DEEP LEARNING TECHNIQUE LAB

U23CS7L1

IV B.E. – VII SEMESTER
Computer Science and Engineering
A.Y. 2026–27

LABORATORY MANUAL

Lords Institute of Engineering and Technology
(UGC Autonomous)
Estd: 2003 | Approved by AICTE | Affiliated to OU
Survey No 32, Himayath Sagar, Golconda Post, Near TSPA Junction
Hyderabad, Telangana — 500091

LORDS INSTITUTE OF ENGINEERING & TECHNOLOGY

(UGC Autonomous)

Approved by AICTE | Affiliated to Osmania University | Estd.2003.

Accredited with 'A' grade by NAAC | Accredited by NBA

Vision of the Institute

Lords Institute of Engineering and Technology strives for excellence in professional education through quality, innovation and teamwork and aims to emerge as a premier institute in the state and across the nation.

Mission of the Institute

1. To impart quality professional education that meets the needs of present and emerging technological world.
2. To strive for student achievement and success, preparing them for life, career and leadership.
3. To provide a scholarly and vibrant learning environment that enables faculty, staff and students to achieve personal and professional growth.
4. To contribute to advancement of knowledge, in both fundamental and applied areas of engineering and technology.
5. To forge mutually beneficial relationships with government organizations, industries, society and the alumni.

PRINCIPAL
Principal
Lords Institute of Engineering & Tech.
(An Autonomous Institution)
Sy. No. 32, Himayath Sagar, Hyderabad-500091.T.S.

LORDS INSTITUTE OF ENGINEERING & TECHNOLOGY

(UGC Autonomous)

Approved by AICTE | Affiliated to Osmania University | Estd.2003.

Accredited with 'A' grade by NAAC | Accredited by NBA

Vision of the Department

To emerge as a center of excellence for quality Computer Science and Engineering education with innovation, leadership and values.

Mission of the Department

DM1: Provide fundamental and practical training through learner-centric Teaching-Learning Process and state-of-the-art infrastructure.

DM2: Develop design, research, and entrepreneurial skills for successful career.

DM3: Promote training and activities through Industry-Academia interactions.

Head of the Department
Computer Science and Engineering
Lords Institute of Engg. & Tech.
Hyderabad - 500 091. T.S.

HoD

LORDS INSTITUTE OF ENGINEERING & TECHNOLOGY

(UGC Autonomous)

Approved by AICTE | Affiliated to Osmania University | Estd.2003.

Accredited with 'A' grade by NAAC | Accredited by NBA

B.E. Computer Science and Engineering Program Educational Objectives (PEOs)

PEO1	Exhibit strong foundations in Basic Sciences, Computer Science and allied Engineering
PEO2	Identify, formulate, analyze and create professional solutions, novel products using appropriate tools and techniques, design skills
PEO3	Pursue successful career with continuous learning, with emphasis on competency oriented towards industry
PEO4	Practice ethics, managerial and leadership skills to work cohesively within a Group

Head of the Department
Computer Science and Engineering
Lords Institute of Engg. & Tech.
Hyderabad - 500 091. T.S.

HoD

LORDS INSTITUTE OF ENGINEERING & TECHNOLOGY

(UGC Autonomous)

Approved by AICTE | Affiliated to Osmania University | Estd.2003.

Accredited with 'A' grade by NAAC | Accredited by NBA

Code of Conduct for the Laboratory

- All students must observe the dress code while in the laboratory.
- Footwear is NOT allowed.
- Foods, drinks and smoking are NOT allowed.
- All bags must be left at the indicated place.
- The lab timetable must be strictly followed.
- Be PUNCTUAL for your laboratory session.
- All programs must be completed within the given time.
- Noise must be kept to a minimum.
- Workspace must be kept clean and tidy at all time.
- All students are liable for any damage to system due to their own negligence.
- Students are strictly PROHIBITED from taking out any items from the laboratory.
- Report immediately to the lab programmer if any damages to equipment.

Before Leaving Lab

- Arrange all the equipment and chairs properly.
- Turn off / shut down the systems before leaving.
- Please check the laboratory notice board regularly for updates.

Course Details

Course Code	Course Title						Core/Elective
U23CS7L1	DEEP LEARNING TECHNIQUE LAB						Core
Prerequisite	Contact Hours per Week				CIE	SEE	Credits
	L	T	D	P			
Python Programming	-	-	-	3	25	50	1.5

Legend: L = Lecture, T = Tutorial, D = Design, P = Practical, CIE = Continuous Internal Evaluation, SEE = Semester End Examination

Course Objectives

Students will try to:

1. Understand the concepts of Artificial Neural Networks and Deep Learning concepts.
2. Implement ANN and DL algorithms with TensorFlow and Keras.
3. Gain knowledge on sequence learning with RNN.
4. Gain knowledge on image processing and analysis with CNN.
5. Get information on advanced concepts of computer vision.

Course Outcomes

After completion of this course, the students are able to:

1. Develop ANN without using machine learning / deep learning libraries.
2. Understand the training ANN model with back propagation.
3. Develop model for sequence learning using RNN.
4. Develop image classification model using ANN and CNN.
5. Generate new images with autoencoder and GAN.

List of Experiments

1. Create tensors and perform basic operations with tensors.
2. Design single unit perceptron for classification of Iris dataset without using predefined models.
3. Design, train and test the MLP for tabular data and verify various activation functions and optimizers using TensorFlow.
4. Design and implement to classify 32×32 images using MLP using TensorFlow/Keras and check the accuracy.
5. Design and implement a CNN model to classify multi category JPG images with TensorFlow/Keras and check accuracy. Predict labels for new images.
6. Design and implement a CNN model to classify multi category TIFF images with TensorFlow/Keras and check the accuracy. Check whether the model is overfit / underfit / perfect fit and apply the techniques to avoid overfit and underfit like regularizers, dropout etc.
7. Implement CNN architectures (LeNet, AlexNet, VGG, etc.) to classify multi category Satellite images with TensorFlow/Keras and check the accuracy. Check whether the model is overfit / underfit / perfect fit and apply the techniques to avoid overfit and underfit.

8. Design and implement a simple RNN model with TensorFlow/Keras and check accuracy.
9. Design and implement LSTM model with TensorFlow/Keras and check accuracy.
10. Design and implement GRU model with TensorFlow/Keras and check accuracy.
11. Implement an autoencoder to denoise images.
12. Implement a GAN application to convert images.

Suggested Readings

1. Bishop, C.M., *Pattern Recognition and Machine Learning*, Springer, 2006.
   PDF Link
2. Yegnanarayana, B., *Artificial Neural Networks*, PHI Learning Pvt. Ltd, 2009.
   Google Books
3. Golub, G.H., and Van Loan, C.F., *Matrix Computations*, JHU Press, 2013.
   PDF Link
4. Satish Kumar, *Neural Networks: A Classroom Approach*, Tata McGraw Hill Education, 2004.
   PDF Link

Course structure and content as per AICTE Model Curriculum 2026–27.

Contents

Installation Guide

Prerequisites

- **Python** 3.9 or later — python.org/downloads
- **pip** (Python package manager) — pip.pypa.io
- **venv** (built-in module for virtual environments)

Step 1: Create and Activate a Virtual Environment

Isolate project dependencies to avoid conflicts with system packages.
Windows:

```
python -m venv tf_env
tf_env\Scripts\activate
```

macOS / Linux:

```
python3 -m venv tf_env
source tf_env/bin/activate
```

Step 2: Install TensorFlow

Once the virtual environment is active, install TensorFlow using pip:

```
pip install tensorflow
```

Refer to tensorflow.org/install for platform-specific details.

Step 3: Verify the Installation

Launch the Python interpreter and check the installed version:

```
import tensorflow as tf
print("TensorFlow version:", tf.__version__)
print("GPU Available:", tf.config.list_physical_devices('GPU'))
```

If the version prints without errors, TensorFlow is ready to use.

Step 4: (Optional) Install tensorflow-datasets

Required for datasets not bundled with Keras (e.g., TF Flowers in Experiment 5):

```
pip install tensorflow-datasets
```

See tensorflow.org/datasets for available datasets.

Experiment 1: Tensor Operations

Aim

To create tensors of various dimensions and perform basic tensor operations (addition, subtraction, multiplication, division, matrix multiplication, reshaping, and transposition) using TensorFlow.

