Assignment 6

Name: Vaishnavi Gosavi **Roll No:** 71
Class: TY CSAI-A **Batch:** 2

Title: Build a Multiclass classifier using the CNN model. Use MNIST or any other suitable dataset.
a. Perform Data Pre-processing
b. Define Model and perform training
c. Evaluate Results using confusion matrix

Description:-

- **Convolutional Neural Network (CNN)**
  A Convolutional Neural Network (CNN) is a class of deep neural networks most commonly used in analyzing visual imagery. CNNs are inspired by the visual cortex of the human brain, which processes visual data hierarchically.

- **Architecture Components:**

1. **Input Layer**
   - Accepts images of shape (28x28x1) for grayscale MNIST data.

2. **Convolutional Layer**
   - Applies filters (kernels) to extract low-level features like edges, textures, and patterns.
   - Operation: Convolution between input and filter.
   - Output is called a feature map.

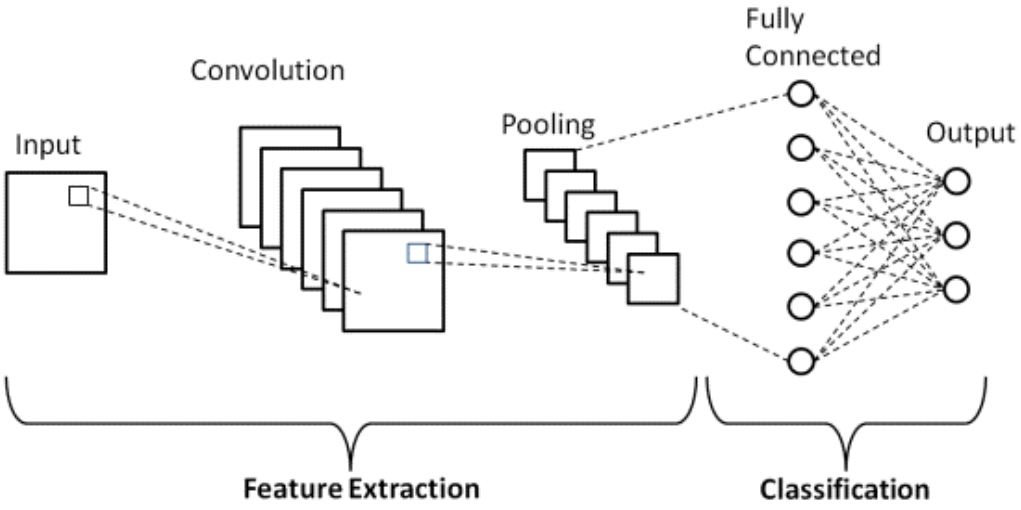
3. **Activation Layer (ReLU)**
   - Applies the ReLU (Rectified Linear Unit) function: f(x) = max(0, x)
   - Introduces non-linearity and prevents vanishing gradients.

4. **Pooling Layer (Max Pooling)**
   - Reduces spatial dimensions (width and height) of feature maps.
   - Helps in making the model more efficient and invariant to minor translations.
   - E.g., 2x2 MaxPooling reduces 28x28 → 14x14.

5. **Flatten Layer**
   - Converts the 2D feature maps into a 1D vector to feed into the Dense (fully connected) layer.

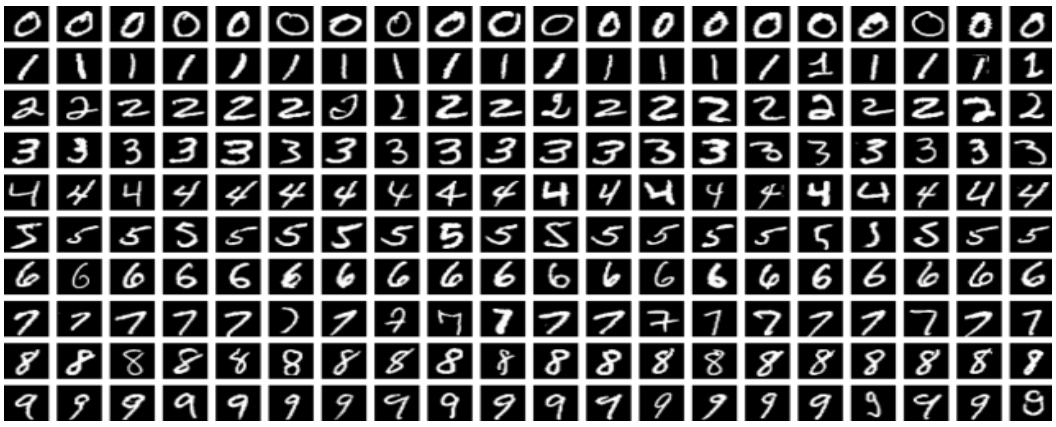
6. **Dense Layer**
   - Fully connected layer where each neuron is connected to all outputs from the previous layer.
   - Learns high-level global patterns.
   - Plot actual vs. predicted stock prices.

