HOMEWORK 8: REINFORCEMENT LEARNING *

10-301 / 10-601 INTRODUCTION TO MACHINE LEARNING (FALL 2026)
https://www.mlcourse.org/10601

OUT: Sunday, November 15
DUE: Friday, November 20 at 11:59 PM
TAs: Sid, Marian, Arjuna, Shlok, Neural

Summary In this assignment, you will use reinforcement learning to train an agent to play the classic Atari game Pong. As a warm-up, the first section will lead you through an on-paper example of how policy iteration works for the algorithms we've seen in lecture. Then, in Section 9, you will implement the Advantage Actor Critic algorithm in PyTorch to train an agent for the Pong environment.

START HERE: Instructions

- **Collaboration Policy**: Please read the collaboration policy here: https://www.mlcourse.org/10601/syllabus
- **Late Submission Policy:** See the late submission policy here: https://www.mlcourse.org/10601/syllabus
- **Submitting your work:** You will use PhiGrade to submit answers to all questions and code. Please follow instructions at the end of this PDF to correctly submit all your code to PhiGrade.
  - **Written:** For written problems such as short answer, multiple choice, derivations, proofs, or plots, please use the provided template. Submissions can be handwritten onto the template, but should be labeled and clearly legible. If your writing is not legible, you will not be awarded marks. Alternatively, submissions can be written in LaTeX. Each derivation/proof should be completed in the boxes provided. You are responsible for ensuring that your submission contains exactly the same number of pages and the same alignment as our PDF template. If you do not follow the template, your assignment may not be graded correctly by our AI assisted grader and there will be a **2% penalty** (e.g., if the homework is out of 100 points, 2 points will be deducted from your final score).
  - **Programming:** You will submit your code for programming questions on the homework to PhiGrade. You run the autograder from your own machine: your program runs locally, and the output it produces is sent to PhiGrade, where it is compared against the reference solution. You are only permitted to use Pytorch modules (see writeup for fully authorized list), the Python Standard Library modules and `numpy`.

    Ensure that the version number of your programming language environment (i.e. Python 3.12.*)

*Compiled on 2026-11-15 at 20:20:25

and versions of permitted libraries (i.e. `numpy` 2.2.4) match those used on PhiGrade. Every run of the autograder is recorded as a submission, and your most recent one counts.

- **Materials:** The data and reference output that you will need in order to complete this assignment is posted along with the writeup and template on the course website.

Instructions for Specific Problem Types

For "Select One" questions, please fill in the appropriate bubble completely:

Select One: Who taught this course?

- ● Matt Gormley
- ○ Henry Chai
- ○ Noam Chomsky

If you need to change your answer, you may cross out the previous answer and bubble in the new answer:

Select One: Who taught this course?

- ● Matt Gormley
- ○ Henry Chai
- ⊗ Noam Chomsky

For "Select all that apply" questions, please fill in all appropriate squares completely:

Select all that apply: Which are instructors for this course?

- ■ Matt Gormley
- ■ Bryan Wilder
- □ Henry Chai
- □ I don't know

Again, if you need to change your answer, you may cross out the previous answer(s) and bubble in the new answer(s):

Select all that apply: Which are the instructors for this course?

- ■ Matt Gormley
- ■ Bryan Wilder
- ⊠ Henry Chai
- ⊠ I don't know

For questions where you must fill in a blank, please make sure your final answer is fully included in the given space. You may cross out answers or parts of answers, but the final answer must still be within the given space.

Fill in the blank: What is the course number?

10-601

10-~~6~~301

Written Questions (51 points)

1 LaTeX Point and Template Alignment (1 points)

1.1. (1 point) **Select one:** Did you use LaTeX for the entire written portion of this homework?

- ◯ Yes
- ◯ No

1.2. (0 points) **Select one:** I have ensured that my final submission is aligned with the original template given to me in the handout file and that I haven't deleted or resized any items or made any other modifications which will result in a misaligned template. I understand that incorrectly responding yes to this question will result in a penalty equivalent to 2% of the points on this assignment.
Note: Failing to answer this question will not exempt you from the 2% misalignment penalty.

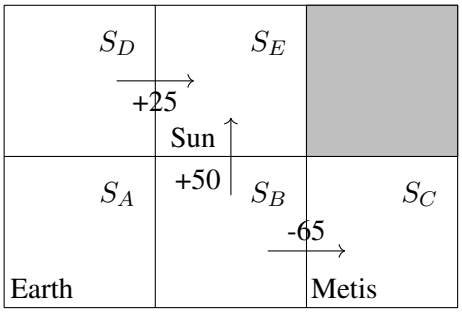
- ◯ Yes

1.3. (0 points) **Select one:** Did you fill out the Exit Poll for the previous HW? Completing the exit poll will count towards your participation grade.

