substantial improvements compared to existing alternatives.

Chapter 1

Introduction

Learning is a crucial aspect of intelligence and it is this phenomenon that we try to translate into formal mathematical rules when we practice machine learning research. Formalizing learning increases our understanding and admiration of human intelligence, as we can more accurately argue what are crucial aspects and limitations of learning machines and organisms. Machine learning (ML) is now recognized as a field of science for a handful of decades. A vast range of different approaches and problems exist within ML. Roughly speaking, we can divide machine learning into three main themes: *supervised learning*, *unsupervised learning* and *reinforcement learning*. In the remainder of this section, we briefly introduce the aforementioned themes in machine learning. In Section 1.1 we narrow down to the main research topic addressed in this thesis, known as deep reinforcement learning. Section 1.2 lists the research questions that will be addressed in the remaining chapters of this thesis.

In supervised learning (SL), we are concerned with the pathological situation where we explicitly tell a *machine learning algorithm* what the correct response y is to some stimuli $\boldsymbol{x}$. For example, we could build an SL algorithm that can recognize handwritten digits (y) from small images ($\boldsymbol{x}$). For *classification* problems, y is a *class label* and in the case of handwritten digit recognition it is simply $y \in \{0, 1, \ldots, 9\}$. Many other forms of supervised learning have been studied and they all have a major factor in common: human-provided labeling. This consequently restricts SL to deal with problems that are well-defined in the sense that it is straightforward to separate different responses and to reliably define the according labels. Although a great deal of human learning is to a certain extent supervised, we are also capable of learning autonomously and without being told *exactly* what would have been the correct response.

Unsupervised learning (UL) focuses on problems in which we try to reveal the underlying structure of data. Of course, given the means for measuring or simulating real world phenomena with satisfactory precision, we can represent almost any entity with data. Once represented as data, for some of these UL algorithms they become vectors $\boldsymbol{x}_i$ that populate some *input space* $\mathcal{X} \subseteq \mathbb{R}^n$. UL algorithms try to extract high-level notions about the data that are useful for e.g. *exploration*, *visualization*, *feature selection* and many other applications. Most UL algorithms achieve this by exploiting the relations between *distances* in $\mathcal{X}$. Different kinds of tasks require different ways of dealing with $\mathcal{X}$ and the corresponding distance measures. Note that in UL, the algorithms are designed to solve problems that do not make use of explicit labeling. Instead, they are used to explain the data with a lower complexity than the raw data itself, preferably such that it fosters our understanding of the phenomenon that the data represents.

Lastly, reinforcement learning (RL) could be considered to operate between SL and UL in terms of supervision. In RL, we do not tell the algorithm explicitly what to do, we rather let it try some specific (sequences of) responses and provide feedback in terms of a *reward* or *penalty*. Note that being rewarded or penalized – as in RL – is a different kind of signal than being told exactly – as in SL – what the correct response should have been. So RL algorithms specialize in problems that can be solved by a trial-and-error process. For many

tasks that we deal with in the real world, we cannot formally describe a desired response or decision at every moment in time. This is partly due to the complexity of the decisions that can be made, but also because of the fact that we simply have limited resources in terms of time and equipment to do so. The existence of RL alleviates this burden and allows machines to solve complex tasks such as elevator control (Crites and Barto, 1998), traffic light control (Wiering, 2000), playing board games (Tesauro, 1995; Silver et al., 2016), playing video games (Mnih et al., 2013), controlling robotic arms (Levine et al., 2016), designing neural network architectures (Zoph and Le, 2017) and many more. In all of these tasks, it is difficult to specify time- and order-dependent desired responses, but it is relatively straightforward to define what are desirable *states* for the system to be in.

In general, allowing an algorithm to solve a problem by means of reinforcement learning instead of SL, requires considerably less effort in terms of supervision. The central ideas of reinforcement learning are further discussed in Chapter 3.

1.1 Deep reinforcement learning

Ultimately, machine learning algorithms should be relying on seemingly few assumptions, design and preprocessing effort. To reduce design and preprocessing effort, we can focus our attention on the improvement of existing methods and introduction of algorithms that consider the inputs in a similar way as we do ourselves. Since our eyes merely require photons, and our ears merely require a sound source and a medium we can attempt to develop algorithms that start at the same point of this processing pipeline. Moreover, artificial intelligence research has shown that taking inspiration from biology – perhaps at different scales – can lead to the inception of powerful machine learning algorithms.

Artificial neural networks are a popular example of biologically inspired machine learning models. In these networks, artificial neurons process their input by applying trivial mathematical operations. When a large number of these neurons are combined and organized in a layer-wise fashion, they can exhibit state-of-the-art performance in several machine learning domains. Using many layers of artificial neurons is referred to as deep learning (Goodfellow et al., 2016; Schmidhuber, 2015; LeCun et al., 2015). Deep learning has become increasingly more prominent since the last decade and is now presumably the most practiced field within machine learning research. A more technical discussion of deep learning in the context of this thesis is provided in Chapter 2.

Although the majority of deep learning applications and research focuses on supervised learning, deep learning for reinforcement learning problems has also been explored relatively recently. The combination of the two is more commonly referred to as deep reinforcement learning (DRL). The use of DRL for old arcade games (Mnih et al., 2013) and the ancient game of Go (Silver et al., 2016) are well-known examples within the DRL community. Both reinforcement learning and deep learning are directions in machine learning that are highly generic in principle. Therefore, advancing the unification of these two paradigms is an appealing focus for further research and likely to advance the implementation of systems that ultimately contribute to our society.

1.2 Research questions

This section states the research questions so that, once answered, the whole contributes to the field of machine learning and reinforcement learning, in particular deep reinforcement learning.

1.2.1 Architectural neural network design

One of the merits of DRL is that – in principle – little feature engineering is necessary. However, the designer of the algorithm still has many important decisions to make. Some of these decisions include how many layers should be used (this is partly an efficiency and

performance trade-off), what kind of layers should be used, how many neurons should a layer have, what kind of activation functions are used etc. Although the available literature mostly reports outcomes of research in which architectural neural network design decisions were made successfully, few if any report design decisions that were unsuccessful. Moreover, given the popularity of DL research, many novel ideas have been introduced over the last few years that are worth exploring. The first research questions that come to mind are:

1. To what extent do architectural design decisions and hyperparameters of an agent's deep neural network affect the resulting performance?
   (a) How sensitive are these algorithms to variations?
   (b) What are well performing architectures?
   (c) Are there any 'brittle' hyperparameters?
   (d) Can spatial consistency in the visual input of video games be exploited?

The above questions will be addressed in Chapter 6. Question (d) will be addressed by proposing a new neural network layer that can exploit spatial consistency, meaning that it can locally specialize for certain features.

1.2.2 Prototype based reinforcement learning

On a coarse grained level, decision making as done by RL agents can be related to classification. Usually, classification is solved through supervised learning. One particular class of SL algorithms is known as nearest prototype classification (NPC). The most prominent NPC algorithm is learning vector quantization (LVQ) (Kohonen, 1990; Kohonen et al., 1996). As opposed to linearly separating different kinds of inputs in the final layer of a neural network, LVQ chooses to place prototype vectors in the input space $\mathcal{X}$. Roughly speaking, a new input $\boldsymbol{x}$ is then classified by looking at the nearest prototypes in $\mathcal{X}$. This particular classification scheme could in principle be used for reinforcement learning with some modifications. More specifically, we will look at how it can be used to frame the agent's decision making as a learning vector quantization problem. In that case the prototypes will be placed in a feature space $\mathcal{H} \subseteq R^n$ in which we compare the prototypes to nearby hidden activations $\boldsymbol{h}$ of a deep neural network. We will address the following research question with corresponding subquestions:

2. Is prototype based learning suited for deep reinforcement learning?
   (a) How does it relate to existing LVQ variants?
   (b) What are important hyperparameters?
   (c) What are proper distance measures for $\mathcal{H}$?
   (d) How does it compare to existing approaches for DRL in terms of performance?

To answer these questions, we propose a novel reinforcement learning algorithm in Chapter 7 which is largely inspired by existing LVQ approaches. Our algorithm can be varied in many aspects and we provide the corresponding experiments to advocate certain design decisions.

Part I

Theoretical Background

Chapter 2

Deep Learning

Deep learning (DL) encompasses neural networks with many-layered computations. These 'layers of computation' might be hidden layers in an ordinary fully connected multi-layer perceptron (MLP), but they can also correspond to repetitive computations in recurrent neural networks (RNNs). In the first half of this decade, the machine learning community has witnessed significant advances in optimizing deep neural networks (DNNs). There are a number of factors that have allowed this field of research to gain such momentum. Nowadays, large labeled datasets are available that are typically required for high dimensional inputs with large neural networks for them to generalize well. Other than that, we have witnessed an increase in computing power. Furthermore, there have been some technical advances that allowed the gradients to be sufficiently large and stable to train deep networks. The most prominent successes to date remain in the field of computer vision with the use of convolutional neural networks (CNNs). As of 2012, the state-of-the-art systems in computer vision tasks ranging from classification, segmentation and localization have been dominated by these networks (Krizhevsky et al., 2012; Simonyan and Zisserman, 2014; Szegedy et al., 2015; Srivastava et al., 2015; He et al., 2015a; Huang et al., 2016). Another highly influencing development is that of the advanced RNNs such as long short-term memory (LSTM) networks (Hochreiter and Schmidhuber, 1997). It is important to stress that most of this research is about supervised learning. Hence, these models consider static learning problems in the sense that they do not involve some artificially intelligent agent that interacts with its environment.

This chapter will cover the deep learning models that are most relevant to a reinforcement learning setting. First, we discuss a basic neural network architecture in Section 2.1. Then we discuss how to train such models in Section 2.2. Next, CNNs are explained in Section 2.3. We emphasize that our account of deep learning is by no means complete. This is partially for brevity and because of the fact that most of our models only require the use of a relatively small subset of ideas from DL. There exist excellent surveys on DL that are worth consulting for further study (Schmidhuber, 2015; LeCun et al., 2015) and the recently published textbook by Goodfellow et al. (2016).

2.1 Multi-layer perceptrons

The fundamental unit in deep learning models is the perceptron. The perceptron is a greatly simplified artificial neuron which can perform a trivial mathematical operation. A perceptron linearly combines a set of incoming connections from *inputs*, which can be provided externally or through the output of other perceptrons. If the input is $\boldsymbol{x} \in \mathbb{R}^n$, the output of a perceptron is $f(\boldsymbol{w} \cdot \boldsymbol{x} + b)$ where $f(\cdot)$ is called the *activation function*, the elements of $\boldsymbol{w} \in \mathbb{R}^n$ are the *weights* that represent the connection strengths for the different inputs in $\boldsymbol{x}$ and $b \in \mathbb{R}$ is the *bias*.

One can combine such perceptrons in multiple layers to make *multi-layer perceptrons*

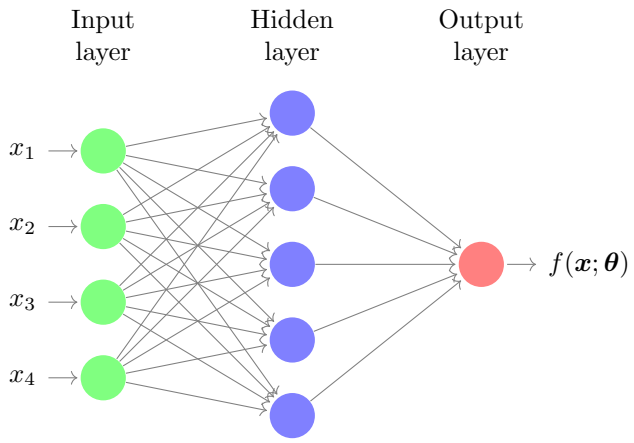


Figure 2.1: Basic multi-layer perceptron (MLP).

(MLPs) as depicted in Figure 2.1. The figure shows a *feedforward neural network* in which there are no connections between the neurons in the same layer and no connections going in the direction of the input layer. The connections are only directed towards the output. In such an approximator, the adjustable parameters are the connections between the layers. The output of this MLP can be used for different kinds of problems such as regression or classification problems. The goal is to find the proper setting of these parameters to maximize the task performance. The next section discusses how this goal can be achieved.

2.2 Optimizing neural networks

This section elaborates on algorithms that are used for training neural networks. We merely discuss approaches that are directly related to our experiments in Part II. The algorithms that we discuss here are a form of gradient descent.

2.2.1 Gradient descent

Gradient descent was first formulated by Cauchy (1847). Gradient descent is an iterative method to find the (local) minimum of a function. For neural networks and many other machine learning method the function to minimize is often referred to as the *loss function* or *cost function*. This function expresses the error of the current approximation to a target distribution. In this text, loss functions are denoted as $\mathcal{L}(\boldsymbol{x}, y; \boldsymbol{\theta})$. The semicolon emphasizes the fact that the role of $\boldsymbol{x}$ and y are conceptually different from the role of $\boldsymbol{\theta}$. The vector $\boldsymbol{x}$ denotes the model's input and y denotes the models *target* (i.e. the desired output). The function should be minimized with respect to $\boldsymbol{\theta}$. To accomplish this, gradient decent methods consider the gradient of the function to find the local direction of steepest descent in the parameter space given by $\boldsymbol{\theta}$. This boils down to iteratively updating $\boldsymbol{\theta}$ as follows:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \boldsymbol{g}_t, \tag{2.1}$$

where

$$\boldsymbol{g}_t = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{x}, y; \boldsymbol{\theta}), \tag{2.2}$$

and $\eta \in (0, 1)$ is the *learning rate* which characterizes the magnitude of the updates with respect to the gradients.

The loss function $\mathcal{L}(\boldsymbol{x}, y; \boldsymbol{\theta})$ should characterize the error of the model with respect to the task it is trained for. For the sake of simplicity, we restrict ourselves to the case of

regression, where the loss function is usually of the form:

$$\mathcal{L}(\boldsymbol{x}, y; \boldsymbol{\theta}) = \frac{1}{2} \sum_{i}^{N} (f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) - y^{(i)})^2, \tag{2.3}$$

where N is the number of examples in the data set and $f(\boldsymbol{x}; \boldsymbol{\theta})$ is the model's *prediction* and $\frac{1}{2}$ is added for mathematical convenience. Evaluating this term repetitively can become computationally expensive in case of large data sets. Moreover, minimizing this term for a train set will not guarantee adequate performance on some unseen *test set*. Ultimately, the model should be able to generalize over unseen data. To ensure stability and convergence during training, the learning rate should generally not exceed a small non-zero constant e.g. 10^{-3}. This can make learning slow, particularly if every update involves computing the entire sum as in Equation 2.3.

2.2.2 Stochastic gradient descent

SGD *approximates* the error gradient by only considering a subset of the training data:

$$\mathcal{L}(\boldsymbol{x}, y; \boldsymbol{\theta}) = \frac{1}{2} \sum_{i}^{M} (f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) - y^{(i)})^2, \tag{2.4}$$

where $M < N$. Originally, the case in which $M = 1$ was referred to as SGD. Nowadays, when $1 < M < N$, it is common to refer to the method as being stochastic batch gradient descent or just SGD. The method is stochastic in the sense that the error gradient is approximated instead of being fully evaluated and in the sense that examples are considered in a random per training epoch. By doing so, the algorithm no longer follows the exact shape of the error surface. It is important to mention that the examples are randomly selected. Note that the method is significantly more efficient, as we only need to evaluate a subset of the data for each update of $\boldsymbol{\theta}$.

2.2.3 RMSprop

The RMSprop algorithm (Tieleman and Hinton, 2012) adapts its gradient updates according to the root of a running average of the square gradient. This means that the gradient updates are given by:

$$\boldsymbol{m} \leftarrow \rho \boldsymbol{m} + (1 - \rho) \boldsymbol{g}_t^2, \tag{2.5}$$

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta \frac{\boldsymbol{g}_t}{\sqrt{\boldsymbol{m} + \epsilon}}, \tag{2.6}$$

Where $\boldsymbol{m}$ is the running average of the squared gradient, ρ is the corresponding decay parameter, $\boldsymbol{g}_t$ is the gradient at time t and ϵ is the fuzz factor that is required for numerical stability. Note that all operations in Equations (2.5) and (2.6) are element-wise. Such adaptive optimizers have become a default choice for optimizing DNNs as they outperform carefully tuned alternatives that use simple SGD.

There are several alternatives to RMSprop that use adaptive learning rates that are omitted for the sake of brevity and because they do not appear elsewhere in this thesis such as Adam (Kingma and Ba, 2014), AdaGrad (Duchi et al., 2011), AdaDelta (Zeiler, 2012), YellowFin (Zhang et al., 2017) or AdaSecant (Gulcehre et al., 2014, 2017).

2.2.4 Backpropagation

The many layers of computation in neural networks means that we can rewrite most gradients as a product of many differentiated terms by means of applying the *chain rule*. Moreover, many terms reappear in the gradients of different weight matrices. Therefore, a lot of computation can be spared by creating an index of already evaluated expressions that might

be reused elsewhere. This is the idea behind the backpropagation algorithm (Rumelhart et al., 1986). For a modern discussion about the implementation of such an algorithm, see chapter 6, section 6.5 of (Goodfellow et al., 2016).

2.3 Convolutional neural networks

Adding many layers can be useful for tackling highly nonlinear problems such as image recognition. Naively stacking layers of neurons does not automatically yield good performance because of potential overfitting. Overfitting occurs when the model becomes too flexible such that the model also describes noise patterns that are not representative of the underlying data distribution, which eventually leads to impeded performance. Specialized architectures such as convolutional neural networks (CNNs) enable many layered computations with proper convergence guarantees and high accuracies. The first description of a modern CNN was posed by LeCun (1989), though many texts discussing the first convolutional networks refer to (LeCun et al., 1998). Such CNNs are related to the Neocognitron architecture (Fukushima, 1980), but the Neocognitron was optimized with a layer-wise unsupervised clustering algorithm. Rather than having fully connected layers in which each hidden neuron is connected to all neurons in the preceding layer, CNNs have layers in which neurons are locally connected. This is similar to how the mammalian brain is organized to process visual stimuli (Hubel and Wiesel, 1962, 1959, 1968). Since the most prominent applications of CNNs are within the field of computer-vision, we will discuss their properties in the context of image recognition. It is important to realize that these properties can also be true for other domains (e.g. time sequences, video data or tactile sensor data), but that they need to be slightly altered to be applicable.

2.3.1 The convolutional layer

From an analytical perspective, convolutional layers (CLs) employ convolution operations instead of a more general matrix-vector product that is used in fully connected layers. There are a number of motivations for using convolutions. First of all, grid-like data such as images often have valuable information that can be extracted locally. A few examples of such local patterns are edges, corners and color transitions. In order to detect these features, we can restrict a hidden neuron to be only connected to a subregion of the image. By doing so, we greatly reduce the amount of parameters of the network with respect to the number of hidden neurons. This reduces the risk of overfitting, as we force the learned representation to be constituted of smaller local features instead of features that are learned globally. Moreover, for image data it is evident that these features can often be detected at almost any position of the image. Hence, it makes sense to share the weights of the neurons in a convolutional layers across the input. This weight sharing additionally reduces the risk of overfitting by further reducing the amount of parameters. Thirdly, using convolutions instead of matrix-vector products is considerably more efficient, especially when computed on specialized hardware such as GPUs or tensor processing units (Jouppi et al., 2017). This speeds up the forward and backward passes, yielding faster training and evaluating.

A CL is typically parameterized by a 4D-tensor $\mathsf{W}_{k,c,i,j}$ in which k is the *kernel* index, c is the *channel* index and i and j are the row and column indexes of the image, respectively. Channels are sometimes referred to as *feature maps*. The input of a CL is also represented as a 4D-tensor $\mathsf{X}_{l,c,i,j}$ with a minibatch index l and similar indices for the remaining dimensions. The output of a CL is also a 4D-tensor $\mathsf{Y}_{l,c,i,j}$ with similar indexing. Usually, convolutions are summed across all channels for each position, which is why a single kernel is represented as a 3D-tensor. The first kernel $\mathsf{W}_{0,c,i,j}$ is used for computing the first feature map of the output tensor $\mathsf{Y}_{l,0,i,j}$, the second kernel is used for the second feature map etc. The output tensor still inherits a grid-like structure. Therefore, multiple convolutional layers can be stacked to form deep CNNs.

Interestingly, the representations that are learned by these hidden CLs automatically

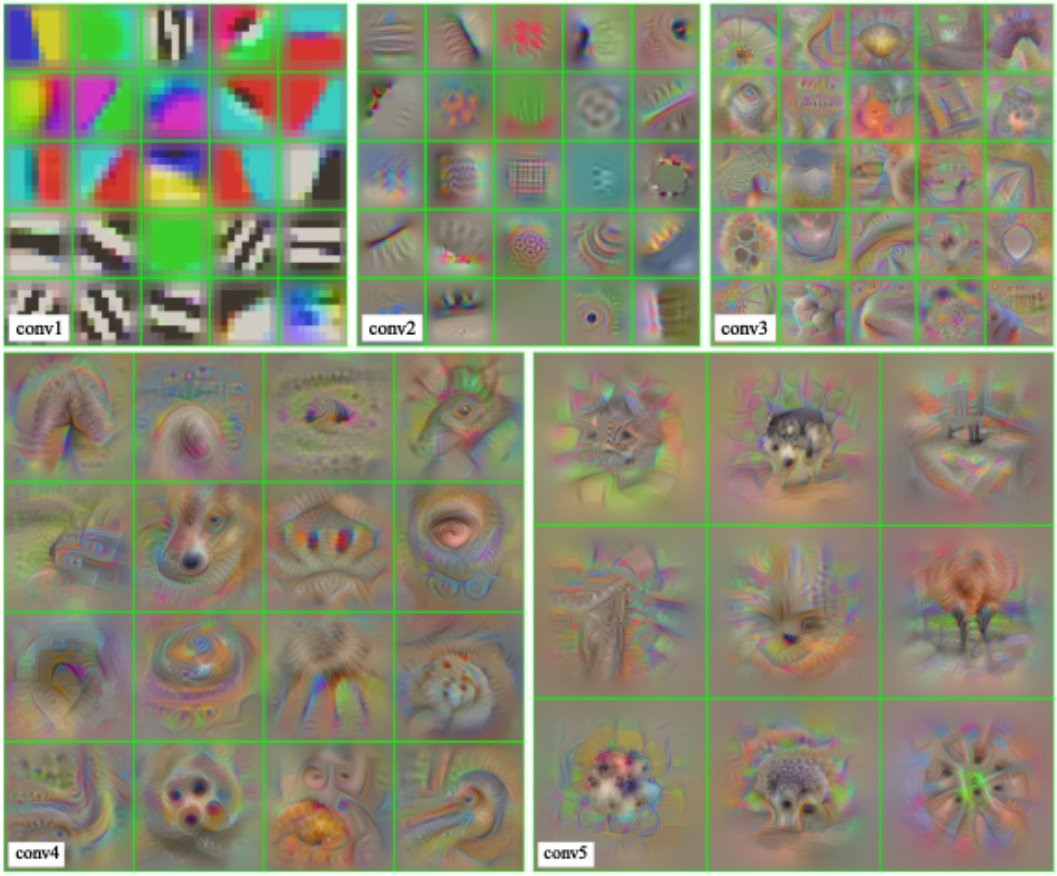


Figure 2.2: Visualization of feature hierarchy that is implicitly learned in a convolutional neural network. Image is taken from (Mahendran and Vedaldi, 2016).

reflect a hierarchical breakdown of the features that are commonly present in images (Zeiler and Fergus, 2014). A visualization of this hierarchy can be found in Figure 2.2. However, this *representational view* is challenged by the fact that highway networks (Srivastava et al., 2015) and residual networks (He et al., 2015a) are insensitive to removing (Srivastava et al., 2015) or shuffling layers (Veit et al., 2016). In (Greff et al., 2016) it is argued that the insensitivity could be explained better by imposing an *unrolled iterative estimation* view. In the latter view, the networks consist of stages in which each state consists of blocks that successively refine the representation of earlier layers. It is also shown that under the corresponding assumptions, residual networks and highway networks can be derived naturally.

2.3.2 Pooling

Another operation that is commonly used in convolutional neural networks is pooling. A pooling operation downsamples the representation in a layer. There are different approaches to pooling such as max-pooling, mean-pooling, L2-norm pooling etc. (Zhou et al., 1988; Goodfellow et al., 2016). The most common method is max-pooling in which the representation is downsampled along the grid-directions of the input. This is accomplished by taking the maximum activation of the output tensor at local $k \times k$ patches. By using pooling, the network becomes less sensitive to small changes of the input such as translation and slight rotations.

However, in principle, pooling operations could make the whole neglect particularly important spatial details. For this reason, pooling operations are avoided when considering problems such as robotic control (Levine et al., 2016) or when learning to play video games based on pixel input (Mnih et al., 2013, 2015).

2.3.3 The full architecture

By combining multiple convolutions and pooling operations, deep CNNs can be efficiently trained on large datasets yielding exceptional performance on a wide variety of tasks. Usually, the last convolutional layer is followed by a few fully connected layers and an output layer. A fully connected layer is just a regular MLP structure in which each neuron is connected to all neurons in the preceding layer. Adding such layers to convolutional neural networks allows for non-local interactions of the features from the convolutional stream.

2.4 Activation functions

As stated before, neural networks use activation functions. These activation functions are often nonlinear such as a hyperbolic tangent function $f = \tanh$, or a sigmoid function $f(x) = 1/(1 + \exp(-x))$. A particularly influential idea was the introduction of the ReLU nonlinearity which is defined as $f(x) = \max\{0, x\}$ (Nair and Hinton, 2010). The ReLU function was designed to overcome the vanishing gradient issue which is caused by the many multiplications that arise in DNNs (Pascanu et al., 2013). A possible cause for gradients to vanish is the fact that activation functions have derivatives that are less than 1. The ReLU's derivative is defined as 1 if $x \geq 0$ and is 0 otherwise. This fosters the propagation of gradients to layers that are close to the input layer. This eventually leads to improved performance.

Since then, the design of a proper activation function has been an actively pursued question, giving rise to many alternatives, such as parametric ReLUs (He et al., 2015b), exponential linear units (Clevert et al., 2015) and scaled exponential linear units (Klambauer et al., 2017).

Chapter 3

Reinforcement Learning

Deep reinforcement learning involves the integration of DNNs into the realm of reinforcement learning (RL). In reinforcement learning problems, an agent is expected to learn to behave optimally given its environment. The environment occasionally provides the agent with rewards which the agent can use to guide its learning process and behavior. Section 3.1 lists the general definitions that we use throughout the remainder of this thesis. The definitions are taken from (Sutton and Barto, 2017). For brevity and clarity, we restrict ourselves to discrete time, discrete action spaces and discrete state spaces. Each of these might individually be altered to its continuous counterpart, but we refrain from further elaboration given that our domain of experiments only requires the discrete definitions.

3.1 General definitions

Formally, we define the following:

State space The state space $\mathcal{S}$ is a finite set of states or a continuous state space. A state is usually the part of the world that the agent observes. In the case of video games, the state space might be given by all different pixel inputs that could be encountered during a game.

