~~ConnectX Kaggle competition~~ Reinforcement Learning on a two-player game

Max Serra Lasierra

Project for the course “Machine Learning 2” WS22/23

University of Potsdam

Problem setting – Connect4

Problem setting – TicTacToe

My approach to the project

- Align code to **reference libraries**:
  - Gym for environments
  - Stable Baselines3 for algorithms
  - Weights and Biases for tracking and evaluating
- Select algorithms to use
- Define evaluation of agents

- Code available on github: https://github.com/maxserra/ml2_project_connectx

My approach to the project – Evaluation

- How to evaluate an agent?
  - Play against another agent
  - Average cumulative reward
    - Win rate is a special case with the reward set to 0 – 1
- How do we choose the reward function?
  - Do we program game rules on the agent?
  - Or, do we let the agent learn them?

TicTacToe environment

kaggle Environments

- Based on Kaggle environment, adapted to Gym

environments > tictactoe.py > TicTacToe > __init__

```
22        # Define required gym fields (examples):
23        self.action_space = spaces.Discrete(3 * 3)
24        self.observation_space = spaces.Dict({'remainingOverageTime': spaces.Discrete(60 + 1),
25                                              'step': spaces.Discrete(3 * 3 + 1),
26                                              'board': spaces.Box(low=0, high=2, shape=(1, 3 * 3), dtype=int),
27                                              'mark': spaces.Discrete(2 + 1)})
28        self.reward_range = (-10, 1)
```

environments > tictactoe.py > TicTacToe > step

```
30    def step(self, action):
31        # Make sure the env is reseted
32        if self.kaggle_env.done:
33            self.reset()
34        # Check validity of action
35        is_valid = (self.obs['board'][int(action)] == 0) and self.action_spac
36        if is_valid:  # Play the move
37            self.obs, old_reward, done, info = self.trainer.step(int(action))
38            reward = self._custom_reward(old_reward, done)
39        else:  # End the game and penalize agent
40            reward, done, info = -10, True, {}
41        return self.obs, reward, done, info
```

environments > tictactoe.py > TicTacToe > _custom_reward

```
61    @staticmethod
62    def _custom_reward(old_reward, done):
63        if old_reward == 1:  # The agent won the game
64            return 1
65        elif done:  # The opponent won the game
66            return -1
67        else:  # Reward 1/9
68            return 1 / (3 * 3)
```

My approach to the project – Algortihms

- Which algorithms could we use?
  - Temporal-Difference:
    - Q-Learning
    - (Expected) Sarsa
  - Monte Carlo Tree Search

My approach to the project – Algortihms

- Which algorithms could we use?
  - Temporal-Difference:
    - Q-Learning
    - (Expected) Sarsa
  - Monte Carlo Tree Search
  - Deep Reinforcement Learning (“out-of-the-box” from Stable Baselines3)
    - DQN
    - PPO

Q-Learning

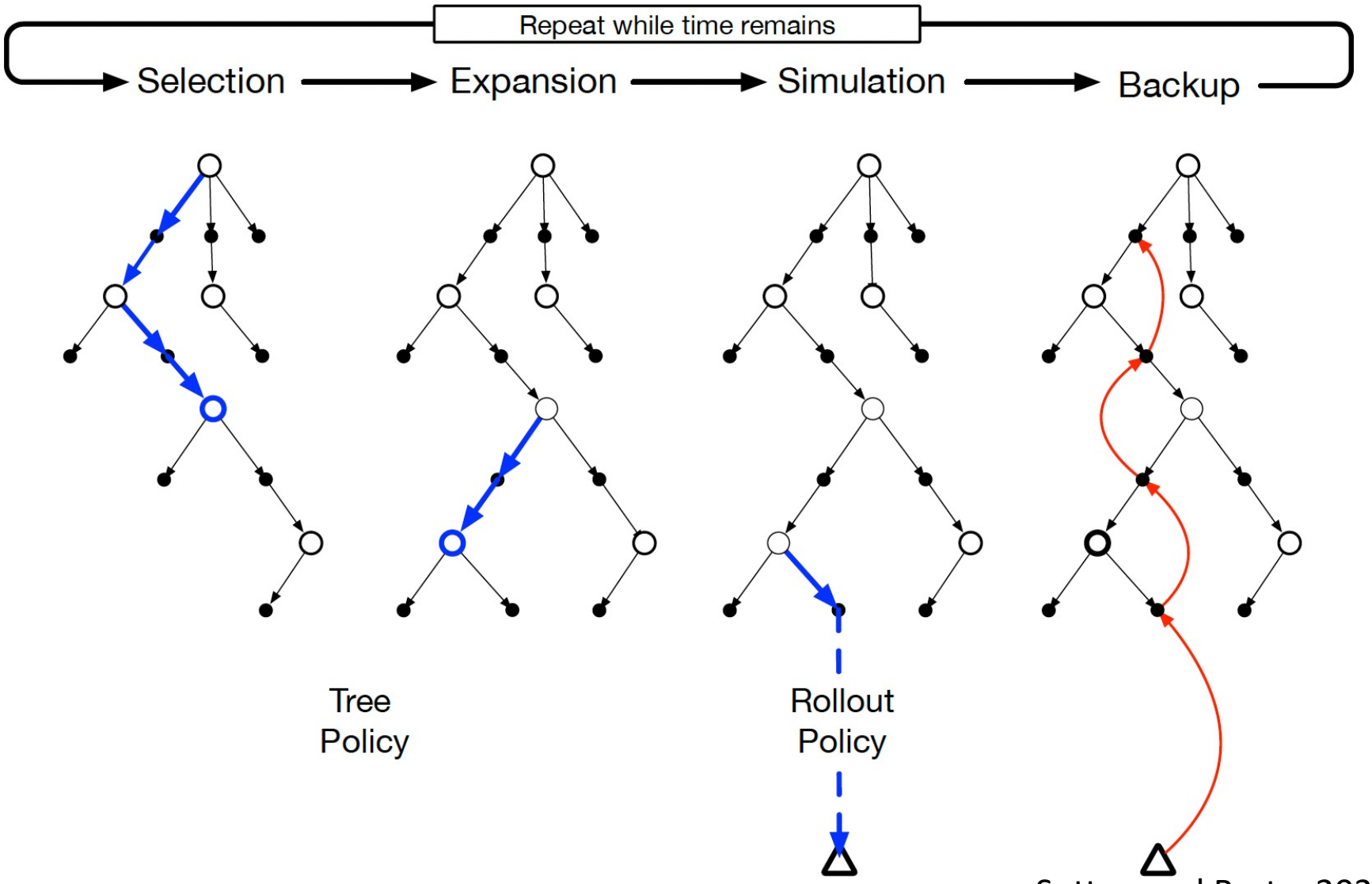
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize S
    Loop for each step of episode:
        Choose A from S using policy derived from Q (e.g., ε-greedy)
        Take action A, observe R, S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
        $S \leftarrow S'$
    until S is terminal

