Master Projekt

Analyse, Entwicklung und Implementierung eines Tools zur Vorhersage des Ergebnis einer Schachstellung basierend auf Deep Learning Modelle

Analysis, Design and Implementation of Tool to Predict the Outcome of a Chess Postion based on Deep Learning Models

Shanthan Rao Kanuganti

Mat.Nr.: 980409

Betreuer:

Prof. Dr. rer. nat. E.-G. Haffner

Datum:

15. November 2024

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht. Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Ort, Datum	Unterschrift

Danksagung

Ich bedanke mich bei all denjenigen, die mich während der Anfertigung dieser Arbeit unterstützt und motiviert haben. Ein ganz besonderer Dank gilt meinem Betreuer Prof. Dr. rer. nat. E.-G. Haffner. Ebenfalls möchte ich der Projektgruppe eTRainer der Hochschule Trier, sowie meinen Korrekturlesenden danken.

Abstract

This project focuses on developing a deep learning model using an MLP algorithm to classify chess positions into three possible outcomes: win, draw, or loss. Subsets of 10, 1000, and 5000 games were extracted from a dataset of 9 million chess games, converted into FEN strings, and transformed into feature vectors for input into the MLP model. The model was created using Keras and TensorFlow.

Extensive optimization was conducted, exploring activation functions, learning rates, batch sizes, and regularization methods. Advanced techniques like dropout, batch normalization, and residual connections were implemented to enhance generalization. Further exploration included grid search, NAS, and alternative architectures such as ResNet, Bayesian optimization, and MobileNet. The consistency of the MLP model was evaluated by comparing its predictions with Stockfish evaluations. This study demonstrates how deep learning may be used to predict chess outcomes, presenting a comprehensive exploration of optimization strategies and architectural enhancements.

Zusammenfassung

Die Entwicklung von Deep-Learning-Modellen war das Hauptziel dieses Projekts, das einen MLP-Algorithmus verwendet, um Schachpositionen in drei mögliche Ergebnisse zu klassifizieren: Sieg, Unentschieden oder Verlust. Teilmengen von 10, 1000 und 5000 Partien wurden aus einem Datensatz von 9 Millionen Schachpartien extrahiert, in FEN-Strings umgewandelt und in Merkmalsvektoren für die Eingabe in das MLP-Modell transformiert. Die Daten wurden in Trainings-, Validierungs- und Testsätze aufgeteilt, und das Modell wurde mit Keras und TensorFlow entworfen.

Es wurde eine umfangreiche Optimierung durchgeführt, bei der Aktivierungsfunktionen, Lernraten, Stapelgrößen und Regularisierungsmethoden untersucht wurden. Fortgeschrittene Techniken wie Dropout, Batch-Normalisierung und Restverbindungen wurden implementiert, um die Generalisierung zu verbessern. Weitere Untersuchungen umfassten Gittersuche, NAS und alternative Architekturen wie ResNet, Bayes'sche Optimierung und MobileNet. Die Vorhersagen des Modells wurden mit Stockfish-Bewertungen verglichen, um die Konsistenz des MLP-Modells zu beurteilen. Dieses Projekt unterstreicht das Potenzial von Deep Learning für die Vorhersage von Schachergebnissen und präsentiert eine umfassende Untersuchung von Optimierungsstrategien und architektonischen Verbesserungen.

Abkürzungsverzeichnis

MLP	Multi-layer Perceptron
FEN	Forsyth–Edwards Notation
NAS	Neural Architecture Search
SGD	Stochastic Gradient Descent
ResNet	Residual Networks

Inhaltsverzeichnis

1 Einleitung

Chess, known for its strategic depth and complexity, has long been a central topic in artificial intelligence research. While traditional engines like Stockfish rely on search algorithms and specialized evaluation functions, deep learning advancements have enabled neural networks to evaluate chess positions directly from data. This project explores a neural network-based approach using an MLP to classify chess positions as win, draw, or loss based solely on board features, without depending on search mechanisms.

The dataset for this study was obtained from a large collection of 9 million chess games from chess.com, with subsets of 10, 1000, and 5000 games chosen for focused analysis. Each game was processed into FEN strings, a notation system that encodes the complete board state, including piece placement, castling rights, en passant status, active color, halfmove clock, and fullmove clock. These FEN strings were converted into numerical feature vectors for input into the MLP model.

The MLP model was implemented using Keras and TensorFlow, enabling extensive experimentation with different architectures and hyperparameters. The project examined various activation functions, learning rates, batch sizes, and regularization methods. Techniques such as dropout, batch normalization, and residual connections were incorporated to improve generalization and mitigate overfitting. The exploration included grid search, NAS, and testing alternative architectures like ResNet, Bayesian optimization, and MobileNet.

A very important part of the project was the comparison of the MLP model's predictions versus the assessments of Stockfish-a chess engine of the highest class. The purpose of this comparison is to understand exactly how neural network predictions align with traditional engine evaluations and give an indication of the possibilities and constraints of deep learning for chess analysis.

In a nutshell, this project provides an in-depth analysis of neural network methodologies applied to the analysis of the positions in chess, emphasizing deep learning abilities related to the prediction of game outcomes.

2 Grundlagen

2.1 Chess Notation

Notation is a standardized technique used in chess to record the game's moves and current state. This becomes important to players, analysts, and AI systems for following a game's proceedings and making sense of any particular play in a match. Generally, there are two important types of notation in wide use, namely algebraic notation and FEN, by which games can be recorded and board configurations analyzed with accuracy.

Algebraic Notation

Nowadays, Algebraic Notation is generally accepted as the method for recording the moves of the game. In algebraic notation, each move is recorded according to the piece's type and the square towards which it is moved. It is simple, convenient, and very efficient to relate all the moves in sequence to whoever reads them. An 8x8 setup, the columns on the chessboard are lettered from a to h and the rows from 1 to 8. This system gives each square a unique identifier such as e4 or g1.

Special Move Notations:

The capture takes an x notation, for example, Nxe5. That would mean that a Knight has captured the piece on e5.A plus sign denotes a check, as in Qd8+. The notation for checkmate is #. A checkmate by the Rook would take Rd1. En passant captures are notated using e.p., as in exd6 e.p. When a pawn is promoted, that is indicated with an equals sign and the chosen piece included such as e8=Q to promote to a Queen. This system of notation allows players to study and work through games gone through by other players. It forms the basis for creating FEN strings that give a more advanced representation of the board for further analysis.

2.2 FEN Notation

FEN stands for yet another popular format in chess; this is used to represent the complete status of the board at any given time in the game. It is concise and, at the same time, detailed, hence ideal for being run by AI models and other kinds of analytical software. Each string in FEN notation is divided into six parts:

• **Piece Layout:** The disposition of every piece on the board in shortened form, using letters for the pieces, with their usual abbreviation - K for King etc. - and digits for empty squares (e.g., 3 for three empty squares).

