Deep Learning 2020 **(Due: [November 25, 2020])**

Submission Assignment #[3]

Instructor: Jakub Tomczak *Name:* [Qinghe Gao], *Netid:* [qgo500]

1 Problem statement

Convolutional neural networks(CNNs) is a powerful tool to do image classification, modelling and so on. In this assignment, we implement a CNN to do image classification on MNIST dataset. Furthermore, we tune all parameters and test statistical significance. Finally, we do the final test prediction on test dataset.

2 Methodology

2.1 Basic structure

CNNs usually have many layers which are responsible for different usage. Convolutional layer is the most important part in CNNs. The basic method is to apply filter to input and repeat application of the same filter to get feature map, which contains detected features: corners, edges and so on. Following layer is activation layer, which is usually a non-linear function to transform the results after convolution. Then, pooling layer is usually used before next convolutional layer. Its function is to reduce spatial size of representation and control overfitting. Finally, we have fully-connected layer, which usually locates at the end of neural network. (CNN)

2.2 Adam optimizer

Adam optimizer is one of gradient descent method, which is quite popular now in deep learning. It uses the first-order moment estimation and the second-order moment estimation of the gradient to dynamically adjust the learning rate of each parameter. The main advantage of Adam is that after bias correction, each iteration of the learning rate has a certain range, making the parameters relatively stable. (Ada)

2.3 Cross entropy

There are many loss function in CNNS and in this project we use cross entropy, also called logarithmic loss, log loss or logistic loss. Equation 2.1 shows the exact equation of cross entropy, where t_i is true label and p_i is probability after softmax function. Our goal is minimize this objective function.

$$L_{CE} = -\sum_{i=1}^{n} t_i log(p_i) \tag{2.1}$$

3 Experiments

3.1 Dataset

In this project, we use MNIST dataset. This dataset consists of 60000 images in 10 classes and each image has the shape of 1*28*28. There are 50000 training images and 10000 test images. Figure 1 shows ten images of each ten classes.

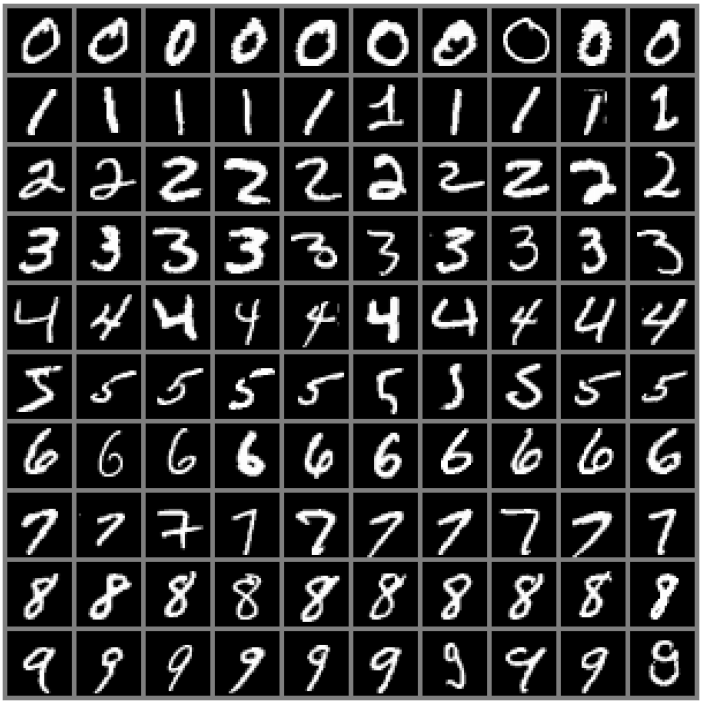

Figure 1: MNIST dataset. 10 images of each class.

3.2 Basic structure

The basic structure of this neural network:

Layer 1: Conv2d: input channel: 1, output channel: 16, kernel size: 3*3, stride : 1 and padding: 1.
Layer 2: ReLu
Layer 3: Maxpool(2x2).
Layer 4: Conv2d: input channel: 16, output channel: 4, kernel size: 3*3, stride : 1 and padding: 1.
Layer 5: ReLu
Layer 6: Maxpool(2x2).
Layer 7: Fully-connected: input channel: 196, output channel: 10.
Layer 8: Softmax.

