Portland State University

PDXScholar

Dissertations and Theses

Dissertations and Theses

11-2-2021

From MDP to AlphaZero

David Robert SeWell
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the [Applied Mathematics Commons](#), and the [Computer Sciences Commons](#)

Let us know how access to this document benefits you.

Recommended Citation

SeWell, David Robert, "From MDP to AlphaZero" (2021). *Dissertations and Theses*. Paper 5867.
<https://doi.org/10.15760/etd.7738>

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

From MDP to AlphaZero

by

David Robert SeWell

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Systems Science

Thesis Committee:
Bruno Jedynak
Anthony Rhodes
Joe Fusion

Portland State University
2021

Abstract

In this paper I will explain the AlphaGo family of algorithms starting from first principles and requiring little previous knowledge from the reader. The focus will be upon one of the more recent versions AlphaZero [Sil17b] but I hope to explain the core principles that allowed these algorithms to be so successful. I will generally refer to AlphaZero as theses core set of principles and will make it clear when I am referring to a specific algorithm of the AlphaGo family. AlphaZero in short combines Monte Carlo Tree Search (MCTS) with Deep learning and self-play. We will see how these three concepts fit together and we will break down each of these pieces and look at examples to clarify understanding. I implemented a simplified version of the algorithm on TicTacToe and Connect4 and the code is available online as well as a simple web app that allows you to play against a trained agent.

<https://github.com/befeltingu/VikingZero>

<https://github.com/befeltingu/VikingDashboard>

Acknowledgements

Thanks to Bruno Jedynak for all the feedback and discussion

Contents

Abstract	i
Acknowledgements	ii
List of Figures	iv
1 Introduction	1
2 Reinforcement Learning	3
2.1 RL Intro	3
2.1.1 Environment	4
2.1.2 State	5
2.1.3 Policy	6
2.1.4 Reward Signal	6
2.1.5 Value	6
2.1.6 Model	7
2.2 Markov Decision Processes	7
2.3 Tictactoe Bellman Equation	14
2.4 Policy evaluation Example	16
2.5 Dynamic Programming	21
2.6 Monte Carlo Methods	24
2.7 TD learning	27
2.8 Function approximation and Deep RL	31

2.9	DQN	34
3	Search	40
3.1	Planning	40
3.2	MCTS	42
3.2.1	Example 1	46
3.3	Multi Agent RL	49
3.4	Self-Play	50
3.5	MiniMax and Alpha Beta Pruning	51
4	AlphaZero	56
4.1	Expert Data	57
4.2	Zero Knowledge	61
5	Experiments	67
5.0.1	TicTacToe Experiments	70
5.0.2	TicTacToe Experiments	75
5.0.3	Connect4 Experiments	83
6	Other Work	85
6.1	MuZero	85
6.2	MCTS as regularized policy optimization	88
6.3	Other work and applications AlphaZero	91
7	Conclusion	93
	Bibliography	93

List of Figures

2.1	RL	3
2.2	Concept Hierarchy	4
2.3	RL	13
2.4	Tictactoe Bellman 1	15
2.5	Ticatactoe Bellman 2	16
2.6	policy eval 1	17
2.7	policy eval 2	18
2.8	policy eval 3	19
2.9	From Sutton Chpt 4.3	23
2.10	First-visit MC prediction	25
2.11	Monte Carlo ES (Exploring Starts)	27
2.12	Tabular TD Sutton Chpt 6	29
2.13	Sarsa: Sutton Chpt 6	30
2.14	Q learning: Sutton Chpt 6	31
2.15	TicTacToe Board	33
2.16	DQN architecture	36
2.17	The DQN algorithm	37
3.1	MCTS: Sutton Figure 8.10	43
3.2	two player policy	51
4.1	Mini game of go	57
4.2	Reduce depth	57

4.3	Reduce breadth	58
4.4	AlphaZero training pipeline	62
4.5	AlphaZero training pipeline	64
5.1	Left Figure is a tictactoe board. Figure on the right is the transformed version	72
5.2	Base Case experiment	77
5.3	Base vs $\tau = 1$	79
5.4	Base vs $\epsilon = 0.2$	80
5.5	Base vs $n_{sim} = 15$	81
5.6	Base Case vs maxmem size	82
5.7	Base Case vs Epochs =50	83
5.8	Base Case vs Epochs =50	84
6.1	Planning in MuZero	86
6.2	MuZero at a Glance	88

1.0 Introduction

Games are a common activity of humans. They exist and have existed in every known society in various forms. The nice thing about games from a computer science or game theory perspective is that they provide a well defined decision making environment. This allows scientists to apply mathematical tools to these toy environments in order to test various hypothesis that would otherwise be intractable in more complex environments. The hope is of course that we can learn some sort of general principles from experimenting in these game setting that we can then apply in the real world. One of the interesting things about the history of AlphaZero is that there is a trend toward simplicity and generalization. That is that the algorithms themselves require less and less input from a human in order to be able to learn in more and more settings. Earlier versions are also more complex in their implementations and so a focus on AlphaZero I think helps to see the core principles at work.

It is interesting to look at the history of AlphaZero a bit. In 2014 Deepmind a UK based artificial intelligence company began their AlphaGo project. The goal was to see how deep learning could be utilized to beat the the game of Go. This was a tall order and up to that point previous algorithms had only managed to play at an amateur level. In 2015 the first version of ALphaGo beats Fan Hui the European Go champion at the time. The first article describing their methodology was released in 2016 and they dubbed the name of this AlphaGo version AlphaGo Fan. AlphaGo Lee was the next version to be released. With only a few tweaks to AlphaGo Fan they were able to defeat the world champion Lee Sedol 4-1. The next version in 2017 AlphaGo Master beat top professionals 60-0. AlphaGo Zero was then released in late

2017 and was trained without any human or expert data. AlphaZero is published in 2018 in which some of the Go specific constraints are dropped. AlphaZero successfully dominates Chess, Go and Shogi (Japanese Chess) in that release. In 2019 a paper entitled "Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model" is released. This version is even more general than previous versions and manages to beat benchmarks in Atari as well. We will briefly discuss the relevance of MuZero but will not go into detail here. In 2020 Jean-Bastien Grill and colleagues published "Monte-Carlo tree search as regularized policy optimization". This paper has some interesting theoretic claims in regards to AlphaGo and to MCTS in general and we will briefly discuss this paper and its relevance. Some very interesting applications have also been produced as a result of this research. AlphaFold in 2018 and AlphaFold2 in 2020 produced state of the art prediction on the protein folding problem. This allows AlphaFold to predict the structure of a protein just from its sequence of amino acids. This particular breakthrough from Deepmind has some really exciting implications for Medicine, computational biology, and just for our understanding of biology in general.

2.0 Reinforcement Learning

2.1 RL Intro

This paper will attempt to understand AlphaZero from a Reinforcement Learning (RL) perspective. A bit of background on RL will help clarify some of the commonly used vocabulary we will need to describe AlphaGo. RL is a branch of machine learning in which an agent attempts to maximize reward by taking actions in an environment. In a typical machine learning setting you are usually given a set of data and a set of labels and you want to be able correctly guess the labels from the given data. This is called supervised learning and is the type of machine learning that most people have seen. In RL instead of being given data we typically generate our own data by interacting with an environment. This process I just described is shown in Figure 1.

Figure 2.1: RL

”Reinforcement learning is learning what to do and how to map situations to actions so as to maximize a numerical reward signal” Sutton (pg 1)

A full discussion of reinforcement learning is not required to understand AlphaZero but it is necessary to look at the fundamental concepts that make up the algorithm. I present a sort hierarchy of concepts that are necessary to understand AlphaZero fully. It is possible to read through the paper or a tutorial to get the mechanics of the algorithm but to understand the "why" you will need to go deeper.

Figure 2.2: Concept Hierarchy

2.1 Environment

Loosely defined as the thing that the agent interacts with. Everything that is exterior to the agent. The environment typically encompasses the system that is trying to be solved. It also provides observations and rewards as the agent takes actions. For example when trying to solve the game of Go the agent (the players)

takes actions (moves pieces) and the environment is the board, the pieces, and the mechanism for telling the agent the current reward (win / loss) or other special states that are specific to the game. So it is the game itself plus a few other mechanisms to make the learning process possible. Occasionally in problems like robotics and similar problems the agents own physical body can be part of the environment. The agent will need to control its own body to achieve a task and so it will usually model its body as part of the environment in this case. The environment is not synonymous with state as observed by the agent. We will look at that next.

2.1 State

State is the representation of all information that is available to the agent at any given time. As said above this is not necessarily the same as the observation provided by the environment. For instance in AlphaGo Zero we will see that the state is actually a function of the current board as well as previous board positions. It is up to the designer of the reinforcement learning algorithm as to what is the best state to present to an agent. In many environments the choice might be more obvious than others. The fact that state is something that needs to be decided upon by the designer of the algorithm hints at a potential need for more generalization within algorithms. MuZero and similar algorithms take a step in this direction by having the agent learn that representation. MuZero as will be discussed towards the end of this paper searches in a latent space. So the state as provided by the environment is projected onto a latent space and then the algorithm searches within this space. This is an interesting approach. For further discussion on this see [\[Sil17a; Sch19\]](#)

2.1 Policy

A policy roughly speaking is just a definition of an agents behavior given a perceived state of the environment. That is it is a mapping $\pi : s \rightarrow a$ where s is a state of the environment and a is an action $a \in A$. A is the possible set of actions the agent can take. It can be either discrete or continuous. In AlphaGo Zero actions are integers representing the place on the board.

2.1 Reward Signal

A reward signal is what defines the goal of a RL algorithm. As an agent observes an environment it receives feedback in the form of rewards. Generally an agent is simply trying to maximize the sum of its reward over the long run. Rewards help the agent to learn a policy. An agent will adjust the actions that it takes based off of the feedback it receives. Roughly speaking the agent will adjust to take actions more frequently that it perceives to return more reward. In environments like AlphGo Zero there is what is called the "sparse rewards" problem. This means that the agent might be interacting with the environment for long extended periods of time before receiving any feedback.

2.1 Value

The value of a state in reinforcement learning is the amount of reward an agent should expect to receive in the long run from a given state. An agent is not given this number and is something that needs to be learned. This is generally expressed like this $V(s) \rightarrow \mathbb{R}$.

From Sutton $v_{\pi}(s) = E_{\pi}[G_t | S_t = s]$

2.1 Model

A model from the perspective of RL typically refers to a predictive model of the environment. It is a way for the agent to predict the behavior of the environment. This allows the agent to make inferences about what will happen when certain actions are taken. This in turn allows the agent to plan based on the model. Algorithms that use models are called *model-based*. Algorithms that don't explicitly model the environment are called *model-free*. The line between the two can easily become blurred as a lot of algorithms will mix between the two approaches. In the context of this paper AlphaZero can be viewed as a Model free algorithm and MuZero as a model based extension to AlphaZero. MuZero attempts to learn the dynamics of the environment from scratch whereas for AlphaZero the dynamics are known upfront.

2.2 Markov Decision Processes

The following section draws pretty heavily from Sutton [Sut18]. Let's put some of the previous concepts together into a framework that we can actually use to make decisions. Markov decision processes (MDP) are an idealized environment for making sequential decisions. MDPs have been studied quite thoroughly and although their formulation is simple it is also quite robust and a surprisingly large number of problems can be formulated as MDPs. Figure 2.1 that we looked at before is an example of a MDP. What kind of data would we get if we allowed the process in that figure to go on for many time steps? First the agent begins in the start state S_0 and takes action A_0 and receives a reward R_1 and the next state S_1 . Then the agent would take another action A_1 and so on. This would result in a series of data points $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, \dots, R_T, S_T$. Here there is some terminating condition at time T but the series could also just go on continuously. R_t, S_t are both random

variables whose probability of occurrence only depends on the previous action and reward. This is what makes the process markovian and is called the Markov property. The only thing that matters in determining the probability of the next state and reward is the current state and action. The dynamics of this process can be described by a function $p : SxRxAxS \rightarrow [0, 1]$. The function can be defined in terms of conditional probability.

$$p(s', r | s, a) = Pr\{S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a\}$$

So the probability given by p describes the entire environment dynamics. The Markov property as described in [Sut18] ”is best viewed as a restriction not on the decision process, but on the state. The state must include information about all aspects of the past agent–environment interaction that make a difference for the future”.

How state, action and time is defined is quite flexible in this formation. Time does not have to refer to an actual interval of time but can just be some set of steps that happen one after the other. Actions likewise can really be any decision that we want to learn how to make.

Now that we have defined the system what is the actual goal? We want to get as much reward as possible. That means not just the reward that we receive right now but all the reward we accumulate. This statement results in the *reward hypothesis* Introduction to reinforcement learning

“..what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward)”.

