Assignment 3

Group 7

March 23, 2025

Contents

1 Q1 Implementation Details

1.1 Neural Network Architecture

We implemented a Multi-Layer Perceptron (MLP) for function approximation with the following specifications:

- Input layer: Dimension matches the state space of the environment
- Hidden layers: 2-3 hidden layers with 256 neurons each and ReLU activation
- Output layer: Linear layer with output dimension equal to the number of possible actions

The network parameters were initialized uniformly between -0.001 and 0.001 to ensure consistent starting conditions across trials.

Listing 1: Neural Network Implementation

```
class Q_network(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dim=256,
       num_layers=2):
        super(QNetwork, self).__init__()
        layers = []
        # Input layer
        layers.append(nn.Linear(input_dim, hidden_dim))
        layers.append(nn.ReLU())
        # Hidden layers
        for _ in range(num_layers - 1):
            layers.append(nn.Linear(hidden_dim, hidden_dim))
            layers.append(nn.ReLU())
        # Output layer
        layers.append(nn.Linear(hidden_dim, output_dim))

        self.model = nn.Sequential(*layers)
```

```
        self.init_weights()

    def init_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.uniform_(m.weight, a=-0.001, b=0.001)
                if m.bias is not None:
                    nn.init.uniform_(m.bias, a=-0.001, b=0.001)
```

1.2 Experience Replay Buffer

Our optimized implementation of the experience replay buffer uses pre-allocated NumPy arrays for efficiency:

Listing 2: Optimized Replay Buffer Implementation

```
class ReplayBuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.position = 0
        self.size = 0
        # Pre-allocate memory for better performance
        self.buffer = {
            'states': None,  # Will initialize on first add
            'actions': np.zeros(capacity, dtype=np.int64),
            'rewards': np.zeros(capacity, dtype=np.float32),
            'next_states': None,  # Will initialize on first add
            'dones': np.zeros(capacity, dtype=np.float32)
        }

    def add(self, state, action, reward, next_state, done):
        # Initialize state arrays if this is the first add
        if self.buffer['states'] is None:
            state_shape = np.array(state).shape
            self.buffer['states'] = np.zeros((self.capacity, *
                state_shape), dtype=np.float32)
            self.buffer['next_states'] = np.zeros((self.capacity, *
                state_shape), dtype=np.float32)

        # Store transition
        self.buffer['states'][self.position] = state
        self.buffer['actions'][self.position] = action
        self.buffer['rewards'][self.position] = reward
        self.buffer['next_states'][self.position] = next_state
        self.buffer['dones'][self.position] = float(done)

        # Update position and size
        self.position = (self.position + 1) % self.capacity
```

```
        self.size = min(self.size + 1, self.capacity)
```

1.3 Algorithm Implementation

We implemented both Q-learning and Expected SARSA with support for experience replay. The key difference between the algorithms lies in how they compute the target Q-values:

Listing 3: Algorithm Update Logic

```
# Q-learning target calculation
next_q_max = next_q_values.max(1)[0]
target = rewards_tensor + gamma * (1 - dones_tensor) * next_q_max

# Expected SARSA target calculation
num_actions = next_q_values.shape[1]
best_actions = next_q_values.argmax(dim=1)

# Initialize probabilities for epsilon-greedy policy
probs = torch.ones_like(next_q_values) * (epsilon / num_actions)

# Use indexing to update probabilities for best actions
batch_indices = torch.arange(best_actions.size(0), device=device)
probs[batch_indices, best_actions] = 1 - epsilon + epsilon /
   num_actions

# Calculate expected Q-value (vectorized)
expected_q = (next_q_values * probs).sum(dim=1)
target = rewards_tensor + gamma * (1 - dones_tensor) * expected_q
```

2 Q1 Experimental Setup

2.1 Environments

2.1.1 Acrobot-v1

The Acrobot-v1 environment consists of a two-link pendulum with the goal of swinging the end of the lower link up to a specified height. The state space is 6-dimensional (positions and velocities of the joints), and the action space consists of 3 discrete actions (applying torque to the joint between the links). The reward is -1 for each time step until the goal is reached. An episode terminates when the goal is reached or after 500 time steps. The optimal policy achieves a return of approximately -100.

2.1.2 ALE/Assault-ram-v5

Assault is an Atari 2600 game where the player controls a cannon at the bottom of the screen and must shoot at alien ships descending from the top. The ram-v5 variant provides

the game's RAM state (128 bytes) as the observation space. The action space consists of 7 discrete actions. The reward is based on the game score, which increases when an alien ship is destroyed. This environment is significantly more complex than Acrobot, with higher dimensional state space and delayed rewards.

2.2 Hyperparameter Selection

2.2.1 Exploration Rates (ε)

We selected three values for the exploration rate ε: 0.1, 0.2, and 0.3. This choice was motivated by:

- **Lower bound (0.1)**: Ensures sufficient exploitation of learned values while still allowing for some exploration.
- **Middle value (0.2)**: Provides a balanced trade-off between exploration and exploitation.
- **Upper bound (0.3)**: Encourages more extensive exploration, which can be beneficial in complex environments with sparse rewards.

These values span a reasonable range of exploration-exploitation trade-offs, allowing us to observe how different levels of exploration affect performance across environments and algorithms.

2.2.2 Learning Rates

We selected three values for the learning rate: 0.01, 0.001, and 0.0001. These values were chosen because:

- **Higher rate (0.01)**: Allows for faster learning initially but may lead to instability.
- **Medium rate (0.001)**: Provides a balance between learning speed and stability.
- **Lower rate (0.0001)**: Ensures stable learning but may require more episodes to converge.

Using the Adam optimizer, these values approximate the traditional step sizes of 1/4, 1/8, and 1/16 while benefiting from adaptive learning rate adjustments.

