Imperial College London

November 8, 2022

Reinforcement Learning – Prof Aldo Faisal & Dr Paul Bilokon
Assignment design: Filippo Valdettaro & Manon Flageat

Lab Assignment 3: Gym and PyTorch

This lab assignment is meant to help you familiarise and play with the tools necessary to work on Deep Reinforcement Learning in practice. We are going to provide you with code skeletons and a simplified task to work through these. We are using two packages that are industry standard for Deep RL, namely OpenAI Gym and PyTorch. This assignment focuses on a classic control tasks, the inverted cart pole or inverted pendulum, the environment is illustrated in Figure 1.

The task consists in controlling the cart's left and right motion so as to keep the pendulum in balance pointing upward. The classic RL task in this environment is that of training an agent to balance an inverted pendulum by controlling the cart that supports it.

However, for **this** lab assignment we want to tackle a simpler task, our aim is to get you started on using function approximation (Deep Learning) in a hands on way. To this end we are going to use the inverted pendulum task, to solve a simpler problem: learning the dynamics of the pendulum form observational data. This is a supervised learning task that we can solve with any function approximator, and here we want to do it with Deep Learning in Pytorch. Down the line learning the dynamics of an environment can be used to improve the efficiency of an agent and is referred to as model-based RL.

We will be learning the dynamics of the environment (without training the agent) in a supervised learning setting: we want to observe the position and motion of the pendulum and the actions on the cart, to predict how at a subsequent moment in time, the cart (i.e. the environment) will have responded to this and thus predict the successor state. All we need to do is to get the cart to do "something" (we provide you with a policy for this) and then observe how state and action (input) lead to a successor state (output).

We will be using the following states arising in the environment as ground truth, to make you familiar both with the AI Gym environments and the deep learning capabilities of PyTorch.

Although you will be asked here to solve only a supervised learning problem, we will explain you this in the context of the more general RL task. You may then use this code as a basis for your own Deep RL projects or e.g. a coursework.

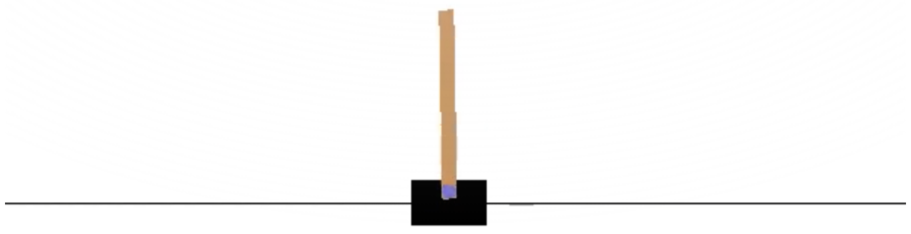

Figure 1: Illustration of the OpenAI Gym CartPole environment, focus of this lab assignment.

The CartPole environment

The usual goal of the CartPole task is to keep the pole on the cart balanced by applying a force to the cart at each time step. The corresponding MDP is defined as follows:

- **State:** At every time step, the state of the CartPole is given by 4 values: the position of the cart on the x-axis, the velocity of the cart along the x-axis, the angle of the pole to the vertical, the angular velocity of the pole.
- **Action:** The agent can choose to give the CartPole an impulse to the right or to the left. There is only one possible impulse force. So there are two possible actions: $a_0 = \textit{impulse to the right}$, $a_1 = \textit{impulse to the left}$.
- **Termination:** The episode terminates once the pole falls past 12° on either side of the vertical, or once the cart reaches the edge of the screen.
- **Reward:** Each transition gives a reward of 1, including transitions that put the CartPole into a terminal state. This encourages policy that keeps the pole up for longer.

Jupyter Notebook

This lab assignment is based on completing tasks in the provided Jupyter Notebook. You are encouraged to run this locally on the lab machines, but if you wish you can also find it directly on Google Colab:

`https://colab.research.google.com/drive/1puAxHVRn7zxjY4fVz5Gk-uaNrazrW3Uc?usp=sharing`

If you are running the notebook locally, you will first need to install PyTorch, OpenAI Gym and PyGame (for visualisation purposes) to your virtual environment. To do this simply activate your virtual environment (e.g. run `source rl/bin/activate`) and then run the command `pip install torch gym pygame moviepy` (the last two for visualisation purposes).

In the following questions, you will be asked to complete code in the sections marked with the tag `"[Action required]"`. The Notebook follows the same structure as the questions.

