University of Macau

Faculty of Science and Technology

Department of Computer and Information Science

澳門大學

UNIVERSIDADE DE MACAU

UNIVERSITY OF MACAU

CISC3024 Pattern Recognition

Classification of Street View House Numbers Using PyTorch

Submitted by:

Agustin Yan, DC125714

Date of Submission: 2023/11/21

Introduction

In this project, A classification network on the Street View House Numbers (SVHN) dataset has been built by deep learning techniques.

SVHN dataset is a real-world image dataset, which contains the house numbers in the Google Street View images. It consists of over 600,000 labeled digit images.

The classification model I built is a Small VGG model with much less number of parameters.

Methodology

Data Processing and Augmentation

I firstly implemented a class named **SVHNAugmentedDataset** to download and process the original SVHN dataset.

```python
class SVHNAugmentedDataset(torch.utils.data.Dataset):
    def __init__(self, root, split='train', download=True,
                 max_rotation=15, min_crop_size=16, max_aspect_ratio_change=0.1, train=True):
        self.dataset = SVHN(root=root, split=split, download=download) # load the dataset
        self.train = train # to control the train and test

        self.augment = A.Compose([ # when in train mode, do the augmentation
            A.Rotate(limit=max_rotation, p=0.5),
            A.RandomResizedCrop(height=32, width=32, scale=(min_crop_size/32, 1.0), ratio=(1-max_aspect_ratio_change,
            A.Normalize(mean=[0.4377, 0.4438, 0.4728], std=[0.1980, 0.2010, 0.1970]),
            ToTensorV2()
        ])

        self.normalize = A.Compose([ # when in test mode, only do normalization
            A.Normalize(mean=[0.4377, 0.4438, 0.4728], std=[0.1980, 0.2010, 0.1970]),
            ToTensorV2()
        ])

    def __len__(self):
        return len(self.dataset)

    def __getitem__(self, idx):
        image, label = self.dataset[idx]
        image = np.array(image)

        if self.train:
            augmented = self.augment(image=image)
        else:
            augmented = self.normalize(image=image)

        image = augmented['image']
        return image, label
```

Figure 1. SVHNAugmentedDataset

To avoid the model from “memorizing” the training data, the images used to train the model need to be augmented. There are three parameters in the __init__ function affect the augmentation: max_rotation, min_crop_size and max_aspect_ratio_change.

The **max_rotation** decides the maximum angle of random rotation during the image augmentation, the **min_crop_size** decides the minimum crop size for random cropping of images and the **max_aspect_ratio_change** decides the maximum changes in aspect ratio when random cropping in images.

These augmentations would effectively increase the number of samples in the dataset, thereby helping the model to learn features better and reduce the risk of overfitting.

Also, a boolean value: **train**, has been added in the __init__ function, this parameter is to split the dataset into training part and testing part. Because the testing image should not be augmented except normalization.

Then use the following codes to load and transform the dataset, and create the dataloader for the model training usage, since the training of neural networks usually uses gradient descent method. And if there is too much training data, it is not possible to send all the data into the calculation (epoch) at once. Now, the data is divided into several parts (batch):

```
# load and transform the dataset
train_dataset = SVHNAugmentedDataset(root='./data', split='train', download=True, train=True)
test_dataset = SVHNAugmentedDataset(root='./data', split='test', download=True, train=False)

# create dataloader
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

Figure 2. Load the dataset and create DataLoader

Neural Network Setup

After loading the dataset, I set up a neural network model named **SmallVGG** for classifying the datasets. This class inherits from nn.Module, which is the base class

for all neural network modules in PyTorch. The detail implementation is shown in the Figure 3:

```
class SmallVGG(nn.Module):
    def __init__(self):
        super(SmallVGG, self).__init__()
        self.conv_layers = nn.Sequential(
            nn.Conv2d(3, 8, kernel_size=3, padding=1), nn.GroupNorm(2, 8), nn.ReLU(),
            nn.Conv2d(8, 16, kernel_size=3, padding=1), nn.GroupNorm(4, 16), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2), nn.Dropout(0.25),

            nn.Conv2d(16, 32, kernel_size=3, padding=1), nn.GroupNorm(8, 32), nn.ReLU(),
            nn.Conv2d(32, 32, kernel_size=3, padding=1), nn.GroupNorm(8, 32), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2), nn.Dropout(0.25),

            nn.Conv2d(32, 32, kernel_size=3, padding=1), nn.GroupNorm(8, 32), nn.ReLU(),
            nn.Conv2d(32, 32, kernel_size=3, padding=1), nn.GroupNorm(8, 32), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2), nn.Dropout(0.25)
        )

        self.fc_layers = nn.Sequential(
            nn.Linear(32 * 4 * 4, 256), nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(256, 10) #output layer 10 classes
        )

    def forward(self, x):
        x = self.conv_layers(x)
        x = x.view(x.size(0), -1)
        x = self.fc_layers(x)
        return x
```

Figure 3. SmallVGG neural network

This model start with **Convolutional operation**, It takes a 3-channel input (RGB images) and outputs 8 channels, using a 3x3 convolution kernel with padding of 1 to maintain the spatial dimensions. After each Convolutional Layer, **GroupNorm** is used for normalization. This helps to speed up training process and improve stability, Then the ReLU **Activation function** will be applied to introduce non-linearity. After that, applied a **Max Pooling Layer** to reduce the size of the feature maps and extract important features. Finally, use a **Dropout** layer to randomly discard some neurons to reduce the overfitting problems.

The output from the Convolutional Layers will be flattened and passed to Fully Connected Layers. The first Fully Connected Layer maps the input features from 32×4×4 to 256 dimensions. Then use ReLU function to introduce non-linearity and user Dropout to randomly discard some neurons to reduce overfitting. And second Fully Connected Layer caps the 256 dimensional features to 10 dimensions, which corresponds to 10 classes in the classification.

The forward function defines how data flows through the model during forward propagation. The input data firstly passed through the convolution layers, then the output of the convolution layer will be flattened and inputs to the fully connected layers.

Training and Evaluation

After setting up the SmallVGG model, A **train_and_evaluate** function is implemented to do the training and evaluation process. The detail implementation has shown in below figure 4.

```
def train_and_evaluate(model, train_loader, test_loader, criterion, optimizer, num_epochs):
    train_losses = []
    test_losses = []
    test_accuracies = []
    all_labels = []
    all_scores = []

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0

        # Training loop
        for images, labels in tqdm(train_loader):
            images, labels = images.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item() * images.size(0)

        epoch_train_loss = running_loss / len(train_loader.dataset)
        train_losses.append(epoch_train_loss)

        model.eval()
        test_loss = 0.0
        correct = 0
        total = 0
```

```
        # Evaluation loop
        with torch.no_grad():
            for images, labels in test_loader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                loss = criterion(outputs, labels)
                test_loss += loss.item() * images.size(0)

                _, predicted = torch.max(outputs, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

                all_labels.append(labels.cpu().numpy())
                all_scores.append(outputs.cpu().numpy())

        epoch_test_loss = test_loss / len(test_loader.dataset)
        test_losses.append(epoch_test_loss)
        accuracy = 100 * correct / total
        test_accuracies.append(accuracy)

        print(f"Epoch [{epoch + 1}/{num_epochs}] - "
              f"Train Loss: {epoch_train_loss:.4f}, "
              f"Test Loss: {epoch_test_loss:.4f}, "
              f"Accuracy: {accuracy:.2f}%")

    # Convert lists to numpy arrays for ROC AUC computation
    all_labels = np.concatenate(all_labels)
    all_scores = np.concatenate(all_scores)

    return train_losses, test_losses, test_accuracies, all_labels, all_scores
```

Figure 4. train_and_evaluate function

The function contains 6 paremeter: model, train_loader, test_loader, criterion, optimizer and num_epochs.

model represents for the model used to training, which is the SmallVGG model I implemented above.

The **train_loader** and **test_loader** stands for the training dataset and testing dataset, which is the augmented SVHN dataset I metioned above.

The **criterion** is a kind of loss function, the input of the function is the scoring vector of the image classification, The smaller the result of the loss function, the more accurate the prediction is.

The **optimizer** is used to improve the neural network. It makes the training process faster and save more time.

The last parameter is **num_epochs**. When a complete dataset passes through a neural network once and returns once, this process is called an epoch. And this parameters is to control the number of epochs during the training.

And it would return the losses during training and testing, accuracies of the during testing process and all labels of image (all_labels) and all predicted results (all_scores) from the model for plot the **Receiver Operating Characteristic (ROC)** curve and the **Area Under Curve**.

Disscussion

When the neural network is set up properly and the training and evaluation function is implemented. I initialize the model, set criterion, optimizer and number of epochs to run the code, the detailed steps are shown in below:

```
model = SmallVGG().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001) #learning rate
# Example usage
num_epochs = 100
train_losses, test_losses, test_accuracies, all_labels, all_scores = train_and_evaluate(model, train_loader, test_loader, criterion, optimiz

# Define your output path and filename
output_dir = 'C:/Users/Administrator/Desktop/GitHub-2024/CISC3024_New/Record'  # Update this path according to your system
output_filename = 'Final.png'  # Specify your desired filename

output_file = os.path.join(output_dir, output_filename)
os.makedirs(output_dir, exist_ok=True)

