CO424 Reinforcement Learning Part 2: Tutorial

Dr. Edward Johns

Wednesday 6th November 2019

1 Introduction

This tutorial will guide you through implementing Deep Q-Learning using Python 3 and Torch, through several stages. It is aligned with the coursework, so it's important that you complete the entire tutorial in order to complete the coursework. Questions 1, 2, and 3 in the coursework are directly related to figures that will be created during this tutorial. When the tutorial mentions a question from the coursework, it is advised to complete this coursework question before moving onto the next stage in the tutorial. You may wish to save your Python script after each section of the tutorial, to keep the different stages of implementation separate. This will help you to return to old code when completing different parts of the coursework.

The overall coursework mark for this course has a high weighting of 50%, so you will be expected to spend significant time outside of timetabled hours working on the tutorial and coursework. The coursework for this second half of the course has two parts to it. The first part (worth 60% of the total) is released alongside this tutorial, and the second part (worth 40% of the total) will be released on Tuesday 12th November.

Be sure to make the most of the two lab sessions, because implementing Deep Q-Learning can be challenging and you are very likely to have questions to ask. You will be free to work on the tutorial and coursework during the lab sessions, as you see fit. For students who have not used Torch before, there is a brief tutorial which is discussed below. The remainder of the tutorial guides you through different stages of implementation, starting off with specific instructions in the early stages, and more general ideas in the later stages which you will have to design your own solutions for. You may need to refer to Python 3 and Torch documentation online to be able to implement certain functionality, and be sure to refer back to the lecture notes for more in-depth explanation of the different components you are implementing.

If you are working on your own laptop, you will need Python 3 installed, together with the following Python packages: torch, numpy, opencv-python, and matplotlib. If you are working on a lab PC, you will need to create a Python virtual environment, before installing these packages using PIP. To be sure that your code will run correctly when it is evaluated in Part 2 of this coursework, you will need to make

sure you have specific versions of the above packages installed. Details on these versions, and on how to set up a virtual environment, can be found on GitLab by clicking here.

2 Torch Tutorial

There are many Torch tutorials online which you may wish 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 can form the basis of the code which you will implement for training your Deep Q-Network (DQN). The script should run as it is by running **python3 torch_example.py** from the command line, and you should see the loss decreasing as the network is trained. 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 batch size, and the network architecture, to study their effect. In this tutorial, you will be using the CPU for training in Torch, and will not be using the GPU.

3 Starter Code

3.1 Code Familiarisation

Take a look at the contents of **starter_code.py**. This provides Python 3 code which you will build upon during this tutorial and the associated coursework. The starter code implements the foundations for Deep Q-Learning. 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 128, 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. Once you understand these loops, you can work backwards to understand the rest of the script. You may also wish to take a look at **environment.py**, but make sure that you do not make any modifications to **environment.py**.

Before you start any coding, in line 115 you will see the line **CID = 123456**. Replace "123456" with the last 6 digits of your CID number. This will create a unique random seed for both NumPy and Torch, which will personalise your results. Deep Reinforcement Learning can be highly sensitive to random seeds, and so your results may be different to those of other students. The general trends, however, should be similar.

Each time the agent takes a step, the environment will be drawn on the screen. Try running the code by typing **python3 starter_code.py** into the command line. You should see a black window, with a grey rectangle, a green circle, and a red circle moving to the right. If the window is too large or small, then take a look at the code in line 121 to adjust the magnification of the display. The black window represents

the entire environment which is available to the agent, the grey rectangle represents the obstacle, the green circle represents the goal, and the red 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 Q-learning! This is a simple enough environment that you can gain some important intuition without the training taking too long. And for tasks which we do use deep reinforcement learning for in practice (e.g. training a robot to control its hand), it is almost impossible for us to manually program the controller.

Note: When the **environment** object is created in line 122, 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**. In Part 2 of the coursework, your code will be timed with this visualisation turned off.

3.2 Defining the Action Space

In line 135, the agent takes a step in the environment. Take a look at line 32, where **Agent.step()** is implemented. Here, the discrete action is set to 0. There are actually four discrete actions: left, right, up, down, but the agent currently always chooses the same action in every step. Therefore, your first task is to edit line 34 so that the agent chooses a random discrete action, from the range [0, 1, 2, 3].