• **Player to Move:** Shows which side should be moved; w stands for white and b for black.

• **Castling Rights:** Indicates which castlings are still feasible: k (black kingside), q (black queenside), Q (white queenside), and K (white kingside). This '-' indicates that castling is no longer possible.

• **En Passant Target:** If there is a possibility to perform an en passant capture, it records a square on which this could happen. This '-' indicates that en passant capture is no longer possible.

• **Halfmove Counter:** Determines how many moves, for the purposes of the fifty-move rule, are free of capture or pawn moves.

• **Fullmove Number:** Keeps track of total moves that have been played. This counter is incremented after each move by black.

Using FEN notation, this is what the setup of a chess game would look like:

r2qk2r/pppb1ppp/2npbn2/4p3/2B1P3/2NP1N2/PPP2PPP/R2QK2R w KQkq - 5 10

2.3 Representation of a FEN String as a Feature Vector

For any machine learning model to learn effectively from this data, the information in one FEN string must be numerically encoded. This would be done by segmenting the FEN string in many ways and encoding those into numerical features to end in some sort of structured feature vector. In fact, the FEN string captures several key aspects of the chessboard state: piece positions, whose turn it is, castling rights, en passant possibilities, and move counters. Each of these components is translated into a numerical format in the following ways:

• **Piece Placement:** Each piece is first encoded as an integer number. Positive integer numbers may serve for white pieces, negative integer numbers for black pieces. White pawns may be encoded by the number 1, and white rooks by the number 4, whereas black pawns are encoded by -1 and black rooks by -4. The empty squares are encoded by 0.

• **Active Player:** The turn indicator that makes sure the move is white's or black's is changed to a single integer feature.

• **Castling:** four binary features store whether the castling is available or not. In particular, each bit is set to 1 if the corresponding castling option is still possible and to 0 otherwise.

• **En Passant Target:** If an en passant capture is possible, this is encoded with two extra features. These features encode for the target square of the en passant, or are set to 0 if there is no en passant capture available.

• **Move Counters:** The halfmove clock, which counts the movements since the last pawn move or capture, and the fullmove number, which counts every move made during the game, are two distinct integer properties.

Resulting Feature Vector

Parsing the FEN string and encoding each element in the manner described gives a numerical feature vector 73 in length. This vector captures in one vector all of the key information within the FEN string in a format digestible by the machine learning model. For example, starting with the FEN string:

r2qk2r/pppb1ppp/2npbn2/4p3/2B1P3/2NP1N2/PPP2PPP/R2QK2R w KQkq - 5 10
The resulting feature vector is 73 elements in length:

[-4, 0, 0, -5, -6, 0, 0, -4, -1, -1, -3, 0, -1, -1, -1, -1, 0, 0, -2, -3, -2, 0, 0, 0, 0, 0, 0, -1, 0, 1, 0, 0, 0, 0, 2, 0, 0, 0, 1, 0, 2, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 4, 0, 0, 5, 0, 0, 6, 4, 0, 0, 0, 1, 1, 1, 0, 0]

It contains a 64-element vector for the board configuration representation, one integer for whose turn it is, four binary features for the castling rights, two features of en passant status, one integer for half move clock, one integer for full-move counter. This finally converts the FEN string to a regular numerical format-in this case, a vector-that is consistent so our machine learning model can take in this standardized input and learn to base its output on this encoded board state.

3 Erstellung des Modells

3.1 Loading and Splitting the Data

Proper data management perhaps forms one of the most important stages of any machine learning project, in that by default, solid model training depends only on the quality of your data, and good evaluation. Examples include feature vectors representative of chess positions, with their labels representing game outcomes. The goal is to load and partition the dataset in order to prepare the data for model training and analysis.

1. Data Loading

First, read data into memory. That typically means reading an input file with an accompanying structured target file. Feature vectors would contain numerical representations of a variety of chess positions, and labels are the results for each position. Properly aligning features with the labels is key in building a strong training set.

2. Data Splitting Strategy

Training Set (70%): This subset contains most of the data for which the machine learning model is fitted. Because most of the data is used as training data, the model can learn the underlying patterns and relationships between the input features and their corresponding labels.

Validation Set (10%):A small portion of the data is held out for the purpose of validation. During the training process, this set is utilized to help gauge the model's performance and do any necessary tuning of the hyperparameters. Overfitting occurs when a model performs incredibly well on the training data but can't generalize to new examples. The only way to really catch this problem is with a validation set.

Test Set (20%): The training and validation data are maintained completely apart from the test set. It is only utilized for the last assessment of the model's functionality following the completion of all training and modification. This guarantees that the evaluation of the model's correctness is objective and accurately reflects its capacity to manage invisible data.

3.2 MLP Model Design and Training

In this project, a neural network based on the MLP structure was created for classifying the chess positions into win, draw, and loss. There are several reasons for choosing the MLP model. First, MLP is comparatively very simple and efficient while dealing with numerical data. So, in this case, it was ideal for analyzing the feature vectors that were derived for chess board states.

1. Design of Model Architecture

Input Layer: In this MLP model, feature vectors that characterize the board state in relation to a chess position are sent into the input layer. Piece placement, player turn, casting rights, en passant availability, and move counters are all encoded in numerical data that is part of the feature vector.

Hidden Layers: The MLP model's hidden layers make use of dense, fully connected layers. An activation function and a predetermined number of neurons would be applied to each buried layer. .

Output layer: Uses the softmax activation, which returns a probability distribution over the three possible classes: win, draw, loss.

2. Compilation

The model has been compiled by the famous and efficient Adam optimizer, which works on adaptive learning. Accuracy was chosen as the metric to monitor performance while compiling the model.

3. Model Training

The training consisted of feeding the training data to the MLP model through several iterations, known as epochs. In training, the model continuously minimizes the loss function. The training process follows monitoring of a validation dataset whereby any problems such as overfitting can be identified early. This is independent validation data that provides performance of the model during training and also an indication of how well it may generalize to new data.

4. Model Evaluation

This stage is crucial because it provides an opportunity to gauge the model's performance using hidden data points—a sort of real-world metric. The accuracy metric gives a quantitative measure about the ability of the model to label the chess positions appropriately.

5. Model Saving
After training and evaluation, the trained model was saved in Keras format, which can easily be reused and deployed for different applications to subsequently analyze or integrate with a chess analysis tool.

3.3 Visualization of Training History

Visualizing the model's training history is crucial for evaluating its learning progress and understanding how well it performs on both the training and test datasets.O-O-O. Plotting accuracy and loss over epochs provides important information about how the model behaves and aids in locating problems such as underfitting or overfitting. Here, we present the analysis for datasets with different sizes (10 games, 1000 games, and 5000 games), and provide detailed explanations for each.