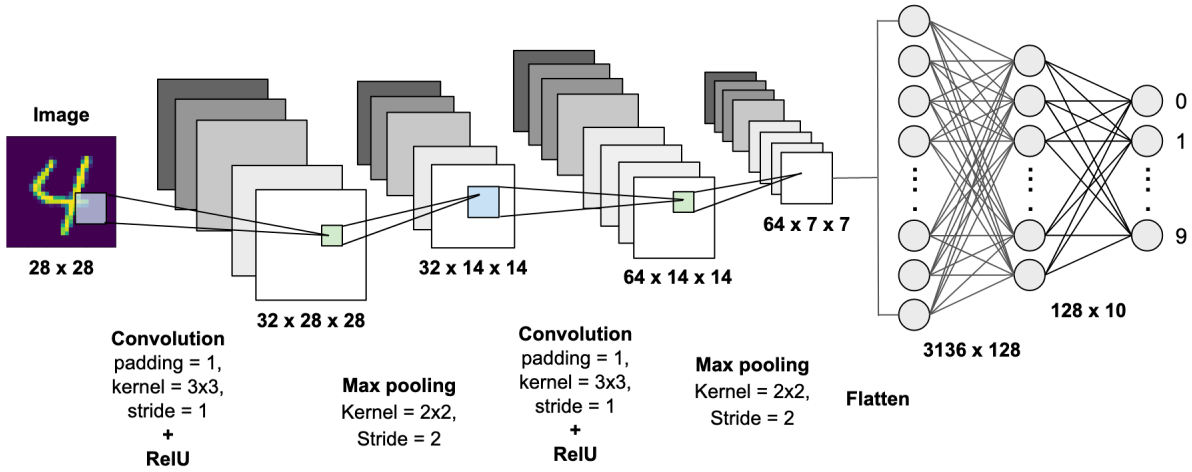
7. **Output Layer**
   - Uses Softmax activation function to output probabilities for each of the 10 classes.
   - Softmax ensures the outputs sum to 1 and represents confidence for each class.



- **Working Mechanism of CNN on MNIST**
  1. Input image (28x28) is fed to the CNN.
  2. First Conv layer extracts edge-level features (horizontal, vertical lines).
  3. Pooling layer reduces dimensions, keeps the most important features.
  4. Second Conv + Pooling layers extract more abstract features like loops, curves.
  5. Flattening makes the data suitable for the Dense layer.
  6. Dense layers learn complex relationships and finally classify the digit using Softmax.

- **What is MNIST?**
  - The MNIST (Modified National Institute of Standards and Technology) dataset contains 70,000 images of handwritten digits:
    - 60,000 for training
    - 10,000 for testing.
  - Each image is 28x28 grayscale and labeled from 0 to 9 (10 classes).





- **Algorithm**

1. Load MNIST dataset.
2. Normalize pixel values to range [0, 1].
3. Reshape dataset to (28, 28, 1) and convert labels to one-hot encoding.
4. Initialize a CNN model:
   a. Add Conv2D layer (filters=32, kernel=3x3)
   b. Add MaxPooling2D
   c. Add another Conv2D + MaxPooling2D
   d. Flatten the output
   e. Add Dense(128) layer with ReLU
   f. Add Dense(10) layer with Softmax
5. Compile the model using:
   - Optimizer: Adam
   - Loss: Categorical Crossentropy
6. Train the model with training data.
7. Predict on test data.
8. Evaluate performance using Confusion Matrix & Classification Report.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import numpy as np
import matplotlib.pyplot as plt
from torchvision import datasets, transforms
from sklearn.metrics import classification_report, confusion_matrix

