Is using a Minimax algorithm better than using self-play to train a neural network to play Connect 4?

GABRIEL MANHAS VIEIRA

Table of Contents

Introduction

For a long time people taught that computers wouldn’t be able to beat humans in complex games such as chess and Go, once to calculate all the possibilities in those games it would take a very long time, but this dogma has been broken since 1997 when a IBM computer could beat the former world chess champion Garry Kasparov[1]. Since then computers have evolved and a new way of machine learning has emerged, artificial neural networks.

Machine learning was defined in the 1950s by Arthur Samuel as “the field of study that gives computers the ability to learn without explicitly being programmed.”. According to Mikey Shulman, a lecturer at MIT Sloan and head of machine learning at Kensho, this definition still holds true, and machine learning models continue to develop themselves[2].

This paper aims to compare two different training methods of reinforcement learning models and discover which is better in learning to play the game called Connect 4. The first method trains the model playing it against itself, what is called self-play, and therefore evolves without the use of any other agent. The second method trains against an algorithm called Minimax, that performs calculations to choose optimal moves. Therefore the purpose of this investigation is to answer the question “Is using a Minimax algorithm better than using self-play to train a neural network to play Connect 4?”.

1 *Deep Blue* (no date) *IBM100 - Deep Blue*. Available at: https://www.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/ (Accessed: March 10, 2023).

2 Brown, S. (2021) Machine Learning, explained, MIT Sloan. Available at: https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained (Accessed: March 10, 2023).

Nowadays, artificial neural network models are widely used in a variety of situations such as games, security systems, autonomous vehicles and image-based medical diagnosis[2]. Therefore studying the efficiency of different methods for training artificial neural network models is important to improve the efficiency of training models that are used in many fields in our society.

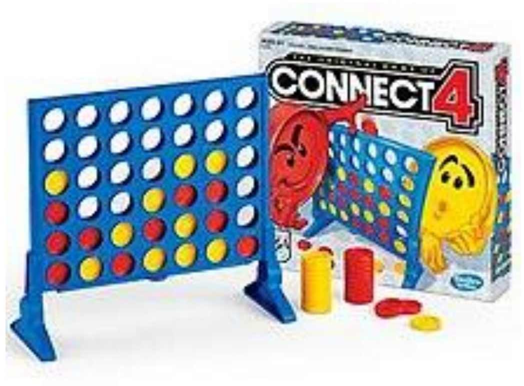

Figure 1: Connect 4 board[3]

The game Connect 4 has a midterm level of complexity, with 1.6 x 10^{13} possible actions[2]. It is more complex than games like Tic Tac Toe that can be easily solved by simple algorithms, demanding the use of an artificial neural network (ANN) for this project. On the other hand, it is less complex than games like chess or Go, minimizing training time and computational resources for our research.

Connect 4 is a two-player strategy game. Each player alternatively drops his own-colored discs in one of the seven columns, as shown in figure 1. The goal is to connect 4 discs in a row, column or in the diagonal.

[3] Hasbro (2015) Connect 4 game board and box. Available at: www.hasbro.com (Accessed: 2023).

The game seems simple but hides a lot of strategy. Research shows that middle slots are more important than side ones. Scientists proved that, if the first player places his first disc in the middle column, he cannot lose on a perfect play[4].

Background Information

Machine Learning

Machine learning are models created to recognize patterns and make decisions based on an input. It mimics the way humans learn to make a machine learn by itself. Machine learning is a subfield of artificial intelligence, which is broadly defined as the capability of a machine to imitate intelligent human behavior. Artificial intelligence systems are used to perform complex tasks in a similar way that humans solve problems.

Machine learning can be divided in 3 main subcategories, depending on the way that the model trains[2]. The first is supervised data, where the model trains with already labeled datasets, and learns to label this data. A classic example is a model trained to classify images of dogs and cats. The second type is unsupervised learning where the model tries to find patterns in data, where humans are not able to.

The third type of category is reinforcement learning, that will be used for this research. Reinforcement learning is a type of machine learning where an agent learns to make decisions through trial-and-error interactions with an environment. The agent takes actions in the environment, and the environment provides feedback in the form of rewards or punishments, which the agent uses to adjust its behavior.

