Technical Milestone Report

Alex Darch
Supervisor: Dr Glenn Vinnicombe

January 17, 2019

Summary

Technical Summary

1 Introduction

AlphaGo Zero is a revolutionary Reinforcement Learning algorithm that achieved superhuman performance in the game of go, winning 100–0 against the previously published, champion-defeating AlphaGo. It's successor, AlphaZero, is a generalised version that can achieve superhuman performance in many games. There are three key sub-algorithms that form the basis of their success: self-play, a convolutional neural network (CNN) and a Monte-Carlo tree search (MCTS).

The aim of both is to learn a policy (train a CNN) that can accurately predict an action-probability and a value function. The action-probability, $p(\boldsymbol{a}|\boldsymbol{s})$, is a $p.m.f.$ over the valid actions given a state, where the probabilities are proportional to the expected outcome for each action, and the value function is the expected outcome of the game (e.g. +1 is a win with 100% certainty). It achieves through self-play: picking the moves that it thinks will do best, whilst inconveniencing the opponent the most (which is itself). Moves/actions are sampled from a MCTS-informed action-probability, which provides a probabilistic balance between exploration and exploitation.

A key feature of AlphaZero is that it only requires the ability to simulate the environment. It does not need to be told how to win, nor does it need an exact model of the system dynamics, $p(s_{t+1}, return_{t+1}|s_t, a_t)$, as this can be learnt through self-play. Furthermore, the algorithm often 'discovers' novel solutions to problems, as shown by 'move 37' in a game of Go against the reigning world champion, Lee Sedol.

AlphaZero differs from AlphaGo Zero, other than not taking advantage of the symmetries of go, by using a different method of policy improvement. For both, the weights and biases are randomly initialized (so there should be a 50% chance of winning). Then a batch of games/episodes are played via self-play. With AlphaGo Zero, the self-play games are generated by the best player from all previous iterations and then the new player was compared with the best player; whereas AlphaZero just maintains a single neural network

that is updated continually: self-play games are generated by using the latest parameters of the neural network. [?]

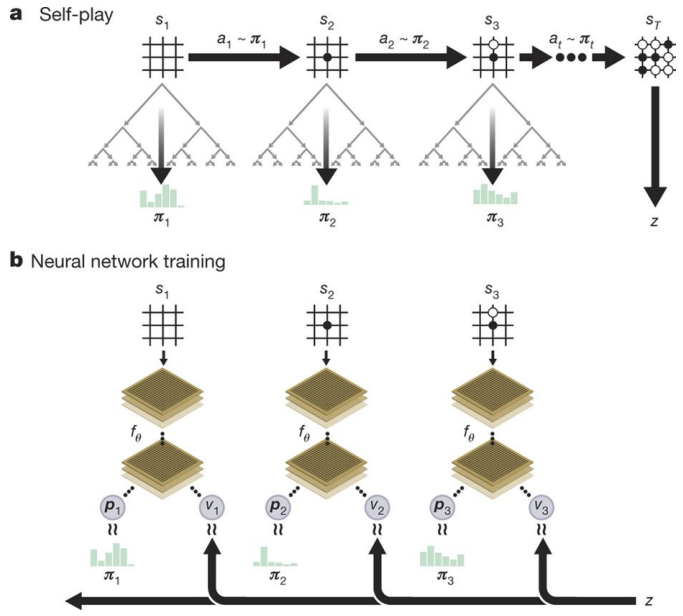

Figure 1: A schematic showing how self-play and policy training are performed. Taken from [?].

Figure ??a shows how self-play is performed in AlphaGo Zero. An episode/game, $\{s_1, ..., s_T\}$ is played against itself. For each state, s_t, a Monte-Carlo Tree Search is performed, guided by the current policy f_θ. The MCTS outputs an improved action-probability, $\boldsymbol{\pi}_t = [\hat{p}(a_1|s_t), \hat{p}(a_2|s_t), ..., \hat{p}(a_N|s_t)]$. The next move is then selected by sampling from $\boldsymbol{\pi}_t$. The final player of the game is then given a score of +1 or -1 if they win or lose respectively. The game score, z, is then appended to each state-action pair depending on who the current player was on that move to give: $[s_t, \boldsymbol{\pi}_t, z]$.

