DEEP LEARNING LAB PRACTICALS

January 3, 2025

1 Practical 1: Implement a simple Perceptron

To implement a simple Perceptron, we can follow these steps:

1. Initialize the weights and bias.
2. Define the activation function (usually a step function for the Perceptron).
3. Implement the training process where weights are updated based on the error between the predicted output and the actual label.
4. Test the model after training.

[5]:
```python
import numpy as np

# Step 1: Define the Perceptron class
class Perceptron:
    def __init__(self, input_size, learning_rate=0.01, epochs=1000):
        self.input_size = input_size
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.weights = np.zeros(input_size)  # Initialize weights to zeros
        self.bias = 0  # Initialize bias to zero

    # Step 2: Define the activation function (Step function)
    def activation(self, x):
        return 1 if x >= 0 else 0  # Step function: outputs 1 if x >= 0, else 0

    # Step 3: Define the training function
    def train(self, X, y):
        for epoch in range(self.epochs):
            for i in range(len(X)):
                linear_output = np.dot(X[i], self.weights) + self.bias
                prediction = self.activation(linear_output)
                error = y[i] - prediction
                # Update weights and bias based on the error
                self.weights += self.learning_rate * error * X[i]
                self.bias += self.learning_rate * error

    # Step 4: Define the prediction function
    def predict(self, X):
        predictions = []
        for i in range(len(X)):
            linear_output = np.dot(X[i], self.weights) + self.bias
            prediction = self.activation(linear_output)
            predictions.append(prediction)
        return predictions

# Step 5: Example usage
if __name__ == "__main__":
    # Example OR gate data (input, expected output)
    # Input data (X) and labels (y)
    X = np.array([[0, 0],
                  [0, 1],
                  [1, 0],
                  [1, 1]])
    y = np.array([0, 1, 1, 1])  # OR gate outputs

    # Step 6: Create a Perceptron instance
    perceptron = Perceptron(input_size=2, learning_rate=0.1, epochs=10)

    # Train the model
    perceptron.train(X, y)

    # Test the trained model
    predictions = perceptron.predict(X)
    print("Predictions:", predictions)

    # Check the learned weights and bias
    print("Weights:", perceptron.weights)
    print("Bias:", perceptron.bias)
```

```
Predictions: [0, 1, 1, 1]
Weights: [0.1 0.1]
Bias: -0.1
```

---

2 Practical 2: Implement Digit Classification: Neural network to classify MNIST dataset

To implement a neural network for classifying the MNIST dataset (handwritten digits), we can use Python libraries like Keras (which is part of TensorFlow). This will simplify the process of building and training the model. Here's an implementation using a neural network with a simple architecture to classify MNIST digits.

2.0.1 Steps for Implementation:

1. Load the MNIST dataset.
2. Preprocess the data (normalize, reshape, etc.).
3. Build the neural network model.
4. Compile the model with an optimizer, loss function, and metrics.
5. Train the model on the MNIST dataset.
6. Evaluate the model on the test set.
7. Make predictions on new data.

Install Required Libraries

!pip install tensorflow

[1]:
```python
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# Step 1: Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Step 2: Preprocess the data
# Normalize the images to a [0, 1] range
x_train = x_train / 255.0
x_test = x_test / 255.0

# Flatten the images from 28x28 to 784-dimensional vectors
x_train = x_train.reshape(-1, 28*28)
x_test = x_test.reshape(-1, 28*28)

# One-hot encode the labels (convert to a binary matrix)
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Step 3: Build the Neural Network model
model = models.Sequential([
    layers.Dense(128, activation='relu', input_shape=(28*28,)),  # Hidden layer␣
 ↪with 128 neurons and ReLU activation
    layers.Dropout(0.2),  # Dropout layer to prevent overfitting
    layers.Dense(10, activation='softmax')  # Output layer with 10 neurons (one␣
 ↪for each digit)
])

# Step 4: Compile the model
model.compile(optimizer='adam',  # Adam optimizer
              loss='categorical_crossentropy',  # Loss function for multi-class␣
 ↪classification
              metrics=['accuracy'])  # Metric to track

# Step 5: Train the model
model.fit(x_train, y_train, epochs=5, batch_size=32, validation_data=(x_test,␣
 ↪y_test))

# Step 6: Evaluate the model on the test set
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc}")

# Step 7: Make predictions
predictions = model.predict(x_test)

# Let's print the first prediction
import numpy as np
print(f"First prediction: {np.argmax(predictions[0])}")
print(f"True label: {np.argmax(y_test[0])}")
```

```
Epoch 1/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3030 -
accuracy: 0.9121 - val_loss: 0.1391 - val_accuracy: 0.9576
Epoch 2/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.1467 -
accuracy: 0.9569 - val_loss: 0.0991 - val_accuracy: 0.9700
Epoch 3/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.1088 -
accuracy: 0.9672 - val_loss: 0.0861 - val_accuracy: 0.9737
Epoch 4/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0908 -
accuracy: 0.9724 - val_loss: 0.0780 - val_accuracy: 0.9766
Epoch 5/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0751 -
accuracy: 0.9768 - val_loss: 0.0765 - val_accuracy: 0.9756
313/313 [==============================] - 0s 1ms/step - loss: 0.0765 -
accuracy: 0.9756
Test accuracy: 0.975600004196167
313/313 [==============================] - 0s 1ms/step
First prediction: 7
True label: 7
```

