Fall 2018

Exploring the Effect of Different Numbers of Convolutional Filters and Training Loops on the Performance of AlphaZero

Jared Prince
Western Kentucky University, jared.prince133@topper.wku.edu

EXPLORING THE EFFECT OF DIFFERENT NUMBERS OF CONVOLUTIONAL FILTERS AND TRAINING LOOPS ON THE PERFORMANCE OF ALPHAZERO

A Thesis
Presented to
The Faculty of the School of Engineering and Applied Sciences
Western Kentucky University
Bowling Green, Kentucky

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science

By
Jared Alexander Prince

December 2018

EXPLORING THE EFFECT OF DIFFERENT NUMBERS OF CONVOLUTIONAL FILTERS AND TRAINING LOOPS ON THE PERFORMANCE OF ALPHAZERO

Date Recommended 12/3/2018

[illegible]

Dr. Uta Ziegler, Director of Thesis

[illegible]

Dr. James Gary

[illegible]

Dr. Zhonghang Xia

[illegible] 12/11/18

Dean, Graduate School Date

ACKNOWLEDGMENTS

To Dr. James Gary and Dr. Zhonghang Xia, thank you for taking the time to be a member of my committee. I appreciate your flexibility in rescheduling the defense (twice).

To Dr. Uta Ziegler, my committee chair, thank you for all the advice, guidance, and support you have given me over the past two years working on both this thesis my honors thesis. I could not have gotten this project off the ground without your time and effort, of which you contributed a great deal.

CONTENTS

List of Figures

EXPLORING THE EFFECT OF DIFFERENT NUMBERS OF CONVOLUTIONAL FILTERS AND TRAINING LOOPS ON THE PERFORMANCE OF ALPHAZERO

Jared Alexander Prince December 2018 62 Pages

Directed by: Dr. Uta Ziegler, Dr. James Gary, and Dr. Zhonghang Xia

School of Engineering and Applied Sciences Western Kentucky University

In this work, the algorithm used by AlphaZero is adapted for dots and boxes, a two-player game. This algorithm is explored using different numbers of convolutional filters and training loops, in order to better understand the effect these parameters have on the learning of the player. Different board sizes are also tested to compare these parameters in relation to game complexity.

AlphaZero originated as a Go player using an algorithm which combines Monte Carlo tree search and convolutional neural networks. This novel approach, integrating a reinforcement learning method previously applied to Go (MCTS) with a supervised learning method (neural networks) led to a player which beat all its competitors.

Chapter 1

Introduction

The goal of this project is to investigate and analyze how the effectiveness of the AlphaZero algorithm changes as the number of convolutional filters per layer or the number of training loops per iteration changes. AlphaZero, which became the world's top Go player in 2017, uses a novel combination of neural networks and Monte Carlo tree searches to learn to play Go more effectively than any previous computer player [**11**]. In an attempt to better understand the learning process of the algorithm, we examine two of of it's parameters (number of convolutional filters and training loops) with respect to the algorithm's effectiveness.

In order to investigate the effect of different numbers of filters and training loops, the AlphaZero algorithm is applied to another game, dots and boxes, which shares similar features with Go. Both are two-player, turn based games with discrete moves. They also share some features which make them more difficult, including states whose value for each player is not easily discernible. Dots and boxes, like Go, has an easily variable board size. By using several different board sizes in testing, the hope is that a relationship may be established between game complexity and an effective number of filters and training loops. Such a relationship replace allows generalization across larger problems and decreases the amount of time needed to tune these parameters. This is advantageous when applying the algorithm to a new problem.

To investigate these parameters, an existing AlphaZero program is modified to play dots and boxes. For three different board sizes (representing 3 different game complexities) the program is run using a variety of filter numbers and training loops. For each test, the network loss is collected. Using this loss, as well as the performance of the different network versions produced, the effectiveness of the test run is evaluated with respect to the game complexity and the parameter tested.

Although AlphaZero has thus far been used primarily in gameplay, the underlying principles can be applied to any number of problems. Neural networks are currently used in a wide variety of interesting areas, such as image recognition and generation. The AlphaZero algorithm allows the application of neural networks to domains in which exact training examples are missing, but approximate examples are producible via simulation.

The remainder of this document provides background on AlphaZero, including background information about the techniques used in it's algorithm. Reinforcement learning is discussed in Chapter 2. Detailed descriptions of the Monte Carlo tree searches and neural networks, the two main components of AlphaZero's algorithm, are given in Chapters 3 and 4, respectively. Chapter 5 explains in detail how these components interact in the algorithm. Dots and Boxes is described in Chapter 6. The approach used in this work is described in Chapter 7. In Chapter 8, the results of the tests outlined in Chapter 7 are analyzed. Finally, Chapter 9 draws conclusions from the results gathered, and Chapter 10 outlines possible future avenues of study.

Chapter 2

Reinforcement Learning

The basic idea of reinforcement learning is for an agent to interact with an environment and to learn (through experience) how to replicate the rewards it obtains. An environment is modeled as a set of states. A state represents a discrete position or circumstance within the environment. For instance, in a board game environment, each state might be a configuration of the board. The environment would have a state representing every possible board configuration. An agent is an entity which interacts with the environment. By 'interact', we mean that the agent is able to recognize the current state of the environment (at least partially) and take actions which change the state. The agent has a set of potential actions it can perform, and some achieved reward (which is obtained from the environment) [**12**]. As an agent performs actions, the environment's state changes. Some of the states the agent reaches cause the environment to dole out a reward to the agent. For example in a game environment, a reward may be given if the player reaches a terminal game position. A reward might be positive or negative, depending on whether the agent should learn to seek to reach a state or to avoid a state. The agent itself has no intrinsic goal but to achieve the maximum possible positive rewards. However, the agent uses the rewards to update its action selection approach, and thus learns to meet whatever goal the environment specifies (ie. what produces positive rewards). A successor function, $S(s_i, a) \rightarrow s_{i+1}$ (where s_i is the state at timestep i), determines the next state, given the current state and an action.

More formally, an environment for reinforcement learning can be described by a quadruple (S, A, δ, ρ), where:

- S is the set of possible states

- A is the set of possible actions and $A_s \subseteq S$ is the set of actions which can be applied in state $s \in S$
- $\delta : SxA \rightarrow S$ is the successor function. $\delta(s,a) = s\prime$ says that performing action a in state s leads to state $s\prime$. This function is used by the agent to determine the effect of its action on the environment.
- $\rho : S \rightarrow \mathbb{R}$ is the reward function. $\rho(s)$ is the reward given for reaching state s. This function determines when rewards are given to the agent (and thus determines what the agent learns).

Note that both δ and ρ may be non-deterministic in general reinforcement learning environments. For example, there may be a successor function such that $\delta : SxA \rightarrow (Sx\mathbb{R})^{|S|}$ or $\delta(s,a) = (s\prime, p(s\prime|s,a))$ where $p(s\prime|s,a)$ is the probability that state $s\prime$ is reached when action a is applied in state s. Note that $\sum_i p(s_i|s,a)$ must equal 1. For this thesis, δ is deterministic (the agent always knows the exact result of every action). The reward function is also deterministic: upon reaching a terminal state, the agent receives a reward of 1 if it won the game and a reward of -1 if it lost. No reward is given for ties.

Apart from the agent and the environment, the other principle component of reinforcement learning is a *policy.* A *policy function* π determines, for each state s, which of the possible actions is to be take by the agent. That is, $\pi : S \rightarrow A$ is a mapping from the set of all states S to the set of all actions A. This is what the agent learns over the course of its interactions with the environment. A more general definition, which also works for non-deterministic environments, says that a policy π determines a probability distribution over the possible actions in that state. $\pi : (S,A) \rightarrow \mathbb{R}^{|A|}$, such that $\sum_a \pi(a|s) = 1$ for any given state s, where $\mathbb{R}^{|A|}$ is the set of probabilities and $\pi(a|s)$ is the probability of choosing action a in state s.

Interaction with the environment is discretized into a sequence of equal timesteps and modeled as a series of (s_i, a_i, r_i) triples, where s_i is the state at timestep i, a_i is the action taken at timestep i, and r_i is the reward acheievd by reaching $s\prime$.

In episodic reinforcement learning, the type used in gameplay, learning takes places as a series of episodes, each with the same starting state s_0 and ending with some terminal state s_n, where $n > 0$ [**12**]. The outcome of each episode determines the reward achieved by the agent, and each episode is distinct. Thus, an agent which uses the rewards to learn a policy by maximizing its future discounted rewards learns to take actions which balance the magnitude of a future reward with the certainty of achieving it. The agent attempts to achieve the best possible terminal state and maximize its reward. In the case of games, all terminal states which represent a win for the player are equally good. Since the reward is the same for all such states are equal, the agent learns a policy which picks actions most likely to result in a win.

2.1. Policy Iteration

Usually *policy iteration*, the process by which a policy is updated as the agent learns, repeats two steps: *policy improvement* and *policy evaluation*, until the policy reaches some quality criteria. The improvements to the policy come from the experiences of the agent. The agent takes actions based on its current policy. If those actions lead to a low reward, then the policy is updated to give a lower probability for that action. If this happens enough times, a new action may become the the one chosen by the policy. Likewise, if the actions chosen by the policy lead to high rewards their probabilities increase. The quality of a policy is often measured by the difference between successive policies, with the criteria being met when the difference falls below some threshold. As a policy approaches optimality, the changes in each new policy taper off. When a new policy is reach which has not changed a significant amount from the previous policy, learning is stopped.

Policy evaluation generates a value for each state using the current policy π. The value $v_\pi(s)$ represents the expected value of cumulative future rewards the agent can earn when starting in state s if it follows the policy π at each step [**12**]. Policy evaluation can be done using a variety of methods. For instance, the Bellman equation gives the value of a policy as:

$$v_\pi(s) = \sum_a [\pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]]$$

where γ is a discount parameter. This equation determines the discounted future expected rewards of the policy for a given state s under a policy π. In other words, $v_\pi(s)$ is computed as a weighted average - $\sum_a[\pi(a|s)E[reward(s,a)]]$ - of the probability of an action a being selected $(\pi(s,a))$ times the expected value of the reward achieved when action a is selected in state s $(E[reward(s,a)])$. It is important to note that the Bellman equation only gives the expected result of the policy. It does not show how to improve the policy. In a non-deterministic environment, the expected value of the reward is itself a weighted average over all pairs of possible next states $s\prime$ and r, where $p(s', r|s, a)$ is the probability of earning reward r when reaching state $s\prime$ given that action a was applied in state s. In a deterministic environment (where every action can lead to only one state), the value of the policy is simply the reward collected when transitioning from the initial state to the terminal state using π. Thus, the Bellman equation would be simplified to:

$$v_\pi(s) = \sum_a [\pi(a|s)[r + \gamma v_\pi(s')]]$$

There is an important balance between the magnitude and likelihood of a reward. Naturally, larger rewards are more desirable for the agent, so it may be beneficial to avoid rewards in the short term in order to achieve greater ones in the long term. In environments in which the value of a state it modeled deterministically, the best action for the agent would always be the one which maximizes the reward achieved by an action and the value of the state reached by the action. However, in many environments (especially those in which there are multiple agents in play) a reward that is more distant is less likely to be achieved. It is therefore necessary to find the balance between more immediate but smaller rewards and those which are larger but more distant. In order to do this, future rewards are discounted based on their distance. The cumulative reward is the sum of the current reward and the discounted future reward $(r+\gamma v_\pi(s'))$. Notice that this cumulative reward is the expected reward used in the Bellman equation.

Another method of determining a state's value is to average the results of many episodic experiences in which state s was reached (using π). The difference between this method and Bellman's equation is that Bellman's equation gives a statistically average value for s (based on π), while averaging the experiences gives a value based on actual experience. The experience based value will be skewed from the statistical average (although the more experiences used, the more closely it should match $v_\pi(s)$). The episodic approach is the one used in this thesis.

Policy Improvement uses the value computed by the evaluation method to create a new policy π' which is an improved version of π. The policy for each state can be adjusted greedily such that the action with the highest expected cumulative reward has the highest probability of being selected. Thus, if episodic experiences are used, the new policy reflects the best results achieved in any of the experiences. In later improvements, new experiences result in a new valuation, so the policy is updated again.

To account for fluctuation between evaluations, the probabilities of each action may be adjusted by only a small amount in each improvement, rather than immediately adjusted to reflect the most recent results. If the policy is changed to match only the most recent experience, the agent will never learn more than a single experience can teach and there would be no use for continuous interaction.

Chapter 3

Monte Carlo Tree Search

The Monte Carlo tree search (MCTS) is a common type of episodic reinforcement learning. It is an adaptation of Monte Carlo methods, which use many simulations to predict values which are difficult to precisely calculate. As described in section 2, a MCTS models an environment with which an agent is interacting. In this thesis, the modeled environment is a two-player game and the agent is a player in that game. The MCTS uses reinforcement learning to build a search tree, consisting of interconnected nodes (representing states of the game) and edges (representing actions taken by the player to get from one state to the other). A series of simulations is used, and each simulation builds upon the information collected in previous simulations. The policy derived from the search is of the non-deterministic variety, and consists of a value representing the probability of each action being selected.

