Comparison of Different classification methods

```
from keras.datasets import mnist
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

```
WARNING:tensorflow:From D:\Python3.11\Lib\site-packages\keras\src\losses.py:2976: The name t
```

```
function= lambda x: x.flatten()
X_train= np.array(list(map(function,X_train)))
X_test= np.array(list(map(function,X_test)))
```

```
results= pd.DataFrame(data=y_test,columns=["true_value"])
```

Logistic Regression

```
from sklearn.linear_model import LogisticRegression
lg = LogisticRegression()
lg.fit(X_train,y_train)
predicted = lg.predict(X_test)
results['Log_Regression']=predicted
```

```
D:\Python3.11\Lib\site-packages\sklearn\linear_model\_logistic.py:460: ConvergenceWarning: l
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
```

```
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

```
predicted
```

```
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

```
y_train.shape
```

```
(60000,)
```

MLP

sklearn.neural_network.MLPClassifier(hidden_layer_sizes=(100,), activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000)[source]¶

MLP with one hidden layer 128

```
from sklearn.neural_network import MLPClassifier
clf = MLPClassifier( alpha=1e-5,
                     hidden_layer_sizes=(128, ), random_state=1)
clf.fit(X_train,y_train)
predicted=clf.predict(X_test)
results['MLP_1']=predicted
```

MLP with two hidden layers 256 - 128

```
clf = MLPClassifier( alpha=1e-5,
                     hidden_layer_sizes=(128,256 ), random_state=1)
clf.fit(X_train,y_train)
predicted=clf.predict(X_test)
results['MLP_2']=predicted
```

```
results
```

	true_value	Log_Regression	MLP_1	MLP_2
0	7	7	7	7
1	2	2	2	2
2	1	1	1	1
3	0	0	0	0
4	4	4	4	4
...	...	...	...	...
9995	2	2	2	2
9996	3	3	3	3
9997	4	4	4	4
9998	5	5	5	5
9999	6	6	6	6

CNN

```
import torch
import torchvision
import torchvision.transforms as transforms
```

```
import torch.nn as nn
import torch.nn.functional as F
```

```
from torch.utils.data import DataLoader
loaders = {
    'train' : torch.utils.data.DataLoader(train_data,
                                          batch_size=100,
                                          shuffle=True,
```

```python
                                          num_workers=1),

    'test'  : torch.utils.data.DataLoader(test_data,
                                          batch_size=100,
                                          shuffle=True,
                                          num_workers=1),
}
loaders
```

```
{'train': <torch.utils.data.dataloader.DataLoader at 0x1c289a791d0>,
 'test': <torch.utils.data.dataloader.DataLoader at 0x1c2899ea410>}
```

```python
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

Importing data

```python
from torchvision import datasets
from torchvision.transforms import ToTensor
train_data = datasets.MNIST(
    root = 'data',
    train = True,
    transform = ToTensor(),
    download = True,
)
test_data = datasets.MNIST(
    root = 'data',
    train = False,
    transform = ToTensor()
)
```

```python
from torch.utils.data import DataLoader
loaders = {
    'train' : torch.utils.data.DataLoader(train_data,
                                          batch_size=100,
                                          shuffle=False,
                                          num_workers=1),
```

10.2.5 Multi-dimensional convolutions

Section 6.3.7

So far we have considered convolutions over a single grey-scale image. For a colour image there will be three channels corresponding to the red, green, and blue colours. We can easily extend convolutions to cover multiple channels by extending the dimensionality of the filter. An image with $J \times K$ pixels and C channels will be described by a *tensor* of dimensionality $J \times K \times C$. We can introduce a filter described by a tensor of dimensionality $M \times M \times C$ comprising a separate $M \times M$ filter for each of the C channels. Assuming no padding and a stride of 1, this again gives a feature map of size $(J-M+1)\times(K-M+1)$, as is illustrated in Figure 10.6.

