Chapter 1

Introduction

This project analyses a chess engine trained by self-play. A chess engine simulates moves with a search algorithm and predicts their chances of winning using an evaluation function. This function was approximated with a neural network and trained by deep reinforcement learning. The neural network, search and learning algorithms were implemented in `C++` and `Python`, interfacing with components from Stockfish. Within a chess game of 80 plys, agents trained with TD(λ) and TD-Leaf(λ) were sigificantly stronger than an opponent playing random moves to 95% confidence, fulfilling the success criteria. Overall, the project demonstrates that deep reinforcement learning can train an agent to discover strategies and subgoals from first principles, without the need for human knowledge or interference.

1.1 Motivation

The dream of Artificial Intelligence (AI) has been described as "our last invention" [5]. Capable of empowering, befriending or destroying humanity, this subfield of Computer Science has been prominent in media and popular culture. However, many AI architectures are not currently deployed in the overwhelming complicated natural world. Instead, their capabilities are evaluated in artificial environments such as chess.

Chess is ideal for evaluation because it is small enough to simulate but not trivially small. Its size is determined by its branching factor, the average number of possible next states. In chess, this number is around 20. Because of this property, research from figures such as Alan Turing, Claude Shannon, IBM and Google DeepMind has developed chess playing agents, or engines, of increasing strength [11] [16] [7] [18]. Nowadays, chess engines of superhuman ability can be found on ubiquitous commercial hardware.

Early chess engines use knowledge engineering algorithms. They relied on hand-crafted evaluation functions which could take years to develop. [9] These cannot be transferred to other games and may even be suboptimal. To resolve these issues, agents should reduce their dependency on human information and bias by learning from its surrounding environment. By augmenting the environment with a reward signal, reinforcement learning

trains an agent to discover strategies and subgoals. Through exploration, an agent learns to maximise the reward, or payoff, obtained when goals are met. [19] In chess, wins are rewarded positively and losses negatively, enabling an agent to learn from successes and mistakes. The agent is trained by self-play, because the ideal opponent when learning is one of equal strength. [4]

Recent achievements in machine learning have been due to the subfield of deep learning. Deep learning algorithms train a neural network to approximate functions by inferring representations from a dataset. [1] By training a neural network to approximate an evaluation function with reinforcement learning, a chess agent can be trained by deep reinforcement learning. [17] [12]

The aim of this project was to train a chess engine using deep reinforcement learning such that the difference in strength between it and an opponent playing random moves is statistically significant to 95 %. Such an agent could be evaluated not just against knowledge engineering or machine learning agents, but against the many centuries of human chess knowledge. A chess engine trained with minimal human knowledge would not only create a new opponent, but may contribute new knowledge to chess.

The only way of determining that the agent has been successfully trained is by playing against an opponent which moves randomly. As the agent is trained by self-play, and it gets stronger with every game, the agent is more likely to perform better against strong opponents, rather than against any opponent. To ensure that the agent can play against all opponents, it should be play against an agent generating random moves, capable of emulating an opponent of any strength. As the average length of a game of chess is 80, it was chosen as an upper bound on the length of a game of chess.

1.2 Related Work

One of the earliest reinforcement learning chess engines is KnightCap. The evaluation function was a linear combination trained using TD-Leaf(λ) against humans. After 300 games, KnightCap was as strong as a human master. [6]

Giraffe was developed in 2015 and is similar to this project. It is one of the first agents to be trained using deep reinforcement learning. The evaluation function was a four layer neural network trained by a variant of TD-Leaf(λ). After 50000 games of self-play, Giraffe reached the standard of an international master. [10]

Chapter 2

Preparation

A literature review of three chess engines and three reinforcement learning algorithms was performed in order to refine the plan in the project proposal. All of the engines examined had open-source codebases. The chess engines Giraffe, Stockfish and Sunfish were examined. Reinforcement learning algorithms were investigated using their codebase and the papers of Giraffe, the chess engine KnightCap and the backgammon agent TD-Gammon. A summary of findings is duly presented, as well as the refined plan.

2.1 Components of a Chess Engine

A rational chess engine plays to maximise its probability of winning. It selects moves with a search algorithm simulating all possible games and outcomes from the current chessboard position. The best move leads to the greatest proportion of positive outcomes. Unfortunately, this strategy is infeasible for chess because there are too many outcomes to search. Chess lasts 10 minutes, so each player has a total of 5 minutes thinking time. An average game has 10^{120} outcomes and 80 moves, known as plys. The average thinking time for each ply is less than 5 seconds, an unreasonably short time to simulate all outcomes. To reduce computation, a heuristic called an evaluation function approximates the probability of winning given a board position. Instead of simulating to conclusion, a search algorithm uses this heuristic after a simulating a few plys.

The strength of a chess engine depends on the speed of its search algorithm and the accuracy of its evaluation function. The evaluation function in this project is a neural network. It is trained by deep reinforcement learning, a process which trains the neural network from its experience of playing games. The most important component of the project is the search algorithm, because it is thus used in training as well as playing. Its speed is bottlenecked by its move generator, a function which the search algorithm is built around. This function is called for every move and returns a list of legal moves, enabling all possible games to be simulated. The project adapted move generators from open-source chess engines due to the complexity of reimplementing and optimising another.

2.2 Search Algorithms

Minimax was the first search algorithm to utilise an evaluation function. Since its publication in 1950 by Claude Shannon, it is used by the majority of chess agents. [16] In 1959, John McCarthy improved Minimax using an alpha-beta pruning heuristic. This recognised that further evaluations of a move were unnecessary if a better alternative had already been found. This could be further improved by ordering the moves such that they were pruned as often as possible. A turning point in computer chess occurred in 1967, when Richard Greenblatt used a transposition table to cache chessboard positions and scores. His agent, Mac Hack IV, became the first chess engine to defeat a human. [14] Since then, increasingly advanced and specific heuristics have been developed. These were not implemented because they are empirically derived from human bias and may limit the teaching ability of reinforcement learning.

2.2.1 Minimax Searching Algorithm

Minimax is a portmanteau of minimise and maximise, two functions which compose its operation. Minimax is applied to two player zero-sum games like chess, where one player's payoff or reward is negated from the other. As a rational player maximises potential payoff, this minimises their opponent's potential payoff. By assuming that the opponent is also rational, Minimax finds moves by recursively maximising what their opponent has minimised. This algorithm has a complexity of $\mathcal{O}(b^d)$. [8] An efficient way of implementing this is Negamax, Algorithm 1.

Algorithm 1 Negamax

This implementation searches to depth *depth* from root position *position*, returning the best move and its score. The function should be called by NEGAMAX(*position*, *depth*−1).

```
1:  function NEGAMAX(position, depth)
2:      if depth = 0 or noFurtherPossibleMoves() then
3:          return _, EVALUATE(position)
4:      max ← −∞
5:      maxMove
6:      for move in moveGenerator(position) do
7:          doMove(position, move)
8:          score ← −NEGAMAX(position, depth − 1)[1]
9:          undoMove(position, move)
10:         if score ⩾ max then
11:             max ← score
12:             maxMove ← move
13:     return maxMove, max
```

In Game Theory, Minimax finds a Nash equilibrium. This is a state of play where changing strategies leads to suboptimal outcomes if their opponent does not change strategies. Hence Minimax should be used to find moves throughout a match. [14] [10] [8]

2.2.2 Alpha-Beta Pruning Heuristic

Instead of evaluating a move with a single score, α and β represent the best and worst possible outcomes. As a rational agent maximises alpha and minimises beta, search can be halted if a move returns a value for β less than the existing value, as neither agent would select this move. Thus the efficiency of a search with branching factor b and depth d can be improved to $\mathcal{O}(\sqrt{b^d})$.

Moves should be ordered such that best moves are searched first, pruning as many nodes as possible. As a poorer move has larger β, it has a higher possibility of being pruned. Ordering can be done with a sorting algorithm operating on scores given by the evaluation function. [14] [8] Pseudocode of alpha-beta search is in Algorithm 2, with move ordering in line 6.

Algorithm 2 Alpha-Beta Negamax

This implementation searches to depth *depth* from root position *position*, returning the best move and its score. Differences from NEGAMAX are emphasised. The function should be called by ALPHABETA(*position*, ∞, $-\infty$, *depth* $-$ 1).

```
1: function alphaBeta(position, alpha, beta, depth)
2:     if depth = 0 or noFurtherPossibleMoves() then
3:         return _, evaluate(position)
4:     maxMove
5:     moves ← moveGenerator(position)
6:     moves ← orderByDescendingScore(moves)
7:     for move in moves do
8:         doMove(position, move)
9:         score ← −alphaBeta(position, −beta, −alpha, depth − 1)[1]
10:        undoMove(position, move)
11:        if score ⩾ beta then
12:            return _, beta
13:        if score > alpha then
14:            alpha ← score
15:            maxMove ← move
16:    return maxMove, alpha
```

2.2.3 Memoisation using Transposition Table

Searching may yield positions which have already been evaluated. If results are memoised, computation is reduced if stored values are reused. Transposition tables memoise the best next move, the score of this position, and the search depth at which these were found. If the depth is less than required, the stored move and score may not be the best values, and should be ignored. Typically, transposition tables are 2^{20} entries in size and are indexed using Zobrist hashing. This compression function represents a chessboard

as an integer. [8] Pseudocode of accessing a transposition table in alpha-beta search is Algorithm 3.

Algorithm 3 Alpha-Beta Negamax with Transposition Table

This implementation searches to depth *depth* from root position *position*, returning the best move and its score. Differences from alphaBeta are emphasised. The function should be called by alphaBetaTT($position, \infty, -\infty, depth - 1$).

```
1: function alphaBetaTT(position, alpha, beta, depth)
2:     if Zobrist(position) in TranspositionTable then
3:         transposition ← TranspositionTable[Zobrist(position)]
4:         if transposition.depth ⩾ depth then
5:             return transposition.move, transposition.score
6:     if depth = 0 or noFurtherPossibleMoves() then
7:         return _, evaluate(position)
8:     maxMove
9:     moves ← moveGenerator(position)
10:    moves ← orderByDescendingScore(moves)
11:    for move in moves do
12:        doMove(position, move)
13:        score ← −alphaBetaTT(position, −beta, −alpha, depth − 1)[1]
14:        undoMove(position, move)
15:        if score ⩾ beta then
16:            return _, beta
17:        if score > alpha then
18:            alpha ← score
19:            maxMove ← move
20:    index ← Zobrist(position)
21:    TranspositionTable[index].score ← alpha
22:    TranspositionTable[index].move ← move
23:    TranspositionTable[index].depth ← depth
24:    return maxMove, alpha
```

2.3 Evaluation Function

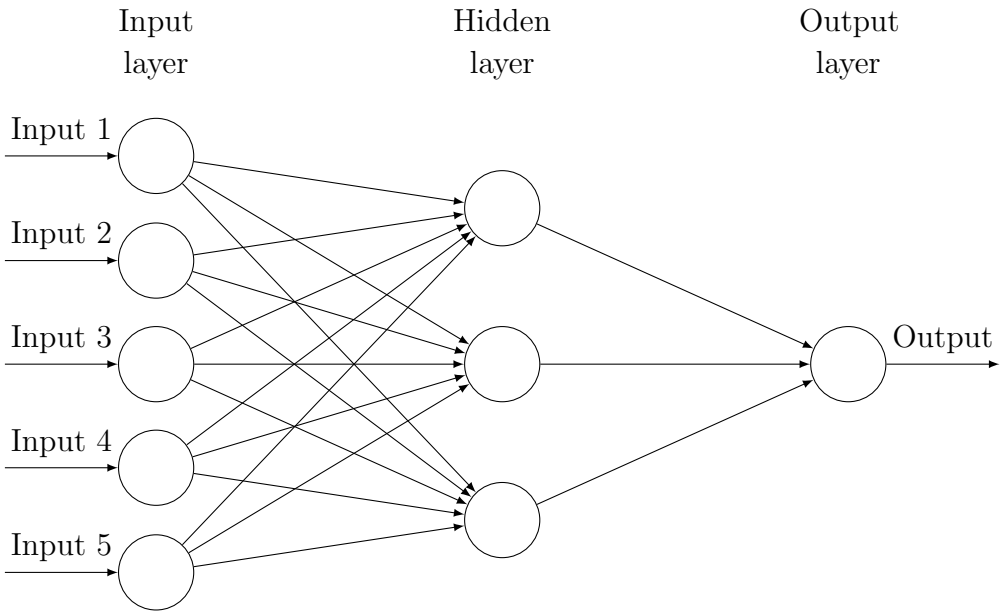
The evaluation function assigns relative scores to chessboard positions such that the best score has the greatest value. In this project, scores are normalised to $(-1, 1)$, where -1 and 1 predicts checkmate for black and white respectively. As -1 and 1 are also the rewards for black and white checkmate in reinforcement learning, the evaluation function is trained to approximate $\mathbb{E}(reward \mid chessboard)$. The evaluation function was calculated by a neural network, due to their versatility and performance. Since 2008, new techniques to train neural networks as well as various architectures have improved their performance. [1] Rebranded as deep learning, neural network inspired algorithms

currently achieve the best performance of all machine learning methods. [1] For simplicity, the project only utilises feedforward neural networks.

