Practical – 5

Aim: To perform the Convolutional Neural Networks which highlight the role of ReLU activation function.

Solution:

```
import torch
import torch.nn as nn import
torch.optim as optim import
torchvision
import torchvision.transforms as transforms # Define a
simple CNN model
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1) self.relu = nn.ReLU()
        # ReLU activation
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2) self.conv2 = nn.Conv2d(32,
        64, kernel_size=3, padding=1) self.fc1 = nn.Linear(64 * 8 * 8, 128)
        self.fc2 = nn.Linear(128, 10)  # For 10 classes def forward(self, x):
        x = self.pool(self.relu(self.conv1(x))) x =
        self.pool(self.relu(self.conv2(x)))
        x = x.view(-1, 64 * 8 * 8)  # Flatten the output
        x = self.relu(self.fc1(x))          # Apply ReLU activation x =
        self.fc2(x)
        return x
# Create the model and print architecture model =
SimpleCNN()
print(model)
```

Output:

```
SimpleCNN(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (relu): ReLU()
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=4096, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=10, bias=True)
)

Process finished with exit code 0
```

Practical No – 6

Aim: To apply the Recurrent Neural Network for Time Series Prediction.

Solution:

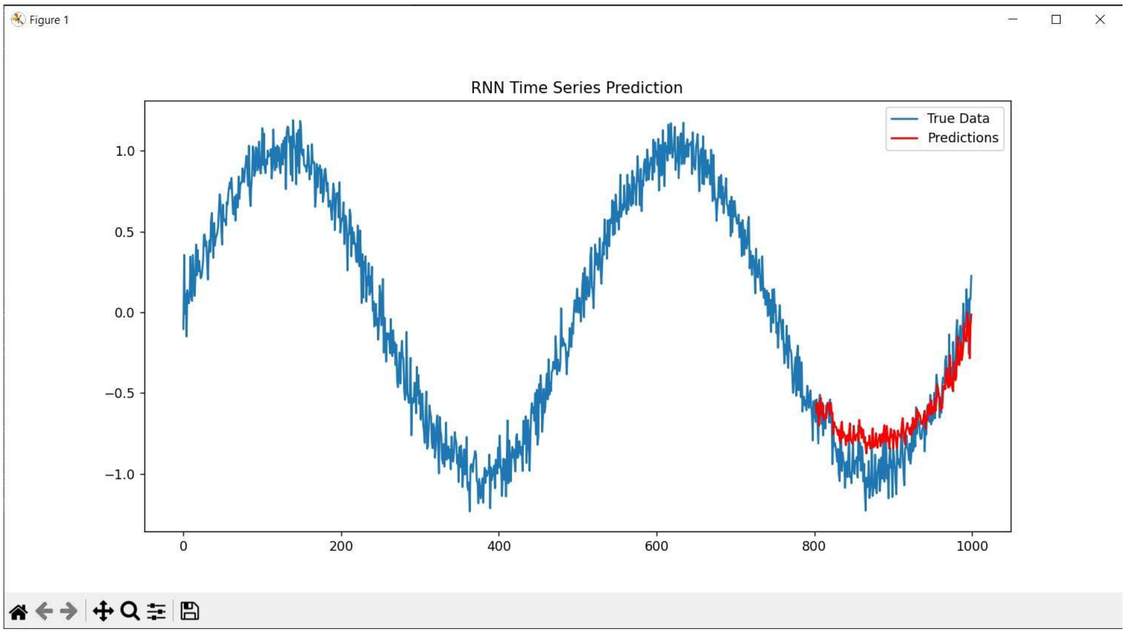
```
import numpy as np
import matplotlib.pyplot as plt


# Generate synthetic sine wave time series data
def generate_sine_wave_data(seq_length=1000, steps=100):
   # Create sine wave data
   x = np.linspace(0, 4 * np.pi, seq_length)
   y = np.sin(x) + 0.1 * np.random.randn(seq_length)  # Add some noise
   return y


# Prepare the dataset for supervised learning (X, y) with time steps
def prepare_data(data, time_steps=10):
   X, y = [], []
   for i in range(len(data) - time_steps):
       X.append(data[i:i + time_steps])
       y.append(data[i + time_steps])
   return np.array(X), np.array(y)


# Define the RNN architecture
class SimpleRNN:
   def __init__(self, input_size, hidden_size, output_size,
learning_rate=0.001):
       self.input_size = input_size
       self.hidden_size = hidden_size
       self.output_size = output_size
       self.learning_rate = learning_rate

       # Initialize the weights for the RNN
       self.Wxh = np.random.randn(hidden_size, input_size) * 0.01  # Input to
hidden
       self.Whh = np.random.randn(hidden_size, hidden_size) * 0.01  # Hidden to
hidden
       self.Why = np.random.randn(output_size, hidden_size) * 0.01  # Hidden to
output

       # Bias terms
       self.bh = np.zeros((hidden_size, 1))  # Hidden bias
       self.by = np.zeros((output_size, 1))  # Output bias

   def forward(self, X):
       h = np.zeros((self.hidden_size, 1))  # Initial hidden state
       self.hs = []  # Stores hidden states for each time step
```

```
        self.outputs = []  # Stores outputs at each time step

        for t in range(X.shape[0]):
            # RNN cell: Update hidden state
            h = np.tanh(np.dot(self.Wxh, X[t].reshape(-1, 1)) + np.dot(self.Whh,