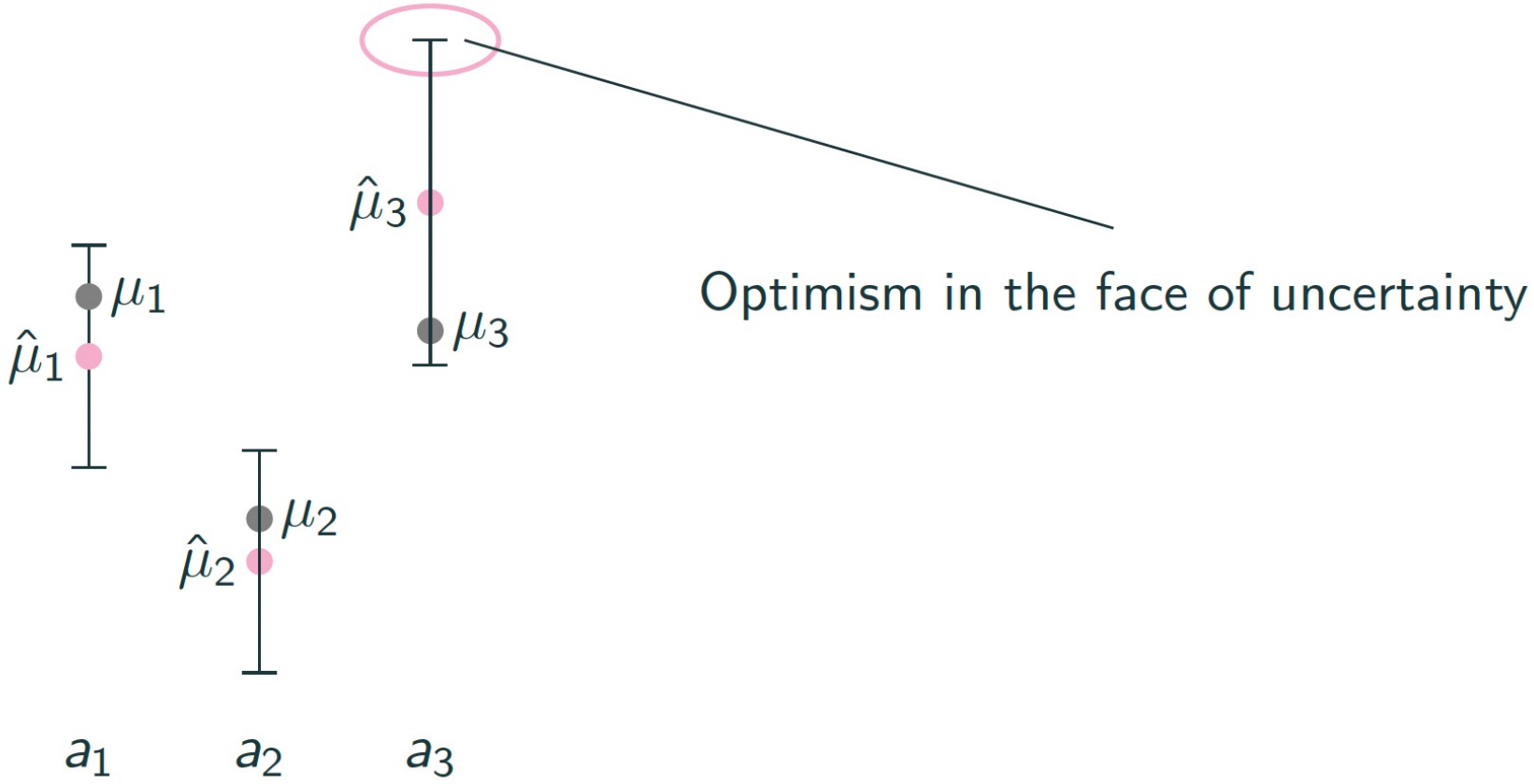
Sutton and Barto, 2020

MCTS



Sutton and Barto, 2020

MCTS – Upper Confidence bound for Trees (UCT)



