DA6400 Reinforcement Learning PA1

Shuvrajeet Das[1] and Rajshekhar Rakshit[2]

[1] DA24D402, Indian Institute of Technology, Madras, DSAI Dept
da24d402@smail.iitm.ac.in,shuvrajeet17@gmail.com
[2] CS24S031, Indian Institute of Technology, Madras, CSE Dept
cs24s031@smail.iitm.ac.in,rajshekharrakshit123@gmail.com

Abstract. This report presents an empirical evaluation of Reinforcement Learning (RL) algorithms in Gymnasium environments, focusing on SARSA and Q-Learning. The selected environments—CartPole-v1, MountainCar-v0, and MiniGrid-Dynamic-Obstacles-5x5-v0—pose varying degrees of complexity in state-space representation and control challenges. The study explores the impact of different hyperparameters and exploration strategies, specifically ϵ-greedy for SARSA and Softmax exploration for Q-Learning. Performance is assessed based on episodic returns, with results averaged over five random seeds to mitigate stochastic variations. Comparative analyses are conducted using statistical measures, and the top three hyperparameter configurations for each environment are identified. The report provides insights into learning stability, convergence behavior, and the trade-offs between on-policy and off-policy learning approaches. Github Link

Keywords: Reinforcement Learning · SARSA · Q-Learning · Gymnasium Environments · Exploration Strategies · Hyperparameter Tuning.

1 Environments

In this programming task, you'll utilize the following Gymnasium environments for training and evaluating your policies. The links associated with the environments contain descriptions of each environment. Please use the exact version of the environment as specified:

1.1 Cartpole[1]

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.

1.2 Mountain Car[2]

The Mountain Car MDP is a deterministic MDP that consists of a car placed stochastically at the bottom of a sinusoidal valley, with the only possible actions

being the accelerations that can be applied to the car in either direction. The goal of the MDP is to accelerate the car to reach the goal state on top of the right hill. There are two versions of the mountain car domain in the gymnasium: one with discrete actions and one with continuous actions. This version is the one with discrete actions.

1.3 MiniGrid-Dynamic-Obstacles-5x5-v0

This environment is an empty room with moving obstacles. The goal of the agent is to reach the green goal square without colliding with any obstacle. A large penalty is subtracted if the agent collides with an obstacle and the episode finishes. This environment is useful to test Dynamic Obstacle Avoidance for mobile robots with Reinforcement Learning in Partial Observability.

2 Algorithm

2.1 Q-Learning Algorithm

Q-Learning is an off-policy reinforcement learning algorithm that learns the optimal action-value function using the Bellman equation. The algorithm iteratively updates the Q-values based on the observed rewards and the maximum estimated future rewards. The update rule is given by:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \tag{1}$$

where α is the learning rate, γ is the discount factor, r is the received reward, and s' is the next state.

Algorithm 1 Q-Learning Algorithm

Initialize $Q(s,a)$ arbitrarily for all states and actions
for each episode **do**
    Initialize state s
    for each step in the episode **do**
        Choose action a using an exploration policy (e.g., ϵ-greedy)
        Take action a, observe reward r and next state s'
        Update Q-value using:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \tag{2}$$

        Set $s \leftarrow s'$
    end for
end for

2.2 SARSA Algorithm

SARSA (State-Action-Reward-State-Action) is an on-policy reinforcement learning algorithm that updates the Q-values based on the action actually taken in the next state. The update rule is given by:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a) \right] \tag{3}$$

where a' is the action chosen in state s' following the current policy.

Algorithm 2 SARSA Algorithm

Initialize $Q(s,a)$ arbitrarily for all states and actions
for each episode **do**
  Initialize state s
  Choose action a using an exploration policy (e.g., ϵ-greedy)
  for each step in the episode **do**
    Take action a, observe reward r and next state s'
    Choose action a' using the current policy
    Update Q-value using:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a) \right] \tag{4}$$

    Set $s \leftarrow s'$, $a \leftarrow a'$
  end for
end for

3 Result Plots

The section contains the comparison plot of SARSA and Q-Learning on CartPole-V1 environment and MountainCar-v0 environment respectively. The plot contains the average of 5 runs to deal with the stochasticity.

