Dayananda Sagar College of Engineering

Department of Computer Science and Engineering

Lab Manual - 22CS63

Deep Learning Lab

Academic Year - 2024-25

Course Outcomes

CO1	**Demonstrate foundational knowledge of deep learning concepts, evaluation metrics, and tools.**
CO2	**Design, implement, and evaluate energy-efficient and scalable deep learning models, utilizing modern optimization and regularization techniques.**
CO3	**Analyze and apply advanced architectures, including modern CNNs, RNNs, and Attention Mechanisms, to solve real-world problems.**
CO4	**Explore emerging paradigms such as GANs, Explainable AI, Federated Learning, and energy efficient computing for innovative solutions.**

Table of Contents

Prepared by
Dr. Anusha

GUIDELINES & INSTRUCTIONS TO STUDENTS

- ➔ **Bring your college ID, class notes, lab observation book, and lab record to each lab session.**
- ➔ **Sign in and out of the lab register.**
- ➔ **Arrive on time; late arrivals exceeding 15 minutes may not be permitted.**
- ➔ **100% lab attendance is mandatory.**
- ➔ **Adhere to the dress code.**
- ➔ **No food or drinks allowed.**
- ➔ **Leave bags in the designated area.**
- ➔ **Seek assistance from lab staff for any queries.**
- ➔ **Respect the lab and fellow students.**
- ➔ **Maintain a clean and tidy workspace.**
- ➔ **Do not use external storage devices (floppy disks, pen drives) without lab in-charge permission.**

PREAMBLE

Deep learning is a specialized subset of machine learning, which itself is a branch of artificial intelligence (AI). It focuses on algorithms inspired by the structure and function of the human brain, specifically artificial neural networks (ANNs). These networks consist of layers of interconnected nodes (or neurons) that process and learn from vast amounts of data.

Difference between Machine Learning and Deep Learning:

Machine learning and deep learning AI both are subsets of artificial intelligence but there are many similarities and differences between them.

Sl. No	Machine Learning	Deep Learning
1	Able to work with a limited amount of data.	More substantial dataset volume.
2	Utilize statistical algorithms to uncover the hidden patterns and relationships within the dataset	The model employs an artificial neural network architecture to uncover the hidden patterns and relationships within the dataset
3	Takes less time to train the model.	Takes more time to train the model.
4	It can operate on a CPU or demands less computational power than deep learning.	It requires a high-performance computer with GPU.
5	Better for the low-label task.	Better for complex task like image processing, natural language processing, etc.
6	A model is created by relevant features which are manually extracted from images to detect an object in the image.	Relevant features are automatically extracted from images. It is an end-to-end learning process.
7	Less complex and easy to interpret the result.	More complex, it works like the black box interpretations of the result are not easy.

Applications

Deep learning has revolutionized various fields by enabling machines to perform tasks that require human-like cognitive abilities. Some notable applications include:

1. **Computer Vision:** Used extensively in image and video recognition tasks.

2. **Natural Language Processing (NLP):** Powers applications like language translation and sentiment analysis.
3. **Speech Recognition:** Enables voice-activated assistants like Amazon Alexa and Google Assistant.
4. **Healthcare:** Assists in diagnostics through image analysis and predictive modeling

Types of Deep Learning Models

Several architectures exist within deep learning, each suited for specific tasks:

1. **Convolutional Neural Networks (CNNs):** Primarily used for image processing and recognition due to their ability to capture spatial hierarchies.
2. **Recurrent Neural Networks (RNNs):** Effective for sequential data such as time series or natural language because they maintain a memory of previous inputs.
3. **Deep Reinforcement Learning:** Combines deep learning with reinforcement learning principles, allowing agents to learn optimal behaviors through trial and error.

Challenges in Deep Learning

Deep learning has made significant advancements in various fields, but there are still some challenges that need to be addressed. Here are some of the main challenges in deep learning:

1. **Data availability:** It requires large amounts of data to learn from. For using deep learning it's a big concern to gather as much data for training.
2. **Computational Resources:** For training the deep learning model, it is computationally expensive because it requires specialized hardware like GPUs and TPUs.
3. **Time-consuming:** While working on sequential data depending on the computational resource it can take very large even in days or months.
4. **Interpretability:** Deep learning models are complex; it works like a black box. it is very difficult to interpret the result.
5. **Overfitting:** when the model is trained again and again, it becomes too specialized for the training data, leading to overfitting and poor performance on new data.

Advantages of Deep Learning:

1. **High accuracy:** Deep Learning algorithms can achieve state-of-the-art performance in various tasks, such as image recognition and natural language processing.
2. **Automated feature engineering:** Deep Learning algorithms can automatically discover and learn relevant features from data without the need for manual feature engineering.

3. **Scalability:** Deep Learning models can scale to handle large and complex datasets, and can learn from massive amounts of data.
4. **Flexibility:** Deep Learning models can be applied to a wide range of tasks and can handle various types of data, such as images, text, and speech.
5. **Continual improvement:** Deep Learning models can continually improve their performance as more data becomes available.

Disadvantages of Deep Learning:

1. **High computational requirements:** Deep Learning AI models require large amounts of data and computational resources to train and optimize.
2. **Requires large amounts of labeled data:** Deep Learning models often require a large amount of labeled data for training, which can be expensive and time- consuming to acquire.
3. **Interpretability:** Deep Learning models can be challenging to interpret, making it difficult to understand how they make decisions.
4. **Overfitting:** Deep Learning models can sometimes overfit to the training data, resulting in poor performance on new and unseen data.
5. **Black-box nature:** Deep Learning models are often treated as black boxes, making it difficult to understand how they work and how they arrived at their predictions.

Setup a TensorFlow & Keras Environment.

Step 1: Download anaconda from https://www.anaconda.com/download/success

Step 2: Open Anaconda Navigator: Launch Anaconda Navigator from your applications.

Step 3: Create a New Environment:

- Click on the "Environments" tab on the left.
- Click "Create" at the bottom, name your environment (e.g., tensorflow_env), and select Python 3.8 as the version.

Step 4: Install TensorFlow and Keras:

- In your new environment, ensure you are viewing "Not installed" packages.
- Search for tensorflow, check it, and click "Apply" to install.
- Repeat this step for keras.

Step 5: Test the Installation:

- Open a terminal or a Python console within Anaconda Navigator.
- Run the following commands to verify installation:

```
import tensorflow as tf
import keras
```

- If there are no errors, the installation is complete.

Step 6: Launch Jupyter Notebook: You can also launch Jupyter Notebook from Navigator to start coding with TensorFlow and Keras.

Video link: https://www.youtube.com/watch?v=L4Y7A44lzpM

Laboratory Task 1: Perform basic tensor operations (like addition, multiplication) using Tensor Flow.

Aim: The aim of performing basic tensor operations using TensorFlow is to manipulate multi-dimensional arrays (tensors) efficiently for various computational tasks, particularly in machine learning and data processing.

Theory: This task involves learning how to conduct fundamental mathematical operations such as addition, multiplication, and reshaping of tensors, which are essential for building machine learning models and performing data analysis.

1. **Tensor Creation**

Write a program to create a tensor with specific values and print its shape and data type.

Example: Create a tensor with values [100, 200, 300].

2. **Element-wise Addition**

Implement element-wise addition on two tensors of the same shape.

Example: Create two tensors of shape (2, 3) filled with random values and add them.

3. **Element-wise Subtraction**

Write a program to perform element-wise subtraction on two tensors.

Example: Subtract one tensor from another and print the result.

4. **Element-wise Multiplication**

Perform element-wise multiplication on two tensors.

Example: Multiply two tensors of shape (3, 3) and display the output.

5. **Element-wise Division**

Implement element-wise division between two tensors.

Example: Divide one tensor by another and print the result.

6. **Tensor Reshaping**

Create a tensor and reshape it into a different shape while maintaining the same number of elements.

Example: Reshape a (4,) tensor into (2, 2).

7. **Tensor Square**

Write a program to square each element in a tensor.

Example: Use tf.square() on a tensor and display the results.

8. Broadcasting Operations

Demonstrate how broadcasting works by adding a scalar to a tensor.

Example: Add 5 to all elements of a (3, 3) tensor.

9. Combining Tensors

Concatenate two or more tensors along a specified axis.

Example: Concatenate two tensors of shape (2, 2) along axis 0.

10. Advanced Element-wise Operations

Implement operations such as minimum, maximum, absolute value, logarithm, and exponential on tensors.

Example: Compute the element-wise maximum between two tensors.

Procedure, code & expected output

1. Tensor Creation

- Begin by importing the TensorFlow library.
- Use the tf.constant() function to create a tensor with specific values.
- Access the .shape and .dtype attributes of the tensor to retrieve its shape and data type.

Code:

```
import tensorflow as tf

# Create a TensorFlow constant tensor with specific values
tensor = tf.constant([100, 200, 300])

# Print the shape and data type
print("Tensor Shape:", tensor.shape)
print("Data Type:", tensor.dtype)
```

Output:

```
Tensor Shape: (3,)
Data Type: <dtype: 'int32'>
```

2. Element-wise Addition

- Import Required Libraries: You need to import TensorFlow and NumPy libraries.
- Create Random Tensors: Generate two tensors of the same shape filled with random values.
- Perform Element-wise Addition: Use TensorFlow's tf.add() function to add the two tensors.

- Print Results: Display the original tensors and the result of the addition.

Code:

```
import tensorflow as tf
import numpy as np

# Enable eager execution in TensorFlow 2.x (if not already enabled)
tf.config.run_functions_eagerly(True)

# Step 1: Create two random tensors with shape (2, 3)
ts1 = tf.constant(np.random.rand(2, 3))
ts2 = tf.constant(np.random.rand(2, 3))

# Step 2: Perform element-wise addition
result_tensor = tf.add(ts1, ts2)

# Step 3: Print the original tensors and the result
print("Original tensors:")
print("Tensor1:")
print(ts1.numpy())  # Convert tensor to numpy array for better readability
print("Tensor2:")
print(ts2.numpy())
```

Output:

```
Original tensors:
Tensor1:
[[0.5488135  0.71518937 0.60276338]
 [0.54488318 0.4236548  0.64589411]]
Tensor2:
[[0.43758721 0.891773   0.96366276]
 [0.38344152 0.79172504 0.52889492]]

Result of Element-wise Addition:
[[0.98640071 1.60696237 1.56642614]
```

3. Element-wise Subtraction

- Import Required Libraries: You need to import TensorFlow libraries.
- Create Tensors: Define the tensors you want to subtract. Both tensors should have the same shape or be compatible for broadcasting.
- Perform Subtraction: Use the tf.math.subtract() function or the subtraction operator - to compute the element-wise difference between the two tensors.
- Print the Result: Output the resulting tensor to see the result of the subtraction.

