Deep Neural Network Plays Connect-4

Authors:
Mohammad Islam mai54@cornell.edu ‘25
Aaron Babu ab2458@cornell.edu ‘25
Cornell University College of Computing and Information Sciences
Ithaca, New York, United States of America
Date of Submission : December 12, 2023

Abstract

Our project is all about getting deep into Connect4, a game that's simple to play but can get really tricky. We're going to build and train a deep neural network (DNN) to become really good at it. To do this, we're pitting our DNN against a tough opponent – a Minimax player with Alpha-Beta Pruning. We're curious to see if our neural network can learn from this experience and maybe even outsmart a logic-based approach to playing games.

The plan is to have our DNN play a ton of games against the Minimax player. Each game is a chance for our DNN to learn something new. We'll keep track of all the moves and decisions to build a big dataset for training. The goal is to make our DNN smarter at figuring out what move to make next. We want it to get so good that it can play Connect4 as well as, or better than, the Minimax player. This could really show off what deep learning can do in games like this.

We're measuring our success in two ways: first, by how well our DNN can play Connect4, and second, by what we learn about combining traditional gaming algorithms and deep learning. We're hoping to find out where each method shines, where they fall short, and how they might work best together. Everything we learn from setting up the project, getting it to work, and analyzing the results will teach us a lot about using machine learning in games. This isn't just about playing Connect4 better; it's about understanding more about deep learning in game AI and maybe even beyond that.

Literature Review

To kick off our project, we need to check out what's already out there and think about the algorithms and models we want to develop. Let's talk about some papers we found and what they're about.

The first paper, "Heuristic Search Planning with Deep Neural Networks using Imitation, Attention and Curriculum Learning," is pretty cool. It mixes heuristic search planning with deep neural networks. They use imitation learning, attention mechanisms, and curriculum learning to make planning strategies better. The idea is to train neural networks by watching experts, focus on what's important using attention mechanisms, and get better step by step with curriculum learning. This method could be a game-changer, maybe even better than old-school techniques. We're using it to get a better grip on heuristic search in deep neural nets, as we're also planning to use a minimax algorithm that involves this kind of search.

The second paper, "Deep Learning in Neural Networks: An Overview," is a great intro to deep neural networks. It goes over the basics and different types, like feedforward and convolutional neural networks, and how they're trained with stuff like backpropagation and gradient descent. It also looks at how they're used in different areas, like image recognition, language processing, and learning by trial and error. This paper is like our starting point to understand deep neural networks and how we might use them in our project.

Next, there's "Optimization Areas of the Minimax Algorithm." This paper dives into making the Minimax algorithm better, especially for games and decision-making in tricky situations. It talks about different ways to make Minimax more efficient and accurate, like using heuristic evaluations, alpha-beta pruning, and other methods to cut down the search space. We're looking into this to see how we can make better decisions in Connect4.

"Automated Architecture Design for Deep Neural Networks" is another interesting read. It explores how to automatically design the structure of deep neural networks. The paper looks at different methods like using reinforcement learning, evolutionary algorithms, or gradient-based optimization to find the best network setup for specific tasks. It also talks about the balance between how complex the design is and how well it performs. We're using these ideas to help build our model.

Lastly, the thesis "Connect-4 using Alpha-Beta pruning with minimax algorithm" focuses on making the minimax algorithm better with alpha-beta pruning. It shows how alpha-beta pruning lets us search deeper without using too much computer power, especially when we're looking further ahead. This paper is guiding us on using alpha-beta pruning to make our

minimax algorithm for Connect4 as good as it can be. We're also using these ideas to train our deep neural network.

