Extending the Deep Q Network to Recurrent Neural Networks

Nicholas Dingwall
Supervised by David Barber

A project submitted in partial fulfillment
of the requirements for the degree of
Master of Science, Machine Learning
of
University College London.

Department of Engineering

September 2, 2015

Abstract

Using neural networks as functional approximations in reinforcement learning has traditionally proved difficult due to instabilities when approximating value functions with nonlinear functions. This is particularly the case with Q Learning, but Google DeepMind have had considerable success with their Deep Q Network (DQN). This suffers from the limitation that you must be able to *observe* a Markov state in order to learn an optimal policy. In this project, I present an extension of the DQN to recurrent neural networks: the Recurrent DQN, or RDQN. I test the new model on three toy games, and show that while it performs poorly on stochastic games, it is able to beat the DQN in a game that requires memory, and to match it on deterministic games that do not.

Acknowledgements

I am indebted to Alexander Botev's prior work on implementing some aspects of the Deep Q Network and Recurrent Deep Q Network used in this project, as well as in programming the emulators for the Pong and Pong2 games.

Also to my supervisor, David Barber, whose advice and direction was instrumental in my completion of this project.

Contents

List of Figures

Chapter 1

Introduction

In this project, I introduce the Recurrent Deep Q Network (RDQN), an extension of Google DeepMind's Deep Q Network [2, 3] to recurrent neural networks. I compare the performance of these two models on three different games.

In chapters 2 and 3, I provide a brief overview of reinforcement learning and neural networks respectively to provide a basis such that the reader may fully understand the following chapters.

Then in chapters 4, 5 and 6, I introduce the three games, Pong, Pong2 and Transporter Man, and compare the performance of both models on each game.

In chapter 7, I detail some of the technical aspects about how to train neural networks that proved important in getting my models to learn to successfully play the games.

Finally, in chapter 8, I offer some conclusions and discuss the possibility of extending the new RDQN to more complex games.

Chapter 2

Reinforcement Learning[1]

2.1 Introduction

Reinforcement learning can be defined as "learning from interaction" [1, p. 3], or more specifically as "a computational approach to understanding and automating goal-directed learning and decision-making" [1, p. 15]. It reflects the way a human infant might learn: when presented with a choice of actions (to move an arm up or down, for example) a child does not receive explicit supervision giving a 'correct' choice. But that does not mean the learning is therefore *unsupervised*. Instead, the child has access to the future *effects* of his or her choice, and if these can be mapped to some quantitative (or qualitative) reward, the child may learn to perform useful actions even without supervision.

We replace the infant with an 'agent' that is able to act in an environment, and allow this agent to observe this environment. It is up to us to define a reward that the agent will seek to maximise by learning some abstract (and possibly stochastic) rules of behaviour that we will call its *policy*, denoted by π.

The ability to learn in these situations is important since in many interesting problems we do not have an expert whose behaviour we can mimic, and if we do, we might consider that problem solved. That is because it is often easier to define the criteria for success (that is, a reward function) than it is to describe how to achieve success. For example, if we wish to teach a robot to run, it would be difficult to explicitly write down the equations that should govern the motion of each joint, but it is easy to measure and reward its speed. The result is that the robot may be able to find a better solution (a higher top speed) than the best one an engineer may be able to programme. That humans are not born able to walk but nevertheless almost all learn to do so hints at the value of this regime of learning: the complexity of the problem, given the variability in limb lengths and so on, is such that it was evidently

[1] Content and notation in this chapter is largely based on [4] and [1].

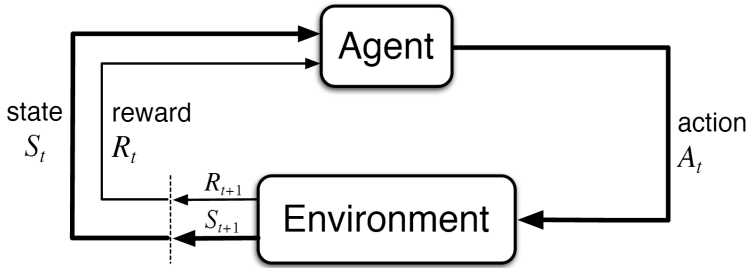


Figure 2.1: The agent-environment interface, from [1, p. 52].

not possible for evolution to have pre-programmed the necessary motor functions into our brains.

So in summary reinforcement learning concerns itself with how to choose from a set of actions to maximise a reward.

2.2 Terminology

We have already introduced some key terminology, but it will be worth being (still) more explicit.

We will consider the *environment* to be anything that affects the future but is not directly under the agent's control. For example, in a game of poker, the environment consists of the order of cards in the deck, every player's hand, the bids (and remaining balance) of each player, any rules regulating the behaviour of the agent and other players (including the dealer), the rules dictating who wins the game and the personality and preferences of the other players.

The environment will also provide the agent with a *reward signal*, a numerical score based on the current state of the environment. In poker, this will be zero until the end of the game, when the reward will be the amount won or lost. We consider that the environment and the agent operate at discrete time intervals; if the environment operates continuously (for instance, the robot running in the real world), then this is analogous to the agent observing the environment at discrete time steps.

We will consider the subset of information from the environment upon which the agent bases its decision the *state*, $\mathcal{S}$. This state may be everything that the agent can observe in the environment, but in many cases we may wish for the agent to learn to find the most pertinent details about the environment and discard everything else. Part of the process of reinforcement learning may be to learn a good low-dimensional representation of the state that is sufficient to behave optimally.

The agent has access to a number of *actions* drawn from a set $\mathcal{A}$ that may or may not affect the environment (but at least one action must have an effect on the environment for there to anything to learn to control). The mechanics of this

interaction is illustrated in figure 2.1. At time step t, the agent observes the current state s_t and uses this to select an action a_t. This action affects the environment, and so at the next time step, $t+1$, the observed state s_{t+1} and the received reward r_{t+1} are determined both by the mechanics of the environment and by the intervention of the agent (in the form of its action a_t). Observing a reward, r_t, allows the agent to evaluate its past behaviour, and to therefore improve its decision-making process. This relies on an assumption that the reward for doing action a in some state s and ending up in state s' comes from some fixed distribution $\mathcal{R}^a_{ss'}$.

This cycle continues until a state is *terminal* where no further actions or rewards are possible. As such, we can compartment a sequence of states, actions and rewards into an *episode* of actual experience, consisting of an ordered list of transition.

$$\mathcal{E} = \{\{s_1, a_1, r_1\}, \{s_2, a_2, r_2\}, \ldots, \{s_T, a_T, r_T\}\} \tag{2.1}$$

After a terminal state, a new episode commences in either a fixed initial state or in a state drawn from a distribution of initial states. We must also note that it is possible for there to be no terminal states; in that case, we can either arbitrarily choose a criteria to end an episode and begin a new one, or learn from a single, unending episode.

2.3 Markov and non-Markov Property

A state is considered to have the *Markov property* if the current state contains all useful information from its past experience. As Sutton and Barto put it,

> [...] we don't fault an agent for not knowing something that matters, but only for having known something and then forgotten it! [1, p. 62]

So a state is Markov if the next state and reward are independent of all but the most recent state and action or, more formally, if it satisfies:

$$\begin{aligned} p(s_{t+1} = s', r_{t+1} = r' \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0) \\ = p(s_{t+1} = s', r_{t+1} = r' \mid s_t, a_t, r_t) \quad \forall\, t \end{aligned} \tag{2.2}$$

where p is the transition distribution of the environment — that is, the probability of transitioning from state s_t to state s_{t+1} after performing action a_t. We include r_t and r_{t+1} in case the reward function depends on more than just the current state.

In some cases, the observations we can make at a given time step may be inadequate. However, we may be able to augment the state representation with additional information from previous time steps such that we do have sufficient information. For example, consider an object moving with constant velocity through 2D

space where our observations are of the position of the object in the form $[x_t, y_t]^\top$. This will not be enough to predict the object's position at the next time step, violating Equation 2.2. However, if we extend the 2-dimensional state space into a 4-dimensional one, $s_t = [x_t, y_t, x_{t-1}, y_{t-1}]^\top$, then we have the information to predict that $x_{t+1} = x_t + (x_t - x_{t-1})$ and $y_{t+1} = y_t + (y_t - y_{t-1})$. An important distinction emerges: it is the *state representation*, and not the *observation*, that is Markov or non-Markov. This will be an important consideration, since the key contribution of this project is a model for reinforcement learning (the Recurrent Deep Q Network – see section 3.10) that copes with states that cannot be augmented into Markov states simply by concatenating observations.

2.4 Value Functions

In order to be able to choose an action, we must have a measure of how valuable each action is. The problem is largely one of attribution: if our rewards are received sparsely, how are we to assign credit to each of the actions the agent took? A simple idea is to measure the total reward, or *return*, we receive after that time step. But intuitively, we would want to assign more credit to the actions that were taken close to the time step in which we received a reward, but still to assign some credit to the earlier actions in case those were instrumental in achieving the reward. We can achieve these aims through *discounting*: at each time step we multiply the future reward by a discount factor $0 < \gamma \leq 1$ to find the *discounted return*:

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \\ &= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \end{aligned} \tag{2.3}$$

where the infinite sum is valid if we treat a terminal state as one we stay in indefinitely without receiving any further reward. It also has the additional benefit of coping with infinitely-long episodes since, provided the rewards are bounded, $\gamma^m r_{t+m+1} \to 0$ as $m \to \infty$ for $\gamma < 1$.

For γ close to 0, we focus predominantly on short-term rewards; as γ approaches 1, we look further and further into the future, reducing the special significance of short-term rewards.

The aim of reinforcement learning, then, is to choose the action from some state that maximises the expected discounted return (hereafter simply referred to as the *return*), and so we define the state-action value function, $Q^\pi(s,a) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, where the π denotes that the value of a state will depend on the policy that the agent

is following:

$$Q^{\pi}(s,a) = \langle R_t | s_t = s, a_t = a, \pi \rangle_p. \tag{2.4}$$

A good reinforcement learner will find a policy $\pi^\star$ that always selects the action $a^\star$ from a given state s that maximises equation 2.4, in which case we define $Q^\star(s,a) = Q^{\pi^\star}(s,a)$ as the *optimal policy*.

We can rewrite the above equation by observing that

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \\ &= r_{t+1} + \gamma R_{t+1} \end{aligned}$$

and so, and in particular for the optimal policy $\pi^\star$,

$$\begin{aligned} Q^\star(s,a) &= \langle R_t | s_t = s, a_t = a, \pi \rangle_p \\ &= \langle r_{t+1} + \gamma \max_a Q^\star(s_{t+1}, a_{t+1}) | s_t = s, a_t = a, \pi \rangle_p \qquad (2.5) \\ &= \sum_{s'} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q^\star(s', a') \right). \qquad (2.6) \end{aligned}$$

where $\mathcal{P}^a_{ss'} = p(s_{t+1} = s' | s_t = s, a_t = a)$ and $\mathcal{R}^a_{ss'}$ is the reward for moving from state s to state s' by performing action a. This is the famous *Bellman optimality equation* for the state-action value function. The true state-action value is the unique solution to this (nonlinear) equation. It is important to note that if we know this state-action value, we have solved the problem: greedily choosing the action $a^\star$ that maximises $Q^\star(s,a)$ is necessarily the optimal policy (since it accounts for the immediate reward and also the expected future rewards).

Different values of γ will lead to different optimal policies: this is to be expected. Consider an agent that operates in a financial market. It should learn to behave differently if we desire large short-term rewards versus a preference for long-term growth. The policy will therefore depend on the choice of γ, but I will –leave this dependence implicit.

2.5 Functional Approximations

Where the state and action spaces are small, we can explicitly learn $Q^\star(s,a)$ for all $(s,a) \in \mathcal{S} \times \mathcal{A}$. However, where the state space is large or continuous, this will be not be possible, and so instead we must use a functional approximation and learn parameters θ such that $f(s,a;\theta) \approx Q^\star(s,a)$. For the remainder of this project, we will only be considering parametrised approximations to Q, which we will call $Q(s,a;\theta)$ or just $Q(s,a)$, leaving the dependence of the parameters implicit.

In general, we are interested in functional approximations where there are fewer parameters of θ than there are state-action pairs. This means that modifying a single parameter will change the associated Q value for more than one state-action, and as a result, even the optimal parameters $\theta^\star$ may not correspond to $Q^\star$. This approximation error can be computed as

$$\mathcal{E}(\theta) = \sum_{s,a} p^\star(s,a) \left[Q(s,a;\theta) - Q^\star(s,a)\right]^2 \tag{2.7}$$

expressed here in terms of the optimal value function $Q^\star$. The function $p^\star(s_t, a_t)$ is the probability of being is state s_t and performing action a_t while following the optimal policy, and it depends on dynamics of the environment (which may be unknown). We could replace $Q^\star$ with Q^π and $p^\star$ with p^π to compute the approximation error for some other policy π: this is relevant since the best parameters will be those that accurately estimate the Q values for state-actions that are visited often, at the expense of accuracy in the estimates of states that are visited rarely.

The reinforcement learning problem then is reduced to learning the parameters that satisfy $\theta^\star = \text{argmin}_\theta \mathcal{E}(\theta)$.

2.6 Model-Based and Model-Free Learning

One way to apply equation 2.6 is to programme the agent with knowledge about the probability distribution $\mathcal{P}^a_{ss'} = p(s_{t+1} = s' | s_t = s, a_t = a)$ (or to explicitly learn it). Knowing this distribution amounts to knowing a model of the reinforcement learning problem. Dynamic programming is a method that will efficiently learn an optimal policy when $\mathcal{P}^a_{ss'}$ is known, but is beyond the scope of this project.