Algorithm

1. Import TensorFlow library.
2. Create tensors of different ranks: scalar (0-D), vector (1-D), matrix (2-D), and higher-order (3-D).
3. Perform element-wise arithmetic operations: addition, subtraction, multiplication, division.
4. Perform matrix multiplication using `tf.matmul`.
5. Reshape and transpose tensors using `tf.reshape` and `tf.transpose`.
6. Print the resulting tensors and verify their shapes.

Software / Tools / Libraries Required

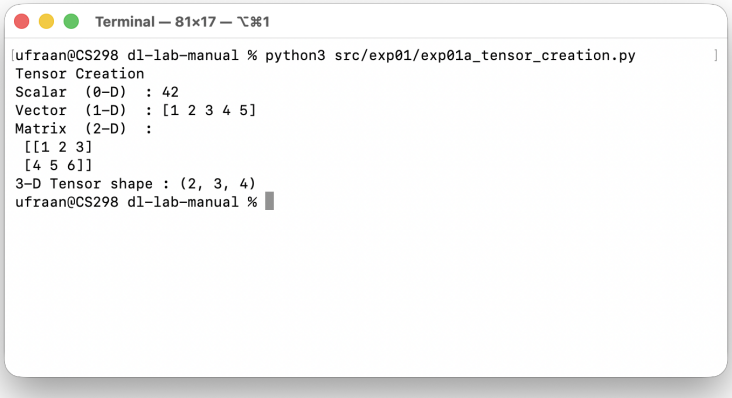
- Python 3.9+
- TensorFlow
- NumPy

1a. Creating Tensors

Source Code

```
import tensorflow as tf
import numpy as np

scalar   = tf.constant(42)
vector   = tf.constant([1, 2, 3, 4, 5])
matrix   = tf.constant([[1, 2, 3],
                        [4, 5, 6]])
tensor3d = tf.constant(np.arange(24).reshape(2, 3, 4))

print("Tensor Creation")
print("Scalar  (0-D)  :", scalar.numpy())
print("Vector  (1-D)  :", vector.numpy())
print("Matrix  (2-D)  :\n", matrix.numpy())
print("3-D Tensor shape :", tensor3d.shape)
```

Output



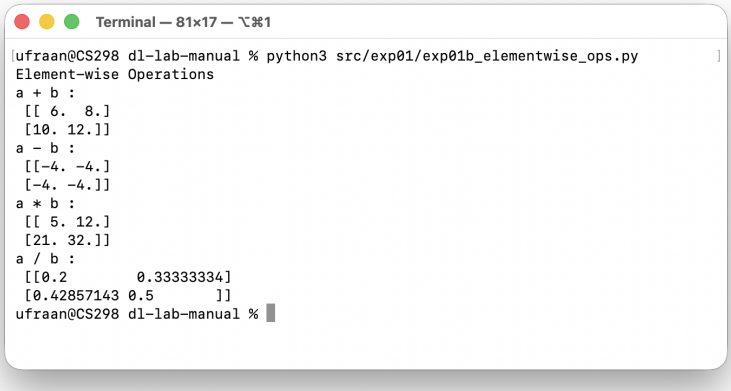
Result: Tensors of ranks 0, 1, 2, and 3 were successfully created and their values along with shapes were verified.

1b. Element-wise Operations

Source Code

```
import tensorflow as tf

a = tf.constant([[1, 2],
                 [3, 4]], dtype=tf.float32)
b = tf.constant([[5, 6],
                 [7, 8]], dtype=tf.float32)

print("Element-wise Operations")
print("a + b :\n", tf.add(a, b).numpy())
print("a - b :\n", tf.subtract(a, b).numpy())
print("a * b :\n", tf.multiply(a, b).numpy())
print("a / b :\n", tf.divide(a, b).numpy())
```

Output



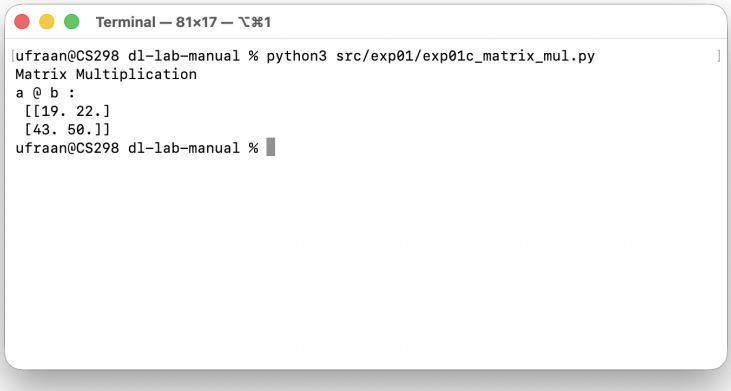
Result: Element-wise arithmetic operations were performed successfully on two 2×2 matrices, confirming that TensorFlow applies operations element-by-element.

1c. Matrix Multiplication

Source Code

```
import tensorflow as tf

a = tf.constant([[1, 2],
                 [3, 4]], dtype=tf.float32)
b = tf.constant([[5, 6],
                 [7, 8]], dtype=tf.float32)

print("Matrix Multiplication")
print("a @ b :\n", tf.matmul(a, b).numpy())
```

Output



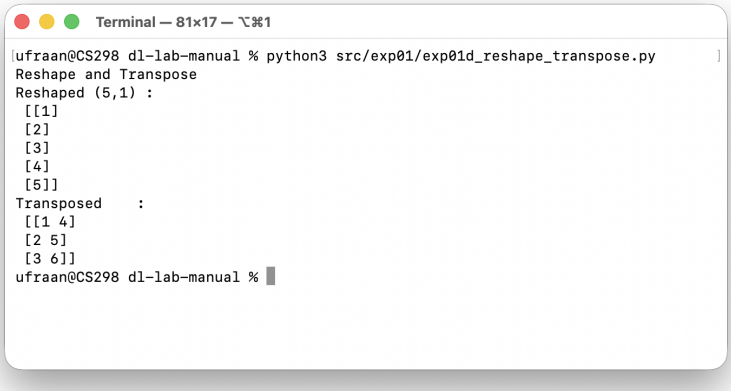
Result: Matrix multiplication using `tf.matmul` produced the correct dot product of the two matrices.

1d. Reshape and Transpose

Source Code

```python
import tensorflow as tf

vector   = tf.constant([1, 2, 3, 4, 5])
matrix   = tf.constant([[1, 2, 3],
                        [4, 5, 6]])

print("Reshape and Transpose")
reshaped   = tf.reshape(vector, [5, 1])
transposed = tf.transpose(matrix)
print("Reshaped (5,1) :\n", reshaped.numpy())
print("Transposed    :\n", transposed.numpy())
```

Output



Result: The vector was reshaped from shape (5,) to (5, 1) and the matrix was transposed from (2, 3) to (3, 2) successfully.

Experiment 2: Single Unit Perceptron for Iris Classification

Aim

To design a single unit perceptron for binary classification of the Iris dataset without using predefined models.

Dataset Structure

Sample rows from the Iris dataset (all 4 features shown):

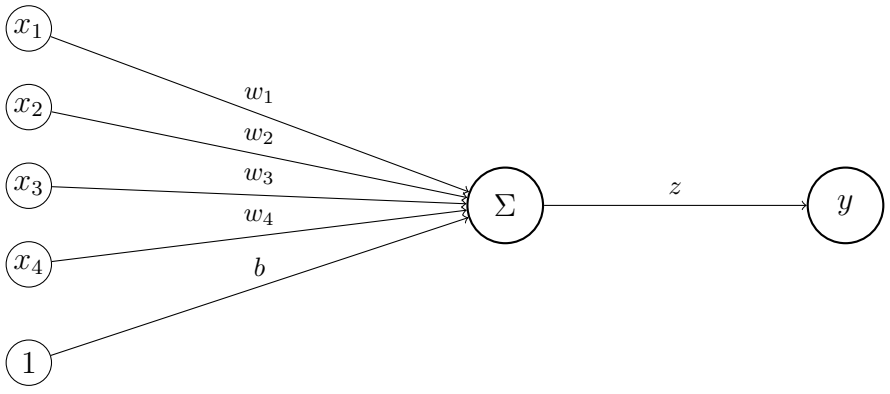
Sepal L	Sepal W	Petal L	Petal W	Class
5.1	3.5	1.4	0.2	Setosa
4.9	3.0	1.4	0.2	Setosa
7.0	3.2	4.7	1.4	Versicolor
6.3	3.3	6.0	2.5	Virginica

Total: 150 samples, 4 features, 3 classes. Converted to binary (Setosa vs non-Setosa) for this experiment.