Implementing the Connect Four Class

The Connect Four game is encapsulated within the Connect Four class of our source code in main.py (Figure 1). It handles game mechanics such as creating the board, placing pieces, checking for winning conditions, and managing the gameplay loop. The crucial aspect is the inclusion of the minimax algorithm to enable the AI player to make strategic moves by evaluating potential future game states. This algorithm iterates through possible moves, assigning scores to each based on the anticipated outcomes. Additionally, the class includes functionalities for player input, game state tracking, and displaying the game board. This allows for a solid foundation on how we represent a Connect Four game state in use for minimax and deep neural networks (Figure 2). Now that we have a representable and playable game state for Connect Four, we can move onto our next step of implementing a minimax algorithm to get the most optimal moves within games.

```
class ConnectFour:
    # creates a new connect four board
    def create_board(self):

    # drops a connect four piece into the board
    def drop_piece(self, board, row, col, piece):

    # determines if a location is valid connect four spot to drop a piece
    def is_valid_location(self, board, col):

    # gets the next open row in a column
    def get_next_open_row(self, board, col):

    # prints the board state to the console
    def print_board(self, board):

    # check if any piece has won either player the game
    def winning_move(self, board, piece):

    # determines if the game is over
    def is_game_over(self):
```

Figure 1: The Connect Four Class and encapsulated functions needed for game functionality

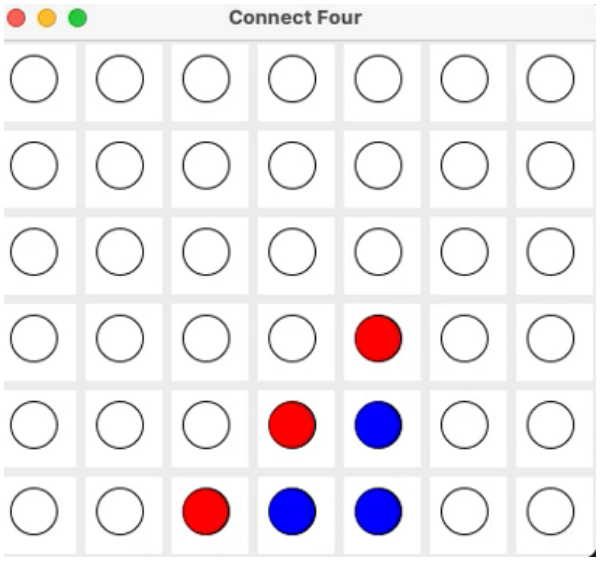


Figure 2: The current game representation of connect four in our code

Implementing the Minimax Algorithm

The Minimax strategy is a decision-making method used in two-player games with zero-sum outcomes, where the aim is to maximize wins while minimizing losses. It's based on the idea that each player wants to increase their own advantages, and thus, the strategy focuses on reducing potential losses (Figure 3). This method works in a recursive manner, considering future moves by simulating the actions of both players, which leads to forming an optimal decision-making tree.

In our project involving Connect Four, the Minimax strategy plays a central role in how the AI makes decisions. It looks at possible moves by going through the game tree step by step, foreseeing future scenarios for both the player and their opponent. The strategy weighs different possibilities, assigning values to each move and picking the one that seems most beneficial. A significant improvement in narrowing down choices was achieved by cutting off less promising paths using alpha-beta pruning, as explained in Source 5.

The development of our Connect Four game with AI features heavily relied on insights from several references. Source 5 was particularly useful in enhancing the Minimax strategy. It specifically helped in incorporating alpha-beta pruning, greatly boosting the strategy's effectiveness in selecting the best moves in the game. Additionally, Source 6, an insightful article on developing AI for Connect Four, provided a detailed guide. It assisted in shaping the Minimax strategy to fit the game's context, outlining a systematic process for tasks like evaluating positions, identifying endgame scenarios, and methodically searching for the most

advantageous moves (Figure 4). The combination of these references played a significant role in shaping the Minimax strategy's design and execution, leading to an effective AI gameplay experience in our Connect Four game.

