Recitation 8
Reinforcement Learning

10-301/601: Introduction to Machine Learning

11/14/2025

1 Introduction to Reinforcement Learning

1.1 Markov Decision Process

A Markov decision process is a tuple $(\mathcal{S}, \mathcal{A}, T, R, \gamma, s_0)$, where:

- $\mathcal{S}$ is the set of states
- $\mathcal{A}$ is the set of actions
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition function
- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function
- $\gamma \in [0, 1)$ is the discount factor
- s_0 is the start state

When we play a game, we can model the game using a Markov decision process as follows:

1. We start in some state s_0.
2. Choose some action $a_0 \in \mathcal{A}$.
3. As a result of our choice, the state of the game transitions to some successor state s_1. For non-deterministic transition, we draw the next state according to our transition function. That is to say,
$$T(s_t, a_t, s_{t+1}) = P(S_{t+1} = s_{t+1} | S_t = s_t, A_t = a_t)$$
4. Then we pick another action a_1, and so on.

We can represent it as

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \dots$$

Let $r_t = R(s_t, a_t, s_{t+1})$. Upon visiting a sequence of states $s_0, s_1, s_2, s_3 \dots$ with respective actions $a_0, a_1, a_2, \dots$ the **total discounted payoff** is given by

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \dots$$

Note that different sequences of states may have a different payoff value, and the reward is discounted if attained later.

1.2 Policy and Value Function

A **policy** is any function $\pi : \mathcal{S} \to \mathcal{A}$ mapping from the states to the actions. We say that we are executing some policy π if, whenever we are in state s, we take action $a = \pi(s)$.

For the optimal policy function π^* we can compute its **value function** at state s as:

$$\begin{aligned} V^{\pi^*}(s) &= V^*(s) \\ &= \mathbb{E}\left[R(s_0, \pi^*(s_0), s_1) + \gamma R(s_1, \pi^*(s_1), s_2) + \gamma^2 R(s_2, \pi^*(s_2), s_3) \cdots \mid s_0 = s, \pi^*\right]. \end{aligned}$$

In other words, this is the best possible expected total payoff that can be attained using **any policy** π.

This **optimal value function** can be represented using Bellman's equations (named **Bellman optimality equation** for state value function), i.e.,

$$V^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s, a)(R(s, a, s') + \gamma V^*(s')).$$

If $R(s, a, s') = R(s, a)$ (deterministic transition), then we have the form we saw in class:

$$V^*(s) = \max_{a \in \mathcal{A}} \left\{ R(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a) V^*(s') \right\}.$$

The interpretation of Bellman equation is also very intuitive: "how valuable is the current state under a policy?" Then it naturally follows that the optimal policy $\pi^* : \mathcal{S} \to \mathcal{A}$ can be found as

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s, a)(R(s, a, s') + \gamma V^*(s')).$$

1.3 Q-Value

We can also define a Q function $Q(s, a)$ that returns the expected discounted future value of taking action a at state s.

Question: When would we want to use $Q(s, a)$ instead of $V(s)$?
Answer:

We can take a part of $V^*(s)$ as $Q^*(s, a)$ so that we have

$$Q^*(s, a) = \sum_{s' \in \mathcal{S}} p(s'|s, a)(R(s, a, s') + \gamma V^*(s'))$$

$$V^*(s) = \max_{a \in \mathcal{A}} \underbrace{\sum_{s' \in \mathcal{S}} p(s'|s, a)(R(s, a, s') + \gamma V^*(s'))}_{Q^*(s,a)} = \max_{a \in \mathcal{A}} Q^*(s, a).$$

Then it follows that the optimal policy can be obtained as

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} Q^*(s, a).$$

2 Value Iteration

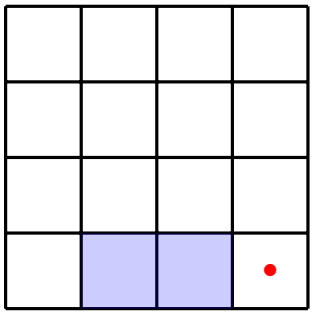


Figure 1: The cliff-walking environment

Here, we assume a 4×4 grid environment. The reward is 1 for reaching the cell with the red dot, -1 for reaching the shaded cells, and 0 for all other cells. The episode ends if the agent lands in either the shaded cells or the red-dot cell. The state space ($\mathcal{S}$) is all the cells. The action space (A) is up, down, left or right. If the agent tries to move out of the grid, it simply goes back to its previous state. The discount factor, γ, is 0.9.