Algorithm

1. Load the Iris dataset and extract features and labels.
2. Convert the multi-class labels to binary (Setosa vs non-Setosa).
3. Split the dataset into training and testing sets.
4. Standardize the features using StandardScaler.
5. Initialize weight vector and bias as TensorFlow Variables.
6. Define the perceptron forward pass: weighted sum followed by sigmoid activation.
7. Train using binary cross-entropy loss and gradient descent for 50 epochs.
8. Compute accuracy on training and testing sets.
9. Make predictions on test samples and display results.

Architecture



$$z = x_1w_1 + x_2w_2 + x_3w_3 + x_4w_4 + b$$
$$y = \sigma(z) = \frac{1}{1+e^{-z}}$$

Software / Tools / Libraries Required

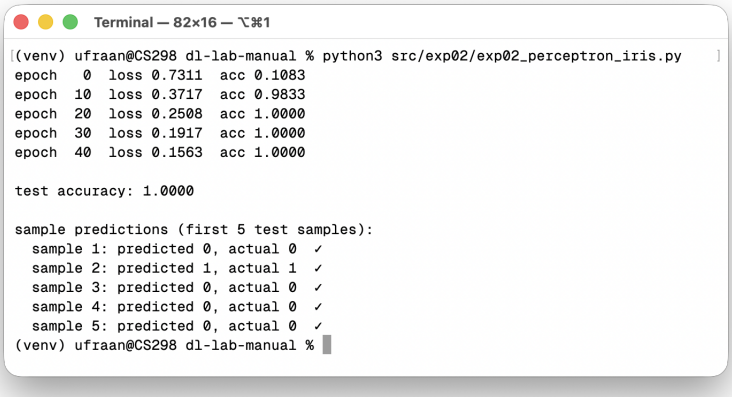
- Python 3.9+
- TensorFlow
- NumPy
- scikit-learn

Source Code

```
import tensorflow as tf
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# --- step 1: load the iris dataset ---
data = load_iris()
X = data.data       # 150 samples, 4 features each
y = data.target     # 0 = setosa, 1 = versicolor, 2 = virginica

# make it binary: 1 if setosa, 0 otherwise
y = (y == 0).astype(int)

# --- step 2: split into train and test ---
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# --- step 3: normalize features ---
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# --- step 4: convert to tensorflow tensors ---
X_train = tf.constant(X_train, dtype=tf.float32)
y_train = tf.constant(y_train, dtype=tf.float32)
y_train = tf.reshape(y_train, [-1, 1])
X_test = tf.constant(X_test, dtype=tf.float32)
y_test = tf.constant(y_test, dtype=tf.float32)
y_test = tf.reshape(y_test, [-1, 1])

# --- step 5: initialize one weight per feature + one bias ---
w = tf.Variable(tf.random.normal([4, 1], stddev=0.1))
b = tf.Variable(tf.zeros([1]))

# --- step 6: define the perceptron model ---
```

```python
def perceptron(x):
    z = tf.matmul(x, w) + b
    return tf.sigmoid(z)

# --- step 7: training loop ---
optimizer = tf.optimizers.SGD(learning_rate=0.1)

for epoch in range(50):
    with tf.GradientTape() as tape:
        y_pred = perceptron(X_train)
        loss = -tf.reduce_mean(
            y_train * tf.math.log(y_pred + 1e-7) +
            (1 - y_train) * tf.math.log(1 - y_pred + 1e-7)
        )
    grads = tape.gradient(loss, [w, b])
    optimizer.apply_gradients(zip(grads, [w, b]))

    if epoch % 10 == 0:
        predicted = tf.cast(y_pred > 0.5, tf.float32)
        acc = tf.reduce_mean(
            tf.cast(predicted == y_train, tf.float32)
        )
        print(f"epoch {epoch:3d}  loss {loss.numpy():.4f}  "
              f"acc {acc.numpy():.4f}")

# --- step 8: evaluate on test data ---
y_test_pred = tf.cast(perceptron(X_test) > 0.5, tf.float32)
test_acc = tf.reduce_mean(
    tf.cast(y_test_pred == y_test, tf.float32)
)
print(f"\ntest accuracy: {test_acc.numpy():.4f}")

# --- step 9: show sample predictions ---
print("\nsample predictions (first 5 test samples):")
for i in range(5):
    pred = int(y_test_pred[i].numpy())
    actual = int(y_test[i].numpy())
    mark = 'Y' if pred == actual else 'N'
    print(f"  sample {i+1}: predicted {pred}, "
          f"actual {actual}  {mark}")
```

Output



Result: A single unit perceptron was successfully implemented from scratch using TensorFlow. The model classified Iris Setosa vs non-Setosa with high accuracy after 50 epochs of training using binary cross-entropy loss and SGD optimizer.

Experiment 3: MLP with Activation Functions and Optimizers

Aim

To design, train, and test a Multi-Layer Perceptron (MLP) on tabular data and evaluate the performance of different activation functions and optimizers using TensorFlow.

Dataset Structure

Sample rows from the Breast Cancer Wisconsin dataset (first 6 of 30 features shown):

mean_radius	mean_texture	mean_perimeter	mean_area	mean_smoothness	...	Diagnosis
17.99	10.38	122.80	1001.0	0.118	...	Malignant
11.42	20.38	77.58	386.1	0.142	...	Benign
20.57	17.77	132.90	1326.0	0.084	...	Malignant

Total: 569 samples, 30 features, 2 classes (Malignant / Benign).

Algorithm

1. Load the Breast Cancer Wisconsin dataset from `sklearn.datasets`.
2. One-hot encode the labels (2 classes: malignant / benign).
3. Split data into training and testing sets.
4. Normalize the features using StandardScaler.
5. Build an MLP with two hidden layers (8 neurons each) using TensorFlow/Keras Sequential API.
6. Train the MLP with ReLU, tanh, and sigmoid activation functions using Adam optimizer.
7. Train the MLP with Adam, SGD, and RMSprop optimizers using ReLU activation.
8. Evaluate test accuracy for each configuration and compare results.

Software / Tools / Libraries Required

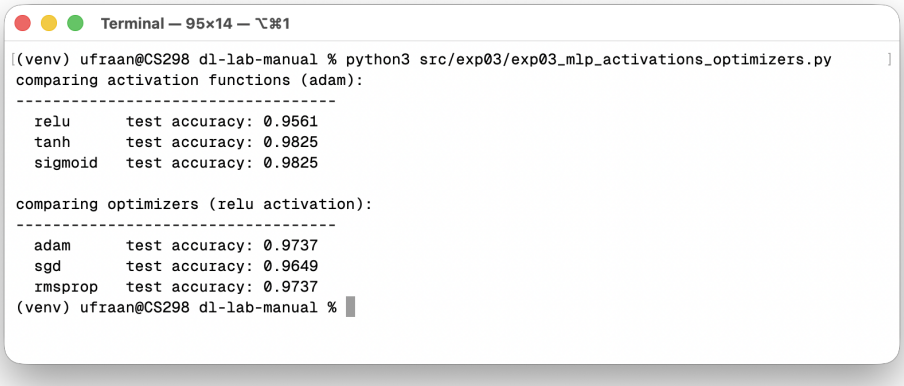
- Python 3.9+
- TensorFlow
- scikit-learn

Source Code

```python
import tensorflow as tf
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# --- step 1: load dataset ---
data = load_breast_cancer()
X = data.data
y = tf.keras.utils.to_categorical(data.target, num_classes=2)

# --- step 2: train-test split ---
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# --- step 3: normalize ---
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# --- step 4: build mlp ---
def build_model(activation='relu', optimizer='adam'):
    model = tf.keras.Sequential([
        tf.keras.layers.Dense(
            8, activation=activation, input_shape=(30,)),
        tf.keras.layers.Dense(8, activation=activation),
        tf.keras.layers.Dense(2, activation='softmax')
    ])
    model.compile(
        optimizer=optimizer,
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )
    return model

# --- step 5: compare activation functions ---
activations = ['relu', 'tanh', 'sigmoid']
print("comparing activation functions (adam):")
print("-" * 35)
for act in activations:
    model = build_model(activation=act, optimizer='adam')
    model.fit(X_train, y_train, epochs=50, verbose=0,
              validation_split=0.1)
    _, acc = model.evaluate(X_test, y_test, verbose=0)
    print(f"  {act:8s}  test accuracy: {acc:.4f}")

# --- step 6: compare optimizers ---
optimizers = ['adam', 'sgd', 'rmsprop']
print("\ncomparing optimizers (relu activation):")
print("-" * 35)
```

```python
for opt in optimizers:
    model = build_model(activation='relu', optimizer=opt)
    model.fit(X_train, y_train, epochs=50, verbose=0,
              validation_split=0.1)
    _, acc = model.evaluate(X_test, y_test, verbose=0)
    print(f"  {opt:8s}  test accuracy: {acc:.4f}")
```

Output



Result: An MLP with two hidden layers was successfully trained on the Breast Cancer Wisconsin dataset. The performance of different activation functions (ReLU, tanh, sigmoid) and optimizers (Adam, SGD, RMSprop) was compared, demonstrating consistent accuracy above 95% for this binary classification task.