Figure ??b depicts how the policy is trained. The state, s_t, is taken as input and is passed through a neural network with parameters θ. The network outputs an action-probability, $\boldsymbol{p}_t$, and a state value - the estimated probability of the current player winning the game. The neural network is trained to more closely match the MCTS-informed action-probabilities $\boldsymbol{\pi}_t$, and the predicted winner (state-value), v_t to the actual winner, z.

The loss function for the neural network is given by:

$$\mathcal{L} = (z - v)^2 - \boldsymbol{\pi} \cdot log(\boldsymbol{p}) + c||\theta||^2 \tag{1}$$

Figure ??a: A MCTS is performed at each step of an episode. The state at which the tree search starts then becomes the root state, s_{root}. From the root state, the tree search can move to an edge (s, a) by selecting an action, a. Each edge stores a prior action-probability output by the policy, p(s, a); a visit count, N(s, a); and an action-value, Q(s, a), which is the value of the state that action a will result in. Actions are selected by maximising an the action-value plus an upper confidence bound:

$$a_{selected} = \underset{\forall a}{argmax}\Big\{Q(s,a) + c \cdot \frac{p(s,a)}{1 + N(s,a)}\Big\} \tag{2}$$

Where c is a constant of magnitude ~ 1.

Figure ??b: Once a leaf node (N(s, a) = 0) is reached, the neural network is evaluated at that state: $f_\theta(s) = (p(s, \cdot), V(s))$. The action-probability and state-value are stored for the leaf state.

Figure ??c: The action-values, Q, are calculated as the mean of state-values in the subtree below that action. The state-value are then calculated as the mean of all the action-values branching from that state, and so on.

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{s_{t+1}|s_t,a_t} v(s_{t+1}) \tag{3}$$

Figure ??d: After a set number of searches, the MCTS-improved action-probabilities $\boldsymbol{\pi} = p(\boldsymbol{a}|s_{root})$ are returned, $\propto N^{1/\tau}$, where N is the visit count of each move from the root state and τ controls the sparseness of the probability mass function ($\{\tau = 0\} \rightarrow argmax\{N(s,a)\}$).

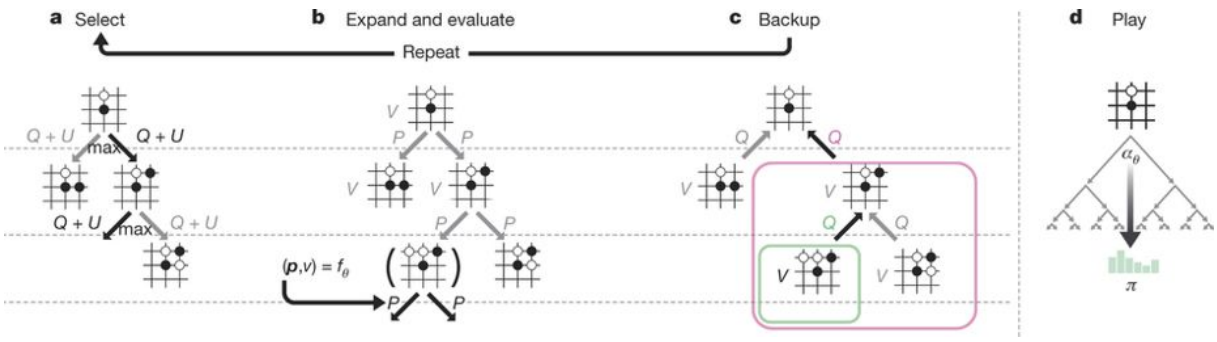

Figure 2: A schematic outlining the steps involved in a monte-carlo tree search. The figure is taken from [**?**].

Control theory is the study of how to get optimal corrective behaviour for dynamical systems. Reinforcement learning has recently found success in control problems such as Atari-games and robotics [**?**]. A common theme with control problems is that they are easy to simulate, but hard to find the dynamics for. Therefore, adapting AlphaZero to control problems an exciting extension to an already powerful AI.

There are a few problems in adapting AlphaZero to a control problem, namely:

- How do we represent a continuous state as a 2D state?
- How do we change the value function to reflect a non-stationary and continuous problem? We cannot use whether we win or lose as the value for the end of the game.
- How do we implement self-play for problems that already have non-intelligent adversaries such as gravity?
- How do we compare policies when one policy has an advantage such as gravity?

The purpose of this project is to try to find an answer to these three questions and therefore show that AlphaZero can be used as a powerful general-purpose control algorithm.

