Homework 1

Xinyuan Lyu

September 2024

Problem 1

In [1]:
```
import pandas as pd
import numpy as np
```

Firstly, load the data and process.

Process the data for (1.)

In [2]:
```
df = pd.read_csv('C:/Users/14857/Desktop/ECE685/homework/HW1/code/archive/Concrete_Data_Yeh.csv')
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values
print(df.head())
print(X.shape,y.shape)
```

```
   cement   slag  flyash  water  superplasticizer  coarseaggregate  \
0   540.0    0.0     0.0  162.0               2.5           1040.0
1   540.0    0.0     0.0  162.0               2.5           1055.0
2   332.5  142.5     0.0  228.0               0.0            932.0
3   332.5  142.5     0.0  228.0               0.0            932.0
4   198.6  132.4     0.0  192.0               0.0            978.4

   fineaggregate  age  csMPa
0          676.0   28  79.99
1          676.0   28  61.89
2          594.0  270  40.27
3          594.0  365  41.05
4          825.5  360  44.30
(1030, 8) (1030,)
```

Process the data for (2.)

In [3]:
```
np.random.seed(10)
indicates = np.arange(X.shape[0])
np.random.shuffle(indicates) #shuffle the indicates of the data.

split_point = int(X.shape[0] * (1 - 0.25)) #get the split point and split the data randomly to test and train
X_train, X_test = X[indicates[:split_point]], X[indicates[split_point:]]
y_train, y_test = y[indicates[:split_point]], y[indicates[split_point:]]

X_trains={}
X_tests={}
models = {}
mse_values = {}

#Generate the data with different dimention of X, i.e. with different number of independent variables.
for i in range(5, 9):
    X_trains[i] = X_train[:, :i]
    X_tests[i] = X_test[:, :i]
    print(X_trains[i].shape,X_tests[i].shape)
```

```
(772, 5) (258, 5)
(772, 6) (258, 6)
(772, 7) (258, 7)
(772, 8) (258, 8)
```

Secondly, construct the model. Using two method.

linear_regression_model_direct_computer uses an analytical solution when solving for parameters, which means it directly calculates them using an explicit optimal formula.

$L(\beta) = (Y - \hat{Y})^2$

$L(\beta) = (Y - \hat{Y})^T(Y - \hat{Y}) = (Y - X\beta)^T(Y - X\beta) = Y^TY - Y^TX\beta - \beta^TX^TY + (X\beta)^TX\beta$

$= Y^TY - \beta^TX^TY - Y^TX\beta + \beta^TX^TX\beta$

$\nabla L(\beta) = -2X^TY + 2X^TX\beta$

Make $\nabla L(\beta) = 0$ so,

The optimal matrix form for β in linear regression is given by:

$$\beta = (X^TX)^{-1}X^Ty$$

In [4]:
```
class linear_regression_model_direct_computer:

    def __init__(self, X, y):
        self.X_matrix = np.hstack([np.ones((X.shape[0], 1)), X])  #Load X and Add the constant term for the model.
        self.y_vector = y  #Load y.
        self.beta = None      #Initialize the beta.
        self.mse = 0    #Initialize the mse.
        self.X_test=None  #Initialize the test X data.
```

```
        self.y_test=None  #Initialize the test y data.
        self.mse_test=0   #Initialize the test mse.

    def compute_beta(self):
        self.beta = np.linalg.inv(self.X_matrix.T @ self.X_matrix) @ self.X_matrix.T @ self.y_vector #Compute the best beta derived from math
        self.y_predict = self.X_matrix @ self.beta #Get the prediction
        return self.beta

    def compute_mse(self):
        self.mse = np.mean((self.y_vector - self.X_matrix @ self.beta) ** 2) #compute the mse
        return self.mse

    def test_model(self,X_test,y_test):
        self.X_test=np.hstack([np.ones((X_test.shape[0], 1)), X_test]) #Load Test data and Add the constant term for the model.
        self.y_test=y_test
        self.mse_test = np.mean((self.y_test - self.X_test @ self.beta) ** 2) #Compute the mse on test dataset.
        return self.mse_test
```

linear_regression_model_direct_trainer uses Gradient Descend when solving for parameters.

The Gradient of β for mse loss function is given by:

$$\frac{1}{n}\left(X^T \cdot (X\beta - y)\right)$$

In [5]:
```
class linear_regression_model_direct_trainer:

    def __init__(self, X, y, n_epochs, learn_rate):
        self.X_matrix = np.hstack([np.ones((X.shape[0], 1)), X])  #Load X and Add the constant term for the model.
        self.y_vector = y  #Load y.
        self.beta = None      #Initialize the beta.
        self.y_predict = None #Initialize the preddiction.
        self.mse = 0    #Initialize the mse.
        self.X_test=None  #Initialize the test X data.
        self.y_test=None  #Initialize the test y data.
        self.mse_test=0   #Initialize the test mse.
        self.n_epochs=n_epochs #Set number of epochs
        self.learn_rate=learn_rate #Set learning rate

    def compute_beta(self):
        self.beta = np.zeros(self.X_matrix.shape[1]) #Initialize the beta.
        n=self.X_matrix.shape[0] #get the n which is the sample size.

        for epoch in range(self.n_epochs):
            loss = np.mean((self.X_matrix @ self.beta - self.y_vector) ** 2) #Define the loss function
            self.beta = self.beta - self.learn_rate*(1/n)*(self.X_matrix.T @ (self.X_matrix @ self.beta - self.y_vector)) #The gradient desce
            print(loss)
        return self.beta

    def compute_mse(self):
        self.mse = np.mean((self.y_vector - self.X_matrix @ self.beta) ** 2)
        return self.mse

    def test_model(self,X_test,y_test):
        self.X_test=np.hstack([np.ones((X_test.shape[0], 1)), X_test])
        self.y_test=y_test
        self.mse_test = np.mean((self.y_test - self.X_test @ self.beta) ** 2)
        return self.mse_test
```

Thirdly, use the data and model to solve (1.) and (2.).

For (1.), use two model defined as above. And get the beta as well as MSE.

In [6]:
```
model_with_all_independent_variables_compute=linear_regression_model_direct_computer(X,y) # Use the full data to train the model.
print(f'The optimal beta {model_with_all_independent_variables_compute.compute_beta()}')
print(f'The final mse {model_with_all_independent_variables_compute.compute_mse()}')
```

```
The optimal beta [-2.33312136e+01  1.19804334e-01  1.03865809e-01  8.79343215e-02
 -1.49918419e-01  2.92224595e-01  1.80862148e-02  2.01903511e-02
  1.14222068e-01]
The final mse 107.19723607486017
```

In [7]:
```
%%capture
model_with_all_independent_variables_train=linear_regression_model_direct_trainer(X,y,100000,0.000001) #100000,0.000001 are number of epochs
model_with_all_independent_variables_train.compute_beta()
```

In [8]:
```
print(f'The optimal beta {model_with_all_independent_variables_train.beta}')
print(f'The final mse {model_with_all_independent_variables_train.compute_mse()}')
```

```
The optimal beta [-0.00026361  0.11510396  0.09830647  0.08285372 -0.18885138  0.20432239
  0.0100605   0.01256648  0.11421329]
The final mse 107.32757392939513
```

We can see that Model use Gradient Descend may has the mse close to Model use analytical solution, but cannot be same as it.

For (2.), we choose i=6,7,8,9 (including the constant term). We may also use two models.

In [9]:
```
for i in range(5, 9):
    X_train = X_trains[i]
    X_test = X_tests[i]
    models[i] = linear_regression_model_direct_computer(X_train, y_train)
    models[i].compute_beta()
    mse_values[i]=models[i].test_model(X_test,y_test)

print(f"Model with 6 characters (including constant term) MSE: {mse_values[5]}")
print(f"Model with 7 characters (including constant term) MSE: {mse_values[6]}")
print(f"Model with 8 characters (including constant term) MSE: {mse_values[7]}")
print(f"Model with 9 characters (including constant term) MSE: {mse_values[8]}")
```

```
Model with 6 characters (including constant term) MSE: 159.64197170664195
Model with 7 characters (including constant term) MSE: 159.1847643187316
Model with 8 characters (including constant term) MSE: 159.19128766119093
Model with 9 characters (including constant term) MSE: 108.20657336621308
```

In [10]:
```
%%capture
for i in range(5, 9):
    X_train = X_trains[i]
    X_test = X_tests[i]
    models[i] = linear_regression_model_direct_trainer(X_train, y_train,100000,0.000001)
    models[i].compute_beta()
    mse_values[i]=models[i].test_model(X_test,y_test)
```

In [11]:
```
print(f"Model with 6 characters (including constant term) MSE: {mse_values[5]}")
print(f"Model with 7 characters (including constant term) MSE: {mse_values[6]}")
print(f"Model with 8 characters (including constant term) MSE: {mse_values[7]}")
print(f"Model with 9 characters (including constant term) MSE: {mse_values[8]}")
```

```
Model with 6 characters (including constant term) MSE: 160.69504467726168
Model with 7 characters (including constant term) MSE: 157.51131666603607
Model with 8 characters (including constant term) MSE: 158.89061737211492
Model with 9 characters (including constant term) MSE: 107.87370194419836
```

From the result we can get the conclusion that:

(1) Models with 6,7,8 characters (including constant term) have very close MSE, and even increasing.

(2) And Model from 8 to 9 have a large descend in MSE.

So it shows that:

It is Not the case that models with more independent variables always perform better.

From (2) I think that character 9 is very important for impacting or predicting y than other variables like 7,8.

And From (1) I think that the increase in MSE suggests that 7,8 may be irrelevant features and cause overfitting or noise for model and make model perform worse.

This highlights the importance of selecting key features rather than simply increasing the number of features.

End of Problem 1.

In [12]:
```
%reset -f
```

Problem 2

In [13]:
```
import torch
import torchvision
import matplotlib.pyplot as plt
import torch.nn.functional as F
import torch.nn as nn
import torch.optim as optim
from IPython import display
from torchvision import datasets, transforms
from torch.utils import data
from torch.utils.data import DataLoader, Dataset, TensorDataset

from Encoder import extractor # import the extractor which is given.
```

Firstly, we load the extractor as the (1.) required.

In [14]:
```
%%capture
from Encoder import extractor

my_extractor=extractor()
extractor_weights = 'C:/Users/14857/Desktop/ECE685/homework/HW1/code/feature_extractor_weights.pth'
```

```
my_extractor.load_state_dict(torch.load(extractor_weights))
my_extractor.eval()
```

Next, we download the MNIST data and transform it to tensor as well as normalizing the data.

In [15]:
```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

Batch_size=32
train_data = torchvision.datasets.MNIST("./data", train=True, download=True, transform=transform)
test_data = torchvision.datasets.MNIST("./data", train=False, download=True, transform=transform)
train_data_loader = DataLoader(train_data, batch_size=Batch_size, shuffle=True)
test_data_loader = DataLoader(test_data, batch_size=Batch_size, shuffle=False)
```

Secondly, we use the extractor to extract the latent representations as (2.) required and generate the datasets for the next step of training.

In [16]:
```
train_features = [] #set empty list.
train_labels = []
test_features = []
test_labels = []

with torch.no_grad():
    for images, label in train_data_loader:
        feature = my_extractor(images)
        train_features.append(feature)
        train_labels.append(label)

train_features = torch.cat(train_features, dim=0) #combine the tensors i.e. make the list to a whole tensoor.
train_labels = torch.cat(train_labels, dim=0) #combine the tensors i.e. make the list to a whole tensoor.

with torch.no_grad():
    for images, label in test_data_loader:
        feature = my_extractor(images)
        test_features.append(feature)
        test_labels.append(label)

test_features = torch.cat(test_features, dim=0) #combine the tensors i.e. make the list to a whole tensoor.
test_labels = torch.cat(test_labels, dim=0) #combine the tensors i.e. make the list to a whole tensoor.
```

Packaging the features tensor and labels tensor to dataset. And then transfor to the dataloader.

In [17]:
```
extracted_train_data = TensorDataset(train_features, train_labels)
extracted_test_data = TensorDataset(test_features, test_labels)

Batch_size=32
extracted_train_data_loader = DataLoader(extracted_train_data, batch_size=Batch_size, shuffle=True) # Shuffle is true for the SGD later.
extracted_test_data_loader = DataLoader(extracted_test_data, batch_size=Batch_size, shuffle=False)
```

Thirdly, define the model, softmax function, cross entropy loss function and the gradient of loss respect to W and B, as (3.) required.

Softmax Regression Gradient Derivation, where y is the one-hot vector of labels of images, and y_k means the k^{th} entry of y.

$$\mathbb{P}(Y|H, w_1, \cdots, w_k, b) = \prod_{n=1}^{N}\prod_{k=1}^{K}\mathbb{P}(C_k|h_n)^{y_{nk}} = \prod_{n=1}^{N}\prod_{k=1}^{K}\hat{y}_{nk}^{y_{nk}}$$

where

$$\hat{y}_{nk} = \mathbb{P}(C_k|h_n) = \frac{\exp(w_k^T h_n + b)}{\sum_j \exp(w_j^T h_n + b)}$$

The loss function $E(w_1, \cdots, w_k, b)$ is the negative log-likelihood:

$$E(w_1, \cdots, w_k, b) = -\ln \mathbb{P}(Y|H, w_1, \cdots, w_k, b)$$

$$= -\sum_{n=1}^{N}\sum_{k=1}^{K} y_{nk} \ln \hat{y}_{nk}$$

$$= -\sum_{n=1}^{N}\sum_{k=1}^{K} y_{nk} \ln \left(\frac{\exp(w_k^T h_n + b)}{\sum_j \exp(w_j^T h_n + b)} \right)$$

$$= -\sum_{n=1}^{N}\sum_{k=1}^{K} \left[y_{nk} \left(w_k^T h_n + b - \ln \sum_j \exp(w_j^T h_n + b) \right) \right]$$

$$= \sum_{n=1}^{N} \left[\ln \sum_{j} \exp(w_j^T h_n + b) - \sum_{k=1}^{K} y_{nk} (w_k^T h_n + b) \right]$$

So the averge loss of every sample point is

$$= \frac{1}{N} \cdot \sum_{n=1}^{N} \left[\ln \sum_{j} \exp(w_j^T h_n + b) - \sum_{k=1}^{K} y_{nk} (w_k^T h_n + b) \right]$$

For the gradient w.r.t. (w_i):

$$\nabla AvgE_{w_i} = \frac{1}{N} \sum_{n=1}^{N} \left(\frac{\exp(w_i^T h_n + b)}{\sum_j \exp(w_j^T h_n + b)} h_n - y_{ni} h_n \right)$$

$$= \frac{1}{N} \sum_{n=1}^{N} ((\hat{y}_{ni} - y_{ni}) h_n)$$

For the gradient w.r.t. (b_i):

$$\nabla AvgE_{b_i} = \frac{1}{N} \sum_{n=1}^{N} \left(\frac{\exp(w_i^T h_n + b)}{\sum_j \exp(w_j^T h_n + b)} - y_{ni} \right)$$

$$= \frac{1}{N} \sum_{n=1}^{N} (\hat{y}_{ni} - y_{ni})$$

So we can write the code to realize the math derivation.

In [18]:
```
def softmax(X): # Define the softmax function as the defination.
    X_exp = torch.exp(X)
    partition = X_exp.sum(1, keepdim=True)
    return X_exp / partition

def model(X,W,b): # Define the softmax regression model.
    return softmax(torch.matmul(X, W) + b)

def to_one_hot(y,num_classes): # Define the transformation function for y to the one hot code of y.
    return torch.nn.functional.one_hot(y, num_classes=num_classes)

def cross_entropy(y_hat, y): # Define the cross entropy loss function.
    return - torch.log(y_hat[range(len(y_hat)), y])

def loss(y_hat,y): # So the loss function is the sum of cross entropy from 1 to n.
    return (cross_entropy(y_hat, y).sum())/Batch_size

def gradient_W(X,y_hat,y_one_hot): # The gradient w.r.t. w is as derived.
    return (torch.matmul(X.T, (y_hat - y_one_hot)))/Batch_size

def gradient_b(X,y_hat,y_one_hot): # The gradient w.r.t. b is as derived.
    return ((y_hat - y_one_hot).sum(dim=0))/Batch_size
```

Initialize the weight and bias, and set the learning rate and number of epochs, and use GPU to train the model.

In [19]:
```
num_inputs = 256 # The dimension of the X i.e. the features of the images.
num_outputs = 10 # The dimension of the one-hot vector of y i.e. y has 10 classes.
learn_rate = 0.001 # set the learning rate.
num_epochs = 50

W = torch.normal(0, 0.01, size=(num_inputs, num_outputs), requires_grad=True)
b = torch.zeros(size=(1, num_outputs), requires_grad=True)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
W = W.to(device)
b = b.to(device)

train_losses = [] # Prepare for the plotting later.
test_losses = []
train_accuracies = []
test_accuracies = []
```

Forthly, train the data and plot the loss as well as accuracy in train data and test data for each epoch, as the (4.) required.

And we will show two method to optimize the model W and b, the First is based on the math and the gradients obtained in (3.) , and the Second is based on the Automatic Differentiation function in PyTorch.

(a.) The First method, manually derive the Gradient and then train the model i.e. from scratch.

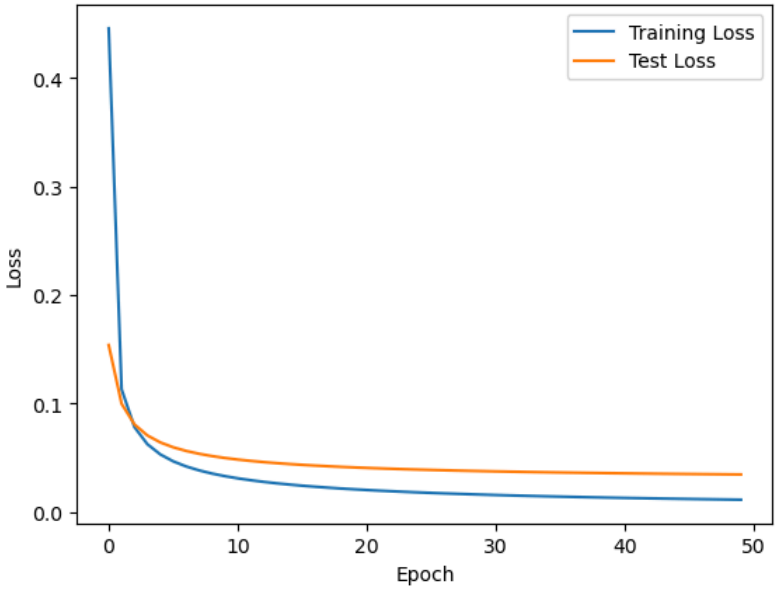
In [20]:
```
for epoch in range(num_epochs):

    total_train_loss = 0 #initialize the loss and accuracy in each ephoch.
    correct_train = 0
    total_train = 0

    for i, data in enumerate(extracted_train_data_loader):

        X, y=data
        X = X.to(device)
        y = y.to(device)

        if X.shape != (Batch_size,num_inputs): # Avoid the shape not matching.
            break

        y_one_hot = to_one_hot(y,10).to(device)
        y_hat = model(X,W,b).to(device)

        ls_train = loss(y_hat, y)
        total_train_loss += ls_train.item()

        W = W - learn_rate * gradient_W(X,y_hat,y_one_hot)
        b = b - learn_rate * gradient_b(X,y_hat,y_one_hot)

        predicted_train = y_hat.argmax(dim=1)
        correct_train += (predicted_train == y).sum().item()
        total_train += y.size(0)

    average_train_loss = total_train_loss / len(extracted_train_data_loader)
    train_losses.append(average_train_loss)
    train_accuracy = correct_train / total_train
    train_accuracies.append(train_accuracy)


    # In each epoch, we also test the model in test data.
    with torch.no_grad():

        total_test_loss = 0
        correct_test = 0
        total_test = 0

        for i, data in enumerate(extracted_test_data_loader):

            X_test, y_test=data
            X_test = X_test.to(device)
            y_test = y_test.to(device)

            if X_test.shape != (Batch_size,num_inputs):
                break

            y_hat_test = model(X_test,W,b)
            test_loss = loss(y_hat_test, y_test)

            total_test_loss += test_loss.item()
            predicted_test = y_hat_test.argmax(dim=1)
            correct_test += (predicted_test == y_test).sum().item()
            total_test += y_test.size(0)

        average_test_loss = total_test_loss / len(extracted_test_data_loader)
        test_losses.append(average_test_loss)
        test_accuracy = correct_test / total_test
        test_accuracies.append(test_accuracy)

    if epoch % 5 == 0: # every 5 epochs print the loss and accuracy.
        print(f'Epoch {epoch+1}, Train Loss: {average_train_loss:.4f}, Test Loss: {average_test_loss:.4f}, Train Accuracy: {train_accuracy:.4
```

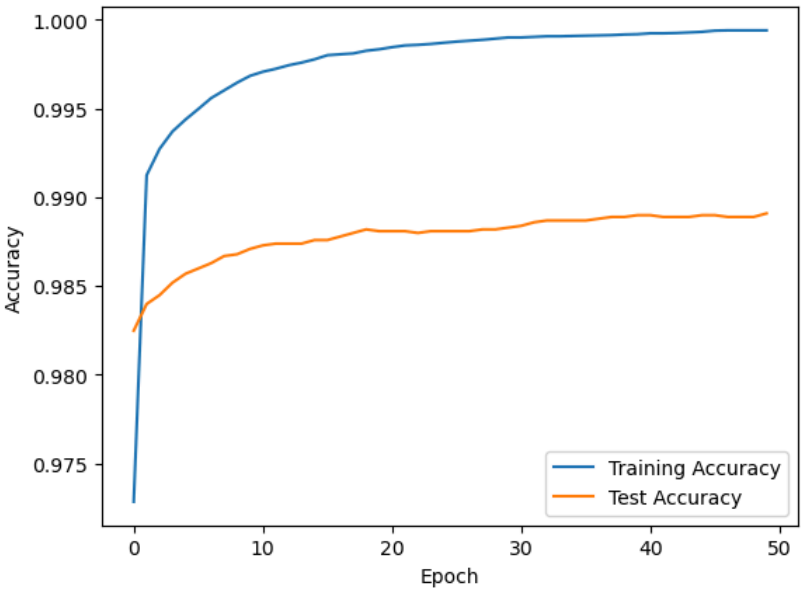
```
Epoch 1, Train Loss: 0.4460, Test Loss: 0.1539, Train Accuracy: 0.9728, Test Accuracy: 0.9825
Epoch 6, Train Loss: 0.0466, Test Loss: 0.0596, Train Accuracy: 0.9950, Test Accuracy: 0.9860
Epoch 11, Train Loss: 0.0309, Test Loss: 0.0483, Train Accuracy: 0.9971, Test Accuracy: 0.9873
Epoch 16, Train Loss: 0.0241, Test Loss: 0.0434, Train Accuracy: 0.9980, Test Accuracy: 0.9876
Epoch 21, Train Loss: 0.0202, Test Loss: 0.0405, Train Accuracy: 0.9984, Test Accuracy: 0.9881
Epoch 26, Train Loss: 0.0175, Test Loss: 0.0387, Train Accuracy: 0.9988, Test Accuracy: 0.9881
Epoch 31, Train Loss: 0.0155, Test Loss: 0.0373, Train Accuracy: 0.9990, Test Accuracy: 0.9884
Epoch 36, Train Loss: 0.0140, Test Loss: 0.0363, Train Accuracy: 0.9991, Test Accuracy: 0.9887
Epoch 41, Train Loss: 0.0129, Test Loss: 0.0355, Train Accuracy: 0.9992, Test Accuracy: 0.9890
Epoch 46, Train Loss: 0.0119, Test Loss: 0.0349, Train Accuracy: 0.9994, Test Accuracy: 0.9890
```

Plot the loss and accuracy for each epoch in the trainning process.

In [21]:
```
def plot_the_model(y1, y2, type='loss'):
    plt.figure()
    if type == 'loss':
        plt.plot(y1, label='Training Loss')
        plt.plot(y2, label='Test Loss')
        plt.ylabel('Loss')
    elif type == 'accuracy':
        plt.plot(y1, label='Training Accuracy')
        plt.plot(y2, label='Test Accuracy')
        plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend()
    plt.show()
```

```
plot_the_model(train_losses, test_losses, type='loss')
plot_the_model(train_accuracies, test_accuracies, type='accuracy')
```





(b.) The Second method, use the package in pytorch and use SGD to train the model.

In [22]:
```
W = torch.normal(0, 0.01, size=(num_inputs, num_outputs), requires_grad=True, device=device)
b = torch.zeros(size=(1, num_outputs), requires_grad=True, device=device)
optimizer = optim.SGD([W, b], lr=learn_rate, weight_decay=0.001) # Define the optimizer. And we set weight_decay=0.001 to avoid the overfitti
```

Others are all the same except the optimizer.

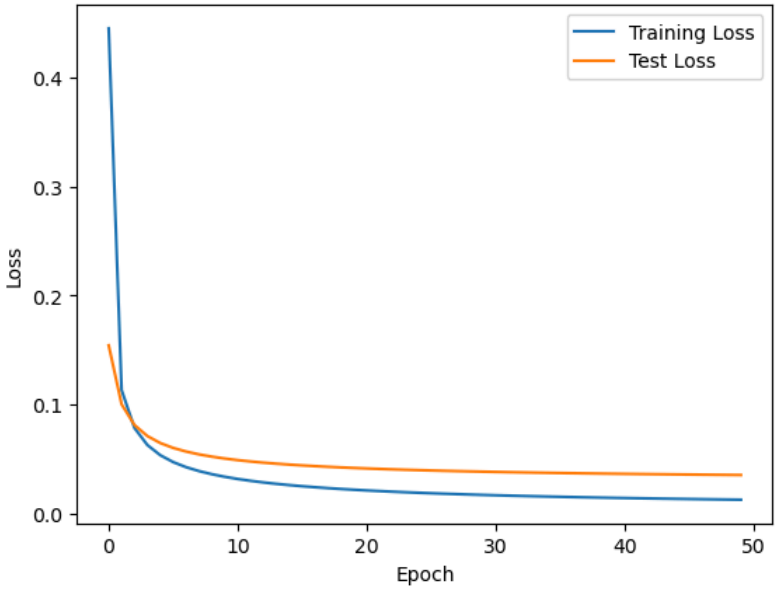
In [23]:
```
train_losses = [] # Prepare for the plotting later.
test_losses = []
train_accuracies = []
test_accuracies = []

for epoch in range(num_epochs):

    total_train_loss = 0 #initialize the loss and accuracy in each ephoch.
    correct_train = 0
    total_train = 0

    for i, data in enumerate(extracted_train_data_loader):

        X, y=data
        X = X.to(device)
        y = y.to(device)

        if X.shape != (Batch_size,num_inputs): # Avoid the shape not matching.
            break

        y_one_hot = to_one_hot(y,10).to(device)
        y_hat = model(X,W,b).to(device)

        ls_train = loss(y_hat, y)
        total_train_loss += ls_train.item()

        optimizer.zero_grad() #Use the optimizer.
        ls_train.backward()
```

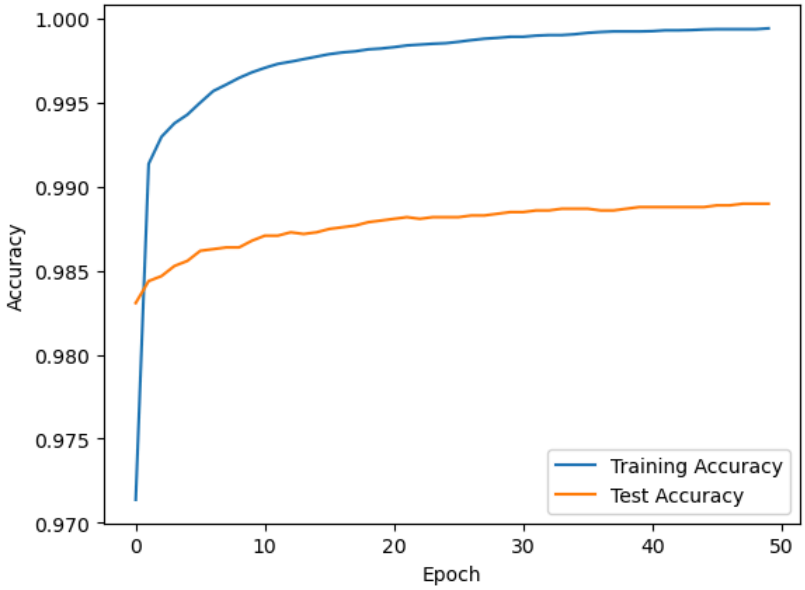
```
        optimizer.step()

        predicted_train = y_hat.argmax(dim=1)
        correct_train += (predicted_train == y).sum().item()
        total_train += y.size(0)

    average_train_loss = total_train_loss / len(extracted_train_data_loader)
    train_losses.append(average_train_loss)
    train_accuracy = correct_train / total_train
    train_accuracies.append(train_accuracy)


    # In each epoch, we also test the model in test data.
    with torch.no_grad():

        total_test_loss = 0
        correct_test = 0
        total_test = 0

        for i, data in enumerate(extracted_test_data_loader):

            X_test, y_test=data
            X_test = X_test.to(device)
            y_test = y_test.to(device)

            if X_test.shape != (Batch_size,num_inputs):
                break

            y_hat_test = model(X_test,W,b)
            test_loss = loss(y_hat_test, y_test)

            total_test_loss += test_loss.item()
            predicted_test = y_hat_test.argmax(dim=1)
            correct_test += (predicted_test == y_test).sum().item()
            total_test += y_test.size(0)

        average_test_loss = total_test_loss / len(extracted_test_data_loader)
        test_losses.append(average_test_loss)
        test_accuracy = correct_test / total_test
        test_accuracies.append(test_accuracy)

    if epoch % 5 == 0: # every 5 epochs print the loss and accuracy.
        print(f'Epoch {epoch+1}, Train Loss: {average_train_loss:.4f}, Test Loss: {average_test_loss:.4f}, Train Accuracy: {train_accuracy:.4
```

```
Epoch 1, Train Loss: 0.4452, Test Loss: 0.1543, Train Accuracy: 0.9714, Test Accuracy: 0.9831
Epoch 6, Train Loss: 0.0471, Test Loss: 0.0601, Train Accuracy: 0.9950, Test Accuracy: 0.9862
Epoch 11, Train Loss: 0.0316, Test Loss: 0.0489, Train Accuracy: 0.9971, Test Accuracy: 0.9871
Epoch 16, Train Loss: 0.0249, Test Loss: 0.0440, Train Accuracy: 0.9979, Test Accuracy: 0.9875
Epoch 21, Train Loss: 0.0211, Test Loss: 0.0412, Train Accuracy: 0.9983, Test Accuracy: 0.9881
Epoch 26, Train Loss: 0.0185, Test Loss: 0.0394, Train Accuracy: 0.9986, Test Accuracy: 0.9882
Epoch 31, Train Loss: 0.0166, Test Loss: 0.0381, Train Accuracy: 0.9989, Test Accuracy: 0.9885
Epoch 36, Train Loss: 0.0152, Test Loss: 0.0371, Train Accuracy: 0.9991, Test Accuracy: 0.9887
Epoch 41, Train Loss: 0.0141, Test Loss: 0.0363, Train Accuracy: 0.9992, Test Accuracy: 0.9888
Epoch 46, Train Loss: 0.0132, Test Loss: 0.0357, Train Accuracy: 0.9994, Test Accuracy: 0.9889
```

In [24]:

```
def plot_the_model(y1, y2, type='loss'):
    plt.figure()
    if type == 'loss':
        plt.plot(y1, label='Training Loss')
        plt.plot(y2, label='Test Loss')
        plt.ylabel('Loss')
    elif type == 'accuracy':
        plt.plot(y1, label='Training Accuracy')
        plt.plot(y2, label='Test Accuracy')
        plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend()
    plt.show()

plot_the_model(train_losses, test_losses, type='loss')
plot_the_model(train_accuracies, test_accuracies, type='accuracy')
```





From the data on the loss function and accuracy on both the training set and test set, we can see that our model exhibits good properties, achieving an accuracy of up to 98% on the test set. Additionally, it is evident that both optimization methods—manual gradient calculation and manual implementation of gradient descent, as well as using PyTorch's automatic differentiation and optimization—are able to effectively train the model. The difference lies in the fact that the second method is more concise and maybe allows the addition of regularization to mitigate overfitting.

End of Problem 2.

Problem 3

I will give two solutions for 1, the first is based on the change of variables rule and the other is based on the pdf of Z=XY where X,Y is independent.

1.

Firstly, we have:

$$\begin{aligned} F_R(r) &= \mathbb{P}(R \le r) = \mathbb{P}(R^2 \le r^2) \quad (\text{since } R \ge 0) \\ &= \mathbb{P}(-2\ln U_2 \le r^2) = \mathbb{P}(\ln U_2 \ge -\frac{r^2}{2}) = \mathbb{P}(U_2 \ge e^{-\frac{r^2}{2}}) \\ &= 1 - \mathbb{P}(U_2 \le e^{-\frac{r^2}{2}}) = 1 - \exp\left(-\frac{r^2}{2}\right) \end{aligned}$$

$$\text{So } f_R(r) = F_R'(r) = r \cdot \exp\left(-\frac{r^2}{2}\right), \quad r \in (0, +\infty)$$

Let $\Theta = 2\pi \cdot U_1$, so $\Theta \sim U(0, 2\pi)$, so

$$F_\Theta(\theta) = \mathbb{P}(\Theta \le \theta) = \frac{\theta}{2\pi}, \quad x \in (0, 2\pi)$$

$$\text{So } f_\Theta(\theta) = F_\Theta'(\theta) = \frac{1}{2\pi}$$

And We have:

$$R \perp\!\!\!\perp \Theta \quad (\text{since } U_1 \perp\!\!\!\perp U_2)$$

So

$$f_{R,\Theta}(r, \theta) = f_R(r) \cdot f_\Theta(\theta)$$

And we have the transformation from (R, Θ) to (Z_1, Z_2) :

$$Z_1 = R\cos\Theta \quad Z_2 = R\sin\Theta$$

Thus

$$R = \sqrt{Z_1^2 + Z_2^2} \quad \Theta = \arctan\left(\frac{Z_2}{Z_1}\right)$$

So for the Jacobian determinant, we have: Jacobian matrix J is defined as:

$$J = \frac{\partial(R, \Theta)}{\partial(Z_1, Z_2)} = \begin{pmatrix} \frac{\partial R}{\partial Z_1} & \frac{\partial R}{\partial Z_2} \\ \frac{\partial \Theta}{\partial Z_1} & \frac{\partial \Theta}{\partial Z_2} \end{pmatrix}$$

And we can compute each term:

$$\frac{\partial R}{\partial Z_1} = \frac{Z_1}{\sqrt{Z_1^2 + Z_2^2}}, \quad \frac{\partial R}{\partial Z_2} = \frac{Z_2}{\sqrt{Z_1^2 + Z_2^2}}$$
$$\frac{\partial \Theta}{\partial Z_1} = -\frac{Z_2}{Z_1^2 + Z_2^2}, \quad \frac{\partial \Theta}{\partial Z_2} = \frac{Z_1}{Z_1^2 + Z_2^2}$$

Thus the Jacobian determinant is

$$J = \begin{pmatrix} \frac{Z_1}{\sqrt{Z_1^2+Z_2^2}} & \frac{Z_2}{\sqrt{Z_1^2+Z_2^2}} \\ -\frac{Z_2}{Z_1^2+Z_2^2} & \frac{Z_1}{Z_1^2+Z_2^2} \end{pmatrix}$$

Therefore

$$|det(J)| = \frac{1}{\sqrt{Z_1^2 + Z_2^2}}$$

So use the Change of the variables rule:

$$\begin{aligned} f_{Z_1,Z_2}(z_1, z_2) &= f_{R,\Theta}\left(\sqrt{z_1^2 + z_2^2}, \arctan\left(\frac{z_2}{z_1}\right)\right) \cdot |det(J)| \\ &= \frac{1}{2\pi} \exp\left(-\frac{z_1^2 + z_2^2}{2}\right) \cdot \sqrt{z_1^2 + z_2^2} \cdot \left(\frac{1}{\sqrt{z_1^2 + z_2^2}}\right) \\ &= \frac{1}{2\pi} \exp\left(-\frac{z_1^2 + z_2^2}{2}\right) \end{aligned}$$

So for Z_1 *and* Z_2 we have the marginal distribution:
Therefore

$$f_{Z_1}(z_1) = \int_{-\infty}^{\infty} f_{Z_1,Z_2}(z_1, z_2)\, dz_2 = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{z_1^2}{2}\right) \cdot \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{z_2^2}{2}\right)\, dz_2$$

And from the property of normal distribution, we know that

$$\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) dx = 1$$

So

$$f_{Z_1}(z_1) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{z_1^2}{2}\right)$$

Similarly, we have:

$$f_{Z_2}(z_2) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{z_2^2}{2}\right)$$

This is exactly the probability density function of the standard normal distribution.
Thus

$$Z_1 \sim \mathcal{N}(0,1) \quad \text{and} \quad Z_2 \sim \mathcal{N}(0,1) \qquad \square$$

2.

From **1.** we have:

$$f_{Z_1,Z_2}(z_1, z_2) = \frac{1}{2\pi} \exp\left(-\frac{z_1^2 + z_2^2}{2}\right)$$

And we have:

$$f_{Z_1}(z_1) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{z_1^2}{2}\right)$$

$$f_{Z_2}(z_2) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{z_2^2}{2}\right)$$

So it is easy to see that:

$$f_{Z_1,Z_2}(z_1, z_2) = f_{Z_1}(z_1) \cdot f_{Z_2}(z_2)$$

Thus, this means that Z_1 and Z_2 are independent. $\square$

3. The Alternative Solution for 1.

We can also derive the pdf of Z1 and Z2 directly using the fact that R and Θ are independent.

Firstly, we have:

$$\begin{aligned} F_R(r) &= \mathbb{P}(R \le r) = \mathbb{P}(R^2 \le r^2) \quad (\text{since } R \ge 0) \\ &= \mathbb{P}(-2\ln U_2 \le r^2) = \mathbb{P}(\ln U_2 \ge -\frac{r^2}{2}) = \mathbb{P}(U_2 \ge e^{-\frac{r^2}{2}}) \\ &= 1 - \mathbb{P}(U_2 \le e^{-\frac{r^2}{2}}) = 1 - \exp\left(-\frac{r^2}{2}\right) \end{aligned}$$

$$\text{So } f_R(r) = F_R'(r) = r \cdot \exp\left(-\frac{r^2}{2}\right), \quad r \in (0, +\infty)$$

Let $X_1 = \cos\Theta$, $X_2 = \sin\Theta$, then

$$\begin{aligned} F_{X_1}(x) &= \mathbb{P}(X_1 \le x) = \mathbb{P}(\cos(2\pi U_1) \le x) \\ &= \mathbb{P}\left(\arccos x \le 2\pi U_1 \le 2\pi - \arccos x\right), \quad x \in (-1, 1) \\ &= \mathbb{P}\left(\frac{\arccos x}{2\pi} \le U_1 \le 1 - \frac{\arccos x}{2\pi}\right) \\ &= 1 - \frac{\arccos x}{\pi}, \quad x \in (-1, 1) \end{aligned}$$

$$\text{So } f_{X_1}(x) = F_{X_1}'(x) = \left(-\frac{1}{\pi} \cdot -\frac{1}{\sqrt{1-x^2}}\right) = \frac{1}{\pi\sqrt{1-x^2}}, \quad x \in (-1, 1)$$

Similarly,

$$\begin{aligned} F_{X_2}(x) &= \mathbb{P}(X_2 \le x) = \mathbb{P}(\sin(2\pi U_1) \le x) \\ &= \mathbb{P}\left(-\pi - \arcsin x \le 2\pi U_1 \le \arcsin x\right) \\ &= \mathbb{P}\left(-\frac{1}{2} - \frac{\arcsin x}{2\pi} \le U_1 \le \frac{\arcsin x}{2\pi}\right) \\ &= \frac{1}{2} + \frac{\arcsin x}{\pi}, \quad x \in (-1, 1) \end{aligned}$$

$$\text{So } f_{X_2}(x) = F_{X_2}'(x) = \frac{1}{\pi\sqrt{1-x^2}}, \quad x \in (-1, 1)$$

Thus,

$$f_{X_2}(x) = f_{X_1}(x) = \frac{1}{\pi\sqrt{1-x^2}} \quad \boxed{1}$$

And We have:

$$R \perp\!\!\!\perp X_1 \text{ and } R \perp\!\!\!\perp X_2 \quad (\text{since } U_1 \perp\!\!\!\perp U_2) \quad \boxed{2}$$

Therefore we have:

$$Z_1 = RX_1 \implies X_1 = \frac{Z_1}{R} \in (-1,1), \text{ so } R > |Z_1|$$

$$Z_2 = RX_2 \implies X_2 = \frac{Z_2}{R} \in (-1,1), \text{ so } R > |Z_2| \quad \boxed{3}$$

Due to $\boxed{1}$; $\boxed{2}$ and $\boxed{3}$, we get:

$$\begin{aligned}
f_{Z_2}(z) = f_{Z_1}(z) &= \int_{|z|}^{\infty} \frac{1}{|r|} f_{R,X_1}(r, z/r)\, dr \\
&= \int_{|z|}^{\infty} \frac{1}{r} \cdot f_R(r) \cdot f_{X_1}(z/r)\, dr \quad (\text{Since } R \perp\!\!\!\perp X_1) \\
&= \int_{|z|}^{\infty} \frac{1}{r} \cdot r \cdot \exp\left(-\frac{r^2}{2}\right) \cdot \frac{1}{\pi} \cdot \frac{r}{\sqrt{r^2 - z^2}}\, dr \\
&= \int_{|z|}^{\infty} \frac{r \cdot \exp\left(-\frac{r^2}{2}\right)}{\pi\sqrt{r^2 - z^2}}\, dr \\
&= \int_{|z|}^{\infty} \frac{\exp\left(-\frac{r^2 - z^2}{2}\right) \exp\left(-\frac{z^2}{2}\right)}{2\pi} \frac{}{\sqrt{r^2 - z^2}}\, d(r^2 - z^2)
\end{aligned}$$

Let $t^2 = r^2 - z^2$, then

$$\begin{aligned}
&= \frac{\exp\left(-\frac{z^2}{2}\right)}{2\pi} \int_0^{\infty} \exp\left(-\frac{t^2}{2}\right) \cdot \frac{1}{t}\, dt^2 \\
&= \frac{\exp\left(-\frac{z^2}{2}\right)}{\pi} \int_0^{\infty} \exp\left(-\frac{t^2}{2}\right)\, dt
\end{aligned}$$

We know that

$$\int_0^{\infty} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{t^2}{2}\right) dt = \frac{1}{2} \text{ Due to the property of normal distribution.}$$

Therefore

$$f_{Z_2}(z) = f_{Z_1}(z) = \frac{\sqrt{2\pi}}{2\pi} \exp\left(-\frac{z^2}{2}\right) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{z^2}{2}\right)$$

This is exactly the probability density function of the standard normal distribution. Thus

$$Z_1 \sim \mathcal{N}(0,1) \quad \text{and} \quad Z_2 \sim \mathcal{N}(0,1) \qquad \square$$

ECE 685 Homework 1

Xinyuan Lyu

September 22, 2024

Problem 1

Let us define $a_1 = W_1^\top X + b_1$, and $a_2 = W_2^\top h_1 + b_2$;
Suppose $X \in \mathbb{R}^{m\times 1}, h_1 \in \mathbb{R}^{d_1\times 1}, h_2 \in \mathbb{R}^{d_2\times 1}$, that is, they are m, d_1, d_2 dims column vectors
Then

$$W_1 \in \mathbb{R}^{m\times d_1}, \quad b_1 \in \mathbb{R}^{d_1\times 1}$$
$$W_2 \in \mathbb{R}^{d_1\times d_2}, \quad b_2 \in \mathbb{R}^{d_2\times 1}$$
$$W_3 \in \mathbb{R}^{d_2\times 1}, \quad b_3 \in \mathbb{R}$$

Let $l_i = (y_i - \hat{y}_i)^2$, so for convenient, let $l = (y - \hat{y})^2$

Notice: Since all the gradient matrices below are written in Hessian form rather than Jacobian form, the chain rule should be applied from back to front.
For the b_1 and W_1:

$$\frac{\partial l}{\partial W_3} = \frac{\partial \hat{y}}{\partial W_3} \cdot \frac{\partial l}{\partial \hat{y}} = h_2 \cdot 2(\hat{y} - y) = 2(\hat{y} - y)h_2$$

$$\frac{\partial l}{\partial b_3} = \frac{\partial \hat{y}}{\partial b_3} \cdot \frac{\partial l}{\partial \hat{y}} = 1 \cdot 2(\hat{y} - y) = 2(\hat{y} - y)$$

Then for b_2 and W_2:

$$\frac{\partial l}{\partial b_2} = \frac{\partial a_2}{\partial b_2} \cdot \frac{\partial h_2}{\partial a_2} \cdot \frac{\partial \hat{y}}{\partial h_2} \cdot \frac{\partial l}{\partial \hat{y}}$$

$$\frac{\partial a_2}{\partial b_2} = \frac{\partial b_2}{\partial b_2} = \mathbf{I}_{d_2\times d_2}; \quad \frac{\partial h_2}{\partial a_2} = \mathrm{diag}\{g'(a_2)_1, \ldots, g'(a_2)_{d_2}\}$$

$$\frac{\partial \hat{y}}{\partial h_2} = (W_3^\top)^\top = W_3$$

$$\frac{\partial l}{\partial b_2} = \mathbf{I}_{d_2 \times d_2} \cdot \text{diag}\{g'(a_2)_1, \ldots, g'(a_2)_{d_2}\} \cdot W_3 \cdot 2(\hat{y} - y)$$
$$= 2(\hat{y} - y)(g'(a_2) \odot W_3)$$

For W_2 we need more process since it is hard to derive the gradient of matrix for vector. To avoid introducing high dim tensor, we write the W_2 as following: Let

$$W_2 = \begin{pmatrix} W_{21}^\top \\ \vdots \\ W_{2d_1}^\top \end{pmatrix} \text{ where } W_{2i} = \begin{pmatrix} w_{i1} \\ \vdots \\ w_{id_2} \end{pmatrix}$$

So

$$\frac{\partial l}{\partial W_{2i}} = \frac{\partial a_2}{\partial W_{2i}} \cdot \frac{\partial h_2}{\partial a_2} \cdot \frac{\partial \hat{y}}{\partial h_2} \cdot \frac{\partial l}{\partial \hat{y}}$$

And we have

$$\frac{\partial a_2}{\partial W_{2i}} = \frac{\partial (h_{11} W_{21} + \cdots + h_{1d_1} W_{2d_1} + b_2)}{\partial W_{2i}} = h_{1i} \cdot \mathbf{I}_{d_2 \times d_2}$$

So

$$\frac{\partial l}{\partial W_{2i}} = h_{1i} \cdot \mathbf{I}_{d_2 \times d_2} \cdot \text{diag}\{g'(a_2)_1, \ldots, g'(a_2)_{d_2}\} \cdot W_3 \cdot 2(\hat{y} - y)$$
$$= 2(\hat{y} - y) \cdot h_{1i} \cdot (g'(a_2) \odot W_3)$$

So combine the rows together, we have:

$$\frac{\partial l}{\partial W_2} = 2(\hat{y} - y) \begin{bmatrix} h_{11}(g'(a_2) \odot W_3)^\top \\ \vdots \\ h_{1d_1}(g'(a_2) \odot W_3)^\top \end{bmatrix}$$

$$= 2(\hat{y} - y) \cdot h_1 \cdot (g'(a_2) \odot W_3)^\top$$

Then for b_1 and W_1:

$$\frac{\partial l}{\partial b_1} = \frac{\partial a_1}{\partial b_1} \cdot \frac{\partial h_1}{\partial a_1} \cdot \frac{\partial a_2}{\partial h_1} \cdot \frac{\partial h_2}{\partial a_2} \cdot \frac{\partial \hat{y}}{\partial h_2} \cdot \frac{\partial l}{\partial \hat{y}}$$

Similarly as for b_2 and W_2, we have:

$$\frac{\partial a_1}{\partial b_1} = \mathbf{I}_{d_1 \times d_1}, \quad \frac{\partial h_1}{\partial a_1} = \text{diag}\{g'(a_1)_1, \ldots, g'(a_1)_{d_1}\}$$

$$\frac{\partial a_2}{\partial h_1} = (W_2^\top)^\top = W_2; \quad \frac{\partial h_2}{\partial a_2} = \text{diag}\{g'(a_2)_1, \ldots, g'(a_2)_{d_2}\}; \quad \frac{\partial \hat{y}}{\partial h_2} = W_3$$

So we have:

$$\frac{\partial l}{\partial b_1} = \mathbf{I}_{d_1 \times d_1} \cdot \text{diag}\{g'(a_1)_1, \ldots, g'(a_1)_{d_1}\} \cdot W_2 \cdot \text{diag}\{g'(a_2)_1, \ldots, g'(a_2)_{d_2}\} \cdot W_3 \cdot 2(\hat{y} - y)$$

$$= 2(\hat{y} - y)(g'(a_1) \odot [W_2(g'(a_2) \odot W_3)])$$

Similarly, Let

$$W_1 = \begin{pmatrix} W_{11}^\top \\ \vdots \\ W_{1d_1}^\top \end{pmatrix} \text{ where } W_{1i} = \begin{pmatrix} w_{i1} \\ \vdots \\ w_{im} \end{pmatrix}$$

So we have:

$$\frac{\partial l}{\partial W_{1i}} = \frac{\partial a_1}{\partial W_{1i}} \cdot \frac{\partial h_1}{\partial a_1} \cdot \frac{\partial a_2}{\partial h_1} \cdot \frac{\partial h_2}{\partial a_2} \cdot \frac{\partial \hat{y}}{\partial h_2} \cdot \frac{\partial l}{\partial \hat{y}}$$

$$= x_i \cdot \mathbf{I}_{d_1 \times d_1} \cdot \text{diag}\{g'(a_1), \cdots, g'(a_{1d_1})\} \cdot W_2 \cdot \text{diag}\{g'(a_2), \cdots, g'(a_{2d_2})\} \cdot W_3 \cdot 2(\hat{y} - y)$$

$$= 2(\hat{y} - y) \cdot x_i \cdot \{g'(a_1) \odot [W_2(g'(a_2) \odot W_3)]\}$$

So combine together, we have:

$$\frac{\partial l}{\partial W_1} = 2(\hat{y} - y) \begin{bmatrix} x_1 \cdot \{g'(a_1) \odot [W_2(g'(a_2) \odot W_3)]\}^\top \\ \vdots \\ x_m \cdot \{g'(a_1) \odot [W_2(g'(a_2) \odot W_3)]\}^\top \end{bmatrix}$$

$$= 2(\hat{y} - y)x\{g'(a_1) \odot [W_2(g'(a_2) \odot W_3)]\}^\top$$

So finally we have the following result. And to avoid confusion, we consistently place the sample index i on the outermost level.

$$\frac{\partial L}{\partial b_3} = \frac{1}{N} \sum_{i=1}^{N} \{2(\hat{y} - y)\}_i, \quad \frac{\partial L}{\partial W_3} = \frac{1}{N} \sum_{i=1}^{N} \{2(\hat{y} - y)h_2\}_i$$

$$\frac{\partial L}{\partial b_2} = \frac{1}{N} \sum_{i=1}^{N} \{2(\hat{y} - y) \cdot (g'(a_2) \odot W_3)\}_i$$

$$\frac{\partial L}{\partial W_2} = \frac{1}{N} \sum_{i=1}^{N} \{2(\hat{y} - y) \cdot h_1 \cdot (g'(a_2) \odot W_3)^\top\}_i$$

$$\frac{\partial L}{\partial b_1} = \frac{1}{N} \sum_{i=1}^{N} \Big\{2(\hat{y} - y) \cdot \{g'(a_1) \odot [W_2 \cdot (g'(a_2) \odot W_3)]\}\Big\}_i$$

$$\frac{\partial L}{\partial W_1} = \frac{1}{N} \sum_{i=1}^{N} \Big\{2(\hat{y} - y) \cdot x \cdot \{g'(a_1) \odot [W_2(g'(a_2) \odot W_3)]\}^\top\Big\}_i$$

Problem 2

Part 1

In [1]:
```
import torch
import torch.nn as nn
import torch.optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

Define the MNISTdata class.

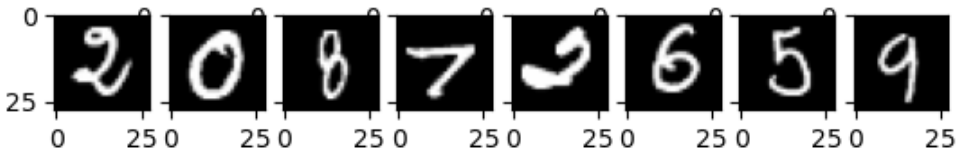
In [2]:
```
class MNISTdata(torch.utils.data.Dataset):

    def __init__(self, root, transform=None):
        self.data = pd.read_csv(root) #get the raw data
        self.transform = transform
        self.X = self.data.iloc[:, 1:].values.astype(np.uint8) #split the data to x and y
        self.Y = self.data.iloc[:, 0].values

    def __getitem__(self, index):
        data = self.X[index].reshape(28, 28) #reshape the data to 28, 28
        label = self.Y[index]

        if self.transform:
            data = self.transform(data)

        return data, label

    def __len__(self):
        return len(self.X)
```

Read and load the data.

In [3]:
```
trainset=MNISTdata("C:/Users/14857/Desktop/ECE685/homework/HW2/code/archive/mnist_train.csv",transform = t
testset=MNISTdata("C:/Users/14857/Desktop/ECE685/homework/HW2/code/archive/mnist_test.csv",transform = tra
trainloader = torch.utils.data.DataLoader(trainset, batch_size=8, shuffle=True)
testloader = torch.utils.data.DataLoader(testset, batch_size=8, shuffle=False)
```

Draw the picture.

In [4]:
```
dataiter = iter(trainloader)
images, labels = next(dataiter)
im = images[0:8]
fig , ax = plt.subplots(1,8)
for i in range(8):
    ax[i].imshow(im[i][0], cmap='gray')
plt.show()
print(f'Labels:{labels[0:8]}')
```




```
Labels:tensor([2, 0, 8, 7, 2, 6, 5, 9])
```

Part 2

In [5]:
```
import os
from PIL import Image
import albumentations as A
from albumentations.pytorch import ToTensorV2
```

```
d:\anaconda\envs\d2l\lib\site-packages\albumentations\__init__.py:13: UserWarning: A new version of Albumen
tations is available: 1.4.16 (you have 1.4.14). Upgrade using: pip install -U albumentations. To disable au
tomatic update checks, set the environment variable NO_ALBUMENTATIONS_UPDATE to 1.
  check_for_updates()
```

Define the mapdata class.

In [6]:
```
class mapdata(torch.utils.data.Dataset):

    def __init__(self, image_root, mask_root, transform=None):
        self.image_root = image_root
        self.mask_root = mask_root
        self.transform = transform
        self.image_list = sorted(os.listdir(image_root)) # since all the pictures are in a file, so we nee
        self.mask_list = sorted(os.listdir(mask_root))

    def __getitem__(self, index):

        image_name = os.path.join(self.image_root, self.image_list[index]) # get the absolute path of imag
        mask_name = os.path.join(self.mask_root, self.mask_list[index])

        image = np.array(Image.open(image_name).convert("RGB")) # open the image to np arrays and all the
        mask = np.array(Image.open(mask_name).convert("RGB"))

        if self.transform:
            augment_data = self.transform(image=image, mask=mask) # do the augment for images and masks.
            image = augment_data['image']
            mask = augment_data['mask']

        return image, mask

    def __len__(self):
        return len(self.image_list)
```

Define the tranform, which will augment the pictures.

In [7]:
```
transform = A.Compose([
    A.RandomResizedCrop(height=256, width=256, scale=(0.8, 1.0)),
    A.HorizontalFlip(p=0.5),
    A.VerticalFlip(p=0.5),
    A.RandomRotate90(p=0.5),
    A.Affine(scale=(0.5, 1.5), translate_percent=(0.3, 0.3), rotate=(-20, 20), shear=(-30, 30), p=0.5),
    A.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5)),
    A.pytorch.ToTensorV2()
])
```

Read and Load the data.

In [8]:
```
the_data_set = mapdata('C:/Users/14857/Desktop/ECE685/homework/HW2/code/archive/images', 'C:/Users/14857/D
the_data_loader = torch.utils.data.DataLoader(the_data_set, batch_size=4, shuffle=True)

image, mask = the_data_set[10]
print(image.shape)
print(mask.shape)
```

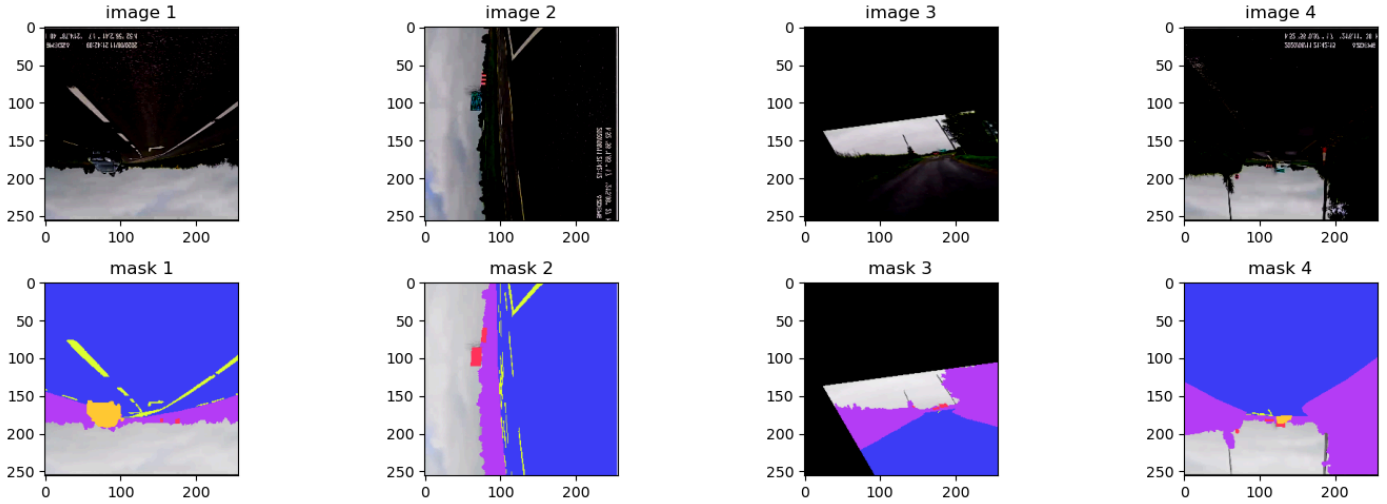
```
torch.Size([3, 256, 256])
torch.Size([256, 256, 3])
```

Draw the Pictures.

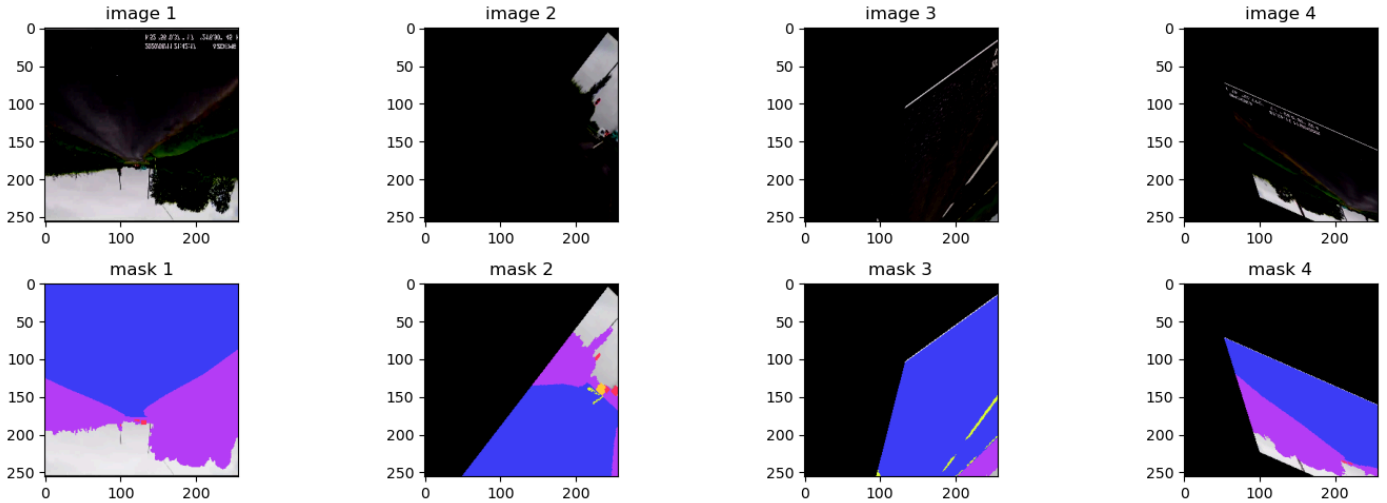
In [9]:
```
dataiter = iter(the_data_loader)

for turn in range(5):
```

```
images, masks = next(dataiter)
fig, ax = plt.subplots(2, 4, figsize=(15, 5))

for i in range(4):

    ax[0, i].imshow(images[i].permute(1, 2, 0))
    ax[0, i].set_title(f"image {i+1}")

    ax[1, i].imshow(masks[i])
    ax[1, i].set_title(f"mask {i+1}")

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

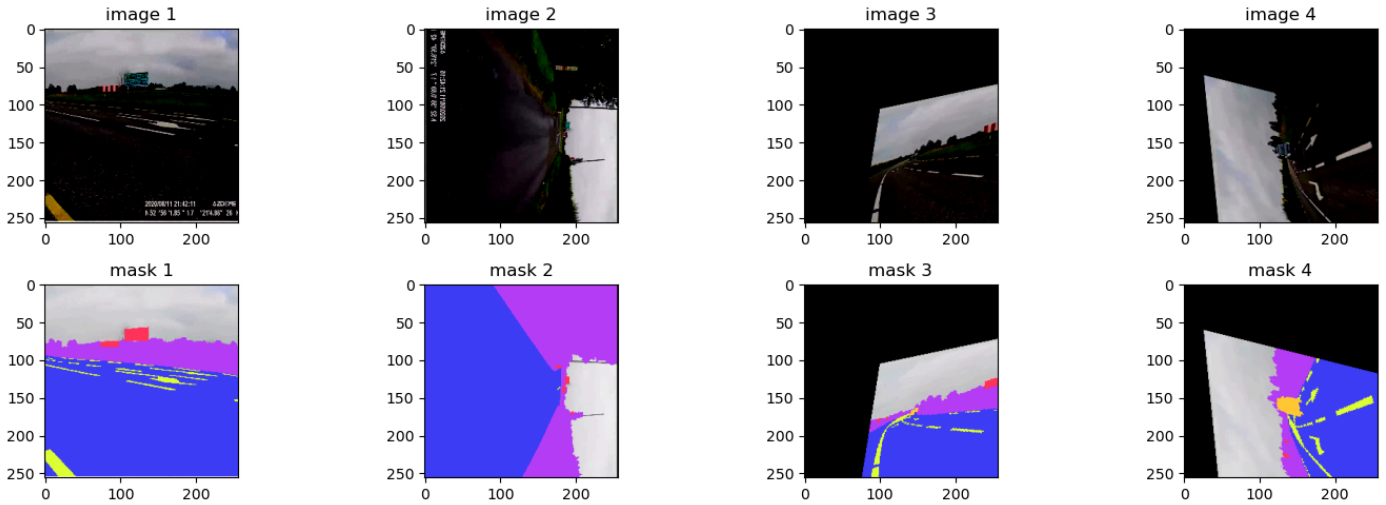
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s). Got range [-1.0..1.0].
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s). Got range [-1.0..1.0].
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s). Got range [-1.0..0.9058824].
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s). Got range [-1.0..1.0].
```



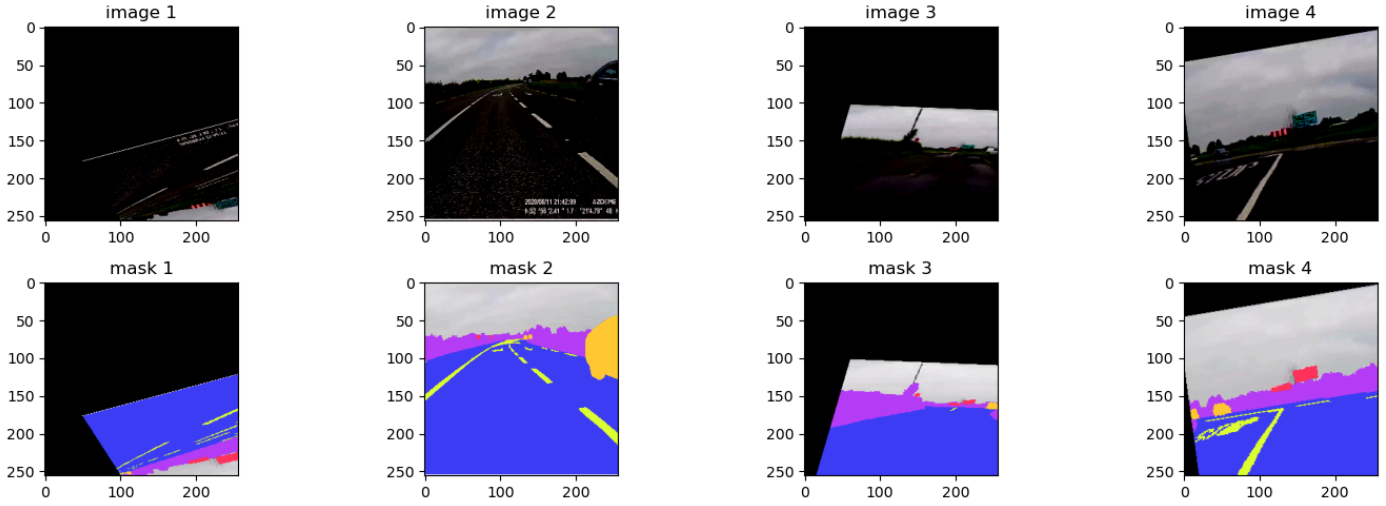

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s). Got range [-1.0..1.0].
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s). Got range [-1.0..0.8117648].
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s). Got range [-1.0..1.0].
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s). Got range [-1.0..1.0].
```



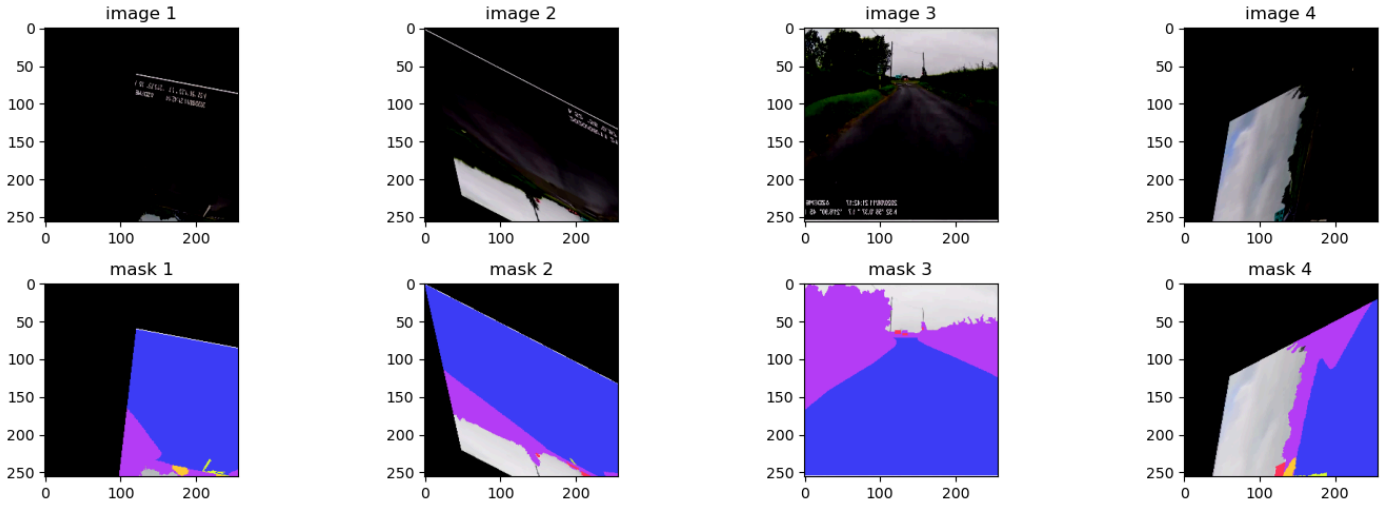
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s). Got range [-1.0..1.0].
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s). Got range [-1.0..1.0].
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s). Got range [-1.0..0.93725497].
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s). Got range [-1.0..0.78823537].
```




```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s). Got range [-1.0..1.0].
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s). Got range [-1.0..1.0].
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s). Got range [-1.0..0.91372555].
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s). Got range [-1.0..0.9686275].
```




```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s). Got range [-1.0..1.0].
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s). Got range [-1.0..1.0].
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s). Got range [-1.0..1.0].
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s). Got range [-1.0..0.8352942].
```



In [10]:
```
%reset -f
```

Problem 3

In [11]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.autograd as autograd

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

Prepare the dataset.

In [12]:
```
root_data = load_breast_cancer()
X_root = root_data.data
y_root = root_data.target

X_train_root, X_test_root, y_train_root, y_test_root = train_test_split(X_root, y_root, test_size=0.3, ran

scaler = StandardScaler()
X_train_root = scaler.fit_transform(X_train_root)
X_test_root = scaler.fit_transform(X_test_root)

X_train_root = np.hstack([np.ones((X_train_root.shape[0], 1)), X_train_root])  # add the constant term.
X_test_root = np.hstack([np.ones((X_test_root.shape[0], 1)), X_test_root])  # add the constant term.

X_train_root = torch.from_numpy(X_train_root).float()
y_train_root = torch.from_numpy(y_train_root).float()
X_test_root = torch.from_numpy(X_test_root).float()
y_test_root = torch.from_numpy(y_test_root).float()

print(X_train_root.shape)
print(X_train_root.shape)
print(X_train_root[0])
```

```
torch.Size([398, 31])
torch.Size([398, 31])
tensor([ 1.0000, -0.5206, -0.3630, -0.4655, -0.5522,  0.6360,  0.4767, -0.1807,
        -0.5793,  0.0030,  1.1679, -0.8164,  0.4358, -0.3619, -0.5394,  0.8984,
         0.3856,  0.6826, -0.0638,  0.1989,  0.8297, -0.7349, -0.4486, -0.5643,
        -0.6622,  0.5583,  0.0179, -0.1864, -0.6758, -0.5811,  0.5192])
```

Model Specification.

In [13]:
```
# define the logistic regression function.
def logistic_reg(X, W):
    a = torch.matmul(X, W)
    a = torch.clamp(a, min=-88, max=88)
    return 1 / (1 + torch.exp(-(a)))

# define the bce loss function.
def BCELoss(W, X, y):
    a = torch.matmul(X, W)
    a = torch.clamp(a, min=-88, max=88) # avoid |a| is too large that the exp will fail.
    prediction = 1 / (1 + torch.exp(-(a)))
    prediction = torch.clamp(prediction, min=1e-4, max=1 - 1e-4) # avoid prediction is too close to 0 or 1
    average_loss = -torch.mean(y * torch.log(prediction) + (1 - y) * torch.log(1 - prediction))
    return average_loss

# define the auto-computing gradient function.
def gradient_function(W, X, y):
    return autograd.grad(BCELoss(W, X, y), W, create_graph=True)[0]

# define the computing hessian matrix function.
def hessian_W(W, X, y):
    grad = gradient_function(W, X, y)
    n = W.shape[0]
    H = torch.zeros((n, n))

    for i in range(n):
        grad2 = torch.autograd.grad(grad[i], W, create_graph=True)[0]
        H[i] = grad2.view(-1)

    return H

# define the computing diagonal approximate hessian matrix function.
def approximate_hessian_W(W, X, y):
    grad = gradient_function(W, X, y)
    n = W.shape[0]
    H = torch.zeros((n, n))

    for i in range(n):
        grad2 = torch.autograd.grad(grad[i], W, create_graph=True)[0]
        H[i][i] = grad2.view(-1)[i].item()

    return H
```

Train the Models.

Exact expression for the Hessian

In [14]:
```
learn_rate = 0.1 # set the learning rate.
num_epochs = 50 # set the number of epochs.
num_inputs=31 # the feature is 30+1 which contains the constant term.

# initialize the W, since this is logistic regression so for sigmoid function, W=0 is suitable.
W = torch.zeros(num_inputs, requires_grad=True)
print(W.shape)

# Prepare for the plotting later.
train_losses_1 = []
test_losses_1 = []
train_accuracies_1 = []
test_accuracies_1 = []
```

```
torch.Size([31])
```

In [15]:
```
for epoch in range(num_epochs):

    correct_train = 0
    total_train = 0

    # get the data.
    X = X_train_root
```

```
    y = y_train_root

    # compute the prediction and loss in this turn; and get the accuracy.
    y_hat = logistic_reg(X, W)
    y_hat = torch.round(y_hat)
    ls_train = BCELoss(W, X, y)

    correct_train += (y_hat == y).sum().item()
    total_train = y.size(0)

    train_losses_1.append(ls_train.item())
    train_accuracy = correct_train / total_train
    train_accuracies_1.append(train_accuracy)


    # do the same computing for the prediction and loss in this turn; and get the accuracy, in the test da
    with torch.no_grad():

        total_test_loss = 0
        correct_test = 0
        total_test = 0

        X_test = X_test_root
        y_test = y_test_root

        y_hat_test = logistic_reg(X_test,W)
        y_hat_test = torch.round(y_hat_test)
        test_loss = BCELoss(W, X, y)

        correct_test += (y_hat_test == y_test).sum().item()
        total_test += y_test.size(0)

        test_losses_1.append(test_loss.item())
        test_accuracy = correct_test / total_test
        test_accuracies_1.append(test_accuracy)


    #Compute the gradient, Hessian and inverse of Hessian.
    Gradient=gradient_function(W, X, y)
    Hessian=hessian_W(W, X, y)
    Hessian_inverse = torch.inverse(Hessian)
    #Save the data
    previous_W = W.clone()
    previous_Gradient = Gradient.clone()
    previous_Hessian = Hessian.clone()
    previous_Hessian_inverse = Hessian_inverse.clone()

    # Renew the Weight.
    W = W - learn_rate*torch.matmul(Hessian_inverse, Gradient)

    if torch.all(W == 0) or torch.isnan(W).any():
        print("Error: W is all 0 or NaN")
        print("Previous W:", previous_W)
        print("Previous Gradient:", previous_Gradient)
        print("Previous Hessian:", previous_Hessian)
        print("Previous Hessian_inverse:", previous_Hessian_inverse)
        break
```

Diagonal approximation for Hessian

In [16]:

```
learn_rate = 0.1 # set the learning rate.
num_epochs = 50
num_inputs=31

W = torch.zeros(num_inputs, requires_grad=True)
print(W.shape)

train_losses_2 = [] # Prepare for the plotting later.
test_losses_2 = []
train_accuracies_2 = []
test_accuracies_2 = []
```

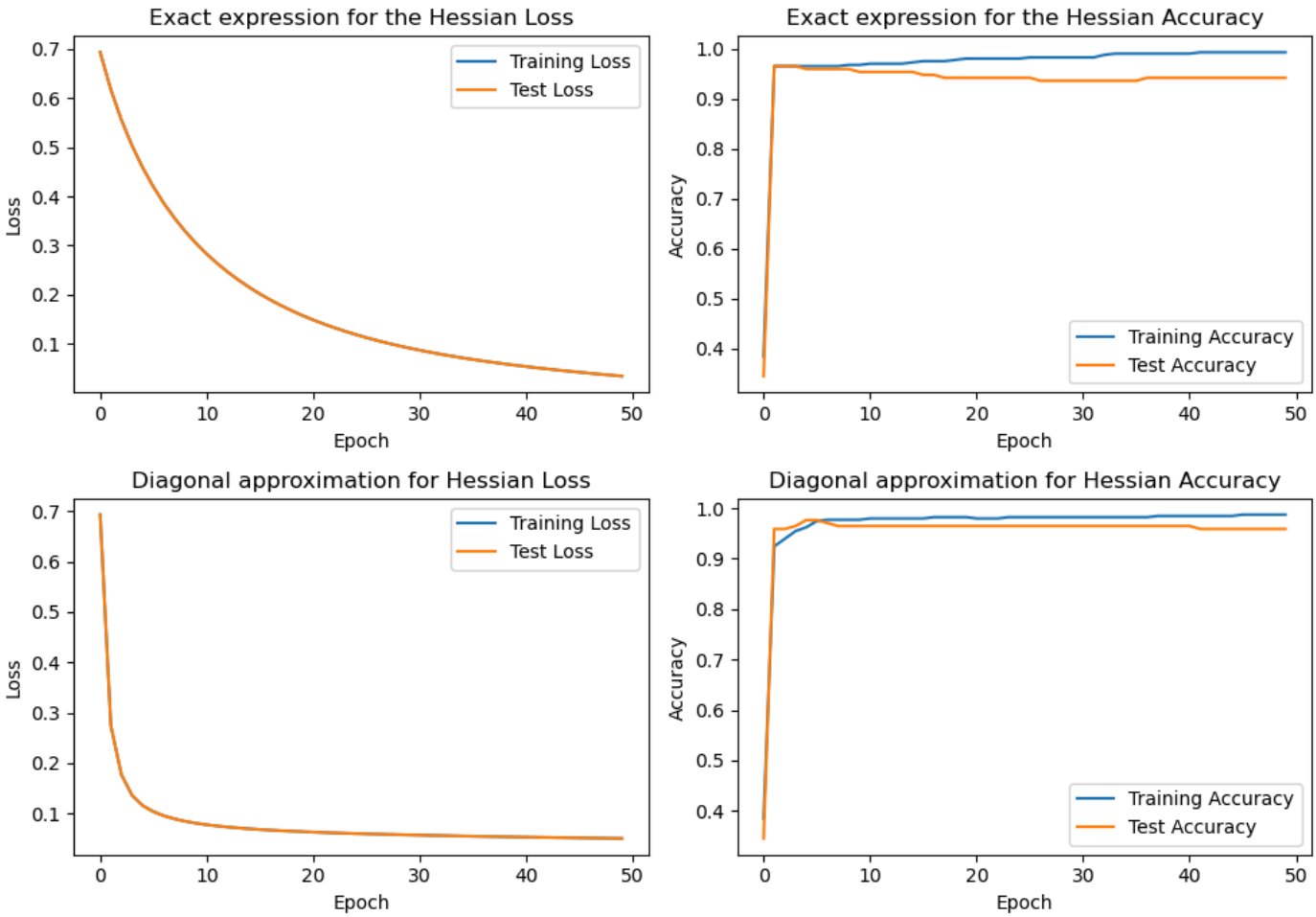
```
torch.Size([31])
```

In [17]:
```
for epoch in range(num_epochs):

    correct_train = 0
    total_train = 0

    # get the data.
    X = X_train_root
    y = y_train_root

    # compute the prediction and loss in this turn; and get the accuracy.
    y_hat = logistic_reg(X, W)
    y_hat = torch.round(y_hat)
    ls_train = BCELoss(W, X, y)

    correct_train += (y_hat == y).sum().item()
    total_train = y.size(0)

    train_losses_2.append(ls_train.item())
    train_accuracy = correct_train / total_train
    train_accuracies_2.append(train_accuracy)

    # do the same computing for the prediction and loss in this turn; and get the accuracy, in the test da
    with torch.no_grad():

        total_test_loss = 0
        correct_test = 0
        total_test = 0

        X_test = X_test_root
        y_test = y_test_root

        y_hat_test = logistic_reg(X_test,W)
        y_hat_test = torch.round(y_hat_test)
        test_loss = BCELoss(W, X, y)

        correct_test += (y_hat_test == y_test).sum().item()
        total_test += y_test.size(0)

        test_losses_2.append(test_loss.item())
        test_accuracy = correct_test / total_test
        test_accuracies_2.append(test_accuracy)

    #Compute the gradient, Hessian and inverse of Hessian.
    Gradient=gradient_function(W, X, y)
    Hessian=approximate_hessian_W(W, X, y)
    diag = torch.diagonal(Hessian)
    inv_diag = 1.0 / diag
    Hessian_inverse = torch.diag(inv_diag)

    # Rewnew the wights.
    W = W - learn_rate*torch.matmul(Hessian_inverse, Gradient)
```

Plot and Compare

In [18]:
```
def plot_the_model(ax, y1, y2, plot_type='loss'):
    if plot_type == 'loss':
        ax.plot(y1, label='Training Loss')
        ax.plot(y2, label='Test Loss')
        ax.set_ylabel('Loss')
    elif plot_type == 'accuracy':
        ax.plot(y1, label='Training Accuracy')
        ax.plot(y2, label='Test Accuracy')
        ax.set_ylabel('Accuracy')
    ax.set_xlabel('Epoch')
    ax.legend()

fig, axes = plt.subplots(2, 2, figsize=(10, 7))

plot_the_model(axes[0, 0], train_losses_1, test_losses_1, plot_type='loss')
```

```
axes[0, 0].set_title('Exact expression for the Hessian Loss')
plot_the_model(axes[0, 1], train_accuracies_1, test_accuracies_1, plot_type='accuracy')
axes[0, 1].set_title('Exact expression for the Hessian Accuracy')

plot_the_model(axes[1, 0], train_losses_2, test_losses_2, plot_type='loss')
axes[1, 0].set_title('Diagonal approximation for Hessian Loss')
plot_the_model(axes[1, 1], train_accuracies_2, test_accuracies_2, plot_type='accuracy')
axes[1, 1].set_title('Diagonal approximation for Hessian Accuracy')

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



Problem 1: Object Detection with Convolutional Neural Networks (CNNs)

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim
import torchvision
import torchvision.transforms as transforms
import torch.optim as optim
from torch.utils.data import Dataset
import torchvision.transforms as T
import matplotlib.pyplot as plt
import numpy as np
import math
import os
import pandas as pd
from PIL import Image
import albumentations as A
from albumentations.pytorch import ToTensorV2
```

```
d:\anaconda\envs\d2l\lib\site-packages\albumentations\__init__.py:13: UserWarning: A new version of Albumentations is available: 1.4.18 (you h
ave 1.4.14). Upgrade using: pip install -U albumentations. To disable automatic update checks, set the environment variable NO_ALBUMENTATIONS_
UPDATE to 1.
  check_for_updates()
```

1.1 Data Preprocessing

Firstly, define the face data class.

In [2]:
```
class facedata(Dataset):
    def __init__(self, image_root, box_root, transform=None):
        self.boxes = pd.read_csv(box_root) # The table of the box data.
        self.image_root = image_root
        self.transform = transform

    def __getitem__(self, index):
        image_name = os.path.join(self.image_root, self.boxes.iloc[index, 0])
        image = Image.open(image_name).convert("RGB")

        x_min = self.boxes.iloc[index, 3]
        y_min = self.boxes.iloc[index, 4]
        x_max = self.boxes.iloc[index, 5]
        y_max = self.boxes.iloc[index, 6]
        box = [x_min, y_min, x_max, y_max]

        if self.transform:
            image = np.array(image)
            transformed = self.transform(image=image, bboxes=[box])
            image = transformed['image']
            box = transformed['bboxes'][0]

        return image, torch.tensor(box, dtype=torch.float32)

    def __len__(self):
        return len(self.boxes)
```

Secondly, define the spatial transformations.

In [3]:
```
transformations = A.Compose([
    A.Resize(224, 224),
    A.HorizontalFlip(p=0.5),
    A.VerticalFlip(p=0.5),
    A.RandomRotate90(p=0.5),
    A.ElasticTransform(alpha=120, sigma=120 * 0.05, alpha_affine=None, p=0.5),
    A.pytorch.ToTensorV2()
], bbox_params=A.BboxParams(format='pascal_voc', label_fields=[]))

# For test data, we do not need augmentation but should resize it for later model.
test_transformations = A.Compose([
    A.Resize(224, 224),
    A.pytorch.ToTensorV2()
], bbox_params=A.BboxParams(format='pascal_voc', label_fields=[]))
```

Thirdly, divide the data into train and test, and drop the duplicates since our model just detect single target.

In [4]:
```
from sklearn.model_selection import train_test_split

boxes = pd.read_csv('C:/Users/14857/Desktop/ECE685/homework/HW3/archive/faces.csv')
print(len(boxes))
boxes = boxes.groupby('image_name').apply(lambda x: x.sample(1)).reset_index(drop=True)
print(len(boxes))
```

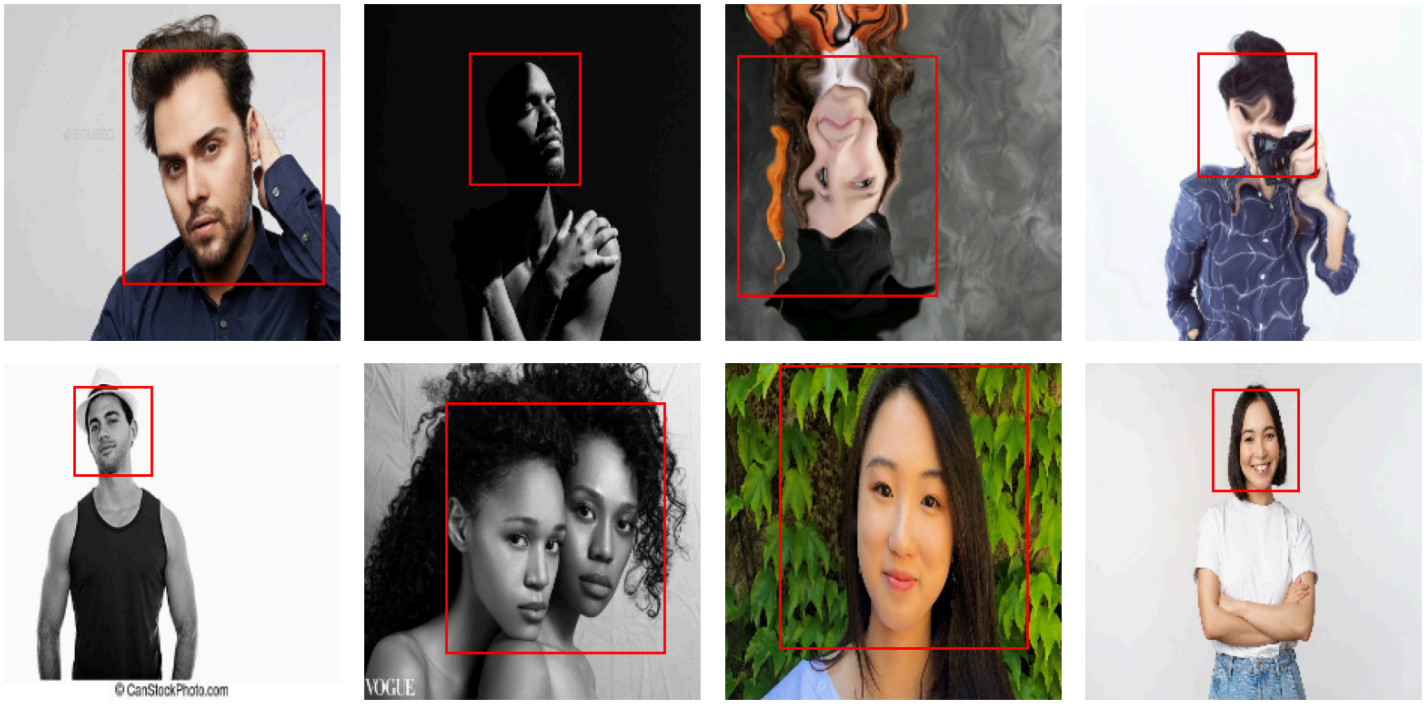
```
train_boxes, test_boxes = train_test_split(boxes, test_size=0.2, random_state=10)
train_boxes.to_csv('C:/Users/14857/Desktop/ECE685/homework/HW3/archive/train_boxes.csv', index=False)
test_boxes.to_csv('C:/Users/14857/Desktop/ECE685/homework/HW3/archive/test_boxes.csv', index=False)

train_dataset = facedata('C:/Users/14857/Desktop/ECE685/homework/HW3/archive/images', 'C:/Users/14857/Desktop/ECE685/homework/HW3/archive/tra
test_dataset = facedata('C:/Users/14857/Desktop/ECE685/homework/HW3/archive/images', 'C:/Users/14857/Desktop/ECE685/homework/HW3/archive/test
```

```
3350
2204
```

```
C:\Users\14857\AppData\Local\Temp\ipykernel_22864\2344910187.py:5: DeprecationWarning: DataFrameGroupBy.apply operated on the grouping column
s. This behavior is deprecated, and in a future version of pandas the grouping columns will be excluded from the operation. Either pass `inclu
de_groups=False` to exclude the groupings or explicitly select the grouping columns after groupby to silence this warning.
  boxes = boxes.groupby('image_name').apply(lambda x: x.sample(1)).reset_index(drop=True)
```

Fourthly, draw the pictures.

In [5]:
```
def draw_sample(dataset, indices):

    fig, axes = plt.subplots(1, len(indices), figsize=(16, 4))

    for i, idx in enumerate(indices):
        sample = dataset[idx]
        image = sample[0].permute(1, 2, 0)
        bbox = sample[1]

        axes[i].imshow(image)

        x_min, y_min, x_max, y_max = bbox
        rect = plt.Rectangle((x_min, y_min), x_max - x_min, y_max - y_min,
                             linewidth=2, edgecolor='r', facecolor='none')
        axes[i].add_patch(rect)

        axes[i].axis('off')

    plt.tight_layout()
    plt.show()

draw_sample(train_dataset, indices=[0, 1, 2, 3]) # The train data sets.
draw_sample(test_dataset, indices=[0, 1, 2, 3]) # The test data sets.
```



1.2 Object Detection with Pre-trained Feature Extractor

In [6]:
```
import torchvision.models as models
from torchvision.ops import generalized_box_iou_loss
from torchvision.ops import box_iou as IoU
```

Firstly, Define the model based on backbone, we use resnet50 as backbone here.

In [7]:
```
class BoundingBoxRegressor(nn.Module):
    def __init__(self, freeze_backbone=True):
        super(BoundingBoxRegressor, self).__init__()

        self.normalize = T.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225))

        self.feature_extractor = models.resnet50(weights='IMAGENET1K_V1').to(device)
        in_features = self.feature_extractor.fc.in_features
        self.feature_extractor.fc = nn.Identity()

        if freeze_backbone:
            for param in self.feature_extractor.parameters():
```

```
                param.requires_grad = False
        else:
            for param in self.feature_extractor.parameters():
                param.requires_grad = True

        self.regressor = nn.Sequential(
            nn.Linear(in_features, 512),
            nn.Linear(512, 206),
            nn.Linear(206, 103),
            nn.Linear(103, 4)
        )

        for param in self.regressor.parameters():
            param.requires_grad = True

    def forward(self, x):
        x = self.normalize(x)
        features = self.feature_extractor(x)
        bbox_pred = self.regressor(features)

        return bbox_pred
```

Secondly, Define the train process. (Rewrite from the trainer present in discussion section)

In [8]:
```
def train(model, device, train_loader, batch_size, criterion, optimizer, epoch, train_losses, train_IoUs):
    model.train()

    train_loss = 0
    train_IoU = 0
    batch_index = 0

    for batch_idx, (data, target) in enumerate(train_loader):
        batch_index = batch_idx
        data, target = data.to(device), target.to(device)
        data, target = data.float(), target.float()

        if len(data) != batch_size:
            continue

        optimizer.zero_grad()

        output = model(data)
        loss = criterion(output, target)

        loss.backward()
        optimizer.step()

        train_loss += loss.item()
        for i in range(batch_size):
                pred_box = output[i]
                true_box = target[i]
                pred_box = output[i].unsqueeze(0)
                true_box = target[i].unsqueeze(0)
                iou = IoU(pred_box, true_box).item()
                train_IoU += iou

        if batch_idx % (len(train_loader)//2) == 0:
            print('Train({})[{:.0f}%]: Loss: {:.4f}, IoU: {:.4f}%'.format(
                epoch, 100. * batch_idx / len(train_loader), train_loss/(batch_idx+1), 100. * train_IoU/(batch_size*(batch_idx+1))))

    train_losses.append(train_loss/(batch_index+1))
    train_IoUs.append(train_IoU/(batch_size*(batch_index+1)))


def test(model, device, test_loader, batch_size, criterion, epoch, test_losses, test_IoUs):
    model.eval()
    test_loss = 0
    test_IoU = 0

    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            data, target = data.float(), target.float()

            if len(data) != batch_size:
                continue

            output = model(data)

            test_loss += criterion(output, target).item()
            for i in range(batch_size):
                pred_box = output[i]
                true_box = target[i]
                pred_box = output[i].unsqueeze(0)
                true_box = target[i].unsqueeze(0)
                iou = IoU(pred_box, true_box).item()
                test_IoU += iou

    test_loss = (test_loss*batch_size) / len(test_loader.dataset)
    test_IoU = test_IoU / len(test_loader.dataset)
    test_losses.append(test_loss)
    test_IoUs.append(test_IoU)
```

```
    print('Test({}): Loss: {:.4f}, IoU: {:.4f}%'.format(
        epoch, test_loss, 100. * test_IoU))
```

Thirdly, Train the freezed model.

In [9]:
```
device = 'cuda'
device = torch.device(device)
```

In [10]:
```
model_freeze = BoundingBoxRegressor().to(device)
criterion = nn.SmoothL1Loss()
trained_params = filter(lambda p: p.requires_grad, model_freeze.parameters())
optimizer = torch.optim.Adam(trained_params, lr=0.0001)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.5)
```

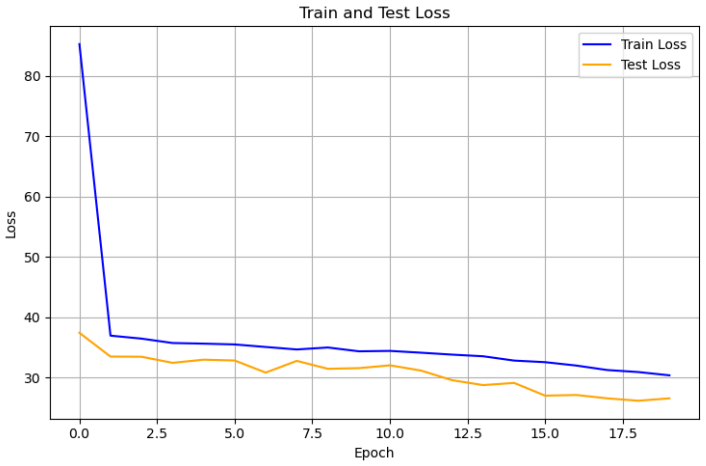
In [11]:
```
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32, shuffle=False)
```

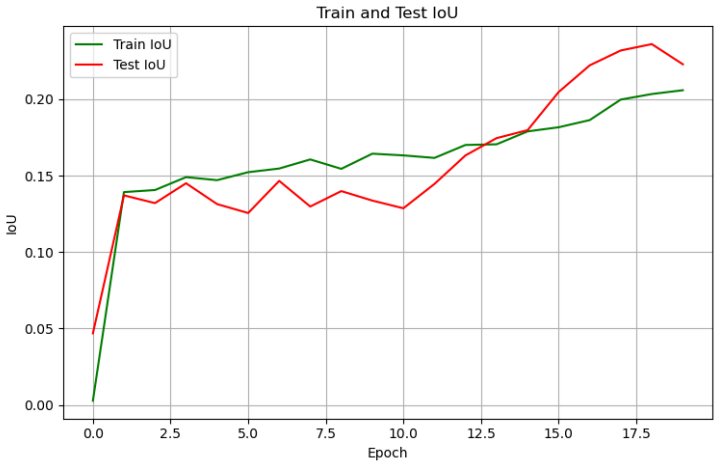
In [12]:
```
num_epochs = 20
batch_size = 32
train_losses_freeze = []
test_losses_freeze = []
train_IoUs_freeze = []
test_IoUs_freeze = []

for epoch in range(1, num_epochs + 1):
    train(model_freeze, device, train_loader, batch_size, criterion, optimizer, epoch, train_losses_freeze, train_IoUs_freeze)
    test(model_freeze, device, test_loader, batch_size, criterion, epoch, test_losses_freeze, test_IoUs_freeze)
```

```
Train(1)[0%]: Loss: 107.1134, IoU: 0.0000%
Train(1)[50%]: Loss: 105.7426, IoU: 0.0000%
Test(1): Loss: 37.4105, IoU: 4.6857%
Train(2)[0%]: Loss: 44.5937, IoU: 2.5791%
Train(2)[50%]: Loss: 37.9039, IoU: 13.4521%
Test(2): Loss: 33.4721, IoU: 13.7142%
Train(3)[0%]: Loss: 35.0977, IoU: 13.5766%
Train(3)[50%]: Loss: 36.9243, IoU: 13.8204%
Test(3): Loss: 33.4420, IoU: 13.2076%
Train(4)[0%]: Loss: 30.6216, IoU: 20.0096%
Train(4)[50%]: Loss: 35.9673, IoU: 15.1425%
Test(4): Loss: 32.4320, IoU: 14.5082%
Train(5)[0%]: Loss: 37.3387, IoU: 11.9081%
Train(5)[50%]: Loss: 36.6842, IoU: 14.6729%
Test(5): Loss: 32.9540, IoU: 13.1431%
Train(6)[0%]: Loss: 37.8008, IoU: 13.6779%
Train(6)[50%]: Loss: 36.3489, IoU: 15.7000%
Test(6): Loss: 32.8169, IoU: 12.5595%
Train(7)[0%]: Loss: 40.1048, IoU: 9.3592%
Train(7)[50%]: Loss: 35.9128, IoU: 15.5333%
Test(7): Loss: 30.8164, IoU: 14.6628%
Train(8)[0%]: Loss: 35.7973, IoU: 12.8183%
Train(8)[50%]: Loss: 35.7249, IoU: 16.8459%
Test(8): Loss: 32.7556, IoU: 12.9835%
Train(9)[0%]: Loss: 36.3662, IoU: 12.8384%
Train(9)[50%]: Loss: 36.1398, IoU: 15.4729%
Test(9): Loss: 31.4469, IoU: 13.9934%
Train(10)[0%]: Loss: 31.7258, IoU: 19.3408%
Train(10)[50%]: Loss: 35.1433, IoU: 16.6951%
Test(10): Loss: 31.5513, IoU: 13.3725%
Train(11)[0%]: Loss: 35.4014, IoU: 15.1041%
Train(11)[50%]: Loss: 34.7345, IoU: 16.5593%
Test(11): Loss: 32.0198, IoU: 12.8700%
Train(12)[0%]: Loss: 32.6085, IoU: 19.8638%
Train(12)[50%]: Loss: 34.5332, IoU: 17.3144%
Test(12): Loss: 31.1530, IoU: 14.4616%
Train(13)[0%]: Loss: 36.3003, IoU: 18.1582%
Train(13)[50%]: Loss: 34.8242, IoU: 17.0479%
Test(13): Loss: 29.5593, IoU: 16.3307%
Train(14)[0%]: Loss: 31.6988, IoU: 18.7994%
Train(14)[50%]: Loss: 34.6328, IoU: 17.2020%
Test(14): Loss: 28.7458, IoU: 17.4596%
Train(15)[0%]: Loss: 33.3648, IoU: 18.0659%
Train(15)[50%]: Loss: 33.4994, IoU: 18.5468%
Test(15): Loss: 29.1038, IoU: 17.9824%
Train(16)[0%]: Loss: 35.5309, IoU: 16.5881%
Train(16)[50%]: Loss: 33.6779, IoU: 18.0851%
Test(16): Loss: 26.9904, IoU: 20.4777%
Train(17)[0%]: Loss: 33.9436, IoU: 15.1996%
Train(17)[50%]: Loss: 32.4749, IoU: 18.9965%
Test(17): Loss: 27.1103, IoU: 22.2234%
Train(18)[0%]: Loss: 33.2289, IoU: 18.6303%
Train(18)[50%]: Loss: 32.3624, IoU: 19.7700%
Test(18): Loss: 26.5488, IoU: 23.1957%
Train(19)[0%]: Loss: 34.5886, IoU: 16.4730%
Train(19)[50%]: Loss: 32.0087, IoU: 19.8285%
Test(19): Loss: 26.1528, IoU: 23.6127%
Train(20)[0%]: Loss: 27.9429, IoU: 23.4397%
Train(20)[50%]: Loss: 31.3822, IoU: 21.0365%
Test(20): Loss: 26.5507, IoU: 22.2917%
```

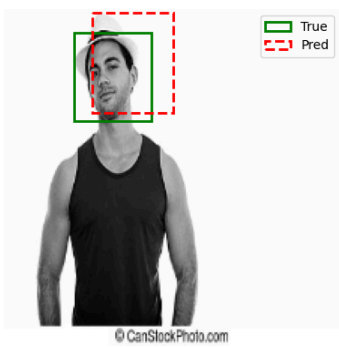
Fourthly, Evaluate the performance.

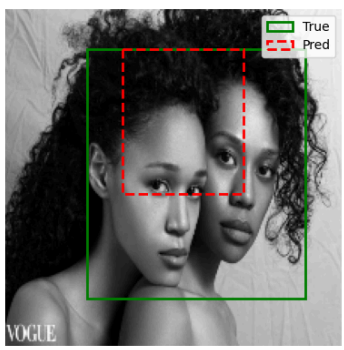
In [13]:
```
fig, axes = plt.subplots(1, 2, figsize=(15, 5))

axes[0].plot(train_losses_freeze, label='Train Loss', color='blue')
axes[0].plot(test_losses_freeze, label='Test Loss', color='orange')
axes[0].set_title('Train and Test Loss')
axes[0].set_xlabel('Epoch')
axes[0].set_ylabel('Loss')
axes[0].legend()
axes[0].grid(True)

axes[1].plot(train_IoUs_freeze, label='Train IoU', color='green')
axes[1].plot(test_IoUs_freeze, label='Test IoU', color='red')
axes[1].set_title('Train and Test IoU')
axes[1].set_xlabel('Epoch')
axes[1].set_ylabel('IoU')
axes[1].legend()
axes[1].grid(True)

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

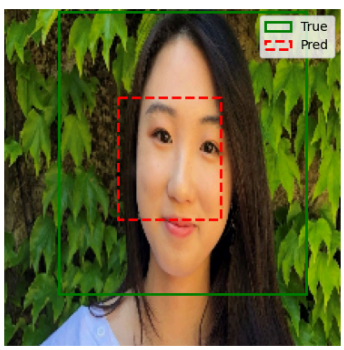


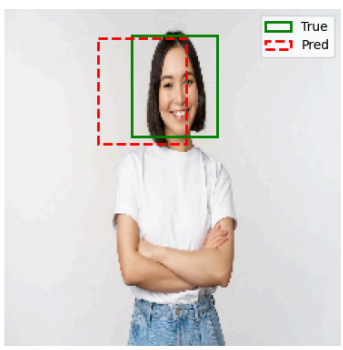


In [14]:
```
def draw_sample_predict(dataset, model, indices, device='cuda'):
    model.eval()
    fig, axes = plt.subplots(1, len(indices), figsize=(16, 4))

    for i, idx in enumerate(indices):
        sample = dataset[idx]
        image = sample[0].unsqueeze(0).to(device)
        image = image.float()
        true_bbox = sample[1]

        with torch.no_grad():
            pred_bbox = model(image).squeeze(0).cpu().numpy()

        image_np = sample[0].permute(1, 2, 0).cpu().numpy()

        axes[i].imshow(image_np)

        x_min, y_min, x_max, y_max = true_bbox
        rect_true = plt.Rectangle((x_min, y_min), x_max - x_min, y_max - y_min,
                                  linewidth=2, edgecolor='g', facecolor='none', label='True')
        axes[i].add_patch(rect_true)

        pred_x_min, pred_y_min, pred_x_max, pred_y_max = pred_bbox
        rect_pred = plt.Rectangle((pred_x_min, pred_y_min), pred_x_max - pred_x_min, pred_y_max - pred_y_min,
                                  linewidth=2, edgecolor='r', facecolor='none', linestyle='--', label='Pred')
        axes[i].add_patch(rect_pred)

        axes[i].legend(loc='upper right')
        axes[i].axis('off')

    plt.tight_layout()
    plt.show()

draw_sample_predict(test_dataset, model_freeze, indices=[0, 1, 2, 3], device='cuda')
```









1.3 Object Detection with End-to-End Fine-Tuning

We just unfreeze the backbone model and train its parameters together with regressor.

In [15]:
```
model_unfreeze = BoundingBoxRegressor(freeze_backbone=False).to(device)
criterion = nn.SmoothL1Loss()
trained_params = filter(lambda p: p.requires_grad, model_unfreeze.parameters())
optimizer = torch.optim.Adam(trained_params, lr=0.0001)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.5)
```

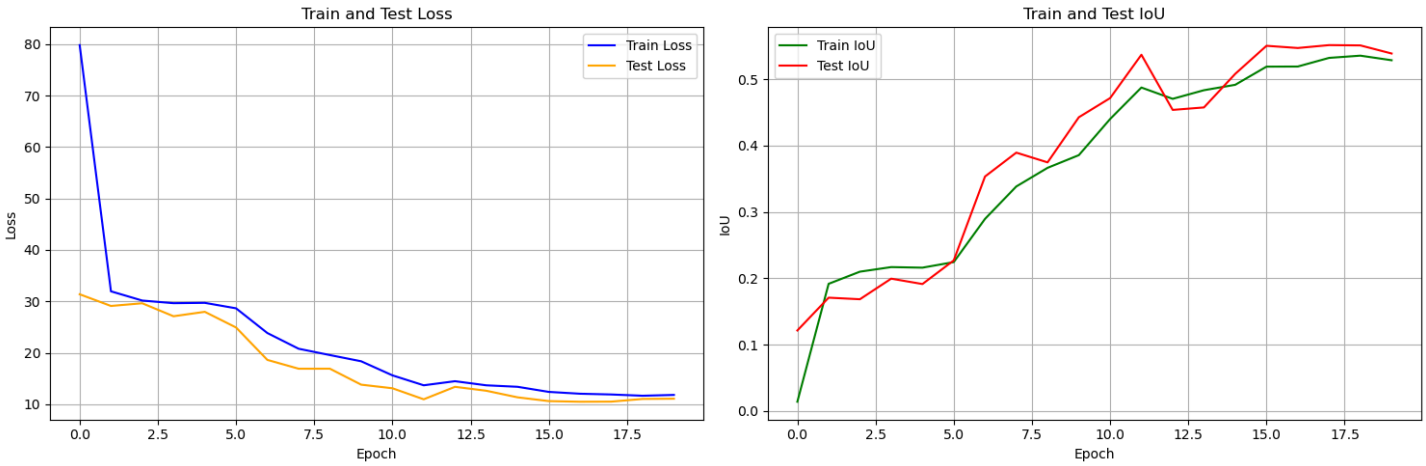
In [16]:
```
num_epochs = 20
batch_size = 32
train_losses_unfreeze = []
test_losses_unfreeze = []
train_IoUs_unfreeze = []
test_IoUs_unfreeze = []

for epoch in range(1, num_epochs + 1):
    train(model_unfreeze, device, train_loader, batch_size, criterion, optimizer, epoch, train_losses_unfreeze, train_IoUs_unfreeze)
    test(model_unfreeze, device, test_loader, batch_size, criterion, epoch, test_losses_unfreeze, test_IoUs_unfreeze)
```

```
Train(1)[0%]: Loss: 110.0982, IoU: 0.0000%
Train(1)[50%]: Loss: 104.9901, IoU: 0.0000%
Test(1): Loss: 31.3872, IoU: 12.1240%
Train(2)[0%]: Loss: 32.9646, IoU: 15.8100%
Train(2)[50%]: Loss: 33.0113, IoU: 19.4700%
Test(2): Loss: 29.1170, IoU: 17.0668%
Train(3)[0%]: Loss: 28.2082, IoU: 25.5726%
Train(3)[50%]: Loss: 30.9625, IoU: 20.5765%
Test(3): Loss: 29.6389, IoU: 16.8309%
Train(4)[0%]: Loss: 31.5087, IoU: 20.7868%
Train(4)[50%]: Loss: 29.6242, IoU: 22.4490%
Test(4): Loss: 27.1182, IoU: 19.9335%
Train(5)[0%]: Loss: 28.1664, IoU: 22.7403%
Train(5)[50%]: Loss: 29.9899, IoU: 21.9182%
Test(5): Loss: 27.9804, IoU: 19.1233%
Train(6)[0%]: Loss: 29.1762, IoU: 22.5122%
Train(6)[50%]: Loss: 29.5591, IoU: 21.8913%
Test(6): Loss: 24.9364, IoU: 22.6841%
Train(7)[0%]: Loss: 29.2265, IoU: 19.5882%
Train(7)[50%]: Loss: 25.6985, IoU: 27.4438%
Test(7): Loss: 18.6393, IoU: 35.3309%
Train(8)[0%]: Loss: 24.8694, IoU: 28.3896%
Train(8)[50%]: Loss: 20.9469, IoU: 34.3704%
Test(8): Loss: 16.9265, IoU: 38.9307%
Train(9)[0%]: Loss: 20.1279, IoU: 34.7158%
Train(9)[50%]: Loss: 20.1605, IoU: 36.1457%
Test(9): Loss: 16.9302, IoU: 37.4639%
Train(10)[0%]: Loss: 13.9551, IoU: 43.7116%
Train(10)[50%]: Loss: 19.5373, IoU: 38.0306%
Test(10): Loss: 13.8270, IoU: 44.2648%
Train(11)[0%]: Loss: 16.2100, IoU: 45.2525%
Train(11)[50%]: Loss: 16.5322, IoU: 44.2315%
Test(11): Loss: 13.1277, IoU: 47.1524%
Train(12)[0%]: Loss: 18.8270, IoU: 38.3069%
Train(12)[50%]: Loss: 14.2585, IoU: 48.1373%
Test(12): Loss: 10.9841, IoU: 53.6854%
Train(13)[0%]: Loss: 11.5899, IoU: 51.2312%
Train(13)[50%]: Loss: 14.5221, IoU: 48.2171%
Test(13): Loss: 13.4054, IoU: 45.3727%
Train(14)[0%]: Loss: 17.0873, IoU: 39.2255%
Train(14)[50%]: Loss: 14.1399, IoU: 48.4099%
Test(14): Loss: 12.6443, IoU: 45.7443%
Train(15)[0%]: Loss: 11.9311, IoU: 51.3749%
Train(15)[50%]: Loss: 13.6309, IoU: 49.5447%
Test(15): Loss: 11.3667, IoU: 50.8058%
Train(16)[0%]: Loss: 14.9576, IoU: 49.8463%
Train(16)[50%]: Loss: 12.1904, IoU: 53.8029%
Test(16): Loss: 10.6327, IoU: 55.0364%
Train(17)[0%]: Loss: 10.6277, IoU: 56.0676%
Train(17)[50%]: Loss: 12.2277, IoU: 52.6095%
Test(17): Loss: 10.5099, IoU: 54.7021%
Train(18)[0%]: Loss: 15.0449, IoU: 49.3998%
Train(18)[50%]: Loss: 11.8952, IoU: 55.1582%
Test(18): Loss: 10.5323, IoU: 55.1385%
Train(19)[0%]: Loss: 11.6282, IoU: 53.8874%
Train(19)[50%]: Loss: 11.5273, IoU: 55.1097%
Test(19): Loss: 11.0535, IoU: 55.0947%
Train(20)[0%]: Loss: 8.6933, IoU: 59.8899%
Train(20)[50%]: Loss: 12.3064, IoU: 52.7703%
Test(20): Loss: 11.0984, IoU: 53.8732%
```

Evaluate the performance.

In [17]:
```
fig, axes = plt.subplots(1, 2, figsize=(15, 5))

axes[0].plot(train_losses_unfreeze, label='Train Loss', color='blue')
axes[0].plot(test_losses_unfreeze, label='Test Loss', color='orange')
axes[0].set_title('Train and Test Loss')
axes[0].set_xlabel('Epoch')
```

```
axes[0].set_ylabel('Loss')
axes[0].legend()
axes[0].grid(True)

axes[1].plot(train_IoUs_unfreeze, label='Train IoU', color='green')
axes[1].plot(test_IoUs_unfreeze, label='Test IoU', color='red')
axes[1].set_title('Train and Test IoU')
axes[1].set_xlabel('Epoch')
axes[1].set_ylabel('IoU')
axes[1].legend()
axes[1].grid(True)

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



In [18]:
```
draw_sample_predict(test_dataset, model_unfreeze, indices=[0, 1, 2, 3], device='cuda')
```

Conclusion: It is clearly evident that training the bounding box prediction module along with the backbone significantly improves the model's performance. The model's IoU has increased remarkably from around 20% to over 50%. This indicates that fine-tuning the backbone model is crucial in transfer learning.

1.4 What if there are multiple objects?

First, my model cannot handle cases with either no objects or multiple objects. This is because my model's output is always a fixed-length vector of 4, representing the coordinates of the bottom-left and top-right corners of a single bounding box. Therefore, it always outputs only one bounding box coordinate. The issue is that the length of the output layer should be variable, not constant, but our model forcibly sets it to a constant.

Second, to address this problem, we can draw on the ideas from algorithms like R-CNN and YOLO. These algorithms first generate a large number of anchors, and then use selective search algorithms to select candidate region proposals. After that, the model performs bounding box regression on these region proposals, and finally, the Non-Maximum Suppression (NMS) algorithm is used to select the final bounding boxes. This approach works by generating far more boxes than the actual number of objects, and then filtering them to retain the appropriate ones. In this way, the number of predicted bounding boxes can be flexibly adjusted, allowing the model to handle both no-object and multi-object scenarios.

Reference: R. Girshick, J. Donahue, T. Darrell and J. Malik, "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation," 2014 IEEE Conference on Computer Vision and Pattern Recognition, Columbus, OH, USA, 2014, pp. 580-587, doi: 10.1109/CVPR.2014.81.

J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 2016, pp. 779-788, doi: 10.1109/CVPR.2016.91.

Problem 2: Optimizers from Scratch.

2.1 Optimizer Implementaiton.

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim
import torchvision
import torchvision.transforms as transforms
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np
import math
import contextlib
import io
```

Firstly, Define the Optimizers.

1.Momentum.

In [2]:
```
class MomentumOptimizer:
    def __init__(self, parameters, lr=0.01, beta=0.9, weight_decay=0.001):
        self.parameters = list(parameters)
        self.lr = lr
        self.beta = beta
        self.weight_decay = weight_decay
        self.m = [torch.zeros_like(p) for p in self.parameters]

    def step(self):
        for i, parameter in enumerate(self.parameters):
            if parameter.grad is not None:
                self.m[i] = self.beta * self.m[i] + (parameter.grad + self.weight_decay * torch.sign(parameter.data)) # Th
                parameter.data -= self.lr * self.m[i] # update the parameters

    def zero_grad(self):
        for parameter in self.parameters:
            if parameter.grad is not None:
                parameter.grad.zero_()
```

2.Nesterov's Accelerated Gradient.

In [3]:
```
class NAGOptimizer:
    def __init__(self, parameters, lr=0.001, beta=0.95, weight_decay=0.001):
        self.parameters = list(parameters)
        self.lr = lr
        self.beta = beta
        self.weight_decay = weight_decay
        self.t1 = [torch.clone(p).detach() for p in self.parameters] # The lookahead point_t
        self.t2 = [torch.zeros_like(p) for p in self.parameters]  # The lookahead point_t+1
        self.lambda1 = 0 # The lambda_t
        self.lambda2 = 0 # The lambda_t+1
        self.gamma = 0

    def step(self):
        for i, parameter in enumerate(self.parameters):
            if parameter.grad is not None:
                self.lambda2 = (1 + math.sqrt(1 + 4 * (self.lambda1**2))) / 2 # Update the lambda and gamma.
                self.gamma = (1 - self.lambda1) / self.lambda2
                self.lambda1 = self.lambda2 # Update the lambda_t, preparing for the next turn.

                grad_with_l1 = parameter.grad + self.weight_decay * torch.sign(parameter.data) # Compute the gradient of l

                self.t2[i] = parameter.data - self.lr * grad_with_l1 / self.beta # Update the lookahead term.

                parameter.data = (1 - self.gamma) * self.t2[i] + self.gamma * self.t1[i] # Update the parameters.

                self.t1[i] = self.t2[i].clone().detach() # Update the lookahead term_t, preparing for the next turn.

    def zero_grad(self):
        for parameter in self.parameters:
            if parameter.grad is not None:
                parameter.grad.zero_()
```

3.RMSprop

In [4]:
```
class RMSpropOptimizer:
    def __init__(self, parameters, lr=0.001, beta=0.95, epsilon=1e-8, gamma=1, weight_decay=0.001):
        self.parameters = list(parameters)
        self.lr = lr
        self.beta = beta
        self.weight_decay=weight_decay
        self.epsilon = epsilon
        self.gamma = gamma
        self.v = [torch.zeros_like(p) for p in self.parameters] # The Accumulated Squared Gradient.

    def step(self):
        for i, parameter in enumerate(self.parameters):
            if parameter.grad is not None:
                grad_with_l1 = parameter.grad + self.weight_decay * torch.sign(parameter.data) # Compute the gradient of l

                self.v[i] = (1 - self.beta) * self.v[i] +  self.beta * (grad_with_l1 * grad_with_l1) # Update the Accumula

                b_k = self.gamma / (torch.sqrt(self.v[i]) + self.epsilon) # Compute the Preconditioning Term.

                parameter.data -= self.lr * b_k * grad_with_l1 # Update the parameters.

    def zero_grad(self):
        for parameter in self.parameters:
            if parameter.grad is not None:
                parameter.grad.zero_()
```

4.Adam.

In [5]:
```
class AdamOptimizer:
    def __init__(self, parameters, lr=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8, weight_decay=0.001):
        self.parameters = list(parameters)
        self.lr = lr
        self.weight_decay = weight_decay
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.m1 = [torch.zeros_like(p) for p in self.parameters]  # First Order Momentum.
        self.m2 = [torch.zeros_like(p) for p in self.parameters]  # Second Order Momentum.
        self.t = 0 # The Step.

    def step(self):
        self.t += 1

        for i, parameter in enumerate(self.parameters):
            if parameter.grad is not None:
                grad_with_l1 = parameter.grad + self.weight_decay * torch.sign(parameter.data) # Compute the gradient of l

                self.m1[i] = self.beta1 * self.m1[i] + (1 - self.beta1) * grad_with_l1 # Update the First Order Momentum.

                self.m2[i] = self.beta2 * self.m2[i] + (1 - self.beta2) * (grad_with_l1 * grad_with_l1) # Update the Secor

                m1_hat = self.m1[i] / (1 - self.beta1 ** self.t) # Debias the First Order Momentum.

                m2_hat = self.m2[i] / (1 - self.beta2 ** self.t) # Debias the Second Order Momentum.

                parameter.data -= self.lr * m1_hat / (torch.sqrt(m2_hat) + self.epsilon) # Update the parameters.

    def zero_grad(self):
        for parameter in self.parameters:
            if parameter.grad is not None:
                parameter.grad.zero_()
```

Secondly, Define the Data, Model and Trainer.

Prepare the datas.

In [6]:
```
train_set,test_set,train_loader,test_loader = {},{},{},{}
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.1307,), (0.3081,))])
train_set['mnist'] = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_set['mnist'] = torchvision.datasets.MNIST(root='./data', train=False, download=True, transform=transform)

# datas of different batch size.
train_loader0 = torch.utils.data.DataLoader(train_set['mnist'], batch_size=100, shuffle=True, num_workers=0)
test_loader0 = torch.utils.data.DataLoader(test_set['mnist'], batch_size=100, shuffle=False, num_workers=0)
train_loader1 = torch.utils.data.DataLoader(train_set['mnist'], batch_size=4, shuffle=True, num_workers=0)
test_loader1 = torch.utils.data.DataLoader(test_set['mnist'], batch_size=4, shuffle=False, num_workers=0)
train_loader2 = torch.utils.data.DataLoader(train_set['mnist'], batch_size=8, shuffle=True, num_workers=0)
test_loader2 = torch.utils.data.DataLoader(test_set['mnist'], batch_size=8, shuffle=False, num_workers=0)
train_loader3 = torch.utils.data.DataLoader(train_set['mnist'], batch_size=16, shuffle=True, num_workers=0)
test_loader3 = torch.utils.data.DataLoader(test_set['mnist'], batch_size=16, shuffle=False, num_workers=0)
```

```
train_loader4 = torch.utils.data.DataLoader(train_set['mnist'], batch_size=32, shuffle=True, num_workers=0)
test_loader4 = torch.utils.data.DataLoader(test_set['mnist'], batch_size=32, shuffle=False, num_workers=0)
```

Define the model. (Use the example model in discussion section)

In [7]:
```
class ExampleNet(nn.Module):
    def __init__(self,in_channels, out_channels):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels, 8, 2, 1)
        self.fc1 = nn.Linear(1352, out_channels)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = torch.flatten(x, 1)
        x = self.fc1(x)

        return x
```

Define the trainer. (Use the code present in discussion section)

In [8]:
```
def train(model, device, train_loader, criterion, optimizer, epoch, train_losses):
    model.train()
    train_loss = 0
    batch_index = 0

    for batch_idx, (data, target) in enumerate(train_loader):
        batch_index = batch_idx
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()
        if batch_idx % (len(train_loader)//2) == 0:
            print('Train({})[{:.0f}%]: Loss: {:.4f}'.format(
                epoch, 100. * batch_idx / len(train_loader), train_loss/(batch_idx+1)))

    train_losses.append(train_loss/(batch_index+1))

def test(model, device, test_loader, criterion, epoch, batch_size, test_losses):
    model.eval()
    test_loss = 0
    correct = 0

    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += criterion(output, target).item() # sum up batch loss
            pred = output.argmax(dim=1, keepdim=True) # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss = (test_loss*batch_size)/len(test_loader.dataset)
    print('Test({}): Loss: {:.4f}, Accuracy: {:.4f}%'.format(
        epoch, test_loss, 100. * correct / len(test_loader.dataset)))
    test_losses.append(test_loss)
```

Some other preparations.

In [9]:
```
device = 'cuda'
device = torch.device(device)
num_epochs = 5
in_channels = 1
out_channels = 10
```

Thiredly, Test Whether the Optimizer is works well.

In [10]:
```
lr = 0.001
train_losses = []
test_losses = []
batch_size = 100
model = ExampleNet(in_channels, out_channels).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = MomentumOptimizer(model.parameters(), lr)

for epoch in range(1, num_epochs + 1):
    train(model, device, train_loader0, criterion, optimizer, epoch, train_losses)
    test(model, device, test_loader0, criterion, epoch, batch_size, test_losses)
```

```
Train(1)[0%]: Loss: 2.3308
Train(1)[50%]: Loss: 0.7269
Test(1): Loss: 0.3142, Accuracy: 91.0700%
Train(2)[0%]: Loss: 0.3048
Train(2)[50%]: Loss: 0.3233
Test(2): Loss: 0.2825, Accuracy: 92.1400%
Train(3)[0%]: Loss: 0.4576
Train(3)[50%]: Loss: 0.3028
Test(3): Loss: 0.2726, Accuracy: 92.3700%
Train(4)[0%]: Loss: 0.2846
Train(4)[50%]: Loss: 0.2844
Test(4): Loss: 0.2640, Accuracy: 92.6200%
Train(5)[0%]: Loss: 0.3421
Train(5)[50%]: Loss: 0.2754
Test(5): Loss: 0.2512, Accuracy: 92.9600%
```

In [11]:
```
lr = 0.001
batch_size = 100
train_losses = []
test_losses = []
model = ExampleNet(in_channels,out_channels).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = NAGOptimizer(model.parameters(), lr)

for epoch in range(1, num_epochs + 1):
    train(model, device, train_loader0, criterion, optimizer, epoch, train_losses)
    test(model, device, test_loader0, criterion, epoch, batch_size, test_losses)
```

```
Train(1)[0%]: Loss: 2.2498
Train(1)[50%]: Loss: 0.5746
Test(1): Loss: 0.1765, Accuracy: 94.8000%
Train(2)[0%]: Loss: 0.1712
Train(2)[50%]: Loss: 0.1895
Test(2): Loss: 0.1783, Accuracy: 94.6600%
Train(3)[0%]: Loss: 0.1918
Train(3)[50%]: Loss: 0.1807
Test(3): Loss: 0.1825, Accuracy: 94.3500%
Train(4)[0%]: Loss: 0.2481
Train(4)[50%]: Loss: 0.1926
Test(4): Loss: 0.1868, Accuracy: 94.2000%
Train(5)[0%]: Loss: 0.2292
Train(5)[50%]: Loss: 0.1925
Test(5): Loss: 0.1836, Accuracy: 94.0200%
```

In [12]:
```
lr = 0.001
batch_size = 100
train_losses = []
test_losses = []
model = ExampleNet(in_channels,out_channels).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = RMSpropOptimizer(model.parameters(), lr)

for epoch in range(1, num_epochs + 1):
    train(model, device, train_loader0, criterion, optimizer, epoch, train_losses)
    test(model, device, test_loader0, criterion, epoch, batch_size, test_losses)
```

```
Train(1)[0%]: Loss: 2.2991
Train(1)[50%]: Loss: 0.5209
Test(1): Loss: 0.2739, Accuracy: 92.3800%
Train(2)[0%]: Loss: 0.4226
Train(2)[50%]: Loss: 0.2753
Test(2): Loss: 0.2338, Accuracy: 93.5200%
Train(3)[0%]: Loss: 0.2224
Train(3)[50%]: Loss: 0.2403
Test(3): Loss: 0.2040, Accuracy: 94.3100%
Train(4)[0%]: Loss: 0.2301
Train(4)[50%]: Loss: 0.2147
Test(4): Loss: 0.1927, Accuracy: 94.6300%
Train(5)[0%]: Loss: 0.1626
Train(5)[50%]: Loss: 0.2111
Test(5): Loss: 0.1820, Accuracy: 94.9100%
```

In [13]:
```
lr = 0.001
batch_size = 100
train_losses = []
test_losses = []
model = ExampleNet(in_channels,out_channels).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = AdamOptimizer(model.parameters(), lr)

for epoch in range(1, num_epochs + 1):
    train(model, device, train_loader0, criterion, optimizer, epoch, train_losses)
    test(model, device, test_loader0, criterion, epoch, batch_size, test_losses)
```

```
Train(1)[0%]: Loss: 2.3880
Train(1)[50%]: Loss: 0.5182
Test(1): Loss: 0.2197, Accuracy: 94.0200%
Train(2)[0%]: Loss: 0.2132
Train(2)[50%]: Loss: 0.2309
Test(2): Loss: 0.1838, Accuracy: 95.0700%
Train(3)[0%]: Loss: 0.2369
Train(3)[50%]: Loss: 0.1907
Test(3): Loss: 0.1802, Accuracy: 95.0100%
Train(4)[0%]: Loss: 0.2736
Train(4)[50%]: Loss: 0.1723
Test(4): Loss: 0.1509, Accuracy: 95.7900%
Train(5)[0%]: Loss: 0.1244
Train(5)[50%]: Loss: 0.1609
Test(5): Loss: 0.1470, Accuracy: 95.7000%
```

From the results above, we can see that all the classification accuracies under different optimizers are good, which are above 90% after 4 epochs.

(92.9600% for Momentum; 94.0200% for NAG; 94.9100% for RMSprop; 95.7000% for Adam). So the implementation is correct.

2.2 Comparing your optimizer efficiency

Define the super parameters.

In [14]:
```
device = 'cuda'
device = torch.device(device)
num_epochs = 5
in_channels = 1
out_channels = 10
criterion = nn.CrossEntropyLoss()

batch_sizes = [4, 8, 16, 32]
learning_rates = [0.001, 0.0005, 0.0001]
train_loaders = [train_loader1, train_loader2, train_loader3, train_loader4]
test_loaders = [test_loader1, test_loader2, test_loader3, test_loader4]
```

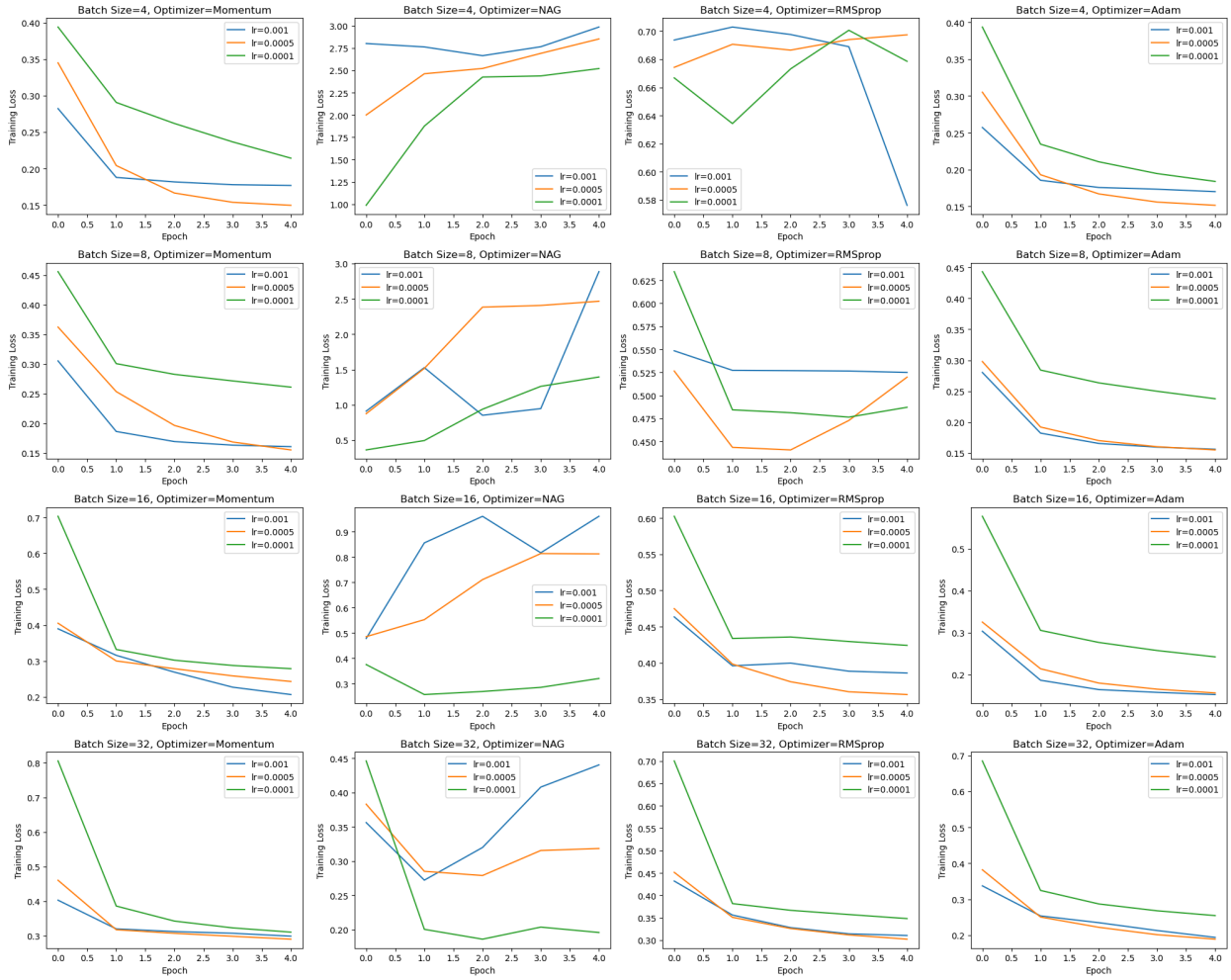
Define the train process.

In [ ]:
```
train_losses_all = []
test_losses_all = []

for i, batch_size in enumerate(batch_sizes):

    batch_size = batch_size

    for j, lr in enumerate(learning_rates):
        model = ExampleNet(in_channels,out_channels).to(device)
        optimizer = MomentumOptimizer(model.parameters(), lr)
        train_losses = []
        test_losses = []

        with contextlib.redirect_stdout(io.StringIO()):
            for epoch in range(1, num_epochs + 1):
                train(model, device, train_loaders[i], criterion, optimizer, epoch, train_losses)
                test(model, device, test_loaders[i], criterion, epoch, batch_size, test_losses)

        train_losses_all.append(train_losses)
        test_losses_all.append(test_losses)

    for j, lr in enumerate(learning_rates):
        model = ExampleNet(in_channels,out_channels).to(device)
        optimizer = NAGOptimizer(model.parameters(), lr)
        train_losses = []
        test_losses = []

        with contextlib.redirect_stdout(io.StringIO()):
            for epoch in range(1, num_epochs + 1):
                train(model, device, train_loaders[i], criterion, optimizer, epoch, train_losses)
                test(model, device, test_loaders[i], criterion, epoch, batch_size, test_losses)

        train_losses_all.append(train_losses)
        test_losses_all.append(test_losses)

    for j, lr in enumerate(learning_rates):
        model = ExampleNet(in_channels,out_channels).to(device)
        optimizer = RMSpropOptimizer(model.parameters(), lr)
        train_losses = []
        test_losses = []
```

```
        with contextlib.redirect_stdout(io.StringIO()):
            for epoch in range(1, num_epochs + 1):
                train(model, device, train_loaders[i], criterion, optimizer, epoch, train_losses)
                test(model, device, test_loaders[i], criterion, epoch, batch_size, test_losses)

        train_losses_all.append(train_losses)
        test_losses_all.append(test_losses)

    for j, lr in enumerate(learning_rates):
        model = ExampleNet(in_channels,out_channels).to(device)
        optimizer = AdamOptimizer(model.parameters(), lr)
        train_losses = []
        test_losses = []

        with contextlib.redirect_stdout(io.StringIO()):
            for epoch in range(1, num_epochs + 1):
                train(model, device, train_loaders[i], criterion, optimizer, epoch, train_losses)
                test(model, device, test_loaders[i], criterion, epoch, batch_size, test_losses)

        train_losses_all.append(train_losses)
        test_losses_all.append(test_losses)

    print(i)
```

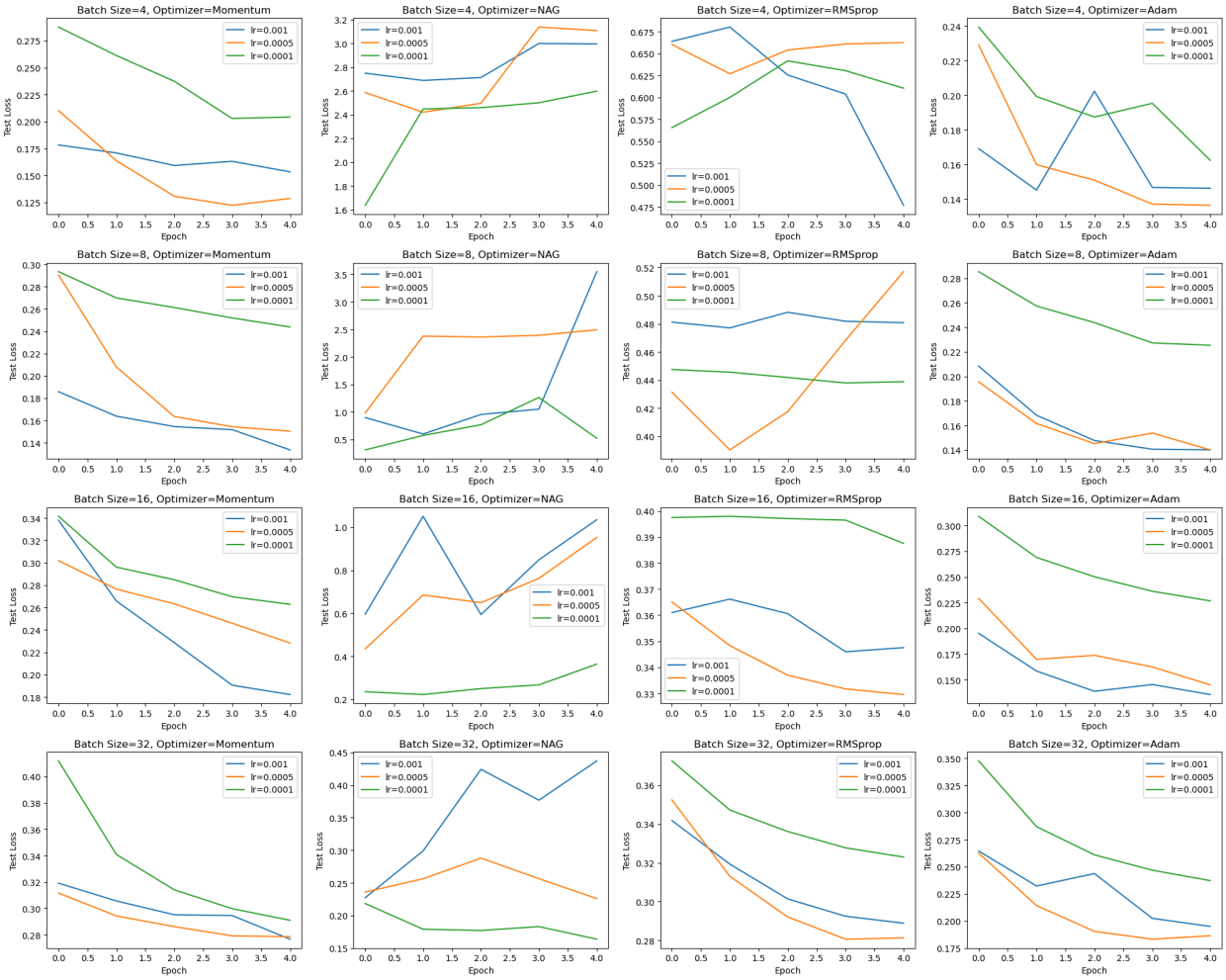
Compare the Train loss under different batch size, optimizers and learning rates.

In [17]:
```
batch_sizes = [4, 8, 16, 32]
optimizers = ['Momentum', 'NAG', 'RMSprop', 'Adam']
learning_rates = [0.001, 0.0005, 0.0001]


fig, axs = plt.subplots(4, 4, figsize=(20, 16))

for i, batch_size in enumerate(batch_sizes):
    for j, optimizer in enumerate(optimizers):
        ax = axs[i, j]

        for k, lr in enumerate(learning_rates):
            idx = i * 12 + j * 3 + k
            ax.plot(train_losses_all[idx], label=f"lr={lr}")

        ax.set_title(f"Batch Size={batch_size}, Optimizer={optimizer}")
        ax.set_xlabel("Epoch")
        ax.set_ylabel("Training Loss")
        ax.legend()

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



Compare the Test loss under different batch size, optimizers and learning rates.

In [20]:
```
batch_sizes = [4, 8, 16, 32]
optimizers = ['Momentum', 'NAG', 'RMSprop', 'Adam']
learning_rates = [0.001, 0.0005, 0.0001]

fig, axs = plt.subplots(4, 4, figsize=(20, 16))

for i, batch_size in enumerate(batch_sizes):
    for j, optimizer in enumerate(optimizers):
        ax = axs[i, j]

        for k, lr in enumerate(learning_rates):
            idx = i * 12 + j * 3 + k
            ax.plot(test_losses_all[idx], label=f"lr={lr}")

        ax.set_title(f"Batch Size={batch_size}, Optimizer={optimizer}")
        ax.set_xlabel("Epoch")
        ax.set_ylabel("Test Loss")
        ax.legend()

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



From the above figures, it can be seen that the NAG algorithm has the worse stability when the batch size is small and the learning rate is high. But it is worth noting that NAG have very good performance under relatively lower learning rate and larger batch size, as shown in figures (lr=0.0001; batch size>=16) and part1.

RMSprop shows a medium performance.

In terms of overall performance, the Adam and Momentum algorithms are undoubtedly the best performers, exhibiting high stability across different batch sizes and learning rates, and always achieving the most optimization in the final loss function.

Problem 1

In [1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np

device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

Define the Sparse Autoencoder, I use MLP as encoder and Dh=x_hat as decoder.

In [2]:
```
class SparseAutoencoder(nn.Module):
    def __init__(self, input_size, hidden_size, sparsity_ratio):
        super(SparseAutoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_size, 1024),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(1024, 1024),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(1024, hidden_size)
        )
        self.decoder = nn.Linear(hidden_size, input_size)
        self.sparsity_ratio = sparsity_ratio

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded, encoded

    def sparse_loss(self, encoded):
        l1 = torch.mean(torch.abs(encoded))
        return self.sparsity_ratio * l1
```

Prepare the data.

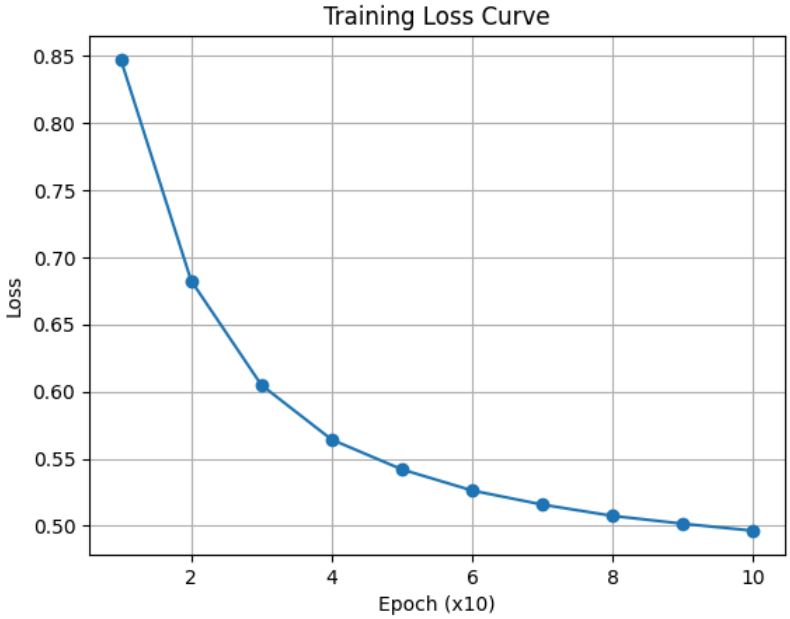
In [3]:
```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,)),
    transforms.Lambda(lambda x: x.view(-1))
])

batch_size = 10000
train_data = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)
test_data = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
test_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=False)
```

Train the model using the MSE as loss and add the L1 of hiddle code as penalty to sparse the h.

In [ ]:
```
learning_rate = 0.0005
training_epochs = 200
input_size = 784
hidden_size = 1174
sparsity_ratio = 1.75

model = SparseAutoencoder(input_size, hidden_size, sparsity_ratio)
model.to(device)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

train_loss_list = []

model.train()
for epoch in range(training_epochs):
    running_loss = 0
    for i, (inputs, _) in enumerate(train_loader):
        inputs = inputs.view(inputs.size(0), -1).to(device)
        inputs = inputs + torch.randn_like(inputs) * 0.5

        optimizer.zero_grad()
        outputs, encoded = model(inputs)
        loss = criterion(outputs, inputs) + model.sparse_loss(encoded)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    if (epoch + 1) % (training_epochs // 10) == 0:
            avg_loss = running_loss / len(train_loader)
```

```
            train_loss_list.append(avg_loss)
            print(f'Epoch [{epoch + 1}/{training_epochs}], Loss: {avg_loss:.4f}')

print('Finished Training')

plt.plot(range(1, len(train_loss_list) + 1), train_loss_list, marker='o')
plt.xlabel('Epoch (x10)')
plt.ylabel('Loss')
plt.title('Training Loss Curve')
plt.grid(True)
plt.show()
```

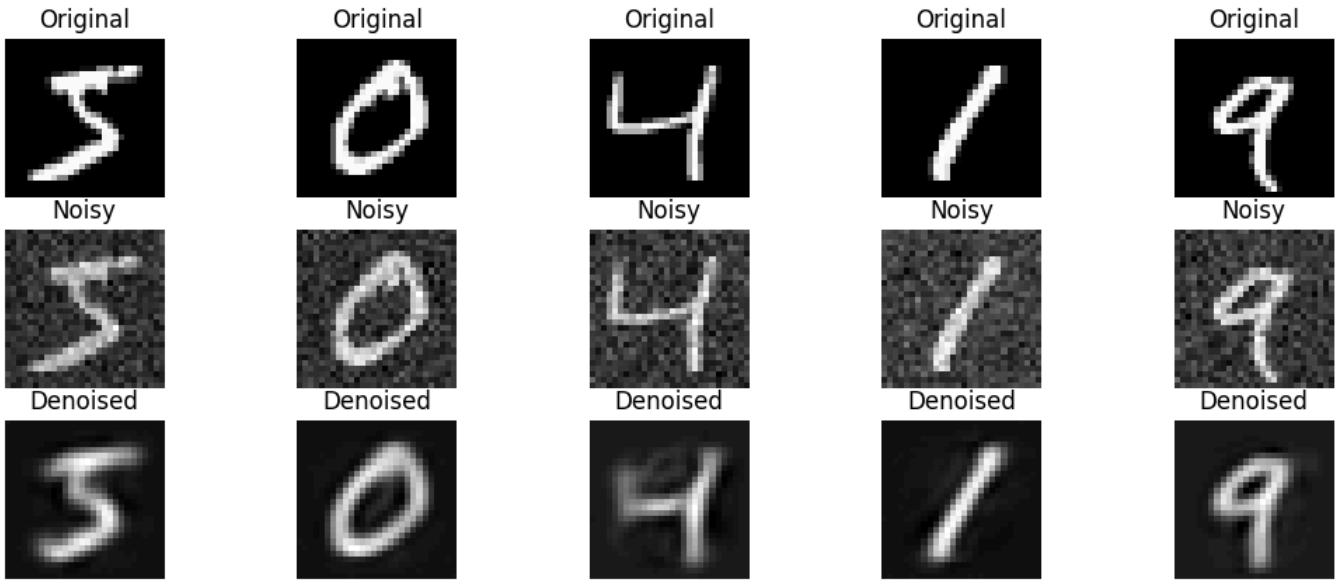
```
Epoch [20/200], Loss: 0.8470
Epoch [40/200], Loss: 0.6825
Epoch [60/200], Loss: 0.6049
Epoch [80/200], Loss: 0.5642
Epoch [100/200], Loss: 0.5420
Epoch [120/200], Loss: 0.5263
Epoch [140/200], Loss: 0.5158
Epoch [160/200], Loss: 0.5075
Epoch [180/200], Loss: 0.5016
Epoch [200/200], Loss: 0.4964
Finished Training
```



Evaluate the performance and plot the comparation of pictures.

In [ ]:

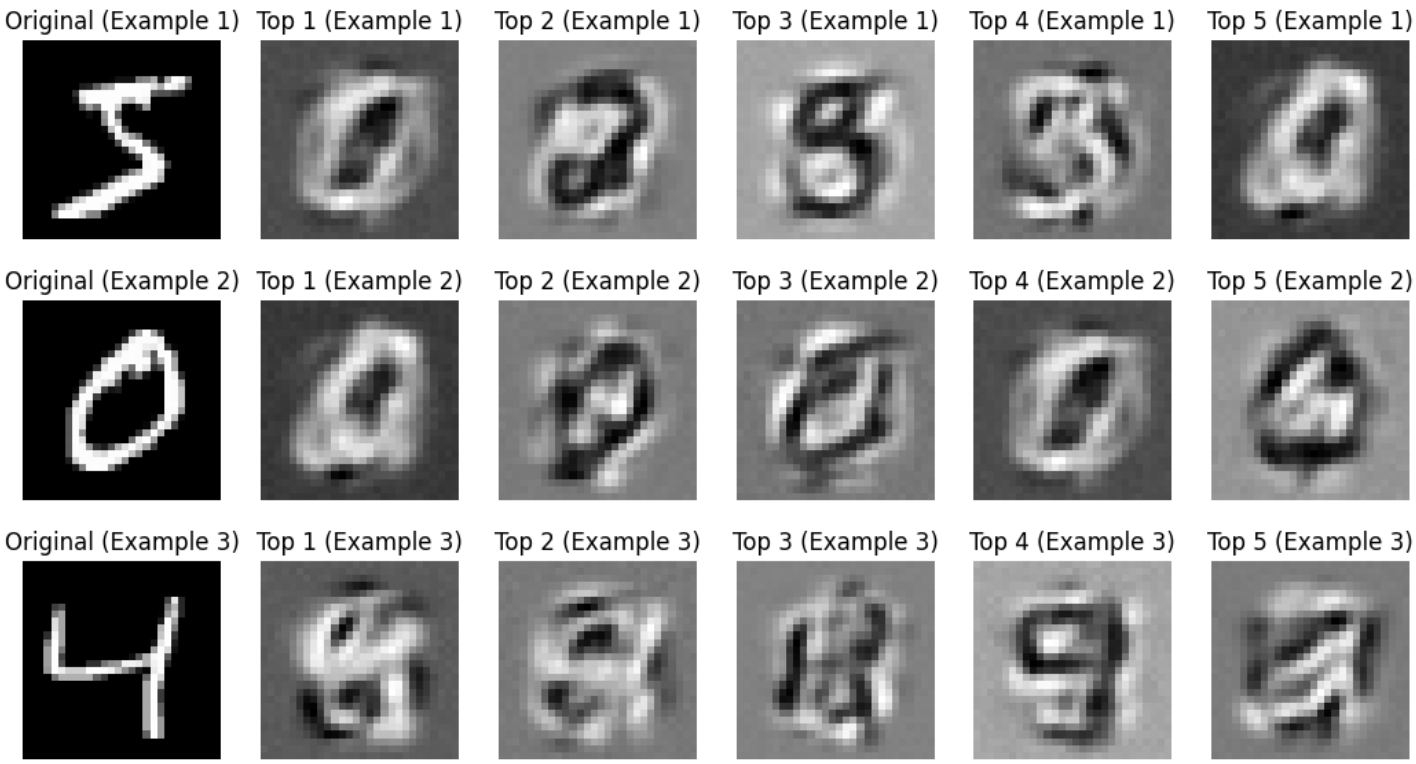
```
model.eval()
with torch.no_grad():
    data_iter = iter(test_loader)
    inputs, _ = next(data_iter)
    inputs = inputs.view(inputs.size(0), -1).to(device)

    noise = torch.randn_like(inputs) * 0.4
    noisy_inputs = inputs + noise

    outputs, encoded = model(noisy_inputs)

f, a = plt.subplots(3, 5, figsize=(13, 5))
for i in range(5):
    a[0][i].imshow(inputs[i].cpu().numpy().reshape(28, 28), cmap='gray')
    a[0][i].axis('off')
    a[0][i].set_title("Original")

    a[1][i].imshow(noisy_inputs[i].cpu().numpy().reshape(28, 28), cmap='gray')
    a[1][i].axis('off')
    a[1][i].set_title("Noisy")

    a[2][i].imshow(outputs[i].cpu().numpy().reshape(28, 28), cmap='gray')
    a[2][i].axis('off')
    a[2][i].set_title("Denoised")
plt.show()
```



Draw the top 5 dictionary vectors for some examples.

In [ ]:

```
D = model.decoder.weight.data.cpu().numpy()

for example_index in range(3):

    h = encoded[example_index].cpu().numpy()
    original_img = inputs[example_index].cpu().numpy().reshape(28, 28) * 0.3081 + 0.1307
    intensities = np.abs(h)

    top_5_indices = np.argsort(intensities)[-5:][::-1]

    f, a = plt.subplots(1, 6, figsize=(13, 5))
    a[0].imshow(original_img, cmap='gray')
    a[0].axis('off')
    a[0].set_title(f'Original (Example {example_index+1})')
    for i, idx in enumerate(top_5_indices):
        dictionary_vector = D[:, idx].reshape(28, 28) * 0.3081 + 0.1307
        a[i+1].imshow(dictionary_vector, cmap='gray')
        a[i+1].axis('off')
        a[i+1].set_title(f'Top {i+1} (Example {example_index+1})')
    plt.show()
```



Problem 2

In [1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.distributions import Normal
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
from IPython.display import clear_output
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

Prepare the data.

In [ ]:
```
batch_size = 1024

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Lambda(lambda x: x.view(-1))
])

train_loader = torch.utils.data.DataLoader(datasets.MNIST('./data', train=True, download=True, transform=transform), batch_size=batch_size, s
test_loader = torch.utils.data.DataLoader(datasets.MNIST('./data', train=False, transform=transform), batch_size=batch_size, shuffle=True)
```

Define the Model as well as the loss function.

PPCA Setup

The generative model of PPCA is given by:

$$x = Wz + \mu + \epsilon$$

where $W \in \mathbb{R}^{D \times K}$, $z \sim N(0, I)$, and $\epsilon \sim N(0, \Psi)$, $\Psi = diag(\sigma_1^2, \ldots\ldots, \sigma_D^2)$.

Loss Function

The negative log-likelihood loss function is given by:

$$\mathcal{L}_{\text{loss}}(W, \Psi) = -likelihood(W, \Psi) = \frac{N}{2}\log|\Sigma| + \frac{1}{2}\sum_{i=1}^{N}(x_i - \mu)^T\Sigma^{-1}(x_i - \mu) + Constant$$

where $\Sigma = WW^T + \Psi$.

Posterior of Latent Variables

The posterior mean of the latent variable z_i given an observation x_i is:

$$z_i^{\text{mean}} = \mathbb{E}[z_i \mid x_i] = M^{-1}W^T\Psi^{-1}(x_i - \mu)$$

where:

$$M = I + W^T\Psi^{-1}W$$

Reconstruction of x

The reconstruction of x_i from the posterior mean of z_i is:

$$\hat{x} = Wz_i^{\text{mean}} + \mu = WM^{-1}W^T\Psi^{-1}(x_i - \mu) + \mu$$

In [ ]:
```
class PPCA(nn.Module):
    def __init__(self, data_dim, latent_dim):
        super(PPCA, self).__init__()
        self.data_dim = data_dim
        self.latent_dim = latent_dim

        self.W = nn.Parameter(torch.randn(data_dim, latent_dim) * 0.01)
        self.log_sigma_sq = nn.Parameter(torch.zeros(data_dim))

    def forward(self, x):
        Psi_inv = torch.diag(torch.exp(-self.log_sigma_sq)).to(device)
        Wt_Psi_inv = self.W.t() @ Psi_inv
        M_inv = torch.inverse(torch.eye(self.latent_dim, device=device) + Wt_Psi_inv @ self.W)
        z_mean = M_inv @ Wt_Psi_inv @ (x - x.mean(dim=0)).t()
        z_mean = z_mean.t()
        x_reconstructed = z_mean @ self.W.t() + x.mean(dim=0)

        return x_reconstructed

    def loss_function(self, x):
        WWt = self.W @ self.W.t()
        Psi = torch.diag(torch.exp(self.log_sigma_sq))
```

```
        Sigma = WWt + Psi
        log_det_Sigma = torch.logdet(Sigma)

        Psi_inv = torch.diag(torch.exp(-self.log_sigma_sq))
        Sigma_inv = Psi_inv - Psi_inv @ self.W @ torch.inverse(torch.eye(self.latent_dim, device=device) + self.W.t() @ Psi_inv @ self.W) @ s

        second_term = torch.sum((x - x.mean(dim=0)) @ Sigma_inv * (x - x.mean(dim=0)), dim=1)
        loss = 0.5 * (log_det_Sigma + second_term.mean())

        return loss
```

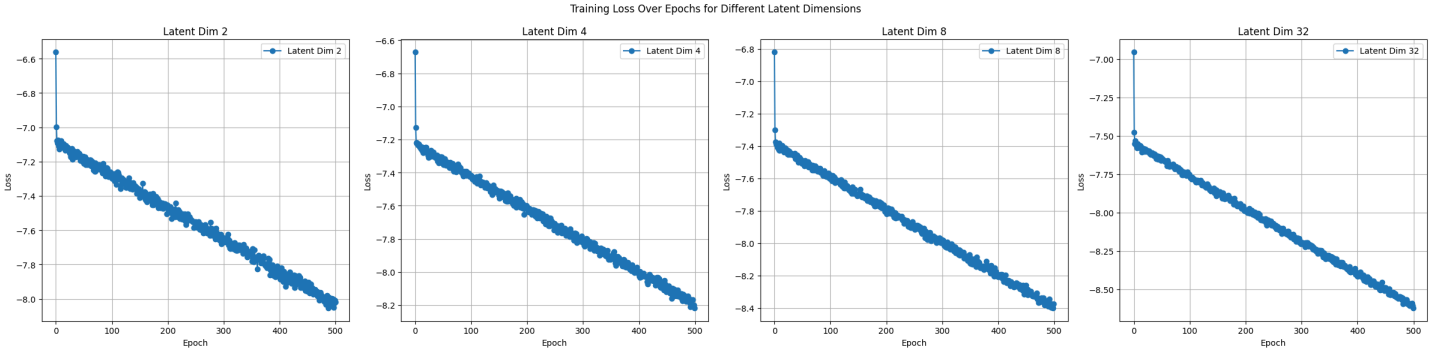
Train the Models.

In [ ]:
```
latent_dims = [2, 4, 8, 32]
all_train_losses = []
trained_models = []
learning_rate = 0.1
num_epochs = 500

def train_ppca(model, train_loader, train_losses, epochs=20, lr=0.01):
    optimizer = optim.Adam(model.parameters(), lr=lr)
    scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[2, 10, 20, 40, 50, 70, 90, 110, 130, 150, 180, 200, 300, 400,], gamma=0

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        for batch_idx, (data, _) in enumerate(train_loader):
            data = data.to(device)
            optimizer.zero_grad()
            loss = model.loss_function(data)
            loss.backward()
            optimizer.step()
            scheduler.step()
        running_loss += loss.item()
        train_losses.append(running_loss/len(train_loader))
    print("Training Complete!")

    return model

for latent_dim in latent_dims:
    model = PPCA(data_dim=28*28, latent_dim=latent_dim).to(device)
    save_path = f"ppca_model_latent_dim_{latent_dim}.pth"
    train_losses = []

    trained_model = train_ppca(model, train_loader, train_losses, epochs=num_epochs, lr=learning_rate)
    all_train_losses.append(train_losses)
    trained_models.append(trained_model)
    torch.save(trained_model.state_dict(), save_path)
```

```
Training Complete!
Training Complete!
Training Complete!
Training Complete!
```

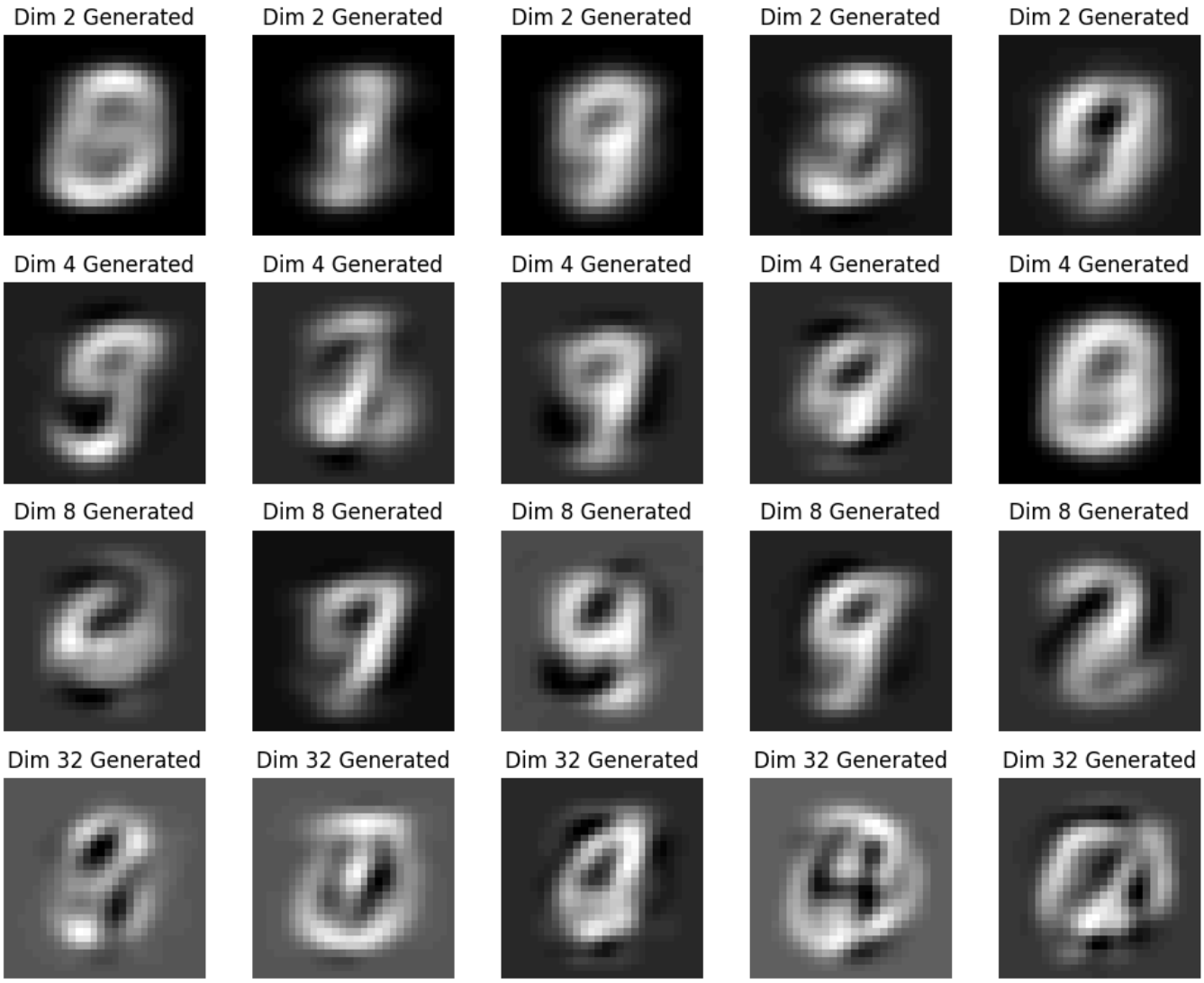
Plot the loss curve.

In [6]:
```
def plot_all_train_losses_in_row(all_train_losses, latent_dims):
    fig, axs = plt.subplots(1, 4, figsize=(24, 6))
    fig.suptitle('Training Loss Over Epochs for Different Latent Dimensions')

    for idx, (train_losses, latent_dim) in enumerate(zip(all_train_losses, latent_dims)):
        axs[idx].plot(train_losses, label=f'Latent Dim {latent_dim}', marker='o')
        axs[idx].set_xlabel('Epoch')
        axs[idx].set_ylabel('Loss')
        axs[idx].set_title(f'Latent Dim {latent_dim}')
        axs[idx].legend()
        axs[idx].grid(True)

    plt.tight_layout()
    plt.subplots_adjust(top=0.88)
    plt.show()

plot_all_train_losses_in_row(all_train_losses, latent_dims)
```



Generate new ranbdom samples with PPCA with the model weights obtained.

In [ ]:
```
def generate_samples_with_noise(models, train_loader, num_samples=5):
    num_models = len(models)
    fig, axes = plt.subplots(num_models, num_samples, figsize=(num_samples * 2, num_models * 2))

    for model in models:
        model.eval()

    with torch.no_grad():
        data_means = []
        for data, _ in train_loader:
            data = data.view(data.size(0), -1).to(device)
            data_means.append(data.mean(dim=0))
        mu = torch.stack(data_means).mean(dim=0).to(device)

        for model_idx, model in enumerate(models):
            std_devs = torch.sqrt(torch.exp(model.log_sigma_sq)).to(device)
            sampled_z = torch.randn(num_samples, model.latent_dim).to(device)
            sigma_noise = torch.randn(num_samples, model.data_dim).to(device) * std_devs

            generated_samples = torch.mm(sampled_z, model.W.t()) + mu
            generated_samples_with_noise = generated_samples + sigma_noise

            for i in range(num_samples):
                    axes[model_idx, i].imshow(generated_samples[i].view(28, 28).cpu().numpy(), cmap='gray')
                    axes[model_idx, i].set_title(f'Dim {latent_dims[model_idx]} Generated')
                    axes[model_idx, i].axis('off')

    plt.tight_layout()
    plt.show()

generate_samples_with_noise(trained_models, train_loader, num_samples=5)
```

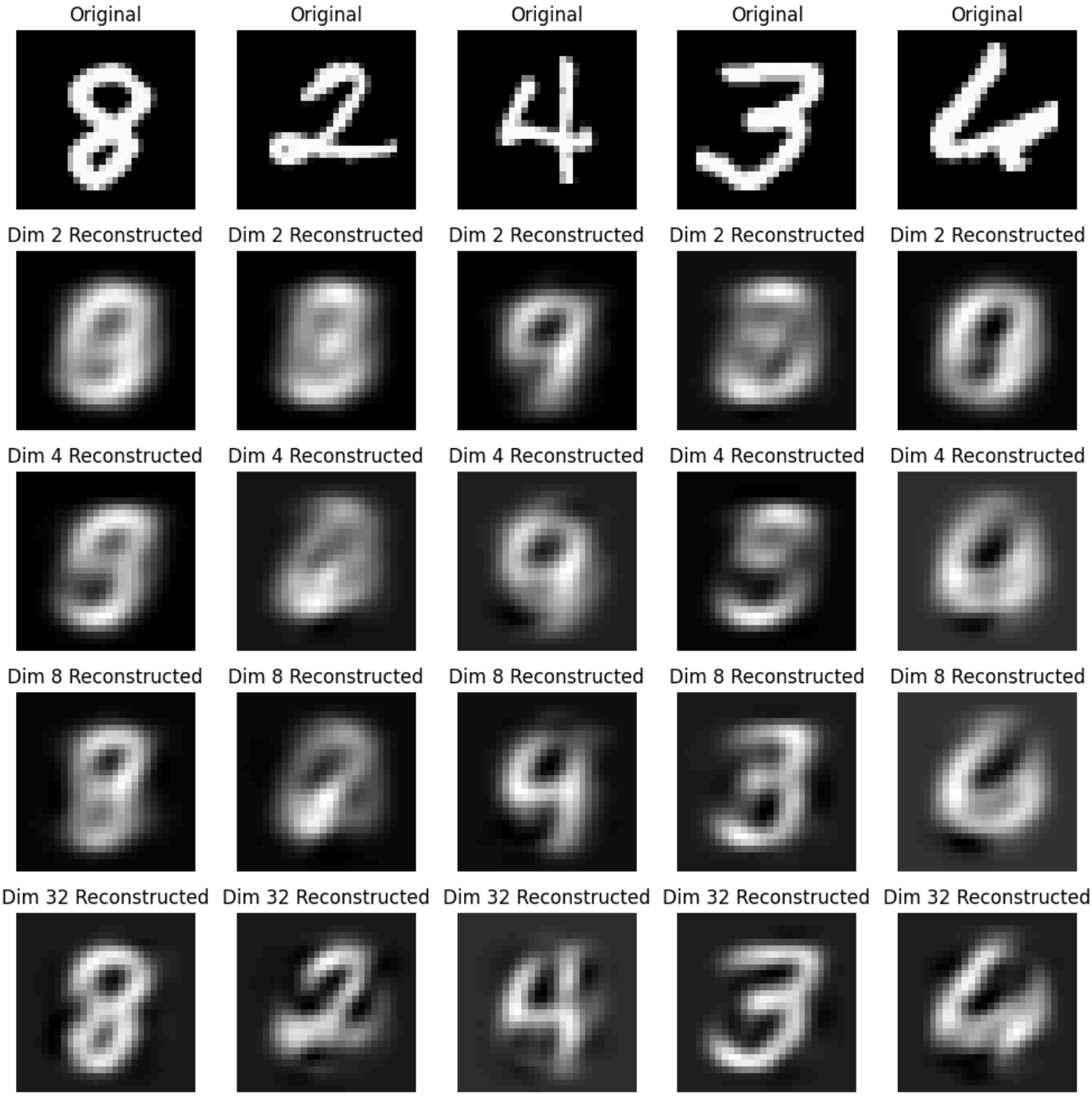


Show the reconstruction effect using the Posterior of Latent Variables.

For showing the correctness of implementation.

In [12]:
```
def visualize_reconstruction(models, test_loader, num_images=5):
    num_models = len(models)
    fig, axes = plt.subplots(num_models + 1, num_images, figsize=(num_images * 2, (num_models + 1) * 2))

    for model in models:
        model.eval()
```

```
    with torch.no_grad():
        data, _ = next(iter(test_loader))
        data = data.view(data.size(0), -1).to(device)

        for i in range(num_images):
            axes[0, i].imshow(data[i].view(28, 28).to('cpu').numpy(), cmap='gray')
            axes[0, i].set_title('Original')
            axes[0, i].axis('off')

        for model_idx, model in enumerate(models):
            x_hat = model(data)
            for i in range(num_images):
                axes[model_idx + 1, i].imshow(x_hat[i].view(28, 28).to('cpu').numpy(), cmap='gray')
                axes[model_idx + 1, i].set_title(f'Dim {latent_dims[model_idx]} Reconstructed')
                axes[model_idx + 1, i].axis('off')

    plt.tight_layout()
    plt.show()

visualize_reconstruction(trained_models, test_loader, num_images=5)
```



Problem 3

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.optim import lr_scheduler
from torch.utils.data import DataLoader
from torchvision import datasets, transforms, models
import matplotlib.pyplot as plt
import numpy as np
import os

device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

Define the AutoEncoder.

In [ ]:
```
class ConvAutoEncoder(nn.Module):
    def __init__(self):
        super(ConvAutoEncoder, self).__init__()

        self.Encoder = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(True),
            nn.Conv2d(32, 48, kernel_size=3, stride=2, padding=1),
            nn.ReLU(True),
            nn.Conv2d(48, 96, kernel_size=3, stride=2, padding=1),
            nn.ReLU(True),
            nn.Conv2d(96, 126, kernel_size=3, stride=2, padding=1),  # 8x8x96 -> 4x4x164
            nn.ReLU(True)
        )
        # After encoder, X encoded from 3x32x32(3072) to 4x4x164(2624).
        self.Decoder = nn.Sequential(
            nn.ConvTranspose2d(126, 96, kernel_size=3, stride=2, padding=1, output_padding=1),
            nn.ReLU(True),
            nn.ConvTranspose2d(96, 48, kernel_size=3, stride=2, padding=1, output_padding=1),
            nn.ReLU(True),
            nn.ConvTranspose2d(48, 32, kernel_size=3, stride=2, padding=1, output_padding=1),
            nn.ReLU(True),
            nn.ConvTranspose2d(32, 3, kernel_size=3, stride=1, padding=1),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.Encoder(x)
        feature = x

        x = self.Decoder(x)

        return x, feature
```

Prepare the Data.

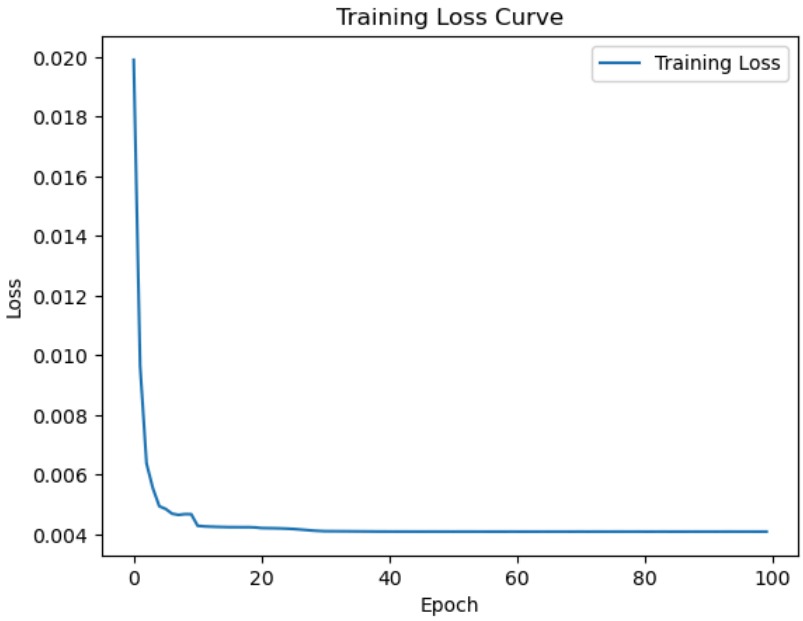
In [4]:
```
batch_size = 128
num_epoch = 100

transform = transforms.Compose([
    transforms.ToTensor(),
])

train_dataset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_dataset = datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
test_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=False)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

Train the Model.

In [5]:
```
trained_model = ConvAutoEncoder().to(device)
criterion  = nn.MSELoss()
min_loss = 10e+10
optimizer = optim.Adam(trained_model.parameters(), lr= 1e-3, weight_decay= 1e-4)
scheduler = lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)

train_loss_list = []

for epoch in range(num_epoch):
    train_loss = 0

    for batch_ind, (image, _) in enumerate(train_loader):
        image = image.to(device)
        optimizer.zero_grad()
        image_out, _ = trained_model(image)
        loss = criterion(image_out, image)
```

```
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    scheduler.step()

    avg_train_loss = train_loss / (batch_ind + 1)
    train_loss_list.append(avg_train_loss)

    if avg_train_loss < min_loss:
        torch.save(trained_model.state_dict(), 'ConvAE_cifar10.pth')
        min_loss = avg_train_loss

plt.plot(range(num_epoch), train_loss_list, label='Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss Curve')
plt.legend()
plt.show()
```



ConvAE_model_random is the model with initial weights and ConvAE_model_trained is the model loaded the trained weights.

In [6]:
```
ConvAE_model_random = ConvAutoEncoder().to(device)
ConvAE_model_trained = ConvAutoEncoder().to(device)
ConvAE_model_trained.load_state_dict(torch.load('ConvAE_cifar10.pth'))
```

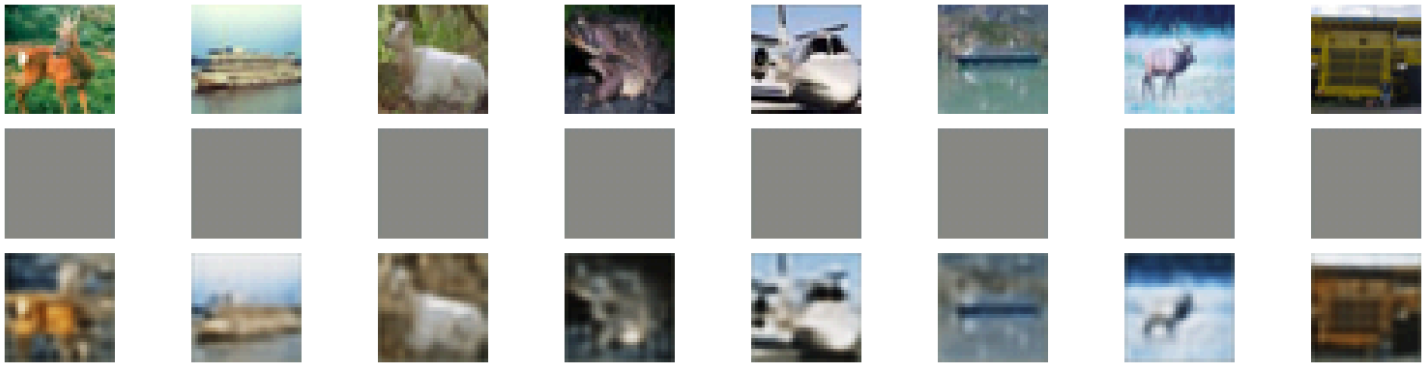
```
C:\Users\14857\AppData\Local\Temp\ipykernel_5520\838969938.py:3: FutureWarning: You are using `torch.load` with `weights_only=False` (the curr
ent default value), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execute arbi
trary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future rel
ease, the default value for `weights_only` will be flipped to `True`. This limits the functions that could be executed during unpickling. Arbi
trary objects will no longer be allowed to be loaded via this mode unless they are explicitly allowlisted by the user via `torch.serializatio
n.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.
  ConvAE_model_trained.load_state_dict(torch.load('ConvAE_cifar10.pth'))
```

Out[6]: <All keys matched successfully>

Show the reconstruction effect.

In [7]:
```
ConvAE_model_trained.eval()
images, _ = next(iter(train_loader))
images = images.to(device)

with torch.no_grad():
    reconstructed1,_ = ConvAE_model_random(images)
    reconstructed2,_ = ConvAE_model_trained(images)

images = images.cpu()
reconstructed1 = reconstructed1.cpu()
reconstructed2 = reconstructed2.cpu()

fig, axes = plt.subplots(3, 8, figsize=(16, 4))

for i in range(8):
    axes[0, i].imshow(images[i].permute(1, 2, 0).numpy())
    axes[0, i].axis('off')

    axes[1, i].imshow(reconstructed1[i].permute(1, 2, 0).numpy())
    axes[1, i].axis('off')

    axes[2, i].imshow(reconstructed2[i].permute(1, 2, 0).numpy())
    axes[2, i].axis('off')
```

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



Instead of Save all the H0 and H1 features, I choose to define the classifier and CombinedModel which combine the autoencoder and classifier. Since I freeze the parameters of the autoencoder, so it is equivalent to save H0/H1 first and then train the classifier.

So the X will firstly be encoded to H0/H1 and then H0/H1 be directly transported to classifier.

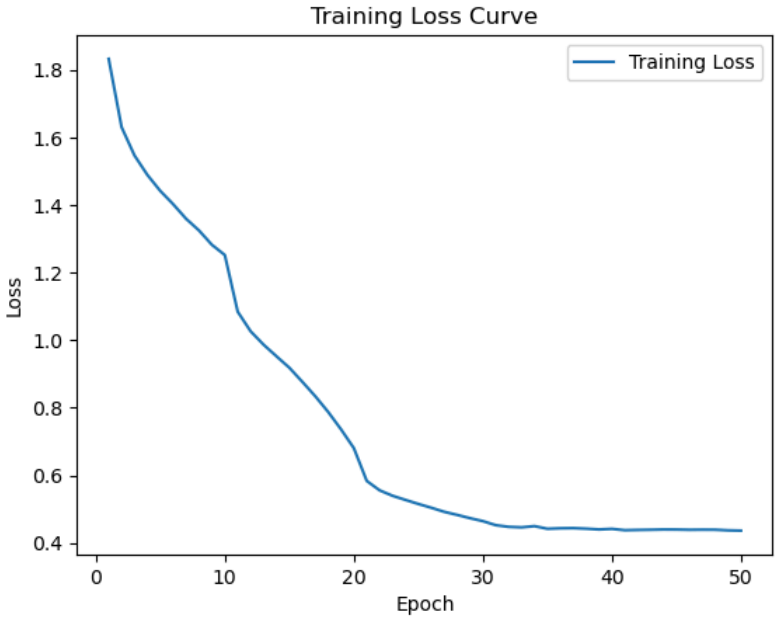
In [ ]:
```
class Classifier(nn.Module):
    def __init__(self):
        super(Classifier, self).__init__()

        self.features = nn.Sequential(
            nn.Conv2d(126, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.01, inplace=True),

            nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.01, inplace=True),

            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.01, inplace=True),

            nn.MaxPool2d(2, stride=2)
        )

        self.classifier = nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)),
            nn.Flatten(),
            nn.Linear(512, 512),
            nn.LeakyReLU(0.01, inplace=True),
            nn.Dropout(0.5),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.01, inplace=True),
            nn.Dropout(0.5),
            nn.Linear(256, 10)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x
```

In [ ]:
```
class CombinedModel(nn.Module):
    def __init__(self, autoencoder):
        super(CombinedModel, self).__init__()
        self.autoencoder = autoencoder
        for param in self.autoencoder.parameters():
            param.requires_grad = False
        self.classifier = Classifier()

    def forward(self, x):
        _, feature = self.autoencoder(x)
        output = self.classifier(feature)
        return output
```

Train the model with H0 features.

In [14]:
```
model_H0 = CombinedModel(ConvAE_model_random).to(device)
criterion = nn.CrossEntropyLoss()
min_loss = float('inf')
optimizer = optim.Adam(model_H0.classifier.parameters(), lr=1e-3, weight_decay=1e-4)
scheduler = lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)

train_loss_list = []

for epoch in range(50):
    model_H0.train()
```

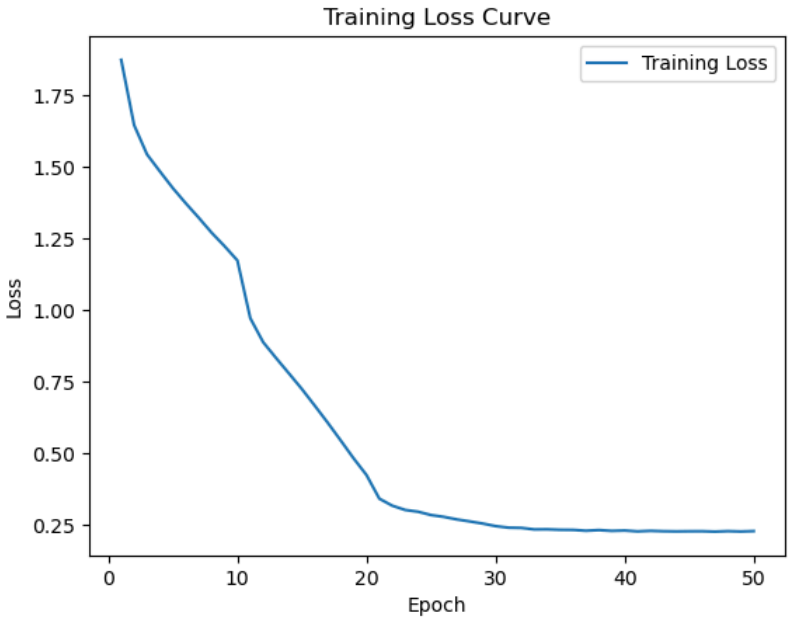
```
    train_loss = 0

    for batch_ind, (image, label) in enumerate(train_loader):
        image = image.to(device)
        label = label.long().to(device)

        optimizer.zero_grad()
        y_hat = model_H0(image)
        loss = criterion(y_hat, label)

        loss.backward()
        optimizer.step()

        train_loss += loss.item()

    scheduler.step()

    avg_train_loss = train_loss / (batch_ind + 1)
    train_loss_list.append(avg_train_loss)

    if avg_train_loss < min_loss:
        torch.save(model_H0.state_dict(), 'random_model.pth')
        min_loss = avg_train_loss

plt.plot(range(1, 51), train_loss_list, label='Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss Curve')
plt.legend()
plt.show()
```



Train the model with H1 features.

In [15]:
```
model_H1 = CombinedModel(ConvAE_model_trained).to(device)
criterion = nn.CrossEntropyLoss()
min_loss = float('inf')
optimizer = optim.Adam(model_H1.classifier.parameters(), lr=1e-3, weight_decay=1e-4)
scheduler = lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)

train_loss_list = []

for epoch in range(50):
    model_H1.train()
    train_loss = 0

    for batch_ind, (image, label) in enumerate(train_loader):
        image = image.to(device)
        label = label.long().to(device)

        optimizer.zero_grad()
        y_hat = model_H1(image)
        loss = criterion(y_hat, label)

        loss.backward()
        optimizer.step()

        train_loss += loss.item()

    scheduler.step()

    avg_train_loss = train_loss / (batch_ind + 1)
    train_loss_list.append(avg_train_loss)

    if avg_train_loss < min_loss:
        torch.save(model_H1.state_dict(), 'trained_model.pth')
```

```
        min_loss = avg_train_loss

plt.plot(range(1, 51), train_loss_list, label='Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss Curve')
plt.legend()
plt.show()
```



In [16]:
```
clf_0 = CombinedModel(ConvAE_model_random)
clf_0.load_state_dict(torch.load('random_model.pth'))
clf_1 = CombinedModel(ConvAE_model_trained)
clf_1.load_state_dict(torch.load('trained_model.pth'))
clf_0.eval()
clf_0.to(device)
clf_1.eval()
clf_1.to(device)
```

```
C:\Users\14857\AppData\Local\Temp\ipykernel_5520\3682011334.py:2: FutureWarning: You are using `torch.load` with `weights_only=False` (the cur
rent default value), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execute arb
itrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future re
lease, the default value for `weights_only` will be flipped to `True`. This limits the functions that could be executed during unpickling. Arb
itrary objects will no longer be allowed to be loaded via this mode unless they are explicitly allowlisted by the user via `torch.serializatio
n.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.
  clf_0.load_state_dict(torch.load('random_model.pth'))
C:\Users\14857\AppData\Local\Temp\ipykernel_5520\3682011334.py:4: FutureWarning: You are using `torch.load` with `weights_only=False` (the cur
rent default value), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execute arb
itrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future re
lease, the default value for `weights_only` will be flipped to `True`. This limits the functions that could be executed during unpickling. Arb
itrary objects will no longer be allowed to be loaded via this mode unless they are explicitly allowlisted by the user via `torch.serializatio
n.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.
  clf_1.load_state_dict(torch.load('trained_model.pth'))
```

Out[16]:
```
CombinedModel(
  (autoencoder): ConvAutoEncoder(
    (Encoder): Sequential(
      (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU(inplace=True)
      (2): Conv2d(32, 48, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (3): ReLU(inplace=True)
      (4): Conv2d(48, 96, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (5): ReLU(inplace=True)
      (6): Conv2d(96, 126, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (7): ReLU(inplace=True)
    )
    (Decoder): Sequential(
      (0): ConvTranspose2d(126, 96, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
      (1): ReLU(inplace=True)
      (2): ConvTranspose2d(96, 48, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
      (3): ReLU(inplace=True)
      (4): ConvTranspose2d(48, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
      (5): ReLU(inplace=True)
      (6): ConvTranspose2d(32, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (7): Sigmoid()
    )
  )
  (classifier): ClassifierFromFeature(
    (features): Sequential(
      (0): Conv2d(126, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.01, inplace=True)
      (3): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): LeakyReLU(negative_slope=0.01, inplace=True)
      (6): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (7): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (8): LeakyReLU(negative_slope=0.01, inplace=True)
      (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
    (classifier): Sequential(
      (0): AdaptiveAvgPool2d(output_size=(1, 1))
      (1): Flatten(start_dim=1, end_dim=-1)
      (2): Linear(in_features=512, out_features=512, bias=True)
      (3): LeakyReLU(negative_slope=0.01, inplace=True)
      (4): Dropout(p=0.5, inplace=False)
      (5): Linear(in_features=512, out_features=256, bias=True)
      (6): LeakyReLU(negative_slope=0.01, inplace=True)
      (7): Dropout(p=0.5, inplace=False)
      (8): Linear(in_features=256, out_features=10, bias=True)
    )
  )
)
```

Test the performance of two models.

In [17]:
```python
def test_clf(model, device, data, target):
    model.eval()
    correct = 0
    with torch.no_grad():
        data, target = data.to(device), target.to(device)
        output = model(data)
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()
    return 100. * correct / len(data)
```

In [18]:
```python
iter_test = iter(test_loader)
image_test, label_test = next(iter_test)
image_test = image_test.to(device)
acc_random = test_clf(clf_0, device, image_test, label_test)
acc_trained = test_clf(clf_1, device, image_test, label_test)
print('Classification accuracy for Model with H0 as input: {}'.format(acc_random))
print('Classification accuracy for Model with H1 as input: {}'.format(acc_trained))
```

```
Classification accuracy for Model with H0 as input: 87.5
Classification accuracy for Model with H1 as input: 96.875
```

The results show that the model trained with the extracted features H1 achieves a classification accuracy of 96.875%, which is higher than the training accuracy of 87.5% obtained with the randomly mapped features H0. The key difference here is that H0 is a random mapping of the image; thus, while it retains some image information, it may capture more non-essential information and lose important details. In contrast, H1, although used for image reconstruction and subject to some information loss, is more likely to retain the more critical aspects of the image. Therefore, in terms of the importance of extracted information, H1 is superior to H0, leading to better performance when using H1 for classification.

Problem 1: GAN

In [1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np
from torch.utils.data import DataLoader, SubsetRandomSampler

device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

1.1 Implement a Deep Convolutional GAN (DCGAN)

Prepare the data.

In [2]:
```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5]),
])

batch_size = 1000
train_data = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)
test_data = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
test_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=False)
```

In [3]:
```
latent_dim = 100
img_shape = (1, 28, 28)
lr = 0.0002
n_epochs = 400
```

Define the generator.

In [4]:
```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.ConvTranspose2d(latent_dim, 256, kernel_size=7, stride=1, padding=0),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.Dropout(0.3),

            nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Dropout(0.3),

            nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Dropout(0.3),

            nn.Conv2d(64, 32, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.Dropout(0.3),

            nn.Conv2d(32, 1, kernel_size=3, stride=1, padding=1),
            nn.Tanh()
        )

    def forward(self, z):
        z = z.view(z.size(0), latent_dim, 1, 1)
        img = self.model(z)
        return img
```

Define the discriminator.

In [5]:
```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),

            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
```

```
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),

            nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),

            nn.AdaptiveAvgPool2d((4, 4))
        )

        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(512 * 4 * 4, 1),
            nn.Sigmoid()
        )

    def forward(self, img):
        features = self.model(img)
        validity = self.fc(features)
        return validity
```

In [6]:
```
generator = Generator().to(device)
discriminator = Discriminator().to(device)

adversarial_loss = nn.BCELoss()
optimizer_G = optim.Adam(generator.parameters(), lr=lr, betas=(0.5, 0.999))
optimizer_D = optim.Adam(discriminator.parameters(), lr=lr, betas=(0.5, 0.999))
```

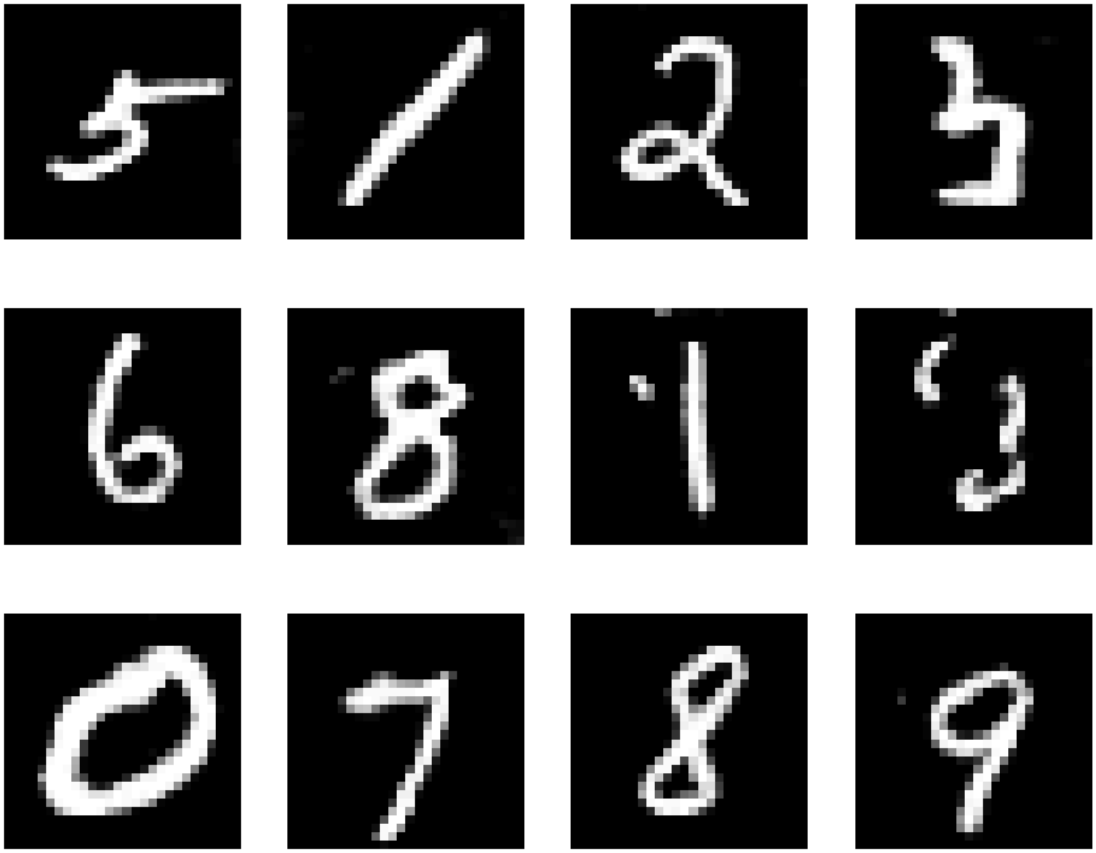
Train the model.

In [7]:
```
for epoch in range(n_epochs):
    for i, (imgs, _) in enumerate(train_loader):
        real_imgs = imgs.to(device)
        batch_size = real_imgs.size(0)

        valid = torch.ones(batch_size, 1, device=device)
        fake = torch.zeros(batch_size, 1, device=device)

        optimizer_G.zero_grad()
        z = torch.randn(batch_size, latent_dim, device=device)
        gen_imgs = generator(z)
        g_loss = adversarial_loss(discriminator(gen_imgs), valid)
        g_loss.backward()
        optimizer_G.step()

        optimizer_D.zero_grad()
        real_loss = adversarial_loss(discriminator(real_imgs), valid)
        fake_loss = adversarial_loss(discriminator(gen_imgs.detach()), fake)
        d_loss = (real_loss + fake_loss) / 2
        d_loss.backward()
        optimizer_D.step()

    if (epoch + 1) % 40 == 0:
        print(f"[Epoch {epoch + 1}/{n_epochs}] [D loss: {d_loss.item():.4f}] [G loss: {g_loss.item():.4f}]")

torch.save(generator.state_dict(), "generator.pth")
torch.save(discriminator.state_dict(), "discriminator.pth")
```

```
/root/miniconda3/lib/python3.12/site-packages/torch/nn/modules/conv.py:456: UserWarning: Plan failed with a cudnnException: CUDNN_BACKEND_EXECUTION_PLAN_DESCRIPTOR: cudnnFinalize Descriptor Failed cudnn_status: CUDNN_STATUS_NOT_SUPPORTED (Triggered internally at ../aten/src/ATen/native/cudnn/Conv_v8.cpp:919.)
  return F.conv2d(input, weight, bias, self.stride,
/root/miniconda3/lib/python3.12/site-packages/torch/autograd/graph.py:744: UserWarning: Plan failed with a cudnnException: CUDNN_BACKEND_EXECUTION_PLAN_DESCRIPTOR: cudnnFinalize Descriptor Failed cudnn_status: CUDNN_STATUS_NOT_SUPPORTED (Triggered internally at ../aten/src/ATen/native/cudnn/Conv_v8.cpp:919.)
  return Variable._execution_engine.run_backward(  # Calls into the C++ engine to run the backward pass
[Epoch 40/400] [D loss: 0.4796] [G loss: 1.7630]
[Epoch 80/400] [D loss: 0.3972] [G loss: 1.2897]
[Epoch 120/400] [D loss: 0.5719] [G loss: 0.7195]
[Epoch 160/400] [D loss: 0.2084] [G loss: 2.9183]
[Epoch 200/400] [D loss: 0.2013] [G loss: 2.9095]
[Epoch 240/400] [D loss: 0.1873] [G loss: 1.6038]
[Epoch 280/400] [D loss: 0.5124] [G loss: 1.0463]
[Epoch 320/400] [D loss: 2.7404] [G loss: 0.0113]
[Epoch 360/400] [D loss: 0.0579] [G loss: 3.8257]
[Epoch 400/400] [D loss: 0.1779] [G loss: 4.9219]
```

Utilize the generator to produce 12 new samples

In [8]:
```
z = torch.randn(12, latent_dim, device=device)
gen_imgs = generator(z).cpu().detach()

fig, axs = plt.subplots(3, 4, figsize=(10, 8))
for i, ax in enumerate(axs.flatten()):
    ax.imshow(gen_imgs[i].squeeze(), cmap='gray')
    ax.axis('off')
plt.show()
```



1.2 GAN as a pre-training framework

Removing the final linear layer of your discriminator to form a feature extractor.

In [9]:
```
class FeatureExtractor(nn.Module):
    def __init__(self, discriminator):
        super(FeatureExtractor, self).__init__()
        self.features = nn.Sequential(*list(discriminator.model.children()))

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

discriminator = Discriminator().to(device)
discriminator.load_state_dict(torch.load("discriminator.pth"))
feature_extractor = FeatureExtractor(discriminator).to(device)
feature_extractor.eval()
```

Out[9]:
```
FeatureExtractor(
  (features): Sequential(
    (0): Conv2d(1, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace=True)
    (3): Dropout(p=0.3, inplace=False)
    (4): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): LeakyReLU(negative_slope=0.2, inplace=True)
    (7): Dropout(p=0.3, inplace=False)
    (8): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (9): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Dropout(p=0.3, inplace=False)
    (12): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (14): LeakyReLU(negative_slope=0.2, inplace=True)
    (15): Dropout(p=0.3, inplace=False)
    (16): AdaptiveAvgPool2d(output_size=(4, 4))
  )
)
```

Define single linear layer classifier.

In [10]:
```
class LinearClassifier(nn.Module):
    def __init__(self, input_dim, num_classes=10):
        super(LinearClassifier, self).__init__()
        self.classifier = nn.Linear(input_dim, num_classes)

    def forward(self, x):
        return self.classifier(x)
```

10% of the training set images.

In [11]:
```
dataset_size = len(train_loader.dataset)
indices = np.random.choice(dataset_size, size=int(0.1 * dataset_size), replace=False)
sampler = SubsetRandomSampler(indices)
subset_loader = DataLoader(train_loader.dataset, batch_size=64, sampler=sampler)
```

In [12]:
```
data, labels = [], []
for imgs, lbls in subset_loader:
    features = feature_extractor(imgs.to(device)).detach().cpu()
    data.append(features)
    labels.append(lbls)

train_features = torch.cat(data)
train_labels = torch.cat(labels)
```

Train the simple classifier.

In [13]:
```
input_dim = 512 * 4 * 4
classifier = LinearClassifier(input_dim).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(classifier.parameters(), lr=0.001)

n_epochs = 1000
for epoch in range(n_epochs):
    classifier.train()
    optimizer.zero_grad()
    outputs = classifier(train_features.to(device))
    loss = criterion(outputs, train_labels.to(device))
    loss.backward()
    optimizer.step()
    if (epoch + 1) % 100 == 0:
        print(f"Epoch [{epoch+1}/{n_epochs}], Loss: {loss.item():.4f}")
```

```
Epoch [100/1000], Loss: 0.1285
Epoch [200/1000], Loss: 0.0680
Epoch [300/1000], Loss: 0.0433
Epoch [400/1000], Loss: 0.0302
Epoch [500/1000], Loss: 0.0223
Epoch [600/1000], Loss: 0.0171
Epoch [700/1000], Loss: 0.0136
Epoch [800/1000], Loss: 0.0111
Epoch [900/1000], Loss: 0.0093
Epoch [1000/1000], Loss: 0.0079
```

Report both the training and testing performance

In [14]:
```
def evaluate(classifier, loader, feature_extractor):
    classifier.eval()
    correct, total = 0, 0
    with torch.no_grad():
        for imgs, labels in loader:
            features = feature_extractor(imgs.to(device)).detach()
            outputs = classifier(features)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted.cpu() == labels).sum().item()
    accuracy = 100 * correct / total
    return accuracy

train_accuracy = evaluate(classifier, subset_loader, feature_extractor)
print(f"Train Set Accuracy: {train_accuracy:.2f}%")
test_accuracy = evaluate(classifier, test_loader, feature_extractor)
print(f"Test Set Accuracy: {test_accuracy:.2f}%")
```

```
Train Set Accuracy: 100.00%
Test Set Accuracy: 97.16%
```

The results demonstrate that the GAN's discriminator effectively captures meaningful features from the images, enabling the linear classifier to achieve excellent performance with the high training accuracy (100%) and strong test accuracy (97.16%).

Training a GAN discriminator to differentiate between real and fake images helps it learn detailed and meaningful features of the data distribution. Even the discriminator's task requires distinguishing real data from the generated data which is not same as classifier, it focuses on high-level representations and subtle patterns in the input images.

Problem 2.1

part 1

Given the probability $P(v_i|h)$ for the Gaussian-Bernoulli Restricted Boltzmann Machine (GBRBM):

$$P(v_i|h) = P(v_i|v_{-i}, h) = \frac{P(v_i, v_{-i}, h)}{P(v_{-i}, h)},$$

we can write it as:

$$P(v_i|h) = \frac{\exp(-E(v,h))/Z}{\int_{v_i} \exp(-E(v|h))/Z \; dv_i},$$

where $E(v,h)$ is the energy function, and Z is the partition function.

After simplifying the expression, we have:

$$P(v_i|h) = \frac{\exp\left(\sum_k \sum_j \frac{v_k}{\sigma_k} w_{kj} h_j + \sum_j a_j h_j - \sum_k \frac{(v_k - b_k)^2}{2\sigma_k^2}\right)}{\int_{v_i} \exp\left(\sum_k \sum_j \frac{v_k}{\sigma_k} w_{kj} h_j + \sum_j a_j h_j - \sum_k \frac{(v_k - b_k)^2}{2\sigma_k^2}\right) dv_i}.$$

Define:

$$\hat{A} = \sum_{k \neq i} \sum_j \frac{v_k}{\sigma_k} w_{kj} h_j + \sum_j a_j h_j - \sum_{k \neq i} \frac{(v_k - b_k)^2}{2\sigma_k^2}.$$

Thus, we can rewrite the denominator as:

$$\exp(\hat{A}) \int_{v_i} \exp\left(\frac{v_i}{\sigma_i} \sum W_{ij} h_j - \frac{(v_i - b_i)^2}{2\sigma_i^2}\right) dv_i.$$

The numerator becomes:

$$\exp(\hat{A}) \cdot \exp\left(\frac{v_i}{\sigma_i} \sum W_{ij} h_j - \frac{(v_i - b_i)^2}{2\sigma_i^2}\right).$$

Therefore:

$$P(v_i|h) = \frac{\exp\left(\frac{v_i}{\sigma_i} \sum W_{ij} h_j - \frac{(v_i - b_i)^2}{2\sigma_i^2}\right)}{\int_{v_i} \exp\left(\frac{v_i}{\sigma_i} \sum W_{ij} h_j - \frac{(v_i - b_i)^2}{2\sigma_i^2}\right) dv_i}.$$

And we have:

$$\frac{v_i}{\sigma_i} \sum W_{ij} h_j - \frac{(v_i - b_i)^2}{2\sigma_i^2} = \frac{-[v_i - (\sigma_i \sum_j W_{ij} h_j + b_i)]}{2\sigma_i^2} + [\frac{(\sigma_i \sum_j W_{ij} h_j + b_i)^2}{2\sigma_i^2} - \frac{b_i^2}{2\sigma_i^2}]$$

Define:

$$B = [\frac{(\sigma_i \sum_j W_{ij} h_j + b_i)^2}{2\sigma_i^2} - \frac{b_i^2}{2\sigma_i^2}]$$

And define:

$$\mu = \sigma_i \sum_j W_{ij} h_j + b_i.$$

So:

$$P(v_i|h) = \frac{\exp(B) \exp\left(-\frac{(v_i - \mu)^2}{2\sigma_i^2}\right)}{\exp(B) \int_{v_i} \exp\left(-\frac{(v_i - \mu)^2}{2\sigma_i^2}\right) dv_i} = \frac{\exp\left(-\frac{(v_i - \mu)^2}{2\sigma_i^2}\right)}{\int_{v_i} \exp\left(-\frac{(v_i - \mu)^2}{2\sigma_i^2}\right) dv_i}$$

And we have:

$$\int_{v_i} \exp\left(-\frac{(v_i - \mu)^2}{2\sigma_i^2}\right) dv_i = \sqrt{2\pi}\sigma_i,$$

Further simplifying, we have:

$$P(v_i|h) = \frac{\exp\left(-\frac{(v_i - \mu)^2}{2\sigma_i^2}\right)}{\sqrt{2\pi}\sigma_i},$$

Thus:

$$P(v_i|h) = \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{1}{2\sigma_i^2}\left[v_i - (b_i + \sigma_i \sum_j W_{ij} h_j)\right]^2\right),$$

which is a Gaussian distribution:

$$v_i|h \sim \mathcal{N}(b_i + \sigma_i \sum_j W_{ij} h_j, \sigma_i^2).$$

part 2

The conditional probability $P(h_j|v)$ can be expressed as:

$$P(h_j|v) = P(h_j|h_{-j}, v) = \frac{P(h_j, h_{-j}, v)}{P(h_j = 1, h_{-j}, v) + P(h_j = 0, h_{-j}, v)}$$

The energy function $E(v, h)$ is given by:

$$E(v,h) = -\left(h_j \sum_i W_{ij} \frac{v_i}{\sigma_i} + \alpha_j h_j + \sum_{k \neq j} \sum_i W_{ik} \frac{v_i}{\sigma_i} h_k + \sum_{k \neq j} \alpha_k h_k - \sum_i \frac{(v_i - b_i)^2}{2\sigma_i^2}\right)$$

Define:

$$A = \sum_{k \neq j} \sum_i W_{ik} \frac{v_i}{\sigma_i} h_k + \sum_{k \neq j} \alpha_k h_k - \sum_i \frac{(v_i - b_i)^2}{2\sigma_i^2}$$

Thus, we have:

$$P(h_j|v) = \frac{\exp(A) \cdot \exp\left(h_j \sum_i W_{ij} \frac{v_i}{\sigma_i} + \alpha_j h_j\right)/Z}{\exp(A) \cdot \exp\left(\sum_i W_{ij} \frac{v_i}{\sigma_i} + \alpha_j\right)/Z + \exp(A) \cdot \exp(0)/Z}$$

Simplifying further:

$$P(h_j|v) = \frac{\exp\left(h_j \sum_i W_{ij} \frac{v_i}{\sigma_i} + \alpha_j\right)}{1 + \exp\left(\sum_i W_{ij} \frac{v_i}{\sigma_i} + \alpha_j\right)}$$

Thus:

$$P(h_j = 1|v) = \frac{\exp\left(\sum_i W_{ij} \frac{v_i}{\sigma_i} + \alpha_j\right)}{1 + \exp\left(\sum_i W_{ij} \frac{v_i}{\sigma_i} + \alpha_j\right)} = sigmoid\left(\sum_i W_{ij} \frac{v_i}{\sigma_i} + \alpha_j\right)$$

Problem 2: Gaussian-Bernoulli Restricted Boltzmann Machines

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim
import torch.optim as optim

import torchvision
import torchvision.transforms as transforms
from torchvision.utils import make_grid

import matplotlib.pyplot as plt
import numpy as np
```

2.2 GB-RBM on KMNIST

Prepare the data.

In [2]:
```
device = 'cuda'
batch_size = 128
transform = transforms.Compose(
    [transforms.ToTensor(),
    ])

train_set = torchvision.datasets.KMNIST(root='./data', train=True, download=True, transform=transform)
test_set = torchvision.datasets.KMNIST(root='./data', train=False, download=True, transform=transform)

train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, shuffle=True, num_workers=0)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size, shuffle=False, num_workers=0)
```

Define the model. Since we will normalize the data, so set σ=1.

In [3]:
```
class RBM(nn.Module):
    """Restricted Boltzmann Machine template."""

    def __init__(self, D: int, F: int, k: int):
        """Creates an instance RBM module.
        Args:
            D: Size of the input data.
            F: Size of the hidden variable.
            k: Number of MCMC iterations for negative sampling.
        The function initializes the weight (W) and biases (c & b).
        """
        super().__init__()
        self.W = nn.Parameter(torch.randn(F, D) * 1e-2)
        self.c = nn.Parameter(torch.zeros(D))
        self.b = nn.Parameter(torch.zeros(F))
        self.k = k

    def sample(self, p):
        """Sample from a Bernoulli distribution defined by a given parameter."""
        return torch.bernoulli(p)

    def sample_gaussian(self, p):
        """Sample from a Gaussian distribution defined by a given parameter."""
        return p + torch.randn_like(p)

    def P_h_x(self, x):
        """Returns the conditional P(h|x)."""
        return torch.sigmoid(F.linear(x, self.W, self.b))

    def P_x_h(self, h):
        """Returns the conditional P(x|h)."""
        return F.linear(h, self.W.t(), self.c)

    def free_energy(self, x):
        """Returns the Average F(x) free energy. (Slide 11, Lecture 14)."""
        vbias_term = (-(x-self.c)*(x-self.c)/2).sum(dim=1)
        wv_b = F.linear(x, self.W, self.b)
        hidden_term = F.softplus(wv_b).sum(dim=1)
        return (-hidden_term - vbias_term).mean()

    def forward(self, v):
        """Performs forward propagation through the model."""
        v_sampled = v
        for _ in range(self.k):
            p_h_given_v = self.P_h_x(v_sampled)
            h_sampled = self.sample(p_h_given_v)

            p_v_given_h = self.P_x_h(h_sampled)
            v_sampled = self.sample_gaussian(p_v_given_h)

            p_h_given_v = self.P_h_x(v_sampled)
            h_sampled = self.sample(p_h_given_v)
```

```
        return v_sampled, p_v_given_h
```

Define the train process.

In [4]:
```
def train(model, device, train_loader, optimizer, epoch):
    model.train()
    train_loss = 0
    for data, _ in train_loader:
        data = data.view(data.size(0), -1).to(device)

        # Normalizing data
        mean, std = data.mean(), data.std()
        data = (data - mean) / std

        v_sampled, p_v_given_h = model(data)

        optimizer.zero_grad()
        loss = model.free_energy(data) - model.free_energy(v_sampled)
        loss.backward()
        optimizer.step()

        train_loss += loss.item()

    train_loss /= len(train_loader.dataset)
    if epoch%5==0:
        print('Train Epoch: {} \tLoss: {:.6f}'.format(epoch, train_loss))
```

Train the 3 models with latent dim = [16, 64, 256].

In [5]:
```
seed = 42
num_epochs = 25
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
M = [16, 64, 256]
models = []
optimizers = []

for i in range(3):
    rbm = RBM(D=28 * 28 * 1, F=M[i], k=5).to(device)
    optimizer = optim.Adam(rbm.parameters(), lr=1e-3)
    models.append(rbm)
    optimizers.append(optimizer)

for i, (rbm, optimizer) in enumerate(zip(models, optimizers)):
    print("Training model {} with M = {}:".format(i + 1, M[i]))
    for epoch in range(1, num_epochs + 1):
        train(rbm, device, train_loader, optimizer, epoch)
```

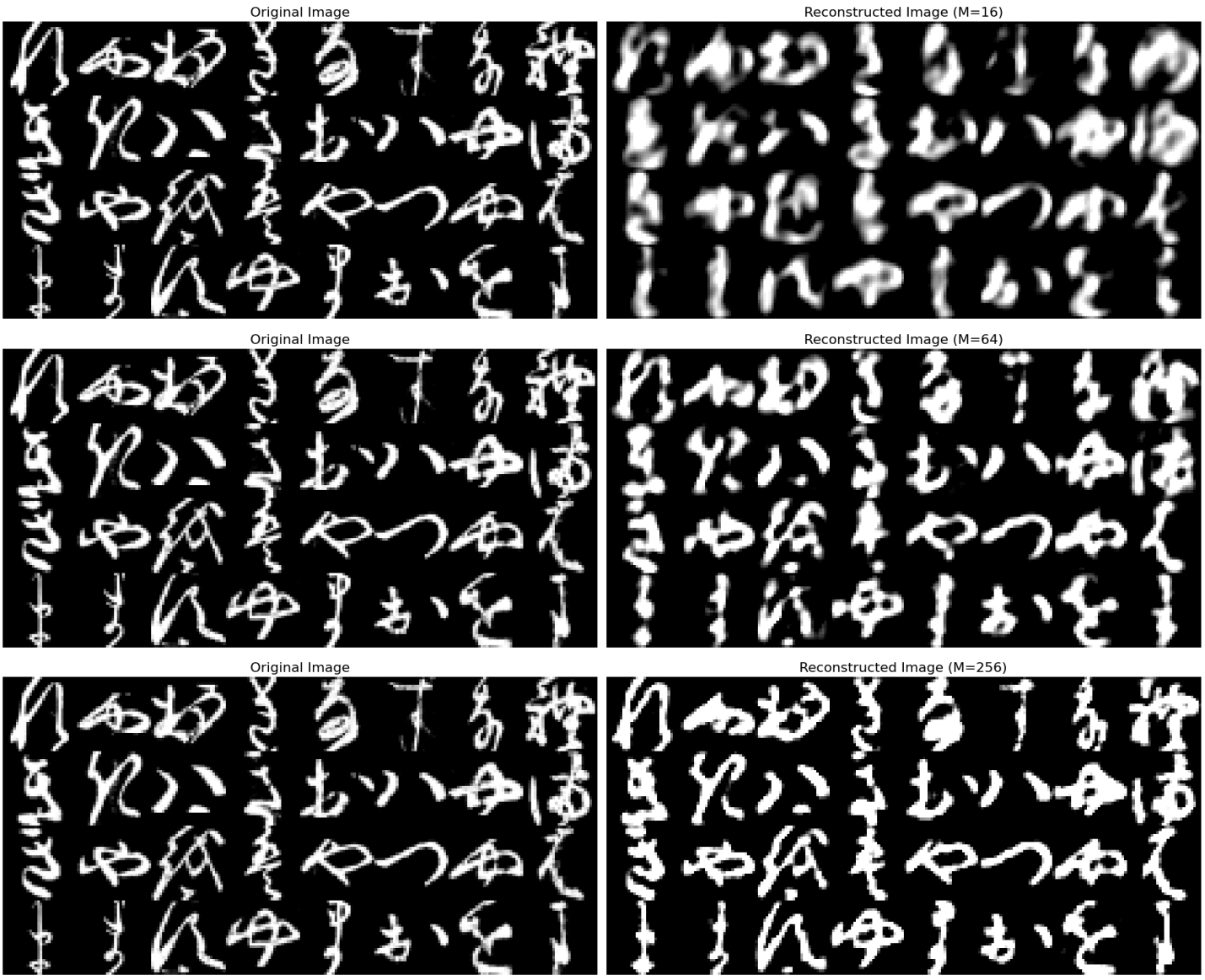
```
Training model 1 with M = 16:
Train Epoch: 5  Loss: -1.125684
Train Epoch: 10         Loss: -1.172729
Train Epoch: 15         Loss: -1.188537
Train Epoch: 20         Loss: -1.197791
Train Epoch: 25         Loss: -1.205249
Training model 2 with M = 64:
Train Epoch: 5  Loss: -1.674580
Train Epoch: 10         Loss: -1.767623
Train Epoch: 15         Loss: -1.812097
Train Epoch: 20         Loss: -1.843711
Train Epoch: 25         Loss: -1.866311
Training model 3 with M = 256:
Train Epoch: 5  Loss: -2.171796
Train Epoch: 10         Loss: -2.285825
Train Epoch: 15         Loss: -2.363311
Train Epoch: 20         Loss: -2.421793
Train Epoch: 25         Loss: -2.465338
```

Define the plot function.

In [6]:
```
def show(img1, img2, M):
    npimg1 = img1.cpu().numpy()
    npimg2 = img2.cpu().numpy()

    fig, axes = plt.subplots(1, 2, figsize=(20, 10))

    axes[0].imshow(np.transpose(npimg1, (1, 2, 0)), interpolation='nearest')
    axes[0].set_title("Original Image", fontsize=16)
    axes[0].axis('off')

    axes[1].imshow(np.transpose(npimg2, (1, 2, 0)), interpolation='nearest')
    axes[1].set_title(f"Reconstructed Image (M={M})", fontsize=16)
    axes[1].axis('off')

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

Show the images in the train set.

In [7]:
```
train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, shuffle=False, num_workers=0)
for i in range(3):
    rbm = models[i]
    data, _ = next(iter(train_loader))
    data = data[:32]
    data_size = data.size()
    data = data.view(data.size(0), -1).to(device)
    mean, std = data.mean(), data.std()
    bdata = (data - mean) / std
    vh_x = rbm.P_h_x(bdata)
    vx_h = rbm.P_x_h(vh_x)
    vx_h = vx_h.detach()
    show(make_grid(data.reshape(data_size), padding=0), make_grid(vx_h.reshape(data_size).clip(min=0,max=1), padding=0), M[i])
    plt.show()
```



Show the images in the test set.

In [8]:
```
for i in range(3):
    rbm = models[i]
    data, _ = next(iter(test_loader))
    data = data[:32]
    data_size = data.size()
    data = data.view(data.size(0), -1).to(device)
    mean, std = data.mean(), data.std()
    bdata = (data - mean) / std
    vh_x = rbm.P_h_x(bdata)
    vx_h = rbm.P_x_h(vh_x)
    vx_h = vx_h.detach()
    show(make_grid(data.reshape(data_size), padding=0), make_grid(vx_h.reshape(data_size).clip(min=0,max=1), padding=0), M[i])
    plt.show()
```

Compute and report the mean squared reconstruction error (MSE) for inferences.

In [13]:
```
def test(model, device, data_loader):
    model.eval()
    total_loss = 0
    total_samples = 0

    mse_loss = nn.MSELoss()

    with torch.no_grad():
        for data, _ in data_loader:
            data = data.view(data.size(0), -1).to(device)

            mean, std = data.mean(), data.std()
            bdata = (data - mean) / std
            vh_x = model.P_h_x(bdata)
            vx_h = model.P_x_h(vh_x)

            loss = mse_loss(vx_h, bdata)
            total_loss += loss.item() * data.size(0)
            total_samples += data.size(0)

    average_loss = total_loss / total_samples
    return average_loss
```

In [14]:
```
for i in range(3):
    model = models[i]
    print('model {} with M = {}'.format(i + 1, M[i]))
    average_loss = test(model, device, train_loader)
    print('Average MSE Loss in Train Set: {:.4f}'.format(average_loss))
    average_loss = test(model, device, test_loader)
    print('Average MSE Loss in Test Set: {:.4f}'.format(average_loss))
```

```
model 1 with M = 16
Average MSE Loss in Train Set: 0.5964
Average MSE Loss in Test Set: 0.6304
model 2 with M = 64
Average MSE Loss in Train Set: 0.3595
Average MSE Loss in Test Set: 0.3900
model 3 with M = 256
Average MSE Loss in Train Set: 0.1446
Average MSE Loss in Test Set: 0.1637
```

Problem 3: Two Variational Autoencoders

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim
import torch.optim as optim

import torchvision
import torchvision.transforms as transforms
from torchvision.utils import make_grid
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, random_split

import matplotlib.pyplot as plt
import numpy as np
from sklearn.manifold import TSNE

device = 'cuda'
```

3.1 Vanilla VAE and 3.2 C-VAE

Define the VAE, c is the conditional information and in Vanilla VAE, just put c=0.

In [ ]:
```
class VAE(nn.Module):
    def __init__(self, n_in, n_hid, z_dim):
        super().__init__()

        self.fc1 = nn.Linear(n_in + 10, n_hid)
        self.fc21 = nn.Linear(n_hid, z_dim)
        self.fc22 = nn.Linear(n_hid, z_dim)

        self.fc3 = nn.Linear(z_dim + 10, n_hid)
        self.fc4 = nn.Linear(n_hid, n_in)

    def encode(self, x, c):
        """Encoder forward pass."""
        x = torch.cat((x, c), dim=1)
        h1 = F.relu(self.fc1(x))
        mu = self.fc21(h1)
        logvar = self.fc22(h1)
        return mu, logvar

    def reparameterize(self, mu, logvar):
        """Implements: z = mu + epsilon * stdev."""
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        z = mu + eps * std
        return z

    def decode(self, z, c):
        """Decoder forward pass."""
        z = torch.cat((z, c), dim=1)
        h3 = F.relu(self.fc3(z))
        recon_x = torch.sigmoid(self.fc4(h3))
        return recon_x

    def forward(self, x, c):
        """Defines the complete forward pass of the VAE."""
        mu, logvar = self.encode(x, c)
        z = self.reparameterize(mu, logvar)
        recon_x = self.decode(z, c)
        return recon_x, mu, logvar
```

Prepare the data.

In [ ]:
```
batch_size = 128

transform = transforms.Compose([transforms.ToTensor(), transforms.Lambda(lambda x: x.view(-1))])
train_dataset = datasets.FashionMNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.FashionMNIST(root='./data', train=False, download=True, transform=transform)

# Splitting training data into training and validation sets (80% train, 20% validation).
train_size = int(0.8 * len(train_dataset))
val_size = len(train_dataset) - train_size
train_dataset, val_dataset = random_split(train_dataset, [train_size, val_size])

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

Define the train and validation process. Use One-hot encode labels and concatenate as conditional information if we train CVAE and Use dummy zeros for vanilla VAE.

In [ ]:
```python
def train(model, optimizer, epoch, use_conditional=False, beta=1):
    model.train()
    train_loss = 0
    total_recon_loss = 0
    total_kld_loss = 0
    for batch_idx, (data, target) in enumerate(train_loader):
        if use_conditional:
            # One-hot encode labels and concatenate as conditional information
            c = F.one_hot(target, num_classes=10).float()
        else:
            # Use dummy zeros for non-conditional VAE
            c = torch.zeros(data.size(0), 10)

        optimizer.zero_grad()
        recon_batch, mu, logvar = model(data, c)
        recon_loss = F.binary_cross_entropy(recon_batch, data, reduction='sum')
        kld_loss = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
        loss = recon_loss + beta * kld_loss
        loss.backward()
        train_loss += loss.item()
        total_recon_loss += recon_loss.item()
        total_kld_loss += kld_loss.item()
        optimizer.step()

    if epoch%5==0:
        print(f'Epoch {epoch}, Train Loss: {train_loss / len(train_loader.dataset):.4f}, Recon Loss: {total_recon_loss / len(train_loader.dat

def validate(model, use_conditional=False):
    model.eval()
    val_loss = 0
    total_recon_loss = 0
    total_kld_loss = 0
    with torch.no_grad():
        for data, target in val_loader:
            if use_conditional:
                c = F.one_hot(target, num_classes=10).float()
            else:
                c = torch.zeros(data.size(0), 10)

            recon_batch, mu, logvar = model(data, c)
            recon_loss = F.binary_cross_entropy(recon_batch, data, reduction='sum')
            kld_loss = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
            val_loss += (recon_loss + kld_loss).item()
            total_recon_loss += recon_loss.item()
            total_kld_loss += kld_loss.item()

    val_loss /= len(val_loader.dataset)
    total_recon_loss /= len(val_loader.dataset)
    total_kld_loss /= len(val_loader.dataset)
    print(f'Validation Loss: {val_loss:.4f}, Recon Loss: {total_recon_loss:.4f}, KLD Loss: {total_kld_loss:.4f}')
```

Train the VAE.

In [ ]:
```python
n_in = 784
n_hid = 400
z_dim = 20
learning_rate = 1e-3
model = VAE(n_in=n_in, n_hid=n_hid, z_dim=z_dim)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
n_epochs = 20
use_conditional = False  # Set to False for vanilla VAE, True for conditional VAE

for epoch in range(1, n_epochs + 1):
    train(model, optimizer, epoch, use_conditional=use_conditional)
    if epoch%5==0:
        validate(model, use_conditional=use_conditional)

# Saving the model weights
torch.save(model.state_dict(), 'cvae_fashion_mnist.pth' if use_conditional else 'vae_fashion_mnist.pth')
print("VAE model training complete and weights saved.")
```

```
Epoch 5, Train Loss: 247.2739, Recon Loss: 232.3511, KLD Loss: 14.9228
Validation Loss: 246.7356, Recon Loss: 231.6768, KLD Loss: 15.0588
Epoch 10, Train Loss: 243.2606, Recon Loss: 227.9328, KLD Loss: 15.3279
Validation Loss: 243.0841, Recon Loss: 227.9738, KLD Loss: 15.1103
Epoch 15, Train Loss: 241.6818, Recon Loss: 226.2114, KLD Loss: 15.4704
Validation Loss: 241.8293, Recon Loss: 226.0744, KLD Loss: 15.7548
Epoch 20, Train Loss: 240.7505, Recon Loss: 225.2116, KLD Loss: 15.5390
Validation Loss: 240.8245, Recon Loss: 225.7334, KLD Loss: 15.0911
VAE model training complete and weights saved.
```

Train the CVAE.

In [ ]:
```python
n_in = 784
n_hid = 400
z_dim = 20
learning_rate = 1e-3
model = VAE(n_in=n_in, n_hid=n_hid, z_dim=z_dim)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
n_epochs = 20
use_conditional = True   # Set to False for vanilla VAE, True for conditional VAE
```

```
for epoch in range(1, n_epochs + 1):
    train(model, optimizer, epoch, use_conditional=use_conditional)
    if epoch%5==0:
        validate(model, use_conditional=use_conditional)

# Saving the model weights
torch.save(model.state_dict(), 'cvae_fashion_mnist.pth' if use_conditional else 'vae_fashion_mnist.pth')
print("CVAE model training complete and weights saved.")
```

```
Epoch 5, Train Loss: 244.9166, Recon Loss: 231.7806, KLD Loss: 13.1360
Validation Loss: 244.4941, Recon Loss: 230.7921, KLD Loss: 13.7020
Epoch 10, Train Loss: 240.3897, Recon Loss: 227.2638, KLD Loss: 13.1259
Validation Loss: 240.3725, Recon Loss: 226.5769, KLD Loss: 13.7956
Epoch 15, Train Loss: 238.5209, Recon Loss: 225.4796, KLD Loss: 13.0412
Validation Loss: 239.1864, Recon Loss: 225.4931, KLD Loss: 13.6934
Epoch 20, Train Loss: 237.4462, Recon Loss: 224.4569, KLD Loss: 12.9894
Validation Loss: 237.6079, Recon Loss: 224.2056, KLD Loss: 13.4023
CVAE model training complete and weights saved.
```

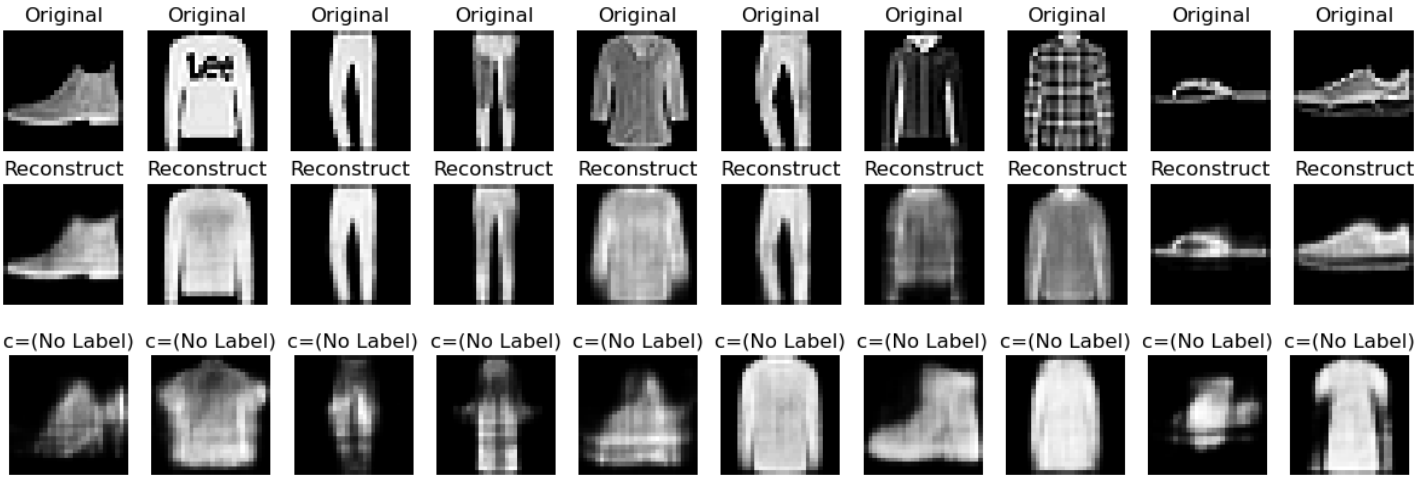
Display original images, reconstructed images, and generate new images

In [ ]:
```
def display_images(model,use_conditional=True):
    data, target = next(iter(test_loader))
    if use_conditional:
        c = F.one_hot(target, num_classes=10).float()
    else:
        c = torch.zeros(data.size(0), 10)

    with torch.no_grad():
        recon_batch, _, _ = model(data, c)

    fig, axes = plt.subplots(2, 10, figsize=(15, 3))
    for i in range(10):
        axes[0, i].imshow(data[i].view(28, 28), cmap='gray')
        axes[0, i].axis('off')
        axes[0, i].set_title('Original')

        axes[1, i].imshow(recon_batch[i].view(28, 28), cmap='gray')
        axes[1, i].axis('off')
        axes[1, i].set_title('Reconstruct')
    plt.show()

    z = torch.randn(10, z_dim)
    if use_conditional:
        c = F.one_hot(torch.arange(10), num_classes=10).float()
    else:
        c = torch.zeros(10, 10)

    label_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat", "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]

    with torch.no_grad():
        generated_images = model.decode(z, c)

    # Plot generated images
    fig, axes = plt.subplots(1, 10, figsize=(15, 3))
    for i in range(10):
        axes[i].imshow(generated_images[i].view(28, 28), cmap='gray')
        axes[i].axis('off')
        label = label_names[i] if use_conditional else 'No Label'
        axes[i].set_title(f'c={label})')
    plt.show()
```

Display the effect of VAE.

In [ ]:
```
use_conditional = False
vae = VAE(n_in=n_in, n_hid=n_hid, z_dim=z_dim)
vae.load_state_dict(torch.load('cvae_fashion_mnist.pth' if use_conditional else 'vae_fashion_mnist.pth'))
vae.eval()

display_images(vae,use_conditional)
```

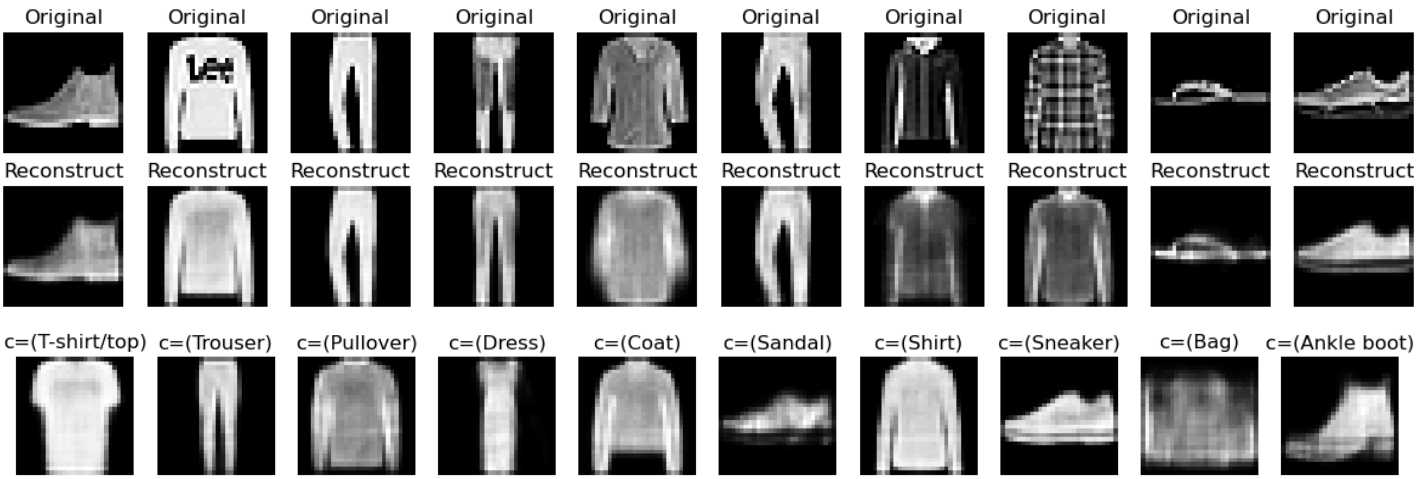
```
C:\Users\14857\AppData\Local\Temp\ipykernel_23368\1504968035.py:3: FutureWarning: You are using `torch.load` with `weights_only=False` (the cu
rrent default value), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execute ar
bitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future r
elease, the default value for `weights_only` will be flipped to `True`. This limits the functions that could be executed during unpickling. Ar
bitrary objects will no longer be allowed to be loaded via this mode unless they are explicitly allowlisted by the user via `torch.serializati
on.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 fil
e. Please open an issue on GitHub for any issues related to this experimental feature.
  vae.load_state_dict(torch.load('cvae_fashion_mnist.pth' if use_conditional else 'vae_fashion_mnist.pth'))
```



Display the effect of CVAE.

In [ ]:
```
use_conditional = True
cvae = VAE(n_in=n_in, n_hid=n_hid, z_dim=z_dim)
cvae.load_state_dict(torch.load('cvae_fashion_mnist.pth' if use_conditional else 'vae_fashion_mnist.pth'))
cvae.eval()

display_images(cvae,use_conditional)
```

```
C:\Users\14857\AppData\Local\Temp\ipykernel_23368\1840366698.py:3: FutureWarning: You are using `torch.load` with `weights_only=False` (the cu
rrent default value), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execute ar
bitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future r
elease, the default value for `weights_only` will be flipped to `True`. This limits the functions that could be executed during unpickling. Ar
bitrary objects will no longer be allowed to be loaded via this mode unless they are explicitly allowlisted by the user via `torch.serializati
on.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 fil
e. Please open an issue on GitHub for any issues related to this experimental feature.
  cvae.load_state_dict(torch.load('cvae_fashion_mnist.pth' if use_conditional else 'vae_fashion_mnist.pth'))
```



It can be observed that both VAE and CVAE achieve good reconstruction results. However, in terms of generation performance, CVAE can generate images corresponding to specific classes based on conditional information, and the image quality is higher.

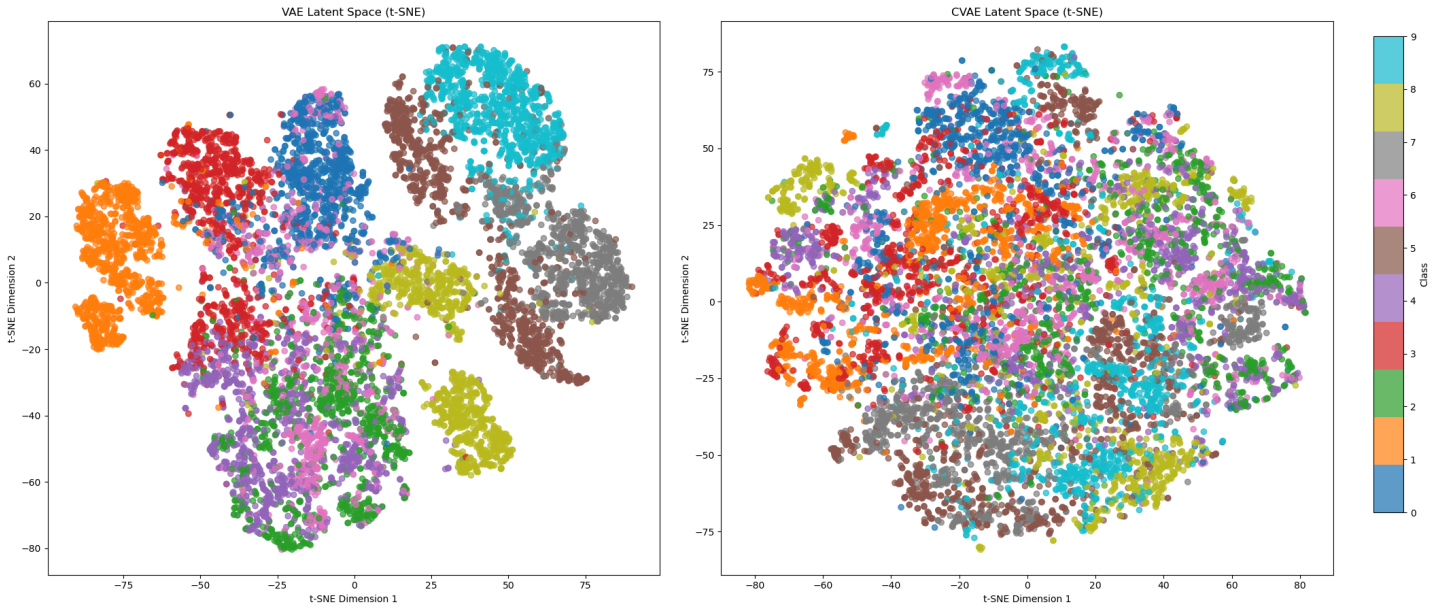
3.3 Manifold Comparison

The visualization of μ.

In [ ]:
```
def visualize_latent_space(model, use_conditional=True):
    model.eval()
    all_mu = []
    all_labels = []
    with torch.no_grad():
        for data, target in test_loader:
            if use_conditional:
                c = F.one_hot(target, num_classes=10).float()
            else:
                c = torch.zeros(data.size(0), 10)

            _, mu, _ = model(data, c)
            all_mu.append(mu)
            all_labels.append(target)

    all_mu = torch.cat(all_mu, dim=0).numpy()
    all_labels = torch.cat(all_labels, dim=0).numpy()

    tsne = TSNE(n_components=2, random_state=42)
    latent_2d = tsne.fit_transform(all_mu)
```

```
    return latent_2d, all_labels

# Get latent spaces for VAE and CVAE
vae_latent, vae_labels = visualize_latent_space(vae, use_conditional=False)
cvae_latent, cvae_labels = visualize_latent_space(cvae, use_conditional=True)

fig, axs = plt.subplots(1, 2, figsize=(22, 10))

# Plot for VAE
scatter_vae = axs[0].scatter(vae_latent[:, 0], vae_latent[:, 1], c=vae_labels, cmap='tab10', alpha=0.7)
axs[0].set_title('VAE Latent Space (t-SNE)')
axs[0].set_xlabel('t-SNE Dimension 1')
axs[0].set_ylabel('t-SNE Dimension 2')
# Plot for CVAE
scatter_cvae = axs[1].scatter(cvae_latent[:, 0], cvae_latent[:, 1], c=cvae_labels, cmap='tab10', alpha=0.7)
axs[1].set_title('CVAE Latent Space (t-SNE)')
axs[1].set_xlabel('t-SNE Dimension 1')
axs[1].set_ylabel('t-SNE Dimension 2')

fig.suptitle('Latent Representation of μ in VAE and CVAE', fontsize=20)

cbar_ax = fig.add_axes([0.92, 0.15, 0.02, 0.7])
cbar = fig.colorbar(scatter_vae, cax=cbar_ax)
cbar.set_label('Class')

plt.tight_layout(rect=[0, 0, 0.9, 0.95])
plt.subplots_adjust(wspace=0.1)
plt.show()
```

```
C:\Users\14857\AppData\Local\Temp\ipykernel_23368\3432734237.py:56: UserWarning: This figure includes Axes that are not compatible with tight_layout, so results might be incorrect.
  plt.tight_layout(rect=[0, 0, 0.9, 0.95])  # Adjust layout to make room for the colorbar and suptitle
```



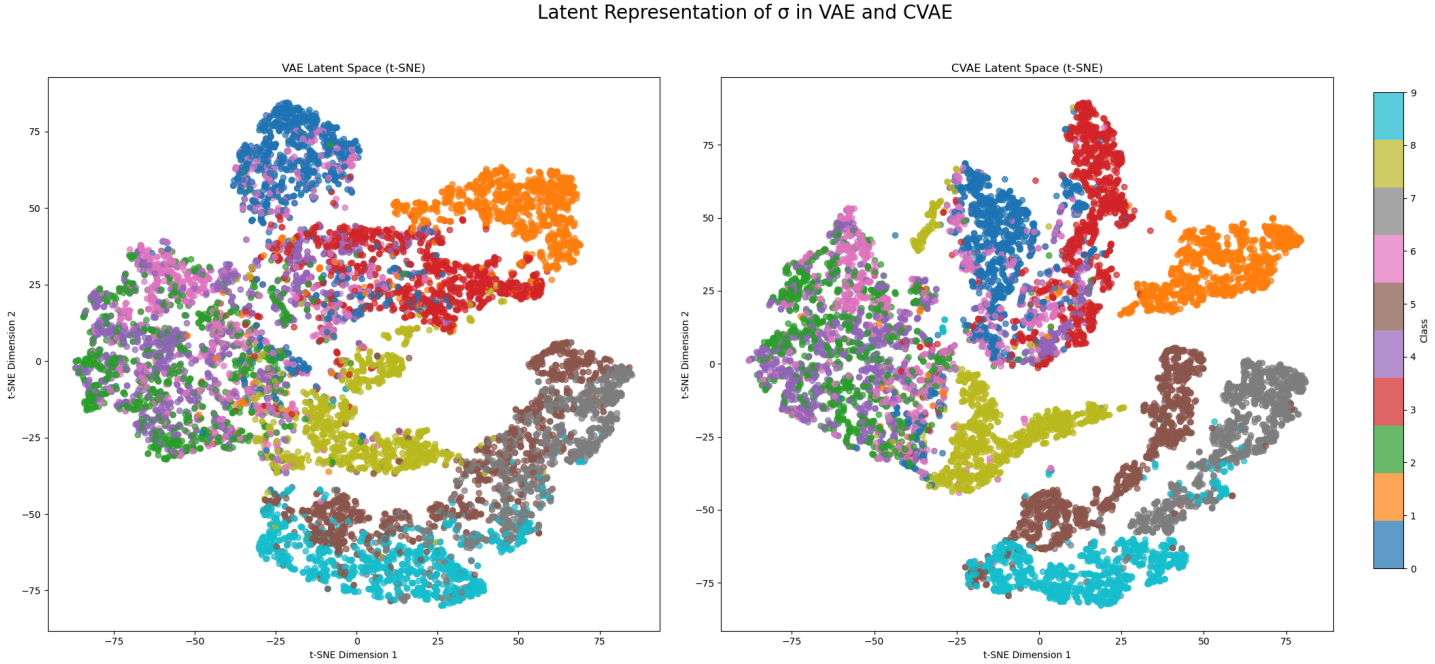
The visualization of σ.

In [ ]:

```
def visualize_latent_space(model, use_conditional=True):
    model.eval()
    all_sigma = []
    all_labels = []
    with torch.no_grad():
        for data, target in test_loader:
            if use_conditional
                c = F.one_hot(target, num_classes=10).float()
            else:
                c = torch.zeros(data.size(0), 10)

            _, _, sigma = model(data, c)
            all_sigma.append(sigma)
            all_labels.append(target)

    all_sigma = torch.cat(all_sigma, dim=0).numpy()
    all_labels = torch.cat(all_labels, dim=0).numpy()

    tsne = TSNE(n_components=2, random_state=42)
    latent_2d = tsne.fit_transform(all_sigma)

    return latent_2d, all_labels

# Get latent spaces for VAE and CVAE
vae_latent, vae_labels = visualize_latent_space(vae, use_conditional=False)
cvae_latent, cvae_labels = visualize_latent_space(cvae, use_conditional=True)

fig, axs = plt.subplots(1, 2, figsize=(22, 10))

# Plot for VAE
```

```
scatter_vae = axs[0].scatter(vae_latent[:, 0], vae_latent[:, 1], c=vae_labels, cmap='tab10', alpha=0.7)
axs[0].set_title('VAE Latent Space (t-SNE)')
axs[0].set_xlabel('t-SNE Dimension 1')
axs[0].set_ylabel('t-SNE Dimension 2')
# Plot for CVAE
scatter_cvae = axs[1].scatter(cvae_latent[:, 0], cvae_latent[:, 1], c=cvae_labels, cmap='tab10', alpha=0.7)
axs[1].set_title('CVAE Latent Space (t-SNE)')
axs[1].set_xlabel('t-SNE Dimension 1')
axs[1].set_ylabel('t-SNE Dimension 2')

fig.suptitle('Latent Representation of σ in VAE and CVAE', fontsize=20)

cbar_ax = fig.add_axes([0.92, 0.15, 0.02, 0.7])
cbar = fig.colorbar(scatter_vae, cax=cbar_ax)
cbar.set_label('Class')

plt.tight_layout(rect=[0, 0, 0.9, 0.95])
plt.subplots_adjust(wspace=0.1)
plt.show()
```

```
C:\Users\14857\AppData\Local\Temp\ipykernel_23368\2402147482.py:56: UserWarning: This figure includes Axes that are not compatible with tight_layout, so results might be incorrect.
  plt.tight_layout(rect=[0, 0, 0.9, 0.95])  # Adjust layout to make room for the colorbar and suptitle
```



When visualizing the latent representations learned by both the VAE and CVAE using t-SNE, the following observations were made: in the latent space of μ, the VAE's latent representation shows a clearer separation between classes, while the CVAE's representation of the latent variable μ appears more mixed across classes. However, the variance σ for the CVAE is more distinct and well-defined compared to that in the VAE.

Hypothesis: This phenomenon occurs because in the CVAE, the class labels are used as conditional information during training. As a result, the need for latent variables to capture the class-related differences is reduced, since the class is already explicitly provided to the model. This leads to a more mixed distribution for μ across classes in the CVAE. On the other hand, since the class differences are explicitly conditioned, the latent variable σ, which represents the variance, can better capture subtle variations within each class, resulting in a clearer representation. This highlights the effect of conditioning in disentangling different factors of variation—specifically, by focusing the model on class-internal differences rather than class distinctions.