h) + self.bh)
            self.hs.append(h)
            y = np.dot(self.Why, h) + self.by  # Output
            self.outputs.append(y)

        # Return the final output after processing the entire sequence
        return self.outputs[-1]

    def backward(self, X, y):
        # Initialize gradients
        dWxh = np.zeros_like(self.Wxh)
        dWhh = np.zeros_like(self.Whh)
        dWhy = np.zeros_like(self.Why)
        dbh = np.zeros_like(self.bh)
        dby = np.zeros_like(self.by)
        dh = np.zeros_like(self.hs[-1])

        # Backpropagate through time (BPTT)
        for t in reversed(range(X.shape[0])):
            dy = self.outputs[t] - y.reshape(-1, 1)  # Gradient of loss with
respect to output
            dWhy += np.dot(dy, self.hs[t].T)
            dby += dy

            # Backpropagate into the hidden state
            dh += np.dot(self.Why.T, dy)
            dhraw = (1 - self.hs[t] ** 2) * dh  # Derivative of tanh

            dbh += dhraw
            dWxh += np.dot(dhraw, X[t].reshape(1, -1))
            dWhh += np.dot(dhraw, self.hs[t - 1].T) if t > 0 else np.dot(dhraw,
np.zeros_like(self.hs[t - 1]).T)

        # Update weights with gradients
        self.Wxh -= self.learning_rate * dWxh
        self.Whh -= self.learning_rate * dWhh
        self.Why -= self.learning_rate * dWhy
        self.bh -= self.learning_rate * dbh
        self.by -= self.learning_rate * dby

    def train(self, X_train, y_train, epochs=100):
        for epoch in range(epochs):
            loss = 0
            for i in range(X_train.shape[0]):
```

```
                X_seq = X_train[i]
                y_true = y_train[i]

                # Forward pass
                self.outputs = []
                self.hs = []
                y_pred = self.forward(X_seq)

                # Compute the loss (Mean Squared Error)
                loss += np.sum((y_pred - y_true) ** 2)

                # Backward pass (Gradient computation)
                self.backward(X_seq, y_train[i])

            if epoch % 10 == 0:
                print(f"Epoch {epoch}, Loss: {loss}")


# Set hyperparameters
time_steps = 10
hidden_size = 50
output_size = 1
learning_rate = 0.001
epochs = 100

# Generate data
data = generate_sine_wave_data()
X, y = prepare_data(data, time_steps)

# Train-test split
train_size = int(0.8 * len(X))
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Create and train the RNN model
model = SimpleRNN(input_size=1, hidden_size=hidden_size,
output_size=output_size, learning_rate=learning_rate)
model.train(X_train, y_train, epochs=epochs)

# Test the model on unseen data
predictions = []
for i in range(len(X_test)):
   pred = model.forward(X_test[i])
   predictions.append(pred.flatten())  # Flatten to make it 1D

# Plot results
plt.figure(figsize=(12, 6))
plt.plot(range(len(data)), data, label='True Data')
```

```
plt.plot(range(train_size + time_steps, len(data)), predictions,
label='Predictions', color='red')
plt.legend()
plt.title('RNN Time Series Prediction')
plt.show()
```

