Homework 4

16-311: Introduction to Robotics

February 10, 2022

Contents

Learning Objectives **1**

1 Discounted Rewards **2**

2 Path Planning **3**

Programming Set Up **4**

3 Imitation Learning **8**
3.1 Questions to Answer in PDF 10

4 Policy Improvement **11**
4.1 Questions to Answer in PDF 12

Debugging Tips **12**

What To Submit **13**

Learning Objectives

1. Implement a discounted reward policy for a simplified motion planning problem by hand.
2. Think critically about path planning and reward applications.
3. Implement an imitation learning program using a neural net.
4. Observe the effect of policy improvement.

1 Discounted Rewards

In this section, you'll calculate the discounted reward for one run of a robot's exploration through the 'maze' shown in table 1. The goal is marked 'G', and obstacles are marked 'O'. The states are mapped as shown in Table 2. The robot starts in the square marked 'R' and randomly chooses to go in one of four directions [Up, Right, Down, Left] ($[1, 2, 3, 4]$). Table 3 shows the states and actions that the robot took for a single run. The robot receives a reward of -1 for making any step that does not make it reach the goal. If the robot reaches the goal it gets a reward of 0 and stops moving.

			G
R	O		
O	O		O

Table 1: The beginning state of the maze, $t = 0$

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Table 2: The state mapping of the maze

1. Please fill in Table 3 with the rewards and discount rewards for every time step. Take the discount factor, γ to be 0.9.

2. What is the benefit of the discount factor, γ being less than one?

t	s_t	a_t	r_t	g_t
0	4	1		
1	0	1		
2	0	2		
3	1	3		
4	1	3		
5	1	2		
6	2	3		
7	6	2		
8	7	2		
9	7	1		
10	3	x		

Table 3: The state, action, reward, and discount reward table for the robot's run.

2 Path Planning

In this section, you'll answer some basic path planning questions and reason about how to apply reinforcement learning to choose the best path.

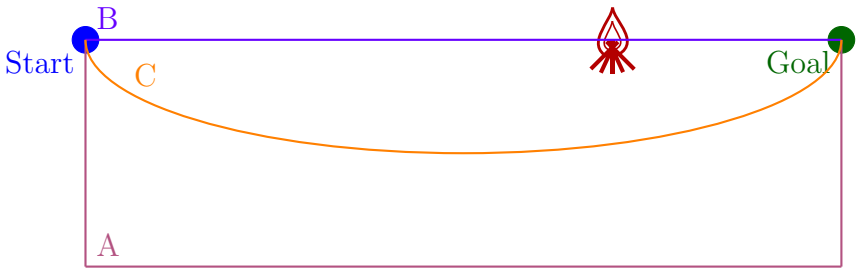

Figure 1: A sample environment with three possible paths.

1. What are the pros and cons of taking path A? List at least one of each.
2. What are the pros and cons of taking path B? List at least one of each.
3. What are the pros and cons of taking path C? List at least one of each.
4. How would you design rewards for the sample environment in Figure 1 such that the robot would learn not to drive through the fire?

Programming Set Up

Introduction

In this homework, you are tasked with designing a neural network to solve the self-balancing robot (SBR) task. You will first train your network to mimic the behavior of a PID contoller using imitation learning. After you get decent performance using imitation learning, you will then use policy improvement, to further train your network.

Note: While there are no restrictions on what functions you use in python/pytorch, you are not required to consult anything outside this document to complete the assignment.

Required code

64 bit Python 3.5 or later

We recommend that you have the following software onto your personal computer. Using Andrew Linux is not recommended, since there are graphical components which may cause issues over ssh.

This assignment was created using Python 3.7. It should function as expected for Python versions 3.5 and above. Pytorch only works with the 64 bit version of Python, so ensure you don't have the 32 bit version.

When installing, make sure you add python to the PATH if prompted.

Package installation

We recommend that you use pip to install the necessary packages. More info can be found here: `https://packaging.python.org/tutorials/installing-packages/`.

