UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria

Corso di Laurea triennale Ingegneria
Informatica e dell'automazione

ALGORITMO NEAT PER GIOCHI DA SCACCHIERA: PROGETTAZIONE, IMPLEMENTAZIONE E ANALISI COMPARATIVA

NEAT ALGORITHM IN CHESS-LIKE GAMES: DESIGN, IMPLEMENTATION, AND COMPARATIVE ANALYSIS

Relatore
Prof. Simone Fiori

Candidato
Alessandro Sebastianelli

Anno Accademico 2023/2024

Contents

1 Introduction **2**
- 1.1 Project objective and design considerations 2
- 1.2 Game setting and rules 3
- 1.3 How AI approach chess 5

2 The NEAT structure **9**
- 2.1 The approach 9
- 2.2 The genomic representation of individuals 10
- 2.3 Crossover and mutation 11
- 2.4 Speciation 13
- 2.5 Topology minimization 14

3 Implementation differences from NEAT **15**
- 3.1 Fitness Calculation 15
- 3.2 Loop Handling 17
- 3.3 Parents maintaining 19

4 Implementation of the algorithm **20**
- 4.1 *Game* and *Visualizer* programs 20
- 4.2 The NEAT program 23
  - 4.2.1 *Functions* 23
  - 4.2.2 *Genes* 24
  - 4.2.3 *Genome* 26
  - 4.2.4 *Network* 31
  - 4.2.5 *Nodes* 31
  - 4.2.6 *Tournament* 32
  - 4.2.7 *TrainingProcess* 34
- 4.3 *PlothGraph* and *PlayVsAI* programs 38

5 Training parametrization and results **39**
- 5.1 Choosing the parameters 39
- 5.2 The training process 40

6 Alternative to NEAT: tournament series **52**

7 Conclusions **56**

1 Introduction

1.1 Project objective and design considerations

The aim of this project is to construct an Artificial Intelligence, 'AI', capable of playing a generic board game, specifically utilizing a neural network that takes the entire game board as input, including the current positions of the pieces, and outputs an evaluation on a scale from -1 to 1. This evaluation serves as the basis for decision tree exploration.

When prompted for a move, the AI explores possible positions within a certain depth, i.e., a number of consecutive moves, and through a call to the network to evaluate the final positions, it decides the most advantageous move to make.

This AI will be built using a machine learning algorithm capable of autonomously learning the game mechanics and improving its performance, guided only by the rules for legal moves and feedback on wins or losses.

1.2 Game setting and rules

To test the effectiveness of the program, a simplified version of the chess game was created, taking into account several important considerations:

- The decision tree must be reduced compared to that of the original chess game to make testing the algorithm feasible within acceptable timeframes. This led to the elimination of the queen and a reduction in the size of the board.
- Pawns are essential to avoid situations of stalemate with repetitions of moves by pieces such as rooks and bishops.

The result is the following composition of the board and pieces:

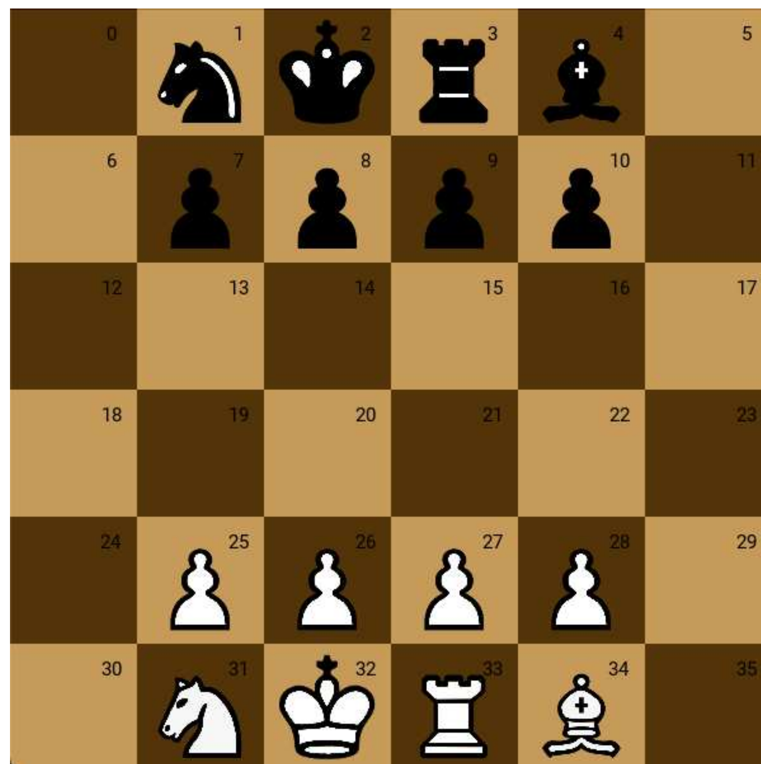


Figure 1: the chessboard initial setting

The objective of the game is to capture the opponent's king. 'Capture' refers to placing one's own piece on the square occupied by the opponent's piece, thereby removing the opponent's piece from the board.

The movement rules are the same as those of the original game:

- In making a move, the destination square cannot already be occupied by a piece of the same color. If there is an opponent's piece, it is captured.
- Bishops can move forward or backward along the diagonals of the starting square for any number of steps, provided they do not leap over another piece.
- Rooks can move forward or backward along the row and column of the starting square for any number of steps, provided they do not leap over another piece.
- Knights move according to the 'L' rule, i.e., one horizontal step followed by two vertical steps (or vice versa). Unlike bishops or rooks, there is no requirement for the trajectory to be free of other pieces.
- Each pawn can move forward along the column one square at a time. It can move one square forward diagonally only if an opponent's piece is present to be captured, otherwise, it cannot capture while moving along the column.
- The king can move to any adjacent square, including diagonally.

In contrast to chess, in this game:

- If a pawn reaches the last rank, it is immediately promoted to a rook. This simplification allows us to maintain this characteristic rule while limiting the high variability that would otherwise result.
- A pawn which has not yet moved from its starting rank cannot move forward two squares, and it cannot perform *en passant* captures.
- If the king is under threat of capture in the next move by an opponent's piece, i.e., it is in *check*, it is not required to move out of check.
- The king can move to a square where it would be in check.
- Castling is not allowed.

1.3 How AI approach chess

The evaluation of a chess position has been the subject of discussion and study for years. It has long been clear that constructing the tree of positions from the first move to find the best one is not feasible due to the high number of possible combinations. The situation is different for endgames, i.e., the study of positions with few pieces left on the board.

As early as 1912, Leonardo Torres y Quevedo built 'El ajedrecista', 'The Chess Player', an automaton capable of delivering checkmate with king and rook against a king moved by a human [1]. With technological advancements, a database has been created containing evaluations of endgames with up to seven pieces on the board [2].

Given the impossibility of constructing a complete tree, in the 1950s, thoughts began to emerge regarding the creation of functions to evaluate positions [3]. Among the early examples is the 'Los Alamos chess', capable of playing a simplified version of chess, which used a function to evaluate based on the number and type of remaining pieces and their positions [4]. The evolution of such functions has led to the development of modern chess engines.
However, this strategy remains confined to the development of specific algorithms for the referenced game. With this project, on the other hand, the aim is to create an AI capable of adapting to the game to which it is applied, using a neural network and machine learning.

A *neural network* is a computational model inspired by the structure and function of the human brain. It consists of interconnected nodes, the *neurons*, typically organized into layers and linked together by connections, the *synapses*. These components of the system work together to process data and make predictions or classifications.

In a neural network, initial information is converted into numerical values and put into the first layer through input neurons. It then passes through one or more hidden layers where computations take place, until it reaches the output neurons, which express the final result.

Each connection is associated with a *weight*, which is a multiplicative factor between the starting neuron and the ending one, determining the strength of the connection. As data progresses through the network, each neuron calculates the weighted sum of its inputs by multiplying each one by the corresponding weight and summing these values.

The result is then added to the *bias*, an offset that allows the network to implement complex relationships between the data. The resulting value is then related to the network's activation function; however, this process will be described when discussing the implementation.

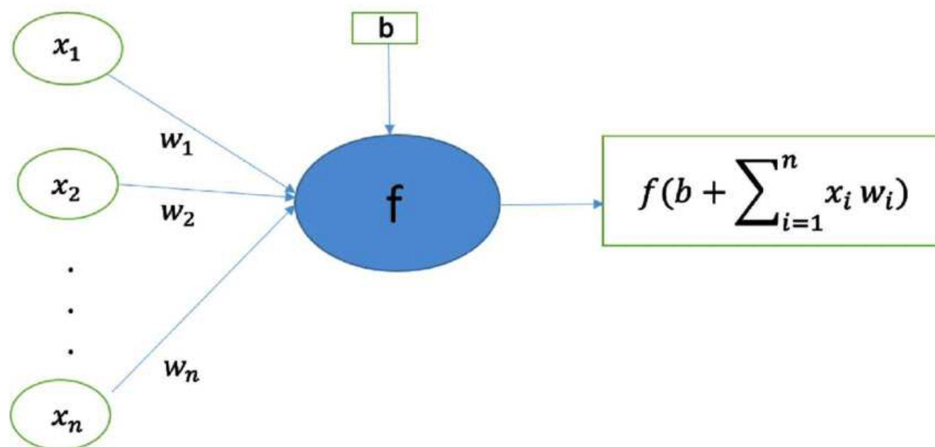


Figure 2: how data flows through the network [5]

During the learning process, the network adjusts weights and biases based on the training system, aiming to minimize the difference between its predictions and the desired outcomes.

In the field of machine learning, there are various types of training, among which the main ones are:

- *Supervised training*
  The model is trained with a set of input data that is already labeled, meaning the desired output is known. The goal is to configure the model correctly to associate inputs with outputs.

- *Unsupervised training*
  The model is trained on a set of unlabeled input data, meaning the desired output is not known. The objective is for the model to automatically identify patterns, structures, or groupings in the data without prior information.

- *Semi-supervised* training
  This type of training combines elements from the aforementioned two types. It uses a set of labeled input data as a basis and then works on unlabeled data. It can be a useful approach when there is an insufficient amount of labeled data.

- *Reinforcement training*
  The model learns to take actions within an environment to maximize a reward. This occurs through a trial-and-error system with feedback, where initial data is not always necessary. The goal is for the model to learn which actions are best to take based on the purpose for which it was trained.

Regarding the structure and training of the network, various solutions were evaluated, which can be grouped into three main categories:

- *Fixed topology with supervised training and backpropagation*
  A fixed topology is easy to manage and computationally efficient. However, supervised training requires a dataset to compare the obtained results, allowing the modulation of positive or negative feedback on the network through the loss function. This methodology is not applicable in this case because, to maintain the game's generic nature, predetermined evaluations of positions and specific cases cannot be relied upon.

- *Fixed topology with reinforcement training*
  To apply this method an external operator needs to decide which events to reward or penalize, and a function to assign bonuses or penalties is required. This activity cannot be performed rigorously for each individual game position due to the reasons outlined in the previous point. Alternatively, if feedback were given for individual games it would be too coarse-grained and could require too much time to be efficient.

- *Variable topology based on evolutionary principles*
  This approach relies on an evolutionary process that mimics natural selection based on the fitness parameter, i.e., how good the performance of a particular individual is. A first generation is formed by networks with minimal topology, which evolve through mutations and crossovers between individuals with better *fitness*. Iterations of this process lead to the exploration of various topologies while retaining the best ones.

Among the mentioned categories, the decision was made to opt for the third one, as it is the most feasible for the project.

2 The NEAT structure

2.1 The approach

To implement the variable topology, the NEAT technique, 'NeuroEvolution of Augmenting Topologies', was utilized, referencing the article *Evolving Neural Networks through Augmenting Topologies* (Stanley and Miikkulainen, 2002).

The NEAT method describes a training technique that employs concepts from biological evolution theory to train artificial neural networks. It represents a unique combination of learning techniques, but among the four traditional types of training, it aligns most closely with reinforcement learning, although it is somewhat different. For example, NEAT utilizes an evolutionary approach that allows it to explore, select, and improve neural networks, enabling them to adapt to the task over time, whereas in true reinforcement learning, the network interacts with the environment, seeking to maximize a reward.

The basic ideas of the algorithm are four: the genomic representation of individuals, crossover and mutation, speciation, and topology minimization.

2.2 The genomic representation of individuals

Each individual in the population is represented by a genome, which consists of a series of genes encoding neurons and synapses. Neurons genes identify the nodes that will constitute the neural network, while synapses genes determine the connections between these nodes within the network.

The peculiarity of this methodology lies in the use of a *global genome*, where each gene is associated with an *innovation number*, unique for each node or connection, which is shared among all individuals in the population. This approach allows evaluating the difference between two network structures based solely on the genotype, without the need for additional topological analysis, and enables crossover between individuals with different genomes.

