The University of Texas at Austin
Texas McCombs
MS Business Analytics
McCombs School of Business

Optimization Project 3

Reinforcement Learning

Group Members:
Téa McCormack (tsm2423)
Hrishi Salitri (hss793)
Aryan Shah (ahs2388)
Aman Sharma (as235548)
Anjali Pilai (ap66745)

Professor:
Dan Mitchell

Table of Contents

1. Problem Overview

Connect 4 is a two player game played on a 7x6 grid with the objective of being the first player to align four discs in a row, either horizontally, vertically or diagonally. It is a deterministic, turn based game, making it the perfect candidate for strategic machine learning approaches. Inspired by DeepMind's success with AlphaGo, this project explores how reinforcement learning (RL) can be applied to build a competitive Connect 4 agent. The goal of this project is to use a neural network that is trained to mimic Monte Carlo Tree Search, and then iteratively improve a neural network through self play, using a combination of policy gradient methods and double deep Q learning techniques in order to strengthen gameplay strategy and decision making over time.

2. Introduction to Reinforcement Learning for Connect 4

Reinforcement learning is a subfield of machine learning where an agent learns optimal behavior through interacting with an environment. It provides a powerful framework for training models to make decisions in sequential and interactive environments. In the context of Connect 4, a reinforcement learning agent learns to play the game of Connect 4 by repeatedly interacting with the game environment, observing different game outcomes, and adjusting its strategy in order to maximize its chances of winning.

Since Connect 4 has a clear reward structure, defined state transitions, and is turn based, the game is a prime example of Markov Decision Process. In this project, self play is the primary learning mechanism for improving our model. The agent learns by playing games against a variety of opponents, including previously trained models. This technique encourages continuous learning and adaptation, in order to ensure that the agent becomes more robust and strategic over time. Our hope is that utilizing this technique will allow our model to outperform other models during the tournament.

When using reinforcement learning, at each step, the agent observes a state, takes an action, receives a reward, transitions to a new state, and repeats the process. In this scenario, the state is the current Connect 4 board, the action is the decision of which column to drop the disc in, and the reward is if the model won or lost. The goal is to learn a policy by mapping states to actions in order to maximize the cumulative reward over time.

Key components of reinforcement learning include:

- **Agent**: The learner or decision maker (e.g., a Connect-4 player).
- **Environment**: The system the agent interacts with (e.g., the game of Connect-4).
- **State**: A representation of the current situation (e.g., the current board).
- **Action**: A possible move the agent can make (e.g., dropping a disc into a column).
- **Reward**: Feedback from the environment (e.g., +1 for a win, -1 for a loss).
- **Policy**: The strategy used by the agent to decide actions.

- **Value Function**: A prediction of future rewards for being in a given state or state-action pair.

Two common approaches in reinforcement learning are policy based methods and value based methods. Policy based methods, such as policy gradients, directly optimize the policy. On the other hand, value based methods, such as Q learning, estimate the expected return from states or state action pairs. These methods allow the agent to improve through self play, by competing against a diverse set of opponents, including improved versions of M1.

Games like Connect 4 serve as good uses for reinforcement learning due to the strategy needed to win, and the clear rules. Reinforcement learning is suitable for game development for a variety of reasons. Connect 4 has well defined rules and outcomes, which is ideal for reinforcement learning training due to its structured environment with unambiguous feedback signals of a win, loss, or draw. In order to develop a strong model, there must be a balance between exploration and exploitation. New strategies must be tried, and known successful moves must be leveraged in order to make the most successful model. Another technique used for game strategy development is the self play mechanism. Models compete against each other, while adding improved models to the available models. This mechanism mimics human learning through practice.

In this project we take advantage of these tactics in order to train agents that evolve through self play, adversarial training, and iterative refinement using policy gradients and double deep Q networks. This mirrors important aspects of the AlphaGo training pipeline, and sets the stage for developing powerful, generalizable game playing strategies.

3. Methodology

Selection of Initial Neural Network

To begin the reinforcement learning process, we selected a baseline neural network, M1, from the models that were developed in Project 1. Each of these models had been trained to approximate Monte Carlo Tree Search, allowing them to mimic strategic gameplay with some degree of foresight. Since these networks already capture a baseline level of intelligence when playing Connect 4, they serve as ideal starting points for further improvement with reinforcement learning.

To identify the strongest candidate to serve as M1, we conducted a “Round Robin” tournament in which each of the five models played 400 games against each other. We recorded information about the games played, including win rate, tie rate, and performance when playing first. The top performing model achieved an overall win rate of 87.75%, maintained a perfect 100% win rate when playing first, and had no ties. These statistics indicate that the model displayed decisive and dominant performance. This model was selected as M1, providing a strong foundation for iterative enhancement through self play and adversarial training.

The model from project 1 that was chosen to serve as M1 was initially trained on a limited dataset generated using 2000 step Monte Carlo Tree Search (MCTS). While the model originally achieved a reasonable accuracy, the limited dataset generation resulted in overfitting and poor gameplay performance. This model lacked the ability to recognize strategic threats, resulting in simplistic behavior during gameplay.

To address this, the training dataset was expanded to include data provided by Professor Mitchell. This dataset was generated using 7500 step MCTS, and included more diverse board states, allowing the model to identify more offensive and defensive strategies. With this higher quality training dataset, the model was introduced to more realistic gameplay situations, such as blocking opponent wins and setting up strategic plays, instead of repetitive strategies.

In addition to using improved data, the architecture of the CNN was also improved. The original CNN consisted of 15000 parameters, and was upgraded to a deeper, more complex network comprising over 4.8 million parameters. Additionally, the model was modified to include more convolutional filters per layer, and max pooling was replaced with stride based convolutions to preserve more spatial information. The fully connected layers were also expanded, allowing the model to better understand high level game patterns.

As a result of these improvements, the upgraded M1 model exhibited substantial improvements in gameplay. Instead of stacking pieces in a single column, the model began proactively blocking its human opponents moves, and executing multi step strategies. In testing, the model was able to beat novice players about 90% of the time, demonstrating a clear improvement in both prediction accuracy and strategic depth.

Policy Gradient Strategy Implementation

Policy gradient methods are a class of reinforcement learning algorithms that directly optimize the policy rather than estimating the value functions first. The idea behind this method is to represent the policy as a parameterized function, typically using a neural network. In the context of Connect 4, the policy is represented using a neural network that outputs a probability distribution over all possible moves given the current state of the game. The objective is to adjust the parameters of this network to increase the likelihood of performing actions that lead to a winning game.

This method involves running complete games, collecting the sequence of states, actions, and rewards, and then computing the gradient of the expected reward with respect to the policy parameters. At each step during training, the agent observes the current state of the game, samples an action from the policy's output distribution, and plays a full game while recording the states, actions, and rewards. After the game, or episode, is completed, the agent computes gradients that push the network to increase the likelihood of actions that led to higher rewards, and decrease the likelihood of those that led to poorer outcomes. One commonly used approach is the REINFORCE algorithm, which updates the policy by weighting the log probability of each action by the total reward achieved. Gradients are estimated as:

$$\bigtriangledown J(\theta) = E[\bigtriangledown_\theta log \pi_\theta(a_t|s_t) \bullet R_t]$$

In this equation, $\pi_\theta(a_t|s_t)$ represents the policy's probability of taking action a_t in state s_t, while R_t represents the total reward received after time t. This allows the agent to learn which sequences of moves tend to lead to successful outcomes, and to increase the probability of taking these actions in similar future states. A major benefit of using policy gradient methods is the ability to learn stochastic policies, which is particularly useful in strategic games like Connect 4, where it is a benefit to being unpredictable.

In this project, we applied the policy gradient method by training our agent through self play against a random opponent, including earlier versions of itself. By playing random opponents, this encourages the agent to continuously adapt and improve, allowing it to learn from its own mistakes and successes across a variety of competitive scenarios. Over time, this will lead to the emergence of more complex and powerful strategies that allow the model to be more successful than its competitors.

Policy Gradient Training Approach

In this project, a policy gradient-based reinforcement learning approach was employed to improve a pre-trained Connect 4 model (denoted as M1), originally selected through a round-robin tournament among models trained by team members.
The training process incorporated several strategies to ensure meaningful learning and robust evaluation.
Below we describe the training workflow and hyperparameter choices in detail.

