Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 260

Evaluation of Reduced Neural Network Models for Predicting Go Game Moves

Thomas Wendt

Course of Study:	Softwaretechnik
Examiner:	PD Dr. rer. nat. habil. Holger Schwarz
Supervisor:	Mark Blaxall (Sony Deutschland GmbH)
Commenced:	September 21, 2015
Completed:	March 22, 2016
CR-Classification:	C.1.3, I.4.8, I.5.1

Abstract

With increasing processing power and the introduction of GPUs, convolutional neural networks are getting more and more complex. While these networks are able to solve more complex tasks, they are less suited for use on a mobile platform where there are stricter memory and power constraints. We will look at neural network reduction methods, which aim to reduce the memory and power requirements of convolutional neural networks, whilst maintaining their quality. These methods are applied and evaluated with a Go move predicting network. Additionally an Android App is developed that is able to recognize a Go board and stone positions in order to use the reduced network to predict the next best moves.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

Deep Neural Networks are getting more and more complex. The introduction of GPUs to machine learning has made it possible to process more data and to generate bigger models. The memory requirements of these models can be in the order of several hundred Megabytes, which can be too much for mobile devices. On battery powered devices it is also important to consider the power consumption needed by the model. These two factors make it difficult, if not entirely impossible, to use these models on mobile and embedded platforms where the hardware is not powerful enough and the available memory is too small. A usual solution is to move these tasks to cloud services where it is possible to use powerful servers equipped with multiple GPUs.

The reduction methods outlined in the thesis are applied and evaluated on a Go move predicting neural network with the goal to produce a reduced model that is more suitable for deployment on a smartphone without losing its prediction performance. Different reduction approaches are applied to the network and evaluated in detail.

With around 10^{170} legal states of the board, Go is one of the most complex board games [TF07; Wik16a]. This poses a difficult problem for traditional tree search based artificial intelligences as they cannot fully search the state space and find good moves. No Go artificial intelligence had been able to beat a non-handicapped professional Go player up until October 2015, when Google DeepMind's AlphaGo won 5 – 0 against the current and three time European Go Champion Fan Hui. In March 2016, AlphaGo wrote history again when it won 4 – 1 against Lee Sedol, one of the worlds strongest Go players[Lib16; Alp16].

In 2014 C. Clark and A. Storkey developed a neural network that can predict an expert players move with an accuracy of 44% and beat GnuGo, a popular Go artificial intelligence, in 86% of the games [CS14]. The network is rated at around 4-5 kyu, which means it is still beatable by advanced Go amateurs, yet stronger than GnuGo. When starting this thesis it was the strongest network publicly available. A trained model was published that is used in this thesis as the baseline. By reducing and integrating this model into a mobile App, it is possible to provide a better artificial intelligence than the majority of available Apps on the Android Play Store, which heavily rely on GnuGo.

This thesis is written in cooperation with Sony, which is interested in exploring the possibilities of neural networks on mobile devices. During an internal Sony conference, which was held in Tokyo in November 2015, the reduction methods and their benefits were presented. As the game of Go is well known in Japan, the reduction methods were shown as part of a Go demo developed in this thesis. To catch the attention of the attendees, a real Go board is analyzed by an Android App, the positions of the stones are detected and the predictions for the next best moves are displayed.

1.1 About Sony

The Stuttgart Technology Center (STC) is Sony's biggest R&D site outside of Japan and the home of the European Technology Center (EuTEC). At EuTEC, Sony's engineers are actively working in the areas of Computational Imaging, Communications & Radar Systems, Speech & Sound Processing and Machine Learning. EuTEC contributions to the product and technology portfolio of Sony include:

- Mm-wave radar systems
- Digital demodulation ICs for worldwide reception of analogue TV standards (audio & video)
- Speech recognition and natural language processing for Sony devices (EU languages)
- 3D surround sound
- Depth estimation technologies by with and without active illumination
- Design for optical engines

Part of EuTEC is the Speech and Sound Group (SSG) which has its roots in Speech Recognition. But nowadays 9 employees and over 10 students from around the world are also doing innovative research in the areas of Natural Language Processing and Machine Learning in general.

1.2 Outline

In Chapter 2 the concepts and rules of Go are introduced and why Go is an interesting topic right now. The developed Android App and its components are covered in Chapter 3 where the most interesting component is the board and stone detection. A short introduction into convolutional deep neural networks is provided into Chapter 4. After that, the neural network reduction methods are explained in Chapter 5 and their implementation is detailed in Chapter 6. This leads to the evaluation of the reduction methods in Chapter 7 and the final words in Chapter 8.

2 The Game of Go

Go is an ancient board game that originated in China 2500 years ago. The board usually consists of a 19×19 grid and two players try to surround areas or capture stones of the opponent by placing their own stones on the board in turns. A typical Go board at the end of a game can be seen in Figure 2.1.

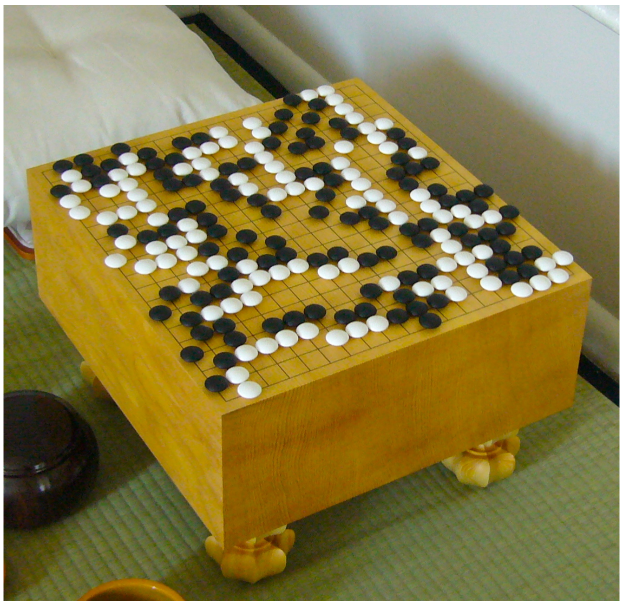
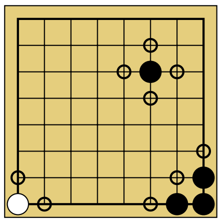
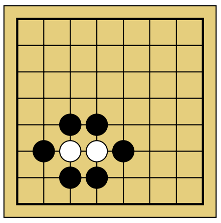


Figure 2.1: A typical end position of a Go game.

The basic Go rules, which are covered in the next section, are relatively simple however, the size of the board and the number of possible moves in each turn make it impossible for a computer to evaluate all of them.

Players in Go are ranked from beginner to professional, with some titles reserved for special players. Beginners are ranked from 30k to 10k (kyu) and intermediate players from 9k to 1k, where a lower number indicates a better player. Advanced players are ranked with increasing numbers from 1d to 9d (dan). Finally the best players are ranked from 1p to 9p (professional).


(a) A white stone with two liberties, a black stone with four and a group with three.


(b) Group of white stones with no liberties. The white stones were captured and have to be removed.

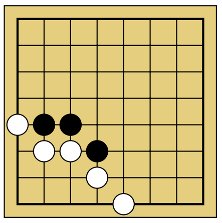

(c) Bottom left territory belonging to white. Remaining territory is neutral.

Figure 2.2: Examples showing stone liberties, capturing stones and conquering territory.

2.1 Rules

There are different rulesets for Go, with the Japanese and Chinese rules being the most common ones. Fortunately, the differences are not too relevant for this thesis as they largely only affect the final scoring.

The basic Go rules are as follows:

- Common sizes of a Go board are 9×9, 13×13 or 19×19, with the latter being the official size for tournaments.
- Black is first to play and has to place a stone on the grid of the board.
- The number of liberties of a stone is the number of free adjacent top, right, bottom or left positions. Figure 2.2a shows the liberties as circles next to the stones.
- Adjacent stones with the same colors are grouped together and share their liberties as shown in the bottom right of Figure 2.2a.
- Single stones or a group of stones can be captured by surrounding them as seen in Figure 2.2b. Captured stones are removed from the board after the move.
- The goal is to capture opponent stones and to conquer territories on the board similar to Figure 2.2c.
- Players are not allowed to place a stone at a previously captured position, if the resulting position appeared previously as shown in Figure 2.3.
- The game ends when a player forfeits or both players pass a move by deciding not to place a stone.
- The score is calculated by adding the number of captured stones to the size of the territory a player holds, and taking a possible handicap into account. The exact way to calculate the score differs from ruleset to ruleset.

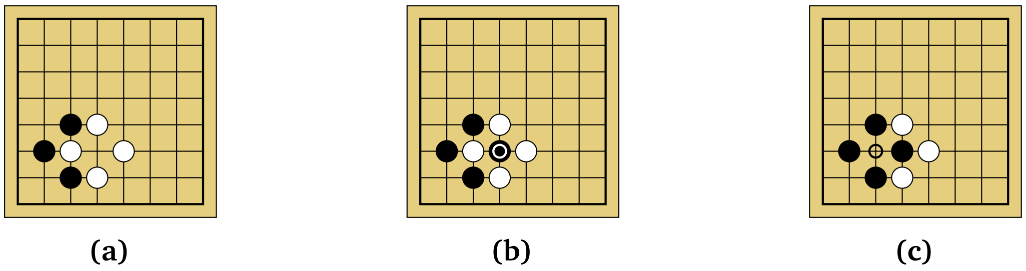


Figure 2.3: The KO rule: Black captures a white stone with the move shown in (b) resulting in the board shown in (c). White is not allowed to capture the previously played black stone, as it would result in the same position, as shown in (a).

2.2 Go AIs

Today's strongest Go AIs, such as Fuego and Pachi, are based on Monte Carlo Tree Search (MCTS) methods that play out the complete game randomly or based on heuristics and then select the move with the highest probably of winning the game. These AIs are ranked between 2 kyu and 2 dan, making them easily beatable by strong Go players. On average, a player can make 250 different moves at any given time. And with an average length of 150 moves, this results in a huge number of combinations [All94]. Therefore, this method can only explore a fraction of the possibilities. The number of possible states the board can have is equally large. Each position on a Go board can be black, white or empty. For a 19×19 board, this gives $3^{19 \cdot 19} \approx 1.74 \cdot 10^{172}$ number of total states. The number of legal states is smaller and calculated to be "only" around 10^{170}. This makes Go many times more complex than, for example, chess. In fact it is one of the most complex board games.

The neural network approach is different, as it avoids the extensive tree search based approach for a purely pattern matching method. By training a neural network with recordings of thousands of Go games and millions of moves it tries to mimic human players. The expectation is that the moves done by the players are good, and by merely repeating them, when a similar board is detected, a good Go playing model can be generated.

A network is then able to predict the next move by just looking at the current board and returns a probability per board position to be an expert's move. No information about previous moves or exploration of the state space is required. In [CS14] this approach showed promising results, predicting the correct moves by 44%. Independently, [MHSS14] achieved a 55% move prediction accuracy. Both networks win against GnuGo in the majority of games and can win some games against stronger Go AIs like Fuego and Pachi.

Based on the new research, a Go AI using MCTS and neural networks managed to beat a professional human player for the first time in October 2015. Google Deepmind's Alpha Go won this match 5 – 0 against Fan Hui, the current European Go Champion. While Go AIs have managed to win against professional players in previous matches, they had advantage of 4 – 9 stones and AlphaGo's achievement was previously thought to be at least a decade away [Lib16; SHM+16].

3 Android App

Part of this thesis was the development of a demonstration setup to present the benefits of the reduced models in terms of computational time, memory usage and energy consumption on mobile devices. The developed reduction technologies were shown at an internal Sony exhibition which took place in Japan during November 2015. As this exhibition is attended by many people with different backgrounds, the demonstration was intended to serve as an eye-catcher for the attendees. The game of Go is well known in Japan, so it was an obvious choice to create a demonstration setup, that lets a human player challenge the neural network, and compare the reduced model to the original one.

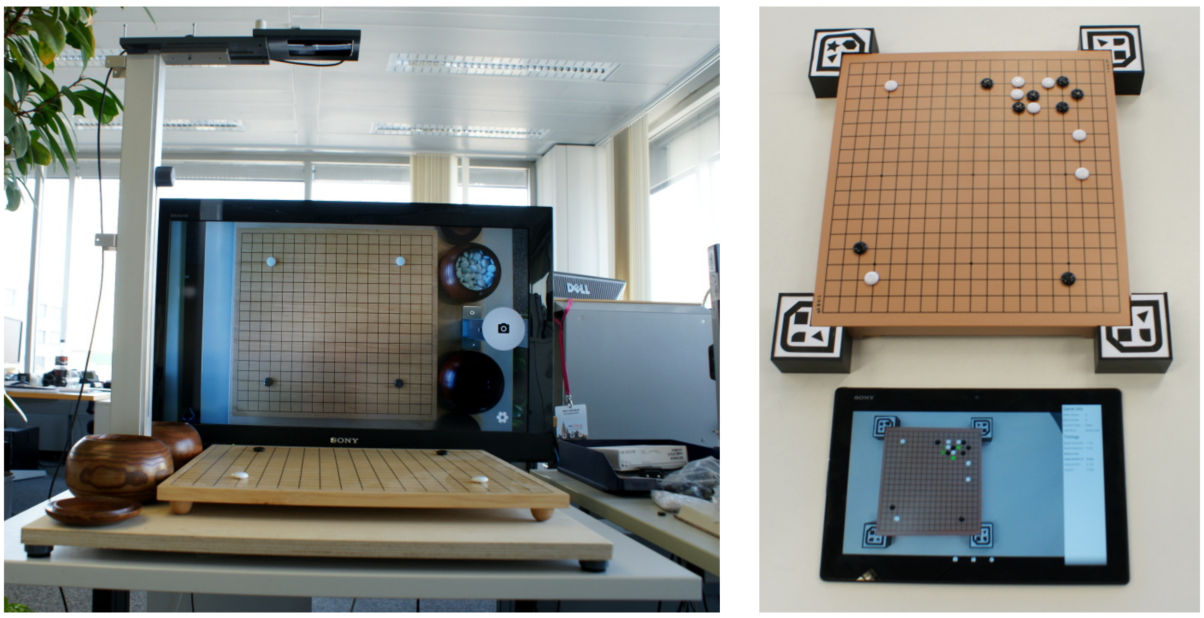


Figure 3.1: The preliminary demonstration setup with a smartphone mounted above a large Go board and a TV mirroring the display from the phone. On the right the final setup with the marked travel board and a mobile device.

Initially, the proposed plan was to mount a smartphone above a fixed Go board to ensure easy detection of the board position. The smartphone would then analyze the images from the camera, calculate the predictions for the original and a reduced model and show the move prediction on its display. The narrow angle of view of the smartphone camera and the big Go board required the camera to be mounted at high position. This made it hard to see the actual results on the smartphone display. Therefore a TV, to mirror the display contents though MHL, was added to the setup as seen in Figure 3.1.

The jig allowed the position of the smartphone to be extended and adjusted in height through two sliders. It was created by an in-house technician with our requirements in mind, which included easy disassembly and assembly for transportation to Japan. However, the jig was too big and heavy for regular luggage and the whole setup exceeded the space that was allocated to us at the exhibition. It was decided to switch to a hand-held operation of the smartphone or tablet, which allowed us to remove the TV and jig and highlight the mobile aspect even more. The initial Go bard was additionally swapped with a more compact travel board.

3.1 Overview

The App can be separated in four distinct components that can be considered useful on their own. Yet there was no focus on making sure the components are totally independent.

Board and Stone Detector: Responsible to extract the position and color of stones on a Go board from an image.

Go Engine: Implementation of the Go rules to indicate illegal placed stones or stones that should be removed. It also keeps track of the board history to make the KO position, the position illegal under the KO rule, available.

Move Predictor: Implementation of the neural network. It calculates the move predictions based on the board state.

GUI: Displays the detection and prediction result and allows interaction such as taking a picture and changing settings.

An overview of the architecture and the communication between the components is shown in Figure 3.2. The individual components will be explained in more detail in the following sections.

3.2 Board and Stone Detector

The Board and Stone Detector is responsible for finding the Go board in an image and to detect the color and position of the individual stones. The basic steps are shown in Figure 3.3. This component is implemented using OpenCV, a computer vision library. OpenCV is a powerful toolkit, providing many useful functions for the given task. OpenCV can be

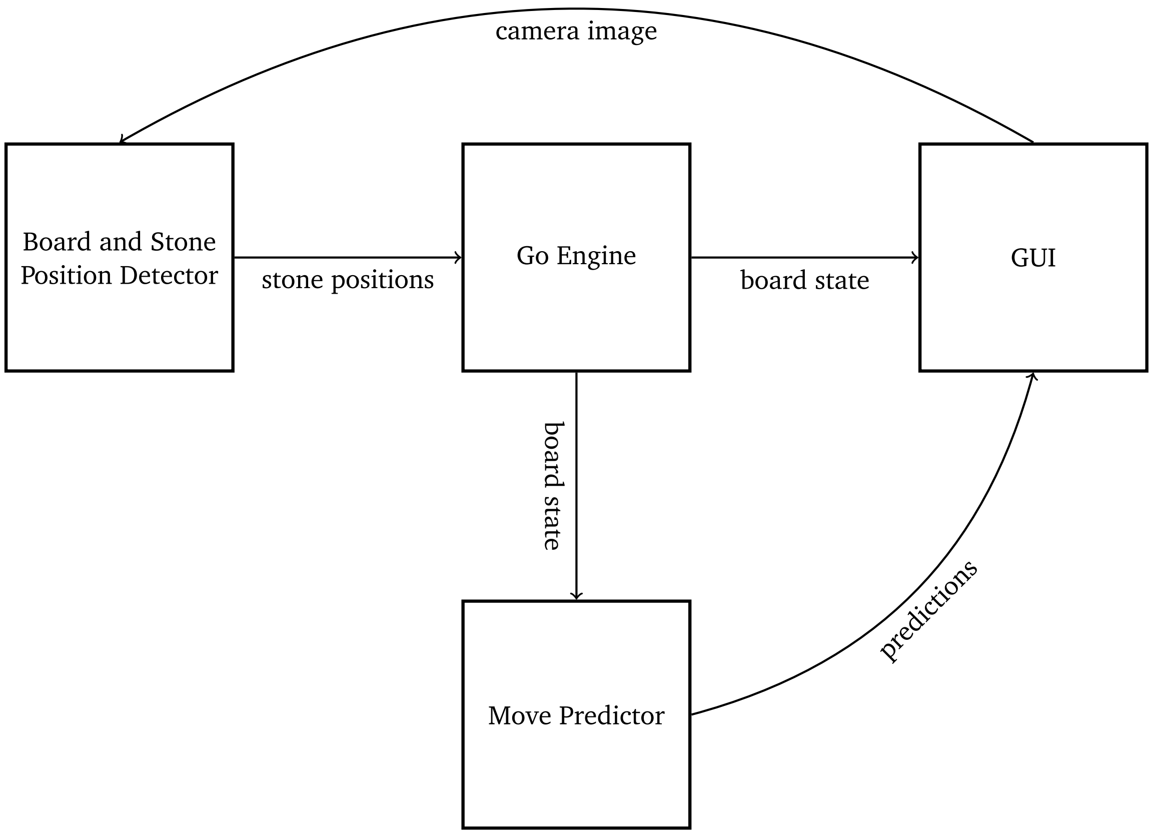


Figure 3.2: The Android App components and communication flow.

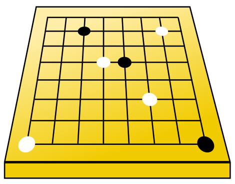
used in conjunction with Java, which is important for the development of an Android App. The OpenCV libraries are also available in the Google Play Store.

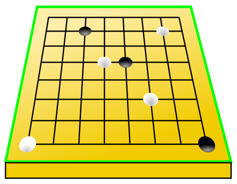
Development of the of the Board and Stone Detector was primarily done under Linux to avoid updating the App on the Android device all the time. To make the move to the Android system as seamless as possible, the first prototypes were created using the OpenCV Java bindings. It was quickly apparent, that due to limitations of the Java language, bad documentation and lack of good Java examples, the development in Java would be cumbersome. An simple example of a Java limitation is the non-existence of operator overloading, which allows the redefinition of basic operators like $+$, $-$ or $*$. To multiply two OpenCV matrices in Java one has to write the following code:

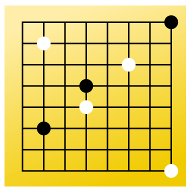
```
Mat C = new Mat();
Core.multiply(A, B, C);
```

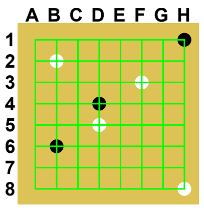
In other languages, such as C++, this can be written as:

```
Mat C = A * B;
```


(a) An image of a Go board.


(b) The exact position of the Go board is detected.


(c) The Go board in the image is transformed into a rectangle.



(d) Average color on a expected stone position is used to detect the stones.

Figure 3.3: The basic steps of the doard and stone detection.

The Java bindings for OpenCV are also very restrictive about the types of arguments functions require. In C++ many functions take anything resembling a set of numbers, such as an OpenCV matrix object, an OpenCV vector object or a C++ vector of numbers, as input. This is not possible when using the Java bindings and causes a constant manual data conversion to the required data type.

Therefore, the programming language was changed to C++, which made the development easier and had the additional benefit of an improved documentation. The Board and Stone detector is then compiled as a native library for the Android platform. A Java class handles converting the input and output for the library.

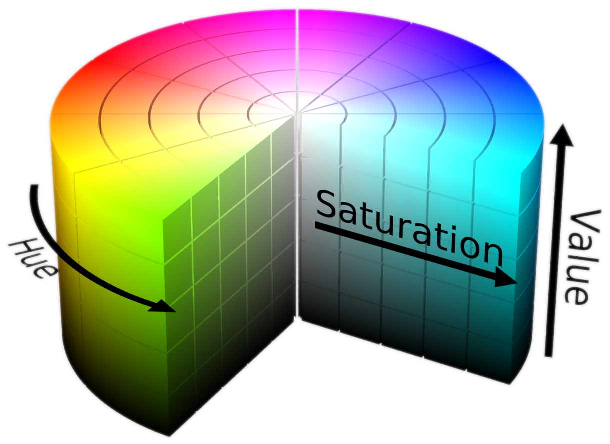
Detecting the exact position of the board is critical as this is the basis for the stone coordinates produced by the stoned detector in Section 3.2.3. Misdetection at this stage, or being just slightly off, will result in stones being placed on the wrong coordinates or not detected at all. Two board detection approaches where tried. The first one is detailed in the next section and tries to detect the board based on colored markings. As the detection results were not good enough, an alternative was necessary. This alternative second approach is explained in Section 3.2.2 and uses special markers for an accurate detection of the board.

3.2.1 Colored Board Detection

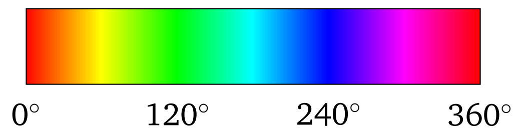
The initial idea was to mark the board using colors, so it can be easily recognized in images. As a first prototype it was asked to detect the board by using the algorithm found in the “color-blob-detection” sample from OpenCV [Ope16a]. The sample App uses the preview frames from the Android camera and outlines the areas with the selected colors. Different colors can be selected by touching areas in the image with the wanted color.

The image processing in the sample consists of the following steps.

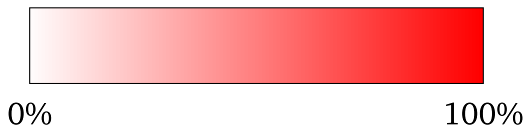
1. Convert the preview frame (720p or 1080p) from the camera API into a RGB image.



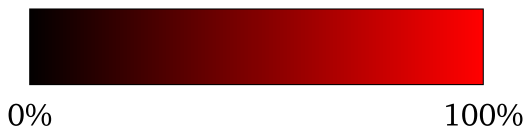
(a) The HSV color space mapped to a cylinder[Wik16b].