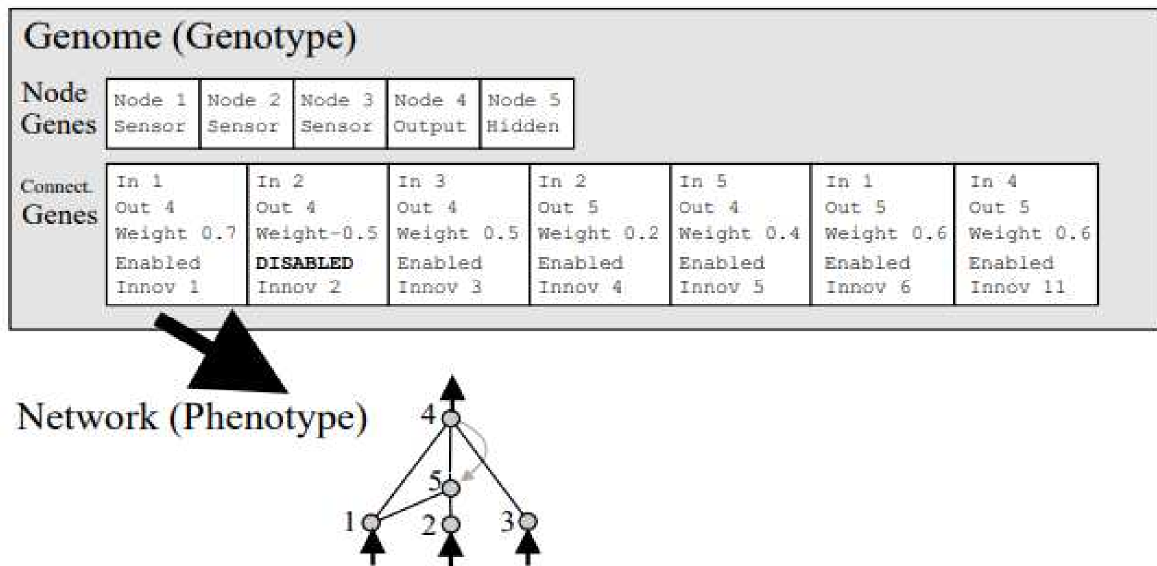


Figure 3: the representation of genotype and phenotype [6]

2.3 Crossover and mutation

After selecting the parents, i.e., individuals with the best fitness, the discarded individuals are replaced with offspring generated by crossing over the genomes of their parents. During crossover, each offspring inherits the structure of the parent with the highest fitness, while the weights and biases are randomly chosen from one of the two parents. To ensure the evolution of the population, a series of random mutations are then applied to the genomes. There are three main types of mutations:

- *Node addition*
  This mutation involves splitting an existing connection and inserting a new node within it.

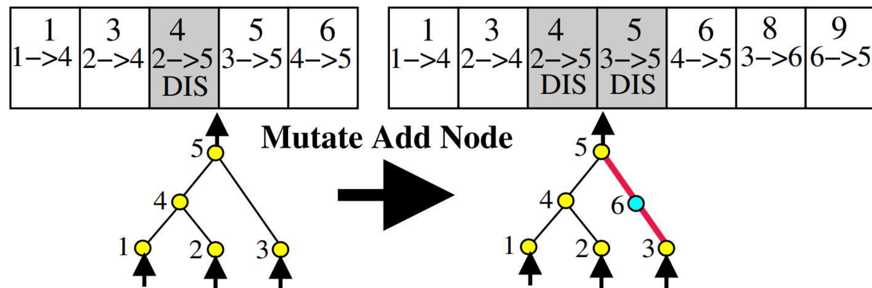


Figure 4: the 'Add Node' mutation [6]

- *Connection addition*
  In this case, a new connection is established between two nodes that were previously not connected.

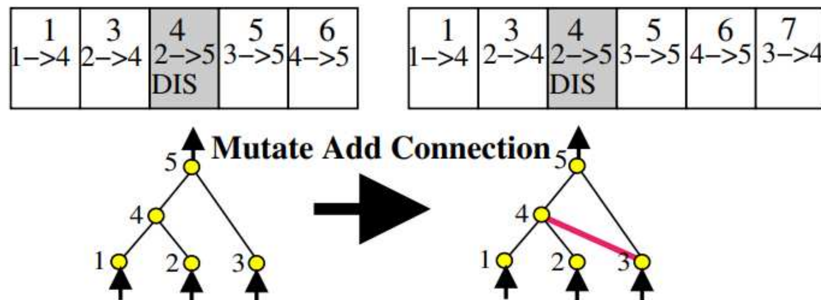


Figure 5: the 'Add Connection' mutation [6]

- *Value mutation*
  All weight and bias values are modified by a certain amount. This variation is obtained randomly from a Gaussian distribution with zero mean and a variance determined by the mutation power.

The first two types of mutations may pose a problem with the concept of the global genome previously described: it could happen that two different individuals undergo the same structural mutation, but each assigns its own innovation number to the new gene. To prevent this from happening, it is necessary to ensure that each mutation is immediately listed uniquely so that the global genome does not explode with duplicates of innovation numbers, which would be detrimental to the proper functioning of crossover.

2.4 Speciation

After a random structural mutation, it's almost inevitable that the fitness of the new individual decreases, putting it at risk of being discarded during the subsequent selection before it has had time to optimize. To prevent new structures from being eliminated prematurely, the speciation process is introduced.

This process involves dividing the population into different species, each with its own representative. Each new individual compares its genome with that of the species representative. If the difference between the two genomes is below a certain threshold, the individual is assigned to the same species as the representative; otherwise, the comparison is made with the next representative, and so on. If the individual is not similar to any of the existing species, a new species is created, and the individual becomes its representative.
This mechanism is important because, if only the absolute fitness of the new offspring were considered, their scores would initially be too low for valid reproduction. However, by being part of a new and therefore small species, their relative fitness can be considered. This means an adjusted fitness is used, where species with fewer elements receive a bonus multiplier.
In the next phase the fitness of each species is compared with the average fitness of the entire population, and the number of offspring that can be created is assigned based on their performance: the better their fitness, the more offspring they can produce.

Thanks to the adjustment of values described earlier, smaller species will be protected and have time to optimize and compete fairly with established individuals belonging to larger species, or potentially become extinct if they are not evolutionarily improvable.

2.5 Topology minimization

NEAT starts from a minimal topology, where the neural network consists only of input neurons directly connected to outputs. From this starting point, the algorithm gradually explores increasingly complex topologies, which are retained only if they have actual positive effects. This approach ensures that neural networks and their corresponding genomes remain small in size, thus favoring overall algorithm optimization.

This strategy of gradual and selective growth allows avoiding the introduction of unnecessary complexity. By initially keeping network and genome topologies simple, the NEAT process can focus on optimizing existing parameters before exploring new structures. This approach promotes greater computational efficiency and faster convergence towards optimal or near-optimal solutions, addressing one of the major challenges of neural network topology for AI.

3 Implementation differences from NEAT

During the design and implementation phase, it was necessary to make some adjustments to the algorithm outlined in the reference document to adapt it to the specifications of the project.

3.1 Fitness Calculation

The method for evaluating the algorithm's performance, as described in the article, revolves around controlling a cart with an inverted pendulum, aiming to keep the pendulum balanced. In this context, fitness is defined based on the duration of the individual's performance, meaning the period during which the network can keep the pendulum balanced by controlling the cart.
Similarly, this method is echoed in the article *Design and Simulation of a Neuroevolutionary Controller for a Quadcopter Drone* [7], where a neural network is used to control a drone: in this case, performance is better based on how long and how far the individual remains above the ground plane.

The common denominator of these methods is that the calculations are performed on an absolute basis: the network must survive within the same environment and is evaluated according to the same parameters.
However, in this case a different challenge is faced because the algorithm must autonomously learn game strategies in a generic gaming context, where absolute fitness evaluation is not possible. An evaluation environment, as in the two previous examples, is not present in this situation.

Therefore, an evaluation method based on the position (i.e., the score) obtained after a double round-robin tournament among all individuals in the population was opted for. It is important to emphasize that while this method is relative to the skill of the participants, the high number of matches and the fact that all individuals compete against each other approximate the evaluation to an absolute one.

It was believed that a double round-robin tournament offers numerous advantages that can improve the effectiveness of the training process compared to a classic tournament. Indeed, each network has the opportunity to compete twice against all other networks in the tournament. This approach allows them to face a variety of challenges and thus confront different opponent strategies, refining their skills over time. This iterative process of learning and adaptation contributes to greater robustness and generalization of neural networks, improving their performance in new and unexpected scenarios.

The double format instead of the single one seemed most suitable to us, as it allows for a more fair and accurate evaluation of network performance, as each network has the opportunity to play as both white and black, reducing any disparities related to playing order.

3.2 Loop Handling

During the implementation, an issue arose regarding the consistency of the neural networks generated through mutation: the reference article does not consider the formation of loops in the network structure.

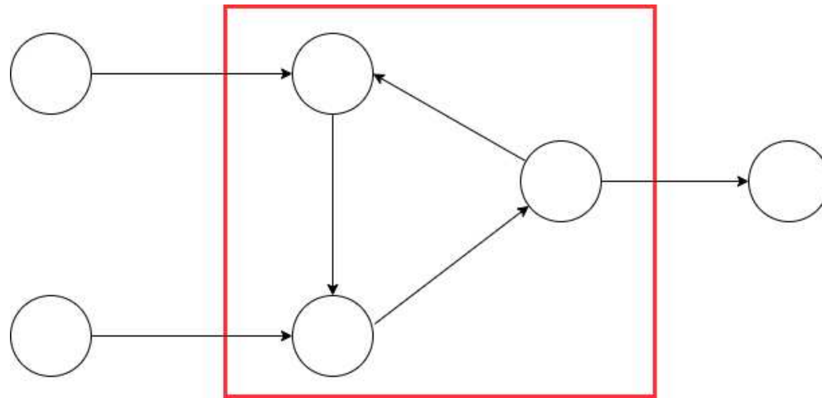


Figure 6: a network loop

This circumstance needs to be duly considered. In a time-based system, a loop connection is transformed into a memory connection, referring to the previous time step. However, in this case using a memory connection loses its relevance, as the evaluation of the network is based on a specific game position and there is no reference to previous or subsequent time steps. Additionally, since the method of exploring possible moves is based on a decision tree, referring to the previous move is not doable.

To solve this problem a loop-solving function was created: after each mutation that could potentially generate a loop, it checks for its presence in the generated phenotype and inhibits the expression of one of the connection genes causing it. In particular, several implementation options were considered:

- *Elimination of the loop connection from the network genomes*
  With this option, upon discovering a loop, one of the connections causing it is selected, and the corresponding gene is removed from the network genome. The advantage of this choice lies in simplifying the genomes, but on the other hand there is a greater risk that loops will be recreated in the same position with the addition of a new connection mutation.

- *Inhibition of the connection with the possibility of reactivation during crossover*
  Another possibility is to inhibit the gene causing the loop without eliminating it. Each gene is associated with an *active* or *inactive* status, and the connection responsible for the loop is deactivated by setting its status to inactive, preventing its expression in the network phenotype during construction. The status can be reactivated by a mutation, to allow the connection to become functional again in the network where it may no longer generate a loop. However, there remains a risk of needing to deactivate it again.

- *Permanent inhibition of the connection*
  In this variant, the active or inactive status is introduced with the same meaning as the previous one, but the status cannot change through mutation. Therefore, a connection that becomes inactive remains so for that network structure and its offspring. This approach limits the variability of the neural network but addresses the inconvenience of repeatedly managing the same loop.

Considering minimal disadvantage compared to the benefit obtained, the third implementation option described was chosen.

3.3 Parents maintaining

In the original version of NEAT, the parents of species are removed during the transition to the next generation after crossover and repopulation. This approach does not pose problems when the benchmark is defined by the environment itself, but it can be problematic when the network is trained for a new game without specific references other than the game rules.
The training process relies on the success rate compared to other players, determining the most performing neural networks that serve as reference standards for subsequent iterations. However, if these parent networks were to be removed, there would be no defined references to establish whether the new generations of networks are superior or inferior to the original ones.
For this reason, the decision was made to adapt the algorithm to the case by retaining the parents of each generation during the transition to the next generation.

4 Implementation of the algorithm

Firstly, it was decided to use the Python programming language, renowned for its speed and ease of writing, within the PyCharm Integrated Development Environment (IDE).
In the initial phase of the project, the user interface of the game was constructed. Kivy was chosen as the graphical framework, enabling the creation of dynamic interfaces in Python.

4.1 *Game* and *Visualizer* programs

The first programs created were *Game*, which facilitates loading the network, playing against it, and providing evaluations of positions, and *Visualizer*, which enables loading a game and navigating forward or backward with the visualization of moves.