Since learning from rewards has become so popular it might not seem like that strange of a concept and indeed it might be fairly intuitive. However this is one of the things that distinguishes reinforcement learning from other forms of machine learning. In a typical supervised learning setting for instance you attempt to minimize a loss function on a set of observed data w.r.t to some given labels. Like attempting to predict credit card fraud from a set of features. You are first given a bunch of examples with labels as to whether fraud occurred in that particular instance. You then want to train a model to be able to generalize from these observed features and labels to unlabeled or data that has not been seen yet. Reinforcement learning is different in that we don't generally start out with a set of features and labels. The agent produces data or observation as we act in the environment. Then through these observations we change the agent's behavior as to achieve a desired goal. There is a bit of an art in setting up the rewards correctly. The designer of the algorithm has to choose the rewards for an environment and it is not always trivial. For GO and other two player zero sum games it is natural to assign a reward of +1 for winning, -1 for losing and 0 for a draw. You might however want to assign a small amount of reward to a draw so that the agent is receiving some amount of feedback. You might want to set up intermediate goals as to "show" the agent the right choices to make or steer them away from obviously bad choices. An example of a miss application of reward assignment would be in chess to give reward to an agent for pieces captured. This can lead the agent to commit detrimental errors to the main goal of winning just to achieve the sub goal so care must be taken that the subgoals align with the main goal.

In this paper we will be focused on situations in which there is a clear start and

end state. There are of course other domains in which this is not the case. Self-driving cars, trading in markets, and playing atari games are a few examples. Since we are trying to maximize an accumulated reward we can define our main goal as the following.

$$G_t = \sum_{k=t+1}^{k=T} R_k \tag{2.1}$$

There is something still not quite right here. In this formulation we would value rewards at all time steps equally. This is usually not the case in most tasks however. The idea that 100\$ today is better than 100\$ 10 years from now applies here. Typically the reward we get now is more valuable then the same amount in the future. So we can modify our goal a bit by discounting rewards that we receive farther into the future. We do this by using a discount factor $\gamma < 1$.

$$G_t = \sum_{k=t+1}^{k=T} \gamma^{k-t+1} R_k \tag{2.2}$$

Ok, great we have a goal now how do we use that goal to start defining how the agent is going to behave? Well we want the agent to take actions that are going to lead to higher G . Taking actions in our MDP formulation leads us to end up in different future states. So we want to end up in better states. What does it mean to be a better state? A better state is one that leads to more G . So we want to be in states in which we expect to collect more reward in the future. Lets try and put this into something concrete. The expected value of a state seems to fit well into what we are going after so we can use that as a starting point.

$$\mathbf{v}_\pi(s) \doteq \mathbb{E}_\pi[G_t | S_t = s] \quad (2.3)$$

The π here refers to the policy of the agent. So this is looking at the expected value assuming the agent is following a given policy. We discussed policies previously and we will discuss them more in the future. It is important for knowing the expected value while following an MDP because a policy will define how we transition from one state to the next. We can also define a value function for a state action pair in a similar manner.

$$\mathbf{q}_\pi(s, a) \doteq \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \quad (2.4)$$

Its quite easy to see that if one actually had access to one of these functions how you might have the agent behave. In the case of $\mathbf{q}_\pi$ simply pick the action with the largest value. We of course don't have access to these and have to learn them from experience. How might we do this? One way is to somehow track the average return that you get from a particular state or state action pair. This approach is called Monte Carlo methods. This is quite relevant for understanding AlphaZero as a form of tree search called Monte Carlo tree search is central to the algorithm.

One central idea in reinforcement learning is the ability to view a value function recursively. Lets look at how we can do that. First we can write our cumulative variable G_t in terms of the next immediate reward and a discounted cumulative return starting from the next timestep.

$$\mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s]$$

Then we can expand this by thinking about what an expectation means here in terms of our random variable R_{t+1} . In words it is the value of all future rewards we would see by transitioning to new states. We have said previously that the probability of transitioning to a new state s' and receiving reward r from the current state given a certain action is given by the function $p(s', r | s, a)$ the system dynamics of our MDP. We know the definition of expected value is the sum over possible outcomes of a random weighted by the probability of that outcome happening. So we can write the expected value of R_t given a specific state and action.

$$\mathbb{E}[R_t | S_t = s] = \sum_{s', r} p(s', r | s, a) [r]$$

We want however to look at the expectation following a specific policy. A policy $\pi(a|s)$ gives us a distribution over possible next actions given the current state. So we can further expand on our previous equation.

$$\mathbb{E}_\pi[R_t | S_t = s] = \sum_a \pi(a|s) \sum_{s', r} p(s', r | s, a) [r]$$

We can now go back to our expectation of the cumulative reward.

$$\mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] = \sum_a \pi(a|s) \sum_{s', r} p(s', r | s, a) [r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s']]$$

Notice how the expectation is now conditioned on the next state. This allows us to rewrite this in a recursive form.

$$\mathbf{v}_\pi(\mathbf{s}) = \sum_a \pi(a|\mathbf{s}) \sum_{\mathbf{s}', r} p(\mathbf{s}', r | \mathbf{s}, a) [r + \gamma \mathbf{v}_\pi(\mathbf{s}')] \quad (2.5)$$

This last form is known as the Bellman equation. The Bellman equation represents values of a state in terms of values of future states. The figure below taken from Sutton represents this as a decision tree. The open circles are states and the black circles are actions. This shows if you are in state s you have 3 actions that are part of policy π . Once you take one of those actions the environment then transitions to one of two next states s' and emits a reward r with probability p . You could then extend the diagram in the same way from each of the next states. If this were a two player game the other player would be part of the environment and the probability of transition to state s' would depend on the other player's policy π . Sutton refers to these as backup diagrams because they show how information could be propagated back up the tree.

Figure 2.3: RL

2.3 Tictactoe Bellman Equation

Lets try and picture what the bellman equation might look like for the game of tictactoe. This example will help to both illustrate how the equation works and also the intractability of calculating it in practice. The figure below shows the initial blank board at the beginning of the game. To calculate the value of this position we will do so from player X's perspective. There are nine possible next moves and so each of those needs to be assigned a probability represented by the policy $\pi(a|s_0)$. Then focusing on action a_0 at the top of the diagram the next thing we would need to do would be to calculate $p(s', r | s, a)[r + \gamma \mathbf{v}_\pi(\mathbf{s}')] for each of the possible next states. So we could start with $p(s_1, r | s_0, a_0)$. What is this probability? We will need to know this beforehand. How it is obtained depends. We might just be given the distribution beforehand as part of our model of the environment. In this instance however a more intuitive candidate would be to ask player "O" for their policy. That is if we had access to π from player "O" perspective that would be then give the associated probability of future states since future states are just the board after "O" has chosen an action. Now that we have $p(s_1, r | s_0, a_0)$ we need $[r + \gamma v(s_1)]$. The reward r here refers to the immediate reward observed. In the case of tictactoe and most other two player board games there will be no reward till the end of the game. So r here is just 0. So we need $\gamma v(s_1)$. Here γ is a discount factor and is constant and chosen beforehand. The $v(s_1)$ calculation is shown next.$

$$\begin{array}{c}
 v_{\pi}(\begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline & & & \\ \hline \end{array}) = \sum_a \begin{array}{c} \pi(a_0 | s_0) \\ \begin{array}{|c|c|c|c|} \hline \text{x} & & & \\ \hline & & & \\ \hline & & & \\ \hline \end{array} \\ \pi(a_1 | s_0) \\ \begin{array}{|c|c|c|c|} \hline & & \text{x} & \\ \hline & & & \\ \hline & & & \\ \hline \end{array} \\ \vdots \\ \pi(a_5 | s_0) \\ \begin{array}{|c|c|c|c|} \hline & & \text{x} & \\ \hline & & & \\ \hline & & & \\ \hline \end{array} \\ \vdots \end{array} * \sum_{s', r} \begin{array}{c} p(s_1, r | s_0, a_0) = ? \\ \begin{array}{|c|c|c|c|} \hline \text{x} & \text{o} & & \\ \hline & & & \\ \hline & & & \\ \hline \end{array} * [r + \gamma v(s_1)] \\ p(s_2, r | s_0, a_0) = ? \\ \begin{array}{|c|c|c|c|} \hline \text{x} & & & \text{o} \\ \hline & & & \\ \hline & & & \\ \hline \end{array} * [r + \gamma v(s_2)] \\ \vdots \\ \begin{array}{|c|c|c|c|} \hline \text{x} & & & \\ \hline & \text{o} & & \\ \hline & & & \\ \hline \end{array} * [r + \gamma v(s_4)] \\ \vdots \\ \vdots * \sum_{s', r} \dots \end{array}
 \end{array}$$

Figure 2.4: Tictactoe Bellman 1

In figure 2.5 below we see the same calculation expanded out but for state s_1 . This same sort of recursive procedure would keep going until a terminal state was reached. For tictactoe this occurs only when the game ends and a reward of 1,-1 or 0 is assigned for win,loss or draw.

$$\begin{array}{c}
 v_{\pi} \left(\begin{array}{|c|c|c|} \hline \text{x} & \text{o} & \\ \hline & & \\ \hline & & \\ \hline \end{array} \right) = \sum_a \pi(a_2 | s_1) \begin{array}{|c|c|c|} \hline \text{x} & \text{o} & \text{x} \\ \hline & & \\ \hline & & \\ \hline \end{array} \\
 \vdots \\
 \pi(a_3 | s_1) \begin{array}{|c|c|c|} \hline \text{x} & \text{o} & \\ \hline \text{x} & & \\ \hline & & \\ \hline \end{array} \\
 \vdots \\
 \pi(a_4 | s_1) \begin{array}{|c|c|c|} \hline \text{x} & \text{o} & \\ \hline & \text{x} & \\ \hline & & \\ \hline \end{array} \\
 \vdots
 \end{array}
 \begin{array}{c}
 p(s_9, r | s_1, a_2) = ? \\
 \begin{array}{|c|c|c|} \hline \text{x} & \text{o} & \text{x} \\ \hline \text{o} & & \\ \hline & & \\ \hline \end{array} * [r + \gamma v(s_9)] \\
 \\
 p(s_{10}, r | s_1, a_2) = ? \\
 \begin{array}{|c|c|c|} \hline \text{x} & \text{o} & \text{x} \\ \hline & \text{o} & \\ \hline & & \\ \hline \end{array} * [r + \gamma v(s_{10})] \\
 * \sum_{s', r} \\
 p(s_{11}, r | s_1, a_2) = ? \\
 \begin{array}{|c|c|c|} \hline \text{x} & \text{o} & \text{x} \\ \hline & & \text{o} \\ \hline & & \\ \hline \end{array} * [r + \gamma v(s_{11})] \\
 * \sum_{s', r} \\
 \vdots
 \end{array}$$

Figure 2.5: Ticatactoe Bellman 2

2.4 Policy evaluation Example

Lets look at an example of evaluating this a value function for a specific state and policy using a more simple tictactoe example as a test case. Consider the board below. It is player X to go. How to you calculate equation (2.5)? Once again we need access to a policy and access to the transition probability distribution.

X		
O	O	
X	X	O

Figure 2.6: policy eval 1

Lets ignore that this is a two player game for a moment. From player X's perspective all we need is to set our own policy. Then the next state we observe after taking an action will be given by the environment. In this case player "O" will just be part of the environment. The situation is displayed in the figure below. We have three possible actions to take. We can see what the correct thing to do is but a computer has no idea! So lets assign probability $\frac{1}{4}$ to the top two actions and $\frac{1}{2}$ to the other action. When the agent takes action a_2 or the top right square he observes a next state s_1 and reward -1 . This event happens with probability one and so no other events occur after taking that action. The same is shown for action a_1 . For action a_5 the next state is observed but it is not a terminal state so a policy needs to be defined. There is only one action so the definition is simple. No next state is observed just a reward of 0 since it is a terminal state. Lets now actually calculate $v_\pi(s_0)$

$$v_{\pi} \left(\begin{array}{|c|c|c|} \hline x & & \\ \hline o & o & \\ \hline x & x & o \\ \hline \end{array} \right) = \sum_a \left(\begin{array}{|c|c|c|} \hline x & & \\ \hline o & o & \\ \hline x & x & o \\ \hline \end{array} \right) \begin{array}{l} \pi(a_2|s_0) = \frac{1}{4} \\ \pi(a_1|s_0) = \frac{1}{4} \\ \pi(a_5|s_0) = \frac{1}{2} \end{array}$$

$\begin{array}{|c|c|c|} \hline x & x & \\ \hline o & o & \\ \hline x & x & o \\ \hline \end{array} \begin{array}{l} p(s_1, -1|s_0, a_2) = 1 \\ * [-1 + 0] \end{array}$

$\begin{array}{|c|c|c|} \hline x & x & \\ \hline o & o & \\ \hline x & x & o \\ \hline \end{array} \begin{array}{l} p(s_2, -1|s_0, a_1) = 1 \\ * [-1 + 0] \end{array}$

$\begin{array}{|c|c|c|} \hline x & & \\ \hline o & o & x \\ \hline x & x & o \\ \hline \end{array} \begin{array}{l} p(s_3, 0|s_0, a_5) = 1 \\ * [0 + \gamma v_{\pi}(s_3)] \end{array}$

$$= \sum_a \left(\begin{array}{|c|c|c|} \hline x & x & o \\ \hline o & o & x \\ \hline x & x & o \\ \hline \end{array} \right) \begin{array}{l} p(s_4, 0|s_3, a_1) = 1 \\ * [0 + 0] \end{array}$$

Figure 2.7: policy eval 2

Lets calculate the value of using this policy from the initial state using equation (2.5). With the small number of states being used this is a fairly easy calculation but its easy to see how this could easily start to become intractable.

