Report for 4YP Project:

Gaming AI

Stephen Lilico

April 29, 2016

FINAL HONOUR SCHOOL OF ENG / EEM (delete as appropriate)

DECLARATION OF AUTHORSHIP

You should complete this certificate. It should be bound into your fourth year project report, immediately after your title page. **Three copies** of the report should be submitted to the Chairman of Examiners for your Honour School, c/o Clerk of the Schools, examination Schools, High Street, Oxford.

Name (in capitals):

College (in capitals): **Supervisor:**

Title of Project (in capitals):

Page count
(excluding risk and COSHH assessments): _____

Please tick to confirm the following:

I have read and understood the University's disciplinary regulations concerning conduct in examinations and, in particular, the regulations on plagiarism (*Essential Information for Students. The Proctors' and Assessor's Memorandum*, Section 9.6; also available at www.admin.ox.ac.uk/proctors/info/pam/section9.shtml). ☐

I have read and understood the Education Committee's information and guidance on academic good practice and plagiarism at www.admin.ox.ac.uk/edc/goodpractice. ☐

The project report I am submitting is entirely my own work except where otherwise indicated. ☐

It has not been submitted, either partially or in full, for another Honour School or qualification of this University (except where the Special Regulations for the subject permit this), or for a qualification at any other institution. ☐

I have clearly indicated the presence of all material I have quoted from other sources, including any diagrams, charts, tables or graphs. ☐

I have clearly indicated the presence of all paraphrased material with appropriate references. ☐

I have acknowledged appropriately any assistance I have received in addition to that provided by my supervisor. ☐

I have not copied from the work of any other candidate. ☐

I have not used the services of any agency providing specimen, model or ghostwritten work in the preparation of this thesis/dissertation/extended essay/assignment/project/other submitted work. (See also section 2.4 of Statute XI on University Discipline under which members of the University are prohibited from providing material of this nature for candidates in examinations at this University or elsewhere: ☐

http://www.admin.ox.ac.uk/statutes/352-051a.shtml#_Toc28142348.)

The project report (excluding risk and COSHH assessments) does not exceed 50 pages (including all diagrams, photographs, references and appendices). ☐

I agree to retain an electronic copy of this work until the publication of my final examination result, except where submission in hand-written format is permitted. ☐

I agree to make any such electronic copy available to the examiners should it be necessary to confirm my word count or to check for plagiarism. ☐

Candidate's signature: …….. Date: ……..

Abstract

Recent advances in reinforcement learning have shown it to be possible to use it to train neural networks for complex tasks. This report sets out to see if these techniques can be used to leverage the significant power of neural networks to implicitly discover powerful policies, in particular to the cases of playing a trading card game and finding the optimum value of an unknown function that is expensive to compute.

Contents

1 Acknowledgements

I wish to express my deepest gratitude to Professor Phillip Torr, who has provided me with all the necessary materials to run this project, and also the driving ideas for it all.

I would like to thank Dr Jack Valmadre, Dr Bernardino Romera Paredes and Alban Desmaison for their input and assistance at various stages throughout this project.

2 Introduction

This report details the research the author did in the academic year 2015-16 concerning reinforcement learning and its uses. The initial goal of the project was to develop an AI that would be able to play some form of video game using reinforcement learning and neural networks as function approximators. However, halfway through an potentially more fruitful line of research opened up, and the project was redirected to consider how to develop a reinforcement learning agent that can optimize any given function in a minimum of steps.

The primary results of this research are an exploration of the limits of current technology to parse complex situations and produce meaningful behaviour, and in particular the limitations of using Reinforcement learning to train neural networks for control in such tasks.

2.1 Context

Beyond immediate applications in terms of producing better quality AI for video games themselves, such research is really helpful to a number of different areas. The work using deep learning to play Atari games [6] was used to develop an end to end training system for visual control for a robot attempting a number of difficult tasks [2], and further advances in reinforcement learning would quickly find application in a number of different areas of robotics. The work on function optimisation in particular has several direct uses, as well as the indirect gains in terms of control and comprehension in a sequentially observed continuous environment. For example, such an optimiser could be used to quickly find the optimal hyper parameter settings for a neural network, a task for which currently Bayesian optimisation is used.

2.2 Objectives

The primary aim of the research in this paper was to develop an agent that trained neural networks using reinforcement learning to perform the task in hand, be that playing a card game or producing the minimum of some given black box function. Within this objective are the sub-goals of gaining a comprehensive understanding of the current state of the art within reinforcement learning, and learning the arcane tricks required to be able to train deep neural networks.

As a secondary objective, it would be desirable to gain further technical understanding about what a neural network can and cannot learn, as well as potentially developing an

architecture upon which further research can easily be added.

2.3 Report structure

This report is organised broadly into two sections, exploring the research into game playing agents and function optimisers respectively. Within each section, the specific implementations and associated challenges are detailed, as well as any experimental results gathered. The details of the code written are deferred to the appendices.

3 Background Material

This section details the mathematical framework and previous research on which this project was based.

3.1 Artificial Neural Networks

This project uses artificial neural networks for the approximation of various functions within the agent. Like with most learning systems, they have parameters and hyperparameters. Parameters are values used for the task that are updated by the learning process. Hyperparameters define aspects of how the learning process works. Artificial neural networks can be considered to be general function approximators - they learn a non-linear mapping between their inputs and outputs, the complexity of which is dependant on the hyper parameters and structure of the network.

At their most basic, a neural network consists of layers of "neurons". Each neuron takes a weighted linear sum of their inputs (often the output of all the neurons in the previous layer, plus an optional bias term, but not always), then applies a non-linear "activation function" to that. So the output of the jth neuron in the layer could be written as:

$$O_j = f\big(\sum_i (w_{ji} I_i)\big) \tag{1}$$

where I_i are the input terms, $f(x)$ is the non-linear activation function - for example the sigmoid function, $\frac{1}{1+e^{-x}}$. $w_i j$ are model parameters that are learned. The non-linearity allows multiple consecutive layers to add further expressiveness to the function, and the choice of what non-linearity is used also significantly affects its behaviour. Which non-linearity is chosen is normally decided based on experimental results.

The two main non-linearities used within the experiments in this paper are ReLU and HardTanh. Both are non-differentiable, but do have sub-gradients. ReLU (Rectified Linear Units) are defined as:

$$\text{ReLU}(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} \tag{2}$$

HardTanh are defined as:

$$\text{HardTanh}(x) = \begin{cases} -1 & x < -1 \\ x & |x| < 1 \\ 1 & x > 1 \end{cases} \tag{3}$$

Their respective advantages are that ReLU don't suffer from having a critical region where the gradients are non-zero, whilst HardTanh has the advantage that it can pass both negative and positive values.

They are trained using gradient descent to minimise the loss function of interest,which varies between applications - where a specific output is desired mean squared error is often used. The gradient of the output with respect to the input is calculated, using the backpropagation algorithm, which is essentially the chain rule applied to the consecutive layers. So for a network with layers a, b, c, where O_a is the output of layer a and I_a is the input to layer a:

$$\frac{dL}{dI} = \frac{dL}{dO_b}\frac{dO_b}{dO_a}\frac{dO_a}{dI} \tag{4}$$

So the gradient can be "back-propagated" through the network by only considering the gradient (or sub-gradient for non-differentiable non-linearities) for the error with respect to the inputs of that layer. This then allows the gradient of the error with respect to the parameters to be easily calculated thus:

$$\frac{dL}{d\theta_b} = \frac{dL}{dO_b}\frac{dO_b}{d\theta_b} \tag{5}$$

which is easily done because $\frac{dL}{dO_b}$ is already known and $\frac{dO_b}{d\theta_b}$ is a calculable property of the layer. This gradient with respect to the parameters can then be used to update the parameters using standard gradient descent. This principle is shown in figure 1

There are some issues with this - the key one being that typically the error function will be non-convex, and so it is possible to get stuck in unprofitable local minima. One standard trick that helps to reduce that is momentum, whereby each gradient update step for the

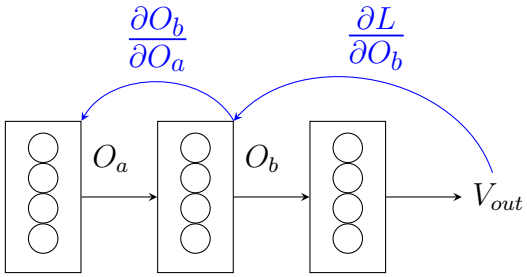


Figure 1: The Backpropagation Algorithm

parameters also includes a weighted multiple of the previous step, encouraging it to keep going in the same direction. So the update equation becomes:

$$\delta_{i,t} = \frac{dL}{d\theta_i} + m\delta_{i,t-1}$$

$$\theta_i = \theta_i + \alpha * \delta_{i,t} \tag{6}$$

The other significant problem is one of over fitting, where the neural network will start learning how to match the noise within the examples to produce an even better fit to the training data, which comes at a significant cost to its ability to generalise. There are a number of ways to combat this, one can keep the number of parameters available to the network low, which means that it doesn't have the ability to fit the much higher order noise. However it is hard to know how large to make the network initially, and training the networks is often computationally expensive, so schemes that iteratively increase the network size take a lot of time. Two better techniques are early stopping and regularisation. In early stopping, a subset of the training data is separated, called the validation data, and after each training epoch the network is tested on the validation set. When the results on the validation set have stopped improving for some number of epochs, the training process is stopped, even if the network is still improving on the training set.

With regularisation, the norm of the parameters in each layer is limited in some way, for example by adding penalty term to the loss function for the total norm of the weights. With neural networks, a similar result can be attained with weight decay or norm limitation. In weight decay after each step each weight is reduced by a small amount, which keeps the values lower. In norm limitation the norm (often the L2 norm) of all the parameters is compared to some limit, which is another hyperparameter. If the norm exceeds the limit, every parameter is scaled down by the same amount so that the norm equals the limit. One additional benefit of this weight decay is if the network initially learns some incorrect behaviour, then it slowly

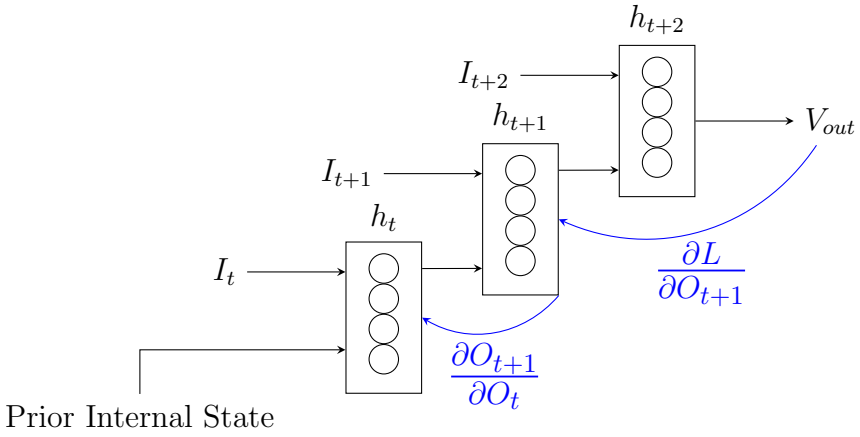


Figure 2: Back-propagation through time

"forgets" what it used to do as those weights are decayed in favour of the ones more recently updated.

3.1.1 Recurrent Neural Networks

A recurrent neural network (RNN) is a particular architecture of an artificial neural network where a layer takes its previous values as an input. This means that there is now a "memory" to the network. With a simple feed forwards network, the outputs are only a ever a function of the current input, whilst with a RNN the output is a function of all previous inputs. This means that RNNs can be used for variable length inputs or outputs. In order to train such a network, the back-propagation algorithm has to be modified to a form called "back-propagation through time". In this the internal states of the network are "rolled out", so that each previous internal state is treated as if it were a separate layer. Then the gradients for each of these rolled out layers are summed together, and this average gradient is used to update the parameters. This idea is shown in figure 2.

More formally, the gradient with which the parameters are updated with can be considered as:

$$\frac{\partial L}{\partial \theta} = \sum_{1<t<T} \frac{\partial L_t}{\partial \theta} \tag{7}$$

$$\frac{\partial L_t}{\partial \theta} = \sum_{1<k<t} \left(\frac{\partial L_t}{\partial x_t} \frac{\partial x_t}{\partial x_k} \frac{\partial^+ x_k}{\partial \theta}\right) \tag{8}$$

$$\frac{\partial x_t}{\partial x_k} = \prod_{t \geq i > k} \frac{\partial x_i}{\partial x_{i-1}} \tag{9}$$

(from [9]) where L_t is the loss from the output of the network at time t, x_t is the internal state at time t and θ are the parameters of the RNN, and $\frac{\partial^+ x_k}{\partial \theta}$ is the immediate gradient of

x_k with respect to θ.

RNNs are very powerful, having found success in a number of different areas, and are capable of handling a much broader range of situations than pure feed-forwards networks, for example multiple inputs to multiple outputs. However they have their own additional issues - they are much more prone to exploding and vanishing gradients. These are where the gradients of elements many steps before the reward either produce exponentially large or exponentially small gradients, dominating any impact of more recent steps or failing to produce any learning at all for such distances. Furthermore, in part due to their power, they tend to produce chaotic responses to variations in the error surface [9], meaning they are much more likely to end up in unhelpful local minima or even just fail to converge.

One trick to help with the exploding gradient is to limit the norm of the gradients. This is done before averaging in a similar way to the hard norm limits. For each layer, the norm of the gradients (again often L2) is calculated, compared to the limit, and if it exceeds the limit then the gradients for that layer are scaled down by the same amount so that the norm of the gradients equals the limit. This means that the closer points will never be dominated, which allows other hyper parameters to be set so as to reduce the vanishing gradient problem.

3.2 Reinforcement Learning

Reinforcement learning (RL) is "a technique where an agent attempts to maximise its reward by repeated interactions with a complex uncertain environment." [12] RL is defined in terms of an agent working within a Markov decision process (MDP), although many applications stretch or break the definition of an MDP. An MDP is a discrete time stochastic control process, where there are some set of states the agent can be in, and in each of those states the agent can take one of a number of actions. Depending what action is chosen, the agent will transition to some state (which may be the same one) with different probabilities depending on what action was chosen, and the agent will receive some reward, $r \in \mathbb{R}$, depending on what transition happened. An important property of an MDP is that it is Markovian, which means what actions can be taken and the transition probabilities are solely a function of the current state, no matter what route was taken to get there. One such process is displayed in figure 3

A reinforcement learning agent follows and assesses some policy, π.The policy determines what action is chosen in each state, so $a_t = \pi(s_t)$. For this policy the agent estimates two functions: the Value function and the Q function, which are shown in figure 4. The value

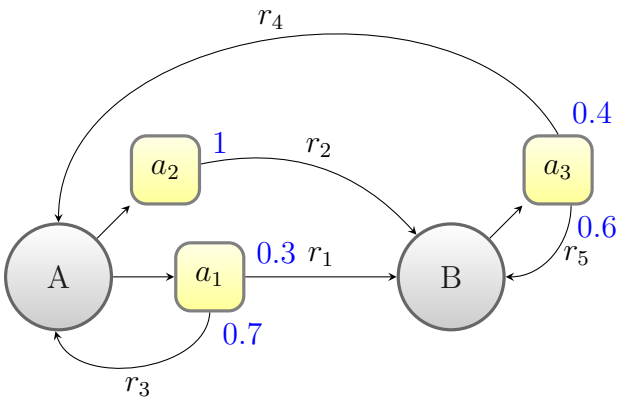


Figure 3: A markov decision process

function is a function of state, which estimates the expected reward that the agent would receive continuing to follow π from that state. The Q function is a function of state and action, which estimates the expected reward the agent would receive if it were to return to following π after taking that action in that state.

All reinforcement learning agents are based upon using the Bellman equations to calculate the values of Q and V. These are defined recursively as:

$$V^{\pi}(s_t) = \mathbb{E}_{s:\pi}(R|s_t) = R(s_t, \pi(s_t)) + \gamma \sum_i P(s_i|s_t, \pi) V^{\pi}(s_i) \tag{10}$$

$$Q^{\pi}(s_t, a) = R(s_t, a) + \gamma \mathbb{E}_{s:\pi}(R|s_{t+1}) = R(s_t, a) + \gamma \sum_i P(s_i|s_t, \pi) \sum_j P(a_j|s_i, \pi) Q^{\pi}(s_i, a_j) \tag{11}$$

Where $R|s$ is the total discounted reward experienced by the agent after state s, $R(s, a)$ is the reward given for taking action a in state s, and $P(s_i|s_t, \pi)$ is the probability of the next state being s_i given the current state is s_t and the policy is π.