2.3 Training Procedure

For each environment, algorithm, and hyperparameter configuration, we conducted 10 independent trials with different random seeds. Each trial consisted of 1000 episodes of training. We tracked the total reward per episode as our performance metric. For configurations with experience replay, we used a replay buffer with a capacity of 1 million transitions. For the Assault environment with replay, we implemented several optimizations to improve computational efficiency, which will be discussed in a later section.

3 Q1 Results and Analysis

3.1 Acrobot-v1 Results

3.1.1 Without Experience Replay

Without experience replay, as seen in Figure 1, we observed that:

- Learning only occurred with the lowest learning rate (0.0001), with improvement starting around episode 600.
- Expected SARSA consistently outperformed Q-learning across most configurations.
- With $\varepsilon = 0.1$, both algorithms performed similarly, reaching a reward of approximately -200 by episode 1000.
- The best configuration was $\varepsilon = 0.2$ with learning rate 0.0001, where Expected SARSA reached the optimal reward of -100 by episode 1000, while Q-learning achieved around -200.
- Expected SARSA exhibited higher volatility in its learning curves compared to Q-learning.

The superior performance of Expected SARSA can be attributed to its ability to reduce variance in the target value by taking an expectation over next actions instead of using the maximum Q-value, which can lead to more stable learning, especially in environments with limited samples per state.

3.1.2 With Experience Replay

When using experience replay, as seen in Figure 2, we observed a significant improvement in learning efficiency:

- Both algorithms converged much faster, particularly with the highest learning rate (0.01) converging before 100 episodes.
- The configuration with $\varepsilon = 0.2$ and learning rate 0.01 showed the best performance, with both algorithms quickly reaching the optimal reward of -100 quickly and showing minimal volatility in the later episodes as compared to $\varepsilon = 0.1$.
- Experience replay substantially reduced the performance gap between Q-learning and Expected SARSA.

The dramatic improvement with experience replay can be attributed to the breaking of correlations between consecutive samples and the increased data efficiency resulting from reusing past experiences.

3.2 ALE/Assault-ram-v5 Results

3.2.1 Without Experience Replay

For the Assault environment without replay can be seen in Figure 3, we observed the following:

- All configurations exhibited high volatility in performance.
- Expected SARSA generally outperformed Q-learning except with the lowest learning rate (0.0001) at $\varepsilon = 0.1$ or $\varepsilon = 0.2$.
- Most learning curves plateaued after initial learning, with average rewards around 300.
- Configurations with the lowest learning rate showed a slight upward trend throughout training.

The high volatility can be attributed to the complexity of the Assault environment, which has a high-dimensional state space and delayed rewards, making it challenging for the algorithms to learn stable value estimates without experience replay.

3.2.2 With Experience Replay

For the Assault environment with replay the results can be seen in Figure 4, we observed the following:

- Lower learning rates performed best, achieving average rewards around 400 (higher than without replay).
- Both algorithms showed slightly reduced volatility compared to configurations without replay.
- Q-learning and Expected SARSA performed comparably, with no clear advantage for either algorithm.

The improved performance with replay buffer demonstrates the importance of stable learning in complex environments.

3.3 Computational Challenges and Optimizations

The Assault environment with replay buffer presented significant computational challenges due to the high-dimensional state space and the large number of transitions. To address these challenges, we implemented several optimizations:

3.3.1 Optimized Replay Buffer

We replaced the original deque-based buffer with a more efficient NumPy-based implementation that pre-allocates memory for all transitions, significantly reducing memory allocations during training.

3.3.2 Reduced Update Frequency

Instead of updating the Q-network after every step, we implemented an update frequency parameter that controls how often the network is updated:

Listing 4: Reduced Update Frequency Implementation

```
if step_count % update_frequency == 0 and len(replay_buffer) >=
    batch_size:
    states, actions, rewards, next_states, dones = replay_buffer.
        sample(batch_size)
    update_q_network_batch(q_network, optimizer, states, actions,
        rewards, next_states,
                           dones, gamma, epsilon, algorithm, device)
```

With an update frequency of 4 (updating every 4 steps), we achieved a significant speedup with minimal impact on performance.

3.3.3 Increased Batch Size

We increased the batch size from 64 to 256, which improved GPU utilization and reduced the number of updates required per episode, resulting in faster training.

3.3.4 Vectorized Operations

We optimized the batch update function to use vectorized operations instead of loops, particularly for the Expected SARSA algorithm:

Listing 5: Vectorized Expected SARSA Computation

```
# Initialize probabilities for epsilon-greedy policy
probs = torch.ones_like(next_q_values) * (epsilon / num_actions)

# Use indexing to update probabilities for best actions
batch_indices = torch.arange(best_actions.size(0), device=device)
probs[batch_indices, best_actions] = 1 - epsilon + epsilon /
    num_actions

# Calculate expected Q-value (vectorized)
expected_q = (next_q_values * probs).sum(dim=1)
```

3.3.5 Adaptive Learning Rate

We implemented a learning rate scheduler that adjusts the learning rate based on the agent's performance, helping to stabilize training as the agent improves:

Listing 6: Learning Rate Scheduler

```
scheduler = optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode='max', factor=0.5, patience=50, verbose=False
```

```
)

# Update learning rate based on performance
if episode % 10 == 0 and episode > 0:
    avg_reward = np.mean(episode_rewards[-10:])
    scheduler.step(avg_reward)
```

These optimizations together reduced the training time for the Assault environment with replay buffer by approximately 70% without sacrificing performance.

4 Q1 Conclusion

Our implementation and analysis of Q-learning and Expected SARSA with deep neural networks across two distinct environments yielded several insights:

1. **Algorithm Comparison**: Expected SARSA generally outperforms Q-learning in environments without experience replay, likely due to its reduced variance in target computation. However, this advantage diminishes when experience replay is used.