plot_graphs(all_labels, all_scores, output_file=output_file)
```

Figure 5. Initialization

There are seven parameters I modified to improve the model: **optimizer types, learning rates, batch size, maximum rotation, minimum crop size, max aspect ratio change and number of epochs**

A. Optimizer with different learning rates

The first two parameters I modified is the optimizer type, along with the learning rate. I tried three types of the optimizer:

1. **SGD (Stochastic Gradient Descent) optimizer** is the most basic and widely used optimization algorithm. Unlike traditional batch gradient descent, SGD uses only one sample (or a small batch of samples) in each iteration to update the model parameters.
2. **RMSProp (Root Mean Square Propagation) optimizer** is an adaptive learning rate optimization algorithm specifically designed to solve the learning rate adjustment problem during gradient descent.
3. **Adam (Adaptive Moment Estimation) Optimizer** is a widely used gradient descent optimization algorithm. It combines the advantages of the momentum method and RMSProp to improve the training efficiency by adaptively adjusting the learning rate.

And test these three optimizers with different learning rate, and other parameters remains the same. The output graph contains 4 charts, the left top is the class-wise ROC curve, the right top is the overall Micro and Macro ROC curve, the left bottom is the training, testing loss and accuracy of the model. And the right bottom is the class-wise performance metrics of the model. The below tables show the performance of different optimizers at different learning rates.