# Load the MNIST dataset and apply transformations
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # Normalize to range [-1, 1]
])

train_data = datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
test_data = datasets.MNIST(root='./data', train=False, download=True,
transform=transform)

train_loader = DataLoader(train_data, batch_size=32, shuffle=True)
test_loader = DataLoader(test_data, batch_size=32, shuffle=False)
```

```
100%|██████████| 9.91M/9.91M [00:00<00:00, 50.9MB/s]
100%|██████████| 28.9k/28.9k [00:00<00:00, 1.67MB/s]
100%|██████████| 1.65M/1.65M [00:00<00:00, 14.0MB/s]
100%|██████████| 4.54k/4.54k [00:00<00:00, 7.62MB/s]
```

```
# Visualization: Random sample images from the training set
fig, axes = plt.subplots(1, 5, figsize=(10, 2))
for i in range(5):
    image, label = train_data[i]
    axes[i].imshow(image.squeeze(), cmap="gray")
    axes[i].axis('off')
plt.suptitle('Sample Images from the Training Set')
plt.show()
```


```
# CNN model definition
class CNNModel(nn.Module):
    def __init__(self):
```

```
        super(CNNModel, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(128 * 3 * 3, 256)
        self.fc2 = nn.Linear(256, 10)
        self.dropout = nn.Dropout(0.4)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = self.pool(torch.relu(self.conv3(x)))
        x = x.view(-1, 128 * 3 * 3)  # Flatten the tensor
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x


# Initialize the model, loss function, and optimizer
model = CNNModel()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
num_epochs = 20
train_losses, val_losses, train_accuracies, val_accuracies = [], [],
[], []

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    for images, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    train_loss = running_loss / len(train_loader)
    train_accuracy = 100 * correct / total
```

```
    # Validation
    model.eval()
    val_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item()

            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    val_loss = val_loss / len(test_loader)
    val_accuracy = 100 * correct / total

    train_losses.append(train_loss)
    val_losses.append(val_loss)
    train_accuracies.append(train_accuracy)
    val_accuracies.append(val_accuracy)

    print(f'Epoch {epoch+1}/{num_epochs}, Train Loss:
{train_loss:.4f}, Train Accuracy: {train_accuracy:.2f}%, Val Loss:
{val_loss:.4f}, Val Accuracy: {val_accuracy:.2f}%')

# Evaluate the model on test data
model.eval()
test_labels = []
test_predictions = []

with torch.no_grad():
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        test_labels.extend(labels.numpy())
        test_predictions.extend(predicted.numpy())
```

```
Epoch 1/20, Train Loss: 0.1540, Train Accuracy: 95.13%, Val Loss:
0.0382, Val Accuracy: 98.75%
Epoch 2/20, Train Loss: 0.0524, Train Accuracy: 98.46%, Val Loss:
0.0297, Val Accuracy: 99.03%
Epoch 3/20, Train Loss: 0.0374, Train Accuracy: 98.88%, Val Loss:
0.0229, Val Accuracy: 99.25%
Epoch 4/20, Train Loss: 0.0301, Train Accuracy: 99.07%, Val Loss:
0.0220, Val Accuracy: 99.35%
Epoch 5/20, Train Loss: 0.0260, Train Accuracy: 99.22%, Val Loss:
0.0273, Val Accuracy: 99.22%
```