In many interesting problems, these transition dynamics are not known *a priori* and may be intractable to learn, especially where the state space is large or continuous. In this model-free setting, rather than explicitly learning a policy, we instead learn $Q^\star$. The optimal policy is then to act greedily with respect to $Q^\star$; that is, to choose the action $a^\star$ that maximises $Q^\star(s,a)$ when in some state s.

First, I will describe two algorithms to learn Q^π, the state-action value function for a specific policy π. Then in section 2.7 I will describe an algorithm to learn $Q^\star$.

If we simulate many episodes of experience following policy π, then we can use equation 2.4 to estimate $Q^\pi(s,a)$. We define a mean squared error

$$E = \frac{1}{2} p(s_t, a_t) \sum_t \left[R_t - Q(s_t, a_t)\right]^2 \tag{2.8}$$

Then we minimise this with respect to the parameters of Q:

$$\frac{\partial E}{\partial \theta} = \sum_{t:s_t=s,a_t=a} p(s_t,a_t)\left[R_t - Q(s_t,a_t)\right]\nabla_\theta Q(s_t,a_t). \tag{2.9}$$

There are good reasons (see section 2.8) to update this estimate in an online fashion (i.e. one transition at a time, and updating our parameters by taking a gradient step in the direction of the summand). Doing this is equivalent to using a single-sample estimate of the gradient, and we call this *Monte-Carlo* (MC) learning, and it relies on the fact that states and actions are distributed according to $p(s_t,a_t)$ when acting according to the policy. With a step-size parameter α we use the update

$$\theta \leftarrow \theta + \alpha\left[R_t - Q(s_t,a_t)\right]\nabla_\theta Q(s_t,a_t). \tag{2.10}$$

We choose the step-size parameter largely by considering the stochasticity of the environment (if the returns are noisy, then we will want to take only a small step in this direction, but if we are confident that the returns have low variance, we can take bigger steps) and the extent to which each parameter affects multiple state-action values. Since R_t provides an unbiased estimate of $Q(s,a)$, this is guaranteed to converge to a local optimum provided we decrease the step-size parameter over time appropriately (see [1, p. 39]).

Note that this method does not rely on knowing the dynamics of the environment. As such, MC learning is model-free.

A significant flaw of this approach is that updates can only be made at the end of an episode since we need to know R_t, which is prohibitively slow if episodes are long. It makes poor use of available data. For example, if we observe a novel state s for the first time, and acting leads us to a state s' that we have observed many times, it seems sensible to use our knowledge of s' to help us evaluate the value of performing action a in s.

We can use this intuition by bootstrapping from our existing estimates of Q, since Q values are the expectations of future reward as per equation 2.4. In a model-based setting, we could estimate the return R_t in equation 2.10 by the immediate reward plus the expectation of future return (called the one-step return):

$$\theta \leftarrow \theta + \alpha\left[r_t + \gamma\langle Q(s_{t+1},a)\rangle_\pi - Q(s_t,a_t)\right]\nabla_\theta Q(s_t,a_t). \tag{2.11}$$

where we use the expectation Q for the next observed state s_{t+1}. If the experience comes from an episode following the policy, it seems reasonable that we should use the actual action chosen in the episode as a single-sample estimate of $\langle Q(s_{t+1},a)\rangle_\pi$,

leading to the *Temporal Difference* (TD) learning update:

$$\theta \leftarrow \theta + \alpha \left[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] \nabla_\theta Q(s_t, a_t). \tag{2.12}$$

An important difference between MC- and TD-learning is that, because the TD target includes an estimate which is based on the same parameters that we wish to update, it is a biased target. Despite this, it has been proved to converge for the table-lookup case (where we store a Q value for every state-action pair) to the true value of Q^π [5] and for linear functional approximations, albeit only to a value close to Q^π [6], with an error bound provided by [7].

It is also known there can exist no guarantee of convergence for the nonlinear case [7]. This is problematic, since neural networks are exactly nonlinear function approximators. This need not threaten us unduly, since neural networks rarely can be shown to find optimal solutions, but they can nevertheless find very good solutions.

It is important to note that the convergence of an approximation may not be sufficient to actually act optimally in all states: if we use too simple an approximation, we may lose detail of important distinctions between similar states.

In this project we will ignore other versions of TD learning where we look further into the future before bootstrapping (for example, using the target $r_{t+1} + \gamma r_{t+2} + \gamma^2 Q(s_{t+2}, a_{t+2})$) because we are mostly interested in a specialisation of this one-step version called Q Learning.

2.7 Q Learning

So far, we have been considering *on-policy* learning methods that bootstrap from the agent's actual experience. However, we are usually not interested in the current behaviour of the agent, but in what behaviours would be optimal. Through *off-policy* learning, we can learn the optimal state-action value function even if we never observe it being followed for an entire episode. The key is to replace the observed action in the TD target $r_t + \gamma Q(s_{t+1}, a_{t+1})$ with the optimal action, so $Q(s_{t+1}, a_{t+1})$ becomes $\max_a Q(s_{t+1}, a)$. This uses our prediction of the optimal value from state s_{t+1}, rather than our estimate of the value of the performed action a_{t+1}. So our update equation (briefly considering $Q(s,a)$ as a lookup table of Q values for each state-action pair) is now

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]. \tag{2.13}$$

We can estimate the max by simply comparing the current estimates of $Q(s_{t+1},a)$ for all $a \in \mathcal{A}$ and choosing the largest. This can be shown [8] to converge to $Q^\star$ regardless of the policy being followed, provided every state-action pair continues to be updated. This requirement is explained in section 2.8. Naturally as our estimate of $Q^\star$ improves, the a that maximises $Q(s_{t+1},a)$ might change, but this doesn't lead to instabilities since the actual numerical values of $Q(s_{t+1},a_1)$ and $Q(s_{t+1},a_2)$ will be close when a_2 'overtakes' a_1.

When using a functional approximation, we modify equation 2.12:

$$\theta \leftarrow \theta + \alpha \left[r_t + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t) \right] \nabla_\theta Q(s_t,a_t). \tag{2.14}$$

However, off-policy bootstrapping can lead to divergence when combined with function approximations [9]. A possible solution that avoids the instability that results is presented in section 3.8.

2.8 Decreasing ε-Greedy Policy

It was noted earlier that the optimal policy is to act greedily with respect to $Q^\star$. However, until we know $Q^\star$ exactly, acting greedily with respect to our estimate may prevent us from discovering more successful behaviours that would result in more long-term reward.

For example, consider a room with two doors, A and B. Opening door A guarantees a reward of +1, while opening door B offers a 50% chance of a +10 reward. The optimal strategy is to always open door B. But if we start by opening door B and get no reward, and then open door A and get a reward of +1, $Q(s,A) = 1$ and $Q(s,B) = 0$. Acting greedily towards these estimates would result in only opening door A.

For this reason, the guarantee of convergence only works if the policy being followed tries every action from each state infinitely often (obviously this is in the limit of infinite experience). An natural solution would be to use a random policy, sampling an action uniformly at randomly from each state.

But this runs into a practical problem: in complex problems, the probability of visiting some important states by following a random policy will be vanishingly small. Consider a computer game with several levels: we can only begin to learn how to beat level 5 once we have, in some episode, completed levels 1 to 4. But completing a level by pressing a random series of buttons is very unlikely. Even if completing levels 1 to 4 only required that we choose the correct button on 10 occasions, with 5 buttons to choose from, the probability of reaching level 5 is $\frac{1}{5}^{10} \approx 10^{-7}$. At a rate of 10 button pushes per second, it would take around 2 years

before we can expect to have pressed the correct buttons in the correct order.

The solution to these conflicting problems is to begin with a random policy, but to then slowly increase the probability with which we act greedily with respect to our estimate of $Q^\star$. This describes the decreasing ε-greedy policy: we start with $\varepsilon_1 = 1$ and slowly decrease it such that $\varepsilon_t \to 0$ as $t \to \infty$. Then at some time step t, we act greedily with probability $1 - \varepsilon_t$, and otherwise select an action at random. As such, if there are n available actions (i.e. $|\mathcal{A}| = n$) then

$$p(a_t = a | s_t = s) = \begin{cases} \frac{\varepsilon_t}{n} & \text{if } a \neq a^\star \\ 1 - \varepsilon_t + \frac{\varepsilon_t}{n} & \text{if } a = a^\star \end{cases}. \tag{2.15}$$

Chapter 3

Neural Networks[1]

3.1 Description and Introduction

A natural neuron network – like the brain and nervous system in mammals – combines a large number of simple biological neurons to create a highly-connected whole that can perform highly-complex tasks.

The process of 'thought' can be understood to be in the interactions of these neurons. Since 1943 when Warren McCulloch and Walter Pitts first modelled an artificial neuron, this structure of many simple but highly-connected parts has inspired computer scientists trying to create learning algorithms.

But before considering how an entire network functions, let us focus on a sin-

[1] A more complete introduction to neural networks can be found in e.g. [10], upon which the notation for this chapter, particularly in sections 3.2.1 and 3.4, is based.

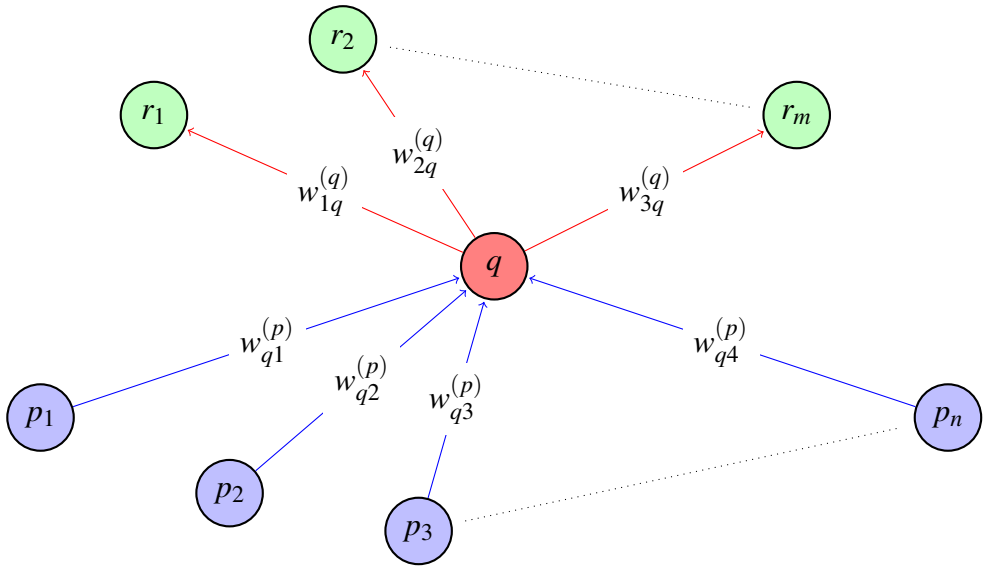


Figure 3.1: A neuron q with n incoming connections (from $p_1, p_2, p_3, \ldots, p_n$) and m outgoing connections (to $r_1, r_2, \ldots, r_m$).

gle neuron. An artificial neuron (except input and output neurons – see section 3.2) combines some numerical inputs, applies a transformation called an *activation function* and emits a real number. To avoid over-estimating the power of these individual parts, they are often referred to as *nodes* or *units*.

Figure 3.1 shows a single node that has n incoming connections and m outgoing connections, where each edge has an associated weight. A node multiplies the values from the incoming connections by the weights on the edges; its *activation* is the sum of these numbers, plus some bias term b:

$$a = w_{q1}^{(p)} p_1 + w_{q2}^{(p)} p_2 + \cdots + w_{qn}^{(p)} p_n + b. \tag{3.1}$$

Then it applies an activation function σ (see section 3.6), and passes the result $z = \sigma(a)$ along each of its outgoing connections (in this case, to $r_1, \ldots, r_m$):

An artificial *neural network* combines many of these nodes in order to approximate a function from inputs to outputs. Training a neural network means learning weights $\mathcal{W}$ that do a good job of approximating the function. In a supervised learning setting, the aim is to find weights such that the output (or *prediction*) $\hat{y}_i$ for an input $\mathbf{x}_i$ closely matches the true output y_i. In most cases, these weights are not unique.

3.2 Feed-Forward networks

The most common neural networks are *feed-forward* networks. The nodes are separated into layers, and each layer is fully connected with the next. This means that there are no cycles in the diagram: data enters the network in some *input* nodes and after a fixed number of computations arrives at the *output* nodes.

In between these input and output layers are some number of *hidden* layers, so-called because we generally never see – and are not interested in – the value of nodes in these layers.

Note that, since the input nodes exist to bring the data into the network, they have no associated activation function. In addition, the output nodes must map to the domain of the outputs, and as such may have a different function to the hidden nodes.

Figure 3.2 shows an example of a network with a single hidden layer. The input layer has four nodes, which would correspond to a 4-dimensional input (perhaps the x, y, z locations of an object plus time), and the output has two nodes, which would correspond to a 2-dimensional prediction (perhaps the polar and azimuth angles of the location where the object will enter a unit sphere).

A gradient-based algorithm to learn weights (*backpropagation* – see section

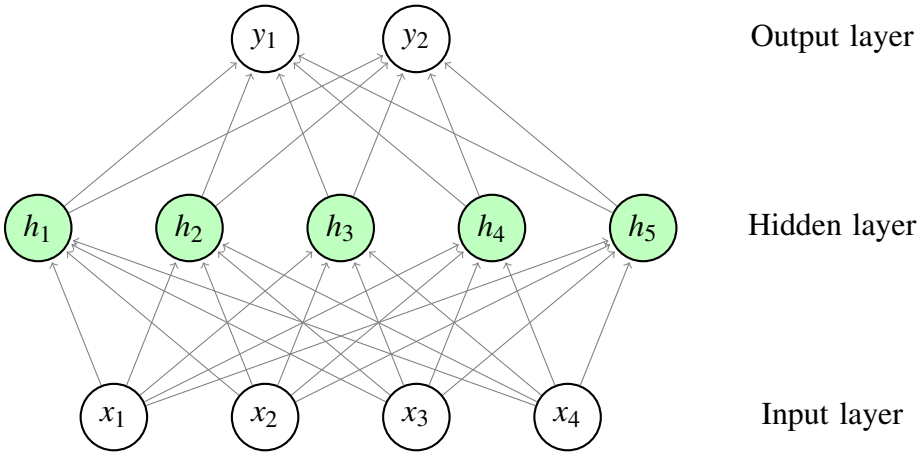


Figure 3.2: A feed-forward network with one hidden layer.

