Report for 4YP Project: Gaming AI

Stephen Lilico

April 13, 2016

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 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 it's 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 in [6] was used to develop an end to end training system for visual control for a robot attempting a number of difficult tasks in [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 Literature Review

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. 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 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 $f(x)$ 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 it's behaviour.

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:

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

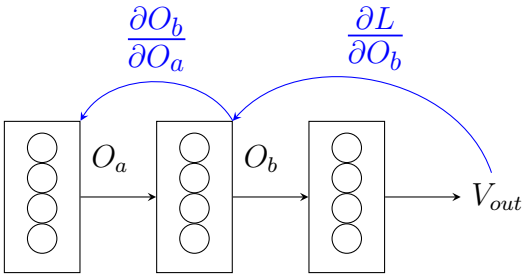


Figure 1: The Backpropagation Algorithm

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 then

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

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 likely to get stuck in unprofitable local minima. One standard trick that help reduce that is momentum, whereby each gradient update step for the 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 it's ability to generalise. There are a number of ways to combat this, one can keep the number of parameters available to the

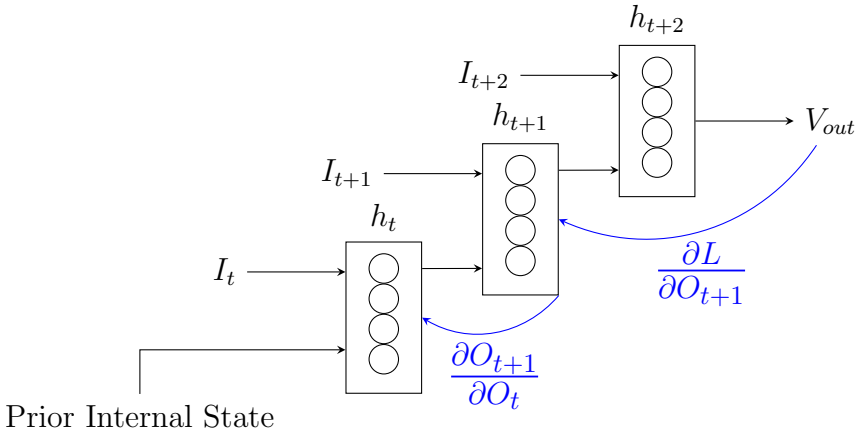


Figure 2: Back-propagation through time

network low, which means that it doesn't have the ability to fit the much higher order noise. However it's 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. In general with neural networks weight decay, whereby each weight is reduced by some amount after every step, or hard norm limits, whereby it scales all the parameters so that they don't exceed some limit on the norm, are used.

3.1.1 Recurrent Neural Networks

A recurrent neural network (RNN) is a particular architecture of an artificial neural network where a layer takes it's 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:

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

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

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

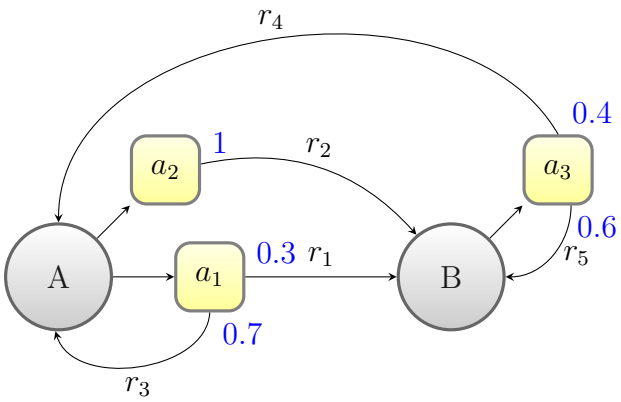


Figure 3: A markov decision process

where L_t is the loss at time step 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. [9]

RNNs are very powerful, having been shown to be technically Turing complete, and are capable of handling a much broader range of situations than pure feed-forwards networks. However they have their own additional issues - they are much more prone to exploding and vanishing gradients, where the gradients of elements many steps before the reward either produce exponentially large or exponentially small gradients, either 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, meaning they are much more likely to end up in unhelpful local minima.

One trick to help with the exploding gradient is to reduce the norm of the gradient of any layer before averaging to some limit by scaling down all the gradients of any layer who's gradient norm is greater than 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 it's 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 depending on what transition happened. An important property of an MDP is that it is Markovian, that is that 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 or anything else like that. One such process is displayed in figure 3

A reinforcement learning agent is primarily concerned with estimating two functions: the Value function and the Q function, which are shown in figure 4, based on the policy, π it is assessing. The policy determines what action is chosen in each state. The value 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,

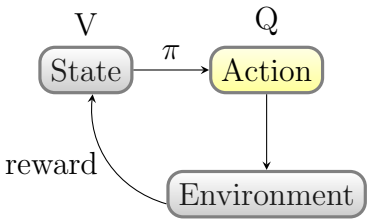


Figure 4: A simplified view of the Reinforcement learning problem

which estimates the expected reward the agent would receive if it were to return to following π after taking that action in that state.

