HW6-Q1

In [70]:
```
import h5py
import torch
import torch.nn as nn
import torch.utils.data as data
import torch.optim as optim
import matplotlib.pyplot as plt

def load_data(train_file,test_file):
    with h5py.File(train_file,'r') as f:
        x_train=torch.tensor(f['xdata'][:]).float() #to numpy
        y_train=torch.tensor(f['ydata'][:]).float()
    with h5py.File(test_file,'r') as f:
        x_test=torch.tensor(f['xdata'][:]).float()
        y_test=torch.tensor(f['ydata'][:]).float()

    train_dataset=data.TensorDataset(x_train,y_train)
    test_dataset=data.TensorDataset(x_test,y_test)

    train_loader=data.DataLoader(train_dataset,batch_size=100,shuffle=True)
    test_loader=data.DataLoader(test_dataset,batch_size=100,shuffle=False)

    return train_loader,test_loader

train_loader,test_loader=load_data('mnist_traindata.hdf5','mnist_testdata.hdf5')

"""
class Logistic_classification(nn.Module):
    def __init__(self,input_dim=784,num_classes=10):
        super(Logistic_classification,self).__init__()
        self.linear=nn.Linear(input_dim,num_classes)
    def forward(self,x):
        return self.linear(x)

model=Logistic_classification()
"""

model=nn.Sequential(
    nn.Linear(784,10)
)

loss_func=nn.CrossEntropyLoss()

learning_rate=0.005
optimizer=optim.SGD(model.parameters(),lr=learning_rate)
```

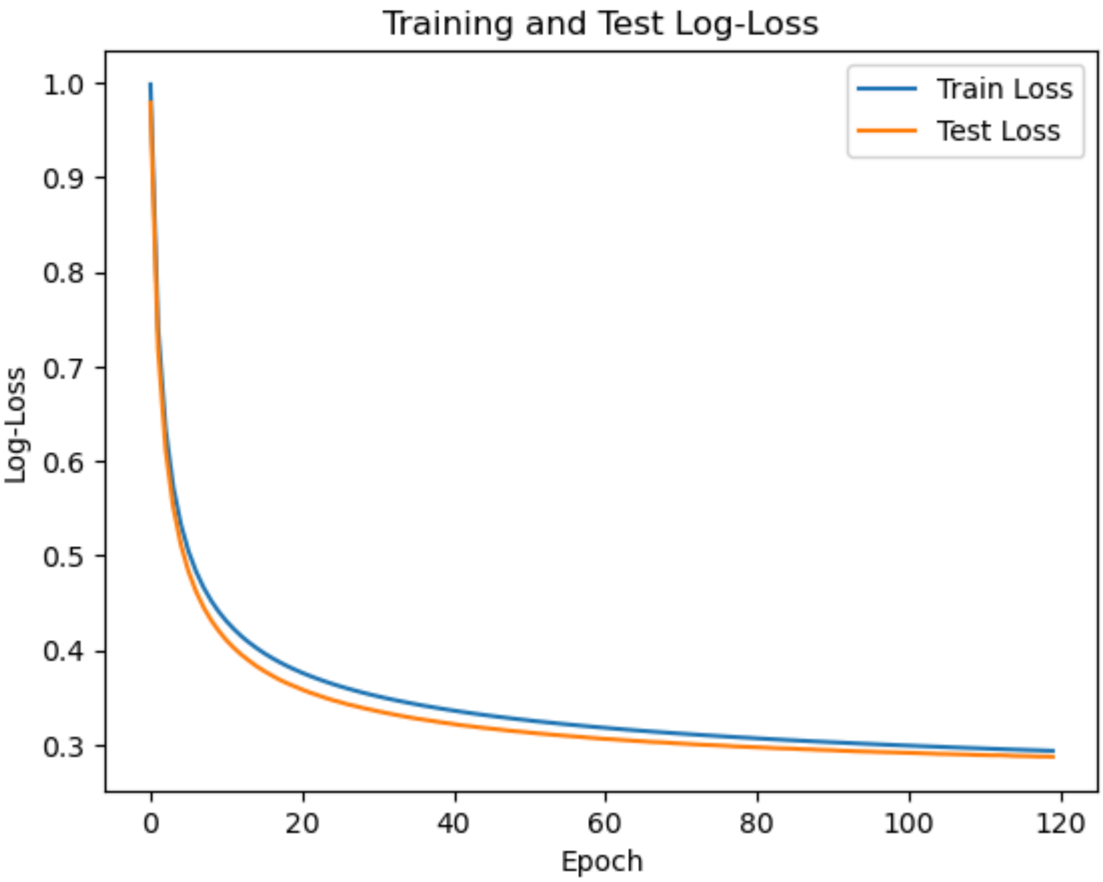
In [71]:
```
#training Loop
num_epochs=120

loss_train=[]
loss_test=[]
accuracy_train=[]
accuracy_test=[]

for epoch in range(num_epochs):
```

