LAB GUIDE 03

LEARNING AGENTS

In this lab, we will implement a method to learn a payoff/utility function for a single agent or multiagent system. With this payoff function, our learning agent can then select the action that has the highest utility at each state. In particular, we will implement an agent that learns a payoff function called the *Q-function*. We will use the Q-learning algorithm to learn from past experiences the utility of selecting action a_t at state x_t. The Q-learning algorithm is extensively used in reinforcement learning to estimate an optimal *Q-function*, which can also be seen as a utility function. This handout briefly presents the Markov decision process (MDP), the theoretical framework underpinning reinforcement learning. Then, it will guide you on how to use reinforcement learning in practice. Section 3.1 introduces the pipeline in a reinforcement learning setting. In section 3.2, we train an autonomous learning agent using reinforcement learning and compare against the baselines from Lab 01. In section 3.3, we train an agent that controls a whole team of predators and compare against a team of agents like the ones trained in the previous section.

1. LEARNING FROM EXPERIENCE

If you remember the previous labs, the step method for the environment yielded four variables: (i) a next observation of the environment; (ii) a reward signal; (iii) a flag telling whether the episode was over; and (iv) additional info. We developed two agents in lab 1 (i.e., a Random agent and the Greedy agent for the Predator-Prey domain), one which did not use any information provided by the environment and another which relied only on the observations. Neither benefited from the reward signal, which indicates how good the agent's last action was.

In this lab, the agent will learn from reward signals provided by the environment. The rewards will guide the agent towards more beneficial long term actions. *Reinforcement learning* is a trial-and-error method implemented by a learning agent to improve the performance of the Predator Prey task. We can extend the autonomous agent and environment iteration loop seen in the previous classes to incorporate this learning method, as seen in Fig. 1.

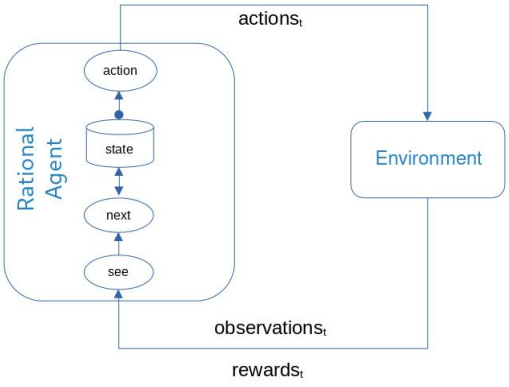


Fig. 1: *See* method now takes as input an *observation* and *reward*. *Next* performs computations and updates the internal agents' *state*.
FInally, the agent chooses an *action* according to the internal *state*.

1.1. The Markov Decision Process

The Markov decision process (MDP) framework unites many of the elements of autonomous agent systems that have so far been discussed throughout the course. It can be defined as the tuple:

$$(S,\ A,\ \mathbb{P},\ R,\ \gamma)$$

Where

- S is a set of (environment) states, where $x_t \in S$ is the observation made by the agent at timestep t.
- A is a set of (agent's) actions and $a_t \in A$ is the actions taken by the agent at timestep t.
- $\mathbb{P}[S_{t+1} = y | S_t = x, A_t = a]$ is the stochastic transition model regulating the environment dynamics in response to the agent's actions.
- $R: S \times A \to \mathbb{R}$. Upon executing an action a in state x, the agent receives a (possibly random) reward with expectation given by $r(x, a)$.
- $\gamma \in (0, 1]$ is a discount rate regulating the indifference the agents have for collecting short term rewards.

At each step t, the agent receives an observation $x_t \in S$ of the environment and selects an action $a_t \in A$. The environment then transitions to a state $S_{t+1} \sim \mathbb{P}[S_{t+1} = y | S_t = x, A_t = a]$. The goal of the agent is to select its actions so as to maximize the expected total discounted reward (TDR),

$$TDR = E[\sum_{t=0}^{\infty} \gamma^t R_t],$$

where R_t is the random reward received at time step t (with $E[R_t] = r(x, a)$) and the scalar γ is a discount factor ($0 \leq \gamma \leq 1$). The long-term value of an action a in a observation x_t is captured by the optimal Q-value (i.e., $Q^*(x, a)$), which can be estimated using, for example, the Q-learning algorithm (Watkins 1989).

