Reykjavík University Project Report, Thesis, and Dissertation Template

by

Sigurður Helgason

Thesis of 60 ECTS credits submitted to the School of Computer Science
at Reykjavík University in partial fulfillment
of the requirements for the degree of
Master of Science (M.Sc.) in Computer Science

December 2019

Examining Committee:

Yngvi Björnsson, Supervisor
Professor, Reykjavík University, Iceland

Tough E. Questions, Examiner
Associate Professor, Massachusetts Institute of Technology, USA

Reykjavík University Project Report, Thesis, and Dissertation Template

Sigurður Helgason

December 2019

Abstract

this is my abstract written in the language of English I am testing thought

alas I knew

ok so now I'm trying something new

asdf

test

Herkænsku mat á djúptauganetum

Sigurður Helgason

desember 2019

Útdráttur

this is my abstract written in the language of íslensku þæöasdf

Important!!! Read the Instructions!!!

If you have not already done so, LaTeX the `instructions.tex` to learn how to setup your document and use some of the features. You can see a (somewhat recent) rendered PDF of the instructions included in this folder at `instructions-publish.pdf`. There is also more information on working with LaTeX at `http://samvinna.ru.is/project/htgaru/how-to-get-around-projects-publish.pdf`. This includes common problems and fixes.

This page will disappear in anything other than draft mode.

I dedicate this thesis to my family who while not understanding most of what I do always support me. The friends that still want to talk to me even though my schedule has rendered me unmeetupwithable. Lastly but certainly not least,
Don't Panic.

Acknowledgements

So long, and thanks for all the fish.

Acknowledgements are optional; comment this chapter out if they are absent Note that it is important to acknowledge any funding that helped in the workThis work was funded by 2020 RANNIS grant "Survey of man-eating Minke whales" 1415550. Additional equipment was generously donated by the Icelandic Tourism Board.

Preface

This dissertation is original work by the author, Sigurður Helgason.

The preface is an optional element explaining a little who performed what work. See `https://www.grad.ubc.ca/sites/default/files/materials/thesis_sample_prefaces.pdf` for suggestions.

List of publications as part of the preface is optional unless elements of the work have already been published. It should be a comprehensive list of all publications in which material in the thesis has appeared, preferably with references to sections as appropriate. This is also a good place to state contribution of student and contribution of others to the work represented in the thesis.

Contents

List of Figures

List of Tables

List of Abbreviations

MSc	Masters of Science
ML	Machine Learning
AI	Artificial Intelligence
ANN	Artificial Neural Network
DNN	Deep Neural Network
MCTS	Monte-Carlo Tree Search
CAV	Concept Activation Vector
DL	Deep Learning
MI	Model Interpretability
ME	Model Explainability

This part of the dissertation introduces the concepts and the field.

Chapter 1

Introduction

State the objectives of the exercise. Ask yourself: Why did I design/create the item? What did I aim to achieve? What is the problem I am trying to solve? How is my solution interesting or novel?

The world of deep learning (DL) is exciting, we have models that can examine images and very reliably be able to identify it's content. Deep learning models have beaten Ophthalmologists in identifying diabetic retinopathy, they've identified cancer cells where others have not. Deep learning models have even predicted the likelihood a person will die in the following year with good accuracy (CITE). These models have done these things extremely accurately, cheap, and fast.

DL in conjunction with a randomized State-space Search Algorithm Monte-Carlo Tree Search was used to defeat the standing world champion Lee Sedol (CITE ALPHAGO ZERO) in the incredibly expansive game of Go. This was done in the year 2017, the researchers used reinforcement learning (RL), where the agent learned by only playing against itself. These results imply that given enough time to learn the computer agent can gain a deep insight into how the game should be played.

The field of examining deep learning models to gain a richer understanding of how they work, and what makes them so much better than humans at various tasks is still new. This is the field of Explainable Artificial Intelligence (XAI). If we wish to continue using artificial intelligence (AI) and machine learning (ML), to improve our lives we must examine how they work, this is not only to improve the models themselves but

could also augment our ability in many cases. Furthermore, these explanations are a legal requirement now in many areas, namely, legal, and medical. Recently, XAI has generally been focused on Model Interpretability (MI), in particular, interpreting the model in such a way that we might predict what the model will do given a particular input. Another class of XAI would be Model Explainability (ME), where we can adequately explain how the internal mechanics of the model works. A simple example of when we're able to apply ME would be when we print out a decision tree that is generated from an ML algorithm, with the decision tree nodes and is conditions we can see exactly what the tree will do with a given input. While in MI we would have to predict the results, without going into details as to why.

In this thesis, we take a look at how we can examine an artificial neural network (ANN) to understand which higher-level concepts (HLC) it deems important for a given state within games. These games can be simple like Tic-Tac-Toe or Breakthrough and extremely complicated like Chess or Go.

The method of examining HLC's within games has generally been done by examining the current state of the game by evaluating them concerning some heuristic. A heuristic within games are evaluations of the end reward for a state given only the state, for example, the number of pieces left within a game of chess. Intuitively, the amount of pieces left is a good estimate for a state in chess, this is a higher-level concept we use to evaluate a chess position. The piece amount can be considered as a lower-level concept than other concepts we use. For instance, grandmaster chess players evaluate a position w.r.t. states where the king is safe from attack or the structure of the pawn positions. This paper examines the evaluation of a neural network of a state regarding those higher-level concepts if human players evaluate the king's safety of a state low but the neural network highly values it, and the neural network plays better than the player. There could be an avenue for us to improve our game by considering king safety more thoroughly.

We define the research questions

1. Given a NN that plays Breakthrough very well, can we evaluate if that NN recognizes the HLCs that human players use.

2. Does a NN that trains itself using self-play learn these HLCs over time, and does it start to emphasize the HLCs that are generally taught later to human players, more as it trains itself more.

3. Does a NN that plays better than some humans focus on HLCs that are not generally considered by those human players and it.

1.0.1 Summary of the thesis

In this thesis, we take an example game of Breakthrough, train a neural network to play the game using only self-play. That is, the neural network is trained only by playing against itself with no outside input other than the rules of the game itself. And we examine the neural networks against popular Higher-level concepts (heuristics) that we use to play the game.

Chapter 2

Background

In this chapter, we discuss traditional artificial intelligence in the context of search the algorithms we used, what some other similar methods exist as well as the field in general. We allocate a large portion of the section to Breakthrough as that will be the testbed for future sections

2.1 Environments

When researching Artificial Intelligence it is important to select an environment that is a suitable abstraction for the task at hand. Environments vary significantly and can be identified by their characteristic. The characteristics that are generally used to describe environments can be seen in Table 2.1. [1]

characteristic	**Values**	**Description**
Observable	Fully, Partially	How much of the environment can your agent percieve.
Agents	Single, Multi	Are there multiple agents playing in the environment.
Deterministic	Deterministic, Stochastic	Do the actions your agent do deterministicly impact the environment
Episodic	Sequential, Episodic	Are actions episodic or sequential
Static	Static, Semi-Static, Dynamic	ehh
Discrete	Discrete, Continuous	Is your environment discrete w.r.t actions (does it end)

Table 2.1: Characteristics of environments

Categorizing environments like this gives you the power to find an environment in which a method works and know it can be applied to different environments with the same characteristics. Talk about agents in the context of environments as the entities that act within the environment. A game player is an agent as well as the neural network model that exists within a car that can drive in the real world.

2.2 Game Environment

With classical Artificial Intelligence Game Environments are commonly used to validate a method, game environments can be games like Tic-Tac-Toe, Breakthrough, and driving simulators. Game environments are a suitable place to apply AI as they serve an important function as an abstraction of the real world, for instance, a self-driving car agent who is verified to avoid driving into walls in a simulation is possibly safer than one who is not.