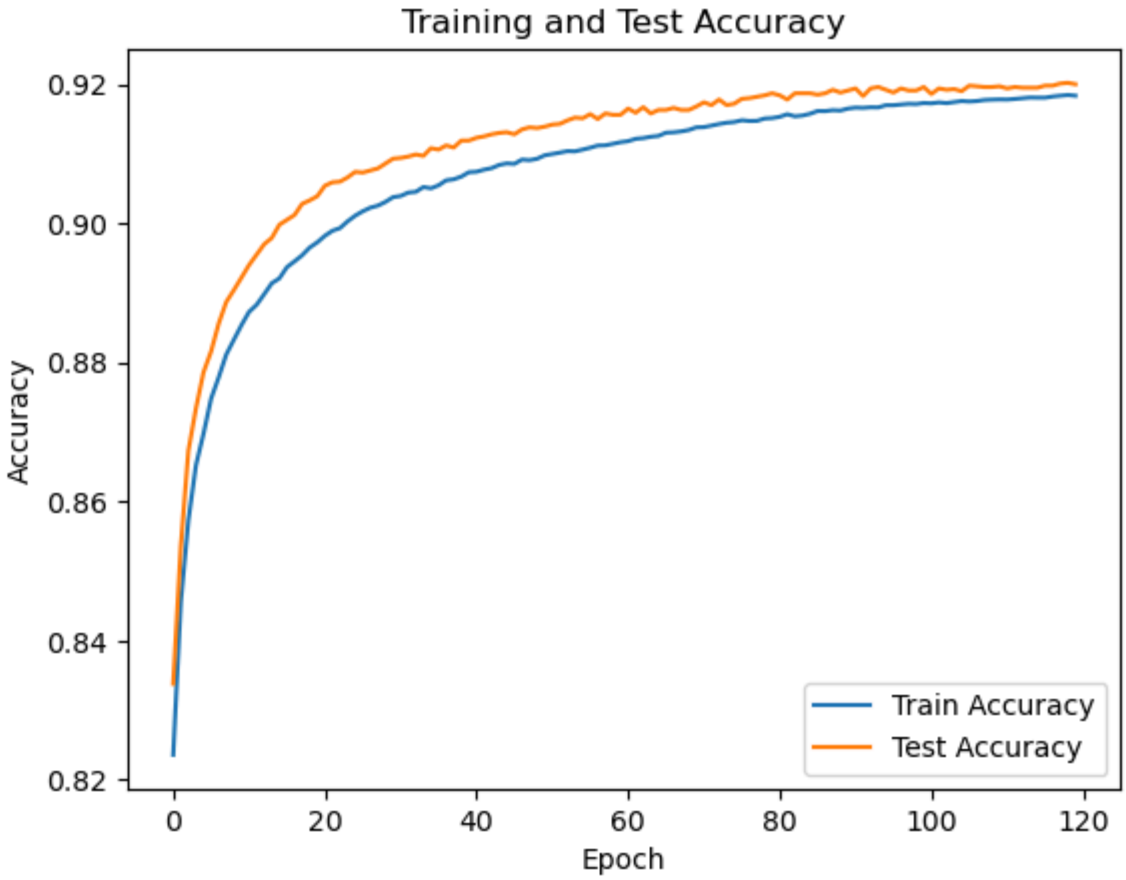
```
    #train
    model.train()
    for images,labels in train_loader:
        images=images.view(-1,784)
        outputs=model(images)
        loss=loss_func(outputs,labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    model.eval()
    #evaluation-train
    correct_train=0
    running_loss=0
    count_sample_train=0
    with torch.no_grad():
        for images,labels in train_loader:
            images=images.view(-1,784)
            outputs=model(images)
            loss=loss_func(outputs,labels)

            running_loss+=loss.item()   #tensor->number
            _,predicted=torch.max(outputs,1)
            true_label=torch.argmax(labels,dim=1)
            correct_train+=(predicted==true_label).sum().item()
            count_sample_train+=labels.shape[0]
    #print(correct_train/count_sample_train)
    loss_train.append(running_loss/len(train_loader))
    accuracy_train.append(correct_train/count_sample_train)

    #evaluation-test
    correct_test=0
    total_loss_test=0
    count_sample_test=0
    with torch.no_grad():
        for images,labels in test_loader:
            images=images.view(-1,784)
            outputs=model(images)
            loss=loss_func(outputs,labels)

            total_loss_test+=loss.item()
            _,predicted=torch.max(outputs,1)
            true_label=torch.argmax(labels,dim=1)
            correct_test+=(predicted==true_label).sum().item()
            count_sample_test+=labels.shape[0]

    loss_test.append(total_loss_test/len(test_loader))
    accuracy_test.append(correct_test/count_sample_test)
```

In [72]:

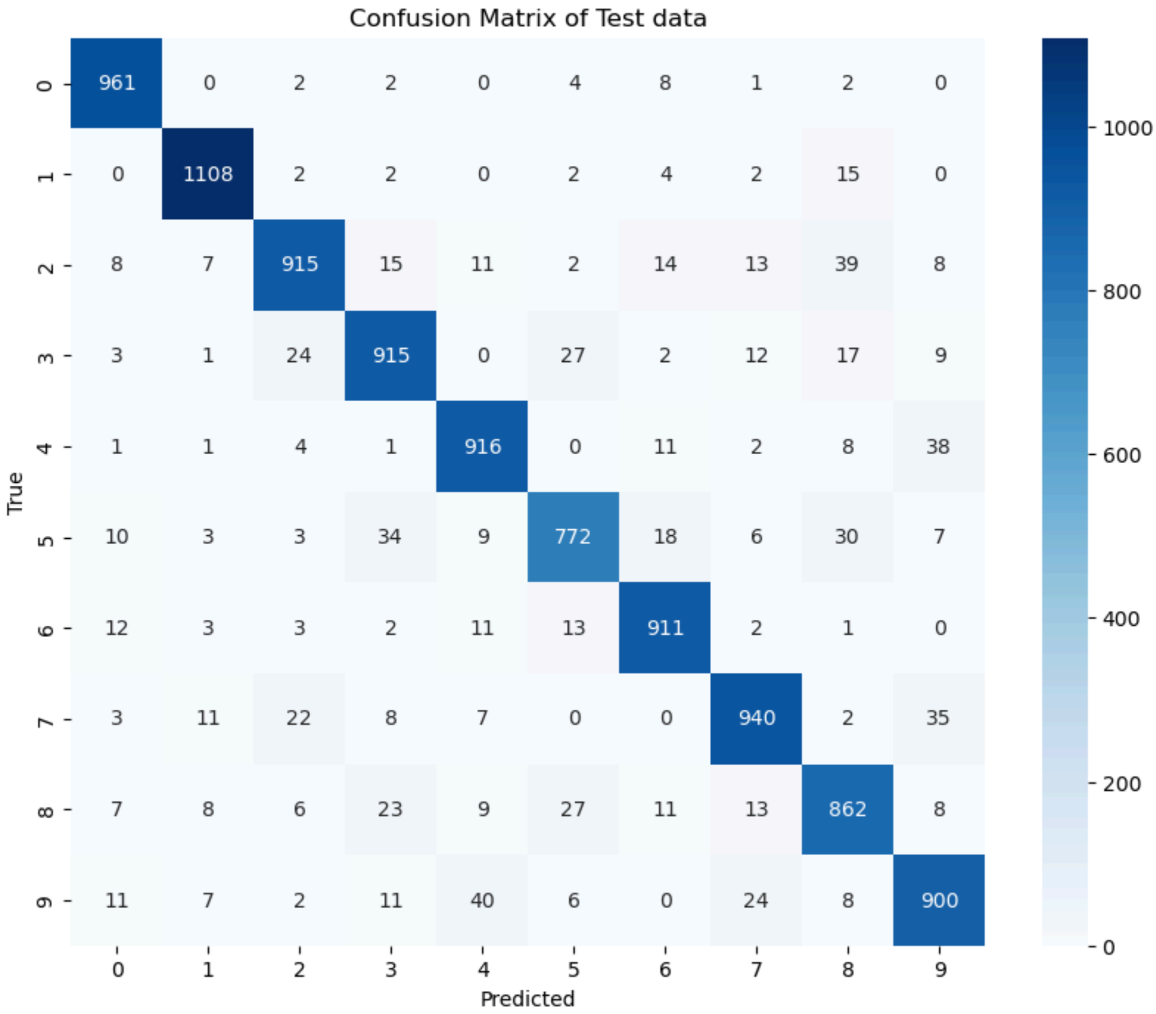
```
#show the plots
plt.figure()
plt.plot(loss_train, label='Train Loss')
plt.plot(loss_test, label='Test Loss')
plt.xlabel('Epoch')
plt.ylabel('Log-Loss')
plt.title('Training and Test Log-Loss')
plt.legend()
plt.show()

plt.figure()
```

```
plt.plot(accuracy_train, label='Train Accuracy')
plt.plot(accuracy_test, label='Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Test Accuracy')
plt.legend()
plt.show()
```