Artificial Neural Network

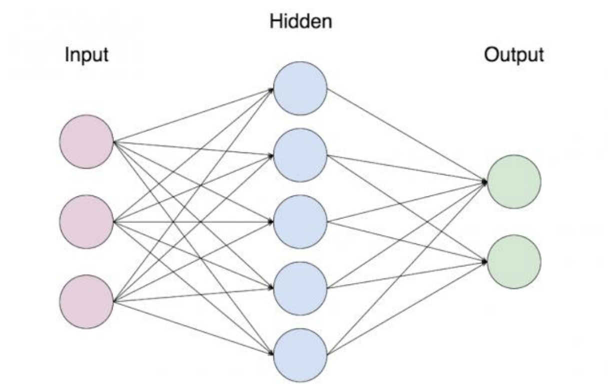

Figure 2: Simple fully connect ANN scheme4

Artificial Neural Networks (ANN) are a type of machine learning models, defined as "a family of information processing systems inspired by the biological nervous system, which are capable of learning from examples" (Bishop, 2013)[5]. Bishop goes on to explain that ANNs are organized in layers with interconnected neurons that receive inputs from previous layers and transform it by using a set of weights that can be updated, and pass the output for the next layers.

TensorFlow and Keras

TensorFlow is an open-source library, created by google, that provides machine learning functionality to create machine learning models[6]. Keras is a user-friendly

[4] Burnett, C. (2006) Illustration of the topology of a generic Artificial Neural Network (ANN). Available at: https://commons.wikimedia.org/wiki/File:Artificial_neural_network.svg (Accessed: 2023).

[5] Bishop, C.M. (2013) Neural Networks for Pattern Recognition. Oxford: Oxford University Press.

[6] Vaughan, J. (2018) What is tensorflow?: Definition from TechTarget, Data Management. TechTarget. Available at: https://www.techtarget.com/searchdatamanagement/definition/TensorFlow (Accessed: March 10, 2023).

open-source neural network library written in Python. That provides a high-level Application Programming Interface (API) to build and train neural networks, without worrying about the low-level details of the underlying implementation[7].

In this project Keras was used together with TensorFlow to create the model that was used to perform the experiments.

Minimax Algorithm

Minimax is a kind of backtracking algorithm that is used in decision making and game theory. It finds the optimal move for a player, assuming his opponent also plays optimally. It is widely used in two-player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, chess, etc. Minimax algorithm uses a depth value, that refers to the number of levels or moves the algorithm searches ahead in the game tree to determine the best move[8].

In Minimax the two players are called maximizer and minimizer. The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible.

Every board state has an associated value. In a given state, if the maximizer has the upper hand, the score of the board will tend to be some positive value. When the minimizer has the upper hand, the board state value will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for each type of game.

[7] DeepAI (2019) Keras, DeepAI. DeepAI. Available at: https://deepai.org/machine-learning-glossary-and-terms/keras (Accessed: March 10, 2023).

[8] Minimax algorithm in Game theory: Set 1 (introduction) (2022) GeeksforGeeks. Available at: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/ (Accessed: (March 10, 2023).

Self-play

Self-play is a method of training an artificial neural network (ANN) in which the network plays against itself. This approach is commonly used in reinforcement learning tasks, where the goal is to learn a policy that maximizes a reward signal[9].

In the context of playing games like chess, Go, or Connect 4, self-play involves having the ANN play against a copy of itself that uses the current version of the policy. This allows the network to learn from its own mistakes and refine its strategy over time. The idea is that the network can discover new and better strategies by exploring the space of possible moves and analyzing their outcomes.

Methodology

Independent variables

- Training method: self-play vs. Minimax

Dependent variables

- Model performance: measured in terms of the win rate against a random player.

Controlled variables

- Neural network model/architecture: The same model is used for both training methods;

[9] Plaat, A. (2020) "Self-play," *Learning to Play*, pp. 195–232. Available at: https://doi.org/10.1007/978-3-030-59238-7_7.

    - Activation function (ReLU)
    - Cost function (categorical cross-entropy loss function)
    - Optimizer (Adam optimizer)
    - Number of neurons in each hidden layer (64 in the first and 32 in the second)
    - Number of hidden layers (2)
    - Number of outputs (1)
- Number of games per generation: the number of games the model plays (1,000)
- Number of generations each model trained for: the number of generations the model plays (1,000)
- Learning rate: the rate at which the weights of the neural network are updated during training (2.0)
- Randomness: the percentage of times when the neural network perform a random action (0.5)
- Minimax depth: the depth wich Minimax can analyze (2)

The ReLU activation function was chosen because it allows the model to learn complex non-linear relationships between the input and output, while also being computationally efficient to evaluate.[10]

[10] Praharsha, V. (2022) Relu (rectified linear unit) activation function, OpenGenus IQ: Computing Expertise &amp; Legacy. Available at: https://iq.opengenus.org/relu-activation/#:~:text=limited%20learning%20capacity.-,What%20is%20ReLU%20%3F,otherwise%2C%20it%20will%20output%20zero. (Accessed: March 10, 2023).