3.4) exists for feed-forward networks. To understand it, we must first go through the computations required for a forward pass from inputs to outputs for a single data point.

3.2.1 Forward Pass

Let us consider a network with d-dimensional inputs, d'-dimensional outputs and m hidden nodes. First we compute the activations for each of the nodes in the hidden layer as per Equation 3.1:

$$a_j^{(1)} = \sum_{i=1}^{m} w_{ji}^{(1)} x_i + b_j \tag{3.2}$$

where $w_{ji}^{(1)}$ denotes the weight from the ith input node to the jth node in the hidden layer. We can absorb the bias term into the sum if we invent a new node $x_0 = 1$; in that case, we can refer to the bias term by $w_{j0}^{(1)}$:

$$a_j^{(1)} = \sum_{i=0}^{m} w_{ji}^{(1)} x_i. \tag{3.3}$$

Now we apply a function σ to the activation in Equation 3.3 to find the output, z, for each of the hidden nodes:

$$z_j = \sigma(a_j^{(1)}). \tag{3.4}$$

We use these outputs to compute the activations of the output layer

$$a_j^{(2)} = \sum_{i=0}^{d} w_{ji}^{(2)} z_i^{(1)} \tag{3.5}$$

and finally apply (a potentially different) function to these:

$$\hat{y}_j = \widetilde{\sigma}\left(a_j^{(2)}\right). \tag{3.6}$$

Here there are d' elements $\hat{y}_j$ that make up the vector prediction $\hat{\mathbf{y}}$.

3.3 Error functions

To train a neural network (that is, to learn the weights), we need a measure of how good the current network weights are. The only requirement is that this function is differentiable. It is often referred to as a *cost* function since we use it to choose how heavily to penalise errors. In this project we will use the squared loss, where the total error on the N datapoints is the sum of squared differences between each correct value and the corresponding prediction:

$$E = \sum_{i=1}^{N} E^i \tag{3.7}$$

$$= \sum_{i=1}^{N} (y_i - \hat{y}_i)^2. \tag{3.8}$$

In some cases, other loss functions will be more effective – see e.g. [11].

3.4 Backpropagation

We can summarise the computations in Equations 3.3 to 3.6 by treating the sums as matrix multiplications:

$$\hat{\mathbf{y}} = \widetilde{\sigma}\left(\mathbf{W}^{(2)}\left(\sigma\left(\mathbf{W}^{(1)}\mathbf{x}\right)\right)\right). \tag{3.9}$$

As a result, we can use the chain rule to efficiently compute exact derivatives with respect to the parameters $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$.

Practically, we begin by noting that

$$\frac{\partial E^n}{\partial w_{ji}} = \frac{\partial E^n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}. \tag{3.10}$$

Using as shorthand $\delta_j = \frac{\partial E^n}{\partial a_j}$, and observing that $\frac{\partial a_j}{\partial w_{ji}} = z_i$ (by Equation 3.5), we obtain

$$\frac{\partial E^n}{\partial w_{ji}} = \delta_j z_i. \tag{3.11}$$

That is, the derivative of the error with respect to a network parameter is the δ_j for a node multiplied by the value of that node.

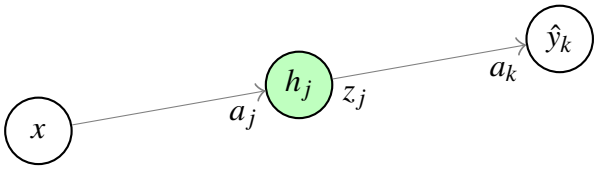


Figure 3.3: Schematic representation of a simple network to illustrate the backpropagation algorithm.

All that remains is to compute the δ_j for each output and hidden node. Figure 3.3 will help clarify the notation, and I use a superscript 2 to indicate that it refers to the output layer, or a superscript 1 for the hidden layer. For the K nodes in the output layer and the squared loss function,

$$\delta_k^2 = \frac{\partial E^n}{\partial a_k} = \frac{\partial y_k}{\partial a_k}\frac{\partial E^n}{\partial y_k} \tag{3.12}$$

$$= \widetilde{\sigma}'(a_k)\left[2(\hat{y}_k - y_k)\right]. \tag{3.13}$$

For a J nodes in the hidden layer, we use the δ_k^2s that we just calculated for the output layer:

$$\delta_j^1 = \frac{\partial E^n}{\partial a_j} = \sum_k \frac{\partial E^n}{\partial a_k}\frac{\partial a_k}{\partial a_j} \tag{3.14}$$

$$= \sum_k \delta_k^2 \frac{\partial a_k}{\partial a_j} \tag{3.15}$$

where the sum runs over all of the nodes to which h_j has an outgoing connection.

Since a_k (an activation in the output layer) is just a weighted sum of its inputs, which in turn are the outputs of the hidden layer, we can express this as

$$a_k = \sum_i w_{ki} z_i = \sum_i w_{ki}\sigma(a_i) \tag{3.16}$$

where i indexes the nodes in the hidden layer which have a connection to $\hat{y}_k$. Then

$$\frac{\partial a_k}{\partial a_j} = w_{kj}\sigma'(a_j) \tag{3.17}$$

since $\frac{\partial a_k}{\partial a_j} = 0$ for $i \neq j$. This leads to the backpropagation formula

$$\delta_j = \sigma'(a_j)\sum_k w_{kj}\delta_k. \tag{3.18}$$

We can summarise the backpropagation algorithm as follows:

1. Perform a feed-forward pass for some training data.
2. Compute errors using the error function 3.8.
3. Compute δ_k for the output layer using Equation 3.13.
4. Propagate this back to all the hidden nodes using Equation 3.18.
5. Use Equation 3.11 to compute the derivatives with respect to each of the network parameters.

These gradients can then be used in a gradient descent algorithm to incrementally improve the parameters.

3.5 Multilayer Networks

It is also worth noting that none of the above relies on the network having only a single hidden layer. In fact, the forward pass and backpropagation algorithms apply to feed-forward networks of any depth. Figure 3.4 shows a three-layer network with two hidden layers, and where the biases are shown in red. (We refer to it as a three-layer network since it has three layers of weights to learn).

To summarise the architecture used, I write the number of nodes in each hidden layer as an ordered list. The number of nodes in the input and output layers are specified by the context, so a game with a 100-dimensional input will necessarily have 100 input nodes, and if there are 5 actions to choose from, the network will have 5 output nodes, one for the Q value of each available action. So $[100, 50, 25, 10]$ indicates a network with four hidden layers in addition to the input and output layers, read from input to output, with 100, 50, 25 and 10 nodes respectively.

The Deep Q Network (section 3.8) uses a network of this type to provide a mapping from states to Q values for each action. The RDQN will use a different architecture (see section 3.10).

3.6 Node functions

We have specified the error function, but have so far left the node functions undefined. For this project, I use a linear output function (i.e. $\widetilde{\sigma}$ is just the identity), and use either a tanh or rectified linear function for the hidden nodes.

3.6.1 Linear node functions

We do not consider linear functions on the hidden nodes, because using them collapses the whole neural network to one with a single layer. We can see this by

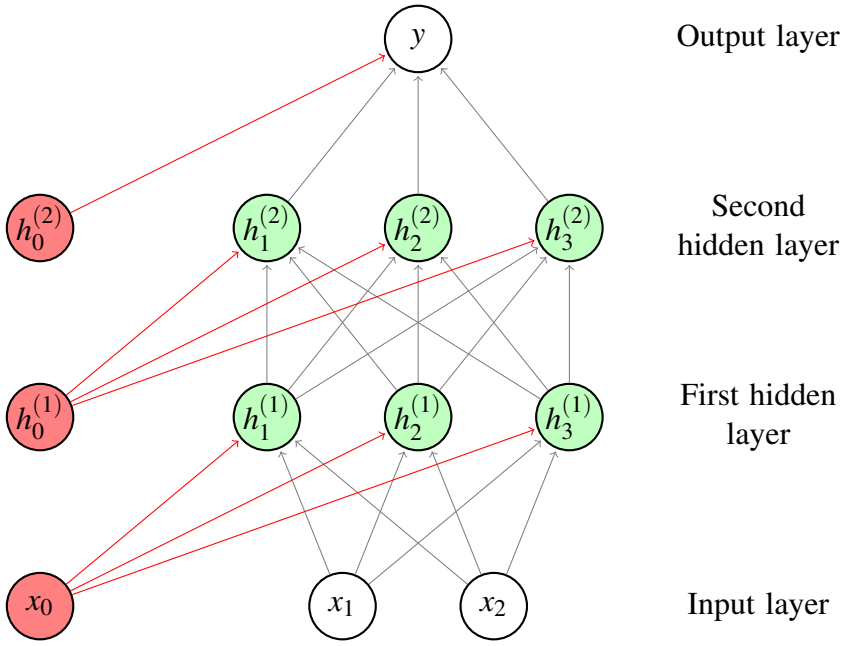


Figure 3.4: Example neural network mapping a 2-dimensional input to a 1-dimensional output, with two hidden layers each with 3 nodes. The red nodes are biases, fixed as equal to 1.

considering Equation 3.9 where $\widetilde{\sigma}$ and σ are both linear functions:

$$\hat{\mathbf{y}} = \mathbf{M}_1\left(\mathbf{W}^{(2)}\left(\mathbf{M}_2\left(\mathbf{W}^{(1)}\mathbf{x}\right)\right)\right) = \widehat{\mathbf{M}}\mathbf{x} \tag{3.19}$$

where $\widehat{\mathbf{M}} = \mathbf{M}_1\mathbf{W}^{(2)}\mathbf{M}_2\mathbf{W}^{(1)}$ is a single linear function of the input $\mathbf{x}$. We conclude that it is imperative that the function of the hidden nodes be a nonlinear function.

3.6.2 Threshold Node Functions

The simplest nonlinearity we can introduce is a threshold unit, or step function. This is designed to replicate the general behaviour of a brain neuron, which outputs a binary state: either firing if some threshold is met, or sending no signal otherwise. This is represented as

$$\text{threshold}(a) = \begin{cases} 1 & \text{if } a > \theta \\ 0 & \text{otherwise.} \end{cases} \tag{3.20}$$

for some specified threshold θ. Unfortunately, the gradient of this function is either 0 or undefined, and so the backpropagation algorithm will fail.

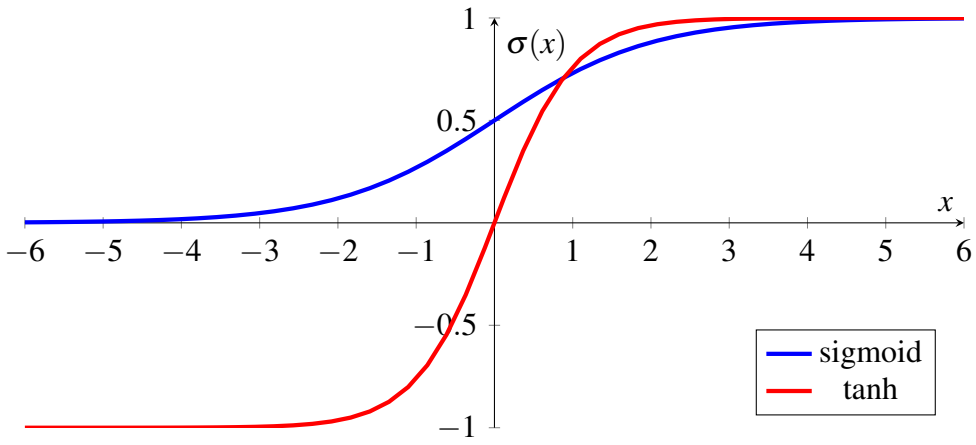


Figure 3.5: Plots of sigmoid and tanh activation functions.

3.6.3 Sigmoid Node Functions

An obvious replacement is the logistic sigmoid function, which has horizontal asymptotes at $\sigma(a) = 1$ and $\sigma(a) = 0$, but is differentiable everywhere.

$$\text{sigmoid}(a) = \frac{1}{1+e^{-x}}. \tag{3.21}$$

This function is illustrated in figure 3.5. Since the outputs are in the range $[0,1]$, this is a popular choice for the output layer (usually paired with the logistic loss function). However, in the hidden layers, it is more common to choose a transformation of this such that the range of outputs is $[-1,1]$.

3.6.4 Tanh Node Functions

From figure 3.5, we can see that the tanh function is a transformation of the sigmoid function, and it is easy to verify that

$$\tanh(a) = 2\,\text{sigmoid}(2a) - 1 = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}. \tag{3.22}$$

The reasons for choosing the tanh over the sigmoid are further discussed in section 7.5. A problem with both the tanh and sigmoid functions is that they saturate for large positive (or negative) x: the difference between $\sigma(x)$ and $\sigma(x+1)$ becomes arbitrarily small as x grows large. This means that the gradient tends towards 0, and so it is very difficult for the backpropagation algorithm to find a good downhill direction.

