Data & Preprocessing

Making the train data

We want to increase out data. Thus we will magnify it in the following way: We will apply transform on the train data and add it to the train set.

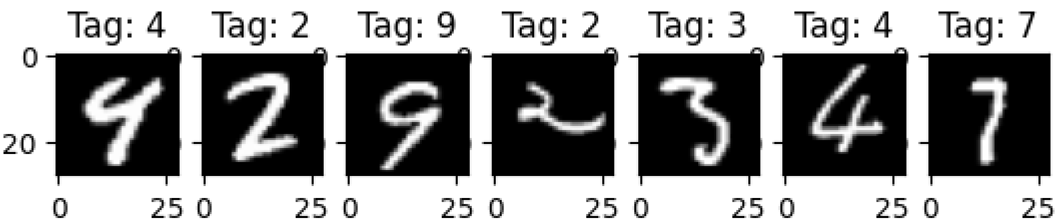
```
import numpy as np
import torch
from torch import nn, optim
import torchvision
from torchvision import datasets, transforms
from PIL import Image
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import torch.nn.functional as F
import torchvision.transforms as T
from torch.autograd import Variable


device = "cuda"

transform = transforms.Compose([transforms.ToTensor(),transforms.Normalize((0.5,), (0.5,))
transformOMNI = transforms.Compose([transforms.Resize((28,28)),transforms.ToTensor()])

transform_aug = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (0.5,)),
                                   ])


#Only Normalization transform:
train_and_valid_set = datasets.MNIST(root="MNIST", download=True, train=True, transform=tr
test_set = datasets.MNIST(root="MNIST", download=True, train=False, transform=transform)
train, valid = torch.utils.data.random_split(train_and_valid_set,[int(0.8*len(train_and_va

```

making the test data

```
Fashion = datasets.FashionMNIST(root="FashionMNIST", download=True, train=False, transform
omniglot = datasets.Omniglot(root="Omniglot", download=True,transform=transformOMNI, targe
test =test_set + Fashion
```

```
test_dataloader = DataLoader(test, batch_size=64, shuffle=True)
baseline_test_dataloader = DataLoader(test_set, batch_size=64, shuffle=True)
train_dataloader = DataLoader(train, batch_size=64, shuffle=True)
validation_dataloader = DataLoader(valid, batch_size=64, shuffle=True)


import random as r
fig, axes = plt.subplots(1,7)
for i in range(len(axes)):
    rnd = int(r.random() * 2000)
    axes[i].imshow(train[rnd][0].permute(1, 2, 0), cmap="gray")
    axes[i].set_title("Tag: " + str(train[rnd][1]))
plt.show()
```

```
Files already downloaded and verified
```



Models

```
from torch import nn, optim

class ConvNet(nn.Module):
    def newWidth(self,W,s,k,p):
      #W stands for width , s stand for Stride , k for Kernel , p for Padding
      #We will need to calculate the dimensions of the image, in order to insialize the ri
      #Thus,we will track the size after every layer
        return int((W-k+p*2)/s+1)
    def __init__(self, kernel):
        super(ConvNet, self).__init__()
        # Conv2d(in_channels, out_channels, kernel_size)
        width = 28
        dimention = 1
        #Next, we will need to calculate the dimensions of the image. Thus, we will trackt
        self.conv1 = nn.Conv2d(in_channels =1, out_channels = 10, kernel_size = kernel, st
        #Given a stride of 1, padding = 1, and a kernel size of 3x3, the new width (w*) of
        #the height is the same.
        width = self.newWidth(width,1,kernel,1)

        self.maxPooling1 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        #From the same reasons, the new width and height can be calculated as follows:
        #Notice that stride = 2. thus, we will divide our results by 2.
        width = self.newWidth(width,2,2,0)