The Adam optimizer function was chosen because it is a well-established and effective optimization algorithm for deep learning tasks, and it allows the model to learn the optimal weights efficiently and effectively during training.[11]

The number of neurons in the hidden layers and the number of hidden layers, was chosen based on experimentation with different architectures, with the aim of achieving a good balance between model complexity and performance.

The number of games per generation and the number of generations each model trained for, was chosen intuitively, thinking that the space of possible actions is large, and the feedback given from each game is limited.

The Minimax depth was chosen based on the time needed to train the models. Since higher depth values increase the training time, the lowest possible depth was used.

Methodology Development

In the beginning of this investigation, I was planning on training the model against the Minimax until it won more than 50% of the games. Because the space of possible actions was so large, the model took longer than I expected to learn and become better. Also, the time needed to train each model was much higher than what I thought it would be, taking a day to train 1000 generations.

A generation was set as a session of 1000 games, that at the end of the session, processes the feedback of the 1000 games played through the model. It's called generation because it refers to the amount of training the model has passed

[11] Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

through, meaning that a model that’s in its 1000th has played 1,000,000 games. Because of this I chose to train each model for 1000 generations, since with this amount of games played the model showed some progress that could be analyzed.

For this I created the environment where the models should be tested and built the model. To create the first model I talked to a computer scientist, who graduated from the University of São Paulo, who provided me with sources to create a ANN model. Using these sources I built a model using TensorFlow 1.0, using a structure of 4 hidden layers with 26 neurons each. But after doing some testing, I observed that the model was very slow and that after a certain amount of generations the model was beginning to crash. After researching more, I came to the conclusion that I should update the model to use TensorFlow 2.0 and that a structure with less hidden layers would best fit my objective. Then I created a second model using 2 hidden layers with 64 and 32 neurons.

To test the development of each training method, after training each of them for 1000 generations, I tested them against a random player. I did the tests making each of them play 1000 games against the random player, splitted in 10 sessions of 100 games each. For making sure that the results aren’t biased and didn’t happen only for a chance, since training against a random player present is a random factor. Also for having a control result, I tested the model untrained against a random player for 1000 games, also splitted in 10 sessions of 100 games each.

