Reinforcement Learning (Second Half): DQN Tutorial

Dr. Edward Johns

Monday 9th November 2020

1 Introduction

Overview. This tutorial will guide you through implementing a Deep Q-Network (DQN) using Python and Torch, from scratch. The tutorial begins with detailed instructions on which line of code you should edit, and as the tutorial progresses, the instructions become less direct, requiring you to think carefully about how you would implement the next part. The tutorial is based on content from Lectures 1 and 2 in the second half of the course. It is also directly aligned with the second coursework, so it's important that you complete the entire tutorial in order to complete this coursework. The first part of the coursework involves answering four questions, and each question corresponds to one section of the tutorial. You will see these questions highlighted in red. For example, if you see **Coursework Question 1** written in the tutorial, this means that this part of the tutorial is directly related to Question 1 in the coursework. You may want to save your code after each section of the tutorial, so that you can re-visit the different stages later, when completing your coursework. Be sure to make use of the three lab sessions on Microsoft Teams, where you will be able to discuss your progress with other students, and ask questions to both me and the GTAs.

Installation. The recommended operating system is Ubuntu. However, running the code in Windows or Mac OS will also work. The code is in Python 3, and you will need to install the following Python 3 packages: torch, numpy, opencv-python, matplotlib. To install these on Ubuntu or Mac OS using Pip, you can run the following: "**pip3 install numpy==1.19.4 opencv-python==4.4.0.46 torch==1.7.0 matplotlib==3.3.2**", which will ensure that the versions you install are compatible with this tutorial and the coursework. Please speak to a GTA during the first lab session if you need assistance with any installations, including how to install these packages on Windows.

Torch. Torch is a Python library for deep learning. For students who have not used Torch before, there is a brief example which is discussed below. You will also need to refer to the Torch documentation online, for guidance on how to use certain

functions. In this tutorial and the coursework, you will be using the CPU for training in Torch, and will not be using a GPU.

2 Torch Example

There are many Torch tutorials online which you may want to look through, if you have not used Torch before. But most of the Torch functionality you will need is present in the file **torch_example.py**. This trains a simple neural network, and it will help you implement your DQN. The script should run as it is by running "**python3 torch_example.py**" from the command line. You should see a graph plotting the loss as a function of the number of training iterations. A decreasing loss means that the network is getting better at predicting the label for a given input. You can try changing the optimiser's learning rate, the mini-batch size, and the network architecture, to study their effect. Read through the code and the comments, to make sure that you understand what is going on.

3 Starter Code

Running the code for the first time. To start off your DQN implementation, run **python3 starter_code.py**. If this results in any errors, such as missing packages, you should fix this here. If the code runs correctly, you should see a yellow circle moving from left to right, along the bottom. If the window is too large or small, then take a look at the code in line 123 to adjust the magnification of the display. The main window represents the entire environment which is available to the agent, the red rectangle represents the obstacle, the blue circle represents the goal, and the yellow circle represents the agent's current state. The aim of this tutorial is to train the agent to be able to reach the goal whilst avoiding the obstacle. Of course, we could simply program this manually (go right ... then go up), but then you wouldn't learn about Deep Reinforcement Learning! This is a simple enough environment that you can gain some important intuition, without the training taking too long as it would do if you were to train an agent to play "real" computer games.

Inspecting the code. Take a look at the contents of **starter_code.py**, which implements the foundations of a Deep Q Network. It is well commented, so try to understand what has been implemented so far, although you will become more familiar with the code as you progress through the tutorial. To start with, take a look at the code below line 118, which shows the main flow of the program. Notice the two loops: the outer one loops over episodes, and the inner one loops over the agent's steps within each episode. When you run the code, each episode starts with the agent in the bottom left, and each step moves the robot a small amount to the right. Once you understand these loops, you can work backwards to try to understand the rest of the script. You may also wish to take a look at **environment.py**, which defines the geometry of the environment in which the agent is acting. But make sure that you do not make any modifications to **environment.py**.

Note: When the **environment** object is created in line 123, the argument **display=True** is used. This means that the environment will be displayed after each agent step. However, this is purely for visualisation, and to speed up training this can be turned off using **display=False**. You can also speed up training whilst still keeping the visualisation on, by removing the "sleep" function in line 138. In Part 2 of the coursework, your code will be timed with this visualisation turned off, and you should remove line 138 when you submit your code, otherwise your code will run very slowly.

4 Defining the Action Space

Increasing the action space. In line 136, the agent takes a step in the environment. Take a look at line 32, where **Agent.step()** is implemented, and the function **_choose_next_action()** is called. Here, the discrete action is always set to 0. But we would like the agent to choose from four different discrete actions: right, left, up, and down. This will allow the agent to move all over the environment. Therefore, your first task is to edit **_choose_next_action()** so that the agent chooses a random discrete action, from the range [0, 1, 2, 3].