2.3 Breakthrough

The game breakthrough is a simplified version of chess, the game is set up on a MxN board with cells like in chess, and each player starts with two rows of pawns at either end. The objective of the game is to move one of the pawn pieces to the opposite end of the board. A player wins if either they have reached the opposite end of the board or have captured all of his opponents' pawns. The pawns differ from chess pawns in such a way that they can not move two squares on the first move and, they can move diagonally as well as forward, this leads to the game being impossible to draw as pieces are always able to move. An example of an initial board in breakthrough can be seen in Figure 2.1

Categorizing Breakthrough with the characteristics described in Section 2.1. We end up with the description shown in Table 2.2. These characteristics are identical to that of Tic-Tac-Toe, and Chess. This categorization is the most common in board games where two player compete.

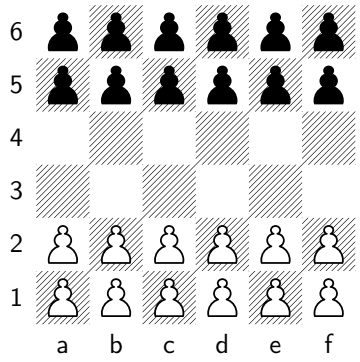

Figure 2.1: Example breakthrough board

Characteristic	Value
Observable	Fully
Agents	Multi
Deterministic	Deterministic
Episodic	Sequential
Static	Static
Discrete	Discrete

Table 2.2: Categorization of Breakthrough

Categorizing Breakthrough with the characteristics described in Section 2.1. We end up with the description shown in Table 2.2. These characteristics are identical to that of Tic-Tac-Toe, and Chess. This categorization is the most common in board games where two-player compete.

2.3.1 Heuristics of Breakthrough

To evaluate the game of Breakthrough we can consider many heuristics (higher-level concepts) for instance a very simple heuristic would be the number of pieces each player has left. This heuristics gives us some insight into how well the game is progressing, but obviously, there are cases where this doesn't tell us much, as in cases when your opponent has a single piece left that is on the row immediately before the row needed for him to win. No matter how many pieces you have left, this state is bad for you if you're not able to capture that piece. An example of such a state can be seen in Figure 2.2.

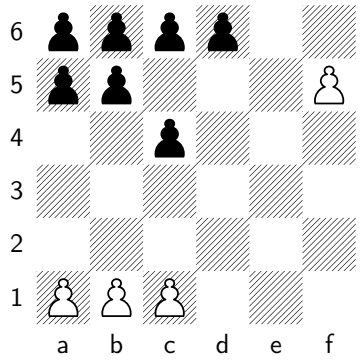

Figure 2.2: Breakthrough board where Heuristic 1 doesn't work well

A different heuristic would be the distance of your most advanced pawn minus your opponents most advanced pawn, this heuristic could give you insight into how close you are to winning the game or how close your opponent is. As a higher-level concept, we can call this concept your aggressiveness, as it closely resembles how aggressive you are going for in for the win. Generally, in Breakthrough, it is favorable to move your whole team as a unit and play more defensively, so this heuristic is probably not one that will lead to the most wins.

2.4 State-Space Search

Traditionally methods for playing games were searching through the environment using a heuristic to guide the search. A simple way of doing a heuristic-based search would be to give all states a 0 as a score and give states where you win a positive score. We say that that search algorithm is not guided, and the algorithm will probably have to evaluate every state in the state-space. This method of searching is generally extremely inefficient as the state-spaces of game environments are often extremely large, and sometimes infinite. For instance, an estimate of the state-space for breakthrough is $M * N * 2^{N*4}$ where M is the height of the board N is the width of the board. The 2^{N*4} represents each piece either being alive or captured. So for a small board, $5x4$ the upper-bound estimation of the state-space is 1310720 states.

This is why a good heuristic is very valuable because we can disregard all states S' that result from doing a move a in state S as they will only lead to worse outcomes.

The algorithms that are used in traditional state-space searches are for instance Depth-first search (DFS), Breadth-first Search (BFS), Alpha-Beta Pruning Search (AB-Search), and Monte-Carlo Tree Search (MCTS).

More modernly, these search methods have been amplified by Machine Learning, in such a way that we do not need to figure out a good heuristic for a given state, but rather, we apply a machine learning model to learn a function that takes in a state and returns an evaluation of that state. This can lead to a significant time reduction as we do not need to simulate a whole game from a state to receive its evaluation we rather receive the evaluation from the model.

2.4.1 Monte-Carlo Tree Search

In the algorithm Monte-Carlo Tree Search, there is an agent within some environment. Where each node in the environment represents a state-action pair of the environment, by state-action pair what is meant it is some state and the action that brought the agent to that state. This pair should be unique within the environment. We will discuss these concepts within game-environments and therefore we might say player instead of agent and game instead of the environment.

MCTS is a method of exploring an environment in a randomized manner (Monte Carlo is the term implying randomness). In MCTS there are four stages. Selection, Expansion, Simulation, and Back-propagation. They happen sequentially and repeatedly. MCTS is initialized with a tree consisting of the unexpanded initial state of the environment.

MCTS is a method of exploring an environment in a randomized manner (Monte Carlo is the term implying randomness). In MCTS there are four stages. Selection, Expansion, Simulation, and Back-propagation. They happen sequentially and repeatedly. MCTS is initialized with a tree consisting of the unexpanded initial state of the environment.

In MCTS there is a tree representing the game-environment. This tree consists of nodes n_i where i represents the point in time of the node, for example, N_0 in chess is the initial position and N_x is some position in the middle of the game and N_e is one of the states representing a position where there is either a draw or one player has won the match. Each of the nodes has 4 values, s, a, Q, and N. These values represent these items, s is the state of the environment, a is the action that brought the previous node $n_(i-1)$ to node n_i, Q is the average value of the node (value meaning the outcome of the game), and N the number of times the MCTS algorithm has visited this node. The values s and a uniquely identify a position in the environment and are often called state-action pairs.

The MCTS algorithms four phases

1. Selection

2. Expansion

3. Simulation/Rollout

4. Backpropagation

$$\text{Child UCT value} = \frac{Q_{(s',a')}}{N_{(s',a')}} + c_{uct} * \frac{\sqrt{log(N_{(s,a)})}}{N_{(s',a')}} \tag{2.1}$$

Selection

During the selection phase, a node (s, a) within the tree which has not yet been expanded is found. This process uses Upper Confidence Bound on Trees (UCT) to find that node (s, a), the formula is described in Equation 2.1. For a parent node (s, a) (initially the root of the tree) we select the child with the highest UCT value. Repeatedly until an unexpanded node is found. This process is done to balance the amount of exploration vs exploitation of nodes in the tree.

Expansion

Then the expansion phase expands the node generating all of (s, a)'s children (s', a') are generated and connected to (s, a). This is done by applying all actions a' to (s, a). These children are the product of applying each action a' to s in the parent node.

Rollout

Next during rollout actions from (s, a) are randomly selected to move to (s', a'), then repeated to go to (s'', a''), until a terminal node within the environment is reached. By terminal, we mean a state in which the game is finished. A terminal node in MCTS can generally return any value, but in the context of this paper, we only return (+1 white wins, -1 black wins, 0 draw).

Backpropagation

The result from the terminal node is then propagated up through the path taken by selection (s, a) up to the root of the tree, updating the $Q_{(s,a)}$ values of each node (s, a).

When training a neural network the UCT formula is modified slightly to prefer selecting nodes that the neural network values highly by introducing a second scalar to the formula $f(s, a) = (p, v)$, the resulting formula is described in Equation 2.2, and is called PUCT. Secondly, the backpropagation process is modified to instead of doing rollout/simulation to receive a reward the predicted value v from the neural network is used instead.

$$\text{Child PUCT value} = \frac{Q_{(s',a')}}{N_{(s',a')}} + c_{uct} * P((s, a)) * \frac{\sqrt{log(N_{(s,a)})}}{N_{(s',a')}} \tag{2.2}$$

2.5 Machine Learning

Machine Learning (ML) is a research field in which machines apply statistical functions on data to achieve a correct output, by *correct* we mean the corresponding result which we expect. Generally, this a repetitive process where we look at examples of the data,

and the algorithm progressively gets closer to the underlying function of the data it is fed. This process is therefore similar to trial and error for humans. ML is a sub-field of Artificial Intelligence. ML algorithms come in the form of two classes *Classification*, where the algorithm should find a class representing the data it is given, and *Regression* where the algorithm should find an underlying continuous numerical function and will return a numerical value representing the input.