3.1. Search Trees

The basic structure used by a MCTS is a tree. A tree is composed of a series of nodes connected to each other by edges. For games, a special type of tree is used called a game tree. In a game tree, each node represents a particular 'state' of the game. A state consists of all information that distinguishes one position in the game from another. For instance, in chess (and many other board games) the state is comprised only of the position of all pieces on the board and which player's turn it is. From the state, one should be able to determine all possible moves a player can make. A full game tree would represent every possible sequence of moves in a game (with each possible game represented as a path from the root node to a terminal node). The edges in a game tree represent the action taken to get from one state to another. Each state in a game tree has a single edge for every possible action and every possible state is represented in the tree. A full game tree for any interesting problem is too large to be explicitly constructed, so generally a partial game tree is used (which is a subsection of the full game tree).

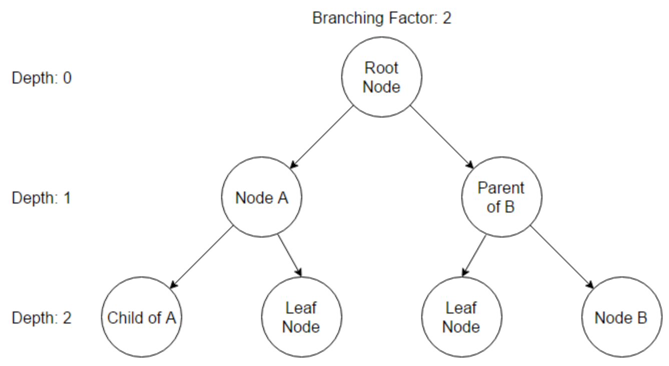


FIGURE 3.1.1. A simple tree example.

Figure 3.1.1 shows a simple example of a tree. The circles represent node and the arrows connecting them represent the edges. If the arrow is pointing into a node, it means that node is the result of taking the action which the arrow represents in the state represented by the node from which the arrow originates. If there is an edge connecting some node X to another node Y, then node Y is said to be a child of X and node X is said to be a parent of Y. The depth of a node is the number of direct ancestors it has, and the height of the tree itself is equal to the depth of its deepest node. A node which has no children is a leaf node, while the node with no parent is the root node. In the case of a game tree, the root node represents the initial state of the game, before any moves are made. The leaves represent terminal state (states at which the game ends).

3.2. Simulation

An MCTS uses simulations to construct a partial game tree. It operates by conducting multiple simulations prior to each move of a game. Before a player using an MCTS makes a move, these simulations will be made, all starting from the current game state. The results are used to determine which move to make in the game. When a move is selected, the state it leads to becomes the root of the tree, and all parts of the tree higher than the new root, or on a different branch, are effectively discarded. This continues until the game ends with result z.

Each node in the tree contains several values. $N(s)$ is the number of times that the node was reached during a simulation. For each possible action a, $N(s, a)$ is the number of times a was selected from the node. Likewise, $Q(s, a)$ is the average reward achieved when action a was selected from the node.

Each simulation has four stages: selection, expansion, playout, and backpropagation. Each node in the tree retains the number of times the node was reached. Each edge retains two value: the number of times the action which it represents was chosen and the total rewards of games in which that action was chosen. From these values, a new value is derived for each action representing how much the player values picking that action. Together, the values of all possible actions from a state are the basis for determining the policy π. MCTS has been shown to converge upon optimal play, given enough simulations [**3**]. Figure 3.2.1 shows the 4 stages of an MCTS simulation: selection, expansion, playout, and backpropagation.

a) In the selection stage, the player traverses the partially built game tree based on the results of previous simulations which had passed through the current node. Starting from the node representing the current state of the game (from which the player will have to make a move), the player compares the average rewards resulting from each possible action and applies a formula to generate an overall value for the action. One commonly used formula, the

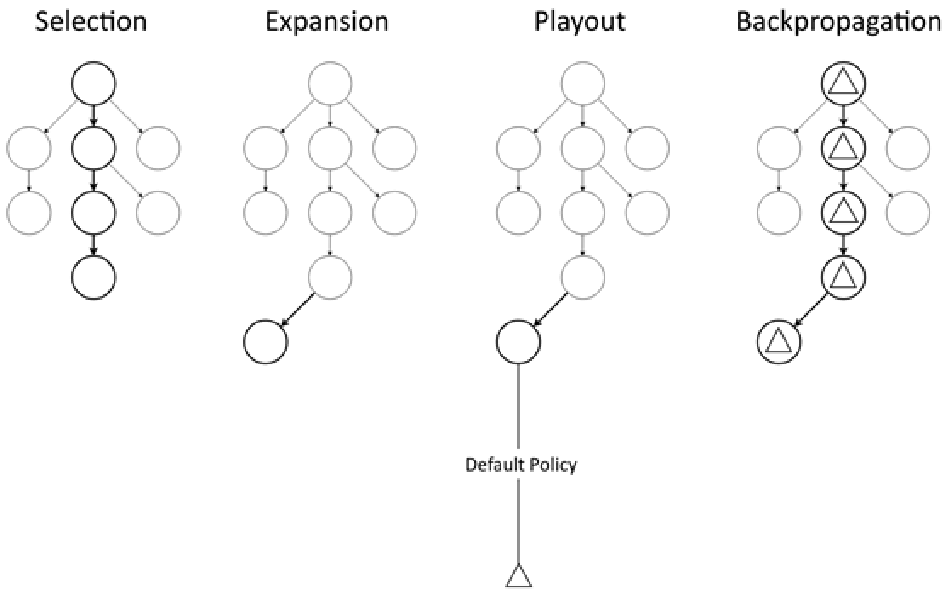


FIGURE 3.2.1. An MCTS simulation.

upper confidence bound for trees, is given as:

$$UCT(s,a) = Q(s,a) + 2c * \sqrt{\frac{2ln(N(s))}{N(s,a)}}$$

where c is a tuning constant [**3**]. Recall that $N(s)$ and $N(s,a)$ are the number of times the node has been reached and the number of times the action was selected, respectively, and $Q(s,a)$ is the average reward achieved by taking the action from this state. The second term is an exploration bonus applied diminishingly to states which been chosen fewer times in the past. This ensures that actions are not discounted after a single attempt results in a loss. An action j is chosen to maximize UCT_j and the player simulates making the action that was chosen.

In section 2.1, the policy for each state is described as a strict probability distribution, summing to 1. In practice, however, the MCTS chooses an action to maximize UCT (or some similar formula), and the sum of the values is likely to be greater than 1. This process continues until the player reaches a node which has yet it be created in the tree (in other words, until it takes an action that has not been taken before). If a terminal node is reached, the simulation proceeds directly to the backpropogation stage.

b) In the expansion stage, a new node is created to represent the new state reached in the selection stage. In some cases, a new node is not created unless the action chosen has been reached previously in another simulation. This is to avoid creating new nodes which are only accessed once.

c) Since a leaf node was created in the previous stages, the playout stage continues outside the bounds of the search tree. Because there is no prior information from this point forward, the player instead uses a default policy (usually random) to pick moves until the simulation reaches a game ending state. At this point, the result of the simulation is found. That result is some value representing the winner (and possibly the margin of victory). For instance, possible results could be {-1, 0, 1}, representing a loss, win, or tie. The result may also be the difference between the winning and losing players' points.

d) In the last stage, backpropogation, the result of the simulation is passed through each tree node traversed in the simulation and used to update the values. For each node, the number of times reached ($N(s)$) is increased by one. For each edge representing an action taken during the simulation, the number of times the action was chosen ($N(s, a)$) is increased by one and the average reward ($Q(s, a)$) is updated based on the result of the game. If the game resulted in a win for the first player, the total rewards resulting from all actions taken by the first player is increased by one, and so $Q(s, a)$ is increased. If it resulted in a loss, the total rewards for each action is decreased by one, and so $Q(s, a)$ is decreased.

Although Monte Carlo Tree Search approaches optimal play, given enough simulations, its effectiveness in large games such as Go is hampered by the massive search space. A major goal of MCTS is to guide the search away from unpromising branches of the tree, but even so, there is simply not enough time to run enough simulations to search even a reasonable fraction of the full tree. A standard game of Go has roughly 3^{361} different states. Thus, although MCTS has, in the last several years, made strides in the field of computer Go [**6**][**5**][**8**], it is not quite powerful enough to succeed against human masters.

Chapter 4

Neural Networks

An artificial neural network (ANN) is a system of interconnected layers of nodes which is given an input and produces an output, with the goal of learning. The weights (which represent the strength of the connection between nodes) vary, and the network learns by adjusting these weights. Hereafter, artificial neural networks are referred to as simply 'neural networks'. One major application of neural networks is classification. Classification is the process of assigning an instance of data to one of a set of pre-defined classes. For instance, the network might be given as input a picture of a handwritten digit, with the goal of the network being to identify which digit was written.

One of the major advantages of neural networks over other techniques is that it requires no information about the task. The only things the network needs are: a structure of nodes and connections (including the weight of each connection) and a set of training examples (each consisting of an input and a matching expected output). The networks evaluates each training example's input and compares the actual output against the expected output. It then adjusts the connection weights based on the accuracy of the example using gradient descent learning, which is described in section 4.5. These adjustments occur over many training loops. In each loop, a batch of training examples are randomly sampled from the training set. These examples are evaluated as a group, after which the weights of the network are updated. The number of examples selected in each batch is the batch size.

There are many different types of neural networks. Some have expected outputs (supervised learning) while others do not (unsupervised learning). The structures of the network can vary wildly. The type discussed in this section (and used in AlphaZero) is a supervised feed-forward network using gradient descent to adjust weights. Sections 4.1 and 4.2 explain in detail the components of networks, while sections 4.3, 4.4, and 4.6 describe specific network designs used by AlphaZero. Finally, Section 4.5 explains how the network weights are updated.

The structure of the network is tailored to the types of learning the network is doing. One network architecture is not suited to all problems. Likewise, the input data must be processed in such a way as to match the network structure. The neural network learning process creates its own features from the available data. A feature is some aspect of the data that can be used to classify it. For instance, if one were to design a program to play chess, features might include the position of the king or how many pawns each player has. Neural networks are particularly beneficial for problems in which the important features of the input are not known. The features the network learns to recognize may be very difficult to describe, and it may not be clear how they relate to the problem.

4.1. Nodes

The component upon which the entire neural network is based is the neuron, or node. A node has a set of inputs $(x_1, x_2, ..., x_m)$ and, for each input, an associated weight $(w_1, w_2, ..., w_m)$. From these it generates a single output. Each input is multiplied with its weight, and the sum of these weighted inputs is the node's activation. This activation value is passed through a differentiable function to generate the output.

The nodes take input from their predecessors, and pass the output to their successors. The output of one node becomes one of the inputs to one or more other nodes. More specifically, the outputs of the nodes are passed through weighted connections which form a one-way link from one node to the next and are introduced as input to the successors. The weight of each connection is w_{ij}, where i is the predecessor and j is the successor. This weight transforms the output of the predecessor to the input of the successor. The activation and output of node j (act_j and out_j) are given as:

$$act_j = \sum_i w_{ij} * out_i$$

$$out_j = f_j(act_j)$$

where i is a node with an outgoing connection to node j and $f(x)$ is the differentiable output function. In other words, the activation of node j is the sum of the weighted outputs of all parents of j. Sometimes the output function uses the previous activation of i or adds a bias term. In simple terms, the output function determines how much of the activation signal the node outputs, and the propagation is how much of the signal from the inputs to the node is received.

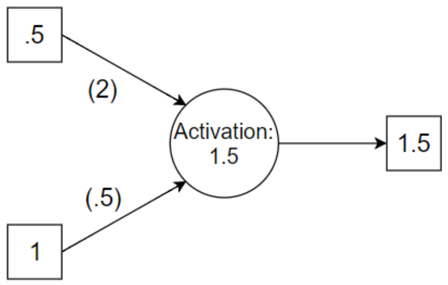


FIGURE 4.1.1. A neural network node.

Figure 4.1.1 shows a single node with inputs from two other nodes. The outputs of those nodes (.5 and 1) are multiplied with the weights of the connection (2 and .5), and the sum is the node's activation (1.5). This node has no bias term, but if it did, its value would be added to the activation. This node has an output function $f(x) = x$, so the output is equal to the activation.

4.2. Structure

In a feed-forward neural network, the entire network consists of a sequence of layers - from the input layer through one or more hidden layers to the output layer. A network layer consists of one or more nodes which share common propagation, activation, output, and weight functions, and which also (generally) have the same position in the linear ordering of the network connections. When visualizing a network, layers are usually depicted as a column of nodes through which all inputs pass before reaching the next layer. Removing any layer would cause there to be no avenue for data to travel from the input nodes to the output nodes. Figure 4.2.1 shows a simple network with two hidden layers The left-most layer is the input layer and the right-most is the output layer. Each layer in the network is fully connected, meaning that every node in the layer has a connection to every node in the next layer.

More formally, a feed-forward network consists of a series of layers $(l_1, l_2, ..., l_m)$, where l_1 is the input layer and l_m is the output layer. Each layer has a set of nodes $(n_{l1}, n_{l2}, ..., n_{lm})$, where l is the layer. The output for each layer is given as $\vec{l}$, a columnar vector. The weights of the layers are matrices $(W_1, W_2, ..., W_m)$, where W_1 is the weight matrix from the input to the first hidden layer. The weight matrix consists of $n_{li} * n_{l(i+1)}$ weights. Thus, each column in W_1 is the set of weights from the nodes in the input layer to a single node in the first hidden layer. Each row in W_1 is the set of weights from a single node in the input layer to all nodes in the first hidden layer. The output of the first hidden layer is $\vec{l_2} = f(W_1^T \vec{x})$, where $f(x)$ is the output function, and T is a transposition.