Results

<u>Model trained against Minimax</u>

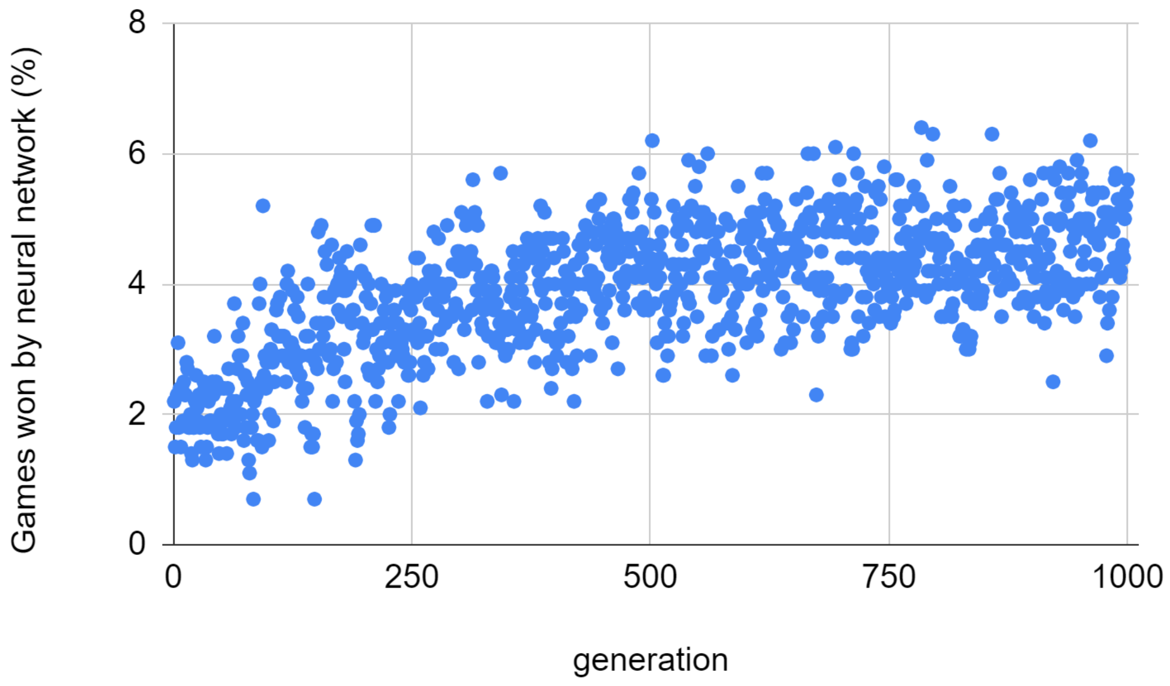

Figure 3: Graph showing the percentage of games won by the neural network against the Minimax during training.

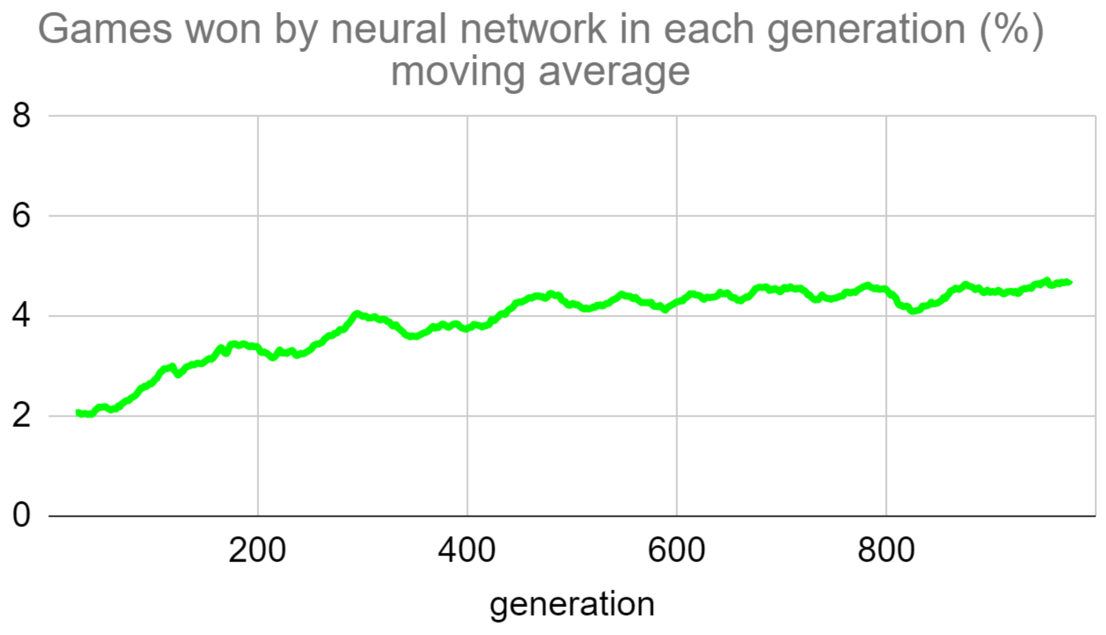

Figure 4: graph showing the moving average, 50 to 50, from the percentages of games won by the neural network in each generation, while training against the Minimax algorithm.

In figure 4, we have the moving average, considering the last 50 generations, from the training data of the model trained against Minimax. I choose to show the moving average to be able to analyze the performance of the model in a cleaner way, since analyzing the figure 3 graph, it's not possible to understand the model performance.

As seen in figure 3 and 4, the model trained against the Minimax obtained a max winning rate against Minimax of 6.5%. The average winning rate against Minimax in the first 50 generations was 2.07%, and in the last 50 generations was 4.74%. Therefore we can understand that after 1000 generations of training, the model was able to increase its first 50 average winning rate against Minimax by 128%.

