DL_Practical 1

```
# Importing libraries
import numpy as np           # Numerical operations
import pandas as pd          # Handling datasets / tables
import matplotlib.pyplot as plt # To plot graphs
import tensorflow as tf      # Deep Learning
import torch                 # Deep Learning (PyTorch)
import theano
import theano.tensor as T    # Symbolic math in Theano

# 1) NumPy Example: Create array and add numbers
arr = np.array([1, 2, 3])
print("NumPy Array:", arr)

# 2) Pandas Example: Create small table
data = pd.DataFrame({"Name": ["A", "B"], "Marks": [90, 85]})
print("\nPandas DataFrame:\n", data)

# 3) Matplotlib Example: Simple Plot
plt.plot([1, 2, 3], [2, 4, 6])
plt.title("Simple Line Plot")
plt.show()

# 4) TensorFlow Example: Add two constants
x = tf.constant(5)
y = tf.constant(7)
print("\nTensorFlow Result:", x + y)
```

```
# 5) PyTorch Example: Add two tensors
a = torch.tensor(5)
b = torch.tensor(7)
print("PyTorch Result:", a + b)

# 6) Theano Example: Simple addition
p = T.scalar('p')
q = T.scalar('q')
r = p + q
f = theano.function(inputs=[p, q], outputs=r)
print("Theano Result:", f(4, 6))
```

Library	**Use in Deep Learning (Simple Explanation)**
TensorFlow	Used to **build and train neural networks**. Supports **GPU acceleration** and is widely used in **production and large-scale applications**.
Keras	A **simple high-level interface** built on top of TensorFlow. Makes it **easy to design and train models** with fewer lines of code. Best for **beginners and rapid prototyping**.
Theano	Used for **mathematical computation** involving **multi-dimensional arrays**. Earlier used for deep learning research but now mostly **outdated**.
PyTorch	Used for **flexible and dynamic deep learning model development**. Very popular in **research and academic projects** due to its **easy debugging and simple syntax**

Practical 2:

```
# Step 1: Import Necessary Libraries
import tensorflow as tf                    # Deep Learning framework
from tensorflow.keras import datasets, models    # To load dataset and build model
from tensorflow.keras.layers import Dense, Flatten  # Layers for Neural Network
import matplotlib.pyplot as plt              # To plot training graphs

# Step 2: Load MNIST Dataset (Handwritten Digits)
(x_train, y_train), (x_test, y_test) = datasets.mnist.load_data()

# Normalize pixel values (0 to 1 scale) for better training
x_train = x_train / 255.0
x_test = x_test / 255.0

# Step 3: Define Feedforward Neural Network Architecture
model = models.Sequential([
    Flatten(input_shape=(28, 28)),      # Convert 28x28 image to 1D vector
    Dense(64, activation='relu'),       # Hidden Layer with ReLU activation
    Dense(10, activation='softmax')     # Output Layer (10 digits)
])

# Display model summary
```

```
model.summary()

# Step 4: Compile Model with Optimizer (SGD), Loss Function & Accuracy Metric
model.compile(optimizer='sgd',
          loss='sparse_categorical_crossentropy',
          metrics=['accuracy'])

# Step 5: Train the Model for 10 Epochs
history = model.fit(x_train, y_train, epochs=10, validation_split=0.1)

# Step 6: Evaluate Model Performance on Test Dataset
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print("Test Accuracy:", test_accuracy)

# Step 7: Plot Training Accuracy and Loss
plt.figure(figsize=(12, 5))

# Accuracy Plot
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label="Train Accuracy")
plt.plot(history.history['val_accuracy'], label="Validation Accuracy")
plt.title("Model Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
```

```
# Loss Plot
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label="Train Loss")
plt.plot(history.history['val_loss'], label="Validation Loss")
plt.title("Model Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()

plt.show()
```