Bellman optimality equation for state value function:

$$V^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s,a)(R(s,a,s') + \gamma V^*(s'))$$

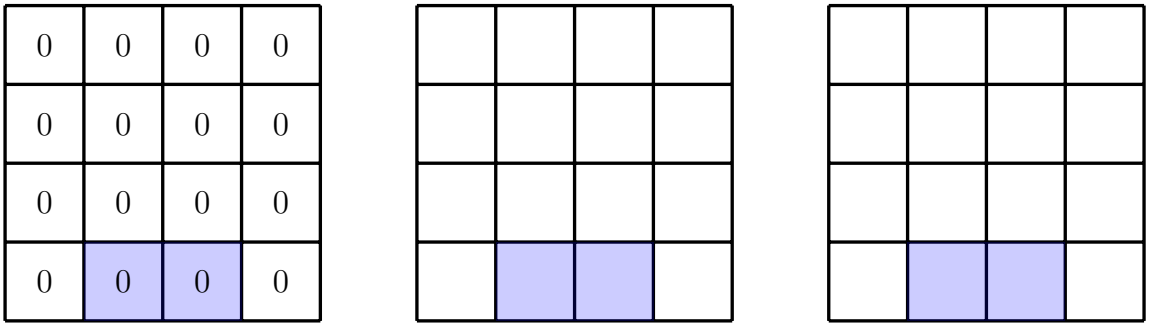
We can numerically approximate V^* using synchronous value iteration and asynchronous value iteration. The recurrence relation is defined as follows for **synchronous** value iteration for all $s \in \mathcal{S}$:

$$V^{(t+1)}(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s,a)(R(s,a,s') + \gamma V^{(t)}(s')).$$

Notice the time term (t), hence the name synchronous. For **asynchronous** value iteration, we use for all $s \in \mathcal{S}$:

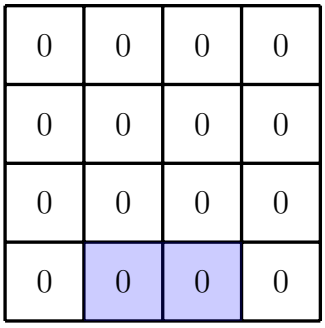
$$V(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s,a)(R(s,a,s') + \gamma V(s')).$$

1. Find the updated value of each cell after the first round and after the second round of synchronous value iteration. Assume deterministic transition function.

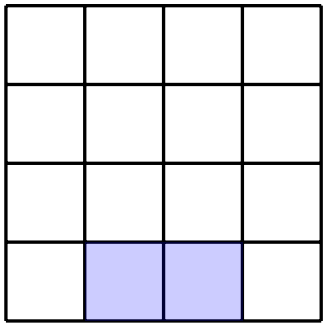


Initial | After 1st round | After 2nd round

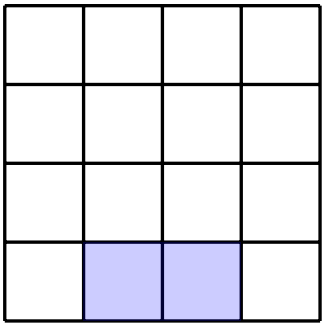
2. Starting over, find the update value of each cell after one round of asynchronous value iteration. Visit each cell in two different orders: (1) bottom right to top left, and (2) top left to bottom right.



