COMPARING PERFORMANCE OF STANDARD Q-LEARNING TO DEEP Q-LEARNING IN CUSTOM GRIDWORLD FOR COMPETITIVE AND COOPERATIVE TASKS

Team 2

CSE 616: Multi-Agent Systems

December 08, 2022

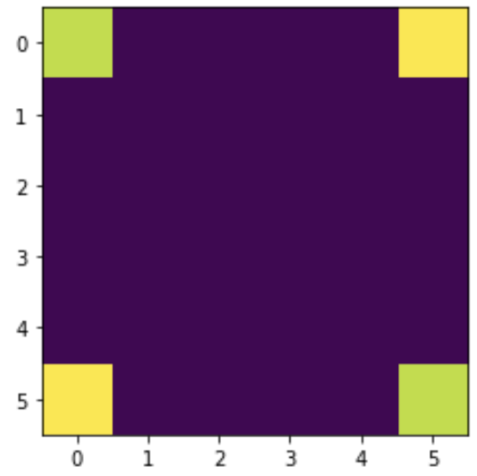
Instructed by: Dr. Vereshchaka

Presented by: Kishan S. Patel

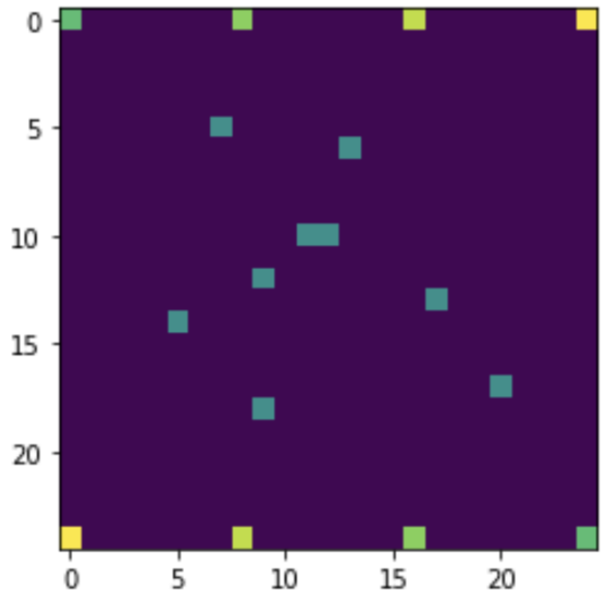
University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

Project Description

- Custom grid worlds created to evaluate Q-Learning and Deep Q-Learning Agents in cooperative and competitive settings
- Size and obstacles of grid worlds varied in order to test the strength of both agents while increasing comlexity
- Agents evaluated on reward per episode and timesteps per episode
- First was a competitive task of agents to reach their goal state as quickly as possible irrespective of the other agents
- Second task was cooperative aimed at navigating the obstacles and reaching the goal states as a collective



2 Agent 6x6 Gridworld



4 Agent 25x25 Gridworld with 10 Landmarks

Background – Standard Q Learning

- First implementation developed by Chris Watkins in 1989
- Q-Learning is a value-based reinforcement learning algorithm using Q-values (action values) to iteratively improve the behavior of the learning agent
- Goal is to maximize the Q-value to find the optimal action-selection policy
- *Q(s, a)* represents estimation of utility of action *a* at state *s*
- *Q(s, a)* is iteratively calculated using temporal difference update rule derived from Bellman equation:

$$Q_t(s,a) = Q_{t-1}(s,a) + \alpha(R(s,a) + \gamma \max Q(s',a') - Q_{t-1}(s,a))$$

- Uses epsilon-greedy action selection policy:

$$\text{Action at time(t)} \begin{cases} \max Q_t(a) & \text{with probability } 1-\epsilon \\ \text{any action (a)} & \text{with probability } \epsilon \end{cases}$$

Background – Deep Q-Learning

- Developed by Google DeepMind in 2014
- Uses deep neural networks to estimate the action-value function Q(s, a)
- Network takes in current state of environment as input, and outputs expected future reward for each possible action; agent chooses action with highest expected reward
- Deep Q-Learning uses experience replay during training to store and learn from past experiences
  - Involves storing a replay buffer of past experiences and sampling from buffer to train the neural network
  - Helps improve stability and convergence of the algorithm