2.3.1 Feedforward Neural Networks

A feedforward neural network consists of multiple layers, as shown in Figure 2.3.1. Its structure is a directed acyclic computational graph where each node is a mathematical abstraction of a neuron and each edge represents a connection. Every neuron in feedforward neural networks is connected to every neuron in the next layer. As there can be any quantity of hidden layers of arbitrary size, finding its optimal architecture is NP-hard.

Figure 2.1: Abstract representation of a feedforward neural network



Given an input vector, a network can infer output values. This is computed by a forward pass which evaluates each neuron in turn, passing its value to the next layer along an edge. Each neuron performs the following calculation and returns y with an input of vector $\mathbf{x}$ in a forward pass.[1]

$$y = \sigma(z) = \sigma\left(b + \sum_i x_i w_i\right)$$

The nonlinear function σ enables the network to learn nonlinear functions. This project uses Rectified Linear Units (ReLU) and hyperbolic tangent activation functions for hidden and output layer neurons respectively. ReLU is the most popular activation function and is calculated by $\mathrm{ReLU}(z) = \max(0, z)$, whereas tanh constrains the output to $(-1, 1)$. [1]

The weight vector $\mathbf{w}$ and bias b are parameters of each neuron. The set of all parameters of the network is referred to as the weights. These are updated in a backwards pass during training. A network with n and m neurons in the input and output layers respectively can be trained to approximate $\mathbb{R}^n \to \mathbb{R}^m$. A popular training algorithm is backpropagation. [1]

2.3.2 Backpropagation Learning Algorithm

Backpropagation utilises the multivariable chain rule to update the parameters of each neuron. It alters their parameters using first-order gradient descent calculated from a loss function measuring the difference between the observed and expected output values. A popular loss function is the Minimum Squared Error (MSE). [1]

$$C_{\text{MSE}} = \frac{1}{2n} \sum_{i=0}^{n} (y_{observed,i} - y_{expected,i})^2$$

Let $\delta_i^l = \frac{\partial C}{\partial z_i^l}$ is the change of cost with respect to neuron i in layer l. Thus changes in output layer parameters are:

$$\frac{\partial C}{\partial b_i^l} = \alpha \delta_i^l$$
$$\frac{\partial C}{\partial w_{i_k}^l} = \alpha (y_k^{l-1} \delta_i^l)$$

Here, the learning rate α amplifies and scales the change in weight for more efficient gradient descent. For weights in the output layer, δ_i^{output} for the i^{th} can be calculated by

$$\delta_i^{output} = \frac{\partial C}{\partial y_i^{output}} \sigma'(z_i^{output})$$

Otherwise,

$$\delta_i^l = \sum_k w_{k_i}^{l+1} \delta_i^{l+1} \sigma'(z_i^l)$$

Weight w_{i_k}, which connects neuron i to k can be updated thus, in a process known as stochastic gradient descent.

$$w := w - \frac{\partial C}{\partial w_{i_k}^l}$$

Before training, the parameters of each neuron can be initialised to any starting value. To pick an unknown starting point, these values were randomised. Generally, larger networks need more data and thus training time to converge to a global minimum. Increasing the number of layers enable more complicated functions to be learnt, due to the composition of more simple functions. Increasing the number of neurons in hidden layers change local minima in gradient descent into saddle points. Training a network using (input, output) pairs is known as supervised learning.

The architecture of the neurons, as well as the learning rate α are hyperparameters which are fixed before training. Hence training is often sensitive to hyperparameters. A method of finding a well-performing network architecture is cascade correlation.

2.3.3 Finding Network Architecture using Cascade Correlation

Starting with a hidden layer of one neuron, cascade correlation finds the best architecture by incrementing neurons and layers, selecting the best performing architecture, as illustrated by Algorithm 2.3.3. The algorithm used in the project differs from the original in two ways. First, the best performing architecture is selected according to their accuracy on a testing set, instead of values of loss. Secondly, weights are randomised when new neurons are added, rather than keeping their values. This is because cascade correlation is used to find an architecture for reinforcement learning, trained with random weights. The original algorithm was intended to be applied to a dynamic supervised learning problem.

Algorithm 4 Cascade Correlation

Cascade correlation utilises the inputs $n_{input}, n_{output}, upper$. These represent the width of the input layer, output layer and the maximum width of a layer. The optimal performing architecture of hidden layers are stored in the parameter *hidden*.

```
1:  function CASCADECORRELATION(n_input, n_output, upper)
2:      hidden ← empty
3:      while hidden[LENGTH(hidden) − 1] > 1 do
4:          bestScore = −1
5:          bestHidden
6:          for each integer i from 0 to LENGTH(hidden) do
7:              architecture ← (n_input, hidden, i, n_output)
8:              network ← MAKEARCHITECTURE(architecture)
9:              TRAIN(network)
10:             score ← EVALUATE(network)
11:             if score > bestScore then
12:                 bestScore ← score
13:                 bestHidden ← i
14:         append i to hidden
```

2.4 Board Representation

As feedforward neural networks require one dimensional inputs, board representations convert two dimensional chessboards with twelve types of pieces into a representation vector. They are normalised to $(-1, 1)$ to increase the learning rate. Most chess engines have additional features in the representation vector which encode additional information, such as whether the King is in check or which pieces protect or attack others. These were not included in this project, because they can be expressed as linear functions of the pieces' coordinates. A good network architecture would be able to infer this information from the location of pieces. In addition, the inclusion of features may be partial to human bias. This is one of the first chess engines not to use additional information. The three main categories of board representations are presented below. [10] [21]

2.4.1 Mailbox

The mailbox representation is a vector of length 64, each element corresponding to the squares on a chessboard. The value of each element is zero if no piece is present on the corresponding square. Otherwise, their value is given by the table below.

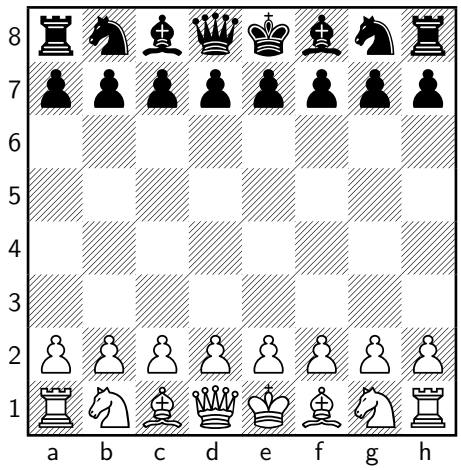
Piece	**White**	**Black**
Pawn	1	-1
Rook	2	-2
Knight	3	-3
Bishop	4	-4
Queen	5	-5
King	6	-6

2.4.2 Piece-List Coordinates

The piece-list representation is a flattened coordinate list of chesspieces. The order the coordinates appear in are given by the following table.

Piece	**Maximum Number of Pieces**	**White**	**Black**
Pawn	8	0 – 7	8 – 15
Rook	10	16 – 25	26 – 35
Knight	10	36 – 45	46 – 55
Bishop	10	56 – 65	66 – 75
Queen	9	76 – 84	85 – 93
King	1	95	96

Rank and file coordinates of each piece are given below, with {a – h} instead of {1 – 8} for the file.



2.4.3 Bitboard

The bitboard board representation is similar to the mailbox board representation. Each piece can be represented as a tuple of (rank, file, piece type). This is used to index a third-order tensor of dimensions $\{8, 8, 12\}$. Elements in the tensor indexed by pieces are set to 1 and 0 otherwise. The resulting tensor is then flattened into a vector.

Piece	White	Black
Pawn	0	6
Rook	1	7
Knight	2	8
Bishop	3	9
Queen	4	10
King	5	11

2.5 Reinforcement Learning

Reinforcement learning teaches an agent to play through experience. In this project, reinforcement learning trains a neural network to approximate an evaluation function by self-play. The resulting evaluation function is known as a value network. After a self-play game, a reward is returned to the agent. This is 1, 0 and -1 for a white win, draw and black win. These rewards are propagated to score every move in the game. Moves which occur closer to the end influence the reward more, whereas moves which occur earlier on may not impact the result as much. Thus rewards for moves which occur earlier should be discounted more. This process is done by temporal difference (TD) algorithms. TD learning algorithms used in this project are TD(λ) [20] and TD-Leaf(λ) [6].

Training was done by self-play because of two reasons. First, an ideal opponent when training is one of similar strength. As the outcome would have an equal ratio of winning and losing, this prevents the agent to learn strategies for attack and defence simultaneously, avoiding bias towards one extreme. Secondly, self-play enables an engine to train without human intervention, which may be unknowingly flawed and suboptimal. [4]

2.5.1 TD(λ)

In a game of T timesteps, let $\mathbf{x}_t$ be the board representation of the game at timestep t. Each $\mathbf{x}_t$ is associated with a value v_t calculated from the evaluation function f such that $f(\mathbf{x}_t) = v_t$. The reward returned by a move at timestep t is thus the difference d_t between two consecutive timesteps $v_{t+1} - v_1$.

$$\begin{aligned} d_T &= reward - v_T \\ d_{T-1} &= v_T - v_{T-1} \\ &\vdots \\ d_0 &= v_1 - v_0 \end{aligned}$$

If both players play optimally, both their value networks are optimal. In addition, if both follow a Minimax strategy, the players reach a Nash equilibrium, ensuring that neither player changes strategy. Therefore the value returned by the evaluation function would remain the same between timesteps: $v_t = v_{t+1}$. By transitivity, $v_0 = v_1 \ldots reward$. The value v_t is said to back up v_{t+1}. Using the Minimum Squared Error, the cost function is

$$\frac{1}{2}\sum_{i=0}^{T-1}\sum_{t=i}^{T-1}(v_{t+1} - f(\mathbf{x}_t))^2$$

During training, the value network will not be performing accurately, which may lead to the agent selecting suboptimal moves. Let the probability that $v_t = v_{t+1}$ is true be λ. This probability is known as the discount rate. Discounts follow a geometric progression such that a pair of values which are more timesteps apart are discounted more. With backups, the cost function is

$$\frac{1}{2}\sum_{i=t}^{T-1}\sum_{t=i}^{T-1}\lambda^{i-t}(v_{t+1} - f(\mathbf{x}_t))^2$$

Gradient descent with the modified cost function yields the formula for updating weights according to TD(λ).

