**EXPERIMENT 8** Date:

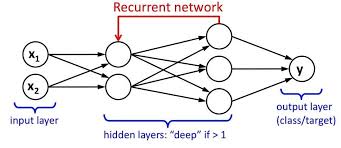
**Problem Definition:** Implementation of Recurrent Neural Network

**Packages Used:** PyTorch, matplotlib

**Dataset Used:** MNIST dataset

**Theory:**

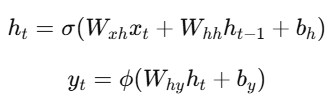
RNNs are a class of artificial neural networks where connections between nodes can create a directed cycle. This cyclical nature gives RNNs an internal memory, allowing them to "remember" information across sequences, which is essential for tasks involving sequential data. This memory makes them especially powerful for tasks where context and order are important, such as language modeling, speech recognition, and time-series prediction



**Sequential Processing and Memory**

In standard neural networks, the outputs are calculated independently, with no link between layers across time. RNNs, however, are designed to operate on sequences by retaining information about previous inputs, passing this information through time steps. At each time step t, the hidden state ht​ is influenced not only by the input at that time step xt​ but also by the hidden state from the previous time step ht−1h\_{t-1}ht−1​.

The key equations for a basic RNN with a single hidden layer are:



where:

* Wxh​: Weight matrix for the input xt​,
* Whh​: Weight matrix for the hidden state ht−1​,
* Why​: Weight matrix for producing the output yt​ from ht​,
* σ: Activation function (typically tanh or ReLU),
* ϕ: Output activation function (e.g., softmax for classification).

**Types of RNNs Based on Sequence Architecture**

RNNs come in various configurations, depending on the task requirements:

* **Many-to-One**: Maps an entire sequence to a single output, e.g., sentiment analysis, where a series of words is classified into positive or negative sentiment.
* **One-to-Many**: Maps a single input to a sequence output, e.g., image captioning, where an image is described in multiple words.
* **Many-to-Many**: Maps a sequence to another sequence, e.g., machine translation, where an input sentence in one language is translated into another language.

**Challenges with RNNs**

**(i) Vanishing and Exploding Gradients**

One of the primary challenges with training RNNs is the vanishing and exploding gradient problem. This issue occurs during backpropagation through time (BPTT), which is used to compute gradients for weight updates. When the gradients are too small, they diminish as they move backward, making it difficult for the network to learn from long-term dependencies. Conversely, large gradients can cause the network to diverge.

**(ii) Short-term Memory**

Basic RNNs are generally only able to remember information from recent time steps. This limitation makes them unsuitable for tasks that require capturing long-term dependencies across time.

To address these issues, more advanced RNN architectures, such as Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU), have been developed.

**Advanced RNN Architectures: LSTMs and GRUs**

Both LSTMs and GRUs are designed to handle long-term dependencies in sequential data better than traditional RNNs. They include gates to control the flow of information, enabling the model to decide which parts of the input sequence to keep or discard.

* **LSTMs (Long Short-Term Memory networks)**: LSTMs have three main gates — input, forget, and output gates — that control the information flow and help prevent vanishing gradients.
* **GRUs (Gated Recurrent Units)**: GRUs are similar to LSTMs but have fewer gates, making them faster to train and simpler to implement. They use update and reset gates to manage information flow.

**Stride and Padding**

Though typically associated with CNNs, strides and padding concepts can also play a role in RNN-based models when used with convolutional layers or dilated RNNs.

* **Stride**: Stride in an RNN context can refer to how the model steps through the input sequence, possibly skipping steps to handle longer sequences efficiently. This is useful in dilated RNNs, which increase the effective sequence length the model can handle by introducing gaps between the inputs.
* **Padding**: In the context of RNNs, padding is often used for handling sequences of varying lengths. Padding involves adding extra "dummy" time steps, usually filled with zeros, to ensure that all sequences in a batch have the same length.

**Applications of RNNs**

RNNs are versatile and have applications in various fields:

* **Natural Language Processing (NLP)**: Tasks such as language translation, text generation, and sentiment analysis.
* **Time-Series Prediction**: Applications in finance, weather forecasting, and sales forecasting.
* **Speech Recognition**: Translating audio signals into text.
* **Sequential Data Processing**: Video classification and processing, where each frame in a video sequence is processed.