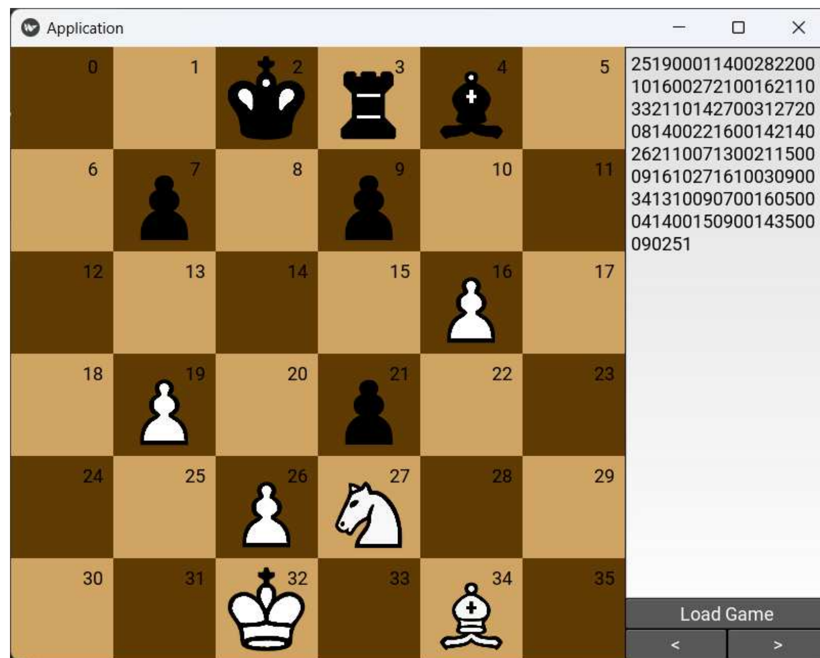


Figure 7: *Visualizer* interface

Figure 8: *Game* interface

Here a brief description of the *Game* classes:

- *NetworkManager*: it has a method to load a neural network from a genome string and one to set the depth level of search in the decision tree. Additionally, it provides a method to display the evaluation of a specific game position using the neural network, if available.
- *Square*: initializes the squares of the chessboard with a corresponding color image and handles their selection.
- *Piece*: initializes the pieces on the chessboard with the corresponding image.
- *Chessboard*: manages the state of the chessboard, including the position of the pieces and user interactions. It has methods to select a square, update the position of the pieces on the chessboard, reset the board to the initial position, and save moves to a text file containing the last played game.
- *Translator*: provides methods to convert between move sequences and text strings.
- *InputMenu*: manages the user interface for input, including buttons for loading the neural network and setting the search depth, as well as displaying the evaluation of the current position.

- *ChessboardVisual*: displays the chessboard on the GUI using a grid layout.
- *PieceBoardVisual*: displays the pieces on the chessboard using a grid layout.
- *InputMenuVisual*: represents the user interface for input, with buttons for actions such as resetting the board and loading the neural network.
- *MainWindow*: represents the main window of the application and contains the widgets for displaying the chessboard, pieces, and input interface.
- *Referee*: manages the game rules, including move validity checks and checking for a winner.
- *WindowManager*: manages the application's screen stack.
- *Application*: starts the application and manages its life cycle.
- *VisualPlayApplication*: initializes and starts the game with the user interface.

In the *Visualizer* program many classes are the same as those present in*Game*, with some exceptions. In particular, the *Referee* class no longer supervises the validity of moves: assuming that the string representing the game is consistent with the request, its main function is now to record and store information about the moves made between one move and the next. This is essential to allow navigation in the visualization of the game positions for both subsequent and previous moves.

4.2 The NEAT program

Subsequently, the core of the NEAT algorithm was developed, including genetic and phenotypic representations, and mutation and crossover processes.

4.2.1 *Functions*

Let's first look specifically at some functions used in the implementation:

- *clamp*: restricts values to ensure they do not surpass specified lower and upper bounds
- *sigmoid*: implements the sigmoid activation function, which maps an arbitrary value to a range between 0 and the *bound_up* parameter. The *stepness* parameter indicates the rate at which the function approaches its extremes

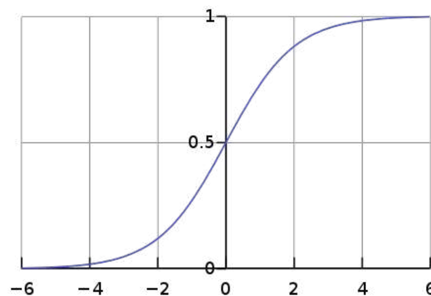


Figure 9: representation of the sigmoid function [8]

- *bilateral_sigmoid*: calculates the bilateral sigmoid, returning a value within the range between the -*bound_up* and the *bound_up* parameter. It relies on the standard sigmoid function to compute the value and scales it accordingly
- *unilateral_sigmoid*: is similar to the bilateral sigmoid but with a minimum value of 0. It ensures that only non-negative values are returned: it behaves like a sigmoid for values greater than 0 and equals 0 for negative values
- *str_to_bias*, *bias_to_str*, *str_to_weight*, *weight_to_str*: convert bias and weight values into genome string language using the base function *format_str*, and vice versa

- *linear_to_matrix*, *matrix_to_linear*: the first one converts a linear coordinate into a matrix coordinate considering the number of columns, while the second one performs the opposite operation
- *linear_range*, *matrix_range*: check if the linear or matricial position is within the acceptable range

4.2.2 *Genes*

Let's now delve into the neural network structure. Within the *Genes* file, the base class *GenericGene* is described, containing the *innov_index* attribute, essential for tracking the innovation number of the global genome. It also contains two abstract methods, *mutate* and *crossover*, which are implemented by derived classes.

The first derived class is *NodeGene*, which contains the bias attribute related to the node's bias and implements the mutate and crossover methods.

```
class NodeGene(GenericGene):
    def __init__(self, innovindex, bias):
        super().__init__(innovindex)
        self.bias = bias

    3 usages (3 dynamic)
    def mutate(self, mut_power):
        self.bias = clamp(self.bias + gauss( mu: 0, mut_power), -BIAS_BOUND, BIAS_BOUND)

    1 usage (1 dynamic)
    def crossover(self, gene_two):
        return NodeGene(self.innov_index, (self.bias + gene_two.bias) / 2)
```

Figure 10: the *NodeGene* code

The *mutate* function introduces random variations within the network according to a Gaussian distribution with a mean of zero, limiting the result to acceptable bias values. This allows us to adjust the mutation rate based on the individual's performance: the worse the performance, the higher the mutation rate.

The implementation of *crossover* is slightly different from that described in the reference articles. It is stated that during the crossover process, the structure of the most performing parent should be taken, while the biases should be randomly selected from either parent, gene by gene.

In this case, it was deemed more appropriate to average the values, both to avoid deviating too much from the performance of the best player and because the children undergo minimal mutation to continue exploring possible values.

The second derived class is *ConnectionGene*, containing information such as the connection weight, the input and output nodes, and a status attribute used for crossover.

```
class ConnectionGene(GenericGene):
    def __init__(self, innovindex, nodeout, nodein, weight, status):
        super().__init__(innovindex)
        self.innov_index = innovindex
        self.node_out = nodeout
        self.node_in = nodein
        self.weight = weight
        self.status = status  # 0: splittabile | 1: non splittabile | 2: morta

    1 usage (1 dynamic)
    def set_status(self, newstatus):
        self.status = newstatus

    3 usages (3 dynamic)
    def mutate(self, mut_power):
        self.weight = clamp(self.weight + gauss( mu: 0, mut_power), -WEIGHT_BOUND, WEIGHT_BOUND)

    1 usage (1 dynamic)
    def crossover(self, gene_two):
        return ConnectionGene(self.innov_index, self.node_out, self.node_in, (self.weight + gene_two.weight) / 2, self.status)
```

Figure 11: the *ConnectionGene* code

The *mutate* and *crossover* functions are analogous to those of the nodes, but they act on the connection weight rather than the bias.

Finally, there is the *InnovationManager* class, which manages the assignment and monitoring of innovation numbers for node and connection genes within evolving neural networks:

- *Innovation Number Management*
  Innovation numbers for node and connection genes are initialized to *INP_NUM* and *INP_NUM-1* respectively, where *INP_NUM* represents the number of input nodes.

The *get_innov_index_nodes* and *get_innov_index_conn* methods are responsible for returning the next innovation numbers for node and connection genes, subsequently incrementing the internal counters.

- *Gene Registration*
  The *genes_dict_conn* and *genes_dict_nodes* dictionaries allow the class to keep track of connection and node genes.
  The *record_connection* method records a new connection, associating its innovation number with the input and output node pairs.
  The *record_node* method records a new node, associating its innovation number and specifying which connection it originated from.

- *Gene Existence Checking*
  The *check_connection_recorded* and *check_node_recorded* methods allow checking if a connection or node has already been recorded in the respective dictionaries.

- *Loading and Resetting*
  The load and reset methods provide functionality to load initial values and reset innovation numbers, useful for managing initialization and restart states.

4.2.3 *Genome*

The *Genome* file contains the *GenomeManager* and *Genome* classes, responsible for managing species and genome IDs and representing individual genomes, respectively.

The *GenomeManager* class is responsible for managing the IDs of species and genomes within the system. It provides functionality to obtain new IDs for species and genomes, reset them according to the game, and load initial values. Acting as a central handler for generating and assigning unique IDs, the *GenomeManager* facilitates organization and control of coherence and consistency in genome creation and manipulation processes within the evolutionary system.

The *Genome* class represents a single genome in an evolutionary system, providing methods for initializing, manipulating, and analyzing genetic information.

Each instance contains data such as species and genome IDs, node and connection genes, as well as core functions such as:

- *init_by_string*: initializes a genome from a properly constructed string. The parameterized structure allows varying the length of various values as needed.

```
def init_by_string(self, genome_str):
    self.species_ID = int(genome_str[0:SPECID_CAR])
    self.genome_ID = int(genome_str[SPECID_CAR:SPECID_CAR + GENID_CAR])
    self.output_node_gene = NodeGene(INP_NUM, str_to_bias(
        genome_str[SPECID_CAR + GENID_CAR:SPECID_CAR + GENID_CAR + BIAS_CAR]))
    extracted_str = genome_str[SPECID_CAR + GENID_CAR + BIAS_CAR: genome_str.find(':')]
    for i in range(0, len(extracted_str), NODEID_CAR + BIAS_CAR):
        innov_index = int(extracted_str[i:i + NODEID_CAR])
        bias = str_to_bias(extracted_str[i + NODEID_CAR: i + NODEID_CAR + BIAS_CAR])
        self.node_genes.update({innov_index: NodeGene(innov_index, bias)})
    extracted_str = genome_str[genome_str.find(':') + 1:]
    for i in range(0, len(extracted_str) - 1, CONNID_CAR + NODEID_CAR + NODEID_CAR + WEIGHT_CAR + 1):
        innov_index = int(extracted_str[i:i + CONNID_CAR])
        node_out = int(extracted_str[i + CONNID_CAR:i + CONNID_CAR + NODEID_CAR])
        node_in = int(extracted_str[i + CONNID_CAR + NODEID_CAR:i + CONNID_CAR + NODEID_CAR + NODEID_CAR])
        weight = str_to_weight(extracted_str[
                          i + CONNID_CAR + NODEID_CAR + NODEID_CAR:i + CONNID_CAR + NODEID_CAR + NODEID_CAR + WEIGHT_CAR])
        status = int(extracted_str[i + CONNID_CAR + NODEID_CAR + NODEID_CAR + WEIGHT_CAR])
        self.conn_genes.update({innov_index: ConnectionGene(innov_index, node_out, node_in, weight, status)})
```

Figure 12: the *init_by_string* code

- *get_string*: returns a string representation of the genome.

```
def get_string(self):
    out_bias_str = bias_to_str(self.output_node_gene.bias)
    nodes_str = ""
    conn_str = ""
    for n in self.node_genes.values():
        nodes_str = nodes_str + format_str(n.innov_index, NODEID_CAR, str_filler: "0") + bias_to_str(n.bias)
    for c in self.conn_genes.values():
        conn_str = conn_str + format_str(c.innov_index, CONNID_CAR, str_filler: "0") + \
                   format_str(c.node_out, NODEID_CAR, str_filler: "0") + \
                   format_str(c.node_in, NODEID_CAR, str_filler: "0") + \
                   weight_to_str(c.weight) + str(c.status)
    genome_str = format_str(self.species_ID, SPECID_CAR, str_filler: "0") + \
                 format_str(self.genome_ID, GENID_CAR, str_filler: "0") + out_bias_str + nodes_str + ":" + conn_str
    return genome_str
```

Figure 13: the *get_string* code

- *construct_net_from_genome*: create a neural network using the information contained in the genome. It is constructed in two parts: the first part contains the neurons and their biases, while the second part contains the connections and the neurons they link.
- *calculate_distance*: calculates the distance between the current and another specified genome. This distance is the result of the sum of several factors:
  - the number of excess genes, i.e., nodes and connections,
  - the number of disjoint genes,
  - the difference in connection weight values.

  According to the reference article [6], it is not necessary to consider the difference in bias values, despite being a distinctive parameter of the network. In the implementation, this indication was adhered to.
- *mutate_structure*: performs a mutation on the genome structure by adding or removing connections or nodes. This occurs only if the result of a random function exceeds a parametric probability threshold.
- *mutate_values*: performs a mutation on the genome values by modifying connection weights and node biases.
- *crossover*: performs the crossover operation between the main genome and another genome specified as the parent.
- *loop_solver*: Solves any loops that may be present in the neural network represented by the genome.

Apart from the core functionalities previously discussed, it is important to highlight that some of the most substantial functions implemented to enable the dynamic evolution of the neural network's structure, namely the addition of a new node and a new connection.

