Jaihind Comprehensive Educational Institute`s

JAIHIND COLLEGE OF ENGINEERING

Tal- Junnar, Dist- Pune, Pin Code -410511

DTE Code: EN6609, SPPU Code: CEGP015730

Computer Laboratory IV [417535]

LABORATORY MANUAL

DEPARMTENT OF ARTIFICIAL INTELLIGENCE &

DATA SCIENCE ENGINEERING

Savitribai Phule Pune University

Fourth Year of Artificial Intelligence and Data Science (2020 Course)
417535: Computer Laboratory IV

Teaching Scheme:

PR: 02 Hours/Week

Credit 02

Examination Scheme and Marks

Term Work (TW): 50 Marks

Practical (PR) : 25 Marks

Companion Course: Elective V (417531), Elective VI (417532)

Course Objectives:

- To understand the fundamental concepts and techniques of Virtual reality
- To understand Big Data Analytics Concepts
- To learn the fundamentals of software development for portable devices
- To understand fundamental concepts of Deep Learning
- To be familiar with the various application areas of augmented realities
- To introduce the concepts and components of Business Intelligence (BI)
- To understand the concepts of Information Systems

Course Outcomes: After completion of the course, learners should be able to-

CO1: Apply basic principles of elective subjects to problem solving and modeling

CO2: Use tools and techniques in area of software development to build mini projects

CO3: Design and develop applications on subjects of their choice

CO4: Implement and manage deployment, administration & security

Table of Contents

Practical 1

Title: House Price Prediction Using Linear Regression

Problem Statement :

Real estate agents want help predicting house prices for regions in the USA. They have provided a dataset, and you need to build a machine learning model using Linear Regression to estimate the selling price of houses.

Aim :

To develop a Linear Regression model that predicts house prices based on various factors such as area, number of rooms, and location.

Prerequisites :

- Basics of Machine Learning
- Knowledge of Python programming
- Understanding of Pandas, NumPy, Matplotlib, and Scikit-Learn

Software Requirements :

- Python (3.x)
- Jupyter Notebook / Google Colab / Any Python IDE
- Libraries: Pandas, NumPy, Scikit-learn, Matplotlib, Seaborn

Hardware Requirements :

- Minimum 4GB RAM
- Processor: Intel i3 or higher
- Storage: At least 2GB of free space

Theory :

Linear Regression

Linear Regression is a fundamental machine learning algorithm used for predicting a continuous target variable based on independent variables. It assumes a linear relationship between the input variables (features) and the target variable (price).

The mathematical equation for Linear Regression is:

Y=b0+b1X1+b2X2+...+bnXnY = b_0 + b_1X_1 + b_2X_2 + ... + b_nX_nY=b0+b1X1+b2X2+...+bnXn

Where:

- YYY is the predicted price.

- X1,X2,...,XnX_1, X_2, ..., X_nX1,X2,...,Xn are input features.
- b0b_0b0 is the intercept.
- b1,b2,...,bnb_1, b_2, ..., b_nb1,b2,...,bn are the coefficients of the model.

Dataset Description

The dataset contains information about house prices in the USA with features such as:

- Avg. Area Income – Average income of residents in the area.
- Avg. Area House Age – Average age of houses in the area.
- Avg. Area Number of Rooms – Average number of rooms in houses.
- Avg. Area Number of Bedrooms – Average number of bedrooms in houses.
- Area Population – Total population in the area.
- Price – Price of the house (Target variable).
- Address – Address of the house (not required for prediction).