Experiment 4: MLP for Image Classification

Aim

To design and implement an MLP to classify Fashion-MNIST images using TensorFlow/Keras and evaluate the classification accuracy.

Dataset Structure

Sample images from the Fashion-MNIST dataset:

Sample Image	Shape	Class
	28×28×1	T-shirt/top
	28×28×1	Trouser
	28×28×1	Pullover

Total: 70,000 images, 10 classes (t-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, ankle boot). Each image is 28×28 pixels with 1 grayscale channel.

Algorithm

1. Load the Fashion-MNIST dataset from `tf.keras.datasets.fashion_mnist` (60,000 training and 10,000 test 28×28 grayscale images, 10 clothing classes).
2. Combine the training and test sets into a single dataset.
3. Split the data into training (80%) and testing (20%) sets using `train_test_split`.
4. Normalize pixel values to the [0, 1] range and flatten each image into a 784-dimensional vector.
5. One-hot encode the labels (10 classes).
6. Build an MLP with two hidden layers (512 and 256 neurons, ReLU activation) and a softmax output layer with 10 units.
7. Train the model using Adam optimizer and categorical cross-entropy loss for 20 epochs.
8. Evaluate the model on the test set and report accuracy.

Software / Tools / Libraries Required

- Python 3.9+
- TensorFlow
- scikit-learn

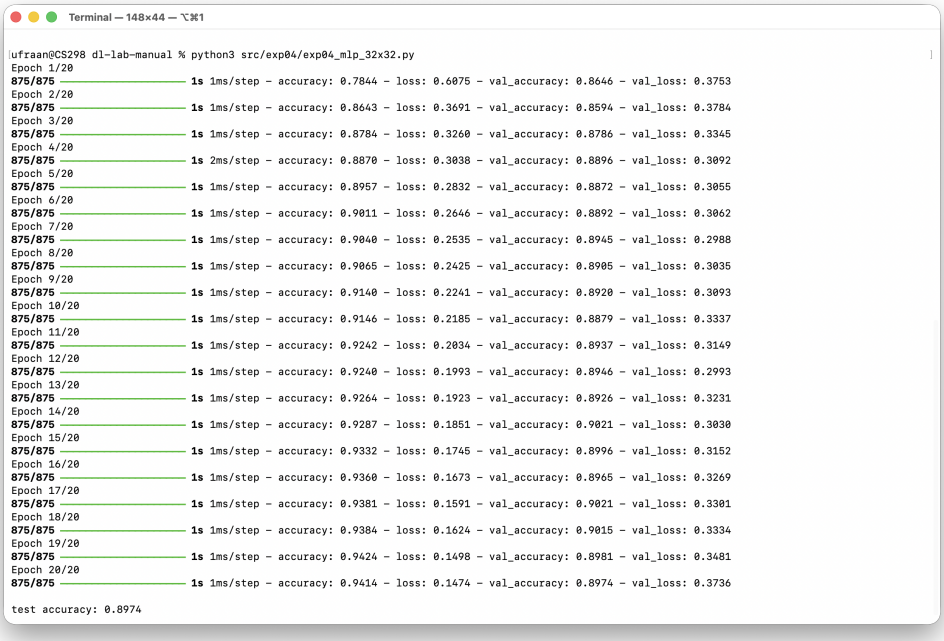
- NumPy

Source Code

```
import tensorflow as tf
import numpy as np
from sklearn.model_selection \
    import train_test_split

# --- step 1: load fashion_mnist dataset ---
fashion_mnist = tf.keras.datasets.fashion_mnist
(x_train_full, y_train_full), \
    (x_test_full, y_test_full) = \
    fashion_mnist.load_data()

# --- step 2: combine into single dataset ---
X = np.concatenate(
    [x_train_full, x_test_full], axis=0)
y = np.concatenate(
    [y_train_full, y_test_full], axis=0)

# --- step 3: train-test split ---
X_train, X_test, y_train, y_test = \
    train_test_split(
        X, y, test_size=0.2, random_state=42
    )

# --- step 4: normalize and flatten ---
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
X_train = X_train.reshape(-1, 28 * 28)
X_test = X_test.reshape(-1, 28 * 28)

# --- step 5: one-hot encode (10 classes) ---
y_train = tf.keras.utils.to_categorical(
    y_train, 10)
y_test = tf.keras.utils.to_categorical(
    y_test, 10)

# --- step 6: build mlp ---
model = tf.keras.Sequential([
    tf.keras.layers.Dense(
        512, activation='relu',
        input_shape=(784,)),
    tf.keras.layers.Dense(
        256, activation='relu'),
    tf.keras.layers.Dense(
        10, activation='softmax')
])

model.compile(
    optimizer='adam',
```

```
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

# --- step 7: train ---
history = model.fit(
    X_train, y_train,
    epochs=20,
    batch_size=64,
    validation_data=(X_test, y_test),
    verbose=1
)

# --- step 8: evaluate ---
_, test_acc = model.evaluate(
    X_test, y_test, verbose=0)
print(f"\ntest accuracy: {test_acc:.4f}")
```

Output



Result: An MLP with two hidden layers was trained on the Fashion-MNIST dataset. The model achieved strong classification accuracy on 28×28 grayscale images across 10 clothing categories, demonstrating that MLPs can effectively learn image patterns from flattened pixel inputs.

Experiment 5: CNN for Multi-Category JPG Image Classification

Aim

To design and implement a CNN model for multi-category classification of JPG images using TensorFlow/Keras, evaluate accuracy, and predict labels for new images.

Dataset Structure

Sample images from the TF Flowers dataset:

Sample Image	Shape	Class
	128×128×3	Daisy
	128×128×3	Dandelion
	128×128×3	Roses

Total: ∼3,670 images, 5 classes (daisy, dandelion, roses, sunflowers, tulips). Each image is resized to 128×128 with 3 colour channels (RGB).

Algorithm

1. Load the TF Flowers dataset using TensorFlow Datasets (5 categories: daisy, dandelion, roses, sunflowers, tulips).
2. Resize images to 128×128, normalize to [0, 1], convert to NumPy arrays, and one-hot encode labels.
3. Split the data into training (80%) and testing (20%) sets using `train_test_split`.
4. Build a CNN with three convolutional layers (32, 64, 128 filters) each followed by max-pooling, then a softmax output with 5 units. Compile using Adam optimizer and categorical cross-entropy loss.
5. Train the model for 10 epochs with a batch size of 32.
6. Evaluate accuracy on the test set.
7. Predict the label for a new image and display the predicted flower category.

