CISC3024 Pattern Recognition Final Project

Group Members:

- Huang Yanzhen, DC126732
- Mai Jiajun, DC127853

0. Project Setup

0.1 Packages & Device ¶

[6]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader, Subset
from tqdm import tqdm

import numpy as np
import cv2
import os
import time
```

[7]:
```
device_name = "cuda" if torch.cuda.is_available() else "cpu"
device = torch.device(device_name)
print(f"Using device: {device_name}")
```

```
Using device: cuda
```

0.2 Global Configurations

[8]:
```
path_dataset = "./data/SVHN_mat"
norm_mean = [0.4377, 0.4438, 0.4728]
norm_std = [0.1980, 0.2010, 0.1970]
```

1. Data Processing and Augmentation

1.1 Download Datasets

Define dataset class, retrieve dataset.

[9]:
```
import albumentations as A
from albumentations.pytorch import ToTensorV2
import scipy.io as sio
```

[10]:
```
class SVHNDataset(Dataset):
    def __init__(self, mat_file, transform=None):
        data = sio.loadmat(mat_file)
        self.images = np.transpose(data['X'], (3, 0, 1, 2))
        self.labels = data['y'].flatten()
        self.labels[self.labels == 10] = 0
        self.transform = transform

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

    def __getitem__(self, idx):
        image = self.images[idx]
        label = self.labels[idx]

        if self.transform:
            image = self.transform(image=image)['image']

        return image, label
```

[11]:
```
transform = A.Compose([
    A.RandomResizedCrop(32, 32),
```

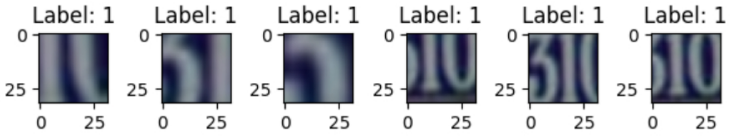
```
    A.Rotate(limit=30),
    A.Normalize(mean=norm_mean, std=norm_std),
    ToTensorV2()
])

train_dataset = SVHNDataset(mat_file=os.path.join(path_dataset,"train_32x32.mat"), transform=transform)
test_dataset = SVHNDataset(mat_file=os.path.join(path_dataset,"test_32x32.mat"), transform=transform)
extra_dataset = SVHNDataset(mat_file=os.path.join(path_dataset,"extra_32x32.mat"), transform=transform)

train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False)
extra_loader = DataLoader(Subset(extra_dataset, indices=list(range(30000))), batch_size=64, shuffle=False)

print(f"Train Size:{train_dataset.__len__()}\nTest Size:{test_dataset.__len__()}\nExtra Size:{extra_dataset.__len__()}")
```

```
Train Size:73257
Test Size:26032
Extra Size:531131
```

1.2 Peak A Data

[12]:
```
import random
```

[9]:
```
def unnormalize(img, mean, std):
    """Revert the normalization for visualization."""
    img = img * std + mean
    return np.clip(img, 0, 1)

# Plotting multiple images in a grid
grid_rows, grid_cols = 1, 6

fig, axes = plt.subplots(grid_rows, grid_cols, figsize=(6, 6))

peak_index = random.randint(0, train_dataset.__len__()-1)

for i in range(grid_cols):
    img_tensor, label = train_dataset.__getitem__(peak_index)
    img = img_tensor.permute(1, 2, 0).numpy()  # Convert to (H, W, C)
    img = unnormalize(img, norm_mean, norm_std)

    ax = axes[i]  # Get subplot axis
    ax.imshow(img)
    ax.set_title(f"Label: {label}")

plt.tight_layout()
plt.show()

print(f"Peaking data from training set of index {peak_index}.\nImage Tnesor Size:{train_dataset.__getitem__(peak_index)[0].shape}")
```




```
Peaking data from training set of index 48643.
Image Tnesor Size:torch.Size([3, 32, 32])
```

2. Neuron Network Structure

2.1 Specify Model Structure

[13]:
```
class SmallVGG(nn.Module):
    def __init__(self, frame_size=32):
        super(SmallVGG, self).__init__()
        self.frame_size = frame_size
        self. conv_layers = nn.Sequential(
            nn.Conv2d(3, 8, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(8, 16, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2), # 16x16

            nn.Conv2d(16, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2), # 8x8

            nn.Conv2d(32, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2), # 4x4
```

```
        )

        self.fc_layers = nn.Sequential(
            nn.Linear(frame_size * 4 * 4, 256),
            nn.ReLU(),
            nn.Linear(256, 10)
        )

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

2.2 Initialize with Hyper Parameters

[14]:
```
num_epochs = 30
learning_rate = 0.001
model = SmallVGG().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

2.3 Train and Evaluate

[15]:
```
def train_and_evaluate(model,
                       train_loader,
                       test_loader,
                       criterion,
                       optimizer,
                       num_epochs=100):
    # Record Losses to plot
    train_losses = []
    test_losses = []

    for epoch in range(num_epochs):
        # Train
        model.train()
        running_loss = 0.0
        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() * len(images)
        train_losses.append(running_loss / len(train_loader))

        # Evaluate
        model.eval()
        test_loss = 0.0
        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() * len(images)

        test_losses.append(test_loss / len(test_loader))
        print(f"Epoch[{epoch+1}/{num_epochs}], Train Loss:{train_losses[-1]:.4f}, Test Loss:{test_losses[-1]:.4f}")

    return train_losses, test_losses
```

•[12]:
```
train_losses, test_losses = train_and_evaluate(model, train_loader, test_loader, criterion, optimizer, num_epochs)
torch.save(model.state_dict(), f"./models/small_vgg_ne-{num_epochs}_lr-{learning_rate:.0e}.pth")
```

```
100%|██████████| 573/573 [00:18<00:00, 31.35it/s]
Epoch[21/30], Train Loss:106.0726, Test Loss:99.3662
100%|██████████| 573/573 [00:17<00:00, 32.34it/s]
Epoch[22/30], Train Loss:105.9276, Test Loss:100.2521
100%|██████████| 573/573 [00:18<00:00, 31.73it/s]
Epoch[23/30], Train Loss:104.3199, Test Loss:99.2494
100%|██████████| 573/573 [00:17<00:00, 32.52it/s]
Epoch[24/30], Train Loss:104.3040, Test Loss:100.1059
100%|██████████| 573/573 [00:17<00:00, 32.01it/s]
Epoch[25/30], Train Loss:104.2286, Test Loss:97.9395
```

SHOW MORE OUTPUTS

2.4 Visualize Result

Multiple functions are defined to evaluate data. Below is a list of them.

`get_predictions`

Get a model prediction on 30,000 data on `extra.mat`.

- params:
  - `model_path`: The directory at which the model is stored.
  - `extra_loader`: The `DataLoader` object for loading `extra.mat`.
- returns:
  - `pred_scores`: A list of softmax-normalized predictions, one list of probabilities per image.
  - `true_labels`: A list of ground-truth labels.
  - `pred_labels`: A list of predicted labels.

`display_cm`

Plots the Confusion Matrix.

- params:
  - `true_labels`: A list of ground-truth labels.
  - `pred_labels`: A list of predicted labels.
- returns: `None`

`get_metrics`

Get evaluation metrics based on the given prediction results.

- params:
  - `true_labels`: A list of ground-truth labels.
  - `pred_labels`: A list of predicted labels.
- returns:
  - `accuracy`: A single universal accuracy. $acc = \frac{\#.\text{ of correctly classified data}}{\#.\text{ of data}}$
  - `precision`: A class-wise list of classification precisions. $P_i = \frac{TP}{TP+FP}$
  - `recall`: A class-wise list of classification recalls. $R_i = \frac{TP}{TP+FN}$
  - `f1`: A class-wise list of harmonic mean of precision and recall. $F_1 = \frac{2}{\frac{1}{P}+\frac{1}{R}}$

`display_pr_curve`

Plot the precision-recall curve.

- params:
  - `true_labels`: A list of ground-truth labels.
  - `pred_scores`: A list of softmax-normalized predictions, one list of probabilities per image.
- returns: `None`

`get_roc_auc`

Get the `roc_auc` scores.

- params:
  - `true_labels`: A list of ground-truth labels.
  - `pred_scores`: A list of softmax-normalized predictions, one list of probabilities per image.
- returns:
  - `roc_auc`: A class-wise list of `roc_auc` scores.

`display_roc_auc`

Plot the `roc` curve.

- params:
  - `true_labels`: A list of ground-truth labels.
  - `pred_scores`: A list of softmax-normalized predictions, one list of probabilities per image.
- returns: `None`

[16]:
```
from sklearn.metrics import (confusion_matrix, accuracy_score,
                             precision_score, recall_score,
                             f1_score, roc_auc_score,
                             roc_curve, auc, precision_recall_curve,
```

