CS 234 Winter 2022: Assignment #2

Due date:
January 28, 2022 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 DeepMind's network on `Pong` takes roughly **12 hours on GPU**, so **please start early!** (Only a completed run will recieve full credit) We will give you access to an Azure GPU cluster. You'll find the setup instructions on the course assignment page.

Introduction

In this assignment, we will first consider a few theoretical questions about Q-learning, and then we will implement deep Q-learning, following DeepMind's paper ([4] and [5]) that learns to play Atari games from raw pixels. 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 the `Pong-v0` environment from OpenAI gym, but the code can easily be applied to any other environment.

In Pong, one player scores if the ball passes by the other player. An episode is over when one of the players reaches 21 points. Thus, the total return of an episode is between -21 (lost every point) and $+21$ (won every point). Our agent plays against a decent hard-coded AI player. Average human performance is -3 (reported in [5]). In this assignment, you will train an AI agent with super-human performance, reaching at least $+10$ (hopefully more!).

1 Distributions induced by a policy (13 pts)

In this problem, we'll work with an infinite-horizon MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, \gamma \rangle$ and consider stochastic policies of the form $\pi : \mathcal{S} \to \Delta(\mathcal{A})$[1]. Additionally, we'll assume that $\mathcal{M}$ has a single, fixed starting state $s_0 \in \mathcal{S}$ for simplicity.

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

We have, just stacking up conditional probabilities and applying the Markov property to simplify the RHS of the conditionals:

$$
\begin{aligned}
p(\tau) &= p(s_0, a_0, s_1, a_1, \ldots,) \\
&= p(s_0)p(a_0|s_0)p(s_1|a_0, s_0)p(a_1|s_1, a_0, s_0)p(s_2|a_1, s_1, a_0, s_0)p(a_2|s_2, a_1, s_1, a_0, s_0)p(s_3|a_2, s_2, a_1, s_1, a_0, s_0)\ldots \\
&= p(s_0)p(a_0|s_0)p(s_1|a_0, s_0)p(a_1|s_1)p(s_2|a_1, s_1)p(a_2|s_2)p(s_3|a_2, s_2)\ldots \\
&= p(s_0)p(a_0|s_0)\prod_{i=1}^{\infty} p(s_i|s_{i-1}, a_{i=1})p(a_i|s_i) \\
&= 1 \cdot p(a_0|s_0)\prod_{i=1}^{\infty} p(s_i|s_{i-1}, a_{i=1})p(a_i|s_i).
\end{aligned}
$$

Note the infinite horizon here is given by the infinite horizon provided in the V^π reminder at the top of the problem.

(b) (**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

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

where $p(s_t = s)$ 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. 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 $f(s, a) = 1, \forall (s, a) \in \mathcal{S} \times \mathcal{A}$.
Hint: What is $p(s_t = s)$?

(LHS)
First, let's manipulate the LHS. We have (using space-saving notation $\sum_{s_\ell, a_\ell} p(s_\ell, a_\ell) := \sum_{s_\ell \in S}\sum_{a_\ell \in A} p(s_\ell =$

[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.

$s, a_\ell = a$), i.e. summing over the possibilities for the ℓth state and action)

$$\begin{aligned}
E_{\tau\sim\rho^\pi}\left[\sum_{t=0}^{\infty}\gamma^T f(s_t,a_t)\right] &= E_{\tau\sim\rho^\pi}(f(s_0,a_0)+\gamma f(s_1,a_1)+\gamma^2 f(s_2,a_2)+\gamma^3 f(s_3,a_3)+\dots) \\
&= \sum_\tau \rho^\pi(\tau)(f(s_0,a_0)+\gamma f(s_1,a_1)+\gamma^2 f(s_2,a_2)+\gamma^3 f(s_3,a_3)+\dots) \\
&= \sum_{s_0,a_0}\sum_{s_1,a_1}\sum_{s_2,a_2}\dots\sum p(s_0,a_0,s_1,a_1,\dots)(f(s_0,a_0)+\gamma f(s_1,a_1)+\gamma^2 f(s_2,a_2)+\dots).
\end{aligned}$$

Let's now look term-by-term in this expansion. Specifically, using the definition of marginal probability, we have:

- For the 0-index term:

$$\begin{aligned}
\sum_{s_0,a_0}\sum_{s_1,a_1}\sum_{s_2,a_2}\dots\sum p(s_0,a_0,s_1,a_1,\dots)f(s_0,a_0) &= \sum_{s_0,a_0} f(s_0,a_0)\sum_{s_1,a_1}\sum_{s_2,a_2}\dots\sum p(s_0,a_0,s_1,a_1,\dots) \\
&= \sum_{s_0,a_0} f(s_0,a_0)p(s_0,a_0) \\
&= \sum_{s\in S}\sum_{a\in A} f(s,a)p(s_0=s,a_0=a) \\
&= \sum_{s\in S}\sum_{a\in A} f(s,a)p(s_0=s)p(a_0=a|s_0=s)
\end{aligned}$$

- For the 1-index term:

$$\begin{aligned}
\sum_{s_0,a_0}\sum_{s_1,a_1}\sum_{s_2,a_2}\dots\sum p(s_0,a_0,s_1,a_1,\dots)\gamma f(s_1,a_1) &= \sum_{s_0,a_0}\sum_{s_1,a_1}\gamma f(s_1,a_1)\sum_{s_2,a_2}\dots\sum p(s_0,a_0,s_1,a_1,\dots) \\
&= \sum_{s_0,a_0}\sum_{a_1,a_1}\gamma f(s_1,a_1)p(s_0,a_0,a_1,a_1) \\
&= \sum_{s_1,a_1}\gamma f(s_1,a_1)\sum_{a_0,a_0} p(s_0,a_0,a_1,a_1) \\
&= \sum_{s_1,a_1}\gamma f(s_1,a_1)p(s_1,a_1) \\
&= \sum_{s\in S}\sum_{a\in A}\gamma f(s,a)p(s_1=s,a_1=a) \\
&= \sum_{s\in S}\sum_{a\in A}\gamma f(s,a)p(s_1=s)p(a_1=a|s_1=s)
\end{aligned}$$

