How can I use machine learning to teach a computer to play Nine Men's Morris?

Matura Paper

22. October 2025

# Preface

We live in an age where artificial intelligence plays a growing role in our daily lives. From recommendation systems on social media to translation tools, and medical diagnostics, as well as chat bots, many of the services we use every day rely on AI. Neural networks are one of the most important technologies in this field. They allow computers to recognize patterns by strengthening connections between layers of neurons mimicking the human brain and thus are able to adapt to new situations, and in some cases even outperform humans. What often appears simple on the surface, like translating a sentence or recognizing a face in a photo, is actually based on large amounts of computation and huge amounts of training data.

The fast-increasing role of AI in our society and the importance of AI in our personal lives drove me to dedicate my Matura project to this topic. I wanted to gain a deeper understanding of how neural networks work, how they can be trained and used, and what their strengths and weaknesses are. I chose to apply a neural network to a game that I have played since childhood: Nine Men’s Morris. The game has an interesting structure with three different phases: placing, moving, and jumping with pieces. This multi-stage form made it especially interesting to me in the context of coding. It is also a game familiar to many people.

In addition to the neural network, I also implemented a classical algorithm, minimax with alpha-beta pruning, as a comparison. This created an interesting contrast between two approaches: one based on explicit, brute-force-like search through possible moves, and the other on adaptive learning and generalization. By setting these two side by side, I was able to learn how different methods can be used to attempt to solve the same problem, and how each one reflects a distinct way of thinking about intelligence.

I would like to thank my supervisor, …, for supporting me throughout this project, my family and friends for their encouragement, and everyone who helped me along the way.

Table of Contents

[Preface 1](#_Toc211682670)

[Introduction 4](#_Toc211682671)

[Theory 5](#_Toc211682672)

[Minimax Algorithm with alpha beta pruning 5](#_Toc211682673)

[Neural Network 5](#_Toc211682674)

[Forward Propagation 6](#_Toc211682675)

[Tensors 6](#_Toc211682676)

[Backpropagation and Training 6](#_Toc211682677)

[Regularization Techniques 7](#_Toc211682678)

[Self-Play Learning with Double DQN 7](#_Toc211682679)

[Tools and Methods 9](#_Toc211682680)

[Research 9](#_Toc211682681)

[Libraries 9](#_Toc211682682)

[Documentation 11](#_Toc211682683)

[Ruleset 11](#_Toc211682684)

[evaluate\_board 11](#_Toc211682685)

[get\_possible\_moves 11](#_Toc211682686)

[apply\_move 11](#_Toc211682687)

[Additional utilities 12](#_Toc211682688)

[Minimax Algorithm 12](#_Toc211682689)

[Neural Network 13](#_Toc211682690)

[Evaluation and playtesting 15](#_Toc211682691)

[Visualisation 16](#_Toc211682692)

[Static Game Board rendering with Tkinter 16](#_Toc211682693)

[Game Outcome Statistics using Matplotlib 16](#_Toc211682694)

[Game Animation using Pygame 16](#_Toc211682695)

[Training progress monitoring with Pygame 17](#_Toc211682696)

[Results 18](#_Toc211682697)

[Reflection 19](#_Toc211682698)

[References 20](#_Toc211682699)

[Images: 20](#_Toc211682700)

[Bibliography: 20](#_Toc211682701)

[Declaration of Originality 21](#_Toc211682702)

# Introduction

Nine Men’s Morris is a traditional two-player strategy game that dates back to the Roman Empire. The objective of the game is to reduce the opponent to fewer than three pieces or block all of their possible moves. Each player begins with nine pieces, which are placed alternately on specific points of a board consisting of three concentric squares connected by lines. The game unfolds in three distinct phases: placement, movement, and when a player has only three pieces remaining, jumping. Forming a straight line of three pieces, known as a mill, allows a player to remove one of the opponent’s pieces. In this project, I implemented a simplified version of the game without the final jumping phase. This decision was made to reduce the complexity of the action space for the neural network and to make the game structure more consistent for testing.

