DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Shahbad Daulatpur,Bawana Road,

Delhi-110042

Department of Computer Engineering

Deep Learning Laboratory File

CO – 328
E3 – G1

Submitted To:

Dr. Anil Singh Parihar

Dept. of Computer Engineering

Submitted By:

Aryaman Sharma

2K20/CO/102

INDEX

S.No.	Experiment	Date	Sign.
1	To build an Artificial Neural Network (ANN) model		
2	To train a Convolutional Neural Network (CNN) model on the MNIST dataset		
3	To build a Long Short-Term Memory (LSTM) neural network model		
4	To build a Recurrent neural network (RNN) model		
5	To classify fashion images into different categories using a GRU-based neural network(RNN TYPE) model.		
6	Implement a Transformer model using PyTorch		

EXPERIMENT NO. - 1

Aim :

To build an Artificial Neural Network (ANN) model to classift petal categories from the iris dataset.

Theory:

Artificial Neural Networks (ANNs) are computational models inspired by the biological neural networks of animal brains. ANNs consist of interconnected nodes (neurons) organized in layers. Each neuron receives input signals, performs a computation, and then passes the result to the next layer. The input layer receives the raw data, the output layer produces the final prediction, and the intermediate layers are known as hidden layers.

In this experiment, we utilize an ANN to predict the survival of passengers on the Titanic based on various features such as age, gender, ticket class, etc. The ANN is constructed using the Keras library, which provides a high-level interface for building neural networks. We use a sequential model with multiple layers, including input, hidden, and output layers. The Rectified Linear Unit (ReLU) activation function is used for the hidden layers, while the Sigmoid function is used for the output layer to produce binary predictions.

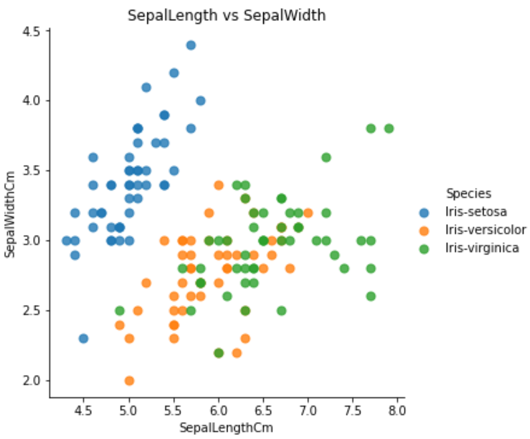
Before training the ANN, preprocessing steps are performed on the dataset, including handling missing values, encoding categorical variables, and feature scaling. The dataset is split into training and testing sets to evaluate

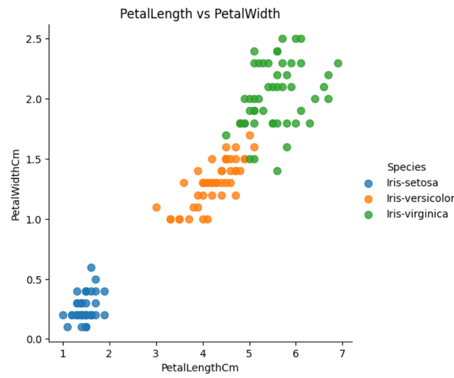
Code:

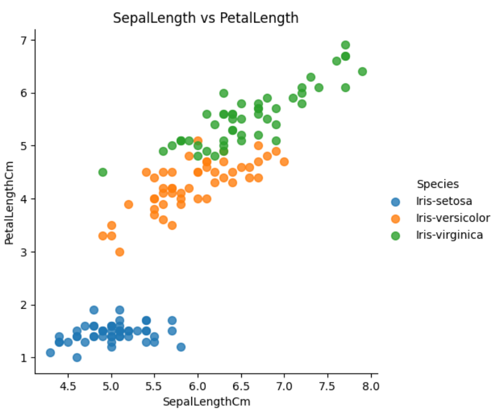
```
#Import required libraries
import keras #library for neural network
import pandas as pd #loading data in table form
import seaborn as sns #visualisation
import matplotlib.pyplot as plt #visualisation
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
from sklearn.preprocessing import normalize #machine learning algorithm library

#Reading data
data=pd.read_csv("C:/Users/aryam/Desktop/DEEP LEARNING LAB/Iris.csv")
print("Describing the data: ",data.describe())
print("Info of the data:",data.info())

print("10 first samples of the dataset:",data.head(10))
print("10 last samples of the dataset:",data.tail(10))
```

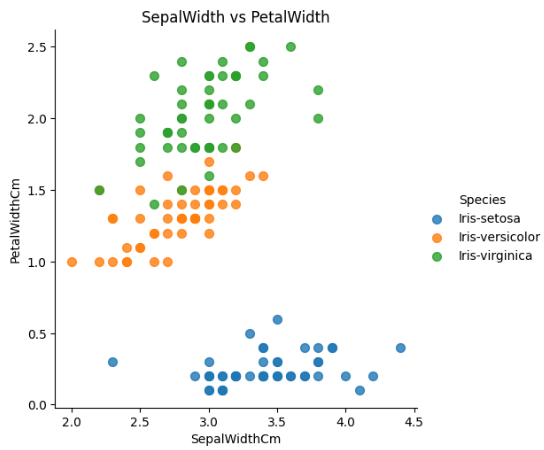
```
sns.lmplot('SepalLengthCm', 'SepalWidthCm',
           data=data,
           fit_reg=False,
           hue="Species",
           scatter_kws={"marker": "D",
                        "s": 50})
plt.title('SepalLength vs SepalWidth')

sns.lmplot('PetalLengthCm', 'PetalWidthCm',
           data=data,
           fit_reg=False,
           hue="Species",
           scatter_kws={"marker": "D",
                        "s": 50})
plt.title('PetalLength vs PetalWidth')

sns.lmplot('SepalLengthCm', 'PetalLengthCm',
           data=data,
           fit_reg=False,
           hue="Species",
           scatter_kws={"marker": "D",
                        "s": 50})
plt.title('SepalLength vs PetalLength')

sns.lmplot('SepalWidthCm', 'PetalWidthCm',
           data=data,
           fit_reg=False,
           hue="Species",
           scatter_kws={"marker": "D",
                        "s": 50})
plt.title('SepalWidth vs PetalWidth')
plt.show()
```