1. Accuracy and Loss Plots for 10 Games The initial training was conducted on a smaller subset of 10 games:

Accuracy Plot: The accuracy of the model increases quickly in the initial epochs. Even with a little dataset, the model exhibits good generalization, as evidenced by the near training and test accuracies. The test accuracy curve stabilizes without significant drops, suggesting minimal overfitting.

Loss Plot: The variations in the training and test losses are notable which decrease in the very beginning and still follow a continuously downward path throughout the rest of the epochs. The two curves being so nearly vertical with each other indicates the model very well has been able to learn the data.

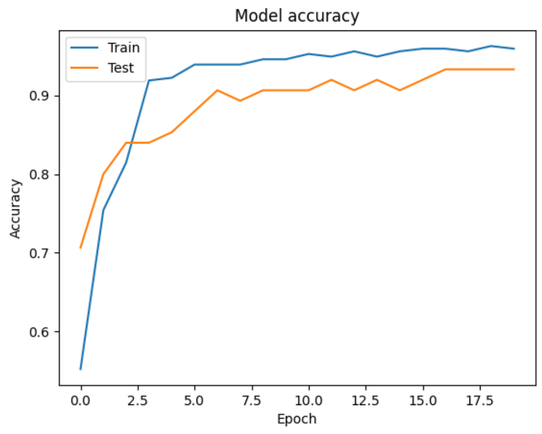

Abbildung 3.1: Accuracy vs Epoch

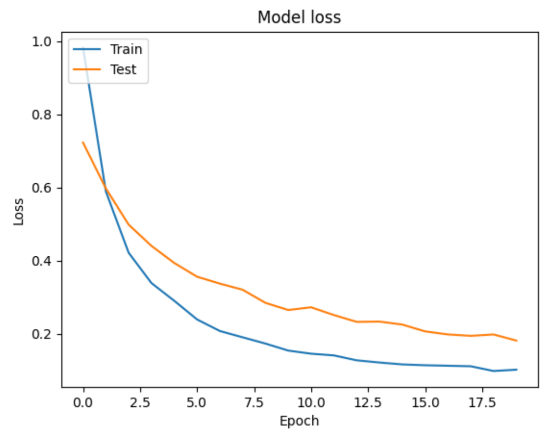

Abbildung 3.2: Loss vs Epoch

2. Accuracy and Loss Plots for 1000 Games With an increase in data size to 1000 games:

Accuracy Plot: The training accuracy improves consistently across epochs, while the test accuracy shows a gradual upward trend. The model takes longer to converge compared to the smaller dataset, but the gap between training and test accuracy remains narrow, demonstrating effective generalization.

Loss Plot: The training loss decreases steadily, while the test loss follows a similar pattern, though with minor fluctuations. These fluctuations could be attributed to the increased data complexity, but the overall trend indicates stable learning and reduced risk of overfitting.

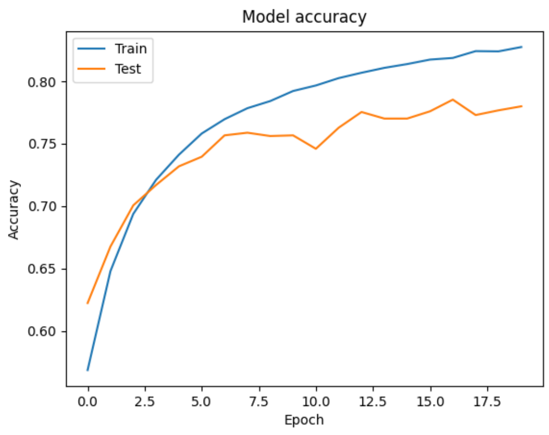

Abbildung 3.3: Accuracy vs Epoch

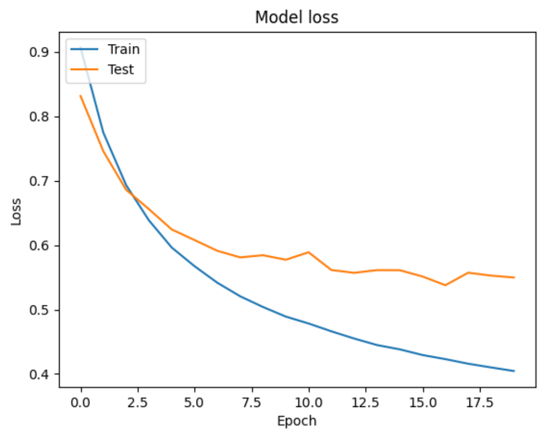

Abbildung 3.4: Loss vs Epoch

3. Accuracy and Loss Plots for 5000 Games The final training was performed using the largest subset of 5000 games:

Accuracy Plot: The accuracy plot shows a slower rise compared to the smaller datasets, which is expected due to the increased data volume. Although the training and test accuracies differ slightly, the training accuracy keeps getting better over the epochs. Given that the model matches the training data better than the test data, this discrepancy raises the probability of slight overfitting.

Loss Plot: The loss curve for both training and test datasets shows a consistent downward trend, indicating effective learning. However, the test loss stabilizes at a higher value than the training loss, further hinting at slight overfitting.In spite of this, the total loss decrease indicates that the model can pick up intricate patterns from the bigger dataset.

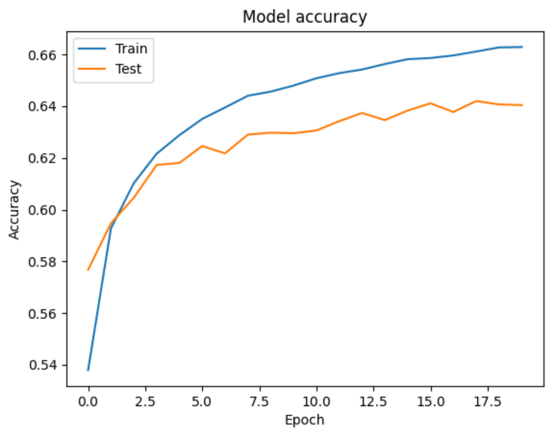

Abbildung 3.5: Accuracy vs Epoch

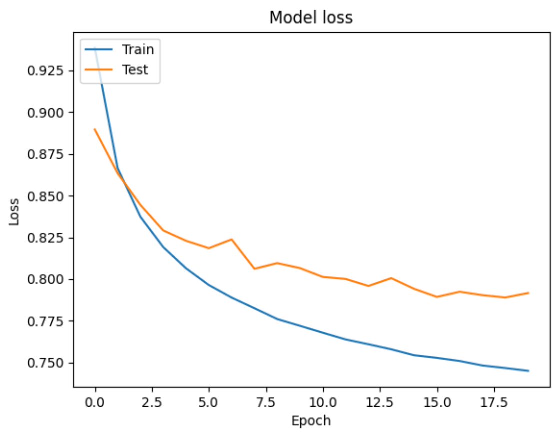

Abbildung 3.6: Loss vs Epoch

Key Observations

• The size of the dataset is what determines how much the model will learn. In the case of a small dataset (10 games), the model converges quite fast but, on the other hand, if it is not properly regularized, it may overfit. With the sizeable datasets (1000 and 5000 games), the training is a longer process, yet the model's ability to generalize better has been demonstrated as a result of decreasing data diversity.

