Report for 4YP Project: Gaming AI

Stephen Lilico

March 30, 2016

Contents

1 Introduction **2**
1.1 Context 2
1.2 Objectives 2
1.3 Report structure 2

2 Literature Review **2**
2.1 Artificial Neural Networks 2
2.1.1 Recurrent Neural Networks 2
2.2 Reinforcement Learning 2
2.2.1 Temporal Difference Methods 3
2.2.2 Policy Gradient Methods 3
2.2.3 Training ANNs with RL 3
2.3 Related Work 3

3 Intial Implimentations **3**
3.1 Maze solving agents 3

4 Magic: the Gathering **3**
4.1 Initial Setup 4
4.1.1 Results from Initial Setup 5

5 Function Optimization **5**
5.1 Recurrent Function Optimisation 7
5.1.1 RFO Results 7
5.2 Issues and Evaluation 7
5.3 Apprenticed RFO 9

6 Conclusions **9**

7 References **9**

Figure 1: The Backpropagation Algorithm

1 Introduction

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 then

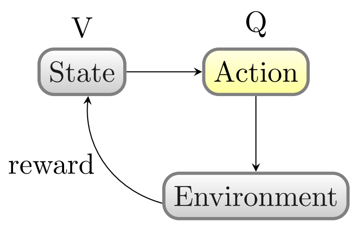
$$\frac{dL}{dI} = \frac{dL}{dO_b}\frac{dO_b}{dO_a}\frac{dO_a}{dI} \tag{2}$$

So the gradient can be "backpropogated" through the network by only considering the gradient (or subgradient for non-differentiable non-linearities) for the error wrt. the inputs of that layer. Then for each layer

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 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

Figure 2: Backpropagation through time



2.2 Reinforcement Learning

Reinforcement learning is

Reinforcement learning is "a technique where an agent attempts to maximise it's reward by repeated interactions with a complex uncertain environment." [**Sutton:1998:IRL:551283**]

A MDP is a discrete time stochastic control process.

- Value function
- Q function
  - On Policy
  - Off Policy

2.2.1 Temporal Difference Methods

- Temporal Difference methods
  - TD(0)

$$V(s_n) = V(s_n) + \alpha(\gamma V(s_{n+1}) + r - V(s_n))$$

  - TD(λ)

$$R_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)}$$

$$V(s_n) = V(s_n) + \alpha(R_t^\lambda - V(s_n))$$

2.2.2 Policy Gradient Methods

REINFORCE Updates the parameters of the function approximation directly

$$\nabla_\theta J = \frac{1}{M}\sum_{i=1}^{M}\sum_{t=1}^{T} \nabla_\theta \log\pi(a_t^i | s_{1:t}^i; \theta)(R^i - b_t)$$

2.2.3 Training ANNs with RL

2.3 Related Work

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 5/6 sets 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 naı́eve 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 1 training with the algorithm in 2

