Strategic Evaluation of Deep Neural Networks in Board Games

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

January 2021

Examining Committee:

Yngvi Björnsson, Supervisor
Professor, Reykjavík University, Iceland

Stephan Schiffel, Examiner
Assistant Professor, Reykjavík University, Iceland

David Thue, Examiner
Assistant Professor, Carleton University, Canada

Strategic Evaluation of Deep Neural Networks in Board Games

Sigurður Helgason

January 2021

Abstract

Deep neural networks have exceeded expectations in many areas, they can generate powerful and accurate predictions of the real world. This is going to be a powerful tool for the future, and we require a method of examining them to understand why and how they come to a specific prediction to alleviate trust issues. This paper examines the strategies of a deep neural network that plays the game Breakthrough uses. We explain the process of creating a neural network that can achieve success in playing board games only from playing against itself, without any human input. We outline the process of creating a model to examine the internal representation of the neural network of the game state to evaluate whether it both recognizes and uses strategies applied by humans that play the game. These strategies are compared against each other to infer which is best, and which the neural network uses the most. We show that the neural network emphasizes strategies that are considered the most successful handcrafted strategies for the game Breakthrough. Lastly, we examine a trained neural network with a suite of simple to complex strategies to understand which is used most.

Herkænsku mat á djúptauganetum í leikjum

Sigurður Helgason

janúar 2021

Útdráttur

Djúptauganet hafa farið fram úr væntingum á mörgum sviðum, þau geta framkallað öflugar og nákvæmar spár um raunvöruleikann. Þetta mun verða öflugt tól í framtíðinni og við munum þurfa aðferðir til að geta rýnt í þessar spár til að styrkja traust okkar á þeim. Þessi ritgerð skoðar hvaða herkænsku aðferðir djúptauganet sem spilar leikinn Breakthrough nýtir sér til að taka ákvarðanir. Við útskýrum ferlið að smíða tauganet sem getur náð árangri í borðspilum með því einungis að spila gegn sjálfu sér, án inngrips frá mönnum. Við útskýrum ferlið að búa til líkan til að skoða innri framsetningu tauganets á leikstöðu til að meta hvort tauganetið þekki eða nýtir sér aðferðir sem menn hafa notað til að spila leikinn vel. Þessar aðferðir eru bornar saman til að álykta hver sé best og hver tauganetið notar mest. Við sýnum að tauganetið leggur áherslu á aðferðir sem eru taldar bestar fyrir leikinn Breakthrough. Að lokum skoðum við þjálfað tauganet með hóp af auðveldum sem og flóknum aðferðum til að rýna í hver herkænsku aðferð er mest notuð af tauganetinu.

I dedicate this thesis to my family, who while not understanding most of what I do, always support me and the friends that still want to talk to me even though my schedule has rendered me unmeetupwithable. Lastly, my girlfriend Hulda, who always helps me get through the tough times, and makes the good times even better.

Acknowledgements

Throughout the writing of my thesis I always had support and assistance from the people around me.

I would first like to thank my supervisor, Yngvi Björnsson, his knowledge in this field is vast and he is always ready to share invaluable wisdom when it's needed. Your critical feedback and consistent positivity has pushed me and my work to a higher level.

Secondly, I would like to thank my colleagues, who provided stimulating conversations on the topic and a helpful eye from a different perspective every time I needed it.

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

Chapter 1

Introduction

The world of deep learning (DL) is exciting. We have models that can examine images and reliably identify their content[1]. Deep learning models have outperformed ophthalmologists in identifying diabetic retinopathy[2] and identified cancer cells where others have not[3]. Deep learning models are used for the bleeding edge of protein folding, to gain further understanding of the underlying structure of organisms, such as viruses[4]. These models have do these things 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 in the incredibly expansive game of Go[5]. 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 a 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. Not only to improve the models themselves but also to augment our ability in many cases. Furthermore, these explanations are a legal requirement now in many areas[6], namely, legal and medical. Recently, XAI has been focused on Model Interpretability (MI), in particular, interpreting the model in such a way that we can predict what the model will do given a particular input[7]. Another class of XAI is Model Explainability (ME), where we attempt to adequately explain how the internal mechanics of the model works. A simple example of ME would be when we explore a decision tree generated by an ML algorithm. The decision tree nodes and conditions explain exactly what the tree will do with a given input. This differs from MI where 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 a game. Such a game can be simple like Tic-Tac-Toe or Breakthrough and 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 using some heuristic. A heuristic within games are evaluations of the expected end reward for a state, for example, by using the number and type of pieces left within a game of chess. Intuitively, the material advantage is a higher-level concept we can use to evaluate a chess position. Of course there are many other relevant concepts to consider, for example whether the king is safe from attack, and the pawn structure. This thesis examines the evaluation of a neural network of a given state with respect to those higher-level concepts. More precisely, we define the following research questions:

1. Given a NN that plays Breakthrough, can we determine if that NN has learned the HLCs that human players use.

2. Does a NN that trains itself using self-play learn some HLCs over time, and does it start to emphasize the HLCs that are generally considered better the more it trains, and conversely, does it start to demphasize HLCs that are considered worse.

Summary

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 external 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, and explain the algorithms we used. Additionally, we discuss similar methods for searching state-spaces. We allocate a large portion of the section to Breakthrough as that will be our testbed. We discuss the different classes of machine learning as well as neural networks.

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 characteristics. The characteristics that are commonly used to describe environments can be seen in Table 2.1[8].

Categorizing environments using Table 2.1 gives one the power to find an environment in which a method works and know it can be applied to different environments with the same characteristics. Additionally, it allowes us to talk about agents in the context of environments as the entities that act within the environment.

Table 2.1: Characteristics of environments

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 agents execute deterministicly impact the environment.
Episodic	Sequential, Episodic	Are actions episodic or sequential.
Static	Static, Semi-Static, Dynamic	Does the environment change without agent input, or does it wait until agents take actions.
Discrete	Discrete, Continuous	Is your environment discrete w.r.t actions.

2.2 Game Environment

Classical artificial intelligence game environments are commonly used to validate a method, e.g game environments can be games like Tic-Tac-Toe or Breakthrough, or simluation environments like driving simulators. Game environments are a suitable place to apply AI as they serve 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 $M \times N$ board with squares like in chess, and each player starts with two rows of pawns at opposite ends. The objective of the game is for a player to move one of their pawns 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 both move and capture diagonally. This leads to the game being impossible to draw as pieces are always able to move or capture. An example of an initial board in Breakthrough can be seen in Figure 2.1.

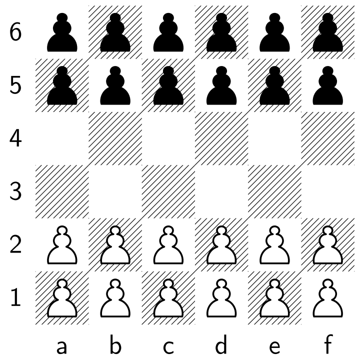

Figure 2.1: Initial Breakthrough board

Table 2.2: Categorization of Breakthrough

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

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 players compete.

2.3.1 Heuristics of Breakthrough

To evaluate the game of Breakthrough we may consider many heuristics (higher-level concepts), for instance a very simple heuristic would be a players material advantage. *Material Advantage* is the amount of pieces the player has minus the amount of pieces the opponent has. This heuristics gives us some insight into how well the game is progressing, but obviously, there are cases where this does not tell us much, for example when your opponent has a single piece left that is on the row immediately before the row needed for them 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.

