DEGREE PROJECT IN INFORMATION AND COMMUNICATION TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, *SWEDEN 2017*

Simulating Human Game Play for Level Difficulty Estimation with Convolutional Neural Networks

PHILIPP EISEN

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

KTH Information and Communication Technology

Simulating Human Game Play for Level Difficulty Estimation with Convolutional Neural Networks

PHILIPP EISEN

Master's Thesis at ICT
Academic Supervisor: Pawel Herman
Industrial Supervisor: Stefan Freyr Gudmundsson
Examiner: Jim Dowling

TRITA-ICT-EX-2017:149

Abstract

This thesis presents an approach to predict the difficulty of levels in a game by simulating game play following a policy learned from human game play. Using state-action pairs tracked from players of the game Candy Crush Saga, we train a Convolutional Neural Network to predict an action given a game state. The trained model then acts as a policy.

Our goal is to predict the success rate (SR) of players, from the SR obtained by simulating game play. Previous state-of-the-art was using Monte Carlo tree search (MCTS) or handcrafted heuristics for game play simulation. We benchmark our suggested approach against one using MCTS. The hypothesis is that, using our suggested approach, predicting the players' SR from the SR obtained through the simulation, leads to better estimations of the players' SR.

Our results show that we could not only significantly improve the predictions of the players' SR, but also decrease the time for game play simulation by at least 50 times.

Referat

Simulering av mänskligt spelande för bedömning av spelbanors svårighetsgrad med Convolutional Neural Networks.

Den här avhandlingen presenterar ett tillvägagångssätt för att förutse svårighetsgrad av spelbanor genom spelsimulering enligt en strategi lärd från mänskligt spelande. Med användning av tillstånd-handlings par insamlade från spelare av spelet Candy Crush Saga, tränar vi ett Convolutional Neural Network att förutse en handling från ett givet tillstånd. Den tränade modellen agerar sedan som strategi. Vårt mål är att förutse success rate (SR) av spelare, från SR erhållen från spelsimulering. Tidigare state-of-the-art använde Monte Carlo tree search (MCTS) eller handgjorda heuristiker för spelsimulering. Vi jämför vårt tillvägagångssätt med MCTS. Hypotesen är att vårt föreslagna tillvägagångssätt leder till bättre förutsägelser av mänsklig SR från SR erhållen från spelsimulering. Våra resultat visar att vi inte bara signifikant kunde förbättra förutsägelserna av mänsklig SR utan också kunde minska tidsåtgången för spelsimulering med åtminstone en faktor 50.

Acknowledgments

Firstly, I would like to thank my academic supervisor, Pawel Herman, at the school of Computer Science and Communication at the Royal Institute of Technology (KTH) for providing me with guidance and support throughout the journey of writing this thesis. I would like to express my sincere gratitude to my industrial supervisor Stefan Freyr Gudmundsson at King.com for his continuous support, advice, helpful discussions and feedback. His vast knowledge and motivation helped me to overcome the obstacles I faced while writing this thesis. I want to thank my friends and colleagues in the Artificial Intelligence team at King.com, Erik Poromaa, Alex Nodet, John Pertoft, Sami Puromenen, and Lele Cao, for supporting me when facing problems and provided me with helpful feedback. I want to direct a special "thank you" to Erik Poromaa, Alex Nodet, and Sami Purmonen, who set the foundations, without which this project would not have been possible. I would like to express my gratitude to Bartlomiej Kozakowski at King.com for providing me with advice and recommendations for statistical questions as well as giving me feedback on my thesis. Further, I want to thank my friends and fellow students at KTH, Guilherme Dinis Jr. and Filip Stojanovski for the fruitful discussions and continuous feedback on my work. I want to thank Prof. Jim Dowling at KTH's school of Information and Communication Technology for taking on the job of examining my thesis.

Finally, I want to thank my family for supporting me in reaching this milestone of my career.

Stockholm, September 7, 2017
Philipp Eisen

Contents

Acronyms

AI Artificial Intelligence.

CI confidence interval.

CNN Convolutional Neural Network.

ELU exponential linear unit.

FDS filtered data set.

GAP global average pooling.

ILSVRC ImageNet Large Scale Visual Recognition Challenge.

MAE mean absolute error.

MCTS Monte Carlo tree search.

MLP multi-layer perceptron.

NN artificial neural network.

np-hard non-deterministic polynomial-time hard.

ReLU rectified linear unit.

ResNet Residual Networks.

RMSE root-mean-squared error.

SGD stochastic gradient descent.

SR success rate.

UCB upper confidence bound.

UDS unfiltered data set.

VGG Visual Geometry Group.

Chapter 1

Introduction

1.1 Problem

Within the recent years, game developers have increasingly adopted a free-to-play business model for their games. This is especially true for mobile games (see e.g. [1, 2, 3, 4, 5]). The free-to-play business model refers to a business model where the core game is available free of charge, and revenue is created through the sales of additional products and services such as additional content or in-game items [2]. As such, users do not need to be convinced to spend money when first obtaining the game, but rather over the course of playing the game. Therefore, game producers more and more tend to continuously add content to the game to keep their users engaged and to be able to continuously monetize on a game title [2]. For this to work out, it is important that the new content lives up to the quality expectations of the players [1].

The difficulty of a game has a considerable impact on a user's perceived quality [1]. If a game is too easy, the player gets bored, if it is too hard, the player gets frustrated. In both cases the game is considered of low quality. In trying to create the desired experience with regards to the difficulty, game designers estimate the players' skill and set game parameters accordingly. Inspired to some extent by web analytics, mobile game companies usually have sophisticated tracking techniques in place to monitor how users interact with their games [6]. This way, measures that reflect the perceived difficulty of the game can be monitored once content has been released to players.

However, if new content would be released directly to players of the game, those would potentially be exposed to content that does not live up to their quality expectations and might abandon the game as a consequence. Therefore, game designers usually let new content be play tested and tune the parameters in an iterative manner based on data obtained from those tests before releasing the new content to players [6, 7]. Play testing can be carried out by human test players that are given access to the new content before it is released. However, human play testing comes at the disadvantages of introducing latency and costs in the development process. Game designers need to wait for the results from the test players before they can continue with the next iteration of their development process and human testers need to be hired [8]. Additionally, results

from test players might not lead to appropriate conclusions about the general player population as the populations' skill levels can differ.

In an attempt to tackle these disadvantages several approaches for automatic play testing have been proposed [8, 9, 10, 11]. Isaksen et al. [9] simulate playing levels using a simple heuristic and then analyze the level design using survival analysis. Zook et al. [8] use Active Learning to automatically tune game parameters to achieve a target value in human player performance. Poromaa [10], similarly to Isaksen, proposes an approach, where the play test is carried out using a Monte Carlo tree search (MCTS) algorithm to simulate game play. Silva et al. [11] evaluate a competitive board game by letting general (MCTS and A*) and custom Artificial Intelligence (AI) agents play against each other.

The methods above, however, do not consider data that can be gathered from content that has been released earlier, when simulating game play. We hypothesize, that taking this data into account could lead to a game play simulation closer to human play, and therefore to better estimates of the difficulty of new content. More specifically, we built a prediction model that predicts moves from a given game state. This model is trained on moves that were executed by players on the previously released content. Once trained, the model acts as a policy, suggesting which move to execute given a game state, for an agent simulating game play. The state-of-the-art methods for predicting a move from a given state are based on CNNs [12, 13, 14]. CNNs are a specific type of artificial neural networks (NNs) that are very well suited for data that comes in a grid-like structure [15]. Since the data of the problem at hand has a grid-like structure and is similar to data used in stat-of-the-art research, CNNs appear to be the most promising approach for the task at hand. Therefore, we are going to rely on this approach for this thesis.

1.2 Research Question

Currently, there exists an implementation of a Monte Carlo tree search based agent to simulate playing different levels of the game. The metrics produced by this agent are then used to estimate the difficulty of a level. Since this is an approach that has delivered the best results to date within the given domain of computer games [10], it will be used as a baseline for the approach suggested in this thesis. The measure for the difficulty used within this thesis is the success rate (SR). Consequently, the research question to be examined shall be:

Can the players' SR estimation be improved by using a CNN-based instead of a MCTS-based agent for game play simulation in the game Candy Crush Saga?

1.3 Scope

This research is conducted in a game domain that can be classified as non-deterministic puzzle game. In this type of game domain, the players' objective is not to beat an opponent, but instead to find a solution to a problem adhering to a set of rules. Further,

the game domain features levels. Each level is an instance of the puzzle game that is subject to the same rules, but differs in other parameters. The objectives of the levels may vary, but are not arbitrary (i.e. there exists a set of objectives that does not change). For example, in one level the objective may be to remove a certain number of elements from the game board, while in another level the objective could be to remove some particular elements from the board within a move limit. Instead of being limited by moves, the objectives can also be limited by time. Levels that are limited by time, however, are out of the scope of this research. The objectives of a level are observable by the player. For all practical purposes, the game can be considered as having the Markov property. There are some levels, where some specific features appear that make the game lose the Markov property. However, due to the rare appearance of those edge cases, they will not be considered in this research. In particular, the game used within the scope of this research is Candy Crush Saga, developed by King.com and released in 2012. Candy Crush Saga problems are non-deterministic polynomial-time hard (np-hard) to play and thus finding a solution is a computationally hard problem to solve [16]. Levels in this game family are defined by the initial state and the objective to be reached.

1.4 Ethical Considerations

This research can be seen as an application of Artificial Intelligence technology. Currently, the impact of AI on society and economy is widely discussed in public media. A lot of the discussion is centered around the issue that AI could potentially make jobs obsolete and therefore cause negative effects on society. The results of this research can make part of the play testing carried out by humans obsolete. In particular, play testing to gauge the difficulty and balance of a game can be affected when the suggested approach is implemented. Therefore, jobs that consist of carrying out this kind of play testing can be endangered. However, we argue that the jobs that previously consisted solely in executing these play testing tasks could be transformed to be jobs, where AI technology augments the human expertise instead of replacing it. Gauging the difficulty or balance of a game is essentially about looking at quantitative measures such as the SR. We argue that AI technology in the domain of play testing is good in producing these quantitative measures. At the same time, it is hard to imagine how AI could make judgments about qualitative perceptions of a game such as aesthetics or general user experience. We can therefore see the job role of play testing evolving into focusing more on the qualitative perception of the game while leaving repetitive tasks to AI agents.

With regards to privacy concerns, it needs to be pointed out that we are tracking data from real players for this thesis. However, all data is anonymized in a way that no information can be traced back to the actual player.

1.5 Environmental Considerations

The approach suggested in this thesis for difficulty prediction involves the storage and processing of fairly large amounts of data. We therefore aim to represent data in a way

it consumes least possible storage while still being practical. In particular, training data is stored in protobuf files. In this file format, smaller integer numbers are represented by less bytes.

In addition to storage we also consume computation power. Training one version of the CNN model takes about 24h on a single machine with 6 CPUs and one Nvidia Tesla K80 GPU. During training the GPU is constantly at 100% utilization, consuming about 150 Watt. Simulating the game play is usually carried out on 32 CPUs in parallel. This number could be increased if faster evaluation is required. All computational resources are used on demand from a cloud service provider. This means, once an evaluation is done, the CPUs will not be used anymore. Therefore, the energy impact of running an evaluation is not dependent on the number of CPUs used for the evaluation as parallel execution scales linearly.

1.6 Outline

The rest of the content of this thesis is structured as follows: Chapter 2 introduces the background of the problem at hand, related work, as well as the background for the methodology used. Chapter 3 gives a detailed explanation of the method applied as well as the design of the experiment to test the research question. Chapter 4 presents the results obtained by executing the method. Chapter 5 discusses the results from different angles. Chapter 6 concludes the thesis by summarizing the findings and their implications.

Chapter 2

Background

2.1 Game Play Related Background

2.1.1 Game Parameters & Game Metrics

Within the context of this thesis, game parameters should be referred to as concepts or values that define a game. This includes among others the game rules, level layout, and game objective.

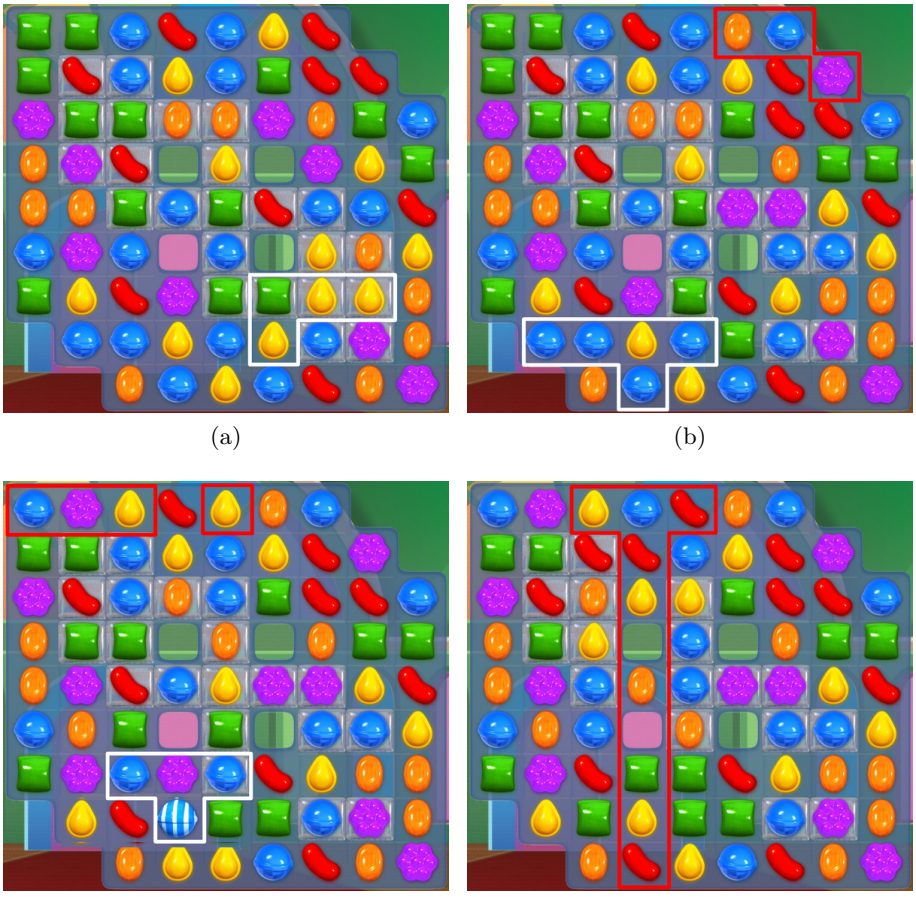
Game metrics should be understood as the measures that result from the interaction with the game. Game metrics include metrics on players' behavior and performance. As this thesis is aiming to estimate the difficulty of levels, that are defined by game parameters, a metric reflecting the difficulty of a level is of particular interest. The metric we used to gauge level difficulty is SR. The human SR for a level i and a considered group of players is calculated as follows:

$$SR_i = \frac{s_i}{a_i}, \tag{2.1}$$

where s_i is the number of successes of the considered player group for level i, and a_i is the number of attempts of the considered player group for a level i. An attempt is one try to solve a level. The attempt is counted as success if the objective of the level is reached within the limit defined for reaching that objective. As mentioned before the limit for the game at hand is always a certain number of moves. We chose this measure as measure for success, as it is easy to query from recordings of player data and reflects how a user perceives the difficulty of a level. If the user needs a lot of attempts to succeed in solving a level and thus the SR is low, they will perceive the level as a hard one. This measure is also commonly used among data scientists within King.com (source: personal communication).