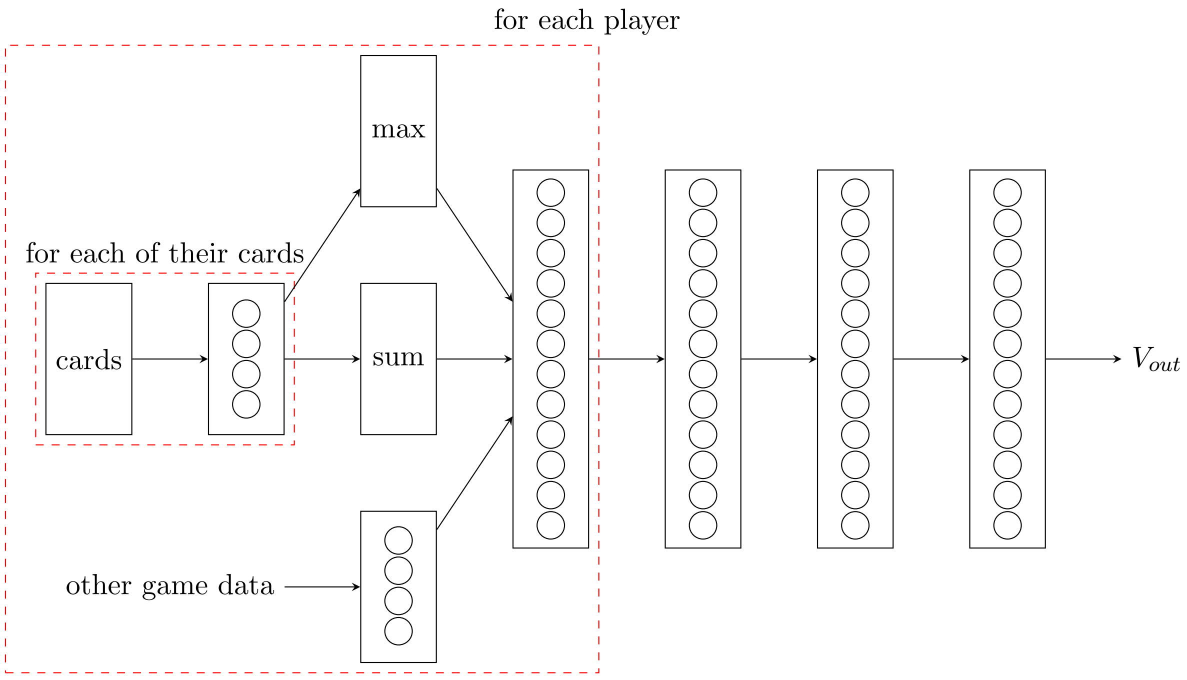


Figure 3: Architecture for the Magic: the Gathering playing agent

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

```
$V(s;\theta) \mapsto \mathbb{R}$
repeat
    Pick some initial state $s_i$
    repeat
        produce a list of actions $\mathbf{a}$ for $s_i$
        for $a$ in $\mathbf{a}$ 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 $\mathbf{a}$
        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 $\mathbf{B}$ from $Replay$
        for $s_b, s_{b+1}, r_b$ in $\mathbf{B}$ 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 4: Value iteration algorithm for MtG agent

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, 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.

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.

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 4. 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 - $Loss(f(\boldsymbol{x}), f(\boldsymbol{x}_{min}) = log(f(\boldsymbol{x}) - f(\boldsymbol{x}_{min} + 1)$

```
$S(\boldsymbol{O}, \boldsymbol{s}_{i-1}; \theta) \mapsto \boldsymbol{s}_i$
$Q(\boldsymbol{s}; \theta) \mapsto \boldsymbol{x}$
$G(\mu, \sigma)$                                         ▷ 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 5: Algorithm for running and training the Recurren Function Optimizer

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**]

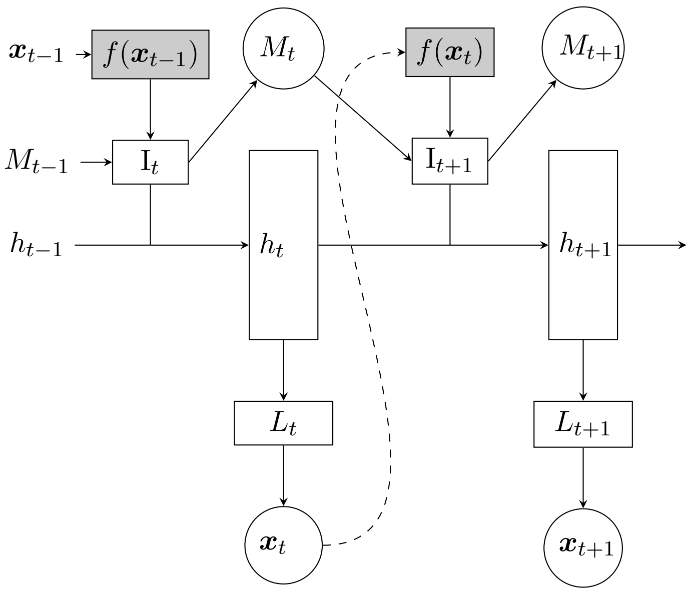


Figure 6: Architecture for recurrent function optimisation

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 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 References