jamk.fi

Go game move prediction using convolutional neural network

Marek Korenciak

Bachelor's thesis
May 2018
School of Technology, Communication and Transport
Information and Communications Technology

Software Engineering

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author(s)	Type of publication	Date
Marek Korenciak	Bachelor's thesis	May 2018
	Number of pages 49	Language of publication: English
		Permission for web publication: yes

Title of publication

Go game move prediction using convolutional neural network

Degree programme

Information Technology, Software Engineering

Supervisor(s)

Salmikangas, Esa

Assigned by

Abstract

The purpose of this paper is to introduce the use of convolutional neural network for prediction of the next appropriate move in the Go game. The paper contains description of the crucial Go game rules, neural networks theory, description of implemented programs and final evaluation of the trained neural networks.

The programs were implemented with programming language C++ using Caffe framework for the initialization and management of predesigned convolutional neural network models.

The thesis compares various models and ways of neural networks learning. The outcome of the experiments was poor; nevertheless, the analysis revealed the main root for this, which was insufficient hardware power. Changes were proposed, probably leading to successful neural network, predicting appropriate moves in the Go game.

Despite the imperfections, the experiments proved convolutional neural networks are applicable for the next step prediction in the Go game if the training process is performed properly.

Keywords

deep machine learning, neural networks, convolutional neural networks, Go game, Caffe framework, next move prediction, artificial intelligence

Miscellaneous

Content

Figures

Tables

Acronyms

AI	Artificial Intelligence
BAIR	Berkeley AI Research
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
cuDNN	cuda Deep Neural Network
GPU	Graphics Processing Unit
IoT	Internet of Things
MCTS	Monte Carlo Tree Search
OS	Operating System
RGB	Red Green Blue
SGD	Stochastic Gradient Descent
UML	Unified Modeling Language
XML	eXtensible Markup Language

1 Introduction

The informatics evolves - alike other fields of science. From time to time, ideas arise that can be marked “revolutionary”. They can even change the whole world - not only their field in the academic world. Definitely, as Examples of such revolutionary ideas are e.g. industrial automatization or, the first graphical operating system or smartphones. Actual upcoming candidate members on the list are Internet of things (IoT) and Artificial intelligence (AI).

Artificial intelligence (AI) is changing our lives even now despite the fact that the field gained momentum only recently. Companies such as Google, Facebook, and Amazon demonstrate that by utilizing AI for analysis and data processing. AI is preferably applied to very complex problems inappropriate for classical deterministic programming with simple rules and relations. Go game can be considered one of them.

Go Game is actually one of the latest AI achievements. Artificial intelligence called Alpha Go won several times in a row in Go game match against the actual world champion. This was not expected as AI knowledge is still considered to be in its the early stages.

In this paper, issues of AI and its application are described in prediction of the next appropriate move in Go game match. Convolutional neural network (CNN) is used for this purpose. Several network models and configurations are explored and evaluated. Designed models are trained and tested with already played publicly available Go game match records.

2 Go game and its rules

Go is a popular strategic board game from ancient China. Its rules are relative simple, however, they allow plenty of moves. That is the source of complexity allowing various strategies. It is played by two players: one with white stones, the other with black ones.

2.1 Board

Go game board is not like chess. In Go, the stones are placed in the intersections of horizontal and vertical lines. The number of parallel horizontal and vertical lines is usually the same. Go play board can have a different number of lines based on game difficulty level. The board usually has 9 x 9, 13 x 13 or 19 x 19 lines. The professional match is always played on a board 19 x 19 as illustrated in Figure 1. For the final experiments in this thesis, this dimension is used.

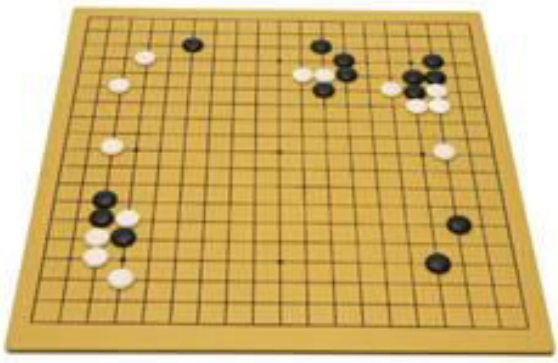

Figure 1. Go game board with dimension 19 x 19

2.2 Course of the game

The black player moves first. The players take turns. After several moves, strings of stones appear on the board. *String* is a structure composed of neighboring stones of the same color directly connected through horizontal or vertical board line (i.e. not diagonally). A stone placed alone (without a neighboring stone of same color) is a string as well. (British Go Association 2018.)

Every placed stone has *liberty* – the number of not placed positions in the neighborhood, directly connected through horizontal or vertical board line. For example, if the first stone is placed on the non-edge position, it has a

liberty of 4. If there is a stone placed on a direct neighbor position (i.e. connected through horizontal or vertical board line), the liberty of original stone decreases by 1. Stones in string share their liberties with each other. Therefore, the string liberty is the sum of liberties of all contained stones.

A string is erased if its liberty drops to zero - when the opposite player encloses it in terms of direct neighbors (Figure 2). If the move decreases the liberty of strings of both players to zero, only the opposite player's string is erased. (British Go Association 2018.)

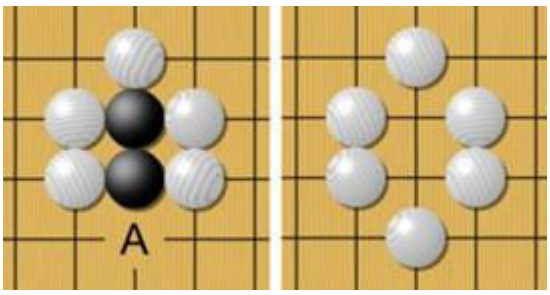

Figure 2. Black string is erased when white stone is placed in position A

Player can play *pass* move, when he/she has no appropriate position to move in his/her turn. The match ends, when both players play a pass move one after the other or one player resigns.

After the end of the match, the final evaluation of both players is made. Points are added for the remaining stones on the board and surrounded territories. The territory consists of free positions on the board. Player controls the territory, if he/she can defend it against an opposite player's attack. Otherwise, the opposite player controls the territory. Territory not controlled by any of the players is neutral, and no one gets points for it. Finally, each player gets one point for each free position in territories under his/her control. The player with the most points wins the match.

There are more Go game rule sets, which are usually regional based. There are two main ones: Japanese (Cano 2018.) and Chinese (Davies 2018.). They differ just in small details. The scoring system also depends on rule set (senseis.xmp.net 2018.).

2.3 Handicap

Go game match should be always on equal terms. However, it is not always possible - for example, in a match of an amateur and an intermediate player. If there is very little difference in the players' skills, the weaker one moves first - it is some advantage. If the skills difference is greater, there are several kinds of handicaps to make the match equal. The first one is to appropriate extra points for the weaker player. The second one is about giving some extra stones for the weaker player. The stones are placed on the board before the game starts. (British Go Association 2018.)

2.4 Ko rule

Go game board contains unique positions of stones. Every move changes the position of stones to another unique position, because on the board there is a different number of stones. However, string erasing can change the position back to in a position already played in the match - not unique. Ko rule bans moves, which lead to not unique board position. (Wikipedia 2018.)

Example is depicted in Figure 3: Black player just placed a stone on the position marked with number 1. That erased the white stone from the position marked by a red circle. If white player placed a stone on the position marked by red circle again, the board would get into the position of stones before the black player moves, which can lead to an infinity loop of moves. Ko rule does not allow the white player to move directly to the position marked by red circle. Instead, the white player must place a stone to a different position. Thereafter, the board has a different position of stones in the next round and the white player can again place his stone to the position marked with a red circle. (Wikipedia 2018.)

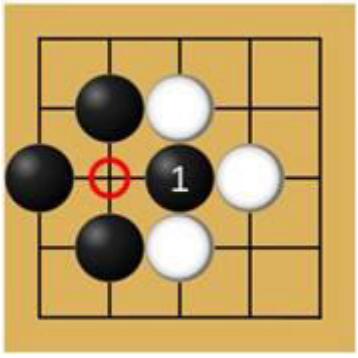

Figure 3. Ko rule

3 Neural networks in Go game

In 2016, the first matches were played between Go game world champion Lee Sedol and AI Alpha Go, developed by corporation DeepMind, part of Google (Deepmind technologies limited 2018.). In the matches Alpha Go, won surprisingly and unambiguously. It became the first AI surpassing the human world champion in Go game. Alpha Go is designed as a combination of two algorithms very often used in Go game: neural network and Monte Carlo tree search. The first part of Alpha Go training was based on records of already played professional games. The training continued with reinforcement learning, where Alpha Go played against itself. (Stanek 2018.)

Using AI and especially Monte Carlo tree search (MCTS) for Go game bots was common before the success of Alpha Go. MCTS is learning to play Go game using the records of already played matches. Every match represents a sequence of moves. In the end of this sequence it is known, which player won the match. MCTS needs to process a huge number of matches to create a tree of moves. The trained tree contains statistical data of moves and it is possible to see, which move led to winning with what possibility. MCTS to prediction of next move just choose that move, which has the biggest probability to win the match. (Burger 2018.)

With the board of dimension 19 x 19, in the match there are approximately 10^{170}unique positions of stones. To compare, it is estimated that the whole universe has around 10^{80} atoms. It again proves how extremely highly complex game Go is. Also, it proves that MCTS can contain just a very small part of all possible moves. Even when MCTS is trained from a high number of records, there are still moves, which MCTS cannot predict correctly. (Burger 2018.)

A new approach to how to predict the next move is to use a convolutional neural network. It is mainly used for processing images and searching for patterns in them. Go game matches are filled with many complicated patterns composed of stones. These patterns define which next move is appropriate for actual position of stones. For example, Go very often uses a pattern called *eye*, which provides a strong defense against erasing of string on board. In Figure

4, there is a common situation from matches, where positions marked by red circles are the centers of “eyes”. (British Go Association 2018.)

