Exploring Machine Learning Through Tic-Tac-Toe

A Computer Science Project By:

Theodore Tasman (PSU ‘25)

&

Benjamin Rodgers (RPI ‘26)

https://github.com/tedtasman/tic-tac-joe

Introduction:

Computer science is a field undergoing rapid and expansive advancements, with limitless potential. Machine learning, in particular, stands out as a crucial pillar for the future of society. Recognizing its significance, we have dedicated ourselves to mastering this transformative technology, ensuring we stay at the forefront of innovation.

Taking inspiration from the 1983 classic, *WarGames*, we decided to see if we could replicate the Tic-Tac-Toe learning machine using our knowledge and research. This project required hours of research and lots of trial and error, as we had not worked on or learned machine learning before.

While Tic-Tac-Toe represents a straightforward application due to its solved nature, our primary objective was to implement a deep-learning model as a learning exercise and a proof of concept for future, more intricate challenges.

We are proud to present our findings in this report, which marks an important milestone in our learning journey. Moving forward, we are eager to tackle more complex machine learning projects and contribute to the advancement of this transformative field.

Setup:

To construct the game, we first required a Tic-Tac-Toe board to handle the game states. This was easily implemented in the Board class. Additionally, we created an IO class to handle interfacing with the board. This class was used when testing the model, in order to get user inputs. When playing against the model, a curses wrapper was used to enable simple and intuitive gameplay.

Before the model could be trained, its architecture needed to be defined. We originally chose to use 3 hidden layers with dimensionality 64, however we later adjusted this to a single layer of 15 nodes. The model has 1x9 input, representing the state of each board position, and outputs the list of Q-values for each square on the board. The activation of the output layer is tanh in order to fit the design of our board and allow for intuitive setting of targets.

Training:

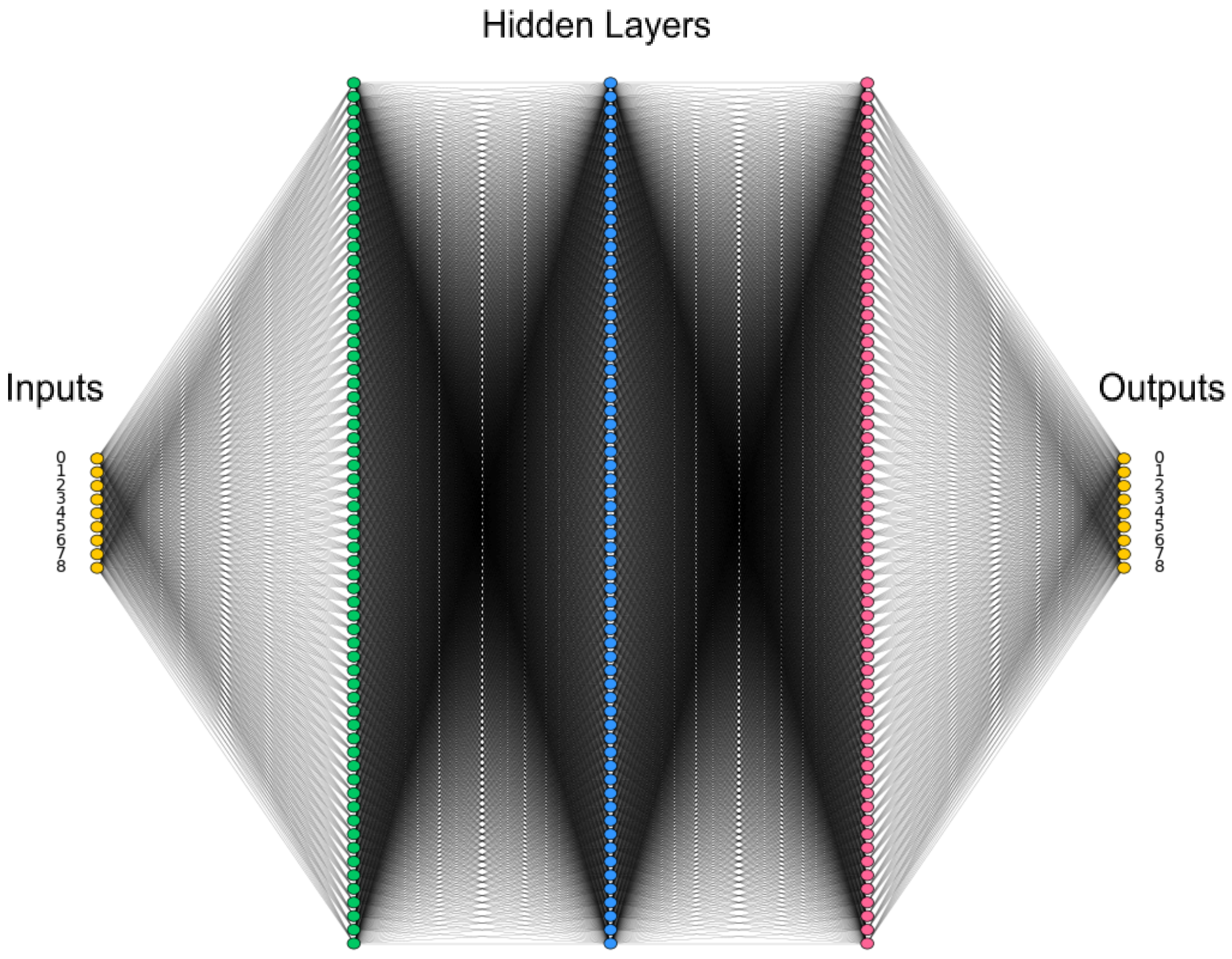
To train the model, we iterate through a number of user-defined epochs, each representing a single game. Within each game, an agent plays against a "smart-random" opponent. This "smart-random" opponent checks if it can win on its current turn by examining every possible move. If there is a possible win, it plays that move; otherwise, it picks a random valid move. The agent alternates between moving first and second, as having the first move provides a significant advantage in Tic-Tac-Toe. On the agent's turn, it chooses its action through either exploration, picking a smart-random move, or exploitation, analyzing the board and using the Q-values to determine the best move. The frequency of exploration versus exploitation is determined by the epsilon hyperparameter, which decays each epoch by a defined amount until it reaches the defined minimum epsilon.

