Report: Homework 1

20203027 Soonjae Kwon

1. Playing Around with Neural Network

1.1. Building Classifier for MNIST Dataset

In 1.1, I wanted to achieve maximum performance using only linear layers and various regularization techniques, not convolutional layers. So based on the 3-layer network(with ReLU activation function, Adam optimizer, and batch normalization), which showed good performance in the skeleton code, Dropout layers and Xavier Initialization are added for higher accuracy. Dropout rate was chosen by several experiments. The code and the results are as follows.

1. **Code**

```python
class MNIST_Net(nn.Module):
    def __init__(self):
        super(MNIST_Net, self).__init__()
        self.fc0 = nn.Linear(28*28, 60) # Layer 1
        self.bn0 = nn.BatchNorm1d(60) # BatchNorm 1
        self.fc1 = nn.Linear(60, 40) # Layer 2
        self.bn1 = nn.BatchNorm1d(40) # BatchNorm 2
        self.fc2 = nn.Linear(40, 20) # Layer 3
        self.bn2 = nn.BatchNorm1d(20) # BatchNorm 3
        self.fc3 = nn.Linear(20, 10) # Layer 4
        self.dropout = nn.Dropout(0.1)
        self.xavier_init()

    def forward(self, x):

        x = x.view(-1,28*28)
        x = self.fc0(x)
        x = self.bn0(x)
        x = F.relu(x)
        x = self.dropout(x)

        x = self.fc1(x)
        x = self.bn1(x)
        x = F.relu(x)
        x = self.dropout(x)

        x = self.fc2(x)
        x = self.bn2(x)
        x = F.relu(x)
        x = self.dropout(x)

        x = self.fc3(x)

        return x

    def xavier_init(self):
        nn.init.xavier_normal_(self.fc0.weight)
        nn.init.xavier_normal_(self.fc1.weight)
        nn.init.xavier_normal_(self.fc2.weight)
        nn.init.xavier_normal_(self.fc3.weight)
```

2. **Test Accuracy**
   97.46% (9746/10000)

3. **Confusion Matrix**

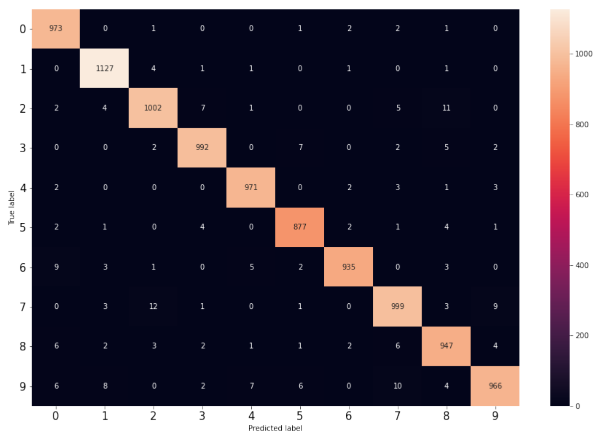

Figure 1: Confusion Matrix of the Implemented Model

4. **Analysis**
   The most confusing case for the model is the mistake of 2(true class) as 7(predicted class). The actual handwriting image sample of MNIST dataset shows that the two letters are quite similar in shape.

1.2. Written Questions

1. **What if we didn't clear up the gradients?**
   Without zero initialization of all gradient values stored

within the optimizer, the exact gradient values cannot be calculated because the gradient values stored in the previous operation remain intact.

2. **Why should we change the network into eval-mode?**
In eval-mode, the dropout layer used for model learning is disabled, and batch normalization is performed with the parameters stored in the training. This enables a more accurate evaluation of the performance of the model.

3. **Is there any difference in performance according to the activation function?**
The performance of model with ReLU(83.36%) is overwhelmingly higher than that of model with sigmoid(59.21%).

4. **Is training gets done easily in exp (3), (4) compared to exp (1), (2)? If it doesn't, why not?**
Since Exp (3) and (4) have one more hidden layer than exp (1), (2), the parameters are also higher and therefore take longer to train. However, the performance was rather poor, indicating that increasing the number of layers does not necessarily improve the performance of the model.