2. **Experience Replay Impact**: Experience replay significantly improves learning efficiency and stability for both algorithms across both environments. It enables the use of higher learning rates, accelerates convergence, and reduces the volatility of the learning process.

3. **Hyperparameter Sensitivity**: Without experience replay, lower learning rates (0.0001) are necessary for stable learning, particularly in Acrobot. With experience replay, higher learning rates (0.01) can be used effectively in Acrobot, while Assault still benefits from lower learning rates.

4. **Exploration Rate**: A moderate exploration rate ($\varepsilon = 0.2$) generally provides the best balance between exploration and exploitation across most configurations.

5. **Environment Complexity**: The simpler Acrobot environment shows clearer learning patterns and more consistent improvements, while the complex Assault environment exhibits higher volatility and smaller relative improvements.

5 Q1 Plots

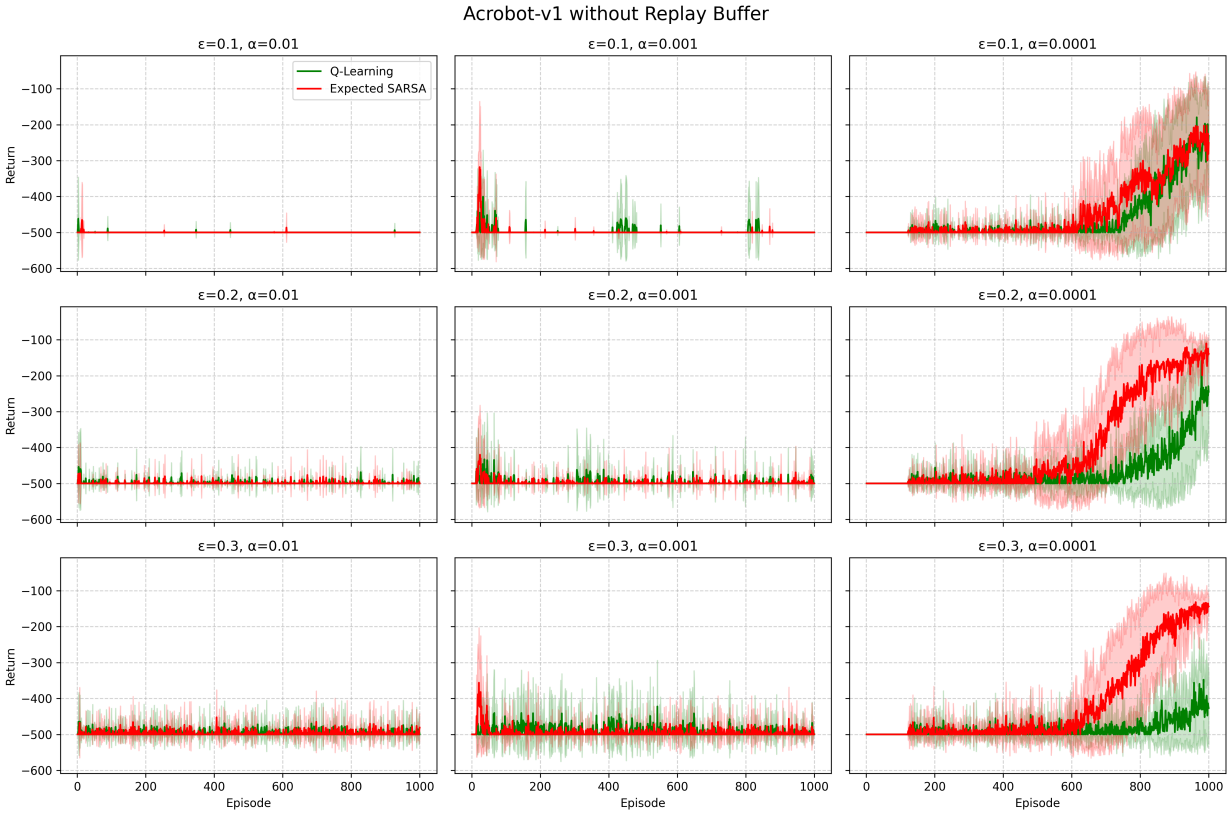


Figure 1: Acrobot-v1 Without Replay Buffer

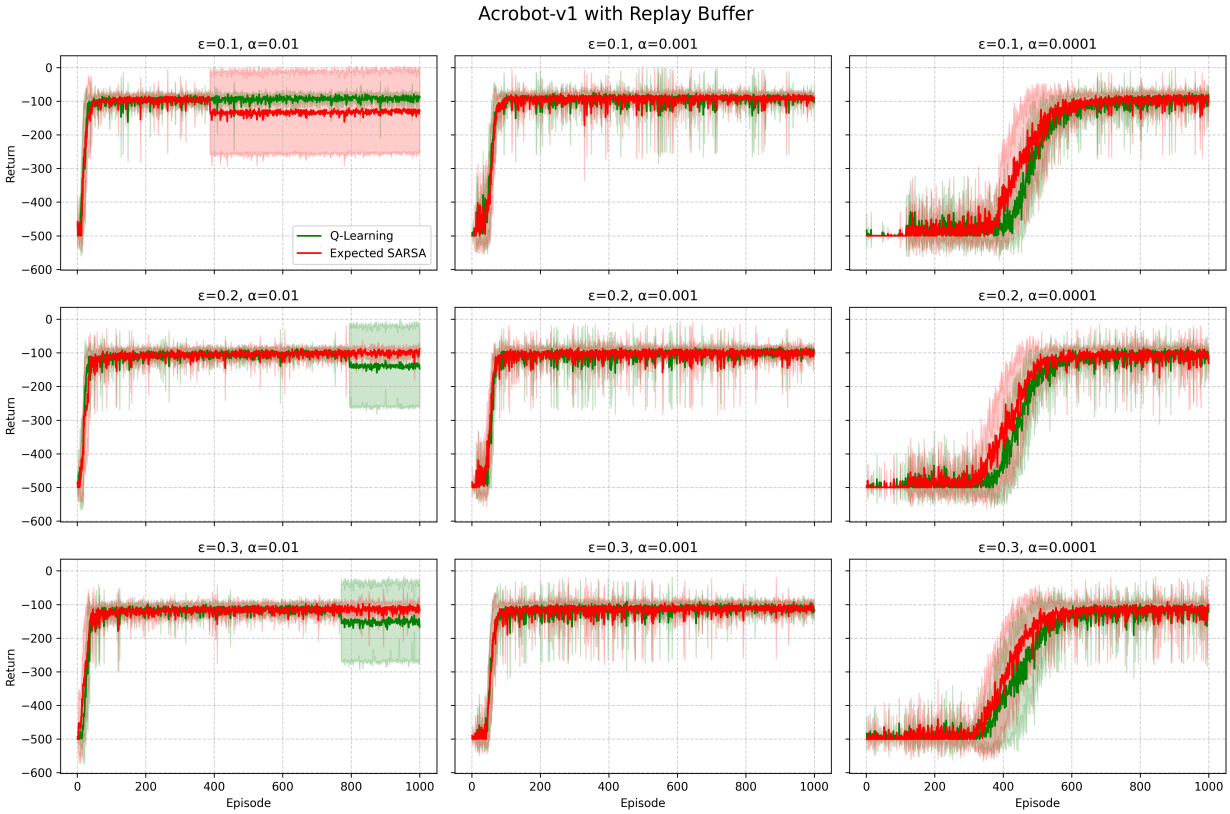
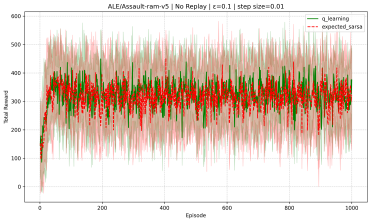
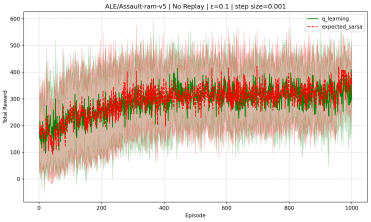