2.1.2 Candy Crush Saga

The game Candy Crush Saga was released in 2012 by King.com. At the time of writing it is still the title generating most of King's revenues (data source: appannie.com). It is



(a) (b)

(c) (d)

Figure 2.1. Basic mechanics of the game Candy Crush Saga (images obtained with permission from Poromaa [10]): White outline indicates the candies involved in the action carried out. Red outline indicates the candies that appeared on the board as a result of the action in the previous state.

a puzzle game that consists of different game items and blockers. An action in the game is a vertical or horizontal swap of the positions of two adjacent game items. An action is legal when the swap results in a vertical or horizontal match of at least three items of the same color or when the two items being swapped are special candies. Special candies are created when a swap results in a match of more than three items. They can also already be on the game board when starting the level. When included in a match, special candies remove more items from the board than just the candies that are part of the match. The player can only execute legal actions. When an action is executed, the items that are part of a match are removed from the board. Removing items from the board earns the player points. Empty space on the board is filled by items of random color. The board is a 9x9 grid that can contain tiles that are not filled by any items. There exist six different game modes, that are all subject to the rules mentioned before: *score levels*, *jelly levels*, *ingredients levels*, *timed levels*, *candy order levels* and *mixed levels*. In *score levels*, the objective is to reach a certain score goal. *Jelly levels* require the player to remove candies at certain areas. Those areas are visualized as tiles covered by jelly in the form of a white highlight. In *ingredients levels* the goal is to move a number of special items, called ingredients, to a certain position on the game board. Those items cannot be part of a match, but can be part of a swap (i.e. they are only removed from the board when they are moved to a certain position - not by including them in a match). *Timed levels* challenge the player to reach a certain points goal within a certain time limit. For *candy order levels*, the objective is to remove a certain number of certain items. A *mixed level* consists of a combination of the previously mentioned game goals. The limiting factor for all of these game modes except for the *timed levels* is the number of actions. On average, there are about 10.28 legal actions per state. However, the number of legal actions can vary significantly. The limit of actions in which a player has to fulfill the objective can be seen as the depth of a level. The average depth of a level is 32. This makes for a game search space of about 10^{32} on average.

Figure 2.1 shows four exemplary, successive states of a *jelly level*. The white outlines indicate the candies involved in a move. The red outlines indicate the candies that appeared on the board as a result of the move. In figure 2.1a a regular three match is selected. By swapping the green and the yellow candy, there are three yellow candies in a row. Those candies then disappear from the board. The resulting void is filled by candies moving down the column. The top of the affected columns is filled with candies of random color (see red outline in figure 2.1b). The white highlight behind the candies involved in that match is the so-called jelly. By removing candies over the jelly, the jelly is removed (see how the white highlight behind the yellow candies is not existing in figure 2.1b anymore). In figure 2.1b a move resulting in a four match is executed. This creates a special, striped candy (see figure 2.1c). In figure 2.1c a move including the striped candy is executed. Including this candy clears the whole column resulting in new candies in that column and a removal of all the jellies (see figure 2.1d). In Figure 2.2 we show four different levels to illustrate the diversity between different levels.

(a) (b) (c) (d)

Figure 2.2. Four different levels in Candy Crush Saga. Demonstrating the diversity in levels.

2.2 Related Work

2.2.1 Automatic Play Testing

Automatic play testing in general refers to the effort of automating parts of the iterative process of adjusting game parameters and testing them by simulating game play with these parameters. The suggested approaches for carrying out play tests differ in the methods used as well as in the aspects of a game the play test should help to analyze.

Silva et al. [11] used different AI agents to collect data about the gameplay of a board game. This data helped discover loop-holes in the game. The AI agents applied were, besides the general agents MCTS and A*, two agents particularly designed for the game in question. The authors found that their game-specific agents outperformed both MCTS and A* agents. They suggest that the general search algorithms do not perform well on the problem of the game at hand as it has a big search space and it is hard to evaluate the value of a particular game state. These properties also apply to the game that is subject to this research.

Isaksen et al. [9] suggest a framework for creating levels of different difficulties based on the game parameters. In their work, a set of game parameters is selected by a game designer. From these game parameters, levels are generated and game play is simulated for a certain number of attempts using a simple heuristic. The scores achieved using this simulation are recorded and analyzed using the technique of survival analysis. This produces a measure of difficulty between 0 (trivial) and 1 (impossible). Using the outcome of this analysis, the game parameters can be adjusted until a desired level of difficulty is reached. The parameters that achieve the desired level of difficulty can then be sampled for visualization and checked for uniqueness in game play.

Khalifa et al. [17] suggest a modification of MCTS for more human-like game playing. In their approach, a term is added to the upper confidence bound formula to bias it according to observed human selections. In the game domain they are researching, this means being more likely to choose the same action as in the frame before. In addition, they add a bonus for map-exploration to the upper confidence bound formula. Further, they change the expected reward from being the average of the observed values to being

a mix between the average and the maximum value of a child node (MixMax) as this yielded the best results.

Poromaa [10] presents a method of evaluating the difficulty of the game Candy Crush Saga using MCTS. The MCTS algorithm suggested considers partial objective fulfillment when a roll-out does not lead to a win, instead of just binary values (win or loss) back-propagated from the leaf node of the roll out. Considering partial objective fulfillment improves the performance of the AI agent as the depth of the search tree only allows for a certain number of simulations per move to be feasible and without the partial goal objective fulfillment term the signal would be too weak.

To the best of our knowledge none of the related work in the field of automatic play testing is simulating human game play by directly training a policy on moves made by human players.

2.2.2 Move Prediction Using Supervised Learning

Most of the research of move prediction using supervised learning was conducted for the game of Go. This is likely due to the fact that this game, until recently, had been one of the last board games, where computers did not reach super-human performance. Compared to other board games such as Chess or Backgammon, the search space in Go is extremely big. This makes tree-search methods, that are commonly applied for in other games, less successful.

Coulom [18] proposed an approach, where different move patterns were rated using an algorithm based on the Bradley-Terry model. The Bradley-Terry model forms the theoretical basis for the Elo-rating system used in the game of Chess. So in its essence the approach by Coulom [18] learns Elo-ratings of specific move patterns. This approach achieved an accuracy of 34.9% on a data set obtained from games played on the KGS Go server.

More recently, CNNs have been used as a supervised learning method to predict expert moves in the game of Go. Clark and Storkey [13] published a paper reporting an approach that achieved a 44.4% prediction accuracy on the KGS data set. Shao et al. [14] report a move prediction accuracy of about 42% on a data set from the same source. Silver et al. [12] (AlphaGo) managed to improve the prediction accuracy to 55.7%, beating the previously best accuracy by about 10%. They report that this increase in accuracy led to a very significant increase in the strength of their Go playing agent. In fact, they report that using the expert move predictions of the neural network without any look-ahead search was able to perform on the same level as state-of-the-art MCTS programs. Predicting expert moves using supervised learning can be considered as one of the major success factors of Google DeepMind's AlphaGo system that gained wide public attention by beating Lee Sedol, one of the world's best Go players, in the game of Go [12].

Runarsson and Lucas [19] propose using preference learning for move prediction in the game *Othello*. In their paper, they discuss the possibility to directly approximate a policy using binary classification (equivalent to the policy network used in [12]) versus using differential preference learning. Differential preference learning forms N-tuples

between the selected and all not selected moves. The learner should then favor the selected move in all tuples. In their work, they found that the differential preference learning lead to better results than direct classification. However, the model used for direct classification was a linear model and thus has low capacity and can be considered very limited in its predictive capabilities.

State-of-the-art research for move prediction using supervised learning shows that CNNs outperform other current methods in the game of Go. The problems of predicting moves in Go and Candy Crush Saga have some parallels such as the grid-like board structure and the large search-space. However, there also exist some important differences. To start with, Candy Crush Saga is a single player game. Further, state transitions are not deterministic and the layout of the board as well as the appearance of items varies from level to level. Still, we argue that CNN is the method with the highest prospect of producing good results for the game of Candy Crush Saga and therefore relied on this approach for this work.

2.3 Methodology Background

2.3.1 Artificial Neural Networks

The most quintessential type of NNs are often called feedforward neural networks or multi-layer perceptrons (MLPs). Their goal is to approximate some function f^*. When being used as a classifier, for example, the MLP tries to approximate the function $y = f^*(x)$ mapping an input x to a class y. It does so by defining a mapping $y = f(x; \theta)$ and learning the parameters θ that best approximate f^*.

MLPs or feedforward neural networks can be, for simplification, thought of as a composition of different functions. For instance, we can look at a MLP with two layers as a chain of two functions $f^{(1)}$ and $f^{(2)}$, that chained together form $f(x) = f^{(2)}(f^{(1)}(x))$. In this example $f^{(1)}$ and $f^{(2)}$ are respectively the first and the second layer of the MLP. The number of layers is often called *depth*. When referring to models with a lot of layers, therefore, the term *deep neural networks* or *deep learning* is commonly used. The chain of computations is typically represented as a graph of computations, making the NN a *network*. The term *neural* stems from a loose inspiration from neuroscience, where biological cells forming a complex network are investigated [20]. NNs as the MLP, in which the information only flows in one direction, from input to output, are called *feedforward* NNs. In contrast, in a recurrent NNs there exist feedback connections [21, 20]. Recurrent neural networks, however, are out of the scope of this thesis.

2.3.2 Convolutional Neural Networks

Convolutional Neural Networks are a specific type of feedforward NN, that are particularly well suited for problems with data, that comes in the form of multiple arrays [15]. A grey-scale image, for instance, is a 2-dimensional array of scalars, each of them representing one pixel. They obtain their name from the convolutional layers that are part of their architecture. In a convolutional layer a, usually squared or cubed, recep-

tive field with weight vectors (also referred to as filter or kernel) is moved step by step over a 2 or 3-dimensional array. At every step the sum of the product of weights and corresponding values of the 2 or 3-dimensional array is formed (a scalar). The result is then typically passed through a non-linear activation function. Striding over the 2 or 3-dimensional array leads to a new 2-dimensional array commonly called activation map. Per convolutional layer there are typically several filters, each of which produce an activation map. Those 2-dimensional activation maps are then stacked, leading to a 3-dimensional array. This output of a convolutional layer can then be used as the input of a subsequent convolutional layer. This way, higher level features can be learned [15, 21].

The idea of Convolutional Neural Networks already appeared around the 1980s [22, 23, 24]. However, it was not until 2012, when Krizhevsky et al. [25] won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [26] with their network "AlexNet" until they gained substantial public interest. Since then the architecture proposed by Krizhevsky found successful applications in fields such as object detection [27], semantic segmentation [28] and super-resolution [29]. Supported by the attention, CNNs with new architectures that show significant prediction accuracy improvements over "AlexNet" appeared. At the time of writing, among the most prominent architectures was the Visual Geometry Group (VGG) network, the inception network, and the residual network. The VGG network was proposed by Simonyan and Zisserman [30] of the VGG of the University of Oxford. It is widely used due to its strong performance and rather simple network architecture. Additionally, the publicly available trained versions of the network allow to be transferred to other problems (transfer learning). The inception network [31] consists of several inception modules that can be seen as mini-networks within the CNN. This allows for a significant reduction in parameter usage and computational resources while maintaining state-of-the-art performance. The Residual Networks (ResNet) architecture was proposed by He et al. [32] and suggests an approach that helps to improve learning neural networks with more layers. The major difference to other CNN architectures is the bypassing of some convolutional layers for better gradient backpropagation [32].

Network Architectural Components

This section presents the network architectural components CNNs that are relevant to this work.

Activation Function: In order to create non-linear decision boundaries, non-linearity has to be introduced into models. This is done using activation functions. In the CNNs mentioned above the activation function used is the rectified linear unit (ReLU): $f(x) = max(0, x)$. Compared to a sigmoidal activation function such as the sigmoidal logistic function $f(x) = \frac{1}{1+e^{-x}}$ or the sigmoidal hyperbolic tangent $f(x) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$, ReLU has the main advantage of avoiding the vanishing gradient problem [33], as the derivative for positive values is always 1 and does not approach 0. In addition, ReLU activation functions create sparse codes as a deactivated state is represented by 0. However, since

ReLUs are non-negative, the mean activations of layers with ReLU activation functions is larger than zero. Therefore, layers with ReLU units or with other activation functions that produce non-zero mean activations act as a bias for the next layer. During learning this can lead to a bias shift. One way of addressing this problem is batch normalization. Batch normalization centers activations around zero with the goal of countering the "internal covariate shift" and reducing the bias shift [34]. The "internal covariate shift" is defined as the "change in the distribution of network activations due to the change in network parameters during training" [34]. Recently, the exponential linear unit (ELU) activation function has been introduced [35]:

$$f(x) = \begin{cases} x, & \text{if } x > 1 \\ \alpha(e^x - 1), & \text{if } x \leq 1 \end{cases} \tag{2.2}$$

The negative deactivation state allows for the bias shift to be reduced. In experiments presented by the authors the networks using a ELU activation function showed faster learning and better generalization capabilities than networks using ReLU activation functions.

