Imperial College London

8 November 2024

Department of Computing
Reinforcement Learning – Dr Edward Johns, Prof Aldo Faisal & Dr Nicole Salomons
Coursework design – Prof Aldo Faisal, Filippo Valdettaro & Charles Pert

Assessed Coursework 2

Version 1.0 To be submitted online via Scientia.

The coursework should be submitted on Scientia by the 22nd of November, 7pm, and consist of the following files:

- A **PDF** of your written report, that has to be named ***coursework2_report.pdf***. The first page of the coursework has to contain your name, your CID, department and course (e.g. "MSc Advanced Computing").
- A **.ipynb** (Python notebook) file named ***coursework2.ipynb*** which includes code you used to run experiments and generate the final plots in your report.
- A **.py** (Python) file named ***utils.py*** with any additional functions or classes you can define to be imported into your notebook.

There is no LabTS submission or automated code marking for this coursework. Please ensure that you are familiar with the Scientia submission process well before the deadline as we are, unfortunately, not allowed to mark emailed or printed hardcopy submissions.

Report: Your report should not be longer than 7 single-sided A4 pages with at least 2-centimetre margins all around and font of size no smaller than 11pt. Appendices are not allowed and will not be marked. 7 pages is a **maximum** length, shorter reports are fine, but a penalty will be incurred for going beyond the page limit. Submissions that don't adhere to these guidelines may receive penalty marks. The cover page, table of contents and references do **not** count towards the page limit.

Code: Please **include the completed and commented source code as part of your submission** in the `utils.py` and `coursework2.ipynb` files. You are provided with an unoptimised (with respect to the hyperparameters) DQN model, implemented in starter versions of the `utils.py` and `coursework2.ipynb` given files. You are allowed to modify both of these as you wish, and you should produce your own code for questions where you are expected to modify the structure of the DQN (e.g. to adjust hyperparameter variables).

You are encouraged to discuss the general coursework questions with other students, but your answers should be your own. This means your answers should be written by you and in your own words, demonstrating your understanding of the content and the question. Your report and code will be automatically verified for plagiarism. Written answers should be clear, complete and concise. Figures should be clearly readable, labelled, captioned and visible. Incomplete answers and figures, irrelevant text not addressing the point and unclear text may lose points.

Marks are shown next to each question. Please note, these marks are only **indicative**. If you have questions about the coursework please make use of the labs or EdStem, but note the Graduate Teaching Assistants (GTAs) cannot provide you with answers that directly solve the coursework.

Overview

Problem description

Your goal is to train an agent to balance a pole attached (by a frictionless joint) to a moving (frictionless) cart by applying a fixed force to the cart in either the left or right direction. Please see Fig. 1 for an illustration. The aim is to train the DQN to keep the pole balanced (upright) for as many steps as possible. We do not control the magnitude of force we apply to the cart, only the direction. The optimal policy will account for deviations from the upright position and push the cartpole such that it remains balanced.

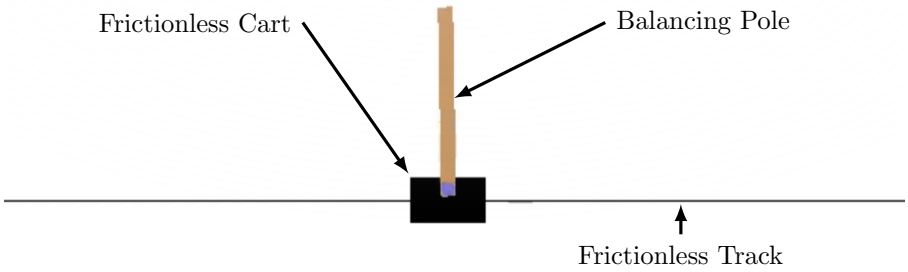

Figure 1: Illustration of the OpenAI Gym CartPole environment.

Our action space is discrete and of size 2. We have action 0, apply a force on the cart to the left, or action 1, apply a force on the cart to the right. The observation state we obtain from the environment is of size 4 of the following positions and velocities: cart position, cart velocity, pole angle and pole angular velocity.

All four observations of the environment are initially assigned a random value between -0.05 and 0.05. So the cart starts close to the origin with the pole almost upright and a low initial angular velocity. The aim is to keep the pole upright for as many steps as possible. This makes the reward quite simple to define: for each step taken (including final step) a reward of $+1$ is returned. The environment will only terminate when it reaches any of the following states:

- pole angle greater than $\pm 12°$ (or equivalently 0.2094 radians)
- cart distance from centre greater than ± 2.4,
- or the number of steps exceeds 500.