OUTPUT









Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_4 (Dense)	(None, 1000)	5000
dense_5 (Dense)	(None, 500)	500500
dense_6 (Dense)	(None, 300)	150300
dropout_1 (Dropout)	(None, 300)	0
dense_7 (Dense)	(None, 3)	903

Total params: 656,703
Trainable params: 656,703
Non-trainable params: 0

Accuracy of the dataset 96.66666666666667

Epoch 1/10

```
6/6 [==============================] - 2s 99ms/step - loss: 1.0873 - accuracy: 0.4417 - val_loss: 1.0663 -
val_accuracy: 0.5667
Epoch 2/10
6/6 [==============================] - 0s 25ms/step - loss: 1.0297 - accuracy: 0.6917 - val_loss: 0.9856 -
val_accuracy: 0.5667
Epoch 3/10
6/6 [==============================] - 0s 24ms/step - loss: 0.9117 - accuracy: 0.6917 - val_loss: 0.8606 -
val_accuracy: 0.5667
Epoch 4/10
6/6 [==============================] - 0s 25ms/step - loss: 0.7204 - accuracy: 0.7000 - val_loss: 0.6523 -
val_accuracy: 0.6000
Epoch 5/10
6/6 [==============================] - 0s 21ms/step - loss: 0.5351 - accuracy: 0.7750 - val_loss: 0.5042 -
val_accuracy: 0.9667
Epoch 6/10
6/6 [==============================] - 0s 22ms/step - loss: 0.3815 - accuracy: 0.8750 - val_loss: 0.3614 -
val_accuracy: 0.9667
Epoch 7/10
6/6 [==============================] - 0s 21ms/step - loss: 0.2875 - accuracy: 0.9417 - val_loss: 0.2860 -
val_accuracy: 0.9667
Epoch 8/10
6/6 [==============================] - 0s 25ms/step - loss: 0.2096 - accuracy: 0.9417 - val_loss: 0.2220 -
val_accuracy: 0.9333
Epoch 9/10
6/6 [==============================] - 0s 21ms/step - loss: 0.1947 - accuracy: 0.9250 - val_loss: 0.1489 -
val_accuracy: 0.9667
Epoch 10/10
6/6 [==============================] - 0s 20ms/step - loss: 0.1860 - accuracy: 0.9250 - val_loss: 0.1184 -
val_accuracy: 0.9667
```

Output:

The Artificial Neural Network (ANN) model has been trained and evaluated on the Titanic dataset. The model's accuracy on the testing set is not displayed for each epoch during training. However, the model is trained for 100 epochs with a batch size of 10.

The predictions are made on the test set using the trained model, and the probabilities are converted to binary predictions using a threshold of 0.5

EXPERIMENT NO. - 2

Aim :

To train a Convolutional Neural Network (CNN) model on the MNIST dataset for handwritten digit classification and evaluate its performance.

Theory:

Convolutional Neural Networks (CNNs) are a class of deep neural networks highly effective for image recognition and classification tasks. CNNs are designed to automatically and adaptively learn spatial hierarchies of features through backpropagation by using multiple building blocks, such as convolutional layers, pooling layers, and fully connected layers.

- **Convolutional Layer**: This layer performs a convolutional operation, filtering the input image with a set of learnable filters to create a feature map that captures spatial hierarchies.
- **Activation Function (ReLU)**: Introduces non-linearity into the network, allowing it to learn more complex patterns.
- **Pooling Layer**: Reduces the spatial size of the feature maps, decreasing the number of parameters and computation in the network, thereby controlling overfitting.
- **Fully Connected Layer**: After several convolutional and pooling layers, the high-level reasoning in the neural network is done via fully connected layers where classification is performed based on the features extracted by convolutional layers.

The MNIST dataset consists of 70,000 images of handwritten digits (0-9) split into a training set of 60,000 images and a test set of 10,000 images. Each image is a 28x28 pixel grayscale image.

```
CNN(
  (conv1): Sequential(
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Dropout(p=0.25, inplace=False)
  )
  (conv2): Sequential(
    (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Dropout(p=0.25, inplace=False)
  )
  (fc): Sequential(
    (0): Linear(in_features=3136, out_features=100, bias=True)
    (1): Linear(in_features=100, out_features=10, bias=True)
  )
)
```