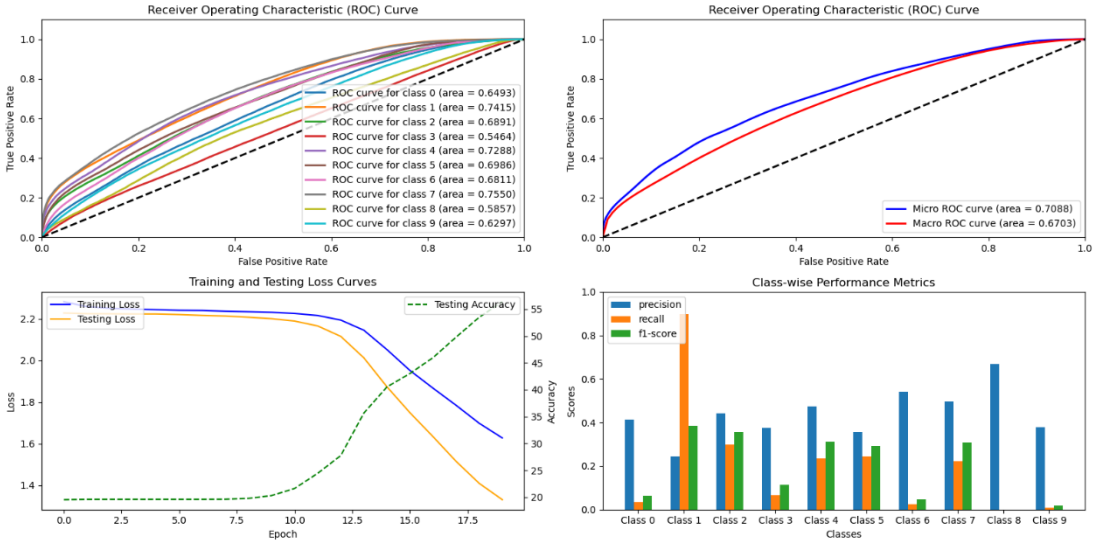

Table 1(A). SGD(Lr=0.0001)

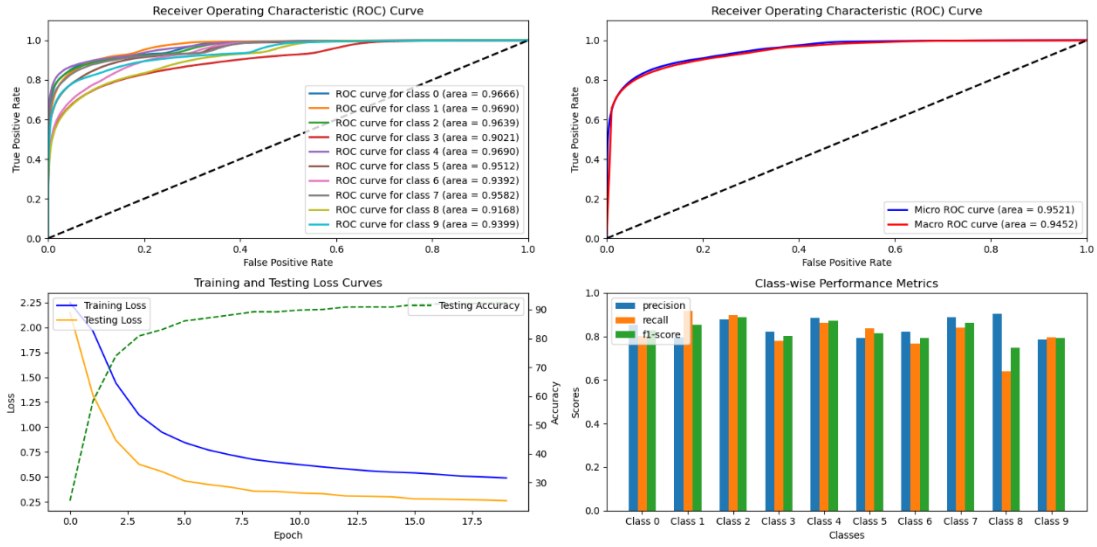

Table 1(B). SGD(Lr=0.001)

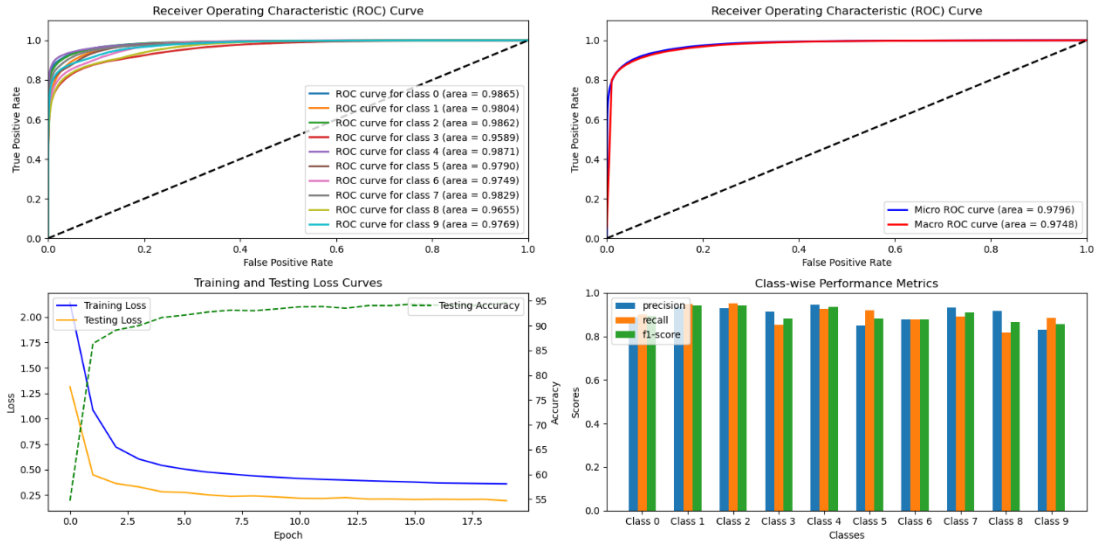

Table 1(C). SGD(Lr=0.01)

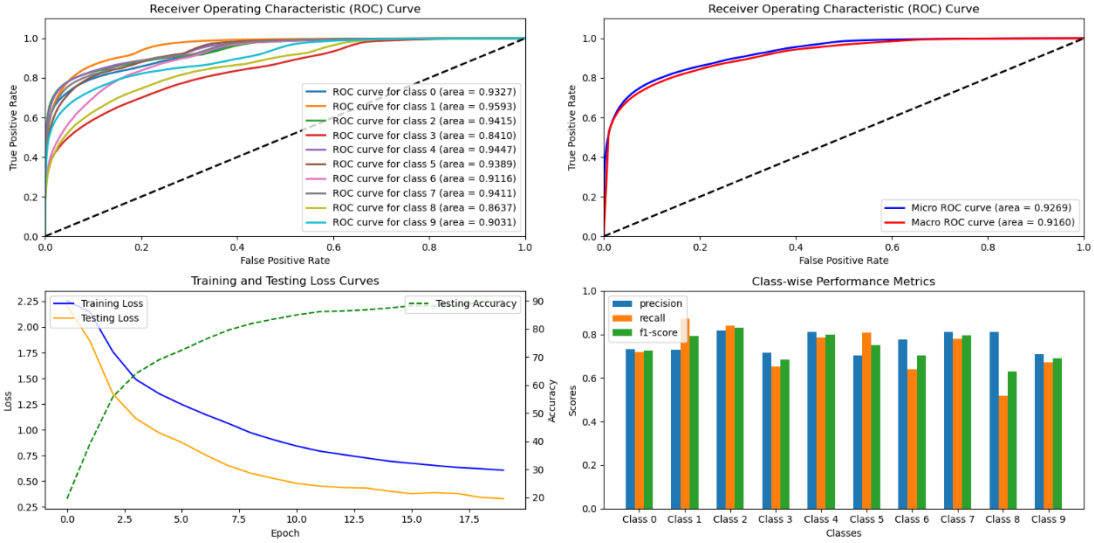

Table 1(D). SGD(Lr=0.0005)

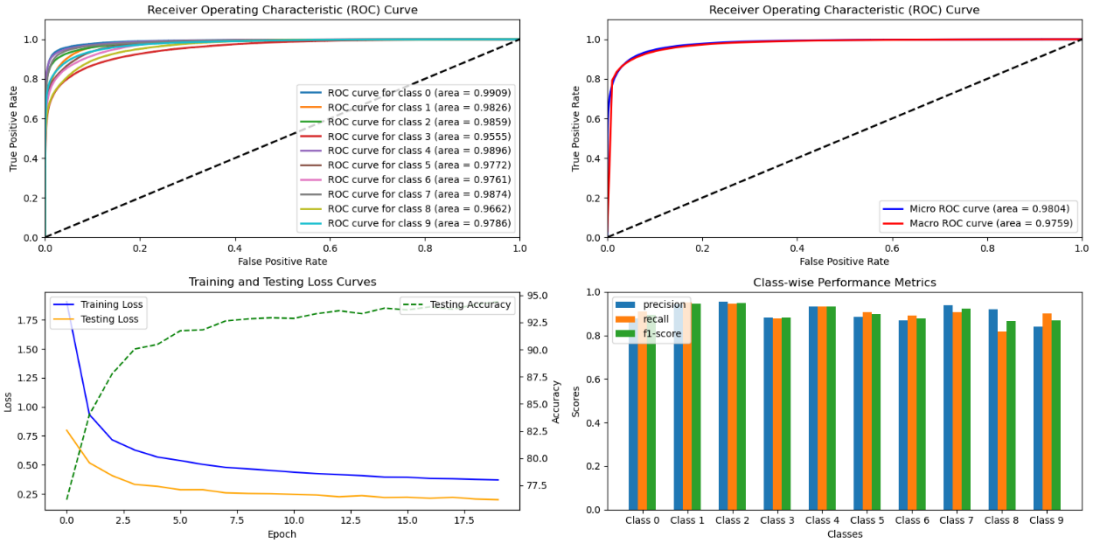

Table 1(E). SGD(Lr=0.005)

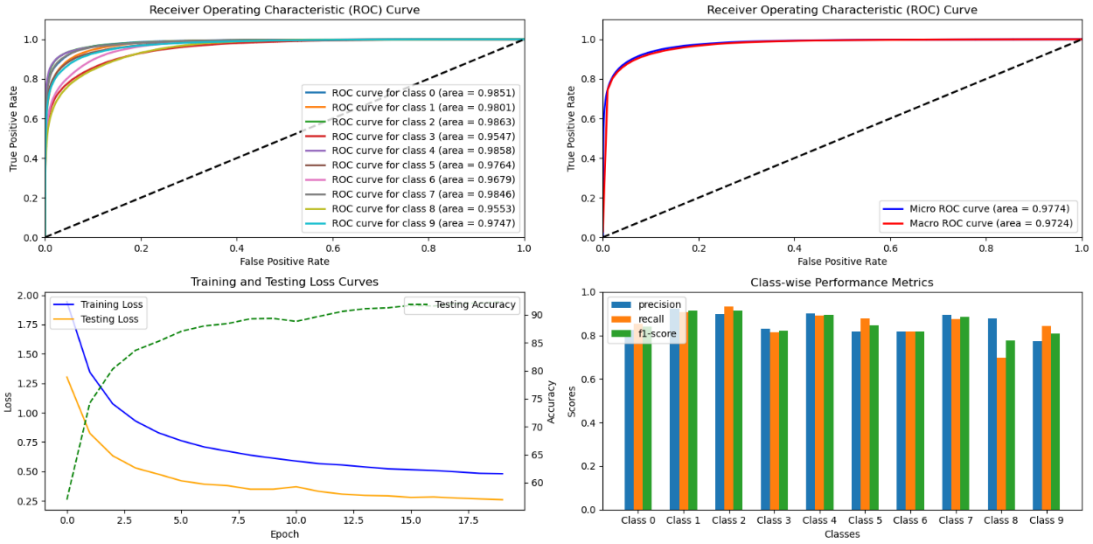

Table 2(A). RMSprop(Lr=0.0001)

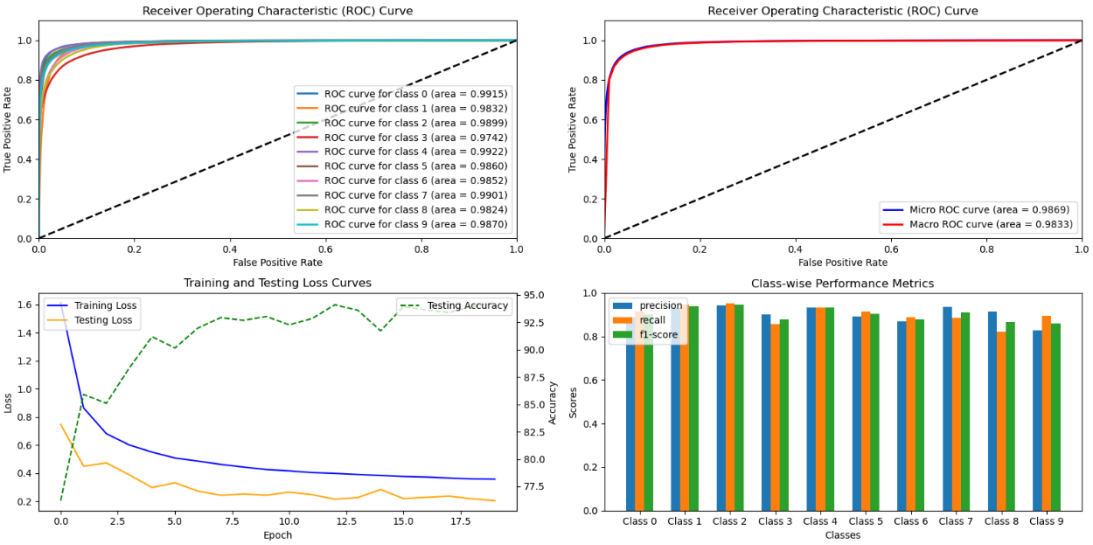

Table 2(B). RMSprop(Lr=0.001)

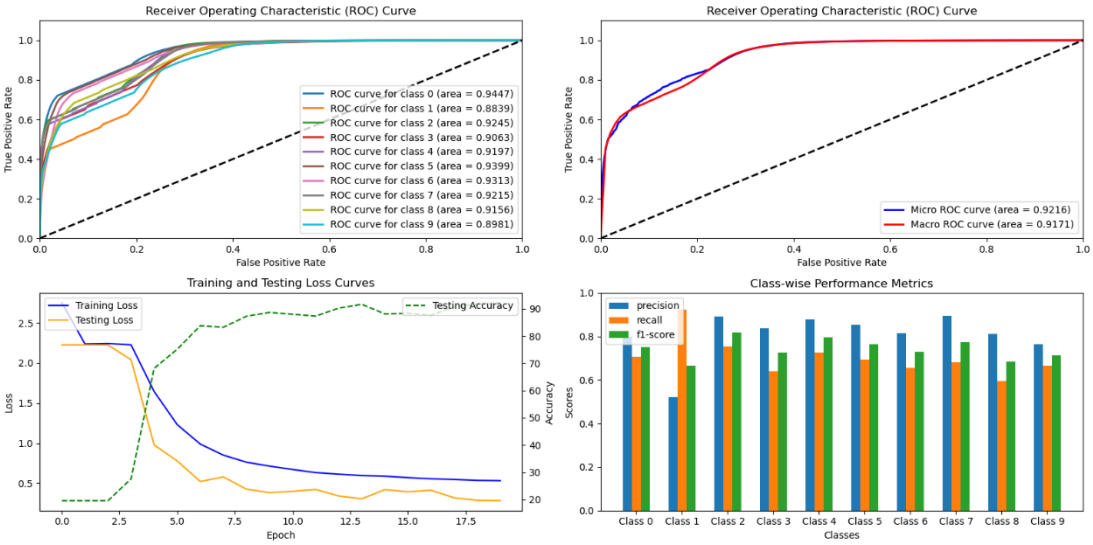

Table 2(C). RMSprop(Lr=0.01)

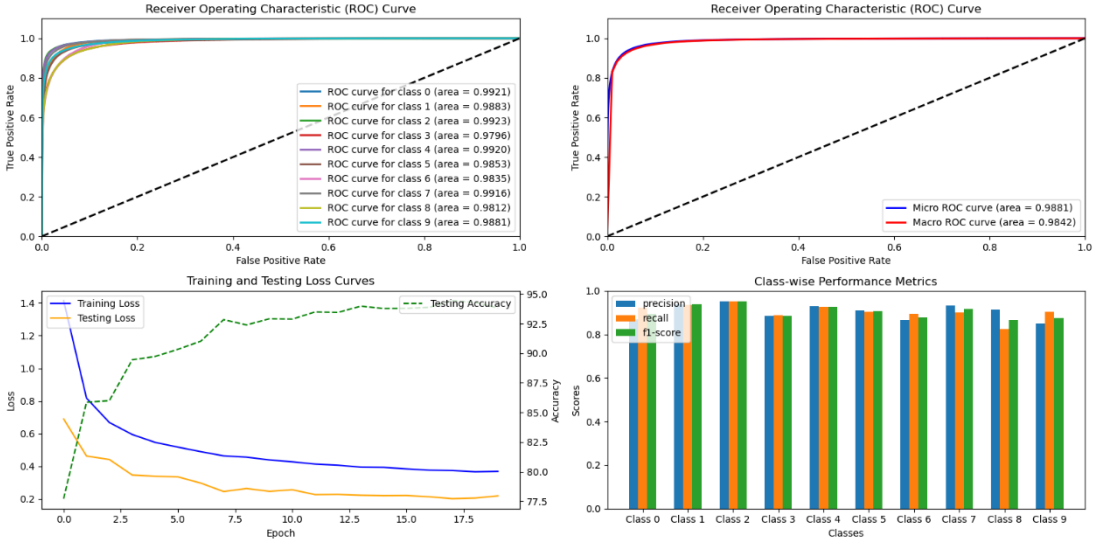

Table 2(D). RMSprop(Lr=0.0005)

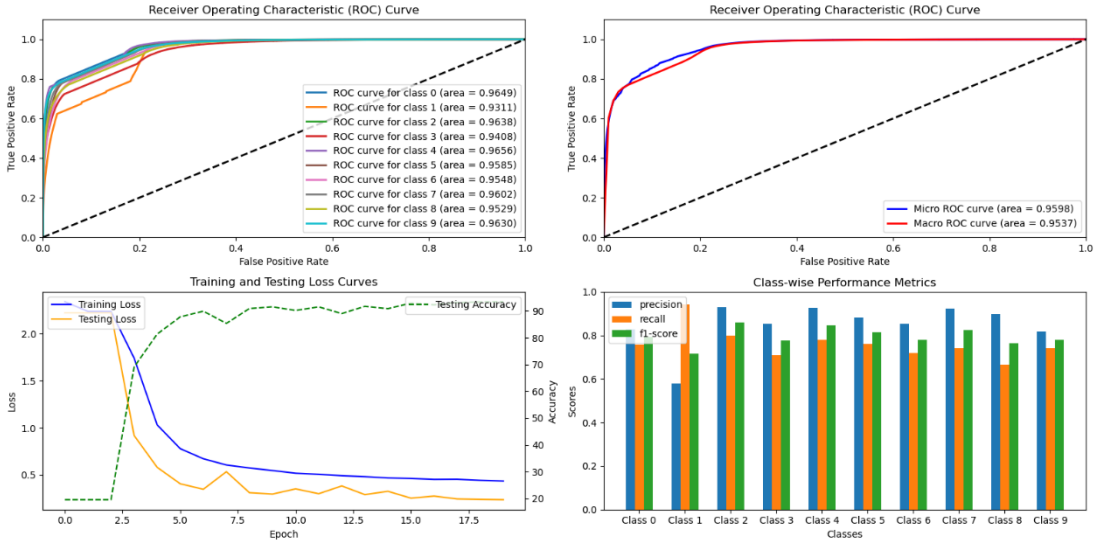

Table 2(E). RMSprop(Lr=0.005)

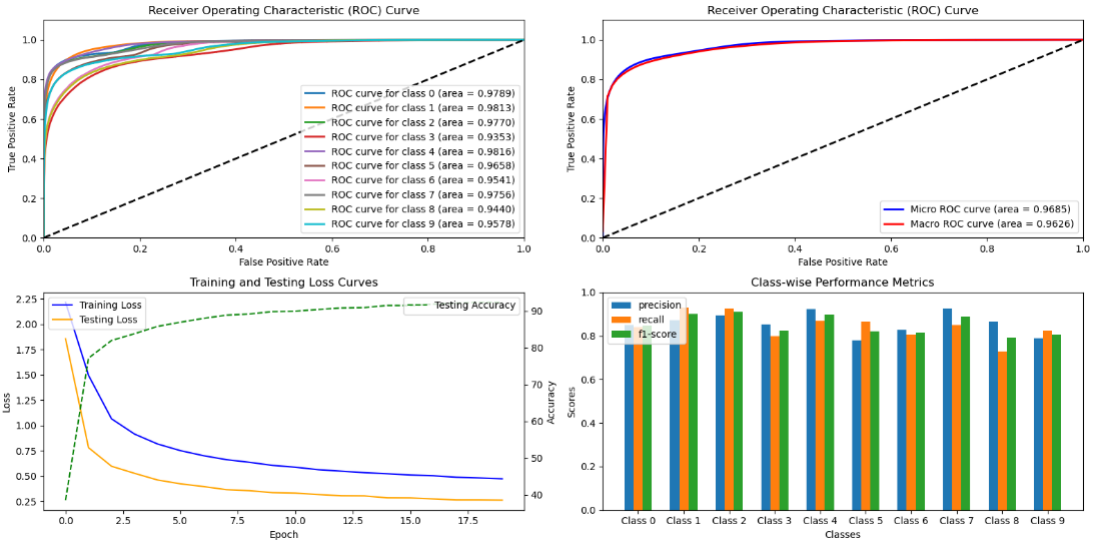

Table 3(A). Adam(Lr=0.0001)

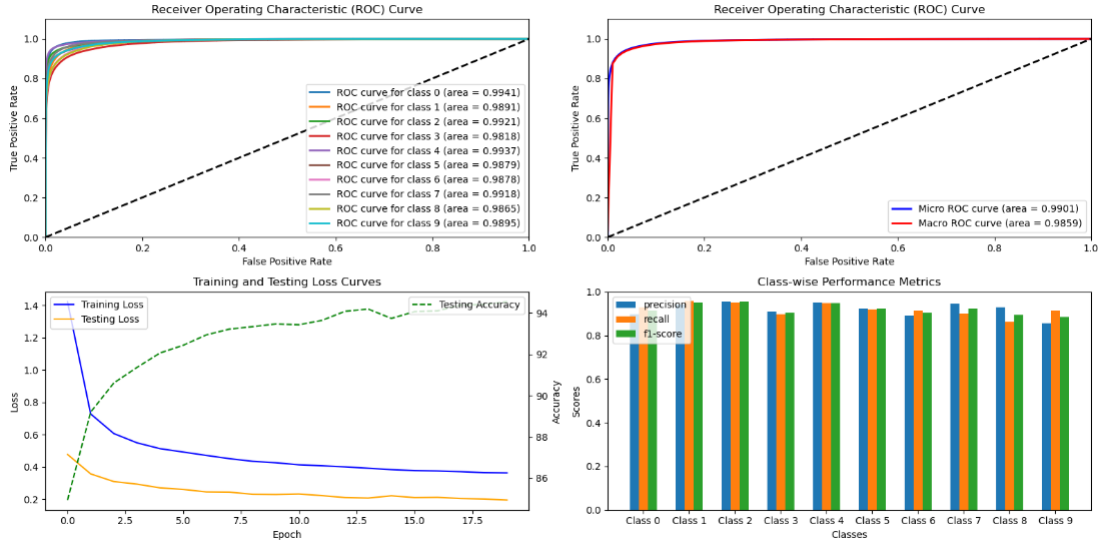

Table 3(B). Adam(Lr=0.001)

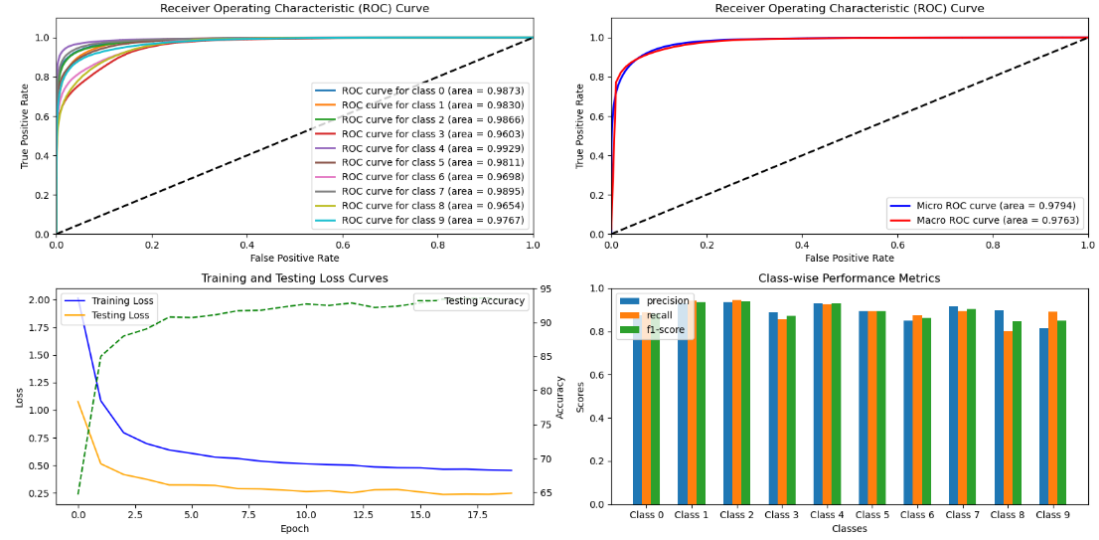

Table 3(C). Adam(Lr=0.01)

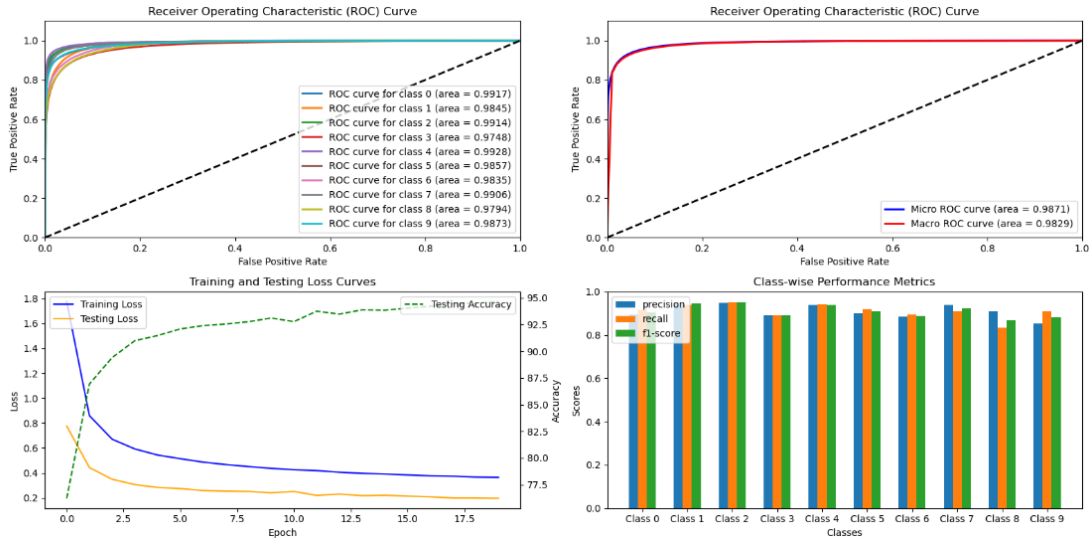

Table 3(D). Adam(Lr=0.0005)

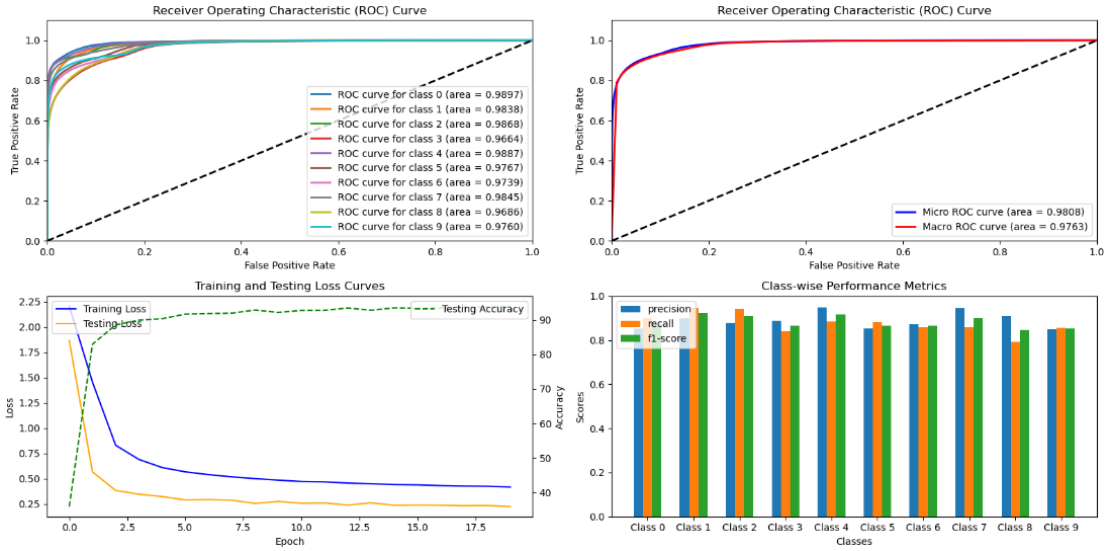

Table 3(E). Adam(Lr=0.005)

And the table 4 is the comparison table for these 15 graphs, I chose the Adam Optimizer with the learning rate at 0.001. Since it maximizes both macro and micro ROC AUC

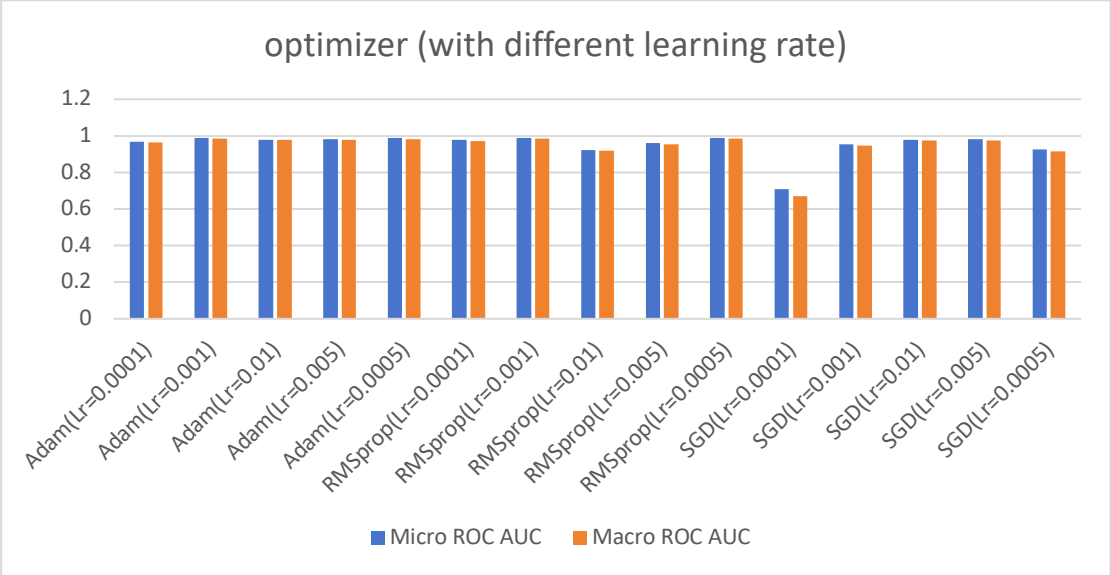