DQN implementation

Recall from the lectures, a DQN is a neural network designed to predict the Q function (for all the possible actions) of the environment given a state vector. The DQN provided works on the Gym "CartPole" environment. Please see Fig. 2 for an example. The first layer takes as input the observed state. The number of outputs in the final layer of the network must be the same number of actions the agent can perform. This is how the state-action values are encoded in the neural network: the DQN takes a state as input, and its n^{th} output neuron's value is the learned Q-value at the input state for the n^{th} action.

The code provided alongside this assignment contains a PyTorch implementation of a simple DQN architecture, along with sample plotting and visualisation code, and trains the model to predict the action the agent should take to balance the cart pole. However, the model is not optimised and therefore does not converge to consistently balance the pole. Below, you can find a description of the functions and classes included in `utils.py`. You are strongly suggested to understand how these are implemented, and you may modify these as you wish.

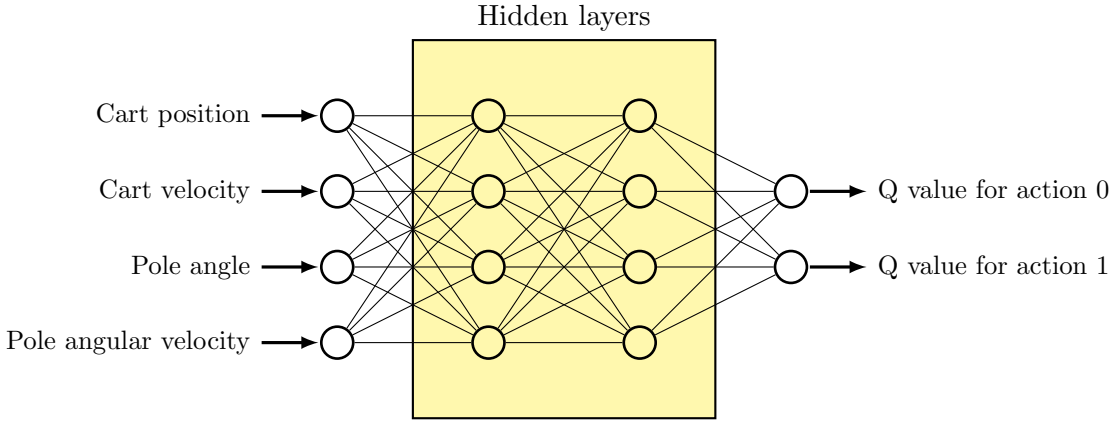

Figure 2: Diagram of a DQN with input layers and output layers matching those required for this environment. Please note, **you** decide the parameters for the hidden layers (i.e., number of layers and number of parameters per layer).

Replay buffer

A replay buffer is implemented in the `ReplayBuffer` class. At initialisation it takes an integer as argument which is the maximum number of transitions it can hold. It includes the following methods:

- `push()` method: Adds the object that is passed as input to the replay buffer's memory. If an item is added when the replay buffer is at full capacity, it discards the oldest item it has in memory and replaces it with the newest. This method returns the updated replay buffer's memory (an iterable object that contains the objects pushed to it as elements).
- `sample()` method: It returns an iterable object of items uniformly sampled (without replacement) from the replay buffer's memory. It takes as parameter an integer defining the number of samples to be returned.

Defining a DQN

The `DQN` class (inherited from `nn.Module`) that will be your multi-layer DQN perceptron. It includes the following `__init__()` and `forward()` methods:

- `__init__()`: At initialisation, it takes a list of integers defining the number of neurons in each layer of the DQN. For example, a network with 2-d input, 1-d output and 2 hidden layers of size 50 will take the list `[2,50, 50, 1]` as initialisation parameter.
- `forward()`: Implements a batched forward pass through the neural network, using a ReLU activation function. Carrying forward the example from above, for an input batch tensor of shape `(N, 2)` the output should have shape `(N, 1)`. The DQN also handles non-batched states: so for an input of size `(2,)` the output is of size `(1,)`.

Action selection

- The function `greedy_action()` takes two parameters as input: a `DQN` object and a (non-batched) state tensor and returns the integer corresponding to the greedy action at the given state according to the `DQN`.

ε-greedy policy

The function `epsilon_greedy()` takes three parameters as input: an `epsilon` float, a `DQN` object and a (non-batched) state tensor. `epsilon_greedy()` returns an integer representing a stochastic sample of the epsilon greedy action at that state according to the `DQN`.

Target network

The function `update_target()` takes two `DQN` objects as arguments, and updates the parameters of the first one copying the weights and biases from the second.