```
                          average_precision_score)
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.preprocessing import label_binarize
```

2.4.0 Predict with Extra Data

First, use the trained model to predict 30,000 image data from `extra.mat`, which is unseen in neither `train.mat` nor `test.mat`. Robustness on `extra` proves the usability of the model.

Results may contain the following:

- Epoch-Loss Curve of model training
- Confusion Matrix
- Accuracy, Precision, Recall and F_1 score.

[18]:
```
def get_predictions(model_path, extra_loader):
    if not isinstance(model_path, str):
        model_state = model_path
    else:
        model_state = torch.load(model_path)
    model = SmallVGG()
    model.load_state_dict(model_state)

    model.to(device)
    model.eval()

    pred_scores = []  # Prob. of predictions
    true_labels = []  # Ground Truth
    pred_labels = []  # Label of prediction, i.e., argmax(softmax(pred_scores))

    with torch.no_grad():
        for images, labels in tqdm(extra_loader):
            images, labels = images.to(device), labels.to(device)

            outputs = model(images)

            pred_scores_batch = nn.functional.softmax(outputs, dim=-1)

            pred_scores.extend(pred_scores_batch.cpu().tolist())
            pred_labels.extend(outputs.argmax(dim=1).tolist())
            true_labels.extend(labels.cpu().tolist())

    return pred_scores, true_labels, pred_labels
```

[15]:
```
pred_scores, true_labels_cpu, pred_labels_cpu = get_predictions("./models/small_vgg_ne-30_lr-1e-03.pth", extra_loader)
print("First 100 true labels:")
[print(num, end=" ") for num in true_labels_cpu[:100]]
print("...\n")

print("First 100 predictions:")
[print(num, end=" ") for num in pred_labels_cpu[:100]]
print("...\n")

print("Prediction Probabilities:")
[print(arr) for arr in pred_scores[:5]]
print("...")
```

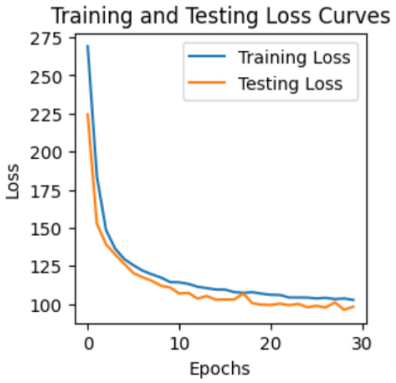
```
D:\Temps\temp\ipykernel_29292\3475512123.py:2: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future release, the default value for `weights_only` will be flipped to `True`. This limits the functions that could be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded via this mode unless they are explicitly allowlisted by the user via `torch.serialization.add_safe_globals`. We recommend you start setting `weights_only=True` for any use case where you don't have full control of the loaded file. Please open an issue on GitHub for any issues related to this experimental feature.
  model_state = torch.load(model_path)
100%|██████████| 469/469 [00:07<00:00, 64.63it/s]
First 100 true labels:
4 7 8 7 1 1 7 4 3 0 2 8 8 3 1 1 7 0 8 1 5 6 4 4 4 6 3 4 4 3 0 1 7 6 0 1 1 0 5 7 5 1 8 5 5 2 9 6 1 5 2 3 5 3 6 9 2 3 4 1 7 7 3 1 2 2 0 1 1 3 1 5 1 1 9 9 4 8 0 5 1 3 8 2 9 5 6 0 7 8 3 0 6 4 0 3 1 1 0 0 ...

First 100 predictions:
4 1 8 7 7 1 7 4 3 0 2 8 0 3 1 1 7 0 8 1 6 6 4 4 4 6 3 4 4 3 0 7 7 6 0 1 1 0 2 7 5 1 4 5 3 2 9 6 1 4 6 3 5 3 6 7 2 3 4 1 7 7 3 1 2 2 2 1 1 3 1 5 1 1 9 9 4 3 0 5 1 3 9 2 0 6 6 0 7 8 3 1 6 4 0 7 1 1 1 8 ...

Prediction Probabilities:
[7.564401312265545e-05, 0.001672823098488152, 0.0010354656260460615, 0.0033441961277276278, 0.9874839186668396, 0.0007910538115538657, 0.000917234749067574, 0.0022566476836800575, 0.0016073824372142553, 0.0008156428812071681]
[0.00012675659672822803, 0.9644230008125305, 0.0007670297054573894, 0.00124127056915313, 0.0025483234785497 19, 6.851315993117169e-05, 0.000612345582339912, 0.029283843934535 98, 0.0005660591996274889, 0.0003628291597124189]
[2.653884045855648e-08, 1.8440124449625728e-06, 1.6757052208049572e-06, 3.0323761166073382e-05, 2.5218845678409707e-08, 2.2668581323159742e-07, 1.681719186308328e-05, 1.955303119416385e-08, 0.9999483823776245, 6.80142079545476e-07]
[6.54899122309871e-05, 0.0005002619000151753, 4.679286575992592e-05, 3.6798678593186196e-07, 6.716557254549116e-05, 4.460994418309383e-08, 1.727928065520245 6e-05, 0.9992994070053101, 1.5789655662956648e-06, 1.5131564623516 17e-06]
[0.1745602786540985, 0.27699077129364014, 0.006574051920324564, 0.0015957633731886744, 0.002193179912865162, 4.513328894972801e-05, 0.0025927152018994093, 0.5319300293922424, 0.0008856524946168065, 0.0026323720812797546]
...
```

2.4.1 Epoch-Loss Curves

2.4.1 Epoch-Loss Curves

[19]:
```
def display_el_curve(train_losses, test_losses):
    plt.figure(figsize=(3,3))
    plt.plot(train_losses, label="Training Loss")
    plt.plot(test_losses, label="Testing Loss")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.title("Training and Testing Loss Curves")
    plt.legend()
    plt.show()
```

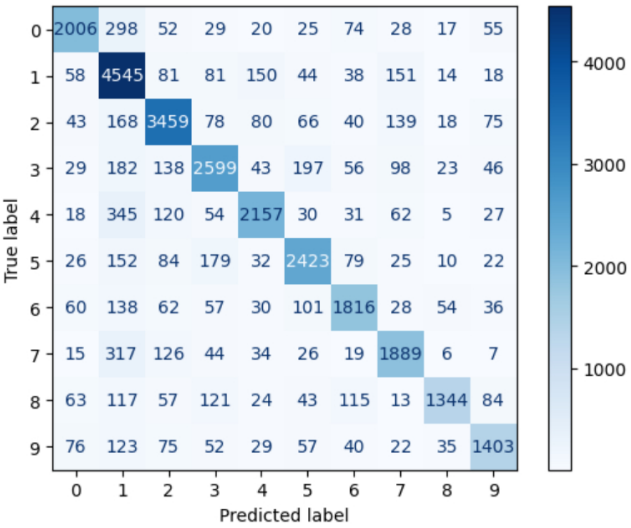
[17]:
```
display_el_curve(train_losses, test_losses)
```



2.4.2 Confusion Matrix

[20]:
```
def display_cm(true_labels, pred_labels):
    cm = confusion_matrix(true_labels, pred_labels)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=range(0,10))
    disp.plot(cmap=plt.cm.Blues)
    plt.show()
```

[19]:
```
display_cm(true_labels_cpu, pred_labels_cpu)
```



2.4.3 Accuracy, Precision, Recall and F_1 Score

[20]:
```
true_labels_bin = label_binarize(true_labels_cpu, classes=range(0,10))
true_labels_bin
```

[20]:
```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 1, 0, 0],
       [0, 0, 0, ..., 0, 1, 0],
       ...,
       [1, 0, 0, ..., 0, 0, 0],
       [1, 0, 0, ..., 0, 0, 0],
       [0, 0, 1, ..., 0, 0, 0]])
```

[21]:
```
pred_labels_bin = label_binarize(pred_labels_cpu, classes=range(0,10))
pred_labels_bin
```

[21]:
```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 1, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 1, 0],
       ...,
       [1, 0, 0, ..., 0, 0, 0],
```

```
       [0, 1, 0, ..., 0, 0, 0],
       [0, 0, 1, ..., 0, 0, 0]])
```

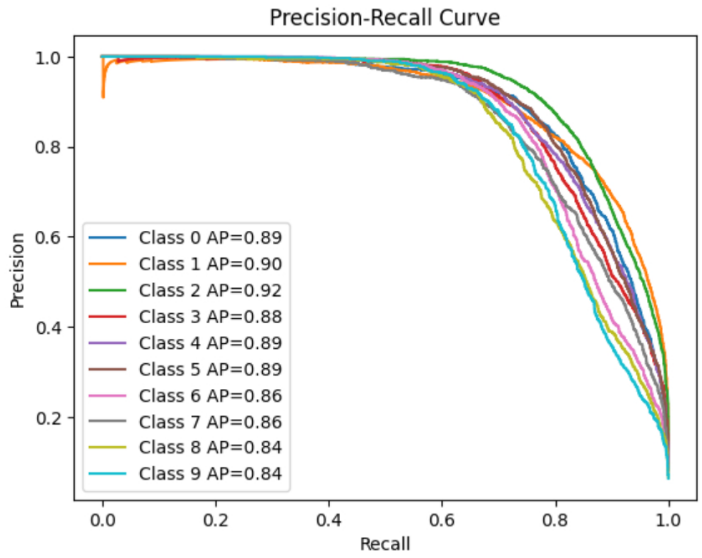
[21]:
```
def get_metrics(true_labels, pred_labels):
    accuracy = accuracy_score(true_labels, pred_labels)
    precision = precision_score(true_labels, pred_labels, zero_division=1, average=None, labels=range(0, 10))
    recall = recall_score(true_labels, pred_labels, zero_division=1, average=None, labels=range(0, 10))
    f1 = f1_score(true_labels, pred_labels, zero_division=0, average=None, labels=range(0, 10))

    return accuracy, precision, recall, f1
```

[23]:
```
accuracy, precision, recall, f1 = get_metrics(true_labels_cpu, pred_labels_cpu)
print(f"Accuracy:{accuracy:.2f}")
for i in range(10):
    print(f"Class {i}: Prec:{precision[i]:.2f}, Recall:{recall[i]:.2f}, F_1 Score:{f1[i]:.2f}")
```