- For the 2-index term:

$$\begin{aligned}
\sum_{s_0,a_0}\sum_{s_1,a_1}\sum_{s_2,a_2}\dots\sum p(s_0,a_0,s_1,a_1,\dots)\gamma^2 f(s_2,a_2) &= \sum_{s_0,a_0}\sum_{s_1,a_1}\sum_{s_2,a_2}\gamma^2 f(s_2,a_2)p(s_0,a_0,s_1,a_1,s_2,a_2) \\
&= \sum_{s_2,a_2}\sum_{s_0,a_0}\sum_{s_1,a_1}\gamma^2 f(s_2,a_2)p(s_0,a_0,s_1,a_1,s_2,a_2) \\
&= \sum_{s_2,a_2}\gamma^2 f(s_2,a_2)\sum_{s_0,a_0}\sum_{s_1,a_1} p(s_0,a_0,s_1,a_1,s_2,a_2) \\
&= \sum_{s_2,a_2}\gamma^2 f(s_2,a_2)p(s_2,a_2) \\
&= \sum_{s\in S}\sum_{a\in A}\gamma^2 f(s,a)p(s_2=s,a_2=a) \\
&= \sum_{s\in S}\sum_{a\in A}\gamma^2 f(s,a)p(s_2=s)p(a_2=a|s_2=s)
\end{aligned}$$

- And more generally, for the j index term (algebra remains the same: margin out the infinite indices, rearrange, and margin out once more):

$$\sum_{s_0,a_0}\sum_{s_1,a_1}\sum_{s_2,a_2}\dots\sum p(s_0,a_0,s_1,a_1,\dots)\gamma^2 f(s_2,a_2) = \sum_{s_j,a_j}\gamma^j f(s_j,a_j)p(s_j,a_j)$$
$$= \sum_{s\in S}\sum_{a\in A}\gamma^j f(s,a)p(s_j=s)p(a_j=a|s_j=s)$$

Thus, when we sum over all of the above terms (as is necessary for the original expansion), we have

$$E_{\tau\sim\rho^\pi}\left[\sum_{t=0}^{\infty}\gamma^T f(s_t,a_t)\right] = \sum_{s_0,a_0}\sum_{s_1,a_1}\sum_{s_2,a_2}\dots\sum p(s_0,a_0,s_1,a_1,\dots)(f(s_0,a_0)+\gamma f(s_1,a_1)+\gamma^2 f(s_2,a_2)+\dots)$$
$$= \sum_{t=0}^{\infty}\sum_{s\in S}\sum_{a\in A}\gamma^t p(s_t=s)p(a_j=a|s_t=s)f(s,a).$$

We will revisit this result shortly.

(RHS)
Then, on the RHS, we have through substitution and rearrangement over sums

$$\frac{1}{1-\gamma}E_{s\sim d^\pi}E_{a\sim\pi(s)}[f(s,a)] = \frac{1}{1-\gamma}E_{s\sim d^\pi}\sum_{a\in A}p(a_t=a|s_t=s)f(s,a)$$
$$= \frac{1}{1-\gamma}\sum_{s\in S}d^\pi(s)\sum_{a\in A}p(a_t=a|s_t=s)f(s,a)$$
$$= \frac{1}{1-\gamma}\sum_{s\in S}(1-\gamma)\sum_{t=0}^{\infty}\gamma^t p(s_t=s)\sum_{a\in A}pi(a_t=a|s_t=s)$$
$$= \sum_{t=0}^{\infty}\sum_{s\in S}\sum_{a\in A}\gamma^t p(s_t=s)p(a_t=a|s_t=s)f(s,a).$$

This shows LHS and RHS are equal, and the proof is complete.

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

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

Prove 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: Try adding and subtracting a term that will let you bring $A^\pi(s,a)$ into the equation

Begin on the LHS:

$$V_\pi(s_0) - V_{\pi'}(s_0) = E_{\tau\sim\rho^\pi}\left[\sum_{t=0}^{\infty}\gamma^t R(s_t,a_t)\right] - V_{\pi'}(s_0) \tag{1}$$

$$= E_{\tau\sim\rho^\pi}\left[\sum_{t=0}^{\infty}\gamma^t R(s_t,a_t) - V_{\pi'}(s_0)\right] \tag{2}$$

$$= E_{\tau\sim\rho^\pi}\left[\sum_{t=0}^{\infty}\gamma^t R(s_t,a_t) - V_{\pi'}(s_0) + \sum_{t=0}^{\infty}\gamma^{t+1}V_{\pi'}(s_{t+1}) - \sum_{t=0}^{\infty}\gamma^{t+1}V_{\pi'}(s_{t+1})\right] \tag{3}$$

$$= E_{\tau\sim\rho^\pi}\left[\sum_{t=0}^{\infty}\gamma^t R(s_t,a_t) + \sum_{t=0}^{\infty}\gamma^{t+1}V_{\pi'}(s_{t+1}) - \sum_{t=0}^{\infty}\gamma^{t+1}V_{\pi'}(s_{t+1}) - V_{\pi'}(s_0)\right] \tag{4}$$

$$= E_{\tau\sim\rho^\pi}\left[\sum_{t=0}^{\infty}\gamma^t\Big(R(s_t,a_t) + \gamma V_{\pi'}(s_{t+1})\Big) - \sum_{t=0}^{\infty}\gamma^{t+1}V_{\pi'}(s_{t+1}) - \gamma^0 V_{\pi'}(s_0)\right] \tag{5}$$