I chose Nine Men’s Morris because it has just the right balance between simplicity and depth. The board’s small size keeps things manageable for computation, which is especially necessary for the minimax algorithm, but the game’s different phases and strategic choices still make it complex enough to be a real challenge for both people and AI. This mix let me create a project that was doable from a technical standpoint while still being interesting to code.

I chose Python as the programming language for the project because it’s readable, flexible, and has access to powerful libraries for both artificial intelligence and visualization. Python’s large number of libraries, especially PyTorch, Pygame, Tkinter, and Matplotlib made it possible to implement complex ideas easily while keeping the code understandable. Although Python is not the fastest language computationally, its integration with low-level libraries written in C and C++ allows most computationally heavy operations, such as tensor manipulation and matrix multiplication, to still run efficiently. Pythons many advantages allowed me to focus on designing, testing, and analysing my results rather than spending an excessive amount of time on low-level optimizations.

# Theory

## Minimax Algorithm with alpha beta pruning

The minimax algorithm is a decision-making method mainly used in two-player games such as Nine Men’s Morris. It tries to find the best possible move by first assuming that both players play optimally. The algorithm then builds a game tree, where each node represents a possible board state and each branch represents a move. It alternates between maximizing the player’s advantage and minimizing the opponent’s, based on a heuristic evaluation of the board.

At the bottom of the tree (either at the end of the game or at a chosen search depth), a heuristic function (see *evaluate\_board* in the documentation) assigns a score. The maximizing player chooses the move with the highest score and assumes the opponent will pick the move with the lowest (see Figure 2). These scores are then passed back up the tree to decide the best move for the root position.

Alpha-beta pruning improves the minimax algorithm by reducing how many nodes it needs to evaluate. It keeps track of two values: alpha, the best score the maximizing player can guarantee, and beta, the best score the minimizing player can guarantee. When a branch cannot change the outcome, for example, if a maximizing move already exceeds the minimizer’s beta, the algorithm stops exploring other moves on that branch (e.g. the crossed-out branches in Figure 2). This greatly reduces computation while keeping the same result, which is especially helpful in games such as Nine Men’s Morris, where each position can lead to upwards of 20 moves.

## Neural Network

Neural networks are computational models based on simplified ideas about how the human brain processes information. Just as the brain is made up of interconnected neurons, a neural network consists of artificial neurons arranged in layers: an input layer, one or more hidden layers, and an output layer (see Figure 3).

Information flows through these layers in a way that loosely resembles how electrical impulses travel through the axons and dendrites of neurons in the brain. Each artificial neuron combines incoming signals, adjusts them using weights and biases (like how synapses strengthen or weaken connections), and applies an activation function to decide whether to “fire” them. In a feedforward neural network like the one used in my project, data flows in only one direction: from input to output, unlike in the human brain, where signals can travel both ways.

### Forward Propagation

During forward propagation, the network processes input data to produce a prediction. In this case, the input is a 28-dimensional vector with the first 24 values describing the board state, each showing whether the space is occupied by player 1, player 2 or none, while the last 4 represent each player’s piece counts.

The data passes sequentially through the network’s layers. At each layer, the following computation is performed:

output = activation(*W* · input + *b*).

where represents the weights, the bias, and the activation function introduces non-linearity. This process continues until the final layer produces the output values. Here they are so-called logits that indicate the probabilities of all possible actions.

### Tensors

All computations within a neural network are carried out using tensors, which are multi-dimensional arrays. A scalar is a 0-dimensional (0D) tensor, a vector a 1-dimensional (1D) tensor, and a matrix 2-dimensional (2D) tensor. This can be extended to any number of dimensions.  
In this network, the board state is a 1D tensor of length 28, while a batch of these states forms a 2D tensor with dimensions (batch\_size × 28). The network’s weights, biases, and activations are also expressed as tensors, which allows large-scale mathematical operations such as matrix multiplication, bias addition, and the application of activation functions to work more efficiently and in parallel.

A major advantage of PyTorch is its ability to perform automatic differentiation on tensors. This means that the framework records every operation applied to a tensor and can automatically compute the gradients required for training. This makes model optimization and application much simpler and faster, without the need for manual derivative calculations, as Pytorch does all of that via integrated functions.