- Can be implemented using centralized and decentralized approach
  - Centralized: neural network is shared between all agents
  - Decentralized: each agent has it's own network

Implementation – Standard Q-Learning Details

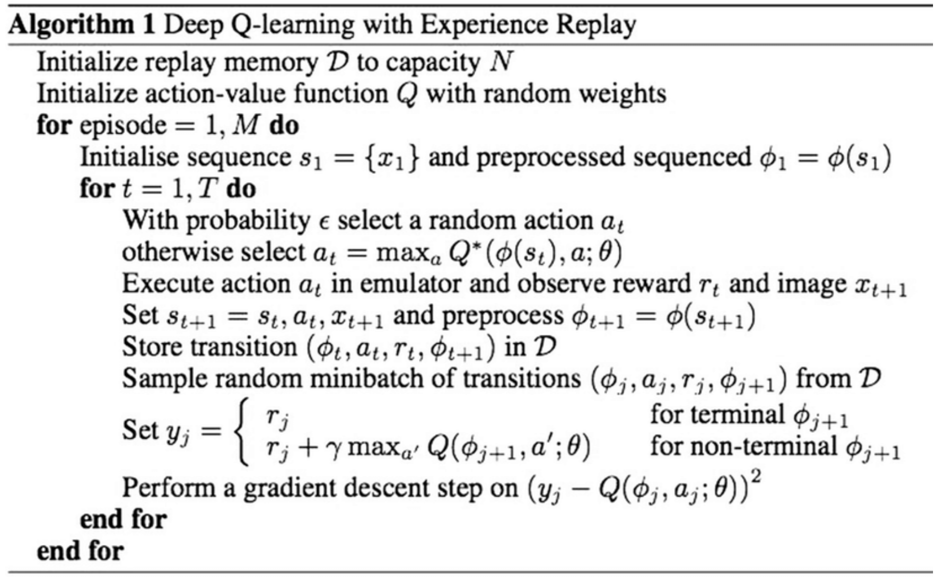
- Initialize discount factor gamma
- Initialize exploration rate and epsilon decay rate
- Initialize learning rate alpha
- Initialize rewards in environment
- Initialize Q matrix as zeros
- Choose action for each agent according to epsilon-greedy policy
- Observe reward of each agent
- Get maximum Q-value for next state based on actions list
- Update Q-value
- Update decay rate equation
- Repeat process until goal states achieved or max episodes reached

```
Algorithm 1: Epsilon-Greedy Q-Learning Algorithm
  Data: $\alpha$: learning rate, $\gamma$: discount factor, $\epsilon$: a small number
  Result: A Q-table containing Q(S,A) pairs defining estimated
          optimal policy $\pi*$
  /* Initialization                                              */
  Initialize Q(s,a) arbitrarily, except Q(terminal,.);
  Q(terminal,.) $\leftarrow$ 0;
  /* For each step in each episode, we calculate the
     Q-value and update the Q-table                              */
  for each episode do
      /* Initialize state S, usually by resetting the
         environment                                             */
      Initialize state S;
      for each step in episode do
          do
              /* Choose action A from S using epsilon-greedy
                 policy derived from Q                           */
              A $\leftarrow$ SELECT-ACTION(Q, S, $\epsilon$);
              Take action A, then observe reward R and next state S';
              Q(S, A) $\leftarrow$ Q(S, A) + $\alpha$ [ R + $\gamma \max_a$ Q(S', a) - Q(S, A)];
              S $\leftarrow$ S';
          while S is not terminal;
      end
  end
```

Implementation – DQN Details

- Define network class for DQN:
    - List of layers that define architecture of the neural network (input/output/hidden)
        - Implemented via TensorFlow
    - Method to initialize weights of each layer using He initialization
    - Method for taking in an input layer and applying network's layers to produce output tensor representing Q-values for each possible action
    - Method for taking in batch experiences and using it to update network weights using gradient descent optimizer