2 Method

Currently, a 2D state representation and a value function have been chosen and implemented, an adversary other than gravity has not been implemented yet.

2.1 OpenAI's Gym Environment

OpenAI's gym is a toolkit for developing reinforcement learning algorithms. They provide environments with common interfaces to allow the writing of general algorithms. Initially, the CartPole environment is being used as it is the simplest classical control problem, see figure **??**.

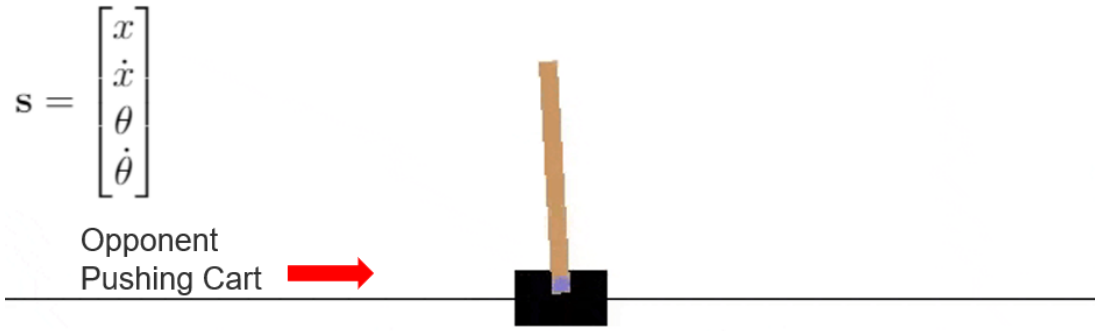

Figure 3: The OpenAI gym CartPole environment. The classical state representation is shown in the top left. Actions by the player and the adversary are taken as an impulse to the left or right.

The cart is defined as having fallen over, or 'done', if the angle from upright exceeds $12^o(\theta_{max})$ or the cart wanders $> 2.4(x_{max})$ units from the origin. We also define steps_beyonds_done as the number of steps that we continue to calculate states for after it has fallen over. For each step/impulse, the 2D state is calculated and a state_loss is calculated as:

$$\mathcal{L} = -\frac{1}{2}\left[\left(\frac{x_t}{x_{max}}\right)^2 + \left(\frac{\theta_t}{\theta_{max}}\right)^2\right] \tag{4}$$

Thus ensuring that $0 \geq \mathcal{L} \geq -1$.

2.2 2D State Representation

The state is a histogrammed and discounted function of previous state positions and angles, i.e. $NewState = Binned\ Current\ Position + \gamma * PreviousState$, where γ is a discounting factor, currently set at 0.7. The numpy library in python has a function histogram2d which allows the binning of two-dimensional arrays.

A 2D representation like this allows us to use a convolutional neural network,which has the benefit of various transformation invariances - these are particularly useful for CartPole since it is highly symmetric.

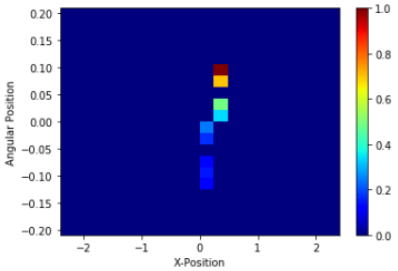

Figure 4: An example of a 2D state representation where there are 20 bins and 17 random actions have been taken.

2.3 The Value Function

A value function is computed after an episode has completed as the discounted future losses at each state with

the constraint that $\gamma^k < \frac{1}{20}$, where $\frac{1}{20}$ was chosen as it is a standard factor for insignificance. Since steps_beyonds_done (= k) must be defined in the CartPoleWrapper class, this is a constant, and therefore γ is calculated as $\gamma < \frac{1}{20}^{\frac{1}{k}}$. Algorithm **??** shows the procedure for calculating this. The discounted value can be written as:

$$v_t = \frac{\sum_{t'=t,\tau=0}^{t+k,k} \gamma^\tau \mathcal{L}_{t'}}{\sum_\tau^k \gamma_\tau}, \qquad \text{where } \gamma^k < \frac{1}{20} \tag{5}$$