$$= E_{\tau\sim\rho^\pi}\left[\sum_{t=0}^{\infty}\gamma^t Q_{\pi'}(s_t,a_t) - \sum_{t=0}^{\infty}\gamma^t V_{\pi'}(s_t)\right] \tag{6}$$

$$= E_{\tau\sim\rho^\pi}\left[\sum_{t=0}^{\infty}\gamma^t (Q_{\pi'}(s_t,a_t) - V_{\pi'}(s_t))\right] \tag{7}$$

$$= \frac{1}{(1-\gamma)}\mathbb{E}_{s\sim d^\pi}\left[\mathbb{E}_{a\sim\pi(s)}\left[A_{\pi'}(s_t,a_t)\right]\right], \tag{8}$$

as desired. Steps are justified by: (1) definition of $V_\pi(s_0)$; (2) linearity of expectation; (3) addition and subtraction of same term; (4) rearrangement; (5) grouping of first two terms/extraction of γ^t coefficient; (6) the lemma set forth below; (7) linearity of expectation; (8) problem b; (9) substitution.

Lemma

The tower property gives, treating the $s_t + a_t \to s_{t+1}$ transition as a RV where the destination s_{t+1} is random from starting state s_t and action a_t; that is, it is the RV describing the transition $s_t + a_t \to s_{t+1}$. We'll use $(s_{t+1}|s_t,a_t)$ as shorthand to denote this RV:

$$E_{\tau\sim\rho^\pi}\left[\sum_{t=0}^{\infty}\gamma^t\Big(R(s_t,a_t) + \gamma V_{\pi'}(s_{t+1})\Big)\right] = E_{\tau\sim\rho^\pi}\left[E_{(s_{t+1}|s_t,a_t)}\left[\sum_{t=0}^{\infty}\gamma^t\Big(R(s_t,a_t) + \gamma V_{\pi'}(s_{t+1})\Big)\Big|(s_{t+1}|s_t,a_t)\right]\right]$$

$$= E_{\tau\sim\rho^\pi}\left[\sum_{t=0}^{\infty}\gamma^t\Big(E_{(s_{t+1}|s_t,a_t)}[R(s_t,a_t)|(s_{t+1}|s_t,a_t)] + \gamma E_{(s_{t+1}|s_t,a_t)}[V_{\pi'}(s_{t+1})|(s_{t+1}|s_t,a_t)]\Big)\right]$$

$$= E_{\tau\sim\rho^\pi}\left[\sum_{t=0}^{\infty}\gamma^t\Big(R(s_t,a_t) + \gamma E_{(s_{t+1}|s_t,a_t)}[V_{\pi'}(s_{t+1})|(s_{t+1}|s_t,a_t)]\Big)\right]$$

$$= E_{\tau\sim\rho^\pi}\left[\sum_{t=0}^{\infty}\gamma^t\Big(R(s_t,a_t) + \gamma\sum_{s_{t+1}}p(s_{t+1}|s_t,a_t)V_{\pi'}(s_{t+1})\Big)\right]$$

$$= E_{\tau\sim\rho^\pi}\left[\sum_{t=0}^{\infty}\gamma^t Q(s_t,a_t)\right].$$

2 SARSA vs. Q-Learning (5 pts)

In the grid world below (1), the agent can move left, right, up or down. Moving onto a green square yields reward -1, while moving onto a red square yields reward -1000 and terminates the episode. The episode also terminates if the agent reaches state 35, the goal state. The agent starts at state 29.

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	32	33	34	35

Figure 1: Grid World.

Suppose we train two algorithms, Q-learning and SARSA, with **fixed** $\epsilon = 0.1$ and $\gamma = 1$, and learn two policies: Policy A and Policy B.
Policy A takes the path

$$29 - 22 - 15 - 8 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 14 - 21 - 28 - 35$$

and Policy B takes path

$$29 - 22 - 15 - 16 - 17 - 18 - 19 - 20 - 21 - 28 - 35$$

Which policy is learned by Q-learning, and which policy is learned by SARSA? Explain how you arrived at your answer in just a couple of sentences.

Policy B: Q
Policy A: SARSA

Here, there are two paths: a long path, as taken by Policy A, vs. the short path, as taken by Policy B. Critically, note that the B path is the optimal path under this reward configuration, achieving a total reward of -10, which is greater than the total -14 reward achieved by the A path.

The giveaway here is the **fixed** ϵ. As set forth in Lecture 4, SARSA must be ϵ-greedy in the limit (GLIE) for the Q-values to converge to the optimal Q^* (4.45), whereas there is no such requirement for $Q-learning$ (4.54). However here, ϵ is not annealed to satisfy GLIE, which means that the Q-learner will still converge to Q^*, but the SARSA learner is unlikely to do so, and may converge to some $\tilde{Q} \neq Q^*$ (by virtue of 4.54 and 4.45, respectively). Thus, when we run both algorithms for a sufficient number of iterations (call the estimates $\hat{Q}_q$ for Q-learning and $\hat{Q}_s$ for SARSA), and then follow a greedy path according to those found $\hat{Q}_q, \hat{Q}_s$, the path from $\hat{Q}_q$ will be optimal (as $\hat{Q}_q$ is the true state-action value), whereas the path from $\hat{Q}_s$ will/may not be optimal. And this is the giveaway – since we know *ex ante* that Policy B is optimal, it must be the path corresponding to a greedy following of $\hat{Q}_q$. So Policy B corresponds to the Q-learner; it then follows that Policy A corresponds to SARSA (the suboptimal policy).

Intuition: Due to the on/off policy difference between SARSA and Q-learning, the intuition behind this difference is relatively straightforward. Specifically, think of what happens in non-GLIE SARSA when walking

along the "cliff" of the grid, which happens to be the optimal path (16-20). Due to the un-annealed/sustained randomness of ϵ, the exploration will at some point fall off the cliff, and move down into the red (23-37). Doing so will incur a massive negative penalty, which will eventually be passed back to an otherwise good state-action pair, thus "torpedoing" the good state-action pair on the cliff. This will encourage the learner to play it safe and go as far away from the cliff, which is why SARSA takes the long way around in the demo above.

