Designing an Artificial Neural Network for state evaluation in Arimaa
- Using a Convolutional Neural Network

Design av ett Artificialt Neuralt Nätverk för evaluering av tillstånd i Arimaa

Simon Keisala

Supervisor: Rita Kovordanyi
Examiner: Erik Berglund
Upphovsrätt


Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Simon Keisala
Abstract

Agents being able to play board games such as Tic Tac Toe, Chess, Go and Arimaa has been, and still is, a major difficulty in Artificial Intelligence. For the mentioned board games, there is a certain amount of legal moves a player can do in a specific board state. Tic Tac Toe have in average around 4-5 legal moves, with a total amount of 255168 possible games. Both Chess, Go and Arimaa have an increased amount of possible legal moves to do, and an almost infinite amount of possible games, making it impossible to have complete knowledge of the outcome.

This thesis work have created various Neural Networks, with the purpose of evaluating the likelihood of winning a game given a certain board state. An improved evaluation function would compensate for the inability of doing a deeper tree search in Arimaa, and the anticipation is to compete on equal skills against another well-performing agent (meijin) having one less search depth.

The results shows great potential. From a mere one hundred games against meijin, the network manages to separate good from bad positions, and after another one hundred games able to beat meijin with equal search depth.

It seems promising that by improving the training and by testing different sizes for the neural network that a neural network could win even with one less search depth. The huge branching factor of Arimaa makes such an improvement of the evaluation beneficial, even if the evaluation would be 10 000 times more slow.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>Contents</td>
<td>ii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>iii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>iv</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Aim</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Research questions</td>
<td>2</td>
</tr>
<tr>
<td>1.4 Delimitations</td>
<td>2</td>
</tr>
<tr>
<td>1.5 Structure of Report</td>
<td>2</td>
</tr>
<tr>
<td>2 Theory</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Machine Learning</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Learning Algorithm</td>
<td>4</td>
</tr>
<tr>
<td>2.3 Temporal Difference</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Minimax algorithm</td>
<td>11</td>
</tr>
<tr>
<td>2.5 Arimaa</td>
<td>13</td>
</tr>
<tr>
<td>2.6 Third party library</td>
<td>15</td>
</tr>
<tr>
<td>2.7 Related Work</td>
<td>16</td>
</tr>
<tr>
<td>3 Method</td>
<td>18</td>
</tr>
<tr>
<td>3.1 Playing matches</td>
<td>18</td>
</tr>
<tr>
<td>3.2 Neural Network Design</td>
<td>20</td>
</tr>
<tr>
<td>3.3 Training of the Neural Network</td>
<td>21</td>
</tr>
<tr>
<td>4 Results</td>
<td>24</td>
</tr>
<tr>
<td>5 Discussion</td>
<td>28</td>
</tr>
<tr>
<td>5.1 Results</td>
<td>28</td>
</tr>
<tr>
<td>5.2 Method</td>
<td>28</td>
</tr>
<tr>
<td>5.3 Future work</td>
<td>29</td>
</tr>
<tr>
<td>6 Conclusion</td>
<td>31</td>
</tr>
<tr>
<td>Bibliography</td>
<td>32</td>
</tr>
<tr>
<td>A Match against meijin</td>
<td>35</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>An armed bandit (Slot machine)</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Model of a neuron</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>A group of neurons fully connected in a layered neural network</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>Miniature example of a feature map. To the left is a two dimensional input, with the width 3 and height 3. In the middle is a 2 by 2 kernel matrix. To the right is the resulting feature map.</td>
<td>8</td>
</tr>
<tr>
<td>2.5</td>
<td>Illustration of a 3x3 kernel matrix being applied on a 7 by 7 matrix with the value of stride set to one.</td>
<td>9</td>
</tr>
<tr>
<td>2.6</td>
<td>Illustration of a 3x3 kernel matrix being applied on a 7 by 7 matrix with the value of stride set to two.</td>
<td>9</td>
</tr>
<tr>
<td>2.7</td>
<td>Illustration of a 3x3 kernel matrix being applied on a 7 by 7 matrix with the value of stride set to 1 and zero-padding (grayed regions) around it.</td>
<td>9</td>
</tr>
<tr>
<td>2.8</td>
<td>A typical configuration of a CNN. By: Aphex34 <a href="https://creativecommons.org/licenses/by-sa/4.0/deed.en">https://creativecommons.org/licenses/by-sa/4.0/deed.en</a></td>
<td>10</td>
</tr>
<tr>
<td>2.9</td>
<td>Minimax tree search with ply 3</td>
<td>12</td>
</tr>
<tr>
<td>2.10</td>
<td>An empty board used to play Arimaa</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>The changes in value of different positions from a game played between meijin and a neural network. The network uses AdaDelta as the optimizer.</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>The same network as figure 4.1, the value is relative to the value of the first position in the game (initial position).</td>
<td>25</td>
</tr>
<tr>
<td>4.3</td>
<td>The changes in value of different positions from a game played between meijin and a neural network. The network uses SGD as the optimizer, with learning rate 0.005.</td>
<td>26</td>
</tr>
<tr>
<td>4.4</td>
<td>The same network as figure 4.3, the value is relative to the value of the first position in the game (initial position).</td>
<td>26</td>
</tr>
<tr>
<td>4.5</td>
<td>The changes in value of different positions from a game played between meijin and a neural network. The network uses SGD as the optimizer, with learning rate 0.015.</td>
<td>27</td>
</tr>
<tr>
<td>4.6</td>
<td>The same network as figure 4.5, the value is relative to the value of the first position in the game (initial position).</td>
<td>27</td>
</tr>
</tbody>
</table>
List of Tables

2.1 List of activation functions and their corresponding formula ........................................ 6
3.1 Starting positions for both players. The four $x$ shows where the traps are positioned on the board. ................................................................. 18
3.2 Example of how the positions and scores may look like, the last position that silver do makes a rabbit get to goal, which make this the last visible position for gold. The scores for each position is not from an actual network ........................................ 22
3.3 Actions done to get the target scores for training. Left column is the input scores, right column is the targets from these inputs. $\lambda$ is 0 in the example, though for the actual training $\lambda$ is set to 0.64. ......................................................... 22
5.1 General Features – adaptation from Giraffe ................................................................. 30
5.2 Piece Features ............................................................................................................. 30
Artificial Intelligence have been of interest from the beginning of 1950s. At the time it was thought as having unlimited potential for intelligence, but it soon was realized that creating artificial intelligence was not as easy to do as they first imagined \[18\].

This chapter aims to briefly introduce the topic and project to the reader. The chapter will cover some background information of the project, along with a motivation and chosen approach.

1.1 Motivation

Self learning machines have been researched within from almost the beginning of the computer’s existence. Using strategic games, such as chess, is a very common way since these provide a clear structure of the rules. Researchers had, already at the end of 1950s, started looking at programs which improve from self practising. Arthur Samuel released a publication in 1959 where he describe an early stage implementation of a self learning machine \[19\]. This is one of the first steps towards machine learning, and Samuel briefly mention artificial neural networks (ANN), although at that time he called it *Neural-Net Approach*, as being a method which "should lead to the development of general-purpose learning machines".

One of the challenges with most board games is that the search space of these games are extremely large, and it would be impossible to do a full tree search of every action which can be made. If it would be possible to use a tree search algorithm to explore the entire search space it would also be possible to predict the best possible move to make in any given state. Games more complex than tic-tac-toe have (in most cases) a too large search space for this approach to be possible. This is why various evaluation functions are used to predict the likelihood of winning given certain states. The same tree search algorithm is instead used along with an evaluation function to do a best effort search of finding a winning terminal state.