- Define experience replay class for DQN:
    - A buffer for storing experiences implemented as deque data structure
    - Method for appending experiences to replay buffer
    - Method for sampling batch of experiences from replay buffer
    - Method for resetting the replay buffer

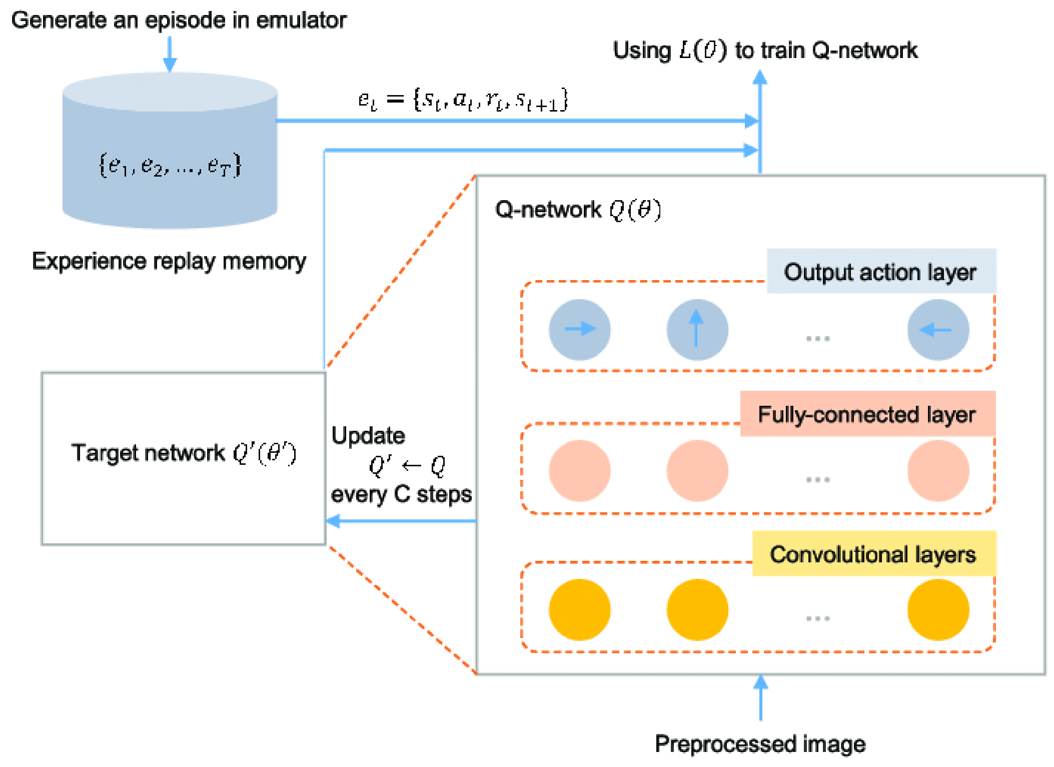


Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
    end for
end for
```

Implementation – DQN Details

- Define agent class for DQN:
  - Network object for DQN
  - Experience Replay object for storing and sampling experiences
  - Method for selecting actions using epsilon-greedy selection policy
  - Method for updating Q-network using sample batches from replay buffer
  - Method for resetting agent and returning initial state of environment
- Define Multi Agent class for DQN:
  - List of agent objects for each agent
