CP8319: Reinforcement Learning

Final Project

Tejas Vyas

R# 501127760

This report contains my responses for the Final Project Task for Dr. Nariman Farsad's CP8319 Reinforcement Learning course. The code, log files and videos for the below parts are available on the following GitHub Repository:

https://github.com/tejas1794/RL-Project

1. Test Environment

(a) (**written** 10 pts) 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.

Using the given state transition table:

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

Table 1: Transition table for the Test Environment

Assuming γ = 1.0 and for an episode lasting 5-time steps, the maximum reward sum we can reach can be calculated using the following ideas:

- We can get maximum reward following 2 → 1 = 2
- Following max reward, we need to go back to state 2 to get our maximum reward again i.e. we can only get the max reward 2 times in 1 episode when starting at state 0
- After this, we can try to get the maximum immediate reward by going back to state 0 (0.1)

Following above ideas, the trajectory can be defined as 0 → 2 → 1 → 2 → 1 → 0
This can used to calculate our max reward sum = 0 + 2 + 0 + 2 + 0.1 = **4.1**

2. Q-Learning and Value Function Approximation

(a) (**written** 5 pts) What is one benefit of representing the Q function as $\hat{q}(s; \mathbf{w}) \in \mathbb{R}^{|\mathcal{A}|}$?

As part of Q-learning, argmax for the action needs to be computed such that q value is highest at the state s'. Representing the Q function as $\hat{q}(s; \mathbf{w}) \in \mathbb{R}^{|\mathcal{A}|}$? means including a state as input and retrieving state-action values for all actions in the input state. This allows us to perform the argmax operation in a single pass instead of taking a linear number of passes (O(|A|)) when both state and action are included as input.

(b) (**coding/written** 5 pts) Implement the `get_action` and `update` functions in `q1_schedule.py`. Test your implementation by running `python q1_schedule.py`. Report if the test pass in the written portion and if you conducted any extra tests.

Performing the tests on q2_schedule.py gives us:

```
Test1: ok
Test2: ok
Test3: ok
```

(c) (**written** 5 pts) Consider the first of these two extremes: standard Q-Learning without a target network, whose weight update is given by equation **(0.2)** above. Is this weight update an instance of stochastic gradient descent (up to a constant factor of 2) on the objective $L(\mathbf{w})$ given by equation **(0.3)**? Argue mathematically why or why not.

The weight update in not and instance of stochastic gradient descent. We can review this by looking at objective function: $L(\mathbf{w}) = \mathop{\mathbb{E}}_{s,a,r,s' \sim \mathcal{D}} \left[\left(r + \gamma \max_{a' \in \mathcal{A}} \hat{q}\left(s', a'; \mathbf{w}\right) - \hat{q}(s, a; \mathbf{w}) \right)^2 \right]$

We know that stochastic gradient descent minimizes the objective function by depicting it in a form: $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} l(x, \mathbf{w})$. Therefore, if we can show L(**w**) is of this form, we can conclude this weight update would be an instance of stochastic gradient descent.

However, in this case, when we take a derivative, we get:

Assuming $\left[\left(r+\gamma \max_{a' \in A} \hat{q}(s', a'; w) - \hat{q}(s, a; w)\right)^2\right]$

$= l(s, a, r, s', w)$

we get eq" 0.3 $\Rightarrow$ $L(w) = \underset{s,a,r,s' \sim D}{E}(l(s, a, r, s', w))$

$\therefore \frac{\partial (L(w))}{\partial w} = 2\left[r+\gamma \max_{a' \in A} \hat{q}(s', a'; w) - \hat{q}(s, a; w)\right] \cdot \frac{\partial \left(r+\gamma \max_{a' \in A} \hat{q}(s', a'; w) - \hat{q}(s, a; w)\right)}{\partial w}$

$\Rightarrow$ as $\frac{\partial (r+\gamma \max_{a' \in A} (s', a'; w) - \hat{q}(s, a; w))}{\partial w}$ has a term reliant on w, $\neq \frac{-\partial (\hat{q}(s, a; w))}{\partial w}$

$\therefore w \leftarrow w - \alpha \nabla_w l(s, a, r, s', w)$ form is not possible by the weight update above.

