CS687 Final

Vin Tyagi
University of Massachusetts
vtyagi@umass.edu

Aarav Nair
University of Massachusetts
aanair@umass.edu

Sam O'Naullain
University of Massachusetts
snaullain@umass.edu

December 2025

Introduction

Background and Pseudo-code

We will be choosing Option 2 for the final project. Each group member has designed an advanced RL algorithm with two domains in mind, and will provide learning curves and detailed analysis of the hyperparameters involved in the optimization process.

True Online SARSA

True Online SARSA provides a step up from the on-policy TD-based SARSA methodology. The progression occurred from SARSA(0), and then added eligibility traces, and then modified the trace calculation in order to prevent exploding gradients and wrong learning. The initial SARSA(0) update equation is as such:

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)$$

The True Online (0) equation is as such:

$$\begin{aligned} z &= \gamma\lambda z + (1 - \gamma\lambda w^\top x)x \\ q &= w^\top x \\ q' &= w^\top x' \\ q_{old} &= \overrightarrow{w_{old}} x \\ \delta &= r + q' - q \\ w &= w + \alpha(\delta + q - q_{old})z - \alpha(q - q_{old})x \end{aligned}$$

This allows us to get exact values for the q-functions at certain states and actions, providing for the grounds to compute optimal policies. What's more, is that it allows us to do so with convergence and stability.

Implementation and Pseudo-code

Since True Online SARSA, and other eligiblity-based SARSA implementations, may be unstable with nonlinear function approximators, we decided to avoid neural network implementations and implement tile coding. We designed custom tile coding algorithms, using general practices implemented in IHT tile coding. A custom agent was designed to facilitate ϵ -greedy action selection, weight updates, trace calculation, and more. A separate runner was designed to store the agent, tile coder, and hyperparameters used in the training process.

Furthermore, our goal was to use OpenAI's `gym` API in order to make the training and application of the algorithms more relevant. The environments that are implemented in this would be the `CartPole-v1` environment and the `LunarLander-v3` environment.
The pseudo-code for the True Online SARSA algorithm is as follows:

```
Initialize weights w arbitrarily
For each episode:
    Initialize state s
    Choose action a using policy derived from Q (epsilon-greedy)
    Convert (s,a) to feature vector x
    Initialize z = 0
    q_old = 0
    While s is not terminal:
        Take action a, observe reward r and next state s'
        convert (s',a') to feature vector x'
        Choose next action a' using policy derived from Q (epsilon-greedy)
        q = w^T * x
        q' = w^T * x'
        delta = r + gamma * q' - q
        z = gamma * lambda * z + (1 - gamma * lambda * w^T * x) * x
        w = w + alpha * (delta + q - q_old) * z - alpha * (q - q_old) * x
        q_old = q'
        s = s'
        a = a'
    decay epislon value
```

Hyperparameters

The grid used to optimize for hyperparameters was as such:

Hyperparameter	Values
α	[0.001, 0.005, 0.01, 0.05]
λ	[0.995, 0.99, 0.95]
final ϵ	[0.01, 0.05, 0.1]
ϵ decay rate	[0.999, 0.995, 0.95, 0.9]
Number of tilings	[4,8,16]
Number of tiles per dim	[4,8,16]

The tuning process was fairly straightforward, we were able to create a parameter grid from the values used, and apply them for 500-750 episodes towards each algorithm and assess the results through learning curves, shown below. When hyperparameter tuning, we averaged the results over 5 different seed values and plotted the mean learning curve. After identifying the top 5 hyperparameter combinations, we then trained the top combination for 1250 episodes to get a more detailed learning curve.

Results and Further Discussion

PoleCart v1

The CartPole environment is a classic control problem where the agent must balance a pole on a cart by moving the cart left or right. The state space is continuous and consists of four variables: the cart's position, velocity, pole angle, and pole angular velocity. The action space is discrete, with two possible actions: move left or move right.

The reward structure is simple: the agent receives a reward of $+1$ for every time step that the pole remains balanced. The episode ends when the pole falls over or when the cart moves out of bounds. The goal is to maximize the total reward over an episode, which encourages the agent to keep the pole balanced for as long as possible.

In this environment, the episodes are truncated to 500 time steps. Thus, the maximum reward an agent can receive per episode is 500. We found that our best hyperparameter combination achieved an average reward of around 475 over 1250 episodes, indicating that the agent was able to learn an effective policy for balancing the pole.