1. Opponent Selection Strategy

During training, the opponent model (M2) was randomly selected from the pool of available pretrained models.
To ensure compatibility:

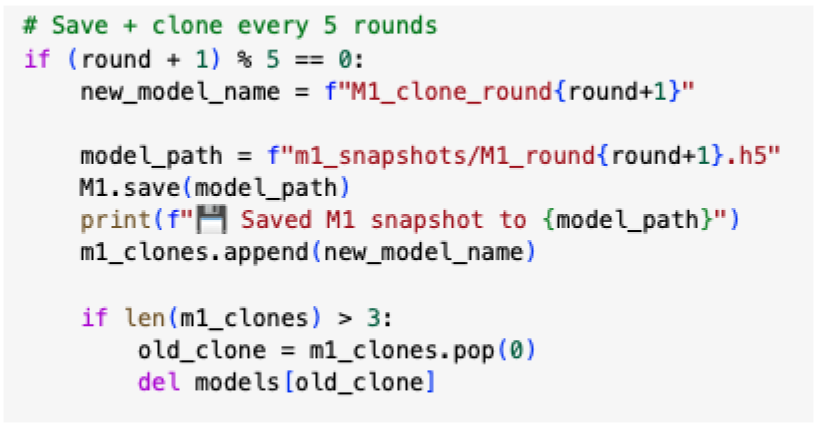
- M2 was required to have the same input format (two-channel input for player and opponent positions).
- M2 was randomly chosen from all models except the currently trained M1.

This random selection introduces variability and diversity in gameplay, helping the agent avoid overfitting to a single fixed opponent.

2. Saving Updated M1 Snapshots
To track progress and preserve strong intermediate versions of the trained model:

- Every 5 training rounds, a snapshot of M1 was saved to disk.
- The snapshot was also cloned into the opponent pool with a name like M1_clone_roundX, making it eligible to be selected as a future opponent.

Only the latest 3 clones were retained to avoid growing memory usage indefinitely.
This dynamic updating of the opponent pool allowed M1 to continually challenge stronger versions of itself, implementing a form of self-play curriculum learning.



```python
# Save + clone every 5 rounds
if (round + 1) % 5 == 0:
    new_model_name = f"M1_clone_round{round+1}"

    model_path = f"m1_snapshots/M1_round{round+1}.h5"
    M1.save(model_path)
    print(f"💾 Saved M1 snapshot to {model_path}")
    m1_clones.append(new_model_name)

    if len(m1_clones) > 3:
        old_clone = m1_clones.pop(0)
        del models[old_clone]
```

3. Valid Move Criteria
Throughout training and gameplay:

- Only legal moves (columns with at least one empty slot) were allowed.
- When sampling an action from the softmax output, moves corresponding to full columns were masked out.
- After masking, the remaining action probabilities were re-normalized to maintain a valid probability distribution.

This ensured that the agent's learning process respected the Connect 4 rules, preventing illegal moves from corrupting the training signal.

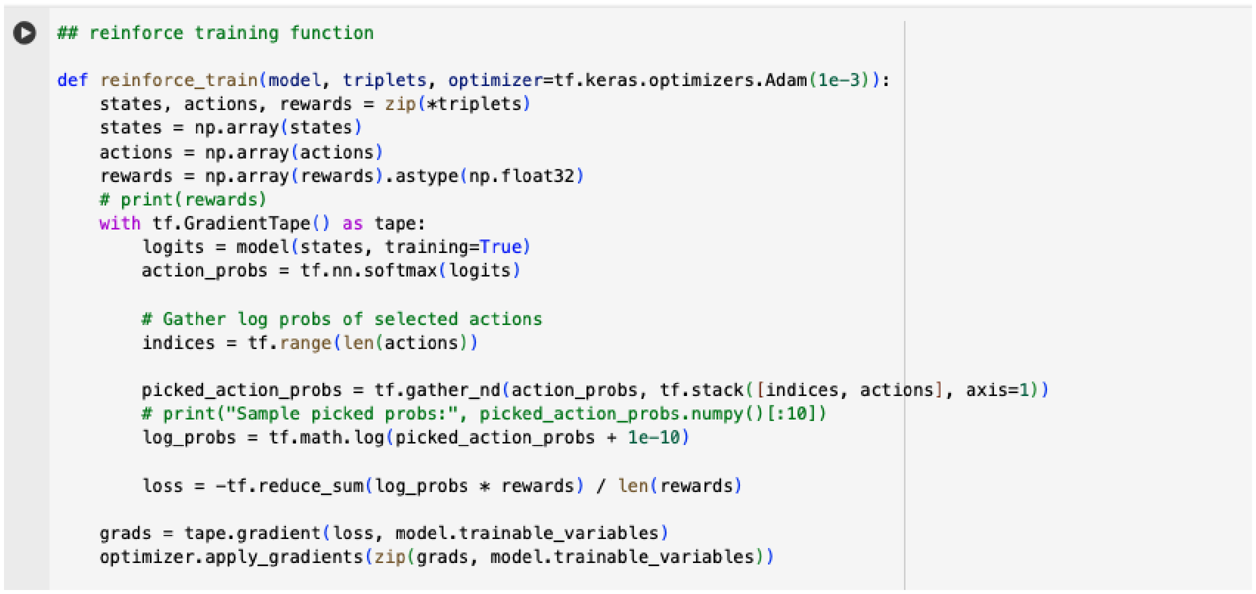
4. REINFORCE and Discounted Reward Computation
The policy gradient updates were based on the REINFORCE algorithm.
Key steps:

- After completing an episode (a full game), rewards were assigned:

- +1 for a win,
- -1 for a loss,
- 0 for a draw.

- The rewards were discounted backward in time using a discount factor \gamma = 0.99 to favor immediate over distant rewards.
- Unlike simple policy gradients, discounted cumulative rewards were used at each step, stabilizing learning and reducing high-variance updates.

Thus, the training process effectively reinforced sequences of moves that eventually led to wins.



```python
## reinforce training function

def reinforce_train(model, triplets, optimizer=tf.keras.optimizers.Adam(1e-3)):
    states, actions, rewards = zip(*triplets)
    states = np.array(states)
    actions = np.array(actions)
    rewards = np.array(rewards).astype(np.float32)
    # print(rewards)
    with tf.GradientTape() as tape:
        logits = model(states, training=True)
        action_probs = tf.nn.softmax(logits)

        # Gather log probs of selected actions
        indices = tf.range(len(actions))

        picked_action_probs = tf.gather_nd(action_probs, tf.stack([indices, actions], axis=1))
        # print("Sample picked probs:", picked_action_probs.numpy()[:10])
        log_probs = tf.math.log(picked_action_probs + 1e-10)

        loss = -tf.reduce_sum(log_probs * rewards) / len(rewards)

    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

5. Hyperparameter Choices

- Discount Factor (γ):

Set at 0.99, emphasizing near-term rewards while still considering long-term consequences.

- Batch Size:

128 triplets (state, action, discounted reward) were sampled per training update, providing stability and smoothing gradients.

- Optimizer:

Adam optimizer with a learning rate of 1 \times 10^{-3} was used, balancing fast convergence with stability.

- Episodes per Batch:

50 episodes (full games) were played per training batch.

- After 50 games, all generated (state, action, reward) triplets were collected and used to compute discounted returns, normalize rewards, and apply a single gradient update.

This setup ensures that each gradient update is based on sufficiently diverse experiences, rather than noisy single-game feedback.

6. MCTS as a Benchmark Opponent

To better benchmark the trained model's strength:

- A Monte Carlo Tree Search (MCTS) based opponent was also implemented.
- MCTS Level: 1000 simulations per move.

Purpose:

The MCTS agent, which explores move trees deeply, provides a strong, principled benchmark far beyond random or weak heuristic opponents.

Impact:

Playing against a consistent, powerful MCTS opponent helped reveal genuine improvements in strategic decision-making during evaluation.

7. Additional Experiment: Actor-Critic Policy Gradient

In addition to vanilla REINFORCE policy gradients, an Actor-Critic (AC) variant was implemented:

- Actor Network: Outputted action probabilities (policy \pi(a|s)).
- Critic Network: Estimated the value function V(s) — expected future return from a state.
- Training:

The Actor was trained using an advantage-weighted policy gradient:

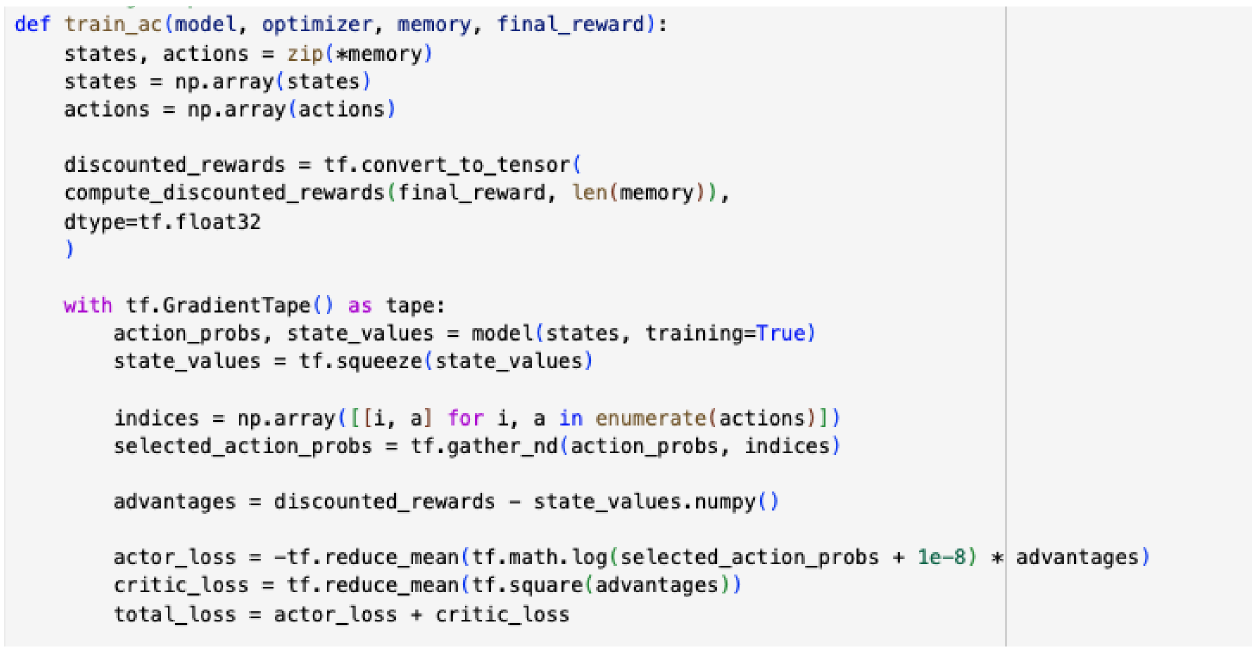
$$Advantage\ (s_t, a_t)\ =\ R_t\ -\ V(s_t)$$

reducing variance and speeding up learning.

Implementation Details:

Both actor and critic shared early layers (a shared feature extractor), then split into separate heads.

Actor-Critic methods are typically more sample-efficient compared to vanilla REINFORCE, and this experiment was conducted to explore further potential improvements.



