

CS 234: Assignment #2

Due date: 5/5 11:59 PM 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. **However, each student must finish the problem set and programming assignment individually, and must turn in her/his assignment.** We ask that you abide by the university Honor Code and that of the Computer Science department, and make sure that all of your submitted work is done by yourself. 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 review any additional instructions posted on the assignment page at <http://cs234.stanford.edu/assignment2>. 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.**

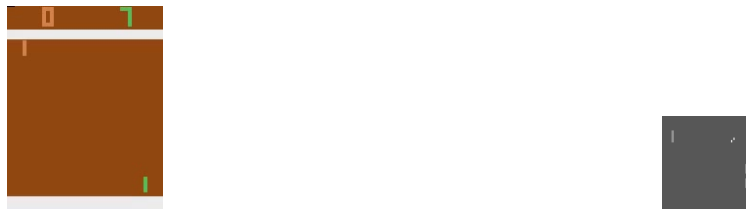
You'll need to download the starter code and fill the appropriate functions following the instructions from the handout and the code's documentation. Training DeepMind's network on Pong takes roughly **12 hours on GPU**, so **please start early!** We will give you access to an Azure GPU cluster. You'll find the setup instructions on the course assignment page.

1 Introduction

In this assignment we will implement **deep Q learning**, following DeepMind's paper ([2] and [3]) that learns to play Atari from raw pixels. The purpose is to understand the effectiveness of deep neural network as well as some of the techniques used in practice to stabilize training and achieve better performance. You'll also have to get comfortable with `Tensorflow`. We will train our networks on the `Pong-v0` environment from OpenAI gym, but the code can easily be applied to any other environment.

The Atari environment from OpenAI gym returns observations of size $(210 \times 160 \times 3)$, the last dimension corresponding to the RGB channels filled with values between 0 and 255 (`uint8`). To reduce the dimension of the input image, we will apply some preprocessing to the observation:

- each time we execute an action, we keep doing it for 4 time steps. We return a pixel-wise max-pooling of the 4 consecutive frames. This reduces the frequency of decisions without impacting the performance too much and enables us to play 4 times more games while training.
- crop the image, convert it to grey scale



(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 1: Pong-v0 environment

In Pong, one player wins if the ball passes through the other player. Winning a game gives a reward of 1, while losing gives a negative reward of -1. An episode is over when one of the two players reaches 21 wins. Thus, the final score is between -21 (lost all) or +21 (won all). Our agent plays against a decent hard-coded AI player. Human performance is -3. If you go to the end of the homework successfully, you will train an AI agent with super-human performance, reaching at least +5 (hopefully more!).

2 Q-learning

Tabular setting In the *tabular setting* (see assignment 1), we maintained a table $Q(s, a)$ for each tuple state-action. Given an experience sample (s, a, r, s') , our update rule was

$$Q(s, a) = Q(s, a) + \alpha \left(r + \gamma \max_{a' \in A} Q(s', a') - Q(s, a) \right), \quad (1)$$

where $\alpha \in \mathbb{R}$ is the learning rate, γ the discount factor.

Approximation setting Due to the scale of Atari environments, we cannot reasonably learn and store a Q value for each state-action tuple. We will represent our Q values as a function $Q_\theta(s, a)$ where θ are parameters of the function (typically, neural network's weights). In this *approximation* setting, our update rule becomes

$$\theta = \theta + \alpha \left(r + \gamma \max_{a' \in A} Q_\theta(s', a') - Q_\theta(s, a) \right) \nabla_\theta Q_\theta(s, a). \quad (2)$$

In other words, we are trying to minimize

$$L(\theta) = \mathbf{E}_{s,a,r,s'} \left[r + \gamma \max_{a' \in A} Q_\theta(s', a') - Q_\theta(s, a) \right]^2 \quad (3)$$

Target Network DeepMind's paper [2] [3] maintains two sets of parameters, θ (to compute $Q(s, a)$) and θ^- (target network, to compute $Q(s', a')$) s.t. our update rule becomes

$$\theta = \theta + \alpha \left(r + \gamma \max_{a' \in A} Q_{\theta^-}(s', a') - Q_\theta(s, a) \right) \nabla_\theta Q_\theta(s, a). \quad (4)$$

The target network's parameters are updated with the Q-network's parameters occasionally and are kept fixed between individual updates. Note that when computing the update, we don't compute gradients with respect to θ^- (these are considered fixed weights).

