RM 294 Project 3: Applying Deep Q-Learning and Policy Gradients to Connect4

Prepared For:

Dr. Daniel Mitchell
McCombs School of Business
The University of Texas at Austin
2110 Speedway Austin, TX 78705

Monday, April 28th, 2025

Project Overview

Single player games require competent, flexible computer program opponents. No one wants to play a game of Pong, chess or Mario Kart in which your opponent always makes the same moves. Reinforcement learning (RL) is a technique used across autonomous vehicle training, real-time bidding and robotics manipulation. It enables learning without a clearly defined external reward function; we can only learn a good move by interacting with the environment. As part of our exploration into enhancing single player gaming experiences, this project evaluates the feasibility of reinforcement learning (RL) to develop intelligent and robust AI video game opponents. By training AI models to play competitively against other AI models, we can create challenging and realistic opponents that enhance single player gameplay. To assess this approach, we have developed and compared RL models that play the popular board game *Connect 4*. This approach was popularized by the DeepMind Technologies team that created AlphaGo, the first computer Go program to beat a professional Go player.

Specifically, we evaluated three RL models: a Policy Gradient model, a Double Deep Q Network model and an Advantage Actor-Critic model (A2C). The Policy Gradient model directly optimizes the probability of choosing correct actions by using the value of future rewards to adjust the model's output distribution after each game. The Double Deep Q Network model learns the estimate of the expected future reward for each possible action and improves by selecting actions that maximize these predicted values. A2C combines the best of these two approaches. This report will expand on the theoretical underpinnings of each approach, how we implemented them, and how they performed against Monte-Carlo Tree Search *Connect 4* opponents.

Methodology

This project builds upon prior work in which convolutional neural networks and transformers were trained to replicate expert strategies generated by Monte Carlo Tree Search (MCTS) algorithms. Expanding on this, our goal was to apply reinforcement learning techniques to develop agents that could learn and improve through self play and dynamic opponent matchups.

To provide some background, *Connect 4* is a player board game played on a 6 x 7 grid in which players take turns dropping checker pieces into a chosen column. The objective is to be the first player to align four pieces in a row vertically, horizontally, or diagonally. If the board fills up without there being a winning move, then the game is considered a tie.

To evaluate RL strategies, we compared three approaches: a Policy Gradient model, a double Deep Q Learning model, and an Advantage Actor-Critic model that combines the best of the first two frameworks. Each of the following subsections will provide theoretical context for the models, as well as how they are used to create better *Connect 4* opponents.

Policy Gradients

Policy gradients train a classification neural network to predict the probability distribution that each action is the best action, given a current board state. The objective is to increase the likelihood of actions that lead to high cumulative rewards, while decreasing the likelihood of actions that perform poorly (e.g. not blocking an opponent). They maximize the following objective:

$$\max_{\theta} \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^{T} r_t \right]$$

During training, the model is evaluated by playing full episodes under the current policy, without updating the weights immediately. After each episode, the agent computes the discounted cumulative reward Gt for each time step:

$$G_t = r_t + \delta r_{t+1} + \delta^2 r_{t+2} + \delta^3 r_{t+3} + \cdots$$

Where delta is the discount factor that determines how much future rewards are valued relative to immediate rewards. This is similar to the time discounting money, but is mainly used to assign rewards to actions that do not lead to immediate point scoring. The model should be rewarded for moves that help it win a game down the road, and additionally rewarded for moves that help it win quickly.

Training occurs on-policy: trajectories are collected under the current policy, and the policy is updated after the batch of trajectories has been played. States are sequentially correlated within episodes, and rewards are discounted over time, which results in earlier actions in an episode influenced less by rewards that happen far in the future. Memory buffers are used to smooth gradient estimates and overcome some of the challenges posed by state correlations.

Implementation

We use a model from a previous project as the starting point for our reinforcement learning. This model will be referred to as M1 throughout this report. We use another pre-trained model as an initial opponent, M2. M1 and M2 played several self-play games and for each game the starting player was chosen at random. Moves were sampled according to the model's probability distribution. Games were optionally initialized with a few random moves. After each game, discounted rewards for M1's moves were calculated. A random sample of (board, move, reward) triplets was used to perform a single step of gradient descent.

The Policy Gradient model optimized the model's move selection probabilities directly based on the discounted sum of future rewards received at the end of each game. After completing a game, the model updated its probability over available legal moves, rewarding decisions that led to successful outcomes. This reinforcement learning approach followed an offline learning structure as the model interacted with the environment, accumulated experiences, and performed weight updates after the full game had been completed.

Opponent Pool and Gameplay Logic

The model under training, referred to as M1, competed against opponents drawn randomly from a curated pool. Initially, the opponent pool included two Monte Carlo Tree Search models, MCTS500 and MCTS750, and a convolutional neural network, referred to as M2. The M2 model was developed during a preceding project, where neural networks were trained to play Connect 4 based on strategies generated by MCTS. These pre-existing opponents provided a challenging yet stable environment for M1's reinforcement learning process.