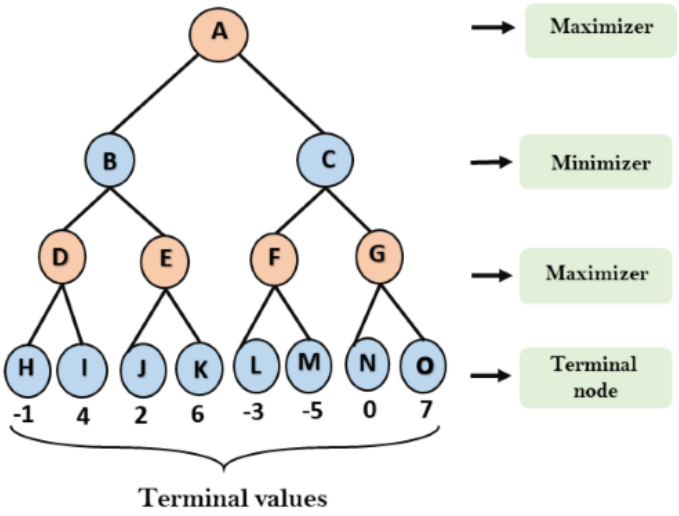


Figure 3: Visualization of the minimax algorithm

```
        # Evaluates the score of a window for a given player's piece.
        def evaluate_window(self, window, piece):

        # Scores the entire board position based on potential winning scenarios for
a player's piece.
        def score_position(self, board, piece):

        # Determines if the current board position is a terminal node.
        def is_terminal_node(self, board):

        # Minimax algorithm with alpha-beta pruning
        def minimax(self, board, depth, alpha, beta, maximizingPlayer):

        # Returns a list of valid locations to drop a piece
        def get_valid_locations(self, board):
```

Figure 4: Functions defined in our Connect Four class to develop and implement the minimax algorithm as guided by the literature review.

Curating a Dataset For The DNN

To create a training dataset for our Deep Neural Network (DNN), we crafted a script designed to simulate Connect Four games. This script sets up matches between a randomly acting AI and the Minimax algorithm. Throughout each game, it recorded distinct board states of Connect Four and the best moves as determined by the Minimax algorithm.

Our script, named Connect4DatasetGenerator (illustrated in Figure 5), built upon the functionalities of our existing Connect Four game class. It was tailored to generate a dataset by playing numerous games and collecting data on game states along with the Minimax algorithm's corresponding moves.

Here's what the script specifically recorded:

- **Game State Records:** These were snapshots of unique board configurations encountered during the games.
- **Best Move Records:** These noted down the optimal moves as figured out by the Minimax algorithm for each game state.

Utilizing this approach, we amassed a significant dataset comprising various Connect Four board states and the Minimax algorithm's predicted best moves. These records were then compiled into a CSV file, named connect4_data.csv, cataloging the game states and their associated best moves. This dataset laid the groundwork for training our DNN model, allowing it to learn from the Minimax algorithm's decision-making process in Connect Four games.

To broaden the dataset, we adapted the script to incorporate human players competing against the Minimax-based AI. We altered the play_game function to record moves made by human players instead of random AI choices. As before, each game state and the corresponding Minimax move were logged. In this setup, human players challenged the AI, creating a diverse set of game states and optimal move pairs.

After several rounds of both Minimax vs Random AI and Human vs Minimax matches, we gathered around 22,000 unique game states and their best moves. To ensure the distinctiveness of the dataset, we implemented a function to eliminate any duplicates in the csv file. This step was crucial to maintain the uniqueness of the game states for effective DNN learning. Moving forward, there's the potential to further augment this dataset with even more unique game states and corresponding best moves, enhancing the neural network's ability to predict patterns more accurately.

```
def play_game(self, seen_states):
      game_over = False
      game_states = []
      best_moves = []
      while not game_over:
          game_state_str = str(np.flip(self.board, 0))

          if self.turn == self.AI:
              best_move, _ = self.minimax(self.board, 5, -sys.maxsize, sys.maxsize,
True)
              row = self.get_next_open_row(self.board, best_move)
              self.drop_piece(self.board, row, best_move, self.AI_PIECE)

              if game_state_str not in seen_states:
                  seen_states.add(game_state_str)
                  game_states.append(np.flip(self.board, 0).flatten())
                  best_moves.append(best_move)
          else:
              random_move = self.random_move()
              row = self.get_next_open_row(self.board, random_move)
              self.drop_piece(self.board, row, random_move, self.USER_PIECE)

          if self.winning_move(self.board, self.turn + 1):
              game_over = True
          elif len(self.get_valid_locations(self.board)) == 0:
              game_over = True
          self.turn = self.USER if self.turn == self.AI else self.AI

      return game_states, best_moves
```