Loss calculation

The function `loss()` computes the Bellman error for a batch of transitions. `loss` takes 7 arguments: two DQN objects (a policy and target network) and batched (i.e. two-dimensional) tensors with states, actions, rewards, next states and 'dones' in that order. For reference, the loss for a batch $\mathcal{B}$ of size N is computed according to the formula

$$L(\theta) = \frac{1}{N} \sum_{s,a,r,s' \in \mathcal{B}} (Q(s,a) - (r + \max_{a'} \hat{Q}(s',a')))^2,$$

with Q the first (policy) `DQN`, $\hat{Q}$ the second (target) `DQN`. This implementation carries out the sum in parallel (as a batch) rather than looping through each transition, as this enables training to be much faster.

Hint: Understanding PyTorch's `gather()` function will be useful to follow the given implementation.

Question 1: Tuning the DQN – total 20 pts

The DQN provided to you in the code is unoptimised in the cart pole environment. Nine standard DQN hyperparameters are present in the provided code with variable names `A`, `B`, ..., `I` and initialised with a default, unoptimised value of 1. In this question, your task will be to inspect and understand the code to identify what hyperparameters these correspond to and to optimise them to obtain a functioning DQN.

Note: the provided code is compatible with the latest version of OpenAI Gym, which may be different to the version present in Colab. If you wish to work on Colab you can upload the notebook and `utils.py` files to Colab, and you may have to make some small modifications to the lines that contain calls to `env.reset()` and `env.step()`. You can refer to the provided Colab notebook for Lab Assignment 3 for how these lines should be modified. Installation of both PyTorch and Gym are covered in Lab Assignment 3 if you wish to instead work locally.

We provide a threshold for success in Fig. 3, along with a reference plot of what the untuned agent's learning curve would look like as well as a successful agent's. Achieving an agent that reaches an average reward of 100 over 10 runs of training for over 50 episodes will grant you full performance marks. It is possible to achieve this with the ingredients that are already present in the provided code by tuning the hyperparameters and introducing basic exploration-exploitation handling (which may also require some tuning).

Label	Hyperparameter	Value
`X`	Batch size	42

Table 1: Example hyperparameter table row. The constant `X` was identified as corresponding to batch size and the value for batch size that was settled on for the final DQN implementation was 42.

Question 1.1: Hyperparameters – 10 pts

Identify and adjust the hyperparameters of the DQN to tune your agent to perform well in the cart pole environment. The performance of the tuned DQN should be comparable to (or better than) the example run found in Fig. 3. Produce a table with three headings: "Label", "Hyperparameter", "Value" (see Table 1). For each row, identify and name which hyperparameter corresponds to each of the constants `A` to `H` in the provided notebook (do not include the `NUM_RUNS` constant) and then report the value that you decided to settle on for your final DQN implementation. Finding a successful combination of hyperparameters will require a combination of intuition and experimentation, and you do not need to justify how you chose the hyperparameter values.

Once you have identified the hyperparameters in the provided code, you are free to rename them or modify the code structure as you wish. You don't need to adhere to the provided code structure and may modify other hyperparameters that are implicitly present or introduce algorithmic tweaks if you wish to do so. If there is some parameter in this list that you don't directly use in your final implementation, for example if you identify one parameter as a constant ϵ but decide to use ϵ-decay instead, mention this in the "Value" column. In such cases, describe how your implementation deviates from directly using this hyperparameter, giving values of any other hyperparameters that you introduce instead.

Finally, **describe what exploration schedule you use and state the value of any additional hyperparameters** you introduce here. You do not need to justify how you chose these.

Question 1.2: Learning curve – 10 pts

Training a DQN is not always a deterministic process and you may get potentially large variation between training runs. Replicate the DQN training 10 times and produce a plot of the mean return per episode with the standard deviation you have observed over the replications.

Hint: Do not go overboard with large computational simulations. For full performance marks, the final learning curve should consistently achieve an average episode length of 100, by achieving such a target for at least 50 consecutive episodes (that is good enough). See Figure 3, which illustrates how a successful averaged learning curve passes this threshold as well as a sample learning curve from the unoptimised DQN that we provide you with, trained for 300 episodes. The number of training episodes shown (300) is only a reference and you can train your agent for a different number of episodes if you want.

Show the graphic and briefly comment on your results.

Question 2: Visualise your DQN policy – total 20 pts

Once you have tuned your DQN, you can proceed to investigate the policy and Q-values learned by the agent. To do so, you can run one more training run and store the resulting DQN.

The fact that the state-space is 4-dimensional means we cannot visualise the full environment at once. Instead, we will fix two dimensions to constant values in order to plot visual 'slices' of the environment. Throughout this section, we will be investigating the subset of the state-space where the cartpole is located at the centre of the track.

