Introduction

In this assignment, we will first implement a modified version of deep Q-learning from DeepMind's paper ([4] and [5]) 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 ([7]), 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.

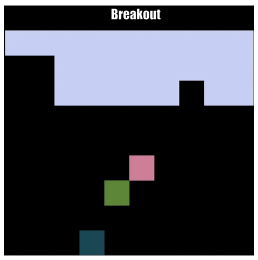


Figure 1: MinAtar's version of Breakout

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

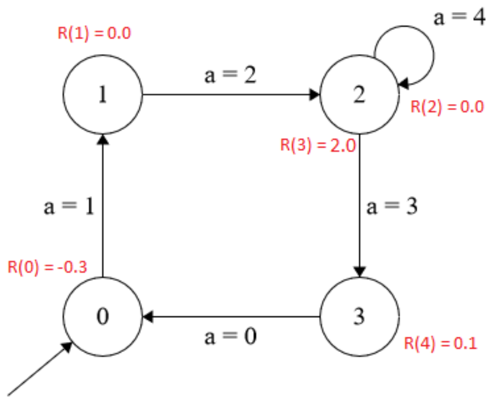


Figure 2: Example of a trajectory in the Test Environment

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.

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`.

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.

(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.

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 [4] 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?

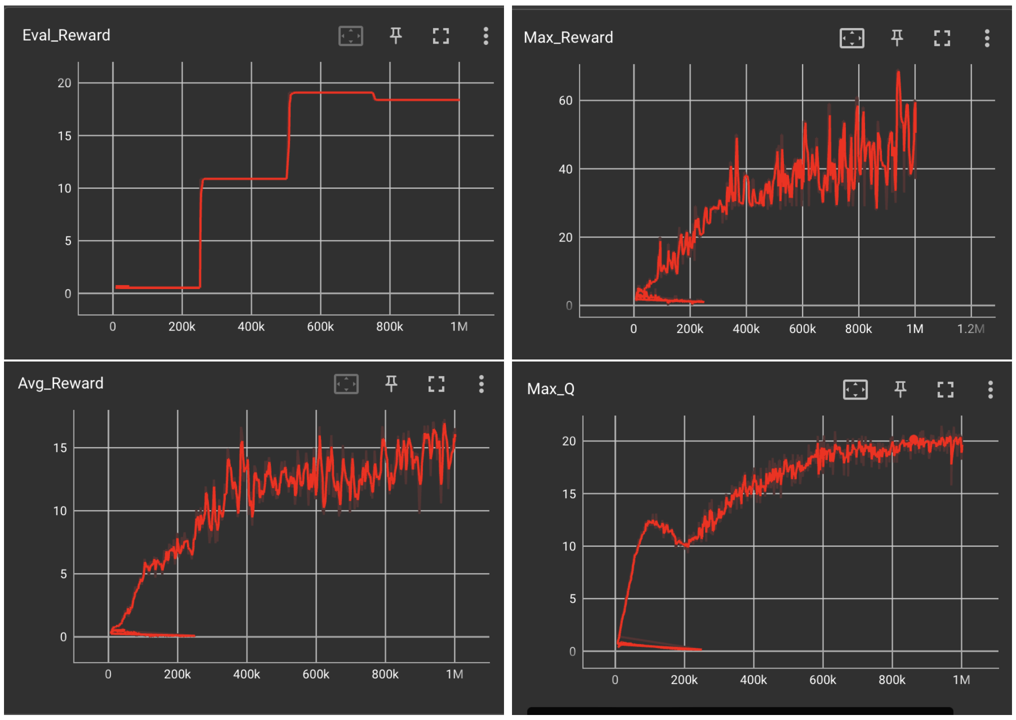
6 DQN on MinAtar (21 pts)

(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.

(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.

The following are some training tips:

- Rewards, scores, and the max of the Q values should 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:



(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?

(d) (**written**, 3 pts) Will the performance of DQN over time always improve monotonically? Why or why not?

7 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 training a model for the task of Pong. A single run on a Titan X GPU takes around 12 hours. 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

(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.

(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?

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

(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 timestep t while following policy π? (Provide an equation)

(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.

[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.

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 $f(s,a) = 1, \forall(s,a) \in \mathcal{S} \times \mathcal{A}$.

(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 $\mathbb{E}[X] = \mathbb{E}[\mathbb{E}[X \mid Y]]$.

After proving this result, you might already begin to appreciate why this represents a useful theoretical contribution. We are often interested in being able to control the gap between two value functions and this result provides a new mechanism for doing exactly that, when the value functions in question belong to two particular policies of the MDP.
Additionally, to see how this result is of practical importance as well, suppose the data-generating policy in the above identity π is some current policy we have in hand and π' represents some next policy we would like to optimize for; concretely, this scenario happens quite often when π is a neural network and π' denotes the same network with updated parameters. As is often the case with function approximation, there are sources of instability and, sometimes, even small parameter updates can lead to drastic changes in policy performance, potentially degrading (instead of improving) the performance of the current policy π. These realities of deep learning motivate a desire to occasionally be conservative in our updates and attempt to reach a new policy π' that provides only a modest improvement over π. Practical approaches can leverage the above identity to strike the right balance between making progress and maintaining stability.

9 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.*

(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?

(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?

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].

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