```python
def train_ac(model, optimizer, memory, final_reward):
    states, actions = zip(*memory)
    states = np.array(states)
    actions = np.array(actions)

    discounted_rewards = tf.convert_to_tensor(
    compute_discounted_rewards(final_reward, len(memory)),
    dtype=tf.float32
    )

    with tf.GradientTape() as tape:
        action_probs, state_values = model(states, training=True)
        state_values = tf.squeeze(state_values)

        indices = np.array([[i, a] for i, a in enumerate(actions)])
        selected_action_probs = tf.gather_nd(action_probs, indices)

        advantages = discounted_rewards - state_values.numpy()

        actor_loss = -tf.reduce_mean(tf.math.log(selected_action_probs + 1e-8) * advantages)
        critic_loss = tf.reduce_mean(tf.square(advantages))
        total_loss = actor_loss + critic_loss
```

Random Move Selection

At the start of each Connect 4 game:

- Between 2 to 4 moves were played entirely at random, regardless of the model's action probabilities.
- This random initialization breaks symmetry, generates more diverse board states early on, and prevents the model from repeatedly encountering similar openings.
- Without this step, especially in early training when the policy is largely untrained, the agent would always encounter near-empty boards and struggle to discover winning strategies.

  Note: Random opening moves were not stored for training. Only the subsequent moves made by the policy network were used for learning.

During normal gameplay:

- Instead of always selecting the highest-probability action (pure exploitation), moves were sampled according to the predicted softmax probability distribution output by the model.
- Higher-probability moves were favored, but lower-probability moves were still occasionally sampled based on their assigned probabilities.

- This method naturally implements a balance between exploration (trying sub-optimal but potentially promising moves) and exploitation (favoring moves known to perform well).

Mathematically, for each board state s_t:

- The model outputs a probability distribution $\pi(a|s_t)$ over 7 possible moves.
- A move a_t is sampled according to π, rather than deterministically choosing $arg\ max_a\ \pi(a|s_t)$.

Q-Learning Approach

While policy gradient methods optimize the action-selection policy directly, Q-learning is a value-based reinforcement learning (RL) algorithm designed for agents to learn optimal policies by estimating the expected cumulative rewards associated with taking specific actions in given states. The core of the Q-learning algorithm is to learn a state-action value function, often called the Q-value, which measures the expected future reward of taking action *a,* in state *s*, following an optimal policy thereafter. This learning process is guided by the Bellman equation:

$$Q(st, at) \leftarrow Q(st, at) + \alpha[rt + 1 + \gamma amaxQ(st + 1, a) - Q(st, at)]$$

In the context of Connect-4, the state is represented by the current board configuration, and actions correspond to column placements. The Q-learning agent learns optimal policies by repeatedly interacting with the environment and adjusting its Q-value estimates based on observed outcomes.

Deep Q Learning (DQN)

Due to the complexity of the Connect-4 state space, we implemented Deep Q-Networks (DQN) to approximate the Q-value function using a neural network. The model takes a board state as input and outputs Q-values for the seven possible actions (columns). We adapted our policy gradient architecture by replacing the softmax output with a linear activation to produce real-valued Q-estimates appropriate for value prediction tasks. An epsilon-greedy policy was used to balance exploration and exploitation.

To ensure valid gameplay, illegal moves (full columns) were either masked or assigned negative values during both training and inference to prevent the agent from selecting them. Training was stabilized using several standard techniques:

- **Experience Replay:** Transitions (s,a,r,s′) were stored and sampled in mini-batches to reduce correlations.
- **Target Network:** A delayed copy of the main network provided stable Q-value targets.
- **Epsilon Annealing:** The exploration rate ϵ was gradually decreased to shift toward exploitation.

Game Initialization: Games began with random moves to expose the agent to a wider range of board states; these moves were excluded from training.

Despite these enhancements, DQN can suffer from Q-value overestimation. To address this, we implemented Double DQN, which decouples action selection and evaluation using separate networks. This significantly improved value estimation accuracy and overall training stability.

Double Deep Q Learning (DDQN)

As mentioned above, a key limitation of standard DQN is its tendency to overestimate Q-values, as it uses the same neural network for both action selection and evaluation. This overestimation is particularly problematic in games like Connect-4, where a single misjudged move can shift the outcome. To mitigate this, we implemented Double Deep Q-Learning (DDQN), which decouples the action selection and evaluation steps. The online network selects the best action for the next state, while the target network evaluates that action:

$$yt = rt + \gamma Qtarget(st + 1, argamaxQonline(st + 1, a))$$

By separating the decision and evaluation components, DDQN reduces the bias in value estimation and leads to more stable learning. This makes DDQN particularly well-suited for Connect-4, where accurate action-value estimation is critical for both defensive and offensive play. Empirically, we found DDQN led to more reliable and cautious strategies, especially in mid-to-late game states, where the cost of overestimating a move is highest.

A critical advancement in the DDQN approach is explicitly considering all possible moves the opponent might make. At each decision step, the agent evaluates Q-values for potential future board states arising from every possible opponent move, allowing the network to anticipate various opponent responses and thus improve its strategic depth.

Integrating a Min-Max strategy further enhances the agent's strategic capability by evaluating the current state's value through an averaging approach. At each decision point, the agent computes an expected value by averaging Q-values across potential opponent responses, employing a Min-Max inspired framework. This approach effectively merges classical game-theoretic decision-making with neural-network-based value estimation, creating a robust and strategically effective mechanism for selecting actions.

DDQN Training Approach

Our training pipeline is designed to stabilize learning and mitigate overestimation bias. Below we will walk through the key components of our implementation

Network:

Our Neural Network architecture had two convolutional layers (64→128 filters, 3×3 kernels, ReLU) followed by a Flatten→Dense(256, ReLU) block and a final linear output layer of size 7.

LayerNormalization after the dense hidden layer. Both “online” and “target” networks share this architecture; the target is periodically synced to the online’s weights. In total it had about 1.4 million trainable parameters. This type of architecture is fairly large compared to a simple network, but not as large as previous networks that mimic MCTS gameplay.

Loss and Optimizer:

We chose to go with Huber loss between our target Q-values and the online network’s predictions, weighted by the importance-sampling weights from our prioritized buffer. The Huber loss acts like mean squared error (MSE) for small residuals but switches to mean absolute error (MAE) for large residuals, preventing any single huge TD-error from dominating the gradient and derailing learning. Also, because we sample transitions according to their priority (i.e. magnitude of past TD-errors), our mini-batches are no longer drawn uniformly from the replay buffer. Multiplying each sample’s loss by its importance-sampling weight effectively re-weights the gradient contributions back toward the true, uniform distribution. This keeps our Q-updates unbiased in expectation, ensuring principled convergence.

Gradients are clipped to a global norm of 0.5 to prevent exploding updates. Clipping the combined gradient norm to 0.5 caps the maximum update size, stopping any single batch from sending weights on a wild swing. Clipping allows us to safely use a moderately aggressive learning rate without risking numerical instability.
We optimize our network with AdamW at a learning rate of 3e-4.

Parameters:

We set a discount factor $\gamma = 0.99$, plan to run 5,001 self-play episodes, and decide that for the first 400 games the agent’s opponent will be purely random. We allocate a prioritized replay buffer of size 5 000 with prioritization exponent $\alpha = 0.6$, and delay training until at least 400 transitions have been collected (‘wait time’). Each gradient update will use a mini-batch of 64 samples, and every 30 updates we will synchronize our target network from the online network to stabilize bootstrap targets. We also schedule an exponential ε-greedy decay: ε starts at 1.0 for 20 % of episodes, then decays to 0.2 over the next 50 %, and remains at 0.2 thereafter. Finally, we prepare containers to log losses, win/draw/loss counts, cumulative win rates, and episode indices.

Training loop:

Each episode begins by sampling the current ε from our precomputed schedule and randomly assigning the agent to play “plus” or “minus.” If we’re still in the first 400 episodes, the opponent is our `RandomOpponent` (uniformly random legal moves); beyond episode 400, the opponent is drawn at random from an ensemble of eight pre-trained CNN agents. We then reset our `Connect4Env`, which returns an empty 6×7×2 board and sets `env.turn` to the agent's color. We also initialize two placeholders, `last_state` and `last_action`, to `None`—these will let us retroactively penalize an agent move if the opponent immediately wins on the following turn.

Within each game, we alternate between the agent's move and the opponent's move until the game ends. On the agent's turn, we call `select_action(online, state, ε)`—which with probability ε picks a random legal column, and otherwise chooses the action with highest Q-value. We save `(state, action)` into `last_state`/`last_action`, then step the environment to obtain `(next_state, reward, done, info)`. That transition `(state, action, reward, next_state, done)` is immediately appended into the prioritized buffer. On the opponent's turn, we run `cnn_choose_move` (or random), step the environment again, and—if that move ends the game in our loss—we insert a special transition that gives our previously saved move a reward of –1 and marks it terminal. This ensures we explicitly teach "that move led to the opponent's win."

Once our buffer holds more than 400 transitions, we begin learning after each environment step. We perform three mini-batch updates per step: computing β = clip(0.4 + (episode / 5 001) × 0.6, 0.4, 1.0), then sample 64 transitions with their importance-sampling weights. For each batch we predict `current_q = online(s)` and copy it to `target_q`, then predict both `q_next_online = online(s′)` and `q_next_target = target(s′)`. Following the DDQN rule, for non-terminal samples we set:

$$\text{target\_q}[i, a_i] = r_i + \gamma\, q\_next\_target[i,\ \arg\max(q\_next\_online[i])]$$

and for terminals simply `target_q = r`. We compute absolute TD-errors, clip them into [1e-5, 10], and feed them back into `buffer.update_priorities` so that future samples focus on the most informative transitions. Finally, we run `train_step(s, target_q, weights)`, which applies a Huber loss weighted by those importance weights and performs gradient clipping before an AdamW update, and we log each loss.

Concurrently, a global `steps` counter tracks how many gradient updates we've performed. Every 30 updates we call `target.set_weights(online.get_weights())`, giving us a fresh but not too volatile target network for our next TD-target computations. This periodic syncing breaks the harmful feedback loop of having a constantly shifting target and is critical for stable value estimation.

To verify that improved loss actually translates to stronger play, every 100 episodes we pause exploration (ε = 0) and have our current online network play 50 games against the full opponent pool. We record wins, draws, and losses, update a running total of games and wins, and compute a cumulative win percentage. In parallel, we maintain a smoothed Huber-loss curve (averaged over the last 100 updates) and occasionally print the mean and max Q-values on a sample state. Together, these evaluation metrics give us real-time insight into whether our agent is truly learning to play better Connect-4, or simply overfitting to its replayed experiences.

DDQN Random Move Selection

To bootstrap learning and expose the agent to a wide variety of opening positions, we begin by having our DDQN agent face a completely random opponent for the first 400 episodes. Concretely, we implement a RandomOpponent class whose predict(board) method returns equal (non–∞) scores for every legal column, forcing uniformly random play. During these episodes (ep < episodes_vsRandom), regardless of the agent's ε‑greedy sampling, the opponent never applies any learned strategy. Meanwhile, on the agent's own turns we still use ε‑greedy: with probability ε the agent picks a random legal column, and otherwise selects the column with highest Q‑value from the online network. By front‑loading 400 games of pure randomness, we quickly fill the replay buffer with 400+ diverse transitions, ensuring our early gradient updates are based on a broad sample of game dynamics rather than a narrow slice of mid‑game positions. Once past episode 400, the opponent switches to a randomly chosen member of our pre-trained‑CNN ensemble, but even then the agent's occasional random moves (ε ≥ 0.2) continue to inject exploration throughout training.

Calculating Discounted Rewards

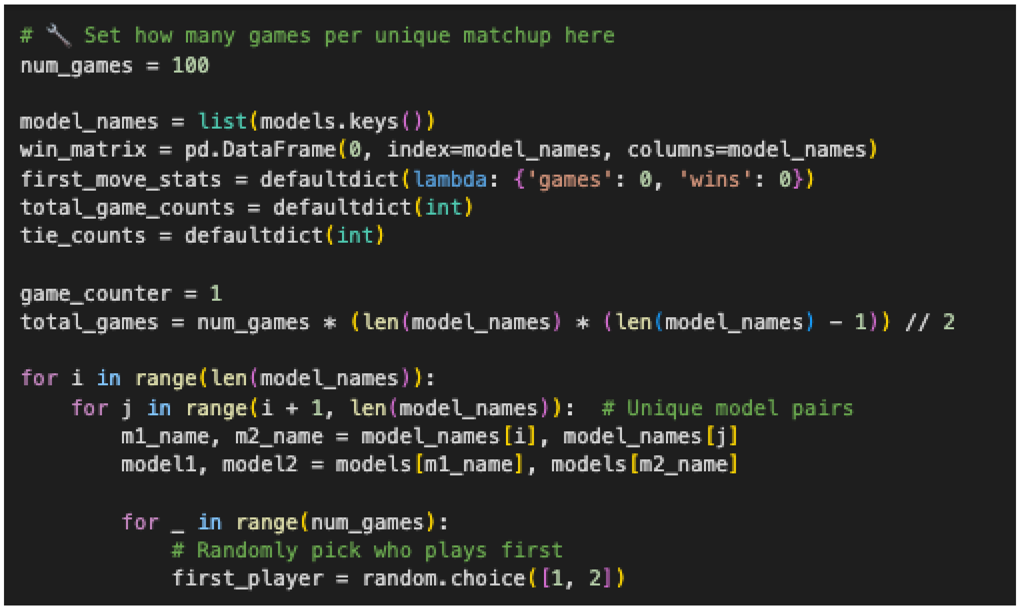
In our DDQN framework we do not accumulate full‑episode returns; instead each transition is updated via a one‑step Bellman backup that incorporates future value estimates discounted by γ = 0.99. During each mini‑batch update, we unpack sampled transitions (s, a, r, s′, done) and predict current Q‑values Q_online(s) as well as next‑state estimates from both Q_online(s′) and the frozen Q_target(s′). For terminal transitions (done=True), the learning target is simply the observed reward r. For non-terminal transitions we apply the Double-DQN target rule. We choose the best next action using the online network, but evaluate its value with the target network, forming:

$$y = r \ + \ \gamma \, Q_{\text{target}}(s\prime, \ a^{*})$$

This one-step backup effectively "propagates" future rewards one move at a time, devaluing distant outcomes by powers of γ. By repeatedly sampling transitions and regressing our online Q-network toward these γ-discounted targets—while weighting each loss by importance‑sampling coefficients—we teach the agent to anticipate long-term consequences without ever needing to sum entire trajectories.

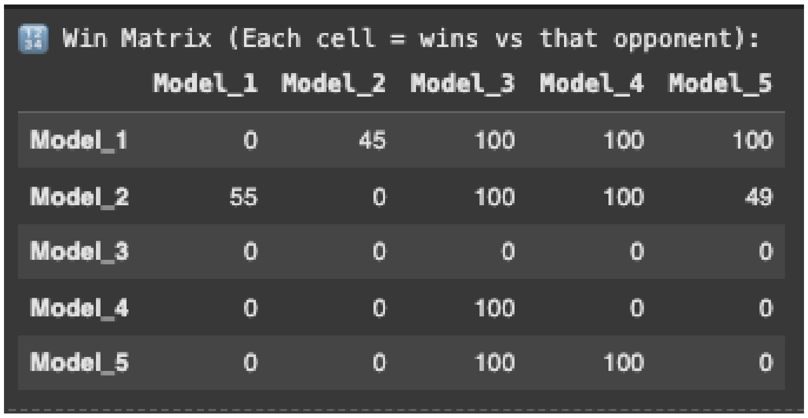
4. Results and Analyses

Round Robin Tournament - Picking M1