Convolutional neural networks search for patterns, which they were trained to find. The training process consists of inserting input data and the expected output value for the neural network. The expected output data is compared to the network’s output. The network can evaluate the error of own output and adapt its internal parameters. The next input should contain the network’s output nearer to expected output.

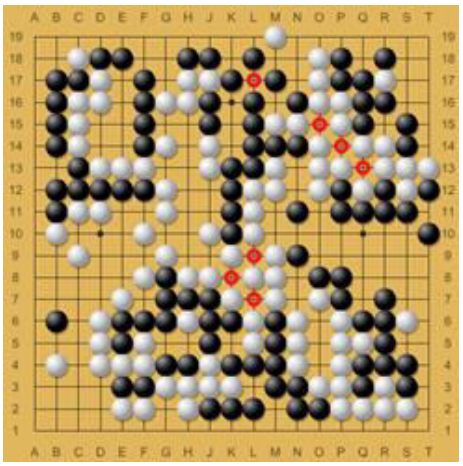

Figure 4. Red circles mark pattern called “eye”

Pattern searching provides a significant advantage in comparison to MCTS, which uses statistics. It creates relationships between trained patterns and output values that lead to the generalization of given task. Thus, correctly trained network can in some measure provide the right output of input data, which was not used for network training. It means that a neural network can cover more possible inputs than MCTS.

In this paper, convolutional neural network is used for prediction of the next appropriate move in Go game match. It is supposed that it is possible to train convolutional networks to search for patterns, which are crucial for playing of Go. Prediction of the next move in Go game match is equivalent to classification problem, where every board position is one independent category, which can be the output of a trained network. The idea of convolutional neural networks used for prediction of the next move has also been explored in research papers, which were used as inspiration for these experiments. (Clark, & Storkey 2018; Huu, Jihoon, & Keechu 2018.)

Sgf records of already played games were used as input data, which is freely available to download. They were transformed to datasets and used for network training.

Neural networks consist of a huge number of implemented algorithms, optimizations and programs managing side hardware (mainly GPU). It was necessary to use a framework that creates and manages the desired network. There was a choice of two frameworks - Caffe and TensorFlow. Caffe framework was chosen for its layer orientated structure and higher specialization for convolutional neural networks issues.

4 Neural networks

4.1 Introduction

Artificial neural networks were originally designed to simulate neural paths of real organic organisms. However, idea of neural networks was developed to an independent field of machine learning, where it is successful for ambiguously defined problems. Neural networks are used mainly for (Karpathy & Johnson 2018a.):

- *Clustering*, grouping of data according to similarities to unknown groups (clusters).
- *Regression*, searching for relationships between data inputs.
- *Classification*, assigning of objects based on similarities to one of already defined collections (categories).

4.2 Structure of neurons

The main unit of neural networks is a neuron. Every neuron consists of (Karpathy & Johnson 2018a.):

- input connections to previous neurons
- input data processing
- output connections to next neurons

For every pair of connected neurons there is an assigned value – *weight (w)*, which is multiplying every value incoming from the input neuron. Neuron basic equation is:

$$v_i = x_i * w_i \qquad (1)$$

where v_i is value from input neuron i after multiplication by the weight w_i, x_i is input value of $i - th$ input neuron. Neuron sums all input neurons values

$$S = \sum_{i=0}^{n} v_i + b = \sum_{i=0}^{n} x_i * w_i + b \qquad (2)$$

and sum is used as input to activation function

$$N = f(S) = f\left(\sum_{i=0}^{n} v_i + b\right) = f\left(\sum_{i=0}^{n} x_i * w_i + b\right) \qquad (3)$$

where

S is sum of all values of all input neurons multiplied by weights,

N is output value of neuron for given inputs,

f is activation function (explained in next chapter)

n is number of inputs,

b is bias value, it is used for adjusting of activation function.

N is output value, e.g. it is used as input value for next connected neurons or it is part of final output of neural network (Figure 5).

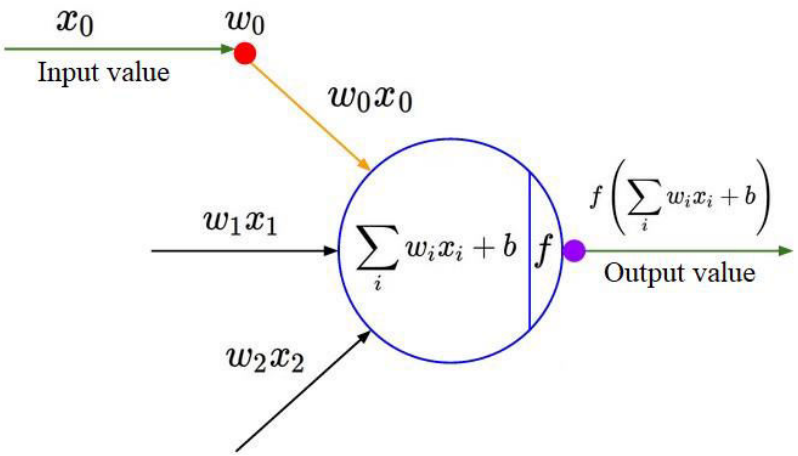

Figure 5. Neuron diagram

4.3 Activation function

Every neuron can respond with a specific output to different inputs. There are inputs, which involve very high output; the neuron is very active. Otherwise, other inputs involve a very small output; the neuron is very inactive. This behavior is caused by activation function of neuron. (Karpathy & Johnson 2018a.)

The input of every activation function is a value of S from equation (2), where a specific mathematical operation is performed. The function output is possible to adjust with bias value in neurons. A positive bias value causes a higher activation of neuron, negative value causes lower activation of neuron, where neuron had same input values.

There are more kinds of activation functions (Karpathy & Johnson 2018a.):

1. Sigmoid: Sigmoid nonlinear function (Figure 6) was often used as activation function in the beginning of neural networks. Domain of the

sigmoid function are all real numbers and output range of function is collection (0,1). Thus, for every real number exists activation of neuron - number from 0 to 1. Sigmoid function has equation:

$$\sigma(x) = 1/(1 + e^{-x})$$

Nowadays, sigmoid is used very rarely. It has a disadvantage:

Sigmoid function saturates and kills gradients. This function has a problem on the very sides of outputs of function, around 0 a 1. It is very inflexible to input changes. For example, if there is high input value (100), then output of function is very near to 1. However, when 10 times higher input value is used, the output of function is still almost the same near to 1. The output is not changing significantly, when the input is high or very small. This behavior causes a problem while in training process of network. Adjusting of weights and bias values is minimal while neuron is high or low activated. This fact can slow down training process.

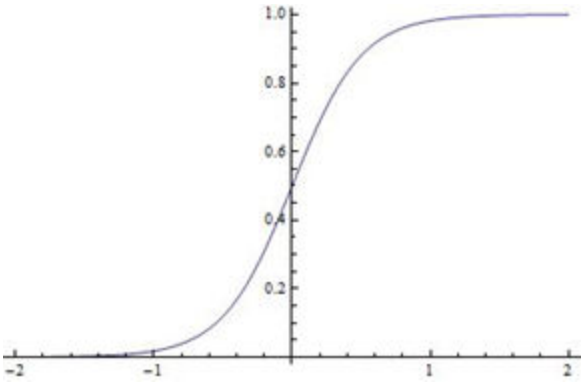

Figure 6. Sigmoid function chart

ReLU: ReLu function (Figure 7) is very used these days as activation function. ReLU function has equation:

$$f(x) = max(0, x)$$

ReLU's output value is not changed input value if it is higher than zero. For inputs smaller than zero, it returns zero. ReLU is simple for computing and easy to implement. It does not have the problem of saturated and killed gradients as sigmoid function.

The disadvantage of ReLU function is the application of high gradient while training process. It can change weights to values, where output value is always lower than zero. Then it is not possible to change weights back, while there is a zero activation. These blocked neurons will output zero values for whole network training process. It can ruin all training process, if there are more blocked neurons. (Karpathy & Johnson 2018a.)

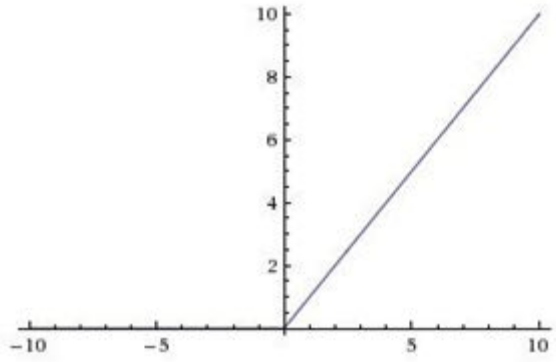

Figure 7. ReLU function chart

2. Leaky ReLU: Similar as ReLU, also leaky ReLU returns input value not changed, if it is positive. If there is input value lower than zero, it returns input value multiplied by constant α, which is set as very small number. This way, output value is never zero, what solves ReLU's blocked neurons problem. Leaky ReLU is massively used in case of neuron activation nowadays. Leaky ReLU function has equation (Karpathy & Johnson 2018a.):

$$f(x) = \begin{cases} x & if\ x > 0 \\ \alpha * x & otherwise \end{cases}$$

4.4 Layer structure of neural networks

Single neuron is not able to create a completely working neural network. Complex behavior is possible to achieve only when there are more connected together. It is helpful to define the group of neurons, which together can solve a certain part of task, which the neural network solves. This group of neurons is called the *layer* of neural network. The whole neural network consists of several layers connected to a single directed acyclic graph. Cycle is not allowed. It would make endless data flow through network layers. Special kind of neural network is recursive network, where is allowed cycle with certain

loop number. (Karpathy & Johnson 2018a.) However, this kind of network is out of the scope of this paper.

The first network layer, *input layer*, provides input data and data preparation. The last network layer, *output layer*, evaluates the output data and provides the final output of the network. Layers between input and output layers are called *hidden layers*. Data flow has direction from input layer, where output of every network layer is input for next layer.

Basic layer commonly used in all kinds of neural networks is *fully connected layer*. It contains neurons, which are not connected each other, but every neuron has input connection to every neuron from previous layer and output connection to every neuron of next layer (Figure 8).