5. **What would happen if there is no activation function?**
Activation functions are used to give a model nonlinearity, and models consisting of only linear combinations are eventually represented as one linear combination, so there are many functions that are not approximated, such as XOR. In addition, the same gradient is always calculated even if the input is different during back-prop, which is not appropriate for the neural network.

6. **Is there any performance difference before/after applying the batch-norm?**
Models with Batch-norm (97.53%) had 1% higher test accuracy than models without it (96.28%).

7. **What may be the potential problems when training the neural network with a large number of parameters?**
As the number of parameters increases, the training time first takes longer, and the computational cost increases due to the higher computational volume. Furthermore, the complexity of the model also increases, thus increasing the possibility of overfitting.

8. **Given input image with shape: (H, W, C1), what would be the shape of output image after applying 2 (F*F) convolutional filters with stride S?**
$(\frac{H-F}{S}+1, \frac{W-F}{S}+1, 2)$

9. **How did the performance and the number of parameters change after using the convolution operation? Why did these results come out?**
Convolution operation resulted in better performance (98.32%) of the model and reduced the number of parameters by a quarter. This is because convolution operations preserve local information and are suitable for computation of 2D images, which is also much more efficient than full connected layers in terms of the number of parameters.

10. **How did the performance change after using the Pooling operation? Why did these results come out?**
Using Pooling operation, the performance of the model (97.89%) decreased by 0.5%, while the number of parameters decreased by a quarter compared to the previous one. This is because MaxPool2d(2) layer extracts only the largest value among the 2*2 values, which can reduce the computational cost instead of risking the loss of information.

2. Playing Around with Convolutional Neural Network

2.1. Implementation: VGG-19

The VGG-19 model is a combination of 16 CONV layers consisted of ConvBlock1 blocks with 2*(CONV+BN+RELU)+POOL and ConvBlock2 blocks with 4* (CONV+BN+RELU)+POOL, and three linear layers added to the end of it. It has a total of 20,365,002 parameters. The implementation code is as follows.

```
class ConvBlock1(nn.Module):

    def __init__(self, in_dim, out_dim):
        super(ConvBlock1, self).__init__()

        self.in_dim = in_dim
        self.out_dim = out_dim

        self.main = nn.Sequential(
            nn.Conv2d(self.in_dim, self.out_dim,
    kernel_size=3, padding=1),
            nn.BatchNorm2d(self.out_dim),
            nn.ReLU(),

            nn.Conv2d(self.out_dim, self.out_dim,
     kernel_size=3, padding=1),
            nn.BatchNorm2d(self.out_dim),
            nn.ReLU(),

            nn.MaxPool2d(2)
        )

    def forward(self, x):
        out = self.main(x)
        return out
```

```python

class ConvBlock2(nn.Module):

    def __init__(self, in_dim, out_dim):
        super(ConvBlock2, self).__init__()

        self.in_dim = in_dim
        self.out_dim = out_dim

        self.main = nn.Sequential(
            nn.Conv2d(self.in_dim, self.out_dim, kernel_size=3, padding=1),
            nn.BatchNorm2d(self.out_dim),
            nn.ReLU(),

            nn.Conv2d(self.out_dim, self.out_dim, kernel_size=3, padding=1),
            nn.BatchNorm2d(self.out_dim),
            nn.ReLU(),

            nn.Conv2d(self.out_dim, self.out_dim, kernel_size=3, padding=1),
            nn.BatchNorm2d(self.out_dim),
            nn.ReLU(),

            nn.Conv2d(self.out_dim, self.out_dim, kernel_size=3, padding=1),
            nn.BatchNorm2d(self.out_dim),
            nn.ReLU(),

            nn.MaxPool2d(2)
        )

    def forward(self, x):
        out = self.main(x)
        return out

class VGG19(nn.Module):

    def __init__(self):
        super(VGG19, self).__init__()

        self.convlayer1 = ConvBlock1(3, 64)
        self.convlayer2 = ConvBlock1(64, 128)
        self.convlayer3 = ConvBlock2(128, 256)
        self.convlayer4 = ConvBlock2(256, 512)
        self.convlayer5 = ConvBlock2(512, 512)
        self.linear = nn.Sequential(
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 128),
            nn.ReLU(),
            nn.Linear(128, 10),
        )
    def forward(self, input):
        out = self.convlayer1(input)
        out = self.convlayer2(out)
        out = self.convlayer3(out)
        out = self.convlayer4(out)
        out = self.convlayer5(out).squeeze()
        out = self.linear(out)
        return out
```