Output

1) 

2)
```
Epoch 0, Loss: 90.44025019810846
Epoch 10, Loss: 13.952511487099246
Epoch 20, Loss: 13.606351398502518
Epoch 30, Loss: 13.040809164630346
Epoch 40, Loss: 12.801199074203758
Epoch 50, Loss: 12.865736070818008
Epoch 60, Loss: 12.905117870473196
Epoch 70, Loss: 12.890310068026817
Epoch 80, Loss: 12.86053872772447
Epoch 90, Loss: 12.83236227255182

Process finished with exit code 0
```

Practical No – 7

Aim: To implement an LSTM and Bi-directional LSTM for deep learning applications.

Solution:

```
import torch
import torch.nn as nn
import torch.optim as optim


# Define the LSTM model (Unidirectional)
class LSTMModel(nn.Module):
   def __init__(self, input_size, hidden_size, output_size):
       super(LSTMModel, self).__init__()

       self.hidden_size = hidden_size

       # LSTM layer
       self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)

       # Fully connected layer to output predictions
       self.fc = nn.Linear(hidden_size, output_size)

   def forward(self, x):
       # Initialize hidden state and cell state
       h0 = torch.zeros(1, x.size(0), self.hidden_size).to(x.device)
       c0 = torch.zeros(1, x.size(0), self.hidden_size).to(x.device)

       # LSTM layer
       lstm_out, _ = self.lstm(x, (h0, c0))

       # Take the last hidden state
       out = lstm_out[:, -1, :]

       # Output layer
       out = self.fc(out)

       return out


# Define the Bi-directional LSTM model
class BiLSTMModel(nn.Module):
   def __init__(self, input_size, hidden_size, output_size):
       super(BiLSTMModel, self).__init__()

       self.hidden_size = hidden_size

       # Bi-directional LSTM layer
       self.bilstm = nn.LSTM(input_size, hidden_size, batch_first=True,
bidirectional=True)
```

```
        # Fully connected layer to output predictions
        self.fc = nn.Linear(2 * hidden_size, output_size)

    def forward(self, x):
        # Initialize hidden state and cell state
        h0 = torch.zeros(2, x.size(0), self.hidden_size).to(x.device)  # 2 for
bidirectional
        c0 = torch.zeros(2, x.size(0), self.hidden_size).to(x.device)  # 2 for
bidirectional

        # Bi-directional LSTM layer
        lstm_out, _ = self.bilstm(x, (h0, c0))

        # Take the last hidden state (for the last time step)
        out = lstm_out[:, -1, :]

        # Output layer
        out = self.fc(out)

        return out


# Create a simple dataset (for example, sequence classification)
def create_data(batch_size=16, seq_length=10, input_size=5):
    X = torch.randn(batch_size, seq_length, input_size)  # Random sequences
    y = torch.randint(0, 2, (batch_size, 1))  # Binary classification labels
    return X, y


# Hyperparameters
input_size = 5  # Number of features in input data
hidden_size = 32  # LSTM hidden state size
output_size = 1  # Output size (binary classification in this case)
batch_size = 16  # Batch size
seq_length = 10  # Length of input sequences
epochs = 10  # Number of training epochs
learning_rate = 0.001  # Learning rate

# Create models
lstm_model = LSTMModel(input_size, hidden_size, output_size).to(
    torch.device("cuda" if torch.cuda.is_available() else "cpu"))
bilstm_model = BiLSTMModel(input_size, hidden_size, output_size).to(
    torch.device("cuda" if torch.cuda.is_available() else "cpu"))

# Loss function and optimizer
criterion = nn.BCEWithLogitsLoss()  # For binary classification
optimizer_lstm = optim.Adam(lstm_model.parameters(), lr=learning_rate)
optimizer_bilstm = optim.Adam(bilstm_model.parameters(), lr=learning_rate)
```

```
# Train the models
for epoch in range(epochs):
   # Create random data (for this example, we're not using a real dataset)
   X, y = create_data(batch_size, seq_length, input_size)
   X, y = X.to(torch.device("cuda" if torch.cuda.is_available() else "cpu")),
y.to(
       torch.device("cuda" if torch.cuda.is_available() else "cpu"))

   # LSTM Model Training
   optimizer_lstm.zero_grad()
   output_lstm = lstm_model(X)
   # Squeeze the target tensor to ensure it has the same shape as the output
   loss_lstm = criterion(output_lstm.squeeze(), y.squeeze().float())
   loss_lstm.backward()
   optimizer_lstm.step()

   # BiLSTM Model Training
   optimizer_bilstm.zero_grad()
   output_bilstm = bilstm_model(X)
   # Squeeze the target tensor to ensure it has the same shape as the output
   loss_bilstm = criterion(output_bilstm.squeeze(), y.squeeze().float())
   loss_bilstm.backward()
   optimizer_bilstm.step()

   # Print the loss for this epoch
   print(f'Epoch [{epoch + 1}/{epochs}], LSTM Loss: {loss_lstm.item():.4f},
BiLSTM Loss: {loss_bilstm.item():.4f}')

# After training, you could use the models for prediction or evaluation.
```