Replay Memory As we play, we store our transitions (s, a, r, s') in a buffer. Old examples are deleted as we store new transitions. To update our parameters, we *sample* a minibatch from the buffer and perform a stochastic gradient descent update.

ϵ -greedy exploration strategy During training, we use an ϵ -greedy strategy. DeepMind's paper [2] [3] decreases ϵ from 1 to 0.1 during the first million steps. At test time, the agent choses a random action with probability $\epsilon_{soft} = 0.05$.

There are several things to be noted:

1. In this assignment, we will update θ every `learning_freq` steps by using a minibatch of experiences sampled from the replay buffer.
2. DeepMind's deep Q network takes as input the state s and outputs a vector of size = number of actions. In the pong environment, we have 6 actions, thus $Q_\theta(s) \in \mathbb{R}^6$.
3. The input of the deep Q network is the concatenation 4 consecutive steps, which results in an input after preprocessing of shape $(80 \times 80 \times 4)$.

3 Setup - TestEnv (15pts)

Before running our code on Atari, it is crucial to test our code on a test environment. You should be able to run your models on CPU in no more than a few minutes on the following environment:

- 3 states: 1, 2, 3
- 4 actions: 1, 2, 3, 4. Action $1 \leq i \leq 3$ goes to state i , while action 4 makes the agent stay in the same state.
- one episode lasts 5 time steps and always starts in state 1.
- rewards depend on history. Going to state i gives a reward $r(i)$ such that $r(1) = -0.1, r(2) = 0, r(3) = 0.1$. If we were in state 2 before the transition, then the reward is multiplied by -10 .

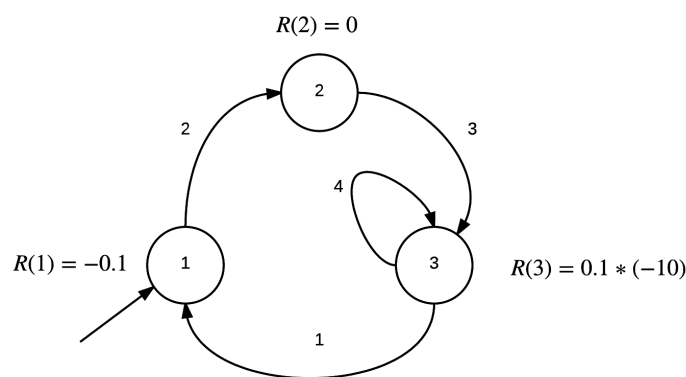


Figure 2: Example of a path in the Test Environment

You can check the code of the environment in `utils/test_env.py`. We'll use the same learning strategy as for Atari.

1. (written 2pts) General - Why do we use the last 4 time steps as input to the network?

Solution: In the Test Env, one of the optimal strategies is $1 \rightarrow 2 \rightarrow 1 \rightarrow 2 \rightarrow 1 \rightarrow 3$. Thus, when we are in state 1, the optimal action depends on the history.

In Pong, if we only have one frame, we cannot determine the velocity nor the direction of the ball, which are essential to play correctly. Thus, to take a good decision, we need different time steps so that we can evaluate the trajectory of the ball.

More generally, we combine multiple steps so that we transform our world dynamics into a (almost) *markovian* world. The choice of 4 is arbitrary and could be adapted in other problems.

2. (written 2pts) General - What would be a benefit of experience replay? What would be a benefit of the target network?