Algorithm 1 Calculate State Values

```
1:  procedure VALUE_FUNCTION(state_losses)
2:      Constants: γ, k                          ▷ discount factor, steps_beyonds_done
3:      Initialise: values ← [ ]
4:      for step_idx := 0 to (length(state_losses) - k) do
5:          state_value ← 0
6:          discount ← 1
7:          for idx := step_idx+1 to (step_idx+k) do
8:              value ← value + discount×state_losses[idx]
9:              discount ← γ×discount
10:         values.append(value/∑γ)
11:     return values                  ▷ values is k elements shorter than state_losses
```

2.4 Program Structure

An outline of the program structure is shown in figure **??**. The green boxes are implemented as described in [**?**] in the introduction with the exceptions that the neural network predicts the value as defined above, not the expected outcome, and episodes are currently compared simply as $mean$(challenger losses) $> 0.95 * mean$(currrent losses).

3 Results

- Graphs of statistics for each policy iteration, eg, errors over time for each nnet
- Performance against random and a greedy algorithm (make the moves that move x pos closer to 0)
- Graphs of steps taken vs expected steps left
- Graphs of return vs time
- If I have time, ill compare different adversaries: vs greedy, vs itself with 1/2 power etc

4 Discussion and Future Work

- Explain the graphs above

- Talk about problems?

- Which adversary was the best

- Which method learnt the quickest

- Talk about which representation of 2D state is the best? I havent done any of the others...

A major decision with using AlphaZero for control problems is how to bin the state both for the neural network and for the monte-carlo tree search, with the latter being more important. For the neural network, because we are using a convolutional neural network, the output is largely invariant to the bin size. Whereas for the MCTS if the bins are too large then there is not enough precision so as to differentiate between different states, and if the bins are too small then no two states will be the same - hence the state counts,Ns, will all be 1 and the MCTS-improved action probability will effectively be uniform. One solution to this problem is to not use uniform binning (for both the state in the MCTS and the 2D representation for the neural net). It is more desirable to know the position closer to the zero point: if the pole is almost over then the move it right it is more obvious. Therefore a binning system that has smaller bin widths towards the zero point, e.g. using a normal distribution to represent bin density could be used.

The first novel task to complete moving forwards is to implement an intelligent adversary. For the CartPole problem, it can't be an equal adversary since we are already fighting against gravity. Some possible adversaries are:

- Alternate steps and only give the adversary 0.5F for each push.

- The adversary only plays every 3 steps.

- A more 'environmental' adversary such as changing gravity or damping.

As a starting point, the adversary will act in the same plane as the force and act ever n steps (not necessarily 3 since this may be too strong).

One of the difficulties of self-play with a control problem is determining how 'good' the policy is compared to other policies. Currently, the AlphaGo Zero style of policy iteration is being used (i.e. comparing a challenger policy with the current best), however, by adopting the AlphaZero strategy of continually updating a single neural network it is difficult to know how to make the adversary roughly equal when it has the advantage of gravity. One possibility is to use a handicapped adversary and measure the performance of itself against itself. However, if the CartPole stays up for longer, is the neural network then better because it is working to keep it up better, or worse because the adversary is less effective? Further thought on this is needed, and moving forward an AlphaGo Zero style of policy iteration will be used.

Once a fully working algorithm has been made for the CartPole, a new, more complex environment will be implemented to improve test and improve the generality of the program. A good next step is to increase the dimensionality of the state space, for example by adding a spring between the players and the base of the cart.

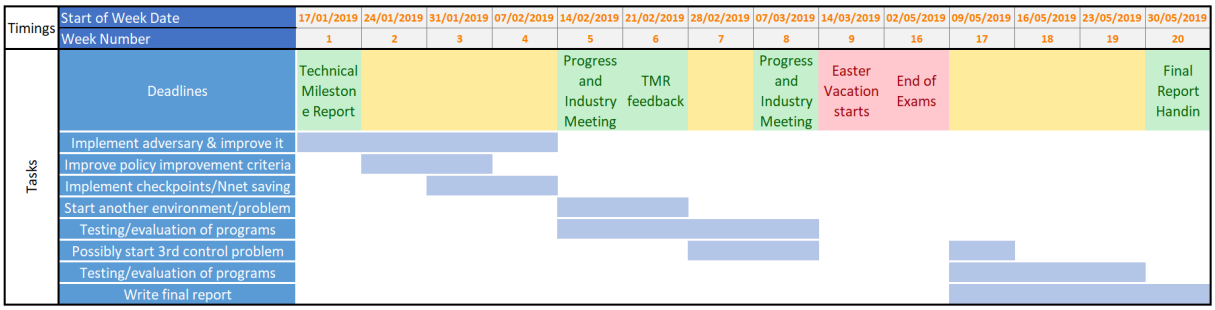

Figure 5: A Gantt Chart of the proposed timeline for lent and easter term.