During each game, M1 selected moves based on the probabilities output by its network, masked to restrict choices to only legal columns. Opponents selected actions either through their MCTS search logic or their CNN outputs, depending on their type. Gameplay interactions were handled through functions such as *play_mcts* and *play_other_opp*, which managed the game state, action selection, turn switching, and recording of state-action pairs for later use.

Figure 1: *Opponent Selection and Gameplay Setup*

```
while total_games < 10000:

    # pick opponent
    if total_games % 10 == 0:
      opponent = random.choice(opponents)
    print(f"\n--- New Opponent Selected (game {total_games}) ---")

    bot_wins = 0  # reset every 10 games

    # Play 10 games against same opponent
    for _ in range(10):
        bot_color = random.choice(["red", "yellow"])

        if isinstance(opponent, tuple) and opponent[0] == "mcts":
            # MCTS opponent
            states, actions, winner = play_mcts(m1, bot_color=bot_color, mcts_level=opponent[1])
        else:
            # CNN opponent
            states, actions, winner = play_other_opp(m1, bot_color, opponent)
        if len(states) == 0:
            print(f"[Game {total_games}] No moves collected. Skipping training.")
            total_games += 1
            continue
```

Reward Structure, Updates and Opponent Pool Evaluation

Following each game, rewards were assigned based on the outcome. Victories were rewarded positively, losses negatively, and ties neutrally. To better distribute credit across the full game, immediate rewards were discounted backward in time, giving higher weight to moves closer to the outcome. This discounted reward was critical in shaping how the model learned from full sequences of actions.

Figure 2: *Discounted Reward Function*

```
def discount_rewards(r, delta=0.99):
    discounted_r = np.zeros_like(r, dtype=np.float32)
    running_add = 0
    for t in reversed(range(0, len(r))):
        running_add = running_add * delta + r[t]
        discounted_r[t] = running_add
    return discounted_r
```

For policy updates, TensorFlow's *GradientTape* functionality was employed. The loss function computed the negative log probabilities of selected actions, weighted by their corresponding discounted rewards.

Gradients of this loss were then calculated and used to perform stochastic gradient descent updates on M1's parameters, progressively improving its weights and biases.

Figure 3: *Policy Loss Calculation and Optimizer Step*

```
with tf.GradientTape() as tape:
    logits = m1(X, training=True)
    action_masks = tf.one_hot(y, depth=7)
    log_probs = tf.reduce_sum(tf.nn.log_softmax(logits) * action_masks, axis=1)
    loss = -tf.reduce_mean(log_probs * sample_weights)

gradients = tape.gradient(loss, m1.trainable_variables)
optimizer.apply_gradients(zip(gradients, m1.trainable_variables))

# Update counters
total_games += 1
if winner == bot_color:
    bot_wins += 1
print(f"Player {bot_color}, Opponent {opponent}, Winner = {winner}")
```

To ensure that training remained challenging, the strength of opponents evolved over time. Every 1000 games, M1 was evaluated against an MCTS1000 opponent. If it achieved a win rate greater than 70 percent, that version of M1 was saved and added to the opponent pool. When the number of CNN based opponents exceeded a fixed limit, older CNNs were removed to maintain pool diversity without excessive computational overhead.

```
# === Save and Promote if good enough ===
if mcts_win_rate >= current_best_win_rate + 0.03:
    current_best_win_rate = mcts_win_rate
    model_name = f"policy_grad_{save_counter}.h5"
    m1.save(model_name)
    print(f"Model promoted. Saved as {model_name}")

    new_opponent = tf.keras.models.load_model(model_name, custom_objects={'LeakyReLU': LeakyReLU})
    opponents.append(new_opponent)
    save_counter += 1

    # Maintain opponent pool size
    cnn_opponents = [opp for opp in opponents if not isinstance(opp, tuple)]
    if len(cnn_opponents) > max_cnn_opponents:
        # Remove oldest CNN
        for idx, opp in enumerate(opponents):
            if not isinstance(opp, tuple):
                removed = opponents.pop(idx)
                print(f"Removed oldest CNN opponent to maintain pool size.")
                break
```

Additionally, a random number of opening moves were made at the start of each game to force M1 to encounter a broader distribution of board states. This design choice prevented overfitting to common early-game patterns and encouraged generalizable play.

Model Initialization and Optimizer Setup

M1 and M2 were loaded from saved CNN architectures using a consistent LeakyReLU activation. This ensured stability across layers and mained comparability between models. M1 was actively trained, while M2 was pre-trained in the opponent pool alongside MCTS500 and MCTS750.

The optimizer was Adam with a relatively low learning rate (1e-4), chosen to encourage meaningful updates without risking large, destabilizing gradient steps. This setup allowed M1 to gradually refine its decision-making through thousands of RL games.