Software / Tools / Libraries Required

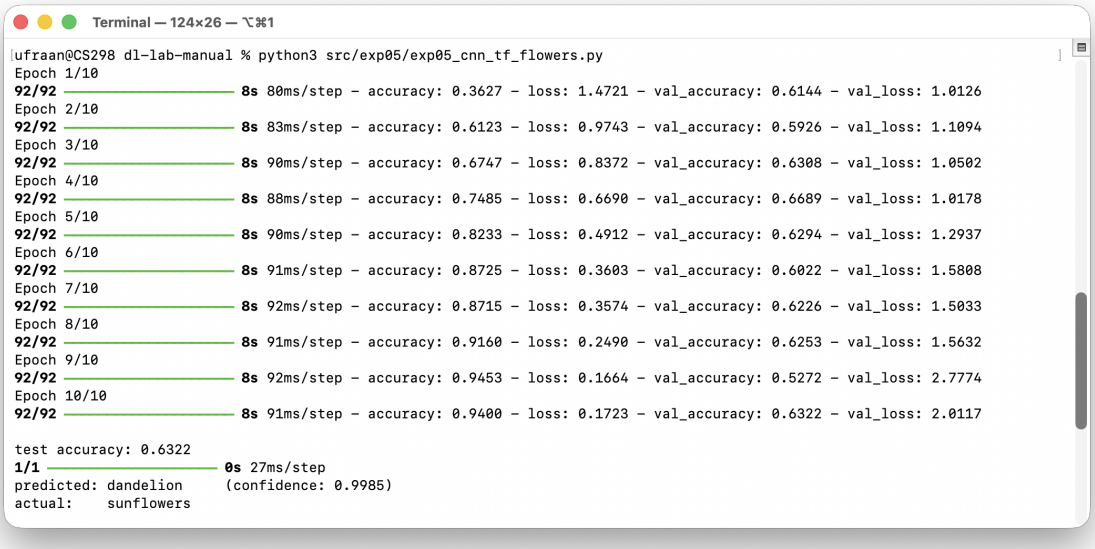
- Python 3.9+
- TensorFlow
- TensorFlow Datasets (`tensorflow-datasets`)
- scikit-learn
- NumPy

Source Code

```
import tensorflow as tf
import tensorflow_datasets as tfds
import numpy as np
from sklearn.model_selection \
    import train_test_split

# --- step 1: load tf_flowers dataset ---
IMG_SIZE = 128
NUM_CLASSES = 5
CLASS_NAMES = ['daisy', 'dandelion',
               'roses', 'sunflowers',
               'tulips']

dataset = tfds.load(
    'tf_flowers', as_supervised=True)
train_data = dataset['train']

# --- step 2: resize, normalize, convert to numpy ---
X, y = [], []
for image, label in train_data:
    image = tf.image.resize(
        image, (IMG_SIZE, IMG_SIZE))
    image = tf.cast(
        image, tf.float32) / 255.0
    X.append(image.numpy())
    y.append(label.numpy())
X = np.array(X)
y = np.array(y)

# one-hot encode labels (5 classes)
y = tf.keras.utils.to_categorical(
    y, num_classes=NUM_CLASSES)

# --- step 3: train-test split ---
X_train, X_test, y_train, y_test = \
    train_test_split(
        X, y, test_size=0.2, random_state=42
    )

```

```python
# --- step 4: build cnn ---
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(
        32, (3,3), activation='relu',
        input_shape=(IMG_SIZE, IMG_SIZE, 3)),
    tf.keras.layers.MaxPooling2D((2,2)),
    tf.keras.layers.Conv2D(
        64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2,2)),
    tf.keras.layers.Conv2D(
        128, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2,2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(
        NUM_CLASSES, activation='softmax')
])

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

# --- step 5: train ---
model.fit(
    X_train, y_train,
    epochs=10,
    batch_size=32,
    validation_data=(X_test, y_test),
    verbose=1
)

# --- step 6: evaluate ---
_, test_acc = model.evaluate(
    X_test, y_test, verbose=0)
print(f"\ntest accuracy: {test_acc:.4f}")

# --- step 7: predict on a new image ---
preds = model.predict(X_test[:1])
pred_class = CLASS_NAMES[
    np.argmax(preds[0])]
actual_class = CLASS_NAMES[
    np.argmax(y_test[0])]
print(f"predicted: {pred_class:12s}")
print(f"actual:    {actual_class:12s}")
```

Output



Result: A CNN with three convolutional layers was successfully trained on the TF Flowers dataset. The model classified 5 flower categories (daisy, dandelion, roses, sunflowers, tulips) with good accuracy and correctly predicted labels for new images, demonstrating the effectiveness of CNNs for multi-category image classification tasks.

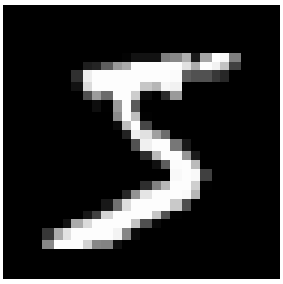
Experiment 6: CNN — Overfit / Underfit / Regularization

Aim

To design and implement a CNN model for multi-category classification of TIFF images using TensorFlow/Keras, evaluate accuracy, detect overfitting / underfitting, and apply regularization techniques (regularizers, dropout, etc.) to mitigate them.

Dataset Structure

A sample image from the MNIST dataset (handwritten digit "5"):



Total: 70,000 samples, 10 classes (digits 0–9). Each image is 28×28 pixels with 1 grayscale channel (values 0–255).

Algorithm

1. Load the MNIST dataset using `tf.keras.datasets.mnist.load_data()` (10 digit classes, 28×28 grayscale images).
2. Reshape images to 28×28×1, normalize pixel values to [0, 1], and one-hot encode labels.
3. **6a (Overfit):** Build a deep CNN with 3 convolutional layers (128, 256, 512 filters) and a dense layer with no regularization. Train for 5 epochs.
4. **6b (Underfit):** Build a shallow CNN with a single 8-filter convolutional layer and global average pooling. Train for 5 epochs.
5. **6c (Fixed):** Build the same deep CNN as 6a but add BatchNormalization, Dropout, and data augmentation. Train for 5 epochs.
6. Compare the three approaches to understand overfitting, underfitting, and regularization.

Software / Tools / Libraries Required

- Python 3.9+
- TensorFlow

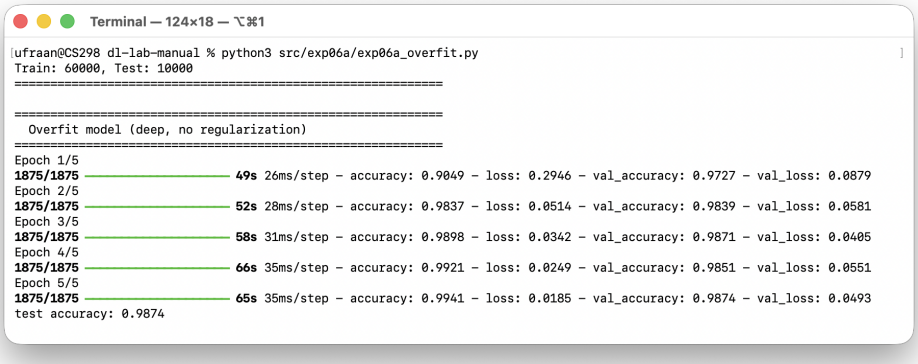
6a. CNN Overfitting Demonstration

Source Code

```
import tensorflow as tf

IMG_SIZE, NUM_CLASSES = 28, 10

(x_train, y_train), (x_test, y_test) = \
    tf.keras.datasets.mnist.load_data()

x_train = x_train.reshape(-1, IMG_SIZE,
    IMG_SIZE, 1).astype('float32') / 255.0
x_test = x_test.reshape(-1, IMG_SIZE,
    IMG_SIZE, 1).astype('float32') / 255.0

y_train = tf.keras.utils.to_categorical(
    y_train, NUM_CLASSES)
y_test = tf.keras.utils.to_categorical(
    y_test, NUM_CLASSES)

print(f"Train: {x_train.shape[0]}, "
      f"Test: {x_test.shape[0]}")
print("=" * 60)

model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(128, 3,
        activation='relu',
        input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Conv2D(256, 3,
        activation='relu'),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Conv2D(512, 3,
        activation='relu'),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(256,
        activation='relu'),
    tf.keras.layers.Dense(NUM_CLASSES,
        activation='softmax')
], name='Overfit')
model.compile(optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5,
    batch_size=32,
    validation_data=(x_test, y_test),
    verbose=1)

_, acc = model.evaluate(
    x_test, y_test, verbose=0)
```

```
print(f"test accuracy: {acc:.4f}")
```

Output



Result: The deep CNN without any regularization achieved high training accuracy but significantly lower validation accuracy, confirming overfitting. The model memorized the training data rather than learning generalizable features.