```
Epoch 6/20, Train Loss: 0.0221, Train Accuracy: 99.33%, Val Loss:
0.0297, Val Accuracy: 99.24%
Epoch 7/20, Train Loss: 0.0181, Train Accuracy: 99.47%, Val Loss:
0.0306, Val Accuracy: 99.16%
Epoch 8/20, Train Loss: 0.0181, Train Accuracy: 99.45%, Val Loss:
0.0290, Val Accuracy: 99.26%
Epoch 9/20, Train Loss: 0.0142, Train Accuracy: 99.59%, Val Loss:
0.0269, Val Accuracy: 99.24%
Epoch 10/20, Train Loss: 0.0133, Train Accuracy: 99.58%, Val Loss:
0.0368, Val Accuracy: 99.14%
Epoch 11/20, Train Loss: 0.0132, Train Accuracy: 99.59%, Val Loss:
0.0310, Val Accuracy: 99.23%
Epoch 12/20, Train Loss: 0.0110, Train Accuracy: 99.68%, Val Loss:
0.0295, Val Accuracy: 99.29%
Epoch 13/20, Train Loss: 0.0124, Train Accuracy: 99.63%, Val Loss:
0.0438, Val Accuracy: 99.09%
Epoch 14/20, Train Loss: 0.0089, Train Accuracy: 99.72%, Val Loss:
0.0354, Val Accuracy: 99.27%
Epoch 15/20, Train Loss: 0.0118, Train Accuracy: 99.65%, Val Loss:
0.0369, Val Accuracy: 99.23%
Epoch 16/20, Train Loss: 0.0101, Train Accuracy: 99.73%, Val Loss:
0.0352, Val Accuracy: 99.16%
Epoch 17/20, Train Loss: 0.0071, Train Accuracy: 99.78%, Val Loss:
0.0356, Val Accuracy: 99.28%
Epoch 18/20, Train Loss: 0.0112, Train Accuracy: 99.69%, Val Loss:
0.0395, Val Accuracy: 99.17%
Epoch 19/20, Train Loss: 0.0085, Train Accuracy: 99.73%, Val Loss:
0.0543, Val Accuracy: 99.14%
Epoch 20/20, Train Loss: 0.0081, Train Accuracy: 99.80%, Val Loss:
0.0505, Val Accuracy: 99.17%
```

```python
# Classification report
print("\nClassification Report:\n", classification_report(test_labels,
test_predictions))
```

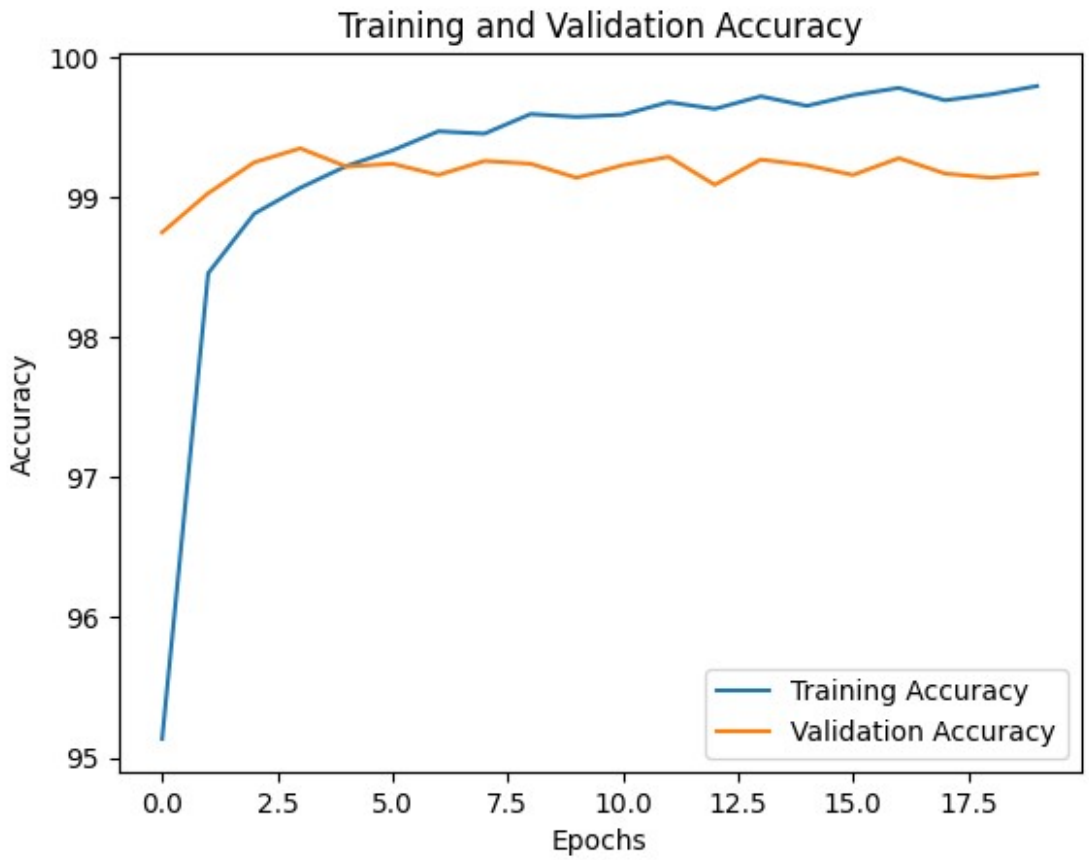
```
Classification Report:
               precision    recall  f1-score   support