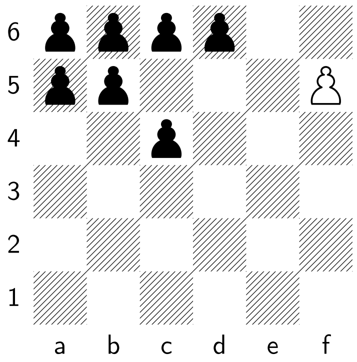

Figure 2.2: Breakthrough board where Material Advantage doesn't work well

A different heuristic would be the distance of your most advanced pawn minus your opponent's most advanced pawn; this 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 the win. Generally, in Breakthrough, it is favorable to move your whole team as a unit and play more defensively[9]. More advanced heuristics for Breakthrough will be discussed in detail later in a later chapter.

2.4 State-Space Search

Traditionally, methods for playing games search through the environment using a heuristic to guide the search. A simple way of doing a heuristic-based search would be to give all non-terminal states a 0 score and terminal states positive or negative scores based on whether it is a win or a loss, repesctively. We say that that a search algorithm is not guided, and the algorithm will probably have to evaluate a large portion of the state-space. This method of searching is generally extremely inefficient as the state-spaces of game environments are often extremely large, even infinite. For instance, an upper-bound estimate of the state-space for Breakthrough is $3^{(M-1)*(N-1)} + 2 * N$ where M is the height of the board N is the width of the board. The $3^{(M-1)*(N-1)}$ represents each position of the board having either a white, or black piece, or being empty. And, the $2 * N$ component represents each square the final piece to move could have moved to. So for a small board, 5×4 the upper-bound estimation of the state-space is 531,449 states.

This is why a good heuristic is needed because we can identify paths in the game tree that will lead to a bad outcome early, allowing us to disregard a significant amount of states, reducing the time required to search.

The algorithms that are used in traditional state-space search are for instance Depth-first search (DFS), Breadth-first Search (BFS), Alpha-Beta Search (AB-Search)[10], and Monte-Carlo Tree Search (MCTS).

More modernly, these search methods have been augmented 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[11]. 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 but rather receive the evaluation from the model.

2.4.1 Monte-Carlo Tree Search

In the algorithm Monte-Carlo Tree Search, described by R. Coulom[12], there is an agent within an environment. Each node in the environment represents a state of the environment, and each edge from the node represents a possible action from that state s to a child state s'. Often, these concepts are considered state-action pairs, which should be unique within the environment. And as such, they create a tree where we are able to search the state-space.

MCTS is a method of exploring an environment in a best-first manner. 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. This tree consists of nodes n_i where i represents the point in time of game state, 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 reward from running the MCTS algorithm from this node, 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.

We describe the MCTS algorithm's four phases: Selection, Expansion, Simulation/Rollout, Backpropagation, in more detail below.

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

Selection

During the selection phase, a node s within the tree which has not yet been expanded is found. This process uses Upper Confidence Bound on Trees (UCT)[13] to find that node s, the formula is described in Equation 2.1. For a parent node s_{parent} (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 s' nodes, the nodes are generated by applying all actions a in s.

Rollout

Next during rollout, actions a from s are randomly selected to move to s', then repeated to go to s'', 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, or -1 black wins).

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).

2.4.2 Neural Network based UCT

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 $f((s, a)) = (p, v)$, where f is the neural network, p is the policy vector returned by the neural network and v is the predicted value from the neural network. The resulting formula is described in Equation 2.2, and is called PUCT[14]. 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.

$$\text{Child PUCT value} = Q(s, a) + c_{puct} * p(s, a) * \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)} \tag{2.2}$$

2.5 Machine Learning

Machine Learning (ML) is a research field in which machines apply statistical functions on data to achieve a correct output, where by *correct* we mean the corresponding result that 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.

We can categorize machine learning into three different sub-categories: Supervised Learning, Reinforcement Learning, and Unsupervised Learning.

2.5.1 Supervised Learning

In Supervised Learning, the ML algorithms attempt to build a model from a data set of labelled examples. The labelled examples are a set of input values and their corresponding output value. The ML algorithm then uses this data set to construct a model that is as accurate as possible at outputting the correct output value given the input value. In supervised learning many techniques are applied to maintain the generality of the model s.t. it does not just represent the data set it is given but also has high accuracy on a possible future data set it has not yet encountered.

Some examples of algorithms that are popular for Supervised learning would be, Decision Trees, Support Vector Machines, or Naive Bayes.

2.5.2 Reinforcement Learning

Reinforcement Learning focuses on the idea of trial and error for an ML algorithm, where the algorithm directly interacts with the environment it operates in, and from operating in the environment it is provided with either positive or negative feedback for its actions. Typically, the environment needs to be modeled as a Markov Decision Process. That is, the selection of actions in a state requires only the knowledge of being in that state, not the actions it took to get to that state.

Many algorithms are popular in reinforcement learning, for instance Q-learning[15], and TD-Lambda[16].

2.5.3 Unsupervised Learning

In Unsupervised Learning, the machine learning algorithms attempt to build a model from a data set of values that do not have a corresponding output value. These algorithms then generally attempt to find pattern within the data set, to which we could then later label upon examination of the patterns. Importantly, in Unsupervised Learning, all columns of values in the data set should be normalized to the same range, and should be standardized s.t. the mean of the values is 0 and it's standard deviation is 1. This is done in order for one value not to dominate the patterns in the data set.

Common algorithms in Unsupervised Learning are, K-Nearest Neighbours (KNN), K-Means[17], and DBScan[18].

2.5.4 Neural Networks

Neural networks (NN) are popular methods within a sub-field of ML known as Deep Learning (DL). Neural networks are generally used on a labelled dataset. NN's are created to resemble how the human brain functions. In the brain, we have neurons which when they receive 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. In NNs we have a layer of neurons f_i, initial layer being f_0 and the last layer being f_n. These layers behave in such a way that when they get some input, generally a vector of numbers, each neuron in the layer takes the sum of that vector, weighs the sum by a learned weighing factor, then applies an 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 reached a layer that corresponds to the prediction of the network. This value can be a binary classification (cat or dog image), a regression value (the value of a property), or even the neural networks representation of the input, e.g where a neural network attempts to create an image from an example image[19]. It can then be said that a neural network is doing a function approximation of the input to some value. And, would be mathematically stated as $f_n(w_n * f_{n-1}(w_{n-1} * \dots f_0(w_0 * i))) = o$.

The core of the learning in neural networks comes from optimizing the weighing factor. That weighting factor is gradually made to move in the direction of the actual correct output by algorithms, e.g the backpropagation algorithm[20].

2.6 Explainable Artificial Intelligence (XAI)

The field of XAI research is still far behind its counterpart AI research[21]. Within XAI two fields are the largest: Model Interpretability and Model Explainability.

2.6.1 Model Interpretability

Model Interpretability is the more common approach of XAI, mainly because it is less constrained than model explainability. In order to achieve model interpretability, we must be able to answer the question of what prediction the model will return on a given input, with a high accuracy. Simple examples would be, given a trained neural network and a picture of a dog, if that model is highly interpretable, we can say with high certainty that the predicted value will be dog.

2.6.2 Model Explainability

Model explainability within the context of neural networks is not possible today. Model explainability refers 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. By following the branches in the tree we can exactly explain why the model predicted the output.

When we talk about neural networks this process is much more difficult; the underlying nodes are generally in the millions and the different layers of the neural network vary in the operations they apply to the input. During this process, the value is modified such that it becomes far removed from the initial input value. That being said, while the possibility of completely monitoring the training process and completely monitoring the evaluation process is possible it is not feasible.