---

3 Practical 3: Implement a simple CNN starting from filtering, Convolution and pooling operations and arithmetic of these with Visualization in PyTorch and Tensorflow

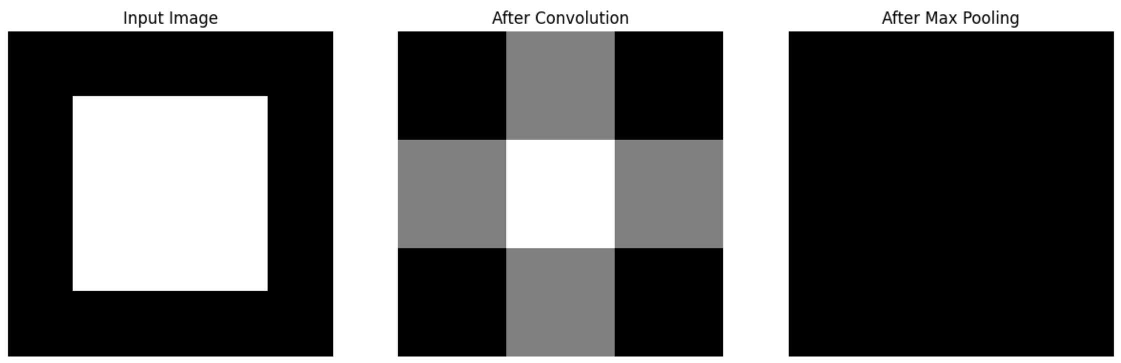
Implementing a simple Convolutional Neural Network (CNN) that demonstrates the filtering, convolution, and pooling operations, along with the arithmetic involved, in both PyTorch and TensorFlow, is a great way to understand how CNNs work. Below is a step-by-step approach to building this in both frameworks.

3.0.1 Step-by-Step Plan:

1. Creating Input Image: Start by creating a simple image (e.g., a small 5x5 image with one channel).
2. Applying Convolution: Use a convolutional filter to perform convolution on the image.
3. Pooling: Apply a pooling operation (e.g., Max Pooling).
4. Visualizing: Visualize the input, output after convolution, and output after pooling for both frameworks.

[4]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt

# Step 1: Create a simple 5x5 image (one channel)
image = torch.tensor([[0., 0., 0., 0., 0.],
                      [0., 1., 1., 1., 0.],
                      [0., 1., 1., 1., 0.],
                      [0., 1., 1., 1., 0.],
                      [0., 0., 0., 0., 0.]])
image = image.unsqueeze(0).unsqueeze(0)  # Add batch and channel dimensions

# Step 2: Define a simple filter (3x3 filter, one channel)
kernel = torch.tensor([[[[0., 1., 0.],
                         [1., -4., 1.],
                         [0., 1., 0.]]]])

# Step 3: Convolution operation (valid padding)
conv = F.conv2d(image, kernel, stride=1, padding=0)

# Step 4: Apply Max Pooling (2x2)
pooled = F.max_pool2d(conv, kernel_size=2, stride=2)

# Step 5: Visualize the input, convolution output, and pooled output
fig, ax = plt.subplots(1, 3, figsize=(15, 5))

# Convert tensor to numpy for visualization
ax[0].imshow(image.squeeze().numpy(), cmap='gray')
```

```
ax[0].set_title('Input Image')
ax[0].axis('off')

ax[1].imshow(conv.squeeze().detach().numpy(), cmap='gray')
ax[1].set_title('After Convolution')
ax[1].axis('off')