### Backpropagation and Training

Learning occurs by comparing the network’s prediction to the correct target using a loss function. I had the most success using CrossEntropyLoss, which measures how far the predicted probability is from the true label. The loss can be written as

where represents the true label (1 for the correct guess and 0 for others) and is the predicted probability.

The network then applies backpropagation, which uses the chain rule to compute how each weight contributed to the error. These gradients are then used to update the weights in a way that reduces the loss. Fortunately, all of these steps are efficiently handled by PyTorch’s built-in functions.

I use AdamW to perform the updates of the weights. AdamW combines the advantages of momentum, which helps the model converge faster, with weight decay, a form of regularization that prevents weights from becoming too large and helps the model generalize better to new data.

### Regularization Techniques

Neural networks can easily overfit, meaning they learn to memorize the training data instead of the underlying patterns. When this happens, the model performs well on known examples but poorly on new, unseen ones. To prevent overfitting and improve generalization, two regularization techniques were applied.

Dropout randomly disables a fraction of neurons during training (in my case with p = 0.3, each neuron has a 30% chance of being ignored). This prevents the network from becoming overly dependent on specific pathways or neurons, encouraging it to learn redundant and more robust representations of the data.

Layer Normalization (LayerNorm) stabilizes training by normalizing the activations across features within each layer. By keeping activations within a consistent range, LayerNorm improves gradient flow and helps the network converge more smoothly, especially in larger architectures. Together, these techniques stabilize the model during training and make it more reliable during evaluation.

### Self-Play Learning with Double DQN

In addition to dataset-based training, I used self-play to train the neural network, meaning the network plays games against itself. This approach allows the network to generate its own training data which is stored in a so-called Experience replay: each game it plays provides new examples of good and bad moves, which it can use to gradually improve. Over time, the agent learns from its own successes and mistakes, refining its strategy in a way that is similar to how people learn by practice.

An important part of this setup is the Double Deep Q-Network (Double DQN). In standard reinforcement learning with a regular Deep Q-Network, the same network is used to both decide the best action and estimate the value of that action. This can lead the network to be overly optimistic, overestimating the potential of certain moves.

Double DQN solves this by splitting these tasks between two networks: one network chooses the best move (the “online” network), and a separate network evaluates that move (the “target” network). By separating decision-making from evaluation, the agent becomes much better at accurately assessing the quality of its actions, leading to more stable and effective learning.

After each self-play game, all recorded moves are assigned rewards based on the outcome. Every stored move includes the game state (board configuration, phase, and current player), the chosen action, and the resulting reward, next state, and termination flag. When assigning rewards, the network determines which player made each move and labels it as +1 for moves made by the eventual winner, –1 for the loser, and 0 for draws. The model is trained using an off-policy Q-learning approach[[1]](#footnote-1): it predicts the value of each move and updates toward a target value, defined as

where is the assigned reward and the next state. Because the state representation includes whose turn it is, the same board pattern can yield different values depending on which player moves, allowing the network to learn effective strategies for both sides simultaneously.

This combination of self-play and Double DQN has been fundamental in training some of the most notable game-playing AI systems, such as AlphaGo and AlphaZero. In those systems, the AI started by playing random games against itself, then gradually learned which moves were strong through repeated practice. By constantly playing against its own improving strategy, the AI discovered strategies that human players had never considered.

Self-play encourages a balance between exploration, so trying new moves to see what works, and exploitation, using moves the agent already knows are strong. Over many iterations, this process allows the model to develop increasingly sophisticated strategies, just as a human player would by studying and practicing countless games.

# Tools and Methods

## Research

In preparation for implementing neural networks and my minimax algorithm, I used a range of external resources. I had already gained some experience with neural networks during my third-year gymnasium project on sketch identification, which provided a basic understanding of the underlying principles. To refresh my knowledge and deepen my understanding, I relied on 3Blue1Brown’s neural network series [1], discussions on Stack Overflow [2], and primarily the official PyTorch documentation [3]. For the minimax algorithm, I used resources from MIT, specifically Lecture 6 of their 2010 artificial intelligence course [4], which offered a detailed theoretical overview. Throughout development, I only used AI tools like ChatGPT [5] to clarify concepts and assist with debugging, rather than for generating solutions. For the dataset based neural network, I modified a preexisting dataset [13] one by replacing the suggested best moves with my own ones.