If you do not have pip installed, this site provides instructions for Linux users: `https://packaging.python.org/guides/installing-using-linux-tools/` and this article provides instructions for Windows users (and also to add Python to your path): `https://github.com/BurntSushi/nfldb/wiki/Python-&-pip-Windows-installation`.

We need to download **pytorch** and **gym** for this assignment. To install these packages, open a terminal window.

If you are using Mac or Linux, type the following command (replace '3.7' in 'python3.7' with your corresponding python version):

python3.7 -m pip install gym torch torchvision

If you are on Windows, run the following commands (for python3.7):

pip install torch===1.4.0 torchvision===0.5.0 -f https://download.pytorch.org/whl/torch_stable.h
pip3 install gym torchvision

Check here for latest download instructions: `https://pytorch.org/get-started/locally/`

Pytorch is a machine learning library, which is used to create neural networks. Gym is a library which provides environments for creating and testing reinforcement learning algorithms.

To test you have installed everything properly you can run the '$test_install.py' file.

Starter Code

You can download the starter code from the links below:

- `https://www.cs.cmu.edu/~16311/current/homework/hw4/test_install.py`
- `https://www.cs.cmu.edu/~16311/current/homework/hw4/pole.py`
- `https://www.cs.cmu.edu/~16311/current/homework/hw4/pid_solution.py`
- `https://www.cs.cmu.edu/~16311/current/homework/hw4/imitation_starter.py`
- `https://www.cs.cmu.edu/~16311/current/homework/hw4/policy_improvement_starter.py`

The starter code contains 5 files. You will edit 2 of the files, but will not edit 'pole.py', 'test_install.py', or 'pid_solution.py'. You can run 'python test_install.py' to ensure you installed your packages correctly. You can make sure the self-balancing robot (SBR) environment works by running 'pole.py' in the same manner.

At this point, everything should be set up. Note that you will not yet be able to run any of the starter codes yet.

Given Files

'pole.py' (You do not need to edit this file)

This file contains the code which runs the SBR simulator. The simulator is accessed via the environment. In this file, there is one class, called 'CartPole'. To use this simulator in your code, you can load it into another file with the line 'from pole import CartPole'. To see a short example on how to use the simulator, scroll down to the bottom of 'pole.py'.

'pole.py' has 2 methods which you will care about/use.

'CartPole.reset()': This method resets the simulator to a random starting configuration. It returns the new state corresponding to this configuration. You can call this function any time, but must call it if you want to continue a new episode after 'step' returns 'True' for 'done'.

'CartPole.step(action)': This method performs one step of simulation. The parameter 'action' is a 'float' representing the desired motor velocity. You can pass any value, but the motor is limited to between -70 and 70. This method returns 3 values: '(observation, done, reward)'. 'observation' is the sensor readings of the robot. This would be of the form (left_sensor_value, right_sensor_value, motor_encoder_value). Note that this does not capture the entire SBR state, since there is no velocity in this tuple. 'done' is a 'bool', either 'True' or 'False', representing if the SBR has entered a terminal step (either after 800 steps, or if it falls over). 'reward' is the reward signal from the environment. It will be '1' for each time step the robot has not fallen over.

You will repeatedly call this function to simulate the episode.

It also has 2 attributes which you may want to change:

'CartPole.rendering' this is a 'bool', whose 'True' value corresponds to rendering the episode (as shown in videos below. 'False' suppresses render output, but simulates the SBR much faster. (You may want 'rendering' to be false when training)

'CartPole.harder' this is how you control the difficulty of the SBR. By default, it is set to 'False'. Solving the environment with this set to 'False' is all that is required to get full points on "PID Implementation" and "Imitation Learning". However, to receive full points on "Policy Improvement", this must be set to 'True'.

For example, these attributes could be modified in the code as follows:

```
from pole import CartPole
SBR = CartPole()
SBR.rendering = True # enables rendering
SBR.harder = True # enables harder mode
SBR.rendering = False # disables rendering
```

```
SBR.harder = False # disables harder mode
```

'pid_solution.py'

This is the solution code file for the expert PID implementation your imitation learning will learn from. It has a class, 'Pid'. Check above for details on the format of 'state' and 'action'. You should be able to run it with python without any command line arguments (i.e. 'python pid_starter.py') and see the cartpole stand up.