Code:

```
import tensorflow as tf
# Step 2: Create two tensors
a = tf.constant([10, 20, 30], dtype=tf.float32)
b = tf.constant([5, 15, 25], dtype=tf.float32)
# Step 3: Perform element-wise subtraction
result = tf.math.subtract(a, b)
# Step 4: Print the result
print('Result of subtraction:', result.numpy())
```

Output:

```
Result of subtraction: [ 5.  5.  5.]
```

4. Element-wise Multiplication

- Import TensorFlow: Ensure that you have TensorFlow installed and import it into your Python script.
- Create Tensors: Define two tensors with the same shape. In this case, we will create two 3x3 tensors.
- Perform Element-wise Multiplication: Use tf.multiply() or the * operator to multiply the two tensors.
- Display the Output: Print the result to verify the output.

Code:

```
import tensorflow as tf
# Step 2: Create two tensors of shape (3, 3)
tensor_a = tf.constant([[1, 2, 3],
                        [4, 5, 6],
                        [7, 8, 9]])
tensor_b = tf.constant([[9, 8, 7],
                        [6, 5, 4],
                        [3, 2, 1]])
# Step 3: Perform element-wise multiplication
result = tf.multiply(tensor_a, tensor_b)  # Alternatively, you can use tensor_a * tensor_b
# Step 4: Display the output
print(result.numpy())
```

Output:

```
[[ 9 16 21]
 [24 25 24]
 [21 16 9]]
```

5. **Element-wise Division**

   - Create Tensors: Define the tensors you want to divide. These can be created using tf.constant() or other methods.
   - Perform Division: Use either tf.divide(tensor1, tensor2) or the / operator to divide the tensors element-wise.
   - Handle Division by Zero: If there is a possibility of division by zero, consider using tf.where() to replace any division by zero with a specified value (like zero) to avoid NaN results.

Code:

```
import tensorflow as tf
# Step 1: Create two constant tensors
tensor1 = tf.constant([6, 8, 12, 15], dtype=tf.float32)
tensor2 = tf.constant([2, 3, 4, 0], dtype=tf.float32)  # Note the zero in tensor2
# Step 2: Perform element-wise division with handling for division by zero
result = tf.where(tensor2 != 0, tf.divide(tensor1, tensor2), tf.zeros_like(tensor1))
# Step 3: Print the result
print('Result of element-wise division:', result.numpy())
```

Output:

```
Result using tf.divide(): [ 3.         2.66666667  3.         inf]
Result using / operator: [ 3.         2.66666667  3.         inf]
```

6. **Tensor Reshaping**

   - Use tf.reshape() to change the shape of the tensor to the new dimensions.

Code:

```
import tensorflow as tf
# Step 1: Create a tensor of shape (4,)
initial_tensor = tf.constant([1, 2, 3, 4])
# Step 2: Display the original tensor and its shape
print("Original Tensor:")
print(initial_tensor.numpy())
print("Shape of Original Tensor:", initial_tensor.shape)
# Step 3: Reshape the tensor into (2, 2)
reshaped_tensor = tf.reshape(initial_tensor, (2, 2))
# Step 4: Display the reshaped tensor and its new shape
print("\nReshaped Tensor:")
print(reshaped_tensor.numpy())
print("Shape of Reshaped Tensor:", reshaped_tensor.shape)
```

Output:

```
Original Tensor:
[1 2 3 4]
Shape of Original Tensor: (4,)

Reshaped Tensor:
[[1 2]
 [3 4]]
Shape of Reshaped Tensor: (2, 2)
```

7. Tensor Square

- Apply the tf.square() function to the tensor to compute the square of each element.

Code:

```
import tensorflow as tf
# Step 2: Initialize the input tensor
a = tf.constant([-5, -7, 2, 5, 7], dtype=tf.float64)
# Step 3: Calculate the square of each element
res = tf.math.square(a)
# Step 4: Display the results
print('Original Tensor:', a.numpy())
print('Squared Tensor:', res.numpy())
```

Output:

```
Original Tensor: [-5. -7.  2.  5.  7.]
Squared Tensor: [25. 49.  4. 25. 49.]
```

8. Broadcasting Operations

- Define a (3, 3) tensor using tf.constant.
- Simply add the scalar value to the tensor. TensorFlow will automatically broadcast the scalar to match the shape of the tensor.

Code:

```
import tensorflow as tf
# Step 2: Create a (3, 3) tensor
tensor = tf.constant([[1, 2, 3],
                      [4, 5, 6],
                      [7, 8, 9]])
# Step 3: Add the scalar value 5
result = tensor + 5
# Step 4: Print the result
print(result)
```

Output:

```
tf.Tensor(
[[ 6  7  8]
 [ 9 10 11]
 [12 13 14]], shape=(3, 3), dtype=int32)
```

9. Combining Tensors

- Define the tensors you want to concatenate using tf.constant().
- Call the tf.concat() function, passing a list of the tensors to concatenate and specifying the axis along which to concatenate them.

Code:

```
import tensorflow as tf
# Define two example tensors
t1 = tf.constant([[1, 2], [3, 4]])  # Shape (2, 2)
t2 = tf.constant([[5, 6], [7, 8]])  # Shape (2, 2)
# Concatenate along axis 0
result_axis_0 = tf.concat([t1, t2], axis=0)
# Print output
print("Concatenated along axis 0:\n", result_axis_0)
```

Output:

```
Concatenated along axis 0:
 tf.Tensor(
[[1 2]
 [3 4]
 [5 6]
 [7 8]], shape=(4, 2), dtype=int32)
```

10. Advanced Element-wise Operations

Operation	Tensor Function	Example Code Snippet
Element-wise Max	`tf.maximum()`	`tf.maximum(tensor_a, tensor_b)`
Element-wise Min	`tf.minimum()`	`tf.minimum(tensor_a, tensor_b)`
Absolute Value	`tf.abs()`	`tf.abs(tensor_c)`
Logarithm	`tf.math.log()`	`tf.math.log(tensor_d)`
Exponential	`tf.exp()`	`tf.exp(tensor_d)`

Code:

```
import tensorflow as tf
# Define two tensors
tensor_a = tf.constant([[1, 2], [3, 4]])
tensor_b = tf.constant([[4, 3], [2, 1]])
# Compute element-wise maximum
max_tensor = tf.maximum(tensor_a, tensor_b)
print("Element-wise Maximum:\n", max_tensor.numpy())

# Compute element-wise minimum
min_tensor = tf.minimum(tensor_a, tensor_b)
print("Element-wise Minimum:\n", min_tensor.numpy())

# Define a tensor with negative values
tensor_c = tf.constant([[-1, -2], [3, -4]])
# Compute absolute value
abs_tensor = tf.abs(tensor_c)
print("Absolute Value:\n", abs_tensor.numpy())

# Define a tensor with positive values
tensor_d = tf.constant([[1.0, 2.0], [3.0, 4.0]])
# Compute logarithm
log_tensor = tf.math.log(tensor_d)
print("Logarithm:\n", log_tensor.numpy())

# Compute exponential
exp_tensor = tf.exp(tensor_d)
print("Exponential:\n", exp_tensor.numpy())
```

Output:

```
Element-wise Maximum:
 [[4 3]
 [3 4]]
Element-wise Minimum:
 [[1 2]
 [2 1]]
Absolute Value:
 [[1 2]
 [3 4]]
Logarithm:
 [[0.        0.6931472]
 [1.0986123 1.3862944]]
Exponential:
 [[ 2.7182817  7.389056 ]
 [20.085537  54.59815  ]]
```

Assignment Problems

1.	Write a program to Add more than two tensors.
2.	Implement safe division operation.
3.	How do you calculate accuracy, and when is it appropriate to use?
4.	What is precision, and why is it important in certain applications?
5.	What is the F1-score, and when should it be used?
6.	How do you interpret the ROC-AUC score?
7.	Explain recall and its significance.

Laboratory Task 2: Build a simple Sequential CNN model for classifying CIFAR-10/ MNIST dataset

Aim: Gain a deep understanding of the underlying principles of convolutional neural networks, including how convolutional layers, pooling layers, and fully connected layers operate.

Theory: Convolutional Neural Networks are specialized neural networks designed primarily for processing grid-like data such as images. They consist of several layers that transform input data into output predictions through a series of operations, including convolution, pooling, and fully connected layers.

- **Input Layer:** This layer accepts the input image data. For MNIST, images are 28x28 pixels in grayscale.
- **Convolutional Layer:** This layer applies filters (or kernels) to the input image to extract features. Each filter slides over the image and performs a dot product operation, creating feature maps that highlight important patterns like edges or textures.
- **Pooling Layer:** After convolution, pooling layers reduce the spatial dimensions of the feature maps (e.g., using max pooling), which helps decrease computation and control overfitting.
- **Fully Connected Layer:** This layer connects every neuron from the previous layer to every neuron in the current layer, making predictions based on the features extracted by earlier layers.

Procedure, code & expected output

The MNIST dataset consists of 28x28 grayscale images of handwritten digits (0-9). You can load the dataset using libraries like Keras or TensorFlow.

- Import required packages
- Load MNIST dataset
- Check the shape of the datasets
- Normalize the data between 0 and 1 for effective neural network model training
- Split train dataset further to seperate 5000 instances to be used as validation set
- To match the input shape of the CNN model, a channel dimention gets added to each dataset
- Check for the updated shape

- Create CNN model by having convoluted, pooling, dropout and dense layer in the specified order for this experiment. Each convoluted layer is further initialized with specific kernel size, padding, activation and initialization.
- Fit the model
- Save the trained model for later reference (Make sure the folder "models" exists under the current working directory)
- Evaluate the model on test dataset.