In [73]:
```
#draw condusion matrix
import seaborn as sns
from sklearn.metrics import confusion_matrix

all_preds=[]
all_labels=[]

model.eval()
with torch.no_grad():
    for images,labels in test_loader:
        images=images.view(-1,784)
        outputs=model(images)
        _, predicted=torch.max(outputs,1)
        true_label=torch.argmax(labels,dim=1)
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(true_label.cpu().numpy())

cm=confusion_matrix(all_labels,all_preds)

plt.figure(figsize=(10, 8))
sns.heatmap(cm,annot=True,fmt="d",cmap="Blues")
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix of Test data')
plt.show()
```



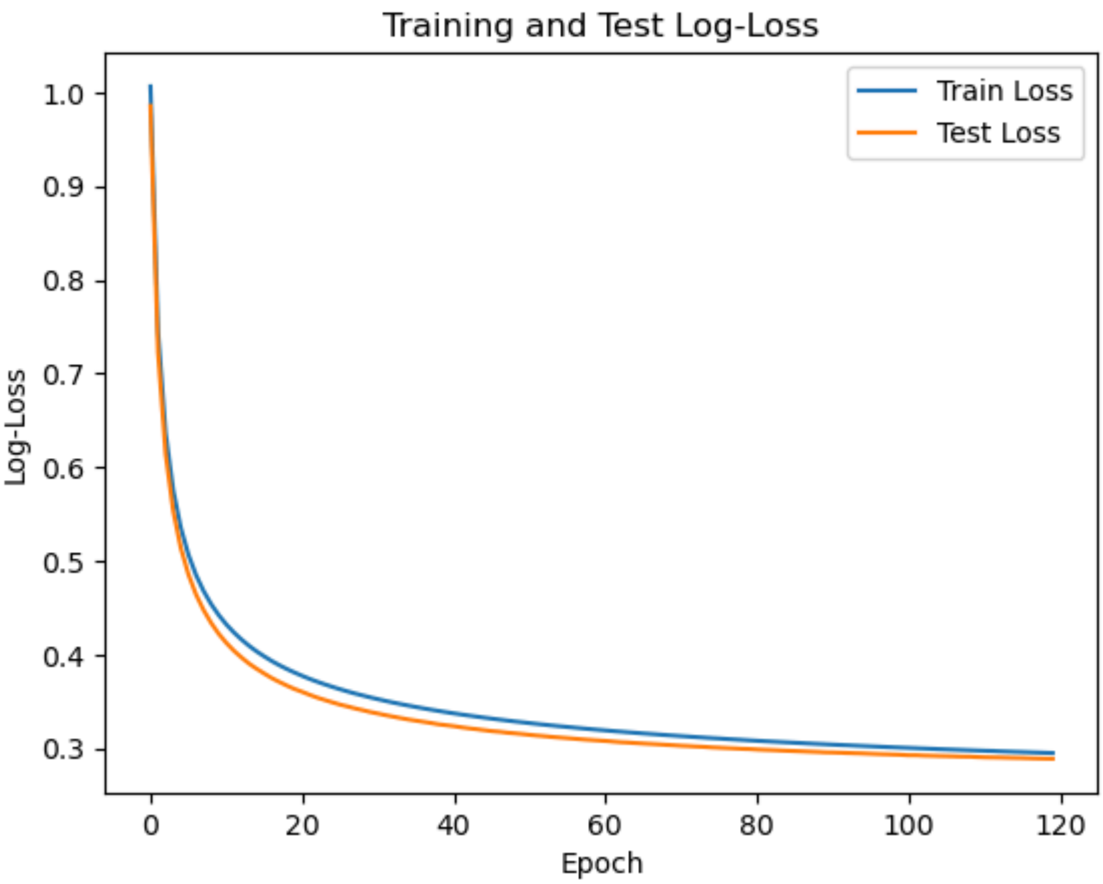
- I conducted experiments with different learning rates of 0.1, 0.01, 0.005, and 0.001. I found that when the learning rate was high, the loss decreased quickly, but as shown in the graph, it was very unstable (the curve had oscillations). When the learning rate was low, the steps were too small, resulting in very slow convergence.
- After comparison, I chose a learning rate of 0.005.

