Machine Learning for Chess Movement Recognition

Amaury George, Albert Troussard, Pïerre-Hadrien Levieil
CS-433, EPFL, Switzerland

***Abstract*—This paper presents a novel approach to recognize chess moves from handwritten score sheets using machine learning. Overcoming data limitations, we generate synthetic samples with diverse visual elements and explore the feasibility of Tesseract and a PyTorch CNN for Optical Character Recognition (OCR). The CNN, trained on a varied dataset, achieves a competitive accuracy. Comparative analyses against other OCR engines highlight the effectiveness of our solution, demonstrating practical applications in handwritten chess move recognition.**

I. Introduction

In our project, we have been requested to implement a handwritten chess move recognition engine. The current implementation of the system can be found on chessreader.org and rely on paid cloud services for accurate optical character recognition (OCR) from handwritten score sheets. In this context, our machine learning project, detailed in this report, aims to develop an internally hosted and open-source solution for recognizing chess moves, paving the way for improved accessibility and cost-effectiveness.

The current landscape involves the utilization of OCR engines such as Tesseract, which, though powerful for printed characters, faces challenges when applied to handwritten data. Our motivation stems from the desire to overcome these limitations and contribute a robust solution tailored to the intricacies of chess notation. To this end, we explore the feasibility of Tesseract and a PyTorch Convolutional Neural Network (CNN) for OCR, emphasizing the generation of synthetic data to enhance model training.

II. Chess Notation

Chess notation is a standardized system designed to record and describe chess moves systematically. It employs a combination of letters and numbers to represent pieces and their respective destinations on the chessboard. Each square on the board is identified by a unique coordinate, with files labeled from 'a' to 'h' and ranks numbered from 1 to 8.

Moves in chess notation typically start with the letter representing the piece making the move, followed by the destination square. The piece identifiers include 'K' for King, 'Q' for Queen, 'R' for Rook, 'N' for Knight, and 'B' for Bishop. Pawns are represented without a letter. The destination square is identified by a combination of a letter and a number, indicating the file and rank, respectively.

Additional notations include 'x' for capturing, denoted between the piece identifier and the destination square (e.g., Bxe4 for a bishop capturing a piece on e4). The equals sign (=) is used to indicate pawn promotion, where a pawn reaching the eighth (or first depending on the colour) rank is promoted to another piece, typically a queen (e.g., e8=Q).

For special situations, '+' is added to indicate a move that puts the opponent's king in check, and '#' is added to signify checkmate. Understanding these characters is essential for effectively interpreting and recording chess games. For example, the move "Nf3" denotes a knight moving to the f3 square, while "e8=Q" signifies a pawn on the e-file being promoted to a queen on the eighth rank.

III. Data Generation

Since the available data of handwritten chess score sheets provided was limited with only $\sim$1000 samples available with some of not usable because they where written in French (the chess notation is different in f French) and because we did not want to over-fit the data (both the score sheets layout and the handwriting), we implemented functions to generate random samples. You can see in the image below examples of such samples.

The implemented data generation functions were designed to introduce diversity into the synthetic data set, mimicking the variations present in real-world handwritten chess score sheets. The randomness is injected across several dimensions, enhancing the model's ability to generalize.

Firstly, the most important part of randomness introduced was the generation of the actual move to recognise. We tried to manually generate moves but it was far from being exhaustive since there is more than 16'000 different chess move possible, with some of them hard to generate logically. Take for example the move "exf8=Q#" it correspond to a pawn initially in e7 moving to f8, capturing a piece in this square, promoting to queen and at the same time putting his opponent in checkmate. To be able to generate such moves, we decided to simulate more than 100'000 chess games and each time a move can be played in a party, we record it so. Doing so, we were able to compute these moves and the number of simulated games they appeared on (it's not the number of games the moves where used on but the number of games where some player could have done this move) enabling us to generate the samples in proportion.