- *add_node_mutation*: adds a node mutation to the genome by splitting an existing connection into two and inserting the node inside with a bias of 0. Uniqueness is ensured by the specificity of the broken connection.

```
1 usage
def add_node_mutation(self, innovmanag):
    conn_to_split = choice([c for c in list(self.conn_genes.values()) if c.status == 0])

    res = innovmanag.check_node_recorded(conn_to_split.innov_index)
    if res[0]:
        new_node = NodeGene(res[1],  bias: 0.0)
    else:
        new_node = NodeGene(innovmanag.get_innov_index_nodes(),  bias: 0.0)
        innovmanag.record_node(new_node, conn_to_split.innov_index)
    self.node_genes.update({new_node.innov_index: new_node})

    res = innovmanag.check_connection_recorded(conn_to_split.node_out, new_node.innov_index)
    if res[0]:
        new_conn_1 = ConnectionGene(res[1], conn_to_split.node_out, new_node.innov_index, FIRST_WEIGHT_VALUE,  status: 0)
    else:
        new_conn_1 = ConnectionGene(innovmanag.get_innov_index_conn(), conn_to_split.node_out, new_node.innov_index,
                                    FIRST_WEIGHT_VALUE,  status: 0)
        innovmanag.record_connection(new_conn_1)
    self.conn_genes.update({new_conn_1.innov_index: new_conn_1})

    res = innovmanag.check_connection_recorded(new_node.innov_index, conn_to_split.node_in)
    if res[0]:
        new_conn_2 = ConnectionGene(res[1], new_node.innov_index, conn_to_split.node_in, conn_to_split.weight,  status: 0)
    else:
        new_conn_2 = ConnectionGene(innovmanag.get_innov_index_conn(), new_node.innov_index, conn_to_split.node_in,
                                    conn_to_split.weight,  status: 0)
        innovmanag.record_connection(new_conn_2)
    self.conn_genes.update({new_conn_2.innov_index: new_conn_2})

    conn_to_split.set_status(1)
    conn_to_split.weight = 0
```

Figure 14: the *add_node_mutation* code

- *add_connection_mutation*: adds a connection mutation by inserting a new one between two previously unconnected nodes of the neural network, which also determines its uniqueness.
  This is done by first creating a list of all possible connections, i.e., nodes not yet connected to each other, while the choice is made by a choice function, and the new weight is decided by the uniform function, which takes a random value between the minimum and maximum parameters passed to it.

```
1 usage
def add_connection_mutation(self, innovmanag):
    possible_connections = []
    for ng1 in self.node_genes.values():
        for ng2 in self.node_genes.values():
            if not (ng1.innov_index == ng2.innov_index):
                connected = False
                for cg in self.conn_genes.values():
                    if ((cg.node_in == ng1.innov_index and cg.node_out == ng2.innov_index) or (
                            cg.node_in == ng2.innov_index and cg.node_out == ng1.innov_index)):
                        connected = True
                        break
                if not connected:
                    possible_connections.append((ng1.innov_index, ng2.innov_index))
        for i in range(INP_NUM):
            connected = False
            for cg in self.conn_genes.values():
                if cg.node_in == ng1.innov_index and cg.node_out == i:
                    connected = True
                    break
            if not connected:
                possible_connections.append((i, ng1.innov_index))
        connected = False
        for cg in self.conn_genes.values():
            if cg.node_in == INP_NUM and cg.node_out == ng1.innov_index:
                connected = True
                break
        if not connected:
            possible_connections.append((ng1.innov_index, INP_NUM))

    if possible_connections:
        conn_to_create = choice(possible_connections)
        res = innovmanag.check_connection_recorded(conn_to_create[0], conn_to_create[1])
        if res[0]:
            new_conn = ConnectionGene(res[1], conn_to_create[0], conn_to_create[1],
                                      uniform(-WEI_CR_BOUND, WEI_CR_BOUND),  status: 0)
        else:
            new_conn = ConnectionGene(innovmanag.get_innov_index_conn(), conn_to_create[0], conn_to_create[1],
                                      uniform(-WEI_CR_BOUND, WEI_CR_BOUND),  status: 0)
            innovmanag.record_connection(new_conn)
        self.conn_genes.update({new_conn.innov_index: new_conn})
```

Figure 15: the *add_connection_mutation* code

4.2.4 *Network*

The *Network* class represents a neural network that can be constructed from a specific genome. Within its constructor, input, hidden, and output nodes are initialized along with their connections based on the genome's specifications. Input nodes are associated with input values provided to the network, while hidden and output nodes process these inputs through weighted connections.

- The *calculate_output* method computes the neural network's output given a list of input values. This method triggers the forward propagation step in the network, where input values are propagated through nodes and connections to the output, producing the result. After output calculation, nodes are reset to their initial states to prepare the network for further computations.

- The *loop_check* method is responsible for detecting any cycles in the neural network. This is a critical step to avoid infinite loops during output computation, as they would cause program execution to stall or fail. If a loop is detected, the method returns a warning signal and provides information on the connections involved in the cycle.

4.2.5 *Nodes*

Now let's consider the *Nodes* file, where provided classes represent nodes in a neural network and are designed to process input, compute intermediate values, and generate output. These nodes are integral to the neural network's structure and play a fundamental role in data processing within the system.

- The *GenericNode* class provides a general base for all node types. Each node has an ID and a series of flags indicating the node's state, which are then used for output calculation and loop detection. The specific implementation of the *calculate* method is left to subclasses, while input nodes have a particular *reset* method.
  Each node has a list of inputs and a list of weights associated with each input. These weights are used during node value computation to determine the importance of each input in the final result.

```python
class GenericNode:
    def __init__(self, nodeid):
        self.value = 0
        self.busy_flag = False
        self.alr_calc_flag = False
        self.node_ID = nodeid
        self.reset_node()

    6 usages (2 dynamic)
    def reset_node(self):
        self.value = 0
        self.busy_flag = False
        self.alr_calc_flag = False

    4 usages (2 dynamic)
    def get_node_value(self, caller_node_id):
        if self.alr_calc_flag:
            return True, self.value
        if self.busy_flag:
            return False, (self.node_ID, caller_node_id)
        return self.calculate()

    1 usage
    def calculate(self):
        pass
```

Figure 16: the *GenericNode* code

- The *HiddenNode*, *InputNode*, and *OutputNode* classes extend the *GenericNode* class, each with specific behavior. Input nodes directly receive input values and maintain their value unchanged, internal nodes process intermediate values using a unilateral sigmoid activation function, while the output node produces the final result by applying a bilateral sigmoid activation function.

4.2.6 *Tournament*

Let's delve into the section concerning the game structure in the *Tournament* file. First, there's the *ChessBoard* class, which initializes the game board position and can save moves as string while determining who starts the game between white and black players.

Subsequently, a series of functions serve various purposes:

- *pair_challengers*: implements logic to pair players, enabling the generation of matches between them. Each pair can play a double round-robin or a single one, depending on the chosen configuration.

- *position_after_move*, *check_win_condition*: manage the game state, such as position, piece promotion, or player turn, and determine if a game has ended and who the winner is.

- *possible_moves*: handles the movements of various pieces and generates a list of all possible legal moves from a given position on the chessboard.

- *calculate_position*: implements a minimax search algorithm with alpha-beta pruning to determine the best move to make in a given position on the chessboard considering a certain number of moves ahead.
  The minimax algorithm is a widely used decision-making approach in turn-based two-player games like chess. It aims to maximize the current player's advantage while minimizing the opponent's advantage, assuming both players play optimally.
  The algorithm works recursively, exploring possible moves to the desired depth and evaluating the effectiveness of each move through an evaluation function. When it reaches the maximum depth or a terminal state (such as a win or a draw), the algorithm returns the position's value.
  Alpha-beta pruning is an optimization technique used to reduce the number of explored branches of the search tree by eliminating those that definitely will not affect the final decision. It works by keeping track of two values, alpha and beta, representing the guaranteed minimum value for the maximizing player and the guaranteed maximum value for the minimizing player, respectively.
  During tree exploration, if a move is found that leads to a value higher than beta for the maximizing player or lower than alpha for the minimizing player, the branch is eliminated because it would not influence the final choice.
  This potentially significantly reduces the number of explored positions, improving the algorithm's efficiency without compromising the accuracy of the decision made.

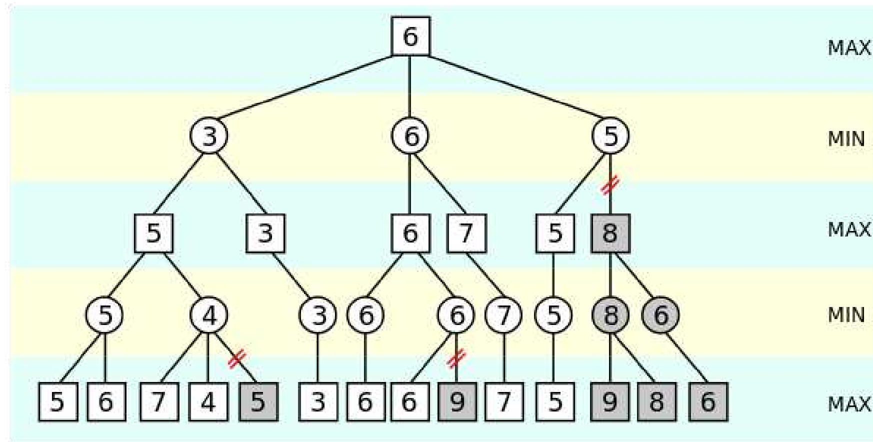


Figure 17: representation of alpha-beta pruning [9]

- *perform_match*: simulates a match between two players, using the previously defined logic to determine the best moves for each player. At the end of the match, it returns the winner, the number of moves made, and a string representing the entire match.

4.2.7 *TrainingProcess*

Finally, the implementation includes an algorithm that generates actual training sessions, incorporating phases of selection, repopulation, and speciation of individuals, along with logic for recording games and genomes.

Session_Record, *Game_Recording*, and *Genome_Recording* are the files used for the saving. The first file stores information regarding the training session (e.g., number of players, search depth, number of games, etc.). The second file records the matches of a tournament during the training session and, through parametrization, it allows the selection of intervals for recording tournaments. The third file preserves the genomes of networks related to the saved tournaments.

Additionally, a backup functionality is incorporated to resume the training from where it was left off, achieved through storage in the *Session_Backup* file.

The main classes within the *TrainingProcess* program are as follows:

- *RecordManager*: *update_res_recording* and *update_game_recordings* methods respectively record tournament results and game data information, while the *clear_recordings* method empties the three recording files. *record_session* store the session information.

- *Result*: tracks the score of each neural network.

- *Session*: manages the saving of session information through the methods *save_session*, *load_session*, and *reset_session*, responsible for saving the session, loading the saved section, and restoring the backup to the initial values, respectively.

The core function, *training*, drives the training process.

It initially loads the previous session or creates a new one. In the case of a new session, the network is created, and an initial mutation is performed.

Then it creates neural networks from their genomes and organizes tournaments with their challenges, utilizing Python's multiprocessing to expedite the process and maximize available resources.

It records tournament results, evaluates network performance, and selects a percentage of parents to create the new generation.

```
parents.clear()
results.sort(reverse=True, key=lambda res: res.score)
if iter_count % REC_INTERVAL == 0:
    rec_manag.update_res_recording(results, s)  # I GENOMI NON SONO ORDINATI IN ORDINE DI ID
for r in results:
    print(str(r.species_ID) + " | " + str(r.genome_ID) + " Score: " + str(r.score))
for spk in s.Species.keys():
    n_parents = max(1, round(PARENT_PERC * len(s.Species.get(spk))))
    parents_list = []
    c = 0
    for r in results:
        if r.species_ID == spk:
            parents_list.append(s.Species.get(spk).get(r.genome_ID))
            c += 1
            if c == n_parents:
                break
    parents.update({spk: parents_list})
```

Figure 18: evaluation of fitness and the selection of species parents

It computes relative fitness and the resulting number of offspring granted.