$$\begin{aligned}
v(s_0) &= \pi(a_0 | s_0) * p(s_1, r | s_0, a_0) * [r] \\
&\quad + \pi(a_1 | s_0) * p(s_2, r | s_0, a_1) * [r] \\
&\quad + \pi(a_2 | s_0) * p(s_3, -1 | s_0, a_2) * [r + \gamma v(s_3)] \\
&= \frac{1}{4} * 1 * [-1] \\
&\quad + \frac{1}{4} * 1 * [-1] \\
&\quad + \frac{1}{2} * 1 * [0 + 0.9 * [0 + 0]] \\
&= -\frac{1}{2}
\end{aligned}$$

Figure 2.8: policy eval 3

We have now discussed how to evaluate given states within our framework. We have not discussed yet how to actually use this information to make good decisions. There is a theorem in RL that says that a policy π is better than or equal to another policy π' if its expected value is greater than or equal to the expected value of π' for all states. In other words $\pi \geq \pi'$ iff $\mathbf{v}_\pi(s) \geq \mathbf{v}_{\pi'}(s)$ for all $s \in S$. It has also been shown and we will restate here without proof that an optimal policy always exists for an MDP. There can be more than one but at least one exists. The notation used for the optimal policy is usually π_* and we will take that notation here as well. This leads also to the fact that there is an optimal state value function.

$$\mathbf{v}_*(s) \doteq \max_{\pi} \mathbf{v}_\pi(s), \forall s \in S \quad (2.6)$$

The bellman equation representation must hold as well. In our previous representation of the bellman equation we had to sum over all actions from our policy $\pi(a|s)$. When looking at an optimal policy in the case of an MDP we will only be

interested in taking the best action. So we can rewrite the bellman equation of the value function with that in mind.

$$\mathbf{v}_*(s) = \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma \mathbf{v}_*(s')] \quad (2.7)$$

A similar analysis can be done for $q(s,a)$. We will not go through that here but you can look to Sutton or other sources for more discussion. We will simply present the result here.

$$\mathbf{q}_*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \max_a \mathbf{q}_*(s',a')] \quad (2.8)$$

I think that the $\mathbf{q}_*$ optimal value function can be more intuitive. It simply says that the value of the optimal action from the current state is a weighted sum over possible next states assuming we take the optimal action from those states as well. If you have $\mathbf{q}_*$ or $\mathbf{v}_*$ then coming up with the correct policy is easy. For $\mathbf{v}_*$ you just have to do one step look ahead to see which actions will give you the max. You don't have to continue to look further because you already have access to the optimal values of the next state. With $\mathbf{q}_*$ you simply look up the best action from the current state. No further search is necessary. Obtaining these optimal functions of course generally takes a lot of work and in any large setting is intractable to compute directly. To actually compute an optimal bellman equation you need to have perfect knowledge of the environment, sufficient resources and the markov property. These assumptions typically don't hold for most interesting problems. For any problem that breaks one of these assumptions some sort of approximation will be needed. One approach that fits in closely with the bellman optimality equation is dynamic programming which we

will discuss shortly. One big takeaway from reading Sutton and others work is that most reinforcement learning techniques come down to different ways of approximating the bellman optimality equation. To me this is quite powerful and means that even though we might come across some powerful approximation techniques such as those that we see in AlphaZero they can all be viewed from the perspective of approximating the bellman equation in some way.

2.5 Dynamic Programming

Dynamic programming (DP) is one of the more foundational topics within reinforcement learning. We will not dwell on this topic too long but there are a few important lessons in which DP can help inform our understanding of RL and AlphaZero. We have just concluded discussing optimal value functions and now we need to find a way to actually compute them. That is where DP comes in. DP is essentially a collection of algorithms used to compute optimal value functions. Recall that (2.5) gave us a way to evaluate the value of state given a certain policy. This equation actually gives us a system of linear equations and can be solved directly if the model of the environment is totally known. We have $|S|$ equations and $|S|$ unknowns. This of course is not computationally very feasible so we need to find a way to compute (2.5) iteratively. We can actually just turn it into an update rule like the following.

$$\mathbf{v}_{\mathbf{k}+1}(\mathbf{s}) = \sum_a \pi(a|\mathbf{s}) \sum_{\mathbf{s}', r} p(\mathbf{s}', r | \mathbf{s}, a) [r + \gamma \mathbf{v}_{\mathbf{k}}(\mathbf{s}')] \quad (2.9)$$

We can first initialize $v_0(s)$ for each state randomly. We can use domain knowledge to set reasonable initial values as well or just set everything to 0 to start. This iterative process will converge to v_π as $k \rightarrow \infty$. This is called *iterative policy evaluation* and it gives us a way to evaluate the current policy π in an iterative fashion. We of course

don't have to wait for $k \rightarrow \infty$. In practice we can implement some stopping condition for which we are satisfied that v_k is close enough to v_π .

We don't want to just know the value of a given policy however we want to find what the best policy is. So we need to find a way to update our policy as well. I will state here what is known as the *policy improvement theorem*.

Policy Improvement Theorem 1. *Let π and π' be any two separate deterministic policies for all $s \in S$ s.t. $q_\pi(s, \pi'(a)) \geq v_\pi(s)$. Then the policy π' is greater than or equal to π . This implies that $\forall s \in S$ each state value function has the relationship that $v_{\pi'}(s) \geq v_\pi(s)$*

The idea here is that if you can take a policy π and change one of its actions at a particular state to guarantee that it is at least as good or better than π at that state then you can just change the policy at that particular state to the new action and you now have a better policy π' . How can we do this at every state to give us a general methodology? We can do what's called acting greedily. Acting greedily means that from a given state you take the action that maximizes return from a one step look ahead perspective according to v_π . We can adjust (2.5) once again to represent this.

$$\pi' = \underset{a}{\operatorname{argmax}} \sum_a \pi(a|s) \sum_{s', r} p(s', r | s, a) [r + \gamma v_\pi(s')] = \underset{a}{\operatorname{argmax}} \mathbf{q}_\pi(\mathbf{s}, \mathbf{a}) \quad (2.10)$$

This just says what we have already said in words. We use the current policy π and take the action that is going to maximize our policy evaluation from a given state using π . This gives us a new policy π' that we know from the policy improvement

theorem will guarantee us an improved policy. The only way its not an improvement once again is if we already had an optimal policy. Now that we have a new policy π' how can we continue to improve it. We need to evaluate $\mathbf{v}_{\pi'}$ using policy evaluation. Then we can use (2.10) again to get a new policy π''

$$\pi'' = \underset{a}{\operatorname{argmax}} \sum_a \pi'(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma \mathbf{v}_{\pi'}(s')]$$

We can then repeat this process until convergence. This algorithm is called policy iteration and is given in full below.

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
 $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$
2. Policy Evaluation
 Loop:
 $\Delta \leftarrow 0$
 Loop for each $s \in \mathcal{S}$:
 $v \leftarrow V(s)$
 $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s)) [r + \gamma V(s')]$
 $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
 until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)
3. Policy Improvement
 $\text{policy-stable} \leftarrow \text{true}$
 For each $s \in \mathcal{S}$:
 $\text{old-action} \leftarrow \pi(s)$
 $\pi(s) \leftarrow \underset{a}{\operatorname{argmax}} \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$
 If $\text{old-action} \neq \pi(s)$, then $\text{policy-stable} \leftarrow \text{false}$
 If policy-stable , then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Figure 2.9: From Sutton Chpt 4.3

This idea of iterating back and forth between evaluation of a policy and improving a policy is a fairly general concept. In fact there is something called *General policy iteration* (GPI) which captures these two concepts. Many reinforcement learning algorithms can be described in terms of GPI. The process is quite interesting because as soon as you perform policy improvement your value function is no longer accurate

because it is still in reference to the old policy. So then you update the value function with the new policy to make it accurate again. The policy however is no longer optimal w.r.t the current value function so you adjust the policy by once again acting greedily. This goes back and forth until convergence is reached. There is a constant pushing and pulling between policy evaluation and policy improvement.

2.6 Monte Carlo Methods

Monte Carlo in regards to reinforcement learning is a bit more specific than its typical meaning. In this discussion by "Monte Carlo" we mean any method that uses an average return for estimation. We will focus on estimating the value $\mathbf{v}_\pi(s)$ of a particular state. We said that our goal with $\mathbf{v}_\pi$ was to compute the expected value of the cumulative returns from a particular state. To apply monte carlo methods to this problem we can just track the average return we observe when visiting that particular state. We can do this directly from experience assuming the agent is following the policy π and in the limit we should have a true estimate of v_π . This is called Monte carlo prediction and the most common form is called first-visit. This means that we simply average the return from the first visit to a particular state. For every visit we average the return following every visit. The first-visit method has better understood theoretic properties. Every-visit however extends more naturally to function approximation. They both however converge to $\mathbf{v}_\pi(s)$ as the number of visits goes to infinity. The outline of the algorithm taken from Sutton is displayed below.

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated

Initialize:

$V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):

Generate an episode following π : $S_0, A_0, R_1, S_1, A_1, R_2, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \dots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless S_t appears in $S_0, S_1, \dots, S_{t-1}$:

Append G to $Returns(S_t)$

$V(S_t) \leftarrow \text{average}(Returns(S_t))$

Figure 2.10: First-visit MC prediction

Some important facts about MC methods

1. One advantage of MC prediction is that we don't need to have a complete model of the environment in order to come up with an estimate for the state s . Since Monte Carlo Methods work with sample episodes they simply can use a given policy to keep generating samples.
2. MC methods always run to the end of an episode whereas DP only looks one step ahead.
3. The estimate for each state is independent. For DP this is not the case because DP bootstraps
4. Computational expense of estimating the value of a state is independent of the number of states

Advantages over DP

- Can learn from actual experience
- Can learn from simulated experience
- Can generate sample episodes starting from some state that your interested in and just average return from only that state and ignore the rest.

Just like we saw with DP we can move from simply evaluating a given policy to actually attempting to find a best policy. We say that General policy iteration (GPI) was a flexible way of framing the problem. We can do a similar thing with MC methods. Just as before we can improve the policy by simply acting greedily w.r.t the value function $q(s, a)$. It is easier to frame a GPI model for MC methods in terms of the action-value function q_π instead of v_π which requires us to do a lookahead search. So the greedy policy is defined as

$$\pi(s) \doteq \underset{a}{\operatorname{argmax}} q(s, a) \tag{2.11}$$

The policy improvement theorem as discussed before applies to π_k the k th step of the algorithm. This means that if at each iteration of the algorithm we improve our policy by acting greedily than we can assume that the following is true.

$$q_{\pi_k}(s, \pi_{k+1}(s)) \geq \mathbf{v}_{\pi_k}(s)$$

There are two assumptions here that are necessary for convergence. The first is that sufficient exploration is happening. That is that each state action pair has some non zero probability of occurring. The second is that we need an in order to evaluate a given policy we need an infinite number of episodes. The Monte Carlo Exploring

Starts algorithm is an example of one that drops the second requirement but satisfies the first by randomly choosing a state action pair to start at. The value function $Q(S_t, A_t)$ is able to converge accumulating and averaging returns for all episodes. Convergence although intuitively seems guaranteed has not so far been proven. The exploring starts assumption can be dropped by doing things like using a ϵ -greedy algorithm that with probability ϵ takes a random action. This will ensure that all state action pairs are visited in the limit and allows you to start each episode in a more "natural" manner.

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:

$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$

$Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$

$Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$

Loop forever (for each episode):

Choose $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0

Generate an episode from S_0, A_0 , following π : $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \dots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1, \dots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t, a)$

Figure 2.11: Monte Carlo ES (Exploring Starts)

2.7 TD learning

Temporal difference learning (TD) is an important concept in reinforcement learning and it combines a lot of the information that we have covered so far. It takes ideas from MC methods and dynamic programming. It can learn from experience like we saw in MC methods but also bootstraps like in DP. In the equations below we can see the differences between TD learning and MC Methods. The first equation

is essentially what MC methods attempts to estimate. The last equation is what DP learning uses and also what TD learning uses. You can see how in the last equation we are bootstrapping by using a future estimate to approximate the current one.

$$\begin{aligned} \mathbf{v}_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma \mathbf{v}_\pi(S_{t+1}) | S_t = s] \end{aligned}$$

In TD learning the last equation gives us our TD target and so in their most basic form TD methods make an update to a value function as follows.

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

We are using $V(S_t)$ here to signify that we don't actually have access to v_π . Likewise the "target" for MC methods is generally G_t and we sample the expected value in order to approximate it. In TD learning we are using sampling as well. We sample the expected value in the third equation and we also maintain an estimate of v_π . This is why we say that TD combines MC methods and DP. It uses the sampling of MC with the bootstrapping of DP. We can use this basic update above to give us a simple method for doing policy evaluation.

