Deep Reinforcement Learning

A case study of AlphaZero

Fredrik Mattisson
Abstract

Deep Reinforcement Learning

Fredrik Mattisson

Using deep neural networks for reinforcement learning has proven very successful, as demonstrated by the AlphaZero algorithm developed by DeepMind in 2018. This algorithm is capable of mastering two-player zero-sum board games entirely by playing against itself. However, a drawback of deep learning in general is the immense computational cost associated with training deep neural networks, and AlphaZero is certainly no exception; an absurd amount of compute power was used by DeepMind to produce their results. This thesis project is a first step towards investigating whether DeepMind’s approach to reinforcement learning could somehow be made more computationally efficient. We implement the AlphaZero algorithm in a modular fashion, so as to facilitate experimentation with its constituent parts, and also attempt to better understand what the neural network learns by visualizing it. The thesis gives an explanation of the algorithm and its theoretical foundations, how it was implemented, and present some preliminary results of training it on the game of Go on a 5 by 5 board. The agents performance was primarily evaluated against basic Monte Carlo tree search, which yielded a win-rate of about 50% with the latter using 5 times as many simulations per move. Although training was only conducted for a short period of time on commodity hardware, the results and empirical analysis indicate that the algorithm managed to learn at least some rudimentary aspects of the game. However, since little further improvement was seen asymptotically in these experiments, the configuration was likely sub-optimal.

Handledare: Arne Andersson
Amnesgranskare: Per Matsson
Examinator: Johannes Borgström
IT 21 074
Tryckt av: Reprocentralen ITC
1. Introduction

The field of artificial intelligence has seen very rapid advancement and many seminal results over the last decade. The availability of huge amounts of data and increasingly powerful hardware has facilitated research into deep learning in particular. That is, machine learning algorithms centered around deep neural networks. Big tech companies have been pouring funds into the development of such algorithms; in 2014 a research team at Facebook achieved 97.35% accuracy in face recognition [12], and in 2016 the AlphaGo algorithm [3], developed at Google’s AI research company DeepMind, became the first computer program to beat a professional human player in a standard competitive game of Go.

While the deep learning approach has certainly yielded impressive results in many applications, it often results in models that require tremendous amounts of computational resources in order to converge within a reasonable time frame. For example, the generative natural language model GPT-3 released by OpenAI in 2020 uses a neural network with some 175 billion parameters and required approximately $3.14 \cdot 10^{23}$ floating point operations (FLOP) to train [2]. To put that into perspective, Nvidia’s A100 compute card built on their 2020 Ampere architecture is advertised as being capable of a theoretical peak performance of $156 \cdot 10^{12}$ single precision (32-bit) FLOP/s (156 TFLOPS). Hence the GPT-3 training process would theoretically take

$$\frac{3.14 \cdot 10^{23}}{156 \cdot 10^{12} \cdot 3600 \cdot 24 \cdot 365} \approx 64$$

years on a single such unit. Assuming perfectly linear speedup, one would need about 3300 units to train the model in one week. In addition, storing the 175 billion parameters values require some $4 \cdot 175 = 700$GB of memory, assuming single precision.

A similar analysis could be made in the case of AlphaZero [6], which is a generalization of AlphaGo’s successor AlphaGo Zero [4]. The algorithm is capable of learning many different board games, entirely from scratch by playing against itself. In producing the results published in [6], 5000 first generation (as of 2018) tensor processing units (TPUs) were used for inference during self-play in the training process, which for the game of Go took 13 days.

The absurd amount of compute power required to achieve these results begs the question of whether the recent development in deep learning is sustainable; perhaps more efficient methods will be required as deep neural networks approach the limit of scalability. It can also be argued that the field of deep learning lacks a robust theoretical foundation; while the workings of the individual components of deep neural networks are well understood, the emergent behaviour of entire systems are less so. Deep learning systems can be very brittle, sometimes failing spectacularly for reasons that are difficult to understand, as exemplified in [7]. The understanding of these systems is to a large extent based on intuition, albeit intuition that often aligns well with the results.

This thesis focuses on the application of machine learning to board games, with DeepMind’s work [3, 4, 6] as a starting point. Specifically, it aims to shed some light on how and why AlphaZero works. The project is a precursor to experimentation with alternatives to deep learning in this domain. We develop a Java framework for this purpose and implement AlphaZero in such a way that its constituents can be replaced. Some rudimentary visualization tools are also developed for the purpose of studying the algorithms behaviour in self-play. The reason for focusing on board games is that this domain lends itself very well to model based reinforcement learning (see Section 2.2), and DeepMind’s algorithms are interesting not only because of the groundbreaking results, but also because they take an interesting approach to the reinforcement learning problem, using a
combination of deep convolutional neural networks and Monte Carlo tree search. Studying these algorithms, DeepMind’s approach to reinforcement learning, and the role that deep learning plays in it might lend some insight that could prove useful in developing other techniques. Although it is certainly not feasible to fully reproduce DeepMind’s results within the scope of this project, we present some experimental results for the game of Go on a reduced (5 x 5) board size.

2. Background

2.1. Machine learning and neural networks In a nutshell, machine learning (ML) is about writing programs that can “learn” to respond to complex patterns or features in the input, increasing the accuracy of the response over time based on some type of feedback. This learning process is often similar to conventional optimization in the sense that the algorithm is trying to minimize (or maximize) some function of its parameters, but a key difference is that ML usually aims to generalize across a larger set of problem instances rather than find a true optimum for a particular instance. ML has numerous applications, for example computer vision, image, text and speech recognition, autonomous vehicles and medical diagnosis.

