10-703 Deep Reinforcement Learning and Control
Assignment 2: Model-Free Learning
Spring 2018

Jagjeet Singh (jagjeets), Satyen Rajpal (satyenr)

Febuary 21, 2018

Due March 7, 2018

Instructions

You have around 15 days from the release of the assignment until it is due. Refer to Gradescope for the exact time due. You may work in teams of **2** on this assignment. Only one person should submit the writeup and code on Gradescope. Additionally you should upload your code to Autolab, please make sure the same person who submitted the writeup and code to Gradescope is the one who submits it to Autolab. Make sure you mark your partner as a collaborator on Gradescope (You do not need to do this in Autolab) and that both names are listed in the writeup. Writeups should be typeset in Latex and submitted as PDF. All code, including auxiliary scripts used for testing should be submitted with a README.

Introduction

In Homework 1, you learned about and implemented various model-based techniques to solve MDPs. These "planning" techniques work well when we have access to a model of the environment; however getting access to such a model is often unfeasible. In this Homework, you will explore an alternative to the model-based regime, i.e. model-free learning. In particular, you will learn about Temporal Difference learning and its variants, and implement a version of TD learning called Q-learning. We will look at implementing Q learning with both tabular methods and deep function approximators.

You will work with the OpenAI Gym environments, and learn how to train a Q-network from state inputs on a gym environment. The goal is to understand and implement some of the techniques that were found to be important in practice to stabilize training and achieve better performance. We also expect you to get comfortable using Tensorflow or Keras to experiment with different architectures, and understand how to analyze your network's final performance and learning behavior.

Please write your code in the file `DQN_Implementation.py`, the template code provided inside is just there to give you an idea on how you can structure your code but is not mandatory to use.

Background in Q-learning and Variants

Function approximators have proved to be very useful in the reinforcement learning setting. Typically, one represents Q-functions using a class of parametrized function approximators $\mathcal{Q} = \{Q_w \mid w \in \mathbb{R}^p\}$, where p is the number of parameters. Remember that in the *tabular setting*, given a 4-tuple of sampled experience (s, a, r, s'), the vanilla Q-learning update is

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

where $\alpha \in \mathbb{R}$ is the learning rate. In the *function approximation setting*, the update is similar:

$$w := w + \alpha \left(r + \gamma \max_{a' \in A} Q_w(s', a') - Q_w(s,a) \right) \nabla_w Q_w(s,a). \tag{2}$$

Q-learning can be seem as a pseudo stochastic gradient descent step on

$$\ell(w) = \mathbb{E}_{s,a,r,s'} \left(r + \gamma \max_{a' \in A} Q_w(s', a') - Q_w(s,a) \right)^2,$$

where the dependency of $\max_{a' \in A} Q_w(s', a')$ on w is ignored, i.e., it is treated as a fixed target. Many of the methods that you will implement in this homework are variants of update (2). We recommend reading through the *deep Q-learning implementation* described in [**?**, **?**].

In this homework, we will also ask you to implement the *dueling deep Q-network* described in [**?**]. This amounts to a slightly different Q-network architecture from the one in [**?**, **?**]. Most models will be trained using *experience replay* [**?**, **?**], meaning that the 4-tuples (s, a, r, s') will be sampled from the replay buffer rather than coming directly from the online experience of the agent.

1 Theory Questions

Question 1.a [5pts]

Consider a sequence of streaming data points $\mathbf{X} : \{x_1, x_2, x_3, ...x_N\}$. At any given time instant k, one has access to data points $\{x_1, x_2, ...x_k\}$; but not $\{x_{k+1}, ...x_N\}$. Prove that the mean μ_k of the available data may be computed incrementally (online) at every time step k, in the form: $\mu_k = \mu_{k-1} + f(\mu_{k-1}, x_k)$.

Solution

Suppose at time instant k, we have access to only $\{x_1, x_2, ...x_k\}$ data points. The mean of the available data

$$\begin{aligned}
\mu_k &= \frac{\sum_{i=1}^{k} x_i}{k} \\
&= \frac{(x_k + \sum_{i=1}^{k-1} x_i)}{k} \\
&= \frac{(x_k + \mu_{k-1}(k-1))}{k} \\
&= \frac{(x_k + k\mu_{k-1} - \mu_{k-1})}{k} \\
\mu_k &= \mu_{k-1} + \frac{(x_k - \mu_{k-1})}{k}
\end{aligned}$$

This is of the form $\mu_k = \mu_{k-1} + f(\mu_{k-1}, x_k)$ which essentially implies that the mean of the next time instant can be computed from the previous time instant or, in other words, incrementally.

Question 1.b [5pts]

The function $f(x_k, \mu_{k-1})$ represents the error between the current estimate of the mean, μ_{k-1}, and the true mean at time k, μ_k. Consider the following scenario - an agent follows some policy in its environment, and receives a return G_t from an episode starting at state s_t. We would like to update the value of state s_t, $V(s_t)$, using the newly obtained G_t.

- Treating the update of $V(s_t)$ as an online learning problem, where G_t is the newly received data point, show that the current estimate of the value of the state $V(s_t)$ can be updated in a form analogous to Question 1.a. You may assume that the state s_t has been visited $N(s_t)$ number of times.
- What is the error function f? What does the update you derived resemble?

Solution

Assume that G_t is the newly received return by the agent by following some policy π starting at state s_t. The value function for the state at the next time step (mean return)-

$$V(s_{t+1}) = S(s_{t+1})/N(s) \dots (1)$$

where $S(s_t)$ is the total return until time t and $N(s)$ is the number of times state s has been visited. Therefore, the total reward till time $(t+1)$ is $S(s_{t+1}) = S(s_t) + G_t$, which implies

$$V(s_{t+1}) = \frac{S(s_t) + G_t)}{N(s)}$$
$$= \frac{G_t + (N(s) - 1)V(s_t)}{N(s)}$$
$$V(s_{t+1}) = V(s_t) + \frac{G_t - V(s_t)}{N(s)}$$

This equation is analogous to the form $\mu_k = \mu_{k-1} + f(\mu_{k-1}, x_k)$, in question 1.
The error function f is the error between the expected received reward and value function of the state. This equation resembles every visit Monte Carlo learning.