Typically ML can be viewed in three different groups, Supervised Learning, Reinforcement Learning (RL), and Unsupervised Learning. Where in Supervised Learning, the algorithm is given data examples and their corresponding outcome. For example, a supervised learning algorithm could be provided with data regarding the weather and the corresponding temperature, the algorithm should then find a pattern within the weather data and find the continuous function represented by the data. This would then be an example of a regression task. Flipping thing example around, if the algorithm would just be provided the temperature and it should tell us whether it is sunny outside or not, that would be a classification task.

In RL the algorithm is given only some input, and then the algorithm tries some outcome generally actions in some environment. Then over an episode [1] once the episode is finished some reward is given. The algorithm will then learn whether the actions were good actions from the reward. Examples of this are agents playing a game like Flappy Bird, where the data they are given is the state of the game, and they try to either jump or not jump.

2.5.1 Supervised Learning

In supervised learning, the data and their corresponding outcomes are already available to us. Examples of the data we can use in supervised learning would be labeled images.

[1] a series of outcomes / a timespan

2.5.2 Reinforcement Learning

Reinforcement Learning is where a model learns from experience. That is, the notion of input/output values changes, the model uses itself to generate input values by acting in an environment. The environment then returns some result, that could be losing in a game, making a correct prediction, or any number of outcomes. The result is then propagated through the model allowing it to improve with this new information.

Many algorithms are popular in reinforcement learning, for instance Q-learning(CITE ME), CARLA (CITE ME), and many others.

2.5.3 Neural Networks

Neural networks (NN) are popular methods within a sub-field of ML which is called Deep Learning (DL). NN's are created to resemble how the human brain functions. In the brain, we have neurons which when they get a signal they apply some function to them and if the resulting signal is high enough, they fire to the next neuron. This is how it is done in the neural network model as well. There we have neurons that when they get some input, generally a vector of numbers. The neuron takes the sum of that vector, weighs the sum by a constant, then applies a non-linear activation function to it. The result of doing this is then passed on to the next neuron. Until a final layer of neurons is reached. At that point, we have a value that the neural network corresponds to the input value. This value can be a binary classification (cat or dog image), a regression value (the value of a property), or any number of outputs. It can then be said that a neural network is doing a function approximation of the input to some value. $f_n(w_n * f_{n-1}(w_{n-1} * \dots f_0(w_0 * i))) = o$.

Neural networks are machine learning models that take in as input anything numerical and they apply continuous differentiable functions to that input s.t. they end with some output. This output is then compared to the expected output the difference between these outputs is called a loss. A loss is backpropagated through the neural network, and each function is derived to find its slope. We can then modify the weights

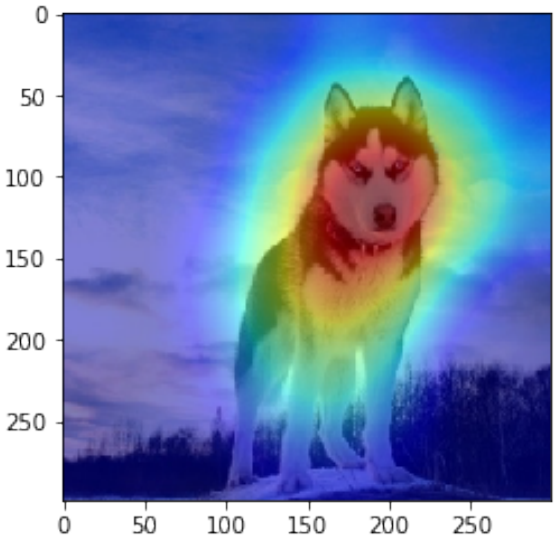

Figure 2.3: Example of a models saliency map for an image of a dog

of the neural network in the direction of the correct output. Leading to a function approximation of this function $f_{neuralnetwork}(X_{input}) = O_{expectedoutput}$

2.6 Explainable Artificial Intelligence (XAI)

The field of XAI research is still very far behind its counterpart AI research, within XAI two fields are the largest, those are Model Interpretability and Model Explainability.

2.6.1 Model Interpretability

2.6.1.1 Saliency Maps

Within XAI many methods have been developed to try to evaluate ANN's, these methods are often referred as model interpretability methods. In the field image recognition there has been a lot of work examining which pixels of an image the model deems important. Arguably the most popular method for this is Saliency Maps[2]. There the pixel values the model deems important are colored in s.t. a human can examine the image and get a sense of what portions of the image are important to the model, an example of a classification of a dog can be seen in Figure 2.3.

While this method is understandable in the context of image recognition it lacks severely when your input is not an image.

2.6.1.2 Shapley Values

Methods for explaining models that aren't image recognition models include shapley values, from the Lundberg & Lee[3]. There the input is examined against it's output, then iteratively input values are selected to be fixed. Then the other input values are varied and an average change in prediction is calculated. With this the shapley value can be estimated for the fixed input value. This is done to examine which input values have the strongest link to the output value. Shapley values on a dataset can give insights on which input values the model deems important.

2.6.1.3 Concept Activation Vectors

A recent paper by Been Kim Et. al [4], shows a method for examining a neural network giving a much more human insight into a prediction. Using Concept Activation Vectors (CAV) a directional derivative for a given input can be examined with respect to some HLC's. For example, when a human looks at an image of an animal and is supposed to decide whether the image is of a horse or a zebra, an intuitive approach would be to check whether the animal has stipes, or the animal has both white and black colors. That method of determining if a horse is a zebra could then be called a higher-level concept, and if we're able to gather if a nerual network uses this strategy for prediction we have a deeper understanding of its underlying structure. Leading to an explanation of the result.

The construction of a CAV requires a method of labelling the values in your dataset with the corresponding concept in order to create a simple binary classifier on the data points.

2.6.2 Model Explainability

Model explainability within the context of neural networks isn't possible today. Model explainability referrs to firstly considering some input and output from a model. Then afterwards the model is examined to determine exactly what led to the predicted output. This concept is simple when we're working with Decision Trees. A decision

tree is a tree whose nodes are representative of an input value and at every node a branch is selected based on the value of the input value. It is therefore easy to see how to examine the tree to explain the output. We just follow the branches in the tree. That being said the branches are created by algorithms like ID3 which construct branches based on the initial dataset used to construct the tree, but again ID3 follows simple statistics and can be explained properly.

When we talk about neural networks this process is much more difficult, the underlying nodes are generally in the thousands, the different layers of the neural network varies in the operations it applies to the input value and such while travelling through the neural network the modified value becomes far removed from the initial input value to the eyes of the reader. That being said, while the possibility of completely monitoring the training process and completely monitoring the evaluation process is truly possible it is not feasible. And secondly the process of seeing an input and it's corresponding output will not be of any value if one were to consider the process of prediction.

This part of the dissertation describes the work done.

Chapter 3

Methods

In this chapter, we describe our implementation of the game Breakthrough and the methods for training a neural network to play the game Breakthrough, described in Chapter 2. The training process uses MCTS described in Section 2.4.1 to guide its training. We outline the architecture used in the neural network.

The training algorithm is a reinforcement learning self-play algorithm based on previous work by DeepMind [5].

3.1 Game Implementation

In this section, we will take a brief look at the implementation of the game Breakthrough, how we model its state representation and the pseudocode for training.

3.1.1 Breakthrough

For a neural network to be able to use input, that input needs to be numeric. We model the board in gamethrough with a single NxM array with three values $\{-1, 0, 1\}$. Where each x, y index on the board represents what value is on the board at the time, -1 represents a white piece, 0 an empty square, and 1 a black piece.

An action vector is coupled with each state, the action vector is a 3-dimensional matrix where the lengths of the dimensions are $N * M * 6$ and each index of the matrix represents an action moving from cell x, y moving to the direction z. The directions

are as follows 0 represents moving upwards and to the left diagonally on the board, 1 upwards, and 2 upwards and to the right diagonally, 4 downwards and to the left diagonally, 5 downwards, and lastly 6 downwards and to the right diagonally.