# Ensure the pooled output retains 2D shape for visualization
pooled_image = pooled.squeeze(0).squeeze(0)  # Remove batch and channel␣
 ↪dimensions
ax[2].imshow(pooled_image.detach().numpy(), cmap='gray')
ax[2].set_title('After Max Pooling')
ax[2].axis('off')

plt.show()
```



---

4 Practical 4: Implement Text processing, Language Modeling using RNN

Implementing text processing and language modeling using a Recurrent Neural Network (RNN) involves the following steps:

1. Text Preprocessing: Clean and tokenize the text data to prepare it for training.
2. Building the RNN Model: Define an RNN-based model for language modeling, which predicts the next word given the previous words in a sequence.
3. Training the Model: Train the model on the processed data.
4. Evaluating the Model: Check how well the model generates or predicts text sequences.

[10]:
```
import torch
import torch.nn as nn
```

```python
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import numpy as np

# Step 1: Prepare the Text Data and Tokenize
text = """
Python is an interpreted high-level general-purpose programming language.␣
 ↪Python's design philosophy emphasizes code readability with its notable use␣
 ↪of significant indentation. Its syntax allows programmers to express␣
 ↪concepts in fewer lines of code than would be possible in languages such as␣
 ↪C++ or Java.
Python is dynamically typed and garbage-collected. It supports multiple␣
 ↪programming paradigms, including structured (particularly procedural),␣
 ↪object-oriented, and functional programming.
"""

# Create a set of unique characters from the text
chars = sorted(set(text))
vocab_size = len(chars)

# Create mappings from characters to indices and vice versa
char_to_idx = {ch: idx for idx, ch in enumerate(chars)}
idx_to_char = {idx: ch for idx, ch in enumerate(chars)}

# Convert the text into a list of indices
text_as_int = [char_to_idx[ch] for ch in text]

# Step 2: Create Dataset for Sequence Prediction
class TextDataset(Dataset):
    def __init__(self, text_as_int, seq_length):
        self.text_as_int = text_as_int
        self.seq_length = seq_length

    def __len__(self):
        return len(self.text_as_int) - self.seq_length

    def __getitem__(self, idx):
        # Create input-output pairs
        input_seq = torch.tensor(self.text_as_int[idx:idx + self.seq_length])
        target_seq = torch.tensor(self.text_as_int[idx + 1:idx + self.
 ↪seq_length + 1])
        return input_seq, target_seq

seq_length = 30
dataset = TextDataset(text_as_int, seq_length)
dataloader = DataLoader(dataset, batch_size=64, shuffle=True)
```

```
# Step 3: Define the RNN Language Model
class RNNLanguageModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, seq_length):
        super(RNNLanguageModel, 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, vocab_size)

    def forward(self, x):
        # Embedding lookup for the input
        x = self.embedding(x)

        # RNN layer
        out, _ = self.rnn(x)

        # Output layer
        out = self.fc(out)
        return out

# Step 4: Initialize the Model, Loss, and Optimizer
embedding_dim = 128
hidden_dim = 256
model = RNNLanguageModel(vocab_size, embedding_dim, hidden_dim, seq_length)

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

# Step 5: Training the Model
def train_model(model, dataloader, criterion, optimizer, epochs=5):
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        for input_seq, target_seq in dataloader:
            optimizer.zero_grad()
            output = model(input_seq)

            # Flatten the outputs and targets
            output = output.view(-1, vocab_size)
            target_seq = target_seq.view(-1)

            loss = criterion(output, target_seq)
            loss.backward()
            optimizer.step()

            total_loss += loss.item()
        print(f"Epoch [{epoch+1}/{epochs}], Loss: {total_loss/len(dataloader)}")

# Train the model
train_model(model, dataloader, criterion, optimizer, epochs=10)

# Step 6: Evaluating the Model (Generate Text with Temperature Sampling)
def generate_text_with_temperature(model, start_text, length=200, temperature=1.
 ↪0):
    model.eval()

    input_seq = [char_to_idx[ch] for ch in start_text]
    input_seq = torch.tensor(input_seq).unsqueeze(0)  # Add batch dimension

    generated_text = start_text
    with torch.no_grad():
        for _ in range(length):
            output = model(input_seq)
            output = output.squeeze(0)  # Remove batch dimension

            # Apply temperature scaling to logits
            output = output / temperature

            # Softmax to get probabilities
            prob = torch.nn.functional.softmax(output[-1], dim=0)

            # Sample a character index based on the probability distribution
            predicted_idx = torch.multinomial(prob, 1).item()

            # Add predicted character to the generated text
            predicted_char = idx_to_char[predicted_idx]
            generated_text += predicted_char

            # Update input sequence with predicted character
            input_seq = torch.cat((input_seq[:, 1:], torch.
 ↪tensor([[predicted_idx]])), dim=1)

    return generated_text

# Example usage with temperature setting
start_text = "Python is"
generated = generate_text_with_temperature(model, start_text, length=300,␣
 ↪temperature=0.7)
print("\nGenerated Text:\n")
print(generated)
```