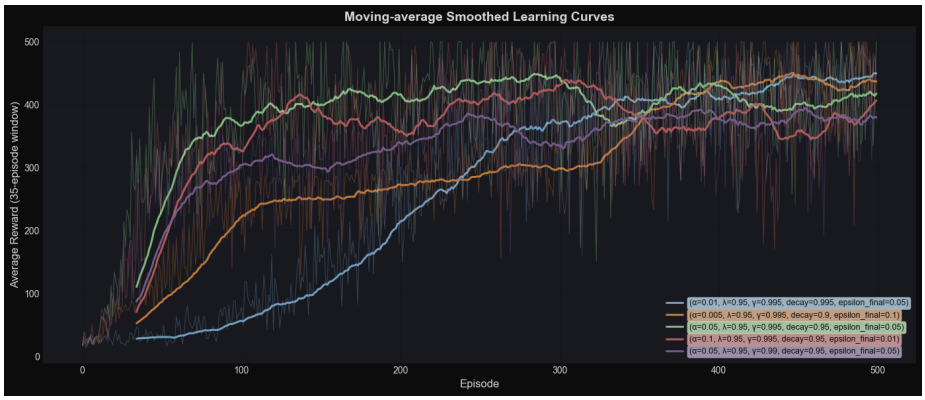


Figure 1: Top 5 Hyperparameter Combinations (with raw curves) for True Online SARSA on CartPole-v1

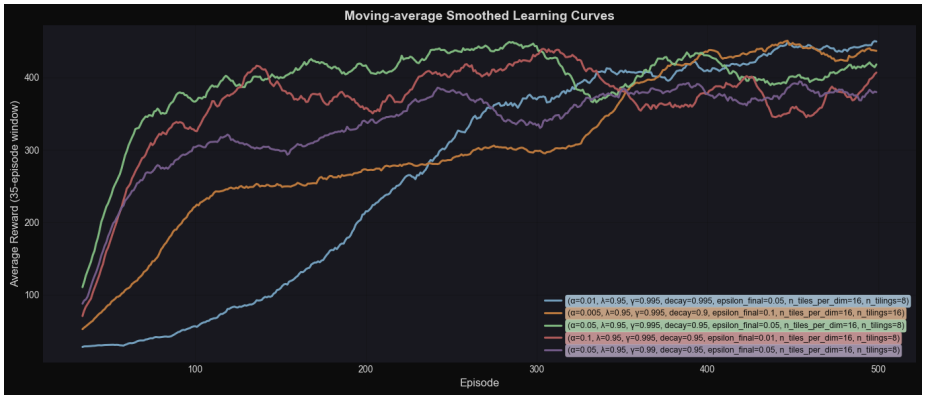


Figure 2: Top 5 Hyperparameter Combinations for True Online SARSA on CartPole-v1

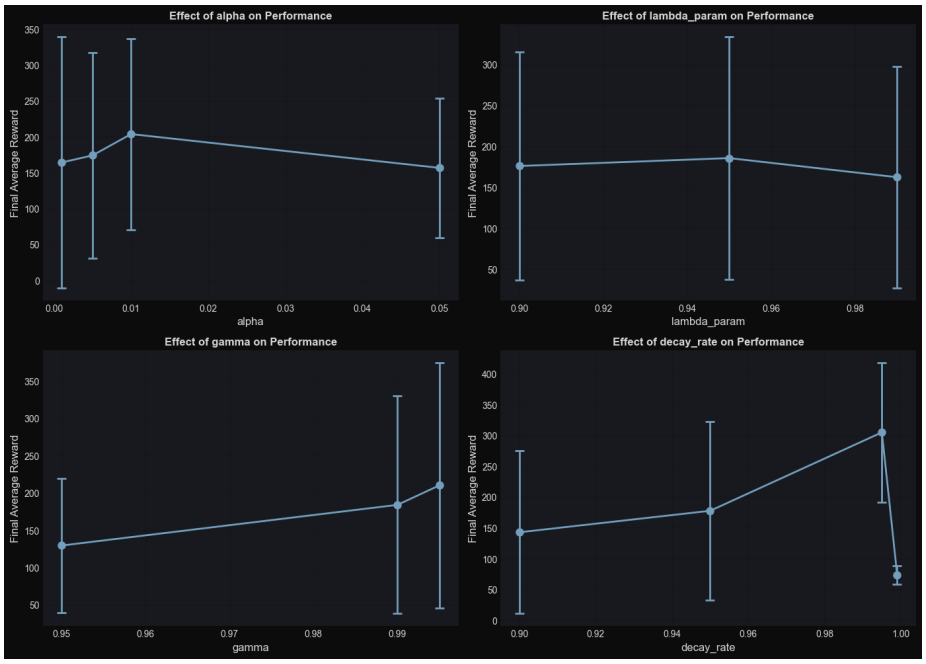


Figure 3: Effect of Various Hyperparameters on Performance for True Online SARSA on CartPole-v1