Tabular TD(0) for estimating v_π

```

Input: the policy  $\pi$  to be evaluated
Algorithm parameter: step size  $\alpha \in (0, 1]$ 
Initialize  $V(s)$ , for all  $s \in \mathcal{S}^+$ , arbitrarily except that  $V(\text{terminal}) = 0$ 

Loop for each episode:
  Initialize  $S$ 
  Loop for each step of episode:
     $A \leftarrow$  action given by  $\pi$  for  $S$ 
    Take action  $A$ , observe  $R, S'$ 
     $V(S) \leftarrow V(S) + \alpha [R + \gamma V(S') - V(S)]$ 
     $S \leftarrow S'$ 
  until  $S$  is terminal

```

Figure 2.12: Tabular TD Sutton Chpt 6

The difference here in the bootstrapping mechanism from that of DP is that a single sample of a successor state $V(S')$ is used instead of looking at all possible next states. The quantity in brackets in the TD update is called the TD error and it comes up a lot in reinforcement learning. It is a sort of error because we have a target ($R_{t+1} + \gamma V(S_{t+1})$) and we have our estimate $V(S_t)$. There is an interesting result here that if $V(s)$ is not updated during an episode then we can actually write the Monte carlo error as a sum of TD errors

$$G_t - V(t) = \sum_{k=t}^{T-1} \gamma^{k-t} \delta_k$$

Here δ_k is the TD error. $\delta_k = R_{k+1} + \gamma V(S_{k+1}) - V(S_t)$

TD methods have some advantages over MC methods and DP methods.

* TD $\hat{=}$ DP because no model of the environment is required. * TD $\hat{=}$ MC methods in that they are naturally implemented in an online, fully incremental fashion. Dont need to wait until the end of an episode. * Has similar convergence guarantees as DP and MC.

Now just like with DP and MC methods we need to go from simply evaluating a policy to having a method for improving a policy. We will use the GPI as before as a framework for achieving this. Just like with MC we focused on estimating $q_\pi(s, a)$ which does not force us to have a model of the environment. The TD update can be rewritten in this notation as

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

This is an on-policy method since we are using our policy to sample actions A_{t+1} for our target value. The algorithm below is an example of an algorithm that utilizes this update. We can see that policy evaluation and policy improvement are coupled a bit more tightly than before. We are using the current value estimate $Q(s, a)$ to find a greedy policy that serves as our policy improvement step. Then for policy evaluation we use the update previously described.

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
 Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$
 Loop for each episode:
 Initialize S
 Choose A from S using policy derived from Q (e.g., ε -greedy)
 Loop for each step of episode:
 Take action A , observe R, S'
 Choose A' from S' using policy derived from Q (e.g., ε -greedy)
 $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
 $S \leftarrow S'; A \leftarrow A';$
 until S is terminal

Figure 2.13: Sarsa: Sutton Chpt 6

A variant of Sarsa is called Q learning and has gained quite a bit of popularity since its recent success with its deep learning variant DQN which we will discuss later. Q learning modifies the TD target by simply taking the max action of the observed next state. So the TD error becomes.

$$\delta_t = R_{t+1} + \gamma \max_a Q(S_{t+1}, A_{t+1}) - Q(S_t, A_s) \quad (2.12)$$

The learned value function in this case directly approximates q_* . This is because it satisfies the policy improvement theorem by always taking the max action. It also still manages to visit all state action pairs by acting ϵ -greedily from the current state. There have been a lot of nice theoretic work to prove convergence (Watkins, 1989). The adjusted algorithm is presented in full below. The change to using the max action in place of an action selected from the policy means that we form a target that does not depend on the policy being used and therefore makes the algorithm an off-policy algorithm. So this off-policy distinction simply means that with off-policy we learn using a different policy than the one we used to take actions with. Q learning has had a lot of success when combine with deep learning which we will look at next.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
 Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$
 Loop for each episode:
 Initialize S
 Loop for each step of episode:
 Choose A from S using policy derived from Q (e.g., ε -greedy)
 Take action A , observe R, S'
 $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
 $S \leftarrow S'$
 until S is terminal

Figure 2.14: Q learning: Sutton Chpt 6

2.8 Function approximation and Deep RL

We have seen so far that no matter what the approach to RL is you need somehow to evaluate a policy and you need a way to improve a policy. We also identified some of the bottlenecks of various approaches. DP for instance suffers from the "curse

of dimensionality” in that as the size of the state space increases the computational requirement grows exponentially. Monte carlo methods and TD learning helped us to get around this by being able to effectively sample and learn from experience. We looked at tabular methods that essentially maintained a lookup table for each state action pair like in Q learning. This has its limits as the state and action space grows. It will become unfeasible to require approximating the value of $Q(s,a)$ for every state and action pair. It requires too much computation to adequately approximate the true q_π for a given policy. If the state space is large it is reasonable to think that an agent will often encounter states that it has never seen before even after a lot of exploration. What if we were able to generalize the information learned from one state to other similar states? This would allow us to more effectively make use of experience. This task of generalizing from a given set of data to unseen data has been studied quite a bit. Function approximation is the typical approach that helps us to learn from examples.

Function approximation usually means that we attempt to parameterize a function with a set of learned weights. Parameterizing a function would mean that we take a set of weights $\mathbf{w} \in \mathbb{R}^d$ and use them to approximate the true value function $\mathbf{v}_\pi$ for a given policy π . This can be represented as $\hat{\mathbf{v}}(s, \mathbf{w}) \approx \mathbf{v}_\pi$. The function $\hat{\mathbf{v}}$ can be something complex like a neural network or something more simple like a linear function. Function approximation helps with generalization in that we now have a set of weights that ”represent” the states. Instead of having a separate representation for each state we can represent many states with a lower dimensional representation since typically $d \ll |S|$. Two states that are similar by some metric will have a similar representation in this new weight space. That means that it should have a similar

output as well. So if we have some similarity metric d then if for two states s_1 and s_2 we have that $d(s_1, s_2; \mathbf{w}) \approx 0$ then we also should have that $|\hat{\mathbf{v}}(s_1, \mathbf{w}) - \hat{\mathbf{v}}(s_2, \mathbf{w})| \approx 0$. Thinking to a tictactoe example you would want two board positions that are similar to have similar values. Such as the following two boards.

		x	x		
	o			o	

Figure 2.15: TicTacToe Board

Supervised learning is a common function approximation technique used in machine learning in which an algorithm is given a set of data X and some labels Y that are the thing the algorithm is attempting to learn. The goal is to find patterns in X that allow one to be able to minimize some loss function $L(f(X'), Y')$ where the algorithm is represented by the function f and X', Y' is some unseen data but that come from the same distribution as the observed data X . Some of the difficulty in applying supervised learning techniques to reinforcement learning is that we actually are not looking at a stationary distribution. The parameters that describe the underlying distribution are capable of changing over time and as the agent interacts with the environment. To further highlight difficulties with using this approach in RL we are generally trying to do policy evaluation using the current policy π . This however will be changing as we run policy improvement. So we must use methods that are able to handle the nonstationarity of the problem.

Nevertheless we can start by phrasing our RL problem as a SL problem. First we

need to actually specify our loss function for RL as well. We know that we want to minimize some distance between what our parameterized prediction says and what our target says. We want to specify $L(\mathbf{v}_\pi, \hat{v}(s, \mathbf{w}))$. A common function for L is the MSE. We can write this as $\mathbf{MSE} = \sum_{s \in S} [\mathbf{v}_\pi(s) - \hat{v}(s, \mathbf{w})]^2$. We don't have access to the underlying function $\mathbf{v}_\pi$ but we can use our target that we have specified in previous examples. In supervised learning terminology we have a target like G_t (MC target) or $R_{t+1} + \gamma \hat{v}(s, \mathbf{w})$ the TD target which is like our labels Y and we have the current state S_t which is like an observation of X

2.9 DQN

We have discussed briefly the idea of applying function approximation to RL. The task is non trivial and we have noted a couple of the dangers involved. I will now use a popular paper to further the discussion of the use of function approximation to enhance the ability of a RL algorithm to better generalize. A nature paper from 2015 entitled "Human-level control through deep reinforcement learning" [Mni+15] showed how you could take a "simple" algorithm like Q learning and scale it to be a state of the art. Previous work on applying Neural networks to Q learning showed that this approach was unstable [Rie05]. The instability comes from several sources.

- Correlations present in the sequence of observations
- Small updates to Q may significantly change the policy and therefore the data distribution
- Correlations between action-values (Q) and the target values $r + \gamma Q(s', a')$

We have already mentioned the second issue which refers to the nonstationarity of the data. The other two refer to a common problem that most supervised learning

(SL) problems are forced to address. That is correlation among samples. There is a common assumption made in most SL tasks that the observed samples are IID (Independently and Identically distributed). A lot of care has to be taken to help at least approximate this assumption. In the case of Q learning the sequences are highly correlated but so are the labels. The DQN paper addressed these issues in two ways.

- Using an experience replay buffer that randomizes the data
- Only periodically update Q and maintain a target network.

The first helps to solve the first two problems. It helps remove correlation in the observations through random sampling of a replay buffer. We will discuss the details of the replay buffer shortly. The randomization also helps to solve the nonstationarity problem as it smooths out the data distribution. DQN only periodically updates the Q function which helps to remove correlation with the target values. Since a separate network is maintained for making target predictions this means that the target values remain much more stable.

DQN uses a neural network to parameterize the state-action function $Q(s, a) \rightarrow Q(s, a, \theta_i)$. Here θ_i signifies the parameters of the network at iteration i . Recall from our discussion of Q learning that the TD error had the form $\delta_t = R_{t+1} + \gamma \max_a Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$. In the DQN paper they translate this to a differentiable loss function as follows.

$$\mathbf{L}_i(\theta_i) = \mathbb{E}_{s,a,r,s' \sim U(D)} [(r + \gamma \max_a Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))^2]$$

The θ^- weights are the weights for the target network. This is simply the mean squared error of the target and predicted values using the parameterized version

of the Q function. Note how the target value depends on the prediction of the target network. This is really a large distinction between reinforcement learning and supervised learning. A picture of their network used is given below. They evaluated their agent on Atari games and used a convolutional neural network for the parameterized Q function. The output of the network that you see on the far right is the set of possible actions with each of the corresponding values.

Figure 2.16: DQN architecture

Prior to looking at the algorithm in full let's look at two of the more important components that made DQN successful. First the target network being used requires a bit more explanation. To be clear it's not actually a separate network. It is simply weights of the Q network at prior iterations. So it is basically "old" weights. We specify the number of iterations that we wait to update the target network up front as a hyper parameter. When we do finally update the target network we simply copy the current network weights and use them as our new target. This does not completely fix the nonstationary problem since in all RL algorithms that bootstrap

we will always have some amount of variability in the target distribution.

We also need to look at what an experience replay buffer is. The experience replay buffer is essentially a database of experience. While the agent interacts with the environment data that is generated from that interaction is stored away. Then to update the network batches of experience are randomly sampled from the buffer. The fact that the experience is sampled randomly is significant in that it helps to decorrelate the data and stabilize training. Experience is typically stored in the form (s_t, a_t, r_t, s_{t+1}) . The current state, action, reward and the next state that the agent transitions to after taking action a_t . A replay buffer will be used in the AlphaZero algorithm so keep it in mind going forward.

The algorithm from the paper is in the figure below. Lets go through each one of the steps. It might look complicated but it is really quite similar to Q learning that we have seen plus some deep learning magic.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
 For $t = 1, T$ **do**
 With probability ϵ select a random action a_t
 otherwise select $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$
 Execute action a_t in emulator and observe reward r_t and image x_{t+1}
 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
 Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
 Every C steps reset $\hat{Q} = Q$
 End For
End For

Figure 2.17: The DQN algorithm

1. Initialization - We simply initialize the weights of the network randomly and

then set the target weights to a copy of those weights. We also initialize the replace buffer which simply means instantiating whatever data structure your using to hold the data such as an array.

2. Inside the main for loop we get the initial sequence. A sequence here is a stack of the 4 most recent frames. Then the function $\phi(s_t)$ scales those frames to be of size 84 x 84.
3. The second for loop takes us through each step of the episode. We use an ϵ -greedy strategy to select an action. The action is executed against the environment and the reward and next state are observed and processed.
4. We store the current data into the replay buffer.
5. We train our network on a random sample of transitions from our replay buffer using gradient decent and the previously discussed loss function.

Thats really it. This is something that you can code up in less than 150 lines of python code using a modern machine learning framework like pytorch or tensorflow. This seemed like a nice elegant algorithm that utilized deep learning to reach state of the art performance on difficult tasks like the Atari games. So why not use this same approach to solve something like Go? There is something that this approach still suffers from. It has difficulty with the credit assignment problem and stability in sparse reward regimes. The credit assignment problem in RL is the problem of knowing whcih action in which state was responsible for a particular outcome. In a game like Go you will generally take many many actions prior to the end of the game and wont receive any feedback until the game is complete. This is what is meant by

sparse rewards. The lack of informative feedback from the environment. Once the game is complete you will know whether you won or lost (+1,-1). Then how do you inform the network which states and actions were responsible for the outcome? In theory it's not totally obvious why the Deep Q learning algorithm should not be able to overcome these issues. In absence of better theory however we must be content with the empirical evidence that points to DQN's instability in these regimes. We now will discuss the idea of planning and how that helps to overcome this problem. AlphaZero is a complex planning algorithm so we will start with something a bit more simple.