Model trained using self-play

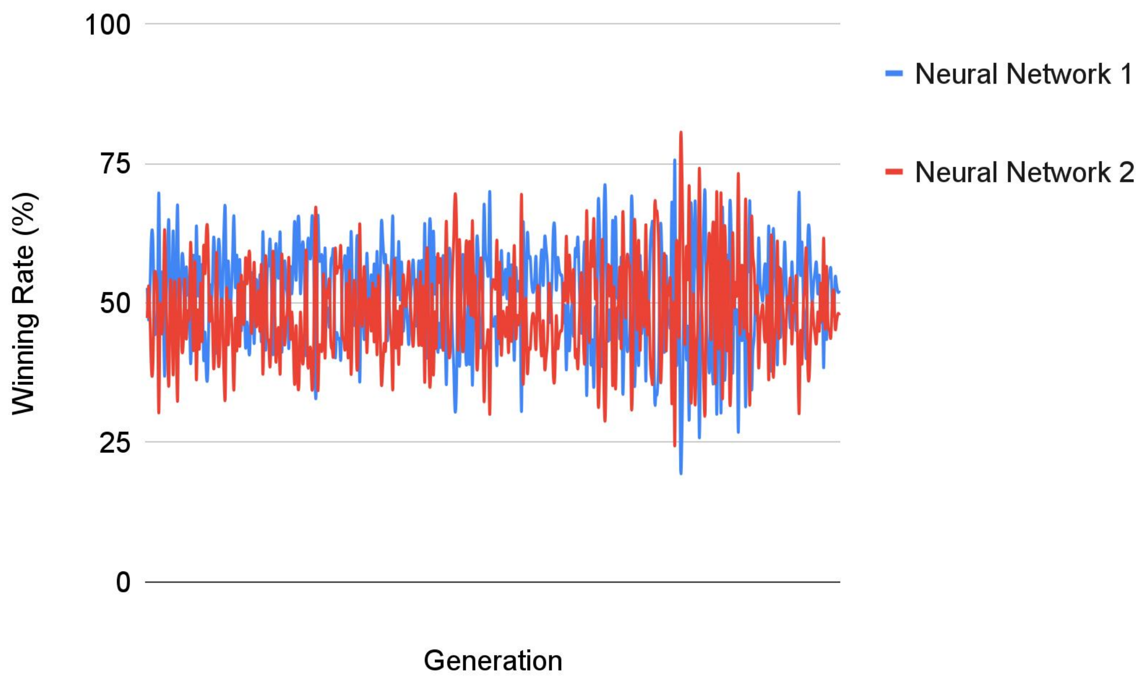

Figure 5: Winning game rate per generation from neural network 1 (blue) and 2 (red), while training against each other

In figure 5, we can see that the model that wins more games each generation varies a lot, almost every generation.

Testing results

Session	Games won by neural network trained using Minimax in each session (%)	Games won by neural network trained using self-play in each session (%)	Games won by neural network untrained in each session (%)
1	96	74	51
2	95	74	60
3	91	78	59
4	97	74	59
5	97	77	47
6	95	79	56
7	99	70	61
8	95	80	56
9	97	71	59
10	94	77	59
Average	95,6	75,4	56,7

Figure 6: Table showing the data of the tests done with each trained model against a random player, and the test of the untrained model against a random player, control group