(b) Hue values in degree and their colors.



(c) The effect of the *saturation* for red.



(d) The effect of the *value* for red.

Figure 3.4: The HSV color space visualized on a cylinder and the effects of the *saturation* and *value* components for $H = 0°$ shown on the right side.

2. Downsample and blur the image to reduce the data size and noise.
3. Convert the RGB image to the HSV color space.
4. Find all pixels within a specific color range.
5. Dilate the found areas to combine areas that are very close together.
6. Remove too small areas.

The HSV color space is particular useful for this task, as it allows to match a specific color independent of the brightness. HSV stands for hue, saturation and value where *hue* component can be interpreted as the desired color (e.g. red or blue), the *saturation* component as the brightness and the *value* component as colorfulness. In Figure 3.4a the HSV color space is visualized on a cylinder and the effects of the components shown on the right side. The *hue* is specified as an angle from 0 to 360°, as shown in Figure 3.4b, and *saturation* from 50 to 100% and *value* also from 50 to 100%.

The board was marked with four red stickers in the corners of the board and the image processing adjusted to the specific use-case to expect exactly four red markers. For each detected color blob, the pixel farthest away from the image center is used to span a polygon around the board as shown in Figure 3.3b.

By having the coordinates of the corner points, it is possible to calculate a transformation, which transforms the image in such a way, that the board contents take up the whole image. To calculate this perspective projection, four points $(x_0, y_0) \dots (x_3, y_3)$ in the camera

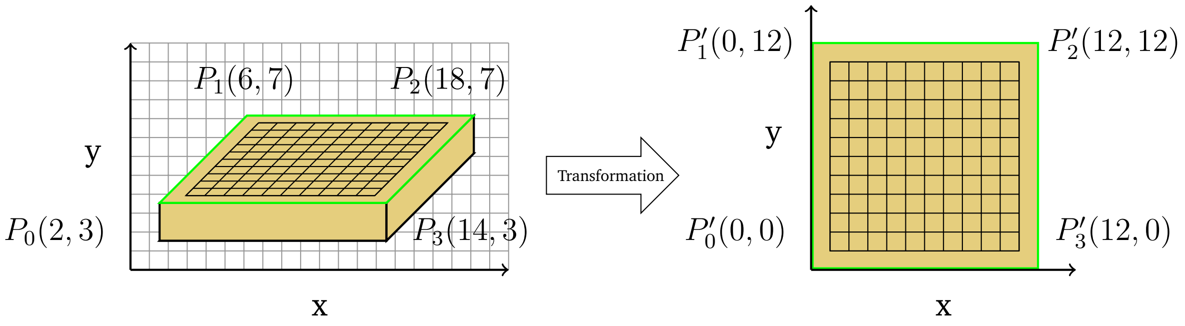


Figure 3.5: A perspective transformation is used to transform the corner points in image coordinates to coordinates on a rectangle.

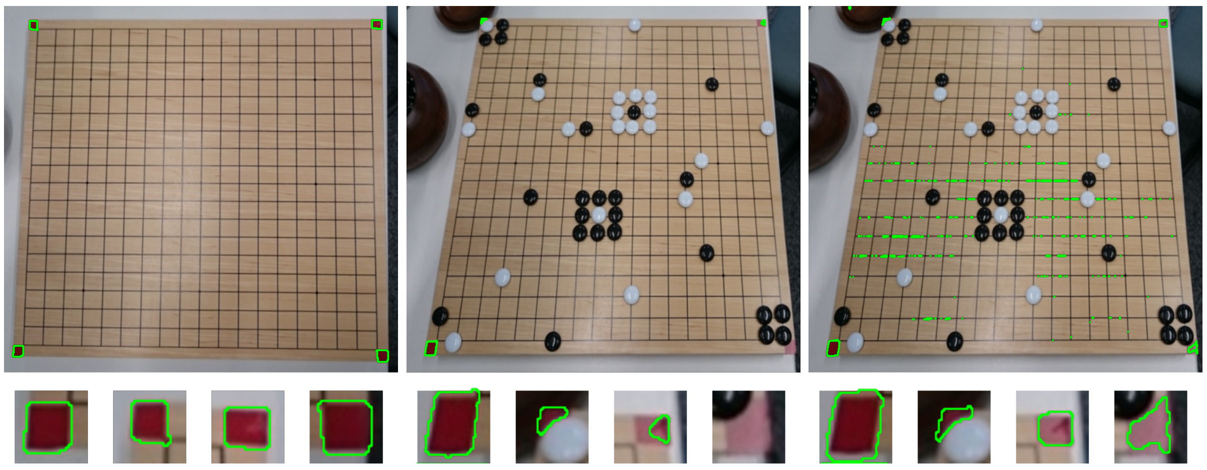
image and four points $(x'_0, y'_0) \ldots (x'_3, y'_3)$ in a reference image are needed (see Figure 3.5). A mapping matrix A can then calculated that performs the necessary mapping[Ope16d]:

$$\begin{bmatrix} x'_i \\ y'_i \\ t_i \end{bmatrix} = A \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \qquad \text{(for } i = 0 \ldots..3\text{)}$$

The reference points are placed on a rectangle to make it easy to interpolate the stone positions on the board. The transformation matrix is calculated using OpenCV's `getPerspectiveTransform` function and the image is transformed by using the `warpPerspective` function.

Even though the board detection based on colored markers was quite fast on a phone and could analyze an input video stream at around 10 FPS on a Xperia Z1 smartphone, the detection robustness was not up to par with our expectations. In many cases the markers were not detected or there were false detections. But even when all markers were properly detected, the detection itself was not accurate enough, causing the stone detector to miss or incorrectly place the stones. The reasons for these problems were identified quickly, but some turned out to be fundamental ones.

1. Depending on the lighting conditions the exact color values were different and could lie outside the expected range.
2. Light reflections from ceiling lights (see Figure 3.6b) or sunlight resulted in white spots on the markers.
3. Unfocused parts of the picture blurred the markers and they were not fully detected.
4. The automatic white balance of the camera was initialized to different values.
5. The red color of the markers is too similar to the color of the board.
6. Stones on the corners cover parts of the markers.



(a) Near perfect detection of the markers. Detected areas are outlined in green.
(b) Bad detection of the right markers because of reflections.
(c) Many misdetections where the black lines blend into the board.

Figure 3.6: The effect of reflections and color range adjustments on the marker detection. The first two detections use the same range to detect the red color. On the right side the range is increased resulting in many misdetections.

Problems 4-6 would have been easily resolved by disabling the automatic white balance, choosing a color that differs more from the board (e.g. green or pink) and placing the markers outside of the board. But the fundamental problems 1-3 remained. Possible solutions to these problems, like using a calibration phase for the color or surrounding the complete border of the board with markers, were discussed. But after a consultation with a computer vision expert within EuTEC, it was advised to not rely on colors for exactly the given reasons. Markers with distinctive shapes that are placed outside of the board were proposed. By extracting features from the images and the markers it should then be relatively easy to detect the position of the markers and therefore also the board.

3.2.2 Marker Board Detection

Special markers are often used for augmented or virtual reality applications. These markers usually contain a high contrast pattern that is surrounded by a black border on a light color background [PSPS11; Fia04]. The pattern is used to distinct between markers and can be used to encode numbers, letters or other information [Hir08].

The markers in Figure 3.7 were designed so that they could be attached to the four corners outside of the board. They are big enough so that they still can be recognized in a 720p image from the camera. The pattern inside contains simple shapes that provide enough feature points for matching them against the reference image on the right. One marker is

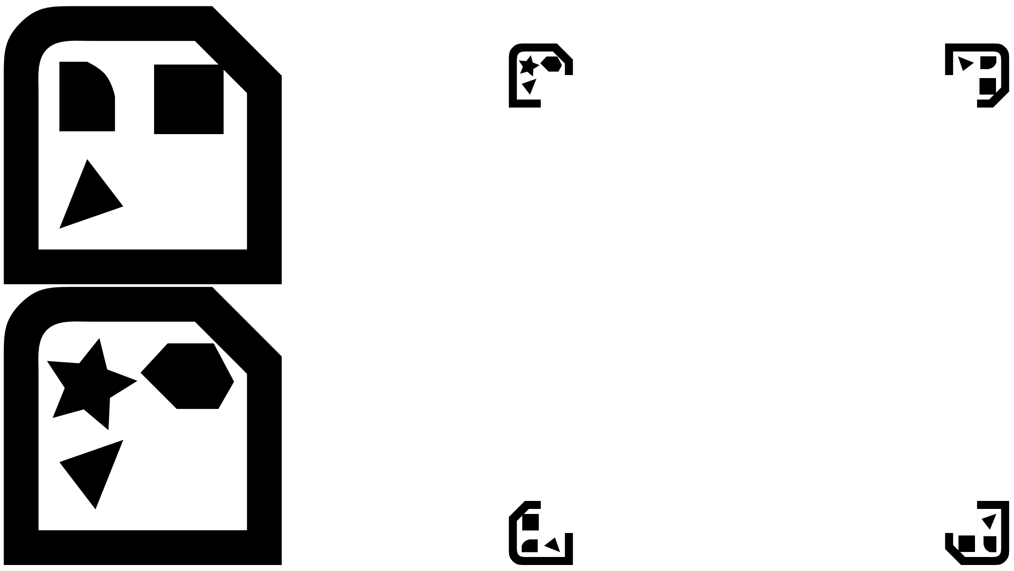


Figure 3.7: The two marker types used for the board detection. The bottom right quarter of the marker is empty as it is cut away so it can be attached to the Go board. On the right is the reference image that is matched against a real picture of the board.

used to mark the top-left position of the board to make sure the reference image is always matched with the same rotation. The marker reference image takes the exact dimensions of the printed markers and the dimensions of the used Go board into account making sure they the proportion matches the ones in the real world. This is important because the later perspective transformation may not find a good mapping otherwise. The marker images themselves have been printed on a regular sheet of paper, cut out and sticked to a 3D printed marker base designed for this use case. More detailed pictures can be found in Appendix A.2.

The reference image in Figure 3.7 is analyzed by a Speeded-Up Robust Features (SURF) detector, that extracts potentially interesting points, so called keypoints. Each keypoint is then described by a SURF vector that is calculated by taking the surrounding area into account. This feature vector is designed to describe a keypoint independent of the rotation and scaling of the source image [SZG+09].

The same feature extraction and description is then applied to the images from the camera, resulting in a second set of keypoints. The keypoints from the source image are matched against the keypoints from the reference image by calculating the Euclidean distance between them. The result of this matching can be seen in Figure 3.8, where keypoints are matched to zero or more similar keypoints in the reference picture.

Doing the perspective transformation, described in the previous chapter, requires four correctly matched keypoint pairs. However, because of the similarities between the

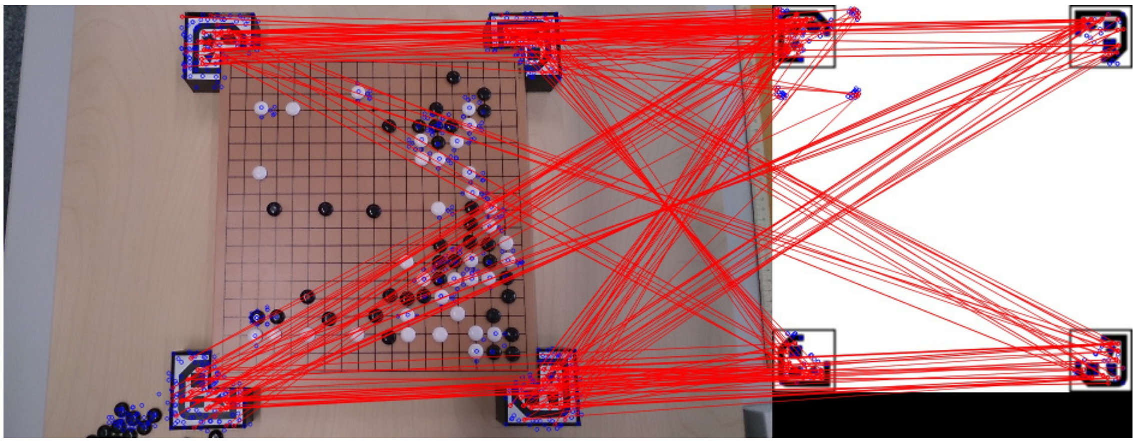


Figure 3.8: The keypoints of an image matched to the keypoints of the reference marker image.

individual markers (in fact, three are exactly the same), it is not directly possible to know which keypoints from the reference picture are correctly matched to the camera picture. A transformation can still be found by taking all matching pairs into account and applying a method called *random sample consensus* (RANSAC). RANSAC, like the name suggests, RANSAC selects a random sample of four from the matching keypoints and calculates the perspective transformation. It then checks how many other matches fit this transformation to estimate the quality of the transformation. The process is repeated until a good initial transformation is found and refined further by taking more points into account [Ope16b]. Matching keypoints that do not fit the resulting transformation are called *outliers*. OpenCV's `findHomography` is used to calculate the perspective transformation using RANSAC.

In Figure 3.9 the transformation, resulting from applying RANSAC to the matches from Figure 3.8, has been inversely applied to the reference marker image. Even though there are many outliers, the image is be perfectly projected.

3.2.3 Stone Detection

The stone detection algorithm expects an image that contains the complete board in order to correctly find the position of the detected stones. Finding the stones can be as easy as interpolating the stone positions between the top-left and bottom-right corners of the image and checking if the average color at a position is white, black or something else. But due to distortions in the transformed image and the fact that a human player rarely places a stone perfectly on an intersection, the actual position of a stone can be too far off the expected position. The average color at an interpolated position might then not be detected to be either a white or black stone.

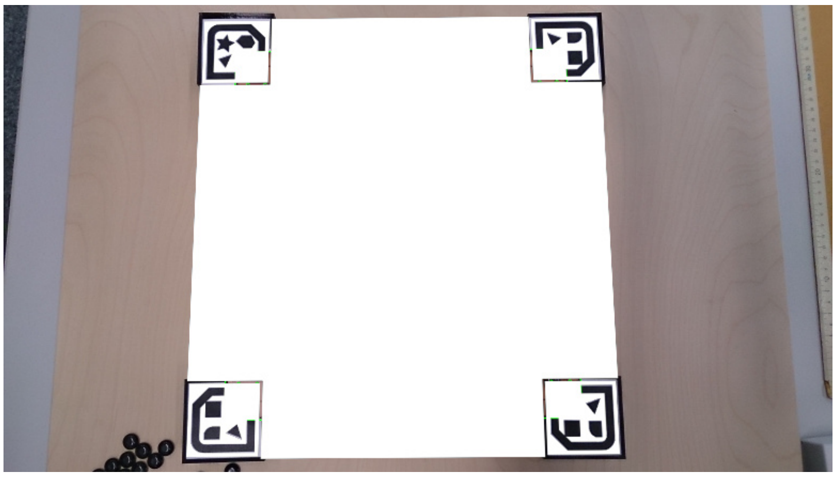
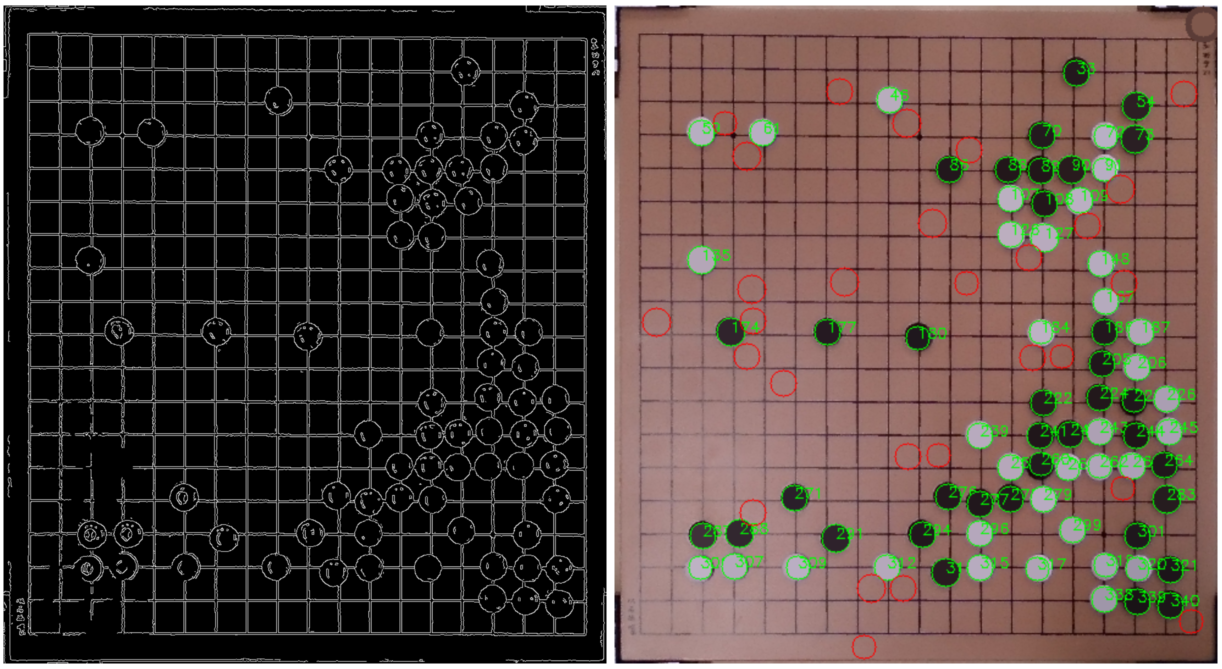


Figure 3.9: The reference marker projected into the source image.



(a) The binary image internally used by the Circle Hough Transform.

(b) Circles detected by the Hough Transform. The red circles are too far way from a possible stone position.

Figure 3.10: The Circle Hough Transform.

A solution to this problem is to find the stones based on their shape. By using a Circle Hough Transform, an image can be analyzed and even imperfect circles of different sizes detected. Internally this transformation operates on a binary image, resulting from an Canny edge detection algorithm as seen in Figure 3.10a [Ope16c]. To filter out possible false detections, the sizes and positions have to be checked. Too big or too small circles or circles not near an expected stone position, are probably no valid stones. The Circle Hough Transform is implemented using the `HoughCircles` function from OpenCV and the result can be seen in Figure 3.10b.

After the circle detection, the position of each circle is checked to be near a possible stone position and the circle is discarded if that is not the case. The expected stone positions are simply calculated by a 19×19 grid that is overlaid on the input. For possible stone candidates, the average color inside the circle is calculated and compared in HSV color space. For a white stone, the *saturation* component is checked to be below and the *value* component to be above a specific threshold. For a black stone, it is enough to check if the *value* component is low enough. See Figure 3.4c and 3.4d for an explanation of the HSV components.

Just checking the average color inside the detected circles has shown to to produce some false white stone detections, in areas where the board is very bright because of light reflections. By comparing the average color inside the detected circle to the average outside color and checking if the inside is darker than the outside, these false detections can be removed.

The white and black stone classification thresholds, as well as the brightness difference for white stones, is adjustable to allow the user to make adjustments at runtime from inside the Android App, if the detection does not produce the desired results.

3.3 Go Engine

Although the primary focus of the App is to highlight the network reduction benefits and the predictive capabilities of the original and reduced networks, a basic knowledge of the Go rules is required to properly create the input for the neural network. For the network input creation the Go Engine needs to be able to

- return the occupied positions for each player,
- calculate the liberties of a stone and
- detect the KO position.

But to improve the user experience and to be able play some moves against the neural networks, the engine also

- knows which player is next,
- is able to detect stones placed on illegal positions,
- and knows which stones have been captured and need to be removed from the board.

The Go Engine is built on top of GnuGo, which is an Open Source Go AI. GnuGo also implements all the above requirements and has support for interfacing with different frontends and user interfaces through a standardized Go Text Protocol or a C-API. GnuGo is compiled as a ARMv7[1] native shared library and a custom JNI[2] is used for changing and retrieving the state the Go board and information about liberties or legality of a move.

3.4 Move Predictor

The Move Predictor is the interface to the neural networks, which are supplied as shared libraries. In Section 6.2 these shared neural network libraries are explained in more detail. The libraries are dynamically loaded through a JNI when the application starts. The JNI returns pointers to the loaded libraries that need to be passed to the JNI again when making a function call. This pattern treats the networks as plugins which can be loaded and unloaded during the runtime of the application. It also allows network implementations to be provided as a separate Android App although that functionality would require more work. Each loaded network is encapsulated within a Java class that handles creating the network input, provided by the Go Engine, and formats the network output to an easily accessible 19×19 Java array containing the prediction probabilities per position.

3.5 GUI

The App user interface is inspired by the camera App that comes pre-installed on Sony Xperia smartphones. When the App is started, a splash screen is displayed during initialization of the other components. After initialization, the App switches to the main screen which can be seen in Figure 3.11. The main screen contains the camera preview, a shutter button and a button to access the settings. Pressing the shutter button takes a picture and transitions to the result screen. The picture is analyzed in the background with the Board and Stone Detector and upon completion the Go Engine is updated with the stone positions. The Go Engine then informs the result screen about the new stones positions and initiates the Move Predictor. When the prediction results are available the, result screen is updated again and will look like Figure 3.12.

