ML Capstone Project Report

Jiaqin (Lucy) Sun

December 2022

1 Introduction

In this project, I will use supervised learning to predict the state of a 4-fingered robot's hand given 3 angles and a depth images: RGBD. The purpose of this project is to output the predictions with small mean squared error (MSE) loss.

2 Method

To load data, build model, and save outputs, I used Python and applied the following libraries: os, torch, numpy, cv2 and pickle, nn from torch, Dataset and DataLoader from torch.utils.data, datasets and transforms from torchvision, F from nn.functional, and optim.

```
import os
import torch
from torch import nn
from torch.utils.data import Dataset, DataLoader
from torchvision import datasets, transforms
from torchvision.transforms import ToTensor
import numpy as np
import cv2
import pickle as pkl

import torch.nn.functional as F
import torch.optim as optim
```

I made sure all my data is saved into the right device, either CPU or GPU.

```
device = "cuda" if torch.cuda.is_available() else "cpu"
```

I used LazyLoadDataset to obtain my dataset from my kaggle input folder and transform img0, img1, img2, which are the RGB images of each data. Within LazyLoadDataset, I read the RGBD images, normalized them, and transformed them. The output for that class will be in form of (data, target), with data containing a length of 4–3 RGB image, 1 depth, and 1 field ID.

The dataset later was loaded into the dataloader with a 64 batch-size. I also checked the size of each output item and printed out the first depth image.

```
class LazyLoadDataset(Dataset):
    def __init__(self, path, train=True, transform=None):
        self.transform = transform
        path = path + ("train/" if train else "test/")
        self.pathX = path + "X/"
        self.pathY = path + "Y/"
        self.data = os.listdir(self.pathX)

    def __getitem__(self, idx):
        f = self.data[idx]

        #X
        #read rgb images
        img0 = cv2.imread(self.pathX + f + "/rgb/0.png")
        img1 = cv2.imread(self.pathX + f + "/rgb/1.png")
        img2 = cv2.imread(self.pathX + f + "/rgb/2.png")
        #read depth images
        depth = np.load(self.pathX + f + "/depth.npy")

        #normalize image
        img0=cv2.normalize(img0, None, 0, 1, cv2.NORM_MINMAX)
        img1=cv2.normalize(img1, None, 0, 1, cv2.NORM_MINMAX)
        img2=cv2.normalize(img2, None, 0, 1, cv2.NORM_MINMAX)
        depth=cv2.normalize(depth/1000, 0, 1, cv2.NORM_MINMAX)

        #read field ID
        field_id = pkl.load(open(self.pathX+f+"/field_id.pkl", "rb"))

        #transform
        if self.transform is not None:
            img0 = self.transform(img0)
            img1 = self.transform(img1)
            img2 = self.transform(img2)
            #depth = self.transform(depth)
            #print("transformed")

        #Y
        Y = np.load(self.pathY+f+".npy")

        return (img0, img1, img2, depth, field_id), Y

    def __len__(self):
        return len(self.data)
```

```
transform=transforms.Compose([transforms.ToTensor()])
dataset= LazyLoadDataset("/kaggle/input/lazydata/lazydata/", transform = transform)
```

```
batch_size = 64
train_dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,shuffle=True)
```

When training data, I stacked the RGB images together, and the resulting stack has a shape of [64, 9, 224, 224]. I then stacked it with the depth data. The resulting data with shape [64, 12, 224, 224] would later be passed into my model. I made sure to erase all gradients before computing new ones, and I used MSELoss to calculate my loss function.

There are many ways of implementing the neural network. I modified CNN model and added 4 Conv2d layers, 3 MaxPool2d layers, and 2 linear layers. My input size was of [12, 224, 224]. The first shape becomes the number of conv features, and the last 2 shapes reduces by (kernel size - 1) for each Conv2d layer. During Maxpool2d, the last 2 shapes reduce by the number of strides and are divided by the kernel size. After all, The resulting shape when being passed into the linear layer is [number conv features, 4, 4]. And the linear layer returns the output size, which is (1, 12).

```
class CNN(nn.Module):
    def __init__(self, input_size, conv_feature, fc_feature, output_size):
        super(CNN, self).__init__()

        self.relu = nn.ReLU()
        self.conv1 = nn.Conv2d(input_size, conv_feature, kernel_size=5)  #12 224 224
        self.pool1 = nn.MaxPool2d(kernel_size=3, stride = 4)  #256 220 220  (=224-5+1)
        self.conv2 = nn.Conv2d(conv_feature, conv_feature, kernel_size=3)  #256 72 72  (=(220-4)/3)
        self.pool2 = nn.MaxPool2d(kernel_size=2)   #256 70 70
        self.conv3 = nn.Conv2d(conv_feature, conv_feature, kernel_size=3)  #256 35 35
        self.pool3 = nn.MaxPool2d(kernel_size=3)   #256 33 33
        self.conv4 = nn.Conv2d(conv_feature, conv_feature, kernel_size=4)  #256 11 11

        self.fc1 = nn.Linear(conv_feature*5*5, fc_feature)
        self.drop = nn.Dropout(0.5)
        self.fc2 = nn.Linear(fc_feature, output_size)

    def forward(self, x):
        x = self.pool1(self.relu(self.conv1(x)))
        x = self.pool2(self.relu(self.conv2(x)))
        x = self.pool3(self.relu(self.conv3(x)))
        x = self.conv4(x)
        #x = torch.flatten(x, 1)
        x = x.view(-1, 256*5*5)
        x = F.relu(self.fc1(x))
        x = self.drop(x)
        x = self.fc2(x)
        return x
```

Next, we build the model using train data. I call the model with 12 input channels, 256 conv features, 50 fc features, and request an output size of 12. I've also lr_scheduler to have a learning rate for for every step-size epoch, and I ran the model for 10 epoches.

