HW2: Supervised Learning of Top Down Grasps

CS4910

Spring 2022

The goal of this assignment is to train a neural network to predict successful top-down grasps. Using the simulator setup described in the previous assignment, a large data set has been collected of attempted grasps. Now, it is up to you to develop a convolutional neural network to fit this data. You will gain experience developing torch.nn.Module's, selecting proper loss functions, and adding data augmentation. If you are stuck, check out the PyTorch documentation or reach out on Piazza.

This assignment is **due February 8 by 3:25pm**. To submit your assignment, send an email with a zipped HW2 folder (with implemented code) to klee.d@northeastern.edu.

0 Data Collection

Before any learning can happen, we must collect a dataset. In 'grasping_env.py', a Pybullet simulator has been set up to perform top-down grasps. It differs from what you implemented in your last assignment in two ways: (1) there is no randomized texture swapping (since this makes learning much slower); (2) the grasping function has been accelerated by teleporting joints when possible. In 'data_collection.py', I have implemented a function to collect a dataset of successful grasps (you can try it out yourself but it can be quite slow even with multiple processes in parallel). Notice that the dataset is saved as an HDF5 file. I have collected two datasets for you: a training dataset of 25k successful grasps and a testing dataset of 1k successful grasps.

You do not need to complete anything for this section, but I wanted to introduce the data collection since it may be helpful to refer to these files for the next part.

1 Dataset and DataLoader [5 pts]

We need a convenient way to access the data for training. Specifically, we need the data as float tensors and split randomly into batches for training. PyTorch uses the `Dataset` and `DataLoader` classes from `torch.utils.data` to do this.

Complete the implement of the `SuccessfulGraspDataset` class in 'dataset.py.' Refer to the docstrings for guidance. Do not forget to transform the sample if `self.transform` is specified. If you have it implemented correctly, then you should be able to call the function `visualize_dataset` without errors. I have provided a small dataset 'mini_dataset.hdf5' so you can test on this file without needing to download the actual datasets (we will do that using Colab for training).

2 Creating Neural Network [10 pts]

Now, you will implement `GraspPredictorNetwork` in 'networks.py.' This network takes images as input and outputs the action as (x,y,θ). The specifications for the network are described in Figure 1. You can create the layers as you like (in terms of using Sequential, etc.). For now, use `nn.MSELoss()` as the loss function. You may wish to debug by running the training script: 'python train.py' and ensuring there are no errors and all the tensor shapes make sense. The training script is set up to run on the mini dataset so you should be able to run it quickly on your laptop without a GPU.

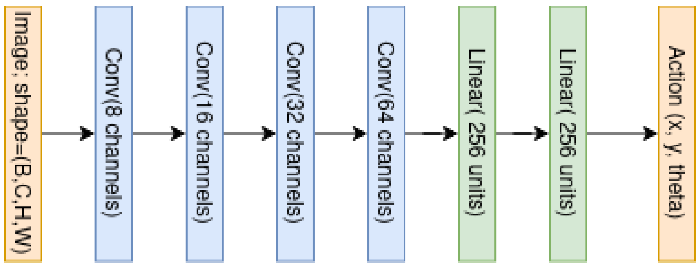

Figure 1: Network architecture. There are four convolutional layers with the number of channels at each layer specified above. Each convolutional layer should use a kernel size of 3, a stride of 2 and padding of 1. Convolutional layers should be followed by a ReLU and Batch Norm. There are two linear layers with 256 hidden units (with ReLU's between them). The input to the network are images of shape (B, 3, 42, 42). The output is the action, (x,y,θ), of shape (B,3). Any activation functions on the output are up to you (but may not be necessary).

3 Training on Full Dataset [5 pts]

Now it is time to use the Colab notebook to train on GPU's. Follow the instructions in the notebook to enable GPU mode, download the dataset files, upload your python files, and train your network. The network should only take about 3 seconds per epoch if you enable GPU mode properly. After training, the model will get saved and two plots will be shown: (1) the loss curves on the train and test data; (2) example predictions on the test data.

You can play around with the hyperparameters like learning rate and batch size, but it is not necessary.

4 Refining Loss Function [10 pts]

The loss curve probably does not look great right now. One reason why is the MSE loss function, which treats position and rotation the same, even though they have different magnitudes. For instance, 1 cm is penalized as much as 0.6 degrees, even though the workspace is only 10 cm wide! Thus, the model is placing way too much emphasis on predicting rotation. We can correct this by weighting the position error more than the rotation error.

In `GraspPredictorNetwork`, swap out `nn.MSELoss` for `WeightedMSELoss`. Test out different weight values to reduce overfitting.

5 Data Augmentation [10 pts]

Another way to improve performance on the test data is to add data augmentation. We must be careful here since certain augmentations will require changing both the image and the action. For instance, if you shift the image by several pixels, then the ground truth action's position must be shifted accordingly. For this assignment, you will implement data augmentation in the form a horizontal image reflection in the `RandomReflectionTransform` class of 'dataset.py'. The transformation should only occur 50% of the time (if it occurred every time then we would not be augmenting, but replacing). Figuring out the corresponding change to the action is left to you (hint: the vertical axis of the image is the x-axis of the simulator). To help you out, I show an example of it working in Figure 2, and you can test it by running the function 'test_reflection_augmentation' on the mini dataset.

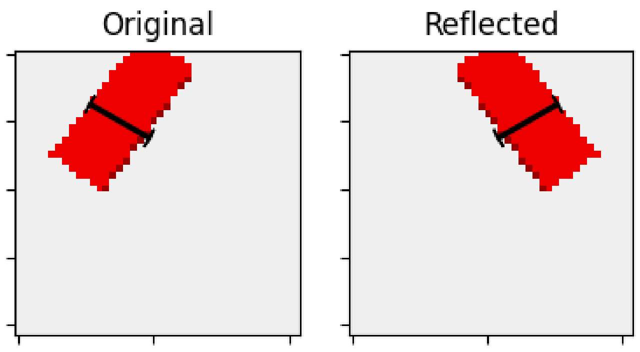

Figure 2: Example of `ReflectionTrasform`. The image is reflected horizontally, and the actions position and rotation changes accordingly.

Once you have it working, go back to your Colab notebook and try training with the `use_augmentation` flag set to True. The improvement in test loss will be marginal, more augmentations would be needed to significantly reduce overfitting.

6 Evaluating Model and Potential Next Steps [10 pts]

You now have a trained model that works reasonably well. It is difficult to say how the loss here translates to actual grasping performance. To evaluate performance objectively, we can test predictions in the simulator. On your laptop, run the script:

```
python grasping_env.py --model-path MODEL_PATH
```

where MODEL_PATH is the path to your model file.

You should get around 60% performance. This is quite impressive, but could be improved with some effort. Propose one additional data augmentation that could be introduced to improve performance (make sure to discuss how both the image and the action would be adjusted). Suggest one change to the architecture of the neural network in terms of the input or output formulation that could improve learning. Record your thoughts in 'improvements.txt' (just a sentence or two on each).