3.2 Neural Network Architecture

The neural network architecture we opted to use was a single convolutional layer with a ReLU activation function, followed by 5 residual layers each containing two layers of convolution, batch normalization, and a ReLu activation. Lastly, a split policy/value head, where the output of the last residual layer is split into two different outputs. The policy head are two layers: first a final convolutional layer, and then a fully connected layer with a *log softmax* layer. The value head consists of three layers: first a convolutional layer, then a fully connected layer, and lastly a fully connected layer with a *tanh* activation function. The Figure 3.1. depicts the architecture.

3.2.1 Convolutional Layers

It is important to understand why we use convolutional layers when dealing with board games as convolution is often associated with an image. This is because a convolutional layer is focused on merging multiple input parameters to a single neuron, making it an input parameter for the next layer representing the locality around the center point of the original input. Playing the game of Breakthrough, a piece is only able to capture pieces in its immediate vicinity. This lead to the selection of a convolutional kernel with a size of $3x3$ and a stride of 1.

The knowledge of each piece moves isn't enough to represent the whole game state within the neural network. This is why we use multiple residual layers with convolution. These residual layers stack convolutions on each other making the final convolution represent the locality of all the other localities, allowing the neural network to have a representation of the whole game state in its parameters.

The selection of parameters was done for these reasons as well as to most closely resemble the architecture described in AlphaZero.

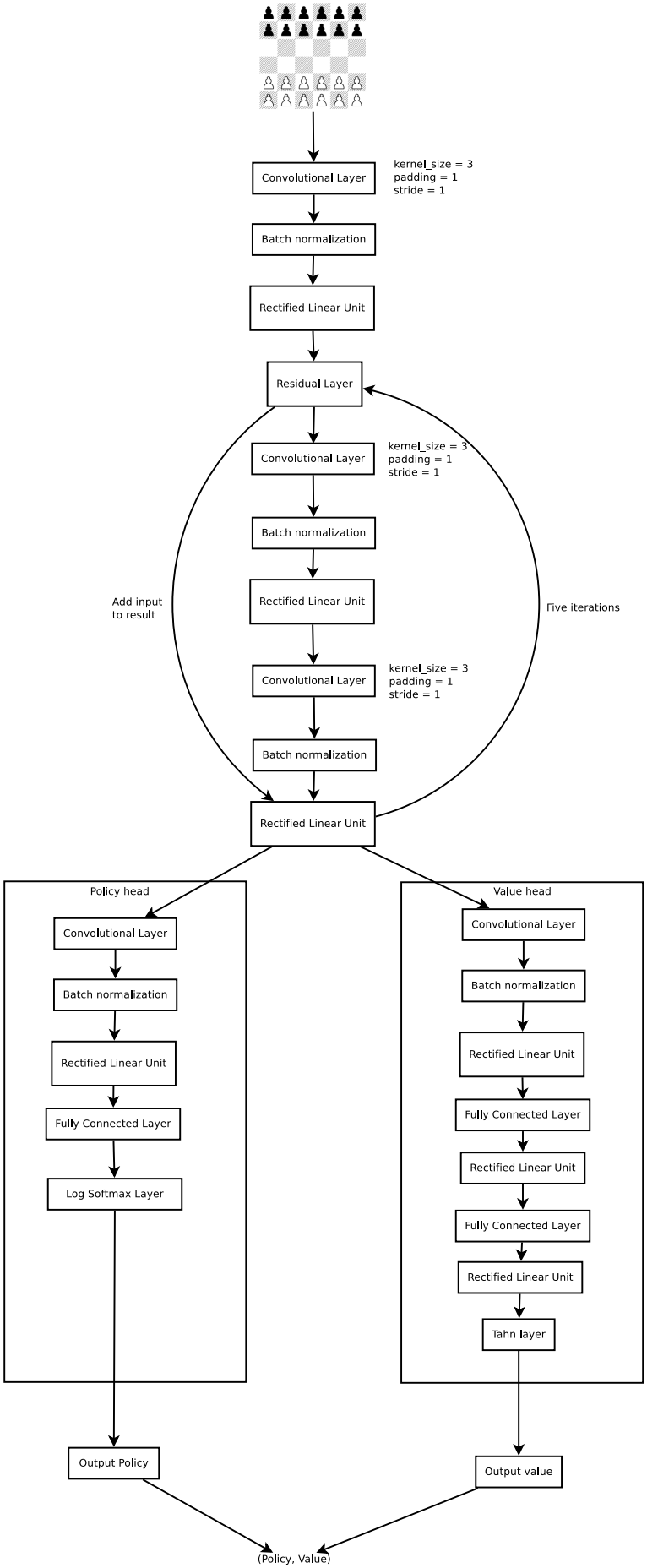

Figure 3.1: Neural Network Architecture

3.2.2 Residual Layers

The residual layers as a concept remember the output of the previous layer and add the results of the residual layer and the previous layer together. On a higher level, this leads to the internal representation of the neural network to maintain the whole game boards in its representation, s.t. two areas that are far away from each other maintain the same level of locality as two that are close.

Why we use a residual layer for applying a convolution layer multiple times is to gain locality of the whole game board. A residual layer applies a function like this $res(x) = x + l(x)$ where $l(x)$ is the function of the layer, commonly a convolutional layer.

3.2.3 Policy Head

The policy head of the neural network returns a vector of the size of the action space $w * h * 6$ where w is the width of the board h is the height of the board and 6 represents the six cardinal direction pawns can move (straight, and diagonally both left and right for the first player, and backward for the second player). The vector is then masked s.t. the values representing moves that can not be played on the board are given the value of 0.

3.2.4 Value Head

The value head of the neural network returns a single value representing the predicted value of the neural network. This predicted value is trained to be the end value of the game after taking the predicted move.

3.2.5 Implementation

This subsection focuses on the implementation of the various functions required to train the neural network to play breakthrough. The description is mainly through pseudo code, and the code is awailable on Github for examination.

Algorithm 1 Neural network selfplay pseudocode

```
1: neural network nn1 = randomizedInitialNN()
2: neural network nn2 = randomizedInitialNN()
3: while true do
4:    dataset = generate_dataset(nn1)
5:    nn1.train_on_examples(dataset)
6:    win_rate = compete(nn1,nn2)
7:    if win_rate > 0.5 then
8:       nn2 = nn1.copy()
9:    else
10:      nn1 = nn2.copy()
11:   save(nn1)
```

Algorithm 2 Neural network compete pseudocode

```
1: Input: neural network nn1, neural network nn2
2: Output: win rate for white player
3: white_wins = 0
4: game = initial_breakthrough()
5: for 1 to 100 do
6:    while not game.is_termina() do
7:       nn1.make_move()
8:       nn2.make_move()
9:    if game.white_wins() then
10:      white_wins++
11: return white_wins/100
```

3.2.6 Self-play

To train the neural network to play Breakthrough we initialize two neural networks nn1 and nn2 with random weights. Then we let nn1 play against itself using MCTS with PUCT to select moves. We collect data from this self-play to train on. The data collected is (s, π, τ, p, v), s is the state, π is the action probability vector provided by MCTS with PUCT, τ the end reward for the episode 1 if white wins, -1 if black wins. p is the action probability vector predicted by the neural network, and v is the predicted reward of the game by the neural network.

The pseudocode can be seen in Algorithm 1.

3.2.7 Compete

The compete function differs from the self-play function in such a way that we select the moves by a single pass through the neural network thereby only evaluating the neural networks ability to predict best actions. The pseudocode is available in Algorithm 2.

3.2.8 Loss Function

The data collected is backpropagated through the NN moving the weights to the direction of this loss function $l = (p * \pi) + (\tau - v)^2$ for each state the NN encountered during self-play. Importantly the variable p has masked illegal actions to 0 to direct the NN to not learn on illegal moves.

3.3 Explainable state representations

In this section we describe the process of trainin a Concept Activation Vector in order to linearly separate each state into points in a space with the concept and ones without the concept.

3.3.1 Testing With Concept Activation Vectors

Once the neural network has learned to play the game of Breakthrough, we examine its internal state w.r.t HLC's that we understand. The first examined HLC is numbers advantage, that is, the number of pieces that a player has over his opponent. The higher-level idea to a human player would be that they are in a better position since they have more pieces.