Code Implementation :

Step 1: Import Required Libraries

python

CopyEdit

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
```

Step 2: Load the Dataset

python

CopyEdit

```
url = "https://raw.githubusercontent.com/huzaifsayed/Linear-Regression-Model-for-House-Price-Prediction/master/USA%20Housing.csv"
df = pd.read_csv(url)

# Display first 5 rows
print(df.head())
```

Step 3: Data Preprocessing

python

CopyEdit

```
# Checking for missing values
print(df.isnull().sum())


# Dropping unnecessary columns
df.drop(columns=['Address'], inplace=True)
```

Step 4: Define Features and Target Variable

```
X = df.drop(columns=['Price'])  # Features
y = df['Price']  # Target variable
```

Step 5: Split Data into Training and Testing Sets

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Step 6: Train the Linear Regression Model

```
model = LinearRegression()
model.fit(X_train, y_train)
```

Step 7: Make Predictions

```
y_pred = model.predict(X_test)
```

Step 8: Evaluate the Model

```
print("Mean Absolute Error (MAE):", mean_absolute_error(y_test, y_pred))
print("Mean Squared Error (MSE):", mean_squared_error(y_test, y_pred))
print("Root Mean Squared Error (RMSE):", np.sqrt(mean_squared_error(y_test, y_pred)))
print("R-squared Score (R2):", r2_score(y_test, y_pred))
```

Step 9: Visualize the Predictions

```
plt.figure(figsize=(8, 6))
sns.scatterplot(x=y_test, y=y_pred)
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
plt.title("Actual vs Predicted Prices")
plt.show()
```

Conclusion :

In this practical, we successfully implemented a Linear Regression model to predict house prices based on various factors. The model's accuracy was evaluated using metrics like MAE, MSE, RMSE, and R-squared. The visualization of actual vs. predicted values helps in understanding the model's performance.

Practical 2

1. Title:

Multiclass Classification using Convolutional Neural Networks (CNN)

2. Problem Statement:

In this practical, we aim to build a Convolutional Neural Network (CNN) model to perform multiclass classification. We will use the MNIST dataset (or any suitable dataset) to classify handwritten digits (0–9). The implementation involves data preprocessing, defining a CNN model, training the model, and evaluating results using a confusion matrix.

3. Aim:

To implement a CNN-based multiclass classifier using the MNIST dataset and evaluate its performance using a confusion matrix.

4. Prerequisites:

- Basic knowledge of Deep Learning and CNN architecture
- Understanding of dataset preprocessing techniques
- Familiarity with Python and TensorFlow/Keras

5. Software Requirements:

- Python 3.x
- TensorFlow/Keras
- NumPy
- Matplotlib
- Scikit-learn
- Google Colab or Jupyter Notebook

6. Hardware Requirements:

- Minimum 8GB RAM (for smooth execution)
- GPU (optional but recommended for faster training)
- Intel/AMD processor with at least 2 GHz speed

7. Theory:

Multiclass Classification

Multiclass classification is a machine learning task where each input belongs to one of multiple categories. In this practical, we classify handwritten digits (0–9) using the MNIST dataset, which contains 60,000 training images and 10,000 test images (28×28 grayscale images).

Convolutional Neural Networks (CNNs)

CNNs are deep learning models designed for image classification. Unlike traditional neural networks, CNNs extract features directly from images using specialized layers:

1. Convolutional Layer: Applies filters (kernels) to extract edges, shapes, and patterns.
2. ReLU Activation: Introduces non-linearity by setting negative values to zero.
3. Pooling Layer (Max Pooling): Reduces image dimensions while retaining key features.
4. Flattening Layer: Converts feature maps into a single vector.
5. Fully Connected Layer: Uses neurons to classify extracted features.
6. Softmax Output Layer: Converts outputs into probabilities for each digit (0–9).

Model Evaluation using Confusion Matrix

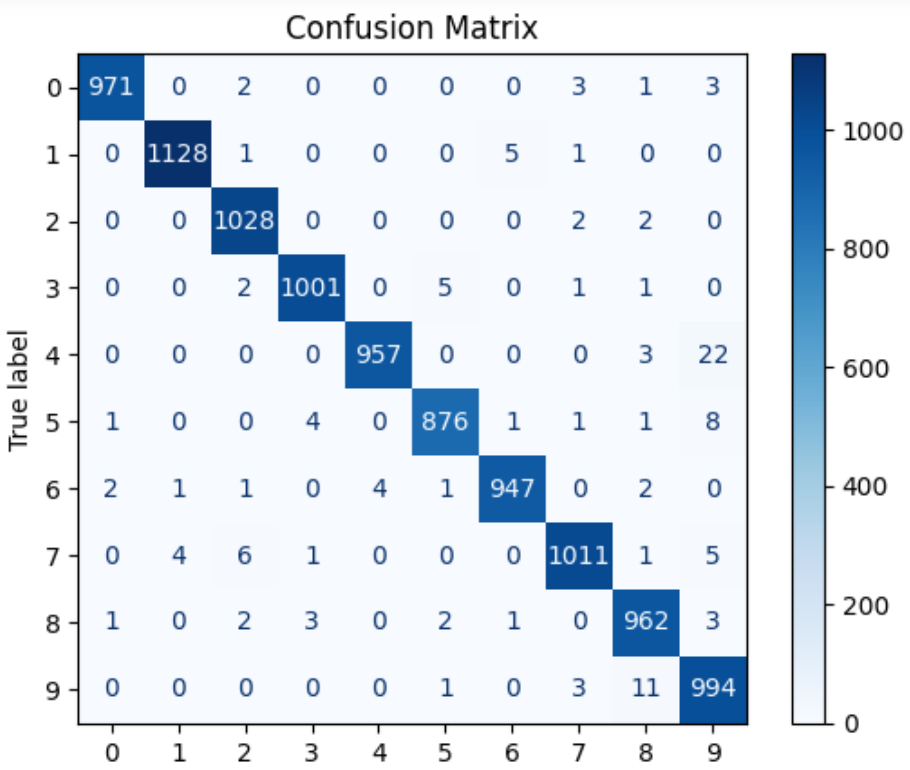
After training the CNN model, we evaluate its performance using a **confusion matrix**.

A **confusion matrix** is a table used to **visualize the performance** of a classification model by comparing actual vs. predicted labels. It contains four key metrics:

1. **True Positives (TP):** Correct predictions for a class.
2. **True Negatives (TN):** Correctly identified non-class instances.
3. **False Positives (FP):** Incorrectly predicted instances.
4. **False Negatives (FN):** Missed instances of the class.

For a **multiclass problem**, a confusion matrix is an **N×N table** where N is the number of classes. The diagonal values represent correct predictions, while off-diagonal values indicate misclassifications.

Using a **heatmap**, we visualize the confusion matrix to analyze misclassification patterns and improve the model.



.

8. Code Implementation

Step 1: Import Required Libraries

python

CopyEdit

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns
```