Solution: Deep Q learning suffers from stability issues:

- (a) strongly correlated samples break the iid assumption necessary to assume convergence of SGD methods, that are based on estimation of gradients.
- (b) we forget rare experiences from which the agent could learn more

- (c) non-stationary distribution of samples, as samples are drawn from the experience of an evolving agent whose policy can quickly change, which can be problematic for deep learning methods.

Experience replay answers

- (a) by sampling independent samples,
- (b) by keeping a large history
- (c) by keeping examples from all the agent's policies, acting as a regularization of the policy over time.

On the other hand, **the target network**

- (a) avoids the oscillation of the target policy (issue c.)
- (b) breaks correlation with the Q-network, that would create a lot of variance in the loss estimation (oscillation).

Note that experience replay, by having samples from different policies, forces us to use an off-policy algorithm like Q-learning.

3. (**written** 2pts) General - What would be a benefit of representing the Q function as $Q(s) \in \mathbb{R}^{\text{num actions}}$ instead of $Q(s, a)$?

Solution: As we have finite number of actions n , it is computationally more efficient to compute a vector $\in \mathbb{R}^n$ from the state than compute n scalar.

4. (**written** 5pts) What's the optimal achievable reward for this environment?

Solution: The optimal reward of the Test environment is

$$2.1$$

To prove this, let's prove an **upper bound** of 2.1 with 3 key observations

- first, the maximum reward we can achieve is 1 when we do $2 \rightarrow 1$.
- second, after having performed this optimal transition, we have to wait at least one step to execute it again.

As we have 5 steps, we can execute 2 optimal moves. Executing less than 2 would yield a strictly smaller result. We need to go to 2 twice, which gives us 0 reward on 2 steps. Thus, we know that 4 steps gives us a max of 2. Then, the best reward we can achieve that is not an optimal move is 0.1, which yields an upper bound of 2.1.

Considering the path $1 \rightarrow 2 \rightarrow 1 \rightarrow 2 \rightarrow 1 \rightarrow 3$ proves that we can **achieve** this upper bound.

5. (**coding** 4pts) Implement `get_action` and `update` functions in `q1_schedule.py`. Test your implementation by running `python q1_schedule.py`.

4 Linear Approximation (30pts)

1. (**written** 5pts) Show that (1) and (2) are exactly the same when $Q_\theta(s, a) = \theta^T \phi(s, a)$, where $\theta \in \mathbf{R}^{|S||A|}$, $\phi : S \times A \rightarrow \mathbf{R}^{|S||A|}$, and

$$\phi(s, a)_{s', a'} = \begin{cases} 1 & \text{if } s' = s, a' = a \\ 0 & \text{otherwise} \end{cases}$$

Solution:

Let's denote $\theta_{s,a}$ the component of θ that corresponds to the entry equal to 1 in $\phi(s, a)$. Let's write (2) for $\theta_{s,a}$:

$$\theta_{s,a} = \theta_{s,a} + \alpha(r + \gamma \max_{a' \in A} \theta_{s', a'}) \nabla_{\theta_{s,a}} \theta_{s,a}$$

And we obtain (1) for $\theta_{s,a}$ that we can identify to $Q(s, a)$.

2. (**written** 5pts) Derive the gradient with regard to the value function parameter $\theta \in \mathbb{R}^n$ given $Q_\theta(s, a) = \theta^T \phi(s, a)$ for any function $\phi(s, a) \mapsto x \in \mathbb{R}^n$ and write the update rule for θ .