$$
\begin{aligned}
w &:= w - \nabla_w \frac{\alpha}{2} \sum_{i=t}^{T-1} \sum_{t=i}^{T-1} \lambda^{i-t} (v_{t+1} - f(\mathbf{x}_t))^2 \\
w &:= w + \alpha \sum_{i=t}^{T-1} \sum_{t=i}^{T-1} \lambda^{i-t} (v_{t+1} - f(\mathbf{x}_t)) \nabla_w f(\mathbf{x}_t) \\
w &:= w + \alpha \sum_{i=t}^{T-1} \nabla_w f(\mathbf{x}_t) \sum_{t=i}^{T-1} \lambda^{i-t} (v_{t+1} - v_t)) \\
w &:= w + \alpha \sum_{i=t}^{T-1} \nabla_w f(\mathbf{x}_t) \sum_{t=i}^{T-1} \lambda^{i-t} d_t \\
w &:= w + \alpha \sum_{i=t}^{T-1} \nabla_w f(\mathbf{x}_t) c_t
\end{aligned}
$$

where $c_t = \sum_{t=i}^{T-1} \lambda^{i-t} d_t$ is the eligibility trace. An easy way of computing it below.

$$
\begin{aligned}
c_t &= \sum_{i=t}^{T-1} \lambda^{i-t} d_i \\
&= d_t + \lambda c_{t+1}
\end{aligned}
$$

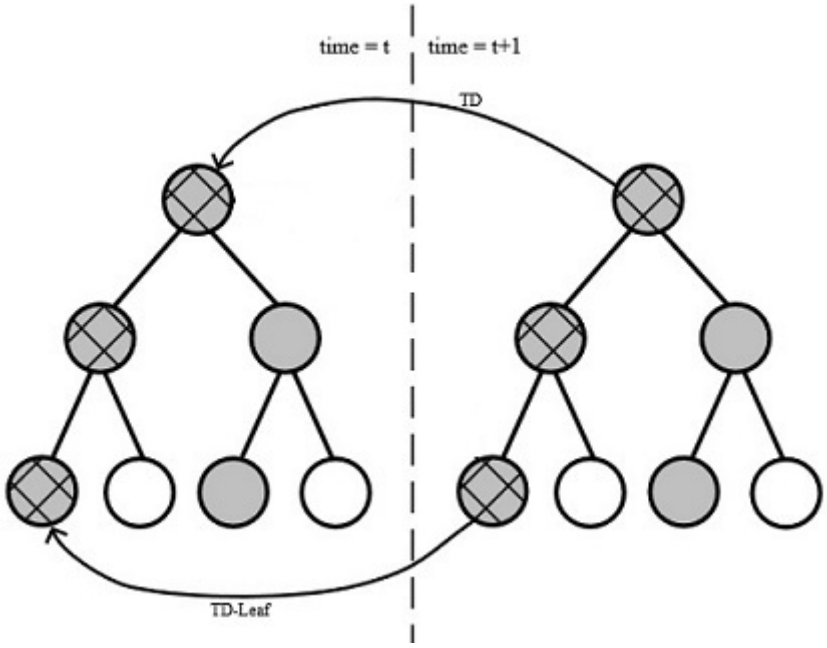
At the edge cases $\lambda = 0$ and $\lambda = 1$, the following identities are noted.

$$
\begin{aligned}
c_t &\equiv reward : \lambda = 0 \\
c_t &\equiv v_{t+1} - v_t : \lambda = 1
\end{aligned}
$$

When $\lambda = 0$, the equation becomes akin to Monte-Carlo methods, whereas $\lambda = 1$ methods are similar to dynamic programming methods. Hence intermediate values of λ interpolate the two methods. A common value for λ is 0.7. [23]

2.5.2 TD-Leaf(λ)

Instead of backing up successive chessboard positions, TD-Leaf(λ) backs up the leaf of the principle variation of each position. The principle variation is the best sequence of moves found by minimax. Its leaf node is the resultant board position after all principle variation moves.



The above picture adapted from [22] shows how values are backed up in TD(λ) and TD-Leaf(λ). Using the leaf instead of the search root speeds up learning. The update rule for TD-Leaf(λ) is thus

$$w := w + \alpha \sum_{i=t}^{T-1} \nabla_w f(pv(\mathbf{x}_t)) c_t$$

where $pv()$ is a function which returns the principle variation leaf of a board position.

2.6 Requirements Analysis

After completing a literature review of chess engines and reinforcement learning, requirements to fulfil the project objective could be was defined, in order to refine a detailed implementation plan from the project proposal. Factors considered during the planning phase were programming languages, hardware, professional practice and ethical considerations.

2.6.1 Deliverables

The initial plan was to adapt Giraffe to utilise `Python` neural networks. However, Giraffe's code is highly optimised and tightly integrated. An easier strategy was to adapt another chess engine such as Stockfish. There is a large research community using Stockfish, and its code is well commented and documented. Thus adapting Stockfish instead of Giraffe would increase accessibility of the project to a wider audience.

A successful project would implement a trainable agent, a 'random agent' playing randomly selected moves and a test suite to pit them against each other. As discussed above, the trainable agent requires search and evaluation functions as well as reinforcement learning algorithms. Performance of the value network could be improved by finding a good board representation and network architecture, as well as selecting good values for hyperparameters λ and α. The components, their priority and difficulty are summarised in the following table.

Component	**Priority**	**Difficulty**
Search Algorithm	Medium	Medium
Evaluation Function	High	Low
Reinforcement Learning Algorithm	High	High
Random Agent	High	Low
Best Board Representation	Medium	Low
Best Network Architecture	Medium	Medium
Hyperparameter Optimisation	Low	Medium

2.6.2 Starting Point

The two languages used in the project are `C++` and `Python`. Most chess engines are written in `C++` due to speed and customisability. `Python` libraries such as `PyTorch` and `Keras` are currently the two most popular deep learning frameworks in research and industry. Both libraries were used to find the best. A table of software used and their starting point for learning is presented below.

Software	**Starting Point**
`Python`	Self-taught
`PyTorch`	Online Tutorials
`Keras`	Online Tutorials
`C++`	Programming in `C/C++` (second year course)
Giraffe	BitBucket Repository
Stockfish	GitHub Repository
Sunfish	GitHub Repository

In addition, the courses Concurrent and Distributed Systems provided useful information in inter-process communications (IPCs) between `Python` and `C++`. Artificial Intelligence was useful for understanding neural networks. Knowledge of reinforcement learning was gained from reading Reinforcement Learning by Sutton and Barto as well as analysing the KnightCap, TD-Gammon and Giraffe.

During implementation, a dataset of chess games was generated from data on the Free Internet Chess Server (FICS). This was used to compare board representations and network architectures.

2.6.3 Ethical Considerations

A newly implemented chess engine could be a novel opponent humans could play against, recreationally or competitively. Chess engines are often used to analyse board positions and possible strategies. Any new knowledge the engine learnt would enable players to improve their understanding.

The obvious commercial application of this project is a chess engine which plays against humans, providing feedback and guidance of possible moves. The engine could come untrained and gradually learn to predict and possibly counter the moves of a user, teaching both parties alike. Each chess engine would therefore be specifically tailored to its owner.

Data taken from FICS was freely available. This was a database of internet chess games played in a public online forum since 2008. [15]

2.7 Refined Plan

Due to the complexity in finding the best components for the project before their deployment in the final codebase, development was laid out in three stages.

1. Create baselines for machine learning and knowledge engineering chess agents, find the best libraries and packages.
2. Implement reinforcement learning algorithms, find best board representation, network architecture and hyperparameters.
3. Deploy neural network and search algorithm in a fast and efficient language such as `C++`, using a codebase such as Giraffe.

2.8 Software Engineering Practices

Software was developed according to the spiral model, with each stage a new spiral. Within each stage, Agile was used for rapid development. A kanban was used to visualise goals through the medium of a development flowchart. Prototypes were iteratively and modularly refined until the stage's goal was met.

To prevent loss of data, code was backed up to `GitHub`, trained network weights were manually shared between computers and backed up to Google Drive. Each phase of the project was modular and kept in a distinct repository to prevent the git logs from being overwhelmingly long. New variants of algorithms were unit tested against previous versions.

2.8.1 Hardware Used

Two machines were used in the project:

Macbook Pro, late 2015 (El Capitan, 16 GB RAM, 2.5 GHz Intel Core i7, 500 GB SSD) Used in code development and dissertation writing.

Desktop Computer (Ubuntu, 32 GB RAM, 3.6 GHz Intel Core i7, Nvidia GeForce GTX 1080, 256 GB SSD) Used in code development, agent training, running and evaluation, due to its higher specs.

If the above machines were to fail, an MCS machine, a research machine from the Computer Laboratory Computational Biology Group and the Cambridge High Performance Computing Services could replace the above devices through cloning the `GitHub` repositories and any required weights.

2.9 Conclusion

Following a brief overview of algorithms and techniques used in the project, a refined plan was created to provide a structure for implementation. Details of how the value network, search algorithm and reinforcement learning algorithms were implemented are explained in the next chapter.

Chapter 3

Implementation

This chapter documents the implementation of the algorithms and plan discussed above. The three developmental phases of the spiral plan are chronologically documented, highlighting milestones in implementation. Challenges in developing and testing each component are elaborated with details of how they were resolved. The chapter concludes with a description of the final codebase. As evaluation of certain components of the chess engine influenced the development of the next stages during spiral development, some evaluations and analysis are presented along the way.

3.1 Deep Learning with Giraffe

In order to gauge the scope of the project, relevant libraries and programming languages were explored and analysed. To achieve this, Giraffe was adapted to utilise `Python` neural networks. The similarity between Giraffe and the architecture of this project's chess engine made it ideal for initial investigation. As it was developed in 2015, before deep learning libraries were developed, it utilised a neural network written in `C++` and ported to `Lua`.

Giraffe was well-documented with publicly available source code and weights. Its evaluation function is a 4 layer neural network. [10] This was reimplemented in `PyTorch` and `Keras`, to compare their performance with the original `C++` network. `Keras` is a library which is interpreted into `TensorFlow`, which itself calls `C++` subprocesses. [2] `PyTorch` also wraps `C++`, but dynamically creates computational graphs and allocates resources. [13] The weights were converted from a `t7` to `h5` format such that they could be imported in `PyTorch` and `Keras`. To interface these networks with `C++`, an IPC between the two languages was implemented such that Giraffe could evaluate chessboard positions using these `Python` networks. The resultant agent was pitted against `FairyMax` and the author within `XBoard` to determine their strength. `XBoard` was used to analyse the speed, search depth and moves considered by the agent. Speed of evaluation was found to be proportional to strength. `C++` was the fastest, followed by `PyTorch` and `Keras`.

After observing the performance of `Python` neural networks, the next step would be to implement TD(λ) and TD-Leaf(λ) in Giraffe. Their implementation required searching to return a list of nodes for temporal difference backups. Thus the search algorithm would have to be reimplemented, requiring the move generator to be isolated. To gain a deeper understanding of the source code, UML diagrams were generated using BoUML. These diagrams revealed a highly optimised and occasionally underdocumented class hierarchy. Modifying other chess engines for reinforcement learning was easier, even though `Python` neural networks had already been successfully incorporated in Giraffe. The alternative would be to implement the search algorithm and move generator from scratch. As many efficient implementations already exist in numerous chess engines, this would be an inefficient use of time to obtain a suboptimal result.

3.2 Rapid Prototyping in `Python`

Having observed the performance of an evaluation function using trained weights from Giraffe, strategies for training it needed to be found. Factors which could affect training rate were the network architecture, board representation and training hyperparameters. The best configuration of all components needed to be determined for efficient training. In addition, reinforcement learning algorithms for training needed to be implemented and verified. As the Giraffe move generator was not going to be used, an alternative was found in Sunfish. This chess engine is written in 111 lines of `Python`, enabling the whole engine to be analysed and the move generator quickly identified. [3] `Python` was chosen as the language for rapid prototyping, testing and evaluation because it is terse, interpretive, and high-level. In contrast, the final codebase would be developed in `C++` for speed and efficiency.

3.2.1 Analysing Representations with Supervised Learning

The most obvious strategy to analyse board representation and network architecture would be to implement the whole reinforcement learning algorithm and observe how changing their configuration affects training. This is inefficient because reinforcement learning does not intensively train the network. More time is spent searching and generating moves instead. A more network-intensive training strategy is supervised learning.

