CS 234 Winter 2023: Assignment #2

Due date:
February 3, 2023 at 6 PM (18:00) PST

These questions require thought, but do not require long answers. Please be as concise as possible.

We encourage students to discuss in groups for assignments. We ask that you abide by the university Honor Code and that of the Computer Science department. If you have discussed the problems with others, please include a statement saying who you discussed problems with. Failure to follow these instructions will be reported to the Office of Community Standards. We reserve the right to run a fraud-detection software on your code. Please refer to website, Academic Collaboration and Misconduct section for details about collaboration policy.
Please review any additional instructions posted on the assignment page. When you are ready to submit, please follow the instructions on the course website. **Make sure you test your code using the provided commands and do not edit outside of the marked areas.** Also note that **all submissions must be typeset. No handwritten submissions will be accepted.** We have released a LaTeX template for your convenience, but you may use any typesetting program of your choice, including, e.g., Microsoft Word.

You'll need to download the starter code and fill the appropriate functions following the instructions from the handout and the code's documentation. Training one seed of our network on MinAtar's Breakout takes roughly **1.5 hours on a laptop's CPU**, and we require 3 seeds, so **please start early**! (Only completed runs will recieve full credit) This year, we will not provide access to cloud GPU credits.

Introduction

In this assignment, we will first implement a modified version of deep Q-learning from DeepMind's paper ([1] and [2]) that learns to play Atari games from raw pixels, and then consider a few theoretical questions about Q-learning. The purpose is to demonstrate the effectiveness of deep neural networks as well as some of the techniques used in practice to stabilize training and achieve better performance. In the process, you'll become familiar with PyTorch. We will train our networks on a simplified version of the Atari game `Breakout-v0` such that we can run the code without using a GPU, but the code can easily be applied to any other environment.

In Breakout, the player controls a bar that can move horizontally, and gets rewards by bouncing a ball into bricks, breaking them. We are going to use MinAtar ([3]), a miniaturized version of the original Atari game. Instead of the original 210×160 RGB image resolution, MinAtar uses a 10×10 boolean grid, which makes it possible to use a significantly smaller model and still get a good performance.

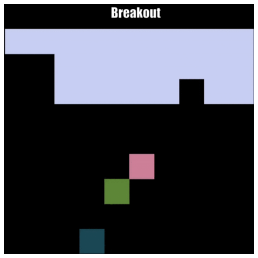


Figure 1: MinAtar's version of Breakout

1 Test Environment (6 pts)

Before running our code on Breakout, it is crucial to test our code on a test environment. In this problem, you will reason about optimality in the provided test environment by hand; later, to sanity-check your code, you will verify that your implementation is able to achieve this optimality. You should be able to run your models on CPU in no more than a few minutes on the following environment:

- 4 states: $0, 1, 2, 3$
- 5 actions: $0, 1, 2, 3, 4$. Action $0 \leq i \leq 3$ goes to state i, while action 4 makes the agent stay in the same state.
- Rewards: Going to state i from states 0, 1, and 3 gives a reward $R(i)$, where $R(0) = 0.1, R(1) = -0.3, R(2) = 0.0, R(3) = -0.2$. If we start in state 2, then the rewards defind above are multiplied by -10. See Table 1 for the full transition and reward structure.
- One episode lasts 5 time steps (for a total of 5 actions) and always starts in state 0 (no rewards at the initial state).

State (s)	Action (a)	Next State (s')	Reward (R)
0	0	0	0.1
0	1	1	-0.3
0	2	2	0.0
0	3	3	-0.2
0	4	0	0.1
1	0	0	0.1
1	1	1	-0.3
1	2	2	0.0
1	3	3	-0.2
1	4	1	-0.3
2	0	0	-1.0
2	1	1	3.0
2	2	2	0.0
2	3	3	2.0
2	4	2	0.0
3	0	0	0.1
3	1	1	-0.3
3	2	2	0.0
3	3	3	-0.2
3	4	3	-0.2

Table 1: Transition table for the Test Environment

An example of a trajectory (or episode) in the test environment is shown in Figure 2, and the trajectory can be represented in terms of s_t, a_t, R_t as: $s_0 = 0, a_0 = 1, R_0 = -0.3, s_1 = 1, a_1 = 2, R_1 = 0.0, s_2 = 2, a_2 = 4, R_2 = 0.0, s_3 = 2, a_3 = 3, R_3 = 2.0, s_4 = 3, a_4 = 0, R_4 = 0.1, s_5 = 0$.

