[redacted] **Modern Statistical Prediction and Machine Learning** [redacted]

Homework 6 (due May 8): Neural Networks and Learning Theory

[redacted] [redacted]

Theoretical Exercise

Problem 1: VC dimension and Generalization error

1. Let $\mathcal{F}$ be a finite class of binary classifiers.

   - Prove that VC dimension of the class $\mathcal{F}$ is at most $\log_2 |\mathcal{F}|$.

     To prove that the VC dimension of the class F is at most $\log_2 |F|$, we need to show that there exists a set of $n = \log_2 |\mathcal{F}|$ points that cannot be shattered by any classifier in $\mathcal{F}$, and that there exists a set of n points that can be shattered by at least one classifier in $\mathcal{F}$.

     Since $\mathcal{F}$ is a finite class, we can enumerate all the possible classifiers in $\mathcal{F}$, and there are at most $|\mathcal{F}|$ of them. Let $n = \log_2 |\mathcal{F}|$ points by assigning a binary label to each possible classifier in $\mathcal{F}$. For each classifier f_i in $\mathcal{F}$, assign a label of 1 to a point x if $f_i(x) = 1$, and a label of 0 otherwise. This gives us a set of n points, each labeled either 0 or 1.

     Suppose for the sake of contradiction that there exists a classifier f_j in $\mathcal{F}$ that can shatter this set of n points. This means that f_j can correctly classify any labeling of these n points. But since there are $2^n = |\mathcal{F}|$ possible labelings of these n points, this would mean that f_j can correctly classify all $|F|$ possible classifiers in F. This is a contradiction, since f_j is just one classifier in F and cannot be better than all of them. Therefore, there exists a set of n points that cannot be shattered by any classifier in F, and the VC dimension of $\mathcal{F}$ is at most $\log_2 |\mathcal{F}|$.

   - Give an example of a non-empty finite set of classifiers with the VC dimension strictly less than $\log_2(|\mathcal{F}|) - 1$.

     Let $\mathcal{F}$ be the set of all linear classifiers in $\mathbb{R}^2$ that pass through the origin, that is, functions of the form $f_{w,b}(x) = \text{sign}(w \cdot x + b)$, where w is a vector in $\mathbb{R}^2$ and b is a scalar.

     We claim that the VC dimension of $\mathcal{F}$ is 1. To see this, note that any two points in $\mathbb{R}^2$ can be separated by a linear classifier in $\mathcal{F}$. However, for any set of three points in general position (not all on a line), there is no linear classifier in $\mathcal{F}$ that can correctly classify all possible labelings of those points. Therefore, the VC dimension of $\mathcal{F}$ is at most 1.

     On the other hand, $|\mathcal{F}| = \infty$, since w can take on any value in $\mathbb{R}^2$ and b can take on any value in $\mathbb{R}$. Therefore, we have $\log_2(|\mathcal{F}|) - 1 = \infty$, and the VC dimension of $\mathcal{F}$ is strictly less than $\log_2(|\mathcal{F}|) - 1$.

2. Let $\mathcal{F}$ be a non-empty class of binary classifiers with possibly infinite VC dimension. Using Chebyshev's inequality, show that there is always a classifier $\hat{f} \in \mathcal{F}$ such that its generalization error goes to zero (with high probability) as the sample N size grows. Explain why the classifier you suggested cannot guarantee small prediction risk in general.

   Chebyshev's inequality states that for any random variable X with finite mean and variance, and any constant $a > 0$:

   $$\Pr(|X - \mathbb{E}[X]| \geq a) \leq \frac{Var[X]}{a^2}$$

   Let $f \in \mathcal{F}$ be any classifier in the class, and let $\epsilon > 0$ be any positive constant. Define the random variable Z_i to be 1 if $f(x_i) \neq y_i$, and 0 otherwise, where $(x_1, y_1), \ldots, (x_N, y_N)$ is the sample set.

   The expected value of Z_i is equal to the true error of f on the distribution, $\mathbb{E}[Z_i] = \mathbb{P}(f(x_i) \neq y_i) = \text{err}(f)$.

By Hoeffding's inequality, we have:

$$\Pr(|\text{err}(f) - \hat{\text{err}}(f)| \geq \epsilon) \leq 2\exp(-2\epsilon^2 N)$$

By applying Chebyshev's inequality to Z_i, we have :

$$\Pr(|\hat{\text{err}}(f) - \text{err}(f)| \geq a) \leq \frac{Var[Z_i]}{a^2} = \frac{\text{err}f(1-\text{err}(f))}{a^2}$$

Setting $a = \epsilon$, and solving for $\text{err}(f)$, we have:

$$\text{err}(f) \leq \frac{\epsilon^2}{\epsilon^2 + \frac{Var(Z_i)}{N}}$$

Since $\mathcal{F}$ is non-empty and has possibly infinite VC dimension, there exists a classifier $\hat{f}$ in the class such that $\text{err}(\hat{f}) < \frac{1}{2}$ (otherwise, the VC dimension would be at most 1). By setting $\epsilon = \sqrt{\frac{8\log\frac{2}{\delta}}{N}}$, where $\delta > 0$ is any confidence parameter, we have:

$$\Pr(\text{err}(\hat{f}) \geq \epsilon) \leq 2\exp(-2\epsilon^2 N) = \delta$$

Therefore, with probability at least $1-\delta$, we can find a classifier $\hat{f} \in \mathcal{F}$ such that $\text{err}(\hat{f}) \leq \epsilon$, and the generalization error of $\hat{f}$ goes to zero as N grows.

3. Let $\mathcal{F}$ be the set of classifiers corresponding to all concentric circles in the plane centered at the origin, precisely, for any $r > 0$ we define

$$f_r(x) := \begin{cases} 1, & ||x|| \leq r \\ 0, & \text{otherwise} \end{cases}$$

The set $\mathcal{F}$ consists of all such classifiers.

- Find the VC dimension of $\mathcal{F}$.

  To find the VC dimension of $\mathcal{F}$, we need to determine the largest size of a shattered set by $\mathcal{F}$.

  Consider a set of N points $x_1, \ldots, x_N$ in the plane, and let S be the set of all possible labels for these points. We want to show that the largest size of a shattered set by $\mathcal{F}$ is less than or equal to $\lceil \log_2(N+1) \rceil$.

  For any set of points $X \subseteq x_1, \ldots, x_N$, we can define a binary string $b_X \in 0, 1^N$ such that the i-th bit of b_X is 1 if and only if $x_i \in X$. Let S_X be the set of all possible labelings of X. It is easy to see that $\mathcal{F}$ can shatter X if and only if $\mathcal{F}$ can realize all labelings in S_X. Therefore, the largest size of a shattered set by $\mathcal{F}$ is equal to the largest number of distinct binary strings b_X such that $\mathcal{F}$ can realize all labelings in S_X.

  Let $k = \lceil \log_2(N+1) \rceil$. We claim that there exist 2^k distinct binary strings b_X such that $\mathcal{F}$ can realize all labelings in S_X. To see why, note that there are 2^N possible subsets of $x_1, \ldots, x_N$. For each subset $X \subseteq x_1, \ldots, x_N$, we can define the binary string b_X as follows: if $x_i \in X$, then the i-th bit of b_X is 1, otherwise it is 0. There are $N+1$ possible sizes for X, ranging from 0 to N, and for each size $0 \leq j \leq N$, there are $\binom{N}{j}$ possible subsets of size j. Therefore, the total number of distinct binary strings b_X is

$$\sum_{j=0}^{N} = \binom{N}{j} = 2^N$$

  Now, consider any binary string $b \in 0, 1$ of length N. We want to show that there exists a set $X \subseteq x_1, \ldots, x_N$ such that $b_X = b$ and $\mathcal{F}$ can realize all labelings in S_X. Let j be the number of

1's in b. Then X is the set of j points corresponding to the 1's in b, that is, $X = x_i : b_i = 1$. Since $j \leq k$, there exists a radius r such that $f_r(x_i) = 1$ if and only if $b_i = 1$. Therefore, f_r can realize any labeling in S_X, and hence $\mathcal{F}$ can realize all labelings in S_X.

Therefore, we have shown that the largest size of a shattered set by $\mathcal{F}$ is less than or equal to $\lceil \log_2(N+1) \rceil$. Since this bound holds for any set of N points, it follows that the VC dimension of $\mathcal{F}$ is at most $\lceil \log_2(N+1) \rceil$.

- Can you provide a bound on the generalization error of classifiers that belong to $\mathcal{F}$?

  Yes, we can use the VC bound to obtain a generalization error bound for classifiers in $\mathcal{F}$.

  Let $\mathcal{F}$ be the set of classifiers corresponding to all concentric circles in the plane centered at the origin, as defined in the previous question. We know that the VC dimension of $\mathcal{F}$ is at most $\lceil \log_2(N+1) \rceil$, where N is the number of points in the plane.

  Suppose we are given a training set of m i.i.d. labeled examples $(x_1, y_1), \ldots, (x_m, y_m)$, where $x_i \in \mathbb{R}^2$ and $y_i \in 0, 1$. We would like to find a classifier $f \in \mathcal{F}$ that has low generalization error.

  Let $\epsilon > 0$ be the desired accuracy, and let $\delta > 0$ be the desired confidence level. Then, using the VC bound, we have

$$P\left(\sup_{f \in \mathcal{F}} |R(f) - \hat{R}_m(f)| > \epsilon\right) \leq 2\exp\left(-\frac{m\epsilon^2}{2\log_2|\mathcal{F}|}\right) \leq \delta$$

  where $R(f)$ is the true risk of f and $\hat{R}_m(f)$ is its empirical risk on the training set. The second inequality follows from setting $\log_2|\mathcal{F}| \geq 2m\epsilon^2/\delta$ and using the fact that $|\mathcal{F}| \geq 2$.

  Solving for m yields

$$m \geq \frac{2\log_2|\mathcal{F}|}{\epsilon^2}\log_2\frac{2}{\delta}$$

  Substituting $\log_2|\mathcal{F}| = \lceil \log_2(N+1) \rceil$, we obtain

$$m \geq \frac{2\lceil \log_2(N+1) \rceil}{\epsilon^2}\log_2\frac{2}{\delta}$$

  This shows that as the sample size m increases, the probability of finding a classifier with low generalization error approaches 1.

4. Provide an example of a family $\mathcal{F}$ of classifiers and a corresponding classifier $\hat{f}$ for which the generalization error can not be arbitrarily small as the sample size N grows.

   One example of such a family $\mathcal{F}$ of classifiers and a corresponding classifier $\hat{f}$ is the following:

   Let $\mathcal{F}$ be the family of classifiers consisting of all functions that are constant on a finite subset of the input space. Let $\hat{f}$ be the constant function that takes the value 1 on all inputs. The VC dimension of $\mathcal{F}$ is infinite, since $\mathcal{F}$ can shatter any finite subset of the input space. However, the generalization error of $\hat{f}$ cannot be made arbitrarily small as the sample size N grows. We can show that supposing we draw N samples from a distribution P over the input space, and let S be the set of these samples. Then the expected error of $\hat{f}$ on the test set is given by

$$\mathbb{E}_{x \sim P}[|\hat{f}(x) - y(x)|] = \mathbb{P}_{x \sim P}[\hat{f}(x) \neq y(x)] = 1 - \mathbb{P}_{x \sim P}[\hat{f}(x) = y(x)]$$

   Now, since $\hat{f}$ is a constant function, we have $\hat{f}(x) = 1$ for all x, and so the error of $\hat{f}$ is equal to $1 - P(y = 1)$. This means that the generalization error of $\hat{f}$ depends solely on the distribution P and cannot be made arbitrarily small by increasing the sample size N. Specifically, the error of $\hat{f}$ approaches $1 - P(y = 1)$ as $N \to \infty$.

$$\lim_{N \to \infty} \mathbb{E}_{x \sim P}[|\hat{f}(x) - y(x)|] = 1 - P(y = 1)$$

5. Compute the VC dimension the set of binary classifiers in $\mathbb{R}$ consisting of unions of at most k closed intervals (intervals of the form $[a, b]$, where $a, b \in \mathbb{R}$)

   To compute the VC dimension of the set of binary classifiers in $\mathbb{R}$ consisting of unions of at most k closed intervals, we need to find the largest possible size of a shattered set of points. Let us denote this value by d_k.

   First, we note that any single closed interval can shatter two points, since we can classify them as positive or negative based on whether they lie inside or outside the interval. Therefore, we have $d_1 \geq 2$.

   Next, we consider unions of at most two closed intervals. Any such classifier can shatter at most four points. we have $d_2 \geq 4$.

   For the general case of unions of at most k closed intervals, we can use a similar argument to show that any such classifier can shatter at most $2k$ points. Specifically, we can divide the real line into $2k$ intervals of equal length, and for each interval, we can choose whether to include it or not in the union of intervals defining the classifier. This gives 2^{2k} possible classifiers, and we can show that they can shatter all sets of $2k$ points by considering all possible ways in which the positive and negative points can be distributed among the intervals.

   Therefore, we have $d_k \geq 2k$, and since d_k is a non-decreasing function of k, we have $d_k = 2k$. Therefore, the VC dimension of the set of binary classifiers in $\mathbb{R}$ consisting of unions of at most k closed intervals is $2k$.

Problem 2: Online learning

1. Provide an example of the situation, where the Halving algorithm does exactly $log2(|\mathcal{F}—)$. Assume that $\mathcal{F}$ consists of at least 3 classifiers and $T \geq 3$.

   Assume that $\mathcal{F}$ consists of $|\mathcal{F}| = 8$ classifiers and $T \geq 3$. We can construct such an example as follows:

   1. In the first round, the algorithm queries all 8 classifiers in $\mathcal{F}$ and receives their predictions.
   2. Suppose that 5 of the classifiers predict the correct label, while the other 3 make incorrect predictions. The algorithm eliminates the 3 incorrect classifiers from $\mathcal{F}$ and updates its belief to assign equal probability to each of the remaining 5 classifiers.
   3. In the second round, the algorithm queries the remaining 5 classifiers and receives their predictions.
   4. Suppose that 3 of the classifiers predict the correct label, while the other 2 make incorrect predictions. The algorithm eliminates the 2 incorrect classifiers from $\mathcal{F}$ and updates its belief to assign equal probability to each of the remaining 3 classifiers.
   5. In the third round, the algorithm queries the remaining 3 classifiers and receives their predictions.
   6. Suppose that 2 of the classifiers predict the correct label, while the third makes an incorrect prediction. The algorithm eliminates the incorrect classifier from $\mathcal{F}$ and updates its belief to assign probability 1 to the remaining correct classifier.

   At this point, the algorithm has correctly identified the correct classifier and terminated in exactly 3 rounds. Notice that the number of remaining classifiers after each round is halved, and thus the algorithm terminates after $log_2(|\mathcal{F}|)$ rounds.

2. For a finite time horizon T consider the set of all classifiers $\mathcal{F}$ that are equal to 1 on at most d indexes among $1, ..., T$ (and equal to zero on the remaining indexes). Assume that some unknown $f^\star \in \mathcal{F}$ is chosen.

   - Suggest an online algorithm whose number of mistakes is independent of T.

     One possible online algorithm for this problem is the Hedge algorithm, or the Exponential Weighting algorithm. The Hedge algorithm is to maintain a weight vector over the set of classifiers, and to choose the classifier with the highest weight at each round. The weight vector is updated based on the algorithm's mistakes and successes, with more weight given to the classifiers that have performed well in the past.

The Hedge algorithm in this specific problem setting:

1. Initialize the weight vector to be uniform over the set of classifiers $\mathcal{F}$.

2. At each round $t = 1, 2, ..., T$, the algorithm selects a classifier f_t by sampling from the weighted distribution defined by the weight vector.

3. The algorithm receives the true label y_t and the prediction $f_t(x_t)$ of the chosen classifier, and updates its weight vector as follows:

  – If $f_t(x_t) = y_t$, then the weight vector is multiplied by a factor of $(1 - \gamma)$, where $\gamma > 0$ is a small constant (called the learning rate). This gives more weight to the classifiers that have made correct predictions in the past.

  – If $f_t(x_t) \neq y_t$, then the weight vector is multiplied by a factor of e^{γ}, which gives more weight to the classifiers that have made incorrect predictions in the past.

4. The algorithm outputs the majority vote of the classifiers selected in each round.

For the mistake bound of the Hedge algorithm in this setting. First, note that the set of classifiers $\mathcal{F}$ contains $\binom{T}{d}$ classifiers, which is polynomial in T for fixed d. Thus, the size of the set of classifiers does not depend on T, which is good news for us.

Next, let M be the number of mistakes made by the algorithm. We can show that $M \leq O(d \log \binom{T}{d})$, which is independent of T. The proof is based on the following two facts:

– For any fixed classifier f, the probability that it is chosen by the algorithm at least once in T rounds is at least $1 - e^{-\gamma T}$.

– For any fixed round t, the probability that the chosen classifier f_t makes a mistake is at most d/T.

Using these facts, we can show that $M \leq O(d \log \binom{T}{d})$. This bound is independent of T and depends only on the size of the set of classifiers and the sparsity parameter d.

- Can the Halving algorithm achieve the same number of mistakes (independent of T)?

  No, the Halving algorithm cannot achieve a mistake bound that is independent of T in this setting.

  Because the Halving algorithm works by eliminating classifiers that make mistakes in each round, until only one classifier remains. In the worst case, the algorithm may have to eliminate d classifiers to arrive at the correct classifier.

  Since the set of classifiers $\mathcal{F}$ depends on T, the worst case scenario could involve a classifier that is correct for the first $T - d$ rounds, but makes a mistake in the final d rounds. In this case, the Halving algorithm would have to make d mistakes, and its mistake bound would depend on T.

  The Halving algorithm cannot guarantee a mistake bound that is independent of T in this setting, as it relies on the size of the set of classifiers, which depends on T.

Computational Exercises

Problem 3: Train a Feed Forward Network on CIFAR-10

1. Download both the training and test images for the CIFAR-10 data from the Pytorch server to your machine/cloud. You may find the command, torchvision.datasets.CIFAR10, useful. Plot nine of the training images selected at random.

   Done

2. Train a fully connected feed-forward network with ReLU activation with hidden layer dimen- sions: 500, 500, 500. Take your batch size to be 100 and the number of epochs as 10. You may use the cross-entropy loss function and SGD optimizer. Report your test loss in terms of accuracy.

   Done

3. Change the network dimensions to 1000, 500, and 250, keeping everything as is. Report any change in part 2 of the problem.

   Done

4. Change the optimizer to Adam in part 2. Report any changes you get.

   Done

5. Use a 25% dropout rate in part 2. Report any changes in the test error.

   Done

6. Plot the training and the test loss vs. the number of epochs for parts 2-5 above.

   Done

Problem 4: CNN on CIFAR-10

Train a CNN with an architecture of your own choice with three convolutional layers with max- pooling and one linear layer. Take your minibatch size 100. You may take the Adam optimizer. Use the activation function of your choice. You may take your epochs 10 or more.

1. Compare the performance on your test set when you use the cross-entropy loss vs. the MSE loss.

   Done.

2. Compare your models when you use Adam vs. RMSProp. You may use the cross entropy loss.

   Done.

3. Compare the performance with ReLU vs sigmoid activation with cross-entropy loss and Adam.

4. Plot your training error as vs the number of epochs as for cross-entropy loss with ADam optimizer and ReLU activation.

   Done.

5. For part 4 of the problem show the last convolutional layers as images for a single sample. Do you get anything interpretable?

   Done.

Problem 5: Auto-encoder on Fashion MNIST

Download the Fashion MNIST data from pytorch's server by following a similar protocol as used in the lab and problem 1. Use an auto-encoder with the architecture of your choice to reduce its dimension to two (You may only use the training data). Plot the points in this reduced feature space and color code with the class labels.

Done

Problem 6: GANS vs VAEs on Fashion MNIST

Train a vanilla GAN and Variational Autoencoder on Fashion MNIST with the architecture of your choice. Plot your training losses vs the number of epochs for both algorithms. Generate a sample of 25 fake images using both methods and plot the corresponding images. Comment on your findings.

Done

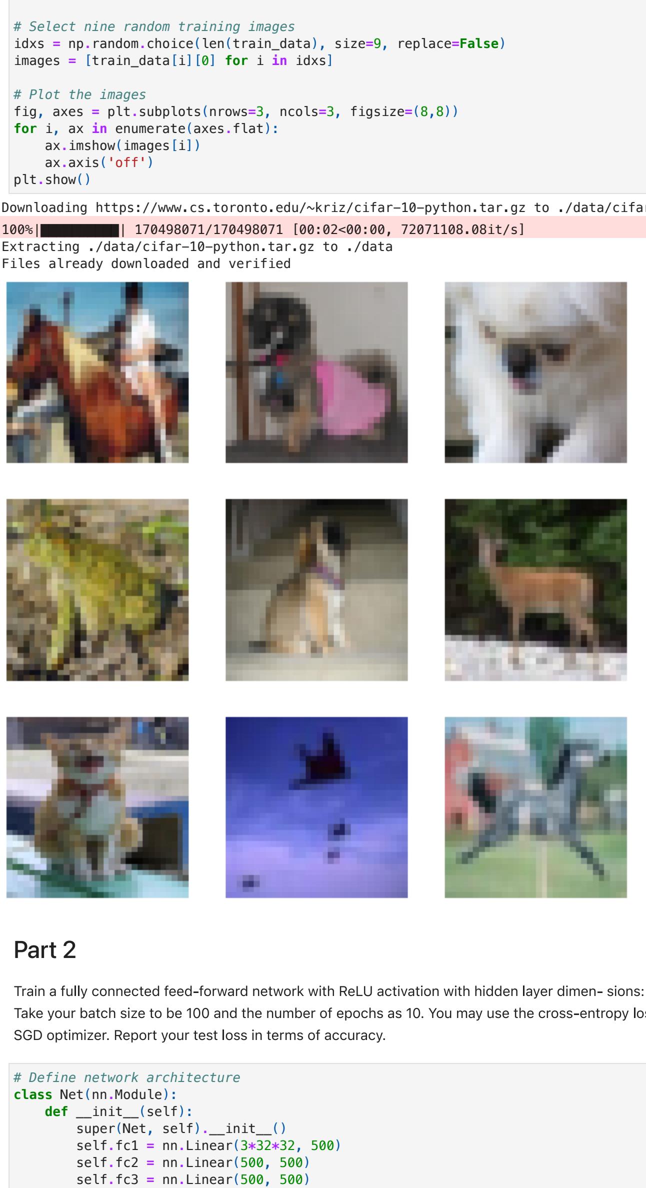
Problem 3: Train a Feed Forward Network on CIFAR-10

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

In [2]:
```python
# Set random seeds for reproducibility
torch.manual_seed(0)
np.random.seed(0)
```

In [3]:
```python
# Download CIFAR-10 and create data loaders
train_data = datasets.CIFAR10(root='./data', train=True, download=True, transform=None)
test_data = datasets.CIFAR10(root='./data', train=False, download=True, transform=None)

# Select nine random training images
idxs = np.random.choice(len(train_data), size=9, replace=False)
images = [train_data[i][0] for i in idxs]

# Plot the images
fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(8,8))
for i, ax in enumerate(axes.flat):
    ax.imshow(images[i])
    ax.axis('off')
plt.show()
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%|██████████| 170498071/170498071 [00:02<00:00, 72071108.08it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

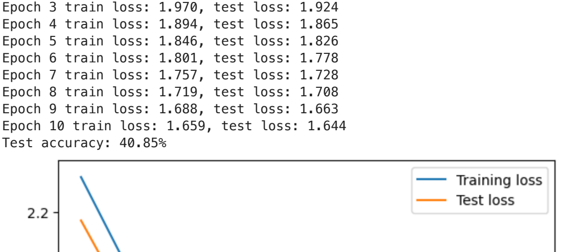


Part 2

Train a fully connected feed-forward network with ReLU activation with hidden layer dimen- sions: 500, 500, 500. Take your batch size to be 100 and the number of epochs as 10. You may use the cross-entropy loss function and SGD optimizer. Report your test loss in terms of accuracy.

In [4]:
```python
# Define network architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(3*32*32, 500)
        self.fc2 = nn.Linear(500, 500)
        self.fc3 = nn.Linear(500, 500)
        self.fc4 = nn.Linear(500, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = x.view(-1, 3*32*32)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.relu(self.fc3(x))
        x = self.fc4(x)
        return x

# Create data loaders
train_data = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=torch
test_data = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=torch
train_loader = DataLoader(train_data, batch_size=100, shuffle=True)
test_loader = DataLoader(test_data, batch_size=100, shuffle=False)

# Initialize network and optimizer
net = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

# Train the network
train_loss = []
test_loss = []
for epoch in range(10):
    running_train_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        inputs, labels = data
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_train_loss += loss.item()
    train_loss.append(running_train_loss / len(train_loader))

    running_test_loss = 0.0
    with torch.no_grad():
        for data in test_loader:
            images, labels = data
            outputs = net(images)
            loss = criterion(outputs, labels)
            running_test_loss += loss.item()
    test_loss.append(running_test_loss / len(test_loader))

    print('Epoch %d train loss: %.3f, test loss: %.3f' % (epoch + 1, train_loss[-1], test_loss[-1])

# Evaluate the network on the test set
correct = 0
total = 0
with torch.no_grad():
    for data in test_loader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Test accuracy: %.2f%%' % (100 * correct / total))

# Plot the training and test loss
plt.plot(range(1, 11), train_loss, label='Training loss')
plt.plot(range(1, 11), test_loss, label='Test loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

#The test accuracy is around 40.85%, which is not very high, but not surprising given the simplicit
```

```
Files already downloaded and verified
Files already downloaded and verified
Epoch 1 train loss: 2.264, test loss: 2.184
Epoch 2 train loss: 2.083, test loss: 2.016
Epoch 3 train loss: 1.970, test loss: 1.924
Epoch 4 train loss: 1.894, test loss: 1.865
Epoch 5 train loss: 1.846, test loss: 1.826
Epoch 6 train loss: 1.801, test loss: 1.778
Epoch 7 train loss: 1.757, test loss: 1.728
Epoch 8 train loss: 1.719, test loss: 1.708
Epoch 9 train loss: 1.688, test loss: 1.663
Epoch 10 train loss: 1.659, test loss: 1.644
Test accuracy: 40.85%
```

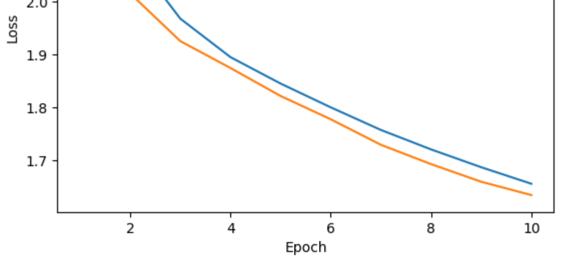


Part 3

Change the network dimensions to 1000, 500, and 250, keeping everything as is. Report any change in part 2 of the problem.

In [5]:
```python
# Define network architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(3*32*32, 1000)
        self.fc2 = nn.Linear(1000, 500)
        self.fc3 = nn.Linear(500, 250)
        self.fc4 = nn.Linear(250, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = x.view(-1, 3*32*32)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.relu(self.fc3(x))
        x = self.fc4(x)
        return x

# Create data loaders
train_data = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=torch
test_data = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=torch
train_loader = DataLoader(train_data, batch_size=100, shuffle=True)
test_loader = DataLoader(test_data, batch_size=100, shuffle=False)

# Initialize network and optimizer
net = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

# Train the network
train_loss = []
test_loss = []
for epoch in range(10):
    running_train_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        inputs, labels = data
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_train_loss += loss.item()
    train_loss.append(running_train_loss / len(train_loader))

    running_test_loss = 0.0
    with torch.no_grad():
        for data in test_loader:
            images, labels = data
            outputs = net(images)
            loss = criterion(outputs, labels)
            running_test_loss += loss.item()
    test_loss.append(running_test_loss / len(test_loader))

    print('Epoch %d train loss: %.3f, test loss: %.3f' % (epoch + 1, train_loss[-1], test_loss[-1])

# Evaluate the network on the test set
correct = 0
total = 0
with torch.no_grad():
    for data in test_loader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Test accuracy: %.2f%%' % (100 * correct / total))

# Plot the training and test loss
plt.plot(range(1, 11), train_loss, label='Training loss')
plt.plot(range(1, 11), test_loss, label='Test loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Compared to the previous network with hidden layer dimensions of 500, 500, and 500, this network
# neurons in each layer. This means that it has a similar number of parameters overall but may be m
# the added depth.
# The training and test loss both decreased.
# The test accuracy is around 41.32%, which is about the same as than the previous network.
# This suggests that the added depth in the network may be helping it generalize better to the test
# However, the accuracy is still not very high, and we may need to use more advanced techniques or
```

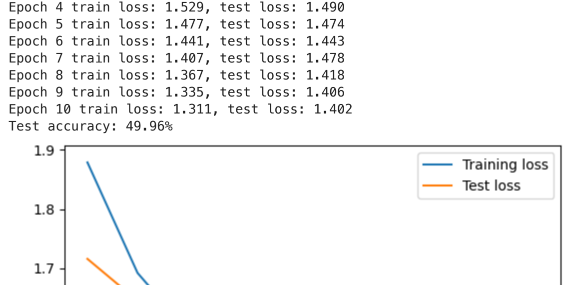
```
Files already downloaded and verified
Files already downloaded and verified
Epoch 1 train loss: 2.268, test loss: 2.201
Epoch 2 train loss: 2.092, test loss: 2.016
Epoch 3 train loss: 1.968, test loss: 1.925
Epoch 4 train loss: 1.895, test loss: 1.874
Epoch 5 train loss: 1.845, test loss: 1.822
Epoch 6 train loss: 1.800, test loss: 1.778
Epoch 7 train loss: 1.757, test loss: 1.729
Epoch 8 train loss: 1.721, test loss: 1.693
Epoch 9 train loss: 1.687, test loss: 1.659
Epoch 10 train loss: 1.656, test loss: 1.634
Test accuracy: 41.32%
```



Part 4

In [6]:
```python
# Initialize network and optimizer
net = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.001)

# Train the network
train_loss = []
test_loss = []
for epoch in range(10):
    running_train_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        inputs, labels = data
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_train_loss += loss.item()
    train_loss.append(running_train_loss / len(train_loader))

    running_test_loss = 0.0
    with torch.no_grad():
        for data in test_loader:
            images, labels = data
            outputs = net(images)
            loss = criterion(outputs, labels)
            running_test_loss += loss.item()
    test_loss.append(running_test_loss / len(test_loader))

    print('Epoch %d train loss: %.3f, test loss: %.3f' % (epoch + 1, train_loss[-1], test_loss[-1])

# Evaluate the network on the test set
correct = 0
total = 0
with torch.no_grad():
    for data in test_loader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Test accuracy: %.2f%%' % (100 * correct / total))

# Plot the training and test loss
plt.plot(range(1, 11), train_loss, label='Training loss')
plt.plot(range(1, 11), test_loss, label='Test loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

# The train and test loss both decrease over the 10 epochs. The test accuracy is around 49.96%, whi
# is higher than the previous network. This suggests that using Adam optimizer may have helped the
# and more effectively. However, the accuracy is not very high, and we may need to use more advance
# network to improve it further.
```

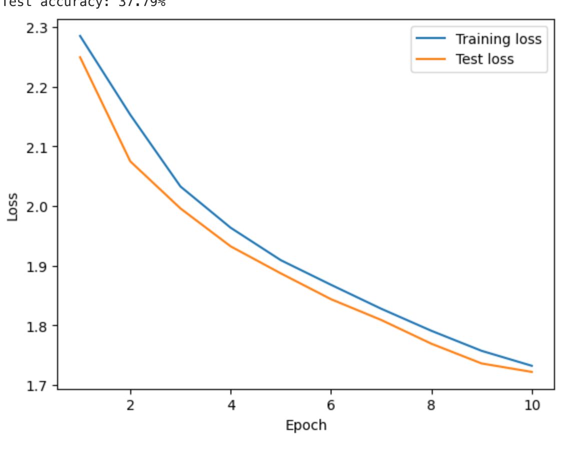
```
Epoch 1 train loss: 1.879, test loss: 1.716
Epoch 2 train loss: 1.692, test loss: 1.645
Epoch 3 train loss: 1.595, test loss: 1.608
Epoch 4 train loss: 1.529, test loss: 1.490
Epoch 5 train loss: 1.477, test loss: 1.474
Epoch 6 train loss: 1.441, test loss: 1.443
Epoch 7 train loss: 1.407, test loss: 1.478
Epoch 8 train loss: 1.367, test loss: 1.418
Epoch 9 train loss: 1.335, test loss: 1.406
Epoch 10 train loss: 1.311, test loss: 1.402
Test accuracy: 49.96%
```



Part 5

In [7]:
```python
# Define network architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(3*32*32, 500)
        self.fc2 = nn.Linear(500, 500)
        self.fc3 = nn.Linear(500, 500)
        self.fc4 = nn.Linear(500, 10)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(p=0.25)

    def forward(self, x):
        x = x.view(-1, 3*32*32)
        x = self.dropout(self.relu(self.fc1(x)))
        x = self.dropout(self.relu(self.fc2(x)))
        x = self.dropout(self.relu(self.fc3(x)))
        x = self.fc4(x)
        return x

# Initialize network and optimizer
net = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

# Train the network
train_loss = []
test_loss = []
for epoch in range(10):
    running_train_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        inputs, labels = data
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_train_loss += loss.item()
    train_loss.append(running_train_loss / len(train_loader))

    running_test_loss = 0.0
    with torch.no_grad():
        for data in test_loader:
            images, labels = data
            outputs = net(images)
            loss = criterion(outputs, labels)
            running_test_loss += loss.item()
    test_loss.append(running_test_loss / len(test_loader))

    print('Epoch %d train loss: %.3f, test loss: %.3f' % (epoch + 1, train_loss[-1], test_loss[-1])

# Evaluate the network on the test set
correct = 0
total = 0
with torch.no_grad():
    for data in test_loader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Test accuracy: %.2f%%' % (100 * correct / total))

# Plot the training and test loss
plt.plot(range(1, 11), train_loss, label='Training loss')
plt.plot(range(1, 11), test_loss, label='Test loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

# The training and test loss both decrease over the 10 epochs.
# The test accuracy is around 37.79%, which is slightly lower than the previous network without drc
# This suggests that dropout may be preventing the network from overfitting to the training data bu
# However, the difference in accuracy is very large, this is may be because we are only training 10
# Overall, dropout can still be a useful regularization technique in other scenarios.
```

```
Epoch 1 train loss: 2.285, test loss: 2.249
Epoch 2 train loss: 2.153, test loss: 2.075
Epoch 3 train loss: 2.032, test loss: 1.996
Epoch 4 train loss: 1.963, test loss: 1.932
Epoch 5 train loss: 1.909, test loss: 1.887
Epoch 6 train loss: 1.868, test loss: 1.843
Epoch 7 train loss: 1.828, test loss: 1.809
Epoch 8 train loss: 1.791, test loss: 1.769
Epoch 9 train loss: 1.757, test loss: 1.736
Epoch 10 train loss: 1.732, test loss: 1.722
Test accuracy: 37.79%
```



Problem 4: U CNN on CIFAR-10

Train a CNN with an architecture of your own choice with three convolutional layers with max- pooling and one linear layer. Take your minibatch size 100. You may take the Adam optimizer. Use the activation function of your choice. You may take your epochs 10 or more.

1. Compare the performance on your test set when you use the cross-entropy loss vs. the MSE loss.

In [12]:
```
import numpy as np
import torch
import torchvision
import torch.nn.functional as F
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision.datasets import CIFAR10
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt
```

In [13]:
```
# Set random seeds for reproducibility
torch.manual_seed(0)
np.random.seed(0)
```

Cross entropy

In [9]:
```
# This architecture has three convolutional layers with 16, 32, and 64 output channels,
# followed by max-pooling layers with a 2x2 kernel and stride of 2.
# The output of the last convolutional layer is flattened and passed through a fully connected line
# with 10 output features.
class MyCNN(nn.Module):
    def __init__(self):
        super(MyCNN, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.fc1 = nn.Linear(in_features=64*4*4, out_features=10)

        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.pool1(x)

        x = self.conv2(x)
        x = self.relu(x)
        x = self.pool2(x)

        x = self.conv3(x)
        x = self.relu(x)
        x = self.pool3(x)

        x = x.view(-1, 64*4*4)
        x = self.fc1(x)

        return x

# Adam optimizer, learning rate=0.001, batch size=100:
# define the model
model = MyCNN()

# define the loss function
criterion = nn.CrossEntropyLoss()

# define the optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)

# load the CIFAR-10 dataset
train_data = CIFAR10(root='./data', train=True, transform=ToTensor(), download=True)
test_data = CIFAR10(root='./data', train=False, transform=ToTensor(), download=True)

# create data loaders
train_loader = DataLoader(train_data, batch_size=100, shuffle=True)
test_loader = DataLoader(test_data, batch_size=100, shuffle=False)

# Print training accuracy after every 100 mini-batches.
# Evaluate the model on the test set after each epoch, and print the test loss and accuracy.
# train the model
num_epochs = 10

for epoch in range(num_epochs):
    running_loss = 0.0
    correct = 0
    total = 0

    for i, data in enumerate(train_loader, 0):
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # compute training accuracy
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        # print statistics
        running_loss += loss.item()
        if i % 100 == 99:    # print every 100 mini-batches
            print('[Epoch %d, Batch %5d] loss: %.3f, accuracy: %.3f' %
                  (epoch + 1, i + 1, running_loss / 100, 100 * correct / total))
            running_loss = 0.0
            correct = 0
            total = 0

    # evaluate on test set
    test_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for data in test_loader:
            inputs, labels = data
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            test_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    print('[Epoch %d] test loss: %.3f, accuracy: %.3f' %
          (epoch + 1, test_loss / len(test_loader), 100 * correct / total))

print('Finished Training')
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%|██████████| 170498071/170498071 [00:12<00:00, 14092848.81it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
[Epoch 1, Batch   100] loss: 2.041, accuracy: 25.300
[Epoch 1, Batch   200] loss: 1.672, accuracy: 39.170
[Epoch 1, Batch   300] loss: 1.558, accuracy: 44.570
[Epoch 1, Batch   400] loss: 1.479, accuracy: 46.960
[Epoch 1, Batch   500] loss: 1.420, accuracy: 49.600
[Epoch 1] test loss: 1.373, accuracy: 51.120
[Epoch 2, Batch   100] loss: 1.388, accuracy: 50.210
[Epoch 2, Batch   200] loss: 1.338, accuracy: 51.990
[Epoch 2, Batch   300] loss: 1.295, accuracy: 54.600
[Epoch 2, Batch   400] loss: 1.268, accuracy: 55.050
[Epoch 2, Batch   500] loss: 1.237, accuracy: 57.000
[Epoch 2] test loss: 1.208, accuracy: 57.470
[Epoch 3, Batch   100] loss: 1.186, accuracy: 58.480
[Epoch 3, Batch   200] loss: 1.165, accuracy: 58.770
[Epoch 3, Batch   300] loss: 1.148, accuracy: 59.880
[Epoch 3, Batch   400] loss: 1.143, accuracy: 59.510
[Epoch 3, Batch   500] loss: 1.145, accuracy: 60.020
[Epoch 3] test loss: 1.162, accuracy: 59.070
[Epoch 4, Batch   100] loss: 1.084, accuracy: 61.800
[Epoch 4, Batch   200] loss: 1.069, accuracy: 62.830
[Epoch 4, Batch   300] loss: 1.064, accuracy: 63.320
[Epoch 4, Batch   400] loss: 1.040, accuracy: 63.420
[Epoch 4, Batch   500] loss: 1.038, accuracy: 63.990
[Epoch 4] test loss: 1.042, accuracy: 63.850
[Epoch 5, Batch   100] loss: 0.981, accuracy: 65.550
[Epoch 5, Batch   200] loss: 0.987, accuracy: 65.630
[Epoch 5, Batch   300] loss: 1.001, accuracy: 65.060
[Epoch 5, Batch   400] loss: 0.996, accuracy: 65.500
[Epoch 5, Batch   500] loss: 0.992, accuracy: 65.410
[Epoch 5] test loss: 0.993, accuracy: 65.680
[Epoch 6, Batch   100] loss: 0.958, accuracy: 66.970
[Epoch 6, Batch   200] loss: 0.931, accuracy: 67.920
[Epoch 6, Batch   300] loss: 0.950, accuracy: 67.210
[Epoch 6, Batch   400] loss: 0.918, accuracy: 68.530
[Epoch 6, Batch   500] loss: 0.906, accuracy: 68.290
[Epoch 6] test loss: 0.958, accuracy: 67.040
[Epoch 7, Batch   100] loss: 0.873, accuracy: 69.800
[Epoch 7, Batch   200] loss: 0.887, accuracy: 69.640
[Epoch 7, Batch   300] loss: 0.869, accuracy: 70.210
[Epoch 7, Batch   400] loss: 0.891, accuracy: 68.530
[Epoch 7, Batch   500] loss: 0.885, accuracy: 69.510
[Epoch 7] test loss: 0.943, accuracy: 67.640
[Epoch 8, Batch   100] loss: 0.832, accuracy: 71.080
[Epoch 8, Batch   200] loss: 0.835, accuracy: 71.110
[Epoch 8, Batch   300] loss: 0.843, accuracy: 70.630
[Epoch 8, Batch   400] loss: 0.847, accuracy: 70.620
[Epoch 8, Batch   500] loss: 0.841, accuracy: 70.910
[Epoch 8] test loss: 0.911, accuracy: 68.210
[Epoch 9, Batch   100] loss: 0.799, accuracy: 72.730
[Epoch 9, Batch   200] loss: 0.802, accuracy: 72.240
[Epoch 9, Batch   300] loss: 0.799, accuracy: 71.790
[Epoch 9, Batch   400] loss: 0.799, accuracy: 72.050
[Epoch 9, Batch   500] loss: 0.808, accuracy: 72.270
[Epoch 9] test loss: 0.883, accuracy: 69.820
[Epoch 10, Batch   100] loss: 0.767, accuracy: 73.150
[Epoch 10, Batch   200] loss: 0.779, accuracy: 72.900
[Epoch 10, Batch   300] loss: 0.764, accuracy: 73.690
[Epoch 10, Batch   400] loss: 0.777, accuracy: 73.330
[Epoch 10, Batch   500] loss: 0.786, accuracy: 73.100
[Epoch 10] test loss: 0.898, accuracy: 69.570
Finished Training
```

MSE loss

In [29]:
```
# modify the training loop to use one-hot encoded targets
# define the model
model = MyCNN()

# define the loss function
criterion = nn.MSELoss()

# define the optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)

# load the CIFAR-10 dataset
train_data = CIFAR10(root='./data', train=True, transform=ToTensor(), download=True)
test_data = CIFAR10(root='./data', train=False, transform=ToTensor(), download=True)

# create data loaders
train_loader = DataLoader(train_data, batch_size=100, shuffle=True)
test_loader = DataLoader(test_data, batch_size=100, shuffle=False)

# train the model
num_epochs = 10

for epoch in range(num_epochs):
    running_loss = 0.0

    for i, data in enumerate(train_loader, 0):
        inputs, labels = data

        # convert labels to one-hot encoding
        labels_onehot = nn.functional.one_hot(labels, num_classes=10).float()

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = model(inputs)
        loss = criterion(outputs, labels_onehot)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 100 == 99:    # print every 100 mini-batches
            print('[Epoch %d, Batch %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 100))
            running_loss = 0.0

        # evaluate on test set
    test_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for data in test_loader:
            inputs, labels = data

            # convert labels to one-hot encoding
            labels_onehot = nn.functional.one_hot(labels, num_classes=10).float()

            outputs = model(inputs)
            loss = criterion(outputs, labels_onehot)
            test_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    print('[Epoch %d] test loss: %.3f, accuracy: %.3f' %
          (epoch + 1, test_loss / len(test_loader), 100 * correct / total))

print('Finished Training')
```

```
Files already downloaded and verified
Files already downloaded and verified
[Epoch 1, Batch   100] loss: 0.084
[Epoch 1, Batch   200] loss: 0.076
[Epoch 1, Batch   300] loss: 0.072
[Epoch 1, Batch   400] loss: 0.070
[Epoch 1, Batch   500] loss: 0.068
[Epoch 1] test loss: 0.067, accuracy: 52.570
[Epoch 2, Batch   100] loss: 0.066
[Epoch 2, Batch   200] loss: 0.064
[Epoch 2, Batch   300] loss: 0.062
[Epoch 2, Batch   400] loss: 0.062
[Epoch 2, Batch   500] loss: 0.062
[Epoch 2] test loss: 0.061, accuracy: 59.810
[Epoch 3, Batch   100] loss: 0.059
[Epoch 3, Batch   200] loss: 0.058
[Epoch 3, Batch   300] loss: 0.059
[Epoch 3, Batch   400] loss: 0.058
[Epoch 3, Batch   500] loss: 0.058
[Epoch 3] test loss: 0.057, accuracy: 64.390
[Epoch 4, Batch   100] loss: 0.056
[Epoch 4, Batch   200] loss: 0.056
[Epoch 4, Batch   300] loss: 0.055
[Epoch 4, Batch   400] loss: 0.055
[Epoch 4, Batch   500] loss: 0.055
[Epoch 4] test loss: 0.055, accuracy: 65.650
[Epoch 5, Batch   100] loss: 0.053
[Epoch 5, Batch   200] loss: 0.053
[Epoch 5, Batch   300] loss: 0.054
[Epoch 5, Batch   400] loss: 0.053
[Epoch 5, Batch   500] loss: 0.052
[Epoch 5] test loss: 0.054, accuracy: 65.820
[Epoch 6, Batch   100] loss: 0.052
[Epoch 6, Batch   200] loss: 0.052
[Epoch 6, Batch   300] loss: 0.051
[Epoch 6, Batch   400] loss: 0.052
[Epoch 6, Batch   500] loss: 0.051
[Epoch 6] test loss: 0.053, accuracy: 67.500
[Epoch 7, Batch   100] loss: 0.049
[Epoch 7, Batch   200] loss: 0.050
[Epoch 7, Batch   300] loss: 0.051
[Epoch 7, Batch   400] loss: 0.050
[Epoch 7, Batch   500] loss: 0.050
[Epoch 7] test loss: 0.052, accuracy: 68.770
[Epoch 8, Batch   100] loss: 0.049
[Epoch 8, Batch   200] loss: 0.049
[Epoch 8, Batch   300] loss: 0.049
[Epoch 8, Batch   400] loss: 0.049
[Epoch 8, Batch   500] loss: 0.049
[Epoch 8] test loss: 0.051, accuracy: 69.010
[Epoch 9, Batch   100] loss: 0.048
[Epoch 9, Batch   200] loss: 0.048
[Epoch 9, Batch   300] loss: 0.049
[Epoch 9, Batch   400] loss: 0.048
[Epoch 9, Batch   500] loss: 0.048
[Epoch 9] test loss: 0.050, accuracy: 70.290
[Epoch 10, Batch   100] loss: 0.047
[Epoch 10, Batch   200] loss: 0.047
[Epoch 10, Batch   300] loss: 0.048
[Epoch 10, Batch   400] loss: 0.048
[Epoch 10, Batch   500] loss: 0.048
[Epoch 10] test loss: 0.050, accuracy: 69.980
Finished Training
```

In [30]:
```
# The test accuracy bottle-necked at around 70%. This is because MSE loss is typically not used for
# classification tasks. Cross-entropy loss is a better choice.
```

2. Compare your models when you use Adam vs. RMSProp. You may use the cross entropy loss.

In [13]:
```
# define the models
model_adam = MyCNN()
model_rmsprop = MyCNN()

# define the loss function
criterion = nn.CrossEntropyLoss()

# define the optimizers
optimizer_adam = optim.Adam(model_adam.parameters(), lr=0.001)
optimizer_rmsprop = optim.RMSprop(model_rmsprop.parameters(), lr=0.001)

# load the CIFAR-10 dataset
train_data = CIFAR10(root='./data', train=True, transform=ToTensor(), download=True)
test_data = CIFAR10(root='./data', train=False, transform=ToTensor(), download=True)

# create data loaders
train_loader = DataLoader(train_data, batch_size=100, shuffle=True)
test_loader = DataLoader(test_data, batch_size=100, shuffle=False)

# train the Adam model
num_epochs = 10

for epoch in range(num_epochs):
    running_loss = 0.0

    for i, data in enumerate(train_loader, 0):
        inputs, labels = data

        # zero the parameter gradients
        optimizer_adam.zero_grad()

        # forward + backward + optimize
        outputs = model_adam(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer_adam.step()

        # print statistics
        running_loss += loss.item()
        if i % 100 == 99:    # print every 100 mini-batches
            print('[Adam, Epoch %d, Batch %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 100))
            running_loss = 0.0

    # evaluate on test set
    test_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for data in test_loader:
            inputs, labels = data
            outputs = model_adam(inputs)
            loss = criterion(outputs, labels)
            test_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    print('[Adam, Epoch %d] test loss: %.3f, accuracy: %.3f' %
          (epoch + 1, test_loss / len(test_loader), 100 * correct / total))

# train the RMSProp model
for epoch in range(num_epochs):
    running_loss = 0.0

    for i, data in enumerate(train_loader, 0):
        inputs, labels = data

        # zero the parameter gradients
        optimizer_rmsprop.zero_grad()

        # forward + backward + optimize
        outputs = model_rmsprop(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer_rmsprop.step()

        # print statistics
        running_loss += loss.item()
        if i % 100 == 99:    # print every 100 mini-batches
            print('[RMSProp, Epoch %d, Batch %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 100))
            running_loss = 0.0

    # evaluate on test set
    test_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for data in test_loader:
            inputs, labels = data
            outputs = model_rmsprop(inputs)
            loss = criterion(outputs, labels)
            test_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    print('[RMSProp, Epoch %d] test loss: %.3f, accuracy: %.3f' %
          (epoch + 1, test_loss / len(test_loader), 100 * correct / total))

# print final results
print('Adam - test accuracy: %.3f' % (100 * correct / total))
print('RMSProp - test accuracy: %.3f' % (100 * correct / total))
```

```
Files already downloaded and verified
Files already downloaded and verified
[Adam, Epoch 1, Batch   100] loss: 2.052
[Adam, Epoch 1, Batch   200] loss: 1.684
[Adam, Epoch 1, Batch   300] loss: 1.566
[Adam, Epoch 1, Batch   400] loss: 1.526
[Adam, Epoch 1, Batch   500] loss: 1.475
[Adam, Epoch 1] test loss: 1.427, accuracy: 48.390
[Adam, Epoch 2, Batch   100] loss: 1.398
[Adam, Epoch 2, Batch   200] loss: 1.360
[Adam, Epoch 2, Batch   300] loss: 1.334
[Adam, Epoch 2, Batch   400] loss: 1.293
[Adam, Epoch 2, Batch   500] loss: 1.271
[Adam, Epoch 2] test loss: 1.215, accuracy: 56.660
[Adam, Epoch 3, Batch   100] loss: 1.198
[Adam, Epoch 3, Batch   200] loss: 1.198
[Adam, Epoch 3, Batch   300] loss: 1.193
[Adam, Epoch 3, Batch   400] loss: 1.167
[Adam, Epoch 3, Batch   500] loss: 1.134
[Adam, Epoch 3] test loss: 1.124, accuracy: 60.640
[Adam, Epoch 4, Batch   100] loss: 1.087
[Adam, Epoch 4, Batch   200] loss: 1.087
[Adam, Epoch 4, Batch   300] loss: 1.086
[Adam, Epoch 4, Batch   400] loss: 1.059
[Adam, Epoch 4, Batch   500] loss: 1.020
[Adam, Epoch 4] test loss: 1.041, accuracy: 63.300
[Adam, Epoch 5, Batch   100] loss: 1.012
[Adam, Epoch 5, Batch   200] loss: 0.997
[Adam, Epoch 5, Batch   300] loss: 0.966
[Adam, Epoch 5, Batch   400] loss: 0.983
[Adam, Epoch 5, Batch   500] loss: 0.984
[Adam, Epoch 5] test loss: 0.969, accuracy: 66.530
[Adam, Epoch 6, Batch   100] loss: 0.913
[Adam, Epoch 6, Batch   200] loss: 0.963
[Adam, Epoch 6, Batch   300] loss: 0.926
[Adam, Epoch 6, Batch   400] loss: 0.924
[Adam, Epoch 6, Batch   500] loss: 0.923
[Adam, Epoch 6] test loss: 0.954, accuracy: 66.710
[Adam, Epoch 7, Batch   100] loss: 0.885
[Adam, Epoch 7, Batch   200] loss: 0.879
[Adam, Epoch 7, Batch   300] loss: 0.905
[Adam, Epoch 7, Batch   400] loss: 0.870
[Adam, Epoch 7, Batch   500] loss: 0.877
[Adam, Epoch 7] test loss: 0.917, accuracy: 68.400
[Adam, Epoch 8, Batch   100] loss: 0.848
[Adam, Epoch 8, Batch   200] loss: 0.858
[Adam, Epoch 8, Batch   300] loss: 0.841
[Adam, Epoch 8, Batch   400] loss: 0.851
[Adam, Epoch 8, Batch   500] loss: 0.851
[Adam, Epoch 8] test loss: 0.926, accuracy: 68.020
[Adam, Epoch 9, Batch   100] loss: 0.813
[Adam, Epoch 9, Batch   200] loss: 0.817
[Adam, Epoch 9, Batch   300] loss: 0.804
[Adam, Epoch 9, Batch   400] loss: 0.824
[Adam, Epoch 9, Batch   500] loss: 0.803
[Adam, Epoch 9] test loss: 0.885, accuracy: 69.450
[Adam, Epoch 10, Batch   100] loss: 0.777
[Adam, Epoch 10, Batch   200] loss: 0.771
[Adam, Epoch 10, Batch   300] loss: 0.803
[Adam, Epoch 10, Batch   400] loss: 0.785
[Adam, Epoch 10, Batch   500] loss: 0.796
[Adam, Epoch 10] test loss: 0.870, accuracy: 70.220
[RMSProp, Epoch 1, Batch   100] loss: 2.053
[RMSProp, Epoch 1, Batch   200] loss: 1.736
[RMSProp, Epoch 1, Batch   300] loss: 1.613
[RMSProp, Epoch 1, Batch   400] loss: 1.520
[RMSProp, Epoch 1, Batch   500] loss: 1.494
[RMSProp, Epoch 1] test loss: 1.482, accuracy: 46.480
[RMSProp, Epoch 2, Batch   100] loss: 1.418
[RMSProp, Epoch 2, Batch   200] loss: 1.384
[RMSProp, Epoch 2, Batch   300] loss: 1.352
[RMSProp, Epoch 2, Batch   400] loss: 1.334
[RMSProp, Epoch 2, Batch   500] loss: 1.304
[RMSProp, Epoch 2] test loss: 1.348, accuracy: 51.390
[RMSProp, Epoch 3, Batch   100] loss: 1.293
[RMSProp, Epoch 3, Batch   200] loss: 1.224
[RMSProp, Epoch 3, Batch   300] loss: 1.204
[RMSProp, Epoch 3, Batch   400] loss: 1.202
[RMSProp, Epoch 3, Batch   500] loss: 1.186
[RMSProp, Epoch 3] test loss: 1.160, accuracy: 58.940
[RMSProp, Epoch 4, Batch   100] loss: 1.146
[RMSProp, Epoch 4, Batch   200] loss: 1.144
[RMSProp, Epoch 4, Batch   300] loss: 1.114
[RMSProp, Epoch 4, Batch   400] loss: 1.112
[RMSProp, Epoch 4, Batch   500] loss: 1.098
[RMSProp, Epoch 4] test loss: 1.108, accuracy: 61.030
[RMSProp, Epoch 5, Batch   100] loss: 1.068
[RMSProp, Epoch 5, Batch   200] loss: 1.070
[RMSProp, Epoch 5, Batch   300] loss: 1.046
[RMSProp, Epoch 5, Batch   400] loss: 1.054
[RMSProp, Epoch 5, Batch   500] loss: 1.026
[RMSProp, Epoch 5] test loss: 1.068, accuracy: 62.490
[RMSProp, Epoch 6, Batch   100] loss: 0.999
[RMSProp, Epoch 6, Batch   200] loss: 1.018
[RMSProp, Epoch 6, Batch   300] loss: 0.998
[RMSProp, Epoch 6, Batch   400] loss: 0.982
[RMSProp, Epoch 6, Batch   500] loss: 0.975
[RMSProp, Epoch 6] test loss: 0.997, accuracy: 65.300
[RMSProp, Epoch 7, Batch   100] loss: 0.946
[RMSProp, Epoch 7, Batch   200] loss: 0.943
[RMSProp, Epoch 7, Batch   300] loss: 0.946
[RMSProp, Epoch 7, Batch   400] loss: 0.967
[RMSProp, Epoch 7, Batch   500] loss: 0.928
[RMSProp, Epoch 7] test loss: 0.983, accuracy: 65.960
[RMSProp, Epoch 8, Batch   100] loss: 0.918
[RMSProp, Epoch 8, Batch   200] loss: 0.910
[RMSProp, Epoch 8, Batch   300] loss: 0.912
[RMSProp, Epoch 8, Batch   400] loss: 0.901
[RMSProp, Epoch 8, Batch   500] loss: 0.902
[RMSProp, Epoch 8] test loss: 1.015, accuracy: 64.800
[RMSProp, Epoch 9, Batch   100] loss: 0.865
[RMSProp, Epoch 9, Batch   200] loss: 0.873
[RMSProp, Epoch 9, Batch   300] loss: 0.863
[RMSProp, Epoch 9, Batch   400] loss: 0.883
[RMSProp, Epoch 9, Batch   500] loss: 0.878
[RMSProp, Epoch 9] test loss: 0.944, accuracy: 67.630
[RMSProp, Epoch 10, Batch   100] loss: 0.822
[RMSProp, Epoch 10, Batch   200] loss: 0.825
[RMSProp, Epoch 10, Batch   300] loss: 0.849
[RMSProp, Epoch 10, Batch   400] loss: 0.849
[RMSProp, Epoch 10, Batch   500] loss: 0.859
[RMSProp, Epoch 10] test loss: 0.904, accuracy: 68.870
Adam - test accuracy: 68.870
RMSProp - test accuracy: 68.870
```

3. Compare the performance with ReLU vs sigmoid activation with cross-entropy loss and Adam

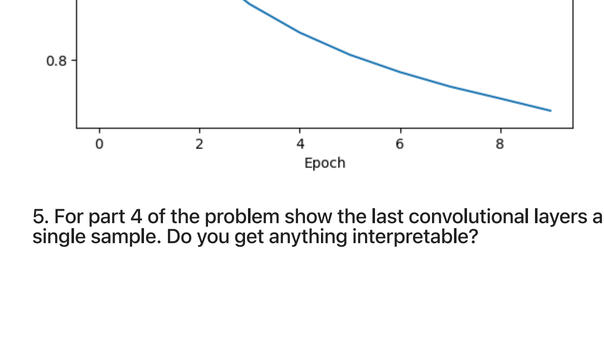
In [9]:
```
import torchvision.transforms as transforms

# Define the CNN architecture
class MyCNN(nn.Module):
    def __init__(self, activation):
        super(MyCNN, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.fc1 = nn.Linear(in_features=64*4*4, out_features=10)

        if activation == 'relu':
            self.activation = nn.ReLU()
        elif activation == 'sigmoid':
            self.activation = nn.Sigmoid()

    def forward(self, x):
        x = self.conv1(x)
        x = self.activation(x)
        x = self.pool1(x)

        x = self.conv2(x)
        x = self.activation(x)
        x = self.pool2(x)

        x = self.conv3(x)
        x = self.activation(x)
        x = self.pool3(x)

        x = x.view(-1, 64*4*4)
        x = self.fc1(x)

        return x

# Set the device to GPU if available
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Define the data transforms and load the CIFAR10 dataset
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5, 0.5, 0.5), (0.5,
trainset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=100, shuffle=True)
testset = datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=100, shuffle=False)

# Define the CNN model with ReLU activation and train it
cnn_relu = MyCNN('relu').to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(cnn_relu.parameters())
for epoch in range(10):
    running_loss = 0.0
    correct_train = 0
    total_train = 0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data[0].to(device), data[1].to(device)
        optimizer.zero_grad()
        outputs = cnn_relu(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total_train += labels.size(0)
        correct_train += (predicted == labels).sum().item()
        if i % 100 == 99:    # print every 100 mini-batches
            print('[Epoch %d, Batch %5d] loss: %.3f, accuracy: %.3f' %
                  (epoch + 1, i + 1, running_loss / 100, 100 * correct_train / total_train))
            running_loss = 0.0
            correct_train = 0
            total_train = 0

    # evaluate on test set
    test_loss = 0.0
    correct_test = 0
    total_test = 0
    with torch.no_grad():
        for data in testloader:
            inputs, labels = data[0].to(device), data[1].to(device)
            outputs = cnn_relu(inputs)
            loss = criterion(outputs, labels)
            test_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total_test += labels.size(0)
            correct_test += (predicted == labels).sum().item()
    print('[Epoch %d] test loss: %.3f, accuracy: %.3f' %
          (epoch + 1, test_loss / len(testloader), 100 * correct_test / total_test))

print('Finished Training')
```

```
Files already downloaded and verified
Files already downloaded and verified
[Epoch 1, Batch   100] loss: 1.923, accuracy: 30.950
[Epoch 1, Batch   200] loss: 1.573, accuracy: 43.080
[Epoch 1, Batch   300] loss: 1.442, accuracy: 48.080
[Epoch 1, Batch   400] loss: 1.389, accuracy: 50.660
[Epoch 1, Batch   500] loss: 1.316, accuracy: 52.990
[Epoch 1] test loss: 1.275, accuracy: 54.710
[Epoch 2, Batch   100] loss: 1.231, accuracy: 55.770
[Epoch 2, Batch   200] loss: 1.223, accuracy: 57.160
[Epoch 2, Batch   300] loss: 1.179, accuracy: 58.360
[Epoch 2, Batch   400] loss: 1.140, accuracy: 60.400
[Epoch 2, Batch   500] loss: 1.118, accuracy: 59.980
[Epoch 2] test loss: 1.136, accuracy: 59.750
[Epoch 3, Batch   100] loss: 1.069, accuracy: 62.690
[Epoch 3, Batch   200] loss: 1.045, accuracy: 63.230
[Epoch 3, Batch   300] loss: 1.041, accuracy: 63.860
[Epoch 3, Batch   400] loss: 0.993, accuracy: 65.470
[Epoch 3, Batch   500] loss: 1.002, accuracy: 65.010
[Epoch 3] test loss: 1.024, accuracy: 64.120
[Epoch 4, Batch   100] loss: 0.942, accuracy: 67.700
[Epoch 4, Batch   200] loss: 0.953, accuracy: 66.480
[Epoch 4, Batch   300] loss: 0.946, accuracy: 67.370
[Epoch 4, Batch   400] loss: 0.919, accuracy: 68.030
[Epoch 4, Batch   500] loss: 0.905, accuracy: 68.470
[Epoch 4] test loss: 0.937, accuracy: 67.350
[Epoch 5, Batch   100] loss: 0.871, accuracy: 69.930
[Epoch 5, Batch   200] loss: 0.873, accuracy: 69.990
[Epoch 5, Batch   300] loss: 0.869, accuracy: 69.650
[Epoch 5, Batch   400] loss: 0.840, accuracy: 71.340
[Epoch 5, Batch   500] loss: 0.849, accuracy: 70.630
[Epoch 5] test loss: 0.888, accuracy: 69.320
[Epoch 6, Batch   100] loss: 0.822, accuracy: 71.470
[Epoch 6, Batch   200] loss: 0.804, accuracy: 71.860
[Epoch 6, Batch   300] loss: 0.804, accuracy: 72.260
[Epoch 6, Batch   400] loss: 0.805, accuracy: 72.400
[Epoch 6, Batch   500] loss: 0.801, accuracy: 72.500
[Epoch 6] test loss: 0.868, accuracy: 69.940
[Epoch 7, Batch   100] loss: 0.766, accuracy: 73.670
[Epoch 7, Batch   200] loss: 0.760, accuracy: 74.010
[Epoch 7, Batch   300] loss: 0.757, accuracy: 73.660
[Epoch 7, Batch   400] loss: 0.751, accuracy: 74.440
[Epoch 7, Batch   500] loss: 0.758, accuracy: 74.150
[Epoch 7] test loss: 0.874, accuracy: 69.750
[Epoch 8, Batch   100] loss: 0.707, accuracy: 76.200
[Epoch 8, Batch   200] loss: 0.715, accuracy: 75.120
[Epoch 8, Batch   300] loss: 0.728, accuracy: 74.540
[Epoch 8, Batch   400] loss: 0.713, accuracy: 74.840
[Epoch 8, Batch   500] loss: 0.730, accuracy: 74.940
[Epoch 8] test loss: 0.838, accuracy: 70.990
[Epoch 9, Batch   100] loss: 0.676, accuracy: 76.420
[Epoch 9, Batch   200] loss: 0.677, accuracy: 76.570
[Epoch 9, Batch   300] loss: 0.676, accuracy: 76.920
[Epoch 9, Batch   400] loss: 0.701, accuracy: 76.110
[Epoch 9, Batch   500] loss: 0.673, accuracy: 77.110
[Epoch 9] test loss: 0.814, accuracy: 72.660
[Epoch 10, Batch   100] loss: 0.643, accuracy: 78.020
[Epoch 10, Batch   200] loss: 0.650, accuracy: 77.770
[Epoch 10, Batch   300] loss: 0.665, accuracy: 76.750
[Epoch 10, Batch   400] loss: 0.680, accuracy: 76.430
[Epoch 10, Batch   500] loss: 0.647, accuracy: 77.560
[Epoch 10] test loss: 0.825, accuracy: 72.390
Finished Training
```

4: Plot your training error as vs the number of epochs as for cross-entropy loss with ADam optimizer and ReLU activation.

In [23]:
```
# epochs=10 , Adam optimizer, ReLU activation
# Plot the training loss (y-axis) vs. of the number of epochs (x-axis)
# Define the CNN model with ReLU activation and train it
cnn_relu = MyCNN('relu').to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(cnn_relu.parameters())
train_loss = []
for epoch in range(10):
    running_loss = 0.0
    correct_train = 0
    total_train = 0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data[0].to(device), data[1].to(device)
        optimizer.zero_grad()
        outputs = cnn_relu(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total_train += labels.size(0)
        correct_train += (predicted == labels).sum().item()
    train_loss.append(running_loss / len(trainloader))
    print('Epoch %d: training loss = %f' % (epoch + 1, train_loss[-1]))

# Plot the training loss as a function of the number of epochs
plt.plot(range(10), train_loss)
plt.xlabel('Epoch')
plt.ylabel('Training loss')
plt.title('Training loss as a function of the number of epochs')
plt.show()
```

```
Epoch 1: training loss = 1.563132
Epoch 2: training loss = 1.217679
Epoch 3: training loss = 1.055168
Epoch 4: training loss = 0.949653
Epoch 5: training loss = 0.873371
Epoch 6: training loss = 0.813927
Epoch 7: training loss = 0.767437
Epoch 8: training loss = 0.728658
Epoch 9: training loss = 0.696770
Epoch 10: training loss = 0.664112
```



5. For part 4 of the problem show the last convolutional layers as images for a single sample. Do you get anything interpretable?

Problem 5: U Auto-encoder on Fashion MNIST

In [1]:
```python
# Step 1: Download the Fashion MNIST data using PyTorch

import torch
from torchvision import datasets, transforms
import torch.nn as nn
import numpy as np

# Define the transformations to be applied to the data
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# Download the Fashion MNIST training dataset
train_data = datasets.FashionMNIST(root='./data', train=True, download=True, transform=transform)

# Create a dataloader to load the data in batches
batch_size = 64
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)
```

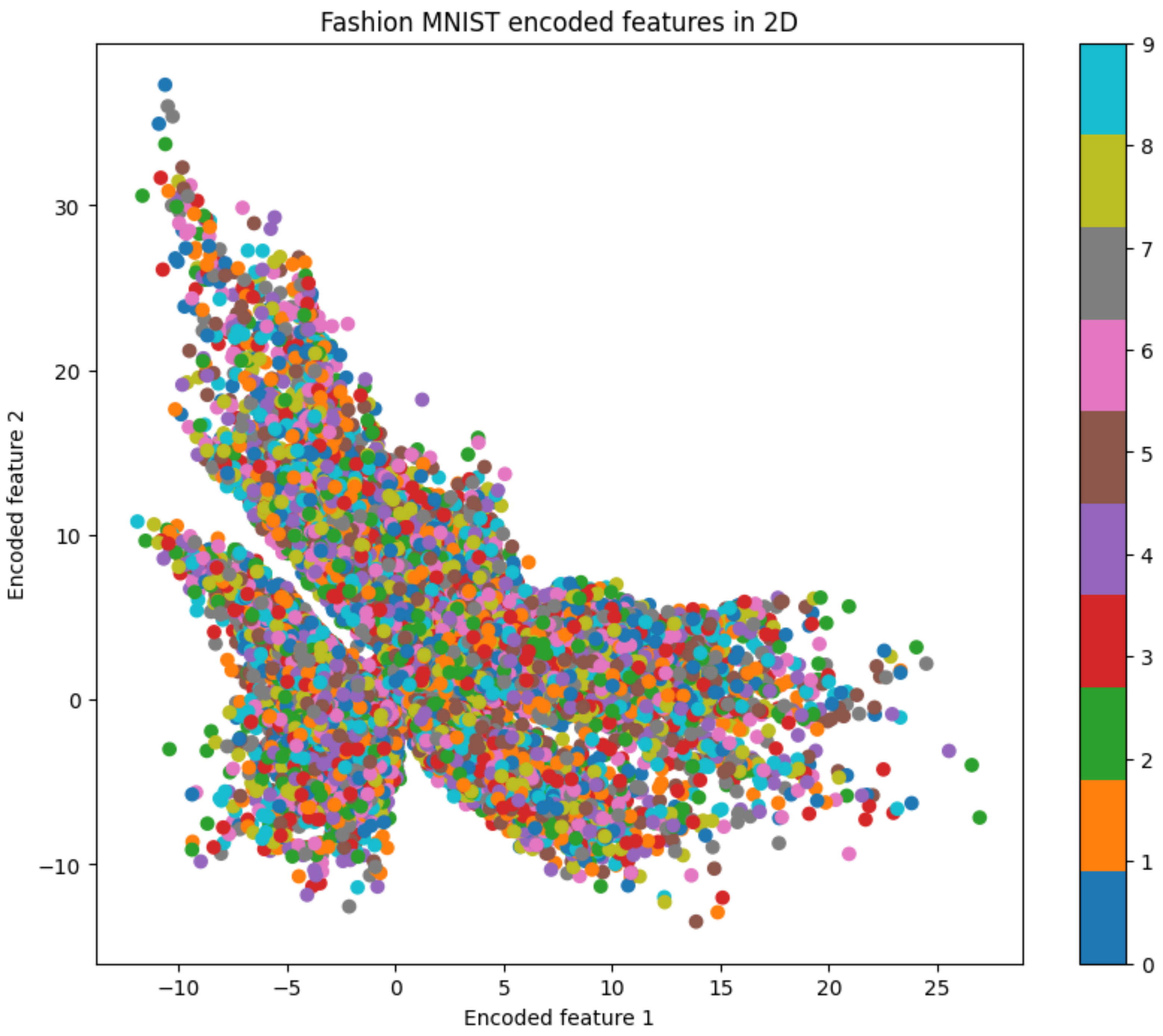
```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to
./data/FashionMNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 26421880/26421880 [00:01<00:00, 15987116.55it/s]
Extracting ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to
./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 29515/29515 [00:00<00:00, 273917.03it/s]
Extracting ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to
./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
100%|██████████| 4422102/4422102 [00:00<00:00, 5078377.16it/s]
Extracting ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to
./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████| 5148/5148 [00:00<00:00, 8032841.14it/s]
Extracting ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw
```

In [2]:
```python
# Step 2: Use an auto-encoder to reduce the dimension to two
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(28*28, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 2)
        )
        self.decoder = nn.Sequential(
            nn.Linear(2, 64),
            nn.ReLU(),
            nn.Linear(64, 128),
            nn.ReLU(),
            nn.Linear(128, 28*28),
            nn.Tanh()
        )

    def forward(self, x):
        x = x.view(x.size(0), -1)
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        decoded = decoded.view(decoded.size(0), 1, 28, 28)
        return encoded, decoded
```

In [3]:
```python
# Device to be used (GPU if available, else CPU),
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Define the autoencoder and move it to the device
autoencoder = Autoencoder().to(device)

# Define the loss function (MSE loss) and optimizer (Adam)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(autoencoder.parameters(), lr=0.001)

# Train the autoencoder
num_epochs = 20
for epoch in range(num_epochs):
    for data in train_loader:
        img, _ = data
        img = img.to(device)

        # Forward pass
        encoded, decoded = autoencoder(img)
        loss = criterion(decoded, img)

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs, loss.item()))

# Get the encoded features for the training data
encoded_train = []
with torch.no_grad():
    for data in train_loader:
        img, label = data
        img = img.to(device)
        encoded, decoded = autoencoder(img)
        encoded_train.append(encoded.cpu().numpy())
encoded_train = np.concatenate(encoded_train) # concatenate all the encoded features into a single
```

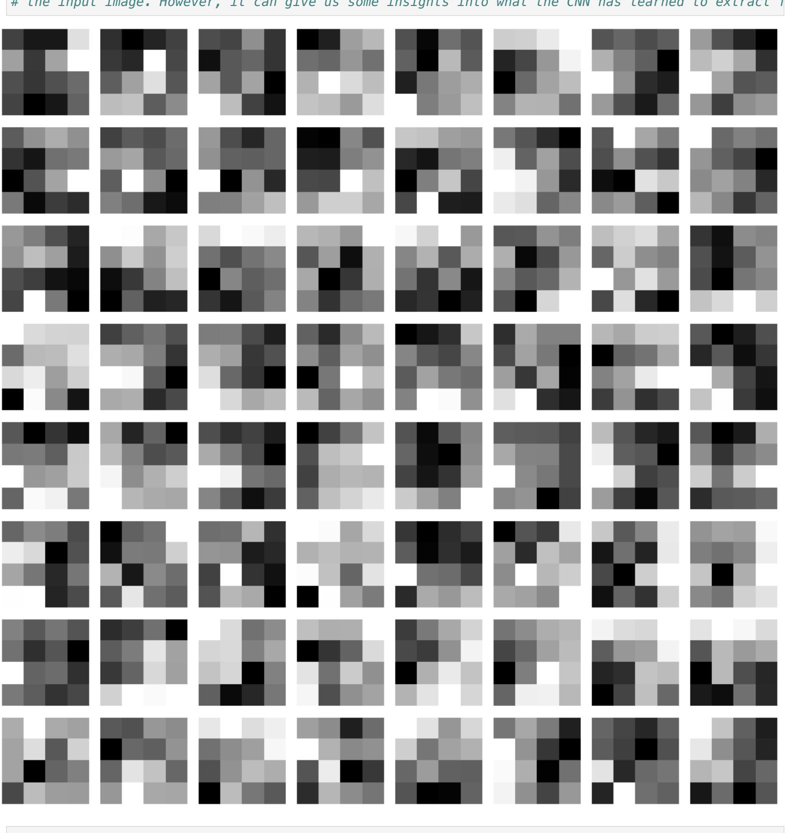
```
Using device: cuda
Epoch [1/20], Loss: 0.1238
Epoch [2/20], Loss: 0.1061
Epoch [3/20], Loss: 0.1094
Epoch [4/20], Loss: 0.1154
Epoch [5/20], Loss: 0.1119
Epoch [6/20], Loss: 0.1194
Epoch [7/20], Loss: 0.0930
Epoch [8/20], Loss: 0.1106
Epoch [9/20], Loss: 0.1174
Epoch [10/20], Loss: 0.0977
Epoch [11/20], Loss: 0.0979
Epoch [12/20], Loss: 0.0993
Epoch [13/20], Loss: 0.1013
Epoch [14/20], Loss: 0.1006
Epoch [15/20], Loss: 0.0880
Epoch [16/20], Loss: 0.1012
Epoch [17/20], Loss: 0.0993
Epoch [18/20], Loss: 0.0980
Epoch [19/20], Loss: 0.1111
Epoch [20/20], Loss: 0.0940
```

In [4]:
```python
# Step 3: Plot the points in the reduced feature space with color-coded class labels
import matplotlib.pyplot as plt
import numpy as np

# Get the class labels for the training data
labels_train = train_data.targets.numpy()

# Plot the encoded features in 2D with color-coded class labels
plt.figure(figsize=(10, 8))
plt.scatter(encoded_train[:, 0], encoded_train[:, 1], c=labels_train, cmap='tab10')
plt.colorbar()
plt.title('Fashion MNIST encoded features in 2D')
plt.xlabel('Encoded feature 1')
plt.ylabel('Encoded feature 2')
plt.show()

# The x-axis represents the first encoded feature, the y-axis represents the second encoded feature
# The color-coded class labels
```


Fashion MNIST encoded features in 2D
Encoded feature 2
Encoded feature 1
30
20
10
0
−10
−10
−5
0
5
10
15
20
25
9
8
7
6
5
4
3
2
1
0

In [26]:

```python
# Define the CNN model with ReLU activation and extract the last convolutional layers
cnn_relu = MyCNN('relu').to(device)
cnn_layers = nn.Sequential(*list(cnn_relu.children())[:-2])

# Retrieve a single sample from the test set
sample, label = next(iter(testloader))
sample = sample.to(device)
label = label.to(device)

# Apply the CNN layers to the sample and get the output of the last convolutional layers
features = cnn_layers(sample)
features_np = features.cpu().detach().numpy()

# Plot the output of the last convolutional layers as images
fig, axs = plt.subplots(8, 8, figsize=(10, 10))
for i in range(8):
    for j in range(8):
        axs[i, j].imshow(features_np[0, i*8+j, :, :], cmap='gray')
        axs[i, j].axis('off')
plt.tight_layout()
plt.show()

# The output of the last convolutional layers is not interpretable, as it represents a high-level f
# the input image. However, it can give us some insights into what the CNN has learned to extract f
```



In [ ]:

Problem 6: U GANS vs VAEs on Fashion MNIST

In [1]:
```
import torch
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
import random
import numpy as np
```

In [2]:
```
# Train a vanilla GAN and Variational Autoencoder (VAE) on the Fashion MNIST dataset using PyTorch.
# We use a simple architecture consisting of fully connected layers for both the generator and disc
# and fully connected layers for the encoder and decoder of the VAE.

# Set seed for reproducibility
random.seed(0)
torch.manual_seed(0)

# Set device to GPU if available, otherwise use CPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Define transform to normalize the data
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5,), std=(0.5,))
])

# Load Fashion MNIST dataset
full_dataset = dsets.FashionMNIST(root='./data', train=True, transform=transform, download=True)

# Sample a subset of 30000 examples from the full dataset for GAN model training
subset_indices = random.sample(range(len(full_dataset)), 30000)
subset = torch.utils.data.Subset(full_dataset, subset_indices)
train_loader = torch.utils.data.DataLoader(subset, batch_size=64, shuffle=True)

# Next, define the generator and discriminator for the GAN
# Define the generator for the GAN
class Generator(nn.Module):
    def __init__(self, latent_dim, img_shape):
        super(Generator, self).__init__()
        self.img_shape = img_shape
        self.model = nn.Sequential(
            nn.Linear(latent_dim, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 1024),
            nn.LeakyReLU(0.2),
            nn.Linear(1024, int(np.prod(img_shape))),
            nn.Tanh()
        )

    def forward(self, z):
        img = self.model(z)
        img = img.view(img.size(0), *self.img_shape)
        return img

# Define the discriminator for the GAN
class Discriminator(nn.Module):
    def __init__(self, img_shape):
        super(Discriminator, self).__init__()
        self.img_shape = img_shape
        self.model = nn.Sequential(
            nn.Linear(int(np.prod(img_shape)), 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, img):
        img_flat = img.view(img.size(0), -1)
        validity = self.model(img_flat)
        return validity

#### define the encoder and decoder for the VAE:
# Define the encoder for the VAE
class Encoder(nn.Module):
    def __init__(self, latent_dim, img_shape):
        super(Encoder, self).__init__()
        self.img_shape = img_shape
        self.latent_dim = latent_dim
        self.model = nn.Sequential(
            nn.Linear(int(np.prod(img_shape)), 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
        )
        self.mean = nn.Linear(256, latent_dim)
        self.logvar = nn.Linear(256, latent_dim)

    def forward(self, img):
        img_flat = img.view(img.size(0), -1)
        x = self.model(img_flat)
        mean = self.mean(x)
        logvar = self.logvar(x)
        return mean, logvar

# Define the decoder for the VAE
class Decoder(nn.Module):
    def __init__(self, latent_dim, img_shape):
        super(Decoder, self).__init__()
        self.img_shape = img_shape
        self.latent_dim = latent_dim
        self.model = nn.Sequential(
            nn.Linear(latent_dim, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, int(np.prod(img_shape))),
            nn.Tanh()
        )

    def forward(self, z):
        img = self.model(z)
        img = img.view(img.size(0), *self.img_shape)
        return img

# Now, define the loss functions and optimizers for both the GAN and VAE
# Define the loss function for the GAN
adversarial_loss = nn.BCELoss()

# Define the optimizers for the GAN
generator = Generator(latent_dim=100, img_shape=(1, 28, 28)).to(device)
discriminator = Discriminator(img_shape=(1, 28, 28)).to(device)
optimizer_G = torch.optim.Adam(generator.parameters(), lr=0.0002, betas=(0.5, 0.999))
optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=0.0002, betas=(0.5, 0.999))

# Define the loss function for the VAE
def vae_loss(recon_x, x, mu, logvar):
    recon_loss = nn.functional.binary_cross_entropy(recon_x, x, reduction='sum')
    kl_divergence = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return recon_loss + kl_divergence

# Define the optimizers for the VAE
encoder = Encoder(latent_dim=20, img_shape=(1, 28, 28)).to(device)
decoder = Decoder(latent_dim=20, img_shape=(1, 28, 28)).to(device)
optimizer_E = torch.optim.Adam(encoder.parameters(), lr=0.0002, betas=(0.5, 0.999))
optimizer_D = torch.optim.Adam(decoder.parameters(), lr=0.0002, betas=(0.5, 0.999))
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 26421880/26421880 [00:01<00:00, 17426852.33it/s]
Extracting ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 29515/29515 [00:00<00:00, 295813.24it/s]
Extracting ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
100%|██████████| 4422102/4422102 [00:00<00:00, 5507208.71it/s]
Extracting ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████| 5148/5148 [00:00<00:00, 22398627.59it/s]
Extracting ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw
```

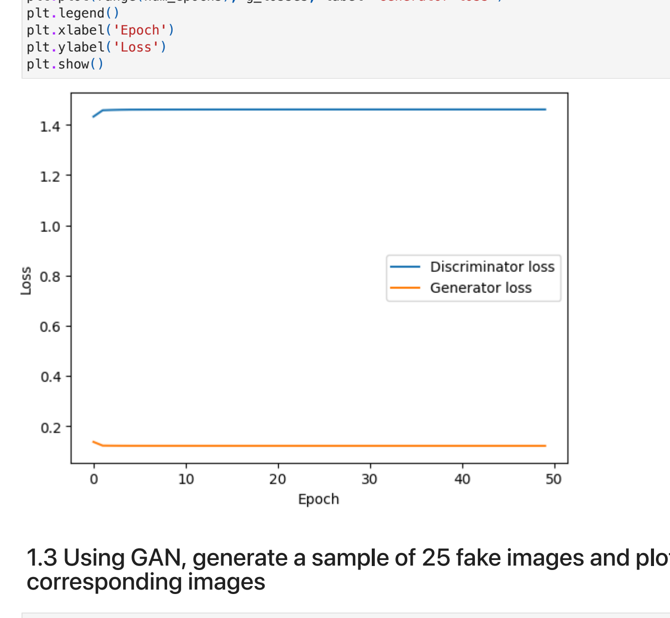
1. GAN

In [3]:
```
# Train the GAN model, save the model weights every 5 epochs into the 'gan_weights' folder.
# Then, save the final model weights after training is complete.

import os
# Create a folder to save the model weights
folder_path = 'gan_weights'
if not os.path.exists(folder_path):
    os.makedirs(folder_path)

# Set the number of epochs and batch size
num_epochs = 50
batch_size = 1024

# Initialize lists to store losses
d_losses = []
g_losses = []

# Train the GAN
for epoch in range(num_epochs):
    d_loss_epoch = 0.0
    g_loss_epoch = 0.0
    for i, (imgs, _) in enumerate(train_loader):
        # Adversarial ground truths
        valid = torch.ones((batch_size, 1)).to(device)
        fake = torch.zeros((batch_size, 1)).to(device)

        # Train the discriminator
        optimizer_D.zero_grad()
        real_imgs = imgs.to(device)
        fake_imgs = generator(torch.randn(batch_size, 100).to(device))
        real_loss = adversarial_loss(discriminator(real_imgs), valid[:real_imgs.size(0), :])
        fake_loss = adversarial_loss(discriminator(fake_imgs.detach()), fake[:fake_imgs.size(0), :]
        d_loss = (real_loss + fake_loss) / 2
        d_loss.backward()
        optimizer_D.step()

        # Train the generator
        optimizer_G.zero_grad()
        fake_imgs = generator(torch.randn(batch_size, 100).to(device))
        g_loss = adversarial_loss(discriminator(fake_imgs), valid[:fake_imgs.size(0), :])
        g_loss.backward()
        optimizer_G.step()

        # Update epoch losses
        d_loss_epoch += d_loss.item()
        g_loss_epoch += g_loss.item()

    # Compute average losses for epoch
    d_loss_epoch /= len(train_loader)
    g_loss_epoch /= len(train_loader)

    # Append losses to lists
    d_losses.append(d_loss_epoch)
    g_losses.append(g_loss_epoch)

    # Print training progress at every epoch
    print(f"[Epoch {epoch}/{num_epochs}] [D loss: {d_loss_epoch}] [G loss: {g_loss_epoch}]")

    # Save model weights every 5 epochs into 'gan_weights' folder
    if epoch % 5 == 0:
        torch.save({
            'generator': generator.state_dict(),
            'discriminator': discriminator.state_dict(),
            'optimizer_G': optimizer_G.state_dict(),
            'optimizer_D': optimizer_D.state_dict(),
        }, os.path.join(folder_path, f'gan_checkpoint_epoch_{epoch}.pth'))

# Save final model weights into 'gan_weights' folder
torch.save({
    'generator': generator.state_dict(),
    'discriminator': discriminator.state_dict(),
    'optimizer_G': optimizer_G.state_dict(),
    'optimizer_D': optimizer_D.state_dict(),
}, os.path.join(folder_path, 'gan_final_weights.pth'))
```

```
[Epoch 0/50] [D loss: 1.4324892055251197] [G loss: 0.13682059367010588]
[Epoch 1/50] [D loss: 1.4577669050139406] [G loss: 0.12192705322875143]
[Epoch 2/50] [D loss: 1.4590366120531615] [G loss: 0.12159822017017967]
[Epoch 3/50] [D loss: 1.4598622777060406] [G loss: 0.12138495058901529]
[Epoch 4/50] [D loss: 1.460305122678468] [G loss: 0.12127082966474582]
[Epoch 5/50] [D loss: 1.4604163271531876] [G loss: 0.12124184120311411]
[Epoch 6/50] [D loss: 1.4605141720537946] [G loss: 0.12121677559131244]
[Epoch 7/50] [D loss: 1.460587620989346] [G loss: 0.1211980123922769]
[Epoch 8/50] [D loss: 1.4606484421280657] [G loss: 0.12118188565502901]
[Epoch 9/50] [D loss: 1.460689204842297] [G loss: 0.12117108500906146]
[Epoch 10/50] [D loss: 1.460734852087269] [G loss: 0.12115970157039191]
[Epoch 11/50] [D loss: 1.4607731253861873] [G loss: 0.1211500847771732]
[Epoch 12/50] [D loss: 1.4608083518583383] [G loss: 0.12114163782042481]
[Epoch 13/50] [D loss: 1.4608364949094208] [G loss: 0.12113421071948273]
[Epoch 14/50] [D loss: 1.4608568151368262] [G loss: 0.121128178910533]
[Epoch 15/50] [D loss: 1.4608839681661967] [G loss: 0.12112180091170614]
[Epoch 16/50] [D loss: 1.4608997800456944] [G loss: 0.12111744624592348]
[Epoch 17/50] [D loss: 1.460917717866552] [G loss: 0.12111258263717582]
[Epoch 18/50] [D loss: 1.4609325923390988] [G loss: 0.12110889599775571]
[Epoch 19/50] [D loss: 1.4609464386632955] [G loss: 0.12110515459895388]
[Epoch 20/50] [D loss: 1.4609592179499709] [G loss: 0.1211022239313451]
[Epoch 21/50] [D loss: 1.4609683955402009] [G loss: 0.12109959476601595]
[Epoch 22/50] [D loss: 1.460978947468658] [G loss: 0.12109695085838659]
[Epoch 23/50] [D loss: 1.4609881514933571] [G loss: 0.12109501172166898]
[Epoch 24/50] [D loss: 1.4609949421018427] [G loss: 0.12109289281785107]
[Epoch 25/50] [D loss: 1.4610021050804969] [G loss: 0.12109133671087496]
[Epoch 26/50] [D loss: 1.4610086631164876] [G loss: 0.12108968544616373]
[Epoch 27/50] [D loss: 1.461011698759441] [G loss: 0.12108859963127291]
[Epoch 28/50] [D loss: 1.461016590915509] [G loss: 0.12108746809618813]
[Epoch 29/50] [D loss: 1.4610179276354531] [G loss: 0.12108667417248684]
[Epoch 30/50] [D loss: 1.4610261312171595] [G loss: 0.1210855711529504]
[Epoch 31/50] [D loss: 1.4610282591919401] [G loss: 0.12108466264281446]
[Epoch 32/50] [D loss: 1.4610280344989508] [G loss: 0.1210840764616344]
[Epoch 33/50] [D loss: 1.461031642549836] [G loss: 0.1210834397467723]
[Epoch 34/50] [D loss: 1.461035011673787] [G loss: 0.12108271870849484]
[Epoch 35/50] [D loss: 1.461035698715812] [G loss: 0.12108232312873482]
[Epoch 36/50] [D loss: 1.4610373584954723] [G loss: 0.12108188294080784]
[Epoch 37/50] [D loss: 1.461040060657428] [G loss: 0.12108152092837576]
[Epoch 38/50] [D loss: 1.4610415727599089] [G loss: 0.12108102496435393]
[Epoch 39/50] [D loss: 1.4610442202736829] [G loss: 0.12108044032412552]
[Epoch 40/50] [D loss: 1.4610421690605342] [G loss: 0.1210804258042307]
[Epoch 41/50] [D loss: 1.461047242953579] [G loss: 0.12107967261248814]
[Epoch 42/50] [D loss: 1.461047676580547] [G loss: 0.12107943293890719]
[Epoch 43/50] [D loss: 1.4610490786241317] [G loss: 0.12107921577592902]
[Epoch 44/50] [D loss: 1.4610505625129] [G loss: 0.12107893254266365]
[Epoch 45/50] [D loss: 1.461050468975547] [G loss: 0.12107873383933293]
[Epoch 46/50] [D loss: 1.461050630632494] [G loss: 0.12107848994004955]
[Epoch 47/50] [D loss: 1.4610522655027507] [G loss: 0.12107825452394323]
[Epoch 48/50] [D loss: 1.4610517307131021] [G loss: 0.12107794903425266]
[Epoch 49/50] [D loss: 1.461054940721882] [G loss: 0.12107787985028996]
```

1.2 GAN loss plot

In [4]:
```
# Plot losses
import matplotlib.pyplot as plt

plt.plot(range(num_epochs), d_losses, label='Discriminator loss')
plt.plot(range(num_epochs), g_losses, label='Generator loss')
plt.legend()
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
```



1.3 Using GAN, generate a sample of 25 fake images and plot the corresponding images

In [7]:
```
# Load the saved model weights
checkpoint = torch.load(os.path.join(folder_path, 'gan_final_weights.pth'))
generator.load_state_dict(checkpoint['generator'])

# Generate a sample of 25 fake images
with torch.no_grad():
    z = torch.randn(25, 100).to(device)
    fake_imgs = generator(z)

# Convert the fake images to numpy arrays
fake_imgs = fake_imgs.cpu().numpy()

# Rescale the fake images from [-1, 1] to [0, 1]
fake_imgs = (fake_imgs + 1) / 2

# Plot the fake images
fig, axs = plt.subplots(5, 5, figsize=(8, 8))
for i in range(5):
    for j in range(5):
        axs[i, j].imshow(fake_imgs[i*5+j].reshape((28, 28)), cmap='gray')
        axs[i, j].axis('off')
plt.show()
```

In [1]:
```
## The vanilla GAN did not do a good job
## Fake images from the generator show that the GAN did not learn from the data. There can be sever
## 1) The generator is stuck in a local minimum.
## 2) There isn't a good balance between the generator and discriminator. The discriminator can be
# that no matter what the generator outputs, the discriminator is able to distinguish the real fron
# Therefore, the generator is unable to improve.
## 3) To improve, we may need to look into Wasserstein GAN (WGAN), and use regularization on the di
# that is too strong in order balance out the training progress.
```

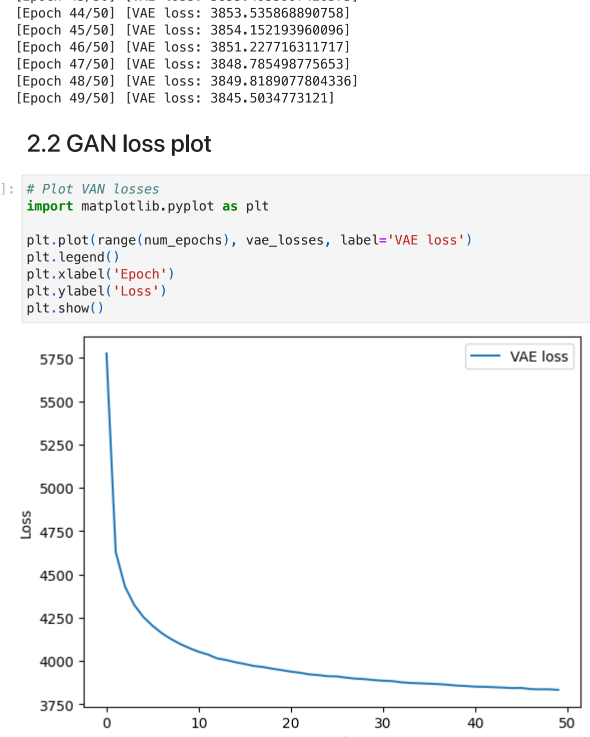
2. Variational Autoencoder

In [2]:
```
# The VAE architecture consists of an encoder with two fully connected layers and a decoder with tw
# The encoder outputs the mean and log variance of the latent variable, which are used to compute t
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

# Define the VAE architecture
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.fc21 = nn.Linear(512, 20)
        self.fc22 = nn.Linear(512, 20)

    def forward(self, x):
        x = x.view(-1, 784)
        x = F.relu(self.fc1(x))
        mu = self.fc21(x)
        logvar = self.fc22(x)
        return mu, logvar

class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.fc1 = nn.Linear(20, 512)
        self.fc2 = nn.Linear(512, 784)

    def forward(self, z):
        z = F.relu(self.fc1(z))
        x_hat = torch.tanh(self.fc2(z))
        return x_hat

class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()
        self.encoder = Encoder()
        self.decoder = Decoder()

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

    def forward(self, x):
        mu, logvar = self.encoder(x)
        z = self.reparameterize(mu, logvar)
        x_hat = self.decoder(z)
        return x_hat, mu, logvar

def vae_loss(x_hat, x, mu, logvar):
    mse_loss = F.mse_loss(x_hat, x.view(-1, 784), reduction='sum')
    kl_loss = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return mse_loss + kl_loss

# Set device to GPU if available, otherwise use CPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Define transform to normalize the data
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5,), std=(0.5,))
])

# Load Fashion MNIST dataset
train_dataset = dsets.FashionMNIST(root='./data', train=True, transform=transform, download=True)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

# Define the VAE model, optimizer, and loss function
vae = VAE().to(device)
optimizer_E = optim.Adam(vae.encoder.parameters(), lr=0.001)
optimizer_D = optim.Adam(vae.decoder.parameters(), lr=0.001)

# Train the VAE model
num_epochs = 50
vae_losses = []
for epoch in range(num_epochs):
    vae_loss_epoch = 0.0
    for i, (imgs, _) in enumerate(train_loader):
        optimizer_E.zero_grad()
        optimizer_D.zero_grad()
        x = imgs.to(device)

        # Forward pass
        x_hat, mu, logvar = vae(x)

        # Compute loss
        loss = vae_loss(x_hat, x, mu, logvar)

        # Backward pass and optimize
        loss.backward()
        optimizer_E.step()
        optimizer_D.step()

        # Update epoch loss
        vae_loss_epoch += loss.item()

    # Compute average loss for epoch
    vae_loss_epoch /= len(train_loader)

    # Append loss to list
    vae_losses.append(vae_loss_epoch)

    # Print training progress
    print(f"[Epoch {epoch}/{num_epochs}] [VAE loss: {vae_loss_epoch}]")

# Save final model weights into 'vae_weights' folder
torch.save({
    'encoder': vae.encoder.state_dict(),
    'decoder': vae.decoder.state_dict(),
    'optimizer_E': optimizer_E.state_dict(),
    'optimizer_D': optimizer_D.state_dict(),
}, 'vae_final_weights.pth')
```

```
[Epoch 0/50] [VAE loss: 5804.955906329124]
[Epoch 1/50] [VAE loss: 4616.661853001316]
[Epoch 2/50] [VAE loss: 4422.015665343067]
[Epoch 3/50] [VAE loss: 4319.797353130414]
[Epoch 4/50] [VAE loss: 4246.09870543002]
[Epoch 5/50] [VAE loss: 4192.90337366946]
[Epoch 6/50] [VAE loss: 4152.832818330224]
[Epoch 7/50] [VAE loss: 4120.29134513676]
[Epoch 8/50] [VAE loss: 4090.6985818500966]
[Epoch 9/50] [VAE loss: 4068.9111203191633]
[Epoch 10/50] [VAE loss: 4050.38137550598]
[Epoch 11/50] [VAE loss: 4034.8337269602043]
[Epoch 12/50] [VAE loss: 4014.3184101291813]
[Epoch 13/50] [VAE loss: 4003.3298100388124]
[Epoch 14/50] [VAE loss: 3992.539830319663]
[Epoch 15/50] [VAE loss: 3981.6189534852247]
[Epoch 16/50] [VAE loss: 3968.314207032291]
[Epoch 17/50] [VAE loss: 3964.046532214069]
[Epoch 18/50] [VAE loss: 3954.973526098581]
[Epoch 19/50] [VAE loss: 3945.248815605635]
[Epoch 20/50] [VAE loss: 3937.6452628910415]
[Epoch 21/50] [VAE loss: 3930.096879971307]
[Epoch 22/50] [VAE loss: 3927.541055968067]
[Epoch 23/50] [VAE loss: 3924.578593500133]
[Epoch 24/50] [VAE loss: 3912.9446916071847]
[Epoch 25/50] [VAE loss: 3912.7934487023585]
[Epoch 26/50] [VAE loss: 3907.6724254876567]
[Epoch 27/50] [VAE loss: 3903.6979946632628]
[Epoch 28/50] [VAE loss: 3897.96149605627]
[Epoch 29/50] [VAE loss: 3893.20838586608]
[Epoch 30/50] [VAE loss: 3891.419111630047]
[Epoch 31/50] [VAE loss: 3887.4369989651354]
[Epoch 32/50] [VAE loss: 3884.558365095907]
[Epoch 33/50] [VAE loss: 3882.061797770356]
[Epoch 34/50] [VAE loss: 3878.279879897388]
[Epoch 35/50] [VAE loss: 3874.1861274247485]
[Epoch 36/50] [VAE loss: 3872.9076666923474]
[Epoch 37/50] [VAE loss: 3869.1922053030053]
[Epoch 38/50] [VAE loss: 3864.3014880345067]
[Epoch 39/50] [VAE loss: 3863.760433497968]
[Epoch 40/50] [VAE loss: 3862.3503165499233]
[Epoch 41/50] [VAE loss: 3859.4060346200777]
[Epoch 42/50] [VAE loss: 3857.079953972465]
[Epoch 43/50] [VAE loss: 3855.4955997426373]
[Epoch 44/50] [VAE loss: 3853.535868890758]
[Epoch 45/50] [VAE loss: 3854.152193960096]
[Epoch 46/50] [VAE loss: 3851.227716311717]
[Epoch 47/50] [VAE loss: 3848.785498775653]
[Epoch 48/50] [VAE loss: 3849.8189077804336]
[Epoch 49/50] [VAE loss: 3845.5034773121]
```

2.2 GAN loss plot

In [4]:
```
# Plot VAN losses
import matplotlib.pyplot as plt

plt.plot(range(num_epochs), vae_losses, label='VAE loss')
plt.legend()
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
```



In [ ]:
```
# To improve model we can do the following:
# 1) Increase batch size
# 2) Train G, not D
```

2.3 Using VAE, generate a sample of 25 fake images and plot the corresponding images

In [4]:
```
# Plot for VAE
# In this code, we first load the saved model weights into a new instance of the VAE model.
# Then, we generate a sample of 25 fake images by sampling from the latent space using `torch.randr
# passing the resulting tensor through the VAE decoder.
# We convert the fake images to numpy arrays, rescale them from [-1, 1] to [0, 1], and then
# plot them using matplotlib. The resulting plot shows 25 generated images in a 5x5 grid.

import matplotlib.pyplot as plt
import numpy as np

# Load the saved model weights
checkpoint = torch.load('vae_final_weights.pth')
vae = VAE().to(device)
vae.encoder.load_state_dict(checkpoint['encoder'])
vae.decoder.load_state_dict(checkpoint['decoder'])

# Generate a sample of 25 fake images
with torch.no_grad():
    z = torch.randn(25, 20).to(device)
    fake_imgs = vae.decoder(z)

# Convert the fake images to numpy arrays
fake_imgs = fake_imgs.cpu().numpy()

# Rescale the fake images from [-1, 1] to [0, 1]
fake_imgs = (fake_imgs + 1) / 2

# Plot the fake images
fig, axs = plt.subplots(5, 5, figsize=(8, 8))
for i in range(5):
    for j in range(5):
        axs[i, j].imshow(fake_imgs[i*5+j].reshape((28, 28)), cmap='gray')
        axs[i, j].axis('off')
plt.show()
```

In [1]:

```
# VAE generated much better images than GAN.
## GAN is more difficult to train and takes longer because it is adversarial, there is the generato
## Overall, I would say that both GAN and VAE are powerful. But it may be hard to get hyperparamete
```

In [ ]:

```
```