Section 2.3 shows that the evaluation function approximates $\mathbb{E}(\mathit{reward} \mid \mathit{representation})$. Another way this can be approximated is to train a network to predict the winner given a chessboard representation. Values for black and white checkmate were set to $-1, 1$, making this a supervised classification problem. An alternate approach was supervised regression against normalised values of Stockfish's evaluation function. The shortcomings of this approach were in difficulty of testing and dissimilarity of datasets. Stockfish's evaluation function ranks the favourability of the next move rather than predicting the eventual winner. [9] Training the network to predict normalised values would be a different function than predicting the winner from a chessboard position. Supervised classification

is a better approach because accuracy of predictions could be easily calculated using a validation set.

Training data was generated from the Free Internet Chess Server (FICS) database, a record of all games played on the eponymous server since 2008. [15] These games can be viewed by the public in real time. Games from 2017 and 2016 were used to generate the supervised learning dataset. The downloaded dataset was in `.pgn` format, consisting of a list of games. Each game recorded the moves played, the online handles of the players, their strengths and the outcome. Only games which ended in checkmate were utilised for training. Games which ended in resignations, draws and stalemates were ignored, in order to simplify the dataset as much as possible. In addition, only games where both players were over 1500 ELO were used. ELO scores the strength of a player, and is around the level of a seasoned amateur. Filtering out games played by weaker players reduced the probability of a player winning through their opponent's blunder, reducing noise in the dataset. Hence a more favourable board positions for a player would more likely lead to their checkmate.

A training and testing set of 500000 and 50000 datapoints was generated, because 500000 is roughly the square of the number of neurons in the largest network. These sets were partitioned such that a datapoint used in training would not reappear in testing. Testing data would be drawn from the same distribution of datapoints but would not have been previously seen.

The chessboard position was represented in Forsyth-Edwards Notation (FEN) string, a representation described fully in Appendix A. Chessboard positions were selected at a random point in the game. The parameter *gamestage* was a fraction representing how far the game has progressed. A value of 0.5 meant that the game was halfway, and a value of 1.0 meant that the game was over. This parameter served as a lower bound from which a position could be drawn, with 1.0 as the upper bound. After converting this position to a FEN, datapoints of (FEN, winner) tuples could be saved as a `.csv`. The following algorithm shows how this file was generated in pseudocode.

The algorithm was implemented in `Python`, utilising the `Python-Chess` library. This library was used to read the FICS `.pgn` file and convert a FEN string to and from its own board representation, that of a mailbox (Section 2.4.1). Functions converting the `Python-Chess` mailbox to seven other board representations were created, such that a FEN could be converted into one of the seven representations to be analysed. In addition to the three board representations described in Section 2.4, one other variant of bitboard and three variants of piece-list coordinates were used.

The alternative bitboard representation reduces its dimensions to $\{8, 8, 6\}$. Elements in the tensor are initialised to 0. A white piece would set its index $(rank, file, pieceType)$ to 1, and -1 if the piece is black. After flattening, this representation was known as an antisymmetric bitboard.

Algorithm 5 Generated Supervised Classification Dataset from FICS

The function GENERATEFEN writes a list of FENs representing board position and the eventual winner from this position in $outFile$. The FENs are generated from $FICSfile$ and is the size of $sizeOfDataset$. The parameter $gameStage$ is a float in the range $(0.0, 1)$.

```
1: function GENERATEFEN(FICSFile, gameStage, sizeOfDataset, outFile)
2:     while integer games < sizeOfDataset do
3:         game ← FICSFile.nextGame
4:         if game.playerStrength > 1500 and game.checkmate then
5:             games ← games + 1
6:             board ← startingPosition
7:             ply ← integer(RANDOMFLOAT(gameStage, 1.0) * game.length)
8:             while integer count < ply, increment count do
9:                 MOVE(board, game.getNextMove)
10:            WRITEASFEN(outFile, board, game.winner)
```

Many of the values within a piece-list coordinate representation are 0 because it was unlikely that pawns were ever going to be promoted to fill them. Creating a piece-list according to the following table would statistically reduce the representation size without losing information. Board representations generated using the following table are called compressed piece-lists.

Piece	Number of Pieces	White	Black
Pawn	8	0 – 7	8 – 11
Rook	4	12 – 15	16 – 19
Knight	4	20 – 23	24 – 31
Bishop	4	32 – 35	36 – 39
Queen	4	40 – 43	44 – 48
King	1	49	50

Two coordinate representations for piece-lists were used, one with a simple $(rank, file)$ and the other with $(rank, 9 - rank, file, 9 - file)$. The second representation ensures that the magnitude of a piece's coordinates do not vary with its location of pieces. Representations which have two coordinates per piece are known as small piece-lists.

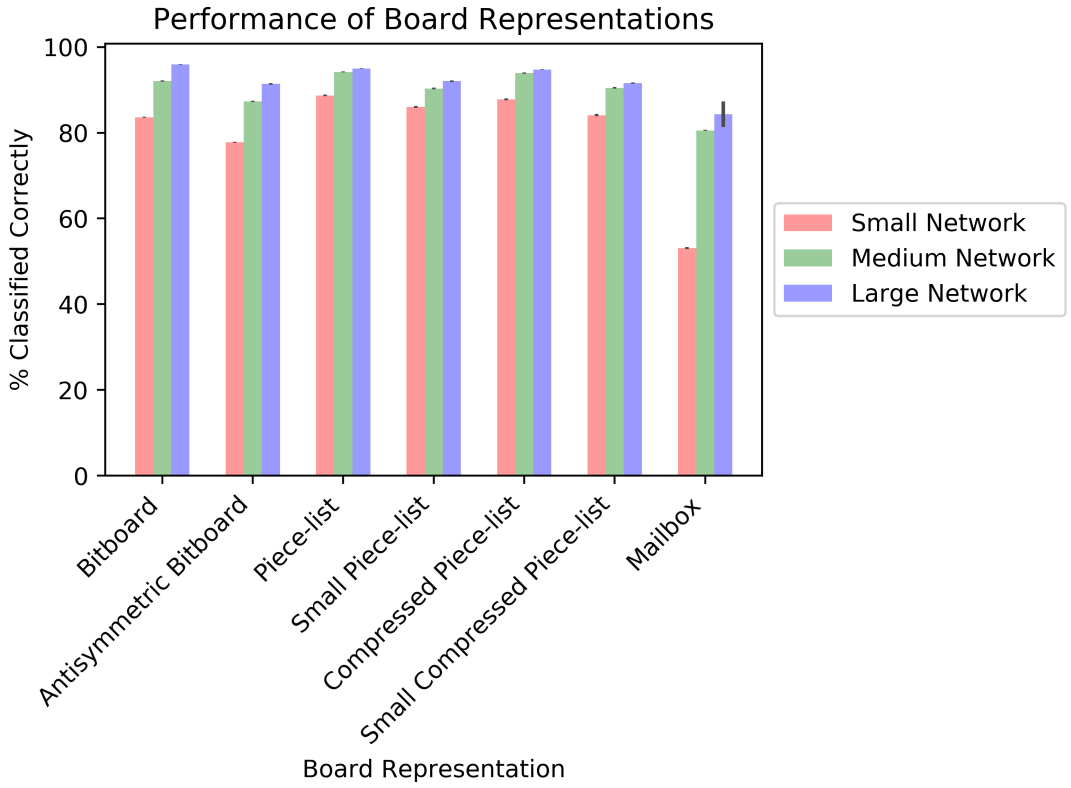
These seven representations were analysed with different network architectures and $gameStage$ values. Values of $gameStage$ were 0.5 and 0.75, and three architectures were used. These architectures were:

1. **Small Network**: Single neuron with tanh activation function. Many chess engines use this because of efficiency. If used without an activation function, the computation is a linear combination.

2. **Medium Network**: Has a hidden layer of 64 neurons.

3. **Large Network**: Giraffe Neural Network with two hidden layers, adapted for different board representations. The first hidden layer has an equal number of neurons as the input layer and the second hidden layer has 64 neurons.

For reliability, the networks were trained ten times, each time starting from a different random parameter instantiation. After testing, the means and standard deviations of the accuracies were computed. Unfortunately, the differences between each network were negligible and statistically insignificant, less so for $gameStage = 0.75$. To obtain as significant a difference as possible, the networks were trained again with a dataset where $gameStage = 1.0$. This dataset only contained checkmate positions. The results after training on this dataset is shown in the graph below, alongside very small standard deviation error bars.



This graph showed significant differences between the accuracies. Overall, the piece-list representation produced the best results. Using this representation, cascade correlation was then used to find a good architecture. It was surprising to observe that the compressed piece-list produced a very similar result, as hypothesised.

3.2.2 Finding Network Architectures with Cascade Correlation

Having concluded that the piece-list representation produced the best results, this was used as the input layer of a neural network. To find the architecture of the hidden layers, cascade correlation was used to find network architectures. The algorithm was

implemented using `Python`, `PyTorch` and `Python-Chess` and trained on the checkmate dataset.

A final architecture returned was {384, 315, 80, 10, 5, 2, 1}. However, this algorithm was not numerically stable given different random initialisations of network weights. Due to the lack of consistent results, and the good scores of networks used in the previous section, the Small and Medium Networks were used as network architectures in subsequent sections. As results between the Large and Medium Networks were similar, the Large Network was not used due to substantially longer training time.

3.2.3 Implementing TD(λ) with Sunfish

Now that a network architecture has been found, deep reinforcement learning algorithms could be developed and verified quickly. This required creating a custom loss function and weight updater. In order to create these functions, an existing implementation of Negamax was modified to return a list of boards and values such that temporal difference values could be calculated. In addition, the neural network was used in evaluation. Overall, extracting the move generator from Sunfish was a lot more straightforward compared to Giraffe.

The move generator returned all possible moves, including illegal moves such as placing the King into check or, through inaction, allowing it to be captured. Such illegal moves had to be filtered out. In addition, Sunfish could not recognise draws, checkmates or stalemates. A function which recognised these different states was created such that rewards could be assigned to players at the conclusion of a game.

The loss function and weight update for TD(λ) was implemented in `Python` using `PyTorch` primitives. This enabled temporal difference arithmetic to be calculated using the GPU, in addition to backpropagation algorithm. As neither TD(λ) or TD-Leaf(λ) is commonly used, they were not part of the `PyTorch` repository, and had to be custom developed from scratch. The temporal difference equation and weight updater are

$$\frac{1}{2}\sum_{i=t}^{T-1}\sum_{t=i}^{T-1}\lambda^{i-t}(v_{t+1}-f(\mathbf{x}_t))^2$$

and

$$w := w + \alpha\sum_{i=t}^{T-1}\nabla_w f(\mathbf{x}_t)c_t$$

`PyTorch` numerically computes gradients by the AutoGrad method, amortising the computation of calculating derivatives. These derivatives can be numerically computer after a forward pass, meaning that the derivatives for each parameter has already been computed. This simplifies the process to write custom loss and gradient descent weight

updaters, as backpropagation does not need to be reimplemented. The combined function for temporal difference learning is presented below.

```
def temporal_difference(self, boards, result):
      'temporal difference learning with boards and results'
      # boards is a list of chessboard positions in the game,
      # going chronologically forwards
      v = torch.cat([result] + [self(Variable(x)) for x in boards[::-1]])
      d = (v[:-1] - v[1:]).detach()
      v = v[:-1]

      # loss is calculated according to the temporal-difference equation
      trace = Variable(torch.cuda.FloatTensor(len(boards)).fill_(0.0))
      trace[0] = trace[0] + d[0]
      for t in range(1, len(boards)):
          trace[t] = trace[t] + d[t] + self.DISCOUNT_RATE * trace[t-1]
      epsilon = -(v * trace).sum()
      epsilon.backward()
      print "change of weights epsilon", epsilon.data[0]

      # p.grad.data is a numerically computed derivative
      for p in self.parameters():
          p.data -= self.LEARNING_RATE * p.grad.data

      # zero all the numerically computed derivatives
      for p in self.parameters():
          p.grad.data.zero_()
```

If `boards` was a list of leaf nodes of searches, the above function would update the weights of a network according to TD-Leaf(λ)

TD(λ) was also implemented for the `Keras` neural network. The `PyTorch` version was superior because of its speed, efficient allocation of GPU and CPU memory and lack of memory leaks. The memory leak in `Keras` had been thoroughly documented at the time of development with no solution. Numerical differentiation used in `PyTorch` utilises dynamic memory management. `Keras` is a wrapper for `TensorFlow`, which allocates memory statically, before the derivatives are calculated. As deep reinforcement learning is an online learning method, calls to run and train are dovetailed, and training depends on what values have just been run. Hence `TensorFlow` was creating a new static graph every time a game was played. Memory used for this graph was not freed after the derivatives were computed and the weights updated, thus consuming increasing memory. Resolving this issue is difficult in `Python` because there are no memory management operators. This complicated by the fact that `Keras` is a wrapper of `TensorFlow`, which

calls the `C++` functions to allocate and deallocate memory. [2] After considering all the above, neural networks in the project would be written in `PyTorch`.