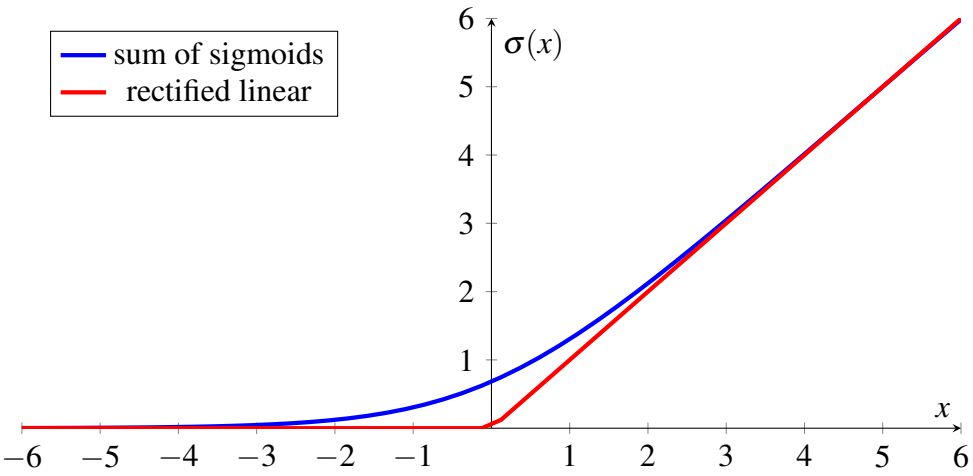


Figure 3.6: The rectified linear unit approximates a sum of sigmoids with differing biases.

3.6.5 Rectified Linear Units

If we consider the sigmoid function to be an analogy of the excitation of a biological neuron, then it would make sense to mix together a number of neurons that require a different level of excitation before becoming active. Figure 3.6 shows the sum of the first 10 terms of $\sum_n \text{sigmoid}(a+0.5-n)$. The rectified linear unit comes from the observation that the gradient of the positive part of this curve approaches 1, and the negative part approaches 0. Therefore, it can be approximated as

$$\text{relu}(a) = \max(0, a) \tag{3.23}$$

and this function is also shown in figure 3.6. That this function doesn't saturate at the positive end, and because it is very fast to compute, makes it a popular choice for neural networks. The gradient (1 for $x > 0$ and 0 elsewhere) is equally fast to compute.

3.7 Training Neural Networks

There is further detail on training neural networks in Chapter 7, but some points will be important before I introduce the DQN and RDQN.

In computing the error in Equation 3.8, it is inefficient to use all the training data for a single update. That is because there is likely to be considerable redundancy in the training samples (many similar samples with similar labels), and also because, particularly in the early stages of training, it will quickly become obvious in which direction to update the parameters. There is no point in getting a very good estimate for this direction, because once we take a step in that direction, the shape of the function that we are trying to minimise will have shifted. That's because

the error function is a function of both the true value and the prediction: once we improve the network's prediction accuracy, the types of mistakes it will make will change, and different parameter updates will be necessary to correct them.

On the other hand, updating parameters after every data point will lead to very high variance gradient updates: an unusual data point might produce a large error which will drag the parameters away from the optimal values. A very slow learning rate would be required to compensate for the noisy updates.

The solution is stochastic gradient descent: we compute the gradient on a small sample and take a small step in that direction, and repeat this until convergence. Section 7.1 goes into further detail about the practicalities of this. The size of the minibatch is a trade-off explored in e.g. [12].

3.8 Deep Q Network (DQN)

The Deep Q Network [2, 3, 13] is a deep learning model for reinforcement learning that the authors claim addresses problems in RL. They contrast the traditional successful applications of deep learning that require huge resources of labelled data with RL, where it must learn from an often-sparse reward signal that may be delayed by many timesteps from the action that directly caused it.

They identify the correlation between observations and the changing data distribution (a result of the agent learning better policies) as challenges to traditional approaches to RL using deep learning.

To address the problem of correlated inputs and the non-stationary distribution, they use *experience replay*, where the observed episodes are converted into a large collection of transitions, with a *transition* consisting of a state, the chosen action, a reward and the subsequent state, $e_t = (s_t, a_t, r_t, s_{t+1})$. The algorithm can then sample at random from this collection of ordered quadruplets. This reduces the variance of the updates, since consecutive experiences are likely to be correlated, while experiences drawn from different episodes or different time points in the same episode are unlikely to be. This sampling also means that a single update is likely to include transitions that corresponded to several different policies, which smooths the data distribution.

The sampling necessitates off-policy learning, and so they choose to use the Q Learning update (section 2.7). They parameterise a neural network Q with parameters θ and use the update

$$\theta \leftarrow \theta + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta') - Q(s_t, a_t; \theta) \right] \nabla_\theta Q(s_t, a_t) \tag{3.24}$$

where the parameters θ' in the target are kept fixed for some number of iterations

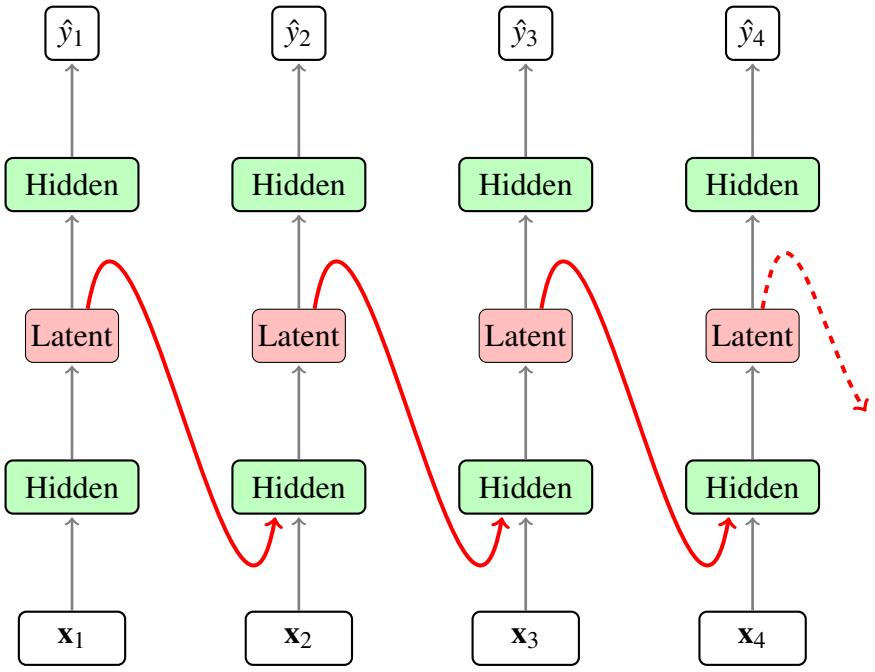


Figure 3.7: A schematic diagram of a recurrent neural network.

and then periodically updated to be equal to the current θ. This helps to reduce correlations between the predictions and the target, particularly in the case of stochastic games. A further discussion of this aspect is presented in section 8.1.

They use a feed-forward architecture with one input for each of the elements in the (pre-processed) observation $\mathbf{x}$, where an 'observation' consists of four consecutive frames concatenated and one output for each available action: these outputs are the Q values for that action in the observed state x.

3.9 Recurrent Neural Nets (RNNs)

Recurrent neural networks (RNNs) relax one of the requirements of neural networks: that they have no cycles. The name comes from the recurrent connections that allow the data to loop around inside the network, even while a new input arrives. They have proved to be wildly successful in some areas (see e.g. [14]).

There are many types of RNNs, but here we will be interested in ones with recurrent connections from one of the hidden layers to the input layer so that this hidden layer can act as additional input. Since this hidden layer may then learn a representation of the state with which to augment the next input, we will refer to it as a *latent layer*.

Figure 3.7 shows a representation of this, where the entire network has been

replicated four times, with each column representing a time step: at each time step, a new data point is fed in, and this is combined with the latent layer from the previous time step.

Note that this replication is simply a tool to visualise the network, since we only maintain a single set of weights: this is equivalent to tying corresponding weights in each column. The only exception is in the first column, where a separate set of weights may be learnt to initialise the latent state.

We call the part of the network from the inputs to the latent state the *filter network* and the part from the latent state to the output, the *output network*. Both the filter and output networks can be of arbitrary depth.

In my experiments, a single output network is learnt, but two distinct sets of weights are learnt for the filter network: one set for the first time step, and another for all remaining time steps. One reason to do this is because the input is a different size: $|\mathbf{x}|$ rather than $|\mathbf{x}| + |\text{latent layer}|$. But this could be easily avoided by inventing a new latent layer at $t = 0$ with all nodes equal to zero. A more important reason is because the latent state is initialised to zero, but we wish to let the network learn a better initialisation. The weights required to initialise it are likely to be different to those used to adjust it when new data arrives.

I use a similar notation to the feed-forward neural networks to describe the architecture, except that I use a semi-colon to indicate the location of the latent layer. For example, $[100, 50; 25, 10]$ indicates a recurrent network with a 50-dimensional latent layer (where each node in this layer is concatenated with the input at the next time step.

3.10 Proposed Recurrent Deep Q Network

An important consequence of this is that the latent state can serve as a *memory* for the network. If some important event occurs, then the DQN will not have access to it after 4 time steps. The DQN with recurrent connections could 'store' this information in the latent state indefinitely.

Also, because there are naturally connections between consecutive time steps, we no longer need to concatenate observations to model motion. In the simplest case, if the latent layer is the same size as the input layer, the network can simply use the identity for the filter network, recreate the observation on the latent layer, and pass this to the next time step, exactly replicating the behaviour of the DQN, but without the need for the programmer to explicitly pass in multiple observations.

The RDQN then, is a simple extension of the DQN. The target function remains the same, and the program for updating the target values is retained. There are two significant differences. The first is simply the addition of the recurrent connections.

The second is that we use a different sampling procedure. While the DQN can use experience replay to sample transitions, the RDQN needs the latent state as well as the observation to be able to predict the Q-values of each action. This means we must sample entire episodes: if we wish for the network to learn to remember significant information, we must show that information to it! To avoid suffering too deeply from correlated inputs, we sample multiple episodes for each gradient update.

In the games tested in this project, that does not pose a problem since the episodes are short, but I discuss some problems with longer episodes and a proposed remedy to this sampling procedure in section 8.4.2.

It does, however, seem to lead to problems with stochastic games, and this is discussed in depth in section 8.1.

Chapter 4

Experiment 1: Pong

4.1 Description of game

Our game of Pong is based on the classic Atari game released in 1972. In the original game, the player operates a bat (which can only be moved vertically) and plays against the computer to win at a tennis-like game. Figure 4.1 shows a still from the original game.

The key skill is to move the bat such that it intercepts the ball. It is this skill that we wanted our RL agent to learn, and so we made the bat a single pixel to test how well it could learn to precisely predict where the ball would cross the axis of the bat.

In order to avoid the added complexities of training convolutional neural networks, we worked on a very small version of the game, with the grid measuring 7 by 21 pixels, and forced the ball to exist precisely within a single pixel. To enforce this, we allowed the ball to travel at only 45° angles from the horizontal (either northeast or southeast). The ball 'bounces' whenever it encounters the top or bottom wall.

In order to focus on the core skill we wanted the RL to learn, we treated an episode as a single trajectory of the ball from the left of the screen to the right. This represents a passage of play from when the ball is hit by the opponent to when it is

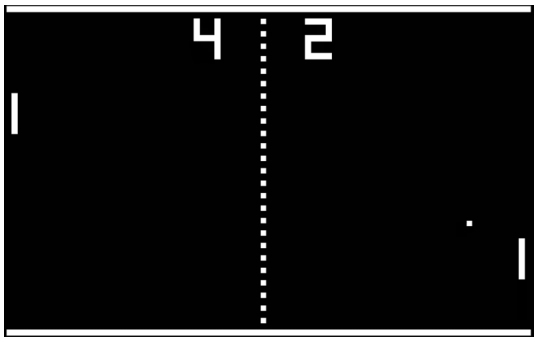


Figure 4.1: Still from the original Atari game of Pong, with the score at 4-2.

hit by the player.

To make the game more challenging, we programmed the ball to move at twice the speed of the bat (that is, the ball moves two pixels both horizontally and vertically in each time step; the bat can only move one pixel).

The limitations severely restricted the number of trajectories seen by the agent: with 7 possible start positions and two directions, there were only 14 trajectories. However, this would serve as a useful 'easy' test of the models that could be quickly evaluated.

4.1.1 Input Data

A frame of input consists of a 7×21 black grid with a white pixel representing the ball and another representing the bat. Where the ball and bat are in the same pixel, only a single white pixel is observed.

This frame is actually observed as a 7×21 binary matrix which was reshaped to make a single 147-dimensional vector. This vector was normalised as discussed in section 7.3.

Since the ball moves at constant velocity, two consecutive frames are sufficient to exactly model its trajectory. So two frames were concatenated for the DQN model; this meant the DQN was not able to select an action in the first time step, so the bat always remains still.

The RDQN was provided with only a single frame of input. It was hope that the bottleneck layer would learn to pass on the useful information to the next frame, so that the network could nevertheless learn to model motion.

4.1.2 State Space

Since the ball moves two pixels horizontally at each time step, and starts in the first column, it can only ever appear in odd-numbered columns. That means the ball can be in any one of $11 \times 7 = 77$ positions.

The bat is restricted to being in one of the 7 pixels in the right-most column, and this is (for a random policy) unrelated to the position of the ball. Therefore, the agent will observe a maximum of $77 \times 7 = 539$ input vectors, and this forms the entire state space.

Note that this is vastly less than the $147^2 = 21609$ possible 147-dimensional binary vectors with either 1 or 2 non-zero elements. That means there exists potential to find good low-dimensional representations of the observed data.

4.1.3 Actions

The action space consists of moving the bat a single pixel up or down, or staying still. These three inputs were represented as choosing from $\{1,2,3\}$, and neither

model was given any prior knowledge of the effect of these actions: they had to learn their effects from observation.

