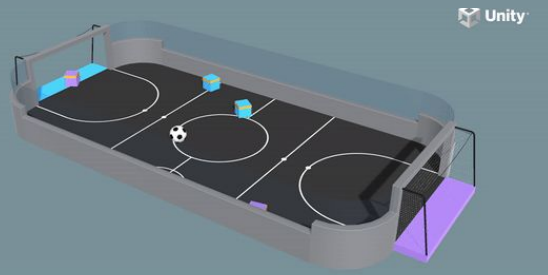
Strategic Interplay: Independent vs. Cooperative Multi-Agent Reinforcement Learning in 2-on-2 Soccer

Nelson Lin (nl2873)



Overview

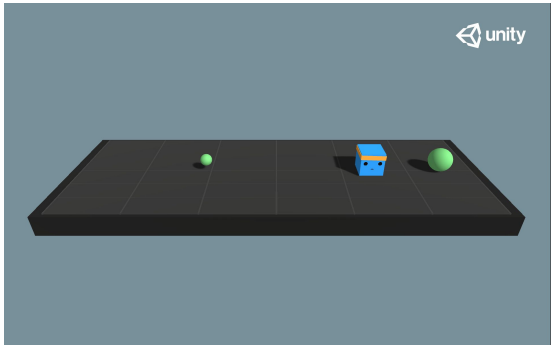
- **Multi-agent reinforcement learning (MARL)** explores how interactions between coexisting agents affects their policies and behavior
- Closely resembles real-world dilemmas, where there can be multiple actors in play at any point
    - i.e. autonomous vehicles, sports, economic markets, networks
- Independent algorithms (i.e. DQN) tend to not perform well in multi-agent settings.
- So implementing strategies (i.e. joint actions) to account for these complex interactions between dynamic actors is vital

Problem Statement

This project aims to discern the differences between single-agent algorithms (i.e. Deep Q-Learning), and multi-agent algorithms (i.e. Joint Action Learning).

Project Environment: ML-Agents

- Open-source project developed by Unity Technologies
- Set within Unity's game engine environment
- Offers a set of pre-built environments and example projects to help developers get started quickly.
- Tools for training, testing, and evaluating AI agents
- Python API for integrating custom training algorithms



Project Environment: SoccerTwos

- Set-up: Four agents competing in a 2 vs 2 soccer game.
- Objective: Score the ball through opponent's goal while preventing them from scoring on own goal

https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/Learning-Environment-Examples.md#soccer-twos

SoccerTwos: Parameters

- Observation Space
  - 336 total units, that can represent 6 objects as well as object distances
  - 264 forward-facing units, 11 ray-casts, spanning 120 degrees of rotation
  - 72 backward, 3 ray-casts, spanning 90 degree
- Action Space
  - Discrete actions
  - 3 directions: forwards-backwards, sideways and rotation
  - Each agent originally had 8 distinct action branches, but was simplified for project
- Reward:
  - Issued after every goal scored or episode end to each team
  - If goal scored: (1 - accumulated time penalty)
  - If goal conceded: -1
- Other Parameters / Information
  - Ball scale, gravity

Algorithms: Deep Q Learning (IDQN)

- Utilization of neural network (DQN) for value function approximation: $Q_\theta(s_t, a)$
- Experience replay
- Epsilon-greedy exploration strategy
- Goal: minimize loss function between Q value and expected reward
- Loss Function: $\frac{1}{N}\sum_{i=1}^{N}(Q_\theta(s_i, a_i) - (r_i + \gamma \max_a Q_\theta(s_i', a))^2$

Algorithm 10 Deep Q-learning

Initialize value network Q with random parameters θ
Repeat for every episode:
for time step $t = 0, 1, 2, \ldots$ **do**
  Observe current state s^t
  With probability ϵ: choose random action $a^t \in A$
  Otherwise: choose $a^t \in \arg\max_a Q(s^t, a; \theta)$
  Apply action a^t; observe reward r^t and next state s^{t+1}
  if s^{t+1} is terminal **then**
    Target $y^t \leftarrow r^t$
  else
    Target $y^t \leftarrow r^t + \gamma \max_{a'} Q(s^{t+1}, a'; \theta)$
  Loss $\mathcal{L}(\theta) \leftarrow \left(y^t - Q(s^t, a^t; \theta)\right)^2$
  Update parameters θ by minimizing the loss $\mathcal{L}(\theta)$

