Tutorial 4: Value Function Approximation

Reinforcement Learning Course

Learning Objectives

By the end of this tutorial, you should be able to:

- Apply stochastic gradient descent (SGD) to value function learning
- Implement semi-gradient TD learning with neural networks
- Understand experience replay and target networks for stability
- Train a Deep Q-Network (DQN) agent on CartPole

Notation Reference

- $\hat{v}(s, \mathbf{w})$: Approximate value function with parameters $\mathbf{w}$
- $\hat{q}(s, a, \mathbf{w})$: Approximate action-value function with parameters $\mathbf{w}$
- $\mathbf{w} \in \mathbb{R}^d$: Weight vector (parameters)
- α: Learning rate (step size)
- U_t: Target value for update
- $\nabla_{\mathbf{w}}$: Gradient with respect to parameters $\mathbf{w}$
- $\mathcal{D}$: Experience replay buffer
- $\mathbf{w}^-$: Target network parameters

1 Stochastic Gradient Descent for RL

1.1 Deriving SGD Updates

We want to minimize the mean squared value error: $\overline{VE}(\mathbf{w}) = \mathbb{E}_\mu[(v_\pi(S) - \hat{v}(S, \mathbf{w}))^2]$

(a) Derive the gradient descent update rule for minimizing this objective.

(b) We don't know the true value $v_\pi(S)$. In Monte Carlo, we replace it with the return G_t. Write the MC SGD update.

(c) In TD(0), we use the TD target $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$ instead. Write the TD(0) SGD update.

(d) Why is TD(0) called a "semi-gradient" method? What part of the gradient is being ignored?

2 Deep Q-Networks (DQN)

2.1 Understanding DQN Components

DQN introduced several key innovations to stabilize deep RL. For each component below, explain what problem it solves.

(a) **Experience Replay:** Store transitions (s_t, a_t, r_t, s_{t+1}) in a replay buffer $\mathcal{D}$ and sample random mini-batches for updates.

(b) **Target Network:** Maintain a separate network $Q(s, a; \mathbf{w}^-)$ with frozen parameters that are updated periodically (e.g., every 1000 steps).

(c) **ϵ-greedy Exploration:** Select random actions with probability ϵ, greedy actions otherwise.

3 Programming Exercises

In these exercises, we'll use the CartPole-v1 environment from Gymnasium. Full documentation is available at: `https://gymnasium.farama.org/environments/classic_control/cart_pole/`

Environment Description: A pole is attached to a cart moving along a frictionless track. The goal is to balance the pole upright by applying forces to move the cart left or right. An episode ends when:

- The pole angle exceeds ±12 from vertical
- The cart position exceeds ±2.4 units from center
- Episode length reaches 500 steps (solved!)

State Space: The observation is a 4-dimensional continuous vector:

Index	**Variable**	**Range**
0	Cart position (x)	$[-4.8, 4.8]$
1	Cart velocity ($\dot{x}$)	$[-\infty, \infty]$
2	Pole angle (θ)	$[-0.418, 0.418]$ rad ($\approx \pm 24$)
3	Pole angular velocity ($\dot{\theta}$)	$[-\infty, \infty]$

Action Space: Discrete actions:

- Action 0: Push cart to the left
- Action 1: Push cart to the right

Rewards:

- +1 for every timestep the pole remains upright
- The task is considered solved when average reward ≥ 195 over 100 consecutive episodes

PyTorch Usage: This implementation uses PyTorch for building and training neural networks. If you are unfamiliar with PyTorch basics, refer to the official tutorials: `https://pytorch.org/tutorials/beginner/basics/intro.html`

3.1 Implementing Semi-Gradient TD with Linear Approximation

Open `tutorial_template.py`. First, you'll implement TD learning with linear function approximation on CartPole with hand-crafted features.

CartPole state: $[x, \dot{x}, \theta, \dot{\theta}]$ where x is cart position, θ is pole angle.

(a) **Implement** `create_features`: Create a feature vector from the state. Use polynomial features up to degree 2 (including cross terms).

(b) **Implement** `LinearQNetwork`: A linear Q-network where $Q(s, a) = \mathbf{w}_a^\top \phi(s)$ with separate weights for each action.

(c) **Implement** `update`: The semi-gradient TD update for linear Q-learning:

$$\mathbf{w}_a \leftarrow \mathbf{w}_a + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]\phi(s)$$

(d) **Test your implementation**: Run the training loop. The agent should solve CartPole (average reward > 195 over 100 episodes) within 500-1000 episodes.

Expected output:

- Training should show gradual improvement
- Episodes will be short initially (pole falls quickly)
- After convergence, episodes should reach maximum length (500 steps)
- Plot shows increasing average reward over episodes

3.2 Implementing Deep Q-Network (DQN)

Now implement the full DQN algorithm with experience replay and target networks.

(a) **Implement** `ReplayBuffer`: Store transitions and sample random mini-batches.
   - Store tuples $(s_t, a_t, r_t, s_{t+1}, \text{done}_t)$
   - Implement `add()` and `sample(batch_size)`

(b) **Implement** `QNetwork`: A neural network with:
   - Input: state vector (4 dimensions for CartPole)
   - Hidden layers: 2 layers with 128 units each, ReLU activation
   - Output: Q-values for each action (2 for CartPole)

(c) **Implement the DQN training loop**:
   - Select actions using ϵ-greedy policy
   - Store transitions in replay buffer
   - Sample mini-batches and compute TD targets using target network
   - Update Q-network using MSE loss
   - Periodically update target network: $\mathbf{w}^- \leftarrow \mathbf{w}$

(d) **Hyperparameters to use:**

- Batch size: 64
- Learning rate: 0.001
- γ: 0.99
- ϵ: start at 1.0, decay to 0.01
- Target network update frequency: every 100 steps
- Replay buffer size: 10,000

Expected output:

- Your code should generate plots showing training progress
- DQN should solve CartPole faster than linear approximation (200-400 episodes)
- Learning curve should be smoother than linear due to replay buffer
- Final policy should consistently balance the pole for 500 steps

4 Reflection Questions

4.1 Comparing Methods

After implementing both linear and deep Q-learning:

(a) Which method learned faster? Why?

(b) Which method achieved better final performance? Why?

(c) What are the advantages and disadvantages of each approach?

(d) When would you choose linear function approximation over deep learning?