**EXPERIMENT 7** Date:

**Problem Definition:** Implementation of Convolutional Neural Network

**Packages Used:** PyTorch, matplotlib

**Dataset Used:** MNIST dataset

**Theory:**

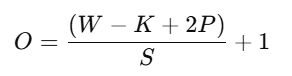
Convolutional Neural Networks (CNNs) are a class of deep neural networks particularly effective for image data processing. They are used widely in image recognition, object detection, and other computer vision tasks. CNNs consist of several types of layers, including convolutional, pooling, and fully connected layers, which allow them to capture spatial and hierarchical patterns in images.

**CNN Architecture**

1. **Convolutional Layer**:
   * **Purpose**: The convolutional layer applies a set of filters (or kernels) across the input image to generate feature maps. Each filter detects a specific feature or pattern, such as edges, lines, or colors.
   * **Operation**: Convolution involves sliding a small matrix (kernel) over the input image and performing element-wise multiplication, followed by summation, to produce a single pixel in the output feature map.
   * **Parameters**: Convolutional layers are defined by filter size, stride, padding, and the number of filters.
2. **Stride**:
   * **Definition**: Stride refers to the number of pixels the filter moves across the image at each step.
   * **Effects**:
     + A stride of 1 means the filter moves one pixel at a time, resulting in a highly detailed feature map.
     + A stride of 2 skips one pixel with each movement, reducing the feature map's size by roughly half. Larger strides decrease the output size more but may reduce detail.
   * **Example**: For an input image of 5x5 and a 3x3 filter with stride 1, the output is 3x3. If the stride is 2, the output becomes 2x2.
3. **Padding**:
   * **Definition**: Padding is the process of adding extra pixels (usually zeros) around the borders of the input image.
   * **Purpose**: Padding helps preserve the spatial dimensions of the input after convolution, allowing features near the image borders to be detected.
   * **Types**:
     + **Valid Padding**: No padding is applied, resulting in an output smaller than the input.
     + **Same Padding**: Padding is added so the output dimensions are the same as the input dimensions.
   * **Example**: If a 3x3 filter is applied on a 5x5 image with stride 1 and no padding, the output will be 3x3. With padding of 1 (one layer of zeros around the image), the output remains 5x5.
4. **Activation Function (ReLU)**:
   * **ReLU (Rectified Linear Unit)** is applied after each convolution to introduce non-linearity. It replaces all negative values with zero, making the network more efficient in handling non-linear relationships.
5. **Pooling Layer**:
   * **Purpose**: Pooling layers reduce the spatial size of the feature maps, making computation more efficient and reducing overfitting by emphasizing the most critical features.
   * **Types**:
     + **Max Pooling**: Takes the maximum value from each patch, capturing the most prominent feature.
     + **Average Pooling**: Averages the values in each patch, providing a smoother abstraction.
   * **Pooling Operation**: Typically applied with a 2x2 filter and a stride of 2, reducing the feature map size by half.
6. **Fully Connected Layer**:
   * This layer flattens the output from the final pooling/convolutional layers into a one-dimensional vector and connects it to the output layer, responsible for final classification. The fully connected layers integrate spatial and hierarchical features learned in the convolutional layers to make predictions.

**Formulas for Output Size Calculation**

1. **Convolutional Layer Output Size**:



where:

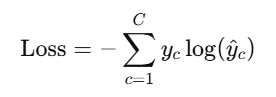
* + O: Output dimension.
  + W: Input dimension.
    1. K: Filter (kernel) size.
    2. P: Padding.
    3. S: Stride.

1. **Pooling Layer Output Size**: Pooling also uses the same formula as above but is typically performed with larger strides to downsample the feature maps.

**Training CNNs**

CNNs are trained similarly to other neural networks using backpropagation and gradient descent. However, they take advantage of weight sharing (using the same filters across the input) to reduce the number of parameters, making them more efficient than fully connected networks for high-dimensional data like images.

1. **Loss Function**:
   * **Cross-Entropy Loss** for classification tasks:



where yc​ is the true label, and y^c​ is the predicted probability for class c.

1. **Optimization**:
   * CNNs use optimizers like **SGD** (Stochastic Gradient Descent) or **Adam** to adjust weights based on the loss.
2. **Evaluation Metrics**:
   * For classification, common metrics include **accuracy**, **precision**, **recall**, and **F1-score**.

**Advantages of CNNs**

* **Spatial Invariance**: Pooling and convolutional layers make CNNs robust to small translations or distortions in images.
* **Parameter Efficiency**: Weight sharing in convolutions reduces the number of parameters, allowing CNNs to learn efficiently with less data.
* **Effective Feature Hierarchies**: CNNs learn spatial hierarchies, where earlier layers detect edges, and deeper layers learn more abstract features like objects.