As a quick example of this principle, consider the following SARSA situation. Suppose you are at state 17; randomness might drop you to the 24. You would incur a reward of -1000, causing $Q(s_t = 17, a_t = \downarrow)$ to take on a large negative value during the update step. Then, later on in the exploration, you might be at 16, and randomly (because it is not GLIE) sample $(a_t = \rightarrow, s_{t+1} = 17, a_{t+1} = \downarrow, s_{t+2} = 24)$ as part of your sampled tuple. Now, when you go to update $Q(s_t = 16, a_t = \rightarrow)$ – what is ostensibly a high-value state-action – the sunken $Q(s_{t+1} = 17, a_{t+1} = \downarrow)$ will unfairly sink your $Q(16, \rightarrow)$ during the update. In the end, any greedy policy derived from the eventual $\tilde{Q}$ will be discouraged from selecting $\rightarrow$ from 16 during the arg max step; hence the policy will go the long way around and play it safe.

3 Test Environment (6 pts)

Before running our code on Pong, 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: $s_0 = 0, a_0 = 1, R_0 = -0.3, s_1 = 1, a_1 = 2, R_1 = 0.0, s_2 = 2, a_2 = 4, R_2 = 0.0, s_3 = 2, a_3 = 3, R_3 = 2.0, s_4 = 3, a_4 = 0, R_4 = 0.1, s_5 = 0$.

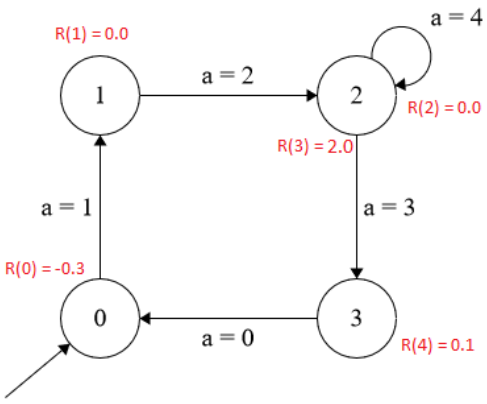


Figure 2: Example of a trajectory in the Test Environment

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.

First, observe the following key facts:

- From any state, we can always maneuver to any of the other states. In other words, our choice of "next state" is not structurally limited; we can always get there for some reward.
- Only two moves – namely, $\langle s_2, a_3, s_3 \rangle$ and $\langle s_2, a_1, s_1 \rangle$ – achieve rewards $> .1$ (2 and 3, respectively).
- The sequence $s_2 \to s_1 \to s_2$ achieves a reward of $3.0 + 0.0$; while the sequence from $s_2 \to s_3 \to s_2$ achieves $2.0 + 0.0$, meaning that the two "big moves" are not discounted by moving off of s_1 or s_3; in other words, moving from $s_2 \to s_1 \to s_2$ can guarantee you an average two-step reward of 1.5, while $s_2 \to s_3 \to s_2$ can guarantee you an average two-step reward of 1.0. This further tells us that whenever we are at s_2, we have that $s_2 \to s_1 \to s_2$ is preferable to $s_2 \to s_3 \to s_2$.
- Given that all other moves incur a reward of < 0.1 over one step, it also follows that the $s_2 \to s_1 \to s_2$ two-steps is preferable to any other two-step sequence.
- Hence, the objective here is to get to 2 as fast as possible, and then bounce between 2 and 1 as many times as possible to incur as many $+3$ rewards as possible. Since s_0 is necessarily the first state, we maximize by going to s_2 as quickly as possible, so that we can go to s_1, and then trying to repeat once more before the episode ends. That is, our first four steps will be

  $$s_0 \overset{+0}{\to} s_2 \overset{+3}{\to} s_1 \overset{+0}{\to} s_2 \overset{+3}{\to} s_1.$$

  At this point $t = 4$, however, we will not have time for another $s_2 \to s_1$ voyage, so for the final step we just want to maximize the $t = 5$ reward. This is achieved by moving from $s_1 \to s_0$ for a reward of .1, as all other moves off of s_1 are rewarded with values ≤ 0.. Hence, if we move

  $$s_0 \overset{+0}{\to} s_2 \overset{+3}{\to} s_1 \overset{+0}{\to} s_2 \overset{+3}{\to} s_1 \overset{+0.1}{\to} s_0,$$

  for a reward of 6.1, we will do the best we could have done.

4 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

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} Q(s', a') - Q(s, a) \right) \tag{9}$$

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 `q4_schedule.py`. Test your implementation by running `python q4_schedule.py`.

5 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

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} Q_{\mathbf{w}}(s', a') - Q_{\mathbf{w}}(s, a) \right) \nabla_{\mathbf{w}} Q_{\mathbf{w}}(s, a) \tag{10}$$

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 (9) and (10) are equal when this linear approximation is used.

By linearity, we know that

$$\nabla_w Q_w(s, a) = \nabla_w w^T \delta(s, a) = \delta(s, a).$$

In other words, it's just an indicator vector, with the 1 switched on in the position corresponding to s, a. Now, under the update rule above, we have

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} Q_{\mathbf{w}}(s', a') - Q_{\mathbf{w}}(s, a) \right) \nabla_{\mathbf{w}} Q_{\mathbf{w}}(s, a) \\ &\leftarrow \mathbf{w} + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} \mathbf{w}^T \delta(s', a') - \mathbf{w}^T \delta(s, a) \right) \delta(s, a). \end{aligned}$$