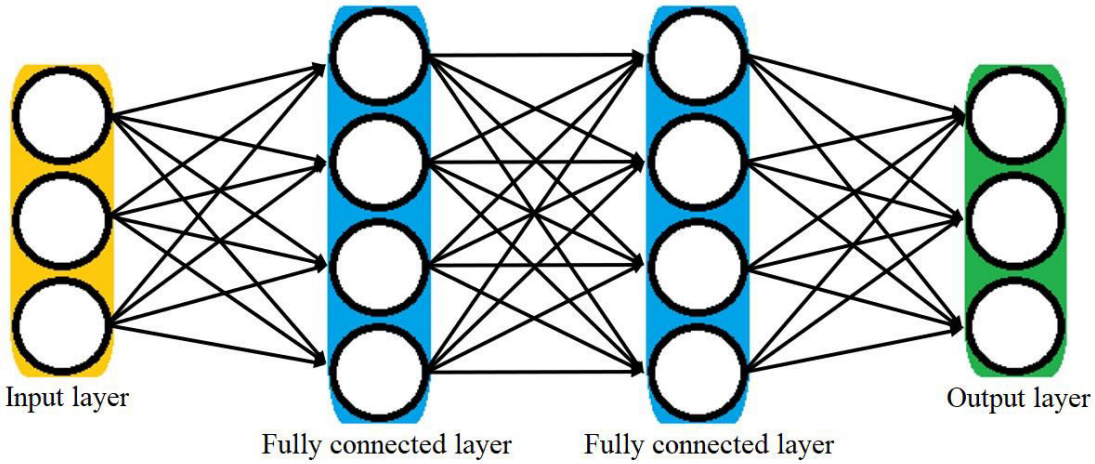

Figure 8. Neural network with two fully connected layers

4.5 Neural network parameters

Neural network can return right outputs only in case it has properly adjusted weights and bias values of neurons. Weight is dedicated for every connection of two neurons. It defines, how much input value of input neuron affects the output of the neuron. Bias is adjusting activation function of neuron. Weights and bias values are modified and adjusted while training process of network to provide more accurate outputs of given task. Weights and bias values are collectively called *learnable parameters* (or just parameters) of network. (Karpathy & Johnson 2018a.)

The complexity of a neural network can be measured by the number of learnable parameters. For a practical example, network model from Figure 8 can be used. The model consists of one input layer with three neurons, two fully connected layers - every of them has 4 neurons, and output layer with three neurons. Input layer has 3 x 4 connections to first fully connected layer, between first and second fully connected layers are 4 x 4 connections and between second fully connected layer and output layer are 4 x 3 connections. Weight is dedicated for every connection between neurons. Thus, the example model has 40 weight parameters. Moreover, every neuron has one bias value. The only exceptions are input neurons, which do not have bias values. Thus, the example model pictured in Figure 8 has 51 learnable parameters - 40 weights and 11 bias values.

4.6 Output calculus

Dividing neural network into layers brings simplification to output calculus. The main reason is, that input data, learnable parameters and outputs of layers can be represented as matrixes and matrix operations can be applied.

For practical example, Figure 8 can be used again. Matrix X represents network's input with dimension [3 x 1], so every neuron of input layer has one data value. All connection weights between input layer and first fully connected layer are represented as matrix W with dimension [4 x 3]. Weights of every neuron for their input connections are in rows of matrix W. Thus, multiplication of matrixes $[W]$ x $[X]$ represents sum of all input values (equation 2) without bias value, for every neuron. Then, the output value of every neuron of the first fully connected layer is possible to calculate this way (Karpathy & Johnson 2018a.):

$$[N_M] = f_M([W]\ x\ [X] + [B]) \qquad (4)$$

where
N_M is matrix containing output values of neurons (equation 3) of layer,
f_M is activation function calculated for every element of matrix,
matrix B has dimension [4 x 1] and represents bias values of neurons of processed layer.

Similarly, the next layer can be calculated, where matrix N_M is input matrix. This way it is possible to easily calculate the output value of neural network. The process of network output calculation is called *forward pass*.

Matrix operations allow computing a larger amount of inputs at once. Insert more inputs at once is mainly used in the network training process, where the whole group of input data is used as input. In the last example, the input matrix X had dimension $[3\ x\ 1]$, which is the amount of single input data. However, a matrix with n of input data sets can be used similarly. In this case, the input matrix has dimension $[3\ x\ n]$. The above described algorithm can process this new matrix in the same way. The benefit is calculation parallelization of all inserted inputs, which boosts the speed of network training. (Karpathy & Johnson 2018a.)

4.7 Input and output data representation

Input data can represent different kinds of information. There can be simple numerical data, when an analytical problem needs to be solved. For example, the number of sunny days in year, or the price of a house in given location. Another type of a problem that can be solved is image processing, where the input is image, pixels with color values. Image input can contain more data dimensions. In the case of black-white image, there is just a simple two-dimensional array $[h\ x\ w]$, where h is height and w is width in pixels and values of array are shades of gray. For images with three colors (RGB), there is a third dimension called *channel*, which represents the value of shade for each of the basic RGB colors. For every pixel of image in every channel it is necessary to define a single neuron in the input layer. Thus, the input layer must be specialized for the expected input form and dimension.

Every neuron of output layer produces just one output number value. Output is matrix of numbers, if there are more neurons in output layer. For example, output of classification problem is usually matrix, which represents probabilities of all possible output categories. Thus, if output matrix has dimension $[4\ x\ 1]$, the values of which are $[0.1\ 0\ 0.9\ 0.2]^T$, then the final output category has index 2 and probability 0.9, because probability of category 2 is the highest in matrix.

4.8 Neural network training

New initialized network has usually just randomly chosen weight values. It means, network cannot provide the right outputs before network's training process. The training process tries to find appropriate network parameters, which can solve given task. Network training is based on the method trial and error. A high amount of analyzed input data is needed for training process. For every one of these inputs the expected right output needs to be known. Data gathered in this way is called *dataset*.

More kinds of datasets are used while training network. The main and the biggest dataset is training dataset, which is used for training process. However, one of the advantages of neural networks is generalization of given task. Thus, outputs of network should be right also for inputs, which were not used for network training. That is the reason, why a different, smaller dataset is needed which will check generalization of the task, testing dataset. These two kinds of datasets must be as much independent of each other as possible. They should not contain the same input data. Testing dataset is usually four times smaller than training dataset. (Shah 2018.)

Neural network does not use all dataset data at once. There is usually not enough RAM memory space in the computer. The network uses just a small group of dataset inputs in one step. It is more effective because parallelization can be used. This small group of inputs is called *batch* input. Batch has the same size during the whole training process. Batch contains just random inputs from dataset; however, the same input is used again only when all other dataset inputs were already used. Processing of one batch is called *iteration*. Processing of whole dataset is called *epoch*. The training process usually consists of several epochs and the whole dataset is processed for more times. (Nielsen 2018b.)

Learning process consists of sending a huge amount of input data to the network. The network compares its outputs to the expected right outputs. The difference between network output and expected output is evaluated by *error* (also called loss) value. There is *cost function* (also called loss function), which is used for computing of error value for every processed iteration. There are

several different cost functions. They are described on the website. (Bourez 2018.)

Error value is an indicator, how well a network is trained. Thus, the training process is an optimization process, where network parameters are to be found with minimal error value, evaluated by cost function. To solve this optimization task, it is necessary to use advanced mathematical methods: backpropagation and stochastic gradient descent (SGD). Detailed descriptions of these methods are on the websites. (Nielsen 2018a; Nielsen 2018b.)

4.9 Hyperparameters of neural networks

Every neural network contains settings - *hyperparameters* describing its behavior during initialization, learning process, testing and practical usage in deployed application. Hyperparameters define, whether a network can solve given task or whether it is possible to train network.

Hyperparameters are the basic structures of network described earlier: number of layers, type of layers and number of neurons inside, activation and cost functions. There are also some other hyperparameters, which define learning process behavior. More detailed description of all hyperparameters is on the website. (Karpathy & Johnson 2018b.)

Learning rate: Learning rate is basic hyperparameter, which defines, how radically will network change parameters while training process. So, it affects, how fast the network will learn while in the training process. If learning rate is too small, the network training is too slow. If it is too high, network is changing parameters too chaotically and it cannot find optimal parameters. The effect of different learning rate values is pictured in Figure 9. The learning rate is usually decreasing while in the training process, which helps to find better network parameters. The learning rate is a very sensitive hyperparameter, the ideal value can be different for every network configuration. The best way to find an ideal learning rate is to try and analyze the training process output. The value of learning rate is usually between 10^{-2} and 10^{-6}. (Karpathy & Johnson 2018b.)

Batch size: The number of inputs in a batch is also one of the hyperparameters. A smaller number of batch inputs causes faster computing

of iterations, yet, the error of iterations will be not stable. A higher number causes smaller number of iterations; however, more stable error during the training process. (colinraffel.com 2018.)

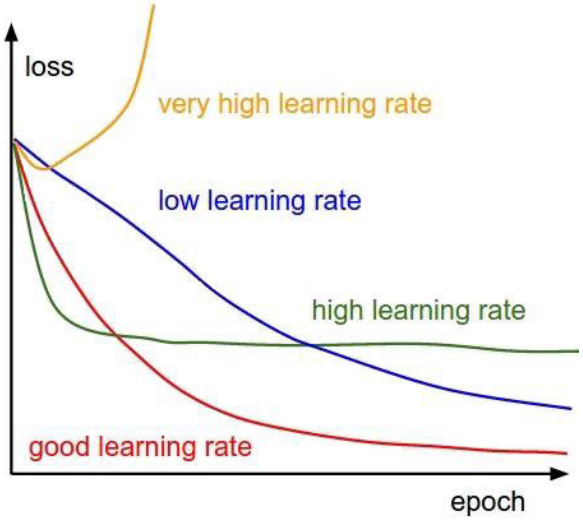

Figure 9. Effect of learning rate to error (loss) while training process

Maximal iteration number: Iterations provides information on how many images were already processed. It is possible to use that as limit for training process. The training process will stop, when it makes a certain number of iterations. (colinraffel.com 2018.)