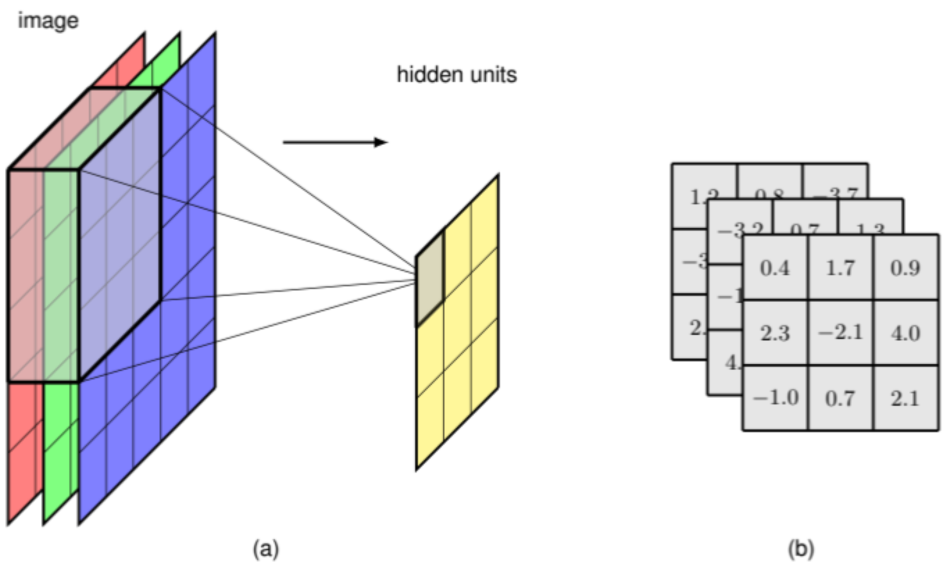


Figure 10.6 (a) Illustration of a multi-dimensional filter that takes input from across the R, G, and B channels. (b) The kernel here has 27 weights (plus a bias parameter not shown) and can be visualized as a $3 \times 3 \times 3$ tensor.

Figure 1: image.png

```python
    'test'  : torch.utils.data.DataLoader(test_data,
                                          batch_size=100,
                                          shuffle=False,
                                          num_workers=1),
}
loaders
```

```
{'train': <torch.utils.data.dataloader.DataLoader at 0x1c2a9766150>,
 'test': <torch.utils.data.dataloader.DataLoader at 0x1c289589cd0>}
```

Calculating the feature map dimensions \$28-5+1 +2*2= 28 / 2 > 14 \$ $14 - 5 + 1 + 2 * 2 = 14/2 = 7$

5 is the filter (kernel) dimensions. A padding of $2 * 2$ adds $2 * 2$ dimensions. A maxpool of (2,2) half the dimensions

```python
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=1,out_channels= 16, kernel_size=5,padding=2)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(in_channels=16,out_channels= 32, kernel_size=5,padding=2)
        self.fc1 = nn.Linear(32*7*7, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        #print(x.shape)

        x = x.view(x.size(0), -1)# flatten all dimensions except batch
        #print(x.shape)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