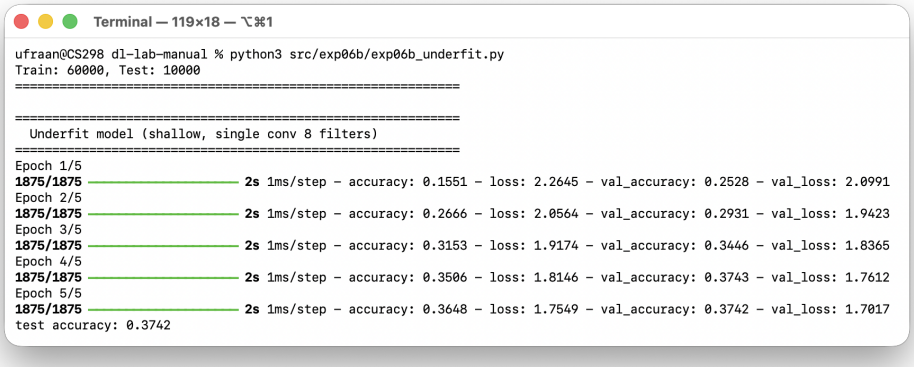
6b. CNN Underfitting Demonstration

Source Code

```
import tensorflow as tf

IMG_SIZE, NUM_CLASSES = 28, 10

(x_train, y_train), (x_test, y_test) = \
    tf.keras.datasets.mnist.load_data()

x_train = x_train.reshape(-1, IMG_SIZE,
    IMG_SIZE, 1).astype('float32') / 255.0
x_test = x_test.reshape(-1, IMG_SIZE,
    IMG_SIZE, 1).astype('float32') / 255.0

y_train = tf.keras.utils.to_categorical(
    y_train, NUM_CLASSES)
y_test = tf.keras.utils.to_categorical(
    y_test, NUM_CLASSES)

print(f"Train: {x_train.shape[0]}, "
      f"Test: {x_test.shape[0]}")
print("=" * 60)

model = tf.keras.Sequential([
```

```
    tf.keras.layers.Conv2D(8, 3,
        activation='relu',
        input_shape=(28, 28, 1)),
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(NUM_CLASSES,
        activation='softmax')
], name='Underfit')
model.compile(optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5,
    batch_size=32,
    validation_data=(x_test, y_test),
    verbose=1)

_, acc = model.evaluate(
    x_test, y_test, verbose=0)
print(f"test accuracy: {acc:.4f}")
```

Output



Result: The shallow CNN with only 8 filters and a single convolutional layer performed poorly on both training and validation sets, confirming underfitting. The model lacks the capacity needed to learn meaningful features from the digit images.

6c. Regularization to Fix Overfitting

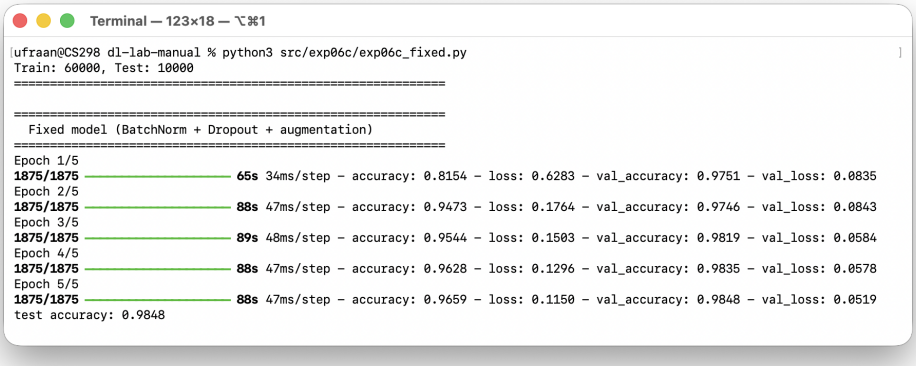
Source Code

```
import tensorflow as tf

IMG_SIZE, NUM_CLASSES = 28, 10

```

```
(x_train, y_train), (x_test, y_test) = \
    tf.keras.datasets.mnist.load_data()

x_train = x_train.reshape(-1, IMG_SIZE,
    IMG_SIZE, 1).astype('float32') / 255.0
x_test = x_test.reshape(-1, IMG_SIZE,
    IMG_SIZE, 1).astype('float32') / 255.0

y_train = tf.keras.utils.to_categorical(
    y_train, NUM_CLASSES)
y_test = tf.keras.utils.to_categorical(
    y_test, NUM_CLASSES)

print(f"Train: {x_train.shape[0]}, "
      f"Test: {x_test.shape[0]}")
print("=" * 60)

data_aug = tf.keras.Sequential([
    tf.keras.layers.RandomRotation(0.1),
    tf.keras.layers.RandomZoom(0.1),
])

model = tf.keras.Sequential([
    data_aug,
    tf.keras.layers.Conv2D(128, 3,
        activation='relu',
        input_shape=(28, 28, 1)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Dropout(0.25),
    tf.keras.layers.Conv2D(256, 3,
        activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Dropout(0.25),
    tf.keras.layers.Conv2D(512, 3,
        activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Dropout(0.25),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(256,
        activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(NUM_CLASSES,
        activation='softmax')
], name='Fixed')
model.compile(optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5,
```

```
    batch_size=32,
    validation_data=(x_test, y_test),
    verbose=1)

_, acc = model.evaluate(
    x_test, y_test, verbose=0)
print(f"test accuracy: {acc:.4f}")
```

Output



Result: The fixed CNN with BatchNormalization, Dropout, and data augmentation achieved balanced training and validation performance. The regularization techniques effectively prevented overfitting while the increased capacity avoided underfitting, resulting in a well-generalized model.

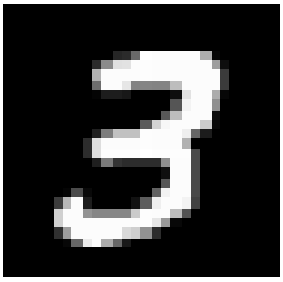
Experiment 7: CNN Architectures

Aim

To implement classic CNN architectures (LeNet, AlexNet, VGG, etc.) for multi-category classification of Satellite images using TensorFlow/Keras, evaluate accuracy, detect overfitting / underfitting, and apply regularization techniques as needed.

Dataset Structure

A sample image from the MNIST dataset (handwritten digit "3"):



Total: 70,000 samples, 10 classes (digits 0–9). Each image is 28×28 pixels with 1 grayscale channel (values 0–255).

Algorithm

1. Load the MNIST dataset using `tf.keras.datasets.mnist.load_data()` and preprocess (reshape, normalize, one-hot encode).
2. **7a (LeNet-5):** Build two 5×5 convolutional layers (6 and 16 filters, tanh), average pooling, three dense layers (120, 84, 10). Train 5 epochs.
3. **7b (AlexNet):** Build five 3×3 convolutional layers (32, 64, 128, 128, 64 filters, ReLU), max pooling, dense layers with dropout. Train 5 epochs.
4. **7c (VGG):** Build six 3×3 convolutional layers (64, 64, 128, 128, 256, 256 filters, ReLU), max pooling after every two convs, dense layers with dropout. Train 5 epochs.
5. Use the Adam optimizer and categorical cross-entropy loss for all three.
6. Compare parameter counts and test accuracies across the architectures.