```
Epoch [1/10], Loss: 3.186570882797241
Epoch [2/10], Loss: 2.4967474043369293
Epoch [3/10], Loss: 2.1279394328594208
Epoch [4/10], Loss: 1.8134359568357468
Epoch [5/10], Loss: 1.537606194615364
Epoch [6/10], Loss: 1.2762676626443863
Epoch [7/10], Loss: 1.0253751650452614
Epoch [8/10], Loss: 0.8073843643069267
Epoch [9/10], Loss: 0.6257664263248444
Epoch [10/10], Loss: 0.49387771263718605

Generated Text:

Python is dynamically typed halizes of code than portaben le er lines ouage-
collected. It supports muledphigh-level general-purpose programming languages
such as C++ or Java.
Python is digh-lont indentation. Its such lewprogralaigns gunctitntedural), obje
ts in fevel geseduraliyntepress concepts in fecertein
```

---

5 Practical 5: Implement Time Series Prediction using RNN

To implement time series prediction using a Recurrent Neural Network (RNN) in PyTorch, we can follow these steps:

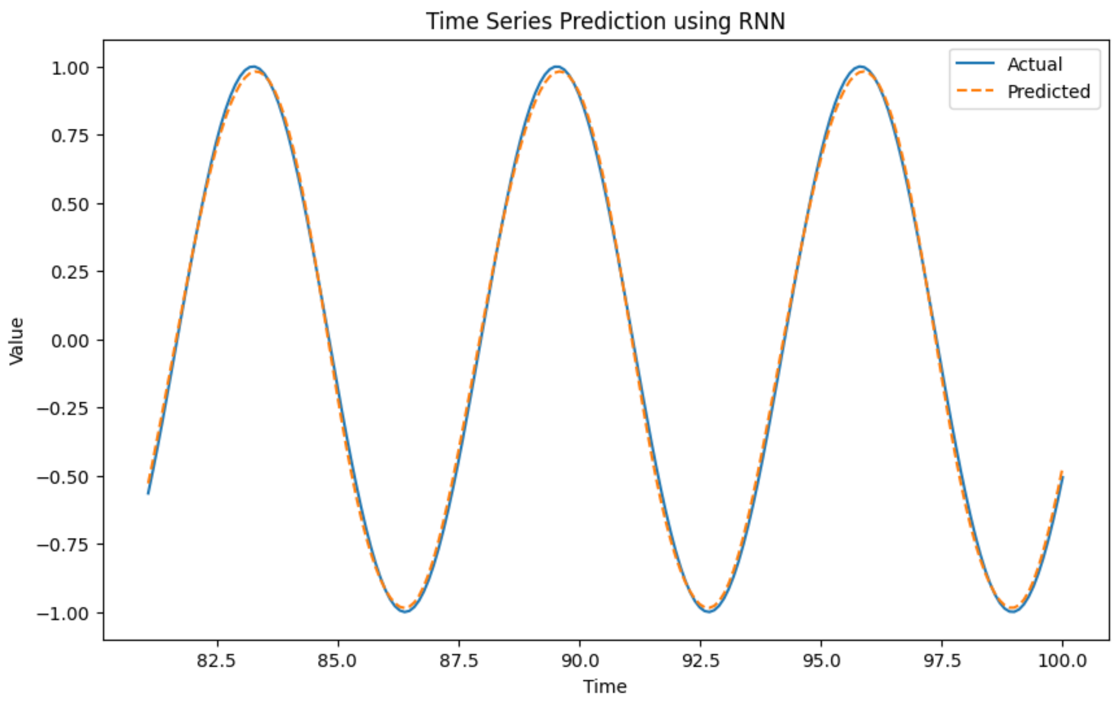
1. Data Preparation: Load and preprocess the time series data, including normalization and sequence generation.
2. Model Architecture: Define the RNN-based model for prediction.
3. Training: Train the model using the training dataset.
4. Evaluation: Use the model to predict future time steps.
5. Visualization: Plot the predicted values vs. actual values.

[12]:
```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler

# Step 1: Generate and Preprocess the Time Series Data
# Generate a sine wave as a time series
time = np.linspace(0, 100, 1000)
data = np.sin(time)

# Normalize the data to the range [0, 1] using MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))
data_normalized = scaler.fit_transform(data.reshape(-1, 1)).reshape(-1)
```

```python
# Convert data to a PyTorch tensor
data_tensor = torch.tensor(data_normalized, dtype=torch.float32)