```
net = Net()
```

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.001)
```

```
from torch.autograd import Variable
num_epochs = 10
def train(num_epochs, net, loaders):

    net.train()

    # Train the model
    total_step = len(loaders['train'])

    for epoch in range(num_epochs):
        for i, (images, labels) in enumerate(loaders['train']):

            # gives batch data, normalize x when iterate train_loader
            b_x = Variable(images)   # batch x
            b_y = Variable(labels)   # batch y
            output = net(b_x)
            loss = criterion(output, b_y)

            # clear gradients for this training step
            optimizer.zero_grad()

            # backpropagation, compute gradients
            loss.backward()
            # apply gradients
            optimizer.step()

            if (i+1) % 100 == 0:
                print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                       .format(epoch + 1, num_epochs, i + 1, total_step, loss.item()))
                pass

            pass
```

```
        pass
train(num_epochs, net, loaders)
```

```
Epoch [1/10], Step [100/600], Loss: 0.3870
Epoch [1/10], Step [200/600], Loss: 0.1651
Epoch [1/10], Step [300/600], Loss: 0.1293
Epoch [1/10], Step [400/600], Loss: 0.0960
Epoch [1/10], Step [500/600], Loss: 0.1082
Epoch [1/10], Step [600/600], Loss: 0.0592
Epoch [2/10], Step [100/600], Loss: 0.0218
Epoch [2/10], Step [200/600], Loss: 0.0706
Epoch [2/10], Step [300/600], Loss: 0.1215
Epoch [2/10], Step [400/600], Loss: 0.0412
Epoch [2/10], Step [500/600], Loss: 0.0093
Epoch [2/10], Step [600/600], Loss: 0.0189
Epoch [3/10], Step [100/600], Loss: 0.0544
Epoch [3/10], Step [200/600], Loss: 0.1261
Epoch [3/10], Step [300/600], Loss: 0.0042
Epoch [3/10], Step [400/600], Loss: 0.0661
Epoch [3/10], Step [500/600], Loss: 0.0916
Epoch [3/10], Step [600/600], Loss: 0.0133
Epoch [4/10], Step [100/600], Loss: 0.0274
Epoch [4/10], Step [200/600], Loss: 0.0165
Epoch [4/10], Step [300/600], Loss: 0.0170
Epoch [4/10], Step [400/600], Loss: 0.0784
Epoch [4/10], Step [500/600], Loss: 0.0035
Epoch [4/10], Step [600/600], Loss: 0.0135
Epoch [5/10], Step [100/600], Loss: 0.0156
Epoch [5/10], Step [200/600], Loss: 0.0174
Epoch [5/10], Step [300/600], Loss: 0.0129
Epoch [5/10], Step [400/600], Loss: 0.0371
Epoch [5/10], Step [500/600], Loss: 0.0282
Epoch [5/10], Step [600/600], Loss: 0.0114
Epoch [6/10], Step [100/600], Loss: 0.0117
Epoch [6/10], Step [200/600], Loss: 0.0452
Epoch [6/10], Step [300/600], Loss: 0.0030
Epoch [6/10], Step [400/600], Loss: 0.0071
Epoch [6/10], Step [500/600], Loss: 0.0033
Epoch [6/10], Step [600/600], Loss: 0.0137
Epoch [7/10], Step [100/600], Loss: 0.0057
Epoch [7/10], Step [200/600], Loss: 0.0015
Epoch [7/10], Step [300/600], Loss: 0.0522
```

```
Epoch [7/10], Step [400/600], Loss: 0.0294
Epoch [7/10], Step [500/600], Loss: 0.0021
Epoch [7/10], Step [600/600], Loss: 0.0029
Epoch [8/10], Step [100/600], Loss: 0.0401
Epoch [8/10], Step [200/600], Loss: 0.0223
Epoch [8/10], Step [300/600], Loss: 0.0030
Epoch [8/10], Step [400/600], Loss: 0.0076
Epoch [8/10], Step [500/600], Loss: 0.0013
Epoch [8/10], Step [600/600], Loss: 0.0238
Epoch [9/10], Step [100/600], Loss: 0.0022
Epoch [9/10], Step [200/600], Loss: 0.0245
Epoch [9/10], Step [300/600], Loss: 0.0274
Epoch [9/10], Step [400/600], Loss: 0.0143
Epoch [9/10], Step [500/600], Loss: 0.0037
Epoch [9/10], Step [600/600], Loss: 0.0181
Epoch [10/10], Step [100/600], Loss: 0.0867
Epoch [10/10], Step [200/600], Loss: 0.0259
Epoch [10/10], Step [300/600], Loss: 0.0014
Epoch [10/10], Step [400/600], Loss: 0.0053
Epoch [10/10], Step [500/600], Loss: 0.0015
Epoch [10/10], Step [600/600], Loss: 0.0076
```

```python
predicted=[]
def test():
    # Test the model
    net.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        for images, labels in loaders['test']:
            test_output = net(images)
            pred_y = torch.max(test_output, 1)[1].data.squeeze()
            predicted.extend(list(pred_y.numpy()))
            accuracy = (pred_y == labels).sum().item() / float(labels.size(0))
            pass

    print('Test Accuracy of the model on the 10000 test images: %.2f' % accuracy)
    pass

test()
```

```
Test Accuracy of the model on the 10000 test images: 1.00
```

```
results['CNN']=predicted
```

```
results
```

	true_value	Log_Regression	MLP_1	MLP_2	CNN
0	7	7	7	7	7
1	2	2	2	2	2
2	1	1	1	1	1
3	0	0	0	0	0
4	4	4	4	4	4
...	...	...	...	...	...
9995	2	2	2	2	2
9996	3	3	3	3	3
9997	4	4	4	4	4
9998	5	5	5	5	5
9999	6	6	6	6	6