```
Accuracy:0.79
Class 0: Prec:0.84, Recall:0.77, F_1 Score:0.80
Class 1: Prec:0.71, Recall:0.88, F_1 Score:0.79
Class 2: Prec:0.81, Recall:0.83, F_1 Score:0.82
Class 3: Prec:0.79, Recall:0.76, F_1 Score:0.78
Class 4: Prec:0.83, Recall:0.76, F_1 Score:0.79
Class 5: Prec:0.80, Recall:0.80, F_1 Score:0.80
Class 6: Prec:0.79, Recall:0.76, F_1 Score:0.77
Class 7: Prec:0.77, Recall:0.76, F_1 Score:0.77
Class 8: Prec:0.88, Recall:0.68, F_1 Score:0.77
Class 9: Prec:0.79, Recall:0.73, F_1 Score:0.76
```

[22]:
```
def display_pr_curve(true_labels_bin, pred_scores):
    for i in range(0, 10):
        precision_i, recall_i, _ = precision_recall_curve(true_labels_bin[:, i], np.array(pred_scores)[:, i])
        average_precision = average_precision_score(true_labels_bin[:, i], np.array(pred_scores)[:, i])
        plt.step(recall_i, precision_i, where="post", label=f"Class {i} AP={average_precision:.2f}")

    plt.xlabel("Recall")
    plt.ylabel("Precision")

    plt.title("Precision-Recall Curve")
    plt.legend(loc="best")
    plt.show()
```

[25]:
```
display_pr_curve(true_labels_bin, pred_scores)
```



2.4.4 ROC curve and ROC AUC

To compute ROC (Receiver Operating Characteristic), we need to first specify positive class using `label_binarize`.

[23]:
```
# Compute ROC AUC for each class
def get_roc_auc(true_labels_bin, pred_labels_bin):
    roc_auc = dict()
    for i in range(0, 10):
        roc_auc[i] = roc_auc_score(true_labels_bin[:,i], np.array(pred_scores)[:, i])
    return roc_auc
```

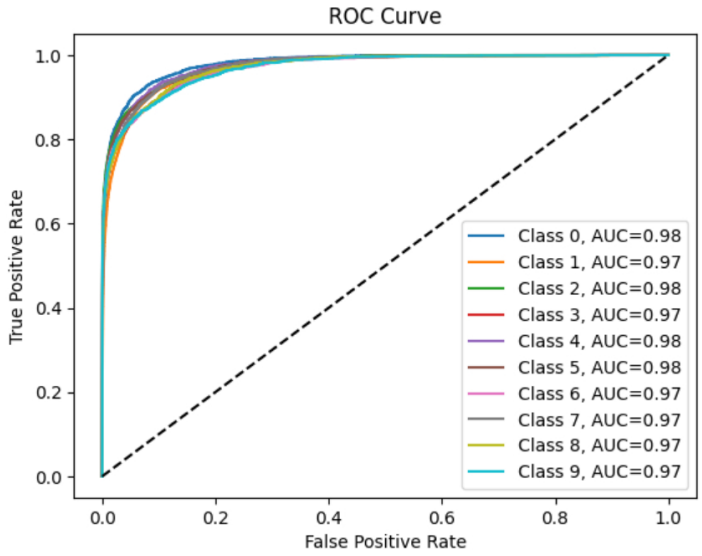
[27]:
```
roc_auc = get_roc_auc(true_labels_bin, pred_labels_bin)
for i, val in roc_auc.items():
    print(f"Class {i}: ROC AUC = {val:.2f}")
```

```
Class 0: ROC AUC = 0.98
Class 1: ROC AUC = 0.97
Class 2: ROC AUC = 0.98
Class 3: ROC AUC = 0.97
Class 4: ROC AUC = 0.98
Class 5: ROC AUC = 0.98
```

```
Class 6: ROC AUC = 0.97
Class 7: ROC AUC = 0.97
Class 8: ROC AUC = 0.97
Class 9: ROC AUC = 0.97
```

[24]:
```python
# Plot ROC AUC Curves
def display_roc_auc(true_labels_bin, pred_scores):
    for i in range(0, 10):
        fpr, tpr, _ = roc_curve(true_labels_bin[:,i], np.array(pred_scores)[:, i])
        plt.plot(fpr, tpr, label=f"Class {i}, AUC={roc_auc[i]:.2f}")

    plt.plot([0, 1], [0, 1], "k--") # Diagnal Line
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    # plt.yscale("logit")
    plt.title("ROC Curve")
    plt.legend(loc="best")
    plt.show()
```

[29]:
```python
display_roc_auc(true_labels_bin, pred_scores)
```



3. Experiment 1: Epoch Number and Learning Rate

[25]:
```python
import itertools
```

Control candidates for different variables

3.1 Design and Run Experiment

3.1.1 Design Hyper Parameger

Set the candidate hyper parameters.

[26]:
```python
candidate_epoch_num = [20, 40, 60, 80]
candidate_lr = [1e-3, 1e-4, 1e-5, 1e-6]
```

From the controlled variables, generate all the possible experiment set.

[27]:
```python
combinations = list(itertools.product(candidate_epoch_num, candidate_lr))
for combo in combinations:
    print(f"[{combo[0]}, {combo[1]:.0e}]", end=" ")
```

```
[20, 1e-03] [20, 1e-04] [20, 1e-05] [20, 1e-06] [40, 1e-03] [40, 1e-04] [40, 1e-05] [40, 1e-06] [60, 1e-03] [60, 1e-04] [60, 1e-05] [60, 1e-06] [80, 1e-03]
[80, 1e-04] [80, 1e-05] [80, 1e-06]
```

3.1.2 Train Models

Train the models for all the generated hyper-parameter combinations.

[28]:
```python
def run_experiment(candidate_epoch_num, candidate_lr):
    combinations = list(itertools.product(candidate_epoch_num, candidate_lr))
    experiments = [] # Experiment Instances

    for combo in combinations:
        print(f"Performing Experiment: epoch_num={combo[0]}, lr={combo[1]}")
        exp_model = SmallVGG().to(device)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.Adam(exp_model.parameters(), lr=combo[1])
```

```
        num_epochs = combo[0]

        train_losses, test_losses = train_and_evaluate(exp_model, train_loader, test_loader, criterion, optimizer, num_epochs)

        # One experiment instance
        experiments.append({
            "num_epoch": combo[0],
            "lr": combo[1],
            "train_losses": train_losses,
            "test_losses": test_losses,
            "model_state_dict": exp_model.state_dict()
        })

        # Prevent CUDA Mem Leak
        del exp_model, criterion, optimizer
        torch.cuda.empty_cache()

    return experiments
```

[45]:
```
experiments = run_experiment(candidate_epoch_num, candidate_lr)
time_str = str(time.time()).replace(".","")
torch.save(experiments, f"./models/experiments_{time_str}.pth")
```