Softmax: The Softmax function transforms the elements of a vector of size K to values in the range $(0, 1)$ that add up to 1. The function is defined as follows:

$$softmax(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{ for } j = 1, ..., K, \tag{2.3}$$

where $\mathbf{z}$ is a vector of arbitrary real numbers, and K is the number of elements in the vector. In applying this transformation to the vector of logits, where every element in the vector corresponds to one of the possible class labels, the softmax function creates a probability distribution over the class labels.

Global Average Pooling: Historically, Convolutional Neural Networks vectorized the output of the last convolutional layer and then fed it to standard fully-connected layers (see e.g. [25]). This way the convolutional layers of a network act as feature extractors that feed their extracted features to a fully-connected classifier. Fully-connected layers, however, are prone to overfitting. Therefore, several regularization methods such as dropout [36] and weight-decay have been proposed and demonstrated better generalization performance.

Lin et al. [37] propose another way of creating outputs from a Convolutional Neural Network called global average pooling. In this approach the last convolutional layer produces as many activation maps as there are output labels. For each of these activation maps a global average is formed and directly fed into a softmax layer. This method is claimed to be more native to the convolutional structure of a network as it forces the feature maps to be corresponding with the output labels [37]. The output feature planes can thus be interpreted as label confidence maps. Replacing the fully-connected layer with global average pooling removes trainable variables in this layer and thus the

possibility to overfit in this layer. Note, that the network could still overfit in the other layers.

Optimizers: Training neural networks is done by optimizing a parametrized objective function, usually minimizing a cost function, with respect to its parameters. The parameters in a neural network are often referred to as weights. If the function is differentiable with respect to its parameters, gradient descent can be used as optimization method. Gradient descent has a relatively low computational complexity. Computing the first-order partial derivatives with respect to the parameters has the same computational complexity as evaluating the function. Further, in neural networks the cost function is usually composed by computing the average of the cost per training example. This makes it possible to take gradient steps with respect to a subset of the training examples, i.e. stochastic gradient descent (SGD). In larger data sets it is especially inefficient or even not feasible within hardware constraints to take gradient steps with respect to all training examples. Therefore, SGD was commonly used for training neural networks (e.g. [38, 25, 39]). One issue with gradient based learning is, however, that the magnitude of the gradients can vary significantly and thus the steps taken can be either too small or too large. When computing the gradients for the whole data set this issue can be mitigated by only considering the sign of the gradient and not its magnitude as it is done in the RPROP algorithm [40]. However, this violates the idea behind stochastic gradient descent, that successive gradients with respect to a weight average each other out. To achieve the advantages of RPROP while maintaining the effect of averaging of successive gradients, RMSProp was proposed [41]. RMSProp keeps a moving average of the squared gradient for each weight and divides the gradient by the square root of that moving average. Another shortcoming of SGD is that weights for infrequently appearing features get update less frequent, even when they can be very predictive features. To address this issue the AdaGrad algorithm was proposed [42]. AdaGrad reduces the step size for frequently occurring features and increases the same for rarely occurring features (i.e. features that are non-zero only on a few data points). The ADAM algorithm combines the advantages of both RMSProp and AdaGrad [43]. It computes the first and second moments of gradients for each parameter and uses these to adjust the step size per parameter individually.

Cost function: The goal of classification models is to approximate the probability distribution for a training instance to be of a specific class. Note that within the scope of this thesis classes are mutually exclusive and therefore an example can only belong to one class. Therefore, the actual probability distribution for a particular example is 0 for all classes that it does not belong to and 1 for the one class that the instance belongs to. As discussed before, the softmax output of the neural network produces the predicted probability distribution for a given instance. To quantify the difference between the predicted and the actual probability distribution of an instance, cross-

entropy is commonly used (e.g. [40, 39, 31, 35]). Cross entropy is defined as:

$$H(p_i, q_i) = -\sum_k p_{i,k} * log q_{i,k}, \tag{2.4}$$

where $p_{i,k}$ is the actual probability for the instance i to be of class k ($p_{i,k} \in \{0, 1\}$) and $q_{i,k}$ is the predicted probability of the instance to be of class k ($q_{i,k} \in (0, 1)$).

2.3.3 Monte Carlo Tree Search

MCTS is not the method used for gameplay simulation in the approach suggested in this work. However, since MCTS serves as a benchmark for the suggested approach, this section summarizing the algorithm is added for the reader's convenience.

At any given state s there exists a set of possible actions A to choose from. Choosing an action $a \in A$ leads to a new state s' with possible actions A'. For choosing an action to take from A MCTS performs a number of simulations. By performing the simulations, MCTS incrementally builds a partial game tree [44]. Each simulation consists of four steps:

1. **Selection:** During the selection phase the current partial game tree is traversed according to a selection strategy. Common selection strategies are ϵ-greedy and upper confidence bound (UCB) [45].

2. **Expansion:** The selection strategy is applied until a leaf node of the partial game tree is reached. A leaf node of the partial game tree means that none of the possible actions of that node have been visited so far. Then, one or more leaves are added to the partial game tree.

3. **Simulation:** Of the newly created leaf nodes one is selected according to a selection strategy. From that leaf node, the game is rolled out following a rollout policy. Often this rollout policy is just the random policy, but it can be replaced by more sophisticated policies.

4. **Backpropagation:** The result obtained from the simulation is then backpropagated to update or initialize the values of actions traversed in that MCTS iteration. For the states visited during the rollout, no statistics are maintained.

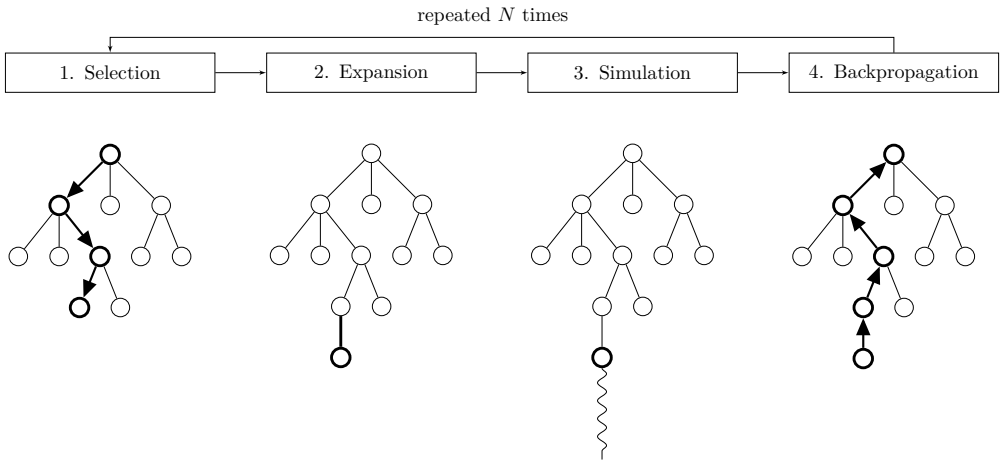


Figure 2.3. Monte Carlo tree search (adapted from [44])

Chapter 3

Method

The approach we used for level difficulty prediction consists of two major stages:

1. **Game play simulation.** In the first stage, game play of several levels is simulated using artificial agents to select moves. This game play simulation results in game metrics. In this thesis, the metric we are interested in is the SR.
2. **Prediction of players' SR.** The SR-measures created through game play simulation then have to be set into a relation with the actual players' SR-measure for each level.

The novelty in the suggested approach lies in stage 1. In particular, the difference from existing approaches lies in the artificial agent. Existing approaches use agents that do not make use of actual player data. In the following, the two stages of the method are described in detail. Then, we describe the design of the experiment used to evaluate the research question. Finally, we briefly explain the evaluation measures used.

3.1 Game play Simulation

For game play simulation, we built a prediction model trained on human player data.

3.1.1 Data

Data Collection

The data required to train the prediction model is obtained through tracking of actual players of the game in production. As there are a lot of daily active players for the game Candy Crush Saga, the possible data obtainable is extensive. To avoid possible problems with clients in the live game, we decided to track moves from only a small subset of the total active devices at any time (~1%) for only a short range of time (~2 weeks). Given the big number of players of Candy Crush Saga this creates a good sample of the players. The devices being tracked were selected at random. If a device is among

the ones selected for tracking, a move triggers the state recording. The state recording records the state, that was present just before the move was executed as well as the move that was executed. Each move is tied to a player through a unique identifier of the player. Players are grouped into quartiles by their average success rate on multiple levels. These quartiles can be considered as groups of players of different skill levels. We were curious if players of different skill levels play differently and whether these differences in playing could be picked up by a CNN. Therefore, we created the following two data sets from the tracked player data:

> **Unfiltered data set (UDS)**: Uniformly distributed over all levels from random players.
>
> **Filtered data set (FDS)**: Uniformly distributed over all levels from randomly selected players of a set of filtered players. The filter applied selects players, that are in the first quartile of players, that on average need least attempts to succeed in levels. This filter is applied as it can be seen as a proxy to the player's skill level.

For training the CNN, the data is separated into a training and a holdout validation set as well as another held out test set. The split is done with 4500 stat-action pairs per level for the training set and 500 state-action pairs per level for both the validation and test set. The validation set helps to gauge whether or not the model is overfitting the training data. As we create a dependency between the model and the validation data set by changing the architecture and hyperparameters to maximize the validation set's accuracy, the test set is needed to draw unbiased conclusions about the performance of the model.

Data Representation

States: A state is represented as a 9x9 grid with 102 binary feature layers. One feature layer is a 2-dimensional array (with dimensions 9x9). When stacked together the feature layers form a 3-dimensional array. There are four types of feature layers:

1. Feature layers that represent items on the board. Those feature layers have a 1 at every position, where the item represented by that feature plane appears on the board. One example of an item is a candy. In total, there are 80 feature planes of this type. If a given item is not at all present, the feature layer if filled with 0.

2. Feature layers that encode special objectives of a level, for instance that the player should create a number of striped special candies. Each of these feature planes stands for a particular objective of a level. Those feature planes are filled with 1, when the particular objective is still to be fulfilled at a given state. Otherwise these feature planes are filled with 0. In total, there are 20 feature planes of this type.

3. One feature layer that encodes all tiles that can be involved in a legal move. The position of any item that can be part of a legal move is represented by a 1 at the respective position on this feature plane.

4. One feature layer filled with 1. We included this feature plane to allow the network to learn a bias depending on the position of the board. This means it could for example learn that swaps at the edge of the board should be preferred over swaps in the middle of the board. Another effect of this plane is that it helps to identify the edges of the board. Since all planes are 0-padded before a convolutional operation to maintain the dimensions, without this feature plane it would not be possible to distinguish between patterns appearing in the middle or the edge of the board.

Moves: Moves can be horizontal or vertical swaps of positions of two items. The moves are encoded as a scalar by enumerating the inner edges of the game grid. The enumeration is done by first enumerating all horizontal swaps from left to right, continuing with the vertical swaps also from left to right. The enumeration is 0-indexed. Figure 3.1 shows the move encoding on the 9x9-grid of the game. On a 9x9-grid there are 144 possible moves.

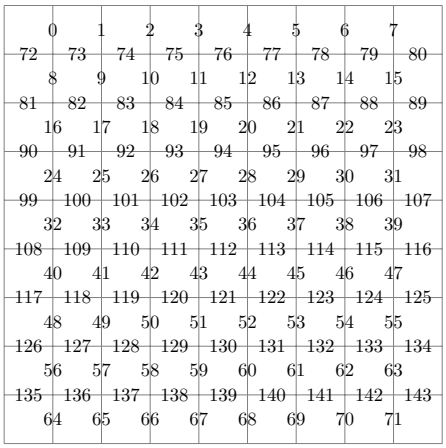


Figure 3.1. Encoding of moves

3.1.2 Prediction Model for Human Moves

CNNs are able to capture structural information from data with grid-like topology [22, 25]. They achieved superior performance for predicting expert moves in other, similar games [13, 12]. Therefore, they will serve as the prediction model for predicting a move given a state. There are many possible ways of designing a CNN. The different components that are used then lead to different hyperparameters that need to be adjusted. Hyperparameters are parameters that influence the CNN, but are not learned during

the training process. In the following we will first discuss the choice of the network architecture and continue by discussing the choice of hyperparameters.

Network Architecture

Implementing a feature for tracking state-action pairs while adhering to quality requirements and the release cycle for Candy Crush Saga required some time. Therefore, we conducted a pre-study using state-action pairs recorded from game rounds played by the MCTS-agent. This pre-study was conducted to gauge which architecture performs well for the problem at hand. Once we obtained the human player data, we tested the most promising architectures with that data.

Convolutional Layers: The parameters of a convolutional layer that can be adjusted are the numbers of filters, the stride as well as the kernel size. We experimented with different number of filters and different kernel sizes. We found that a kernel size of 3x3 for all convolutional layers performed best. Changing only the kernel size of the first layer from 3x3 to 5x5 lead to a performance decrease of 3% points. For the number of filters, we started with 102 filters per layer and continued decreasing the number of filters gradually. We found that there was no decrease of prediction accuracy on the validation data set until 35 filters per layer. As we prefer less complex models over more complex models, we chose to use 35 filters per convolutional layer. As the grid of the game is only 9x9, we do not want to reduce it further by using a stride larger than 1.