## Libraries

One of the main reasons for using Python as the language for this project is the fact that I was able to use a lot of libraries that run in faster programming languages, whilst keeping the simplicity of the Python environment. This meant that I was able to focus more on my goals in this project instead of having to start from scratch. The following libraries were used:

PyGame [6] provided the interface for visualizing the game board and managing user interactions. It handled the display window, the rendering of game states and real-time updates, which made it possible to compare both the neural networks and the minimax algorithm.

Ast [7] was used to safely evaluate string representations of data and convert them back into Python objects. I used it to read and restore stored game states.

Tkinter [8] was used as a simpler alternative to PyGame for displaying visual outputs, such as quick status windows or debug visualizations, especially during testing phases where full graphical rendering wasn’t needed.

Matplotlib [9] is used to show training results and statistical data in the shape of graphs.

Math [10] provides basic mathematical functions such as *sqrt()* and *log()*, and most importantly *inf()* as a so-called sentinel value to initialize variables before searching for their real value.

PyTorch [3] provides the core tools for both training approaches. For the dataset-based model, I use *torch.nn* modules for the the feedforward architecture, *torch.nn.functional* for softmax and cross-entropy loss, and *torch.optim.AdamW* for optimization. *torch.utils.data* handles batching of minimax-generated state–action pairs, while autograd is used for gradient computation. In the self-play setup, PyTorch enables the parallel use of policy and reference networks, efficient tensor operations for experience replay and discounted reward calculations.

Random [11] is used to introduce controlled stochasticity into the system, such as random move selection during exploration phases and data shuffling before training.

Copy [12] provides deep and shallow copy utilities, allowing safe duplication of mutable game objects. This is used for simulating recursive game trees in the minimax algorithm without altering the original board state.

# Documentation

## Ruleset

I designed the functions in rules.py to support gameplay of a simplified version of the game and also help the training and testing of the neural network (NN). This module provided the core rules, state management, and utilities. By keeping this part of the game logic modular, I could integrate it into both my minimax algorithm for AI opponents and the NN for policy-based move selection. The board is represented as a 7x7 grid (list of lists), where valid positions can hold 0 (empty), 1 (Player 1), or 2 (Player 2), and invalid spots are set to -1 and ignored in logic. I omitted the "flying" phase (when a player has 3 pieces near the end of a game) to ease NN training, as it reduces complexity in action spaces and state representations. The rules.py functions are used in most other programs, including the main game loop (e.g., in NNEvaluation.py) for testing NN vs. minimax, human, or random players, multi-match simulations, and post-game animation, as well as training of the NN.

The module also includes global data like connections (adjacencies for moves), mill\_lines (lines for mill detection) and VALID\_POSITIONS (24 playable spots), which are crucial for all functions to work.

### evaluate\_board

I created *evaluate\_board* as a simple heuristic way for the minimax algorithm to assess board states during search. It calculates the difference in piece counts between the player and opponent, returning a positive score if the player has more pieces. This is rudimentary and only used in minimax for quick value estimation and is not part of the NN, as the NN learns from data without explicit heuristics. For example, on a board with 5 pieces for Player 1 and 4 for Player 2, it returns +1 for Player 1.

I tried enhancing this function by rewarding mills, partial mills, and player mobility using the respective functions, but after thorough testing, I’ve discovered that these changes only extend processing time without providing any significant improvement in performance.

### get\_possible\_moves

I designed this function to generate legal moves for a player based on the game phase, determined by move\_count, which I track externally in the game loop. For the placement phase (move\_count < 18), it returns all empty valid positions as tuples (eg. 0, 0). After the placement phase, it instead looks at every position where the player already has a piece and finds all adjacent empty spots. These possible moves are returned as pairs in lists, for example [(3, 0), (3, 1)]. This is used in gameplay (e.g. for validating human inputs) as well as in NN training and usage (masking out invalid actions) and in the minimax algorithm. So, in the early game, *get\_possible\_moves* might return [(0, 0), (0, 3), ...] and later, after the placing phase is finished [[(0, 0), (0, 3)], ...].