```
100%|██████████| 573/573 [00:16<00:00, 34.30it/s]
Epoch[18/20], Train Loss:108.5583, Test Loss:105.9967
100%|██████████| 573/573 [00:16<00:00, 34.30it/s]
Epoch[19/20], Train Loss:109.5603, Test Loss:103.4721
100%|██████████| 573/573 [00:16<00:00, 34.84it/s]
Epoch[20/20], Train Loss:108.3114, Test Loss:105.9967
Performing Experiment: epoch_num=20, lr=0.0001
100%|██████████| 573/573 [00:16<00:00, 35.12it/s]
Epoch[1/20], Train Loss:286.3824, Test Loss:282.5474
100%|██████████| 573/573 [00:16<00:00, 34.02it/s]
Epoch[2/20], Train Loss:277.8285, Test Loss:261.4198
100%|██████████| 573/573 [00:17<00:00, 31.92it/s]
Epoch[3/20], Train Loss:247.3757, Test Loss:232.7131
```

3.1.3 Load Experiments

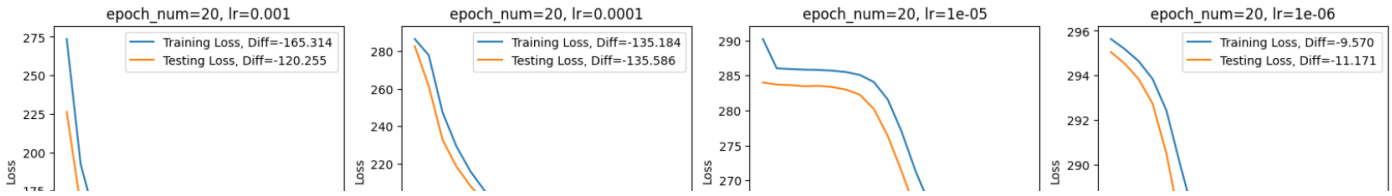
Load the saved experiments, and plot the epoch-loss curve to inspect training performance.

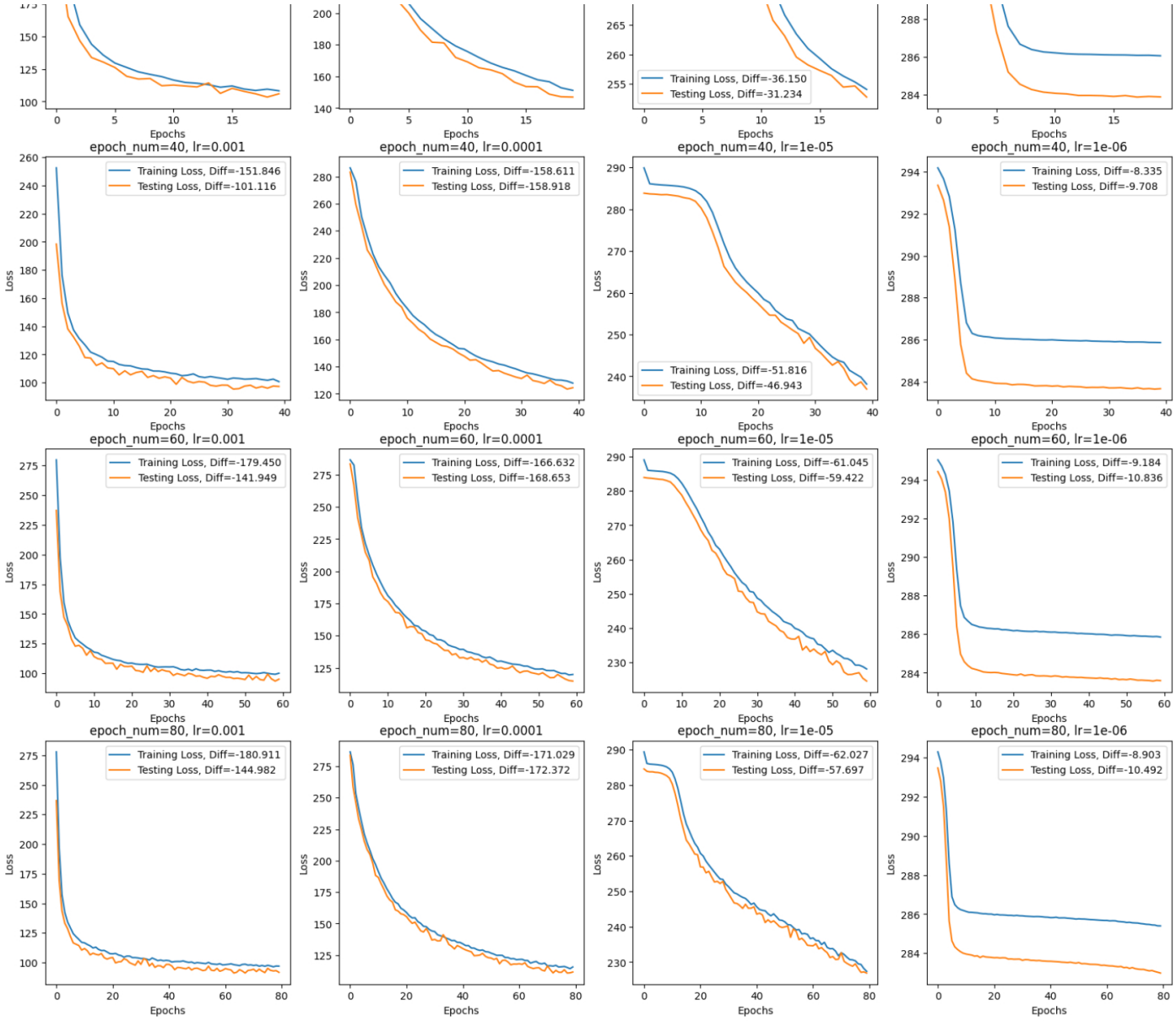
[29]:
```
loaded_experiments = torch.load("./models/experiments_17296227919579012.pth")
```

```
D:\Temps\temp\ipykernel_13748\2506138615.py:1: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future release, the default value for `weights_only` will be flipped to `True`. This limits the functions that could be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded via this mode unless they are explicitly allowlisted by the user via `torch.serialization.add_safe_globals`. We recommend you start setting `weights_only=True` for any use case where you don't have full control of the loaded file. Please open an issue on GitHub for any issues related to this experimental feature.
  loaded_experiments = torch.load("./models/experiments_17296227919579012.pth")
```

[30]:
```
def plot_el(loaded_experiments):
    n_rows = 4
    n_cols = 4
    fig_size = (20, 20)

    fig, axes = plt.subplots(nrows=n_rows, ncols=n_cols, figsize=fig_size)
    # plt.tight_layout()

    for i, ax in enumerate(axes.flat):
        train_losses, test_losses = loaded_experiments[i]["train_losses"], loaded_experiments[i]["test_losses"]

        train_difference = train_losses[-1]-train_losses[0]
        test_difference = test_losses[-1]-test_losses[0]

        ax.plot(train_losses, label=f"Training Loss, Diff={train_difference:.3f}")
        ax.plot(test_losses, label=f"Testing Loss, Diff={test_difference:.3f}")
        ax.set_xlabel("Epochs")
        ax.set_ylabel("Loss")
        ax.set_title(f"epoch_num={loaded_experiments[i]['num_epoch']}, lr={loaded_experiments[i]['lr']}")
        ax.legend()

    plt.show()
```

[31]:
```
plot_el(loaded_experiments)
```





3.2 Apply Model, Get Result

[32]:
```python
def get_experiment_results(loaded_experiments, extra_loader):
    experiment_results = []
    for i, exp in enumerate(loaded_experiments):
        pred_scores, true_labels_cpu, pred_labels_cpu = get_predictions(exp["model_state_dict"], extra_loader)
        print(f"Experiment {i+1}, num_epoch={exp['num_epoch']}, lr={exp['lr']}")
        print("First 100 true labels:")
        [print(num, end=" ") for num in true_labels_cpu[:100]]
        print("...\n")

        print("First 100 true predictions:")
        [print(num, end=" ") for num in pred_labels_cpu[:100]]
        print("...\n")

        print("First 5 prediction Probabilities:")
        [print(num, end=" ") for num in pred_scores[:5]]
        print("...")

        experiment_results.append({
            "epoch_num": exp['num_epoch'],
            "lr": exp['lr'],
            "true_labels": true_labels_cpu,
            "pred_labels": pred_labels_cpu,
            "pred_scores": pred_scores
        })

