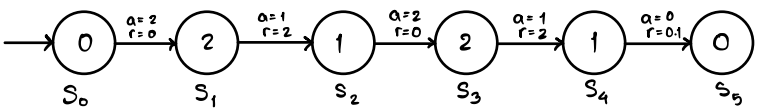
(a) (**written** 10 pts) What is the maximum sum of rewards that can be achieved in a single trajectory in the test environment, assuming $\gamma = 1$? Show first that this value is attainable in a single trajectory, and then briefly argue why no other trajectory can achieve greater cumulative reward.

An episode lasts for 5 time steps and we always start at State 0.

Maximum sum of rewards achievable is 4.1.
State 2 has the largest rewards compared to the other states, thus it must be in the episode. As the max reward from this state is a reward of 2 when the agent transitions to state 1, we must utilize this in the episode. Now, to achieve this reward value again, we must transition back to state 2. Even though the second largest reward value is 1 when transitioning from state 2 to state 3 transitioning back to state 2 from both states 1 and 3 give no rewards. So, with an episode of 5 time steps the highest reward achievable is when we execute the best action with the highest reward twice. Thus, to achieve the maximum reward, the agent must follow this episode:



Sum of rewards = 4.1

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

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

Representing the Q function as $\hat{q}(s;\vec{w}) \in \mathbb{R}^{|A|}$ allows the network to return the state-action values for each action $a \in A$, thus no longer needing to compute the argmax which would have required a further $O(|A|)$ computations to find the optimal action.

(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.

All the tests passed successfully and no further tests were conducted.

(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.

Want to prove $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} \hat{q}\left(s', a'; \mathbf{w}\right) - \hat{q}\left(s, a; \mathbf{w}\right) \right) \nabla_{\mathbf{w}} \hat{q}(s, a; \mathbf{w})$. is not an instance of stochastic gradient descent.

↪ Assume that this update rule is an instance of SGD.

For an objective of the form $J(\mathbf{w}) = \mathbb{E}_{x \sim \mathcal{D}}[l(x, \mathbf{w})]$
Stochastic gradient descent update rule: $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} l(x, \mathbf{w})$

Given $L(\mathbf{w}) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}} \left[\left(r + \gamma \max_{a' \in \mathcal{A}} \hat{q}\left(s', a'; \mathbf{w}\right) - \hat{q}(s, a; \mathbf{w}) \right)^2 \right]$ —— 0.3

We can take $l(s,a,r,s') = \left(r + \gamma \max_{a' \in A} \hat{q}(s',a';\vec{w}) - \hat{q}(s,a;\vec{w}) \right)^2$ as our loss function.

Thus, we want to show $\nabla_w l(s,a,r,s') = -\left(r + \gamma \max_{a' \in A} \hat{q}(s',a';\vec{w}) - \hat{q}(s,a;\vec{w}) \right) \nabla_w \hat{q}(s,a;\vec{w})$

However, $\nabla_w l(s,a,r,s') = 2\left(r + \gamma \max_{a' \in A} \hat{q}(s',a';\vec{w}) - \hat{q}(s,a;\vec{w}) \right) \nabla_w \left(\gamma \max_{a' \in A} \hat{q}(s',a';\vec{w}) - \hat{q}(s,a;\vec{w}) \right)$

As $\nabla_w \gamma \max_{a' \in A} \hat{q}(s',a';\vec{w}) \neq 0$,

This leads to a contradiction where the computed update rule does not follow the form of SGD. Thus, it is not an instance of SGD.

QED

(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.

Want to prove $\mathrm{w} \leftarrow \mathrm{w} + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} \hat{q}\left(s', a'; \mathrm{w}^-\right) - \hat{q}\left(s, a; \mathrm{w}\right) \right) \nabla_{\mathrm{w}} \hat{q}(s, a; \mathrm{w}).$ is an instance of stochastic gradient descent.

↪ Assume that this update rule is an instance of SGD.

For an objective of the form $J(\mathbf{w}) = \mathbb{E}_{x \sim \mathcal{D}}[l(x, \mathbf{w})]$
Stochastic gradient descent update rule: $\mathrm{w} \leftarrow \mathrm{w} - \alpha \nabla_{\mathrm{w}} l(x, \mathrm{w})$

Given $L^-(\mathrm{w}) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}} \left[\left(r + \gamma \max_{a' \in \mathcal{A}} \hat{q}\left(s', a'; \mathrm{w}^-\right) - \hat{q}(s, a; \mathrm{w}) \right)^2 \right]$ —— 0.5

We can take $l(s,a,r,s') = \left(r + \gamma \max_{a' \in A} \hat{q}(s', a'; \vec{w}^-) - \hat{q}(s, a; \vec{w}) \right)^2$ as our loss function.