Output

```
Epoch [1/10], LSTM Loss: 0.7368, BiLSTM Loss: 0.7057
Epoch [2/10], LSTM Loss: 0.7039, BiLSTM Loss: 0.7005
Epoch [3/10], LSTM Loss: 0.6832, BiLSTM Loss: 0.6907
Epoch [4/10], LSTM Loss: 0.6832, BiLSTM Loss: 0.6906
Epoch [5/10], LSTM Loss: 0.6560, BiLSTM Loss: 0.6877
Epoch [6/10], LSTM Loss: 0.6955, BiLSTM Loss: 0.6907
Epoch [7/10], LSTM Loss: 0.6980, BiLSTM Loss: 0.6975
Epoch [8/10], LSTM Loss: 0.7173, BiLSTM Loss: 0.6983
Epoch [9/10], LSTM Loss: 0.6872, BiLSTM Loss: 0.6884
Epoch [10/10], LSTM Loss: 0.7079, BiLSTM Loss: 0.6974


Process finished with exit code 0
```

Practical No – 8

Aim: To apply the use of Auto encoder for feature optimization.

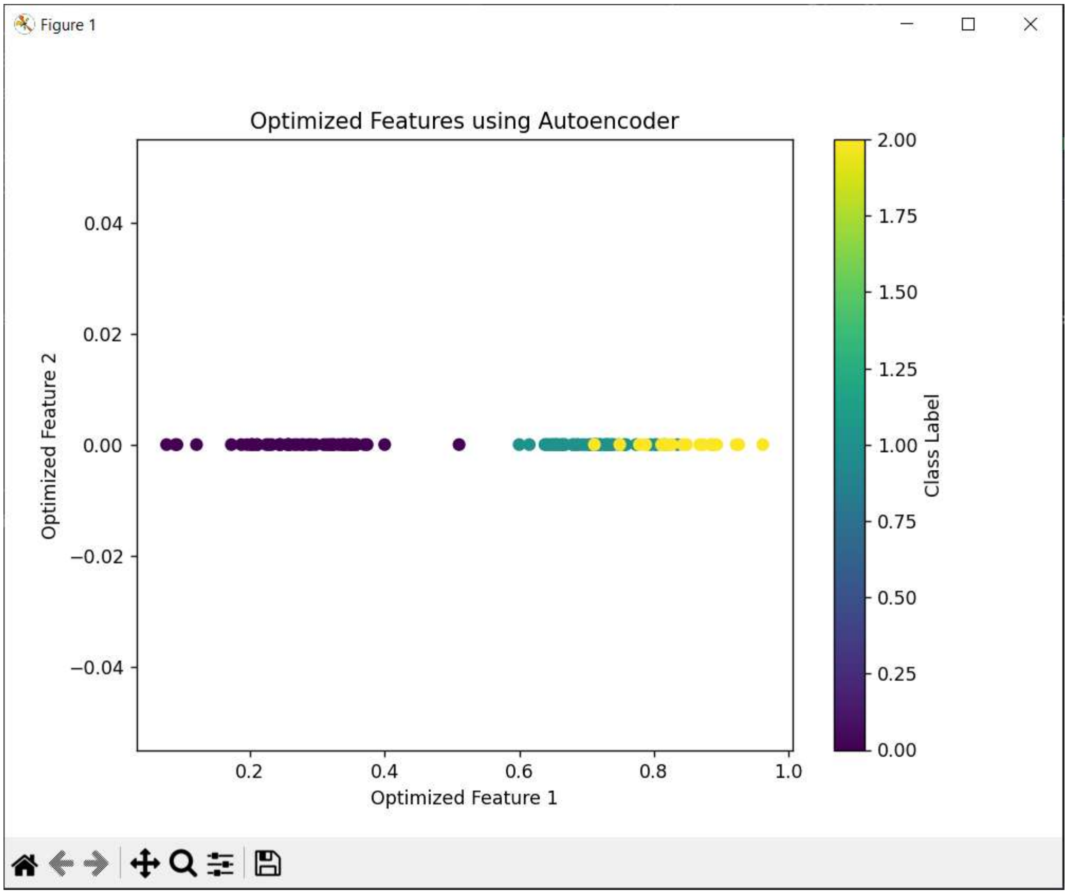
Solution:

```
#Import necessary libraries import
numpy as np
import seaborn as sns import torch
import torch.nn as nn import
torch.optim as optim
import matplotlib.pyplot as plt
from sklearn.preprocessing import minmax_scale  # Used for scaling from sklearn.preprocessing
import LabelEncoder  # For label encoding

# 1. Load Iris dataset using Seaborn data =
sns.load_dataset('iris')
X = data.drop('species', axis=1).values y =
data['species'].values

# 2. Standardize the features using min-max scaling X_scaled =
minmax_scale(X)

# 3. Train-test split (manually using numpy) train_size = int(0.8 * X.shape[0])
X_train, X_test = X_scaled[:train_size], X_scaled[train_size:] y_train, y_test = y[:train_size],
y[train_size:]

# 4. Encode the class labels as numeric values label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train) y_test_encoded =
label_encoder.transform(y_test)


# 5. Define the Autoencoder model in PyTorch class
Autoencoder(nn.Module):
    def __init__(self, input_dim, encoding_dim): super(Autoencoder, self).__
        init__() self.encoder = nn.Sequential(
            nn.Linear(input_dim, encoding_dim), nn.ReLU(True)
        )
        self.decoder = nn.Sequential( nn.Linear(encoding_dim, input_dim),
            nn.Sigmoid()  # Use sigmoid for output layer