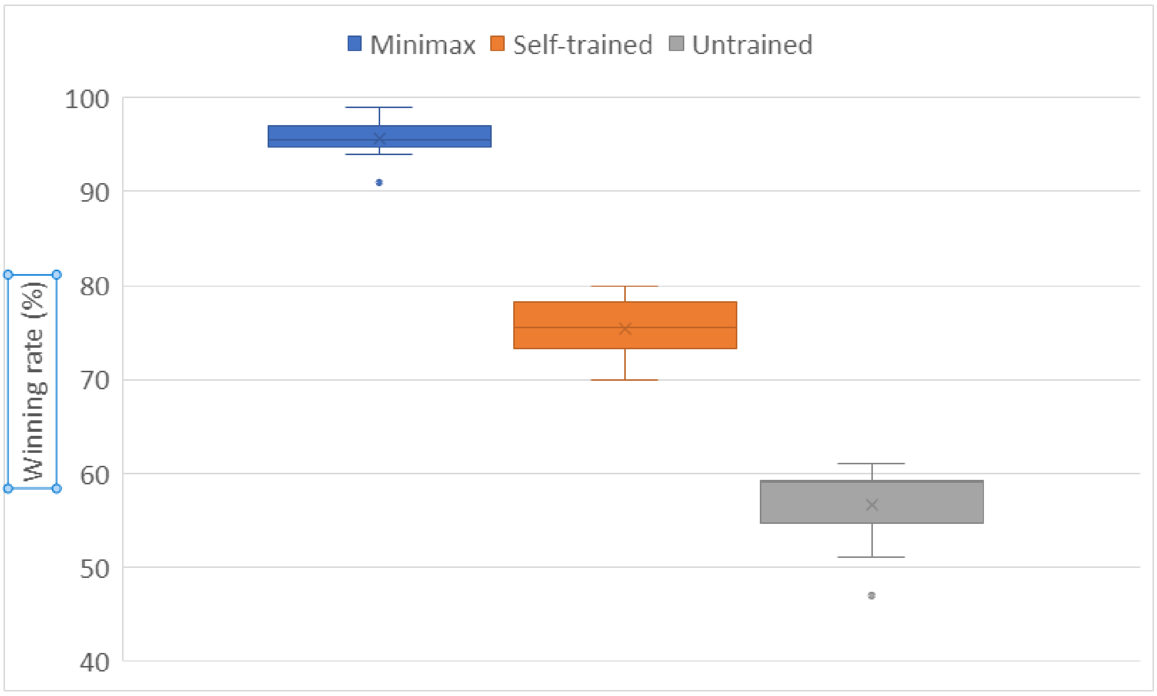

Figure 7: graph comparing the results of each model against a random player

The results from testing the model trained with Minimax against a random player, as shown in figure 7, are: Maximum winning rate per session of 99%; Average winning rate was 95.6%.

The results from testing the model trained with self-play against a random player, as shown in figure 7, are: Maximum winning rate per session of 80%; Average winning rate per session of 75.4%.

The results from testing the untrained model against a random player, as shown in figure 7, are: Maximum winning rate per session of 60%; Average winning rate per session of 56.7%.

Analysis and evaluation

Overall the model trained against Minimax achieved better results when testing against the random player than the model trained using self play.

Model trained against Minimax

For training the model against Minimax, the depth value used was 2, meaning that the algorithm will search only 2 moves ahead in the game tree to determine the best move. This is the smallest possible depth value, since 1 would mean it would only analyze the current position, and wouldn't analyze anything. But even using these settings, the model still couldn’t win more than 6.5% of the games. This shows that the model has still a lot to be developed. We can understand that a possible explanation for that is because of the large number of possible actions in connect 4, and that, compared to the possible action, very little feedback is given from each game. And because of that the model takes a lot of training to develop.

Analyzing the figure 3 graph, we can see that during all the process of training the spreadness of the data was high, meaning that close generations got very different results, as an example we can see that the highest winning rate was at generation 784, with a win rate of 6.4%, but only 5 generations later, on generation 789, there was already a much lower winning rate of only 3%. We can associate this to the fact that randomness is applied to the neural network moves, what can lead to bad decisions, and can impact the results of that generation.

Also the data seems to increase at a lower rate at the end. But by analyzing the moving average line of this graph, shown in figure 4, we can understand that the winning rate doesn’t go up in a linear way, and that it takes “steps” by going down a bit and then going higher than before, and that this pattern repeats. This can be caused by the way that it “learns”, once that when it becomes better at the game it encounters new strategies played by the Minimax algorithm and it has to develop new strategies itself to counter that, that’s why the random factor is important. At the end it seems to have stabilized but we can hypothesize that it was doing one of its “steps” and that after that it would go up again.

Analyzing the average winning rate from the first 50 generations, compared to the average winning rate from the last 50 generations, we can understand that the model trained was able to increase its average results by 128%, showing that the model was able to learn and evolve during the process.

Model trained using self-play

The figure 5 graph shows a high variation, and because of that it may be analyzed as a noise that doesn’t mean anything. But we can hypothesize that this happens because of the way the code is structured, it only processes the feedback of each game at the end of the generation, and since each generation has a 1000 games, the feedback that is processed is large. Therefore a possible explanation for the high variation is that because the process of the feedback happens at the end of the generation, the model that has lost more games, has a lot of data on the strategies used by the other one, and because it is losing it needs to evolve. While the other model doesn't need to evolve since it is winning more games. And because of that the model that was losing has a big advantage over the other one, which can make it win more games in the next generation.

Also, another important concept regarding the self-play method is that, both agents start from scratch and therefore at the beginning both agents play as random players. Because neither model has a source to learn from, it takes longer to both become better, since both agents need to create strategies from scratch.

Comparison between the 2 models results

Overall both models were able to evolve in comparison to the untrained one, but the model trained against the Minimax was able to get better results having the same amounts of games played.

Analyzing the data shown in figure 6 and 7, we can understand that the model trained against Minimax was able to win on average 95.6% of the games against the random player, while the model trained with self play could only win, on average, 75.4% of them. Since the random player doesn’t take in consideration any parameters from the board it was expected for the models to easily beat it, therefore we can understand that results closer to 100% were expected. We may hypothesize that an explanation for this is that neither of the training methods were able to present to the model enough diversity for the models to create good strategies. We can understand that maybe reinforcement learning may not be the best approach, or at least not the fastest approach to solve the game.