Code & Output:

Code	`# Imports required packages`
	
	`import numpy as np`
	`import tensorflow as tf`
	`from tensorflow.keras.datasets import mnist`

O/p	-

Code	`# Loads MNIST dataset`
	`# NOTE: Downloading for the first time may take few minutes to complete`
	
	`mnist = tf.keras.datasets.mnist.load_data()`

O/p	`Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz` `11490434/11490434 [==============================] - 3s 0us/step`
Code	`# Considering dataset is organized in tuple, items are referenced as follows` `(X_train_full, y_train_full), (X_test, y_test) = mnist`
O/p	-

Code	`# Checks the shape of the datasets`
	
	`print("Full training set shape:", X_train_full.shape)`
	`print("Test set shape:", X_test.shape)`

O/p	`Full training set shape: (60000, 28, 28)` `Test set shape: (10000, 28, 28)`

Code	`# Normalizes the data between 0 and 1 for effective neural network model training`
	`X_train_full = X_train_full / 255.`
	`X_test = X_test / 255.`

O/p	-

Code	`# Splits train dataset further to seperate 5000 instances to be used as validation set`
	
	`X_train, X_val = X_train_full[:-5000], X_train_full[-5000:]`
	`y_train, y_val = y_train_full[:-5000], y_train_full[-5000:]`

O/p	-

Code	`# To match the input shape of the CNN model, a channel dimention gets added to each dataset`
	
	`X_train = X_train[..., np.newaxis]`
	`X_val = X_val[..., np.newaxis]`

	`X_test = X_test[..., np.newaxis]`
O/p	-
Code	`# Checks for the updated shape` `X_train.shape`
O/p	`(55000, 28, 28, 1)`

Code	`tf.random.set_seed(42)`
	
	`model = tf.keras.Sequential([`
	`    tf.keras.layers.Conv2D(32, kernel_size=3, padding="same", activation="relu", kernel_initializer="he_normal"),`
	`    tf.keras.layers.Conv2D(64, kernel_size=3, padding="same", activation="relu", kernel_initializer="he_normal"),`
	`    tf.keras.layers.MaxPool2D(),`
	`    tf.keras.layers.Flatten(),`
	`    tf.keras.layers.Dropout(0.25),`
	`    tf.keras.layers.Dense(128, activation="relu", kernel_initializer="he_normal"),`
	`    tf.keras.layers.Dropout(0.5),`
	`    tf.keras.layers.Dense(10, activation="softmax")`
	`])`
	
	`model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam", metrics=["accuracy"])`

O/p	-
Code	`# Fits the model.` `model.fit(X_train, y_train, epochs=10, validation_data=(X_val, y_val))`
O/p	*Epochs running* … … …

Code	`# Saves the trained model for later reference`
	`# NOTE: Make sure the folder "models" exists under the current working directory`
	
	`model.save("./models/my_mnist_cnn_model.keras")`

O/p	-
Code	`# Evaluates the model on test dataset` `model.evaluate(X_test, y_test)`
O/p	*Outputs the Accuracy*

Assignment Problems

1. Create a CNN model to classify images from a custom dataset of your choice (e.g., fruits, vehicles). Include data preprocessing steps and evaluate the model's performance.
2. Add batch normalization layers to an existing CNN model for the CIFAR-10 dataset. Analyze how this affects training speed and model accuracy.
3. Train multiple CNN models using different activation functions (ReLU, Leaky ReLU, ELU) in the hidden layers. Compare their performance on the MNIST dataset.

4.	Provide an overview of the layers used in your model, including convolutional layers, activation functions (e.g., ReLU), pooling layers (e.g., MaxPooling), and dropout layers if applicable.
5.	Explain how each layer contributes to feature extraction and classification.
6.	What techniques did you use to prevent overfitting during training?
7.	What challenges did you encounter while building or training your CNN model? Discuss any issues related to convergence, overfitting, or data handling and how you addressed them.

Laboratory Task 3: Experiment with different optimizers (e.g., Adam vs. RMSProp) and compare their impact on accuracy and convergence.

Aim: The overarching goal of this experimentation is to identify the optimizer that best suits a given deep learning task by examining how each affects the model's ability to learn efficiently and accurately. We'll experiment with three optimizers:

- SGD (Stochastic Gradient Descent)
- Adam (Adaptive Moment Estimation)
- RMSprop (Root Mean Square Propagation)

Theory: In deep learning, optimizers play a crucial role in training models by adjusting the weights based on the gradients of the loss function. Two popular optimizers are Adam (Adaptive Moment Estimation) and RMSProp (Root Mean Square Propagation).

Both Adam and RMSProp are effective optimizers with distinct advantages. Adam is often preferred for its speed and robustness across diverse tasks, while RMSProp may excel in scenarios requiring stable convergence. Experimentation with both optimizers is recommended to determine which works best for specific applications, as their performance can vary significantly depending on the dataset and model architecture used.

RMSProp (Root Mean Square Propagation) is designed to address some limitations of the AdaGrad algorithm, particularly its aggressive diminishing learning rates. It achieves this by maintaining a moving average of the squared gradients, allowing it to adaptively adjust the learning rate for each parameter based on recent gradient behavior.

Adam (Adaptive Moment Estimation) combines the advantages of both RMSProp and momentum-based methods. It not only adapts the learning rates but also keeps track of momentum by maintaining a moving average of both the gradients and their squared values.

Procedure, code & expected output

Optimizers are like guides that help your neural network find the best solution. Imagine your neural network is a hiker trying to find the lowest point in a hilly landscape (representing the minimum loss). The optimizer is the strategy or tool the hiker uses to get to the lowest point as quickly and efficiently as possible.

- Import Libraries: Import necessary libraries including TensorFlow and Keras.
- Load Dataset: Use a standard dataset, such as CIFAR-10, for training.

- Preprocess Data: Normalize the data to improve convergence speed.
- Define Model: Create a Convolutional Neural Network (CNN) architecture.
- Compile Model: Compile the model with different optimizers (Adam and RMSProp).
- Train Model: Train the model using each optimizer and record the accuracy and loss.
- Evaluate Performance: Compare the performance of each optimizer based on training accuracy, validation accuracy, and convergence speed.

Code & Output:

```
import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.callbacks import EarlyStopping, ModelCheckpoint
from keras.optimizers import SGD, Adadelta, Adam, RMSprop,
Adagrad, Nadam, Adamax

SEED = 2017
```

```
data = pd.read_csv('Data/winequality-red.csv', sep=';')
y = data['quality']
X = data.drop(['quality'], axis=1)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=SEED)
X_train, X_val, y_train, y_val = train_test_split(X_train,
y_train, test_size=0.2, random_state=SEED)
```

```
def create_model(opt):
    model = Sequential()
    model.add(Dense(100, input_dim=X_train.shape[1],
    activation='relu'))
    model.add(Dense(50, activation='relu'))
    model.add(Dense(25, activation='relu'))
    model.add(Dense(10, activation='relu'))
    model.add(Dense(1, activation='linear'))
    return model
```

```
def create_callbacks(opt):
    callbacks = [
    EarlyStopping(monitor='val_acc', patience=200, verbose=2),
    ModelCheckpoint('checkpoints/optimizers_best_' + opt +
'.h5', monitor='val_acc', save_best_only=True, verbose=0)
    ]
    return callbacks
```

```
opts = dict({
    'sgd': SGD(),
     'sgd-0001': SGD(lr=0.0001, decay=0.00001),
     'adam': Adam(),
```

```
     'adadelta': Adadelta(),
     'rmsprop': RMSprop(),
     'rmsprop-0001': RMSprop(lr=0.0001),
     'nadam': Nadam(),
     'adamax': Adamax()
    })
```

```
batch_size = 128
n_epochs = 1000

results = []
# Loop through the optimizers
for opt in opts:
    model = create_model(opt)
    callbacks = create_callbacks(opt)
    model.compile(loss='mse', optimizer=opts[opt],
metrics=['accuracy'])
    hist = model.fit(X_train.values, y_train,
batch_size=batch_size, epochs=n_epochs,
validation_data=(X_val.values, y_val), verbose=0,
    callbacks=callbacks)
    best_epoch = np.argmax(hist.history['val_acc'])
    best_acc = hist.history['val_acc'][best_epoch]
    best_model = create_model(opt)
    # Load the model weights with the highest validation
accuracy
    best_model.load_weights('checkpoints/optimizers_best_' +
opt + '.h5')
    best_model.compile(loss='mse', optimizer=opts[opt],
metrics=['accuracy'])
    score = best_model.evaluate(X_test.values, y_test,
verbose=0)
    results.append([opt, best_epoch, best_acc, score[1]])
```

```
res = pd.DataFrame(results)
res.columns = ['optimizer', 'epochs', 'val_accuracy',
'test_accuracy']
res
```

Output:

	optimizer	epochs	val_accuracy	test_accuracy
0	rmsprop	216	0.574219	0.571875
1	adamax	251	0.585938	0.603125
2	sgd-0001	167	0.562500	0.571875
3	nadam	133	0.582031	0.553125
4	adam	139	0.578125	0.581250
5	sgd	0	0.000000	0.000000
6	rmsprop-0001	62	0.550781	0.565625
7	adadelta	208	0.578125	0.575000

Assignment Problems

1. What are the main differences in convergence rates between Adam and RMSProp
2. In which cases would Adam be preferred over RMSProp, and vice versa? Justify your answer with examples.
3. Modify the learning rates of both Adam and RMSProp (e.g., 0.01, 0.001, 0.0001) and observe how they impact model performance. What do you conclude?
4. Implement a custom mini-batch gradient descent optimizer with momentum. Compare its convergence with Adam and RMSProp.
5. You trained two models:
   - Model A (Adam, LR=0.001, Batch Size=32)
   - Model B (RMSProp, LR=0.01, Batch Size=64)

   The validation accuracy of Model A is 92%, while Model B reaches only 85%. Suggest possible reasons and ways to improve Model B.

Laboratory Task 4: Fine-tune a pretrained model like ResNet50 or EfficientNet on a custom dataset.

Aim:

- ➢ Utilize transfer learning by leveraging a pretrained model to adapt to a new dataset.
- ➢ Adjust the model's last layers to classify new categories in the custom dataset.
- ➢ Optimize the model for better accuracy while reducing training time and computational cost.

Theory: Fine-tuning a pretrained model like ResNet50 or EfficientNet on a custom dataset involves leveraging a model trained on a large dataset (e.g., ImageNet) and adapting it to a new, smaller dataset for a specific task. This process allows faster convergence and improved performance with limited data. Fine-tuning is a part of transfer learning, where knowledge from one task (source domain) is transferred to another task (target domain). In deep learning, pretrained models like ResNet50 and EfficientNet are trained on large-scale datasets (e.g., ImageNet with millions of images) and can be reused for different tasks. Fine-tuning typically involves unfreezing some layers of the pretrained model and training them on new data, allowing them to adapt to the specific features of the target dataset.