'imitation_starter.py'

This is the starter code for your imitation learning solution. It has a few helpful functions, as well as the 'Net' class, and starter code for training your network.

'log(...)' Use this like a print function, instead of the actual print function. This is a drop-in replacement for python's print, but this also will log anything that is printed to the file 'LOG_FILE', to save your training results.

'SpeedMemory' Helper class which computes and stores the $\frac{\Delta error}{\Delta time}$. Initialize this like you would any python object.

'nearest_action(actions, desired_action)' This function should take in a list of possible actions (i.e. a list of floats), and a 'desired_action' (a float), and returns the index of the action in 'actions' which has the smallest absolute difference with 'desired_action'.

'Net' This class is your main neural network class. You will fill out the methods '__init__' and 'forward'.

'__init__' This is the initialization method for the class. 'in_size' corresponds to the size of the input of your net, 'out_size' corresponds to the size of the output of your net (i.e. layer sizes). 'hidden_size' corresponds to the number of nodes in your neural network's hidden layers.

'forward(x)' This method computes the "forward pass" of your neural network (You have to fill this out), given some input 'x'.

'save(file_path)' This saves your neural network to a file (you don't need to edit this method)

'load(file_path)' This loads a file into this neural network object. (you don't need to edit this method).

'policy_improvement_starter.py'

This is the starter code for your policy improvement solution. It has some of the same helper functions as are in 'imitation_starter.py', which are exactly the same.

Note that this file also has 'IMITATE_MODEL_FILE'. This should correspond to the path to your saved, fully-trained imitation learning network model.

'improve_policy(agent, discounted_rewards, discount_factor)' This function performs a step of policy improvement for a given 'agent' and discounted rewards for an episode. In the docstring, a more detailed description of the parameters are given. Note that 'agent' is simply the object corresponding to your neural network.

3 Imitation Learning

In this section, you will create and train a neural network to "imitate" a PID solution for the Cartpole problem, and complete the same task. To do this, you will be editing the 'imitation_starter.py' file. Below is a summary of the steps for you to complete. They are then listed in more detail below this.

Step 1: Create a list of actions that your neural net can choose from.

Step 2: Design your Neural Net

Step 3: Find the closest expert action to the possible actions designed in step 1.

Step 4: Answer the question about the Neural Net listed in 3.1

The following points describe in more detail how to complete the programming tasks for this section.

- **Step 1 in imitate_starter.py:** Select the values for the actions which your neural network can output. First, pick the number of actions you want to output. (Anything between 15 and 31 should work, but you can play with this). You may want to choose an odd number, so you can have it of the form '[-c, -b, -a, 0, a, b, c]'. Then, pick out the values for the actions. To pick out what values seem good, you can use intuition, but you could also log the output of the PID solution during a run, and then analyze the most used forcing values in the PID.

- **Step 4 in imitate_starter.py:** Designing your neural network.

  In the '**__init__**' method, you will design your neural network architecture. In torch (ML library), you define a neural network through the notion of a computation graph. Note that the parameters of the linear combination are automatically tracked by the library.

There are two main types of layers you will be interested in:

'torch.nn.Linear(in_size,out_size)' — This is a layer of linear combination nodes (i.e. $Ax + b$, where A is 'in_size'x'out_size', and b is 'out_size'x1).

'torch.nn.ReLU()' — This is a layer of ReLU nodes, which you will use as your activation function for your layers.

So, to create a layer, you would write code like

```
self.layer1 = torch.nn.Linear(in_size, out_size)
self.relu1 = torch.nn.ReLU()
```

You will then chain layers together to create your network with your desired number of layers. The output size of a previous layer should be the input size of a later layer. Additionally, the last layer should not have a ReLU, and should be immediately before the 'softmax' layer which is given in the starter code.