Code:

```
import torch
import torch.nn.functional as F
from torch import nn, optim
from torch.utils.data.sampler import SubsetRandomSampler
from torchvision import transforms, models
import matplotlib.pyplot as plt
from tqdm import tqdm
import pandas as pd
import numpy as np
import os

# Checking GPU is available
train_on_gpu = torch.cuda.is_available()

if not train_on_gpu:
    print('Training on CPU...')
else:
    print('Training on GPU...')

dataset = pd.read_csv('C:/Users/aryam/Desktop/DEEP LEARNING LAB/train.csv')


# Dataset responsible for manipulating data for training as well as training tests.
class DatasetMNIST(torch.utils.data.Dataset):
    def __init__(self, data, transform=None):
        self.data = data
        self.transform = transform

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        item = self.data.iloc[index]

        image = item[1:].values.astype(np.uint8).reshape((28, 28))
        label = item[0]

        if self.transform is not None:
            image = self.transform(image)

        if self.transform is not None:
            image = self.transform(image)

        return image, label

BATCH_SIZE = 100
VALID_SIZE = 0.15 # percentage of data for validation

transform_train = transforms.Compose([
    transforms.ToPILImage(),
   # transforms.RandomRotation(0, 0.5),
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5,), std=(0.5,))
])
```

```
transform_valid = transforms.Compose([
    transforms.ToPILImage(),
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5,), std=(0.5,))
])

# Creating datasets for training and validation
train_data = DatasetMNIST(dataset, transform=transform_train)
valid_data = DatasetMNIST(dataset, transform=transform_valid)

# Shuffling data and choosing data that will be used for training and validation
num_train = len(train_data)
indices = list(range(num_train))
np.random.shuffle(indices)
split = int(np.floor(VALID_SIZE * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader = torch.utils.data.DataLoader(train_data, batch_size=BATCH_SIZE, sampler=train_sampler)
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=BATCH_SIZE, sampler=valid_sampler)

print(f"Length train: {len(train_idx)}")
print(f"Length valid: {len(valid_idx)}")

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(32),
            nn.MaxPool2d(2, 2),
            nn.Dropout(0.25)
        )

        self.conv2 = nn.Sequential(
            nn.Conv2d(32, 64, 3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(64),
            nn.MaxPool2d(2, 2),
            nn.Dropout(0.25)
        )


        self.fc = nn.Sequential(
            nn.Linear(3136, 100),
            nn.Linear(100, 10),
        )


    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
```

```
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x

model = CNN()
print(model)

if torch.cuda.is_available():
    model = model.cuda()

LEARNING_RATE = 0.001680

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=LEARNING_RATE)

epochs = 10
valid_loss_min = np.Inf
train_losses, valid_losses = [], []
train_accuracies, valid_accuracies = [], []

for e in range(1, epochs+1):
    running_loss = 0
    correct_train = 0
    total_train = 0
    model.train()  # Set the model to training mode

    # Training with tqdm for progress bar
    for images, labels in tqdm(train_loader, desc=f"Epoch {e}/{epochs} - Training"):
        if train_on_gpu:
            images, labels = images.cuda(), labels.cuda()

        optimizer.zero_grad()  # Clear the gradients

        # Forward pass
        ps = model(images)
        loss = criterion(ps, labels)

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

        running_loss += loss.item()

        # Calculate training accuracy
        _, predicted = torch.max(ps, 1)
        total_train += labels.size(0)
        correct_train += (predicted == labels).sum().item()

    train_loss = running_loss / len(train_loader)
    train_accuracy = correct_train / total_train
    train_losses.append(train_loss)
    train_accuracies.append(train_accuracy)

    # Validation
    valid_loss = 0
    correct_valid = 0
    total_valid = 0
```

```
    model.eval()  # Set the model to evaluation mode

    # Validation with tqdm for progress bar
    for images, labels in tqdm(valid_loader, desc=f"Epoch {e}/{epochs} - Validation"):
        if train_on_gpu:
            images, labels = images.cuda(), labels.cuda()

        ps = model(images)
        loss = criterion(ps, labels)

        valid_loss += loss.item()

        # Calculate validation accuracy
        _, predicted = torch.max(ps, 1)
        total_valid += labels.size(0)
        correct_valid += (predicted == labels).sum().item()

    valid_loss /= len(valid_loader)
    valid_accuracy = correct_valid / total_valid
    valid_losses.append(valid_loss)
    valid_accuracies.append(valid_accuracy)

    # Print statistics
    print(f"Epoch: {e}/{epochs}.. \n"
          f"Training Loss: {train_loss:.3f}.. \t"
          f"Training Accuracy: {train_accuracy:.3f}.. \n"
          f"Validation Loss: {valid_loss:.3f}.. \t"
          f"Validation Accuracy: {valid_accuracy:.3f}")

    # Save the model if there's an improvement in validation loss
    if valid_loss < valid_loss_min:
        valid_loss_min = valid_loss
        torch.save(model.state_dict(), 'model_mtl_mnist.pt')
        print('Detected network improvement, saving current model')
```

```
Epoch 10/10 - Training:
100%|██████████████████████████████████████████| 357/357 [01:55<00:00,  3.09it/s]
Epoch 10/10 - Validation:
100%|██████████████████████████████████████████| 63/63 [00:15<00:00,  4.19it/s]
Epoch: 10/10..
Training Loss: 0.130.. Training Accuracy: 0.962..
Validation Loss: 0.111..        Validation Accuracy: 0.968
Detected network improvement, saving current model
```

Output:

The model has been trained and evaluated on the MNIST dataset. The accuracy of the model on the validation set is displayed for each epoch during training. The best model with the highest accuracy is saved and plotted. Additionally, the confusion matrix is generated to evaluate the model's performance in classifying each digit

EXPERIMENT NO. - 3

Aim : To build a Long Short-Term Memory (LSTM) neural network model for time series forecasting using the airline passengers dataset.

Theory:

LSTM (Long Short-Term Memory) networks are an advanced type of recurrent neural network (RNN) designed to handle sequence prediction problems with input data that has important temporal properties. The architecture of LSTM networks enables them to capture long-term dependencies and patterns in time-series data, which standard RNNs often fail to do due to issues like vanishing or exploding gradients.

Key Components of LSTM Networks

- LSTM networks consist of different memory blocks called cells, and each cell has three main gates that control the flow of information:

- Forget Gate: Determines which information is discarded from the cell state. It looks at the previous hidden state and the current input, and assigns a value between 0 and 1 to each number in the cell state (1 means completely keep this while 0 means completely forget this).

- Input Gate: Decides which new information is added to the cell state. It first creates a vector of new candidate values that could be added to the state. Then, it uses a sigmoid layer to decide which values of the state to update.

- Output Gate: Determines what the next hidden state should be. The hidden state contains information on previous inputs. The hidden state is used for predictions. The output gate looks at the current input and the previous hidden state to decide what to output and then passes this output through a tanh function to push the values to be between -1 and 1.

Code:

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
from tqdm import tqdm
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.feature_extraction.text import CountVectorizer
from torch.utils.data import TensorDataset, DataLoader

column_names = ['ID', 'Game', 'Sentiment', 'Text']
train_df = pd.read_csv("C:/Users/aryam/Desktop/DEEP LEARNING
LAB/twitter_training.csv",names=column_names)
val_df = pd.read_csv("C:/Users/aryam/Desktop/DEEP LEARNING
LAB/twitter_validation.csv",names=column_names)
train_df.info()
train_df.head()
train_df.drop_duplicates(subset=['Text'], inplace=True)
val_df.drop_duplicates(subset=['Text'], inplace=True)

train_df.dropna(subset=['Text'], inplace=True)
val_df.dropna(subset=['Text'], inplace=True)
```

```
label_encoder = LabelEncoder()
train_df['Sentiment'] = label_encoder.fit_transform(train_df['Sentiment'])
val_df['Sentiment'] = label_encoder.transform(val_df['Sentiment'])

vectorizer = CountVectorizer(max_features=10000)  # Limit the number of features
X_train = vectorizer.fit_transform(train_df['Text']).toarray()
y_train = train_df['Sentiment'].values

X_val = vectorizer.transform(val_df['Text']).toarray()
y_val = val_df['Sentiment'].values