# Create input-output sequences for training
def create_sequences(data, seq_length):
    input_seq = []
    output_seq = []
    for i in range(len(data) - seq_length):
        input_seq.append(data[i:i + seq_length])
        output_seq.append(data[i + seq_length])
    return torch.stack(input_seq), torch.stack(output_seq)

# Sequence length for RNN input
seq_length = 50

# Create training sequences
X, y = create_sequences(data_tensor, seq_length)

# Split into train and test sets
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Step 2: Define the RNN Model
class RNNTimeSeriesModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNNTimeSeriesModel, self).__init__()
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out, _ = self.rnn(x)
        out = self.fc(out[:, -1, :])  # Only use the last hidden state
        return out

# Model parameters
input_size = 1   # One feature (sine wave value)
hidden_size = 64
output_size = 1  # Predicting one value
learning_rate = 0.001

# Instantiate the model, loss function, and optimizer
model = RNNTimeSeriesModel(input_size, hidden_size, output_size)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

```python
# Step 3: Training the Model
def train_model(model, X_train, y_train, criterion, optimizer, epochs=100):
    model.train()
    for epoch in range(epochs):
        optimizer.zero_grad()
        output = model(X_train.unsqueeze(-1))  # Add feature dimension
        loss = criterion(output, y_train.unsqueeze(-1))
        loss.backward()
        optimizer.step()

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

# Train the model
train_model(model, X_train, y_train, criterion, optimizer, epochs=100)

# Step 4: Evaluating the Model (Prediction)
model.eval()
with torch.no_grad():
    # Predict on test data
    predictions = model(X_test.unsqueeze(-1))

# Inverse transform the predictions and actual values back to original scale
predictions = scaler.inverse_transform(predictions.numpy())
y_test_original = scaler.inverse_transform(y_test.numpy().reshape(-1, 1))

# Step 5: Visualization
plt.figure(figsize=(10, 6))
plt.plot(time[train_size + seq_length:], y_test_original, label="Actual")
plt.plot(time[train_size + seq_length:], predictions, label="Predicted",␣
 ↪linestyle='--')
plt.title('Time Series Prediction using RNN')
plt.xlabel('Time')
plt.ylabel('Value')
plt.legend()
plt.show()
```

```
Epoch [10/100], Loss: 0.0694
Epoch [20/100], Loss: 0.0469
Epoch [30/100], Loss: 0.0205
Epoch [40/100], Loss: 0.0118
Epoch [50/100], Loss: 0.0042
Epoch [60/100], Loss: 0.0024
Epoch [70/100], Loss: 0.0010
Epoch [80/100], Loss: 0.0004
Epoch [90/100], Loss: 0.0002
Epoch [100/100], Loss: 0.0002
```



---

6 Practical 6: Implement Sentiment Analysis using LSTM

Implementing sentiment analysis using an LSTM (Long Short-Term Memory) model in PyTorch involves the following steps:

1. Data Preprocessing: Tokenizing the text and converting it to numerical representations (like word embeddings).
2. Model Building: Defining the architecture using LSTM layers for sequential processing.
3. Training the Model: Training the model with the dataset and monitoring its performance.
4. Evaluation: Evaluating the model on unseen test data.
5. Prediction: Using the trained model to predict the sentiment of new text inputs.

[23]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from torch.utils.data import DataLoader, Dataset
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import re
import nltk
```

```
from nltk.tokenize import word_tokenize
from collections import Counter
from torch.nn.utils.rnn import pad_sequence

# Download punkt for tokenization
nltk.download('punkt')

# Step 1: Load and Preprocess the Data (IMDB Dataset)
# For simplicity, we'll use a small subset of the dataset. In real␣
 ↪applications, use the full dataset.
def preprocess_text(text):
    text = text.lower()
    text = re.sub(r'\W', ' ', text)  # Remove non-alphabetical characters
    text = re.sub(r'\s+', ' ', text)  # Remove multiple spaces
    return text

# Example dataset (simplified for demonstration)
positive_reviews = [
    "I loved this movie, it was fantastic!",
    "This is one of the best movies I've ever seen.",
    "Amazing performance by the lead actor. Great movie.",
    "Great direction and screenplay. Really enjoyed it!"
]

negative_reviews = [
    "The movie was boring and predictable.",
    "Waste of time. Terrible plot and bad acting.",
    "Not worth watching, I fell asleep halfway through.",
    "This movie is awful. Don't waste your time."
]

# Create labels
positive_labels = [1] * len(positive_reviews)  # Positive reviews labeled as 1
negative_labels = [0] * len(negative_reviews)  # Negative reviews labeled as 0

# Combine the reviews and labels
reviews = positive_reviews + negative_reviews
labels = positive_labels + negative_labels