Question 1.c [5pts]

The update you derived in Question 1.b is based on the return from the entire episode, G_t. However, this method only learns from the end of the episode. Instead, it is possible to accelerate the learning process by bootstrapping.

1. Consider the agent is in state s_t, takes an action a_t, and transitions to state s_{t+1}, receiving reward r. Write out the estimate of the return of this episode based on this reward r and the current estimate of the value function V.

2. In the update from Question 1.b, replace the return G_t with the above estimate of return, and write out the new update for the value function $V(s_t)$. What does this update resemble?

Solution

Considering the agent in state s_t, by taking action a_t transitions into state s_{t+1} and receives a reward r. The estimate of the return of this episode based on this reward r is $r + \gamma V(s_{t+1})$, where γ is discounting factor.
Replacing the return G_t with an estimate, the new update for the value function $V(s_t)$, we get -

$$V(s_{t+1}) = V(s_t) + \frac{1}{N(s)}(r + \gamma V(s_{t+1}) - V(s))$$

This resembles TD(0) learning where $\frac{1}{N(s)}$ is analogous to learning rate.

Question 2.a [5pts]

Update 6 defines the TD update when the representation of the states and actions are in tabular form. Show that for a certain construction of the feature function ϕ, the tabular representation representation is a special case of the TD update 2 where the function in $\mathcal{Q}$ is of the form $Q_w(s, a) = w^T \phi(s, a)$. Give detailed description about your feature function

and your proof for equivalence to earn full credits.

Solution

Feature function:
The feature function can be represented as shown below. It is a column vector of size $|S| * |A|$. Each value of the vector can be either 1 or 0, depending on which state and action pair is the input. As such, it will eventually be a one-hot vector.

$$\phi(s,a) = \begin{bmatrix} 1(s=s_1,a=a_1) \\ 1(s=s_1,a=a_2) \\ . \\ . \\ 1(s=s_1,a=a_{|A|}) \\ 1(s=s_2,a=a_1) \\ 1(s=s_2,a=a_2) \\ . \\ . \\ 1(s=s_2,a=a_{|A|}) \\ .... \\ .... \\ 1(s=s_{|S|},a=a_1) \\ 1(s=s_{|S|},a=a_2) \\ . \\ . \\ 1(s=s_{|S|},a=a_{|A|}) \end{bmatrix}$$

Proof of equivalence:
$Q_w(s,a) = w^T\phi(s,a)$
Since $\phi(s,a)$ is a one-hot vector, there will be one weight element corresponding to each (state,action) pair. Considering a sample input (s_i, a_j), let's say w_{ij} is the weight corresponding to it. Then $Q_w(s,a) = w^{ij}$.
Gradient of Q becomes:

$$\Delta_w Q_w(s,a) = \begin{bmatrix} 0 & 0 & .... & 1 & .... & 0 & 0 \end{bmatrix}$$

where the element corresponding to differential wrt w_{ij} is 1.
Now using this in equation 2,

$$\begin{bmatrix} w_1 \\ w_2 \\ .. \\ w_{ij} \\ w_{|S|,|A|} \end{bmatrix} := \begin{bmatrix} w_1 \\ w_2 \\ .. \\ w_{ij} \\ w_{|S|,|A|} \end{bmatrix} + \alpha(r + \gamma max Q(s'_i, a'_j) - Q(s_i, a_j)) \begin{bmatrix} 0 \\ 0 \\ .... \\ 1 \\ .... \\ 0 \\ 0 \end{bmatrix}$$

This is equivalent to Tabular representation because for each state-action pair, there is just 1 weight. Therefore, similar to a table lookup, we can pick the corresponding weight and that will give us the Q-value for that pair. The above equation can be simplified to resemble the table look up update Equation 1 as follows:
From the above equation, it is clear that in each iteration, there is only 1 component of weight vector which is getting updated. Therefore,

$$w_{ij} := w_{ij} + \alpha \left(r + \gamma \max_{a' \in A} Q_{w^-}(s'_i, a'_j) - Q_w(s_i, a_j) \right) .1 \tag{3}$$

Plugging this in $Q_w(s, a) = w^T \phi(s, a)$, the equation simplifies to:

$$Q(s, a) := Q(s, a) + \alpha \left(r + \gamma \max_{a' \in A} Q(s', a') - Q(s, a) \right), \tag{4}$$

Therefore, we can conclude that Q-values can be found in similar lookup approach as done in tabular representation and update equation also boils down to the similar form. Hence, the tabular representation is a special case of TD update.

Question 2.b [5pts]

State aggregation improves generalization by grouping states together, with one table entry used for each group. Let $\mathcal{X} : s \to x$ be the grouping function that maps a input state s to its group x and usually, the number of groups $|X| \ll |S|$. During training, whenever a state in a group is encountered, the groups' entry and the action a is used to determine the state's value, and when the state is updated, the group's entry is updated. Assume $\mathcal{Q}$ is still in the the form $Q_w(s, a) = w^T \phi(s, a)$. Show that a tabular representation with state aggregations is also a special case of TD update 2 under certain construction of feature function ϕ. Give detailed description about your feature function and your proof for equivalence to earn full credits.

Solution

Feature function:
The feature function can be represented as shown below. It is a column vector of size $|X| * |A|$. Each value of the vector can be either 1 or 0, depending on which group to which the state belongs and action taken at that state. As such, it will eventually be a one-hot vector.

$$\phi(s,a) = \begin{bmatrix} 1(s \in x_1, a = a_1) \\ 1(s \in x_1, a = a_2) \\ . \\ . \\ 1(s \in x_1, a = a_{|A|}) \\ 1(s \in x_2, a = a_1) \\ 1(s \in x_2, a = a_2) \\ . \\ . \\ 1(s \in x_2, a = a_{|A|}) \\ .... \\ .... \\ 1(s \in x_{|X|}, a = a_1) \\ 1(s \in x_{|X|}, a = a_2) \\ . \\ . \\ 1(s \in x_{|X|}, a = a_{|A|}) \end{bmatrix}$$