The buttons on the result screen allow the user to go back to the main screen, save the taken picture to the SD card and access the App settings. The “Game Info” part on the right

[1]ARMv7 is the CPU architecture of the used Android phones
[2]Java Native Interface

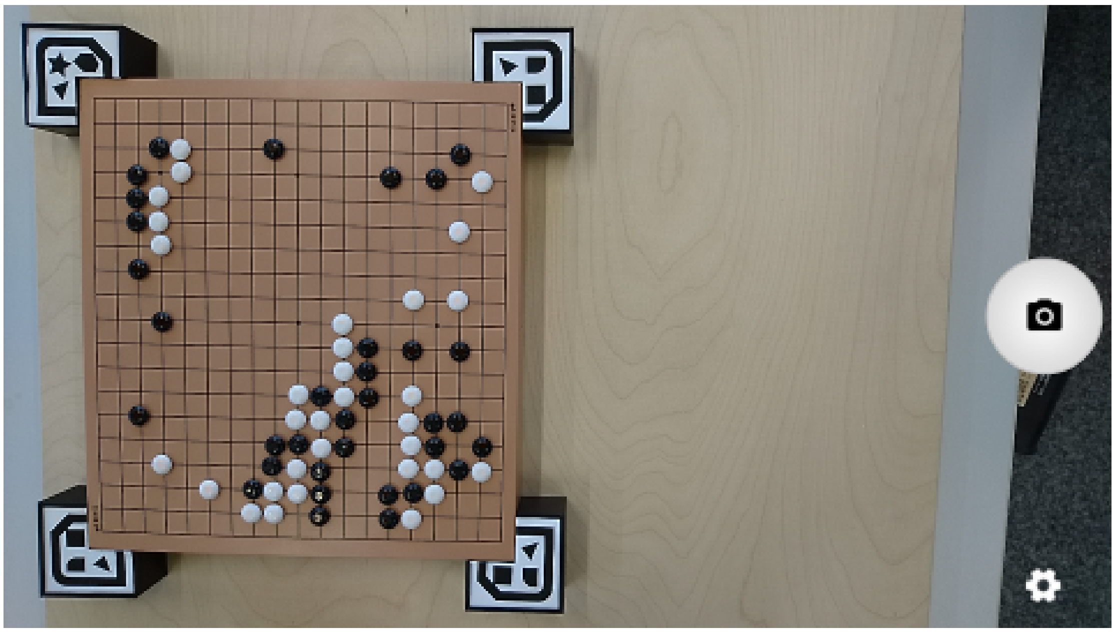
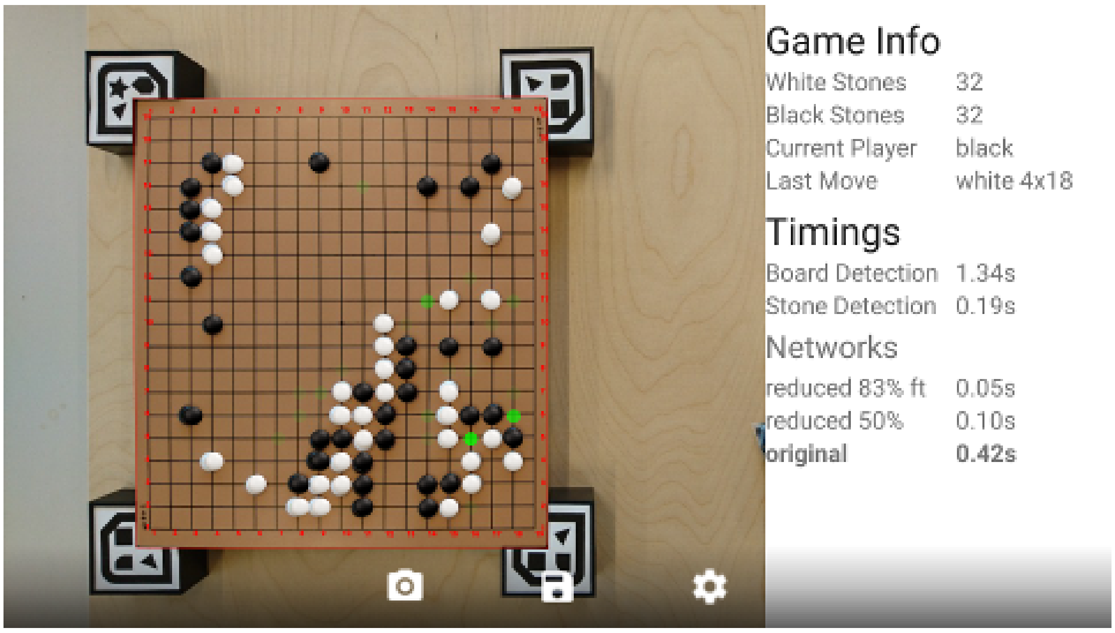
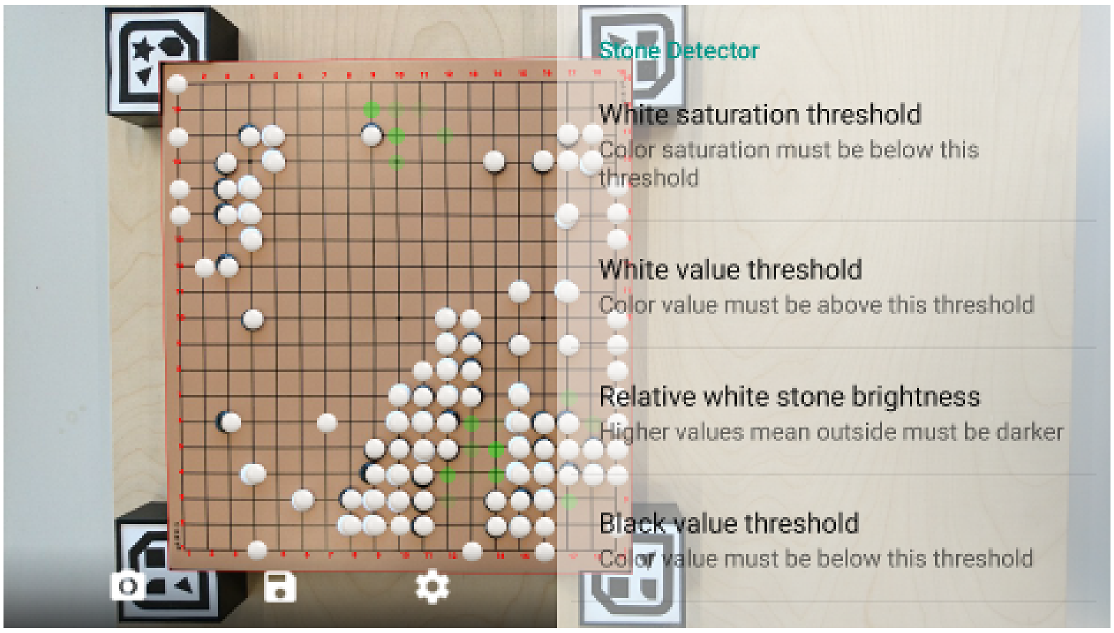


Figure 3.11: Main screen after starting the App.



```
Game Info
White Stones      32
Black Stones      32
Current Player    black
Last Move         white 4x18
Timings
Board Detection   1.34s
Stone Detection   0.19s
Networks
reduced 83% ft    0.05s
reduced 50%       0.10s
original          0.42s
```

Figure 3.12: Analyzed board with the predictions in green, game and timing information. Better moves are more opaque.



Stone Detector

White saturation threshold
Color saturation must be below this threshold

White value threshold
Color value must be above this threshold

Relative white stone brightness
Higher values mean outside must be darker

Black value threshold
Color value must be below this threshold

Figure 3.13: The settings screen of the App

side displays the number of stones on the board, the current player which should place the next stone and the move done by the previous player. Below the game information, the App displays the time it took to detect the board and stones and the prediction time of the loaded networks. By tapping on the name of a network it is highlighted in bold and the move predictions of this network are displayed.

The settings screen shown in Figure 3.13, that is accessed by touching the cog button, allows the user to tweak various parameters of the Stone Detection. Changing the parameters will immediately update the stone detection and to see the result, the background was made slightly transparent. The adjustable parameters are explained in detail in Section 3.2.3.

4 Convolutional Deep Neural Networks

In this chapter Convolutional Deep Neural Networks (CDNNs) are introduced. It is however no general introduction into neural networks and skips over many details. A good introduction into neural networks can be found in [Bis06].

Neural networks find their origin by attempting to recreate the way biological systems process information, where neurons are connected and interchange information when activated. Since then neural networks have partly moved away from sticking to the biological model and new types of networks emerged. For pattern recognition CDNNs have shown impressive results in various domains and achieved state of the art results for image classification tasks [SZ14].

Neural networks are separated by layers and besides an input and output layer, the network can contain an arbitrary number of so called hidden layers. Input data is passed top-down from the input layer, through the hidden layers, to the output layer, which then contains the prediction or output of the network. This is also called a forward pass.

CDNNs introduce the convolution layers as hidden layer type, which are commonly placed before any single affine layer [SZ14]. A typical network structure for a CDNN is shown in Figure 4.1. In CDNNs the data, that is passed though the network, takes the form of a multi-dimensional matrix. The first dimension is called the *feature map*, or when talking about the input/output of a layer the *input/output feature map* or just *input/output map*. The remaining dimensions are the contents of the feature map. The sizes of the dimensions may change, when the data is passed though a layer.

Considering an image classification DCNN, that maps images to the classes “whale”, “dog” and “cat”. The input could be a single feature map with a RGB image. In this case the second dimension could be the height of the image, the third the width and the last dimension would be the color channel, so a $1 \times height \times width \times 3$ matrix. The output may be a 1×3 vector with the probability distribution over the three classes.

4.1 Notation

This and the following chapter will use the following notation:

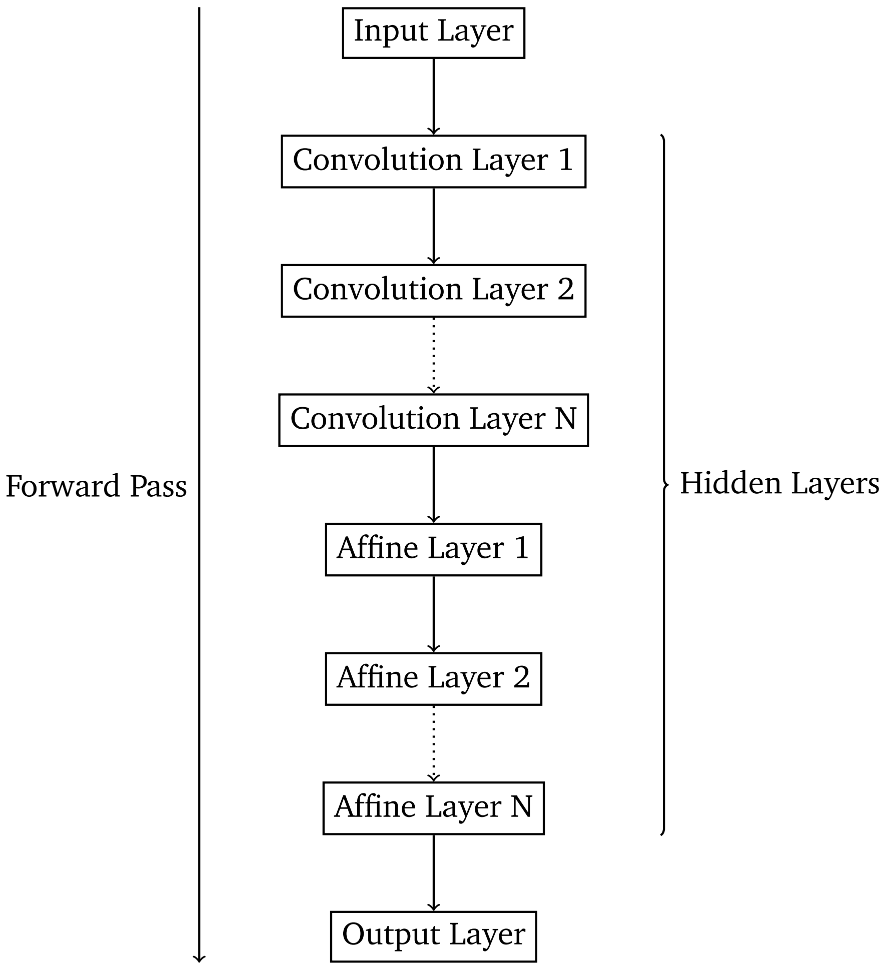
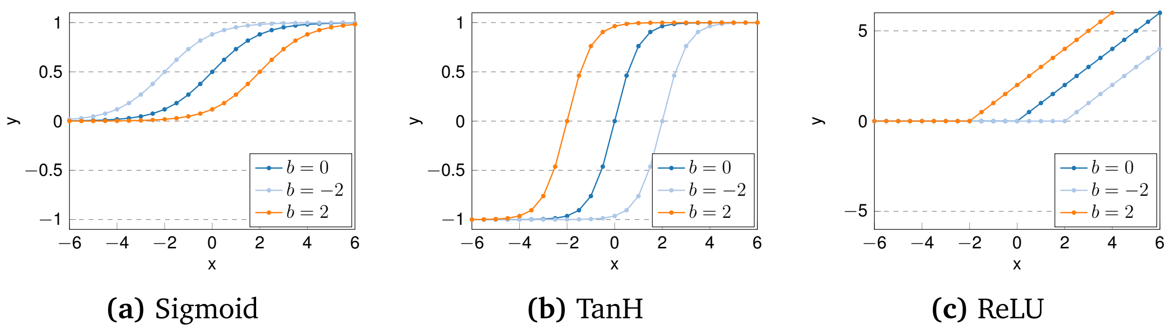


Figure 4.1: A feed-forward CDNN.

- $\mathcal{A}$ is a tensor and the element at (i, j, k) is a_{ijk}
- $\mathbf{A}$ is a matrix and the element at (i, j) is a_{ij}
- $\mathbf{a}$ is a vector and the ith element is a_i
- Indices range from 1 to their capital version, so $i = 1, \dots, I$

For simplification of some formulas the following is defined additionally:

- The letter X is used to denote the input of a layer
- The Letter Y is used to denote the output of a layer
- I_m, O_m are the number of input and output maps
- I_h, O_h, K_h are the heights of the input, output and kernel
- I_w, O_w, K_w are the widths of the input, output and kernel



(a) Sigmoid **(b)** TanH **(c)** ReLU

Figure 4.2: Different non-linear activation functions used in CDNNs and the effect of the bias b.

- $I^D = I_m \cdot I_h \cdot I_w$ is the number of elements in the input
- $O^D = O_m \cdot O_h \cdot O_w$ is the number of elements in the output
- $K^D = K_h \cdot K_w$ is the number of elements in the kernel

4.2 Hidden Layer

A hidden layer in a CDNN is a non-linear function y, which receives the input x and takes the following form:

$$y(x) = f\left(\phi(x, w) + b\right)$$

f is a non-linear function, called the activation function, ϕ is the layer function, w are the weights for the layer function and b is the bias. Together the weights and the bias form the parameters of a layer.

An activation function f is an important part of a neural network, as it breaks the linearity between the different layers. Without a non-linear activation function, layers that have a linear layer function, may be reduced to a single layer.

Commonly used activation functions are:

Sigmoid: $f(x) = \frac{1}{1+e^{-x}}$;

TanH: $f(x) = \frac{2}{1+e^{-2x}} - 1$;

ReLU: $f(x) = max(0, x)$;

Figure 4.2 shows the of these functions on the output of a simple layer function $\phi(x, w) = x$ and different values for the bias b, which shifts the resulting curves to the left or right.

4.3 Training

The training of a neural network is the process of finding values for the layer function weights w and the bias b that produce the desired outputs for as many inputs as possible. Often these parameters are initialized to random values and then gradually refined by comparing the output of the network to the expected output, calculating an error value and then updating the parameters to reduce the error. The parameters are usually optimized by a Gradient Descent, which chooses the update to make a small step in the direction of the negative gradient of the error function [Bis06]. The parameters are updated for all layers in the network by propagating the error backwards in a process known as *backpropagation*.

The backpropagation and optimization process is repeated until all samples from a training set have been processed, which is when one *epoch* has passed. Most of the time, one epoch is not enough and the whole process is repeated until the error stops improving. A training set may contain many million of samples and updating the parameters after each sample can be a costly operation. A common solution is to use *mini-batches* to process many samples at once and only update the parameters after each batch. The size of these *mini-batches* is referred to as the *batch size*.

The granularity of the parameters updates can be influenced by adjusting the *learning rate* of the training. A lower rate decreases the step size of the Gradient Descent and increases the training time. A higher rate has the opposite effect with the disadvantage that it might not find the optimal parameters. It is therefore advisable to adjust the *learning rate* during the training process by lowering it after or even during an epoch.

4.4 Affine Layer

An affine, or fully connected, layer combines each input element with each output element as shown in Figure 4.3. Every input x_i is weighted with a separate weight w_{ji} and added to the output y_j. This way an affine layer can map an arbitrary number of inputs to an arbitrary number of outputs. This is often used to make the final decision in a CDNN, which assigns probabilities to the output classes.

A single output y_j can be calculated with the following formula:

$$y_j = f\left(\sum_{i=1}^{I} w_{ji} \cdot x_i + b_j\right)$$

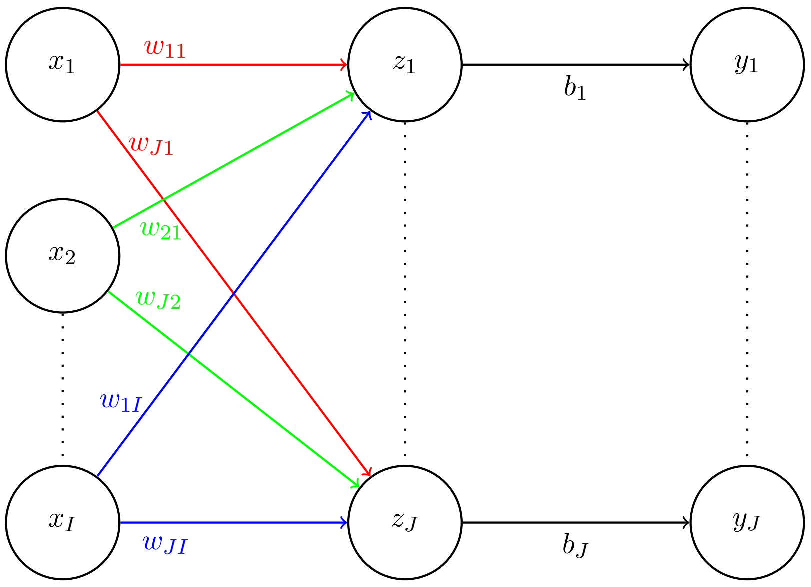


Figure 4.3: An affine layer with I inputs and J outputs. Each input i is weighted with w_{ji} and added to output j.

Which, for a one dimensional input, can be represented as a vector multiplication:

$$y_j = f\left(\begin{pmatrix} w_{j1}, w_{j2}, \dots, w_{JI} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_I \end{pmatrix} + b_j\right)$$

The complete output vector can therefore be calculated as a matrix multiplication:

$$\begin{aligned} y &= f\left(\begin{pmatrix} w_{11} & \dots & w_{1I} \\ \vdots & \ddots & \vdots \\ w_{J1} & \dots & w_{JI} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_I \end{pmatrix} + \begin{pmatrix} b_1 \\ \vdots \\ b_J \end{pmatrix}\right) \\ &= f\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right) \end{aligned} \tag{4.1}$$

The last formula is similar to the formula of affine transformation in geometry, which gives us the name for this layer. To handle multi-dimensional input, the input can be reshaped into a long vector, analogous for the output that can be reshaped into the desired shape. It therefore makes sense to save the weights $\mathbf{W}$ as a $O^D \times I^D$ matrix.

4.5 Convolution Layer

Like the name implies, a convolution layer applies a convolution operation to the input and is often used when handling image data. Common convolution operations on images

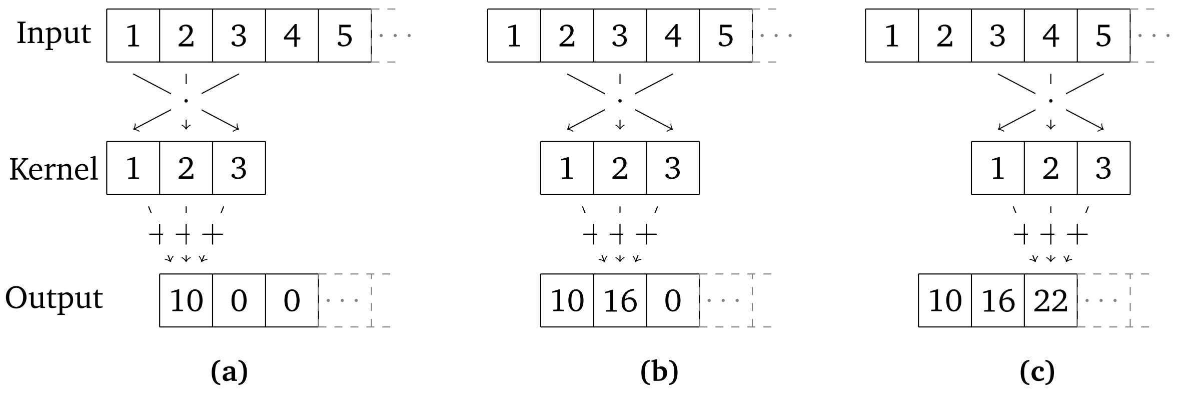


Figure 4.4: Three steps of a 1D convolution with a kernel of size 3.

include blurring and sharpening the picture. The operation is done by applying a convolution kernel to each pixel of the input. The convolution kernel linearly combines the current and neighboring pixels into a new pixel.

The idea of a convolution layer is, that nearby pixels are more strongly correlated than pixels far away and therefore local features are extracted. These layers also handle data, that is invariant to rotation, better than affine layers [Bis06]. Rotational invariant data can for example be picture of a dog, which, when rotated by 180°, is still a picture of a dog.

The convolution operation slides a flipped kernel over the input, multiplies each input element with the corresponding flipped kernel element and sums the results to get one output element as shown in Figure 4.4.

For a 1D convolution using a 1D kernel $\mathbf{w} \in \mathbb{R}^{K^D}$, this can be written as:

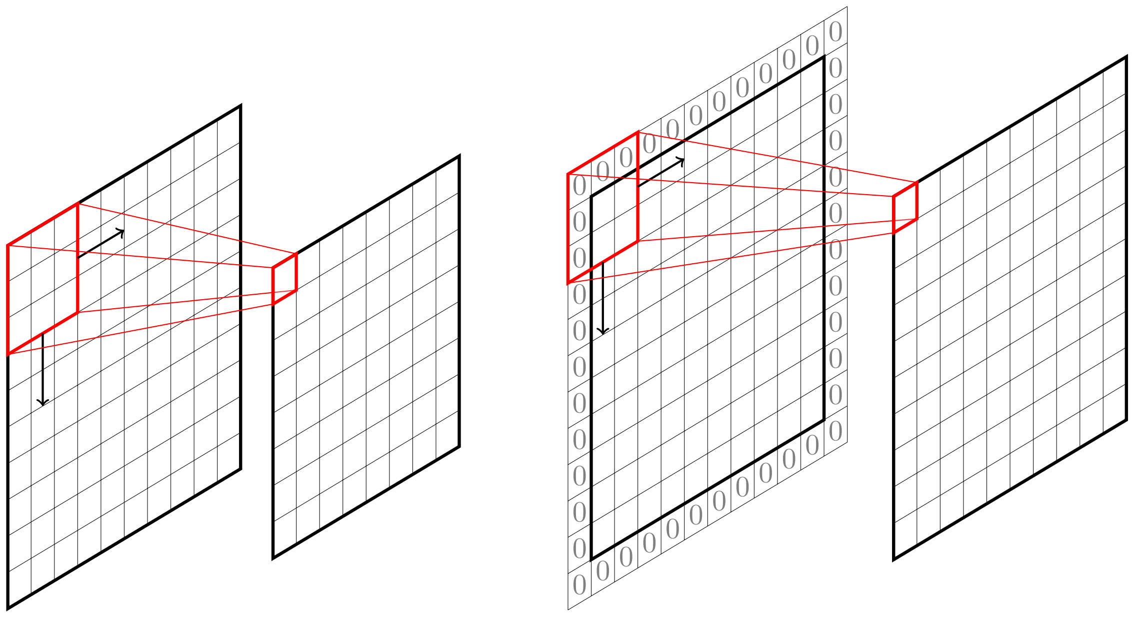
$$y_j = f\left(\sum_{k=1}^{K^D} x_{j-(k-K^D+1)} \cdot w_{(K^D-k+1)} + b_j\right) \tag{4.2}$$

The indexing of $\mathbf{w}$ will cause the kernel to be flipped. Using the convolution operator $*$ this can be simplified to:

$$\mathbf{y} = f\left(\mathbf{x} * \mathbf{w} + \mathbf{b}\right)$$

Input and output for convolution layers is separated into feature maps and each map combination is convoluted with a separate kernel. All input maps have the same dimensions, the same applies to the output maps. The formula to calculate the result for an output map is:

$$\mathbf{Y}_o = f\left(\sum_{i=1}^{I_m} \mathbf{X}_i * \mathbf{W}_{io} + \mathbf{b}_o\right) \tag{4.3}$$



(a) Only valid input is used, output is smaller than input.

(b) Input is implicitly padded with zeros, output has the same shape.

Figure 4.5: A 10×10 input convoluted with a 3×3 kernel using different border handling.

where $\mathbf{W}_{io}$ is the kernel for input map i and output map o.

The size of a output maps depends on the size of the input map and the border handling as visualized in Figure 4.5. On the left side the *valid* border mode handling is used, which means only outputs where the kernel does not need elements outside of the input are returned. See (4.2) where the calculation of y_1 would need a value for x_0. This reduces the output map dimension by half the kernel size -1. This can be avoided by assuming values outside the input are zero. When the input and output dimensions are equal then this is called the *same* border mode.

For the later formulas is important to know that the convolution is

- commutative: $\mathbf{x} * \mathbf{y} = \mathbf{y} * \mathbf{x}$,
- associative: $\mathbf{x} * (\mathbf{y} * \mathbf{z}) = (\mathbf{x} * \mathbf{y}) * \mathbf{z}$,
- distributive: $\mathbf{x} * (\mathbf{y} + \mathbf{z}) = \mathbf{x} * \mathbf{y} + \mathbf{x} * \mathbf{z}$.