Momentum: Momentum simulates the inertia value from physics. Every change of parameters represents movement. While changing a parameter, there is still the effect of "momentum" from changes before. Momentum helps to optimize algorithms to overcome local minimum, where algorithms would stay normally. (colinraffel.com 2018.)

Regularization: When training process is stopped too late, then there can be a problem with overlearning of dataset inputs. This state is called *overfitting*. The network tries to adjust parameters to return the same outputs as dataset's expected outputs. While overfitting, network adjusted parameters too well. The network was able to solve all inputs from the dataset; however, it did not generalize given task. An example is pictured in Figure 10. (Karpathy & Johnson 2018c.)

There are many methods, how to reduce overfitting. Some of them are L1 and L2 regularization. More detailed description of these methods is on the website (Scheau 2018.)

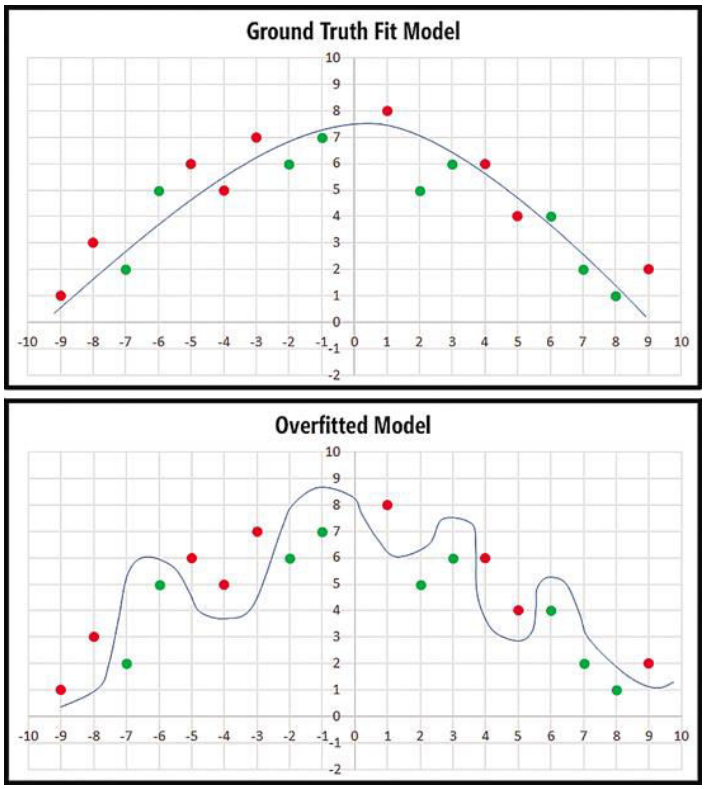

Figure 10. Visualization of regularized (upper) and overfitted (lower) network

4.10 Convolutional neural networks

Convolutional neural networks are a special kind of networks. They specialize in image processing tasks (face recognition, object detection, object tracking on the sequence of images). It is also possible to use convolutional networks for tasks, which can be transformed to image processing tasks. In general, convolutional networks can solve tasks with a fixed structure of data; changed data order would change interpretation. For example, image data would change interpretation if some rows or columns of pixels were changed in the image. Text would be interpreted in a different way, when the order of words in sentences is changed. However, processing of database data is an inappropriate task for convolutional networks. The reason for this is that database data can be represented in a different order of columns or rows, however, data information is still valid in the same way. (Karpathy & Johnson 2018d.)

It is supposed, that a network's input data is an image or its data representation. Every image pixel is represented in a computer as value of shade of some basic color. One channel image has just one value of shade of gray color for every pixel. A three-channel image (RGB image) has three values for every pixel, the shade of red, green and blue color. Thus, an image input is represented in a computer as c two-dimensional arrays with dimension $h\ x\ w$, where h is height, w is width and c is number of channels. Every array consists of pixel values of a specific color. Moreover, convolutional networks can process image inputs which have even more than three channels of data.

4.10.1 Convolution

Convolution is an image data processing method. It tries to detect patterns in image input. Convolution method output is the map of pattern occurrences in image input. Patterns that convolution is trying to detect are called *filters* (or kernels). If an image input has dimension $h\ x\ w\ x\ c$, then the dimension of every filter is $hf\ x\ hf\ x\ c$, where hf is usually much smaller than h and w. The channel number of filter is always the same as the channel number of input image. Thus, filter is c of square matrices with dimension hf, which contains values describing pattern. (Santos 2018.)

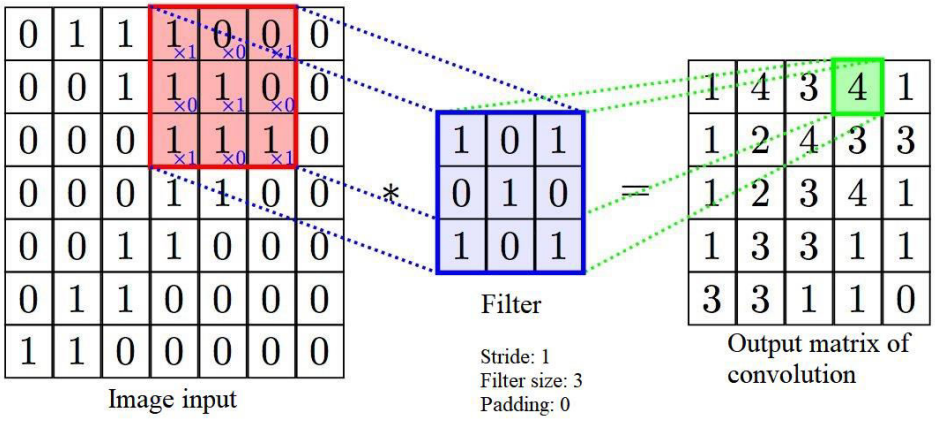

Figure 11. Convolution process

The filter is applied to every position of the input image. The filter output for these positions is the value defining how a good filter pattern fits for the applied position of input image. The output value is higher, when filter pattern fits more with the pattern on input image. The output of the entire convolution

process is a matrix containing values of the filter applied on the input image (Figure 11).

Convolution process can be configured by three attributes: stride, filter size and padding. Stride defines how many pixels are between two neighbor positions, where a filter is applied. The filter size defines the dimension of filter square matrix.

Output convolution matrix has smaller dimension than input image. It can be a problem sometimes when a very small image is in processing. Padding deals with this problem. Padding is a border around the whole image input and through all channels of image input. It is usually filled with zeroes. This border scales up the input image and scales up the output matrix (Figure 12). The padding value represents the width of the applied border. (Santos 2018.)

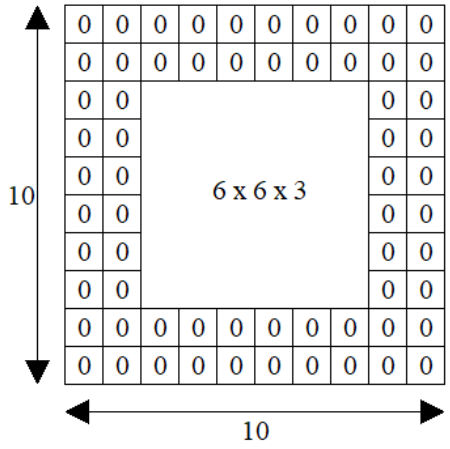

Figure 12. Padding of image input with value 2

4.10.2 Layers of convolutional networks

Convolutional networks have other extra layer types than the *fully connected layer*: *convolutional layer*, *activation layer* and *pooling layer*.

Convolutional layer can contain more filters applied for input data at once. All filters of a layer have the same dimension. The output of every applied filter is a matrix. If convolutional layer has n filters, then the output of this layer is n output matrices. All these output matrices are joined into one output, the dimension of which is $h' \times w' \times n$, where h' is height, w' is the width of filter output matrix and n is the number of applied filters. For every convolutional

layer it is necessary to define attributes: stride, padding, filter size and number of filters. (Karpathy & Johnson 2018d.)

In a convolutional network, models usually consist of plenty of convolutional layers. The output of one convolutional layer is the input for the next layer. Thus, every convolutional layer tries to find patterns in the output of the previous one. While training, the processes are convolutional layers specialized in detecting some certain patterns. The first layers usually detect basic features: horizontal, vertical, diagonal and curved lines. The output contains information, where features of this kind are on the image. This output is the input for the next convolutional layer, which uses its patterns to join the basic features to more complicated features: corner, circle, or triangle. Every following convolutional layer detects a more complicated object. (Karpathy & Johnson 2018d.)

Activation layer represents activation function in the network model. Activation layer performs a specific mathematical operation of activation function for every input value. Output data has the same dimension as input data. Activation layer is usually placed after convolutional layer.

Pooling layer shrinks the dimension of an input matrix. Pooling layer has to define similar attributes as the convolutional layer: stride and filter size. Filter is also applied in the same way as convolutional layer filter. From positions to output matrix, the pooling layer takes just maximal values, where filters were applied. The output matrix is smaller than input matrix and contains just maximal values the from input matrix (Figure 13). Every channel is processed individually and the number of channels stays the same. Pooling layer helps to decrease the number of learnable parameters of a network and reduces the overfitting effect. (Karpathy & Johnson 2018d.)

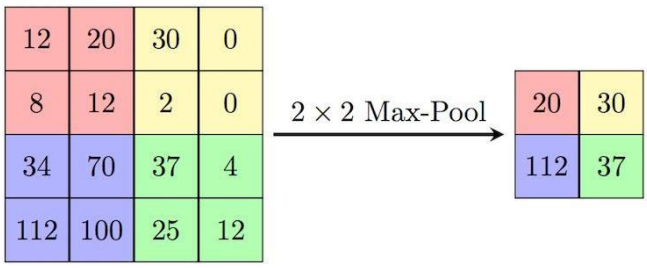

Figure 13. Pooling layer with dimension 2 x 2 and stride 2

5 Caffe framework

Caffe is an Open source platform designed for deep learning and developed by Berkeley AI Research (BAIR) and community contributors. It is characterized by modularity and speed. Caffe was designed and created as the dissertation thesis by Yangqing Jia at UC Berkeley. (Jia 2018b.)