Proof of equivalence:
$Q_w(s,a) = w^T \phi(s,a)$
Assuming each state can belong to just 1 group, $\phi(s,a)$ will be a one-hot vector. As such, there will be exactly one weight element corresponding to each (state,action) pair. Considering a sample input (s_i, a_j), let's say w_{ij} is the weight corresponding to it. Then $Q_w(s,a) = w^{ij}$.
Gradient of Q becomes:

$$\Delta_w Q_w(\mathcal{X}(s), a) = \begin{bmatrix} 0 & 0 & .... & 1 & .... & 0 & 0 \end{bmatrix}$$

where the element corresponding to differential wrt w_{ij} (i.e., component corresponding to group x_i and action a_j) is 1.
Now using this in equation 2,

$$\begin{bmatrix} w_1 \\ w_2 \\ .. \\ w_{ij} \\ w_{|S|,|A|} \end{bmatrix} := \begin{bmatrix} w_1 \\ w_2 \\ .. \\ w_{ij} \\ w_{|S|,|A|} \end{bmatrix} + \alpha(r + \gamma max Q(\mathcal{X}(s'_i), a'_j) - Q(\mathcal{X}(s_i), a_j)) \begin{bmatrix} 0 \\ 0 \\ .... \\ 1 \\ .... \\ 0 \\ 0 \end{bmatrix}$$

This is equivalent to Tabular representation because for each state-action pair, there is just 1 weight. Therefore, similar to a table lookup, we can pick the corresponding weight and that will give us the Q-value for that pair. The above equation can be simplified to resemble the table look up update Equation 1 as follows:
From the above equation, it is clear that in each iteration, there is only 1 component

of weight vector which is getting updated. This means the weights corresponding to any group will get updated everytime a state belonging to that group is given as an input. Therefore,

$$w_{ij} := w_{ij} + \alpha \left(r + \gamma \max_{a' \in A} Q_{w^-} \mathcal{X}(s'_i), a'_j) - Q_w(\mathcal{X}(s_i), a_j) \right) .1 \tag{5}$$

Combining this with $Q_w(s, a) = w^T \phi(s, a)$, the equation simplifies to:

$$Q(\mathcal{X}(s), a) := Q(\mathcal{X}(s), a) + \alpha \left(r + \gamma \max_{a' \in A} Q(\mathcal{X}(s'), a') - Q(\mathcal{X}(s), a) \right), \tag{6}$$

Therefore, we can conclude that Q-values for state aggregation can be found in similar lookup approach as done in tabular representation and update equation also boils down to the similar form. Hence, the tabular representation with state aggregations is also a special case of TD update 2 under certain construction of feature function ϕ

2 Programming Questions

Before starting your implementation, make sure you have the environments correctly installed. For this assignment you will solve Cartpole (`Cartpole-v0`) and Mountain Car-v0 (`MountainCar-v0`). For extra-credit [**+10pts**], you will need to additionally solve the more challenging Space Invaders (`SpaceInvaders-v0`) environment. For all of the following implementations we would like you to document your findings in the writeup as described after the questions.

Try to keep your implementation modular. Once you have the basic DQN working it will only be a little bit of work to get DQN and dueling networks working. Please write your code in the file `DQN_implementation.py`, the template code provided inside is just there to give you an idea on how you can structure your code but is not mandatory to use.

1. **[20pts]** Implement a linear Q-network (no experience replay, i.e. experience is sampled directly from the environment online). Train your network (separately) on both the `CartPole-v0` environment and the `MountainCar-v0` environment. Use the full state of each environment (such as angular displacement and velocity in the case of Cartpole) as inputs.

   In case of the simple environments, you may observe good performance even with a linear Q-network. If you can get this running you will have understood the basics of the assignment.

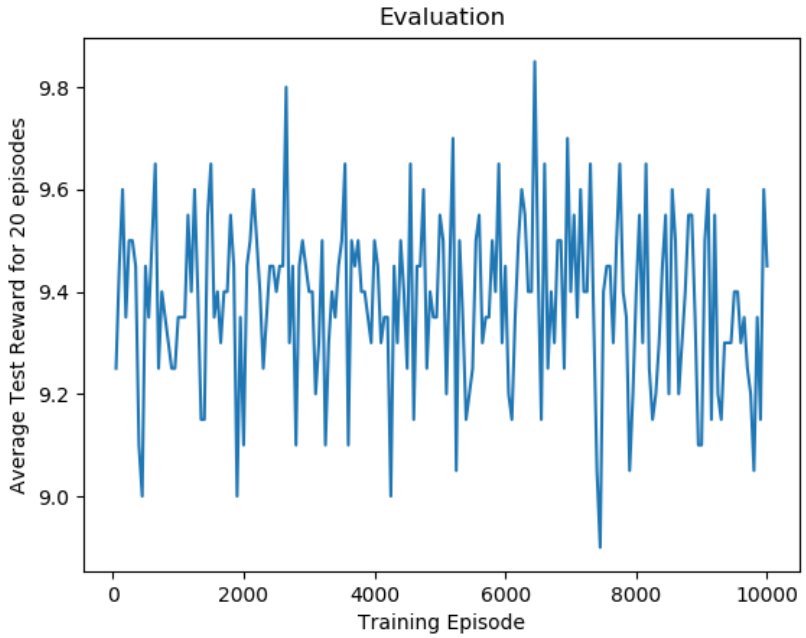
   For this question, look at the `QNetwork` and `DQN_Agent` classes in the code. You will have to implement the following:

   - Create an instance of the Q Network class.

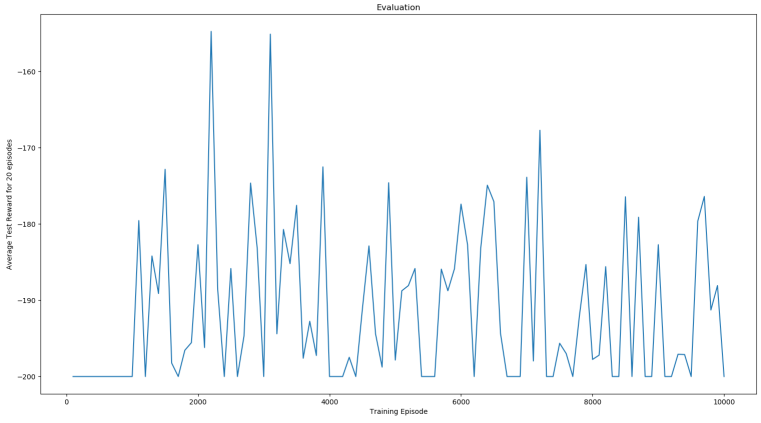
- Create a function that constructs a policy from the Q values predicted by the Q Network - both epsilon greedy and greedy policies.
- Create a function to train the Q Network, by interacting with the environment.
- Create a function to test the Q Network's performance on the environment.