```
# 🔧 Set how many games per unique matchup here
num_games = 100

model_names = list(models.keys())
win_matrix = pd.DataFrame(0, index=model_names, columns=model_names)
first_move_stats = defaultdict(lambda: {'games': 0, 'wins': 0})
total_game_counts = defaultdict(int)
tie_counts = defaultdict(int)

game_counter = 1
total_games = num_games * (len(model_names) * (len(model_names) - 1)) // 2

for i in range(len(model_names)):
    for j in range(i + 1, len(model_names)):  # Unique model pairs
        m1_name, m2_name = model_names[i], model_names[j]
        model1, model2 = models[m1_name], models[m2_name]

        for _ in range(num_games):
            # Randomly pick who plays first
            first_player = random.choice([1, 2])
```

This code implements the core logic of a round-robin tournament where every unique pair of models plays 100 games against each other. The nested loops generate all possible model matchups without repetition, and for each game, the players alternate who goes first in order to ensure fairness. This setup allows for a comprehensive comparison across models by capturing head-to-head performance in varied playing conditions.



🔢 Win Matrix (Each cell = wins vs that opponent):

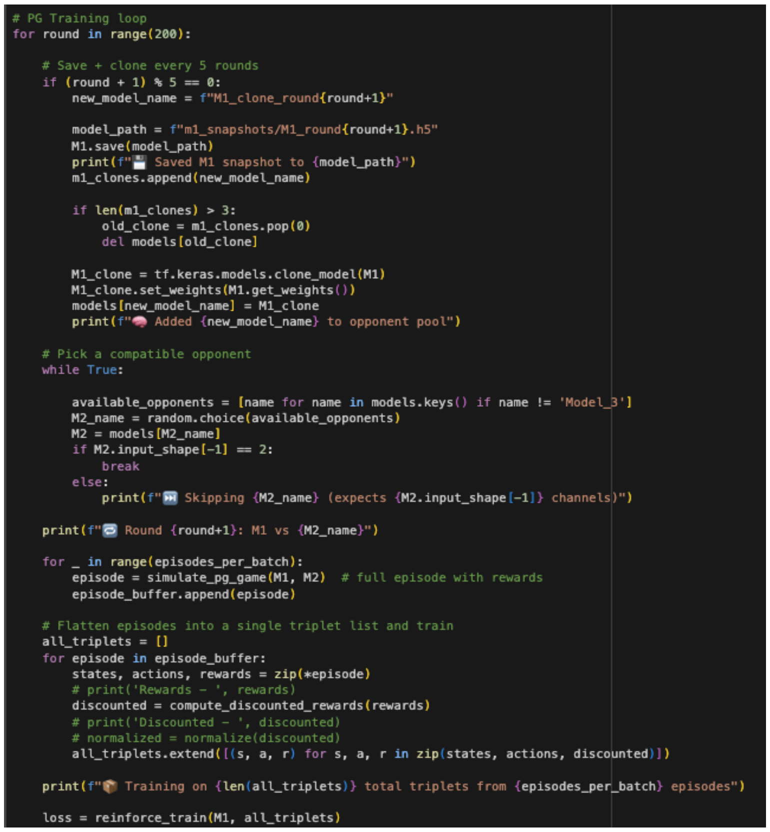
	Model_1	Model_2	Model_3	Model_4	Model_5
Model_1	0	45	100	100	100
Model_2	55	0	100	100	49
Model_3	0	0	0	0	0
Model_4	0	0	100	0	0
Model_5	0	0	100	100	0

The win matrix above shows the results of the round-robin tournament, where each cell represents the number of games a model won against a specific opponent. We can see that Model_1 won 45 games against Model_2, and 100 games each against Models 3, 4, and 5. Model_1 consistently outperformed every other model, winning 445 out of 500 total games. In contrast, all other models either lost most of their games or failed to beat stronger opponents.

This dominant performance across all matchups clearly identifies Model_1 as the strongest and most reliable model in the pool.

Performance of Policy Gradient

To improve M1, we applied the Policy Gradient (REINFORCE) strategy, where M1 plays self-play games against a variety of opponents. During each game, M1 selects moves probabilistically based on its softmax outputs, while opponents (such as M2) follow stronger logic like blocking wins or completing their own. After each batch of games, M1's policy is updated using the collected rewards to reinforce actions that led to better outcomes.



```python
# PG Training loop
for round in range(200):

    # Save + clone every 5 rounds
    if (round + 1) % 5 == 0:
        new_model_name = f"M1_clone_round{round+1}"

        model_path = f"m1_snapshots/M1_round{round+1}.h5"
        M1.save(model_path)
        print(f"💾 Saved M1 snapshot to {model_path}")
        m1_clones.append(new_model_name)

        if len(m1_clones) > 3:
            old_clone = m1_clones.pop(0)
            del models[old_clone]

        M1_clone = tf.keras.models.clone_model(M1)
        M1_clone.set_weights(M1.get_weights())
        models[new_model_name] = M1_clone
        print(f"🧠 Added {new_model_name} to opponent pool")

    # Pick a compatible opponent
    while True:

        available_opponents = [name for name in models.keys() if name != 'Model_3']
        M2_name = random.choice(available_opponents)
        M2 = models[M2_name]
        if M2.input_shape[-1] == 2:
            break
        else:
            print(f"⏭ Skipping {M2_name} (expects {M2.input_shape[-1]} channels)")

    print(f"🔁 Round {round+1}: M1 vs {M2_name}")

    for _ in range(episodes_per_batch):
        episode = simulate_pg_game(M1, M2)  # full episode with rewards
        episode_buffer.append(episode)

    # Flatten episodes into a single triplet list and train
    all_triplets = []
    for episode in episode_buffer:
        states, actions, rewards = zip(*episode)
        # print('Rewards - ', rewards)
        discounted = compute_discounted_rewards(rewards)
        # print('Discounted - ', discounted)
        # normalized = normalize(discounted)
        all_triplets.extend([(s, a, r) for s, a, r in zip(states, actions, discounted)])

    print(f"📦 Training on {len(all_triplets)} total triplets from {episodes_per_batch} episodes")