Now put this update in the context of Q-learning. Suppose we have our sampled tuple (s, a, r, s'). By definition, $\delta(s, a)$ will be switched on in the position corresponding to the s, a interaction, and zero elsewhere. Hence, for the position in $\mathbf{w}$ corresponding to the s, a interaction, the update will be (looking element-wise):

$$\begin{aligned} \mathbf{w}_{(s,a)} &\leftarrow \mathbf{w}_{(s,a)} + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} \mathbf{w}^T \delta(s', a') - \mathbf{w}^T \delta(s, a) \right) \delta(s, a)_{(s,a)} \\ &\leftarrow \mathbf{w}_{(s,a)} + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} \mathbf{w}^T \delta(s', a') - \mathbf{w}_{(s,a)} \right) \cdot 1. \end{aligned}$$

Since $\mathbf{w}^T \delta(s, a) \approx Q(s, a)$, this form above matches 9 identically, i.e. the update is

$$\underbrace{\mathbf{w}_{(s,a)}}_{\approx Q(s,a)} + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} \underbrace{\mathbf{w}^T \delta(s', a')}_{\approx Q(s', a')} - \underbrace{\mathbf{w}_{(s,a)}}_{\approx Q(s,a)} \right)$$

Meanwhile, for all other positions $s'' \neq s$ and/or $a'' \neq a$, the $\delta(s, a)$ term will be zero, giving:

$$\begin{aligned} \mathbf{w}_{(s'',a'')} &\leftarrow \mathbf{w}_{(s'',a'')} + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} \mathbf{w}^T \delta(s', a') - \mathbf{w}_{(s,a)} \right) \delta(s, a)_{(s'',a''} \\ &\leftarrow \mathbf{w}_{(s'',a'')} + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} \mathbf{w}^T \delta(s', a') - \mathbf{w}_{(s,a)} \right) \cdot 0 \\ &\leftarrow \mathbf{w}_{(s'',a'')}, \end{aligned}$$

i.e. things do not change outside if the position in $\mathbf{w}$ corresponding to (s, w). This further matches Q-learning, only the s, a entry for Q is updated within each loop, and everything else is unchanged.

In sum, the inner-product between $\mathbf{w}$ and δ uses indicator functions to effectively provide us a lookup table for the (s, a) on hand – a mirror image of Q-learning.

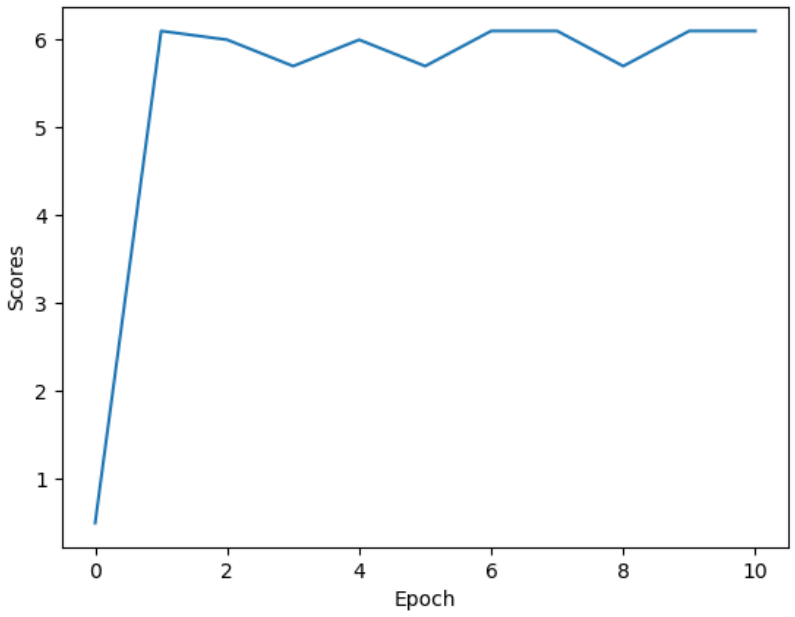
(b) (**coding**, 15 pts) We will now implement linear approximation in PyTorch. This question will set up the pipeline for the remainder of the assignment. You'll need to implement the following functions in `q5_linear_torch.py` (please read through `q5_linear_torch.py`):

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

Test your code by running `python q5_linear_torch.py` **locally on CPU**. This will run linear approximation with PyTorch on the test environment from Problem 0. Running this implementation should only take a minute. Completed in coding submission.

(c) (**written**, 3 pts) Do you reach the optimal achievable reward on the test environment? Attach the plot `scores.png` from the directory `results/q5_linear` to your writeup.

As evidenced below, the optimal achievable reward of 6.1 is achieved after 10 training epochs:



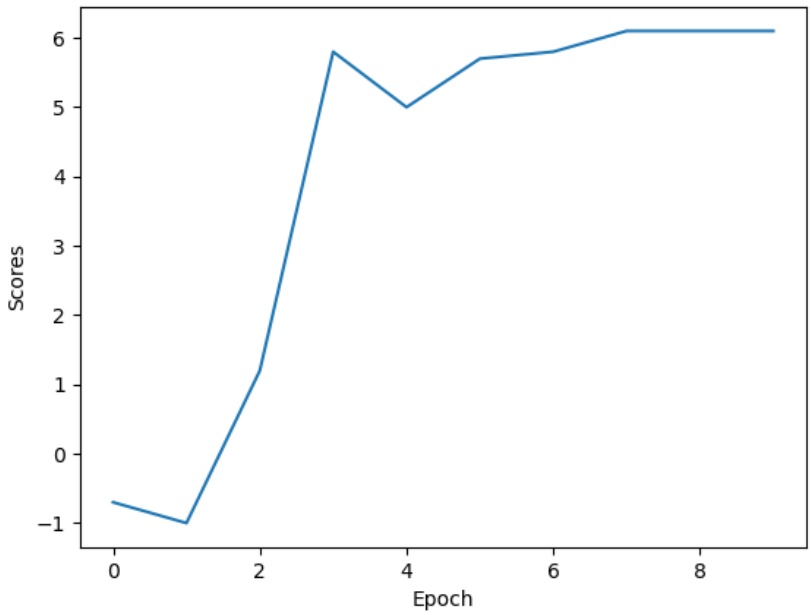
(a) Reward achieved by epoch

6 Implementing DeepMind's DQN (13 pts)

(a) (**coding** 10pts) Implement the deep Q-network as described in [4] by implementing `initialize_models` and `get_q_values` in `q6_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 q6_nature_torch.py`. Running this implementation should only take a minute or two.