By using an ANN, it may be possible to improve the evaluation of a state, and thus be able to give a better evaluation even when the search depth is not very deep. This thesis work intend to explore the possibilities of creating a generalized ANN implementation which should evaluate the board state and thus act as the evaluation function.
1.2 Aim

In this thesis work we aim to create a generalized module using an ANN for evaluating the various states of a game. The purpose is to see whether or not this module can provide with a better evaluation compared to the evaluation from hard-coded features and thus compensate the inability of doing a deep lookahead.

By studying what others have done in the field of ANN and artificial deep neural networks, and draw further conclusions of how it can be used for board games we hope that this module will evaluate a state better than the original evaluation function described later in chapter 3.

1.3 Research questions

The question we have raised is thus:

- **How should the input features be structured to suit the game Arimaa?**
  Arimaa have unique rules in comparison to chess. Structure in the input data to the ANN could have a large impact in the performance.

- **How could a Convolutional Neural Network be designed and trained for learning to play Arimaa?**
  A convolutional neural network can utilize a simple configuration of the input, and the design of the neural network is important.

1.4 Delimitations

The design of an ANN is almost unlimited, and to not delve too much into testing all possible implementations we limit the neural network to only take into consideration Convolutional Neural Network Architecture.

In this thesis work the main focus lies on designing the architecture, and prepare for further work within machine learning algorithms for game agents. A proof of concept implementation of a convolutional neural network will be tested against a given agent.

1.5 Structure of Report

The structure of this report is as following: *Chapter 2* will describe the background theory necessary to understand the work, and inform the reader about third party libraries used to generate the agent. *Chapter 3* will in detail cover the methods used for training the ANNs. *Chapter 4* describe the results achieved from the work. *Chapter 5* evaluates the results and method. The chapter also propose improvements and alternative implementations for future work. *Chapter 6* concludes the work done in this thesis work.
In this chapter we will introduce the basics required to understand the thesis work. The chapter also describes the basis of machine learning and the different disciplines within the field that the module will have to use.

2.1 Machine Learning

Within Artificial Intelligence, this work is focusing about machine learning, and how well it could work for solving the problem described in the introduction.

Machine learning can generally be divided into four different subcategories. These different subcategories within machine learning are; supervised learning, reinforcement learning, unsupervised learning and semi-supervised learning. The artificial neural network utilize one of these four styles, reinforcement learning.

Reinforcement Learning

Reinforcement learning uses rewards and punishments to teach an algorithm which actions that are good and bad respectively for different states. The original idea of reinforcement learning lies within psychology. Edward L. Thorndike once wrote:

> “Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond.” [22, p. 244]

The simplest example of reinforcement learning is the multi-armed bandit (n-armed bandit) problem, where Figure 2.1 shows a single armed bandit. The problem is formed to have ‘n’

\[\text{A state is a setup of input data.}\]
2.2 Learning Algorithm

Different levers to pull, each having an expected mean reward value. An agent randomly start pulling the levers, and depending on the reward received from each lever, pulling that specific lever become more or less likely to happen in the future. This kind of action-reward is in the same terms as Thorndike’s findings within animal intelligence.

Figure 2.1: An armed bandit (Slot machine)

In general, you could say that reinforcement learning consist of two different action types. One of the types is the greedy action. The aim of a greedy action is to exploit the current knowledge of the environment and make an action that provide the highest known reward. The second action which the agent can do is an exploratory action. The exploratory action is needed to discover new and possibly better rewards for the agent, which it could later exploit. The agent can do a random action to further explore that action’s reward – which in return increase the certainty of that action’s reward – or it can choose an unknown action, which have not yet been explored.

The ratio between greedy and exploratory actions varies depending on the problem which you try to solve. Most commonly used ratios are either \( \epsilon \)-greedy or tempered greedy action.

The algorithm will use reinforcement learning to play against itself or against the original algorithm to improve in performance (tutored games or self-learned games).

2.2 Learning Algorithm

An artificial neural network is a learning algorithm which consists of a network of nodes. Each node either connected to one or more other nodes, or is an output to the network with no outgoing connections. The ANN tries to solve optimization problems, such as speech or image recognition, for board games or even for self-driving car.

This section will cover both the general concept of ANN’s as well as a special structure of the neural network architecture called convolutional neural network.

Neural Network Architecture

The ANN contains three different parts; input, hidden. One way of designing an ANN is to use a static amount of input nodes, and connect nodes in layers, where each layer is fully or

\[ \text{The agent is the entity performing various tasks from what it perceives.} \]
sparsely connected to the next layer in the network. After the input layer there are one or more hidden layers in the network, followed finally by an output layer. Each layer, except for the input layer, is connected to a node-weight pair of each entity in the previous layer. Figure 2.2 shows a single node with the inputs and outputs.

![Diagram of a neuron](image)

**Figure 2.2: Model of a neuron**

The update of a single neuron is done in two steps, and is done to propagate the input through the network. The first step is to sum the product of each input with its corresponding weight. This gives us the formula: $in_j = \sum_{i=0}^{n} (o_i \cdot w_{i,j})$, where $j$ is the position of the neuron in the current layer, $o_i$ the input of position $i$ in the previous layer, and $w_{i,j}$ the weight for neuron $j$ of input $i$, $n$ is the number of neurons in the previous layer. Afterwards the sum is passed to an activation function $g(x)$ and set to the output value of neuron $j$. Section 2.2 describe six different activation functions and their respective properties.

A bias is usually required for the network to function under all circumstances, and its value cannot be set to 0. The purpose of the bias node is to provide some stimuli when all the inputs are set to 0. Without the bias node it would be impossible to generate any output other than 0 if all the inputs are set to 0.

Figure 2.3 shows how one of these artificial neural networks can be structured. The figure shows three layers. First layer is the input layer, this layer is representing the input in node form.

Second layer is the hidden layer and it receives information from the first input layer. In the illustration this hidden layer is fully connected to the previous layer (later referred as dense layer). Last layer is the output layer. The output from this layer is accessed outside of the neural network and is the result from the network.
There are various activation functions which are commonly used for neural networks. These have various advantages and disadvantages. Table 2.1 lists six different activation functions used for neural networks, although these are not the only activation functions which can be used.

<table>
<thead>
<tr>
<th>Name</th>
<th>activation function</th>
<th>output range</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear</td>
<td>$g(x) = x$</td>
<td>$[-\infty, \infty]$</td>
</tr>
<tr>
<td>ReLU</td>
<td>$g(x) = \max(x, 0)$</td>
<td>$[0, \infty]$</td>
</tr>
<tr>
<td>Leaky ReLU</td>
<td>$g(x) = \max(x, 0.01x)$</td>
<td>$[-\infty, \infty]$</td>
</tr>
<tr>
<td>softplus</td>
<td>$g(x) = \ln(1 + e^x)$</td>
<td>$[0, \infty]$</td>
</tr>
<tr>
<td>sigmoid</td>
<td>$g(x) = \frac{1}{1 + e^{-x}}$</td>
<td>$[0, 1]$</td>
</tr>
<tr>
<td>hyperbolic</td>
<td>$g(x) = \tanh(x)$</td>
<td>$[-1, 1]$</td>
</tr>
</tbody>
</table>

![Figure 2.3: A group of neurons fully connected in a layered neural network.](http://www.texample.net/tikz/examples/neural-network/)

**Activation Function**

The choice of which activation function to use, for not only the hidden layers, but also the output layer, could have a large impact of how fast the neural network learns its task.

The neural network can use a linear activation function for the output. However, for the hidden layers there is no value in using a linear activation function, since this would result in only being able to produce a linear output.

The softplus activation function have properties where large values on $x$ in the activation function yield values close to $x$ in the output, while large negative values results in a zero as output. This allow neurons to disable themselves when certain inputs are received by having a large negative weight to some input. The softplus function also does not limit the output to a maximum value. Rectified linear unit (ReLU) is very similar to softplus, except around zero, since it’s output is a combination of two linear functions.