5 Deep Neural Network Reduction

In this chapter methods for the reduction of affine and convolution layers are introduced and explained. The methods are based on two matrix/tensor decomposition approaches (SVD and CP), that allow the original weights of a layer to be represented as a low rank approximation with fewer parameters. Interestingly, it is possible to directly work with reduced number of parameters and no reconstruction of the original weights is needed. This not only saves space to store the network on disk, but also reduces memory consumption and computation costs.

The goal of the reduction methods is to produce a smaller network, that uses less memory and computes results faster, whilst maintaining a similar quality as the original network. Without the introduced techniques one would need to

- reduce the number of layers,
- reduce the size of the layers or
- reduce the size of the convolution kernels

in order to produce such a smaller network. This has the negative side effect that the new network needs to be trained from scratch, which can be very time consuming. The smaller network architecture can also make it harder to achieve the original accuracy.

5.1 Number of Parameters and Operations

Before introducing the reduction methods, the size and computational costs of the original affine and convolution layers should be known, to make comparisons possible. The size of a layer is simply measured as the number of parameters required to store the weights. This excludes the bias because it will not be changed by any of the introduced methods. For the computational cost, the number of Floating Point Operations (FLOPs), which are required for one forward pass, are counted. Adding the bias is also excluded and a multiplication with an additional addition is counted as one operation to simplify the formulas shown in Table 5.1.

An affine layer combines each input element with each output element by multiplying it with a weight specific for this combination and adding it to the corresponding output

Type	**# Parameters**	**FLOPs**
Affine	$I_m \cdot I_h \cdot I_w \cdot O_m \cdot O_h \cdot O_w$	$I_m \cdot I_h \cdot I_w \cdot O_m \cdot O_h \cdot O_w$
Convolution	$I_m \cdot O_m \cdot K_h \cdot K_w$	$I_m \cdot O_m \cdot O_h \cdot O_w \cdot K_h \cdot K_w$

Table 5.1: Formulas to calculate the number of parameters and FLOPs (without bias).

$$\underbrace{\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ a_{31} & a_{32} & \dots & a_{3n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{n2} & \dots & a_{mn} \end{pmatrix}}_{\mathbf{A}} = \underbrace{\begin{pmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1m} \\ u_{21} & u_{22} & u_{23} & \dots & u_{2m} \\ u_{31} & u_{32} & u_{33} & \dots & u_{3m} \\ \vdots & \vdots & \ddots & \vdots & \\ u_{m1} & u_{m2} & u_{m3} & \dots & u_{mm} \end{pmatrix}}_{\mathbf{U}} \cdot \underbrace{\begin{pmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ \vdots & \vdots & 0 & \sigma_n \\ 0 & 0 & 0 & 0 \end{pmatrix}}_{\mathbf{\Sigma}} \cdot \underbrace{\begin{pmatrix} v_{11} & v_{12} & \dots & v_{1n} \\ v_{21} & v_{22} & \dots & v_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ v_{n1} & v_{n2} & \dots & v_{nn} \end{pmatrix}}_{\mathbf{V}_T}$$

Figure 5.1: The SVD of $\mathbf{A}$ into $\mathbf{A} = \mathbf{U\Sigma V}^T$ with $n < m$.

element. Assuming the input of the layer has the dimension $I_m \times I_h \times I_w$ and the output has the dimension $O_m \times O_h \times O_w$, then the number of parameters and FLOPs for an affine layer is simply calculated by multiplying the number of input and output elements.

A convolution layer has a separate $K_h \times K_w$ kernel that needs to be stored for each input and output map combination. $K_h \cdot K_w$ multiplications and additions are required to calculate a single output element for one output map. For the total number of operations this has to be multiplied with the number of output elements and the number of kernels.

5.2 Affine Layer Reduction

The affine layer reduction method is based on a low rank approximation using a Singular Value Decomposition (SVD). Using the SVD we can decompose a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ into $\mathbf{A} = \mathbf{U\Sigma V}^T$ where $\mathbf{U}$ and $\mathbf{V}^T$ are $m \times m$ and $n \times n$ unitary matrices, $\mathbf{\Sigma}$ is a $m \times n$ diagonal matrix of the positive singular values σ ordered by magnitude[Ste93]. The decomposition is visualized in Figure 5.1.

Is $\mathbf{u}_i$ the ith column of $\mathbf{U}$ and $\mathbf{v}_i^T$ the ith row of $\mathbf{V}^T$ then we can calculate $\mathbf{A}$ with a series of weighted matrix multiplications

$$\begin{aligned} \mathbf{A} &= \mathbf{U\Sigma V}^T \\ &= \sum_{i=1}^{R} \sigma_i \mathbf{u}_i \mathbf{v}_i^T \qquad\qquad R = min(m, n) \end{aligned}$$

where $R = min(m, n)$ is the upper bound of the rank. The singular values σ_i can be seen as the significance of $\mathbf{u}_i\mathbf{v}_i^T$ contributing to $\mathbf{A}$. An approximation of $\mathbf{A}$ is created by setting some singular values to zero.

By choosing $R < rank(\mathbf{A}) \leq min(m, n)$ we can create a low rank approximation

$$\begin{aligned}\tilde{\mathbf{A}} &= \sum_{i=1}^{R} \sigma_i \mathbf{u}_i \mathbf{v}_i^T && R < rank(\mathbf{A}) \\ &= \tilde{\mathbf{U}}\tilde{\mathbf{V}}^T && (\mathbf{U} \text{ and } \mathbf{\Sigma} \text{ combined})\end{aligned}$$

that minimizes Frobenius norm of the approximation difference $\tilde{\mathbf{A}} - \mathbf{A}$ [Ste93]:

$$\left\|\tilde{\mathbf{A}} - \mathbf{A}\right\|_F^2 = \sqrt{\sum_{i,j=1}^{m,n} \tilde{a}_{ij}^2 - a_{ij}^2}$$

So in simple terms, by cutting off $\mathbf{U}$ after R columns, σ after R singular values and $\mathbf{V}^T$ after R rows we create a low rank approximation of $\mathbf{A}$ that minimizes the Frobenius norm of the parameter differences.

By applying the SVD to the $O^D \times I^D$ layer weights from Equation 4.1 and doing a low rank approximation of $\mathbf{W}$, we can approximate the weights and the affine layer output by

$$\begin{aligned}\mathbf{y} &= f\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right) \\ &= f\left(\mathbf{U}_W \mathbf{\Sigma}_W \mathbf{V}_W^T \mathbf{x} + \mathbf{b}\right) \\ &\approx f\left(\tilde{\mathbf{U}}_W \tilde{\mathbf{V}}_W^T \mathbf{x} + \mathbf{b}\right) && (\mathbf{U} \text{ and } \mathbf{\Sigma} \text{ combined}) \qquad (5.1)\end{aligned}$$

where $\tilde{\mathbf{U}}_W$ is a $O^D \times R$ matrix and $\tilde{\mathbf{V}}_W^T$ is a $R \times I^D$ matrix. The approximation therefore has the following number of parameters and FLOPs:

$$\begin{aligned}params &= O^D \cdot R + R \cdot I^D \\ &= R(O^D + I^D) \\ flops &= R(O^D + I^D)\end{aligned}$$

5.3 Convolution Layer Reduction

Convolution layers are often responsible for the majority of FLOPs in a CDNN [JVZ14]. For larger networks it is critical to decrease these layers if a deployment in the mobile domain is planned. But even outside the mobile domain the reductions are worthwhile to explore.

Recent literature quite successfully reduced networks with convolution layers by using a low rank approximation of the kernel weights. In [JVZ14] a 4.5$\times$ speedup is achieved with only a 1% drop in accuracy in a character recognition task. In [LGR+14] this result was topped by achieving a 8.5$\times$ speedup on the same network. And in [KPY+15] they measured the energy consumption improvement of a Samsung Galaxy S6 smartphone to be higher than the FLOPs reduction rate, which they attribute to an improved CPU cache efficiency. The good reduction ratios that are responsible for the these speedups are traced back to high redundancy of the kernels for different maps and inside the kernel itself [JVZ14].

5.3.1 Singular Value Decomposition

The SVD, which was introduced in the previous section, can also be used to reduce the weights of a convolution layer. The SVD however can only operate on two-dimensional matrices, yet the weights for a convolution layer are often stored inside a $I_m \times O_m \times K_h \times K_w$ tensor, which makes accessing a kernel for a specific input map and output map easy. The two dimensions for the kernel may be flattened, giving us a $I_m \times O_m \times K^D$ tensor.

As the the SVD requires 2D matrices the kernel weights are separated into I_m matrices $\mathbf{W}_0, \cdots \mathbf{W}_I$ with the dimensions $O_m \times K^D$, where each row represents a kernel $\mathbf{w}_{io}$ to convolve with input map i and add the result to output map o. The matrices are separately approximated using a SVD low rank approximation

$$\begin{aligned} \mathbf{W}_i &= \mathbf{U}_i \mathbf{\Sigma}_i \mathbf{V}_i^T && \text{(SVD factorization)} \\ &\approx \tilde{\mathbf{U}}_i \tilde{\mathbf{V}}_i^T && \text{(approximation with rank } R\text{)} \end{aligned}$$

where $\tilde{\mathbf{U}}_i$ is a $O_m \times R$ and $\tilde{\mathbf{V}}_i^T$ is a $R \times K^D$ matrix.

A single kernel $\mathbf{w}_{io}$ is now approximated by R basis kernels $\mathbf{B}_i = \tilde{\mathbf{V}}_i^T$.

$$\begin{aligned} \mathbf{w}_{io} &\approx \tilde{\mathbf{u}}_{io} \tilde{\mathbf{V}}_i^T && \text{(one row taken from } \tilde{\mathbf{U}}_i\text{)} \\ &= \tilde{u}_{io} \mathbf{B}_i \\ &= \sum_{r=1}^{R} \tilde{u}_{io}^{(r)} b_{ri} \end{aligned}$$

where $\tilde{u}_{io}^{(r)}$ is the rth element of the ith row of $\tilde{U}_i$ and b_{ri} the rth row of B_i.

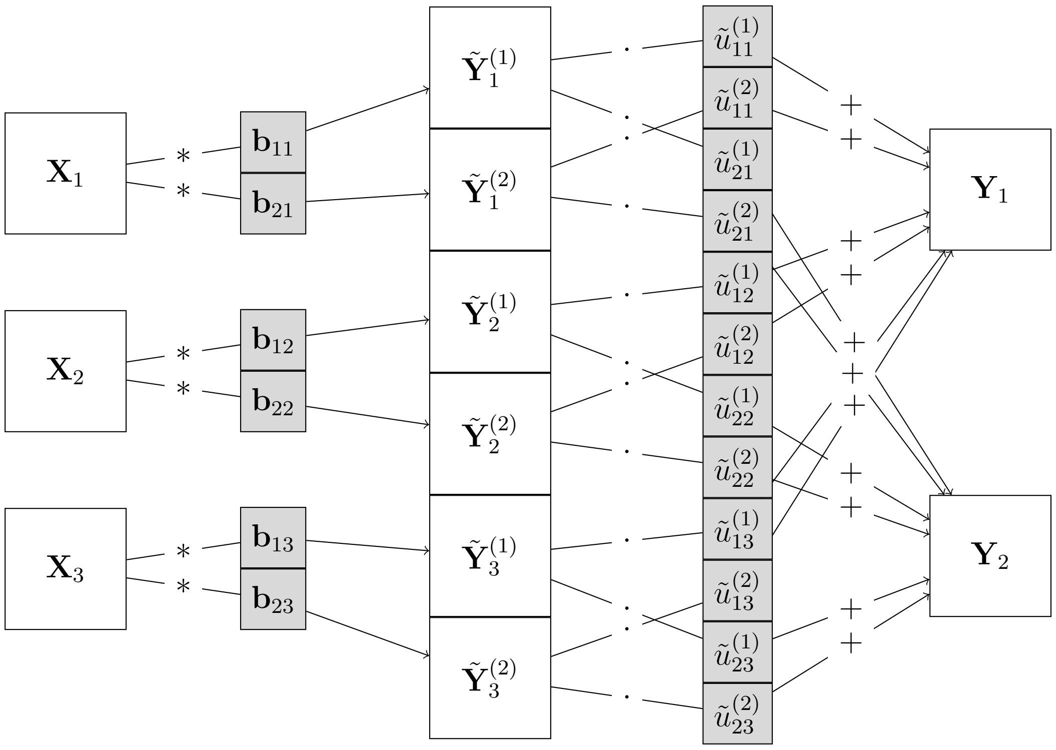


Figure 5.2: A SVD convolution with 3 input maps, 2 output maps and $R = 2$. Each input map is convolved with an intermediate kernel b_{oi} and then decompressed with the output factors $\tilde{u}_{oi}^{(r)}$ into the corresponding output maps.

The whole output map Y_o can be computed like:

$$\begin{aligned} Y_o &= f\left(\sum_{i=1}^{I_m} X_i * W_{io}\right) \\ &\approx f\left(\sum_{i=1}^{I_m} X_i * \left(\sum_{r=1}^{R} \tilde{u}_{io}^{(r)} b_{ri}\right)\right) \\ &= f\left(\sum_{i=1}^{I_m}\sum_{r=1}^{R} \tilde{u}_{io}^{(r)} \left(X_i * b_{ri}\right)\right) \end{aligned} \tag{5.2}$$

The linearity of the convolution allows us to reorder the operations resulting in (5.2). Inspecting this formula shows, that the convolution operation itself is independent from a specific output map. The index o is not required to calculate all results for $X_i * b_{ri}$. This allows the convolution results to be reused between output maps.

An efficient implementation would convolve each input X_i with R basis kernels b_{ri} and then weight each result with the factor $\tilde{u}_{oi}^{(r)}$ before adding it to the output map. This is

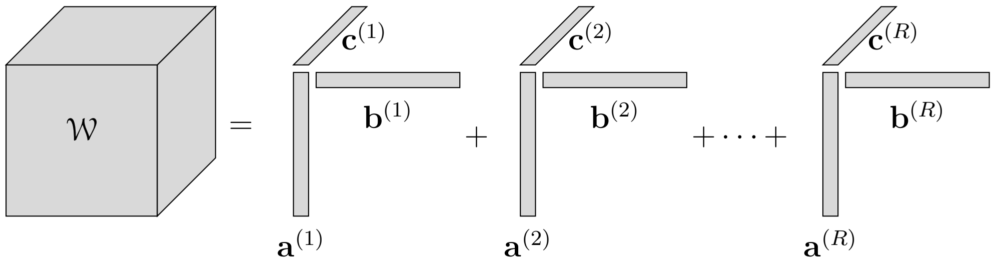


Figure 5.3: A 3D CP decomposition of $\mathcal{W}$ into R rank-one tensors [KB09].

visualized in Figure 5.2. By this the number of convolutions is reduced from $I_m \cdot O_m$ to $I_m \cdot R$ if $R < O_m$. The total number of parameters and FLOPs are:

$$\begin{aligned} params &= \underbrace{R \cdot I_m \cdot K^D}_{\text{basis kernels}} + \underbrace{R \cdot I_m \cdot O_m}_{\text{output factors}} \\ &= R \cdot I_m \cdot (O_m + K^D) \\ flops &= R \cdot I_m \cdot O_h \cdot O_w \cdot (O_m + K^D) \end{aligned}$$

5.3.2 Tensor Rank Decomposition

Another method of reducing the weights for a convolution layer is by approximating them using a Tensor Rank Decomposition. A tensor is a multidimensional array where the order of a tensor denotes the number of dimensions. This section introduces the Canonical Polyadic Decomposition (aka CANDECOMP/PARAFAC), which will be called CP decomposition from here on. This method can be seen as a higher-order extension of the SVD [KB09].

The CP decomposition has the advantage that it can approximate the full third-order $I_m \times O_m \times K^D$ weights instead of operating on a set of separated $O_m \times K^D$ matrices like the previous SVD low rank approximation. This promises a better compression compared to the SVD approach as this method allows to take advantage of correlations between kernels from different input maps.

By applying the CP decomposition, a N-order tensor $\mathcal{X}$ is decomposed into a series of rank-one tensors where the sum of a subset of these tensors, is an approximation of $\mathcal{X}$. A N-order rank-one tensor is a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ that can be computed by the outer product of N vectors and is written as:

$$\mathcal{X} = \mathbf{a}^{(1)} \circ \mathbf{a}^{(2)} \circ \cdots \circ \mathbf{a}^{(N)} \qquad \text{(with } \mathbf{a}^{(i)} \in \mathbb{R}^{I_i}\text{)}$$

The outer product $\circ$ means that to compute element $x_{i_1 i_2 \ldots i_N}$ the corresponding elements of all vectors are multiplied:

$$x_{i_1 i_2 \ldots i_N} = a^{(1)}_{i_1} a^{(2)}_{i_2} \ldots a^{(N)}_{i_N}$$

Unfortunately, finding the best CP approximation for a given rank is an ill-posed problem. The CP decomposition cannot be easily computed and a possible solution needs to be found and optimized [dSL08; LGR+14]. Depending on the optimization, this may result in different approximations for the same input and rate.

With the above definitions the formula for the approximated kernel weights $\mathcal{W} \in \mathbb{R}^{I_m \times O_m \times K^D}$ is

$$\mathcal{W} \approx \sum_{r=1}^{R} \mathbf{a}^{(r)} \circ \mathbf{b}^{(r)} \circ \mathbf{c}^{(r)} \,.$$

Where $a^{(r)} \in \mathbb{R}^{I_m}$, $b^{(r)} \in \mathbb{R}^{O_m}$ and $\mathbf{c}^{(r)} \in \mathbb{R}^{K^D}$. A visualization can be seen in Figure 5.3.

A single kernel $\mathbf{W}_{io}$ is approximated multiplying the $\mathbf{c}$ vector with the ith element of $\mathbf{a}$ and oth element of b for each rank:

$$\mathbf{W}_{io} = \sum_{r=1}^{R} a_i^{(r)} b_o^{(r)} \mathbf{c}^{(r)} \tag{5.3}$$

To compute output map $\mathbf{Y}_o$, (5.3) is inserted into the convolution (4.3) and the operations reordered in the following way:

$$\begin{aligned}
\mathbf{Y}_o &= f\left(\sum_{i=1}^{I_m} \mathbf{X}_i * \mathbf{W}_{oi}\right) \\
&\approx f\left(\sum_{i=1}^{I_m} \mathbf{X}_i * \left(\sum_{r=1}^{R} a_i^{(r)} b_o^{(r)} \mathbf{c}^{(r)}\right)\right) \\
&= f\left(\sum_{i=1}^{I_m}\sum_{r=1}^{R} a_i^{(r)} b_o^{(r)} \left(\mathbf{X}_i * \mathbf{c}^{(r)}\right)\right) \\
&= f\left(\sum_{r=1}^{R} b_o^{(r)} \sum_{i=1}^{I_m} a_i^{(r)} \left(\mathbf{X}_i * \mathbf{c}^{(r)}\right)\right) \\
&= f\left(\sum_{r=1}^{R} b_o^{(r)} \left(\sum_{i=1}^{I_m} a_i^{(r)} \left(\mathbf{X}_i * \mathbf{c}^{(r)}\right)\right)\right) \\
&= f\left(\sum_{r=1}^{R} b_o^{(r)} \overbrace{\left(\underbrace{\left(\sum_{i=1}^{I_m} a_i^{(r)} \mathbf{X}_i\right)}_{\tilde{\mathbf{X}}_r} * \mathbf{c}^{(r)} \right)}^{\tilde{\mathbf{Y}}_r}\right)
\end{aligned} \tag{5.4}$$

From (5.4) we can see that there is an efficient way to calculate the result by

1. compressing $\mathbf{X}_i$ into R intermediate input maps $\tilde{\mathbf{X}}_r$,
2. convolving them with $\mathbf{c}^{(r)}$ to produce R intermediate output maps $\tilde{\mathbf{Y}}_r$

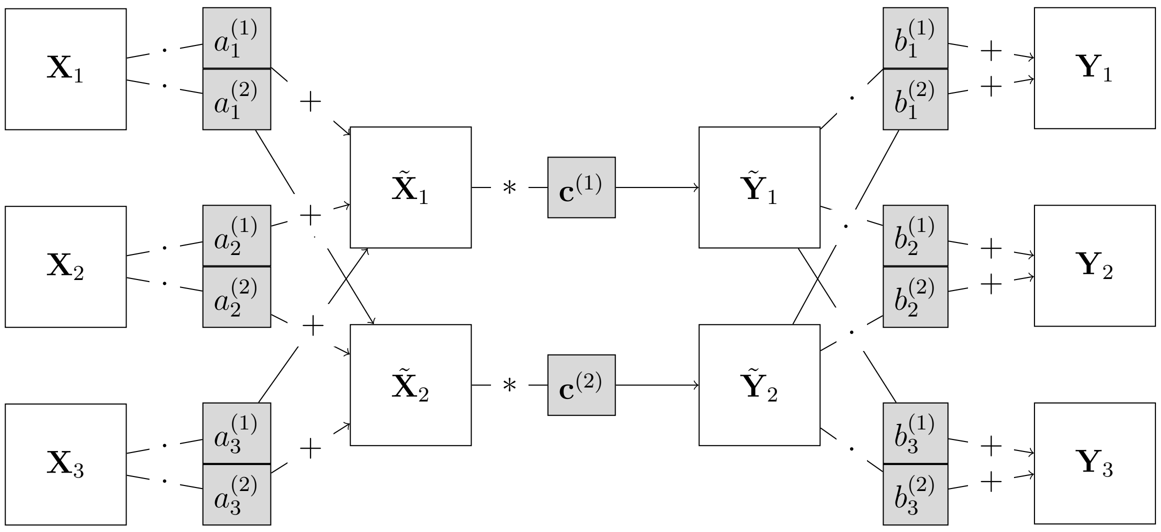


Figure 5.4: A CP convolution with 3 input maps, 3 output maps and $R = 2$. The input maps are compressed with the input factors $a_i^{(r)}$ into intermediate maps $\tilde{\mathbf{Y}}_r$, then convolved with the basis kernels $\mathbf{c}^{(r)}$ and finally decompressed with the output factors $b_o^{(r)}$ into the corresponding output maps.

3. and finally decompressing them with $b_o^{(r)}$ into the output map.

Figure 5.4 visualizes this approach.

Assuming that the height and width of the input and output stay the same, the number of parameters and FLOPs for the reduced layer are calculated as follows:

$$\begin{aligned} params(R) &= \underbrace{R \cdot I_m}_{a} + \underbrace{R \cdot K^D}_{b} + \underbrace{R \cdot O_m}_{c} \\ &= R \cdot (I_m + O_m + K^D) \\ flops(R) &= \underbrace{R \cdot I_m \cdot I_h \cdot I_w}_{\text{Step 1}} + \underbrace{R \cdot O_h \cdot O_w \cdot K^D}_{\text{Step 2}} + \underbrace{R \cdot O_m \cdot O_h \cdot O_w}_{\text{Step 3}} \\ &= R \cdot O_h \cdot O_w (I_m + O_m + K^D) \end{aligned}$$

5.4 Rank selection

We have now seen three reduction methods Affine (SVD), Convolution (SVD) and Convolution (CP). The resulting parameter count and FLOPs are summarized in Table 5.2. The only adjustable variable for the reduction methods is the rank R of the low rank approximation. In order to get an reduction of either the parameters or the FLOPs, the rank R has to be chosen so that the number of parameters in the reduced layer is smaller than the number of parameters in the original layer.