Practical 3

```
# --------------------------------------
# 1. LOADING & PREPROCESSING IMAGE DATA
# --------------------------------------

import tensorflow as tf                    # Deep Learning framework
from tensorflow.keras.datasets import mnist      # MNIST Dataset
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
import matplotlib.pyplot as plt              # For plotting accuracy & loss

# Load Dataset: Split into Train & Test
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Normalize pixel values (convert range from 0–255 to 0–1)
X_train = X_train / 255.0
X_test = X_test / 255.0

# Reshape data to add channel dimension (grayscale = 1 channel)
X_train = X_train.reshape(-1, 28, 28, 1)
X_test = X_test.reshape(-1, 28, 28, 1)

# Convert labels to one-hot encoded format
y_train = to_categorical(y_train, 10)
```

```
y_test = to_categorical(y_test, 10)

# --------------------------------------
# 2. DEFINING MODEL ARCHITECTURE (CNN)
# --------------------------------------

model = Sequential([
   Conv2D(32, (3,3), activation="relu", input_shape=(28,28,1)), # Extract
Features
   MaxPooling2D(pool_size=(2,2)),                    # Reduce size
   Flatten(),                                  # Convert to 1D
   Dense(100, activation="relu"),                    # Fully Connected Layer
   Dense(10, activation="softmax")                    # Output Layer (10 digits)
])

# Compile Model (Optimizer, Loss & Evaluation Metric)
model.compile(optimizer="adam",
          loss="categorical_crossentropy",
          metrics=["accuracy"])

# Show model structure
model.summary()

# --------------------------------------
# 3. TRAINING THE MODEL
# --------------------------------------
```

```
history = model.fit(X_train, y_train, epochs=10, batch_size=32,
validation_split=0.1)


# --------------------------------------
# 4. EVALUATING MODEL PERFORMANCE
# --------------------------------------


test_loss, test_accuracy = model.evaluate(X_test, y_test)
print("\nTest Accuracy:", test_accuracy)


# Plot Accuracy & Loss
plt.figure(figsize=(12,5))


# Accuracy Plot
plt.subplot(1,2,1)
plt.plot(history.history['accuracy'], label="Train Accuracy")
plt.plot(history.history['val_accuracy'], label="Validation Accuracy")
plt.title("Model Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()


# Loss Plot
plt.subplot(1,2,2)
plt.plot(history.history['loss'], label="Train Loss")
plt.plot(history.history['val_loss'], label="Validation Loss")
plt.title("Model Loss")
```

```
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()

plt.show()
```

Practical 4

```
# Import necessary libraries
import pandas as pd  # For loading and handling dataset
import numpy as np  # For numerical operations
import tensorflow as tf  # For building neural networks
import matplotlib.pyplot as plt  # For plotting graphs
from sklearn.model_selection import train_test_split  # For splitting data into train and test
from sklearn.preprocessing import StandardScaler  # For normalizing features
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report  # For evaluating model

# Load the dataset
data = pd.read_csv("E:\dl_practicals\creditcard.csv")

# Check dataset info
print("Dataset shape:", data.shape)
print("Class distribution:")
print(data['Class'].value_counts())
print("0 = Normal transaction, 1 = Fraud transaction")

# Separate features (X) and target (y)
X = data.drop('Class', axis=1)  # All columns except 'Class' are features
y = data['Class']  # 'Class' column is the target

# Normalize the feature values
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split dataset into training and testing sets
```

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

# Convert to numpy arrays before filtering
X_train_np = X_train
X_test_np = X_test
y_train_np = y_train.values  # Convert to numpy array
y_test_np = y_test.values    # Convert to numpy array

# Get only normal transactions for training (using numpy arrays)
normal_train_mask = (y_train_np == 0)  # Create mask for normal transactions
normal_train_data = X_train_np[normal_train_mask]  # Filter normal transactions

print(f"Training samples - Normal: {len(normal_train_data)}, Fraud: {len(X_train_np) -
len(normal_train_data)}")

# Convert to TensorFlow tensors after filtering
X_train_tf = tf.cast(normal_train_data, tf.float32)  # Use only normal data for training
X_test_tf = tf.cast(X_test_np, tf.float32)

# Build Autoencoder model for anomaly detection
input_dim = X_train_tf.shape[1]  # Number of input features

# Create Autoencoder model
autoencoder = tf.keras.Sequential([
    # Encoder part - compresses the input
    tf.keras.layers.Dense(14, activation='relu', input_shape=(input_dim,)),
    tf.keras.layers.Dense(7, activation='relu'),  # Bottleneck layer

    # Decoder part - reconstructs the input
```

```
    tf.keras.layers.Dense(14, activation='relu'),
    tf.keras.layers.Dense(input_dim, activation='sigmoid')  # Output layer
])