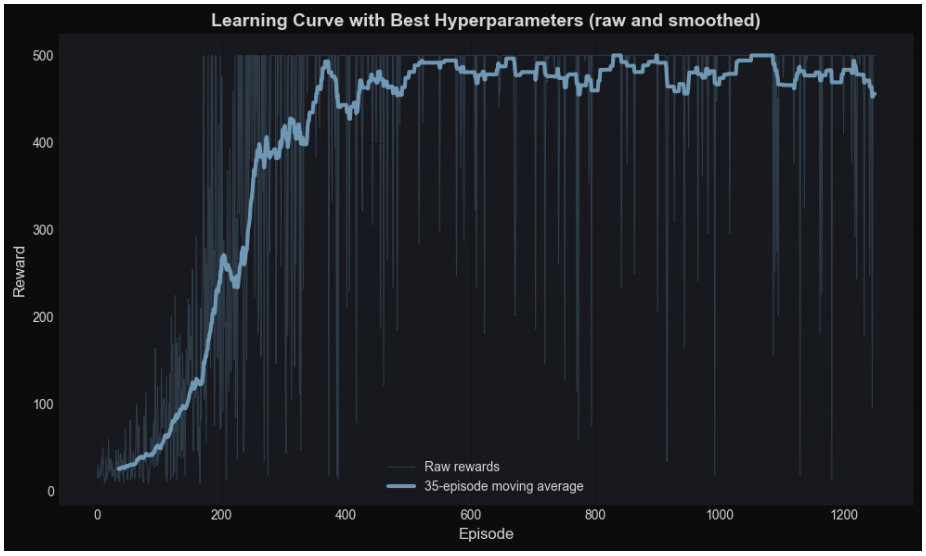


Figure 4: Best Hyperparameter Combination Detail for True Online SARSA on CartPole-v1

MountainCar v0

MountainCar is another classic control problem that is often used to benchmark reinforcement learning algorithms. The objective is to drive a car up a steep hill, but the car's engine is not powerful enough to climb the hill directly. Instead, the car must build up momentum by driving back and forth between two hills.

Since this environment has a smaller state space, we were able to use a larger tiling spaces, as shown in the hyperparameter curves. Additionally, success in the environment is defined as reach the flag at the top of the hill, which gives a reward of +100. Each time step the agent receives a reward of -1 until it reaches the goal. Thus, the agent is incentivized to reach the goal as quickly as possible. We found that the net reward was about -100 to -120 on average for successful runs.

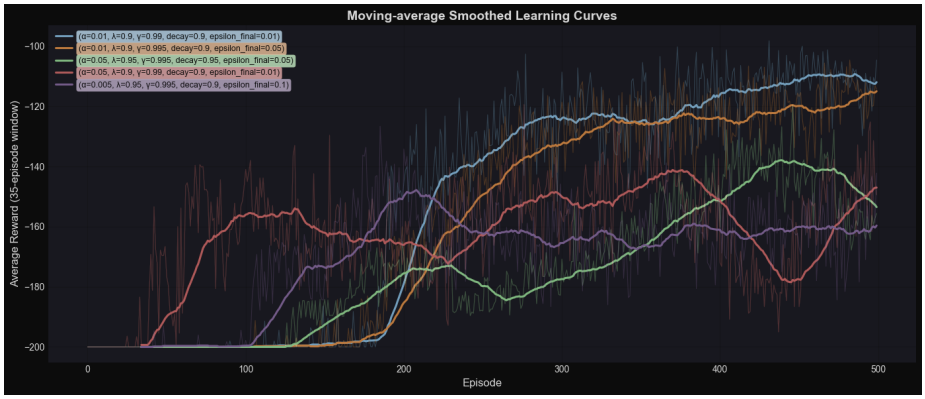


Figure 5: Top 5 Hyperparameter Combinations (with raw curves) for True Online SARSA on MountainCar-v0

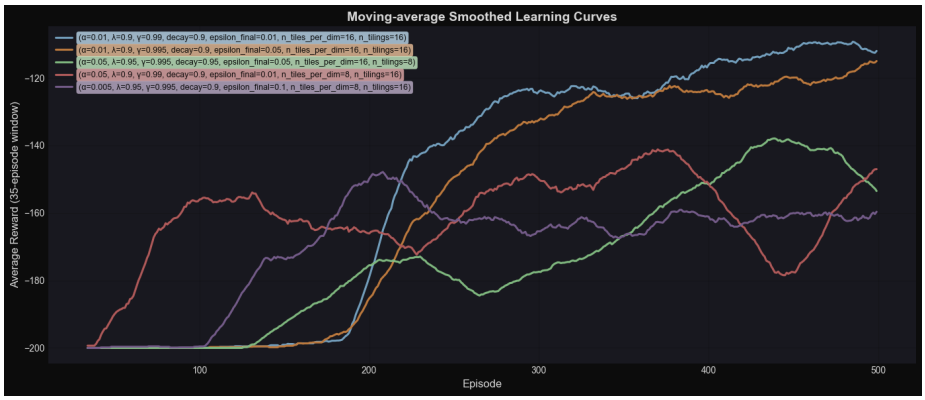


Figure 6: Top 5 Hyperparameter Combinations for True Online SARSA on MountainCar-v0