2.6.3 Saliency Maps

Within XAI many methods have been developed to try to explain ANN's. In the field of image recognition there has been a lot of work examining which pixels of an image the model deems important. One such method is Saliency Maps[22]. There the pixel values the model deems important are colored s.t. a human can examine the image

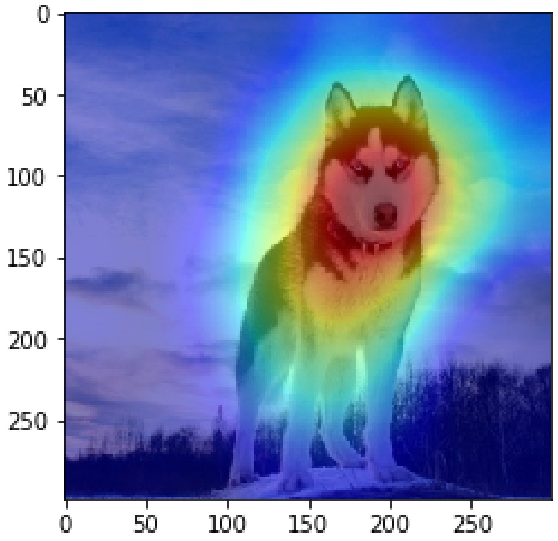

Figure 2.3: Example of a model's saliency map for an image of a dog

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. This method is understandable to a human when the saliency map lines up to what we would focus on. However, this achieves no explanation on the prediction if the saliency map doesn't line up to human intuition or the prediction, for example, if in the same Figure 2.3 we had the same saliency map but the prediction would be dragon, or if the salience map was on the trees and the prediction was dog.

2.6.4 Shapley Values

Methods for explaining models that are not image recognition models include Shapley values, from Lundberg & Lee[23]. There the input is examined against its 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 most important.

2.6.5 Concept Activation Vectors

A recent paper by Been Kim et al.[24], shows a method for examining a neural network giving a much more humanlike 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 binary classifier on data. The binary classifier is constructed on the internal representation of the data points within the neural network. After training the neural network, and constructing the classifier, we run a new datapoint through the neural network and we examine the directional derivative of that datapoint. If the direction is in the direction of the binary classifier we say that the datapoint contains the concept.

2.7 Summary

This section discussed some of the important algorithms related to the research done in this thesis, as well as giving a foundation in the readers understanding of the test bed we will be using in future sections.

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. The training process uses MCTS, as described in Section 2.4.1, to guide its training. We outline the architecture we used for our 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 look at the implementation of the Breakthrough neural network, how we model its state representation and the pseudo-code for training.

3.1.1 Description of the input and output

For a neural network to be able to use input, that input needs to be numeric. We model the board with a single $N \times M \times 3$ array with two values $\{0, 1\}$, where the first two indices on the z axis represent the players, and the last layer on the z axis represents which players turn it is. The first z index represents whites pawns, that is, each (x, y) position on the board where white has a pawn has the value 1 otherwise it has 0, and similarily on the second layer for the black player. The last z layer, contains all 1's if it is the white players turn otherwise all 0's.

One of the outputs of the neural network is the policy vector. The policy 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, 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. The cell values indicate the probability distribution of the search selecting that move. Importantly, before we select a move from the neural networks prediction we zero out the illegal moves, and renormalize the array such that it sums to 1.

3.2 Neural Network Architecture

The neural network architecture we opted to use was a single convolutional layer with a ReLU[25] 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. Figure 3.1 depicts the architecture. Using this architecture we end up with 1,737,986 trainable parameters.

3.2.1 Convolutional Layers

It is important to understand why we use convolutional layers when dealing with board games as convolution is more typically 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 3-d convolutional kernel with a size of $3x3$ and a stride of 1.

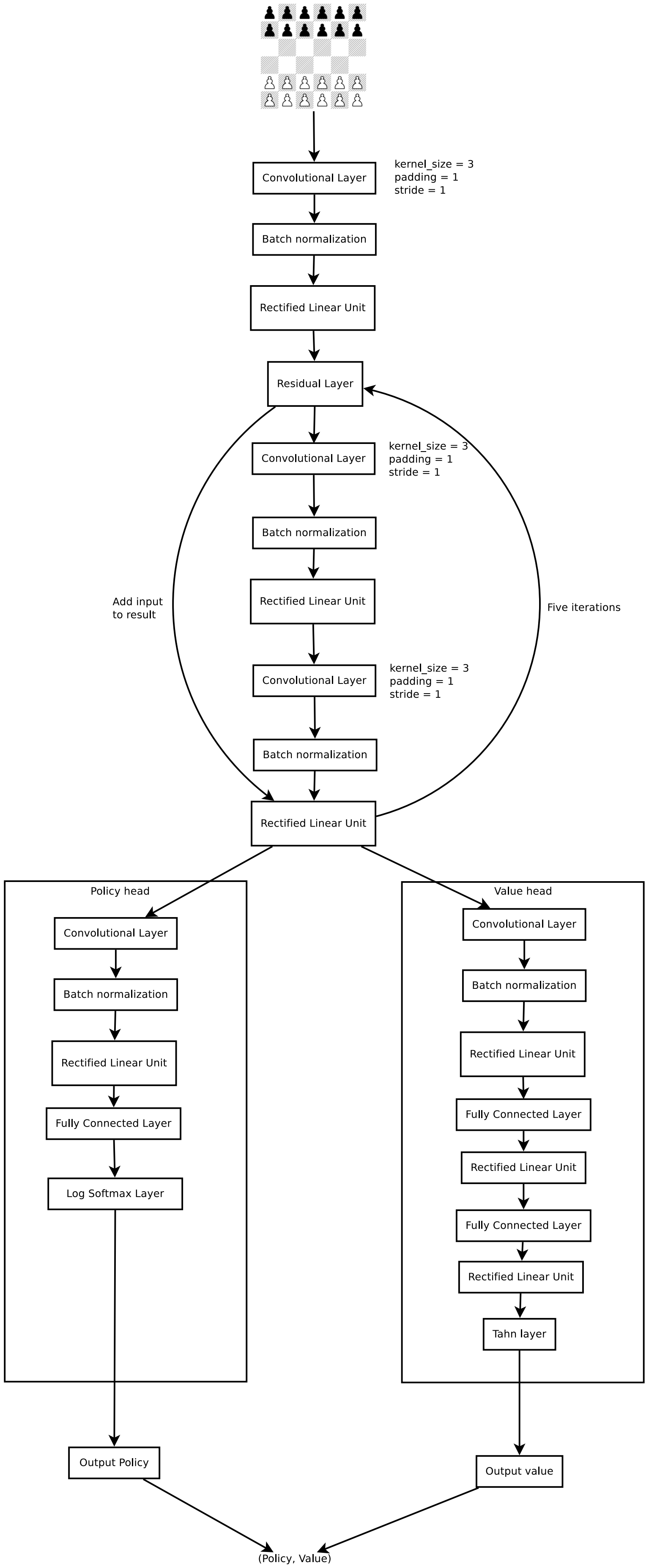

Figure 3.1: Neural Network Architecture

We require a global view of the board. 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[26].

3.2.2 Residual Layers

The residual layers add the output of the previous layer and the results of the residual layer. 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.

The reason for 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 x is the input and $l(x)$ 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.