MCTS – Upper Confidence bound for Trees (UCT)

rl_agents > monte_carlo_tree_search > policies.py > MCTSPolicy > _compute_uct_values

```python
    @staticmethod
    def _compute_uct_values(
        action_values: Dict[Any, float],
        selection_count: Dict[Any, int],
        exploration_cte: float = 2
    ) -> Dict[Any, float]:

        if sorted(action_values) != sorted(selection_count):
            raise RuntimeError("Unexpected difference in entries for 'action_values' and 'selections_count'." +
                               f"'{action_values.keys()}' and '{selection_count.keys()}'")

        actions_sorted = sorted(action_values.keys())
        values_sorted = np.array(list(map(action_values.get, actions_sorted)))
        count_sorted = np.array(list(map(selection_count.get, actions_sorted)))

        uct_values = values_sorted + exploration_cte * np.sqrt(count_sorted.sum() / (1 + count_sorted))

        return {action: value for action, value in zip(actions_sorted, uct_values)}

```

MCTS – Adaptive UCT

rl_agents > monte_carlo_tree_search > policies.py > MCTSPolicy > _compute_adaptive_uct_values

```python
    @staticmethod
    def _compute_adaptive_uct_values(
        action_values: Dict[Any, float],
        selection_count: Dict[Any, int]
    ) -> Dict[Any, float]:

        if sorted(action_values) != sorted(selection_count):
            raise RuntimeError("Unexpected difference in entries for 'action_values' and 'selections_count'." +
                               f"'{action_values.keys()}' and '{selection_count.keys()}'")

        actions_sorted = sorted(action_values.keys())
        values_sorted = np.array(list(map(action_values.get, actions_sorted)))
        count_sorted = np.array(list(map(selection_count.get, actions_sorted)))

        exploration_cte = scipy.special.softmax(values_sorted)

        uct_values = values_sorted + exploration_cte * np.sqrt(count_sorted.sum() / (1 + count_sorted))

        return {action: value for action, value in zip(actions_sorted, uct_values)}
```

The algorithms I coded

- Temporal Difference:
  - Q-Learning
  - Sarsa and expected Sarsa
- Monte Carlo Tree Search

arena.py > main_MCTS

```
34      mcts_agent = MCTS(policy=MCTSPolicy,
35                        env=env,
36                        max_episode_steps=10,
37                        verbose=1,
38                        tensorboard_log=f"logs/{config['algorithm']}_tictactoe"
39                        )
40
41      mcts_agent.learn(total_timesteps=config["total_timesteps"],
42                       log_interval=config["log_interval"],
43                       selfplay_start=config["selfplay_start"],
44                       reset_num_timesteps=False,
45                       callback=WandbCallback(log="all",
46                                              model_save_path=f"pretrained_agents/{run.name}",
47                                              )
48                       )
```