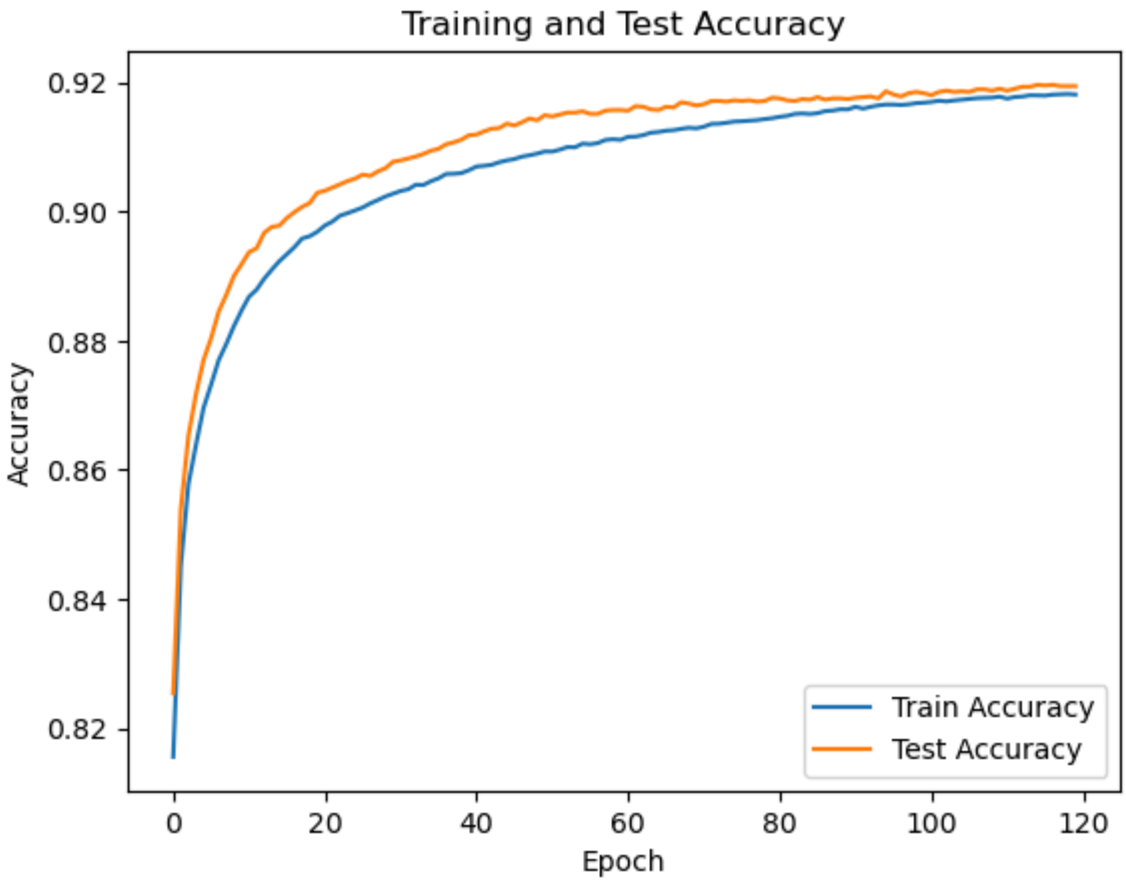
when use L2 regularization

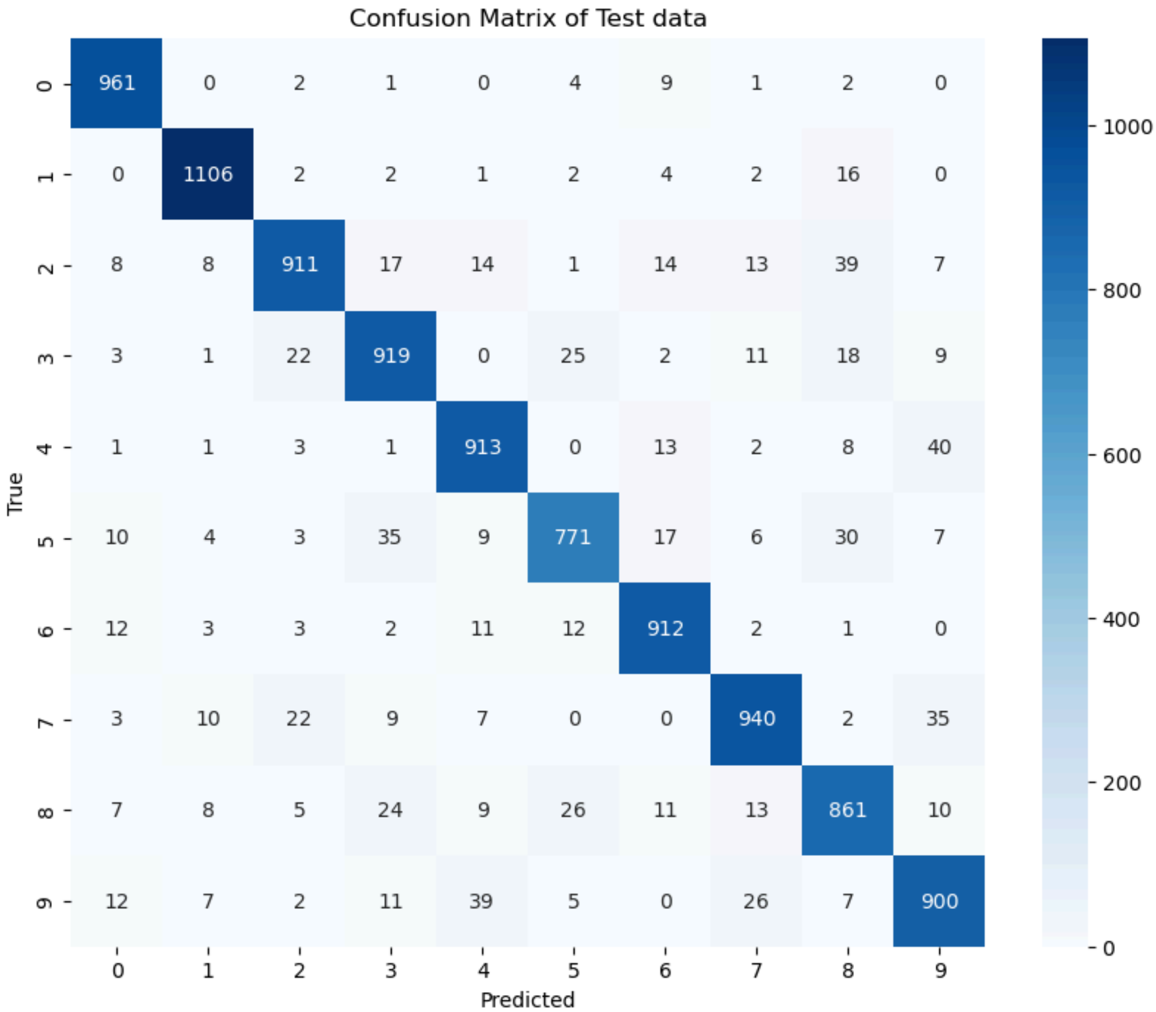
In [78]:
```
model=nn.Sequential(
    nn.Linear(784,10)
)

l2_lambda=0.0001
optimizer_l=optim.SGD(model.parameters(),lr=learning_rate,weight_decay=l2_lambda

#training loop
num_epochs=120

loss_train=[]
loss_test=[]
accuracy_train=[]
accuracy_test=[]

for epoch in range(num_epochs):
```

```
    #train
    model.train()
    for images,labels in train_loader:
        images=images.view(-1,784)
        outputs=model(images)
        loss=loss_func(outputs,labels)
        optimizer_l.zero_grad()
        loss.backward()
        optimizer_l.step()

    model.eval()
    #evaluation-train
    correct_train=0
    running_loss=0
    count_sample_train=0
    with torch.no_grad():
        for images,labels in train_loader:
            images=images.view(-1,784)
            outputs=model(images)
            loss=loss_func(outputs,labels)

            running_loss+=loss.item()   #tensor->number
            _,predicted=torch.max(outputs,1)
            true_label=torch.argmax(labels,dim=1)
            correct_train+=(predicted==true_label).sum().item()
            count_sample_train+=labels.shape[0]
    #print(correct_train/count_sample_train)
    loss_train.append(running_loss/len(train_loader))
    accuracy_train.append(correct_train/count_sample_train)

    #evaluation-test
    correct_test=0
    total_loss_test=0
    count_sample_test=0
    with torch.no_grad():
        for images,labels in test_loader:
            images=images.view(-1,784)
            outputs=model(images)
            loss=loss_func(outputs,labels)

            total_loss_test+=loss.item()
            _,predicted=torch.max(outputs,1)
            true_label=torch.argmax(labels,dim=1)
            correct_test+=(predicted==true_label).sum().item()
            count_sample_test+=labels.shape[0]

    loss_test.append(total_loss_test/len(test_loader))
    accuracy_test.append(correct_test/count_sample_test)

#show the plots
plt.figure()
plt.plot(loss_train, label='Train Loss')
plt.plot(loss_test, label='Test Loss')
plt.xlabel('Epoch')
plt.ylabel('Log-Loss')
plt.title('Training and Test Log-Loss')
plt.legend()
plt.show()

plt.figure()
```

```
plt.plot(accuracy_train, label='Train Accuracy')
plt.plot(accuracy_test, label='Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Test Accuracy')
plt.legend()
plt.show()

#draw condusion matrix
import seaborn as sns
from sklearn.metrics import confusion_matrix

all_preds=[]
all_labels=[]

model.eval()
with torch.no_grad():
    for images,labels in test_loader:
        images=images.view(-1,784)
        outputs=model(images)
        _, predicted=torch.max(outputs,1)
        true_label=torch.argmax(labels,dim=1)
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(true_label.cpu().numpy())
cm=confusion_matrix(all_labels,all_preds)

plt.figure(figsize=(10, 8))
sns.heatmap(cm,annot=True,fmt="d",cmap="Blues")
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix of Test data')
plt.show()
```