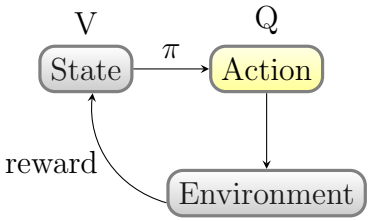


Figure 4: A simplified view of the Reinforcement learning problem

Reinforcement learning can be used for policy evaluation, where V and Q are estimated for the given policy π, though that requires the policy to have been explicitly defined elsewhere. This is done by running the agent and updating the estimates until convergence. In order for

it to produce an estimate for every V and Q it has to visit each state and action an unbounded number of times. However, often what is desired is the discovery of the optimal policy π^*. This can be found by a process called policy iteration. In policy iteration some initial policy is chosen, then evaluated, then improved using the information from the evaluation, then the improved policy is evaluated and the process repeated until convergence. Often it is expedient to not wait for the policy evaluation to converge, but rather perform partial steps of both the evaluation and the improvement. These smaller steps often lead to much faster convergence, provided it still is able to visit every state.

A RL agent can either be following the policy it is evaluating, in which is called an on-policy method, or it can be following a different policy to the one it is evaluating, called an off policy method. On policy methods are simpler, but require the policy to naturally explore the whole state space. Depending on the situation, it is often desirable to have a final policy that doesn't do the exploration on its own, in which case an off policy method would be required.

3.2.1 Temporal Difference Methods

Temporal difference methods are all based on using the bellman step to update the estimates of V(s) and Q(s,a) after every transition.

$$V(s_n) \leftarrow r + \gamma V(s_{n+1}) \tag{12}$$

r is the reward, $\gamma < 1$ is a constant that discounts future rewards, so that for environments with unbounded episode length the value function remains finite. There are several methods to estimate the Q function - the two key ones are SARSA and Q-Learning.

SARSA is an on-policy algorithm which assesses the policy it is following. It follows an update step of:

$$Q(s_n, a_n) \leftarrow r + \gamma Q(s_{n+1}, a_{n+1}) \tag{13}$$

Where a_n is the action chosen by π at step n. As this is an on-policy algorithm, π has to be sufficiently exploratory. When performing policy iteration, the normal procedure is to make π greedy with respect to the calculated Q values. But this won't explore enough on its own, so in addition, π is modified so that it has a small chance ϵ to take a random action on any step. This "epsilon greedy" algorithm is detailed in figure 5. After each Temporal difference update to the Q function, the policy is effectively updated in that region.

In state s, with available actions $\boldsymbol{a}$
with probability ϵ :
    Choose a from $\boldsymbol{a}$ with uniform probability
    Perform action a
else
    for each a in $\boldsymbol{a}$ **do**
        Evaluate Q(s, a)
        if Q(s, a) >Q$_{max}$ **then**
            Q$_{max} \leftarrow$ Q(s, a)
            $a_{max} \leftarrow a$
        end if
    end for
    Perform action a_{max}
end

Figure 5: Epsilon greedy policies

Q learning also uses an epsilon greedy policy to choose its actions, however Q learning is an off policy algorithm, which actually learns about the purely greedy policy. In Q learning the update for Q is:

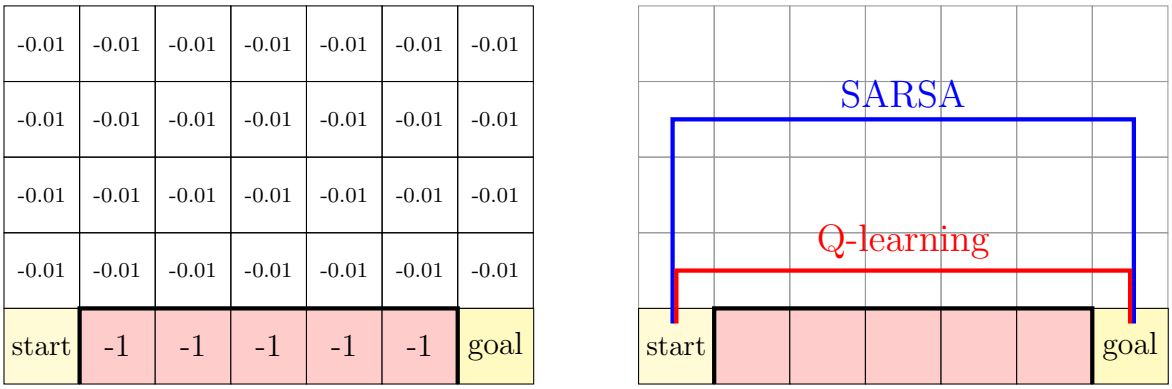
$$Q(s_n, a_n) \leftarrow r + \gamma \max_a \{Q(s_{n+1}, a)\} \tag{14}$$

The max term ensures that, no matter what action is actually chosen in the next state, it learns about the value if it were following the purely greedy policy.

These two algorithms converge to fundamentally different policies, even if we base a purely greedy policy on the final output of SARSA, as a simple example will show. In the simple grid world in figure 6a there is a start, a goal and a cliff. The agent starts at the start, can always choose to move in cardinal directions, gets a reward of 1 for getting to the goal, -1 for stepping off the cliff, both of which end the episode, and a reward of -0.01 otherwise. The two agents converge to the different policies shown in figure 6b. Because the SARSA agent learns on policy, it learns a policy that takes account of the random steps it takes, and so ends up travelling further away from the cliff edge. On the other hand the Q-Learning agent only learns about the states as if it always follows the greedy action, so the Q learning agent travels right up against the cliff edge, as that is the optimal path if it always takes greedy actions.

3.3 Function Approximators

So far all of the above maths has implicitly been assuming that V(s) and Q(s,a) are actually looking up values from a table, such that the function can take any arbitrary value for any



(a) The gridworld

(b) The paths taken by the agents

Figure 6: A demonstration of the difference in pathing for SARSA and Q-learning

of the states and actions. For many applications this is unrealistic - it requires the agent to be able to experience every possible state during training to learn the values for them, which may not be possible for practical reasons and in any case is computationally prohibitive. Far preferable would be to use some function approximation to V(s) and Q(s,a) that could generalise from the experiences it has had to those it hasn't.

There are several key challenges this brings up however: ones of stability, generalisation and expressiveness. If the function is not expressive enough to describe the optimal policy then the agent will converge to a suboptimal policy, if at all. In general there is no guarantee that the function will converge, and often it may well diverge - in particular there are issues where the initial errors in estimates for the Q values of local states can be amplified by the bootstrapping, which is where the current estimates are used in place of the true values. This can be reduced by sticking to linear function approximators, but then there are issues with the expressiveness. Lastly, the feature set chosen for the function approximators needs to be able to generalise sufficiently whilst also being able to tell the difference between good and bad states. Incorrect or and insufficient number of features won't be able to distinguish the relevant differences between states, whilst an excessive quantity of features is likely to lead to over-fitting and so fail to generalise.

One other interesting impact of using a function approximation is that, because by its nature the function approximation is unlikely to yield the true Q value everywhere, it is most desirable for it to be correct in states where the agent is likely to travel, whilst wrong about states that the agent shouldn't end up in. So although it still needs to be able to explore every state to check to see what are better, more of the learning effort should be focussed on more profitable states.

Because of those reasons, for many years neural networks were largely neglected as the

function approximators in reinforcement learning in favour of linear approximators. However, in "Human-level control through deep reinforcement learning" [6] the team at Google Deepmind managed to train a deep network to play Atari games using reinforcement learning. The key changes are that they used experience replay and a target network.

With experience replay they store a set of the previous transitions, then at each learning step, rather than just update with the last transition, they produce a mini-batch of a random selection of previous transitions to learn from, and apply Q-learning for each of them to create the targets to update the network weights with. This helps reduce the chance of the network "forgetting" something that it learned from a previous transition as it learns new things in later states. The random selection helps improve learning stability because it decouples the experiences from each other allowing it to learn from each transition individually.

The target network is a copy of the network that evaluated the Q values but with different weights. In their implementation they copied across their weights to the target network after a large fixed number of steps. The target network was used in the update steps in place of the Q network values for producing the value of the next state, as follows, where θ is the network weights and θ' are the target network weights :

$$L(s,a) = (Q(s,a;\theta) - (r + \gamma Q(s,a;\theta')))^2$$

$$\theta = \theta + \alpha \frac{\partial L}{\partial \theta} \tag{15}$$

This produces a significant gain in stability due to it removing a lot of the instability caused by the bootstrapping, as now a local overestimate of value cannot generate a positive feedback loop in the same manner.

3.3.1 Policy Gradient Methods

In many applications of reinforcement learning, it is necessary to deal with continuous state and action spaces. This is a departure from the strict definition of a Markov decision process, but in many cases sufficient discretisation leads to intractably large state and action spaces anyway. In such cases both Q learning and SARSA face issues due to the need to calculate the maximum of an arbitrary function at each action step. Instead what is used is some policy function $\pi(s;\theta)$ which outputs the continuous action a for any particular step. Then this policy is updated after some number of steps based on the gradient of the expected total rewards with respect to the parameters. This expectation is notated as $J(\theta)$, and is defined

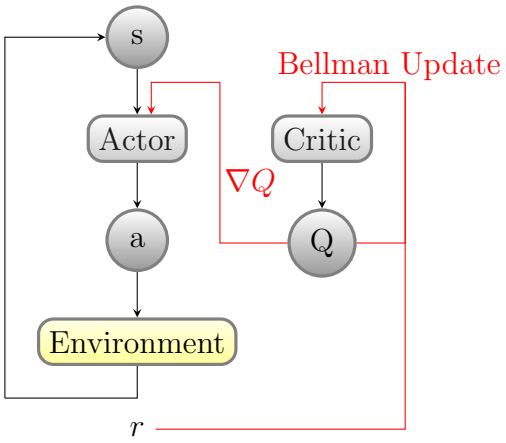


Figure 7: An Actor-Critic System

as:

$$J(\theta) = \mathbb{E}\big[\sum_{t=1}^{T} r_t\big] = \mathbb{E}[R] \tag{16}$$

One family of policy gradient methods are called REINFORCE. In REINFORCE the sample approximation to this gradient is formed as, after running through M episodes:

$$\nabla_\theta J = \frac{1}{M}\sum_{i=1}^{M}\sum_{t=1}^{T} \nabla_\theta \log\pi(s^i_{1:t};\theta)(R^i_t - b_t) \tag{17}$$

This is produced by approximating the action value function as if it were the sample return.

This method is again on-policy, and indeed only works for stochastic policies, as the log trick used to remove the dependence on the gradient of state distribution from the performance gradient depends on the policy having a non-zero probability of taking any action. Indeed, for a long time it was thought that in order to calculate the policy gradient for a deterministic policy a model of the environment is needed to work out the state distribution.

However, in "Deterministic policy gradient algorithms"[10], it was shown that, providing some basic properties of the function are true, the gradient of a deterministic policy $\mu(s)$ is:

$$\nabla_\theta J = \mathbb{E}\big[\nabla_\theta \mu(s;\theta^\mu)\nabla Q^\mu(s,a)|_{a=\mu(s)}\big] \tag{18}$$

This can be implemented using an Actor-Critic method, shown in figure 7, whereby there are separate actor and critic networks, the actor implementing μ and the critic Q^μ. The actor's weights are updated according to the above gradient, whilst the critic can use SARSA or Q learning updates as in the discrete case, but taking action as an input.

In "Continuous control with deep reinforcement learning" [3] the above was combined

with the insights from the Atari paper [6] to produce an actor critic system that used the deterministic policy gradient to train deep neural networks to produce the control for various continuous tasks. The main additional innovation was that the target network parameters are slowly updated towards the current parameters at each step, rather than copying across after some number of steps, to better keep the systems disjoint.

3.4 Related Work

In "Latent Predictor Networks for Code Generation"[4] Google Deepmind look at the parsing and analysis of Magic: the Gathering cards using deep networks, which is a crucial step in developing a competent AI to play the game as a whole.

In "Mastering the game of Go with deep neural networks and tree search"[11] the team at Google Deepmind produced an agent trained by reinforcement learning that beat the world champion at the board game Go. There are many interesting developments on the standard RL trained model, such as retaining a smaller network for their monte-carlo tree searches, but the most interesting technique for this paper was their method of initially training the agent. They first taught the agent to predict expert moves from a large series of board states in a supervised manner. Having trained this agent, the same networks were then used to initialise a reinforcement learning agent that they then played against itself for an extended period of time to produce the value network with which they made the initial decisions with the AlphaGo agent.

4 Initial Implementations

The initial aim of this project was to produce an agent that could learn to play some video game using reinforcement learning, based on the results from the Google Deepmind Atari paper [6] This section details the work done and results from that work.

4.1 Maze solving agents

To provide understanding about reinforcement learning and develop familiarity with training neural networks on such tasks, a trivial Maze world was created. In the maze world the agent always has the same four actions - move up, left, right or down. In the world are walls, which if the agent attempts to enter it instead remains where it is, pits, which terminate the

episode and give a negative reward, and one goal, which terminates the episode and gives a positive reward. In all cases the agent followed an epsilon greedy policy.

For comparison and understanding, simple tabular agents were created for various tabular paradigms as well, where the agent stores a separate value for each location in the maze. With these, it was found that Monte Carlo methods (whereby updates are only done upon episode termination and are done using the exact reward) are unsuitable in such an environment due to the fact that there are many non-terminal policies. Even when the episode is forcefully terminated, these tend to cause the agent to excessively penalise various areas, harming the learning. 1 step (which perform the bellman update after each step using the current estimate for the next value, a process known as bootstrapping) and TD lambda methods (which use a weighted aggregate of the reward over all the future steps, typically implemented using eligibility traces) both found optimal policies in the maze very efficiently whilst using a tabular representation of Q and V.

The function approximation used was deep feed forward neural network given as input a matrix whose value was zero at all points except the one corresponding to the current location. Experience replay and assessment networks were also implemented. However, the problem proved to be more complex than anticipated. It could be observed that the agent was indeed learning something, however the learning was unstable, and suffered from a plethora of local minima. Although there are many more advanced techniques available to further improve its behaviour, it was decided to move on from this issue as many of the techniques involved would be tangential to the issues in training to play a card game, and time was limited.

4.2 Magic: the Gathering

The particular game type that was chosen was a collectible card game called Magic: the Gathering (MtG). This type of game provide a number of interesting and difficult challenges for AI: uncertain information, stochastic results, variable action spaces, along with additional opportunity for further depth should deck building also be considered. They are also turn based and don't require a physics engine, so they can run through many iterations of play quickly. MtG was chosen in particular as it represents both a significant breadth of possibilities and different interactions without excessive card complexity or specificity and it has a more approachable learning curve than most for human players. Hearthstone was considered, but a suitable emulator was hard to find due to the copyright issues.

4.2.1 Initial Setup

A suitable open source emulation environment for playing MtG was identified, and modifications were made to it to allow the learned AI to be used in it and trained against the extant rules based AI. Neural Networks libraries for Java that use the GPU were installed and unit tested. An overview of AI techniques and the particulars of standard reinforcement learning algorithms were read.

One significant factor that affects learning is the size of the state and action space, and MtG does also include many cards with unique and complex interactions, so some additional restrictions were made on the type of cards that would be used within MtG's 15,000 card pool to reduce the initial complexity. More specifically, it was decided that it would just learn to play with basic lands (the games resource type), what are termed "French vanilla" creatures (the fundamental unit of game play), that is creatures with no extra rules text beyond one of a standard set of keywords, and a carefully chosen set of unconditional removal spells so that it has a decent chance of learning something that would generalize. Other options were also considered, such as limiting the card pool to one of the standardized collections of cards used for competitive play (for example Standard, which uses cards from the past 3 blocks released) and allowing it to learn the specifics for each card there. However, it was unclear how to then shape its generalization well, and the naı́eve implementation of that would have a state size of $O(n^n)$.