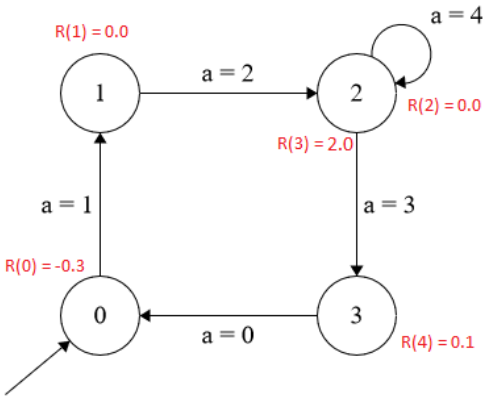


Figure 2: Example of a trajectory in the Test Environment

What is the maximum sum of rewards that can be achieved in a single trajectory in the test environment, assuming $\gamma = 1$? Show first that this value is attainable in a single trajectory, and then briefly argue why no other trajectory can achieve greater cumulative reward.

Solution: The maximum sum of rewards that can be achieved in a single trajectory in the test environment is 6.1 in 5 actions. The trajectory that yields this return exploits the return gained from transitioning from state 2 to 1: $s_0 = 0, a_0 = 2, R_0 = 0, s_1 = 2, a_1 = 1, R_1 = 3, s_2 = 1, a_2 = 2, R_2 = 0, s_3 = 2, a_3 = 1, R_3 = 3, s_4 = 1, a_4 = 0, R_4 = 0.1, s_5 = 0$. Its clear no other trajectory yields a larger reward because of large of an outlier action 1 in state 2 is. Since we dont have enough turns to return to state 2, we simply take the best action we can from state 3 which is 0.1. □

2 Setting up the environment (0 pts)

Create a new virtualenv or a Conda environment (to set up Conda follow this guide) with python 3.8 or above. The environment files are provided in the starter code folder for different operating systems. `cd` into the starter code folder, and create a conda environment using the following:

```
conda env create -f cs234-torch-<your-system>.yml
conda activate cs234-torch
```

Then install the package requirements by running "`pip install -r requirements.txt`" on a terminal. To train on MinAtar, we first need to install it. Clone the GitHub repo by running `git clone https://github.com/kenjyoung/MinAtar.git` on a terminal, then `cd` into MinAtar and run "`pip install .`". Line-by-line instructions on how to create the environment and set it up can be found on the README.md file of the assignment starter code.

3 Tabular Q-Learning (3 pts)

If the state and action spaces are sufficiently small, we can simply maintain a table containing the value of $Q(s, a)$, an estimate of $Q^*(s, a)$, for every (s, a) pair. In this *tabular setting*, given an experience sample

(s, a, r, s'), the update rule is

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} Q(s', a') - Q(s,a) \right) \tag{1}$$

where $\alpha > 0$ is the learning rate, $\gamma \in [0, 1)$ the discount factor.

ϵ-Greedy Exploration Strategy For exploration, we use an ϵ-greedy strategy. This means that with probability ϵ, an action is chosen uniformly at random from $\mathcal{A}$, and with probability $1 - \epsilon$, the greedy action (i.e., $\arg\max_{a \in \mathcal{A}} Q(s, a)$) is chosen.

(a) (**coding**, 3 pts) Implement the `get_action` and `update` functions in `q3_schedule.py`. Test your implementation by running `python q3_schedule.py`.

Solution: See code

4 Linear Approximation (23 pts)

Due to the scale of Atari environments, we cannot reasonably learn and store a Q value for each state-action tuple. We will instead represent our Q values as a parametric function $Q_{\mathbf{w}}(s, a)$ where $\mathbf{w} \in \mathbb{R}^p$ are the parameters of the function (typically the weights and biases of a linear function or a neural network). In this *approximation setting*, the update rule becomes

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} Q_{\mathbf{w}}(s', a') - Q_{\mathbf{w}}(s, a) \right) \nabla_{\mathbf{w}} Q_{\mathbf{w}}(s, a) \tag{2}$$

where (s, a, r, s') is a transition from the MDP.