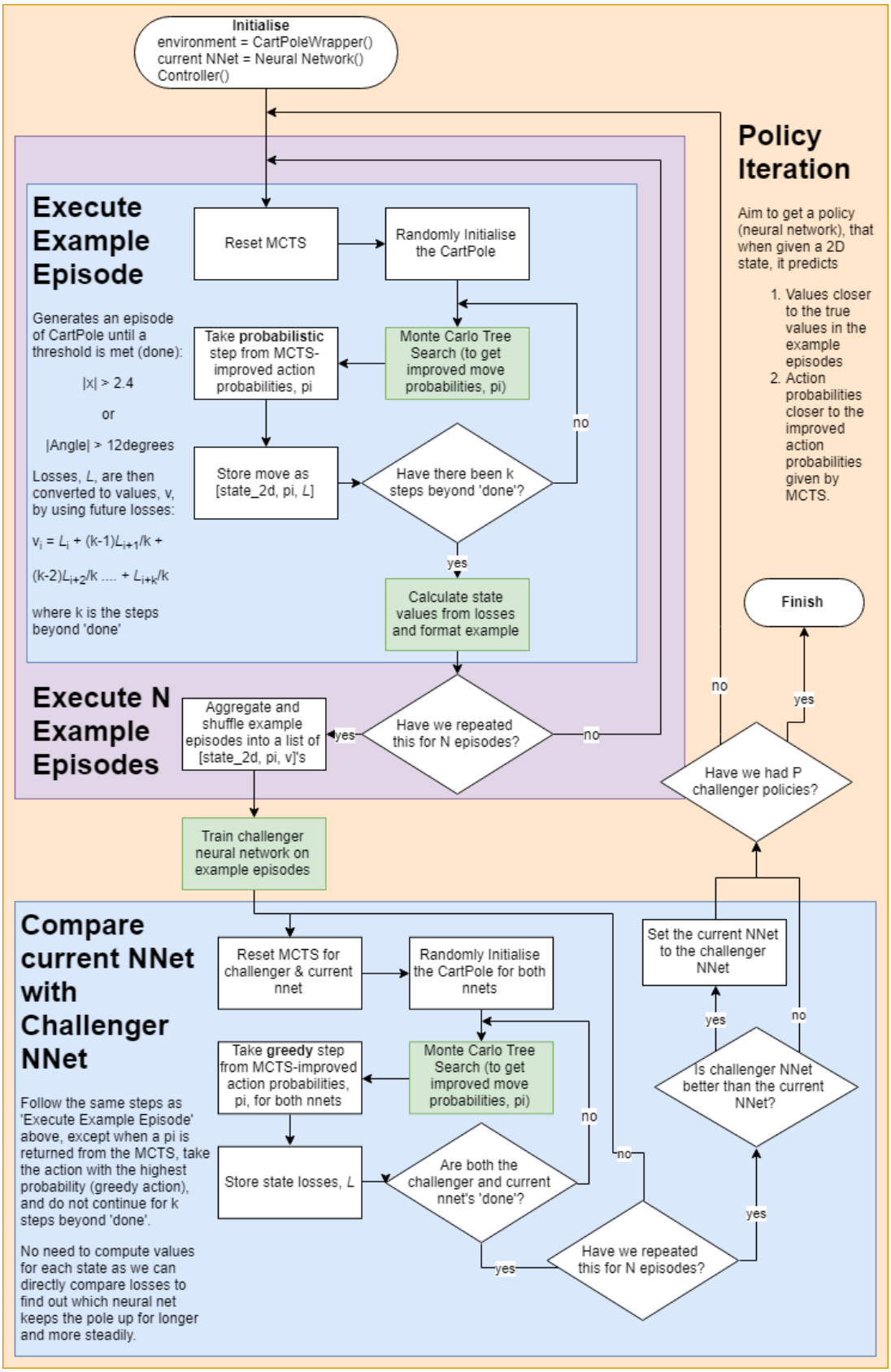

Figure 6: A flowchart showing the control flow of the current program. Green boxes depict more complex processes with further explanation in the main text.