Algorithm 1 Neural network self-play pseudo-code

```
neural network nn1 = randomizedInitialNN()
neural network nn2 = randomizedInitialNN()
while true do
    dataset = generate_dataset(nn1)
    nn1.train_on_examples(dataset)
    win_rate = compete(nn1,nn2)
    if win_rate > 0.5 then
        nn2 = nn1.copy()
    else
        nn1 = nn2.copy()
    save(nn1)
```

3.3 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 available on GitHub for examination [27].

3.3.1 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), where s is the state, π is the action policy vector provided by MCTS with PUCT, τ the end reward for the episode 1 if white wins, -1 if black wins. p is the policy vector predicted by the neural network, and v is the predicted reward of the game by the neural network.

The pseudo-code is shown as Algorithm 1.

3.3.2 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 pseudo-code shown as Algorithm 2.

Algorithm 2 Neural network compete pseudo-code

```
Input: neural network nn1, neural network nn2
Output: win rate for white player
white_wins = 0
game = initial_breakthrough()
for 1 to 100 do
    while not game.is_terminal() do
        nn1.make_move()
        nn2.make_move()
    if game.white_wins() then
        white_wins++
return white_wins/100
```

3.3.3 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.

3.4 Explainable State Representations

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

3.4.1 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 material advantage, that is, the number of pieces that a player has over the opponent. The higher-level idea to human players 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, and run it through the neural network, and while the state is propagating through the network we select a layer to

Algorithm 3 CAV creation pseudo-code

```
Input: neural network nn, column_name, breakpoint, data_set, sample_size
Output: A concept activation vector for the column with column_name
positive_set = select_positive_samples(data_set, column_name, sample_size)
negative_set = select_negative_samples(data_set, column_name, sample_size)
labelled_set = positive_set + negative_set
unlabelled_set = dataset - unlabelled_set
model = StochasticGradientDescentClassifier()
for state in labelled_set do
    //Modify labelled dataset to contain internal representations
    state = nn.get_internal_representation(state)
train_set, test_set = train_test_split(labelled_dataset)
model.train(train_set)
prediction_score, roc_auc_score = model.predict(test_set)
return model
```

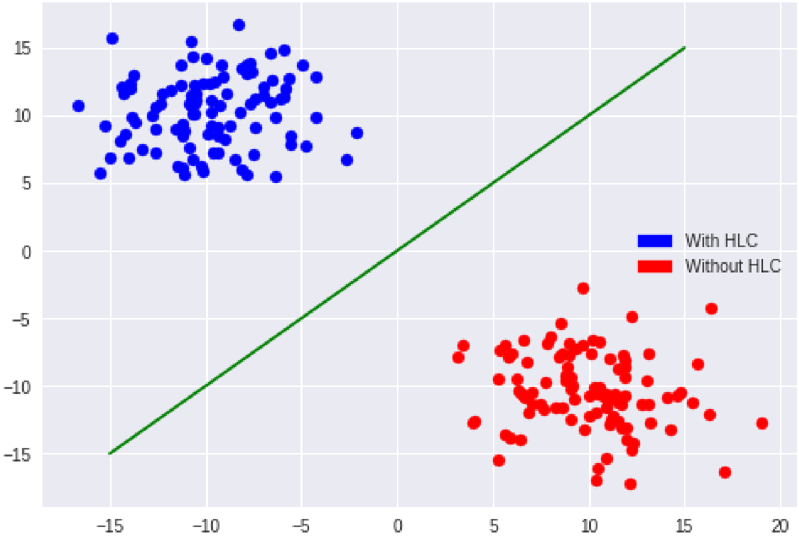

Figure 3.2: Trained linear classifier on a 2-dimensional space

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 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 hyper-plane that splits the space into two binary states, one that contains the HLC and another that doesn't. We show pseudo-code of this process in Algorithm 3.

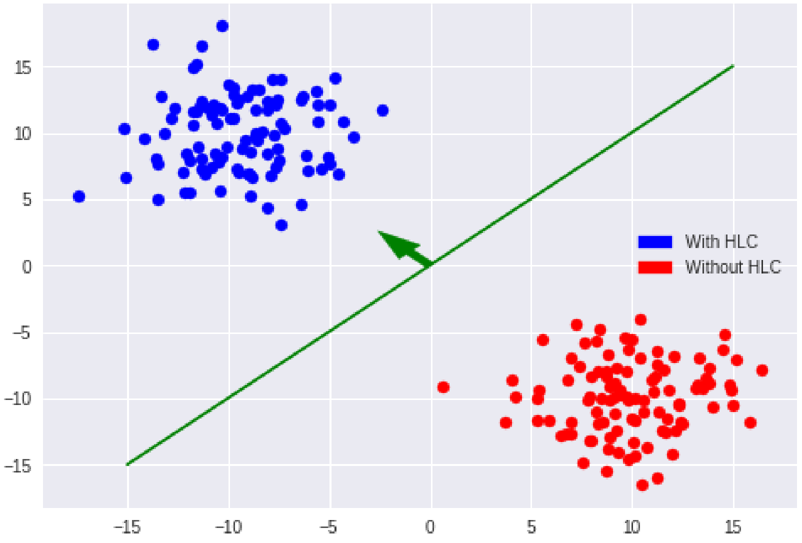

Figure 3.3: Trained linear classifier on a 2-dimensional space with arrow representing gradient

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 then 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 that it doesn't if the gradient 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.

In Algorithm 4 we show pseudo-code depicting how we evaluate if a neural network recognizes a given concept. The resulting value from running the algorithm is, the proportion of states that the neural network encountered during playing against it self that contain the HLC.

Algorithm 4 Evaluate a neural network w.r.t concepts pseudo-code

```
Input: neural network nn, concept activation vector cav
Output: Portion of encountered states that contain the concept of the concept activation vector
state_count = 0
has_tcav = 0
for 1 to 100 do
    current_state = Breakthrough.initial_state
    while ! current_state.terminal() do
        current_internal_representation =
        nn.get_internal_representation(current_state)
        output = nn.finish_propagating(current_internal_representation)
        gradient_direction =
        nn.get_gradient(current_internal_representation, output)
        tcav_score = cav · gradient_direction
        if same_direction(tcav_score, cav) then
            has_tcav++
        state_count++
        current_state = nn.get_next_state(current_state)
return has_tcav / state_count
```

3.5 Summary

In this chapter we described the architecture of the neural network, including how we model Breakthrough to be able to learn to play the game, we described the processes we use to learn from self play. In the following chapter we will evaluate the neural network.