Training and Test Accuracy
Accuracy
Epoch
Train Accuracy
Test Accuracy
0.82
0.84
0.86
0.88
0.90
0.92
0
20
40
60
80
100
120


Confusion Matrix of Test data
True
Predicted
961 0 2 1 0 4 9 1 2 0
0 1106 2 2 1 2 4 2 16 0
8 8 911 17 14 1 14 13 39 7
3 1 22 919 0 25 2 11 18 9
1 1 3 1 913 0 13 2 8 40
10 4 3 35 9 771 17 6 30 7
12 3 3 2 11 12 912 2 1 0
3 10 22 9 7 0 0 940 2 35
7 8 5 24 9 26 11 13 861 10
12 7 2 11 39 5 0 26 7 900
0 1 2 3 4 5 6 7 8 9
0
200
400
600
800
1000

The results do not clearly show the effect of L2 regularization; the outcomes are almost the same.

HW6-Q2

In [37]:
```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

train_set=torchvision.datasets.FashionMNIST(root="./data",train=True,download=Tr
                                            transform=transforms.ToTensor())

train_loader=torch.utils.data.DataLoader(train_set,batch_size=100,shuffle=True)

learning_rate=0.001
num_epochs=40
```

In [38]:
```
class Model1(nn.Module):
    def __init__(self):
        super(Model1,self).__init__()
        self.hidden=nn.Linear(28*28,128)
        self.relu=nn.ReLU()
        self.output=nn.Linear(128,10)
    def forward(self,x):
        x=x.view(-1,28*28)
        x=self.hidden(x)
        x=self.relu(x)
        x=self.output(x)
        return x

class Model2(nn.Module):
    def __init__(self):
        super(Model2,self).__init__()
        self.hidden=nn.Linear(28*28,48)
        self.relu=nn.ReLU()
        self.dropout=nn.Dropout(0.2)
        self.output=nn.Linear(48,10)
    def forward(self,x):
        x=x.view(-1,28*28)
        x=self.hidden(x)
        x=self.relu(x)
        x=self.dropout(x)
        x=self.output(x)
        return x
```

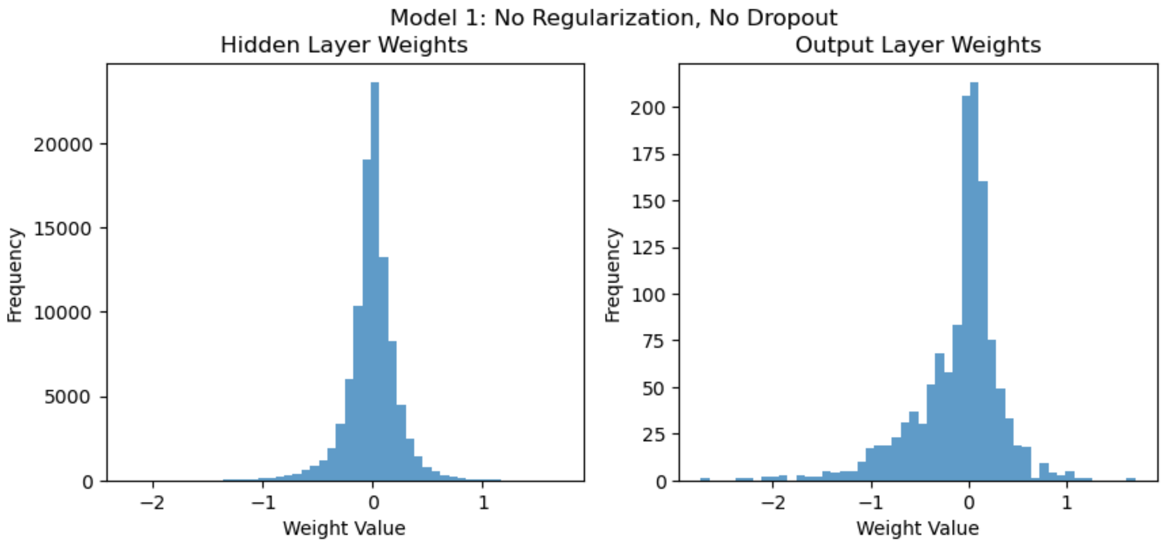
In [39]:
```
def final_wights(model):
    weights_hidden=model.hidden.weight.detach().numpy().flatten()
    weights_outpurt=model.output.weight.detach().numpy().flatten()
    return weights_hidden,weights_outpurt

def plot_histogram(weights_hidden,weights_output,title):
    plt.figure(figsize=(10, 4))
    plt.subplot(1, 2, 1)
    plt.hist(weights_hidden, bins=50, alpha=0.7)
    plt.title('Hidden Layer Weights')
    plt.xlabel('Weight Value')
    plt.ylabel('Frequency')
```

```
    plt.subplot(1, 2, 2)
    plt.hist(weights_output, bins=50, alpha=0.7)
    plt.title('Output Layer Weights')
    plt.xlabel('Weight Value')
    plt.ylabel('Frequency')
    plt.suptitle(title)
    plt.show()
```

Model 1

In [40]:
```
model_1=Model1()

loss_func_1=nn.CrossEntropyLoss()
optimizer_1=torch.optim.Adam(model_1.parameters(),lr=learning_rate)

for epoch in range(num_epochs):
    model_1.train()
    running_loss=0
    count=0
    for images,labels in train_loader:
        count+=1
        images=images.view(-1,28*28)
        outputs=model_1(images)
        loss=loss_func_1(outputs,labels)
        running_loss+=loss.item()
        optimizer_1.zero_grad()
        loss.backward()
        optimizer_1.step()
        #if count%100==0:
        #    print('iter {}: loss: {}'.format(count,loss.item()))
    print('Epoch {}:  running loss:{}'.format(epoch+1,running_loss))

weight_hidden_1,weight_output_1=final_wights(model_1)

plot_histogram(weight_hidden_1,weight_output_1,title='Model 1: No Regularization
```