The first relevant challenge that was faced on this was how to define the state space well. The agent had access to the internal state of the game, which would be necessary for it to make any kind of sensible decision, and is roughly equivalent to it learning about the state of the game from the screen, but without the enormous overheads in training time. However, the conceptual representation of the state space is not immediately amenable to use with neural networks - there is an unbounded number of possible entities that could exist, each of which could have their own unique properties, which are represented in the engine as a string of text. Also, the ordering is irrelevant - the only spatial information that matters is which player they belong to. Furthermore, the situation depends heavily on what specific cards are in the players hands, and what cards they have left that they can draw. The content of the opponent's hands are unknown to the agent, so only the quantity is relevant for defining the state space. Additional complexity occurs in the fact that players can play cards in response to the opponents cards that will resolve first, meaning that the state also has to consider what cards both players have played that have not yet resolved and what, of the unbounded

set of cards already in play, if any, they are targeting.

In order to simplify this, as well as helping the learning to generalize well, it was decided to use a state value based RL system rather than Q values, that is learn how "good" any particular state is, instead of how good any particular action is in any particular state. Then, the state could be further simplified by only considering the creatures on the board and a set of relevant hand picked features from the total game state (life total, available mana, cards in hand, phase). This is still an unbounded set, but due to the restriction of the cards to "French vanilla", the feature set of each creature is a fixed number of indicator variables and three natural numbers. These features could then either be passed through an evaluation network then pooled, or fed into a recurrent neural network to produce an output that a neural network can learn with.

In order for this to be used for control, a model of each state transition from an action has to be used, or some form of actor-critic method created. Fortunately, already within the game engine was an option for the AI to model the results of its actions and choose according to a heuristic score on the resulting states. So this was simply commandeered, with the heuristic score replaced with the value output of the learned network.

All of this produces an architecture as shown in figure 8 training with the algorithm in 9

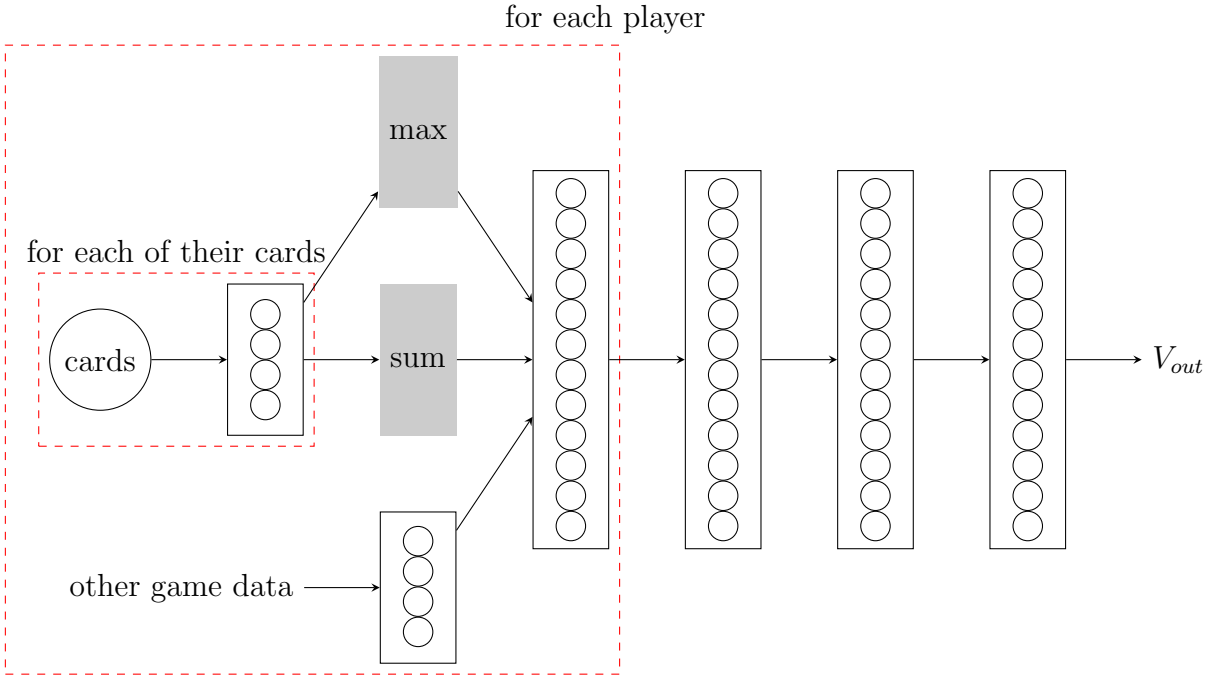


Figure 8: Architecture for the Magic: the Gathering playing agent

$V(s;\theta) \mapsto \mathbb{R}$
repeat
  Pick some initial state s_i
  repeat
    produce a list of actions **a** for s_i
    for a in **a do**
      simulate transition $s' \leftarrow T(s_i, a)$
      if $V(s';\theta) > r_{max}$ **then**
        $a_{max} \leftarrow a$
        $r_{max} \leftarrow V(s';\theta)$
      end if
    end for
    with $P(\epsilon)$ take action a_{max}
    else take action chosen uniformly from **a**
    wait for in-game stack to complete
    observe new state s_{i+1}
    if s_{i+1} is terminal **then**
      $r \leftarrow \begin{cases} 1 & win \\ -1 & loss \end{cases}$
    else
      $r \leftarrow 0$
    end if
    store transition $\{s_i, s_{i+1}, r\}$ in *Replay*
    select random batch of transitions **B** from *Replay*
    for s_b, s_{b+1}, r_b in **B do**
      $y_b = r_b + \gamma V(s_{b+1};\theta')$
      perform gradient descent step on $(y_b - V(s_b;\theta)^2$ with respect to θ
    end for
  until s_i is terminal
  every c steps $\theta' \leftarrow \theta$
until Max epochs

Figure 9: Value iteration algorithm for MtG agent

4.2.2 Results from Initial Setup

A number of practical issues plagued the initial run-throughs of this agent. There turns out to be a memory leak within the emulator used for training the AI, so batches of experiences of sufficient size couldn't be gathered to train the agent with. Furthermore, for the initial behaviour it evaluated every state as being equal, for which the default behaviour was to do nothing, which is probably sensible but hampers learning. So it was adjusted to instead pick one of the highest valued actions at random.

With all of these things bar the memory leak having been dealt with, it still wasn't performing very well - it wasn't at all clear that the agent learned anything useful, as its

win rate never improved, and when tested against a human player its actions seemed entirely random. This will partially be due to the limited experience length that it could be trained with but it can probably also be ascribed to its inability to clearly discern the state space, as the max/sum pooling inherently carries a lot of data loss, and it might not be able to tell the difference between importantly different states.

The way to fix this issue would be to replace the pooling with recurrent neural networks, which are able to take sequences of any given length and turn them into a set of features. However, the state space would continue to be very complicated. Another possible improvement to the learning would be to treat the opponent's actions as observations of off policy transitions, which could allow it to explore useful areas of the state space much more quickly, particularly if it is losing most of its early games. Nevertheless, most of the design effort would have to go to very situation specific details to make sure that the state space was properly represented.

Further potential improvement could be found that would generalize the agent to be able to play any card by using and improving upon the work in "Latent Predictor Networks for Code Generation"[4]. The core idea would be to use their method to parse the card text to some set of features that would then be used in place of the hand crafted features. Because the emulator stores additional effects given to cards on the card text for that specific card, that would allow it to parse a broad range of additional effects, and possibly help it better evaluate the quality of the cards it has in its hand, allowing it to make better decisions about when to allocate the resources it has.

5 Function Optimization

Given the vast state complexity present within MtG, an alternative avenue of research with potentially more useful applications was suggested. The task was to train some agent to be able to, given some black box function $f(\boldsymbol{x})$, find $\underset{\boldsymbol{x}_i}{\operatorname{argmin}}(f(\boldsymbol{x}_i))$. Current methods that are used for such situations either require a very large number of iterations (pattern searching, simplex method) or some form of prior for the expected shape of the function (Bayesian optimisation). So, if an agent could be trained to compute the minima within a small number of steps with no explicit prior, that could be useful in a number of cases.

The particular gains of such an agent would probably be most apparent in situations where there is strong, but unknown, similarity between the functions. This is because the

agent, if it is to outperform the all purpose methods, is likely to learn an implicit prior over the functions it is trained on. So if it is trained exclusively on functions of the type that it will be used on, then theoretically it can produce better results for such systems without need to define an explicit prior.

Another interesting result from this experiment would be to see what sort of behaviour the agent learns - how does it balance exploration versus exploitation? Does it attempt to perform newton steps or similar to find the lowest point? This could demonstrate better what types of computation is preferred by neural networks of this type. By getting the agent to occasionally save the output to disk, the set of points explored and the values they returned can be analysed, and the behaviour of the agent better understood as well.

This could be defined as a Markov decision process, where the action is either to trial some $\boldsymbol{x}_i inf(\boldsymbol{x}_i)$ or stop, the state is set of previous observations of $f(\boldsymbol{x}_i)$ and the reward is:

$$r = \begin{cases} steppenalty & \text{non-terminal} \\ -Loss(f(\boldsymbol{x}), f(\boldsymbol{x}_{min})) & \text{terminal} \end{cases}$$

$Loss(a, b)$ is some function that is at a minimum when $a = b, \forall a \geq b$ and *steppenalty* is some non-positive constant that encourages the agent to reach a minimum in the smallest number of steps. In order to define the problem in such a way that it can learn reasonably and fair comparisons could be done it was decided that it was known (or constrained to be) that the minima would lie within some known finite subspace of $\mathbb{R}^n$. In practice this means that the search space and minima were constrained by $x_i \leq x_{max}$ where x_{max} is some known constant.

The reward scheme that simply rewards it for the difference between the final return value and the optimum was chosen not only because of its simplicity, but also because of its independence from the x coordinate checked. Under schemes where the x coordinate needs to be close to the minimum x coordinate, it gets penalised for following optimal behaviour into an unfortunate local minimum. Take the example of a function with two minima, the global one,M_g at x_{min} and a local one M_l at x_{local}, where x_{min} and x_{local} are far apart. Further suppose that $f(x_{min_l}) = f(x_{min_g}) + \epsilon$ where ϵ is small. Suppose the agent explored nearby to both of these minima, and happened to observe a point $\hat{x}_l$ that is closer enough to the local minima that $f(\hat{x}_l) < f(\hat{x}_g)$, where $\hat{x}_g$ is the closest point observed to the global minima. This means the agent would return $\hat{x}_l$ as the minima. Under a reward scheme based on x that would be heavily penalised, due to the large distance in x from the global, despite being

entirely logical. So a reward scheme based purely on $f(x)$ is desirable.

For the experimentation, a series of polynomial functions were defined so that their parameters could be passed as an input, allowing existing neural network training architectures to be used. Each polynomial was defined by randomly choosing a set of roots from within the search space, then producing a series of coefficients by multiplying out $\int \prod_i (x - r_i)dx$, where r_i is the ith root. Where $\boldsymbol{x}$ has multiple dimensions, in each dimension a separate polynomial is defined this way, so that $f(\boldsymbol{x}) = \sum_i \text{Poly}_i(x_i)$ where $\text{Poly}_i(x)$ is the polynomial function for the ith dimension. Then $f(\boldsymbol{x}$ is evaluated at every combination of roots, and the one with the lowest value is the global minima. The full algorithm is detailed in figure 10.

Given the variance of these polynomials, and in particular how much the reward changes with higher orders or dimensions, it was necessary to define a more even comparison between the architectures. One useful statistic was the error rate, defined as the proportion of final values that lay more that 5% of the average absolute value of the minima away from the global minimum. This indicates how many were "close enough" to the target. To further normalize things, two baseline agents were created to give a scale for the rewards to be put on. For simplicity of comparison, the number of steps the agent could take was fixed, and *steppenalty* was set to 0. The baseline agents were a brute force agent that simply divided the search space into equal blocks and looked across all of these, ignoring the values it received. The reward this equal search agent received was defined as 0 relative reward. The other agent uses pattern search, where it checks a grid around the current best location, moves to the new best if there is one, or reduces the grid size if there isn't. The reward this pattern search agent achieved was set as 1 relative reward.

It was decided that, rather than get the agent to learn how to make the comparisons internally (or potentially learn to interpolate between observed values) that the minimum value observed overall would be worked out externally and fed to the neural network as an additional variable. The idea behind this decision is that this frees up the learning in the network to be devoted to finding the best behaviour for the system, rather than having to additionally use up neurons storing this value and performing the comparison in a potentially harmful manner. This does remove the opportunity for the agent to learn how to guess what the true minimum would be based on its observations, but it was decided that such a behaviour would be extremely prone to over fitting and the gains from it would not be worth the cost in terms of the additional learning overheads. Furthermore such an output would simply learn to return lower numbers under the current reward scheme, which means that

```
Randomly select N_dim × N_roots values within range [−x_max, x_max] as the roots
function expandTerms(inds, maxind, numloops, d)
    if numloops = 0 then
        val = 1
        for each ind stored in inds do
            val ← val ∗ ind
        end for
        return val
    else
        val = 0
        for  i = maxind, numloops, -1 do
            inds[numloops] = -roots[i][d]                ▷ roots correspond to (x - a) terms
            val ← val+ expandTerms(inds, i-1,numloops -1, d)
        end for
        return val
    end if
end function
for  j = 1, ndim do
    for  i = 1, nroots do
        params[j][i] = loopick({}, nroots, i, j)
    end for
end for
find min by checking all roots
params used to make function of x thus:
function F(coords, params)
    out = 0
    for i = 1, dim do
        res = 1/(npar + 1)
        x = coords[i]
        for j = 1, npar do
            res = res * x + (params[i][j] /(npar +1 -j))
        end for
        out = out + (res*x)
    end for
     return out
end function
```

Figure 10: algorithm to generate the functions

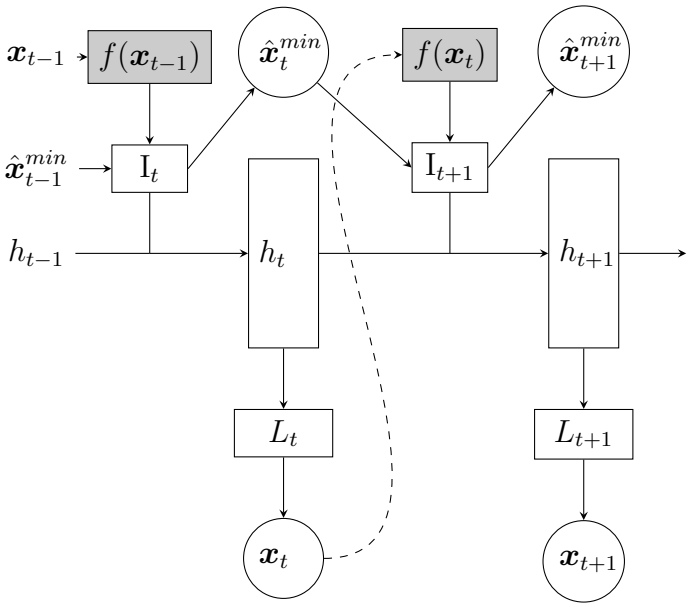


Figure 11: Architecture for recurrent function optimisation

the reward would also need to consider the x value of the optimum, which had already been discarded.