3.0 Search

3.1 Planning

We will now discuss the idea of planning in RL. Planning is fundamental to the AlphaZero algorithm. Planning in the context of RL simply means any way in which an agent can use a model of the environment to improve upon a current policy. Either a global or local policy. We have mentioned models already but just to reiterate that a model is anything by which an agent can make predictions about what the environment will do next based on its own actions / interaction with that environment. This is actually intuitive as it is what a lot of us do well trying to make a decision in various domains. Think about trying to make a decision in your favorite game. You usually think along the lines "If I do this then they will do that and then if they do that I will do this...". You think about possible options and then what those various options could lead to. This usually leads to local improvements in your policy since you are just thinking about the current state and what is the best thing to do from there. There are various ways to do planning in RL but we will focus on what is called decision time planning. Decision time planning is what you do when thinking about making a decision in a game as we just discussed. The other way would be to plan from a more global perspective as we do with dynamic programming. That is before we select an action for any particular state we have done a whole lot of planning to compute a value function across all states. In the decision time planning perspective we are focused on a particular state and once we have decided on what action to take the planning process starts over again on the next state. We will see that this is what monte carlo tree search does and it the decision time planning algorithm used

by AlphaZero.

Monte Carlo tree search is also a form of what is called a "rollout algorithm". Rollout algorithms are decision-time planning algorithms but they are based on Monte Carlo control which we discussed earlier. They use Monte Carlo control on simulated trajectories. Simulated trajectories are trajectories of state action pairs starting from a given state and go until a terminal state is reached. These state action pairs are sampled according to the current policy. So one simply samples an action from a starting state and then uses a model of the environment to get a next state then from that state the process is repeated until a terminal state is reached. This can be done until you have a number of trajectories sampled from a given state. Then you can simply average the returns for each of the actions from the current state to get an estimate of the state-action value for each one.

Thinking back to the policy improvement theorem we know that if we can come up with a policy π' that is identical to the current policy except that for some action $q_\pi(s, a) \geq v_\pi(s)$ then we can say that π' is better than π . The sampled trajectories that we got from our rollout algorithm gives us an estimate of $q_\pi(s, a')$ for each action $a' \in \mathbf{A}(s)$. Then if we take the max action here $\max_a q_\pi(s, a')$ is a good estimate for an improved policy π' . So that is what rollout algorithms give us a way to continuously improve a policy. The rollout policy being used is important here. On the one hand you could do exhaustive search through the state space like DP and then you would get an exact estimate of the state-action values. This would just be the policy iteration algorithm we have previously seen when discussing dynamic programming. That looks at every possible trajectory and is untractable so we need something that samples less trajectories but also does so in an intelligent manner. The better this

rollout policy is the more likely you are to give an improvement to the current policy. We will now look at a specific instance of this with Monte Carlo Tree Search that provides a good way of making use of sampled trajectories.

3.2 MCTS

Here we will describe MCTS in its entirety. It is such an essential part of AlphaZero that it needs to be understood well. We will introduce some new vocabulary in this section as MCTS builds off of previous ideas in RL that are not obvious just from the description of the algorithm. I think it helps to start by seeing the algorithm in its entirety. The algorithm alternates between planning and acting. The planning step consists of a guided search through state space and the acting step consists simply on choosing the best action based on what happened during the planning step. The figure below gives a nice pictorial representation of the four main steps of the planning step. States are represented as white nodes and black circles as possible actions.

Figure 8.10: Monte Carlo Tree Search. When the environment changes to a new state, MCTS executes as many iterations as possible before an action needs to be selected, incrementally building a tree whose root node represents the current state. Each iteration consists of the four operations **Selection**, **Expansion** (though possibly skipped on some iterations), **Simulation**, and **Backup**, as explained in the text and illustrated by the bold arrows in the trees. Adapted from Chaslot, Bakkes, Szita, and Spronck (2008).

Figure 3.1: MCTS: Sutton Figure 8.10

The goal is to find the path through state space that will yield the highest return on average. You start with a game tree that only contains the root node. In the case of Go this would be the node representing the start of the game. You then proceed and repeat the following 4 steps as are represented in figure-1 for some predefined number of iterations. Once you have completed all iteration you will move on to the action phase in which you use the statistics built up in the tree to select an action in the actual environment. Each node in the tree associate with a unique state from the environment has the following statistics. For each possible action from the current state the state action value is represented by $Q(s, a)$. Visit count values for each state and action is represented by $N(s, a)$.

1. **Selection** In this phase of the algorithm child nodes are selected until a leaf node is reached. Leaf node here means either a terminating node (game ending node) or a node that has not been explored yet. The child nodes are selected based on values of the edge connecting parent to child. This value can be assigned in different ways. An example is UCT (Upper confidence bound 1). The way exploration vs exploitation is balanced is using the following formula.

$$UCT = Q(s, a) + c * \frac{\sqrt{\sum N(s, b)}}{N(s, a) + 1}$$

includegraphics[width=300px,height=100px]images/uct.png

2. Expansion

Once a leaf node has been reached as long as its not a terminating node than add one or more child nodes of that leaf node to the tree.

3. Simulation

From the selected child node run a simulation to the end of the episode (game). The simulation decisions use a rollout policy (off policy. Sometime uniform random) to select actions. This is usually done multiple times. So from the current selected node you would just select random actions until the game ends and save the result. You would repeat the process some number of times and then the average of those rollouts would be used as the estimate for the new node.

4. Backup

The reward returned from the simulation is backed up through the edges of the tree that were used during the current episode. The reward is not saved

to states that exist beyond the current tree. If the simulation step is run more than once then the average from the simulation step is backed up through the tree. If you are running MCTS for a two player game than you would flip the value of the return for the opponent node. If the node is a player one node for instance than all player 2 nodes would get the negative of whatever player 1 received from the simulation outcome. If that was confusing we will see some examples that will help clarify.

As stated before these 4 steps would continue for a predefined number of iterations and then the acting step would commence. Remember those for steps were a “thinking” phase where from the current state the algorithm is thinking about what to do. Once the thinking phase is over the agent acts by selecting an action based off statistic in the tree. One common one is to select the action with the most visit counts or the action with highest value.

Monte Carlo tree search is a decision time planning algorithm. A decision time planning algorithm is one in which a full simulation of planning is run on every state to produce each action. This is in contrast to *background planning*. In background planning a policy or value function is gradually learned by simulated experience. The learning might happen at different time intervals. In short the two differences are this, decision time planning uses simulated experience to select an action in the current state whereas background planning uses simulated experience to improve a policy or value function and then uses that improved policy or value function to then select an action.

For me this definition was a bit confusing and I suppose that there is likely some overlap in the two approaches in different algorithms. MCTS is a decision time

algorithm because from the current state S_t you might run the above 4 step 1k times or 10k times to accumulate statistics in the tree. Then once those simulations are over you will select an action from S_t using the statistics accumulated in the tree. The mechanism for selection can be different from algorithm to algorithm. You might just select an action by trying to maximize value or you might take into account the number of times an action has been taken from S_t in the past.

MCTS is also considered a rollout algorithm. “Rollout algorithms are decision-time planning algorithms based on Monte Carlo control applied to simulated trajectories that all begin at the current environment state. They estimate action values for a given policy by averaging the returns of many simulated trajectories that start with each possible action and then follow the given policy”. (Sutton Page 152).

(Leaving this out) I list a few interesting properties that make MCTS a desirable algorithm.

1. Planning time is independent of the size of the state space
2. Need lots of samples to get a good estimate
3. Running time is exponential in the horizon size. $O(|A| * n)^H$ where A is the actions space. n is the number of samples you want to run per simulation. H is the horizon size

3.2 Example 1

We will go through a quick toy example to understand each of the steps in MCTS. Below is a tree representing some game between two players blue and red. The current node of interest is the root node with value 5/10 (value / count) .

We start in the select phase and the node has three actions (children nodes) to choose from. Using the UCT algorithm given above the algorithm would ascribe the following values to each node. Starting from the left most node and working right. Let $c = 1$

$$\text{UCT}(\text{node1}) = \frac{2}{5} + c\sqrt{\frac{\ln(10)}{2}} = 1.07$$

$$\text{UCT}(\text{node2}) = \frac{2}{3} + c\sqrt{\frac{\ln(10)}{3}} = 1.54$$

$$\text{UCT}(\text{node3}) = \frac{0}{1} + c\sqrt{\frac{\ln(10)}{1}} = 1.51$$

So we select node2 as is indicated with the bold arrow. Now for the children of node2 (node4,node5)

$$\text{UCT}(\text{node4}) = \frac{0}{1} + c\sqrt{\frac{\ln(3)}{1}} = 1.048$$

$$\text{UCT}(\text{node5}) = \frac{1}{1} + c\sqrt{\frac{\ln(3)}{1}} = 2.048$$

So we select node5 and since node5 is a leaf node we go to our expansion step.

We randomly select one of the two child nodes and assign the new node values of 0 / 0 (value / count). Now to estimate the value of the new node we run a rollout policy. This is usually random and so the agent will select random actions until a terminal node is reached.

In this example the terminal node was a win for blue and so now we can back up the values of the tree with this new result. Somewhat counter intuitively since blue wins that means we increase the value of all the red nodes along with their count but

only increase the count values of all the blue nodes. This is because the red nodes are actions selected by blue and therefore since blue won that child node is more valuable to blue in the future. This is unique to multi player MCTS and you would not have to worry about this in a non-adversarial or cooperative setting.

3.3 Multi Agent RL

Let us now discuss how our previous discussed concepts in RL can be extended to the multi agent setting. Multi agent means that you have two or more players acting in the same environment with either opposing or aligned goals. We will stick to the zero-sum two player setting. Zero sum here means that the reward of one player comes at the expense of the other player. If one player wins the other must lose. Allowing for draws somewhat violates this if you allow for each player to receive some reward. In tictactoe we generally ascribe a reward of 0 for draws but there can be good reasons for at least giving an agent a small reward.

Lets think back to the tictactoe example that we used to demonstrate the intractability of policy iteration for dynamic programming 2.4. Here we said that we needed $p(s', r|s, a)$ in order to evaluate the current policy π . Well this probability in

the context of a game like tictactoe is the probability of the other player taking a certain action from the state that you just presented to them. Previously we imagined that the environment just gave us this for free but certainly this is not the case in general. So how do we get the other players policy? There are a couple of possible approaches here. The first would be to try and estimate the other players policy from experience. You could for instance run some sort of rollout algorithm where you sample trajectories and maintain a lookup table of state-action observations for how often an opponent took an action given a state. So if out of 100 times of seeing state s_1 they take action a_1 10 times then you would describe their policy as $\pi_{p2}(a_1|s_1) = 0.10$. This approach would require maintaining a lookup table and so you would probably want to apply some function approximation technique to estimate a parameterized function for π_{p2} . This is a viable approach but it is not what was done in AlphaZero. AlphaZero uses a technique called self-play.

3.4 Self-Play

Self-play is a concept that has been around in AI for a long time depending on how liberal your willing to be with the term AI. In 1951 Brown came out with a paper describing fictitious play a self-play algorithm In 1959 Checkers was solved using self-play with expert knowledge. There have been many other successes including AlphaZero. Self-play in short is any algorithm for which there are 2 or more players and the agent makes actions for every player. So the agent just plays against itself and improves play based off of what it observes during this self-play. This is easiest to see in zero sum two player games since they are symmetric. In this setting a self-play algorithm will play as both player 1 and player 2. The agent will play against itself switching perspectives between player 1 and player 2. AlphaZero does just this. We

do this as humans occasionally when playing a game or perhaps other activities. For instance when thinking about how to develop better chess strategies you might see how well a strategy does by attempting to play against that strategy yourself and observing the outcome.

This does not actually require learning two separate policies. It does however generally require adjusting how state is represented. We will add an encoding into the states representation to signify whose turn it is. This clearly designates one state from the other. In a game like tic-tac-toe you can always see whose turn it is by simply looking how many pieces are on the board for each player. This is not true of other games like chess and Go however. So our policy will give us a distribution over actions from both the below state. The first for player "X" the other for player "O".

Figure 3.2: two player policy

3.5 MiniMax and Alpha Beta Pruning

The MiniMax algorithm is a classic game tree search algorithm. It can be used to arrive at guaranteed optimal solutions in two player zero sum games.

Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Looking at figure 5.1 above taken from the classic text “Artificial intelligence a modern approach” [Rus21] we can view the game tree of a two player game as a game between a player MAX and a player MIN. It is player MAX’s turn and he wants to maximize his utility. MIN knows that MAX is a greedy human and will be taking actions to maximize his own reward. What should MIN do? MIN will then take actions that will minimize the utility of MAX. MAX in turn knows about the devious plans of MIN and will take actions in accordance with this knowledge. That is to say that each player will act assuming that the other acts optimally. This setup gives rise to our first game-tree algorithm. The optimal strategy of a given game tree can be determined by working out the minimax value of each state in the tree. The minimax value is the utility for the state. The MiniMax algorithm is the first game tree algorithm that we are going to look at. It is relevant for pedagogical purposes but we will also later use MiniMax as a baseline for comparison vs our more complex agents such as MCTS and AlphaGoZero.

MiniMax Search Example: The example below demonstrates the algorithm with

an oversimplified example. Moving left to right starting with the max player. The leaf node values are already known.