Each feed-forward neural network has (at least) one input layer and (at least) one output layer, connected by a series of hidden layers. The input layer is the interface by which the user passes data to the network for analysis. Likewise, the output layer is the interface by which the user receives the network's response. In theory, only the

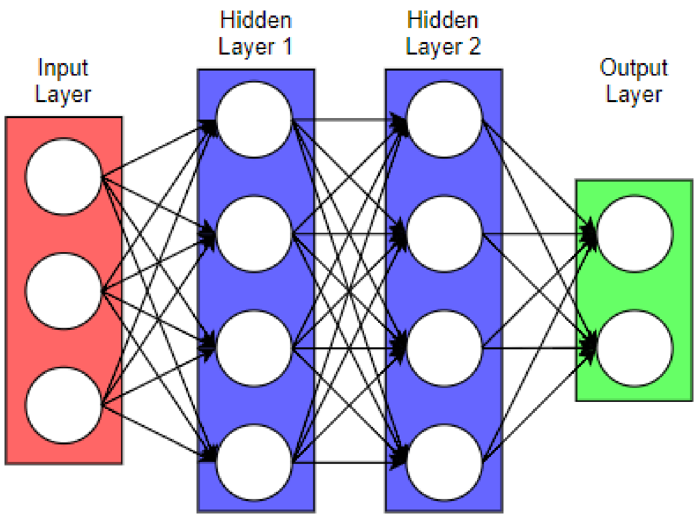


FIGURE 4.2.1. A simple neural network structure.

input and output layers are important to the user, with the hidden layers acting as a black box between the two. Exactly what patterns the network 'learns' to recognize are not known.

Representation of the input is an important step in constructing the network. Whatever input the network takes must be converted into numeric data in some way. For instance, text is a sequence of characters, each of which can be represented by an ASCII character code. So text input can be represented by a series of input nodes which take the ASCII character codes. An image is usually represented with a 2D grid of input nodes in which each node maps to an individual pixel, with the node taking the color code of the pixel. The output to the node may also take a variety of formats. In a network designed for classification, the output layer may consist of a single node representing each class, with the activation of the node representing the probability of the input being an instance of the class represented by the node.

Figure 4.2.2 shows a simple neural network with two inputs (x_1, x_2), a single output (o_1), six nodes $(n_1, n_2, ..., n_6)$, and a weight vector of nine weights (each denoted

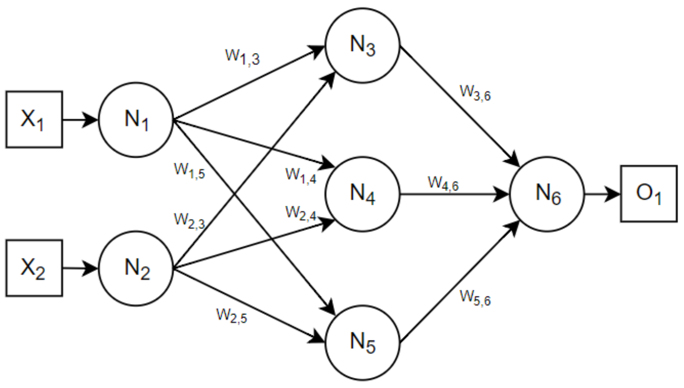


FIGURE 4.2.2. An example network.

as $w_{i,j}$ where i is the node the from which the input is coming and j is the node to which it is going) $(w_{1,3}, ..., w_{2,3}, ..., w_{5,6})$. The output function of each node is a non-linear function. This function is non-differentiable, but it is only non-differentiable at a single point. This function can be computed very fast, and when the a differentiable function is needed the non-continuity is dealt with. The non-linear function is:

$$f(x) = \begin{cases} x \text{ if } x > .5 \\ 0 \text{ if } x <= .5 \end{cases}$$

In other words, activations greater than .5 are propagated forward, while activations less than or equal to .5 are not. Consider a set of weights $w_{1,x} = 1, w_{2,x} = .75, w_{x,6} = 2$. If the input of the network is $x_1 = .5, x_2 = 1$, then the output from node 1 is 0 and the output from node 2 is 1. The input for each node in the hidden layer is .75 $(0 * w_{1,x} + 1 * w_{2,x})$. Then the input to node 6 (the output node) is 1.5 $(.75 * w_{x,6})$ for each node in the hidden layer. So the final output of the network is 4.5.

4.3. Convolutional Neural Networks

A convolutional neural network (CNN) is most often used in image processing, particularly image classification. A CNN consists of one or more convolutional layers.

The goal of a convolutional layer is to learn to recognize patterns in the input data (for instance, a vertical line in an image) using location independent units called filters. Each layer contains multiple filters, and each filter learns to recognize a single pattern, called a feature. Think of a filter as a mini-network which learns to find a feature from a data set smaller than the one used as input to the CNN. The mini-network has an input layer and outputs to a single node. The filter is given as input each possible subsection of the input data and checks if that section contains the feature. Thus, a convolutional layer outputs where (or to what extent) the feature learned by each filter occurs in the input. The feature learned by the filter is not known originally. It arrives organically through the updates. The output by a single filter is a map of the recognized feature in the input. This map is called a feature map.

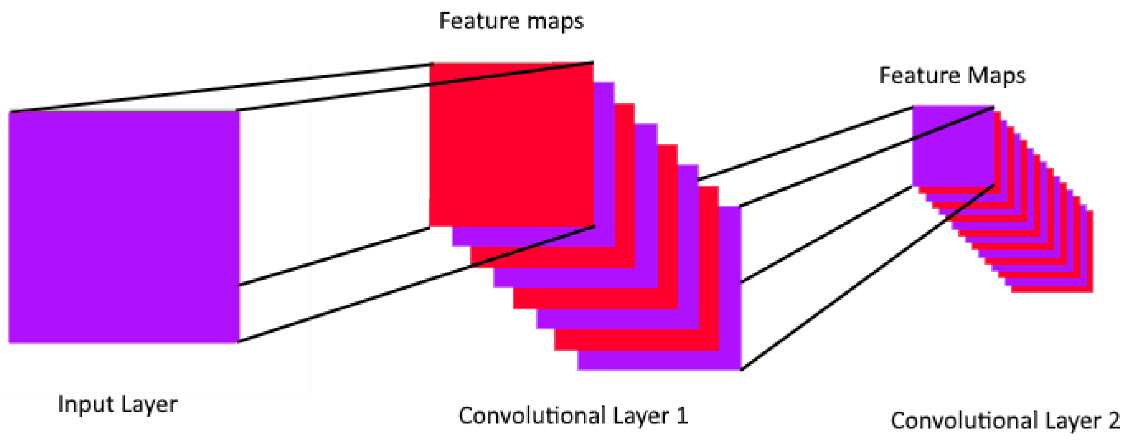
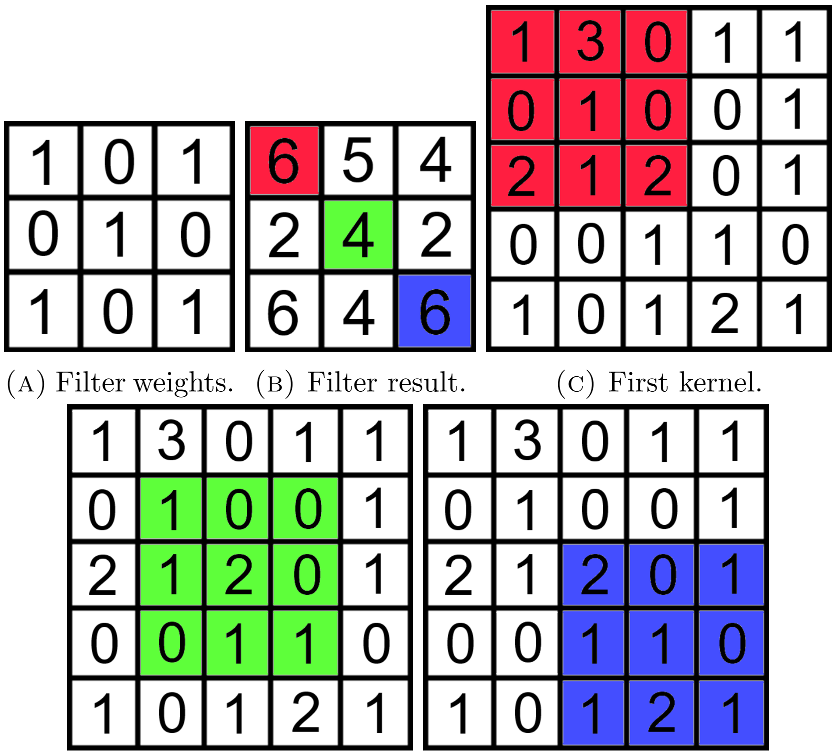


FIGURE 4.3.1. A convolutional neural network.

Figure 4.3.1 is an example of a convolutional network architecture. It shows how a single input layer is pass through 2 convolutional layers with 8 and 16 filters, respectively. An input layer is processed using a set of filters into a feature maps. Each filter is a set of weights which maps the nodes in a $k * k$ subsection of the input (a kernel) into a single node in the feature map. The feature map has a width equal to $w - k + 1$, where w is the input width and k is the kernel size. While the feature map is sometimes smaller than the original input (as shown in Figure 4.3.2), each filter creates a feature map so the output of a convolutional layer is 3-dimensional, and the size of the input for successive layers grows. Feature maps also often have padding around the border to maintain the original input size.

When the input to the network passes through a convolutional layer, a feature map is created for each filter. This stack of maps is the input for the next convolutional layer. So if the original input is of shape $n*m$, each filter of the first convolutional layer produces a 2-dimensional map. The stack formed by multiple feature maps makes the input to the second convolutional layer 3-dimensional. The filters for the second convolutional layer are 3-dimensional as well (with a depth equal to the number of filters in layer 1), but each still produces a 2-dimensional feature map. Because the combination of multiple features in one layer into a single feature in the next, the information represented becomes more and more complex.

In Figure 4.3.2, a $3*3$ filter (the nine weights of which are shown in subfigure a) is moved across the $5*5$ input layer and down to cover every possible $3*3$ subsection of nodes. Each subsection of the nodes to which the filter is applied is called a kernel. The three snapshots in subfigures c, d, and e show the first, fifth, and ninth kernels being evaluated. The highlighted section in each snapshot shows the region to which the filter is being applied in the kernel. The result of the kernel is the sum of the products of each box and the filter weight being applied. For example, in the first kernel, the sum of each box in the kernel multiplied by the weight of the filter at that box is 6. Moving from left to right and top to bottom gives $(1*1)+(0*3)+(1*0)+(0*0)+(1*1)+(0*0)+(1*2)+(0*1)+(1*2)=6$. Thus, in subfigure b, the box representing the result of the first kernel is 6. The sum of the weighted connections in that subsection are mapped to a single value in the feature map. The same weights are applied in each kernel. This allows the filter to find a specific feature in each



(A) Filter weights. (B) Filter result. (C) First kernel.

(D) Fifth kernel. (E) Ninth kernel.

FIGURE 4.3.2. The feature map formed from a single filter.

subsection and combine the results into a layer which gives a map of all occurrences of the feature in the image. This is the feature map. In this example, the stride of the filter is 1, meaning that the filter is shifted by 1 row or column in each move, but there can also be larger strides.

4.4. Residual Neural Networks

Deep neural networks (networks which have a large number of layers) can often be more effective at learning. Intuitively, this makes sense, as more layers allow for more complex operations to be performed on the input data. However, such networks suffer from degradation in accuracy as they converge upon optimal results. As the signal from the network's input passes through more and more layers, it tends to decrease in strength, which makes it difficult for the network to converge. Increasing the depth further only make the problem worse. When a deep network is needed to solve a problem the network must be able to counter this problem. One method of reducing this signal degradation is the use of residual blocks. A residual block is a series of layers in which the input of the first layer is combined with the output of the last layer [**7**]. To be more precise, in a simple network, the output of one layer is used as the input in the next. In a residual network, the output of a layer is combined with the output of a previous layer. The result is then used as the input for the next layer.

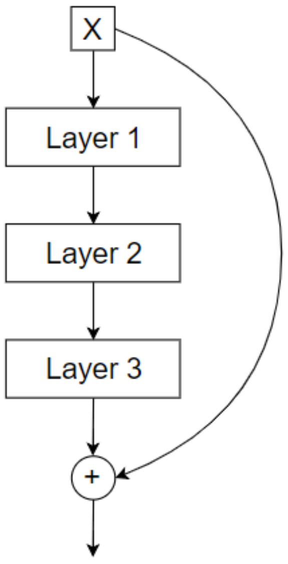


FIGURE 4.4.1. A residual block.