Table 4. Comparison among different optimizer with different learning rates

B. Batch Size

After setting the optimizer properly. The second parameter I adjust is the Batch Size. Batch Size is the number of samples used to train the model in one iteration. It is an important hyperparameter in the training process, and usually choosing the right batch size has a significant impact on the performance of the model and the training efficiency.

I tried four batch size: 64, 128, 256 and 512. The following tables shows the training results:

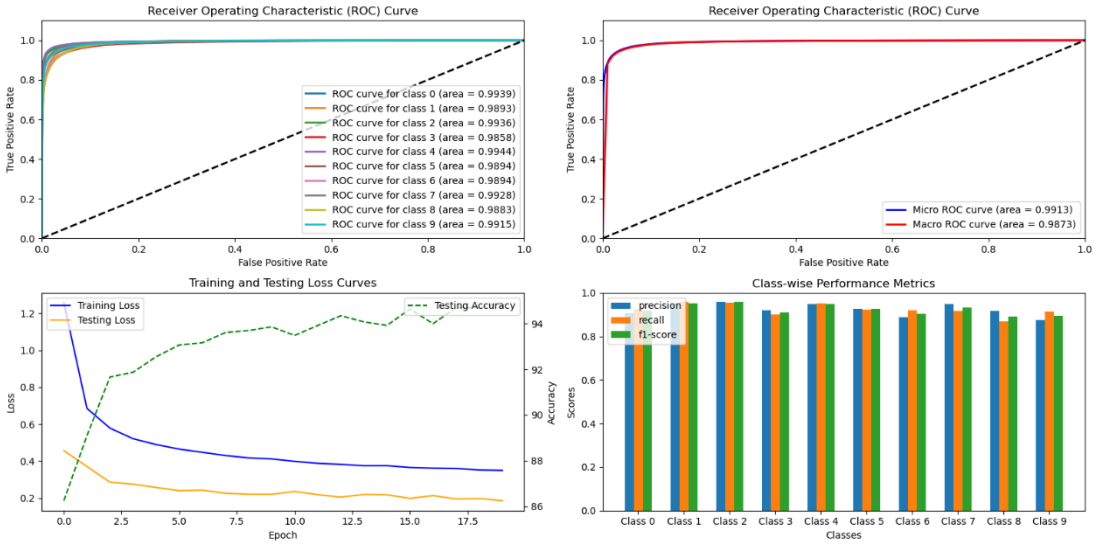

Table 5(A). BatchSize=64

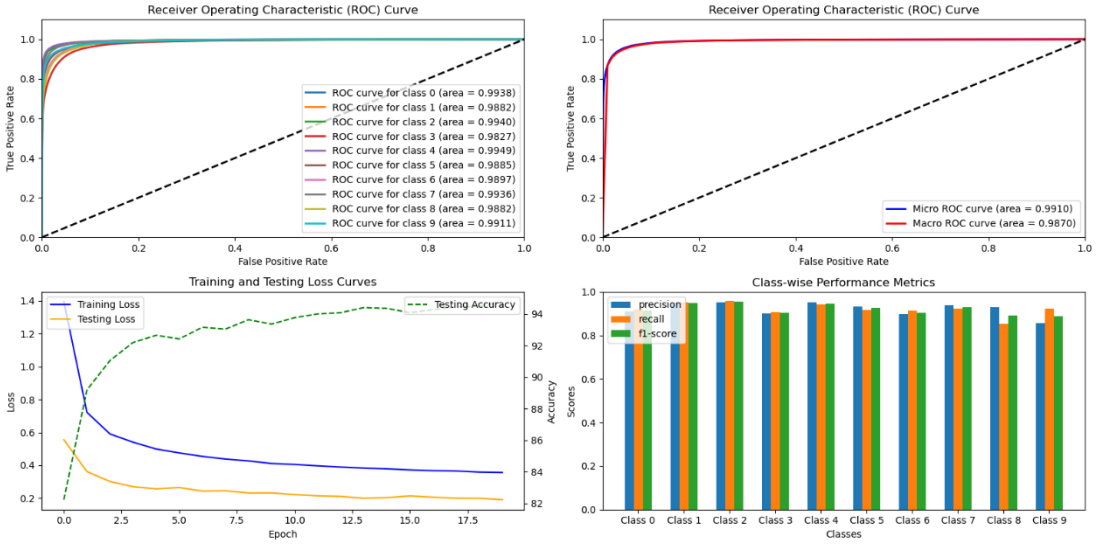

Table 5(B). BatchSize=128

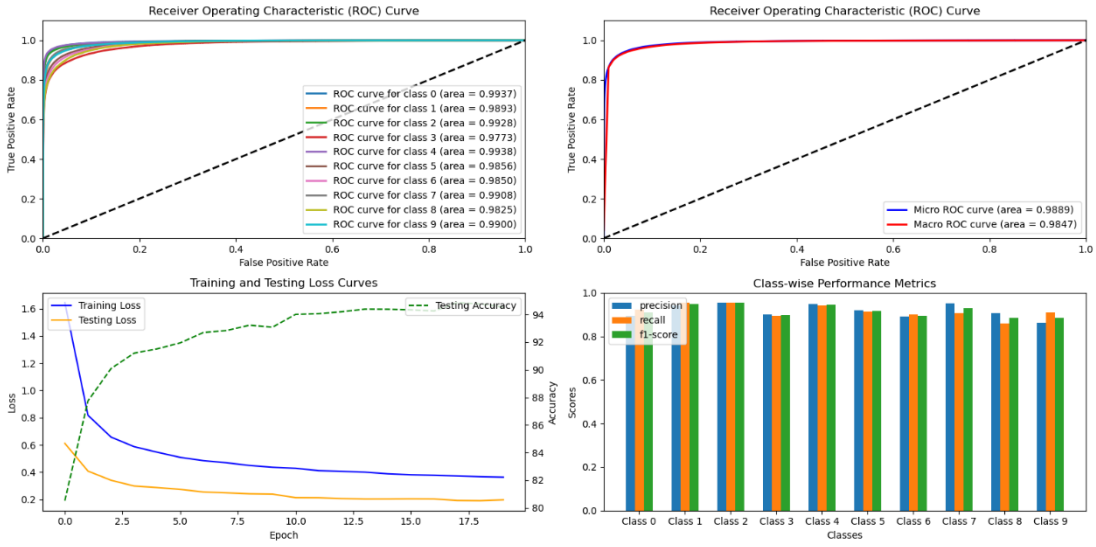

Table 5(C). BatchSize=256

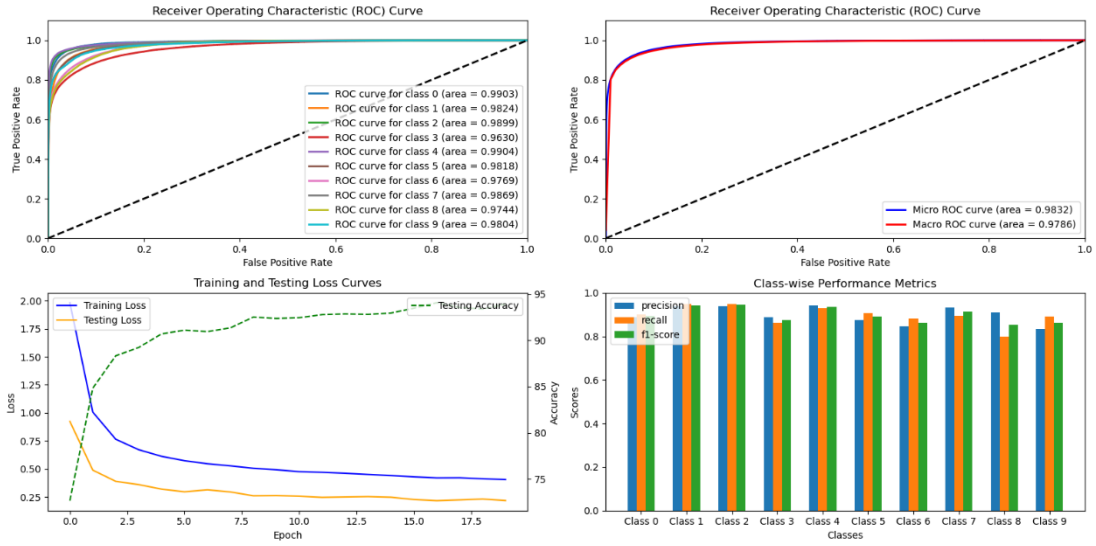

Table 5(D). BatchSize=512

And the Table 6 shows the comparison between different batch sizes. As shown in the table, BatchSize 64 maximize both Micro and Macro ROC AUC and minimize the training and testing losses at 20th epoch.

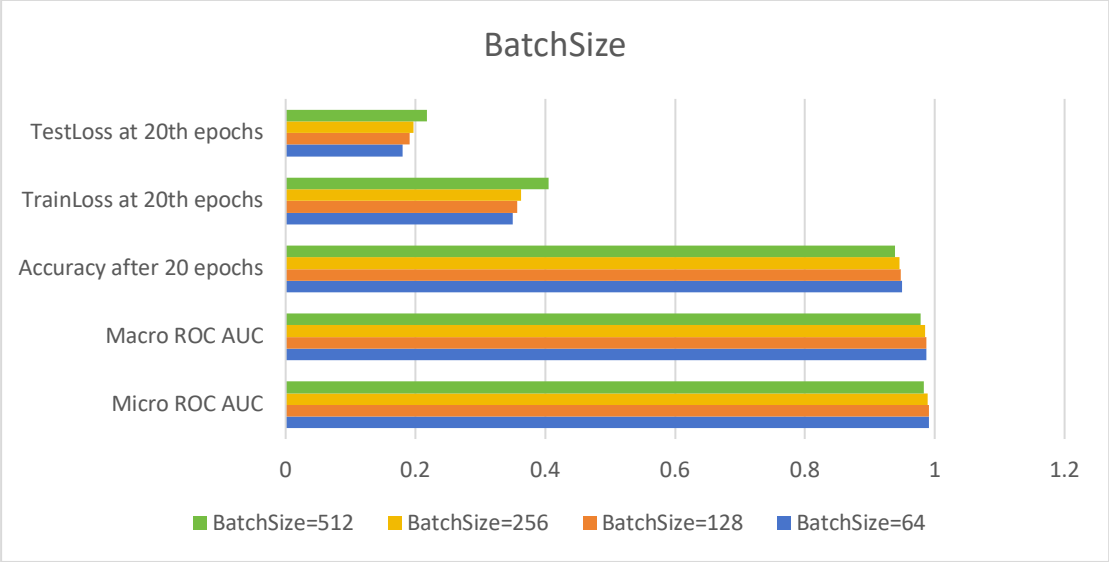

Table 6. Comparison among different BatchSize

C. Maximum Rotation

The maximum Rotation controls the maximum angle of random rotation during the image augmentation. I test with 4 different angles: 15, 30, 45 and 60. The below tables shows the results:

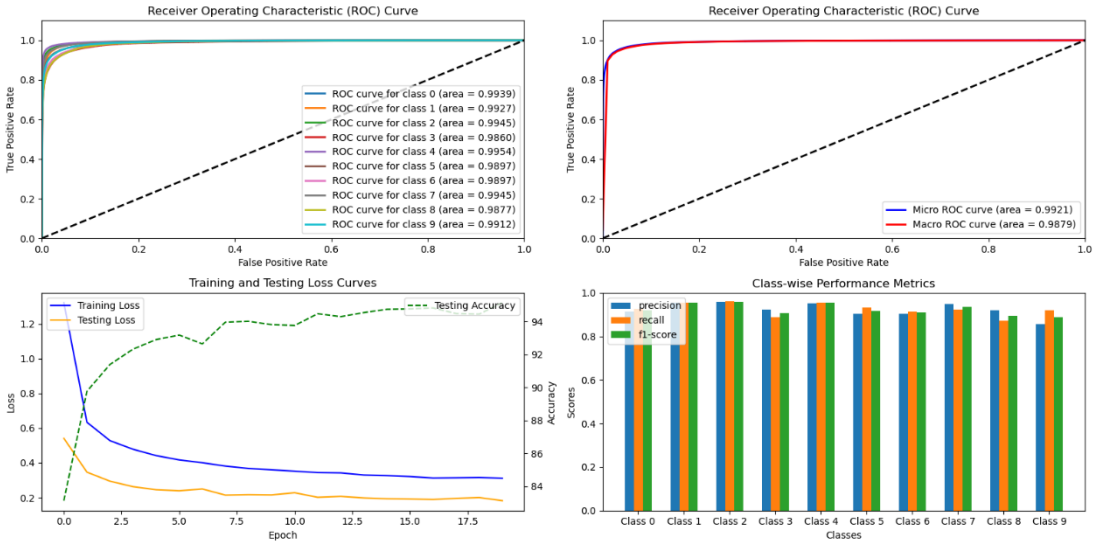

Table 7(A). max_rotation=15

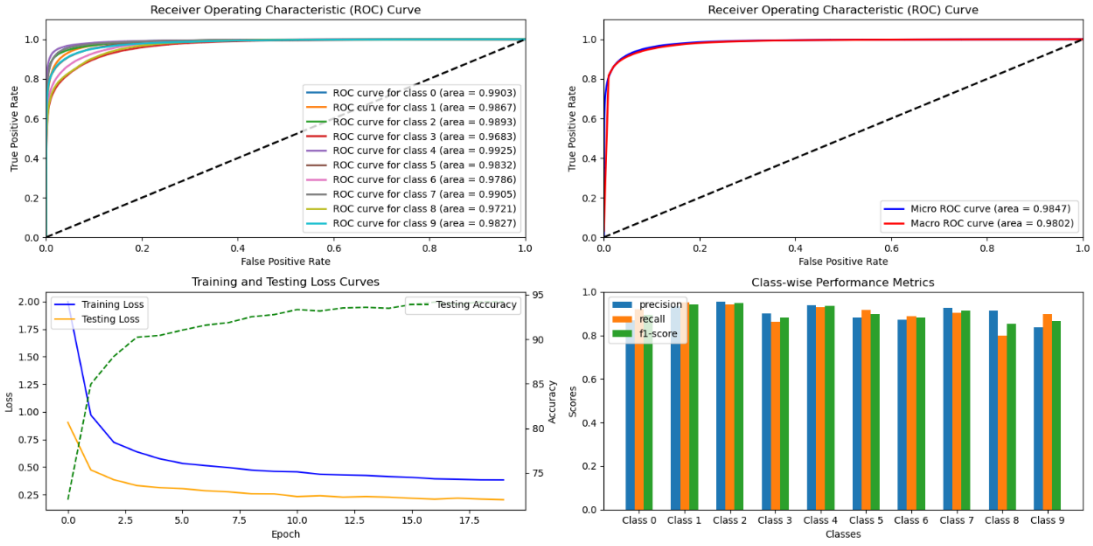