3.3 Tuning parameters

To analyze the performance of model we need to tune parameters. Firstly, we analyze the statistical significance. Since our model contains stochastic parameters we need to run several simulations to confirm that our model is stable. And we run five simulations and use mean and one standard deviation to represent results. Secondly, we tune learning rate by using 0.01, 0.001, 0.0001 and 0.00001.

4 Results and discussion

The code is in **.ipython** file and main implementation are listed in code snippet part.

4.1 Parameters comparison

Firstly, we discuss the parameters of CNNs and MLP. Table 1 shows the results of comparison. Though our CNNs model has more layers than MLP, it is obvious that the number of parameters is much smaller than MLP. The reason is that convolutional layer shares the weight parameters for the same channel, which significantly decreases the number of parameters when compared with fully-connected layer and makes model efficient.

CNNs Layer	Number of parameters	MLP Layer	Number of parameters
Convolutional layer 1	(3*3+1)*16=160	Fully-connected layer 1	300*(784+1)= 235500
Convolutional layer 2	(3*3+1)*16*4=640	Fully-connected layer 2	10*(300+1)= 3010
Fully-connected layer 1	10*(196+1)=1970		
Total	2770	Total	38510

Table 1: Parameter comparison of CNNs and MLP

4.2 Loss and Accuracy

Figure 2 shows the result of one simulation. Left plot is the result of loss with respect to batch size. It is obvious that the loss converges and fluctuates around at a certain range. Middle plot shows loss with respect each epoch. Overfitting can be observed after five epochs since loss of test is bigger than training. We can also see overfitting after five epochs in accuracy plot. Besides, after 9 epochs both training and test accuracy decrease.

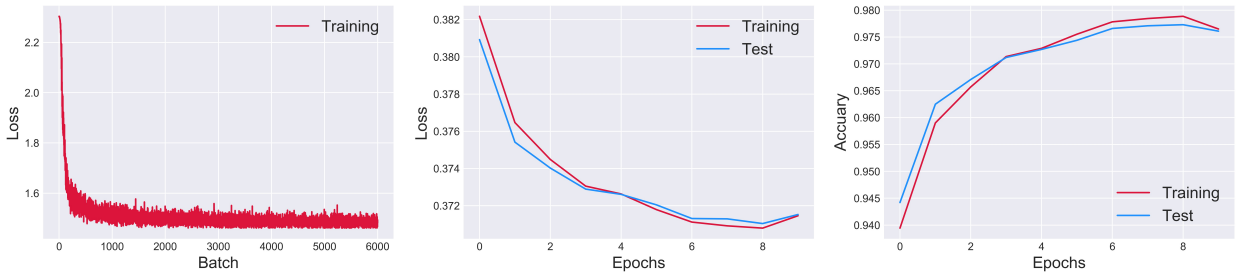

Figure 2: One simulation of CNNs model. Learning rate is 0.001, epoch is 10 and batch size is 100.

4.3 Statistical significance

Additionally, since our model contains statistical parameters we need to explore statistical significance. Figure 3 shows the results. The shading part is the one standard deviation. It is obvious that the results are not very decent because the shading part is very large for both loss and accuracy, which means the results are not really consistent and fluctuates greatly. One solution is to increase the number of simulation times and epoch size.

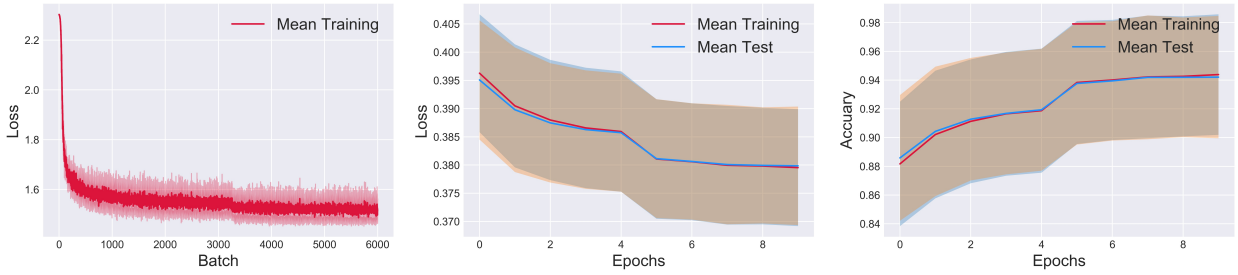

Figure 3: Five simulations of CNNs model. Then we calculate mean and standard deviation(shading part). Learning rate is 0.001, epoch is 10 and batch size is 100.

