AI604 Assignment 1

20214196 Sultan Rizky

April 11, 2021

1 Playing around with Neural Network

1.1 Building classifier for MNIST dataset

In this assignment, I successfully built a classifier that is able to accurately perform classification task in the MNIST dataset with an accuracy of **97.93%**, which is higher than 97%.
The model is a four-layer network with total of 89240 parameters, activation function of ReLU and normalization method of batch-normalization embedded between each layers. The model is trained using Adam optimizer and cross-entropy loss is used to evaluate the performance of the corresponding classifier. The in-depth workflow of our classifier is as follows:

- It first receives an input of 28x28 image from the MNIST dataset.
- Next, it enters the first hidden linear layer which outputs 100 features, where each feature is later subjected to batch-normalization and then followed by rectified linear unit function element-wise.
- Then, it enters the second hidden linear layer which outputs 70 features, where each feature is later subjected to batch-normalization and then followed by rectified linear unit function element-wise.
- After than, it enters the third hidden linear layer which outputs 40 features, where each feature is later subjected to batch-normalization and then followed by rectified linear unit function element-wise.
- Finally, it enters the final layer which outputs 10 features, which corresponds to the prediction metric for each classes. The metric which has the maximum value among these features is selected to become the class of an image which our classifier predicts.

Figure 1 below features the snippet of code that describes our build-in network that is able to classify digits in MNIST test dataset with accuracy of **97.93%**.

```python
class MNIST_Net(nn.Module):
    def __init__(self):
        super(MNIST_Net, self).__init__()
        self.fc0 = nn.Linear(28*28, 100) # Layer 1
        self.bn0 = nn.BatchNorm1d(100) # Batchnorm 1
        self.fc1 = nn.Linear(100, 70) # Layer 2
        self.bn1 = nn.BatchNorm1d(70) # Batchnorm 2
        self.fc2 = nn.Linear(70, 40) # Layer 3
        self.bn2 = nn.BatchNorm1d(40) # Batchnorm 3
        self.fc3 = nn.Linear(40, 10) # Layer 4

    def forward(self, x):
        x = x.view(-1,28*28)
        x = self.fc0(x)
        x = self.bn0(x)
        x = F.relu(x)
        x = self.fc1(x)
        x = self.bn1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = self.bn2(x)
        x = F.relu(x)
        x = self.fc3(x)
        return x
```

Figure 1: Proposed MNISTNet

The main motivation behind this design of network is simply because I simply believe that adding an additional hidden layer and increasing the number of output features in each hidden layer affects the performance of our classifier for the better. However, I also think that adding too much parameters within our classifier will result in worse classification accuracy compared to the current model.

To examine how well the network performs on different categories, aside of denoting that the proposed model achieves classification accuracy of **97.93%** in the test MNIST dataset, I generate a confusion matrix, indicating for every actual digit in an image from the MNIST dataset (rows) which digit the network guesses (columns). Figure 2 below denotes the confusion matrix for this digit classification problem in MNIST dataset.

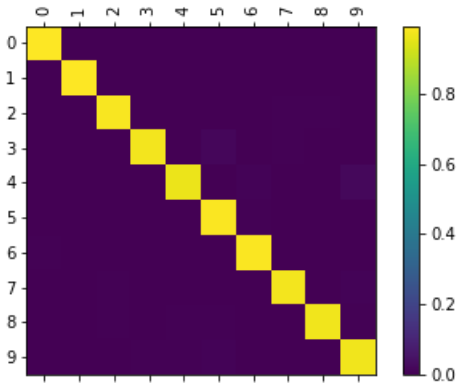


Figure 2: Confusion Matrix for Classification in MNIST Dataset

According to Figure 2 above, it almost seems like the classifier that I've made accurately predicts the digit that is given from an image in MNIST dataset. However, by taking a closer look at the confusion tensor in the attached `hw1-1(assn).ipynb` code file, one may notice that when our model tries to classify an image of digit '4' in the MNIST test dataset, it has a **2.24%** chance that it will classify the corresponding image as digit '9'. The possible reason that

caused this is due to the fact they resemble each other, especially as denoted in the last row of Figure 3 below.

Figure 3: Sample Digits in MNIST Dataset

1.2 Written Questions

The following section is dedicated to answer questions that is specified in the handout.

1. What if we didn't clear up the gradients?
   Ans) If we didn't clear up the gradients while starting our training loop, the parameter gradients will be accumulated or summed up on subsequent backward passes, or in other words, every loss.backward() function call. This will cause the gradient to point in some other direction than the intended direction, which is towards the minimum in this case.