4.2 Experimental Set Up

Since we do not have access to the true state-action values, we cannot evaluate the performance of the neural network in the usual way by reporting some measure of errors. However, since the objective is to train the network to perform well when playing the game, an alternative metric presents itself: the average reward. Since there are only 14 possible trajectories, and the game is deterministic, we can simply run through each trajectory and allow the agent to greedily choose the best action at each time step. At the end of each episode, the agent receives a reward of ± 1. If we average this reward over all 14 games, we get a measure of the success of the agent's learned policy.

To do this, I ran a grid search over a number of different parameters for each model. For the DQN, I used minibatches of $\{25, 50, 100\}$ transitions, learning rates from $\{10^{-2}, 10^{-3}, 10^{-4}\}$, target network updates every $\{25, 50, 100\}$ iterations, with architectures of $\{[1000, 500, 200, 50], [200, 50, 25]\}$, and I tried each combination of these with both rectified linear and tanh units.

I repeated each of these experiments 5 times and averaged the results. Aside from the learning rate, where 10^{-3} was by far the best, few of the parameters had a significant impact. Whether the target network was updated every 25 or every 100 iterations, it seems that an approximately equal number of gradient updates is required to learn an optimal policy.

I also tried learning rates close to 10^{-3}, but they all diminished the performance, and indeed this is the learning rate recommended by Kingma [15].

For the RDQN, I tried minibatches of $\{5, 10, 25\}$ games, target network updates every $\{25, 50, 100\}$ iterations, and architectures of $\{[1000, 500, 200; 200, 50]$, $[200, 50; 50, 25]\}$ (where the semi-colon separates the filter and output networks). Again, I tried both rectified linear and tanh units.

In all experiments, I used an ε-greedy policy with ε annealed linearly from 1.0 to 0.1 over a fixed number of gradient updates. In fact, good results were achieved very quickly, while ε was still at the top end of its range.

Rewards were discounted with $\gamma = 0.95$.

4.3 Results on deterministic version

The performance of the two models are shown in figure 4.2. It is interesting that both models improve the performance fairly monotonically. This is surprising since the loss function is defined in terms of the state-action value functions, and not

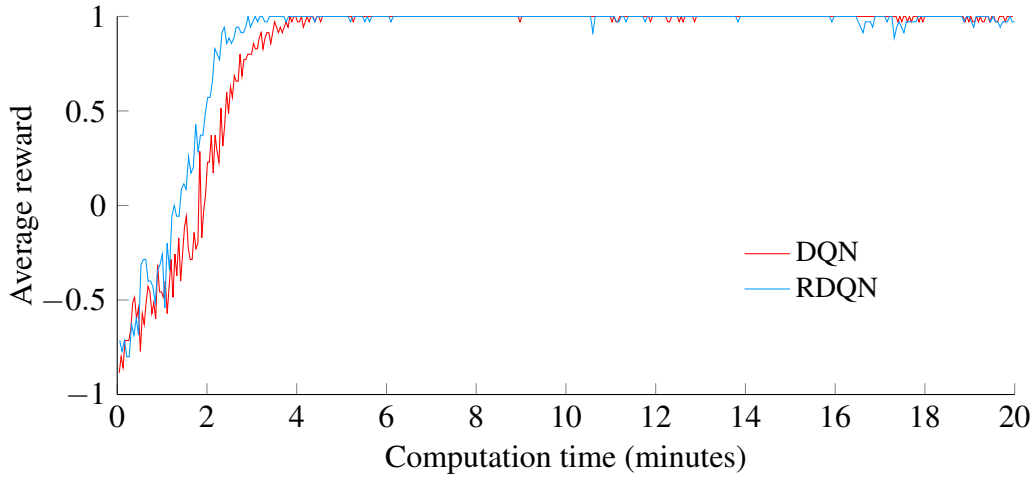


Figure 4.2: Average reward plotted against training time on the deterministic game (averaged over 5 experiments using the parameters that showed the fastest policy improvement). For both models, the optimal learning rate was 10^{-3}, and the rectified linear units were (as expected) faster. For the DQN, the other parameters were a minibatch of 25 transitions and target updates every 50 iterations. For the RDQN, they were a minibatch of 5 games and target updates every 25 iterations. It should be noted that the time taken to get an optimal policy is not sensitive to the choice of parameters, aside from learning rate.

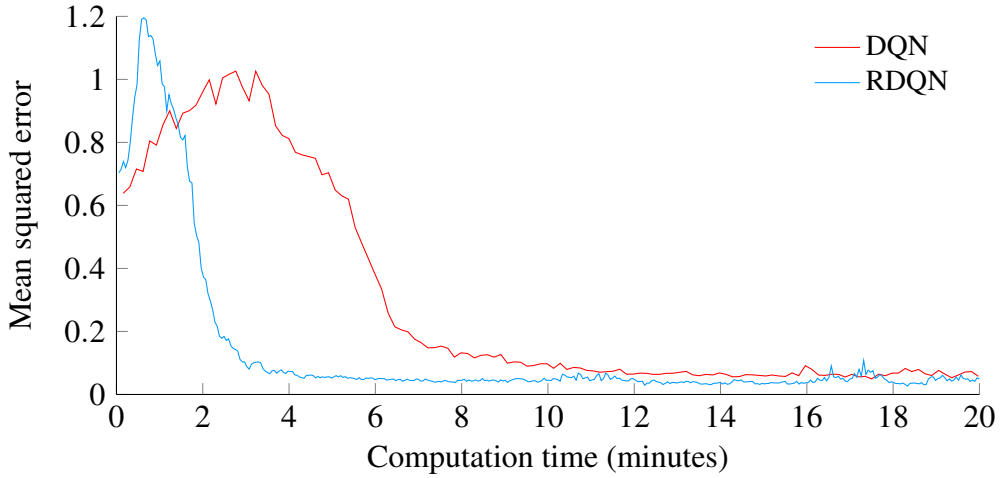


Figure 4.3: Mean squared error computed after each target update, with ground truth approximated by a DQN trained until convergence.

the policy directly. While good estimates of state-action values will lead to a good greedy policy, they are not necessary: all that matters is that the ordering for the 3 available actions in a given state is correct. It is not clear that improving the value function for a particular action will enforce this, but it appears that this is usually the case. The regular small drops in performance however confirm that improving value functions doesn't *always* improve the policy.

The results for the best parameter settings in each case are shown in figure 4.2. Both models learn optimal policies in around 5 minutes of computation on a single core, and although the RDQN was slightly faster, the difference between them is comparable to the difference between different parameter settings. This suggests that a wider parameter search might lead to the DQN overtaking the RDQN.

I was able to get another measure of performance over time by comparing the predicted state-action values to those from a pre-trained network. I left this 'ground truth' network to train until the errors $|r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)| < 10^{-4}$ for all visited state-actions (following a random policy). Treating this as the ground truth allows us to compute the mean squared error over all states visited while following a greedy policy (with the current learned parameters), and this is shown in figure 4.3.

It is interesting that in both cases, the mean squared error initially rises before falling. That is probably due to the bootstrapping leading to some instability in the early stages of training. It is also surprising that the RDQN manages to reduce the mean squared error much more quickly than the DQN, but it shows very little improvement in terms of the rate at which the policy improves. That's likely to be because there are fewer parameters to learn in the RDQN: a latent state layer of size 50 replaces the additional observation of size 147.

4.4 Stochastic Pong

To make the game more challenging, I introduced an element of stochasticity into the game: whenever an action (other than staying still) is chosen, there is a 20% chance of the opposite action being chosen. So on 20% of the time steps in which the agent chooses to go up, the bat actually goes down.

If we assume that the final moves of an optimal policy are equally likely to be up, up or stay, then we would expect that in the two thirds of games which do not end with staying still, 20% of the time the bat will move the wrong way. That's $14 \times \frac{2}{3} \times 0.2 = 1.87$ games that we expect to lose if the policy doesn't account for the stochasticity. In fact, it's slightly higher than that because some games would require two consecutive moves in the same direction in the final two time steps (e.g. the episode ending '$\cdots$⇩⇩'). Again, if we assume a uniform mix of moves,

we might expect one third of penultimate moves to be the same as the final move, so we can add $14 \times \frac{2}{3} \times \frac{1}{3} \times 0.2 = 0.62$ to the previous estimate. This obviously continues, but it becomes increasingly unlikely since we must multiply by a third for each additional time step, and the width of the grid would make it less likely still (it is only possible to do 6 consecutive moves in the same direction).

That means that we expect an average return of approximately $1 \times \frac{14-2.5}{14} + (-1) \times \frac{2.5}{14} = 0.64$ if the game does not account for the stochasticity. An average reward above this suggests that the agent has learnt to deal to some extent with the stochasticity.

4.5 Results of Stochastic Pong

Figure 4.4 shows the results after training both the DQN and RDQN for 925 minutes of computation time, using the best parameter settings selected from the parameters detailed in section 4.2, except that learning rates were selected from $\{10^{-3}, 5 \times 10^{-4}\}$ and target network updates every $\{100, 250, 500\}$ iterations. This was because the stochasticity of the game makes it important to aggressively separate the target and current networks. The results shown are the averages of 3 different experiments, each with a random initialisation.

We observe that the DQN indeed learns to deal with the stochasticity and its average reward approaches 1 (that is, it wins almost every game by learning a strategy to move early in the episode).

The RDQN, however, never consistently exceeds the score of 0.64 (indicated by the horizontal line) expected for an agent that learnt a policy based on a deterministic game. One possible explanation is that the outcome here is sensitive to the initialisation of the network weights, since the results are consistent with two runs having perfect results and one result performing poorly (with mean reward close to 0). However, figure 4.5b shows that the variance between different runs was very low.

We must conclude, then, that the RDQN fails to account for the stochasticity. This is despite trying with a wide variety of parameters. To assess whether the problem was that the RDQN simply can't predict where to put the bat early enough in an episode, I tried adding a small cost for each move that is proportional to the time step and so penalises moves near the end of an episode more than moves at the beginning of an episode. With this modification, the RDQN was able to learn an optimal strategy in the stochastic version very quickly.

For a further discussion of this result, see section 8.1.

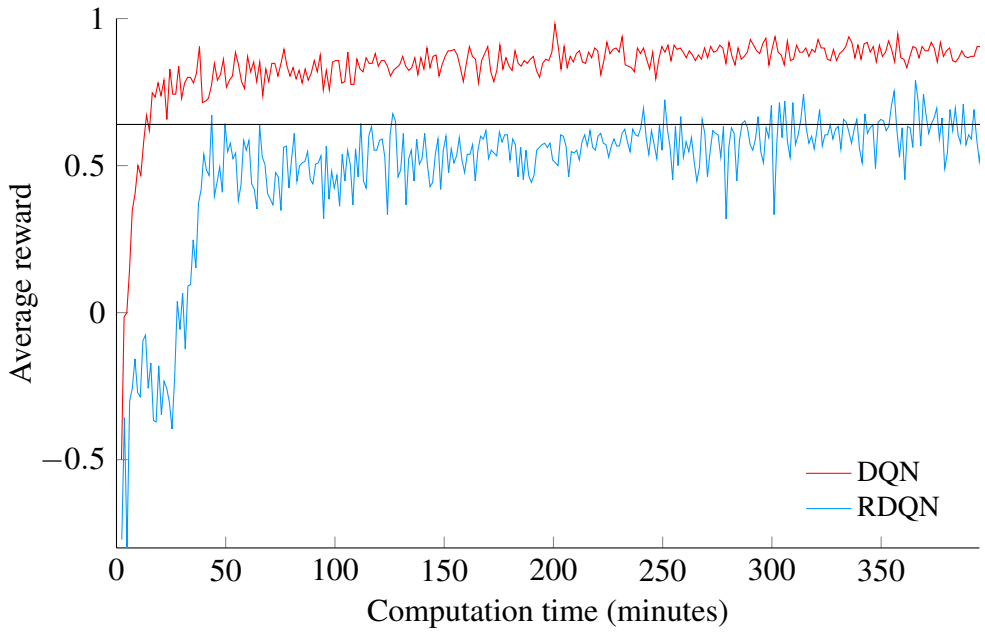
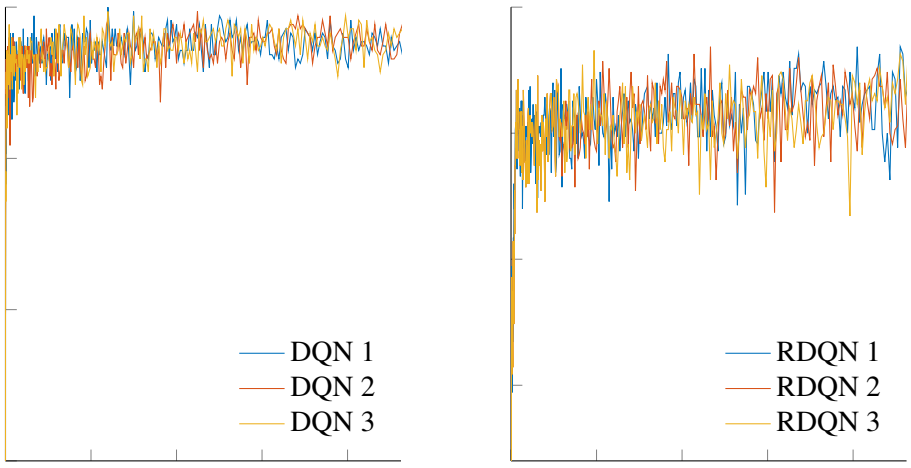


Figure 4.4: Average reward after each iteration when training on stochastic Pong, where the controls have a 20% chance of being inverted. The best learning rate for both models was 10^{-3}. The choice of minibatch size and number of inner loops for the DQN made little difference. Similarly the RDQN showed similar results for each setting of the hyperparameters, but a minibatch of 10 episodes and target updates every 250 iterations did perform slightly better than other settings.