Due to approximation, maintaining a constant number of elements in the population emerged as an issue. To address this, the *choice* function is utilized to select a random species to reward or cut for rounding up or down the number of offspring.

```
adj_results_list = []
for r in results:
    adj_results_list.append(Result(r.species_ID, r.genome_ID, r.score / len(s.Species.get(r.species_ID))))

num_elem_in_spec = {}
fit_sum = 0
for adj_fit in adj_results_list:
    fit_sum += adj_fit.score
adj_fit_mean = (fit_sum / s.PopNum)
for spk in s.Species.keys():
    fit_sum = 0
    for adj_fit in adj_results_list:
        if adj_fit.species_ID == spk:
            fit_sum += adj_fit.score

    num_elem = round(fit_sum / (adj_fit_mean + 0.00001))
    if num_elem < 2:
        num_elem = 0
    num_elem_in_spec.update({spk: num_elem})
```

Figure 19: calculation of adjusted fitnesses and the new number of elements in each species

```
while sum(num_elem_in_spec.values()) < s.PopNum:
    num_elem_in_spec[choice([spk for spk in list(num_elem_in_spec.keys()) if num_elem_in_spec[spk] != 0])] += 1
    print("A corrective addition was made to the population.")

while sum(num_elem_in_spec.values()) > s.PopNum:
    max_key = None
    max_num = 0
    for key in num_elem_in_spec.keys():
        num = num_elem_in_spec.get(key)
        if num > max_num:
            max_key = key
            max_num = num
    num_elem_in_spec[max_key] -= 1
    print("A corrective subtraction was made to the population.")

s.PopNum = sum(num_elem_in_spec.values())
```

Figure 20: maintaining a constant population

It generates offspring neural networks through crossover and mutates them according to selected parameters at the beginning of the training.

```
for spec_key in num_elem_in_spec.keys():
    children_num = max(0, (num_elem_in_spec.get(spec_key) - len(parents.get(spec_key))))
    for i in range(children_num):
        parent_1 = choice(parents.get(spec_key))
        parent_2 = choice(parents.get(spec_key))

        offspring = parent_1.crossover(parent_2, s.genome_manager.get_new_genome_id())
        childrens.append(offspring)

for child in childrens:
    child.mutate_structure(s.innov_manager)
    child.mutate_values(BIAS_MUTATION_POWER, WEIGHT_MUTATION_POWER)

s.Species.clear()
for spec_key in parents.keys():
    for i in range(min(len(parents.get(spec_key)), num_elem_in_spec.get(spec_key))):
        specie = s.Species.get(spec_key)
        p = parents.get(spec_key)[i]
        if specie is None:
            s.Species.update({p.species_ID: {p.genome_ID: p}})
        else:
            specie.update({p.genome_ID: p})
```

Figure 21: offspring creation and mutation, and reinsertion of parents into species

It manages speciation, the process of classifying networks into species based on their genetic similarity with species representatives taken from parents.

```
compare_sub_dict = {}
for spec_key in parents.keys():
    compare_sub_dict.update({spec_key: choice(parents.get(spec_key))})

while childrens:
    child = childrens.pop(0)
    assigned = False
    for spec_key in compare_sub_dict.keys():
        delta = child.calculate_distance(compare_sub_dict.get(spec_key))
        if delta < SPECIATION_DELTA:
            child.species_ID = spec_key
            spec = s.Species.get(spec_key)
            if spec:
                spec.update({child.genome_ID: child})
            assigned = True
            break
    if not assigned:
        child.species_ID = s.genome_manager.get_new_species_id()
        s.Species.update({child.species_ID: {child.genome_ID: child}})
        compare_sub_dict.update({child.species_ID: child})
```

Figure 22: selection of species representatives and speciation of offspring

Finally it updates and saves the training session state for the next cycle.

4.3 *PlothGraph* and *PlayVsAI* programs

Upon completing the training algorithm, two additional programs were created: *PlotGraph*, for obtaining graphical visualization of the network, and *PlayVsAI*, for playing against a specific network and assessing the effectiveness of its evaluations. In particular, *PlayVsAI* allows the selection of both the game color and the depth of move searches that the neural network must calculate.

Figure 23: *PlayVsAI* interface

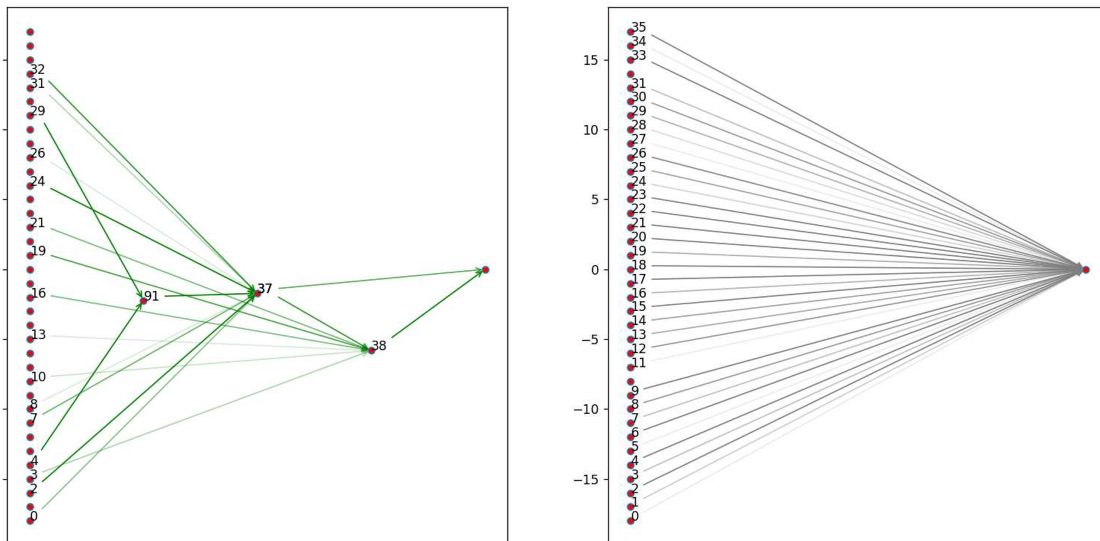


Figure 24: a network representation. The direct connections between the inputs and outputs are depicted in a separate graph for the sake of readability, while the transparency of the connection depends on its strength: the weaker the connection, the more transparent it appears

5 Training parametrization and results

5.1 Choosing the parameters

Before starting the training sessions, the characteristic parameters governing its execution needed to be carefully selected [10].
Firstly, a population consisting of fifty neural networks was opted for, to optimize the balance between sample size and the number of games played during each training session. This choice is justified by the structure of the double round-robin tournament, where increasing the number of participants would result in an exponential increase in the total number of matches. As a maximum number of moves per game, forty seemed to be an appropriate starting point.

Another aspect related to timing concerns the depth of search used by the algorithm during the training of neural networks. A depth of 1 was adopted for two reasons: firstly, for computational efficiency, and secondly, because this depth initially proves to be the most effective, allowing the network to examine up to its next move and receiving positive feedback in case of capturing the opponent's piece. Indeed, higher depths would have been substantially suboptimal as they would require a longer adaptation period to understand advantageous situations.
Given the population size, it was chosen to select the top quarter of elements from each species as parents. This decision represents a compromise between selecting too few, to maintain sufficient variety, and selecting too many, which would render the concept of using parents less meaningful.

The next parameters to determine concern speciation, for which NEAT articles do not provide specific guidance on the values to assign. To address this issue, a strategy was selected as follows.
The maximum difference between species was normalized to a unitary value. Additionally, the parameter concerning the difference in the number of excess or deficit nodes was set to 0.1, intentionally a low value. This choice is motivated by the strong correlation of this parameter with the number of connections within the neural network, which are typically more numerous and of greater relevance than the number of nodes. This approach limits the number of variables to consider, focusing on the most relevant ones for the optimization process.

5.2 The training process

In the succeeding phase of the experimental process, initial training tests were conducted while keeping the speciation parameters of connections and their weights constant, while varying the mutation parameters. These parameters include both the number of nodes and connections to be introduced during a mutation, and those related to the magnitude of the bias and connection weight modifications for the new generations.

It is noteworthy that, in cases where the mutation of adding connections is activated, five attempts are made instead of a single one. This procedural adaptation was introduced considering that, unlike the specific examples presented in the NEAT-related articles, the neural networks involved in this project are characterized by a total of 36 inputs. Therefore, it was deemed necessary to increase the rate of connection addition to allow nodes to handle a significant number of incoming connections.

The evaluation of the network involves several steps. Firstly, the networks that achieved the highest scores in the last series of tournaments are selected from the *game_recording* file. Subsequently, using the *Visualizer* program, some of their games are observed to assess their performance. After that, the most characteristic games are input into the *Game* program to observe the network's actual evaluation of the position move by move.
This initial screening allows us to ascertain the extent to which the network has learned the dynamics of the game.

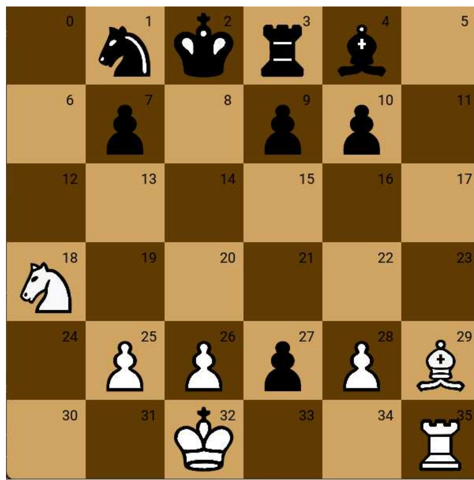
At this point, to quantify the learning of each network, an evaluation scale was created based on four parameters with respective scores:

- **SK**: 'save king', if the network protects its own king from capture by the opponent. 0 to 10 points
- **MK**: 'mate king', if the network captures the opponent's king. 0 to 10 points
- **SP**: 'save pieces', if the network defends its own pieces in hierarchical order, for example saving a rook rather than a pawn. 0 to 5 points

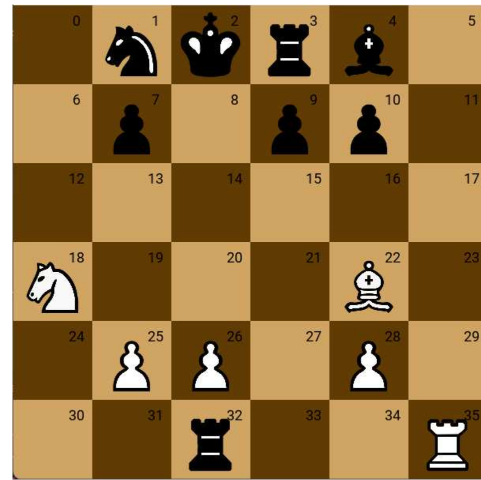
- **MP**: 'mow pieces', if the network captures opponent pieces in hierarchical order, for example prioritizing the capture of a rook over a pawn. 0 to 5 points

It was considered that introducing other types of evaluations might risk diverting from the game's main objectives.
Scores are assigned based on how often a network reflects the individual parameter during a series of games played against a human opponent.

The initial assessment was conducted on an untrained network. As expected, the AI played randomly, without any discernible purpose:



The AI plays as white, it's its turn



It AI moved the bishop in sq. 22, so black pawn can now take the white king

Figure 25: the AI moved the bishop in a not useful position instead of protecting its own king

ID	Evaluation				
	SK:	MK:	SP:	MP:	TOT:
0	0	6	0	1	7

Figure 26: evaluation of the untrained AI

In the first set of evaluations, three categories of cases were examined, characterized by low, medium, and high variability, in training sessions consisting of 1.000 iterations. The parameter values used for each category are shown in the figure 27.

ID	Iter. (Depth)	Connec. Diff. Spec.	Weight Diff. Spec.	Add Node	Add Conn. - N° try	Bias / Weight Mut. Pow.
1	1000 (1)	0.9	0.1	0.1	0.1 - 5	0.2
2	1000 (1)	0.9	0.1	0.1	0.1 - 5	0.5
3	1000 (1)	0.9	0.1	0.1	0.1 - 5	0.8
4	1000 (1)	0.9	0.1	0.3	0.5 - 5	0.2
5	1000 (1)	0.9	0.1	0.3	0.5 - 5	0.5
6	1000 (1)	0.9	0.1	0.3	0.5 - 5	0.8
7	1000 (1)	0.9	0.1	0.5	0.9 - 5	0.2
8	1000 (1)	0.9	0.1	0.5	0.9 - 5	0.5
9	1000 (1)	0.9	0.1	0.5	0.9 - 5	0.8

Figure 27: the first set of training sessions

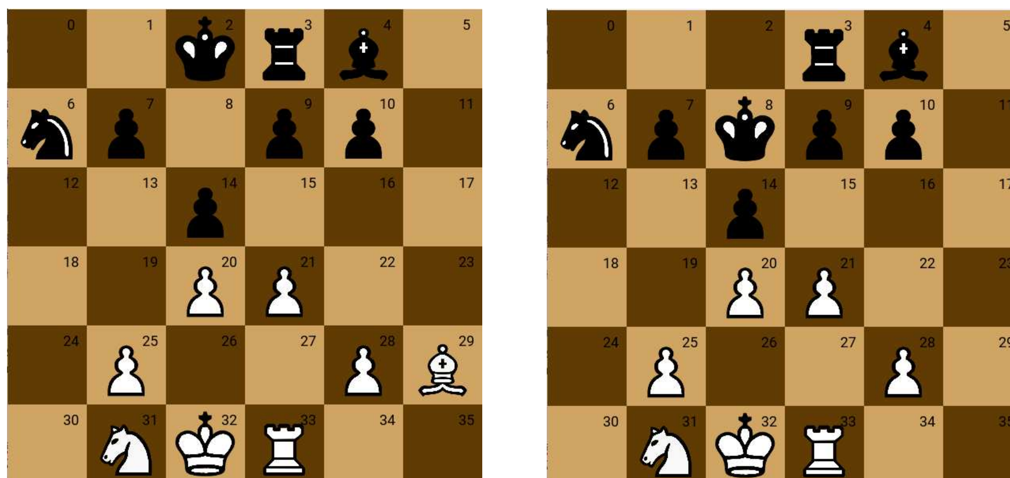
- **ID**: the training's ID
- **Iter. (Depth)**: the number of tournaments and the AI depth of move search
- **Connec. Diff. Spec.**: the value attributed to the difference in the number of connections, as a percentage normalized to 1
- **Weight Diff. Spec.**: the value attributed to the difference of connections weights, as a percentage normalized to 1
- **Weight Diff. Spec.**: the value attributed to the difference of connections weights, as a percentage normalized to 1
- **Add Node**: the probability of adding a node, as a percentage normalized to 1
- **Add Conn. - N° try**: the probability of adding a connection, as a percentage normalized to 1, and the number of attempts per time
- **Bias / Weight Mut. Pow.**: the variance of the Gaussian mutation for bias and weight

The approximate results are shown in figure 28.