- ◯ Yes

2 Policy Iteration (8 points)

While attending an ML conference, you meet scientists at NASA who ask you to develop a reinforcement learning agent capable of carrying out a space-flight from Earth to the Sun. You model this problem as a Markov decision process (MDP). The figure below depicts the state space.



Here are the details:

1. Each grid cell is a state $S_A, S_B, ..., S_E$ corresponding to a position in the solar system. The start state is S_A (Earth). The terminal states include both the S_E (Sun) and S_C (Metis).
2. The action space includes movement `up`/`down`/`left`/`right`. Transitions are **non-deterministic**. With probability 80% the agent transitions to the intended state. With probability 10% the agent slips left *of the intended direction*. With probability 10% the agent slips right *of the intended direction*. In other words, we can think of *the intended direction* as the direction *the agent faces*, so the notions of *left* and *right* are based on the direction the agent faces. For example, if the agent is in state S_B and takes action `left`, it moves to state S_A with 80% probability, it moves to state S_B (left of the intended direction is down off the board, so the agent remains where it was) with 10% probability, and it moves to state S_E (right of the intended direction) with 10% probability.
3. It is not possible to move to the blocked state (shaded grey) since it contains another planet. If the agent's action moves them off the board or to the blocked state, it remains in the same state.
4. Non-zero rewards are depicted with arrows. Flying into the Sun from below gives positive reward $R(S_B, \texttt{a}, S_E) = +50 \ \forall \texttt{a} \in \{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\}$, since it is more fuel-efficient than flying into the sun from the left (the agent can use the gravitational field of the planet in the blocked state and Metis). However, approaching the Sun from below has risks, as flying too close to Metis is inadvisable and gives negative reward $R(S_B, \texttt{a}, S_C) = -65 \ \forall \texttt{a} \in \{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\}$. Note that flying into the Sun from the left still achieves the goal and gives positive reward $R(S_D, \texttt{a}, S_E) = +25 \ \forall \texttt{a} \in \{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\}$. All other rewards are zero.

Below, let $V^*(s)$ denote the value function for state s using the optimal policy $\pi^*(s)$.

2.1. (2 points) Assume an initial policy that attempts to move up at every step. Report the value of each state (including terminal states) after a single step of **synchronous** policy iteration in the table below using the policy above. Initialize the value table $V^0(s) = 0, \forall s \in \{S_A \dots S_E\}$ and assume $\gamma = 0.9$. Visit each state in *reverse alphabetical order* (does this matter for synchronous policy iteration?). Ignore the blocked states. Round your *answers only* to the first decimal place. **Do not round intermediate values when calculating your answers.**

S_D	S_E	
S_A	S_B	S_C

2.2. (2 points) Update the policy $\pi(s)$ based on the results from the iteration of policy iteration above. Write one of up, down, left, or right for each state. If multiple actions are acceptable, choose the one that comes alphabetically first. For terminal states, write terminal. Ignore the blocked states.

S_D	S_E	
S_A	S_B	S_C

2.3. (2 points) Using your updated policy, run another step of policy iteration and report the update $V(s)$.

S_D	S_E	
S_A	S_B	S_C

2.4. (2 points) After many steps of policy iteration, we reach the value function below. What is the policy, $\pi^*(s)$, that corresponds to $V^*(s)$? Write one of `up`, `down`, `left`, or `right` for each state. If multiple actions are acceptable, choose the one that comes alphabetically first. For terminal states, write `terminal`. Ignore the blocked states.

S_D 25	S_E 0	
S_A 30	S_B 36	S_C 0

S_D	S_E	
S_A	S_B	S_C

3 Compare and Contrast (9 points)

Select all that apply: Each of the following methods estimates V^π from sampled episodes generated by following a fixed policy π in a finite MDP (all episodes reach a terminal state in a finite number of actions). For each statement below, select **all** methods for which it is true.

3.1. (1 point) The update to $V(s_t)$ cannot be performed until the episode has terminated.

- ☐ Monte Carlo
- ☐ TD(0)
- ☐ n-step TD
- ☐ None

3.2. (1 point) The update target bootstraps, i.e. it uses the current estimate V of a later state.

- ☐ Monte Carlo
- ☐ TD(0)
- ☐ n-step TD
- ☐ None

3.3. (1 point) The update target is an unbiased estimate of $V^\pi(s_t)$, even when the current estimate V is inaccurate.

- ☐ Monte Carlo
- ☐ TD(0)
- ☐ n-step TD
- ☐ None

3.4. (1 point) The update target has the lowest variance of the three methods, because it depends on the fewest random rewards and transitions.

- ☐ Monte Carlo
- ☐ TD(0)
- ☐ n-step TD
- ☐ None

3.5. (1 point) Has a tunable hyperparameter that interpolates between the bias–variance trade-offs of the other two methods.