### apply\_move

This function applies a move (placement or movement) to the given game board matrix and handles mill formation, returning a tuple (success, board\_or\_message) for error handling in the game loop (e.g., invalid spot). For placements (tuples), it sets the cell to the player and checks for mills via is\_mill, triggering piece removal if needed. For movements (lists), it swaps pieces and does the same. I randomized piece removal to shrink the NN's action space (no need to predict removal targets) and level the playing field vs. minimax, which could otherwise optimize removals. The minimax itself uses a deterministic subfunction for the the game tree simulation to work consistently. For example, in a match, apply\_move(board, (0, 0), 1) would place Player 1 at (0,0) if that board position is empty.

### Additional utilities

I designed several utility functions to support both gameplay and neural network integration. *check\_game\_over* determines game termination after the placement phase (move\_count > 18) by checking if either player has fewer than 3 pieces, enabling the main loop to detect wins/losses, with draws handled externally via repetition checks. *get\_actions* generates the complete action space (placements as tuples, movements as lists) for the neural network’s policy output. *board\_to\_key* converts the board to a string key (e.g., ‘(0,-1,-1,0,...)’) for storing game history, enabling easier post-game animation and repetition detection. *count\_pieces* returns a tuple of piece counts (count1, count2) for NN input features, which seems to improve its accuracy.

## Minimax Algorithm

I implemented the minimax algorithm with alpha-beta pruning to be able to evaluate the neural network’s performance and the accuracy of the training dataset. The function uses the following parameters: the current board (a 7×7 grid from rules.py), the search depth (default 2 in NNEvaluation.py for faster testing), the current player (1 or 2), a Boolean flag is\_maximizing (True for the player’s turn, False for the opponent’s), the move\_count (to track the game phase), and the alpha/beta bounds used for pruning.

The opponent is set, and best\_score is initialized to –∞ for the maximizing side and +∞ for the minimizing side.The best\_move and best\_move\_chain are set to None and empty, respectively. The algorithm then iterates over all possible moves returned by *rules.get\_possible\_moves*. The function uses move\_count to determine whether the game is in the placement phase (<18 moves) or the movement phase. When the depth reaches zero, the function evaluates the board state using *rules.evaluate\_board*.

For each possible move, a deep copy of the board with that move applied is created. The move is applied using *rules.apply\_move*, which automatically handles piece removals after mills are formed. The algorithm then recursively calls itself on the deep copy of the board with a decremented depth, incremented move\_count, and the is\_maximizing flag swapped to switch turns. The alpha and beta values are updated and sent through the recursion, allowing unpromising branches to be pruned, saving computational time.

The minimax returns a tuple (best\_score, best\_move\_chain, best\_move). best\_move is used in NNEvaluation.py to determine the next move, while best\_move\_chain, a sequence of evaluated moves such as [(0, 0), [(3, 0), (3, 1)], ...]—is mainly used for debugging the minimax itself.

I limited the default search depth to 3 in NNEvaluation.py because deeper searches become slow due to the large branching factor of up to 24 placement options early in the game and multiple movement options later. The simple piece-count heuristic seems to work best, as mill-based evaluations (e.g., count\_mill or count\_potential\_mills) often encouraged short-term strategies that weakened overall play. With alpha-beta pruning fully implemented, the algorithm cuts off branches that cannot improve the final decision.

## Neural Network

When I first set out to build a neural network, my idea was to take a traditional, dataset-driven approach. I designed a feedforward network trained with supervised learning, where the goal was to minimize cross-entropy loss. The input to the network was a flattened board state combined with the piece counts for each player, and the output consisted of 88 logits corresponding to all possible actions from rules.get\_actions(). After applying a softmax, these logits turned into action probabilities, essentially letting the network “choose” a move.

The architecture I settled on was pretty straightforward: three hidden layers with sizes stepping down from 512 to 256 to 128 units. Each layer was followed by layer normalization, GELU activation, and a dropout layer with a rate of 0.3. Dropout was necessary to keep the network from memorizing the dataset and to get it to generalize better.