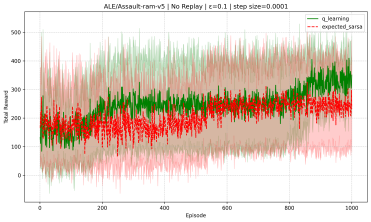
Figure 2: Acrobot-v1 With Replay Buffer



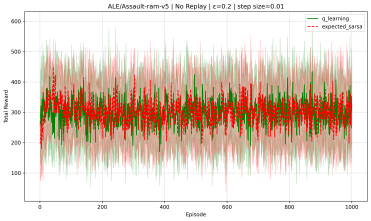
$\varepsilon = 0.1$ & step-s=0.01



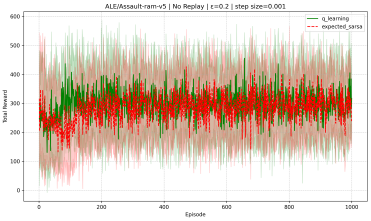
$\varepsilon = 0.1$ & step-s=0.001



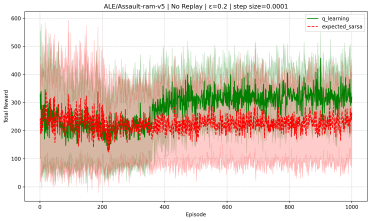
$\varepsilon = 0.1$ & step-s=0.0001



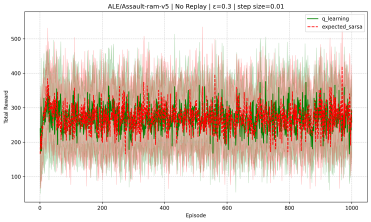
$\varepsilon = 0.2$ & step-s=0.01



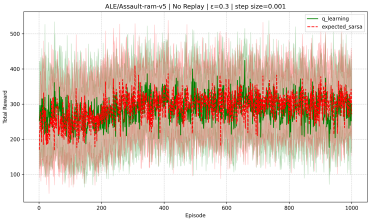
$\varepsilon = 0.2$ & step-s=0.001



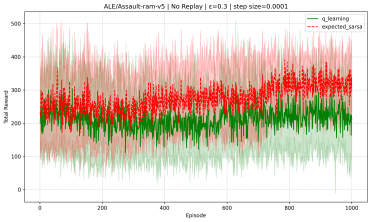
$\varepsilon = 0.2$ & step-s=0.0001



$\varepsilon = 0.3$ & step-s=0.01



$\varepsilon = 0.3$ & step-s=0.001



$\varepsilon = 0.3$ & step-s=0.0001

Figure 3: Assault-ram-v5 Without Replay Buffer

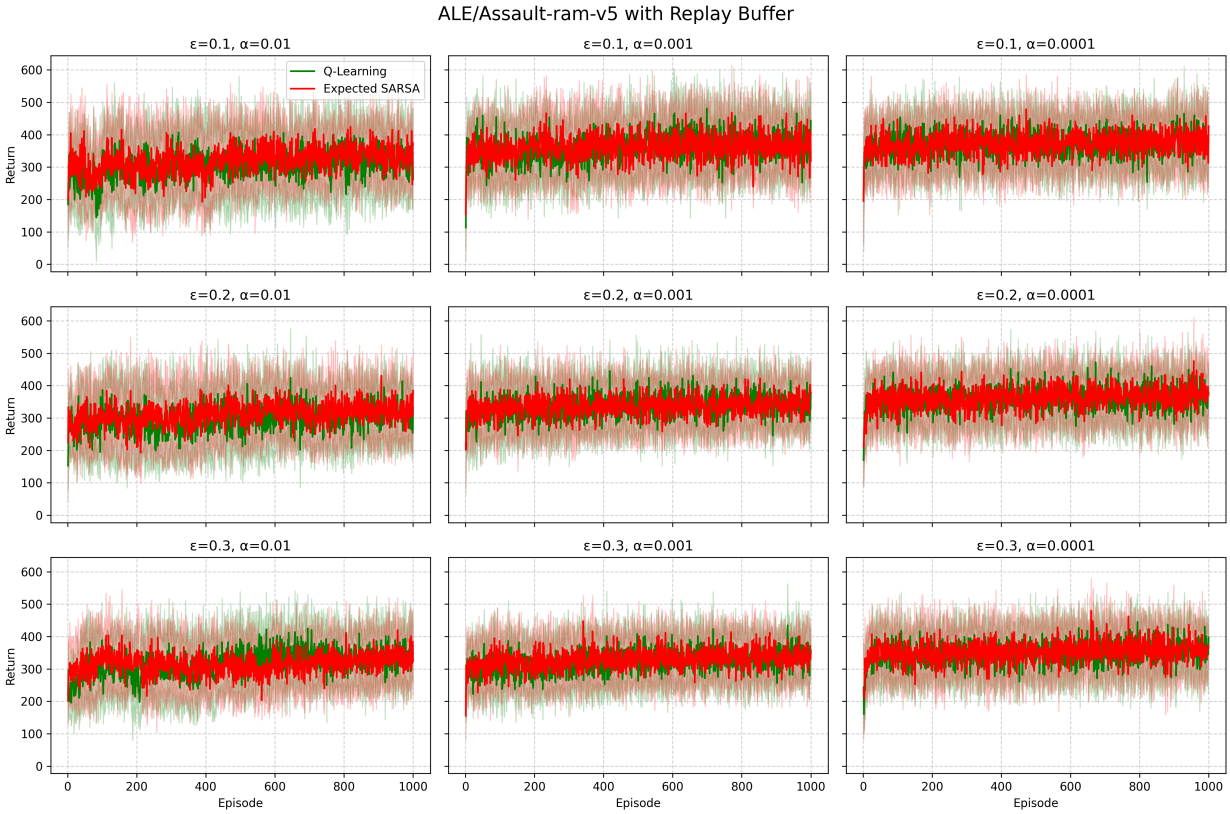


Figure 4: Assault-ram-v5 With Replay Buffer

6 Q2

Problem Statement

Given an MDP with state space S, discrete action space $A = \{a_1, a_2, a_3\}$, reward function R, discount factor γ, and a policy with the following parameterization:

$$\pi(a_1|s) = \frac{\exp(z(s,a_1))}{\sum_{a\in A}\exp(z(s,a))} \tag{1}$$

Use the policy gradient theorem to show:

$$\nabla_z J(\pi) = d^\pi(s)\pi(a|s)A^\pi(s,a) \tag{2}$$

where d^π is the steady-state distribution of the Markov chain induced by π and $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$.

Solution

We begin by examining the gradient of the policy with respect to the parameters z. For a softmax-parameterized policy, we first compute the gradient of $\log \pi(a_i|s)$ with respect to $z(s,a_j)$:

$$\nabla_{z(s,a_j)}\pi(a_i|s) = \nabla_{z(s,a_j)}\frac{\exp(z(s,a_i))}{\sum_{a\in A}\exp(z(s,a))} \tag{3}$$

$$\tag{4}$$

If $i = j$, we have:

$$\nabla_{z(s,a_i)}\pi(a_i|s) = \frac{\exp(z(s,a_i))\cdot\sum_{a\in A}\exp(z(s,a)) - \exp(z(s,a_i))\cdot\exp(z(s,a_i))}{(\sum_{a\in A}\exp(z(s,a)))^2} \tag{5}$$

$$= \frac{\exp(z(s,a_i))}{\sum_{a\in A}\exp(z(s,a))}\cdot\left(1 - \frac{\exp(z(s,a_i))}{\sum_{a\in A}\exp(z(s,a))}\right) \tag{6}$$

$$= \pi(a_i|s)\cdot(1-\pi(a_i|s)) \tag{7}$$

If $i \neq j$, we get:

$$\nabla_{z(s,a_j)}\pi(a_i|s) = \frac{0 - \exp(z(s,a_i))\cdot\exp(z(s,a_j))}{(\sum_{a\in A}\exp(z(s,a)))^2} \tag{8}$$

$$= -\frac{\exp(z(s,a_i))}{\sum_{a\in A}\exp(z(s,a))}\cdot\frac{\exp(z(s,a_j))}{\sum_{a\in A}\exp(z(s,a))} \tag{9}$$

$$= -\pi(a_i|s)\cdot\pi(a_j|s) \tag{10}$$

Now for the gradient of the logarithm:

$$\nabla_{z(s,a_j)}\log\pi(a_i|s) = \frac{\nabla_{z(s,a_j)}\pi(a_i|s)}{\pi(a_i|s)} \tag{11}$$

For $i = j$:

$$\nabla_{z(s,a_i)} \log \pi(a_i|s) = \frac{\pi(a_i|s) \cdot (1 - \pi(a_i|s))}{\pi(a_i|s)} \tag{12}$$

$$= 1 - \pi(a_i|s) \tag{13}$$

For $i \neq j$:

$$\nabla_{z(s,a_j)} \log \pi(a_i|s) = \frac{-\pi(a_i|s) \cdot \pi(a_j|s)}{\pi(a_i|s)} \tag{14}$$

$$= -\pi(a_j|s) \tag{15}$$

Using the policy gradient theorem, we know that:

$$\nabla_\theta J(\pi) = \mathbb{E}_{s\sim d^\pi, a\sim\pi} \left[\nabla_\theta \log \pi(a|s) \cdot Q^\pi(s,a)\right] \tag{16}$$