**Implementation of a Recurrent Neural Network in Pytorch:**

# Cell 1: Import Libraries

import torch

import torch.nn as nn

import torch.optim as optim

import torch.nn.functional as F

from torch.utils.data import DataLoader

from torchvision import datasets

from torchvision import transforms

import matplotlib.pyplot as plt

# Set device

device = 'cuda' if torch.cuda.is\_available() else 'cpu'

# Cell 2: Define the RNN model

class RNN(nn.Module):

    def \_\_init\_\_(self, input\_size, hidden\_size, num\_layer, num\_classes=10):

        super(RNN, self).\_\_init\_\_()

        self.hidden\_size = hidden\_size

        self.num\_layer = num\_layer

        self.rnn = nn.RNN(input\_size, hidden\_size, num\_layer, batch\_first=True)

        self.fc = nn.Linear(hidden\_size\*seq\_len, num\_classes)

    def forward(self, x):

        h0 = torch.zeros(self.num\_layer, x.size(0), self.hidden\_size).to(device)

        out, \_ = self.rnn(x, h0)

        out = out.reshape(out.shape[0], -1)

        out = self.fc(out)

        return out

# Cell 3: Define hyperparameters and load data

# Hyperparameters

input\_size = 28

seq\_len = 28

num\_layer = 2

hidden\_size = 256

num\_classes = 10

learning\_rate = 0.001

batch\_size = 64

num\_epochs = 10  # Adjusted to 10 epochs

# Load the dataset

train\_dataset = datasets.MNIST(root='Dataset/', train=True, transform=transforms.ToTensor(), download=True)

train\_loader = DataLoader(dataset=train\_dataset, batch\_size=batch\_size, shuffle=True)

test\_dataset = datasets.MNIST(root='Dataset/', train=False, transform=transforms.ToTensor(), download=True)

test\_loader = DataLoader(dataset=test\_dataset, batch\_size=batch\_size, shuffle=True)

# Cell 4: Initialize the network, define loss function and optimizer

model = RNN(input\_size=input\_size, num\_layer=num\_layer, hidden\_size=hidden\_size).to(device)

loss\_fn = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=learning\_rate)

# Cell 5: Define function to display images for an epoch

def display\_images(loader, model, epoch):

    model.eval()

    data, \_ = next(iter(loader))

    data = data[:5].squeeze(1).to(device)  # Select first 5 images for visualization

    with torch.no\_grad():

        preds = model(data)

    images = data.cpu()

    plt.figure(figsize=(10, 2))

    for i in range(5):

        plt.subplot(1, 5, i + 1)

        plt.imshow(images[i].numpy(), cmap='gray')

        plt.title(f"Pred: {preds[i].argmax().item()}")

        plt.axis('off')

    plt.suptitle(f"Epoch {epoch + 1}")

    plt.show()

    model.train()

# Cell 6: Train the model and display images for each epoch

for epoch in range(num\_epochs):

    print(f"Epoch: {epoch + 1}")

    for batch\_idx, (data, targets) in enumerate(train\_loader):

        data = data.squeeze(1).to(device)

        targets = targets.to(device)

        # Forward pass

        scores = model(data)

        loss = loss\_fn(scores, targets)