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

print("Autoencoder model summary:")
autoencoder.summary()

print("\nTraining Autoencoder on normal transactions...")
history = autoencoder.fit(
    X_train_tf, X_train_tf,  # Input and target are the same (reconstruction)
    epochs=20,
    batch_size=64,
    validation_data=(X_test_tf, X_test_tf),
    verbose=1
)

# Make predictions on test data
test_predictions = autoencoder.predict(X_test_tf)

# Calculate reconstruction error
reconstruction_error = np.mean(np.square(X_test_np - test_predictions), axis=1)

# Set threshold for anomaly detection
threshold = 1.0  # Adjust this value based on requirements

# Classify based on reconstruction error
```

```
y_pred = (reconstruction_error > threshold).astype(int)

# Calculate accuracy
accuracy = accuracy_score(y_test_np, y_pred)
print("\nModel Accuracy:", accuracy)

# Plot training history
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend()

plt.tight_layout()
plt.show()

# Plot reconstruction errors
```

```
plt.figure(figsize=(10, 6))

# Normal transactions
normal_indices = np.where(y_test_np == 0)[0]
normal_errors = reconstruction_error[normal_indices]
plt.scatter(normal_indices, normal_errors, alpha=0.6, label='Normal', s=10)

# Fraud transactions
fraud_indices = np.where(y_test_np == 1)[0]
fraud_errors = reconstruction_error[fraud_indices]
plt.scatter(fraud_indices, fraud_errors, alpha=0.8, label='Fraud', color='red', s=20)

plt.axhline(y=threshold, color='black', linestyle='--', label=f'Threshold ({threshold})')
plt.title('Reconstruction Errors - Normal vs Fraud Transactions')
plt.ylabel('Reconstruction Error')
plt.xlabel('Transaction Index')
plt.legend()
plt.show()

# Print confusion matrix
cm = confusion_matrix(y_test_np, y_pred)
print("\nConfusion Matrix:")
print(cm)
print("\nClassification Report:")
print(classification_report(y_test_np, y_pred))

# Print summary
print("\n=== AUTOENCODER ANOMALY DETECTION SUMMARY ===")
print("• Trained only on normal transactions")
```

```
print("• Learns to reconstruct normal patterns well")
print("• High reconstruction error indicates fraud")
print(f"• Using threshold: {threshold}")
print(f"• Final accuracy: {accuracy:.4f}")
```

Practical_5

```
# -------------------------
# a. DATA PREPARATION
# -------------------------

# Input sentence (small corpus)
sentences = ["I like deep learning", "I like machine learning"]  # sample text

# Convert sentences to words
words = " ".join(sentences).split()

# Create a vocabulary (unique words)
vocab = sorted(list(set(words)))

# Create mapping of words to numbers and vice-versa
word_to_index = {word: i for i, word in enumerate(vocab)}
index_to_word = {i: word for word, i in word_to_index.items()}

# Show vocabulary
print("Vocabulary:", vocab)

# -------------------------
# b. GENERATE TRAINING DATA (CBOW)
# -------------------------

import numpy as np
```

```
window_size = 1  # number of words before and after the target word

X = []  # input context
Y = []  # target word

for sentence in sentences:
    word_list = sentence.split()
    for i in range(window_size, len(word_list) - window_size):
        context = [word_list[i - 1], word_list[i + 1]]  # surrounding words
        target = word_list[i]  # middle word

        # One-hot encode context words and target word
        x = np.zeros(len(vocab))
        for word in context:
            x[word_to_index[word]] += 1   # bag-of-words vector

        y = np.zeros(len(vocab))
        y[word_to_index[target]] = 1     # one-hot target vector

        X.append(x)
        Y.append(y)

X = np.array(X)
Y = np.array(Y)

print("\nTraining Input X:\n", X)
print("\nTarget Output Y:\n", Y)

# --------------------------
```

```
# c. TRAIN MODEL (Simple Neural Network for CBOW)
# --------------------------

# Define model weights
input_dim = len(vocab)   # vocabulary size
embedding_dim = 5        # vector size (latent representation)