For training, I relied on AdamW as my optimizer, with a learning rate of 1e-4 and weight decay of 1e-5. Cross-entropy loss was the natural choice for the loss function. I tracked top-1 accuracy alongside it to get a clear picture of whether the model was actually predicting the right moves. The dataset itself was built from minimax-generated state–action pairs, which gave me a ready-made source of “optimal” moves to imitate. I even built a small monitoring system with Pygame to visualize training progress, showing the loss, accuracy, and board states as the model learned.

This approach worked up to a point. The network learned to imitate the minimax decisions contained in the dataset reasonably well, but it became obvious that performance was capped by the dataset itself. The model could only be as good as the data it was fed, and minimax-generated moves seemed to not always be optimal. Worse, the supervised setup didn’t adapt as well as I had hoped: when the model played actual full games, it struggled. This was visible when comparing the model’s performance on a test set and in a real game, where it seemed decent against the test set but worse in actual play.

This motivated me to try self-play. Instead of using a fixed dataset, I let the AI generate its own training data by playing against itself. After each 100 matches, it would stop playing and analyse which moves had led to a win or loss and adjust its strategy accordingly, similar to a real player reviewing his past games. This was done using an Experience buffer holding the last 500,000 moves played, to prevent overfitting to only recent experiences. This created a feedback loop where the AI was both the teacher and the student, constantly forced to deal with its own strategies. Early on, the AI experimented with all sorts of random strategies. Over time, it shifted toward exploiting the approaches that worked best, without ever fully discarding the possibility of trying new ideas.

In practice, this self-play approach completely outperformed the dataset-based one. It removed the ceiling imposed by precomputed data and gave the network room to invent its own strategies. It also became far more adaptable. Since it was always playing against itself, it was constantly forced to respond to unfamiliar situations and strategies, which made it much more resilient in actual games. Interestingly the self-play based network performed very poorly on the dataset I used to train the other.

# Evaluation and playtesting

The performance of the neural network was evaluated using several complementary methods designed to assess both its predictive accuracy and practical gameplay strength.

The first stage of testing used the traditional approach of dataset-based evaluation. Out of a total of 525,000 recorded game states, 500,000 were used for training and 25,000 were reserved for testing. The metric used in this phase was the accuracy of the network’s top prediction matching the correct move from the dataset. While this provided an initial estimate of how well the model learned the data distribution, it quickly became clear that dataset accuracy alone was not a reliable indicator of actual playing strength.

To combat this and gain a more realistic understanding of in-game performance, I implemented a second evaluation method in which the neural network played directly against a range of opponents. These included a random move generator, a low-depth minimax algorithm, and the user. The outcomes were recorded in a win/loss/draw format, with thousands of games typically played against the random and minimax opponents to ensure statistically accurate results. The “play against user” mode served mainly as a qualitative feature, providing an intuitive sense of how the neural network’s decisions compared to human reasoning.

I implemented a third testing framework in a separate program called EvalEpochs to compare the performance of different training checkpoints (see Figures 5 and 6). Since each training session saved the model state every few epochs, EvalEpochs allowed automated testing of these saved versions against both the randomizer and minimax opponents, as well as on the test set. For each epoch, the program generated graphs displaying the win, loss, and draw rates across the different opponent types.

Interestingly, results from this evaluation revealed a strong divergence between dataset accuracy and real gameplay strength. The self-play-trained network achieved only random-like accuracy when evaluated on the dataset but performed significantly better in actual games (see Figures 5 and 6), consistently defeating the randomizer and achieving decent results against the minimax algorithm at lower depths. This confirmed that the network had learned general gameplay principles rather than memorizing dataset-specific patterns and showed me once again that the dataset was flawed in some way.

Across experiments, EvalEpochs proved to be the most informative evaluation method, as it integrated both dataset testing and live playtesting. It also revealed that the self-play model’s performance peaked after a certain number of epochs, with diminishing returns and even mild regression beyond that point.