        )

    def forward(self, x): encoded =
        self.encoder(x)
        decoded = self.decoder(encoded)
```

```
        return decoded


# Convert data to PyTorch tensors
X_train_tensor=torch.tensor(X_train, dtype=torch.float32) X_test_tensor = torch.tensor(X_test,
dtype=torch.float32)

# 6. Initialize the model, loss function, and optimizer input_dim = X_train.shape[1]
encoding_dim = 2
model=Autoencoder(input_dim=input_dim, encoding_dim=encoding_dim) criterion =
nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# 7. Training the Autoencoder epochs = 50
batch_size = 10

for epoch in range(epochs):
    model.train()
    for i in range(0, len(X_train_tensor), batch_size): batch_data = X_train_tensor[i:i +
        batch_size] output = model(batch_data)
        loss = criterion(output, batch_data)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch + 1}/{epochs}], Loss: {loss.item():.4f}')

# 8. Extract the optimized features using the encoder model.eval()
with torch.no_grad():
    X_train_encoded = model.encoder(X_train_tensor).numpy() X_test_encoded =
    model.encoder(X_test_tensor).numpy()

# 9. Output the original and optimized feature shapes print("Original feature shape:",
X_train.shape) print("Optimized feature shape:", X_train_encoded.shape)

# 10. Visualize the optimized features in 2D space plt.figure(figsize=(8, 6))
plt.scatter(X_train_encoded[:, 0], X_train_encoded[:, 1], c=y_train_encoded, cmap='viridis')
plt.xlabel('Optimized Feature 1')
plt.ylabel('Optimized Feature 2') plt.title('Optimized Features using
Autoencoder')
```

```
plt.colorbar(label='Class Label')
plt.show()
```

Output



1)

```
Epoch [10/50], Loss: 0.0781
Epoch [20/50], Loss: 0.0744
Epoch [30/50], Loss: 0.0744
Epoch [40/50], Loss: 0.0765
Epoch [50/50], Loss: 0.0788
Original feature shape: (120, 4)
Optimized feature shape: (120, 2)

Process finished with exit code 0
```

2)

Practical No – 9

Aim: To apply different Deep Learning Models for Natural Language Processing.