4.4 Learning rate

Furthermore, we tune the learning rate. Figure 4 and 5 show the loss and accuracy of training and test dataset respectively. Small learning rate has largest loss and smallest accuracy. Because within 10 epochs small learning rate is difficult to converge. Moreover, large learning rate can not have best results either. We can observe both loss and accuracy of 0.01 learning rate are consistent but worse than 0.001 and 0.0001 learning rate. The reason is large learning rate can easily stuck on local minimum and can not escape. Besides, we can also observe that the 0.001 learning rate has comparatively stable performance since the shading area is smaller than other learning rate.

The best accuracy is achieved by 0.001 learning rate, which is 97.91%.

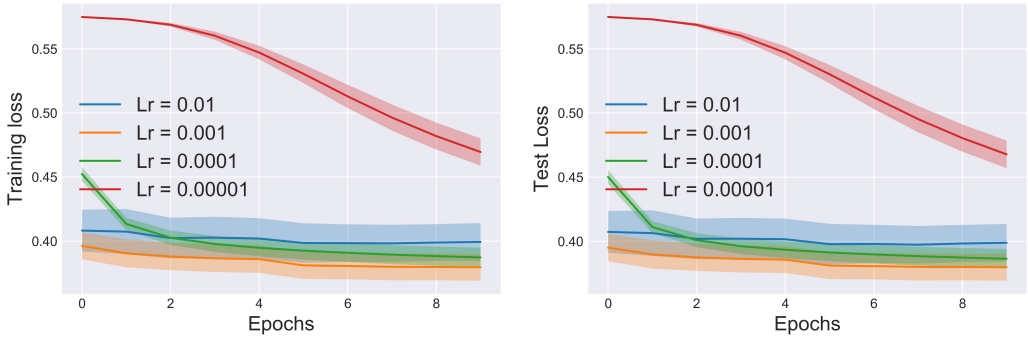

Figure 4: Training loss and test loss with different learning rates. Five simulations of CNNs model. Then we calculate mean and standard deviation(shading part). Learning rate are 0.01, 0.001,0.0001 and 0.00001, epoch is 10 and batch size is 100.

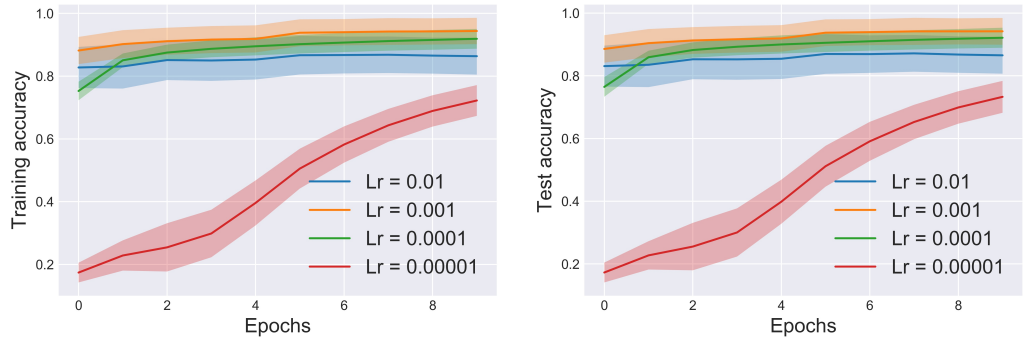

Figure 5: Training accuracy and test accuracy with different learning rates. Five simulations of CNNs model. Then we calculate mean and standard deviation(shading part). Learning rate are 0.01, 0.001,0.0001 and 0.00001, epoch is 10 and batch size is 100.

4.5 Final result

We also decided to try different number of epoch. However, according to the previous experiments models are already overfitting within 10 epochs. Thus, final parameters are Epoch:10, Learning rate: 0.001, Batch Size: 100, Optimizer: Adam, and Batch normalization: No. And the best accuracy is 97.91%.

