COMP3314 Assignment 2 Report

1. Introduction

We will use different optimizer to try to find the best settings and tune the parameters for each optimizer, during which we will use the control variate method. Also, we may change the net architecture and the other parameters like the convolution kernel size.

2. Code to Print Accuracy for Each Class

As we still need to output the accuracy for each class, which was not given in the template, we use the `confusion_matrix` from `sklearn.metrics` package. The code to show the accuracy for each class is shown in the following:

```
with torch.no_grad():
    for data in test_dataloader:
        images, labels = data[0].to(device), data[1].to(device)
        # calculate outputs by running images through the network
        outputs = model(images)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        #caculate the accuracy for each class
        y_pred.extend(predicted.view(-1).tolist())
        y_true.extend(labels.view(-1).tolist())
accuracy = 100 * correct / total
print('Accuracy of the network on test images: {accuracy:.2f}%'.format(accuracy=accuracy))

# Calculate the confusion matrix
conf_matrix = confusion_matrix(y_true, y_pred)
# Calculate the accuracy for each class
```

```
class_accuracy = 100 * conf_matrix.diagonal() /
conf_matrix.sum(1)
# Print the accuracy for each class
for idx, acc in enumerate(class_accuracy):
    print(f'Accuracy for class {idx}: {acc:.2f}%')
```

3. Tuning of the Parameters in the Template

Using `Adam` optimizer:

EPOCHES	LR	STEP SIZE	GAMMA	0	1	2	3	4	5	6	7	8	9	OVERALL	TIME(S)
15	0.001	10	0.7	90.6%	90.4%	93.8%	91%	95.4%	91.2%	92.4%	94.2%	94.6%	92.6%	92%	600.54
15	0.0001	10	0.7	89.4%	91.8%	88.8%	85.4%	91.8%	87.8%	88.4%	93.6%	82.4%	86.4%	85.5%	676.41
15	0.0001	10	0.8	89.6%	91.8%	90.2%	87.4%	90.8%	82.4%	85.4%	92.8%	81.6%	88.6%	88%	646.44
15	0.005	10	0.7	75%	83.2%	79.8%	71.6%	82.6%	80.2%	66.8%	81.4%	78.8%	75.2%	77%	686.10
15	0.0015	10	0.7	91.2%	93.6%	91.4%	90.4%	93.6%	89.8%	92.4%	93.8%	90.2%	92.4%	91%	602.21
15	0.001	10	0.7	94%	92%	96%	89.4%	95.6%	91.2%	92%	94.4%	91.2%	92.6%	92.4%	689.22
15	0.001	10	0.5	92.2%	91.8%	92.4%	87.2%	97.2%	94.2%	90.8%	94.8%	93.8%	94.2%	92.8%	615.78
15	**0.001**	**10**	**0.6**	**94%**	**91%**	**92.4%**	**89%**	**95.8%**	**92%**	**92%**	**93.5%**	**93%**	**93%**	**92.9%**	**665.49**
15	0.001	10	0.4	94.2%	92.4%	93.8%	90%	96.2%	90.2%	88.2%	92.6%	91.8%	89.6%	92.8%	667.70
15	0.001	5	0.7	91.8%	91%	94.8%	89.2%	94.4%	92%	94.8%	93.8%	89.8%	93.4%	92.4%	686.32
15	0.001	5	0.6	91.4%	93.2%	93%	89.4%	95.8%	91.8%	91.2%	94.6%	92.4%	93.6%	92.7%	599.81
15	0.001	5	0.5	91.6%	94.4%	91.2%	88.6%	92%	93.2%	92.4%	93.6%	90.2%	94.2%	92.8%	608.27
15	0.001	15	0.7	92.2%	93.4%	92..6%	89.2%	95.2%	89.2%	92.8%	95.4%	89.6%	92.2%	92.5%	650.07
15	0.001	15	0.6	94.4%	92.2%	93%	88.8%	95%	92.2%	93%	96%	90.4%	92.6%	92.6%	639.91
15	0.001	15	0.5	93.6%	91.6%	93.2%	91.2%	95.4%	90.8%	89.8%	94.4%	92.4%	91%	92.7%	651.10

We follow the following steps to tune the several parameters:

- First, we tune the `learning rate` with the other parameters remaining the same, and we found that if we tune the learning rate bigger or smaller, the overall performance (accuracy) will be worse than when it is 0.001.
- Then, we tune the `gamma` parameter, which is about the decay strategy, with the other parameters remaining the same. We found that the overall performance is the best when the gamma parameter is 0.6.