To examine the higher-level component we take a look at the internal state of the neural network itself. To do this we take a state, we run the state through the neural network, and while the state is propagating through the network we select a layer to split the network, we call that layer l_{split}. At that point in time the state is a mutated vector $l_{split}(l_{split-1}(...l_0(state)))$ of the initial state. We save that vector as a point in the N-dimensional space that it represents, and once we've gathered enough points

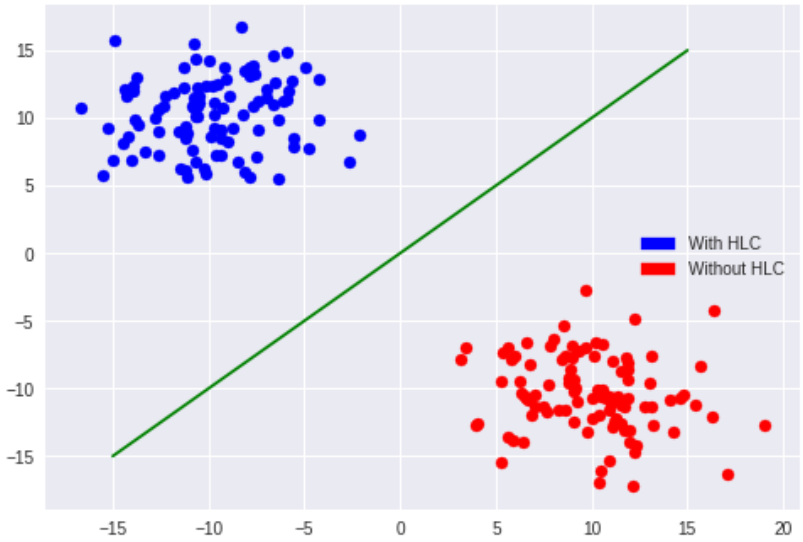

Figure 3.2: Trained linear classifier on a 2-dimensional space

in that N-dimensional space we're able to train a linear classifier using the HLC as a label. We train a Stochastic Gradient Descent classifier, from the SKLearn library, to construct a hyperplane that splits the space into two binary states, one that contains the HLC and another that doesn't.

An example of how this would look with only two dimensions is shown in Figure 3.2 Once we've successfully trained this linear classifier we can run another state s through the neural network. Once the prediction has finished, we view the selected action a of s from the policy vector p from the neural network. We view the loss of that selected action a. Then we apply the backpropagation algorithm up to that same layer l_{split} and evaluate the gradient there. If that gradient moves in the direction of the HLC we say that the state includes the HLC, and doesn't if it moves away from the HLC.

We show an example in Figure 3.3. where the arrow in the image represents the direction the gradient is moving, if it moves toward the HLC the state s includes the HLC otherwise it does not. Using this method we evaluate the internal representation of the state within the neural network and can see whether it recognizes the HLC.

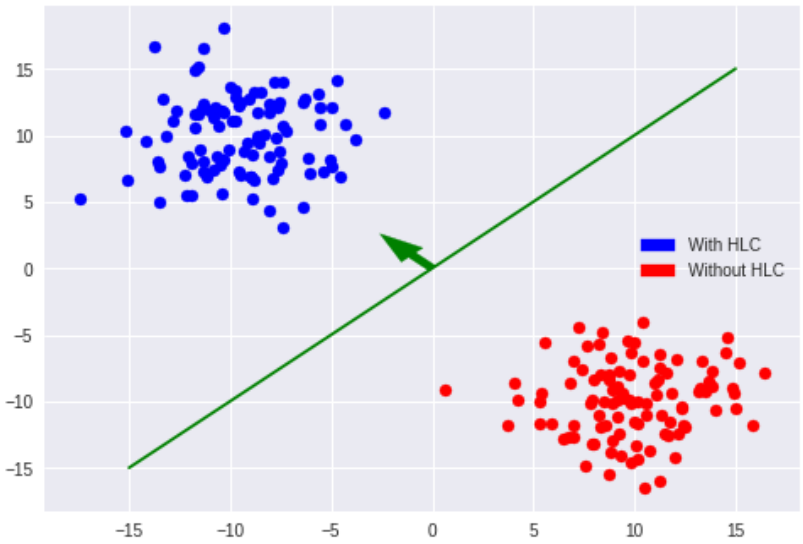

Figure 3.3: Trained linear classifier on a 2-dimensional space with arrow representing gradient

Chapter 4

Results

In this section you discuss any issues that came up while developing the system. If you found something particularly interesting, difficult, or an important learning experience, put it here. This is also a good place to put additional figures and data.In this section we discuss the results of considering HLC within the game Breakthrough, using a Concept Activation vector for evaluating how the neural network recognizes the HLC's. We first evaluate the neural network against various agents to validate that it has learned how to play the game. Secondly, we take a look at the changes in the emphasis of the neural network during training. The main point of interest there being whether the neural network notices simple HLCs early then stops considering them as the network improves.

4.1 Evolution of the neural networks win rate

Examining the neural network is only interesting if the neural network can effectively play the game. We first examine the history neural network playing against an agent that only takes random moves, on every 10 generations. The win rate over generations can be seen in Figure 4.1. It is clear that very early on in the training process, almost immediately after generation 10 our agent performs nearly perfectly against the random agent. This implies that the agent does indeed learn some strategy in the game. The next agent we tried the neural network against, was an agent that does