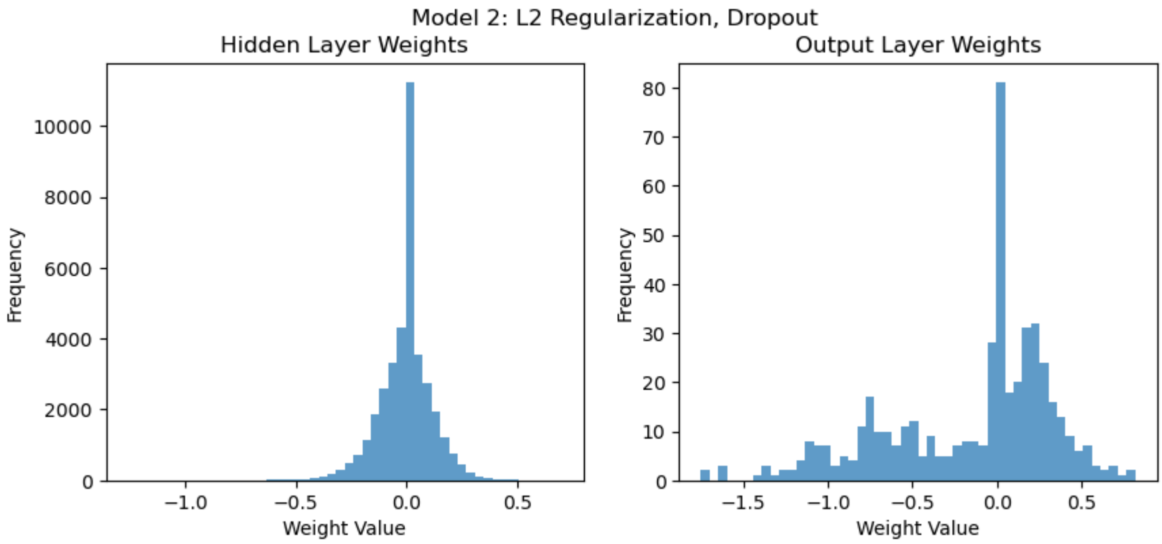
```
Epoch 1:  running loss:350.0387229323387
Epoch 2:  running loss:253.21531301736832
Epoch 3:  running loss:228.11090353131294
Epoch 4:  running loss:211.82642583549023
Epoch 5:  running loss:197.74381721019745
Epoch 6:  running loss:188.72676077485085
Epoch 7:  running loss:179.23757431656122
Epoch 8:  running loss:172.65830976516008
Epoch 9:  running loss:165.5878498852253
Epoch 10:  running loss:161.36181253939867
Epoch 11:  running loss:155.6843022108078
Epoch 12:  running loss:151.33857349306345
Epoch 13:  running loss:146.36987422406673
Epoch 14:  running loss:142.17977952212095
Epoch 15:  running loss:137.88039484620094
Epoch 16:  running loss:134.36534622311592
Epoch 17:  running loss:130.93339972943068
Epoch 18:  running loss:127.753139346838
Epoch 19:  running loss:124.35137838870287
Epoch 20:  running loss:122.77036865800619
Epoch 21:  running loss:118.14901960641146
Epoch 22:  running loss:116.63690157979727
Epoch 23:  running loss:114.65617284551263
Epoch 24:  running loss:110.76291616261005
Epoch 25:  running loss:109.05033781379461
Epoch 26:  running loss:106.20766574516892
Epoch 27:  running loss:103.83371820673347
Epoch 28:  running loss:102.4905216768384
Epoch 29:  running loss:99.38352140039206
Epoch 30:  running loss:96.92633238434792
Epoch 31:  running loss:94.76054545864463
Epoch 32:  running loss:95.01874250918627
Epoch 33:  running loss:91.65957027301192
Epoch 34:  running loss:89.71663848683238
Epoch 35:  running loss:88.00729666277766
Epoch 36:  running loss:85.33955998718739
Epoch 37:  running loss:84.96442718803883
Epoch 38:  running loss:83.23873998969793
Epoch 39:  running loss:81.71314726024866
Epoch 40:  running loss:80.64586884342134
```



Model 2

In [41]:
```
model_2=Model2()

l2_lambda=0.0001
loss_func_2=nn.CrossEntropyLoss()
optimizer_2=torch.optim.Adam(model_2.parameters(),lr=learning_rate,weight_decay=

for epoch in range(num_epochs):
    model_2.train()
    running_loss=0
    count=0
    for images,labels in train_loader:
        count+=1
        images=images.view(-1,28*28)
        outputs=model_2(images)
        loss=loss_func_2(outputs,labels)
        running_loss+=loss.item()
        optimizer_2.zero_grad()
        loss.backward()
        optimizer_2.step()
        #if count%100==0:
        #    print('iter {}: loss: {}'.format(count,loss.item()))
    print('Epoch {}:  running loss:{}'.format(epoch+1,running_loss))

weight_hidden_2,weight_output_2=final_wights(model_2)

plot_histogram(weight_hidden_2,weight_output_2,title='Model 2: L2 Regularization
```

```
Epoch 1:  running loss:432.8510921597481
Epoch 2:  running loss:294.19424054026604
Epoch 3:  running loss:264.96625447273254
Epoch 4:  running loss:251.96432764828205
Epoch 5:  running loss:241.13772474229336
Epoch 6:  running loss:234.85241790115833
Epoch 7:  running loss:228.36463464796543
Epoch 8:  running loss:224.66773469746113
Epoch 9:  running loss:220.19999679923058
Epoch 10:  running loss:215.74759720265865
Epoch 11:  running loss:213.7591621428728
Epoch 12:  running loss:210.80267351865768
Epoch 13:  running loss:209.5185059159994
Epoch 14:  running loss:206.46534241735935
Epoch 15:  running loss:204.12598338723183
Epoch 16:  running loss:202.9848313331604
Epoch 17:  running loss:201.1302878111601
Epoch 18:  running loss:198.93640778958797
Epoch 19:  running loss:198.46974915266037
Epoch 20:  running loss:196.7248513251543
Epoch 21:  running loss:195.54534024000168
Epoch 22:  running loss:193.4704591035843
Epoch 23:  running loss:191.45627450942993
Epoch 24:  running loss:192.28253589570522
Epoch 25:  running loss:189.72827829420567
Epoch 26:  running loss:188.1090660393238
Epoch 27:  running loss:188.0213242173195
Epoch 28:  running loss:187.0044487863779
Epoch 29:  running loss:186.72674468159676
Epoch 30:  running loss:185.7998018413782
Epoch 31:  running loss:185.9963295608759
Epoch 32:  running loss:184.98358605057
Epoch 33:  running loss:184.15589614212513
Epoch 34:  running loss:183.63885389268398
Epoch 35:  running loss:180.99878773093224
Epoch 36:  running loss:182.51904601603746
Epoch 37:  running loss:179.7699525654316
Epoch 38:  running loss:181.43181063234806
Epoch 39:  running loss:180.9161752462387
Epoch 40:  running loss:181.05362345278263
```



Q1: Describe the qualitative differences between these histograms.