(a) (**written**, 5 pts) Let $Q_{\mathbf{w}}(s, a) = \mathbf{w}^\top \delta(s, a)$ be a linear approximation for the Q function, where $\mathbf{w} \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$ and $\delta : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$ with

$$[\delta(s,a)]_{s',a'} = \begin{cases} 1 & \text{if } s' = s, a' = a \\ 0 & \text{otherwise} \end{cases}$$

In other words, δ is a function which maps state-action pairs to one-hot encoded vectors. Compute $\nabla_{\mathbf{w}} Q_{\mathbf{w}}(s, a)$ and write the update rule for $\mathbf{w}$. Show that equations (1) and (2) are equal when this linear approximation is used.

Solution: $\nabla_{\mathbf{w}} Q_{\mathbf{w}}(s, a) = \nabla_{\mathbf{w}} \mathbf{w}^\top \delta(s, a) = \delta(s, a)$ where $\delta(s, a)$ is the one hot vector with the 1 located in the location at (s, a). Plugging this gradient into the update rule above for the linear approximation setting, we get that:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} Q_{\mathbf{w}}(s', a') - Q_{\mathbf{w}}(s, a) \right) \delta(s, a)$$

Note that in the general linear approximation setting, each update step alters all of the learnable parameters. When we use the above one hot encoded representation of the state action value function, we are reducing our gradient update step to only be updating the learnable parameters which correspond to the pair (s, a). When we go back and look at the update rule for state action value pairs for the tabular case, we see that this update rule is the same because it considers updating the state action value pair one at a time:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} Q(s', a') - Q(s,a) \right)$$

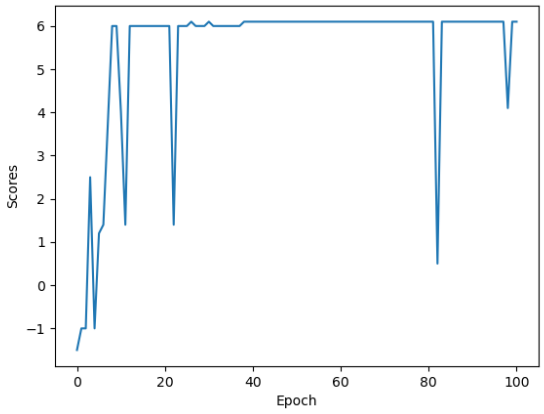
(b) (**coding**, 15 pts) We will now implement linear approximation in PyTorch. This question will set up the pipeline for the remainder of the assignment. You'll need to implement the following functions in `q4_linear_torch.py` (please read through `q4_linear_torch.py`):

- `initialize_models`
- `get_q_values`
- `update_target`
- `calc_loss`
- `add_optimizer`

Test your code by running `python q4_linear_torch.py` **locally on CPU**. This will run linear approximation with PyTorch on the test environment from Problem 3. Running this implementation should only take a minute.

(c) (**written**, 3 pts) Do you reach the optimal achievable reward on the test environment? Attach the plot `scores.png` from the directory `results/q4_linear` to your writeup.

Solution: After running the linear Q network on the toy environment from Question 1, we got an average reward of $6.10 + / - 0.00$.



5 Implementing DeepMind's DQN (13 pts)

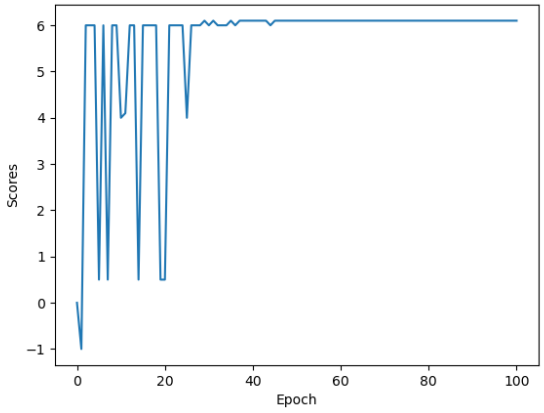
(a) (**coding** 10pts) Since we want this assignment to run on a laptop's CPU, we will implement a smaller version of the deep Q-network described in [1] by implementing `initialize_models` and `get_q_values` in `q5_nature_torch.py`. The rest of the code inherits from what you wrote for linear approximation. Test your implementation **locally on CPU** on the test environment by running `python q5_nature_torch.py`. Running this implementation should only take a minute or two.

Use the following architecture:

- One convolution layer with 16 output channels, a kernel size of 3, stride 1, and no padding.
- A ReLU activation.
- A dense layer with 128 hidden units.

- Another ReLU activation.
- The final output layer.