        del pred_scores, true_labels_cpu, pred_labels_cpu
        torch.cuda.empty_cache()
    return experiment_results
```

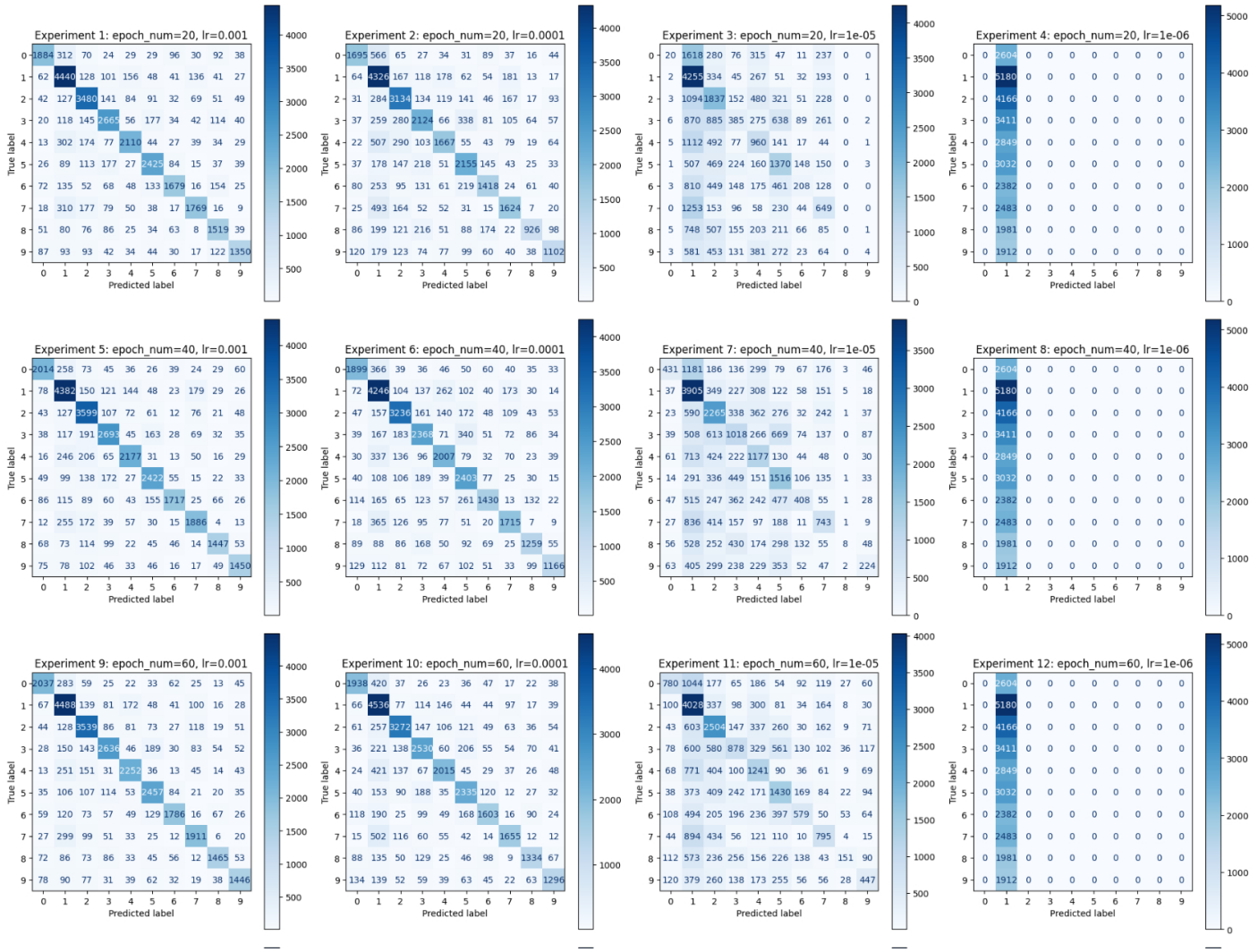
[33]:
```python
experiment_results = get_experiment_results(loaded_experiments, extra_loader)
```

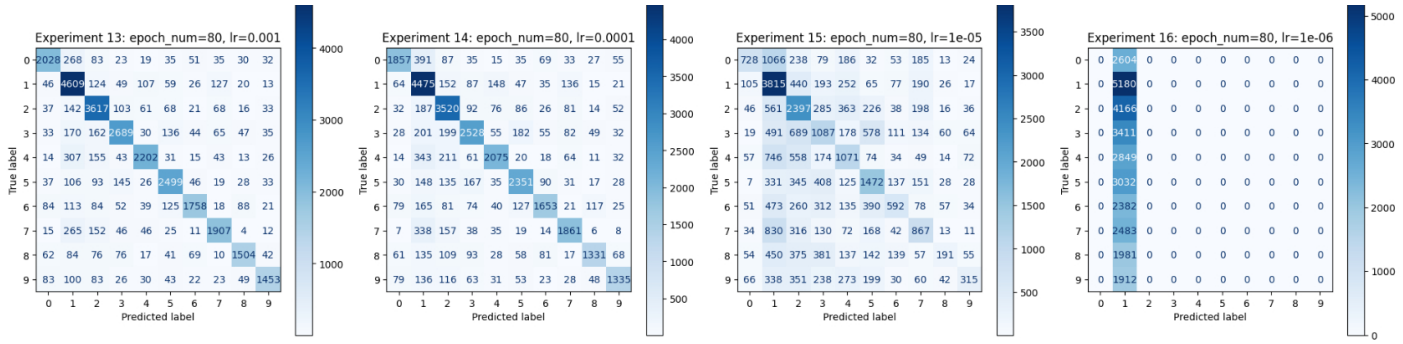
```
0.6128180623054504, 0.00019326117762830108, 0.0015668770065531135] ...
100%|██████████| 469/469 [00:07<00:00, 62.32it/s]
Experiment 14, num_epoch=80, lr=0.0001
First 100 true labels:
4 7 8 7 1 1 7 4 3 0 2 8 8 3 1 1 7 0 8 1 5 6 4 4 4 6 3 4 4 3 0 1 7 6 0 1 1 0 5 7 5 1 8 5 5 2 9 6 1 5 2 3 5 3 6 9 2 3 4 1 7 7 3 1 2 2 0 1 1 3 1 5 1 1 9 9
4 8 0 5 1 3 8 2 9 5 6 0 7 8 3 0 6 4 0 3 1 1 0 0 ...

First 100 true predictions:
4 4 8 7 1 7 1 4 3 0 2 8 8 1 1 1 5 0 8 3 5 6 4 4 2 6 3 4 4 0 0 1 1 6 0 1 1 0 2 7 5 1 8 5 5 2 1 6 5 5 2 3 3 3 6 2 2 2 9 1 7 1 3 1 2 2 0 1 6 1 7 5 1 1 2 9
4 8 0 5 1 3 8 2 1 5 6 0 7 8 3 0 6 4 0 3 1 7 0 6 ...

First 5 prediction Probabilities:
[0.028719374909996986, 0.15402953326702118, 0.030455615371465683, 0.054125770926475525, 0.5040739178657532, 0.028038598597049713, 0.014246501959860325,
0.10302358120679855, 0.035454072058200836, 0.047833044081926346] [0.04627608135342598, 0.07979067414999008, 0.04209180176258087, 0.014844846911728382,
0.5789914727210999, 0.029636671766638756, 0.0039027431048452854, 0.05828758329153061, 0.007270190399140 1196, 0.13890793919563293] [2.888228436859208e-0
6, 0.0003862071025650948 3, 0.14740674197673798, 0.010127495042979717, 0.0001277478877454996, 0.0001454872399335727, 0.0005510617629624903, 0.00186865252
90831923, 0.8045840859413147, 0.034799642860889435] [0.004236710723489523, 0.1941903829574585, 0.0015110233798623085, 0.007185585796833038, 0.0051199737
93625832, 7.590687934225425e-05, 0.0019936922471970 32, 0.7849059760269165, 0.0004868529213126749, 0.0002957638644147664 3] [0.005654376931488514, 0.71883
```

3.2.1 Confusion Matrix

[34]:
```
def plot_cm(experiment_results, hyper_param_names):
    fig, axes = plt.subplots(4,4, figsize=(20,20))
    axes = axes.flatten()

    hparam_1, hparam_2 = hyper_param_names

    for i, exp_rs in enumerate(experiment_results):
        true_labels, pred_labels = exp_rs['true_labels'], exp_rs['pred_labels']
        cm = confusion_matrix(true_labels, pred_labels)
        disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=range(0,10))
        disp.plot(ax=axes[i], cmap = plt.cm.Blues)
        axes[i].set_title(f"Experiment {i+1}: {hparam_1}={exp_rs[hparam_1]}, {hparam_2}={exp_rs[hparam_2]}")

    for j in range(i+1, 16):
        fig.delaxes(axes[j])