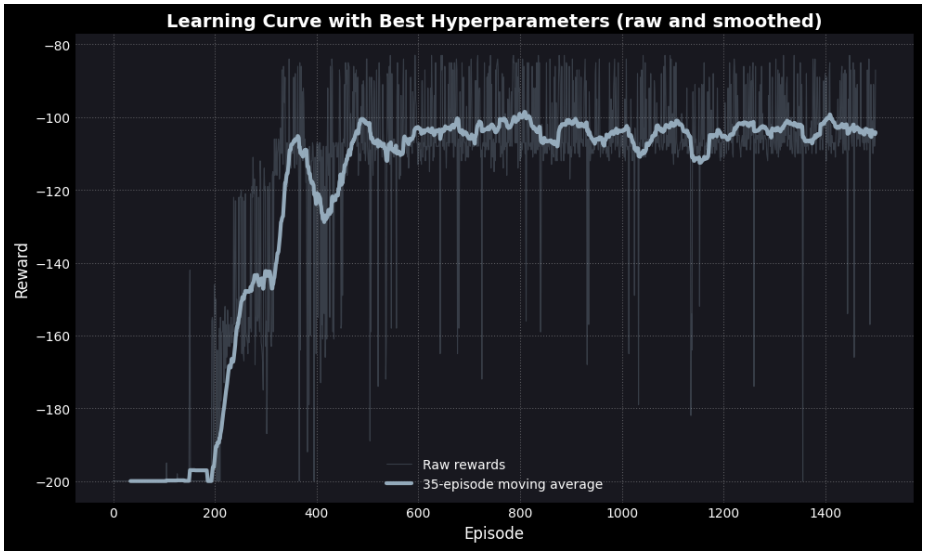


Figure 7: Top Hyperparameter Combination Detail for True Online SARSA on MountainCar-v0

n-Step SARSA

n-step SARSA is a middle ground between updating after a single move (TD learning) and waiting for the entire game to finish (Monte Carlo). Instead of learning from the only immediate next step, the agent waits to observe a sequence of n actual rewards before updating its value estimate. This approach helps the agent propagate reward information faster than 1-step methods while maintaining lower variance than full Monte Carlo updates.
The function approximation of G is given below.

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}, A_{t+n})$$

The n-step update is

$$\mathbf{w}_{t+n} = \mathbf{w}_{t+n-1} + \alpha[G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1})]\Delta\hat{q}(S_t, A_t, \mathbf{w}_{t+n-1})$$

Algorithm 1 Episodic semi-gradient n-step SARSA

Input: Differentiable action-value function parameterization $\hat{q}(s, a, \mathbf{w})$
Parameters: Step size α, discount γ, exploration ϵ, step parameter n
Initialize weights $\mathbf{w}$
for each episode **do**
    Initialize and store S_0
    Select and store $A_0 \sim \epsilon$-greedy$(S_0, \mathbf{w})$
    $T \leftarrow \infty$
    for $t = 0, 1, 2, \ldots$ **do**
        if $t < T$ **then**
            Take action A_t, observe R_{t+1}, S_{t+1}
            Store R_{t+1}, S_{t+1}
            if S_{t+1} is terminal **then**
                $T \leftarrow t + 1$
            else
                Select and store $A_{t+1} \sim \epsilon$-greedy$(S_{t+1}, \mathbf{w})$
            end if
        end if
        $\tau \leftarrow t - n + 1$
        if $\tau \geq 0$ **then**
            $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
            if $\tau + n < T$ **then**
                $G \leftarrow G + \gamma^n \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w})$
            end if
            $\mathbf{w} \leftarrow \mathbf{w} + \alpha[G - \hat{q}(S_\tau, A_\tau, \mathbf{w})]\nabla\hat{q}(S_\tau, A_\tau, \mathbf{w})$
        end if
        break if $\tau = T - 1$
    end for
end for

Tuning Hyperparameters

We implemented a custom tile coding algorithm. A custom agent wrapper was designed to facilitate ϵ-greedy action selection and handle the n-step lookahead. A separate runner class manages the training loop, storing the agent, tile coder, and hyperparameters.
Furthermore, we used OpenAI's `gymnasium` API to make the training and application of the algorithms standardized. The environmens tested in this implementation are `CartPole-v1` and `LunarLander-v3`.