Are you able to solve both environments with a linear network? Why or why not?

Solution



Performance plot for CartPole in Linear Q-network with NO experience replay



Performance plot for MountainCar in Linear Q-network with NO experience replay

Inference:

For Mountain Car problem, the car was able to reach the flag very frequently. If we define 'solved' as reward¿-110 consistently, then we were not able to solve. If we define 'solved' as reaching the flag consistently, then we were able to solve it. For CartPole problem, we were not able to solve it using this model.
Theoritically, both MountainCar-v0 and CartPole-v0 are solvable by Linear Q-network without experience replay.
Our observation was that in case of CartPole, there can be a large difference in Q-values even for nearby states. As such, it's more difficult in CartPole than MountainCar for a linear function to predict the values.
Hyperparameters used: learning rate = 0.001, discount = 0.99 (CP) and 1 (MC), $\epsilon_{start} = 0.9$, batchsize=32

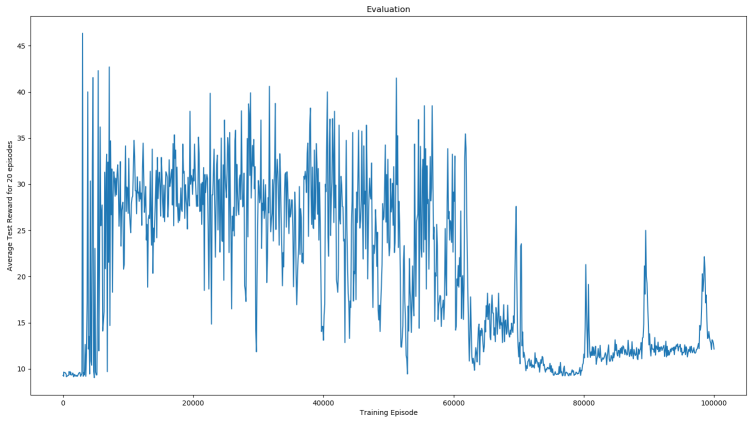
2. **[15pts]** Implement a linear Q-network with experience replay. Use the experimental setup of [**?**, **?**] to the extent possible. Use this as an opportunity to work out any bugs with your replay memory.

   You can refer to the `Replay_Memory` class in the code. You may need the following functions:

   - Appending a new transition from the memory.
   - Sampling a batch of transitions from the memory to train your network.
   - Initially burn in a number of transitions into the memory.
   - You will also need to modify your training function of your network to learn from experience sampled *from the memory*, rather than learning online from the agent.

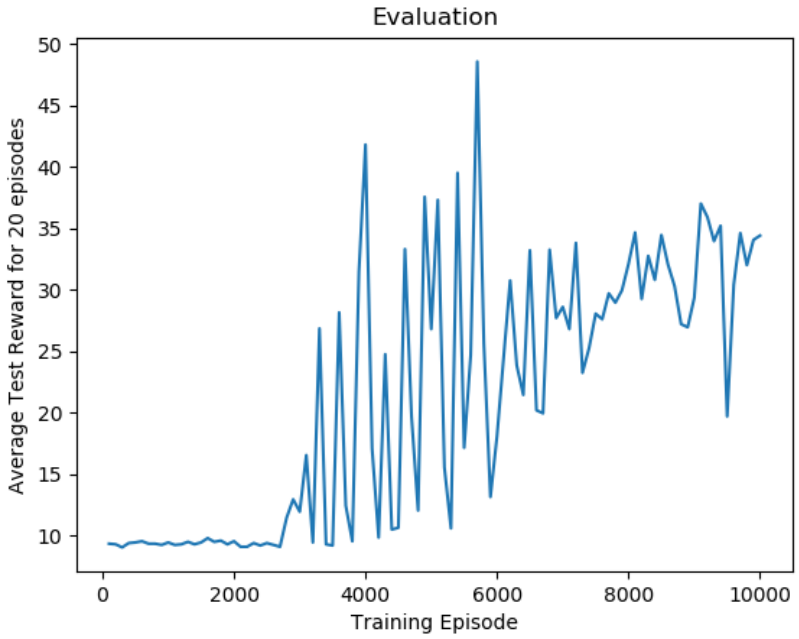
   As above, train your network (separately) on both the `CartPole-v0` environment and the `MountainCar-v0` environment. Does adding experience replay improve learning?

Solution

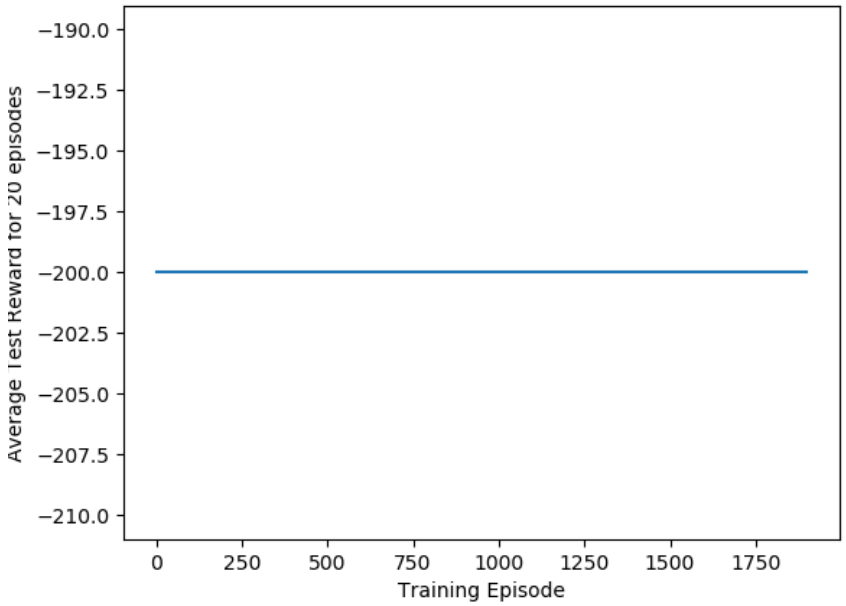


Performance plot for CartPole in Linear Q-network with experience replay for

100000 episodes



Performance plot for CartPole in Linear Q-network with experience replay for 10000 episodes



