CS 234 Winter 2019: Assignment #2

Li Quan Khoo (SCPD)

Introduction

1 Test Environment (5 pts)

1. (**written** 5pts) By inspection, action 4 of each state has the same reward as one other action in that state, which means they can be ignored in this analysis. Where the rows are the current state s, the columns are the next states s', and the entries are the rewards:

	0	1	2	3
0	0.1	-0.2	0.0	-0.1
1	0.1	-0.2	0.0	-0.1
2	-1.0	2.0	0.0	1.0
3	0.1	-0.2	0.0	-0.1

By inspection, the optimal sequence of states is $s_0, s_2, s_1, s_2, s_1, s_0$, with a reward sequence of $0, 2, 0, 2, 0.1$, for a total of 4.1.

Observations:

- There are three non-trivial rewards in the transition matrix. Two positive, one negative.
- All other rewards and penalties in a state do not affect the choice of path, as no sequence of these is sufficient to cancel out a non-trivial reward in a sequence length of 5.
- Disregarding the starting state condition, the maximal reward in a sequence of two forms a cycle between s_2 and s_1, with reward of 2. This is easily determined by inspection. Meaning for a sequence of 4, we just run this cycle twice.

Reasoning: We follow a greedy policy. The initial s_0 has two non-negative actions. The self-transition results in a total reward of 0.5. a_2 is the shortest path that leads to s_2, which is where both positive non-trivial rewards originate from. Both of these high reward actions have return transitions of 0 to s_2 or otherwise have trivial rewards. The greedy choice is to take action a_1 twice from s_2, by returning to s_2 via the shortest path. Finally, when we consider the final transition in s_1, the only reasonable choice is the greedy one, netting us the final return of 0.1.

2 Q-learning (12 pts)

1. (**written** 3pts) Experience replay allows for learning from the same (s, a, s', r) tuple multiple times. As discussed in the lectures, Q-learning does not propagate information backwards like Monte-Carlo methods do, since the trajectory has been lost. Depending on how we minibatch the tuples, the data may be less temporally-correlated, which may also give us better convergence properties.

2. (**written** 3pts) Off-policy Q-learning with function approximation has no convergence guarantees, and using experience-replay means we are necessarily operating off-policy i.e. we are using (s, a, s', r) tuples collected by running a previous policy. The target network makes sure that the target Q-function (which is supposed to be approximating the optimal Q function) doesn't change within each 'batch' during training, in order to improve the chances of the algorithm eventually converging onto the optimal Q function and hence the optimal policy.

3. (**written** 3pts) I assume the question is asking about the benefit of representing the Q-function as a as a parametric function approximator $\tilde{q}(s, \mathbf{w})$ e.g. a neural net. Here I assume $\mathbf{w}$ is a finite vector in $\mathbb{R}^N$. The Q-function is a mapping $Q : S \times A \to \mathbb{R}^K$. The most obvious choice for this representation is that when even S or A or both are either countably infinite or uncountably infinite (continuous), the set of parameters $\mathbf{w}$ is still finite, and hence is always representable in finite memory.

4. (**coding** 3pts) -code-

3 Linear Approximation (26 pts)

1. (**written** 3pts) Since $x(s,a)$ is a one-hot vector, $\hat{q}(s,a;\mathbf{w}) = \mathbf{w}^T x(s,a) = \mathbf{w}_{s,a}$, which corresponds to the single 1 entry in x.

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

$$\mathbf{w}_{s,a} \leftarrow \mathbf{w}_{s,a} + \alpha \left(r + \gamma \max_{a' \in A} \mathbf{w}_{s',a'} - \mathbf{w}_{s,a} \right) 1$$

We can see that the above update rule becomes independent of $\hat{q}$, and so our representation becomes a table of size $|S| \times |A|$, which is represented by the vector $\mathbf{w}$ of length $|S| \times |A|$. Let $\mathbf{w}_{s,a} = Q(s,a)\ \forall s \in S, a \in A$ and we have the equation we need. QED.