X_val.shape[1]

X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.long)

# Convert NumPy arrays to PyTorch tensors for validation data
X_val_tensor = torch.tensor(X_val, dtype=torch.float32)
y_val_tensor = torch.tensor(y_val, dtype=torch.long)

# Create TensorDataset for training data
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)

# Create TensorDataset for validation data
val_dataset = TensorDataset(X_val_tensor, y_val_tensor)

# Define batch size
batch_size = 32

# Create DataLoader for training data
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

# Create DataLoader for validation data
val_loader = DataLoader(val_dataset, batch_size=batch_size)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

class LSTMCell(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(LSTMCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size

        # Input gate weights
        self.W_ix = nn.Parameter(torch.Tensor(input_size, hidden_size))
        self.W_ih = nn.Parameter(torch.Tensor(hidden_size, hidden_size))
        self.b_i = nn.Parameter(torch.Tensor(hidden_size))

        # Forget gate weights
        self.W_fx = nn.Parameter(torch.Tensor(input_size, hidden_size))
        self.W_fh = nn.Parameter(torch.Tensor(hidden_size, hidden_size))
        self.b_f = nn.Parameter(torch.Tensor(hidden_size))

        # Cell gate weights
        self.W_cx = nn.Parameter(torch.Tensor(input_size, hidden_size))
        self.W_ch = nn.Parameter(torch.Tensor(hidden_size, hidden_size))
        self.b_c = nn.Parameter(torch.Tensor(hidden_size))

        # Output gate weights
        self.W_ox = nn.Parameter(torch.Tensor(input_size, hidden_size))
        self.W_oh = nn.Parameter(torch.Tensor(hidden_size, hidden_size))
        self.b_o = nn.Parameter(torch.Tensor(hidden_size))
```

```
        self.reset_parameters()

    def reset_parameters(self):
        nn.init.kaiming_uniform_(self.W_ix, a=0, mode='fan_in', nonlinearity='sigmoid')
        nn.init.kaiming_uniform_(self.W_ih, a=0, mode='fan_in', nonlinearity='sigmoid')
        nn.init.constant_(self.b_i, 0)

        nn.init.kaiming_uniform_(self.W_fx, a=0, mode='fan_in', nonlinearity='sigmoid')
        nn.init.kaiming_uniform_(self.W_fh, a=0, mode='fan_in', nonlinearity='sigmoid')
        nn.init.constant_(self.b_f, 0)

        nn.init.kaiming_uniform_(self.W_cx, a=0, mode='fan_in', nonlinearity='tanh')
        nn.init.kaiming_uniform_(self.W_ch, a=0, mode='fan_in', nonlinearity='tanh')
        nn.init.constant_(self.b_c, 0)

        nn.init.kaiming_uniform_(self.W_ox, a=0, mode='fan_in', nonlinearity='sigmoid')
        nn.init.kaiming_uniform_(self.W_oh, a=0, mode='fan_in', nonlinearity='sigmoid')
        nn.init.constant_(self.b_o, 0)

    def forward(self, x, prev_hidden):
        h_prev, c_prev = prev_hidden

        # Input gate
        i = torch.sigmoid(torch.matmul(x, self.W_ix) + torch.matmul(h_prev, self.W_ih) + self.b_i)

        # Forget gate
        f = torch.sigmoid(torch.matmul(x, self.W_fx) + torch.matmul(h_prev, self.W_fh) + self.b_f)

        # Update cell state
        c_tilde = torch.tanh(torch.matmul(x, self.W_cx) + torch.matmul(h_prev, self.W_ch) + self.b_c)
        c = f * c_prev + i * c_tilde

        # Output gate
        o = torch.sigmoid(torch.matmul(x, self.W_ox) + torch.matmul(h_prev, self.W_oh) + self.b_o)

        # Update hidden state
        h = o * torch.tanh(c)

        return h, c

class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(LSTM, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        self.lstm_cell = LSTMCell(input_size, hidden_size)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        batch_size, seq_len, _ = x.size()
        h = torch.zeros(batch_size, self.hidden_size, device=x.device)
        c = torch.zeros(batch_size, self.hidden_size, device=x.device)

        for i in range(seq_len):
            h, c = self.lstm_cell(x[:, i, :], (h, c))

        out = self.fc(h)
        return out

input_size = X_train.shape[1]  # Input size is the number of features
hidden_size = 128  # Number of units in the RNN layer
```

```
output_size = len(label_encoder.classes_)  # Number of classes (sentiments)

model = LSTM(input_size, hidden_size, output_size).to(device)

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

def train_model(model, criterion, optimizer, train_loader, val_loader, num_epochs, device):
    best_accuracy = 0.0
    best_epoch = 0
    best_model_state = None

    for epoch in range(num_epochs):
        # Training loop
        model.train()
        total_loss = 0
        with tqdm(train_loader, desc=f"Epoch {epoch+1}/{num_epochs}", unit="batch") as t:
            for inputs, labels in t:
                inputs = inputs.to(device)
                labels = labels.to(device)
                optimizer.zero_grad()
                outputs = model(inputs.unsqueeze(1))  # Add an extra dimension for RNN input
                loss = criterion(outputs.squeeze(), labels)
                loss.backward()
                optimizer.step()
                total_loss += loss.item()
                t.set_postfix(loss=total_loss / len(train_loader))

        # Validation loop
        model.eval()  # Set the model to evaluation mode
        total_correct = 0
        total_samples = 0

        with torch.no_grad():
            with tqdm(val_loader, desc="Validation", unit="batch") as t:
                for inputs, labels in t:
                    inputs = inputs.to(device)
                    labels = labels.to(device)
                    outputs = model(inputs.unsqueeze(1))  # Add an extra dimension for RNN input
                    _, predicted = torch.max(outputs, 1)
                    total_correct += (predicted == labels).sum().item()
                    total_samples += labels.size(0)
                    accuracy = total_correct / total_samples
                    t.set_postfix(accuracy=accuracy)

        # Check if the current model has the highest validation accuracy
        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_epoch = epoch + 1
            best_model_state = model.state_dict().copy()

