Deep Learning 2020 **(Due: [25.11])**

Submission Assignment #[3]

Instructor: Jakub Tomczak *Name:* [Xingkai Wang], *Netid:* [xwg510]

1 Problem statement

Convolutional Neural Network (CNN) is a feed-forward neural network, its artificial neurons can respond to a part of the surrounding units in the coverage area, has excellent performance for large image processing. The convolutional neural network consists of one or more convolutional layers and fully connected layer at the top (corresponding to a classic neural network), and also includes associated weights and a pooling layer. This structure enables convolutional neural networks to use the two-dimensional structure of the input data. Compared with other deep learning structures, convolutional neural networks can give better results in image and speech recognition. This model can also be trained using backpropagation algorithms. Compared with other deep, feed-forward neural networks, convolutional neural networks need to consider fewer parameters, making it an attractive deep learning structure.

In this research, we will build a Convolutional Neural Network (CNN) by using Pytorch. Same as classic neural network, there is a loss function that need to be minimized by using backpropagation, and then update the parameters to make our neural network can learn something from training data. In this case, we are interested in minimizing the negative logarithm of the likelihood function:

$$p(y|x;w) = e^{xi}/(\sum e^{xi}) = softmax(CNN_w(x)) \tag{1.1}$$

$$l(w) = -\sum_n \log p(y_n|x_n;w) \tag{1.2}$$

2 Methodology

The initial approach of this research is building a Convolution neural network (CNN) by using Pytorch. We have two convolutional layers, the first convolutional layer has 1 input channel, 16 output channels, and 3x3 kernels, with 1 stride step size and 1 padding size. Then the output will go into relu function and pooling layer. The pooling layer is using MaxPool layer (2x2) which select the maximum value of the pixel value of the feature map of the channel as the representative of the channel. The second convolutional layer has 16 input channels, 4 output channels, and 3x3 kernels, with 1 stride step size and 1 padding size. Then the output will also go into a relu function and maxpool layer. After finishing the convolutional layers, the fully connected layer with 196 inputs and 10 outputs, and softmax function as activation function will be used to obtain the final output of neural network. The framework of the CNN we build is shown below:

$$Conv2d(1 \to 16 channels, 3x3 kernels, 1 stride, 1 padding) \to RELU \to MaxPool(2x2)$$
$$\to Conv2d(16 \to 4 channels, 3x3 kernels, 1 stride, 1 padding) \to RELU \to MaxPool(2x2)$$
$$\to Linear(196 \to 10) \to Softmax$$

Figure 1 presents how to create a CNN and the implementation of forward propagation by using Pytorch.

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, 3, padding = 1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 4, 3, padding = 1)
        self.fc = nn.Linear(196, 10)
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 196)
        x = self.fc(x)
        x = F.log_softmax(x, dim = 1)
        return x
```

Figure 1: implementation of CNN and forward propagation

During the training loop, we will first do the forward propagation and calculate the loss, then we do the backpropagation to find the gradient direction that can minimize the loss and we update the parameters of the neural network. In the backpropagation process, the ADAM optimizer is chosen to do the gradient descent. The adaptive moment estimation (ADAM) algorithm is different from traditional stochastic gradient descent. ADAM designs independent adaptive learning rates for different parameters by calculating the first-order moment estimation and the second-order moment estimation of the gradient. It can greatly improve the efficiency of the neural network. In Pytorch, we do not need to calculate the gradient ourselves, the loss.backward() function can calculate the gradient automatically and optimizer.step() function will update the parameters of neural network. Figure 2 presents the implementation of training process.

```python
criterion = nn.NLLLoss()
optimizer = optim.Adam(net.parameters(), lr = 0.01)
```

```python
total_loss = []
for epoch in range(30):
    epoch_loss = []
    running_loss = 0
    for i,data in enumerate(trainloader,0):
        inputs, labels = data
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs,labels)
        loss.backward()
        optimizer.step()


        running_loss += loss.item()
        epoch_loss.append(loss.item())
        if i % 750 == 749:
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 750))
            running_loss = 0.0
    total_loss.append(np.mean(epoch_loss))