Transfer Learning vs. Fine-Tuning

Aspect	Transfer Learning	Fine-Tuning
Frozen Layers	Most layers frozen	Some layers unfrozen
Trainable Parameters	Only final classifier layers are trained	Some or all pretrained layers are also trained
Learning Rate	High for classifier layers	Lower for pretrained layers
Use Case	When dataset is small	When dataset is large or similar to original dataset

Procedure, code & expected output

Load a Pretrained Model

- Choose a model (e.g., ResNet50, EfficientNet) with pretrained weights (typically from ImageNet).

- Remove or modify the last classification layer to match the number of classes in the custom dataset.

Prepare the Custom Dataset

- Load images with labels and apply necessary transformations (resizing, normalization, augmentation).
- Split the dataset into training, validation, and test sets.

Modify and Fine-tune the Model

- Replace the original classification head with a new fully connected (dense) layer.
- Optionally, freeze early layers to retain pretrained features while training only the later layers.

Compile and Train the Model

- Choose an appropriate loss function (e.g., cross-entropy for classification).
- Use an optimizer like Adam or SGD with a learning rate scheduler.
- Train the model on the dataset and monitor performance using validation accuracy/loss.

Evaluate and Optimize

- Assess performance on the test set.
- Apply techniques such as hyperparameter tuning, learning rate adjustments, and data augmentation.
- Optionally, unfreeze more layers and retrain to improve feature adaptation.

Code & Output:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.image import
ImageDataGenerator
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.layers import Dense,
GlobalAveragePooling2D, Dropout
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import os
```

```
# Define dataset directories
data_dir = "path/to/dataset"  # Change this to your dataset path
train_dir = os.path.join(data_dir, "train")
val_dir = os.path.join(data_dir, "val")

# Define parameters
img_size = (224, 224)
batch_size = 32
num_classes  =  len(os.listdir(train_dir))    #  Assuming  each
subdirectory is a class
epochs = 10  # Adjust as needed

# Data Augmentation and Preprocessing
datagen_train = ImageDataGenerator(
    rescale=1.0/255,
    rotation_range=30,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True,
    validation_split=0.2  # Split train into train/val
)

datagen_val = ImageDataGenerator(rescale=1.0/255)

train_generator = datagen_train.flow_from_directory(
    train_dir,
    target_size=img_size,
    batch_size=batch_size,
    class_mode='categorical'
)

val_generator = datagen_val.flow_from_directory(
    val_dir,
    target_size=img_size,
    batch_size=batch_size,
    class_mode='categorical'
)

# Load Pretrained Model
base_model  =  ResNet50(weights='imagenet',  include_top=False,
input_shape=(224, 224, 3))
base_model.trainable = False  # Freeze the base model

# Add Custom Layers
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(512, activation='relu')(x)
x = Dropout(0.5)(x)
```

```
out = Dense(num_classes, activation='softmax')(x)

# Compile Model
model = Model(inputs=base_model.input, outputs=out)
model.compile(optimizer=Adam(learning_rate=0.001),
loss='categorical_crossentropy', metrics=['accuracy'])

# Train Model
model.fit(
    train_generator,
    validation_data=val_generator,
    epochs=epochs,
    steps_per_epoch=train_generator.samples // batch_size,
    validation_steps=val_generator.samples // batch_size
)

# Fine-tune the model by unfreezing some layers
base_model.trainable = True
for layer in base_model.layers[:100]:  # Keep some layers frozen
    layer.trainable = False

# Compile again with a lower learning rate
model.compile(optimizer=Adam(learning_rate=0.0001),
loss='categorical_crossentropy', metrics=['accuracy'])

# Train again for fine-tuning
model.fit(
    train_generator,
    validation_data=val_generator,
    epochs=epochs // 2,
    steps_per_epoch=train_generator.samples // batch_size,
    validation_steps=val_generator.samples // batch_size
)

# Save Model
model.save("fine_tuned_resnet50.h5")
```

OUTPUT:

- The console will show progress updates during training, including epoch numbers, loss values, accuracy metrics, and validation results.
- At the end of execution, there will be no explicit output other than confirmation that the model has been saved successfully.

Assignment Problems

1.	What are the key differences between traditional CNNs and Residual Networks (ResNets)?
2.	How do skip connections in ResNets help mitigate the vanishing gradient problem?
3.	In what scenarios would you prefer using a Network in Network (NiN) architecture over a standard CNN?
4.	Explain the concept of inception blocks and their role in improving CNN performance?
5.	What are the trade-offs between model complexity and accuracy when using deeper architectures like ResNets?

Laboratory Task 5: Explore a pretrained model (e.g., MobileNet) on a transfer learning task.

Aim: ***Applies transfer learning to reuse pretrained layers to experiment if it improves model performance with less data.*** Transfer learning involves taking a pretrained model, which has already learned features from a large dataset, and adapting it to a new, but related task. This approach is beneficial because it allows you to leverage existing knowledge, reducing the amount of data and time needed for training.

Theory: MobileNet is a series of efficient convolutional neural network (CNN) architectures designed primarily for mobile and embedded vision applications. Developed by Google, MobileNet models utilize depthwise separable convolutions, which significantly reduce the number of parameters and computational cost compared to traditional CNNs, making them suitable for devices with limited processing power.

MobileNet employs depthwise separable convolutions, which consist of two main operations:

- Depthwise Convolution: Applies a single filter per input channel.
- Pointwise Convolution: Combines the outputs from the depthwise convolution using a 1×1 convolution.

MobileNet introduces two global hyperparameters that allow developers to adjust the model's size and speed:

- Width Multiplier (α): Scales the number of channels in each layer. For example, if $\alpha=0.5$, the model will have half the number of channels, reducing both computational cost and model size.
- Resolution Multiplier (ρ): Adjusts the resolution of input images. By scaling down the input image size, it reduces the computational load further.

The original MobileNet architecture has evolved into several versions, including:

- MobileNetV1: Introduced the depthwise separable convolution concept.
- MobileNetV2: Features an inverted residual structure with linear bottlenecks, enhancing performance on mobile devices

Procedure, code & expected output

- Select MobileNet (or another suitable model) that has been trained on a large dataset like ImageNet. MobileNet is particularly efficient for mobile and edge devices due to its lightweight architecture.

- Load the MobileNet architecture along with its pretrained weights.
- Freeze the initial layers of the model to retain their learned features. This prevents them from being updated during training.
- Add new layers on top of the base model tailored to your specific task. For instance, if you're classifying images into two categories, you might add a dense layer.
- Compile the model with an appropriate optimizer and loss function. For binary classification, you might use binary cross-entropy.
- Train your model using your dataset. Make sure your data is preprocessed to match the input requirements of MobileNet.
- If performance is not satisfactory, consider unfreezing some of the later layers of the base model and retraining with a lower learning rate.

Instead of taking an already trained model (containing pretrained layers), a model gets trained in this experiment to be considered as a pretrained model. To train that model, data for 8 classes out of total 10 classes in Fashion MNIST dataset are used. This is a dataset of 60,000 28x28 grayscale images of 10 fashion categories, along with a test set of 10,000 images. a drop-in replacement for MNIST.

Then a binary classification model (the target model) gets trained (from scratch) on the data from remaining two clssses from the same dataset and its prediction performance gets observed.

Then the same classification model is build by apply transfer learning using pretrained layers from the model created in first step.

Lastly the prediction performance of the target model is compared with that of the model created in the second step. Also, analysis is performed to appreciate if transfer learning speeds up training and make training possible with less data.

Code & Output:

Code

```
# Imports required packages

import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
```

Code

Loading and Preparing Data

```
# Loads fashion mnist dataset
fashion = tf.keras.datasets.fashion_mnist.load_data()
```

	`# Each training and test example is assigned to one of the following labels.`
	`class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat", "Sandal", \`
	`                "Shirt", "Sneaker", "Bag", "Ankle boot"]`
	
	`# Considering dataset is organized in tuple, items are referenced as follows`
	`(X_train_full, y_train_full), (X_test, y_test) = fashion`
	
	`# Checks the shape of the datasets`
	`print("Train dataset shape:", X_train_full.shape)`
	`print("Test dataset shape:", X_test.shape)`

o/p	`Train dataset shape: (60000, 28, 28)` `Test dataset shape: (10000, 28, 28)`
Code	`# Checks the data type of the data` `X_train_full.dtype`
o/p	`dtype('uint8')`

Code	`# Considering the data type of the data, it normalizes the data between 0 and 1`
	`# to make neural network model training efficient`
	`X_train_full, X_test = X_train_full / 255., X_test / 255.`
	
	`# Prints the labels for refer to the class index`
	`y_train_full`

o/p	`array([9, 0, 0, ..., 3, 0, 5], dtype=uint8)`
	Note: Considering the target binary classification model is expected to classify "Pullover" and "T-shirt/top", it separates data for these two classes leaving data for remaining 8 classes to build a model to be considered as pretrained model later.

Code	`# Finds the index for the target class "Pullover" and "T-shirt/top" as`
	`# dataset labels contains class indexes instead of class names`
	
	`class_0_index = class_names.index("Pullover")`
	`class_1_index = class_names.index("T-shirt/top")`
	
	`print("Index of class_0:", class_0_index)`
	`print("Index of class_1:", class_1_index)`

o/p	`Index of class_0: 2` `Index of class_1: 0`

Code	`# Gets the indexes of training label containing either classes`
	`class_0_1_index_flag = [True if (x==class_0_index or x==class_1_index) else False for x in y_train_full]`
	
	`# Shows few flags`
	`print(class_0_1_index_flag[:10])`

o/p	`[False, True, True, False, True, True, False, True, False, False]`
Code	`# Seperates dataset containing data for two classes` `X_train_2_classes_full = X_train_full[class_0_1_index_flag]`

```
# Checks the shape of the dataset
X_train_2_classes_full.shape
```

o/p

```
(12000, 28, 28)
```

Code

```
# Flips bool values (True to False and False to True) to get the flags against
# other classes in the training label
class_0_1_index_flag_flipped = [not flag for flag in class_0_1_index_flag]

# Shows few flags
print(class_0_1_index_flag_flipped[:10])
```

o/p

```
[True, False, False, True, False, False, True, False, True, True]
```

Code

```
# Seperates dataset containing data for the remaining 8 classes
X_train_8_classes_full = X_train_full[class_0_1_index_flag_flipped]

# Checks the shape of the dataset
X_train_8_classes_full.shape
```

o/p

```
(48000, 28, 28)
```

Code

```
# Sum of the first dimension value of both the dataset should be equal to the total number of training instances
X_train_2_classes_full.shape[0] + X_train_8_classes_full.shape[0]
```

o/p