Performance plot for MountainCar in Linear Q-network with experience replay

Inference:

Yes, experience replay improves learning. This is because just like any

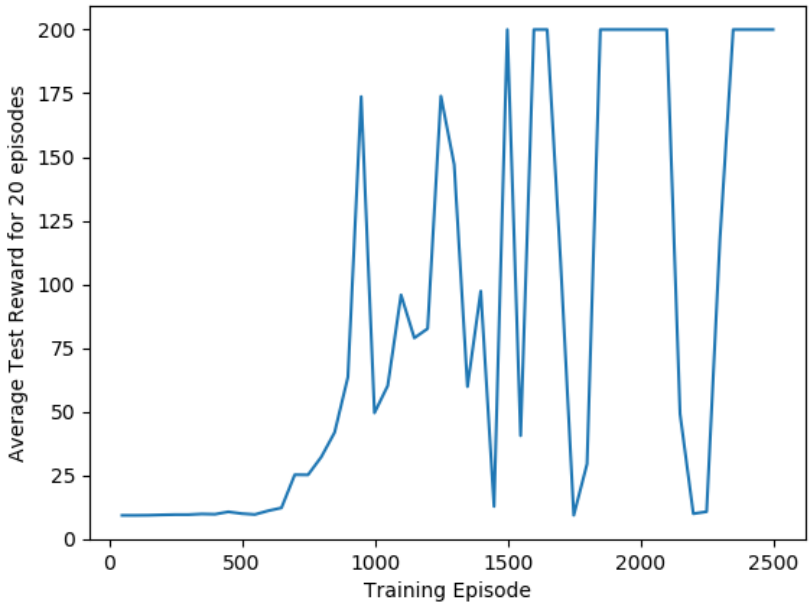
supervised learning algorithm, random sampling of experiences breaks the correlations among successive states encountered in online learning. This allows the model to generalize the value function. In addition, experience replay prevents unwanted feedback loops where the parameters could get stuck in a poor local minimum, or even diverge catastrophically. Consider a case where a robot is navigating itself in a room to reach a goal. if the maximizing action is to move left then the training samples will be dominated by samples from the left-hand side; if the maximizing action then switches to the right then the training distribution will also switch. When learning on-policy the current parameters determine the next data sample that the parameters are trained on as described by the example. Experience replay is one such way to counter this. However, in our implementation we were NOT able to solve both MountainCar-v0 and CartPole-v0. It is attributed to the randomness introduced by the concept of sampling.
Why we couldn't solve CartPole? - In practice, CartPole is difficult to be learnt by a linear function because the values vary a lot from state to state. If we had more time, we would have tried some tweaks like reward shaping and prioritized replay. After a lot of hyperparameter tuning and running it for 100000 episodes, we got the performance as showed in the first chart. We noticed that the performance (although very noisy) increases till almost 10000 episodes and remains the same thereon. So we did a second round or tuning and tweaking the params and tried everything for 10000 episoded now. The final performance is reported in the second chart.
Why we couldn't solve MountainCar? - The performance degraded compared to the case without replay because the concept of sampling from a memory made the rewards even more sparse. There are very few occasions when the car reaches the flag and when it does reach, the transition is appended to the memory. Now in the next update, there is very less probability that the same transition containing a non-negative reward which was already very sparse will be sampled. As such, we saw a decrease in performance compared to the previous question. Using prioritized replay could have avoided this issue.
Hyperparameters used: learning rate = 0.0001, discount = 0.99 (CartPole) and 1 (MountainCar), $\epsilon_{start} = 0.75$, batch size=32, memory size=50000, burn in=10000

3. **[20pts]** Implement a deep Q-network, similar to that described in [**?**, **?**]. While these papers use a convolutional architecture to address the image based state representation of Atari environments, a multi-layer Perceptron with 3 hidden units should suffice for the environments.

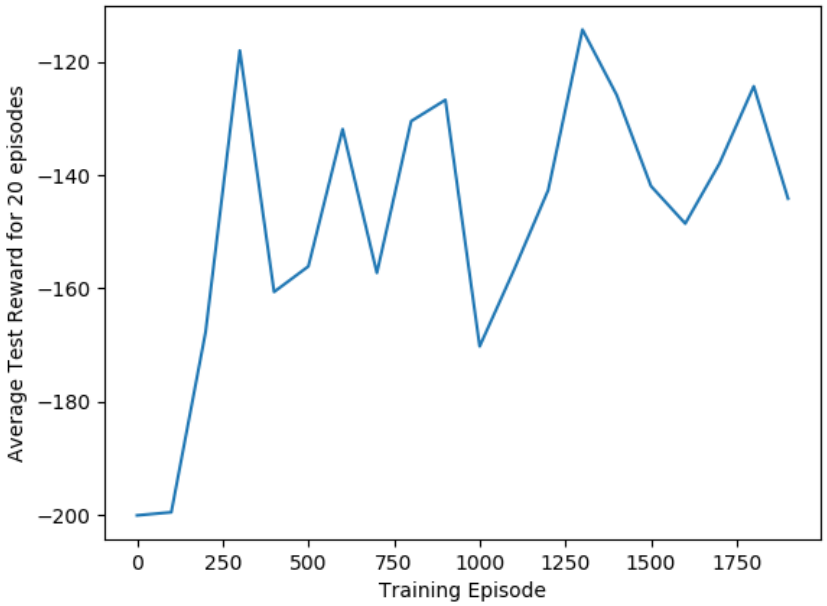
   Document any changes to your training procedure, optimization, etc. that you make to train your deep Q-network, as compared to the linear case. How does using this deep Q-network affect performance, convergence rate, and stability of training on both `Cartpole-v0` and `MountainCar-v0`?

You may use the cluster to train this agent *once you have worked out the bugs*, although these environments are simple enough that you may solve them on your own laptop in a few hours.