Broadly speaking, ML can be subdivided into three categories; Supervised learning, where a large set of labelled example inputs is used during the learning process, and the algorithm is trying to minimize the discrepancy between its output and the expected output; Unsupervised learning, where the algorithm tries to find patterns and structure in the input data without guidance from predefined labels; Reinforcement learning, which will be discussed in Section 2.2.

At the heart of many ML algorithms are Neural networks, a class of data structures that offer a generic way of approximating arbitrary multivariate functions. As the name suggests, the inspiration comes from the human brain and its intricate network of neurons. Different stimuli cause different neurons in the brain to fire, and repeated stimulus can strengthen connections between neurons. A neural network (NN) tries to model this behaviour. In its most basic form, a NN consists of an input layer, a hidden layer, and an output layer. Each layer is comprised of a number of neurons, and is fully interconnected with the subsequent layer. With each connection is associated a weight, and the output of a neuron is a nonlinear activation function that takes as input the weighted sum of the outputs from the preceding layer. Figure 1 illustrates an example NN, depicted as a weighted directed graph:

![Diagram of a neural network](image)

Figure 1: Depiction of a neural network with one hidden layer and a single output.

The output of the $h_1$ neuron in the hidden layer in Figure 1 would be

$$h_1 = \sigma(w_1x_1 + w_2x_2),$$

where $\sigma$ is the activation function. Using this example network for supervised learning could proceed as follows: The weights are initialized to random values, and each example input vector $(x_1, x_2)$ is labelled with a desired output $y$. Following each evaluation, a loss function $E(w_1, \ldots, w_6)$ is computed, whose value is to be minimized. For example, we might have $E(w_1, \ldots, w_6) = (y - \hat{y})^2$. The loss function is differentiated with respect to each weight, and the weights are updated according to the value of the partial derivative at that point:

$$w_i \leftarrow w_i - \gamma \frac{\partial E}{\partial w_i} = w_i - 2\gamma(y - \hat{y}) \frac{\partial \hat{y}}{\partial w_i},$$
where $\gamma$ is a parameter called learning rate. Such parameters are typically referred to as hyperparameters, whereas the weights in the network are often called just parameters.

Each neuron would typically also have an additional parameter called a bias added to its input. This can improve accuracy by effectively allowing for translation of the neuron’s output. The parameter optimization technique described here is aptly named gradient descent, as it follows the negative gradient of the loss function toward a minimum, thus fitting the function computed by the network to the example data.

There is of course a multitude of different NN architectures, activation and loss functions, parameter optimization schemes, etc. The loss function in the example is a simple statistical measure called the mean squared error (MSE), applied here to a single data point. Typically, a batch consisting of many training examples would be evaluated, and the loss averaged over the batch. Batching the data gives a less noisy estimate of the error, and can improve numerical stability. Since the computations are typically performed on a GPU, it also helps in mitigating the overhead of data transfer. Section 2.3 describes a particular architecture called a convolutional neural network.

### 2.2. Reinforcement learning

Reinforcement learning (RL) is motivated by a simple observation; humans learn from experience. We observe the situation we are in, think about how we might improve it, make decisions based on that, and evaluate the outcome. If the outcome is not the desired one, we learn from our mistakes and try a different strategy.

RL typically models this process of learning by doing as a Markov decision process (MDP) [11, ch. 3]. A MDP is a discrete stochastic process in which an agent interacts with an environment. The environment is defined by a state space $\mathcal{S}$ and an action space $\mathcal{A}$. At each time step $t$, the agent chooses an action $a_t \in \mathcal{A}$, causing a state transition from the current state $s_t \in \mathcal{S}$ to a new state $s_{t+1} \in \mathcal{S}$ with probability $\Pr(s_{t+1} | a_t, s_t)$. The agent then receives a reward $r_t = r(s_t, a_t, s_{t+1}) \in \mathbb{R}$. The goal of the agent is to learn a policy $\pi : \mathcal{A} \times \mathcal{S} \to [0, 1]$ that maximizes a state value function $V_\pi : \mathcal{S} \to \mathbb{R}$. The policy defines a probability distribution over actions; $\pi(a, s) = \Pr(a | s)$, the probability of choosing action $a$ in state $s$. The state value function is typically defined as the expected total reward from acting according to the current policy:

$$V_\pi(s) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s \right],$$

where $\gamma \in (0, 1)$ is a parameter that exponentially reduces the influence of expected future rewards. Similarly, an action value function $Q : \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is defined as

$$Q_\pi(a, s) = \mathbb{E}_\pi \left[ r_t + \gamma V_\pi(s_{t+1}) \mid s_t = s, a_t = a \right],$$

which gives the expected total reward of taking action $a$ in state $s$ and then acting according to $\pi$ from the resulting state.