Figure 5: Function to generate random game states and best moves as per Minimax AI

Designing and Implementing the DNN Architecture

To create a highly effective and pattern-savvy Deep Neural Network (DNN) for our Connect Four game, we decided to use a Convolutional Neural Network (CNN) setup. We preprocessed our dataset, which included game states and their best moves, to get it ready for building our DNN model.

For Data Preprocessing, we started by pulling data from a CSV file. We then transformed the game state information, originally in string format, into lists of integers. This was

done to accurately represent the configurations of the Connect Four board in a format the model could understand.

When it came to Model Construction, we set up the CNN with two Conv2D layers, each having different filter sizes. This design was chosen to pick up a range of features from the Connect Four board. After these layers, we added a Flatten layer, which turns the output into a one-dimensional array. This was necessary to link up with the Dense layers that followed. We used two Dense layers in our model, enhancing its ability to learn and recognize complex patterns from the game states.

We chose a CNN architecture because Connect Four's board shares some similarities with image data (as noted in Source 2). Using CNNs helps the model identify spatial relationships and patterns on the Connect Four board, crucial for pinpointing the best moves. Our goal was to boost the DNN's skill in spotting strategic patterns and making smart moves, ultimately improving its ability to play Connect Four.

We trained our model on the dataset, splitting it into 80% for training and 20% for validation. To fine-tune the model, we used the RMSprop optimizer and a sparse categorical cross-entropy loss function. The training process went on for 100 epochs with a batch size of 32. After some experimentation, we found that a learning rate of 0.001 worked best, along with the sparse_categorical_crossentropy loss function, especially when we looked at validation accuracy (as shown in Figure 6). We included ReLU in our hidden layers to better detect the nonlinearity of patterns and improve feature extraction (according to Source 4). For the output layer, we used softmax, which helps in determining the likelihood of each move in all seven columns. This approach provides a clear probability-based understanding of the model's predictions, making it easier to choose the best move based on the highest probability.

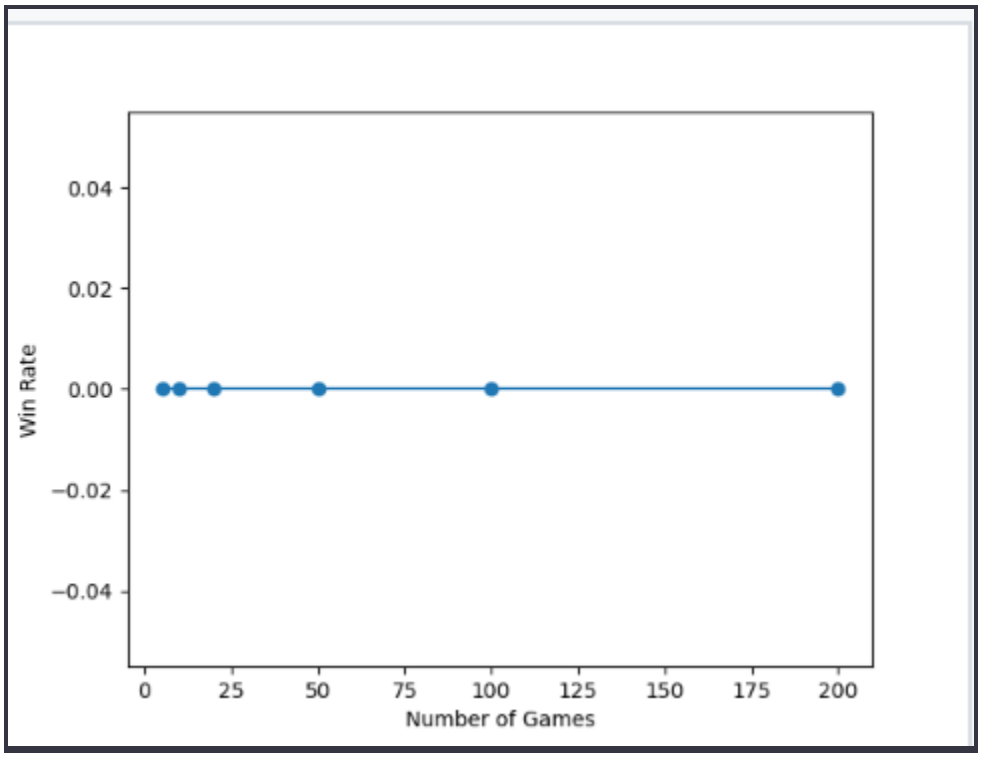
After training, we evaluated the model's accuracy and loss to see how well it could predict optimal moves based on game states. Our highest achievement was a validation accuracy of 81.03%, reached after several training rounds. The model essentially predicts the best move probabilities for each of the seven columns in a one-dimensional array, allowing the AI to choose the most likely move.

```
# Define the model
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(6, 7, 1)))
```

