# HOMEWORK 8: REINFORCEMENT LEARNING *

10-301 / 10-601 INTRODUCTION TO MACHINE LEARNING (FALL 2025)
http://www.cs.cmu.edu/~mgormley/courses/10601/

OUT: Sunday, November 16
DUE: Monday, November 24 at 11:59 PM
TAs: Santiago, Akhil, Doris, Joyce, Markov, Neural

**Summary** In this assignment, you will use reinforcement learning to train an agent to play the classic Atari game Pong. As a warmup, the first section will lead you through an on-paper example of how value iteration and Q-learning work. Then, in Section 6, you will implement the Advantage Actor Critic algorithm in pytorch to train an agent for the Pong enviroment.

## START HERE: Instructions

- **Collaboration Policy**: Please read the collaboration policy here: http://mlcourse.org/index.html#7-collaboration-and-academic-integrity-policies
- **Late Submission Policy:** See the late submission policy here: http://mlcourse.org/index.html#6-general-policies
- **Submitting your work:** You will use Gradescope to submit answers to all questions and code. Please follow instructions at the end of this PDF to correctly submit all your code to Gradescope.
  - **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 Gradescope. After uploading your code, our grading scripts will autograde your assignment by running your program on a virtual machine (VM). 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.*) and versions of permitted libraries (i.e. `numpy` 2.2.4) match those used on Gradescope. You

*Compiled on 2025-11-17 at 16:12:25

have 10 free Gradescope programming submissions, after which you will begin to lose points from your total programming score. We recommend debugging your implementation on your local machine (or the Linux servers) and making sure your code is running correctly first before submitting your code to Gradescope.

- **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
- ■ Geoff Gordon
- □ 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
- ■ Geoff Gordon
- ⊠ 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 (30 points)

## 1 LaTeX Point and Template Alignment (1 points)

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

   ◯ Yes

   ◯ No

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.

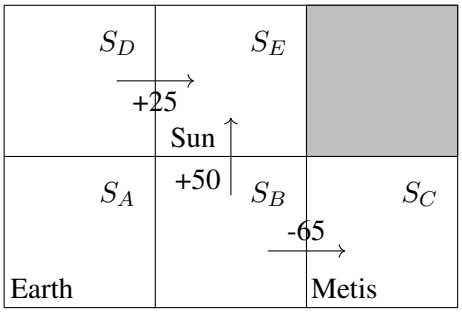
   ◯ Yes

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 Value Iteration (12 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 Synchronous Value Iteration

1. (3 points) Report the value of each state (including terminal states) after a single round of **synchronous** value iteration in the table below. 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 value 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 Asynchronous Value Iteration

1. (3 points) Starting over, report the value of each state for a single round of **asynchronous** value iteration in the table below. Initialize the value table $V(s) = 0, \forall s \in \{S_A \dots S_E\}$ and assume $\gamma = 0.9$. Visit each state in *reverse alphabetical order*. Ignore the blocked states. Round your *answers only* to the first decimal place. **Do not round any intermediate values, including state values, when calculating your answers.**

| | | |
|---|---|---|
| $S_D$ | $S_E$ | |
| $S_A$ | $S_B$ | $S_C$ |

2. (3 points) Below, we give you the value of each state one round before the convergence of **asynchronous** value iteration.[1] What is the value of each state, $V'(s)$, after another round of value iteration? Be sure to use **asynchronous** value iteration, and visit each state in *reverse alphabetical order*. Ignore the blocked states. Round your *answers only* to the first decimal place. **Do not round any intermediate values, including state values, when calculating your answers.**

| | | |
|---|---|---|
| $S_D$ 25 | $S_E$ 0 | |
| $S_A$ 30 | $S_B$ 36 | $S_C$ 0 |

**Your solution:**

| | | |
|---|---|---|
| $S_D$ | $S_E$ | |
| $S_A$ | $S_B$ | $S_C$ |

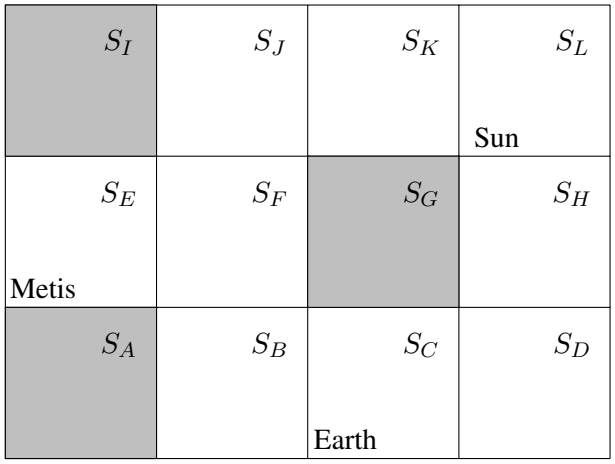
3. (3 points) What is the policy, $\pi^*(s)$, that corresponds to $V'(s)$? Write one of `up`, `down`, `left`, or `right` for each state. If multiple actions are acceptable, choose the one that comes alphabetically first. For terminal states, write `terminal`. Ignore the blocked states.