- **Model 1 (No regularization, no dropout)**:
  - The hidden layer weights have a relatively wide distribution range. Most weights are concentrated around 0, but the weights span approximately from -1 to 1.
  - The output layer weights also have a relatively wide distribution, with values approximately between -2 and 1.
  - Overall, Model 1's weight distribution is more "spread out," showing larger weight values, which is common when there is no regularization.
- **Model 2 (L2 regularization and dropout)**:
  - The hidden layer weights are more concentrated around 0, with a narrower range, with most weights between -0.5 and 0.5.
  - The output layer weights are similarly concentrated around 0 and have a smaller range compared to Model 1, mostly between -1.5 and 0.5.
  - In general, Model 2's weight distribution is more "compressed," with weights closer to 0.

Q2: What effect does regularization have on the distribution of weights

- **L2 Regularization** tends to penalize large weight values, encouraging weights to stay closer to 0. This results in the weights being more tightly clustered around 0 in Model 2, giving a narrower distribution.

- **Dropout** randomly "drops" neurons during training, which indirectly limits the size of the weights. The model needs to maintain performance despite the absence of certain neurons, thus contributing to limiting the weights' magnitude.

HW6-Q3

In [15]:
```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import torch.utils.data as data
import torch.optim as optim
import matplotlib.pyplot as plt

train_set=torchvision.datasets.CIFAR10(root="./data",train=True,download=True,
                                       transform=transforms.ToTensor())
test_set=torchvision.datasets.CIFAR10(root="./data",train=False,download=True,
                                      transform=transforms.ToTensor())

train_loader=data.DataLoader(train_set,batch_size=100,shuffle=True)
test_loader=data.DataLoader(test_set,batch_size=100,shuffle=False)

class MLP(nn.Module):
    def __init__(self):
        super(MLP,self).__init__()
        self.network=nn.Sequential(
            nn.Flatten(),
            nn.Linear(3*32*32,256),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(256,128),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(128,10)
        )
    def forward(self,x):
        return self.network(x)

model=MLP()

learning_rate=0.01
l2_lambda=0.0001

loss_func=nn.CrossEntropyLoss()
optimizer=optim.SGD(model.parameters(),lr=learning_rate,weight_decay=l2_lambda)
```

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

In [16]:
```
num_epoch=100

for epoch in range(num_epoch):
    model.train()
    running_loss=0
    for images,labels in train_loader:
        outputs=model(images)
        loss=loss_func(outputs,labels)
        running_loss+=loss
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

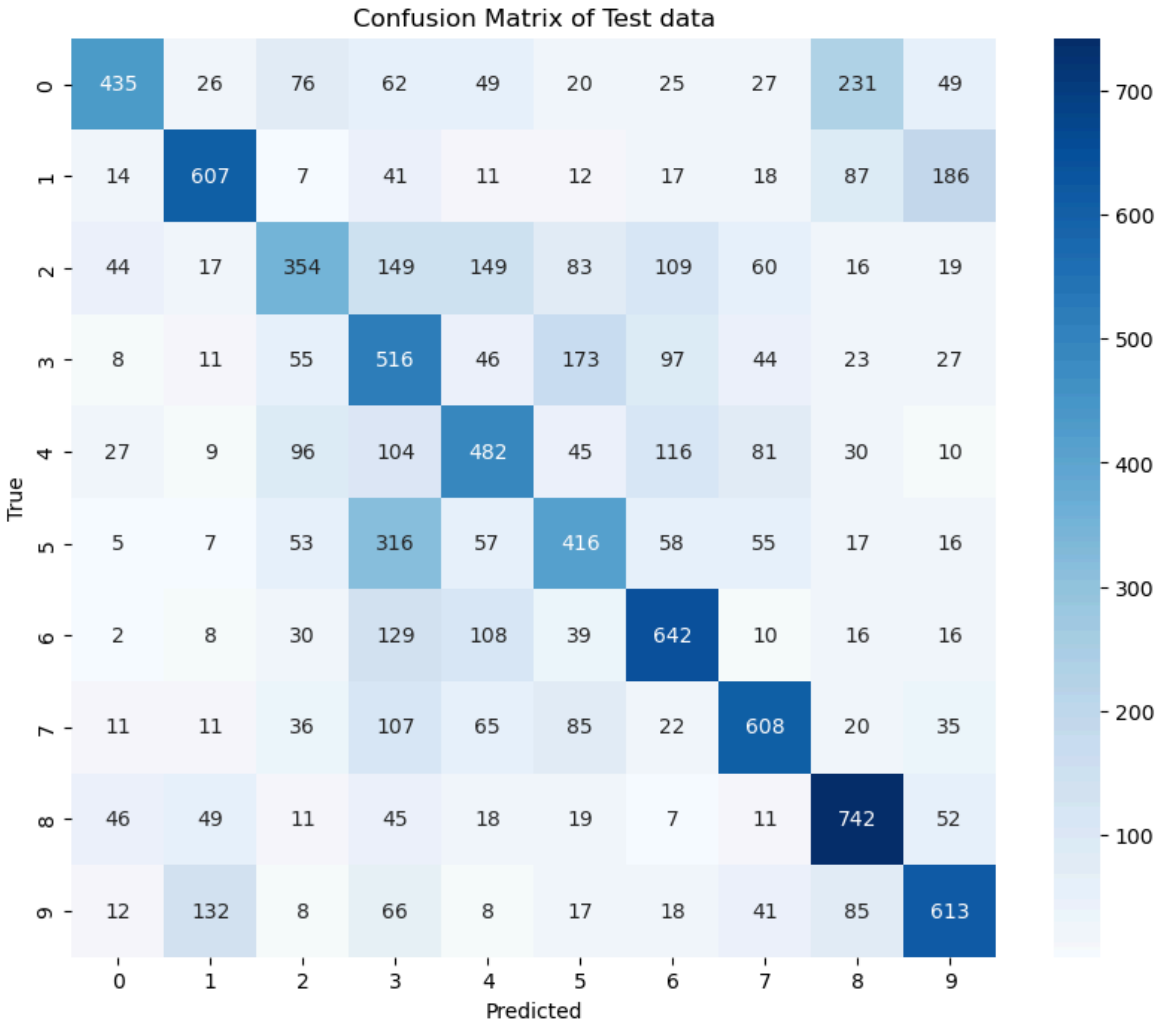
```
    print('Epoch {} - loss: {}'.format(epoch,running_loss))

all_preds=[]
all_labels=[]

model.eval()
with torch.no_grad():
    for images,labels in test_loader:
        outputs=model(images)
        loss=loss_func(outputs,labels)
        _,predicted=torch.max(outputs,1)
        all_preds.extend(predicted.numpy())
        all_labels.extend(labels.numpy())
```