Figure 4.4.1 shows the structure of a single residual block. In the figure, the input is given as x. This input is usually the output of a previous network layer. The input to the network is not the input of a residual block because the block's input must have the same shape of its output and the network's input often does not match the output of its layers. Input x is then pass through the three layers and transformed. We can think of the transformation x undergoes as a function:

$$f(x) = f_3(f_2(f_1(x)))$$

where $f_i(x)$ is the function transforming the input of layer i. Then the result of passing x through the residual block is:

$$f_R(x) = f(x) + x$$

In a residual network, the output of one residual block is usually the input to the next block, so a sequence of residual blocks may be modeled by the function:

$$f(x) = f_{R_3}(f_{R_2}(f_{R_1}(x)))$$

The use of residual blocks in a network, rather than fully sequential layer, allows the signal to propagate through the network more easily because the input signal is reintroduced into every few layers of the network. The authors of [**7**] showed that this approach produced a network which did not suffer from optimization problems and achieved a low training error even with over 1,000 layers.

4.5. Gradient Descent

Gradient descent is a method by which a series of values used for a function are altered iteratively such that the result of the function is minimized. In the case of artificial neural networks, which consist of a series of weighted connections between nodes, a gradient descent learning approach adjusts the weights in such a way as to minimize the error of the network. A network's error is the difference between the target output and the computed output. In other words, the goal is to reduce the difference between the value the network computes for a training example and the target of that example. The weight vector w (all the weights of the output layer) is adjusted iteratively in the direction that would most decrease the error of the training examples used. That is,

$$w_{t+1} = w_t - \alpha' \Delta(E)$$

where E is the mean squared error:

$$\frac{1}{n}\sum_{i}^{n}(\hat{l_{mi}} - l_{mi})^2$$

The formula to adjust the weight is given as:

$$w_{t+1} = w_t + \alpha[\hat{l_{mi_t}} - l_{mi_t}]\Delta l_{mi_t}$$

where w_t is the weight vector at timestep t, α is a learning rate which weights the value of the change, l_{mi_t} is the set of computed values, and $\hat{l_{mi_t}}$ is the set of target values [**12**]. For any function $f(w)$, $\Delta f(w)$ is the vector of partial derivatives with respect to the individual values of the vector [**12**]:

$$\Delta f(w) = (\frac{\partial f(w)}{\partial w_1}, \frac{\partial f(w)}{\partial w_2}, ..., \frac{\partial f(w)}{\partial w_n})$$

In other words, $\Delta v(S_t, w_t)$ is the vector of partial derivatives of v with regard to the weights in w_t. So to adjust the weights, one adds the product of the loss of the examples and Δv, the partial derivative of the value function. In updating the network, the mean squared error is used, rather than the error itself.

The above method works to update the weights of the output layer, but in the hidden layer, there is no direct way of knowing what the output should be (and thus no way of knowing the error). However, the weights can be adjusted in a similar manor by using the error of all weights in the output layer to adjust each weight in the hidden layer. The error produced in each layer is used to update the weights of the previous layer.

4.6. Batch Normalization

The training of deep neural networks is hindered by the tendency of the input to shift it's distribution between successive layers, due to learning. This effect is called internal covariance shift. Because of this tendency, the learning rate of a deep network must be decreased in order to maintain training effectiveness. In order to address this problem, the authors of [**9**] present an approach they call batch normalization. They normalize the training data input in each minibatch, which reduces the internal covariance shift and results in the network performing many times faster. For more details about batch normalization, see [**9**] and [**2**].

Chapter 5

AlphaZero

AlphaZero uses reinforcement learning techniques to learn to play Go at a higher level than any human player, or any other computer player. The player combines two valuable learning methods (Monte Carlo tree search and neural networks) to produce better results than any other current player. These two methods are explained in detail in sections 3 and 4, but a short summary will be given below. Both methods work by interacting with the game environment and 'learning' to play better over the course of many iterations, rather than being pre-programmed with known strategies or simply evaluating every possible move to find the best one. The MCTS does this by simulating games and learning from the results, while the neural network evaluates the difference between the values it predicts for a set of states and the target values for those states. The interaction between these two methods is detailed in sections 5.1, 5.2 and 5.3 below.

As described in the MCTS section, the weakness of a MCTS is that it requires an amount of memory too large to accommodate interesting problems and does not support generalization across similar states, and thus takes an inordinate amount of time to find an optimal (or near optimal) solution for a large search space. In order to determine the expected reward for a given state, many simulations need to be preformed and pass through a majority of the substates. The neural network, on the other hand, requires very little memory, and comparatively little time to train. Its weakness lies in the fact that a prerequisite for training is having a large set of training examples which provide information about which action to select in which state for the network to use during training. In order to produce such examples, one needs to already know a target for the network to learn.

Monte Carlo tree search, the first method utilized by AlphaZero, involves generating a tree (a collection of nodes and edges corresponding to possible game states and actions) to represent the possible moves. This tree is built by sampling the search space with a number of simulated games which is relatively small compared to a standard MCTS. The results from previous simulations inform the moves picked in the current simulation and the move picked in the actual game. Neural networks also involve a series of nodes and edges, but rather than representing a sequence of moves, the network is used to perform complex calculations based on the input (the current state), the result of which is a recommended move. The creators of AlphaZero use a neural network to guide a Monte Carlo tree search. In return, the results of the MCTS are used to train the network.

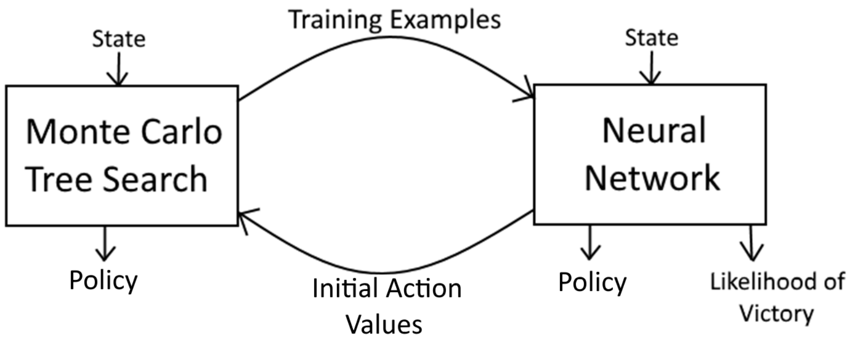


FIGURE 5.0.1. The AlphaZero Architecture.

Figure 5.0.1 show the architecture of AlphaZero. In AlphaZero, the MCTS plays a series of games, approximating the value of each possible action for each state it encounters. The value of each action is initialized with the policy output of a network which starts with random weights. Having learned a value for actions, the MCTS creates a set of training examples from the states and values it learned, to be used by the network in training. This produces a new network which is slightly better than the initial one. A tournament is played between two players, using the original and new networks. If the player using the new network wins, that network becomes the standard. Another set of games is played by an MCTS player using the best network. If the network is better, the values to which it initializes the actions in the tree provide with MCTS with a better starting point for the search, so the values it learns for the actions will be slightly better than in the previous iteration. The new values are used to train the network again. This process repeats indefinitely, allowing the MCTS and the network to feed off each other and slowly learn through this cooperation. The MCTS is able to learn a little bit at a time about the possible actions, while the network is able to generalize and condense the information learned.

5.1. Monte Carlo Tree Search Details

In AlphaZero, many games are played with an MCTS. The pieces of info kept at each edge are $W(s,a)$ (the total reward achieved by picking action a in state s), $N(s,a)$, and $P(s,a)$. $P(s,a)$ is not used in a general MCTS. It is the value supplied by the network. At each step in the game (each move), a series of simulations are performed. In the selection stage of each simulation, moves are selected using the results of former simulations and a bias term based on the probabilities for each action the neural network learned for the state. The move selected in state s is the one that maximizes, over all actions a possible in state s, the sum of $Q(s,a) = \frac{W(s,a)}{N(s,a)}$, the average result of previous simulations, and U(s, a), the bias term. This bias term incorporates the probabilities given by the neural network, and is calculated as:

$$U(s,a) = c * P(s,a)\frac{\sqrt{N(s)}}{1 + N(s,a)}$$

which is a variation of UCT called PUCT [**11**]. The action selected is:

$$a = max_a(Q(s,a) + U(s,a))$$

In the expansion stage, when a leaf node is created, the state s of that node is evaluated by the network. The output of the network's policy head is used to initialize the $P(s, a)$ values for the new node. In a normal MCTS algorithm, the simulation would be followed by a playout stage which uses some default policy to determine the result. In AlphaZero, the result is assumed to be the value of v (the likelihood of the current player winning, as predicted by the neural network). When the MCTS reaches the playout stage, the current state is evaluated by the network and the value v is the result of the simulation. This saves the MCTS time searching a branch of the tree for which no information about the actions taken is recorded. This value is then backpropagated up the tree. The values of Q(s,a), W(s,a), and N(s,a) are updated normally, while the value of P(s,a) is never changed.

After all simulations have been completed, a move is selected using the formula: $max_a(N(s, a))$. In other words, the action which was taken most often in the past is the one selected, since this action is the one found to be most valuable by the network. At this point, the entire remaining tree is discarded. This repeats for each move of the game. Each move made in the game is saved as a triple (s, π, z) containing the state (s), the policy π (the probabilities of choosing each move next), and the result (z), (0 if the player making the move lost and 1 if they won).

5.2. Neural Network Details

The purpose of the neural network is to learn a set of weights which produce the most accurate output across all possible states. To accomplish this, the neural network takes as input a state and outputs two things: a value estimation (v), which is an estimation of the probability of the current player winning the game from the given state, and a probability set (p), which is the probability of each action being picked from the current state. Each probability is a value between 0 and 1, with the sum of all probabilities equal to 1. The weights of the network are first initialized to small random values.

The neural network takes as input a state and outputs a value v (the likelihood of the current player winning the game), and a policy p (a vector representing the probability of choosing each edge). The value v is used by the MCTS as the result of a simulation. The neural network is trained using a random sampling of the (s, π, z) triplets generated by the MCTS such that the difference is minimized between p and pi (the network's predicted probabilities and the tree's observed probabilities) and between v and z (the network's predicted result and the tree's observed result), when evaluating state s. When the network is used by the MCTS, A mini-batch of size 8, which consists of 8 randomly selected states in the same symmetry group as s (the group of 8 states symmetrical to s), is evaluated using the network. The results are averaged to give the P(s, a) and the value for the winning prediction v.

A deep neural network is used consisting of a single convolutional block followed by multiple residual blocks. A convolutional block is a convolutional layer with batch normalization and and a rectifier nonlinearity (a function which takes only the positive part of its argument). A residual block is two convolutional blocks with a skip connection. The final section of the network is split into two heads: the policy and value heads are convolutional blocks followed by a fully connected layer.

The set of examples (inputs and outputs) used by the network show the results of interactions the MCTS has with the game environment; a (s, π, v) triple shows that when state s is evaluated, the policy head should output policy π and the value head should output value v. When the network evaluates state s, the discrepancy between the expected and actual output is used to update the network weights, which informs the next iteration.

5.3. Algorithm

The following values are used in the MCTS for each edge in the partially built game tree: $P(s,a)$, $N(s,a)$, $W(s,a)$, and $Q(s,a)$. $P(s,a)$ is the probability of choosing action a from state s (which is the policy output by the network). $N(s,a)$ is the number of times action a has been chosen from state s. $W(s,a)$ is the sum of all rewards from games in which action a was chosen in state s. Finally, $Q(s,a)$ is the average reward of choosing action a in state s ($W(s,a)$ / $N(s,a)$). $P(s,a)$ and $Q(s,a)$ are both measures of the value of an action; the first comes from the trained network, and the second directly from the previous simulations. $N(s,a)$, $W(s,a)$, and $Q(s,a)$ are all initialized to 0 for new nodes added to the tree.

For game h, $P(s,a)$ is the probability for selecting action a in state s as computed by the neural network based on all prior iterations of the algorithm. This probability is the output of the policy head of the neural network when given state s as input after it trained with data from $h - 1$ games. The probabilities for all the actions possible in state s add up to 1. P(s, a) values are not updated during MCTS. After a node is created, the $P(s,a)$ remains the same for the entire game.

The following two algorithms outline the procedure described in the 'Methods' section of [**11**]. Algorithm 1 gives the interaction between the network and the MCTS, while Algorithm 2 is for making a single move during a self play or tournament game. The initial state of a game is referred to as s_0. In AlphaGo, there are 25,000 games of self player (h) per iteration and the MCTS performs 1,600 simulations (k) per move.

Algorithm 1 AlphaZero Iteration

```
(1) Initialize the deep neural network with random weights. This network is now the
    best player.
(2) Play $h$ games of self play using the current best network for both players.
  (a) Create a new MCTS tree and initialize the $P(s_0, a)$ values of its the root for
      each possible action $a$ which can be taken in the initial state $s_0$ with the values
      obtained from the current best neural network as described above.
  (b) Make move $m$ (for m = 1, 2, .... ) during the game, using Algorithm 2.
  (c) The game ends with result $z$.
  (d) Turn every temporary training example (see Algorithm 2) generated for this game
      into a triple of <state, policy, and winner> by replacing the < placeholder > with
      the result $z$, relative to the player making the move.
  (e) The tree is discarded.
(3) Train the neural network with a random sampling of (s, $\pi$, z) triples. Each triple
    is one training example: $s$ is used as the input to the neural network and $\pi$ and $z$
    as the target outputs for the policy and value heads.
(4) Play a tournament between the old best player and the newly trained player. If
    the new player wins 55% of the games, it replaces the best player.
(5) Repeat steps 2-4.
```

Algorithm 2 Making a move in the game

(1) The state in the root position of the tree is s_{m-1}. Perform a MCTS search with k simulations using Algorithm 3.