Initial

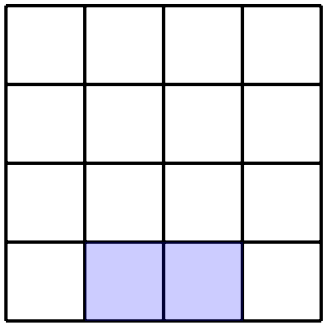


BR to TL

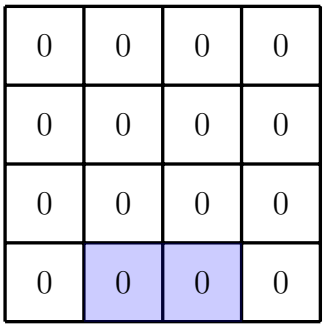


TL to BR

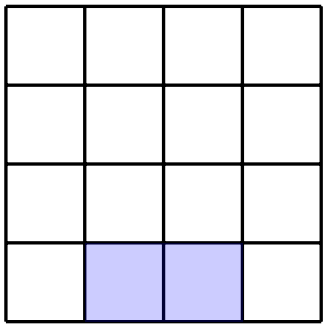
3. What is the policy learned after the one round of asynchronous value iteration when the cells were visited from bottom right to top left? If there is a tie, pick the action that comes first alphabetically (i.e., priority: $\downarrow > \leftarrow > \rightarrow > \uparrow$).



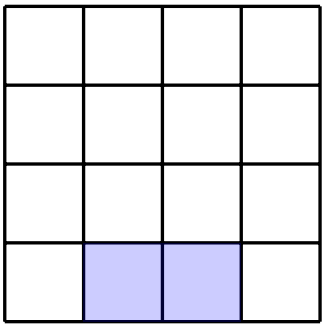
4. Now suppose that the environment is sloped downward towards the cliff, with all the other settings unchanged. For every action taken, there is a 0.5 probability that the agent will move as intended and a 0.5 probability that the agent will slip and move 1 cell down instead. Report the values after two rounds of synchronous value iteration.



Initial



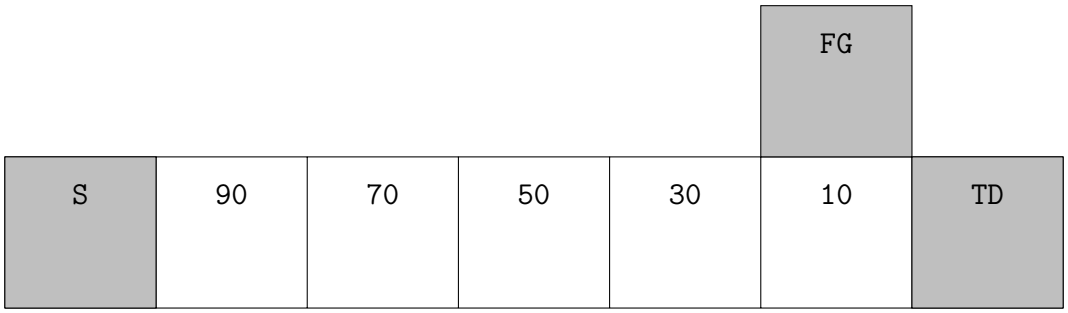
After 1st round



After 2nd round

3 Q-Learning

Let's play some football (the ~~real~~ American kind). We will divide the field into eight states (labeled slightly differently from an official football field for clarity):



Here are some basic rules (slightly different than official football rules, but let's work with it for the sake of this example):

- There are two teams. For simplicity, let's call them Offense and Defense.
- The length of the field is 100 yards; we will set each 20-yard interval on this field as its own state, in addition to the end zones (`S`, `TD`, and `FG`).
- The drive (or, in reinforcement learning terms, episode) terminates if the ball is moved into the end zone.
- At each state, the Offense may either `run` the ball or `pass` it (always in the direction of `TD`).

The head coach argues that it is not a good idea to assume a value for the transition probability from one state to the next. She also argues that since it is not the case that the Offense will always score seven points when going into the `TD` state (they can also score six points or eight) it is better to not assume a reward function. It is also possible to start the drive from anywhere in the field. For all these reasons, she encourages the offensive coordinators to try Q-learning.

1. The offensive coordinators, now convinced, decide to build a Q-value table that matches the state and action space. What is the size of this table?