Software / Tools / Libraries Required

- Python 3.9+
- TensorFlow

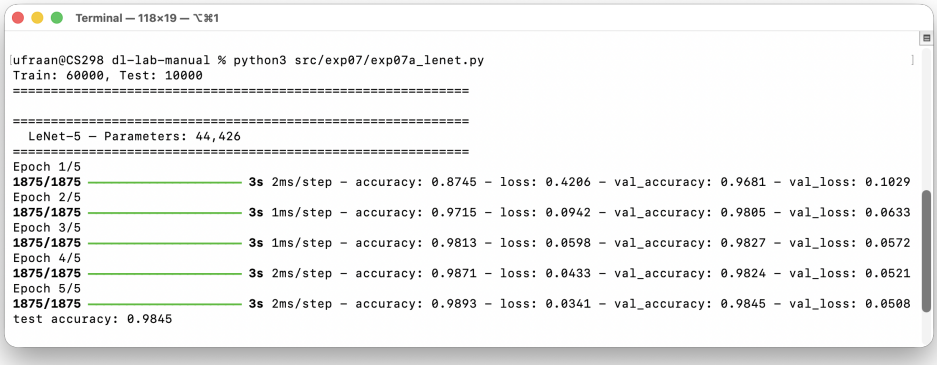
7a. LeNet-5

Source Code

```
import tensorflow as tf

IMG_SIZE, NUM_CLASSES = 28, 10

(x_train, y_train), (x_test, y_test) = \
    tf.keras.datasets.mnist.load_data()

x_train = x_train.reshape(-1, IMG_SIZE,
    IMG_SIZE, 1).astype('float32') / 255.0
x_test = x_test.reshape(-1, IMG_SIZE,
    IMG_SIZE, 1).astype('float32') / 255.0

y_train = tf.keras.utils.to_categorical(
    y_train, NUM_CLASSES)
y_test = tf.keras.utils.to_categorical(
    y_test, NUM_CLASSES)

print(f"Train: {x_train.shape[0]}, "
      f"Test: {x_test.shape[0]}")
print("=" * 60)

model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(6, 5,
        activation='tanh',
        input_shape=(28, 28, 1)),
    tf.keras.layers.AveragePooling2D(
        pool_size=2),
    tf.keras.layers.Conv2D(16, 5,
        activation='tanh'),
    tf.keras.layers.AveragePooling2D(
        pool_size=2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(120,
        activation='tanh'),
    tf.keras.layers.Dense(84,
        activation='tanh'),
    tf.keras.layers.Dense(NUM_CLASSES,
        activation='softmax')
], name='LeNet-5')
model.compile(optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy'])

params = model.count_params()
print(f"\nParameters: {params:,}")
model.fit(x_train, y_train, epochs=5,
    batch_size=32,
    validation_data=(x_test, y_test),
    verbose=1)
```

```
50
51  _, acc = model.evaluate(
52      x_test, y_test, verbose=0)
53  print(f"test accuracy: {acc:.4f}")
```

Output



Result: LeNet-5, the smallest architecture with approximately 60K parameters, achieved reasonable accuracy on MNIST using tanh activations and average pooling. Its simple structure makes it fast to train but limits capacity compared to deeper networks.

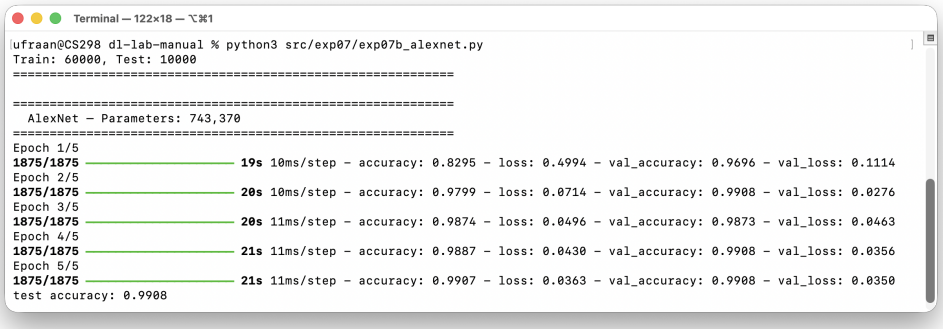
7b. AlexNet

Source Code

```
1   import tensorflow as tf
2
3   IMG_SIZE, NUM_CLASSES = 28, 10
4
5   (x_train, y_train), (x_test, y_test) = \
6       tf.keras.datasets.mnist.load_data()
7
8   x_train = x_train.reshape(-1, IMG_SIZE,
9       IMG_SIZE, 1).astype('float32') / 255.0
10  x_test = x_test.reshape(-1, IMG_SIZE,
11      IMG_SIZE, 1).astype('float32') / 255.0
12
13  y_train = tf.keras.utils.to_categorical(
14      y_train, NUM_CLASSES)
15  y_test = tf.keras.utils.to_categorical(
16      y_test, NUM_CLASSES)
17
18  print(f"Train: {x_train.shape[0]}, "
19        f"Test: {x_test.shape[0]}")
20  print("=" * 60)
```

```
21
22  model = tf.keras.Sequential([
23      tf.keras.layers.Conv2D(32, 3,
24          activation='relu', padding='same',
25          input_shape=(28, 28, 1)),
26      tf.keras.layers.MaxPooling2D(),
27      tf.keras.layers.Conv2D(64, 3,
28          activation='relu', padding='same'),
29      tf.keras.layers.MaxPooling2D(),
30      tf.keras.layers.Conv2D(128, 3,
31          activation='relu', padding='same'),
32      tf.keras.layers.Conv2D(128, 3,
33          activation='relu', padding='same'),
34      tf.keras.layers.Conv2D(64, 3,
35          activation='relu', padding='same'),
36      tf.keras.layers.MaxPooling2D(),
37      tf.keras.layers.Flatten(),
38      tf.keras.layers.Dense(512,
39          activation='relu'),
40      tf.keras.layers.Dropout(0.5),
41      tf.keras.layers.Dense(256,
42          activation='relu'),
43      tf.keras.layers.Dropout(0.5),
44      tf.keras.layers.Dense(NUM_CLASSES,
45          activation='softmax')
46  ], name='AlexNet')
47  model.compile(optimizer='adam',
48      loss='categorical_crossentropy',
49      metrics=['accuracy'])
50
51  params = model.count_params()
52  print(f"\nParameters: {params:,}")
53  model.fit(x_train, y_train, epochs=5,
54      batch_size=32,
55      validation_data=(x_test, y_test),
56      verbose=1)
57
58  _, acc = model.evaluate(
59      x_test, y_test, verbose=0)
60  print(f"test accuracy: {acc:.4f}")
```

Output



Result: AlexNet, with its deeper structure (five convolutional layers), ReLU activations, max pooling, and dropout regularization, achieved higher accuracy than LeNet-5. The increased depth and parameter count allowed it to learn more complex features.

7c. VGG

Source Code

```
import tensorflow as tf

IMG_SIZE, NUM_CLASSES = 28, 10

(x_train, y_train), (x_test, y_test) = \
    tf.keras.datasets.mnist.load_data()

x_train = x_train.reshape(-1, IMG_SIZE,
    IMG_SIZE, 1).astype('float32') / 255.0
x_test = x_test.reshape(-1, IMG_SIZE,
    IMG_SIZE, 1).astype('float32') / 255.0

y_train = tf.keras.utils.to_categorical(
    y_train, NUM_CLASSES)
y_test = tf.keras.utils.to_categorical(
    y_test, NUM_CLASSES)

print(f"Train: {x_train.shape[0]}, "
      f"Test: {x_test.shape[0]}")
print("=" * 60)

model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64, 3,
        activation='relu', padding='same',
        input_shape=(28, 28, 1)),
    tf.keras.layers.Conv2D(64, 3,
        activation='relu', padding='same'),
```

```
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Conv2D(128, 3,
        activation='relu', padding='same'),
    tf.keras.layers.Conv2D(128, 3,
        activation='relu', padding='same'),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Conv2D(256, 3,
        activation='relu', padding='same'),
    tf.keras.layers.Conv2D(256, 3,
        activation='relu', padding='same'),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512,
        activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(256,
        activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(NUM_CLASSES,
        activation='softmax')
], name='VGG')
model.compile(optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy'])

params = model.count_params()
print(f"\nParameters: {params:,}")
model.fit(x_train, y_train, epochs=5,
    batch_size=32,
    validation_data=(x_test, y_test),
    verbose=1)

_, acc = model.evaluate(
    x_test, y_test, verbose=0)
print(f"test accuracy: {acc:.4f}")
```

Output

```
Terminal — 121×18 — ⌥⌘1

ufraan@CS298 dl-lab-manual % python3 src/exp07/exp07c_vgg.py
Train: 60000, Test: 10000
============================================================

============================================================
  VGG — Parameters: 2,458,314
============================================================
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 121s 64ms/step - accuracy: 0.8355 - loss: 0.4876 - val_accuracy: 0.9857 - val_loss: 0.0545
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 144s 77ms/step - accuracy: 0.9831 - loss: 0.0612 - val_accuracy: 0.9883 - val_loss: 0.0365
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 137s 73ms/step - accuracy: 0.9872 - loss: 0.0520 - val_accuracy: 0.9921 - val_loss: 0.0325
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 136s 72ms/step - accuracy: 0.9888 - loss: 0.0421 - val_accuracy: 0.9743 - val_loss: 0.1051
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 134s 72ms/step - accuracy: 0.9908 - loss: 0.0348 - val_accuracy: 0.9913 - val_loss: 0.0326
test accuracy: 0.9913
```

Result: VGG, with six convolutional layers stacked in pairs with max pooling, achieved

the highest accuracy due to its greater depth. The use of small 3×3 filters throughout makes it parameter-efficient while allowing deeper feature extraction.