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 it's 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.

$$V(s_n) \leftarrow r + \gamma V(s_{n+1}) \tag{10}$$

$\gamma < 1$ 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:

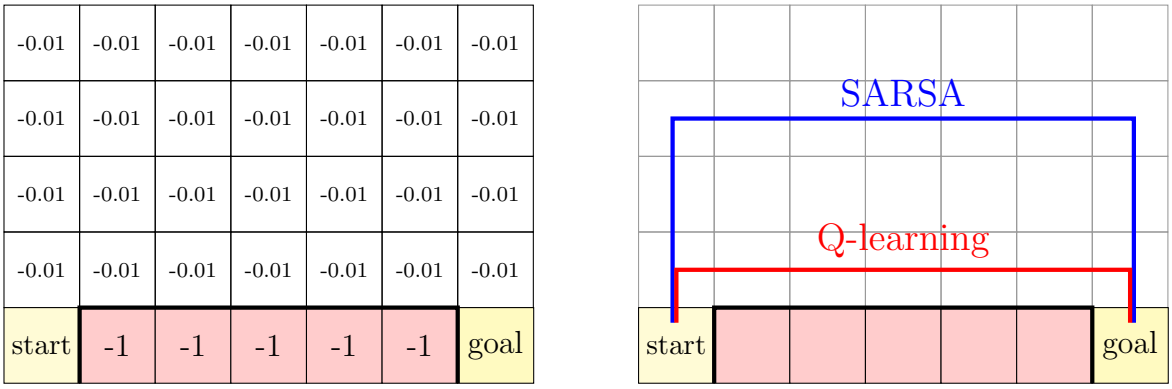
$$Q(s_n, a_n) \leftarrow r + \gamma Q(s_{n+1}, a_{n+1}) \tag{11}$$

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 it's 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.

Q learning also uses an epsilon greedy policy to choose it's actions, however Q learning is an off policy algorithm, which actually learns about the purely greedy policy. In Q learning

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



(a) The gridworld

(b) The paths taken by the agents

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

the update for Q is:

$$Q(s_n, a_n) \leftarrow r + \gamma \max_a \{Q(s_{n+1}, a)\} \tag{12}$$

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 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 isn't 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 local updates due to the spreading from the function approximation. 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.

One other interesting impact of using a function approximation is that, because by it's nature the function approximation can't produce the true Q value everywhere, it's 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 it was considered impossible to use neural networks as the function approximators in reinforcement learning. However, in [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. The randomness helps reduce the chance of the network "forgetting" something that it learned from a previous transition and helps improve learning stability.

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 :

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

$$\theta = \theta + \alpha \frac{\partial L}{\partial \theta} \tag{13}$$

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 discretization 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 as:

$$J(\theta) = \mathbb{E}\big[\sum_t = 1^T r_t\big] = \mathbb{E}[R] \tag{14}$$

In REINFORCE the sample approximation to this gradient is formed as, after running through M episodes:

$$\nabla_\theta J = \frac{1}{M}\sum_{i=1}^{M}\sum_{t=1}^{T}\nabla_\theta \log\pi(s^i_{1:t};\theta)(R^i_t - b_t) \tag{15}$$

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

$$\nabla_\theta J = \mathbb{E}\big[\nabla_\theta \mu(s;\theta^\mu)\nabla Q^\mu(s,a)|_{a=\mu(s)}\big] \tag{16}$$

This can be implemented using an Actor-Critic method, whereby there is a separate actor and critic networks, the actor implementing μ and the critic Q^μ. The actors 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 [3] the above was combined with the insights from [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 [4] google deeepmind 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 [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. 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. However 1 step and TD lambda methods (which use a weighted aggregate of the reward over the future steps) both found optimal policies in the maze very efficiently whilst using a tabular representation of the correct behaviour.

The function approximation used to feed it into the neural network was an indicator function across all locations fed into a deep feed forward neural network. 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 it's 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 it's 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 it's 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 7 training with the algorithm in 8