2. The offensive coordinators initialize $Q(s, a) = 0 \ \forall s, a$. They decide to update it by sampling random drives from games previously played by the Offense this season (note that this approach uses experience replay and not a standard MDP since individual plays in a football game are not independent unlike moves in a grid world). They give you the empty Q-value table for ease:

   Q(·, `run`):

					FG	
S	90	70	50	30	10	TD

   Q(·, `pass`):

					FG	
S	90	70	50	30	10	TD

   Time to watch some football :) Assume the learning rate $\alpha = 0.1$ and the discount factor $\gamma = 0.5$. Update the Q-table above for each of the following episodes using the Temporal Difference error update:

   (a) **Episode 1, iteration 1**: Beginning at the `30` state, Offense `run`s the ball to `10`. Offense collects zero reward.
   $Q(\texttt{30}, \texttt{run}) =$

   (b) **Episode 1, iteration 2**: Offense `pass`es the ball from the `10` but stays at the `10`. Offense collects zero reward.
   $Q(\texttt{10}, \texttt{pass}) =$

   (c) **Episode 1, iteration 3**: From the `10`, Offense `pass`es the ball again and reaches the `TD` state. The drive terminates. Offense collects a reward of 7.
   $Q(\texttt{10}, \texttt{pass}) =$

   (d) After updating the Q-function for the first episode, the head coach decides to quiz the offensive coordinators. If the Offense is in state `10`, what is the best action?
   $\pi(\texttt{10}) =$

(e) **Episode 2, iteration 1**: Beginning at the `90` state, the Offense `run`s the ball, causing a safety (they enter the `S` state). The drive terminates. Offense collects -2 reward.
$Q(\texttt{90}, \texttt{run}) =$

(f) After updating the Q-function for the second episode, the head coach decides to quiz the offensive coordinators again. If the Offense is in state `90`, what is the best action to take?
$\pi(\texttt{90}) =$

(g) **Episode 3, iteration 1**: Beginning at `50`, the Offense `pass`es the ball but stays at the `50`. Offense collects zero reward.
$Q(\texttt{50}, \texttt{pass}) =$

(h) **Episode 3, iteration 2**: Beginning at the `50`, the Offense `pass`es the ball to the `30`. Offense collects zero reward.
$Q(\texttt{50}, \texttt{pass}) =$

(i) **Episode 3, iteration 3**: Beginning at the `30`, the Offense `run`s the ball to the `10`. Offense collects zero reward.
$Q(\texttt{30}, \texttt{run}) =$

(j) **Episode 3, iteration 4**: Beginning at the `10`, the Offense `run`s the ball and ends the drive after entering the `FG` state. Offense collects a reward of 3 points.
$Q(\texttt{10}, \texttt{run}) =$

(k) After updating the Q-function for the third episode, the head coach decides to quiz the offensive coordinators for the last time. If the Offense is in state `10`, what is the best action to take?
$\pi(\texttt{10}) =$

4 Policy Gradient: Advantage Actor Critic (AAC or A2C)

4.1 Example

Policy: $P(a_t = R \mid s_t) = \frac{1}{1+e^{-(ws_t+b)}}$, $P(a_t = L \mid s_t) = \frac{1}{1+e^{ws_t+b}}$, with $\theta = [w, b]^T$

Gradient Estimate Formula: $g = \sum_t A_t \cdot \nabla_\theta \ln P(a_t \mid s_t)$, where $A_t = Q_\phi(s_t, a_t) - V_\phi(s_t)$ and $a_t \in \{L, R\}$

1. Calculate $\nabla_\theta \ln P(L \mid s_t)$ and $\nabla_\theta \ln P(R \mid s_t)$.
2. To calculate A_t, we need to know values of $Q_\phi(s_t, a_t)$, and $V_\phi(s_t)$, which come from the critic. For this example, the values are provided below, but in practice, we would fit V_ϕ, then estimate $Q_\phi - V_\phi$. Use the formula above to calculate A_t.