Type	**# Parameters**	**FLOPs**
Affine	$I^D \cdot O^D$	$I^D \cdot O^D$
Affine (SVD)	$R(O^D + I^D)$	$R(O^D + I^D)$
Convolution	$I_m \cdot O_m \cdot K^D$	$I_m \cdot O_m \cdot O_h \cdot O_w \cdot K^D$
Convolution (SVD)	$R \cdot I_m(O_m + K^D)$	$R \cdot I_m(O_m + K^D) \cdot O^D$
Convolution (CP)	$R(I_m + O_m + K^D)$	$R \cdot O_h \cdot O_w(I_m + O_m + K^D)$

Table 5.2: Formulas to calculate the number of parameters and FLOPs for original and reduced layers.

A convolution layer with $I_m = 3$, $O_m = 4$ and $K_h = K_w = 5$ needs $3 \cdot 4 \cdot 5 \cdot 5 = 300$ parameters. The SVD convolution for this layer requires $R \cdot 3 \cdot (4 + 25) = R \cdot 87$ parameters. In order to get an reduction of the parameters by a factor of 2, the rank of the SVD convolution has to be chosen such that:

$$R \cdot 87 = rate \cdot 300$$
$$R = \frac{1}{2} \cdot \frac{300}{87} \approx 1.72$$

Because the rank is an integer, it has to be rounded down to 1, resulting in an actual reduction of $1 - \frac{87}{300} = 0.71 = 71\%$ instead of the desired 50%.

The CP convolution requires $R \cdot (3 + 4 + 25) = R \cdot 32$ parameters and for the same reduction the rank has to be chosen so that:

$$R \cdot 32 = rate \cdot 300$$
$$R = \frac{1}{2} \cdot \frac{300}{32} \approx 4.69$$

The rank is rounded down to 4, giving us an actual reduction rate of $1 - \frac{4 \cdot 32}{300} \approx 0.57 = 57\%$.

6 Implementation

When working with neural networks there are several open-source frameworks like Caffe or Torch, which assist the user during development and evaluation. Nowadays it is critical that a given framework supports GPU assisted computations, as they can massively speed up the training process. Together with colleagues from Japan, SSG members are developing an internal Deep Learning toolkit called sDeePy, which is used in this thesis. The sDeePy toolkit allows the definition of neural networks as directed graphs where the nodes are the layers of the network. It contains implementations of the convolution and affine layers needed for CDNNs and provides a gradient descent method for training networks. sDeePy is written in Python and is heavily using the NumPy and Theano packages. NumPy is a powerful package which adds support for working with multidimensional data and implements many operations used in linear algebra such as matrix multiplications. The Theano package builds on this and extends NumPy by allowing the definition of mathematical expressions which are then optimized for the execution on a CPU or GPU.

Even though sDeePy is a proprietary framework, I decided to use it because

- employees at SSG are familiar with it and can provide first-class support,
- it has the ability to export models into C code (e.g. for Android),
- it already implemented the SVD affine and CP/SVD convolution reduction techniques in Python,
- and it is written in Python (a familiar and popular language).

However sDeePy was not completely usable right out of the box. The biggest missing feature was an implementation of the reduced layers for the C-Exporter, which was a requirement to execute the model under Android. However, the Python side of things provided all required functionality from the beginning and besides some small bug fixes here and there, the only improvement added was a faster variant of the CP convolution that will be explained later.

6.1 Layers

This section covers how the different layers are implemented in the Python part of sDeePy. Fortunately, the used libraries, like Theano, provide many useful building blocks making the actual implementations in sDeePy often just a couple of lines of code. But at least in the case of the reduced convolution layers, an optimized solution is not available and a simple solution is challenging within the limits of the high-level Python interface of the Theano library. The implementation of the activation function in sDeePy is separate from the actual layer, that is why it does not appear in the following sections.

6.1.1 Affine, SVD Affine and Convolution

An affine layer with one-dimensional input is simply implemented by the dot product of the weights with the input. For multi-dimensional input this is not possible anymore. Thankfully, Theano provides a `tensordot` function which covers this case.

Listing 6.1 Simplified Python implementation of the affine layer.

```python
def affine(input, params):
   output = tensordot(input, params.weights)

   return output + params.bias
```

The SVD affine layer does not require a special implementation as it can simply be implemented by replacing the previous affine layer with two affine layers that contain the reduced weights from (5.1). In this case the first affine layer does not have an activation function nor a bias.

When available, the convolution implementation uses cuDNN, a library provided by NVIDIA, which contains a very fast and GPU optimized CUDA implementation of the convolution layer[NVI16]. Otherwise it falls back to the `nnet.conv2d` function from Theano, which implements the same operation in a platform independent way.

Listing 6.2 Simplified Python implementation of the convolution layer.

```python
def conv(input, params):
   # Dimensions input: I_m x I_h x I_w
   #        weights: I_m x O_m x K_h x K_w
   if CUDNN_INSTALLED:
      output = cudnn.dnn_conv(input, params.weights)
   else:
      output = nnet.conv.conv2d(input, params.weights)

   return output + params.bias
```

6.1.2 SVD Convolution

The SVD convolution is implemented as explained at the end of Section 5.3.1 and as shown in Figure 5.2. Each input map is convolved with R basis kernels $\mathbf{b}_{ri}$ to produce the intermediate output maps. The intermediate output maps are then decompressed into the respective output maps by using the output factors $\tilde{u}_{oi}^{(r)}$.

Listing 6.3 Simplified Python implementation of the SVD convolution layer.

```python
def conv_svd(input, params):
   # Convolve each input with R basis kernels.
   # Dimensions input: I_m x I_h x I_w
   #   basis_kernels: I_m x rank x K_h x K_w
   output = theano.map(fn=conv, sequences=(input, params.basis_kernels))

   output = tensordot(output, params.output_factors)

   return output + params.bias
```

The `map` function of Theano loops over the first dimension of the elements in the `sequences` tuple and calls the function `fn` for each pair. It basically executes:

```python
for m in range(InMaps):
   output[m] = conv(input[m], params.basis_kernels[m])
```

And herein lies a limitation of Theano because the operations could be executed in parallel, thus making the SVD convolution layer slower than the optimized standard convolution layer although it requires less operations.

6.1.3 CP Convolution

The implementation of the CP convolution is similar to the implementation of the SVD convolution and explained at the end of Section 5.3.2 and shown in Figure 5.4. The input is compressed with the input factors $\mathbf{a}$ into R intermediate input maps, which are convolved with the basis kernels $\mathbf{c}$ producing the intermediate output maps, which are decompressed with the output factors $\mathbf{b}$ into the output maps.

Listing 6.4 Simplified Python implementation of the CP convolution layer.

```
def conv_cp(input, params):
   intermediate = tensordot(input, params.input_factors)

   # Convolve each intermediate input with one basis kernel.
   # Dimensions intermediate: rank x I_h x I_w
   #        basis_kernels: rank x 1 x K_h x K_w
   output = theano.map(fn=conv, sequences=(intermediate, params.basis_kernels))

   output = tensordot(output, params.output_factors)

   return output + params.bias
```

The same limitation of the `map` operation, that was mentioned in the previous section, also applies to the CP convolution. Unfortunately, there is no optimized implementation for CP convolution available and the logic has to be implemented in sDeePy itself, making the straight forward implementation very slow when compared to the implementation of standard convolution which is heavily parallelized and optimized.

A workaround was implemented that uses the standard convolution instead of the `map` function. For R input maps and R output maps, the standard convolution expects $R \times R$ kernels, one kernel for each input/output combination. By using only the kernels where the index of the input and output map are the same, the behavior of the CP convolution can be simulated. All other kernels have to be set to 0. See Figure 6.1, where the kernels W_{ij} with $j \neq j$ have been grayed out to indicate that they are set to 0. Instead of only R sequential convolutions, $R \times R$ parallel convolutions are calculated. This obviously increases the amount of required calculations but has the potential to result in a speedup, when the rank R is not too big. If this is not the case, then this workaround will have a negative effect. Basic testing showed, that a reduction by 50% is enough to ensure that the rank R is low enough to result in a speedup.

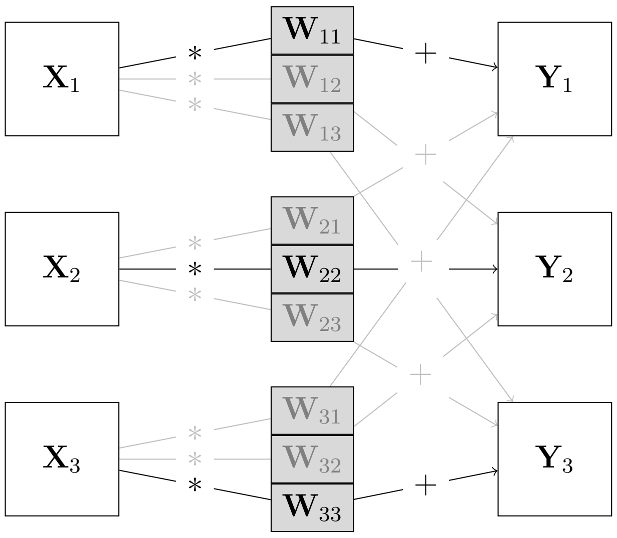


Figure 6.1: The standard convolution layer convolves each input/output map combination with a separate kernel. By using only specific kernels, the behavior of the CP convolution can be simulated.

Listing 6.5 Simplified Python implementation of the optimized CP convolution layer.

```
def conv_cp2(input, params):
   intermediate = tensordot(input, params.input_factors)

   # Create a rank x rank x kernel_height x kernel_width tensor
   # that contains the basis kernels on the diagonal axis
   kernels = numpy.zeros((params.rank, params.rank, params.kernel_height,
       params.kernel_height))
   rng = numpy.arange(params.rank)
   kernels[rng, rng] = params.basis_kernels

   # Dimensions intermediate: rank x I_h x I_w
   #             kernels: rank x rank x K_h x K_w
   conv(intermediate, kernels)

   output = tensordot(output, params.output_factors)

   return output + params.bias
```

6.2 C-Exporter

With the help of the sDeePy C-Exporter it is possible to create a pure C implementation of a network implemented in sDeePy. This is necessary for the Android App as there is no way to use sDeePy itself on a mobile phone. The exporter saves the layer parameters into a binary file and generates a C file that declares the network structure. When compiled, this C file is linked against a static C library, provided by sDeePy, that implements the algorithms for the layers and support code to load the parameters from a file. A compiled library of a model exports three main functions:

`mallocNet(char *path)` loads the parameters from the file pointed to by `path` and allocates all memory required by the layers.

`forward(float *input, float *output)` calculates the result of one forward pass.

`freeNet()` frees the memory allocated by the model.

The C-Exporter however was incomplete and had to be extended to be usable in this thesis. The two biggest missing features were no support for border mode of type *same* for the convolution layer (see Section 4.5) and no implementation for the CP convolution layer. To add support for the *same* convolution border mode, I implemented the 1D convolution that can be seen, in Algorithm 6.1.

Algorithmus 6.1 1D convolution

```
function CONV1D(In, InSize, Kernel, KernelSize, OutSize)
    Out[$0 \dots$ OutSize] $\leftarrow 0$
    Padding $\leftarrow \lfloor$KernelSize$/2\rfloor$
    for o $\leftarrow 0 \dots$ OutSize do
        i $\leftarrow$ o - Padding
        for k $\leftarrow 0 \dots$ KernelSize do
            if i+k $\geq 0$ or i+k $<$ InSize then
                Out[o] $\leftarrow$ Out[o] + Kernel[KernelSize - k - 1] $\cdot$ In[i + k]
            end if
        end for
    end for
    return Out
end function
```

Inside the inner loop it is checked, if the index `i+k` for the input is inside valid bounds and the calculations are performed. If that is not the case, then nothing is done, thus simulating zero-padded input.

A 2D convolution can then be implemented by calling the 1D convolution for the rows of input and kernel, as seen in Algorithm A.2.

Algorithmus 6.2 2D convolution

```
function CONV2D(In, InHeight, InWidth, Kern, KernHeight, KernWidth, OutHeight,
        OutWidth)
    Out[$0 \dots$ OutHeight][$0 \dots$ OutWidth] $\leftarrow 0$
    Padding $\leftarrow \lfloor$KernHeight/2$\rfloor$
    for o $\leftarrow 0 \dots$ OutHeight do
        i $\leftarrow$ o - Padding
        for k $\leftarrow 0 \dots$ KernHeight do
            if i+k $\geq 0$ or i+k $<$ InHeight then
                Out[o] $\leftarrow$ Out[o] + CONV1D(In[i], InWidth, Kern[KernSize - k - 1],
                    KernWidth, OutWidth)
            end if
        end for
    end for
    return Out
end function
```

When implementing and testing this code in C, it was noticed that the if-statement inside the 1D convolution slowed down the algorithm significantly. The statement checks unnecessarily often, if the index is inside the bounds and also prevents the compiler from applying optimizations to the loop. The actual C implementation of the 1D convolution, which can be seen in Listing A.1, optimizes this by splitting the algorithm into three parts. The first and third part, which handle the left and right zero-padded cases, still check the index, but the middle part can omit the check. Otherwise the algorithm is straight forward and does not use special SIMD[1] or NEON[2] instructions, that could be used to process multiple inputs at once. These instructions would make the generated code faster but also harder to read and understand. The usage might also cause problems with portability to other platforms that do not support these instruction sets. In the optimal case a compiler would detect that it can vectorize the calculations and generate optimized code.

[1]Single Instruction, Multiple Data
[2]ARM Advanced SIMD extension

7 Go Network Reduction and Evaluation

In Chapter 5 techniques for the CDNN reduction were introduced and explained. This chapter shows how these techniques can be applied to an already existing Go move predicting model and how the performance is affected.

7.1 The Original Network

The original neural network was developed and trained by C. Clark and A. Storkey as part of their research for “Training Deep Convolutional Neural Networks to Play Go” [CS14] which was first published in late 2014. They claim that their most recent 8-layer CNN beats previous networks by significant margins and that it wins 9 out of 10 games against GnuGo on the highest difficulty. They score the network at around 4-5 kyu, which is considered an intermediate amateur. This is evident, as it loses most games against Fuego, a advanced Go artificial intelligence, which is currently rated just above 3d on the KGS server [KGS16].

#	Type	Input	Output	Kernel	# Parameters	# FLOPs
1	Convolution	8x25x25	64x19x19	7x7	25 088	9 056 768
2	Convolution	64x19x19	64x19x19	5x5	102 400	36 966 400
3	Convolution	64x19x19	64x19x19	5x5	102 400	36 966 400
4	Convolution	64x19x19	48x19x19	5x5	76 800	27 724 800
5	Convolution	48x19x19	48x19x19	5x5	57 600	20 793 600
6	Convolution	48x19x19	32x19x19	5x5	38 400	13 862 400
7	Convolution	32x19x19	32x19x19	5x5	25 600	9 241 600
8	Affine	32x19x19	361x 1x 1		4 170 272	4 170 272
Total (Convolution)					428 288	154 611 968
Total (Affine)					4 170 272	4 170 272

Table 7.1: Layers and sizes of the original model with the totals shown in the last row for each type of layer.

A trained network was published as part of a web page where one can play against the said network [CS15]. This network will be used as our baseline and the structure is shown in Table 7.1. The last two lines of the table show the total number of parameters and FLOPs for each layer type summed up. One can immediately see that most parameters are contained in the single affine layer (≈ 91%) but most calculations are done inside the convolution layers (≈ 97%). This means that in order to reduce the number of calculations we need to reduce the convolution layers and to reduce the memory usage, we need to reduce the affine layer.

Assuming all $428288 + 4170272 = 4598560$ parameters are stored in memory as a 4 Byte floating point number, the model consumes around $\frac{4Byte \cdot 4598560}{1024 \cdot 1024} \approx 17.5MiB$ of memory and no modern smartphone should have problems loading the model into memory. Therefore, the focus is to reduce the number of FLOPs to speed up the calculation and to reduce the power consumption.

The input of the Go network is separated into 8 25×25 feature maps (see first layer in Table 7.1). Elements in the maps are binary and either set to 0 or 1. A 19×19 area in the center of each map represents the positions of the Go board while the area surrounding the board is always set to 0. The singe exception is the last feature map, where the padded area is set to 1 but everything else is 0. Maps 1, 2 and 3 contain the stones with 1, 2 or more liberties of the current player. Maps 4, 5 and 6 contain the same for the opponent. Map 7 marks the single ko position if it exits. The result of the network is a 361×1 vector containing the probabilities for each position to be a move an expert would do. To make interpretation of the result easier, the output can be reshaped into a 19×19 matrix.

7.1.1 Databases

Training of the original network is done using recorded Go games dating back to the year 196. Only moves from players above a specific rank should be considered, to ensure that the network learns only from good moves. The Go Games on Disk[1] (GoGoD) and KGS[2] databases provide Go games where most players are 5 dan or better and only very few players are beginners. Training, validation and testing of the original network was done by using all 81 000 games from the GoGoD database and 86 000 games (out of 171 000) from the KGS database. With around 16 Million moves per database this results in a total of 32 Million individual Go moves used for the training, validation and testing. Unfortunately it is unknown which exact games and moves are part of the training, validation and test sets. But the use of any games, that were used for training, should not be used for testing because it is not possible to properly evaluate, if the model is generalized enough. Using

[1]http://gogodonline.co.uk
[2]http://www.gokgs.com/

the same data for training and testing may result in unusual good results because the model could be exactly fitted to the training data. The same model might have very bad results when tested on data that was not used for the training. A separate set of testing data is therefore required.

C. Clark and A. Storkey first published their results on the 10th December 2014, so it is assumed that no games from 2015 or newer can be part of their data set. As the GoGoD and KGS databases used are now more recent, I have access to over 1 400 new games that contain over 300 000 moves from the GoGoD database and over 4 500 new games and 900 000 moves from the KGS database. To make sure no data, that was used to train the original model, is used for testing, only the new games are used. The remaining games from the GoGoD database (everything including December 2014) are used for training. The KGS database contains 85 000 games older than December 2014 but not used to train the original model. Training on these additional games might give the finetuned models an advantage over the original model, because it ultimately has seen more games. To avoid this problem the KGS database is not used for training, resulting in a total of only 81 000 games with 16 Million moves from the GoGoD database available for training.

7.1.2 Reflection Preserving Kernels

It is important to note that the original model was trained using *reflection preserving* kernels, which improved the accuracy. These *reflection preserving* kernels can be mirrored horizontally, vertically and diagonally and the result will be the same kernel, as seen in Figure 7.1. The weights with the same color are tied together and always contain the same value. If one weight is updated during training, so are the others. The idea behind this method is, that a Go board can be mirrored the same way without changing its meaning and the authors of the original network noticed that even without tying the weights together, some some of the learned kernels were already reflection preserving. They claim that the same could be achieved by applying the reflections on the training data, but this would increase the data by factor of 8, thus also increasing the training time by the same factor.

The use of reflection preserving kernels means that the kernel weights are highly redundant. This should allow a good compression of the convolution layers without losing accuracy. In theory, we only need to save 10 out of 49 weights for a 7×7 kernel and for a 5×5 kernel, 6 out of 25 weights are necessary. This results in a theoretical loss-less reduction of 76.56% of all convolution layers in the Go model.

The reflection preserving kernels were not implemented during this thesis and the available training data is also used as-is. As the previous section explained, only half the training data, used to train the original model, is available in the first place and with the assumption

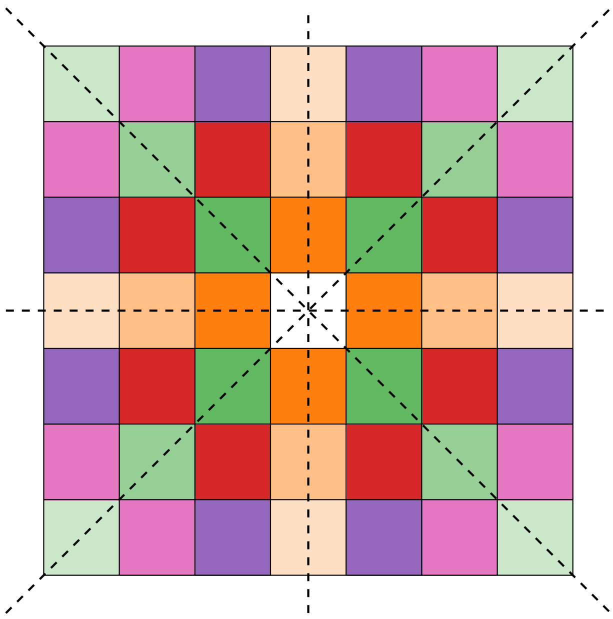


Figure 7.1: A 7×7 kernel where the tied weights have the same color.

that reflection preserving kernels artificially increase the data by a factor of 8, this means that effectively only 1/16th of the data can be used for training.

7.2 Metrics

Overview and explanation of the metrics used to assess the quality of Go networks.

FLOPs: The number of floating point operations required to calculate the result of a single layer or the total network. The formulas can be taken from Table 5.2.

Reduction Rate: The reduction is based on the number of parameters or the number of FLOPs. It is possible to use the definitions interchangeably, as a reduction of the parameters by half also results in half the FLOPs. When talking about convolution layers, it makes sense to relate the reduction rate to the number of FLOPs and when talking about affine layers, to relate it to the number of parameters. The rate will be reported as either a number between 0 and 1 or as a percentage between 0 and 100%.

Accuracy: The most important metric is the accuracy of the predictions done by a model as it does not really matter how fast or small a model is when its output is worthless. It is measured by comparing the position with the highest probability, with the move done by a human player from the test database. The number of correctly predicted moves is divided by the number of total moves to get the total average accuracy.

Accuracy	**Win Rate**
40.12%	87%
37.37%	85%
33.36%	75%
26.14%	50%

Table 7.2: Win rates of different DCNNs against GnuGo on the highest difficulty.

If multiple predictions have the same exact probability, which mostly happens for the first move, then only the move which happens to be first in a sorted list will be considered.

Size: The number of parameters for a single layer or the complete network. The formulas can be taken from Table 5.2. This number is used to roughly estimate the memory consumption needed.

It is not entirely obvious that a good accuracy will result in a good Go playing AI. At the same time it is also not obvious that a bad accuracy will result in a particular bad AI. Just because the move with the highest probability does not match the exact move taken by a human player, does not mean that it is a bad move. To make sure that move prediction accuracy is a good metric, different models were matched against GnuGo on the highest difficulty. The results can be seen in Table 7.2. With decreasing accuracies the win rate against GnuGo drops, thus supporting using accuracy to measure how good a network is performing. Based on the win rates against GnuGo, I make the arbitrary decision to consider accuracies above 39% as good, between 39% and 37% as acceptable and below 37% as bad.

7.3 Methods

As there are multiple layers in our model, the reduction techniques can be combined in various ways. We will first look at the affine and convolution layers separately and this section will explain how the techniques are applied to the layers and how the reduction rates are chosen. Later on we will look at the global network reduction by considering all layers.

7.3.1 Uniform reduction

A simple way to reduce a model is to apply the same reduction rate to all layers. E.g. to get a total reduction rate of 50%, every layer is reduced by 50%. This does not take the size of the layer into account.

7.3.2 Non-uniform Reduction based on Layer Sizes

Contrary to the uniform reduction, we can try to be more intelligent in choosing the the individual reduction rates. The idea is to avoid bottlenecking the smaller layers when reducing them too much. My colleagues at SSG proposed the following formula to calculate the individual reduction rates, where L is the number of layers, P_l is the number of parameters for layer l, P is the number of total parameters, p is the non-uniformity parameter and τ the desired overall compression rate.

$$\tau_l = \tau_0 \left(\frac{P_l}{P}\right)^p$$

with

$$\tau_0 = \tau \frac{P^{p+1}}{\sum_{l=1}^{L} P_l^{p+1}}$$

Using p we can adjust the flatness of the reduction curve, with $p = 0$ giving us the uniform reduction. In Figure 7.2 four reduction curves for $\tau = 0.5$ and different values for p are shown together with the number of parameters of the convolution layers.