Move Selection, Reward Discounting, and Offline Updates

At each decision point, M1's probability outputs were masked to restrict moves to legal columns. M1 masked its logit output to allow only legal moves, ensuring compliance with the game rules. Instead of deterministically choosing the highest-probability move, actions were sampled according to the model's full output distribution, promoting broader exploration and preventing overfitting.

After each game was completed, immediate rewards were assigned based on the outcome. +1 for a win, -1 for a loss, and 0 for a tie. These rewards were discounted backward in time, using a standard discounting function with $delta = 0.99$.

```
def discount_rewards(r, delta=0.99):
    discounted_r = np.zeros_like(r, dtype=np.float32)
    running_add = 0
    for t in reversed(range(0, len(r))):
        running_add = running_add * delta + r[t]
        discounted_r[t] = running_add
    return discounted_r
```

Training proceeded in an offline manner: random batches of (board, move, reward) triplets were drawn, and TensorFlow's GradientTape was used to compute gradients of negative log-likelihood loss, weighted by discounted rewards. GradientTape was used to speed up training, as model.fit() can be very slow. The resulting gradients were used to update M1's weights and biases using SGD.

```
with tf.GradientTape() as tape:
    logits = m1(X, training=True)
    action_masks = tf.one_hot(y, depth=7)
    log_probs = tf.reduce_sum(tf.nn.log_softmax(logits) * action_masks, axis=1)
    loss = -tf.reduce_mean(log_probs * sample_weights)

gradients = tape.gradient(loss, m1.trainable_variables)
optimizer.apply_gradients(zip(gradients, m1.trainable_variables))

# Update counters
total_games += 1
if winner == bot_color:
    bot_wins += 1
print(f"Player {bot_color}, Opponent {opponent}, Winner = {winner}")
```

Dynamic Opponent Management and Game Diversification

To ensure the training environment remained challenging, a formal evaluation was triggered after every 30 games. M1 was evaluated by playing 20 games against a very strong MCTS1000 agent. The MCTS win rate was calculated using a moving average method, blending results from previous evaluations and the current batch.

```
# mcts100 evaluation
if total_games % 30 == 0:
    print(f"\n=== Formal MCTS1000 Evaluation at {total_games} games ===")
    mcts_wins = 0
    for _ in range(20):
        bot_color_eval = random.choice(["red", "yellow"])
        _, _, winner_eval = play_mcts_eval(m1, bot_color=bot_color_eval, mcts_level=1000)
        if winner_eval == bot_color_eval:
            mcts_wins += 1
    mcts_eval_games += 20
    mcts_win_rate = ((current_best_win_rate * (mcts_eval_games - 20)) + mcts_wins) / (mcts_eval_games + 20)

    print(f"MCTS1000 Win Rate: {mcts_win_rate:.2f}")
```

If M1's new win rate improved by at least 3% compared to the previous best (current_best_win_rate + 0.03), the model was saved and promoted to the opponent pool to provide new competition for future training games. To maintain a diverse but manageable pool, older CNN opponents were removed if the number exceeded the maximum allowed (max_cnn_opponents = 6).

```
# === Save and Promote if good enough ===
if mcts_win_rate >= current_best_win_rate + 0.03:
    current_best_win_rate = mcts_win_rate
    model_name = f"policy_grad_{save_counter}.h5"
    m1.save(model_name)
    print(f"Model promoted. Saved as {model_name}")

    new_opponent = tf.keras.models.load_model(model_name, custom_objects={'LeakyReLU': LeakyReLU})
    opponents.append(new_opponent)
    save_counter += 1

    # Maintain opponent pool size
    cnn_opponents = [opp for opp in opponents if not isinstance(opp, tuple)]
    if len(cnn_opponents) > max_cnn_opponents:
        # Remove oldest CNN
        for idx, opp in enumerate(opponents):
            if not isinstance(opp, tuple):
                removed = opponents.pop(idx)
                print(f"Removed oldest CNN opponent to maintain pool size.")
                break
```

Opponent Sampling and Learning Rate Adjustments

To improve training stability and accelerate performance specifically against MCTS1000, two targeted adjustments were made to the policy-gradient loop.

First, the opponent selection was modified to increase the model's exposure to MCST1000. Every 10 games, instead of uniformly sampling an opponent, we now select MCTS1000 with a 50% probability and random samples from the remaining opponents. This adjustment ensures the model trains directly against MCTS1000, the strongest opponent, more often. The thought behind this was to promote faster learning of high-quality strategies, as well as avoiding the overfitting to weaker opponents.

Second, manual learning-rate decay was introduced to stabilize gradient updates as training progresses. Every 50 games, the optimizer's learning rate is reduced by multiplying it by 0.96. The decay schedule helps with larger updates during early training, when the model is far from optimal, and smaller updates to fine tune behavior as time goes on. The learning rate adjustment is immediately applied after each increment of total_games, both in normal training and when skipping games with no collected moves.

Although the modification showed promising improvements, increased training against MCTS1000 led to heavier reliance on strong opponents and made training slower overall due to the computational load. Given more time, we hoped that the model would have achieved significantly better convergence against MCTS1000.