  - Same methods in agent class but with joint execution

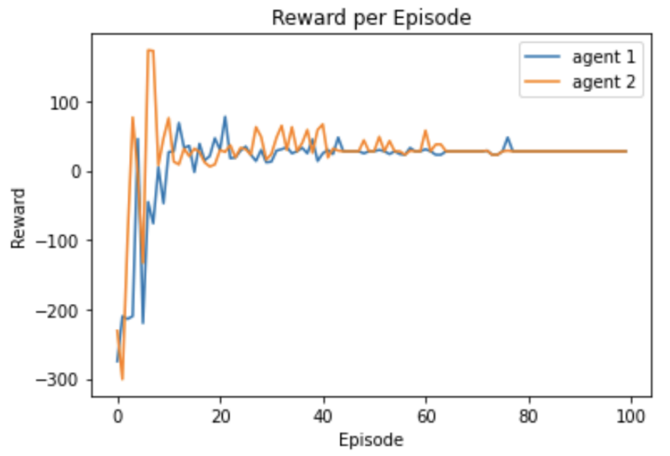

Generate an episode in emulator
Using $L(\theta)$ to train Q-network
$e_t = \{s_t, a_t, r_t, s_{t+1}\}$
$\{e_1, e_2, \ldots, e_T\}$
Experience replay memory
Q-network $Q(\theta)$
Output action layer
Fully-connected layer
Convolutional layers
Target network $Q'(\theta')$
Update $Q' \leftarrow Q$ every C steps
Preprocessed image

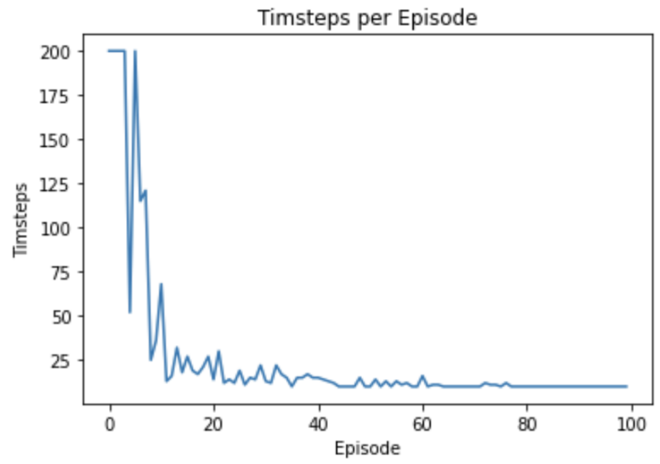
Implementation Steps – DQN

1. Define the grid world environment, including the size of the grid, the locations of obstacles and rewards, and any other relevant information.
2. Define the agents that will operate in the environment, including their goals, policies, and initial positions.