Last but not least, I saved my model, predicted the target for the test data, and outputted a csv file containing the prediction.

3 Experimental Results

```
# Training settings
conv_features = 256 # number of feature maps
input_channels = 12 # number of input channels
fc_features = 50
output_size = 12

# create CNN model
model_cnn = CNN(input_channels, conv_features, fc_features, output_size) # create CNN model


model_cnn.to(device)
# use Adam with learning rate 0.001
optimizer = optim.Adam(params = model_cnn.parameters(), lr = 0.001)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size = 3, gamma = 0.8)

print('Number of parameters: {}'.format(get_n_params(model_cnn)))

test_accuracy = []
for epoch in range(0, 10):
    train(epoch, model_cnn, optimizer)
    scheduler.step()
```

```
Number of parameters: 2626710
Train Epoch: 0 [0/3396 (0%)]    Loss: 0.009956
Train Epoch: 0 [1280/3396 (37%)]        Loss: 0.004820
Train Epoch: 0 [2560/3396 (74%)]        Loss: 0.002314
Train Epoch: 1 [0/3396 (0%)]    Loss: 0.002290
Train Epoch: 1 [1280/3396 (37%)]        Loss: 0.001737
Train Epoch: 1 [2560/3396 (74%)]        Loss: 0.001306
Train Epoch: 2 [0/3396 (0%)]    Loss: 0.001435
Train Epoch: 2 [1280/3396 (37%)]        Loss: 0.001022
Train Epoch: 2 [2560/3396 (74%)]        Loss: 0.001156
Train Epoch: 3 [0/3396 (0%)]    Loss: 0.000832
Train Epoch: 3 [1280/3396 (37%)]        Loss: 0.000911
Train Epoch: 3 [2560/3396 (74%)]        Loss: 0.000916
Train Epoch: 4 [0/3396 (0%)]    Loss: 0.000968
Train Epoch: 4 [1280/3396 (37%)]        Loss: 0.000859
Train Epoch: 4 [2560/3396 (74%)]        Loss: 0.000875
Train Epoch: 5 [0/3396 (0%)]    Loss: 0.000761
Train Epoch: 5 [1280/3396 (37%)]        Loss: 0.000730
Train Epoch: 5 [2560/3396 (74%)]        Loss: 0.000823
Train Epoch: 6 [0/3396 (0%)]    Loss: 0.000635
Train Epoch: 6 [1280/3396 (37%)]        Loss: 0.000747
Train Epoch: 6 [2560/3396 (74%)]        Loss: 0.000593
Train Epoch: 7 [0/3396 (0%)]    Loss: 0.000721
Train Epoch: 7 [1280/3396 (37%)]        Loss: 0.000707
Train Epoch: 7 [2560/3396 (74%)]        Loss: 0.000659
Train Epoch: 8 [0/3396 (0%)]    Loss: 0.000616
Train Epoch: 8 [1280/3396 (37%)]        Loss: 0.000591
Train Epoch: 8 [2560/3396 (74%)]        Loss: 0.000537
Train Epoch: 9 [0/3396 (0%)]    Loss: 0.000552
Train Epoch: 9 [1280/3396 (37%)]        Loss: 0.000647
Train Epoch: 9 [2560/3396 (74%)]        Loss: 0.000566
```

The MSE Loss for my train data looks good over all. It starts with 0.01 and decreases to around 0.000566. The decreasing speed slows down after 5 epoches, and when it gets to the 9th to 10th epoches, the loss is around 0.0005 to 0.0007. The lowest lost is 0.000537, and it happened on the 9th epoch.

4 Discussion

I tried out different optimizers, loss functions, learning rate, model and layers, number of conv features, and ran for different number of epoches.

I experienced with several different optimizers: SGD, Adam, AdamW, AdadeIta, and Adagrad. Adam tends to give me an output with a smaller MSE loss, so I ended up using optim.Adam. Similarily, for loss functions, I tried out cross entropy loss and MSE loss. They both measure the differences between actual probability and predicted probability. I chose nn.MSELoss as my loss function

mainly because this is the measurement the score is based on.

For my neural network, I examined between from 2 Conv2d and 2 MaxPool2d and 2 linear layers (a total of 6 layers) to 4 Conv2d, 3 MaxPool2d, and 2 linear layers (9 layers total). The more layers, the longer the running time. However, the loss seems to decrease as I add in more layers. Likewise, a greater number of conv features would output a higher accuracy. However, it also represents greater memory required and a longer running time. I run my code with a laptop, and ended up using Kaggle's GPU with 16 GB memory constraint. I noticed that a 16 GB memory would only allow me to run up to 300 conv features. Perhaps in the future with more advanced devices, I will be able to produce higher accuracy.

The learning rate seems to vary a lot. So far I can not tell any major correlations between learning rate and loss. For a different model, a smaller learning rate could result in a greater or smaller loss.

The more epoches, the longer running time will be. Overall, for the first few epoches, the loss decreases a lot. That decreasing slows down, and the loss value fluctuates after 5-10 epoches. This is how I decided to use 10 epoches.

5 Future Work

In the future, I would consider trying out some of the prebuilt models such as resnet50 and resnet 101, or add conv layers on these models to see if there is a higher accuracy. If possible, I would like to try more and attempt to understand if there is any correlation between learning rate and loss. Also, it would be good to learn what are the causes of overfitting data as well. In addition, due to the device I used to code, I experienced some memory and system constraints. Therefore, it would definitely be worth trying if in the future we can run and modify these prebuilt models on a desktop that has a cpu.