3.1 CartPole Environment Comparison Plot

The comparison plot of the CartPole environment illustrates the performance of different reinforcement learning algorithm based on episodic returns. Each experiment is conducted over five independent runs with distinct random seeds to ensure robustness and reduce stochastic variability. The graph presents the episodic return as a function of the episode number, allowing for a clear visualization of learning progress over time. To enhance interpretability, the plotted curves represent the mean episodic return across the five runs, while the shaded regions indicate the variance, capturing the uncertainty and stability of each policy's learning process.

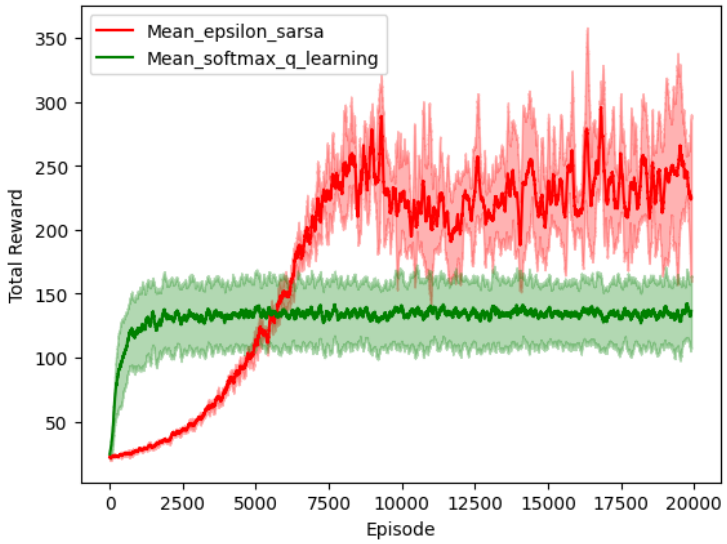

Fig. 1. Comparison plot of the CartPole-v1 environment using different policies for 5 runs

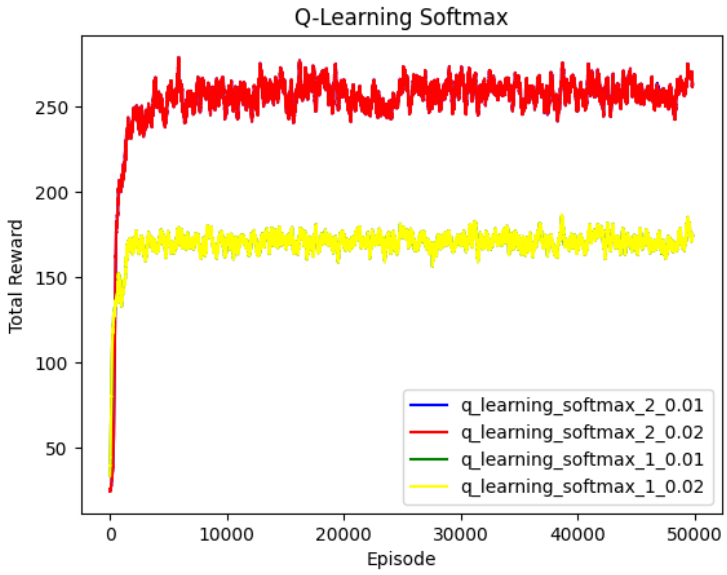

Fig. 2. Q-Learning softmax policy plot for different hyper-parameter for CartPole-V1