7.3.3 Non-uniform Reduction based on Approximation Error

This approach assumes that some layers in a network can be compressed more than other layers, independent of the layer size. The individual layers are reduced with increasing rates until the approximation error surpasses a specific difference threshold. This threshold is calculated by comparing the approximated weights to the original weights and depends on the specific norm used. As both the SVD and CP based reduction try to minimize the Frobenius norm $\left\|\tilde{\mathbf{W}} - \mathbf{W}\right\|_F^2$, where $\tilde{\mathbf{W}}$ are the approximated weights and $\mathbf{W}$ are the original weights, the same norm is used to calculate the approximation error.

7.3.4 Finetuning

Higher compression rates are expected to have a severe negative effect on the quality of the reduced networks. If training data is available, it is possible to counteract this by

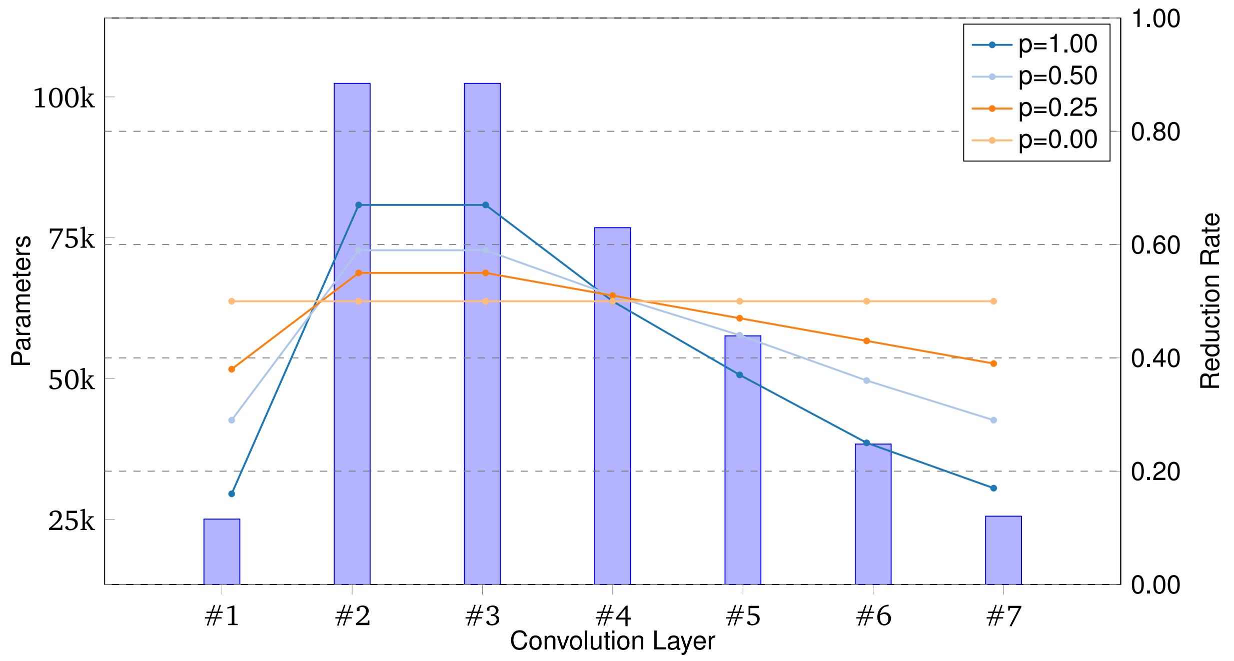


Figure 7.2: The effect of p on the reduction rates when the rates are chosen based on the layer sizes. The bars show the size of the layers as the number of parameters.

retraining the reduced network, or parts thereof. The distinction between normal training and finetuning is that during finetuning, the network is only trained for one or two epochs, on a subset of the original training data. In comparison, the original network was trained for 9 epochs on 34 Million samples and the training process took around four days.

The available data usable for finetuning is limited anyways, as I explained in Section 7.1.1. This means that at most 17 Million samples are used when finetuning. Due to the unoptimized implementations of the CP convolution (see Section 6.1.3) the training process is also too slow to handle more data. For low reduction rates the training can take up to 4 days for one epoch and 17 Million moves on a GeForce GTX TITAN X graphics card.

A learning rate of 0.00001 is used when training convolution layers and set to 0.001 for affine layers. These values have shown to provide good results during finetuning. The batch size is always set to 128. The finetuning process is restricted to only the reduced layers, leaving the other layers untouched. This mostly means that either all convolution layers are trained or the single affine layer.

7.4 Evaluation

In this section, I will show the results of the evaluation of the different reduction techniques and methods from the previous section. The previous sections made some assumptions about possible reduction rates and performance of the techniques. In summary, the following can be expected:

1. CP based reduction results in higher accuracy than the SVD based reduction when using the same rate.
2. The convolution layers can be reduced by about 75% without losing accuracy.
3. Choosing a non-uniform distribution of the reduction rates results in a higher accuracy for the same reduction rate.
4. Finetuning the reduced models improves the accuracy.
5. With increasing reduction rates, the reduced models compute results faster and require less memory. A 50% parameter reduction should result in at least a 2 $\times$ speedup and require no more than half the memory.
6. Reduced models with fewer FLOPs consume less power on a mobile device.

7.4.1 SVD vs. CP

In Section 5.3 the two reduction methods SVD and CP were introduced. The assumption was made that the CP decomposition will provide better reduction results because it can operate on the complete weight tensor instead of 2D slices. For the evaluation I uniformly reduced all convolution layers with different rates using the SVD reduction. The SVD reduction only works on two-dimensional slices of the original weights, where the size of one dimension is relatively small (25 - 49). This means that it often cannot exactly match the desired reduction rate because it has to cut off the matrices at a higher rank. However, the CP reduction can match a desired rate very closely. Therefore, to get a fair comparison, the resulting reduction rates of the SVD reduction were used for the CP reduction.

The result of the evaluation is shown in Table 7.3 where the resulting SVD reduction rates for a 50%, 60%, 65%, 70%, 75% and 80% reduction are shown with the number of parameters, FLOPs and the accuracy of both reduction methods. The difference in the number of parameter for the two methods shows, that the CP reduction can relatively accurately match the SVD reduction rate.

From Figure 7.3 we can immediately see that both reduction techniques can reduce the convolution layers by 62% within an acceptable loss of accuracy. With the CP reduction it

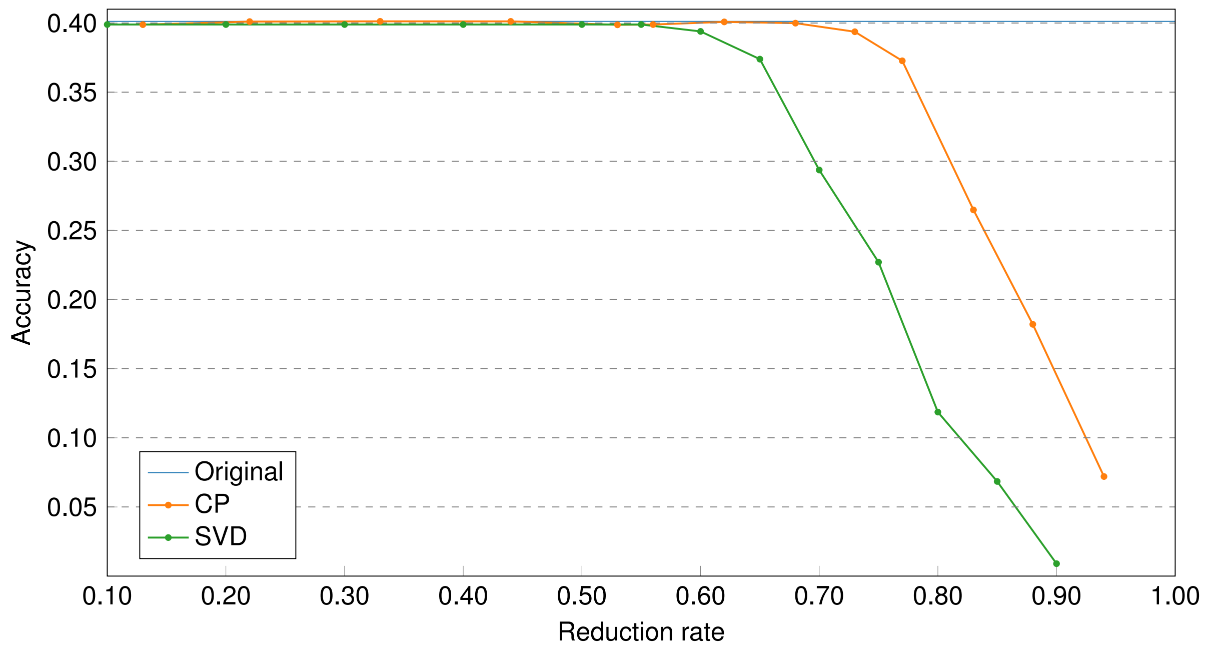


Figure 7.3: Accuracy comparison between SVD and CP based reduction for different reduction rates of the convolution layers

	SVD			CP		
Rate	# Params	FLOPs	Accuracy	# Params	FLOPs	Accuracy
Original	428 288	154 611 968	40.12%	428 288	154 611 968	40.12%
56.44%	186 576	67 353 936	39.89%	187 980	67 860 780	39.89%
62.38%	161 544	58 317 384	39.39%	162 314	58 595 354	40.08%
68.34%	135 608	48 954 488	37.39%	136 858	49 405 738	39.99%
73.12%	115 136	41 564 096	29.37%	115 053	41 534 133	39.23%
77.26%	97 376	35 152 736	22.70%	97 924	35 350 564	37.26%
83.11%	72 344	26 116 184	11.86%	72 290	26 096 690	26.48%

Table 7.3: Comparison between SVD and CP for different reduction rates. The colors indicate good, acceptable and bad accuracy results.

is possible to increase the reduction rate to 73% without a major loss in accuracy. This confirms our first assumption that the CP reduction is better than the SVD reduction.

7.4.2 CP Reduction Stability

In the previous section it was shown that the CP reduction seems to be superior to the SVD reduction for all rates. Yet before we can argue about results of the CP reduction we

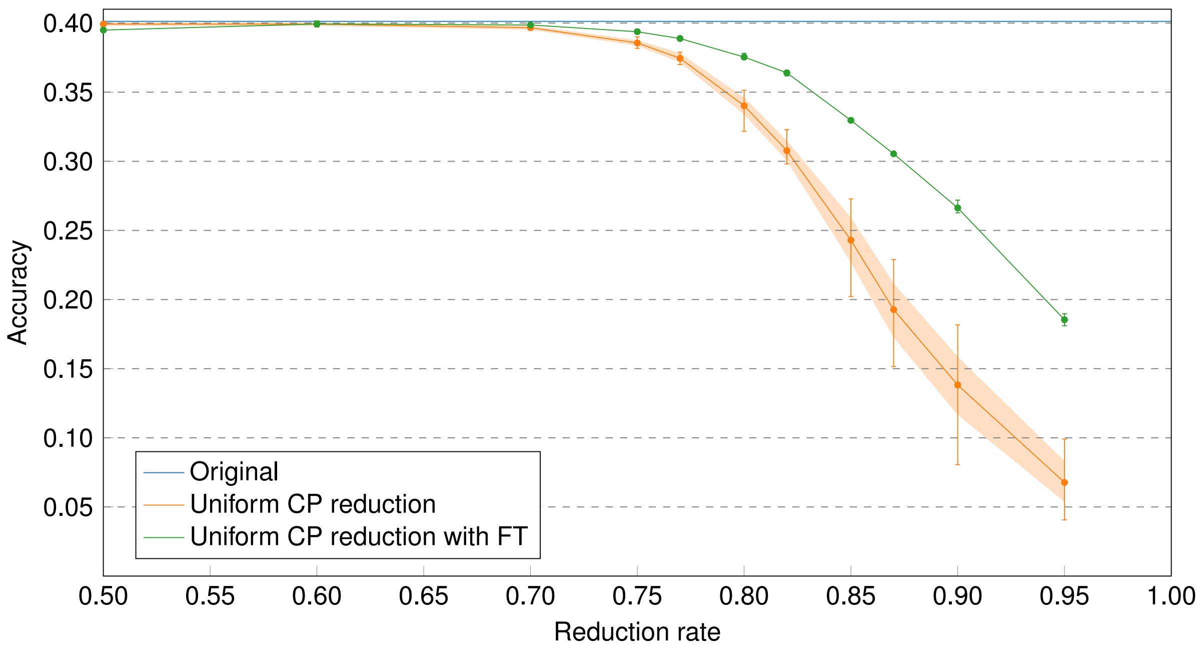


Figure 7.4: Stability of the CP uniform reduction showing the mean, minimum and maximum accuracy. The orange area is the standard deviation of the non-finetuned models.

need to keep in mind that there is no stable algorithm to calculate the best approximation for a given rank, as mentioned in Section 5.3.2. This means that multiple reductions using the same rank or reduction rate can result in models having a different accuracy.

To get an idea about the instability, 50 models for different reduction rates were reduced and evaluated. Additionally 10 models, including the best and worst, were finetuned with 1M moves to see if the accuracy of the approximation has an effect on the finetuned model. The result can be seen in Figure 7.4. While I initially expected the variations in accuracy to be small and negligible, I was proven to be wrong. CP reduced models with the same reduction rate can vary by up to 10 percentage points for certain reduction rates. By finetuning the models we can reduce the differences substantially and the instability of the CP reduction is alleviated. Based on these findings it is not meaningful to compare differently reduced models on higher rates, when no finetuning is involved, as potential differences inside the standard deviation can attributed solely to the instability of the approximation.

From Table 7.4 we can see that without finetuning the maximum accuracy is 39.00% for model reduced by 75%. This somewhat confirms the second assumption, that states that the convolution layers can be reduced by 75% without a loss in accuracy. Also by finetuning the models with only 1 Million samples, we can recover about half of the lost accuracy, which should improve even more when using more data.

Rate	Without Finetuning				With Finetuning			
	Mean	Max	Min	Error (σ)	Mean	Max	Min	Error (σ)
50%	39.92%	40.13%	39.84%	± 0.09%	39.48%	39.56%	39.37%	± 0.07%
60%	39.92%	40.14%	39.83%	± 0.10%	39.92%	40.16%	39.71%	± 0.15%
70%	39.66%	39.89%	39.48%	± 0.11%	39.88%	40.01%	39.76%	± 0.10%
75%	38.56%	39.00%	38.17%	± 0.17%	39.37%	39.55%	39.24%	± 0.11%
77%	37.49%	38.01%	37.00%	± 0.30%	38.91%	38.99%	38.82%	± 0.08%
80%	34.00%	35.14%	32.16%	± 0.57%	37.57%	37.81%	37.35%	± 0.16%
82%	30.77%	32.29%	29.81%	± 0.67%	36.40%	36.56%	36.18%	± 0.13%
85%	24.31%	27.28%	20.22%	± 1.56%	32.96%	33.12%	32.86%	± 0.09%
90%	13.76%	18.17%	8.06%	± 2.06%	26.59%	27.19%	26.27%	± 0.29%
95%	6.74%	9.91%	4.06%	± 1.43%	18.50%	18.98%	18.10%	± 0.30%

Table 7.4: Results of the analysis of over 500 CP reduced models and 100 finetuned models. The table shows the selected reduction rate together with the mean, minimum, maximum and standard deviation of the accuracies.

7.4.3 Non-uniform Reduction based on Layer Sizes

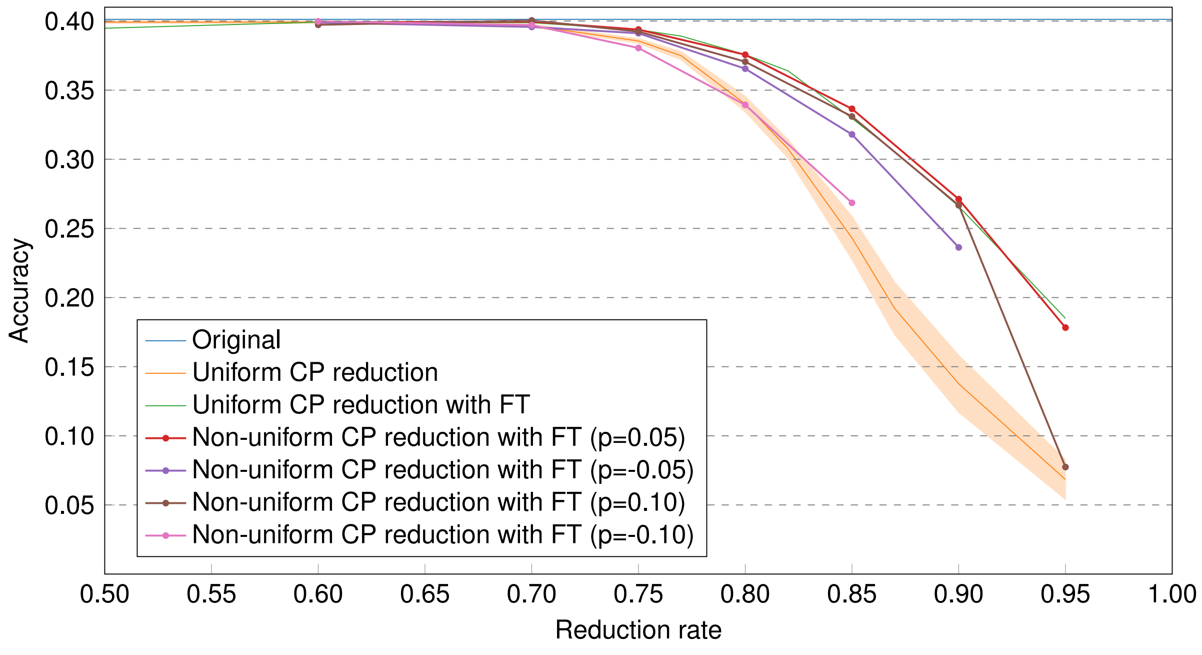
While good results can already be achieved with a uniform reduction of the layers, the reduction rate can hopefully be increased even more by reducing the layers non-uniformly based on their size. As the layers in our network vary greatly in size, this approach seems so make sense.

The method introduced in Section 7.3.2 requires that a value for the non-uniformity parameter p is chosen. Therefore different values for p also need to be evaluated. From Figure 7.2 it can be seen that sensible values for p are in the range from 0 to 0.5 as higher values will reduce the bigger layers by too much.

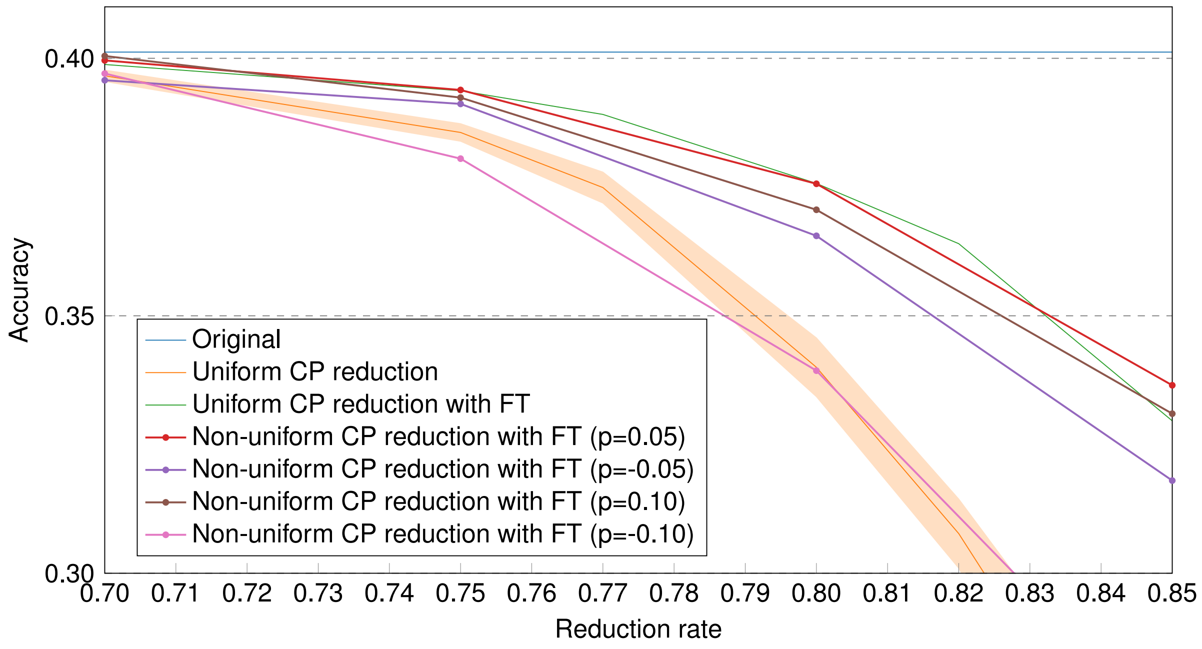
In Figure 7.5 four different non-uniform reductions are plotted against the uniform reduction. Unfortunately the result is not what I expected. The best accuracy was achieved using $p = 0.05$, which shows a small improvement for the 85% and 90% reductions. With $p = 0.05$, the individual reduction rates of the convolution layers come close to a uniform reduction. When increasing or even negating p the accuracy drops very quickly making this approach essentially useless for our model. This unfortunately does not confirm the third assumption made at the beginning of this section.

7.4.4 Non-uniform Reduction based on Approximation Error

Instead of defining a desired reduction rate, this non-uniform reduction requires a threshold parameter to find the highest reduction rate, that keeps the approximation error below



(a) Full graph comparing the finetuned non-uniform reduction against the uniform reduction.



(b) Magnification of the range from 0.7 to 0.8 from of the above figure

Figure 7.5: Accuracy comparison between uniform and non-uniform CP based reduction for different reduction rates for the convolution layers.

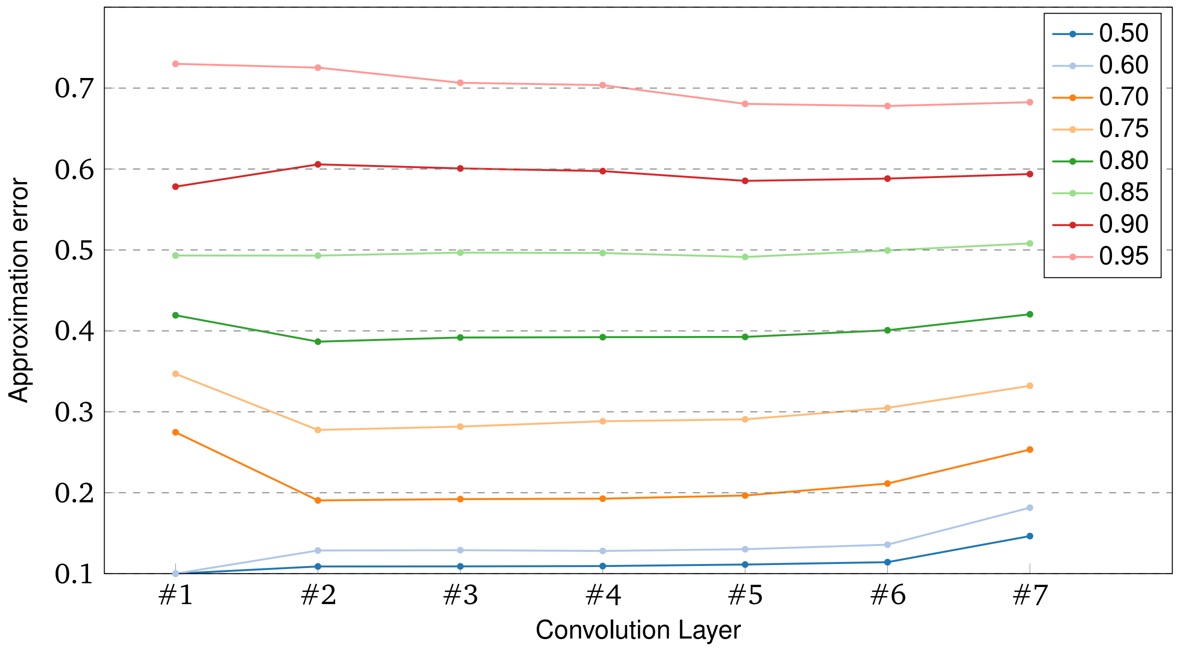


Figure 7.6: Mean values of the weight approximation error per layer for different reduction rates.

this threshold. Before such a threshold can be chosen, we have to know in which range the approximation error lies. I therefore analyzed the models created for the CP stability analysis and calculated the error for each layer separately. In Figure 7.6 the approximation error of all convolution layers using different reduction rates are shown. A bigger value corresponds to a worse approximation of the weights, which is quite clear as it increases for higher reduction rates.