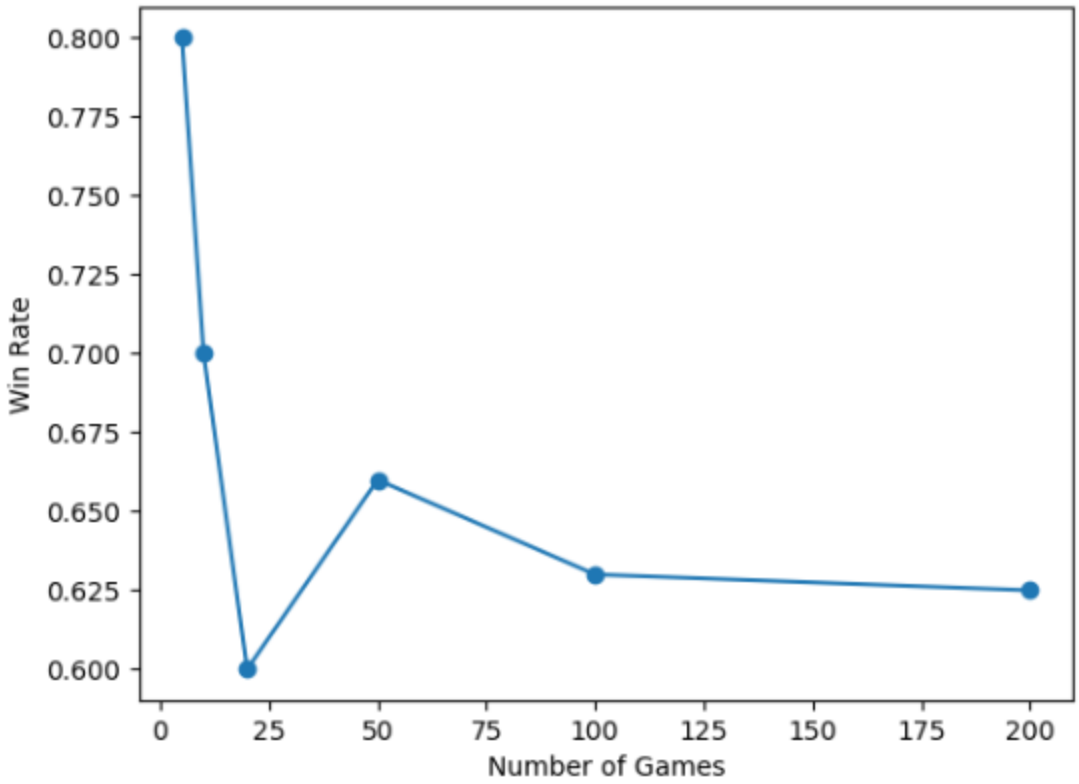
```
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(7, activation='softmax'))

# Compile the model
model.compile(loss='sparse_categorical_crossentropy', optimizer=RMSprop(lr=0.001),
metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=100, batch_size=32)
```

Figure 6: DNN Architecture for the Connect4 Model

Evaluation of the CNN Model



This figure is the gameplay of the CNN model against minimax AI. Never winning or being able to replicate the moves of minimax AI.