Double Deep Q Learning

The DQN model took a value approach. Instead of outputting move probabilities, it estimated the expected future reward associated with each possible action in a given state. Using an epsilon-greedy policy, the agent initially explored the action space widely, gradually shifting toward exploiting actions with the highest predicted rewards as training progressed. Stability was reinforced through the use of a target network, experience replay. To further address the known issue of overestimation bias in Q-learning, we implemented Double Deep Q-Learning. In DDQL, the main network selects the best action for the next state, but the target network evaluates the value of that action. This decoupling helps prevent the model from over-optimistically estimating reward values which could inflate the value of risky moves and reduce long-term stability.

For both the PG and DQN models, opponents were sampled from a curated pool that initially included a fixed, pretrained CNN (trained to mimic an 800-simulation MCTS policy) and early versions of self-trained networks. As training progressed, stronger agents were added to the pool after passing win-rate benchmarks (≥ 55 % vs. pool or ≥ 50 % vs. MCTS1000), fostering an adaptive and competitive environment.

Performance evaluation primarily focused on win rates against a standardized MCTS opponent (starting at 1000 simulations and increased by 200 simulations each time the DQN achieved ≥ 80 % win rate). This methodology enabled a controlled, progressive evaluation of reinforcement learning's potential to generate compelling AI opponents for single-player games.

For this project, we implemented a Double Deep Q-Learning (DDQN) agent designed specifically to play Connect-4. Our DQN agent makes use of a pretrained convolutional neural network originally developed to imitate Monte Carlo Tree Search from an earlier assignment.

Environment (Connect4Env)

Our Connect4 environment maintains the board as a 6×7 numpy array, where the agent's moves are represented by +1, the opponent's moves by −1, and empty cells by 0. To feed the neural network, this board is transformed into a 6×7×2 one-hot encoded tensor using a function called convert_to_3d. When the environment is reset, it randomly simulates up to 15 legal moves to expose the agent to diverse mid-game scenarios from the outset.

Gameplay progresses in turns: first, the agent selects and applies a move using the step(action) function, after which the environment evaluates for win or draw conditions, returning the appropriate reward. Next, if an opponent model has been set, the environment queries it for predicted Q-values, masks illegal moves by assigning $-\infty$ to their values, and selects the highest-value legal move. Rewards are straightforward: +1 for an agent's win, −1 for an agent's loss, and 0 for draws or ongoing games. The environment also exposes a straightforward API allowing easy interaction with the game state.

```
class Connect4Env:
    def __init__(self, starting_player="plus", max_init_random_moves=15):
        self.rows = 6
        self.cols = 7
        self.starting_player = starting_player  # "plus" or "minus"
        self.init_random_moves = np.random.randint(max_init_random_moves)
        self.board = np.zeros((self.rows, self.cols), dtype=int)
        self.current_player = starting_player
        self.done = False
        self.winner = None
        self.opponent_model = None
        self.opponent_color = "minus"
```

Network Architecture

The neural network builds upon a pretrained convolutional backbone (cnn_model3.h5), originally trained to imitate an 800-step MCTS policy. After the convolutional and LeakyReLU layers, we flatten the feature maps and append a seven-unit Dense layer with a tanh activation, so that each output—one per column—yields a bounded Q-value. We maintain both an online network, which is updated every training step, and a cloned target network whose weights are synchronized every 300 episodes to stabilize learning. Rather than mean-squared error, we now optimize with the Huber loss—wrapped in a custom, @tf.function-decorated train_step_tf—to reduce sensitivity to outliers. Finally, we train using the Adam optimizer at a learning rate of 1×10^{-4}, ensuring smooth, robust updates across our 100,000-episode curriculum

```
optimizer = Adam(1e-4)
dqn_model.compile(optimizer=optimizer, loss=custom_huber_loss)
dqn_model.call = tf.function(dqn_model.call, experimental_relax_shapes=True)


@tf.function
def train_step_tf(x_, y_, model, max_grad_norm=1.0):
    with tf.GradientTape() as tape:
        predictions = model(x_, training=True)
        loss = custom_huber_loss(y_, predictions)
    gradients = tape.gradient(loss, model.trainable_weights)
    gradients, _ = tf.clip_by_global_norm(gradients, max_grad_norm)
    model.optimizer.apply_gradients(zip(gradients, model.trainable_weights))
    return loss
```