Action space The action space $\mathcal{A}$ is a finite set of actions or a continuous action space. These are the actions that the agent can take. For video games, this might correspond to the joystick inputs that are possible. The joystick could in principle be controlled by an actual robot, but it might also be controlled programmatically.

Model An environment's model describes the exact transitions between states and might be conditioned by the agent's actions. Mathematically, this might be written as $\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$. For many problems, the model is not readily available. We will come back to this issue later.

Reward function The reward function $\mathcal{R}$ describes the rewards that are associated with a certain state or a certain state-action pair. Generally speaking, positive rewards *reinforce* the agent to act in a certain way, while negative rewards should discourage the agent to do so. The magnitude of the reward can express the relative importance of different reinforcing or punishing signals.

Return The return G_t is the total discounted reward from time-step t:

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{3.1}$$

where R_{t+1} is the reward at time t and γ is the discount factor which is explained next.

Discount factor The discount factor $\gamma \in [0, 1]$ describes how future rewards are discounted. In the extreme case γ is either 0 or 1. When $\gamma = 0$, the agent considers only the immediate reward, which is also referred to as a strictly *myopic* agent. When $\gamma = 1$ the agent requires the problem to be finite in time, since otherwise the rewards might infinitely accumulate. There are several reasons to have $\gamma < 1$. Firstly, it guarantees the fact that rewards cannot infinitely accumulate, thus avoiding numerical overflow and allowing for easier mathematical analysis. Second, many problems in RL are solved reasonably well when one considers only rewards in the near future. Moreover, the variance of the empirical returns will be lower than the undiscounted case, allowing for more stable training of function approximators.

Policy The policy π of the agent defines how the agent chooses its actions given the observed states. It fully defines the behavior of an agent:

$$\pi(a|s) = \mathbb{P}[A_t = a \mid S_t = s]$$

Ultimately, the agent should acquire the optimal policy. The optimal policy is the policy that maximizes cumulative rewards (the return).

Markov decision process A reinforcement learning task that satisfies the Markov property is called a *Markov decision process* (MDP). A system has the Markov property if the probability distribution of the next state given some current state is fully determined by just the current state and not by any other state i.e. $\mathbb{P}[S_{t+1} \mid S_t, A_t, S_{t-1}, A_{t-1}, \ldots, S_0, A_0] = \mathbb{P}[S_{t+1} \mid S_t, A_t]$. An MDP is defined by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$. Hence the probability transition matrix as introduced above can be defined as $\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a]$.

Value functions We often use value functions to solve reinforcement learning problems. The state-value function $V_\pi(s)$ tells us the expected cumulative reward for being in state s and following policy π. The state-action value function $Q_\pi(s, a)$ gives us the expected cumulative reward for being in state s, taking action a and then following π.

In an MDP, $V_\pi(s)$ is defined as

$$V_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s \right], \tag{3.2}$$

where $\mathbb{E}_\pi[\cdot]$ denotes the expected value of a random variable under the policy π. The definition of $Q_\pi(s, a)$ for an MDP is as follows:

$$Q_\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s, A_t = a \right]. \tag{3.3}$$

Optimality In reinforcement learning, the agent needs to learn optimal behavior. If we consider a policy π and a policy π', we have that $\pi \geq \pi'$ if and only if $V_\pi(s) \geq V_{\pi'}(s)$ for all $s \in \mathcal{S}$. Since there is always at least one policy that is greater than or equal of value compared to all other policies, we can say that this policy is optimal, which we denote by π_*. It is evident that the corresponding value functions need to maximize their values over all policies, meaning:

$$V_*(s) = \max_\pi V_\pi(s), \tag{3.4}$$

$$Q_*(s, a) = \max_\pi Q_\pi(s, a), \tag{3.5}$$

are the *optimal state-value function* and the *optimal state-action value function*, respectively.

3.2 Reinforcement learning algorithms

In order to find the optimal policy, one can use a range of methods. For example, the optimal policy can be found by finding the optimal value function. The simplest of these methods relies on policy evaluation and policy iteration. In policy evaluation, we want to know $V_\pi(s)$ given π for all $s \in \mathcal{S}$. Once we have determined $V_\pi(s)$, we can improve our current policy based on the new estimated values such that $V_{\pi'}(s) \geq V_\pi(s)$ for at least some $s \in \mathcal{S}$.

In practice, policy iteration is rarely suitable for the RL problem at hand. This is because either the model is unavailable, or the state space $\mathcal{S}$ is simply too large for the algorithm to find the optimal solution in reasonable time.

The remainder of this section discusses algorithms that can be applied in a model-free manner. In many cases an environment model is not available. This is either because the true dynamics are unknown or it is too costly to implement. Moreover, model-free algorithms constitute a more general approach. Hence, the developments that can be made toward improving model-free algorithms have the potential to be applied to more problems than improvements that are made on model-based algorithms. Our experiments are restricted to model-free approaches, which is why further discussion of model-based algorithms is not included. The algorithms discussed below are basic and central to RL. The specific notation and naming conventions are taken from (Sutton and Barto, 2017), which can be consulted for further study. For a more detailed overview of relatively recent RL algorithms, see (Wiering and Van Otterlo, 2012).

3.2.1 Monte Carlo evaluation and control

The first step towards a more practical approach is to omit the environment model and learn from actual experience instead. The simplest of these methods is the Monte Carlo evaluation algorithm. In this algorithm, the agent obtains an estimate of a value function by generating a so-called episode that starts at some state S_0 and ends in terminal state S_T. For every state (or state-action pair) that was part of this episode, we obtain an empirical return. This return is then used to update the estimates of the value function.

One can express a Monte Carlo update mathematically:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t)), \tag{3.6}$$

where $\alpha \in [0, 1]$ is a step-size parameter, often referred to as the learning rate. Alternatively, we can use Monte Carlo control. Since we do not have an environment model now, we cannot greedily act with respect to solely $V(s)$. We need to act greedily with respect to $Q(s, a)$. However, if we behave purely greedily by always taking the action that maximizes our expected reward, we are at risk of not exploring the parts of the state space that are potentially much better. A straightforward trick is to act ε-greedily, which means that with a probability of $\varepsilon \in (0, 1]$ we choose a random action.

3.2.2 Temporal difference learning

Instead of generating a full episode from any state S_t for a single update, one can take only a single step and use the estimated value of the next state $V(S_{t+1})$ to update $V(S_t)$. This is the idea behind the TD(0) algorithm. Its update rule is:

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{3.7}$$

It is possible to unify TD(0) with Monte-Carlo by using n-step returns $G_t^{(n)}$. If we then combine all n-step returns we can average them to obtain:

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}, \quad \text{where} \quad G_t^{(n)} = \sum_{k=0}^{n} \gamma^k R_{t+k+1} + \gamma^n V(S_{t+n}) \tag{3.8}$$

where $\lambda \in [0, 1]$. The update rule now becomes:

$$V(S_t) \leftarrow V(S_t) + \alpha\Big(G_t^\lambda - V(S_t)\Big) \tag{3.9}$$

Equation (3.8) can be considered to be the *forward-view* of TD(λ). An alternative approach is to use *eligibility traces*. For every state that is visited, we raise its eligibility, as it has now gained some credit towards the final outcome of this episode. Then at each step we update all $s \in \mathcal{S}$. The updates are then in proportion to the single step TD-error $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ and the eligibility trace.

TD learning can also be applied to control. The TD(0) equivalent of this method is called Sarsa. The generalization to Sarsa(λ) is completely analogous to the extension of TD(0) to TD(λ).

Q-learning (Watkins and Dayan, 1992) differs from Sarsa in the sense that it always chooses the state-action pair that maximizes the bootstrapped value of $Q(s, a)$. Its update rule is given by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{3.10}$$

In this case $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$ can be seen as the target. Note that the targets in Q-learning are greedy, while the agent will select actions in the same ε-greedy way as done in Sarsa. Algorithms that use a different mechanism for selecting targets than for selecting actions are known as *off-policy* methods while the alternative methods that use the same mechanism for both are known as *on-policy* methods.

3.3 Function approximation

Section 3.2 discussed tabular methods for solving reinforcement learning problems. This means that all states and actions are enumerated in a table-like fashion. The downside of table lookup approaches is that there is no generalization across the state space. Hence, in large state spaces it is intractable to use such methods, as there are simply not enough resources to visit all states sufficiently often. For example, the game of Go has approximately 10^{200} different board states. Even a single dynamic programming evaluation iteration would take an exorbitant amount of time.

In such cases, it helps to use function approximators for the value functions e.g. $V(s) \approx V(s; \boldsymbol{\theta})$ and $Q(s, a) \approx Q(s, a; \boldsymbol{\theta})$ where $\boldsymbol{\theta}$ contains the adjustable *parameters*. The function approximators make use of features that are either learned or manually engineered to allow the algorithm to generalize across the state space. By generalizing, the algorithm transfers knowledge learned for a particular state to similar states in the future, without necessarily having seen them before. The next section elaborates on *supervised learning*, a common application of function approximation that is useful for RL methods.

3.3.1 Supervised learning

Supervised learning (SL) is a branch of machine learning that considers problems in which some input is related to a desired target. There is a vast amount of different SL algorithms, too large to list here. For RL, it is relevant to consider *linear* SL methods and *nonlinear* SL methods. Linear methods simply use a linear combination of features that are extracted from some input. By combining features linearly, one can solve regression or classification problems. Note that the $Q(s, a; \boldsymbol{\theta})$ and $V(s; \boldsymbol{\theta})$ functions are real-valued functions and so approximating these functions comes down to a regression problem. Deep reinforcement learning in particular is done with DNNs which are obviously highly nonlinear.

In RL, the simplest function approximator is a linear approximator where

$$v(s; \boldsymbol{\theta}) = \boldsymbol{\phi}(s)^T \boldsymbol{\theta}, \tag{3.11}$$

in which $\boldsymbol{\phi}(s)$ is a feature vector corresponding to the state s. An important property of linear models is that they are guaranteed to converge to a least-squares fit of the actual value function (Tsitsiklis et al., 1997). For neural networks, this guarantee has not been

established and clearly, the optimization process is based on many assumptions that can become challenging to combine with RL. For example, changing the weights at earlier layers changes the input distribution for other layers later on, while there is no explicit mechanism to account for these distribution shifts. Second, the gradients can be inaccurate because the inputs in a batch only make up a small subset of the full input space. Thirdly, the fact that the agent alters its behavior through time causes the input distribution to change as well.

Gradient descent

A very common method for DNN optimization is gradient descent (Cauchy, 1847). In order to apply gradient descent, we need to define the loss-function first. The loss function expresses the performance penalty of our SL model which we seek to minimize. See Section 2.2 for an introduction.

Note that the update rules that we encounter in reinforcement learning (such as Equation (3.9) and (3.10)) can be framed in relation to gradient descent updates. We could define the following parameter updates $\Delta\boldsymbol{\theta}$ for the range of algorithms discussed above:

- For Monte Carlo evaluation we have G_t as the target:

$$\Delta\boldsymbol{\theta} = \alpha\Big(G_t - V(s;\boldsymbol{\theta})\Big)\nabla_{\boldsymbol{\theta}} V(s;\boldsymbol{\theta}) \tag{3.12}$$

- For TD(0) we have $R + \gamma V(s';\boldsymbol{\theta})$ as the target:

$$\Delta\boldsymbol{\theta} = \alpha\Big(R + \gamma V(s';\boldsymbol{\theta}) - V(s;\boldsymbol{\theta})\Big)\nabla_{\boldsymbol{\theta}} V(s;\boldsymbol{\theta}) \tag{3.13}$$

- And for TD(λ) we have the λ-return G_t^λ:

$$\Delta\boldsymbol{\theta} = \alpha\Big(G_t^\lambda - V(s;\boldsymbol{\theta})\Big)\nabla_{\boldsymbol{\theta}} V(s;\boldsymbol{\theta}) \tag{3.14}$$

3.3.2 Policy gradient methods

The algorithms discussed so far optimize performance by finding the optimal value function from which the corresponding policy is derived. *Policy gradient methods* on the other hand, seek to maximize some performance measure with respect to policy weights. In this case we perform gradient ascent:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\nabla\xi(\boldsymbol{\theta}), \tag{3.15}$$

Where α is the learning rate and $\xi(\boldsymbol{\theta})$ is the performance measure. In the case of discrete action spaces a common way to parameterize the policy is to use an exponential softmax distribution:

$$\pi(a \mid s;\boldsymbol{\theta}) = \frac{\exp(h(s,a;\boldsymbol{\theta}))}{\sum_{a'}\exp(h(s,a';\boldsymbol{\theta}))}, \tag{3.16}$$

Where $h(s,a;\boldsymbol{\theta})$ is some function approximator. In a way this method predicts action preferences. A major advantage of doing so is that it might converge to an optimal *stochastic* policy, which is not possible when using e.g. ε-greedy action picking. Moreover, the policy may be a simpler function to estimate than the exact Q-function, as the algorithm now only has to figure out which actions work best, rather than what the expected return is for each action.

Theorem

The *policy gradient theorem* is that (Sutton and Barto, 2017):

$$\nabla\xi(\boldsymbol{\theta}) = \sum_s d_\pi(s)\sum_a Q_\pi(s,a)\nabla_{\boldsymbol{\theta}}\pi(a \mid s;\boldsymbol{\theta}), \tag{3.17}$$

In which $d_\pi(s)$ is the stationary distribution over π. In the episodic case, this corresponds to the expected number of visits in an episode divided by the total number of

states for that episode if we follow π.

The policy gradient theorem provides an analytical expression for the policy gradient which can be used in gradient ascent.

The REINFORCE algorithm

The REINFORCE algorithm (Williams, 1992) is a Monte Carlo policy gradient algorithm. Its update rule is given by:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G_t \nabla_{\boldsymbol{\theta}} \log \pi(A_t \mid S_t, \boldsymbol{\theta}), \tag{3.18}$$

which is motivated by the fact that:

$$\nabla\xi(\boldsymbol{\theta}) = \mathbb{E}_\pi \left[\gamma G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t \mid S_t, \boldsymbol{\theta})}{\pi(A_t \mid S_t, \boldsymbol{\theta})} \right]. \tag{3.19}$$

An extension to this algorithm is to include a baseline that varies with a state. It can be shown that the baseline subtraction does not cause the expected value of the gradient to change as long as it does not vary with a. The update rule now becomes:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t \Big(G_t - b(S_t) \Big) \nabla_{\boldsymbol{\theta}} \log \pi(A_t \mid S_t, \boldsymbol{\theta}) \tag{3.20}$$

A common choice for $b(S_t)$ is to use $V(s; \boldsymbol{w})$, where $\boldsymbol{w}$ is the set of parameters for the critic. This causes the updates to have a lower variance which should improve the stability of the optimization through gradient ascent.

Actor-Critic methods

Actor-critic methods are similar to the REINFORCE algorithm with a value function as a baseline, but are different in the sense that they also bootstrap. The 'actor' in this method is $\pi(a|s; \boldsymbol{\theta})$ and the critic is $V(s; \boldsymbol{w})$. Both the actor and critic learn on-policy.

Chapter 4

State-of-the-Art Deep Reinforcement Learning

The two former chapters have introduced deep learning and reinforcement learning which are the two major components of deep reinforcement learning. Recently, deep neural networks have been successfully implemented in RL approaches. The reason for this delayed introduction of deep learning to the field of RL, is mainly that it was unclear how to ensure that the networks were trained in a stable manner. For linear function approximators, this was not a problem as much, since these functions are guaranteed to converge to their optimal fit of the actual value function they approximate (Tsitsiklis et al., 1997). Through surprisingly modest changes to the way in which these networks were trained, numerous successful applications of DRL have been established and it currently is a popular field of research.

In this chapter, we will discuss the foremost advances in DRL. As the field is still relatively young, we are able to describe most of the major contributions in satisfying detail. First, we will consider the deep Q-network (DQN) by (Mnih et al., 2013). We will see that their research forms the basis of many other improvements as we discuss these in detail. Later on, we elaborate on an actor-critic algorithm that accounts for the basis of our experiments in Part II. For another extensive overview of deep reinforcement learning, see (Li, 2017). There are other sources available that list state-of-the-art reinforcement learning algorithms that are not necessarily combined with deep learning, such as the detailed work by Wiering and Van Otterlo (2012).

The developments regarding DRL are discussed in a (roughly speaking) chronological order. Many of the ideas presented here outperformed former state-of-the-art ideas at the time they were published. If so, we will say they 'outperform the state-of-the-art' while in fact later ideas discussed in this chapter might surpass the particular idea in terms of performance. This structure is mainly intended for brevity, to avoid being repetitive and to limit referring to individual performance differences.

4.1 Deep Q-learning in the arcade learning environment

The influential algorithm proposed by (Mnih et al., 2013) uses Q-learning (Watkins and Dayan, 1992) with a function approximator $Q(s, a; \theta)$ and a replay memory $\mathcal{D}$ which consists of experienced transitions $(S_t, A_t, R_{t+1}, S_{t+1})$. The replayed experience is adopted such that data is reused for learning, rather than training on a single experience only once. In the DQN network, the function approximator is trained to minimize the difference between its prediction and the *target* given in equation 4.1

$$y_t = R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a'; \boldsymbol{\theta}_t) \tag{4.1}$$

In which $R_{t+1} \in \mathbb{R}$ is the immediate reward, $\gamma \in [0, 1]$ is the discount factor, S_{t+1} denotes the new state at time $t + 1$, a' is chosen such that the maximum Q-value as estimated by the DNN is returned and $\boldsymbol{\theta}_t$ is the parameterization of the function approximator. A major problem with using such bootstrapped values of a nonlinear function approximator, is that convergence to a good approximation is rare if it is trained naively. This is gradient descent methods that are commonly used to train neural networks are based on an assumption that the underlying distribution of a function to approximate does not change during the training process. Obviously, if the target network itself is learning, the target distribution changes. Moreover, subsequent observations and valuations of states or actions are highly correlated, which can quickly lead to overfitting, as the data from a single batch is highly biased towards the neighborhood of the current state in the state space. To this end, Mnih et al. proposed to use a separate target network $\boldsymbol{\theta}^-$, which contains snapshots of another DQN that is continuously updated. The parameters $\boldsymbol{\theta}^-$ are only periodically synchronized with $\boldsymbol{\theta}_t$. In other words, the target distribution is more constant compared to the naive Q-learning approach, both by freezing the parameters and by averaging over the already experienced transitions. Another important consideration is to use $|\mathcal{A}|$ different outputs, where each output predicts the Q-value of an action from $\mathcal{A}$. By doing so, only a single forward pass is required to compute all Q-values. This is considerably more efficient than computing a separate forward pass for each action.

Their experiments were performed in the Arcade Learning Environment (Bellemare et al., 2013), which is a collection of Atari 2600 games designed to be a benchmark of artificially intelligent (reinforcement learning) agents of which six different games were considered. Only pixel inputs were used as state observations with some minor preprocessing steps. Since the agent is dealing with grid-like inputs, they employed a CNN as their function approximator with two convolutional layers, another fully connected layer and an output layer as described above. A ReLU activation function was used for all layers. These games employ different scoring systems, which is why all positive rewards were clipped at 1, all negative rewards at -1, and all 0 rewards unchanged. This also eases up the hyperparameter optimization for the algorithms, as the magnitude of the gradients does not vary significantly across games.

In (Mnih et al., 2015) a similar approach was explored on 49 different Atari games. Mnih et al. dove further into stabilizing their network and they made it one layer deeper. First, they increased the size of the network by adding another convolutional layer and increasing the number of hidden neurons in the final fully connected hidden layer. Other than that, they clipped their temporal difference error to be between -1 and 1 by which they argue to improve the stability of the algorithm in terms of hyperparameter sensitivity.

The next subsections discuss several alternative approaches to the ALE which are mostly based on the work by (Mnih et al., 2013, 2015). The ALE in particular is a good candidate to show an algorithm's generality in the sense that no feature-engineering is needed for the algorithms and the same algorithm is applied to up to 49 different Atari games which can be quite different in terms of appearance and complexity. It is important to realize that there are several other platforms available for training reinforcement learning agents from pixel input (Beattie et al., 2016; Wymann et al., 2000; Kempka et al., 2016; Synnaeve et al., 2016). However, we have decided not to discuss the efforts on the latter frameworks in detail, because they are typically more recent and, consequently, they involve less relevant influential literature.

4.2 Reducing overestimations and variance

A problem that is both empirically and theoretically shown to be present in Q-learning is that the Q-values that are learned can be highly overestimating the actual reward, which slows down the learning process. This overestimations have been said to be caused by the fact that the function approximator was not flexible enough (Thrun and Schwartz, 1993), or because of noise (Hasselt, 2010). In (Van Hasselt et al., 2015), it is show that these overestimations can have a considerable negative effect in many cases. In their paper, they

extend on the tabular version of the double Q-learning algorithm (Hasselt, 2010), such that it is applicable to function approximation. In double Q-learning with function approximation, the target is changed to:

$$y_t = R_t + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}'), \tag{4.2}$$

Where $\boldsymbol{\theta}'$ is another set of weights. For each minibatch in training, the role of $\boldsymbol{\theta}$ and $\boldsymbol{\theta}'$ might be randomly switched. They show that adopting double Q-learning exhibits state-of-the-art performance by improving on DQN in the Atari domain.

Another way of reducing overestimations and variance is introduced in (Anschel et al., 2016). They provide theoretical arguments for the reduction of variance through averaging a set of DQN target networks that are simply previously stored checkpoints of the DQN network. They show that their averaged DQN target yields lower value estimates that are typically more stable through time. Moreover, the algorithm exhibits superior performance compared to DQN across a handful of Atari games.

4.3 Prioritized replay memory

Perhaps not surprisingly, the magnitude of gradient descent updates that occur during a training process for a DQN agent vary largely across states, depending on how well the function approximator predicts in that particular part of the state-action space. Therefore, it is likely that there are transitions in the replay memory that are more useful than others, simply because the error in that particular case was larger. This observation was the main motivation behind developing the prioritized experience replay DQN agent (Schaul et al., 2015), partly inspired by the work of Moore and Atkeson (1993). By prioritizing the right transitions, learning can be significantly sped up. To that end, Schaul et al. propose to measure the importance of an update by the magnitude of the temporal difference error. The priority of picking a transition is given by $P(i) = p_i^\alpha / \sum_k p_k^\alpha$ in which p_i^α is the priority of transition i. In their paper, Schaul et al. explore the effectiveness of using proportional prioritizing where $p_i = |\delta_i| + \epsilon$ with $\epsilon > 0$ to make the probability guaranteed to be nonzero or rank-based prioritizing with $p_i = 1/\text{rank}(i)$. They show that their method outperforms the double Q-learning approach from (Van Hasselt et al., 2015) and that rank-based prioritizing works better than proportional prioritizing.

A similar approach is adopted in the work by (Narasimhan et al., 2015). Although their application domain is not the ALE, they employ a DQN with prioritized sampling which was also inspired by the prioritized sweeping method of (Moore and Atkeson, 1993). They distinguish between positive and negative rewards in the replay memory and sample a certain fraction ρ from the positive rewards and $1 - \rho$ from the remaining experiences. Narasimhan et al. also show that using a prioritized experience replay memory can significantly improve the agent's performance.

4.4 Adaptive normalization of targets

As discussed previously, the rewards of the games in Atari were clipped to be in the range of $[-1, 1]$ (Mnih et al., 2013, 2015). By doing so, Mnih et al. were able to find a hyperparameter setting that would yield good performance across almost all 49 Atari games. However, there are a few drawbacks of introducing this clipping mechanism. First of all, it is domain specific. Many Atari games have rewards that go outside the range of $[-1, 1]$. By performing this clipping we are no longer optimizing the sum of rewards directly, but rather indirectly through maximizing the frequency of positive rewards compared to negative rewards. Consider the two episodes of three states in which we obtain the following rewards and have no discount: $\{+2, 0, +2\}, \{+1, +1, +1\}$. Note that the first episode would result in a higher return if no clipping was used, whereas the second episode would be preferred if we do use clipping.

In (Van Hasselt et al., 2016) the authors establish a method to alleviate this dependency with theoretical justifications. By adaptively normalizing the targets and preserving the transformations to exactly reconstruct the actual output, they are able to robustly train the same DQN architecture without reward clipping. Perhaps surprisingly, they find that the improvement in the Atari domain is not consistent across all 49 games. For several games, the normalization strategy yields worse results. The authors suspect that this might be due to the fact that the optimal strategy is sometimes reached sooner when preferring reward frequency (clipped) over the exact reward value (normalized).

4.5 Massive parallelization

In (Nair et al., 2015), the authors propose a distributed architecture named Gorila for massively parallel reinforcement learning. Their architecture consists of (i) actors that have a locally accessible replica of the Q-network, (ii) experience replay memory that contains experiences that are gathered by the actors, (iii) learners that compute the gradients and have a target Q-network and (iv) a parameter server which is a distributed storage of parameters and it is responsible for applying the gradient updates. They not only show a significant speed up of training time, but the method also yields agents that play considerably better when given the same amount of input frames.

4.6 A dueling network architecture

In the work by (Wang et al., 2015), a dueling network architecture is introduced. The dueling network architecture is effectively a Q-network with explicit inheritance of a separate value estimate and an advantage estimate. Interestingly, they also explore a combination of this method and the prioritized replay memory from (Schaul et al., 2015). They show that the combination of these methods yields state-of-the-art performance in the ALE, outperforming all previously discussed methods.

4.7 Encouraging exploration

An important consideration in RL in general is that an agent should have the right balance between *exploration* (covering parts of the state space that are not well known to potentially discover a better policy at the risk of a decreased return), and *exploitation* (acting greedily with respect to the currently obtained value estimates such that the expected reward under that valuation is optimal at the risk of not discovering better alternatives). As pointed out by (Osband et al., 2016), a DQN can suffer from an insufficient exploration strategy. They propose a *bootstrapped* DQN neural network architecture. In their approach, the neural network has the same hidden layers as the standard DQN from (Mnih et al., 2015). However, at the end Osband et al. have K different 'heads' that each have their own Q-value estimates and their own targets. During training, the agent randomly chooses one amongst these heads and executes a full episode. Each experience that is added to the replay buffer is accompanied with a bootstrap mask, which determines which of the K bootstrap heads will be involved in actually updating the parameters based on the new experience. They demonstrate how this technique yields networks with a better exploration, outperforming the DQN form (Mnih et al., 2015) across most games in the ALE.

4.8 Optimality tightening

He et al. (2016) directly address the sparsity and delay of the reward signal in reinforcement learning tasks by augmenting the objective function with some additional terms that consider

long-term future and past rewards as well. By doing so, rewards are propagated faster. To see why, if we look at the standard target of a Q-learning agent we have:

$$y_j^{DQN} = R_j + \gamma \max_a Q(S_{j+1}, a_j; \boldsymbol{\theta}). \tag{4.3}$$

It is evident that information only travels from state s_{j+1} to s_j. For this reason He et al. (2016) propose to make use of the following inequality:

$$y_j^{DQN} = R_j + \gamma \max_a Q(S_{j+1}, a_j; \boldsymbol{\theta}) \geq \cdots \geq \sum_{i=0}^{k} \gamma^i R_{j+i} + \gamma^{k+1} \max_a Q(S_{j+k+1}, a) = L_{j,k}, \tag{4.4}$$

which provides a lower bound of the target $L_{j,k}$. Intuitively, this inequality is valid since the empirical part ($\sum_{i=0}^{k} \gamma^i R_{j+i}$) should not be greater than the difference between the single step bootstrapped return at state j and the discounted return at state S_{j+k+1}. Similarly, they also define an upper bound by looking at preceding rewards. These bounds are then used to minimize the Bellman equation with respect to constraints $Q(s, a; \boldsymbol{\theta}) \geq L_j^{\max} = \max_{k \in \{1,\ldots,K\}} L_{j,k}$ and $Q(s, a; \boldsymbol{\theta}) \leq U_j^{\min} = \min_{k \in \{1,\ldots,K\}}$. They show that across the 49 Atari games, the performance is better than a default DQN (Mnih et al., 2015) and comparable to the double Q-learning method (Van Hasselt et al., 2015) which were both trained for 10 times as long.