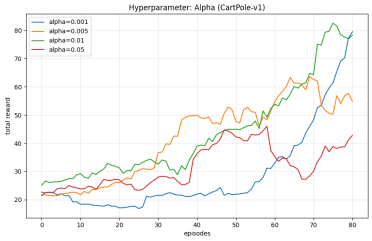
Hyperparameters

The grid used to optimize for hyperparameters was as follows:

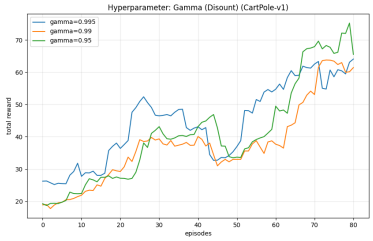
Hyperparameter	Values
α (Learning Rate)	[0.001, 0.005, 0.01, 0.05]
γ (Discount)	[0.995, 0.99, 0.95]
ϵ Decay Rate	[0.999, 0.995, 0.95, 0.9]
Final ϵ	[0.01, 0.05, 0.1]
n (Step Size)	[1, 5, 10, 20]
Number of Tilings	[4, 8, 16]
Number of Tiles per Dim	[4, 8, 16]

We isolated each hyperparamteter by varying it while keeping the others constant. This meant we trained the model in 23 different ways, with each getting 100 episodes. The results were assessed through learning curves, which were smoothed using a moving average with a window size of 20 to reduce variance.

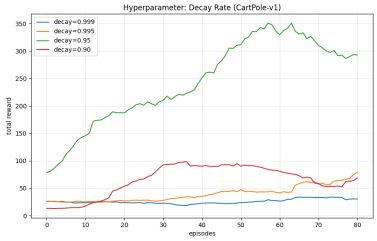
CartPole-v1



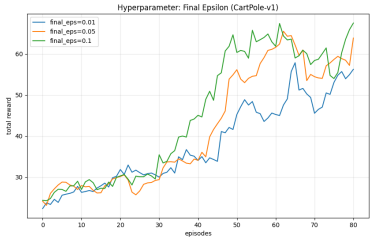
(a) Learning Rate (α)



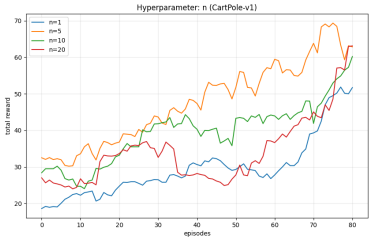
(b) Discount Factor (γ)



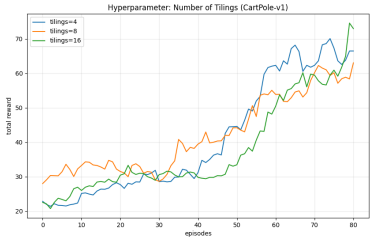
(c) Epsilon Decay Rate



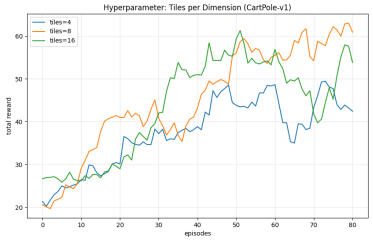
(d) Final Epsilon



(e) Step Size (n)



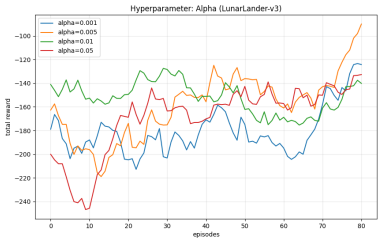
(f) Number of Tilings



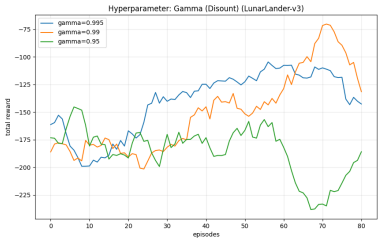
(g) Tiles per Dimension

Figure 8: Hyperparameter tuning results for the CartPole environment with n-step SARSA.

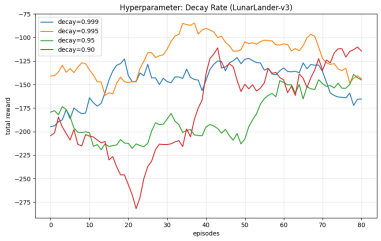
LunarLander-v3



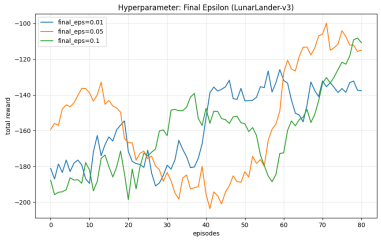
(a) Learning Rate (α)



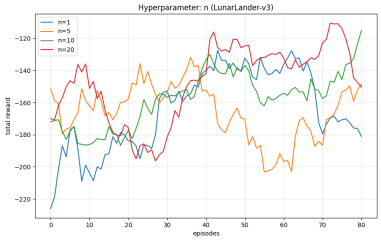
(b) Discount Factor (γ)



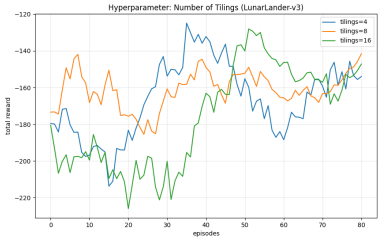
(c) Epsilon Decay Rate



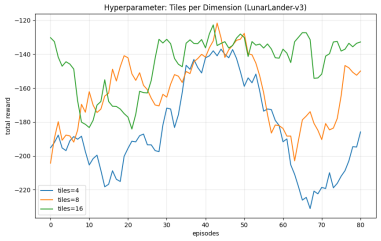
(d) Final Epsilon



(e) Step Size (n)



(f) Number of Tilings



(g) Tiles per Dimension

Figure 9: Hyperparameter tuning results for the LunarLander environment with n-step SARSA.