DQN Pseudocode [2]

Algorithms: Joint Action Learning (JAL)

- Similar approach to DQN
  - Uses experience replay and epsilon-greedy
- But, value function is not based solely off actions of the current agent
- Instead it attempts to learn a joint-action value function conditioned on the actions taken by every agent at a given time step

Algorithm 23 Deep joint-action learning

Initialize n value networks with random parameters $\theta_1, \ldots, \theta_n$
Initialize n target networks with parameters $\overline{\theta}_1 = \theta_1, \ldots, \overline{\theta}_n = \theta_n$
Initialize n policy networks with parameters $\phi^1_{-1}, \ldots, \phi^n_{-n}$
Initialize a replay buffer for each agent $D_1, D_2, \ldots, D_n$
for time step $t = 0, 1, 2, \ldots$ **do**
  Collect current observations $o^t_1, \ldots, o^t_n$
  for agent $i = 1, \ldots, n$ **do**
    With probability ϵ: choose random action a^t_i
    Otherwise: choose $a^t_i \in \arg\max_{a_i} AV(h_i, a_i; \theta_i)$
  Apply actions $(a^t_1, \ldots, a^t_n)$; collect rewards $r^t_1, \ldots, r^t_n$ and next observations $o^{t+1}_1, \ldots, o^{t+1}_n$
  for agent $i = 1, \ldots, n$ **do**
    Store transition $(h^t_i, a^t_i, r^t_i, h^{t+1}_i)$ in replay buffers D_i
    Sample random mini-batch of B transitions $(h^k_i, a^k_i, r^k_i, h^{k+1}_i)$ from D_i
    if s^{k+1} is terminal **then**
      Targets $y^k_i \leftarrow r^k_i$
    else
      Targets $y^k_i \leftarrow r^k_i + \gamma \max_{a'_i \in A_i} AV(h^{k+1}_i, a'_i; \overline{\theta_i})$
    Critic loss $\mathcal{L}(\theta_i) \leftarrow \frac{1}{B} \sum_{k=1}^{B} \left(y^k_i - Q(h^k_i, \langle a^k_i, a^k_{-i} \rangle; \theta_i) \right)^2$
    Model loss $\mathcal{L}(\phi^i_{-i}) = \sum_{j \neq i} \frac{1}{B} \sum_{k=1}^{B} -\log \hat{\pi}^i_j(a^k_j \mid h^k_i; \phi^i_j)$
    Update parameters θ_i by minimizing the loss $\mathcal{L}(\theta_i)$
    Update parameters ϕ^i_{-i} by minimizing the loss $\mathcal{L}(\phi^i_{-i})$
    In a set interval, update target network parameters $\overline{\theta}_i$

JAL Pseudocode [2]

Joint Actions

- Each agent, i, estimates the policy of the other agents, j, parametrized as:

$$\hat{\pi}_{i,-i} = \{\hat{\pi}_{i,j}\}_{j \neq i}$$

- This can be done by modelling each policy, with parameters, $\phi_{i,j}$
- Then, we can generate a probability distribution of each agent's action at any given state, and using that to evaluate a greedy action:

$$AV_{\theta_i}(s_i, a_i) = \sum_{a_{-i} \in A_{-i}} Q_{\theta_i}(s_i, \langle a_i, a_{-i} \rangle) \prod_{j \neq i} \hat{\pi}_{\phi_{i,j}}(a_j | s_i)$$

Joint Actions

- But, a problem arises: this method scales poorly as the number of agents (or even the action space) increases
- A solution was found where the general probability distributions can be bootstrapped by using actions from history
- Sample a fixed number K joint actions, and compute the expected Q values:

$$AV_{\theta_i}(s_i, a_i) = \frac{1}{K} \sum_{k=1}^{K} Q_{\theta_i}(s_i, \langle a_i, a_{-i}^k \rangle)|_{a_j^k \sim \hat{\pi}_{i,j}(\cdot|s_i)}$$ [2]