Caffe framework allows to design and use one's own neural network. Caffe can run the training process based on configuration files, which have extension ".prototxt". These files define the network architecture and hyperparameters of network using a declarative language like XML. (Jia 2018d.)

It is also possible to use the interface of one of the higher languages - Python, C++, Matlab. Interfaces allow users to manual insert inputs to network, check outputs of individual layers or values of learnable parameters. (Jia 2018c.)

Caffe training process periodically creates two types of snapshot files, which record the actual state of network training. The first file has extension ".caffemodel" and represents the record of all learnable parameters of the network. This file is used for deploying a trained solution or for testing purposes. The second created file has extension ".solverstate". It allows users to continue training process from the state, where the file was created. (Jia 2018d.)

Caffe framework allows running neural network processing on CPU or GPU, however, often it is several times faster to use GPU in compare to CPU. It is ideal to use graphics card by Nvidia with CUDA core technology. Thus, Caffe supports cuDNN (cuda deep neural network) libraries, which provides a speed boost of fundamental neural networks calculations. (Jia 2018c.)

Caffe installation primarily consists of support software installation and downloading of source codes from free GitHub repository. Then it is necessary to modify the configuration file and compile the project. The whole installation tutorial is on Caffe home websites (Jia 2018a; Xin 2018.).

6 Implementation

6.1 Created programs

All programs were implemented on Linux OS in C++ of version 11. The final program has two parts:

- Go_game_and_data_preparation: the program allows to create datasets for training process and to play Go with prediction provided by trained network.
- Go_game_train: the program controls training and testing of network based on created datasets.

6.2 Program - Go game and data preparation

Figure 14 pictures the UML diagram of the program Go_game_and_data_preparation.

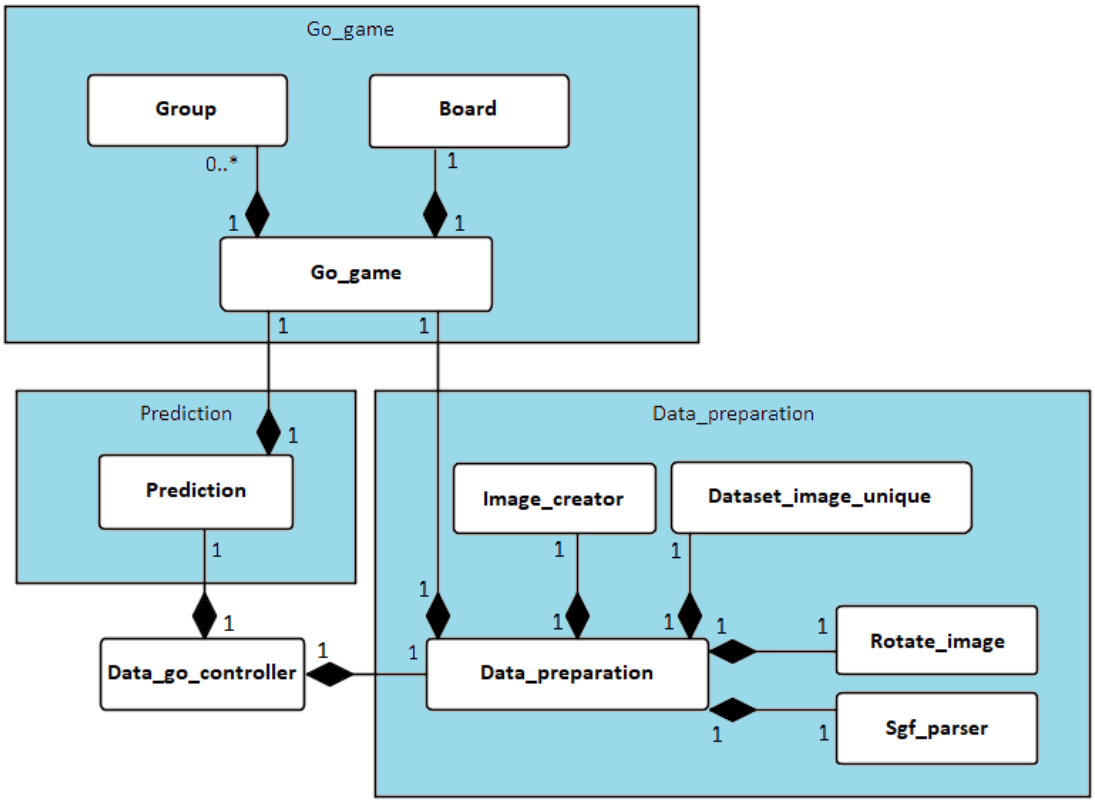

Figure 14. Class diagram of Go_game_and_data_preparation program

The independent part of Go_game_and_data_preparation program is game Go_game consists of classes Go_game, Group and Board. It is an implementation of Go game, which allows to simulate games from game

records, or play a game against trained network. Class Go_game receives move positions by players, evaluates moves by used rule set and returns output - actual stone positions of board if move was legal; otherwise, illegal move notice. Class Board contains a two-dimensional array representing the game board. Moreover, it provides basic board calculations: liberty level of stone groups. Class Group is the side data structure of board representing a connected group (string) of stones with the same color.

The next part of Go_game_and_data_preparation program is class Prediction. It is an extension for the Go game implementation described above. It is used for playing Go game, where a player can see the suggested next move by the trained network. The suggested move is shown, when a player sets any unparsable input, at least an empty “enter” button. Class Prediction uses Caffe for network initialization from a snapshot file created while the training process. Actual stone positions on the game board are transformed to network compatible data form, OpenCV Mat dense array. It is used as input for initialized network. The returned output is the suggested next move.

The last Go_game_and_data_preparation program part is Data_preparation. It consists of classes Data_preparation, Sgf_parser, Image_creator, Dataset_image_unique and Rotate_image. Class Sgf_parser is used as game record parser and record validity checker. The parsed records are processed by Data_preparation class. Data_preparation initialises Go game instance and simulates the game with a parsed game record. Data_preparation gathers data from the simulated game; stone positions on the game board. The gathered data is sent to Image_creator class instance, which creates a dataset of images for network training.

Classes Dataset_image_unique and Rotate_image allow the program to make special dataset modifications. Dataset_image_unique can create a dataset, which contains just the unique image inputs. Rotate_image allows making data augmentation of the dataset (Described more in detail in chapter 6.5).

The last Go_game_and_data_preparation program class is Data_go_controller. This class controls all other program parts and activates them with a set of input parameters.

6.3 Program - Go game train

Figure 15 illustrates the UML diagram of program Go_game_train.

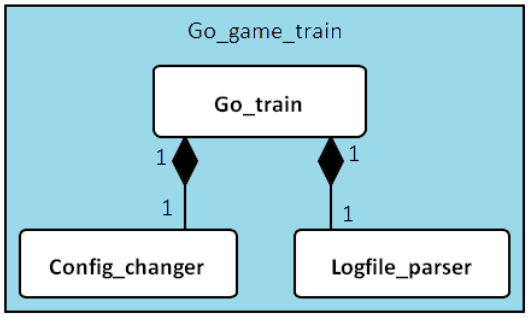

Figure 15. Class diagram of Go_game_train program

Program Go_game_train consists of three simple classes: Go_train, Config_changer and Logfile_parser. The main run class is Go_train. It uses Caffe framework to start and control the training process with specified configuration. Caffe text file output report contains important data gathered during the training process: network errors while training and number of iterations. Caffe output report file is parsed by Logfile_parser class into a more clear form at the end of the training process. Class Go_train also provides continue training and evaluate trained network methods. The last class is Config_changer, which allows automatic run of training processes with prepared configuration sets. It helps with continual searching for appropriate network configuration.

6.4 Game records

It is necessary to gather raw data, which can be used for dataset creating. Go game has an advantage in this case: huge community and popularity of this game. There are several websites, where it is possible to download reports of games played by players of different ranks. These games are usually saved in text file record format with extension ".sgf" (smart game format).

More than 100 thousand Go game records played by players of rank from intermediate (rank 1–7d) to professional (rank 1–9p) have been gathered. These records have usually been downloaded from a website (Görtz 2018.).

Sgf format is very simple and intuitive. At the beginning of the record there is general match information: size of board, name and rank of players, handicaps

of players, winner color and final score, rule set and so. The next part contains moves of players one after the one. Every move has the color of player and position of move.

Every information in the record consists of tag and value in square brackets. For example, code SZ[19] represents information “size of board is 19 x 19”. Black player has moves with tag B, and white player has moves with W, delimited by semicolon. The position of a player’s move is in square brackets. Figure 16 shows the coordinate system used in sgf files. The formal and strict structure of Go records allows simple parsing of data.

Example of sgf record:

(;SZ[19]	Size of board
PW[player1]	White player name
WR[6d]	White player rank
PB[player2]	Black player name
BR[6d]	Black player rank
DT[2018-03-01]	Date, when match was played
PC[The KGS Go Server]	Server, where match was played
KM[6.50]	Handicap - points added to white player
RE[W+Resign]	Match result - white player won, when black player resigned
RU[Japanese]	Used rule set
;B[pd];W[dp];B[qp];W[dc]; ...)	Player’s moves (pictured just four of them)

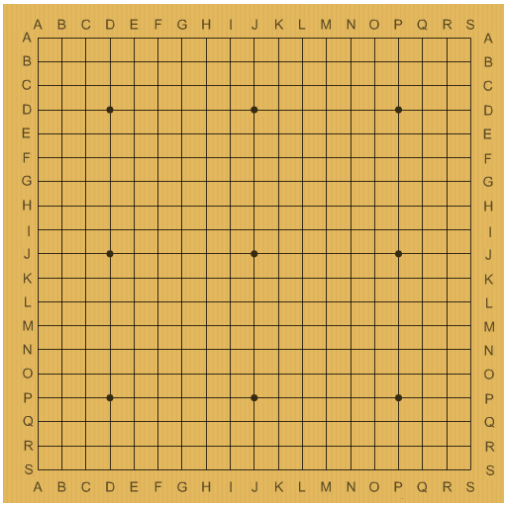

Figure 16. Go board with coordinate system

6.5 Dataset creating

The dataset for Go game contains pairs: the position of stones placed on the board and the expected right next move to this position. The positions of stones were gathered after the moves of the player who loses the match. The right moves belonging to these positions are the moves played by the player who wins. Thus, the network trains to play moves played only by the winners of matches.