We have seen that a reduction rate of 70% can result in a model with nearly no loss in accuracy and that by going beyond 80% reduction, the accuracy drops dramatically. Therefore it seems sensible to choose threshold values between 0.1 and 0.3. Looking at the three curves for the 70%, 75% and 80% reduction rates, it can be seen that the first and the last two layers seem to be the layers most sensitive to reduction as they have the biggest error. The second and third layer have the smallest error and therefore seem to be least sensitive to reduction. This seems to support the idea behind the non-uniform reduction from the previous section, that small layers may be the bottle neck when reducing. Interestingly, this does not seem to be the case for the 85% and 90% reductions where the previous method had the best results.

In Figure 7.7 the results for different thresholds are shown. The reduction rates for the individual models were chosen by a binary search algorithm which starts with a 50% reduction and selects 25% or 75% in the next iteration depending on the approximation error. With 8 iterations this results in a reduction rate step size of $\frac{1}{2^8} \approx 0.0039$. The result is

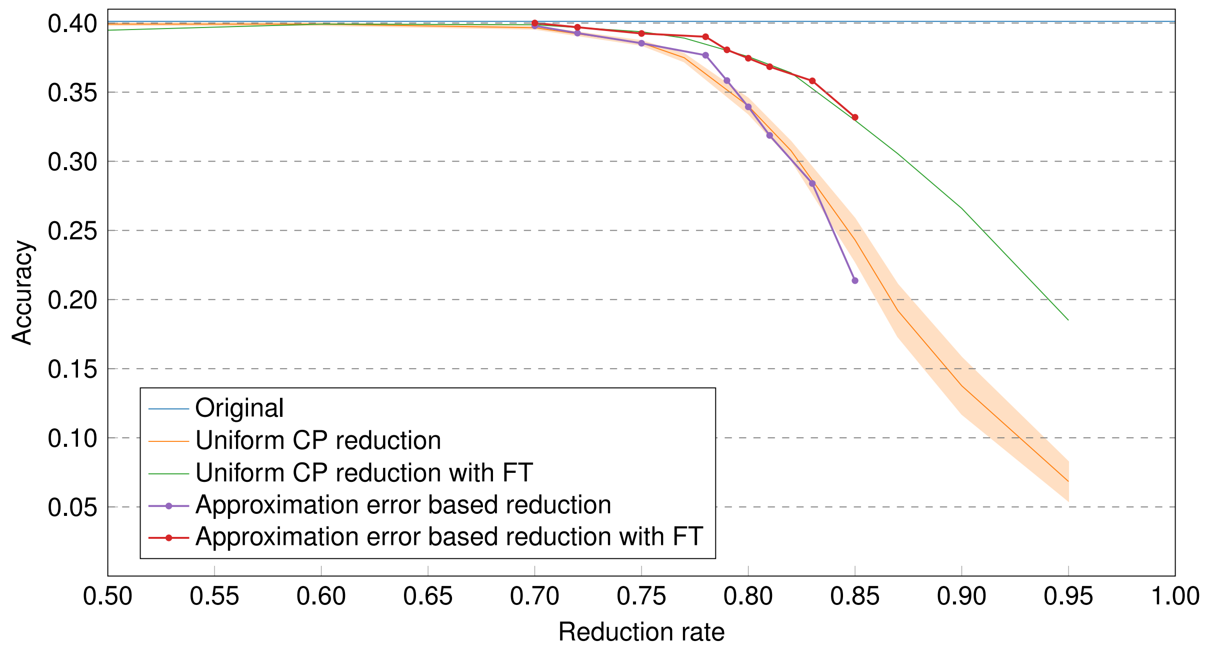


Figure 7.7: Accuracy of the weight difference based reduction for the thresholds 0.1, 0.15, 0.2, 0.25, 0.275, 0.3, 0.325, 0.35 and 0.4.

similar to the layer size based non-uniform reduction in that there is no big improvement over the uniform reduction.

The results in Figure 7.7 have to be interpreted with caution because of the instability of the CP approximation covered in Section 7.4.2. If an approximation happens to be particular bad, then the algorithm that chooses the reduction rates will select a lower reduction rate. I noticed that only after I discovered the big fluctuations in the accuracy of non-finetuned models.

Comparing the layer weights may however still be useful to find reduction rates that have a low approximation error and therefore have a good accuracy. This is confirmed by looking at the first data point of the non-finetuned curve that has an accuracy of 39.79% and corresponds to a model created using a 0.1 threshold. If no data for testing is available, then this might be a good way to conservatively reduce a model while still retaining the original accuracy.

7.4.5 Affine Reduction

So far we have concentrated on reducing the convolution layers and achieved good results there. The only other type of layer in our Go model is the last affine layer and we will have a look at it in this section. The SVD, which is used to create an approximation of the

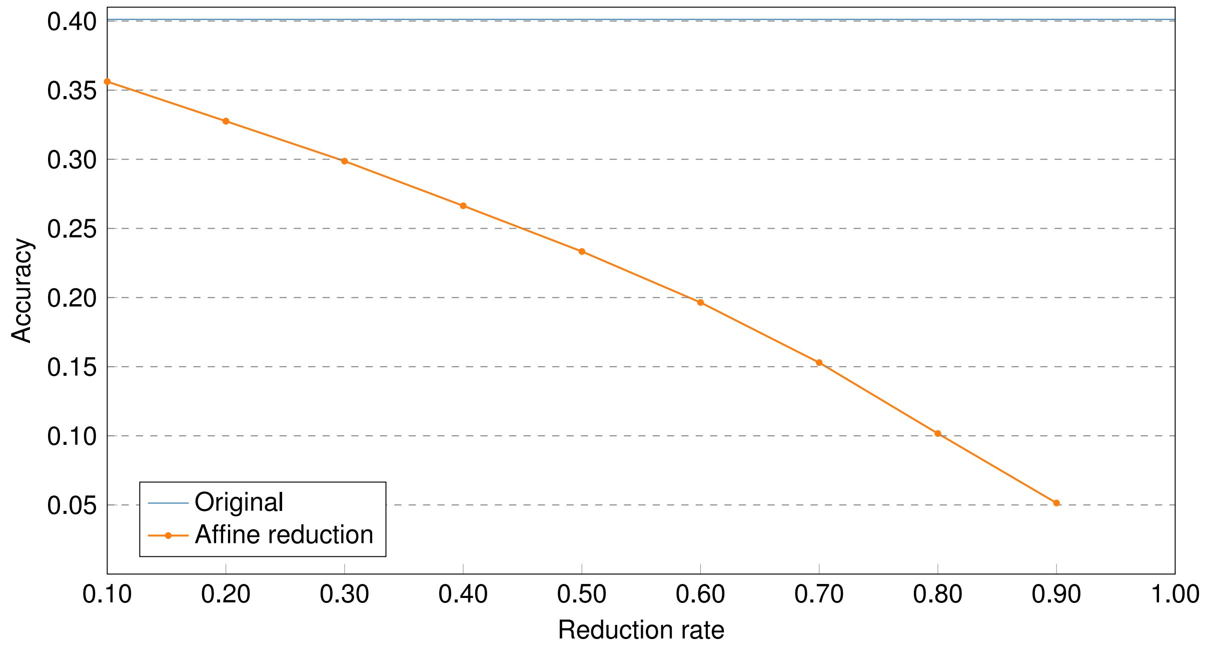


Figure 7.8: Accuracy of the affine layer reduction.

affine weights, has a stable algorithm. Therefore no stability analysis has to be performed as the approximation will always be the same.

Unfortunately reducing the affine layer even by low rates has a severe impact on the accuracy of the Go model, as is evident from Figure 7.8. A reduction by 10% already reduces the accuracy to 35.62% and it decreases near linear to 5.14% for a reduction rate of 90%. This likely indicates that the affine weights do not contain redundant information in a away that is exploitable by a SVD. In the next section we will see how we can improve the accuracy with finetuning.

7.4.6 Finetuning

Finetuning was already used to counteract the instability of the CP reduction when comparing reductions of the convolution layers. To speed up the finetuning process only 1 Million instead of all 17 Million moves were used. In this section we have a look at the improvements when using all available data and when training for more than one epoch.

The finetuning process does not try to improve the approximation of the layer weights. Instead, the approximation can be seen as a good initialization of the weights that will be refined by a classical neural network training. Instead of the usual random initialization of the weights, the approximated weights already give good results. It is therefore possible to improve a reduced model, even when using only a small amount of training data.

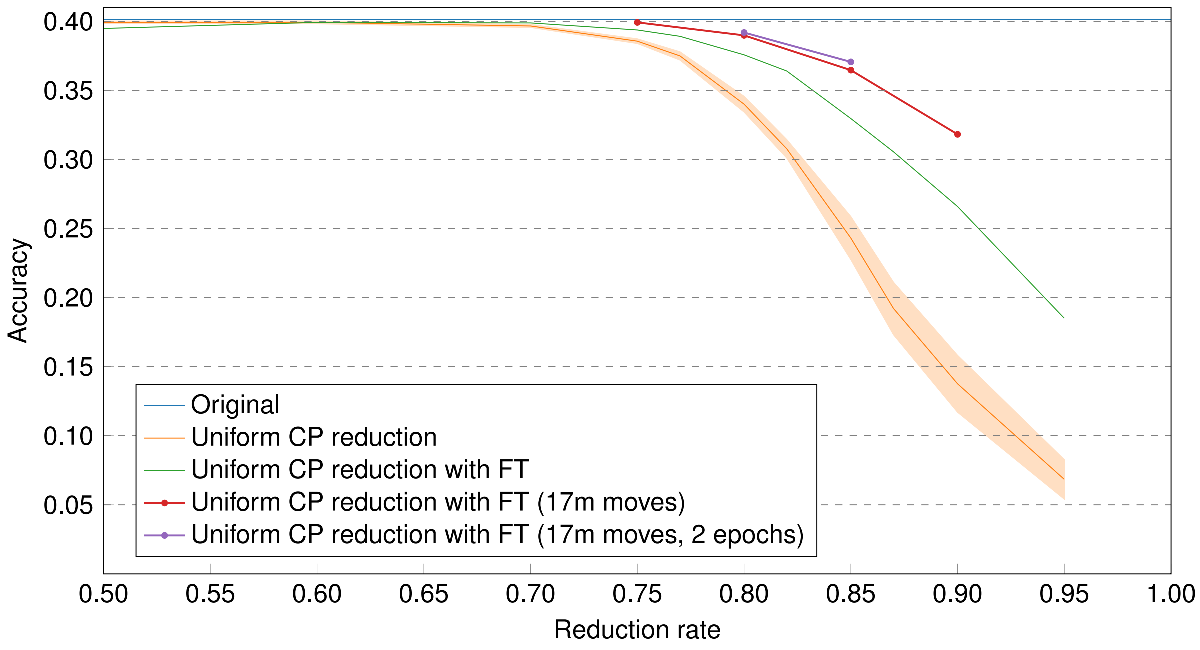


Figure 7.9: Accuracy of finetuned models using 1m and 17m moves.

Rate	1M moves	17M moves	17M moves, 2 epochs
75%	39.37%	39.92%	
80%	37.57%	38.98%	39.18%
85%	32.96%	36.46%	37.06%
90%	26.59%	31.82%	
95%	18.50%		

Table 7.5: Accuracies of finetuned convolution layer reductions.

As explained in Section 7.1, the original network was trained with reflection preserving kernels and 32 Million moves for 9 epochs. We are limited to at most 17 Million moves and all of that will be used for finetuning.

We have already seen in the CP stability analysis that finetuning with 1 Million moves can recover about half of the lost accuracy for convolution layers. By using all 17 Million moves, the accuracy is again improved by a significant amount, as can be seen in Figure 7.9 and Table 7.5. Another small improvement is achieved by finetuning the model for two epochs.

Looking at Figure 7.10 and Table 7.6 we can observe similar results when finetuning the single affine layer. With 1 Million moves, the accuracy improves by a big amount and another big jump is achieved when using all 17 Million moves. It is very likely that training for additional epochs will improve the accuracy even more.

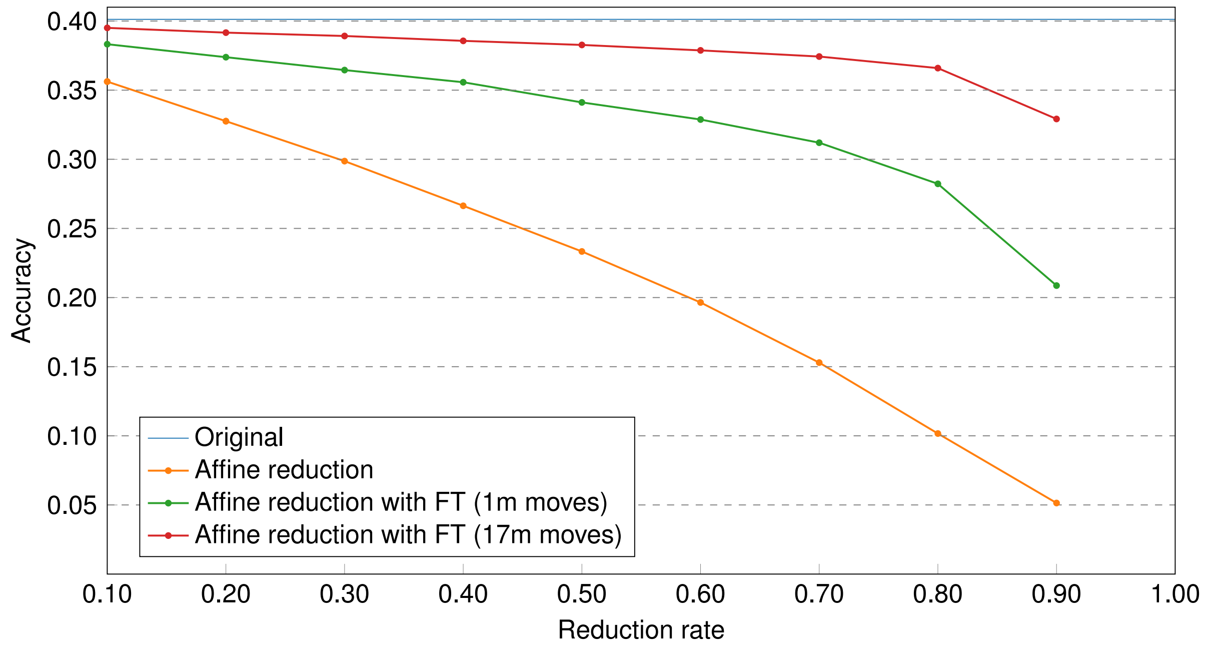


Figure 7.10: Accuracy of affine layer reductions with finetuning.

Rate	no FT	1M moves	17M moves
10%	35.62%	38.32%	39.50%
20%	32.76%	37.38%	39.16%
30%	29.87%	36.45%	38.92%
40%	26.64%	35.57%	38.57%
50%	23.33%	34.11%	38.27%
60%	19.64%	32.88%	37.88%
70%	15.29%	31.20%	37.43%
80%	10.16%	28.23%	36.59%
90%	5.14%	20.87%	32.92%

Table 7.6: Accuracies of affine layer reductions.

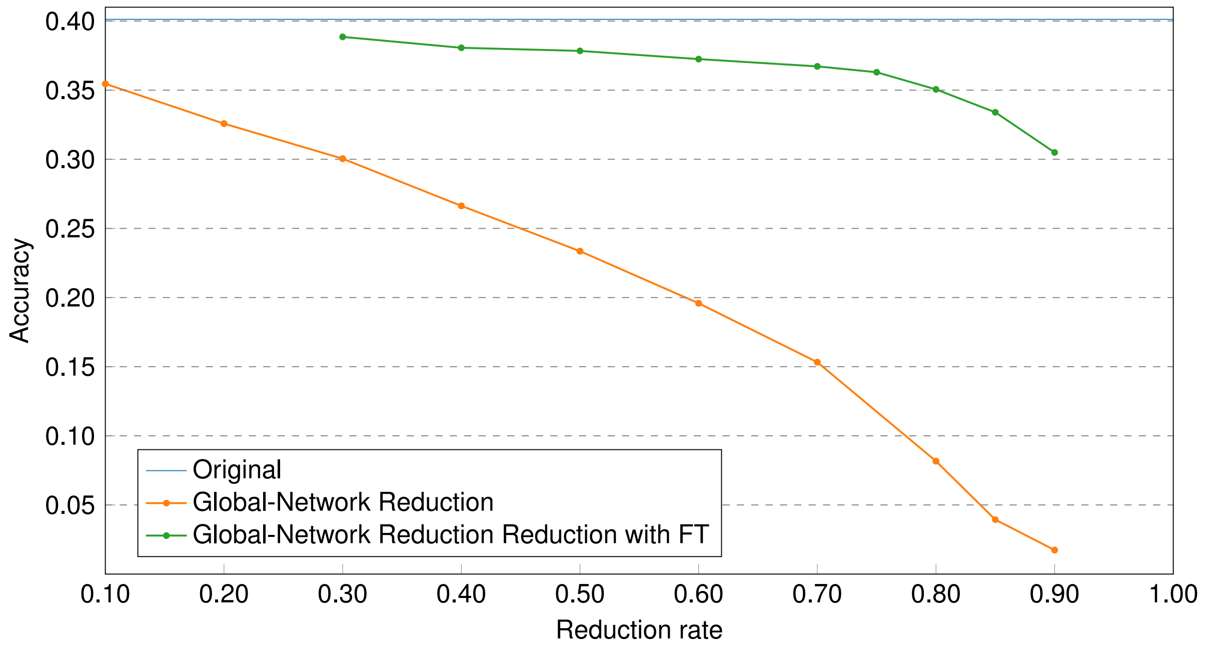


Figure 7.11: Accuracies of models were all layers have been reduced uniformly.

Rate	no FT	17M moves
10%	35.46%	–
20%	32.58%	–
30%	30.04%	38.85%
40%	26.63%	38.06%
50%	23.35%	37.83%
60%	19.58%	37.24%
70%	15.32%	36.72%
80%	8.17%	35.06%
85%	3.94%	33.40%
90%	1.74%	30.49%

Table 7.7: Accuracies of uniform global-network reductions with and without the use of finetuning.

7.4.7 Global-Network Reduction

So far we have either reduced the convolution layers or the affine layer. What is missing is applying reductions to both layer types at the same time. Due to the bad performance of the SVD affine reduction, resulting models are expected to be quite bad and unusable without finetuning.

Rate	Time	Speedup	Memory	Improvement
None	2714 ms	–	18981 KiB	–
10%	1807 ms	1.50 ×	17183 KiB	1.10 ×
20%	1598 ms	1.69 ×	15386 KiB	1.23 ×
30%	1397 ms	1.94 ×	13590 KiB	1.40 ×
40%	1204 ms	2.25 ×	11793 KiB	1.61 ×
50%	994 ms	2.73 ×	9997 KiB	1.89 ×
60%	794 ms	3.41 ×	8201 KiB	2.31 ×
70%	595 ms	4.56 ×	6404 KiB	2.96 ×
80%	396 ms	6.85 ×	4608 KiB	4.12 ×
90%	198 ms	13.71 ×	2811 KiB	6.75 ×

Table 7.8: The computation time and memory usage of models reduced with increasing rates together with their speedup/improvement over the original model.

Figure 7.11 shows the results of a uniform reduction of all layers and confirms the expected bad results. Upon closer inspection, the accuracies of the non-finetuned models are very similar to the accuracies of non-finetuned affine reduction up until a 70% reduction. This makes sense when we consider that the convolution layers can be reduced by up to 75% without a loss of accuracy. Even the finetuned results are similar to the finetuning results of the affine layer, indicating that the affine layer is still the limiting factor. Further improvements are likely achieved by training for more epochs and adjusting the learning rate. But the unoptimized implementations of the CP convolution layers prevented more experiments, especially for lower rates as finetuning the model reduced by 50% already took over three days.

7.4.8 Prediction Speed and Memory Usage

At the beginning of Section 7.4 it was mentioned that a reduction of the parameters should lead to a model, that can compute results faster and consumes less memory. To test if this assumption is true, a benchmark program was written, that measures the time it takes to execute 20 forward passes and outputs the maximum memory consumption. The benchmark program is written in C and uses the code from the sDeePy C-Exporter, because the implementation of the convolution and CP convolution layers is comparable. Both implementations use the same underlying convolution algorithm and are not parallelized. Memory usage was measured using the memory tracking mechanism, that is built into the sDeePy Exporter. The benchmark was compiled with GCC 4.8.3 using the `-Ofast` optimization level and executed on a Intel Xeon X5570 CPU that runs at 2.93GHz.

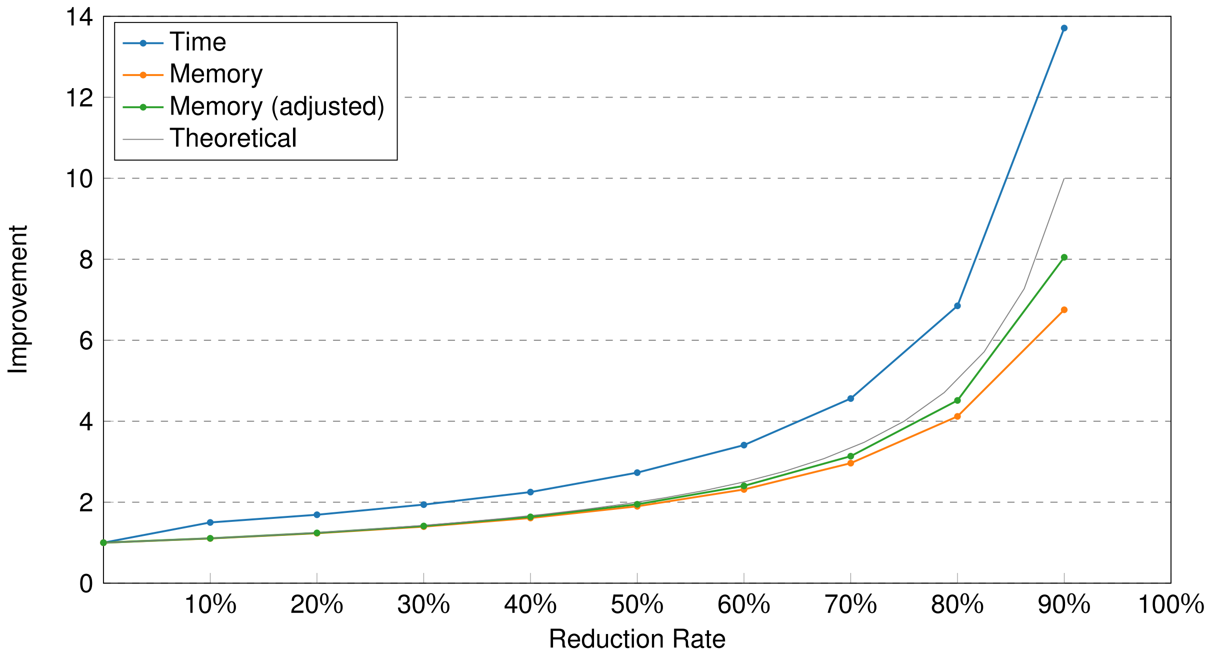


Figure 7.12: Improvements of the computation time and memory consumption for different reduction rates. The gray curve is the theoretical expected improvement.