- In some cases, this converges faster and performs better than the original [2]

Methodology

IDQN

- Each agent has a unique replay history
- Allowing them to learn their own Q-values
- The environment gets an action from every agent before taking a step
- Then individual transitions and rewards are distributed to each agent

JAL

- Each agent has a unique replay history
- But, this buffer also includes actions of other agents as well.
- For this project, only the actions of agent's teammates are added to the buffer
- So, each agent only explicitly learns to cooperate
- Evolutionary strategy: updates each agent to the best performing one at defined intervals

Methodology: Hyperparameter Setting

- episodes: 50000
- epsilon strategy: adaptive decaying epsilon from 1 to 0.001
- learning_rate: 0.0005
- gamma: 0.99
- model updates: every 256 actions
- target model updates: every 1000 actions
- generation updates: every 100 episodes
- replay batch size: 128
- replay max. memory: 10000
- state done penalty: 0

Methodology: Experiments

IDQN

- Trained for 4502 episodes, 580000 time steps

JAL

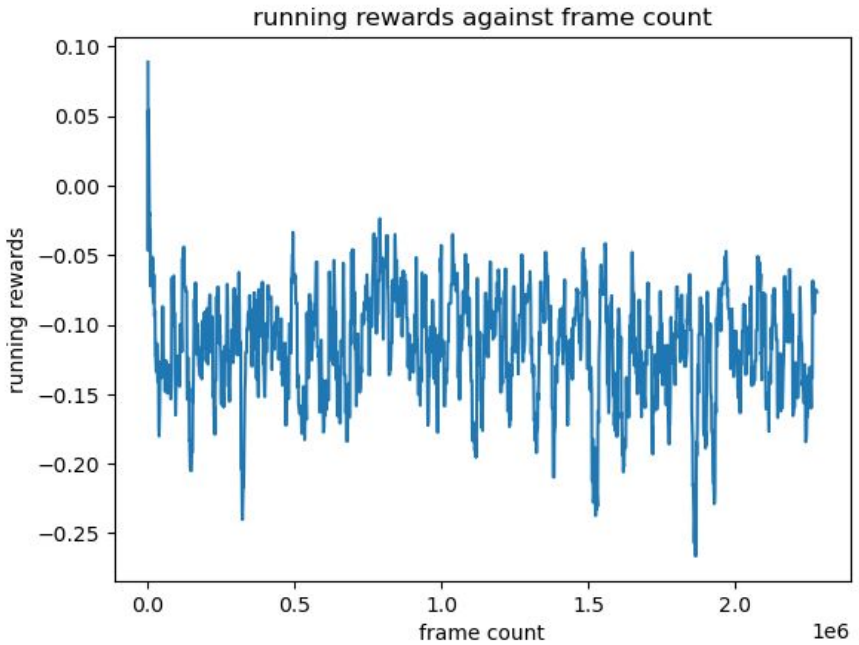
- Trained for 11233 episodes, 1367000 time steps

IDQN vs JAL

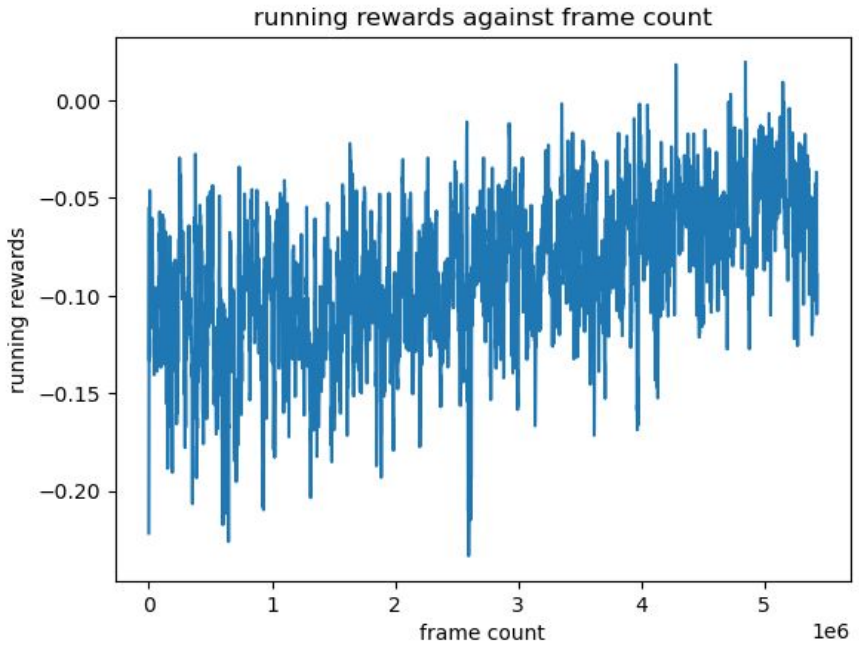
- Tested against each other for 100 episodes

