Homework8

```
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler
```

```
train_dataset = dsets.MNIST(root='./data',
                            train=True,
                            transform=transforms.ToTensor(),
                            download=True)
test_dataset = dsets.MNIST(root='./data',
                           train=False,
                           transform=transforms.ToTensor())
```

```
train_set = [ex for ex in train_dataset if ex[1]==5][:500] + [ex for ex in train_dataset if ex[1]==8][:500]
test_set = [ex for ex in test_dataset if ex[1]==5][:500] + [ex for ex in test_dataset if ex[1]==8][:500]
```

Create training and test set with 500 examples of classes 5 and 8 respectively and change labels to 0 and 1. At the same time flip 30% of the labels.

```
train_set_01_images = []
test_set_01_images = []
train_set_01_labels = []
test_set_01_labels = []
num_flips = 500//3 #num_flips=0 to have no flips
for i,ex_tr in enumerate(train_set):
    ex_tr = list(ex_tr)
    if ex_tr[1]==5:
        if i<num_flips:
            ex_tr[1]=1
        else:
            ex_tr[1]=0
    else:
        if i<num_flips+500:
            ex_tr[1]=0
        else:
            ex_tr[1]=1
    train_set_01_images.append(ex_tr[0])
    train_set_01_labels.append(ex_tr[1])

for ex_te in test_set:
    ex_te = list(ex_te)
    if ex_te[1]==5:
        ex_te[1]=0
    else:
        ex_te[1]=1
    test_set_01_images.append(ex_te[0])
    test_set_01_labels.append(ex_te[1])

train_set_01_images = torch.stack(train_set_01_images)
train_set_01_images = train_set_01_images.view(-1, 28*28)
test_set_01_images = torch.stack(test_set_01_images)
test_set_01_images = test_set_01_images.view(-1, 28*28)
train_set_01_labels = torch.tensor(train_set_01_labels)
test_set_01_labels = torch.tensor(test_set_01_labels)

# shuffle training set
n_sample = len(train_set_01_images)
np.random.seed(0)
order = np.random.permutation(n_sample)
train_set_01_images = train_set_01_images[order]
train_set_01_labels = train_set_01_labels[order]
```

```
print("size train set:", train_set_01_images.shape)
print("size test set:", test_set_01_images.shape)
```

```
size train set: torch.Size([1000, 784])
size test set: torch.Size([1000, 784])
```

Problem 1

Both methods are designed to prevent overfitting in a same manner.

As training processes, the model tends to include the noise in training data, in high dimensional space, it leads to a solution with large magnitute.

However, how the methods deal with the problem is different. L2 regularization introduce a $||\cdot||_2^2$ based panalty on the solution with large magnitute, while the early stopping will halt the trainning process when the empirical risk of the model is minimized.

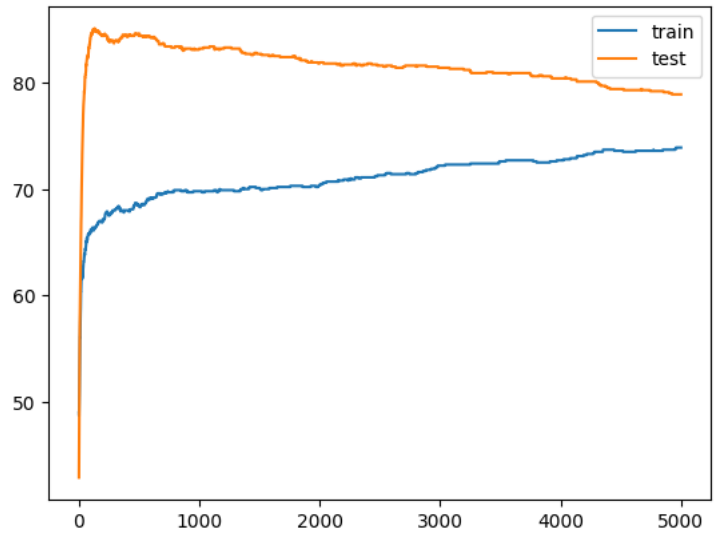
Therefore the resulting solutions are not sufficiently equivalent, but the behavior is similar.