The Q-learning algorithm estimates the optimal Q-values as the agent interacts with the environment. Given a transition (x_t, a_t, r_t, x_{t+1}) experienced by the agent, Q-learning performs the update at each time step t as follows:

$$\hat{Q}_{t+1}(x_t, a_t) \leftarrow \hat{Q}_t(x_t, a_t) + \alpha_t(r_t + \gamma \max_{a'} \hat{Q}_t(x_{t+1}, a') - \hat{Q}_t(x_t, a_t)),$$

where α_t is the learning rate ($0 < \alpha \leq 1$). Upon computing the optimal Q-values (Q^*), the agent can act optimally by selecting the action that maximizes the optimal Q-values at each observation $x \in S$, as follows:

$$\pi^*(x) = \operatorname{argmax}_a Q^*(x, a),$$

where the mapping $\pi^*: S \to A$, from observation x to the corresponding optimal action a, is known as the optimal policy for the MDP.

1.2. Q-learning.

The Q-learning algorithm utilizes the Q update expression above with a criteria for the action selection, according to pseudocode on Listing 1:

```
Input α in (0,1], small ε in [0,1]
initialize Q(x, a) = 0, for all x in S and a in A
For Each episode Do
      Observe x
      Until x is terminal Do
            # Choose action a using a policy derived by Q (e.g, ε-greedy)
            p ~ Uniform[0, 1]
            If p < ε Do       # ε-greedy criteria
                  Choose a ∈ A arbitrarily
            Else
                  Choose a = argmax_u Q(x, u)
            End If
            Take action a and observe y and r
            Q(x, a) = Q(x, a) + α[r + γ * max_u Q(y, u)- Q(x, a)]
            x = y
      Loop
Loop
```

Listing 1: The Q-learning algorithm (Sutton and Barto, 2018).

The action selection criteria allows the agent to balance **exploration** and **exploitation.** In other words, the selection of actions that are not properly evaluated or the selection of the actions that are considered to be (close to) optimal. The well known **ε-greedy criteria,** uses a random uniformly distributed variable $p \sim Uniform[0, 1]$ and the threshold ε to decide whether to randomly choose an action or choose an action according to the best known policy, the policy derived from the Q-table.

2. Pulling the Latest AASMA Code

Download the code for this lab at: https://github.com/GAIPS/aasma-spring-23/tree/lab3

3. Exercises

3.1. The train-evaluation pipeline and rational autonomous agent.

Go ahead and open the *exercise_1_rational_agent.py* file.

In this exercise you will learn how to set up, train and evaluate a rational agent (in your case, the Q-Learning algorithm).

3.1.1 Complete The Train-Eval Loop

Your first task is to complete the train-eval loop for single agent environments.

A train-eval loop (Fig. 2) performs iterations of training and evaluation steps, in which the agent is trained on an instance of the environment for a number of episodes and then evaluated on another instance of the environment for another number of episodes (respectively).

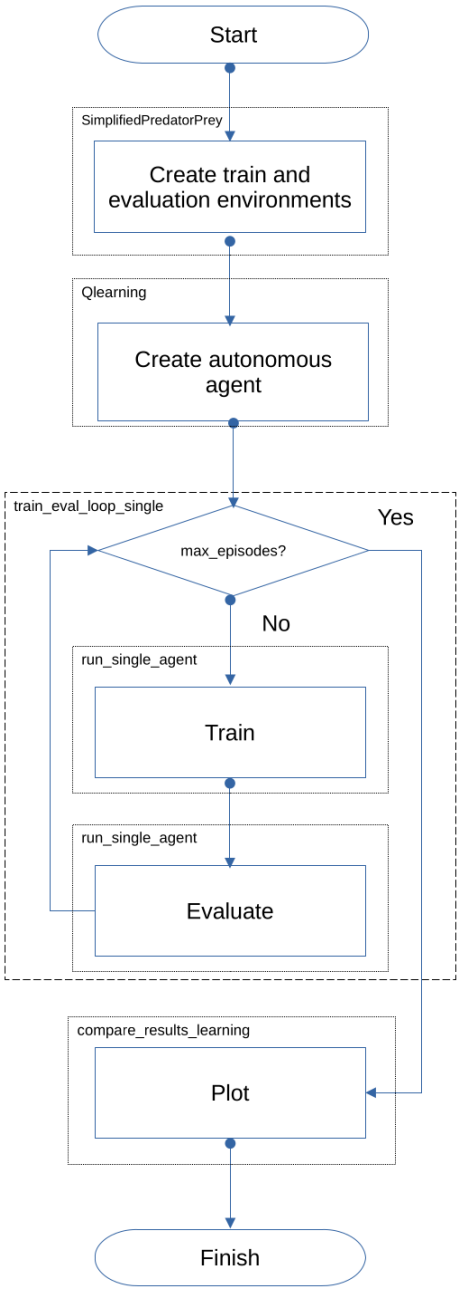


Fig. 2: Train and evaluation loop pipeline implemented *exercise_1_rational_agent.py*. Dotted lines show the class or function that implements that particular block. Dash lines represent the main loop. In exercise 1 you are provided with a partial implementation and must fill in the missing code.