Results and Further Discussion

CartPole-v1

The best performing hyperparameters for CartPole were:

- Learning Rate: 0.01
- Discount Factor: 0.95
- Epsilon Decay Rate: 0.95
- Final Epsilon: 0.1
- Step Size: 5
- Number of Tilings: 16
- Tiles per Dimension: 8

Using these hyperparameters, the agent was trained for 30 epochs with 500 episodes each, eventually consistantly reaching near-optimal performance with rewards approaching the maximum of 500 per episode.

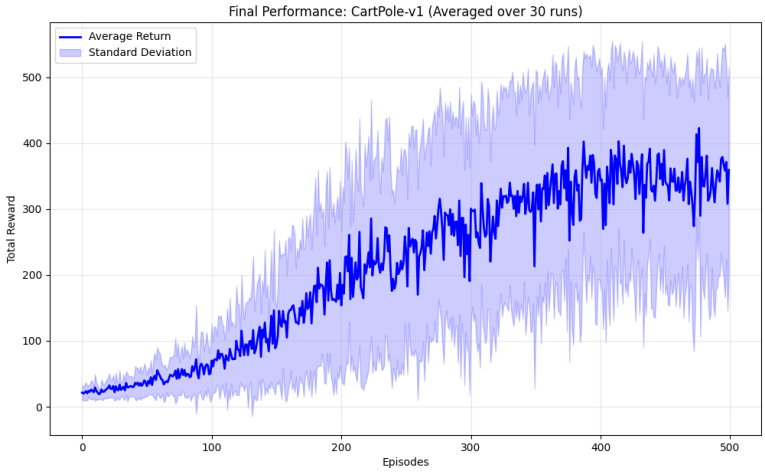


Figure 10: Final CartPole environment with n-step SARSA

LunarLander-v3

The best performing hyperparameters for LunarLander were:

- Learning Rate: 0.005
- Discount Factor: 0.99
- Epsilon Decay Rate: 0.995
- Final Epsilon: 0.05
- Step Size: 20
- Number of Tilings: 16
- Tiles per Dimension: 16

Using these hyperparameters, the agent was trained for 30 epochs with 500 episodes each, consistently land the lunar module successfully.

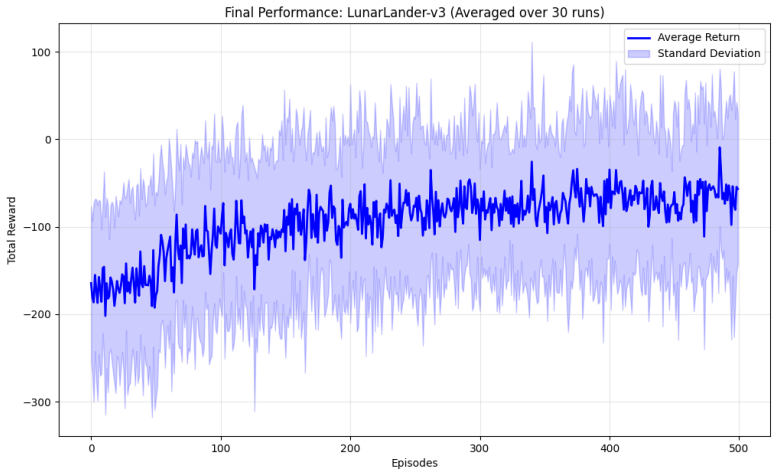


Figure 11: Final LunarLander environment with n-step SARSA

Advantage Actor-Critic with GAE

Vanilla Policy Gradient (VPG) with Generalized Advantage Estimation (GAE) is a policy gradient method that directly optimizes the policy by maximizing expected returns. The algorithm uses a shared neural network backbone that outputs both policy (action probabilities) and value estimates, allowing for efficient learning of both components simultaneously.

The policy gradient theorem states that the gradient of the expected return can be estimated using:

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A_t]$$

where A_t is the advantage function. GAE [2] provides a way to estimate advantages with reduced variance:

$$A_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ is the temporal difference error.

The value function is updated using mean squared error loss between predicted values and empirical returns:

$$L_{value} = (V_\phi(s_t) - R_t)^2$$

Implementation and Pseudo-code

The implementation follows the OpenAI Spinning Up tutorial on Vanilla Policy Gradient [1]. A shared neural network backbone was used to reduce hyperparameter tuning time and potentially improve performance through shared representations. Small network architectures were chosen to reduce training time and computational requirements while maintaining sufficient capacity for the tasks.