(b) (**written** 3 pts) Attach the plot of scores, `scores.png`, from the directory `results/q6_nature` to your writeup. Compare this model with linear approximation. How do the final performances compare? How about the training time?

Here, we see performance of the DQN. It runs fairly quickly on this toy dataset, taking ~ 70 seconds to finish up on local CPU, which is admittedly longer than the near-instantaneous run for the linear. No shock there, this is a much more complex model with many more parameters to optimize over, so we should expect a slightly bloated runtime.



(a) Reward achieved by epoch

We see that ultimately, the DQN gets to a similar reward as linear (6.1). However (likely due to its architectural/parameter complexity), it takes a few more epochs to get there than does linear, which jumps to 6 almost immediately.

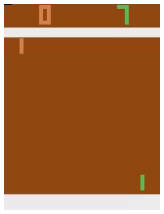
7 DQN on Atari (21 pts)

Reminder: Please remember to kill your VM instances when you are done using them!!

The Atari environment from OpenAI gym returns observations (or original frames) of size $(210 \times 160 \times 3)$, the last dimension corresponds to the RGB channels filled with values between 0 and 255 (`uint8`). Following DeepMind's paper [4], we will apply some preprocessing to the observations:

- Single frame encoding: To encode a single frame, we take the maximum value for each pixel color value over the frame being encoded and the previous frame. In other words, we return a pixel-wise max-pooling of the last 2 observations.
- Dimensionality reduction: Convert the encoded frame to grey scale, and rescale it to $(80 \times 80 \times 1)$. (See Figure 5)

The above preprocessing is applied to the 4 most recent observations and these encoded frames are stacked together to produce the input (of shape $(80 \times 80 \times 4)$) to the Q-function. Also, for each time we decide on an action, we perform that action for 4 time steps. This reduces the frequency of decisions without impacting the performance too much. You can refer to the *Methods Section* of [4] for more details.



(a) Original input $(210 \times 160 \times 3)$ with RGB colors

(b) After preprocessing in grey scale of shape $(80 \times 80 \times 1)$

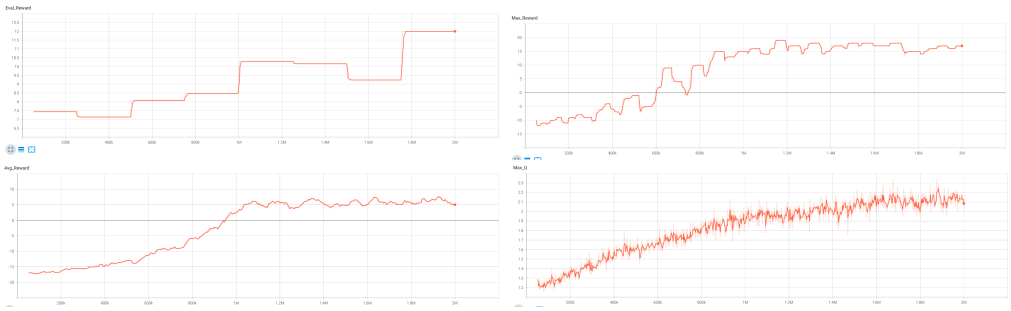
Figure 5: `Pong-v0` environment

(a) (**coding and written**, 5 pts). Now we're ready to train on the Atari `Pong-v0` environment. First, launch linear approximation on pong with `python q7_train_atari_linear.py` **on Azure's GPU**. This will train the model for 500,000 steps and should take approximately an hour. Briefly qualitatively describe how your agent's performance changes over the course of training. Do you think that training for a larger number of steps would likely yield further improvements in performance? Explain your answer. I ran the linear approximation on GPU – runtime took approximately 5,800 seconds, or 1.6 hours. And judging by the performance of the linear model over this span, it's hopeless: it consistently returned rewards around -20, with no sign of improvement over the course of training (see the plot in the next problem). This is no surprise though – the linear approximation simply can't handle the complexity of an Atari game, so it's probably unfair to expect anything other than poor performance.

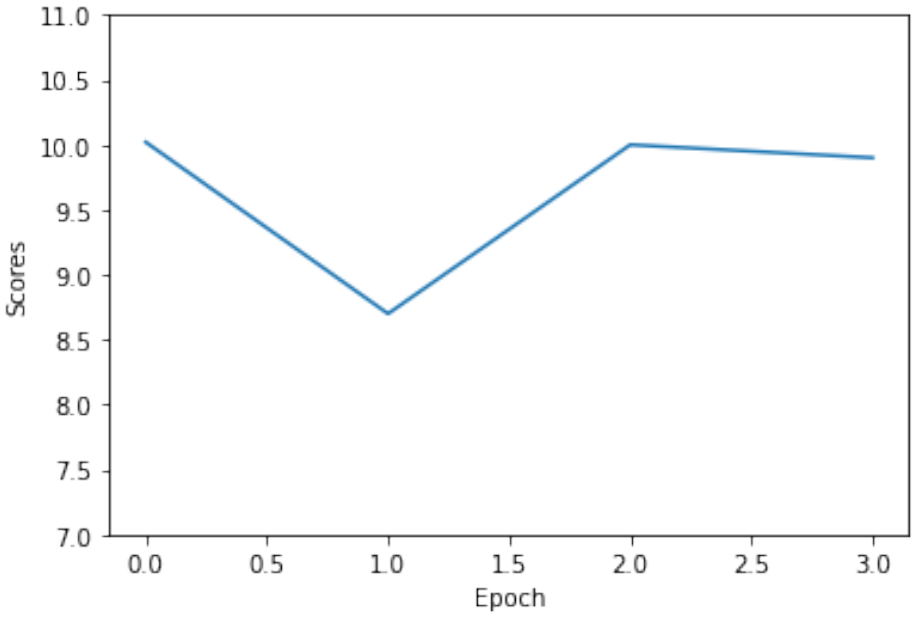
(b) (**coding and written**, 10 pts). In this question, we'll train the agent with DeepMind's architecture on the Atari `Pong-v0` environment. Run `python q7_train_atari_nature.py` **on Azure's GPU**. This will train the model for 4 million steps. To speed up training, we have trained the model for 2 million steps. You are responsible for training it to completion, which should take **12 hours**. Attach the plot `scores.png` from the directory `results/q7_train_atari_nature` to your writeup. You should get a score of around 9-11 after 4 million total time steps. As stated previously, the DeepMind paper claims average human performance is -3.