Solution:

```
import torch
import torch.nn as nn import
torch.optim as optim import re
from sklearn.model_selection import train_test_split from
sklearn.preprocessing import LabelEncoder
from collections import Counter
from torch.utils.data import DataLoader, TensorDataset

# 1. Load and Preprocess Data def
load_imdb_data():
    # Sample data (for demonstration purposes) data = [
        ("This movie was awesome", "positive"),
        ("I hated this movie, it was terrible", "negative"), ("I really enjoyed this film,
        amazing!", "positive"), ("Not good, not bad, just okay", "neutral"),
        ("Worst movie I have ever seen", "negative"), ("An absolute
        masterpiece", "positive")
    ]
    return data

# 2. Preprocessing: Clean text and create vocabulary def preprocess_text(text):
    text = text.lower()  # Convert to lowercase
    text = re.sub(r"[^a-zA-Z\s]", "", text)  # Remove punctuation and non-alphabetic characters
    return text

def simple_tokenizer(text):
    # Tokenize by splitting on whitespace after preprocessing text = preprocess_text(text)
    return text.split()  # Split by whitespace

def build_vocab(data, max_vocab_size=1000): all_words = []
    for sentence, _ in data:
        words = simple_tokenizer(sentence) all_words.extend(words)

    # Count frequency of each word word_count =
    Counter(all_words) # Limit vocabulary size
    vocab = {word: idx+1 for idx, (word, _) in enumerate(word_count.most_common(max_vocab_size))}
```

```
    vocab['<PAD>'] = 0  # Padding token return vocab

def text_to_sequence(text, vocab, max_len=50): words =
    simple_tokenizer(text)
    # Convert words to word indices based on vocabulary
    sequence = [vocab.get(word, vocab.get('<PAD>')) for word in words] # Pad or truncate sequences
    to max_len
    return sequence[:max_len] + [vocab.get('<PAD>')] * (max_len - len(sequence))

def encode_labels(labels):
    # Map the labels 'positive' and 'negative' to 1 and 0 return [1 if label == "positive" else 0
    for label in labels]

# 3. Load Data, Build Vocabulary, and Encode Data data = load_imdb_data()
sentences, labels = zip(*data) vocab =
build_vocab(data)
max_len = 50  # Maximum length of each sentence
X = [text_to_sequence(sentence, vocab, max_len) for sentence in sentences] y = encode_labels(labels)  # Make
sure labels are either 0 or 1

# 4. Convert to PyTorch Tensors and Create DataLoader
X = torch.tensor(X, dtype=torch.long) y = torch.tensor(y,
dtype=torch.float)

# Split dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

train_dataset = TensorDataset(X_train, y_train) test_dataset =
TensorDataset(X_test, y_test)

train_loader = DataLoader(train_dataset, batch_size=2, shuffle=True) test_loader=
DataLoader(test_dataset, batch_size=2, shuffle=False)

# 5. Model Definitions class

RNNModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim): super(RNNModel, self).__
        init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim) self.rnn =
        nn.RNN(embedding_dim, hidden_dim, batch_first=True) self.fc = nn.Linear(hidden_dim,
        1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.embedding(x) out, _ =
        self.rnn(x)
```

```
        out = out[:, -1, :]  # Get the output from the last time step out = self.fc(out)
        return self.sigmoid(out)

class LSTMModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim): super(LSTMModel, self).__
        init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim) self.lstm =