In exploration, every action on an empty board space is examined. If an action will result in a win, that move is played, and the target Q-value of that action is updated to reflect this result. If an action results in a tie, that move is played, as it will, by definition, be the only possible move. Similar to wins, the target Q-value is updated for this action to the tie reward (hyperparameter). If there is no possible win or tie, a random valid action is selected and played.

In exploitation, we examine every valid action on the current board, checking for immediate wins. If none are found, we then check "double futures" (two moves ahead) for any potential losses. This trains the model to prevent losses and seek wins, as moves that can lead to a loss will be punished, and moves leading to a win will be rewarded. After checking for any future wins or losses, the best action is chosen and played.

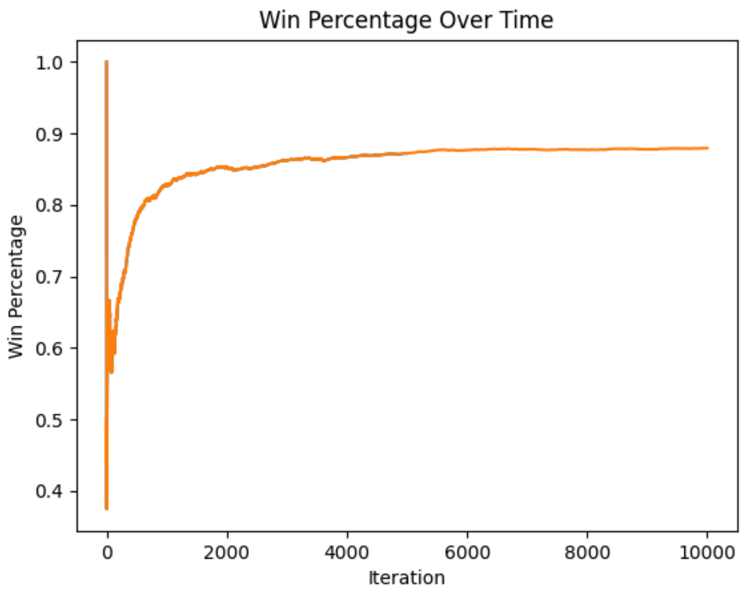
After the game concludes, we backpropagate through the board states to enable the development of multi-move strategies. Every time the agent plays a move, the board state and action taken are saved to a list of past states. This list is iterated through backwards, updating the target Q-value of the action taken to the result of the game multiplied by gamma (hyperparameter). At each step backwards, the result is multiplied by gamma, such that the result of the game has a lesser impact on older moves. This discounting is done to prevent overfitting.