1 Introduction to Gym environments

1.1 Gym environments

OpenAI Gym is a set of environments released by OpenAI. It is used as a standard benchmark in the RL community across the world. It includes multiple environments ranging from classical control tasks to robotics locomotion tasks or robotics manipulation tasks. You can find short videos of these environments on the OpenAI website: `https://gym.openai.com/envs/`.

The simplest environments of OpenAI Gym are what we refer to as the classic control environments. In this assignment, we are going to focus on one of them: the CartPole environment `https://gym.openai.com/envs/CartPole-v0/`.

Each Gym environment is implemented as a single class. For example, the class `CartPole-v0` implement the CartPole environment as described before. Each class implements the dynamics of the corresponding environment and defines the states, actions, termination criterion and rewards.

1.2 Interacting with Gym environments

To interact with a Gym environment and use it with a reinforcement learning agent, we use two methods of its class (similar to the one you had in the first Coursework in this course):

- `reset()` that reset the environment and output the initial state.
- `step(action)` that takes as input an action, perform this action in the environment and output the states it leads to as well as the corresponding reward and a boolean indicating if the episode has terminated after this transition

Question 1: Creating the CartPole environment and performing an episode

Try to execute the provided notebook until Question 1 and study the code that is performing a simple episode in the CartPole environment to see how `reset()` and `step(action)` are used. Make sure you can access the saved `mp4` file of the episode in the `random_episode` folder. In Colab, This folder can be found in the file explorer on the left of the interface (little folder pictogram).

Question 2: Implementing a simple hand-designed policy

To help you understand how to interact with a Gym environment, we propose to implement a simple hand-crafted policy acting in the environment. We define a 'sensible' action as one that pushes the cart right if the pole's angle is leaning to the right and pushes the cart left otherwise. Fill in the function `simple_policy` to return a random action with probability p_random or a 'sensible' action with probability $1 - p\_random$.

2 Introduction to PyTorch

Deep reinforcement learning allows to apply Reinforcement Learning theory to continuous and high-dimensional domains, using Deep Neural Networks (DNN). DNNs are function approximators with a high degree of expressivity: their parameters can be trained from data to approximate a large set of complicated functions.

Among the various functions DNNs are used to approximate in deep reinforcement learning are the value function, the state-action value function, or even the policy. Here, we will be approximating the function that describes the dynamics of the CartPole, what the next state should be given the current state and action, $s_{t+1} = f(s_t, a_t)$.

We will be using PyTorch to implement DNNs, as this library offers a range of convenient features that greatly simplify their implementation. Throughout this lab assignment you will have the opportunity to use PyTorch and build familiarity with it.

2.1 Tensors in PyTorch

Tensors are n-dimensional arrays and are the building blocks of PyTorch. They behave very similar to NumPy arrays, sharing many of the same indexing rules, but have additional capabilities that allow them to be used for automatic differentiation and faster parallel processing with hardware acceleration. If you are familiar with NumPy arrays, you can treat PyTorch tensors in an equivalent fashion for most operations.

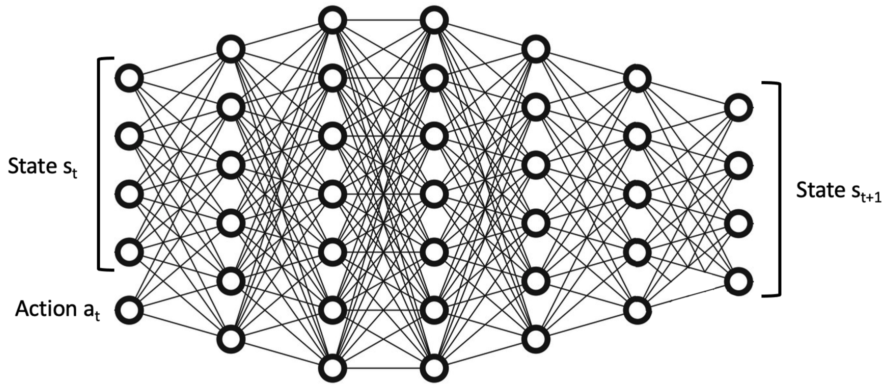

Figure 2: Illustration of the predictor that will be implemented during this lab assignment. The input is the current state s_t and current action a_t, the output is a prediction of the next state s_{t+1}. This illustration shows a DNN with 5 hidden layers, but your implementation may vary.