4.9 Asynchronous methods

Despite the significant successes that have been obtained through the usage of a DQN with an experience replay memory, the replay memory itself has some disadvantages. First of all, a replay memory requires a larger amount of computer memory. Second, the amount of computation per real interaction is higher and third, it restricts the applicable algorithms to be off-policy RL methods. Mnih et al. (2016) introduce asynchronous algorithms for DRL. By running multiple agents in their own instance of the environment in parallel, one can also decorrelate subsequent gradient updates. This opens the way for on-policy methods such as Sarsa, n-step methods and actor-critic methods. A major disadvantage of using single step methods is that in the case of a reward, only the value of the current state-action pair (s, a) is affected directly, whereas n-step methods directly update the valuation of multiple state-action pairs. Mnih et al. (2016) show that an asynchronous advantage actor-critic (A3C) design outperforms all previously mentioned approaches in the ALE in less training time. Their best performing model uses a similar architecture as (Mnih et al., 2015), but with an additional LSTM layer preceding the output layer. Note that the output layer contains both the critic's output and the actor's output. Other than that, Mnih et al. (2016) explore the A3C model in other domains such as a Labyrinth environment which is made publicly available by (Beattie et al., 2016), the TORCS 3D car racing simulator (Wymann et al., 2000) and MuJoCo (Todorov et al., 2012) which is a physics simulator with continuous control tasks. The wide applicability of their approach supports the notion that the A3C algorithm is a robust DRL method.

There are a number of notable extensions of the A3C method. Jaderberg et al. (2016) show that the performance of A3C can be substantially improved by introducing a set of auxiliary RL tasks. These tasks constitute pseudo-rewards that are given for *feature control* tasks and *pixel control* tasks. Given a set $\mathcal{C}$ of these tasks, let $c \in \mathcal{C}$ be a task from this set and let $\pi^{(c)}$ be the corresponding policy for that task. The objective is now defined as:

$$\arg\max_\theta \mathbb{E}_\pi[R_{1:\infty}] + \lambda_{\mathcal{C}} \sum_{c \in \mathcal{C}} \mathbb{E}_{\pi^{(c)}}[R^{(c)}_{1:\infty}] \tag{4.5}$$

where $R^{(c)}_{t:t+n}$ is the discounted return for the auxiliary reward $r^{(c)}$. For each task c an n-step Q-learning loss is optimized. In the pixel control task an agent is learned to maximize the

pixel change in a set of $n \times n$ non-overlapping cells of the input image. By similar reasoning, they define feature control as a task with the objective to maximize the change of specific hidden neural units. In addition to these control tasks, the agent is given the task of reward prediction, which comes down to a supervised learning task in which the agent predicts $\text{sign}(R_{t+1})$ given a subset of the state history. For improving data-efficiency, they adopt experience replay for the auxiliary tasks as well as value function replay, which is effectively an off-policy value regression that improves the value estimator of the actor-critic network. Experience replay also provides sampling freedom to ensure that different targets for the reward prediction task are equally represented in the training curriculum. The loss function of the UNREAL algorithm is given by:

$$\mathcal{L}_{UNREAL}(\theta) = \mathcal{L}_{\text{A3C}} + \lambda_{\text{VR}}\mathcal{L}_{\text{VR}} + \lambda_{\text{PC}} \sum_{c} \mathcal{L}_{Q}^{(c)} + \lambda_{\text{RP}}\mathcal{L}_{\text{RP}} \tag{4.6}$$

where $\mathcal{L}_{\text{A3C}}$ is the A3C loss, $\lambda_{\text{VR}}\mathcal{L}_{\text{VR}}$ is the weighted value function replay loss, $\lambda_{\text{PC}} \sum_c \mathcal{L}_Q^{(c)}$ is the auxiliary control loss and $\lambda_{\text{RP}}\mathcal{L}_{\text{RP}}$ is the reinforcement signal prediction loss. Although adding these auxiliary tasks yields an agent that learns faster, the newly introduced loss coefficients need careful and costly tuning. Jaderberg et al. (2016) show that reward prediction contributes most to the improvement over A3C, whereas pixel control had the least effect of improvement.

4.10 Policy gradient Q-learning

In (O'Donoghue et al., 2016) the authors establish an analytical connection between action-value and actor-critic methods with entropy regularization as in (Mnih et al., 2016). By combining the merits of Q-learning and policy gradient methods, the authors propose a Policy Gradient and Q-learning (PGQL) method. By including Q-learning, the authors can make use of a replay buffer and prioritized experience replay. They parameterize the Q-estimate as follows:

$$Q(s, a; \boldsymbol{\theta}, \boldsymbol{w}) = \alpha\Big(\log \pi(s, a; \boldsymbol{\theta}) + H^{\pi}(s)\Big) + V(s; \boldsymbol{w}) \tag{4.7}$$

In which H is the regularization entropy, π and $\boldsymbol{\theta}$ and $\boldsymbol{w}$ are the parameters of the policy and value estimates respectively.

4.11 Episodic control

Although part of the research in (Blundell et al., 2016) does not rely on deep learning, their results show that brain inspired algorithms for reinforcement learning can yield agents that perform remarkably well on the Atari domain. The authors propose a new method for reinforcement learning which aims to rapidly memorize previous experiences by explicitly storing feature vectors in a growing Q-table. Whenever the current feature vector already exists in the table, the new Q-value in the table is found by taking the maximum of the old Q value and the currently experienced return. When the current feature vector does not yet exist, the table is extended with this feature vector and the corresponding return. It is important to realize that these greedy updates would result in poor performance in non-deterministic environments.

For determining the Q-values during action selection, the algorithms copies the Q-value from the table when the feature vector is present and otherwise it averages the Q-values of k nearest neighbors. The authors show that both random projections of the raw pixel values and the embeddings obtained from a variational auto encoder (VAE) provide useful feature representations that outperform many previous approaches, especially in the early training episodes. Interestingly, the authors used a discount factor of $\gamma = 1$ allowing for a more direct optimization of the total score.

As an alternative to the nearest neighbor mechanism in (Blundell et al., 2016), the neural episodic control (NEC) as presented in (Pritzel et al., 2017) performs lookups using an attention mechanism that takes Q-values by matching an embedding vector $\boldsymbol{h}$ to key vectors $\boldsymbol{k}_i$. Each key vector is accompanied with a Q value. The Q values from the table are combined by taking a weighted average over the Q values of the 50 nearest key vectors $\boldsymbol{k}_i$ where the weights are given by the normalized distance between $\boldsymbol{h}$ and $\boldsymbol{k}_i$. The embedding vectors are the hidden activations of the last hidden layer of a DNN that has the same architecture as (Mnih et al., 2013). The authors show that NEC outperforms most previously mentioned approaches across 49 Atari games.

Part II

Experiments

Chapter 5

General Implementation

In this chapter we will discuss our general implementation in detail. The material here is general in the sense that all specific experiments and research questions depend on the design decisions discussed here.

5.1 Asynchronous advantage actor-critic

The baseline RL model for our experiments will be the asynchronous advantage actor-critic method (A3C) as proposed by (Mnih et al., 2016). This model has already been introduced briefly in Section 4.9. Since this model is so important to our experiments here, we will now cover it in detail.

The A3C method is asynchronous in the sense that multiple RL agents are deployed in their own instance of the environment. Each agent has its own 'local' copy of a deep neural network (which is discussed further in Section 5.2). The agent itself runs an actor learner thread that is similar to the standard advantage actor-critic algorithm. An important difference is that the parameter updates that are computed by the agent are sent to a global network that is shared between all agents. Note that by the definition taken from (Sutton and Barto, 2017), an actor-critic algorithm is also an n-step method. Similar to the approach that is taken in (Mnih et al., 2016), the baseline model will be implemented with the forward-view of n-step returns. An actor acts for five steps after which a parameter update is computed for the currently obtained batch. This parameter update is then sent to the global network. As pointed out by Mnih et al., the global network update can be done without actual thread locking. Although this might entail occasionally unpredictable behavior of gradient updates, they empirically found that this effect was more than compensated by the relative speed up due to less thread handling overhead.

The A3C learner thread algorithm is listed in Algorithm 1. Note that T_{max} is a hyper-parameter that is specified beforehand. There is an additional term $\beta \nabla_{\boldsymbol{\theta}'} H(\pi(s_t; \boldsymbol{\theta}'))$ added to the gradient update accumulate $d\boldsymbol{\theta}$ of the policy network, where $H(\pi(s_t; \boldsymbol{\theta}'))$ denotes the entropy of the policy and β is a weighting factor for the entropy loss. This is suggested by (Mnih et al., 2016), as it encourages exploration.

5.2 Deep neural network

For most of our experiments, we will adopt the architecture as presented by (Mnih et al., 2013). This architecture may have been outperformed by the slightly deeper neural network by (Mnih et al., 2015), but this is solely true for deep Q-learning. The A3C method has been shown to outperform the deeper architecture from (Mnih et al., 2013).

Algorithm 1 Asynchronous advantage actor-learner thread

// *Assume global shared parameter vectors* $\boldsymbol{\theta}$ *and* $\boldsymbol{\theta}_V$ *and global shared counter* $T = 0$
// *Assume thread-specific parameter vectors* $\boldsymbol{\theta}'$ *and* $\boldsymbol{\theta}'_V$
repeat
    Reset gradient updates: $d\boldsymbol{\theta} \leftarrow 0$ and $d\boldsymbol{\theta}_V \leftarrow 0$.
    Synchronize thread-specific parameters $\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta}$ and $\boldsymbol{\theta}'_V \leftarrow \boldsymbol{\theta}_V$
    $t_{start} = t$
    Get state s_t
    repeat
        Perform a_t according to policy $\pi(a_t|s_t;\boldsymbol{\theta}')$
        Receive reward R_t and new state s_{t+1}
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    until terminal s_t **or** $t - t_{start} == t_{max}$
    $G = \begin{cases} 0 & \text{for nonterminal } s_t \\ V(s_t;\boldsymbol{\theta}'_V) & \text{for non-terminal } s_t \text{ i.e. bootstrap from last state} \end{cases}$
    for $i \in \{t-1, \ldots, t_{start}\}$ **do**
        $G \leftarrow R_i + \gamma G$
        $d\boldsymbol{\theta} \leftarrow d\boldsymbol{\theta} + \nabla_{\boldsymbol{\theta}'} \log \pi(a_i|s_i;\boldsymbol{\theta}')(R - V(s_i;\boldsymbol{\theta}'_V)) + \beta \nabla_{\boldsymbol{\theta}'} H(\pi(s_t;\boldsymbol{\theta}'))$
        $d\boldsymbol{\theta}_V \leftarrow d\boldsymbol{\theta}_V + \lambda_V \partial (G - V(s_i;\boldsymbol{\theta}'_V))^2 / \partial \boldsymbol{\theta}'_V$
    Perform asynchronous update of $\boldsymbol{\theta}$ using $d\boldsymbol{\theta}$ and $\boldsymbol{\theta}_V$ using $d\boldsymbol{\theta}_V$
until $T > T_{max}$

5.2.1 Neural network architecture

The simplest architecture that we consider in our experiments is the architecture as shown in Figure 5.1. This architecture is feed-forward, so it does not have any recurrent layers. The first hidden layer is a convolutional layer with 8×8 convolutions and a stride of 4×4 with 32 kernels. The second hidden layer is also a convolutional layer with 4×4 convolutions and a stride of 2×2 and contains 64 kernels. The third hidden layer is fully connected, contains 256 neurons and takes the flattened output volume of the second layer as input. The final hidden layer is followed by two output streams: one for the actor and one for the critic. The actor's output is a softmax policy output with $|\mathcal{A}|$ neurons which gives the probability of performing each action and the second output is the value approximator (which is obviously linear). All hidden layers use ReLU activation functions (Nair and Hinton, 2010).

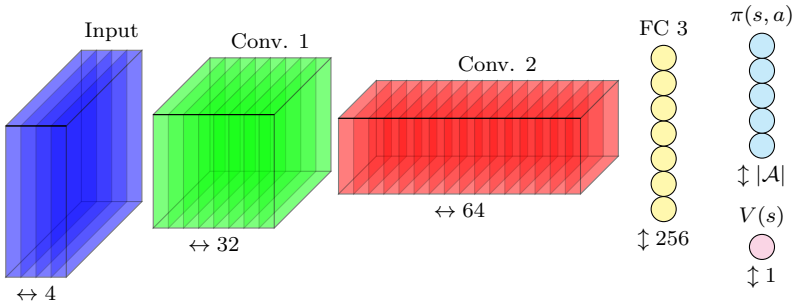
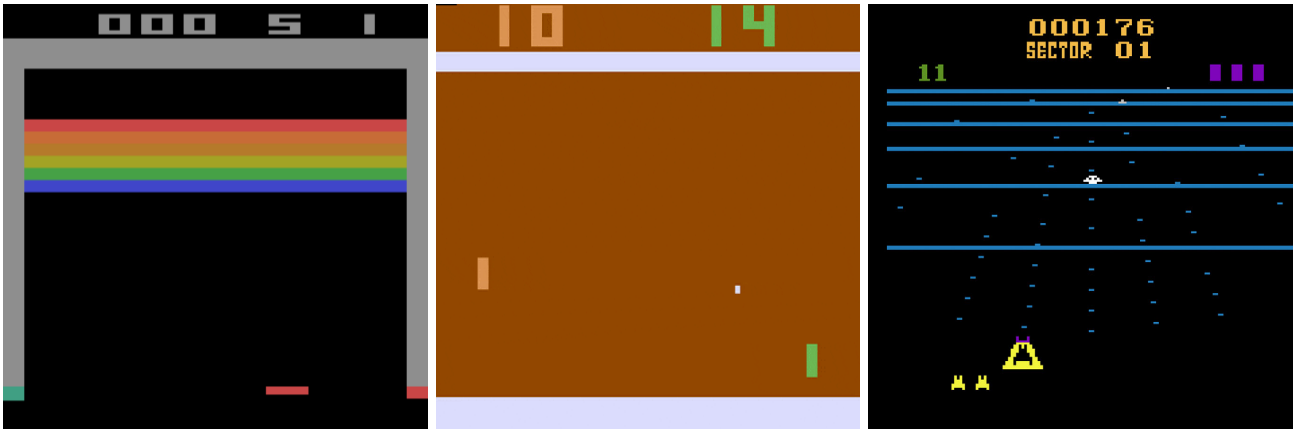


Figure 5.1: A schematic view of the A3C FF network. All hidden layers use ReLU activation functions, while the value function uses a linear output activation function and a softmax is applied to the policy output. The kernel sizes are 8×8 and 4×4 for the first and second layer, respectively. The corresponding strides are 4×4 and 2×2.



(a) Breakout (b) Pong (c) Beam Rider

Figure 5.2: Screenshots of Atari games that were considered in our experiments. When viewed in Adobe Acrobat Reader, Pong and Breakout are animated.

5.2.2 Optimization

Our network will be optimized using the RMSprop algorithm (Tieleman and Hinton, 2012) in which we share the running averaged gradient across all learner threads. This decision is motivated by the observations done by (Mnih et al., 2016), who concluded that this yielded better performance than using a separate RMSprop optimizer for each actor-learner thread. The equations for this algorithm are listed in Section 2.2.3.

5.3 Arcade environment interface

Part of our experiments will be done on a subset of Atari 2600 games from the Arcade Learning Environment (ALE) (Bellemare et al., 2013). Since our implementation is written in Python code, we adopted the `gym` package as released by (Brockman et al., 2016). This package provides a high level programming interface to the Atari 2600 games. We follow previously discussed approaches by using action repeats. This means that in order to simplify the environment interaction and to decrease the computational load of the algorithm, we choose to repeat each action for 4 frames after picking it according to our current policy. Other than that we perform some simple preprocessing. A state is given by taking the last k frames. For each of these frames we first deal with some Atari emulator artifacts that cause objects to be invisible in some frames by taking the max pixel value of the current frame and the previous frame for each pixel location. Then, we extract the luminance channel and resize the 160×120 input frames to 84×84. The state is given by concatenating the k resulting preprocessed frames. Figure 5.2 displays the subset of games that are considered in the experiments. Both Breakout and Pong are animated when viewed in Adobe Acrobat Reader[1].

5.4 A simple game for fast experimenting

For a lot of the Atari games that are available through the ALE, convergence towards a good policy can take a significant amount of time, often requiring over 30 million frames. For testing and hyperparameter tuning purposes, we have implemented the game Catch as described in (Mnih et al., 2014). In our version of the game, the agent only has to catch a ball that falls from top to bottom, potentially bouncing off walls. The world is a

[1] https://get.adobe.com/nl/reader/

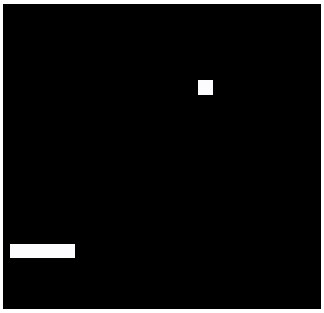


Figure 5.3: Image of Catch game. To see the animated version, use Adobe Acrobat Reader.

Table 5.1: Overview of default settings for hyperparameters as used in our experiments.

Name	Symbol	Value
Learning rate	η	$7 \cdot 10^{-4}$
Discount factor	γ	0.99
RMSprop root mean square decay	ρ	0.99
RMSprop stability constant	ϵ	0.1
Entropy factor	β	0.01
Value loss factor	λ_V	0.5
Steps lookahead	t_{max}	20
Total steps ALE	T_{max}	$1 \cdot 10^8$
Total steps Catch	T_{max}	$1 \cdot 10^6$
Frames per observation	NA	4
Action repeat ALE	NA	4
Action repeat Catch	NA	1
Number of threads	NA	12

24×24 grid where the ball has a vertical speed of $v_y = -1$ $cell/s$ and a horizontal speed of $v_x \in \{-2, -1, 0, 1, 2\}$. If the agent catches the ball by moving a little bar in between the ball and the bottom of the world, the episode ends and the agent receives a reward of $+1$. If the agent misses the ball, the agent obtains a zero reward. With such a game, a typical run of the A3C algorithm only requires about 15 minutes of training time to reach a proper policy. For our experiments, we resize the 24×24 world to have the size of 84×84 pixels per frame. See Figure 5.3 for an impression of the game.

5.5 Default hyperparameters

There are many hyperparameters that need to be set. Table 5.1 provides an overview. Most of these parameters were copied from (Mnih et al., 2016) with the exception of the number of threads and the amount of lookahead steps, which is motivated by the configurations in (Jaderberg et al., 2016).

Chapter 6

Neural Designs for Deep Reinforcement Learning

An appealing property of DRL algorithms as opposed to many other RL approaches, is that DRL algorithms require no manual feature engineering. On the other hand, manually engineered features can be more appropriate for the task at hand. Moreover, they can be combined with linear function approximators, so that the algorithm as a whole is guaranteed to converge under the right theoretical conditions (Tsitsiklis et al., 1997). Since the introduction of the DQN algorithm, many other ideas in DRL have been explored (see also Chapter 4). However, to the best of our knowledge, there has not been an extensive comparison of DNN architectures for DRL. This chapter will attempt to characterize the dependency of an agent's performance on DNN design decisions. We will address the first main research question of this thesis: *To what extent do architectural design decisions and hyperparameters of an agent's deep neural network affect the resulting performance?*

Note on experiment setup This chapter is structured by discussing each variation to the default A3C FF architecture briefly together with the outcomes of one or more parameter sweeps. The parameter sweeps required fast experimenting, which is why in these sweeps Catch (see Section 5.4) was used as the test-bed. All parameter sweep plots are created with at least 50 to a 100 different runs. For each run the parameter was either sampled from a uniform distribution or a log-uniform distribution. In the concluding section of this chapter, we compare a few architectural designs on a small set of Atari games. The exact configuration of these alternative architectures are motivated by the outcomes of the parameter sweeps on Catch.

It is important to stress that the equations in this chapter do not include a batch index for the tensors, as the batch index does not contribute to further explanation of the ideas introduced here and might even be distractive for that reason.

6.1 Local weight sharing

Intuitively, the idea of shared weights in the visual cortex seems to be extremely restrictive and implausible to be identical to the 'hard' weight sharing that can be found in convolutional neural networks. Especially the fact that biological processes would be able to govern the synapses by such extent that they are exactly the same across all receptive fields of the visual cortex seems a questionable assumption.

Alternatively, one could use locally connected layers (van den Dries and Wiering, 2012; Taigman et al., 2014). Locally connected layers also use local connections such as those found in convolutional layers. However, in locally connected layers, there is no weight sharing between these neurons. This allows for local specialization that can be advantageous if the spatial structure of the data remains constant through the full input space. In (Taigman

et al., 2014) such consistency is ensured by considering face verification in which they use a frontalization algorithm to preprocess images of faces such that the faces all appear to be facing the camera. Taigman et al. (2014) argue that their locally connected layers yield improved performance over convolutional layers or fully connected layers. Presumably, this improvement is due to the local specialization of hidden neurons that would have been impossible with convolutional layers given their explicit weight sharing. In (van den Dries and Wiering, 2012), local connections were used so that certain hidden neurons consider only a subset of inputs for an Othello playing agent. In that case the local connections were mainly intended for reducing the amount of parameters.

On the other hand, using locally connected layers clearly has two disadvantages compared to convolutional layers: they have more parameters which makes them prone to overfitting and the fact that these are locally unique prohibits the use of a convolution operation to compute the activations efficiently. To this end, we now propose a novel layer that can be seen as a compromise between a convolutional layer and a locally connected layer which we call a *local weight sharing layer*.

6.1.1 Local weight sharing layer

Instead of the global weight sharing that forces all weights to be exactly the same irrespective of the spatial location which is normally found in convolutional layers, a local weight sharing layer uses kernels that are each given a *centroid*. For brevity, we will refer to these units as kernel centroid pairs (KCP). We want the activations of neurons near a KCP's centroid to be close to those that would have been obtained when using the KCP's kernel, and less like the activations that would have been obtained when using the kernels of more distant KCPs. Additionally, when a spatial location is somewhere in between two centroids, it should inherit some of the weights of both kernel centroid pairs.

We can accomplish this by defining a similarity function that is defined on the spatial domain of the convolutional output. The column and row indices will be translated to Cartesian coordinates which will be used to compute local weighting coefficients. In a local weight sharing layer, each KCP first computes its activation, just like a single convolutional layer would do. Then, we linearly combine the results of these KCPs by using spatial coefficients which are determined by the spatial cells in the output tensors with respect to the centroids of each KCP. As a result, when a spatial cell is close to a certain KCPs centroid, its local kernel will look most like the kernel of that particular KCP. Note that each locally determined output is still factored by several convolution kernels. Hence, to a certain extent, one could regard this as soft weight sharing or distance based weight sharing. This should result in a gradual change of local kernels when moving from centroid to centroid, allowing the features to be significantly more complex with relatively few extra parameters. Mathematically, we define the output of a spatial weight sharing layer as follows (where we do not include a batch dimension):

$$\mathsf{Y} = f\left(\sum_{s=0}^{S-1}\left(\mathsf{W}_s * \mathsf{X} \oplus \boldsymbol{b}_s\right) \odot \mathsf{S}_s \oslash \bar{\mathsf{S}}\right), \tag{6.1}$$

where Y is the output tensor, f is the activation function, W_s are rank 3 weight tensors, X is the rank 4 input tensor, S is a rank 4 similarity tensor with $\mathsf{S}_{b,i,j,c} = \varsigma(i, j, i_s, j_s)$ where (i_s, j_s) parameterizes the centroid of the feature map in the case of feature-wise local sharing or the centroid of the kernel group in case of group-wise local sharing. Note that $\oplus$, $\odot$ and $\oslash$ denote element-wise addition, multiplication and division with optional broadcasting along singleton dimensions. The function ς gives the 'similarity' of a spatial cell located at (i, j) with respect to some convolution's centroid (i_s, j_s). Finally, the $\bar{\mathsf{S}}$ tensor contains the sum of the similarities of all KCPs, such that we still preserve the spatial and feature dimensions. By dividing element-wise by $\bar{\mathsf{S}}$, the contributions of all KCPs always add up to 1 for each neuron in the output volume.

The function ς is defined as:

$$\varsigma(i, j, i_s, j_s) = \exp\big(-(\boldsymbol{x} - \boldsymbol{\mu}_s)^\top \Sigma (\boldsymbol{x} - \boldsymbol{\mu}_s)\big), \tag{6.2}$$

where $\boldsymbol{\mu}_s = [i_s, j_s]^\top$ and $\boldsymbol{x} = [i, j]^\top$. The Σ matrix is initially set to the identity matrix I_2. When we allow the centroids $\boldsymbol{\mu}_s$ to be trainable in our experiments, we also allow the elements of Σ to be trained through gradient descent.

For a graphical illustration of what happens before the activation function is applied, see Figure 6.1. The image displayed here corresponds to a feature-wise configuration for S.

We perform experiments with feature wise configurations as shown here, but we also investigate KCP group-wise configurations. In the KCP group-wise configuration, there would only be a single surface plot for the red and blue convolution features in Figure 6.1. This means that all KCPs in a kernel group have the same centroid. It is interesting to see whether the network can also learn optimal centroids by also letting (i_s, j_s) and Σ be trainable parameters.

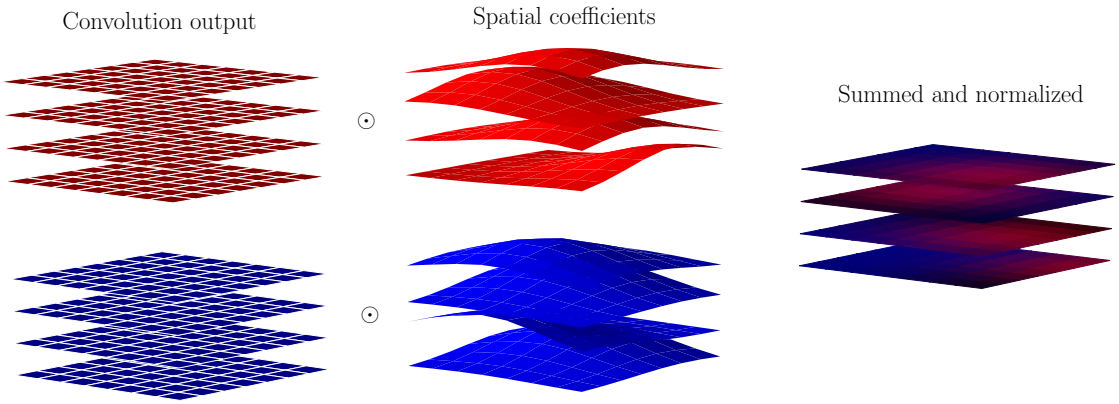


Figure 6.1: Graphical depiction of what is computed inside a spatial weight sharing layer. For simplicity, we have drawn the situation for only two sets of kernel centroid pairs in a feature-wise configuration. First we have the two output volumes of convolutions on the left. These are multiplied element-wise with tensors that contain the spatial weight sharing coefficients, denoted S_s in equation 6.1 and depicted with surface plots in the middle of the figure. On the right we see a visualization of the spatially normalized output, which is obtained through the element-wise multiplication with S_s, followed by summing over all s element-wise and normalizing with $\bar{\mathsf{S}}$ (see equation (6.1)).