Then, the handwriting component exhibits variability by selecting from a pool of over 350 different fonts. On top of that, the size of the text is also randomised. This ensures

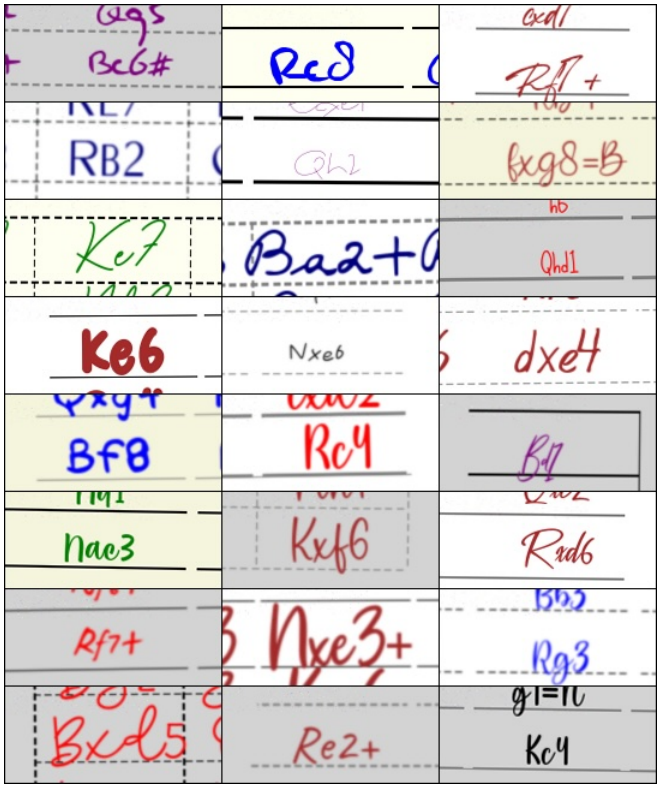

Figure 1. Example of automatically generated samples

that the model learns to recognize moves across a spectrum of writing styles.

To emulate real-world scenarios where additional information may be present around the filled-in moves, the functions introduce variability in terms of boxes and annotations surrounding the chess notation. This helps the model to discern and focus on relevant information amidst potential distractions.

Moreover, the background color, text color, line width, and type (full or dashed) are all subject to randomization. These factors contribute to the robustness of the model, ensuring it can handle diverse visual environments.

To account for the imperfections inherent in real-world data, a certain level of noise as well as small rotations is incorporated into the generated samples. This noise simulates the artifacts that might be present in images captured from a physical page, reinforcing the model's ability to handle the inherent variability in image data.

By incorporating these random features, the synthetic dataset becomes a more comprehensive representation of the challenges posed by real-world handwritten chess score sheets. This diversity is instrumental in training a robust machine learning model capable of recognizing moves across a wide spectrum of visual characteristics.

IV. Tesseract

Our strategy for enhancing the optical character recognition (OCR) model involved fine-tuning a pre-existing model to specialize in recognizing chess moves. The initial analysis, based on a referenced paper, advocated the utilization of different OCR providers, including Google Vision, ABBY Cloud and Tesseract. Developed by Google since 2006, Tesseract is an open-source OCR model primarily designed for printed characters. Despite its specialization in printed characters, we chose Tesseract due to its unique feature of allowing fine-tuning, coupled with its open-source nature and comprehensive documentation.

The training process involved using randomly generated chess moves, as detailed in the previous section. Tesseract requires ground truth data for fine-tuning, including the image, associated text, and a box file specifying character locations. Google provides various repositories on GitHub, including those containing models for languages other than English, facilitating our training using the tesstrain repository and the *make* utility.

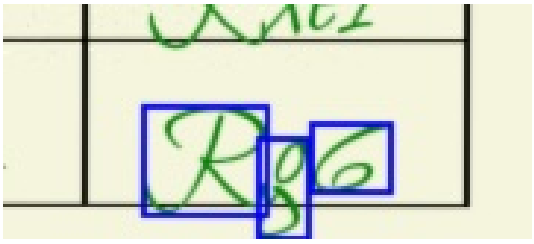

Figure 2. Example of box file visualization on the corresponding image

Despite our efforts, fine-tuning Tesseract proved to be a formidable challenge. The model yielded an error rate of 20.83% on the complete set of 100,000 generated samples, with particularly poor performance on handwritten data. Several factors contributed to these challenges. Firstly, the large size of the model necessitated a considerable amount of data and iterations for effective training, resulting in a substantial time and computational resource investment. Additionally, Tesseract is language-oriented, relying on Natural Language Processing (NLP) techniques. While we fine-tuned the English model, the inherent nature of our task rendered these language-oriented techniques ineffective. Lastly, Tesseract's design for recognizing printed characters posed a significant hurdle, as contrary to handwritten data, all printed caracters look exactly the same. Thus, training the model on different looking letters was inadequate to its pretraining. When trained on a single font, it exhibited around a 4% error rate during training and proved unusable on real handwritten data (due to over-fitting on the specific font). Attempts to address this issue, such as including a top layer in our training, yielded only marginal improvements.