3. Initialize the neural network that will be used to approximate the action-value function Q(s, a) for each agent. This network should take in the current state of the environment and the actions of all the agents, and output the expected rewards for each agent.
4. Initialize the replay buffer that will be used to store and learn from past experiences. This buffer should store tuples of (state, action, reward, next state) for each agent.
5. For each time step in the environment:
    - Have each agent choose an action based on its current policy and the current state of the environment.
    - Update the state of the environment based on the actions of the agents.
    - Compute the rewards for each agent based on the new state of the environment and the agents' goals.
    - Store the experience (state, action, reward, next state) in the replay buffer for each agent.
    - Sample a mini-batch of experiences from the replay buffer, and use these experiences to update the weights of the neural network using the Bellman equation.
6. Continue this process for a specified number of time steps or until the agents reach their goals.
7. Evaluate the performance of the agents and the neural network, and make any necessary adjustments to improve the learning process.

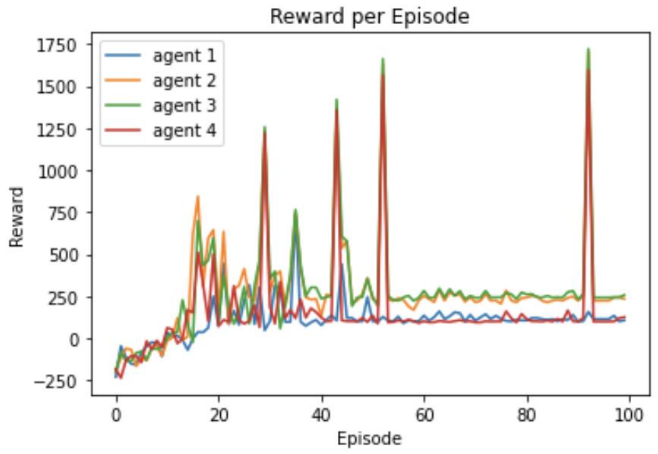
Results: Competitive Standard Q-Learning

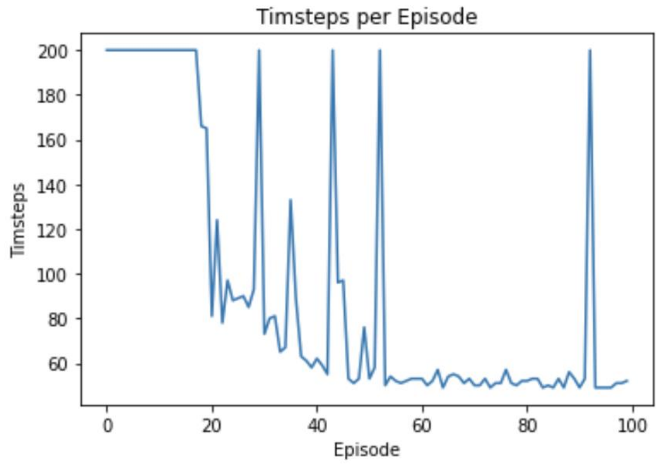
2 Agents, 6x6 Grid, No Obstacles



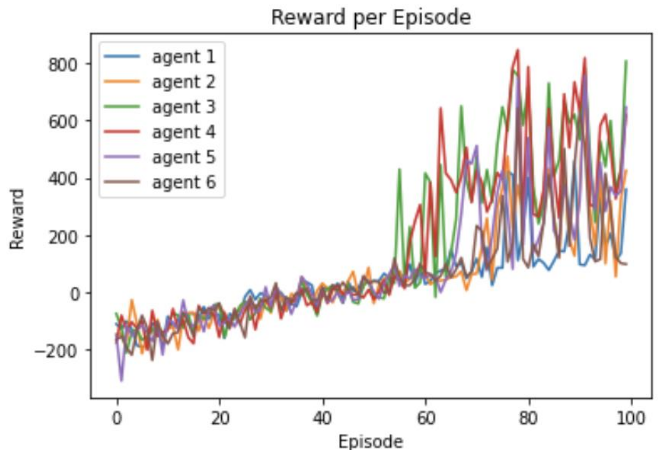


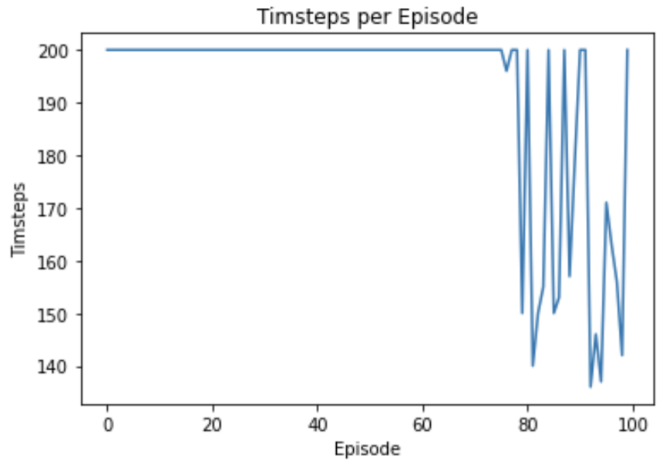
4 Agents, 25x25 Grid, 10 Obstacles





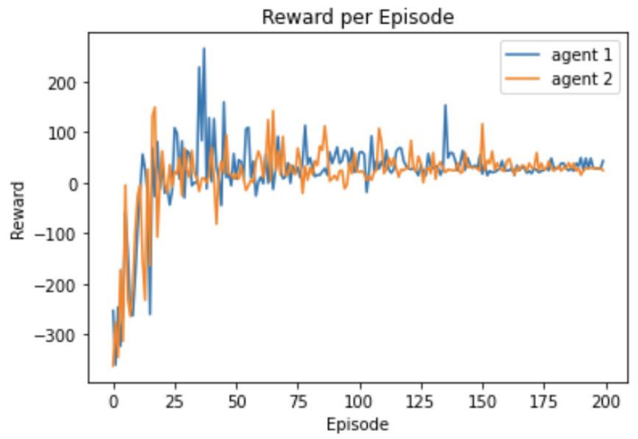
6 Agents, 100x100 Grid, 30 Obstacles

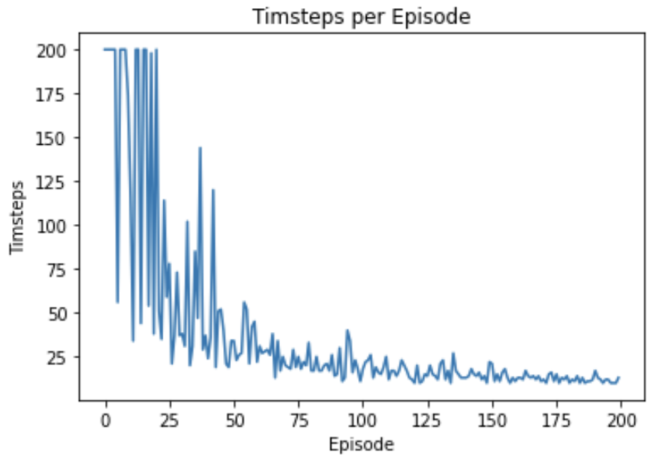




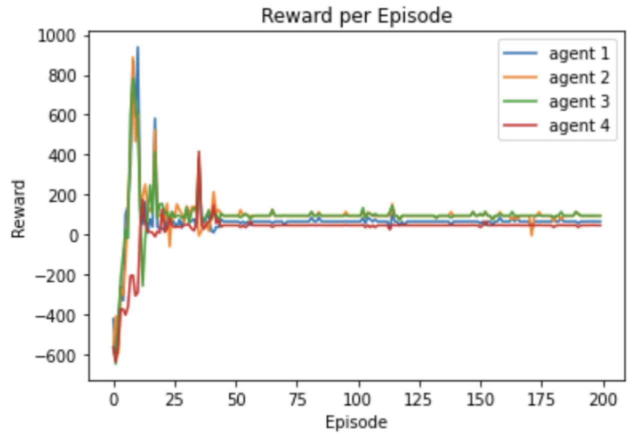
Results: Competitive Deep Q-Learning

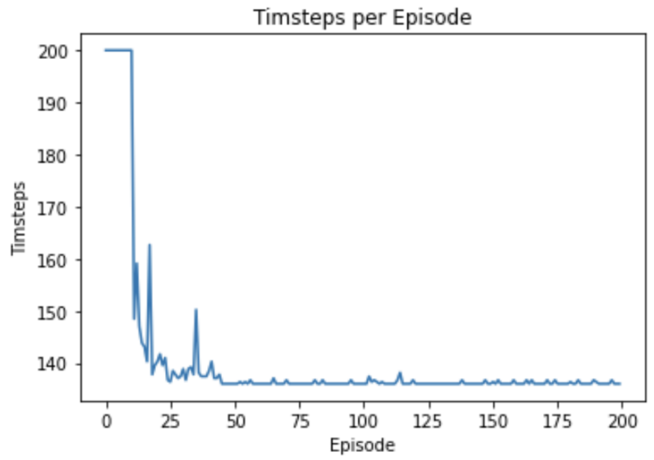
2 Agents, 6x6 Grid, No Obstacles