Old Visualization:



Above is a visualization of our previous neural network. While it had dropout layers, the drawing highlights the essential components: inputs, three hidden layers, and outputs. From left to right, you can see the inputs, hidden layers 1 through 3, and the outputs. This neural network served as the core of our Tic-Tac-Toe AI, a sophisticated system requiring substantial computational resources. However, this network proved to be overly complex for the simple game of Tic-Tac-Toe, leading to overfitting and suboptimal performance.

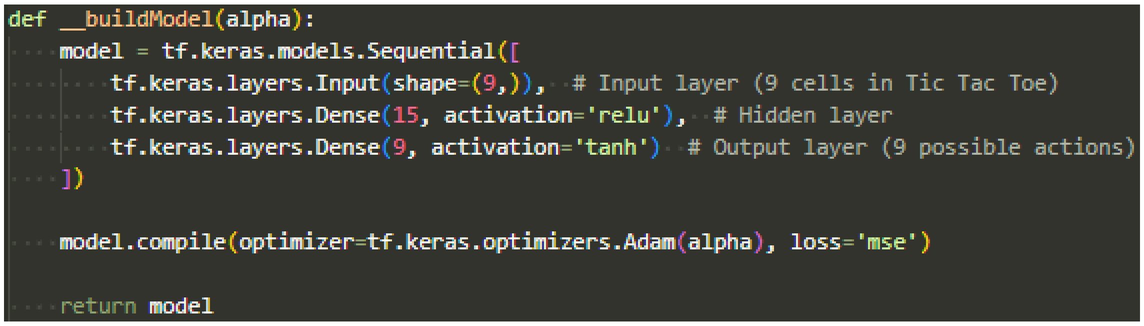
Old Results:



The graph above illustrates the win percentage over time for our older model. Although it performed well during training, it was highly unstable due to overfitting. When playing against a human user, it often made random, nonsensical moves, reflecting this issue. We identified the cause of the overfitting: our neural network was significantly deeper than necessary. We addressed this problem in our next version, discussed below.

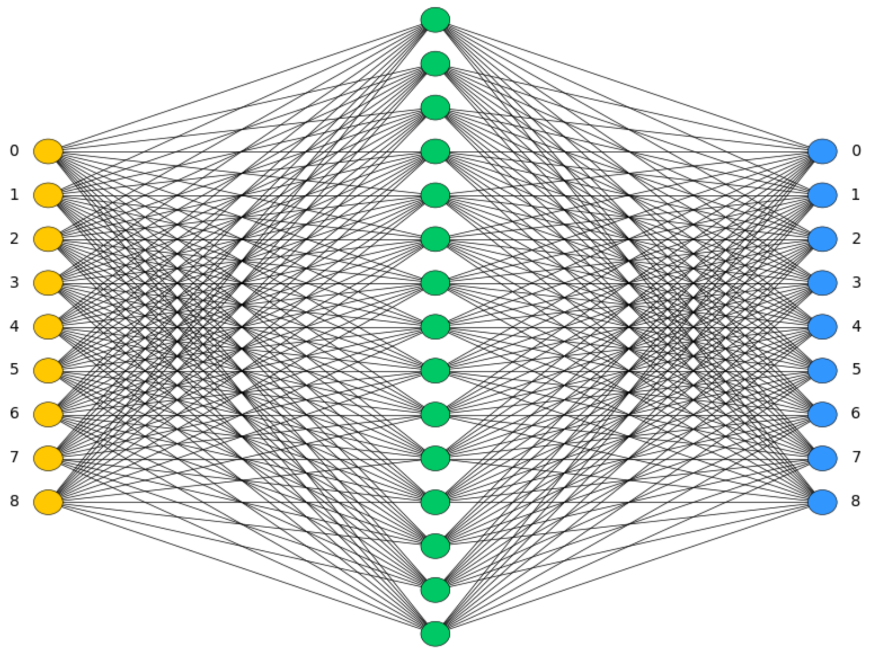
Current Neural Network - Post Improvement