2. Why should we change the network into eval-mode?
   Ans) The reason why we should change the network into eval-mode before testing/evaluating our network's performance is to change the behavior of specific layers in our network such as BatchNorm and Dropout layer into inference or testing mode. In case of BatchNorm, for instance, when it is set to eval-mode, instead of normalizing the activations using the mean and standard deviations of each minibatch, it uses mean and standard deviation that has been estimated by the BatchNorm layer across all the training examples during training. Similarly, for Dropout layer, setting the network to eval-mode will cause the layer to not drop any activations.

3. Is there any difference in performance according to the activation function?
   Ans) As it can be noticed in the Exp (1) and Exp (2) within the `hw1-1(assn).ipynb` file, changing the activation function of a two-layer network from sigmoid to ReLU increases the test accuracy from 53.85% to 84.89%. Doing the similar thing on a three-layer network increases the test accuracy from 11.35% to 70.36%, as denoted in Exp (3) and Exp (4).

4. Is training gets done easily in experiment (3),(4) compared to experiment (1),(2)? If it doesn't, why not?
   Ans) If we are talking about how easy the training process is in terms of training loss at the end of training steps, it can be noticed from the `hw1-1(assn).ipynb` file that training loss for 3-layer network at the end of iteration is higher than that of 2-layer network under both sigmoid and ReLU activation functions. Similarly, according to the previous answer, test accuracy for 3-layer network in Exp (3) and (4) (11.35% for sigmoid and 70.36% for ReLU) is lower compared to that of 2-layer network in

Exp (1) and (2) (53.85% for sigmoid and 84.89% for ReLU). In addition, it takes longer time to train 3-layer networks than 2-layer networks in the following set of experiments. Due to these reasons, it can be safely said that training doesn't get easier in Exp (3) & (4) compared to Exp (1) & (2).

5. What would happen if there is no activation function?
   Ans) If there's no activation function within the network, it will act just like a linear regression model. This is because activation function aims to introduce non-linearity towards the network, which results in higher network complexity in comparison to a linear regression model.

6. Is there any performance difference before/after applying the batch-norm?
   Ans) According to the result of experiment (5) and (6) where batch normalization layer is not embedded within both 3-layer and 2-layer network respectively and experiment (7) and (8) where batch normalization layer is embedded within both 2-layer and 3-layer network respectively, one can notice that in case of two-layer network, adding batch normalization layers within the network increases the test accuracy from 95.63% to 96.73%. Similarly, in case of three-layer network, adding batch normalization layers within the network increases the test accuracy from 96.54% to 97.44%. Note that these networks are trained using Adam optimizer and ReLU activation function. From these results, one can conclude that applying the batch-norm layers increase the test accuracy of our network compared to the case where batch normalization layers are not implemented.

7. What may be the potential problems when training the neural network with a large number of parameters?
   Ans) When we are training a neural network with large number of parameters, the neural network will have a huge potential to overfit when no regularization technique is employed. In other words, the model will perform extremely well on the training dataset but doesn't perform well on the test dataset. Another problem that might be encountered by having a large number of parameters within our neural network is that it requires more computational power to train the network compared to a neural network with relatively smaller number of parameters. However, with various regularization methods such as dropout, weigh decay, and data augmentation, these overfitting problems can be eliminated and the corresponding neural network can generalizes better since it can represent more complicated functions than that with smaller number of parameters.

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?
   Ans) For an input image with shape (H, W C1), once we apply 2 (F * F) convolutional filters with stride S towards the corresponding image, the output image will have a shape of (H_{out}, W_{out}, C_{out}), with

$$H_{out} = \lfloor \frac{H-F}{S} + 1 \rfloor$$
$$W_{out} = \lfloor \frac{W-F}{S} + 1 \rfloor$$
$$C_{out} = 2$$

9. How did the performance and the number of parameters change after using the Convolution operation? Why did these results come out?
Ans) Comparing the performance of the network that we made in section 1.1 and the one that utilizes convolution operation in Exp (9), it can be noticed that by replacing the network with convolution operation instead of linear operation improves the classification performance from 97.93% to 98.23% when at the same time, the number of parameters within our network decreases significantly from 89240 to 11842. The reason of this performance improvement under smaller number of parameters is because instead of just performing a simple linear transformation to the incoming data in form of $y = xA^T + b$, the convolutional layer performs a 2D convolution over an input signal composed of several input planes. In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{out}, H_{out}, W_{out})$ can be precisely described as follows:

$$\text{out}(N_i, C_{out_j}) = \text{bias}(C_{out_j}) + \sum_{k=0}^{C_{in}-1} \text{weight}(C_{out_j}, k) \star \text{input}(N_i, k)$$

where $\star$ is the valid 2D cross-correlation operator, N is batch size, C denotes a number of channels, H is height of input planes in pixels, and W is width in pixels. Since this is more computationally complex than operations performed in linear layer which involves less number of parameters, this results in better performance than the previous network.