• The unbroken lower lines in both training and test losses over the entire set of datasets mea- sure the fact that we have selected the MLP architecture and hyperparameters that are proper for the task.

• The small discrepancy seen in the performance of the 5000-game dataset implies that besides using additional regularization techniques or more training data, the model's robustness will also increase.

3.4 Using Tanh Activation Function for MLP Optimization

In this MLP model version, the tanh activation function will be used in hidden layers instead of ReLU. The tanh function is one of the common activation functions used in neural networks because it restricts the input to the range between -1 and +1 which is suitable for dealing with both positive and negative data.

Advantages of Tanh Activation

Centered Output: The tanh function shrinks the outputs down to values between -1 and 1, center the activation around zero, which helps the network learn more quickly as the data becomes more balanced.

Smooth Gradient Flow: The tanh function gives a continuous gradient all over the whole input range and hence reduces the risk of having inactive neurons as compared to ReLU while maintaining consistent updates during backpropagation.

Capturing Non-Linearities: Tanh provides a non-linear mapping which the model can utilize to recognize complicated patterns in the data; this is particularly important when the relevant features come from the vector representations of chess board states.

Handling Negative Inputs: Tanh can take on negative values which means that it is suitable for data that has features which are both positive and negative like the numerical encodings for the chess pieces.

Potential Drawbacks

Vanishing Gradient Problem: The gradient for the extreme input values with the use of the tanh function goes to zero which in turn could by some means, cause learning at the deeper layers to slow down.

Higher Computation Cost: Tanh requires more computation than ReLU, but this is generally manageable in models with fewer layers.

3.5 Utilizing Leaky ReLU Activation in MLP

This choice helps address one of the primary challenges with ReLU - the potential for neurons to become inactive.

Benefits of Leaky ReLU

Keeps Neurons Active: Traditional ReLU can result in neurons that become inactive when they receive negative inputs, as the output is zero. In contrast, Leaky ReLU applies a small slope to negative inputs, ensuring that these neurons still contribute to learning, preventing them from "dyingänd remaining idle.

Better Gradient Flow: Unlike ReLU, which can lead to the vanishing gradient problem in deeper networks, Leaky ReLU ensures that there is a small but consistent gradient even for negative inputs. This approach helps maintain a steady flow of gradients, leading to more reliable updates during training, thus improving the model's learning process.

Enhanced Learning Capability: The small slope applied to negative values allows the model to learn more effectively, especially when handling a broad range of input data, including negative features. This helps the model adjust more efficiently, improving overall performance on diverse datasets like the chess feature vectors used in this project.

Less Likely to Saturate: Unlike functions like tanh and sigmoid, Leaky ReLU does not push the activations towards extreme values, which means it avoids the issue of saturation. This leads to more stable training and avoids potential slowdowns in learning.

Drawbacks to Consider

Sensitivity to Slope Value: The performance of Leaky ReLU can depend on the chosen slope parameter. Setting the slope too high may cause overly aggressive updates, potentially impacting the model's stability. However, typical choices of this parameter generally avoid this issue.

Slight Increase in Computation: While Leaky ReLU requires a minor additional calculation compared to standard ReLU, the increase in computational load is minimal and does not significantly affect model performance.

3.6 Using SparseMax Activation Function for MLP Optimization

SparseMax provides a different approach to output activation, addressing certain limitations of softmax and introducing a sparse, interpretable output representation. SparseMax is an activation function that transforms logits (raw model outputs) into a probability distribution like softmax. However, unlike softmax, SparseMax can produce sparse outputs where some values are exactly zero. This property can be beneficial in classification tasks where a clear and confident decision is desired. It essentially ßelectsä subset of the most relevant classes, assigning zero probability to the less relevant ones.

Key Advantages of SparseMax Activation

Sparsity and Interpretability: One of the standout features of SparseMax is its ability to produce sparse output probabilities. This means that it can set some of the output probabilities to zero, effectively ïgnoring"less relevant classes. This feature helps if the model is arrogant when making predictions under ambiguous circumstances where the model does not clearly understand the outcome. Rather than adopting extreme probabilities as a possible explanation, it will expose better the case of uncertainty.

Reduced Overconfidence: Unlike softmax, which can sometimes output very high probabilities for certain classes, SparseMax tends to produce more balanced and realistic probabilities. This characteristic helps reduce overconfidence in predictions, particularly when the model is unsure about the outcome. It provides a better handling of uncertain cases by avoiding extreme probabilities.

Linear Output for Certain Regions: SparseMax outputs are piecewise linear, meaning that it provides a linear response over certain regions of the input space.. In those parts of the input space for which SparseMax response is exactly linear, one can say that SparseMax does not select a particular point among many equally suitable ones. This technique is useful in scenarios where the model's boundaries are not too complex, thus allowing the optimization process to be faster by not requiring the full non-linear mapping of softmax.

Better Gradient Flow: SparseMax has a well-defined gradient that is similar to softmax but avoids certain issues related to vanishing gradients. Because of the good gradient flow, this process has the model learn even better, especially when the model is deep and backpropagation can be tough..

Potential Limitations

Complexity of Computation: The calculation of SparseMax is somewhat more complicated than softmax, as it requires arranging the logits and much more calculations to be done to get to the threshold for sparsity. Nevertheless, with the widespread use of deep learning frameworks, the additional overheads are usually not a problem.

Suitability for All Tasks: While SparseMax can be beneficial in certain classification tasks, it may not always outperform softmax, especially when the output classes are not expected to be sparse. In tasks where all classes are likely relevant, softmax might still be the better choice.

3.7 Optimization and adjustment of hyperparameters

Hyperparameter tuning is a critical step in building an effective deep learning model. It involves selecting the best configuration for parameters that control the learning process but are not directly updated during training.

Learning Rate:

Is a key parameter that influences how quickly the model adjusts its weights during the training process:

Lower Learning Rate: This leads to smaller adjustments in the model's weights, resulting in more careful learning.

Higher Learning Rate: While it speeds up the training process by making larger adjustments, it risks overshooting the optimal point, potentially causing the model to learn in an unstable manner.

Batch Size:

Specifies the number of training samples the model uses before updating its internal parameters:

Smaller Batch Size: Using fewer samples can provide a more detailed estimate of the gradient, often helping the model to generalize better.