1. Max player looks for max action over children nodes.
2. No value exists on the children nodes so MIN player finds the min action of the terminal states. First for the left node. Then the right.
3. MAX can now calculate the max of MIN's nodes. So in this toy example it is concluded that MAX should take the right action.

```

function MINIMAX-SEARCH(game, state) returns an action
  player  $\leftarrow$  game.TO-MOVE(state)
  value, move  $\leftarrow$  MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v  $\leftarrow -\infty$ 
  for each a in game.ACTIONS(state) do
    v2, a2  $\leftarrow$  MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move  $\leftarrow$  v2, a
  return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v  $\leftarrow +\infty$ 
  for each a in game.ACTIONS(state) do
    v2, a2  $\leftarrow$  MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move  $\leftarrow$  v2, a
  return v, move

```

Figure 5.3 An algorithm for calculating the optimal move using minimax—the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

The pseudocode for the algorithm above can be implemented almost completely with three functions and a tree representation of whatever space you're interested in. This algorithm is guaranteed to give you the optimal solution but is quite computa-

tionally expensive as every node needs to be visited at least once. One way to help reduce the search space is with alpha-beta pruning. The basic Idea of the algorithm is that there are points during the minimax search that searching further down a particular path is unnecessary to reach the optimal minimax decision.

The above figure [2] (fig 5.5) gives a nice example of why you don't need to search the entire tree. Let the two unsearched children nodes of node C be x and y respectively. Then the Minimax evaluation of the root node of the tree following the steps we just learning is evaluated like the following

$$\text{MINIMAX}(s_0) = \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2))$$

$= \max(3, \min(2, x, y), 2) = 3$ since $\min(2, x, y)$ will never evaluate to anything larger than 2 and 2 is less than 3 then the max player will never select node C and it is unnecessary to know what x and y actually evaluate to. The figure below demonstrates this idea nicely. Consider the node n , if another node at the same level as n say m' or another node further up the tree m is better than n then we don't need to continue to explore n further. So as soon as we have enough information about n to make this conclusion we can stop and prune whatever is left of n . Alpha-beta

search as the name implies uses to values to help us know when pruning can take place.

Alpha - The value of the best choice we have found so far at any choice point along the path for MAX.

Beta - The value of the best choice we have found so far at any choice point along the path for MIN.

Figure 5.6 The general case for alpha-beta pruning. If m or m' is better than n for Player, we will never get to n in play.

Another nice way to represent this constraint is with the following graphic. Any node we are considering must be able to be between alpha and beta as represented by the number line. $\alpha \leq N \leq \beta$

4.0 AlphaZero

We have looked at minimax, alphabeta pruning and MCTS. They all have nice theoretic guarantees and have been used in many applications. Their main limitation as previously mentioned with minimax is computation. MCTS on its own works fairly well in larger state spaces. There is a limit to this however and being able to solve a game as large as Go will require some modification to the algorithm. We saw how alphabeta pruning cuts out parts of the tree that we know are not adding any information to the decision making process. Can we accomplish a similar thing with MCTS? The answer is yes, by utilizing deep learning. There have actually been many attempts to improve upon MCTS but fundamentally there are two ways to improve the search. You can either improve the depth of the search or the breadth of the search. In AlphaZero both depth and breadth are improved using neural networks. To give a quick visualization of this process look at the following three figures. They give a picture of a miniature game of Go. You start with an extremely large state space which can be seen in the first figure. You can imagine how large this picture would be if attempting to visualize the entire state space of the full game of Go or Chess.

You can see that we have two neural networks a value network and a policy network. The value network (figure 4.2) and the policy network (figure 4.3) work in unison to reduce the state space to something more manageable. The obvious question is how do we actually go about learning both of these networks?

Exhaustive Search

Figure 4.1: Mini game of go

Reduce Depth with Value Network

Figure 4.2: Reduce depth

4.1 Expert Data

Lets first start by actually defining what a value network and a policy network are. First recall that we define the value of a given state as $v_{\pi}(s) = E_{\pi}[G_t | S_t = s]$ In the context of a game like Go there are two possible outcomes $z = \{-1, 1\}$. -1 for

Reduce Breadth with Policy Network

Figure 4.3: Reduce breadth

a loss and $+1$ for a win. So we can translate the value at a given state to mean a prediction of the outcome z from a given state s . A value network is a neural network with weights θ s.t. $v(s, \theta) \approx v_\pi(s)$. A policy network on the other hand is a network that helps us with the task of mapping states to actions. Given network weights θ we have $\pi(a|s, \theta) = \Pr\{A_t = a | s_t = s, \theta_t = \theta\}$. So the policy π is actually a distribution over actions. This policy will give you a probability of taking an action from a given state or board position. So this means our policy is stochastic it does not give us a single action to take.

The approach first introduced by deepmind [2] made use of expert data. We will not go into detail here about their approach but I want to highlight at least the intention of that paper. Assume we have an perfect oracle. That is we have some function $f(s) \rightarrow v$ that can tell us the true value of being at any possible state? Then we are almost done and can happily do a single pass through the search space to obtain optimal path. At each node starting from the root node of the tree we just take

the action that gives us the highest value. We can use our oracle for this by looking at each possible action and seeing which one leads to a state with a higher value. Typically we would have to know something about our opponents policy but here we can plan by acting as both players. We can just assume that our opponent has access to the same oracle and will play perfectly. This can be summed up with the following bellman equation. $a = \underset{a}{argmax} \sum_{s',r} p(s',r|s,a)[r + \gamma f(s')]$. This equation is a form of the bellman optimally equation. Lets break down each part. We are interested in finding the action that gives us the highest return. $\sum_{s',r} p(s',r|s,a)$ simply means that we are going to look at each of the possible next states assuming we have taken a specific action from our starting state s . This will gives us a probability of each one of those states which we will use to weight the second part of the equation by. The second part $[r + \gamma f(s')]$ is the return that we expect to get This assumes you have knowledge of the environment (i.e. you can compute $p(s',r|s,a)$). This is an assumption made by MCTS and by the AlphaGo algorithms. Since we dont have an oracle we can start thinking about other ways to approximate $f(s)$. Most ways that have been tried historically are taking advantage of expert knowledge. This can be in the form of explicitly hard coding a way to evaluate the value of a state. This has been done in games as complex as chess. You might do something naive like assign a value to each piece. Then count up all the pieces that each side has and assign them a score.

You could do as more advanced chess programs did and create a database of positions with the actions that were taken by experts at each position. Then you could create a lookup table or another way to approximate a good policy function $\pi^*(s) \rightarrow a$. This second approach is what the first version of AlphaGo did and we

will now look at this more deeply. Assume we have created a database of expert games (or other environments). So we have an unordered list of tuples of positions, actions and outcomes. that were played during actual games. That is we have $(s_0, a_0, z_0), (s_1, a_1, z_1), (s_2, a_2, z_2), \dots, (s_n, a_n, z_n) = \mathbf{D}$ where $s \in \mathbb{R}^d, a \in \mathbb{R}^{|A|}$ and as stated before $z = \{-1, 1\}$. There are n samples in our database, each state is a vector of size d and each action is of size $|A|$ the possible number of actions. We will assume for convenience that d is the dimensions of the environment. If we were playing chess than the board could be represented by a vector of size $8 \times 8 = 64$. So the set of legal actions will be something ≤ 64 but the state will most likely be $\geq d$. For instance we will see that Deepmind used something $> d$ to represent state. We can sample from the expert $(s, a) \sim \mathbf{E}$ where s is some state and a is the action taken by the expert in that state. Sampling from $\mathbf{E}$ will be done uniform randomly s.t. there is no correlation from one sample to the next to help avoid over fitting concerns. We can then employ any of a number of Supervised Learning techniques to learn a policy function $\hat{f} : s \rightarrow a$ by sampling a large number of data points from $\mathbf{E}$ and minimizing some loss function $L(\hat{f}, \mathbf{E})$. We can do a similar thing to learn a value function by sampling state outcome pairs from the database as well $(s, z) \sim \mathbf{E}$. The loss function used for the policy network is cross entropy loss and for the value network it is mean squared error. For AlphaGo Fan the first AlphaGo algorithm this was just the seeding step. Once you have your expert networks than you use self-play and MCTS to further improve upon them. There are many details that go into AlphaGo Fan and I encourage the reader to go read the paper if interested [2]. The main thing I want us to think about here is that we are able to use generated data to reduce the depth and breadth of our search. Recall that in MCTS we select actions by

$$a = \underset{a}{\operatorname{argmax}}(Q(s, a) + U(s, a)) \quad (4.1)$$

In AlphaGo Fan the breadth of the MCTS is reduced by using the policy network to help estimate the uncertainty term $U(s, a)$.

$$U(s, a) \propto \frac{P(s, a)}{(1 + N(s, a))} \quad (4.2)$$

Where $P(s, a)$ comes from the policy network. Reducing the depth is done by replacing the rollout step in MCTS by a rollout network trained on the expert data as well as a value network to help in estimating $Q(s, a)$

$$Q(s, a) \propto v(s, \theta) \quad (4.3)$$

AlphaGo Zero actually makes the architecture much simpler. We will now turn to looking at that and again the reader if interested is encouraged to look at [2] for a more thorough reading of AlphaGo Fan.

4.2 Zero Knowledge

I first want to highlight some of the main differences between AlphaZero and past implementation like AlphaGo Fan.

- AlphaGo Zero is trained through self-play only. Starting from random play. There is no expert data used whatsoever.
- The input features are only the black and white stones from the board.
- It uses a single neural network to represent both the policy and values network.

- It uses a simpler tree search that relies on the single network. It manages to avoid the need for rollouts.

A big change in accomplishing the above was to move the lookahead search inside the training loop. In AlphaGo Fan the networks were trained first and then those networks were used to run MCTS. In AlphaGo Zero we will see that the network is trained during MCTS.

The network in short takes in the current board representation and history and outputs a vector of move probabilities and a value. $(p, v) = f_{\theta}(s)$. p gives the probability distribution of selecting action a in state s . $p_a = Pr(a|s)$

The below diagram gives a nice summary of the training pipeline. Lets look at part **a** and **b** separately.

Figure 4.4: AlphaZero training pipeline

a. Self-play For each state s_t in an episode (game) MCTS is performed and an action is sampled from the move probabilities it outputs. So we can think of MCTS as a function $\alpha_{\theta}(s_t) \rightarrow \pi_t$ that takes in the current state and outputs the policy at that point in time. An action is then selected $a_t \sim \pi_t$ and executed against the environment. We repeat this process until we reach a terminal state and the game

ends. We receive a reward from the environment $r_T \in (-1, 1)$ which we can then use to train the network.

b. Neural Network Training From part **a** we have a series of states and outcomes that we can use to improve the network. A single iteration of MCST gives us a series of data points like the following $(s_1, \pi_1, z_1), (s_2, \pi_2, z_2), \dots, (s_T, \pi_T, z_T)$ where $z_t = \pm r_T$. We can now use these to train the network. We will get into the details of the network architecture later but for now it will suffice to know that the loss function is minimized using SGD with the following loss $l = (z - v)^2 - \pi^T \log p + c||\theta||^2$. So the loss has three parts. MSE between the network's value prediction and the true outcome, a cross-entropy loss between the distribution output by MCTS and the distribution given by the network, and L2 weight regularization.

This process can be viewed as an attempt to make the network more closely resemble the outcome that is being given by MCTS. In this way MCTS acts as the guide for continual improvement. The search probabilities π are generally going to give better action selections than what the current network advises. So MCTS acts as a policy improvement operator. Also the outcome of games $\mathbf{z}$ were obtained using MCTS and so they can be viewed as a policy evaluation operator. If you have a way to both improve a policy and evaluate the new policy then you have a way to perform policy iteration.

Ok now we will dig into the MCTS a bit more and then take a look at how the network is trained after that. The below diagram taken from the paper displays each of the steps. Let's go through each

Figure 4.5: AlphaZero training pipeline

Overview Each **edge** in the tree stores a set of statistics. For all legal actions from a given node $a \in A(s)$ the set $\{N(s, a), W(s, a), Q(s, a), P(s, a)\}$ is stored. $N(s, a)$ is the visit count, $W(s, a)$ it the total action-value, $Q(s, a)$ is the mean action value, and $P(s, a)$ is the prior probability given by the network for the (s, a) edge.

The algorithm starts at the root node of the tree and then repeats steps (a-c) a given number of times. After these simulations are complete an action is selected based off of the accumulated statistics. The current node of the tree is moved to the next node based off of the action that was selected. The process is repeated for the new node until a terminal node is reached. The results are stored in order to train the network and the process is repeated from the root node once again.

Selection As with all the versions of MCTS we have looked at so far selections are made by taking actions by taking the max of a value term plus an upper confidence bound term. $\underset{a}{\operatorname{argmax}}(Q(s, a) + U(s, a))$ In this case both terms come from the network. $U(s, a) \propto P(s, a)/(1 + N(s, a))$ where $P(s, a)$ is given by the probability distribution over action $\mathbf{p}$ from the network. $Q(s, a) = \frac{1}{N(s, a)} \sum_{s' | s, a \rightarrow s'} V(s')$ So $Q(s, a)$ is just the mean over the value of the next states

Expand and evaluate Once a leaf node is reached the leaf node is expanded and the leaf node is evaluated by the neural network. $(P(s_L), V(s_L)) = f_\theta(s)$ Then each edge is initialized with the following $N(s, a) = 0, W(s, a) = 0, Q(s, a) = 0, P(s, a) = p_a$. The value of the leaf node v is backed up.

Backup For each step that was visited from step (a,b) the statistics for each of the visited edges are updated. $N(s_t, a_t) += 1$, $W(s_t, a_t) += v$, and $Q(s_t, a_t) = \frac{(s_t, a_t)}{N(s_t, a_t)}$

Play After a given number of iterations from the current node an action is selected and executed against the environment. The policy that is output by MCTS is given by the following formula. $\pi(a|s) = \frac{N(s,a)^{\frac{1}{\tau}}}{\sum_b N(s,b)^{\frac{1}{\tau}}}$

τ is called a temperature parameter that controls the level of exploration. For each time step the statistics in the tree below the node of interest are retained. The statistics above are discarded and will be repopulated in the future if that node is revisited.