Double Q-Learning Algorithm

At each training iteration, we sample a mini-batch of transitions $(s_i, a_i, r_i, s_{i+1}, done_i)$ from the replay buffer and construct our temporal-difference targets via Double Q-Learning. First, the **online** network predicts Q-values for each next state and we pick the arg max action $a^* = argmax_a Q_{online}(s_{i+1}, a)$. We then ask the target network for its Q-value estimate at (s_{i+1}, a) . *If* $done_i$ *is true, the target* y_i *is simply the immediate reward* r_i*; otherwise we add the discounted future estimate:*

$$y = r_i \; if \; terminal$$

$$y = r_i + \gamma Q_{target}(s_{i+1}, a) \; (\gamma = 0.99) \; otherwise$$

We then fit the online network by minimizing the Huber loss between y_i and $Q_{online}(s_i, a_i)$, using the Adam optimizer with a learning rate of 1×10^{-4}. This combination of Double Q-Learning and Huber loss yields stable, robust updates even in the presence of outliers.

```
while not done:
    action = select_action(state, dqn_model, epsilon, env, train=True)
    next_state, reward, done = env.step(action, train=True)
    replay_buffer.add(state, action, reward, next_state, done)
    state = next_state
    total_reward += reward

    if len(replay_buffer) >= batch_size:
        states, actions, rewards, next_states, dones = replay_buffer.sample(
            batch_size
        )
        targets = np.zeros((batch_size, 7))
        q_next_target = target_model.predict_on_batch(next_states)
        q_next_online = dqn_model.predict_on_batch(next_states)
        for i in range(batch_size):
            a = actions[i]
            if rewards[i] == 0 and not dones[i]:
                best_next_action = np.argmax(q_next_online[i])
                targets[i][a] = delta * q_next_target[i][best_next_action]
            else:
                targets[i][a] = rewards[i]
```

Hyperparameters & Curriculum

We trained our DQN agent for 100 000 episodes using a replay buffer capped at 6 000 transitions and a batch size of 32. The discount factor was set to $\gamma = 0.99$, and we decayed ε exponentially from 1.0 down to a floor of 0.05 (decay factor 0.9991 per episode). To stabilize learning, we synchronized the target network every 300 episodes. We also incorporated an early‑stopping mechanism: if the average training loss failed to improve by at least 1×10^{-4} over 200 episodes, we halted updates, restored the best weights, and ran a formal evaluation.

Our opponent curriculum began with one fixed, pretrained CNN (the MCTS-800 mimic) and up to five DQN clones in a deque (maxlen = 6). Every 2,000 episodes—and whenever early stopping was triggered—we ran 500-game evaluations against both an MCTS opponent (starting at 1 000 simulations, +200 sims each time it was beaten at ≥ 80 %) and the current opponent pool. If the agent achieved ≥ 45 % win rate versus the pool, we cloned its policy into the deque.

Component	**Value**
Episodes	100,000
Batch Size	32
Replay Buffer Capacity	6,000
Discount Factor (γ)	0.99
Initial ε	1.0 → 0.05 (decay 0.9991 per ep.)
Target Sync Frequency	Every 300 episodes
Opponent Eval Frequency	Every 2,000 episodes
MCTS Initial Depth	1,000 simulations
Win Rate to add opp	≥45%
Win Rate to upgrade MCTS	≥80% (increase depth by 200 sims)
Opponent Pool Max Length	6 (1 fixed CNN + up to 5 clones)
Early-Stopping	patience=200 eps; min_delta= 1×10^{-4}

Training Loop & Evaluation

Each episode proceeds by randomly sampling an opponent from the pool, resetting the Connect4Env, and playing to termination under an ε-greedy policy. After every action, we store the transition (s, a, r, s', $done$) in the replay buffer. Once the buffer holds at least 32 samples, we:

1. Sample a mini-batch of transitions.
2. Compute Double Q-Learning targets by using the online network to select the best next action and the target network to estimate its value.
3. Run a custom @tf.function training step that applies the Huber loss and Adam optimizer (LR = 1×10^{-4}) to update the online model.
4. Record the episode's average loss for early-stopping checks.

We copy the online weights to the target network every 300 episodes. At each 2,000-episode checkpoint—and upon early stopping—we call evaluate_vs_mcts(...) and evaluate_vs_opponent_pool(...). Models meeting the win-rate thresholds are cloned into the opponent pool and saved with descriptive filenames. Finally, if training is interrupted or completed, we plot the loss curve over all episodes.

```
    # Store average loss for the episode
    if episode_loss:
        avg_episode_loss = np.mean(episode_loss)
        episode_losses.append(avg_episode_loss)
        pbar.set_postfix_str(
            f"epsilon: {epsilon:.3f}, loss: {avg_episode_loss:.4f}, patience: {wait}/{patience}"
        )
```

```
    # Add to opponent pool if performance is good enough
    if win_rate >= add_dqn_win_rate:
        with tf.keras.utils.custom_object_scope(custom_objects):
            new_opp = clone_model(dqn_model)
        new_opp.set_weights(dqn_model.get_weights())
        dqn_opponents.append(new_opp)
        tqdm.write("Model added to opponent pool due to early stopping")
        dqn_model.save(
            f"Project3/checkpoints1/dqn_episode{episode}_early_stopping_win{win_rate}.keras"
        )
```

```
    if episode % add_opponent_freq == 0 and episode > 0:
        win_rate_mcts = evaluate_vs_mcts(dqn_model, depth=mcts_depth, games=500)
        tqdm.write(
            f"Evaluation vs MCTS{mcts_depth}: {win_rate_mcts:.2%} win rate @ Episode {episode}"
        )
        win_rate, info = evaluate_vs_opponent_pool(
            model=dqn_model,
            permanent_opponents=permanent_opponents,
            dqn_opponents=dqn_opponents,
            games=500,
        )
```