ID	Evaluation				
	SK:	MK:	SP:	MP:	TOT:
1	4	9	2	1	16
2	3	7	2	2	14
3	2	5	1	2	10
4	3	7	2	2	14
5	1	8	0	2	11
6	1	5	2	1	9
7	4	8	3	2	17
8	2	7	2	1	12
9	1	8	0	0	9

Figure 28: evaluation of the first set

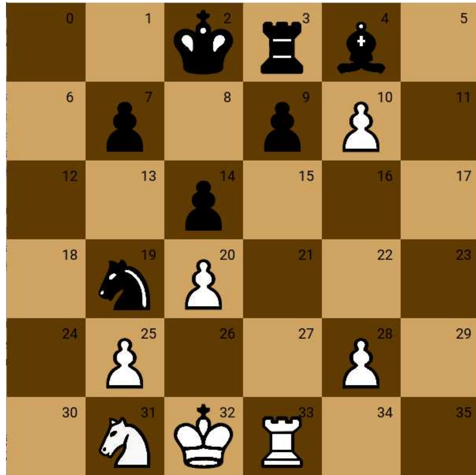
It has been observed that the AIs have indeed learned the basics of the game, although they still make fairly significant errors.
In the figures 29 and 30 are shown some good gameplay behaviors by the network 35.338 from the session 1:



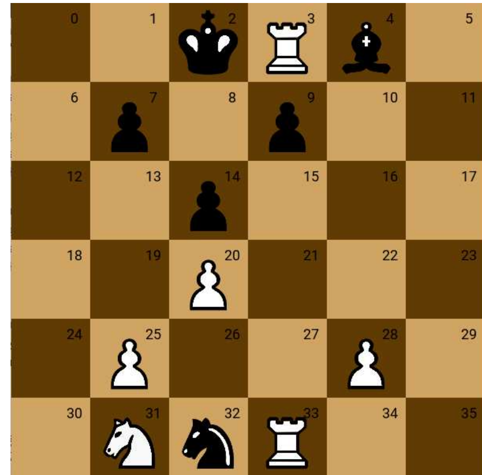
The AI plays as black, it's white turn

The white bishop is captured after moving to sq. 8

Figure 29: the AI has chosen to take the bishop instead of the pawn in sq. 21

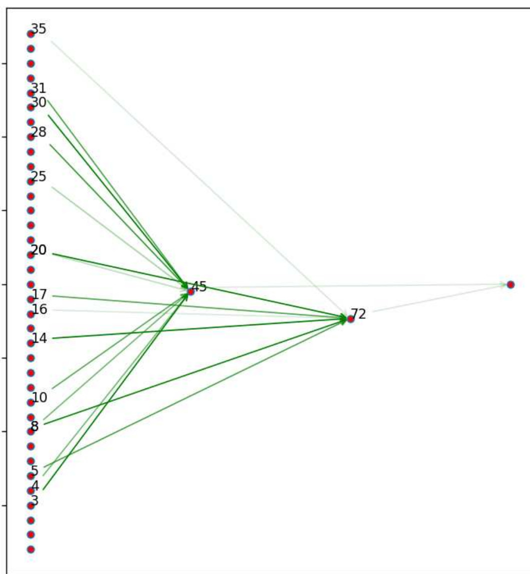


The AI plays as black, it's white turn



The AI takes the king and wins the game

Figure 30: even if the white pawn was going to take the rook and be promoted, the AI recognised the best choice



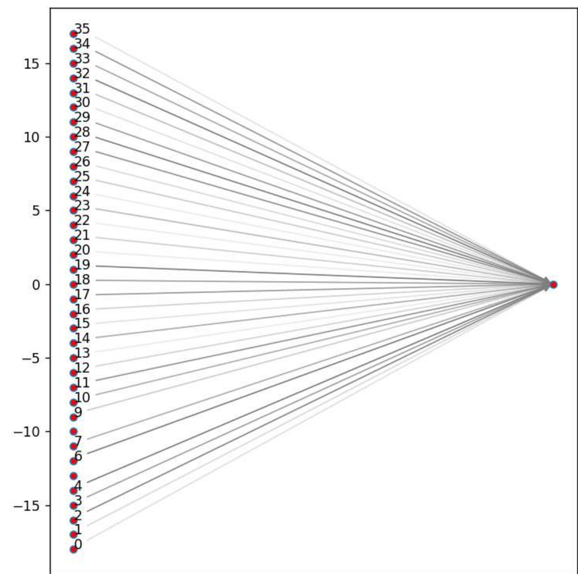
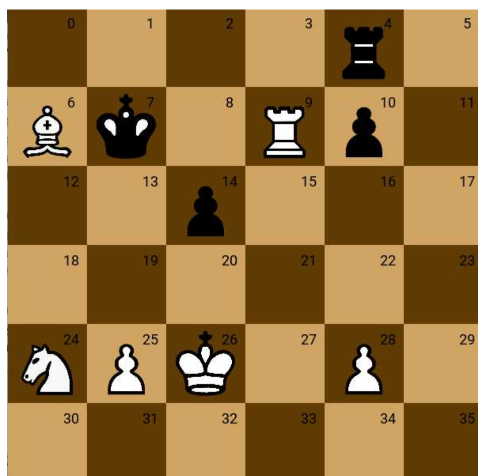
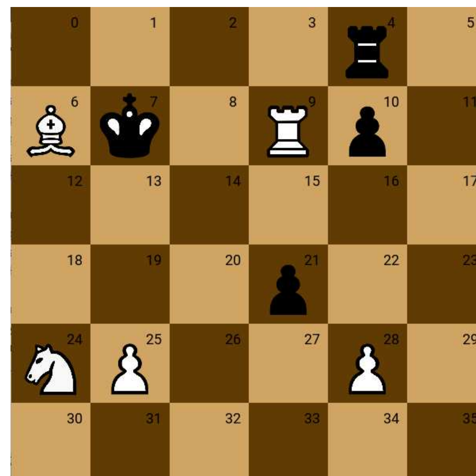


Figure 31: the network 35.338 from the session 1

However, as said before, there are still some serious errors. An example is shown in figure 32 by the network 33.919 from the session 9.

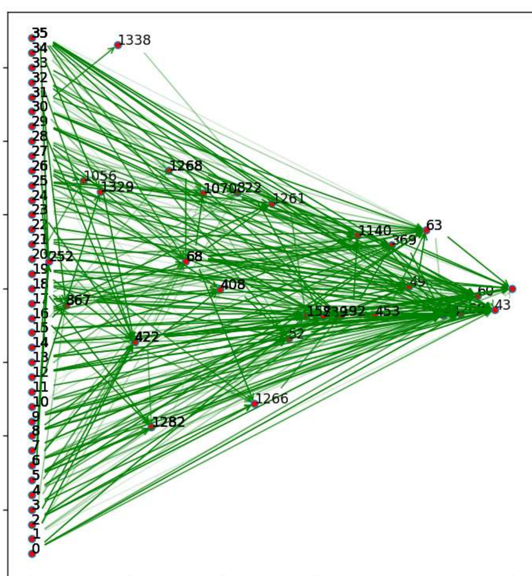


The AI plays as white, it's its turn



Beside having plenty of possible moves, it decides to move the king to sq. 21, where it's taken from the black pawn

Figure 32: the AI still not recognising the loss of the king as the worst move possible



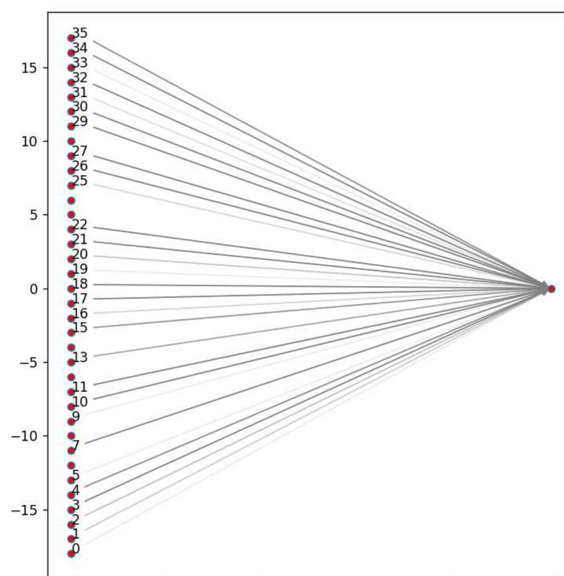


Figure 33: the network 33.919 from the session 9

It's worth noting that knights seem to be worth very little, perhaps because their optimal use is quite complex. Since the networks are not sufficiently trained, they may be unable to exploit it.

The focus was on training with low variability, as in this set networks appear to play better, perhaps because they explore smaller structures more precisely and mutations are not excessively random.

The parameter values used for the next set category and the results are shown in figures 34 and 35.

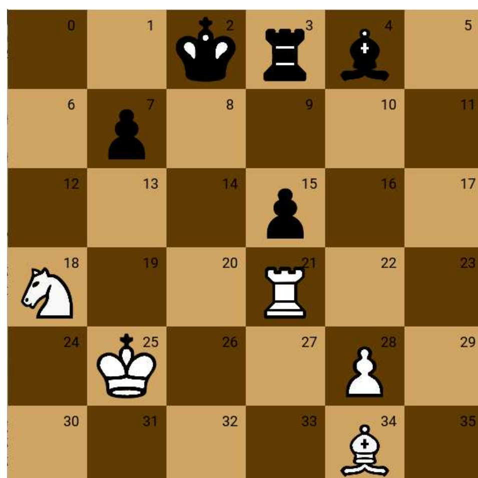
ID	Iter. (Depth)	Connec. Diff. Spec.	Weight Diff. Spec.	Add Node	Add Conn. - N° try	Bias / Weight Mut. Pow.
10	1000 (1)	0.9	0.1	0.1	0.1 - 5	0.2
11	1000 (1)	0.9	0.1	0.1	0.5 - 5	0.2
12	1000 (1)	0.9	0.1	0.1	0.9 - 5	0.2
13	1000 (1)	0.9	0.1	0.3	0.1 - 5	0.2
14	1000 (1)	0.9	0.1	0.3	0.5 - 5	0.2
15	1000 (1)	0.9	0.1	0.3	0.9 - 5	0.2
16	1000 (1)	0.9	0.1	0.5	0.1 - 5	0.2
17	1000 (1)	0.9	0.1	0.5	0.5 - 5	0.2
18	1000 (1)	0.9	0.1	0.5	0.9 - 5	0.2

Figure 34: the second set of training sessions

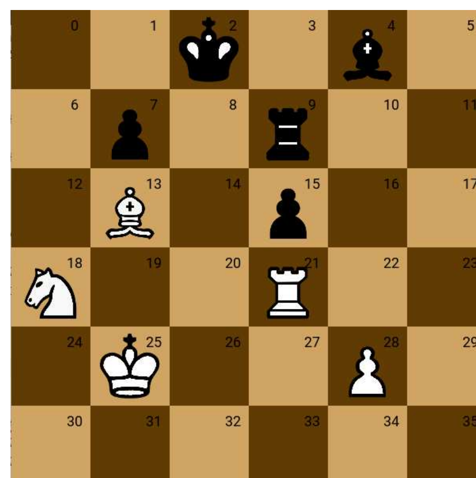
ID	Evaluation				
	SK:	MK:	SP:	MP:	TOT:
10	4	8	2	1	15
11	3	10	0	1	14
12	2	6	1	1	10
13	5	7	2	2	16
14	3	8	2	2	15
15	6	6	1	0	13
16	7	10	1	0	18
17	3	6	2	1	12
18	0	10	0	3	13

Figure 35: evaluation of the second set

The results haven't changed significantly; however, there has been a slight improvement. In particular, network 30.457 from session 16 has shown some promising behaviors.