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

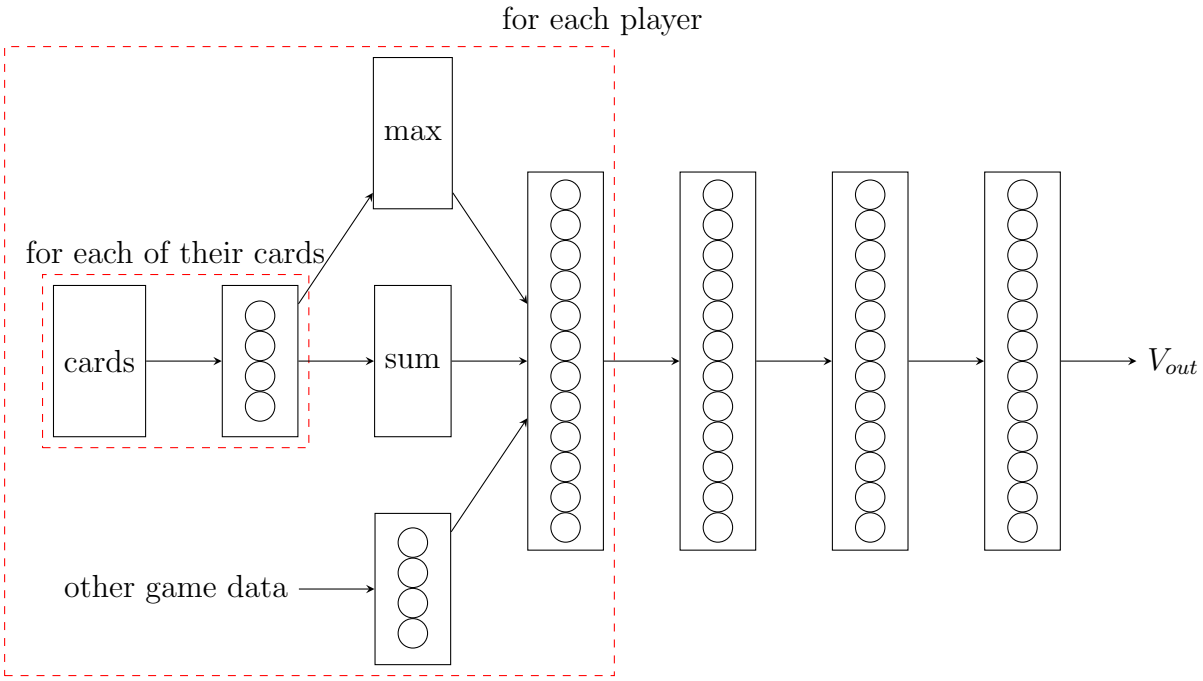


Figure 7: Architecture for the Magic: the Gathering playing agent

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 it's win rate never improved, and when tested against a human player it's 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 it's 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 it's 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 [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 it's hand, allowing it to make better decisions about when to allocate the resources it has.

$V(s;\theta) \mapsto \mathbb{R}$
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 8: Value iteration algorithm for MtG agent

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 $\begin{cases} steppenalty & \text{non-terminal} \\ -Loss(f(\boldsymbol{x}), f(\boldsymbol{x}_{min})) & \text{terminal} \end{cases}$ where $Loss(a, b)$ 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 further considered that it was known (or constrained to be) that the minima would lie within some known finite subspace of $\mathbb{R}^n$, which in practice meant 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 it's simplicity, but also because of it's 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 and one with a distant x value for which $f(x_{min_l}) = f(x_{min_g}) + \epsilon$. Suppose the agent explored nearby to both of these minima, and happened to observe a point enough closer to the local one that that value was lower than the one close to the global and so returns the one close to the local minima. Under a reward scheme based on x that would be heavily penalised, despite being entirely logical. So one based purely n $f(x)$ 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 9.