```
60000
```

Code

```
# Similarly, separates targets to contain only respective labels
y_train_2_classes_full = y_train_full[class_0_1_index_flag]
y_train_8_classes_full = y_train_full[class_0_1_index_flag_flipped]

# Checks the shape of the targets
print(y_train_2_classes_full.shape)
print(y_train_8_classes_full.shape)
```

o/p

```
(12000,)
(48000,)
```

NOTE: Modeling
Training Model to be Considered as Pretrained
Preprocesses Datasets

Code

```
# Separates validation dataset
X_train_8_classes, X_val_8_classes, y_train_8_classes, y_val_8_classes = train_test_split(
    X_train_8_classes_full, y_train_8_classes_full, test_size=5000, random_state=42, stratify=y_train_8_classes_full)


# Prints the shape of the separated datasets both containing 8 classes
print(X_train_8_classes.shape)
print(X_val_8_classes.shape)
```

o/p

```
(43000, 28, 28)
(5000, 28, 28)
```

Code

```
# Then standardizes the datasets by first calculating mean and standard deviation, and then
# by subtracting the mean from the data and then dividing the data by standard deviation

pixel_means_8_classes = X_train_8_classes.mean(axis=0, keepdims=True)
pixel_stds_8_classes = X_train_8_classes.std(axis=0, keepdims=True)
```

```
X_train_8_classes_scaled = (X_train_8_classes -
pixel_means_8_classes) / pixel_stds_8_classes
X_val_8_classes_scaled = (X_val_8_classes - pixel_means_8_classes) /
pixel_stds_8_classes

# As the labels ranges from [1, 3, 4, 5, 6, 7, 8, 9], it normalizes
the label from 0 through 7
label_encoder_8_classes = LabelEncoder()
y_train_8_classes_encoded =
label_encoder_8_classes.fit_transform(y_train_8_classes)
y_val_8_classes_encoded =
label_encoder_8_classes.transform(y_val_8_classes)

# Initializes the following densed neural network with arbirary
number of layers and compiles it

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(100, activation="relu",
kernel_initializer="he_normal"),
    tf.keras.layers.Dense(100, activation="relu",
kernel_initializer="he_normal"),
    tf.keras.layers.Dense(100, activation="relu",
kernel_initializer="he_normal"),
    tf.keras.layers.Dense(8, activation="softmax")
])

model.compile(
    loss="sparse_categorical_crossentropy",
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.001),
    metrics=["accuracy"])

# Checks for model summary [optional]
model.summary()
```

o/p

Model: "sequential_1"

Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 784)	0
dense_4 (Dense)	(None, 100)	78,500
dense_5 (Dense)	(None, 100)	10,100
dense_6 (Dense)	(None, 100)	10,100
dense_7 (Dense)	(None, 8)	808

Total params: 99,508 (388.70 KB)

Trainable params: 99,508 (388.70 KB)

Non-trainable params: 0 (0.00 B)

Code	`# Fits the model over specific number iterations (epochs) and validation data`
	`# to observe the learning performance during training`
	`model_history = model.fit(X_train_8_classes_scaled, y_train_8_classes_encoded, epochs=20,`
	`validation_data=(X_val_8_classes_scaled, y_val_8_classes_encoded))`

o/p	*Epochs running* … … …

Code	`# Saves the trained model on disk to be used as pretrained model later.`
	`# NOTE: Folder "model" must exist for model file to be saved into.`
	`model.save("./models/my_fashion_mnist_model.keras")`

	Note: Training Target Model from Scratch ***Preprocesses Datasets***

Code	`# Separates validation dataset from the data containg 2 classes`
	`X_train_2_classes, X_val_2_classes, y_train_2_classes, y_val_2_classes = train_test_split(`
	`X_train_2_classes_full, y_train_2_classes_full, test_size=3000, random_state=42, stratify=y_train_2_classes_full)`

Code	`# Prints the shape of the separated datasets containing both classes`
	`print(X_train_2_classes.shape)`
	`print(X_val_2_classes.shape)`

o/p	(9000, 28, 28) (3000, 28, 28)

Code	`# Then standardizes the datasets by first calculating mean and standard deviation, and then`
	`# by subtracting the mean from the data and then dividing the data by standard deviation`
	
	`pixel_means_2_classes = X_train_2_classes.mean(axis=0, keepdims=True)`
	`pixel_stds_2_classes = X_train_2_classes.std(axis=0, keepdims=True)`
	
	`X_train_2_classes_scaled = (X_train_2_classes - pixel_means_2_classes) / pixel_stds_2_classes`
	`X_val_2_classes_scaled = (X_val_2_classes - pixel_means_2_classes) / pixel_stds_2_classes`

Code	`# As the labels ranges from [1, 3, 4, 5, 6, 7, 8, 9], it normalizes the label from 0 through 7`
	
	`label_encoder_2_classes = LabelEncoder()`
	`y_train_2_classes_encoded = label_encoder_2_classes.fit_transform(y_train_2_classes)`
	`y_val_2_classes_encoded = label_encoder_2_classes.transform(y_val_2_classes)`

Code	`# Clears the name counters and`
	`# sets the global random seed for operations that rely on a random seed`
	`tf.keras.backend.clear_session()`
	`tf.random.set_seed(42)`

```
# Initializes the following densed neural network with arbirary
number of layers and compiles it
model_from_scratch = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(100, activation="relu",
kernel_initializer="he_normal"),
    tf.keras.layers.Dense(100, activation="relu",
kernel_initializer="he_normal"),
    tf.keras.layers.Dense(100, activation="relu",
kernel_initializer="he_normal"),
    tf.keras.layers.Dense(1, activation="sigmoid")
])

model_from_scratch.compile(
    loss="binary_crossentropy",
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.001),
    metrics=["accuracy"])
```

Code

```
# Checks for model summary [optional]
model_from_scratch.summary()
```

o/p

Model: "sequential"

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 100)	78,500
dense_1 (Dense)	(None, 100)	10,100
dense_2 (Dense)	(None, 100)	10,100
dense_3 (Dense)	(None, 1)	101

Total params: 98,801 (385.94 KB)

Trainable params: 98,801 (385.94 KB)

Non-trainable params: 0 (0.00 B)

Code

```
# Fits the model over specific number iterations (epochs) on all the
training data available for the 2 classes
# and validation data to observe the learning performance during
training
model_from_scratch_history =
model_from_scratch.fit(X_train_2_classes_scaled,
y_train_2_classes_encoded, epochs=20,

validation_data=(X_val_2_classes_scaled, y_val_2_classes_encoded))
```

o/p

Epochs running

…

…

…

```
# Gets the indexes of test label containing either classes
class_0_1_index_flag = [True if (x==class_0_index or
x==class_1_index) else False for x in y_test]
```

<table>
<tr><td>Code</td><td><pre><code># Seperates dataset containing data for two classes from the whole test set also containing other classes
X_test_2_classes = X_test[class_0_1_index_flag]

Checks the shape of the dataset
X_test_2_classes.shape</code></pre></td></tr>
<tr><td>o/p</td><td><pre><code>(2000, 28, 28)</code></pre></td></tr>
<tr><td>Code</td><td><pre><code># Similarly, separates targets to contain only respective labels
y_test_2_classes = y_test[class_0_1_index_flag]

Normalizes the test labels for the 2 classes using the already fitted encoder
y_test_2_classes_encoded =
label_encoder_2_classes.transform(y_test_2_classes)

Prints the encoded classes for reference
y_test_2_classes_encoded</code></pre></td></tr>
<tr><td>o/p</td><td><pre><code>array([1, 1, 0, ..., 0, 0, 1])</code></pre></td></tr>
<tr><td>Code</td><td><pre><code># Standardizes the test set by subtracting the mean from the data and then dividing the data by standard deviation
X_test_2_classes_scaled = (X_test_2_classes - pixel_means_2_classes)
/ pixel_stds_2_classes

Evaluates the test prediction performance on the model built from scratch
model_from_scratch.evaluate(X_test_2_classes_scaled,
y_test_2_classes_encoded)</code></pre></td></tr>
<tr><td>o/p</td><td><pre><code>63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 921us/step - accuracy: 0.9
640 - loss: 0.1104
[0.1123715341091156, 0.9620000123977661]</code></pre></td></tr>
<tr><td></td><td>NOTE: The above model that was built from scratch over 9000 [12000 total - 3000 validation instances] training instances containing data for 2 classes, reached 96.20% accuracy in test set. The experiment continues to apply transfer learning by reusing pretrained layers from first model built over other 8 classes to check if new model trained over less data can achieve accuracy from the model built from scratch.</td></tr>
<tr><td>Code</td><td>Transfer Learning<pre><code># Loads the saved model created to be used as pretrained model
model_using_pretrained_layers =
tf.keras.models.load_model("./models/my_fashion_mnist_model.keras")

Checks the model summary especially to refer to the last layer i.e. the output layer
model_using_pretrained_layers.summary()</code></pre></td></tr>
<tr><td>o/p</td><td></td></tr>
</table>

	Model: "sequential_1"

Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 784)	0
dense_4 (Dense)	(None, 100)	78,500
dense_5 (Dense)	(None, 100)	10,100
dense_6 (Dense)	(None, 100)	10,100
dense_7 (Dense)	(None, 8)	808

Total params: 99,510 (388.71 KB)
Trainable params: 99,508 (388.70 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 2 (12.00 B)

Code

```
# Removes the last layer (containing 8 output) to add task specific
binary output layer
model_using_pretrained_layers.pop()

# And then adds a binary output layer
model_using_pretrained_layers.add(tf.keras.layers.Dense(1,
activation="sigmoid", name="output"))

# Then verifies the same visualizing the model summary
model_using_pretrained_layers.summary()
```

o/p

Model: "sequential_1"

Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 784)	0
dense_4 (Dense)	(None, 100)	78,500
dense_5 (Dense)	(None, 100)	10,100
output (Dense)	(None, 1)	101

Total params: 88,703 (346.50 KB)
Trainable params: 88,701 (346.49 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 2 (12.00 B)

Code

Fine-tuning already pretrained model

```
# Considers only 60% of the 2-classes training set to check the
effectiveness of the transfer learning