- ☐ Monte Carlo
- ☐ TD(0)
- ☐ n-step TD
- ☐ None

3.6. (3 points) Recall that the advantage function is defined as

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$$

Prove that for every state s,

$$\mathbb{E}_{a\sim\pi(\cdot|s)}[A^\pi(s,a)] = 0$$

Your Answer

3.7. (1 point) Explain how the choice of discount factor γ affects the behavior of policy iteration. More specifically, what happens as $\gamma \to 0$ and as $\gamma \to 1$?

$\gamma \to 0$

$\gamma \to 1$

4 RL Algorithm Selection (8 points)

For each scenario below, select the **most appropriate** learning paradigm from the following options and **justify your choice in 2–4 sentences**.

Paradigms:

(A) Monte Carlo/TD

(B) Policy-Gradient/Actor-Critic RL

(C) Imitation Learning/Behavioral Cloning

(D) Do not use RL (supervised/unsupervised learning is more appropriate)

4.1. (2 points) A quantitative analyst wants to predict whether a given stock's price will be higher or lower tomorrow, using the past 30 days of price and volume data as features. They frame it as an RL problem: the "agent" observes the market state and "acts" by outputting a prediction; the "reward" is $+1$ if the prediction is correct and -1 otherwise.

◯ A ◯ B ◯ C ◯ D

Explanation

4.2. (2 points) A casino offers a house variant of blackjack. On each turn, the player sees its current card total, whether it holds an ace that can count as 11, and the dealer's face-up card, then chooses to hit (take a card), stand (stop), or double down (double the bet and take exactly one final card). The casino does not reveal how many decks are in its shoe or the exact rule its dealer follows, but both stay fixed, and the shoe is reshuffled after every hand. A developer's bot first plays one million hands at the casino's free play-money tables. Its strategy is then frozen and used unchanged at the real-money tables. Each hand pays $+1$ for a win, -1 for a loss, and 0 for a tie, doubled if the player doubled down. No records of expert players' decisions are available.

◯ A ◯ B ◯ C ◯ D

Explanation

4.3. (2 points) An energy company wants to train a controller that adjusts the blade pitch angle of a wind turbine in real time to maximize power output while minimizing mechanical stress. The blade pitch is a continuous value between $0°$ and $90°$. A high-fidelity physics simulator is available for training. A scalar reward (power generated minus a stress penalty) is easy to compute, but the aerodynamic transition dynamics are too complex to write down analytically.

◯ A ◯ B ◯ C ◯ D

Explanation

4.4. (2 points) A self-driving car team wants to teach their vehicle to execute smooth, human-like lane changes on a highway. They have dashcam recordings from 1,000 hours of expert human drivers. Attempts to hand-craft a reward function (penalize jerk, reward progress, etc.) have repeatedly produced unintended behaviors such as aggressive cutting-off of other vehicles. The action space is continuous (steering angle, throttle).

◯ A ◯ B ◯ C ◯ D

Explanation

5 Numerical REINFORCE Walkthrough (9 points)

A small campus delivery robot must decide how to move at each intersection while delivering snacks to students. At each timestep it observes a scalar state s_t, which summarizes how favorable the current path looks (for example, based on congestion, battery level, or obstacle density).

The robot can choose among three actions:

$$a_t \in \{L, R, S\},$$

where $L =$ Left, $R =$ Right, and $S =$ Straight.

For each action a, define the feature vector

$$x(s) = \begin{bmatrix} 1 \\ s \end{bmatrix},$$

and let the action-specific parameter vector be

$$\boldsymbol{\theta}_a = \begin{bmatrix} \theta_{a,1} \\ \theta_{a,2} \end{bmatrix}.$$

The score for action a at state s is

$$z_a(s) = \boldsymbol{\theta}_a^\top x(s) = \theta_{a,1} + \theta_{a,2} s.$$

We model the policy using a softmax over these scores:

$$\pi_\Theta(a \mid s) = \frac{e^{z_a(s)}}{\sum_{a' \in \{L,R,S\}} e^{z_{a'}(s)}}.$$

Let the full parameter vector be ordered as

$$\Theta = [\boldsymbol{\theta}_L^\top, \boldsymbol{\theta}_R^\top, \boldsymbol{\theta}_S^\top]^\top = [\theta_{L,1}, \theta_{L,2}, \theta_{R,1}, \theta_{R,2}, \theta_{S,1}, \theta_{S,2}]^\top \in \mathbb{R}^6.$$

It is often convenient to use the block feature vector $\phi(s, a) \in \mathbb{R}^6$, defined as

$$\phi(s, L) = \begin{bmatrix} 1 \\ s \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \phi(s, R) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ s \\ 0 \\ 0 \end{bmatrix}, \quad \phi(s, S) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ s \end{bmatrix}.$$

Assume the robot experiences the following episode of length $T = 3$:

$$(s_1, a_1, r_1) = (1, R, 2), \qquad (s_2, a_2, r_2) = (2, L, -1), \qquad (s_3, a_3, r_3) = (0, S, 3).$$

Assume all parameters are initially zero:

$$\Theta = 0.$$

Therefore, all three actions are initially equally likely at every state.