Solution



Performance plot for CartPole in Deep Q-network



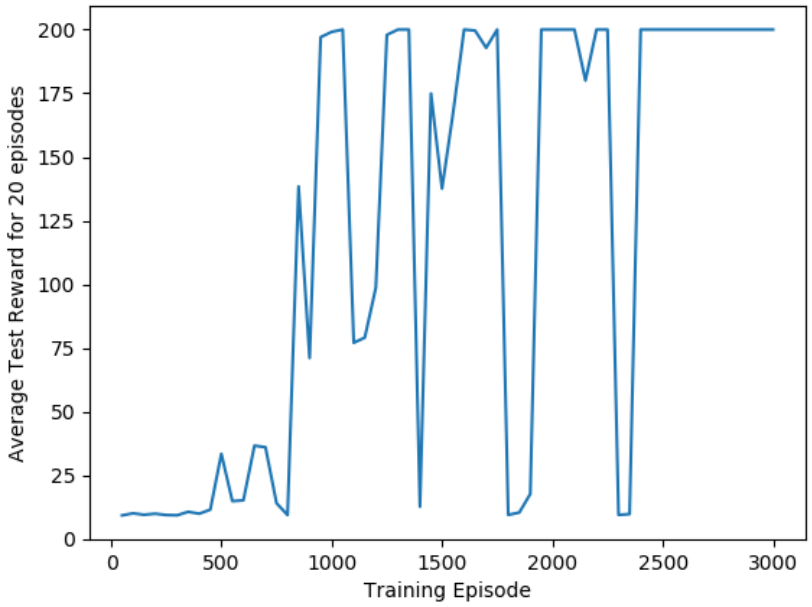
Performance plot for MountainCar in Deep Q-network

Inference:

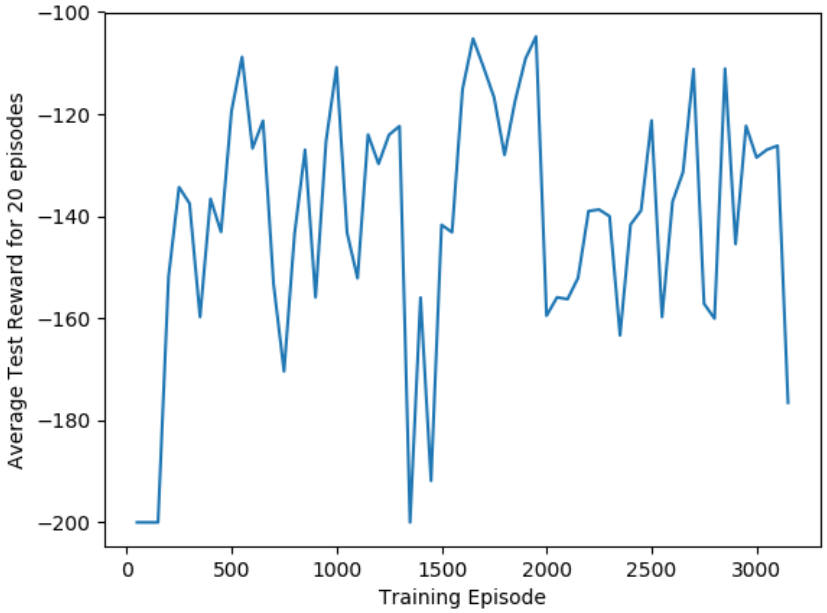
We used a 2 layered Perceptron network each with 24 hidden neurons for training both environments and were successfully solved. In comparison to the linear case, the Mountain Car environment achieved stability much before and a solution to the Cart Pole environment was obtained. Here stability refers to the case where the car is able to reach the flag or obtain a reward > -200 while testing. A deep Q network is able to generalize the non-linear Q function much better than a linear network because of non linearity layers (ReLU) and network depth.
When testing the model, the Mountain Car environment after 250 episodes consistently returned an average reward greater than -200 implying that the car was able to reach the flag. As for the CartPole environment, an almost consistent return of 200 average reward started around 1500 training episodes. Hyper-parameters used: learning rate = 0.001, discount = 0.99 (Cart Pole) and 1 (Mountain Car), $\epsilon_{start} = 0.75$,$\epsilon_{decay} = 0.99995$, batch size=32, memory size=20000, burn in=1000

4. **[15pts]** Implement the dueling deep Q-network as described in [**?**]. Train your network (separately) on both the `CartPole-v0` environment and the `MountainCar-v0` environment. How does adding the two stream architecture affect performance?

Solution



Performance plot for CartPole in Deep Q-network with Duelling



Performance plot for MountainCar in Deep Q-network with Duelling

Inference: Adding the two stream architecture significantly improved the stability and convergence rate for both environments when compared to DQN. The CartPole environment achieved an almost consistent reward of 200 after just 1000 episodes (about 7-8 min) and for the MountainCar environment the same began at about 350 episodes. With every update of Q values, the dueling network updates the value stream whereas in the standard single steam architecture, only one of the actions are updated. This allows for better approximation of the state values and in turn increases the convergence rate of the model.
Why duelling is better? -

(a) Consider the Mountain Car problem and assume that in a particular episode the car has reached the flag for the first time. Now, for every state-action (s, a) pair encountered in this episode, the Q value is updated for all the possible actions in each state encountered in this episode. The action a would be given a higher Q value over the other possible actions in state s. In contrast, the DQN algorithm only updates a. Thus, all the possible action values for each state are updated in one gradient descent step making the convergence rate faster.

(b) Furthermore, for a given state, the difference in Q-values for the best action v/s the next best action is very small compared to the average difference in Q across states. For example, in the case of CartPole, the effect of taking to different actions at a particular state on Q-value is very small compared to the change in Q value when a state is changed. As

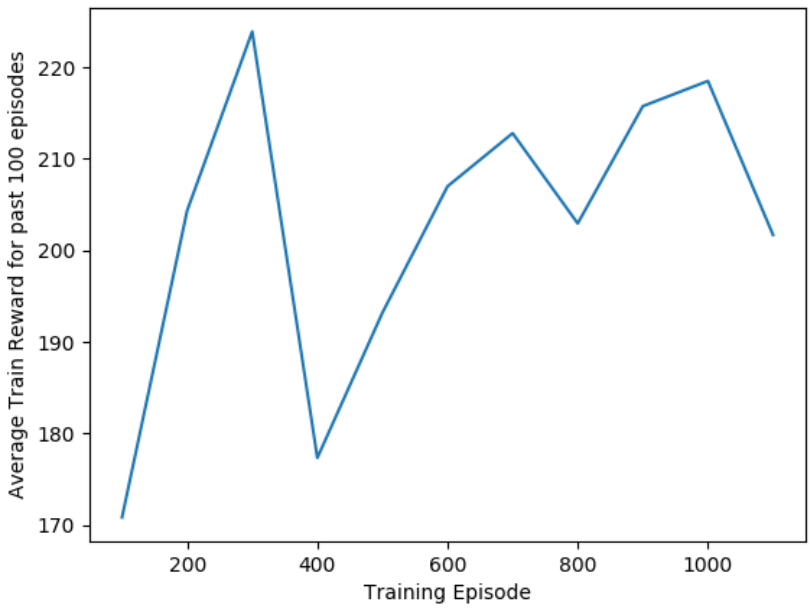
such, conventional DQN will inherit some noise and occasionally lose track of which action is the best action at any given state. Duelling helps avoid such a situation by treating value of state and state independent actions separately.