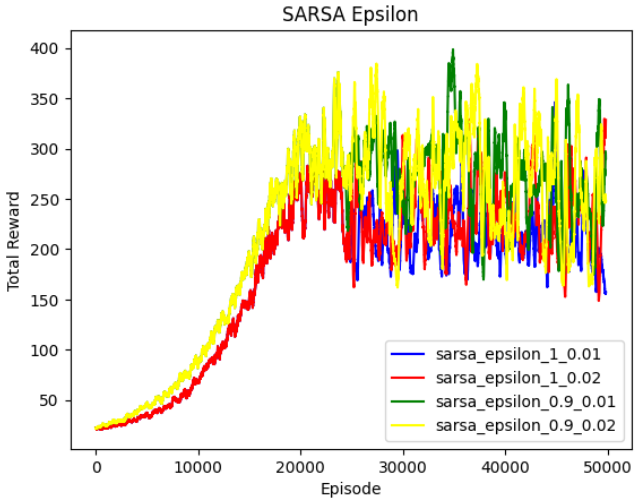

Fig. 3. Sarsa Epsilon-greedy policy plot for different hyper-parameter for CartPole-V1

3.2 Mountain-Car Environment Comparison Plot

The comparison plot of the MountainCar-v0 environment illustrates the performance of different reinforcement learning algorithms based on episodic returns. Each experiment is conducted over five independent runs with distinct random seeds to ensure robustness and reduce stochastic variability.

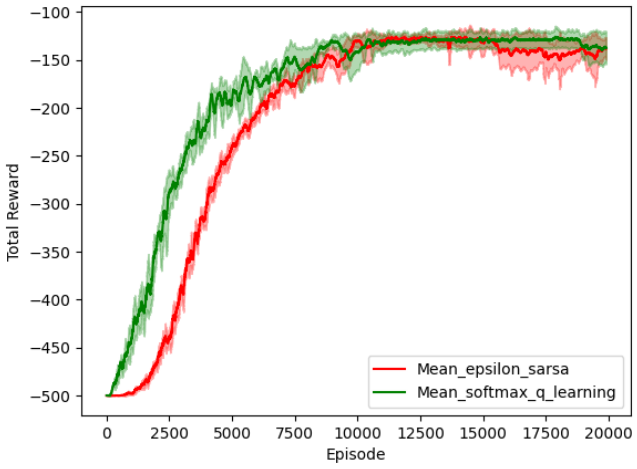

Fig. 4. Comparison plot of the MountainCar-v0 environment using different policies.

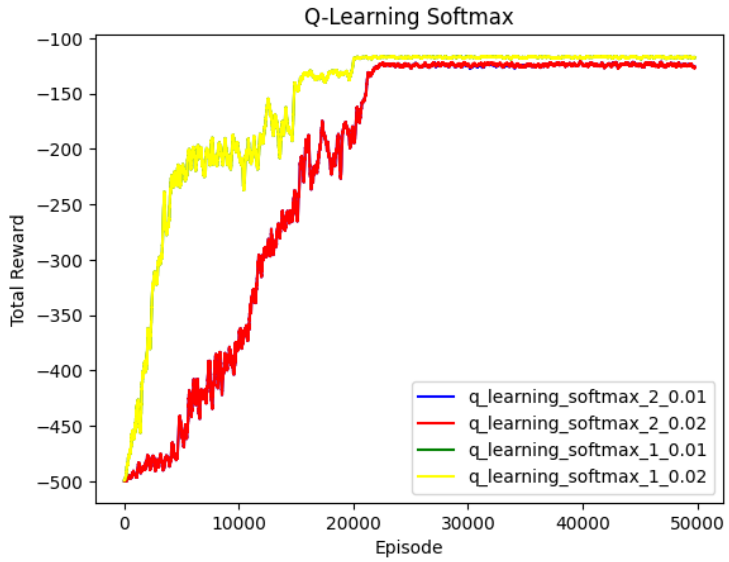

Fig. 5. Q-Learning softmax policy plot for different hyper-parameter for MountainCar-V0