print('finish training')
```

Figure 2: implementation of training process

3 Experiments

In this research, we will use the mnist data to train and test our CNN. The batch size is 16 and the number of training epochs will be set to 30 during all experiments. Therefore, there is only one hyperparameter left which is the learning rate of ADAM optimizer that we can play around. In the first experiment, the learning rate will be set to 0.0001, we will analyze the loss curve during 30 epochs training. In the second experiment, the learning rate will be set to 0.0001 as well, and the ADAM optimizer will be run 3 times to analyze the average

and the standard deviation of the loss value. In the final experiment, the learning rate will be set to 0.0001, 0.01 and 0.05 to train the neural network respectively, and the test data will be used to calculate the accuracy to analyze how different learning rate influences the final performance of neural network.

4 Results and discussion

Figure 3 illustrates the training loss per epoch. These values are the average training loss during each training epoch. The first significant observation we can obtain is that the overall training loss shows a decrease trend, from 0.75 to 0.05, which shows that our neural network has learned something from the training set and backpropagation process minimize the loss value successfully. The second information we can get from this plot is that after 25 epoch training, the decrease of loss becomes very slow, indicating that we are close to the optimal solution at this time. Therefore, stopping training after 25 epoch training may prevent the neural network from overfitting. Meanwhile, it can improve the training efficiency of neural networks.

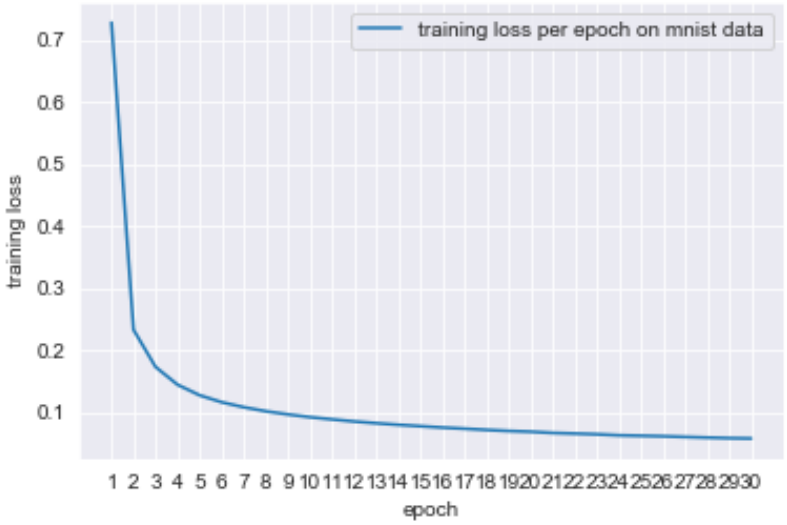

Figure 3: training loss per epoch on mnist data

Figure 4 presents the average loss with standard deviation after running 3 times ADAM optimizer. We can conclude that in the early stage of training, the training loss has a smaller standard deviation, which shows that the training is more stable at this time. However, in the later stage of training, the mean value fluctuates greatly, and accompanied by a large standard deviation, which means that at this time here is a big difference between most of the values and the average value. In other words, although the hyperparameters are the same, the final degree of convergence of different training will also have large differences.

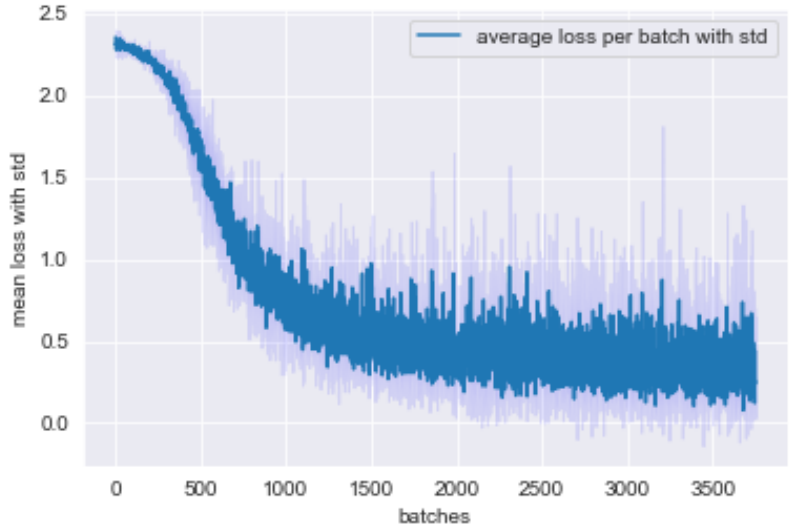

Figure 4: average training loss with std