Weights & Biases

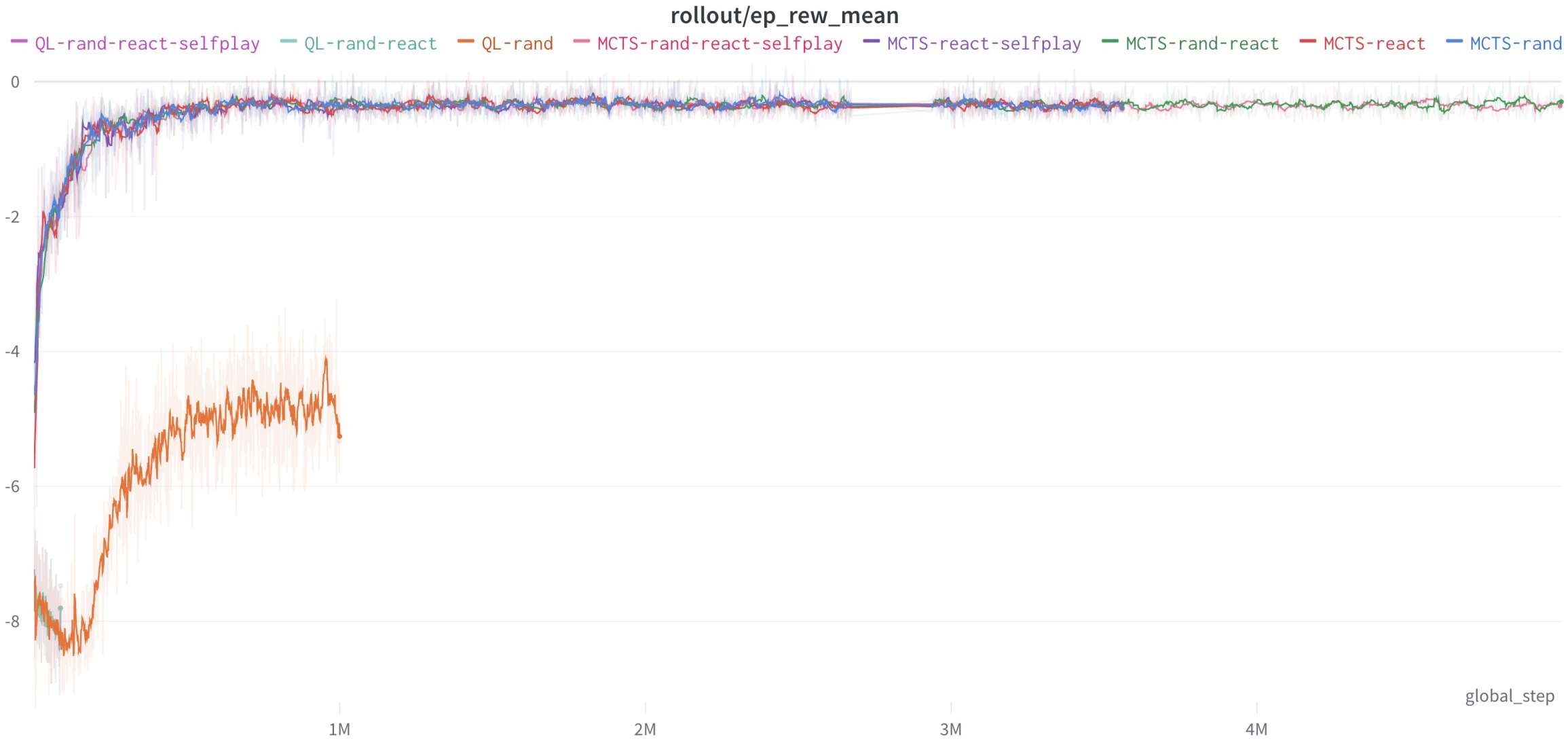
The algorithms I coded

- The agent should learn to play both X and O!

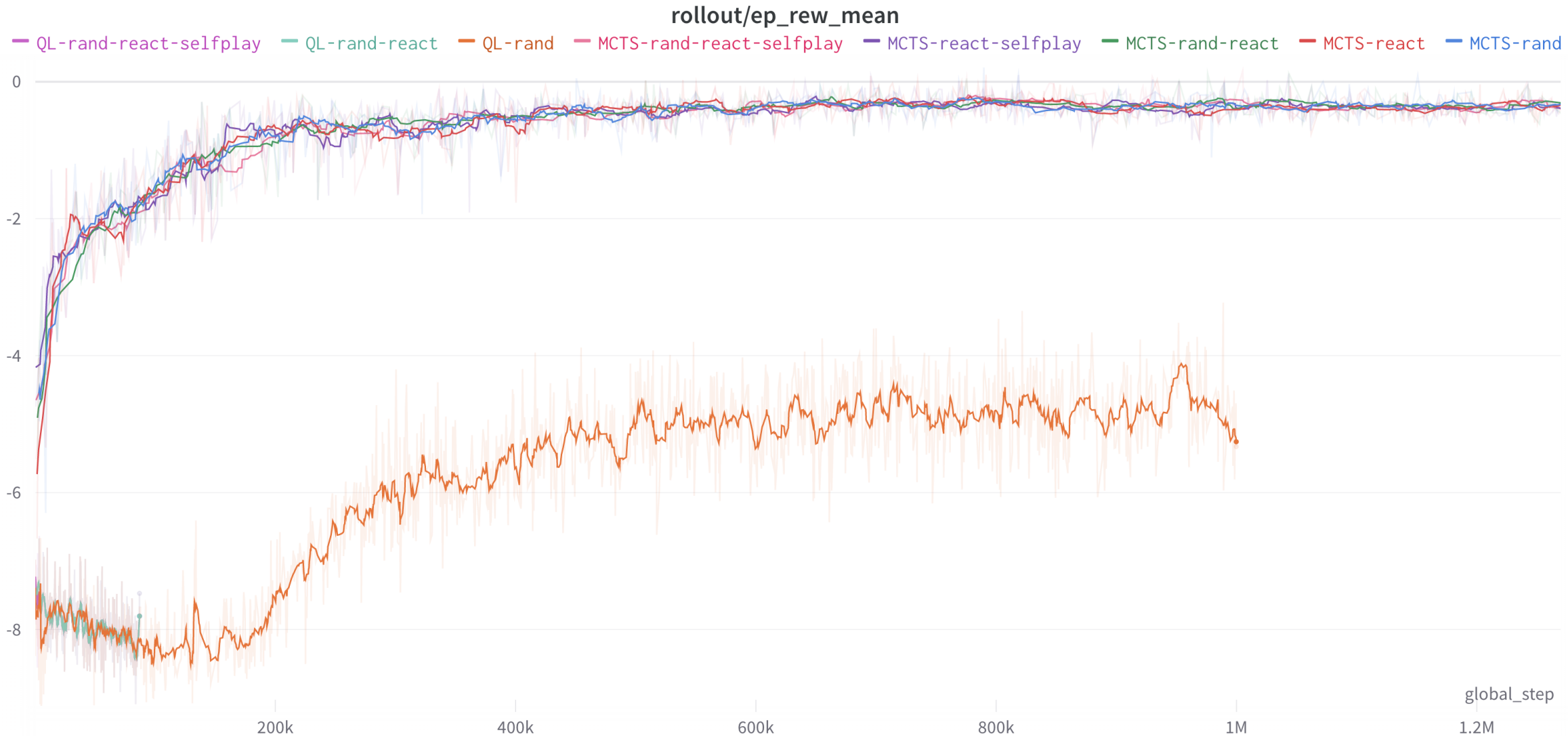
environments > tictactoe.py > TicTacToe > _make_new_pair

```
112     def _make_new_pair(self):
113         return [None, np.random.choice(self.opponents)]
114
115     def _switch_trainer(self):
116         # Change opponent
117         if np.random.random() < self.switching_prob:
118             self.pair = self._make_new_pair()
119         # Reverse roles
120         if np.random.random() < self.switching_prob:
121             self.pair = self.pair[::-1]  # reverse list order
122         self.trainer = self.kaggle_env.train(self.pair)
123
```