We use two independent environments, one for training and one for evaluation in order to prevent possible biases in the evaluation.

```
def train_eval_loop_single(
        train_environment, eval_environment, agent,
        n_evaluations, n_training_episodes, n_eval_episodes):

    print(f"Train-Eval Loop for {agent.name}\n")

    results = np.zeros((n_evaluations, n_eval_episodes))

    for evaluation in range(n_evaluations):

        print(f"\tIteration {evaluation+1}/{n_evaluations}")

        # Train
        print(f"\t\tTraining {agent.name} for {n_training_episodes} episodes.")
        agent.train()    # Enables training mode
        # TODO - Run train iteration
        run_single(..., ..., ...) # FIXME

        # Eval
        print(f"\t\tEvaluating {agent.name} for {n_training_episodes} episodes.")
        agent.eval()     # Disables training mode
        # TODO - Run eval iteration
        result = run_single(..., ..., ...) # FIXME
        results[evaluation] = result
        print(f"\t\tAverage Steps To Capture: {round(results[evaluation].mean(), 2)}")
        print()

    return results
```

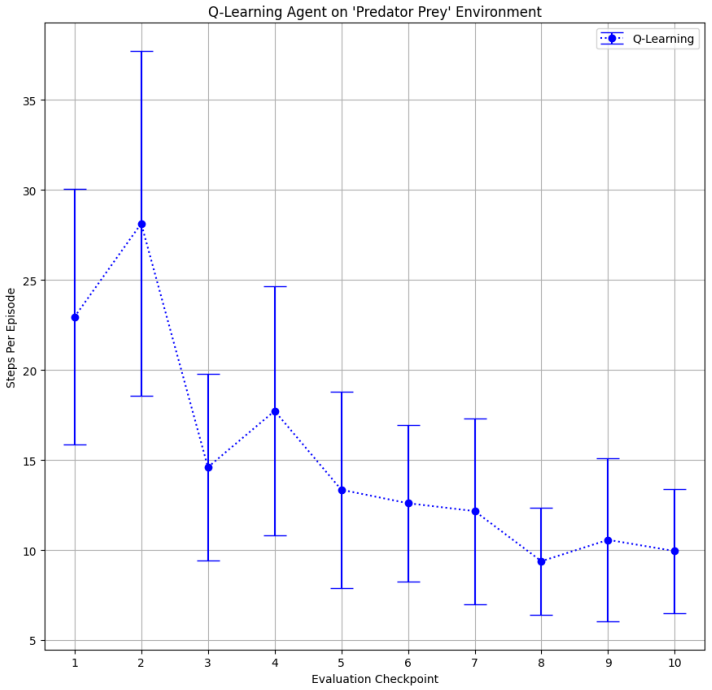
3.1.2 Implement the rational agent.

Go ahead and complete the code for the rational agent Q-Learning.

```
class QLearning(Agent):

    def __init__(self, n_actions,
                 learning_rate=0.3, discount_factor=0.99,
                 exploration_rate=0.15, initial_q_values=0.0):

        self._Q = defaultdict(lambda: np.ones(n_actions) * initial_q_values)
        self._learning_rate = learning_rate
        self._discount_factor = discount_factor
        self._exploration_rate = exploration_rate
        self._n_actions = n_actions

        super(QLearning, self).__init__("Q-Learning")
```

```python
    def action(self, explore=True):

        x = tuple(self.observation)
        # TODO - Access Q-Values for current observation
        q_values = ... # FIXME

        if not self.training or \
            (self.training and np.random.uniform(0, 1) > self._exploration_rate):
            # Exploit
            actions = np.argwhere(q_values == np.max(q_values)).reshape(-1)
        else:
            # Explore
            actions = range(self._n_actions)

        return np.random.choice(actions)

    def next(self, observation, action, next_observation, reward, terminal, info):

        x, a, r, y = tuple(self.observation), action, reward, tuple(next_observation)
        alpha, gamma = self._learning_rate, self._discount_factor

        Q_xa, Q_y = self._Q[x][a], self._Q[y]
        max_Q_ya = max(Q_y)

        # TODO - Update rule for Q-Learning
        self._Q[x][a] = ... # FIXME
```

After completing the tasks, you may check if everything is working correctly by running the script. You should obtain a plot similar to that below.