X_train_2_classes_scaled_subset, _, y_train_2_classes_encoded_subset,
_ = train_test_split(
```

	<pre> X_train_2_classes_scaled, y_train_2_classes_encoded,
	train_size=0.60, stratify=y_train_2_classes_encoded)
	
	# First sets all the pretrained layers (except for the newly added
	output layer) non-trainable
	for layer in model_using_pretrained_layers.layers[:-1]:
	    layer.trainable = False
	
	# Then trains the just the output layer
	
	tf.keras.backend.clear_session()
	tf.random.set_seed(42)
	
	model_using_pretrained_layers.compile(
	    loss="binary_crossentropy",
	optimizer=tf.keras.optimizers.SGD(learning_rate=0.001))
	
	model_using_pretrained_layers_history =
	model_using_pretrained_layers.fit(
	    X_train_2_classes_scaled_subset,
	y_train_2_classes_encoded_subset, epochs=5,
	    validation_data=(X_val_2_classes_scaled,
	y_val_2_classes_encoded))</pre>

o/p	**Epochs running** … … …

Code	<pre># Now, makes all the pretrained layers trainable and performs
	retraining over small smaller
	# learning rate for longer iterations
	
	for layer in model_using_pretrained_layers.layers[:-1]:
	    layer.trainable = True
	
	# Recompiles the model due to change of trainability of the layers
	model_using_pretrained_layers.compile(
	    loss="binary_crossentropy",
	optimizer=tf.keras.optimizers.SGD(learning_rate=0.001))
	
	model_using_pretrained_layers_history =
	model_using_pretrained_layers.fit(
	   X_train_2_classes_scaled_subset, y_train_2_classes_encoded_subset,
	epochs=100,
	   validation_data=(X_val_2_classes_scaled, y_val_2_classes_encoded))</pre>

o/p	**Epochs running** … … …

Code	<pre># Evaluates the test prediction performance on the model built using
	pretrained layers</pre>

	`model_using_pretrained_layers.evaluate(X_test_2_classes_scaled, y_test_2_classes_encoded)`
o/p	`63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 798us/step - accuracy: 0.9687 - loss: 0.0929` `[0.09692149609327316, 0.9674999713897705]`
	NOTE: Though this model built over pretrained layers using on 60% of the available training set, but could also achieved **96.75%** test accuracy as compared to **96.2%** accuracy of the model built from scratch over the full training set. The error rate was improved by **14%** [(96.75−96.2)÷(100−96.20)×100].

Assignment Problems

1.	What layers of the MobileNet model are usually frozen in transfer learning, and why?
2.	How does MobileNet handle computational efficiency?
3.	What improvements could be made to your transfer learning approach?
4.	How do you modify the final layers of MobileNet for a new classification task?
5.	How does input image size affect the performance of MobileNet?

Laboratory Task 6: Create a denoising autoencoder to remove noise from images.

Aim: develop a neural network that can effectively remove noise from images. This is done by training an autoencoder to learn a mapping from noisy images to clean images. The key objectives include:

1. **Noise Reduction** – The model learns to remove various types of noise (e.g., Gaussian noise, salt-and-pepper noise) while preserving important image details.
2. **Feature Learning** – The autoencoder extracts robust features that help in reconstructing a denoised version of the input image.
3. **Unsupervised Learning** – Since autoencoders do not require labeled data, they can be trained on large datasets where only clean images are available.
4. **Generalization** – The trained model should work well on different levels of noise and generalize to unseen noisy images.

How It Works

- **Encoder**: Compresses the noisy input into a lower-dimensional latent representation.
- **Decoder**: Reconstructs the cleaned image from the latent representation.

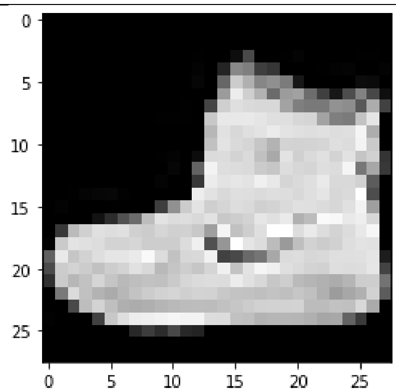
Theory: A denoising autoencoder (DAE) is a type of neural network used to remove noise from images by learning to reconstruct clean images from noisy ones. It is a variant of the standard autoencoder but is explicitly trained to reduce noise. An autoencoder is a neural network architecture that compresses input data into a lower-dimensional representation (encoding) and then reconstructs it back to its original form (decoding). A standard autoencoder learns to reproduce the input but does not explicitly handle noise. A denoising autoencoder, however, is trained with deliberately added noise, so it learns to recover the clean image.

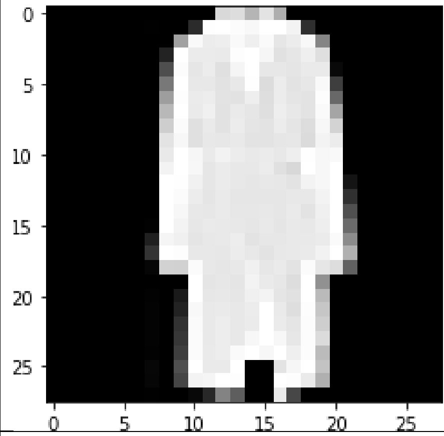
Procedure, code & expected output

1. **Input Image:** A clean image is taken from a dataset.
2. **Add Noise:** Artificial noise (Gaussian noise, salt-and-pepper noise, etc.) is added to the image.
3. **Encoder:** The noisy image is passed through a neural network to extract important features.
4. **Latent Representation:** The network learns a compressed representation of the image.

5. **Decoder:** The compressed representation is used to reconstruct the original clean image.
6. **Output:** The output is compared with the clean image to compute the loss and update the model.

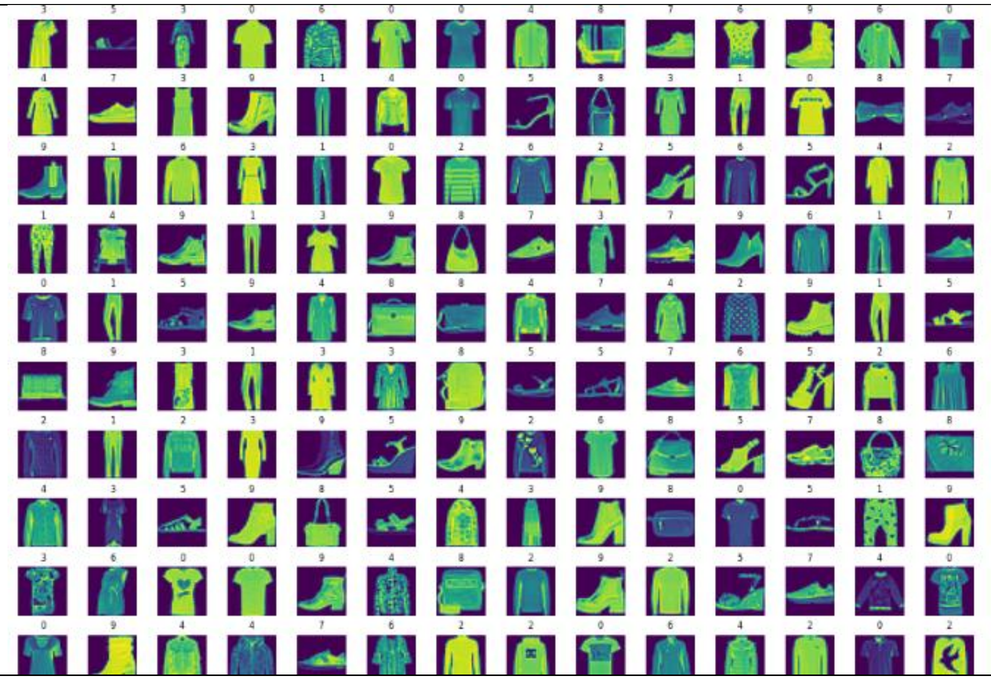
Code & Output:

	STEP #1: IMPORT LIBRARIES AND DATASET
Code	**import** tensorflow **as** tf **import** pandas **as** pd **import** numpy **as** np **import** matplotlib.pyplot **as** plt **import** seaborn **as** sns **import** random
Code	*# Alternatively, you can use the same dataset made readily available by keras Using the following lines of code:* (X_train, y_train), (X_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
Code	plt.imshow(X_train[0], cmap="gray")
o/p	
Code	X_train.shape
o/p	(60000, 28, 28)
Code	X_test.shape
o/p	(10000, 28, 28)
	STEP #2: PERFORM DATA VISUALIZATION
Code	*# Let's view some images!* i = random.randint(1,60000) *# select any random index from 1 to 60,000* plt.imshow(X_train[i] , cmap = 'gray') *# reshape and plot the image*

	
Code	`label = y_train[i]` `label`
o/p	4
Code	see below

```
# Let's view more images in a grid format
# Define the dimensions of the plot grid
W_grid = 15
L_grid = 15

# fig, axes = plt.subplots(L_grid, W_grid)
# subplot return the figure object and axes object
# we can use the axes object to plot specific figures at various
locations

fig, axes = plt.subplots(L_grid, W_grid, figsize = (17,17))

axes = axes.ravel() # flaten the 15 x 15 matrix into 225 array

n_training = len(X_train) # get the length of the training dataset

# Select a random number from 0 to n_training
for i in np.arange(0, W_grid * L_grid): # create evenly spaces
variables

    # Select a random number
    index = np.random.randint(0, n_training)
    # read and display an image with the selected index
    axes[i].imshow( X_train[index] )
    axes[i].set_title(y_train[index], fontsize = 8)
    axes[i].axis('off')

plt.subplots_adjust(hspace=0.4)
```

o/p



STEP #3: PERFORM DATA PREPROCESSING

Code

```
X_train = X_train / 255
X_test = X_test / 255
```

Code

```
noise_factor = 0.3

noise_dataset = []

for img in X_train:
  noisy_image = img + noise_factor * np.random.randn(*img.shape)
  noisy_image = np.clip(noisy_image, 0., 1.)
  noise_dataset.append(noisy_image)
```

Code