Solution: We have

$$\begin{aligned} \nabla_\theta Q_\theta(s, a) &= \nabla_\theta \theta^T \phi(s, a) \\ &= \phi(s, a) \end{aligned}$$

And the update rule becomes

$$\theta = \theta + \alpha \left(r + \gamma \max_{a' \in A} Q_\theta(s', a') - Q_\theta(s, a) \right) \phi(s, a) \quad (5)$$

3. (**coding** 15pts) Implement linear approximation with Tensorflow. Tensorflow will be particularly useful for more complex functions, and this question will setup the whole pipeline (this step is crucial for the rest of the assignment). You'll need to fill the relevant functions in `q2_linear.py`. Test your code with `python q2_linear.py` that will run linear approximation with Tensorflow on the test environment.
4. (**written** 5pts) Do you reach the optimal achievable reward on the test env? Attach the plot `scores.png` to your writeup.

Solution:

Each experiment creates a folder in `results/` and load a config from `configs/`. You can edit the configuration file.

5 Implementing DeepMind's DQN (40pts)

1. (**coding** 10pts) Implement the deep Q-network as described in [2] by filling `get_q_values_op` in `q3_nature.py`. The rest of the code inherits from what you wrote for linear approximation. Be sure that it works! Test your implementation on the test environment by running `python q3_nature.py`.

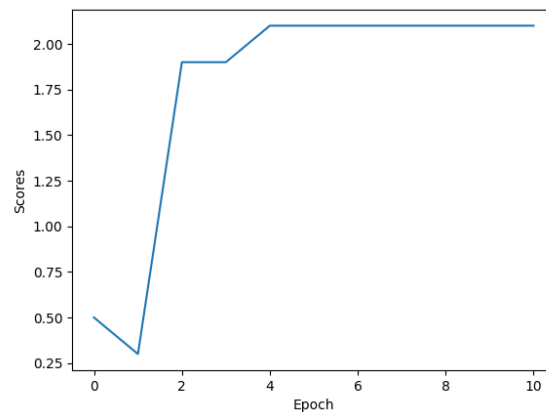


Figure 3: Reference plot for linear approximation with Tensorflow on the Test Env

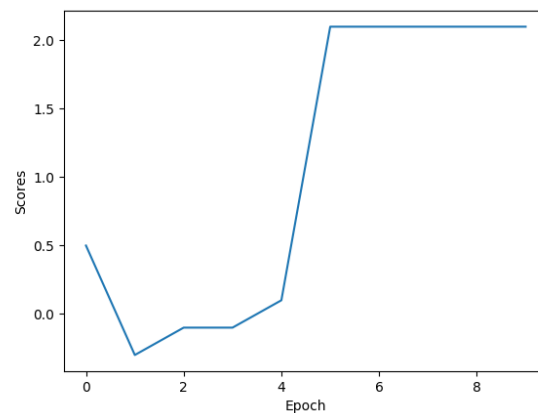


Figure 4: Reference plot for the nature network with Tensorflow on the Test Env

2. (**written** 5pts) Attach the plot of scores. Compare the model with linear approximation. What about the final performance? Training time?

Solution:

We manage to get the same performance. We also observe that the training time for the nature paper is much slower (61s against 16s). Also, we do 10 times less updates for the nature paper, which sums up to a ratio of $10 \times 4 = 40$!

This points out the necessity of finding the right complexity of model given a problem.

3. (**written** 5pts) What's the number of parameters of this model if the input to the Q network is a tensor of shape $(80, 80, 4)$? Compare with linear approximation.

Solution:

The number of parameters by layer is

- $8 \times 8 \times 4 \times 32 + 32$ for the first convolution
- $4 \times 4 \times 32 \times 64 + 64$ for the second convolution
- $3 \times 3 \times 64 \times 64 + 64$ for the third convolution

- $6,400 \times 512 + 512$ for the first fully connected layer (with SAME padding)
- $512 \times 6 + 6$ for the last layer (for Pong)

The total is

$$3,358,374$$

(notice that the convolution only accounts for 77,984 parameters!)

For linear approximation, the number of parameters is

$$25,600 * 6 + 6 = 153,606$$

Which is roughly 20 times less!