10. How did the performance change after using the Pooling operation? Why did these results come out?
Ans) Comparing the result of Exp (9) and Exp (10), where an additional pooling operation is implemented in Exp (10), it can be noticed from the `hw1-1(assn).ipynb` that the classification accuracy within the MNIST test dataset decreases from 98.23% to 97.88%. This is because pooling operation that is performed by the pooling layer acts in a similar fashion to a downsampling operation, which makes the representation smaller and more manageable. In this case, the network in Exp (10) employs a max-pooling operation, where it performs 2D max pooling over an input signal composed of several input planes. This results in lesser number of parameters contained within the network, which in turns, downgrades the classification performance in MNIST test dataset.

2 Playing Around with Convolutional Neural Network

2.1 Implementation: VGG-19

To complete our task in properly implementing the VGG-19 network, we first take a look at the following snippet of code that describes the overall structure of VGG-19 in Figure 4 below.

```
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() # 16 x 512 x 1 x 1에서 뒤 1 x 1 축약
        out = self.linear(out)
        return out
```

Figure 4: VGG-19 snippet in `hw1-2(assn).ipynb`

Here, all we need to do is to complete ConvBlock1 and ConvBlock2 module within the notebook file `hw1-2(assn).ipynb`. In order to do so, I simply refer to the works of Simonyan and Zisserman [1] which describes the implementation of VGG-19, specifically described in column E of Figure 5 below.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

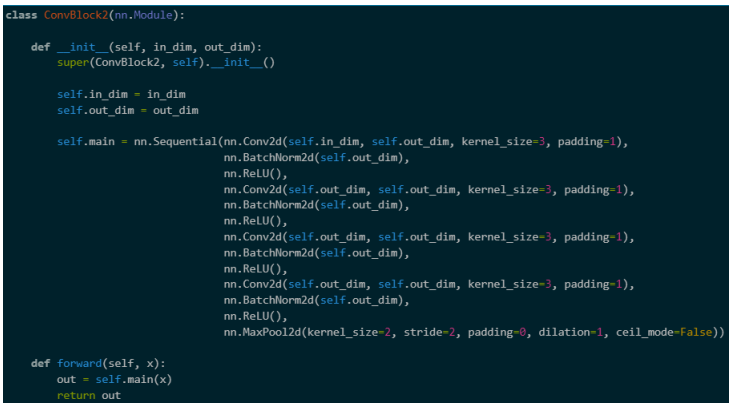
Figure 5: Detailed Specification of VGG[1] (VGG-19 is in column E)

To implement the module ConvBlock1 in the code snippet shown in Figure 4, by following the architecture explaiantion from Simonyan and Zimmerman[1] and network specification shown in column E of Figure 5 above, we first stack a two-dimensional convolution layer with input of in_dim, output of out_dim, kernel_size of 3, and padding of 1. This is then followed by a batch normalization 2D layer with size of out_dim and activation function of ReLU. Next, we follow it with another two-dimensional convolution layer with input out_dim, output of out_dim, kernel_size of 3, and padding of 1, followed by a batch normalization 2D layer with size of out_dim and activation function of ReLU. Finally, a two-dimensional max pooling layer with kernel_size of 2, stride of 2, zero padding, and dilation of 1 is embedded at the end. The code for ConvBlock1 is denoted in Figure 6 below.

```
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(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False))

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

Figure 6: ConvBlock1

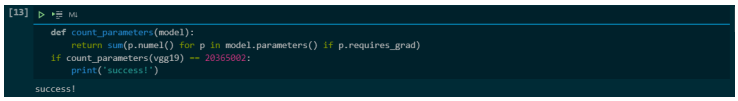
To implement the module ConvBlock2 in the code snippet shown in Figure 4, by following the architecture explaiantion from Simonyan and Zimmerman[1] and network specification shown in column E of Figure 5 above, we first stack a two-dimensional convolution layer with input of in_dim, output of out_dim, kernel_size of 3, and padding of 1. This is then followed by a batch normalization 2D layer with size of out_dim and activation function of ReLU. Next, we embed three stacks of layers consecutively where each stack consists of two dimensional convolution layers with input of out_dim, output of out_dim, kernel_size of 3, and padding of 1, followed by a batch normalization 2D layer with size of out_dim and activation function of ReLU. Finally, a two-dimensional max pooling layer with kernel_size of 2, stride of 2, zero padding, and dilation of 1 is embedded at the end. The code for ConvBlock2 is denoted in Figure 7 below.