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

```
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)          ▷ (higest order term always has param of 1/it's order)
        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 9: algorithm to generate the functions

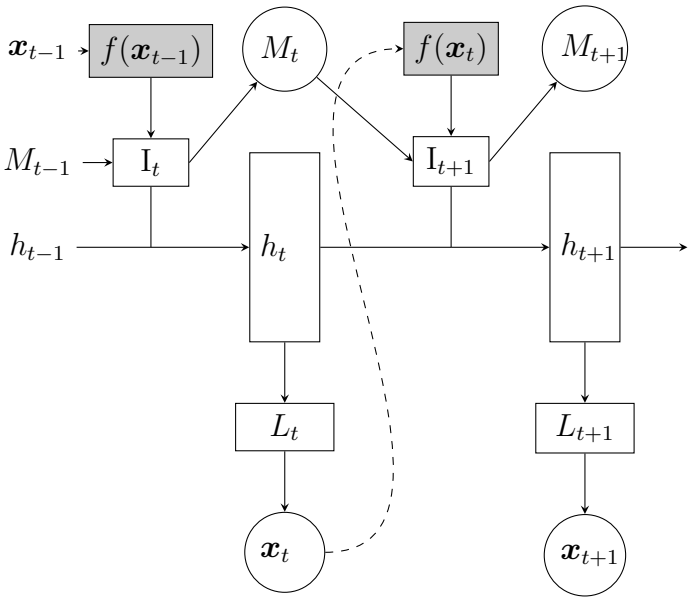


Figure 10: Architecture for recurrent function optimisation

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 dodgy manner. This does remove the opportunity for the agent to learn how to guess what the true minimum would be based on it's 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 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 [7]. It used the overall structure shown in figure 10 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 architecture that was designed can be seen in fig 10. 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 [7], there is no classification, so no classification loss to train the RNN with, so it is only being trained by the REINFORCE module.

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 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 it's 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, it seems that the log loss massively slowed learning down, and the agent seemed to stabilize to some policy relatively quickly. This is probably because the ideal loss function curve should be linear with the error for large error, whilst the log term is less than that.

One other variation that was attempted to try and improve the results, based on how the agent in [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 it's 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 11.

5.1.1 RFO Results

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, a comparison of giving the reward at every step or only the first, and comparison of the agent's performance with functions of different dimensions.