Also another concept that may have affected the results is that the training may have become repetitive on both models, since both agents, minimax and player 2 model in self-play, may have stuck to similar approaches during the whole training, which may have caused the models to be trained only for specific scenarios. This phenomenon, in machine learning, is called overfitting.

Overfitting is a common problem in machine learning and occurs when a model is trained only on specific scenarios, to the point that it begins to memorize the training data rather than learn the underlying patterns and generalize to new, unseen data. This means that the model may perform well on specific scenarios, but perform poorly on new, unseen data[12].

Also another factor that may have influenced the training and made the model that trained against Minimax get higher winning rates quickly, was that Minimax was able to give a more diverse and robust training and feedback. Since the Minimax algorithm is designed to find optimal moves for each scenario, it was able to provide different strategies to the model. Additionally, training the model against itself (self-play) may have done the opposite and not provided enough diversity in its gameplay.

Another aspect may be that the model trained against self-play would take longer to learn than the model trained against Minimax. Since training using self-play means both agents don't know how to react to any scenarios and have to learn it from scratch. Which implies that both models have to learn. But when training against Minimax just the model has to learn and the opponent always reacts with optimal moves, meaning that only the model has to learn. Because the training had a relatively small amount of games to learn from, it may be an explanation why the model playing against Minimax got better results.

[12] Brownlee, J. (2019) Overfitting and underfitting with machine learning algorithms, MachineLearningMastery.com. Available at: https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/ (Accessed: March 10, 2023).

Limitations of Investigation

1. The amount of generations that could be simulated were low compared to the space of possible action of the connected 4 games. This happened mainly because of the time needed to train each model in my computer. This was a limitation because I couldn’t get to the point where the models were able to beat Minimax, if this was ever going to happen. Also because of this I was unable to understand if the model trained self-play was ever to get better than the one trained against the Minimax, as has happened in other games. Because of this, simulating more games would be important in further investigation, and having a better computer and more time would make it easier.
2. The optimization of the code was another limitation. Because i’m not graduated in any courses in evolving programming, my code was not even close to being the most optimal ever. Because of this the time needed to train my model could get significantly lower, which would make the investigation better. Because of this analyzing the whole code again and trying to make it more optimal would be important in further investigation.
3. Investigating how the hyperparameters affect the training would be important as well. Since I had a limitation of words and time I was not able to investigate hyperparameters a lot, and used numbers that i choose intuitively, but is known that this change a lot the training results, and comparing the methods of training using different hyperparameters would be important to understand

the difference of the two methods. Because of this doing testing on the hyperparameters would be important for further investigation

Conclusion

In conclusion, the results showed that training an ANN model using the Minimax algorithm for a 1000 generations is better than training an ANN model using self-play for a 1000 generations. But we have to take into consideration the limitations of each method. Training against the Minimax algorithm has a limitation that because the Minimax algorithm cannot solve the game, the model won't be training against the best player and therefore won't its training knowledge will be limited to the knowledge of the minimax algorithm. Opposed to training against using self-play, where both agents are continuously evolving and the knowledge won't be limited. On the other hand, because training using self-play both agents has to evolve, it takes longer for the models to evolve, and therefore training against minimax is faster and requires a smaller amount of games.

Therefore we can understand that this research showed that there is no better method of training, since each method has its pros and cons. And understanding those are important and should be taken into consideration depending on the objective of the training.

Bibliography

Alderton, E., Koffman, J. and Wopat, E. (no date) “Reinforcement Learning for Connect Four,” Stanford Edu [Preprint]. Available at: https://web.stanford.edu/class/aa228/reports/2019/final106.pdf (Accessed: 2023).

Deep Blue (no date) *IBM100 - Deep Blue*. Available at: https://www.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/ (Accessed: March 10, 2023).

Brown, S. (2021) *Machine Learning, explained*, *MIT Sloan*. Available at: https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained (Accessed: March 10, 2023).

Bishop, C.M. (2013) *Neural Networks for Pattern Recognition*. Oxford: Oxford University Press.

Vaughan, J. (2018) *What is tensorflow?: Definition from TechTarget*, *Data Management*. TechTarget. Available at: https://www.techtarget.com/searchdatamanagement/definition/TensorFlow (Accessed: March 10, 2023).

DeepAI (2019) *Keras*, *DeepAI*. DeepAI. Available at: https://deepai.org/machine-learning-glossary-and-terms/keras (Accessed: March 10, 2023).