In the **'forward'** method, you will define how the computation is done to compute the forward pass. The machine learning library automatically performs backpropagation for you, based on how you define the forward pass, so you do not have to worry about that!

To calculate the value through a layer, simply call the layer on its input. It will return its output. For example, if you defined 1 layer as described above in the first code snippet, you could calculate its output as follows:

```
out = self.layer1(x)
out = self.relu1(out)
```

At the very end of your network, you should see where 'out' is passed into the softmax layer. This simply normalizes your output into a probability distribution. You may see that your probabilities are negative! don't worry about this — for optimization reasons, the output is actually the log probabilities (so they should be negative). Since log is monotonic, though, you can still use comparison to see which probabilities are greatest.

Finally, replace the 'None's in the initialization of the agent with the proper values.

- **Step 3:** Now you must find the closest action from your previously defined set of actions to the expert action given by the PID solution. We recommend you use (and complete) the defined **'nearest_action(actions, desired_action)'** function. You should write this function such that it takes in a list of possible

actions (i.e. a list of floats), and a ‘desired_action‘ (a float), and returns the index of the action in ‘actions‘ which has the smallest absolute difference with ‘desired_action‘.

- Now run your code. You can do this simply by running the python file from a terminal. If everything is set up, you should see the iteration and loss printed to the screen. This should take between 30 seconds and 5 minutes (If it takes longer, you probably need to change other things). Eventually, once performance is sufficient, you should see the rendering appear as well. The loss should start around 3, and as you train it should decrease to close to 0.3. The lower your loss, the better! By default, the render will start when the average loss is less than 0.3. You should see good performance by this point. The code will automatically save your network and exit training once the loss is low enough. You can change the thresholds for saving and exiting, if you feel the need.

3.1 Questions to Answer in PDF

Please answer the following questions in you pdf solution.

1. What is ‘in_size’? What is ‘out_size’? How many layers did you use? (Recommended: 3-5) How wide is each layer be? (Recommended: 16-64 nodes)

2. What is the difference between ‘action’ and ‘target’?

3. Change the learning rate variable in the code. Try both increasing and decreasing it. What happens to the model as you do this. How does it effect loss? How much did you need to change it by before seeing significant change?

4. Once your agent is fully trained, take a video of a run of the SBR balancing, and include the link in your document. If you cannot get your learned agent to balance for an entire run, show an example of a good run for your agent.

   To record the video, you can modify your ‘imitate_solution.py’ file as follows: below the line which says ‘agent = Net(. . .)’, add the line ‘agent.load(MODEL_FILE)’, below the line which says ‘cartpole = CartPole()’, add the line ‘cartpole.rendering = True’. Then, when you rerun the file, it will immediately render an episode with your trained agent (provided you have already successfully trained, and the model was saved.)

   You can expect to have a well-trained robot in less than 50k iterations (10-15 minutes), with noticeable improvement at 10k iterations. Training a neural

network to successfully complete an entire run will award you with full points for this section. If you are having trouble getting your neural network to finish an entire run, it is possible to move onto the next section (policy improvement) with only a partially good network here.

4 Policy Improvement

In this step, we will use policy improvement to improve the policy you trained in the 'imitation learning' section. This will allow the SBR to perform well, even if 'harder = True' (you are actually required to have 'harder = True' enabled for this portion). Since you are using the same neural network as the previous section, the starter code automatically imports the 'Net' class into this file.

If you remember from lecture, there are two challenges you will need to do to perform policy improvement. Firstly, you will need to calculate the discounted rewards for each time step of an episode. Then, these discounted rewards will be used to update the parameters of your neural network to improve the policy. The policy improvement step has been given to you, and is detailed in the function 'improve_policy' within the 'improve_policy_starter.py' file. In fact, the 'improve_policy' function will even finish your training once the agent is fully trained (when the agent successfully completes 10 episodes without falling over).