For the size experiments, the number of neurons in the input and location layers was half that of the number in the RNN, and all other parameters were kept constant.

Both the everystep and the ReLU vs HardTanh experiments ran the same parameters and expanding number of neurons as the

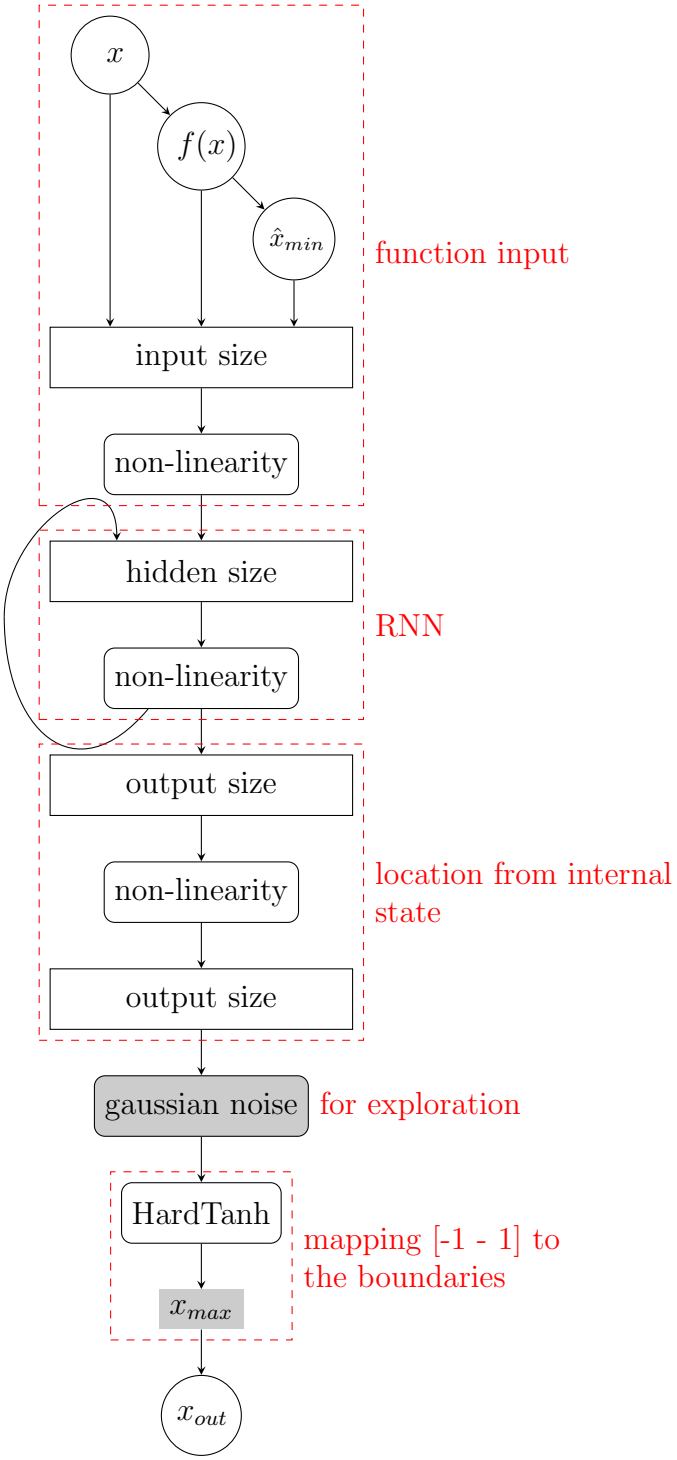


Figure 11: Layerwise setup for all of the RFO agents

Figure 12: Points checked by the agent and their return values

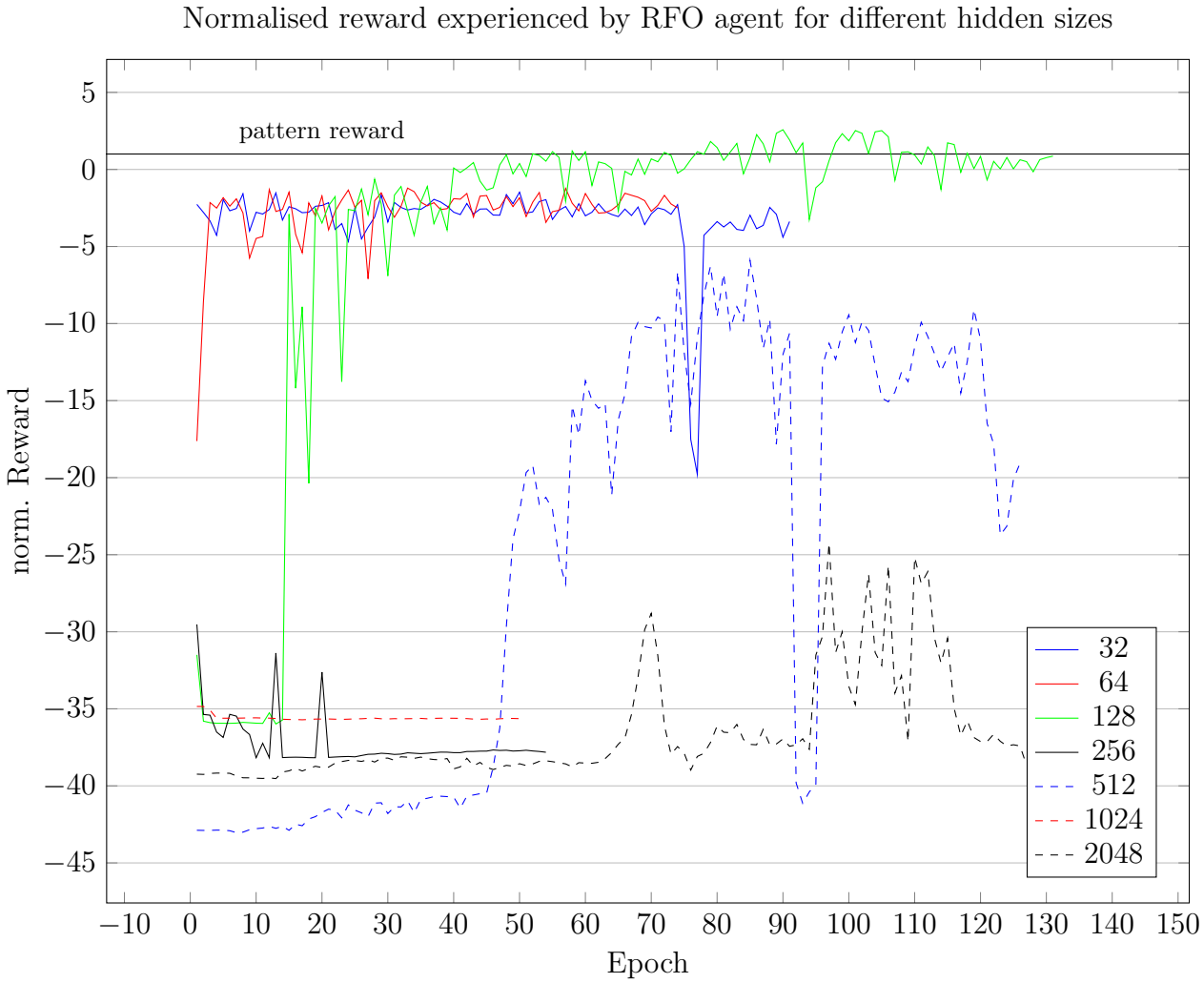


Figure 13: Comparison of Agent learning for different hidden sizes

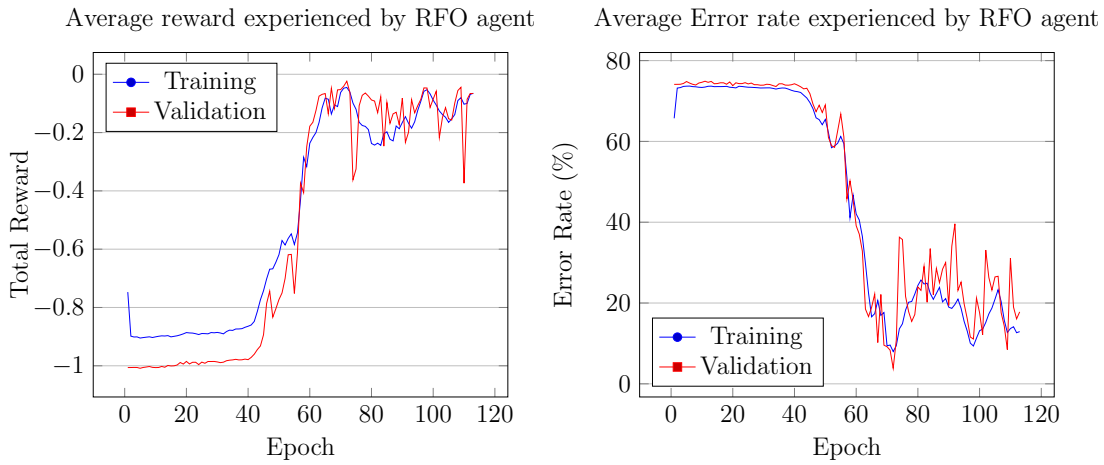


Figure 14: RFO learning behaviour
These values are not normalised as they include training values

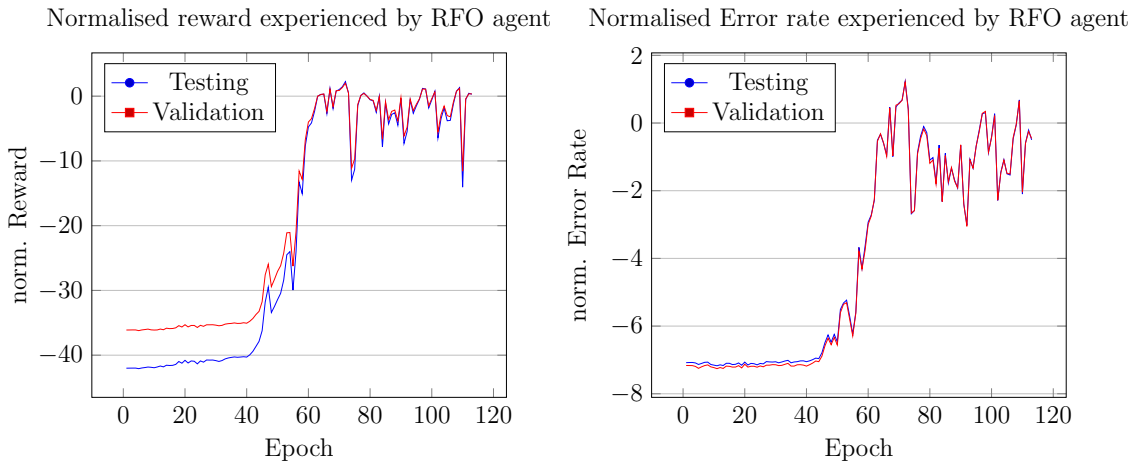


Figure 15: RFO normalised performance across learning

5.1.2 Issues and Evaluation

Two key problems seem to be hampering the behaviour of this agent. One is that it is very easy for it to get stuck in local minima, whereby the local gradient of the policy is zero, but it is not at the optimum policy. Although that can be reduced by having a larger training dataset, it cannot be avoiding within this architecture. This problem is further exacerbated by the fact that there is nothing guiding the formation of the internal state except for the reinforce, which means that potentially it isn't retaining various pieces of salient information that would allow it to make better decisions.

It should be noted that in general recurrent neural networks are particularly susceptible to local minima, as they produce chaotic responses to changes in the error surface. [1].

5.2 Apprenticed RFO

One suggestion to improve the performance, based on the work in [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.