# Random weight initialization
W1 = np.random.randn(input_dim, embedding_dim)   # Input → Hidden
W2 = np.random.randn(embedding_dim, input_dim)   # Hidden → Output

learning_rate = 0.05

# Training using Gradient Descent
for epoch in range(2000):   # number of training cycles
    # Forward pass
    H = np.dot(X, W1)        # hidden layer (context to vector)
    Y_pred = np.dot(H, W2)   # output layer
    Y_pred = np.exp(Y_pred) / np.sum(np.exp(Y_pred), axis=1, keepdims=True)  # softmax

    # Compute error
    loss = np.mean(-np.sum(Y * np.log(Y_pred + 1e-7), axis=1))

    # Backpropagation
    error = Y_pred - Y
    dW2 = np.dot(H.T, error)
    dW1 = np.dot(X.T, np.dot(error, W2.T))

    # Update weights
```

```
    W1 -= learning_rate * dW1
    W2 -= learning_rate * dW2

    if epoch % 400 == 0:
        print(f"Epoch {epoch}, Loss = {loss:.4f}")

# --------------------------
# d. OUTPUT - Check prediction
# --------------------------

def predict(context_words):
    # Create context vector
    x = np.zeros(len(vocab))
    for word in context_words:
        x[word_to_index[word]] += 1
    h = np.dot(x, W1)
    out = np.dot(h, W2)
    softmax = np.exp(out) / np.sum(np.exp(out))
    return index_to_word[np.argmax(softmax)]

print("\nPrediction Example:")
print("Context: ['I', 'deep'] → Predicted word:", predict(["I", "deep"]))
```

Practical 6

```
# Import necessary libraries
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt

# -----------------------------------------------------------
# 1. Load and Prepare CIFAR-10 Dataset
# -----------------------------------------------------------
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()

y_train = to_categorical(y_train, 10)     # Convert labels to one-hot representation
y_test = to_categorical(y_test, 10)

# Normalize pixel values (Convert range from 0-255 to 0-1)
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# -----------------------------------------------------------
# 2. Create a function to resize images (done batch-wise to avoid OOM)
# -----------------------------------------------------------
def preprocess(image, label):
    image = tf.image.resize(image, (128, 128))   # Resize each image to 128x128
    return image, label

# Create batched datasets
```

```
batch_size = 32

train_ds = tf.data.Dataset.from_tensor_slices((x_train,
y_train)).map(preprocess).batch(batch_size)

test_ds = tf.data.Dataset.from_tensor_slices((x_test,
y_test)).map(preprocess).batch(batch_size)


# ----------------------------------------------------------

# 3. Load Pretrained MobileNetV2 (Transfer Learning Base Model)

# ----------------------------------------------------------

base_model = MobileNetV2(

    weights='imagenet',         # Load weights learned from ImageNet dataset

    include_top=False,          # Remove last classification layer

    input_shape=(128, 128, 3)   # Input size after resizing

)


base_model.trainable = False    # Freeze base model weights (do not retrain them)


# ----------------------------------------------------------

# 4. Add Custom Layers on Top (Classifier Head)

# ----------------------------------------------------------

model = models.Sequential([

    base_model,                          # Feature extractor

    layers.GlobalAveragePooling2D(),         # Flatten features

    layers.Dense(128, activation='relu'),     # Dense layer for learning new patterns

    layers.Dropout(0.3),                     # Dropout to avoid overfitting

    layers.Dense(10, activation='softmax')    # Output layer (10 classes)

])


# ----------------------------------------------------------

# 5. Compile Model
```

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

# ------------------------------------------------------------
# 6. Train the Model
# ------------------------------------------------------------
history = model.fit(
    train_ds,
    validation_data=test_ds,
    epochs=5
)

# ------------------------------------------------------------
# 7. Plot Training Accuracy
# ------------------------------------------------------------
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title("Model Accuracy")
plt.legend()
plt.show()

# ------------------------------------------------------------
# 8. Plot Training Loss
# ------------------------------------------------------------
plt.plot(history.history['loss'], label='Train Loss')
```

```
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title("Model Loss")
plt.legend()
plt.show()

# ------------------------------------------------------------
# 9. Final Evaluation
# ------------------------------------------------------------
test_loss, test_acc = model.evaluate(test_ds)
print("✅ Final Test Accuracy:", round(test_acc * 100, 2), "%")
```