- Next, we tune the `step_size`, which is also about the decay strategy, with the other parameters remaining the same. We found that the overall performance is the best when this parameter is 10.

Therefore, we fix the `learning rate = 0.001`, `gamma=0.6`, `step_size=10` and then tune the epoch number. Below is the data table we have

EPOCHES	0	1	2	3	4	5	6	7	8	9	OVERALL	TIME(S)
10	92.4%	92%	92.8%	91.2%	95%	92.4%	90.8%	93.4%	91.2%	91.8%	92.1%	429.22
14	94.4%	92.4%	94.6%	85.8%	95.8%	92%	92.2%	95.2%	90.6%	92.8%	92.1%	602.35
16	88.8%	91%	91.8%	91.8%	95%	91.8%	94.2%	95.6%	92.2%	92%	92.3%	670.11
17	93.6%	92.6%	92%	89.8%	94.2%	90.6%	91%	94.4%	90%	94.6%	92.8%	718.35
20	91.8%	92.4	93.6%	90.2%	95.8%	93.4%	95.8%	94.6%	90.2%	90.6%	92.9%	817.10
30	94.0%	89.6%	93.4%	91.4%	92.8%	94.4%	91.0%	92.4%	92.0%	94.4%	93.01%	1022.56

The accuracy is the highest when epoch is 30. Further increasing the epoch number would result in a decline in accuracy. The reason of it might be overfitting when setting the epoch too high.

We found that if we tune the epoch number larger or smaller, the overall performance will not be better than when the epoch is 30. Therefore, we find the best settings for `AdamW` optimizer is

$$\begin{cases} epoch = 30 \\ \gamma = 0.6 \\ learning\ rate = 0.001 \\ step\_size = 10 \end{cases} \Rightarrow accuracy = 93.01\%$$

4. Change of Optimizers

Now, we change the optimizer to see if there are any better choices:

- `SGD`: the accuracy is always around 10%, which is bad for training.
- `ASGD`: the accuracy is always around 10%, which is bad for training.
- `Adagrad`: The accuracy is always around 15%, which is still not good.
- `Adamax`: The accuracy is much better than the previous three, we try more parameters to see if its accuracy can be greater than 93.6%.

EPOCHES	LR	STEP SIZE	GAMMA	0	1	2	3	4	5	6	7	8	9	OVERALL	TIME(S)
30	0.001	3	0.7	91.6%	91.2%	93.4%	86%	94.4%	91.8%	89.2%	94.6%	90.8%	89.8%	91.2%	603.38
30	0.001	3	0.6	92.4%	89.6%	92.2%	88.6%	95.8%	90.6%	93.0%	94.8%	86.0%	89.2%	91.4%	635.23
30	0.001	3	0.5	93.6%	93.4%	92.6%	89.2%	94.4%	89.2%	90.8%	94.4%	89.2%	89.6%	91.6%	619.02
30	0.001	3	0.8	89.2%	93.2%	93.8%	86.6%	94.6%	90.0%	91.4%	94.0%	93.2%	91.2%	91.8%	648.81

We found that the accuracy is not bad but still not better than 93.6%.

- `Adadelta`: The maximum accuracy we found is around 30%, which is not good enough.
- `RMSprop`: The accuracy is around 14%, which is bad for training.

Therefore, we will continue using the `AdamW` optimizer.

5. Change of Architecture and Data Augmentation

After optimizing the hyperparameters and identifying the most suitable optimizer, the next step is to consider modifying the architecture of the convolutional neural network (CNN) or adjusting the data augmentation techniques to further enhance accuracy of the model.

5.1 Change of Network Architectures

The architecture of a neural network plays a crucial role in determining the accuracy of the digit recognition process. Our objective is to identify an architecture that surpasses the performance of the provided template.