Larger Batch Size: This approach speeds up training by using more data per update, but it may lead the model to miss finer details in the data, potentially resulting in less optimal performance.

Depth of the Network and Number of Neurons:

The model's architecture was designed with several layers, and each layer had a set number of neurons:

Increased Network Depth: Adding more hidden layers helps the model to understand deeper insights. However, with a deeper architecture, the risk of overfitting increases, especially if the dataset is small.

Regularization Techniques:

Batch Normalization: This process stabilizes training by normalizing the output of each layer, making the model less affected by changes in weight initialization or learning rate adjustments.

Dropout: Dropout helps the model avoid becoming too dependent on specific neurons by dropping some neurons very randomly while training.

Epoch Count:

The model's processing of the complete training set is referred to as epochs:

Fewer Epochs: When we have only few epochs learning the patterns would be difficult for the model and often leads to underfitting.

More Epochs: While additional epochs give the model more opportunities to learn,but the overfitting chances are high and one more drawback is it struggles on new data.

3.8 Final Optimized Model: Enhancements and Architectural Changes

In the final iteration of this project, the MLP model was significantly enhanced to improve the classification of chess positions. The updated version integrates various advanced techniques and modifications designed to boost both the stability and accuracy of the model.

Expanded Model Architecture

The optimized model features an expanded architecture with a total of eight layers, including five hidden layers that have progressively decreasing units ranging from 1024 down to 128. For getting non-linear correlations in the data, the Swish activation function was selected for the hidden layers. Swish provides superior gradient flow over ReLU.

Enhanced Regularization:

To address the problem with the over-fitting, the model incorporates two main regularization techniques:

Dropout Layers: With dropout rates set at between 0.3 and 0.5, these layers were added to help out against over-reliance on particular neurons by randomly turning off a fraction of them during the training.

Batch Normalization: Helps to reduce the sensitivity of the model to initial weight values and accelerates the learning process, contributing to more consistent and reliable convergence.

Adaptive Learning Rate Strategy

One of the critical improvements in the optimized model was the use of an exponential decay schedule for learning rate:

Benefit: The model may generate bigger updates early in training thanks to the dynamic learning rate adjustment, which speeds up convergence. The reduced learning rate aids in fine-tuning the model's weights as training goes on, avoiding overshooting and assisting in its convergence to the ideal answer.

Introduction of Residual Connections

To enhance training stability, a residual connection was added in the third block of the network. This method creates a shortcut path by immediately adding the block's input to its output:

Benefit: The skip connection enables smoother gradient flow throughout the network, which is particularly helpful in deeper architectures. It helps the model avoid the vanishing gradient problem, allowing for more efficient training and better learning of complex patterns.

Training Optimizations with Callbacks

Early stopping, a method that tracks the validation loss and stops training when no improvement is seen over a predetermined number of epochs, was added to the model's training procedure to improve it. This prevents unnecessary training, reduces overfitting, and saves computational resources.In earlier experiments, the ReduceLROnPlateau callback was explored as an alternative approach to adjust the learning rate based on validation performance. However, the exponential decay strategy showed better results and was therefore retained for the final model configuration.

3.9 Initial and Optimized Model Comparison

Aspect	**Initial Model**	**Optimized Model**
Model Architecture	3 layers	8 layers
Hidden Layers	2 hidden layers with 64 units (ReLU)	5 hidden layers with units from 1024 to 128 (Swish)
Output Layer	Softmax activation	Swish activation with Batch Norm and Dropout layers
Activation Functions	ReLU for hidden layers	Swish for hidden layers
Output Activation	Softmax for output	Softmax for output
Regularization	No regularization applied	Dropout (0.3-0.5), Batch Normalization
Learning Rate Strategy	Fixed (0.001)	Exponential decay starting at 0.001
Residual Connections	None	Added in the third block
Optimizer	Adam with fixed learning rate	Adam with exponentially decaying learning rate
Batch Size	32	64
Epochs	20	Up to 100 (with EarlyStop-ping)
Callbacks	None	EarlyStopping and Redu-ceLROnPlateau
Accuracy	Achieved 63.8%	Improved to 83.8%
Training Time	Shorter due to fewer para-meters	Longer due to deeper and more complex architecture

Tabelle 3.1: Comparison of Initial and Optimized Model

3.10 Stockfish Evaluation Process

Stockfish Analysis

In this phase of the project, Stockfish was the one used as a benchmark chess engine to assess different chess positions. The engine was set up using the Universal Chess Interface (UCI), which allows a direct analysis of the positions represented in FEN strings. This setup was chosen because of Stockfish's very strong evaluation capabilities, giving exact information about the relative advantage of either side in a given position.

Steps for Stockfish Analysis

The review workflow went hand in hand with every step taken to ensure quality and consistency in the review process. It started with the following stages:

Data Preparation: Extracting FEN strings from the dataset, where every FEN string is a representation of a certain position of chess. The FEN string condenses all the board information: placement of pieces, color of the turn, castling rights, en passant conditions, and move counters. Preparing the position carefully so that the engine can work on it.

Engine Evaluation: For each position, Stockfish was invoked to analyze the board state with a time constraint of 0.1 seconds per move. The evaluation score provided by Stockfish offers a numerical indicator of which side (White or Black) holds an advantage:

Positive Scores: Show that White is in a good position.

Negative Scores: Show that Black is in a good position.

Mate Scores: In cases where the engine detects a forced mate, it outputs high values (e.g., 10000 for a winning sequence, -10000 for a losing sequence).

Position Validation: Each FEN string was checked using a validation function to ensure it followed the correct format. Positions failing this validation step were skipped, preventing any errors during analysis.

Error Handling: If the engine encountered any issues while evaluating a position, the evaluation was defaulted to 0 to indicate neutrality, ensuring that incomplete or problematic evaluations did not skew the comparison.

3.11 Comparison with MLP Model Predictions

Once the Stockfish evaluations were completed, they were systematically compared against the predictions made by the MLP model. The primary aim was to determine how well the neural network model approximated Stockfish's analysis, using the evaluation scores as a reference.

Consistency Check: The FEN strings used in both the Stockfish evaluation and the MLP model predictions were cross-verified to ensure they matched exactly. This step was crucial to maintain consistency and reliability in the comparison.

Evaluation Metrics: The comparison involved calculating the absolute difference between the Stockfish evaluation score and the MLP model's output for each position. The following metrics were observed:

Average Difference: Across a representative sample, the average difference was calculated as 29.4, indicating reasonable alignment between the MLP predictions and Stockfish evaluations for many positions.

Overall Comparison: When comparing the entire dataset, the average discrepancy was 178.68, suggesting that while the MLP model performs well generally, it struggles with certain complex scenarios.

Maximum Difference: The highest recorded difference was 8226, typically occurring in situations where Stockfish identified a decisive sequence (e.g., a forced mate) that the MLP model could not accurately predict.