        # Backward pass and optimization

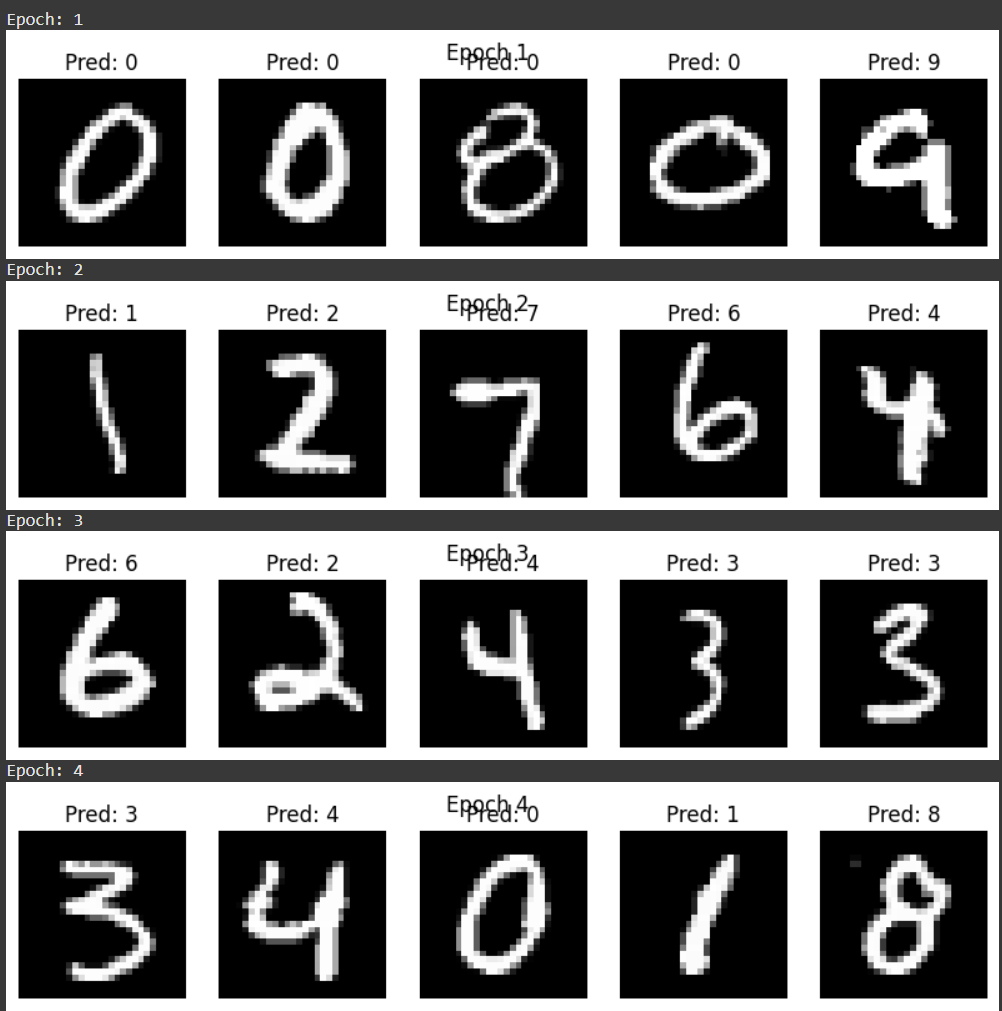
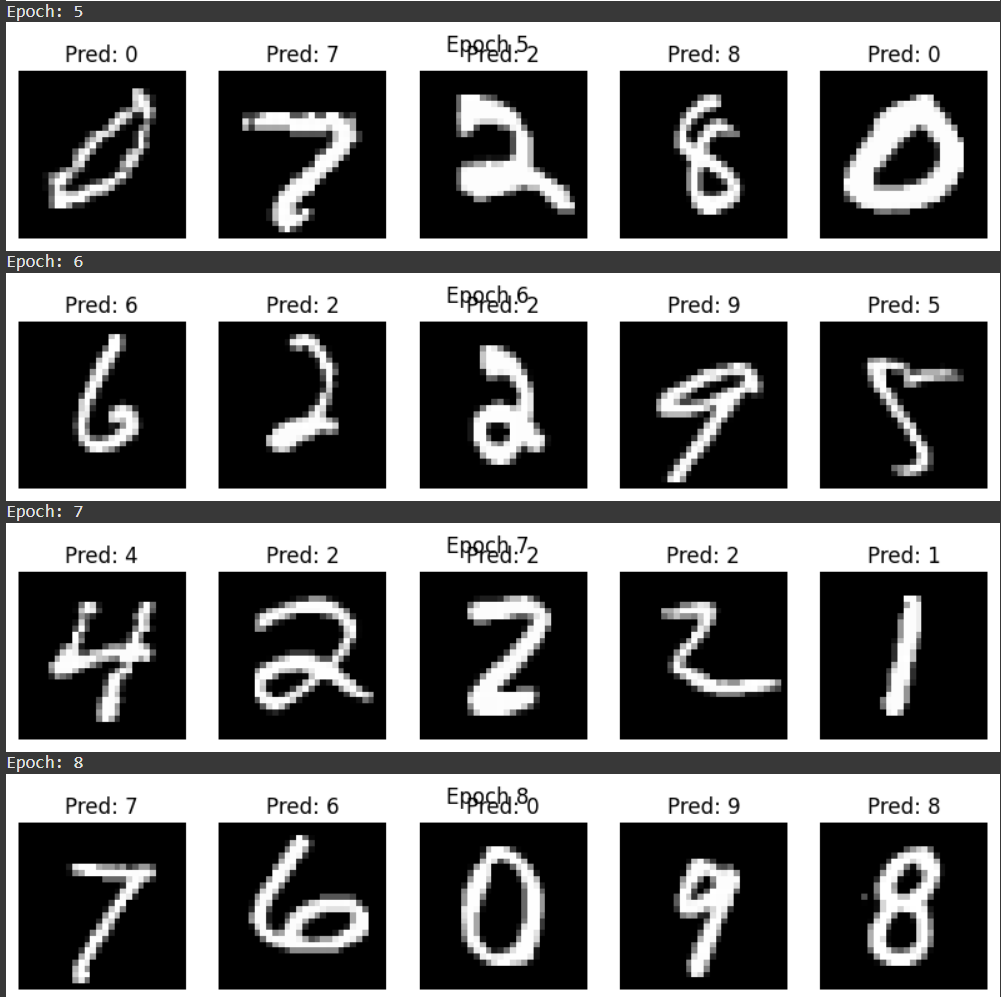
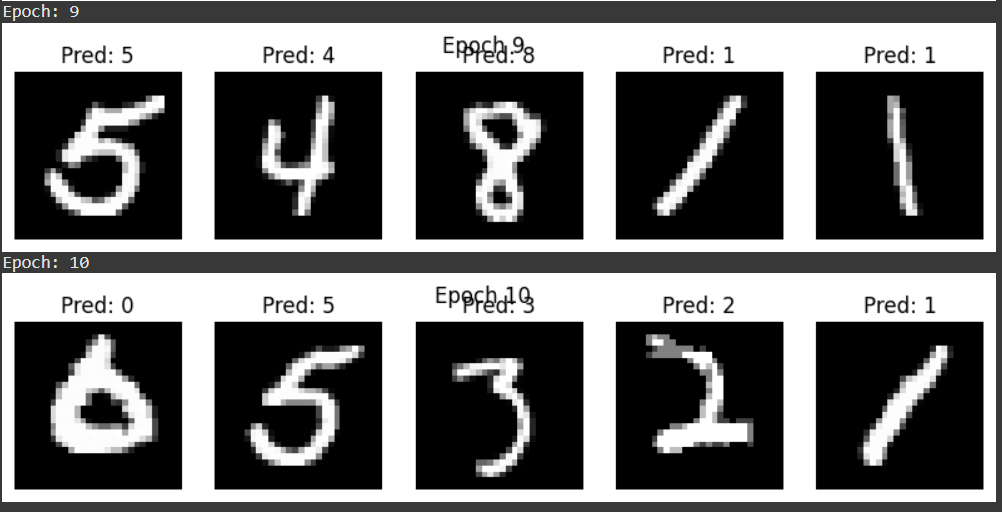
        optimizer.zero\_grad()

        loss.backward()

        optimizer.step()

    # Display images and predictions for the epoch

    display\_images(test\_loader, model, epoch)

# Cell 7: Define a function to check accuracy on the train and test sets

def check\_accuracy(loader, model):

    if loader.dataset.train:

        print("Checking accuracy on train dataset")

    else:

        print("Checking accuracy on test dataset")

    num\_correct = 0

    num\_samples = 0

    model.eval()

    with torch.no\_grad():

        for x, y in loader:

            x = x.to(device).squeeze(1)

            y = y.to(device)

            scores = model(x)

            \_, pred = scores.max(1)

            num\_correct += (pred == y).sum().item()

            num\_samples += pred.size(0)

        accuracy = 100.0 \* num\_correct / num\_samples

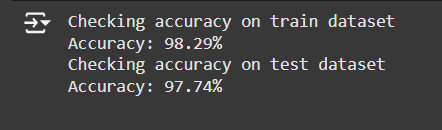
        print(f"Accuracy: {accuracy:.2f}%")

    model.train()

# Cell 8: Run accuracy checks

check\_accuracy(train\_loader, model)

check\_accuracy(test\_loader, model)



**Conclusion:**

Recurrent Neural Network was studied and implemented successfully.