Thus, we want to show $\nabla_w l(s,a,r,s') = -\left(r + \gamma \max_{a' \in A} \hat{q}(s', a'; \vec{w}^-) - \hat{q}(s, a; \vec{w}) \right) \nabla_w \hat{q}(s, a; \vec{w})$

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

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

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

This gives us the following update rule:

$$\vec{w} \leftarrow \vec{w} - 2\alpha \left(r + \gamma \max_{a' \in A} \hat{q}(s', a'; \vec{w}^-) - \hat{q}(s, a; \vec{w}) \right) \nabla_w \hat{q}(s, a; \vec{w})$$

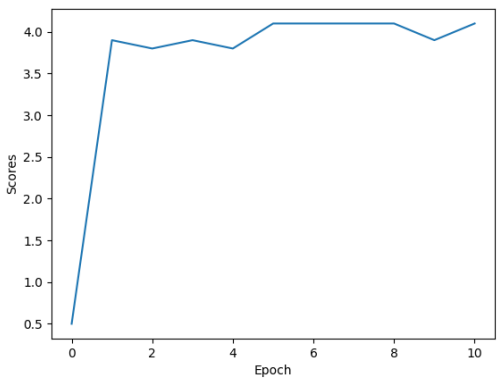
Which follows the form of SGD.
Thus, it is an instance of SGD. QED

(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`

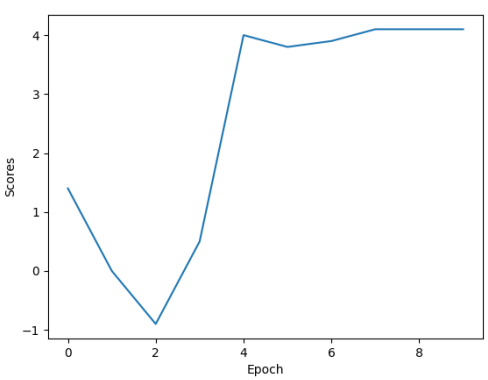
Test your code by running `python q3_linear.py` **locally on CPU**. This will run linear approximation with PyTorch on the test environment from Question 1. Running this implementation should only take a minute or two. Do you reach the optimal achievable reward on the test environment? Attach the plot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

Yes, an optimal reward of 4.1 is achieved.



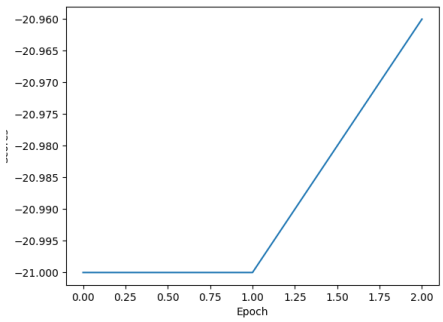
(b) (**coding/written**, 10 pts) Implement the deep Q-network as described in [2] by implementing `get_q_values_op` in `q4_nature.py`. The rest of the code inherits from what you wrote for linear approximation. Test your implementation **locally on CPU** on the test environment by running `python q4_nature.py`. 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 maximum reward of 4.1 was also achieved by this model. Thus, both were able to achieve optimal performance. However, this network took longer to train, needing around 13 seconds to train while the linear model only needed 5 seconds.



(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.

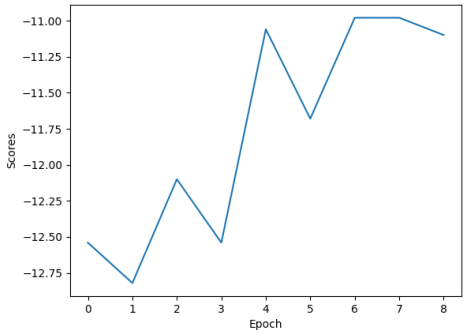
Upon observation of the generated videos demonstrating the performance of the model, we can see that there have not been any noticeable improvements over the course of training. Throughout the training, the total reward achieved does not drastically change. This can be observed in the plot generated.



Further training will likely not yield any more significant improvement as it is evident that the linear model is not complex enough for this task.

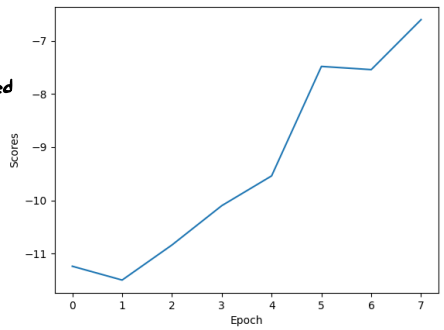
(b) (**coding/written**, 10 pts). In this question, we'll train the agent with DeepMind's architecture on the Atari `Pong-v0` environment. Run `python q6_train_atari_nature.py` (preferably on a GPU). We have trained the model for a few million steps and this is the model that is initially loaded. Feel 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.