Insights and Observations

General Alignment: For a majority of positions, the MLP model's predictions were closely aligned with Stockfish's evaluations, showcasing the model's strong predictive capabilities in capturing positional advantages.

Challenging Positions: Larger discrepancies were observed in more complex or critical positions, particularly endgames, where Stockfish's deep search algorithms provided an edge over the pattern recognition approach of the MLP model.

4 Erkundung anderer Architekturen

4.1 Grid Search

Grid Search was the systematic method for optimizing the neural network hyperparameters in this project. Using it allowed me to do an extensive search for the best hyperparameters through combinations because this technique will try all combinations. This was applied on subsets of 10, 1000, and 5000 chess games. More precisely, these very configurations allowed me to rigorously test my model in order to further improve its predictive performance.

Key Hyperparameters Adjusted

Batch size: In terms of the model's learning, this hyperparameter affects both the rate and stability of convergence. I handpicked two options: 32 and 64 samples per batch. A small batch size, 32, meant more frequent model updates. This helped to increase convergence speed but possibly at the expense of higher variability. In contrast, increasing the batch size to 64 gave the model more stable gradient updates that come at the cost of higher memory but resulted in a smoother learning curve.

Epochs: I considered two cases: 20 and 50 epochs. Fewer epochs might not give the model enough time to learn; more epochs run the danger of overfitting..

Activation Functions: An activation function introduces nonlinearity in the model, helping it to learn essential patterns.

ReLU: Has the "dying ReLU"problem, in which some neurons may become inactive during training, while being straightforward and effective.

Swish: Due to its smooth and non-monotonic curve, this activation has been explored because it can capture complex relationships from data, especially in the higher layers of neural networks.

Number of Neurons: I used two settings for the hidden layers: one with 128 neurons and one with 256 neurons. Increasing this number improves the capability of the model in learning complex patterns but, if not properly regularized, may lead to overfitting.

Choices of Optimizer: I have tested out two different optimizers to determine which one gives the model the best performance.

Adam: This adaptive optimizer automatically adjusts the learning rate while training for faster convergence. It basically had carried out a little better when most of the scenarios tested were concerned. The point is, the method implements an adaptive learning rate for every parameter. So this makes it converge much faster than the standard learning rate in stochastic gradient descent.

SGD: A simpler optimizer but its learning rate has to be carefully tuned. Most often, this converges more slowly than Adam.

Learning Rate Tuning: For subsequent groups of experiments, an exponential decaying learning rate strategy was employed. This adaptive approach starts with a larger learning rate that accelerates the early phases of training and then gradually decreases, enabling finer updates as the model approaches the optimum solution.

Regularization Techniques: In order to prevent overfitting, I used the following techniques of regularization:

Dropout Layers: Variating dropout rates of 0.3 to 0.5 were employed after every hidden layer.

Batch Normalization: Standardizing inputs to every layer by batch normalization helped in improving training stability.

Residual Connections: The skip connection was added to the third hidden layer. This immediately adds the block's input to its output, helping in mitigating the problem of vanishing gradients and enhancing the stability of training also. Training Optimizations with Callbacks:

Early stopping: This callback watched validation loss and stopped training. This avoided overfitting, apart from wasting expensive computational resources

Exponential Learning Rate Decay: Instead of using ReduceLROnPlateau, I decided to implement an exponential decay strategy with the learning rate to make smoother and quite efficient adjustments of this hyperparameter during training.

4.2 NAS

In the context of this project, Neural Architecture Search (NAS) was employed as an advanced technique to systematically explore and identify the best performing architecture for the deep learning model. Unlike manual tuning or basic optimization techniques, NAS automates the process of discovering optimal neural network structures. By this method, human intuition does not serve as the main guide in the tuning of hyperparameters such as layers configuration, neurons number, activation functions, and so on, thus human assistance-free can be used effectively for the tasks of chess game outcome prediction. The NAS procedure entailed testing and comparing different model types in order to develop a model which would not only have high predictive accuracy but would also be the most computationally efficient. The research was characterized by some concrete objectives and aimed at the following:

Changing the quantity of layers : The most prevalent model types have been created with different architectures through layer depth changes, starting from only 2-3 layers to more complicated ones with up to 5 layers.

Optimize Neuron Counts: The investigation into specifying different numbers of neurons per layer, such as 128, 256, and 320, was part of the search to establish which number of neurons is adequate for feature learning without overfitting the model.

Vary Activation Functions: The commonly used activation functions, namely ReLU and Swish, were studied. As a simple and efficient function, ReLU was opted. In addition, Swish was able to take the non-linearities effectively.

Experiment with Optimizers: NAS involved testing different optimizers, such as Adam, known for its adaptive learning rate capabilities, and SGD, which, although simpler, can give stable performance when fine-tuned correctly.

Regularization Techniques: The search also integrated methods like Dropout and Batch Normalization to assess their impact on reducing overfitting and improving generalization.

Advantages of Using NAS

The primary benefit of NAS lies in its ability to automate the design process of neural networks, making it easier to find a well-suited model architecture for a specific problem. For this project, NAS provided a structured way to:

Systematically Test Architectures: By evaluating a range of architectures, NAS reduced the guesswork involved in selecting the best model configuration.

Optimize Hyperparameters: The automated search fine-tuned hyperparameters that directly influences the model's learning process.

Save Time: While grid search and manual tuning can be exhaustive and time-consuming, NAS streamlined the process, allowing for quicker identification of optimal architectures.

4.3 ResNet

For this project, a ResNet was adopted to enhance the neural network's performance in predicting chess game outcomes. ResNet's design is distinct from traditional neural networks due to its residual connections, which help the model train effectively even when it has a deep architecture. This technique addresses challenges like the vanishing gradient problem, which often hampers deep networks.

Key Components of the ResNet Model

Residual Connections:

The hallmark of ResNet is its use of shortcut connections, where the input to a layer is added directly to its output. This connection allows the model to bypass one or more layers, effectively enabling the network to skip certain transformations if they do not provide meaningful improvements. By allowing these skip connections, the network can maintain stronger gradient flow during backpropagation, which helps avoid issues where the gradients become too small (vanishing gradient problem).

Structure of a Residual Block:

Two dense (completely connected) layers make comprise a residual block in this architecture.Prior to batch normalization and ReLU activation, a dense layer is added to the input. Then passes through another dense layer with similar processing steps. After this, the original input is combined with the output of these two layers (the "residual"connection), and the result is passed through a ReLU activation function again. This process helps the network learn both the direct features and any changes that need to be captured (residuals).

Batch Normalization

In this model, batch normalization was applied immediately after each dense layer within the residual blocks. This technique adjusts and scales the layer outputs, ensuring consistent data distribution throughout the network. It helps stabilize training by reducing the sensitivity to the initial weight values, making the learning process faster and more reliable.