```
noise_dataset = np.array(noise_dataset)
```

Code

```
noise_dataset.shape
```

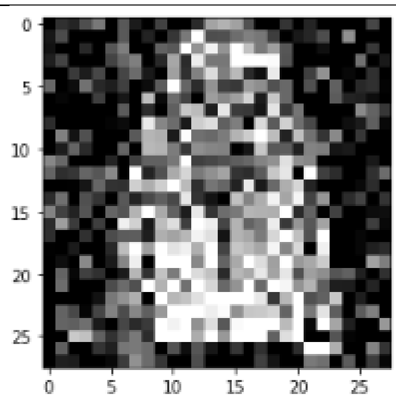
o/p

```
(60000, 28, 28)
```

Code

```
plt.imshow(noise_dataset[22], cmap="gray")
```

o/p



Code

```
noise_test_set = []
for img in X_test:
  noisy_image = img + noise_factor * np.random.randn(*img.shape)
```

	`noisy_image = np.clip(noisy_image, 0., 1.)`
	`noise_test_set.append(noisy_image)`
	
	`noise_test_set = np.array(noise_test_set)`
	`noise_test_set.shape`

o/p	`(10000, 28, 28)`
	STEP #4: BUILD AND TRAIN AUTOENCODER DEEP LEARNING MODEL

Code

```
autoencoder = tf.keras.models.Sequential()

#Encoder
autoencoder.add(tf.keras.layers.Conv2D(filters=16, kernel_size=3,
strides=2, padding="same", input_shape=(28, 28, 1)))
autoencoder.add(tf.keras.layers.Conv2D(filters=8, kernel_size=3,
strides=2, padding="same"))

#Encoded image
autoencoder.add(tf.keras.layers.Conv2D(filters=8, kernel_size=3,
strides=1, padding="same"))

#Decoder
autoencoder.add(tf.keras.layers.Conv2DTranspose(filters=16,
kernel_size=3, strides=2, padding="same"))
autoencoder.add(tf.keras.layers.Conv2DTranspose(filters=1,
kernel_size=3, strides=2, activation='sigmoid', padding="same"))
```

Code

```
autoencoder.compile(loss='binary_crossentropy',
optimizer=tf.keras.optimizers.Adam(lr=0.001))
autoencoder.summary()
```

o/p

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 14, 14, 16)        160
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 7, 7, 8)           1160
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 7, 7, 8)           584
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 14, 14, 16)        1168
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 28, 28, 1)         145
=================================================================
Total params: 3,217
Trainable params: 3,217
Non-trainable params: 0
_________________________________________________________________
```

Code

```
autoencoder.fit(noise_dataset.reshape(-1, 28, 28, 1),
                X_train.reshape(-1, 28, 28, 1),
                epochs=10,
                batch_size=200,
                validation_data=(noise_test_set.reshape(-1, 28, 28,
1), X_test.reshape(-1, 28, 28, 1)))
```

o/p

```
Epoch 1/10
300/300 [==============================] - 11s 36ms/step - loss: 0.3955 - val_loss: 0.3246
Epoch 2/10
300/300 [==============================] - 12s 41ms/step - loss: 0.3157 - val_loss: 0.3138
Epoch 3/10
300/300 [==============================] - 12s 41ms/step - loss: 0.3091 - val_loss: 0.3090
Epoch 4/10
300/300 [==============================] - 12s 42ms/step - loss: 0.3053 - val_loss: 0.3065
Epoch 5/10
300/300 [==============================] - 13s 42ms/step - loss: 0.3035 - val_loss: 0.3050
Epoch 6/10
300/300 [==============================] - 12s 42ms/step - loss: 0.3023 - val_loss: 0.3039
Epoch 7/10
300/300 [==============================] - 15s 50ms/step - loss: 0.3013 - val_loss: 0.3031
Epoch 8/10
300/300 [==============================] - 15s 49ms/step - loss: 0.3007 - val_loss: 0.3028
Epoch 9/10
300/300 [==============================] - 13s 44ms/step - loss: 0.3004 - val_loss: 0.3024
Epoch 10/10
300/300 [==============================] - 11s 37ms/step - loss: 0.3000 - val_loss: 0.3022
```

STEP #5: EVALUATE TRAINED MODEL PERFORMANCE

Code

```
evaluation = autoencoder.evaluate(noise_test_set.reshape(-1, 28, 28,
1), X_test.reshape(-1, 28, 28, 1))
print('Test Accuracy : {:.3f}'.format(evaluation))
```

o/p

```
313/313 [==============================] - 1s 4ms/step - loss: 0.302
2
Test Accuracy : 0.302
```

Code

```
predicted = autoencoder.predict(noise_test_set[:10].reshape(-1, 28,
28, 1))
```

Code

```
predicted.shape
```

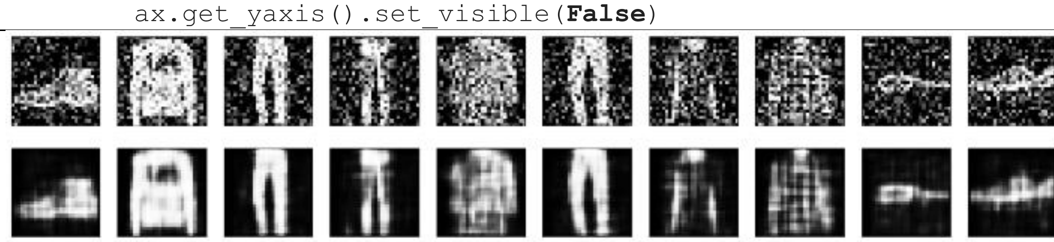
o/p

```
(10, 28, 28, 1)
```

Code

```
fig, axes = plt.subplots(nrows=2, ncols=10, sharex=True,
sharey=True, figsize=(20,4))
for images, row in zip([noise_test_set[:10], predicted], axes):
    for img, ax in zip(images, row):
        ax.imshow(img.reshape((28, 28)), cmap='Greys_r')
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
```

o/p



Assignment Problems

1. What kind of neural network architecture is typically used for denoising autoencoders?
2. How do you prevent overfitting in an autoencoder?
3. How can you improve the performance of your denoising autoencoder?

4.	How does a denoising autoencoder compare to traditional filtering techniques like median or Gaussian filters?
5.	What is the role of the encoder and decoder in an autoencoder?

Laboratory Task 7: Implement a basic RNN for sequence prediction.

Aim: To implement a basic Recurrent Neural Network (RNN) for sequence prediction.

Theory: Recurrent Neural Networks (RNNs) are a class of neural networks designed to process sequential data. Unlike traditional feedforward networks, RNNs have connections that allow information to persist across time steps. This makes them well-suited for tasks like time series prediction, language modeling, and speech recognition.

Key Concepts in RNNs

- Hidden State: Maintains a memory of previous inputs.
- Weight Sharing: The same weights are used across time steps.
- Backpropagation Through Time (BPTT): Used for training RNNs by unrolling them over time.
- Limitations: Standard RNNs suffer from vanishing and exploding gradients, making them inefficient for long sequences.

For sequence prediction, an RNN takes a sequence as input and predicts the next element(s) in the sequence.

Procedure, code & expected output

Steps to Implement a Basic RNN for Sequence Prediction

1. Import necessary libraries.
2. Generate synthetic sequential data.
3. Preprocess the data and prepare training samples.
4. Build a simple RNN model using TensorFlow/Keras.
5. Train the model on the dataset.
6. Evaluate the model and test predictions.

Code & Output:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense

# Generate synthetic sequential data (e.g., a sine wave)
def generate_sequence(n_timesteps):
    x = np.linspace(0, 50, n_timesteps)
    y = np.sin(x)
    return y

# Prepare dataset
n_timesteps = 100
sequence = generate_sequence(n_timesteps)

# Create input-output pairs for training (Sliding window method)
X, y = [], []
```

```
seq_length = 10  # Number of previous steps used for prediction

for i in range(len(sequence) - seq_length):
    X.append(sequence[i:i+seq_length])
    y.append(sequence[i+seq_length])

X, y = np.array(X), np.array(y)

# Reshape input for RNN [samples, timesteps, features]
X = X.reshape((X.shape[0], X.shape[1], 1))

# Build RNN model
model = Sequential([
    SimpleRNN(10, activation='relu', return_sequences=False, input_shape=(seq_length, 1)),
    Dense(1)
])

model.compile(optimizer='adam', loss='mse')

# Train model
model.fit(X, y, epochs=100, verbose=1)

# Make predictions
predictions = model.predict(X)

# Print expected vs. predicted output
print(f"Expected Output: {y[:5]}")
print(f"Predicted Output: {predictions[:5].flatten()}")
```

OUTPUT:
The model will try to learn the sine wave pattern and predict future values.
The Mean Squared Error (MSE) loss should gradually decrease.
The printed predicted values should be close to the expected sine wave values.

Assignment Problems

1. Implement a simple RNN model to predict the next number in a given numerical sequence
2. Modify the RNN model to use different activation functions (tanh, relu, sigmoid) and compare their effects on performance.
3. Change the sequence length used for training (e.g., from 10 to 20) and observe its impact on prediction accuracy.
4. Train the RNN using different optimizers (adam, sgd, rmsprop) and compare their performance.
5. Implement a function to visualize the expected vs. predicted output for a given sequence using Matplotlib.

Laboratory Task 8: Build an LSTM-based model for time-series forecasting or text generation.

Aim: To develop an LSTM-based model for either time-series forecasting or text generation, demonstrating the ability of recurrent neural networks (RNNs) to capture sequential dependencies.

Theory:

Long Short-Term Memory (LSTM) Networks:

LSTM is a type of Recurrent Neural Network (RNN) designed to handle the vanishing gradient problem in standard RNNs. It achieves this by using gates (input, forget, and output) that regulate the flow of information.

Applications of LSTM:

Time-Series Forecasting: Used for predicting stock prices, weather, sales, etc.

Text Generation: Used for generating text based on trained patterns, such as poetry, song lyrics, or chatbot responses.

Procedure, code & expected output

1. Load the dataset (e.g., stock prices, temperature data).
2. Preprocess the data (normalize, reshape, and convert into sequences).
3. Build the LSTM model using TensorFlow/Keras.
4. Train the model and evaluate its performance.
5. Use the model to make predictions.

Code & Output:

```
import tensorflow as tf
import numpy as np
import string

# Load text data
text = open("shakespeare.txt", "r").read().lower()
chars = sorted(set(text))

# Map characters to indices
char_to_idx = {c: i for i, c in enumerate(chars)}
idx_to_char = {i: c for i, c in enumerate(chars)}

# Convert text to sequence of numbers
seq_length = 100
sequences = []
next_chars = []
for i in range(len(text) - seq_length):
    sequences.append([char_to_idx[c] for c in text[i:i+seq_length]])
    next_chars.append(char_to_idx[text[i+seq_length]])
```

```
X = np.array(sequences)
y = np.array(next_chars)