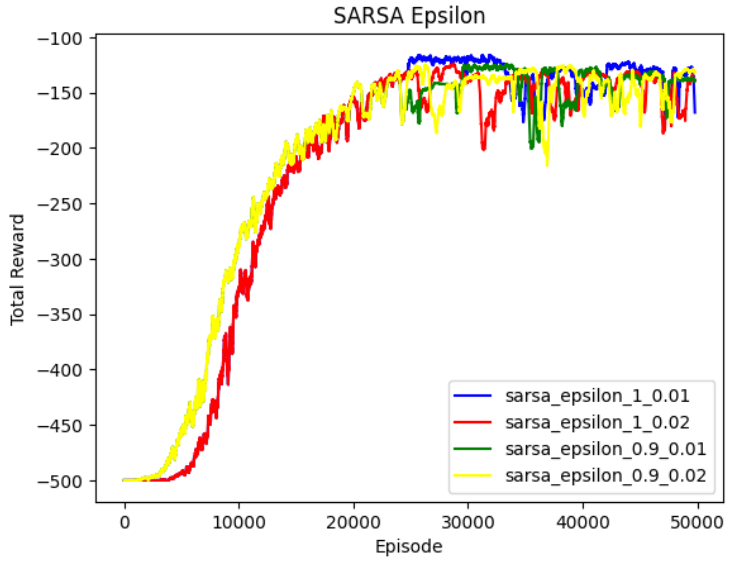

Fig. 6. Sarsa Epsilon-greedy policy plot for different hyper-parameter MountainCar-V0

4 Top 3 Hyperparameter

4.1 CartPole-V1 Environment

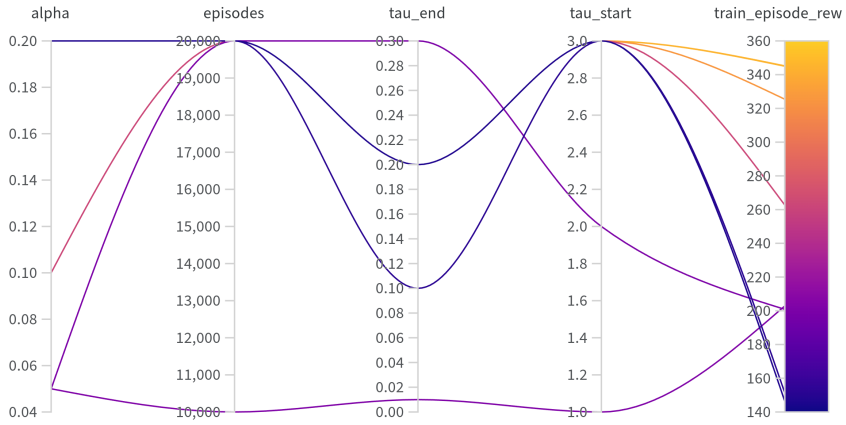

Fig. 7. Q-Learning with Softmax policy hyperparameter plot

Q-Learning with Softmax Policy

1. $\alpha = 0.05$, $\tau_{start} = 3$, $\tau_{end} = 0.2$
2. $\alpha = 0.5$, $\tau_{start} = 3$, $\tau_{end} = 0.1$
3. $\alpha = 0.1$, $\tau_{start} = 3$, $\tau_{end} = 0.2$

Description of the Plot The plot in Figure 7 visualizes the impact of different hyperparameters on the performance of a Q-learning agent employing a Softmax policy. The x-axis represents the number of **training episodes**, while the y-axis depicts different parameters and performance metrics:

- **Lower Learning Rate** ($\alpha = 0.05$): Leads to slower, more stable learning. Combined with $\tau_{start} = 3$ and $\tau_{end} = 0.2$, it ensures a gradual transition from exploration to exploitation.
- **Higher Learning Rate** ($\alpha = 0.5$): Accelerates learning but may lead to instability. A final temperature of $\tau_{end} = 0.1$ forces stronger exploitation at the end.
- **Moderate Learning Rate** ($\alpha = 0.1$): Provides a balanced trade-off between stability and adaptability, with an exploration-exploitation schedule similar to case 1.