# Step 2: Tokenize the text and create vocabulary
def tokenize_text(reviews):
    return [word_tokenize(preprocess_text(review)) for review in reviews]

tokenized_reviews = tokenize_text(reviews)

# Create vocabulary (mapping words to integers)
```

```
word_counter = Counter([word for review in tokenized_reviews for word in␣
↪review])
vocab = {word: idx + 1 for idx, (word, _) in enumerate(word_counter.
↪most_common())}  # Start indices from 1 (0 is reserved for padding)

# Step 3: Convert text data to numerical data (using the vocabulary)
def text_to_sequence(text, vocab):
    return [vocab.get(word, 0) for word in text]  # 0 for words not in␣
↪vocabulary

sequences = [text_to_sequence(review, vocab) for review in tokenized_reviews]

# Step 4: Padding sequences to ensure uniform input size
def pad_sequences(sequences):
    return pad_sequence([torch.tensor(seq) for seq in sequences],␣
↪batch_first=True, padding_value=0)

# Padding the sequences
padded_sequences = pad_sequences(sequences)

# Step 5: Create a custom dataset for PyTorch
class SentimentDataset(Dataset):
    def __init__(self, padded_sequences, labels):
        self.data = padded_sequences
        self.labels = torch.tensor(labels, dtype=torch.long)

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

    def __getitem__(self, idx):
        return self.data[idx], self.labels[idx]

# Create a DataLoader
dataset = SentimentDataset(padded_sequences, labels)
train_data, test_data = train_test_split(dataset, test_size=0.2,␣
↪random_state=42)
train_loader = DataLoader(train_data, batch_size=2, shuffle=True)
test_loader = DataLoader(test_data, batch_size=2, shuffle=False)

# Step 6: Build the LSTM Model
class SentimentLSTM(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim):
        super(SentimentLSTM, 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, output_dim)
        self.softmax = nn.Softmax(dim=1)
    def forward(self, x):
        embedded = self.embedding(x)
        lstm_out, (hidden, cell) = self.lstm(embedded)
        output = self.fc(hidden[-1])  # Use the last hidden state
        return self.softmax(output)

# Hyperparameters
vocab_size = len(vocab) + 1  # Add 1 for padding token
embedding_dim = 100
hidden_dim = 128
output_dim = 2  # Positive (1) or Negative (0)
learning_rate = 0.001

# Initialize model, loss function, and optimizer
model = SentimentLSTM(vocab_size, embedding_dim, hidden_dim, output_dim)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Step 7: Train the Model
def train(model, train_loader, criterion, optimizer, epochs=5):
    model.train()
    for epoch in range(epochs):
        epoch_loss = 0
        epoch_accuracy = 0
        for data, labels in train_loader:
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, labels)
            loss.backward()
            optimizer.step()

            epoch_loss += loss.item()
            preds = torch.argmax(output, dim=1)
            epoch_accuracy += accuracy_score(labels.numpy(), preds.numpy())

        print(f'Epoch [{epoch+1}/{epochs}], Loss: {epoch_loss/len(train_loader):
 ↪.4f}, Accuracy: {epoch_accuracy/len(train_loader):.4f}')

# Train the model
train(model, train_loader, criterion, optimizer, epochs=5)

# Step 8: Evaluate the Model
def evaluate(model, test_loader):
    model.eval()
    predictions = []
    labels = []
    with torch.no_grad():
        for data, target in test_loader:
            output = model(data)
            preds = torch.argmax(output, dim=1)
            predictions.extend(preds.numpy())
            labels.extend(target.numpy())

    accuracy = accuracy_score(labels, predictions)
    print(f'Test Accuracy: {accuracy:.4f}')

evaluate(model, test_loader)

# Step 9: Make Predictions
def predict_sentiment(model, text, vocab):
    tokenized = word_tokenize(preprocess_text(text))
    sequence = torch.tensor(text_to_sequence(tokenized, vocab)).unsqueeze(0)
    prediction = model(sequence)
    sentiment = torch.argmax(prediction, dim=1).item()
    return "Positive" if sentiment == 1 else "Negative"

# Test the prediction function
new_review = "This movie was awesome, I really enjoyed it!"
print(f"Predicted Sentiment: {predict_sentiment(model, new_review, vocab)}")

new_review = "This movie is awful. Don't waste your time for this."
print(f"Predicted Sentiment: {predict_sentiment(model, new_review, vocab)}")
```