(2) After the required number of simulations are completed, the action with the highest $N(s,a)$ value - the one chosen most often during simulation - is chosen as the action a_m for move m.

(3) The action a_m is executed, leading to state s_m which becomes the root of the MCTS tree for the next move. Any parts of the tree above s_m (or who is not a descendant of s_m) are discarded; the subtree below the node with state s_m is kept. The search tree is not saved between MCTS games.

(4) A temporary training example for the neural network is generated for move m. The probability distribution for each possible action a is extracted using:

$$\pi_m(a|s_{m-1}) = \frac{N(s_{m-1},a)^{1/\tau}}{\sqrt{N(s_{m-1},b)^{1/\tau}}}.$$

where τ is a temperature parameter controlling exploration. The triplet $\{s_{m-1}, \pi_m, < \mathit{placeholder} >\}$ is added to the training data for the neural network, to be used after the completion of the current game.

Algorithm 3 MCTS simulation

(1) Selection: Starting from the root of the search tree in each simulation (the current state s_{m-1} in the game), actions are selected by taking the action that maximizes $Q(s,a) + U(s,a)$, where $U(s,a) = c * P(s,a)\frac{\sqrt{N(s)}}{1+N(s,a)}$.

(2) Expansion: A leaf node with state s_L is created. Initialize its $P(s,a)$ values (one for each possible move) using the neural network. The sum of all $P(s_L, a) = 1$.

(3) Playout: This phase is skipped - the value-output v of the neural network is used as an estimate of the probability that the current player wins or loses. This probability is substituted for the observed result of a normal MCTS simulation. The network is given as input a mini-batch of 8 random states from the same symmetry group as s_L (8 states symmetrical to s_L) and the results are averaged.

(4) Backpropagation: Then the value v from 0 to 1 given by the network as the likelihood of the current player winning is assumed as the result of the simulation. W, Q, and N are updated normally. The result of the simulation is added to $W(s,a)$ and $N(s,a)$ is incremented by 1 (where a is the action which was taken during the simulation in state s).

Chapter 6

Dots and Boxes

Dots and boxes is a simple game in which two players take turns drawing an edge (horizontally or vertically) connecting two dots in a grid of n x m boxes. When a player draws the final edge forming a square between four adjacent dots in the grid, that player receives a point and takes another turn. However, the player may choose not to take a box (assuming there is another edge they can take instead). The game ends when there are no more possible edges to play on the board and the player with the most points wins. Dots and boxes is a zero sum game, meaning that what is good for one player is always bad for the other player. Any box taken by one player is a box that cannot be taken by the other. It is also a perfect information game, meaning that each player knows all information about the game. Figure 6.0.1 shows the 12 moves of a $2 * 2$ (squares) game. In this game, the first player (blue) is the winner.

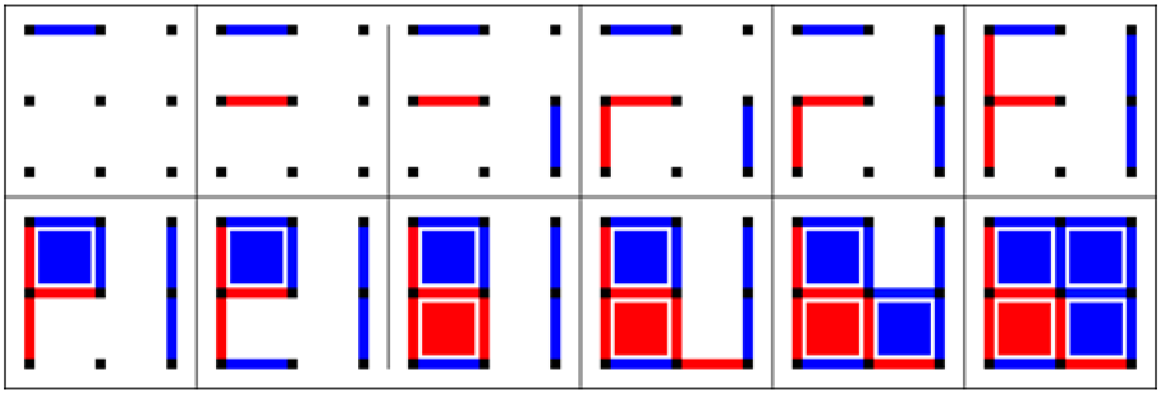


Figure 6.0.1. A $2 * 2$ Dots and Boxes game.

Because dots and boxes is also a perfect information game (a game in which each player knows every possible action and the result), one could - at least theoretically - calculate the best move using a (deterministic) search. In other words, if there was enough time and memory one could simply check every possible sequence of moves to see what the best move is in any situation. This has been done for board sizes smaller than $5 * 5$. However, to do so is computationally expensive because of the size of the search space.

The number of edges on an $n * m$ board is given by $(n+1) * m + (m+1) * n$ or, for a square board, $2 * n * (n+1)$. For a board with n edges, there are 2^n possible states (board configurations) and $n!$ different playable games. For the smallest interesting board of size $2 * 2$, there are 12 edges, 4096 states, and nearly 500 million possible games. The number of states and possible games increases dramatically with larger boards, making it difficult or impossible to definitively calculate the best moves in larger games. Doing so would require exhaustively checking every combination of moves, and even accounting for board symmetries this is an unrealistic goal for any large board size.

The goal then is not to find the optimal move in every possible case, but to design a player that can learn a move which approaches optimality for the states it actually encounters. In other words, the player should be able to 'look' at a given board and make an approximation of which move will be most successful, just as human players do. This is where reinforcement learning can help.

Table 1. Dots and Boxes Data

Board Size	Edges	Boxes	States	Games	Net Score
1	4	1	16	24	1
2	12	4	4096	$4.79 * 10^8$	2
3	24	9	16,777,216	$6.2 * 10^{23}$	-3
4	40	16	1.099 Trillion	$8.15 * 10^{47}$	0
5	60	25	$1.15 * 10^{18}$	$8.32 * 10^{81}$	1
6	84	36	$1.93 * 10^{25}$	$3.31 * 10^{126}$	?

Table 1 shows the number of edges, boxes, states, and games for square boards. It also shows the optimal net score for the starting player (for board sizes which have been solved) [**13**]. This shows the extreme scale of the search space for even small boards. Remember that a standard Go game uses a 19 x 19 board and has $1.74 * 10^{172}$

states. That a computer player is able to play well (much less beat all other players) in a game with so many possibilities shows the power of the AlphaZero approach.

Although dots and boxes is the game used in this work, the focus of the research is on the AlphaZero algorithm, rather than the dots and boxes game itself. For a more detailed description of dots and boxes and its features and strategies, see [**1**]. [**10**] gives a more detailed description of playing dots and boxes with a Monte Carlo tree search. Although the rules are simple, the strategies involved in play are surprisingly complex. For large board sizes, it is often difficult to predict who has the advantage until very near the end of the game, and a player may gain many points during a turn.

Chapter 7

Implementation

The AlphaZero implementation used in this project comes from a Github repository [**4**], under the GNU General Public License. The implementation is written in python 3.7 and uses Keras with a backend of Tensorflow. Keras and Tensorflow are both packages designed for working with neural networks. The core of the implementation was maintained, although certain pieces were adapted to better fit the needs of the project.

The first step in adapting this implementation was to correct a few mistakes in the code which caused runtime errors. These errors were likely caused by variation between the installed package versions and the versions used by the original author. However, since details of the original environment were not given, the code had to be modified to fit out environment. The next step was to analyze the code in detail (and the extensive log files output by the program) and ensure that each piece matched the details described in [**11**]. In doing so, a few errors were discovered, including a duplication of the memory - the (s, π, z) triples - which caused the saved memory to hold half as many previous moves as it should. The parameter used to introduce some randomness into the first move of a simulation was also adjusted to matched what was described in [**11**].

In order to adapt the implementation for dots and boxes, several changes needed to be made. The program was written in such a way that in order to run a new game only the game.py file needed to be changed. The file contains all the implementation details about how the game is represented, what moves are possible, how the state changes, etc. However, some assumptions were made about the game that did not fit dots and boxes, so other files needed to be adapted as well. Originally, the code assumed that every game would maintain a strict A, B, A... ordering of moves between players and that the player to make the last move would always be the winner. This is not the case in dots and boxes, where players can sometimes take multiple turns in a row and the winner is based upon the final score. Modifications were made to account for this and other such assumptions when they were counter to the rules of dots and boxes. Other changes were made to eliminate features which were unnecessary to this work and increased the runtime. The internal logic of the program, however, remains largely the same.

7.1. Representation

A dots and boxes board configuration is represented as a binary number between 0 and 2^n, where n is the number of edges. To determine this number, each edges on the board is numbered (starting at zero) moving from the top left corner, right and down. The numbering is shown in figure 7.1.1. The board is then converted to binary number with each filled edges a one and each empty edge a zero.

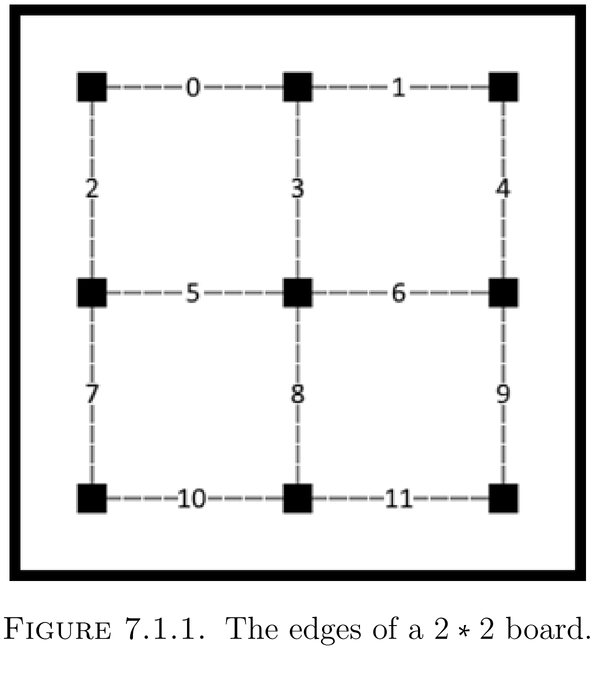


FIGURE 7.1.1. The edges of a $2 * 2$ board.

In the MCTS tree, each state has a unique ID of the form configuration:player:score. The player one is represented with a value of 1 and player two with a value of -1. Strictly speaking, the player is not a necessary part of the state. However, the implementation becomes simpler when the current player is part of the information kept by the nodes. For example, the id '2048:-1:0' is the state with a single edge taken (edge 0), with the current player being player two and that player's net score being zero. When a leaf on the tree is expanded, each child of the leaf is added to the tree. This means that not each expansion will add multiple nodes. If a child has an ID matching a node already in the tree, a new node is not created; the matching node is used instead.

The resulting structure is a directed acyclic graph (DAG) in which some nodes have multiple parents. An example of such a tree is shown in Figure 7.1.2. The shape of the graph in the figure is the one formed in a dots and boxes game. Until the halfway point in the game, each node has more children than it has parents. After the halfway point, each node has more parents than it has children. The result is that the root node has a number of children equal to the number of edges and there is a single terminal node representing each possible net score (one more than the number of boxes). The particular graph shown below is the graph formed by a 1x1 game. There is a single terminal node (because there is only one possible result to the game) and there are 16 nodes (one for each board configuration).

This structure is used because if a state s is reached which is comprised of a board configuration and a net score (relative to the current player), it doesn't matter what sequence of states were reached previously. The optimal move is not affected by previous moves (assuming the net score is the same). So using a structure in which every unique state is represented only once saves the search from duplicating the effort of learning which actions are best in the state.

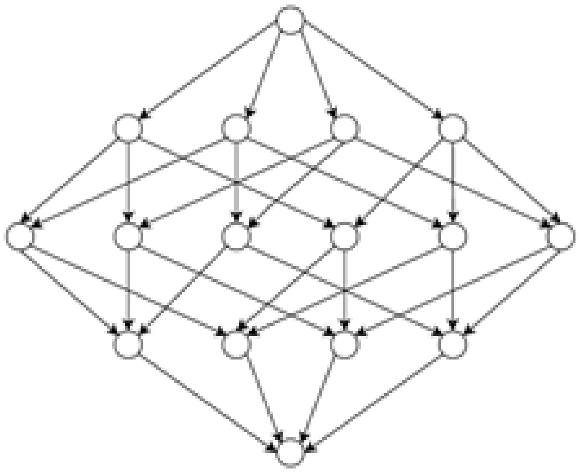


Figure 7.1.2. A directed acyclic graph.

The neural network input of the dots-and-boxes board is presented as a 2-dimensional board representing the dots and the edges. Thus for a board with n boxes per row, the board size is $2n+$. For instance, a game of size 2 would have three input sections of size $5 * 5$. The greater size is used to pad the input and position it relative to the board it represents. This is necessary because a dots and boxes game has rows of length n, $n+1$, n and so on. Rows of length n represent horizontal edges, while rows of length $n+1$ represent vertical edges. However, in order for the convolutional filters to detect patters from regions of the input, that input must be consistently spaced. Because of this padding, some nodes in the input layer represent the dots on the board and provide no information to the network. When a state is evaluated with the neural network, the input to the network consists of a $3 * boardsize$ array. Each subsection of input is a $n * n$ array; this is so all layers of the input have the same dimensions.