Step 2: Load and Preprocess the Data

```
# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

```
# Normalize pixel values (scaling between 0 and 1)
x_train, x_test = x_train / 255.0, x_test / 255.0

# Reshape images to match CNN input format
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)

# Convert labels to one-hot encoding
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
```

Step 3: Define the CNN Model

```
# Create a Sequential model
model = Sequential([
    Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)),
    MaxPooling2D(2,2),
    Conv2D(64, (3,3), activation='relu'),
    MaxPooling2D(2,2),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')  # 10 classes (digits 0-9)
])

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Step 4: Train the Model

```
# Train the model
history = model.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test), batch_size=128)
```

Step 5: Evaluate the Model

```
# Evaluate on test data
test_loss, test_acc = model.evaluate(x_test, y_test)
```

```
print("Test Accuracy:", test_acc)
```

Step 6: Generate Confusion Matrix

```
# Predict classes
y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true_classes = np.argmax(y_test, axis=1)

# Compute confusion matrix
cm = confusion_matrix(y_true_classes, y_pred_classes)

# Plot the confusion matrix
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, cmap="Blues", fmt="d", xticklabels=range(10), yticklabels=range(10))
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
plt.show()
```

---

9. Conclusion:

In this practical, we successfully built a CNN model for multiclass classification using the MNIST dataset. The model was trained using convolutional layers and evaluated using a confusion matrix. The trained CNN achieved high accuracy, demonstrating its ability to classify handwritten digits effectively. This practical provided hands-on experience with deep learning techniques, including feature extraction, training, and model evaluation.