Output Layers: As mentioned above, we encode a move as a scalar in the range $[0, 143]$. This can be seen as equivalent to having 144 classes in an image classification problem. We therefore looked into the state-of-the-art for output layers in image classification problems. The early successful image classification networks such as AlexNet [25] and VGG-Net [30] use fully-connected layers with dropout regularization before applying the softmax function to create a vector of probabilities of each class. More recently Lin et al. [37] suggested an approach where the model's last convolutional layer applies $k =$*number of classes* filters. Then global average pooling is applied to the activation maps to obtain k scalars. To those scalars, the softmax function is applied. Some common network architectures then add a linear combination layer to those k scalars [46, 47].

We experimented with networks with the following layers at the end of the network. All of those layers were followed by a softmax layer: (1) two drop-out regularized fully-connected layers, (2) a global average pooling (GAP) layer as well as (3) GAP layer followed by a linear combination layer. We found that models with output layers using global average pooling were performing significantly better ($\tilde{3}\%$) than the ones using drop-out regularized fully-connected layers. Adding the linear combination layer after the GAP layer did not decrease the training loss nor improve the validation accuracy. We therefore chose to use a GAP layer only followed by a softmax layer as output layer.

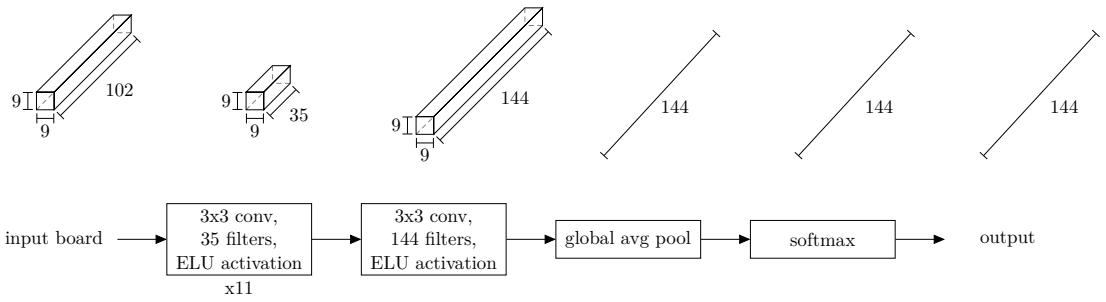


Figure 3.2. Architecture of network and dimensions of data at different stages

Activation Function: A big number of the prominent CNN architectures in academia use ReLU activation functions. This includes the CNN networks used for the closely related problem of predicting moves in Go [12, 13, 14], as well as popular architectures such as the VGG network [30], the inception network [31] or the ResNet [32]. Despite the popularity of the ReLU activation function, we experimented with ELU activation function proposed in [35]. We found that using ELU activation function improved validation accuracy by about 2.5% points.

Other Components: We considered adding batch normalization between layers. This, however, did not improve the performance of the network. We hypothesize that this is because the ELU-activation function as opposed to the ReLU-activation function does not create as much of a bias-shift for the next layers as it has a deactivated state of -1 as opposed to 0. A bias shift occurs when the mean activation of a layer is greater than 0. This layer then acts as a bias to the subsequent layer [35].

Additionally, we experimented with constructing the network as a residual network. The architecture used for this experiment consisted of 20 blocks of two convolutional operations and a shortcut connection around those two convolutional layers. This network has a much higher complexity as the networks described previously. However, we found that this architecture was not able to reduce the loss during training further than the other model, and consequently was not able to reduce the validation accuracy either. As the improved complexity of the model did not lead to any increase in accuracy, we opted to stick with the less complex architecture described in figure 3.2.

Hyperparameters

The architecture used only requires the learning rate and batch size to be set as hyperparameters. We found that a learning rate of 0.0005 and a batch size of 1024 lead to the best results.

Move Selection

When using the neural network as an agent, moves are selected greedily from the predictions. We also tried selecting moves by choosing them at random with the probability of the value they were assigned. However, this approach yielded worse results.

3.2 Prediction of Players' Success Rate

The SR per level obtained by the agents does not directly map to the SR per level obtained by players. We also neither expect nor require that. What we require is a strong relation between the players' SR and the agents' SR.

The distributions of SR are strongly right-skewed for both the players' SR and SR obtained by each agent (see Figure 4.2 in the results chapter). We therefore apply a log transform to both variables to obtain less skewed distributions. After observing the log-log transformed bivariate distributions between the players' and each of the agents SR, we determine a cut-off point of the agents' SR until which a linear regression model can be fit. For the values beyond the cut-off points, we form the mean to give a prediction. We found that below a certain SR there was no linear relationship between the log of the agents' and the log of the players' SR. This can be explained by the fact that the levels have been designed to be solvable for human players within a certain number of attempts. Under a linear regression model for very low SR values of the agents this would require there to exist also proportionally low players' SR values. Since those very low players' SR do not exist, there is no linear relationship between the agents' and the players' SR below a certain SR of the agent.

3.3 Experiment Design

To evaluate the research question, we conduct an experiment comparing the different approaches. The experiment consists of three stages:

1. Training of CNNs
2. Game play simulation
3. Creation and evaluation of predictions

1. Training of neural networks: We train a CNN for both the data set containing moves from players of any skill level (UDS), and the data set containing moves from the first quartile of players (FDS). Both models are trained with the same architecture and hyperparametes for the same number of steps. There are about 2400 levels released in the game Candy Crush Saga. For training the neural network we use 4500 state-action pairs per level for the levels 1-2150, both for FDS and UDS. This way each level is represented with the same number of state-action pairs. The validation set and test set

both consist of 500 state-action pairs per level. Figure 3.3 shows the first stage of the experiment.

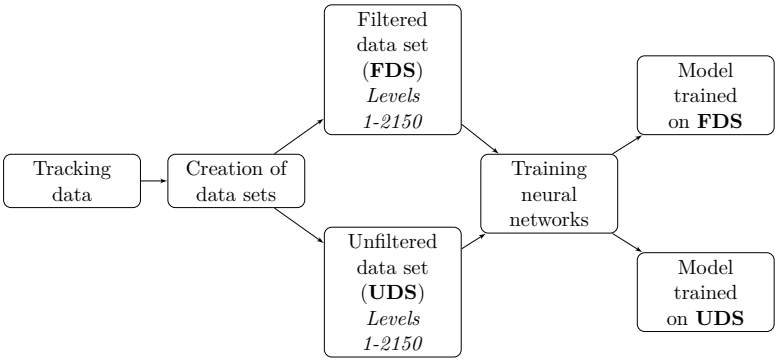


Figure 3.3. Experiment: Training of CNNs. Data is tracked from a fraction of all active users. We then create two data sets. One sampled from all players and one sampled from players that on average use less moves per level. Those data sets are then used to train two different models.

2. Game play simulation: For game play simulation, an agent interface is integrated into the game engine. In its essence, this interface receives a number of possible moves, as well as the game state from the game engine and selects one of the possible moves. As the way of selecting moves from the available moves is abstracted from this interface it is capable to use different approaches for choosing moves. In other words, the agent interface can be used with different agents for selecting moves. For this work, the bot interface uses either a MCTS-based agent or a CNN-based agent. When using the CNN-based agent, at each step, the agent interface sends the state obtained from the game engine as a request to the model serving component. The response to this request is the predicted probability distribution over all moves. The agent interface then chooses the move with the highest value in a greedy manner.

We ran a simulation of the game play on levels in the range [2151, 2400]. This range is selected as it does not include any of the levels that the CNN was trained on and includes the latest released level at the time of writing. Levels where time is the limiting constraint are out of the scope of this research, as it would require a way to approximate the time a real player needs to choose a move within a time level. Additionally, the simulation of the game play would have to be run taking the time a user needs to make a move, leading to a significantly longer time to execute simulations of a level. Excluding the time-constrained levels in the previously mentioned range of levels leads to a total of 231 levels.

For each level, we run a number of attempts on solving the level. In comparing the performance of the different agents, we could argue for using the same number of attempts per agent or constraining the number of attempts by the time required for executing the same amount of attempts on the same hardware. We chose to run the

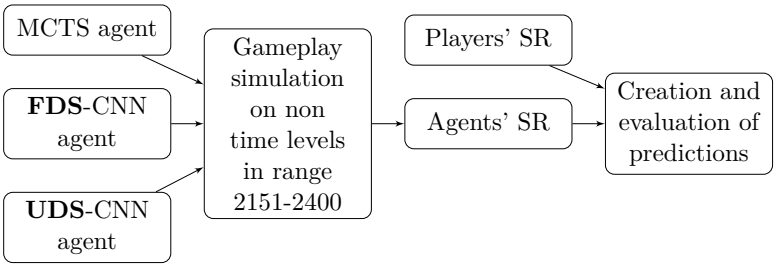


Figure 3.4. Experiment: Game play simulation, and creation and evaluation of predictions. This flow chart outlines the last two stages of the experiment. The predictions of the models trained in the previous stage are used as an agent to selecting a move given a state. These agents, alongside the MCTS agent, are then used for game play simulation. Each of the agent creates SR measures per level. We then form a model between each of the agents' and the players' SR. The predictions of these models are then evaluated.

same number of attempts per agent. This number is constrained by what can be run in a reasonable amount of time using the MCTS-based agent. We chose to run a all agents for 100 attempts per level. This took about one week when using the MCTS based agent and few hours when using the CNN-based agent on 36 CPUs. Due to the lower computational complexity, a CNN based agent could run significantly more attempts under the same time and hardware constraints than a MCTS based agent. To demonstrate the effects of running more attempts per level in addition to running 100 attempts for the CNN-based agents, we also experiment with running 1000 attempts. Running 1000 attempts per level with the CNN-based agent still requires considerably less time than running only 100 attempts using the MCTS-based agent.

3. Creation and evaluation of predictions: For each of the simulated levels, we query the SR of players from recordings of game rounds played in the live environment. We then form predictions for the players' SR following the approach outlined in section 3.2.

For evaluating the predicted players' SR that was inferred from the SR of the different agents, we calculate both the mean absolute error and root-mean-squared error evaluation measures. Those measures are explained in the following section. Figure 3.4 visualizes this stage and the previous of the experiment.

3.4 Evaluation Measures

3.4.1 CNN Predictions

For evaluating the predictions of the CNN, we use prediction accuracy as a measure. The number of possible switches on a board and therefore the number of prediction classes is 144. If at every state we would consider all 144 possible switches, the probability of picking the same move as the player for that state when picking at random would be

$\frac{1}{144} \approx 0.69\%$. However, at any given state out of the possible switches only a subset is legal. The tiles that can be part of a legal move are known both to the actual players and the CNN. Therefore, it would be possible to just pick randomly from all legal moves. As such, the probability of picking the same move as a player when randomly picking from all legal moves is a more appropriate baseline than picking randomly from all moves (including the illegal ones). The number of legal moves varies from state to state. Therefore, the probability of choosing the same move as a player when picking from all legal moves at random is calculated by taking the average of the probabilities for each state:

$$\bar{P}_{lm} = \frac{1}{n} \sum_{i=0}^{n} \frac{1}{lm_i} \ , \tag{3.1}$$

where $\bar{P}_l$ is the probability of picking the same move as the player when choosing randomly from all legal moves for the whole data set, n is the number of states in the data set, and l_i is the number of legal moves for state i. The probability of choosing the same move as a player when picking randomly from all legal moves on average over the data set is 16.67%.

3.4.2 Regression Models

In order to draw a more complete picture of how close predictions of the different agents' regression models are to the actual observed values, both mean absolute error (MAE) and root-mean-squared error (RMSE) are used as evaluation measures. The MAE is formed as the mean of the absolute differences between the predicted players' $\widehat{SR}_i$ and observed SR_i of all levels n:

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} |SR_i - \widehat{SR}_i| \tag{3.2}$$

RMSE is calculated by taking a square root of the mean of squared differences between prediction $\widehat{SR}_i$ and actual observation SR_i of all levels n:

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (SR_i - \widehat{SR}_i)^2} \tag{3.3}$$

RMSE has the property of penalizing large deviations from the predicted value strongly as the errors are squared before they are averaged. This is a desirable property since large deviations from the actual values make the prediction model less reliable. However, RMSE also has the property of tending to be increasingly larger than MAE for larger sample sizes. This tendency can be revealed, when looking at using MAE to form upper and lower bounds for RMSE:

RMSE $\geq$ MAE: RMSE will be equal to MAE if all errors have the same magnitude, otherwise RMSE will be larger than MAE.

RMSE $\leq$ MAE $*$ $\sqrt{n}$: The difference between RMSE and MAE is largest when the error is produced by one sample only and is 0 for all other samples. In that case RMSE = MAE $* \sqrt{n}$ and is therefore related to the sample size. This is an undesired property, since the number of sample considered for the linear regression depends on how many of the data points are beyond the cut-off point and thus how many data points are considered for the linear regression. This number varies between the agents.

Chapter 4

Results

4.1 Training of CNNs

4.1.1 Prestudy using data produced by game play of the MCTS-based agent

In choosing an architecture for the CNN, we conducted a pre-study on data that have been obtained by recording state-action pairs encountered when simulating game play using the MCTS-based agent. We combined several architectural elements that are currently popular in academia. As training the models took around 24 hours to converge on a Nvidia Tesla K80 GPU using the numerical computation library Tensorflow [48], we only searched over a small grid of hyperparameters. The search values per hyperparameter are listed in Table 4.1.

The names of the models are composites of their main characterizing structural elements.

ReLUFCDropout: A network consisting of 12 convolutional layers followed by two dropout regularized fully connected layers. For all activations, the ReLU activation function has been used.

ReLUAvgPool: This network replaces the fully connected layers by changing the number of filters of the last convolutional layer to the number of classes and then performing global average pooling. The activation function used was ReLU.

ELUAvgPool: The architecture of this network is the same as the previous one with the only difference of replacing the ReLU activation function with the ELU activation function.

ResELUAvgPool: In this architecture, the regular convolutional layers used in the other architectures were replaced by 20 blocks of two convolutional operations and a shortcut connection around these two.