    loss = reinforce_train(M1, all_triplets)
```

Policy Gradient Training Loop

In this training loop, M1 was improved over 200 rounds using the policy gradient approach. Every 5 rounds, the current version of M1 was saved and cloned, with these snapshots added to the pool of opponents to gradually increase training difficulty. Only the three most recent clones were kept active to manage memory usage.

For each round, a random compatible opponent (M2) was selected from the pool, and M1 played 50 games against it. In these games, M1's moves were chosen based on its current softmax policy, while M2 played more logically by looking for winning and blocking moves.

After the games, discounted rewards were computed for each move to emphasize the long-term value of the earlier moves. The collected (state, action, reward) triplets were then used to update M1's policy via the REINFORCE algorithm, helping the agent learn to favor the actions that led to better outcomes and gradually improve its decision-making.

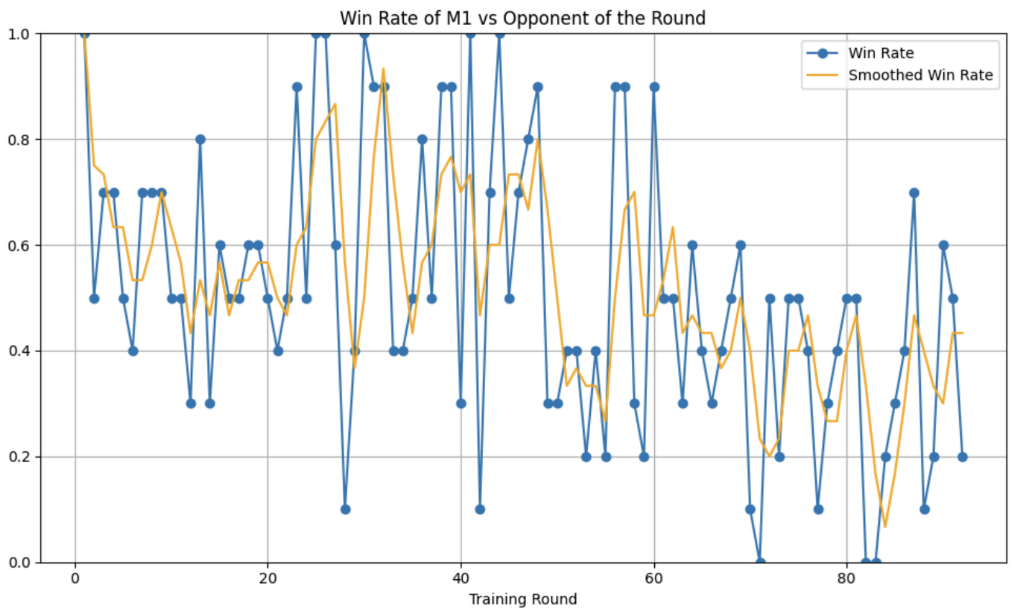


Fig - Training performance of Policy gradient - model (Y-axis - win rate based on 50 games, X-axis: training round).

From the graph above, at the start of training (rounds 0–20), the win rate fluctuates a lot but tends to stay between 50% and 70%. This suggests that M1 quickly learned some basic strategies but was still inconsistent.

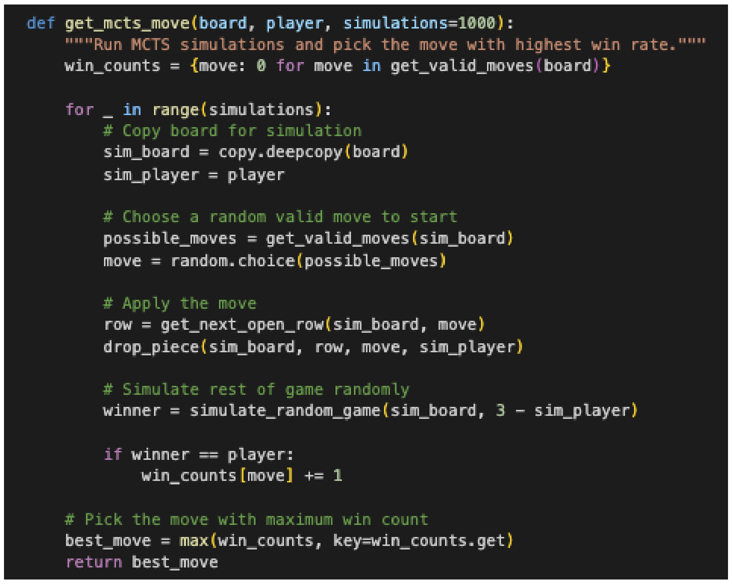
During the middle of training (rounds 20–60), M1 reached higher peaks in performance, sometimes winning up to 90–100% of games against certain opponents. However, there were also sudden drops, showing that M1 struggled against stronger or newly introduced opponents, especially the clones of itself.

In the later rounds (after round 60), the win rate overall became lower and more unstable, hovering around 30% to 50%. This is because the opponents became harder as training progressed, making it more difficult for M1 to consistently win.

Overall, the graph shows that Policy Gradient helped M1 improve its gameplay early on, but training became harder as stronger opponents were introduced. Although M1 learned effective strategies, it was not always able to adapt quickly to harder situations, which is a common challenge when training reinforcement learning agents through self-play.

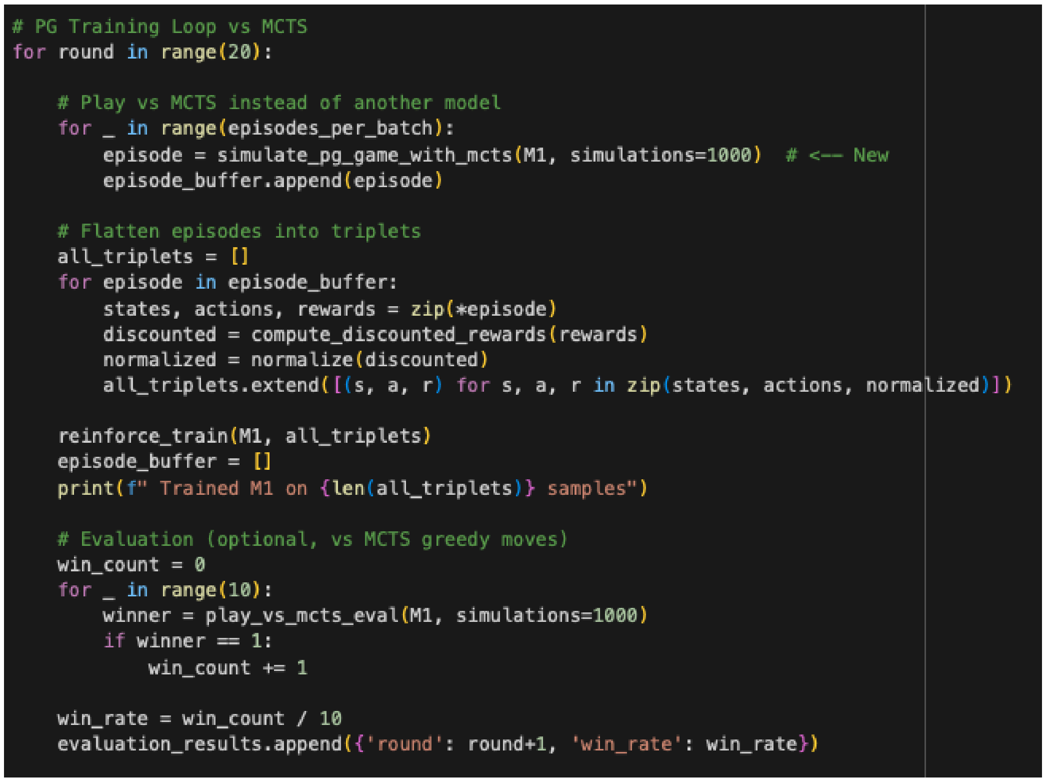
Performance of Policy Gradient Against MCTS

In order to evaluate M1's performance, we retrained it using Policy Gradient against a MCTS player. The MCTS opponent simulated 1000 random games per move and always chose the action with the highest estimated win rate, making it a much more strategic challenge than the previous neural network opponents.



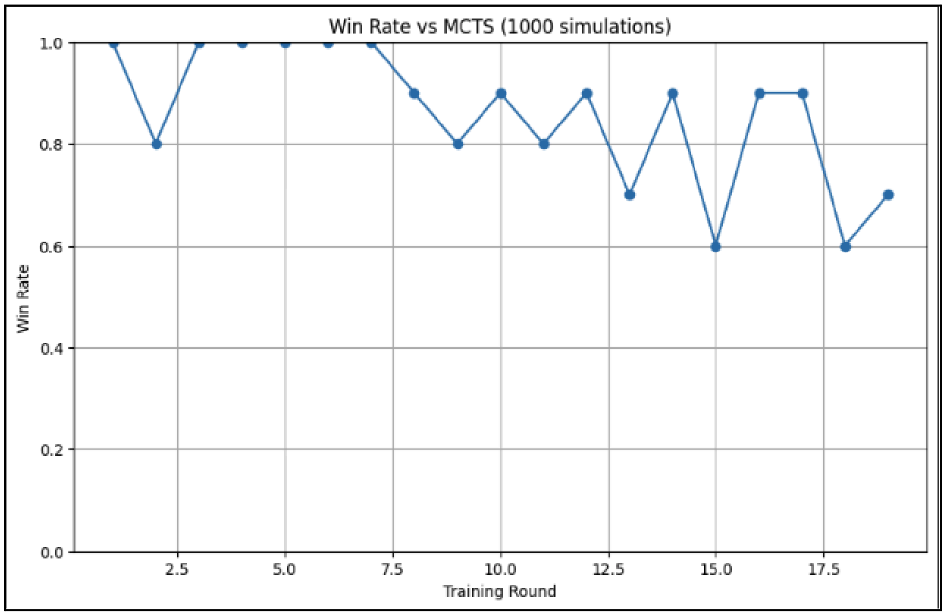
```python
def get_mcts_move(board, player, simulations=1000):
    """Run MCTS simulations and pick the move with highest win rate."""
    win_counts = {move: 0 for move in get_valid_moves(board)}

    for _ in range(simulations):
        # Copy board for simulation
        sim_board = copy.deepcopy(board)
        sim_player = player

        # Choose a random valid move to start
        possible_moves = get_valid_moves(sim_board)
        move = random.choice(possible_moves)

        # Apply the move
        row = get_next_open_row(sim_board, move)
        drop_piece(sim_board, row, move, sim_player)

        # Simulate rest of game randomly
        winner = simulate_random_game(sim_board, 3 - sim_player)

        if winner == player:
            win_counts[move] += 1

    # Pick the move with maximum win count
    best_move = max(win_counts, key=win_counts.get)
    return best_move
```

This defines the MCTS opponent. Simulates the 1000 games for each possible move, keeps track of the number of wins for each of the moves, and picks the move with the highest win count.