To refresh your memory, an episode will be a sequence of tuples of the form $(s_1, a_1, r_1), (s_2, a_2, r_2), (s_3, a_3, r_3), ...(s_n, a_n, r_n)$. You will need to calculate the discounted reward (g_t) for each time step t. You can do that with this equation, where γ is your discount factor (recommended: $0.8 < \gamma < 1$) $g_T = \sum_{t=T}^{n} \gamma^{t-T} r_t$. Refer to the reinforcement learning slides for more details on this part.

Complete the following steps and then run your code. The structure and expected output are explained in the 'Run the code' bullet.

- **Step 1 in policy_improvement_starter.py:** Copy over the parameters for initializing your net and the actions.
- **Step 2 in policy_improvement_starter.py:** Calculate the discounted reward.
- **Run the code.** There will then be an outer loop. Each run of the outer loop represents one episode. In each loop, the code will:
    1. Reset the environment

2. Run an entire episode, saving the 'state', the 'action', and the corresponding 'reward' received for that action (note that the 'reward' corresponding to 'action' is the 'reward' received from the environment when calling 'step(action)'. The 'action' you choose will be the action of highest probability from the neural net. Be sure not to confuse the new 'state' (received with the previous state!).
3. Compute the discounted rewards for each step. After you compute the discounted rewards, you should have a list of tuples of the form $(g_1, s_1, a_1, r_1), (g_2, s_2, a_2, r_2), (g_3, s$
4. You will pass this list, into the 'improve_policy' function. This calculates the correct gradients for the update rule, as given in the slides: $\nabla\theta = \alpha((G_t - B)\nabla_\theta(\pi(\theta, s)[a]))$
5. Then, perform one step of optimization. Repeat until 'improve_policy' terminates your program.

4.1 Questions to Answer in PDF

Now answer the following questions in you pdf.

1. Take a video of your system. Once your agent is fully trained (successfully finishing episodes for at least 10 times in a row), you will have a new file, as defined by 'MODEL_FILE' in 'policy_improvement.py'. This contains the fully trained network model. You can load this and run it in the environment by using the same code you used to render an episode for the imitation solution video. Make sure you run this with 'harder = True' and 'rendering = True', and record the output. Again if you cannot manage to fully train you Cartpole, take a video of a good run, or as many good runs in a row as your model manages.

2. What does changing the parameter 'harder' to 'True' do? You may need to check 'pole.py' for this. Comment on your SBR's success with this parameter turned on and off.

Debugging Tips

- You can get math functions (such as 'sqrt', 'sin', 'asin', etc. by putting 'from math import *' at the top. If you need more linear algebra tools, you can also do 'import numpy as np', referring to the NumPy wiki on Google.

- If you get a cryptic pytorch error, make sure you are passing tensors, not python lists or tuples into your neural network.
- Additionally, if your pytorch error is complaining about a 'size mismatch, m1: [1x...]', try adding an extra '[]' when you create your tensor. E.g. if you create your state tensor like:
  'state = torch.tensor([2,3,5,1])'
  try instead
  'state = torch.tensor([[2,3,5,1]])'
  This is because pytorch expects a 'batch' of data, and is expecting to evaluate possibly many different inputs at the same time.
- To get the argmax of a tensor 'tens', you can use the lines
  index = tens.argmax(1).item() print(index) (This will print the index of the maximum index of 'tens')
- It is very easy to make simple mistakes which don't immediately cause apparent bugs. If it looks like it's somewhat working, double check each step, using print debugging or your preferred debugging method to search for bugs.

What To Submit

Submissions are due on Autolab/Gradescope by the date specified in the Syllabus.

1. Create a .pdf file with the written answers ALL THE SECTIONS named hw4.pdf and submit to Gradescope.
2. Submit one .zip file to Autolab with ALL code, logs, and model files. In this file, you should have all the code you edited and used for this assignment, so that it could be unzipped and run. Finally, include any relevant logs (i.e. the logs created when you trained each portion of your network). Make sure your logs only include the relevant training output (The logs should be less than 100 KB each). This means you should clear the logs before running the code for the last time.