For a neural network it is easier to represent the output category (suggested next move) by the number of category. It is necessary to assign a number of category for every position on the board. For a board with dimension 19 x 19, 362 numbers are assigned, where 0 is the number of position in the left top corner of the board, 18 is the number of position in the right top corner of the board and 360 is the number of position in the right bottom corner of the board. Number 361 was assigned for pass move.

Sgf records give a huge amount of data from already played games. However, sgf records do not contain full information of the position of stones placed on the board. Sgf records are missing information of stones removed from the board. It was necessary to implement a Go game with rules. For every sgf record a new Go game match was started, where the record provides the moves. Match simulated within implemented game rules ensured that the position of stones is valid after every played move. Thus, every played move simulated within the implemented game can be used as the next dataset input.

For every position of stones on the board in dataset one three- channel RGB image was created with the same dimension as Go game board (19 x 19). Blue color channel is dedicated to stones of the first player, for whom the neural network makes prediction. Green channel is dedicated to the second player's stones. Red channel is dedicated to the positions without any stone. Every pixel in the dataset images has a maximum value (255) just in one channel. Other two channels are zeroes.

Thus, datasets contain images with the same dimension as Go game board; the background of images is red, blue pixels are stones of the first player and green pixels are stones of the second player (Figure 17). Dataset images were

saved with png extension, since it was necessary to use lossless compression image format to keep images in valid form.

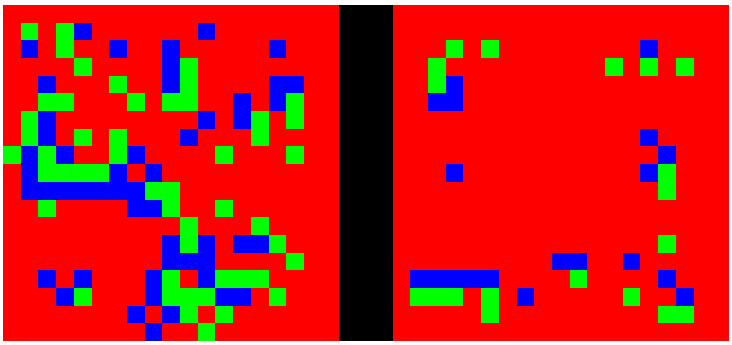

Figure 17. Two dataset images with dimension 19 x 19

Go board can be rotated by 90°, 180° and 270°, however, the board still contains the same information. The board is symmetric as well. It has four symmetry axes: parallel with the x axis and y axis, both through the middle of the board, and two diagonal axes. Thus, every image of a dataset has rotated representations, which are as valid as the original dataset image. However, for neural network it is not the same dataset input. The rotated image is a different input, and it must return a different (rotated) output. It is very common to add these rotated images to dataset as well. The process of dataset input modification (and multiplication) where inputs stay valid is called data *augmentation*.

It is possible to create combinations of the rotations and symmetry flips described above by using one of symmetry flip and rotation to create new image input. However, there are just several unique combinations. All other combinations are just mirroring to these unique ones. For example, flipping of original image by y axis through middle of the board is equal to flipping of the original image by x axis through middle of the board and rotated by 180°. In final, there are just eight fully unique augmented images (original one included) of board stone positions. Thus, the augmented dataset can be eight times bigger than the dataset without augmentation.

While training the neural network, it is very important to have the same number of outputs for every possible category in dataset. In other case, in output are statistically more preferred categories, which have higher number of inputs in dataset. To avoid this problem, inputs of categories with a lower number of inputs in dataset were multiplied. Thus, every created dataset has

the same number of inputs for every possible output category (for every possible suggested move position).

Every dataset has the main text file containing the paths to dataset images and the expected right move position for every image. This is one of the possible ways how to define a dataset for Caffe framework.

The entire dataset creating process is shown in Figure 18.

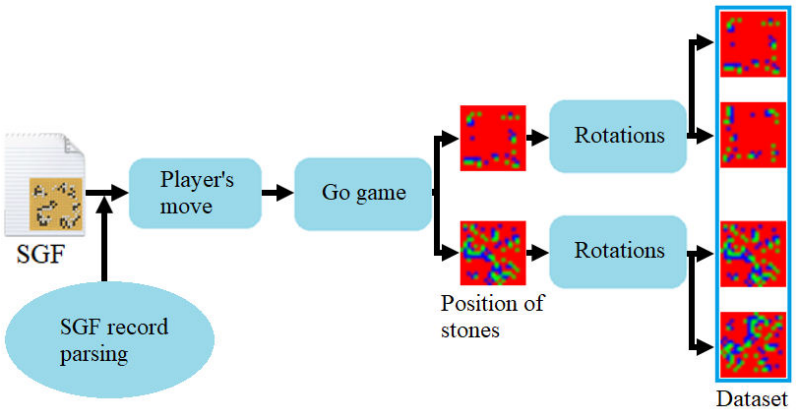

Figure 18. Dataset creating process

6.6 Dataset function testing

A properly working dataset is a crucial element of neural network training. There are many possible problems with the dataset that can ruin training: data saved in images does not represent the real position of stones on the board, or the network does not understand these images. It was necessary to prove that the dataset is created in a correct way.

Two very simple experiments were conducted. It was necessary to overfit the network. The first experiment consisted of dataset images of Go board with dimension 5 x 5. Just one stone was placed on the board (Figure 19). The dataset contained all possible positions of a stone on the board and the expected move was the position of the stone on the board. Thus, the network's task was to return the position of only stone placed on the board. This experiment was designed to check if the network can return the right number of category (right number of output board position).

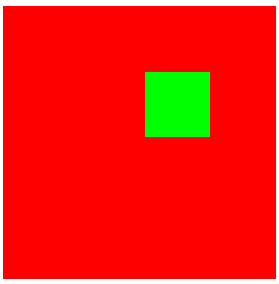

Figure 20. Dataset image with dimension 5 x 5, prepared to return stone position

The second experiment consisted of dataset images of Go board with dimension 19 x 19. Every image of the dataset contains one square with dimension 2 x 2 created by stones of the same color. The square can be placed on any location of the dataset image. The square placed in the image can be complete (with all 4 stones) or one of the stones is missing. The expected move for every image was the position where the stone was missing. Thus, the network's task was to complete the square of stones on the board (Figure 20). In the case the square was already complete, the network returned pass move. This experiment was designed to check if the network could recognize the stone structures on the board.

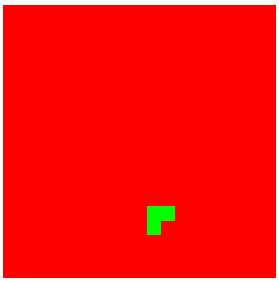

Figure 19. Dataset image with dimension 19 x 19, prepared to complete square

Both experiments were successfully tested. They demonstrated that the created datasets are appropriate and it is possible to use them for neural network training. Moreover, it is possible to train the network to recognize the basic stone structures on the board.

7 Testing of trained models

In this paper, more models of neural networks with different hyperparameters settings were trained. More dataset modifications were also used. Network training consisted of setting up hyperparameters and checking the actual training process error. The point of training process was to minimize errors. The training process was stopped when training process error did not significantly decrease in the last several 10 thousand iterations.

For network training, Nvidia GeForce GTX 1080 Ti graphics card was used. The training process of one model (while error stopped decreasing) took several days, in some cases even more than one week. Therefore, the training process was extremely time consuming, which is the reason why just a very limited number of experiments was carried out.

7.1 Training datasets

To make experiments less time consuming, smaller datasets were chosen containing from 280 to 650 thousand of image inputs. There were used from 600 to 1200 sgf records to create these datasets. To compare, Alpha Go beginning training phase dataset had around 30 million of image inputs (Stanek 2018.). Every network was trained with one of four main datasets:

- Dataset 1 - basic small dataset created by processing of 600 game records. Multiplied to four times bigger dataset by augmentation. Contains approximately 356 thousand image inputs.
- Dataset 2 - basic big dataset created by processing of 1200 game records. Multiplied to four times bigger dataset by augmentation. Contains approximately 650 thousand image inputs.
- Dataset 3 - unique dataset - every dataset image is unique position of stones in dataset. Multiplied to eight times bigger dataset by augmentation. Contains approximately 290 thousand image inputs.
- Dataset 4 - unique dataset with images normalized to 0 and 1. Every dataset image is unique position of stones in dataset. Moreover, maximal image pixel value is not 255, but 1. Multiplied to eight times bigger dataset by augmentation. Contains approximately 434 thousand image inputs.

The first two datasets can contain several the same position of stones with different expected right moves. It can cause problems while network training, because several the same inputs have more expected outputs. The third and fourth datasets contain just the unique position of stones.

Dataset four was created to check if the network has better results when it is trained with image inputs normalized to scale from 0 to 1. Normalization to scale from 0 to 1 is very popular and should help in backpropagation process while network training.

Every dataset contains the same number of inputs for every possible output category.

7.2 Designed models

20 different network models were trained for prediction of next move in Go game in this paper. Also models of different sizes were tried. The bigger tested models had the best error decrease while training process. The model architectures of these networks were very similar to a model in a research paper by Huu, Jihoon, & Keechu (2018.): five convolutional layers with one fully connected layer at the end of the model.

In the tested models, all convolutional layers are activated by ReLU or leaky ReLU activation layer. The convolutional layers have filter size of 5, padding 2 and stride 1 that keeps the same dimension (dimension of the board - 19 x 19) of data flow during the whole forward pass process. All convolutional layers have higher number of filters, except the last convolutional layer, which has just one filter. Thus, when a board with size 19 x 19 is used, then the output of the last convolutional layer is 19 x 19 x 1. This makes it easier to set up parameters to the fully connected layer. The fully connected layer is the last layer of these models. It is mapping the output of the last convolutional layer to 362 categories; therefore, there is one category for every position on the board and one category for pass move action. Every tested model has 64 image inputs in one batch.