Chapter 4

Results

In this chapter, we first evaluate the neural network against various agents to validate that it has learned how to play the game. Secondly, we describe each concept we intend to examine the neural network against. Thirdly, 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 of the neural network playing against

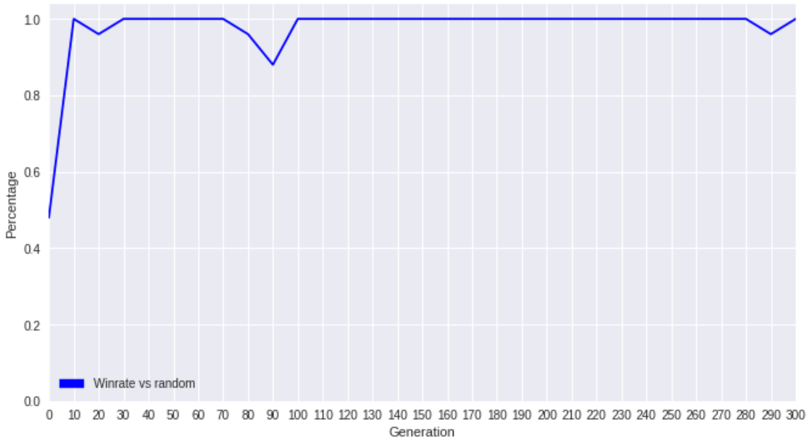

Figure 4.1: Winrate vs. random agent

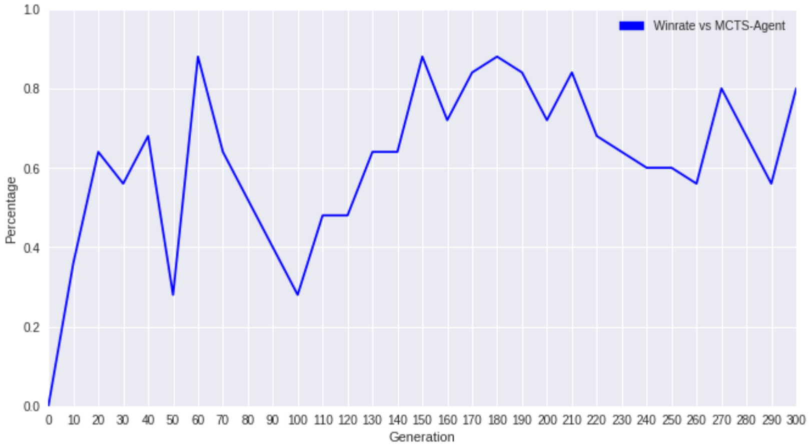

Figure 4.2: Winrate vs. MCTS agent

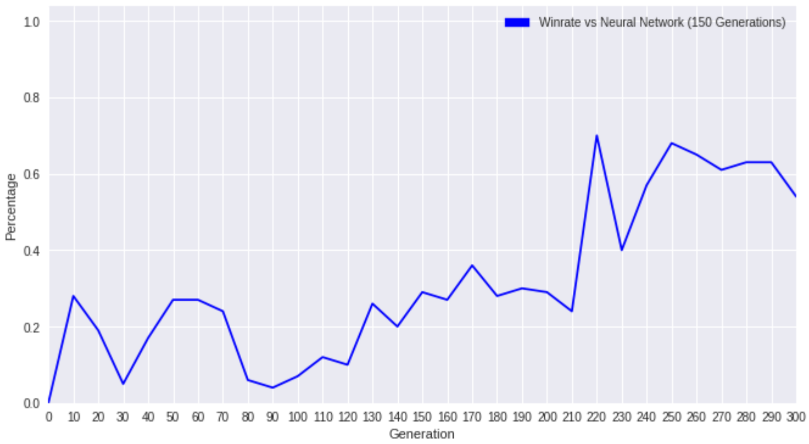

Figure 4.3: Winrate vs. neural network trained for 150 generations

an agent that only takes random moves. The win rate over every 10 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 regular Monte-Carlo tree search on the game space with 100 iterations of random UCT rollout for each move. The MCTS agent has a c_{uct} value of 0.9. A graph showing the win rate over generations is depicted in Figure 4.2. Again, our agent quickly learns some strategy and can win often, although not achieving a perfect win rate even after 300 generations. The last agent we tested against was the neural network itself, although only trained to 150 generations. The graph in Figure 4.3 shows win rate. As one would expect, the agent that has trained for less time 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, an intermediate level human player would most often be able to beat the agent, based on the author's experience playing against it. Note that the main focus of this was not to create an expert level agent. Rather, to gauge into the concepts the agent learns as described in the next section.

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. We then 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 *Material Advantage*, *Aggressiveness*, *Unity*, and *Lorentz-Horey score.*

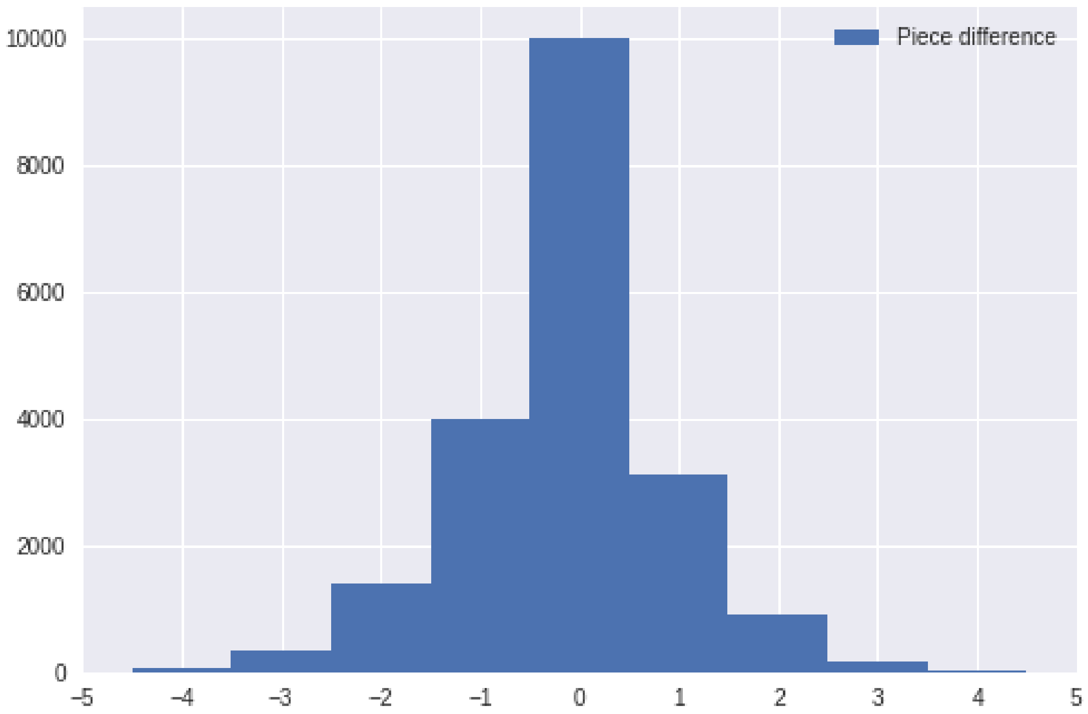

Figure 4.4: Distribution of piece amount difference

4.2.1 Description of Concepts

The first concept we examine is Material Advantage which conveys the idea of having more pawns than the opponent. When faced with a board it is difficult to argue for whether having a single pawn or three up on your opponent constitues as significant Material Advantage. In order to select an appropriate value for this case we used the neural network that had been trained for 300 generations to generate 20,000 unique states. The distribution of these states' difference of pawns can be seen in Figure 4.4. 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 20,000 states only 5.5% of states have this concept.