Hyper-parameters used: learning rate = 0.0005, discount = 0.99 (Cart Pole) and 1 (Mountain Car), $\epsilon_{start} = 0.7$,$\epsilon_{decay} = 0.99995$, batch size=32, memory size=35000, burn in=5000

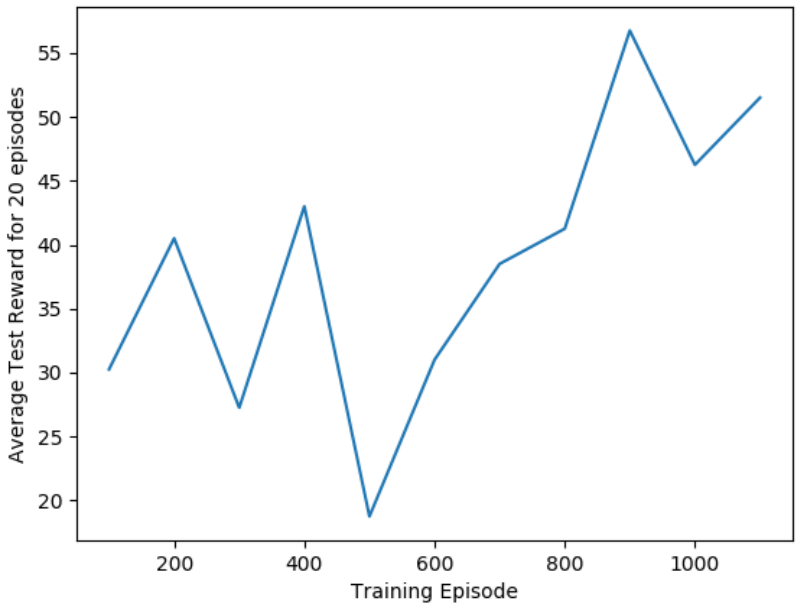
5. **Extra Credit: [+10 pts]** Implement a deep Q-network (dueling deep Q-network not required) using a convolutional architecture to solve the `SpaceInvaders-v0` environment. Train your network by passing in 4 consecutive frames from the environment, as described in [**?**, **?**].

   Modify your replay memory to store images, preprocess them as necessary (resizing / normalization, etc.), and sample frames to provide to the network for training. You may find it necessary to implement a Target Q Network (as in the double Q learning paper [**?**]) to stabilize training of the networks. Document these changes in your report.

Solution



Performance plot for Atari showing training rewards



Performance plot for Atari showing test rewards

Inference:

(a) Preprocessing done:

- RGB to GrayScale()
- Resizing to (118,84)
- Center Crop to (84,84)
- concatenating 4 frames (eg. f1,f2,f3,f4) and treat them as a single state for Q-network input
- taking next 4 frames (eg. f2,f3,f4,f5) as the next state

(b) Added a target Q-network to improve performance

(c) Hyperparameters used as suggested in the paper: lr = 0.0001, $\epsilon_{start} = 0.9$, $\epsilon_{end} = 0.1$, optim=RMSProp

(d) Because of limited computing power, we could not run it for as many updates as shown in the paper. We were able to run it for 100000 updates

(e) To deal with high memory requirements, we converted the image into ByteTensor before storing them in the memory and converted it back to FloatTensor after sampling.

(f) Also, we used a batch size of 8, unlike the paper, to deal with high memory requirements

5pts out of the total of **100pts** are reserved for overall report quality.

For each of the above questions, we want you to generate a *performance plot across time*. To do this, you should periodically run (e.g., every 10000 or 100000 updates to the Q-network) the policy induced by the current Q-network for 20 episodes and average the total reward achieved. Note that in this case we are interested in total reward without discounting or truncation. During evaluation, you should use a tinier ϵ value in your ϵ-greedy policy. We recommend 0.05. The small amount of randomness is to prevent the agent from getting stuck. Also briefly comment on the training behavior and whether you find something unexpected in the results obtained.

Additionally, for each of the models, we want you to generate a *video capture* of an episode played by your trained Q-network at different points of the training process (0/3, 1/3, 2/3, and 3/3 through the training process) of both environments, i.e. `Cartpole-v0` and `MountainCar-v0`.

An episode is defined as 200 successful time steps in Cartpole, and reaching the top of the mountain in Mountain Car. You can use the `Monitor` wrapper to generate both the performance curves (although only for ϵ-greedy in this case) and the video captures. Look at the OpenAI Gym tutorial for more details on how to use it. We recommend that you periodically checkpoint your network files and then reload them after training to generate the evaluation curves. It is recommended you do regular checkpointing anyways in order to ensure no work is lost if your program crashes.

Finally, construct a *table* with the average total reward per episode in 100 episodes achieved by your fully trained model. Also show the information about the standard deviation, i.e., each entry should have the format mean $\pm$ std. There should be an entry per model. Briefly comment on the results of this table.

We recommend you to follow closely the hyperparameter setup described below, or as in the references for training on the image based environment. Even if you fully replicate the experimental from the paper, we expect you to summarize it briefly in the report once. After that you can simply describe differences from it and mention that you used the same experimental setup otherwise if that was the case.

You should submit your report, video captures, and code through Gradescope. Your code should be reasonably well-commented in key places of your implementation. Make sure your code also has a README file.