2.2 DNNs in PyTorch

PyTorch streamlines the implementation of DNN architectures as much as possible. In the provided Notebook the class `MLP` is an example of such DNN. A DNN in Pytorch is implemented as a class that inherits from `nn.Module` and has the following methods:

- `__init__`: defines the DNN's architecture and parameters.
- `forward`: takes an input through the DNN and computes the corresponding DNN's output. We call this a forward pass through the DNN.

PyTorch comes equipped with several common network building blocks, such as implementations of linear and convolutional layers. In this lecture, we will be employing architectures that only use linear layers, of the `nn.Linear` class. A `nn.Linear` object corresponds to the linear transformation applied to a layer of your DNN. For example, the first `nn.Linear` object of the `MLP` class corresponds to the transformation from the input of the network to the pre-activation of the first hidden layer. The `__init__` function of `MLP` initialises all the `nn.Linear` of the DNN.

To pass an input `x` through one of these transformations `lin`, you can call `lin(x)`. This is what is done in the `forward` method of `MLP`: the input `x` is passed through each of the network's layers. `forward` also applies relu activation to the output of each layer through the `nn.functional` class.

Question 3: Understanding the MLP class

As described above, the MLP class implements a simple not-yet-trained DNN. Try to execute the code for Question 1 and to feed inputs into the MLP instance to understand the forward function. Test different value of hidden size, input size and output size for the network.

2.3 Training DNNs with PyTorch

2.3.1 Loss functions

To train a DNN you need to equip it with an objective for it's parameters to minimise during training. This is what we call the loss function. In Deep learning for regression, we often use the mean squared error (MSE) between a target value and the prediction of the network.

Our CartPole state-predictor predicts the next-state and we want to compare it to the observed next-state. We can choose to compute the MSE between the target next-state and the predicted next-state.

2.3.2 Optimizer

Optimizers are PyTorch objects that act on a DNN's parameters to optimise them through gradient descent given a loss function. `torch.optim` contains the implementation for a number of commonly used optimisers.

2.3.3 Training Loop

To train our network, we perform the following loop through the training data gathered by observing the evolution of the environment:

- `optimiser.zero_grad()` to reset the optimiser's gradients to prepare for a new step.
- $pred_i$ `= model(`x_i`)` to predict the output of the model associated with x_i.
- `loss = loss(`$pred_i$`, `y_i`)` to compute the error associated with the model prediction.
- `loss.backward()` which stores the gradient of the loss with respect to each parameter in the Parameter objects.
- `optimisers.step()` to carry out the update of the parameters and carry out the gradient descent step.

Question 4: Collecting data to train the state-predictor model

We are now going to gather the data necessary to train a DNN state-predictor model that takes as input the current state s_t and action a_t and predicts the next state s_{t+1}. To do so, we need data on how the CartPole environment evolves, which we will collect by rolling out our `simple_policy` across many episodes.

Fill in the `collect_data` function to perform `num_episodes` trajectories with the `simple_policy` and store at each time step t the current state s_t, current action a_t and next state s_{t+1}.

Question 5: Training a state predictor model

The provided notebook provides a skeleton implementation of the training loop for the predictor. Complete it following the training loop described above.

Does the video with the series of states outputted by your model look believable? Does varying the policy rolled out (for example by changing p_{random}) affect the model learned? Can you think in what ways having such a model could be used and in which RL applications it may be particular beneficial or even necessary?

Question 6: Trying different loss functions

The loss function is a crucial choice when training DNNs. We provide you with one alternative loss functions, and an example of how to use the loss functions in torch. Try each of them to see their impact on the learning. Try also with your own loss function.

3 Extensions (optional)

You may find investigating following topics related to this particular task interesting.

Pre-training predictions

How does the visualisation of your dynamics prediction look before and in the very early stages of network training? Is it what you would expect?

Extending the training range

Modify the `ShowCartPolePredictions` class' `step` method to always set `done = False` just before ending and set the termination condition in the simulation loop to stop after a number of time steps (for example 100). This will prevent the visualisation from ending when the episode would normally terminate. What happens to the quality of the dynamics predictions after the point where the state would normally end? Can you explain why this happens?

Now modify the episode termination condition in `collect_data` to also end the episode after 100 time steps and train your network again. Does your model now accurately predict dynamics for states beyond the typical episode range?