The model that achieved the best performance or on par with more complex models (within the margin of statistical errors), while being the one that required the least

hyperparameters to be tuned as well as the least amount of network parameters, was the ELUAvgPool model. The architecture is visualized in Figure 3.2. Table 4.2 shows the best validation accuracy achieved by different model architectures.

Table 4.1. Hyperparamerter search grid

Hyperparameter	Search Values
Learning rate	{5e-4, 1e-4, 5e-5}
Batch size	{128, 512, 1024}
Dropout keep probability*	{0.4, 0.5, 0.6}

*if applicable

Table 4.2. Neural network training results on MCTS data

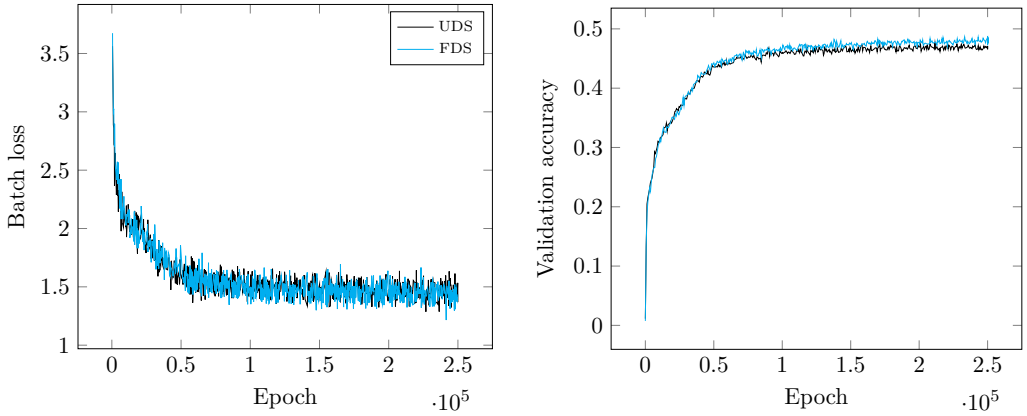
Model	Hyperparameters	Val. Acc.
ReLUFCDropout	learning rate, batch size, keep probability	25.72
ReLUAvgPool	learning rate, batch size	28.59
ELUAvgPool	learning rate, batch size	32.35
ResELUAvgPool	learning rate, batch size	30.01

4.1.2 Training on human player data

We used the best performing architecture in the pre-study as a starting point for further exploring different architectures with the data obtained by tracking players. For the human player data, the model achieved a validation and test accuracy of around 46% when trained on unfiltered player data (UDS) and of around 47% when trained on the first quartile of best players (FDS). It is interesting to note that the achieved accuracy was significantly higher when using human player data instead of data produced by the MCTS-agent. We tried to improve the accuracy by adding complexity to the model in form of more filters per layer as well as adding a layer of linear combination after the average pooling layer. However, none of these complexity increasing measures improved the model's performance significantly. Since the ELUAvgPool model was the simplest model that provided the highest accuracy, we went on to use this network architecture to predict actions during game play. Table 4.3 shows the validation and test accuracy obtained during one training run of the ELUAvgPool model on both the filtered (FDS) and unfiltered data set (UDS). Figure 4.1a shows the batch loss obtained during training the neural network, Figure 4.1b shows the validation accuracy over the number of epochs taken. We can see both in Table 4.3 and Figure 4.1b that the validation accuracy is slightly higher when training and evaluating on the filtered data set. The same is true for the test accuracy. Running the training on both data sets, each for 5 times, yielded very similar test accuracy values for each run. The standard deviation of the test accuracy was 0.082% and 0.092%, when trained the FDS and UDS data set respectively. Therefore, the differences in the test accuracy between the two data sets are statistically significant.

Table 4.3. Neural network training results on player data

Model	Data set	Val. acc.	Test acc.
ELUAvgPool	UDS	46.60	46.51
ELUAvgPool	FDS	47.71	47.57



(a) Training batch loss

(b) Validation accuracy

Figure 4.1. Visualization of training process of ELUAvgPool model

4.2 Game Play Simulation

After obtaining the prediction models from training the CNN on the two data sets, we use them as a policy. During game play, the predictions of the models, as well as the values obtained through MCTS serve as agents, evaluating all actions $a \in A$ given a state s. The action a_{max} that received the highest rating is then executed by the agent interface. Note, that while executing the simulations of MCTS, actions are not selected greedily, but following a UCB selection strategy (see section 2.3.3). Only after all MCTS simulations (one MCTS simulation refers to a full cycle of steps 1 to 4 in Figure 2.3) for a given state are executed, the action with the highest value is chosen. The MCTS based agent within this thesis uses 200 Monte Carlo search simulations. This number proved in previous experiments to produce good results while still running in a tolerable amount of time [10]. Selecting and executing an action leads to a new state s' with a new set of possible actions A'. The available actions are then again evaluated by the respective agents.

This loop of executing an action given a state is continued until a terminal state is reached. In the levels within the scope of this research, the terminal state is reached, when either the objective is fulfilled or the maximum number of moves have been carried out. When the objective has been achieved the level is won and the attempt is counted as a success. Dividing the number of successes by the number of attempts leads to the success rate of the respective agent for each level.

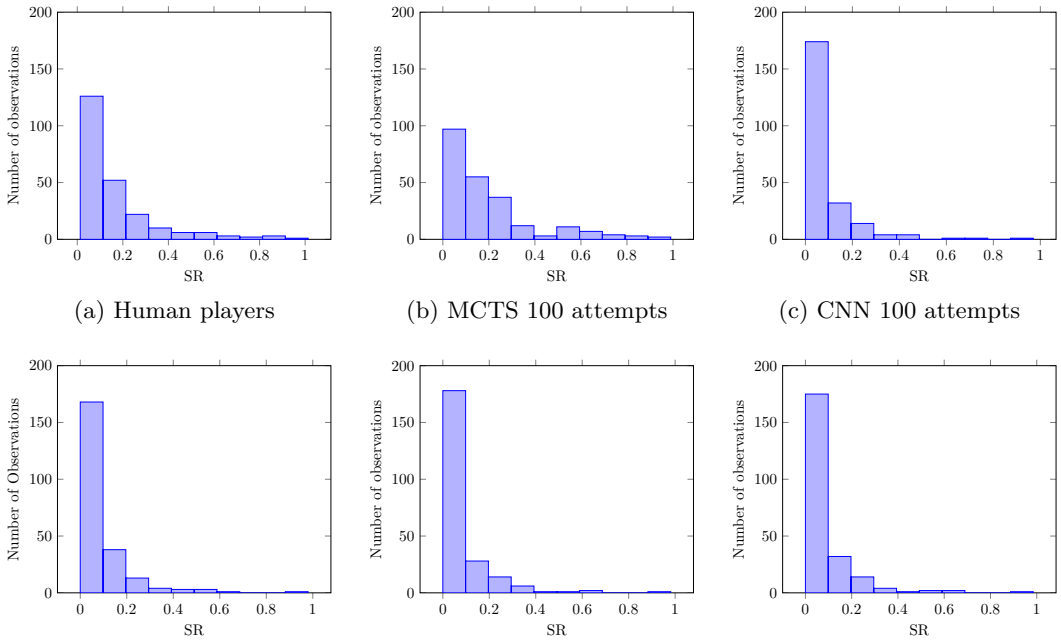
Table 4.4. Game play simulation time per agent

Agent	Attempts	Simulation Time (minutes)
MCTS	100	5,250
UDS-CNN	100	105
FDS-CNN	100	99
UDS-CNN	1000	1,052
FDS-CNN	1000	992

The different agents run either 100 or 1000 attempts per level. One could argue to compare the results obtained by the different agents under constraint of the same number of attempts, or under the same time constraint on the same hardware. As simulating game play using the CNN-based agents takes considerably less time on the same hardware and we wanted to demonstrate the effect of running more attempts. In Table 4.4 we present the time it took to simulate game play the levels 2151-2400 with the respective agents. We can see that the CNN-based agents run about 50 times faster than the MCTS-based agent.

To obtain the corresponding players' SR to the agents' SR, we query the number of attempts and number of successes of all players for the respective levels from recordings of the game rounds. As we can see in Figure 4.2, the distribution of the SR of all artificial agents and human players is highly right-skewed. We can also observe that the distribution of the MCTS-based agent is slightly less right-skewed than the other distributions. The MCTS-based agent more often achieves higher SR compared to both the players and the CNN-based agents.

Figure 4.3 shows the difference between the SR of the three artificial agents (ΔSR), when run for 100 attempts, ordered by that difference (rank ΔSR) We can see that the MCTS outperforms both the CNN trained on data of all players as well as the CNN trained on data of only the best first quartile. There are only a few levels, where the CNN-based agents achieve a higher SR than the MCTS-based agents. These are the data points of the red solid line and the black dotted line that fall into the negative range. The sum of the differences in the SR over all considered levels $\sum \Delta\text{SR}$ is 25.84 and 24.95 favoring the MCTS agent when comparing against the UDS-CNN and the FDS-CNN respectively. Comparing both CNN-based agents against each other produces an approximately point-symmetric line centered around the median of the ΔSR rank, indicating that the two agents perform similarly well. When taking the sum of differences in SR, we obtain a slightly positive value of 0.89 favoring the FDS-CNN agent. This indicates a very minute increase in performance when training the agent on the quartile of best players (FDS).



(a) Human players (b) MCTS 100 attempts (c) CNN 100 attempts

(d) FDS-CNN 100 attempts (e) UDS-CNN 1000 attempts (f) FDS-CNN 1000 attempts

Figure 4.2. SR distributions: Note that the player SR has been scaled (see beginning of section 4.3)

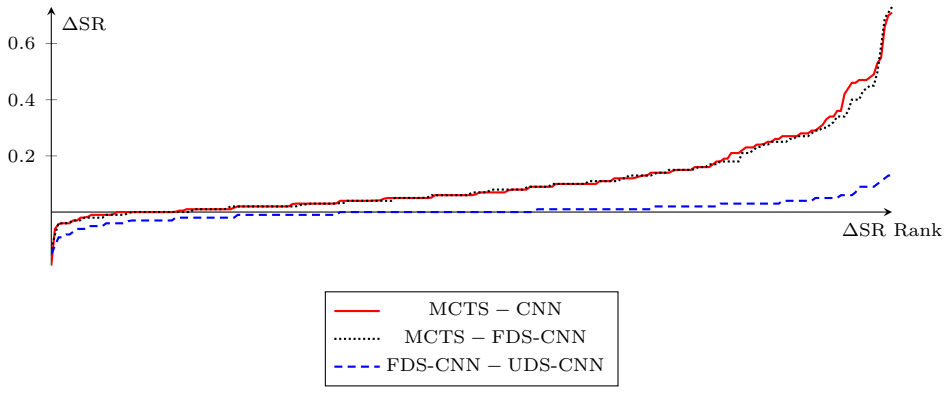


Figure 4.3. Difference of the agents' performance per level. The values are ordered by that difference.

4.3 Prediction of Players' Success Rate

Note, that throughout this thesis the players' success rate has been scaled by dividing the actual players' SR by the difference between the maximum and minimum value of players' SR observed in the data set. This has been done to not expose the actual players' SR for privacy purposes, while maintaining the shape of the distribution:

$$SR_i^{(sp)} = \frac{SR_i^{(p)}}{max(SR^{(p)}) - min(SR^{(p)})} \ , \tag{4.1}$$

where $SR_i^{(sp)}$ is the scaled players' SR for the ith level, $SR_i^{(p)}$ is the actual players' SR for the ith level, $max(SR^{(p)})$ denotes the maximum actual players' SR observed within all levels, and $min(SR^{(p)})$ denotes the minimum players' SR observed within all levels. Note, that scaling is applied over the SR of all players, not each player individually. The agent SR has not been scaled. Since both the players' and agents' SR are highly skewed (see Figure 4.2), we applied a log transform on both to achieve less skewed distributions.

Depending on the success rate of the bot, we use a different method for predicting the players' success rate. When the agents' SR is 0 the log is not defined and thus those data points cannot be part of a linear regression model, where the two variables have been transformed using a log. When running the agents for 100 attempts, therefore the lowest considerable agents' SR is 0.01. However, we found that even when running the CNN-based agents for 1000 attempts, for data points where the agents' SR was below 0.01 there was no linear relationship between the log transforms of players' and agents' SR.

Therefore, we use a linear regression model for data points where the agent SR ≥ 0.01 and estimate the SR for all other points using the mean of the players' SR.

4.3.1 Linear regression model for players' success rate prediction

Figure 4.4 plots the log of the SR of each of the agents against the log of the scaled success rate of the players. The data points, where the agents failed to succeed within the number of attempts (i.e. SR=0) are not part of this plot as $log(0)$ is not defined. This should be taken into account when reading the plots.