6.1.2 LWS architectures

In our experiments, we consider three different architectures. All LWS architectures use LWS layers with 3 randomly initialized groups of KCPs. The first architecture also uses trainable centroid locations (i_s, j_s) and a feature-wise configuration within a KCP (LWS + T + FW), the second architecture uses a per-feature configuration without the trainable centroids (LWS + FW). The third architecture uses a group-wise configuration within a KCP (LWS + T + GW). For all LWS architectures, we have replaced the convolutional layers as found in the default architecture for A3C FF with LWS layers. The first LWS layer of the network has 32 output feature maps, meaning that there are $32 \cdot 3$ convolution kernels of dimensions $8 \times 8 \times 4$. The second LWS layer of the network has 64 output feature maps, meaning that there are another $64 \cdot 3$ kernels of dimensions $4 \times 4 \times 32$. The default A3C FF architecture has 1,336,320+256$|\mathcal{A}|$ parameters, whereas the LWS architectures have

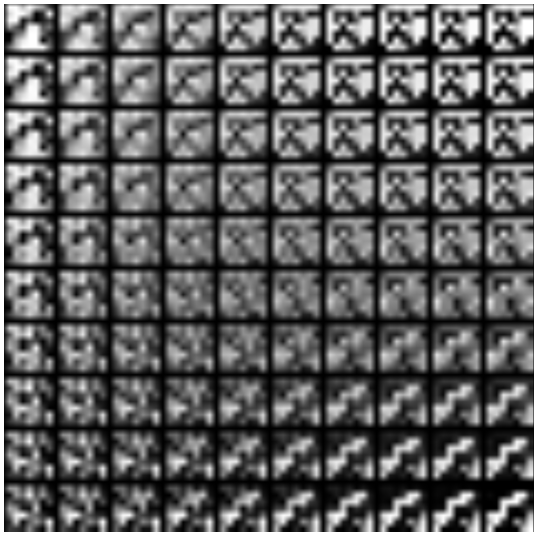


Figure 6.2: Locally weighted kernels from a spatial soft weight sharing layer where the centroids were allowed to vary.

1,449,984+256$|\mathcal{A}|$ parameters (excluding the kernel centroids). Although the amount of kernels is 3 times as large, the dimensions of the LWS layer's output volume is the same as that of the convolutional layers. Since the majority of the weights is at the fully connected layers, the LWS architectures have only slightly more parameters.

6.1.3 Performance analysis on Catch

One of the motivations behind the idea of the spatial weight sharing layers is that neurons can locally specialize for certain kinds of input patterns. For such local specializations of neurons to be advantageous, we require the input data to have the same spatial structure throughout the whole dataset. Fortunately, the frames from a single Atari game have this property to a certain extent. For example, in the game Breakout, the player's bar with which he/she should hit the ball is always at the bottom of the screen. We also tested our local weight sharing layer on gender recognition using the Adience dataset (Levi and Hassner, 2015). The corresponding results are listed in Appendix B and suggest that these layers can yield superior performance when compared to convolutional layers for such tasks.

Figure 6.3 provides the results of a learning rate sweep on the Catch game. For each of the learning rate sweeps that follow, similar extrema were used for the parameters. In the case of the learning rate η, we sampled a learning rate randomly from a log uniform distribution between 10^{-6} and 10^{-2}. It can be seen that in the range above 10^{-3}, the LWS models seem to slightly outperform the default A3C FF architecture. Apparently this architecture is slightly more robust to higher learning rate settings. This could be explained by the fact that the random initialization of the kernels will cause the KCP activations to cancel out when computing the sum per spatial cell. This will then yield activations that are typically lower on average than a convolutional layer would have. Apart from that, the slight improvement might be the consequence of using a more flexible layer that allows for spatial specialization.

6.2 Spatial softmax

A spatial softmax layer was initially proposed in the context of deep learning for robotic control (Levine et al., 2016). This layer downsamples a feature map into coordinates in Cartesian space (x_k, y_k). It accomplishes this by computing a softmax output per feature

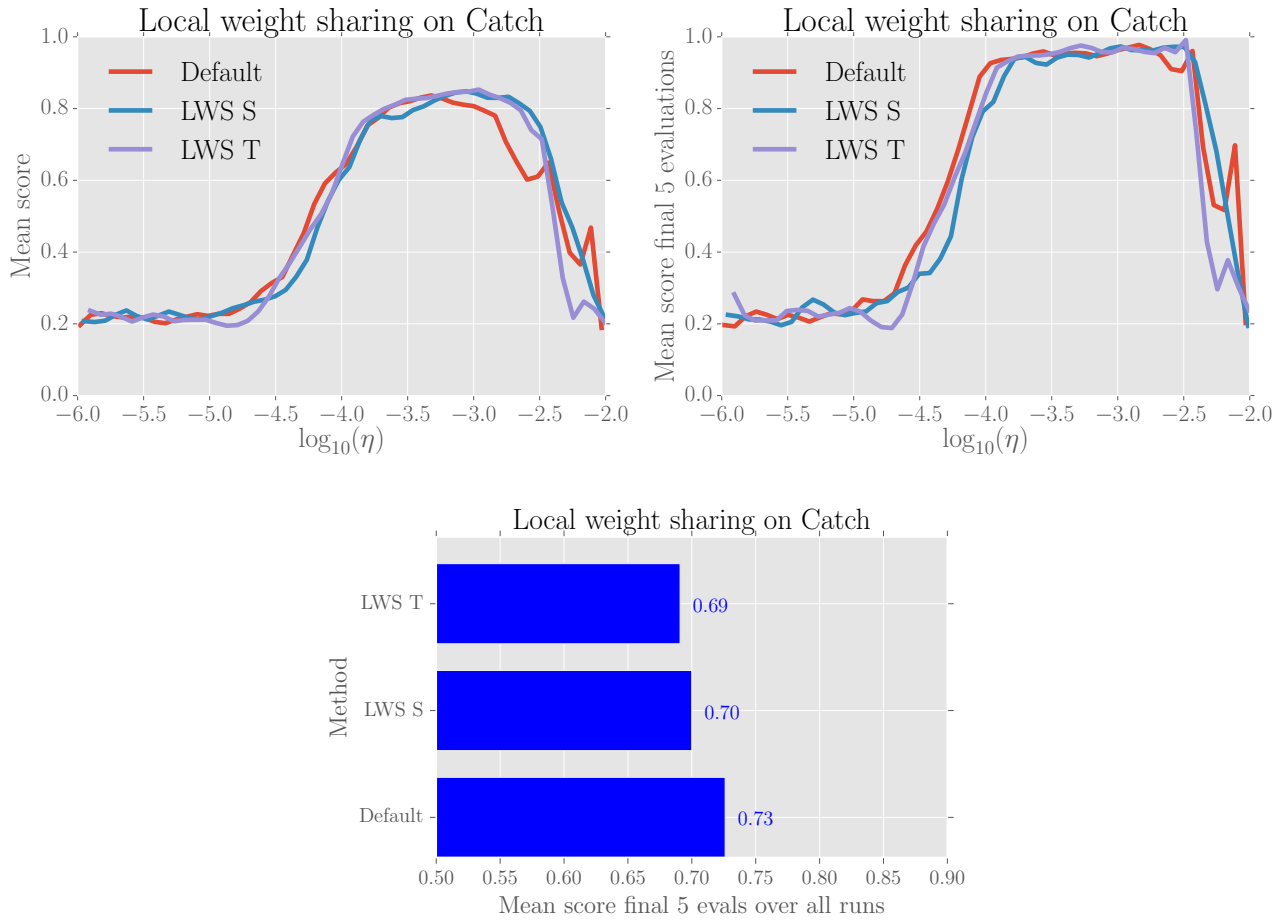


Figure 6.3: Comparison of local weight sharing models vs. default A3C FF model. The plot shows a smoothed curve for the mean score on 20 evaluations (upper left) and the mean score on the final 5 evaluations (upper right) taken during the training of 1 million frames for different learning rates. The bar plot below reports the mean score of the final 5 evaluations averaged over all learning rates between 10^{-5} and 10^{-2}. It can be seen that the performance of the LWS model is comparable. for the higher learning rates, the LWS models tend to perform better than the default A3C FF architecture.

map, after which the softmax output is multiplied by a mesh grid that contains the coordinates (x_{ij}, y_{ij}) for the spatial locations in the feature map. In other words, for each output feature map F_k, we compute the softmax $\psi(\mathsf{F}_k)$. Then, we compute the layer's output as:

$$\begin{pmatrix} \mathsf{Y}_{2k} \\ \mathsf{Y}_{2k+1} \end{pmatrix} = \begin{pmatrix} \sum_{ij}(\psi(F_k) \odot \mathsf{C}_x)_{ij} \\ \sum_{ij}(\psi(F_k) \odot \mathsf{C}_y)_{ij} \end{pmatrix} \qquad \text{for } k = 1, \ldots, K, \tag{6.3}$$

where the output tensor Y is a 1D concatenated array containing all pairs of $\mathsf{Y}_{2k}, \mathsf{Y}_{2k+1}$ and the row and column indices are denoted i and j respectively. The C_x and C_y tensors contain the Cartesian coordinates for the x and y axis, respectively. Finally, K is the total number of features in the layer's input. In our implementation, the Cartesian coordinates are placed at equidistant points in the range of $[-0.5, 0.5]$ for both axes. The softmax function will be applied feature-wise, so it can be expressed as:

$$\psi(\mathsf{F}_k)_{ij} = \frac{\exp(\mathsf{F}_{kij}/\tau_k)}{\sum_{uv} \exp(\mathsf{F}_{kuv}/\tau_k)} \qquad \text{for } k = 1, \ldots, K \tag{6.4}$$

(a) Input (b) Linear convolution (c) Softmax output (d) Cartesian coordinates

Figure 6.4: Schematic visualization of the processing in a spatial softmax layer. The order of transformations can be read from left to right. Note that this process occurs for all feature maps. The image is a frame taken from the BeamRider game.

in which i and u are row indices, j and v are column indices and τ_k is the softmax temperature parameter that controls the steepness of the function. The different phases in the computation described by Equations (6.3) and (6.4) are visualized in Figure 6.4. In our experiments, we consider different configurations for this τ_k parameter. Note that this parameter is subscripted with k and so in principle we allow each feature to have its own temperature. In our experiments, we allow τ_k to be trained by gradient descent.

Intuitively, these layers will be especially useful in situations where the relative position of different objects in the input frames will be crucial for determining the optimal policy. Therefore, they seem to be a good candidate component of a DNN for an actor-critic agent in the ALE environment.

6.2.1 Spatial softmax architectures

We consider several designs for DNNs that use spatial softmax layers. First of all, we have a model in which the second convolutional layer is replaced by a spatial softmax layer. The remainder of the architecture is identical to that of A3C FF. We refer to this architecture as A3C SS (spatial softmax). A second architecture draws inspiration from the dedicated dorsal and ventral streams of information in the visual cortical areas of the human brain (Mishkin et al., 1983; Ungerleider and Haxby, 1994). The ventral stream is also referred to as the *what* pathway and the dorsal stream as the *where* pathway. The ventral stream is mostly responsible for object recognition and requires high resolution representations. These are typically provided by default convolutional layers. The dorsal stream is mostly responsible for the spatial aspects of vision, such as the relative position of salient objects in the scene. Such representations could be provided by the spatial softmax operation. The novel architecture is referred to as the A3C WW (what and where). It applies a spatial softmax operation to 32 of the 64 feature maps and a ReLU nonlinearity to the other 32 feature maps. The remainder of the model is identical to A3C FF. Other than that we explore a few variations on the A3C SS model. We will consider a model with trainable temperature for every feature map (A3C SS TT) and a model with a global temperature that is trainable (A3C SS GT TT).

Performance analysis on Catch

Similar to what we have seen in Section 6.1.3, we conducted a learning rate sweep on the simple Catch game. The results are given in Figure 6.5. It can be seen that the A3C WW model outperforms the other architectures except for A3C FF. When compared to FF, the WW architecture tends to perform marginally worse when the learning rate is higher than

0.001. We can see that using feature specific temperatures (A3C SS TT) yields improved performance compared to global temperatures (A3C SS GT TT).

6.3 Hyperparameter sensitivity

In this section we look at several experiments where we assess the influence of hyperparameters such as activation functions, weight initializations, bias initializations, clipping methods and more.

6.3.1 Gradient clipping

An important element of the DRL agents that we discussed is gradient clipping. Especially when combined with RNNs such as LSTMs (Hochreiter and Schmidhuber, 1997), gradient clipping can help to alleviate the problem of exploding gradients (Pascanu et al., 2013). By default, the A3C architectures use gradient clipping by value, meaning that every scalar element g_i of a gradient $\boldsymbol{g}$ is replaced with $\min\{c_v, g_i\}$ for positive g_i and replaced with $\max\{-c_v, g_i\}$ for negative g_i, where c_v is a clipping bound. While this is an efficient way of clipping, it might alter the direction of a gradient severely, potentially leading to catastrophic divergence. To this end, we compared clipping by value to global norm clipping as advocated in (Pascanu et al., 2013). Global norm clipping limits the magnitude of the gradients by rescaling the gradients with $c_g / \max\{c_g, \sqrt{\sum_i \|\boldsymbol{g}_i\|_2^2}\}$ for all gradients, where $\boldsymbol{g}_i$ can be the vector representation of the gradient of any tensor and c_g is a global clipping constant. Although it is more expensive to compute, global norm clipping does not alter the direction of the gradients. The particular choice for the global clipping constant $c_g = 25$ was found using a parameter search given in Appendix A. The per-value clipping was set at $c_v = 40$.

Results The resulting comparison is displayed in Figure 6.6. It can be seen that despite the attempt to find a proper value for c_g, the global norm clipping configuration performs worse than per-value clipping. This is mainly true for lower learning rates, suggesting that the clipping is applied more than necessary. It was decided not to look for further optimizations, since it was unlikely to yield substantial improvements given the results obtained here and in Appendix A. However, it is still interesting to see what this difference is when using recurrent neural networks such as the A3C LSTM architecture since exploding gradient problems are more often observed for such architectures.

6.3.2 Activation functions

After a quick analysis of some of the fully trained agents on the Atari games, it became clear that some features in the first hidden layer would be completely zero while playing the game. This is presumably caused by an unfortunate initialization of the weights combined with the fact that ReLU units can have zero gradients. Therefore, it makes sense to employ different activation functions that do not come with such issues. It is interesting to see whether adopting the exponential linear unit (ELU) would alleviate the zero-feature map issue. This activation function was introduced by Clevert et al. (2015). They show that using the ELU unit can yield comparable or even superior performance when compared to deep CNNs with (relatively computationally expensive) batch normalization (Ioffe and Szegedy, 2015) and ReLU activations Nair and Hinton (2010). The ELU activation is given by:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}, \tag{6.5}$$

and its derivative is defined as:

$$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ f(x) + \alpha & \text{if } x \leq 0 \end{cases}. \tag{6.6}$$

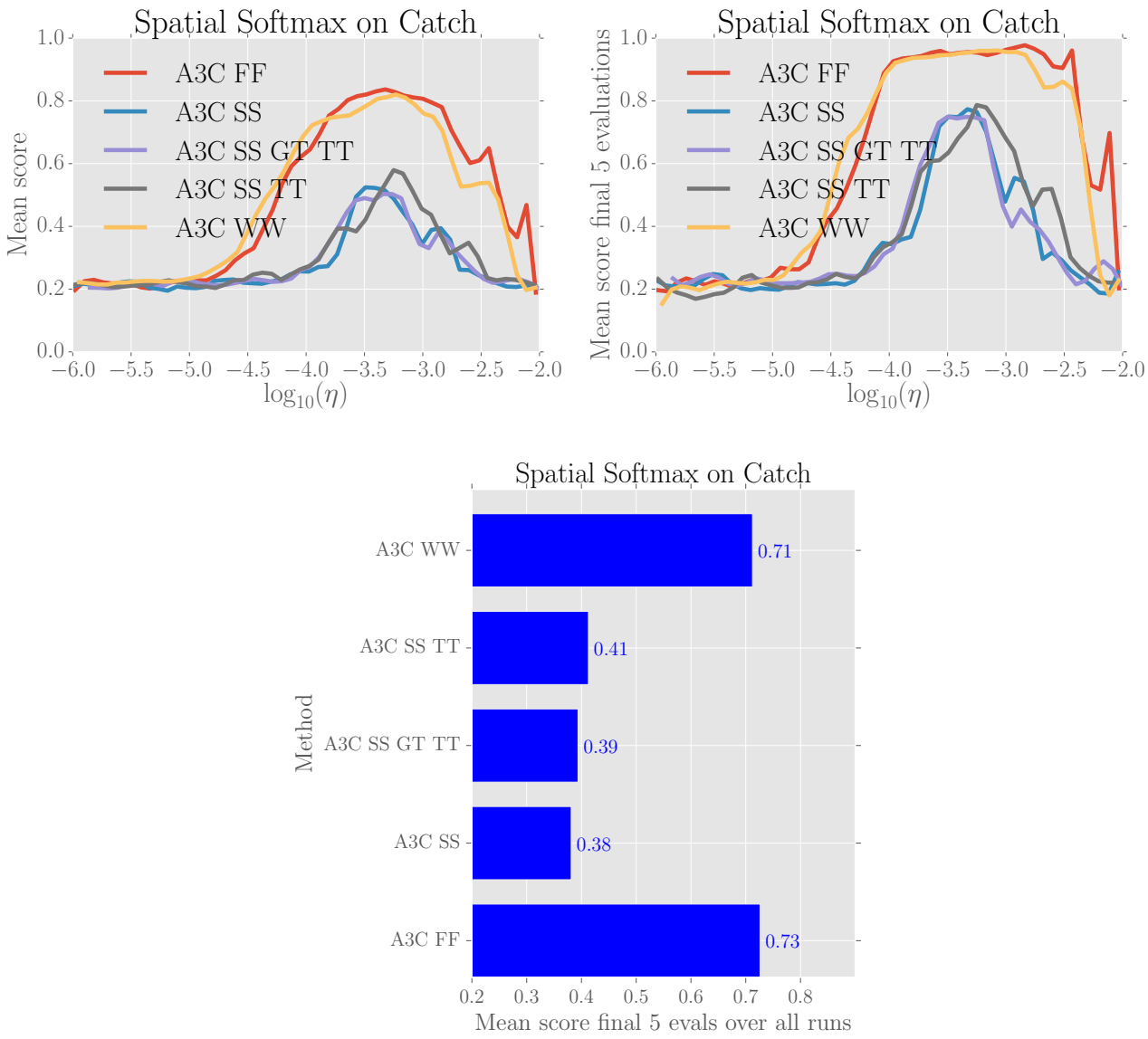


Figure 6.5: Comparison of A3C SS and A3C WW models vs. default A3C FF model. The plot shows a smoothed curve for the mean score on 20 evaluations (upper left) and the mean score on the final 5 evaluations (upper right) taken during the training of 1 million frames for different learning rates. The bar plot below reports the mean score of the final 5 evaluations averaged over all learning rates between 10^{-5} and 10^{-2}. It can be seen that the performance of the SISWS is comparable. for the higher learning rates, the SISWS models tend to perform better than the default A3C FF architecture.

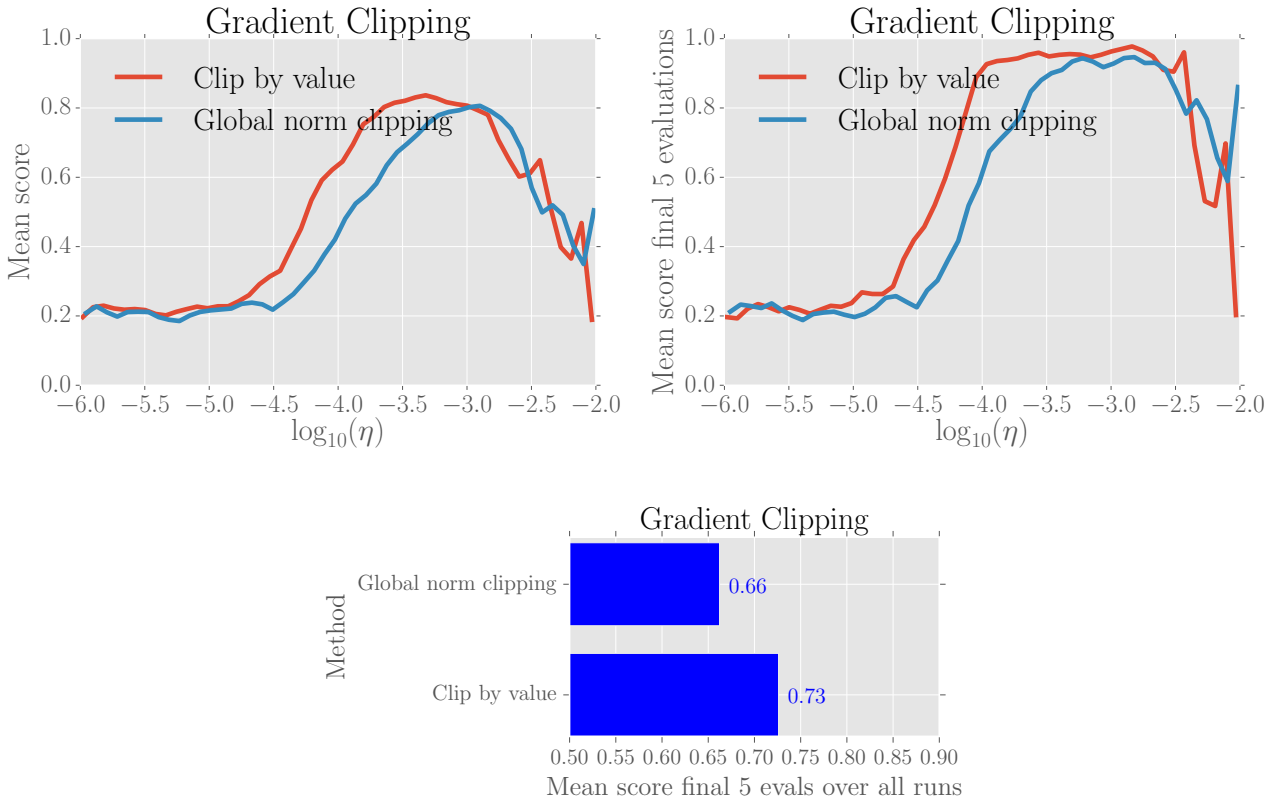


Figure 6.6: Comparison of different gradient clipping strategies. Section 6.3.1 provides details on the two strategies. The plot shows a smoothed curve for the mean score on 20 evaluations (upper left) and the mean score on the final 5 evaluations (upper right) taken during the training of 1 million frames for different learning rates. The bar plot below reports the mean score of the final 5 evaluations averaged over all learning rates between 10^{-5} and 10^{-2}. It can be seen that clipping by value outperforms global norm clipping. The suboptimal performance of global norm clipping might indicate a poorly set clipping parameter c_g.

Results The outcomes of our sweeps for the ELU units vs. ReLU units are given in Figure 6.7. We can see that despite of the fact that these activation functions are guaranteed to have non-zero gradients, the models with ELU activations perform worse than the default ReLU activations. This contrasts the findings in (Clevert et al., 2015) where these activation functions were compared with ReLU activations in the context of computer vision tasks with DNNs. Perhaps our results here underline the sensitivity of the algorithm to small variations of the hyperparameters. On the other hand, the decreased performance is mainly observable when the learning rates are greater than 10^{-3}. This might be explained by the fact that the magnitude of the gradients in ELUs are greater in general. Moreover, the default A3C settings clip the gradients element-wise by value. In certain cases, the direction of the gradients might be severely affected by this operation, resulting in parameter updates that are far from the desired direction.

6.3.3 Weight initializations

In order to reproduce results close to the default A3C as reported in (Mnih et al., 2016), many things have to be taken care of. For many domains in DL, weight initialization is only marginally important for actual convergence and there exist different strategies for initializ-

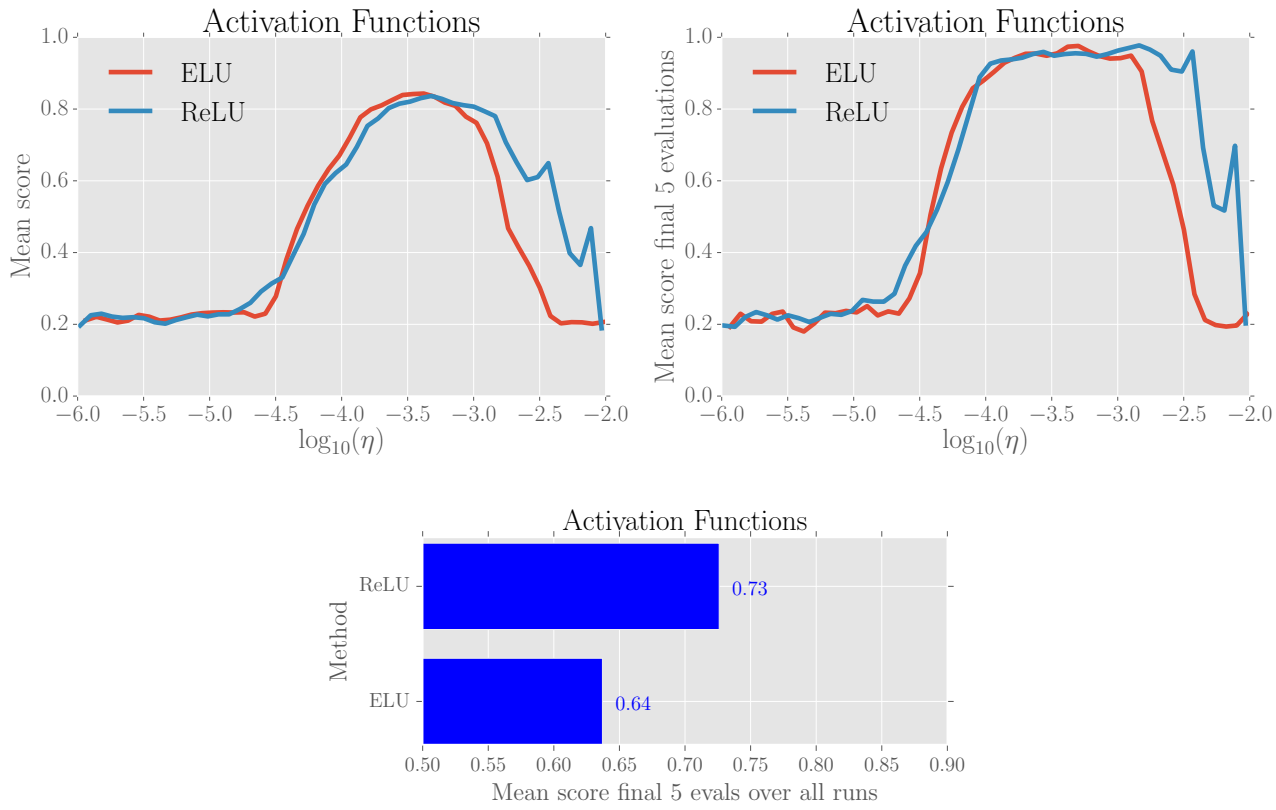


Figure 6.7: Comparison of ELU vs. default ReLU for the A3C FF model. The plot shows a smoothed curve for the mean score on 20 evaluations (upper left) and the mean score on the final 5 evaluations (upper right) taken during the training of 1 million frames for different learning rates. The bar plot below reports the mean score of the final 5 evaluations averaged over all learning rates between 10^{-5} and 10^{-2}. It can be seen that the ELU activation tends to be less likely to reach a proper policy for higher learning rates. This might be caused by the fact that the expected gradient is greater, causing unstable learning.

ing the weights in a neural network. This is reflected by the fact that different DL libraries tend to use different initialization schemes. The default weight initialization strategy used by Mnih et al. (2016) is that of the Torch library (Collobert et al., 2011). By default, for Torch the weights are initialized from a uniform distribution between $[-d, d]$ where $d = 1/\sqrt{fan_{in}}$ and where fan_{in} is the amount of connections per neuron with the previous layer. This initialization originates from the observations by Glorot and Bengio (2010). An alternative initialization is to draw the weights from a truncated normal distribution with mean 0 and variance $\sqrt{2/fan_{in}}$ as put forward by He et al. (2015b). TFLearn (Tang, 2016) tends to initialize the weights according to Sussillo and Abbott (2014) that suggest to initialize the weights by drawing from a uniform distribution between $[-\sqrt{3}d, \sqrt{3}d]$ where d equals that of (Glorot and Bengio, 2010).