The next concept is Aggressiveness, which describes how much closer your furthest piece is to the opponent's edge than your opponent's furthest piece to your edge. This is the minimum amount of how many moves it will take you to possibly 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

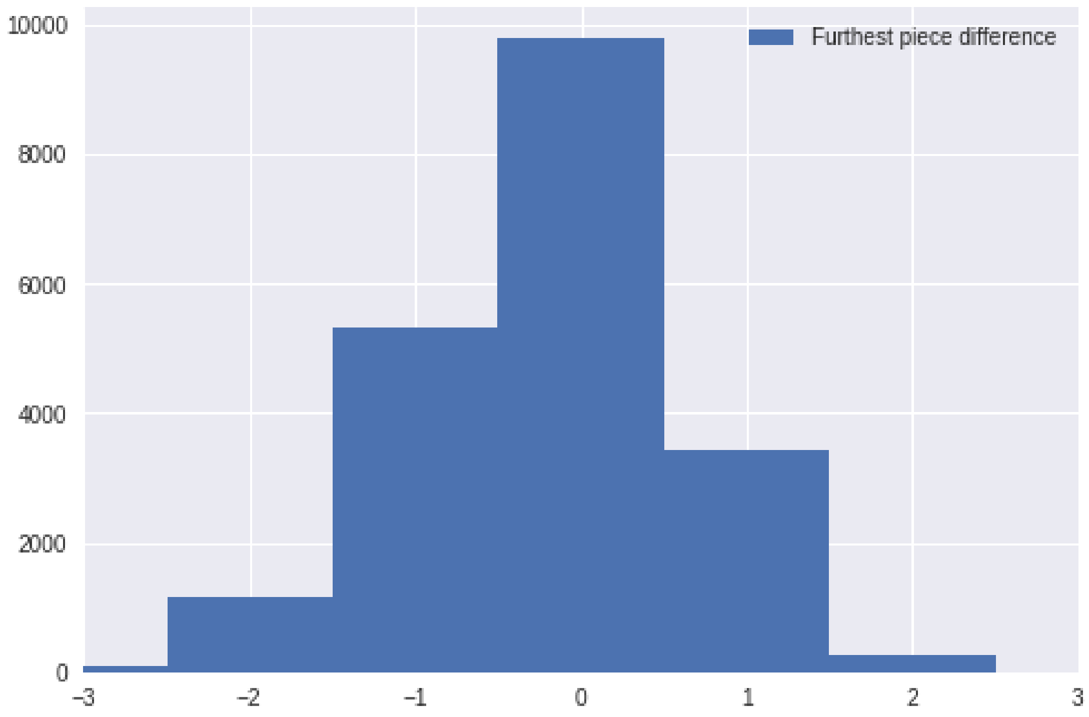

Figure 4.5: Distribution of difference of furthest pawns

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. From the dataset 18.3% of states have this concept.

Our third concept is Unity. This concept represents the absolute average distance of your pawns from the center row of your pawns, and is calculated as follows: 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 all of the players pawns to that row. The distribution of the average distance values is shown in Figure 4.6. From sampling states from several points we decided on a breakpoint of ≤ 0.25 for identifying states as states where the Unity concept is precent. The value of 0.25 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. From the dataset of 20,000 states 20% of states have this concept.

The last concept we examined was the Lorentz-Horey score. This concept is a popular heuristic in Breakthrough defined in the paper by Lorentz & Horey [9]. For this heuristic, we give each cell on the board a point value. For a given player the

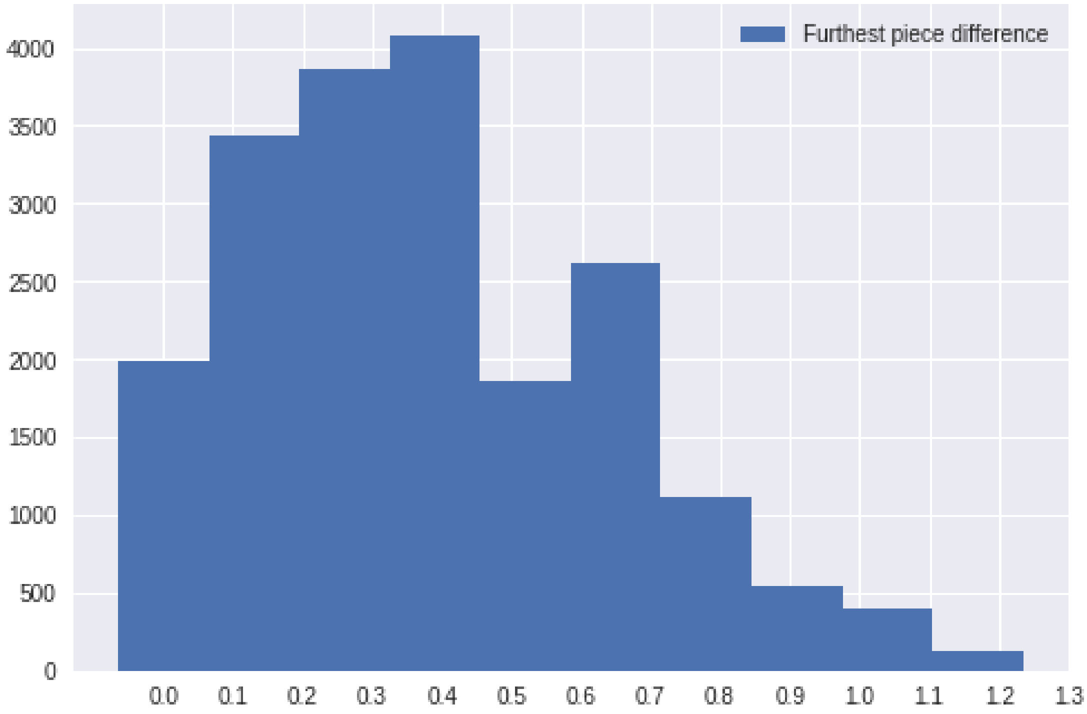

Figure 4.6: Difference of distance from center

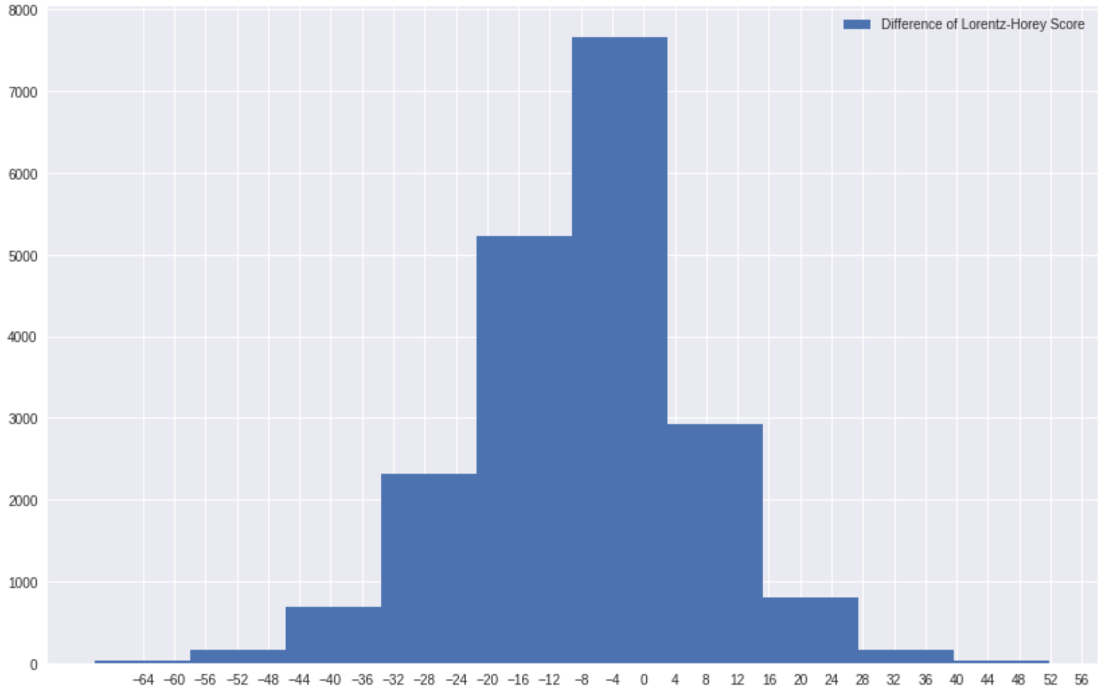

Figure 4.7: Difference of Lorentz-Horey Score

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

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

heuristic is then the sum of the cell values where they have pawns. The cell values are shown in Table 4.1. The matrix shown is flipped from the black player's point of view. The values are selected in such a way that as your pawns 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 generate a distribution of the difference in Lorentz-Horey Score over our dataset. The resulting distribution is shown in Figure 4.7. Our examination lead us to select the value of 5, as a breakpoint indicating that a state has the concept of Lorentz-Horey. Out of our dataset of 20,000 states 29.4% of states have this concept.