| | | |
|---|---|---|
| $S_D$ | $S_E$ | |
| $S_A$ | $S_B$ | $S_C$ |

[1] This is actually one round before the *policy* convergence, not the *value* convergence. The values we provide are the values after the second iteration, rounded to the nearest whole number for ease of calculation.

# 3 Q-Learning (9 points)

Let's consider an environment that is similar to the grid world we saw before, but has more states:




This time, however, suppose we **don't know** the reward function or the transition probability between states. Some rules for this setup are:

1. Each grid cell is a state $S_A, S_B, \ldots, S_L$ corresponding to a position in the solar system.
2. The action space of the agent is: $\{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\}$.
3. If the agent hits the edge of the board, it remains in the same state. It is not possible to move into blocked states, which are shaded grey, since they contain other planets.
4. The start state is $S_C$ (Earth). The terminal states include both the $S_L$ (Sun) and $S_E$ (asteroid Metis).
5. Use the discount factor $\gamma = 0.9$ and learning rate $\alpha = 0.1$.

We will go through three iterations of Q-learning in this section. Initialize $Q(s, a)$ as below:

| $a \setminus s$ | $S_A$ | $S_B$ | $S_C$ | $S_D$ | $S_E$ | $S_F$ | $S_G$ | $S_H$ | $S_I$ | $S_J$ | $S_K$ | $S_L$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Up | 0.4 | 0.1 | 0.1 | 0.7 | 0.0 | 0.9 | 0.7 | 0.8 | 0.0 | 0.1 | 0.8 | 0.8 |
| Down | 1.0 | 0.8 | 0.2 | 0.5 | 0.1 | 0.2 | 0.7 | 0.2 | 1.0 | 0.9 | 0.1 | 0.3 |
| Left | 0.9 | 0.4 | 0.3 | 0.4 | 0.9 | 0.6 | 0.5 | 0.1 | 0.2 | 0.3 | 0.9 | 0.1 |
| Right | 0.3 | 0.8 | 0.3 | 0.2 | 0.0 | 0.2 | 0.2 | 0.3 | 0.9 | 0.4 | 0.2 | 0.3 |

1. (1 point) **Select all that apply:** If the agent were to act greedily, what action(s) could it take at this time from state $S_C$?

   ☐ `up`

   ☐ `down`

   ☐ `left`

   ☐ `right`

2. (1 point) Beginning at state $S_C$, you take the action `right` and receive a reward of 0. You are now in state $S_D$. What is the new value for $Q(S_C, \texttt{right})$, assuming the update for deterministic transitions? If needed, round your answer to the fourth decimal place.

$Q(S_C, \texttt{right})$

3. (1 point) What is the new value for $Q(S_C, \texttt{right})$, using the temporal difference error update? If needed, round your answer to the fourth decimal place.

$Q(S_C, \texttt{right})$

4. (1 point) **Select all that apply:** Assume your run has brought you to state $S_H$ with no updates to the Q-function in the process. If the agent were to act greedily, what action(s) could it take at this time?

   ☐ `up`

   ☐ `down`

   ☐ `left`

   ☐ `right`

5. (1 point) Beginning at state $S_H$, you take the action `up`, receive a reward of +25, and the run terminates. What is the new value for $Q(S_H, \texttt{up})$, assuming the update for deterministic transitions? If needed, round your answer to the fourth decimal place.

$Q(S_H, \texttt{up})$

6. (1 point) What is the new value for $Q(S_H, \texttt{up})$, using the temporal difference error update? If needed, round your answer to the fourth decimal place.

$Q(S_H, \texttt{up})$

7. (1 point) **Select all that apply:** You start from state $S_C$ again since the previous run terminated. Assume you manage to make it to state $S_F$ with no updates to the Q-function. If the agent were to act greedily, what action(s) could it take at this time?