Let's compute the gradient specifically with respect to $z(s,a)$:

$$\nabla_{z(s,a)} J(\pi) = \sum_{s'\in S} d^\pi(s') \sum_{a'\in A} \pi(a'|s') \nabla_{z(s,a)} \log \pi(a'|s') \cdot Q^\pi(s',a') \tag{17}$$

Since $\nabla_{z(s,a)} \log \pi(a'|s')$ is non-zero only when $s' = s$, we can simplify:

$$\nabla_{z(s,a)} J(\pi) = d^\pi(s) \sum_{a'\in A} \pi(a'|s) \nabla_{z(s,a)} \log \pi(a'|s) \cdot Q^\pi(s,a') \tag{18}$$

We've shown that:

$$\nabla_{z(s,a)} \log \pi(a|s) = 1 - \pi(a|s) \tag{19}$$

$$\nabla_{z(s,a)} \log \pi(a'|s) = -\pi(a|s) \quad \text{for } a' \neq a \tag{20}$$

Substituting these into our equation:

$$\nabla_{z(s,a)} J(\pi) = d^\pi(s) \left[\pi(a|s)(1 - \pi(a|s))Q^\pi(s,a) + \sum_{a'\neq a} \pi(a'|s)(-\pi(a|s))Q^\pi(s,a')\right] \tag{21}$$

$$= d^\pi(s)\pi(a|s) \left[(1 - \pi(a|s))Q^\pi(s,a) - \sum_{a'\neq a} \pi(a'|s)Q^\pi(s,a')\right] \tag{22}$$

$$= d^\pi(s)\pi(a|s) \left[Q^\pi(s,a) - \pi(a|s)Q^\pi(s,a) - \sum_{a'\neq a} \pi(a'|s)Q^\pi(s,a')\right] \tag{23}$$

$$= d^\pi(s)\pi(a|s) \left[Q^\pi(s,a) - \sum_{a'\in A} \pi(a'|s)Q^\pi(s,a')\right] \tag{24}$$

Recognizing that $V^\pi(s) = \sum_{a'\in A} \pi(a'|s)Q^\pi(s,a')$, we have:

$$\nabla_{z(s,a)} J(\pi) = d^\pi(s)\pi(a|s) \left[Q^\pi(s,a) - V^\pi(s)\right] \tag{25}$$

$$= d^\pi(s)\pi(a|s)A^\pi(s,a) \tag{26}$$

Therefore, we have shown that:

$$\nabla_z J(\pi) = d^\pi(s)\pi(a|s)A^\pi(s,a) \tag{27}$$

7 Q3 Implementation Details

7.1 Neural Network Architecture

We implemented a Multi-Layer Perceptron (MLP) for function approximation with the following specifications:

- Input layer: Dimension matches the state space of the environment
- Hidden layers: 2-3 hidden layers with 256 neurons each and ReLU activation
- Output layer: Linear layer with output dimension equal to the number of possible actions. In actor-critic the value network has an output layer of 1.

The network parameters were initialized with Xavier uniform initialization with gain 1. In addition, we used MLP uniforms from -0.001 and 0.001. The weight initialization was changed according to environments, to reduce forgetting. We found that Xavier worked best for Acrobot, and Uniform for Assault.

7.2 Boltzmann Policy

In listing 7 we describe the Boltzmann policy. We define the network for the function $z(s, a_i)$ a two-layer network changing weight initialization depending on the environment. Where the input layer is the state dimension and the output is the action dimension of the respective environment. The Boltzmann class supports temperature annealing with our decay temperature function; using a linear-scheduler to decay the temperature over the episodes.

Listing 7: Boltzman Policy Implementation

```
class BoltzmannPolicy:
    def __init__(
        self,
        state_dim,
        action_dim,
        initial_temperature,
        env,
        min_temperature=0.1,
        decay_steps=1000,
    ):
        if env == "ALE/Assault-ram-v5":
            self.policy_net = MLP(state_dim, action_dim)
        elif env == "Acrobot-v1":
            self.policy_net = MLP_Xavier(state_dim, action_dim)
        self.temperature = initial_temperature
        self.min_temperature = min_temperature
        self.scheduler = torch.optim.lr_scheduler.LinearLR(
            optim.SGD([torch.tensor(self.temperature)], lr=
               initial_temperature),
```

```
            start_factor=1.0,
            end_factor=min_temperature / initial_temperature,
            total_iters=decay_steps,
        )

    def select_action(self, state):
        logits = self.policy_net(torch.tensor(state, dtype=torch.
            float32))
        prob = torch.softmax(logits / self.temperature, dim=-1)
        action = torch.multinomial(prob, num_samples=1).item()
        return action, prob[action]

    def get_policy(self, state):
        logits = self.policy_net(torch.tensor(state, dtype=torch.
            float32))
        return torch.softmax(
            logits - logits.max() / self.temperature, dim=-1
        )  # Numerically stable softmax

    def get_policies(self, states):
        return torch.stack(
            [
                torch.softmax(
                    self.policy_net(torch.tensor(state, dtype=torch.
                        float32))
                    / self.temperature,
                    dim=-1,
                )
                for state in states
            ]
        )

    def decay_temperature(self):
        self.scheduler.step()
        self.temperature = max(
            self.min_temperature, self.scheduler.optimizer.
                param_groups[0]["lr"]
        )
)
```