(b) (**written** 3 pts) Attach the plot of scores, `scores.png`, from the directory `results/q5_nature` to your writeup. Compare this model with linear approximation. How do the final performances compare? How about the training time?
Solution: The DQN also got the perfect score of 6.1 average reward. It was more noisy at the start of its training but maintained perfect performance after 20 epochs. The training time for the DQN was longer than the linear model.



6 DQN on MinAtar (21 pts)

(a) (**coding and written**, 5 pts). Now we're ready to train on the MinAtar `Breakout-v0` environment. First, launch linear approximation on Breakout with `python q6_train_atari_linear.py` on your CPU. This will train the model for 1,000,000 steps and should take approximately 20 minutes for a single run. Briefly qualitatively describe how your agent's performance changes over the course of training. Attach the plot `scores.png` from the directory `results/q6_train_atari_linear` to your writeup. **Note:** In the starter code provided, the models are trained 3 times and the final saved `scores.png` is the average of the three runs. Do you think that training for a larger number of steps would likely yield further improvements in performance? Explain your answer.
Solution: Training for a larger number of steps would not improve the performance of the model further because the linear model's representation is to limited. In other words, the model cannot represent the true value of taking an action because the class of functions which it can learn is too small.

(b) (**coding and written**, 10 pts). In this question, we'll train the agent with our custom architecture on the MinAtar `Breakout-v0` environment. Run `python q6_train_atari_nature.py` on your CPU. This will train the model for 1,000,000 steps. You are responsible for training it to completion, which should take roughly **1.5 hours** on a *CPU* for a single run. Attach the plot `scores.png` from the directory `results/q6_train_atari_nature` to your writeup. **Note:** In the starter code provided, the models are trained 3 times and the final saved `scores.png` is the average of the three runs. You should get a max score of around 16-18.

(c) (**written**, 3 pts) In a few sentences, compare the performance of the custom CNN architecture with the linear Q value approximator. How can you explain the gap in performance?
Solution: The gap in performance between the linear model and the CNN is due to the larger number of

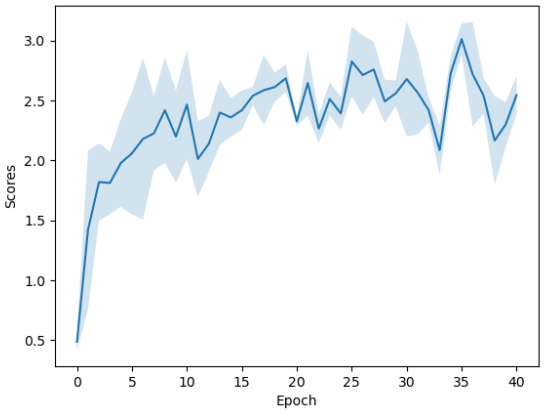


Figure 3: Linear Atari

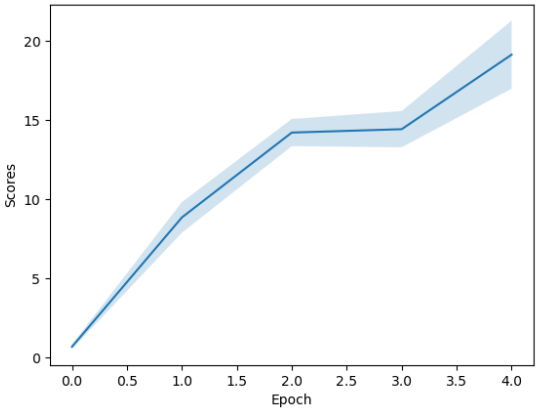


Figure 4: DQN Atari

learnable parameters and stronger inductive bias of the CNN. The increase in learnable parameters helps to expand the class of functions which our model can learn, enabling it to find more optimal functions to represent during training. Additionally, the Atari game uses pixels to represent the state space and CNNs are designed to capture image features well.

(d) (**written**, 3 pts) Will the performance of DQN over time always improve monotonically? Why or why not?
Solution: One reason why the performance wouldn't increase monotonically is if our DQN model still didn't have the capacity to represent the optimal action for each state. If this was the case, we would see the performance of the model hit an upper bound just like it did in the linear case. Additionally, since the rewards for the game of Atari are upper bounded by the number of blocks on the screen.