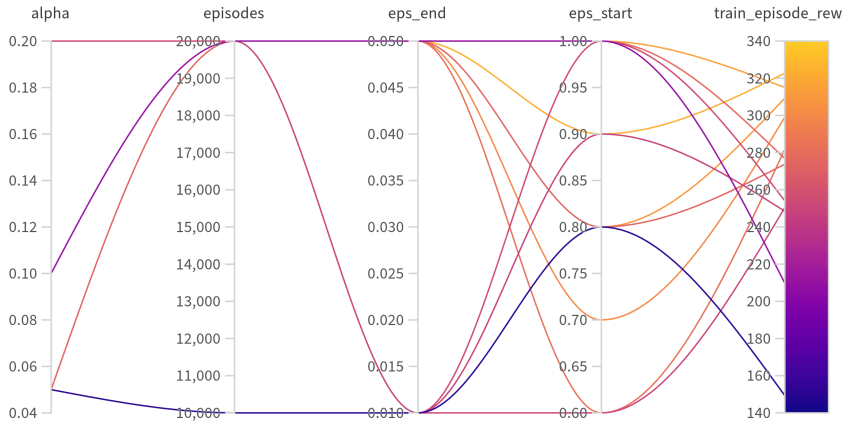

Fig. 8. Sarsa with Epsilon-greedy policy hyperparameter plot

Sarsa with Epsilon-Greedy Policy

1. $\alpha = 0.2$, $\epsilon_{start} = 0.9$, $\epsilon_{end} = 0.05$
2. $\alpha = 0.2$, $\epsilon_{start} = 1$, $\epsilon_{end} = 0.05$
3. $\alpha = 0.2$, $\epsilon_{start} = 0.8$, $\epsilon_{end} = 0.05$

Description of the Plot

The plot in Figure 8 visualizes the impact of different hyperparameters on the performance of a Q-learning agent employing a Softmax policy. The x-axis represents the number of **training episodes**, while the y-axis depicts different parameters and performance metrics:

- **Exploration-Exploitation Trade-off**: The choice of ϵ_{start} and ϵ_{end} affects how the agent balances exploration and exploitation over time.
- **High Initial Exploration** ($\epsilon_{start} = 1$): Ensures maximum exploration at the beginning, helping the agent discover better policies but might slow down convergence.
- **Moderate Initial Exploration** ($\epsilon_{start} = 0.9$): Provides a slightly reduced exploration rate, balancing learning stability and adaptability.
- **Lower Initial Exploration** ($\epsilon_{start} = 0.8$): Starts with less exploration, which may help in faster exploitation but risks premature convergence.

4.2 MountainCar-V0 Environment

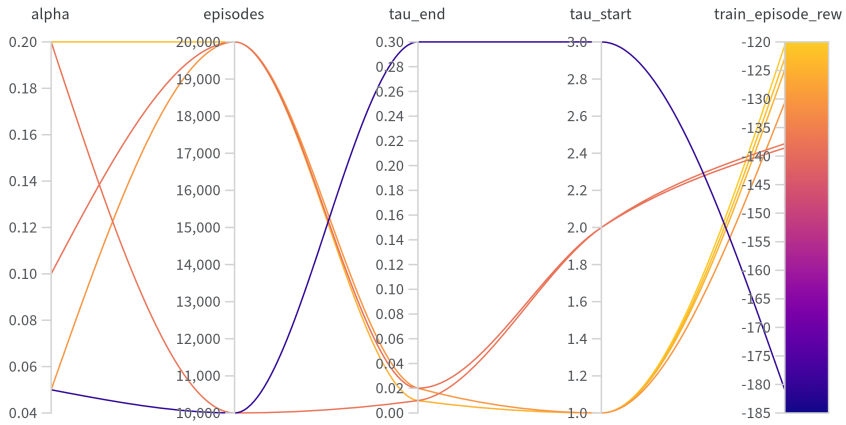

Fig. 9. Q-Learning with softmax policy hyperparameter plot

Q-Learning with Softmax Policy

1. $\alpha = 0.1$, $\tau_{start} = 1$, $\tau_{end} = 0.02$
2. $\alpha = 0.2$, $\tau_{start} = 1$, $\tau_{end} = 0.02$
3. $\alpha = 0.1$, $\tau_{start} = 1$, $\tau_{end} = 0.01$