In order to help the network identify which nodes represent actual input and which are only used for padding, the first of the three subsections of input gives a value of one in each space representing an edge and a value of zero in each space used as padding. Figure 7.1.3a shows this layer for a $2 * 2$ game.

0	1	0	1	0
1	0	1	0	1
0	1	0	1	0
1	0	1	0	1
0	1	0	1	0

(A) Board input.

-2	-2	-2	-2	-2
-2	-2	-2	-2	-2
-2	-2	-2	-2	-2
-2	-2	-2	-2	-2
-2	-2	-2	-2	-2

(B) Score input.

0	0	0	1	0
1	0	1	0	0
0	0	0	1	0
1	0	0	0	1
0	1	0	1	0

(C) State input.

FIGURE 7.1.3. Network input for state 1847 (edges 0, 1, 2, 3, 6, 7, 9, 10, and 11) with score -2.

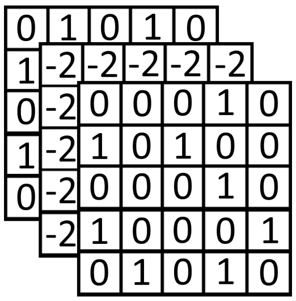


FIGURE 7.1.4. Combined network input for state 1847 with score -2.

The second subsection of input to the network represents the current score (Figure 7.1.3b). In games like Go, the score can be determined from just the board configuration, but in dots and boxes, a single configuration can have several possible scores. This is because who gets the point from a box is based on who took the last edge. So just knowing which boxes are taken is not enough to determine how many points each player has. The net score (relative to the player making the next move) is given as input to the network so that the value head is able to output a correct value for the current player.

The final subsection of input represents the current board configuration. Edges which are filled are represented with ones and edges that are not filled are represented with zeros (as are the spaces in the board which provide padding). This input is shown in Figure 7.1.3c. These three inputs are combined into a single 3D input layer, as shown in Figure 7.1.4.

7.2. Testing

Tests were conducted for boards of size 2, 3, and 4. For all tests, a number of MCTS simulations proportional to the number used by [**11**] was used (50 for a $2 * 2$ board). For a $2 * 2$ board, a memory size of 1000 was used (that is, 1000 moves - enough for roughly 8 iterations of 10 self-play games) and a proportional number was used for other board sizes. These values are shown in Table 1. Each test used a convolutional neural network with six residual blocks, with a batch size of 256 and a learning rate of 0.1. 30 games were player between different versions in the tournament. In half the games, the best player stared, and in the other half the newly trained player started. The best player was replaced if the new player won greater than 1.3 times times the games won by the best player. For the purposes of the tournament, a draw was considered half a point for each player.

When testing the number of filters, we start with a number of filters proportional to the number used by [**11**] with respect to the input size. We then test filters sizes of .25, .5, .75, 1, 1.25, 1.5, 1.75, and 2 times that standard number. The number of filters in each test is shown in Table 2. For each board size, the column labeled 1 shows the proportional number of filters used as the baseline for that board size.

Table 1. Proportional Parameters

Size	Memory	Simulations
2	1000	50
3	2000	100
4	3333	167

In order to test the effect of different numbers of training loops, we use the standard number of filters for that board size. The numbers of loops tested are 5, 10, 15, 120, and 25.

Table 2. Filter Tests

Size	.25	.5	.75	1	1.25	1.5	1.75	2
2	9	18	26	35	44	53	61	70
3	18	35	53	70	88	105	123	140
4	29	57	86	114	143	171	200	228

Since the goal of this research is not to produce the best possible player, but rather to explore the effect of the number of filters and training loops on the learning approach, we reduced the length of the individual training runs to be able to check out a wider variety of filter sizes and training loops.

In each iteration of a test, we collect information about the three stages of the iteration (self play, retraining, and tournament). For each, the time taken during that stage was collected. During the retraining stage, the policy head error, value head error, and total error was collected for each training loop. During the tournament stage, the final score of each game was recorded, as well as a tally of the total number of wins for each player and wins for the starting and non-starting players.

Chapter 8

Results

8.1. Win Rate

In order to ensure that the implementation actually learns to player the game, the tournament data was examined. Figure 8.1.1 shows the number of wins (out of 30 games) by the starting player. For this graph, draws are counted as a half point. The starting player wins in optimal play in a $2 * 2$ game. The current and best player each played as the starting player in half of the games. Overall, the average number of wins increases. There are several deep trenches which likely reflect a point when the new player learned a strategy which allowed it to beat the best player even as the non-starting player. There is still some fluctuation even in the final iterations.

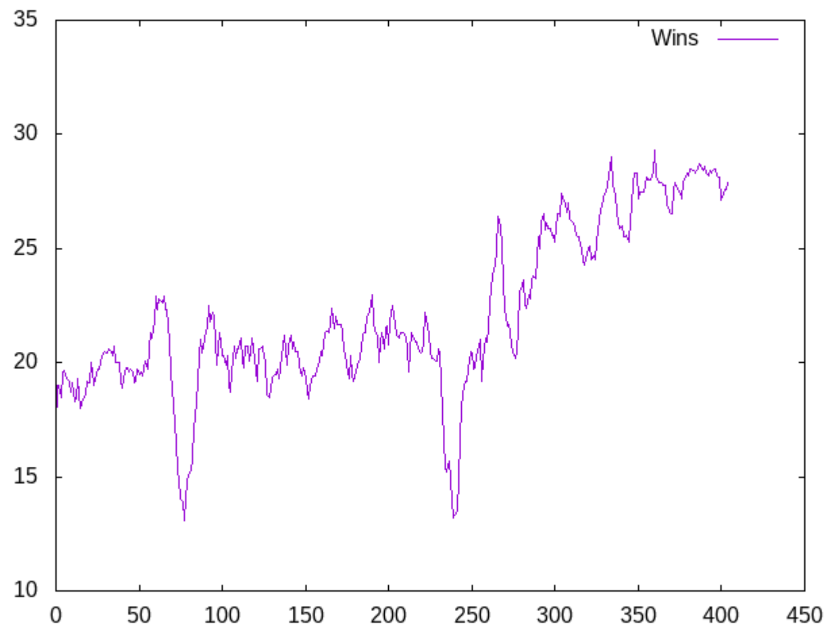


Figure 8.1.1. Number of wins for the starting player in a 30 game tournament.

In order to determine what role the newly trained player had on this fluctuation, Figure 8.1.2 shows the number of wins for the best player when it was the starting player. There are 15 such games in each tournament. In the final iterations, the average approaches 15. This shows that at least some of the fluctuation in Figure 8.1.1 is the result of a newly trained player being worse than the best player, and thus losing games which it should be able to win.

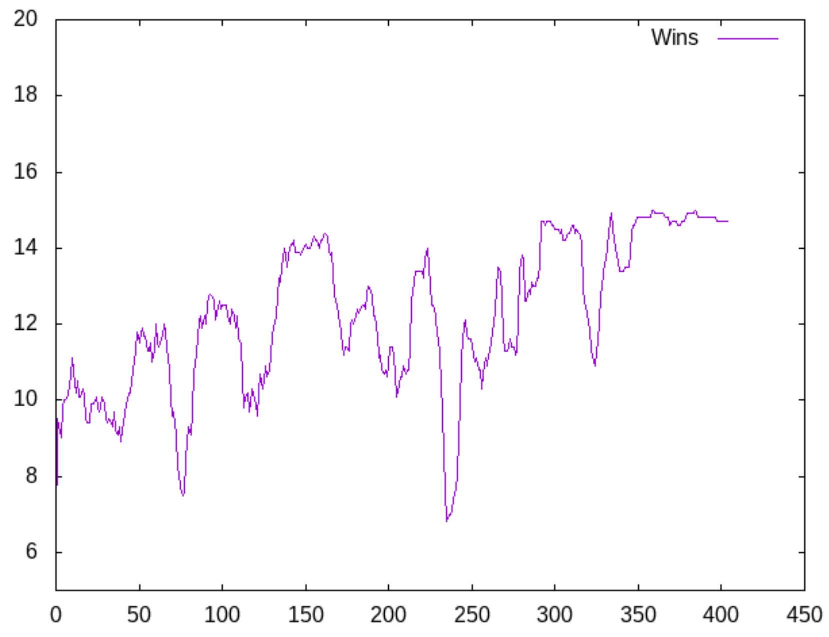


FIGURE 8.1.2. Number of wins for the best player when starting in a 30 game tournament (max 15).

8.2. Timing

The first thing to consider when evaluating performance is the computation time. Since memory is not a limiting factor for AlphaZero, the time taken to reach a certain level of play determines how effective the algorithm is. If doubling the number of filters doubles the time per iteration but only slightly decreases the number of iterations needed to learn, then it is not beneficial.

Figure 8.2.1 shows the average time in seconds needed to play a single game using board sizes of 2, 3 and 4. As the board size increases, the time per game increases at a much higher rate. Between a 2 ∗ 2 and 3 ∗ 3 game, the input size doubles, but the time per game increases by a factor of 8. The difference between the time to play 3 ∗ 3 and 4 ∗ 4 games is again much higher than the difference in input size. A 4 ∗ 4 game has roughly 1.6 times the edges of a 3 ∗ 3 game, but the time increases by a factor of 5.

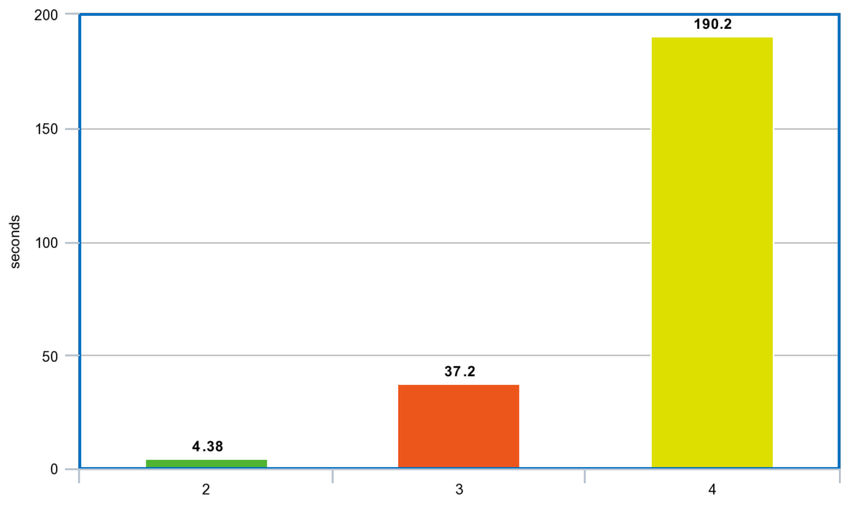


FIGURE 8.2.1. Time in seconds for a game with various board sizes.

Figure 8.2.2 shows the time to play a single game for each number of filters used by a 3 ∗ 3 game. The x-axis is labeled relative to the standard filter amount. So for a 3 ∗ 3 game, the bar labeled 1 represents 70 filters. The change in game time based on number of filters is roughly linear. This is to be expected, since the number of filters only affects the computation time when evaluating a state. During a game, this is only done to initialize each a node.

Figure 8.2.3 shows the time needed to perform 10 training loops in networks of size 2, 3, and 4. The training time slightly more than doubles between the 2 ∗ 2 and 3 ∗ 3 games. Likewise, it more than doubles between a 3 ∗ 3 and 4 ∗ 4 (even though the input size less than doubles). This makes sense because as the board size increases the number of subsections to which the filter must be applied increases.

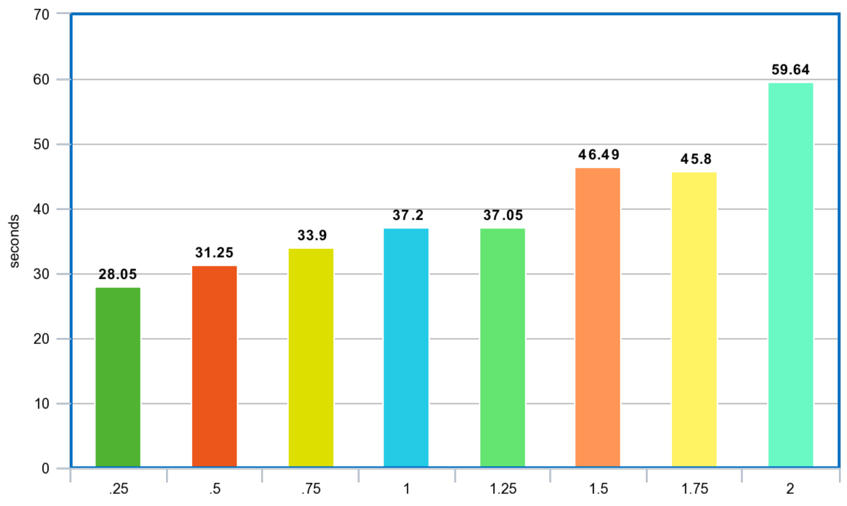


FIGURE 8.2.2. Time in seconds for a $3 * 3$ game with various filters.