However, whilst the action is discrete, the environment itself works with continuous actions, as with all real-world environments. 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()** in line 55. As it stands, the function always returns [0.1, 0] as the continuous action, because the discrete action is always 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 the agent currently moves to the right when you run the code: 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 currently moves to the right, and then stays still for a few steps, until the episode resets. Similarly, the agent cannot move through the obstacle.

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. 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.

3.3 Defining the Loss Function

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. 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 short-term reward and does not consider future rewards in its Q-values.

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. This loss 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 47 and 55.

4 Online Learning and Mini-Batch Learning

Run the new script and plot a graph showing the loss as a function of the number of steps the agent has taken in the environment. This graph can be plotted using MatPlotLib. You should see the loss decrease as the Q-network learns to predict the reward. This graph will be used in Coursework Question 1.

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 batches. To do this, create a class called **ReplayBuffer**, which has one class attribute: a **collections.deque** container. This container stores a set of transition tuples, and it must be initialised with a maximum capacity; use 1 million for now. 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 to the container. This can be done using the **collections.deque.append()** function. Then, you need to add a function to sample a random batch of transitions from the replay buffer. Random indices can be generated using NumPy, and these can then be used to index a **colletions.deque** object.

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 class function you implemented 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. For now, use a mini-batch size of 50. You will also need to implement functionality to wait until the replay buffer has at least 50 transitions, before starting to train the Q-network. Until then, the agent can just generate transitions without doing any training.

Once this is done, run the new script and plot a graph showing the loss as a function of the number of steps the agent has taken in the environment. You should see different behaviour in the graph compared to the graph with online learning. This graph will be used in Coursework Question 1. You may wish to save a copy of your current script at this point.

5 Visualisation

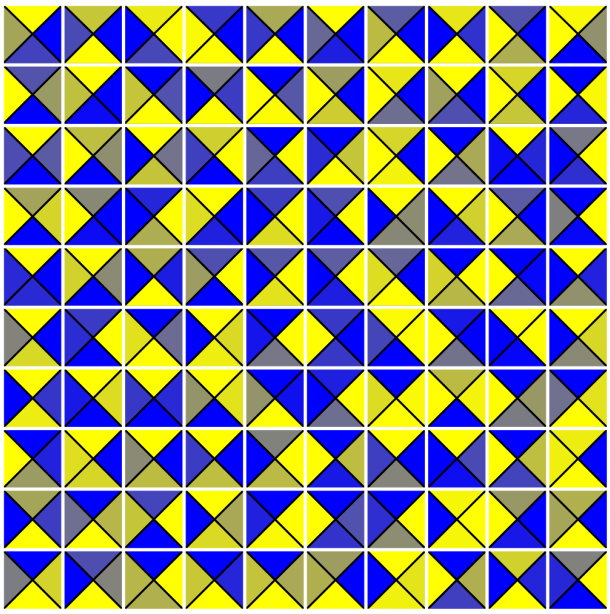
It is very useful to be able to visualise the predicted Q-values of a DQN during training, to gain some intuition behind the algorithm, and to help with any debugging that may be required. Your task is now to create an image, using OpenCV, which will visualise the Q-network's predictions for all four actions, across a grid structure covering the environment's full state space. You can read how to draw basic shapes in OpenCV by clicking here. The **Environment.draw()** function in **environment.py** may also help.

First, create a square NumPy array, of three dimensions: x-position, y-position, and colour. This array represents the pixels in your image. The size of the x and y dimensions is your choice, but the size of the colour dimension should be three. Then, draw a 10-by-10 grid of cells with white borders, each representing a 0.1 x 0.1 region of environment space (recall that the agent's state lies in the range 0 to 1 in both the x and y dimensions). This can be done using the **cv2.line()** function. Note that the origin when drawing using OpenCV is in the top-left of the image, whereas the origin of the state space is in the bottom-left of the image. Therefore, you will need to flip the y-coordinates when converting from state space to OpenCV image space.