2.2. Building Classifier for CIFAR-10 Dataset

In the same four epochs, the direct-implemented model showed 72.86% test accuracy(Figure 2), while the pre-trained model recorded 86.86% test accuracy(Figure 3), indicating a performance difference of 14%. This demonstrates that it is much more efficient to train a pre-trained model than from scratch when training a huge model like VGG-19. Both models confused classes within similar categories, such as dog and cat, and automobile and truck, but the mistakes on the pre-trained model side was much lower.

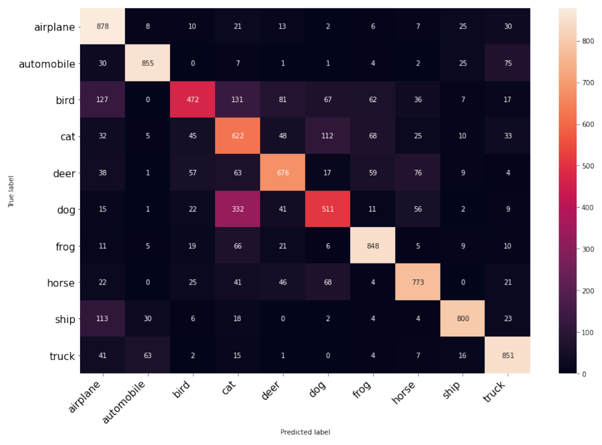

Figure 2: Confusion Matrix of the Implemented VGG-19

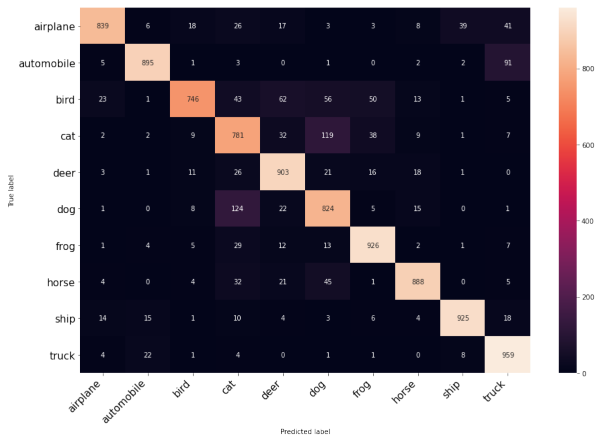

Figure 3: Confusion Matrix of the Pre-trained VGG-19

However, re-training the entire pre-trained VGG-19 model requires a high computational cost as well. So I tried transfer learning, leaving the CNN part intact, and only training the linear layers at the end. As a result, the training time was reduced to 1/300, but it recorded a test accuracy of 46.08%(Figure 4), clearly showing poor result in terms of the model's performance.

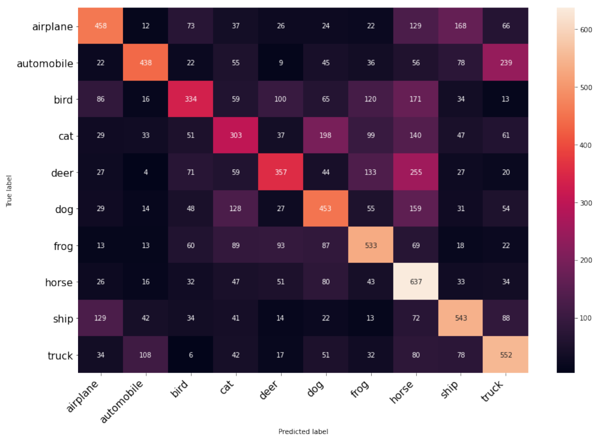

Figure 4: Confusion Matrix of the VGG-19 with Transfer Learning