The simple calculation of ReLU benefits this activation function over softplus when the neural network is very large, where million of calculations may be needed. ReLU activation function does however have a problem. In the initial state, when the weights haven’t stabilized, there is a high risk that many nodes in the network will die off – where the nodes no longer produce any output except for 0. According to Andrej Karpathy [11], up to 40 % of all the nodes could become inactive and unable to produce a non-zero output when using
ReLU activation function. The back-propagation algorithm fail to update these nodes since their derivative also become zero when their output is negative before applying the activation function.

Leaky ReLU use a similar activation function as ReLU activation function. Although when the output is negative, a small fraction of the output “leaks” through, which result in a small negative value, and a non-zero derivative. The nodes which would have been completely disabled when using ReLU activation function continue to update their weights during all circumstances by having a minor fraction of the output when its output is negative.

The sigmoid and hyperbolic activation function introduces non-linearity in the network, although they limit the output of each node within -1 and 1 (or within 0 and 1). Creating a deep neural network using sigmoid/hyperbolic activation function results in a diminishing gradient. The derivative of these two activation functions is approaching zero near the two extreme points, which is propagated back to each other layer. These two activation functions are suitable when classifying data, since their range is limited to two final values.

Back-propagation

In its core, the back-propagation algorithm is solving gradient descent problems. For a neural network this process is recursive. The goal of using the back-propagation algorithm is to modify all of the weights in the neural network in regards to their influence in the error between the actual output and the expected output.

The steps for the back-propagation algorithm is:

1. Propagate the input all the way through the network.
2. Find the error gradient of the output (target – output).
3. Find the error gradient of all the hidden layers recursively in reverse order.
4. Update the weights using the error gradient and learning rate.

The propagation of the error for the hidden layers does not have any target values in the same way as the output. Instead, the gradient is retrieved from calculating the sum of the error times the weight of each neuron in the next layer. Steven Miller have written a more in-depth description with step-by-step examples of how the back-propagation algorithm works [15].

On top of the back-propagation algorithm an optimizer – optimization function – is used together with the gradient to update the weights of the nodes. The optimizer have, in its simplest form, a parameter deciding the learning rate, which decide the step size in the descending gradient that the weights are update to. The Stochastic Gradient Descent optimization algorithm have an extra parameter for including a momentum of the update step.

For deep neural networks there is a problem with diminishing gradient for layers further away from the output layer. Research of alternative optimizers have been done, and one other optimizer is the AdaDelta algorithm by Zeiler [24]. AdaDelta use the gradient and update value for deciding the learning rate, and the learning rate is individual per node in the network.

The learning rate for AdaDelta uses the following formula:

\[ l_r = \frac{RMS[\Delta x]_{t-1}}{RMS[g]_t} \]  

\[ RMS[x]_t = \sqrt{E[x^2]_t} + \epsilon \]  

\[ E[x^2]_t = \rho E[x^2]_{t-1} + (1 - \rho) x_t^2 \]  

4 Deep neural networks have more than one hidden layer
2.2. Learning Algorithm

Where $g_t$ is the calculated error gradient for time $t$, $\Delta x$ the computed update by minimizing the gradient using the learning rate from function 2.1. $\rho$ is used to decay the stored gradient over time and $\epsilon$ is a small value to both start off the training and to ensure that the training will not die off.

Before the algorithm can be used the variables for accumulating the gradient and update rate have to be initialized; $E[g^2]_0 = E[\Delta x^2]_0 = 0$.

**Convolutional Neural Network Architecture**

A convolutional neural network is excellent for generating deep networks thanks to reduced computational requirements, although they have some limitations in the structure of their input.

The input for convolutional layers first have to specify how many dimensions which is expected for the input. Given a dimension, the input have to be ordered in a fixed size of the specified dimension. For images this dimension is two; an image contains a width and height.

Each convolutional layer contains several feature maps, which is a set of nodes with the same dimension as the input. Each feature map contains kernel matrices, each channel of the input layer for the feature map have its own kernel matrix. A colored image would have one channel for each color. Figure 2.4 shows a miniature example of a two dimensional convolutional feature map with one input channel.

<table>
<thead>
<tr>
<th>Input</th>
<th>Kernel Matrix</th>
<th>Feature Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.2</td>
<td>+0.3</td>
<td>-0.3</td>
</tr>
<tr>
<td>+0.6</td>
<td>-0.9</td>
<td>-0.1</td>
</tr>
<tr>
<td>+0.3</td>
<td>-0.9</td>
<td>-0.4</td>
</tr>
<tr>
<td>+0.9</td>
<td>+0.5</td>
<td>-0.72</td>
</tr>
<tr>
<td>-1.0</td>
<td>+0.1</td>
<td>+1.01</td>
</tr>
<tr>
<td>-0.3</td>
<td>+0.0</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 2.4: Miniature example of a feature map. To the left is a two dimensional input, with the width 3 and height 3. In the middle is a 2 by 2 kernel matrix. To the right is the resulting feature map.*

These kernel matrices have similar properties as filters used for shaders in computer graphics; a small region of the entire input is analyzed, and the kernel matrix is applied to the smaller region. The value of each node in the feature map is the resulting sum of each value in the kernel matrix times the value of the input matrix of the smaller region.

For convolutional layers the weights of these matrices are learned throughout the training to find unique relationships for the task it tries to solve. The convolutional layers have one additional setting, called “stride”. The value of the stride represents how many shifts in the input for the next sample.
Figure 2.5: Illustration of a 3x3 kernel matrix being applied on a 7 by 7 matrix with the value of stride set to one.

Figure 2.5 illustrate which region a 3 by 3 kernel matrix is applied and stored for three cells. In the illustration the stride value of the convolutional layer is set to one. Figure 2.6 shows its behaviour when the stride-value is instead set to two.

Figure 2.6: Illustration of a 3x3 kernel matrix being applied on a 7 by 7 matrix with the value of stride set to two.

From figure 2.5 it can be seen that the output dimension reduces from 7x7 to 5x5. In some cases it may be unwanted to have this reduction. To compensate for this reduction the input matrix can have a padding on each of its four sides, with zero-valued data. Figure 2.7 shows how such an input would be.

Figure 2.7: Illustration of a 3x3 kernel matrix being applied on a 7 by 7 matrix with the value of stride set to 1 and zero-padding (grayed regions) around it.
Directly after a convolutional layer it is common to add a pooling layer. The pooling layer exists to downsample the output from a convolutional layer by grouping smaller regions into a single output. The pooling is not necessary for a convolutional network, but for image analysis it could improve the performance since the downsampling helps in reducing noise which can exist in images.

![Figure 2.8: A typical configuration of a CNN. By: Aphex34](https://creativecommons.org/licenses/by-sa/4.0/deed.en)

Figure 2.8 shows a typical setup for a convolutional network. The figure consists of an input, followed by a convolutional layer with feature maps. Afterwards there is a layer for downsampling, or subsampling, of the convolutional layer. Then there is another pair of convolutional and downsampling. Finally a fully connected layer looks at this very abstract output from the last feature-map based layer.

### 2.3 Temporal Difference

The artificial neural network mentioned in chapter 2.2 need a way of evaluating and updating the weights for its connections. Using back-propagation for the error correcting is trivial, although finding what value for the back-propagation algorithm to correct towards is the main problem.

Various methods have been proven successful at this task, one of them being a variety of Temporal Difference (TD) learning called TD-Leaf. The TD-algorithm is an algorithm which use information received from the next or many following new states to update the knowledge of an old state \[21\]. The TD-algorithm will for each time step read the approximated value of the new state it arrives to, and adjust the value of the previous state to become closer to this new value.