Results We will now assess the performance using each of these initialization strategies. Figure 6.8 displays the comparison for the weight initialization strategies. We can see that the agent is quite sensitive to different strategies. Moreover, we see that the default Torch initialization yields the best results. The difference is quite substantial for the mean score over all evaluations as can be seen on the top left. This indicates that the algorithm is still sensitive to design decisions that are often not thought to be crucial, given that each of them has theoretical justifications in the literature.

6.4 Experiments on arcade games

We now consider 3 different architectures on two games of the ALE. We have our (i) standard A3C FF algorithm, (ii) a A3C LWS architecture with trainable centroids and (iii) an A3C WW architecture. The FF and WW architectures have two convolution layers with the same kernel size (8×8 and 4×4) and strides (4×4 and 2×2). In case of the LWS architecture, the first two layers are local weight sharing layers with 3 trainable group-wise centroids per layer. The WW model applies a ReLU nonlinearity to 32 out of 64 feature maps of the second convolutional layer and a spatial softmax operation on the remaining 32 kernels. The remainder of the architecture is the same as the default as introduced in Section 5.2.

Results The results are shown in Figure 6.9. For each line, five independent runs were performed. For each run, the agent was evaluated by taking the average score over 50 episodes every 1 million steps. The figure displays the average over 5 runs. For Beam Rider, we see that our A3C WW model outperforms the A3C FF and A3C LWS models substantially. It seems that the added spatial features can greatly aid the agent to find a proper policy sooner. The LWS and FF models tend to perform comparably. For Breakout, we can see that the LWS and WW architectures tend to perform comparable and worse than FF. Initially, the score progress is slower. This could be explained by the fact that the spatial consistency of the frames does not meet the requirements for LWS layers to be beneficial.

The Breakout game visually does not lend itself well to spatial softmax layers since a lot of spatial structure is reappearing at many locations in the screen. For example, the 'wall' that has to be broken cannot be well assigned to a single Cartesian coordinate by means of finding the softmax of a convolution output. In that case, the features that are generated by the spatial softmax might even be impeding the potential of obtaining good policies. The Beam Rider game is graphically more detailed than the Breakout game, which allows the convolution kernels to more specifically target certain salient features.

6.4.1 Value loss factor

In Algorithm 1 we added a value loss coefficient to line 18 with symbol λ_V. By default, our results are of runs where $\lambda_V = 0.5$. However, the original paper by Mnih et al. (2016) does not mention such a value loss coefficient, implying that $\lambda_V = 1$. Nevertheless, others that

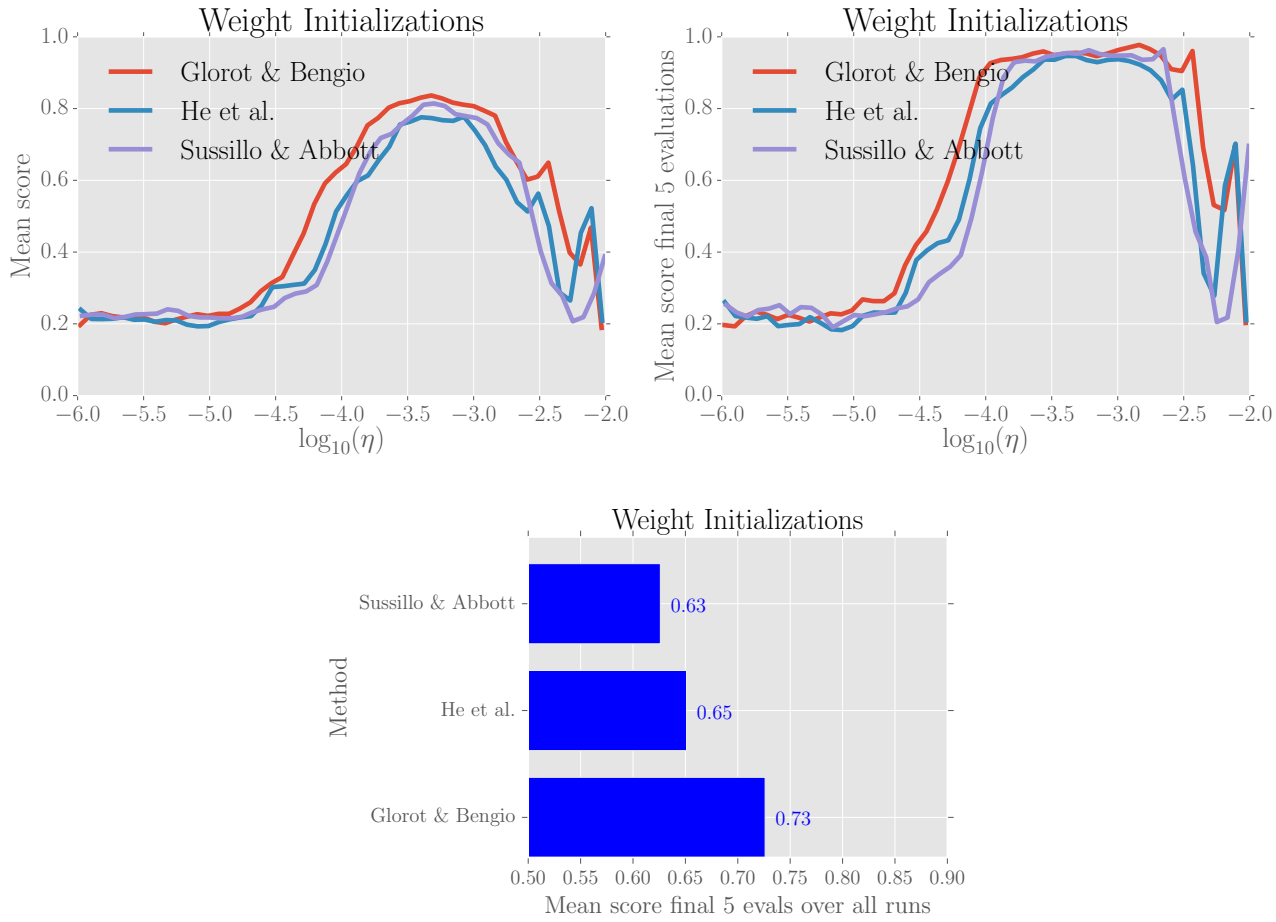


Figure 6.8: Comparison between weight initializations with default A3C FF. The plot shows a smoothed curve for the mean score on 20 evaluations (upper left) and the mean score on the final 5 evaluations (upper right) taken during the training of 1 million frames for different learning rates. The bar plot below reports the mean score of the final 5 evaluations averaged over all learning rates between 10^{-5} and 10^{-2}. Clearly, the initialization strategy can affect performance. Moreover, deviating from the default strategy as proposed by Glorot and Bengio (2010) yields impeded performance. More details are provided in Section 6.3.3.

have attempted to reproduce their results have contacted the authors who confirmed that they actually multiplied the value loss by 0.5 [1].

We noticed the importance of this parameter initially when we were considering Pong. Being unaware of this multiplication, we observed surprisingly poor performance. After realizing the difference in implementation, we reran the experiments. The results are given in Figure 6.10. For each line, five independent runs were performed. For each run, the agent was evaluated by taking the average score over 50 episodes every 1 million steps. The figure displays the average over 5 runs. The performance is clearly affected by the loss coefficient. When $\lambda_V = 1.0$ there is no learning at all. When $\lambda_V = 0.5$, learning is considerably better, although the shaded areas indicate that the standard errors are substantially large. This suggests that one or two runs resulted in poor performance. Obviously, altering the value loss coefficient does not only change the relative size of the policy loss vs. the value loss, but it also lowers the expected magnitude of the gradients. It is unlikely that the observed failure for $\lambda_V = 1$ can be devoted solely to the ratio between the policy and value loss. It is

[1] https://github.com/muupan/async-rl/wiki

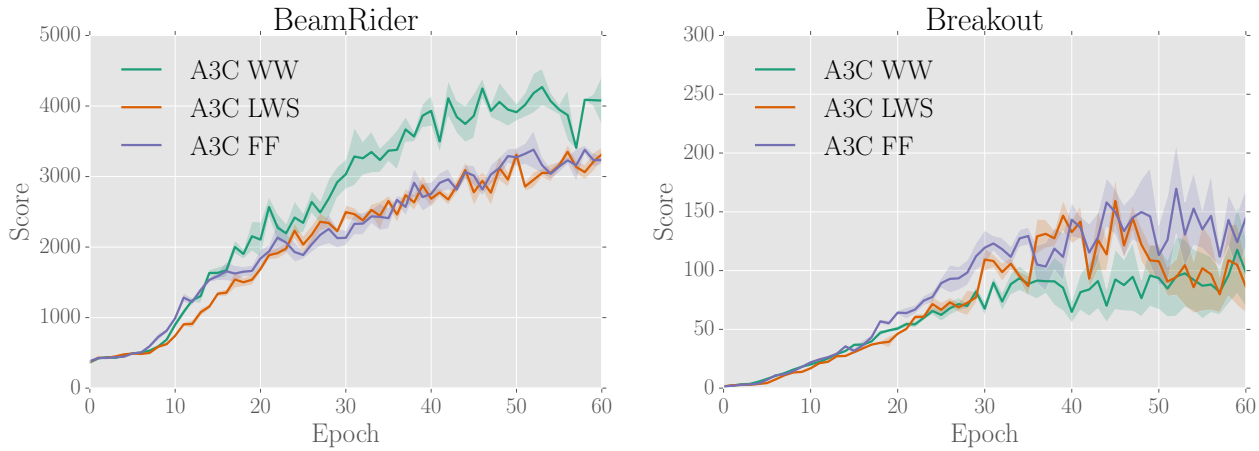


Figure 6.9: Performance of several architectures on two Atari games. Each epoch corresponds to 1 million steps. For each run, the agent is evaluated by taking the average score of 50 episodes after every 1 million steps. The plot displays the average of 5 separate runs. For Beam Rider (a), the WW architecture performs best, while the FF architecture and the LWS architecture perform comparably. For Breakout (b), the FF architecture performs best and the other two architectures perform comparably.

more likely that the greater magnitude of gradients caused the failure to learn.

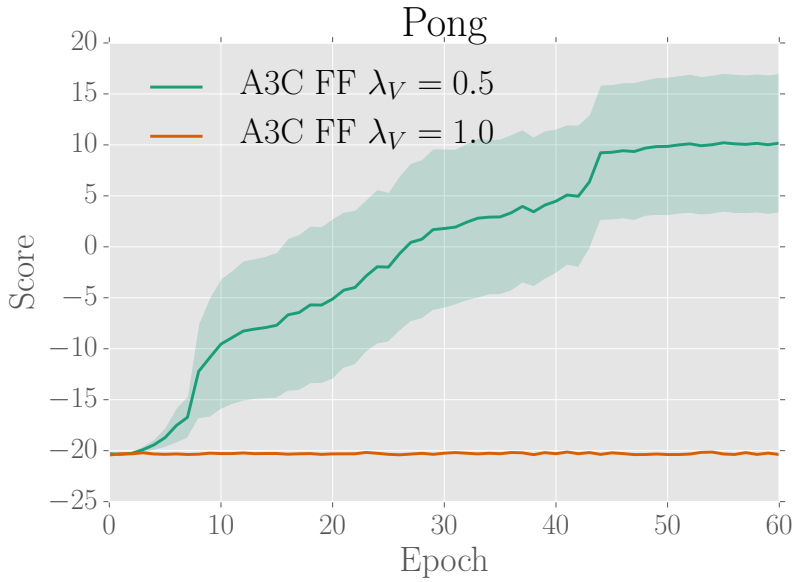


Figure 6.10: Comparison of different settings for λ_V which clearly affect performance. Each epoch corresponds to 1 million steps. For each run, the agent is evaluated by taking the average score of 50 episodes after every 1 million steps. The plot displays the average of 5 separate runs. The shaded areas indicate standard errors. When $\lambda_V = 1.0$, the default A3C FF model does not learn at all. When $\lambda_V = 0.5$, the model performs much better on average. The size of the shaded area seems to suggest that the agent failed to learn for one or two runs.

Chapter 7

Prototype Based Deep Reinforcement Learning

Prototype based supervised learning algorithms can be used for classification tasks. In these algorithms, there are usually several prototypes that populate an input space $\mathcal{X}$ or a feature space $\mathcal{H}$. There can be several prototypes per class and the algorithm typically classifies by determining the similarity (or distance) to nearest prototypes of a new input $\boldsymbol{x}$ or feature $\boldsymbol{h}$. Note that this procedure is quite different from an ordinary softmax output, which is a multinomial logistic regression operator, possibly on top of a deep neural network. In this chapter, we will explore whether prototype based learning can be used for deep reinforcement learning by proposing a novel actor-critic algorithm. We accomplish this by altering the A3C algorithm to use a prototype based policy prediction. The introduction of prototypes solely requires us to alter the construction of the policy output $\pi(s, a; \boldsymbol{\theta})$. This chapter covers the second main research question: *Is prototype based learning suited for deep reinforcement learning?*

7.1 Learning policy quantization

This section introduces a new kind of actor-critic algorithm: learning policy quantization (LPQ). It draws inspiration from learning vector quantization (LVQ) (Kohonen, 1990, 1995; Kohonen et al., 1996). LVQ is a supervised classification method in which some feature space $\mathcal{H}$ is populated by a set of prototypes $\mathcal{W} = \{\boldsymbol{w}_i\}_{i=1}^{W}$. Each prototype belongs to a particular class so that $c(\boldsymbol{w})$ gives the class belonging to $\boldsymbol{w}$. The LVQ1 algorithm classifies a new vector $\boldsymbol{x}$ by saying that $\hat{y} = c(\arg\min_{\boldsymbol{w}} d(\boldsymbol{x}, \boldsymbol{w}))$, where $d : \mathbb{R}^n \times \mathbb{R}^n \mapsto \mathbb{R}$ is a distance function. Given a supervised class label $c(\boldsymbol{x})$, the algorithm learns by moving the closest prototype towards $\boldsymbol{x}$ if $c(\boldsymbol{x}) = \hat{y}$. If $c(\boldsymbol{x}) \neq \hat{y}$, then the prototype is moved in the opposite direction.

Throughout the rest of this chapter, we come across some variations of LVQ. We acknowledge that many other variations to LVQ exist that are out of the scope of this thesis. The interested reader can consult (Kohonen, 1990, 1995; Kohonen et al., 1996).

7.1.1 Useful ideas from LVQ

There exist several ways of designing an LVQ classifier. We will now discuss a few options that are relevant to the LPQ algorithm.

Multiple prototypes per class An LVQ's division of the feature space can be made more complex by defining multiple prototypes per class. In the extreme case the total number of prototypes equals the number of data points. Storing separate prototypes this way then yields a nearest neighbor classifier.

Soft competition To find the winning prototype $\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} d(\boldsymbol{x}, \boldsymbol{w})$, one can either choose to select the closest prototype only, or to have a majority vote among the $m < M$ closest prototypes. Ties can be broken by also selecting the labels with the least average distance. Alternatively, one can use a distance weighted sum for each class such that closer prototypes have a larger contribution to the competition's outcome.

Distance function The distance function can greatly affect the behavior of the classifier. The Euclidean distance or squared Euclidean distance are common choices. More complicated distance functions can also be used. Matrix LVQ uses a parameterized matrix to learn feature relevances (Schneider et al., 2009). Its distance function is defined as:

$$d(\boldsymbol{x}, \boldsymbol{w}) = (\boldsymbol{x} - \boldsymbol{w})^T \Lambda (\boldsymbol{x} - \boldsymbol{w}), \tag{7.1}$$

where $\Lambda \in \mathbb{R}^{n \times n}$ is a matrix that is initially set to an identity matrix I_n and which will be iteratively updated through gradient descent such that it scales certain features and learns to exploit correlations between different features.

Soft class assignments Rather than having an all-or-nothing classification, class confidences can also be modeled by framing the set of prototypes as defining a density function over the input space $\mathcal{X}$. This is the idea behind robust soft learning vector quantization (RSLVQ) (Seo and Obermayer, 2003).

Generalized Learning Vector Quantization In (Sato and Yamada, 1996), the authors define the following objective function to be minimized:

$$\textstyle\sum_i \Phi(\mu_i) \quad \text{where} \quad \mu_i = \dfrac{d_+(\boldsymbol{x}_i) - d_-(\boldsymbol{x}_i)}{d_+(\boldsymbol{x}_i) + d_-(\boldsymbol{x}_i)}, \tag{7.2}$$

where $\Phi : \mathbb{R} \mapsto \mathbb{R}$ is any monotonically increasing function, $d_+(\boldsymbol{x}) = \min_{\boldsymbol{w}_i : c(\boldsymbol{w}_i) = c(\boldsymbol{x})} d(\boldsymbol{x}, \boldsymbol{w})$ is the distance to the closest *correct* prototype and $d_-(\boldsymbol{x}) = \min_{\boldsymbol{w}_i : c(\boldsymbol{w}_i) \neq c(\boldsymbol{x})} d(\boldsymbol{x}, \boldsymbol{w}_i)$ is the distance to the closest prototype with a *wrong* label.

Deep Learning Vector Quantization Recently, De Vries et al. (2016) proposed a deep LVQ algorithm in which the distance function is defined as follows:

$$d(\boldsymbol{x}, \boldsymbol{w}) = \|f(\boldsymbol{x}; \boldsymbol{\theta}) - \boldsymbol{w}\|_2^2, \tag{7.3}$$

where f is a DNN with parameter vector $\boldsymbol{\theta}$. Note that if we were to choose $f(\boldsymbol{x}; \boldsymbol{\theta}) = A\boldsymbol{x}$, we arrive at the GMLVQ algorithm as previously mentioned. Their deep LVQ algorithm serves as an alternative to the softmax function. The softmax function has a tendency to severely extrapolate such that certain regions in the parameter space attain high confidences for certain classes while there is no actual data that would support this level of confidence. Moreover, they propose to use the objective function as found in generalized LVQ. They show that a DNN with a GLVQ cost function outperforms a DNN with a softmax cost function. In particular, they show that their approach is significantly less sensitive to adversarial examples. An important design decision is to no longer define prototypes in the input space, but in the feature space. This saves forward computations for prototypes and can simplify the learning process as the feature space is generally a lower-dimensional representation compared to the input itself.

7.1.2 The learning policy quantization algorithm

In this section we generalize the actor of the actor-critic algorithm such that we obtain a novel RL method that bears similarities with LVQ in many aspects.

To a certain extent, the parameterized policy $\pi(s, a \mid \boldsymbol{\theta})$ of the DNN in A3C draws some parallels with a DNN classifier. In both cases, there is a certain notion of confidence toward a particular class label (or action) and both are often parameterized by a softmax function. Moreover, both can be optimized through gradient descent. We will now generalize the actor's output to be compatible with an LVQ classification scheme. Let us first rephrase the softmax function as found in the standard A3C architecture:

$$\pi(s, a; \boldsymbol{\theta}) = \frac{\exp\left(\boldsymbol{w}_a^T \boldsymbol{h}_{L-1}(s; \boldsymbol{\theta}_{L-1}) + b_a\right)}{\sum_{a' \in \mathcal{A}} \exp\left(\boldsymbol{w}_{a'}^T \boldsymbol{h}_{L-1}(s; \boldsymbol{\theta}_{L-1}) + b_{a'}\right)}, \tag{7.4}$$

where $\boldsymbol{h}_{L-1}$ is the hidden activation of the last hidden layer which is computed from state s and the vector $\boldsymbol{\theta}'$ that contains all parameters up to layer L. The vectors $\boldsymbol{w}_a$ are the column vectors of the weight matrix in the softmax layer and b_a denotes the bias corresponding to action a.

We generalize Equation (7.4) for LPQ as follows:

$$\pi(s, a; \boldsymbol{\theta}) = \frac{\exp\left(-d\big(\boldsymbol{w}_a, \boldsymbol{h}_{L-1}(s; \boldsymbol{\theta}_{L-1})\big)\right)}{\sum_{a' \in \mathcal{A}} \exp\left(-d\big(\boldsymbol{w}_{a'}, \boldsymbol{h}_{L-1}(s; \boldsymbol{\theta}_{L-1})\big)\right)}, \tag{7.5}$$

where d is a distance function that maps the hidden activation and a prototype $\boldsymbol{w}_a$ to a scalar distance. In our experiments we explore the performance of several distance functions. Note that we still use a softmax operation on the negative distances to compute a proper probability distribution.

Generalized Learning Policy Quantization

Equation (7.5) can be seen as the LPQ counterpart of the LVQ2.1 algorithm. The LPQ variant that is related most to the GLVQ extension as given in Equation (7.2) can be obtained by defining.

$$\pi(s, a; \boldsymbol{\theta}) = \Phi(\mu_a, s; \boldsymbol{\theta}) = \frac{\exp\left(\tau \mu_a\right)}{\sum_{a' \in \mathcal{A}} \exp\left(\tau \mu_{a'}\right)}, \tag{7.6}$$

where

$$\mu_a = \frac{d_-(\boldsymbol{h}, \boldsymbol{w}_{a'}) - d(\boldsymbol{h}, \boldsymbol{w}_a)}{d_-(\boldsymbol{h}, \boldsymbol{w}_{a'}) + d(\boldsymbol{h}, \boldsymbol{w}_a)}, \tag{7.7}$$

where $d_-(\boldsymbol{h}, \boldsymbol{w}_{a'}) = \min_{a': c(\boldsymbol{w}_{a'}) \neq c(\boldsymbol{w}_a)} d(\boldsymbol{h}, \boldsymbol{w}_{a'})$ is the distance to the closest prototype belonging to another action class. The τ parameter in Equation (7.6) is referred to as the temperature, which controls the steepness of the GLPQ operator.

There are several reasons for the precise formulation of Equation (7.6) and (7.7). Note that Equation (7.7) is different from the relative similarity measure as put forward by Sato and Yamada (1996). Suppose a is the correct action, then this leads to a high value of μ whenever $d(\boldsymbol{w}_a, \boldsymbol{h})$ is small. In other words, the actor will prefer the closer prototypes.

Attracting or repelling without supervision

Note that there is another subtle yet important difference in the definition of μ_a in Equation (7.7) compared to the definition in Equation (7.2). We can no longer directly determine whether a certain prototype $\boldsymbol{w}_a$ is *correct*. Therefore, Equation (7.7) does not use $d_+(\boldsymbol{h}, \boldsymbol{w}_a)$ but simply $d(\boldsymbol{h}, \boldsymbol{w}_a)$. In other words, we do not have supervised class labels anymore. The

obvious question that comes to mind is: how should we determine when to move a prototype toward some hidden activation vector $\boldsymbol{h}$? The answer is provided by the environment interaction and the critic. To see this, consider the standard A3C algorithm in which we find the following update rule (see also Algorithm (1) and Section 3.3.2 on the policy gradient theorem):

$$d\boldsymbol{\theta} \leftarrow d\boldsymbol{\theta} + \nabla_{\boldsymbol{\theta}'} \log \pi(a_i \mid s_i; \boldsymbol{\theta}')(G_t - V(s_i; \boldsymbol{\theta}'_V)) + \beta \nabla_{\boldsymbol{\theta}'} H(\pi(s_t; \boldsymbol{\theta}')), \tag{7.8}$$

where $G_t = \sum_{k=0}^{n-1} \gamma^k R_{t+k} + \gamma^n V(s_{t+n}; \boldsymbol{\theta}_V)$ is the n-step return obtained through environment interaction and $V(s_t; \boldsymbol{\theta}'_V))$ is the output of the critic. It is important to realize that all prototypes $\boldsymbol{w}_a$ are contained in both $\boldsymbol{\theta}$ and $\boldsymbol{\theta}'$. The factor $(G_t - V(s_i; \boldsymbol{\theta}'_V))$ is also known as the *advantage*. The advantage of a certain action a gives us the relative gain in expected return after taking action a compared to the expected return of state s_t. In other words, a positive advantage corresponds to a *correct* prototype, whereas a negative advantage would correspond to a *wrong* prototype. Intuitively, applying Equation (7.8) now corresponds to increasing the result of μ_i (that is by repelling $\boldsymbol{w}_a$ and attracting $\boldsymbol{w}_{a'}$) whenever the advantage is negative and decreasing the result of μ_i (that is by attracting $\boldsymbol{w}_a$ and repelling $\boldsymbol{w}_{a'}$) whenever the advantage is positive. Hence, the learning process closely resembles that of LVQ while we only modify the construction of $\pi(s, a; \boldsymbol{\theta})$.

Note that we can obtain the normal LPQ algorithm by replacing μ_a with $-d(\boldsymbol{w}_a, \boldsymbol{h})$. In that case, the prototype gradients that are obtained when determining $\nabla_{\boldsymbol{\theta}'} \log \pi(a_i | s_i; \boldsymbol{\theta}')$ are identical to those obtained for RSLVQ (Seo and Obermayer, 2003). For further details, see Appendix C.