Description of the Plot

The plot in Figure 9 visualizes the impact of different hyperparameters on the performance of a Q-learning agent employing a Softmax policy. The x-axis represents the number of **training episodes**, while the y-axis depicts different parameters and performance metrics:

- **Higher Learning Rate** ($\alpha = 0.2$): Accelerates learning but may lead to instability. A final temperature of $\tau_{end} = 0.02$ forces stronger exploitation at the end.
- **Lower Learning Rate** ($\alpha = 0.1$): Leads to slower, more stable learning. A lower τ_{end} ensures gradual transition from exploration to exploitation.
- **Lower Final Temperature** ($\tau_{end} = 0.01$): Encourages stronger exploitation towards the end of training, which may help in convergence but reduce adaptability.

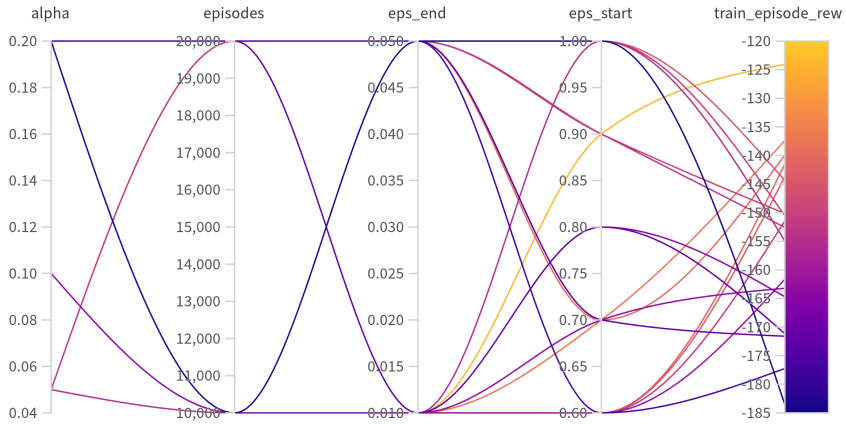

Fig. 10. Sarsa with Epsilon-greedy policy hyperparameter plot

Sarsa with Epsilon-Greedy Policy

1. $\alpha = 0.1$, $\epsilon_{start} = 0.9$, $\epsilon_{end} = 0.01$
2. $\alpha = 0.05$, $\epsilon_{start} = 0.7$, $\epsilon_{end} = 0.01$
3. $\alpha = 0.05$, $\epsilon_{start} = 0.7$, $\epsilon_{end} = 0.05$

Description of the Plot

The plot in Figure 10 visualizes the impact of different hyperparameters on the performance of a Q-learning agent employing a Softmax policy. The x-axis represents the number of **training episodes**, while the y-axis depicts different parameters and performance metrics:

- **Higher Learning Rate** ($\alpha = 0.1$): Encourages faster learning but may introduce instability. A high initial $\epsilon_{start} = 0.9$ promotes exploration, while a low $\epsilon_{end} = 0.01$ enforces exploitation at later stages.
- **Lower Learning Rate** ($\alpha = 0.05$): Ensures more stable learning. With $\epsilon_{start} = 0.7$ and $\epsilon_{end} = 0.01$, the agent transitions smoothly from exploration to exploitation.
- **Moderate Exploration Decay** ($\epsilon_{end} = 0.05$): A slightly higher final exploration rate retains some exploratory behavior, potentially preventing premature convergence.

