Report for 4YP Project: Gaming AI

Stephen Lilico

April 1, 2016

Contents

1 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.

1.1 Context

1.2 Objectives

1.3 Report structure

2 Literature Review

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

2.1 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 hyperparameters 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:

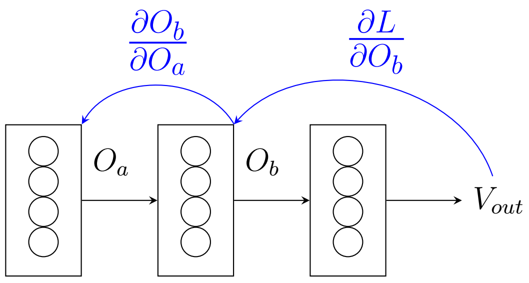
$$O_j = f\big(\sum_i (w_{ji} I_i)\big) \tag{1}$$

where $f(x)$ is the non-linear activation function - for example the sigmoid function, $\frac{1}{1+\mathrm{e}^{-x}}$. $w_i j$ are model parameters that are learned. The non-linearity allows multiple consecutive layers to add further expressivity to the function, and the choice of what non-linearity is used also significanty affects it's behaviour.

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

$$\frac{dL}{dI} = \frac{dL}{dO_b}\frac{dO_b}{dO_a}\frac{dO_a}{dI} \tag{2}$$

So the gradient can be "backpropagated" through the network by only considering the gradient (or subgradient for non-differentiable non-linearities) for the error with respect to the



$\frac{\partial O_b}{\partial O_a}$
$\frac{\partial L}{\partial O_b}$
O_a
O_b
V_{out}

Figure 1: The Backpropagation Algorithm

inputs of that layer. This then allows the gradient of the error with respect to the parameters to be easily calculated thus:

$$\frac{dL}{d\theta_b} = \frac{dL}{dO_b}\frac{dO_b}{d\theta_b} \tag{3}$$

which is easily done because $\frac{dL}{dO_b}$ is already known and $\frac{dO_b}{d\theta_b}$ is a calcuable 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:

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

$$\theta_i = \theta_i + \alpha * \delta_{i,t} \tag{4}$$

The other significant problem is one of overfitting, 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 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.

2.1.1 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 feedforwards network, the outputs are only a ever a function

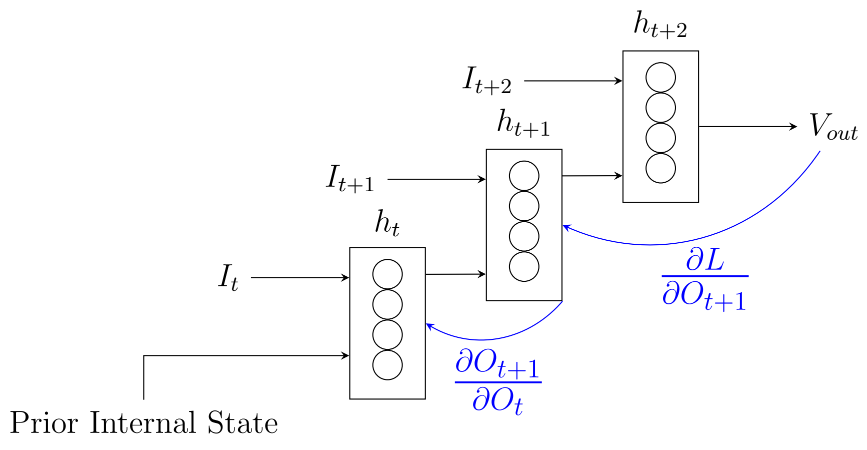

h_{t+2}
I_{t+2}
h_{t+1}
V_{out}
I_{t+1}
h_t
I_t
$\frac{\partial L}{\partial O_{t+1}}$
$\frac{\partial O_{t+1}}{\partial O_t}$
Prior Internal State

Figure 2: Backpropagation through time

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 backpropagation algorithm has to be modified to a form called "backpropagation 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:

$$\frac{\partial L}{\partial \theta} = \sum_{1<t<T} \frac{\partial L_t}{\partial \theta} \tag{5}$$

$$\frac{\partial L_t}{\partial \theta} = \sum_{1<k<t} \left(\frac{\partial L_t}{\partial x_t} \frac{\partial x_t}{\partial x_k} \frac{\partial^+ x_k}{\partial \theta}\right) \tag{6}$$

$$\frac{\partial x_t}{\partial x_k} = \sum_{t \geq i > k} \frac{\partial x_i}{\partial x_{i-1}} \tag{7}$$

where L_t is the loss at timestep 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. [**pascanu13**]

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 hyperparameters to be set so as to reduce the vanishing gradient problem.