    plt.tight_layout()
    plt.show()
```

[35]:
```
plot_cm(experiment_results, ['epoch_num', 'lr'])
```

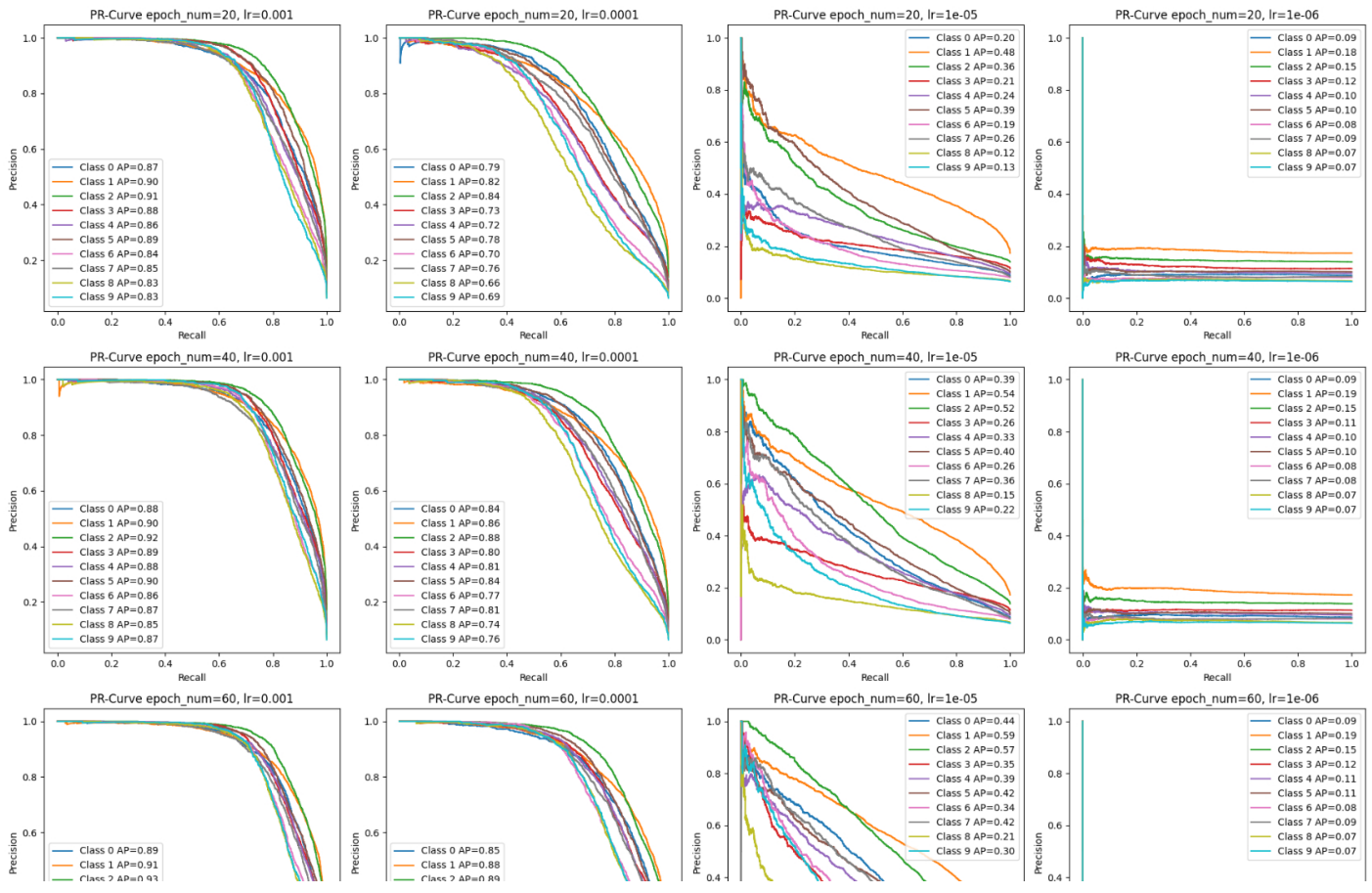


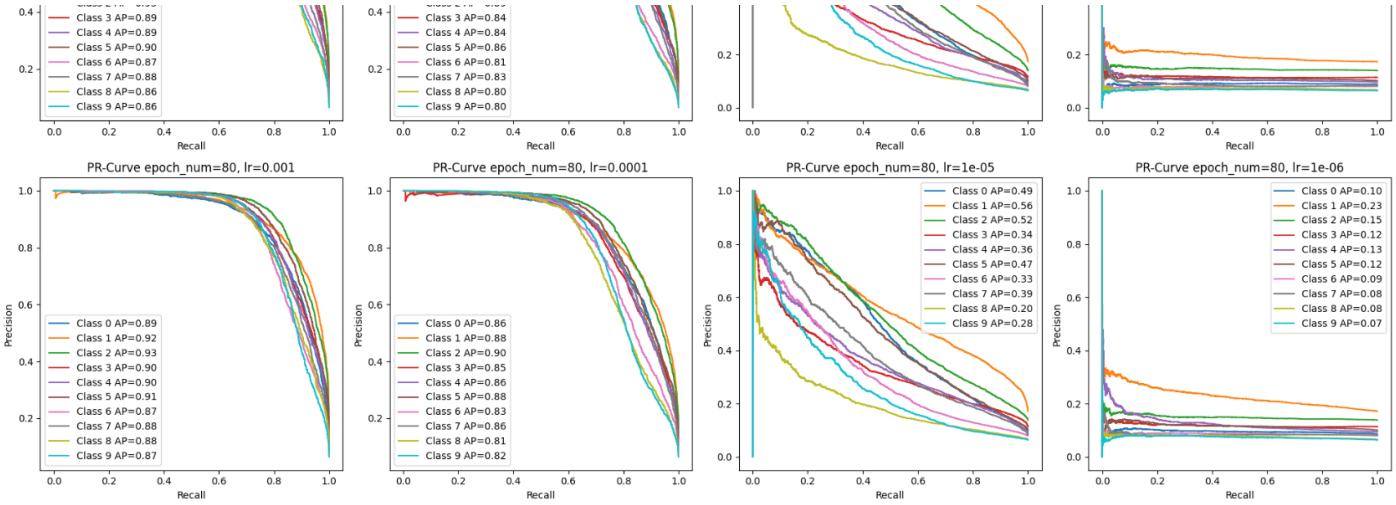


3.2.2 Precision-Recall Curve

[37]:
```
def plot_pr(experiment_results, hyper_param_names):
    fig, axes = plt.subplots(4,4, figsize=(20,20))
    axes = axes.flatten()

    hparam_1, hparam_2 = hyper_param_names

    for i, exp_rs in enumerate(experiment_results):
        true_labels, pred_labels, pred_scores = exp_rs['true_labels'], exp_rs['pred_labels'], exp_rs['pred_scores']
        true_labels_bin, pred_labels_bin = label_binarize(true_labels, classes=range(0,10)), label_binarize(pred_labels, classes=range(0,10))

        accuracy, precision, recall, f1 = get_metrics(true_labels, pred_labels)
        for j in range(0,10):
            # print(f"Class {j}: Prec:{precision[j]:.2f}, Recall:{recall[j]:.2f}, F_1 Score:{f1[j]:.2f}")
            precision_i, recall_i, _ = precision_recall_curve(true_labels_bin[:, j], np.array(pred_scores)[:, j])

            average_precision = average_precision_score(true_labels_bin[:, j], np.array(pred_scores)[:, j])
            axes[i].step(recall_i, precision_i, where="post", label=f"Class {j} AP={average_precision:.2f}")
            axes[i].set_title(f"PR-Curve {hparam_1}={exp_rs[hparam_1]}, {hparam_2}={exp_rs[hparam_2]}")
        axes[i].legend()
        axes[i].set_xlabel("Recall")
        axes[i].set_ylabel("Precision")

    for j in range(i+1, 16):
        fig.delaxes(axes[j])

    plt.tight_layout()
    plt.show()
```

[38]:
```
plot_pr(experiment_results, ['epoch_num', 'lr'])
```





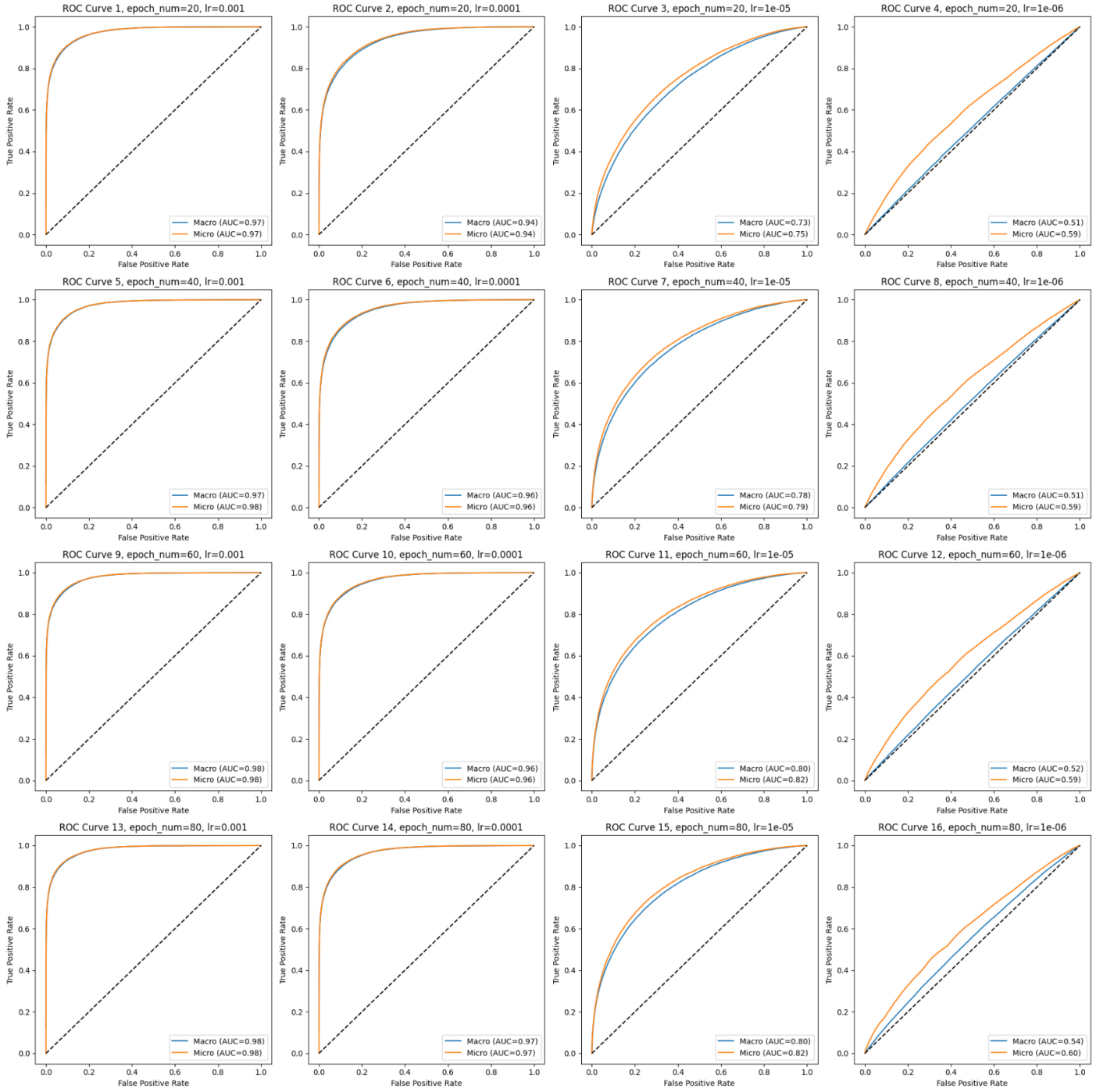
3.3.3 ROC AUC Curve

[39]:
```python
def plot_roc_auc(experiment_results, hyper_param_names):
    fig, axes = plt.subplots(4,4, figsize=(20,20))
    axes = axes.flatten()