        nn.LSTM(embedding_dim, hidden_dim, batch_first=True) self.fc = nn.Linear(hidden_dim, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.embedding(x)
        out, (hn, _) = self.lstm(x)
        out = hn[-1, :, :]  # Get the output from the last LSTM cell out = self.fc(out)
        return self.sigmoid(out)

class GRUModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim): super(GRUModel, self).__
        init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim) self.gru =
        nn.GRU(embedding_dim, hidden_dim, batch_first=True) self.fc = nn.Linear(hidden_dim,
        1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.embedding(x) out, _ =
        self.gru(x)
        out = out[:, -1, :]  # Get the output from the last time step out = self.fc(out)
        return self.sigmoid(out)

class TransformerModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, num_heads, hidden_dim, num_layers):
        super(TransformerModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim) self.transformer =
        nn.TransformerEncoder(
            nn.TransformerEncoderLayer(d_model=embedding_dim, nhead=num_heads), num_layers=num_layers
        )
        self.fc = nn.Linear(embedding_dim, 1) self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.embedding(x)  # Apply embedding
```

```
        x = x.permute(1, 0, 2)  # Transformer expects input of shape (seq_len, batch_size, embedding_dim)
        out = self.transformer(x)
        out = out[-1, :, :]  # Get the output of the last token out = self.fc(out)
        return self.sigmoid(out) # 6.

Training and Evaluation

# Loss function and optimizer criterion =
nn.BCELoss() optimizer = optim.Adam

def train_model(model, train_loader, criterion, optimizer, epochs=3): for epoch in range(epochs):
        model.train() running_loss
        = 0.0
        correct_preds = 0
        total_preds = 0
        for inputs, labels in train_loader: optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs.squeeze(), labels.float()) loss.backward()
            optimizer.step()

            running_loss += loss.item()
            predicted = (outputs.squeeze() > 0.5).float() correct_preds += (predicted ==
            labels).sum().item() total_preds += labels.size(0)

        accuracy = 100 * correct_preds / total_preds print(f'Epoch
        [{epoch+1}/{epochs}], Loss:
{running_loss/len(train_loader):.4f}, Accuracy: {accuracy:.2f}%')

def evaluate_model(model, test_loader): model.eval()
    correct_preds = 0
    total_preds = 0
    with torch.no_grad():
        for inputs, labels in test_loader: outputs = model(inputs)
            predicted = (outputs.squeeze() > 0.5).float() correct_preds += (predicted ==
            labels).sum().item() total_preds += labels.size(0)

    accuracy = 100 * correct_preds / total_preds print(f'Test Accuracy:
    {accuracy:.2f}%')
```

```
#Instantiate and train models vocab_size =
len(vocab)

rnn_model = RNNModel(vocab_size, embedding_dim=50, hidden_dim=128) lstm_model=
LSTMModel(vocab_size,embedding_dim=50,hidden_dim=128) gru_model =
GRUModel(vocab_size, embedding_dim=50, hidden_dim=128)
transformer_model=TransformerModel(vocab_size,embedding_dim=64,num_heads=4, hidden_dim=128,
num_layers=2)

# Choose a model for training
model=rnn_model  #Change this to lstm_model, gru_model, or transformer_model to train other models
optimizer=optim.Adam(model.parameters(), lr=0.001)

train_model(model,train_loader,criterion,optimizer) evaluate_model(model, test_loader)
```