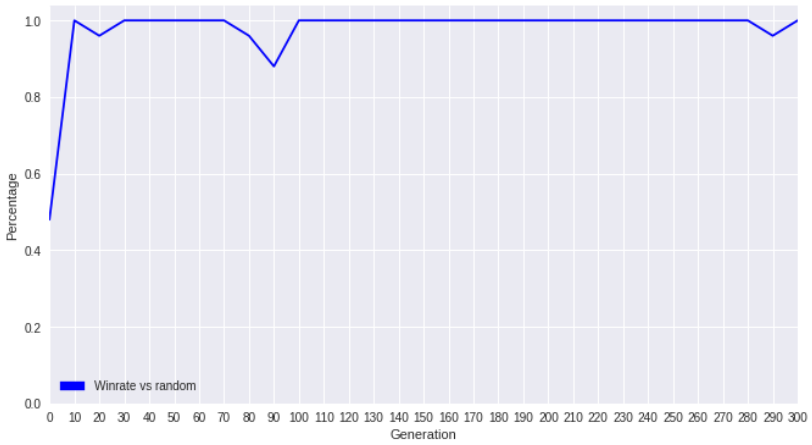

Figure 4.1: Winrate vs random agent

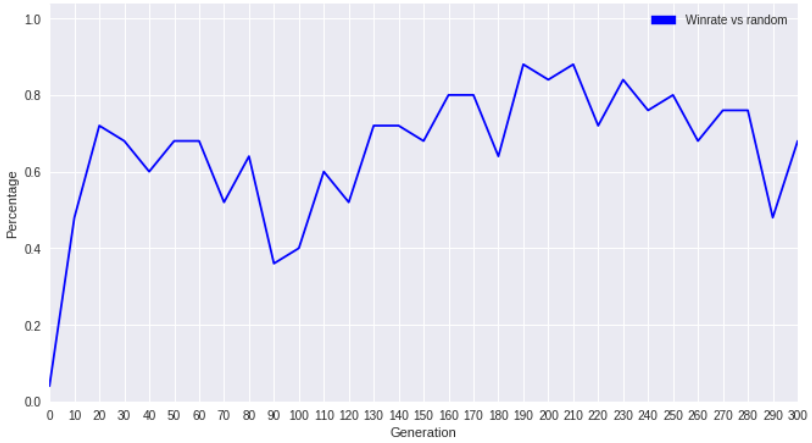

Figure 4.2: Winrate vs monte carlo tree search agent

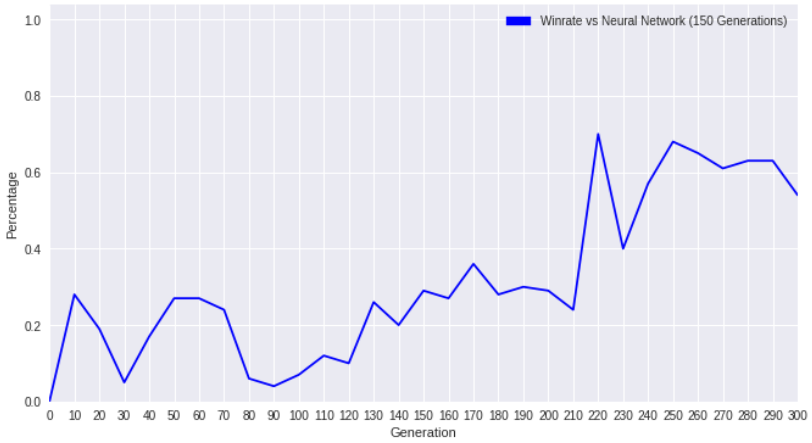

Figure 4.3: Winrate vs neural network trained for 150 generations

regular Monte-Carlo tree search on the game space with 100 iterations of rollout for each move. A graph showing the win rate over generations can be seen in Figure 4.2. Again, quickly our agent learns some strategy and can win often, but doesn't achieve a perfect win rate after 300 generations. The last agent we tested against was the neural network itself, although only trained to 150 generations. A graph showing the win rate can be seen in Figure 4.3. As expected the agent performs poorly until it has trained for more than 150 generations.

These results imply that our agent certainly understands how to play the game of breakthrough to a certain level. However, it should be stated that as we the creators attempt to compete against the agent it generally loses in a way that we would hope the agent should be able to play against. It is likely that as we train the neural network more it will become better.

4.2 Evaluating the improvements of the neural network over generations

To test which HLC the neural network places its emphasis on during training we trained a neural network for 300 generations, taking snapshots of the network every 10 generations. Then we had the neural network play against itself for 100 games collecting the states it encountered during play. These states were then examined by a concept activation vector representing these HLCs.

We test four concepts *Numbers Advantage*, *Aggressiveness*, *Unity*, and *Lorentz-Horey*.

4.2.1 Description of concepts

The first concept we examine is Numbers Advantage which conveys the idea of having more pawns than your opponent. When faced with a board it is difficult to argue for whether having a single pawn up on your opponent constitues as Numbers Advantage or three. In order to an appropriate value for this case we used the neural network that

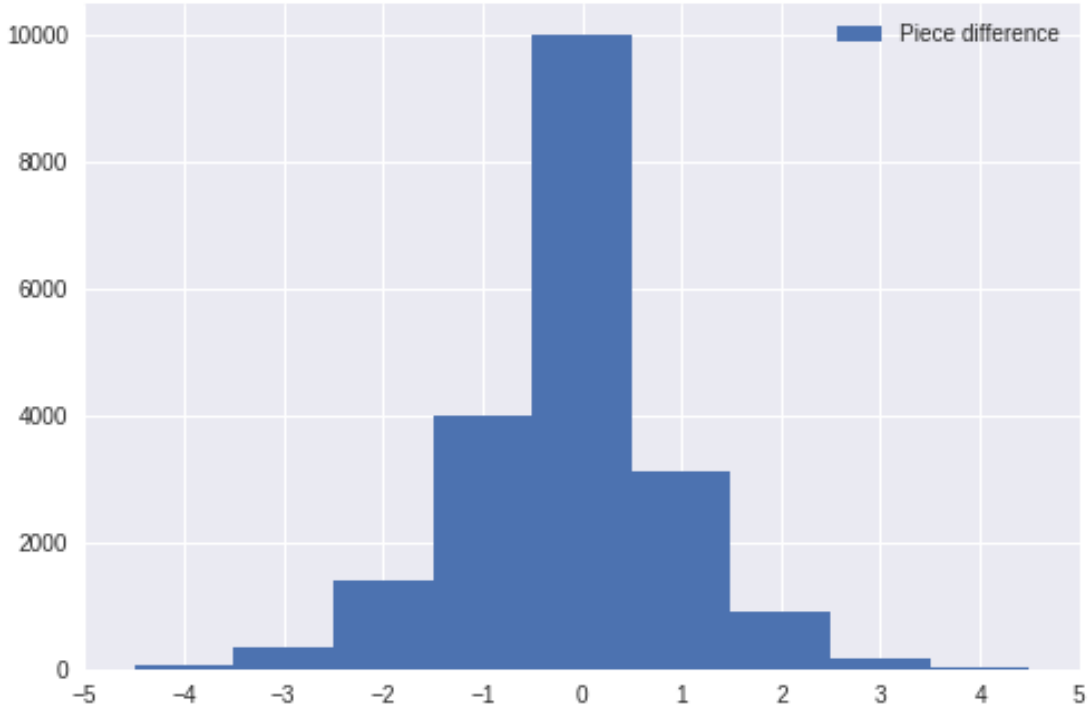

Figure 4.4: Distribution of piece amount difference

had been trained for 300 generations to generate 20000 unique states. The distribution of the difference of pawns for each player can be seen in Figure **??**. From examining the distribution and examining samples we selected the breakpoint of 2, meaning that if you have 2 pieces on your opponent you are in a state that has this concept. From the 20000 states only 5.5% of states have this property.

The next concept is Aggressiveness, which describes how much closer your furthest piece is to the opponents edge than your opponents furthest piece to your edge. This is then the minimum amount of how many moves it will take you to win the game. A distribution of these values can be seen in Figure 4.5. We examined the distribution, and sampled states from various breaking points and decided on the value of ≥ 1. That is if the value of a state's furthest piece difference is greater than or equal to 1 that state has the concept of aggressiveness.