    hparam_1, hparam_2 = hyper_param_names

    for i, exp_rs in enumerate(experiment_results):
        true_labels, pred_labels, pred_scores = exp_rs['true_labels'], exp_rs['pred_labels'], exp_rs['pred_scores']
        true_labels_bin, pred_labels_bin = label_binarize(true_labels, classes=range(0,10)), label_binarize(pred_labels, classes=range(0,10))

        roc_auc = get_roc_auc(true_labels_bin, pred_scores)

        for j in range(0,10):
            fpr, tpr, _ = roc_curve(true_labels_bin[:, j], np.array(pred_scores)[:, j])
            axes[i].plot(fpr, tpr, label=f"Class {j}, AUC={roc_auc[j]:.2f}")

        axes[i].plot([0, 1],[0, 1], "k--")
        axes[i].set_xlabel("False Positive Rate")
        axes[i].set_ylabel("True Positive Rate")
        axes[i].set_title(f"ROC Curve {i+1}, {hparam_1}={exp_rs[hparam_1]}, {hparam_2}={exp_rs[hparam_2]}")
        axes[i].legend()

    plt.tight_layout()
    plt.show()
```

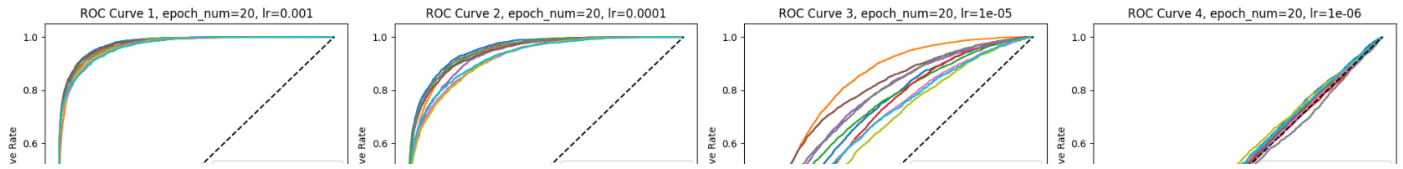
[40]:
```python
def plot_roc_auc(experiment_results, hyper_param_names, curve_type):
    fig, axes = plt.subplots(4, 4, figsize=(20, 20))
    axes = axes.flatten()

    hparam_1, hparam_2 = hyper_param_names

    for i, exp_rs in enumerate(experiment_results):
        true_labels, pred_scores = exp_rs['true_labels'], exp_rs['pred_scores']
        true_labels_bin = label_binarize(true_labels, classes=range(0, 10))

        # All Classes' ROC curve & ROC Area Under Curve
        fpr = dict()
        tpr = dict()
        roc_auc = dict()

        for j in range(10):
            fpr[j], tpr[j], _ = roc_curve(true_labels_bin[:, j], np.array(pred_scores)[:, j])
            roc_auc[j] = auc(fpr[j], tpr[j])

        # Macro-Average ROC & ROC-AUC
        all_fpr = np.unique(np.concatenate([fpr[j] for j in range(10)]))
        mean_tpr = np.zeros_like(all_fpr)
        for j in range(10):
            mean_tpr += np.interp(all_fpr, fpr[j], tpr[j])
        mean_tpr /= 10

        fpr["macro"] = all_fpr
        tpr["macro"] = mean_tpr
        roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

        # Compute micro-average ROC curve and ROC area
        fpr["micro"], tpr["micro"], _ = roc_curve(true_labels_bin.ravel(), np.array(pred_scores).ravel())
        roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

        # Plot only Macro or Micro ROC curves
        if curve_type == "macro_micro":
            axes[i].plot(fpr["macro"], tpr["macro"], label=f"Macro (AUC={roc_auc['macro']:.2f})")
```

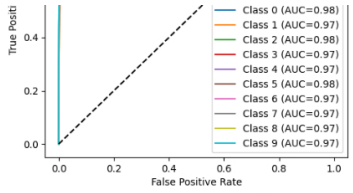
```
            axes[i].plot(fpr["micro"], tpr["micro"], label=f"Micro (AUC={roc_auc['micro']:.2f})")
        elif curve_type == "all":
            # Plot all ROC curves
            for j in range(10):
                axes[i].plot(fpr[j], tpr[j], label=f"Class {j} (AUC={roc_auc[j]:.2f})")

        axes[i].plot([0, 1], [0, 1], "k--")
        axes[i].set_xlabel("False Positive Rate")
        axes[i].set_ylabel("True Positive Rate")
        axes[i].set_title(f"ROC Curve {i+1}, epoch_num={exp_rs['epoch_num']}, lr={exp_rs['lr']}")
        axes[i].legend(loc='lower right')