We use the REINFORCE gradient estimator

$$g = \sum_{t=1}^{T} Q_t \, \nabla_\Theta \log \pi_\Theta(a_t \mid s_t),$$

where the return-to-go (undiscounted) is

$$Q_t = \sum_{i=t}^{T} r_i.$$

5.1. (1 point) Compute the return-to-go values Q_1, Q_2, Q_3.

Q_1, Q_2, Q_3

5.2. (1 point) Under $\Theta = 0$, what are the action probabilities?

- ○ $\pi(L \mid s) = 1$
- ○ $\pi(a \mid s) = \frac{1}{2}$
- ○ $\pi(a \mid s) = \frac{1}{3}$
- ○ Depends on s

5.3. (1 point) What is the gradient form?

- ○ $\phi(s, a) + \sum_{a'} \pi(a' \mid s)\phi(s, a')$
- ○ $\phi(s, a) - \sum_{a'} \pi(a' \mid s)\phi(s, a')$
- ○ $\sum_{a'} \phi(s, a')$
- ○ $\pi(a \mid s)\phi(s, a)$

5.4. (1 point) Compute $Q_1 \nabla_\Theta \log \pi(R \mid 1)$.

Your Answer

5.5. (1 point) Compute $Q_2 \nabla_\Theta \log \pi(L \mid 2)$.

Your Answer

5.6. (1 point) Compute $Q_3 \nabla_\Theta \log \pi(S \mid 0)$.

Your Answer

5.7. (1 point) Compute g.

Your Answer

5.8. (1 point) Compute $\Theta' = \Theta + 0.1g$.

Your Answer

5.9. (1 point) After the update, which actions have probability greater than $\frac{1}{3}$ at each visited state?

Your Answer

6 Actor-Critic and Advantage Actor-Critic (A2C) (10 points)

Consider a reinforcement learning agent that parameterizes:

- A policy $\pi_\theta(a \mid s)$ (the **actor**)
- A value function $V_w(s)$ (the **critic**)

The agent interacts with an environment with discount factor $\gamma = 0.9$.

6.1. (1 point) **Select all that apply:** Which of the following statements about the actor and critic are correct?

- ☐ The actor directly estimates the value function $V(s)$.
- ☐ The critic evaluates how good the current policy is.
- ☐ The actor outputs a distribution over actions.
- ☐ The critic selects actions to maximize reward.

6.2. (1 point) **Select all that apply:** In actor-critic methods, what is the main purpose of the critic?

- ☐ To ensure exploration of all actions
- ☐ To compute exact Q-values for all (s, a)
- ☐ To reduce variance in policy gradient updates
- ☐ To replace the policy network

6.3. (2 points) Consider the following trajectory of length 2:

$$s_t \xrightarrow{a_t, r_t=1} s_{t+1} \xrightarrow{a_{t+1}, r_{t+1}=3} s_{t+2}$$

Given:

$$V(s_t) = 2.0, \quad V(s_{t+1}) = 2.5, \quad V(s_{t+2}) = 4.0, \quad \gamma = 0.9$$

1. Compute the 1-step TD error δ_t.
2. Compute the 2-step return:

$$G_t^{(2)} = r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2})$$

3. The 1-step TD return ($G_t^{(1)} = r_t + \gamma V(s_t + 1)$) and your 2-step return $G_t^{(2)}$ both estimate the same return. What happens as we keep increasing n in $G_t^{(n)} = r_t + \gamma r_{t+1} + \ldots + \gamma^{(n-1)} r_{t+n-1} + \gamma^n V(s_{t+n})$? How does the influence of value estimates compared to sampled rewards change during training?

δ_t

$G_t^{(2)}$

Your Answer

6.4. (1 point) **Select all that apply:** In Actor-Critic methods, when we say the agent "updates the actor" at time step t, we mean:

Adjusting the policy parameters so as to increase or decrease the probability of the action a_t taken in state s_t, based on feedback from the critic.

Which of the following quantities are used directly in this update?

- ☐ The TD error δ_t
- ☐ The reward r_t only
- ☐ The gradient of $\log \pi_\theta(a_t \mid s_t)$
- ☐ The transition probabilities $P(s' \mid s, a)$

6.5. (1 point) **Select all that apply:** Which of the following statements correctly describe differences between on-policy and off-policy methods?