```

```
        self.conv2 = nn.Conv2d(in_channels =10, out_channels = 20, kernel_size = kernel, s
        width = self.newWidth(width,1,kernel,1)

        self.maxPooling2 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        width = self.newWidth(width,2,2,0)

        # We alredy calculated the new width. the new height equals to the new width, and
        # Thus, the input layer size is width*width*20.
        self.hidden1 = nn.Linear(width*width*20, 64)
        self.hidden2 = nn.Linear(64, 10)

    def forward(self, x, bs=False):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.maxPooling1(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = self.maxPooling2(x)
        x = torch.flatten(x,1)
        if bs :
            x = torch.flatten(x)
        x = self.hidden1(x)
        x = F.relu(x)
        x = self.hidden2(x)
        return x
convNet = ConvNet(5).to(device)
print(convNet)
```

```
ConvNet(
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1), padding=(1, 1))
  (maxPooling1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fa
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1), padding=(1, 1))
  (maxPooling2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fa
  (hidden1): Linear(in_features=500, out_features=64, bias=True)
  (hidden2): Linear(in_features=64, out_features=10, bias=True)
)
```

```
class ML_autoencoder(nn.Module):
    def __init__(self, code_size ,hiden1=400 ,hiden2=200 ,hiden3=64):
        super(ML_autoencoder, self).__init__()
        self.FC1 = nn.Linear(28 * 28 * 1 , hiden1)
        self.FC2 = nn.Linear(hiden1 , hiden2)
        self.FC3 = nn.Linear(hiden2 , hiden3)
        self.FC4 = nn.Linear(hiden3 , code_size)
        self.FC5 = nn.Linear(code_size , hiden3)
        self.FC6 = nn.Linear(hiden3 , hiden2)
        self.FC7 = nn.Linear(hiden2 , hiden1)
        self.FC8 = nn.Linear(hiden1 , 28 * 28 * 1)

        self.n_classes = 10
```

```
        # Add classification head
        self.clf = nn.Sequential(
            nn.Linear(code_size, self.n_classes),
            nn.LogSoftmax(dim=1))

    def encoder(self,x):
        x = self.FC1(x)
        x = F.relu(x)
        x = self.FC2(x)
        x = F.relu(x)
        x = self.FC3(x)
        x = F.relu(x)
        x = self.FC4(x)
        x = F.relu(x)
        return x

    def decoder(self,x):
        x = self.FC5(x)
        x = F.relu(x)
        x = self.FC6(x)
        x = F.relu(x)
        x = self.FC7(x)
        x = F.relu(x)
        x = self.FC8(x)
        x = F.tanh(x)
        return x

    def forward(self, x):
        encoded_vector = self.encoder(x)
        recon = self.decoder(encoded_vector)
        preds = self.clf(encoded_vector)

        return recon, preds
model3 = ML_autoencoder(20).to(device)
print(model3)
```

```
ML_autoencoder(
  (FC1): Linear(in_features=784, out_features=400, bias=True)
  (FC2): Linear(in_features=400, out_features=200, bias=True)
  (FC3): Linear(in_features=200, out_features=64, bias=True)
  (FC4): Linear(in_features=64, out_features=20, bias=True)
  (FC5): Linear(in_features=20, out_features=64, bias=True)
  (FC6): Linear(in_features=64, out_features=200, bias=True)
  (FC7): Linear(in_features=200, out_features=400, bias=True)
  (FC8): Linear(in_features=400, out_features=784, bias=True)
  (clf): Sequential(
    (0): Linear(in_features=20, out_features=10, bias=True)
    (1): LogSoftmax(dim=1)
  )
)
```

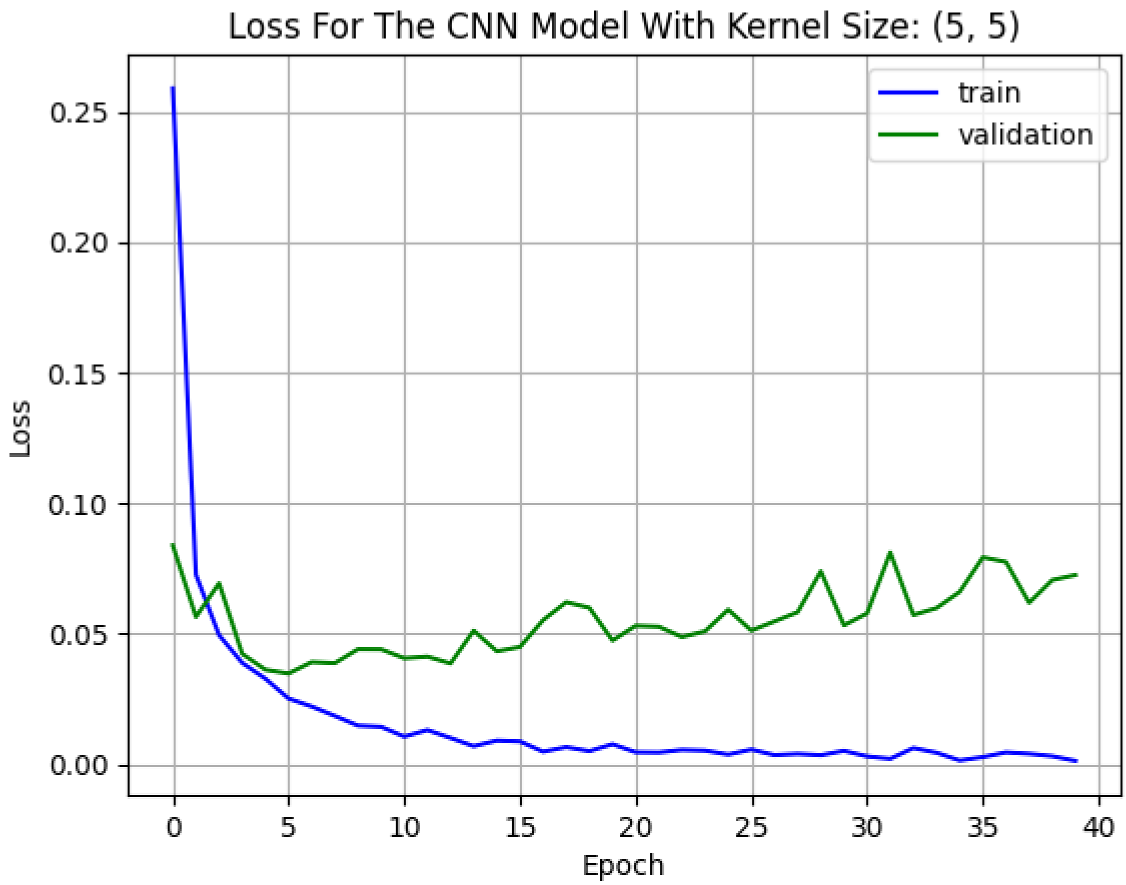
Training

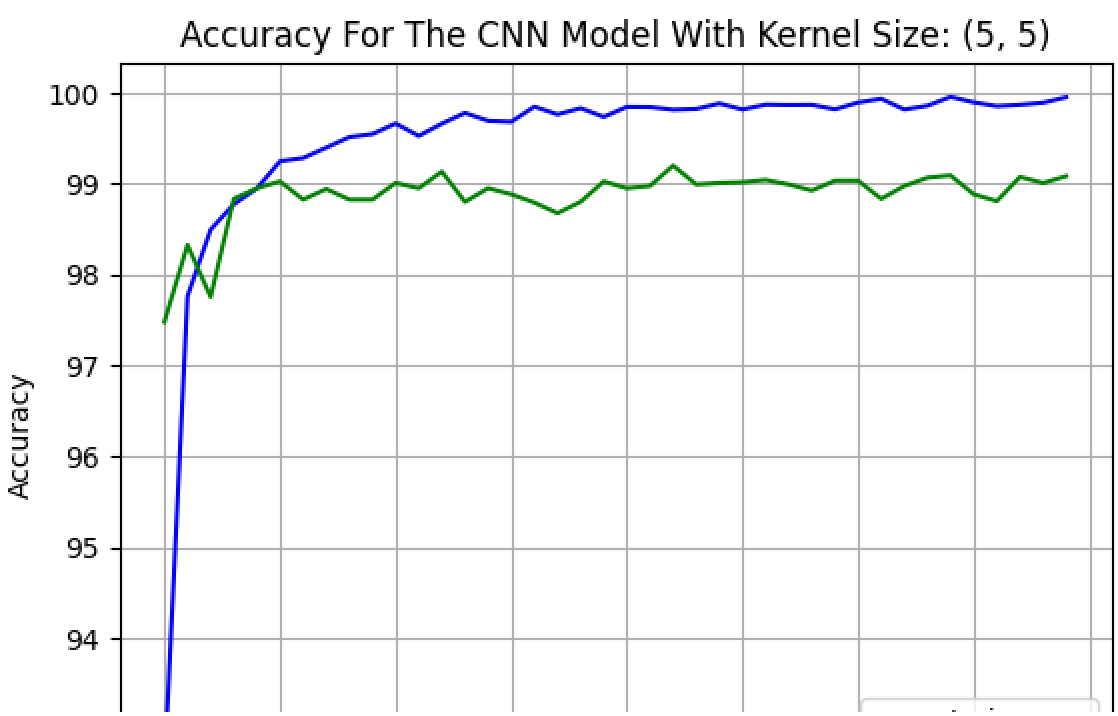
```
def trainML_AE(model, num_epochs, trainloader):
    learning_rate = 0.0001
    criterion = nn.MSELoss(reduction='sum')
    clf_criterion = nn.NLLLoss()

    optimizer = torch.optim.Adam(
    model.parameters(), lr=learning_rate, weight_decay=1e-5)
    # a list to hold the loss across epochs
    loss_train = []

    for epoch in range(num_epochs):
        loss_epoch = 0
        for data in trainloader:
            img, labels = data
            img = img.view(img.size(0), -1)
            img = Variable(img).to(device)
            labels = labels.to(device)
            # ===================forward=====================
            recon, preds =  model(img)
            loss = criterion(recon, img)
            clf_loss = clf_criterion(preds, labels)
            loss = loss + clf_loss
            # ===================backward====================
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            loss_epoch += loss.item()
        # divide by number of batchs
        loss_epoch = loss_epoch / len(trainloader)
        #print("epoch: ",epoch, " loss: ",loss_epoch)
        loss_train.append(loss_epoch)

    plt.plot(loss_train,color='blue',label='batch loss')
    plt.xlabel("Epoch")
    plt.ylabel("batch loss")
    plt.grid()
    plt.title("train autoencoder Loss")
    plt.legend()
    plt.show()

    model.eval()
    return model

```

```
def train_model( model, dataloaders, optimizer):
    criterion = nn.CrossEntropyLoss()
    lossHistory = {'train':list() ,'val': list()}
    AccHistory = {'train':list() ,'val': list()}
    for epoch in range(40):  # loop over the dataset multiple times
        running_loss = 0.0
        for type1 in ["train","val"]:
            if type1 == "train":
            # Set model to training mode
                model.train()
            else:
                # Set model to evaluate mode
                model.eval()
            running_loss = 0.0
            predictionAcc =0.0
            for i, data in enumerate(dataloaders[type1], 0):
                # get the inputs; data is a list of [inputs, labels]
                inputs, labels = data
                # mode to device/cuda
                inputs, labels = inputs.to(device), labels.to(device)
                # zero the parameter gradients
                optimizer.zero_grad()

                # forward + backward + optimize
                outputs = model(inputs)

                _, prediction = torch.max(outputs, 1)
                predictionAcc += torch.sum(prediction == labels.data).cpu()

                loss = criterion(outputs, labels)
                if(type1 == "train"):
                    loss.backward()
                    optimizer.step()

                # print statistics
                running_loss += loss.item() * inputs.size(0)
            AccHistory[type1].append((predictionAcc.double() / dataset_sizes[type1]) * 100
            lossHistory[type1].append(running_loss/dataset_sizes[type1])


    params = str(model.conv2.kernel_size)
    params ="Model With Kernel Size: " +params
    plt.plot(lossHistory["train"],color='blue',label='train')
    plt.plot(lossHistory["val"],color='green',label='validation')


    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.grid()
    plt.title("Loss For The CNN "+params)
    plt.legend()
```

```
    plt.show()

    plt.plot(AccHistory["train"],color='blue',label='train')
    plt.plot(AccHistory["val"],color='green',label='validation')
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.grid()
    plt.title("Accuracy For The CNN "+params)
    plt.legend()
    plt.show()

    return model
```

```
dataloaders = {'train':train_dataloader ,  'val': validation_dataloader}
dataset_sizes = {'train':len(train) ,'val': len(valid)}
optimizer1=optim.Adam(convNet.parameterss(), lr=0.001)
net_long = convNet.to(device)
convNet = train_model(convNet , dataloaders=dataloaders , optimizer=optimizer1)
model3 = ML_autoencoder(20).to(device) #20 is code size, can be hard coded into the model
model3 = trainML_AE(model3,40, train_dataloader)
```





Evaluation

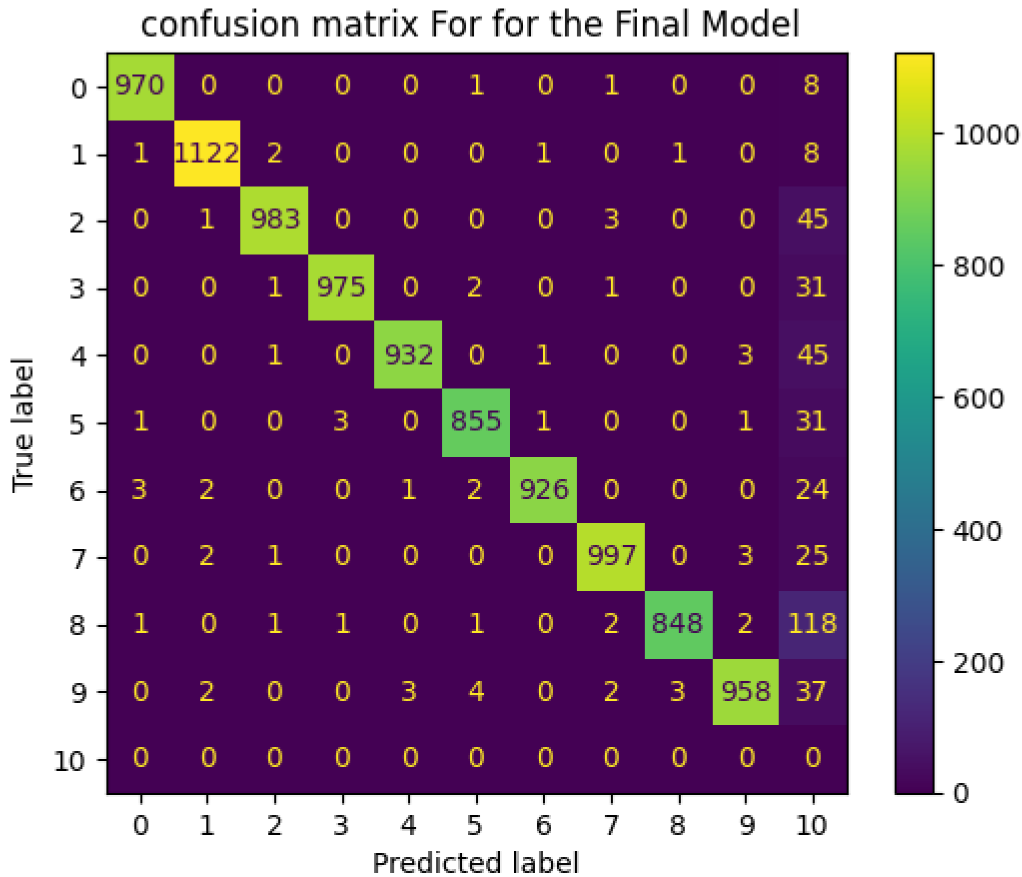
Epoch

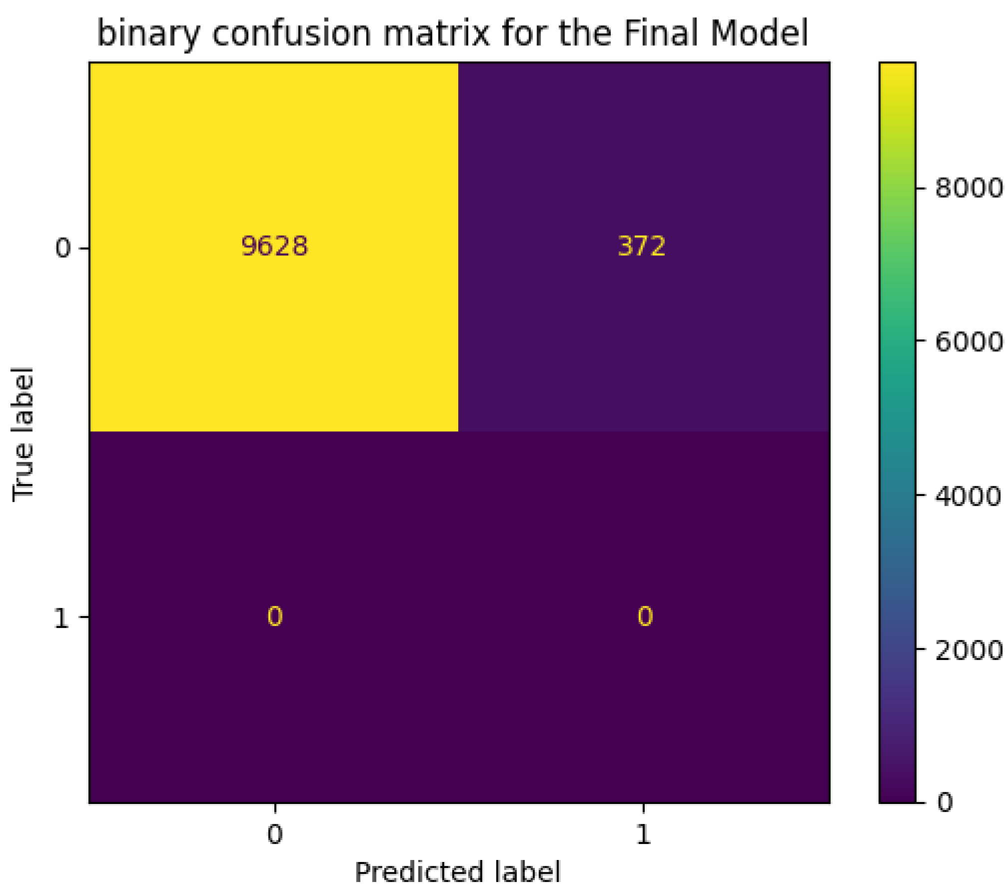
```
from sklearn import metrics
from sklearn.metrics import confusion_matrix

```

```
def test_model(model, ae, dataloaders,size):
    model.eval()
    predictionAcc=0
    pred=[]
    target=[]
    pred_odd=[]
    target_odd=[]
    criterion = nn.MSELoss(reduction='sum')
    for i, data in enumerate(dataloaders, 0):
        inputs, labels = data
        for input1,lable1 in zip(inputs,labels):
            AEinput, preds= ae(input1.view(1,28*28).to(device))
            _,preds = torch.max(preds,dim=1)
            AEinput = AEinput.view(1,28,28)
            AEinput = AEinput.to(device)
            input1 = input1.to(device)
            loss = criterion(input1 , AEinput)
            if(loss < 150):
                if(loss >30):
                    CNN_Pred1 = model(AEinput.view(1,28,28),bs=True)
                    _ , AEprediction = torch.max(CNN_Pred1,dim=0)
                    CNN_Pred0 = model(input1,bs=True)
                    _ , prediction1 = torch.max(CNN_Pred0,dim=0)
                    if (AEprediction != prediction1):
                        prediction = torch.tensor(10)
                    else:
                        prediction = AEprediction
                else:
                    CNN_Pred0 = model(input1,bs=True)
                    _ , prediction = torch.max(CNN_Pred0,dim=0)

            else:
                prediction = torch.tensor(10)
            predictionAcc += torch.sum(prediction == lable1).cpu()
            pred.append(prediction.item())
            target.append(lable1.item())
            if(lable1<=9):
                target_odd.append(0)
            else:
                target_odd.append(1)
            if(prediction<=9):
                pred_odd.append(0)
            else:
                pred_odd.append(1)


    print("The Accuracy of the Final Model Is",float((predictionAcc.double() / size * 100)

    confusion_matrix = metrics.confusion_matrix(target, pred)
    cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix, displ
    cm_display.plot()
```

```
    plt.title("confusion matrix For for the Final Model ")
    plt.show()

    confusion_matrix1 = metrics.confusion_matrix(target_odd, pred_odd)
    cm_display1 = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix1, dis
    cm_display1.plot()
    plt.title("binary confusion matrix for the Final Model ")
    plt.show()

print("The baseline Final Model accracy and he's confusion matrix")
test_model(convNet , model3 , baseline_test_dataloader , len(test_set))
print("The OSR Final Model accracy and he's confusion matrix")
test_model(convNet , model3 , test_dataloader , len(test))


```

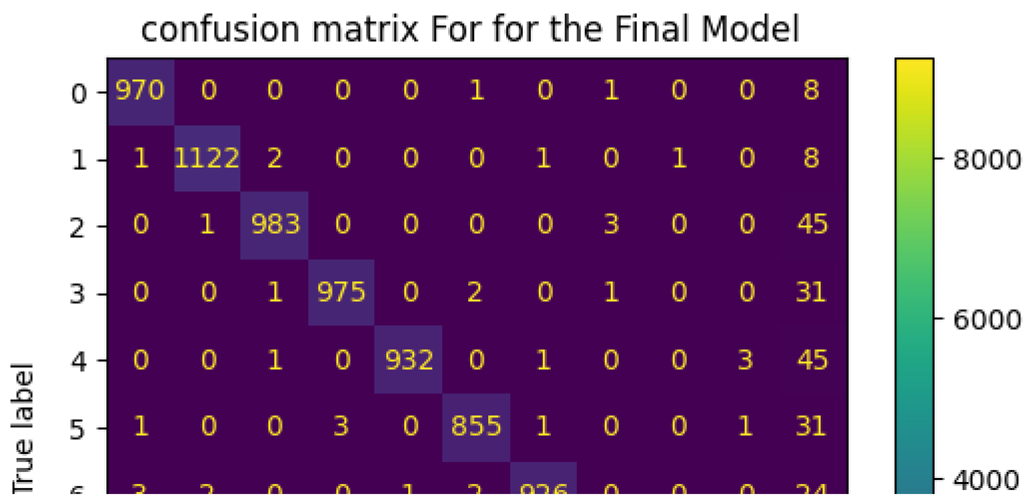
```
The baseline Final Model accracy and he's confusion matrix
The Accuracy of the Final Model Is 95.66 %
```






```
The OSR Final Model accracy and he's confusion matrix
The Accuracy of the Final Model Is 94.11 %
```

The Accuracy of the Final Model is 94.11 %




```
final_model =[convNet,model3]

def eval_model(model1, data_loader, device):
    """ Evaluation function for the OSR task.
    Given your OSR predictions, comptues the accuracy on MNIST, OOD set and both.
    Note - this function does NOT computes the MNIST baseline accruacy.
    Returns:
     - acc_mnist
     - acc_ood
     - acc_total
    """
    criterion = nn.MSELoss(reduction='sum')
    correct_mnist = 0
    total_mnist = 0
    correct_ood = 0
    total_ood = 0
    # No need to track gradients for evaluation, saves memory and computations
    with torch.no_grad():
        for data, labels in data_loader:
            data, labels = data.to(device), labels.to(device)
            #outputs = model(data)

            ### Modify output if needed ###
            ae = model1[1]
            # Ensure model is in evaluation mode
            ae.eval()
            model = model1[0]
            # Ensure model is in evaluation mode
            model.eval()
            for input1,lable1 in zip(data,labels):
                #working with each picture at the time to calculate the loss individually
                #slower work but more accurate for each input
                AEinput, preds= ae(input1.view(1,28*28).to(device))
                AEinput = AEinput.view(1,28,28)
                AEinput = AEinput.to(device)
                input1 = input1.to(device)
                loss = criterion(input1 , AEinput)
```

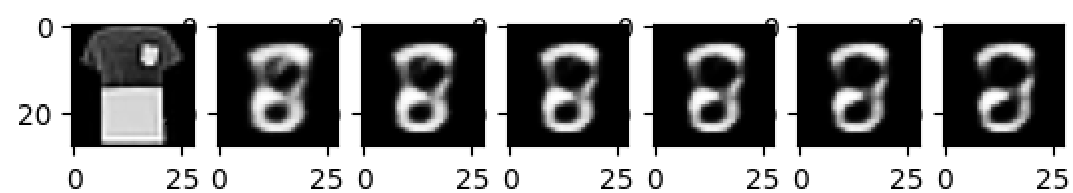
```
                if(loss < 150):
                    if(loss >30):
                        CNN_Pred1 = model(AEinput.view(1,28,28),bs=True)
                        _ , AEprediction = torch.max(CNN_Pred1,dim=0)
                        CNN_Pred0 = model(input1,bs=True)
                        _ , prediction1 = torch.max(CNN_Pred0,dim=0)
                        if (AEprediction != prediction1):
                            prediction = torch.tensor(10)
                        else:
                            prediction = AEprediction
                    else:
                        CNN_Pred0 = model(input1,bs=True)
                        _ , prediction = torch.max(CNN_Pred0,dim=0)

                else:
                    prediction = torch.tensor(10)
                if lable1==10:
                    correct_ood += torch.sum(prediction == lable1).cpu()
                    total_ood +=1
                else:
                    correct_mnist += torch.sum(prediction == lable1).cpu()
                    total_mnist +=1


            ### Modify output if needed ###

            # y pred should be a vector of size (N_batch,) -> [5, 2, ..., 10]
            # and not one-hot. You can handle this either in your model or here.

            # Assuming the model returns an (N_batch, 11) size output
            #probas, y_pred = torch.max(outputs, 1)

            # Assuming the model retuns the predicted label (N_batch, )
            #y_pred = outputs

            # Split MNIST and OOD predictions and labels
            # Assuming numerical labels, which is MNIST/CIFAR datasets default
            # Note: Not one-hot!
    acc_mnist = correct_mnist / total_mnist
    acc_ood = correct_ood / total_ood
    acc_total = (correct_mnist + correct_ood) / (total_mnist + total_ood)

    return acc_mnist.item(), acc_ood.item(), acc_total.item()


acc_mnist, acc_ood, acc_total=eval_model(final_model, test_dataloader, device)
print(acc_mnist, acc_ood, acc_total)
```

```
0.95660001039505 0.925599992275238 0.941100001335144
```

```
print(f'MNIST Accuracy: {acc_mnist*100:.2f}%')
print(f'OOD Accuracy: {acc_ood*100:.2f}%')
print(f'Total Accuracy: {acc_total*100:.2f}%')
```

```
MNIST Accuracy: 95.66%
OOD Accuracy: 92.56%
Total Accuracy: 94.11%
```

```
fig1, axes1 = plt.subplots(1,7)

pic, preds= model3(Fashion[120][0].view(1,28*28).to(device))
pic1 , _ =model3(pic)
pic2 , _ =model3(pic1)
pic3 , _ =model3(pic2)
pic4 , _ =model3(pic3)
pic5 , _ =model3(pic4)
pic = pic.detach().cpu().reshape(28, 28)
pic1= pic1.detach().cpu().reshape(28, 28)
pic2= pic2.detach().cpu().reshape(28, 28)
pic3= pic3.detach().cpu().reshape(28, 28)
pic4= pic4.detach().cpu().reshape(28, 28)
pic5= pic5.detach().cpu().reshape(28, 28)

axes1[0].imshow(Fashion[120][0].permute(1, 2, 0), cmap="gray")
axes1[1].imshow(pic, cmap="gray")
axes1[2].imshow(pic1, cmap="gray")
axes1[3].imshow(pic2, cmap="gray")
axes1[4].imshow(pic3, cmap="gray")
axes1[5].imshow(pic4, cmap="gray")
axes1[6].imshow(pic5, cmap="gray")

plt.show()
```




```
import torch

# Assuming you have a CNN model named "model3" that you want to download its weights

# Step 1: Save the model's state dictionary to a file in the Colab environment
# Replace 'model3_weights.pth' with the desired filename for the weights file
torch.save(model3.state_dict(), 'model3_weights.pth')
torch.save(convNet.state_dict(), 'convNet_weights.pth')


```

1