```python
# PG Training Loop vs MCTS
for round in range(20):

    # Play vs MCTS instead of another model
    for _ in range(episodes_per_batch):
        episode = simulate_pg_game_with_mcts(M1, simulations=1000)  # <-- New
        episode_buffer.append(episode)

    # Flatten episodes into triplets
    all_triplets = []
    for episode in episode_buffer:
        states, actions, rewards = zip(*episode)
        discounted = compute_discounted_rewards(rewards)
        normalized = normalize(discounted)
        all_triplets.extend([(s, a, r) for s, a, r in zip(states, actions, normalized)])

    reinforce_train(M1, all_triplets)
    episode_buffer = []
    print(f" Trained M1 on {len(all_triplets)} samples")

    # Evaluation (optional, vs MCTS greedy moves)
    win_count = 0
    for _ in range(10):
        winner = play_vs_mcts_eval(M1, simulations=1000)
        if winner == 1:
            win_count += 1

    win_rate = win_count / 10
    evaluation_results.append({'round': round+1, 'win_rate': win_rate})
```

In each training round, M1 played 10 games against MCTS with random initial openings. The resulting gameplay data was used to perform a policy gradient update. After every training round, M1's performance was evaluated by playing another set of 10 games against MCTS, and the win rate was recorded. This setup allowed us to track how well M1 was able to adapt to and defeat a search-based opponent over time.

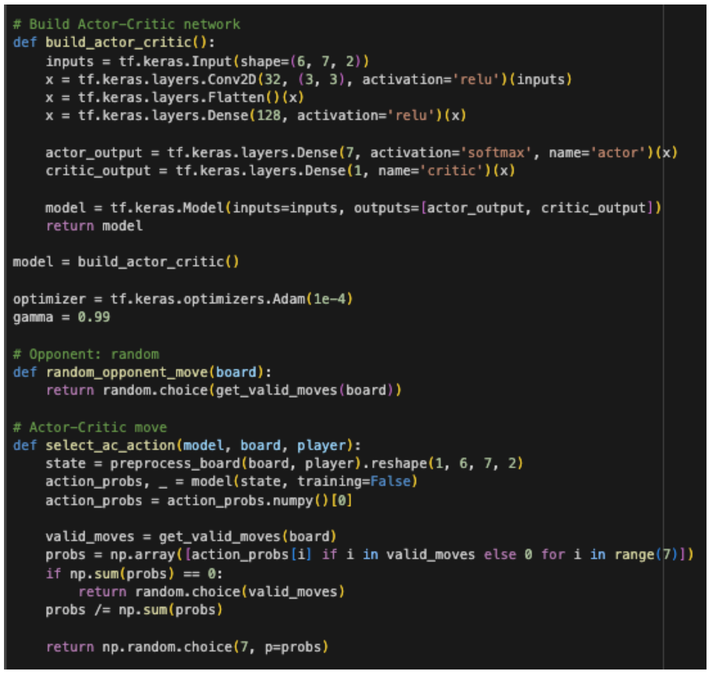


The graph shows that M1 initially performed very strongly against MCTS, winning most games during the first few rounds. This indicates that even before heavy retraining, M1's learned policy was competitive against a search-based player relying on random simulations. However, as training continued, the win rate gradually decreased, with more variability appearing in later rounds. Around Rounds 10–15, win rates dropped to approximately 60–70%, and while some recovery was observed, M1 did not consistently return to its early dominant performance.

This shows that MCTS as an opponent is difficult to beat. The Policy Gradient training does introduce a lot of exploration noise, which can make it hard for the agent to consistently improve when facing stronger opponents.

Actor Critic

Actor-Critic Reinforcement Learning was also used to improve the agent's gameplay against a random opponent. In the set-up below, the agent uses a shared network where the actor predicts move probabilities, and the critic estimates the value of the board state.



```
# Build Actor-Critic network
def build_actor_critic():
    inputs = tf.keras.Input(shape=(6, 7, 2))
    x = tf.keras.layers.Conv2D(32, (3, 3), activation='relu')(inputs)
    x = tf.keras.layers.Flatten()(x)
    x = tf.keras.layers.Dense(128, activation='relu')(x)

    actor_output = tf.keras.layers.Dense(7, activation='softmax', name='actor')(x)
    critic_output = tf.keras.layers.Dense(1, name='critic')(x)

    model = tf.keras.Model(inputs=inputs, outputs=[actor_output, critic_output])
    return model

model = build_actor_critic()

optimizer = tf.keras.optimizers.Adam(1e-4)
gamma = 0.99

# Opponent: random
def random_opponent_move(board):
    return random.choice(get_valid_moves(board))

# Actor-Critic move
def select_ac_action(model, board, player):
    state = preprocess_board(board, player).reshape(1, 6, 7, 2)
    action_probs, _ = model(state, training=False)
    action_probs = action_probs.numpy()[0]

    valid_moves = get_valid_moves(board)
    probs = np.array([action_probs[i] if i in valid_moves else 0 for i in range(7)])
    if np.sum(probs) == 0:
        return random.choice(valid_moves)
    probs /= np.sum(probs)