We will now turn to focusing on the deep learning aspect of the algorithm. We have already discussed how data is collected and what loss function is used while training. Lets start with what the input looked like and than look at the architecture of the network itself. The input to the NN is a 19x19x17 image stack. There are 8 images for the current player and 8 for opponent and The final layer being a representation of who is to play. So the feature input at time t is $s_t = [X_t, Y_t, X_{t-1}, Y_{t-1}, \dots, X_{t-7}, Y_{t-7}, C]$. Here X_t is a 19 x 19 frame of ones and zeros. The ones representing the position of all of blacks stones at time t . Likewise Y_t is the same except all ones for whites stones and zero elsewhere. C here is a 19 x 19 frame representing the player to act at time t . Its a frame of all ones if it is black to act and all zeros if it is white to act.

As mentioned previously the network used in AlphaZero is a two headed network.

There is a single convolutional layer that processes the input followed by 39 residual blocks. Then the policy head consists of a convolutional layer, batch norm, relu and a fully connected layer. The value head is similar but with an extra convolutional layer and the output contains a *tanh* function which works well for two player games since that constrains values to be between -1 and 1. They found that residual blocks performed better than convolutional layers. I think for understanding the core concepts it is not necessary to fully understand how all of the pieces of the deep learning architecture work. In the next section we will look at what it takes to build a smaller example of the architecture on your own. We will also look at some experiments that I ran using the game of ticatactoe and connect4 to help clarify some of the concepts discussed so far.

5.0 Experiments

It is one thing to read a paper and understand something at a high level and it is another to actually implement the concept described. Here I will discuss my own efforts and struggles in implementing AlphaZero on smaller environments. I think it good to clear up some terminology confusion that can arise. When I say that I implemented AlphaZero and what most people mean when they talk about AlphaZero is the general concepts used to solve the game of Go. So this means that the specific hyperparameters or network architecture is not necessary implement your own version on a different environment. I will recap here what those core concepts of AlphaZero are.

Algorithm 1: AlphaZero Core

```
init Network;  
  
init ReplayBuffer;  
  
for iter in iters do  
    run SelfPlay  
    run TrainNetwork  
    run Eval  
end
```

In Algorithm one above you can see that the algorithm can actually be condensed to a small number of parts. Each one of those parts has a lot going on but it helps to have a high level understanding of the algorithm before getting into the details. Let go through each piece. The first thing that happens is the network is randomly initialized. We refer to both policy and value networks as a single network. The algorithm is run for a predefined number of iterations. You could also have a stopping

criterion if you wanted based on performance. For each iteration of the algorithm we do three main things. Self-play, network training and evaluation. The self play step is what generates the data for our network to train on. We play some predetermined number of games against ourselves. The results of those games are saved off for training the network. Playing against yourself here means that the role of player 1 and player 2 is assumed by the same AlphaZero agent and therefore the same network. Training the network means random sampling a batch of data from the replay buffer some number of epochs. The network is updated each epoch using backpropagation to minimize the loss function previously mentioned.

$$l = (v - z)^2 - \pi^T \log \hat{\pi}_\theta + c||\theta||^2 \tag{5.1}$$

The final step is the evaluation step. In the AlphaGo Zero paper this was performed by keeping track of a best network. You keep a best network in memory and then after every time you update the current network you have your newly updated network play against the current best. If the current network beats the current best by a certain threshold then the current network is now set the best network otherwise the current network is discarded. So in this scheme you dont actually keep your updated network until it is able to beat the previous best version. In my own implementation I found that this hurt performance and I opted to continually update the network each iteration. This is actually what they do in their follow up paper

AlphaZero [Sil17b]. Lets look at self play in a little more detail.

```
Def SelfPlay(Agent, NumberGames):  
    for game in Games do  
        reset Tree;  
        reset Env;  
        while True do  
             $s_t \sim Env$   
             $a \sim MCTS(s)$   
             $s_t, a, s_{t+1}, r, done \sim Env.next(s)$   
            Agent.StoreMemory( $s_t, a, r$ )  
            if done then  
                | break;  
            end  
        end  
    end
```

A few things of note in the SelfPlay algorithm above. The first is that at the beginning of each game the tree is being reset. When the agent runs MCTS to select an action all of the statistics above the current node in the tree are forgotten about. It is possible to have different criterion for how often you reset the statistics in the tree but the idea is that old statistics were generated by an old network and therefore not as reliable. We will discuss specifics of MCTS and the different hyper parameters

used later. We next give a brief look at the network training step.

Def TrainNetwork(*Agent*, *Epochs*):

```
for epoch in Epochs do
     $s, \hat{\pi}, z \sim \text{Agent.Memory}$  Sample data from replay buffer that was
    created during self play
     $(\pi_\theta, v) \sim \text{Agent.NN}(s)$  Pass  $s$  through the network
     $l = (z - v)^2 + \pi^T \log \hat{\pi}_\theta + c ||\theta||^2$ 
     $\theta_t = \theta_{t-1} - \alpha \frac{\partial l}{\partial \theta}$ 
end
```

5.0 TicTacToe Experiments

Now that we have a reasonably good idea of what the high level logic looks I believe you should have a good idea of how the algorithm works. Lets look at a few experiments to try and get a better feel for how the algorithm works in practice. These first few set of experiments we will be running AlphaZero on the game of tictactoe. Tictactoe is played on a 3x3 grid. The players place pieces onto the grid with goal of connecting 3 in a row. You can connect 3 in row horizontally, vertically or diagonally. The game is nice because it is small enough that we can actually find optimal solutions. I will use the same neural network architecture for all of the experiments that follow.

NN Architecture

- Convolutional Layer with padding + BatchNorm
- Convolutional Layer with padding + BatchNorm
- Convolutional Layer + BatchNorm

- policy head is just a single fully connected layer with a log softmax as an activation function
- value head is also a fully connected layer but with *tanh* as the activation function

You can see that there are no residual connections used here. This could certainly be added for improved performance but the goal here was to get use something small but still had some of the properties of what a larger application would need. I did for instance use just a simple Neural architecture that only had fully connected layers and the performance seemed to suffer quite noticeably. If the reader is interested in attempting different architectures please go the github page linked to in the abstract. It is quite easy to change if desired. I used pytorch for building the network and for training the network. Pytorch [\[fac\]](#) is an open source machine learning framework that is supposed to be used for "research to production". The VikingZero codebase does not depend heavily on the framework used and would not take much effort to swap out pytorch for tensorflow or something else.

In the AlphaGo Zero paper the input to the network is an actual stack of frames. This is nice because it automatically encodes some temporal knowledge. In the VikingZero implementation I dont look back any time steps. So the input is a single frame for each player and then a layer representing whose turn it is. So here is what that looks like.

Figure 5.1: Left Figure is a tic-tac-toe board. Figure on the right is the transformed version

Now let's discuss the relevant hyper parameters that require tuning. In the AlphaGo Zero paper they used Bayesian hyper parameter optimization to tune their network. I did not have that luxury and so I tuned the network "by hand". That basically just means you start with a best guess at an architecture, run an experiment, view the outcome, change a single hyperparameter and repeat. AlphaZero has a lot of hyperparameters which makes tuning it rather cumbersome. The list below is quite long but I hope that it helps demonstrate that it is not as simple as aiming the algorithm at a problem and hitting go and expecting good results.

Exploration parameters These are the hyper-parameters that changed how much exploration was done throughout training. First recall that actions during MCTS are selected by the following equation.

$$a_t = \underset{a}{\operatorname{argmax}}(Q(s_t, a) + U(s_t, a)) \quad (5.2)$$

$$U(s_t, a) = c_{puct} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)} \quad (5.3)$$

More weight is given to $U(s, a)$ when $N(s, a)$ is small relative to $\sum_b N(s, b)$. That is when the action a associated with node s has not been visited often relative to

the other actions. More weight is given to $U(s, a)$ when the policy network $P(s, a)$ is larger. That means that the more opinionated the policy network becomes more likely it is to select that action during MCTS. This is a potential area for over fitting.

- c_{puct} This simply weights $U(s, a)$. Increasing c_{puct} increases exploration. I found that the commonly used $\sqrt{2}$ was best.
- **dirchlet noise.** This simply adds extra noise to the root of the search tree during MCTS and encourages additional exploration. This is **Not** used on any other nodes of tree other than the current root. Actions are then selected by $P(s, a) = (1 - \epsilon)p_a + \epsilon\eta_a$ where $\eta \sim Dir(\alpha_d)$ In my experiments I found that $epsilon = 0.1$ and $\alpha_d = 0.03$ (same as paper) was best. Epsilon $\downarrow$ 0.1 tended to cause too much variability in the action selected by MCTS resulting in large variability during network training.
- τ threshold. In the AlphaGo Zero paper actions against the actual environment are selected according to $\pi(a|s) = \frac{N(s,a)^{\frac{1}{\tau}}}{\sum_b N(s,b)^{\frac{1}{\tau}}}$ for a given number actions. After that given number of actions the action with the highest probability is selected. This translate into setting $\tau = 1$ during the beginning of play and then setting $tau \rightarrow 0$ once the agents has taken more actions than the threshold. I found that having the threshold = 3 in tictactoe worked well.

Network Parameters

Now lets look at the hyperparameters w.r.t network training.

- **Optimizer.** Throughout these experiments I use the Adam optimizer provided by pytorch. In the paper they use SGD with momentum. I found that Adam outperformed for all of my experiments.

- **lr** learning rate. The learning rate that I used was 0.001
- **Batch size**. This is how many samples are randomly sampled for each epoch of training. I used a batch size of 32 but found that sizes between [16,64] gave similar results.
- **epochs**. See the **TrainNetwork** function above. Running 25 epochs worked best for me but this seemed like a bit large for a game like tictactoe but thats what worked best for me.
- **Maxmemsize**. How much memory you allow to be kept in the replay buffer can have a significant affect on performance. For instance I kept a max size of 20k for a long time and then realized that this kept too much old data and did not allow the network to correctly capitalize on a newly "found" strategies.

More Parameters..

- **number of sims**. This refers to the number of times you run an iteration of MCTS from your root node. I found that running 25 simulations worked well. It was enough simulations that it gave your policy network a good target. Remember that running MCTS outputs $\hat{\pi}$ with the policy network tries to approximate. So if you dont run enough simulations $\hat{\pi}$ wont be very good. On the other hand running a lot is very costly computationally. So ideally you want the least number that is necessary to continuously be able to improve your policy network.
- **number of games**. The number of games that you play (see Def SelfPlay) per iteration changes how much data that you end up generating. Generally in

deep learning a lot of data is required. It is no different in this setting. If you dont play enough games than you wont have enough data to train on but if you play too many games you can end up overfitting to the current network. For tictactoe I found that playing 30 games was best.

Evaluation

For evaluating the performance I used a few different metrics. The first metric was to play the agent vs a minimax agent and track the total score over iterations. For tictactoe this allowed me to get a comparison against an optimal agent. For Connect4 a minimax or full depth alphabeta agent was too slow. I used a depth limited alphabeta agent. Please see the Search section for a discussion on the alphabeta algorithm. The loss values for the policy and value networks are tracked every iteration. This is mostly to ensure the algorithm is actually learning. As we will see having a small error does not translate to winning. Another metric I used exclusively for the tictactoe experiments was an optimal play lookup. I used a minimax agent to look at every possible state and create a dictionary lookup of state $\rightarrow$ action. The size of the dictionary is 4520. This ignores illegal positions and positions where the board is full since in that case there is no action to look at. To compare against the current agent I do an exhaustive comparison of the agents max actions vs the minimax action. I then report the mean score. This is an expensive test and so running it every iteration is not feasible but it gives a thorough look at the agents performance.

5.0 TicTacToe Experiments

In these series of experiments we will show our best models performance and the hyperparameters used. We will then do a comparison vs other settings and do a side by side comparison.

Base model settings

- Batch size: 32
- c_{puct} : 1.41
- dirchlet noise: 0.03
- epochs: 25
- epsilon: 0.1
- learning rate: 0.001
- max memory size: 5000
- number simulations: 25
- tau threshold: 3
- episodes: 300
- number training games: 30

Figure 5.2: Base Case experiment

For the base case experiment looking at the figures left to right. `tot_p1_wins` is the total score out of 5 games. The agent gets a +1 for a win a -1 for a loss and 0 for a draw. Since minimax plays perfectly the best the agent can hope to do is get a score of 0. We can see that this is achieved fairly quickly from the perspective of player 1. From the perspective of player 2 we can see from the second figure that eventually the agent reaches a point of always getting a draw vs minimax but is much

more volatile at the beginning of training. The

Experiment 1

In our first comparison we will look at the affect of the tau threshold on performance. Lowering the threshold equates to less variability in the data seen by the network. This in turn leads to less exploration in future iterations of training as the network learns to more quickly prefer those higher probability actions. This can lead to over fitting if there is not enough exploration in the MCTS portion of the algorithm. Below we show our base model in comparison with a model that has $\tau = 1$ instead of 3. In this case there does seem to be over-fitting happening. We see from the loss graphs that the network quickly goes to having almost 0 loss. This would be good if we were doing supervised learning but here in the RL setting that does not always translate to a good outcome since we are generating our own dataset from interacting with the environment. The most striking difference I think here is looking at the "mcts_minimax_score". This shows how the $\tau=1$ model quickly hits max performance and plateaus. We can also see that in performance vs minimax the $\tau=1$ model was bouncing around quite a bit more.