Understanding the continuous environment. However, whilst the action is discrete, the "real-world" environment itself works with continuous actions. Therefore, you also need to convert the discrete action into a continuous action before we can apply it to the environment. Take a look at the function **Agent._discrete_action_to_continuous()**. The function returns [0.1, 0] as the continuous action when the discrete action is 0. The environment interprets this as [x-movement, y-movement], where positive x-movement is to the right on your screen, and positive y-movement is upwards on your screen. This is why, in the original starter code, the agent always moved to the right: it moves 0.1 to the right on each step, and 0 upwards on each step. The environment itself has a width of 1.0 and height of 1.0, and the agent's state space is between 0.0 and 1.0 in both the x and y dimensions. Therefore, currently the agent moves 10% of the width of the window on each step. And the agent cannot move outside the environment's perimeter; any action attempting to do so results in the agent staying still for that step. This is why the agent stayed still for a few steps once it reached the wall, until the episode was reset. Similarly, the agent cannot move through the obstacle.

Converting discrete actions to continuous actions. Your next task is to edit the function **Agent._discrete_action_to_continuous()** so that it returns a different continuous action for each discrete action. This will allow the agent to move in all four directions: right, left, up, and down. Each discrete action should return a different continuous action, represented by a 2-dimensional NumPy array in the form [x-movement, y-movement]. It does not matter which discrete action corresponds to which continuous action, so this is your choice. But the magnitude of the continuous action must be 0.1, such that the agent takes a step of size 0.1 when executing an action. Once you have completed this function, run the script again. You should see the agent moving randomly all over the window now, because each random discrete action is then converted into a continuous action, which is

then applied to the environment.

5 Introducing the DQN

(Relevant to Coursework Question 1).

Sending a transition to the DQN. Currently, the agent just moves randomly around the environment. We will now start training the DQN so that the agent can choose actions more intelligently than random selection. First, add the line **loss = dqn.train_q_network(transition)** directly below the line **transition = agent.step()** in the inner loop of the main program loop. The transition is a tuple of [*state*, *action*, *reward*, *next state*], and we are going to train the DQN on each transition that the agent experiences.

Predicting the reward. In the function **DQN._calculate_loss()**, we need to calculate the Q-network's loss based on this transition. For now, let us train the Q-network to just predict the reward for this transition. In the context of regular Q-learning, this is the same as having a discount factor of 0, i.e. the agent only cares about its immediate reward and does not consider future rewards in its Q-values. In the code, you can see that the reward is defined as $0.1 \times (1 - d)$, where d is the distance between the agent and the goal.

Defining the loss function. Your task now is to complete the **DQN._calculate_loss()** function, so that the Q-network learns to predict the reward it will receive for taking a particular action in a particular state. The loss function for the neural network should be the mean squared error between the Q-network's prediction for a particular state and action, and the actual reward for this state and action. You can use the code in **torch_example.py** to help you implement the loss, paying particular attention to the comments in lines 60 and 68.

Plotting the loss. Run the new script and plot a graph showing the loss as a function of the number of episodes the agent has experienced. This graph can be plotted using MatPlotLib, using the code in **torch_example.py**. You should see the loss decrease as the Q-network learns to predict the reward. Throughout the tutorial, you can plot a similar graph after each section you complete, and see if the behaviour in the graph makes sense to you.

6 Introducing the Experience Replay Buffer

(Relevant to Coursework Question 1).

Adding a container. So far, the Q-network is trained on each transition online. Your task is now to implement an experience replay buffer, to allow for training of transitions in mini-batches. To do this, create a class called **ReplayBuffer**, which has one class attribute: a **collections.deque** container. Read up on the official documentation for this container, to understand how it works. In summary, it allows you to dynamically create a large dataset of data tuples, where you can

easily add new tuples whilst removing old ones. This container must be initialised with a maximum capacity; use 5000 for now. For example, defining this class atribute could look this: **buffer = collections.deque(maxlen=5000)**.

Interacting with the buffer. Transitions will be added to this container as they are generated by the agent, so you also need to add a class function which can append a transition tuple to the container. This can be done using the **collections.deque.append()** function. Then, you need to add some code to sample a random mini-batch of transitions from the replay buffer. Indexing a **collections.deque** container is very similar to indexing a regular Python list.

Training with the replay buffer. After creating this class, you should decide how to incorporate the replay buffer into your main training loop. As the agent steps through the environment, instead of training on each transition online, you should add the transition to the buffer. So to start with, you can remove the line **loss = dqn.train_q_network(transition)**, and replace it with a line that will add the transition to the buffer, using the code you wrote above. Then, after each step in the environment, write some code to sample a mini-batch from the replay buffer, and train the Q-network using this mini-batch. You should look at **torch_example.py** to understand how to train on mini-batches of data. For now, use a mini-batch size of 100. You will also need to implement functionality to wait until the replay buffer has at least 100 transitions, before starting to train the Q-network. Until then, the agent can just generate transitions without doing any training.