Note that the policy optimization problem has optimal substructure; if $\pi^*$ is an optimal policy, then

$$V_{\pi^*}(s) = \operatorname{argmax}_a Q_{\pi^*}(a, s).$$

That is, making locally optimal decisions will maximize the state value function. It follows, using the definition

$$Q_\pi(\pi', s) = \sum_{a \in \mathcal{A}} \pi'(a, s) Q_\pi(a, s),$$

that if $\pi, \pi'$ are policies such that

$$Q_\pi(\pi', s) \geq V_\pi(s)$$

for all $s \in \mathcal{S}$, then $V_{\pi'}(s) \geq V_{\pi}(s)$ for all $s$. That is, if the expectation of acting according to $\pi'$ is at least as good as acting according to $\pi$ in each state, then $\pi'$ is at least as strong as $\pi$. This is known as the policy improvement theorem [11, p. 95].
There are many different approaches to optimizing the policy. Which methods are applicable and suitable will depend on a number of factors, but one important consideration is whether or not there exists a model of the environment. That is, if the probability distribution for state transitions and the reward function are either known or can be estimated to some degree of accuracy. An algorithm in which the agent makes use of a model to simulate interaction with the environment are called model-based, as opposed to model-free. A classic policy optimization algorithm, which in various forms can be applied in both model-based and model-free settings, is policy iteration. It involves iterating two steps; the policy is evaluated by approximating either $V_\pi$ or $Q_\pi$, and then improved using this approximation. Repeating these two steps yields a sequence of policies $\pi_0, \pi_1, \ldots, \pi_n$ that, by the policy improvement theorem, converges to $\pi^*$ as $n \to \infty$. Note that although the latter is predicated on the policy evaluation theorem, convergence can still be guaranteed in certain settings even when the value function is only approximated [11, p. 100].

In a model-based setting where both the reward function and state transition probabilities are known, policy iteration can be implemented in a straight-forward manner using dynamic programming, by starting with a random policy and value function, and then using the recurrence

$$V_{i+1}(s) = \mathbb{E}_\pi [r_t + \gamma V_i(s_{t+1}) | s_t = s]$$

to compute a sequence of approximations $V_0, V_1, \ldots, V_n$ that converges to $V_\pi$ as $n$ goes to infinity. This constitutes the policy evaluation step, and an improved policy can then be derived using $V_n$, for example by making the new policy greedy with respect to $Q_n$. However, this approach is not feasible for large problems, since the policy evaluation requires traversing the entire state space $n$ times. Assuming a model is available that is at least capable of generating sample state transitions and rewards, Monte Carlo simulation can instead be used to approximate the action value function. In general, Monte Carlo simulation refers to a method of approximating the expected outcome of a stochastic process based on random samples, leveraging the fact that if $n$ randomly sampled outcomes are averaged, then by the law of large numbers, the average approaches the expected value as $n \to \infty$.

Monte Carlo policy iteration [11, ch. 5] is best used when $S$ contains terminal states, i.e. when there is a natural notion of when the task is completed, be it successfully or not. In this case, we can define an episode as an ordered sequence of state-action pairs generated by the agents simulated interaction with the environment, ending in a terminal state $s_T$. If $E$ is a set of episodes generated by acting according to the current policy $\pi$, then $Q_\pi(a, s)$ is approximated as the average over total rewards observed following each occurrence of $(s, a)$ in $E$:

$$Q_\pi(a, s) \approx \frac{1}{N(a, s)} \sum_{e \in E} \sum_{t = u}^{T-1} \gamma^{t-u} r_t ,$$

where $N(a, s)$ is the number of occurrences of $(a, s)$ in $E$ and $u$ denotes the time step at which $(a, s)$ occurs in $e$. This approximation converges to $Q_\pi$ as $|E| \to \infty$, hence it can be used to generate a new policy $\pi'$ where $\Pr(a|s) \propto Q_{\pi'}(a, s)$.

### 2.3. Convolutional neural networks

A convolution, in the context of image processing, is an operation where each pixel value is replaced by a weighted sum of the values in the $n \times n$ submatrix centered at that pixel. The weights are given by a $n \times n$ matrix, often called a kernel or a filter. The filter is moved across the image such that it is centered over each successive pixel, like a typewriter. At each point, the new pixel value is the sum over the elementwise product of the filter with the part of the image on which it is superimposed. Mathematically, if $f(x, y)$ are the pixel values of an image and $\omega$ is a $(2m+1) \times (2m+1)$ filter, then the convolution
\( \omega * f \) can be expressed as
\[
(\omega * f)(x, y) = \sum_{dx = -m}^{m} \sum_{dy = -m}^{m} \omega(dx, dy) f(x + dx, y + dy). 
\]

This technique can be used to apply various transformations such as blur, sharpening, and edge detection. For example, if
\[
\omega = \begin{bmatrix}
-1 & -1 & -1 \\
-1 & 8 & -1 \\
-1 & -1 & -1 
\end{bmatrix},
\]
then
\[
\omega * f = \text{Figure 2: Example of convolution with an edge detection filter.} \quad \text{Image source: Wikipedia}
\]

There are different ways of dealing with border regions. The image can for example be padded with zeros to accommodate the filter. If padding is not introduced, then the image is effectively cropped in the process of convolution. The step size, or stride, with which the filter is moved across the image can also vary. A stride greater than one effectively down-samples the image in the process.

In a convolutional neural network (CNN), each convolutional layer convolves a set of filters with the input, producing as output a set of feature maps. The entries in the filters are network parameters. The idea is that the network learns to detect increasingly complex features in the input. The entries in the feature maps are passed through an activation function to introduce non-linearity, since the convolutions are linear operations. A class of activation functions commonly used in CNNs is rectified linear units (ReLU), which suppress negative values. An example of a simple but effective ReLU function is \( f(x) = \max(0, x) \).

The network architecture typically includes other layers as well, an important example being pooling layers, also called downsampling or subsampling layers. A pooling layer downsamples the feature maps by partitioning them into equally sized regions and taking for example the maximum or average value from each region. This downsampling reduces the computation and memory requirements of subsequent layers, the idea being that the exact location of a feature is less important than the locations of the features relative to each other.

The sizes of the filters and pooling regions, and the strides used are examples of hyper-parameters of a CNN.