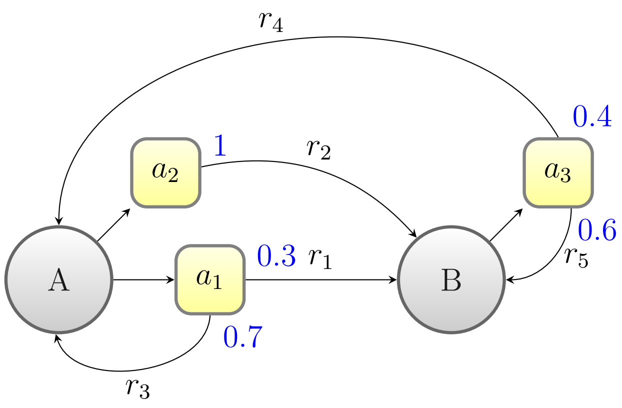


Figure 3: A markov decision process

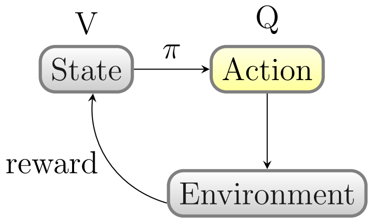


Figure 4: A simplified view of the Reinforcement learning problem

2.2 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." [**Sutton:1998:IRL:551283**] 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 recieve 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 proabilities 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 recieve 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 recieve 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 explicity 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

```
In state $s$, with available actions $\boldsymbol{a}$
with probability $\epsilon$ :
    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

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 desireable 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.

2.2.1 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.

$$V(s_n) \leftarrow r + \gamma V(s_{n+1}) \tag{8}$$

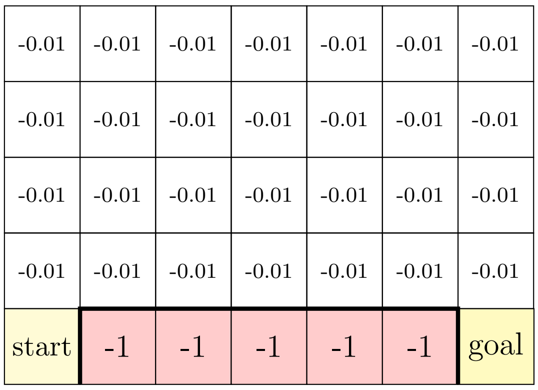
$\gamma < 1$ is a constant that discounts future rewards, so that for enviroments with unbounded epsisode length the value function remains finite. There are several metods to estimate the Q function - the two key ones are SARSA and Q-Learning.

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

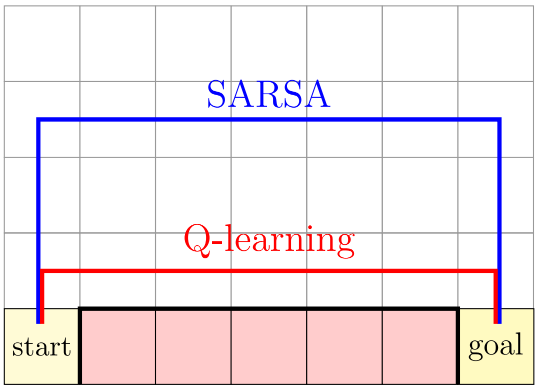
$$Q(s_n, a_n) \leftarrow r + \gamma Q(s_{n+1}, a_{n+1}) \tag{9}$$

Where a_n is the action chosen by π at step n. As this is an on-policy algorithm, π has to be sufficiently explorative. 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



(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:

$$Q(s_n, a_n) \leftarrow r + \gamma \max_a \{Q(s_{n+1}, a)\} \tag{10}$$

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 gridworld 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.

2.3 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 prohibitative. 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 expressivity. 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 expressivity. 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 desireable 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 [**dpmind atari**] 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 minibatch 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 implimentation 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 :

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

$$\theta = \theta + \alpha \frac{\partial L}{\partial \theta} \tag{11}$$

2.3.1 Policy Gradient Methods

In many applications of reinforcement learning, it is necesarry 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 numer 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:

$$J(\theta) = \mathbb{E}\big[\sum_t = 1^T r_t\big] = \mathbb{E}[R] \tag{12}$$

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

$$\nabla_\theta J = \frac{1}{M} \sum_{i=1}^{M} \sum_{t=1}^{T} \nabla_\theta \log \pi(s^i_{1:t};\theta)(R^i_t - b_t) \tag{13}$$