In conclusion, our analysis indicates the impracticality of using Tesseract for handwritten character recognition in

our context. The model's size, specialization, and initial training data make it unsuitable for such tasks. A potential solution would involve retraining the model from scratch, demanding an extensive dataset with a diverse array of fonts, significant computational resources, and an extended training period. Subsequently, our project pivoted towards training a new model on a smaller scale, tailored to the specific requirements of our implementation.

V. Torch CNN

In parallel to trying to train a Tesseract model, we found an open-source public library *MLTU* alongside a pre-built architecture designed around character and word recognition. The library contained helper functions such as custom loss functions. The architecture of the model is a PyTorch CNN with a total of 44 layers.

The provided model architecture is a deep neural network designed for handwritten recognition, particularly suited for tasks like Optical Character Recognition (OCR). The model incorporates residual blocks, which enable the training of very deep networks while mitigating the vanishing gradient problem. The use of residual connections allows for the efficient learning of identity mappings, facilitating the flow of gradients through the network during backpropagation. Additionally, the bidirectional Long Short-Term Memory (BLSTM) layer captures both forward and backward dependencies in the sequential data, which is essential for recognizing handwritten patterns where the context of previous and future inputs is crucial. The model's architecture, with its combination of residual blocks and BLSTM layers, enables it to capture intricate spatial dependencies and temporal context, making it well-suited for the complex and variable nature of handwritten character recognition tasks. The normalization of input images within the model itself, rather than in a preprocessing step, can also contribute to the model's effectiveness in handling diverse and scaled input data. The softmax activation in the output layer allows the model to produce probability distributions over possible classes, facilitating the recognition of characters in handwritten text. Overall, the architecture's depth, residual connections, and bidirectional sequential processing make it a powerful choice for handwritten recognition tasks.

Much of the original code was preserved, although we had to change the way the data was loaded to fit to our samples. Before training, each sample went through some augmentation through randomizing the brightness, rotating and dilating the pictures. Our original training set consisted of exclusively data generated as described in section III. This lead us to a CER (Character Error Rate) of approximately 1% when testing on similarly generated data. But what we failed to initially realise was that this was due to our model largely over-fitting on the fonts used for training. That is when we started to add more fonts, and tried to mix in some of the real data we were provided with. Our best-performing model was achieved by training on a dataset comprised of 1000 samples of the data provided and 3000 samples generated by ourselves. This lead us to a 3% CER and a 8% WER (Word Error Rate). However, when building the confusion matrix we rapidly realised that the most common errors were always the same, such as but not limiting to : R/K, c/e, +/#, b/h etc. Some of these errors might be resolvable by more cautiously selecting fonts for training that would allow to differentiate the different characters. To compensate some of the frequent errors, we looked at what the second most likely prediction could be and found that more often than not, the correct prediction would be the second most likely if not the first.

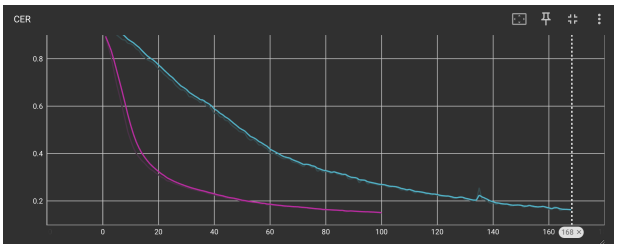

Figure 3. TensorBoard training CER as a function of Epoch, Pink curve is the first model over-fitted on the training fonts, Blue is the second, final model