The algorithms I coded – wandb project



The algorithms I coded – wandb project



The algorithms I **did not** code

- Deep Reinforcement Learning ("out-of-the-box" from Stable Baselines3)
  - DQN (Deep Q Network)
  - PPO (Proximal Policy Optimization)

- The NN is (obviously) exposed and can be modified.

The algorithms I **did not** code – Kaggle notebook

```python
class PlainFullyConnected(BaseFeaturesExtractor):

    def __init__(self,
                 observation_space: gym.spaces.Dict,
                 features_dim: int=128
                ):
        super().__init__(observation_space, features_dim)
        self.input_shape = (-1, 9)

        n_input = th.as_tensor(observation_space["board"].sample()).reshape(self.input_shape).shape[1]

        self.fully_connected = nn.Sequential(
            nn.Linear(n_input, 32),
            nn.ReLU(),
            nn.Linear(32, 32),
            nn.ReLU(),
            nn.Linear(32, features_dim),
            nn.ReLU(),
        )

        # test the forward pass
        with th.no_grad():
            forward_test = self.forward(observation_space.sample())
            print(f"Shape of the forward output:  {forward_test.shape}")
            print(f"Sample of the forward output: \n{forward_test}")

    def forward(self, observations) -> th.Tensor:
        observations = th.as_tensor(observations["board"]).reshape(self.input_shape).float()
        return self.fully_connected(observations)
```

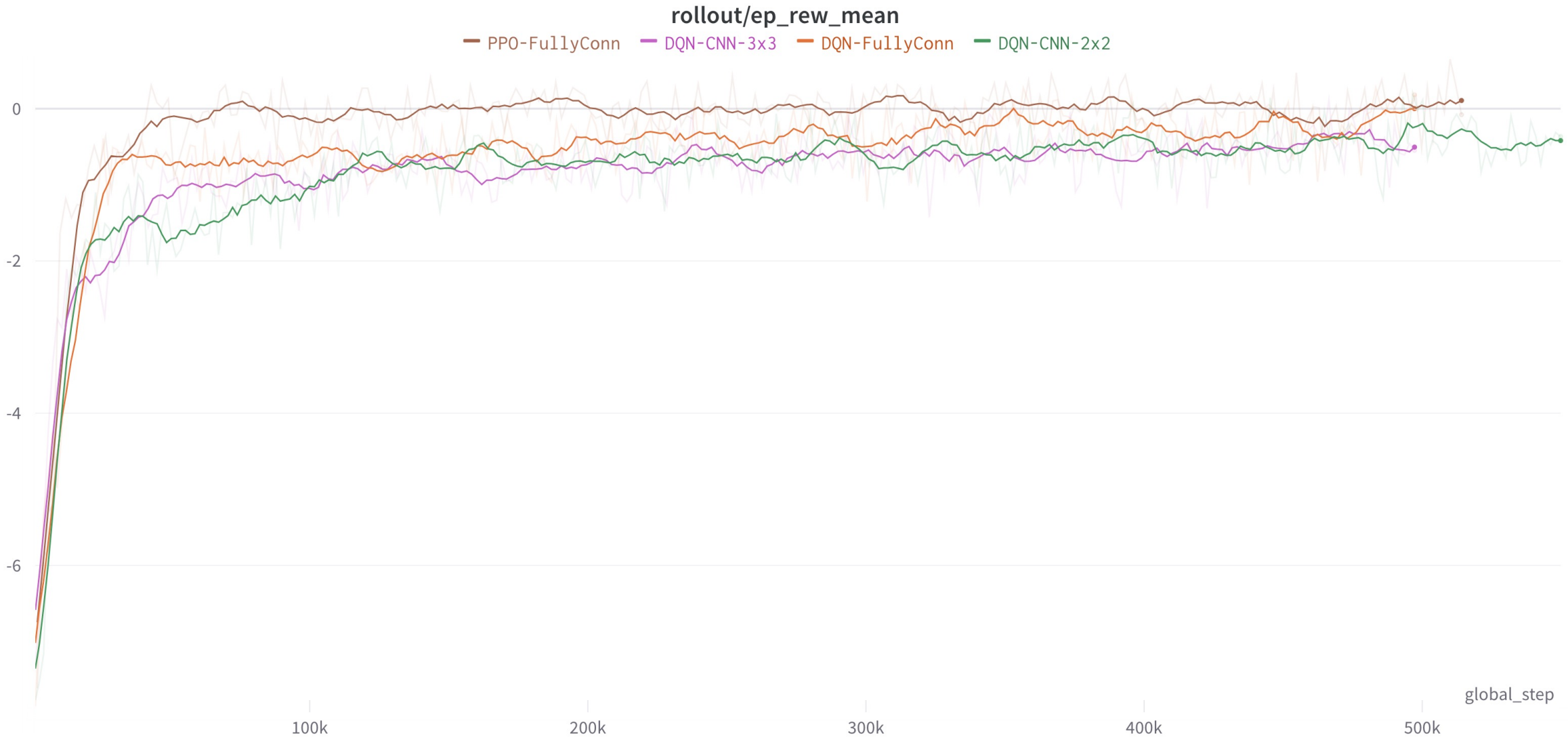
The algorithms I **did not** code – Kaggle notebook

```python
class TicTacToeCNN(BaseFeaturesExtractor):

    def __init__(self,
                 observation_space: gym.spaces.Dict,
                 cnn_channels: List[int] = [6, 12],
                 features_dim: int=128,
                ):
        super().__init__(observation_space, features_dim)
        self.input_shape = (-1, 1, 3, 3)

        if len(cnn_channels) > 2:
            raise ValueError(f"Passed cnn_channels '{cnn_channels}' are too long. " +
                             "Given kernel size of 2, you can only pass a list of max length 2.")

        self.cnn = nn.Sequential(nn.Conv2d(in_channels=1, out_channels=cnn_channels[0], kernel_size=2, stride=1),
                                 nn.ReLU(),)

        for i in range(1, len(cnn_channels)):
            self.cnn.append(nn.Sequential(nn.Conv2d(in_channels=cnn_channels[i-1],
                                                    out_channels=cnn_channels[i],
                                                    kernel_size=2, stride=1),
                                          nn.ReLU(),))

        self.cnn.append(nn.Sequential(nn.Flatten()))

        # Compute shape by doing one forward pass
        with th.no_grad():
            n_flatten = self.cnn(
                th.as_tensor(observation_space["board"].sample()).reshape(self.input_shape).float()
            ).shape[1]

        self.cnn.append(nn.Sequential(nn.Linear(n_flatten, features_dim),
                                      nn.ReLU(),))

        # test the forward pass
```

