HW3: Deep Q-Learning for Object Grasping

CS4910

Spring 2022

The goal of the assignment is to use deep q-learning to solve the top-down grasping task that we previously solved with supervised learning. Because q-learning requires a discrete action space, we use a fully convolutional network to predict the q-values for each pixel in an image. This is called a pixel-wise action space and is commonly used in literature for solving top-down grasping tasks. By completing this assignment you will gain practice manipulating tensors, create a more complex convolutional architecture called UNet, and implement a more advanced q-learning update equation called Double Q-Learning.

This assignment is **due February 22 by 3:25pm**. To submit your assignment, send an email with a zipped HW3 folder (with implemented code and ipynb file) to klee.d@northeastern.edu.

1 Top-Down Grasping Environment

The top-down grasping environment ('grasping_env.py') is very similar to what you implemented in HW1. The main difference is that it now follows the `gym.Env` standard, with a reset and step function. You will notice that the action space is a discrete space of pixel locations (called pixel-wise action space). Given a pixel location as an action, the corresponding position in xy-space is calculated and a top-down grasp is performed at said position. Notice that we are not predicting orientation of the grasp, since this would unnecessarily complicate the implementation. As you work through the implementation of the q-network, you may have some ideas for how this could be extended to include discrete orientations.

You do not have to do anything for this section, but it may be useful to look through the environment implementation to gain understanding. To watch a random policy, you can run: 'python grasping_env.py'.

2 Implement Fully-convolutional Deep Q-Network [10 pts]

Here, you will implement the q-network for predicting q-values for the pixel-wise action space. You will implement a UNet-style architecture, that is composed of a down-sampling portion followed by an up-sampling portion. Features from the down-sampling process are concatenated to features produced during the up-sampling process; this allows the network to process information at multiple resolutions. The down-sampling portion will be familiar to you, and is achieved using `nn.Conv2d` layers with `stride=2`. The up-sampling portion will be performed using `nn.ConvTranspose2d` layers with `stride=2`. A full guide to the layers used is in Figure **??**. You know you have implemented it correctly if the height and width of the output equals the height and width of the input. The output is called a q-map since it is a 2d-array of q-values.

For this section, instantiate the layers in `PixelWiseQNetwork.__init__` and implement `PixelWiseQNetwork.forward` in the file 'networks.py'. You can test the implementation by running 'python networks.py'.

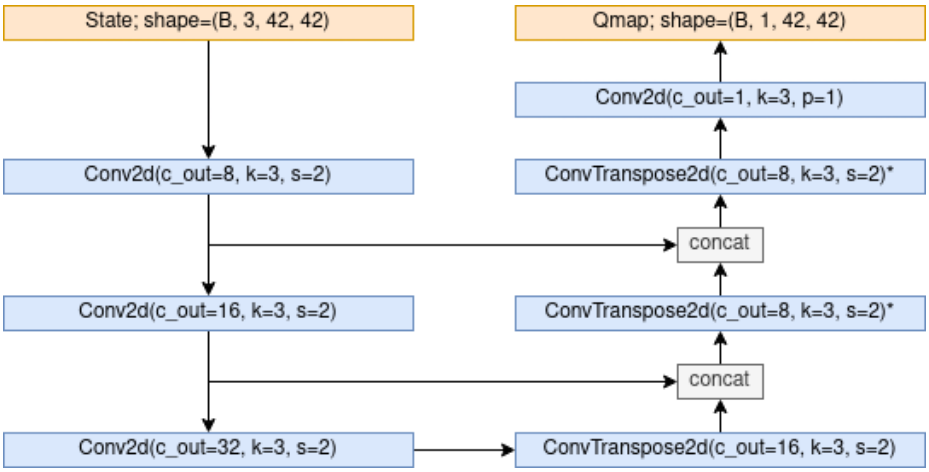

Figure 1: UNet architecture details (now you might see why its called "U"-Net). Make sure to place ReLU activation functions after all layers (except final layer!). Do not use BatchNorm here. For some ConvTranspose2d layers (denoted with asterisk), you will need to set `output_padding=1` to get the shape to work out. Remember that the concatenating will be along the channel dimension.

3 Finish Agent Implementation [10 pts]

Most of the code for training the agent has already been implemented. However, you still need to implement the methods: `DQNAgent.policy`, `DQNAgent.select_action` and `DQNAgent.prepare_batch` in the file 'agent.py'. The first two you have seen during class for the Reacher implementation. The last method will be similar to what you did for the data loader during supervised learning: converting numpy arrays to tensors so they can be sent to the network.

Once you have implemented these methods you should be able to run 'python agent.py' without errors (errors will occur within a few steps, so you can run it locally to debug).

4 Implement Double DQN Update Equation [10 pts]

In class we went over the q-learning update equation:

$$Q_\theta(s,a) \leftarrow Q_\theta(s,a) + \alpha\big(r + \gamma \max_{a'} Q_{\theta_t}(s',a') - Q_\theta(s,a)\big)$$

The target q-value is determined using the maximum of the target q-network at the next state. However, this can introduce a bias in the target q-value that overestimates its value. To overcome this, we can use the double q-learning update equation:

$$Q_\theta(s,a) \leftarrow Q_\theta(s,a) + \alpha\big(r + \gamma Q_{\theta_t}(s', \arg\max_{a'} Q_\theta(s',a')) - Q_\theta(s,a)\big)$$

Rather than using the maximum value of the target q-network, we use the target network's estimate of the action predicted by the online q-network. Empirically, this reduces overestimation of q-values and improves learning.

Implement the double q-learning update rule in `DQNAgent.optimize` of 'agent.py'. Double check the shapes are correct. In the next section, you will compare the performance to using the standard q-learning update equation.

5 Training and debugging [10 pts]

It is now time to train the DQN agent. Open the collab notebook and follow what is written in the cells. Spend some time playing around with the hyperparameters and write your findings in the cell title "Observations during hyperparameter tuning". Be as descriptive as possible with what you find (i.e. did it change speed of learning, did it introduce instability, etc). If you only notice a drop in performance, that is fine.

Once you are satisfied with your hyperparameters, train the Agent with the standard update rule in the first cell, and then with the double q-learning update rule in the following cell (use the same hyperparameters for consistency!). Which did better?

6 Thinking it through [10 pts]

For supervised learning, we could report prediction accuracy based on evaluation data. However, to get probability of grasp success, we had to run the trained method in the simulator. Reinforcement learning is slightly different. What can we say about the probability of grasp success based on the metrics we plotted during training? Write one to two sentences in the Colab notebook. Consider the influence of epsilon during training, and whether the 'success rate' plot reflects probability of grasp success. Do you have any thoughts on whether this policy would work on the real robot (assuming I set up the camera in same position and used similarly shaped object)?

When you are done with the Colab notebook, select 'File → Download → Download .ipynb' and download the notebook file. Include it in your zip file when you submit the assignment