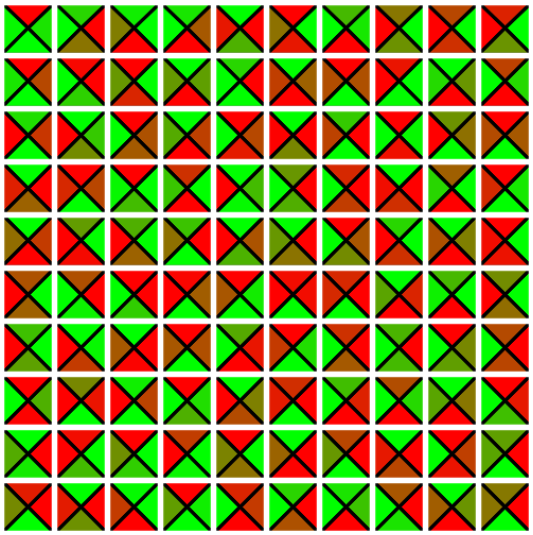
7 Visualisation

(Relevant to **Coursework Question 2**).

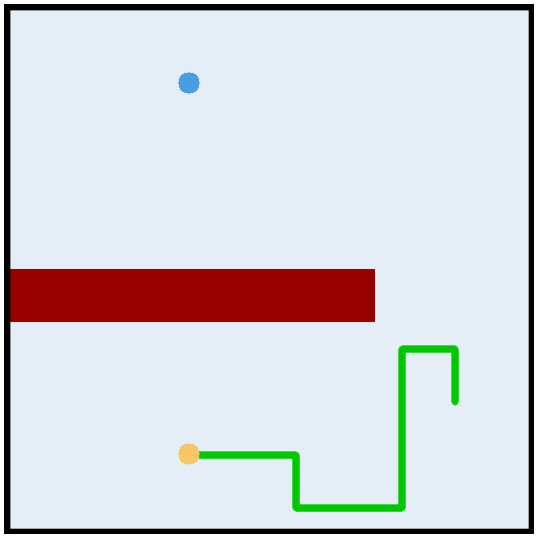
Visualising Q-values. It is very useful to be able to visualise the predicted Q-values of a DQN during training, to gain some intuition behind the algorithm, and to help with any debugging that may be required. Your task is now to create an image which visualises the Q-values for each state, across all four actions. The image should have the following structure:

Here, each square represents one state, and each triangle represents one action within that state. The more green a triangle is, the higher the Q-value is. The more red a triangle is, the lower the Q-value is. Each action is normalised relative to the other actions in that state, which makes it clearer which action in a state has the highest Q-value. So the figure just visualises the relative Q-values within a state, not the absolute Q-values. In the above image, the Q-values are random, but for a trained agent, the visualisation of the Q-values should reveal what the agent has learned about how good different actions are.

You can create this image using whatever method you choose. The recommendation is to use OpenCV, since this allows basic shapes and lines to be drawn easily. To assist you, the file **q_value_visualiser.py** contains code that, when run, will create an image with random Q-values, similar to the above image. You should inspect this code and work out how to use it to plot the Q-values from your trained DQN.



Visualising greedy policy. You can also visualise the greedy policy. This is the policy that always takes the action with the highest Q-value. Try creating another image, where you plot the greedy policy as a green line, starting from the agent's initial state. This would look something like the image below, where in this example, an episode length of 10 is shown:



To do this, take a look at the code in **environment.py** that is used to display an image of the environment. You can then add to this image by plotting a sequence of green lines, each line joining one state to the next.

8 Long-Term Prediction

(Relevant to **Coursework Question 3**).

Introducing the Bellman equation. So far, the Q-network you have been training has actually just been predicting the rewards for a particular state and action. Your task is to now improve upon this, and introduce the full Bellman Equation. To

do this, you will need to modify the function you have written which computes the loss for the Q-network, such that instead of computing the Q-network's error compared to the reward, you compute the error compared to the expected discounted sum of future rewards. You will therefore need to use the Q-network to predict the Q-values for the next state in each transition, and then find the maximum Q-value in this state, across all actions. To select the maximum Q-values in the Bellman equation, across an entire mini-batch, you may want to use the **gather()** function, in a line similar to the following:
state_action_q_values = state_q_values.gather(
dim=1, index=action_tensor.unsqueeze(-1)).squeeze(-1).
Use a discount factor 0.9 in the Bellman equation.

Introducing the target network. One component of a DQN that can help with stability, is a target network. Introduce a target network into your **DQN** class, with exactly the same network architecture as the Q-network. Modify your DQN training function so that the target network is used, instead of the Q-network. Then, create a function in your **DQN** class which, when called, will update the target network by copying the weights of the Q-network over to the target network. Use the functions **torch.nn.Module.state_dict()** and **torch.nn.Module.load_state_dict()** to get and set a network's weights, respectively. Then, modify the code in your main loop so that the target network is updated every N agent steps, and study the effect of varying N. You should also think about how the Q-network and the target network differ, in terms of how each is updated. The Q-network is updated using gradients, whereas the target network is updated just by copying the Q-network. So when you compute the Q-values from the target network and use these Q-values to update the Q-network, you want to make sure that this doesn't also update the target network. To do this, you may want to use the **detach()** function, with a line similar to the following: **next_state_values = next_state_values.detach()**.

End of tutorial