Problem 2

1

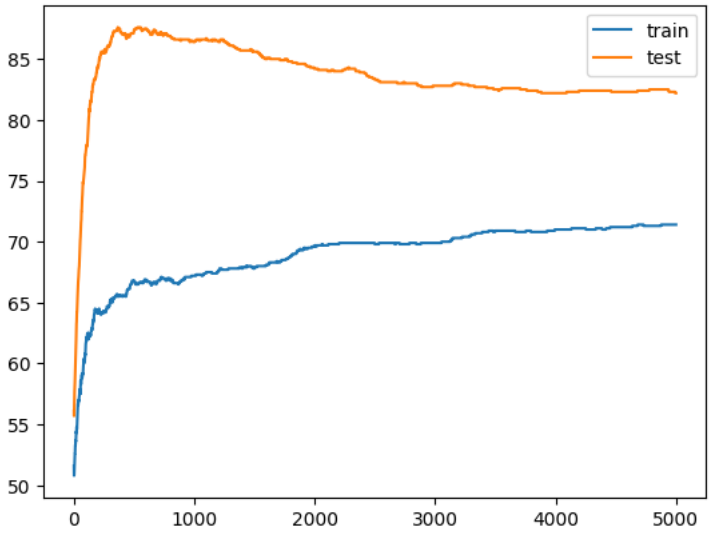
```
class LogisticRegressionModel(nn.Module):
    def __init__(self, dim_in):
        super(LogisticRegressionModel, self).__init__()
        # Linear layer with input size 28*28 (for flattened image inputs) and output size 2 (for binary classification)
        self.linear_layer = nn.Linear(dim_in, 2)
        # Cross entropy loss function for classification
        self.loss_function = nn.CrossEntropyLoss()

    def forward(self, x):
        # Forward pass through the linear layer
        return self.linear_layer(x)

def evaluateLogisticRegression(optimizer, model, train_acc, test_acc,
                       train_set, test_set,
                       train_label = train_set_01_labels,
                       test_label = test_set_01_labels):
  # Initialize
  optimizer.zero_grad()
  outputs = model(train_set)
  loss = model.loss_function(outputs, train_label)

  # Compute gradient of the loss w.r.t model parameters.
  loss.backward()

  optimizer.step()

  _, predicted = torch.max(outputs.data, 1)
  total_tr = train_label.size(0)
  correct_tr = (predicted == train_label).sum()
  accuracy_tr = 100 * correct_tr / total_tr
  train_acc.append(accuracy_tr)

  outputs = model(test_set)
  _, predicted = torch.max(outputs.data, 1)
  total = test_label.size(0)
  correct = (predicted == test_label).sum()

  accuracy = 100 * correct / total
  test_acc.append(accuracy)

  return train_acc, test_acc
```

```
LRM = LogisticRegressionModel(28 ** 2)
learning_rate = 0.01
optimizer = torch.optim.SGD(LRM.parameters(), lr=learning_rate)

train_acc = []
test_acc = []

for i in range(5000):
 train_acc, test_acc = evaluateLogisticRegression(optimizer, LRM, train_acc, test_acc,
                                                  train_set_01_images, test_set_01_images)

plt.plot(train_acc, label = "train")
plt.plot(test_acc, label = "test")

plt.legend()

plt.show()
```



2

```
# Random feature model
num_featurs = train_set_01_images.shape[1]
ReLU = nn.ReLU()

F = ReLU(torch.from_numpy( np.random.normal(loc=0, scale=1, size=[num_featurs,2*num_featurs]) ).type(torch.FloatTensor))
x_train_rf = train_set_01_images @ F
x_test_rf = test_set_01_images @ F

# Normalize feature
x_train_rf -= x_train_rf.mean(0)[None, :]
x_train_rf /= x_train_rf.std(0)[None, :]
x_test_rf -= x_test_rf.mean(0)[None, :]
x_test_rf /= x_test_rf.std(0)[None, :]

LRM_rf = LogisticRegressionModel((28**2)*2)
learning_rate_rf = 0.01
optimizer_rf = torch.optim.SGD(LRM_rf.parameters(), lr = 0.001)

train_acc_rf = []
test_acc_rf = []

for i in range(5000):
  train_acc_rf, test_acc_rf = evaluateLogisticRegression(optimizer_rf, LRM_rf,
                                                         train_acc_rf, test_acc_rf,
                                                         x_train_rf, x_test_rf)

plt.plot(train_acc_rf,label='train')
plt.plot(test_acc_rf,label='test')
plt.legend()
plt.show()
```


train
test
85
80
75
70
65
60
55
50
0
1000
2000
3000
4000
5000