7 Distributions induced by a policy (13 pts)

Suppose we have a single MDP and two policies for that MDP, π and π'. Naturally, we are often interested in the performance of policies obtained in the MDP, quantified by V^{π} and $V^{\pi'}$, respectively. If the reward function and transition dynamics of the underlying MDP are known to us, we can use standard methods for

policy evaluation. There are many scenarios, however, where the underlying MDP model is not known and we must try to infer something about the performance of policy π' solely based on data obtained through executing policy π within the environment. In this problem, we will explore a classic result for quantifying the gap in performance between two policies that only requires access to data sampled from one of the policies.

Consider an infinite-horizon MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma \rangle$ and stochastic policies of the form $\pi : \mathcal{S} \to \Delta(\mathcal{A})$[1]. Specifically, $\pi(a|s)$ refers to the probability of taking action a in state s, and $\sum_a \pi(a|s) = 1$, $\forall s$. For simplicity, we'll assume that this decision process has a single, fixed starting state $s_0 \in \mathcal{S}$.

(a) (**written**, 3 pts) Consider a fixed stochastic policy and imagine running several rollouts of this policy within the environment. Naturally, depending on the stochasticity of the MDP $\mathcal{M}$ and the policy itself, some trajectories are more likely than others. Write down an expression for $\rho^\pi(\tau)$, the probability of sampling a trajectory $\tau = (s_0, a_0, s_1, a_1, \ldots)$ from running π in $\mathcal{M}$. To put this distribution in context, recall that $V^\pi(s_0) = \mathbb{E}_{\tau \sim \rho^\pi}\left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t) \,|\, s_0\right]$.

Solution:

$$\rho^\pi(\tau) = \prod_{i=0}^{\infty} \pi(a_i|s_i) \times P(s_{i+1}|a_i, s_i)$$

□

(b) (**written**, 1 pt) What is $p^\pi(s_t = s)$, where $p^\pi(s_t = s)$ denotes the probability of being in state s at time step t while following policy π? (Provide an equation)

Solution: One way to express the probability of being in state s at time step t is by marginalizing across all possible trajectories which end up in state s at time step t. By leveraging the formula from the previous question, we see that:

$$p^\pi(s_t = s) = \sum_{\tau} p(\tau | s_t(\tau) = s)$$

where $p(\tau|s_t(\tau) = s)$ is considering all trajectories where the tth step is at state s. □

(c) (**written**, 5 pts) Just as ρ^π captures the distribution over trajectories induced by π, we can also examine the distribution over states induced by π. In particular, define the *discounted, stationary state distribution* of a policy π as

$$d^\pi(s) = (1 - \gamma) \sum_{t=0}^{\infty} \gamma^t p^\pi(s_t = s),$$

where $p^\pi(s_t = s)$ denotes the probability of being in state s at timestep t while following policy π; your answer to the previous part should help you reason about how you might compute this value.

The value function of a policy π can be expressed using this distribution $d^\pi(s, a) = d^\pi(s)\pi(a \mid s)$ over states and actions, which will shortly be quite useful.

Consider an arbitrary function $f : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Prove the following identity:

$$\mathbb{E}_{\tau \sim \rho^\pi}\left[\sum_{t=0}^{\infty} \gamma^t f(s_t, a_t)\right] = \frac{1}{(1-\gamma)} \mathbb{E}_{s \sim d^\pi}\left[\mathbb{E}_{a \sim \pi(s)}\left[f(s, a)\right]\right].$$

Hint: You may find it helpful to first consider how things work out for $f(s, a) = 1, \forall (s, a) \in \mathcal{S} \times \mathcal{A}$.

Solution: Before writing out the formal proof for this problem, lets first consider some intuition about what we are reasoning over. The left hand side of this equation is taking an expectation of the discounted reward over the distribution of all trajectories under our policy, where a single trajectory is an infinite

[1] For a finite set $\mathcal{X}$, $\Delta(\mathcal{X})$ refers to the set of categorical distributions with support on $\mathcal{X}$ or, equivalently, the $\Delta^{|\mathcal{X}|-1}$ probability simplex.

sequence of states and actions. The right hand side is taking a double expectation over the the long-run stationary state distribution and then the action distribution for each state. Note that when we follow the hints guide and plug in a constant reward function, we get that these two expectations indeed both compute all possible trajectories in the MDP under a given policy.
For showing the above identity, consider all of the trajectories of the MDP under a given policy in an infinite matrix where each row of the matrix corresponds to a single trajectory. The left hand side of the expression we are trying to prove computes the discounted rewards be taking an expectation over the rows of this matrix. I wish to re-express this computation by summing over the columns of this matrix, where each column represents being at a state at a given time step. We can then use the above identity which relates the stationary state distribution to the probability of being in a state at a given time step.