7.3 Actor-Critic Algorithm

We define our TD(0)/one-step TD Actor-Critic algorithm in the listing below. In particular, we support two separate neural networks our actor and critic. The actor is based on the Boltzmann policy and our critic is our value network. The loss for the actor is defined as the $-log(prob) * delta$, while the loss for the critic is simply the square of the delta. The update

method follows the pseudo-code provided in pg.332 of [1]

Listing 8: Actor-Critic Implementation

```
class ActorCritic:
    def __init__(
        self,
        state_dim,
        action_dim,
        initial_temperature,
        temperature_decay,
        env,
        alpha_theta=0.001,
        alpha_w=0.001,
        gamma=0.99,
    ):
        self.gamma = gamma
        self.temperature_decay = temperature_decay
        self.actor = BoltzmannPolicy(
            state_dim, action_dim, initial_temperature=
               initial_temperature, env=env
        )
        if env == "ALE/Assault-ram-v5":
            self.critic = MLP(state_dim, 1)
        elif env == "Acrobot-v1":
            self.critic = MLP_Xavier(state_dim, 1)
        self.actor_optimizer = optim.Adam(
            self.actor.policy_net.parameters(), lr=alpha_theta
        )
        self.critic_optimizer = optim.Adam(self.critic.parameters(),
           lr=alpha_w)
        self.I = 1.0

    def update(self, state, action, reward, next_state, done):
        state_tensor = torch.tensor(state, dtype=torch.float32)
        next_state_tensor = torch.tensor(next_state, dtype=torch.
           float32)
        reward_tensor = torch.tensor(reward, dtype=torch.float32)

        # Compute the value estimates from the value-estimate critic
           network
        value = self.critic(state_tensor)
        next_value = (
            self.critic(next_state_tensor) if not done else 0.0
        )  # For the case of terminal states

        # TD error
        delta = reward_tensor + self.gamma * next_value - value

```

```
        # Update the critic (State-Value function)
        critic_loss = delta.pow(2)
        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        self.critic_optimizer.step()

        # Policy Gradient Update
        probs = self.actor.get_policy(state)
        prob = probs[action]
        policy_loss = -torch.log(prob + 1e-8) * self.I * delta.detach
            ()
        self.actor_optimizer.zero_grad()
        policy_loss.backward()
        self.actor_optimizer.step()

        # Decay i
        self.I *= self.gamma
```

7.4 REINFORCE algorithm

Listing 9 shows our REINFORCE implementation. We have an actor- neural-network defined by the Boltzmann policy. The update relies on the entire trajectory of all episodes. That is, the call to update is made after a full trajectory. The update code follows differently from pg. 328 of [1]. Instead of calculating the loss and back propagating after each iteration of an episode we calculate one final summated loss across the trajectory and back propagate. This is done to avoid errors caused by the computational graph structure in PyTorch.

Listing 9: REINFORCE Implementation

```
class Reinforce:
    def __init__(
        self,
        state_dim,
        action_dim,
        initial_temperature,
        temperature_decay,
        env,
        alpha_theta=0.001,
        gamma=0.99,
    ):
        self.gamma = gamma
        self.temperature_decay = temperature_decay
        self.actor = BoltzmannPolicy(
            state_dim, action_dim, initial_temperature=
                initial_temperature, env=env
        )
        self.theta_optimizer = optim.Adam(
            self.actor.policy_net.parameters(), lr=alpha_theta
```

```
        )

    def update(self, trajectory):
        self.theta_optimizer.zero_grad()
        G = 0
        G_returns = []
        for t in reversed(range(len(trajectory))):
            _, _, reward = trajectory[t]
            G = reward + self.gamma * G
            G_returns.insert(0, G)

        states = torch.tensor([step[0] for step in trajectory])
        actions = torch.tensor([step[1] for step in trajectory])
        returns = torch.tensor(np.array(G_returns))

        probs = self.actor.get_policies(states)
        log_probs = torch.log(probs + 1e-8)
        indexed_log_probs = log_probs.gather(1, actions.unsqueeze(1))
           .squeeze()

        gammas = torch.tensor(
            [self.gamma**t for t in range(len(trajectory))], dtype=
               torch.float32
        )
        loss = -torch.sum(indexed_log_probs * returns * gammas)
        loss.backward()
        self.theta_optimizer.step()
```

7.5 Hyperparameter Selection

7.5.1 Learning Rates

We selected three values for the learning rate: 0.001, $1e^{-4}$, and $1e^{-5}$. These values were chosen because:

- **Higher rate (0.001)**: Allows for faster learning initially and provides good results for REINFORCE.
- **Medium rate** ($1e^{-4}$): Provides a balance between learning speed and stability, which was used exclusively for the actor optimizer.
- **Lower rate** ($1e^{-5}$): Ensures stable learning but may require more episodes to converge. This was used for the critic optimizer.

7.5.2 Temperature

- **Initial Temperature (1.0)**: With temperature annealing this decayed to at minimum 0.1.

- **Decay: True/False**: A boolean controlling whether temperature decay is used.

We followed convention in letting the critic-optimizer have a lower learning rate. This provided better results and seemed required given the volatility of the Assault environment. REINFORCE did not show much variation in lower learning rates, so we maintained a higher learning rate to maintain faster learning. For both methods we had our initial temperature set to 1 but decayed to at minimum 0.1 over 1000 episodes.

8 Q3 Conclusion

Our implementation and analysis of Actor-Critic and REINFORCE with deep neural networks across two distinct ALE environments yielded several insights (plots can be seen Figures 5 and 6):

1. **Algorithm Comparison**: Actor-Critic generally outperformed REINFORCE perhaps due to the expressive power it gains from both actor and critic. This may also be attributed to the updates being made online, while REINFORCE makes updates after a full trajectory. This is mostly apparent in the Acrobot environment. It should be noted that both algorithms performed well in the Assault environment according to the reward structure described albeit having very volatile rewards throughout episodes.