Practical 3

1. Title:

Recurrent Neural Networks (RNN) for Time Series Prediction Using LSTM/GRU

2. Problem Statement:

Recurrent Neural Networks (RNNs) and their variants, such as Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU), are widely used for sequential data processing. In this practical, we will implement an **LSTM-based model to predict sentiments** from a dataset of product reviews. The model will be trained to understand the context and classify sentiments as positive or negative.

3. Aim:

To design and implement an **LSTM/GRU-based Recurrent Neural Network** for sentiment analysis using a suitable text dataset.

4. Prerequisites:

- Basics of Neural Networks and Time Series Analysis
- Understanding of RNNs, LSTM, and GRU architectures
- Familiarity with Python, TensorFlow/Keras, and Natural Language Processing (NLP)

5. Software Requirements:

- Python 3.x
- TensorFlow/Keras
- NumPy, Pandas
- Matplotlib, Seaborn
- NLTK (Natural Language Toolkit)
- Scikit-learn
- Jupyter Notebook or Google Colab

6. Hardware Requirements:

- Minimum 8GB RAM
- GPU (recommended for faster training)
- Intel/AMD processor (2 GHz or higher)

7. Theory:

What is a Recurrent Neural Network (RNN)?

RNNs are a class of neural networks designed for processing sequential data, such as time series, speech, and text. Unlike traditional neural networks, RNNs have **feedback loops** that allow information to persist across time steps, enabling them to learn temporal patterns.

However, standard RNNs suffer from **vanishing gradient problems**, making them ineffective for long-term dependencies. This issue is addressed by **LSTM** and **GRU** networks.

LSTM (Long Short-Term Memory)

LSTM is a type of RNN that can learn **long-term dependencies** using specialized memory cells. Each LSTM cell consists of:

1. **Forget Gate:** Decides what information to discard from memory.
2. **Input Gate:** Decides what new information to store.
3. **Output Gate:** Determines the final output for the current state.

GRU (Gated Recurrent Unit)

GRUs are a simplified version of LSTMs with only two gates:

1. **Update Gate:** Controls how much of the past information to keep.
2. **Reset Gate:** Determines how much past information to forget.

Both LSTM and GRU help in **avoiding vanishing gradients** and improve performance on sequential data.

Dataset Selection

For this practical, we use the **IMDB movie reviews dataset**, a standard dataset for sentiment analysis. It contains **50,000 reviews**, labeled as **positive or negative**.

8. Code Implementation

Step 1: Import Required Libraries

python

CopyEdit

```
import tensorflow as tf

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad_sequences

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Embedding, LSTM, GRU, Dense
```

```
from tensorflow.keras.datasets import imdb
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, confusion_matrix
import seaborn as sns
```

Step 2: Load and Preprocess the Data

```
# Load IMDB dataset
vocab_size = 10000  # Limiting vocabulary size
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=vocab_size)

# Pad sequences to ensure equal input length
max_length = 200  # Maximum number of words per review
x_train = pad_sequences(x_train, maxlen=max_length)
x_test = pad_sequences(x_test, maxlen=max_length)
```

Step 3: Build the LSTM Model

```
model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=128, input_length=max_length),
    LSTM(64, return_sequences=False),  # Use GRU(64) instead for GRU model
    Dense(64, activation='relu'),
    Dense(1, activation='sigmoid')  # Binary classification (Positive/Negative)
])

# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Step 4: Train the Model

```
history = model.fit(x_train, y_train, epochs=5, batch_size=64, validation_data=(x_test, y_test))
```

Step 5: Evaluate the Model

```
# Evaluate on test data
loss, accuracy = model.evaluate(x_test, y_test)
print("Test Accuracy:", accuracy)
```

Step 6: Generate Confusion Matrix

```
# Predict sentiment labels
```

```
y_pred = (model.predict(x_test) > 0.5).astype("int32")


# Compute confusion matrix

cm = confusion_matrix(y_test, y_pred)