Output

```
Epoch [1/3], Loss: 0.7039, Accuracy: 50.00%
Epoch [2/3], Loss: 0.6958, Accuracy: 50.00%
Epoch [3/3], Loss: 0.7013, Accuracy: 50.00%
Test Accuracy: 50.00%

Process finished with exit code 0
```

Practical No – 10

Aim: To apply different deep learning models for Health Informatics.

Solution:

```
import numpy as np import pandas
as pd
from sklearn.model_selection import train_test_split from
sklearn.preprocessing import StandardScaler from sklearn.linear_model
import LogisticRegression from sklearn.ensemble import
RandomForestClassifier from sklearn.svm import SVC
from sklearn.metrics import accuracy_score import torch
import torch.nn as nn import
torch.optim as optim

# Load the dataset (Pima Indians Diabetes dataset) url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabe tes.data.csv"
column_names = ["Pregnancies", "Glucose", "BloodPressure", "SkinThickness", "Insulin", "BMI",
"DiabetesPedigreeFunction", "Age", "Outcome"]
data = pd.read_csv(url, names=column_names)

# Preprocessing
X = data.drop("Outcome", axis=1) y =
data["Outcome"]

# Standardize the features scaler =
StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Convert to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32) X_test_tensor = torch.tensor(X_test,
dtype=torch.float32) y_train_tensor = torch.tensor(y_train.values, dtype=torch.long) y_test_tensor =
torch.tensor(y_test.values, dtype=torch.long)

# --- Logistic Regression Model ---
# 1. Logistic Regression using scikit-learn logreg_model =
LogisticRegression() logreg_model.fit(X_train, y_train) logreg_predictions =
logreg_model.predict(X_test)
logreg_accuracy = accuracy_score(y_test, logreg_predictions) print(f"Logistic Regression Accuracy:
{logreg_accuracy:.4f}")
```

```
# --- Random Forest Classifier ---
# 2. Random Forest Classifier using scikit-learn
rf_model=RandomForestClassifier(n_estimators=100, random_state=42) rf_model.fit(X_train,
y_train)
rf_predictions = rf_model.predict(X_test) rf_accuracy=
accuracy_score(y_test, rf_predictions) print(f"Random Forest Accuracy:
{rf_accuracy:.4f}")

# --- Support Vector Classifier (SVC) ---
#3. Support Vector Classifier using scikit-learn svc_model = SVC()
svc_model.fit(X_train, y_train) svc_predictions=
svc_model.predict(X_test)
svc_accuracy=accuracy_score(y_test, svc_predictions) print(f"SVC Accuracy:
{svc_accuracy:.4f}")

# --- Feedforward Neural Network (FFNN) using PyTorch --- # 4. PyTorch FFNN
Model

class FFNN(nn.Module):
    def __init__(self, input_dim): super(FFNN, self).__init__()
        self.fc1 = nn.Linear(input_dim, 64)  # Input layer to hidden layer self.fc2 = nn.Linear(64, 32)
                                                        # Hidden layer
        self.fc3 = nn.Linear(32, 2)                     # Output layer (2 classes for binary
classification)
        self.relu = nn.ReLU() self.softmax=
        nn.Softmax(dim=1)

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return self.softmax(x)

# Instantiate and train the model
ffnn_model = FFNN(X_train.shape[1])
criterion = nn.CrossEntropyLoss()  # For binary classification,
CrossEntropyLoss is appropriate
optimizer = optim.Adam(ffnn_model.parameters(), lr=0.001)

# Train the FFNN model
epochs = 10
for epoch in range(epochs):
    ffnn_model.train()
    optimizer.zero_grad()

    # Forward pass
```

```
        outputs = ffnn_model(X_train_tensor) loss=
        criterion(outputs, y_train_tensor)

        # Backward pass and optimization
        loss.backward()  optimizer.step()

        if (epoch+1) % 2 == 0:
            print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')

# Evaluate the FFNN model
ffnn_model.eval()
with torch.no_grad():
    outputs = ffnn_model(X_test_tensor)
    _, predicted = torch.max(outputs, 1)
    ffnn_accuracy = accuracy_score(y_test, predicted.numpy()) print(f"FFNN Accuracy:
    {ffnn_accuracy:.4f}")

# --- Conclusion --- #
Compare all models
models_accuracies = {
    'Logistic Regression': logreg_accuracy, 'Random Forest':
    rf_accuracy,
    'SVC': svc_accuracy, 'FFNN':
    ffnn_accuracy
}

best_model = max(models_accuracies, key=models_accuracies.get) print(f"\nBest performing model:
{best_model} with accuracy
{models_accuracies[best_model]:.4f}")
```

Output

```
Logistic Regression Accuracy: 0.7532
Random Forest Accuracy: 0.7273
SVC Accuracy: 0.7273
Epoch [2/10], Loss: 0.6877
Epoch [4/10], Loss: 0.6825
Epoch [6/10], Loss: 0.6775
Epoch [8/10], Loss: 0.6725
Epoch [10/10], Loss: 0.6674
FFNN Accuracy: 0.7208

Best performing model: Logistic Regression with accuracy 0.7532

Process finished with exit code 0
```