(a) DQN results. **(b)** RDQN results.

Figure 4.5: Each trial for a given set of parameters had very similar results.

Chapter 5

Experiment 2: Pong 2

5.1 Description of Game

Pong2 removes some of the restrictions in Pong to allow for vastly more trajectories and a much larger state space. Rather than only moving at 45° angles, every game randomly selects a horizontal velocity from $\{1,2,3\}$ and a vectical velocity from $\{-3,-2,-1,0,1,2,3\}$. This meant we had to use a 7×25 grid to ensure that all trajectories include the final column, which corresponds to a 175-dimensional binary vector.

There are $7 \times 3 \times 7 = 147$ possible trajectories, and since the ball can be in any position on the grid and the bat can be in any position in the final column, the agent can observe any one of $7 \times 7 \times 25 = 1225$ possible input vectors.

One important result of this version of Pong is that the episodes are of differing durations (since the horizontal velocity which determines the length of the game can have three values). This will test the RDQN since its latent state must be robust to different numbers of updates.

Again, the DQN was provided with two consecutive frames concatenated into a single 350-dimensional input vector, and the RDQN was provided with the original 175-dimensional vectors.

5.2 Experimental Set Up

Again, both models were trained with an ε-greedy policy, annealed from 1.0 to 0.1 over the course of training, and results were averaged over 5 runs. The same grid search was carried out as in section 4.2, except that the learning rate was fixed at 10^{-3} since this proved so much more successful.

5.3 Comparison of Performance

Figure 5.1 shows the average reward against computation time for the best DQN model and the best RDQN model. The DQN quickly achieves good performance,

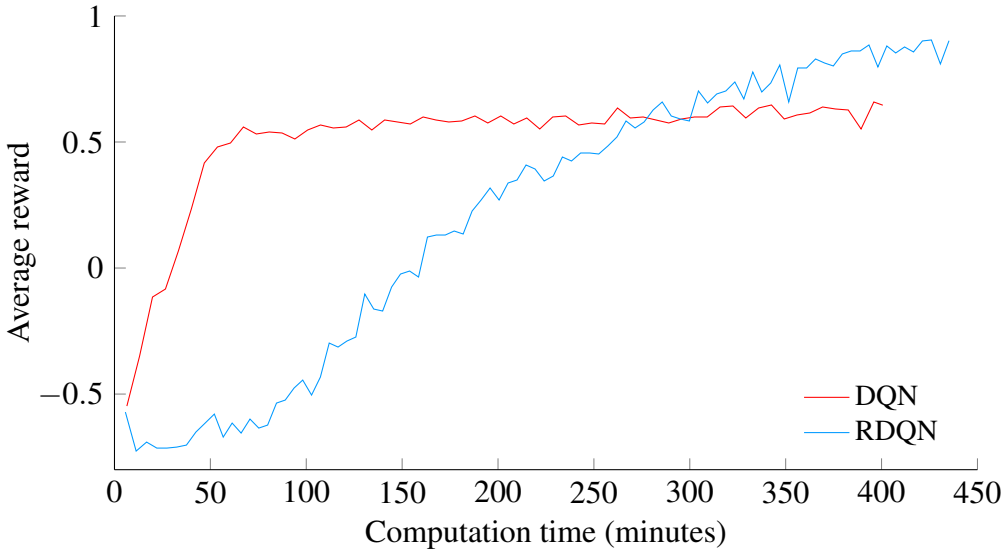


Figure 5.1: Average reward after each iteration when training on Pong 2. The best learning rate for both models was 10^{-3}. The best minibatch for the DQN was 250 transitions and 25 episodes for the RDQN. The best target update frequency was 50 iterations for the DQN and 25 for the RDQN.

winning approximately 80% of its games (which corresponds to a average reward of around 0.6). However, it seems to plateau at this point and fails to improve its performance.

The RDQN on the other hand takes much longer to improve to a good level, only reaching an average reward of 0 after 3 hours of computation, but it eventually overtakes the DQN's performance, with an average reward of around 0.9 after 7 hours of computation. It isn't clear if this improvement would continue to perfect performance: time constraints mean I wasn't able to train the model for longer.

It is not clear what limits the performance of the DQN in this task. Perhaps with just two consecutive frames, it is difficult to accurately model the possible range of motion of the ball, although this seems unlikely: there are only 21 possible velocities.

I repeated the DQN experiment with more nodes in each layer in case the problem was the limited representational ability of the network. However, this led to broadly similar results.

Another possibility is that the DQN cannot make a move in the first time step; however, there are still sufficient time steps to reach any part of the grid, and in the first time step the agent can not know in which direction the ball is travelling, so it cannot make a useful move in any case.

Chapter 6

Experiment 3: Transporter Man

6.1 Description of the Game

Transporter Man takes places on a 5×5 grid, as shown in figure 6.1. The player is randomly initialised somewhere in the columns B, C or D and must reach column A to 'collect a package'. This sets a flag c to have value 1. The player is rewarded when they reach the column E to 'deliver the package' (and c is switched back to 0). There is no reward if the player arrives in column E with $c = 0$. An episode lasts 25 time steps, so there is time to repeat this left-to-right phase 3 times (each of which I refer to as a *trip*).

The agent sees only a 25-dimensional binary vector with a single nonzero entry in the location corresponding to the player's location. Crucially, c is not observed. As such, the state is not Markov: the optimal strategy depends on whether $c = 0$ (move left) or $c = 1$ (move right). We can make the state Markov if we concatenate enough frames. However, until the agent has learnt an optimal strategy, the player may spend arbitrarily many time steps with c in either state, and so in principle

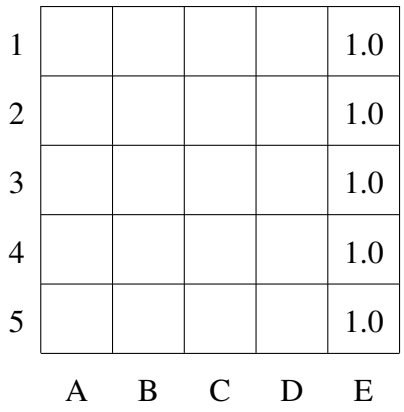


Figure 6.1: The Transporter Man grid. Rewards shown in column E are only awarded when the player is carrying a package (which is collected from column A).

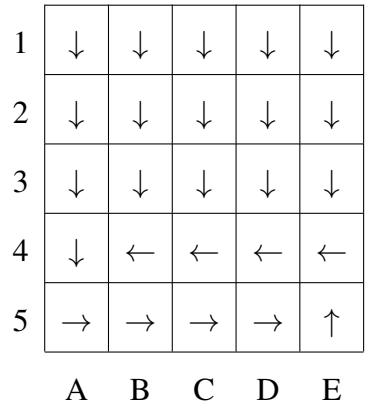


Figure 6.2: A (suboptimal) Transporter Man policy that a DQN can learn.

we'd have to concatenate all 25 frames to make a state guaranteed to be Markov. Since the DQN requires inputs to be the same length, this would mean only one observation per episode, and so the DQN has no opportunity to act: it would be a supervised learning problem, albeit probably one of no interest.

On the other hand, since the RDQN can learn its own low-dimensional representation of the state that changes at each time step, in principle it *can* learn an optimal strategy.

However, the sparsity of rewards may make this difficult: in 1000 episodes of random play, only 145 included any reward, and none included more than one reward.

There does exist a strategy that guarantees good, but suboptimal, rewards that only requires a single observation, by encoding the value of c in the vertical position (since this vertical position is irrelevant to the reward). Now the optimal policy is to move down anywhere in rows 1—3, and then cycle anti-clockwise in rows 4 and 5, as shown in figure 6.2.

6.2 Experimental Set Up

Once again, I provide the DQN with two concatenated frames, and the RDQN with a single observation.

A search of parameters was carried out as described in section 4.2, and again all experiments were averaged over 5 runs. The usual ε-greedy policy was followed throughout and rewards were discounted with $\gamma = 0.95$.

The DQN was trained with an architecture of $[200, 50, 100, 50, 25]$ and the DQRN with $[200, 50; 100, 50, 25]$. That means the RDQN is allowed to learn a state representation with 50 dimensions, which includes a lot of redundancy. In section 6.5 we explore the results with a much smaller latent layer.

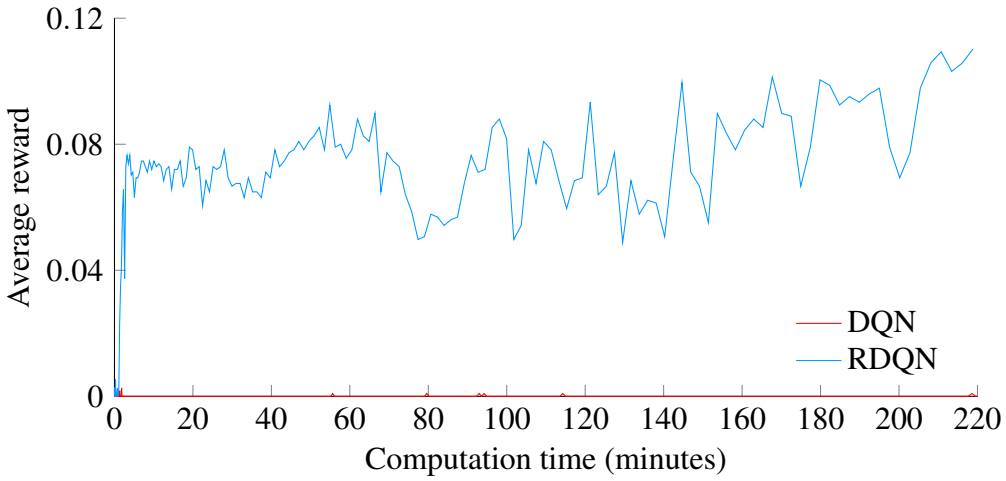


Figure 6.3: Average reward after each iteration when training on Transporter Man. The best settings for the RDQN were a learning rate of 10^{-3}, a minibatch of 25 episodes and target updates every 50 iterations. The DQN achieved similarly poor performance regardless of the hyperparameters.

6.3 Comparison of Performance

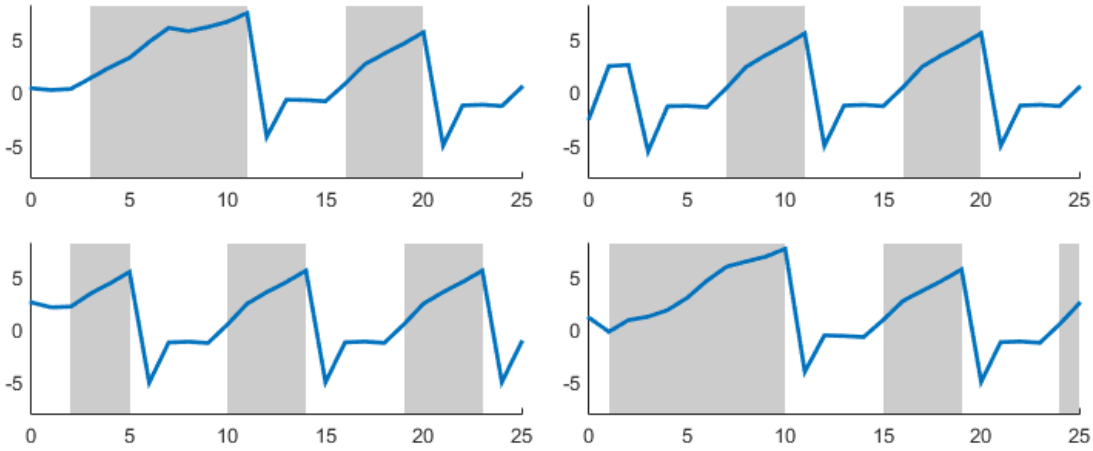
Figure 6.3 shows the average reward after each iteration, and we clearly see that the DQN fails to learn a good policy, with average rewards only rarely rising above zero when following the learned policy. The RDQN, on the other hand, quickly learns a good policy and within an hour of computation time consistently completes two trips in an episode (corresponding to an average reward of $2/25 = 0.08$. The trend of the average reward seems to be continuing upwards towards the theoretical maximum reward of 0.12 (three trips).

6.4 Analysis of Latent State Layer in RDQN

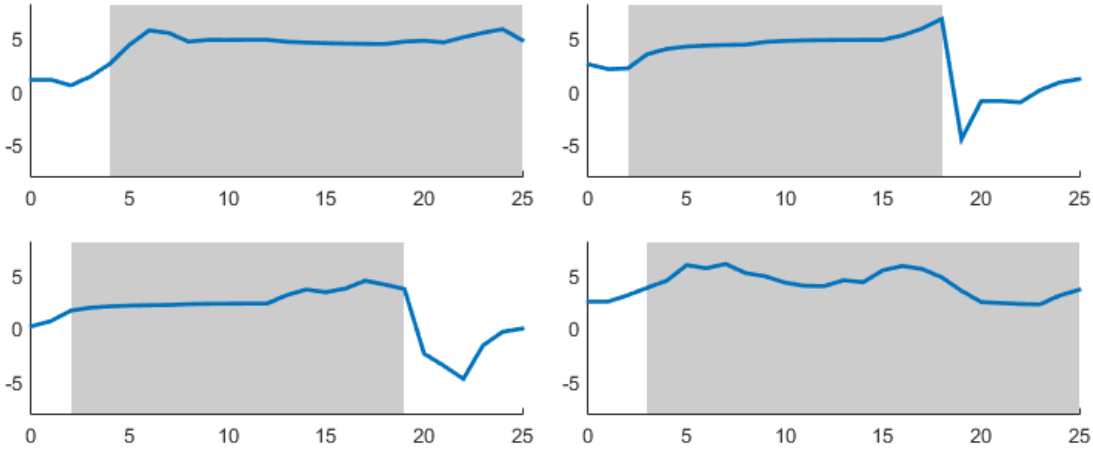
In addition to comparing the performance of the RDQN compared to the DQN, it is interesting to see what representation the RDQN learns in its latent layer. This is the layer that is passed as part of the input to the next time step, and since the RDQN manages to learn a near-optimal policy, it must have learnt a state representation that is Markov.