(d) (**written** 5 pts) Now consider the second of these two extremes: using a target network that is never updated (i.e. held fixed throughout training). In this case, the weight update is given by equation **(0.4)** above, treating $\mathbf{w}^-$ as a constant. Is this weight update an instance of stochastic gradient descent (up to a constant factor of 2) on the objective $L^-(\mathbf{w})$ given by equation **(0.5)**? Argue mathematically why or why not.

In this case, the weight update would be an instance of SGD. We can show this by using the same idea above, except with objective function: $L^-(\mathbf{w}) = \underset{s,a,r,s' \sim D}{\mathbb{E}}\left[\left(r + \gamma \max_{a' \in A} \hat{q}\left(s', a'; \mathbf{w}^-\right) - \hat{q}(s, a; \mathbf{w})\right)^2\right]$

We can perform the same steps of derivative, however, in this case since w- is treated as constant, we do get the form,

$\therefore \frac{\partial (L(w))}{\partial w} = 2\left[r+\gamma \max_{a' \in A} \hat{q}(s', a'; w) - \hat{q}(s, a; w)\right] \cdot \frac{\partial \left(r+\gamma \max_{a' \in A} \hat{q}(s', a'; w) - \hat{q}(s, a; w)\right)}{\partial w}$

$\frac{\partial (L(w))}{\partial w} = \nabla_w (l(s, a, r, s', w)) = 2 \cdot \left(r+\gamma \max_{a' \in A} \hat{q}(s', a'; w^-) - \hat{q}(s, a; w)\right) - 1 \cdot \hat{q}(s, a; w)$

which is of form

$w \leftarrow w - 2\nabla_w l(s, a, r, s', w)$

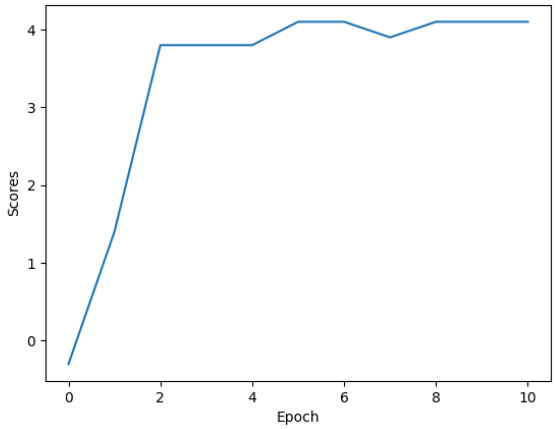
3. Linear Approximation and DQN

(a) (**coding/written**, 10 pts) First, we implement linear approximation in PyTorch. This question will set up the pipeline for the remainder of the project. You'll need to implement the following functions in `q3_linear.py` (please read through `q3_linear.py`):

- `add_placeholders_op`
- `get_q_values_op`
- `add_update_target_op`
- `add_loss_op`
- `add_optimizer_op`

Test your code by running `python q3_linear.py` **locally on CPU**. This will run linear approximation with PyTorch on the test environment from Question 1. Running this implementation should only take a minute or two. Do you reach the optimal achievable reward on the test environment? Attach the plot `scores.png` from the directory `results/q3_linear` to your writeup.

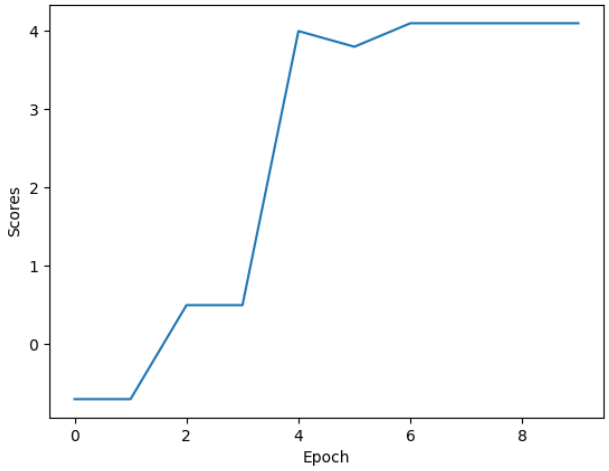
The optimal reward of 4.1 was reached after implementing the functions and running the script. The plot below shows the training epochs and scores:



(b) (**coding/written**, 10 pts) Implement the deep Q-network as described in [2] by implementing `get_q_ values_op` in `q4_nature.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 q4_nature.py`. Running this implementation should only take a minute or two. Attach the plot of scores, `scores.png`, from the directory `results/q4_nature` to your writeup. Compare this model with linear approximation. How do the final performances compare? How about the training time?