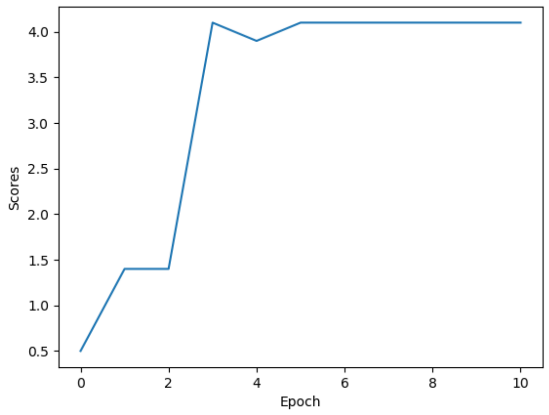
2. (**written** 3pts) Given: $\hat{q}(s,a;\mathbf{w}) = \mathbf{w}^T x(s,a)$

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

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \left(r + \gamma \max_{a' \in A} \mathbf{w}^T x(s', a') - \mathbf{w}^T x(s,a) \right) x(s,a)$$

3. (**coding** 15pts) -code-

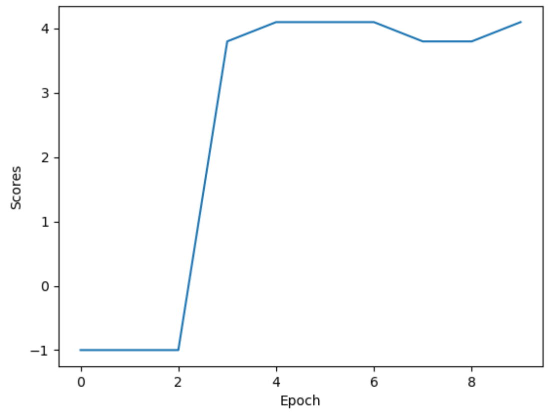
4. (**written** 5pts) Yes, the model converges to a final score of 4.1.



4 Implementing DeepMind's DQN (15 pts)

1. (**coding** 10pts) -code-

2. (**written** 5pts) The models have comparable performance, but the Q-network takes longer to train.



5 DQN on Atari (27 pts)

1. (**written** 2pts) The agent requires at least 2 frames to determine the velocity of the ball. The other argument that was made by Mnih 2015 relies on the assumption that the computational load for running the emulator (environment) forward by one frame is less than computing an action for the agent. It allows the agent to observe roughly 4 times as many games (states) without also increasing the computational load by an equal amount. Having the agent make only a single every 4 frames should also have a negligible performance impact, since these Atari games are meant to be played by humans in the first place.

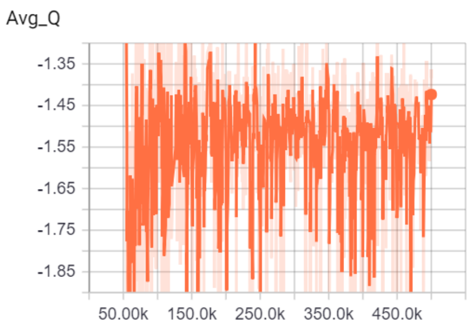
2. (**written** 5pts) The number of parameters in a conv2d layer is kernelHeight * kernelWidth * nInputChannels * nFilters + nBias, where the number of bias parameters is equal to the number of filters the layer has.

   The number of parameters in an FC layer is nInput * nOutput + nBias, where the number of bias parameters is equal to the number of outputs the layer has.

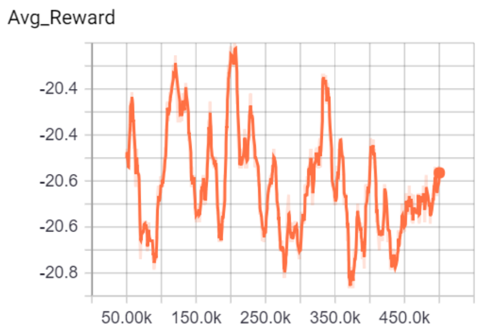
Layer	Output shape	Parameters
(input)	(80,80,4)	0
Conv1 filters=32, kernel=8, stride=4, pad=same	(20,20,32)	8*8*4*32+32 = 8192+32
Conv2 filters=64, kernel=4, stride=2, pad=same	(10,10,64)	4*4*32*64+64 = 32768+64
Conv3 filters=64, kernel=3, stride=1, pad=same	(10,10,64)	3*3*64*64+64 = 36864+64
FC1	(512)	6400*512+512 = 3276800+512
FC2	(6)	512*6+6 = 3072+6

   Total = 3358374

3. (**coding and written** 5pts) The best the model did was get a max reward of -18, with mean reward of -21. This is comparable to the SARSA model in Mnih 2015, section 5.3. Perhaps this is because we are simply computing a set of 80*80*4=25600 weights for the states, the model is not expressive enough to do well, considering how large the state space actually is, even in grayscale.