    return np.random.choice(7, p=probs)
```

The code above builds the Actor-Critic Model. The actor outputs a probability distribution over the seven possible columns to play, while the critic outputs a single value estimating how favorable the current board state is. This setup allows the agent to learn both action selection and state evaluation together during training.

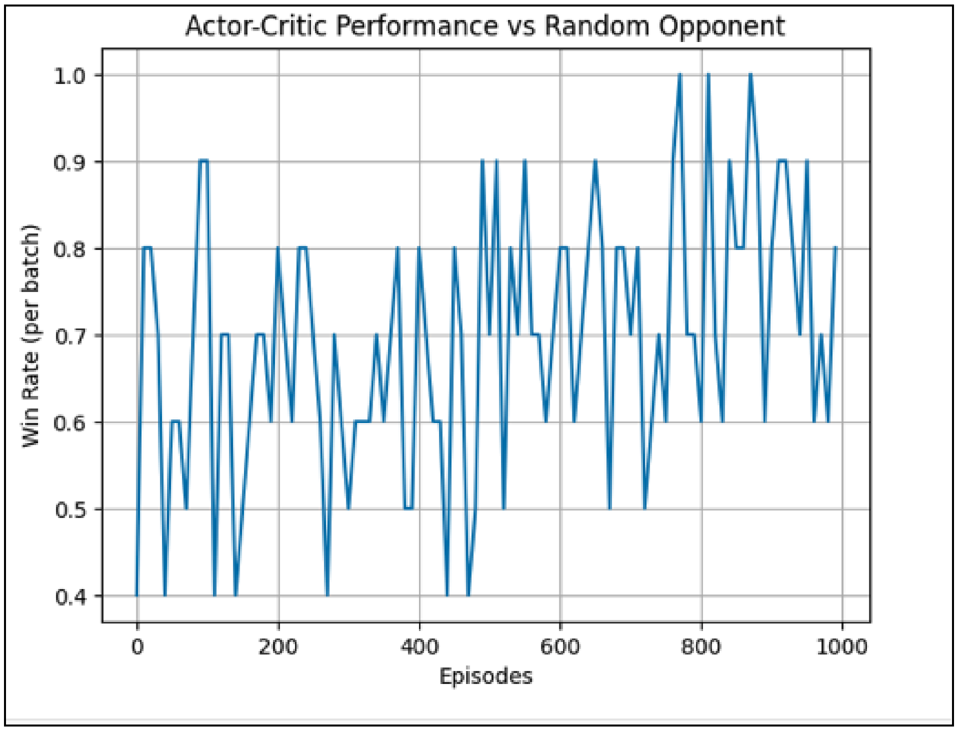


Fig - Training performance of actor critic

The graph shows the win rate of the Actor-Critic model as it trained against a random opponent. At the beginning of training (0–200 episodes), the agent's performance was unstable, fluctuating between 40% and 90% win rates. This variability reflects the exploration phase, where the agent is still learning which actions are better.

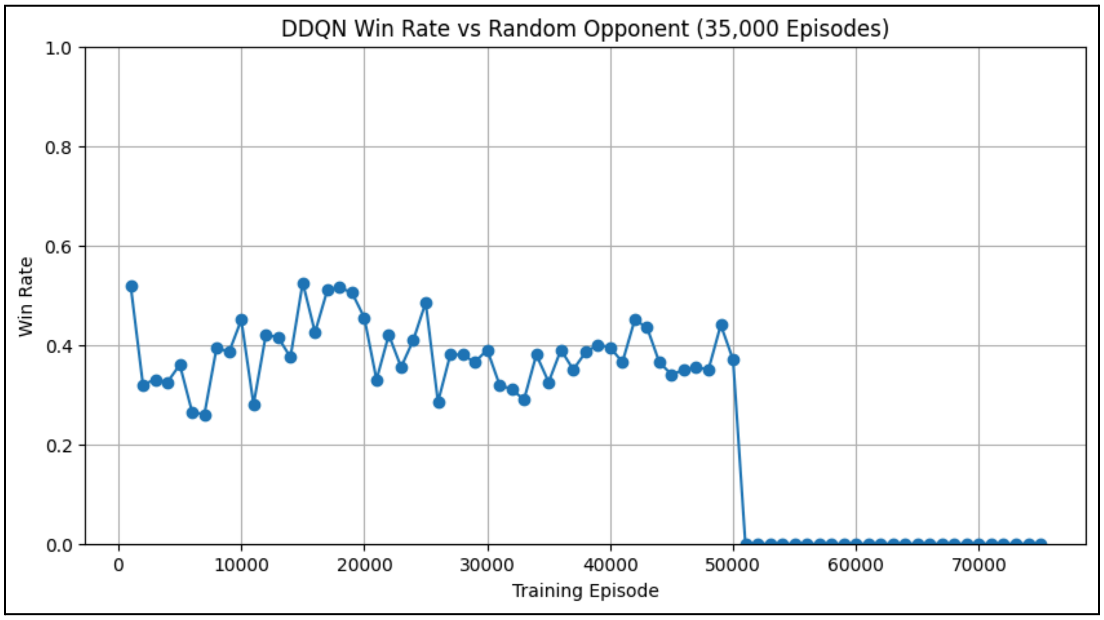
From about 200 to 600 episodes, the performance becomes slightly more consistent, showing fewer sharp drops and a slow upward trend. After 600 episodes, the agent's win rate improves significantly, frequently reaching 90%–100% wins per batch.

This trend shows that the Actor-Critic method helped M1 gradually learn better strategies, becoming much more effective at exploiting the random opponent by the end of training.

Performance of DDQN

Throughout the project, multiple DDQN training strategies were explored, including training against random opponents, Monte Carlo Tree Search (MCTS) opponents, and ensembles of pretrained CNN agents. Across all approaches, the DDQN models demonstrated limited success. While early training against random opponents occasionally showed moderate win rates, typically fluctuating between 30% and 50%, these results were inconsistent and failed to translate into robust gameplay improvements. When evaluated against stronger opponents such as MCTS agents, the DDQN models struggled significantly, often achieving near-zero win rates.

Despite experimenting with different hyperparameters, exploration strategies, and opponent types, none of the DDQN models were able to consistently outperform the Policy Gradient models. The Policy Gradient approach, in contrast, demonstrated clearer improvements over time and stronger adaptability to increasingly difficult opponents. These results suggest that, in the context of Connect 4, policy-based methods offered a more stable and effective learning pathway than value-based methods like DDQN.

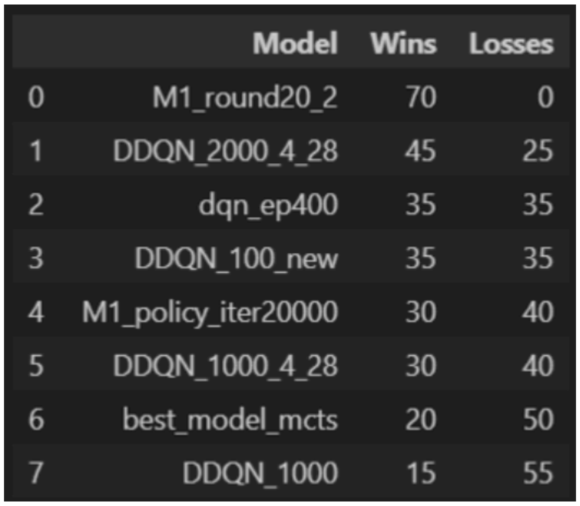


This graph is one such example of the relatively poor performance of an early-stage DDQN. For the first 50,000 episodes, the DDQN was evaluated against random opponents, and the win rate hovered between 30% and 50%. The assumption was that this would be enough time to learn the right moves, and start beating a random opponent 100% of the time, after which we would evaluate it against our MCTS model. However, as the chart shows, the DDQN was struggling to beat the random opponent even after 50,000 episodes, and was unable to beat the MCTS trained CNN even a single time.

Looking ahead, there are several directions that could be explored to improve DDQN performance. Incorporating more sophisticated exploration strategies, such as Noisy Networks or entropy-based regularization, could encourage deeper exploration of strategic moves. Additionally, modifying the reward structure to provide intermediate feedback (e.g., for setting up three-in-a-row patterns) might help guide learning more effectively. Expanding the opponent pool to include adaptive or curriculum-based opponents that progressively increase in difficulty could also make training more robust.

Comparative Analysis: PG vs DDQN

When comparing the Policy Gradient (PG) and Double Deep Q-Learning (DDQN) approaches, Policy Gradient consistently demonstrated stronger and more reliable performance. The PG models showed steady improvements during self-play training, successfully adapting to increasingly stronger opponents and achieving competitive win rates even against search-based players like MCTS. In contrast, the DDQN models exhibited unstable learning curves, struggled to consistently defeat even random opponents, and were unable to match the strategic depth achieved through policy-based learning.



	Model	Wins	Losses
0	M1_round20_2	70	0
1	DDQN_2000_4_28	45	25
2	dqn_ep400	35	35
3	DDQN_100_new	35	35
4	M1_policy_iter20000	30	40
5	DDQN_1000_4_28	30	40
6	best_model_mcts	20	50
7	DDQN_1000	15	55

Several factors likely contributed to this disparity. Policy Gradient methods directly optimized action probabilities based on game outcomes, allowing for smoother and more targeted policy improvements. DDQN, on the other hand, relied on estimating future rewards through Q-values, which introduced additional challenges such as instability in value estimation and sensitivity to opponent behavior. In a highly strategic and sequential environment like Connect 4, where small mistakes can have cascading effects, these issues made DDQN less effective. Overall, Policy Gradient methods proved to be a more suitable framework for evolving strong, adaptable gameplay strategies in this project.

5. Challenges and Takeaways

One of the primary challenges we encountered during this project was ensuring consistent and meaningful improvement in our models' gameplay performance. With the policy gradient agent, we observed early gains during training, particularly when facing weaker opponents, however, as the agent was tested against stronger models such as MCTS agents, its performance was not as good. A major contributor to this limitation was the exploratory noise present in policy gradient methods. While exploration is critical for discovering better strategies, the extra noise caused the agent to struggle against highly optimized opponents, leading to inconsistent policy updates and slower convergence to more strategic behavior.

The DDQN model presented a different set of challenges, and we found it to perform worse than our policy gradient model. Although the DDQN agent performed well against random or less strategic opponents, it consistently underperformed when competing against MCTS. This suggests that the DDQN model focused heavily on easy plays rather than developing deeper strategic foresight. When we played against this model, we noticed that it seemed to favor starting in the left most column, making it easy to beat early on and in a few moves. Without sufficient experience against strong and diverse opponents, the DDQN model likely overfits short-term rewards instead of building a robust understanding of delayed consequences.

When directly comparing the two models, the policy gradient agent outperformed the DDQN agent across almost all evaluations, particularly in games requiring multi-turn strategic thinking. The policy gradient model was better able to recognize threats, block opponent wins, and set up multi-step strategies, whereas the DDQN model tended to prioritize immediate, short-sighted gains. This difference highlights the strengths of policy gradient methods in situations where one may need to think and plan about the future and future moves, while also exposing the difficulty that value-based methods like DDQN face in non-stationary training settings.

The challenges illustrated by both of these models show that the quality and diversity of the opponent pool available heavily influences the learning process of the model. Models that only train against weak opponents tend to remain weak, overfit, and do not play well against stronger players/models. Balancing exploration with exploitation continues to be an important aspect, especially in situations where the opponent's difficulty is constantly changing.

For future improvement, several strategies could be beneficial. Using curriculum learning would allow a model to begin by playing weaker models, and gradually begin to play against stronger opponents. This would allow the model to gradually learn from different opponents, instead of having it play against random opponents or extremely strong opponents. It would also be beneficial to increase the size of the pool of opponents. This would provide the model with more opponents to play against, and therefore increase the possibility of the model learning more advanced techniques and different patterns.

This project allowed us to see the value of utilizing reinforcement to improve models in different environments. One of the key takeaways was understanding the importance of balancing exploration and exploitation. We learned that model performance is dependent on the opponents faced during training, and that this must be taken into account in order to create a better model. Additionally, using both policy gradient and DDQN shows the importance of testing different methods for strategic development. This project emphasized the importance of designing an effective training process to balance a variety of factors. The lessons learned from this project can be applied to more than just Connect 4, such as real world decision making systems and robotics.

6. Conclusion

In this project, we explored the use of reinforcement learning techniques, specifically policy gradients and double deep Q-learning, to improve the gameplay of a Connect 4 agent through self-play and iterative refinement. We began by building on a strong baseline model, trained initially with Monte Carlo Tree Search data and refined through architectural improvements. Through a detailed comparison of policy gradient and DDQN methods, we observed that while both models showed some capacity to learn, the policy gradient model was significantly more successful at developing strategic and adaptive gameplay, particularly against more sophisticated opponents. The DDQN model, although capable against random opponents, struggled when facing stronger, more strategic adversaries like those guided by MCTS. When comparing the two models in terms of their success against opponents, the policy gradient performed better than DDQN.

Throughout the development process, we encountered challenges common to reinforcement learning, such as instability, high exploratory noise, and difficulty in maintaining steady improvement when the competition improved. These challenges highlight the importance of thoughtful training environment design, opponent diversity, and careful algorithmic tuning.

Based on our experiences and results, it is clear that reinforcement learning is a powerful approach to developing intelligent decision-making systems. Hiring a reinforcement learning expert would be highly beneficial for a company aiming to build intelligent agents, optimize dynamic processes, or solve problems that involve sequential decision making. An RL expert can help navigate the challenges of algorithm selection, environment design, reward shaping, and model evaluation, in order to replicate a real world scenario. Reinforcement learning is used in a variety of industries, and hiring an RL expert would strengthen how a company makes decisions, allowing them to outperform their competitors.