4. (**coding and written** 5pts). Now, we're ready to train on the Atari Pong-v0 environment. First, launch linear approximation on pong with `python q4_train_atari_linear.py`. What do you notice?

Solution:

After a few epochs, we notice that the score doesn't really progress, around -19.6 . Linear approximation is not powerful enough for Pong.

5. (**coding and written** 10 pts). In this question, we'll train the agent with DeepMind's architecture. Run `python q5_train_atari_nature.py`. We provide you with a default config file. Feel free to edit it. You can also decide to change the exploration strategy and learning rate schedule, that are both linearly decaying at this point. We expect you to run for at least 5 million steps (default in the config file). We provide you with our plot of performance (evaluation score) to help you debug your network.

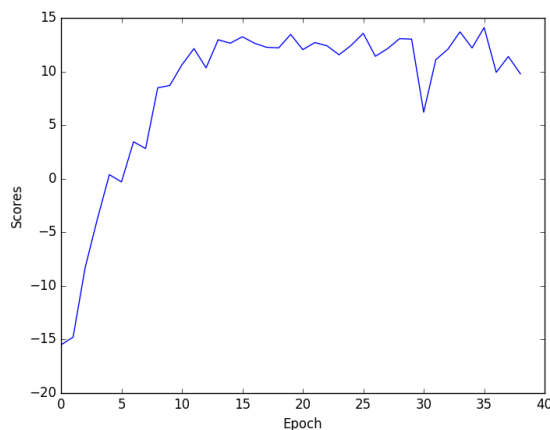


Figure 5: Evolution of the evaluation score on Pong-v0 for the first 5 million time steps. One epoch corresponds to 250k time steps.

Deliverable Include your own plot to the writeup. You should get a score of at least 5 after 5 million time steps (note that according to DeepMind's paper, human's performance is -3). Include your changes to the config to the writeup, if any.

As the training time is roughly **12 hours**, you may want to check after a few epochs that your network is making progress. A good way to check that is

- look at the progress of the evaluation score printed on terminal (it 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
- we advise you to use Tensorboard. The starter code writes summaries of a bunch of useful variables that can help you monitor the training process. More on how to use Tensorboard on the assignment page.

6. (**written** 5pts) Compare the performance with previous linear Q-network. How can you explain the gap in the performance?

Solution:

We reach around 15 with the nature network, compared to -19 with the linear approximation. This gap in performance is explained by the insufficient complexity of the linear approximation model, and the exceptional efficiency of the convolutional networks to analyse the image without having too much parameters.

6 Bonus: Architecture of the Network (10pts)

This is an open question. Design a different network architecture or try a more recent paper, like Double Q Learning [1] or Dueling Networks [4]. Explain your architecture and implement it in `q6_bonus_question.py`. You can build on the existing code architecture or start from scratch. Note that this is a **bonus** question.

7 Uniform Approximation (15pts)

Recall the universal approximation theorem that states a single-hidden-layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units. In this question, we explore the uniform approximation property of neural networks in the case of boolean functions. Assume the input x is a binary vector of length n , $x \in \{0, 1\}^n$. Let the activation function be the step function: $f(z) = 1$ if $z > 0$ and 0 otherwise. Note that the weights can still be real numbers. The output will be the result of a single unit and hence binary, either 0 or 1. We use such neural networks to implement boolean functions.

- (a) (**written** 3pts) Let $n = 2$ and the input be a pair of binary values. Consider a neural network with just a single output unit and no hidden units, i.e., $y = f(w^T x + b)$ where y is the output. Choose w and b to implement boolean AND (i.e., $y = 1$ only when $x = (1, 1)$). Choose w and b to implement boolean OR.