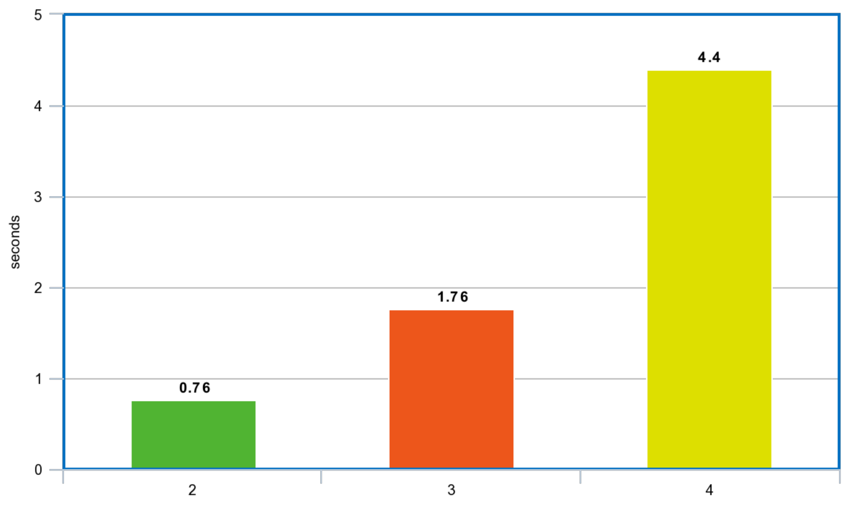


FIGURE 8.2.3. Time in seconds to perform 10 training loops in games with various board sizes.

Figure 8.2.4 shows the time to train 10 loops for a $3 * 3$ board with various numbers of filters. As the number of filters increases, the time to train increases nearly linearly. This is to be expected, since majority of the weights (and thus the calculations to be performed) in the network come from the filters.

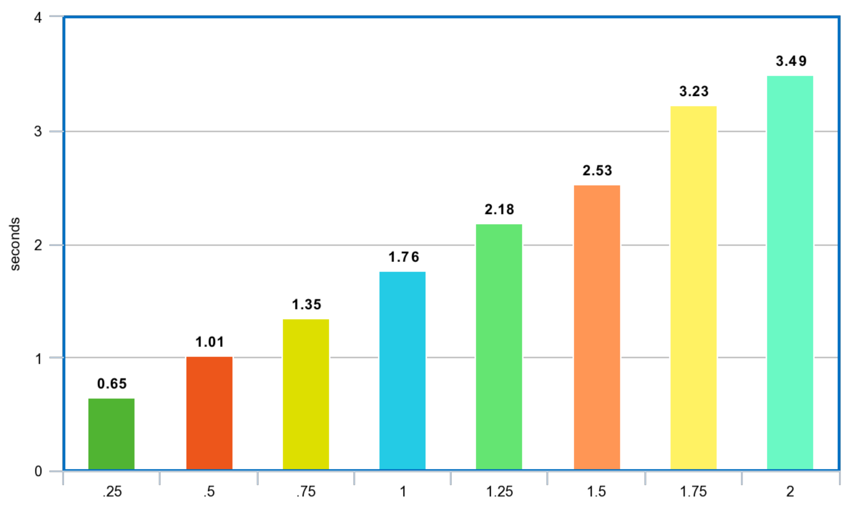


FIGURE 8.2.4. Time in seconds to perform 10 training loops in a $3 * 3$ game.

8.3. Convolutional Filters

The error measured by the network during training is one measure of the performance of the network. As the network becomes better, the error decreases. In the following figures, the error of the tests are shown. Figure 8.3.1 shows the error of a player using a network with 61 filters in a $2 * 2$ game. Three values are shown: the policy error is the error of the policy head of the network, the value error is the error of the value head of the network, and the total error is a combination of the two. It is expected that the value loss is much smaller than the policy loss, since the policy head outputs a collection of values, while the value head outputs only one. In Figure 8.3.2, the same data is shown using a rolling average with a window size of 50, in order to make the data more readable. For the remainder of the figures, the rolling average is used (unless otherwise noted). For $2 * 2$ games the window size is 50, while the window size is 10 for $3 * 3$ games.

Figure 8.3.3 shows the total error one a $2*2$ game with various numbers of filters. Overall, it seems that the 61 filter player (with the second most filters) learned the

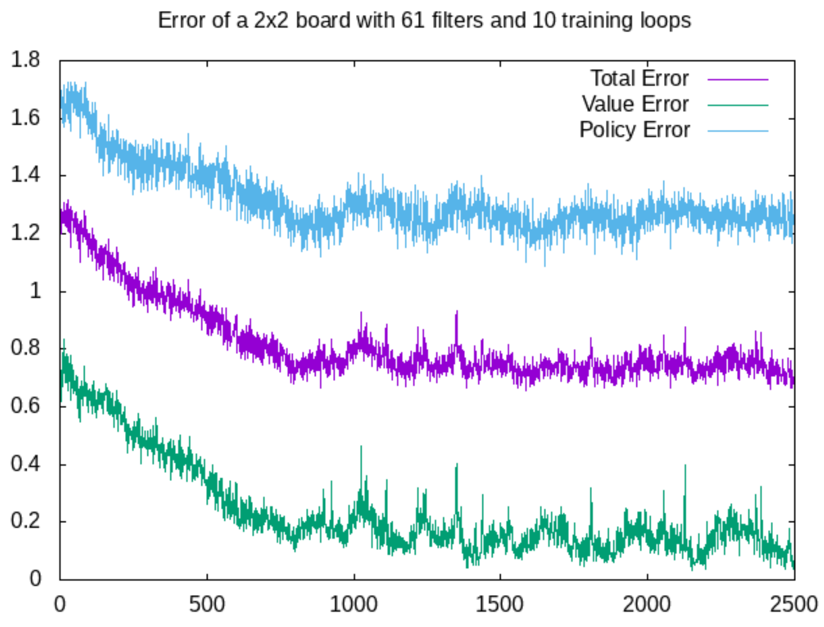


FIGURE 8.3.1. Error for a $2 * 2$ player with 61 filters.

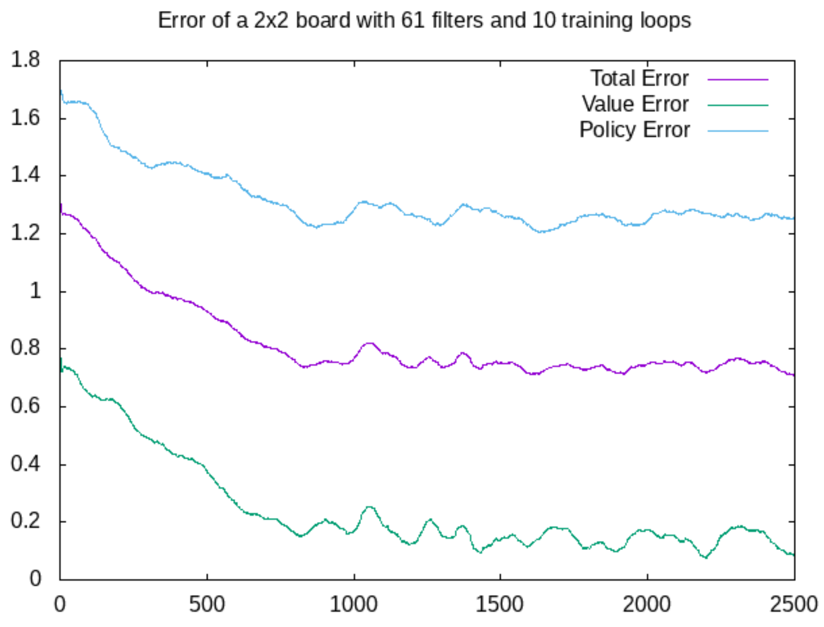


FIGURE 8.3.2. Error for a $2 * 2$ player with 61 filters using a rolling average.

best. However, in the early iterations is was nearly the worst. In fact, The player with the least filters (9) performed best initially, while players with more filters performed worse. At the end, however, players with more filters generally seemed to surpass those with fewer filters.

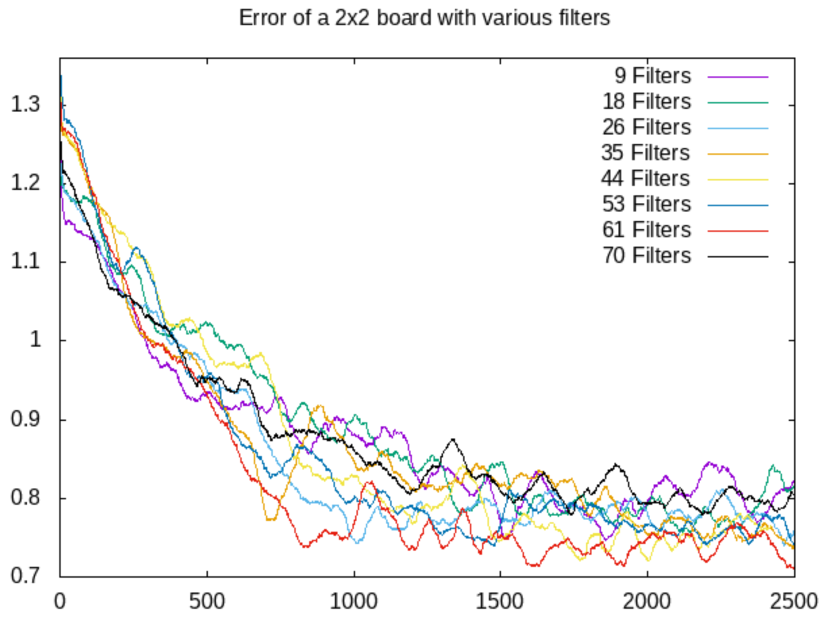


FIGURE 8.3.3. Error for all tested filter numbers on a $2 * 2$ game.

In Figure 8.3.4, the error is shown for only the most, least, and default number of filters. In the long run, the 35 filter player clearly performs better than the 9 filter player. However, the 70 filter player is no better than the 9 filter player.

The error for different numbers of filters in a $3 * 3$ game is show in Figure 8.3.5. However, none of the test produced enough data to show definitively what their end behavior would be. The same behavior is seen here as in Figure 8.3.3. Players with more filters perform worse early one, and seem to overtake the other players in later iterations.

Figure 8.3.6 shows the same data for only the most, least, and default filters. The 18 filter player performs best in the beginning, but the 140 filter player appears to

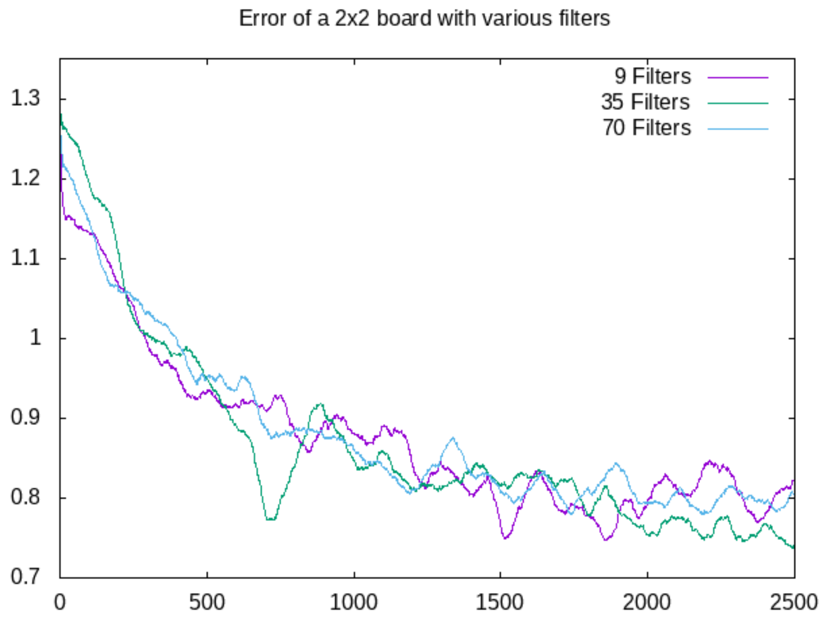


Figure 8.3.4. Error for 9, 35, and 70 filters on a $2 * 2$ game.

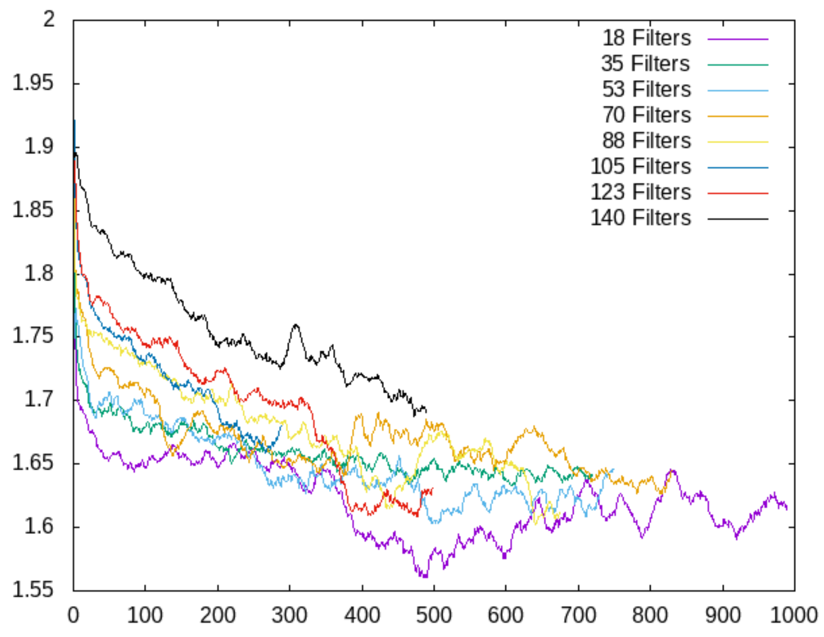


Figure 8.3.5. Error for various filters on a $3 * 3$ game.

be trending towards surpassing it. However, it is not certain that this would happen if more data was collected.