Actor-Critic A2C Implementation

In addition to policy gradient improvements, we implement an Advantage Actor-Critic (A2C) model built on top of our pre-trained CNN. A2C combines the best of both policy gradients and deep q-learning. Policy gradients are good at picking the best action, whereas DQN is better at predicting the expected reward from each action. It therefore trainas two models simultaneously. The model is made up of a policy head, which outputs a softmax over the possible actions, here the 7 columns available to play, and a value head which predicts the expected value of the state. The policy loss is based on advantage, which measures how much better an action performed compared to what the value head predicted. Moves that perform better than expected are more heavily rewarded, encouraging exploration and preventing sub-optimal equilibria. The critic implements temporal difference learning to provide the estimated discounted reward of the move selected by the actor. The critic loss is the squared error between the predicted value and the Bellman target:

$$\mathcal{L}_{\text{critic}} = \left(r_t + \delta V(s_{t+1}) - V(s_t)\right)^2$$

The actor loss encourages selecting actions with a higher advantage. Advantage represents the difference between actual and predicted losses; the model is rewarded for performing better than expected. The critic loss function encourages accurate value estimation. The total loss is a weighted sum of actor and critic losses, discounting the critic loss compared to the actor loss.

```
@tf.function
def train_step_a2c(states, actions, rewards, next_states, dones, model, gamma=0.99):
    with tf.GradientTape() as tape:
        action_probs, state_values = model(states, training=True)
        _, next_state_values = model(next_states, training=True)

        targets = rewards + gamma * tf.squeeze(next_state_values) * (1 - dones)
        advantages = targets - tf.squeeze(state_values)

        action_masks = tf.one_hot(actions, 7)
        log_probs = tf.math.log(tf.reduce_sum(action_probs * action_masks, axis=1) + 1e-8)
        actor_loss = -tf.reduce_mean(log_probs * advantages)
        critic_loss = tf.reduce_mean(tf.square(targets - tf.squeeze(state_values)))
        total_loss = actor_loss + 0.5 * critic_loss

    grads = tape.gradient(total_loss, model.trainable_variables)
    model.optimizer.apply_gradients(zip(grads, model.trainable_variables))
    return actor_loss, critic_loss
```

Training used experience replay (buffer size: 1500, batch size: 32), sampling random batches of gameplay against a pool of opponents consisting of our previous models. Every 200 episodes, if the A2C model's win rate exceeded 30% against the opponent pool, it was added to the pool to increase difficulty, similarly to what we had done previously with our dynamic opponent management. The agents were selected by sampling from the softmax output, while still masking illegal moves.

```python
def select_action_a2c(state, model, env: Connect4Env, train=False):
    action_probs, _ = model.predict(state[np.newaxis, ...], verbose=0)
    if train:
        actions = env.legal_actions()
    else:
        actions = env.optimal_actions()
    masked_probs = np.zeros_like(action_probs[0])
    masked_probs[actions] = action_probs[0][actions]
    total_prob = masked_probs.sum()

    if total_prob == 0.0:
    # If total prob is zero, fallback: uniform random legal move
      masked_probs = np.ones_like(masked_probs)
      masked_probs[actions == 0] = 0  # Still enforce legality: only legal actions allowed
      masked_probs /= masked_probs.sum()
    else:
      masked_probs /= total_prob

    action = np.random.choice(len(masked_probs), p=masked_probs)
    return int(action)
```

We trained our agent by playing against an MCTS1000 opponent and saved checkpoints of the model every 1,000 episodes. Training continued for the full 100,000 episodes, allowing the policy sufficient time to converge and providing us with a comprehensive range of snapshots to select the strongest-performing model.

The A2C training allowed the model to learn better action selection policies and value estimation, resulting in more stable and faster convergence.

Performance

The model's starting performance is terrible, which may be surprising given that the baseline model wins against MCTS1000 71% of the time. It's important to remember that we use new output layers for both the actor and the critic. The underlying structure, the brain, of our model might have a good understanding of how to play and win Connect4 games, but it's not good at communicating that. The weights of these output layers are randomly initialized and lead to a model that is easily beaten by MCTS1000 when training starts. Over time the

Tracking the loss of both the actor and the critic reveals that while we are training them simultaneously, they are learning at different rates. In general, the actor learns more quickly than the critic. This can also be attributed to the loss function that weights actor loss more heavily than critic loss.