3.2.4 Implementing a Random Agent

The random agent is an adversary which plays randomly selected moves. Pseudocode of the agent is given below.

Algorithm 6 Random Agent

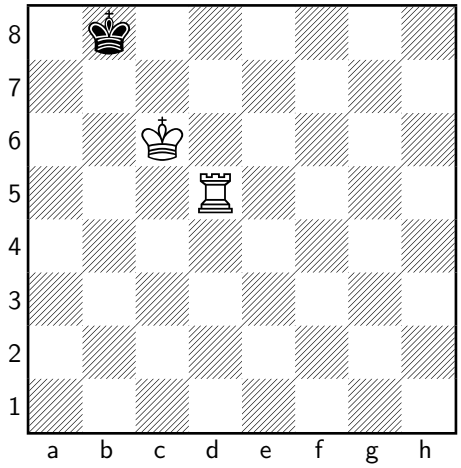
The function moveRandom returns a randomly generated move from chessboard position *position*.

```
1: function moveRandom(position)
2:     if noFurtherPossibleMoves() then
3:         return _
4:     max ← −∞
5:     maxMove
6:     for move in moveGenerator(position) do
7:         score ← random
8:         if score ⩾ max then
9:             max ← score
10:            maxMove ← move
11:    return maxMove
```

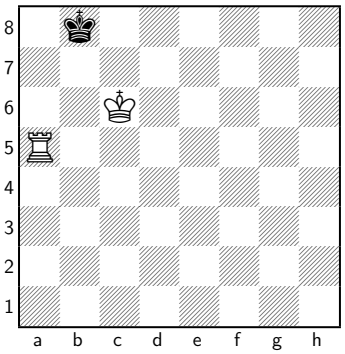
3.2.5 Convergence of Learning Algorithms Using an Endgame

To verify that the learning algorithms had been implemented correctly, agents were trained in a subproblem of chess known as an endgame. As its name suggests, an endgame occurs towards the end of a chess game, where only a few pieces remain. Checkmate for one player is usually a foregone conclusion depending on which pieces are present. The few pieces reduces the branching factor and the problem size, thus enabling learning to converge faster. This project used the king rook and king (KRvK) endgame because it was small enough such that a unique dominant strategy for checkmate exists. In contrast, a king, queen and king (KQvK) endgame has many strategies for checkmate. It is easier for the checkmate to occur by chance in KQvK, making it harder to distinguish between whether the agent has learnt a good strategy or has blundered into checkmate.

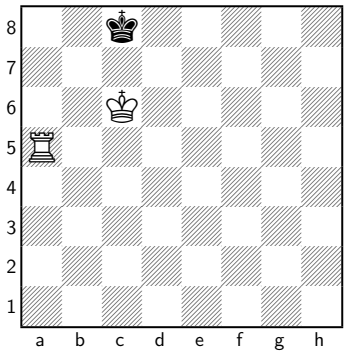
The starting point for the KRvK endgame is shown below, with White to move.



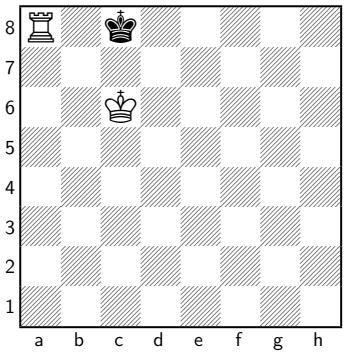
The three move dominant strategy yielding checkmate is described below. As this is the only way to generate checkmate in three moves, the optimality of an agent's behaviour can be determined by human inspection.



White moves their castle from $d5$ to $e5$ ...



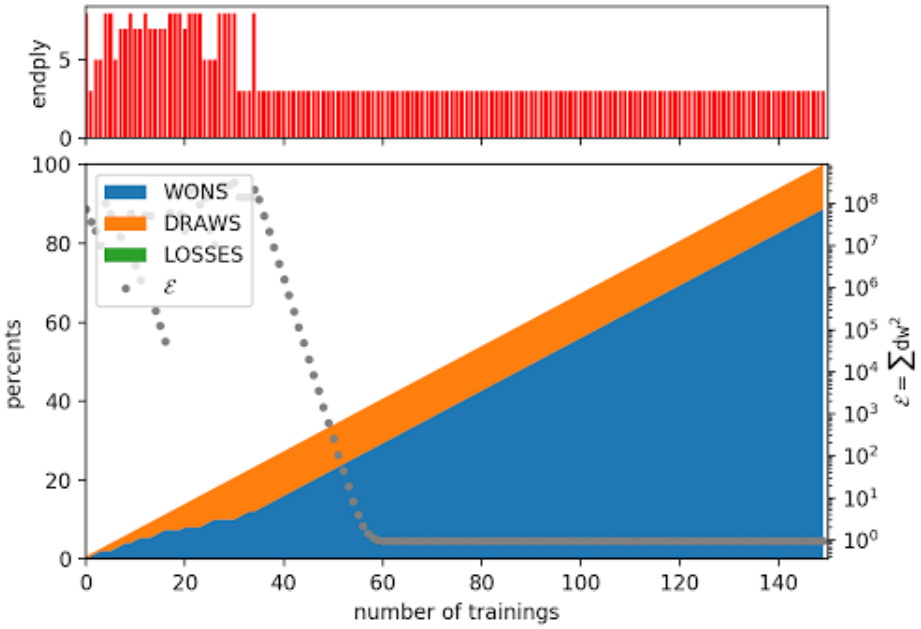
forcing black to move from $b8$ to $c8$ ...



white moves $e5$ to $a8$: Checkmate!

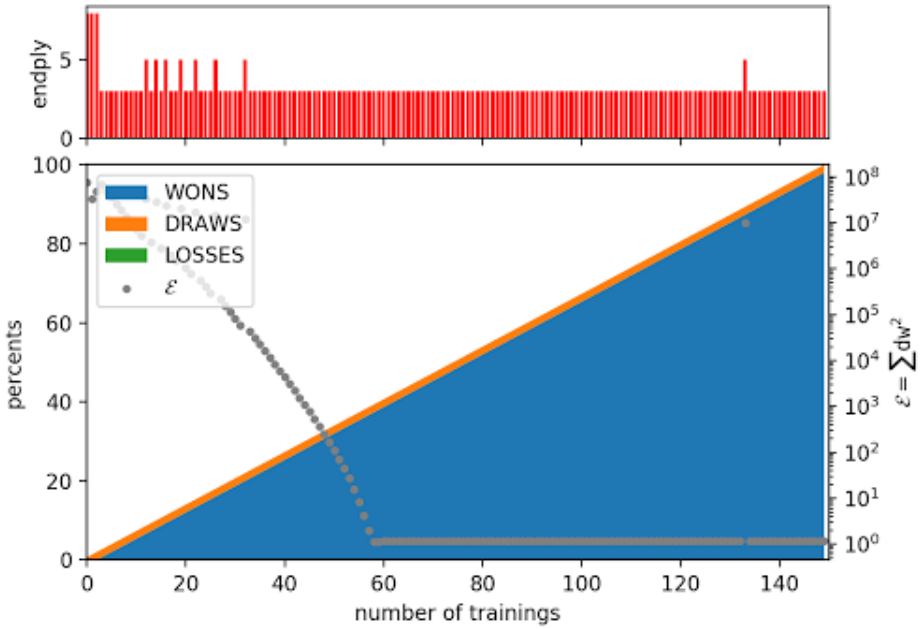
In addition, training speed was increased by using the small network architecture. The following graphs confirms that the algorithm has been implemented correctly. The upper subplot then reveals that the strategy which is being converged upon yields a checkmate in three moves – the strategy described above. The grey dots representing loss tends towards zero with the number of games increasing, implying that the strategy learnt by the agent is converging. The cumulative number of wins and draws show a mixture of both results at first, before the agent learns to win every time.

Training an Agent on a KRvK Endgame using TD(λ)



TD-Leaf(λ) was trained on the same subproblem as TD(λ). The results were similar, with TD-Leaf(λ) converging faster than TD(λ). Values for hyperparameters were found using trial and error. α and λ were set to 0.01 and 0.7.

Training an Agent on a KRvK Endgame using TD-Leaf(λ)



Loss gradually converged in TD-Leaf(λ), rather than fluctuating a lot in TD(λ). This implies that TD-Leaf(λ) is a stronger training algorithm capable of producing better

results. This is confirmed by the ratio of draws : wins being much lower in TD-Leaf(λ). An agent trained with TD-Leaf(λ) would be capable of learning better strategies in fewer time.

The loss in training converged after 60 training games for both algorithms because the same architecture and hyperparameters were used. Hence their convergence rates are expected to be at the same rate.

3.2.6 Preparing for Deployment

Since all the components had been implemented and had been used to solve a toy subproblem, the next step would be to deploy and train the algorithm on the entire game of chess. However, there were some limitations to the Sunfish engine. There was a lack of underpromotion and en passant, and the move generator was error-prone. Hence the Sunfish move generator was replaced with one from the `Python-Chess` library. Although not intended to deploy competitive chess engines, the `Python-Chess` was straightforward to use and faster than Sunfish.

An obvious problem with this engine is its speed of searching. Within a ten minute game, an agent could only search to 2 plys depth, a lot slower than the average value of 14 for many chess engines. This problem would be solved by deploying in `C++` in the next stage. Despite the slow speed, agent behaviour shared some similarity with other reinforcement learning projects.

The agent would often play the same sometimes suboptimal game again and again, instead of finding a better strategy. This is because the agent is not currently exploring the environment and is exploiting the same strategy. Balancing exploration and exploitation is a fundamental tradeoff in reinforcement learning. After the agent has explored the environment sufficiently, it can then learn and exploit the best strategy.

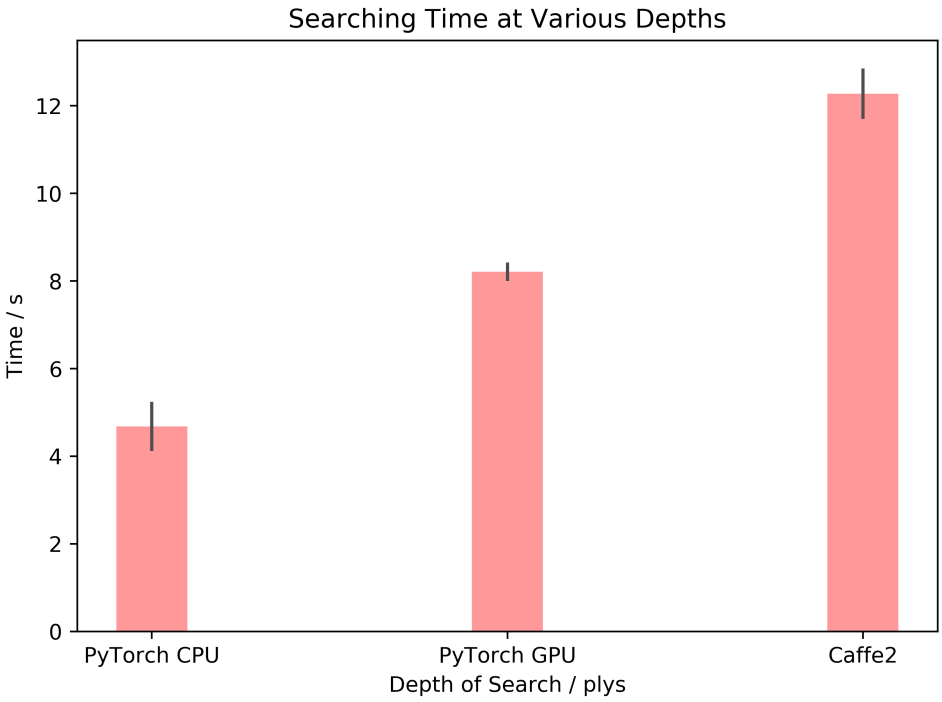
To increase exploration, draws by repeated moves or 50-plys were ignored in favour of playing towards checkmate or stalemate, such that board positions which would hitherto be unseen could be explored. In addition, the first four moves of each game would be randomly selected, ensuring that there would be a variety of games played.