Solution

Algorithm	Environment	Avg Reward	Std
Linear Q-network with NO Replay	CartPole-v0	9.32	0.75
Linear Q-network with NO Replay	MountainCar-v0	-183.8	28.8
Linear Q-network with Replay	CartPole-v0	24.29	10.16
Linear Q-network with Replay	MountainCar-v0	-200	0
Deep Q-network with NO Replay	CartPole-v0	200	0
Deep Q-network with NO Replay	MountainCar-v0	-142.01	21.3539
Duelling DQN with NO Replay	CartPole-v0	200	0
Duelling DQN with NO Replay	MountainCar-v0	-138.85	24.53

Inference:

1. Linear Q-Network was unable to solve CartPole with mean Avg Reward of 9.32 and 24.39. It's performance on MountainCar was also unsatisfactory with Avg Reward of -183.8 and -200 (this means it didn't even reach the flag once).
2. DQN has greate performane improvement over Linear Q-network
3. Theoretically, Duelling should have a significant improvement over DQN but because the environments were simple, the difference is small.

Guidelines on references

We recommend you to read all the papers mentioned in the references. There is a significant overlap between different papers, so in reality you should only need certain sections to implement what we ask of you. We provide pointers for relevant sections for this assignment for your convenience

The work in [**?**] contains the description of the experimental setup. Read paragraph 3 of section 4 for a description of the replay memory; read Algorithm 1; read paragraphs 1 and 3 of section 4.1 for preprocessing and model architecture respectively; read section 5 for the rest of the experimental setup (e.g., reward truncation, optimization algorithm, exploration schedule, and other hyperparameters). The methods section in [**?**], may clarify a few details so it may be worth to read selectively if questions remain after reading [**?**].

In [**?**], look at equation 11 and read around three paragraphs up and down for how to set up the dueling architecture; read paragraph 2 of section 4.2 for comments on the experimental setup and model architecture. It may be worth to skim additional sections of all these papers.

Guidelines on hyperparameters

In this assignment you will implement improvements to the simple update Q-learning formula that make learning more stable and the trained model more performant. We briefly comment on the meaning of each hyperparameter and some reasonable values for them.

- Discount factor γ: 1. for MountainCar, and 0.99 for CartPole and Space Invaders.
- Learning rate α: 0.0001; typically a schedule is used where learning rates get increasingly smaller.
- Exploration probability ϵ in ϵ-greedy: While training, we suggest you start from a high epsilon value, and anneal this epsilon to a small value (0.05 or 0.1) during training. We have found decaying epsilon linearly from 0.5 to 0.05 over 100000 iterations works well. During test time, you may use a greedy policy, or an epsilon greedy policy with small epsilon (0.05).

- Number of training sampled interactions with the environment: For MountainCar-v0, we recommend training for 3000 to 5000 episodes, though you should see improvements much before this. For CartPole-v0 , train for 1000000 iterations.

  Look at the average reward achieved in the last few episodes to test if performance has plateaued; it is usually a good idea to consider reducing the learning rate or the exploration probability if performance plateaus.

- Replay buffer size: 50000; this hyperparameter is used only for experience replay. It determines how many of the last transitions experienced you will keep in the replay buffer before you start rewriting this experience with more recent transitions. **(For Extra Credit:)** Use a replay buffer size of 1000000.

- Batch size: 32; typically, rather doing the update as in (2), we use a small batch ofsampled experiences from the replay buffer; this provides better hardware utilization.

- **(For Extra Credit:)** Number of frames to feed to the Q-network: 4; as a single frame may not be a good representation of the current state. Multiple frames are fed to the Q-network to compute the Q-values; note that in this case, the state effectively is a list of last few frames.

- **(For Extra Credit:)** Input image resizing: Using the original input size of 210x160x3 is computationally expensive. Instead, you may resize the image input to 84x84x3 to make it more manageable. Further, you may convert the RGB image to a Grayscale image.

In addition to the hyperparameters you also have the choice of which optimizer and which loss function to use. We recommend you use Adam as the optimizer. It will automatically adjust the learning rate based on the statistics of the gradients its observing. Think of it like a fancier SGD with momentum. Both Tensorflow and Keras provide versions of Adam.

For the loss function, you can use Mean Squared Error - if you are doing so, **please demonstrate** how the vanilla Q-learning update is equivalent to minimizing a quadratic loss function.

The implementations of the methods in this homework have multiple hyperparameters. These hyperparameters (and others) are part of the experimental setup described in [**?**, **?**]. For the most part, we strongly suggest you to follow the experimental setup described in each of the papers. [**?**, **?**] was published first; your choice of hyperparameters and the experimental setup should follow closely their setup. [**?**, **?**] follow for the most part the setup of [**?**, **?**]. We recommend you to read all these papers. We give pointers for the most relevant portions for you to read in a future section.

Guidelines on implementation

This homework requires a significant implementation effort. It is hard to read through the papers once and know immediately what you will need to be implement. We suggest you to think about the different components (e.g., replay buffer, Tensorflow or Keras model

definition, model updater, model runner, exploration schedule, learning rate schedule, ...) that you will need to implement for each of the different methods that we ask you about, and then read through the papers having these components in mind. By this we mean that you should try to divide and implement small components with well-defined functionalities rather than try to implement everything at once. Much of the code and experimental setup is shared between the different methods so identifying well-defined reusable components will save you trouble. We provide some code templates that you can use if you wish. Contrary to the previous assignment, abiding to the function signatures defined in these templates is not mandatory you can write your code from scratch if you wish.

Please note, that while this assignment has a lot of pieces to implement, most of the algorithms you will use in your project will be using the same pieces. Feel free to reuse any code you write for your homeworks in your class projects.

This is a challenging assignment. **Please start early!**

Solution

Write-up for implementation section:

Vanilla Q-learning update is equivalent to minimizing a quadratic loss function: Consider a Quadratic loss function $\mathcal{L}(\theta) = (\mathcal{T} - \mathcal{A})^2$ where $\mathcal{T}$ is the target. To minimize this loss function, we will have to update θ -

$$\theta := \theta + \alpha \nabla \mathcal{L}(\theta) \tag{7}$$
$$:= \theta + \alpha(\mathcal{T} - \mathcal{A}) \nabla_\theta \mathcal{A} \tag{8}$$

which is equivalent to vanilla q-learning update -

$$\mathrm{w} := \mathrm{w} + \alpha \left(r + \gamma \max_{a' \in A} Q_w(s', a') - Q_w(s, a)\right) \nabla_w Q_w(s, a).$$