Results: Running Rewards

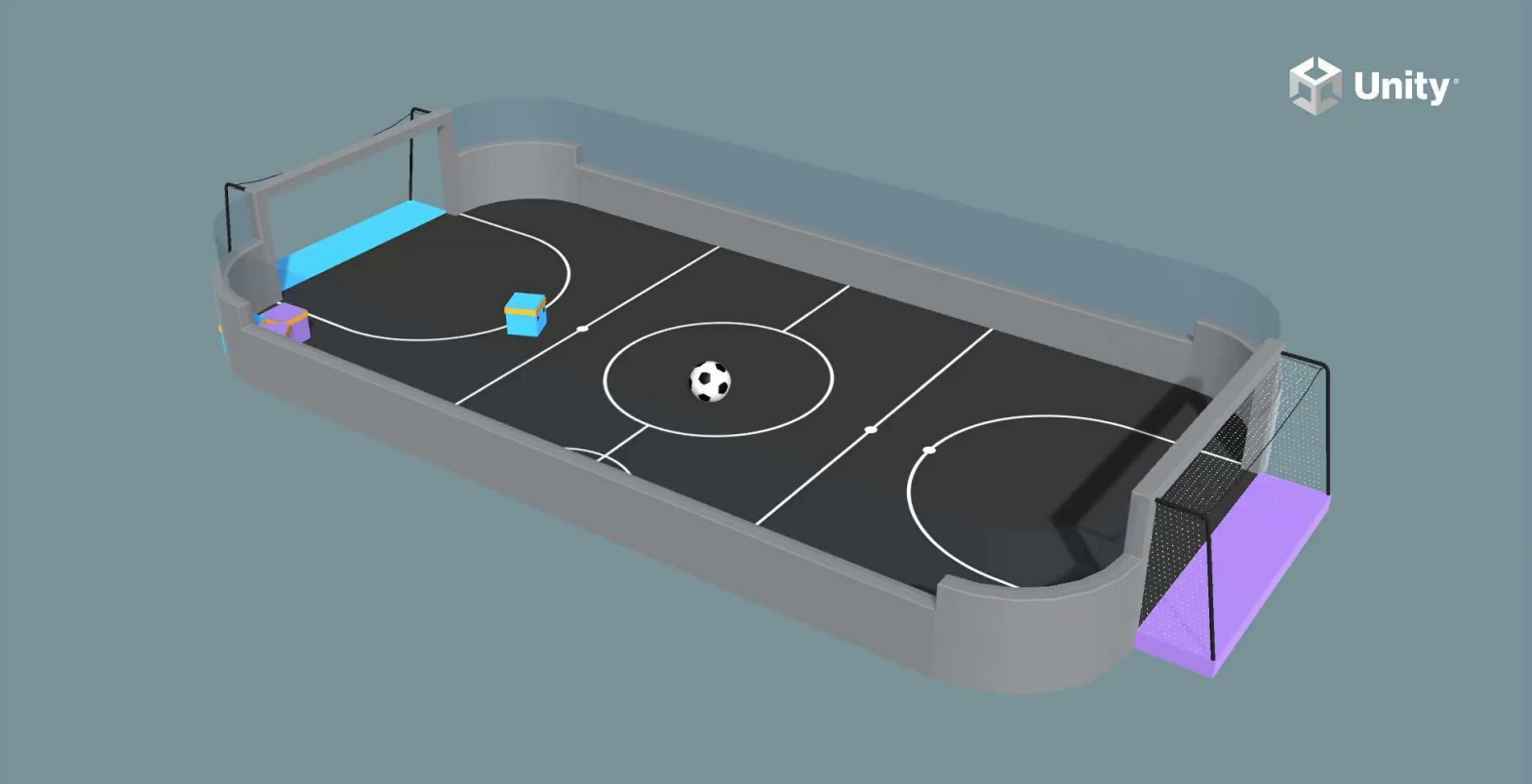
IDQN



JAL



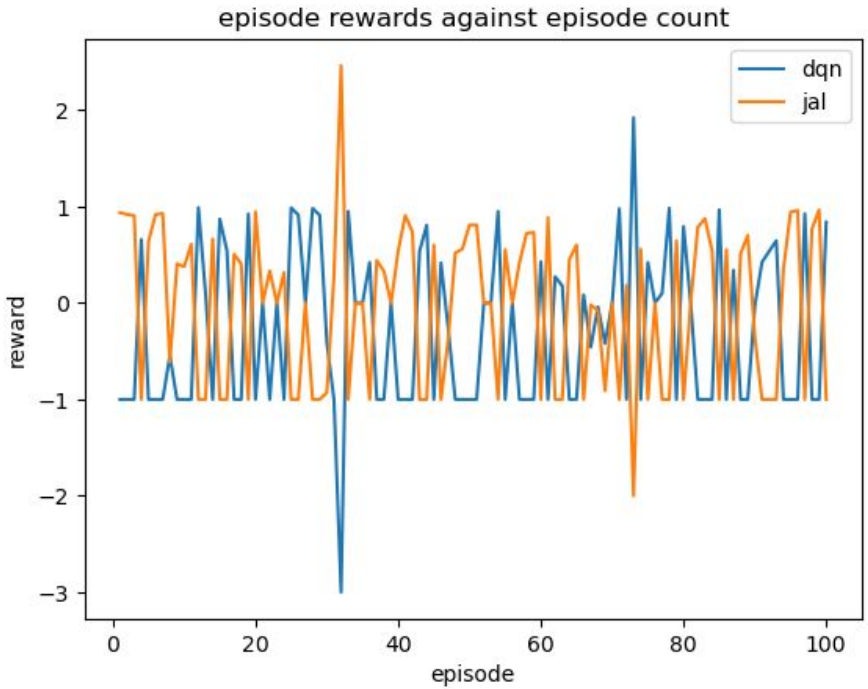
Results: DQN vs JAL



Results: DQN vs JAL

Total Rewards (over 100 episodes):

- DQN = -29.311
- JAL = -4.717



Discussion: Analysis

- Visually, the movements of the JAL agent are more controlled, while DQN's are more erratic
- JAL quantitatively performed better both in a standalone test and head-to-head against DQN
    - Could be attributed to the learning of joint actions between teammates
- Yet, both models still exhibit random movements for a majority of testing.

Discussion: Improvements

- Longer training time: since positive rewards are very sparse, it might benefit the model to train for longer
- Removing simplifications made to environment and models
  - Using other (or all) agents in Joint Action approximation
  - Utilizing full range of actions
- More comprehensive hyperparameter search
- Experiment with improving the existing algorithms: Double-DQN
- Exploring more complex NN architectures: RNNs [1]

Discussion: Future Steps

- Comparing other single vs multi-agent algorithms: Actor-Critic
- Exploring other multi-agent algorithms: V-Learning, Value-Decomposition Networks
- Exploring single agent algorithms: PPO (has shown promise in MARL)

References

[1] J. Posor, L. Belzner, A. Knapp. "Joint Action Learning for Multi-Agent Cooperation using Recurrent Reinforcement Learning," in *Digitale Welt*, vol. 4, pp. 79-84, 2020.

[2] S. Albrecht, C. Filippos, L. Schäfer, *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches.* MIT Press, 2024.

[3] Mnih, V., et al, "Playing Atari with Deep Reinforcement Learning," 2013.

[4] Wong, A., et al, "Deep Multiagent Reinforcement Learning: Challenges and Directions," 2022.

[5] A. Hafiz, G. Bhat, "Deep Q-Network Based Multi-agent Reinforcement Learning with Binary Action Agents," 2020.