By letting \( r_t \) be the reward for time \( t \) and \( \hat{V}_t \) be the correct prediction, the formula for the TD-algorithm is:

\[
\hat{V}_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i}
\]

(2.4)

\[
\hat{V}_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i}
\]

(2.5)

Where \( \gamma \) is the discount factor and \( 0 \leq \gamma < 1 \).

This formula can be further expanded
2.4 Minimax algorithm

The minimax algorithm is used in games where there are two opposing forces, and one player (force) tries to maximize a certain value, whether it being heuristic or pre-determined, and the other player (force) tries to minimize the value, whereas the name minimax. Figure 2.9 illustrates the decision that Max and Min do during their move respectively.

Formula 2.10 shows that the reward \( r_t \) is the difference between the correct prediction and the next prediction.

For a tree-search algorithm Baxter et al. [3] suggested an alternative to TD(\( \lambda \)) called TDLeaf(\( \lambda \)). In their paper they compared the training performance of two different alternatives to TD(\( \lambda \)): TD-directed(\( \lambda \)) and TDLeaf(\( \lambda \)), both using minimax search in different degrees for chess. The original TD(\( \lambda \)) algorithm did not include any tree search, and for chess it is “difficult to accurately evaluate a position by looking only one move or ply ahead” [3].

TD-directed(\( \lambda \)) in the paper use minimax tree search, which will be described in the next section, for evaluating and finding which play to make, but the learning algorithm does not use the value from the leaf node in the minimax search during its training. Instead the TD-directed(\( \lambda \)) evaluates the actual position which it encounters and applies TD(\( \lambda \)) on that position. On the contrary, TDLeaf(\( \lambda \)) would not only use minimax search to evaluate a move, but also match each position with its leaf node’s value, found during the minimax search, instead of its own actual value.

From the experiments done by Baxter et al. [3] they found that TDLeaf converges faster than TD-directed, though that both methods still are able to learn while playing. They also noted that when the AI learns from self-play the result is far worse than when playing against other opponents around the same skill level. This due to the fact that most self-play games are played very differently compared to games played by humans around the same skill level.

\[
\begin{align*}
\bar{V}_t &= r_t + \sum_{i=1}^{\infty} \gamma^i r_{t+i} \\
\bar{V}_t &= r_t + \sum_{i=0}^{\infty} \gamma^{i+1} r_{t+i+1} \\
\bar{V}_t &= r_t + \sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \\
\bar{V}_t &= r_t + \gamma \bar{V}_{t+1} \\
r_t &= \bar{V}_t - \gamma \bar{V}_{t+1}
\end{align*}
\]
2.4. Minimax algorithm

![Minimax tree search with ply 3](image)

**Figure 2.9:** Minimax tree search with ply 3.

Code 2.1 represents the pseudocode for the minimax algorithm. As can be seen from the code, the minimax algorithm is recursive, and does not end until the maximum search depth is reached or until the position is a terminal position. Both terminal positions and positions which have reached a maximum depth are then evaluated. These positions in the search tree are also called leaf nodes, since they do not branch to any other position.

**Code 2.1: Pseudocode for minimax algorithm**

```cpp
int minimax(position node, int depth_left, bool maximizingPlayer) {
    if (depth_left == 0 || terminalNode(node)) {
        return heuristicValue(node);
    }
    if (maximizingPlayer) {
        int bestValue = INT_MIN;
        for (position child : children(node)) {
            bestValue = max(bestValue, minimax(child, depth_left - 1, false));
        }
        return bestValue;
    } else {
        int bestValue = INT_MAX;
        for (position child : children(node)) {
            bestValue = min(bestValue, minimax(child, depth_left - 1, true));
        }
        return bestValue;
    }
}
```

An improvement of the minimax algorithm is the alpha-beta algorithm. The algorithm have the same basic principle as minimax, although by using two extra values, alpha and beta, large branches of the tree search can be cut off, since the nodes following the branch is worse than any other alternative for the player, and does no longer have to be searched. Code 2.2 shows the changes made for having an alpha-beta cutoff.

**Code 2.2: Pseudocode for alphabeta algorithm**

```cpp
int alphabeta(position node, int depth_left, int alpha, int beta, bool maximizingPlayer) {
    if (depth_left == 0 || terminalNode(node)) {
        return heuristicValue(node);
    }
    if (maximizingPlayer) {
        int bestValue = INT_MIN;
        for (position child : children(node)) {
            bestValue = max(bestValue, alphabeta(child, depth_left - 1, alpha, beta, false));
        }
        return bestValue;
    } else {
        int bestValue = INT_MAX;
        for (position child : children(node)) {
            bestValue = min(bestValue, alphabeta(child, depth_left - 1, alpha, beta, true));
        }
        return bestValue;
    }
}
```

For board games the search depth is typically defined as number of plies (singular ply).
2.5 Arimaa

Arimaa is a board game specifically designed of being difficult for an AI-agent to solve. The game have very simple rules, but these rules results in a branching factor far greater than both Chess and Go \[^1\]. The mean number of legal unique moves available given a state is around 16000. In comparison, chess is being said to have around 35 legal moves for each state.

The result of this extremely large branching factor is that a minimax algorithm cannot search very deep and thus cannot see the final consequence of making an action until it is too late.

Arimaa is a board game being played on a 8x8 board, same as chess, checker, Othello and some other board games. Figure [2.10] shows an empty board for Arimaa.
Arimaa rules

The rules for Arimaa are very simple. A player have 16 pieces, all ranging from Elephant (strongest) to Rabbit (weakest). The player can arrange these pieces into any order on the first two rows on his or her side.

The number of pieces of each type is the following:

- 1x Elephant (E/e) – Strongest
- 1x Camel (M/m)
- 2x Horse (H/h)
- 2x Dog (D/d)
- 2x Cat (C/c)
- 8x Rabbit (R/r) – Weakest

With these piece types and amounts, and with the freedom of arranging the pieces in any combination, it results in $64,864,800$ opening configurations, for each player, as seen in equation (2.11).

$$\binom{16}{8} \times \binom{8}{2} \times \binom{6}{2} \times \binom{4}{2} \times \binom{2}{1} \times \binom{1}{1} = 64,864,800$$  \hspace{1cm} (2.11)

\textsuperscript{6} Short notation for each piece for gold/silver player in parenthesis
A player have three different actions which he or she can do during his or her turn. These actions are moving pushing or pulling. Each different action costs action points to do, and for each turn a total amount of 4 action points exists.

Because a turn consists of separate actions, and the player can distribute them amongst several pieces, the branching factor becomes humongous compared to other existing board games. This is why Arimaa is such a complex board game for an artificial intelligence to master.

The actions available, with their corresponding cost and description, are:

**1AP Move**
Move a piece one step forward, backward (except for rabbit), left or right.

**2AP Push**
Move a nearby weaker opponent piece to a free nearby location and move yourself to the opponent’s piece location.

**2AP Pull**
Move a nearby weaker opponent piece to your own location, move yourself to an empty location (excluding the location of pulled opponent piece).

A player wins the game when one of the players rabbits have reached the other player’s side or when all the opponents rabbits have been captured.

**Special Rules**
Arimaa has two special rules. These are *frozen pieces* and *captured pieces*.

A piece is frozen when a stronger piece is next to it and another friendly piece is not next to it. A frozen piece cannot move, push or pull, but it can still be pushed or pulled by the opponent.

A piece is captured if the piece is standing on one of the four traps (located at c3, c6, f3 and f6) and there is no friendly piece next to it.

### 2.6 Third party library

To avoid the problem of having a non-working neural network, and to avoid several problems when designing the neural network, we will make use of a framework. Below we will give a brief introduction regarding the relevant framework that is used in our project.