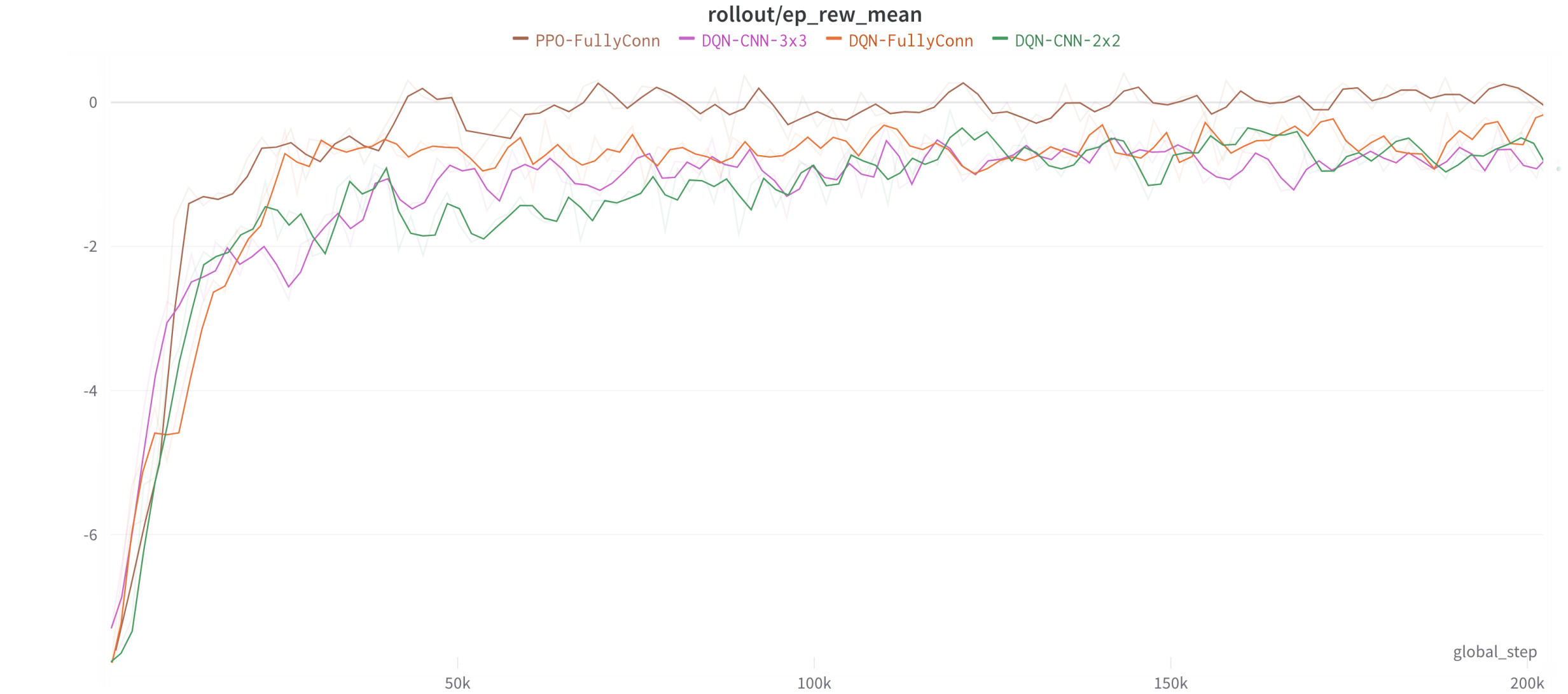
The algorithms I **did not** code – Kaggle notebook

```python
class TicTacToeCNN_3x3(BaseFeaturesExtractor):

    def __init__(self,
                 observation_space: gym.spaces.Dict,
                 cnn_channels: List[int] = [12],
                 features_dim: int=128,
                ):
        super().__init__(observation_space, features_dim)
        self.input_shape = (-1, 1, 3, 3)

        if len(cnn_channels) > 1:
            raise ValueError(f"Passed cnn_channels '{cnn_channels}' are too long. " +
                             "Given kernel size of 3, you can only pass a list of max length 1.")

        self.cnn = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=cnn_channels[0], kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Flatten(),
        )

        # Compute shape by doing one forward pass
        with th.no_grad():
            n_flatten = self.cnn(
                th.as_tensor(observation_space["board"].sample()).reshape(self.input_shape).float()
            ).shape[1]

        self.cnn.append(
            nn.Sequential(
                nn.Linear(n_flatten, features_dim),
                nn.ReLU(),
            )
        )

        # test the forward pass
        with th.no_grad():
```