Table 7(B). max_rotation=30

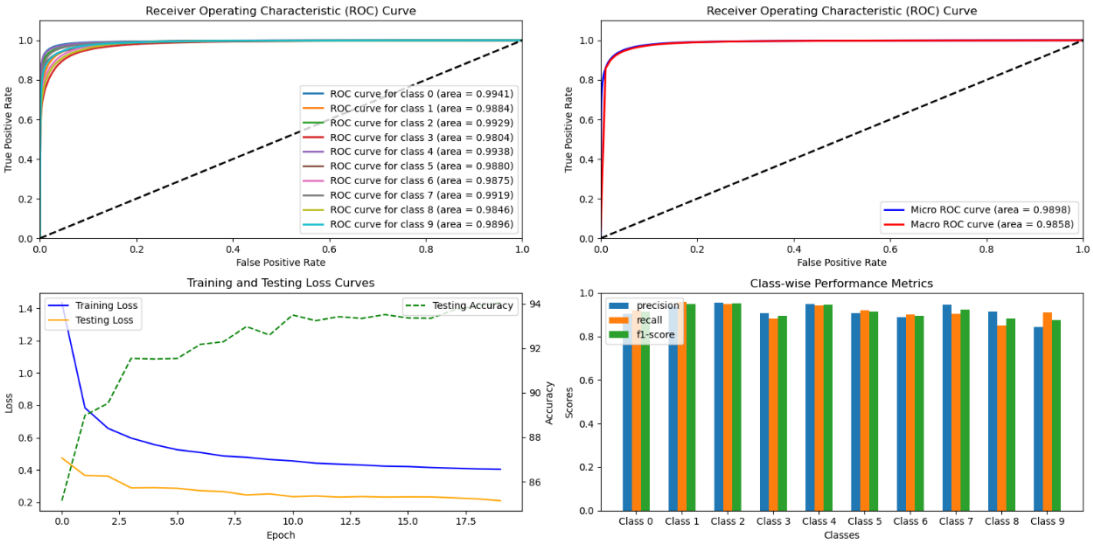

Table 7(C). max_rotation=45

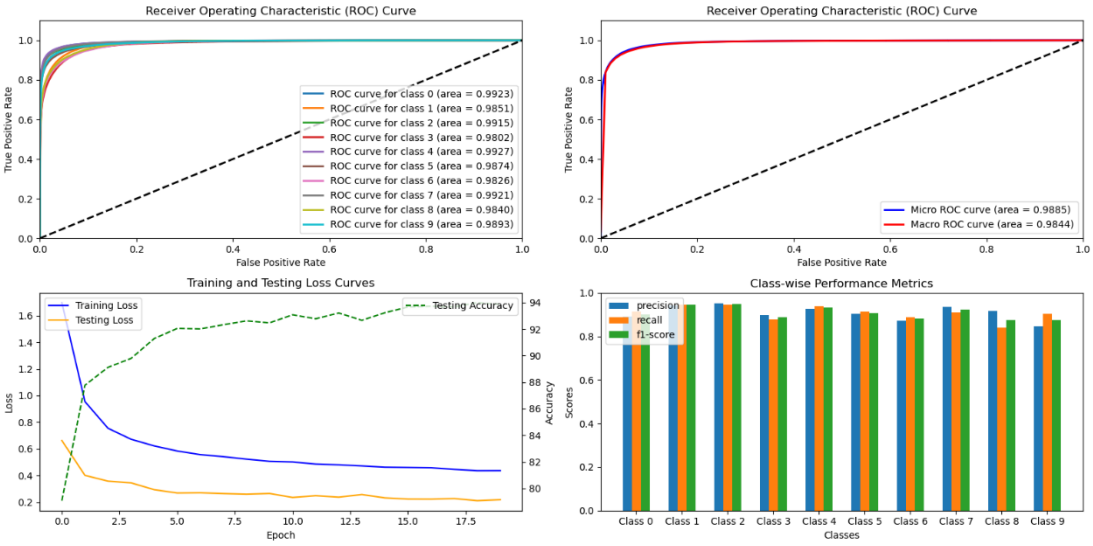

Table 7(D). max_rotation=60

The Table 8 shows the comparison of Micro and Macro ROC AUC among the different max_rotation. As shown in the table, the angle 15 has the highest ROC AUC for both Macro and Micro. Therefore I chose the 15 as the final parameter for max_rotaiton.

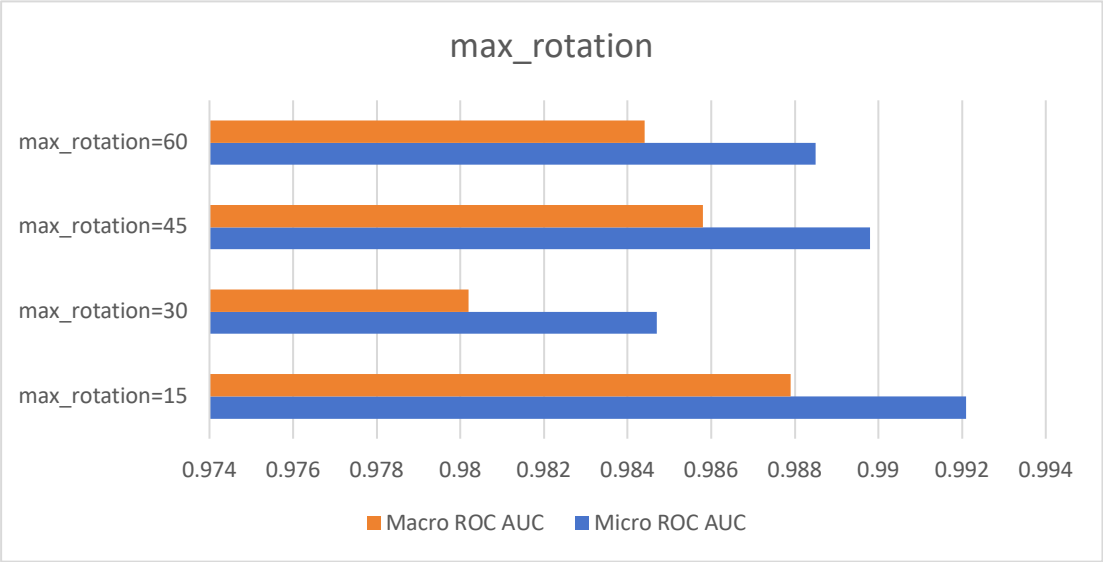

Table 8. Comparison among different max_rotation

D. Minimum Crop Size

The min_crop_size control the Minimum crop size of the random cropping. Different min_crop_size might affect the performance of the model. Consequently, I test four value: 8, 16, 24, 32 to find which one utilizes the model. The below tables shows the results:

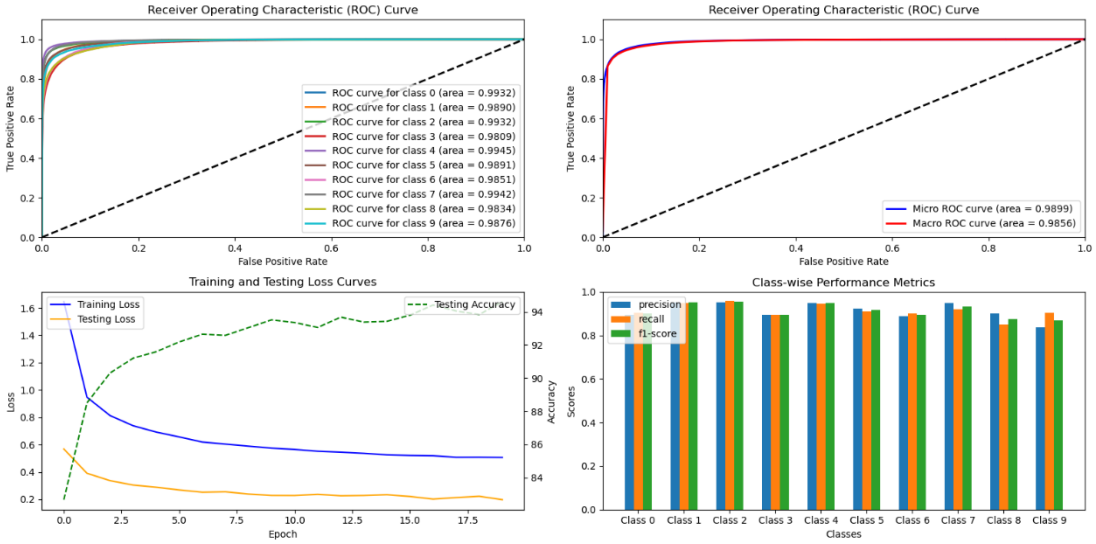

Table 9(A). min_crop_size=8

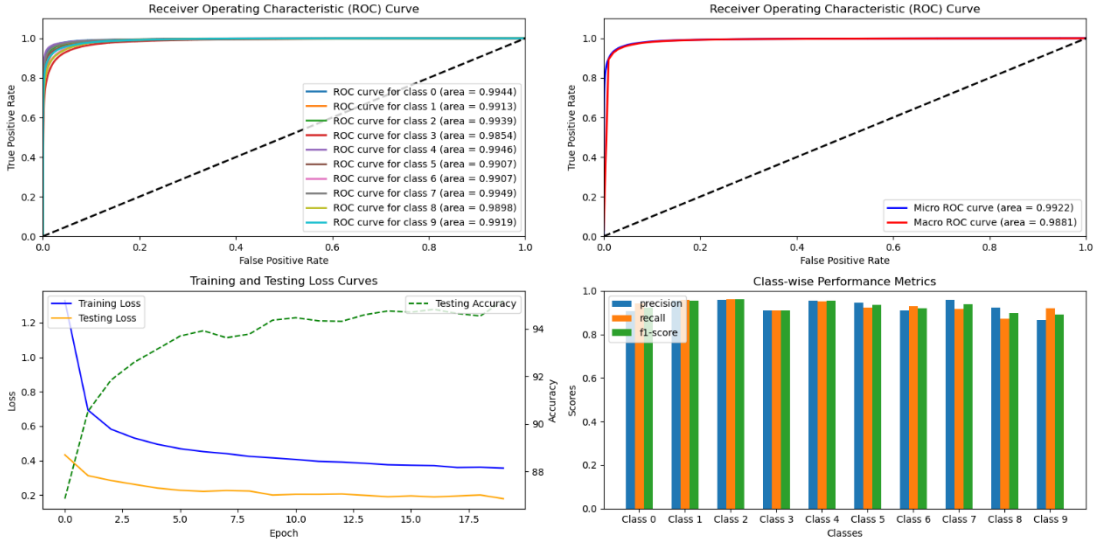

Table 9(B). min_crop_size=16

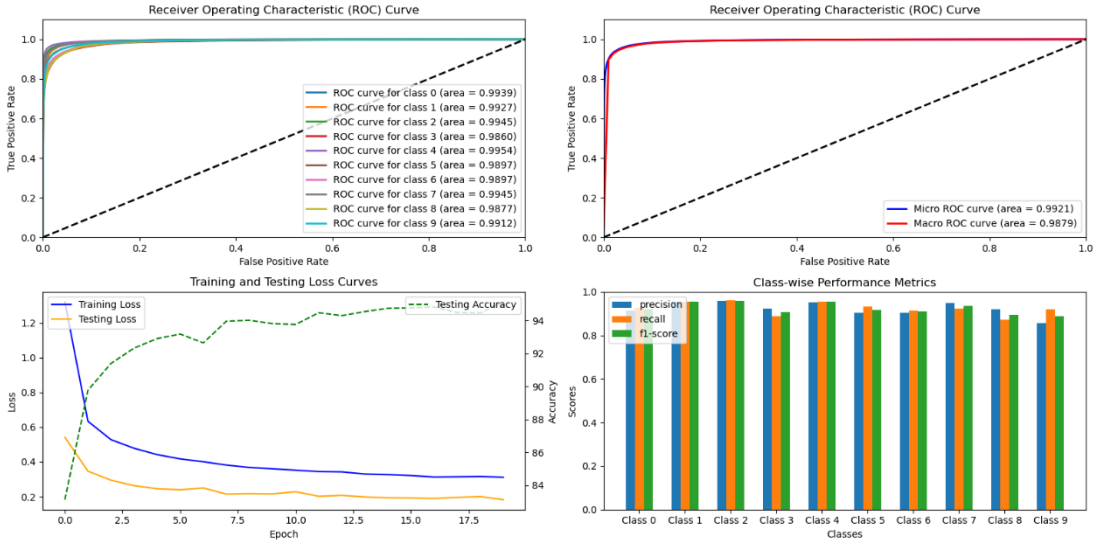

Table 9(C). min_crop_size=24

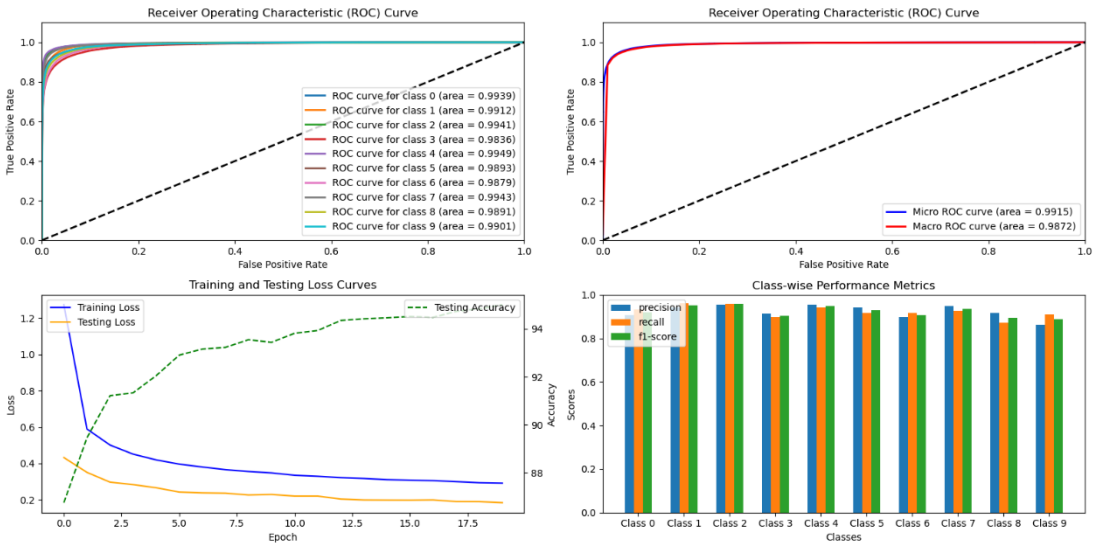

Table 9(D). min_crop_size=32

The Table 10 shows the comparison of Micro and Macro ROC AUC among the different min_crop_size. As shown in the table, size 16 maximize the ROC AUC among these four values. Therefore, I chose the 16 as the final parameter for min_crop_size.

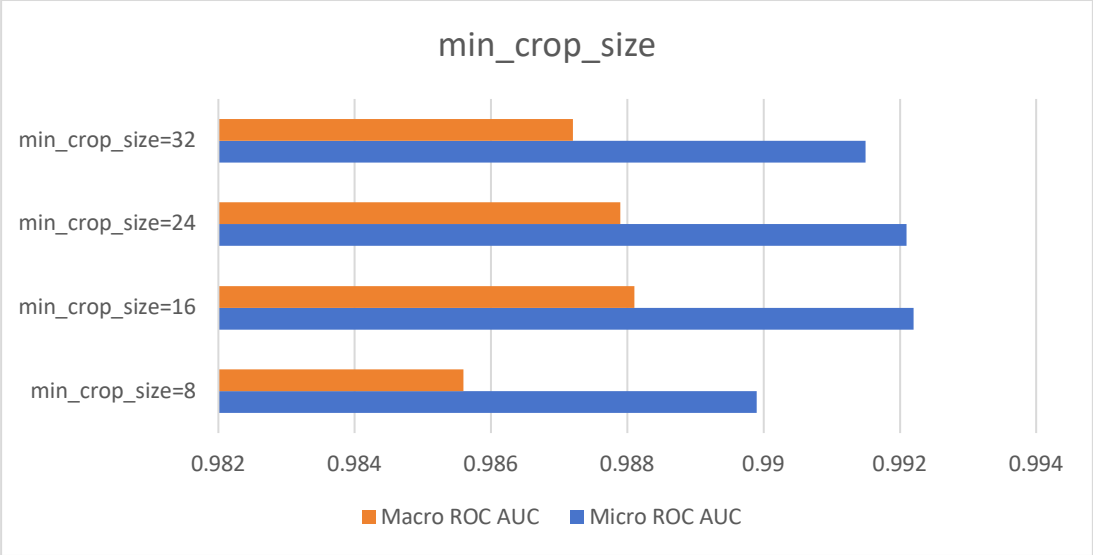

Table 10. Comparison among different min_crop_size

E. Maximum Aspect Ratio Change