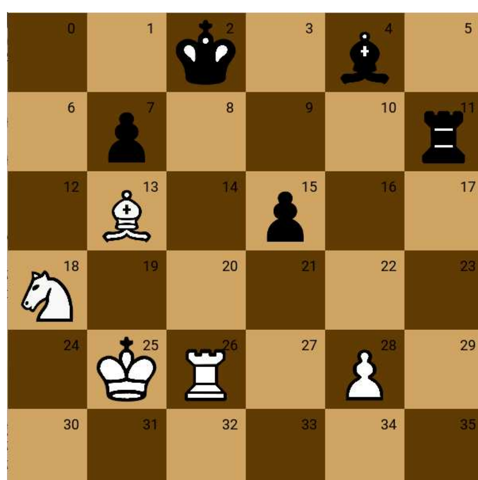


the AI plays as black, it's white turn

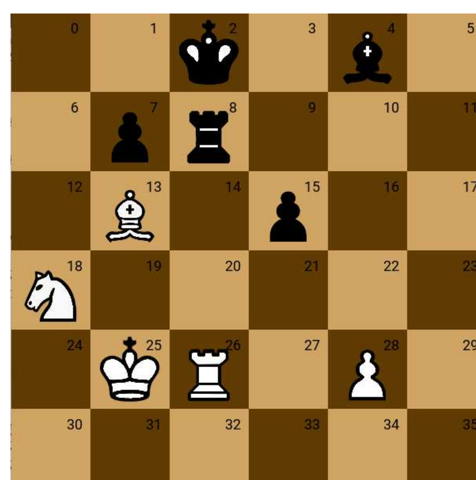


After the white bishop moved to sq. 13, the AI saves the rook moving it to sq. 9

Figure 36: the AI recognises the importance of the rook



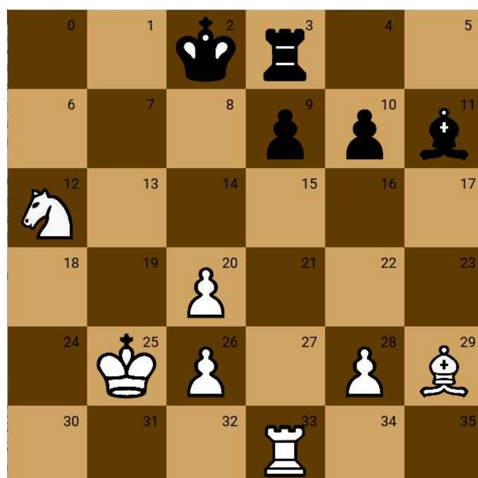
The AI plays as black, it's white turn



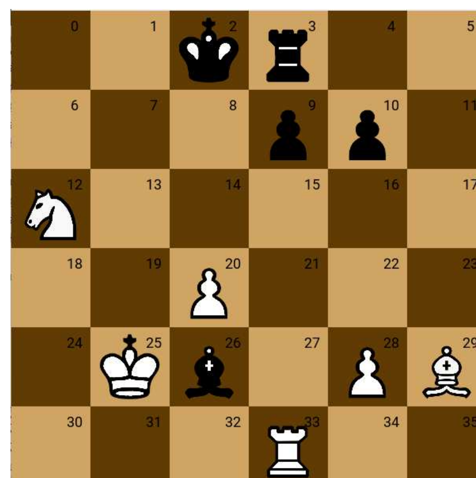
After the white rook threatened to capture the king, the black rook defends it by moving to sq. 8

Figure 37: the AI finds another way to defend the king rather than moving it

Still, some definitely strong pieces are exchanged with pawns, which are worth less, as shown in figure 38.



The AI plays as black, it's its turn



It decides to move the bishop to take the pawn at sq. 26, where the king can capture the bishop in response

Figure 38: the AI takes the pawn for a bishop, not recognising its value

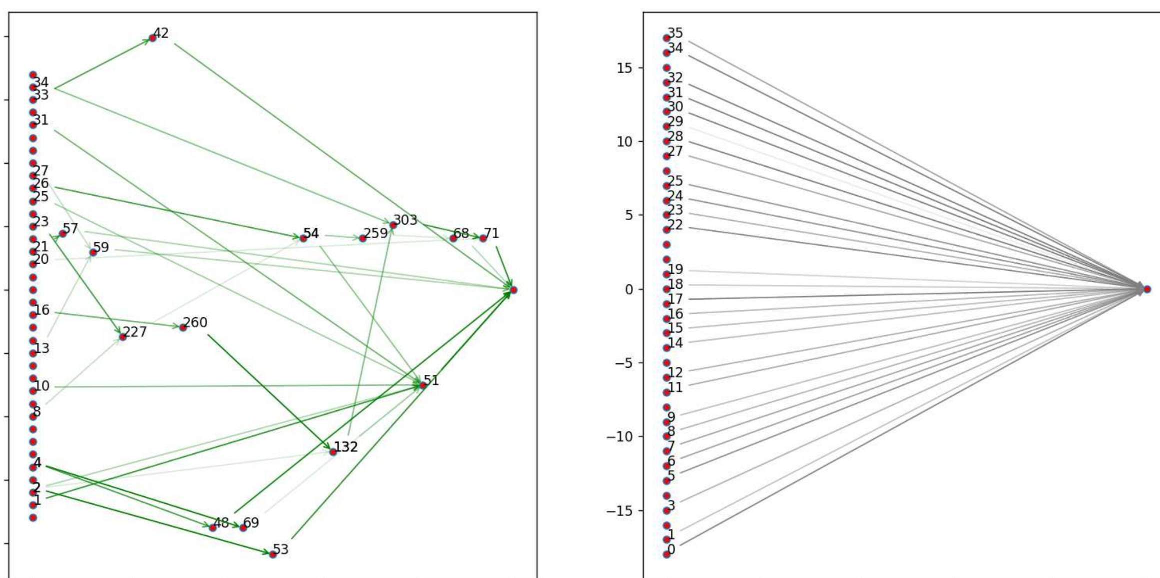


Figure 39: the network 30.457 from the session 16

While the level of play has increased, with an high rate of connection addition the networks do not seem to play better; indeed, they seem to perform worse.

Additionally, the problem of loop recognition and elimination cannot be overlooked, as it significantly increases training times.

For these reasons it was decided to focus on parameters with a low rate of connection addition. After few more tests, the number of training tournaments was increased from 1.000 to 2.000.

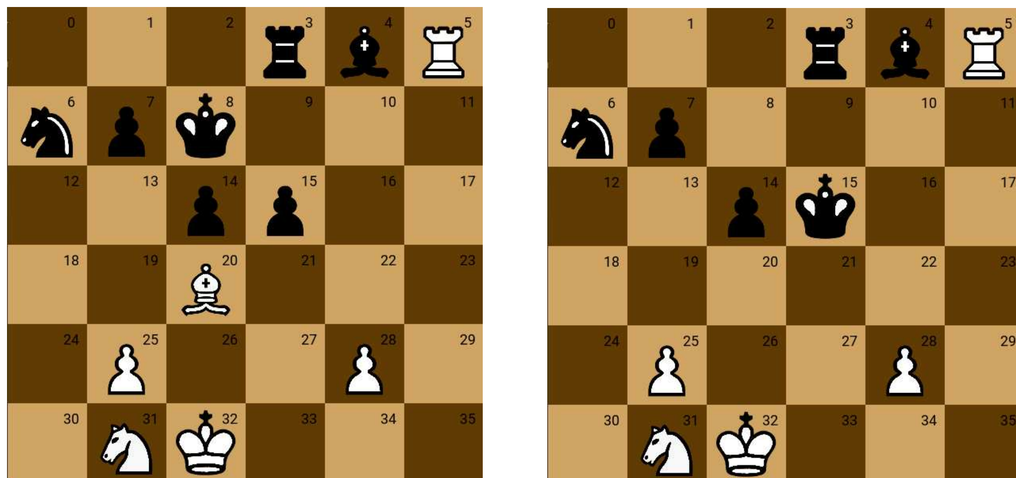
The data is shown in figures 40 and 41, followed by an example.

ID	Iter. (Depth)	Connec. Diff. Spec.	Weight Diff. Spec.	Add Node	Add Conn. - N° try	Bias / Weight Mut. Pow.
28	2000 (1)	0.9	0.1	0.1	0.1 - 5	0.2
29	2000 (1)	0.9	0.1	0.5	0.1 - 5	0.2
30	2000 (1)	0.9	0.1	0.5	0.9 - 5	0.2
31	2000 (1)	0.9	0.9	0.1	0.1 - 5	0.2

Figure 40: the sessions with 2.000 iterations

ID	Evaluation				
	SK:	MK:	SP:	MP:	TOT:
28	6	7	1	2	16
29	6	9	0	2	17
30	7	8	1	1	17
31	6	5	1	2	14

Figure 41: evaluation of the 2.000 iterations set



The AI plays as white, it's its turn

While the bishop is threatened, it chooses to sacrifice it for a pawn instead of saving it

Figure 42: the AI still appears to struggle with recognizing the mechanism of saving pieces other than the king

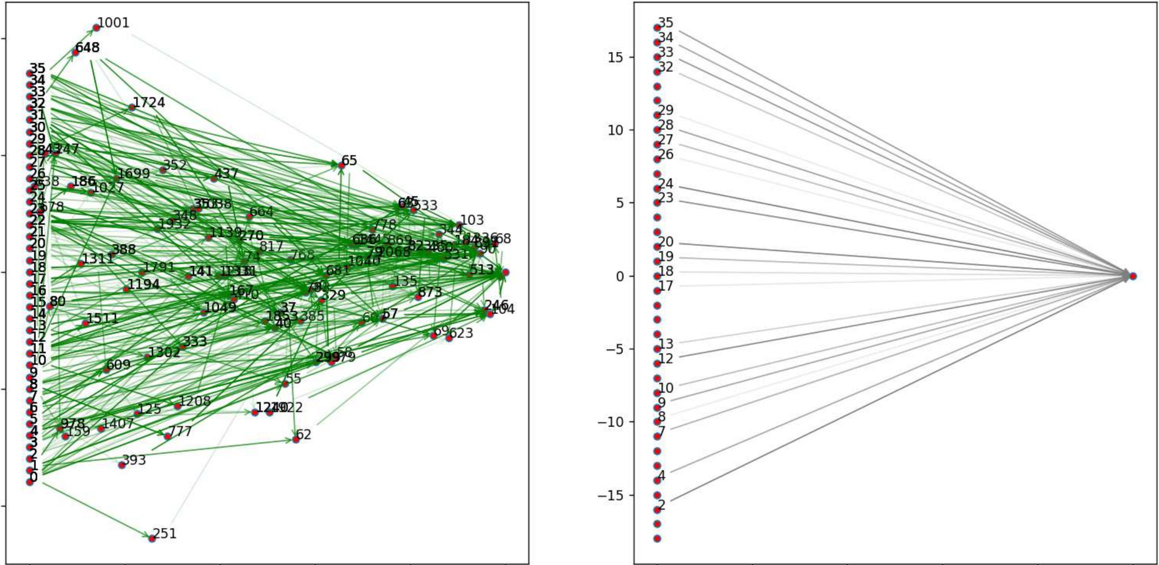


Figure 43: the network 75.953 from the session 31

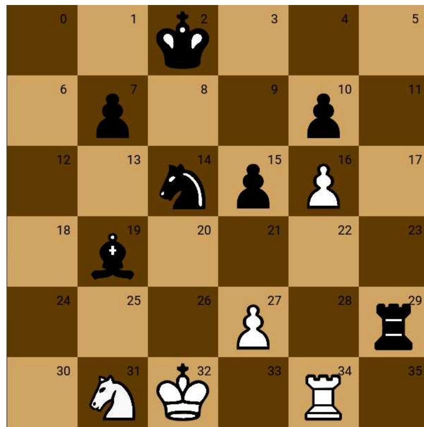
The results obtained prompted us to extend the training sessions to 4.000 iterations. However, the outcome achieved was unexpected.

ID	Evaluation				
	SK:	MK:	SP:	MP:	TOT:
34	0	8	2	0	10
35	3	6	2	0	11
36	2	6	1	1	10
37	4	5	2	1	12

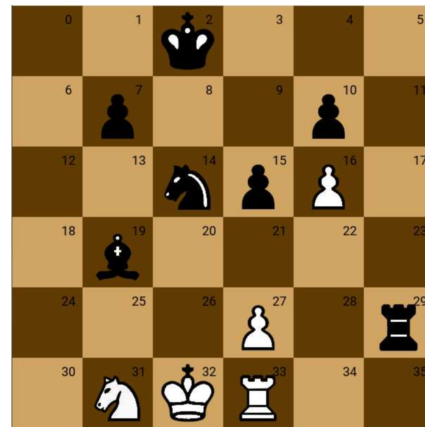
Figure 44: some examples from evaluation of the 4.000 iterations set

After an initial notable improvement curve, an unexpected decline in the performance of the neural networks was observed as the tournament iterations increased to 4.000. This outcome suggests that there might be a limit beyond which the quality of the model is no longer determined by the number of iterations, but rather by the proper parametrization of the system.

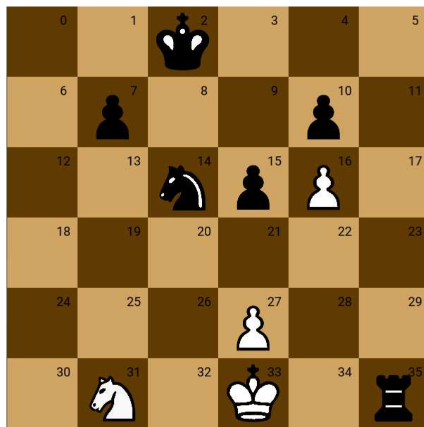
Two illustrative example are shown in figure 45 by the network 100.342 from session 35, followed by its corresponding representation.