# Reshape input for LSTM
X = X.reshape(X.shape[0], X.shape[1], 1) / len(chars)

# Build LSTM model
model = tf.keras.Sequential([
    tf.keras.layers.LSTM(128, input_shape=(seq_length, 1), return_sequences=True),
    tf.keras.layers.LSTM(128),
    tf.keras.layers.Dense(len(chars), activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")

# Train the model
model.fit(X, y, epochs=20, batch_size=64)

# Function to generate text
def generate_text(seed_text, length=200):
    generated = seed_text
    for _ in range(length):
        x_input = np.array([[char_to_idx[c] for c in generated[-seq_length:]]]) / len(chars)
        x_input = x_input.reshape(1, seq_length, 1)
        predicted_idx = np.argmax(model.predict(x_input))
        generated += idx_to_char[predicted_idx]
    return generated

# Generate new text
print(generate_text("shall i compare thee to a summer's day? "))
```

OUTPUT:

A trained LSTM model that generates text similar to the dataset.

shall i compare thee to a summer's day? thou art more lovely and more temperate:
rough winds do shake the darling buds of may,
and summer's lease hath all too short a date...

Assignment Problems

1. Train an LSTM model on a dataset of your choice for **time-series forecasting** (e.g., weather prediction, stock prices).
2. Modify the **text generation** model to work with words instead of characters.
3. Experiment with **different LSTM architectures**, such as adding more layers or using Bidirectional LSTMs.

Laboratory Task 9: Implement a simple GAN to generate images from random noise (e.g., MNIST digit generation).

Aim: To implement a simple Generative Adversarial Network (GAN) to generate images from random noise, using the MNIST dataset.

Theory:

Generative Adversarial Networks (GANs) consist of two neural networks, the **Generator** and the **Discriminator**, which are trained simultaneously through adversarial learning.

- **Generator (G)**: Takes random noise as input and generates realistic-looking images.
- **Discriminator (D)**: Classifies images as real (from the MNIST dataset) or fake (generated by G).
- **Adversarial Training**: The generator tries to fool the discriminator, while the discriminator tries to correctly distinguish real from fake images.

Loss Function:

- The generator is trained to **minimize** the discriminator's ability to distinguish real from fake images.
- The discriminator is trained to **maximize** the classification accuracy between real and fake images.

Procedure, code & expected output

1. Load the MNIST dataset
2. Preprocess the data
3. Define the Generator network
4. Define the Discriminator network
5. Define the Loss functions and Optimizers
6. Train the GAN
7. Generate and visualize new images

Code & Output:

```
import tensorflow as tf
from tensorflow.keras.layers import Dense, Flatten, Reshape, LeakyReLU, BatchNormalization
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt

# Load and preprocess MNIST dataset
(x_train, _), (_, _) = tf.keras.datasets.mnist.load_data()
x_train = (x_train.astype(np.float32) - 127.5) / 127.5  # Normalize to [-1, 1]
x_train = np.expand_dims(x_train, axis=-1)
```

```
# Define Generator
def build_generator():
    model = Sequential([
        Dense(256, input_dim=100),
        LeakyReLU(0.2),
        BatchNormalization(),
        Dense(512),
        LeakyReLU(0.2),
        BatchNormalization(),
        Dense(1024),
        LeakyReLU(0.2),
        BatchNormalization(),
        Dense(28 * 28 * 1, activation='tanh'),
        Reshape((28, 28, 1))
    ])
    return model

# Define Discriminator
def build_discriminator():
    model = Sequential([
        Flatten(input_shape=(28, 28, 1)),
        Dense(512),
        LeakyReLU(0.2),
        Dense(256),
        LeakyReLU(0.2),
        Dense(1, activation='sigmoid')
    ])
    return model

# Compile models
generator = build_generator()
discriminator = build_discriminator()
discriminator.compile(loss='binary_crossentropy', optimizer=Adam(0.0002, 0.5),
metrics=['accuracy'])

discriminator.trainable = False  # Freeze discriminator during GAN training

# Build GAN
gan_input = tf.keras.Input(shape=(100,))
gan_output = discriminator(generator(gan_input))
gan = tf.keras.Model(gan_input, gan_output)
gan.compile(loss='binary_crossentropy', optimizer=Adam(0.0002, 0.5))

# Training function
def train_gan(epochs=10000, batch_size=128, sample_interval=1000):
    valid = np.ones((batch_size, 1))
    fake = np.zeros((batch_size, 1))
```

```
    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, valid)
        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, valid)

        if epoch % sample_interval == 0:
            print(f"Epoch {epoch}, D Loss: {d_loss[0]}, G Loss: {g_loss}")
            sample_images(epoch)

# Function to generate images
def sample_images(epoch, rows=5, cols=5):
    noise = np.random.normal(0, 1, (rows * cols, 100))
    generated_images = generator.predict(noise)
    generated_images = 0.5 * generated_images + 0.5  # Rescale to [0, 1]

    fig, axs = plt.subplots(rows, cols, figsize=(5, 5))
    count = 0
    for i in range(rows):
        for j in range(cols):
            axs[i, j].imshow(generated_images[count, :, :, 0], cmap='gray')
            axs[i, j].axis('off')
            count += 1
    plt.show()

# Train the GAN
train_gan(epochs=10000, batch_size=128, sample_interval=1000)
```

OUTPUT:
The GAN will generate images of handwritten digits similar to MNIST.
As training progresses, the generated digits will improve in quality.
The loss values (D Loss and G Loss) will be displayed during training.
The function sample_images(epoch) will display generated digits every 1000 epochs.

Assignment Problems

Modify the Generator and Discriminator

- Increase or decrease the number of layers and neurons.
- Change the activation functions (e.g., use ReLU instead of LeakyReLU).
- Experiment with different architectures such as CNN-based GANs.

Experiment with Different Hyperparameters

- Change the learning rate of the optimizer.
- Modify the batch size.
- Train for a different number of epochs and observe the changes.

Use a Different Dataset

- Replace MNIST with CIFAR-10 or Fashion-MNIST.
- Preprocess the dataset accordingly.

Laboratory Task 10: Implement quantization and pruning techniques in a neural network to reduce its size and computational demands compare results with the baseline models

Aim: To implement quantization and pruning techniques in a neural network to reduce its size and computational demands and compare the results with baseline models.

Theory:

1. Quantization

Quantization reduces the precision of numerical values in a model, typically by lowering floating-point precision (e.g., from FP32 to INT8). This reduces model size and speeds up inference on specialized hardware (e.g., CPUs, edge devices).

Types of Quantization:

- **Post-training quantization (PTQ):** Applied after training the model.
- **Quantization-aware training (QAT):** Incorporates quantization into training to improve accuracy.

2. Pruning

Pruning removes redundant or less significant weights from a neural network, reducing its complexity without significant loss in performance.

Types of Pruning:

- **Weight pruning:** Removes individual weights with small magnitudes.
- **Neuron/channel pruning:** Eliminates entire neurons or filters from layers.
- **Structured vs. unstructured pruning:** Structured pruning removes specific patterns (e.g., entire layers), while unstructured pruning removes arbitrary connections.

Procedure, code & expected output:

Baseline Model Training:
- Train a simple neural network on a dataset (e.g., MNIST, CIFAR-10).

Apply Pruning:
- Use techniques like weight pruning and structured pruning.
- Fine-tune the model to recover accuracy.

Apply Quantization:
- Convert the model into INT8 precision using post-training quantization.
- Use TensorFlow/Torch quantization APIs.

Evaluate and Compare:
- Measure accuracy, model size, and inference time.
- Compare results with the original model.

Code & Output:

```
import tensorflow as tf
import tensorflow_model_optimization as tfmot
import numpy as np
import tempfile

# Load MNIST dataset
def load_data():
    (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
    x_train, x_test = x_train / 255.0, x_test / 255.0  # Normalize
    x_train = x_train[..., tf.newaxis].astype(np.float32)
    x_test = x_test[..., tf.newaxis].astype(np.float32)
    return (x_train, y_train), (x_test, y_test)

# Define a simple CNN model
def create_model():
    model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
        tf.keras.layers.MaxPooling2D((2, 2)),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dense(10, activation='softmax')
    ])
    return model

# Train baseline model
def train_model(model, x_train, y_train, x_test, y_test):
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
    model.fit(x_train, y_train, epochs=5, validation_data=(x_test, y_test))
    return model

# Apply Pruning
def prune_model(model):
    pruning_params = {
        'pruning_schedule': tfmot.sparsity.keras.PolynomialDecay(initial_sparsity=0.2,
final_sparsity=0.8, begin_step=0, end_step=1000)
    }
    pruned_model = tfmot.sparsity.keras.prune_low_magnitude(model, **pruning_params)
    pruned_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
    return pruned_model

# Convert to a TFLite Model (Quantization)
def quantize_model(model):
    converter = tf.lite.TFLiteConverter.from_keras_model(model)
    converter.optimizations = [tf.lite.Optimize.DEFAULT]
    quantized_model = converter.convert()
```

```
    return quantized_model

# Evaluate model
def evaluate_model(model, x_test, y_test):
    loss, accuracy = model.evaluate(x_test, y_test, verbose=0)
    return accuracy

# Main Execution
(x_train, y_train), (x_test, y_test) = load_data()

# Train baseline model
baseline_model = create_model()
baseline_model = train_model(baseline_model, x_train, y_train, x_test, y_test)
baseline_accuracy = evaluate_model(baseline_model, x_test, y_test)
print(f'Baseline Accuracy: {baseline_accuracy:.4f}')

# Apply pruning and retrain
pruned_model = prune_model(baseline_model)
pruned_model.fit(x_train, y_train, epochs=2, validation_data=(x_test, y_test))
pruned_accuracy = evaluate_model(pruned_model, x_test, y_test)
print(f'Pruned Model Accuracy: {pruned_accuracy:.4f}')

# Convert and apply quantization
quantized_model = quantize_model(pruned_model)
print(f'Quantized Model Size: {len(quantized_model) / 1024:.2f} KB')
```

OUTPUT:
Baseline Model Accuracy: ~98%
Pruned Model Accuracy: Slight drop (~1-2%)
Quantized Model Size: Reduced significantly (up to 75%)

Assignment Problems

1. **Quantization:** Reduces the precision of weights and activations (e.g., converting 32-bit floating-point numbers to 8-bit integers).
2. **Pruning:** Eliminates unnecessary weights (zeroing out small values) to make the model sparse.
3. Use **Post-Training Quantization** (PTQ) or **Quantization-Aware Training** (QAT) and Measure accuracy loss, size reduction, and inference speed.