```
Epoch [1/5], Loss: 0.6984, Accuracy: 0.5000
Epoch [2/5], Loss: 0.6780, Accuracy: 0.6667
Epoch [3/5], Loss: 0.6894, Accuracy: 0.5000
Epoch [4/5], Loss: 0.6574, Accuracy: 0.5000
Epoch [5/5], Loss: 0.6356, Accuracy: 0.8333
Test Accuracy: 0.5000
Predicted Sentiment: Positive
Predicted Sentiment: Negative
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\ramak\AppData\Roaming\nltk_data…
[nltk_data]   Package punkt is already up-to-date!
```

---

7 Practical 7: Implement an image generation model using GAN

Implementing an image generation model using a Generative Adversarial Network (GAN) involves training two networks: a Generator and a Discriminator. The Generator tries to generate realistic images from random noise, while the Discriminator tries to distinguish between real and fake images. The two networks are trained together, with the Generator improving to create more realistic images over time.

7.0.1 Steps:

1. Define the Generator Network: This network takes random noise as input and outputs an image.
2. Define the Discriminator Network: This network takes an image as input and outputs a probability of whether the image is real or fake.
3. Train the GAN: The Generator and Discriminator are trained together. The Generator tries to fool the Discriminator, while the Discriminator tries to correctly classify real and fake images.

[24]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

# Hyperparameters
batch_size = 64
latent_dim = 100
epochs = 50
lr = 0.0002
beta1 = 0.5

# Step 1: Prepare the dataset (MNIST in this case)
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5])  # Normalize to range [-1, 1]
])

# Load the MNIST dataset
dataset = datasets.MNIST(root='./data', train=True, download=True,␣
 ↪transform=transform)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# Step 2: Define the Generator Network
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(latent_dim, 256),
            nn.ReLU(True),
            nn.Linear(256, 512),
            nn.ReLU(True),
            nn.Linear(512, 1024),
            nn.ReLU(True),
            nn.Linear(1024, 28*28),
            nn.Tanh()  # Output the image in the range [-1, 1]
        )

    def forward(self, z):
        return self.fc(z).view(-1, 1, 28, 28)

# Step 3: Define the Discriminator Network
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(28*28, 1024),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 1),
            nn.Sigmoid()  # Output probability between 0 and 1
        )

    def forward(self, img):
        return self.fc(img.view(-1, 28*28))

# Step 4: Initialize the networks and optimizers
generator = Generator()
discriminator = Discriminator()

# Use binary cross-entropy loss
criterion = nn.BCELoss()

# Optimizers
optimizer_G = optim.Adam(generator.parameters(), lr=lr, betas=(beta1, 0.999))
optimizer_D = optim.Adam(discriminator.parameters(), lr=lr, betas=(beta1, 0.
 ↪999))

# Step 5: Training the GAN
real_labels = torch.ones(batch_size, 1)
fake_labels = torch.zeros(batch_size, 1)

# Function to save generated images during training
def save_generated_images(epoch, generator):
    z = torch.randn(16, latent_dim)  # Random noise
    generated_images = generator(z)
    generated_images = generated_images.detach().cpu().numpy()
    fig, axes = plt.subplots(4, 4, figsize=(4, 4))
    for i in range(4):
        for j in range(4):
            axes[i, j].imshow(generated_images[i*4 + j, 0], cmap='gray')
            axes[i, j].axis('off')
    plt.tight_layout()
    plt.savefig(f'gan_generated_images_epoch_{epoch}.png')

# Training loop
for epoch in range(epochs):
    for i, (imgs, _) in enumerate(dataloader):
        # Train the Discriminator
        real_imgs = imgs
        batch_size = real_imgs.size(0)

        # Real images
        optimizer_D.zero_grad()
        outputs = discriminator(real_imgs)
        d_loss_real = criterion(outputs, real_labels[:batch_size])
        d_loss_real.backward()

        # Fake images
        z = torch.randn(batch_size, latent_dim)