7.2 Variations on LPQ

We now discuss some variations that could be added to the ideas above. Most of these variations were assessed on the GLPQ algorithm, since the GLPQ turned out to outperform the standard LPQ algorithm slightly (initially). Since we have explored many different variations, we accompany each of the elaborations with the corresponding parameter sweeps immediately, rather than listing all experiments afterwards at once. This will make the whole more readable and will spare otherwise necessary repetition and reference to earlier sections.

From the extensions that follow we see that the main analogies between LVQ and LPQ are that (i) in both cases several prototypes potentially correspond to the same class, (ii) multiple distance/similarity functions can be used and (iii) only a subset of the prototypes might be involved in determining the actual output of the system.

7.2.1 Distance functions

We have implemented and tested the following distance functions:

- Euclidean distance: $d(\boldsymbol{h}, \boldsymbol{w}) = \sqrt{(\boldsymbol{h} - \boldsymbol{w})^T (\boldsymbol{h} - \boldsymbol{w})}$,
- Squared Euclidean distance: $d(\boldsymbol{h}, \boldsymbol{w}) = (\boldsymbol{h} - \boldsymbol{w})^T (\boldsymbol{h} - \boldsymbol{w})$,
- Manhattan distance: $d(\boldsymbol{h}, \boldsymbol{w}) = \sum_i |(\boldsymbol{h} - \boldsymbol{w})|_i$.
- Cosine distance: $d(\boldsymbol{h}, \boldsymbol{w}) = 1 - \dfrac{\sum_i w_i h_i}{\sqrt{\sum_i w_i^2} \sqrt{\sum_i h_i^2}}$
- Pearson distance: $d(\boldsymbol{h}, \boldsymbol{w}) = 1 - \dfrac{\sum_i (w_i - \bar{w})(h_i - \bar{h})}{\sqrt{\sum_i (w_i - \bar{w})^2} \sqrt{\sum_j (h_i - \bar{h})^2}}$

- Inverse squared Euclidean[1]: $d(\boldsymbol{h}, \boldsymbol{w}) = -\dfrac{1}{1 + (\boldsymbol{h} - \boldsymbol{w})^T(\boldsymbol{h} - \boldsymbol{w})}$

Note that for the sake of readability, we have omitted the subscripts for both vectors and replaced $\boldsymbol{h}(s; \boldsymbol{\theta}')$ with $\boldsymbol{h}$.

Experiments on Catch We directly assess the distance functions on the Catch game. In the results of Figure 7.1, we have used the standard LPQ algorithm which could be seen as the reinforcement learning counterpart of the RSLVQ algorithm (Seo and Obermayer, 2003). In these experiments, there were 16 prototypes per action. The usage of multiple prototypes per class and how it affects the performance of the agent is discussed in Section 7.2.2. Clearly, the squared Euclidean distance yields the best results, followed by the Manhattan distance function. This is consistent with the analysis of (Sato and Yamada, 1996). The other distance functions lead to considerably worse results. Given the clear outcome of this analysis, the Euclidean distance function was chosen as the default distance function from hereon.

7.2.2 Multiple action prototypes

Note that Equation (7.5) assumes that we have only a single prototype per class. We can involve multiple prototypes per class in the competition by doing the following:

$$\pi(s, a; \boldsymbol{\theta}) = \frac{\sum_{i:c(\boldsymbol{w}_i)=a} \exp\Big(- d\big(\boldsymbol{w}_i, \boldsymbol{h}_{L-1}(s; \boldsymbol{\theta}_{L-1})\big)\Big)}{\sum_j \exp\Big(- d\big(\boldsymbol{w}_j, \boldsymbol{h}_{L-1}(s; \boldsymbol{\theta}_{L-1})\big)\Big)} \qquad \text{or,} \tag{7.9}$$

$$\pi(s, a; \boldsymbol{\theta}) = \frac{\sum_{i:c(\boldsymbol{w}_i)=a} \exp\big(\tau \mu_i\big)}{\sum_j \exp\big(\tau \mu_j\big)}, \tag{7.10}$$

where we the numerator sums over the prototypes belonging to action a and the denominator sums over all prototypes. Note that we can unify the proposed policy function and the standard softmax output by simply setting the number of prototypes per action to 1 and by letting $d(\boldsymbol{h}, \boldsymbol{w}) = -(\boldsymbol{w}^T \boldsymbol{h} + b)$.

Experiments on Catch Figure 7.2 displays the results for different amounts of prototypes per action. We assessed the robustness for different learning rates for 16, 32 or 64 prototypes per action. All prototypes were allowed to participate in the competition and the softmax outputs for each of the prototypes was summed as given in Equation (7.9) above. Especially when looking to the plot on the left, it can be seen that the score seems to be greater when the number of prototypes is either 16 or 32.

7.2.3 Sizing the prototype competition

Rather than involving all prototypes of the actor corresponding to any of the actions, we can also choose a subset of *neighboring* prototypes, which we denote $\mathcal{B}$. The subset $\mathcal{B}$ is found by taking the k closest prototypes for each class:

$$\mathcal{B} = \bigcup_{a \in \mathcal{A}} \arg\min_{B^{(k)} \subset W} \sum_{\boldsymbol{w} \in B^{(k)}} d(\boldsymbol{h}, \boldsymbol{w}), \tag{7.11}$$

which takes the union of action-specific subsets of size k that minimize the sum of distances. Prior to the experiments, it was hypothesized that this might affect the local specificity of

[1] This can be seen as a negative inverse squared Euclidean similarity, which is technically not a distance metric.

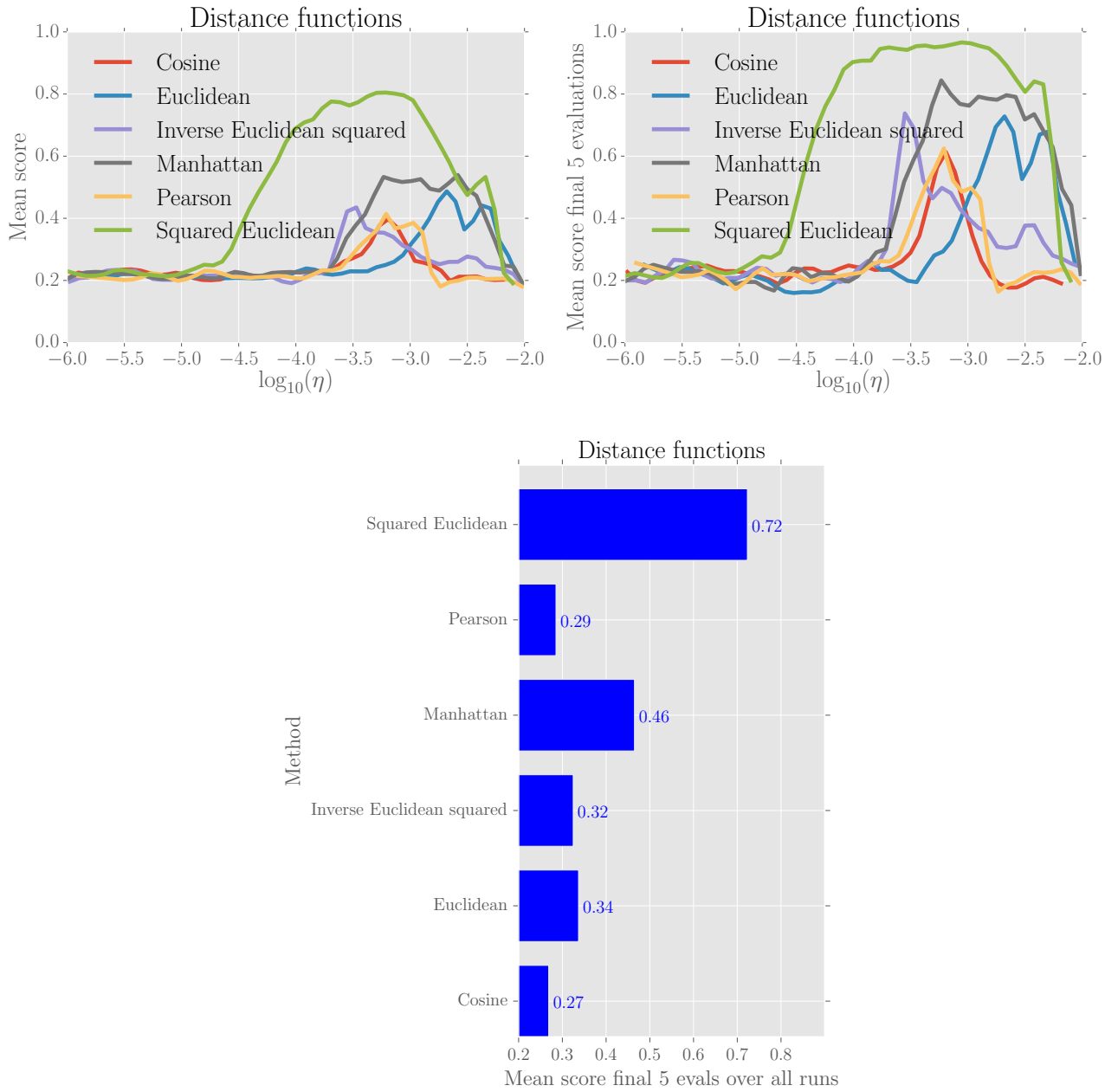


Figure 7.1: Comparison of A3C FF with LPQ models in which we vary the distance functions. The plot shows a smoothed curve for the mean score on 20 evaluations (upper left) and the mean score on the final 5 evaluations (upper right) taken during the training of 1 million frames for different learning rates. The bar plot below reports the mean score of the final 5 evaluations averaged over all learning rates between 10^{-5} and 10^{-2}. The default LPQ algorithm was used as described by Equation (7.9) where we used 16 prototypes per action and all prototypes participated in the competition.

the parameter updates, as now only a subset of the closest prototypes would be updated. Ideally, this would allow for higher learning rates for the prototypes in particular, as only a part of the separation of $\mathcal{H}$ would be affected for each update.

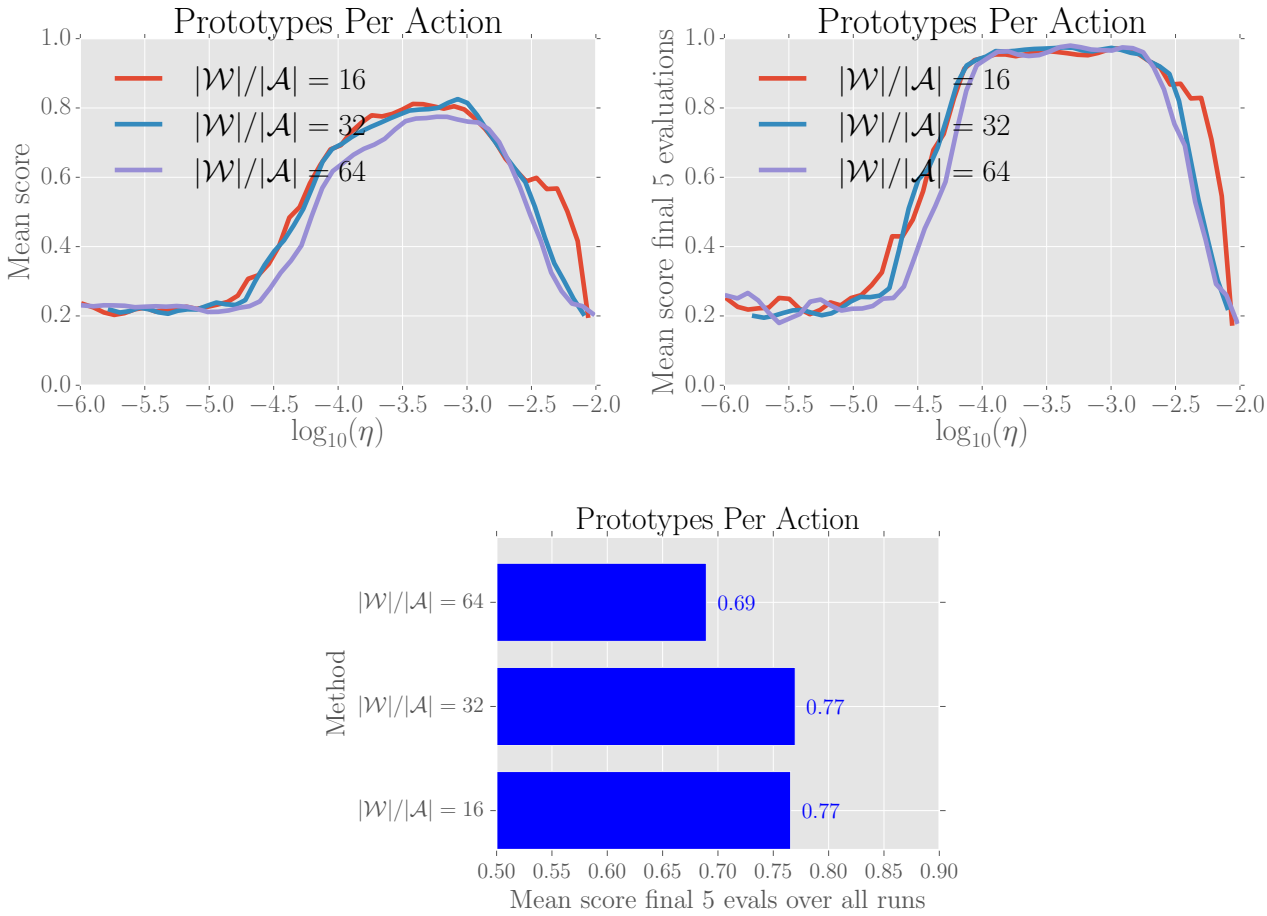


Figure 7.2: Comparison of A3C FF with GLPQ models in which we vary the amount of prototypes per action. The plot shows a smoothed curve for the mean score on 20 evaluations (upper left) and the mean score on the final 5 evaluations (upper right) taken during the training of 1 million frames for different learning rates. The bar plot below reports the mean score of the final 5 evaluations averaged over all learning rates between 10^{-5} and 10^{-2}. Clearly, using 64 prototypes yields deteriorated performance, while using 16 or 32 prototypes yields comparable performance.

Experiments on Catch Figure 7.3 illustrates the results on the varying competition size and how it affects the agent's performance on the Catch game. The agent used the GLPQ output as given in Equation 7.6 with 16 prototypes per action. It can be seen that the algorithm is roughly invariant to the different competition sizes although we have restricted our experiments to only a few different settings for reasons of limited time.

7.2.4 Softmax temperature

The GLPQ output has an interesting property as a consequence of the fact that $\mu_i \in [-1, 1]$.

Theorem 1. *Consider a GLPQ output operator as in Equation (7.10). Also assume that each action has k corresponding prototypes.*

We then know that

$$\max \frac{\sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i)}{\sum_j (\tau\mu_j)} = \frac{\exp(2\tau)}{\exp(2\tau) + |\mathcal{A}| - 1} \tag{7.12}$$

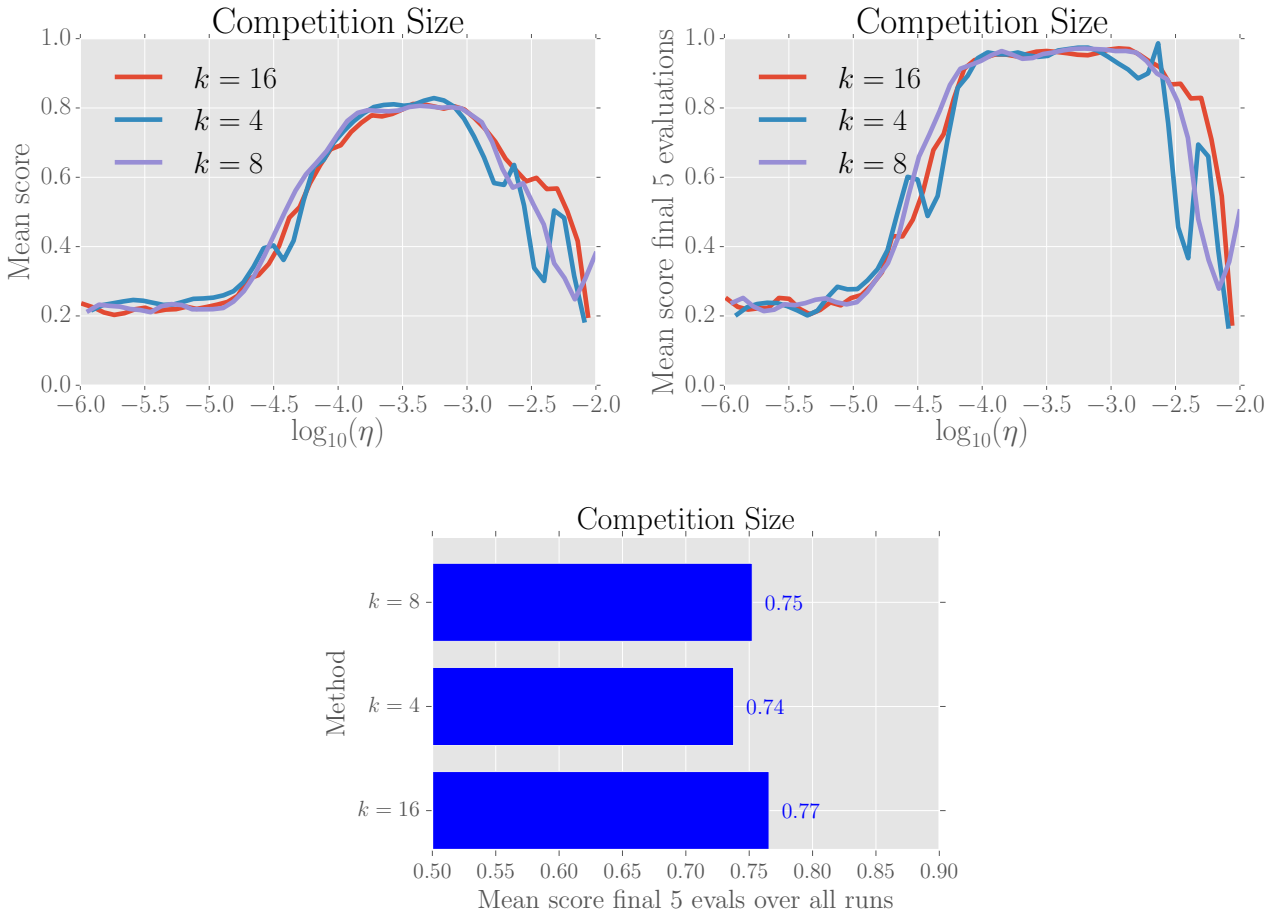


Figure 7.3: Comparison of A3C FF with GLPQ models in which we vary the competition size. In all of these cases there were 16 prototypes per class. The k parameter represents the competition size. For example, $k = 4$ implies that the 4 closest prototypes of each action were used to construct $\tilde{\mathcal{A}}^k$ as mentioned in Equation (7.11). The plot shows a smoothed curve for the mean score on 20 evaluations (upper left) and the mean score on the final 5 evaluations (upper right) taken during the training of 1 million frames for different learning rates. The bar plot below reports the mean score of the final 5 evaluations averaged over all learning rates between 10^{-5} and 10^{-2}. In general, greater competition sizes yield the best performance.

Proof. We automatically know that the output is maximized for some action a if for all prototypes for which $c(\boldsymbol{w}_i) = a$ we have that $\boldsymbol{w}_i = \boldsymbol{h}$. In such cases, we obtain $\mu_i = 1$ and $\mu_j = -1$ for $j \neq i$. Therefore, the resulting maximum value of the policy is

$$\frac{k \exp(\tau)}{k \exp(\tau) + k(|\mathcal{A}| - 1) \exp(-\tau)} = \frac{\exp(2\tau)}{\exp(2\tau) + |\mathcal{A}| - 1} \tag{7.13}$$

□

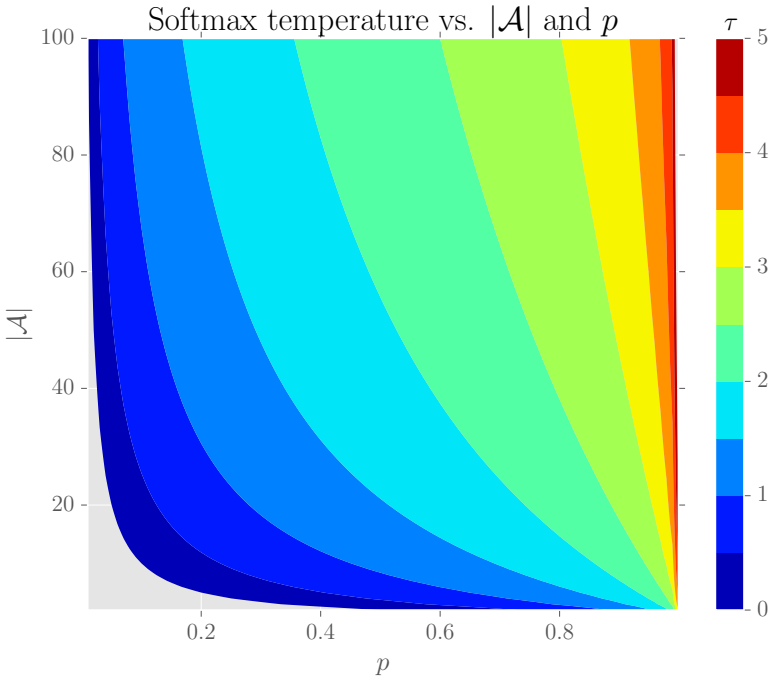


Figure 7.4: A pseudo color plot that displays the relation between the number of actions $|\mathcal{A}|$, some desired maximum policy value p and the corresponding softmax temperature τ.

Corollary 1. *Let p be the maximum value of the policy, we then have that*

$$p = \frac{\exp(2\tau)}{\exp(2\tau) + |\mathcal{A}| - 1} \tag{7.14}$$

$$\Rightarrow p\exp(2\tau) + p(|\mathcal{A}| - 1) = \exp(2\tau) \tag{7.15}$$

$$\Rightarrow \exp(2\tau) = -\frac{p(|\mathcal{A}| - 1)}{p - 1} \tag{7.16}$$

$$\Rightarrow \tau = \frac{1}{2}\ln\left(-\frac{p(|\mathcal{A}| - 1)}{p - 1}\right), \tag{7.17}$$

which we could use as a way of choosing a value for τ. The way that τ depends on $|\mathcal{A}|$ and p is displayed in Figure 7.4.

In this particular case we have assumed that for all $\boldsymbol{w} \in B^{(k)}$ for which $c(\boldsymbol{w}) = a$ we have that $\boldsymbol{w} = \boldsymbol{h}$. This is not a desirable situation, as multiple prototypes now have the same position, which is perhaps not optimal. Instead we can act *as if* only a single prototype can be responsible for a policy output of p. In that case, we have:

$$\tau = \frac{1}{2}\ln\left(-\frac{p(|B^{(k)}| - 1)}{p - 1}\right) = \frac{1}{2}\ln\left(-\frac{p(k|\mathcal{A}| - 1)}{p - 1}\right), \tag{7.18}$$

in which we can see that the role of k with respect to τ is identical to the role of $|\mathcal{A}|$. Hence, we can conclude that the shape of the surface of τ is the same as in Figure 7.4 in which we replace $|\mathcal{A}|$ with k and set $|\mathcal{A}|$ to be fixed. Perhaps a good strategy would be to initialize τ using Equation (7.17) and slowly increase it to the value given by Equation (7.18).

Introducing such a softmax parameter severely influences the magnitude of the gradients. Consequently, it also affects the agent's performance if we are not careful. In Appendix C it is shown that the gradient of the prototypes are proportional to the temperature. Note that by similar reasoning, the other gradients that flow through the network are also proportional to τ. We could compensate for this factor by multiplying the policy loss by $\frac{1}{\tau}$.

Results on Catch

We will now consider a range of different temperature configurations for GLPQ on the Catch game.

Temperature configurations First we consider whether adding the temperature as in Equation (7.17) yields improved performance. Figure 7.5 provides the results on the Catch game. We have used three different configurations here. First, we let $\tau = 1$ which is the 'vanilla' GLPQ algorithm. Then, we let τ depend on the time step by linearly increasing p in Equation (7.17) from 0.9 to 0.99. This corresponds to the line indicated by $\tau(t)$. The third configuration also linearly increases p in the same range and multiplies the policy loss by $\frac{1}{\tau}$. We can see that using the schedule for p yields worse results than using either $\tau = 1$ or combining the schedule with the policy loss correction. Second, we can see that in general, the line of the temperature schedule with loss correction ends up marginally higher than when $\tau = 1$. We have also experimented with a trainable temperature setting, which is indicated by 'τ trainable'. For the trainable τ we chose to multiply the loss with $\frac{1}{\tau}$ since we by then knew that this would yield slightly better results.

Temperature schedule range We can choose to use Equation (7.17) or (7.18) for determining our temperatures. The first equation yields a lower temperature than the second. We will now assess the performance using each of these configurations where we refer to the model that uses the first Equation (7.17) as the 'cold' model and Equation (7.18) as the 'hot' model. We linearly increase p from 0.9 to 0.99 over a million steps and we multiply the policy loss by $\frac{1}{\gamma}$ as this led to better results in the previous paragraph. The searing model refers to the case where (7.18) where p was increased from 0.95 to 0.999. The results are provided in Figure 7.6. In the plot we compare both hot and cold configurations with the default A3C with softmax output. We can see that especially the hot and searing configurations yield a higher performance when the learning rate is smaller than $10^{-3.5}$. This might reflect the fact that this schedule allows for a less stochastic policy. Moreover, single prototypes are now more decisive, such that the algorithm converges to a good policy while not having to move a whole range of prototypes towards a certain position in the feature space.

7.2.5 GLPQ vs. LPQ

It is also interesting to see whether GLPQ outperforms LPQ on the Catch game. Note that the relative similarity measure as given in Equation (7.7) ensures that the attractive forces are greater than the repulsive forces as shown in (Sato and Yamada, 1996). This ultimately guarantees convergence under the standard stochastic convergence conditions for GLVQ.

The comparison of GLPQ and LPQ is illustrated in Figure 7.7. We have chosen the best configuration of GLPQ that we obtained thus far, which corresponds to the 'hot' temperature schedule as determined in the previous paragraph. Both GLPQ and LPQ used 16 prototypes per action and all prototypes participate in the competition. It can be seen that GLPQ outperforms LPQ marginally, yet consistently for most of the learning rates that were sampled.