```
Epoch 0 - loss: 1115.5830078125
Epoch 1 - loss: 1021.7300415039062
Epoch 2 - loss: 978.0801391601562
Epoch 3 - loss: 952.0846557617188
Epoch 4 - loss: 932.6698608398438
Epoch 5 - loss: 916.4351196289062
Epoch 6 - loss: 902.0715942382812
Epoch 7 - loss: 889.9658203125
Epoch 8 - loss: 879.0838623046875
Epoch 9 - loss: 869.3945922851562
Epoch 10 - loss: 860.4180297851562
Epoch 11 - loss: 851.1945190429688
Epoch 12 - loss: 843.663330078125
Epoch 13 - loss: 835.9638671875
Epoch 14 - loss: 830.2979125976562
Epoch 15 - loss: 823.1368408203125
Epoch 16 - loss: 816.7457275390625
Epoch 17 - loss: 811.6392211914062
Epoch 18 - loss: 805.4404907226562
Epoch 19 - loss: 799.4823608398438
Epoch 20 - loss: 793.977294921875
Epoch 21 - loss: 788.6806030273438
Epoch 22 - loss: 783.8004760742188
Epoch 23 - loss: 778.8466186523438
Epoch 24 - loss: 775.9365844726562
Epoch 25 - loss: 771.15673828125
Epoch 26 - loss: 767.33349609375
Epoch 27 - loss: 762.5327758789062
Epoch 28 - loss: 760.2673950195312
Epoch 29 - loss: 755.841796875
Epoch 30 - loss: 752.6836547851562
Epoch 31 - loss: 748.1321411132812
Epoch 32 - loss: 745.6010131835938
Epoch 33 - loss: 741.7354125976562
Epoch 34 - loss: 738.0830688476562
Epoch 35 - loss: 734.119140625
Epoch 36 - loss: 730.2850341796875
Epoch 37 - loss: 727.86328125
Epoch 38 - loss: 724.5682983398438
Epoch 39 - loss: 722.3261108398438
Epoch 40 - loss: 720.2451171875
Epoch 41 - loss: 715.8592529296875
Epoch 42 - loss: 712.8264770507812
Epoch 43 - loss: 711.372802734375
Epoch 44 - loss: 707.0244140625
Epoch 45 - loss: 705.709228515625
Epoch 46 - loss: 702.57177734375
Epoch 47 - loss: 698.5643920898438
Epoch 48 - loss: 698.3269653320312
Epoch 49 - loss: 693.7485961914062
Epoch 50 - loss: 692.7638549804688
Epoch 51 - loss: 691.4204711914062
Epoch 52 - loss: 687.90625
Epoch 53 - loss: 684.0046997070312
Epoch 54 - loss: 681.9351806640625
Epoch 55 - loss: 679.922119140625
Epoch 56 - loss: 681.682373046875
Epoch 57 - loss: 679.185546875
Epoch 58 - loss: 675.5855712890625
Epoch 59 - loss: 672.2362670898438
```

```
Epoch 60 - loss: 670.3721923828125
Epoch 61 - loss: 667.3853149414062
Epoch 62 - loss: 667.07080078125
Epoch 63 - loss: 664.8051147460938
Epoch 64 - loss: 662.336181640625
Epoch 65 - loss: 660.668701171875
Epoch 66 - loss: 658.658447265625
Epoch 67 - loss: 658.1076049804688
Epoch 68 - loss: 656.8859252929688
Epoch 69 - loss: 652.122314453125
Epoch 70 - loss: 650.6231079101562
Epoch 71 - loss: 651.99755859375
Epoch 72 - loss: 648.3004150390625
Epoch 73 - loss: 646.3712768554688
Epoch 74 - loss: 642.7102661132812
Epoch 75 - loss: 642.4271850585938
Epoch 76 - loss: 640.4236450195312
Epoch 77 - loss: 639.564453125
Epoch 78 - loss: 639.0795288085938
Epoch 79 - loss: 637.21826171875
Epoch 80 - loss: 634.5625610351562
Epoch 81 - loss: 632.9602661132812
Epoch 82 - loss: 633.1918334960938
Epoch 83 - loss: 629.15966796875
Epoch 84 - loss: 628.8482055664062
Epoch 85 - loss: 627.7118530273438
Epoch 86 - loss: 626.3252563476562
Epoch 87 - loss: 623.9664306640625
Epoch 88 - loss: 622.150634765625
Epoch 89 - loss: 621.0182495117188
Epoch 90 - loss: 620.7442016601562
Epoch 91 - loss: 619.5210571289062
Epoch 92 - loss: 616.933837890625
Epoch 93 - loss: 616.4555053710938
Epoch 94 - loss: 615.8231811523438
Epoch 95 - loss: 612.2374267578125
Epoch 96 - loss: 611.1432495117188
Epoch 97 - loss: 611.5711059570312
Epoch 98 - loss: 609.9839477539062
Epoch 99 - loss: 605.0579833984375
```

In [17]:

```python
import seaborn as sns
from sklearn.metrics import confusion_matrix

cm=confusion_matrix(all_labels,all_preds)

plt.figure(figsize=(10, 8))
sns.heatmap(cm,annot=True,fmt="d",cmap="Blues")
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix of Test data')
plt.show()
```



(a) Consider class m. List the class most likely confused for class m for each object type.

In [22]:

```
import numpy as np
for i in range(10):
    confused_classes=np.argsort(-cm[i,:])   #minus + sort increase = the biggest
    for confused_class in confused_classes:
        if confused_class!=i:
            print('For class {} ({}), the most likely confused class is {} ({})'
                  i,test_set.classes[i],confused_class,test_set.classes[confuse
            break
```

```
For class 0 (airplane), the most likely confused class is 8 (ship)
For class 1 (automobile), the most likely confused class is 9 (truck)
For class 2 (bird), the most likely confused class is 3 (cat)
For class 3 (cat), the most likely confused class is 5 (dog)
For class 4 (deer), the most likely confused class is 6 (frog)
For class 5 (dog), the most likely confused class is 3 (cat)
For class 6 (frog), the most likely confused class is 3 (cat)
For class 7 (horse), the most likely confused class is 3 (cat)
For class 8 (ship), the most likely confused class is 9 (truck)
For class 9 (truck), the most likely confused class is 1 (automobile)
```

(b) Which two classes (object types) are most likely to be confused overall?

In [23]:
```
cm_no_diagnal=cm-np.eye(10)*cm.diagonal()
idx_max_error=np.argmax(cm_no_diagnal,axis=None)
most_confused_classes = np.unravel_index(idx_max_error,cm.shape)
print('class {} ({}) and class {} ({}) are most likely to be confused overall'.f
    most_confused_classes[0],test_set.classes[most_confused_classes[0]],most_con
```

```
class 5 (dog) and class 3 (cat) are most likely to be confused overall
```