# Visualisation

Most of the processes inside a neural network are completely invisible to the user. The flow of data through layers, the adjustment of weights, and the formation of internal representations all happen within high-dimensional spaces that are hard to understand conceptually. This makes visualization a critical tool in understanding what a network actually does, both for debugging purposes and for building intuition about its decision-making. As Cynthia Rudin notes in her article; “Many of these algorithms are black boxes […] either because they’re proprietary or because they’re too complicated for a human to understand.”[[2]](#footnote-2) Visualization techniques provide a way of opening that box, offering glimpses into the abstract structures the model builds while learning.

To facilitate this, I developed several different programs that render the network’s behavior in more concrete ways. Some of these tools focus on tracking loss curves and weight distributions over time, while others attempt to illustrate the network’s reasoning by highlighting which features or inputs have the strongest influence on its predictions. By experimenting with multiple approaches, I aimed to not only verify that the network was learning in a stable way but also to better understand how it was developing strategies during training.

## Static Game Board rendering with Tkinter

In display.py, the *display\_board* function takes an array which represents the current board state as an input and creates a window displaying it using the library Tkinter. This works by assigning a colour to each integer using a dictionary. These colors are then shown in a 7x7 grid. The function is used in the NNEvaluation function to allow a human player to play against any of the other opponents. The player can directly input the move they choose to make in the display. Upon entering a move and confirming it by pressing enter, the move is then returned as an output to be used in the simulation.

## Game Outcome Statistics using Matplotlib

The *graph\_wins* function in display.py uses Matplotlib to create a pie chart summarizing multi-game outcomes. This was the main function I used to determine the accuracy of my neural network, with its simple design allowing the user to assess the fitness of a network at a glance. Using TkAgg (forces matplolib to use the Tkinter backend) allowed me to easily modify or save the resulting graph. I also used matplotlib to show the neural network’s progress across epochs by graphing it’s win rate against the randomizer every couple of epochs in a bar chart (see Figures 5 and 6).

## Game Animation using Pygame

To be able to gain more insight into why my neural net was winning or losing, I chose to program a function which used pygame to convert a list of board states which where continuously saved in NNEvaluation.py as board\_history into an animation. This animation can be viewed in a pygame window and can be paused, sped up, slowed down or restarted using keyboard inputs.

## Training progress monitoring with Pygame

To monitor the training of my neural network I added a training monitor which graphed the accuracy as well as the training loss every few steps to show me whether the accuracy of the network was actually improving with time. Tracking both metrics was important, since accuracy alone does not always reveal how well the model is learning as loss can continue to decrease even when accuracy temporarily stalls. By observing the two curves together, I could identify whether the model was genuinely improving, overfitting to the training data, or failing to capture the underlying patterns.

The visualization also served as a debugging tool: sharp spikes in loss often indicated a learning rate that was too high, while a very flat curve suggested that training was proceeding too slowly. I was also able to compare training and validation data side by side to see whether the network’s performance was generalizing beyond the training set. Overall, the training monitor gave me a much clearer picture of the learning process than numerical logs alone could provide.

# Results

Over the course of the project, the neural network was evaluated extensively through automated playtesting and statistical monitoring. For each training epoch, approximately 4,000 games were played between the neural network and various opponents, including the random move generator and the low-depth minimax algorithm.

The supervised, dataset-based network achieved a limited level of performance. It reproduced minimax-like decisions on familiar board states but quickly reached a plateau. Its accuracy on the dataset did not translate into meaningful success during live play, indicating that it had mainly learned to imitate patterns rather than develop general strategies.

The self-play network, on the other hand, showed much clearer improvement. In early epochs, it performed only marginally better than random, but as training progressed, it began to consistently outperform the randomizer. Around the 2,600th epoch, the network reached its peak, achieving a win rate of roughly 90 - 95% against the randomizer (see Figure 5) and occasionally managing to defeat the minimax algorithm at low depths. Beyond this point, performance started to fluctuate, suggesting that the model had reached a point of diminishing returns where additional training introduced instability rather than steady progress.

These results highlight the difference between memorization and adaptation. The dataset-trained model was strong within the boundaries of its training data but failed to generalize beyond it. The self-play network, by contrast, demonstrated genuine learning ability. Despite this improvement, the neural network still remained weaker than both the minimax algorithm and human players, who performed comparably to a minimax depth of around five. This confirmed that while the self-play method produced meaningful learning, it did not yet reach the level of more exhaustive search-based or human reasoning approaches.