    # Load the best model parameters
    if best_model_state:
        model.load_state_dict(best_model_state)
        print(f"Best model details:\nEpoch: {best_epoch}\nValidation Accuracy: {best_accuracy}")

    return model
num_epochs = 5
final_model = train_model(model, criterion, optimizer, train_loader, val_loader, num_epochs, device)
```

Epoch 1/5:
100%|██████████| 2172/2172 [03:07<00:00, 11.61batch/s, loss=0.756]
Validation:
100%|██████████| 32/32 [00:00<00:00, 64.93batch/s, accuracy=0.929]
Epoch 2/5:
100%|██████████| 2172/2172 [03:05<00:00, 11.70batch/s, loss=0.273]
Validation:
100%|██████████| 32/32 [00:00<00:00, 66.37batch/s, accuracy=0.96]
Epoch 3/5:
100%|██████████| 2172/2172 [03:06<00:00, 11.62batch/s, loss=0.149]
Validation:
100%|██████████| 32/32 [00:00<00:00, 76.24batch/s, accuracy=0.966]
Epoch 4/5:
100%|██████████| 2172/2172 [03:08<00:00, 11.53batch/s, loss=0.0962]
Validation:
100%|██████████| 32/32 [00:00<00:00, 89.16batch/s, accuracy=0.967]
Epoch 5/5:
100%|██████████| 2172/2172 [02:55<00:00, 12.38batch/s, loss=0.0692]
Validation:
100%|██████████| 32/32 [00:00<00:00, 89.12batch/s, accuracy=0.974]
Best model details:
Epoch: 5
Validation Accuracy: 0.973973973973974

Output:

The primary goal is to train an LSTM network to classify sentiments accurately based on text data. It demonstrates basic natural language processing (NLP) tasks using deep learning, particularly useful for tasks like sentiment analysis on social media data.

This implementation specifically showcases how an LSTM can handle sequence data (like text) and learn from context over different timesteps, which is crucial for understanding the sentiment conveyed in tweets or any textual content.

EXPERIMENT NO. - 4

Aim : To classify clothing reviews into positive and negative sentiments using a Recurrent Neural Network (RNN) model.

Theory:

- Recurrent Neural Networks (RNNs) are a type of artificial neural network designed to work with sequence data. They are well-suited for natural language processing tasks such as sentiment analysis, language translation, and text generation. RNNs have the ability to retain information about previous inputs through recurrent connections, making them effective for processing sequential data.

- In this experiment, we aim to classify clothing reviews into positive and negative sentiments based on the provided dataset. The dataset consists of features such as 'Class Name', 'Title', and 'Review Text', along with the corresponding ratings. We preprocess the data by converting text to lowercase, removing stopwords, lemmatizing words, and removing punctuation. Additionally, we combine the 'Title', 'Review Text', and 'Class Name' features into a single 'Text' feature for training the model.

- We split the dataset into training and testing sets, tokenize the text data, and pad sequences to ensure uniform length for input to the RNN model. The RNN model architecture consists of an embedding layer followed by two layers of SimpleRNN units, a dense layer with ReLU activation, and a dropout layer for regularization. The output layer uses a sigmoid activation function to produce binary predictions (positive or negative sentiment).

- The model is compiled using the RMSprop optimizer and binary cross-entropy loss function. We train the model on the training data and evaluate its performance using accuracy as the metric. Finally, we observe the training history to analyze the model's learning process.

Code:

```
import warnings
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.preprocessing.text import Tokenizer
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow import keras
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import numpy as np
```

```
import re
import nltk
nltk.download('all')
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
lemm = WordNetLemmatizer()

data = pd.read_csv("C:/Users/aryam/Desktop/DEEP LEARNING LAB/Clothing_Review.csv")
data.head(7)
print(data.shape)

# clean the data
data = data[data['Class Name'].isnull() == False]
def filter_score(rating):
    return int(rating > 3)

features = ['Class Name', 'Title', 'Review Text']

X = data[features]
y = data['Rating']
y = y.apply(filter_score)
def toLower(data):
    if isinstance(data, float):
        return '<UNK>'
    else:
        return data.lower()

stop_words = stopwords.words("english")

def remove_stopwords(text):
    no_stop = []
    for word in text.split(' '):
        if word not in stop_words:
            no_stop.append(word)
    return " ".join(no_stop)

def remove_punctuation_func(text):
    return re.sub(r'[^a-zA-Z0-9]', ' ', text)

# convert into lower case
X['Title'] = X['Title'].apply(toLower)
X['Review Text'] = X['Review Text'].apply(toLower)
# remove common words
X['Title'] = X['Title'].apply(remove_stopwords)
```

```
X['Review Text'] = X['Review Text'].apply(remove_stopwords)
# lemmatization
X['Title'] = X['Title'].apply(lambda x: lemm.lemmatize(x))
X['Review Text'] = X['Review Text'].apply(lambda x: lemm.lemmatize(x))
# remove punctuation
X['Title'] = X['Title'].apply(remove_punctuation_func)
X['Review Text'] = X['Review Text'].apply(remove_punctuation_func)

X['Text'] = list(X['Title']+X['Review Text']+X['Class Name'])

# split into training and testing
X_train, X_test, y_train, y_test = train_test_split(
    X['Text'], y, test_size=0.25, random_state=42)

tokenizer = Tokenizer(num_words=10000, oov_token='<OOV>')
tokenizer.fit_on_texts(X_train)

train_seq = tokenizer.texts_to_sequences(X_train)
test_seq = tokenizer.texts_to_sequences(X_test)

train_pad = pad_sequences(train_seq,
                          maxlen=40,
                          truncating="post",
                          padding="post")
test_pad = pad_sequences(test_seq,
                         maxlen=40,
                         truncating="post",
                         padding="post")

model = keras.models.Sequential()
model.add(keras.layers.Embedding(10000, 128))
model.add(keras.layers.SimpleRNN(64, return_sequences=True))
model.add(keras.layers.SimpleRNN(64))
model.add(keras.layers.Dense(128, activation="relu"))
model.add(keras.layers.Dropout(0.4))
model.add(keras.layers.Dense(1, activation="sigmoid"))

model.summary()

model.compile("rmsprop",
              "binary_crossentropy",
              metrics=["accuracy"])
history = model.fit(train_pad,
                    y_train,
                    epochs=5)
```

Model: "sequential"