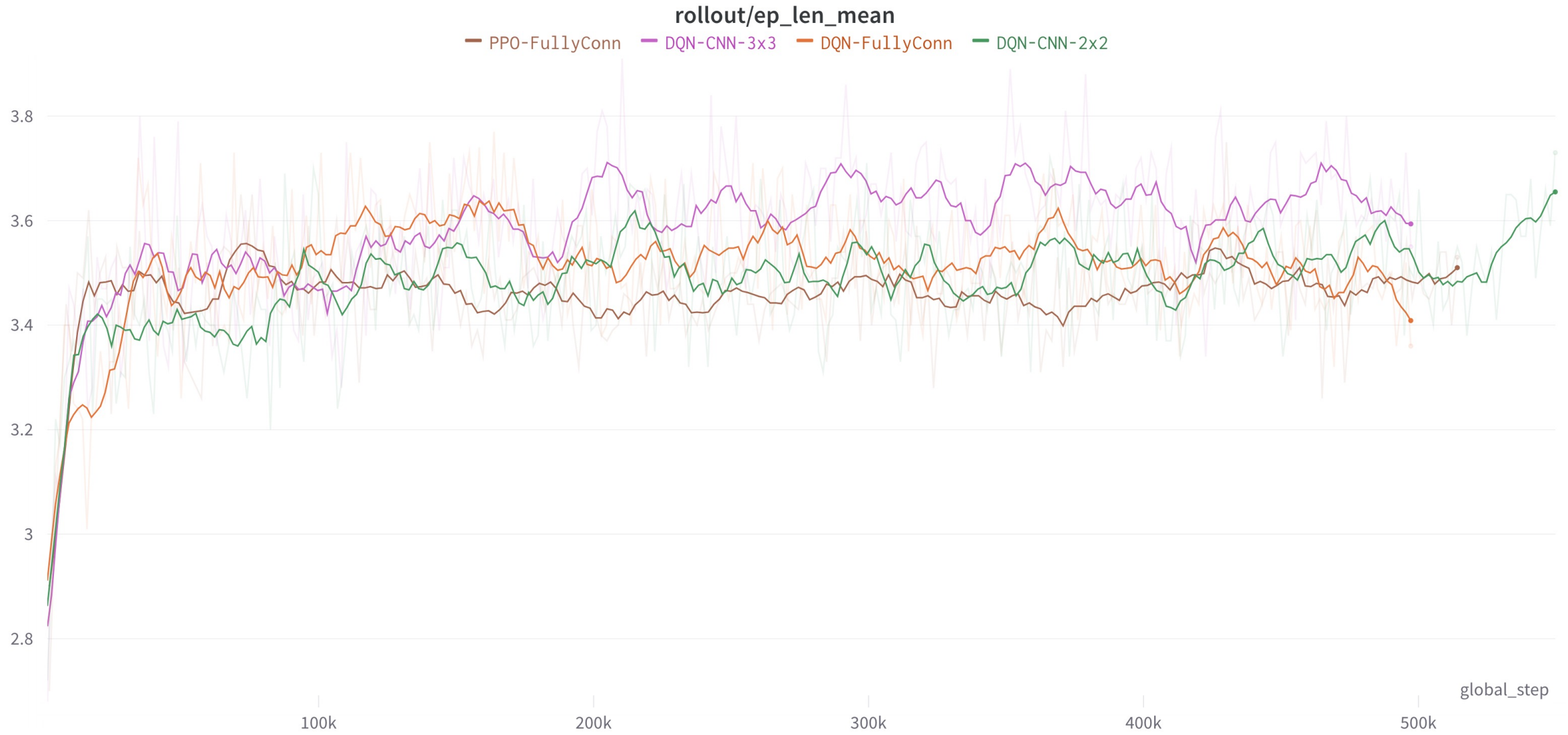
The algorithms I **did not** code – wandb project