   - ☐ `up`
   - ☐ `down`
   - ☐ `left`
   - ☐ `right`

8. (1 point) Beginning at state $S_F$, you take the action `left`, receive a reward of -50, and the run terminates. What is the new value for $Q(S_F, \texttt{left})$, assuming the update for deterministic transitions? If needed, round your answer to the fourth decimal place.

   $Q(S_F, \texttt{left})$

9. (1 point) What is the new value for $Q(S_F, \texttt{left})$, using the temporal difference error update? If needed, round your answer to the fourth decimal place.

   $Q(S_F, \texttt{left})$

# 4 Empirical Questions (8 points)

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

1. (4 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

2. (4 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

## 5 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

# 6 Programming [68 Points]

Your goal in this assignment is to implement the Advantage Actor Critic (A2C) algorithm 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 Gradescope.

## 6.1 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`). You can set up a conda environment called "HW8" as such:

```
$ conda create -n HW8 python=3.12
$ conda activate 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).

We have provided a `requirements.txt` file to make library installation straightforward. To install the libraries, use the command:

```
$ pip install -r requirements.txt
```

## 6.2 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. Specifically, your goal is to train an agent to play Pong, one of the most popular Atari games. 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. You play as the paddle on the right, and your 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.

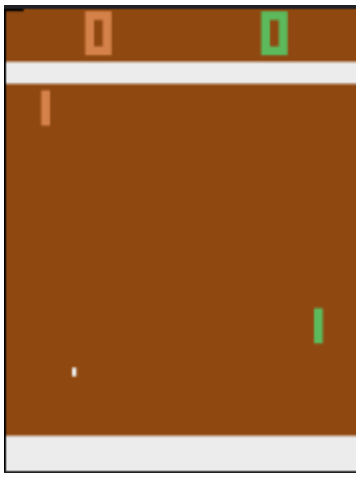


Figure 1: Rendered frame of the Pong environment. You play as the green paddle (to the right), and your opponent is the orange paddle (to the left). The white pixel in the bottom left is the ball.

## 6.3 The Advantage Actor Critic Algorithm (A2C)

### 6.3.1 Background: The Reinforcement Learning Paradigm

In reinforcement learning (RL), we face a fundamental challenge that distinguishes it from supervised learning: *we do not have labeled data*. Instead of receiving input-output pairs $(x, y)$, an RL agent must learn from sequential interactions with an environment, receiving only sparse reward signals that may be delayed in time (maybe an action taken at timestep $t$ leads to high rewards at timestep $t + 1000$). With this information, the agent must discover which actions lead to desirable outcomes through trial and error.

### 6.3.2 The RL Objective: Expected Reward Maximization

The goal in RL is to find a policy $\pi_\theta(a|s)$ that maximizes the expected cumulative discounted reward:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[ \sum_{t=0}^{T} \gamma^t r_t \right] = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] \tag{1}$$

where $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots)$ is a trajectory or episode, $\gamma \in [0, 1]$ is the discount factor, and the expectation is taken over trajectories or episodes obtained by deploying the policy $\pi_\theta$ on the environment.

### 6.3.3 The REINFORCE Algorithm

To optimize $J(\theta)$ using gradient ascent, we need to compute $\nabla_\theta J(\theta)$. Using the likelihood ratio trick, we can derive:

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] \tag{2}$$
$$= \nabla_\theta \sum_\tau [R(\tau)p_\theta(\tau)] \tag{3}$$
$$= \sum_\tau [R(\tau)\nabla_\theta p_\theta(\tau)] \tag{4}$$
$$= \sum_\tau [R(\tau)p_\theta(\tau)\frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)}] \tag{5}$$
$$= \sum_\tau [R(\tau)p_\theta(\tau)\nabla_\theta \log p_\theta(\tau))] \tag{6}$$
$$= \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)\nabla_\theta \log p_\theta(\tau)] \tag{7}$$
$$= \mathbb{E}_{\tau\sim\pi_\theta}\left[R(\tau)\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\right] \tag{8}$$
$$= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{t'=t}^{T}\gamma^{t'}r_{t'}\right] \tag{9}$$

The last step uses the fact that actions at time $t$ cannot influence rewards received before time $t$. Defining the return from time $t$ as $G_t = \sum_{t'=t}^{T}\gamma^{t'-t}r_{t'}$, we obtain the **REINFORCE policy gradient**:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\cdot G_t\right] \tag{10}$$

In practice, we estimate this expectation by sampling trajectories (also known as Monte Carlo estimation).

#### 6.3.4 From REINFORCE to A2C

While REINFORCE is theoretically sound, it suffers from high variance. The returns $G_t$ and score vectors $\nabla_\theta \log \pi_\theta$ can vary dramatically between trajectories, leading to noisy gradient estimates and slow, unstable learning.

**Advantage Actor-Critic (A2C)** addresses this issue through baseline subtraction. Instead of using raw returns $G_t$, A2C uses the **advantage function**:

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t) \tag{11}$$