2.4. Monte Carlo tree search

Monte Carlo tree search (MCTS) is a stochastic search algorithm for decision processes that uses heuristics and Monte Carlo simulations to explore the search space. Different variants of MCTS are widely used in domains where exhaustive search is infeasible, which for example is the case in many games. If in each state there are \( n \) possible actions that could be taken, then exhaustively searching \( k \) decisions ahead requires evaluation of \( n^k \) states. \( n \) is called the branching factor of the decision tree. The idea of MCTS is to selectively explore branches of the search tree based on statistics acquired through the simulations. In the case of a game, that could be something like the observed win ratio for that subtree.

The basic MCTS algorithm iterates three steps; Selection – actions are heuristically selected based on statistics gathered thus far, until a previously unseen state is encountered; Rollout – actions are selected randomly until a terminal state is reached (or some other stopping criteria are met); Backpropagation – the terminal state is evaluated and the result is propagated back up the tree, updating the statistics of all the nodes on the path. We will refer to one execution of these steps as one simulation.
The idea is that the more simulations are run, the more accurate the information contained in the nodes will be, which guides the search toward exploring the most promising branches. After running a number of simulations, a decision can be made based on the number of times each child of the current root node has been visited.

Some heuristic for balancing exploitation of the information currently available and exploration of new possibilities is typically incorporated in the selection step. A common approach is Upper Confidence bounds applied to Trees (UCT) [8], where move selection is made according to

$$\arg\max_a \left( \bar{Q}(a, s) + \sqrt{\frac{2 \ln(N(s))}{N(a, s)}} \right),$$

where $\bar{Q}(a, s)$ is the mean value of the subtree reached by action $a$ in state $s$, $N(s)$ is the number of times $s$ has been visited, and $N(a, s)$ the number of times $a$ has been selected in $s$.

2.5. AlphaZero As was mentioned in Section 1, AlphaZero (AZ) uses a variant of MCTS, guided by a CNN. Like its predecessor AlphaGo Zero (AGZ), AZ does not use any expert knowledge such as example games played by professional human players; it learns entirely from scratch by playing against itself. The biggest difference between AZ and AGZ is that AZ generalizes the approach to arbitrary two-player zero-sum games. DeepMind demonstrates [6] that it is capable of mastering Go, Chess and Shogi (a Chinese game somewhat similar to chess).

The algorithm can be classified as a model-based RL algorithm that implements Monte Carlo policy iteration, using a variant of MCTS for both policy improvement and evaluation. It is essentially based on two observations; a CNN can be trained to recognize complex features in the board state and accurately predict the policy and the value function, which can then be used to increase the efficiency of MCTS; the policy generated by MCTS will be stronger than that predicted by the CNN, hence it can be used along with the results obtained through self-play, to train the CNN. This concept essentially yields a more intricate version of the evaluation–improvement loop described in Section 2.2. Algorithm 1 provides a high-level overview of the self-play and training steps, although it does not accurately reflect how the algorithm was implemented. The latter is detailed in Section 3.

Let $f_\theta$ denote the CNN, where $\theta$ are the network parameters. It takes as input a representation $s$ of the board state and returns a policy vector $p$ of move probabilities as well as a scalar $v$; $f_\theta(s) = (p, v)$, where $p \approx \pi$ has components $p_a \approx \Pr(a|s)$ and $v \approx V_\pi(s) = \mathbb{E}_z[z|s]$. Here, $z \in \{-1,0,1\}$ is the outcome of the game (loss, draw or win), and the policy $\pi$ is obtained through MCTS. Note that there are no intermediate rewards in this model; the reward is determined only in the terminal states.

The search proceeds as follows: An action $a_t$ is selected at step $t$ using a variant of UCT:

$$a_t = \arg\max_a \left( Q(a_t, s_t) + U(a_t, s_t) \right).$$

The approximate action value $Q(a, s)$ is given by the mean value of the subtree reached by $a$ from $s$, and $U$ is computed as

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

where $P(a, s) = p_a$ except in the root state (explained below), and $N(s), N(a, s)$ are the total number of actions taken from $s$ and the number of times action $a$ has been taken from $s$, respectively. $C(s)$ controls the exploration rate, computed as

$$C(s) = c_{\text{init}} + \ln \left( \frac{1 + N(s) + c_{\text{base}}}{c_{\text{base}}} \right),$$

where $c_{\text{init}}$ and $c_{\text{base}}$ are algorithm parameters. The idea is that the term $U(a, s)$ promotes sensible exploration by favouring actions for which $N(a, s)$ is small and $p_a$ is large.
Additionally, some noise is added to the move probabilities in the root state; if $s_0$ is the current root of the tree, i.e. the state from which simulations are run, then

$$P(a, s_0) = (1 - \varepsilon)p_a + \varepsilon \eta_a,$$

where $\eta \sim \text{Dir}(\alpha)$, i.e. the noise vector is sampled from a Dirichlet distribution with uniform parameters $\alpha_1, \ldots, \alpha_k = \alpha$. Both $\alpha$ and $\varepsilon$ are algorithm parameters.

The rollout is replaced entirely by evaluating $(p_L, v_L) = f_\theta(s_L)$, once a leaf node $s_L$ is reached. $v_L$ is propagated back up the search path, updating $Q(a_t, s_t)$ for each $t \leq L$.