Answer the following questions:

1. During the training procedure the policy changes, conversely, during the evaluation procedure the policy is kept fixed. What is the point of evaluation?

2. According to the confidence intervals above, is the Q-learning agent in fact learning? Why?

3. In code Listing 2 the exploration rate, or ε, is set to 15% meaning that according to the ε-greedy criteria, the Q-learning agent is expected to select the best known action for observation x with what proportion?

3.2. Comparing Q-Learning with Baselines from Lab 1

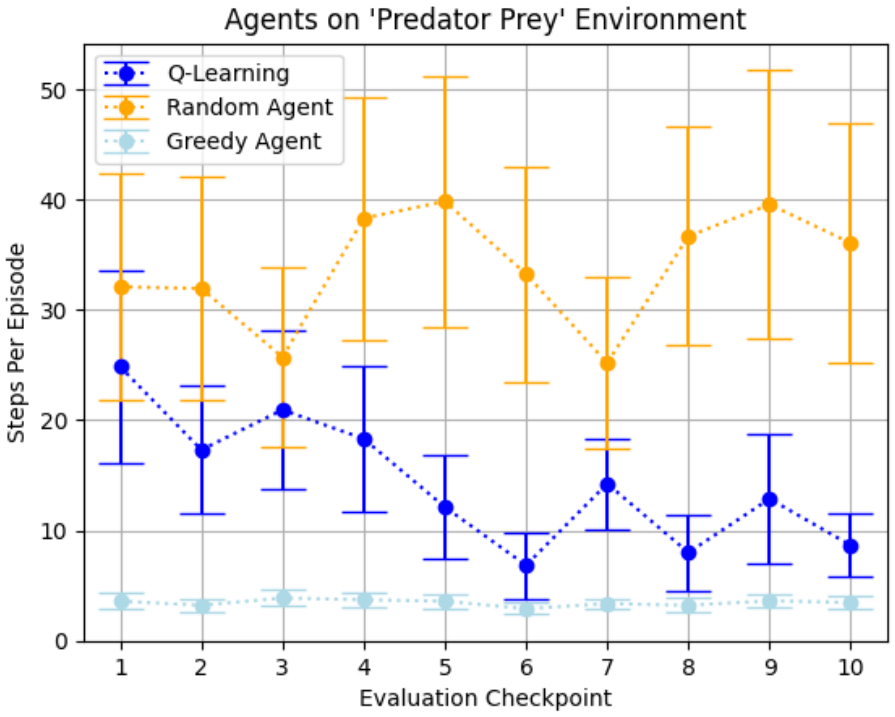
Go ahead and open exercise_2_rational_vs_reactive_agents.py.

This exercise is more of a demonstration rather than an assignment, meant to show you how rational agents improve their performance with experience (while reactive agents do not).

Go ahead and fix the main file by instantiating the missing agents (with their respective parameters).

```
# 2 - Setup agents
agents = [
    ...,                    # FIXME - Instantiate Q-Learning Agent (already imported)
    RandomAgent(train_environment.action_space.n),
    ...                     # FIXME - Instantiate Greedy Agent (already imported)
]
```

Running the code you should expect to see the following plot, which showcases how the rational agent (Q-Learning) improves its performance with experience while the reactive agents (Random and Greedy) do not. Due to the intrinsic variability from the stochastic simulation you may obtain slightly different results.



Answer the following questions:

1. From the three presented agents, which has the best performance? Justify your answer.

2. From what evaluation(s) checkpoint(s) can we say that the rational agent is better than the random agent?

3. Analyze, from the standpoint of the agents' architecture and use of information, the plot above.

Bonus programming challenge:

You may have noticed that although the performance of the learning agent approaches the greedy agent the plot above does not show an overlapping of the confidence intervals. Theoretically, it is possible for the learning agent to reach the performance of the greedy agent. In practice there are many factors influencing the training performance of the Q-learning agent. Try running the training procedure with different combinations of the training hyper parameremeters:

(i) Decrease the learning rate α and increase the number of episodes.