7.3 Experiments on arcade games

Just as we did in Section 6.4 we have selected a few models that we assess in two Atari games. We selected GLPQ with a temperature schedule following Equation (7.17), which corresponds to the 'cold' model. Due to time constraints, we were unable to explore whether the 'hot' model would yield better results, based on the experiments that we have seen before, this might have been the case. The LPQ model that is displayed here also has suboptimal parameter settings when compared to the outcomes of the parameter sweeps on Catch. The

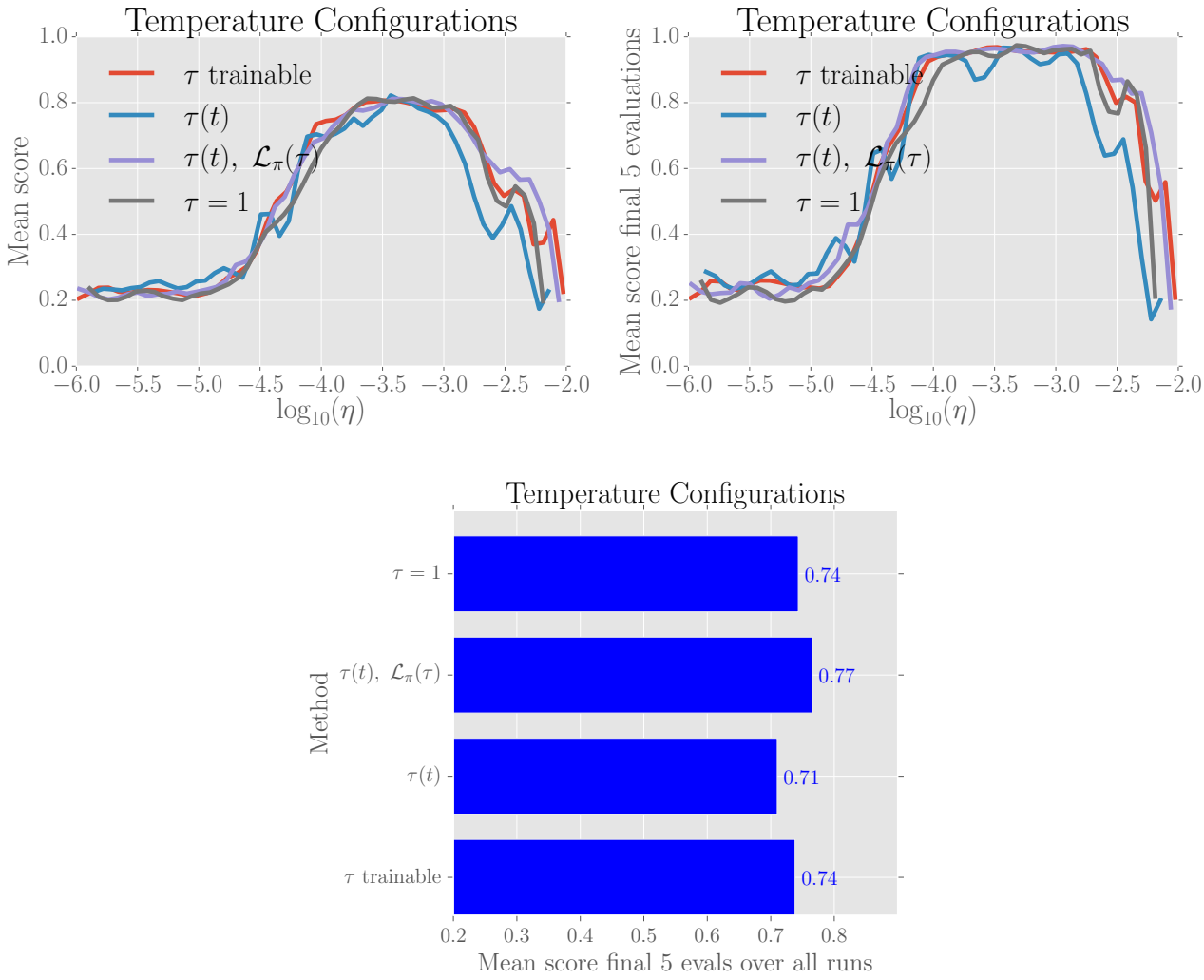


Figure 7.5: Comparison of temperature configurations where $\tau(t)$ refers to a schedule for τ where p from Equation (7.17) is increased linearly from 0.9 to 0.99. The same holds for other models that mention $\tau(t)$. The line labeled with $\tau(t)$ and $\mathcal{L}_\pi(\tau)$ refers to the model where the policy loss is multiplied by $1/\tau$. The plot shows a smoothed curve for the mean score on 20 evaluations (upper left) and the mean score on the final 5 evaluations (upper right) taken during the training of 1 million frames for different learning rates. The bar plot below reports the mean score of the final 5 evaluations averaged over all learning rates between 10^{-5} and 10^{-2}. For the trainable configuration of τ, the loss was also corrected similarly, since this clearly yielded better performance.

GLPQ model uses 16 prototypes per action, while the LPQ model uses 15. The GLPQ model includes all prototypes in the competition, while the LPQ only includes 10 for each action.

The results are shown in Figure 7.8. It can be seen that GLPQ and LPQ generally train slower than the default architecture for both games. For Breakout, the final performance for LPQ and A3C FF is similar. For BeamRider, GLPQ starts improving considerably slower than LPQ and FF, but surpasses LPQ eventually. This might indicate that the temperature should have been higher for GLPQ, since the performance gradually approaches that of A3C FF.

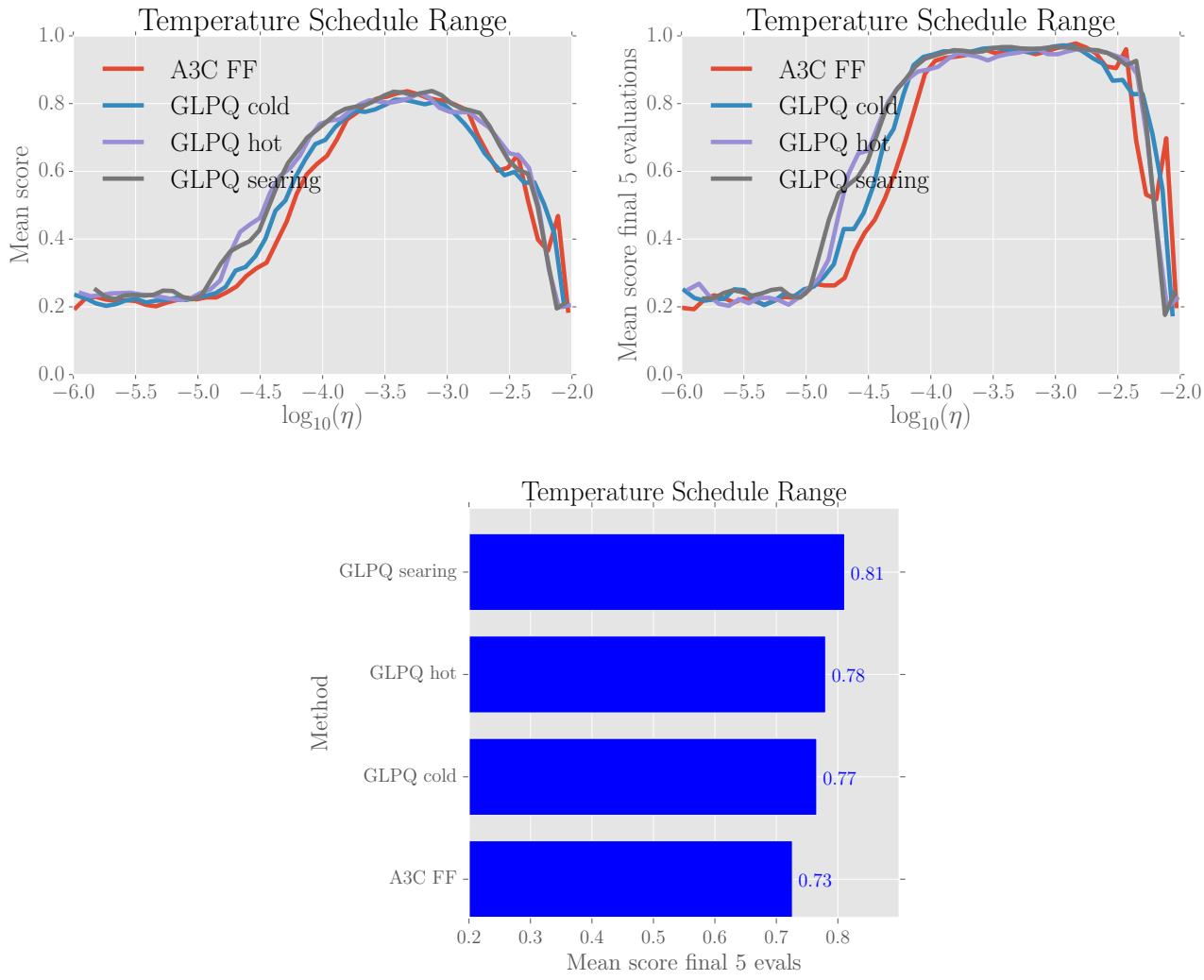


Figure 7.6: Comparison of different temperature schedules. The plot shows a smoothed curve for the mean score on 20 evaluations (upper left) and the mean score on the final 5 evaluations (upper right) taken during the training of 1 million frames for different learning rates. The bar plot below reports the mean score of the final 5 evaluations averaged over all learning rates between 10^{-5} and 10^{-2}. The greater the temperature, the better the performance on the last evaluations on Catch. This is particularly true for lower temperatures. The bar plot indicates the average score on the five last evaluations for all runs where $\alpha \in [10^{-5}, 10^{-2}]$.

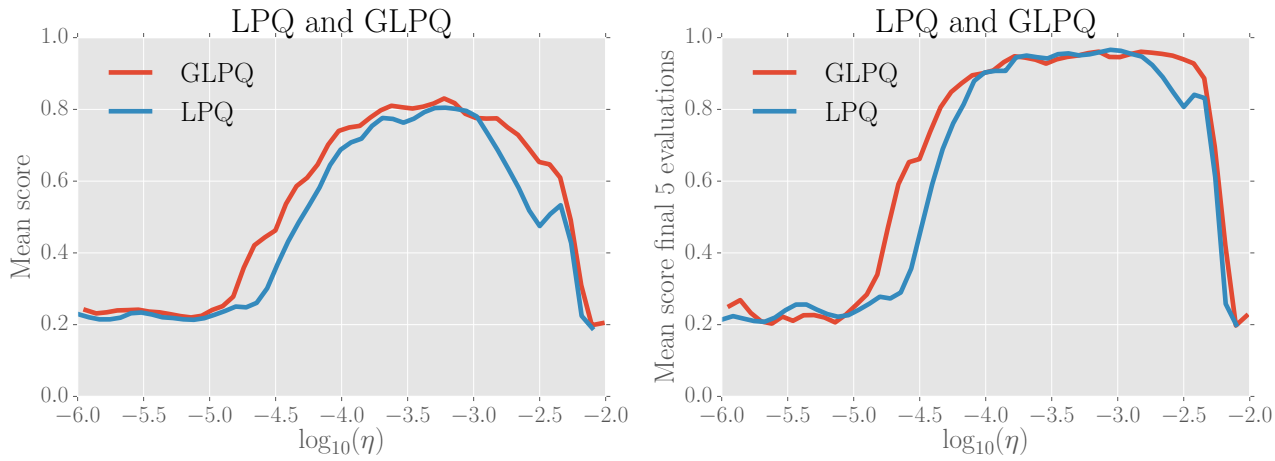


Figure 7.7: Comparison of GLPQ vs. LPQ on Catch. The plot shows a smoothed curve for the mean score on 20 evaluations (left) and the mean score on the final 5 evaluations (right) taken during the training of 1 million frames for different learning rates. It can be seen that GLPQ tends to perform marginally better for the extrema of relatively high or low learning rates. Both algorithms used the squared Euclidean distance with 16 prototypes per class and a competition size of 16 per class.

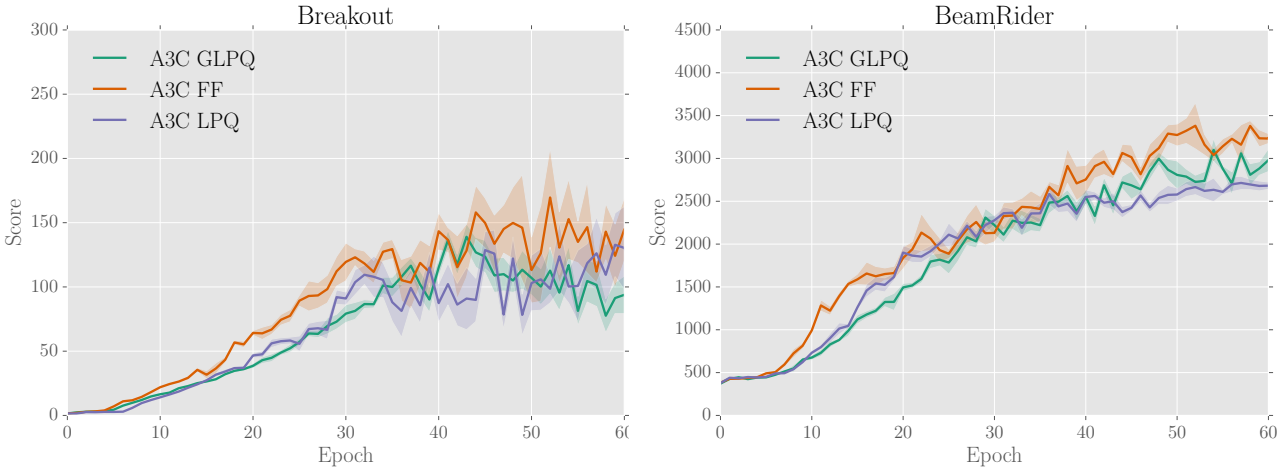


Figure 7.8: Results on two Atari games where we compare the performance of GLPQ and LPQ with A3C FF. Each epoch corresponds to 1 million steps. For each run, the agent is evaluated by taking the average score of 50 episodes after every 1 million steps. The plot displays the average of 5 separate runs. The shaded areas indicate standard errors. For Breakout, the final performance of the GLPQ and FF models are comparable, whereas the GLPQ approach performs worse. For Beam Rider, the performance of the FF approach surpasses that of GLPQ followed by LPQ.

Chapter 8

Concluding Remarks and Discussion

In this thesis we have explored several variations to a state-of-the-art deep reinforcement learning algorithm. Given the fact that the body of literature on DRL is still relatively small and the fact that architectural design decisions were not addressed in detail in other research thus far, we have provided a comparison of different variations to the default architectures as found in other literature. Fostering the comprehension of the influence of such decisions is a valuable contribution to the field as we have seen that these variations can greatly affect performance. The second major contribution is the introduction of a learning vector quantization algorithm designed for an actor-critic agent. In the introduction, we have stated two main research questions that were addressed in Chapters 6 and 7. We will now reflect on these questions and provide the relevant discussion and directions for future research.

1. To what extent do architectural design decisions and hyperparameters of an agent's deep neural network affect the resulting performance? In Chapter 6 we have assessed variations to the A3C approach in terms of neural network architectures and hyperparameter settings. We have confirmed that it appeared challenging to maintain performance when parameters were varied, such as weight initialization, activation functions or gradient clipping strategies. This seems to indicate that the default settings of the A3C as proposed by Mnih et al. (2016) are sensitive in the sense that deviating from them generally results in worse performance. In other words, the domain of optimal settings seems to be narrow. This undermines the guarantee that results within relatively simple domains such as the Catch game or Atari games can be generalized to other domains. Indeed, Jaderberg et al. (2016) acknowledge that other types of video games require different parameter settings to ensure proper performance, albeit an acceptable set of modifications given their impressive results. The outcomes here suggest that further research into the hyperparameter sensitivity and ensuring robustness could be a valuable contribution to the current literature in DRL.

Other than our exploration on hyperparameters we have looked at different neural layers. We have seen that local weight sharing can yield improved performance for certain learning rate settings when compared to convolutional layers on the Catch game. Moreover, Appendix B seems to support the fact that such layers can outperform default convolutional architectures while using less parameters for certain domains. Unfortunately, the improvements in our smaller scale experiments were not observable in the Atari domain, where we have seen that the usage of LWS layers yielded deteriorated performance. Perhaps further research on the similarity functions for kernel-centroid pairs and grid locations could improve the performance of this algorithm. Alternatively, the local weight sharing might be made *online* much like an attention mechanism. This could be achieved by creating a component of the network that predicts the appropriate centroid location given some input. This would make the whole architecture more adaptive to small variations between inputs.

We have shown that using a combination of spatial softmax operations and regular convolutions with ReLU nonlinearities can result in improved performance for the Beam Rider game. However, this improvement was not observable for the Breakout game due to the spatial structure of the game that cannot be represented well by spatial softmax layers. Many improvements to state-of-the-art approaches in the literature performed worse for a subset of games, while outperforming earlier approaches on most other games. Hence, it is difficult to decide what our limited research effort entails for more general claims about the performance of different architectures. Obviously, this decision would be easier to make if we were to do more experiments, but we were unable to complete more experiments because of limited time and limited computational resources.

2. Is prototype based learning suited for deep reinforcement learning? Chapter 7 described a new algorithm for reinforcement learning which is inspired by the learning vector quantization (LVQ) algorithm. The new learning policy quantization (LPQ) algorithm was combined with a deep neural network and tested on a simple Catch game and on two games in the Atari domain. We have elaborated on several variations of the LPQ algorithm from which we distilled an optimal set of parameters given the limited amount of experiments to support these parameters. The variations included the usage of different distance functions, LPQ vs. GLPQ, the competition size, the amount of prototypes per action and softmax temperature configurations. All of these variations were accompanied with experiments on the Catch game. We have seen that the temperature parameter and the choice of the distance function seem to have a considerable effect on performance.

The best results on the Catch game seem to suggest that the GLPQ algorithm performs slightly better than LPQ for lower learning rates and competitive for others when using a temperature scheme which is based on the decisiveness of a single prototype vector. Because of limited time and computational resources, we were unable to further improve our parameters and our understanding thereof. The best results obtained with GLPQ on Catch seem to outperform the softmax layer.

Both GLPQ and LPQ yielded competitive results for the Catch game when compared to a softmax layer, but they result in slightly worse performance on the two Atari games that were assessed. Given the limited amount of experiments, it is difficult to make general claims about the difference in performance of these algorithms for the full Atari domain, let alone outside the ALE.

The novel application of learning vector quantization to actor-critic algorithms opens up the door for a wide array of further research. For example, one could look into more advanced strategies for softmax temperature settings, the proper initialization of prototypes, dynamically adding or removing prototypes, alternative distance functions, adding learning vector regression for value function approximation and more. Potentially, progress made towards improving the LPQ algorithms will yield a better alternative to the default softmax output.

Appendices

Appendix A

Supplementary Material Neural Architecture Design

During the project, many other architectural variations were explored that presumably do not deserve as much attention as the ones mentioned in Chapter 6. This appendix section lists the ideas that were explored. Some of these ideas could use further exploration to show their full potential, and some of these ideas are clearly performing poorly.

A.1 Experiments

A.1.1 Gradient norm clipping

In order to find a suitable setting for the global norm clipping parameter c_g as explained in Section 6.3.1, we performed parameter search where we vary the learning rate and c_g both by sampling them from a log uniform distribution and using the Catch game as a testbed. The resulting qualitative plot can be seen in Figure A.1. It can be seen that only a small area within the search space gave good results. From the results here we chose $c_g = 25 \approx 10^{1.4}$ for further experimenting in Section 6.3.1.

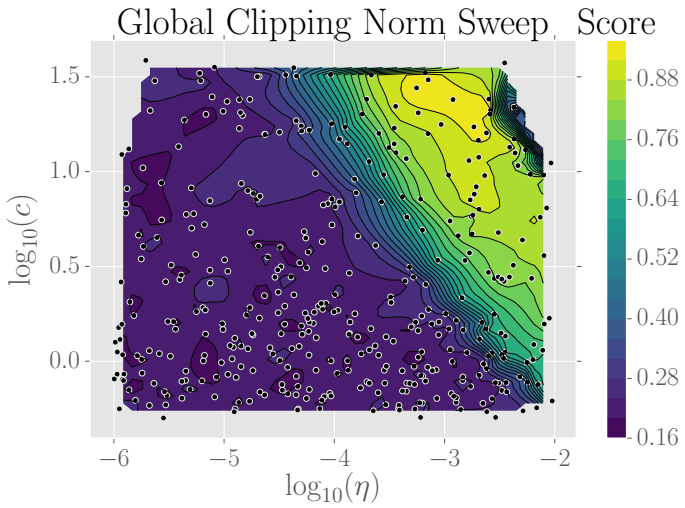


Figure A.1: Gradient norm clipping c and learning rate η. Both were sampled from a log uniform distribution in the range depicted by the axes.

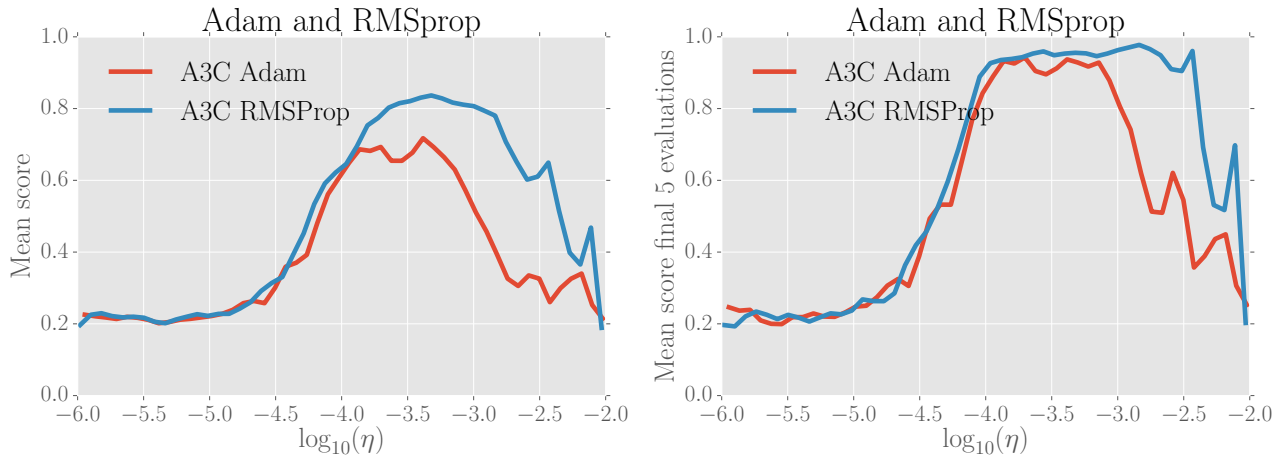


Figure A.2: Comparison of RMSprop vs. Adam. It can be seen that the RMSprop optimizer outperforms Adam substantially. However, it should be noted that improvements to the Adam optimizer could be achieved by further tuning the decay parameters.

A.1.2 Adam as a gradient descent optimizer

Other than the RMSprop algorithm, the Adam optimizer is a popular choice for gradient descent optimization. The Adam optimizer was introduced by (Kingma and Ba, 2014). Similar to the RMSprop algorithm as described above, this algorithm makes use of a running average of the squared gradient. However, (Kingma and Ba, 2014) explicitly refer to this as being the second order expectation of the gradient. They also keep track of the first order expectation of the gradient, which is simply a running average of the gradient itself. Moreover, (Kingma and Ba, 2014) suggest to rescale the moving averages to take into account the initial bias towards zero. They define the following update equations:

$$\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} + (1 - \beta_1)\boldsymbol{g}, \tag{A.1}$$

$$\boldsymbol{v} \leftarrow \beta_2 \boldsymbol{v} + (1 - \beta_2)\boldsymbol{g}^2, \tag{A.2}$$

$$\hat{\boldsymbol{m}} \leftarrow \frac{\boldsymbol{m}}{1 - \beta_1^t}, \tag{A.3}$$

$$\hat{\boldsymbol{v}} \leftarrow \frac{\boldsymbol{v}}{1 - \beta_2^t}, \tag{A.4}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\hat{\boldsymbol{m}}}{\sqrt{\hat{\boldsymbol{v}} + \epsilon}}, \tag{A.5}$$

in which $\boldsymbol{m}$ and $\boldsymbol{v}$ are the first and second order moments of the gradient, respectively. The rescaled moments are given by $\hat{\boldsymbol{m}}$ and $\hat{\boldsymbol{v}}$. Again, all operations are element-wise.

Due to limited time and the fact that this variation would presumably not deliver major improvements, we halted further investigation of the Adam optimizer. We now briefly discuss the initial attempt for the Adam optimizer on Catch. We set $\beta_1 = 0.9$ and $\beta_2 = 0.99$, following the advise of (Kingma and Ba, 2014). Similar to the RMSprop algorithm, we shared the gradient statistics across all threads. Figure A.2 provides the results of our single parameter sweep. It can be seen that the Adam optimizer with these parameters performs worse than the RMSprop optimizer. Perhaps using the first order moment to build up an averaged direction can be problematic for RL, as the target function's distribution is guaranteed to change over time, as well as the input distribution. Therefore, the desired direction might abruptly change, which implies that the average direction is not always a safe choice.

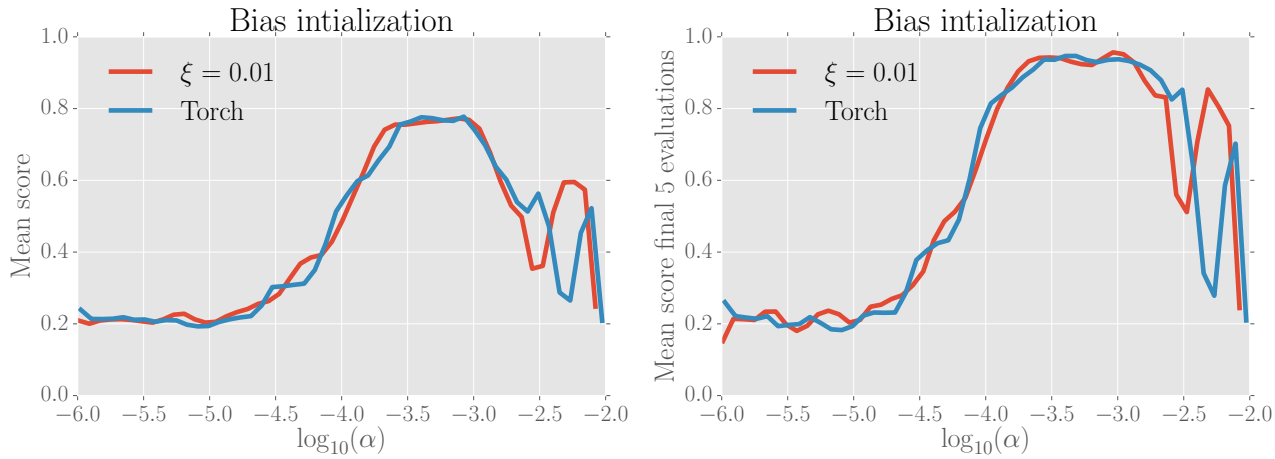


Figure A.3: Comparison bias initializations with default A3C FF.

A.1.3 Bias initialization

Another way of counteracting the zero-gradient as discussed in Section 6.3.2 issue is by initializing the biases to a small non-zero constant rather than initializing them from a uniform distribution as done in the Torch library. We have explored the performance difference when initializing the biases at $\epsilon = 0.01$. The results are depicted in Figure A.3. Despite the fact that the feature maps are now less likely to become populated by zeros by chance, we cannot observe significant improvements. Given that we are only viewing the results of between 50 or 100 different runs per line with a sampled learning rate, it is likely that sudden deviations from smooth patterns are due to outliers in terms of training processes.