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 perfomance gradient depends on the policy being stochastic. Indeed, for a long time it was thought that in order to calculate the policy gradient for a deterministic policy a model of the enviroment is needed to work out the state distribution.

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

$$\nabla_\theta J = \mathbb{E}\big[\nabla_\theta \mu(s;\theta^\mu)\nabla Q^\mu(s,a)|_{a=\mu(s)}\big] \tag{14}$$

This can be implimented using an Actor-Critic method, whereby there is a separate actor and critic networks, the actor implimenting 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 [**detcontcont**] the above was combined with the insights from [**deepmindatari**] 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.

2.4 Related Work

In [**deepmindMtg**] google deeepmind look at the parsing and anlysis 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 [**Reinforcement-like-parameter**]

3 Intial Implimentations

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 [**ataripaper**] This section details the work done and results from that work.

3.1 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. Monte Carlo methods were found to be unsuitable in such an enviroment due to the fact that there are many non-terminal policies, and even if the episode is forcefully terminated, these tend to cause the agent to excessively penalise various areas, harming the learning. 1 step and TD lambda methods 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 feedforward neural network. Experience replay and assesment networks were also implimented. However, the problem proved to be more complex than anticipated. It could be observed that the agent was indeeed 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 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.

4.1 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 gameplay), 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 generalise. Other options were also considered, such as limiting the card pool to one of the standarised collections of cards used for competative 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 generalisation well, and the naı́'eve implimentation 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 necesarry 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 enourmous overheads in training time. However, the conceptual representation of the state space is not immdeiately ameniable 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 statespace. 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 unboundeded set of cards already in play, if any, they are targeting.

In order to simplify this, as well as helping the learning to generalise 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 gamestate (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 featureset 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 nerual 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

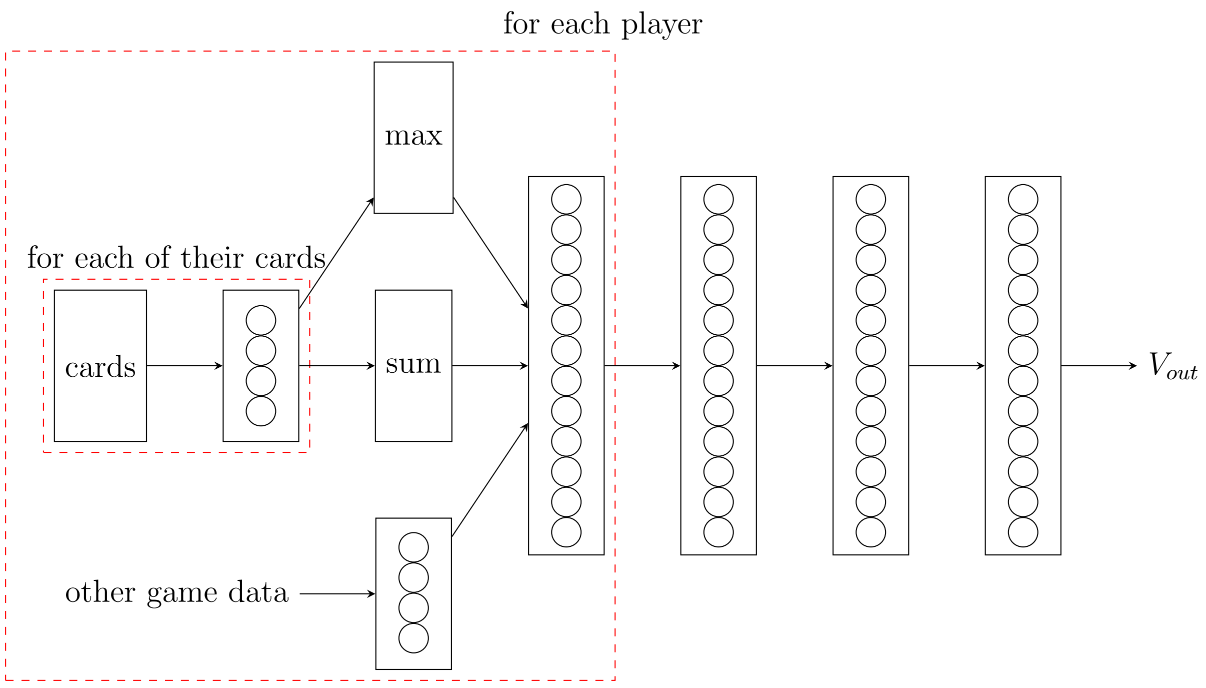


Figure 7: Architecture for the Magic: the Gathering playing agent

4.1.1 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 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,