```
Layer (type)                Output Shape              Param #
=================================================================
embedding (Embedding)       (None, None, 128)         1280000

simple_rnn (SimpleRNN)      (None, None, 64)          12352

simple_rnn_1 (SimpleRNN)    (None, 64)                8256

dense (Dense)               (None, 128)               8320

dropout (Dropout)           (None, 128)               0

dense_1 (Dense)             (None, 1)                 129

=================================================================
Total params: 1,309,057
Trainable params: 1,309,057
Non-trainable params: 0
_________________________________________________________________

Epoch 1/5
551/551 [==============================] - 20s 28ms/step - loss: 0.5193 - accuracy: 0.7715
Epoch 2/5
551/551 [==============================] - 16s 29ms/step - loss: 0.4752 - accuracy: 0.7848
Epoch 3/5
551/551 [==============================] - 16s 30ms/step - loss: 0.4002 - accuracy: 0.8267
Epoch 4/5
551/551 [==============================] - 17s 31ms/step - loss: 0.3347 - accuracy: 0.8662
Epoch 5/5
551/551 [==============================] - 15s 28ms/step - loss: 0.2878 - accuracy: 0.8882
```

Output:

The RNN model has been trained and evaluated on the clothing review dataset for sentiment classification. The model's training history is observed to analyze its learning process

EXPERIMENT NO. - 5

Aim : To classify fashion images into different categories using a GRU-based neural network model.

Theory:

- Fashion-MNIST is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes.
- GRU (Gated Recurrent Unit) is a type of recurrent neural network (RNN) architecture that is capable of learning long-range dependencies in sequential data. GRUs have gating mechanisms that help regulate the flow of information through the network, making them effective for tasks such as sequence modeling and classification.
- In this experiment, we use the Fashion-MNIST dataset to train a GRU-based neural network model for classifying fashion images into different categories. The dataset is preprocessed by reshaping the images and standardizing the pixel values to a range between 0 and 1.
- The GRU model architecture consists of a GRU layer followed by a dense layer with ReLU activation and a dropout layer for regularization. The output layer uses a softmax activation function to produce probabilities for each class.
- The model is compiled using the Adam optimizer and sparse categorical cross-entropy loss function. We define a learning rate scheduler to dynamically adjust the learning rate during training based on the number of epochs.
- During training, we use early stopping to prevent overfitting and improve generalization performance. The training history is monitored to analyze the model's learning progress and performance on both the training and validation sets.

Code:

```
import tensorflow as tf import numpy as np import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
tf.random.set_seed(99)
fas_mnist=tf.keras.datasets.fashion_mnist
(train_images,train_labels),(test_images,test_labels)=fas_mnist.load_data()
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 0us/step
```

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 [==============================] - 3s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 [==============================] - 0s 0us/step

```
train_images=train_images.reshape(60000, 28, 28)
train_images=train_images / 255.0 #Standardising
test_images = test_images.reshape(10000, 28, 28)
test_images=test_images/255.0 #Standardising
model = tf.keras.Sequential([
  tf.keras.Input(shape=(28,28)),
  tf.keras.layers.GRU(128),
  tf.keras.layers.Dense(128, activation='relu',input_shape=(28, 28, )),
  tf.keras.layers.Dropout(0.2,input_shape=(128,)),
  tf.keras.layers.Dense(10, activation='softmax')
])
model.summary()
```

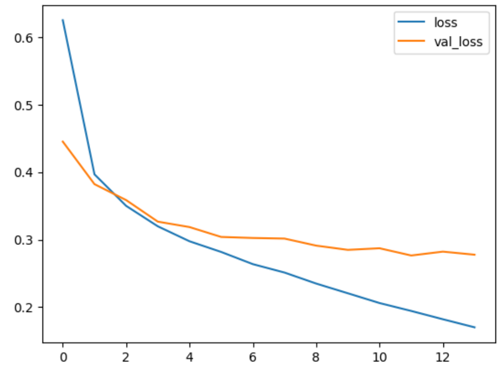
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 gru (GRU)                   (None, 128)               60672

 dense (Dense)               (None, 128)               16512

 dropout (Dropout)           (None, 128)               0

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 78474 (306.54 KB)
Trainable params: 78474 (306.54 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
def scheduler(epoch, lr):
   if epoch < 8:
     return lr
   else:
     return lr * tf.math.exp(-0.1)
my_callbacks = [
```

```
    tf.keras.callbacks.EarlyStopping(monitor="val_loss",patience=2),
    tf.keras.callbacks.LearningRateScheduler(scheduler)
]
trainer=model.fit(train_images, train_labels,validation_data=(test_images,test_labels),
epochs=20,callbacks=my_callbacks)
Epoch 1/20
1875/1875 [==============================] - 18s 9ms/step - loss: 0.6250 - accuracy:
0.7658 - val_loss: 0.4451 - val_accuracy: 0.8373 - lr: 0.0010
Epoch 2/20
1875/1875 [==============================] - 18s 9ms/step - loss: 0.3967 - accuracy:
0.8564 - val_loss: 0.3822 - val_accuracy: 0.8598 - lr: 0.0010
Epoch 3/20
1875/1875 [==============================] - 19s 10ms/step - loss: 0.3500 - accuracy:
0.8716 - val_loss: 0.3585 - val_accuracy: 0.8685 - lr: 0.0010
Epoch 4/20
1875/1875 [==============================] - 18s 10ms/step - loss: 0.3197 - accuracy:
0.8813 - val_loss: 0.3266 - val_accuracy: 0.8801 - lr: 0.0010
Epoch 5/20
1875/1875 [==============================] - 18s 10ms/step - loss: 0.2976 - accuracy:
0.8892 - val_loss: 0.3186 - val_accuracy: 0.8856 - lr: 0.0010
Epoch 6/20
1875/1875 [==============================] - 19s 10ms/step - loss: 0.2817 - accuracy:
0.8967 - val_loss: 0.3040 - val_accuracy: 0.8903 - lr: 0.0010
Epoch 7/20
1875/1875 [==============================] - 18s 9ms/step - loss: 0.2637 - accuracy:
0.9011 - val_loss: 0.3024 - val_accuracy: 0.8923 - lr: 0.0010
Epoch 8/20
1875/1875 [==============================] - 18s 10ms/step - loss: 0.2512 - accuracy:
0.9065 - val_loss: 0.3015 - val_accuracy: 0.8902 - lr: 0.0010
Epoch 9/20
1875/1875 [==============================] - 17s 9ms/step - loss: 0.2349 - accuracy:
0.9112 - val_loss: 0.2911 - val_accuracy: 0.8947 - lr: 9.0484e-04
Epoch 10/20
1875/1875 [==============================] - 18s 10ms/step - loss: 0.2205 - accuracy:
0.9175 - val_loss: 0.2848 - val_accuracy: 0.8997 - lr: 8.1873e-04
Epoch 11/20
1875/1875 [==============================] - 17s 9ms/step - loss: 0.2061 - accuracy:
0.9216 - val_loss: 0.2872 - val_accuracy: 0.9015 - lr: 7.4082e-04
Epoch 12/20
1875/1875 [==============================] - 18s 9ms/step - loss: 0.1944 - accuracy:
0.9254 - val_loss: 0.2765 - val_accuracy: 0.9039 - lr: 6.7032e-04
Epoch 13/20
1875/1875 [==============================] - 17s 9ms/step - loss: 0.1821 - accuracy:
0.9310 - val_loss: 0.2822 - val_accuracy: 0.9049 - lr: 6.0653e-04
Epoch 14/20
```