Figure 5.3: Base vs tau = 1

Experiment 2

In this second comparison we will look at the affect of increasing the amount of noise during search by setting $\epsilon = 0.2$. Recall that this increased noise happens only at the root of the search tree during MCTS. It seems that this model actually performs slightly better than our base model in the "mcts_minimax_score" but quite a bit more variability in its performance vs minimax. Not suprisingly the value and policy loss values are much higher. This is not surprising since we have a lot of variability in our training data. This highlights the point that increased exploration even if it comes at the cost of worse network performance might be worth it.

Figure 5.4: Base vs epsilon = 0.2

Experiment 3

In this third comparison we will look at the affect of decreasing the number of simulations performed during search. This really hurts performance more than any other metric. This was suprising to me given such a small environment such as tictactoe. It appears that even in non complex domains that a minimum amount of exploration is required and 15 simulations was not enough to both explore and exploit the better paths.

Figure 5.5: Base vs nsim = 15

Experiment 4

In this fourth comparison we will look at the affect of increasing the number of games played per episode as well as increasing the maximum memory size. Here we set the max memory to 20000 and we play 50 games per iteration. This of course generates much more data than previous and allows us to keep more in memory. This should help to prevent over-fitting. Playing more games equates to seeing more board positions especially when the network is less certian about withc is best. Here we found that we did not increase performance and might have even hurt performance slightly. From the experiments that we ran it seemed however that this is something that will require a good amount of tuning. Large domains of course will require much

more training data and will require you to hold on to older data for longer as well.

Figure 5.6: Base Case vs maxmem size

Experiment 5

Lets now look at an interesting experiment in which we increase the number of epochs the network runs. In the base case we run 25 and so what should doubling them do? My first guess was that it was going to over-fit and I am sure there is some high number of epoochs that you could run that would cause that to happen but I found that doubling the number of epochs actually seemed to help performance. The player 1 performance vs minimax immediately goes to optimal and stays there for the rest of training. The policy and mcts minimax scores are both better as well.

Figure 5.7: Base Case vs Epochs = 50

5.0 Connect4 Experiments

Lets now look at how our implementation does on the game of connect4. In these tests we were not able to use the minimax scores that we could do with tictactoe since we needed a dictionary action lookup over the full state space. So we will drop those tests. The game of connect 4 is the same as tictactoe expect the goal is to get 4 in a row instead of 3. It is played on a 6 x 7 board.

Parameters

- Batch size: 64
- number simulations: 35
- max memory size: 20000
- tau threshold: 10

- number training games: 20

Figure 5.8: Base Case vs Epochs =50

6.0 Other Work

In this section we will discuss some of the work that has come out since AlphaZero. There have been a number of improvements as well as an increased theoretical understanding. This will not be an exhaustive treatment of the work and is only intended to give the reader a taste of what is out there in the hopes that will make further pursuit of these topics easier. It will not be exhaustive in depth or in breadth. There is more work being produce seemingly all the time that builds on top of what we have discussed in this paper.

6.1 MuZero

Recall that while training a AlphaGo algorithm you need a model of the environment. In order to perform MCTS you have to have an accurate model of the environment that is capable of giving you the next state given a current state and action. The environment also needs to tell you when the game has ended and what the associated reward is. This is a barrier for being able to apply the concepts learned in AlphaGo to other domains. In 2019 Deepmind released a paper [Sch19] that helped to address these issues. The algorithm has many similarities to AlphaZero and so lets look at where it differs. MuZero performs MCTS in a latent space with a learned model. That is as they are performing each step of MCTS in a state space that is not directly representative of an actual state in the environment. Let look at a figure of this to see what I mean.

Planning with a Learned Model

representation $h_\theta(o_1, \dots, o_t) = s^0$

prediction $f_\theta(s^k) = p^k, v^k$

dynamics $g_\theta(s^{k-1}, a^k) = r^k, s^k$

Figure 6.1: Planning in MuZero

The current state of the environment is used for the root node as usual except now there is a **representation** network that takes the environment state and produces a vector representation of that state in a latent space. From this new space you need to run all the normal MCTS except without a model. This is achieved by learning a dynamics model that sits in place of the true environment. The interesting thing is that this dynamics model is not learned by attempting to make it be able to model the true environment more and more accurately which is the standard approach. Here the dynamics model is learned so as to maximize reward (more to come on reward). The idea is that in some cases it might not be necessary and even detrimental to plan with the true model. From the figure you can see that the dynamics function takes in a state action pair and produces the next state and reward. This means that they are also learning a representation for reward. This has the flexibility of potentially learning intermediate reward states. There is still a network that outputs a policy and a value from a given state represented in the figure as the **prediction** f_θ . Since

there is no environment to tell the search when its hit a leaf node they just search for a predefined set of steps. The hope is that the dynamics will start to treat leaf nodes as a sort of steady state and if it is asked to predict the next state past a leaf node then it will output the same reward and state as it was just in. The result of the search is as before with a distribution over possible actions.

The data generated during training is similar as before. The figure below gives more of a birds eye view of the training loop. In part **B** we see that MuZero acts in the real environment in a similar matter as before. MuZero samples from the policy distribution given by the search. The environment than provides the next state which is fed into the search. For training the network a trajectory of state action pairs is sampled from the replay buffer. This is different from AlphaZero that just samples state action pairs randomly and not necessarily as a trajectory. This is necessary because to learn the dynamics model we actually feed in actions from the trajectory. The dynamics network is "unrolled" for K hypothetical steps and the output is used to train the network. As mentioned before there are 3 different networks and that means three different loss functions. They are trained jointly using backpropagation through time. The policy network is made to more closely resemble the output that is given by search. The reward output from the dynamics function is made to more closely match what is given by the true environment. The value function is trained to predict the result of the sample return. The overall algorithm is shown below.

maximization problem that is trying to be solved.

$$\pi_{\theta'} \triangleq \arg \max_{y \in S} Q_{\pi_{\theta}}^T \mathbf{y} - R(y, \pi_{\theta}) \quad (6.1)$$

Q is the same Q function that we have seen in AlphaZero. The subscript just means that it is a value estimate w.r.t a certain policy. $\mathbf{y}$ is a vector that represents the action distribution that we are looking for to maximize the equation. $R(y, \pi_{\theta})$ is a regularization term. Their main claim in this paper is that the selection criterion use by AlphaZero "can be interpreted as approximating the solution to a regularized policy-optimization objective". Some notation will be needed to understand the core concepts of the paper. First they define a multiplier that is used to help simplify some expressions.

$$\lambda_N(s) \triangleq c * \frac{\sqrt{\sum_b n_b}}{|\mathbf{A}| + \sum_b n_b} \quad (6.2)$$

Most of these terms we have seen already. $|\mathbf{A}|$ is the number of possible actions in the given domain and n_b is the number of times a particular node has been visited. c is the *puct* constant. This allows them to rewrite the MCTS select criterion.

$$a^* \triangleq \arg \max [\mathbf{q} + \lambda_N \frac{\pi_{\theta}}{\hat{\pi}}] \quad (6.3)$$

Here $\mathbf{q} = Q(s, a)$ and $\pi_{\theta} = \pi_{\theta}(a|s)$ the policy distribution from our network and $\hat{\pi}$ is the policy distribution given by MCTS. They show that $\hat{\pi}$ is an approximation to a solution ($\bar{\pi}$) of a regularized policy optimization problem.

$$\bar{\pi} \triangleq \arg \max_{y \in S} \mathbf{q}^T \mathbf{y} - \lambda_N KL(\pi_{\theta}, y) \quad (6.4)$$

Here the regularization term is KL-divergence and is in reverse of how it is normally written. They show that $\hat{\pi}$ actually tracks $\bar{\pi}$. They also show how the MCTS UCT that we looked at early in this paper is actually approximating a similar $\bar{\pi}$ but with a different regularization term. They then describe why just calculating $\bar{\pi}$ directly is preferable. Since $\bar{\pi}$ can be substituted in place of $\hat{\pi}$ you can just swap out one for the other in any place that it is used. In AlphaZero $\hat{\pi}$ is used in three places. It is used in selecting actions, during search and also when training the network.

To demonstrate the applicability of their approach they ran a series of experiments using MuZero as a baseline. The first figure below helps demonstrate that MuZero is indeed approximating their algorithm and also the large impact that the number of simulations has on performance for MuZero. "All" in this figure stands for the fact that they swapped in $\bar{\pi}$ for each one of the three steps that were mentioned above. We can see that especially for a small number of simulations "All" vastly outperforms MuZero but they converge to equal performance as the simulations are increased.

Figure 1. Comparison of the score (median score over 3 seeds) of MuZero (red: using $\hat{\pi}$) and ALL (blue: using $\bar{\pi}$) after 100k learner steps as a function of N_{sim} on Cheetah Run of the Control Suite.

They performed experiments a number of Atari games and seem to demonstrate

quite definitively the performance gains achieved using their method. This paper is quite impressive and well worth your time to go and read it if you want to have a deeper understanding of AlphaZero.

Figure 3. Comparison of median score (solid lines) over 6 seeds of MuZero and ALL on four Atari games with $N_{\text{sim}} = 50$. The shaded areas correspond to the range of the best and worst seed. ALL (blue) performs consistently better than MuZero (red).

6.3 Other work and applications AlphaZero

I will just point the reader at a few papers that extend AlphaZero or use it in a novel way. There is many more I am sure at this point as the literature continues to expand at a rapid pace in this area.

- AlphaZero is intended to work in domains in which the action space is discrete. Extending the work to continuous actions space would greatly help in being able to use it in "real" world applications such as robotics. **A0C: Alpha Zero in Continuous Action Space** [Moe+18]
- Improved protein structure prediction using potentials from deep learning [Dee20]
- TOWARDS "ALPHA CHEM": CHEMICAL SYNTHESIS PLANNING WITH TREE SEARCH AND DEEP NEURAL NETWORK

POLICIES [Seg+17]

- Optimizing quantum dynamics **Dalgaard, Mogens, et al.** "Global optimization of quantum dynamics with AlphaZero deep exploration." **npj Quantum Information** 6.1 (2020). [Dal+20]

7.0 Conclusion

In this paper we have looked at the AlphaGo family of algorithms in the hopes of shedding some light on how the internals work. We went through the hierarchy of concepts presented in the introduction. We have taken the perspective of looking at 2 player zero sum games and how they have historically been solved. We then showed how AlphaGo fits into this history. We started with basic RL concepts, MDP, Dynamic Programming, Monte Carlo Methods, Monte Carlo Tree search, Deep RL, and finally AlphaZero. We discussed a bit about what goes into actually building out your own version of the program. Using our own version we looked at a number of experiments that helped us understand how the different parameters of the algorithm affect overall performance. The AlphaZero algorithm is quite elegant but it requires a tremendous amount of tuning. This tuning cost is prohibitively costly in larger domains if you are on a limited budget. That being said the algorithm has proven to be very successful in the game of Go and in other domains. I hope that this paper has helped you to understand AlphaZero more thoroughly and allow you to explore this interesting work by Deepmind.

Bibliography

- [Rie05] Martin Riedmiller. “Neural fitted Q iteration—first experiences with a data efficient neural reinforcement learning method”. In: *European Conference on Machine Learning*, pp. 317-328. Springer, Berlin, Heidelberg (2005).
- [Mni+15] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *nature* 518.7540 (2015), pp. 529–533.
- [Seg+17] Segler et al. “Chemical synthesis planning with tree search and deep neural network policies”. In: *arXiv preprint arXiv:1702.00020* (2017).
- [Sil17a] et al. Silver David. “The predictron: End-to-end learning and planning”. In: *International Conference on Machine Learning*. PMLR (2017).
- [Sil17b] et al. Silver D. Schrittwieser. “Mastering the game of Go without human knowledge”. In: *Nature* 550, 354-359 (2017). URL: <https://doi.org/10.1038/nature24270>.
- [Moe+18] Thomas M. Moerland et al. “A0C: Alpha Zero in Continuous Action Space”. In: *arXiv preprint arXiv:1805.09613* (2018).
- [Sut18] Richard S. Sutton. *Introduction to reinforcement learning (2nd edition)*. Cambridge: MIT press, 2018.
- [Sch19] et al. Schrittwieser Julian. “Mastering atari, go, chess and shogi by planning with a learned model.” In: *arXiv preprint arXiv:1911.08265* (2019).
- [Dal+20] Dalgaard et al. “Global optimization of quantum dynamics with AlphaZero deep exploration”. In: *npj Quantum Information* (2020).

- [Dee20] Deepmind. *AlphaFold*. 2020. URL: <https://deepmind.com/blog/article/alphafold-a-solution-to-a-50-year-old-grand-challenge-in-biology>.
- [Rus21] Norvig P. Russell S. *Artificial intelligence: a modern approach (4th Edition)*. Pearson, 2021.
- [fac] facebook. *pytorch*. URL: <https://pytorch.org/>.