Data points that have a value below this can only occur when run for more than 100 attempts. Further, for data points that fall in agent SR < 0.01, there seems to be no linear relationship between agent SR and player SR. A linear regression model makes the following assumptions:

1. A linear relationship between dependent and independent variable
2. Statistical independence of the errors
3. Homoscedasticity of the errors
4. A normal distribution of errors

Plots for checking against violations of these assumptions are available in the appendix of this document. To check against violations against (1) the linear relationship assumption, we plot the predicted values of player SR against the observed values of players' SR (see Figure A.1). The points in these plots appear to be symmetrically distributed around $y = x$ with constant variance, indicating that there is no violation of the linear relationship assumption. Checking for violations against (2) the statistical independence of errors, we plot the agent SR against the residuals (see Figure A.2). The residuals are randomly and symmetrically distributed around 0. Consecutive errors are not correlated. Thus, there appears to be no violation of the statistical independence of errors assumption. Checking for violations of (3) the homoscedasticity of errors assumption, we plot the predicted player SR against the residuals (see Figure A.4) as well as the agent SR against the residuals (see Figure A.2). For the CNN based agents the variance appears to be a bit lower when the predicted SR is higher, however there are much less data points, where the SR is predicted to be higher. This can also be explained by how we obtain the values for the agent's SR. While we present the agents' SR as ratios, those ratios stem from the discrete number of successes made per the (also discrete) number of attempts. This means a lower SR is connected to a lower number of successes and thus a higher uncertainty. For instance, if we compare observing 2 successes versus 50 successes per 100 attempts, being off by 1 success in the first case means being off by 50% in the former case versus being off only by 2% in the latter case. Still, in total the errors do not seem to grow as a function neither of the predicted players' SR nor the agents' SR, and thus there appears to be no major violation of this assumption. To check if the errors appear to be (4) normally distributed, we use a normal probability plot [49]. For agents that run for 100 attempts, the points fall in a fairly straight line indicating that there is no violation of (4) the normal distribution of errors assumption. This is also confirmed by the p-values of the Anderson-Darling test [50] testing for the null hypothesis that a sample is drawn from a normal distribution. They are 0.67 for the MCTS-based agent, 0.52 for the UDS-CNN agent and 0.35 for the FDS-CNN agent, when those agents are run for 100 attempts. However, when the CNN-based agents are run for 1000 attempts, there appears to be a slight violation of (4) the normal distribution of errors assumption. Some points at both ends of the theoretical quantiles do not fall on the line. The Anderson-Darling test p-values for both the FDS and UDS agents are close to 0, indicating that the errors are not normally distributed. However, it appears that these deviations are only caused by a small number of samples and therefore do not indicate a major violation of the assumption.

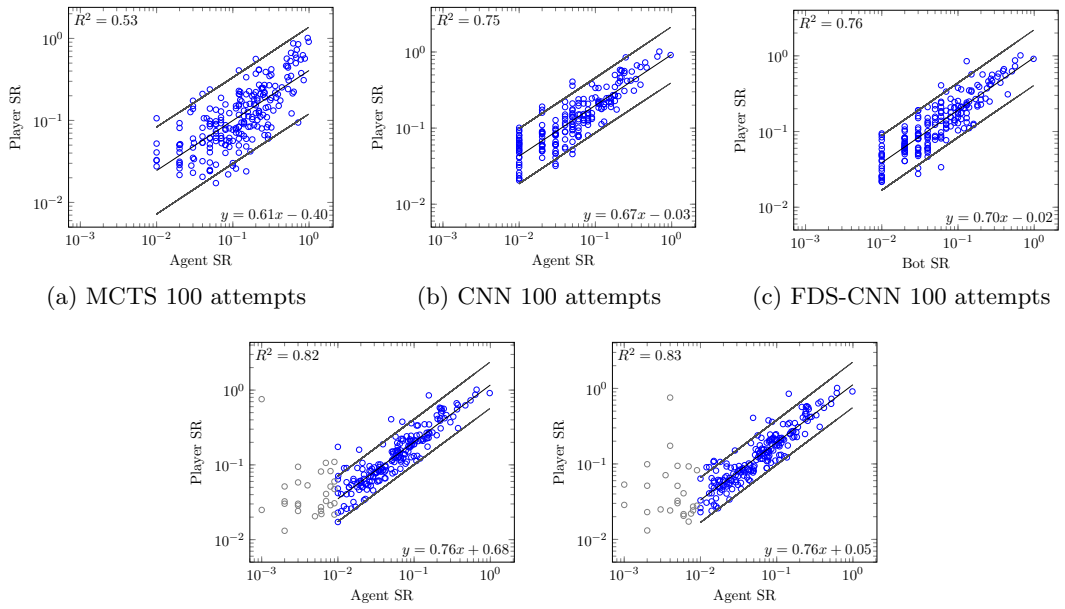
In Figure 4.4 we can observe that the MCTS agent produces a much higher variance than all the other agents. The linear regression models of the CNN-based agents are able to explain a larger proportion of the variance of the dependent variable from the independent variable. This is reflected in the larger values of the coefficient of determination R^2 for the CNN-based models. The R^2 values for data obtained from the agents are 0.53, 0.75, and 0.76 the MCTS-based agent, the UDS-CNN-based agent, and FDS-CNN-based agent respectively, when the agents were run for 100 attempts. When the CNN-based agents are run for 1000 attempts the R^2 values are 0.82 and 0.83 for

Table 4.5. Number of data points within different ranges of agent SR

Agent	Attempts	SR ≥ 0.01	SR < 0.01	SR $= 0$	Total
MCTS	100	220	11	11	231
UDS-CNN	100	192	39	39	231
FDS-CNN	100	195	36	36	231
UDS-CNN	1000	191	40	12	231
FDS-CNN	1000	195	36	11	231

the UDS-CNN-based agent, and FDS-CNN-based agent respectively. However, we have to note that the MCTS agent was able to succeed in more levels than any of the CNN-based agents (see Table 4.5). Therefore, the plot of the MCTS based agent features more data points than the other plots. At the same number of attempts, the MCTS-based agent fails to win in 11 levels. The CNN-based agent trained on the unfiltered data set of player state-action pairs failed to win in 39 levels, the CNN-based agent trained on the first quartile of players that needed least actions, failed to win in 36 levels. When running the CNN-based agents for 1000 attempts the number of levels where the SR is 0 decreases to 12 and 11 for the CNN-based agent trained on unfiltered data and the CNN-based agent trained on filtered data respectively. However, as we can observe from Figure 4.4 there seems to be no linear relationship between agent SR and player SR for agent SR < 0.01. This can be explained by the fact that levels are designed to be solvable by human players within a certain number of attempts. This acts as a lower bound for the players' SR. When assuming a linear model for very low agents' SR values, we would also need to find players' SR values that are proportionally low. Since this is not the case, we can assume that a very low agent SR stems from an issue with the agent playing the level and not the difficulty of the level itself.

Still, running the agents for more attempts decreases the variance and increases the coefficient of determination of the linear regression model that was fit for agent SR $\geq$ 0.01. We can also observe a narrower 95% prediction interval. Meaning that when predicting in which range the players' SR should fall given a new observation of the agent's SR, we can predict a narrower range. There appears to be no major difference in results between the UDS and FDS CNN-based agents.



(a) MCTS 100 attempts (b) CNN 100 attempts (c) FDS-CNN 100 attempts

(d) CNN 1000 attempts (e) FDS-CNN 1000 attempts

Figure 4.4. Success rate obtained by the different agents plotted against the scaled success rate of players: The black line shows the linear regression for the data points color coded in blue. The grey line above and below the black line shows the 95% prediction interval. The points color coded in grey are excluded from the linear regression. On the top left corner of each graph the coefficient of determination R^2 of the regression model is displayed. The bottom right shows the equation for the fitted line.

4.3.2 Prediction using the mean of the observed players' success rate

For data points that are outside the range of consideration for the linear regression model, a prediction of the SR is formed using the mean of the values that fall in that range. Figure 4.5 visualizes the players' SR of these data points as well as the mean with standard error bars for each agent. Since the agent fails to win at these levels, it would be desirable that the observed values in this range have a low player SR indicating that the levels are hard. In Figure 4.5 we can see that the observed player SR tends to be fairly low, however it shows a fairly high variance over a rather large range of values. Table 4.6 lists the mean of the observed values for each agent as well as the number of samples n, the standard deviation (std. dev.) and standard error (std. err.). The data produced by the MCTS agent shows a higher standard deviation than the data produced by the other agents. This is mostly attributable to the smaller sample size. The mean of the log of the scaled observed player data is between -1.33 and -1.30 for the CNN-based agents and -1.18 for the MCTS-based agent. This mean is used as the predicted player SR for agent SR < 0.01.

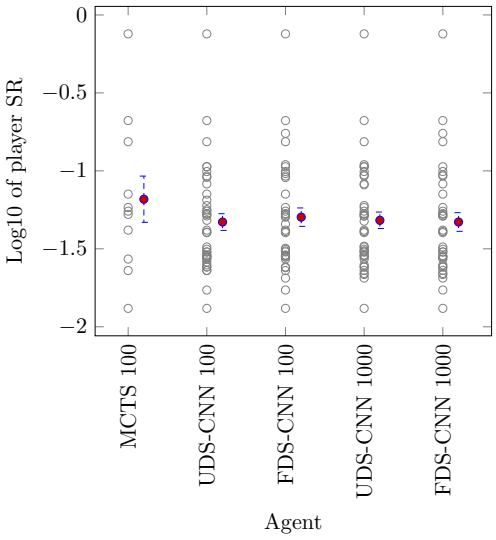


Figure 4.5. Data points that fall into the mean prediction part. The red dots mark the mean. The blue whiskers mark standard error bars.

Table 4.6. Mean observed log of scaled player SR

Agent	n	mean	std. dev.	std. err.
MCTS 100	11	-1.18	0.47	0.15
UDS-CNN 100	39	-1.33	0.33	0.05
FDS-CNN 100	36	-1.30	0.35	0.06
UDS-CNN 1000	40	-1.32	0.33	0.05
FDS-CNN 1000	36	-1.33	0.35	0.06

4.3.3 Performance of SR prediction per agent

In Table 4.7 we list values for the performance measures MAE and RMSE for each agent split by the two ways predictions are made for different ranges of the agents' SR. Within the linear prediction part, the CNN-based agent trained on unfiltered human state-action pairs, that ran for 100 attempts (UDS-CNN 100) achieved a 50% lower MAE and a 39% lower RMSE than the MCTS-based agent. Within the mean prediction part the same CNN-based agent achieved a 40% lower MAE and a 42% lower RMSE. Considering all predictions, the same CNN-based agent achieved a 37.5% lower MAE and a 33% lower RMSE than the MCTS-based agent. Note, that the difference in performance is lower when considering all parts as when looking at the individual prediction parts. This is due to the different number of samples in the different prediction parts and the fact that for the MCTS-based agents less samples fall into the range, where a prediction is made using the mean (see Table 4.5). The errors of the predictions in the mean part of the CNN-based model are higher than the errors of the predictions made by the linear regression model of the MCTS-based agent.

Table 4.7. Prediction performance measures

		MAE			RMSE		
Agent	Attempts	Linear part	Mean part	Total	Linear part	Mean part	Total
MCTS	100	0.21	0.36	0.22	0.26	0.47	0.28
UDS-CNN	100	0.14	0.25	0.16	0.18	0.33	0.21
FDS-CNN	100	0.14	0.27	0.16	0.18	0.35	0.22
UDS-CNN	1000	0.11	0.26	0.14	0.15	0.33	0.20
FDS-CNN	1000	0.11	0.28	0.14	0.15	0.35	0.20

Running the CNN-based agents for 1000 attempts decreases the MAE by 21% and RMSE by 20% for predictions made with the linear regression model for both agents when comparing with the performance measures for when the respective agents are run for 100 attempts. In total running the CNN-based agents for 1000 attempts translates to a 57% decrease of the total MAE and a 40% decrease of the total RMSE when comparing against the results produced by the MCTS-based agent, that ran for 100 attempts.

Chapter 5

Discussion

5.1 Differences in Results Caused by Different Data Sets

When changing the training data set for the neural network we found in section 4.1 that the validation accuracy slightly increased when using the filtered data set instead of the unfiltered one. This suggests that the way more skilled players play is more predictable than the way players from all skill levels play. The small effect, however, suggests that the neural network is not able to capture any major differences in the way the two population play. This could be either due to absence of any such differences or the network not being able to capture the differences. In section 4.2 we could observe a very minute positive improvement in terms of SR when simulating game play using a model trained on more skilled players compared to less skilled players. The changes only being minute could also stem from the interval of one quartile being too big and thus including too many players that do not expose a more advanced strategy.

5.2 Practical Implications

We found that while the MCTS-based agent is able to outperform the CNN-based agents, the SR results produced by the CNN-based agent allow for a better prediction of the SR of players. At the same time, executing the moves predicted by the CNNs greedily has a much lower computational complexity than making moves with a MCTS-based agent. Therefore, it is possible to run orders of magnitudes more attempts per level in less time on the same hardware. Our results showed that game play simulation using a CNN-based agent ran 50 times faster than using a MCTS-based agent. Additionally, in recent experiments we found that the bottleneck for the CNN-based agent was the database that was used to save the results of an attempt. Replacing the database sped up the game play simulation by another 150 times. This has concrete practical implications for the use case scenarios this approach can be used in. When supported with the appropriate infrastructure it is possible to run an evaluation of a single level, with 1000 attempts, in less than a minute. This would then make it possible to add an evaluation feature to the level editor, level designers use for creating levels. Level designers could

then get immediate feedback on the difficulty of the current version of the level they are designing allowing for more iterations in the design process. Being much faster in selecting moves using the CNN-based agent also allows for the possibility of watching the agent play the game live. In addition, using a model that can suggest a move based only on the current state allows for more use case scenarios than a MCTS based method that relies on signals obtained by play outs to a terminal state. The dependency of the MCTS algorithm on play outs make it not trivial to change the moves limit of a level dynamically. So, for instance, if we wanted to know how many moves more than the limit the agent would have needed to succeed in an attempt, this would be non-trivial. Using the CNN-based agent on the other hand, it is possible to simply play on following the predictions of the network even after the maximum number of moves has been reached. This is also of practical interest as players in Candy Crush Saga can acquire more moves, paying with in-game currency, when they fail to achieve the objective within the limit of moves. Therefore, letting the agent play beyond the limit of moves, may help to get a more granular view of the difficulty of a level.

The approach suggested in this thesis allows for a fast and cost-efficient evaluation of levels. Releasing levels that are well balanced in terms of their difficulty has a direct effect on the revenue made with these levels. The approach suggested in this thesis therefore combines recent findings in the field of AI into an approach that is commercially interesting. At King, for instance, we were able to reduce the time to give feedback on the difficulty of a level from 7 days, when using human test players, to currently 7 minutes using the approach described in this thesis. Improving the infrastructure, in which this approach is run, will further decrease the time to get feedback on the difficulty of new levels.