The resulting policy $\pi$ is given by

$$\Pr(a|s_0) = \frac{N(a, s_0)}{\sum_{b \in A} N(b, s_0)}.$$

During self-play, the agent runs MCTS a number of times from the current state $s_t$, and then samples a move $a_t \sim \pi$. The latter is either done proportionally, to further promote exploration, or greedily, to fully exploit the current policy. Once the game reaches a terminal state $s_T$, the outcome $z$ is determined, and each encountered state $s_t$, for $t \leq T$, is labelled accordingly (see Algorithm 1). The network parameters $\theta$ are updated by gradient descent with momentum (see [9, p. 248–249]) on a loss function that combines MSE with cross-entropy loss (see [9, p. 57–58]):

$$E(\theta) = (z - v)^2 - \pi_s^T \ln(p) + \lambda ||\theta||^2,$$

where $\pi_s$ denotes the probability vector for $s$ according to $\pi$, and $(p, v) = f_\theta(s)$. The last term, weighted by the parameter $\lambda$, comes from a technique called weight decay or weight regularization (see [9, p. 226–227]), the purpose of which is to reduce overfitting. That is, to prevent the network from fitting the data too well and thus fail to generalize.

**Algorithm 1: Synchronous self-play and training**

```plaintext
for number of games do
  $S \leftarrow \emptyset$;
  $s \leftarrow$ initial board state;
  while game has not ended do
    add $s$ to $S$;
    for number of simulations do
      $\text{MCTS}(f_\theta, s)$;
      compute $\pi_s$;
      $a \leftarrow \text{SAMPLE}(\pi_s)$;
      $s \leftarrow \text{NextState}(a, s)$;
      $z \leftarrow \text{DetermineOutcome}(s)$;
      $p \leftarrow \text{Player}(s)$;
      for $s \in S$ do
        if Player($s$) = $p$ then
          add $(s, \pi_s, z)$ to Examples;
        else
          add $(s, \pi_s, -z)$ to Examples;
    for number of training steps do
      $B \leftarrow \text{RandomSample}(\text{Examples})$;
      $\theta \leftarrow \text{UpdateWeights}(f_\theta, B)$;
  end while
end for
```

The CNN consists of a large "body" of convolutional layers followed by two separate "heads", a policy head and a value head, which output $p$ and $v$ respectively. The body consists of stacked residual blocks, i.e. blocks where the unmodified block input is added to the output of the block’s hidden layers prior to applying the activation function. Each residual block applies the following transformations in sequence:

1. Convolution with 256 filters of size $3 \times 3$ with stride 1
2. Normalization
3. ReLU
4. Convolution with 256 filters of size $3 \times 3$ with stride 1
5. Normalization
6. Addition with the block input
7. ReLU
The body first applies a single convolution with 256 filters of size $3 \times 3$ with stride 1, followed by normalization and ReLU, and then a sequence of 19 residual blocks. The policy head applies the following transformations:

1. Convolution with 2 filters of size $1 \times 1$ with stride 1
2. Normalization
3. ReLU
4. Game-specific output layer

In the case of Go, the output layer of the policy head is a linear layer of $19^2 + 1 = 362$ neurons, corresponding to the number of possible moves. The output is normalized to a probability distribution by applying the softmax function, which is defined as:

$$ \text{softmax}(x)_i = \frac{e^{x_i}}{\sum e^{x_j}}. $$

The value head applies the following transformations:

1. Convolution with 1 filter of size $1 \times 1$ with stride 1
2. Normalization
3. ReLU
4. A linear layer of 256 neurons with ReLU activation
5. Output layer of 1 neuron with tanh activation

2.6. The game of Go  

Go is an ancient Chinese board game, played by 2 players on a grid of $19 \times 19$ lines. The players take turns placing black and white stones respectively, on the intersections of the lines, with the objective of controlling as much territory as possible. The rules of the game are few and quite simple, but the emergent complexity is vast; there are about $2 \cdot 10^{170}$ legal board states [13], and the average branching factor (see Section 2.4) is 250 [1].

The adjacent intersections of an intersection are those orthogonally adjacent to it, i.e. each intersection (except for those on the perimeter of the board) has 4 adjacent intersections. Every stone on the board must at all times have at least one liberty. The liberties of a stone are the adjacent empty intersections of that stone, and liberties are shared among groups of stones. Adjacent stones of the same color belong to the same group. Groups are also called strings. If a stone is placed on the last liberty of an opponent group, then that group is captured, and the stones are removed from the board. The game ends when both players pass.

There are a few different rule sets used in different parts of the world. They differ slightly in which moves are legal, but mainly in how the game is scored. Described here are the basic Chinese rules: A player may not capture one of their own groups (such a move is called suicide), and may not play a move that results in a repetition of a previous board state (this rule is known as positional superko). A simpler version of this rule, usually referred to as basic ko, prohibits only repetition of the previous board state. The score is computed as the number of stones a player has on the board, plus the number of empty intersections surrounded by that player (this is called area scoring). White is given additional points, called komi, to compensate for black’s advantage of having the first move. Komi of 7.5 points is standard on the $19 \times 19$ board.

3. Method

3.1. Implementation  

The AZ algorithm was implemented in Java, and the open-source ML framework Deep Java Library (https://djl.ai) was used to implement the neural network. The reason for using Java is that object oriented abstraction facilitates a modular implementation, which is important for the usability of the codebase for future experimentation. For example, the neural network component could be replaced
by any other algorithm wrapped in a class that implements the appropriate interfaces. The implementation of Go uses the basic ko rule and area scoring.