4.2.2 Material Advantage

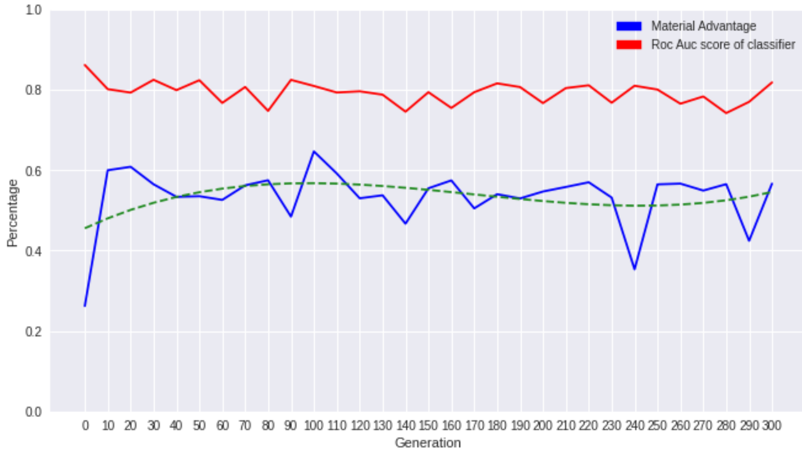

Figure 4.8: Percentage of selected states containing the HLC Material Advantage

The Figure 4.8. shows that throughout training the Material 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.

4.2.3 Agressiveness

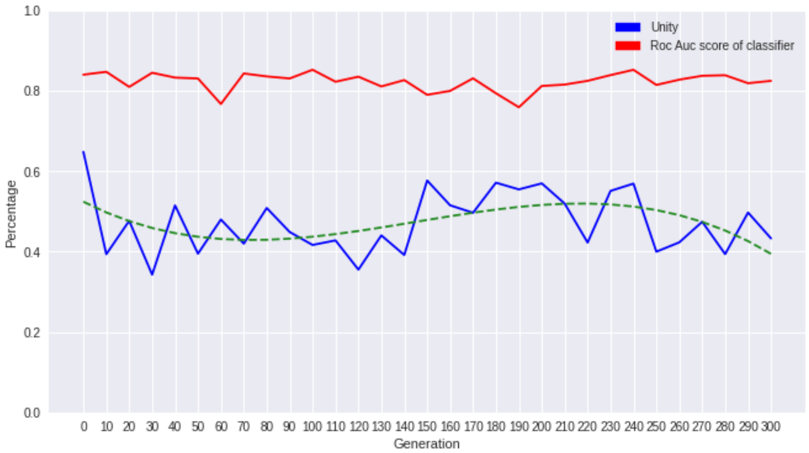

Figure 4.9: Percentage of selected states containing the HLC aggressiveness

From Figure 4.9, we can see that the aggressiveness HLC is a growing factor over as the neural network is trained. As a piece can always move forward, if your opponent does not capture that piece, this concept relates to how many moves it will take for you to win. Generally, when your opponent is not playing well this concept does well, and quickly becomes a bad heuristic in classical search algorithms.

4.2.4 Unity

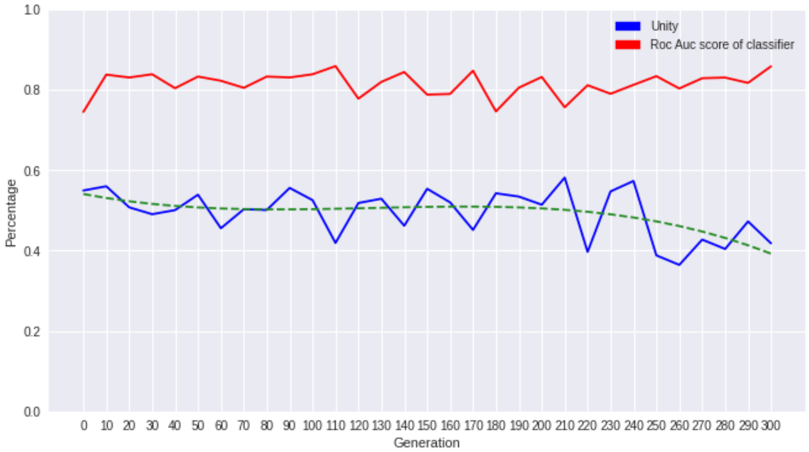

Figure 4.10: Percentage of selected states containing the HLC unity

Examining Figure 4.10, we see that the Unity HLC is steady, but then it drops as the network is trained. As this is considered a decent strategy in Breakthrough this is unexpected. However, this concept become increasingly unreliable as you lose pieces because it is the average of your remaining pieces distance to their center.

4.2.5 Lorentz-Horey Score

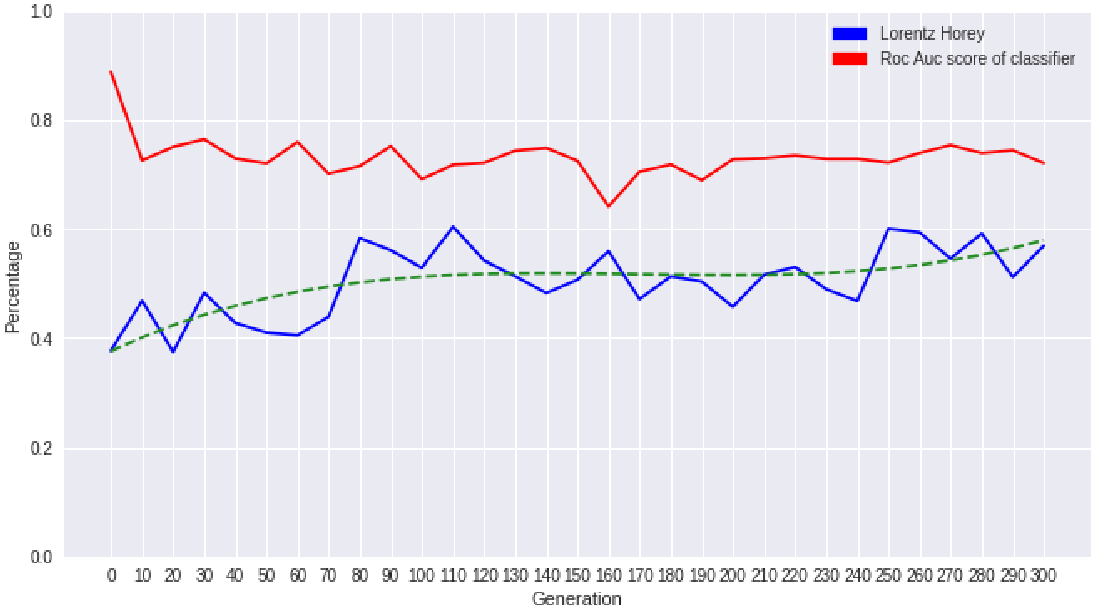

Figure 4.11: Percentage of selected states containing the HLC Lorentz-Horey

Examining Figure 4.11, we can see a trend where as the neural network is trained its use of this HLC increases over time. As this heuristic is the one most often cited as a successful strategy in Breakthrough, it is interesting to see that our Breakthrough agent is gradually learning a similar concept.