This figure is the win rate percentage of the CNN model against random AI, converging to about a 63.5% win rate.

The evaluation of our DNN model revealed promising results: it consistently outperformed a random AI, securing victories around 63% of the time in Connect Four. However, when challenged by the sophisticated Minimax algorithm, our model fell short. This outcome highlights the complexity of the game and emphasizes the need for more sophisticated strategies to compete effectively. Limitations were apparent, notably the dataset's size, which potentially restricted the model's ability to generalize across diverse gameplay scenarios. Moreover, the game's complexity might demand a larger and more varied dataset for improved decision-making in diverse situations as there are 3^42 different board states in Connect Four. These limitations underscore the importance of continual progression in a larger dataset and algorithmic enhancements to the DNN's performance against competitive opponents like the Minimax algorithm in Connect Four.

Other Models: DQN Model

In our Connect 4 AI project, we didn't just stop at the Convolutional Neural Network (CNN) model; we also dived into using a Deep Q-Network (DQN) model. This model takes advantage of Reinforcement Learning (RL) principles, allowing the AI to pick up optimal game strategies by interacting with the game itself.

The design of our DQN model was tailored to understand the Connect 4 game's state and come up with strategic moves. The input layer was set up to match a flattened 6x7 board, giving us 42 inputs and ensuring the model could interpret the entire board at each decision-making moment. We included several hidden layers in the model, with sizes of 512, 256, 128, 64, and 64, all using Tanh as the activation function. This setup helped the model to learn and identify complex patterns in Connect 4. The output layer has 7 units, one for each column of the Connect 4 board, providing Q-values that estimate the benefit of placing a piece in each column.

```
class DQN(nn.Module):
    def __init__(self, input_size, output_size, hidden_layer_sizes):
        super(DQN, self).__init__()
        self.input_size = input_size
        self.output_size = output_size
        self.layers = nn.ModuleList()
        M1 = input_size
        for M2 in hidden_layer_sizes:
            layer = HiddenLayer(M1, M2)
            self.layers.append(layer)
            M1 = M2

        # Final layer without activation
        self.layers.append(nn.Linear(M1, output_size))

    def forward(self, X):
        for layer in self.layers:
            X = layer(X)
        return X

    def save(self, filename):
        torch.save(self.state_dict(), filename)

    def load(self, filename, device):
        self.load_state_dict(torch.load(filename, map_location=device))
```

We trained the DQN model using an Experience Replay technique, where a Replay Buffer stored different game experiences (like state, action, reward, next state, done) and picked them out randomly. This was crucial for making the training process more stable and less dependent on consecutive experiences. The heart of the DQN's training was the Q-learning

update rule. This rule calculates the target Q-value for each action based on immediate rewards and the projected future rewards, adjusted by a discount factor.

```
class ReplayBuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = []
        self.position = 0

    def push(self, state, action, reward, next_state, done):
        """Saves a transition."""
        if len(self.buffer) < self.capacity:
            self.buffer.append(None)
        self.buffer[self.position] = (state, action, reward, next_state, done)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        """Randomly sample a batch of experiences from memory."""
        return random.sample(self.buffer, batch_size)

    def __len__(self):
        return len(self.buffer)
```

We used an ε-greedy approach to strike a balance between exploring new actions and exploiting known good ones. Initially, the model leaned more towards trying out new things, but gradually, as it learned more about the game, it started focusing on exploiting the best strategies it knew.