As the training time is roughly 12 hours, you may want to check after a few epochs that your network is making progress. The following are some training tips:

- If you terminate your terminal session, the training will stop. In order to avoid this, you should use `screen` to run your training in the background.
- The evaluation score printed on terminal should start at -21 and increase.
- The max of the q values should also be increasing
- The standard deviation of q shouldn't be too small. Otherwise it means that all states have similar q values
- You may want to use Tensorboard to track the history of the printed metrics. You can monitor your training with Tensorboard by typing the command `tensorboard --logdir=results` and then connecting to `ip-of-you-machine:6006`. Below are our Tensorboard graphs from one training session:



Scoring results are below – indeed, the performance gets into the desired 9-11 range, maxing out around 10. Training took $\sim$ 9720 seconds, or about 2.7 hours.



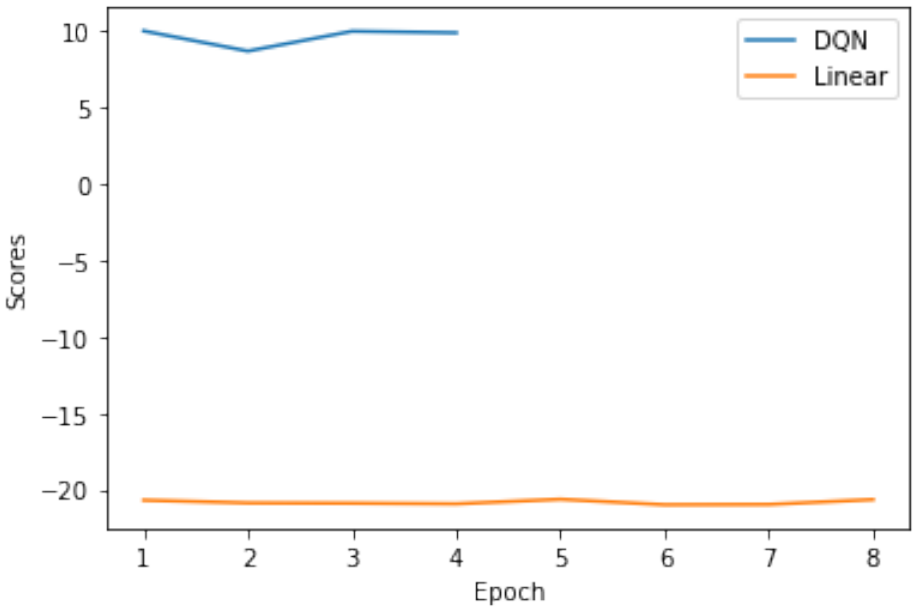
(a) Reward achieved by epoch

Note that for whatever reason, Azure would not allow me to copy the '.png' directly off of the virtual machine – instead, I had to save the scoring as a '.txt', and remake the plot locally.

(c) (**written**, 3 pts) In a few sentences, compare the performance of the DeepMind DQN architecture with the linear Q value approximator. How can you explain the gap in performance?

The delta in performance in stark: the linear approximation learns almost nothing, whereas the DQN learns to play better than the average human (whose score is -3), achieving roughly 10 point rewards. This gap in performance can likely be chalked up to the differing model complexities – the linear approximation is too simple to learn any of the nuances of Atari, whereas the CNN has sufficient parameters and structure to do so (in a computationally feasible manner).

Plotted below is the linear approximation vs the DQN – as we see, the DQN is worlds ahead. Perhaps more iterations could help DQN, though I'm skeptical – rewards seem to have plateaued after the 4 epochs, so I'd expect any gains to be marginal.



(a) Reward achieved by epoch

(d) (**written**, 3 pts) Will the performance of DQN over time always improve monotonically? Why or why not? Under DQN, we have no guarantees of convergence: because DQN is in the "deadly triad" of off-policy + bootstrapping + function approximation, wherein learning diverges entirely. While we can take steps to mitigate this risk (experience replay, double-DQN, dueling, etc.), it always exists, and thus monotonic improvement to convergence is never a sure thing in this setting. Hence, we cannot *guarantee* monotonic improvement.

More generally, because it is $\epsilon - greedy$ and off policy, a single "bad"/"unfavorable" randomized step may induce a sub-optimal update, potentially dragging down performance momentarily and breaking monotonicity.

8 The Environmental Impact of Deep RL

Researchers increasingly turn to deep learning models trained on GPUs to solve challenging tasks such as Atari. However, training deep learning models can have a significant impact on the environment [3][1]. According to a recent study [6], training BERT on GPU produces carbon emissions roughly equivalent to a trans-American flight. Closer to home, [1] considers a "Deep RL class of 235 students (such as Stanford's CS234). For a homework assignment, each student must run an algorithm 5 times on Pong. The class would

save 888 kWh of energy by using PPO versus DQN, while achieving similar performance. This is roughly the same amount needed to power a US home for one month" [1]. In the following questions, we will investigate the carbon footprint of our assignment and discuss researcher responsibility.