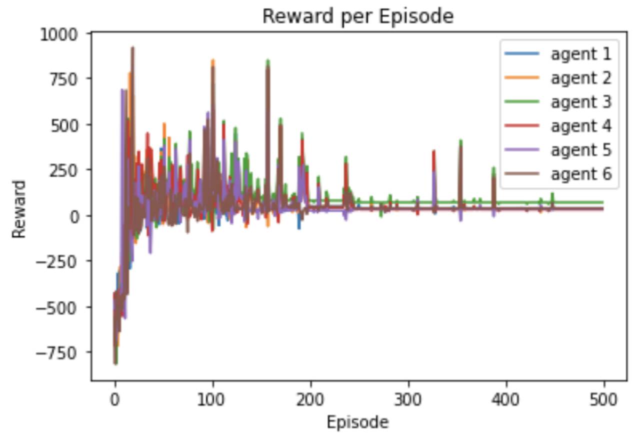


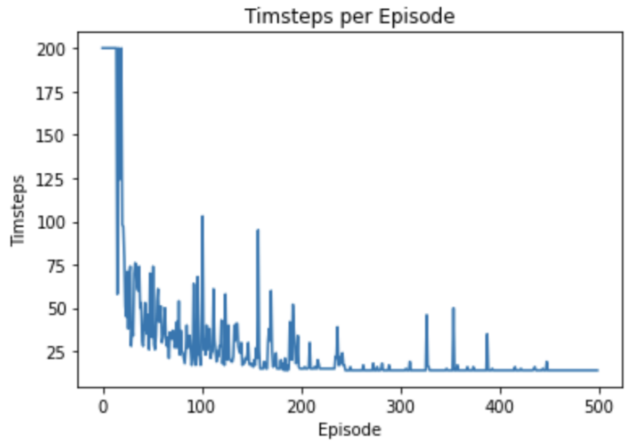
4 Agents, 25x25 Grid, 10 Obstacles





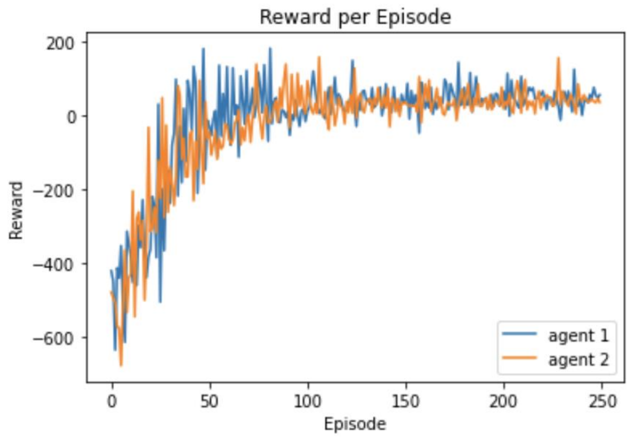
6 Agents, 100x100 Grid, 30 Obstacles

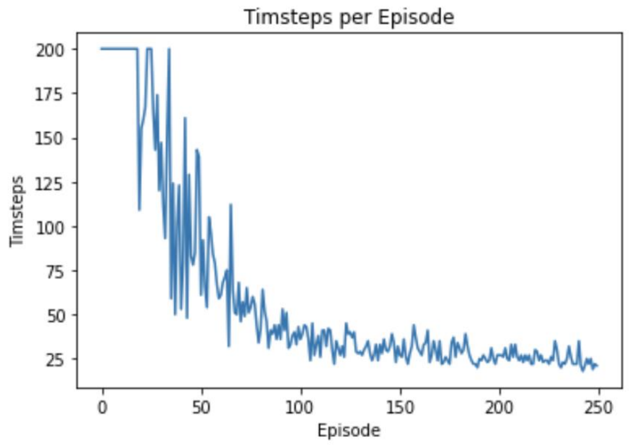




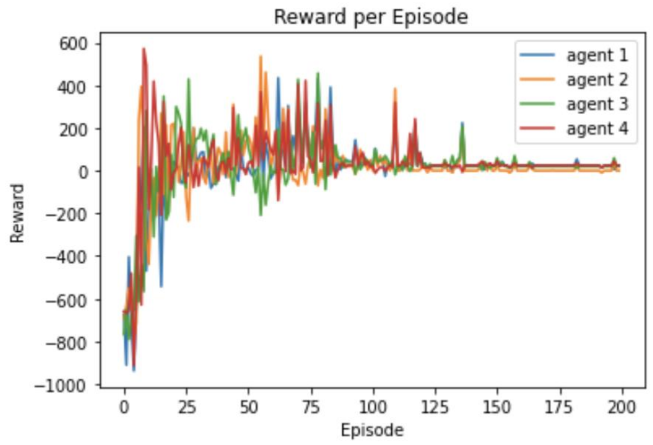
Results: Cooperative Deep Q-Learning

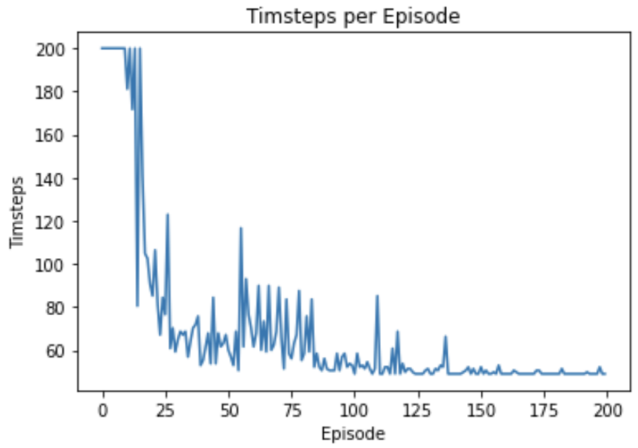
2 Agents, 6x6 Grid, No Obstacles



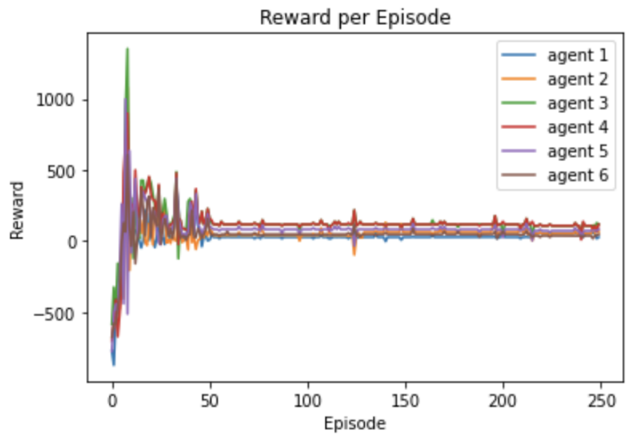


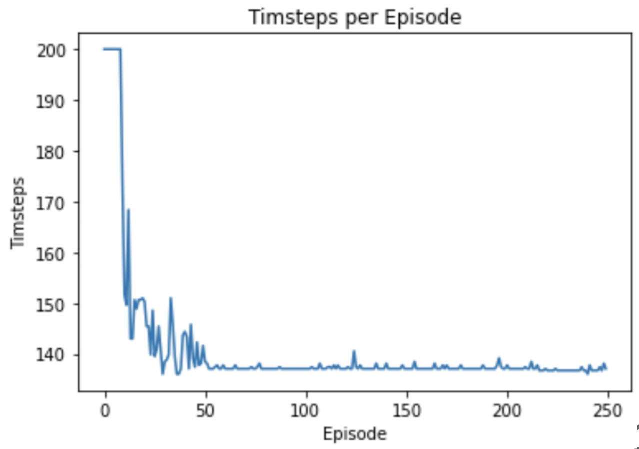
4 Agents, 25x25 Grid, 10 Obstacles





6 Agents, 100x100 Grid, 30 Obstacles





Results Summary

Table 1: Evaluation of Episodes to Convergence

	Comp Q-Learning	Comp Deep Q-Learning	Coop Deep Q-Learning
Scenario 1	85 episodes	200 episodes	250 episodes
Scenario 2	100 episodes (noisy)	65 episodes	150 episodes
Scenario 3	Unstable	300 episodes	125 episodes

Table 2: Shared Model Parameters

Model Parameters	
Learning Rate	0.7
Exploration Decay Rate	0.03
Discount Factor	0.9
Max Timesteps	200

Key Observations

- In a complex environment, it can be difficult for agent acting alone to learn an effective policy. Shared experience can help multiple agents learn more efficiently and improve their performance

- In simpler environments, increased complexity of shared experience DQN may not be necessary, as regular Q-learning may be a more suitable choice

- Cooperative tasks benefit greatly from centralized learning

Main Takeaway: Model complexity should be in line with environment/task complexity in order to allocate resources most effectively and efficiently. Just because a model is more intensive does not mean it will display superior performance.

Contribution Summary

Team Member	Contribution %
Kishan Patel	100

Thank You!