It is unlikely to be possible to interpret the latent state when it contains 50 nodes as in the main experiment above, so I trained a new network with a latent layer with only 3 nodes. In principle, this is sufficient: all we need in addition to the new input is a flag that tells the network the state of c.

In practice, the network proves to be quite volatile, with performance significantly dropping at several points during training. However, once it began to converge, the third hidden node showed a strong correlation with the state of c. Figure



(a) Episodes generated by following the optimal policy.



(b) Episodes generated by following a random policy.

Figure 6.4: Value of the third node in the latent state layer during some sampled episodes, with time step on the x-axis and node value of the y-axis. The state of c is indicated by the background colour: white when $c = 0$ and gray when $c = 1$.

6.4 shows how the value of this node changes over the course of 8 different episodes, with the background shading indicating the state of c.

In four of the graphs, the agent follows its learned policy, although with only 3 nodes in the hidden layer, this policy never stabilised at a truly optimal policy (although it is possible that a slower learning rate and longer training would have achieved an optimal policy). Here we see that the value of this node drops from approximately $+5$ to approximately -5 in the time step after it delivers the package. It then slowly rises again until it next drops off the package. The top-right graph is particularly interesting, since it appears to begin 'thinking' that the player has the

package. However, upon reaching the right-hand side, it doesn't receive a reward, but the node value still drops to around -5 and then proceeds as per the other episodes.

In the other four graphs, the agent follows a random policy. We see that the value of this node still increases while $c = 0$ and drops after the package is delivered, but we notice that it stays at its maximum value while $c = 1$.

6.5 Variable Rewards

Given the success of the RDQN on the game as described, I introduced an extension: the reward received would be proportional to the y-ordinate when the player reaches the right-hand side. Figure 6.5 shows the rewards received in each row of column E (provided the package has been collected in row A). So, with $c = 1$, arriving in E1 would receive a reward of 1, but coordinates of E5 would only receive a reward of 0.2.

Now the optimal strategy is to use any spare moves to go up. It isn't immediately obvious whether it is worth missing out on a reward by using additional moves to increase the y-ordinate. For example, going from row 5 to row 4 would double the reward received, so two trips at this higher reward would achieve a greater total reward than three trips in row 5. But going from row 2 to row 1 only increases the reward by 25%, and so it is better to stay on row 2 and complete three trips.

To evaluate the performance of the agent's policy, we need to compare the average reward to the best expected average reward. We can compute this explicitly:

Three trips takes 20 time steps ($R^4L^4R^4L^4R^4$). That leaves 5 time steps in which to reach column A and to get up as far up the grid as possible. The player always starts in column B, C or D. From column B, there are always enough time

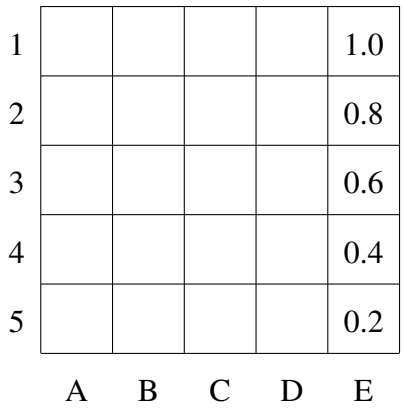


Figure 6.5: The Transporter Man grid, with different rewards upon delivering the package to column E.

steps to get to the first row. From column C, we can get into row 1 as long as we do not begin in row 5. In this case, the best reward is 3 trips in row 2 to give a total reward of $0.8 \times 3 = 2.4$ (versus $1.0 \times 2 = 2.0$ in row 1). From column D, we can reach row 1 from rows 1–3. From row 4, we should again stop in row 2. From row 5, it is better to reach row 1 and do just two trips (total reward 2.0) rather than do three trips in row 3 (total reward $0.6 \times 3 = 1.8$. Therefore, the expected average reward for a perfect strategy is

$$\begin{aligned}\bar{r} &= \frac{1}{25}\left[\frac{1}{3}\times 3 + \frac{1}{3}\left(\frac{4}{5}\times 3.0 + \frac{1}{5}\times 2.4\right) + \frac{1}{3}\left(\frac{3}{5}\times 3.0 + \frac{1}{5}\times 2.4 + \frac{1}{5}\times 2.0\right)\right] \\ &\approx 0.114. \end{aligned} \tag{6.1}$$

On the other hand, using the policy that was optimal in the earlier version of the game would result in an expected average reward of

$$\bar{r} = \frac{1}{25}\left[\frac{1}{5}\times 3\times (1.0+0.8+0.6+0.4+0.2)\right] \approx 0.072. \tag{6.2}$$

Figure 6.6 shows the results of a single run with these variable rewards. The horizontal line shows the expected average reward for an optimal policy, and it is clear that the agent here has begun to reach that level after 60 minutes of computation time.

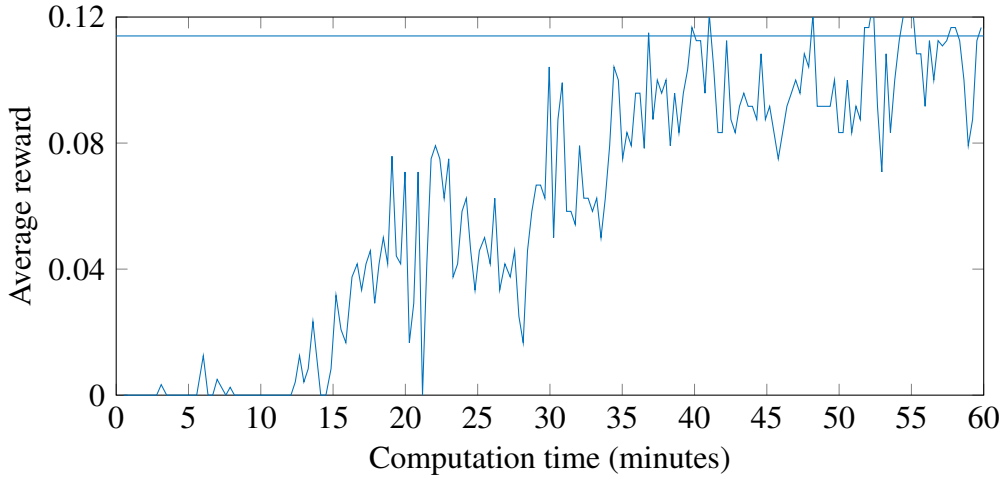


Figure 6.6: Average reward after each iteration when training on Transporter Man. The horizontal line shows the approximate expected return of a theoretically optimal policy.

Chapter 7

Training Tricks for Neural Networks

In this section I summarise some of the details of training neural networks that proved important in achieving good results on these experiments.

7.1 Optimiser

A vanilla gradient descent doesn't optimise a neural network effectively: some kind of momentum is necessary to carry the search over flat regions in parameter space [16, 17]. A common choice is Nesterov's Accelerated Gradient [18], but Adam [15], which includes per-parameter adaptive learning rates, has recently proved to be more effective in some situations and so is the one I chose for this project.

7.2 Targets

I used a different Q target for terminal states, which corresponds to including prior knowledge that there will be no further reward from a terminal state (so the Q-value of a terminal state is zero). This is a slight modification of equation 3.24:

$$Q(s_t, a_t; \theta) \leftarrow Q(s_t, a_t; \theta) + \alpha \left[\text{Qtarget} - Q(s_t, a_t; \theta)\right] \tag{7.1}$$

where

$$\text{Qtarget} = \begin{cases} r_{t+1} & \text{if terminal} \\ r_{t+1} + \max_a Q(s_{t+1}, a; \theta') & \text{otherwise.} \end{cases} \tag{7.2}$$

7.3 Normalising the inputs

It is commonly advised to normalise the input vectors (i.e. to force them to have zero mean and unit variance) [19]. That is so that the inputs into the node functions use the full range of the function, without extending too far into the asymptotic region (in the case of the tanh function). Doing so actually had little effect on the results in these experiments since the observations were close to zero-mean before

normalisation.

7.4 Initialisation of Weights

For a similar reason, initialising the weights such that the activations $a_j = \sum_i w_{ji} z_i$ are also approximately zero mean and unit variance. This can be achieved by drawing the weights from a uniform distribution on $\left[-\sqrt{\frac{3}{W}}, \sqrt{\frac{3}{W}}\right]$ [20].

7.5 Transforming the node functions

If a number x is drawn from a distribution with unit variance, then we expect $\tanh(x)$ to have variance less than 1. That's because $|\tanh(x)| < |x|$ for all x. If we compose a number of tanh functions, each taking the output of the previous function as its input, the size of the outputs of this composition will become very small. To deal with this, we can modify the function to multiply its outputs by the constant required to maintain unit variance [19].

In fact, better results come by choosing constants to enforce unit second moments [20]. I therefore used as transfer functions:

$$\sigma(x) = 1.5930 \tanh(x) \tag{7.3}$$

$$\text{and} \quad \sigma(x) = \sqrt{2}\,\text{relu}(x). \tag{7.4}$$

7.6 Scaling rewards

In the early implementations of the Transporter Man game, the reward for delivering a package was +10. However, since δ_k in the backpropagation algorithm is proportional to $(\hat{y}_k - y_k)$ (see equation 3.13) and multiplies the parameter updates, this can lead to larger than expected steps in parameter space. It is better to use rewards in the $[-1, 1]$ range, and indeed in their implementation, DeepMind [3] clip all rewards at ± 1. Changing the reward for delivering a package to +1 resulted in slightly faster training and fewer exploding errors in the early stages of training.

7.7 Recurrent Connections

Recurrent neural networks are difficult to train due to the disappearing/exploding gradients problem. Since backpropagation through time 'unwraps' the recurrent net into a very deep network, gradients tend to either disappear or explode as you propagate the errors through the node function gradients. Careful tuning of the node functions as described in section 7.5 can help, but it is suggested that other architectures can learn long-term effects more easily, for example the Long Short Term Memory network [21], and these networks have been explored in relation to

reinforcement learning in e.g. [22].

More recently it has been claimed that these results can be replicated by initialising the recurrent connections with the identity matrix [23]. This has the effect of simply replicating the latent layer in the absence of new inputs, and where there are new inputs, it simply adds them to the existing latent state. Of course, from here we allow the network to optimise these parameters, but starting from this simple setting allows longer-term effects to be learnt.

In Transporter Man, we only need a network with 25 layers of recurrent connections, and so this is all of less concern. It seems that a normal initialisation as described in section 7.4 would work, but in all of these experiments I used the identity for recurrent weights.

7.8 Improvements to Training

It would have been better to randomly select hyperparameters in a given range, rather than choosing from limited pre-selected values. That's because each trial tests unique values for each hyperparameter, and in the case where one is more important than the others (such as learning rate in this project), this increases the chance of identifying the optimal value — see [24] for empirical results. In particular, in my final experiments on the extended Transporter Man problem, I found that a learning rate of 10^{-3} was too high: the network would find a good solution, but would not converge at this solution; instead updates would lead it to a much poorer solution. On the other hand, 10^{-4} was too low: the network would take inconveniently long to start to find good solutions. An intermediate learning rate worked better, and I may have found this more quickly had I randomly selected learning rates all along.

Chapter 8

Discussion

8.1 Stochastic games

Earlier in section 2.6 we derived the Monte Carlo update, Equation 2.12, by differentiating the mean squared error between predictions and returns. We then derived the TD update and the Q-Learning update by replacing the target in this expression with the TD or Q-Learning target.

However, we can instead begin by taking the derivative of the Bellman error (where we would replace the inner sum with a max over the actions to make the Q-Learning error),

$$E = \sum_t p(s_t, a_t) \left[r_t + \gamma \sum_{s_{t+1}} \mathcal{P}^{a_t}_{s_t s_{t+1}} \sum_a \pi(a_{t+1} = a | s_{t+1}) Q(s_{t+1}, a) - Q(s_t, a_t) \right]^2 . \tag{8.1}$$

The square bracket evaluates the difference between the expected return from a specific state s_t and the bootstrapped prediction. The inner sum is over all the possible actions from state s_{t+1}, weighted by the probability π of performing that action according to the policy from some particular successor state s_{t+1}, and the outer sum is over each of these successor states, weighted by the probability of reaching that state when performing action a_t in state s_t (which is given by the environment).

Writing $\langle Q(s_{t+1}, a) \rangle_\pi = \sum_a \pi(a_{t+1} = a | s_{t+1}) Q(s_{t+1}, a)$ results in a gradient

$$\frac{\partial E}{\partial \theta} = \sum_{t: s_t = s, a_t = a} p(s_t, a_t) \left[r_{t+1} + \gamma \sum_{s_{t+1}} \mathcal{P}^{a_t}_{s_t s_{t+1}} \langle Q(s_{t+1}, a) \rangle_\pi - Q(s_t, a_t) \right] \times \left[\gamma \sum_{s_{t+1}} \mathcal{P}^{a_t}_{s_t s_{t+1}} \langle \nabla_\theta Q(s_{t+1}, a) \rangle_\pi - \nabla_\theta Q(s_t, a_t) \right] \tag{8.2}$$

since the expectation of the gradient is equal to the gradient of the expectation. Here

the first square bracket is a scalar (the magnitude of the error) and the second is a vector (the direction in which we should update the parameters θ).