Appendix B

Gender Classification with Local Weight Sharing Layers

Gender classification

To determine whether our local weight sharing layers work outside the Atari domain, we consider a five-fold cross-validation experiment on gender recognition using the Adience dataset as put forward by (Eidinger et al., 2014). After selecting the images that have a gender defined at all, there remain 6422 images of women and 5772 images of men. For a more detailed specification of the folds in the data see Table B.1. For the sake of validation, we compare the performance of a DNN that uses our spatial interpolation weight sharing layer with a 'default' DNN. The default in our case is almost identical to the DNN used by (Levi and Hassner, 2015) with which they obtained state-of-the-art results on gender recognition and age estimation. Our exact hyperparameter settings are given in Table B.2.

Results Figure B.1 displays the accuracies that were measured for default DNN model on all five folds. It can be seen that the SISWS model with trainable centroid locations seems to marginally outperform the default CNN model. This supports the notion that such models would be beneficial for problems that are spatially consistent. A more detailed overview of all tested configurations is given in Table B.3. This table shows that choosing per-feature configurations for the centroids does not necessarily lead to improved performance. This could be caused by the fact that the performance is close to an upper bound given the amount of ambiguous and unclear images in the dataset. For a good impression of typical misclassifications, see (Van de Wolfshaar et al., 2015). Another explanation would be that the per-feature configuration employs a more difficult function to optimize which leads to neutralize the potential gain in complexity of the function.

Table B.1: Distribution among classes for the Adience dataset (Eidinger et al., 2014).

Fold	#men	#women
0	1474	1410
1	1157	1329
2	987	1207
3	1103	1275
4	1051	1201
Total	5772	6422

Table B.2: Overview of the hyperparameters for the DNNs for gender classification.

Hyperparameter	Value
Learning rate α	10^{-4}
Optimizer	Adam
Decay 1st moment estimate β_1	0.9
Decay 2nd moment estimate β_2	0.999
Numerical stability ϵ	10^{-8}
Activation function	ELU
Minibatch size	128

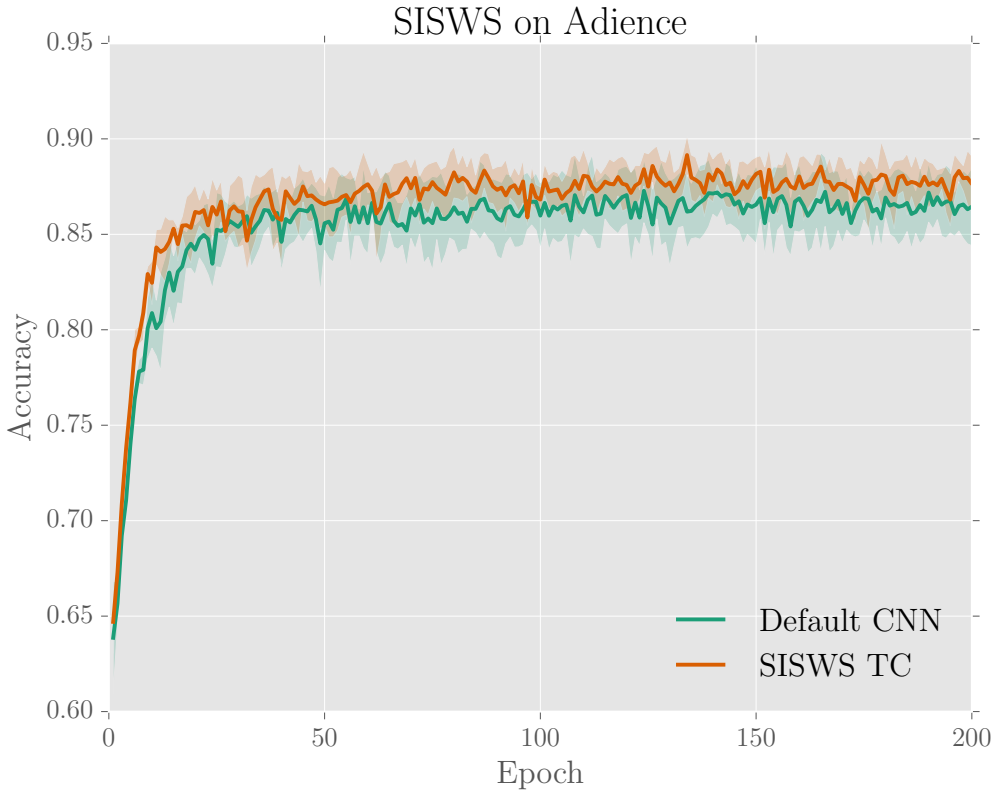


Figure B.1: Comparison of mean test accuracy for gender recognition on all five folds of the Adience dataset for the default DNN model vs the SISWS DNN model with trainable centroids. It can be seen that the SISWS model for nearly all evaluations performed slightly better than the default CNN architecture.

Table B.3: Overview of performances for gender classification on Adience.

Model	Accuracy (%) ± SE
Default CNN	87.33 ± 1.00
LWS	$\mathbf{88.56 \pm 0.70}$
Levi and Hassner (2015)	86.8 ± 1.4
Van de Wolfshaar et al. (2015)	87.20 ± 0.7
Afifi and Abdelhamed (2017)	**90.59**±?.?

Appendix C

Supplementary Material Learning Policy Quantization

C.1 Prototype gradients

To be able to explain the effects of the parameter settings in Chapter 7, we determine the gradients as present in GLPQ for the prototypes. Assume that we use the squared Euclidean distance function. Also assume that for the other prototypes, $\boldsymbol{w}_a$ is not the closest alternative of another class, then:

$$\frac{\partial \log \pi(s,a;\boldsymbol{\theta})}{\partial \boldsymbol{w}_k} = \frac{\partial}{\partial \boldsymbol{w}_i} \log \frac{\sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i)}{\sum_j \exp(\tau\mu_j)} \tag{C.1}$$

$$= \frac{\partial}{\partial \boldsymbol{w}_k} \log \sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i) - \frac{\partial}{\partial \boldsymbol{w}_k} \log \sum_j \exp(\tau\mu_j) \tag{C.2}$$

$$= \frac{1}{\sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i)} \frac{\partial \exp(\tau\mu_k)}{\partial \boldsymbol{w}_k} - \frac{1}{\sum_j \exp(\tau\mu_j)} \frac{\partial \exp(\tau\mu_k)}{\partial \boldsymbol{w}_k} \tag{C.3}$$

$$= \tau \exp(\tau\mu_k) \frac{-2d_-}{(d_k+d_-)^2} \frac{\partial d_k}{\partial \boldsymbol{w}_k} \left(\frac{1}{\sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i)} - \frac{1}{\sum_j \exp(\tau\mu_j)} \right) \tag{C.4}$$

$$= \tau \frac{4d_-}{(d_k+d_-)^2} (\boldsymbol{h} - \boldsymbol{w}_k) \left(\frac{\exp(\tau\mu_k)}{\sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i)} - \frac{\exp(\tau\mu_k)}{\sum_j \exp(\tau\mu_j)} \right), \tag{C.5}$$

from which we can see that the gradient is proportional to a Hebbian term $(\boldsymbol{h} - \boldsymbol{w}_k)$. Moreover, we see that the last factor could be reinterpreted:

$$\frac{\exp(\mu_k)}{\sum_{i:c(\boldsymbol{w}_i)=a} \exp(\mu_i)} - \frac{\exp(\mu_k)}{\sum_j \exp(\mu_j)} = P(\boldsymbol{w}_k \mid c(\boldsymbol{w}_k) = a, s) - P(\boldsymbol{w}_k \mid s), \tag{C.6}$$

where $P(\boldsymbol{w}_k \mid c(\boldsymbol{w}_k) = a, s)$ is the probability that the model assigns prototype k to a feature vector assuming that the k-th prototype belongs to action a, and $P(\boldsymbol{w}_k \mid s)$ is the probability that the feature vector is generated by the prototype $\boldsymbol{w}_k$. This is similar to the interpretation as seen in the work on the robust soft learning vector quantization algorithm (Seo and Obermayer, 2003). To let both cases be most similar, we would have to modify $\mu_a = d_a$ and minimize the corresponding cost function. It should be clear that the result of Equation (C.6) is guaranteed to be positive.

Note, if $\boldsymbol{w}_k$ is the closest alternative prototype for other classes such that $d_- = d_k$ for

$i \neq k$, then we obtain the following:

$$\frac{\partial \log \pi(s,a;\boldsymbol{\theta})}{\partial \boldsymbol{w}_k} = \frac{\partial}{\partial \boldsymbol{w}_k} \log \frac{\sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i)}{\sum_j \exp(\tau\mu_i)} \tag{C.7}$$

$$= \frac{\partial}{\partial \boldsymbol{w}_k} \log \sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i) - \frac{\partial}{\partial \boldsymbol{w}_k} \sum_j \exp(\tau\mu_j) \tag{C.8}$$

$$= \frac{1}{\sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i)} \frac{\partial \exp(\tau\mu_k)}{\partial \boldsymbol{w}_k} - \frac{1}{\sum_j \exp(\tau\mu_j)} \left(\frac{\partial \exp(\tau\mu_k)}{\partial \boldsymbol{w}_k} + \sum_{i \neq k} \frac{\partial \exp(\tau\mu_i)}{\partial \boldsymbol{w}_k} \right) \tag{C.9}$$

$$= \tau \frac{4d_-}{(d_k + d_-)^2} (\boldsymbol{h} - \boldsymbol{w}_k) \left(\frac{\exp(\tau\mu_k)}{\sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i)} - \frac{\exp(\tau\mu_k)}{\sum_j \exp(\tau\mu_j)} \right) - \frac{\sum_{i \neq k} \exp(\tau\mu_i) \frac{\partial \mu_i}{\partial \boldsymbol{w}_k}}{\sum_j \exp(\tau\mu_j)} \tag{C.10}$$

$$= \begin{aligned} & \tau \frac{4d_-}{(d_k + d_-)^2} (\boldsymbol{h} - \boldsymbol{w}_k) \left(\frac{\exp(\tau\mu_k)}{\sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i)} - \frac{\exp(\tau\mu_k)}{\sum_j \exp(\tau\mu_j)} \right) \\ & + 4\tau(\boldsymbol{h} - \boldsymbol{w}_k) \frac{\sum_{i \neq k} \exp(\tau\mu_i) d_i / (d_k + d_i)^2}{\sum_j \exp(\tau\mu_j)}, \end{aligned} \tag{C.11}$$

which can be interpreted as the addition of another attractive or repelling force to Equation (C.10). Whether the force is attractive of repulsive depends on the sign of the advantage. The added force is proportional to the probability that is assigned to other classes and the distance to other classes. Note that it has the same direction as Equation (C.10), since the quotient of the last term is guaranteed to be greater than 0.

Also, the above holds for any $\boldsymbol{w}_k$ such that $c(\boldsymbol{w}_k) = a$. The gradient for the prototypes such that $c(\boldsymbol{w}_\ell) \neq a$ are as follows:

$$\frac{\partial \log \pi(s,a;\boldsymbol{\theta})}{\partial \boldsymbol{w}_\ell} = \frac{1}{\sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i)} \frac{\partial \sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i)}{\partial \boldsymbol{w}_\ell} - \frac{1}{\sum_j \exp(\tau\mu_j)} \frac{\partial \sum_j \exp(\tau\mu_j)}{\partial \boldsymbol{w}_\ell}, \tag{C.12}$$

which means that if $\boldsymbol{w}_\ell$ does not correspond to the prototype such that $\boldsymbol{w}_\ell = \arg\min_{\boldsymbol{w}:c(\boldsymbol{w})\neq a} d(\boldsymbol{w}, \boldsymbol{h})$, the left term becomes zero. The remaining expression then becomes:

$$\frac{\partial \log \pi(s,a;\boldsymbol{\theta})}{\partial \boldsymbol{w}_\ell} = -\frac{1}{\sum_j \exp(\tau\mu_j)} \frac{\partial \exp(\tau\mu_\ell)}{\partial \boldsymbol{w}_\ell} \tag{C.13}$$

$$= -\frac{1}{\sum_j \exp(\tau\mu_j)} \tau \exp(\tau\mu_\ell) \frac{-2d_-}{(d_\ell + d_-)^2} \frac{\partial d_\ell}{\partial \boldsymbol{w}_\ell} \tag{C.14}$$

$$= -\tau \frac{4d_-}{(d_\ell + d_-)^2} (\boldsymbol{h} - \boldsymbol{w}_\ell) \frac{\exp(\tau\mu_\ell)}{\sum_j \exp(\tau\mu_j)}, \tag{C.15}$$

which is similar to the gradient obtained for incorrect prototypes in the case of RLSVQ (Seo and Obermayer, 2003) when $\mu_\ell = -d_\ell$. To see this, note that

$$\frac{\exp(\tau\mu_\ell)}{\sum_j \exp(\tau\mu_j)} = \mathbb{P}(\boldsymbol{w}_\ell \mid s). \tag{C.16}$$

It could also be that $\boldsymbol{w}_\ell = \arg\min_{\boldsymbol{w}_i:c(\boldsymbol{w}_i)\neq a} d(\boldsymbol{w}_i, \boldsymbol{h})$. In that case, the first term of equation C.12 becomes:

$$\frac{1}{\sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i)} \frac{\partial \sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i)}{\partial \boldsymbol{w}_\ell} = -\tau(\boldsymbol{h} - \boldsymbol{w}_\ell) \frac{\sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i) d_i / (d_\ell + d_i)^2}{\sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i)} \tag{C.17}$$

So that the full gradient is:

$$-\tau \frac{4d_-}{(d_k + d_-)^2}(\boldsymbol{h} - \boldsymbol{w}_\ell)\frac{\exp(\tau\mu_\ell)}{\sum_j \exp(\tau\mu_j)} - 4\tau(\boldsymbol{h} - \boldsymbol{w}_\ell)\frac{\sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i) d_i/(d_\ell + d_i)^2}{\sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i)} \tag{C.18}$$

The extension can again be interpreted as another attractive or repulsive force depending on the sign of the advantage.

Finally, there is the possibility that $\boldsymbol{w}_\ell$ is the closest alternative for all classes and not only for the class of a. Then, the gradient is:

$$\begin{aligned} -\tau \frac{4d_-}{(d_a + d_-)^2}(\boldsymbol{h} - \boldsymbol{w}_\ell)\frac{\exp(\tau\mu_\ell)}{\sum_j \exp(\tau\mu_j)} - 4\tau(\boldsymbol{h} - \boldsymbol{w}_\ell)\frac{\sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i) d_i/(d_\ell + d_i)^2}{\sum_{i:c(\boldsymbol{w}_i)=a} \exp(\tau\mu_i)} \\ - 4\tau(\boldsymbol{h} - \boldsymbol{w}_\ell)\frac{\sum_{i:c(\boldsymbol{w}_i)\neq c(\boldsymbol{w}_\ell)} \exp(\tau\mu_i) d_i/(d_\ell + d_i)^2}{\sum_j \exp(\tau\mu_j)} \end{aligned} \tag{C.19}$$

From the above it follows that GLPQ in terms of gradients puts more emphasis on those prototypes that are located near class boundaries.

C.2 Supplementary experiments

Temperature sweep for LPQ We also tested the effect of the temperature parameter without any scheduling on the default LPQ algorithm. The resulting sweep for the temperature can be seen in Figure C.1.

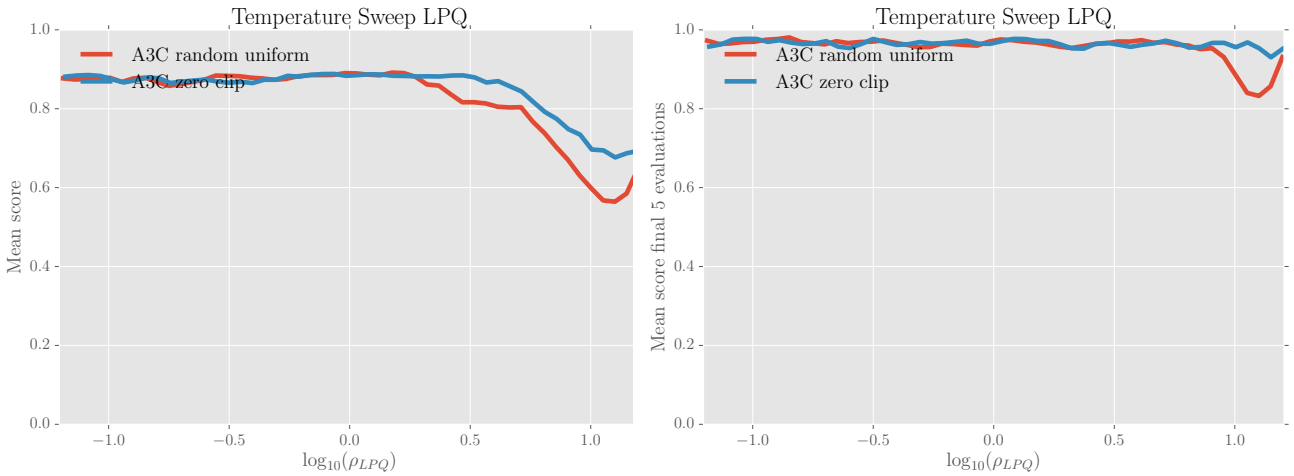


Figure C.1: Temperature sweep for the default LPQ algorithm. The random uniform and zero clipping refer to different initialization strategies for the prototypes.

Bibliography

Mahmoud Afifi and Abdelrahman Abdelhamed. Afif4: Deep gender classification based on adaboost-based fusion of isolated facial features and foggy faces. *arXiv preprint arXiv:1706.04277*, 2017.

Oron Anschel, Nir Baram, and Nahum Shimkin. Deep reinforcement learning with averaged target DQN. *CoRR*, abs/1611.01929, 2016.

Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen. Deepmind lab. *CoRR*, abs/1612.03801, 2016. URL http://arxiv.org/abs/1612.03801.

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47: 253–279, 06 2013.

Charles Blundell, Benigno Uria, Alexander Pritzel, Yazhe Li, Avraham Ruderman, Joel Z Leibo, Jack Rae, Daan Wierstra, and Demis Hassabis. Model-free episodic control. *arXiv preprint arXiv:1606.04460*, 2016.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

Augustin Cauchy. Méthode générale pour la résolution des systemes déquations simultanées. *Comp. Rend. Sci. Paris*, 25(1847):536–538, 1847.

Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

Robert H Crites and Andrew G Barto. Elevator group control using multiple reinforcement learning agents. *Machine learning*, 33(2):235–262, 1998.

Harm De Vries, Roland Memisevic, and Aaron Courville. Deep learning vector quantization. In *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, 2016.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul): 2121–2159, 2011.

Eran Eidinger, Roee Enbar, and Tal Hassner. Age and gender estimation of unfiltered faces. *IEEE Transactions on Information Forensics and Security*, 9(12):2170–2179, 2014.

Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4): 193–202, 1980.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016. URL http://www.deeplearningbook.org.

Klaus Greff, Rupesh K Srivastava, and Jürgen Schmidhuber. Highway and residual networks learn unrolled iterative estimation. *arXiv preprint arXiv:1612.07771*, 2016.

Caglar Gulcehre, Marcin Moczulski, and Yoshua Bengio. Adasecant: robust adaptive secant method for stochastic gradient. *arXiv preprint arXiv:1412.7419*, 2014.

Caglar Gulcehre, Jose Sotelo, Marcin Moczulski, and Yoshua Bengio. A robust adaptive stochastic gradient method for deep learning. *arXiv preprint arXiv:1703.00788*, 2017.

Hado van Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems*, pages 2613–2621, 2010.

Frank S He, Yang Liu, Alexander G Schwing, and Jian Peng. Learning to play in a day: Faster deep reinforcement learning by optimality tightening. *arXiv preprint arXiv:1611.01606*, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015a.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015b.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Gao Huang, Zhuang Liu, and Kilian Q Weinberger. Densely connected convolutional networks. *arXiv preprint arXiv:1608.06993*, 2016.

David H Hubel and Torsten N Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology*, 148(3):574–591, 1959.

David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154, 1962.

David H Hubel and Torsten N Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1):215–243, 1968.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre

luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, and Jonathan Ross. In-datacenter performance analysis of a tensor processing unit. 2017. URL https://arxiv.org/pdf/1704.04760.pdf.

Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. ViZDoom: A Doom-based AI research platform for visual reinforcement learning. In *IEEE Conference on Computational Intelligence and Games*, pages 341–348, Santorini, Greece, Sep 2016. IEEE. URL http://arxiv.org/abs/1605.02097. The best paper award.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. *arXiv preprint arXiv:1706.02515*, 2017.

T. Kohonen. Improved versions of learning vector quantization. In *IJCNN International Joint Conference on Neural Networks*, pages 545–550 vol.1, June 1990. doi: 10.1109/IJCNN.1990.137622.

Teuvo Kohonen. Learning vector quantization. In *Self-Organizing Maps*, pages 175–189. Springer, 1995.

Teuvo Kohonen, Jussi Hynninen, Jari Kangas, Jorma Laaksonen, and Kari Torkkola. LVQ PAK: The learning vector quantization program package. Technical report, Technical report, Laboratory of Computer and Information Science Rakentajanaukio 2 C, 1991-1992, 1996.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Y. LeCun. Generalization and network design strategies. In R. Pfeifer, Z. Schreter, F. Fogelman, and L. Steels, editors, *Connectionism in Perspective*, Zurich, Switzerland, 1989. Elsevier. an extended version was published as a technical report of the University of Toronto.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553): 436–444, 2015.

Gil Levi and Tal Hassner. Age and gender classification using convolutional neural networks. In *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR) workshops*, June 2015. URL http://www.openu.ac.il/home/hassner/projects/cnn_agegender.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.

Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

Aravindh Mahendran and Andrea Vedaldi. Visualizing deep convolutional neural networks using natural pre-images. *International Journal of Computer Vision*, pages 1–23, 2016.

Mortimer Mishkin, Leslie G. Ungerleider, and Kathleen A. Macko. Object vision and spatial vision: two cortical pathways. *Trends in Neurosciences*, 6:414 – 417, 1983. ISSN 0166-2236. doi: http://dx.doi.org/10.1016/0166-2236(83)90190-X. URL http://www.sciencedirect.com/science/article/pii/016622368390190X.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Volodymyr Mnih, Nicolas Heess, Alex Graves, and Koray Kavukcuoglu. Recurrent models of visual attention. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2204–2212. Curran Associates, Inc., 2014. URL http://papers.nips.cc/paper/5542-recurrent-models-of-visual-attention.pdf.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy P Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *arXiv preprint arXiv:1602.01783*, 2016.

Andrew W Moore and Christopher G Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13(1):103–130, 1993.

Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. Massively parallel methods for deep reinforcement learning. *CoRR*, abs/1507.04296, 2015. URL http://arxiv.org/abs/1507.04296.

Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

Karthik Narasimhan, Tejas D Kulkarni, and Regina Barzilay. Language understanding for textbased games using deep reinforcement learning. 2015.

Brendan O'Donoghue, Remi Munos, Koray Kavukcuoglu, and Volodymyr Mnih. PGQ: Combining policy gradient and q-learning. *arXiv preprint arXiv:1611.01626*, 2016.

Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped DQN. *arXiv preprint arXiv:1602.04621*, 2016.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

Alexander Pritzel, Benigno Uria, Sriram Srinivasan, Adrià Puigdomènech, Oriol Vinyals, Demis Hassabis, Daan Wierstra, and Charles Blundell. Neural episodic control. *arXiv preprint arXiv:1703.01988*, 2017.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter Learning Internal Representations by Error Propagation, pages 318–362. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-68053-X. URL http://dl.acm.org/citation.cfm?id=104279.104293.

Atsushi Sato and Keiji Yamada. Generalized learning vector quantization. In *Advances in neural information processing systems*, pages 423–429, 1996.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61: 85–117, 2015.

Petra Schneider, Michael Biehl, and Barbara Hammer. Adaptive relevance matrices in learning vector quantization. *Neural Computation*, 21(12):3532–3561, 2009.

Sambu Seo and Klaus Obermayer. Soft learning vector quantization. *Neural Computation*, 15(7):1589–1604, 2003. doi: 10.1162/089976603321891819. URL http://dx.doi.org/10.1162/089976603321891819.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

David Sussillo and LF Abbott. Random walk initialization for training very deep feedforward networks. *arXiv preprint arXiv:1412.6558*, 2014.

Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. 2017.

Gabriel Synnaeve, Nantas Nardelli, Alex Auvolat, Soumith Chintala, Timothée Lacroix, Zeming Lin, Florian Richoux, and Nicolas Usunier. Torchcraft: a library for machine learning research on real-time strategy games. *arXiv preprint arXiv:1611.00625*, 2016.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, 2014.

Yuan Tang. Tf. learn: Tensorflow's high-level module for distributed machine learning. *arXiv preprint arXiv:1612.04251*, 2016.

Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

Sebastian Thrun and Anton Schwartz. Issues in using function approximation for reinforcement learning. In *Proceedings of the 1993 Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum*, 1993.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2), 2012.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

John N Tsitsiklis, Benjamin Van Roy, et al. An analysis of temporal-difference learning with function approximation. *IEEE transactions on automatic control*, 42(5):674–690, 1997.

Leslie G. Ungerleider and James V. Haxby. 'what' and 'where' in the human brain. *Current Opinion in Neurobiology*, 4(2):157 – 165, 1994. ISSN 0959-4388. doi: http://dx.doi.org/10.1016/0959-4388(94)90066-3. URL http://www.sciencedirect.com/science/article/pii/0959438894900663.

Jos Van de Wolfshaar, Mahir F Karaaba, and Marco A Wiering. Deep convolutional neural networks and support vector machines for gender recognition. In *Computational Intelligence, 2015 IEEE Symposium Series on*, pages 188–195. IEEE, 2015.

S. van den Dries and M. A. Wiering. Neural-fitted td-leaf learning for playing othello with structured neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 23(11):1701–1713, Nov 2012. ISSN 2162-237X. doi: 10.1109/TNNLS.2012.2210559.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR, abs/1509.06461*, 2015.

Hado P Van Hasselt, Arthur Guez, Matteo Hessel, Volodymyr Mnih, and David Silver. Learning values across many orders of magnitude. In *Advances in Neural Information Processing Systems*, pages 4287–4295, 2016.

Andreas Veit, Michael Wilber, and Serge Belongie. Residual networks are exponential ensembles of relatively shallow networks. *arXiv preprint arXiv:1605.06431*, 1, 2016.

Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

MA Wiering. Multi-agent reinforcement learning for traffic light control. In *Machine Learning: Proceedings of the Seventeenth International Conference (ICML'2000)*, pages 1151–1158, 2000.

Marco Wiering and Martijn Van Otterlo. Reinforcement learning: state-of-the-art. *Adaptation, Learning, and Optimization*, 12, 2012.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Bernhard Wymann, E Espié, C Guionneau, C Dimitrakakis, R Coulom, and A Sumner. Torcs, the open racing car simulator. *Software available at http://torcs. sourceforge. net*, 2000.

Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European Conference on Computer Vision*, pages 818–833. Springer, 2014.

Jian Zhang, Ioannis Mitliagkas, and Christopher Ré. Yellowfin and the art of momentum tuning. *arXiv preprint arXiv:1706.03471*, 2017.

Y-T Zhou, Rama Chellappa, Aseem Vaid, and B Keith Jenkins. Image restoration using a neural network. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 36(7):1141–1151, 1988.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. 2017. URL https://arxiv.org/abs/1611.01578.