The core algorithm is implemented as per the description in [6], but a few tweaks were made based on ideas and observations found in an article series [10] on the subject. The implementation uses concurrent MCTS and an asynchronous self-play and training procedure:

Self-play is executed by $W$ worker threads, each one using a pool of $S$ search threads for running MCTS simulations, for a total of $W$ parallel games and $SW$ concurrent simulations. The generated examples are stored in shared memory. To prevent all of the search threads from taking the same path down the tree, a technique called virtual loss is used; when a node is visited, a virtual loss counter $\ell$ is incremented in the node. When the node is queried for its mean subtree value, that value is adjusted as if $\ell$ prior visits to the node had resulted in a loss (from the perspective of the player whose turn it is in the querying thread’s state). The counter is decremented upon backpropagation.

Inference is executed by $I$ inference service threads, each one serving $W$ workers. The inference service threads are independent and can use different devices for executing the forward pass of the network. They each have a request buffer and a minimum target batch size $B$, and will wait some amount of time to accumulate $B$ requests before executing inference. If the timeout expires they will execute with however many requests are currently pending. The reason for this mechanism is that running inference on single examples is extremely inefficient, especially if run on a GPU to which the data must first be copied. To further increase efficiency, inference results are cached in shared memory, so that each unique request need only be processed once between parameter updates. In this instance, a inference batch size of 8 and a timeout of 1 ms was used.

Training is executed in a single thread, which continually samples batches of $N$ example positions from shared memory and updates the network parameters. Batch sampling is done uniformly random across all available examples. The updated network parameters are written to shared memory at regular intervals, where they are available to the inference service threads, which continually check for new parameters.

The buffer where training examples are kept is implemented using a queue, so that the oldest examples are pushed out as the queue fills up. An initial and maximum buffer size is set, and the buffer size is then dynamically increased over time based on the rate at which examples are generated. The increments are sized so that a target percentage $p$ of the buffer contents are replaced over the next 1000 training steps, where $p$ decays over time. The idea is that starting off with a small buffer and replacing the early examples quickly can speed up convergence, since the early examples will have rather low quality training targets. DeepMind uses a fixed buffer size of 500 000, however using a large fixed buffer size could potentially cause slower convergence when sufficient hardware is not available to generate training examples fast enough to fill up the buffer in reasonable time, since the early examples would then be present in the data set for most of the training process.

In order to increase the rate at which training data is generated, this implementation exploits the fact that the rules of Go are invariant under reflection and rotation. For each board state and target policy produced during self-play, the corresponding reflections and rotations are also computed. This way of augmenting the training data was employed by DeepMind in the AG and AGZ algorithms, but not in AZ since it is specific to Go.

The input to the network is a $(2t+1) \times n \times n$ matrix with binary entries, where $n$ is the board size, and the first $2t$ feature planes indicate the positions of each player’s stones.
in the current state and the \( t - 1 \) preceding states, from the perspective of the current player. The last feature plane is all ones or all zeros, indicating black to move or white to move respectively. Since this implementation of Go uses the basic ko rule, only two positional feature planes are required to make the state fully observable, hence \( t = 2 \) is used.

The value component of the training targets are weighted between the outcome \( z \) and the mean subtree value \( \overline{v} \) of the corresponding MCTS node, as \((1 - \omega)z + \omega\overline{v}\). While \( z \) will typically be more accurate, it can also be quite noisy; if a bad move is chosen from a good state, ultimately resulting in a loss, then that state will be labelled with \(-1\), which might not be a fair evaluation. Since \( \overline{v} \) is the result of many simulations from that state, it represents a more balanced evaluation. On the other hand, \( \overline{v} \) is based only on network inference, which makes it very inaccurate early on in the training process. Combining the two could make for an overall better training target.

The network parameters are initialized using uniformly random Xavier initialization (see [5]), which helps prevent vanishing and exploding gradient issues by initializing the weights with approximately equal variance across layers.

Rather than decreasing the learning rate at a few predetermined intervals like a step function, this implementation uses a continuous cyclical learning rate. Initial and target learning rates \( \gamma_0, \gamma_T \) are set, and the learning rate at step \( i < T \) is then computed as

\[
\gamma(i) = r(i) + r(i) \left(1 - \frac{i}{T}\right) \sin^2 \frac{\pi i}{f},
\]

where

\[
r(i) = \gamma_0 + \frac{\gamma_0 - \gamma_T}{2} \left(1 + \cos \frac{\pi i}{T}\right)
\]

and \( f \) controls the oscillation frequency. The resulting curve looks locally like a sinusoid with period \( f \) and amplitude \( \frac{1}{2}r(i) \):

The reasoning behind using using a cyclic learning rate goes as follows: If the learning rate is too small, optimization can easily get stuck in a local minimum. If it is too large, optimization can "overshoot" the target and fail to converge. By varying the learning rate we can potentially mitigate both of these issues, at least to some extent. It can also make the model a bit less sensitive to the choice of initial learning rate.

3.2. Evaluation

Ideally, the performance of the agent would be evaluated against several other independent agents such as other Go playing programs with known approximate Elo ratings. However, due to time and resource constraints, this project settles for more rudimentary evaluation; the agent is trained on a \(5 \times 5\) board using 0 points komi, and evaluated in self-play against previous iterations, as well as an agent using basic MCTS. The basic MCTS agent is allowed 5 times the number of simulations per move compared to the AZ agent, and uses UCT to balance exploration and exploitation. Each match-up consists of 100 games.

It is worth noting that Go on a \(5 \times 5\) board is actually solved [15]; if black plays optimally, then white cannot make a living group, hence if komi is set to 25 points, optimal play from both sides results in a draw under area scoring. The decision to use 0 komi is motivated
by the empirical observation that using the correct komi of 25 seems to make it much harder for the algorithm to produce any sensible play. This is probably because it quite severely dissociates the reward signal from the agents performance; black will not see anything but negative reward until it takes the entire board, and if black does not, then white can win without even playing.