# Reflection and future prospects

Looking back, the initial, data-driven approach was an important first step, but it soon became clear that it had fundamental limitations. Realizing that the dataset-based network could not truly “understand” the game was both frustrating and instructive. It forced me to rethink my entire setup and eventually led to implementing the self-play system. This was the most significant technical challenge I faced, as it required reworking large parts of my codebase and developing an entirely new training pipeline from scratch.

Conceptually, the project taught me that not every complex problem is best solved with complex models. In fact, I learned that with the current speed of modern processors, simpler but more direct algorithms such as minimax can sometimes outperform neural networks. While neural networks can generalize and adapt, algorithms like minimax benefit from the immense amount of computational power current machines have.

If I had more time and computing power, I would explore several directions for improvement. First, running the self-play training for significantly more epochs or letting it play a few games against the minimax algorithm during training could lead to performance improving further. Second, optimizing minimax performance by porting it to a faster language such as C++ or using Cython would allow for deeper searches and more accurate comparisons. I would also like to fine-tune the architecture and hyperparameters of the neural network to test whether a more efficient configuration could push it further. Finally, a particularly promising avenue would be combining both approaches by using the neural network to evaluate board states within the minimax framework, similar to how modern chess engines like AlphaZero[[3]](#footnote-3) or Stockfish NNUE[[4]](#footnote-4) operate.

Overall, this project deepened my understanding of artificial intelligence as a technical field. It showed me the importance of iteration and of combining theory with experimentation. Most of all, it made me appreciate the huge amount of data and computation power needed to create neural networks.

# References

## Images:

Figure 1 : <https://upload.wikimedia.org/wikipedia/commons/thumb/a/ab/Nine_Men%27s_Morris_board_with_coordinates.svg/1200px-Nine_Men%27s_Morris_board_with_coordinates.svg.png>

Figure 2 : <https://upload.wikimedia.org/wikipedia/commons/thumb/6/6f/Minimax.svg/800px-Minimax.svg.png>

Figure 3 : <https://upload.wikimedia.org/wikipedia/commons/thumb/4/46/Colored_neural_network.svg/290px-Colored_neural_network.svg.png>

## Bibliography:

[1] 3Blue1Brown on Neural Networks:

<https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi>

[2] Stack overflow: <https://stackoverflow.com/questions> (used mostly for debugging and training loss visualisation; <https://stackoverflow.com/questions/74754493/plot-training-and-validation-loss-in-pytorch> )

[3] Pytorch documentation: <https://docs.pytorch.org/docs/stable/torch.html>

[4] MIT Minimax guide: <https://www.youtube.com/watch?v=STjW3eH0Cik>

[5] ChatGPT: <https://chatgpt.com>

[6] PyGame: <https://www.pygame.org/docs/>

[7] Ast (Abstract Syntax Trees): <https://docs.python.org/3/library/ast.html>

[8] Tkinter: <https://docs.python.org/3/library/tkinter.html>

[9] Matplotlib: <https://matplotlib.org/stable/contents.html>

[10] Math: <https://docs.python.org/3/library/math.html>

[11] Random: <https://docs.python.org/3/library/random.html>

[12] Copy: <https://docs.python.org/3/library/copy.html>

[13] The dataset was taken from the website <http://ai.unibo.it/DatasetNineMenMorris> which seems to have been shut down.

# Declaration of Originality

I hereby declare that I have completed this Matura thesis independently and without any unauthorized external assistance, and that all sources, aids, and websites have been truthfully used and referenced.

I have read and understood §11 of the School Leadership Conference’s guidelines on Matura theses.

[Place, Date, Signature]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. <https://stackoverflow.com/questions/6848828/what-is-the-difference-between-q-learning-and-sarsa> [↑](#footnote-ref-1)
2. <https://www.quantamagazine.org/cynthia-rudin-builds-ai-that-humans-can-understand-20230427> [↑](#footnote-ref-2)
3. <https://jonathan-laurent.github.io/AlphaZero.jl/stable> [↑](#footnote-ref-3)
4. <https://official-stockfish.github.io/docs> [↑](#footnote-ref-4)