- ☐ On-policy methods typically require fresh data collected from the current policy
- ☐ Off-policy methods can reuse data collected from older policies
- ☐ On-policy methods always converge faster than off-policy methods
- ☐ Off-policy methods require deterministic policies

6.6. (1 point) **Select all that apply:** Which of the following correctly defines the advantage function used in A2C?

- ☐ $A(s, a) = Q(s, a) - V(s)$
- ☐ $A(s, a) = r + \gamma V(s') - V(s)$
- ☐ $A(s, a) = V(s) - Q(s, a)$
- ☐ $A(s, a) = r$

6.7. (1 point) Suppose in A2C, you compute the advantage:

$$A_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

If $A_t > 0$, what should happen to the probability of action a_t?

- ☐ Increase
- ☐ Decrease
- ☐ Stay the same

6.8. (1 point) **Select all that apply:** What is a key benefit of using Advantage Actor-Critic (A2C) over REINFORCE?

- ☐ Lower variance updates
- ☐ Eliminates need for a value function
- ☐ Guarantees optimal policy
- ☐ Removes need for exploration

6.9. (1 point) **Select all that apply:** Which of the following statements about A2C are true?

- ☐ A2C uses multiple parallel environments.
- ☐ A2C updates its policy using advantage estimates.
- ☐ A2C directly selects actions using the value function.
- ☐ A2C reduces variance using a baseline.

7 Empirical Questions (6 points)

The following parts should be completed after you work through the programming portion of this assignment.

7.1. (3 points) Run the actor critic algorithm. Show the two plots you produced: the training and evaluation rewards over episodes.

Plot of Training Rewards over episodes

Plot of Testing Rewards over episodes

7.2. (3 points) Evaluate your agent after training to get videos of it playing Pong. Comment on your findings from the video. For example, What behaviors did it learn? What mistakes does it still make? How do you think these mistakes could be addressed?

Findings from video of the game

8 Collaboration Questions

After you have completed all other components of this assignment, report your answers to these questions regarding the collaboration policy. Details of the policy can be found here.

1. Did you receive any help whatsoever from anyone in solving this assignment? If so, include full details.
2. Did you give any help whatsoever to anyone in solving this assignment? If so, include full details.
3. Did you find or come across code that implements any part of this assignment? If so, include full details.

Your Answer

9 Programming [50 Points]

In this assignment, you will implement **Advantage Actor Critic (A2C)** to train an agent to play the Atari Pong game. You will implement all the functions needed to optimize the policy and value networks and to deploy your agent in the Pong environment. We provide a simplified version of the Pong environment for you to use. The program you write will be automatically graded using PhiGrade.

9.1 Libraries and Setup (IMPORTANT)

In this assignment, we **highly** recommend you use a conda environment, as we will be using several libraries (e.g. `gymnasium`, `ale_py`, `pytorch`). The handout provides an `environment.yml` that creates a conda environment called `601-hw8` with all of these libraries, plus `phigrade` and `pytest`:

```
$ conda env create -f environment.yml
$ conda activate 601-hw8
```

To deactivate the conda environment, use the command:

```
$ conda deactivate
```

Once you have created this environment, you can activate/deactivate this environment as much as you want without having to recreate it each time (assuming you did not also delete the environment).

9.2 The Pong Environment

In this assignment, you will be using the Arcade Learning Environment (ALE), a library with a set of Atari games that run on top of Gymnasium, a library for reinforcement learning. You are provided with code that wraps the ALE/Gymnasium Pong environment to simplify its API and to help you focus on the algorithm implementation.

In Pong, there are two paddles that bounce a ball back and forth from left to right. The agent will control the right paddle, and its objective is to continuously return the ball. The paddle that misses the ball first loses and receives a reward of -1, while the other wins and receives a reward of 1, and the episode terminates. There are no other rewards.

The state of the environment is represented by a vector of size 6 which contains:

- The position of your paddle along the y axis
- The position of the opponent's paddle along the y axis
- The position of the ball along the x axis
- The position of the ball along the y axis
- The velocity of the ball along the x axis
- The velocity of the ball along the y axis

The actions the agent can take at any state are defined as $\{0, 1\}$, corresponding to the actions: (0) move down, and (1) move up.