        fake_imgs = generator(z)
        outputs = discriminator(fake_imgs.detach())  # Detach to avoid updating␣
 ↪generator
        d_loss_fake = criterion(outputs, fake_labels[:batch_size])
        d_loss_fake.backward()

        optimizer_D.step()

        # Train the Generator
        optimizer_G.zero_grad()
        outputs = discriminator(fake_imgs)
        g_loss = criterion(outputs, real_labels[:batch_size])  # We want to␣
 ↪fool the discriminator
        g_loss.backward()

        optimizer_G.step()

    print(f"Epoch [{epoch+1}/{epochs}], D Loss: {d_loss_real.item() +␣
 ↪d_loss_fake.item()}, G Loss: {g_loss.item()}")

    # Save generated images every few epochs
    if (epoch + 1) % 10 == 0:
        save_generated_images(epoch + 1, generator)
```

```python
# Save final generator model
torch.save(generator.state_dict(), "generator.pth")
```

```
Epoch [1/50], D Loss: 0.5450341179966927, G Loss: 2.878868579864502
Epoch [2/50], D Loss: 0.18310445547103882, G Loss: 3.3271563053131104
Epoch [3/50], D Loss: 0.1819976568222046, G Loss: 3.868860960006714
Epoch [4/50], D Loss: 0.14792035520076752, G Loss: 3.6470236778259277
Epoch [5/50], D Loss: 0.13293325155973434, G Loss: 2.605674982070923
Epoch [6/50], D Loss: 0.00015677801945912506, G Loss: 8.693428039550781
Epoch [7/50], D Loss: 0.07591450400650501, G Loss: 5.403898239135742
Epoch [8/50], D Loss: 7.522566738771275e-05, G Loss: 9.57593059539795
Epoch [9/50], D Loss: 2.565526847320143e-05, G Loss: 10.601685523986816
Epoch [10/50], D Loss: 3.4475977370791895e-16, G Loss: 35.60368347167969
Epoch [11/50], D Loss: 3.6619301599590296e-16, G Loss: 35.543373107910156
Epoch [12/50], D Loss: 3.9381221930702393e-16, G Loss: 35.47065734863281
Epoch [13/50], D Loss: 4.2006105565838545e-16, G Loss: 35.40613555908203
Epoch [14/50], D Loss: 4.561077603386539e-16, G Loss: 35.32380294799805
Epoch [15/50], D Loss: 5.067992941000347e-16, G Loss: 35.218414306640625
Epoch [16/50], D Loss: 5.61294429311061e-16, G Loss: 35.11628723144531
Epoch [17/50], D Loss: 6.65764243712295e-16, G Loss: 34.945594787597656
Epoch [18/50], D Loss: 8.940543931343467e-16, G Loss: 34.65076446533203
Epoch [19/50], D Loss: 9.824003530494439e-16, G Loss: 34.55653381347656
Epoch [20/50], D Loss: 1.2399244152778792e-15, G Loss: 34.323726654052734
Epoch [21/50], D Loss: 0.0538494810461998, G Loss: 4.157371520996094
Epoch [22/50], D Loss: 0.0004930826885924944, G Loss: 7.643937110900879
Epoch [23/50], D Loss: 0.0015491027170355665, G Loss: 6.584824085235596
Epoch [24/50], D Loss: 4.7994718443078455e-05, G Loss: 10.041024208068848
Epoch [25/50], D Loss: 1.3537006453123723e-05, G Loss: 11.204964637756348
Epoch [26/50], D Loss: 4.6131285675876654e-06, G Loss: 12.293279647827148
Epoch [27/50], D Loss: 6.4334122598097565e-06, G Loss: 11.94644546508789
Epoch [28/50], D Loss: 1.108562344143138e-06, G Loss: 13.774178504943848
Epoch [29/50], D Loss: 0.0003794255171669647, G Loss: 9.251630783081055
Epoch [30/50], D Loss: 3.0258550850703614e-05, G Loss: 10.58187484741211
Epoch [31/50], D Loss: 8.560067499274737e-06, G Loss: 11.817144393920898
Epoch [32/50], D Loss: 4.574557664227541e-06, G Loss: 12.355972290039062
Epoch [33/50], D Loss: 2.409948578474541e-06, G Loss: 13.040305137634277
Epoch [34/50], D Loss: 4.589434560386962e-06, G Loss: 12.355203628540039
Epoch [35/50], D Loss: 3.1998293508195275e-06, G Loss: 12.74063777923584
Epoch [36/50], D Loss: 3.7595689228453466e-07, G Loss: 14.87509822845459
Epoch [37/50], D Loss: 100.0, G Loss: 0.0
Epoch [38/50], D Loss: 100.0, G Loss: 0.0
Epoch [39/50], D Loss: 100.0, G Loss: 0.0
Epoch [40/50], D Loss: 100.0, G Loss: 0.0
Epoch [41/50], D Loss: 100.0, G Loss: 0.0
Epoch [42/50], D Loss: 100.0, G Loss: 0.0
Epoch [43/50], D Loss: 100.0, G Loss: 0.0
Epoch [44/50], D Loss: 100.0, G Loss: 0.0
Epoch [45/50], D Loss: 100.0, G Loss: 0.0
Epoch [46/50], D Loss: 100.0, G Loss: 0.0
Epoch [47/50], D Loss: 100.0, G Loss: 0.0
Epoch [48/50], D Loss: 100.0, G Loss: 0.0
Epoch [49/50], D Loss: 100.0, G Loss: 0.0
Epoch [50/50], D Loss: 100.0, G Loss: 0.0
```