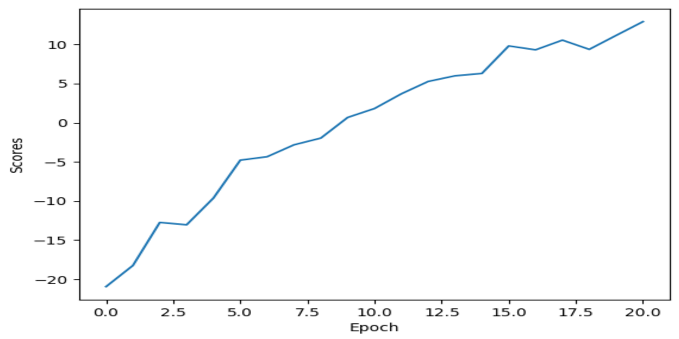


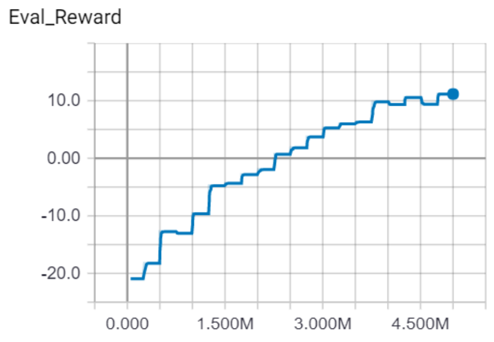
(a) Average Q

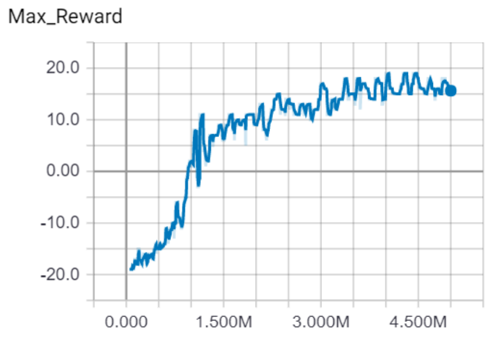


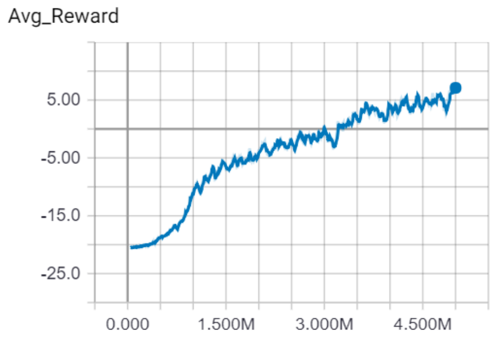
(b) Average r

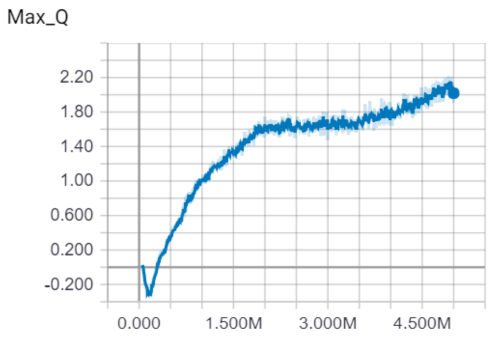
4. (**coding and written** 10 pts)











5. (**written** 5pts) The DeepMind architecture, with its convolutional layers, is much more suited to generalizing over the possible graphical inputs. It is also a higher-capacity model, just from the number of parameters in the FC layers, so it appears to not suffer as much as the linear model from underparameterization.

6 Real world RL with neural networks (10 pts)

1. (**written** 4pts) SGD assumes that the data we are fitting on is iid (independently and identically distributed) so that the loss function we are fitting on is stationary w.r.t the data. Since our data comes from a stream of consecutive environment interactions, this is unlikely to be the case. Depending on our choice of learning rate and choice of optimization (e.g. momentum etc) we might not converge to an optimum at all.

2. (**written** 3pts) We need to bootstrap $V(s')$, either by running our policy and taking a single-sample estimate (SARSA), i.e. $Q(s', a'; \mathbf{w})$, or by bootstrapping with an off-policy maximization over Q (Q-learning), i.e. $max_{a'}[Q(s', a'; \mathbf{w})]$.

3. (**written** 3pts) Unless the state space is also finite, which brings us to the tabular case, SARSA converges under linear function approximation only, and otherwise there are no convergence guarantees.