(ii) Try setting the initial exploration rate ε = 1 and reducing at every episode until ε = 0.01.

3.3. Multi-Agent Learning

Unlike single-agent learning, where each predator learns its individual Q-values, we will now consider a multi-agent learning setting in which all predators are controlled by a centralized learner. If a team of two agents, each with two possible actions (**0**: *Move*, **1**: *Stay*) is controlled by a centralized multi-agent, both actions can be parameterized as if they were a single joint-action, which in this simple example would contain four joint-actions (**0**: (*Move*, *Move)*; **1**: (*Move*, *Stay)*; **2**: (*Stay*, *Move)*, **3**: (*Stay*, *Stay*)).

If you are able to adapt your environment in order to consider this abstraction (mapping out in its internal logic the joint-action into two individual actions), then you may directly use the Q-Learning algorithm from exercise 1 without any additional modifications (as the number of actions is now the number of joint-actions).

Go ahead and open exercise_3_multi_agent_learning.py

```
if __name__ == '__main__':

    parser = argparse.ArgumentParser()
    parser.add_argument("--episodes-per-training", type=int, default=100)
    parser.add_argument("--episodes-per-evaluation", type=int, default=64)
    parser.add_argument("--evaluations", type=int, default=10)
    opt = parser.parse_args()

    # 1 - Setup environments
    # We set up two instances of the same environment
    # One for training, one for evaluation
    train_environment = SimplifiedPredatorPrey(grid_shape=(5, 5), n_agents=2, n_preys=1, max_steps=100, required_captors=2)
    eval_environment = SimplifiedPredatorPrey(grid_shape=(5, 5), n_agents=2, n_preys=1, max_steps=100, required_captors=2)

    # 2 - Setup centralized multi-agent learner
    # This can be represented by a single agent controlling a joint action space
```

```
joint_train_environment = JointActionWrapper(train_environment)
joint_eval_environment = JointActionWrapper(eval_environment)
centralized_multi_agent_learner = QLearning(joint_train_environment.n_joint_actions)

# 3 - Setup decentralized learners
decentralized_single_agent_learners = [
    QLearning(train_environment.action_space[0].n),
    QLearning(train_environment.action_space[1].n)
]

# 4 - Reactive Teams
greedy_reactive_team = [
    GreedyAgent(0, 2),
    GreedyAgent(1, 2)
]

# 4 - Evaluate
results = {

    # Note here we can treat Central-Agent RL as a single agent system,
    # Since the environment now accepts team_actions from a single algorithm
    "1 Centralized Multi-Agent Learner": train_eval_loop_single(
        joint_train_environment, joint_eval_environment, centralized_multi_agent_learner,
        opt.evaluations, opt.episodes_per_training, opt.episodes_per_evaluation),

    "2 Decentralized Single-Agent Learners": train_eval_loop_multi(
        train_environment, eval_environment, "QLearning Team", decentralized_single_agent_learners,
        opt.evaluations, opt.episodes_per_training, opt.episodes_per_evaluation),

    "2 Reactive Agents (Greedy)": train_eval_loop_multi(
        train_environment, eval_environment, "Greedy Team", greedy_reactive_team,
        opt.evaluations, opt.episodes_per_training, opt.episodes_per_evaluation),

}

# 5 - Compare results
compare_results_learning(results,
                         title="Multi vs Single-Agent Learning 'Predator Prey' Environment\nReactive Agents for comparison",
                         colors=["blue", "lightblue", "green"])
```