Models three and four have one special *batch normalization* layer after every convolutional layer. Batch normalization modifies all values of data flow to the

same scale. So, big values are shrunk to common scale with defined bounds. The final effect of this layer is to decrease overfitting of the model.

The next chapters describe and evaluate four of the most promising tested models. Table 1 shows basic information about the tested models.

Model name	Number of convolutional layers	Number of filters
Model 1	5, leaky ReLU	50
Model 2	5, ReLU	64
Model 3	5, leaky ReLU + BatchNorm	64
Model 4	5, leaky ReLU + BatchNorm	64

Table 1. Basic information about tested models

If too high learning rate is used, then the error of training process starts to increase. The error becomes unstable and the network is not learning at this state. However, by experimental trying it was discovered it is very appropriate to set the basic learning rate very near below this unstable limit. Thus, it is necessary to find the limit first where learning rate starts to be too high and the error becomes unstable. The basic learning rate of the training process was set to the first lower number within the same number of decimal numbers.

Decreasing of learning rate is very important to while training process. It should be decreased, when the error of training process has stacked for longer time and it is not decreasing anymore. Table 2 shows information about training process of tested models.

Model name	Used dataset	Learning iterations (in millions)	Min. learning error	Avg. error in last 5% of learning	Basic learning rate
Model 1	Dataset 2	10	1.2	1.9	0.00008
Model 2	Dataset 1	13	0.2	0.6	0.00006
Model 3	Dataset 3	7.2	0.5	0.9	0.0002
Model 4	Dataset 4	5.8	0.8	1.4	0.0002

Table 2. Information about training process of tested models

Model two was trained with three breaks, where the learning rate was changed and the training process was continued. In the error diagram of model two (Figure 21) there are very strong fluctuations in iterations, where the training

was stopped: iterations 4 000 000, 6 000 000 a 10 000 000. Other fluctuations were caused by automatic changes of learning rate - iterations 8 000 000 a 12 000 000.

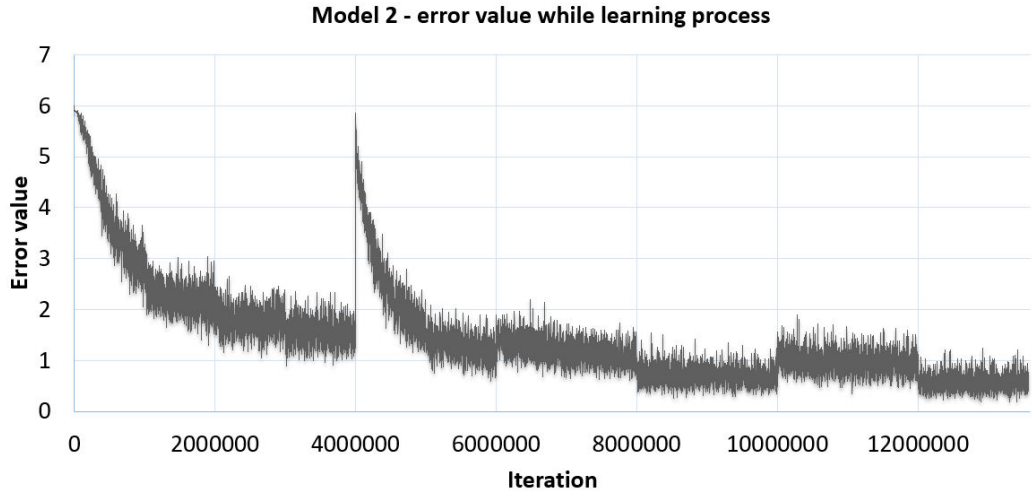

Figure 21. Model 2 - training process error diagram

7.3 Metrics

Metrics are used for evaluation of how well the network is trained and prepared for a given task. Basic and often used metrics for neural networks are probability: how often is the network output same with the expected right output? Thus, if there is network trained to recognize images of dogs and cats, the network is trained very well when it outputs the right animal for all input images.

The testing of trained networks shows that these metrics are not appropriate for the prediction of the next move task. The problem is in Go game itself. For every position of stones on the board, more than just one appropriate move exists. It is not true that the expected right move is the only good move the player can do. However, for this basic metrics, just one right move exists. Thus, bad results of this metrics do not mean network is trained improperly.

Nevertheless, this concept is still possible to use for network testing, when special data is used for testing. The first metrics test how well the network had trained data from the training dataset. A low percentage number for this metrics means that the network was not able to learn the given task. There are two possibilities for this result: the network did not have enough time to train

(training process was stopped too early) or the network had bad model architecture and it did not have not enough capacity to train the given task.

The second metrics are based on Go game's last moves in the match. The match ends, when both players play pass move in a sequence or one player resigns. Thus, both players want to finish the match; even the one going to lose. It is supposed that the winner player's last moves were right enough to change the mind of the opposite player to finish the match. Thus, there were probably not too many other good moves, which would have had the same effect. Good trained network should be able to find these appropriate moves at the end of matches.

A testing dataset was created containing input data and the expected right moves just from last two moves of matches. This dataset consists of more than 17 thousand image inputs. Two different approaches were tested. The first one compared the first network output to the expected right move. The second one compared the first *valid* network output to the expected right move. Both approaches were tried because not all network outputs are strictly valid.

7.4 Evaluation of trained models

Some networks were trained for a longer time (higher number of iterations), so there are more snapshots from the training process as well. Thus, a single training process has evaluated more snapshots. It is possible to see the progress of networks while training process. Results of evaluated networks are illustrated in Table 3.

Used model	Snapshot iteration	Metrics 1 (%)	Metrics 2 (%)	
			First move	First valid move
Model 1	8 000 000	52	4.8	5.1
	10 000 000	55	4.5	4.8
Model 2	4 000 000	62	0.6	0.9
	8 000 000	80	2.3	2.9
	13 200 000	83	2.3	2.8
Model 3	4 000 000	48	4.7	5.0
	7 200 000	81	4.5	4.8
Model 4	4 000 000	62	6.9	7.1
	5 600 000	64	7.0	7.2

Table 3. Metrics results of evaluated networks

The results point to the fact that networks were not trained enough to completely solve the given task. However, the results show the potential of convolutional neural networks to solve this task much better. The results of metrics one for all models prove that all evaluated networks were able to train the given task. It is possible to argue that models one and four had much worse results in metrics one. However, these models have one of the better results of metrics two. Moreover, datasets used for the training of models two and four were much bigger than other datasets. Thus, it is very possible that models one and four did not have enough time to feed all information from datasets. That is also a sign how crucial is to have dataset with appropriate size.

The results of metrics two for models one, two and three show that earlier snapshots (the middle of training process) have better results than the snapshots from the end of the training process. It means that these networks are slightly overfitted; the training process was very long. Models one and two did not use some special layer to reduce overfitting effect. However, in model three, batch normalization layer was used, which should partly reduce overfitting. It would be appropriate to also use some other ways to reduce overfitting effect, e.g. use layers similar to batch normalization layer (dropout layer) and a bigger dataset and stop the training process before overfitting occurs.

Model four has the best result of metrics two, where the overfitting effect did not appear yet. An important difference between this model and others is the dataset. It contains image data normalized to 0 and 1. Other datasets have image data normalized to 0 and 255. It means that a stone placed on the board in the image normalized to 0 and 1 has value 1 (in player's channel of image) and a board position without stone has value 0. This result proves that normalization to 0 and 1 is more effective for network training.

On the internet there was no free available training bot for Go, which would be appropriate for a beginner player. Thus, for testing an advanced bot was used (Clark 2018.), which has rank 7 kyu; it is equivalent to an intermediate player. This bot was not integrated in this program. Testing was just manual to check the real skills of the trained networks.

The results from testing of trained networks against bot show that trained networks can play basic moves and structures of Go. For example, in the beginning of the Go match some special moves are usually played, which represent a strategic advantage in the game played later. These opening moves are oriented to corners and sides of the Go board. Positions in the middle of the board are usually placed later. These opening moves were played by the trained networks in the right way. To compare, on the left side of Figure 22 there are the opening moves played by one of the trained networks (black stones), on the right side are the opening moves played by two professional players.

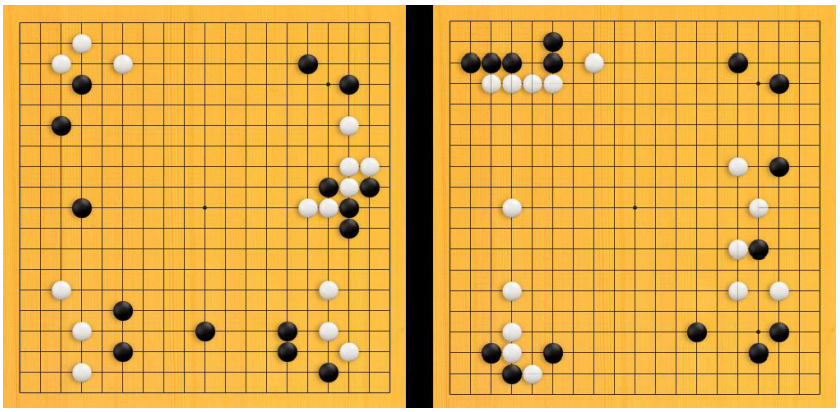

Figure 22. Opening moves by trained network (left), by professional players (right)

In the rest of match, the trained networks were able to create strings of stones and react to the opponent bot's moves; however, not always good moves were played. An example is shown in Figure 23, where on the left side is the position of stones on the board, where the trained network has black stones and the bot has white stones. The last move played by the trained network is marked by a red circle. The bot's natural move is to play on the position marked by a blue circle, which removed the stone marked by a red circle on the right side of the figure. These inappropriate moves ruined all games played against the bot (Clark 2018.).