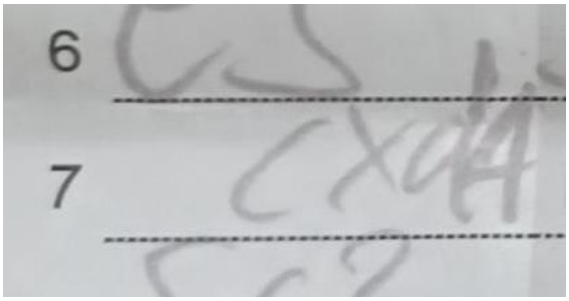

Figure 4. Example of a misclassification, True Label being "cxd4", our model predicting "xd4" with "cxd4" as second most likely

VI. Results

We were provided with the predictions of 1300 sample images of different OCR engines, as well as ChessReader's current prediction. Computing the accuracy of each prediction gives us the following table :

Model	Accuracy
Google	67.47%
Azure	52.10%
Amazon	47.38%
Abby	80.13%
ChessReader	97.15%
Our Model	93.18%

Figure 5. Accuracy of each model

As we see in the table, our model performs rather well compared to the other models. Given ChessReader's current prediction is done by taking into account the prediction of all the models, then filtering for moves suitable to the game state, our model could be a great addition to ChessReader's current algorithm. Note that this is without taking into account the additional predictions done when our algorithm is unsure. If we take these suggestions into account, the accuracy now jumps to 96.18%.

VII. Ethical Risk

In examining the ethical landscape of our machine learning project centered on chess move recognition from handwritten score sheets, our analysis focused on key stakeholders to ensure ethical considerations were appropriately addressed.

Stakeholders and Roles

Developers and Users: Developers play a crucial role in designing, training, and implementing the machine learning model. Users, encompassing chess enthusiasts and potential integrators, interact with the model for move recognition.

Machine Learning Community: Indirect stakeholders include the broader machine learning community, which may be impacted by the practices and outcomes of our project.

Risk Evaluation Process

The risk evaluation process involved a thorough literature review and analysis of potential ethical concerns associated with similar machine learning projects. The primary focus areas were:

Biases and Fairness: Given that our training data was randomly generated, without reliance on real-world data that may carry inherent biases, we mitigated the risk of introducing biases in the model. The synthetic data generation process aimed to eliminate biases by ensuring a diverse and representative set of fonts, making it able to predict anyone's handwriting. Consequently, the absence of real-world data and the random nature of data generation contribute to the project's low risk in terms of biases and fairness concerns.

Privacy: Privacy concerns related to the synthetic data generation process were addressed by ensuring no sensitive or personal information was used. The nature of chess move notations inherently limits privacy risks, as the task involves interpreting moves without delving into private information.

Transparency: The use of open-source tools, including Tesseract and MLTU, contributes to transparency by allowing scrutiny of the model's inner workings. The generation of synthetic data aimed at providing transparency in training data, reducing potential biases.

Societal Implications: Potential societal implications were considered by assessing the project's scope, ensuring it remains confined to chess-related tasks without direct societal impact.

Given these considerations, we assert that the ethical risks associated with our machine learning project are negligible. The design choices made throughout the project aimed at minimizing any potential negative impact on privacy, transparency, and fairness.

VIII. Conclusion

In conclusion, our machine learning project successfully tackled the nuanced task of recognizing chess moves from handwritten score sheets. The exploration of OCR models, including the widely-used Tesseract and a PyTorch CNN, provided valuable insights into their strengths and limitations. Despite encountering challenges with Tesseract, such as its impracticality for handwritten character recognition due to over-fitting and its emphasis on printed characters, we strategically transitioned to a PyTorch CNN.

This pivot resulted in a robust model achieving a competitive accuracy of 93.18%. Comparative analyses against other OCR engines highlighted the effectiveness of our solution, positioning it as a promising addition to ChessReader's algorithm. The iterative process of model development underscored the importance of adaptability in the face of challenges, emphasizing the need for a thoughtful approach to OCR tasks involving handwritten content. Overall, the successful implementation contributes to the evolving landscape of OCR methodologies, showcasing practical applications in the domain of handwritten chess move recognition.

References

[1] *mltu*, Machine Learning Training Utilities, https://pypi.org/project/mltu/.

[2] Tesseract repository, https://github.com/tesseract-ocr/tesseract.

[3] Tesseract Training Documentation, https://tesseract-ocr.github.io/.