The optimal reward of 4.1 was reached after implementing the functions and running the script. When compared against linear run, the results were similar, however, the performance was much better. While the DQN run took 11 seconds, linear was able to achieve the results in simply 6 seconds on CPU

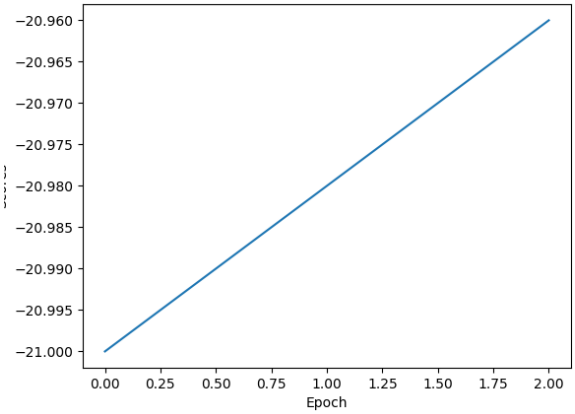
The plot below shows the training epochs and scores:



4. Testing on Atari

(a) (**coding/written**, 10 pts). Now we're ready to train on the Atari `Pong-v0` environment. First, launch linear approximation on pong with `python q5_train_atari_linear.py`. This will train the model for 500,000 steps and should take approximately a few hours on CPU. Briefly qualitatively describe how your agent's performance changes over the course of training. Also plot the total reward the agent achieves as it trains and present it in your report. Do you think that training for a larger number of steps would likely yield further improvements in performance? Explain your answer.

Running the model on CPU for 500,000 steps, barely changed the reward, and was within the error range. The reward was increased from -21 to -20.96, but no significant improvement was observed. The plot obtained through training can be seen below:



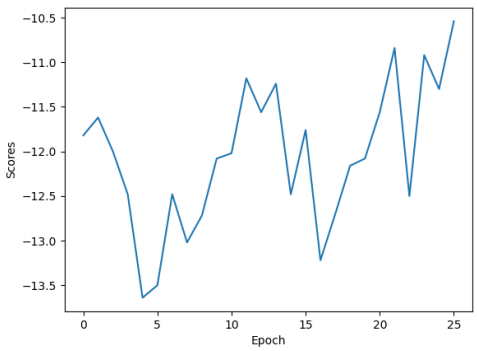
While the model had a minor improvement in scores, running the model training for a larger number of steps would likely not yield further improvements in performance as the linear model doesn't appear to be responding well to the complexity of the given task.

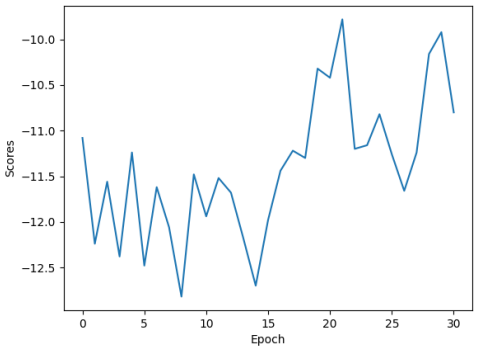
(b) (**coding/written**, 10 pts). In this question, we'll train the agent with DeepMind's architecture on the Atari `Pong-v0` environment. Run `python q6_train_atari_nature.py` (preferably on a GPU). We have trained the model for a few million steps and this is the model that is initially loaded. Feel free to continue the training and improve the model. Attach the plot `scores.png` from the directory `results/q6_train_atari_nature` to your writeup. Also check the `results/q6_train_atari_nature/monitor` folder for sample videos of the agent playing at various stages of training. The DeepMind paper claims average human performance is −3. The trained model performs around −11. See the files `SamplePerformanceBeforeTraining.mp4`, `SamplePerformanceAfterAFewMillionTraining.mp4` that show what the performance was initially and after training for a few million iterations. Please note that to continue the training to perform past human-level performance, you still need to train this for another half day to a day on a GPU. Report your plots and observations.