5.1 Recurrent Function Optimisation

The first design that was attempted was based on the work in Recurrent Models of Visual Attention[7]. It used the overall structure shown in figure 11 The idea is that the internal state of the recurrent neural network would be able to describe the state so that the feed-forwards network can decide what location to look at next. The network would be trained directly with REINFORCE, so only the rewards are needed, not any critic or similar structure. The final output is the minimum value observed, which is tracked at each function evaluation step, and also passed to the RNN to help describe the state space better, as then it doesn't have to learn to do that as well. The recurrence for the architecture that was designed can be seen in fig 11. On top of that structure, the average reward b was also learned as a parameter of the network so that the variance reduction in REINFORCE could be used. One crucial difference is that, unlike with the visual attention paper [7], there is no classification, so no classification loss to train the RNN with, so it is only being trained by the REINFORCE module. The algorithm used to train it is detailed in figure 12s

Initially it was very unstable, only wanting to search values at the boundaries of the search space. It seemed to be the case that the internal parameters were exploding to massive values and saturating on every pass. So two normalisation hyper parameters were

$S(\boldsymbol{O}, \boldsymbol{s}_{i-1}; \theta) \mapsto \boldsymbol{s}_i$
$Q(\boldsymbol{s}; \theta) \mapsto \boldsymbol{x}$
$G(\mu, \sigma)$ ▷ Gaussian Noise
choose some initial b ▷ Baseline reward
repeat
  Pick some initial $\boldsymbol{x}_i$
  repeat
    observe $f(\boldsymbol{x}_i)$
    if then $f(\boldsymbol{x}_i) < f(\hat{\boldsymbol{x}}_{min})$
      $\hat{\boldsymbol{x}}_{min} \leftarrow \boldsymbol{x}_i$
    end if
    $\boldsymbol{O}_i = \{f(\boldsymbol{x}_i), \boldsymbol{x}_i, f(\hat{\boldsymbol{x}}_{min}), \hat{\boldsymbol{x}}_{min}\}$
    $s_{i+1} = S(\boldsymbol{O}_i, \boldsymbol{s}_i; \theta)$
    $\boldsymbol{x}_{i+1} = G(Q(\boldsymbol{s}_{i+1}; \theta), \sigma)$
  until Max steps
  $R = f(\hat{\boldsymbol{x}}_{min}) - f(\boldsymbol{x}_m in)$
  $b \leftarrow b + \alpha(R - b)$ ▷ MSE gradient step for $b = \mathbb{E}[R]$
  $\delta = (R - b)\alpha \nabla_\theta \log(Q(\boldsymbol{s}_{i+1})$
  Update θ in the direction of δ using backpropagation through time
until Max epochs

Figure 12: Algorithm for running and training the Recurrent Function Optimizer

used - the cutoffNorm and the maxOutNorm. The cutoff norm is the maximum L2 norm of all of the gradients of the parameters for all the layers. If the norm exceeds that value, all the gradients are scaled down by the same amount so that the L2 norm for the gradients equals the cutoffNorm. This prevents exploding gradients within the recurrent elements of the network. The maxOutNorm sets the maximum L2 norm of any one layer of the network. Then, like with the cutoffNorm, if the L2 norm for the layer exceeds the maxOutNorm then all of the parameters of that layer are scaled down by the same amount until their L2 norm is that value. This, like weight decay, allows the network to "forget" unhelpful learning and also keeps the outputs bounded. Once both of these were applied the agent began to use much more of the space.

Another important trick that significantly improved performance was normalising the inputs to the network. Particularly in the presence of the above parameters, it is necessary to have every input value to the network to be of the same order of magnitude. However, the range of the output of the function isn't known exactly, and its relative magnitude to itself is very important for working out the minimum. The idea that was hit upon to standardise the outputs was to use an approximate upper bound on the output of the function and divide it by that. Because of how the function was defined, the highest order term of the polynomial always has a coefficient of 1 (as any other function could be scaled to be like that anyway).

Furthermore, given that the roots are within known bounds, the agent will never ask for x values above those bounds. So if the output of the function approximator is divided by x_{max}^{p+1}, where p is the number of parameters used to define the function, then the vast majority of the outputs should lie within the range $[0-1]$. Given that actually the agent spends a lot more time within tighter bounds than those, and the other inputs were in the range $[0-x_{max}]$ the output was actually divided by x^p. This did result in immediate improvement in performance.

Two different forms for the loss function were tried - $Loss(f(\boldsymbol{x}), f(\boldsymbol{x}_{min})) = log(f(\boldsymbol{x}) - f(\boldsymbol{x}_{min}+1))$ and $Loss(f(\boldsymbol{x}), f(\boldsymbol{x}_{min})) = f(\boldsymbol{x}) - f(\boldsymbol{x}_{min})$. The log form was proposed because it was noted that the linear loss produced potentially very large gradients, and there were concerns about stability. However, given more iterations the linear loss proved to reach good results as well.

One other variation that was attempted to try and improve the results, based on how the agent in the visual attention paper [7] was trained, was to calculate what the reward would be at every step, and use the gradient based on these to train the agent instead. The results are labelled everystep below, and generally seemed to do worse. The key difference seems to be that the output is based on the lowest observed value, rather than its current estimate of the truth at each step, so it ended up producing large penalties for what were potentially reasonable exploratory steps, and so producing more nuisance gradients that drove the policy away from optimality.

The exact setup chosen for the experimentation, in terms of number of layers and location of non-linearities, is detailed in figure 13. Each white rectangle is a fully connected layer with size equal to the variable written on it.

5.1.1 Experimental Method

There were four main experiments performed: a comparison of the impact of number of hidden units in the network to performance, a comparison between using ReLU and HardTanh for the transfer functions and with different reward schemes, a comparison of the agent's performance on higher order polynomials, and a comparison of the agent's performance with functions of different dimensions.

As can be seen in figure 14, the agent's performance on the validation data is noisier than it's performance on the training data, but they do track each other quite closely in general. It can also been clearly seen that the performance on the testing and validation data sets

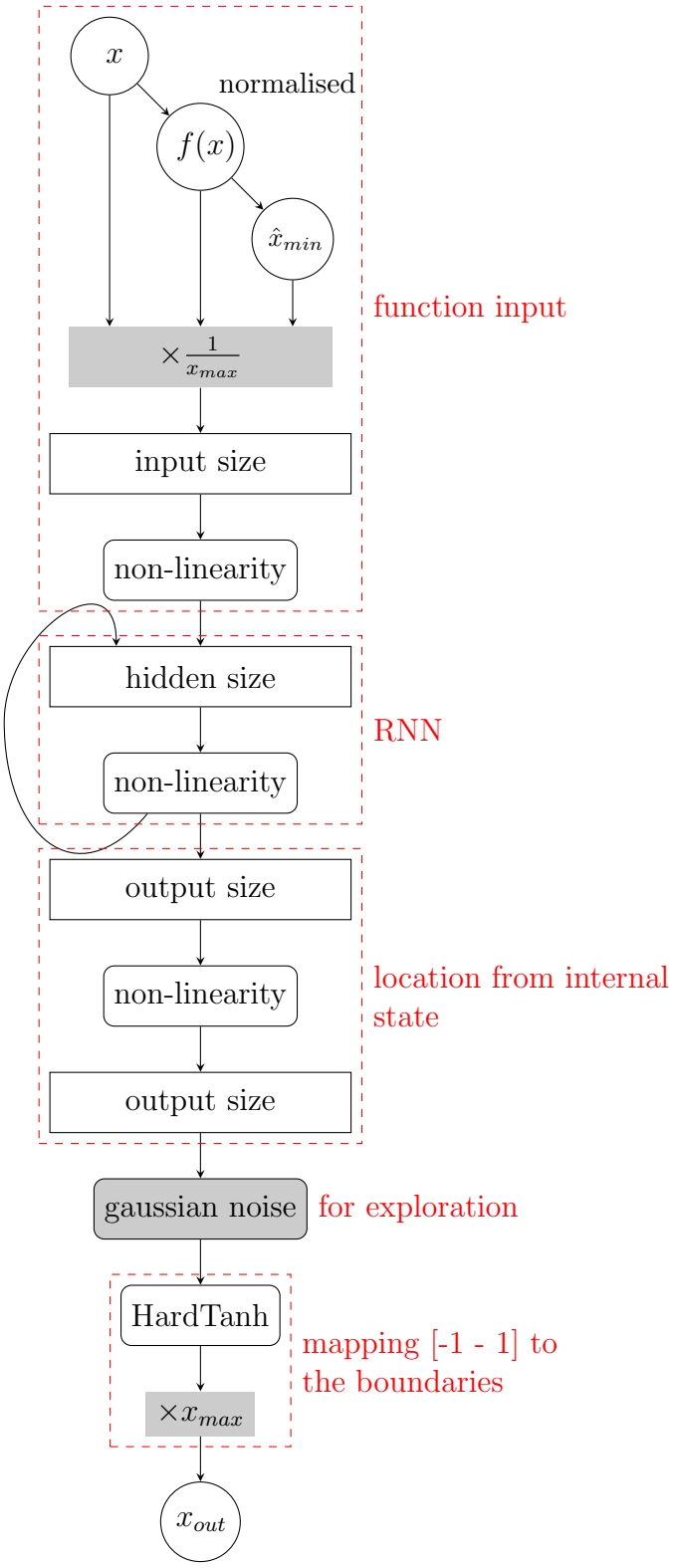


Figure 13: Layerwise setup for all of the RFO agents

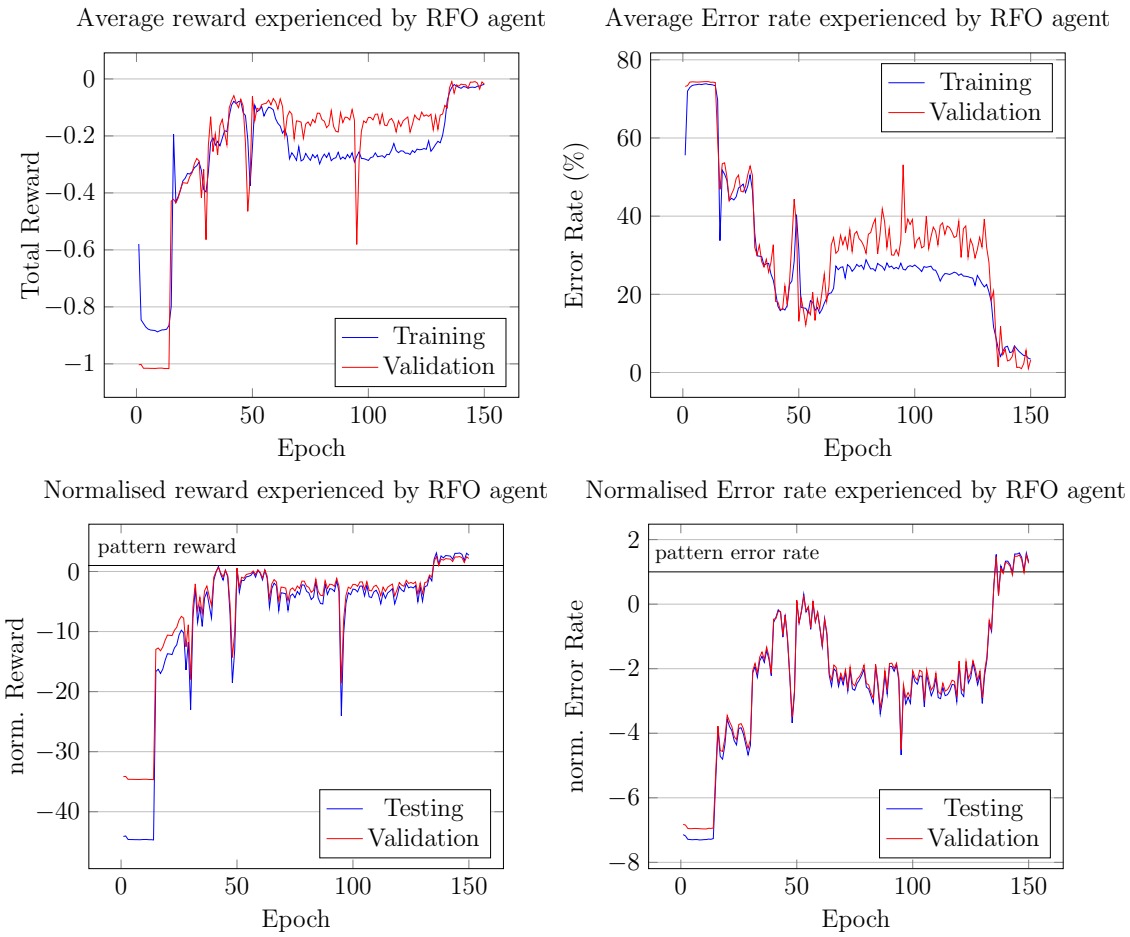


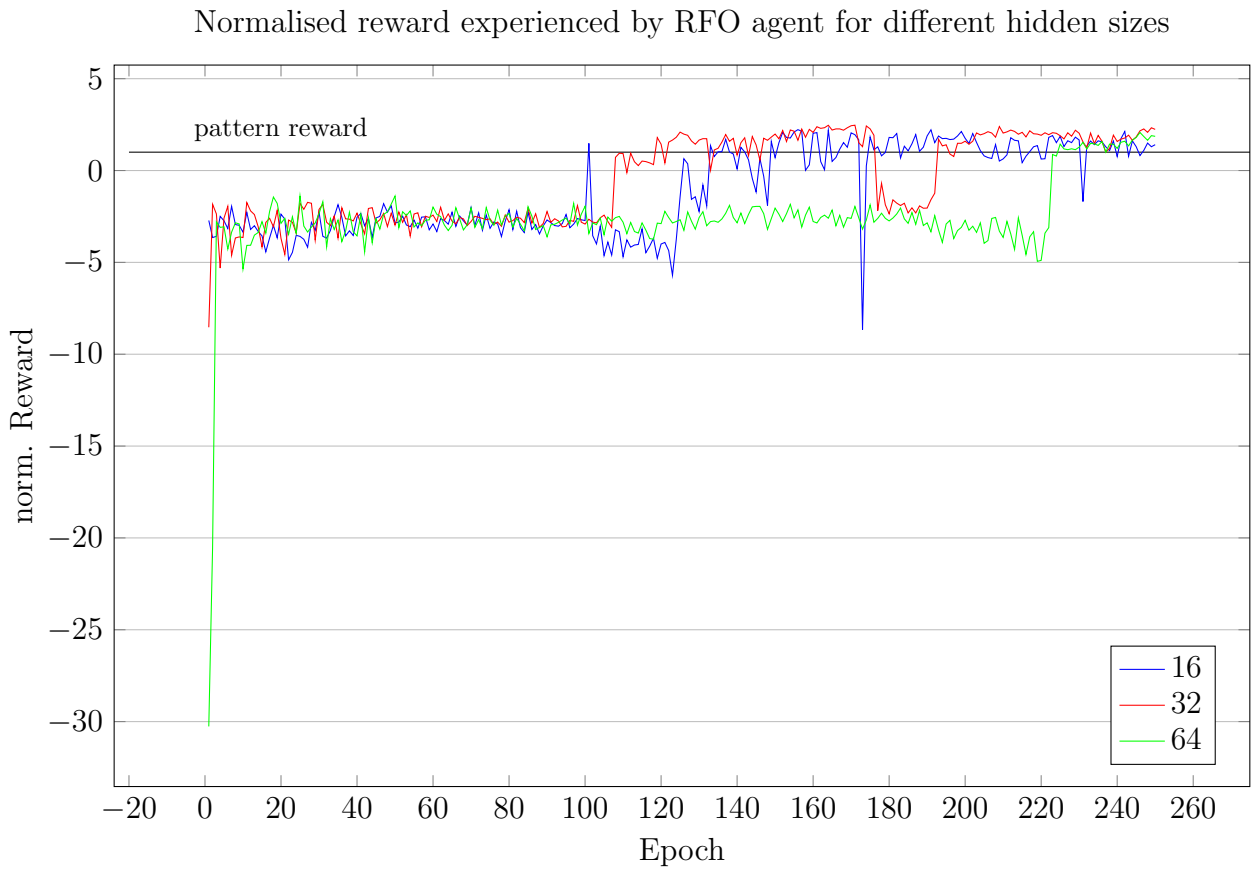
Figure 14: RFO learning behaviour and normalised performance across learning

are near identical. Furthermore, although there are important differences in what the data shows, the error rate and the reward show similar enough behaviour that, in the interests of space, for all further graphs only the agent's normalised test reward data will be shown. If the agent performing well, then it would have a normalised reward somewhat larger than 1, indicating that it is performing significantly better than the pattern search agent.

The implementation of the agent was ran on a single GPU for 250 epochs, and tested on the test data after each epoch. There were 163840 training examples generated and 20480 test examples. To simplify the search space for network sizes, the input and output sizes were always set to half the hidden size. The agent was given a linear reward only at the last step for all the experiments except the one comparing reward schemes.