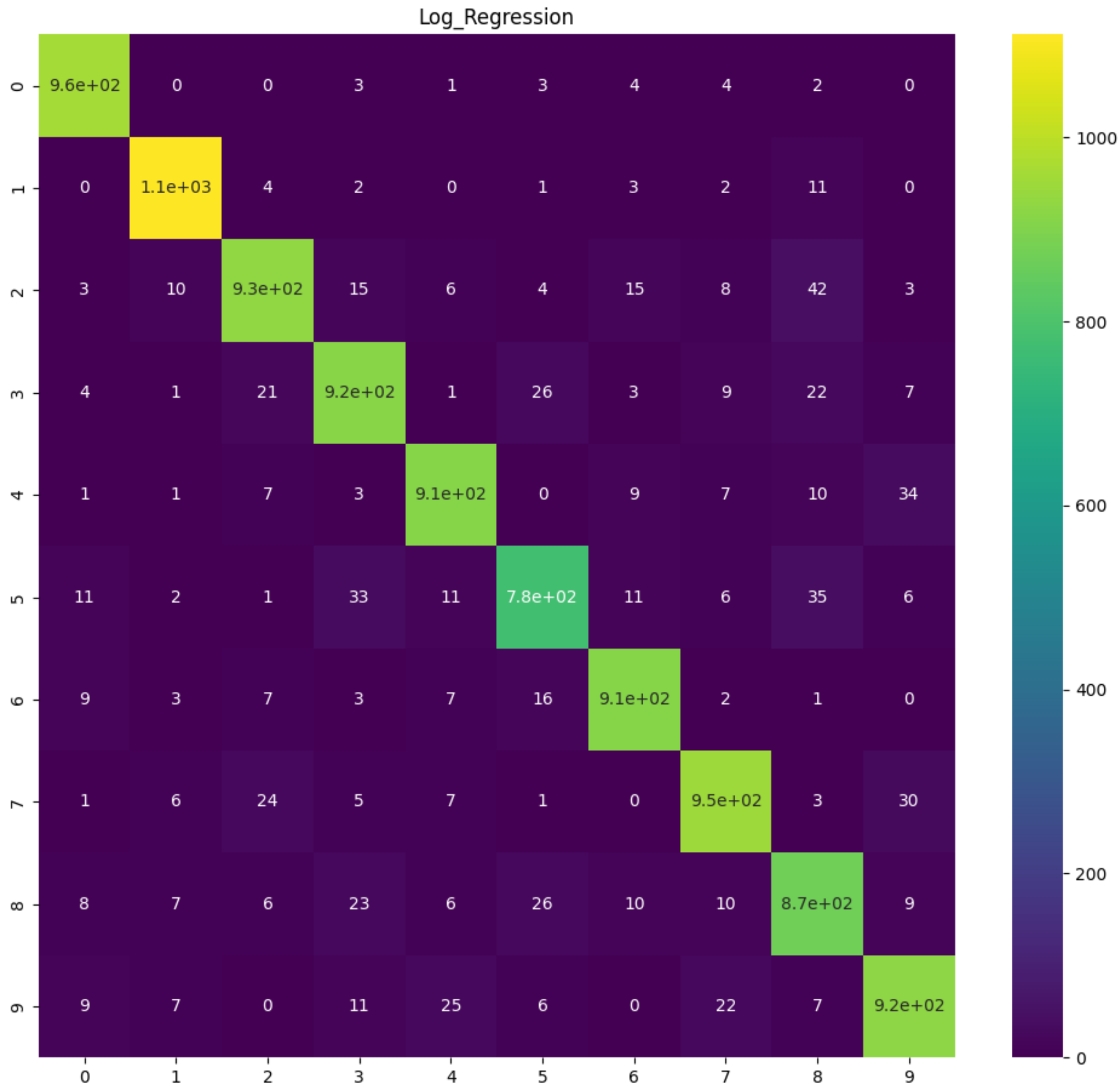
Comparison

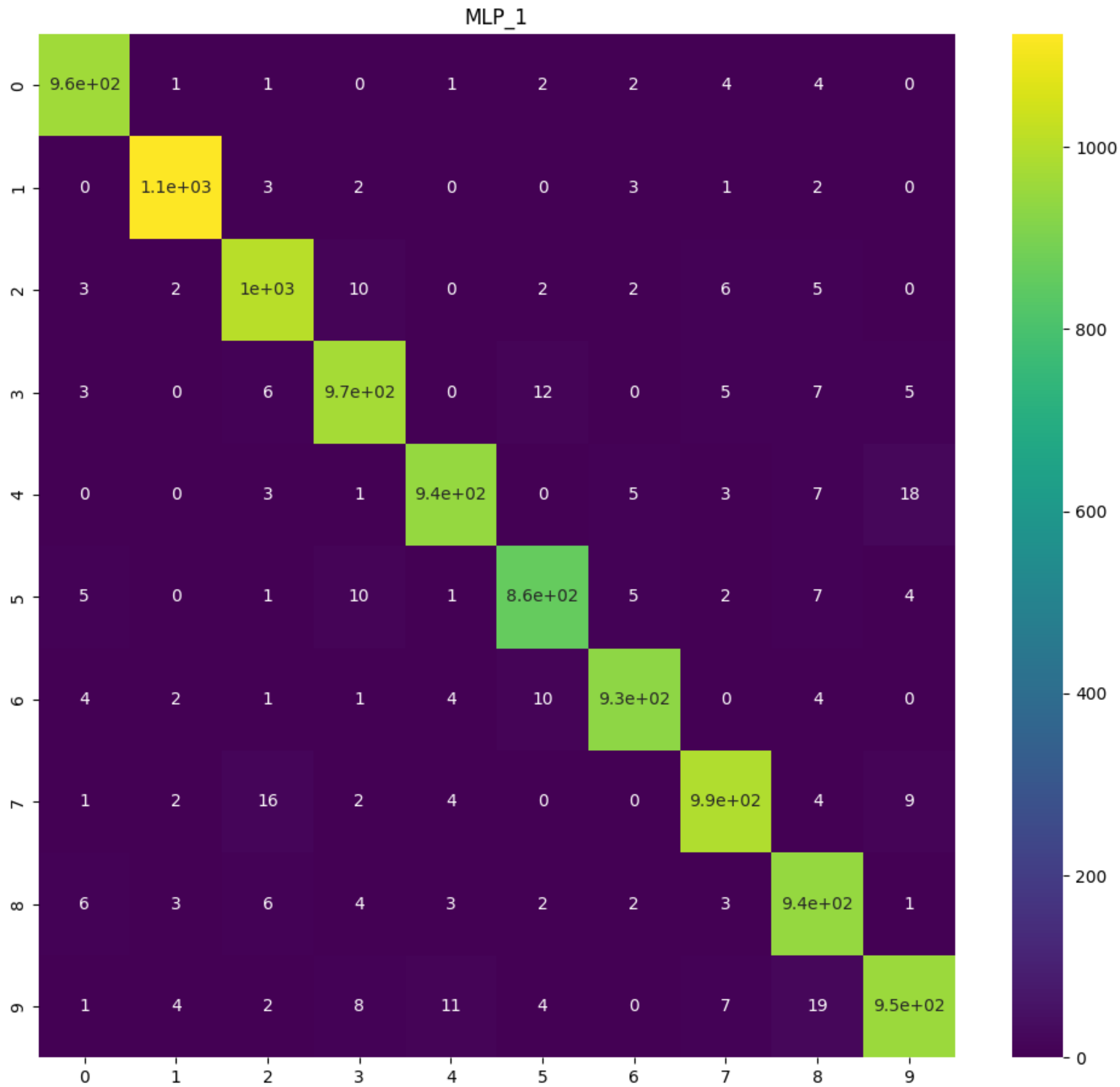
```
from sklearn.metrics import confusion_matrix,classification_report
performance_metrics= pd.DataFrame()
i=0
accuracy={}
for method in results.columns[1:]:
    print(f"                      ******{method}******")

    print(100*"*")
    fig,ax=plt.subplots(1,1,figsize=(10,10))
    confusion_mat= confusion_matrix(results['true_value'],results[method])
    report= classification_report(results['true_value'],results[method])
    report_dic=classification_report(results['true_value'],results[method],output_dict=Tru
    accuracy[method]=report_dic["accuracy"]
    print(report)
    ax.set_title(f"{method}")
    sns.heatmap(confusion_mat,annot=True,cmap="viridis")
    fig.tight_layout(rect=[0, 0.03, 1, 0.95])
```