For the benchmark, the models of the global network reduction were used. Although this combines the reduction of the convolution and affine layers, this should still allow us to independently look at the effects of the respective reductions because of the unequal distribution of parameters and FLOPs between the layer types (see Section 7.1). The timing and memory values from this benchmark are shown in Table 7.8 and reveal suprising results.

Already a model reduced by only 10% shows a significant 1.5 $\times$ speedup in computation time. And a 80% FLOPs reduction should theoretically result in a $\frac{1}{1-0.8} = 5\times$ speedup, yet it is a 6.85 $\times$ speedup. The proportionally high speedups can be easily seen in Figure 7.12, where the curve for the computation speedup is above the curve for the theoretical improvement for all rates. By reducing the models, the number of parameters is reduced and individual parameters may be reused more often for the calculation of the model output. This improves the CPU cache efficiency of the algorithms and thus explaining the speedups. The same effect was observed on the Xperia Z3 smartphone, where the original model took 558 ms and a 80% reduced model improved it by a factor of 9.78 to 57 ms for a single forward-pass.

Improvements in the memory usage are more in line with the theoretical improvement, but fall behind when surpassing the 50% reduction. After analyzing how the C-code allocates memory, it was noticed that the exported models always keep the input for all layers in memory. The size of the layer input is not affected by the used reduction methods and

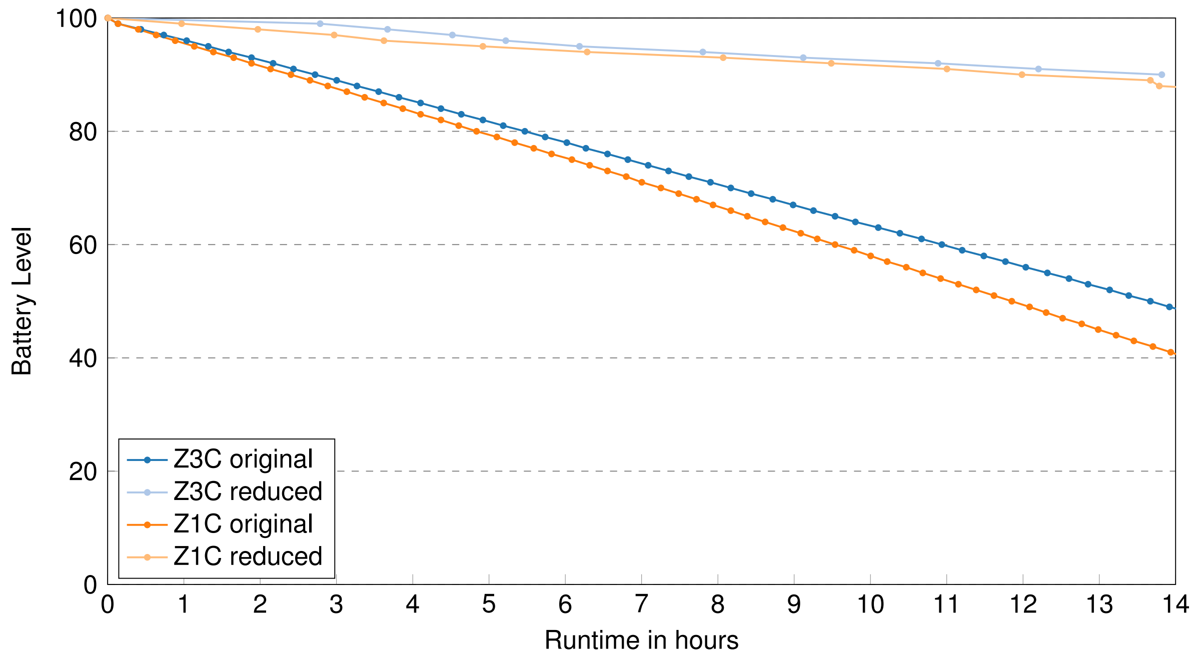


Figure 7.13: Battery consumption of the original and a 80% reduced model over the course of 14 hours.

therefore constant for all reduced models. After accounting for the 132072 float numbers (see Table 7.1) needed to keep all layer inputs in memory, the memory improvements come close to the theoretical expected ones.

7.4.9 Energy Consumption

Besides a faster computation and lower memory usage, the power consumption is an important factor on a mobile device. The energy consumption was measured using an Android service that executes 20 predictions every minute and records the battery level after each run. A `PARTIAL_WAKE_LOCK` is acquired when calculating the predictions to make sure the CPU of the smartphone can use the highest frequency and get it done as quickly as possible. The wakelock is released immediately afterwards to allow the phone to get back into a deep sleep state as soon as possible. To ensure the system wakes up in a timely manner, a `RTC_WAKEUP` is used.

The test itself was executed on a Xperia Z1 Compact and on a Z3 Compact with the original and a model where the FLOPs were reduced by 80%. To prevent other Apps and updates to interfere with the test, the phones were put into airplane mode, which disables all connectivity. The result of this can be seen in Figure 7.13. It is immediately obvious that the battery level dropped significantly faster when using the original model compared to the reduced one. Over the course of 14 hours the battery dropped to 41% on the Z1 and to 49% on the Z3 when using the original model. When using the reduced model, the battery dropped only to 88% and 90%, which is an 5$\times$ improvement on both phones.

When looking at the previous 6.85 $\times$ speedup in computation time, a bigger improvement in battery consumption can be expected. But relying on the battery information from the Android system is not a very accurate method because other hardware and software components still consume power, even though the phone is idle. Additionally, the CPU might not immediately go into a low power state after the computations are done and thus drawing more power than necessary. A more accurate method would be to measure the power consumption of the CPU directly with an external device like done in [KPY+15].

7.5 Summary

Of the introduced reduction methods, the CP reduction of the convolution layers shows the best results. These good results are explained by the redundancy of the kernels, due to their reflection preservation. The convolution layers can be reduced by up to 75% without finetuning and still produce models with a good accuracy. When finetuning the convolution layers, acceptable models can be produced with a reduction of 85%. While finetuning improves the already good results of the CP convolution reduction, it is absolutely necessary when reducing the single affine layer. Without finetuning a 10% reduction already drops the accuracy of the resulting model to an unacceptable 35.62%. By finetuning the affine layer, the layer can be reduced by up to 70% with an acceptable drop in accuracy.

8 Summary and Outlook

In this bachelor thesis, a Go game demo in form of an Android App was developed. The App is able to detect a marked Go board and the position of the stones in images taken directly from within the App by the smartphone or tablet's camera. Knowing the stone positions, the App overlays the predictions for the next best moves on the image. The App was used to show the benefits of the Convolutional Deep Neural Network reduction at an internal Sony exhibition in Tokyo. It allows the selection of different reduced models to compare the move predictions and computation speed. The App managed to catch the attention of various managers and executives, including Sony's Chief Financial Officer Kenichiro Yoshida. The booth at the exhibition was well attended and visitors had to form a queue to have a look at the demonstrations, which included a video playing back a Go game to highlight the speedup of the a reduced Go model. The app was also shown multiple times to in-house visitors in Stuttgart and based on the received feedback, can be seen as a success.

When concentrating on the convolution layers, it is possible to reduce the model by 70% with only dropping the accuracy by 0.46 percentage points to 39.66%. With the use of finetuning, the reduction rate can be increased to 80% with a drop of 0.94 percentage points. Unfortunately, the single affine layer could not be reduced by anywhere near the same amount. Without finetuning, the affine layer reduction did not produce models with acceptable accuracy. When finetuning the single affine layer, a large drop in accuracy could be recovered and a 70% reduction, with an acceptable accuracy loss of 2.69 percentage points, is possible. When reducing both layer types at the same time, the Go model can be reduced by 60% within an acceptable drop of accuracy to 37.24%.

The reduced models show their benefits when measuring the computation time, memory consumption or power usage. Due to improved utilization of the CPU caches, a small reduction of 10% already speeds up the computation by 1.5 $\times$ on a Xeon X5570 CPU. Reducing the model by 80% gives us a 6.85 $\times$ speedup on the same CPU and a 9.78 $\times$ speedup on a Xperia Z3 smartphone. Compared to the original model, the 80% reduced model also reduces the power consumption of the smartphone by the same amount.

8.1 Future Work

Based on the results of this thesis there are many ideas that can be realized in future work. A new App could be created that uses a neural network for the artificial intelligence. With a strength of 4-5 kyu, this App would provide a stronger computer controlled opponent compared to many existing Android Apps that are based on GnuGo. For power users, this app may also improve the battery life because the move predictions can be calculated quicker and use less battery. The augmented reality aspect of the current App could also be improved to speed up the board detection. A possible board and stone detection in real time might be possible when utilizing the CPU and GPU together. Alternatively, the board could be detected without the need for special markers based on the grid on the board and shape of the stones. When considering the network reduction aspect of this thesis, then an implementation of a parallel and fast CP convolution is something I was missing and could potentially realized in various ways.

A Appendix

A.1 C-Exporter Source Code

Listing A.1 C implementation of the 1D convolution.

```c
inline void convf(float *in, float *kernel, float * out, int in_size,
                  int kernel_size, int out_size, int padding) {
  assert( in_size - kernel_size + 1 + 2 * padding == out_size );

  int k, o, idx;
  const float *subin;
  float sum;
  // Slide kernel over input

  // Left part with zero padding
  for (o = 0, subin = &in[o - padding]; o < padding; ++o, ++subin) {
    sum = 0;
    for (k = 0, idx = o - padding; k < kernel_size; ++k, ++idx) {
      // Ignore values outside of the input to simulate zero-padding
      if (idx >= 0) {
         sum += kernel[kernel_size - k - 1] * subin[k];
      }
    }
    out[o] += sum;
  }

  // Middle part of the input where no zero padding is needed
  for (o = padding, subin = &in[0]; o < out_size - padding; ++o, ++subin) {
    sum = 0;
    for (k = 0; k < kernel_size; ++k) {
       sum += kernel[kernel_size - k - 1] * subin[k];
    }
    out[o] += sum;
  }

  // Right part with zero padding
  for (o = out_size - padding, subin = &in[o - padding]; o < out_size; ++o, ++subin) {
    sum = 0;
    for (k = 0, idx = o - padding; k < kernel_size; ++k, ++idx) {
      // Ignore values outside of the input to simulate zero-padding
      if (idx < in_size) {
         sum += kernel[kernel_size - k - 1] * subin[k];
      }
    }
    out[o] += sum;
  }
}
```

Listing A.2 C implementation of the 2D convolution.

```
/*
 * 2D float convolution with optional zero padding around the input.
 */
inline void convf2d(float *in, float *kernel, float *out, int in_height, int in_width,
               int kernel_height, int kernel_width, int out_height, int out_width,
               int padding_height, int padding_width) {
  assert( in_height - kernel_height + 1 + 2 * padding_height == out_height );
  assert( in_width - kernel_width + 1 + 2 * padding_width == out_width );

  int o, k, idx;

  for (o = 0; o < out_height; ++o) {
    float *out_row = &out[out_width * o];
    idx = o - padding_height;
    for (k = 0; k < kernel_height; ++k) {
      // Ignore values outside of the input to simulate zero-padding
      if (idx >= 0 && idx < in_height) {
        // Flip kernel
        float *kernel_row = &kernel[kernel_width * (kernel_height - k - 1)];
        float *in_row = &in[in_width * idx];
        convf(in_row, kernel_row, out_row, in_width, kernel_width, out_width, padding_width);
      }
      ++idx;
    }
  }
}
```

Listing A.3 C implementation of the 2D CP convolution layer (part 1).

```c
void cpconvolution(float *input, int in_maps, int in_height, int in_width,
              float *output, int out_maps, int out_height, int out_width,
              int k_height, int k_width,
              CPConvolutionConfig *config) {
  // Shape: outmaps x rank
  float *o_factors = config->o_factors;
  // Shape: inmaps x rank
  float *i_factors = config->i_factors;
  // Shape: kernelelements x rank
  float *k_factors = config->k_factors;
  // Shape: outmaps
  float *b_factors = config->b_factors;

  int rank = config->rank;

  int pad_height = config->padding_height;
  int pad_width = config->padding_width;

  int m, r, i, j;

  // k_factor shape is k_height x k_width x rank. But we need to pass the whole
  // kernel into the convolution function. So move the rank dimension to the
  // beginning.
  float (*kernels)[k_height][k_width][rank] = (float (*)[k_height][k_width][rank])k_factors;
  float sk[rank][k_height][k_width];
  for (i = 0; i < k_height; ++i) {
    for (j= 0; j < k_width; ++j) {
      for (r = 0; r < rank; ++r) {
        sk[r][i][j] = (*kernels)[i][j][r];
      }
    }
  }

  // Initialize output to bias values
  for (m = 0; m < out_maps; ++m) {
    float *map = &output[m * out_height * out_width];
    float bias = b_factors[m];
    for (i = 0; i < out_height * out_width; ++i) {
      map[i] = bias;
    }
  }
```

Listing A.4 C implementation of the 2D CP convolution layer (part 2).

```
  float tmp_map[in_height * in_width];
  float conv_map[out_height * out_width];

  for (r = 0; r < rank; r++) {
    memset(&tmp_map, 0, sizeof(tmp_map));
    memset(&conv_map, 0, sizeof(conv_map));

    // Multiply each input map with the corresponding factor and sum them to
    // create an intermediate input map.
    for (m = 0; m < in_maps; m++) {
      float *map = &input[m * in_height * in_width];
      float w = i_factors[m * rank + r];

      for (i = 0; i < in_height * in_width; ++i) {
        tmp_map[i] += w * map[i];
      }
    }

    // Convolve intermediate map

    // Kernel for current rank
    float *kernel = &sk[r][0][0];

    convf2d(&tmp_map[0], kernel, &conv_map[0], in_height, in_width, k_height,
          k_width, out_height, out_width, pad_height, pad_width);

    // Apply the convolved output to each output map
    for (m = 0; m < out_maps; ++m) {
      float *map = &output[m * out_height * out_width];
      float w = o_factors[m * rank + r];

      for (i = 0; i < out_height * out_width; ++i) {
        map[i] += w * conv_map[i];
      }
    }
  }
}
```

A.2 3D Printed Marker Bases

Because the markers are placed outside the actual board, one needs to make sure that the marker surface is on a plane with the board surface. Without a supporting structure, markers printed on paper or cardboard drop a little at their corners and thus make it harder for the board detection to find a good perspective projection. The developed solution uses 3D printed marker bases, that have been designed with the height of the Go

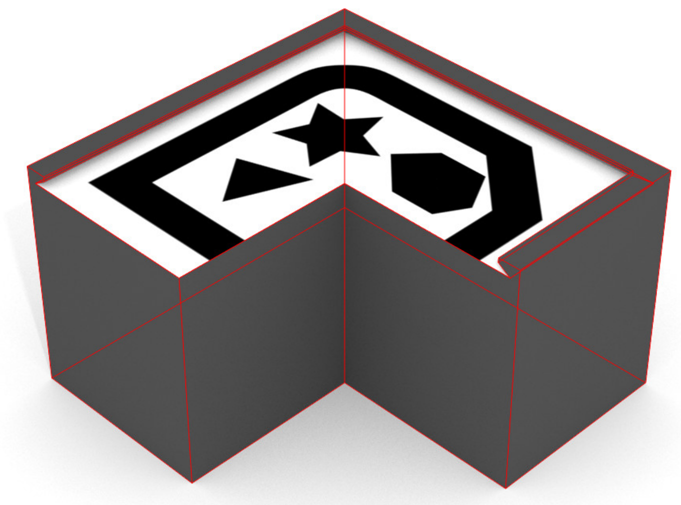


Figure A.1: A render of the marker base with a paper sheet of the marker image stuck on top.

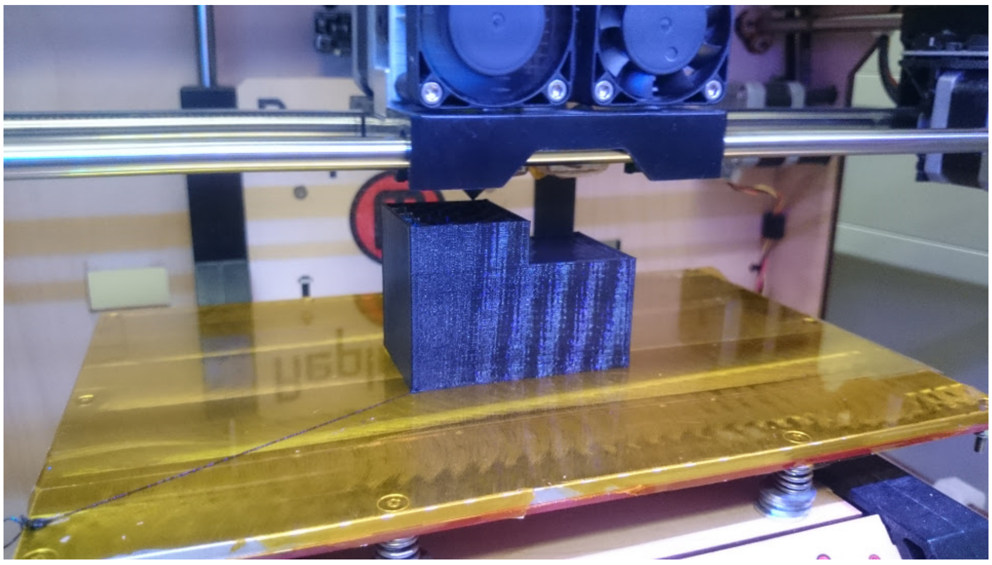


Figure A.2: The marker base in the process of being printed by a MakerBot. Printing one marker base took 3 hours.

board in mind and allow printed marker images to be slided in and out. A render of the maker can be seen in Figure A.1. An overhang is used to fix the sheet of paper. Figure A.2 shows the process of printing the marker using a MakerBot 3D printer. The marker base has been rotated to the side to avoid having problems when printing the overhang.

Bibliography

[All94] L. V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. 1994 (cit. on p. 15).

[Alp16] AlphaGo. *AlphaGo | Google DeepMind*. https://deepmind.com/alpha-go.html. 2016 (cit. on p. 11).

[Bis06] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006 (cit. on pp. 31, 34, 36).

[CS14] C. Clark and A. J. Storkey. "Teaching Deep Convolutional Neural Networks to Play Go." In: *CoRR* abs/1412.3409 (2014). URL: http://arxiv.org/abs/1412.3409 (cit. on pp. 11, 15, 57).

[CS15] C. Clark and A. J. Storkey. *Play Go Against a Deep Neural Network*. https://chrisc36.github.io/deep-go/. 2015 (cit. on p. 58).

[dSL08] V. de Silva and L.-H. Lim. "Tensor Rank and the Ill-Posedness of the Best Low-Rank Approximation Problem." In: *SIAM J. Matrix Anal. Appl.* 30.3 (Sept. 2008), pp. 1084–1127. URL: http://dx.doi.org/10.1137/06066518X (cit. on p. 45).

[Fia04] M. Fiala. *ARTag revision 1, a fiducial marker system using digital techniques*. Tech. rep. NRC 47419/ERB-1117. 2004 (cit. on p. 23).

[Hir08] M. Hirzer. *Marker Detection for Augmented Reality Applications*. http://lrs.icg.tugraz.at/pubs/hirzer_tr_2008.pdf. 2008 (cit. on p. 23).

[JVZ14] M. Jaderberg, A. Vedaldi, and A. Zisserman. "Speeding up Convolutional Neural Networks with Low Rank Expansions." In: *CoRR* abs/1405.3866 (2014). URL: http://arxiv.org/abs/1405.3866 (cit. on pp. 41, 42).

[KB09] T. G. Kolda and B. W. Bader. "Tensor Decompositions and Applications." In: *SIAM Rev.* 51.3 (Aug. 2009), pp. 455–500. URL: http://dx.doi.org/10.1137/07070111X (cit. on p. 44).

[KGS16] KGS-Go-Server. *KGS-Go-Server-Ranggraph*. http://www.gokgs.com/graphPage.jsp?user=fuego19. 2016 (cit. on p. 57).

[KPY+15] Y. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. "Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications." In: *CoRR* abs/1511.06530 (2015). URL: http://arxiv.org/abs/1511.06530 (cit. on pp. 42, 78).

[LGR+14] V. Lebedev, Y. Ganin, M. Rakhuba, I. V. Oseledets, and V. S. Lempitsky. "Speeding-up Convolutional Neural Networks Using Fine-tuned CP-Decomposition." In: *CoRR* abs/1412.6553 (2014). URL: http://arxiv.org/abs/1412.6553 (cit. on pp. 42, 45).

[Lib16] S. Library. *Current state of computer go*. http://senseis.xmp.net/?ComputerGo. 2016 (cit. on pp. 11, 16).

[MHSS14] C. J. Maddison, A. Huang, I. Sutskever, and D. Silver. "Move Evaluation in Go Using Deep Convolutional Neural Networks." In: *CoRR* abs/1412.6564 (2014). URL: http://arxiv.org/abs/1412.6564 (cit. on p. 15).

[NVI16] NVIDIA. *NVIDIA cuDNN*. https://developer.nvidia.com/cudnn. 2016 (cit. on p. 50).

[Ope16a] OpenCV. *OpenCV Color Blob Detection*. https://github.com/Itseez/opencv/tree/master/samples/android/color-blob-detection/src/org/opencv/samples/colorblobdetect/. 2016 (cit. on p. 20).

[Ope16b] OpenCV. *OpenCV Find Homography*. http://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html?highlight=findhomography. 2016 (cit. on p. 25).

[Ope16c] OpenCV. *OpenCV HoughCircles*. http://docs.opencv.org/2.4/modules/imgproc/doc/feature_detection.html?highlight=houghcircles. 2016 (cit. on p. 26).

[Ope16d] OpenCV. *OpenCV Perspective Transformation*. http://docs.opencv.org/2.4/modules/imgproc/doc/geometric_transformations.html. 2016 (cit. on p. 22).

[PSPS11] D. Prochazka, M. Stencl, O. Popelka, and J. Stastny. "Mobile Augmented Reality Applications." In: *CoRR* abs/1106.5571 (2011). URL: http://arxiv.org/abs/1106.5571 (cit. on p. 23).

[SHM+16] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. "Mastering the game of Go with deep neural networks and tree search." In: *Nature* 529 (2016), pp. 484–503. URL: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html (cit. on p. 16).

[Ste93] G. W. Stewart. "On the Early History of the Singular Value Decomposition." In: *SIAM Rev.* 35.4 (Dec. 1993), pp. 551–566. URL: http://dx.doi.org/10.1137/1035134 (cit. on pp. 40, 41).

[SZ14] K. Simonyan and A. Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." In: *CoRR* abs/1409.1556 (2014). URL: http://arxiv.org/abs/1409.1556 (cit. on p. 31).

[SZG+09] F. Schweiger, B. Zeisl, P. Georgel, G. Schroth, E. Steinbach, and N. Navab. "Maximum Detector Response Markers for SIFT and SURF." In: *Vision, Modeling and Visualization Workshop (VMV)*. Braunschweig, Nov. 2009 (cit. on p. 24).

[TF07] J. Tromp and G. Farnebäck. "Combinatorics of Go." In: *Computers and Games*. Ed. by H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers. Vol. 4630. Lecture Notes in Computer Science. Springer, Oct. 1, 2007, pp. 84–99. URL: http://dblp.uni-trier.de/db/conf/cg/cg2006.html%5C#TrompF06 (cit. on p. 11).

[Wik16a] Wikipedia. *Complexities of some well-known games*. https : / / en . wikipedia . org / wiki / Game_ complexity # Complexities_ of_ some_ well\discretionary{-}{}{}known_games. 2016 (cit. on p. 11).

[Wik16b] Wikipedia. *HSL and HSV*. https://en.wikipedia.org/wiki/HSL_and_HSV. 2016 (cit. on p. 21).

All links were last followed on March 20, 2015.

Declaration