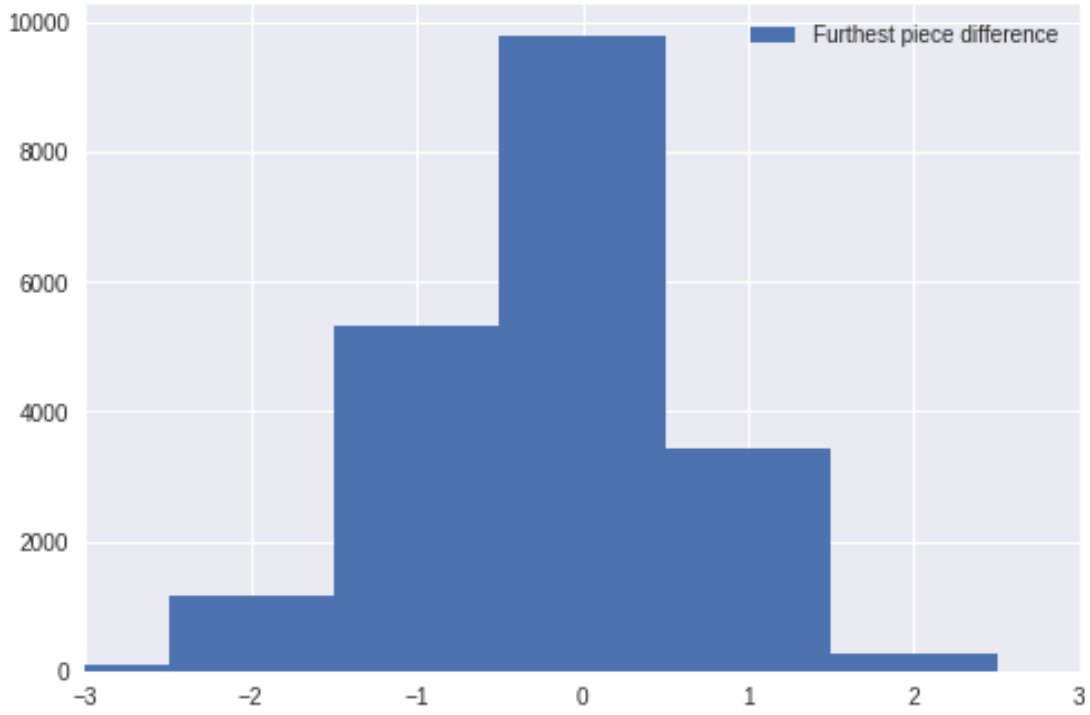

Figure 4.5: Distribution of difference of furthest pawns

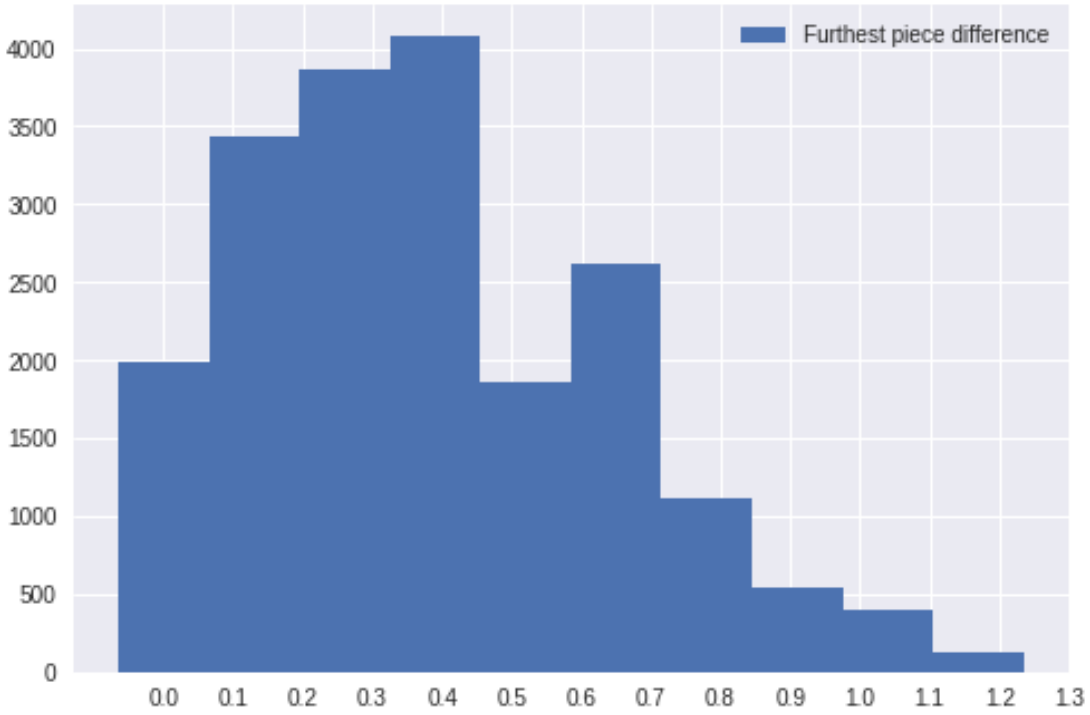

Figure 4.6: Difference of distance from center

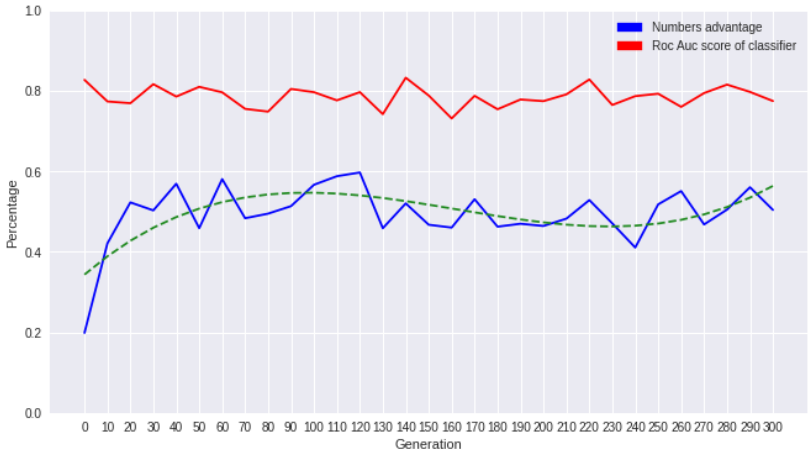

Figure 4.7: Percentage of selected states containing the HLC numbers advantage

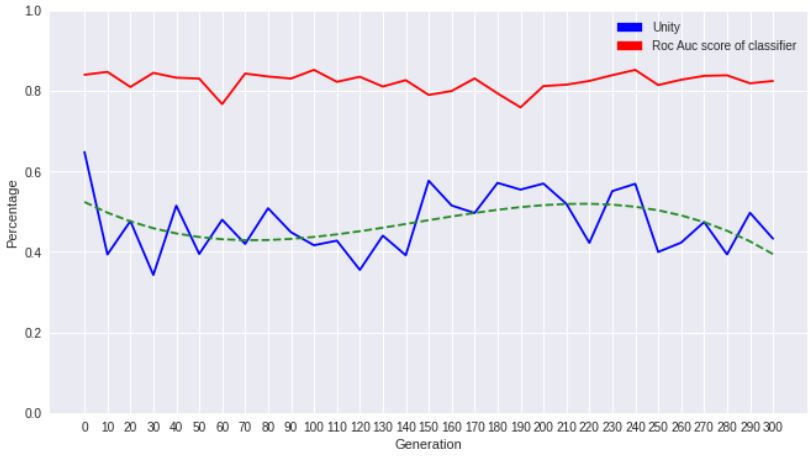

Figure 4.8: Percentage of selected states containing the HLC aggressiveness

4.2.2 Numbers Advantage

Firstly there is the numbers advantage HLC, where the number of pawns the player has minus the number of pawns the opponent had, the breakpoint we selected was 2 meaning that if you have 2 more pawns then your opponent, you're in a position where a HLC called numbers advantage is present.

The Figure 4.7. shows that throughout training the numbers advantage HLC is only ever a slight factor in the selection of states and we can say that the neural network doesn't consider number advantage in its selection process.

The second HLC we examined was aggressiveness, which is a state in which your most advanced pawn is 2 or more squares further than the opponents most advanced pawn.

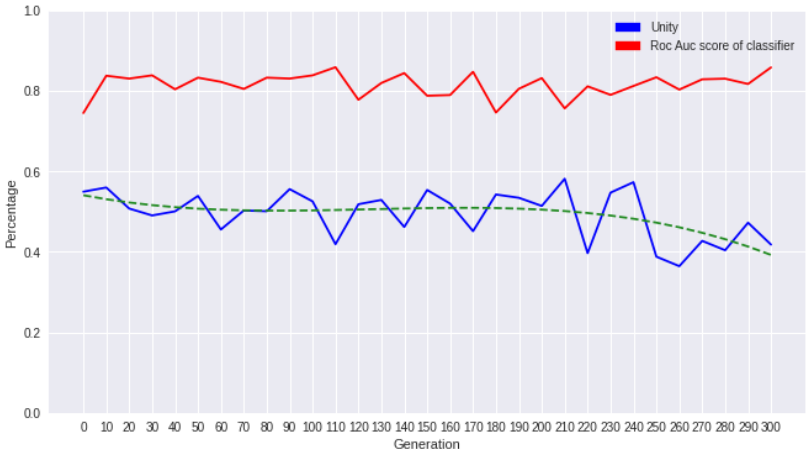

Figure 4.9: Percentage of selected states containing the HLC unity

From Figure 4.8. we can see that the aggressiveness HLC is a growing factor over as the neural network is trained. Generally, when your opponent is not skilled this strategy is considered a good one.

The third HLC that we examined was unity, the unity HLC represents the absolute average distance of your pawns from the center row of your pawns. This HLC is calculated as the row of your furthest pawn from the starting row r_{far}, the row of your nearest pawn from the starting row r_{near}. Finding the middle row is then $\frac{r_{far}+r_{near}}{2}$, we then take the absolute of the average distance from the pawns to that row. To find the point at which this value relates to a state with the HLC unity, we sampled a myriad of states and decided on the value of 0.35. The value of 0.35 generally allows your states to have two rows that have the majority of the pawns and one or two pawns one row away from the group.

The literature of breakthrough implies that being patient and waiting until your opponent makes a mistake is generally the preferred strategy for winning. the concept activation vector for Unity is not often triggered and is relatively stable at 50% througout training.