Minimax algorithm in Game theory: Set 1 (introduction) (2022) *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/ (Accessed: March 10, 2023).

Plaat, A. (2020) “Self-play,” *Learning to Play*, pp. 195–232. Available at: https://doi.org/10.1007/978-3-030-59238-7_7. (Accessed: March 10, 2023)

Praharsha, V. (2022) *Relu (rectified linear unit) activation function*, *OpenGenus IQ: Computing Expertise & Legacy*. OpenGenus IQ: Computing Expertise & Legacy. Available at: https://iq.opengenus.org/relu-activation/#:~:text=limited%20learning%20capacity.-,What%20is%20ReLU%20%3F,otherwise%2C%20it%20will%20output%20zero. (Accessed: March 10, 2023).

Brownlee, J. (2019) *Overfitting and underfitting with machine learning algorithms*, *MachineLearningMastery.com*. Available at: https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/ (Accessed: March 10, 2023)

Appendix A – Neural network code

```
import numpy as np
import random
import tensorflow as tf
import os
import connect4
import sys


class NeuralNetworkB:
   def __init__(self, player):
       # Define the model architecture
       self.model = tf.keras.Sequential([
           tf.keras.layers.Dense(64, activation='relu', input_shape=(42,)),
           tf.keras.layers.Dense(32, activation='relu'),
           tf.keras.layers.Dense(7, activation='softmax')
       ])

       # Compile the model with a loss function and an optimizer
       learning_rate = 2.7
       self.model.compile(loss="sparse_categorical_crossentropy",
optimizer=tf.keras.optimizers.Adam(learning_rate))

       self.player = player

       self.folder_path = 'saved_graphs' + str(self.player)

       model_path = os.path.join(self.folder_path, "connect4.h5")
       if not os.path.exists(model_path):
           print("Graph cannot be restored")
       else:
           self.model = tf.keras.models.load_model(model_path)
           print("graph has been restored")

       if player == 1:
           self.model1 = tf.keras.models.clone_model(self.model)
       else:
           self.model2 = tf.keras.models.clone_model(self.model)

       self.number_of_games = 0

   def __enter__(self):
       return self

   # Play a game of Connect 4 between two players
   def play_game(self, player1, player2):
        board = np.zeros((6, 7))  # Initialize the board
        player = 1  # Player 1 starts
        game_over = False
        while not game_over:
            if player == 1:
                player = player1
            else:
                player = player2
            # Get the next move from the player
            if callable(player):  # The player is a function
                move = player(board, player)
            else:  # The player is a model
                move_probs = player.predict(board.reshape(1, -1))[0]
                move = np.argmax(move_probs)
            # Check if the move is valid
            if board[5][move] == 0:
                for row in range(5, -1, -1):
                    if board[row][move] == 0:
                        board[row][move] = player
                        break
                # Check if the game is over
                # ...
                player = -player  # Switch to the other player
        # Return the final board state and the winner
        return board, np.sign(player)

    def main(self, board, depth, column_count, row_count):
        if self.player == 1:
            player = self.model1
        else:
            player = self.model2

        legal_moves = connect4.get_legal_moves(board, 7, 6)

        move_probs = player.predict(board.reshape(1, -1))[0]
        move = np.argmax(move_probs)

        legal = False
        for x in legal_moves:
            if x == move:
                legal = True
                break

        while not legal:
            move_probs = player.predict(board.reshape(1, -1))[0]
            move = np.argmax(move_probs)

            for x in legal_moves:
```

```
            if x == move:
                legal = True
                break

        y = 1
        move_probs = np.argsort(move_probs)[::-1]
        while not legal:
            #print("neural net B has choosen a full collummn: " + str(move))
            move = move_probs[y]

            for x in legal_moves:
                if x == move:
                    legal = True
                    break
            y += 1

    #print(move_probs)
    #print("move: " + str(move))

    return move

def save_model(self):
    if not os.path.exists(self.folder_path):
        os.makedirs(self.folder_path)
    model_path = os.path.join(self.folder_path, "connect4.h5")
    self.model.save(model_path)
    print("Gen Graph Saved")

def game_feedback(self, board, winner):
    self.number_of_games += 1

    # Update the model based on the outcome of the game
    X = [board.reshape(-1)]
    y = [winner]
    self.model.fit(np.array(X), np.array(y), epochs=1, verbose=0)

    if self.number_of_games % 100 == 0:
        self.save_model()

def __exit__(self, type, value, traceback):
    pass
```