    plt.tight_layout()
    plt.show()
```

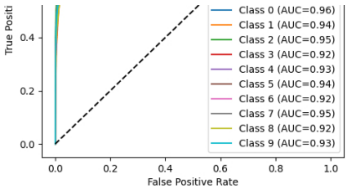
[41]:
```
plot_roc_auc(experiment_results, ['epoch_num','lr'], "macro_micro")
```

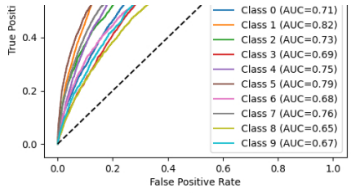


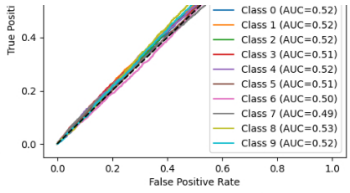
[42]:
```
plot_roc_auc(experiment_results, ['epoch_num','lr'], "all")
```

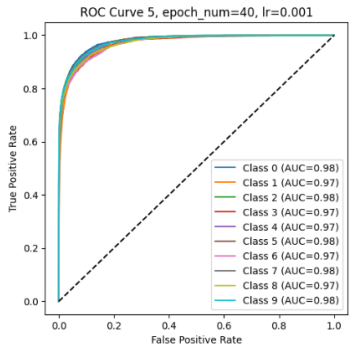


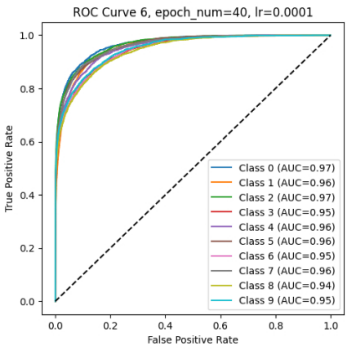

True Positi
False Positive Rate
Class 0 (AUC=0.98)
Class 1 (AUC=0.97)
Class 2 (AUC=0.98)
Class 3 (AUC=0.97)
Class 4 (AUC=0.97)
Class 5 (AUC=0.98)
Class 6 (AUC=0.97)
Class 7 (AUC=0.97)
Class 8 (AUC=0.97)
Class 9 (AUC=0.97)

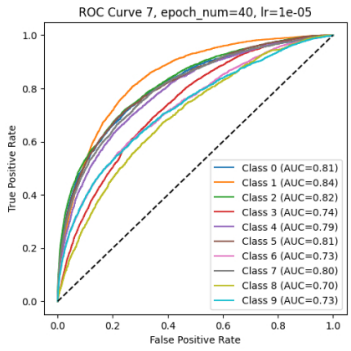

True Positi
False Positive Rate
Class 0 (AUC=0.96)
Class 1 (AUC=0.94)
Class 2 (AUC=0.95)
Class 3 (AUC=0.92)
Class 4 (AUC=0.93)
Class 5 (AUC=0.94)
Class 6 (AUC=0.92)
Class 7 (AUC=0.95)
Class 8 (AUC=0.92)
Class 9 (AUC=0.93)

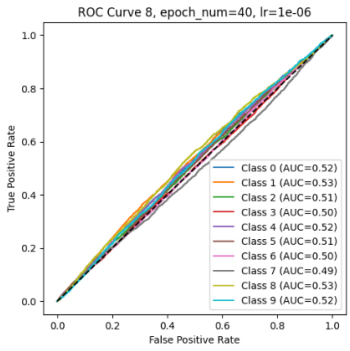

True Positi
False Positive Rate
Class 0 (AUC=0.71)
Class 1 (AUC=0.82)
Class 2 (AUC=0.73)
Class 3 (AUC=0.69)
Class 4 (AUC=0.75)
Class 5 (AUC=0.79)
Class 6 (AUC=0.68)
Class 7 (AUC=0.76)
Class 8 (AUC=0.65)
Class 9 (AUC=0.67)

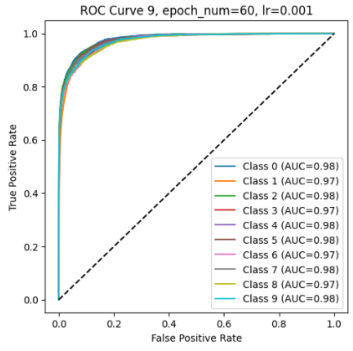

True Positi
False Positive Rate
Class 0 (AUC=0.52)
Class 1 (AUC=0.52)
Class 2 (AUC=0.52)
Class 3 (AUC=0.51)
Class 4 (AUC=0.52)
Class 5 (AUC=0.51)
Class 6 (AUC=0.50)
Class 7 (AUC=0.49)
Class 8 (AUC=0.53)
Class 9 (AUC=0.52)

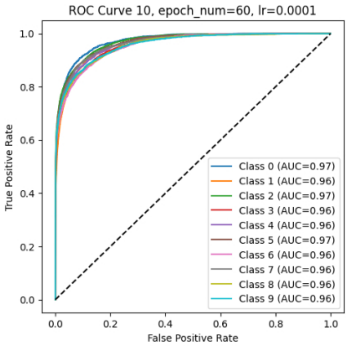

ROC Curve 5, epoch_num=40, lr=0.001
True Positive Rate
False Positive Rate
Class 0 (AUC=0.98)
Class 1 (AUC=0.97)
Class 2 (AUC=0.98)
Class 3 (AUC=0.97)
Class 4 (AUC=0.97)
Class 5 (AUC=0.98)
Class 6 (AUC=0.97)
Class 7 (AUC=0.98)
Class 8 (AUC=0.97)
Class 9 (AUC=0.98)

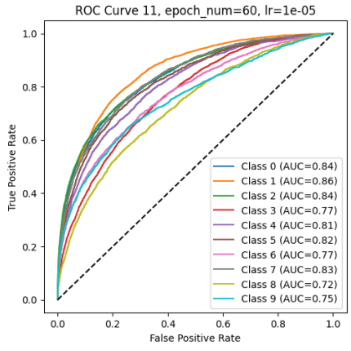

ROC Curve 6, epoch_num=40, lr=0.0001
True Positive Rate
False Positive Rate
Class 0 (AUC=0.97)
Class 1 (AUC=0.96)
Class 2 (AUC=0.97)
Class 3 (AUC=0.95)
Class 4 (AUC=0.96)
Class 5 (AUC=0.96)
Class 6 (AUC=0.95)
Class 7 (AUC=0.96)
Class 8 (AUC=0.94)
Class 9 (AUC=0.95)

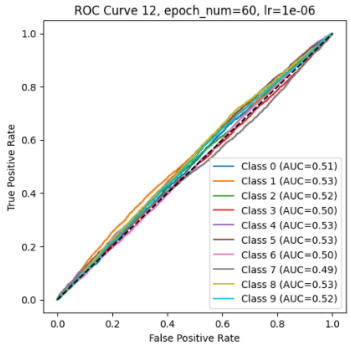

ROC Curve 7, epoch_num=40, lr=1e-05
True Positive Rate
False Positive Rate
Class 0 (AUC=0.81)
Class 1 (AUC=0.84)
Class 2 (AUC=0.82)
Class 3 (AUC=0.74)
Class 4 (AUC=0.79)
Class 5 (AUC=0.81)
Class 6 (AUC=0.73)
Class 7 (AUC=0.80)
Class 8 (AUC=0.70)
Class 9 (AUC=0.73)

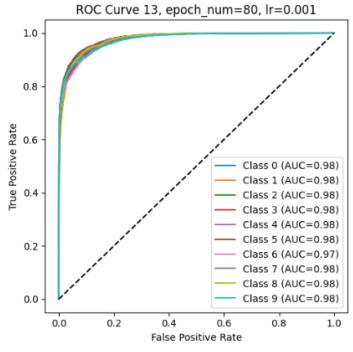

ROC Curve 8, epoch_num=40, lr=1e-06
True Positive Rate
False Positive Rate
Class 0 (AUC=0.52)
Class 1 (AUC=0.53)
Class 2 (AUC=0.51)
Class 3 (AUC=0.50)
Class 4 (AUC=0.52)
Class 5 (AUC=0.51)
Class 6 (AUC=0.50)
Class 7 (AUC=0.49)
Class 8 (AUC=0.53)
Class 9 (AUC=0.52)

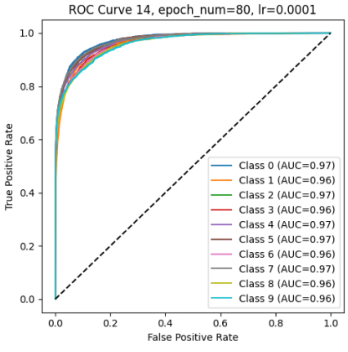

ROC Curve 9, epoch_num=60, lr=0.001
True Positive Rate
False Positive Rate
Class 0 (AUC=0.98)
Class 1 (AUC=0.97)
Class 2 (AUC=0.98)
Class 3 (AUC=0.97)
Class 4 (AUC=0.98)
Class 5 (AUC=0.98)
Class 6 (AUC=0.97)
Class 7 (AUC=0.98)
Class 8 (AUC=0.97)
Class 9 (AUC=0.98)

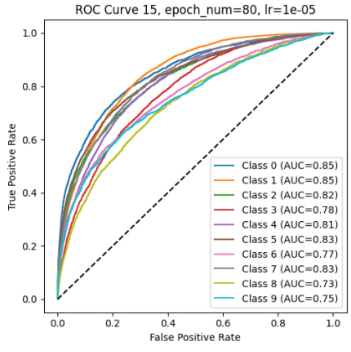

ROC Curve 10, epoch_num=60, lr=0.0001
True Positive Rate
False Positive Rate
Class 0 (AUC=0.97)
Class 1 (AUC=0.96)
Class 2 (AUC=0.97)
Class 3 (AUC=0.96)
Class 4 (AUC=0.96)
Class 5 (AUC=0.97)
Class 6 (AUC=0.96)
Class 7 (AUC=0.96)
Class 8 (AUC=0.96)
Class 9 (AUC=0.96)

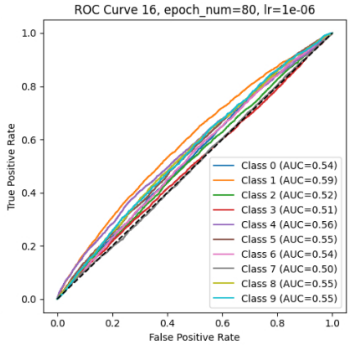

ROC Curve 11, epoch_num=60, lr=1e-05
True Positive Rate
False Positive Rate
Class 0 (AUC=0.84)
Class 1 (AUC=0.86)
Class 2 (AUC=0.84)
Class 3 (AUC=0.77)
Class 4 (AUC=0.81)
Class 5 (AUC=0.82)
Class 6 (AUC=0.77)
Class 7 (AUC=0.83)
Class 8 (AUC=0.72)
Class 9 (AUC=0.75)


ROC Curve 12, epoch_num=60, lr=1e-06
True Positive Rate
False Positive Rate
Class 0 (AUC=0.51)
Class 1 (AUC=0.53)
Class 2 (AUC=0.52)
Class 3 (AUC=0.50)
Class 4 (AUC=0.53)
Class 5 (AUC=0.53)
Class 6 (AUC=0.50)
Class 7 (AUC=0.49)
Class 8 (AUC=0.53)
Class 9 (AUC=0.52)


ROC Curve 13, epoch_num=80, lr=0.001
True Positive Rate
False Positive Rate
Class 0 (AUC=0.98)
Class 1 (AUC=0.98)
Class 2 (AUC=0.98)
Class 3 (AUC=0.98)
Class 4 (AUC=0.98)
Class 5 (AUC=0.98)
Class 6 (AUC=0.97)
Class 7 (AUC=0.98)
Class 8 (AUC=0.98)
Class 9 (AUC=0.98)


ROC Curve 14, epoch_num=80, lr=0.0001
True Positive Rate
False Positive Rate
Class 0 (AUC=0.97)
Class 1 (AUC=0.96)
Class 2 (AUC=0.97)
Class 3 (AUC=0.96)
Class 4 (AUC=0.97)
Class 5 (AUC=0.97)
Class 6 (AUC=0.96)
Class 7 (AUC=0.97)
Class 8 (AUC=0.96)
Class 9 (AUC=0.96)


ROC Curve 15, epoch_num=80, lr=1e-05
True Positive Rate
False Positive Rate
Class 0 (AUC=0.85)
Class 1 (AUC=0.85)
Class 2 (AUC=0.82)
Class 3 (AUC=0.78)
Class 4 (AUC=0.81)
Class 5 (AUC=0.83)
Class 6 (AUC=0.77)
Class 7 (AUC=0.83)
Class 8 (AUC=0.73)
Class 9 (AUC=0.75)


ROC Curve 16, epoch_num=80, lr=1e-06
True Positive Rate
False Positive Rate
Class 0 (AUC=0.54)
Class 1 (AUC=0.59)
Class 2 (AUC=0.52)
Class 3 (AUC=0.51)
Class 4 (AUC=0.56)
Class 5 (AUC=0.55)
Class 6 (AUC=0.54)
Class 7 (AUC=0.50)
Class 8 (AUC=0.55)
Class 9 (AUC=0.55)