9.3 The Advantage Actor Critic Algorithm (A2C)

In the coding lab, you implemented REINFORCE and plotted returns across episodes. One pattern you should have noticed is that REINFORCE suffers from very high variance in returns. This is primarily because returns G_t and score vectors $\nabla_\theta \log \pi_\theta(a_t \mid s_t)$ can vary dramatically between trajectories even in

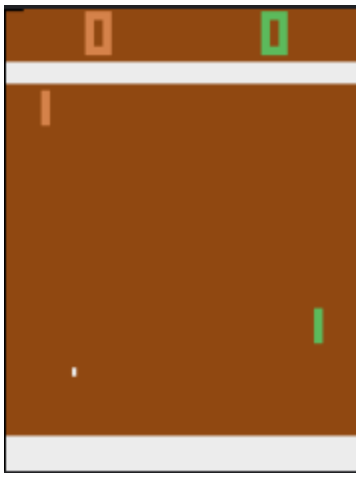


Figure 1: Rendered frame of the Pong environment. You play as the green paddle (to the right), and your opponent is the orange paddle (to the left). The white pixel in the bottom left is the ball.

consecutive episodes, leading to noisy gradient estimates and inefficient and unstable learning. **Advantage Actor-Critic (A2C)** primarily addresses these issues in two ways.

First, instead of using raw returns G_t, A2C uses an **advantage function**:

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t) \tag{1}$$

where $V(s_t)$ is the value function (expected return from state s_t) and $Q(s_t, a_t)$ is the Q-value function (expected return from taking action a_t at state s_t). The advantage measures how much better action a_t is compared to the policy's expected return from state s_t. Subtracting a baseline reduces variance without changing the expected policy gradient.

Second, A2C learns a separate **critic** network $V_\phi(s)$ to approximate the value function. As discussed in lecture, we often treat $Q_\phi(s_t, a_t) = r_t + \gamma V_\phi(s_{t+1})$. Then, we can estimate the advantage using:

$$A(s_t, a_t) \approx \hat{A}_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t) \tag{2}$$

Therefore, the gradient of our actor π_θ becomes

$$\nabla_\theta J(\theta) = \sum_t \hat{A}_t \nabla_\theta \log \pi_\theta(a_t \mid s_t) \tag{3}$$

and the gradient of our critic V_ϕ becomes

$$\nabla_\phi J(\phi) = \sum_t \nabla_\phi (r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t))^2 \tag{4}$$

This allows us to obtain gradient estimates for every step rather than once per full trajectory, improving sample efficiency.

9.3.1 *n*-Step TD Returns: Balance between bias and variance

While 1-step TD learning (using $Q(s_t, a_t) = r_t + \gamma V_\phi(s_{t+1})$) provides low-variance estimates, it can be biased when the value function is inaccurate. Conversely, Monte Carlo returns (full trajectories) are unbiased

but high-variance. Note that ***n*-step TD returns**, covered in lecture, provide a middle ground that can improve learning.

The **n-step TD return** from time t is defined as:

$$\hat{R}_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V_\phi(s_{t+n}) \tag{5}$$

This sums the actual rewards for n steps, then bootstraps from the value estimate at step $t+n$. The N-step advantage estimate thus becomes:

$$\hat{A}_t^{(n)} = \hat{R}_t^{(n)} - V_\phi(s_t) \tag{6}$$

Similarly, the critic's target becomes the *n*-step TD return $\hat{R}_t^{(n)}$ instead of the 1-step TD target.

Efficient PyTorch 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 `(B, T, 1)`, we can compute the *n*-step TD 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 (B, T, 1), rewards at timesteps t, t+1, ...
# next_state_values: tensor of shape (B, T, 1), estimated value
  for states at timesteps t+1, t+2, ...
# terminated: tensor of shape (B, T, 1), True for the terminal states
  in next_states_values
# gamma: scalar discount factor
# n_steps: number of steps for N-step return

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

# 1. Apply discount factors to the rewards
# 2. Pad discounted rewards with a zero on the left and n - 1 zeros
  to the right (total shape T + n)
# 3. Calculate the cumulative sum (the discounted cumulative rewards
  at every timestep)
# 4. Use cumsum[t+n] - cumsum[t], which calculates the sum
  of discounted rewards from timestep t -> t+n (our *discounted*
  n-step rewards, since we applied the discount factors in step 1)
# 5. Use the discount factors to bring every sum of discounted rewards back to
  their corresponding timestep value. (Larger timesteps t
  have their sum of rewards from t -> t+n discounted with additional
  discount factors. Can you understand why this is the case? Look at step 1.
  We want to discount the rewards s.t. reward_t * 1, reward_{t+1} * gamma,
  reward{t+2} * gamma^2 ... )
```

```
# 6. Get the value estimates at timesteps t+n for bootstrapping return for
  t+n onwards  (remember the first entry in next_state_values already is t + 1)
# 7. If value estimate s_{t+n} exists and is nonterminal, bootstrap
  using this estimate from the critic
# 8. If the episode had already terminated before reaching t+n
  or s_{t+n} does not exist, use a value of 0
# 9. Discount values using gamma and n
# (Think about how you can use padding to create tensors for these
  value estimates).