One possible advantage of this set up for training the agent is that it makes training the agent for an expensive poorly understood task a lot easier. Instead of either 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.

For each training function the agent produced an output, and in parallel the pattern search agent produced it's 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 learned to produce something similar to the output after a reasonable number of iterations, at which point it was achieving only slightly worse scores than the pattern

$S(\boldsymbol{O}, \boldsymbol{s}_{i-1}; \theta) \mapsto \boldsymbol{s}_i$
$Q(\boldsymbol{s}; \theta) \mapsto \boldsymbol{x}$
$G(\mu, \sigma)$ ▷ 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 16: Algorithm for running and training the Recurrent Function Optimizer

search. Then when it was switched to reinforcement learning it produced a temporary drop in performance, followed by a significant improvement, exceeding that of the pure RL agent. However this result seems to be unstable, as upon further iterations it produces decaying performance. The improvement in performance of the agent after the initial batch could be attributed to it learning to explore better, as in all the cases the function used was non-convex.

In order for it to produce adequate output to match the performance of the tutor, a number of hyper parameters had to be adjusted as well, which was useful for tuning the hyper parameters for the less well defined straight RL case.

5.2.1 Apprenticed RFO results

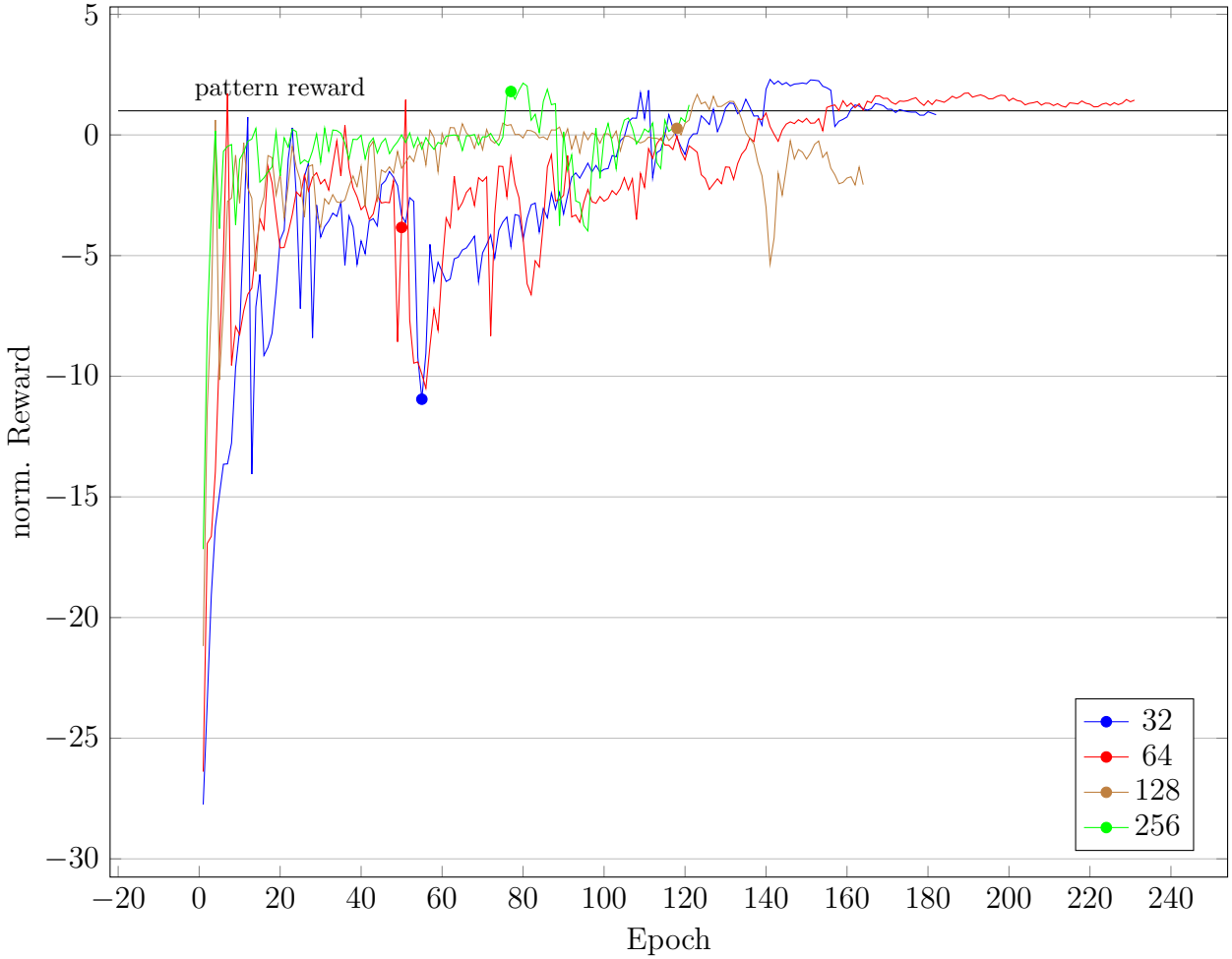
The same set of experiments was ran with the apprenticed RFO as with the pure reinforcement learned agent.

5.3 Deterministic RFO

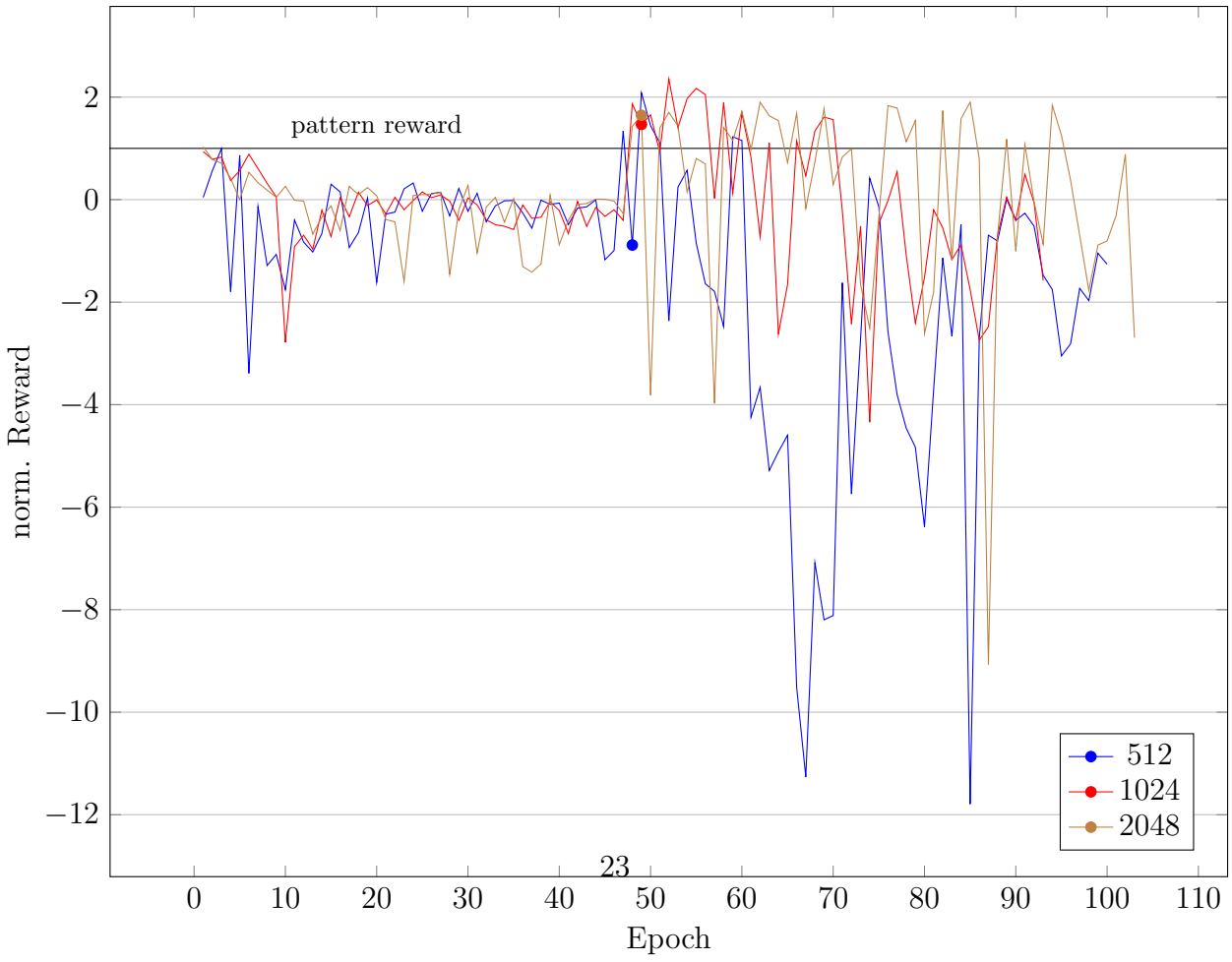