```
$V(s;\theta) \mapsto \mathbb{R}$
**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**
        waith for ingame 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 $\theta$
        **end for**
    **until** $s_i$ is terminal
    every $c$ steps $\theta' \leftarrow \theta$
**until** Max epochs
```

Figure 8: Value iteration algorithm for MtG agent

and when tested against a human player it's actions seemed entirely random. These issues can probably 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 nerual 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 generalise the agent to be able to play any card by using and improving upon the work in [**deepmind:mtg**]. The core idea would be to use their method to parse the cardtext to some set of features that would then be used in place of the handcrafted features. Because the emulator stores additional effects given to cards on the cardtext 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.

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 com 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.

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 radomly 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 **??**.

Given the variane of these polynomials, and in particular how much the reward changes with higher orders or dimensions, it was necesarry 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 minimmum. This indicates how many were "close enough" to the target. To further normalise 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 bruteforce agent that simply divided the search space into equal blocks and looked across all of these, ignoring the values it recieved. The reward this equal search agent recieved 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.

5.1 Recurrent Function Optimisation

The first design that was attempted was based on the work in [**RVA**]. The idea is that the internal state of the recurrent neural network would be able to describe the state so that the feedforwards 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 evalution 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. One crucial difference is that, unlike with [**RVA**], 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. However, once some regularisation terms were used, it stabalised to actually start searching the space.

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 potentitally 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 stabilise to some policy relatively quickly.

One other variation that was attempted to try and improve the results, based on how the agent in [**RVA**] was trained, was to calcualate 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 explorative steps, and so producing more nuisance gradients that drove the policy away from optimality.

5.1.1 RFO Results

1d results - need to produce these

5.2 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 reucrrent neural networks are particularly susceptible to local minima, as they produce chaotic responses to changes in the error surface. [**rnns**]

5.3 Apprenticed RFO

One suggestion to improve the performance, based on the work in [**alphaGO**], 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

```
$S(\boldsymbol{O}, \boldsymbol{s}_{i-1}; \theta) \mapsto \boldsymbol{s}_i$
$Q(\boldsymbol{s}; \theta) \mapsto \boldsymbol{x}$
$G(\mu, \sigma)$                                              ▷ 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 $\theta$ in the direction of $\delta$ using backpropagation through time
until Max epochs
```

Figure 9: Algorithm for running and training the Recurren Function Optimizer

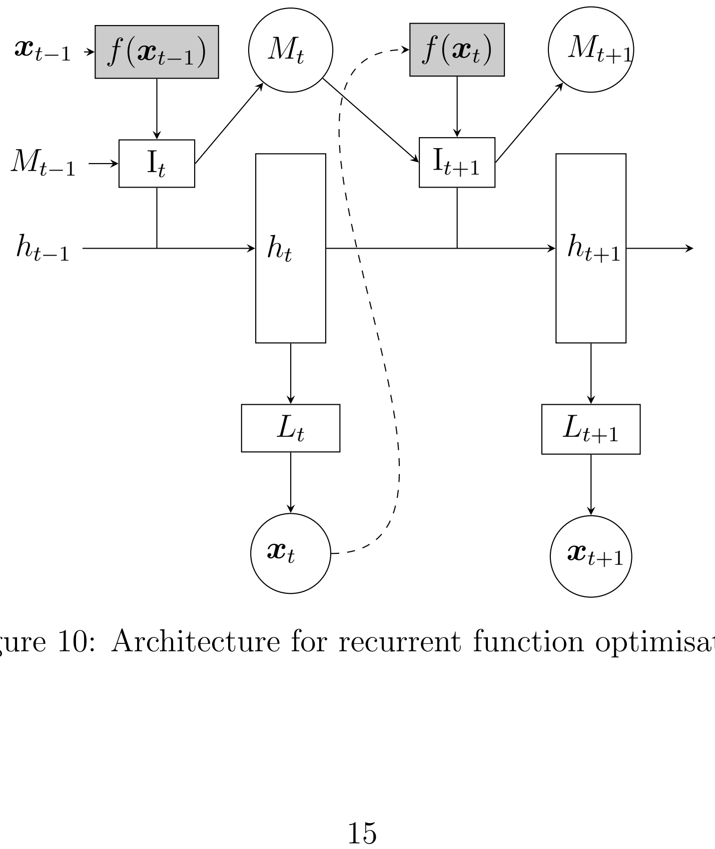


Figure 10: Architecture for recurrent function optimisation

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 implimentations of simplex agents within the systems constraints were found to be consisitently outperformed by the pattern search agent. So it was decided to use the pattern search agent instead.

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.

It proved quite difficult to actually get the agent to reproduce the output of the pattern search.

6 Conclusions

7 Explaination of Code

7.1 Code for the Function Optimiser

8 References