Table 1 presents how the different learning rate influences the final performance of the neural network. 0.0001, 0.01, 0.05 three learning rate are compared. It is obviously that the neural network has the best performance when the learning rate is 0.0001, which has the highest accuracy of 97% on the test data. Figure 5, figure 6, figure 7 shows the training loss per epoch on the mnist data. When the training loss is set to 0.0001, we get a perfect loss curve, the loss decrease smoothly and eventually converge at around 0.05. When the learning rate is 0.01, we can find that although the learning rate has increased, the decrease of loss has slowed down, only dropping from 0.2 to about 0.135, and there is a small fluctuation in the later stage of training, when the learning rate is further increased to 0.05 At this time, we can find that the loss hardly drops, maintaining at around 2.31, and the test accuracy at this time is only 9%, which shows that when the learning rate is too large, the neural network will not be able to fit the data, which significantly reduce the performance of the neural network.

$\text{learning}_r ate$	0.0001	0.01	0.05
accuracy	97%	93%	9%

Table 1: test accuracy with different learning rate

Figure 5: training loss per epoch with 0.0001 learning rate

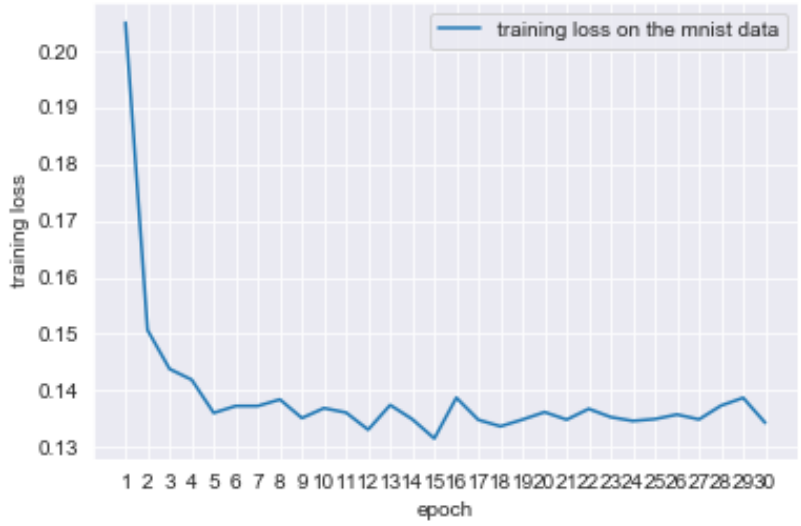

Figure 6: training loss per epoch with 0.01 learning rate

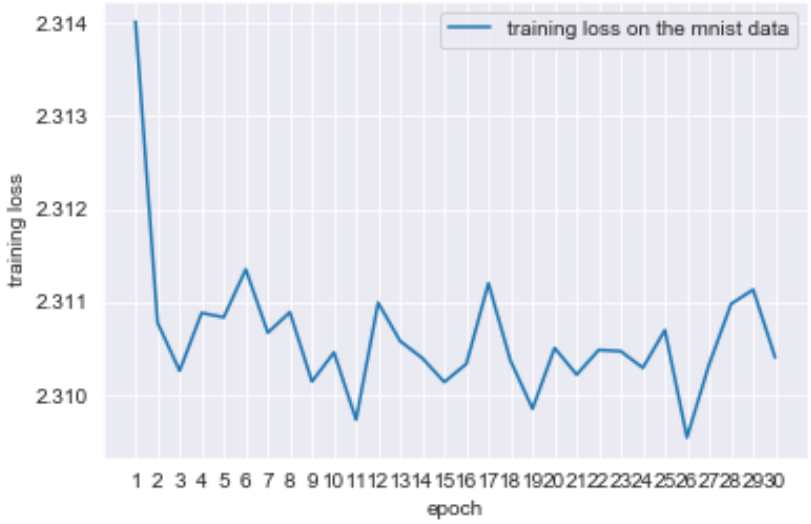

Figure 7: training loss per epoch with 0.05 learning rate

According to the results above, we can draw the choice of hyperparametersis particularly important for the performance of neural networks. For example, if the learning rate is too small, the convergence speed will be too slow, which will affect the learning efficiency. When the learning rate is too large, the neural network will be unable to fit the data. Therefore, we should make full use of the validation data for testing to find the optimal set of hyperparameters, such as the learning rate, batch size, etc. In addition, stop training early at an appropriate training time point, although it may slightly reduce the performance of the neural network, but it may avoid the neural network from overfitting and improve the training efficiency of the neural network.

5 A code snippet

```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        test, labels = data
        outputs = net(test)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))
```

Figure 8: implementation of test accuracy calculation

References

Jiuxiang Gu, Zhenhua Wang, 2017, "Recent Advances in Convolutional Neural Networks"
Diederik P. Kingma, Jimmy Lei Ba, 2017, "ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION"