SYDE 675 Assignment-3

Tzu-Ting Huang
20988611

Exercise 1

1-1

```
mnist = fetch_openml('mnist_784', version=1, as_frame=False, parser='liac-arff')
X = mnist.data
y = mnist.target

X_train, y_train = X[:60000], y[:60000]
X_test, y_test = X[60000:], y[60000:]

mask_train = np.isin(y_train, ['3','4'])
mask_test = np.isin(y_test, ['3','4'])
X_train, y_train = X_train[mask_train], y_train[mask_train]
X_test, y_test = X_test[mask_test], y_test[mask_test]

y_train = y_train.astype(int)
y_test = y_test.astype(int)

y_train[y_train == 3] = 0
y_train[y_train == 4] = 1
y_test[y_test == 3] = 0
y_test[y_test == 4] = 1

pca_train = PCA(n_components=2)
X_pca_train = pca_train.fit_transform(X_train)
X_pca_test = pca_train.transform(X_test)
```

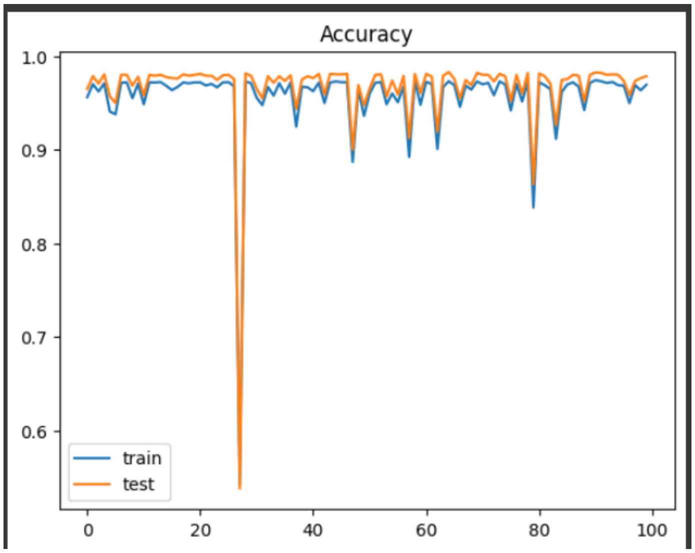
```
def _shuffle(X, y):
  # shuffles two equal-length list/array, X and Y, together.
  randomize = np.arange(len(X))
  np.random.shuffle(randomize)
  return (X[randomize], y[randomize])

def _sigmoid(z):
  # avoid overflow, minimum/maximum output value is set
  return np.clip(1 / (1.0 + np.exp(-z)), 1e-8, 1-(1e-8))

def logisticRegression(X, w, b):
  # X: input data, shape = [batch_size, data_dimension]
  # w: weight vector, shape = [data_dimension, ]
  # b: bias, scalar
  return _sigmoid(np.matmul(X, w)+b)

def _accuracy(y_pred, y_label):
  acc = 1 - np.mean(np.abs(y_pred - y_label))
  return acc
```

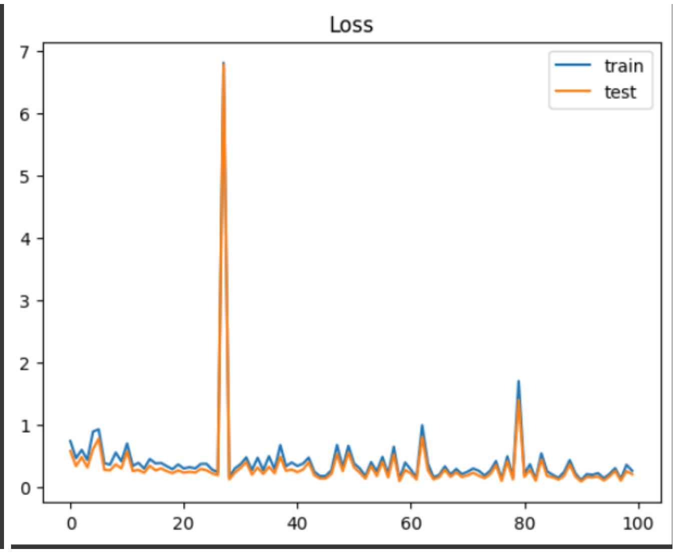
```
[ ]
def _cross_entropy_loss(y_pred, Y_label):
    # y_pred: probabilistic predictions, float vector
    # Y_label: ground truth labels, bool vector
    # Output:cross entropy, scalar
    cross_entropy = -np.dot(Y_label, np.log(y_pred)) - np.dot((1 - Y_label), np.log(1 - y_pred))
    return cross_entropy

def _gradient(X, Y_label, w, b):
    # computes the gradient of cross entropy loss with respect to w and b
    y_pred = logisticRegression(X, w, b)
    pred_error = Y_label - y_pred
    w_grad = -np.sum(pred_error * X.T, 1)
    b_grad = -np.sum(pred_error)
    return w_grad, b_grad
```

```python
# Zero initialization
w = np.zeros((data_dim,))
b = np.zeros((1,))

train_size = X_pca_train.shape[0]
test_size = X_pca_test.shape[0]
data_dim = X_pca_train.shape[1]

max_iter = 100
batch_size = 8
learning_rate = 0.01 #0.01

# Keep the loss and accuracy at every iteration for plotting
train_loss = []
train_acc = []
test_loss = []
test_acc = []

step = 1

# Iterative training
for epoch in range(max_iter):
    # Random shuffle at the begging of each epoch
    X_pca_train, y_train = _shuffle(X_pca_train, y_train)

    # Mini-batch training
    for idx in range(int(np.floor(train_size / batch_size))):
        X = X_pca_train[idx*batch_size:(idx+1)*batch_size]
        Y = y_train[idx*batch_size:(idx+1)*batch_size]

        # Compute the gradient
        w_grad, b_grad = _gradient(X, Y, w, b)
        # gradient descent update
        # learning rate decay with time
        w = w - learning_rate/np.sqrt(step) * w_grad
        b = b - learning_rate/np.sqrt(step) * b_grad

        step = step + 1

    # Compute loss and accuracy of training set and development set
    y_train_pred = logisticRegression(X_pca_train, w, b)
    Y_train_pred = np.round(y_train_pred) ###########
    train_acc.append(_accuracy(Y_train_pred, y_train))
    train_loss.append(_cross_entropy_loss(y_train_pred, y_train) / train_size)

    y_test_pred = logisticRegression(X_pca_test, w, b)
    Y_test_pred = np.round(y_test_pred)
    test_acc.append(_accuracy(Y_test_pred, y_test))
    test_loss.append(_cross_entropy_loss(y_test_pred, y_test) / test_size)

print('Training loss: {}'.format(train_loss[-1]))
print('Test loss: {}'.format(test_loss[-1]))
print('Training accuracy: {}'.format(train_acc[-1]))
print('Test accuracy: {}'.format(test_acc[-1]))
```

```
<ipython-input-26-5d78856a0128>:9: RuntimeWarning: overflow encountered in exp
  return np.clip(1 / (1.0 + np.exp(-z)), 1e-8, 1-(1e-8))
Training loss: 0.428138677694149
Test loss: 0.2965474917626056
Training accuracy: 0.9728555917480999
Test accuracy: 0.981425702811245
```

1-2





1-3

```
def _shuffle(X, y):
  # shuffles two equal-length list/array, X and Y, together.
  randomize = np.arange(len(X))
  np.random.shuffle(randomize)
  return (X[randomize], y[randomize])

def logisticRegression(X, w, b):
  # X: input data, shape = [batch_size, data_dimension]
  # w: weight vector, shape = [data_dimension, ]
  # b: bias, scalar
  return np.matmul(X, w)+b

def _accuracy(y_pred, y_label):
  acc = 1 - np.mean(np.abs(y_pred - y_label))
  return acc

def _cross_entropy_loss(y_pred, Y_label):
  # y_pred: probabilistic predictions, float vector
  # Y_label: ground truth labels, bool vector
  # Output:cross entropy, scalar
  epsilon = 1e-8
  cross_entropy = -np.dot(Y_label, np.log(y_pred + epsilon)) - np.dot((1 - Y_label), np.log(1 - y_pred + epsilon))
                  #-np.dot(Y_label, np.log(y_pred)) - np.dot((1 - Y_label),          np.log(1 - y_pred))
  return cross_entropy

def _gradient(X, Y_label, w, b):
    # computes the gradient of cross entropy loss with respect to w and b
    y_pred  = logisticRegression(X, w, b)
    pred_error = Y_label - y_pred
    w_grad = -np.sum(pred_error * X.T, 1)
    b_grad = -np.sum(pred_error)
    return w_grad, b_grad
```

```python
# Zero initialization
w = np.zeros((data_dim,))
b = np.zeros((1,))

max_iter = 20
batch_size = 8
learning_rate = 0.1

# Keep the loss and accuracy at every iteration for plotting
train_loss = []
train_acc = []
test_loss = []
test_acc = []

step = 1

# Iterative training
for epoch in range(max_iter):
    # Random shuffle at the begging of each epoch
    X_pca_train, y_train = _shuffle(X_pca_train, y_train)

    # Mini-batch training
    for idx in range(int(np.floor(train_size / batch_size))):
        X = X_pca_train[idx*batch_size:(idx+1)*batch_size]
        Y = y_train[idx*batch_size:(idx+1)*batch_size]

        # Compute the gradient
        w_grad, b_grad = _gradient(X, Y, w, b)

        # gradient descent update
        # learning rate decay with time
        w = w - learning_rate/np.sqrt(step) * w_grad
        b = b - learning_rate/np.sqrt(step) * b_grad

        step = step + 1

    # Compute loss and accuracy of training set and development set
    y_train_pred = logisticRegression(X_pca_train, w, b)
    Y_train_pred = np.round(y_train_pred)
    # print(f'y_train_pred:{y_train_pred}')
    train_acc.append(_accuracy(Y_train_pred, y_train))
    train_loss.append(_cross_entropy_loss(y_train_pred, y_train) / train_size)

    y_test_pred = logisticRegression(X_pca_test, w, b)
    Y_test_pred = np.round(y_test_pred)
    # print(f'y_test_pred:{y_test_pred}')
    test_acc.append(_accuracy(Y_test_pred, y_test))
    test_loss.append(_cross_entropy_loss(y_test_pred, y_test) / test_size)


print('Training loss: {}'.format(train_loss[-1]))
print('Test loss: {}'.format(test_loss[-1]))
print('Training accuracy: {}'.format(train_acc[-1]))
print('Test accuracy: {}'.format(test_acc[-1]))
```

```
/usr/local/lib/python3.10/dist-packages/numpy/core/fromnumeric.py:88: RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/usr/local/lib/python3.10/dist-packages/numpy/core/fromnumeric.py:88: RuntimeWarning: invalid value encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
<ipython-input-22-b178355bcd25>:32: RuntimeWarning: invalid value encountered in subtract
  w = w - learning_rate/np.sqrt(step) * w_grad
Training loss: nan
Test loss: nan
Training accuracy: nan
Test accuracy: nan
```

The classifier without any activation function doesn't perform well and results in nan values. This is because logistic regression is designed to predict probabilities, which is why we set classes as 0 and 1 and use the sigmoid function.

Without the sigmoid function, the output of the logistic regression can be any real number, which doesn’t make sense in the context of probability prediction. Furthermore, the large output values can cause numerical instability when they are used in subsequent computations, such as the logarithm in the loss function, leading to nan values.

1-4

```python
# Zero initialization
w = np.zeros((data_dim,))
b = np.ones((1,))

max_iter = 100
batch_size = 8
learning_rate = 0.1 #0.01

# Keep the loss and accuracy at every iteration for plotting
train_loss = []
train_acc = []
test_loss = []
test_acc = []

step = 1

# Iterative training
for epoch in range(max_iter):
    # Random shuffle at the begging of each epoch
    X_pca_train, y_train = _shuffle(X_pca_train, y_train)

    # Mini-batch training
    for idx in range(int(np.floor(train_size / batch_size))):
        X = X_pca_train[idx*batch_size:(idx+1)*batch_size]
        Y = y_train[idx*batch_size:(idx+1)*batch_size]

        # Compute the gradient
        w_grad, b_grad = _gradient(X, Y, w, b)
        # gradient descent update
        # learning rate decay with time
        w = w - learning_rate/np.sqrt(step) * w_grad
        b = b - learning_rate/np.sqrt(step) * b_grad

        step = step + 1

    # Compute loss and accuracy of training set and development set
    y_train_pred = logisticRegression(X_pca_train, w, b)
    Y_train_pred = np.round(y_train_pred)
    train_acc.append(_accuracy(Y_train_pred, y_train))
    train_loss.append(_cross_entropy_loss(y_train_pred, y_train) / train_size)

    y_test_pred = logisticRegression(X_pca_test, w, b)
    Y_test_pred = np.round(y_test_pred)
    test_acc.append(_accuracy(Y_test_pred, y_test))
    test_loss.append(_cross_entropy_loss(y_test_pred, y_test) / test_size)

print('Training loss: {}'.format(train_loss[-1]))
print('Test loss: {}'.format(test_loss[-1]))
print('Training accuracy: {}'.format(train_acc[-1]))
print('Test accuracy: {}'.format(test_acc[-1]))
```

```
<ipython-input-26-5d78856a0128>:9: RuntimeWarning: overflow encountered in exp
  return np.clip(1 / (1.0 + np.exp(-z)), 1e-8, 1-(1e-8))
Training loss: 0.428138677694149
Test loss: 0.2965474917626056
Training accuracy: 0.9728555917480999
Test accuracy: 0.981425702811245
```

The result shows that there is an increase in loss, which could be due to the initial shift, but it doesn't necessarily mean that the model is worse. Changing the bias to 1 allows the decision boundary to shift away from the origin. The accuracy might be slightly affected by this shift in the decision boundary, which could explain why the accuracy doesn't change significantly. The bias and weights are parameters that are learned during training, so they get updated. Therefore, the initial value is less important than the learning process itself.

1-5

Accuracy MED: 0.9764056224899599
Accuracy MMD: 0.9819277108433735
Accuracy kNN: 0.9649
Accuracy Logistic regression: 0.983433734939759

According to result, the performance: Logistic regression >MMD>MED>kNN

All classifiers aim to estimate $P(C_1|x) = \sigma(w \cdot x+b)$, but they do so differently. Logistic regression, a discriminative model, estimates w and b directly without making assumptions about the data distribution. MED and MMD, generative models, estimate parameters of the class-conditional densities and class priors, which are used to compute w and b. They assume Gaussian distribution or Bernulli or some other distributions. The kNN algorithm, an instance-based learner, classifies based on instance similarity. Discriminative models often outperform generative models when the decision boundary is complex, generative models can excel when their distributional assumptions hold.

Exercise 2 CNN
2-1
```
# Define a transform to normalize the data
transform = transforms.Compose([transforms.Resize((32, 32)),
                                transforms.ToTensor(),
                                transforms.RandomGrayscale(), ##not sure
                                transforms.Normalize((0.5,), (0.5,))])

# Load MNIST dataset
full_train_data = datasets.MNIST('~/.pytorch/MNIST_data/', download=True, train=True, transform=transform)
full_test_data = datasets.MNIST('~/.pytorch/MNIST_data/', train=False, download=True, transform=transform)

train_loader = DataLoader(full_train_data, batch_size = 64, shuffle = True)
test_loader = DataLoader(full_test_data, batch_size = 64, shuffle = False)
```

```
class VGG11(nn.Module):
    def __init__(self):
        super(VGG11, self).__init__()
        # torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)
        # torch.nn.MaxPool2d(kernel_size, stride, padding)
        self.features = nn.Sequential(
            nn.Conv2d(1, 64, 3, 1, 1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0),

            nn.Conv2d(64, 128, 3, 1, 1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0),

            nn.Conv2d(128, 256, 3, 1, 1),
            nn.BatchNorm2d(256),
            nn.ReLU(),

            nn.Conv2d(256, 256, 3, 1, 1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0),

            nn.Conv2d(256, 512, 3, 1, 1),
            nn.BatchNorm2d(512),
            nn.ReLU(),

            nn.Conv2d(512, 512, 3, 1, 1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0),

            nn.Conv2d(512, 512, 3, 1, 1),
            nn.BatchNorm2d(512),
            nn.ReLU(),

            nn.Conv2d(512, 512, 3, 1, 1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0),
        )
        self.classifier = nn.Sequential(
            nn.Linear(512, 4096),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(4096, 4096),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(4096, 10)
        )
```

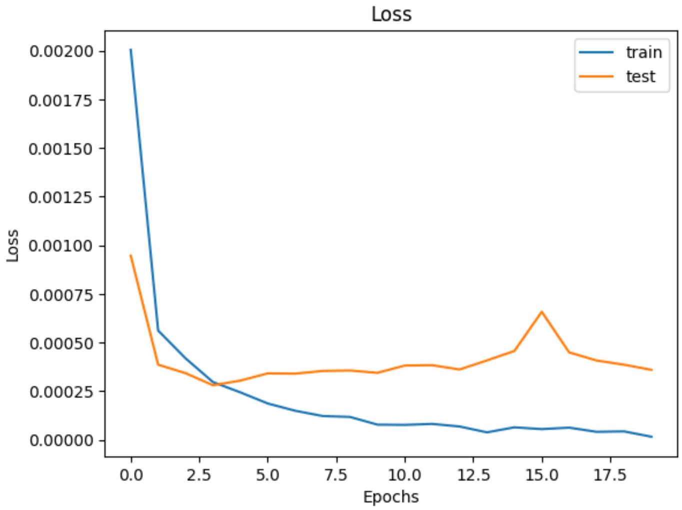
```python
    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x
```

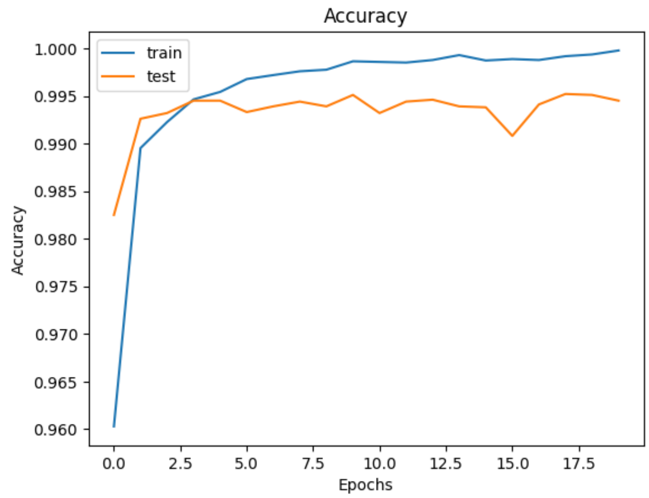
```python
model = VGG11().cuda()

# Define the loss function and optimizer
loss = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

if torch.cuda.is_available():
   model = model.cuda()

train_acc_values = []
train_loss_values = []
test_acc_values = []
test_loss_values = []
num_epoch = 20
for epoch in range(num_epoch):
    epoch_start_time = time.time()
    train_acc = 0.0
    train_loss = 0.0
    test_acc = 0.0
    test_loss = 0.0

    model.train() # 確保 model 是在 train model (開啟 Dropout 等...)
    for i, data in enumerate(train_loader):

        optimizer.zero_grad() # 用 optimizer 將 model 參數的 gradient 歸零
        train_pred = model(data[0].cuda()) # 利用 model 得到預測的機率分佈 這邊實際上就是去呼叫 model 的 forward 函數
        batch_loss = loss(train_pred, data[1].cuda()) # 計算 loss （注意 prediction 跟 label 必須同時在 CPU 或是 GPU 上）
        batch_loss.backward() # 利用 back propagation 算出每個參數的 gradient
        optimizer.step() # 以 optimizer 用 gradient 更新參數值

        train_acc += np.sum(np.argmax(train_pred.cpu().data.numpy(), axis=1) == data[1].numpy())
        train_loss += batch_loss.item()

    model.eval()
    with torch.no_grad():
        for i, data in enumerate(test_loader):
            test_pred = model(data[0].cuda())
            batch_loss = loss(test_pred, data[1].cuda())

            test_acc += np.sum(np.argmax(test_pred.cpu().data.numpy(), axis=1) == data[1].numpy())
            test_loss += batch_loss.item()

        #將結果 print 出來
        print('[%03d/%03d] %2.2f sec(s) Train Acc: %3.6f Loss: %3.6f | Test Acc: %3.6f loss: %3.6f' % \
            (epoch + 1, num_epoch, time.time()-epoch_start_time, \
             train_acc/full_train_data.__len__(), train_loss/full_train_data.__len__(), test_acc/full_test_data.__len_

    train_acc_values.append(train_acc/full_train_data.__len__())
    train_loss_values.append(train_loss/full_train_data.__len__())
    test_acc_values.append(test_acc/full_test_data.__len__())
    test_loss_values.append(test_loss/full_test_data.__len__())
```

```
[001/020] 41.08 sec(s) Train Acc: 0.960300 Loss: 0.002004 | Test Acc: 0.982500 loss: 0.000946
[002/020] 42.16 sec(s) Train Acc: 0.989517 Loss: 0.000561 | Test Acc: 0.992600 loss: 0.000385
[003/020] 52.74 sec(s) Train Acc: 0.992250 Loss: 0.000418 | Test Acc: 0.993200 loss: 0.000342
[004/020] 47.63 sec(s) Train Acc: 0.994633 Loss: 0.000296 | Test Acc: 0.994500 loss: 0.000280
[005/020] 49.21 sec(s) Train Acc: 0.995417 Loss: 0.000243 | Test Acc: 0.994500 loss: 0.000304
[006/020] 43.95 sec(s) Train Acc: 0.996767 Loss: 0.000186 | Test Acc: 0.993300 loss: 0.000341
[007/020] 46.55 sec(s) Train Acc: 0.997183 Loss: 0.000149 | Test Acc: 0.993900 loss: 0.000339
[008/020] 44.57 sec(s) Train Acc: 0.997583 Loss: 0.000122 | Test Acc: 0.994400 loss: 0.000353
[009/020] 48.57 sec(s) Train Acc: 0.997750 Loss: 0.000117 | Test Acc: 0.993900 loss: 0.000356
[010/020] 45.17 sec(s) Train Acc: 0.998633 Loss: 0.000077 | Test Acc: 0.995100 loss: 0.000344
[011/020] 47.57 sec(s) Train Acc: 0.998567 Loss: 0.000076 | Test Acc: 0.993200 loss: 0.000381
[012/020] 44.43 sec(s) Train Acc: 0.998500 Loss: 0.000081 | Test Acc: 0.994400 loss: 0.000383
[013/020] 46.22 sec(s) Train Acc: 0.998767 Loss: 0.000068 | Test Acc: 0.994600 loss: 0.000361
[014/020] 46.32 sec(s) Train Acc: 0.999283 Loss: 0.000038 | Test Acc: 0.993900 loss: 0.000408
[015/020] 46.15 sec(s) Train Acc: 0.998717 Loss: 0.000064 | Test Acc: 0.993800 loss: 0.000456
[016/020] 51.56 sec(s) Train Acc: 0.998867 Loss: 0.000054 | Test Acc: 0.990800 loss: 0.000658
[017/020] 42.54 sec(s) Train Acc: 0.998767 Loss: 0.000062 | Test Acc: 0.994100 loss: 0.000449
[018/020] 42.73 sec(s) Train Acc: 0.999167 Loss: 0.000040 | Test Acc: 0.995200 loss: 0.000407
[019/020] 41.72 sec(s) Train Acc: 0.999350 Loss: 0.000043 | Test Acc: 0.995100 loss: 0.000386
[020/020] 42.33 sec(s) Train Acc: 0.999767 Loss: 0.000015 | Test Acc: 0.994500 loss: 0.000359
```

The reason why we must resize the images is due to the architecture of the VGG11 model. There are 5 MaxPool2d in this model and this reduces the size of each feature map by half.

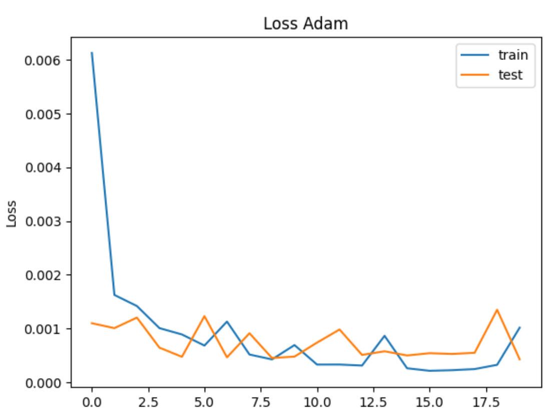
2-2

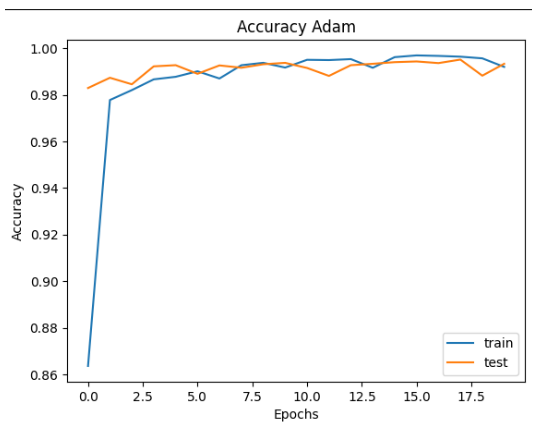




2-3 Adam

```
[001/020] 47.00 sec(s) Train Acc: 0.863583 Loss: 0.006125 | Test Acc: 0.982900 loss: 0.001099
[002/020] 45.97 sec(s) Train Acc: 0.977750 Loss: 0.001625 | Test Acc: 0.987300 loss: 0.001008
[003/020] 45.74 sec(s) Train Acc: 0.982000 Loss: 0.001419 | Test Acc: 0.984500 loss: 0.001204
[004/020] 45.69 sec(s) Train Acc: 0.986617 Loss: 0.001008 | Test Acc: 0.992200 loss: 0.000645
[005/020] 45.50 sec(s) Train Acc: 0.987717 Loss: 0.000890 | Test Acc: 0.992700 loss: 0.000475
[006/020] 46.02 sec(s) Train Acc: 0.990083 Loss: 0.000684 | Test Acc: 0.989000 loss: 0.001230
[007/020] 45.87 sec(s) Train Acc: 0.986983 Loss: 0.001130 | Test Acc: 0.992600 loss: 0.000465
[008/020] 45.82 sec(s) Train Acc: 0.992683 Loss: 0.000517 | Test Acc: 0.991600 loss: 0.000913
[009/020] 44.93 sec(s) Train Acc: 0.993717 Loss: 0.000426 | Test Acc: 0.993100 loss: 0.000452
[010/020] 45.68 sec(s) Train Acc: 0.991683 Loss: 0.000693 | Test Acc: 0.993700 loss: 0.000478
[011/020] 46.14 sec(s) Train Acc: 0.995000 Loss: 0.000331 | Test Acc: 0.991500 loss: 0.000738
[012/020] 45.68 sec(s) Train Acc: 0.994883 Loss: 0.000331 | Test Acc: 0.988100 loss: 0.000982
[013/020] 45.58 sec(s) Train Acc: 0.995300 Loss: 0.000312 | Test Acc: 0.992700 loss: 0.000510
[014/020] 45.48 sec(s) Train Acc: 0.991583 Loss: 0.000865 | Test Acc: 0.993300 loss: 0.000577
[015/020] 45.44 sec(s) Train Acc: 0.996150 Loss: 0.000262 | Test Acc: 0.994000 loss: 0.000500
[016/020] 45.39 sec(s) Train Acc: 0.996900 Loss: 0.000214 | Test Acc: 0.994300 loss: 0.000542
[017/020] 45.42 sec(s) Train Acc: 0.996683 Loss: 0.000226 | Test Acc: 0.993600 loss: 0.000528
[018/020] 46.25 sec(s) Train Acc: 0.996350 Loss: 0.000247 | Test Acc: 0.995100 loss: 0.000549
[019/020] 45.29 sec(s) Train Acc: 0.995633 Loss: 0.000326 | Test Acc: 0.988200 loss: 0.001349
[020/020] 45.34 sec(s) Train Acc: 0.991967 Loss: 0.001017 | Test Acc: 0.993300 loss: 0.000428
```

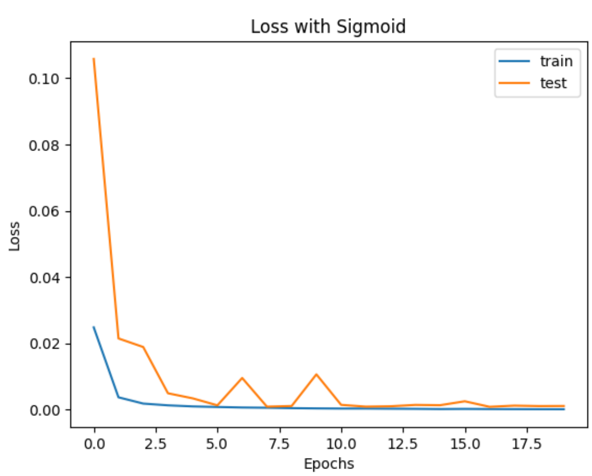


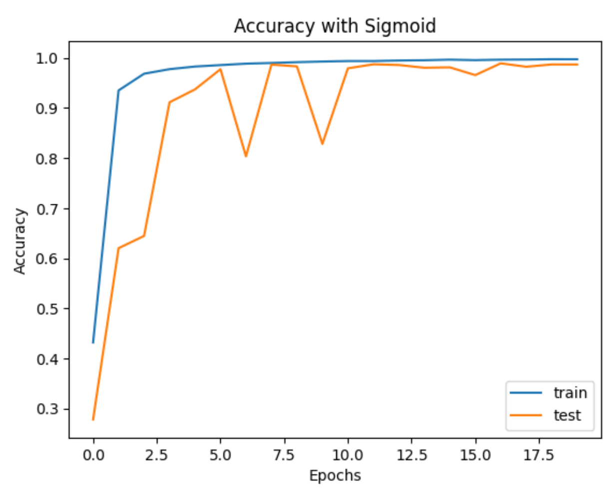


According to my result, their performances are quite similar. I'm unsure if that is because I only trained 20 epochs and I set learning rate differently. I set 0.01 for SGD and 0.001 for Adam, cause when I set Adam with learning as 0.01, the results are poor. However, in general Adam usually performs better than SGD. Cause Adam includes bias correction to adjust the learning rates, which can lead to more efficient learning. It also uses momentum to help the optimizer navigate along the relevant directions. Additionally, Adam is well-suited for problems with sparse gradients, as it uses moving averages of the gradient instead of the gradient itself like SGD.

2-4 with Sigmoid

```
[001/020] 41.64 sec(s) Train Acc: 0.431700 Loss: 0.024819 | Test Acc: 0.277900 loss: 0.105784
[002/020] 42.49 sec(s) Train Acc: 0.935183 Loss: 0.003725 | Test Acc: 0.620000 loss: 0.021477
[003/020] 41.21 sec(s) Train Acc: 0.968583 Loss: 0.001833 | Test Acc: 0.644500 loss: 0.018894
[004/020] 42.38 sec(s) Train Acc: 0.977650 Loss: 0.001303 | Test Acc: 0.911500 loss: 0.004943
[005/020] 41.40 sec(s) Train Acc: 0.982933 Loss: 0.000976 | Test Acc: 0.937300 loss: 0.003413
[006/020] 41.71 sec(s) Train Acc: 0.985833 Loss: 0.000808 | Test Acc: 0.977600 loss: 0.001303
[007/020] 41.70 sec(s) Train Acc: 0.988667 Loss: 0.000649 | Test Acc: 0.803400 loss: 0.009553
[008/020] 41.83 sec(s) Train Acc: 0.990017 Loss: 0.000578 | Test Acc: 0.986800 loss: 0.000908
[009/020] 42.11 sec(s) Train Acc: 0.991650 Loss: 0.000454 | Test Acc: 0.983200 loss: 0.001098
[010/020] 42.14 sec(s) Train Acc: 0.992917 Loss: 0.000378 | Test Acc: 0.828400 loss: 0.010653
[011/020] 42.93 sec(s) Train Acc: 0.993867 Loss: 0.000331 | Test Acc: 0.979300 loss: 0.001452
[012/020] 42.10 sec(s) Train Acc: 0.993900 Loss: 0.000331 | Test Acc: 0.987400 loss: 0.000912
[013/020] 42.25 sec(s) Train Acc: 0.994967 Loss: 0.000287 | Test Acc: 0.986000 loss: 0.001025
[014/020] 42.36 sec(s) Train Acc: 0.995500 Loss: 0.000251 | Test Acc: 0.980500 loss: 0.001433
[015/020] 42.27 sec(s) Train Acc: 0.996683 Loss: 0.000185 | Test Acc: 0.981300 loss: 0.001344
[016/020] 42.34 sec(s) Train Acc: 0.995717 Loss: 0.000229 | Test Acc: 0.965800 loss: 0.002525
[017/020] 42.23 sec(s) Train Acc: 0.996683 Loss: 0.000188 | Test Acc: 0.989300 loss: 0.000862
[018/020] 42.85 sec(s) Train Acc: 0.996900 Loss: 0.000161 | Test Acc: 0.982500 loss: 0.001217
[019/020] 42.26 sec(s) Train Acc: 0.997550 Loss: 0.000144 | Test Acc: 0.987100 loss: 0.001082
[020/020] 42.53 sec(s) Train Acc: 0.997433 Loss: 0.000130 | Test Acc: 0.987000 loss: 0.001105
```

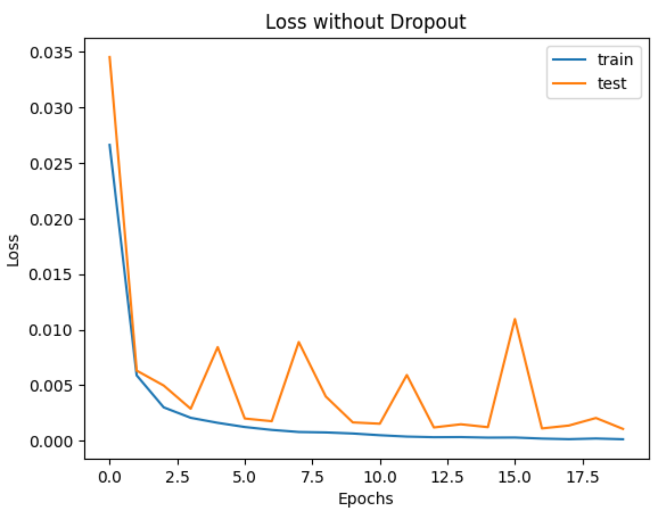


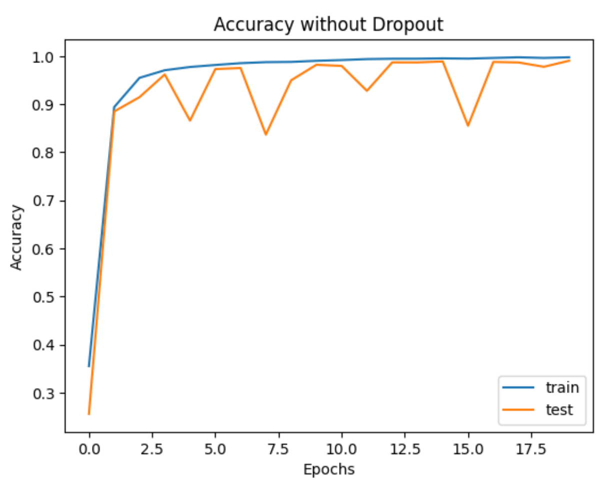


ReLU performs better than sigmoid. The benefits of ReLU are sparsity and a reduced likelihood of vanishing gradient. ReLU is h=max(0,a) where a=Wx+b. when a>0. In this regime, the gradient has a constant value. In contrast, the gradient of sigmoid becomes increasingly small as the absolute value of x increases. The other thing is Sparsity arises when $a \le 0$. The more such units that exist in a layer the sparser the resulting representation. Sigmoid on the other hand is always likely to generate some non-zero value resulting in dense representations. Sparse representations seem to be more beneficial than dense representations.

2-5 without Dropout

```
[001/020] 41.53 sec(s) Train Acc: 0.355183 Loss: 0.026623 | Test Acc: 0.255800 loss: 0.034526
[002/020] 42.00 sec(s) Train Acc: 0.894200 Loss: 0.005887 | Test Acc: 0.885100 loss: 0.006313
[003/020] 42.01 sec(s) Train Acc: 0.954983 Loss: 0.003006 | Test Acc: 0.914800 loss: 0.004971
[004/020] 42.09 sec(s) Train Acc: 0.970817 Loss: 0.002067 | Test Acc: 0.962100 loss: 0.002878
[005/020] 41.41 sec(s) Train Acc: 0.977617 Loss: 0.001613 | Test Acc: 0.866100 loss: 0.008429
[006/020] 41.45 sec(s) Train Acc: 0.981783 Loss: 0.001239 | Test Acc: 0.973200 loss: 0.002003
[007/020] 40.96 sec(s) Train Acc: 0.985533 Loss: 0.000976 | Test Acc: 0.975500 loss: 0.001759
[008/020] 41.54 sec(s) Train Acc: 0.987667 Loss: 0.000790 | Test Acc: 0.836900 loss: 0.008884
[009/020] 40.88 sec(s) Train Acc: 0.988167 Loss: 0.000749 | Test Acc: 0.949900 loss: 0.003983
[010/020] 41.73 sec(s) Train Acc: 0.990450 Loss: 0.000654 | Test Acc: 0.982200 loss: 0.001653
[011/020] 40.98 sec(s) Train Acc: 0.992033 Loss: 0.000500 | Test Acc: 0.979800 loss: 0.001532
[012/020] 41.60 sec(s) Train Acc: 0.994033 Loss: 0.000377 | Test Acc: 0.928000 loss: 0.005917
[013/020] 40.81 sec(s) Train Acc: 0.994783 Loss: 0.000322 | Test Acc: 0.987200 loss: 0.001198
[014/020] 41.53 sec(s) Train Acc: 0.994833 Loss: 0.000332 | Test Acc: 0.987100 loss: 0.001482
[015/020] 41.07 sec(s) Train Acc: 0.995517 Loss: 0.000282 | Test Acc: 0.989100 loss: 0.001225
[016/020] 41.45 sec(s) Train Acc: 0.995017 Loss: 0.000292 | Test Acc: 0.855400 loss: 0.010953
[017/020] 41.37 sec(s) Train Acc: 0.996400 Loss: 0.000196 | Test Acc: 0.988100 loss: 0.001111
[018/020] 41.43 sec(s) Train Acc: 0.997800 Loss: 0.000142 | Test Acc: 0.987000 loss: 0.001371
[019/020] 41.36 sec(s) Train Acc: 0.996417 Loss: 0.000203 | Test Acc: 0.978000 loss: 0.002049
[020/020] 41.51 sec(s) Train Acc: 0.997800 Loss: 0.000135 | Test Acc: 0.990600 loss: 0.001068
```





With dropout model performs better than without dropout one. Dropout, applied during training, creates an ensemble of varied networks, enhancing performance. And it's a regularization technique preventing overfitting by randomly deactivating neurons, forcing the model to learn robust features. During testing, all neurons are used, but their outputs are scaled down by the dropout rate, maintaining output magnitude and model stability. Thus, dropout results in a more robust model with better generalization to unseen data.

Exercise 3- MLP

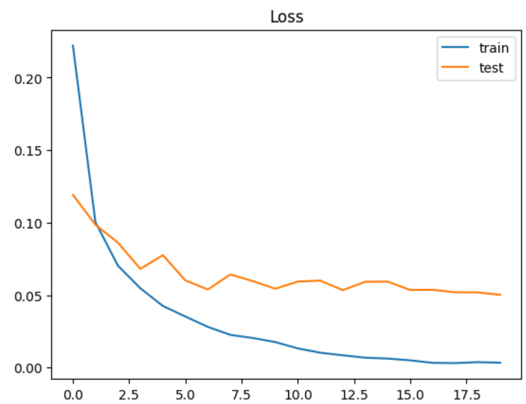
3-1

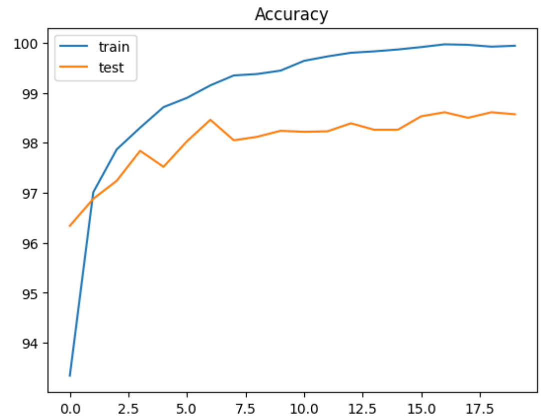
```python
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(784, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

    def forward(self, x):
        x = x.view(x.size(0), -1)  # Flatten the input images
        x = self.layers(x)
        return x
```

```
Epoch: 1, Training Loss: 0.2221, Training Accuracy: 93.34%, Test Loss: 0.1191, Test Accuracy: 96.34%
Epoch: 2, Training Loss: 0.1003, Training Accuracy: 97.01%, Test Loss: 0.0986, Test Accuracy: 96.88%
Epoch: 3, Training Loss: 0.0703, Training Accuracy: 97.87%, Test Loss: 0.0862, Test Accuracy: 97.24%
Epoch: 4, Training Loss: 0.0547, Training Accuracy: 98.30%, Test Loss: 0.0681, Test Accuracy: 97.84%
Epoch: 5, Training Loss: 0.0425, Training Accuracy: 98.71%, Test Loss: 0.0776, Test Accuracy: 97.52%
Epoch: 6, Training Loss: 0.0352, Training Accuracy: 98.90%, Test Loss: 0.0601, Test Accuracy: 98.03%
Epoch: 7, Training Loss: 0.0281, Training Accuracy: 99.15%, Test Loss: 0.0539, Test Accuracy: 98.46%
Epoch: 8, Training Loss: 0.0226, Training Accuracy: 99.35%, Test Loss: 0.0642, Test Accuracy: 98.05%
Epoch: 9, Training Loss: 0.0204, Training Accuracy: 99.38%, Test Loss: 0.0597, Test Accuracy: 98.12%
Epoch: 10, Training Loss: 0.0176, Training Accuracy: 99.44%, Test Loss: 0.0544, Test Accuracy: 98.24%
Epoch: 11, Training Loss: 0.0132, Training Accuracy: 99.64%, Test Loss: 0.0593, Test Accuracy: 98.22%
Epoch: 12, Training Loss: 0.0103, Training Accuracy: 99.73%, Test Loss: 0.0600, Test Accuracy: 98.23%
Epoch: 13, Training Loss: 0.0085, Training Accuracy: 99.80%, Test Loss: 0.0534, Test Accuracy: 98.39%
Epoch: 14, Training Loss: 0.0068, Training Accuracy: 99.83%, Test Loss: 0.0592, Test Accuracy: 98.26%
Epoch: 15, Training Loss: 0.0062, Training Accuracy: 99.86%, Test Loss: 0.0593, Test Accuracy: 98.26%
Epoch: 16, Training Loss: 0.0050, Training Accuracy: 99.91%, Test Loss: 0.0535, Test Accuracy: 98.53%
Epoch: 17, Training Loss: 0.0033, Training Accuracy: 99.97%, Test Loss: 0.0536, Test Accuracy: 98.61%
Epoch: 18, Training Loss: 0.0031, Training Accuracy: 99.96%, Test Loss: 0.0519, Test Accuracy: 98.50%
Epoch: 19, Training Loss: 0.0037, Training Accuracy: 99.92%, Test Loss: 0.0518, Test Accuracy: 98.61%
Epoch: 20, Training Loss: 0.0034, Training Accuracy: 99.94%, Test Loss: 0.0502, Test Accuracy: 98.57%
```

3-2

3-2





3-3

VGG11 network performs better than MLP. The depth of VGG11 allows it to learn complex features. And its convolutional layers are designed for image data, unlike MLPs. VGG11 also includes dropout layers for regularization, preventing overfitting.