Given that the environment is both stochastic and non-adversarial, the optimal policy for the agent should be deterministic. So REINFORCE can only ever asymptotically converge to the optimal policy, and given the actual implementation, not even that. So the methods described in [3] ought to be able to 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 structure of the agent is detailed in figure 18. 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.

However, so far the agent has been very unstable, as is common for RL agents being trained using neural networks. The likely culprit is that the experience replay is insuffi-

Normalised reward experienced by Apprenticed RFO agent for different hidden sizes



Normalised reward experienced by Apprenticed RFO agent for different hidden sizes



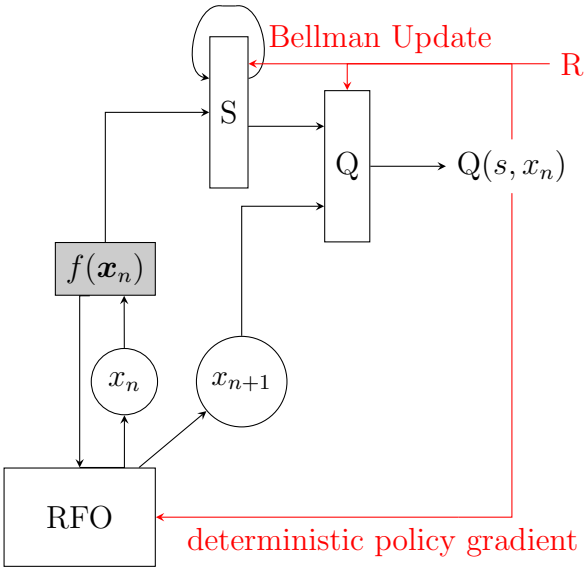


Figure 18: Deterministic RFO structure

ciently random, due to fixed batches still being used. However, it is hard to produce a computationally efficient implementation with the hardware available. It is also possible that different architectures for the critic may produce better results, but it's been a challenge to produce them and time ran out. Another possible solution could be to use the network's internal state to produce the input to the critic network, but in that case it is unclear how it would learn to produce a meaningful internal state.

6 Going forwards

There are a number of areas in which this project could be advanced. One, much like with [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 it's 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 [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.

7 Conclusions

A Explaination of Code

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

Due to the emulator being written in Java, the agent to play M:tG was also written in Java, which isn't ideal for optimisation. 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 it's experience replay.

A.2 Code for the Function Optimiser

The whole system is based on the implementation of Recurrent Visual Attention from [8]. It's 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.

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.

A.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 9. 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.

A.2.2 FunctionInput

This is an NN module which takes a reference to a function at it's 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 it's 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 it's effects are unchanged by the training.

A.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.

A.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 it's 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.

A.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 it's input if it's training, but not if it is testing.

A.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 it's elements and the updates for their parameters.

A.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 hardcoded 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 impliment 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.

A.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 it's 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 it's 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.

A.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 [3]. It expects it's 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

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \tag{17}$$

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 it's 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

$$Q(s,a;\theta) = r + \gamma Q(s,a';\theta') \tag{18}$$

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.

B 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.