$$\mathbb{E}_{\tau\sim\rho^\pi}\left[\sum_{t=0}^{\infty}\gamma^t f(s_t,a_t)\right] = \sum_{t=0}^{\infty}\gamma^t\sum_{s,a}p^\pi(s_t=s)\pi(a|s)f(s,a)$$

$$= \sum_{s,a}\sum_{t=0}^{\infty}\gamma^t p^\pi(s_t=s)\pi(a|s)f(s,a)$$

$$= \frac{1}{(1-\gamma)}\sum_{s,a}d^\pi(s)\pi(a|s)f(s,a)$$

$$= \frac{1}{(1-\gamma)}\sum_{s}d^\pi(s)\mathbb{E}_{a\sim\pi(s)}\left[f(s,a)\right]$$

$$= \frac{1}{(1-\gamma)}\mathbb{E}_{s\sim d^\pi}\left[\mathbb{E}_{a\sim\pi(s)}\left[f(s,a)\right]\right]$$

□

(d) (**written**, 5 pts) For any policy π, we define the following function

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

$A^\pi(s,a)$ is known as the advantage function and shows up in a lot of policy gradient based RL algorithms, which we shall see later in the class. Intuitively, it is the additional benefit one gets from first following action a and then following π, instead of always following π. Prove that the following statement holds for all policies π, π':

$$V^\pi(s_0) - V^{\pi'}(s_0) = \frac{1}{(1-\gamma)}\mathbb{E}_{s\sim d^\pi}\left[\mathbb{E}_{a\sim\pi(s)}\left[A^{\pi'}(s,a)\right]\right].$$

Hint 1: Try adding and subtracting a term that will let you bring $A^{\pi'}(s,a)$ into the equation. What happens on adding and subtracting $\mathbb{E}_{\tau\sim\rho^\pi}\sum_{t=0}^{\infty}\gamma^{t+1}V^{\pi'}(s_{t+1})$ on the LHS?
Hint 2: Recall the tower property of expectation which says that $\mathbb{E}[X] = \mathbb{E}[\mathbb{E}[X \mid Y]]$.

Solution: We hope to invoke the previous theorem to prove this statement. First, let us consider expanding out the definition of of the value of state s_0 under policy π:

$$V^\pi(s_0) = \mathbb{E}_{\tau\sim\rho^\pi}\left[\sum_{t=0}^{\infty}\gamma^t\mathcal{R}(s_t,a_t)\right].$$

where we drop the conditional notation and its known that the first state in all trajectories is at s_0. Now, following the hint, lets add and subtract $X = \mathbb{E}_{\tau\sim\rho^\pi}\sum_{t=0}^{\infty}\gamma^{t+1}V^{\pi'}(s_{t+1})$ from the left hand side:

$$V^\pi(s_0) + X - X = \mathbb{E}_{\tau\sim\rho^\pi}\left[\sum_{t=0}^{\infty}\gamma^t\mathcal{R}(s_t,a_t) + \gamma^{t+1}V^{\pi'}(s_{t+1}) - \gamma^{t+1}V^{\pi'}(s_{t+1})\right].$$

Recall that the state-action value function is defined as the reward received from taking an action and then the expected value of the next state:

$$V^{\pi}(s_0) + X - X = \mathbb{E}_{\tau\sim\rho^{\pi}}\left[\sum_{t=0}^{\infty}\gamma^t(\mathcal{R}(s_t,a_t) + \gamma V^{\pi'}(s_{t+1}) - \gamma V^{\pi'}(s_{t+1}))\right].$$

$$V^{\pi}(s_0) + X - X = \mathbb{E}_{\tau\sim\rho^{\pi}}\left[\sum_{t=0}^{\infty}\gamma^t(Q^{\pi'}(s_t,a_t) - \gamma V^{\pi'}(s_{t+1}))\right].$$

Now lets bring in the value of the initial state s_0 with respect to the policy π':

$$V^{\pi'}(s_0) = \mathbb{E}_{\tau\sim\rho^{\pi'}}\left[\sum_{t=0}^{\infty}\gamma^t\mathcal{R}(s_t,a_t)\ |s_0\right].$$