max_aspect_ratio_change is a parameter commonly used during image augmentation. It is used to control the maximum aspect ratio change allowed when cropping or transforming an image. The tables below shows the performance of the model at different max_aspect_ratio_change.

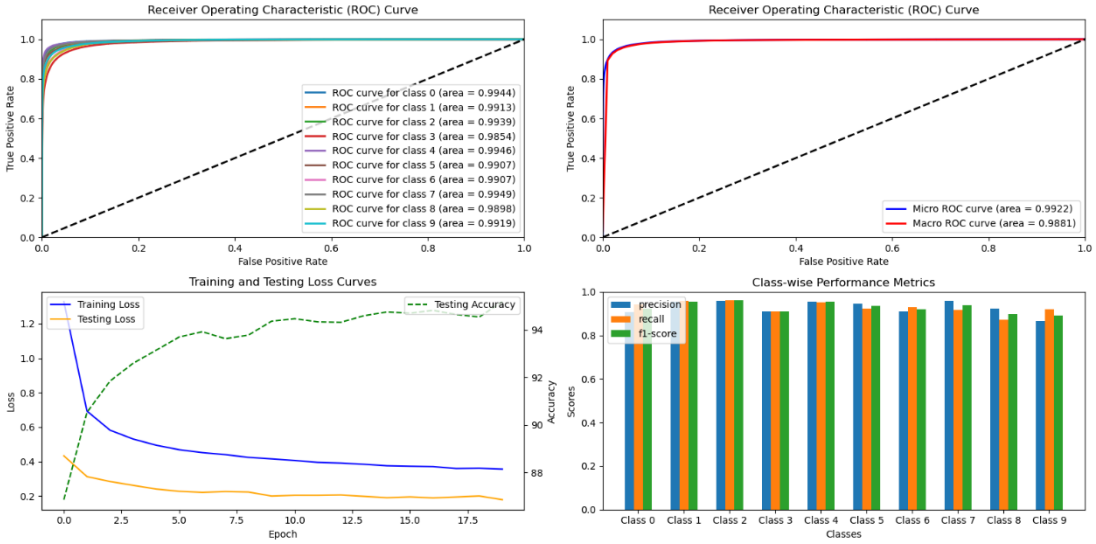

Table 11(A). max_aspect_ratio=0.1

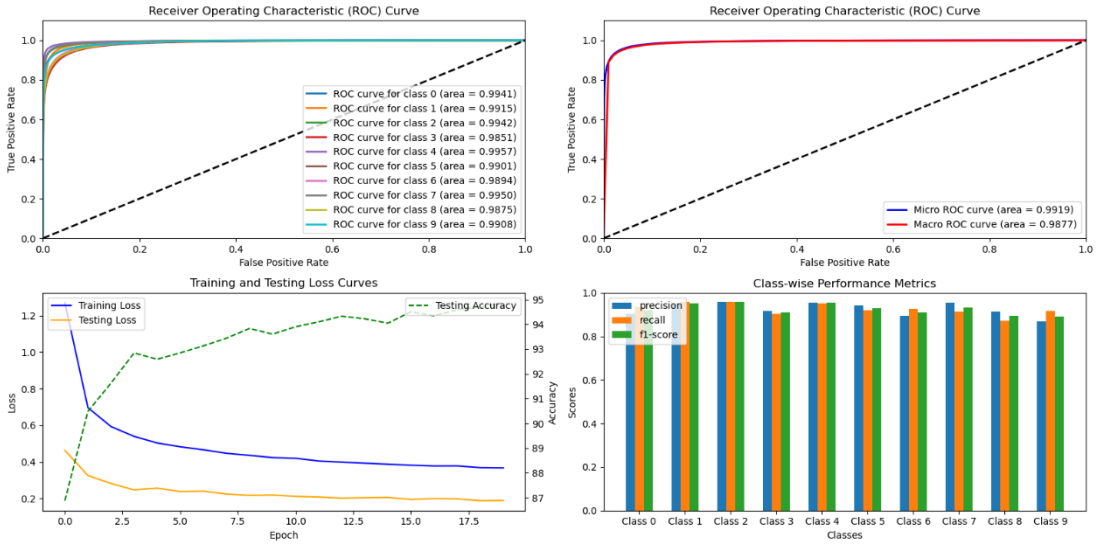

Table 11(B). max_aspect_ratio=0.2

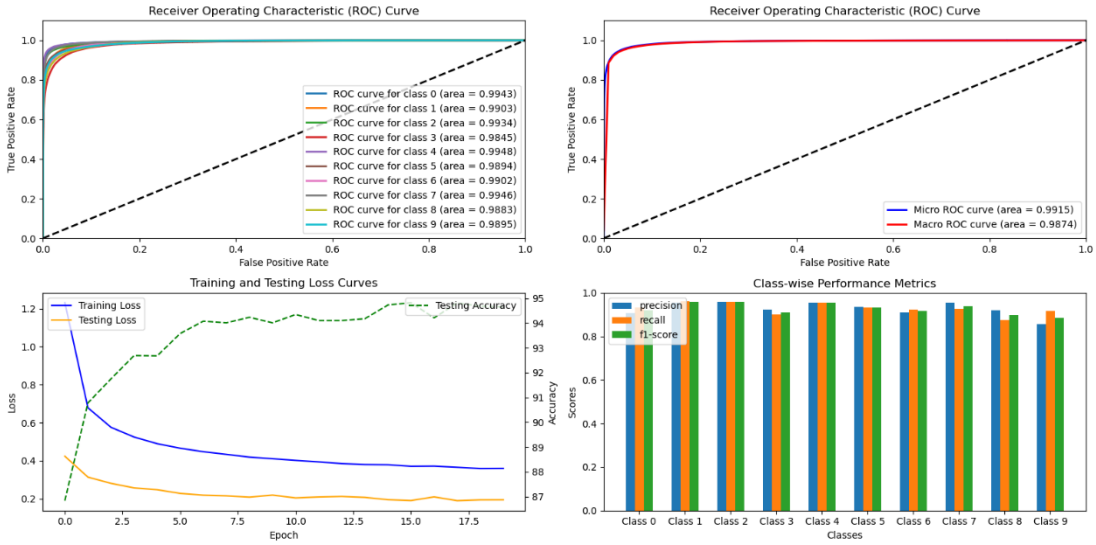

Table 11(C) max_aspect_ratio=0.05

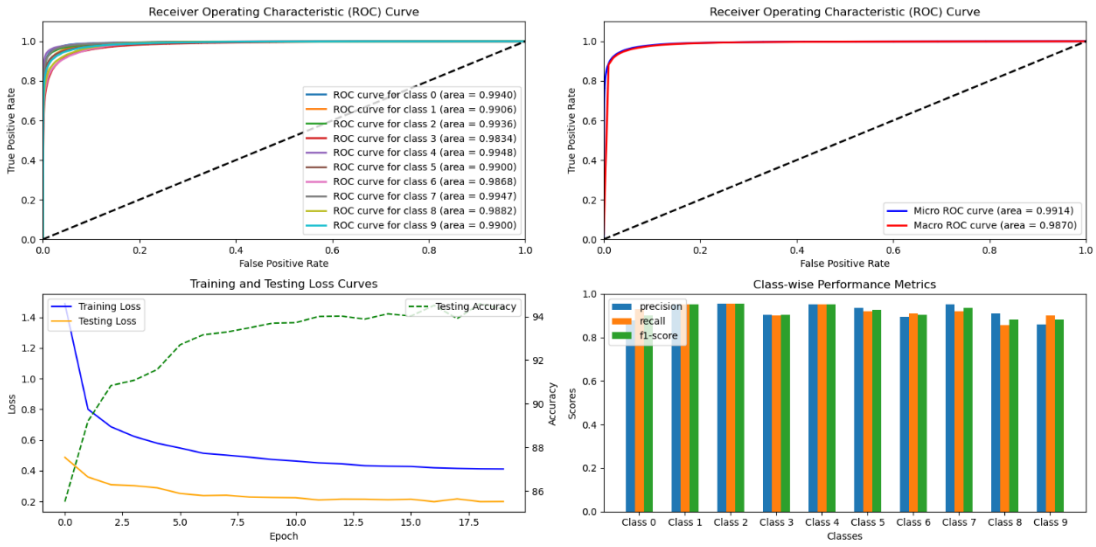

Table 11(D) max_aspect_ratio=0.5

The Table 12 shows the comparison of Micro and Macro ROC AUC among the different max_aspect_ratio_change. As shown in the table, 0.1 maximize the ROC AUC among these four values. Moreover, it also has the best class-wise performance among the four values. Hence, I chose the 0.1

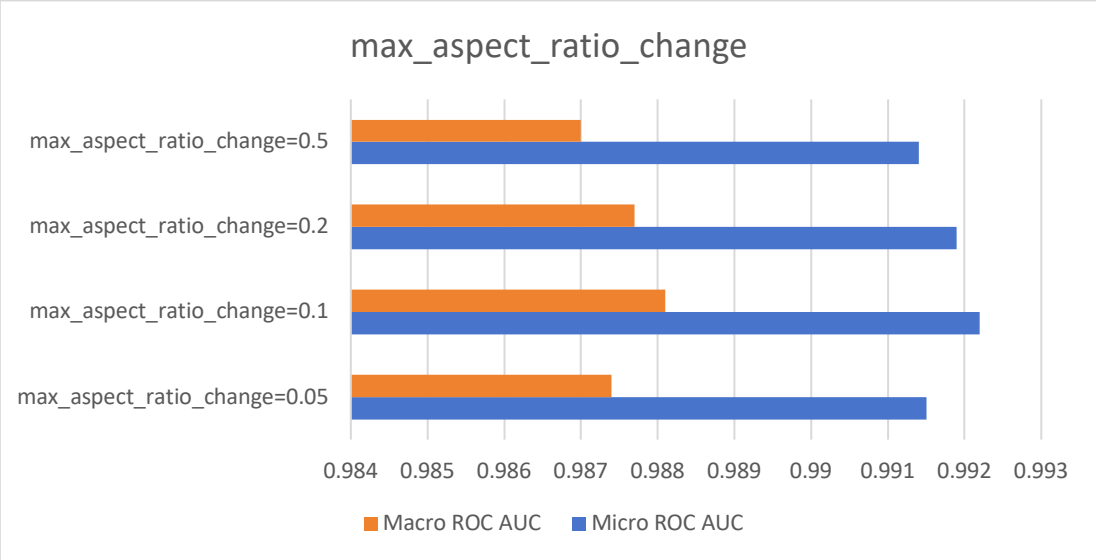

Table 12. Comparison among different max_aspect_ratio_change

F. Number of Epochs

num_epoch is an important hyperparameter in machine learning model training, it represents the number of times the model iterates over the entire training dataset. In other words, it determines the number of times the model sees the training data in its

entirety during the training process. I tested for 20, 50, 100, 200 and 1000 epochs and below tables shows the performance of the model.

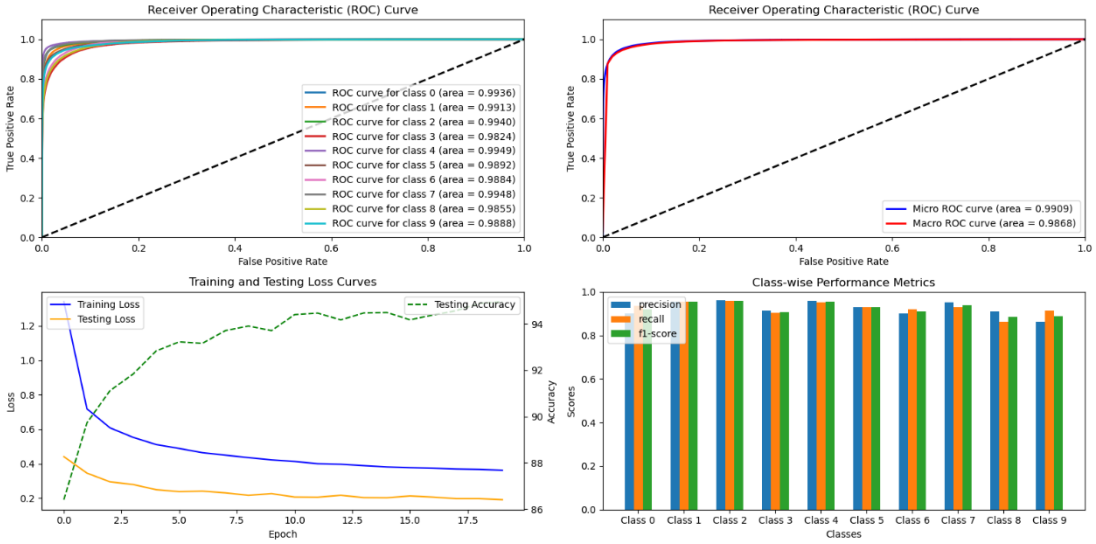

Table 13(A) 20 Epochs