$V_\phi(s_t)$	-4.9	-4.1	-3.8	-2.7	-1.5
$Q_\phi(s_t, a_t)$	-5.4	-4.8	-3.6	-2.4	-1.1
Q_t	-5	-4	-3	-2	-1
(s_t, a_t)	(1.6, L)	(0.2, R)	(1.3, R)	(2.4, L)	(2.1, R)

3. Use the gradient estimate formula above to compute g, assuming that we start at $w = b = 0$ (our policy is uniform random for all s_t).

4.2 Implementation Details

4.2.1 N-Step-Returns

Why N-Step Returns Are Useful in Advantage Actor-Critic (A2C)

In the Advantage Actor-Critic (A2C) algorithm, we need to estimate the expected return from a given state in order to compute the *advantage*, which measures how much better an action is compared to the "average" action under the current policy. In other words, the difference between $Q_\phi(s_t, a_t)$ and $V_\phi(s_t)$. One common challenge in reinforcement learning is how to estimate these returns in a way that balances **bias** and **variance**.

Definition. The *N-step return* from time step t is defined as:

$$G_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V(s_{t+n}),$$

where $r_{t:t+n-1}$ are the rewards observed over the next n steps, γ is the discount factor, and $V(s_{t+n})$ is the estimated value of the state reached after n steps.

Bias-Variance Tradeoff. N-step returns provide a compromise between two extremes:

- **Monte Carlo (MC) or TD(0) returns:** Using the full return until the end of the episode (large n) produces an unbiased estimate of the true expected return. However, it suffers from very high variance, since it depends on stochastic trajectories over potentially long episodes.
- **One-step Temporal Difference returns [TD(1)]:** Using $n = 1$ gives low variance because it uses the critic's value estimate $V(s_{t+1})$ immediately, but it introduces bias, since that estimate may be inaccurate early in training or might not be perfectly in sync with the current policy.

N-step returns strikes a balance between these two methods. By selecting a moderate value of n (e.g., 5 or 10), we reduce the variance compared to Monte Carlo estimation while also reducing the bias compared to the purely bootstrapped 1-step TD return.

Practical Benefits in A2C. In A2C, using N-step returns stabilizes learning by:

- Providing smoother and more reliable gradient estimates for both the actor and the critic.
- Accelerating training, since the agent benefits from partial bootstrapping rather than waiting until the end of an episode.

- Reducing sensitivity to the length of episodes and making learning more stable across different environments.

Thus, N-step returns are an effective way to compute the returns used in the advantage estimation step of A2C, offering a practical tradeoff between bias and variance that leads to faster and more stable convergence.

Pytorch Efficient Implementation We can precompute a tensor of *cumulative discount factors*:

$$\boldsymbol{\gamma} = [1, \gamma, \gamma^2, \ldots, \gamma^{n-1}]$$

and use this to efficiently weight and sum the next n rewards.

Given a tensor of rewards `rewards` of shape `(N, T, 1)`, we can compute the N-Step-Returns by making use of the `cumsum` method. This avoids Python loops and uses efficient GPU or vectorized operations.

Implementation Sketch in PyTorch.

```
# rewards: tensor of shape (N, T, 1)
# next_state_values: tensor of shape (N, T, 1)
# terminated: tensor of shape (N, T, 1)
# gamma: scalar discount factor
# n_steps: number of steps for N-step return

# Precompute powers of gamma: [1, gamma, gamma^2, ..., gamma^(n-1)]

# Apply discount factors to the rewards
# Pad rewards with a zero on the left and n - 1 zeros
  to the right (total shape T + n)
# Calculate the cumulative sum
# Use cumsum[k + n] - cumsum[k] to get the rewards from k -> k + n
# Use the discount factors to bring every value back to their
  corresponding timestep value

# Get the values at timesteps t + n (remember the first
  entry in next_state_values already is t + 1)
# If t + n goes beyond T, use the last value (from time T)
# If the episode had already terminated by T, use a value of 0
# Discount values using gamma and n

# Add discounted rewards with values
```

Why This Is Efficient.

- The `torch.cumsum` operation is fully vectorized and runs efficiently on the GPU.

- By precomputing the discount factors, we avoid recomputing powers of γ inside loops.
- This approach avoids explicit Python for-loops over time steps.