3.3 Deployment in Stockfish and C++

Stockfish is one of the most powerful chess agents in the world, used and maintained by an active community. The source code is open-source and liberally commented, making Stockfish easy to adapt and this project accessible to a wide audience. The final codebase would modify Stockfish to train and use a neural network as an evaluation function.

In the previous phase, TD(λ) and TD-Leaf(λ) were implemented and verified. In addition, good board representations and network architectures were found and implemented. In order to utilise these features in as strong a chess agent as possible, the fastest possible configurations of searching algorithm, neural network library and move generator should be found.

Weights trained with `PyTorch` could be exported and run in `Caffe2`, a deployment-oriented submodule of `PyTorch`. Both libraries could be run with or without GPU acceleration. The time to evaluate 10000 board positions was calculated, producing the following graph. Ten repeated readings were generated with varying load, enabling a standard deviation to be found and plotted as error bars. Variable load was achieved by running a different number of invocations of the program in parallel.

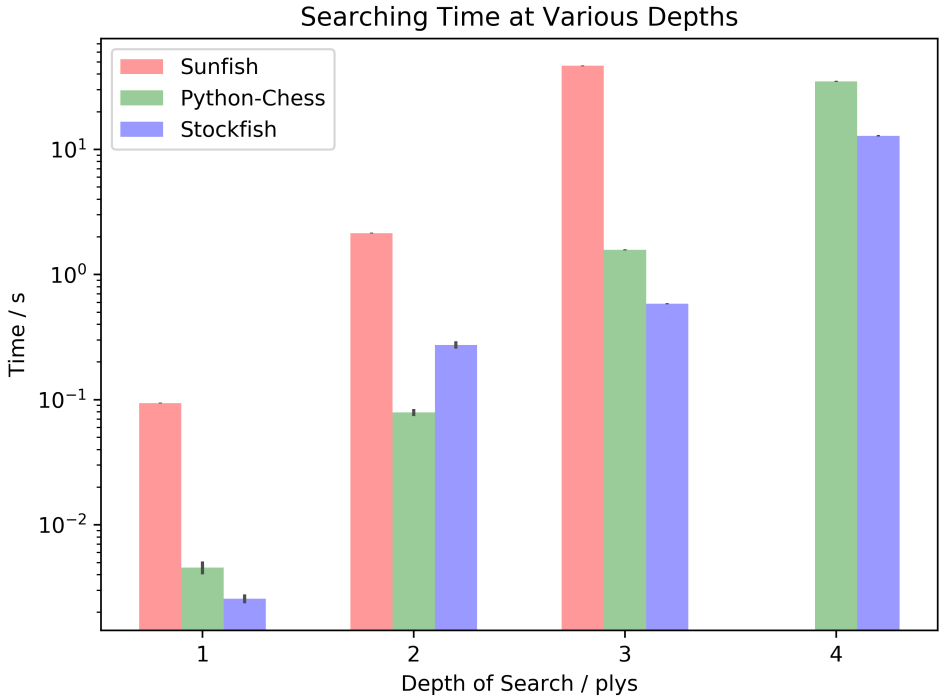


As the neural networks used are relatively small compared to state-of-the-art deep networks, the GPU is underutilised and is slower than running on just the CPU. Copying arrays to and from the GPU added an overhead to which slowed down inference. If the network was larger, this overhead would have been negligible compared to the speedup the GPU could provide due to computing many arithmetic operations in parallel.

The `Caffe2` framework has a structure similar to that of `TensorFlow`, as graphs were also statically allocated. This similarity could be the reason why `Caffe2`, like `TensorFlow` is slower than `PyTorch`.

An even faster implementation of neural networks would be to implement the neural networks from scratch in `C++`, as Giraffe does. This possible future research direction is left open to the interested reader.

To compare the speed of Stockfish's move generator with Sunfish and `Python-Chess`, searches of depth $0-5$ were timed for Negamax with no pruning, producing the following graph with a logarithmic y-axis. Reliability was ensured by repeating the timings ten times, again with varying CPU load. This enabled standard deviation error bars to be plotted on the following graph.



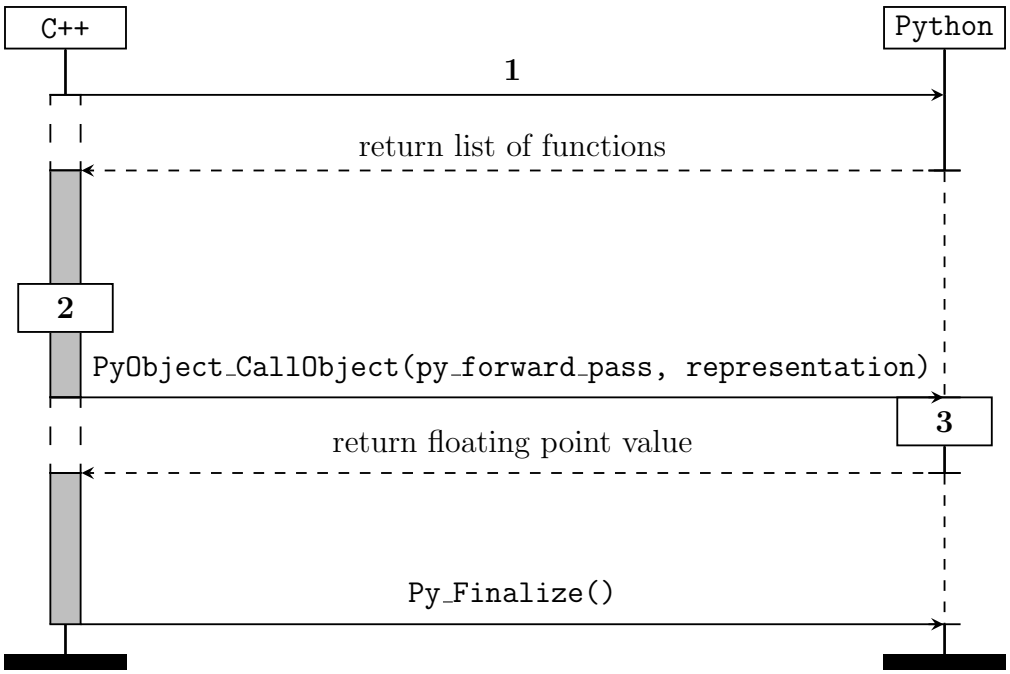
This confirmed the premise that for deeper searches, Stockfish was fastest and Sunfish slowest. Searching to a depth of 5 plys, or 4 in the case of Sunfish would have taken too much time. It is anticipated that the pruning heuristics would be able to speed up search even more. The scale factor of around 20 between times of successive plys confirm the fact that 20 is the average branching factor.

As the speed of Stockfish and `PyTorch` has thus been shown, the next step would be to implement a search algorithm in `C++` which can train and use the `PyTorch` neural network as an evaluation function. Now that the prototyping phase has ended, the final codebase can be written and deployed in in `C++` for speed and efficiency.

3.3.1 Inter-Process Communication between `Python` and `C++`

One of the first tasks was to embed `Python` in `C++` such that the neural network could be accessed with Inter-Process Communications (IPCs). Initially, a FEN string was passed from `C++` to `Python`. `Python` would convert it to the piecelist board representation, input it in the `PyTorch` neural network and return the output as a float. To improve the speed of this process, the board representation was created in `C++` and organised in `NumPy` memory format. A pointer to the array was passed to `Python`, which could pass the array to the network without any need for memory management. Correct implementation was ensured

by checking that the network returned the same floating point. The overall process is presented in the following Message Sequence Chart.



1. `Python` is embedded in `C++` when an interpreter session is started. The `PyTorch` neural network module is imported. A list of function pointers is returned to `C++`, enabling them to be accessed.

```
Py_Initialize();
PySys_SetArgv(argc, argv);
PyObject* module = PyString_FromString(‘‘<path/to/network/file>’’);
PyObject* valueNetwork = PyImport_Import(module);
PyObject* functions = PyModule_GetDict(valueNetwork);
PyObject* fp = PyDict_GetItemString(functions, ‘‘forward_pass’’)
```

2. When a position is ready for evaluation, it is converted into a `NumPy` array `input`. A pointer to its memory location is passed into the function `fp`, returning a reference to an output float. `Py_DECREF()` frees memory.

```
PyObject* output = PyObject_CallObject(fp, input);
double d = PyFloat_AsDouble(output);
Py_DECREF(output);
```

3. `fp` performs a forward pass of the `PyTorch` neural network.

IPCs for training the network follows the same framework, but with the weight update function instead of `forward_pass`. To verify that it was correctly implemented, the agent was trained to play the KRvK endgame and the optimal value observed.

3.3.2 Alpha-Beta Pruning Negamax with Transposition Tables

The Stockfish codebase contained an implementation of the Negamax algorithm, heavily optimised with 27 pruning heuristics. For simplicity, Negamax was reimplemented with alpha-beta pruning, move ordering and a transposition table. Alpha-beta pruning and move ordering was implemented from scratch, but Stockfish methods for accessing the transposition table, such as Zobrist hashing, was reused. To verify Negamax, Minimax without any pruning heuristics was also implemented. Stockfish's evaluation function was ported to `C++` and used to compare the output of Negamax and Minimax with the `Python` Negamax implementation.

This was the last key component to be implemented. Following its implementation, the agent could then be trained on the full chess game. The remainder of the implementation time was spent on tuning hyperparameters and investigating how different board representations and architectures affected engine performance. Due to the similar scores of bitboard and piece-list representations described in Section 2.3.1, neural networks utilising both representations were explored.

3.3.3 Training and Tuning Hyperparameters

The two parameters to be tuned were α and λ. Out of the two, training was more sensitive to α. When training on the KRvK endgame, the parameters were adjusted to observe their effect on the learning rate. The value of α fas first set as low as possible, before being gradually incremented to 0.01, the largest stable value of α where convergence of loss is observed. As training was not very sensitive to λ, a standard value of 0.7 was used.

The effectiveness of altering the hyperparameters was observed by playing 100 games against a random agent. Other hyperparameters which affected training was the search depth of Negamax and the number of plys each game lasted. While the number of plys in a game used for evaluation was 80, training was improved if the number of plys in a training game was more than 80. As agents became stronger, there would not be a winner within 80 plys, and the reward signal returned to the agents would be 0. By increasing the number of plys, the occurrence of -1 and 1 in the reward signal increased, enabling the agent to learn how to win and lose, rather than draw. After trial and error, the optimal depth for this value is 160 plys. Any more plys would only yield diminishing rewards in the face of an attenuated training time.

3.3.4 Piece Square Tables

Many chess engines such as Sunfish utilise piece square tables in their evaluation function. This is a vector of positive integers whose inner product with a bitboard chessboard representation returns a value. This computation can be achieved by running a single neuron with $bias = 0$ and a linear activation function. This network architecture is known as the piece square table.