5.3 Transferability of Results to other Games

The approach suggested in this thesis was tested on one specific game, Candy Crush Saga. Therefore, the question arises how well the results obtained generalize to other games. We were able to successfully apply the approach suggested in this thesis to two other games. However, those two games were very similar to Candy Crush Saga with regards to the core game play. In general, we believe that specific features of a game make the method suggested applicable for that game. Among the most important features we consider a discrete or discretizable action space. Since the suggested method essentially looks at predicting moves as a classification problem, there needs to be a way of defining fixed classes. Also, the suggested approach assumes the Markov property. This means that the conditional probability of the subsequent states only depends on the current state and not a sequence of states. Games that cannot fulfill these two constraints, are not suitable for the approach suggested. In addition, this approach is most useful in games that are designed to evolve over time or have content added, because then findings from the simulations can be used to optimize the player experience.

5.4 Comparison between CNN and MCTS-based Method

From the previous sections of this chapter it became clear that the CNN based method proposed in this thesis offers several advantages over the MCTS based method. It is able to give better estimates for the success rate and therefore resembles human play better than the MCTS based method. Further, and probably equally importantly, the CNN based method has a considerably lower computational complexity and therefore opens up opportunities for new use cases that would not have been possible to execute using the MCTS based method.

These two advantages, however, come at a cost. For the MCTS based method to run, the only thing that is required is the set of possible moves at different states. The method in its most basic implementation with random roll-outs does not even require any information about the state itself other than the available moves. Even though in its most basic implementation the MCTS-based method is likely to not perform very well. This makes the MCTS-based method implementable with very little prerequisites.

The CNN based method on the other hand needs a way to record as much as possible information about the state as well as the available moves. In addition, for this method to be successful, data of state-action pairs created by human players need to be collected. Once the data has been obtained, it needs to be brought into a format suitable as input for the CNN. Using that data, the CNN needs to be trained. Only when having a trained model, the benefits mentioned before, become accessible. Essentially, the prediction performance increase and time improvement comes at the cost of a higher upfront effort. Still, these two benefits are essential for the method to provide real value to level designers. In addition, once a framework of executing these upfront steps is established, the time they require can be reduced significantly. At King, for example, we were able to completely deploy the same method to other, similar games within a few weeks with only one or two engineers working part of their day on implementing the method.

5.5 Future Work

5.5.1 More Granular Difficulty Measures

As mentioned in the previous section, it is possible to let the agent play beyond the limit of moves. This possibility can be used to get a more granular impression of the difficult of a level. The number of moves used in addition to the limit could then be used as another metric. This would allow to get a different perspective on the difficulty of a level as opposed to looking merely on the success rate. Therefore, we might be able to find out about how close the agent was to a success when it loses. This might be an indication of how well a level is perceived. If players are constantly far from reaching the goal on the level they might be less motivated than if they fail just slightly.

5.5.2 Investigation of Outliers

One problem we found when using CNN-based agents for game play simulation was that on some levels the agent was not able to succeed within the moves limit at all. We carried out a preliminary investigation on the type of levels, where the the CNN-based agents struggle. We found that levels that feature a special item, that is not possible to be triggered by the agent interface due to limitations in the headless mode of the game engine, were more often among the levels, where the CNN-based agents struggled. However, this special item only explains the low success rate on some levels. Therefore, a more thorough analysis of the levels with low agent's SR should be carried out, searching for more features that levels with a low agent's SR share.

5.5.3 Performance Improvement

Another way of tackling the problem with levels, where the agent is not able to achieve any success it to try and improve the performance of the CNN-based agents in terms of SR One way of doing so could be to improve the policy network obtained by training on player data, using policy gradient methods [51]. We could particularly improve the model towards levels that it failed to succeed on or had a very low success rate by including only those when adjusting the policy. However, on levels that the agent never succeeded on, this would be rather hard as we would not be able to get a positive reward from actions taken in those levels. This could be tackled by running a lot of attempts on those levels and hoping that at some of them the agent will succeed.

It could also be possible that continuing training on human player data only for levels where the agent failed on could improve the performance on those levels. However, this would have to be done carefully to not improve the performance at one level at the cost of other levels. Additionally, this approach would raise some issues regarding the possibility of overfitting to those levels and thus having a biased model.

Yet another way of tackling this problem would be to combine the MCTS-based agent with the policy network in a similar fashion as it was done in AlphaGo [12]. The policy of the CNN could be used to guide the roll out and search of the MCTS algorithm. However, this would come at the disadvantage of increased complexity compared to both the approach of greedy selection of the moves predicted by the CNN as well as using an MCTS algorithm with random roll out. However, it may be possible to mitigate this effect to some extent, by only using a small number of Monte Carlo simulations per move.

Chapter 6

Summary and Conclusions

In this thesis we examined, whether using a CNN-based agent would produce SR measures that could improve the prediction of players' SR in the game Candy Crush Saga compared to SR measures produced by a MCTS-based agent. To test this hypothesis, we first trained a CNN on 4500 state-action pairs of the first 2150 levels of Candy Crush Saga, that were recorded from human players. To see if the network would be able to learn a different strategy when trained on different skill levels of players, we obtained those state-action pairs once by randomly sampling from all players and once by randomly sampling from players that were in the first quartile of average number of attempts needed per level. This means those players on average needed less attempts to succeed on a level and can therefore be considered of a higher skill level.

We then ran an agent based on those trained models as well as a MCTS-based agent, each for 100 attempts, on 231 levels from which there were no state-action pairs in the training set of the CNN. Those levels were released after the levels in the training set. This resembles how the approach would be used in practice. The CNN would be trained on a number of levels that have been released to estimate the difficulty of levels that have not been released yet. We then formed prediction models to predict the players' SR from each of the agents' SR and evaluated the performance of these prediction models.

We found that, predicting the players' SR from the SR of a CNN based agent, that was trained on the set of state-action pairs randomly sampled from players of all skill levels, lead to a 37.5% lower MAE and a 33% lower RMSE than when predicting the players' SR from the SR of a MCTS-based agent, when run for the same number of attempts.

We found that the CNN-based agent was a weaker player than the MCTS-based agent. This was reflected by the number of levels that the CNN-based agent failed to win in 100 attempts. The CNN-based agent trained on the unfiltered data did no succeed on 39 levels where the MCTS based agent only failed to succeed in 11 levels. It was also clearly observable when looking at the ΔSR per level. The MCTS-based agent had on almost all levels a higher SR. When comparing the CNN-based agents that were trained on data sets obtained from differently skilled players, we found that there was only a very minute, not statistically significant, difference in their performance. The minute

difference in performance was also reflected in the fact that players' SR predictions made from either of the agents' SR lead to nearly the same MAE and RMSE.

Running simulations of levels using the CNN-based agent has a much lower computational complexity. In our experiments, the CNN based agents ran 50 times faster than the MCTS-based agent. It is therefore possible to run 1000 attempts per level using a CNN-based agent in much less time than running 100 attempts using the MCTS-based agent. We, therefore, also looked into how running the CNN-based agents for 10 times as many attempts affected the ability to predict the players' SR from the agent's SR. We found, that running the agents for 1000 attempts, the coefficient of determination increased from 72% to 82% and from 76% to 83% for the CNN-based agent trained on unfiltered and filtered data respectively. This translates to a 57% decrease of the MAE and a 40% decrease of the RMSE when comparing against the results produced by the MCTS-based agent that ran for 100 attempts.

All in all, these results suggest that the presented approach can considerably improve the usability of game play simulation for difficulty estimation in the game Candy Crush Saga. In addition, this approach should be transferable to a number of other games that have the Markov property and have a discrete action space.

Bibliography

[1] Juho Hamari, Nicolai Hanner, and Jonna Koivisto. "Service quality explains why people use freemium services but not if they go premium: An empirical study in free-to-play games". In: *International Journal of Information Management* 37.1 (2017), pp. 1449–1459.

[2] Kati Alha et al. "Free-to-Play Games: Professionals ' Perspectives". In: *In Proceedings of Nordic Digra 2014 Gotland,Sweden* (2014), pp. 1–14.

[3] Kati Alha et al. "Critical Acclaim and Commercial Success in Mobile Free-to-Play Games". In: *Proceedings of 1st International Joint Conference of DiGRA and FDG* (2016), pp. 1–16.

[4] Juho Hamari. "Why do people buy virtual goods? Attitude toward virtual good purchases versus game enjoyment". In: *International Journal of Information Management* 35.3 (2015), pp. 299–308.

[5] Tobias Brockmann, Stefan Stieglitz, and Arne Cvetkovic. "Prevalent Business Models for the Apple App Store". In: *Proceedings der 12. Internationale Tagung Wirtschaftsinformatik (WI)* March (2015), pp. 4–6.

[6] Magy Seif El-Nasr, Anders Drachen, and Alessandro Canossa. *Game Analytics*. London: Springer, 2013.

[7] Anders Drachen and Alessandro Canossa. "Towards gameplay analysis via gameplay metrics". In: *Proceedings of the 13th International MindTrek Conference: Everyday Life in the Ubiquitous Era on - MindTrek '09*. New York, New York, USA: ACM Press, 2009, p. 202.

[8] Alexander Zook, Eric Fruchter, and Mark O Riedl. "Automatic Playtesting for Game Parameter Tuning via Active Learning". In: *Foundations of Digital Games* (2014).

[9] A. Isaksen, D. Gopstein, and A. Nealen. "Exploring game space using survival analysis". In: *Proceedings of the 10th International Conference on the Foundations of Digital Games* Fdg (2015).

[10] Erik Ragnar Poromaa. ”Crushing Candy Crush: Predicting Human Success Rate in a Mobile Game using Monte-Carlo Tree Search”. 2017.

[11] Fernando Silva, Scott Lee, and Nicholas Ng. ”AI as Evaluator: Search Driven Playtesting in Game Design”. In: *Aaai* (2016).

[12] David Silver et al. ”Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529.7587 (Jan. 2016), pp. 484–489.

[13] Christopher Clark and Amos Storkey. ”Teaching Deep Convolutional Neural Networks to Play Go”. In: *Journal of Machine Learning Research* (Dec. 2014), p. 9.

[14] Kun Shao et al. ”Move Prediction in Gomoku Using Deep Learning”. 2016.

[15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. ”Deep learning”. In: *Nature* 521.7553 (2015), pp. 436–444.

[16] Toby Walsh. *Candy Crush is NP-hard*. Mar. 2014.

[17] Ahmed Khalifa et al. ”Modifying MCTS for Human-like General Video Game Playing”. In: *Ijcai* (2016), pp. 2514–2520.

[18] Rémi Coulom. ”Computing "Elo ratings" of move patterns in the game of Go”. In: *ICGA Journal* 30.4 (2007), pp. 198–208.

[19] Thomas Philip Runarsson and Simon M. Lucas. ”Preference learning for move prediction and evaluation function approximation in Othello”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 6.3 (2014), pp. 300–313.

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[21] Juergen Schmidhuber. ”Deep Learning in neural networks: An overview”. In: *Neural Networks* 61 (2015), pp. 85–117.

[22] Y. LeCun et al. ”Backpropagation Applied to Handwritten Zip Code Recognition”. In: *Neural Computation* 1.4 (Dec. 1989), pp. 541–551.

[23] Kunihiko Fukushima. ”Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position”. In: *Biological Cybernetics* 36.4 (1980), pp. 193–202.

[24] Kunihiko Fukushima. ”Neocognitron: A hierarchical neural network capable of visual pattern recognition”. In: *Neural Networks* 1.2 (1988), pp. 119–130.

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. *ImageNet Classification with Deep Convolutional Neural Networks*. 2012, pp. 1–9.

[26] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision* 115.3 (2015), pp. 211–252.

[27] Ross Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". Nov. 2013.

[28] Jonathan Long, Evan Shelhamer, and Trevor Darrell. "Fully Convolutional Networks for Semantic Segmentation ppt". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015), pp. 3431–3440.

[29] Chao Dong et al. "Learning a deep convolutional network for image super-resolution". In: *European Conference on Computer Vision (ECCV)* (2014), pp. 184–199.

[30] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *Information and Software Technology* 51.4 (Sept. 2014), pp. 769–784.

[31] Christian Szegedy et al. "Rethinking the Inception Architecture for Computer Vision". In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 2818–2826.

[32] Kaiming He et al. "Deep Residual Learning for Image Recognition". 2015.

[33] Sepp Hochreiter. "The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 06.02 (1998), pp. 107–116.

[34] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Journal of Physics A: Mathematical and Theoretical* 44.8 (Feb. 2015), p. 085201.

[35] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". Nov. 2015.

[36] Geoffrey E. Hinton et al. "Improving neural networks by preventing co-adaptation of feature detectors". 2012.

[37] Min Lin, Qiang Chen, and Shuicheng Yan. "Network In Network". 2014.

[38] Li Deng et al. "Recent advances in deep learning for speech research at Microsoft". In: *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings* (2013), pp. 8604–8608.

[39] Geoffrey E Hinton and Ruslan R Salakhutdinov. "Reducing the Dimensionality of Data with Neural Networks". In: *Science* 313.5786 (2006), pp. 504–507.

[40] Martin Riedmiller and Heinrich Braun. "A direct adaptive method for faster backpropagation learning: The RPROP algorithm". In: *IEEE International Conference on Neural Networks - Conference Proceedings* 1993-Janua (1993), pp. 586–591.

[41] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural networks for machine learning* 4.2 (2012), pp. 26–31.

[42] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12 (2011), pp. 2121–2159.

[43] Diederik P. Kingma and Jimmy Lei Ba. "Adam: a Method for Stochastic Optimization". In: *International Conference on Learning Representations 2015* (2015), pp. 1–15.

[44] Guillaume Chaslot et al. "Monte-Carlo Tree Search: A New Framework for Game AI." In: *Aiide* (2008), pp. 216–217.

[45] Levente Kocsis and Csaba Szepesvári. *Bandit Based Monte-Carlo Planning*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Berlin, Heidelberg, 2006.

[46] Bolei Zhou et al. "Learning Deep Features for Discriminative Localization". Dec. 2015.

[47] Christian Szegedy et al. "Going Deeper with Convolutions". 2014.

[48] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning TensorFlow: A system for large-scale machine learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. 2016, pp. 265–284.