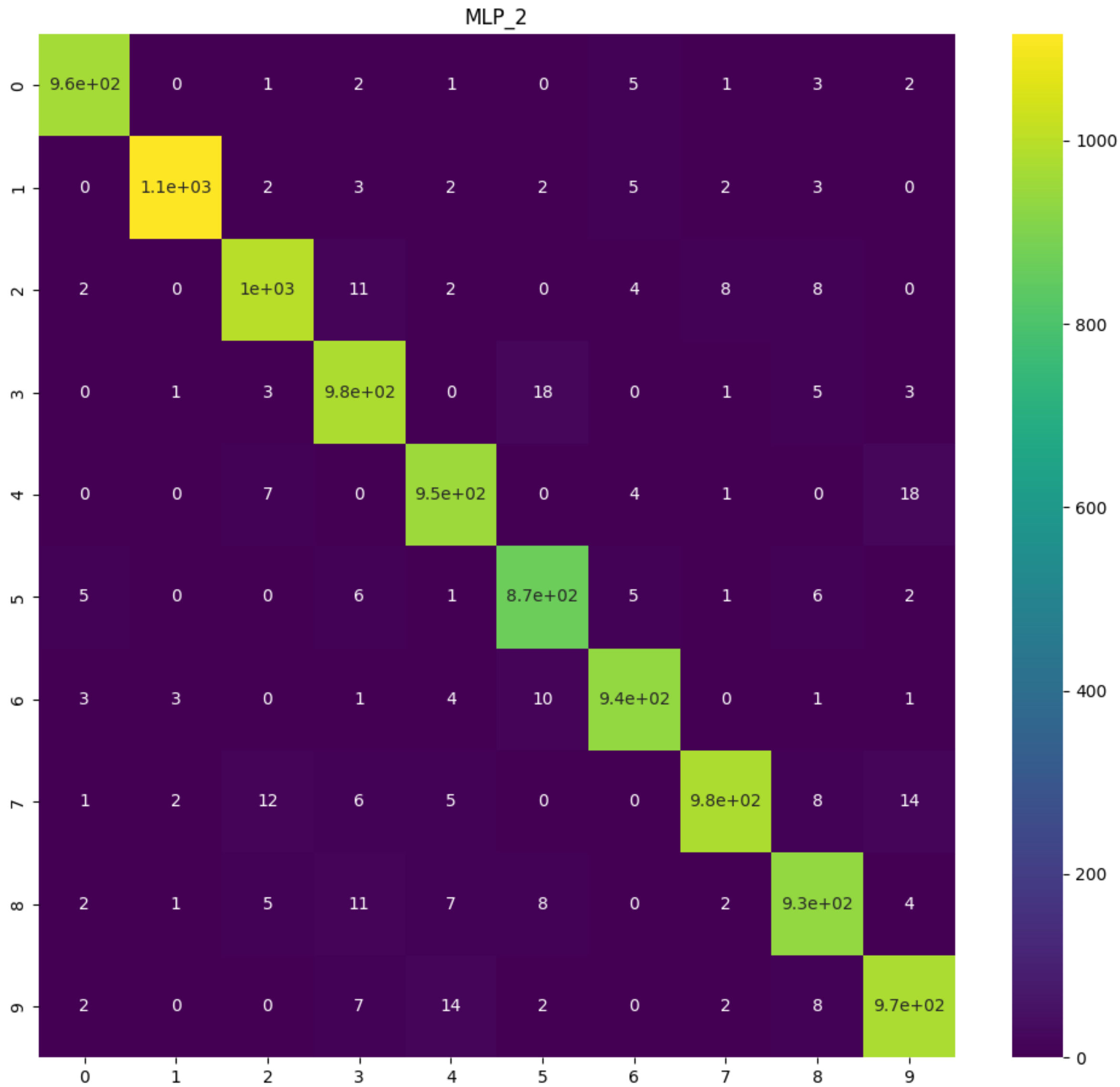
```python
plt.show()
print(100*"*")
```

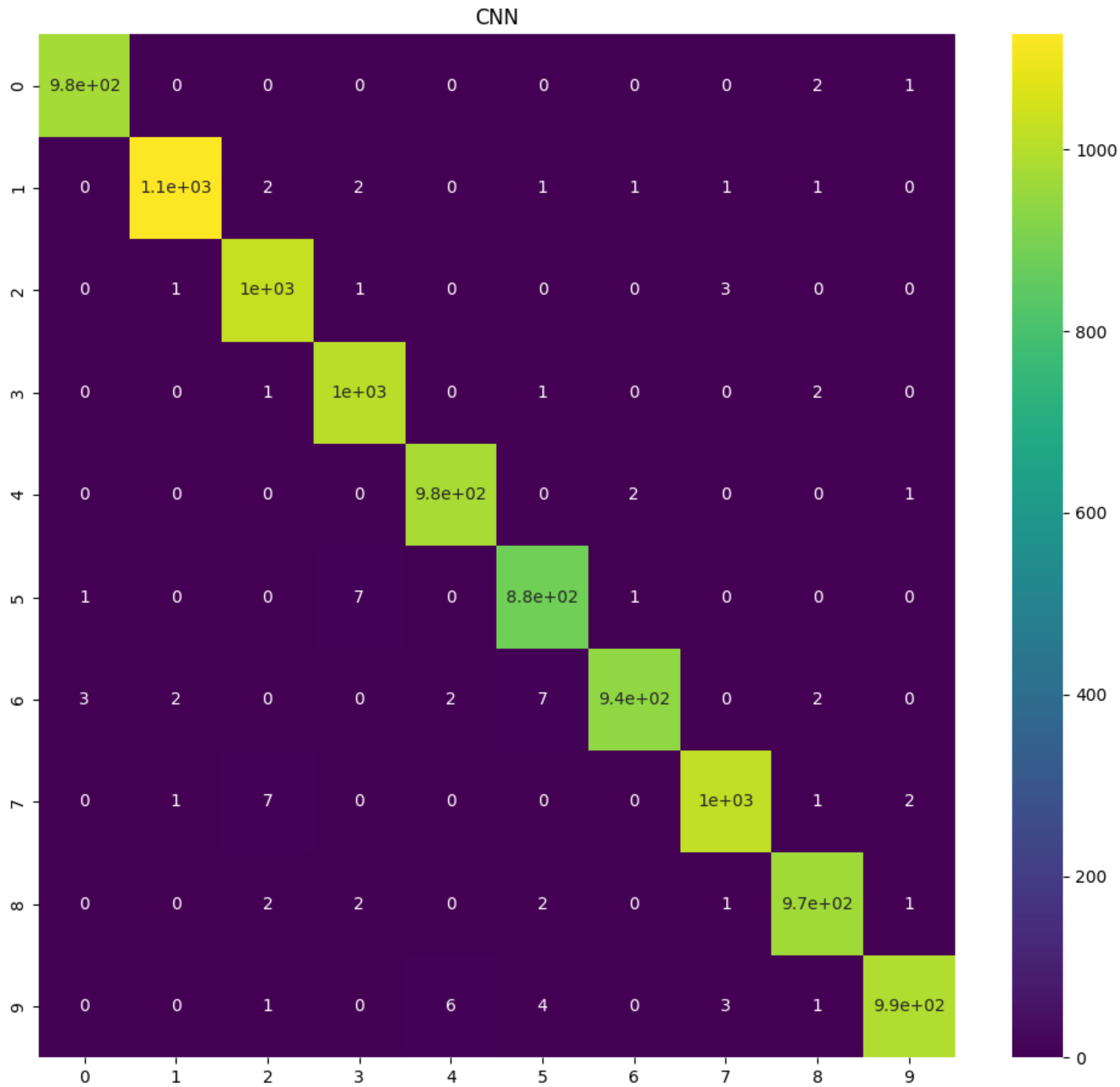
```
                      ******Log_Regression******
****************************************************************************************************
              precision    recall  f1-score   support

           0       0.95      0.98      0.97       980
           1       0.97      0.98      0.97      1135
           2       0.93      0.90      0.91      1032
           3       0.90      0.91      0.91      1010
           4       0.93      0.93      0.93       982
           5       0.90      0.87      0.89       892
           6       0.94      0.95      0.95       958
           7       0.93      0.93      0.93      1028
           8       0.87      0.89      0.88       974
           9       0.91      0.91      0.91      1009

    accuracy                           0.93     10000
   macro avg       0.92      0.92      0.92     10000
weighted avg       0.93      0.93      0.93     10000

****************************************************************************************************
                      ******MLP_1******
****************************************************************************************************
              precision    recall  f1-score   support

           0       0.98      0.98      0.98       980
           1       0.99      0.99      0.99      1135
           2       0.96      0.97      0.97      1032
           3       0.96      0.96      0.96      1010
           4       0.98      0.96      0.97       982
           5       0.96      0.96      0.96       892
           6       0.98      0.97      0.98       958
           7       0.97      0.96      0.97      1028
           8       0.94      0.97      0.95       974
           9       0.96      0.94      0.95      1009

    accuracy                           0.97     10000
   macro avg       0.97      0.97      0.97     10000
```

```
weighted avg       0.97      0.97      0.97     10000

*********************************************************************************************
                      ******MLP_2******
*********************************************************************************************
              precision    recall  f1-score   support

           0       0.98      0.98      0.98       980
           1       0.99      0.98      0.99      1135
           2       0.97      0.97      0.97      1032
           3       0.95      0.97      0.96      1010
           4       0.96      0.97      0.97       982
           5       0.96      0.97      0.96       892
           6       0.98      0.98      0.98       958
           7       0.98      0.95      0.97      1028
           8       0.96      0.96      0.96       974
           9       0.96      0.97      0.96      1009

    accuracy                           0.97     10000
   macro avg       0.97      0.97      0.97     10000
weighted avg       0.97      0.97      0.97     10000

*********************************************************************************************
                      ******CNN******
*********************************************************************************************
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       980
           1       1.00      0.99      0.99      1135
           2       0.99      1.00      0.99      1032
           3       0.99      1.00      0.99      1010
           4       0.99      1.00      0.99       982
           5       0.98      0.99      0.99       892
           6       1.00      0.98      0.99       958
           7       0.99      0.99      0.99      1028
           8       0.99      0.99      0.99       974
           9       0.99      0.99      0.99      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000

*********************************************************************************************
```