Upon analyzing the given template architecture, we observe that it consists of three distinct blocks. The first two blocks comprise two convolutional layers followed by a pooling layer in each block. The final block is composed of linear layers.

To improve the architecture, our primary focus is to explore modifications that can enhance its performance. Our main idea is to deepen the layers, so that the model can catch more features.

5.1.1 Adding one block of convolutional layers

One approach we propose is to introduce an additional block of convolutional layers, following the same pattern as the existing two blocks, just before the linear layers:

```
self.network = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=7, padding=3, stride=2),
            nn.ReLU(),
            nn.Dropout(p=0.1),
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Dropout(p=0.1),
            nn.MaxPool2d(2, 2),
            nn.Dropout(p=0.1),

            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Dropout(p=0.1),
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Dropout(p=0.1),
            nn.MaxPool2d(2, 2),
            nn.Dropout(p=0.1),

            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Dropout(p=0.1),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Dropout(p=0.1),
            nn.MaxPool2d(2, 2),
            nn.Dropout(p=0.1),

            nn.Flatten(),
            nn.Linear(256*2*2, 128),
            nn.ReLU(),
            nn.Linear(128, 10)
```

```
            )
```

In our attempt to improve the architecture, we made a specific modification by adding dropout layers behind each ReLU activation function layer in the convolutional layer blocks. Additionally, we adjusted the dropout rate to 0.1 to mitigate the risk of information loss. However, upon executing the code, we observed a noticeable decrease in accuracy, resulting in a value of 91.96%. This decline suggests that the new architecture is unsuitable for our task.

5.1.2 Adding two convolutional layers

We reserved the idea of the dropout layers behind each ReLU activation function layer in the convolutional layer blocks. However, we abondoned the additional block. Instead, we just added one convolutional layer into each block existing originally. Moreover, we added one batch normalization layer behind each convolutionaly layer block Batch to make the model faster and more stable through normalization of the layers' inputs by re-centering and re-scaling:

```
self.network = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=7, padding=3,
stride=2),
            nn.ReLU(),
            nn.Dropout(p=0.1),
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Dropout(p=0.1),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Dropout(p=0.1),
            nn.MaxPool2d(2, 2),
            nn.Dropout(p=0.1),
            nn.BatchNorm2d(64),

            nn.Conv2d(64, 128, kernel_size=3, stride=1,
padding=1),
            nn.ReLU(),
            nn.Dropout(p=0.1),
            nn.Conv2d(128, 256, kernel_size=3, stride=1,
padding=1),
            nn.ReLU(),
            nn.Dropout(p=0.1),
```

```
            nn.Conv2d(256, 256, kernel_size=3, stride=1,
padding=1),
            nn.ReLU(),
            nn.Dropout(p=0.1),
            nn.MaxPool2d(2, 2),
            nn.Dropout(p=0.1),
            nn.BatchNorm2d(256),

            nn.Flatten(),
            nn.Linear(256*4*4, 128),
            nn.ReLU(),
            nn.Linear(128, 10)
            )
```

Indeed, the modified architecture proved to be more appropriate, as it led to a significant increase in accuracy from 93.01% to 93.7%. However, upon further analysis, we observed that this model exhibited high variance. The training accuracy reached 96.4%, but the test accuracy showed a noticeable drop to 93.7%.

The high variance suggests that the model may be overfitting the training data, resulting in reduced generalization performance on unseen test data. Due to overfitting, we turned towards data augmentation to adjust the data, making them more appropriate for our task.

5.2 Change of Data Augmentation

5.2.1 torchvision.transforms.RandomAffine()

In our efforts to address the issue of overfitting, we incorporated two additional parameters, namely `scale` and `shear`, to the existing function. The `scale` parameter allows for random scaling of the images, with the given factor as the maximum scaling value. On the other hand, the `shear` parameter controls the range of random shearing angles applied to the images.

Upon implementing these additional parameters, we observed a decrease in the training accuracy. However, the test accuracy experienced a notable improvement, surpassing the 94% threshold for the first time. This outcome suggests that the model's generalization ability has improved, as indicated by the enhanced performance on unseen test data.