ResNet Model Structure

The ResNet architecture begins with an input layer, followed by a thick layer made up of 128 neurons. That layer is batch-normalized and activated using the ReLU function. Two residual blocks follow, each containing two fully connected layers with 128 neurons. Finally, the network ends with an output layer using the softmax function, which provides the predicted probabilities for each class (win, draw, or loss).

4.4 Bayesian Optimization

Unlike traditional methods such as Grid Search, which test all possible combinations exhaustively, Bayesian Optimization takes a smarter approach. It leverages a probabilistic model to make informed guesses about which hyperparameter settings are likely to yield the best performance. This reduces the need for testing a large number of configurations, saving time and computational resources.

Key Aspects of Bayesian Optimization:

Adaptive Search Strategy:

Instead of randomly exploring the entire parameter space, Bayesian Optimization uses past performance data. This helps to predict the potential performance of different hyperparameter sets, allowing the search for finding important areas.

Hyperparameters Tuned:

The following key hyperparameters were optimized using Bayesian Optimization:

Batch Size: The search explored values between 32 and 64, balancing the trade-off between faster updates (with smaller batches) and more stable learning (with larger batches).

Learning Rate: Values ranged from 0.0001 to 0.001. This parameter controls the step size of weight updates, affecting how quickly the model learns.

Activation Function: The optimization process compared the performance of ReLU and Swish, evaluating their impact on capturing complex patterns in the data.

Dropout Rate: Values between 0.1 and 0.4 were tested to determine the optimal level of neuron deactivation for reducing overfitting.

Optimizer Type: The process included comparisons between Adam and SGD optimizers, examining their effectiveness in adjusting the model's learning rate dynamically.

Efficient Optimization Process:

Bayesian Optimization utilizes a surrogate model to predict the outcomes of various configurations before they are tested. This guided approach minimizes the number of trials needed by focusing on the parameter space that will most likely improve model performance rather than testing every combination blindly.

4.5 MobileNet

An effective and lightweight neural network design, MobileNet is perfect for usage with limited computational resources. Its design prioritizes speed and accuracy while minimizing the computational load and the number of parameters.

Key Features and Architectural Design

Optimized Convolution Operations

MobileNet employs depthwise separable convolutions, a method that divides the standard convolution process into two distinct stages.A depthwise convolution applies a distinct filter to every input channel independently in the first stage. The outputs from the first stage are then combined by applying a 1x1 convolution across all channels in a pointwise convolution. This two-step process significantly reduces computational complexity while maintaining a high level of accuracy.

Transfer Learning for Specialized Tasks

For this project, a pre-trained MobileNet model was fine-tuned using transfer learning. The base layers, originally trained on the extensive ImageNet dataset, were preserved to retain the valuable features they had learned. On top of this foundation, additional layers were added to adapt the model for classifying chess outcomes. By freezing the base layers, the model leveraged prior knowledge while focusing its training on the new task, thus requiring less data and computational power.

Global Average Pooling

To streamline the model and reduce the likelihood of overfitting, a global average pooling layer replaced traditional fully connected layers. This pooling layer condensed the spatial information from the final convolutional outputs into a single vector for each feature map, reducing the total number of parameters.

Customized Layers for Task-Specific Learning

To tailor the model for chess classification, custom dense layers were appended. These layers used ReLU activation functions for non-linearity and were followed by a softmax output layer to categorize predictions into one of three classes: win, draw, or loss.

5 Evaluation

Metric	Outcome
10 games	93.3% Accuracy
1000 games	78.1% Accuracy
5000 games	63.8% Accuracy
Tanh Activation Function	63.12% Accuracy
Leaky ReLU Activation Function	64.2% Accuracy
Sparsemax Activation Function	58.14% Accuracy
Hyperparameter Tuning	57.12% Accuracy
Final Optimized Model (5000 Games)	83.8% Accuracy
Verification of Stockfish	29.4
Comparison with MLP (Average Difference)	178.68
Comparison with MLP (Maximum Difference)	8226
Grid Search (10 Games)	94.95% Accuracy Parameters: {'batch_size': 64, 'epochs': 50, 'model_activation': 'relu', 'model_neurons': 128, 'optimizer': 'adam'}
Grid Search (1000 Games)	81.56% Accuracy Parameters: {'batch_size': 64, 'epochs': 50, 'model_activation': 'relu', 'model_neurons': 128, 'optimizer': 'adam'}
Grid Search (5000 Games)	79.56% Accuracy Parameters: {'batch_size': 64, 'epochs': 50, 'model_activation': 'swish', 'model_neurons': 128, 'optimizer': 'adam'}

NAS (10 Games)	89.33% Accuracy
NAS (1000 Games)	76.89% Accuracy
NAS (5000 Games)	70.8% Accuracy
ResNet (10 Games)	78.33% Accuracy
ResNet (1000 Games)	80.50% Accuracy
ResNet (5000 Games)	72.92% Accuracy
Bayesian Optimization (10 Games)	93.33% Accuracy Parameters: {'batch_size': 35, 'learning_rate': 0.005, 'model_activation': 'relu', 'dropout_rate': 0.14, 'optimizer': 'sgd'}
Mobilenet (10 Games)	84.95% Accuracy

Tabelle 5.1: Summary of Results

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

This project employed a deep learning methodology using a MLP model to predict the outcomes of chess positions based on numerical features derived from FEN strings. The research began with an extensive preprocessing phase, where a large dataset of chess games was curated. From this dataset, subsets of varying sizes (10, 1000, and 5000 games) were extracted to facilitate detailed experimentation and analysis.

The MLP model was developed using Keras with a TensorFlow backend. Extensive optimization was carried out, involving adjustments to key hyperparameters such as activation functions, learning rates, and regularization techniques. Techniques like dropout and batch normalization were employed to mitigate overfitting and improve generalization. This iterative process of model refinement led to significant improvements in prediction accuracy.

Beyond the basic MLP framework, the project explored more advanced neural network techniques, including the integration of residual connections and NAS. Additionally, alternative architectures like ResNet and MobileNet were investigated to benchmark performance. A noteworthy aspect of the study was the comparison of the MLP's predictions with evaluations from Stockfish, one of the leading traditional chess engines. This comparison demonstrated that neural networks have the potential to rival, and in certain scenarios, surpass traditional engines in predicting game outcomes, showcasing their ability to capture nuanced board features.

6.2 Ausblick

Building on these promising results, future research could focus on expanding the dataset to include millions of chess positions, thereby enhancing the model's generalization capabilities. Incorporating sequential data or adopting time-series approaches might enable the model to better understand the dynamics of chess games, capturing the progression of strategies over multiple moves.