4.3 Summary

In this section we discussed some concepts that we believed would be representative of concepts that a nerual network would learn over time playing Breakthrough. An important note is that there are an infinite amount of concepts, and there could exist some that the neural network uses extensively that we overlooked. We examined the neural network against these concepts. In the following chapter we will discuss the next steps for this research.

Chapter 5

Discussion

This chapter 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, our trained agent does indeed quickly rise to using the higher-level concepts, thus answering our research question 1. The results are promising, as we see a popular concepts like Lorentz-Horey score rise in emphasis for the neural network, and a simple but effective concept like Material Advantage rise slowly but not exessively. We find the Unity concept to be disappointing as stated earlier in this paper; maintaining closeness of your pawns tends to be preferential. And lastly, for a the concept Aggressiveness, the fact that it falls off early is what we expected. It would have been the preferred result if we saw a greater variability in the concepts, but this could be a result from not enough training iterations, or not a complex enough model. Considering that the most emphazised HLC is the Lorentz-Horey score, and that the least is Unity, we can answer our research question 2. As we know from the literature, Lorentz-Horey is considered the leading heuristic for Breakthrough, and that HLC is highly emphazised by the neural network. Secondly, the Unity heuristic is not a validated heuristic, moreso, an attempt at creating a simple heuristic that closely resembles how tight-packed your pawns are. However, the fact that the resulting graphs are relatively steady the results are not as strong as we would have hoped.

Importantly, these concepts are only a few possibilities of an infinite set of concepts, and the neural network could be learning a completely different concept then those that we tested. However, that was not the focus of this research, it was to examine if a neural network does move towards HLC's that we humans use in games.

5.2 Future Work

To iterate on this research, training a neural network with greater computing power, or with a larger neural network architecture, will allow the us to hopefully achieve a super-human neural network in Breakthrough. This would give greater confidence in the learned concepts, possibly allowing for pedagogical research on the topic regarding which concepts are optimal to train people in the game environment.

The work done in this thesis was only a proof of concept of the process of explaining the internal concept a neural network uses while acting in a board game environment. Ideally, one would like to extend this work to a larger set of board games.

Additionally applying this method to a more diverse set of environments 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 gender bias.

5.3 Conclusion

The 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 it 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 dynamic systems with discrete human level concepts. Our results show that for an agent that clearly doesn't achieve human level performance its actions are often selected with respect to human level concepts.

Bibliography

[1] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, 2014. [Online]. Available: http://arxiv.org/abs/1409.1556.

[2] J. Brown, J. P. Campbell, A. Beers, K. Chang, S. Ostmo, R. P. Chan, J. Dy, D. Erdogmus, S. Ioannidis, J. Kalpathy-Cramer, and M. F. Chiang, "Automated diagnosis of plus disease in retinopathy of prematurity using deep convolutional neural networks", pp. 803–810, Jul. 2018, ISSN: 2168-6165. [Online]. Available: http://eprints.lincoln.ac.uk/35638/; http://doi.org/10.1001/jamaophthalmol.2018.1934.

[3] K. Kourou, T. P. Exarchos, K. P. Exarchos, M. V. Karamouzis, and D. I. Fotiadis, "Machine learning applications in cancer prognosis and prediction", en, 2014. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC; http://www.ncbi.nlm.nih.gov/pubmed/25750696; http://dx.doi.org/10.1016/j.csbj.2014.11.005.

[4] A. W. Senior, R. Evans, J. Jumper, et al ..., and D. Hassibis, "Protein structure prediction using multiple deep neural networks in the 13th critical assessment of protein structure prediction (casp13)", *WileyOnlineLibrary*, Oct. 2019.

[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] B. Goodman and S. R. Flaxman, "European union regulations on algorithmic decision-making and a right to explanation", *AI Magazine*, vol. 38, no. 3, pp. 50–57, 2017.

[7] M. T. Ribeiro, S. S. 0001, and C. Guestrin, "Model-agnostic interpretability of machine learning", *CoRR*, vol. abs/1606.05386, 2016. [Online]. Available: `http://arxiv.org/abs/1606.05386`.

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

[9] 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.

[10] T. P. Hart and D. J. Edwards, "The alpha-beta heuristic", Massachusettes Institute of Technology, Cambridge, MA, Tech. Rep. 30, Oct. 1963.

[11] D. Michulke and M. Thielscher, "Neural networks for state evaluation in general game playing", in *Machine Learning and Knowledge Discovery in Databases, European Conference, ECML PKDD 2009, Bled, Slovenia, September 7-11, 2009, Proceedings, Part II*, W. L. Buntine, M. Grobelnik, D. Mladenic, and J. Shawe-Taylor, Eds., ser. Lecture Notes in Computer Science, vol. 5782, Springer, 2009, pp. 95–110, ISBN: 978-3-642-04173-0.

[12] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search", in *Computers and Games*, H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, Eds., ser. Lecture Notes in Computer Science, vol. 4630, Springer, 2006, pp. 72–83, ISBN: 978-3-540-75537-1.

[13] L Kocsis and C. Szepesvari, *Bandit based monte-carlo planning*, Conference or Workshop Item; PeerReviewed, Sep. 2006. [Online]. Available: `http://eprints.pascal-network.org/archive/00006352/; http://www.sztaki.hu/~szcsaba/papers/ecml06.pdf`.

[14] C. D. Rosin, "Multi-armed bandits with episode context", *Ann. Math. Artif. Intell*, vol. 61, no. 3, pp. 203–230, 2011.

[15] C. J. C. H. Watkins, "Learning from delayed rewards", Ph.D. dissertation, King's College, 1989.

[16] R. S. Sutton, “Learning to predict by the methods of temporal differences”, *Machine Learning*, vol. 3, pp. 9–44, 1988.

[17] S. P. Lloyd, “Least squares quantization in PCM”, *IEEE Transactions on Information Theory*, vol. 28, pp. 128–137, 1982.

[18] R. F. Ling, “On the theory and construction of k -clusters”, *Comput. J*, vol. 15, no. 4, pp. 326–332, 1972.

[19] T. Karras, S. Laine, and T. Aila, “A style-based generator architecture for generative adversarial networks”, *CoRR*, vol. abs/1812.04948, 2018. [Online]. Available: `http://arxiv.org/abs/1812.04948`.

[20] D. Rumelhart, G. Hinton, and R. Williams, “Learning representations by back-propagating errors”, *Nature*, vol. 323, pp. 533–536, Oct. 1986.

[21] E. Tjoa and C. Guan, “A survey on explainable artificial intelligence (xai): Towards medical xai”, *CoRR*, vol. abs/1907.07374, 2019. [Online]. Available: `http://arxiv.org/abs/1907.07374`.

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

[23] 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`.

[24] 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/`.

[25] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines”, in *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, J. Fürnkranz and T. Joachims, Eds., Omnipress, 2010, pp. 807–814.

[26] D. Silver, et al ..., and D. Hassabis, “Mastering chess and shogi by self-play with a general reinforcement learning algorithm”, *arXiv*, Dec. 2017.

[27] S. Helgason, *Tcav-Breakthrough*, `https://github.com/sigurdurhelga/msc-chess`, 2020.