We may then wish to use the same trick of making a single sample to estimate the gradient:

$$\theta \leftarrow \theta - \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] \left[\gamma \nabla_\theta Q(s_{t+1}, a_{t+1}) - \nabla_\theta Q(s_t, a_t) \right] \tag{8.3}$$

But there is a problem with this update. The gradient in equation 8.2 has two expectations involving s_{t+1} which are multiplied together, and so to get an unbiased estimate, we would need two independent samples. This is achievable for an environment with a small state space where we can sample each state-action pair multiple times, but in general, we may only see a specific state once (and this is one of the difficulties the functional approximation is designed to address). If we sample from our actual experience, we only have a single sample to use, and so the update 2.12 is biased.

8.1.1 Tabular Q Values

In the case of tabular Q values, this is not a problem. Let us consider the gradient of a single element of the table, $\frac{\partial E}{\partial \theta_{(s_t, a_t)}}$, where there exists a one-to-one mapping between elements of θ and state-action pairs. This gradient term depends only on a linear factor (the first line of equation 8.2) and $\nabla_\theta Q(s_t, a_t)$, since $\nabla_{\theta_{(s_t, a_t)}} Q(s_{t+1}, a_{t+1}) = 0$ for $(s_{t+1}, a_{t+1}) \neq (s_t, a_t)$. As a result, the update is the same as the TD update in equation 2.14.

Note that there is an additional nonzero term in $\frac{\partial E}{\partial \theta}$ corresponding to (s_{t+1}, a_{t+1}), but we can simply choose not to update this: it will not affect the update to the table entry for (s_t, a_t).

In the case where we can go from a state back to the same state, and both times perform the same action, this will not be true. But now the relevant term is $\gamma \mathcal{P}^{a_t}_{s_t s_t} \langle \nabla_\theta Q(s_{t+1} = s_t, a; \theta) \rangle_\pi$ which is guaranteed to be smaller than $\nabla_\theta Q(s_t, a_t; \theta)$ for $\gamma < 1$ since $\mathcal{P}^{a_t}_{s_t s_t} \leq 1$. So in this case, the element of the gradient vector corresponding to (s_t, a_t) will have the same sign whether or not we include the s_{t+1} term. This means that if we include this additional term, the gradient descent algorithm will still converge on the same solution, albeit at a slower rate.

8.1.2 Linear Function Approximation

A linear functional approximation is a representation of the form $Q(s,a) = \theta^\top \phi(s,a)$ where ϕ are M linear functions $\phi_i : \mathcal{S} \times \mathcal{A} \to \mathcal{R}$ that represent the state-action space and θ are the weights of these functions. Under this scenario, provided certain criteria are fulfilled, then this can also be shown to converge to the true

solution, although a discussion of this is beyond the scope of this project, see for example [25].

8.1.3 Nonlinear Function Approximations

In their DQN, DeepMind address the bias by fixing the parameters for the target function for some number of iterations. Explicitly, in equation 8.2 we switch the parameters θ in terms involving s_{t+1} to θ', and so it becomes

$$\frac{\partial E}{\partial \theta} = \sum_{t:s_t=s,a_t=a} p(s_t,a_t)\left[r_{t+1}+\gamma\sum_{s_{t+1}}\mathcal{P}^{a_t}_{s_t s_{t+1}}\langle Q(s_{t+1},a;\theta')\rangle_\pi - Q(s_t,a_t;\theta)\right] \times\left[\gamma\sum_{s_{t+1}}\mathcal{P}^{a_t}_{s_t s_{t+1}}\langle \nabla_\theta Q(s_{t+1},a;\theta')\rangle_\pi - \nabla_\theta Q(s_t,a_t;\theta)\right] \quad (8.4)$$

where θ' are fixed parameters from a previous iteration. As a result, the term $\nabla_\theta Q(s_{t+1},a_{t+1};\theta') = 0$ and so the update simply reduces to a modified Q learning update similar to equation 2.14.

8.1.4 The RDQN

In the RDQN, this solution is not available to us. While the error calculation for each individual transition borrows the same formula as the DQN and so the term involving s_{t+1} uses old parameters θ', each minibatch includes the proceeding transition for which the current state *is* s_{t+1}, and this does use the current parameters θ.

The computation of the total error in a minibatch sums over entire episodes, and so each state (except the terminal state) appears as both the current state and the next state. This is a possible explanation for the failure of the RDQN to deal with stochastic games.

8.2 Deterministic Games

This all suggests that separating the weights on the target network is *only* necessary in stochastic games. Keeping these two networks equal to one another corresponds to updating the target equation 3.24 to include only a single set of parameters:

$$\theta \leftarrow \theta + \alpha\left[r_{t+1}+\gamma\max_a Q(s_{t+1},a;\theta) - Q(s_t,a_t;\theta)\right]\nabla_\theta Q(s_t,a_t). \quad (8.5)$$

On a deterministic game, this should find an optimal solution as well as (or better than) the DQN. To test this, I repeated the original Pong game (Chapter 4) using this update. Figure 8.1 shows that indeed this does work.

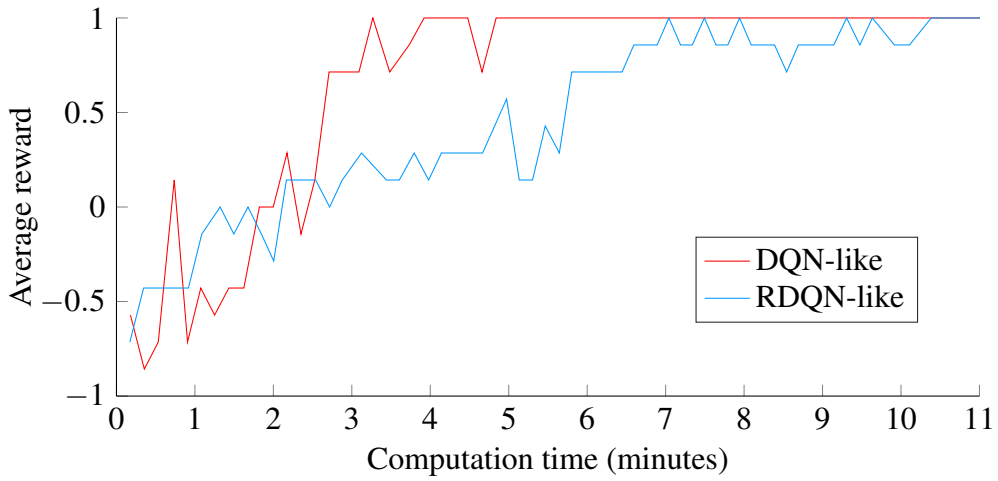


Figure 8.1: Comparison of performance of a DQN-like and RDQN-like game using the update 8.5 after a single run. The hyperparameters and architecture is identical to those identified in section 4.3, with the only difference being a target update frequency of 1 iteration.

8.3 Conclusions

As a result, we can conclude that the RDQN is not suitable for stochastic settings. However, it clearly showed superior performance to the DQN in a setting where the Markov state is not observable, due to its ability to maintain its own state representation. This ability to maintain its own memory (even in only a deterministic setting) is valuable: many real life problems are deterministic, and being able to act in an optimal way, taking into account events that happened arbitrarily far in the past, will be important in many of these problems.

The games attempted here are simple toy problems designed to test the effectiveness of the RDQN without undue computational cost. A natural extension would be to extend this to more complex games.

8.4 Possible Extensions

8.4.1 Increasing the State Space

One obvious extension is to test the RDQN on more complex games such as the Atari games in [2, 3]. However, the observations here are 210×160, and a normal neural network is unlikely to be able to learn from 33600-dimensional vectors. Convolutional layers would be required to provide some prior knowledge about the distribution of pixels on the display itself. Implementing this in Matlab is a significant challenge and beyond the time constraints of this project! These games also incorporate some degree of stochasticity (such as the location where new enemies appear), although it is not clear whether some limited stochasticity can be dealt with

by the RDQN.

8.4.2 Longer Episodes

Another feature of more complex games is that the episodes may last significantly longer. For example, a game that lasts 10 minutes at 25 frames per second will have 15000 time steps.

Therefore using entire episodes in the minibatches would result in very slow updates, especially because the transitions in each episode are highly correlated, so we would need to either sample a large number of them, or have a very low learning rate.

One apparent solution would be to store the state of the latent layer with each transition, and then sample from them. We could apply experience replay to these transitions, appending the latent state to the new observation to create a quasi-Markov state. However, the transitions would be a function of the parameters θ_t at some time point t when they were stored. When we come to update $\theta_{t+\delta}$ for $\delta \geq 1$, we will be basing our update on predictions made using a mixture of θ_t and $\theta_{t+\delta}$. While it is possible that the agent might learn a good approximation of $Q^\star$ regardless (perhaps with an aggressive ε-greedy policy that generates a large number of new transitions after each update to replace the oldest transitions), it seems likely that this will lead to instabilities.

The alternative is to re-compute the latent layer after each gradient update. While we would only need to do a forward pass through the filter part of the network, this is still likely to be prohibitively expensive for all but a very small pool of transitions.

Note that these concerns apply only when computing errors; at test time, or when generating new experiences following an ε-greedy policy, we run full episodes and so the latent state is allowed to build naturally using the current best parameters.

This suggests a potential solution: sample minibatches that consist of m ordered sets of T consecutive transitions. For each ordered set, the latent state could be built up over the first $T/2$ time steps (burn in), and then we start to compute errors using equation 7.1 on the final $T/2$ time steps. This would of course mean that state memory is limited to T time steps, and so events occurring more than T time steps apart cannot be associated to one another. As a result, any such causal relationships will not be learnt. But this would still represent a significant improvement over the DQN, which can only learn relationships that occur within four time steps of each other.

To ensure that the latent state isn't sensitive to the choice of T, we could instead

sample a length t within some bounds. If this allowed for occasional very long sequences, it is possible that the model could still learn long term dependencies provided the reward signal is sufficiently strong (and insufficiently explained by other, closer, events).

Appendix A

Colophon

All the code in this project was written in MATLAB and extends Alexander Botev's code available upon request via `github.com/Botev`. Most of the experiments were run on the UCL Legion servers[1] on single cores. While the bulk of the computation was in performing the error sums on each minibatch, and this can be readily parallelised, there is a significant overhead to setting up parallel pools in MATLAB and this outweighs the time saved in performing the computation.

Code is available from `github.com/ndingwall/RDQN`. The main functions to run experiments are in `runPongExperiment`, `runPong2Experiment` and `runTManExperiment` for the games in Chapters 4 to 6 respectively. These functions call other functions that do the gradient updates and test the performance of the networks. The DQN and RDQN are each implemented as classes, and the gradient computations are carried out by their methods `evaluateSample.m`.

This document was typeset using LaTeX and BibTeX, and composed with TeXmaker.

[1] Information at `wiki.rc.ucl.ac.uk/wiki/Legion`.

Bibliography

[1] Richard S Sutton and Andrew G Barto. Introduction to Reinforcement Learning. *Learning*, 4(1996):1–5, 1998.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv: ...*, pages 1–9, 2013.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei a Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[4] David Silver. Lecture notes from UCL Advanced Topics in Machine Learning course, 2015.

[5] R S Sutton and R S Sutton. Learning to Predict by the Methods of Temporal Di erences. *Machine Learning*, vol:3pp9–44, 1988.

[6] P Dayan. The Convergence of TD(lambda) for General lambda. 362:341–362, 1992.

[7] John N. Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997.

[8] Christopher J C H Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

[9] L Baird. Residual Algorithms: Reinforcement Learning with Function Approximation. *Proceeding of the 12ˆ{th} International Conference on Machine Learning*, pages 30–37, 1995.

[10] C M Bishop. *Neural Networks for Pattern Recognition*, volume 92. 1995.

[11] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7700 LECTU:437–478, 2012.

[12] Thomas M. Breuel. The Effects of Hyperparameters on SGD Training of Neural Networks. 2015.

[13] Volodymyr Mnih and Koray Kavukcuoglu. Methods and apparatus for reinforcement learning, 2015.

[14] Andrej Karpathy. The Unreasonable Effectiveness of Recurrent Neural Networks, 2015.

[15] Diederik P. Kingma and Jimmy Lei Ba. Adam: a Method for Stochastic Optimization. *International Conference on Learning Representations*, pages 1–13, 2015.

[16] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. *Jmlr W&Cp*, 28(2010):1139–1147, 2013.

[17] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. Advances in optimizing recurrent networks. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, pages 8624–8628, 2013.

[18] Y Nesterov. A method of solving a convex programming problem with convergence rate O (1/k2). *Soviet Mathematics Doklady*, 27(2):372–376, 1983.

[19] Yann LeCun, Léon Bottou, Genevieve B. Orr, and Klaus Robert Müller. Efficient backprop. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7700 LECTU:9–48, 2012.

[20] David Barber. Deep Learning : Some Thoughts on Transfer Function Scaling (unpublished). 2015.

[21] Felix Gers. *Long Short-Term Memory in Recurrent Neural Networks*. PhD thesis, Universität Hannover, 2001.

[22] Bram Bakker. Reinforcement learning with long short-term memory. In *Advances in Neural Information Processing Systems (NIPS)*, 2002.

[23] Quoc V Le, Navdeep Jaitly, and Geoffrey Hinton. A Simple Way to Initialize Recurrent Networks of Rectified Linear Units. *ArXiv e-prints*, 2015.

[24] James Bergstra and Yoshua Bengio. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*, 13:281–305, 2012.

[25] F S Melo, S P Meyn, and M I Ribeiro. An analysis of reinforcement learning with function approximation. *Proceedings of the 25th international conference on Machine learning*, pages 664–671, 2008.