Solution:

We observe that the output $y = f(w^T x + b)$ is equivalently determined by the separating hyperplane corresponding to (w, b) . For boolean AND, we choose $w = (1, 1)$ and $b = -2 + \epsilon$ where $\epsilon \in (0, 1)$ is arbitrarily small. If $x = (1, 1)$, then $w^T x + b = \epsilon > 0$ and $y = 1$. Otherwise, $w^T x + b \leq 1 + b = -1 + \epsilon < 0$ and $y = 0$. For boolean OR, we choose $w = (1, 1)$ and $b = -1 + \epsilon$. Note that ϵ is there to handle the 0-1 boundary in the activation function.

- (b) (**written** 3pts) Under the same conditions as above, what boolean function of two variables cannot be represented? Briefly explain why the function cannot be implemented.

Solution:

Boolean XOR is one example that cannot be represented. Using the hyperplane interpretation, we see that there is no separating hyperplane that correctly identifies these ‘True’ and ‘False’ points: ‘False’ points at $(0, 0)$ and $(1, 1)$ and ‘True’ points at $(1, 0)$ and $(0, 1)$.

- (c) (**written** 3pts) Suppose we now allow a single layer of hidden units, i.e., $y = f(w^T z + b)$, $z_j = f(w_j^T x + b_j)$ where z_j ’s present components of z . Construct a neural network that represents the boolean function you provided in Part (b). There is no restrictions on the number of hidden units, but try to keep it as simple as possible.

Solution: To construct Boolean XOR, we use two units for the hidden layer (z_1, z_2) . Each of these units indicate whether one of the input variables is strictly greater than the other one. More specifically:

$$\begin{aligned}w_1 &= [1, -1]^T, b_1 = 0 \\w_2 &= [-1, 1]^T, b_2 = 0\end{aligned}$$

For the output unit, set:

$$w = [1, 1]^T, b = 0$$

This insures, the output y will be one only if either $z_1 = 1$ or $z_2 = 1$.

- (d) (**written** 3pts) Describe a general procedure to construct a neural network with single hidden layer to represent any boolean function.

Solution: Every boolean function can be presented with a conjunctive normal form (AND of ORs). Implement the clauses with the hidden layer. The output unit should implement an AND across all hidden units.

In the rest, we will show how AND and OR operations with more than two inputs can be constructed. We will also show how to find the negative of a literal. These should suffice to carry out the procedure mentioned above. The OR and negation operations will occur in the hidden layer and the AND operation will take place in the output unit.

Consider a unit with n inputs. Hence for this unit, $y = f(w^T x + b)$ and $x \in \{0, 1\}^n$. This unit can implement OR across its inputs using the following configuration:

$$w = [1, \dots, 1]^T, b = 0$$

In addition, this unit will implement AND across its inputs using the following configuration:

$$w = [1, \dots, 1]^T, b = n - 1$$

Further, the negative of an input x_i is $(1 - x_i)$. So, for each negation in a unit, set the coefficient of x_i to -1 and add the value 1 to the bias.

- (e) (**written 3pts**) The result from Part (d) implies that a single hidden layer is sufficient to implement any boolean function. Why would you want to consider neural networks with multiple hidden layers?

Solution:

Based on the procedure in part (d), with a single hidden layer, the number of required units in the hidden layer can grow exponentially with the number of inputs. So, it is not scalable. Sub-clauses can be created in earlier hidden layers to construct clauses in further layers. This can significantly decrease the number of required units.

References

- [1] Hado van Hasselt, Arthur Guez, and David Silver. “Deep Reinforcement Learning with Double Q-learning”. In: *CoRR* abs/1509.06461 (2015). URL: <http://arxiv.org/abs/1509.06461>.
- [2] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (2015), pp. 529–533.
- [3] Volodymyr Mnih et al. “Playing Atari With Deep Reinforcement Learning”. In: *NIPS Deep Learning Workshop*. 2013.
- [4] Ziyu Wang, Nando de Freitas, and Marc Lanctot. “Dueling Network Architectures for Deep Reinforcement Learning”. In: *CoRR* abs/1511.06581 (2015). URL: <http://arxiv.org/abs/1511.06581>.