# 10. Add discounted rewards with values
```

In your implementation of `n_step_returns`, start by setting `n = min(n, T)`, where `n` is the value that you receive as an argument to the function and `T` is the second dimension of your rewards, which have shape `(B, T, 1)`

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.

Benefits of *n*-step TD returns:

- **Bias-variance tradeoff**: Larger n reduces bias (relying less on potentially inaccurate value estimates) but increases variance.
- **Faster credit assignment**: Rewards propagate n steps backward in a single update rather than just 1 step, accelerating learning in sparse-reward environments.
- **Better gradient quality**: Advantages based on actual observed rewards tend to provide more reliable policy gradient signals.

9.3.2 A2C Algorithm Summary

The **A2C algorithm** maintains two networks that are trained jointly:

- **Actor** $\pi_\theta(a|s)$: The policy network, parameterized by θ. Updated using policy gradients with N-step advantage estimates:
$$\theta \leftarrow \theta + \eta_\theta \nabla_\theta J(\theta) \tag{7}$$
where $\nabla_\theta J(\theta) = \sum_t \hat{A}_t^{(n)} \nabla_\theta \log \pi_\theta(a_t \mid s_t)$
- **Critic** $V_\phi(s)$: The value network, parameterized by ϕ. Updated to minimize the squared error between predictions and N-step returns:
$$\phi \leftarrow \phi - \eta_{value} \nabla_\phi J(\phi) \tag{8}$$
where $\nabla_\phi J(\phi) = \sum_t \nabla_\phi (\hat{R}_t^{(n)} - V_\phi(s_t))^2$

In both of these networks, $\hat{A}_t^{(n)} = \hat{R}_t^{(n)} - V_\phi(s_t)$ and $\hat{R}_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V_\phi(s_{t+n})$

Both networks need to be updated synchronously after accumulating gradients over a batch of collected trajectories.

9.4 File Structure

The handout contains the following files:

- `environment.py`: Pong Environment (provided for you)
- `agent.py`: File in which you will implement the A2C algorithm
- `utils.py`: Utility functions for plotting rewards and setting random seeds (provided for you)
- `test_runner.py`: Script to run unit tests to debug your implementation
- `test_cases`: Unit tests
- `test_data`: Auxiliary data for the unit tests
- `test_agent.py`: PhiGrade test suite used to grade and submit your code
- `environment.yml`: Conda environment for the assignment

9.5 TODOs

You will need to implement the following in `agent.py`

- `Policy`: Pytorch module that models the policy as a neural network
- `Value`: Pytorch module that models the value function as a neural network
- `Agent.get_action`: Method that samples an action from the policy
- `Agent.n_step_returns`: Method that calculates the N-Step Returns given a sequence of rewards and value estimates
- `Agent.policy_loss`: Method that calculates the loss we will minimize to optimize the parameters of the policy network
- `Agent.value_loss`: Method that calculates the loss we will minimize to optimize the parameters of the value network
- `Agent.update_policy`: Method that takes a step of gradient descent to update the parameters of the policy network
- `Agent.update_value`: Method that takes a step of gradient descent to update the parameters of the value network
- `deploy_agent`: Function that deploys the agent in the Pong environment

There are other initializations you need to implement which are marked with TODOs in `agent.py`. You need to complete those TODOs to train and evaluate your agent; see the starter code for details.

9.6 The Pong Environment API

The Pong environment provides the following API to interact with it.

- `__init__(max_steps, record)`: Initializes the environment such that each episode takes at most `max_steps` number of steps. Setting `record = True` allows you to record and save a video of a game. Only set `record = True` when evaluating the agent, otherwise training will be too slow.
- `reset()`: Resets the environment and returns the initial state.

- `step(action)`: Take a step in the environment with the given action. `action` must be 0 (down) or 1 (up). `step(action)` returns a tuple of (`state`, `reward`, `terminated`, `truncated`) which are the next state, the reward observed, a boolean indicating whether the episode has terminated, and a boolean indicating if we reached the maximum number of steps. The `state` will be the new state that the agent is in after taking its specified action. If you observe `terminated = True` or `truncated = True` then you should `reset` the environment and end the episode. Failure to do so will result in undefined behavior.

9.7 Implementation Details

Policy and Value Networks You will implement two networks, a policy network and a value network, each with exactly two hidden linear layers of size 256 (any other architectures will not pass the autograder tests). Use ReLU activations in between layers.

The policy network should output action logits, not probabilities. Do **not** apply a softmax activation on the outputs of the network (this will cause numerical instability later when we calculate the log probabilities for the policy loss implementation).

Sampling actions from the policy Use `torch.nn.functional.softmax` and `torch.multinomial` to sample stochastically from the logits returned by the policy network.

N-Step Returns efficient implementation This function should implement equation 5, with $n = 10$. Pay special attention to how you are supposed to handle values for terminal states. If s_t is a terminal state, you should use $V(s_t) = 0$ (hint: remember that the `step()` method in the environment returns whether the new state is a terminal state). If the trajectory ends before $t + n$, only sum the rewards that exist.

Policy Loss This function should return $-\hat{A}_t^{(n)} \log \pi_\theta(a_t|s_t)$, so that when we perform backpropagation on this loss we get the gradient that shows up at end of equation 7. Remember that gradient descent is used to minimize an objective function. Therefore if we want to maximize the advantage weighted log probabilities, we can minimize the negative advantage weighted log probabilities.

To calculate the advantages, you will need to calculate the N-Step Returns and the value for the current state. Call your value network to get value estimates for the current and next states. In this step, we are only using the value network to get advantages, so we do **NOT** want to propagate gradients through it (Hint: use `torch.no_grad()`).

Once you have the advantages you are going to need the log probabilities of the actions taken. Directly applying a softmax followed by a logarithm can lead to numerical instability when logits have large magnitude. PyTorch provides a stable implementation through `torch.nn.functional.log_softmax()`.

Value Loss This function should return $\left(\hat{R}_t^{(n)} - V_\phi(s_t)\right)^2$, so that when we perform backpropagation on this loss we get the gradient that shows up at end of equation 8: .

To calculate the loss, you will need the N-Step Returns and the value for the current state. Call your value network to get value estimates for the current and next states. Here we only want to backpropagate through the value estimates for the **current** state, not for other states (Hint: `torch.no_grad()`). You can use `torch.nn.functional.mse_loss` to get the average MSE across all timesteps.

Update Policy and Value In this function, you need to use the optimizers to take a gradient descent step and then make sure to set all gradients to zero. Only set gradients to zero inside this function, which gets called after several backpropagation calls. In this assignment we are accumulating gradients over several trajectories to make learning more stable. We only want to set gradients to zero *after* taking a step with the optimizer.

Deploying the Agent Review `environment.py` for guidance on how to interact with the environment. The handout has comments that will guide you through the implementation of this function. Pay close attention to the data types and shapes of the objects this function should return. Ignoring this will cause problems in other parts of your code. In `deploy_agent`, please use a `LongTensor` for the actions `(dtype=torch.int64)`

9.8 Running the Agent