5.1.2 RFO Results

In figure 15 the test performance for the agent given different hidden sizes is detailed. The transfer function was HardTanh, the agent had 20 steps to find the minimum in, the function was a one dimensional quartic, and the rewards were linear and given at episode termination.



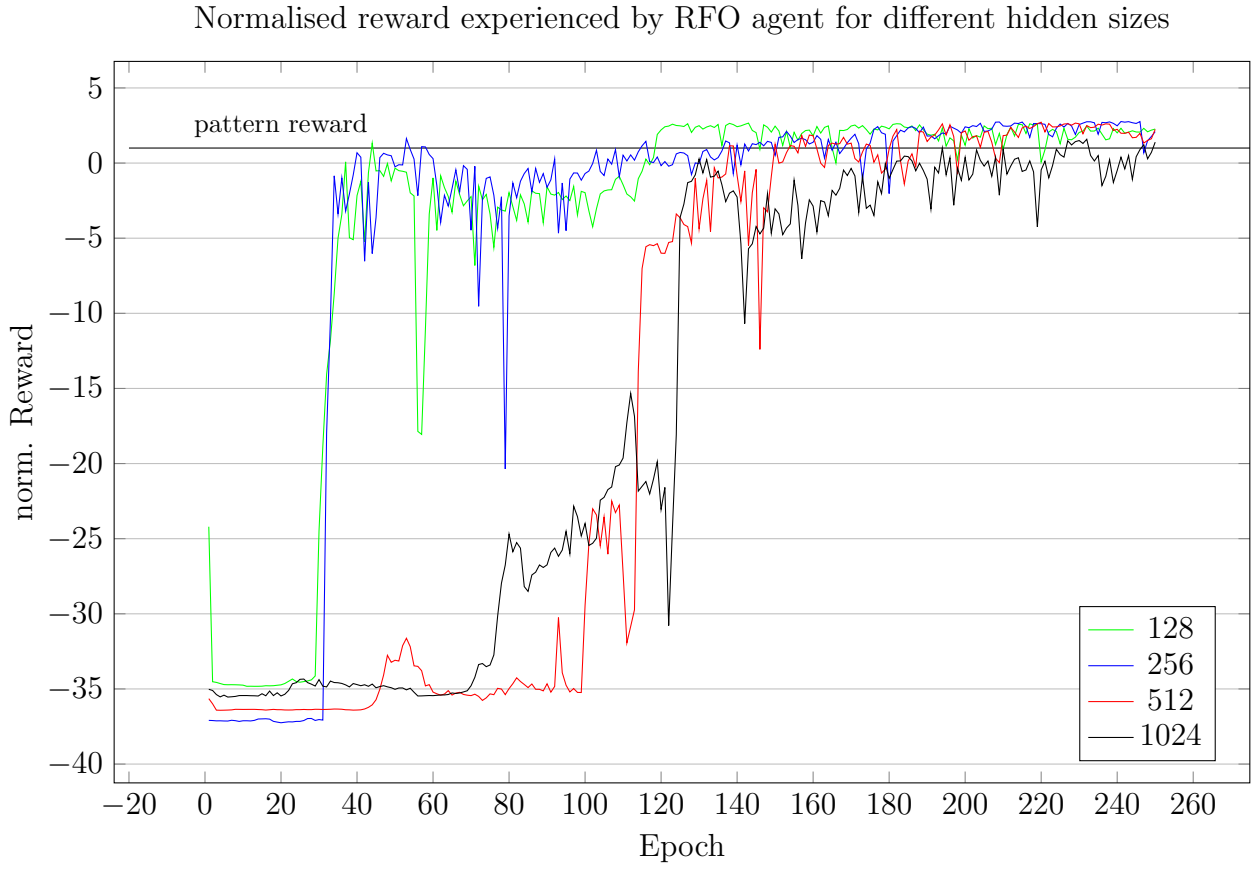
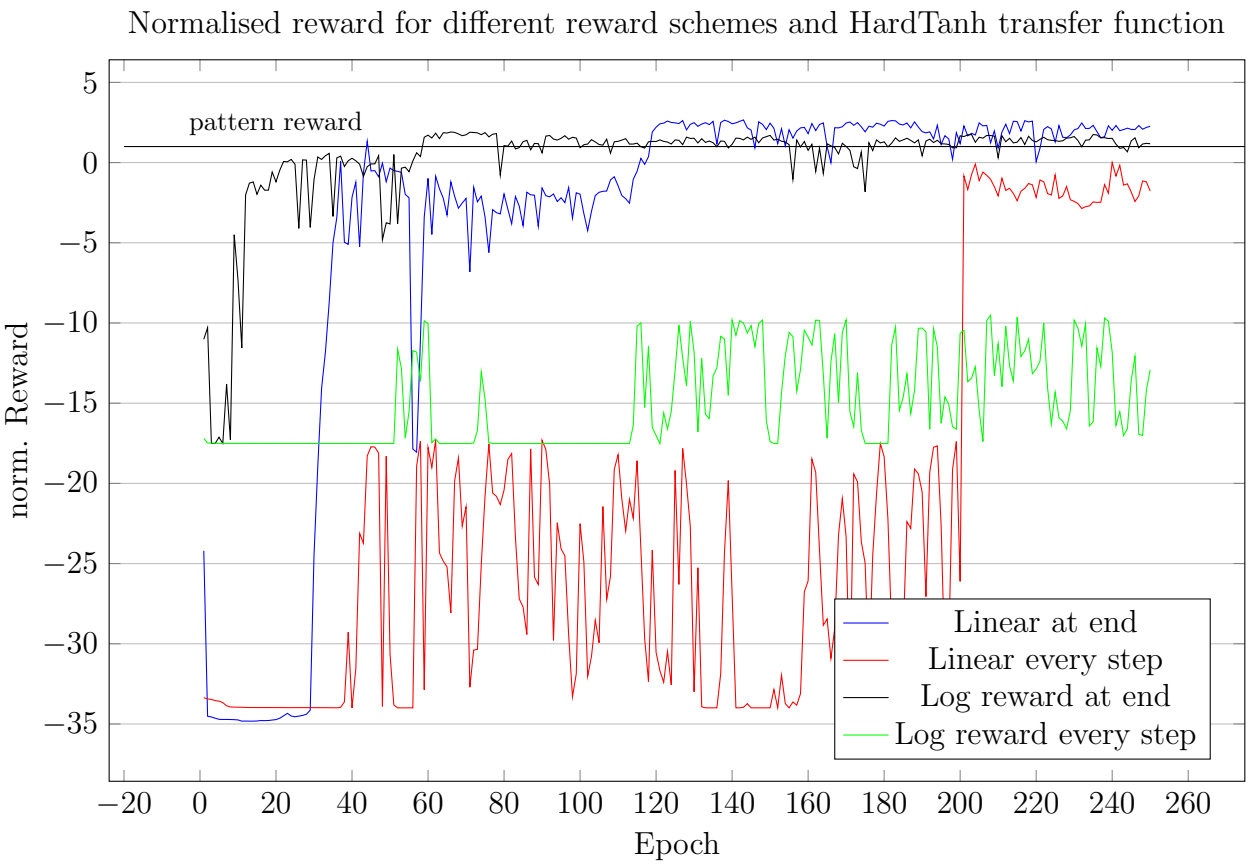


Figure 15: Comparison of Agent learning for different hidden sizes

Every single network size managed to produced a policy that exceeded the performance of the pattern search agent, though the networks with 128 and 256 hidden neurons produced the highest final reward. As the networks get larger, it takes the agent longer to find the initial jump in performance, which the small networks make within the first epoch or so. This jump is probably the agent learning some initial useful representation of state, which then allows it to learn how to make better decisions. For expediency and peak performance, a hidden size of 128 was chosen to be the norm for other experiments.

In figure 16 the agent's performance with different reward schemes and transfer functions is detailed. The hidden size was 128 neurons, the agent had 20 steps to find the minimum in, and the function was a one dimensional quartic. What is immediately apparent is that, except with a logarithmic reward scheme, the agent is unstable if it uses a ReLU transfer function. This is probably due to the effect noted when the log reward was devised: that large errors produce very large gradients, which would then dominate the learning and harm convergence. This is far more true for ReLU, which passes large positive values unhampered, than with HardTanh, which cuts off both very high and very low values. Even with the log reward, the HardTanh function produced better peak performance, though interestingly less stably. There could be an argument to use logarithmic rewards along with ReLU based on the apparent relative stability here, however it was decided that better peak performance was preferable here. With HardTanh, log rewards produced similar looking performance,with slightly lower peak values. However the comparison isn't quite fair, as with the same improvement in locations the difference in the log rewards will be less than the difference of the linear rewards ($\log b - \log a < b - a, 1 < a < b$). Telling the agent what reward to receive at each step proved counterproductive, probably because it then put a penalty on every action that didn't improve the observed minimum value, which hampers exploration and muddies learning, as it largely rewards the agent for being lucky with the first few values it observed, rather than behaving optimally.

In figure 17 the agent's performance on polynomials of increasing order is detailed (2 minima - quartic, 3 minima x^6 etc). The transfer function was HardTanh, the agent had 20 steps to find the minimum in, the hidden size was 128 neurons, and the rewards were linear and given at episode termination. The relative performance of evenly searching the whole space compared to a pattern search changed dramatically as the order increased. Because of that, for this experiment a different normalisation was used, which defines the reward on a scale where 0 is the performance of the pattern search agent, and 1 is the perfect result (0



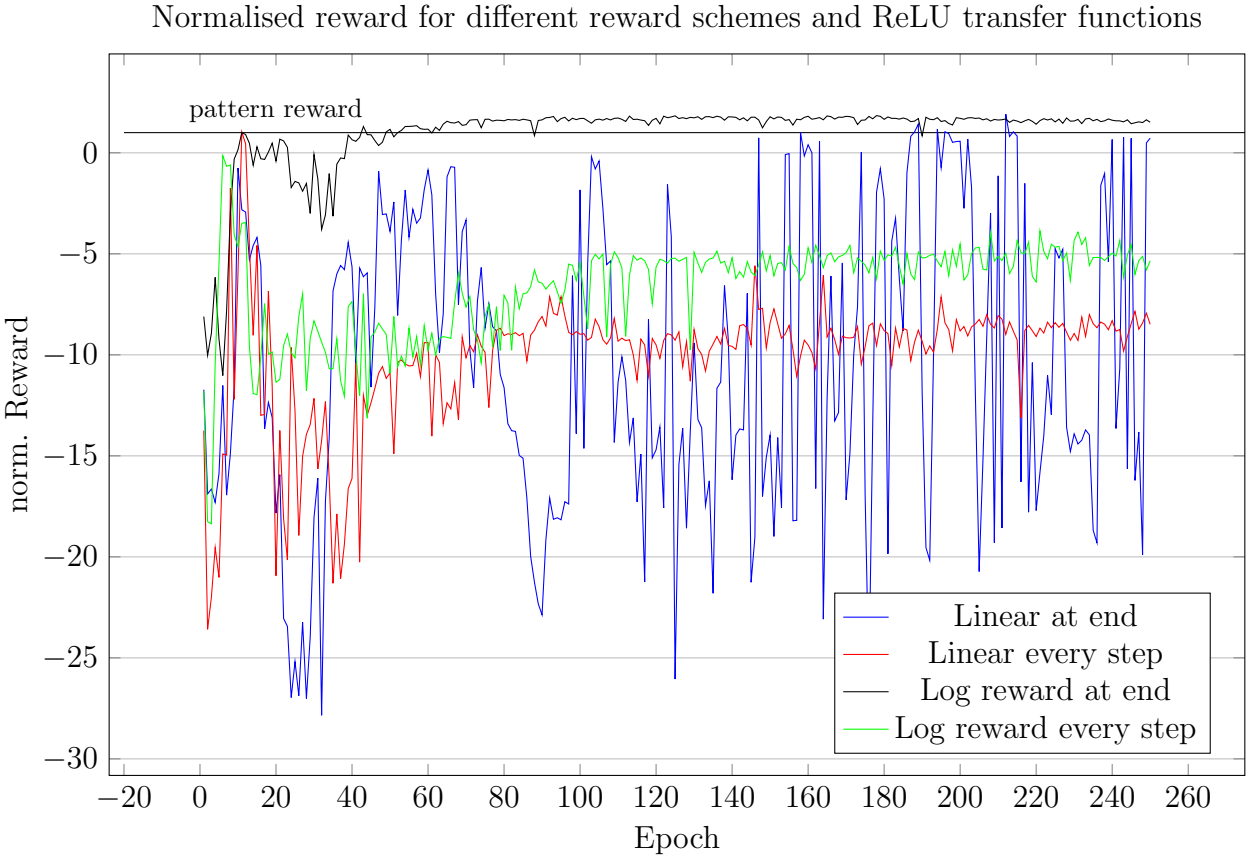


Figure 16: Comparison of Agent learning for different reward schemes and transfer functions

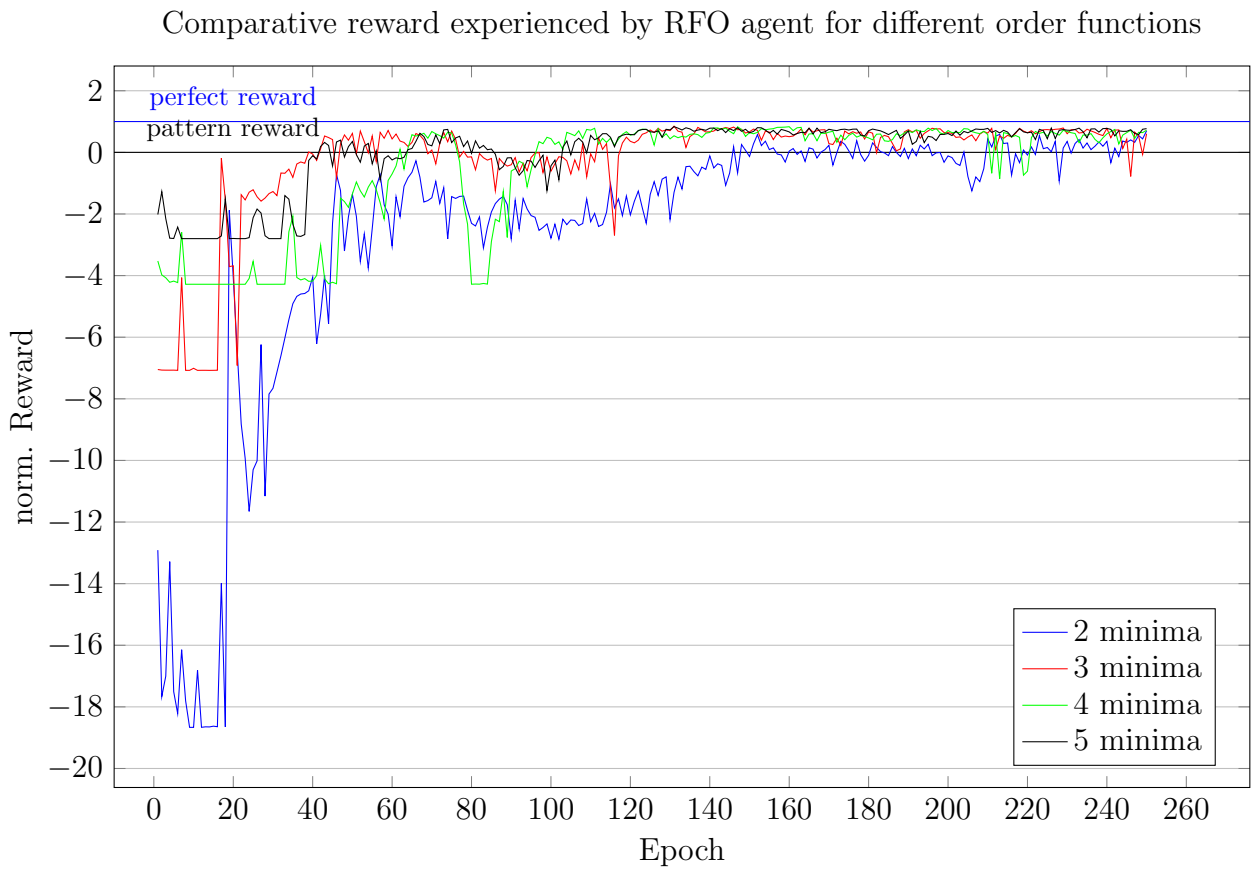


Figure 17: Comparison of Agent learning for different order polynomials

total reward). It should be noted that the agent attained similar results after the first epoch for each test, but as the order increased, the pattern search agent did worse, so the RFO agent did relatively better. The system consistently outperforms the pattern search, arriving to around 0.85 in this normalisation in each case. This means that the agent's absolute performance does decrease as the order increases, but at 15% of the rate at which pattern search falls off. It should also be noted that for very high order polynomials it is actually quite likely that several of the minima are quite close in value to the global one, improving the chances of the agent attaining a positive reward quickly.