**Keras Framework**

One very highly abstracted framework for artificial neural networks, and with good support of deep artificial neural networks, is the python framework Keras, developed by Chollet [7]. The framework supports two different backend implementations of neural networks, TensorFlow developed by Google team [14] and Theano made by LISA / MILA Lab at Université de Montréal [17].

The framework have from the beginning of 2017 received further support from google thanks to its simplicity and will formally be integrated into TensorFlow [8].

**Meijin**

As a basis of the AI, an agent with the core features of performing various tasks of Arimaa is given. Meijin has been developed by Inge Wallin [9], and provides with alpha-beta pruning, move-generation and decision making of which move to make using an evaluation function. On top of this, the agent use an evaluation function to determine which move to execute.
The evaluation function is easily replaced using an input consisting of a certain board state. Meijin expects to receive an integer from the evaluation function. The integer is positive when the state have a high likelihood of leading to a win, and negative when the state most likely will lead to a loss.

2.7 Related Work

In recent years there have been plenty of successful research within machine learning and artificial neural networks. Two related works from recent years are AlphaGo, created by Google DeepMind, and an attempt of making a well playing Arimaa agent by Haizhi Zhong.

Another recent work is the work on Giraffe by Matthew Lai. Giraffe is an agent using neural networks for playing chess.

**AlphaGo - Google DeepMind Deep Neural Network Agent**

AlphaGo is the first Go playing program ever winning a fair match against a professional human Go player. This is a breakthrough within artificial intelligence, and was not expected to happen for another 10 years \[6\].

Unlike Chess, where features easily can be made which evaluate how good a certain board state is for a player, the evaluation of a state in Go is a lot more complex. This complexity in evaluating a board state in Go make it impossible to generate good features to predict the strength of a state. AlphaGo use two deep artificial neural networks for its evaluation. One network is in charge of predicting where the next played position will be. The other network estimates the likelihood of winning in a certain state. Except for these neural networks the program use Monte Carlo Tree Search to do a deeper analysis of moves \[20\].

The Neural Network design and training approach used for Google DeepMind AlphaGo serves as a good source of inspiration.

**Building a Strong Arimaa-playing Program**

In 2005, only two years after Arimaa was invented, Haizhi Zhong made an attempt of building a strong program for playing Arimaa \[25\]. The strength of a player (program or human) is measured using a similar system compared with Elo \[7\]. The strength starts at 1400, and for each game the score will go either up or down, depending on whether the player won or lost, and depending on the strength of the opponent the player was facing. In Haizhi Zhong’s attempt of making a strong program he briefly investigates the usage of Reinforcement Learning for tuning the weights of each feature analysed in his thesis work.

Unfortunately Zhong was unsuccessful in his implementation of his temporal difference (TD(\(\lambda\))) learning. He assume that the reason of their failure might be either that the code had some bug that they were unable to find, that the algorithm did not have enough time to find all necessary features to learn how to play well or that they could not do a deep enough lookahead for the learning algorithm to completely understand why a move is good or not.

**Giraffe: Using Deep Reinforcement Learning to Play Chess**

In 2015, Matthew Lai worked on implementing a deep artificial neural network for playing chess. His work received positive results from the use of a neural network \[13\]. The work from Lai shows another approach of using neural networks. Many aspects covered from his work could be utilized in the implementation of a dense neural network for an Arimaa-playing agent.

Matthew Lai evaluated the performance of the algorithm using a Strategic Test Suite. This test suite is used as an indicator of the strength for chess agents. The results received by his

---

7 Elo is a ranking system which is used for estimating the skill level of a player in a game.
agent was compared against other agents running the same test. The results showed that the agent successfully could learn various strategic problems that may occur in chess, along with other points of interest. Some interesting points that he found is the varying bishop and knight value in different situations, different openings and the value of centre control.

Although Lai did not use convolutional neural networks for chess, the analysis of feature representation for chess serves as an interesting point for dense neural networks. A different feature representation could have a huge impact on the training of a neural network, and Lai made use of a different design of the input in his work.
This chapter will be separated into three parts. First the method for executing moves and playing matches is described, secondly the design of the neural network which was tested. Finally this chapter will describe the entire process for training the network.

3.1 Playing matches

The method for playing matches is done in two different steps. First action done for each match is to initiate all the pieces.

As mentioned in section 2.5, each player is able to place the pieces in any possible order. Although for the matches against meijin, these positions have been fixed. Table 3.1 shows the starting position for each player.

<table>
<thead>
<tr>
<th>8</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>d</th>
<th>d</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>r</td>
<td>h</td>
<td>c</td>
<td>e</td>
<td>m</td>
<td>c</td>
<td>h</td>
</tr>
<tr>
<td>6</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>R</td>
<td>H</td>
<td>C</td>
<td>M</td>
<td>E</td>
<td>C</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>D</td>
<td>D</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
</tr>
</tbody>
</table>

Table 3.1: Starting positions for both players. The four x shows where the traps are positioned on the board

After the two setup moves (one from each player) the two players use tree search for finding a move to execute, this process is divided into three parts:

1. Move generator
2. Tree search (Alphabeta algorithm)
3. Evaluator
3.1. Playing matches

The move generator receives a position describing where all the pieces are located on the board along with which player that should perform the next move. The resulting value is a list of all moves that can be performed for that player. On top of generating all moves, the move generator also remove duplicates. Since a move in Arimaa consists of four separate actions, there are many moves which end up in the same position. Moving the elephant one step to the north and then to the east puts it in the same position as moving to the east followed by north. Similarly, moving camel south and then elephant south is the same as moving elephant south followed by camel south. All these duplicates which result in the same position are removed.

The tree search algorithm use this list to create a tree structure, where the list of moves from the move generator are the nodes in the tree. The tree search is a recursive function, which continues until the given search depth (number of plies), or a terminating position, is reached.

The evaluator evaluates a position. From the original position and the list of moves it is possible to see how the board looks after a fixed search depth. For the evaluator the task is to give each position a value. The value is used to choose which move to finally apply.

Using these three parts a move can be generated, evaluated and executed, and the tree search allows for deeper search before a move finally is applied.

Opponent

Arimaa is a two player game, and for the agent to be able to play, there must also be an opponent for it to play against. Below three different choices for the agents opponent is listed, and the opponent could be either one of them, with its own advantages and disadvantages.

1. Opponent is the same agent (self-play)
   **Pros:** There is no need for any other agent and the opponent is always on equal level.
   **Cons:** Games often become very similar, which reduces the variety between each match. The end result could in this case be that many positions remains unexplored for the agent.

2. Opponent is a human player
   **Pros:** Human high ranked players make actions with in-depth thoughts. These actions are shown to the agent in early stages of the training, and the agents are receiving plenty of variety in matches against human players.
   **Cons:** Human actions are slower than that from an agent, as well as not being available 24 hours a day.

3. Opponent is another agent (tutored-play)
   **Pros:** Stronger agents can serve as tutors for the learning agent, while still partially have an in-depth reasoning for their actions.
   **Cons:** Other agents have strict rules that are followed. If only a single agent is used as a tutor there are risks that the neural network gets specialized to counter that specific opponent’s actions.

When Baxter et al.\(^3\) did their experiments they noticed that an agent training against human players perform a lot better (89 out of 100 wins) against an agent learning from self-play, even when the self-playing agent was allowed to play twice as many games. Since meijin originally have an evaluator for positions there is already another agent available able to act as the tutor for the neural network.
3.2 Neural Network Design

The main problem when designing a neural network is to structure the input in a suitable form which contains all information of the board state.

For the convolutional network the input data have to be ordered with neighbouring regions having some connection between themself. For Arimaa this correlation can be the 8 by 8 grid of the board. Knowing this, there are two reasonable ways to structure the input data for Arimaa.