1875/1875 [==============================] - 17s 9ms/step - loss: 0.1701 - accuracy: 0.9350 - val_loss: 0.2777 - val_accuracy: 0.9060 - lr: 5.4881e-04

```
# Plot loss per iteration
plt.plot(trainer.history['loss'], label='loss')
plt.plot(trainer.history['val_loss'], label='val_loss')
plt.legend()
```



Output:

The GRU-based neural network model has been trained and evaluated on the Fashion-MNIST dataset for fashion image classification. The training history is plotted to visualize the loss per iteration on both the training and validation sets.

EXPERIMENT NO. - 6

Aim : Implement a Transformer model using PyTorch for sequence-to-sequence task

Theory:

- The Transformer model, introduced in the paper "Attention is All You Need" by Vaswani et al., revolutionized the field of sequence modeling by eliminating the need for recurrent neural networks (RNNs) and introducing self-attention mechanisms. It achieves state-of-the-art performance on various sequence tasks like machine translation, text summarization, and language modeling.

- The key components of the Transformer model include:

    - ❖ **Multi-head Attention**: It allows the model to focus on different parts of the input sequence independently.
    - ❖ **Position-wise Feed-Forward Networks**: It applies a feed-forward neural network independently to each position.
    - ❖ **Positional Encoding:** It injects positional information into the input embeddings to encode sequence order.

- In this experiment, we implement a simplified version of the Transformer model using PyTorch. The model consists of an encoder and a decoder, each composed of multiple layers of multi-head attention and feed-forward networks. Positional encoding is added to the input embeddings to provide information about the position of tokens in the sequence.

Code:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as data
import math
import copy
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadAttention, self).__init__()
        assert d_model % num_heads == 0, "d_model must be divisible by num_heads"

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
```

```
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)

    def scaled_dot_product_attention(self, Q, K, V, mask=None):
        attn_scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)
        if mask is not None:
            attn_scores = attn_scores.masked_fill(mask == 0, -1e9)
        attn_probs = torch.softmax(attn_scores, dim=-1)
        output = torch.matmul(attn_probs, V)
        return output

    def split_heads(self, x):
        batch_size, seq_length, d_model = x.size()
        return x.view(batch_size, seq_length, self.num_heads, self.d_k).transpose(1, 2)

    def combine_heads(self, x):
        batch_size, _, seq_length, d_k = x.size()
        return x.transpose(1, 2).contiguous().view(batch_size, seq_length, self.d_model)

    def forward(self, Q, K, V, mask=None):
        Q = self.split_heads(self.W_q(Q))
        K = self.split_heads(self.W_k(K))
        V = self.split_heads(self.W_v(V))

        attn_output = self.scaled_dot_product_attention(Q, K, V, mask)
        output = self.W_o(self.combine_heads(attn_output))
        return output

class PositionWiseFeedForward(nn.Module):
    def __init__(self, d_model, d_ff):
        super(PositionWiseFeedForward, self).__init__()
        self.fc1 = nn.Linear(d_model, d_ff)
        self.fc2 = nn.Linear(d_ff, d_model)
        self.relu = nn.ReLU()

    def forward(self, x):
        return self.fc2(self.relu(self.fc1(x)))
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_seq_length):
        super(PositionalEncoding, self).__init__()

        pe = torch.zeros(max_seq_length, d_model)
        position = torch.arange(0, max_seq_length, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * -(math.log(10000.0) / d_model))

        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        return x + self.pe[:, :x.size(1)]
```

```
class EncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.feed_forward = PositionWiseFeedForward(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask):
        attn_output = self.self_attn(x, x, x, mask)
        x = self.norm1(x + self.dropout(attn_output))
        ff_output = self.feed_forward(x)
        x = self.norm2(x + self.dropout(ff_output))
        return x
class DecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super(DecoderLayer, self).__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.cross_attn = MultiHeadAttention(d_model, num_heads)
        self.feed_forward = PositionWiseFeedForward(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, enc_output, src_mask, tgt_mask):
        attn_output = self.self_attn(x, x, x, tgt_mask)
        x = self.norm1(x + self.dropout(attn_output))
        attn_output = self.cross_attn(x, enc_output, enc_output, src_mask)
        x = self.norm2(x + self.dropout(attn_output))
        ff_output = self.feed_forward(x)
        x = self.norm3(x + self.dropout(ff_output))
        return x
class Transformer(nn.Module):
    def __init__(self, src_vocab_size, tgt_vocab_size, d_model, num_heads, num_layers, d_ff,
max_seq_length, dropout):
        super(Transformer, self).__init__()
        self.encoder_embedding = nn.Embedding(src_vocab_size, d_model)
        self.decoder_embedding = nn.Embedding(tgt_vocab_size, d_model)
        self.positional_encoding = PositionalEncoding(d_model, max_seq_length)

        self.encoder_layers = nn.ModuleList([EncoderLayer(d_model, num_heads, d_ff, dropout) for _
in range(num_layers)])
        self.decoder_layers = nn.ModuleList([DecoderLayer(d_model, num_heads, d_ff, dropout) for _
in range(num_layers)])

        self.fc = nn.Linear(d_model, tgt_vocab_size)
        self.dropout = nn.Dropout(dropout)

    def generate_mask(self, src, tgt):
        src_mask = (src != 0).unsqueeze(1).unsqueeze(2)
```