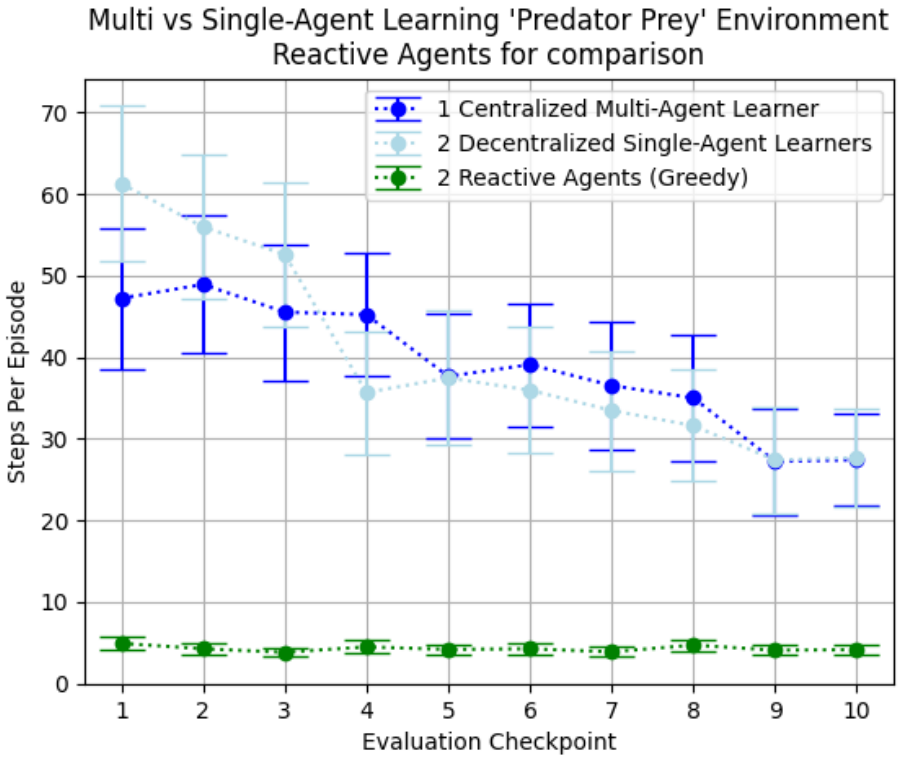
As you can see, the instantiation for the Q-Learning algorithm is the same - all we need is to complete the wrapper which maps a joint-action into the regular actions.

Your task is, therefore, to go ahead and fix the *step* method of the **JointActionWrapper** class in order to convert the joint action into the list of individual actions.
Keep in mind that this code must work for any arbitrary number of agents.

```
class JointActionWrapper(Wrapper):

    """ A Wrapper for centralized multi-agent environments.

    * Allows a single agent to control all agents via a global joint-action.
    * Reduces the N action spaces (where N is the number of agents) to a single joint-action space.

    Example
    -------
    >> N = 2
    >> Available Actions Agent 0 = Available Actions Agent 1 = [ Move (0), Stay (1) ]

    | Action 1 | Action 2 | Team Action |
    |----------|----------|-------------|
    | 0        | 0        | 0           |
    | 0        | 1        | 1           |
    | 1        | 0        | 2           |
    | 1        | 1        | 3           |
    """

    def __init__(self, env):

        super(JointActionWrapper, self).__init__(env)
```

```
        self.n_agents = env.n_agents

        self.action_spaces = [list(range(env.action_space[a].n)) for a in range(self.n_agents)]
        self.joint_action_space = list(itertools.product(*self.action_spaces))

        self.action_meanings = [env.get_action_meanings(a) for a in range(self.n_agents)]
        self.joint_action_meanings = list(itertools.product(*self.action_meanings))

        self.n_joint_actions = len(self.joint_action_meanings)

    def reset(self):
        observations = super(JointActionWrapper, self).reset()
        observation = observations[0]   # For the predator-prey domain, the observations are
shared.
        return observation

    def step(self, joint_action: int):

        individual_actions: Sequence[int] = ... # TODO FIXME
        next_observations, rewards, terminals, info = \
                    super(JointActionWrapper, self).step(individual_actions)

        # For the predator-prey domain, the observations are shared.
        next_observation = next_observations[0]
        equal_rewards = all(rewards[0] == reward for reward in rewards)
        assert equal_rewards, "Multi-Agent Learning requires same reward signal for all agents"
        reward = rewards[0]
        terminal = all(terminals)

        return next_observation, reward, terminal, info

    def get_action_meanings(self):
        return self.team_action_meanings
```

After this is complete, go ahead and run the code. You should be able to see a similar result as below:



Answer the following questions:

1. What can you tell from these results? For the Predator-Prey environment with two predators, does it look advantageous to use a centralized learner instead of individual learners? Why?

Bonus programming challenge:

You may have noticed that although the performance of the team of agents approaches the single agent, whereby the plot above shows an overlap of the confidence intervals. Theoretically, it is possible for the single central agent to have a better performance than the team of individual agents. Note that the central agent doesn't need to coordinate, because it explicitly learns the best joint actions. In practice, there are many factors influencing the training performance of the central-single agent but also the task itself that may not need complex coordination strategies. Try to find a setting (by tinkering the number of agents, the number of required captors, the grid size and the maximum number of steps per episode) to show that the centralized multi-agent learner is able to achieve better performance after 10 train-eval iterations. What conclusions can you reach?