5.1.3 Issues and Evaluation

In figure 18 the points that the agent checks as it moves through one episode are detailed. Notably there seems to be some combination of exploration and exploitation going on, with the agent hanging close to good values it's observed, whilst also choosing to go right across the range of possible values. It isn't performing some form of gradient step or anything else so intuitive, but it does seem to be stepping in direction for some reason. What exactly is going on would probably require detailed analysis of the neurons within this network, but in

any case the results are clear enough - this agent attained a normalised score of 1.8 with an error rate of <1%.

Overall, from the learning charts it can be noted that the learning is far from smooth, and, at least within the limited number of epochs given, not strictly convergent. This will partially be a product of reinforcement learning, but also due to the use of REINFORCE to train a deterministic policy - the method expects every step the agent takes to be stochastic, whilst with these experiments the noise was only added for the training. For all cases where the agent learned anything there are several large jumps in the agents performance, often after a long plateau of similar performance. These jumps are probably due to the agent learning some representational change that allowed it to improve it's behaviour suddenly, as the agent is learning not only how to make good decisions based on the state, but also how to represent this state to itself. These jumps mean that early stopping is not a good method to use with these networks, as important learning may be occurring without immediate change in received reward.

Ideally these jumps could be avoided by starting the agent off in a better area of the search space, and the performance with larger networks would certainly benefit from escaping from more of the local minima that such networks are naturally susceptible to. So whilst the agent's performance so far has been promising, the hope would be to do even better.

5.2 Apprenticed RFO

One suggestion to improve the performance, based on the work in Mastering the game of Go with deep neural networks and tree search[11], was to train the agent to first replicate "expert" output, then further improve the policy using reinforcement learning as before. The idea is that by first moving to a space where it is making good actions already, it would escape local minima and have a more defined manner in which it would learn to define the state space. The first key challenge with this is to choose what sort of agent it should learn from. Initially it was proposed to get it to learn from a simplex agent, but the implementations of simplex agents within the systems constraints were found to be consistently outperformed by the pattern search agent. So it was decided to use the pattern search agent instead. Ideally the agent would be trained by a Bayesian optimiser, but sadly time constraints prevented this from happening.

One possible advantage of this set up for training the agent is that it makes training the agent for an expensive task without a decent model for it a lot easier. Instead of either

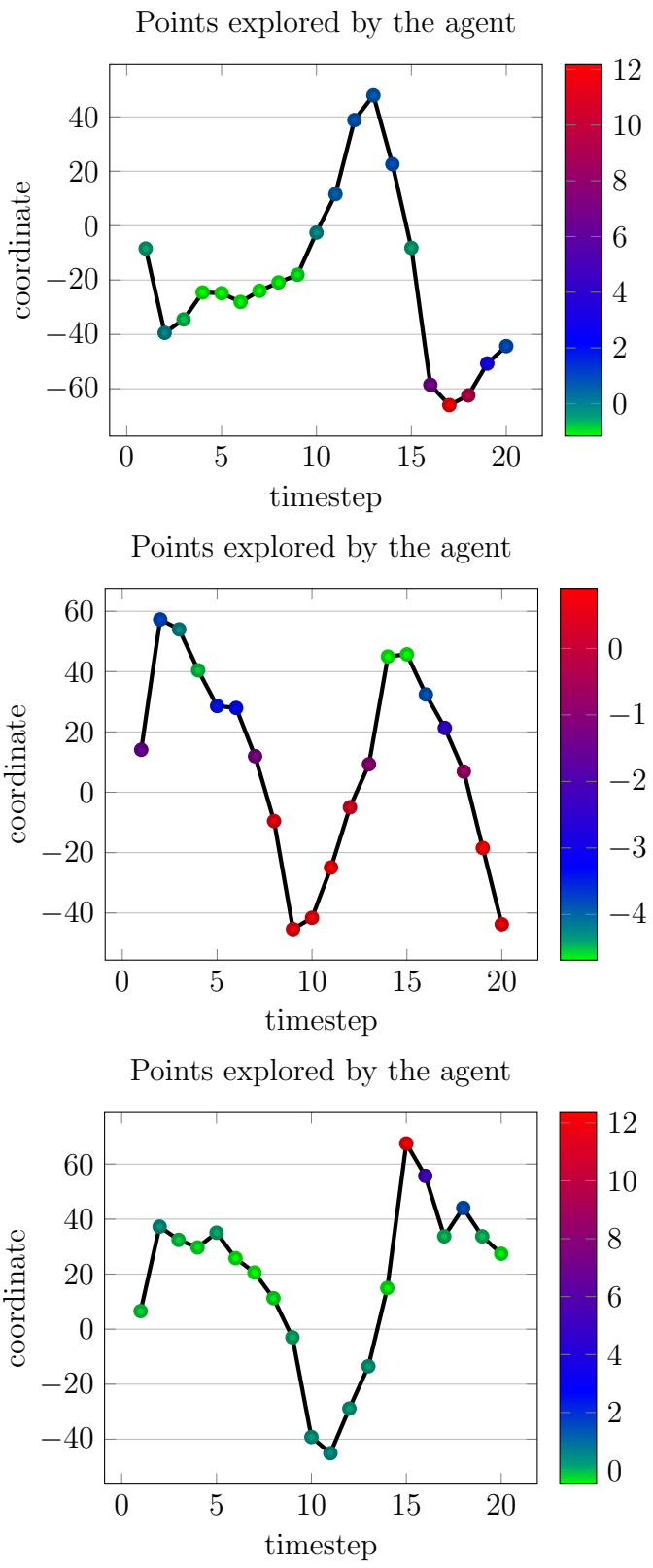


Figure 18: Points checked by the agent and their return values

training the agent directly on the expensive task, which would take a very long time and lots of computation, instead output data from previously used optimisers could be used to train the agent along with some loose approximation based on the data from those, which should reduce how much computational time the agent has to spend on "live" examples.

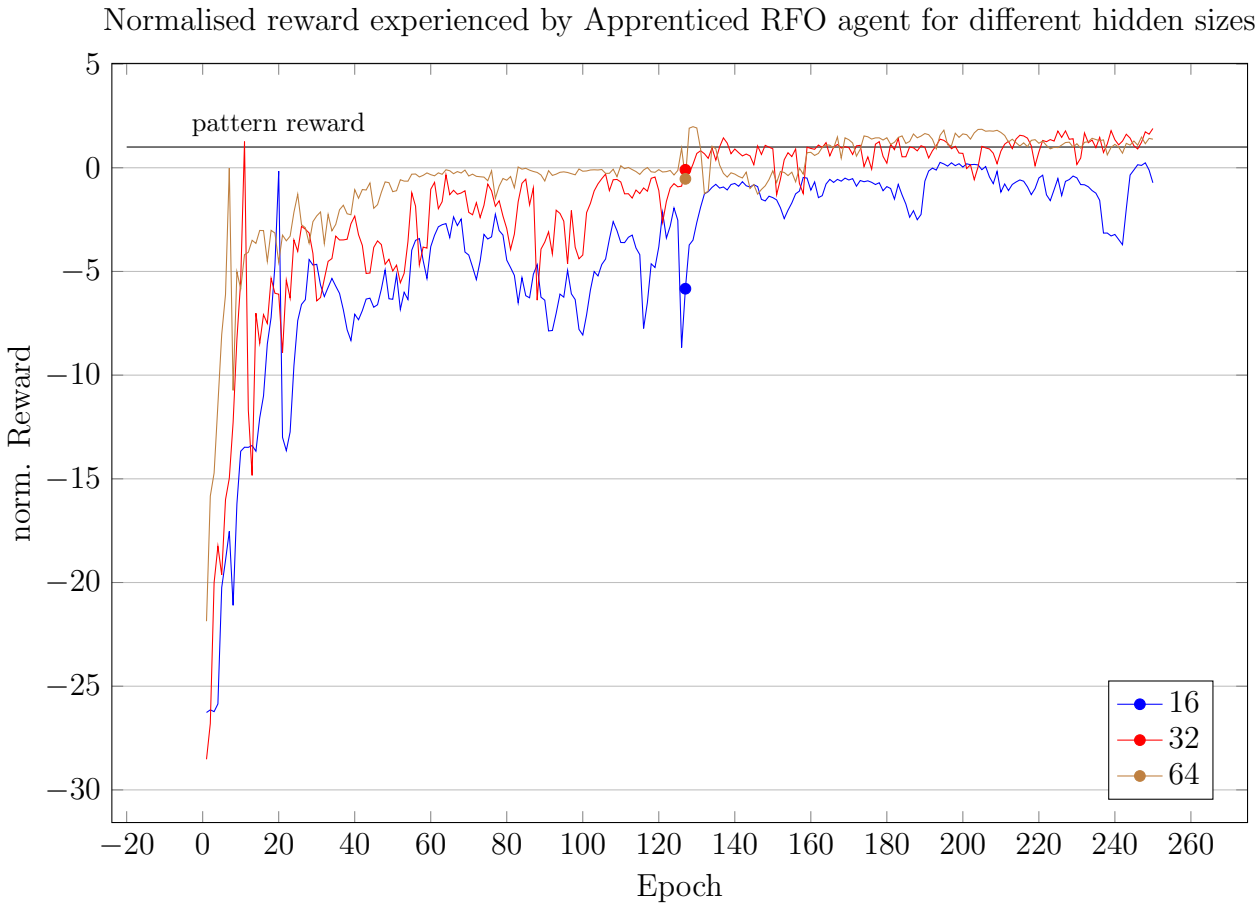
The agent was trained as follows. For each training function the agent produced an output, and in parallel the pattern search agent produced its own output. Mean squared error loss was used to produce the gradient, where the error was defined as the difference between the output of the agent for that step and the pattern searcher. The agent was trained with this for 125 epochs. Then the same networks were trained using reinforcement learning as with the RFo agent for a further 125 epochs.

5.2.1 Apprenticed RFO results

The same set of experiments with the same parameters was ran with the apprenticed RFO as with the pure reinforcement learning agent. The graphs are normalised in the same way, and the pattern search level is again indicated. Additionally, the step where the apprenticeship was turned off and the agent swapped to reinforcement learning is marked on each chart with a dot at that point.

Figure 19 details the normalised reward the apprenticed agent received for different sizes of network. The transfer function was HardTanh, the agent had 20 steps to find the minimum in, the function was a one dimensional quartic, and the rewards were linear and given at episode termination. Of immediate note is that, except for the 16 neuron network, by the end of the apprenticeship every network had converged to a policy that performed at roughly 0 reward, or as well as dividing the search space up evenly. They also all arrived there in very similar manners, as one might expect from a set of sufficiently large networks with regularisation. The response after the reinforcement learning starts is interesting - all networks with 128 or more neurons make this characteristic jump to a policy in region of 2 relative reward before falling away and descending into seemingly chaotic behaviour. The networks with less neurons fail to make such a jump, but also then are less chaotic. Again a network with 128 neurons seems to produce the best result, though less stably.

Figure 20 details the agents performance with different reward schemes and transfer functions. The hidden size was 128 neurons, the agent had 20 steps to find the minimum in, and the function was a one dimensional quartic. Again ReLU without log rewards is a flop, being unstable even in the apprenticed behaviour, though again the log rewards produce the most