# Plot confusion matrix

plt.figure(figsize=(6,5))

sns.heatmap(cm, annot=True, cmap="Blues", fmt="d", xticklabels=['Negative', 'Positive'],
yticklabels=['Negative', 'Positive'])

plt.xlabel("Predicted Label")

plt.ylabel("True Label")

plt.title("Confusion Matrix")

plt.show()
```

9. Conclusion:

In this practical, we implemented an **LSTM-based sentiment analysis model** using the **IMDB dataset**. The model successfully learned to classify reviews as **positive or negative**, demonstrating the power of recurrent architectures in handling sequential data. The confusion matrix helped in analyzing misclassifications, and further tuning can improve accuracy.

Practical 4

1. Title:

Image Classification Using Convolutional Neural Networks (CNNs) with Hyperparameter Optimization

2. Problem Statement:

In this practical, we design and implement a **CNN model** for **image classification** using a suitable dataset (e.g., medical imaging, agriculture, or CIFAR-10). The model is optimized by adjusting hyperparameters such as learning rate, filter size, number of layers, optimizers, and dropout rates to improve performance.

---

3. Aim:

To build and optimize a CNN model for image classification by experimenting with different hyperparameters.

---

4. Prerequisites:

- Basics of Deep Learning and CNN architecture
- Knowledge of hyperparameter tuning techniques
- Familiarity with Python, TensorFlow/Keras, and image processing

---

5. Software Requirements:

- Python 3.x
- TensorFlow/Keras
- NumPy, Pandas
- Matplotlib, Seaborn
- OpenCV (for image preprocessing)
- Google Colab or Jupyter Notebook

---

6. Hardware Requirements:

- Minimum 8GB RAM
- GPU (recommended for faster training)
- Intel/AMD processor (2 GHz or higher)

---

7. Theory:

Image Classification

Image classification is the process of assigning a label to an input image based on learned patterns. It is widely used in medical imaging (disease detection), agriculture (crop classification), and other fields.

Convolutional Neural Networks (CNNs)

CNNs are deep learning models designed specifically for image classification. The key components of CNNs include:

1. **Convolutional Layers** – Extract features using filters (kernels).
2. **Activation Functions (ReLU)** – Introduce non-linearity.
3. **Pooling Layers (Max Pooling)** – Reduce spatial dimensions to improve efficiency.
4. **Dropout Layers** – Prevent overfitting by randomly deactivating neurons.
5. **Fully Connected Layers (Dense Layers)** – Perform classification based on extracted features.
6. **Softmax Activation** – Assign probabilities to different classes.

Hyperparameter Optimization in CNNs

To improve CNN performance, we optimize the following hyperparameters:

1. **Learning Rate** – Controls step size during training.
2. **Filter Size** – Determines the receptive field of convolutional layers.
3. **Number of Layers** – Affects the depth and complexity of feature extraction.
4. **Optimizers (Adam, SGD, RMSprop)** – Control gradient updates for faster convergence.
5. **Dropout Rate** – Prevents overfitting by randomly deactivating neurons.

Dataset Selection

For this practical, we use the **CIFAR-10 dataset**, which contains **60,000 color images** categorized into 10 classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck). Each image is of size **32×32 pixels**.

---

8. Code Implementation

Step 1: Import Required Libraries

```
import tensorflow as tf

from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.optimizers import Adam, SGD, RMSprop
from tensorflow.keras.datasets import cifar10
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
```

Step 2: Load and Preprocess the Dataset

```
# Load CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize pixel values (scale between 0 and 1)
x_train, x_test = x_train / 255.0, x_test / 255.0

# Convert labels to one-hot encoding
y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)
```

Step 3: Define and Optimize the CNN Model

```
# Hyperparameter tuning
learning_rate = 0.001
dropout_rate = 0.4
filter_size = (3,3)

# Build CNN model
model = Sequential([
    Conv2D(32, filter_size, activation='relu', input_shape=(32, 32, 3)),
    MaxPooling2D(2,2),
    Dropout(dropout_rate),