**Implementation of Convolutional Neural Network in Pytorch:**

import torch

import torch.nn as nn

import torch.optim as optim

from torchvision import datasets, transforms

import matplotlib.pyplot as plt

transform = transforms.ToTensor()

mnist\_data = datasets.MNIST(root='./data', train=True, download=True, transform=transform)

data\_loader = torch.utils.data.DataLoader(dataset=mnist\_data, batch\_size=64, shuffle=True)

dataiter = iter(data\_loader)

images, labels = next(dataiter)

print(torch.min(images), torch.max(images))



#cnn

class Autoencoder(nn.Module):

  def \_\_init\_\_(self):

    # N, 1, 28, 28

    super().\_\_init\_\_(

    )

    self.encoder = nn.Sequential(

        nn.Conv2d(1, 16, 3, stride=2, padding=1),  #N, 16, 14, 14

        nn.ReLU(),

        nn.Conv2d(16, 32, 3, stride=2, padding=1), #N, 32, 7, 7

        nn.ReLU(),

        nn.Conv2d(32, 64, 7) #N, 64, 1, 1

    )

    #N, 64, 1, 1

    self.decoder = nn.Sequential(

        nn.ConvTranspose2d(64, 32, 7), #N, 32, 7, 7

        nn.ReLU(),

        nn.ConvTranspose2d(32, 16, 3, stride=2, padding=1, output\_padding=1), #N, 16, 14, 14

        nn.ReLU(),

        nn.ConvTranspose2d(16, 1, 3, stride=2, padding=1, output\_padding=1), #N, 1, 28, 28

        nn.Sigmoid()

    )

  def forward(self, x):

    encoded = self.encoder(x)

    decoded = self.decoder(encoded)

    return decoded

  #note: [-1, 1] -> nn.Tanh()

  #nn.MaxPool2d -> nn.MaxUnpool2d

model = Autoencoder()

criterion = nn.MSELoss()

optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight\_decay=1e-5)

num\_epochs = 10

outputs = []

for epoch in range(num\_epochs):

  for (img, \_) in data\_loader:

    recon = model(img)

    loss = criterion(recon, img)

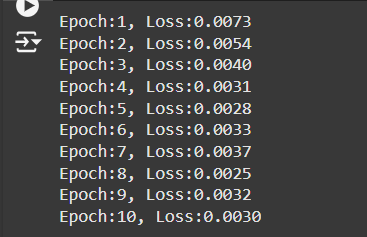
    optimizer.zero\_grad()

    loss.backward()

    optimizer.step()

  print(f'Epoch:{epoch+1}, Loss:{loss.item():.4f}')

  outputs.append((epoch, img, recon))



for k in range(0, num\_epochs, 4):

  plt.figure(figsize=(9, 2))

  plt.gray()

  imgs = outputs[k][1].detach().numpy()

  recon = outputs[k][2].detach().numpy()

  for i, item in enumerate(imgs):

    if i >= 9:

      break

    plt.subplot(2, 9, i+1)

    #item: 1, 28, 28

    plt.imshow(item[0])

  for i, item in enumerate(recon):

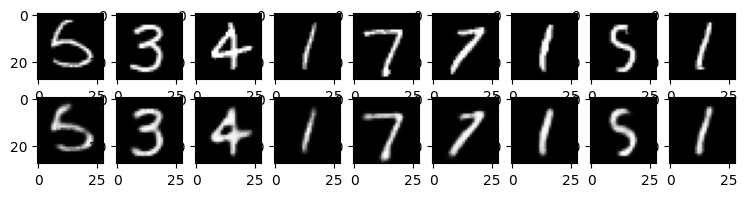
    if i >= 9:

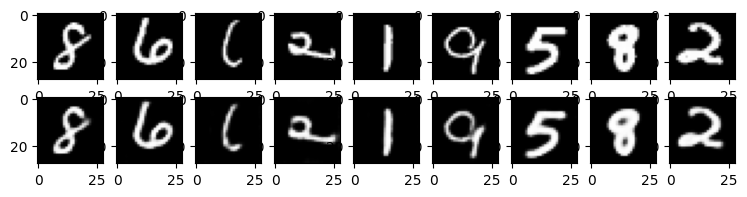
      break

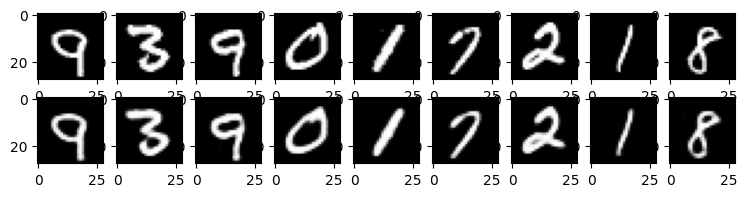
    plt.subplot(2, 9, 9+i+1) #row\_length + i + 1

    #item: 1, 28, 28

    plt.imshow(item[0])







**Conclusion:**

Convolutional Neural Network was studied and implemented successfully.