Then, for each cell, use the Q-network to predict the Q-values across all four actions, for the state representing the centre of the cell. These should be visualised by drawing four triangles in each cell, representing the four different directions. Each triangle should be coloured in, using the function **cv2.fillConvexPoly()**, and given a black border using the function **cv2.polylines()**. The "points" argument of **cv2.fillConvexPoly()** is a NumPy array of the form **numpy.array([[point_1_x, point_1_y], [point_2_x, point_2_y], [point_3_x, point_3_y]], dtype=np.int32)**.

The colour of each triangle should reflect the Q-value of that particular action,

normalised with respect to the other actions in that state. In this way, it should be clear which action in each cell has the highest Q-value. For each state, use a dark blue colour for the action with the lowest Q-value, a yellow colour for the action with the highest Q-value, and linear interpolation in colour space for the other two actions. Note that OpenCV requires pixel colours to be in the following order: blue, green, red. The image below shows an example of the structure of this image. Here, the Q-values are randomised, although actions within one cell are normalised. To ensure that the white grid cell lines are shown as in the image below, you should draw the grid cells after drawing the triangles. This image will be used in Coursework Question 2.

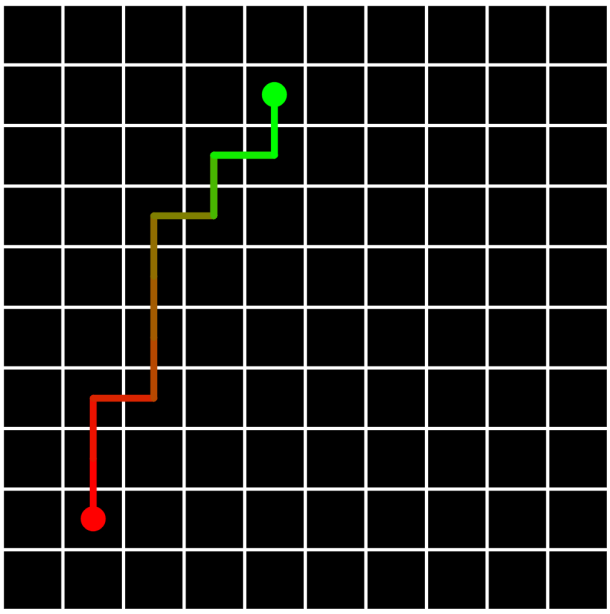


Visualisation of Q-values

As well as visualising the Q-values, it is useful to be able to visualise the agent's policy, if it were to take the action with the highest Q-value in each state, i.e. the *greedy* policy. To do this, you will first need to add some code to your script which will compute this greedy policy. This can be done in the main loop during one episode, where instead of the agent taking a random action, the agent takes the action corresponding to the highest Q-value in each state. This will provide a set of states which the agent observes as it executes this policy. Note that computing the greedy policy should be separate from the main training loop. Otherwise, if you remove any random exploration, then the agent will never learn. In the coursework, you will implement ϵ-greedy exploration, which combines random exploration with the greedy policy. But for now, the Q-network should only be trained with random exploration, and computing the greedy policy for visualisation should not interfere with this.

For visualisation, create an image similar to the Q-value visualisation, with a 10-by-10 grid of states. Then, draw the states from the greedy policy on the image, with a series of connected lines between the states. The first state in the trajectory should be the agent's initial state, and the final state in the trajectory should be the

state at the end of the trajectory. Therefore, the number of states to draw should be one greater than the number of steps in the episode. The start of the trajectory should have a red circle, the end of the trajectory should have a green circle, and the connected lines should change from red to green along the trajectory, with linear interpolation in colour space. The image below shows an example of this image. Here, the policy is random. This image will be used in Coursework Question 2. You may wish to save a copy of your current script at this point.



Visualisation of Greedy Policy

6 Long-Term Prediction

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. Use a discount factor 0.9 in the Bellman equation. Once this is done, train the agent again, and plot the losses as a function of the number steps the agent has taken in the environment. This graph will be used in Coursework Question 3.

One important component of a DQN is a *target network*, which is used instead of the Q-network in the Bellman equation. 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 20 agent steps. Once this is done, train the agent again, and plot the losses as a function of the number of steps the agent has taken in the environment. This graph will be used in Coursework Question 3. You may wish to save a copy of your current script at this point.

End of tutorial