Potential Overfitting

In our early implementation, we observed that the model achieved a 100% win rate against all opponents in the training pool. However, it was unable to win many games against MCTS1000. This indicated overfitting rather than true generalization and a high, accurate win rate. This iteration of the model had a low threshold for adding new opponents to the pool and did not preserve any of the original convolutional neural nets. This resulted in a race to the bottom, where the model learned to play against bad opponents and as a result added new, equally bad (or worse) opponents. We addressed this overfitting by incorporating a gradually increasing threshold for adding opponents, as well as a check that removed opponents when the model was beating them more over 90% of the time. The original models were also preserved to ensure that the "ground truth" remained consistent. The model trained against MCTS1000

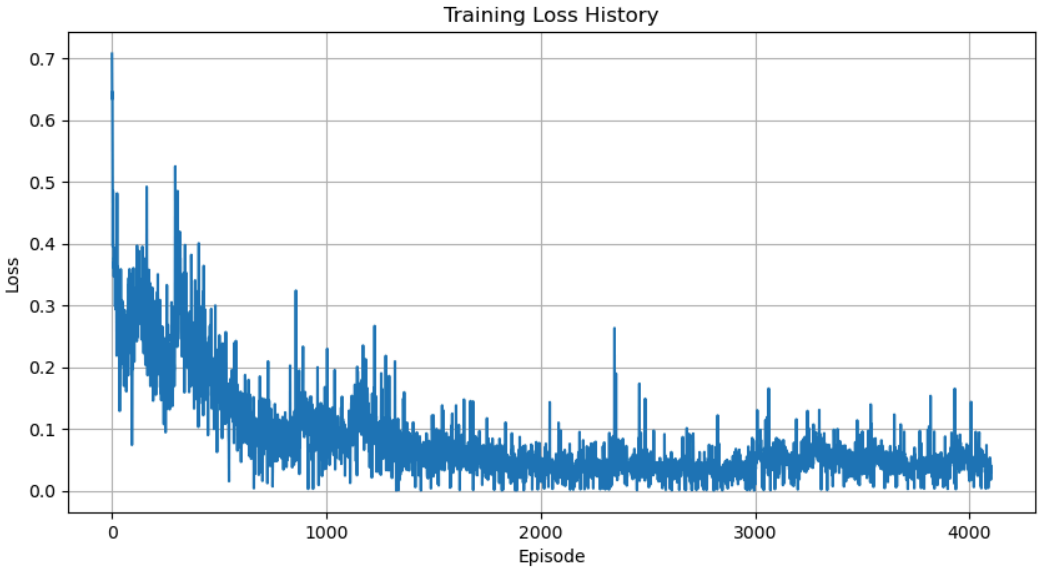
every 50 games and evaluated its performance against MCTS1000 over 50 games every 500 games (10% of the time).

The model may have memorized previous patterns from models it had been exposed to, rather than learning appropriate Connect 4 strategies. We then moved to prioritize evaluating performance against MCTS1000 to better assess generalization.

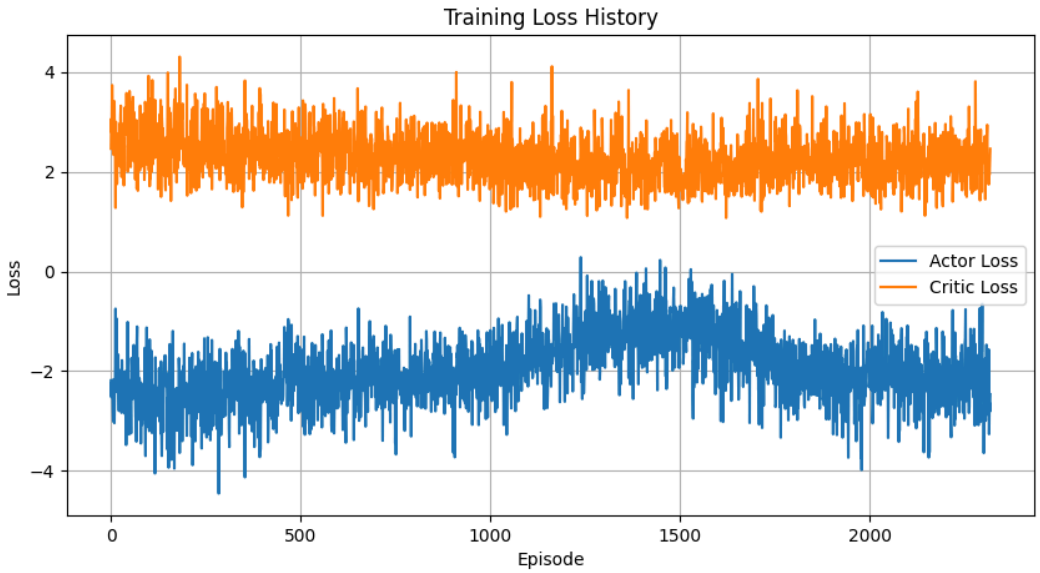
Results

Policy gradients help improve model performance from 71% against MCTS1000 to 79%. These improvements come from training against varying levels of MCTS opponents, other convolutional neural networks, and the "upgraded" model as we introduced self-play into training. This improvement was highly variable; there were iterations of our training where policy gradients seemingly got stuck in a low-performance equilibrium and gradually decreased in performance quality. The random process of winning games when both the model and opponent are at a low level of performance means that there is no deterministic path to adding new opponents. This can drastically alter the trajectory of model training. Also surprisingly, our best model performed better against MCTS1000 than MCTS500. This may be because we evaluate model performance on MCTS1000, so the model is skewing towards that style of play and can occasionally be beaten by MCTS500 when it prioritizes short term rewards and quick winning over drawn out play.