Exploring newer architectures, such as Vision Transformers, or leveraging transfer learning with pretrained models could further improve prediction accuracy and model efficiency. Another potential direction is integrating the developed model into existing chess engines or real-time analysis platforms. Such integration could offer valuable insights for both competitive play and training, pushing the boundaries of AI-driven chess evaluation. This would open new avenues for developing innovative analysis tools that enhance strategic understanding and decision-making in chess.

Literaturverzeichnis

[Bis95] Bishop, Christopher M.: *Neural Networks for Pattern Recognition.* Oxford University Press, 1995. – ISBN 9780198538646

[BM02] Bradley, Paul S. ; Mangasarian, Olvi L.: Feature Selection via Concave Minimization. In: *Journal of Machine Learning Research* 3 (2002), S. 1157–1182

[BYC11] Bergstra, James ; Yamins, Daniel ; Cox, David D.: Algorithms for Hyper-Parameter Optimization. In: *Advances in Neural Information Processing Systems* (2011)

[FHT01] Friedman, Jerome ; Hastie, Trevor ; Tibshirani, Robert: *The Elements of Statistical Learning.* Springer, 2001. – ISBN 9780387848570

[GWFM13] Goodfellow, Ian J. ; Warde-Farley, David ; Mirza, Mehdi: Maxout Networks. In: *International Conference on Machine Learning* (2013)

[HD12] Hinton, Geoffrey E. ; Deng, Li: Deep Learning for Feature Extraction in Speech and Image Recognition. In: *IEEE Signal Processing Magazine* (2012), S. 82–97

[HLW16] Huang, Gao ; Liu, Zhuang ; Weinberger, Kilian Q.: Densely Connected Convolutional Networks. In: *IEEE Conference on Computer Vision and Pattern Recognition* (2016)

[HZRS17] He, Kaiming ; Zhang, Xiangyu ; Ren, Shaoqing ; Sun, Jian: Identity Mappings in Deep Residual Networks. In: *IEEE Conference on Computer Vision and Pattern Recognition* (2017)

[IS15] Ioffe, Sergey ; Szegedy, Christian: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In: *International Conference on Machine Learning* (2015)

[KB15] Kingma, Diederik P. ; Ba, Jimmy: Adam: A Method for Stochastic Optimization. In: *International Conference on Learning Representations* (2015)

[KSH12] Krizhevsky, Alex ; Sutskever, Ilya ; Hinton, Geoffrey E.: ImageNet Classification with Deep Convolutional Neural Networks. In: *Neural Information Processing Systems* (2012)

[Lar19] Larsen, Mark: FEN and PGN Formats in Chess Software. In: *Journal of Computer Chess* (2019)

[LBH15] LeCun, Yann ; Bengio, Yoshua ; Hinton, Geoffrey: Deep Learning. In: *Nature* 521 (2015), S. 436–444

[LC17] Lample, Guillaume ; Chaplot, Devendra S.: Playing FPS Games with Deep Reinforcement Learning. In: *arXiv preprint arXiv:1609.05521* (2017)

[LGG17] Lin, Tsung-Yi ; Goyal, Priya ; Girshick, Ross: Focal Loss for Dense Object Detection. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2017)

[MCM13] Michalski, Ryszard S. ; Carbonell, Jaime G. ; Mitchell, Tom M.: *Machine Learning: An Artificial Intelligence Approach*. Springer, 2013. – ISBN 9781461387603

[MKS15] Mnih, Volodymyr ; Kavukcuoglu, Koray ; Silver, David: Human-Level Control Through Deep Reinforcement Learning. In: *Nature* 518 (2015), S. 529–533

[Nar18] Narkhede, Vidhya: Overview of Bayesian Optimization Techniques. In: *Journal of Data Science* (2018)

[Nun10] Nunn, John: *Understanding Chess Endgames*. Gambit Publications, 2010. – ISBN 9781906454278

[Sha50] Shannon, Claude E.: Programming a Computer for Playing Chess. In: *Philosophical Magazine* 41 (1950), S. 256–275

[SHZ18] Sandler, Mark ; Howard, Andrew ; Zhu, Menglong: MobileNetV2: Inverted Residuals and Linear Bottlenecks. In: *IEEE Conference on Computer Vision and Pattern Recognition* (2018)

[SSS18] Silver, David ; Schrittwieser, Julian ; Simonyan, Karen: A General Reinforcement Learning Algorithm that Masters Chess, Shogi, and Go Through Self-Play. In: *Science* 362 (2018), Nr. 6419, S. 1140–1144

[TZ15] Tian, Yuandong ; Zhu, Yan: Better Computer Go Player with Neural Network and Long-Term Prediction. In: *arXiv preprint arXiv:1511.06410* (2015)

[ZVS18] Zoph, Barret ; Vasudevan, Vijay ; Shlens, Jon: Learning Transferable Architectures for Scalable Image Recognition. In: *IEEE Conference on Computer Vision and Pattern Recognition* (2018)

Abbildungsverzeichnis

Tabellenverzeichnis

Code-Auszugs-Verzeichnis

Glossar

- **C++**:
  Hardwarenahe, objektorientierte Programmiersprache.
- **HTML**:
  Hypertext Markup Language - textbasierte Auszeichnungssprache zur Strukturierung elektronischer Dokumente.
- **HTTP**:
  Hypertext Transfer Protocol - Protokoll zur Übertragung von Daten auf der Anwendungssicht über ein Rechnernetz.
- **iARS**:
  innovative Audio Response System - System mit zwei Applikationen (iARS-master-App; iARS-student-App), dass sich zum Einsetzten von e-TR-ainer-Inhalten in Vorlesungen eignet.
- **ISO**:
  Internationale Vereinigung von Normungsorganisationen.
- **JavaScript**:
  Skriptsprache zu Auswertung von Benutzerinteraktionen.
- **Konstruktor**:
  Beim Erzeugen einer Objektinstanz aufgerufene Methode zum Initialisieren von Eigenschaften.
- **MySQL**:
  Relationales Datenbankverwaltungssystem.
- **OLAT**:
  Online Learning and Training - Lernplattform für verschiedene Formen von webbasiertem Lernen.
- **OOP**:
  Objektorientierte Programmierung - Programmierparadigma, nach dem sich die Architektur eine Software an realen Objekten orientiert.
- **Open Source**:
  Software, die öffentlich von Dritten eingesehen, geändert und genutzt werden kann.
- **PHP**:
  Skriptsprache zur Erstellung von Webanwendungen.
- **Python**:
  Skript- und Programmiersprache, die unter Anderem objektorientiertes Programmieren ermöglicht.

- **Shell**:
  Shell oder auch Unix-Shell - traditionelle Benutzerschnittstelle von Unix-Betriebssystemen
- **Spyder**:
  Entwicklungsumgebung für wissenschaftliche Programmierung in der Programmiersprache Python.
- **SymPy**:
  Python-Bibliothek für symbolische Mathematik.