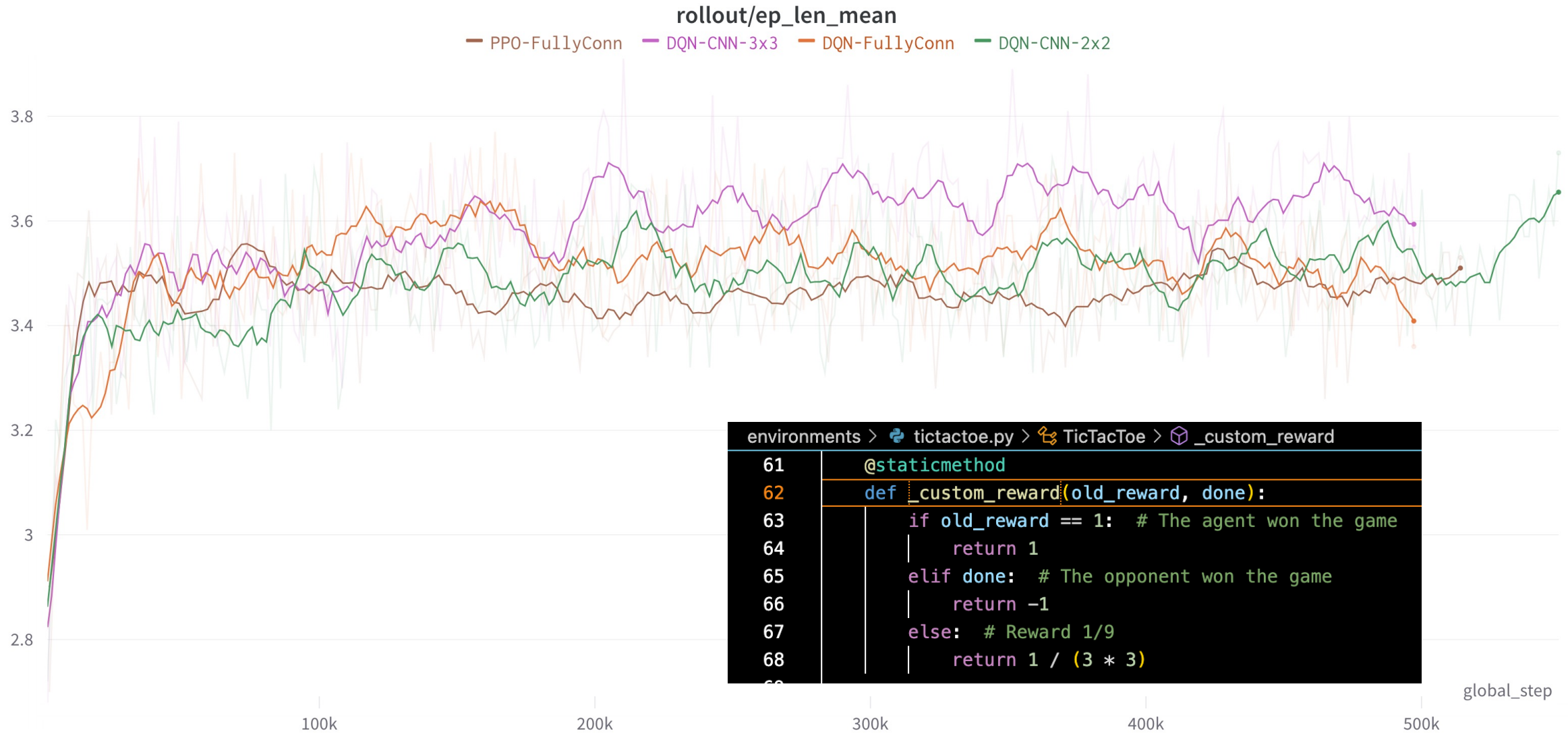
The algorithms I **did not** code – wandb project



The algorithms I **did not** code – wandb project



The algorithms I **did not** code – wandb project



environments > tictactoe.py > TicTacToe > _custom_reward

```
61      @staticmethod
62      def _custom_reward(old_reward, done):
63          if old_reward == 1:  # The agent won the game
64              return 1
65          elif done:  # The opponent won the game
66              return -1
67          else:  # Reward 1/9
68              return 1 / (3 * 3)
```

Conclusions and outlook

- MCTS outperforms Q-Learning
- DeepRL is very promising

- Investigate why Q-Learning underperformed so much
- Investigate why selfplay did not have an impact
- Tune NN hyperparameters

- Actually train an agent for Connect4 and submit it to Kaggle!

Questions on the project