Log_Regression
9.6e+02
1.1e+03
9.3e+02
9.2e+02
9.1e+02
7.8e+02
9.1e+02
9.5e+02
8.7e+02
9.2e+02
1000
800
600
400
200
0


MLP_1


MLP_2
9.6e+02
0
1
2
1
0
5
1
3
2
0
1.1e+03
2
3
2
2
5
2
3
0
2
0
1e+03
11
2
0
4
8
8
0
0
1
3
9.8e+02
0
18
0
1
5
3
0
0
7
0
9.5e+02
0
4
1
0
18
5
0
0
6
1
8.7e+02
5
1
6
2
3
3
0
1
4
10
9.4e+02
0
1
1
1
2
12
6
5
0
0
9.8e+02
8
14
2
1
5
11
7
8
0
2
9.3e+02
4
2
0
0
7
14
2
0
2
8
9.7e+02
0 1 2 3 4 5 6 7 8 9
0
200
400
600
800
1000




```python
print(pd.DataFrame(data=accuracy,index=range(1)))
```

```
   Log_Regression   MLP_1   MLP_2     CNN
0          0.9255  0.9684  0.9698  0.9918
```

References:

Websites:

1- https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
2- https://scikit-learn.org/stable/modules/neural_networks_supervised.html 3- https://pytorch.org/tutorials/b
4- fastai 5- Elements of statistical learning and introduction to statistical learning

GPT prompts:

Asking to fix the dimensions of feature map of the CNN. Because initially I got the matrix multipication incomaptile dimensions error