5 Bonus

Furthermore, we do the comparison about our Custom Convolution layer with Pytorch Conv2d. The detailed code and explanation are in code snippet part. We calculate the time cost for each method. One simulation costs 0.0001053 second for our Custom model, while for Pytorch Conv2d it costs 0.0001197 second. We think it is because in our Custom model we only implement forward. But for Pytorch Conv2d it contains many parameters and models inside which will slow down the computation.

6 A code snippet

```
def Custom_Conv(x,W,padding=1,stride=1):

    # extract images'parameter(batchsize, channels, height, width)
    (batchsize, channel, Height, Width) = x.shape

    # extract output_channel, input_channel, height, width
    (w_number, w_channel, w_height, w_width) = W.shape


    # calculating height after conv
    Height = 1 + int((Height + 2*padding - w_height) / stride)

    # calculating weight after conv
    Width = 1 + int((Width + 2*padding - w_width) / stride)

    # (batchsize, channel*width*height, L)
    # L is the shape of image after Conv
    x_unfold = torch.nn.functional.unfold(x, (Height, Width),padding=padding,
    stride=stride)

    #1.transfrom x_unfold into (batchsize, channel*width*height, L)
    #2. matmul (w_number,w_channel*w_height*w_width).t
    #3. This time we get (batchsize,L,c)-> (batchsize,c,L)
    out_matmul = x_unfold.transpose(1,2).matmul(W.view(W.size(0),-1).t()).transpose(1,2)

    #4. fold (batchsize, channel, Height, Width)
    Z_conv = torch.nn.functional.fold(out_matmul,(Height,Width),kernel_size=(1,1))

    assert(Z_conv.size() == (batchsize, channel, Height, Width))

    return Z_conv
```

```
class CNNs(nn.Module):
    def __init__(self, num_classes=10):
        super(CNNs, self).__init__()
        self.models = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=16,
            kernel_size=3, stride=1, padding=1),

            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),

            nn.Conv2d(in_channels=16, out_channels=4,
            kernel_size=3, stride=1,padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
            )

        self.classifier = nn.Sequential(
            nn.Linear (196, 10),
            nn.Softmax())

    def forward(self, x):
        x = self.models(x)
        x = x.view(-1,196)
        x = self.classifier(x)
        return x
```

```
def find_model(net, trainset, trainloader, testloader,
criterion, optimizer, device, epochsize=10,
batch_size=500):
    trainloss_epoch = []
    testloss_epoch = []
    trainacc_epoch = []
    testacc_epoch = []

    trainloss_batch = []
    for epoch in range(epochsize):
        running_loss = 0.
        for i, data in enumerate(
                torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                            shuffle=True, num_workers=2), 0):
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            trainloss_batch.append(loss.item())

            loss.backward()
            optimizer.step()
#
print('[%d, %5d] loss: %.4f' %(epoch + 1, (i+1)*batch_size, loss.item()))

        with torch.no_grad():

            trainloss = 0.0
            traincorrect = 0
            num = 0

            for inputs, classes in trainloader:
                inputs, classes = inputs.to(device), classes.to(device)
                y = net(inputs)
                loss = criterion(y, classes)
                _, predicted = torch.max(y.data, 1)

                num += classes.size(0)
                trainloss += loss.item()
                traincorrect += (predicted == classes).sum().item()

            trainloss_epoch.append(trainloss / num)
            trainacc_epoch.append(traincorrect / num)
            print(trainacc_epoch)

            testloss = 0.0
            testcorrect = 0
            num = 0

            for inputs, classes in testloader:
                inputs, classes = inputs.to(device), classes.to(device)

                y = net(inputs)
                loss = criterion(y, classes)
                _, predicted = torch.max(y.data, 1)

                num += classes.size(0)
                testloss += loss.item()
                testcorrect += (predicted == classes).sum().item()

            testloss_epoch.append(testloss / num)
            testacc_epoch.append(testcorrect / num)
            print(testacc_epoch)
```

References

Adam. https://blog.csdn.net/brucewong0516/article/details/78838124?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.channel_param&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.channel_param.

Cnns. https://cs231n.github.io/convolutional-networks/#fc.