Deliverable 2 - Connect 4 Model:

1) Goal: Agent that can play Connect 4 optimally using RL. We will build the agent to self-learn how to play efficiently through game experiences and feedback on each game action taken by the model.

Quick Note: Games like Connect 4 can be solved in simpler ways (such as a minimax algorithm) but we will attempt a RL learning approach for the theme of the course.

---

2) Date-Preprocessing:

It is difficult and expensive to collect pre-existent data on Connect 4 games (around 4.5 trillion possible game states) to train the model. Instead, the model will interact with the environment (which we will build) and perform actions in it. It will create its own game samples that way to learn from during training.

What would our model need to play the game and train?

- Game states as input: board with the occupied tokens of each color
- Legal moves: it should only look at the legal moves it can take at the input state
- Rewards: it should receive feedback on how each legal move influences its position
- Current score (possibly): to estimate the position of the model at the current state

Game Setup:

- State: 6x7 board
- Actions: 7 (or less) possible columns to play
- Reward: model should be rewarded for winning (+1) and punished for losing (-1). No progress for draws (0). We could also reward the model for being close to winning/forcing the opponent to play a defensive move ($0 < x < 1$). For any other actions (0).

State Representation:

- 6x7 matrix or array of size 6*7 = 42 where we represent each player's tokens by a unique number (player 1's token = 1, player 2's token = -1).
- We might attempt an M = 2x6x7 matrix where M[0] represents current player's tokens with a 1 (and 0 elsewhere), and M[1] represents the opponents' tokens in the same way (for normalization reasons which make NNs more stable).

Action Limitations: The model will not need to be penalised for making illegal moves as we simply only allow it to make the legal ones.

---

3) Reinforcement Learning algorithm:

We will try this with a Deep Q-Network as it is good at learning general strategy instead of perfect play (with brute-force search models).

Framework & Tools:

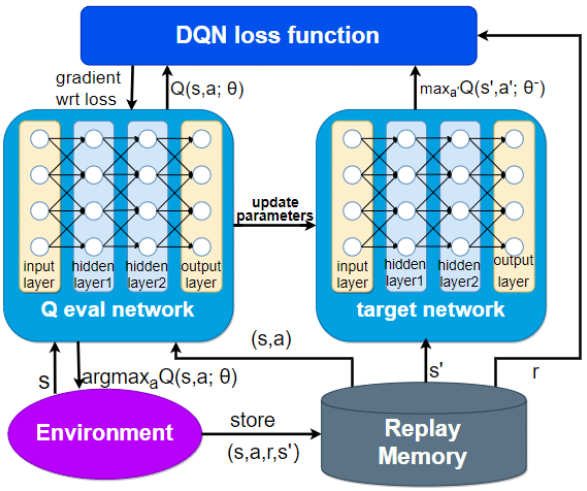
Libraries: Pytorch for constructing the NN (standard and efficient library). Numpy for state representations and game logic (efficient).

Model Architecture: We will first try a model with 3 hidden layers, with decreasing width.

- Input layer: vector of size 84 (for the 2x42 matrix)
- Hidden layer #1: 256 neurons
- Hidden layer #2: 128 neurons
- Hidden layer #3: 64 neurons
- Output layer: vector of size 7 (for each possible column play)

The activation function will be ReLU.
We will use a replay buffer with a max capacity of 10,000.

Example image of Deep Q-Network architectures:



Training:

- The agent will play against itself. The target network will be updated every 10 iterations/episodes.
- Learning rate: default Adam optimizer learning rate of 0.001.
- Epsilon: starting at 1 for exploration then decreasing by a factor of 0.995 at each iteration. Goes to a minimum 0f 0.1.
- Gamma: 0.99 to encourage winning as early as possible. Might lower it even more since Connect 4 can be a relatively short game.
- Reward System: +1 for a win, -1 for a loss, 0 for a draw. Could add rewards for strategies/threats and close to winning positions in the future.

Validation:

- The agent performance will be tested on its win rate on a certain number of games (1000 for now).

Self-Critics/questions: We should also test model performance in smaller iterations at the start to understand model learning better? Making the model play against itself might not be the right approach as it might appear to be performing well only against itself?

Problems during implementation: A lot of type related errors from the tensors. Need better Validation metrics to fully ensure that the model can play well against any opponent (like game length, forcing it to play first/second and comparing results,...).

---

4) Preliminary Results:

Using win rate evaluation metric:

- Episode (0-500): +99% win rate
- Episode (500-1000): +97% win rate
- …
- Episode 20000: ~87% win rate (converges to this each time)

The model seems to forget important strategies (might need a larger replay buffer) or the evaluation metric is misleading considering the model is playing itself (starting win rates seem almost “too good to be true”).

Note: We changed the training and testing length since the previous ones were not revealing enough information and appeared misleading.

---

5) Next Steps:

- Increase replay buffer size.
- Change reward function. Reward model for close to winning strategies (3 or 2 in a row, for playing in the center). And penalise it for allowing close losses.
- Train the model not only against itself but against different opponents (could find online models). Training it against a minimax model is probably better to force the model to learn optimal play.