Double DQNs surprisingly did not yield any model significantly better than the base model. The model performance against MCTS1000 started off increasing until around 2000 games, where the win rate was 72%, then decreased quickly to 30% at 5000 episodes when we decided to kill the process. This was likely because bad models were added to the opponent pool, making the model think it made some good moves although they were bad. However, as we can see the loss function for double DQN below, the model converges to a steady loss function after approximately 1500 games (not including the intermittent evaluations). Continued improvements come from adding more opponents for the model to play against; this might not improve the overall loss function, but it can continue to make the model more robust to different play styles. This is one of the difficulties of exclusively tracking the loss function: it does not tell the entire picture of model performance.



A2C promises big improvements over our original model, but at implementation we found that we were not able to find the correct combination of opponent models and training schedule to yield performance improvements over the original M1. We believe that longer training and an expanded pool of opponents could make this a viable reinforcement learning technique in the future, but for now we believe we should move forward with either our policy gradients or DDQN approaches. Below we show the loss for the actor and critic throughout the model's training. Critic loss is represented as a negative number to separate more clearly from actor loss and show both trends on a single graph. This visual highlights the weighted loss function we mentioned previously, as well as how losses are unstable throughout training but are not trending towards zero in the way we could expect or hope. This is likely a large part of the reason why A2C performs so poorly against MCTS. While this strategy does not currently produce a viable single player *Connect 4* opponent, we will continue to train this model for longer and believe once it converges we will be able to take advantage of the best elements of policy gradients and DQN. We also experimented with different learning rates and found that the magnitude of both actor and critic loss could swing wildly when learning rate was not well calibrated. Too high and the model would never converge to a single set of best action for a given state, too low and the model took an unreasonably long time to train.



Recommendation

Our project results show that Reinforcement Learning has real potential for building smarter and more realistic AI opponents in single-player video games. Reinforcement Learning can be technically challenging to build and fine tune, so we believe it would be very beneficial to hire an RL expert. We have already seen the tangible benefits that leveraging RL can bring, so having someone who understands how to shape rewards, manage training environments, and drive the learning process would make it much easier to create AI video game opponents that are challenging and realistic.

As such, RL is a strong foundation for improving single player gaming experiences. Choosing the correct method for the type of game and bringing the right expertise onto the team would make a big difference in creating more intelligent AI models.

Conclusion and Future Scope

This project showed that Reinforcement Learning, particularly Policy Gradients and Double Deep Q Learning, can be successfully applied to train AI models to play Connect 4. The Policy Gradient model improved by directly optimizing move probabilities based on the discounted sum of all future rewards. The Double Deep Q learning model learned to estimate the expected reward of actions and gradually shifted toward selecting actions that maximized that value. Both approaches benefited from training against a dynamic opponent pool, which helped the models continue to adapt as they improved.

While both methods made good progress, we encountered challenges such as overfitting to weaker opponents and unstable training runs. Our results showed that Reinforcement Learning could create a competitive game playing model, but it was important that we carefully evaluated our training logic, evaluation methods, and reward metrics in order to have success training the model. A2C suffers from high variance, and in the future we may introduce small, non-zero rewards for non-losing steps as a way to speed up progress. Getting three in a row or blocking an opponent are ways to introduce more frequent

rewards and increase the rate of model improvement. These would need to be removed once the model reaches sufficient performance, as at a certain point they will start to hinder long term learning.

Further, if we were to run this project again, a few things stand out as really important. First, it is critical to have a strong, stable opponent like MCTS1000 to measure progress. Without that benchmark, it is easy to get filled by the model winning against weaker opponents. Second, we found that skipping training updates after bad games helped a lot. By only updating when games ended in wins or ties, the model avoided reinforcing poor strategies. Also, we learned that keeping the opponent pool under control made a big difference. Adding every new model without checking if it was actually strong enough just made the training worse. Careful promotion and removal kept the training challenging and more rigorous.

Finally, training in an offline way helped the models assign credit properly to different moves. Especially in a game like Connect 4 where previous moves are really important but their effects are not immediate, this made a big difference.

There are a couple of things we could do beyond what was implemented in this project. One direction we could take would be to build on the early success we had with the Advantage Actor Critic Models, which showed signs of faster and more stable learning. Another idea would be to add stronger MCTS opponents, like MCTS1500 or MCTS2000, to push the model harder.