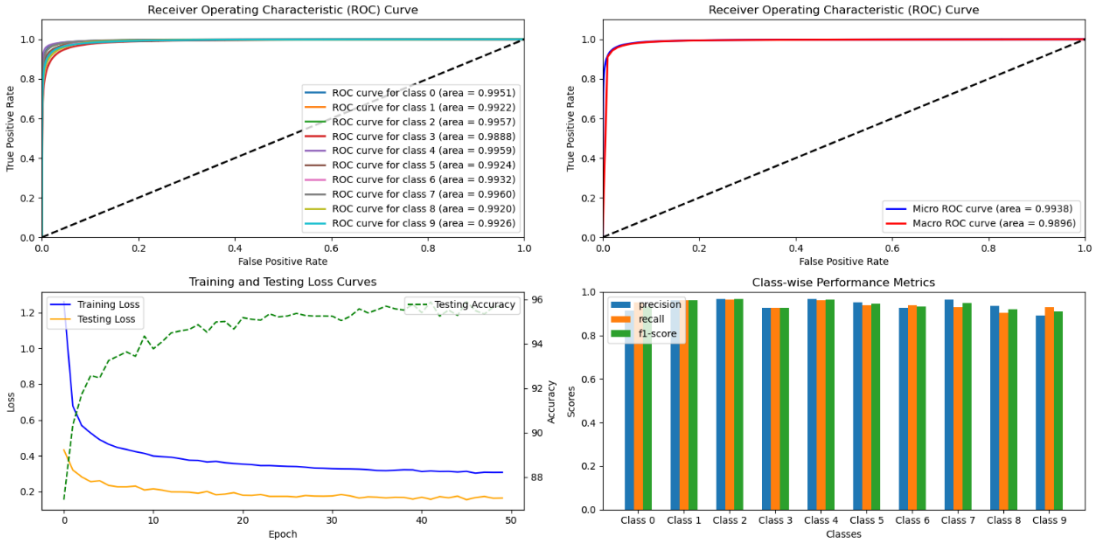

Table 13(B) 50 Epochs

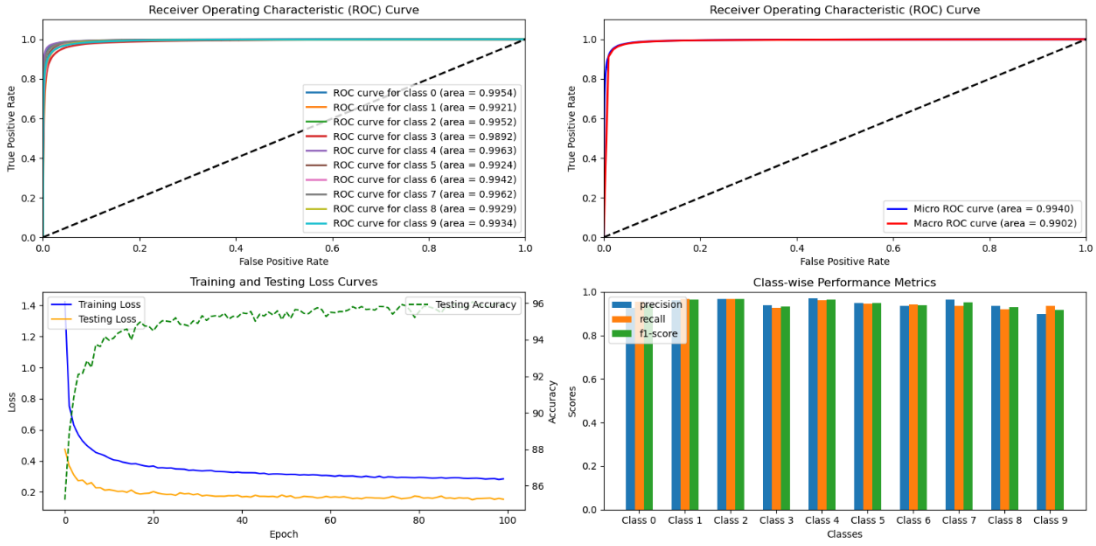

Table 13(C) 100 Epochs

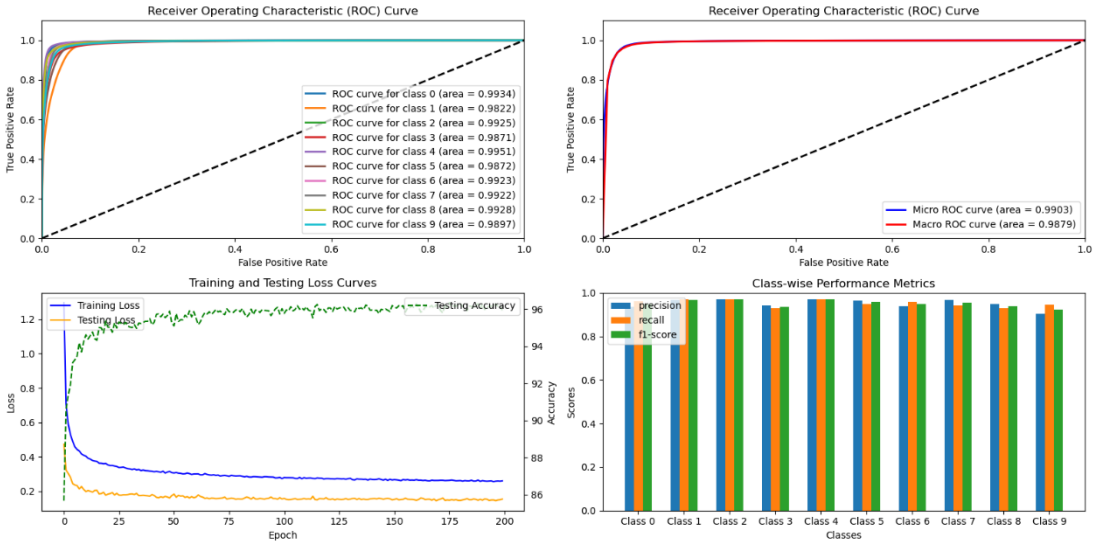

Table 13(D) 200 Epochs

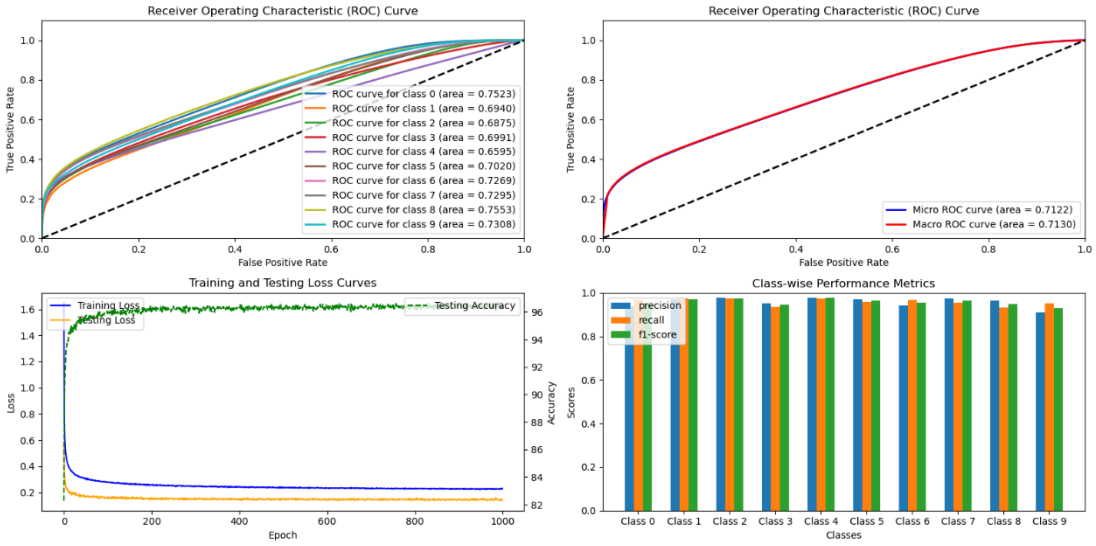

Table 13(E) 1000 Epochs

Although the 1000 Epochs has a better performance in class-wise classification, but it has a low ROC AOC no matter in class-wise, Macro or Micro. And on balance, The 100 epochs has the highest ROC AOC in both Micro and Macro, and has a relatively good class-wise performance. The below Table 14 shows the comparison among the five values.

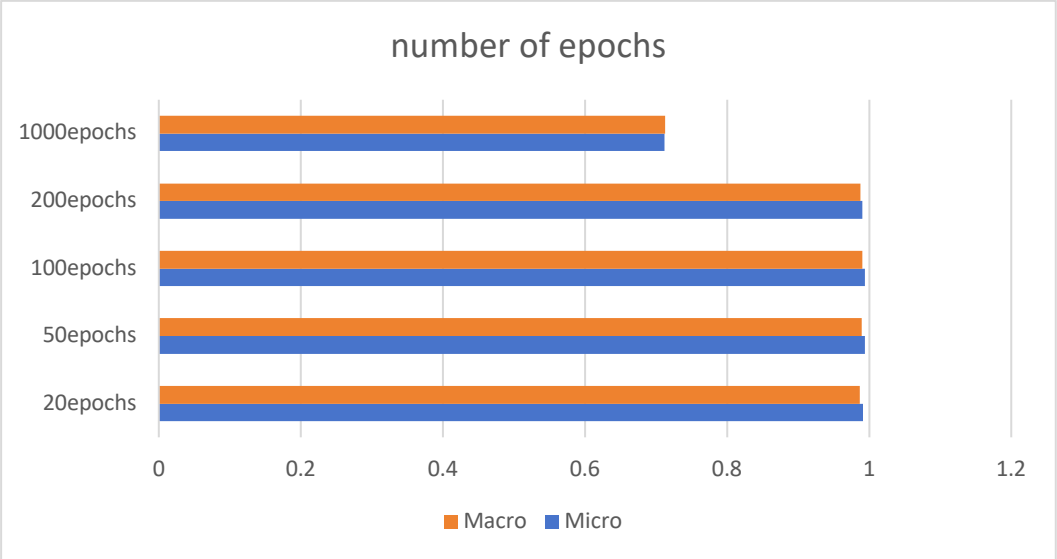

Table 14. Comparison among different number of epochs

Result

The final value of the hyperparameter I mentioned above is:

- Optimizer: Adam optimizer with learning rate at 0.001
- BatchSize: 64
- max_rotation: 15
- min_crop_size: 16
- max_aspect_ratio_change: 0.1
- num_epochs: 100

The below table 15 shows the performance of the model.

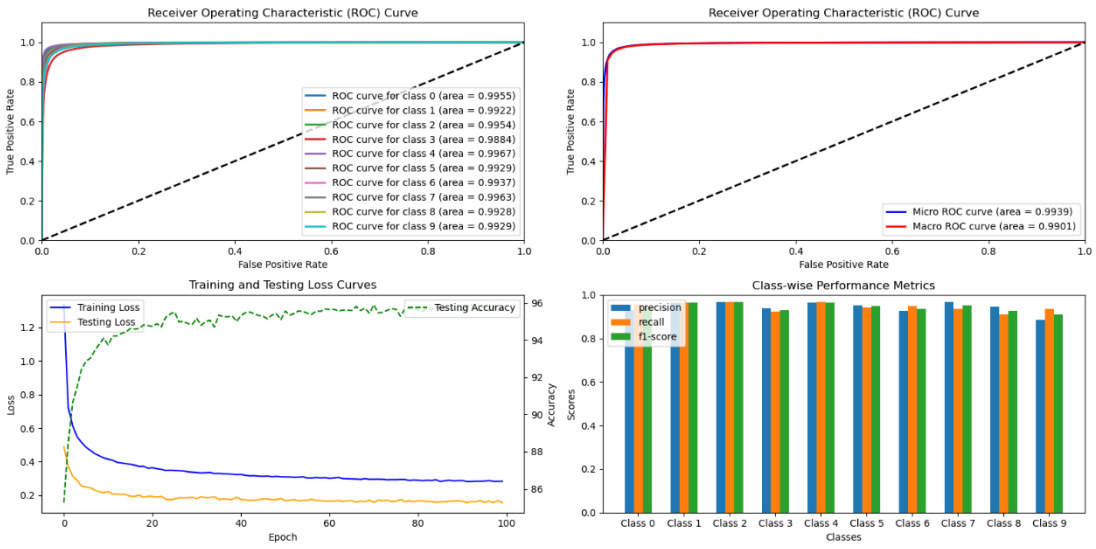

Table 15. The performance of the final model.

Model Demonstration

Also, I implemented a model demonstration function using the Sketchpad in Gradio. It allows user to draw a digit, then the model could output the classification results and the corresponding probabilities for each class. The figure 6 shows a sample of the demo

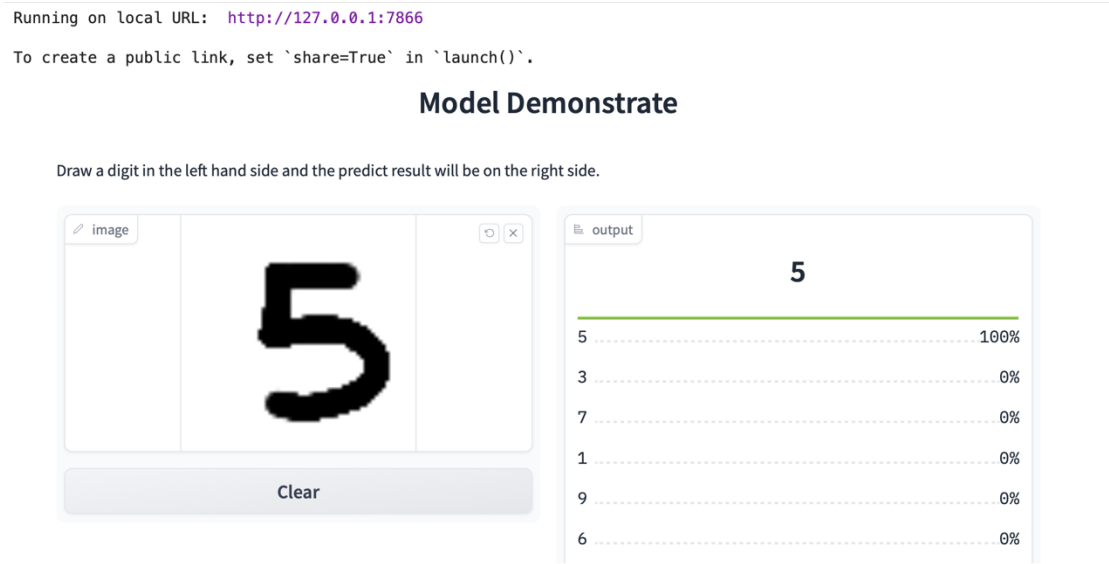

Table 16. Model Demonstration

Conclusion

In this project, I successfully developed a classification network for the Street View House Numbers (SVHN) dataset using a SmallVGG model, tailored for efficiency with fewer parameters. Our methodology included data augmentation techniques to reduce overfitting, using transformations such as random rotations, cropping, and aspect ratio adjustments.

The SmallVGG model was designed with convolutional layers for feature extraction, group normalization for stability, and dropout layers to mitigate overfitting. The network architecture effectively captured the essential features of the SVHN dataset, improving classification accuracy.

Through extensive experimentation, we optimized several hyperparameters, including optimizer types, learning rates, batch sizes, and augmentation settings. The Adam optimizer with a learning rate of 0.001, a batch size of 64, and specific augmentation parameters yielded the best performance. The model achieved high ROC AUC scores, indicating robust predictive capability.

Our final model demonstrates strong performance and generalization in digit classification, validated by training and evaluation. Furthermore, the implementation of a Gradio-based demonstration allows for interactive testing of the model.

Overall, this project highlights the effectiveness of a compact VGG model and strategic data augmentation in achieving high accuracy in digit classification tasks. Future work may explore further applications of the model.