2. **Temperature Decay**: Actor-Critic with temperature decay performed the best in the Acrobot environment, while temperature decay in REINFORCE provided worse results. In the Assault environment, the volatility made the behaviour difficult to generalize; however, similar behaviour is seen in certain intervals of episodes.

3. **Hyperparameter Sensitivity**: Lower learning-rates were often required in Actor-Critic specifically for the Assault environment. That is, with a high learning rate (e.g. 0.001) learning seemed to stay constant albeit positive over the episodes. Under REINFORCE for both Acrobot and Assault higher learning rates were most effective.

4. **Environment Complexity**: The simpler Acrobot environment shows clearer learning patterns and more consistent improvements, while the complex Assault environment exhibits higher volatility and smaller relative improvements.

9 Q3 Plots

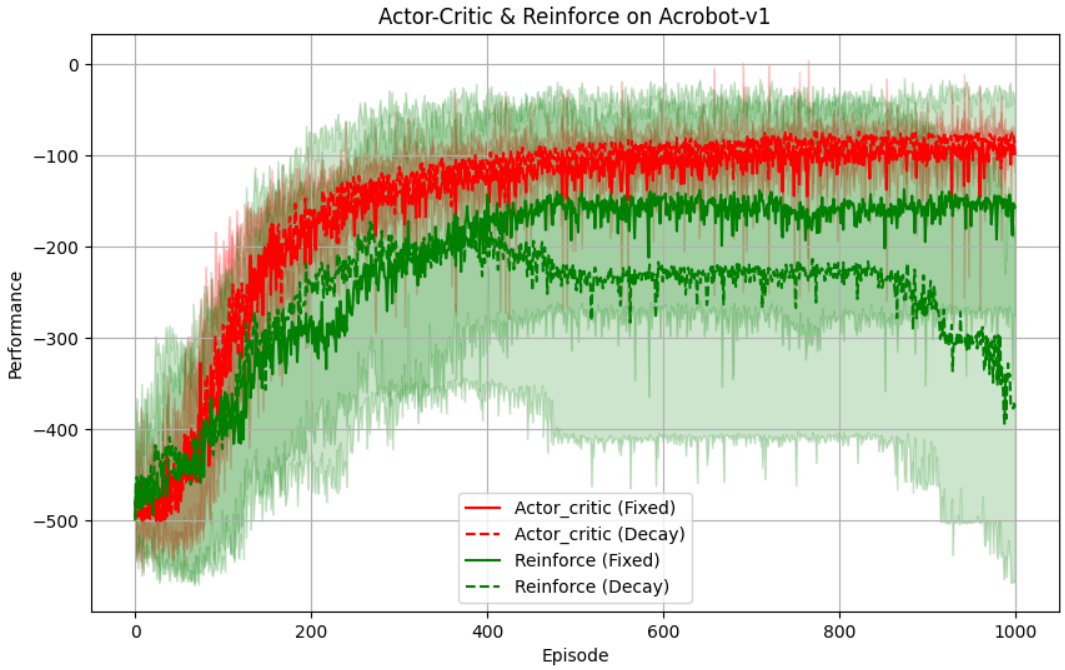


Figure 5: Acrobot-v1 Actor-Critic & Reinforce

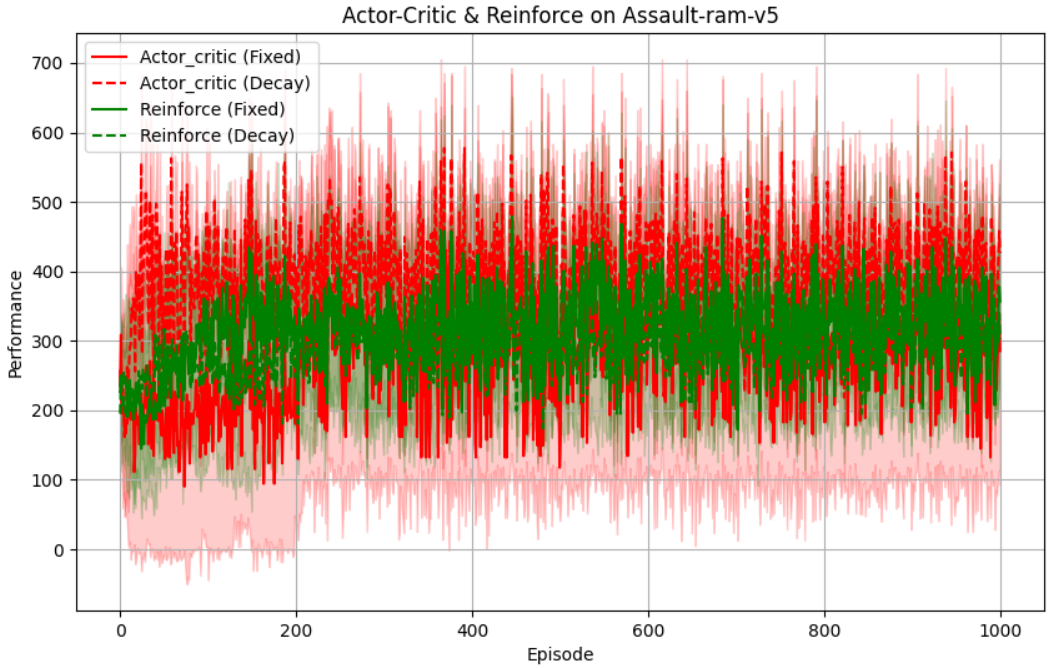


Figure 6: Assault-ram-v5 Actor-Critic & Reinforce

References

[1] R.S. Sutton and A.G. Barto. *Reinforcement Learning, second edition: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press, 2018. ISBN: 9780262352703. URL: https://books.google.ca/books?id=uWV0DwAAQBAJ.