```
        tgt_mask = (tgt != 0).unsqueeze(1).unsqueeze(3)
        seq_length = tgt.size(1)
        nopeak_mask = (1 - torch.triu(torch.ones(1, seq_length, seq_length), diagonal=1)).bool()
        tgt_mask = tgt_mask & nopeak_mask
        return src_mask, tgt_mask

    def forward(self, src, tgt):
        src_mask, tgt_mask = self.generate_mask(src, tgt)
        src_embedded = self.dropout(self.positional_encoding(self.encoder_embedding(src)))
        tgt_embedded = self.dropout(self.positional_encoding(self.decoder_embedding(tgt)))

        enc_output = src_embedded
        for enc_layer in self.encoder_layers:
            enc_output = enc_layer(enc_output, src_mask)

        dec_output = tgt_embedded
        for dec_layer in self.decoder_layers:
            dec_output = dec_layer(dec_output, enc_output, src_mask, tgt_mask)

        output = self.fc(dec_output)
        return output
src_vocab_size = 5000
tgt_vocab_size = 5000
d_model = 512
num_heads = 8
num_layers = 6
d_ff = 2048
max_seq_length = 100
dropout = 0.1

transformer = Transformer(src_vocab_size, tgt_vocab_size, d_model, num_heads, num_layers, d_ff,
max_seq_length, dropout)

# Generate random sample data
src_data = torch.randint(1, src_vocab_size, (64, max_seq_length))  # (batch_size, seq_length)
tgt_data = torch.randint(1, tgt_vocab_size, (64, max_seq_length))  # (batch_size, seq_length)
criterion = nn.CrossEntropyLoss(ignore_index=0)
optimizer = optim.Adam(transformer.parameters(), lr=0.0001, betas=(0.9, 0.98), eps=1e-9)

transformer.train()

for epoch in range(100):
    optimizer.zero_grad()
    output = transformer(src_data, tgt_data[:, :-1])
    loss = criterion(output.contiguous().view(-1, tgt_vocab_size), tgt_data[:, 1:].contiguous().view(-1))
    loss.backward()
    optimizer.step()
    print(f"Epoch: {epoch+1}, Loss: {loss.item()}")
```

```
Epoch: 1, Loss: 8.67308521270752
Epoch: 2, Loss: 8.549959182739258
Epoch: 3, Loss: 8.477030754089355
```

Epoch: 4, Loss: 8.420647621154785
Epoch: 5, Loss: 8.357834815979004
Epoch: 6, Loss: 8.278790473937988
Epoch: 7, Loss: 8.195917129516602
Epoch: 8, Loss: 8.11185073852539
Epoch: 9, Loss: 8.032673835754395
Epoch: 10, Loss: 7.950951099395752
Epoch: 11, Loss: 7.86921501159668
Epoch: 12, Loss: 7.780338287353516
Epoch: 13, Loss: 7.696788311004639
Epoch: 14, Loss: 7.617959499359131
Epoch: 15, Loss: 7.527339458465576
Epoch: 16, Loss: 7.441591262817383
Epoch: 17, Loss: 7.355259895324707
Epoch: 18, Loss: 7.282275199890137
Epoch: 19, Loss: 7.202417373657227
Epoch: 20, Loss: 7.125113487243652
Epoch: 21, Loss: 7.036799430847168
Epoch: 22, Loss: 6.95314884185791
Epoch: 23, Loss: 6.883727550506592
Epoch: 24, Loss: 6.806533336639404
Epoch: 25, Loss: 6.734233379364014
Epoch: 26, Loss: 6.657886981964111
Epoch: 27, Loss: 6.583030700683594
Epoch: 28, Loss: 6.509462833404541
Epoch: 29, Loss: 6.436800479888916
Epoch: 30, Loss: 6.363864421844482
Epoch: 31, Loss: 6.2950921058654785
Epoch: 32, Loss: 6.232440948486328
Epoch: 33, Loss: 6.155529975891113
Epoch: 34, Loss: 6.095452785491943
Epoch: 35, Loss: 6.02095365524292
Epoch: 36, Loss: 5.961978912353516
Epoch: 37, Loss: 5.895207405090332
Epoch: 38, Loss: 5.838871479034424
Epoch: 39, Loss: 5.7874436378479
Epoch: 40, Loss: 5.714027404785156
Epoch: 41, Loss: 5.649787902832031
Epoch: 42, Loss: 5.5863471031188965
Epoch: 43, Loss: 5.529016494750977
Epoch: 44, Loss: 5.467198848724365
Epoch: 45, Loss: 5.408061981201172
Epoch: 46, Loss: 5.354319095611572
Epoch: 47, Loss: 5.289463996887207
Epoch: 48, Loss: 5.230424404144287
Epoch: 49, Loss: 5.171853542327881
Epoch: 50, Loss: 5.119553565979004
Epoch: 51, Loss: 5.058375835418701
Epoch: 52, Loss: 5.00487756729126
Epoch: 53, Loss: 4.946175575256348
Epoch: 54, Loss: 4.903557777404785
Epoch: 55, Loss: 4.841684818267822
Epoch: 56, Loss: 4.786761283874512

Epoch: 57, Loss: 4.735513687133789
Epoch: 58, Loss: 4.675220966339111
Epoch: 59, Loss: 4.624691009521484
Epoch: 60, Loss: 4.575934410095215
Epoch: 61, Loss: 4.525548458099365
Epoch: 62, Loss: 4.479837894439697
Epoch: 63, Loss: 4.429911136627197
Epoch: 64, Loss: 4.373790264129639
Epoch: 65, Loss: 4.3232808113098145
Epoch: 66, Loss: 4.275763988494873
Epoch: 67, Loss: 4.2176008224487305
Epoch: 68, Loss: 4.169626235961914
Epoch: 69, Loss: 4.1125006675720215

Output:

The model is trained using random sample data for 100 epochs, and the loss is printed after each epoch. The loss should gradually decrease over epochs, indicating that the model is learning to generate accurate predictions for the target sequen.