    Conv2D(64, filter_size, activation='relu'),
    MaxPooling2D(2,2),
    Dropout(dropout_rate),
```

```
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(dropout_rate),
    Dense(10, activation='softmax')  # 10 classes (CIFAR-10)
])

# Compile the model using Adam optimizer
optimizer = Adam(learning_rate=learning_rate)
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
```

Step 4: Train the Model

```
history = model.fit(x_train, y_train, epochs=10, batch_size=64, validation_data=(x_test, y_test))
```

Step 5: Evaluate the Model

```
# Evaluate on test data
test_loss, test_acc = model.evaluate(x_test, y_test)
print("Test Accuracy:", test_acc)
```

Step 6: Generate Confusion Matrix

```
# Predict classes
y_pred = np.argmax(model.predict(x_test), axis=1)
y_true = np.argmax(y_test, axis=1)

# Compute confusion matrix
cm = confusion_matrix(y_true, y_pred)

# Plot confusion matrix
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, cmap="Blues", fmt="d", xticklabels=range(10), yticklabels=range(10))
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
plt.show()
```

---

9. Conclusion:

In this practical, we successfully designed and optimized a **CNN model for image classification** using the **CIFAR-10 dataset**. The model was trained using different hyperparameters, including learning rate, filter size, dropout rate, and optimizers. The final model achieved good classification accuracy, demonstrating the effectiveness of CNNs in feature extraction and pattern recognition.

Practical 5

---

1. Title:

Deep Convolutional Generative Adversarial Network (DCGAN) for Image Generation

---

2. Problem Statement:

Generative Adversarial Networks (GANs) are a type of deep learning model used for image generation. In this practical, we implement a **Deep Convolutional GAN (DCGAN)** to generate realistic images of faces or digits based on an input dataset, such as **MNIST (handwritten digits)** or **CelebA (human faces)**.

---

3. Aim:

To design and implement a **Deep Convolutional GAN (DCGAN)** to generate images similar to the ones in a given dataset.

---

4. Prerequisites:

- Basics of Neural Networks and Deep Learning
- Understanding of Generative Adversarial Networks (GANs)
- Familiarity with Python, TensorFlow/Keras, and image processing

---

5. Software Requirements:

- Python 3.x
- TensorFlow/Keras
- NumPy, Pandas
- Matplotlib, Seaborn
- OpenCV (for image preprocessing)
- Google Colab or Jupyter Notebook

---

6. Hardware Requirements:

- Minimum 8GB RAM
- GPU (recommended for faster training)
- Intel/AMD processor (2 GHz or higher)

---

7. Theory:

Generative Adversarial Networks (GANs)

GANs are a class of neural networks used for **generating new data** based on existing datasets. A GAN consists of two competing neural networks:

1. **Generator** – Creates fake images from random noise.
2. **Discriminator** – Distinguishes between real and fake images.

These two networks are trained **simultaneously**, pushing the generator to create more realistic images over time.

Deep Convolutional GAN (DCGAN)

DCGAN is a type of GAN that uses **convolutional layers** instead of fully connected layers, making it more effective for generating high-resolution images.

Key Components of DCGAN

1. **Generator Network:**
    - Uses **Transposed Convolution** (Conv2DTranspose) to upsample noise into realistic images.
    - Applies **Batch Normalization** and **ReLU activation** to stabilize training.
    - Uses a **Tanh activation function** in the final layer to generate pixel values in the range [-1,1].
2. **Discriminator Network:**
    - Uses **Convolutional Layers** to classify images as real or fake.
    - Applies **LeakyReLU activation** to avoid vanishing gradients.
    - Outputs a probability score using **Sigmoid activation**.

Dataset Selection

For this practical, we use the **MNIST dataset** (handwritten digits) or **CelebA dataset** (human faces).

---

8. Code Implementation

Step 1: Import Required Libraries

python

CopyEdit