```
$ python agent.py [args...]
```

where above `[args...]` is a placeholder for command-line arguments: `<max-steps> <gamma> <train-episodes> <lr> <batch-size> <store-every> <eval-only> <eval-every> <eval-episodes>`. These arguments are described in detail below:

1. `<max-steps >`: Maximum steps per episode (default: 300)
2. `<gamma>`: Discount factor (default: 0.98).
3. `<train-episodes>`: Total number of training episodes (default: 30000).
4. `<lr>`: Learning rate (default: 0.0003)
5. `<batch-size>`: Episodes for each policy and value network update (default: 4).
6. `<store-every>`: Episodes between each checkpoint (default: 2000).
7. `<eval-only>`: Flag to record a video of agent playing (default: False). If set to True, the agent will not be trained. Use only after training an agent.
8. `<eval-every>`: Training episodes between evaluation episodes (default: 500).
9. `<eval-episodes>`: Number of episodes per evaluation (default: 20).

Run the following command to train your agent (use the default hyperparameters). This should take approximately 30 minutes to run on a CPU.

```
$python agent.py
```

Your program should produce two plots, one with the training rewards for every episode, and another one with the mean evaluation rewards across `eval-episodes` sampled every `eval-every` training episodes.

After training your agent, run the following command to record a set of videos of the agent playing Pong!

```
$python agent.py --eval-only
```

Your program will print which video had the longest game in which your agent won. Feel free to skim over the other videos as well to answer the empirical questions.

9.9 Testing

The autograder will use the following commands to test your implementation:

```
$python agent.py 300 0.98 30000 0.0003 4 2000 False 500 20
```

9.10 Submission Instructions

Please ensure you have completed the following files for submission.

`agent.py`

Submission happens through PhiGrade by running the test suite from inside the handout folder. There is no file upload. Create an API token on your PhiGrade account page, then, from the handout folder:

```
$ conda env create -f environment.yml   # once
$ conda activate 601-hw8
$ export PHIGRADE_API_KEY="<YourAPIToken>"
$ pytest test_agent.py
```

Your code runs locally and only its output is uploaded, so `agent.py` must sit in the folder you run `pytest` from, alongside the provided `environment.py` and `utils.py`. Do *not* use other file names.

Every run of `pytest` is a submission, and your most recent one counts. The first run asks which slot to submit to: **Slot A** (for human work only) or **Slot B** (for human or AI work). Set `PHIGRADE_SLOT` to skip the prompt on later runs.