The network was scaled down to make training at least somewhat feasible on the available hardware. 64 filters were used in the convolutional layers rather than 256, and 5 residual blocks were used rather than 19. The choice of 64 filters is simply due hardware limitations, whereas the number of filters was chosen to match the board size. The latter is based on the intuition that, since a convolution with a $3 \times 3$ filter produces a feature map in which each pixel value is a function of the 8 surrounding pixel values in the original image, a sequence of 5 convolutions is enough for the "pixel value" in one of the corners of the board to potentially affect that in the opposite corner.

Table 1: Configuration of AZ parameters. 600 simulations per move is used in evaluation.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch size</td>
<td>1024</td>
</tr>
<tr>
<td>Simulations / move</td>
<td>400 (600)</td>
</tr>
<tr>
<td>Residual blocks</td>
<td>5</td>
</tr>
<tr>
<td>Filters</td>
<td>64</td>
</tr>
<tr>
<td>Initial learning rate</td>
<td>1E-2</td>
</tr>
<tr>
<td>Target learning rate</td>
<td>2E-4</td>
</tr>
<tr>
<td>Momentum</td>
<td>0.9</td>
</tr>
<tr>
<td>Weight decay</td>
<td>1E-4</td>
</tr>
<tr>
<td>Dirichlet noise ($\alpha$)</td>
<td>0.4344</td>
</tr>
<tr>
<td>Noise weight ($\varepsilon$)</td>
<td>0.25</td>
</tr>
<tr>
<td>Value target weight ($\omega$)</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Table 1 summarizes the final configuration of the most important parameters. DeepMind states [6] that the noise parameter $\alpha$ is scaled in inverse proportion to the average number of legal moves, but they do not go into more detail. They use $\alpha = 0.03$ for $19 \times 19$ Go, which yields a proportionality constant of 7.5, given an average of 250 legal moves. Assuming the latter scales linearly with the number of possible moves, $\alpha$ is set to

$$\alpha = \frac{7.5 \cdot 362}{250(n^2 + 1)}$$

for a $n \times n$ board. For comparison, DeepMind used a value of $\alpha = 0.3$ for chess.

The exploration parameters $c_{\text{base}}$ and $c_{\text{init}}$ were set to 19652 and 1.25 respectively, as per [6]. The values for initial and final learning rates, momentum, weight decay and noise weight ($\varepsilon$) were also found in [6].

4. RESULTS

Figures 6 and 5 show the results of evaluation in self-play and against basic MCTS respectively. Figure 7 shows the performance against basic MCTS when AZ only blays black. The 121000 training steps took roughly 16 hours to execute using a single GPU and a 6-core (12 thread) CPU.

Figures 8 and 9 show examples of positions from self-play along with examples of feature maps produced by the first two convolutional layers of the network.

![Figure 4: Percentage of games won and won/drawn against an agent using basic MCTS with UCT and 600 simulations per move.](image)
Figure 5: Percentage of games won and won/drawn against an agent using basic MCTS with UCT and 3000 simulations per move.

Figure 6: Percentage of games won and won/drawn in self-play against the agent 10 generations prior.

Figure 7: Percentage of games won and won/drawn when AZ plays only black, against an agent using basic MCTS with UCT and 3000 simulations per move.

Figure 8: Opening moves. The top two rows show, left to right, the board position and move played, the probability distribution resulting from search, and the probability distribution produced by the network. The bottom row shows selected feature maps (whites move) as probability distributions (left), and scaled with min-max feature scaling (right).

Figure 9: Capture by black and whites response. The top two rows show, left to right, the board position and move played, the probability distribution resulting from search, and the probability distribution produced by the network. The bottom row shows selected feature maps (whites move) as probability distributions (left), and scaled with min-max feature scaling (right).
5. Conclusions

The AZ agent performs roughly on par with the basic MCTS agent when the latter uses 5 times as many simulations per move. Although this performance is not impressive, the quantitative and qualitative evaluations together give some confidence that the implementation works as intended. The results from both self-play and play against basic MCTS seem to indicate that learning quickly stalls, with no apparent improvement on these metrics beyond 30–40 generations. There are many possible explanations, but a few that seem plausible are:

Premature convergence – If the model converges too quickly in the early stages of training it may overfit, which could slow or hinder further improvement. Slow generation of training data could potentially cause or worsen this kind of problem, as the size and diversity of the data set is an important factor in preventing overfitting. With the hardware and parameter configuration used in this instance, the training process evaluated positions roughly 4–5 times as fast as they were being produced by self-play; a rate of about 2000 positions per second compared to about 400–500.

Insufficient network capacity – It may be the case that 64 filters and 5 residual blocks is not nearly enough, and that the network is in fact incapable of recognizing more complex board features in this configuration.

Problematic evaluation setting – The fact that Go on a 5 × 5 board is so heavily skewed in favour of black, and that no komi was given, makes the results a bit more difficult to interpret. It could be the case that the algorithm fails to improve the strategy for white due to being starved of positive examples, which could consequently stall improvement for black as well.

It should be noted however, that 121000 training steps is not much to go by; it cannot be ruled out that further training would eventually yield improvements. It should also be noted that 100 games per match-up is not enough to yield very reliable data, hence Figure 6–7 should be seen as a fairly rough indication.