5 Important Part of Code

5.1 About Jax:

JAX is an open-source numerical computing library that combines NumPy-like APIs with automatic differentiation (Autograd) and GPU/TPU acceleration. It provides:

- **Automatic Differentiation**: Forward-mode and reverse-mode differentiation using `jax.grad`.
- **Just-In-Time (JIT) Compilation**: Accelerates code execution using XLA with `jax.jit`.
- **Vectorization**: Efficient batch computations via `jax.vmap`.
- **Parallelism**: Multi-device and distributed computing using `jax.pmap`.
- **NumPy Compatibility**: Seamless transition from NumPy to JAX with `jax.numpy`.

JAX enables researchers to write high-performance machine learning models while leveraging modern hardware acceleration.

5.2 About Numpy:

NumPy (Numerical Python) is a core library for numerical computing in Python. It provides:

- **Multidimensional Arrays**: Efficient handling of large datasets using `ndarray`.
- **Broadcasting**: Enables element-wise operations on arrays of different shapes.
- **Linear Algebra**: Supports matrix operations, eigenvalues, and singular value decomposition.
- **Random Sampling**: Includes a variety of probability distributions via `numpy.random`.
- **Fourier Transforms**: Fast computation of Discrete Fourier Transforms using `numpy.fft`.
- **Integration with C and Fortran**: Optimized performance with low-level language interoperability.

NumPy serves as the foundation for many scientific computing and machine learning libraries, making it a crucial tool for numerical analysis and data processing.

```
@functools.partial(jax.jit, static_argnums=(0,))
def update(self, carry, transition):
    Q = carry
    state, action, reward, next_state, done = transition
    q_current = Q[tuple(state) + (action,)]
    q_next = jnp.max(Q[tuple(next_state)])
    target = reward + (1 - done) * self.gamma * q_next
    updated_q = q_current + self.alpha * (target - q_current)
    Q = Q.at[tuple(state) + (action,)].set(updated_q)
    return Q
```

Listing 1.1. Q-Learning update

```
@functools.partial(jax.jit, static_argnums=(0,))
def act(self, carry, state, tau):
    rng, Q = carry
    q_values = Q[tuple(state)] / tau
    probabilities = jax.nn.softmax(q_values)
    rng, subkey = jax.random.split(rng)
    action = jax.random.categorical(subkey, jnp.log(probabilities))
    return (rng, Q), action
```

Listing 1.2. Softmax Policy

```
@functools.partial(jax.jit, static_argnums=(0,))
def update(self, carry, transition):
    Q = carry
    state, action, reward, next_state, next_action, done = transition
    q_current = Q[tuple(state) + (action,)]
    q_next = Q[tuple(next_state) + (next_action,)]
    target = reward + (1 - done) * self.gamma * q_next
    updated_q = q_current + self.alpha * (target - q_current)
    Q = Q.at[tuple(state) + (action,)].set(updated_q)
    return Q
```

Listing 1.3. Sarsa update

```
@functools.partial(jax.jit, static_argnums=(0,))
def act(self, carry, state, epsilon):
    rng, Q = carry
    q_values = Q[tuple(state)]

    rng, subkey = jax.random.split(rng)
    explore = jax.random.uniform(subkey) < epsilon
    @jax.jit
    def random_action():
        return jax.random.randint(subkey, (), 0, q_values.shape[0])
```

```
@jax.jit
def greedy_action():
    return jnp.argmax(q_values)

action = jax.lax.cond(explore, random_action, greedy_action)
return (rng, Q), action
```

Listing 1.4. ϵ-greedy policy

References

1. Barto, A.G., Sutton, R.S., Anderson, C.W.: Neuronlike adaptive elements that can solve difficult learning control problems. IEEE Transactions on Systems, Man, and Cybernetics **SMC-13**(5), 834–846 (1983). https://doi.org/10.1109/TSMC.1983.6313077
2. Moore, A.W.: Efficient memory-based learning for robot control. Tech. rep., University of Cambridge (1990)