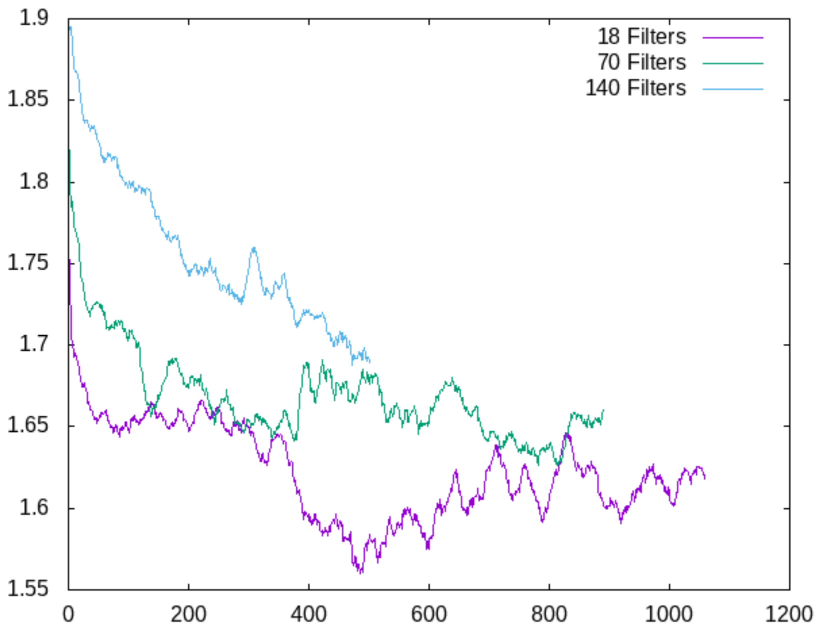


FIGURE 8.3.6. Error for 18, 70, and 140 filters on a $3 * 3$ game.

Due to the massive increase in time needed for each iteration as the board size increases, the data for the $4 * 4$ game is very sparse. Figure 8.3.7 shows the error of a $4 * 4$ player with 228 filters. The total loss trends slightly downward, but there is not enough data to draw any conclusions. The other $4*4$ tests are similarly uninformative. For this reason, no other results for the $4 * 4$ game are shown.

8.4. Training Loops

In testing different numbers of training loops, the default number of filters was used (35 for a $2 * 2$ board). Figure 8.4.1 shows the error for a $2 * 2$ player with 5, 10, 15, 20, and 25 training loops per iteration. With training loops, a fewer loops, the longer it takes to collect data (the opposite of what happens with filters), since between each set of loops the self play and tournament games are played. Again, the players with more training loops had more error in the early iterations but less near the end. Figure 8.4.2 shows the error of only 10 and 25 training loops. At around

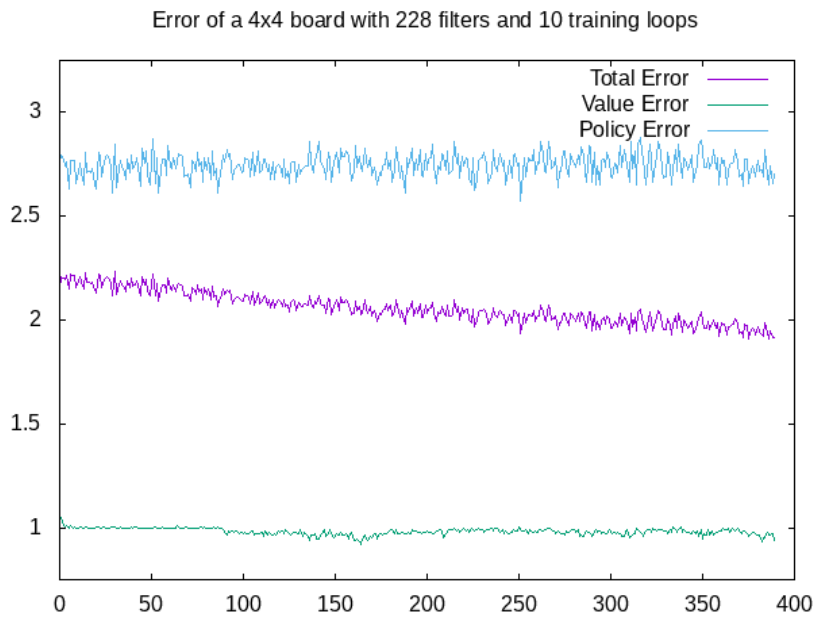


FIGURE 8.3.7. Error of a $4 * 4$ player with 228 filters.

700 loops, the 10 loop player suffered a major setback, which allowed the 25 loop player to surpass it. The data for the $3 * 3$ and $4 * 4$ player is too limited to make any valuable comparisons.

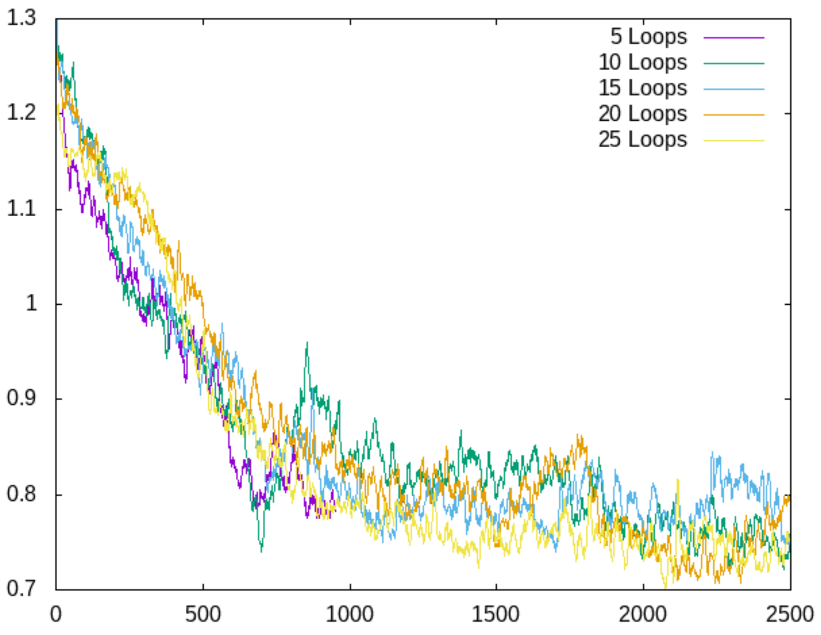


FIGURE 8.4.1. Error of a $2 * 2$ player with various numbers of training loops.

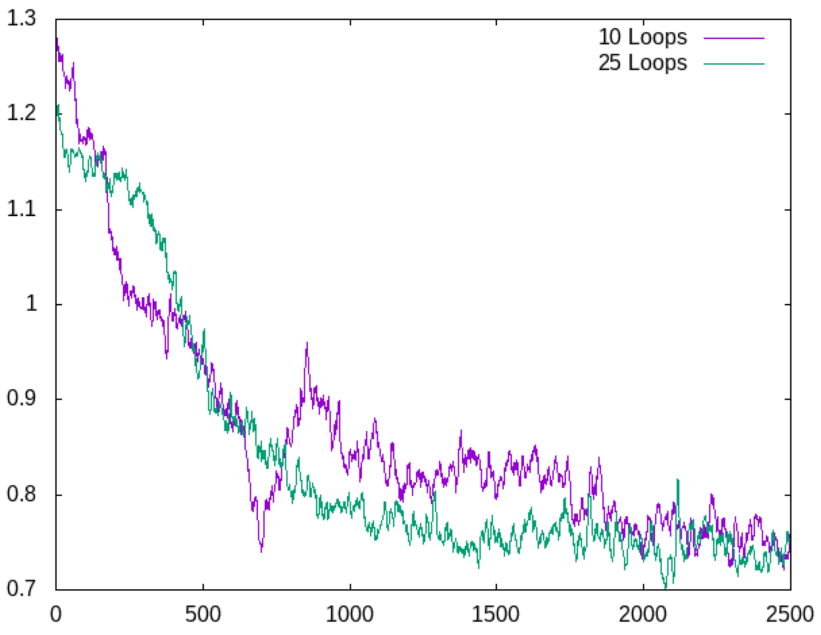


FIGURE 8.4.2. Error of a $2 * 2$ player with 10 and 25 training loops.

Chapter 9

Conclusions

Although the goal of this thesis was to determine how the input size is related to the optimal number of convolutional filters and training loops, the data from games of size $3 * 3$ and $4 * 4$ is too limited to draw any hard conclusions. However, $2 * 2$ and $3 * 3$ data does show a few trends. It appears that generally as the number of filters increases the player performs worse in the early iterations and better in the late iterations. This is likely because in the early iterations learning is not as effective for a large number of filters (due to the larger number of weights to be adjusted). In later iterations, the larger number of filters allows the network to capture more information about the board and discover more features. As with the filters, the more training loops the network has the worse it performs early on and the better it performs in later iterations.

Overall, the increase in performance for larger numbers of filters seems to be beneficial (at least in the long run), since adding more filters does not significantly affect the total time per iteration. Likewise, adding more training loops is generally beneficial, since the training stage takes the shortest time of all stages in each iteration.

Chapter 10

Future Work

In future work, it would be beneficial to expand the tests performed in this work to include larger board sizes and a wider range of filter numbers and training loops. With more data encompassing more complex games, the relationship between these parameters and game complexity would become more clear.

Although only two parameters were examined in this research, there are many more parameters involved in the AlphaZero algorithm than the number of filters and training loops. Each parameter can be tuned, either to improve the learning rate of the algorithm or to reduce time or memory requirements. The memory size, number of games of self play, number of tournament games, number of convolutional layers, batch size, and many other factors affect the speed and effectiveness of the algorithm. For instance, increasing the number of MCTS simulations per move increases the accuracy of the policy the search produces during self play. This gives the neural network better examples to learn from. However, this also increases the time taken during self play and the memory used. There are similar benefits and problems with each parameter. In future work, it would be beneficial to examine these parameters.

In future work it would also be interesting to investigate different neural network structures. In this work, a dots and boxes state is represented using three layers as described in section 7. One layer shows which edges in the board are filled, one shows which nodes in the input represent board edges, and one represents the net score for the current player. This net score is necessary for the network to determine the value of a state. Otherwise, a state in which the player to move has ten out of ten taken boxes would be valued the same as one in which the player to move has zero of ten taken boxes. However, in order to determine the optimal move (the one which results in the highest net gain for the player), the score is not needed. Therefor, it might be beneficial to use a network with two input layers. The first layer would represent which edges are filled and which nodes in the layer are edges. This would be the main input layer at the head of the network. Another input layer would give only the net score and would be combined with the input to the value head. In such a structure, the score would only be input into the network at the point in which it is needed. Because the input to the network body would be smaller, it would likely require fewer convolutional layers or filters in order to achieve an equivalent learning rate.

BIBLIOGRAPHY

[1] Elwyn Berlekamp. *The Dots and Boxes Game: Sophisticated Child's Play*. Routledge, 2000. ISBN: 9781568811291.

[2] Johan Bjorck, Carla P. Gomes, and Bart Selman. Understanding batch normalization. *Computing Research Repository*, abs/1806.02375, 2018. url: https://arxiv.org/abs/1806.02375, Date of Last Access: Dec. 5, 2018.

[3] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4:1:1–43, 03 2012.

[4] David Foster. Deepreinforcementlearning. https://github.com/AppliedDataSciencePartners/DeepReinforcementLearning, 2018. Date of Last Access: Nov. 18, 2018.

[5] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michèle Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. The grand challenge of computer go: Monte carlo tree search and extensions. *Commun. ACM*, 55(3):106–113, March 2012.

[6] Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856 – 1875, 2011. url: http://www.sciencedirect.com/science/article/pii/S000437021100052X.

[7] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016.

[8] David P. Helmbold and Aleatha Parker-Wood. All-moves-as-first heuristics in monte-carlo go. In *International Conference on Artificial Intelligence*, pages 605–610, 2009.

[9] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *Computing Research Repository*, abs/1502.03167, 2015. url: http://arxiv.org/abs/1502.03167, Date of Last Access: Dec. 5, 2018.

[10] Jared Prince. Game specific approaches to monte carlo tree search for dots and boxes. *Honors College Capstone Experience/Thesis Projects*, (701), 2017. url: https://digitalcommons.wku.edu/stu_hon_theses/701.

[11] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy

Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354, 2017.

[12] Richard S. Sutton and Andrew Barto. *Reinforcement Learning: An Introduction.* The MIT Press, 2017. url: http://incompleteideas.net/book/bookdraft2017nov5.pdf, Date of Access: Nov. 2, 2018.

[13] David Wilson. Dots-and-boxes analysis results. http://wilson.engr.wisc.edu/boxes/results.shtml. Date of Access: Nov. 12, 2018.