- **Raw data**
  Each position in the 8 by 8 grid stores the strength of the piece on that position; 0 for empty slot, 1 for rabbit, 2 for cat, ..., 5 for camel and 6 for elephant. There are two of these matrices in total, one matrix containing all the pieces for gold, and the other all the pieces for silver.

- **One-hot matrices**
  The above raw data matrices are separated into one matrix per piece type, resulting in six matrices per side. For each x-y-pair, only one of the 12 matrices can have its value set to 1, since Arimaa does not allow two pieces to be on the same square.

For a convolutional neural network the one-hot choice for input provide the most clear data, where the kernel feature receive an overview of surrounding regions, and each separate piece type is separated into its own channel, this approach would in theory suit the network well. The total number of inputs for the network using the one-hot matrix is $8 \times 8 \times 6 \times 2 = 768$.

Another part of information which is missing with these 768 inputs is who’s turn it is to make a move. The moving player have in most cases the advantage, and it is necessary to also consider this information when evaluating a position.

Instead of providing an input showing who’s move it is, the input will instead be configured to always assume that gold is the moving player. If the agent is playing as silver, the board and all its pieces are flipped along the x-axis, for example golden elephant on a5 (notated as Ea5), is changed to become silver elephant on a4 (notated as ea4). This is done for all the pieces on the board. In this way, all the positions are learnt from gold’s perspective, and the network will naturally learn to evaluate a board knowing that the next move is gold’s move.

Between each layer there will not be any down-sampling since the dimensionality of the input is already quite small, 8x8, and the patterns found from the feature maps may have to be very precise in their position. Down-sampling the output between each convolutional layer may worsen the performance for this reason. Unlike traditional convolutional networks, the convolutional layers used for evaluating Arimaa positions does not reduce in size.

As mentioned in section 2.2 there are a few different choices for the activation function. The one activation function which has proven successful in recent research is the ReLU activation function $ReLU$. The activation function provides a non-linearity while still being cheap to calculate – a single comparison if less than zero. In this implementation the Leaky ReLU will be used as the activation function for the hidden layers, which, in comparison to ReLU requires one extra multiplication in the case where the output is negative. The output layer uses tanh activation function to limit the output within -1.0 and 1.0.

Between each hidden layer there must be some non-linearity, otherwise two linear operations could be simplified into a single linear operation. The Leaky ReLU fulfill these requirements while not being a computationally heavy operation.

Each hidden convolutional layer use zero-padding to keep the same dimension as the input, and since the calculation for a convolutional layer is very cheap compared to that of a fully connected layer this padding is cheap to calculate.

1. In very few cases the best move is not to move, in these cases it is said that the player is in zugzwang.
2. Max-pooling is a method frequently used for image recognition to group together smaller sections into a single neuron.
connected layer\(^3\), these earlier convolutional layers are able to be a lot larger than the last dense layer.

For this thesis work the following design of the neural network was tested for Arimaa:

1. Input consisting of 12 channels of 8 by 8 matrices

2. Convolutional layer with 256 feature maps, 3 by 3 kernel matrices, stride 1 and one layer zero-padding.
   \[
   \text{Input: } 8*8*12 \\
   \text{Output: } 8*8*256
   \]

3. Convolutional layer with 32 feature maps, 3 by 3 kernel matrices, stride 1 and one layer zero-padding.
   \[
   \text{Input: } 8*8*256 \\
   \text{Output: } 8*8*32
   \]

4. Dense layer with 512 neurons.
   \[
   \text{Input: } 8*8*32 \text{ (flattened into one channel)} \\
   \text{Output: } 512
   \]

5. Single neuron for output, fully connected to the 512 neurons on previous layer.

### 3.3 Training of the Neural Network

The training of the network is done after each match have finished. The API which handle the communication between the two agents saves the actions and result of each match. An example of the output looks like this:

```
1g Ra1 Rb1 Rc1 Dd1 De1 Rf1 Rg1 Rh1 Ra2 Hb2 Cc2 Md2 Ec2 Cf2 Hg2 Rh2
1s ra7 hb7 cc7 ed7 me7 cf7 hg7 rh7 ra8 rb8 re8 de8 dd8 rf8 rg8 rh8
2g Ec2n Md2n Md3n Hg2n
2s ed7s ed6s ed5e Md4n
3g Hb2n Hb3n Hb4n Hb5n
3s me7s me6w md6w hg7s
...
6s mc5n Hc4n mc6e Hc5n Hc6x
7g Ra2n Ra3n Ra4n Ra5n
...
16g re5e Ee6s Ec5w Dd4w
16s ed3s ed2n Ce2e rb3s
17g Rg2e De1n Cd2w De2n
17s ed3s ed2s Cv2e rb2s
0-1
```

The first two lines (1g and 1s) are the two initial moves for the agents, where first character shows which piece type it is, and the other two shows the position on the board where it is placed. Every other move starting from 2g are the moves each player makes. Each action in a move have four characters.

The last character on these other moves shows which direction the piece is moving: north (n), south (s), east (e) or west (w).

Except for showing which direction a piece moves, the last character can also be \(x\), which indicate that a piece standing on a trap got killed, and they do not cost any action point.

---

\(^3\) \(8*8*12*9 = 6912\) multiplications for 64 outputs from a convolutional feature map Vs. \(8*8*12*64 = 49152\) calculations for 64 outputs from a fully connected layer
The last line of a match states the winner. If the winner is gold, the last line is 1-0; if however silver won, the line is instead 0-1.

With this information it is possible to recreate how each position looked before each player made a move, and the individual positions are created from knowing the initial position and the move history. These positions are then separated to two lists, one containing all the positions before player gold made a move, the other before silver made a move.

Final result from this is two lists containing all positions before each player made a move, and a score for that position. Table 3.2 shows an example of these lists.

<table>
<thead>
<tr>
<th>Gold Positions</th>
<th>gold pos. score</th>
<th>Silver Positions</th>
<th>silver pos. score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ec2 Md2 ...me7</td>
<td>0.05</td>
<td>Ec3 Md4 ...me7</td>
<td>-0.01</td>
</tr>
<tr>
<td>...</td>
<td>0.07</td>
<td>...</td>
<td>0.01</td>
</tr>
<tr>
<td>...</td>
<td>0.00</td>
<td>...</td>
<td>0.04</td>
</tr>
<tr>
<td>...</td>
<td>-0.10</td>
<td>...</td>
<td>0.08</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>-0.43</td>
<td>Ed5 Cc2 ...ed3 rb2</td>
<td>0.38</td>
</tr>
<tr>
<td>...</td>
<td>-0.48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ed5 Cd2 ...ed1 rb1</td>
<td>-0.58</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Example of how the positions and scores may look like, the last position that silver do makes a rabbit get to goal, which make this the last visible position for gold. The scores for each position is not from an actual network.

After these lists have been generated the target score for both gold and silver’s positions are generated. Table 3.3 shows how the target scores are generated. First a +1.00 or -1.00 is appended to the list, (win/loss score), this is the static reward for the final position. Next, the entire list is passed through a TD(\(\lambda\)) function to slightly alter all of the scores. This will have a high influence of positions near the end of the game, and a smaller influence of positions during earlier moves. The extra +1.00 or -1.00 is then removed to keep the number of values same as the number of positions (states). This last slightly altered list of scores is then considered to be the target scores for all positions in the list, and the error is the difference between the actual score and the target scores.