Architecture:



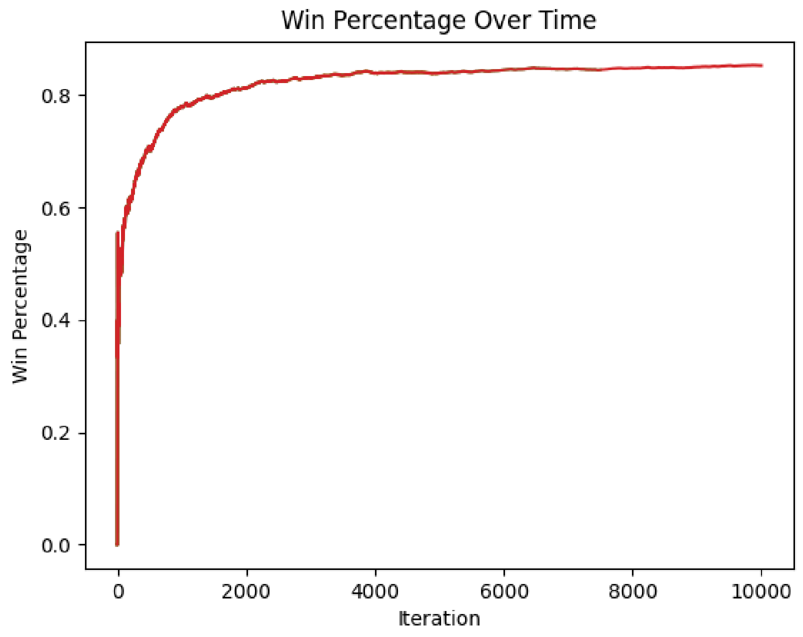
```
def __buildModel(alpha):
    model = tf.keras.models.Sequential([
        tf.keras.layers.Input(shape=(9,)),  # Input layer (9 cells in Tic Tac Toe)
        tf.keras.layers.Dense(15, activation='relu'),  # Hidden layer
        tf.keras.layers.Dense(9, activation='tanh')  # Output layer (9 possible actions)
    ])

    model.compile(optimizer=tf.keras.optimizers.Adam(alpha), loss='mse')

    return model
```

Visualization:



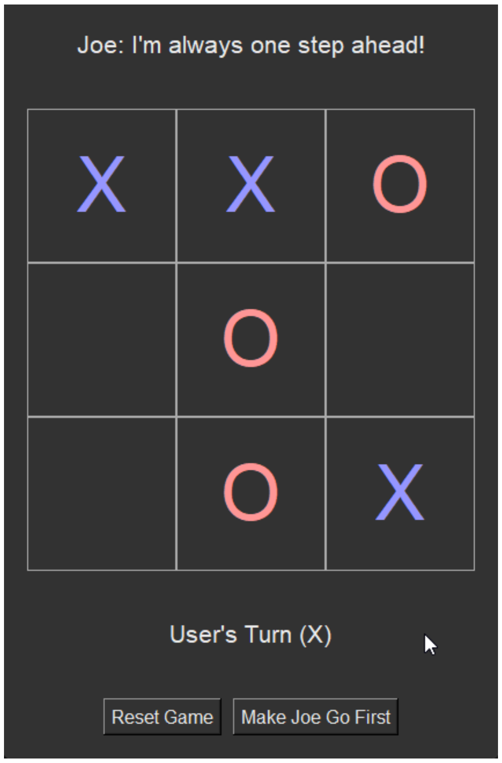
Above is our current and best neural network. Through a tedious amount of optimization, we have found that a 9 neuron input, one 15 neuron hidden layer, and a 9 neuron output layer, allows for the best fit.

Results - Post Improvement:



This simple win percentage over time graph showcases how efficient the new neural network is at learning the game. When compared to the previous graph, it is clear that the right decision was made in getting rid of the old network.

GUI:



To add more depth to this project we have made the user interface of this program a GUI. The GUI not only helps to make our project look more neat, as you don’t have to play it out of the terminal, but it also allows for a simple and easy play experience. Initially, we did the process of playing through the terminal and had the user pick the “row” and “column” they wish to play on. Now all the user has to do is simply click and the move will register.