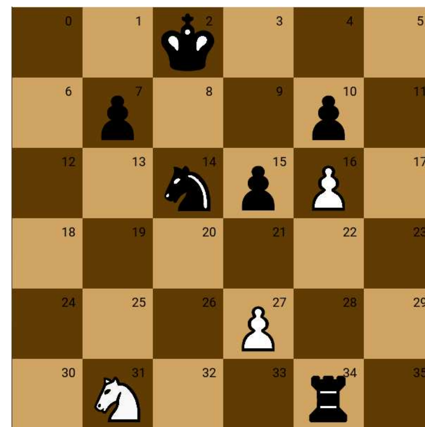
The AI plays as white, it's its turn



It moved the rook to sq. 33, where it can be taken by the black bishop

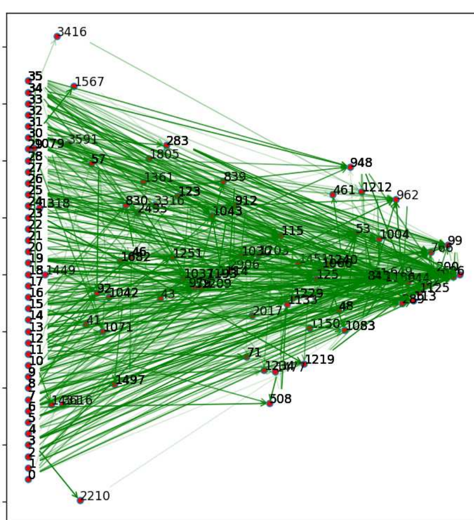


Few moves later, it's white turn



The AI moved the king to sq. 34, granting victory to the black

Figure 45: the AI committing very serious errors



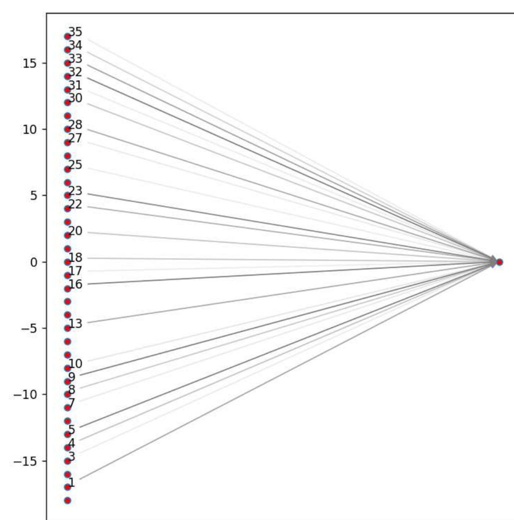


Figure 46: the network 100.342 from the session 36

6 Alternative to NEAT: tournament series

Given the promising yet cautiously optimistic results, it was decided to explore a modified algorithmic approach. Rather than advancing the network's evolution solely through the elimination of underperforming networks after each tournament, an alternative strategy was implemented: an attempt was made conducting multiple tournament sessions, during which such networks would undergo only weight and bias mutations rather than structural alterations, providing them time to adjust these values.

The objective of this strategy is to prioritize fine-tuning the network parameters, reducing reliance on optimization that may progress more rapidly but also more randomly across the various network prototypes.

The new function presents a straightforward and simplified approach to population management and parent selection during the training process. Initially, a training session is created, and a record manager is initialized to log results and monitor training progress.
During the training loop, tournaments are organized among the neural networks in the population. Like before, the networks are converted into game networks and paired to form challenger couples. These challenger couples participate in tournament series, the outcome of which is then used to evaluate network performance and assign scores. The genomes of the neural networks and their scores are then recorded for analysis and evaluation.

During the tournament series, parents are kept unchanged to serve as benchmarks, while the children are mutated. At this point, the process iterates with the next tournament until the completion of the tournament series.

Only after a series has concluded, for example 10 tournaments, the rankings are compiled, the least performing children are removed, and the species is repopulated with the children of the parents from the last tournament executed.

This new training method is characterized by its simplicity and linearity in approaching population management. While less complex than the first one, it still provides an effective approach to training neural networks through controlled reproduction and mutation. However, it certainly slows down the training times, and it is necessary to understand if and how much it is more effective.

The used parameters and the results are shown in figures 47 and 48.

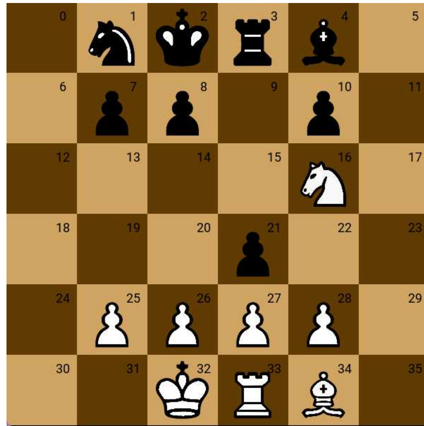
ID	Iter. (Depth)	Tournaments for series	Add Node	Add Conn. - N° attempts	Bias / Weight Mut. Pow.
TT 1	4000 (1)	20	0.1	0.1 - 5	0.2
TT 2	4000 (1)	20	0.1	0.1 - 5	0.5
TT 3	4000 (1)	20	0.3	0.5 - 5	0.2
TT 4	4000 (1)	20	0.3	0.5 - 5	0.5
TT 5	4000 (1)	20	0.3	0.5 - 5	0.8
TT 6	4000 (1)	20	0.5	0.9 - 5	0.2
TT 7	4000 (1)	20	0.5	0.9 - 5	0.5
TT 8	10000 (1)	20	0.1	0.1 - 5	0.2

Figure 47: the sessions with tournaments series algorithm

ID	Evaluation				
	SK:	MK:	SP:	MP:	TOT:
TT 1	6	10	0	4	20
TT 2	2	8	2	1	13
TT 3	3	5	2	4	14
TT 4	7	10	2	3	22
TT 5	7	10	1	1	19
TT 6	6	10	1	2	19
TT 7	5	7	1	1	14
TT 8	6	10	0	3	19

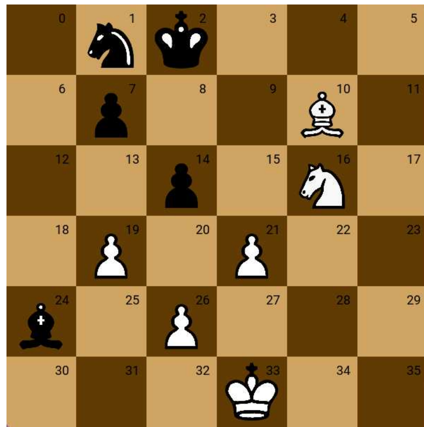
Figure 48: the series evaluations

An attempt was made to challenge the best performing AI from TT8 session, still trained at a depth of 1, using a depth of 4. The outcome was an AI capable of playing adequately, and effectively recognizing some of the best available combinations, as shown in figure 49.

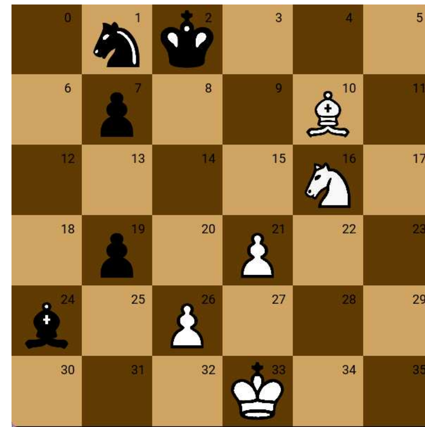


The AI plays as black, it looks like it's going to lose the rook

The AI answered taking the pawn in sq. 28, then taking the rook in sq. 33, without leaving white the chance to save it

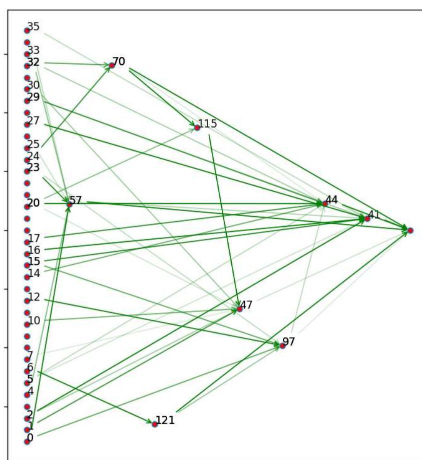


Few moves later it appears that the AI has the bishop trapped



Instead of moving the bishop or sacrificing it, the AI manages to save it by first taking the pawn in sq. 19

Figure 49: some highlights of the match



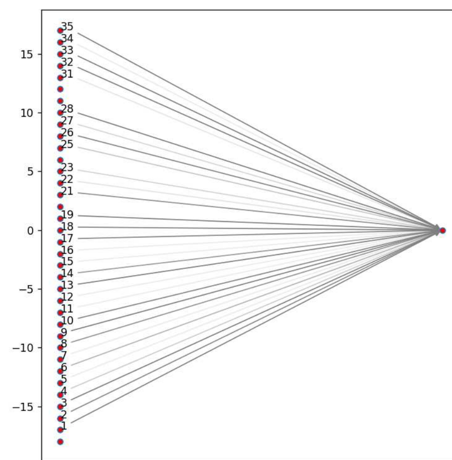


Figure 50: the network from tournament series representation

Observing the outcomes, it becomes evident that this method has resulted in slight yet discernible improvements.

It is noteworthy that conducting a session of 4.000 iterations with series of 20 tournaments entails 200 eliminations and subsequent replenishment of the population, equivalent to one-fifth of a session consisting of 1.000 iterations under the original NEAT strategy.

Given these considerations, it can be argued that through appropriate parametrization of the tournament series algorithm, it is conceivable to explore effectiveness curves that are likely superior to those observed previously.

7 Conclusions

The training sessions have provided valuable insights into the learning process of the neural networks within the NEAT framework. Despite the initial randomness and lack of strategy displayed by the untrained networks, successive iterations have showcased a gradual improvement in gameplay quality. Through a meticulous parameter selection process and careful observation of network behavior, key factors influencing the training outcome began to be identified.

The introduction of variability categories allowed for a comprehensive exploration of parameter combinations, revealing various effects on network performance. Lower variability settings appeared to yield more consistent and promising results, indicating that networks benefit from focused exploration of smaller structural changes. This insight guided the decision to prioritize training sessions with low variability, fostering more precise adaptation and strategic decision-making.

One notable challenge encountered during the training process was the recognition and elimination of loop structures within network genomes. The implementation of loop-solving mechanisms proved crucial in maintaining the integrity and functionality of evolving networks. By inhibiting connections prone to loop formation, potential disruptions to the training process were mitigated, ensuring continued progress towards near-optimal network architectures.

Furthermore, the evaluation of network performance revealed promising developments in strategic gameplay. Networks exhibited a growing understanding of game dynamics, showcasing improved decision-making and strategic prioritization. While certain weaknesses persisted, such as undervaluation of knight pieces, continued training iterations and parameter adjustments promise further refinement and optimization.

It is also crucial to take into account that for neural networks evaluating and therefore playing endgame scenarios remains significantly more challenging. Indeed, as the number of pieces decreases, the potential moves exponentially increase, resulting in progressively less accurate explorations from which to learn.

Nonetheless, there remains a scope for refinement through comprehensive parametrization investigations.
In particular, delving into the population size and the corresponding percentage of parents to select from the top-performing networks warrants attention. The speciation parameters remain among the most critical to identify, as they are pivotal for the proper functioning of network protection and evolution.
On the other hand, it is believed that the frequency and type of mutations depend not only on the NEAT algorithm but also on the type of game to which it is applied. In this case the high number of inputs has generated a minimally complex structure initially, making it difficult to parameterize without adequate study.

In conclusion, through iterative refinement and rigorous evaluation, it is believed that further increasing the effectiveness of a NEAT-based algorithm is possible, enabling the creation of an AI capable of efficient decision-making for generic board games.

References

[1] "Les automates". In: *La Nature* 2141-2152 (1914), pp. 56–62. URL: https://cnum.cnam.fr/redir?4KY28.87.

[2] *Lomosonov Tablebases*. URL: https://tb7.chessok.com/. (accessed: 18.04.2024).

[3] Claude E. Shannon. "Programming a Computer Playing Chess". In: *Philosophical Magazine* Ser.7, 41.312 (1959). URL: https://redirect.cs.umbc.edu/courses/graduate/CMSC671/fall15/resources/ProgrammingaComputerforPlayingChess.pdf.

[4] Herbert L. Anderson. "Metropolis, Monte Carlo, and the MANIAC". In: *Los Alamos Science* 14 (1986), pp. 96–108. URL: https://lib-www.lanl.gov/cgi-bin/getfile?00326886.pdf.

[5] Wei Hu. "Towards a Real Quantum Neuron". In: *Natural Science,* 10.3 (2018). DOI: https://doi.org/10.4236/ns.2018.103011.

[6] Kenneth O. Stanley and Risto Miikkulainen. "Evolving Neural Networks Through Augmenting Topologies". In: *Evolutionary Computation* 10.2 (2002), pp. 99–127. URL: http://nn.cs.utexas.edu/?stanley:ec02.

[7] Manuel Mariani and Simone Fiori. "Design and Simulation of a Neuroevolutionary Controller for a Quadcopter Drone". In: *Aerospace* 10.5 (2023). DOI: https://doi.org/10.3390/aerospace10050418.

[8] *Sigmoid function*. URL: https://en.wikipedia.org/wiki/Sigmoid_function. (accessed: 18.04.2024).

[9] *Alpha–beta pruning*. URL: https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning. (accessed: 18.04.2024).

[10] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. "Exploration and Exploitation in Evolutionary Algorithms: A Survey". In: *ACM Computing Surveys* 45.3 (2013). DOI: https://doi.org/10.1145/2480741.2480752.