In addition to the earlier trained weights of a few million steps, I performed 2 additional sets of training with 25 and 30 shorter epochs (50000 step epoch) i.e., 1.25 and 1.5 million steps totaling 2.75 million steps.

I received the model that was able to reach a higher reward value of **-9.92 +/- 0.60**, with one episode evaluation reaching as high as **-9.76**.

The learning curves are presented below:





---

5. Train Your Best Model

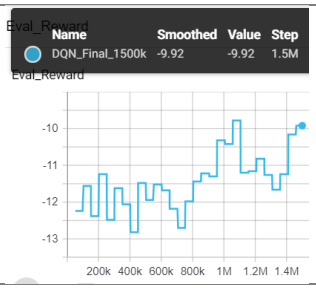
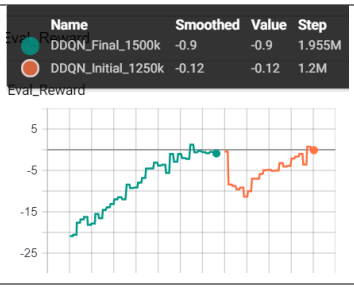
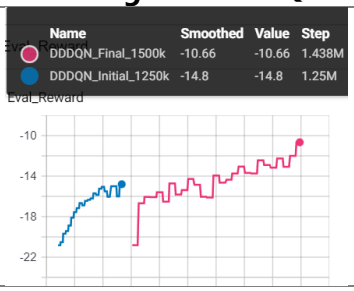
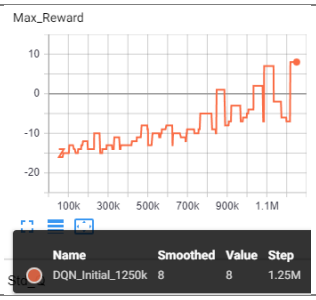
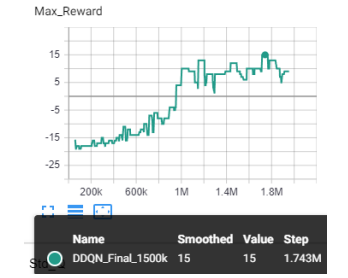
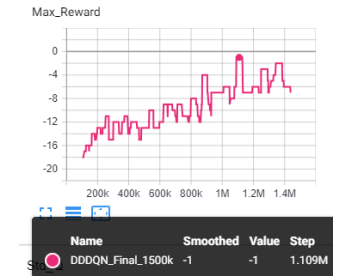
(30 points) **(Train Your Best Model)** Now that you have trained the DeepMind's architecture, feel free to implement any other method(s) and report the best set of results you can achieve on the game of pong. A big portion of the grade associated with part of the project will be graded based on the relative performance achieved by other students in the class. In your report, clearly describe the method(s) that you implemented and the performance you achieved on the best method. Also includes plots for the total reward obtained as the agent learns.

For this part of the project, I tried 2 models to retrieve better scores than baseline DQN from the two papers [Deep Reinforcement Learning with Double Q-learning] and [Dueling Network Architectures for Deep Reinforcement Learning]:

- Double DQN
- Dueling Double DQN

I implemented both algorithms by tweaking the nature paper model [please see *q6_atari_nature_doubledqn.py* (updates loss) and *q6_atari_nature_dddqn.py* (updates model architecture)] and used the same hyper-params as the original nature paper, training in 2 instances (1.25 million and 1.5 million steps respectively).

Here's a comparison of all 3 techniques (DQN, DDQN, DDDQN) through tensorboard and my final max score:

	DQN	**Double DQN**	**Dueling Double DQN**
Evaluation Reward Curve			
Final Reward	**-9.92 +/- 0.60**	**-0.90 +/- 0.97**	**-10.66 +/- 0.51**
Max Reward Curve			
Max Reward	**-9.92**	**+15**	**-1**

My best results were from the Double DQN model, after training it 3.1 million steps, reaching the final reward of **-0.90 +/- 0.97**, and the model was able to reach up to **+1.22** evaluation reward and max reward of **+15** during training. I believe it can go higher with further training, but my results were constrained by the limited training time.