(a) (**written**, 4pts) Papers such as [3][1] strive for precise calculations of the carbon footprints of models, considering factors such as the type of GPUs used, the location of the data center, and length of model training. Use the tool on this webpage to calculate the likely carbon emissions resulting from your training of Pong on this assignment. What was your result for a single training attempt? For the whole assignment? Explain how you arrived at the final numbers. Compare your answers to the carbon footprint of the original DeepMind Atari paper [5] which takes about 50 hours to train a single game (without hyperparameter tuning). What is your estimate of DeepMind Atari's carbon footprint for a single game? **Clearly state the type of GPU used, time taken and the server location.** If you don't know the exact values for fields such as the type of GPU used or the server location, choose the settings given below:

- GPU: Titan RTX
- Server location: West US

Projected emission totals are as follows, using GPU Titan RTX, location East US, and Azure:

- Linear ran for 1.6 hours. By MLC02, this added .17 units (kg) of carbon emissions.
- DQN ran for 2.7 hours. By MLC02, this added .28 units (kg) of carbon emissions.
- Additionally, I estimate that over the course of getting acquainted with the server, figuring out how to copy files, initializing packages/environments, and making sure things generally worked, I probably spent an additional hour with the VM active. So by MLC02, that's another .1 units of carbon emissions.

Altogether, this put roughly .55 kg of carbon into the atmosphere. By contrast, the Deepmind Run (assuming 50 hrs, Titan RTX, West US, and Private infrastructure) figures to emit roughly 6.05 kg of carbon over a single game. Yikes!

(b) (**written**, 4pts) What are at least two specific ways to reduce the carbon footprint of training your DQN model (or any other deep RL model)? For example, [3] states that one way to reduce the number of models you train is to use random hyperparameter search as opposed to a grid search as the former is more efficient. The linked articles have general ideas for reducing carbon emissions of models. You are welcome to refer to them or other research, and please cite the sources of your ideas in doing so. One straightforward method – as posited by Lacoste *et al.* – is to pick a good provider. For example, Google leans on wind farms for energy, which is a comparatively sustainable solution. Likewise, Azure is certified to be carbon neutral. According to Lacoste's research, AWS is not quite there. So picking a good and comparatively sustainable service – most notably, Google Cloud or Azure – is a good place to start.

Second, once you have selected a decent provider, Lacoste *et al.* note that you should also be judicious about server location from that provider. For instance, if you know that a server location relies on fossil fuels and you are about to deploy a big training run, you might want to avoid that location. Instead, seek out a server location powered by hydroelectricity (or wind?) – according to Lacoste *et al.*, this can save you hundreds of carbon emissions.

So really – as the Lacoste paper notes – you can save a lot just by doing some basic research.

(c) (**written**, 5pts) Avram Hiller [2] has argued that since without extenuating circumstances it is "wrong to perform an act which has an expected amount of harm greater than another easily available alternative," we ought to take actions to avoid even minor climate emissions when there are easily available alternatives. Read over the mitigation strategies for individual machine learning researchers and for industry developers/framework maintainers referenced on page 26 of [1]. According to this principle, which of these proposed actions would RL researchers have a responsibility to take? List all the actions you think would be relevant and provide justifications for why two of the actions you listed ought to be taken according to this principle. Finally, do you agree with this principle? Why or why not?

Henderson et al. recommend the following best practices (italicized comments are my comments/answers)

- Running cloud jobs in low carbon regions only. *This action should be taken, and aligns with Hiller: it's easy enough to do some quick research, find a lower-carbon region, and click the option for that when initializing the cloud instance. So there is an easily available alternative here, and the Hiller doctrine should be followed.*
- Reporting metrics to make energy configurations more accessible. *As experienced firsthand in the exercise above, this is easy enough to do through MLC02 – it took mere minutes. Hence, the alternative (reporting) is easy and should be pursued, as it might inspire others to be more efficient, thus preventing other minor climate actions.*
- Working on energy-efficient systems and creating energy leaderboards. *For a rank-and-file employee or student, this one may be a bit more difficult – for instance, if you're company/school is under contract to AWS, you may have no say in whether you can do the more efficient Google or Azure over AWS. But decision makers/supervisory employees/instructors should be keenly aware of this when signing up for plans (particularly as they're priced competitively), so for them it should easy to sign up for the efficient platforms. Similarly, they can easily set the tone by encouraging energy leaderboards, and set the standard for reporting.*
- Encouraging climate-friendly initiatives at conferences. *This is straightforward – as we've shown above, it's easy enough to make thoughtful energy decisions and follow best practices of efficiency reporting – so why not translate this to your conferences?*

In general, I agree with Hiller's doctrine – to me, it boils down to maintaining attention to detail and consciously making easy and good decisions at each decision point. The line about "easily available alternatives" imposes reasonability on the principle, and as we've seen above there are plenty of points at which an ordinary researcher can do the easy but good thing. It just takes a bit of attention to detail; I'm in favor.

References

[1] Peter Henderson et al. *Towards the Systematic Reporting of the Energy and Carbon Footprints of Machine Learning*. 2020. arXiv: 2002.05651 [cs.CY]. URL: https://www.jmlr.org/papers/volume21/20-312/20-312.pdf.

[2] Avram Hiller. "Climate Change and Individual Responsibility". In: *The Monist* 94.3 (2011), pp. 349–368. ISSN: 00269662. URL: http://www.jstor.org/stable/23039149.

[3] Alexandre Lacoste et al. *Quantifying the Carbon Emissions of Machine Learning*. 2019. arXiv: 1910.09700 [cs.CY].

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

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

[6] Emma Strubell, Ananya Ganesh, and Andrew McCallum. *Energy and Policy Considerations for Deep Learning in NLP*. 2019. arXiv: 1906.02243 [cs.CL].