Empirically, it seems that the algorithm has learned at least some rudimentary aspects of the game. One interesting observation is that black consistently opens in the centre, which is in fact optimal play on the 5 × 5 board. As shown in Figure 8, the network gives a 100% probability to that move. The feature maps shown in Figure 8 could be interpreted as implying that the networks internal representation of the board contains some information about adjacency, in particular positions that would hinder the opponents construction of a coherent group in this example. In Figure 9, one of the feature maps identify the position that the opponent (black) just captured, and the other one seems to indicate the positions of the players own stones.

When it comes to the performance of the algorithm itself, the biggest bottleneck is generating training examples through self-play. Each new state encountered during search requires network inference, which incurs a significant computational cost. Even with inference caching and exploiting symmetry, this overhead makes the algorithm very demanding in terms of hardware. Network inference during self-play cannot be done in large batches since this would incur too much latency, hence the inherent overhead of data transfer to and from a GPU becomes very large. This is especially true with a small network.

6. Discussion

Most real-world domains where RL is applicable have vast state- and action spaces, making it infeasible to store the policy and the value function explicitly. This is certainly the case for Go, even for reduced board sizes, hence the need for approximations. This is essentially the purpose of the neural network in the AZ algorithm; it allows the agent to en-
code an approximation of the learnt policy and value function in the network parameters. It is therefore natural to consider the possibility of a more computationally efficient way of approximating arbitrary multivariate functions than a neural network. However, such a general problem setting becomes subject to the "no free lunch (NFL) theorem" (see [16] and [9, p. 24–25]), which, roughly speaking, states that all optimization algorithms perform equally when the performance is averaged across every possible objective function. In other words, an algorithm that performs better than other algorithms on a particular class of problems must consequently perform worse on some other problem class. In the context of fitting a function to data, this means that better performance on some class of problems is in a sense equivalent to leveraging assumptions about the structure of the data.

The use of a CNN in the AZ algorithm is intuitive, since the natural state representation is 2-dimensional and contains local features whose relative positions and orientations form some global structure. In that sense, the task is very similar to image classification. Intuitively, it seems likely that there is more structure to be leveraged in this kind of "image" representation of a board state than there is in say images of human faces, since the former is free of noise. Hence it might be possible to somehow use a CNN more efficiently for this particular application by deriving more domain-specific techniques.

7. Future work

First and foremost, there is more testing to be done within the scope of this project. In particular, training an agent on a reasonably effective hardware setup would give more room for tweaking parameters that have significant impact on runtime, such as the network size and number of simulations per move. This could help identify limiting factors and allow for better evaluation of the agents asymptotic performance.

As for further experimentation, the implementation provides a good starting point by allowing components such as the CNN or the search algorithm to be easily replaced. As discussed in Section 5, network inference is a big performance bottleneck, hence this should probably be the first thing to focus on in trying to improve performance. It might for example be interesting to see if something like the Viola-Jones object detection algorithm [14] could be adapted to this end. Of course, despite a decade of rapid advances in the field, there is likely room for further improvements in neural networks as well, and any techniques that improve computational efficiency could potentially be useful here.
REFERENCES


[2] Tom B. Brown; Benjamin Mann; Nick Ryder; Melanie Subbiah; Jared Kaplan; Prafulla Dhariwal; Arvind Neelakantan; Pranav Shyam; Girish Sastry; Amanda Askell; Sandhini Agarwal; Ariel Herbert-Voss; Gretchen Krueger; Tom Henighan; Rewon Child; Aditya Ramesh; Daniel M. Ziegler; Jeffrey Wu; Clemens Winter; Christopher Hesse; Mark Chen; Eric Sigler; Mateusz Litwin; Scott Gray; Benjamin Chess; Jack Clark; Christopher Berner; Sam McCandlish; Alec Radford; Ilya Sutskever; Dario Amodei. “Language Models are Few-Shot Learners”. In: Advances in Neural Information Processing Systems. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. URL: https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfa4967418bf8ac142f64a-Paper.pdf.

[3] David Silver; Aja Huang; Chris J. Maddison; Arthur Guez; Laurent Sifre; George van den Driessche; Julian Schrittwieser; Ioannis Antonoglou; Veda Panneershelvam; Marc Lanctot; Sander Dieleman; Dominik Grewe; John Nham; Nal Kalchbrenner; Ilya Sutskever; Timothy Lillicrap; Madeleine Leach; Koray Kavukcuoglu; Thore Graepel; Demis Hassabis. “Mastering the game of Go with deep neural networks and tree search”. In: Nature 529 (2016), pp. 484–489. DOI: 10.1038/nature16961.

[4] David Silver; Julian Schrittwieser; Karen Simonyan; Ioannis Antonoglou; Aja Huang; Arthur Guez; Thomas Hubert; Lucas Baker; Matthew Lai; Adrian Bolton; Yutian Chen; Timothy Lillicrap; Fan Hui; Laurent Sifre; George van den Driessche; Thore Graepel; Demis Hassabis. “Mastering the game of Go without human knowledge”. In: Nature 550 (2017), pp. 354–359. DOI: 10.1038/nature24270.


[6] David Silver; Thomas Hubert; Julian Schrittwieser; Ioannis Antonoglou; Matthew Lai; Arthur Guez; Marc Lanctot; Laurent Sifre; Dharshan Kumaran; Thore Graepel; Timothy Lillicrap; Karen Simonyan; Demis Hassabis. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: Science 362 (2018), pp. 1140–1144. DOI: 10.1126/science.aar6404. URL: https://science.sciencemag.org/content/362/6419/1140.