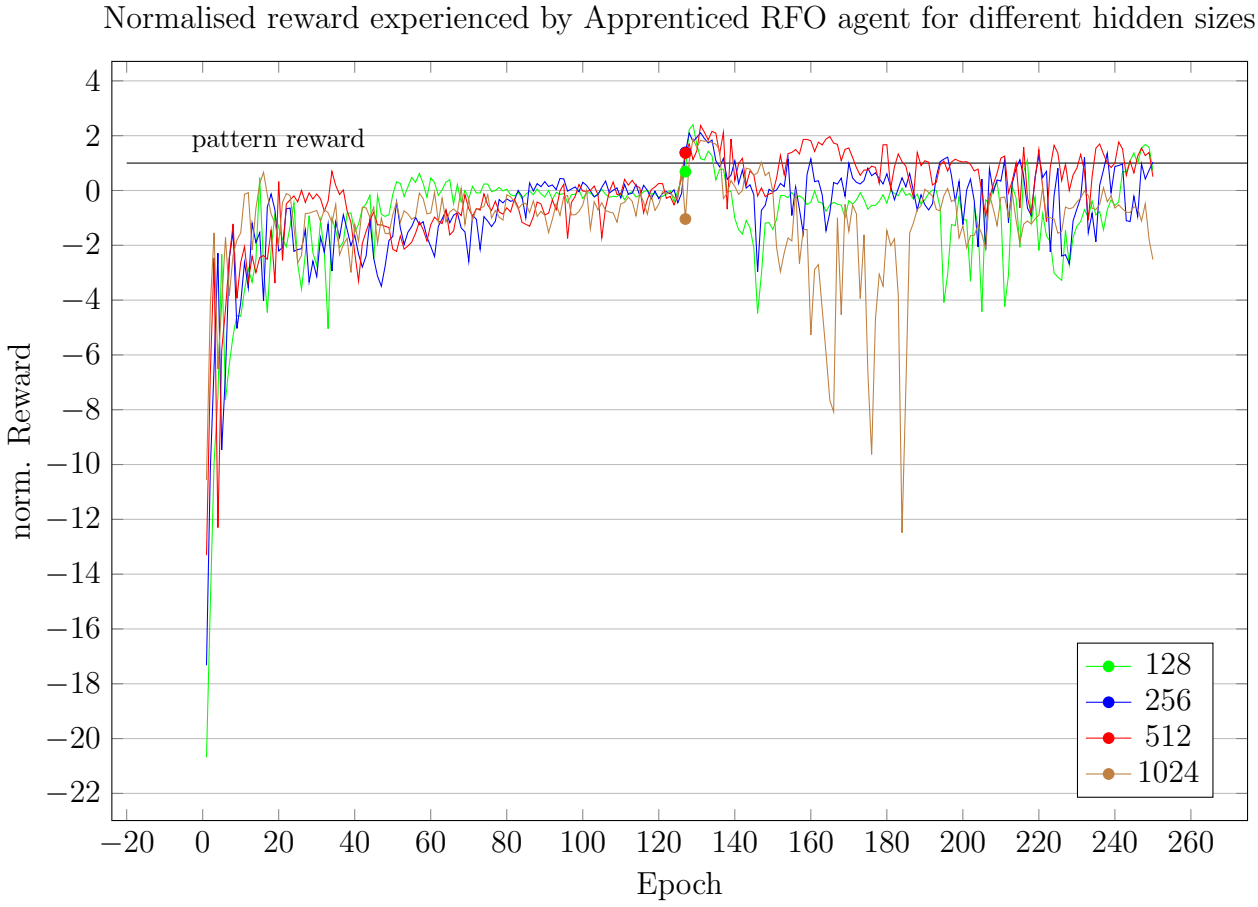


Figure 19: Comparison of Apprenticed RFO learning for different hidden sizes

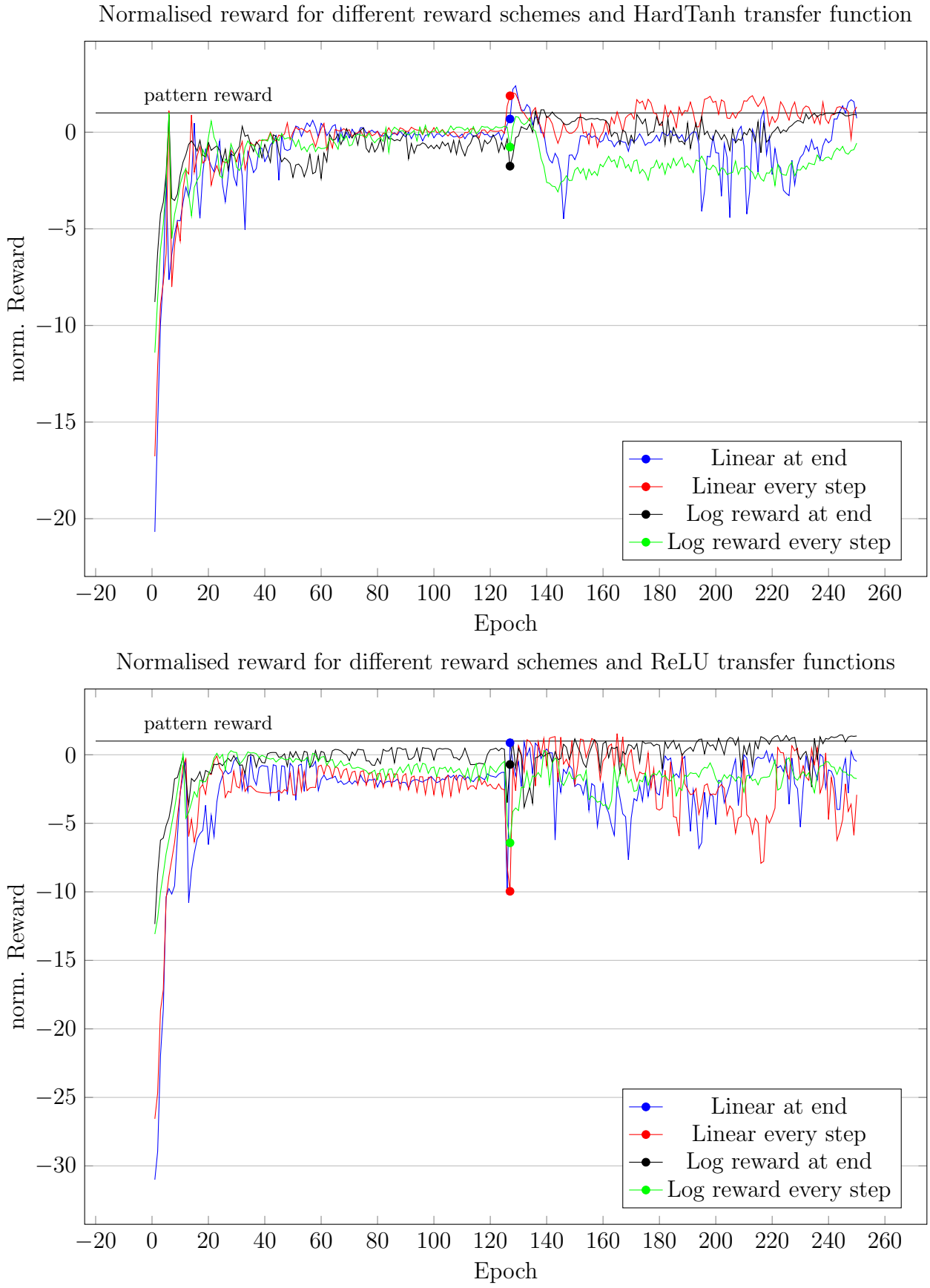


Figure 20: Comparison of Apprenticed Agent learning for different reward schemes and transfer functions

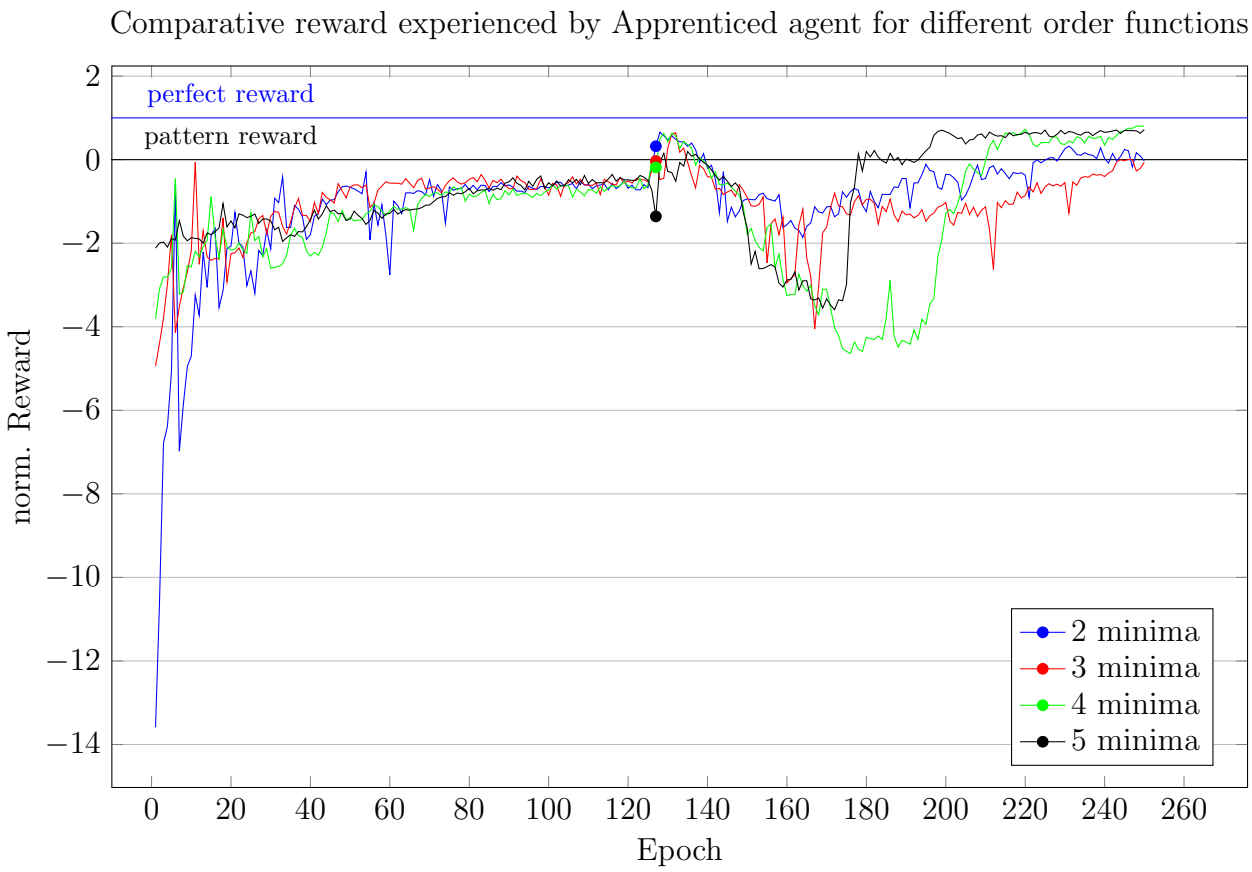


Figure 21: Comparison of Apprenticed Agent learning for different order polynomials

stable looking output. This may also just be a product of the log rewards being kinder on errors, as can be seen by it's apparent better performance during apprenticeship, even though during apprenticeship the only variable that matters is the transfer function. Here giving a linear reward at every step with HardTanh looks less terrible, perhaps because the agent was already taking sensible initial actions, so the harm that does to initial actions was lessened. In any case, this data shows the key benefit of the apprenticeship - even in set-ups more susceptible to local minima and instability, by starting the reinforcement learning in a better region it stayed in a better region.

Figure 21 shows how the apprenticed agent performs on polynomials of increasing order. The transfer function was HardTanh, the agent had 20 steps to find the minimum in, the hidden size was 128 neurons, and the rewards were linear and given at episode termination. Again, because the pattern search agent's performance compared to blindly searching the space changes with polynomial order, the score is normalised on a scale where 0 is the pattern search agent's reward and 1 is zero total reward. There is again the characteristic bounce just as it starts reinforcement learning, but for higher order polynomials the agent recovers to a policy better than the pattern search faster and more smoothly, probably due

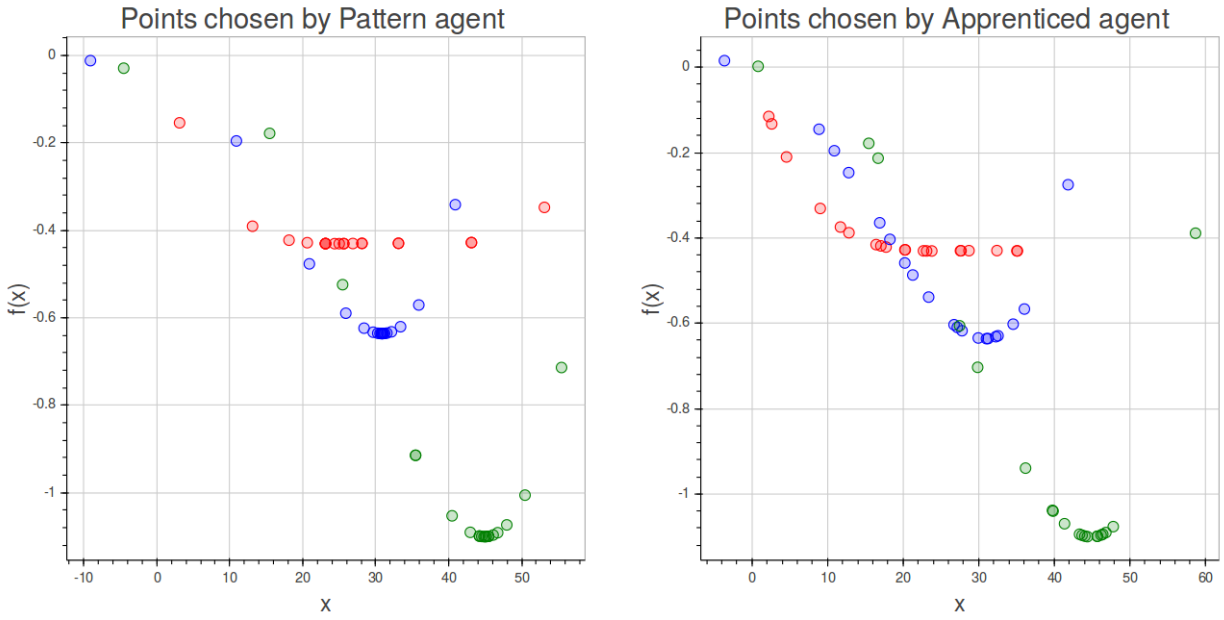


Figure 22: Points explored by pattern search agent and apprenticed RFO during apprenticeship

to the plurality of "good" minima available.

5.2.2 Issues and Evaluation

One of the key points of interest to look at are that the agent, although it is trained by the pattern search agent, only manages to attain a similar reward to blindly dividing the space into equal chunks. As can be seen in figure 22, which is a snapshot taken just before the reinforcement learning starts of three of the episodes, the agent's behaviour is very similar to the pattern search one. In fact, the apprenticed agent gets a normalised error rate pretty close to 1, so it is checking the same minima as the pattern search agent with a similar idea. But unlike the pattern search agent, the apprentice spends a bunch more checks coming down the sides of minima, with a much more even step size, so it doesn't drill down to the absolute minimum value as well. This means that, whilst it makes the same mistakes as the pattern search agent, it additionally gets worse rewards than the pattern search agent does when it is right because it doesn't get as close to the minima.

The second question is why does it make that bounce as it starts reinforcement learning. The answer is probably that most of it's learning at this point comes from the cases where the pattern search agent does badly, as it does reasonably well on the rest. So initially it learns to explore a bit more so that it can get to those points. It does so, whilst still largely imitating the pattern search agent, and so it gets a reasonable score on most of the cases, resulting

in normalised score somewhat above 1. However it is still getting gradients because it isn't making the perfect score. As it applies these changes, it makes some representational change that actually leads to a significant decrease in performance. If left to continue learning for a while, it may reach a different policy, or it may converge to the same policy as the pure RL agent did, depending on the shape of the error surface. However that would take more computational steps than just running the RL, so unless that unknown future policy is even better than just running RL for that long, it doesn't seem worth it.

Hidden Size	RFO	Apprenticed
16	2.23	0.26
32	2.47	1.89
64	2.07	1.98
128	2.66	2.40
256	2.76	2.10
512	2.71	2.37
1024	1.58	1.88

No. minima	RFO	Apprenticed
2	0.70	0.66
3	0.85	0.64
4	0.83	0.80
5	0.84	0.72

Table 1: Best normalised rewards for each of the networks

Suppose then that what is desired is for the best agent to be produced in a minimum of steps, and taking the networks from the peak of the bounce is seen as acceptable. The peak normalised rewards each network produced can be seen in table 1. For the same total number of epochs of training, the pure RL agent consistently outperforms the apprenticed agent except with the network with 1024 hidden units. Given that the RL got slower as network size increased, whilst the apprenticed part of the training stayed at the same rate, it can be extrapolated that the apprenticed RFO is better once the RL is that slow or worse. From the shape of the graphs, it would probably be preferable in any implementation using the apprenticeship to end the apprenticeship at 80 epochs rather than 125 so that the subsequent RL has more time to converge.

5.3 Deterministic RFO

Given that the environment is both deterministic and non-adversarial, the optimal policy for the agent should be deterministic. So REINFORCE can only ever asymptotically converge to the optimal policy, and given the actual implementation, not even that. So the methods described in Continuous control with deep reinforcement learning[3] ought to be able to produce better performance. A deterministic actor critic agent was created, within the same code structure as the REINFORCE agents, but this one is, in theory, off policy. The total

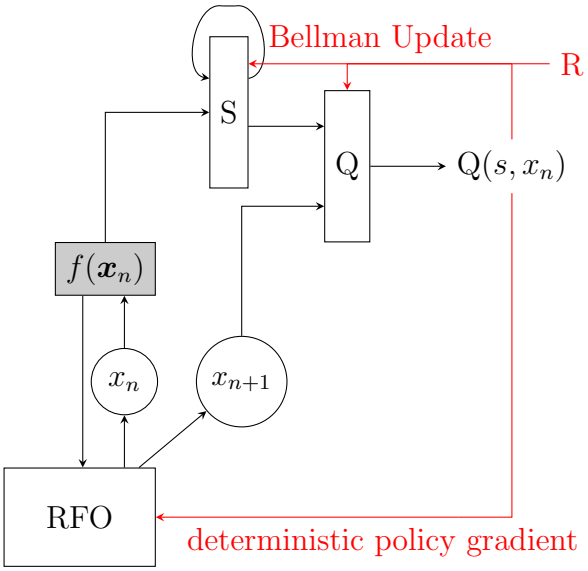


Figure 23: Deterministic RFO structure

structure of the agent is detailed in figure 23. The key additions are the critic networks, which consist of both an RNN that produces an observation of the state, that is the series of value-coordinate pairs so far observed, and the Q-network, which estimates the value of that state.

So far the learning with this has been much slower, though more stable than that with REINFORCE, it seems to also get stuck in an unprofitable local minima really easily, as can be seen in figure 24. It should be noted, however, that it was continuing to improve the policy along that line, just really slowly, which either means its converging into some minima or it's about to jump in a similar manner as with the RFO agent, but at epoch 3000 rather than 30. Given more time, the next step would be to find the hyperparameter set up that gives it the fastest learning rate whilst continuing being stable and running it with the apprenticeship method to get it away from the worse of the local minima. With the apprenticeship method, the network would be trained by error compared to the tutor whilst the critic learns from the experiences as well so that both are well initialised when it switches to reinforcement learning. Given how slowly it was learning there, the apprenticeship method should significantly save time. However time constraints prevent this from being done within the scope of this project.