<table>
<thead>
<tr>
<th>gold pos. score</th>
<th>win/loss score</th>
<th>TD((\lambda))</th>
<th>win/loss removed</th>
<th>targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>0.05</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>0.07</td>
<td>0.07</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>-0.10</td>
<td>-0.10</td>
<td>-0.10</td>
</tr>
<tr>
<td>-0.10</td>
<td>-0.10</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>-0.43</td>
<td>-0.43</td>
<td>-0.43</td>
</tr>
<tr>
<td>-0.43</td>
<td>-0.43</td>
<td>-0.48</td>
<td>-0.48</td>
<td>-0.48</td>
</tr>
<tr>
<td>-0.48</td>
<td>-0.48</td>
<td>-0.58</td>
<td>-0.58</td>
<td>-0.58</td>
</tr>
<tr>
<td>-0.58</td>
<td>-0.58</td>
<td>-1.00</td>
<td>-1.00</td>
<td>-1.00</td>
</tr>
<tr>
<td>-1.00</td>
<td>-1.00</td>
<td>-1.00</td>
<td>-1.00</td>
<td>-1.00</td>
</tr>
</tbody>
</table>

Table 3.3: Actions done to get the target scores for training. Left column is the input scores, right column is the targets from these inputs. \(\lambda\) is 0 in the example, though for the actual training \(\lambda\) is set to 0.64.

To further increase the training, and to make the network more accurate, all the positions for gold and silver are duplicated, and mirrored along the y-axis.
Consider the two positions:

<table>
<thead>
<tr>
<th>8</th>
<th>r</th>
<th>r</th>
<th>d</th>
<th>d</th>
<th>r</th>
<th>r</th>
<th>8</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>d</th>
<th>d</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>r</td>
<td>r</td>
<td>m</td>
<td>h</td>
<td>7</td>
<td>h</td>
<td>m</td>
<td>r</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>h</td>
<td>c</td>
<td>e</td>
<td>c</td>
<td>6</td>
<td>c</td>
<td>e</td>
<td>c</td>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>r</td>
<td>5</td>
<td>r</td>
<td>E</td>
<td>5</td>
<td>r</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>M</td>
<td>4</td>
<td>M</td>
<td>4</td>
<td>M</td>
<td>4</td>
<td>M</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>R</td>
<td>H</td>
<td>H</td>
<td>R</td>
<td>3</td>
<td>R</td>
<td>H</td>
<td>H</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>C</td>
<td>2</td>
<td>C</td>
<td>C</td>
<td>2</td>
<td>C</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>D</td>
<td>D</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
</tr>
</tbody>
</table>

Except for the right one being mirrored along the y-axis, everything is identical. It would be wise for the network to also evaluate these positions the same, and by adding the mirrored position to the training data it is provided with this symmetry. The positions are then trained using back-propagation with smaller batches of random positions from the last game. Positions from both sides are used for the training, and the steps in table 3.3 is also done for each position for silver.

The neural network continuously play games against meijin, switching between being gold and silver player. After each game it trains on the latest match using the method described in this chapter.

On top of training against the latest match, the network also evaluated the 10 matches played before the latest one. These matches were added to the training data, although the target values were not passed through the steps seen in table 3.3. Instead the evaluated score was used as the target.

The idea of adding extra historical data is to make the network not only learn how to evaluate new positions and states which it recently saw, but to try and maintain a similar evaluation towards old positions from previous games.
Chapter 3 described both how the input for a convolutional neural network could be structured. The chapter also described how to provide information about the moving player without forcing the network to learn each position depending on the moving player.

For gaining a proof-of-concept to see whether a convolutional network is in fact able to evaluate complex tasks, five neural networks was put to work. Each neural network was initialized with random weights, and the design match the structure from chapter 3. One network uses AdaDelta for the training, and the other four use SGD with momentum.

For the networks using SGD for their optimizing algorithm the momentum is fixed to the value 0.84, and the learning rate is set to 0.005, 0.01, 0.015 and 0.02.

Each network played tutored matches against meijin, and the configuration was set for meijin to have a search depth of two plies, while the neural networks were only allowed one ply. With a search depth of one less than meijin, the network would have to learn a deep enough evaluation to also take into consideration how the opponent can counter a move, without actually being evaluate all the counter-moves from the opponent.

Unfortunately the network never managed to win with one less ply, although when applying the already trained network against meijin with both agents search depth set to 1, the network was able to win a few times.

Appendix A shows one match where a neural network managed to win against meijin when both agent’s search depth was set to 1. In total 29 matches have been played using the same network. The network managed to win 14 of these matches, 8 times playing as first player (gold), and 6 times as the second player (silver). This confirms that the network have gained knowledge in playing Arimaa with similar strength as meijin.

After each match the network performed same methods as described in section 3.3, which in turn made every match against meijin unique, since the evaluation of the network was slightly changed between the matches.

From the game in appendix A each position before every twentieth move for both gold and silver was observed and evaluated by every intermediate version of three different networks.

\[^1\text{An intermediate version of a network contains the weights after training on the played match}\]
Figure 4.1: The changes in value of different positions from a game played between meijin and a neural network. The network uses AdaDelta as the optimizer.

Figure 4.2: The same network as figure 4.1, the value is relative to the value of the first position in the game (initial position).

Figure 4.1 and figure 4.2 shows the value which a network trained using AdaDelta gave the position before gold and silver made their 20'th, 40'th, and 60'th move respectively, and the score for the initial positions. From figure 4.1 it can be seen that a change in evaluation of one move affects the evaluation of different moves. This is due to the fact that the update for evaluating one score affects the weights for every different position. Using the relative value with the initial position as a basis shows more accurately the different evaluation of each position.

The same can be noted from evaluating the other networks on the same positions.
Figure 4.3: The changes in value of different positions from a game played between meijin and a neural network. The network uses SGD as the optimizer, with learning rate 0.005.

Figure 4.4: The same network as figure 4.3, the value is relative to the value of the first position in the game (initial position).

Figure 4.3 and 4.4 is the score from a network using SGD with a very low learning rate of 0.005. After around 150 iterations the network begins to become able to separate the winning side from the losing side, with later positions becoming more and more clear of the winner.
Figure 4.5: The changes in value of different positions from a game played between meijin and a neural network. The network uses SGD as the optimizer, with learning rate 0.015.

Figure 4.6: The same network as figure 4.5, the value is relative to the value of the first position in the game (initial position).

Figure 4.5 and 4.6 shows the score of the network playing the match which is evaluated. The network learned to separate tgood positions from bad early on in the training. From figure 4.6 it can be seen that after a mere 100 iterations of training it was able to completely separate the losing (silver’s) positions from the winning (gold’s) positions.
This chapter will take a look at the results achieved by using the proposed method. The method itself will once again be considered, and evaluated.

5.1 Results

From the result this method of training a neural network looks promising. After only about one hundred iterations against meijin the networks were able to separate good positions from bad positions. After another one hundred iterations the same network was able to compete on equal conditions against another well performing agent, meijin, which have an Elo around 1800 – ranking meijin one level below experts \[1, 2, 3\].

Something that was similar for all the networks was that the score for the initial position became more and more negative when using the network to evaluate the positions. The exact cause for this is unclear, and it would be preferable for the initial position to be evaluated closer to zero.

5.2 Method

The training process when only looking at the match history from the agents own matches made the process of learning very slow. There are already several game archives containing hundreds of pro-level games which could be used instead of using historical data from games played by the neural network agent itself.

There are still plenty of modifications and improvements for speeding up or enhance the training, and many parameters to fine-tune for both the design of the neural network and for the optimizer used during training. The thesis work have analysed how one design of the neural network would perform for Arimaa, although with the large amount of parameters to tune the work only grasped on the surface.

Further work on optimizing the network and training parameters could in the end out-perform the evaluation of traditionally handmade features used for evaluating board game positions. The largest benefit is that features are discovered by the algorithm itself, and only a terminating score have to be known.

The choice of using historical data from several recently played games may have influenced the learning for the neural network. Considering that the optimizers use internal states for
momentum and learning rate, this extra input could potentially ruin the training by counteract the benefits which these optimizers should provide.