The last heuristic we examined was the Lorentz-Horey heuristic. This heuristic is a popular heuristic in breakthrough defined in the paper by Lorentz & Horey [6]. For this heuristic, we give each cell on the board a point value. For a given player the heuristic is then the sum of the cell values where they have pawns. The cell values can

5	15	5	5	15	5
2	3	3	3	3	2
4	6	6	6	6	4
7	10	10	10	10	7
11	15	15	15	15	11
21	21	21	21	21	21

Table 4.1: Lorentz-Horey cell values, from white players point of view

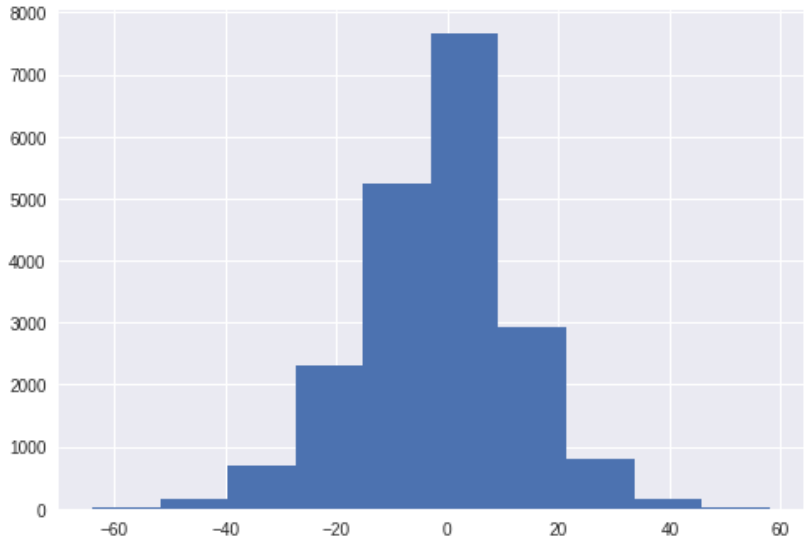

Figure 4.10: Distribution of Lorentz Horey values

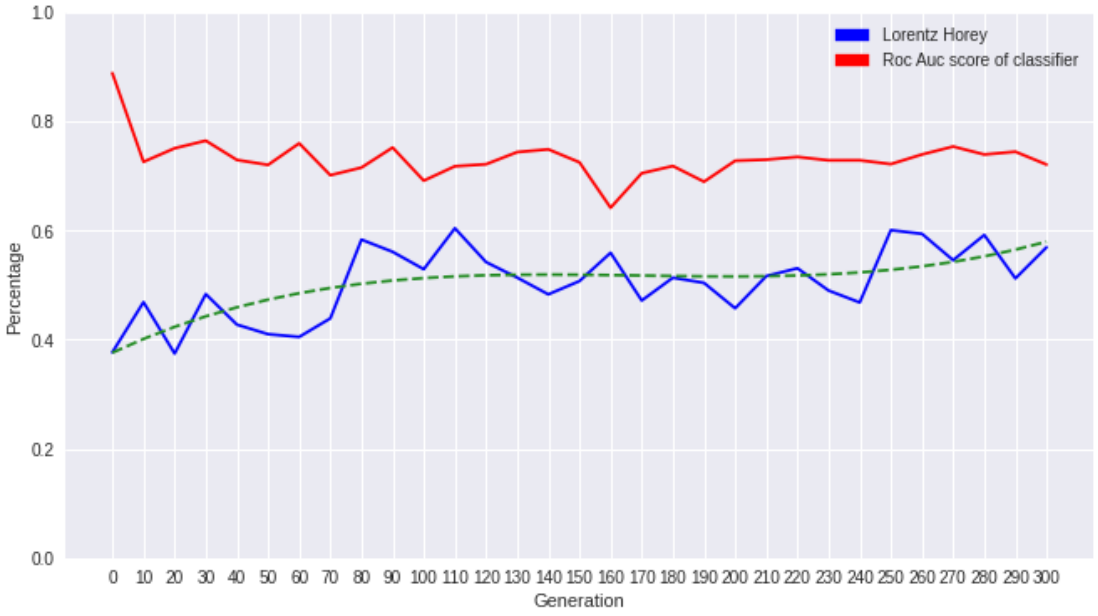

Figure 4.11: Percentage of selected states containing the HLC Lorentz-Horey

be seen in Table 4.1. the matrix shown is flipped in the black player's point of view. The values are selected in such a way that as your pawn approach the opponent's side their value increases, having pieces on the edge isn't optimal, as such a pawn will only be able to capture in one direction and can not escape capture as easily. To select the breakpoint for Lorentz-Horey heuristic we generated 20000 states and considered the values, the resulting distribution can be seen in Figure 4.10. This distribution leads us to select the value of 5, as a breakpoint indicating that a state has the HLC of Lorentz-Horey. The evolution of the neural network using this HLC for selection can be seen in Figure 4.11

Examining Figure 4.11, we can see that the trend of using this HLC increases over time. As this heuristic is the one most often cited as a successful strategy in breakthrough this result is promising.

4.3 Examining state properties

To be able to view the results from the concept activation vectors it is important to know how many states we expect to have a concept so that we can know whether the model is only guessing at random or not.

This part of the dissertation talks about future work discussion concludes the work and

Chapter 5

Discussion

Here I will discuss the myriad of fields research like this can help for instance for health organizations, patients, taxpayers, and pharmaThis section discusses the results and the future work that could span from this research. Additionally, we conclude the work.

5.1 Summary

As seen in Chapter 4. the our trained agent does indeed quickly rise to using the higher level concepts. However, there is not a lot of difference between each heuristic, and not much variability in the heuristics once it reaches $\tilde{5}0\%$. These results are not what we hoped we would see, as a concept like numbers advantage clearly isn't a great heuristic. A reasonable assumption would be that as we train the agent more it will start to use these strategies more, or to increase the size of the neural network to gain performance equal or better to that of a human player.

There is a possibility that the described HLC's are not the HLC's that a neural network uses to play the game, and thus we don't see a connection between the network and the concepts. The amount of HLC however is infinite and they must be selected carefully, we selected only ones that are common beginner heuristics in board games. A possible avenue of research would be to examine a fully trained neural network that does compete in games at a super human level, a network like AlphaZeroGo and

examine it's play style. From its play style it would be possible to gather which HLC's it focusess on, then retraining the model examining these HLC's over time to find new pedagogical method of teaching Go.

Additionally applying this method to a greater set of environment would be an interesting field of research. We envision a self driving car agent being examined with respect to a higher level concept of aggressiveness, or a mortgage agent being examined with respect to a higher level concept of racism.

5.2 Conclusion

conclude the work, discuss the significance of the workThe work done in this paper is only a first step in examining working agents in active environments with respect to human level concepts. If improved could have a great impact in our trust on neural network that improve our daily lives. While the testbed for the research was a discrete game environment, there is a clear way forward to a dynamic system with discrete human level concepts. Our results show that for an agent that clearly doesn't achieve human level performance it's actions are often selected with respect to human level concepts, up to 60% of the actions.

Bibliography

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[2] C. Koch and S. Ullman, "Shifts in selective visual attention: Towards the underlying neural circuitry", *Human Neurbiology*, vol. 4, pp. 219–227, 1985.

[3] S. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions", *CoRR*, vol. abs/1705.07874, 2017. [Online]. Available: `http://arxiv.org/abs/1705.07874`.

[4] B. Kim, M. Wattenberg, J. Gilmer, C. J. Cai, J. Wexler, F. B. Viégas, and R. Sayres, "Interpretability beyond feature attribution: Quantitative testing with concept activation vectors (tcav)", in *ICML*, J. G. Dy and A. K. 0001, Eds., ser. Proceedings of Machine Learning Research, vol. 80, PMLR, 2018, pp. 2673–2682. [Online]. Available: `http://proceedings.mlr.press/v80/`.

[5] D. Silver and et al, "Mastering the game of go without human knowledge", *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017.

[6] R. Lorentz and T. Horey, "Programming breakthrough", in *Computers and Games*, H. J. van den Herik, H. Iida, and A. Plaat, Eds., ser. Lecture Notes in Computer Science, vol. 8427, Springer, 2013, pp. 49–59, ISBN: 978-3-319-09164-8.