The algorithm proceeds in the following steps:

1. Initialize policy and value networks with shared backbone
2. For each training iteration:
   (a) Collect a batch of episodes using the current policy
   (b) Compute rewards, value estimates, and action probabilities for each episode
   (c) Compute advantages using GAE: $A_t = \delta_t + \gamma\lambda A_{t+1}$
   (d) Update policy network using: $L_{policy} = -\log \pi_\theta(a_t|s_t) \cdot A_t$
   (e) Update value network using: $L_{value} = (V_\phi(s_t) - R_t)^2$

Hyperparameters

The grid used to optimize for hyperparameters was as such:

Hyperparameter	CartPole Values	LunarLander Values
Learning Rate (α)	[0.001, 0.0005]	[0.001, 0.0005]
Discount Factor (γ)	[0.99, 0.95]	[0.99, 0.95]
GAE Lambda (λ)	[0.95, 0.9]	[0.95, 0.9]
Batch Size	[50]	[50, 75, 100]

For CartPole, grid search was performed over the hyperparameters with a fixed batch size of 50, running each combination for 50 epochs. The average return over the last 10 episodes was used to evaluate performance. For LunarLander, batch size was also included in the search, and each combination was run for 100 epochs.

Results and Further Discussion

CartPole-v1

The best performing hyperparameters for CartPole were:

- Learning Rate: 0.001
- Discount Factor: 0.99
- GAE Lambda: 0.95
- Batch Size: 50

Using these hyperparameters, the agent was trained for 200 epochs. The learning curve shows steady improvement in average return over time, eventually reaching near-optimal performance with consistent rewards approaching the maximum of 500 per episode.

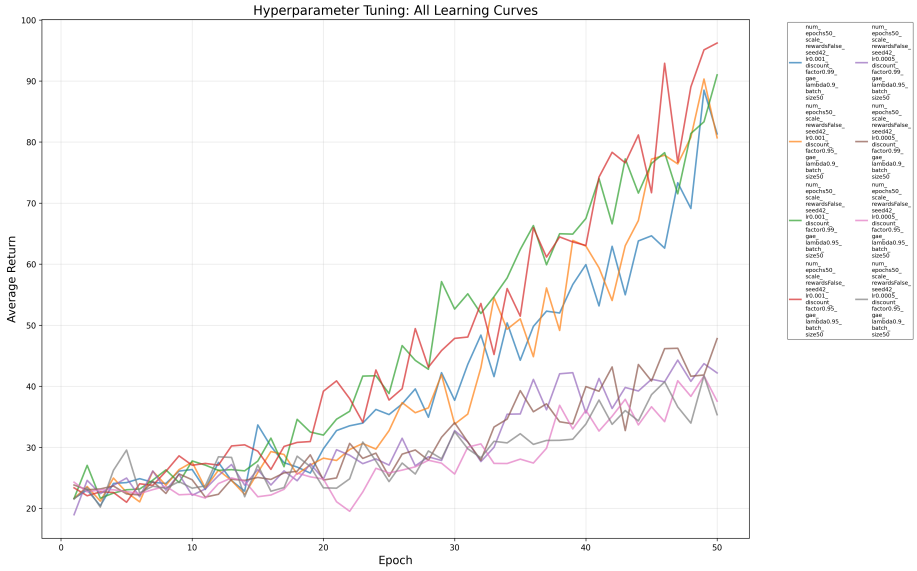


Figure 12: Hyperparameter tuning learning curves for VPG with GAE on CartPole-v1

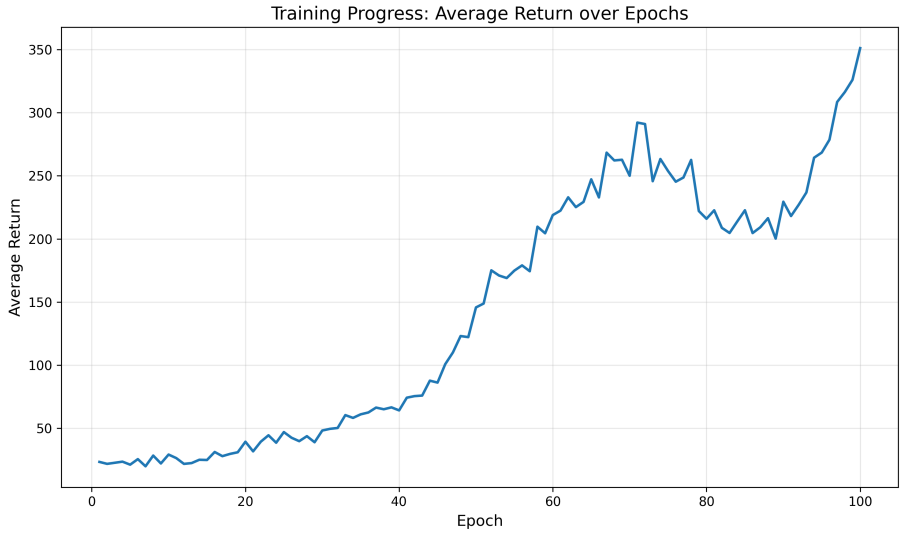


Figure 13: Final training run with best hyperparameters for VPG with GAE on CartPole-v1