Another change which could improve the performance is the structure of the input to the convolutional neural network. Instead of having one channel for each piece type, a more accurate design would be to rank each piece depending on how many pieces the opponent have which are stronger than itself. In Arimaa, since the movements are uniform among all pieces, the value depends more on how many pieces the opponent have which are stronger than oneself is more important than the actual piece type.

\section*{5.3 Future work}

The training of neural networks is, according to earlier studies, beneficial from pre-training of the weights \cite{1,2,3}, and there exist some game archives containing plenty of games played by high-ranked players. By allowing the network to analyse these matches and using the same method for training the networks could improve a lot faster and a lot more stable, since these matches avoid sacrificial moves. Since the proposed method only analyse matches where the network have played itself using trial and error, the “bad” decisions take extra time to remove.

\textbf{Dense neural network design}

Another alternative instead of using convolutional layers is to have a fully connected (dense) neural network. The feed forward neural network could use the same structure as the convolutional neural network for its input (one-hot matrices). Although the dense layers would not benefit from this input compared to convolutional layers.

Instead of using one-hot matrices, Matthew Lai proposed to feed the network with another structure of information.

The proposed input from Lai is to provide a set of inputs describing the general features, such as the material information – seen on table 5.1, and another set of features describing each piece of both players, as described on table 5.2.

To keep the order the same all the time, when having multiple of the same piece, the one with the lowest value in x-axis will be sorted first, and if two pieces share the same line, the lowest value in y-axis is the first one. By using the ReLU activation function it could be possible to position dead pieces at [-10, -10]. In this way the network could learn different conditions which does not apply when certain pieces are dead, by having this strong negative value effectively shutting down neurons.
5.3. Future work

Table 5.1: General Features – adaptation from Giraffe.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gold Elephants</td>
<td>1</td>
</tr>
<tr>
<td>Gold Camels</td>
<td>1</td>
</tr>
<tr>
<td>Gold Horses</td>
<td>2</td>
</tr>
<tr>
<td>Gold Dogs</td>
<td>1</td>
</tr>
<tr>
<td>Gold Cats</td>
<td>2</td>
</tr>
<tr>
<td>Gold Rabbits</td>
<td>5</td>
</tr>
<tr>
<td>Silver Elephants</td>
<td>1</td>
</tr>
<tr>
<td>Silver Camels</td>
<td>1</td>
</tr>
<tr>
<td>Silver Rabbits</td>
<td>4</td>
</tr>
</tbody>
</table>

Unlike chess, each piece in Arimaa share the same set of rules for moving themselves or push and pull opponent’s pieces. Thus there is no need for any features describing defending squares nor attacking squares.

The features listed on table 5.1 and 5.2 are concatenated together, and used in a fully connected layered neural networks. Similar to the convolutional neural network, this design does not have any input stating which player that will do the next move. If it would be silver’s move, the board is instead flipped.

Using only dense layers in the neural network and structure the input with the above configuration would be another implementation for the network, and testing these networks would be an interesting approach for Arimaa.

Table 5.2: Piece Features.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gold Elephant</td>
<td></td>
</tr>
<tr>
<td>x_position</td>
<td>3</td>
</tr>
<tr>
<td>y_position</td>
<td>4</td>
</tr>
<tr>
<td>alive</td>
<td>true</td>
</tr>
<tr>
<td>frozen</td>
<td>false</td>
</tr>
<tr>
<td>on_trap</td>
<td>false</td>
</tr>
<tr>
<td>Gold Camel</td>
<td></td>
</tr>
<tr>
<td>x_position</td>
<td>3</td>
</tr>
<tr>
<td>y_position</td>
<td>5</td>
</tr>
<tr>
<td>alive</td>
<td>true</td>
</tr>
<tr>
<td>frozen</td>
<td>false</td>
</tr>
<tr>
<td>on_trap</td>
<td>false</td>
</tr>
<tr>
<td>Gold Horse_one</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Gold Horse_two</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Gold Rabbit_five</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Conclusion

From this thesis work we have proven that it is possible to use convolutional neural networks for complex board games. Arimaa have properties which make the convolutional networks more beneficial compared to chess since all of the pieces have the same rules for moving, and effectively have a radius of influence.

As a proof-of-concept, the results show great potential, and exploring further with different configurations is highly interesting. There are still many choices for designing a convolutional neural network and the optimizers.

Unfortunately the initial goals for the evaluation was not met. Although with more experiments on the size and configuration of the neural network it does not look impossible to outperform a handmade evaluation function. Since Arimaa have a very high amount of possible moves on each position it would be very beneficial to reduce the search depth. If an evaluation function manage to perform better than another with one less search depth it would still be an improvement even if the evaluation would be 10 000 times slower.

Considering the time constraints of the thesis work, the results are very pleasing. After training each network for around a hundred iterations the evaluation begins to take form, which was faster than originally anticipated.

In this thesis work the structure of the input for a neural network have been described and tested for a convolutional neural network. On top of this, another design of the input have been described if using a dense neural network instead of a convolutional neural network.
Bibliography


[8] François Chollet on Twitter: “@rbhar90 @tensorflow we will be integrating Keras (TensorFlow-only version) into TensorFlow.” URL: https://twitter.com/fchollet/status/820746845068505088 (visited on 05/03/2017).


Match against meijin

Gold: Neural Network, Silver: Meijin

1g Ra1 Rb1 Rc1 Dd1 De1 Rf1 Rg1 Rh1 Ra2 Hb2 Cc2 Md2 Ee2 Cf2 Hg2 Rh2
1s ra7 hb7 cc7 ed7 me7 cf7 hg7 rh7 ra8 rb8 rc8 dd8 de8 rf8 rg8 rh8

2g Hb2n Hg2n Cc2n Rc1n
2s ed7s me7s hb7s hg7s
3g Ee2n De1n Cf2e De2e
3s ed6s ed5s ed4s me6w
4g Rb1e Dd1e De1n Cc3n
4s mb4n Hb3n mb5w Hb4n
5g Ee3n Ee4v Ra2n Ra3n
5s md6s md5w mc5w mb5s
6g Ed4n Ed5w Hb5s Hb4s
6s ed3e Md2n ma5e mb5s
7g Ec5w hb6w Ee5n
7s dd8s dd7s de8s de7s
8g Eb6n Eb7s cc7w Rc1w
8s ce7w cd7w
9g De2v Md3w Dd2n
9s Dd3s ee3w ed3e Mc3e
10g Md3n Md4n dd6s Md5n
10s ee3n ee4n de6e de5n
11g Re1e Dd2n Dd3w
11s Cc4n mb4e mc4w Dc3n
12g Cc5e Md6n Md7n cc7e dc6x
12s cb7e cd7n Md6n ee6w
13g Eb6n Eb7s cc7w Rf1w
13s ch8s ce8s ce7s
14g Eb6s Eb5e Re1e Dc4s
14s rb7e mc2w mb2e Hb3n
15g Ee6n Eb7s cc7w Rf1w
15s re1n Re2n cb3w Hb2n
16g cb8e cd8e ce8s ce7s
16s re2e cb3w Hb3e
e17 g Ha3s Ha2e Cd5s Cd4w
17s cb7e cd7e Md6n Md7n
18g Ha2n Rain Ha3e Dc3e
18s mc2v ha5n Re4n
19g Rd2v Dd3s Rc2n Dd2w
19s De2e mc2e cb6s ha6e
20g cb5s Ed5v hb6w Ee5n
20s mc2v mb2e Hb3s cb4s
21g Rb1v Rf1v cb3w Hb2n
21s mc2v me2b Hb3s ca3e

41g Rg1v Rf1v cb3w Hc3w
41s mc2w mb2e Hb3s ca3e