By introducing the `scale` and `shear` parameters, we introduced additional variations to the training data, making it more diverse and robust. This increased variability likely helped the model learn more generalized representations, leading to improved performance on unseen examples.

It is worth noting that finding the optimal values for these parameters may require experimentation and fine-tuning. Adjusting the `scale` and `shear` ranges can further enhance the model's ability to generalize. The final accuracy was as follows:

0	1	2	3	4	5	6	7	8	9	OVERALL
96.2%	93.4%	95.0%	92.0%	95.0%	92.4%	94.0%	95.4%	92.6%	94.8%	94.08%

5.2.2 torchvision.transforms.ColorJitter()

During our online search for data augmentation techniques, we discovered a function that we believe could further enhance our model's performance. This function incorporates parameters such as `brightness`, `contrast`, `saturation`, and `hue` to control various aspects of the image.

The `brightness` parameter allows us to manipulate the brightness level of the image, while the `contrast` parameter modifies the difference between dark and light colors, enhancing the image's contrast. The `saturation` parameter enables us to adjust the intensity of different colors, while the `hue` parameter alters the hue or color tone of the image.

Through many trials, we found the optimal values for these parameters.

After conducting five runs of the code, we calculated the average accuracy to be 94.44% (94.48%, 94.00%, 94.68%, 94.42%, 94.62%). Among these runs, the highest achieved accuracy was 94.68%.

```
Finished Training
Accuracy of the network on test images: 94.68%
Accuracy for class 0: 95.20%
Accuracy for class 1: 94.60%
Accuracy for class 2: 95.60%
Accuracy for class 3: 92.80%
Accuracy for class 4: 95.20%
Accuracy for class 5: 94.20%
Accuracy for class 6: 95.00%
Accuracy for class 7: 95.60%
Accuracy for class 8: 93.60%
Accuracy for class 9: 95.00%
```

0	1	2	3	4	5	6	7	8	9	OVERALL
96.2%	93.6%	94.6%	91.2%	96.0%	94.0%	94.4%	95.6%	94.6%	94.6%	94.48%
96.0%	94.2%	94.8%	89.2%	95.6%	93.2%	93.8%	96.0%	95.2%	93.6%	94.00%
95.2%	94.6%	95.6%	92.8%	95.2%	94.2%	95.0%	95.6%	93.6%	95.0%	94.68%
96.6%	93.8%	95.4%	92.8%	94.8%	92.0%	95.8%	95.4%	94.8%	93.8%	94.42%
95.4%	94.6%	95.2%	90.8%	97.2%	93.8%	95.0%	95.4%	94.4%	94.4%	94.62%

The data augmentation code at last was as follows:

```
'train': transforms.Compose([
        transforms.RandomAffine(degrees=10, translate=(0,0.1), scale=(0.9,1.1), shear=10),
        transforms.ColorJitter(brightness=0.5, contrast=(1.1,1.2), saturation=0.4, hue=0.1),
        transforms.Resize((32,32)),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),

        ]),
```

6. Conclusion

Our computer is Dell XPS 15 9520 with i9-23900K CPU. It took 2633.41 seconds to run our model.

Optimizing the parameters, selecting an appropriate architecture, and applying suitable data augmentation techniques are crucial steps in maximizing the accuracy of a CNN model for a specific task.

Through careful experimentation and analysis, we can fine-tune the model's hyperparameters, such as learning rate, epoch, step size, gamma and optimizer, to achieve better performance.

Additionally, exploring different architectures, such as adding or adjusting convolutional layers or increasing network depth or width, can contribute to improved accuracy.

Furthermore, implementing effective data augmentation techniques, such as scaling, shearing, adjusting brightness, contrast, saturation, and hue, can enhance the model's ability to generalize and recognize diverse patterns.

It is essential to evaluate the model's performance by monitoring both training and test accuracy, ensuring that any modifications or additions do not lead to overfitting or a decline in generalization capability. Regular experimentation and analysis enable us to identify the optimal parameters, architecture, and data augmentation strategies that maximize accuracy for the specific task at hand.