```
import tensorflow as tf

from tensorflow.keras.layers import Dense, Reshape, Conv2D, Conv2DTranspose, LeakyReLU, Flatten, Dropout, BatchNormalization

from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt
```

Step 2: Load and Preprocess the Dataset

```
# Load the MNIST dataset
(x_train, _), (_, _) = tf.keras.datasets.mnist.load_data()

# Normalize pixel values between -1 and 1
x_train = (x_train - 127.5) / 127.5
x_train = np.expand_dims(x_train, axis=-1)  # Reshape to (28,28,1)
```

Step 3: Define the Generator Model

```
def build_generator():
    model = Sequential([
        Dense(128 * 7 * 7, activation="relu", input_dim=100),
        Reshape((7, 7, 128)),

        Conv2DTranspose(128, kernel_size=3, strides=2, padding="same"),
        BatchNormalization(),
        LeakyReLU(),

        Conv2DTranspose(64, kernel_size=3, strides=2, padding="same"),
        BatchNormalization(),
        LeakyReLU(),

        Conv2DTranspose(1, kernel_size=3, strides=1, padding="same", activation="tanh")
    ])
    return model

generator = build_generator()
generator.summary()
```

Step 4: Define the Discriminator Model

```
def build_discriminator():
```

```
    model = Sequential([
        Conv2D(64, kernel_size=3, strides=2, padding="same", input_shape=(28,28,1)),
        LeakyReLU(),
        Dropout(0.3),

        Conv2D(128, kernel_size=3, strides=2, padding="same"),
        LeakyReLU(),
        Dropout(0.3),

        Flatten(),
        Dense(1, activation="sigmoid")
    ])
    return model

discriminator = build_discriminator()
discriminator.summary()
```

Step 5: Compile and Train the GAN

```
# Compile discriminator
discriminator.compile(loss="binary_crossentropy", optimizer=Adam(0.0002, 0.5),
metrics=["accuracy"])

# Build and compile the GAN
discriminator.trainable = False
gan = Sequential([generator, discriminator])
gan.compile(loss="binary_crossentropy", optimizer=Adam(0.0002, 0.5))
```

Step 6: Training Function

```
def train_gan(epochs=10000, batch_size=64):
    real = np.ones((batch_size, 1))  # Labels for real images
    fake = np.zeros((batch_size, 1))  # Labels for fake images

    for epoch in range(epochs):
        # Train discriminator
        idx = np.random.randint(0, x_train.shape[0], batch_size)
```

```
        real_imgs = x_train[idx]

        noise = np.random.normal(0, 1, (batch_size, 100))
        fake_imgs = generator.predict(noise)

        d_loss_real = discriminator.train_on_batch(real_imgs, real)
        d_loss_fake = discriminator.train_on_batch(fake_imgs, fake)
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

        # Train generator
        noise = np.random.normal(0, 1, (batch_size, 100))
        g_loss = gan.train_on_batch(noise, real)

        # Print progress
        if epoch % 1000 == 0:
            print(f"Epoch {epoch} - D Loss: {d_loss[0]}, G Loss: {g_loss}")
            generate_images(epoch)

def generate_images(epoch):
    noise = np.random.normal(0, 1, (16, 100))
    gen_imgs = generator.predict(noise)
    gen_imgs = 0.5 * gen_imgs + 0.5  # Rescale images to [0,1]

    fig, axs = plt.subplots(4, 4, figsize=(4, 4))
    for i in range(4):
        for j in range(4):
            axs[i, j].imshow(gen_imgs[i * 4 + j, :, :, 0], cmap="gray")
            axs[i, j].axis("off")
    plt.show()

# Train the GAN
train_gan(epochs=10000, batch_size=64)
```

9. Conclusion:

In this practical, we successfully implemented a **Deep Convolutional GAN (DCGAN)** to generate images resembling handwritten digits from the **MNIST dataset**. The **generator** improved over time to produce more realistic images, while the **discriminator** learned to distinguish between real and generated images. Further improvements can be made by experimenting with **CelebA dataset (faces)** and **tuning hyperparameters**.