6 Conclusions

Concerning creating an AI to play card games, this project made the following discoveries:

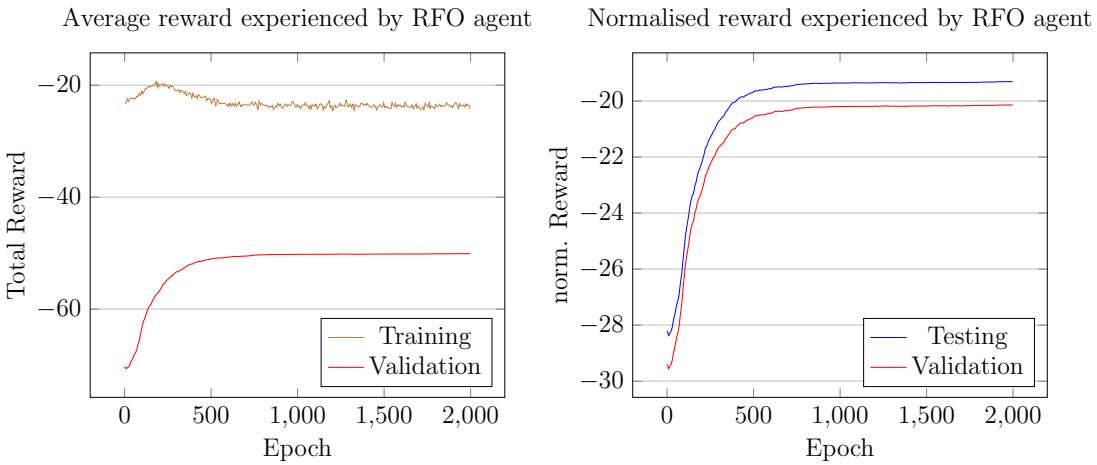


Figure 24: Performance of RFO with deterministic policy gradient

- Due to the existence of an exact model for the immediate effect of any action, and the highly variable available action set, it is typically expedient to produce a function that estimates the value of any state ($V(s)$) and using the model to work out what state one ends up in rather than one that calculates the value of taking an action in some state ($Q(s, a)$).

- Games with unbounded numbers of possible entities present complex challenges to the correct analysis of state with a neural network

- Indicator variables across the set of possible cards are not sufficiently expressive for the complexities of most card games

Concerning using Reinforcement Learning to train a neural network to optimise general functions, given that the optimum lies within known bounds, it was found that:

- Pure reinforcement learning is sufficient to train a recurrent network to sufficiently represent the state consisting of all previous location-value pairs observed

- Training the agent to replicate "expert" output prior to starting reinforcement learning removes the need to wait for the sudden jump in performance of the agent observed with pure reinforcement learning, but it doesn't improve convergence time. This is probably because the desirable policy for the agent is very different from the desirable policy to replicate the tutor with. It probably is only useful in cases where the learning is very slow.

- Learned stochastic policies are able to out-perform simple pattern search agents on non-convex polynomial functions.

Further analysis has to be done concerning the convergence for the learning for such algorithms, but if the snapshot of the network that performed the best on a sufficiently large test data is used, it seems that it is likely to perform well. It is unclear whether the good performance obtained would be replicable in situations where the location of the minimum isn't strictly bounded.

7 Going forwards

There are a number of areas in which this project could be advanced. One, much like with what they did in the Recurrent Models of Visual Attention paper[7], would be to add an extra output value to the agent that would choose whether or not to stop at that iteration. Then the agent could receive a penalty per step taken and learn initially inefficient methods that do produce good results before iteratively improving its efficiency. This would be particularly useful for more classical optimisation situations, where the precision rather than the computation available is the defined variable.

Another obvious area for improvement would be to modify the deterministic Actor Critic agent for this system so that it actually learns, applying many of the ideas laid out at the end of section 5.3. This may also require significantly more computational power to be made available to the system, or perhaps some form of the parallel learning described in Asynchronous Methods for Deep Reinforcement Learning [5], which looks like it could allow for such learning to be reasonably performed on a high end CPU.

Another clear oversight of this paper, due to a lack of available time, is the absence of any comparison to the industry standard Bayesian optimisers. This is probably the most urgent next step, that of creating Bayesian optimisation agent that can be both used for comparison and for tutoring the apprenticed learner a better policy. The impact for the apprenticed learner is likely to be particularly strong, as unlike with the pattern search, the Bayesian optimiser won't be drawn just into exploiting some local minima, but is much better behaved in terms of exploring other likely locations and reasonably reducing the likelihood of it not finding anything.

One further area to explore would be to test the agent on more advanced functions than just polynomials, for example by checking its performance on the Rosenbrock function or similar, or indeed simply to explore for minima where the location is unbounded.

Department of Engineering Science – Risk Assessment

Generic Display Screen Equipment Risk Assessment	Pages: 2
In Building: All Buildings	
	Review Date: 10/11/2015
Assessment undertaken by: Stephen Lilico Date: 23/11/2015	Signed:

Please Note: This is a generic Risk Assessment and highlights some common hazards identified with the use of Display Screen Equipment. Please note that: Students should read the University Policy: http://www.admin.ox.ac.uk/safety/ups0809.shtml.

Hazard	Persons at Risk	Risk Controls In Place	Further Action Necessary To Control Risk
Eyestrain / Headaches	User	**Take regular breaks every hour**. - undertake a different task. - adjust screen location to prevent glare or bright reflections. - Angle screen downwards to prevent reflection. - ensure no screen flicker. - ensure screen surface is clean. - ensure lighting is adequate for the task. - have an eye test if problems persist. - close blinds to prevent glare (as appropriate)	Consult Supervisor and advise Departmental Safety Officer (DSO) if problems persist. Please refer to the following link for a picture of good posture: http://www.hse.gov.uk/pubns/indg36.pdf
Back pain	User	**Ensure Workplace is correctly set up** - e.g. height of chair needs to be set so that forearms are parallel to desk. - ensure good posture at all times, sitting upright or slightly reclining. - Lower back supported to maintain natural curves.	Refer any medical issues to Supervisor or Departmental Safety Officer (DSO)
Aching shoulders, wrists	User	**Check seat height is correct** - forearms horizontal, level with top of desk. - keep wrists straight, use wrist rest. - No overreaching, exercise muscles. - Arms relaxed by side.	Refer any medical issues to Supervisor or Departmental Safety Officer (DSO)
Aching neck	User	**Check screen height is correct**	

		- eyes level with top of screen. - use document holder. - exercise muscles. - Check chair height e.g. forearms horizontal, level with top of desk	
Hazard	**Persons at Risk**	**Risk Controls In Place**	**Further Action Necessary To Control Risk**
Aching legs	User	**Check space under desk** to stretch legs, feet rest comfortably on floor otherwise get footrest. - exercise muscles. - Knees level with pelvis or slightly below. - Feet flat on the floor or use a footrest.	Remove items under desk which are preventing correct use e.g. boxes.
Water/Liquids	User	Ensure no water containers, coffee machines, kettles etc are located on or in close proximity to your workstation.	Building Inspections.
240V AC Electrical shock	User	User to check that all electrical leads to their PC are in good working order. Contact Electronics (Thom 5th floor) if Portable Appliance Label ‘out of date’ or not visible.	Supervisor/Student to check validity of PAT test label.

B Explaination of Code

B.1 Summary of code for the Magic the Gathering agent

The agent to play M:tG was also written in Java, which isn't ideal for optimisation, but was necessary because the emulator being used was written in Java. The neural networks library found did in fact have a c based back end and GPU implementation, though that turned out not to be the performance bottleneck anyway. The neural networks library was called neuralnetworks, found at https://github.com/ivan-vasilev/neuralnetworks. The emulator used is called forge, found at http://www.slightlymagic.net/wiki/Forge. The full code for this agent can be found at https://github.com/thesilencelies/LearnForge.

There were several adjustments to the overall infrastructure of the emulator to allow the additional option of a learned-ai player, rather than just human or the hard coded ai, which needed to be kept so that is could be used to train the learning agent. Asides from those, all the additional materials can be found in the module titled learnedai. This uses entities titled Q-cards to store the assessment of each card, then combines them together into a matrix that can be fed to the neural network that assesses play states. The elements that model the future states and choose the highest scoring actions already existed, and were largely left as is, with the main adjustment being the replacement of the heuristic score system with the neural network. The neural network is preserved because it is a static entity, and it was manoeuvred around the memory leak by regularly saving and loading the network, though that did delete the saved experiences from its experience replay.

B.2 Code for the Function Optimiser

The whole system is based on the implementation of Recurrent Visual Attention from the torch blog[8]. It is written in lua using the Torch nn libraries, and additionally depends on the dpnn and rnn packages. All of the files can be found in the package rl-optim, though the init file will need modifying. It is formed as an extension to the package dp.

The following sections detail the exact behaviour and interface of each of the additional Torch objects created for this project. Everything has been coded so that it can be ran in a batched manner.

B.2.1 FunctionData

This object is a dataSource for the dpnn experiment. It contains a function that takes an input of some parameters and a coordinate of the same dimension and returns a scalar. This function can be accessed by calling *getFunction()* It defines a series of parameters and calculates the minimum value for the function given those parameters using the algorithm defined in figure 10. These are used as the inputs and targets for the experiment, the targets being necessary to calculate the reward. Unlike with standard dataSources, the data is only created at runtime, to a quantity specified by "trainDataSize" and "testDataSize" parameters.

B.2.2 FunctionInput

This is an NN module which takes a reference to a function at its creation. This function should take a set of parameters and a coordinate of the same dimension as the parameters as an input and return a scalar value. It also requires the normalisation constant being used to be given to it, so that it can normalise the output of the function.

When *updateOutput* is called, the expected inputs are the batch of function parameters, the new coordinate to check, and its previous output for this batch, or an empty tensor if this is the first time. It runs the given coordinates through the function it was given at initialisation, then it returns a tensor containing the minimum value received so far from the function, the coordinate that was input to the minimum value, the last coordinate checked, and the value it received for it.

When *updateGradInput* is called it returns two tensors, one for the parameters, which is an empty matrix because the parameters are hidden from the rest of the model, and one for the location, which is the section of the gradient given to it that corresponded to the last location checked.

It has no internal parameters, so its effects are unchanged by the training.

B.2.3 RLFeedback

This is an observer from dpnn that calculates the total reward the agent received across the task and the agent's accuracy. These parameters are useful both for analysis and to enable early stopping. It has to be told whether or not to use log rewards on initialisation, and otherwise the behaviour is standard.

B.2.4 OptimReward

This function observes the output of the whole network, calculates the reward, subtracts the baseline and transmits the reward to the REINFORCE modules.

More specifically, this is a Criterion which takes as additional inputs the network being used (so that it can transmit the reward to it), the dimensionality of the function being explored (so that it can return tensors of the correct size), the normalisation constant being used, so that the targets can be normalised correctly, and whether to give log rewards and/or rewards on every step.

When *updateOutput* is called, it compares the output of the agent with the target and calculates the reward that it should receive.

When *updatedGradInput* is called, it checks its inputs for a second input that is the baseline reward, then reduces the rewards by that value before transmitting them back to the agent by calling *reinforce* on the module. The returned gradient is also the rewards so that the baseline reward can be learned.

B.2.5 reinforceEveryStep

A modification of ReinforceNormal from dpnn, This receives the reward received by the agent not only as if it stopped at the last step, but as if it had stopped after each step. Then it creates the gradient for each step based on that step's rewards. To do this, it has to receive a call from the recursor telling it which step it is at using the function *declareStepNo(stepNo.)*. In the forwards direction, this module adds Gaussian noise to its input if it is training, but not if it is testing.

B.2.6 RecurrentFunctionOptim

This module is a recurrent wrapper that handles the way the data is passed across the network, based upon RecurrentAttention from the rnn package. It takes as inputs on instantiation the recurrent network that estimates the internal state (self.rnn), the network that estimates the next location based on the internal state (self.action) and the module that handles the function calculation along with some parameters (self.minvalmod). It runs the algorithm for a fixed number of steps and then returns the concatenation of all the outputs of the function module.

More specifically, at each forwards step it asks self.action for the next location to check based on the current internal state of self.rnn, passes that value through self.minvalmod, then

feeds the output from that into self.rnn to produce a new internal state. This is repeated until it runs out of steps, and the final output is the concatenation of all of the outputs from self.minvalmod, to allow for everystep or other unusual reward schemes.

On the backwards pass it handles the back-propagation through time for all of its elements and the updates for their parameters.

B.2.7 EqualSearch, PatternSearch and AmoebaSearch

All of these objects are designed to produce the same type of output as RecurrentFunctionOptim, using the same number of limited calls to self.minvalmod, but instead of using internal neural networks to decide what location to check next, they use a hard coded algorithm to choose the next location. With EqualSearch it divides the search space into even sections and essentially ignores the output of self.minvalmod. PatternSearch maintains a centre coordinate, which it then checks around a fixed distance in each dimension, before moving the centre to the new lowest observed value, or reduces the search distance if no new lowest value is observed. The amoeba search attempts to implement the simplex algorithm, though it is a little more complex as the simplex algorithm often wants to make multiple function calls per step, which means that often the steps of the simplex algorithm are spread across several steps that the agent takes.

B.2.8 ApprenticedRFO

This module is based on RecurrentFunctionOptim, but it first implements an example based gradient descent training - teaching the modules to attempt to replicate the output of a pattern search. It has two additional function calls beyond the standard module function calls - *beFree()* and *backToSchool()* which turn this "apprenticed" behaviour on and off.

Whilst running the apprenticed mode, in addition to the standard forwards pass from RecurrentFunctionOptim, it also produces a separate internal value called tutor, which is the equivalent results that would be produced by a pattern search agent were they to start at the same location as the agent under training, which is calculated using the same code as that ran in PatternSearch. Then, as long as the apprenticed behaviour is on, the gradient given to the agent isn't based upon the reward received (although the early stopping is), but instead on the difference between its output and the tutors for that step. In order to achieve this, a slightly modified form of ReinforceNormal was made that had the additional function *enableReinforce(bool)* which told it whether or not to add noise and produce a gradient based

on the reward, or simply pass through its inputs and gradients.

When not running the apprenticed mode, it is nearly identical to RecurrentFunctionOptim with the exception that it has a randomised starting location.

B.2.9 DetRFO

This module takes the same inputs as RecurrentFunctionOptim, plus a state generation (self.Qstate) and value-from-state-action (self.Qact) network for the critic. It implements an actor critic approach to the learning problem, along with a deterministic policy gradient, following the algorithms laid out in the paper on continuous control using the deterministic policy gradient[3], which are explained in section 5.3. It expects its function reinforce to be called by some criterion telling it what reward it has received.

Upon initialisation, it creates copies of self.action, self.Qstate and self.Qact to be the target networks. These networks do not have their parameters updated using the standard learning rules, rather they are updated to track the values from the originals using the update step

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \tag{19}$$

Where τ is very small.

updateOutput is just like with RecurrentFunctionOptim. However, when *updateGradInput* is called, first it saves the last experience, inputs, actions and rewards, into its experience replay memory, then it loads a new batch of experiences at random from the experience replay. Then it runs the critic networks on the experiences, and produces the gradient with which to update the actor networks. Then it produces the gradients for the critic to update, using a target generated from the bellman step for that experience, that is

$$Q(s,a;\theta) = r + \gamma Q(s,a';\theta') \tag{20}$$

where a' is the action produced by the target actor network for that state. Then it creates the internal state for that experience in the actor networks, before performing the back propagation for the actor, using the deterministic policy gradient.

C References

References

[1] M.P. Cuéllar, M. Delgado, and M.C. Pegalajar. "Enterprise Information Systems VII". In: ed. by Chin-Sheng Chen et al. Dordrecht: Springer Netherlands, 2006. Chap. an application of non-linear programming to train recurrent neural networks in time series prediction problems, pp. 95–102. ISBN: 978-1-4020-5347-4. DOI: 10.1007/978-1-4020-5347-4_11. URL: http://dx.doi.org/10.1007/978-1-4020-5347-4_11.

[2] Sergey Levine et al. "End-to-end training of deep visuomotor policies". In: *arXiv preprint arXiv:1504.00702* (2015).

[3] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

[4] Wang Ling et al. "Latent Predictor Networks for Code Generation". In: *arXiv preprint arXiv:1603.06744* (2016).

[5] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *arXiv preprint arXiv:1602.01783* (2016).

[6] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[7] Volodymyr Mnih et al. "Recurrent Models of Visual Attention". In: *CoRR* abs/1406.6247 (2014). URL: http://arxiv.org/abs/1406.6247.

[8] Element Research Nicolas Leonard. *"Torch blog : Recurrent Model of Visual Attention"*. 2016. URL: http://torch.ch/blog/2015/09/21/rmva.html.

[9] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. "On the difficulty of training recurrent neural networks". In: *arXiv preprint arXiv:1211.5063* (2012).

[10] David Silver et al. "Deterministic policy gradient algorithms". In: *ICML*. 2014.

[11] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489.

[12] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. 1st. Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262193981.