4.2.2 Getting log-probabilities of executed actions

A stochastic policy can be implemented with a network that outputs a set of *logits* for each possible action. To compute the policy gradient, we need the log-probabilities of the actions that were actually taken by the agent during the episode. These logits can be extracted efficiently using `torch.gather` and their log-probabilities can be computed in a numerically stable way using the `torch.nn.functional.log_softmax` function.

Numerical Stability. Directly applying a softmax followed by a logarithm,

$$\log \pi(a_t \mid s_t) = \log \left(\frac{e^{z_{t,a_t}}}{\sum_j e^{z_{t,j}}} \right),$$

can lead to numerical instability when logits have large magnitudes.

Instead, PyTorch provides a stable implementation through:

$$\texttt{log\_probs\_all} = \texttt{torch.nn.functional.log\_softmax(logits, dim=-1)},$$

which computes the logarithm of the softmax in a single, stable step by subtracting the maximum logit internally to prevent overflow.

Gathering Log-Probabilities for Executed Actions. Once we have the tensor of log-probabilities, we can extract the log-probabilities corresponding to the actions that were actually executed at each timestep. Suppose:

- `logits` has shape (N, T, A),
- `actions` has shape $(N, T, 1)$.

We can obtain the relevant log-probabilities as:

```
log_probs = log_probs_all.gather(dim=-1, index=actions)
```

4.2.3 Stopping gradient propagation

One difference between supervised learning and deep RL is that: In traditional supervised learning, we are given labeled data, but in RL we collect various trajectories by deploying our agent in the environment. This raises the issue of identifying targets for training deep neural networks. PyTorch builds a dynamic computation graph during training for automatic differentiation when `loss.backward()` is called. However, in RL target computation, we use the network being trained to generate the target. This could also cause the gradient to propagate through the targets if mitigation steps are not taken. The following are the steps taken to stop this gradient propagation:

- `torch.no_grad()`

  This is used to disable gradient tracking when you don't need gradients, such as during evaluation or when performing manual tensor updates. PyTorch stops building the computation graph, reducing memory usage, and improving speed. However, tensors created within the block still maintains autograd metadata if they were originally required to grad.

- `torch.inference_mode()`

  This is used specifically for inference. Not only disables gradient tracking, it also freezes the autograd engine. It prevents accidentally modifying inference tensors in ways that could lead to undefined autograd states.

- `torch.Tensor.detach()`

  This returns a new tensor that shares the same data but is not connected to the computation graph. It is used to convert model output into "plain tensors" without affecting training, and also to reuse intermediate results without tracking gradients.

`torch.Tensor.detach()` is tensor-level, while `torch.no_grad()` / `torch.inference_mode()` are block-level context managers.

PyTorch Techniques to prevent gradient propagation

```
import torch
import torch.nn as nn

model = nn.Linear(10, 1)
x = torch.randn(3, 10, requires_grad=True)

# ---------------------------
# Usual forward (gradients ON)
# ---------------------------
y = model(x)           # builds graph
loss = y.sum()
loss.backward()        # works normally

# ---------------------------
# Inference with torch.no_grad()
# ---------------------------
with torch.no_grad():
    y_eval = model(x)  # no graph is built
    # y_eval still has metadata but requires_grad=False

# ---------------------------
# Inference mode
# ---------------------------
with torch.inference_mode():
    y_prod = model(x)  # no autograd metadata kept

# ---------------------------
# Detach example
# ---------------------------
y_detached = y.detach()   # breaks gradient flow
# y_detached shares storage with y but has no grad history
```

N-Step A2C PyTorch Implementation sketch

```
# Enter a no-grad region when updating Actor and Crtic/Value network

# Actor and Critic/Value
# with torch.no_grad():
#     use the current value network to predict value of state t + n
#     use the current value network to predict value of state t

# Utilize the two values to compute the advantage and loss
```

```
# loss.detach().cpu().item() can be used to return the per step loss

# NOTE: To update value network we use mean squarred error loss.
# To update policy network we use compute the loss as the summation
# over timestep of the product of advantage and the log probabilities.
```