The next research of this field should avoid fault, which causes problems with network training. The main problem, why networks were not trained properly is probably a too small dataset. Smaller datasets were used to make the training process faster. It is necessary to use much more powerful computer

with more graphics cards to train a network with a dataset of appropriate size. The used graphics card was insufficient for a task of this size and complexity.

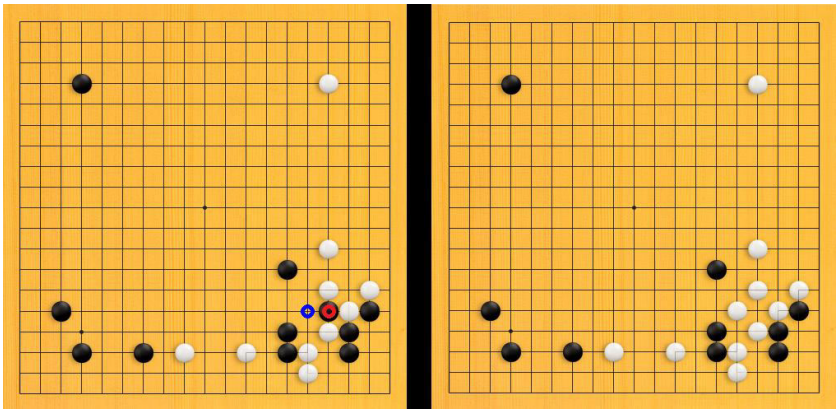

Figure 23. On the left side, the inappropriate move marked by red circle, on the right side the effect of this move

The results of evaluated models show it is necessary to use a bigger dataset, which can provide more information about Go game stone structures for the network while training. Moreover, network training would be probably more effective if datasets contain just processed data inputs without augmentation. Data augmentation did not bring the expected improvement. The dataset should contain image data normalized to 0 and 1. This way of normalization provides better results than the classical normalization to 0 and 255.

It is very important to set the right learning rate and decrease it while in the training process. If there is no expected improvement while the training process, it is still possible to stop it and continue again with a different configuration. It is necessary to use more methods to overfitting reduce, e.g. a batch normalization layer or dropout layer, regularization or stop training process in right time.

8 Conclusion

The thesis objective was to create Convolutional Neural Network (CNN) to predict the next appropriate move in the Go Game (such as Go game bot player). It is a non-trivial task. Its main point was to train the network to classify output data to 362 categories. Every category represented one position on the Go board or passing the move.

In this paper, SGF (Smart Game Format) files as data input were used. They are free for download from several Go game websites. The paper describes the process of data preparation, dataset creation from SGF records and the program used for this task. The resulting four types of datasets were used to train 20 networks with different model architectures. Four of the trained networks were evaluated in more detail in the paper. Unfortunately, not one of them got the expected results.

Despite this, the paper presents successful usage of the CNN; the experiments hardly lacked the hardware power to make machine learning extensive enough to achieve significant results. The used hardware with limited power allowed providing CNN with only a small portion of the dataset available. On the other hand, a configuration leading to quicker learning was found. The experiments also showed that input data normalization to 0 and 1 speeds up the computation and provides better results.

With a higher power machine, the same software should be able to process much more input data in reasonable time; thus giving significant accuracy. This expectation is at least based on the experiments with small dataset portion used.

Nevertheless, the paper has no impact on practical life, actual economy or industry, however, it has educational value: to employ CNN one needs to have either a smaller problem or great hardware power to even try to solve it. It also shows the best practice to experiment on smaller problem scale to find appropriate CNN configuration and not waste development time and time of a high power computer too early.

This thesis was my first practical neural network project. I learned much in many fields. I now understand much better what the neural network is, how it works and what its potential for practical life is. I am more familiar with Caffe framework, which is actually one of the most used deep learning frameworks. I had to make all programs in C++ language, which I had not used so much before. Now the C++ language is not a problem for me anymore. Moreover, I used only Linux systems while working on the thesis, because it was easier to install all the necessary software there. I also used remote computer for the experiments, thus I had to use it over console only. This forced me to learn to administrate Linux systems.

In general, I have learned many new skills and the project has inspired me for my next career. I look forward to working with neural networks and AI in the future.

The thesis source codes and user manual are available at the following link: https://github.com/kOrenOs/Go_CNN_bot

References

Cano, J. 2018. The Japanese Rules of Go. Accessed on 13 May 2018. Retrieved from http://www.cs.cmu.edu/~wjh/go/rules/Japanese.html
(Cano 2018.)

Davies, J. 2018. The Chinese Rules of Go. Accessed on 13 May 2018. Retrieved from https://www.cs.cmu.edu/~wjh/go/rules/Chinese.html
(Davies 2018.)

Scoring. Accessed on 13 May 2018. Retrieved from https://senseis.xmp.net/?Scoring
(senseis.xmp.net 2018.)

Deepmind technologies limited. 2018. The story of AlphaGo so far. Accessed on 13 May 2018. Retrieved from https://deepmind.com/research/alphago/
(Deepmind technologies limited 2018.)

Clark, C. - Storkey, A. 2018. Teaching Deep Convolutional Neural Networks to Play Go. Accessed on 13 May 2018. Retrieved from https://arxiv.org/pdf/1412.3409.pdf
(Clark, & Storkey 2018.)

Huu, H. - Jihoon, L. - Keechu, J. 2018. Suggesting Moving Positions in Go - Game with Convolutional Neural Networks Trained Data. Accessed on 13 May 2018. Retrieved from http://www.sersc.org/journals/IJHIT/vol9_no4_2016/5.pdf
(Huu, Jihoon, & Keechu 2018.)

Karpathy, A. - Johnson, J. 2018. Neural Networks Part 1: Setting up the Architecture. Accessed on 13 May 2018. Retrieved from http://cs231n.github.io/neural-networks-1/
(Karpathy & Johnson 2018a.)

Scheau, C. 2018. Regularization in deep learning. Accessed on 13 May 2018. Retrieved from https://chatbotslife.com/regularization-in-deep-learning-f649a45d6e0
(Scheau 2018.)

Bourez, C. 2018. About loss functions, regularization and joint losses: multinomial logistic, cross entropy, square errors, euclidian, hinge, Crammer and Singer, one versus all, squared hinge, absolute value, infogain, L1 / L2 - Frobenius / L2,1 norms, connectionist temporal classification loss. Accessed on 13 May 2018. Retrieved from http://christopher5106.github.io/deep/learning/2016/09/16/about-loss-functions-multinomial-logistic-logarithm-cross-entropy-square-errors-euclidian-absolute-frobenius-hinge.html
(Bourez 2018.)

Nielsen, M. 2018. How the backpropagation algorithm works. Accessed on 13 May 2018. Retrieved from http://neuralnetworksanddeeplearning.com/chap2.html
(Nielsen 2018a.)

Nielsen, M. 2018. Using neural nets to recognize handwritten digits. Accessed on 13 May 2018. Retrieved from http://neuralnetworksanddeeplearning.com/chap1.html
(Nielsen 2018b.)

Karpathy, A. - Johnson, J. 2018. Neural Networks Part 3: Learning and Evaluation. Accessed on 13 May 2018. Retrieved from http://cs231n.github.io/neural-networks-3/
(Karpathy & Johnson 2018b.)

Karpathy, A. - Johnson, J. 2018. Neural Networks Part 3: Setting up the data and the model. Accessed on 13 May 2018. Retrieved from http://cs231n.github.io/neural-networks-2/
(Karpathy & Johnson 2018c.)

Karpathy, A. - Johnson, J. 2018. Convolutional Neural Networks (CNNs / ConvNets). Accessed on 13 May 2018. Retrieved from http://cs231n.github.io/convolutional-networks/
(Karpathy & Johnson 2018d.)

British Go Association. 2018. How to Play. Accessed on 13 May 2018. Retrieved from https://www.britgo.org/intro/intro2.html
(British Go Association 2018.)

Wikipedia. 2018. Rules of Go. Accessed on 13 May 2018. Retrieved from https://en.wikipedia.org/wiki/Rules_of_Go
(Wikipedia 2018.)

Stanek, M. 2018. Understanding AlphaGo. Accessed on 13 May 2018. Retrieved from https://machinelearnings.co/understanding-alphago-948607845bb1
(Stanek 2018.)

Burger, C. 2018. Google DeepMind's AlphaGo: How it works. Accessed on 13 May 2018. Retrieved from https://www.tastehit.com/blog/google-deepmind-alphago-how-it-works/
(Burger 2018.)

Neural Network Hyperparameters. Accessed on 13 May 2018. Retrieved from http://colinraffel.com/wiki/neural_network_hyperparameters
(colinraffel.com 2018.)

Shah, T. 2018. Accessed on 13 May 2018. Retrieved from https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7
(Shah 2018.)

Jia, Y. 2018. Caffe | Installation. Accessed on 13 May 2018. Retrieved from http://caffe.berkeleyvision.org/installation.html
(Jia 2018a.)

Xin, W. 2018. Ubuntu 16.04 or 15.10 Installation Guide. Accessed on 13 May 2018. Retrieved from https://github.com/BVLC/caffe/wiki/Ubuntu-16.04-or-15.10-Installation-Guide
(Xin 2018.)

Görtz, U. 2018. Game records. Accessed on 13 May 2018. Retrieved from https://u-go.net/gamerecords/
(Görtz 2018.)

Clark, C. 2018. Play Go Against a Deep Neural Network. Accessed on 13 May 2018. Retrieved from https://chrisc36.github.io/deep-go/
(Clark 2018.)

Santos, L. 2018. Convolution. Accessed on 13 May 2018. Retrieved from https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/convolution.html
(Santos 2018.)

Jia, Y. 2018. Caffe. Accessed on 13 May 2018. Retrieved from http://caffe.berkeleyvision.org/
(Jia 2018b.)

Jia, Y. 2018. Interfaces. Accessed on 13 May 2018. Retrieved from http://caffe.berkeleyvision.org/tutorial/interfaces.html
(Jia 2018c.)

Jia, Y. 2018. Caffe Model Zoo. Accessed on 13 May 2018. Retrieved from http://caffe.berkeleyvision.org/model_zoo.html
(Jia 2018d.)