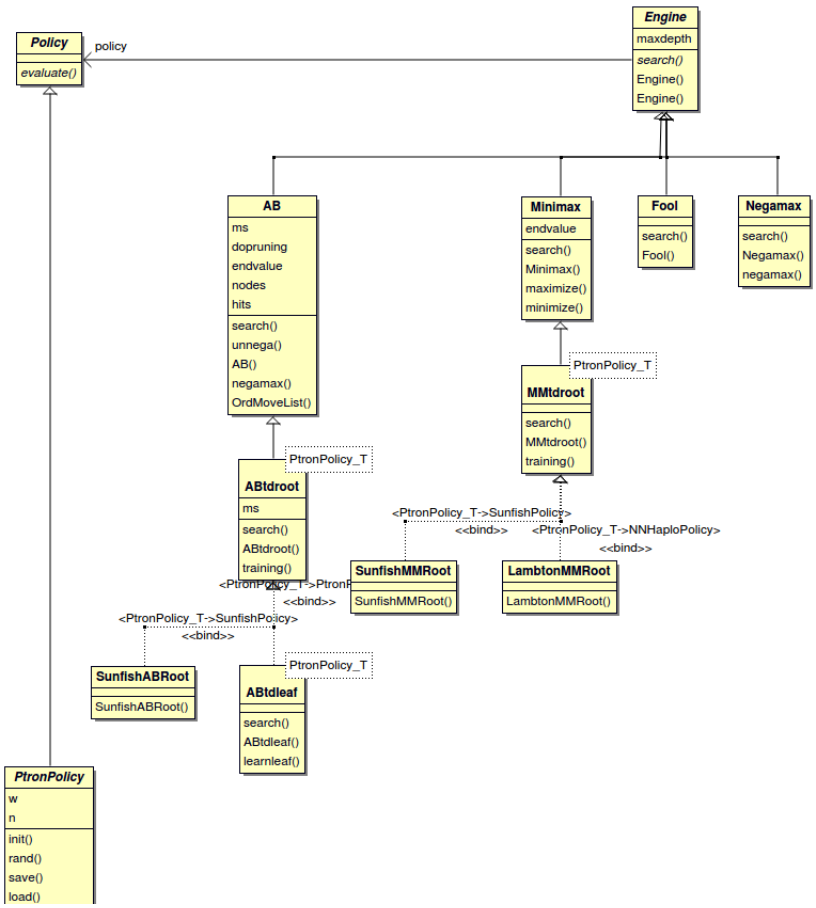
When finding a move, Sunfish evaluates the following expression using the evaluation function f. $\mathbf{x}^{-1}$ is a flipped board representation which rotates a board by 180^o and swaps white pieces for black and vice versa.

$$\begin{aligned} score &= f(\mathbf{x}) - f(\mathbf{x}^{-1}) \\ &= w \cdot \mathbf{x} - w \cdot \mathbf{x}^{-1} \\ &= w \cdot (\mathbf{x} - \mathbf{x}^{-1}) \end{aligned}$$

The above expression simplifies the expression to one inner product and a subtraction. As Sunfish uses a bitboard board representation, $\mathbf{x} - \mathbf{x}^{-1}$ is thus equivalent to the antisymmetric bitboard described in Sectino 3.2.1. Thus the piece square table architecture consisting of a linear combination of an antisymmetric bitboard and weight vector was implemented in `C++` and trained with TD(λ) and TD-Leaf(λ).

3.3.5 Overview of Final Codebase

The implemented components was a chess engine, reinforcement learning algorithms and an agent which played randomly selected moves. A UML class diagram of the final codebase is shown below.

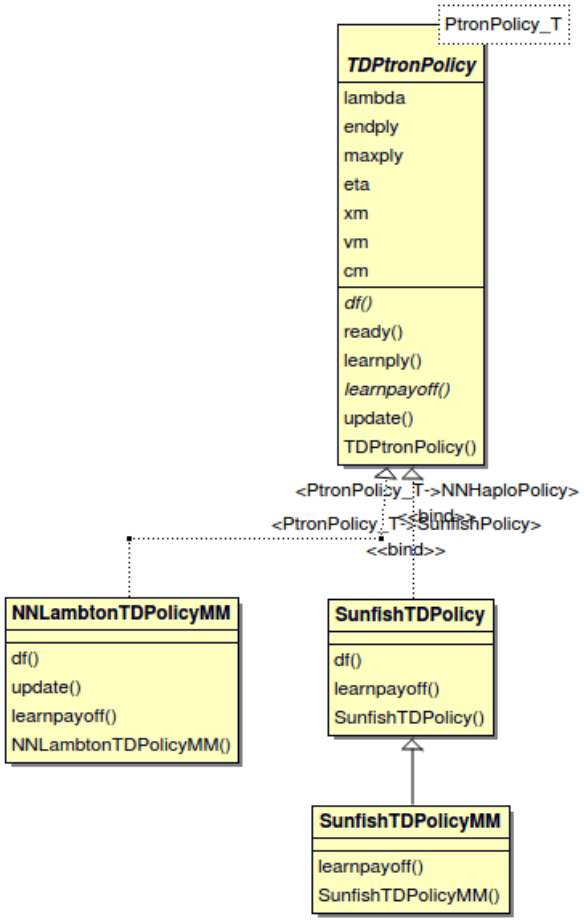


The main class in the final codebase is Policy. This class is used to find the best move given a chessboard position. It has an Engine which is responsible for searching and evaluation. PtronPolicy is the simplest policy of all, and is an implementation of a perceptron in `C++`.

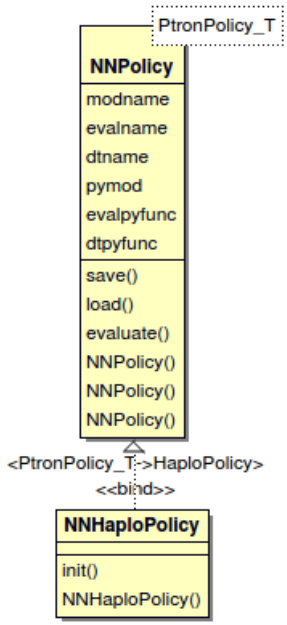
Engine is a class which can be AB (alpha-beta), Minimax, Negamax or Fool (an agent playing a random move). As AB is used in training, it can be ABtdroot (TD(λ)) or ABtdleaf (TD-Leaf(λ)), depending on which training algorithm is used. This, as described, utilised the SunfishABLeaf (Sunfish piece square tables) as an evaluation function.

For debugging, Minimax was used to verify that AB gave correct evaluations. It has two interchangeable policies, Sunfish and Lambton MMRoot (standing for MiniMaxRoot). SunfishMMRoot was another implementation of the Sunfish piece square tables, whereas lambton is the Medium Network written in `PyTorch`, so called because The Lambton Worm was a snake (`Python` is a snake) which lived in water. There has been a longstanding tradition of chess engines named after fish.

The following template heirarchy illustrates how various policies were derived from a perceptron.



Finally, the NNpolicy (Neural Network Policy) is a supertemplate of NNHaploPolicy, used to interface with the `PyTorch` neural network. This policy is so named because the Haplochromine fishes are bright and luminous, like torches.



3.4 Conclusion

The chapter presents an insight into the project's implementation. Following a fruitful exploration, experiments regarding board representation, network architecture and training on a chess subproblem were undertaken in stage 2. In stage 3, Stockfish was modified to use a `PyTorch` neural network with a custom loss function and weight updater.

Chapter 4

Evaluation

In the previous chapter, evaluation of some components such as the board representation, network architecture and hyperparameter values were undertaken during implementation, such that they can be used to guide the implementation process. This chapter evaluates of the final codebase and trained agent, and explains why the examinable objective has been achieved. First, an empirical evaluation of the examinable objective is presented, before its validity and reproducibility is analysed in summative evaluation.

4.1 Objective

The examinable objective was declared in the form of a null and alternate hypothesis. The goal of the project was to show that there is no reason to reject the alternate hypothesis to 95 % significance.

Null Hypothesis: Within a chess game of 80 plys, there is no significant difference between a trained agent and one that plays randomly generated moves.

Alternate Hypothesis: Within a chess game of 80 plys, a trained agent performs better than one that plays randomly generated moves.

To compute significant difference, the win ratio of a trained agent to a random agent was compared against two random agents playing against each other.

4.1.1 Win Ratio of Agent Trained by TD(λ)

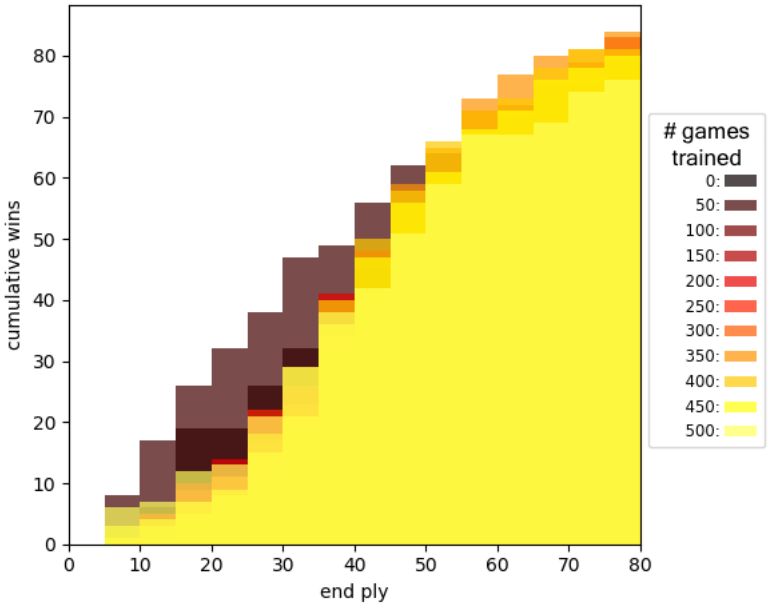
When two random agents played against each other, the win ratio was consistently 0 %, because it was very unlikely that a random walk leads to checkmate. This number was confirmed by counting the number of wins in 10 batches of 100 80-ply games.

A piece square table architecture of an evaluation function was trained on 500 games of self-play using TD(λ). Hyperparemeters used were derived from experimentation as mentioned in the previous chapter.

Hyperparameter	Value
α	0.01
λ	0.7
Negamax Search Depth	4
Training Ply Cutoff	160
Initial Random Plys	4

After every 50 games of self-play, 100 games were played against the random agent. The graph shows the cumulative wins in these 100 games vs the ply at which the win was achieved. The win ratio for a trained agent against a random agent could be as high as 84 %.

Cumulative Win % of Agent Trained by TD(λ) vs End Ply



The emerging sigmoidal shape of the cumulative wins graphs indicates that the agent is overfitting to playing against itself. In supervised learning with a training and evaluation set, overfitting means the machine learning model is learning features locally specific to the training set which are not present in the underlying distribution. This would reduce the accuracy of evaluation. Overfitting is particularly prevalent in large neural networks with plenty of neurons.

Although the piece square table architecture – a single neuron – would not overfit in supervised learning, this can occur in reinforcement learning. Instead of learning general strategies, the agent is overfits and learns how to play against itself. Preventing overfitting in reinforcement learning is more challenging than in supervised learning because the online nature of reinforcement learning algorithms generate data depending on the behaviour of the agent, rather than having a constant underlying distribution. The emerging sigmoidal shape is indicative of overfitting because the agent learns how to play more effectively towards 60 – 80 plys, and forgets how to win from 10 – 40 plys. The ideal

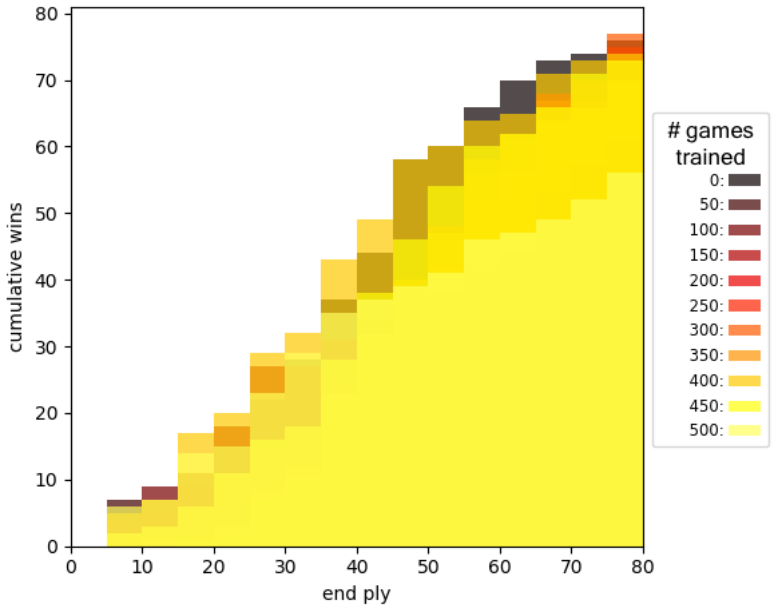
shape of the graph is linear with a gradient of 1, because the agent should be equally likely to win at any ply, given that the opponent is a random agent and will not change strategy based on the point at the game. This shows that instead of learning how to play the same game with the same moves or the same strategies, the agent is capable of identifying a checkmate situation and exploiting it.