The DQN model worked in tandem with our custom ConnectFourEnv, which took care of the game mechanics and gave the AI feedback like the game state, rewards, and termination signals. We also had a CompeteEnv subclass where the DQN model could challenge the Minimax algorithm, creating a more dynamic and challenging training environment.

```
    class CompeteEnv(ConnectFourEnv):
        def __init__(self, dqn_agent):
            super().__init__()
            self.dqn_agent = dqn_agent

        def choose_action(self):
            if self.game.turn == self.game.AI:
                # DQN Agent's turn
                with torch.no_grad():
                    state = torch.tensor(self.game.board.flatten(),
dtype=torch.float32)
                    q_values = self.dqn_agent(state)
```

```
                    action = q_values.argmax().item()
            else:
                # Minimax's turn
                action, _ = self.game.minimax(self.game.board, 5,
-float("inf"), float("inf"), True)

            return action
```

Through this DQN model implementation, we've pushed our boundaries in exploring sophisticated AI techniques and how they apply to strategic games. Along with our CNN model, this approach has helped us gain deeper insights into the possibilities and hurdles of using machine learning in deterministic games. However, it's worth noting that the results from the DQN model didn't quite meet our expectations.

Evaluation of DQN Model

After we finished setting up our Deep Q-Network (DQN) model for Connect 4, we moved on to a thorough evaluation phase. Our goal was to see how well the DQN model could learn and play against both human players and the Minimax algorithm. But, honestly, the results showed that the DQN model didn't perform as well as we'd hoped.

During the tests, we noticed that the DQN model often made moves that seemed random or kept dropping pieces into the same column over and over. This showed us that the model wasn't really picking up on the strategies or goals of Connect 4. One big issue we identified was with the reward system in Connect 4. The game is mostly about winning or losing, without much in the way of ongoing scoring to help guide the model's learning.

```
            # Check for win, loss, or draw
            if self.game.winning_move(self.game.board,
self.current_player + 1):
                # If the current player wins
                print("AI wins")
                reward = 1 if self.current_player == self.game.USER else
-1
                done = True
            elif self.game.winning_move(self.game.board, 1 -
self.current_player + 1):
```

```
                # If the opponent wins
                print("Minimax wins")
                reward = -1 if self.current_player == self.game.USER
else 1
                done = True
            elif len(self.game.get_valid_locations(self.game.board)) ==
0:
                # Draw
                reward = 0.5
                done = True
```

Because Connect 4 is mostly a win-lose kind of game, the DQN model didn't get much useful feedback on how good its moves were unless they directly led to a win or loss. This all-or-nothing style of feedback wasn't enough for the model to really grasp the more subtle strategies of Connect 4 gameplay. As a result, the model didn't get much better than just picking basic moves, which meant it couldn't really compete with human players or the Minimax algorithm.

In short, our evaluation of the DQN model showed that using reinforcement learning for games like Connect 4 can be tricky, especially when the game doesn't offer much feedback on strategy. This experience has taught us a lot about the need to think carefully about the game itself and whether the AI technique we're using is a good fit. We'll definitely keep these lessons in mind as we try out AI techniques in other strategic games.

SOURCES:

(1) **Heuristic Search Planning with Deep Neural Networks using Imitation, Attention and Curriculum Learning**
https://arxiv.org/pdf/2112.01918.pdf

(2) **Deep Learning in Neural Networks: An Overview**
https://arxiv.org/pdf/1404.7828.pdf

(3) **Optimization Areas of the Minimax Algorithm**
https://www.diva-portal.org/smash/get/diva2:1778372/FULLTEXT01.pdf

(4) **Automated Architecture Design for Deep Neural Networks**
https://arxiv.org/pdf/1908.10714.pdf

(5) **Connect-4 using Alpha-Beta pruning with minimax algorithm**
https://dif7uuh3zqcps.cloudfront.net/wp-content/uploads/sites/11/2022/01/15124632/Volume6_Issue1_Paper10_2022.pdf

(6) **Evaluation of the Use of Minimax Search in Connect-4 —How Does the Minimax Search Algorithm Perform in Connect-4 with Increasing Grid Sizes?**
https://www.scirp.org/journal/paperinformation.aspx?paperid=125554

(7) **Artificial Intelligence at Play — Connect Four (Mini-max algorithm explained**
https://medium.com/analytics-vidhya/artificial-intelligence-at-play-connect-four-minimax-algorithm-explained-3b5fc32e4a4f