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

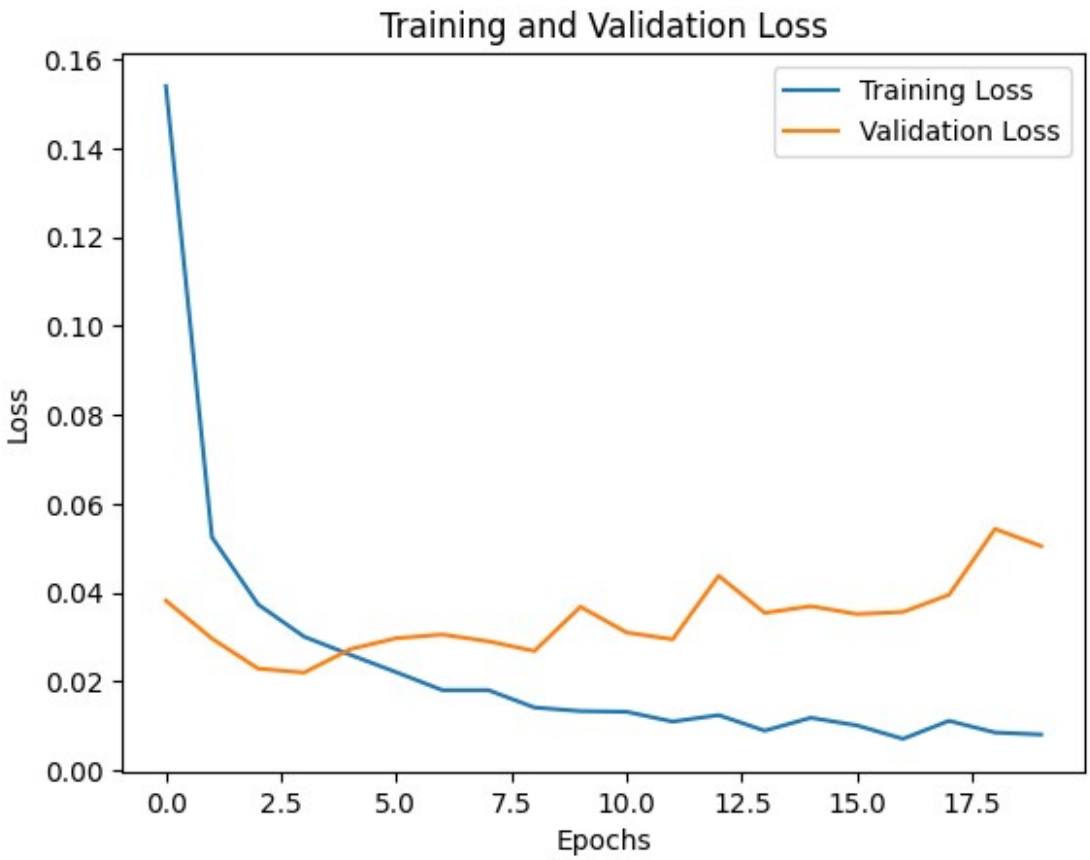
```
# Confusion matrix
conf_matrix = confusion_matrix(test_labels, test_predictions)
print("\nConfusion Matrix:\n", conf_matrix)
```

```
Confusion Matrix:
 [[ 975    0    2    0    0    0    0    2    1    0]
 [   0 1128    1    1    0    0    0    5    0    0]
 [   1    0 1025    2    0    0    0    3    1    0]
 [   0    0    2 1005    0    1    0    1    1    0]
 [   0    0    0    0  978    0    0    1    0    3]
 [   0    0    0    8    0  883    0    0    0    1]
 [   1    1    3    0    2    4  946    0    1    0]
 [   0    1    1    3    0    0    0 1021    0    2]
 [   1    0    1    3    0    0    0    0  968    1]
 [   0    0    0    3    9    3    0    6    0  988]]
```

```
# Plotting training and validation accuracy
plt.plot(train_accuracies, label='Training Accuracy')
plt.plot(val_accuracies, label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training and Validation Accuracy')
plt.show()
```




```
# Plotting training and validation loss
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss')
plt.show()
```


Training and Validation Loss
Training Loss
Validation Loss
Loss
Epochs
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5