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 average action in state $s_t$. Replacing raw returns with advantages reduces variance without introducing bias because $\mathbb{E}[V(s_t)]$ is constant with respect to $a_t$.

The gradient becomes:

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\cdot A(s_t, a_t)\right] \tag{12}$$

To estimate advantages, A2C learns a separate **critic** network $V_\phi(s)$ to approximate the value function. We can then estimate the advantage using:

$$A(s_t, a_t) \approx r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t) \tag{13}$$

Remember that $Q(s_t, a_t) = r_t + \gamma V(s_{t+1})$ where $r_t$ is the reward obtained after taking action $a_t$ at state $s_t$. This allows us to obtain a gradient estimate for every step rather than once per full trajectory, improving sample efficiency.

#### 6.3.5 N-Step Returns: Balance between bias and variance

While 1-step TD learning (using $r_t + \gamma V_\phi(s_{t+1})$) 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. **N-step returns** provide a middle ground that can improve learning.

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

$$G_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V_\phi(s_{t+n}) \tag{14}$$

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

$$A_t^{(n)} = G_t^{(n)} - V_\phi(s_t) = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V_\phi(s_{t+n}) - V_\phi(s_t) \tag{15}$$

**Benefits of N-step 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.

Similarly, the critic's target becomes the N-step return $G_t^{(n)}$ instead of the 1-step TD target.

#### 6.3.6 A2C Algorithm Summary

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

- **Actor** $\pi_\theta(a|s)$: The policy network, parameterized by $\theta$. Updated using policy gradients with N-step advantage estimates:

$$\theta \leftarrow \theta + \eta_{\textit{policy}} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot A_t^{(n)} \tag{16}$$

- **Critic** $V_\phi(s)$: The value network, parameterized by $\phi$. Updated to minimize the squared error between predictions and N-step returns:

$$\phi \leftarrow \phi - \eta_{\textit{value}} \nabla_\phi \left( V_\phi(s_t) - G_t^{(n)} \right)^2 \tag{17}$$

Both networks need to be updated synchronously after collecting experience for $B$ episodes.

### 6.4 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
- `requirements.txt`: Dependencies for the assignment

### 6.5 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.

### 6.6 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.

### 6.7 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 return the action logits. 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).

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

**N-Step Returns efficient implementation** Review the HW8 recitation for instructions/hints on how to calculate the N-Step Returns. This function should implement equation 14. 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)

**Policy Loss** This function should return $-\log \pi_\theta(a_t|s_t) \cdot A_t^{(n)}$, so that when we perform backpropagation on this loss we get the gradient that shows up at end of equation 16. 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. Review the HW8 recitation for instructions/hints on how to stop the gradient propagation through the value network. Here we are using the value network only to get the advantages. We do not want to back-propagate the loss through the value network on this function. Use a value of $N = 10$ for the N-Step Returns.

Once you have the advantages you are going to need the log probabilities of the actions taken. Review the HW8 recitation for instructions/hints on how to calculate these in a numerically stable way.

**Value Loss** This function should return $\left(V_\phi(s_t) - G_t^{(n)}\right)^2$, so that when we perform backpropagation on this loss we get the gradient that shows up at end of equation 17: .

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. Review the HW8 recitation for instructions/hints on how to stop the gradient propagation. Here we only want to back-propagate through the value estimates for the current state. Use a value of $N = 10$ for the N-Step Returns. You can use `torch.nn.functional.mse_loss` to get the average MSE for all timesteps.

**Update Policy and Value** This methods are really simple to implement. You just need to use the optimizers to take a gradient descent step and then make sure to set all gradients to zero. Only set gradient to zero inside this function, which gets called after several back propagation 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.

### 6.8 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
```

You 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.

### 6.9 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
 \ 4 200 0.05 0.99 0.01 1 500
```

### 6.10 Gradescope Submission

You should submit your `agent.py` to Gradescope. **Any other files uploaded will be discarded or reverted back to the original version provided in the handout.** Do *not* use other file names.