I have trained this model for 2 separate training sessions and have included the plots below:



2 million steps

continued →



1,750,000 steps

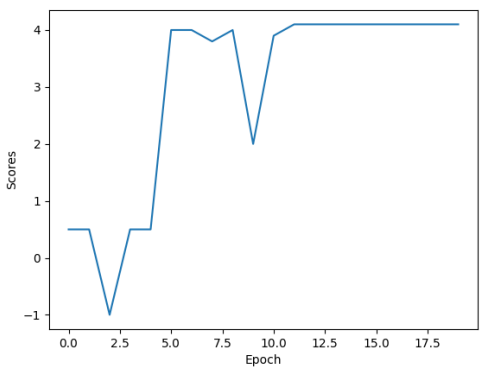
Average Reward at the end of the sessions was -6.60 ± 0.92

5. (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.

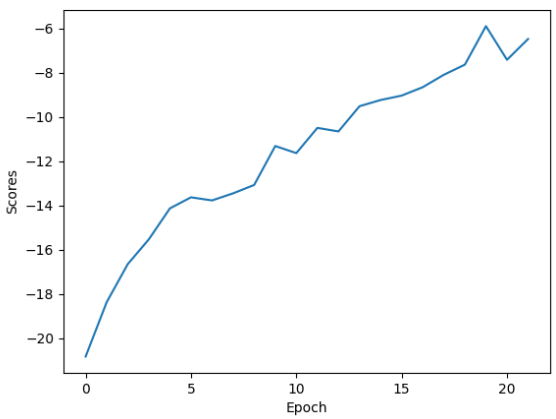
To improve upon the nature paper, I implemented a dueling network architecture. As the project already made use of a replay buffer as well as the double Q network architecture, involving a Q network and a target network, the resultant network architecture was a Dueling Double Deep Q-Learning Network (DDDQN).

The "Dueling Network Architecture for Deep Reinforcement Learning" paper from Google DeepMind is the paper that first introduced this dueling architecture.

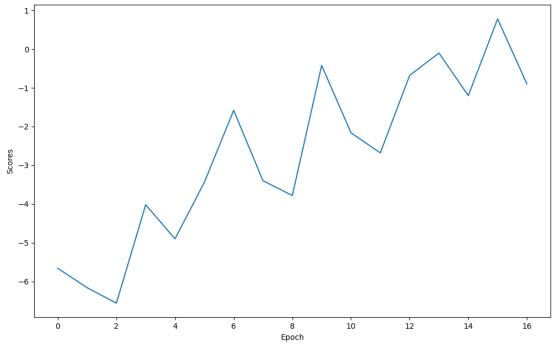
I have added the implementation of this network in the q7_dddqn_torch.py file, where the model is first trained and evaluated on the test environment. As shown in the plot below, it is able to achieve optimal performance by achieving the max reward of 4.1. Note, however, that this network took longer to train and achieve this than the nature paper.

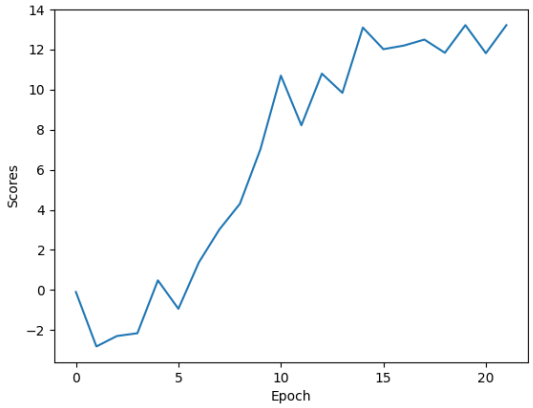


Next, the DDDQN model was trained on the "pong-v0" network using the same hyperparameters as the nature paper.

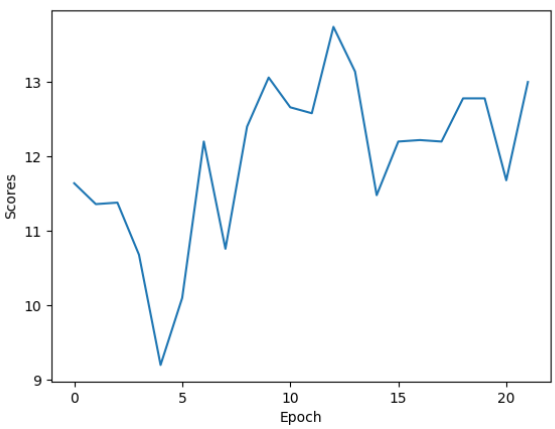


This is the plot of the rewards during the first training session of just over 20 epochs with each epoch being 250,000 training steps. The network learned quite fast and well.





Training continued for two more training sessions and managed to begin achieving a positive average reward, allowing the agent to begin winning the games.



During a fourth and final training session, the increase in rewards had slowed down and the final average reward that the model achieved at the end of training was an average reward of 13 ± 0.53