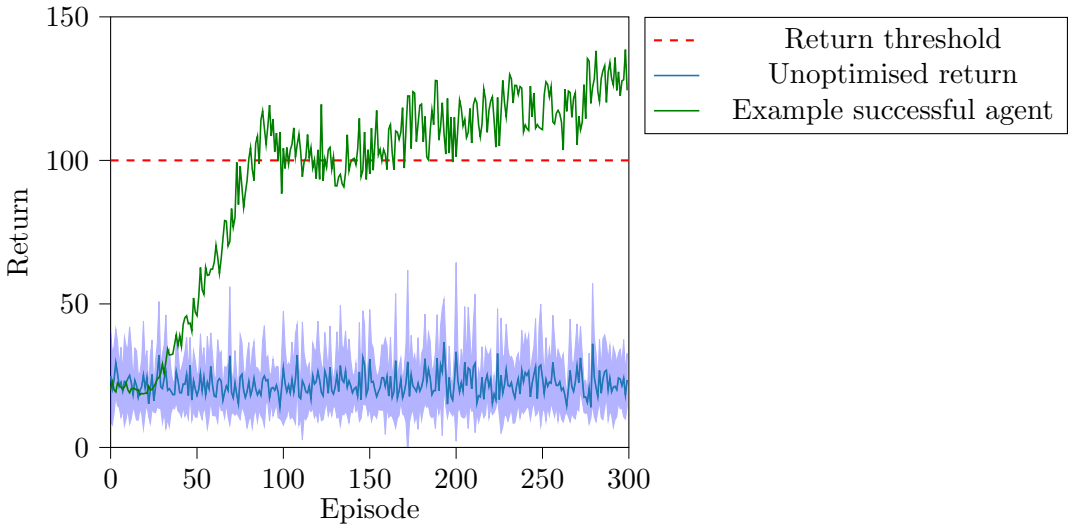

Figure 3: An example plot of return versus episode number during training.

Question 2.1: Slices of the greedy policy action – 10 pts

Plot the greedy policy according to your DQN in 4 separate two dimensional plots displaying pole angular velocity on the y-axis against pole angle on the x-axis. Fix the cart position to zero (centre of the track) and use cart velocities of 0, 0.5, 1 and 2. Use the "Cividis" colourmap ensuring you specify whether pushing the cart to the left is denoted by yellow or blue.

Show the graphic and comment on your results. In particular, explain and comment on what you expect a plot from an optimal agent to look like and whether your agent's learned policy matches it in terms of:

- The regions of the plot where the agent chooses to push left or right
- The general shape of the action decision boundary
- The symmetries of the action decision boundary when the velocity is 0
- How the action decision boundary shifts as velocity increases

Question 2.2: Slices of the Q function – 10 pts

Plot the greedy Q-values according to your DQN in 4 separate two dimensional plots displaying pole angular velocity on the y-axis against pole angle on the x-axis. Fix the cart position to zero (centre of the track) and use cart velocities of 0, 0.5, 1 and 2. Use the "Cividis" colourmap ensuring you specify how the colourmap denotes different Q-values.

Show the graphic and comment on your results. In particular, explain and comment on what you expect a plot from an optimal agent to look like and whether your agent's learned values match this in terms of:

- The regions of the plot where values are relatively higher or lower
- The range of values your agent has learned, both close and far from the edge of the episode termination region
- The symmetries of the learned values when the velocity is 0
- How the values change as velocity increases

How to make appropriate figures and graphs

There a basic standards in academia for how to make attractive and interpretable figures. Following this checklist below will not only help you avoid being penalised for poor figures, but also prepares you for the quality of figures that are needed for your final thesis:

- Are all axes are labeled, e.g. "Height"?
- Do all axis labels have units, do you specify them using the square bracket convention "Height [cm]" or if there are no units did you specifcy "[au]" for arbitrary units? If the axis is self explanatory, e.g. "Number of steps", you do not need to provide a unit bracket. (or blue and red, but do not use red and green, please)?
- If there are multiple lines/dots, is each a different line style and color?
- Are all lines sufficiently thick?
- Are all font sizes are legible?
- Is the figure shown at sufficiently high resolution e.g. DPI (400+), so that lines are crisp and not poor quality graphics?
- If there are multiple lines, is there is a legend, or other textual description of each line? whether they are standard error or standard deviation? If they are not either, is there a good justification provided for that?
- Are you axes tight (that is, are the bounds of the axes just larger than the max and min of what you want to show)? If not, do you have good reason for the excess, e.g. because you need to show absolute levels.
- Do all axes have either clear tick marks or gridlines indicating magnitudes of everything?
- Does the figure have a proper caption explaining what is shown, e.g. "Learning curve of DQN agent plotted as return per episode over time".