[49] John M Chambers et al. *Graphical methods for data analysis*. Vol. 5. 1. Wadsworth Belmont, CA, 1983.

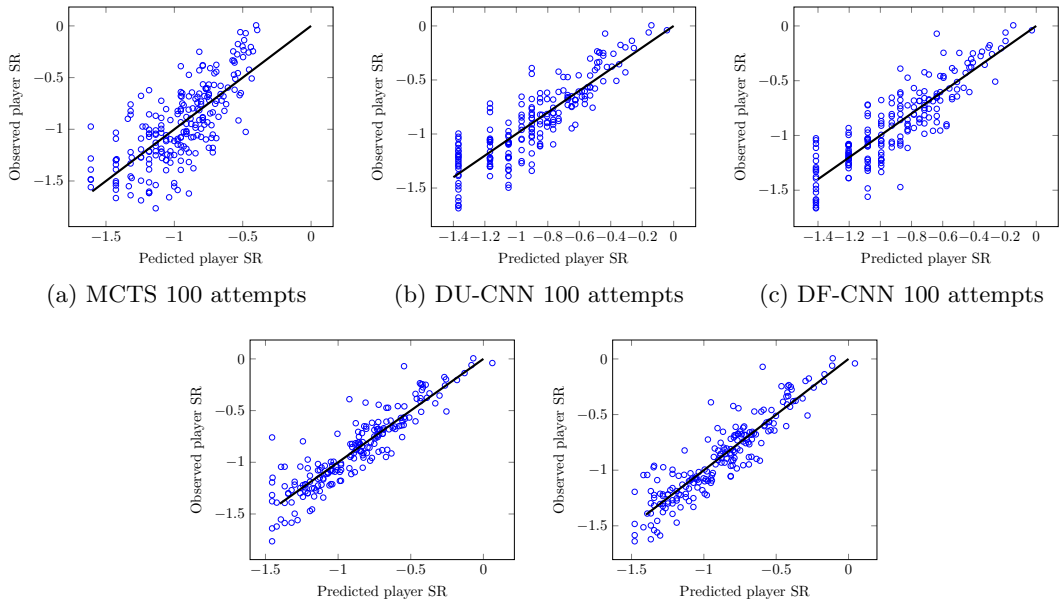
[50] M a Stephens. "EDF Statistics for Goodness of Fit and Some Comparisons". In: *Journal of the American Statistical Association* 69.347 (1974), pp. 730–737.

[51] Richard S. Sutton et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *In Advances in Neural Information Processing Systems 12* (1999), pp. 1057–1063.

Appendix A

Appendix

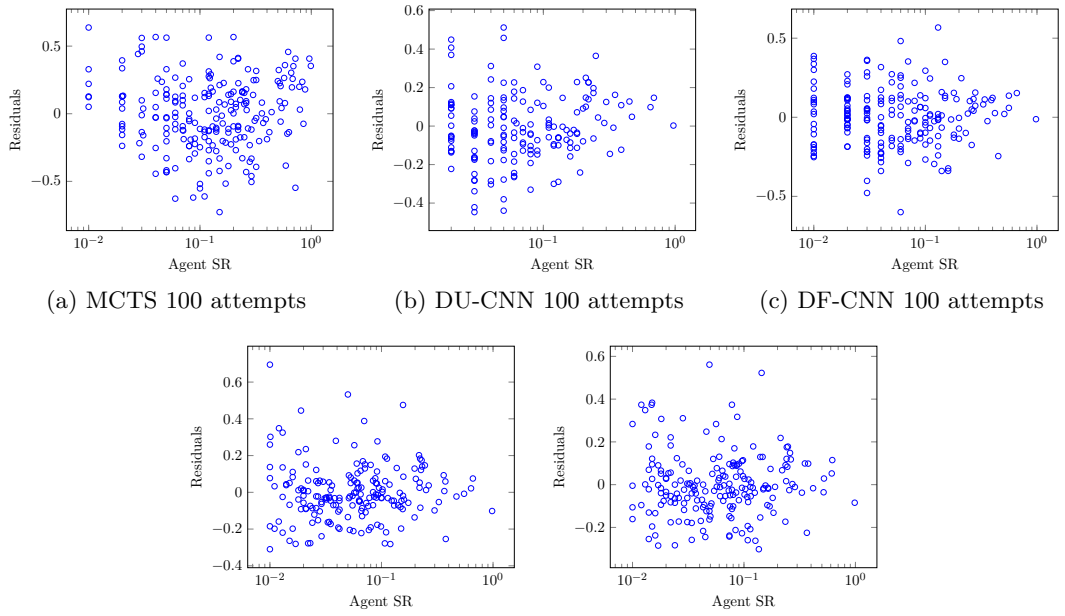
A.1 Supplement Plots for Linear Regression Models



(a) MCTS 100 attempts (b) DU-CNN 100 attempts (c) DF-CNN 100 attempts

(d) DU-CNN 1000 attempts (e) DF-CNN 1000 attempts

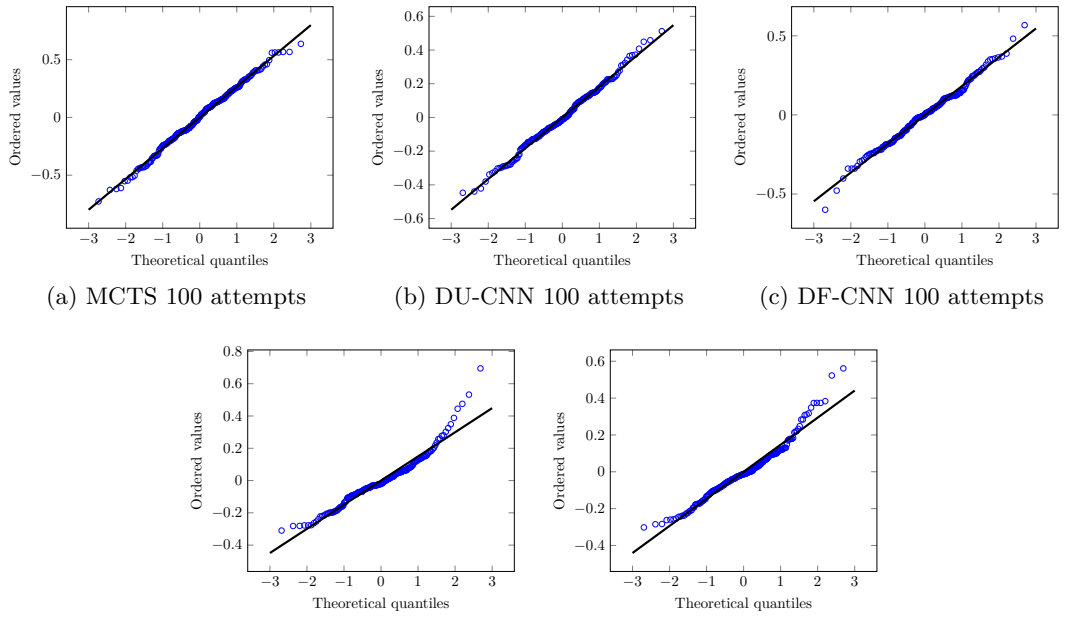
Figure A.1. Observed player SR vs. predicted player SR



(a) MCTS 100 attempts (b) DU-CNN 100 attempts (c) DF-CNN 100 attempts

(d) DU-CNN 1000 attempts (e) DF-CNN 1000 attempts

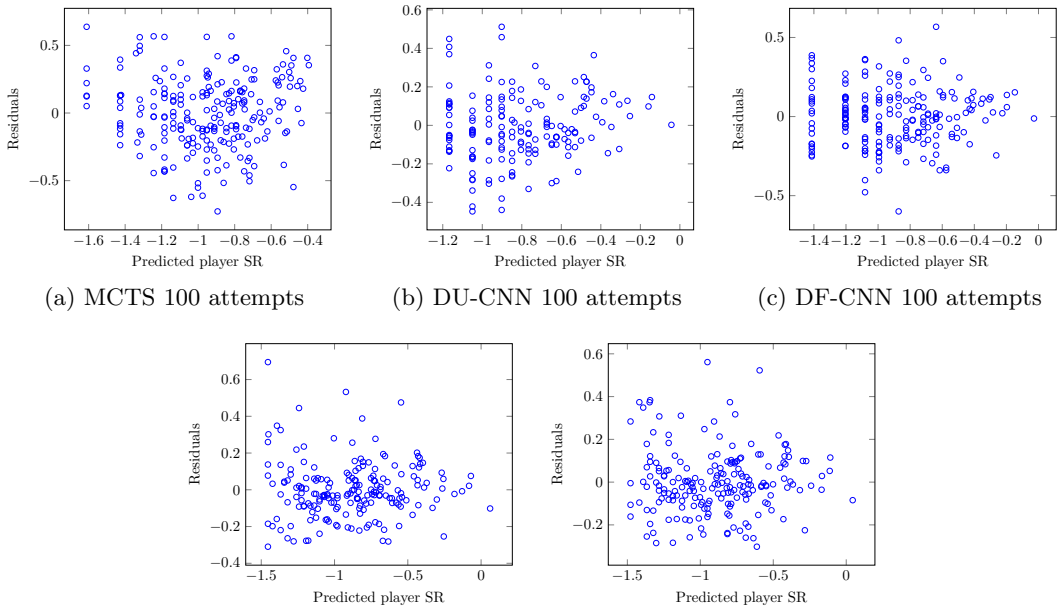
Figure A.2. Agent SR against residuals for linear regression model



(a) MCTS 100 attempts (b) DU-CNN 100 attempts (c) DF-CNN 100 attempts

(d) DU-CNN 1000 attempts (e) DF-CNN 1000 attempts

Figure A.3. Normal probability plots of residuals for linear regression model



(a) MCTS 100 attempts (b) DU-CNN 100 attempts (c) DF-CNN 100 attempts

(d) DU-CNN 1000 attempts (e) DF-CNN 1000 attempts

Figure A.4. Predicted player SR vs. residuals

A.2 Supplement Information for Neural Network Input Features

The following tables list all input features that were used as input for the CNN. Each item in the table refers to one feature layer. For more information on the features, we want to refere to publicly available resources such as www.candycrush.wikia.com.

Table A.1. Full list of input features (1/2)

Layers
ONES
LEGAL_MOVES
REGULAR_CANDY
PEPPER_CANDY
MYSTERY_CANDY
CHAMELEON_CANDY
CANDY_COLOR
CANDY_COLOR_RED
CANDY_COLOR_YELLOW
CANDY_COLOR_BLUE
CANDY_COLOR_GREEN
CANDY_COLOR_ORANGE
CANDY_COLOR_PURPLE
FISH
CANDY_STRIPED_LINE
CANDY_STRIPED_COLUMN
CANDY_WRAPPED
LUCKY_CANDY
VOID
EMPTY
LIGHT_1
LIGHT_2
CANDY_CANNON
STATIC_BLOCKER
FROSTING
LIQUORICE_LOCK
FUDGE
INGREDIENT_COLLECTOR
PORTAL
PORTAL_ENTER_POINT
PORTAL_EXIT_POINT
PORTAL_VISIBLE
PORTAL_VISIBLE_ENTER_POINT
PORTAL_VISIBLE_EXIT_POINT
LICORICE_SQUARE
MULTI_FROSTING_1
MULTI_FROSTING_2
MULTI_FROSTING_3
MULTI_FROSTING_4
MULTI_FROSTING_5
CHOCOLATE_SPAWNER
MARMELADE_LOCK
CANDY_CANNON_AMMO_CANDY
CANDY_CANNON_AMMO_INGREDIENT
CANDY_CANNON_AMMO_LIQUORICE_SQUARE
CANDY_CANNON_AMMO_PEPPER
CANDY_CANNON_AMMO_MULOCK_CANDY
CANDY_CANNON_AMMO_MYSTERY_CANDY
CAKE_BOMB
JELLY_FROG
MULOCK_1
MULOCK_2
MULOCK_3
MULOCK_4
MULOCK_5

Table A.2. Full list of input features (2/2)

Layers
COCONUT_WHEEL
INGREDIENT
EXTRA_TIME
MULOCK_KEY
CHOCOLATE_FROG
POPCORN
UFO
EVIL_SPAWNER
FROGGER_EXIT
JELLY_COLOR_GREEN
JELLY_COLOR_RED
BOBBER
CANDY_CANNON_AMMO_CHAMELEON
CANDY_CANNON_AMMO_LUCKY
CANDY_CANNON_AMMO_TIME
CONVEYOR_BELT
CONVEYOR_BELT_UP
CONVEYOR_BELT_RIGHT
CONVEYOR_BELT_DOWN
CONVEYOR_BELT_LEFT
CONVEYOR_BELT_PORTAL
CONVEYOR_BELT_PORTAL_NONE
CONVEYOR_BELT_PORTAL_RED
CONVEYOR_BELT_PORTAL_BLUE
CONVEYOR_BELT_PORTAL_GREEN
DIVINE_DROP
SUGAR_DROP
ORDER_CANDY_COLOR_RED
ORDER_CANDY_COLOR_BLUE
ORDER_CANDY_COLOR_YELLOW
ORDER_CANDY_COLOR_ORANGE
ORDER_CANDY_COLOR_PURPLE
ORDER_CANDY_COLOR_GREEN
ORDER_CANDY_WRAPPED
ORDER_CANDY_STRIPED
ORDER_CANDY_COLOR
ORDER_CANDY_FUDGE
ORDER_CANDY_FROSTING
ORDER_CANDY_POPCORN
ORDER_LIQUORICE_SQUARE
ORDER_STRIPED_STRIPED
ORDER_STRIPED_WRAPPED
ORDER_STRIPED_CANDY_COLOR
ORDER_WRAPPED_WRAPPED
ORDER_CANDY_COLOR_CANDY_COLOR
ORDER_CANDY_COLOR_WRAPPED

Declaration

I hereby certify that I have written this thesis independently and have only used the specified sources and resources indicated in the bibliography.

Stockholm, September 7, 2017

.....
Philipp Eisen

TRITA TRITA-ICT-EX-2017:149