By the tower property of expectation, we have that:

$$\mathbb{E}_{\tau\sim\rho^{\pi}}\left[\mathbb{E}_{\tau\sim\rho^{\pi'}}\left[\sum_{t=0}^{\infty}\gamma^t\mathcal{R}(s_t,a_t)\ |s_0\right]\right] = \mathbb{E}_{\tau\sim\rho^{\pi}}\left[V^{\pi'}(s_0)\right] = V^{\pi'}(s_0)$$

Thus, we can include the value of the initial state within the expectation over the trajectories of the policy π:

$$V^{\pi}(s_0) - V^{\pi'}(s_0) + X - X = \mathbb{E}_{\tau\sim\rho^{\pi}}\left[-V^{\pi'}(s_0) + \sum_{t=0}^{\infty}\gamma^t(Q^{\pi'}(s_t,a_t) - \gamma V^{\pi'}(s_{t+1}))\right].$$

Bringing the negative term into the summation fixes the off by one issue with the time step!

$$V^{\pi}(s_0) - V^{\pi'}(s_0) = \mathbb{E}_{\tau\sim\rho^{\pi}}\left[\sum_{t=0}^{\infty}\gamma^t(Q^{\pi'}(s_t,a_t) - V^{\pi'}(s_t))\right] = \mathbb{E}_{\tau\sim\rho^{\pi}}\left[\sum_{t=0}^{\infty}\gamma^t A^{\pi'}(s_t,a_t)\right].$$

Now, by the previous question's theorem, we have that:

$$V^{\pi}(s_0) - V^{\pi'}(s_0) = \frac{1}{(1-\gamma)}\mathbb{E}_{s\sim d^{\pi}}\left[\mathbb{E}_{a\sim\pi(s)}\left[A^{\pi'}(s,a)\right]\right]$$

□

8 Real world RL with neural networks (10 pts)

Given a stream of batches of n environment interactions (s_i, a_i, r_i, s_i'), we want to learn the optimal value function using a neural network. The underlying MDP has a finite-sized action space.

(a) (**written**, 4 pts) Your friend first suggests the following approach:

- Initialize parameters ϕ of a neural network V_ϕ
- For each batch of k tuples (s_i, a_i, r_i, s_i') (sampled at random), do stochastic gradient descent with the loss function $\sum_{i=0}^{k}|y_i - V_\phi(s_i)|^2$, where $y_i = \max_{a_i}[r_i + \gamma V_\phi(s_i')]$, where the $\max_{a_i}$ is taken over all the tuples in the batch of the form $(s_i, a_i, *, *)$.

What is the problem with this approach? *Hint: Think about the type of data we have.*

Solution: One problem with this solution is that because the data is sampled at random from the batch we are making an iid assumption which is not true in RL since the data we are learning from is sampled from our agent and is highly correlated with each other. This, in addition to the fact that we are using the same network parameters to estimate our target as we are to evaluate our state, which causes maximization bias, will lead to very unstable and slow converging training. □

(b) (**written**, 3 pts) Your friend now suggests the following approach:

- Initialize parameters ϕ of a neural network for the state-action value function $Q_\phi(s, a)$
- For each batch of k tuples (s_i, a_i, r_i, s'_i) (sampled at random), do stochastic gradient descent with the loss function $\sum_{i=0}^{k} |y_i - Q_\phi(s_i, a_i)|^2$, where $y_i = [r_i + \gamma V(s'_i)]$

Now as we just have the network $Q_\phi(s, a)$, how would you determine $V(s)$ from the above training procedure?

Solution: The above parameterization is for the optimal state-action value function. That is, for each action in a given state, it approximates the value of taking that actions in the given state and then following the optimal policy there after. The optimal value function represents the value of following the optimal policy from that state. The optimal policy takes the best action in a givens state, and thus we can construct the optimal value function by taking the max of Q with respect to actions for a fixed state. □

(c) (**written**, 3 pts) Is the method in part (b) for learning the Q network guaranteed to give us an approximation of the optimal state-action value function?

Solution: In general, whenever doing function approximation with deep networks there is not any guarantees we can prove about optimally of the output. □

References

[1] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[2] Volodymyr Mnih et al. "Playing Atari With Deep Reinforcement Learning". In: *NIPS Deep Learning Workshop*. 2013.

[3] Kenny Young and Tian Tian. "MinAtar: An Atari-Inspired Testbed for Thorough and Reproducible Reinforcement Learning Experiments". In: *arXiv preprint arXiv:1903.03176* (2019).