LunarLander-v3

The best performing hyperparameters for LunarLander were:

- Learning Rate: 0.001
- Discount Factor: 0.99
- GAE Lambda: 0.95
- Batch Size: 75

Using these hyperparameters, the agent was trained for 150 epochs. The learning curve demonstrates steady improvement in average return over time, eventually reaching a policy that could consistently land the lunar module successfully. The agent learned to efficiently manage fuel consumption while controlling its descent to achieve safe landings on the designated pad.

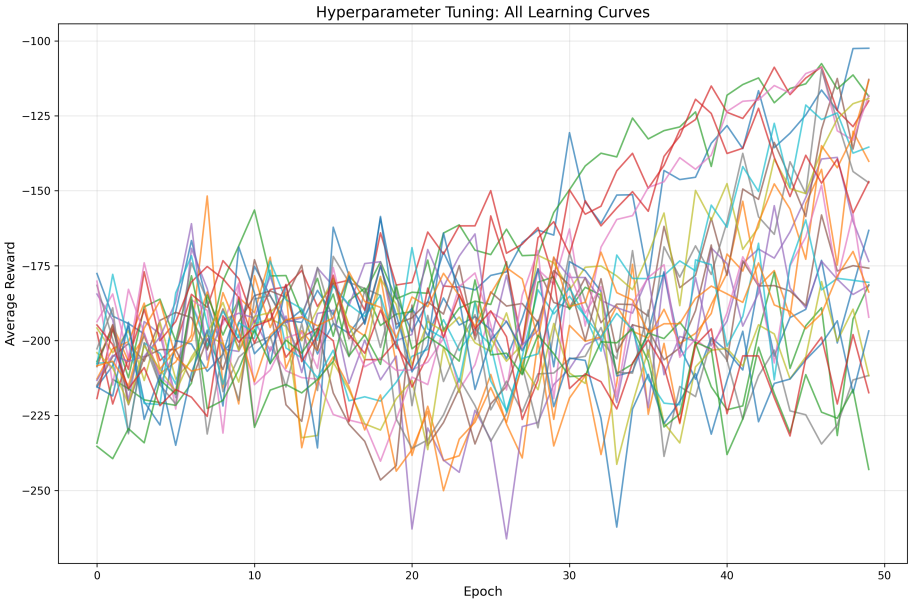


Figure 14: Hyperparameter tuning learning curves for VPG with GAE on LunarLander-v3

Figure 15: Legend for Figure 14

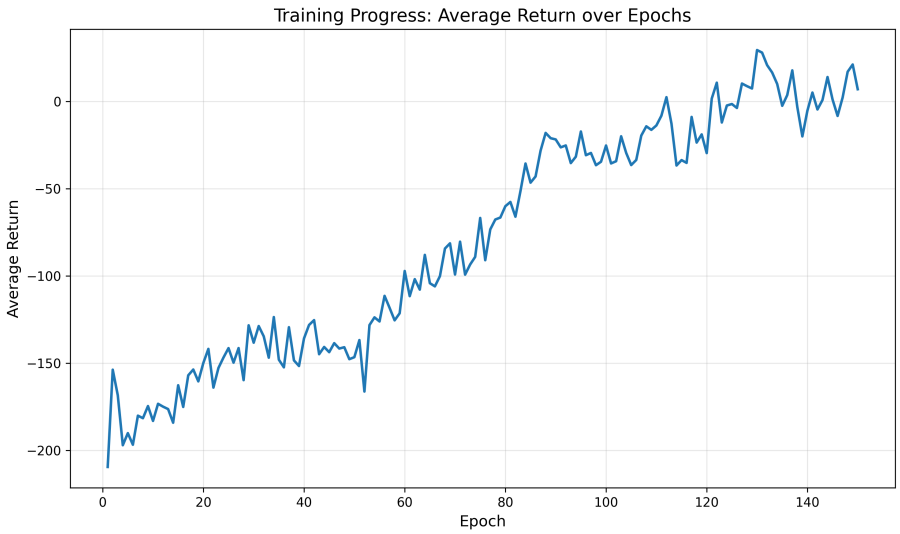


Figure 16: Final training run with best hyperparameters for VPG with GAE on LunarLander-v3

Works Cited

1. Sutton, Richard S., and Andrew G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

2. TileCoding Software `http://incompleteideas.net/tiles/tiles3.html`

3. OpenAI Spinning Up. "Vanilla Policy Gradient." `https://spinningup.openai.com/en/latest/algorithms/vpg.html`

4. Schulman, John, et al. "High-dimensional continuous control using generalized advantage estimation." *arXiv preprint arXiv:1506.02438* (2015).