```
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(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False))

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

Figure 7: ConvBlock2

To check the correctness of our implemented VGG-19, we use the count_parameters function that is defined in `hw1-2(assn).ipynb`. In our notebook, a 'success!' message is successfully printed, meaning that our network matches with the specifications restricted for VGG-19, as shown in Figure 8 below.



```
[13]
    def count_parameters(model):
        return sum(p.numel() for p in model.parameters() if p.requires_grad)
    if count_parameters(vgg19) == 20365002:
        print('success!')
success!
```

Figure 8: Result of Implementation Test

2.2 Building Classifier for CIFAR-10 dataset

Using the implemented VGG-19 that has been specified in 2.1, by referring to the `hw1-2(assn).ipynb` module included in the submission, the corresponding model achieves a test accuracy of **66%** on the whole CIFAR-10 test dataset. To analyze the accuracy of our model with respect to each class labels, one can refer to the confusion matrix for the image classification problem in CIFAR-10 using our implemented VGG-19 in Figure 9 below. It follows the convention that is mentioned for the confusion matrix in section 1.1, where each row suggests the true labels and each column represents the prediction made by our network.

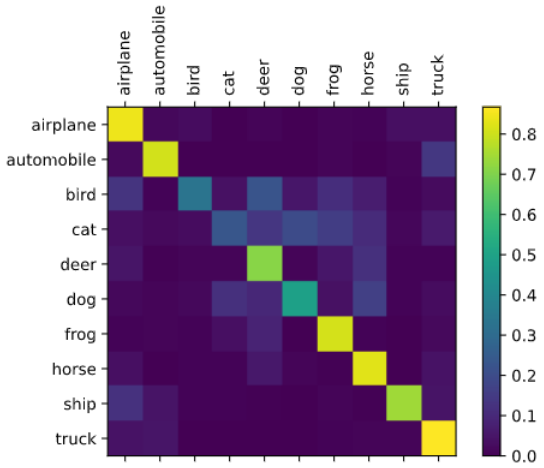


Figure 9: Confusion Matrix for the Image Classification problem in CIFAR-10 using VGG-19

From the figure above, notice that our network performs classification reasonably well (with $\geq$70% classification accuracy) on airplane, automobile, deer, frog, horse, ship, and truck images, while it struggles to classify bird, cat, and dog images accurately. Our classifier especially struggles in classifying a cat image correctly (only with accuracy of 23.2%), often mistaking it for other classes such as dog (with 19.7% chance), frog (with 15.9% chance), or deer (with 13.8% chance). Similarly, our classifier also struggles in classifying a bird image (only with accuracy of 33.2%) where our model often mis-classifies it as a deer (with 22.5% chance of this happening). In case of dog images in CIFAR-10 test dataset, our classifier has a 49.3% chance of guessing it correctly, and has a chance of misclassifying it as a horse (with a chance of 16.5%). This suggests that our model still needs to be trained longer in order to achieve better test accuracy.

Now, using the pretrained VGG-19 to perform the classification task in CIFAR-10 dataset, by referring to the `hw1-2(assn).ipynb` module included in the submission, one may notice that the corresponding model achieves an improved test accuracy of **87%** on the whole CIFAR-10 test dataset. To analyze the accuracy of our model with respect to each class labels, one can refer to the confusion matrix for the image classification problem in CIFAR-10 using the pretrained VGG-19 in Figure 10 below.

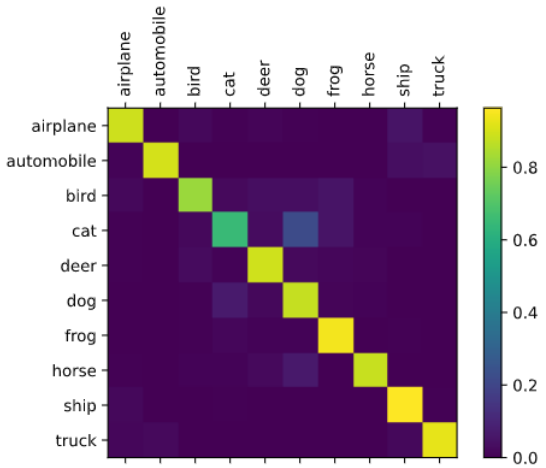


Figure 10: Confusion Matrix for the Image Classification problem in CIFAR-10 using Pretrained VGG-19

From the figure above, notice that our network performs classification in a pretty accurate fashion (with $\geq$80% classification accuracy) on almost every classes except cat, where it only has 65.2% chance of classifying cat correctly. When the pretrained model performs classification on cat images within the CIFAR-10 dataset, it has 21.9% chance of mis-classifying cat as a dog, which is probably caused by the similarity in their features such as shape or color. Overall, the pretrained VGG-19 achieves higher classification accuracy on every class labels compared to the implemented VGG-19 without any pretraining.

References

[1] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *International Conference on Learning Representations*. 2015.