By averaging the values of 10 batches of 100 games against a random agent, 95 % confidence intervals were calculated.

$$\overline{x} \pm 1.96 \times \frac{\sigma}{\sqrt{n}} = 73 \text{ to } 79.8\%$$

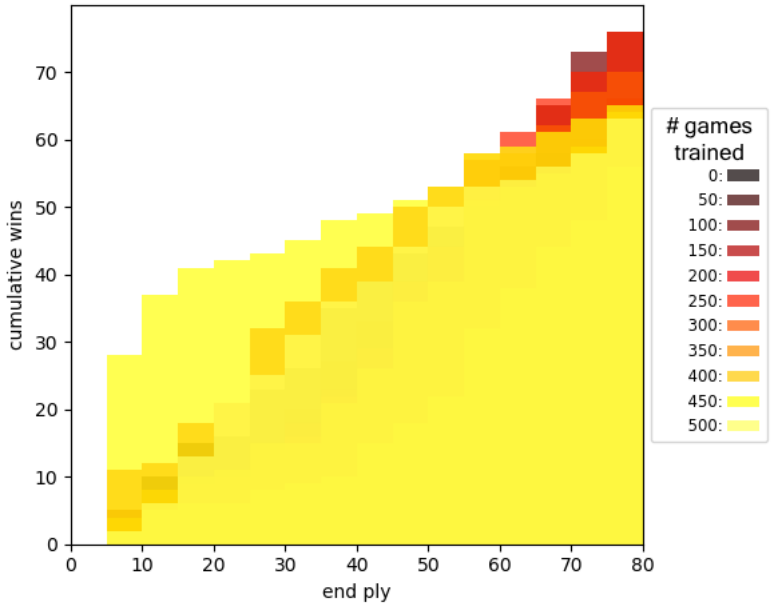
As 0 was not in the range of this interval, there is no reason to reject the alternate hypothesis. Hence the criteria of the project has been met. When the Negamax search depth was reduced to 2, the peak win ratio was reduced from 84 % to 76 %, as illustrated in the following graph.

Cumulative Win % of Agent Trained by TD(λ) vs End Ply



This shows that the depth of search only marginally affects the overall performance of the agent when trained by TD(λ). An extreme of overfitting is presented in the following graph, generated from the same training conditions as the graph above but with a different weight initialisation.

Cumulative Win % of Agent Trained by TD(λ) vs End Ply

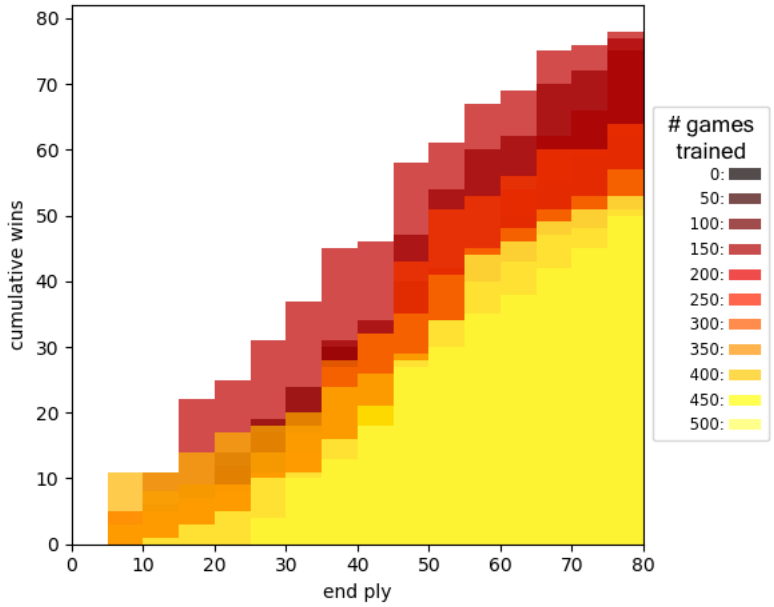


In this graph, the network overfits and learns strategies which are more effective when the plys are $0 - 40$, at the expense of learning strategies later on in the game. This is the opposite of the sigmoidal cumulative curve shown in previous graphs.

4.1.2 Win Ratio of Agent Trained by TD-Leaf(λ)

The agent was trained for 500 games of self-play using TD-Leaf(λ). The following graph shows the cumulative wins in the batch of 100 games vs the ply at which the win was achieved, for every 50 games of self-play.

Cumulative Win % of Agent Trained by TD-Leaf(λ) vs End Ply

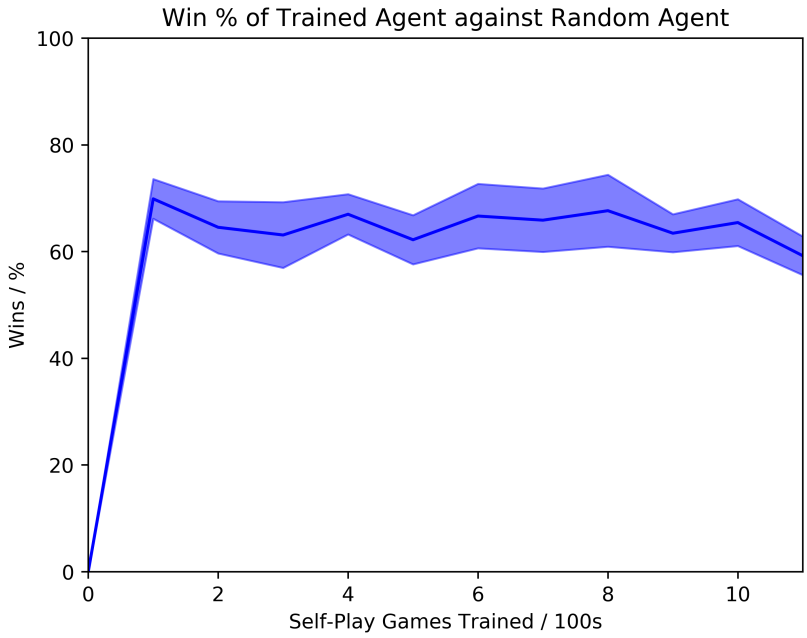


As the maximum win ratio of 75 % was rather similar to the value generated by training with TD(λ), this could imply that training of this network architecture cannot be improved any further. To improve the power and capacity of the value network, a larger network should be used. Overfitting was more severe in TD-Leaf(λ) than TD(λ), implying that TD-Leaf(λ) is a stronger learning algorithm.

However, results obtained through training the Medium Network using TD(λ) and TD-Leaf(λ) yielded similar results to those presented above. It is possible that some bugs remain, preventing the network from training efficiently. Due to time constraints, these had not been fully debugged.

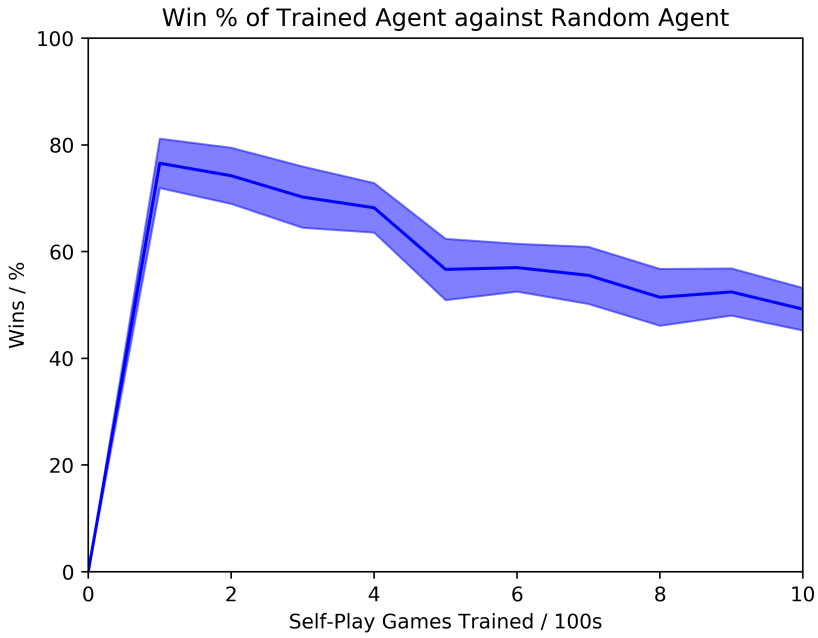
4.1.3 Reproducibility

To show that the results were reproducible, the weights of the value network were initialised to different random values. The agent was then trained for 1100 games, at which point the win ratio started to decrease because the neural network was overfitting.



Overfitting occurred because the agent is becoming better at playing against itself but not against a random agent. This can be combatted by increasing exploration to vary the games the network is being trained on. During self-play, the first four moves were randomly selected, enabling the agent to explore various games. By increasing this value, the agent could be trained with a greater amount of exploration.

The same graph for an agent trained with TD-Leaf(λ) is presented below.



As TD-Leaf(λ) was a more powerful training algorithm, it overfitted more quickly. Hence the decline in win rate is accentuated more for the graph produced by TD-Leaf(λ).

4.2 Summary

This chapter shows that the objective has been successfully met, and confirms initial results in the Implementation chapter that TD-Leaf(λ) is a stronger training algorithm than TD(λ). Evaluation of individual components of the chess engine have been presented in the Implementation chapter, because results from these evaluations shaped the structure of the resultant chess engine.

Chapter 5

Conclusion

Overall, this dissertation describes how a chess engine can be trained using deep reinforcement learning. A novel board representation only encoding piece coordinates was used, as well as a new network architecture. The results were a success, fulfilling the requirements of the examinable objective. In addition, the agent was trained by self-play with minimal human knowledge, making this project one of the few chess agents trained in this way. This project also showed that it was possible to train chess agents on commercial hardware.

Although the agent did not attain superhuman performance, it learned the relative value of pieces and was significantly stronger than a random agent, fulfilling the core objective. With a longer training time and larger network, the agent's strength would increase, perhaps leading to better performances.

5.1 Lessons Learned

The most challenging component of the project was getting the search algorithm to run at an adequate speed. This was achieved by adapting the move generator in `Stockfish` in a searching algorithm. This reimplemented the search algorithm which had already been written in `Python`. Time could have been saved by implementing the search algorithm directly in `C++` without using `Python`.

Although easy to implement, cascade correlation took a long time to run. As the strength of the final agent did not depend much on the network architecture, this step would not have been necessary.

5.2 Further Research Directions

An obvious next step would be to improve the efficiency of the final codebase and removing any lingering bugs such that deeper searching could be achieved with a larger neural network, increasing the strength of the agent.

Compared to other chess engines, the evaluation function was implemented in `Python`. To increase speed, a possible research direction would be to reimplement neural networks in `C++`, reducing the need for IPC and increasing speed. In addition, multithreading could be used in searching, increasing the number of board positions evaluated per game.

Deep learning has developed other loss functions and weight updaters. Rather than using the minimum squared error and stochastic gradient descent, the impact of using other methods could be investigated.

5.3 Summary

Overall, learning about deep reinforcement learning and implementing its algorithms in chess has been a challenging but enjoyable experience. By building on the foundations of this project, there a lot of potential ideas for future research directions.