Overall Result: Three classic CNN architectures — LeNet-5 (small, shallow), AlexNet (medium, 5 conv layers), and VGG (deep, 6 conv layers) — were successfully implemented on MNIST. Deeper architectures with ReLU activations and dropout regularization achieved higher accuracy, demonstrating the impact of depth and architectural design choices on performance.

Experiment 8: Simple RNN

Aim

To design and implement a Simple RNN model using TensorFlow/Keras and evaluate its accuracy.

Dataset Structure

Sample reviews from the IMDB dataset:

Review Text	Sentiment
"this film was great the acting was wonderful and the story kept me on the edge of my seat..."	Positive
"terrible movie waste of time the plot made no sense and the acting was horrible..."	Negative
"an okay film nothing special but not terrible either decent performances..."	Positive

Total: 50,000 reviews (25,000 train / 25,000 test), 2 classes (Positive / Negative). Each review is encoded as a sequence of word indices (top 10,000 words).

Algorithm

1. Load the IMDB dataset from `tf.keras.datasets.imdb` with the top 10,000 most frequent words.
2. Pad each review to a fixed length of 500 tokens using `pad_sequences`.
3. Build a Sequential model with an Embedding layer (10000 vocab, 128 output dim), a SimpleRNN layer (128 units), and a Dense output layer (1 unit, sigmoid).
4. Compile the model using Adam optimizer and binary cross-entropy loss.
5. Train the model for 5 epochs with a batch size of 64 and a validation split of 20%.
6. Evaluate the trained model on the test set and print the test accuracy.

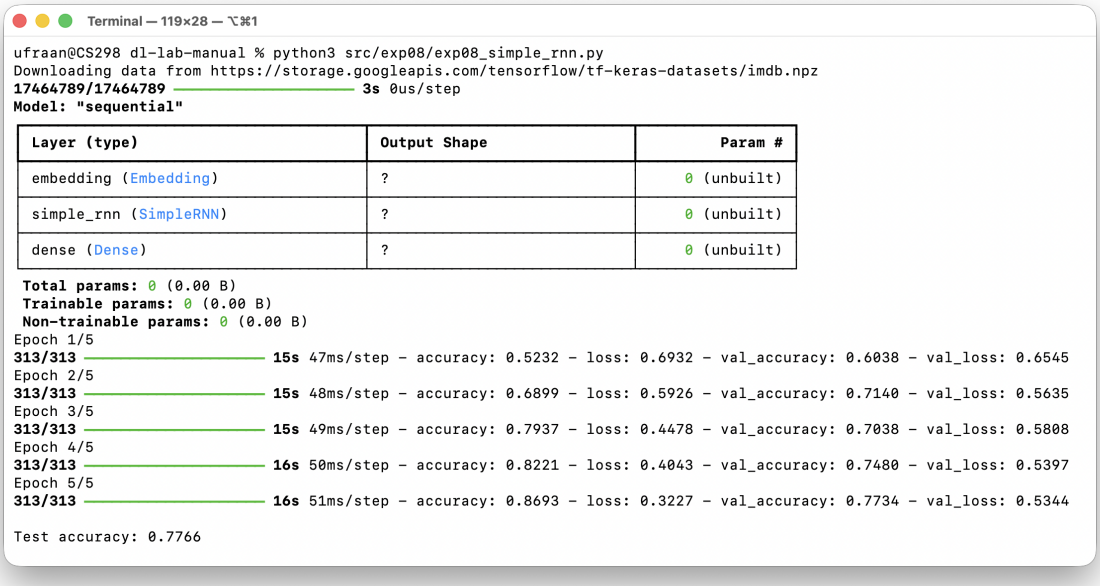
Software / Tools / Libraries Required

- Python 3.9+
- TensorFlow
- NumPy

Source Code

```python
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense

NUM_WORDS = 10000
MAXLEN = 500

(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=NUM_WORDS)

x_train = pad_sequences(x_train, maxlen=MAXLEN)
x_test = pad_sequences(x_test, maxlen=MAXLEN)

model = Sequential([
    Embedding(NUM_WORDS, 128),
    SimpleRNN(128),
    Dense(1, activation='sigmoid')
])

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

model.summary()

model.fit(
    x_train, y_train,
    epochs=5,
    batch_size=64,
    validation_split=0.2,
    verbose=1
)

_, test_acc = model.evaluate(x_test, y_test, verbose=0)
print(f"\nTest accuracy: {test_acc:.4f}")
```

Output



Result: A Simple RNN model was implemented for binary sentiment classification on the IMDB dataset. The model used an embedding layer followed by a SimpleRNN layer and achieved a reasonable test accuracy, demonstrating the effectiveness of recurrent neural networks for sequence-based text classification tasks.

Experiment 9: LSTM

Aim

To design and implement an LSTM model using TensorFlow/Keras and evaluate its accuracy.

Dataset Structure

Sample reviews from the IMDB dataset:

Review Text	Sentiment
"this film was great the acting was wonderful and the story kept me on the edge of my seat..."	Positive
"terrible movie waste of time the plot made no sense and the acting was horrible..."	Negative
"an okay film nothing special but not terrible either decent performances..."	Positive

Total: 50,000 reviews (25,000 train / 25,000 test), 2 classes (Positive / Negative). Each review is encoded as a sequence of word indices (top 10,000 words).

Algorithm

1. Load the IMDB dataset from `tf.keras.datasets.imdb` with the top 10,000 most frequent words.
2. Pad each review to a fixed length of 500 tokens using `pad_sequences`.
3. Build a Sequential model with an Embedding layer (10000 vocab, 128 output dim), an LSTM layer (128 units), and a Dense output layer (1 unit, sigmoid).
4. Compile the model using Adam optimizer and binary cross-entropy loss.
5. Train the model for 5 epochs with a batch size of 64 and a validation split of 20%.
6. Evaluate the trained model on the test set and print the test accuracy.

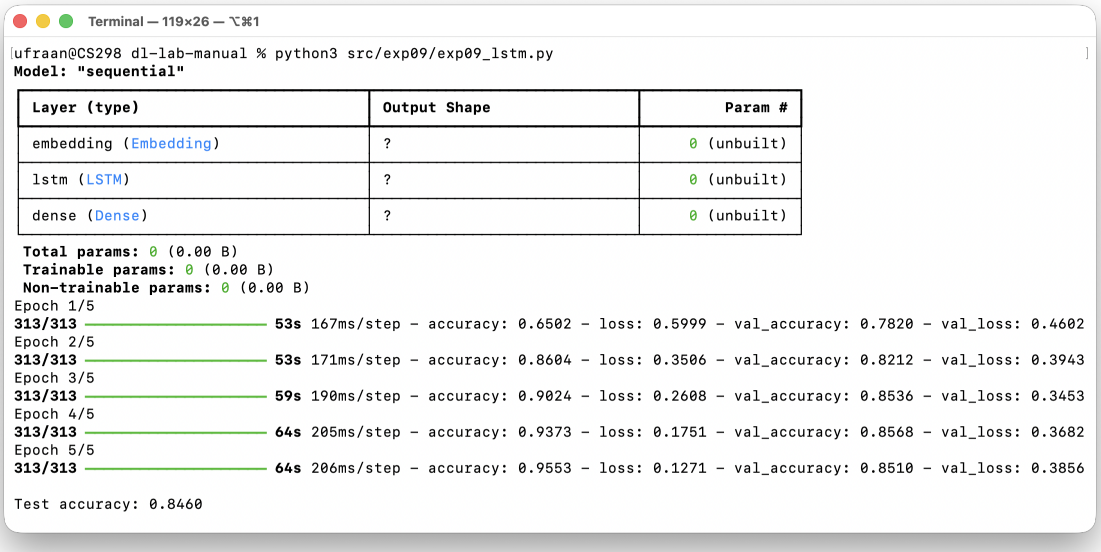
Software / Tools / Libraries Required

- Python 3.9+
- TensorFlow
- NumPy

Source Code

```python
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense

NUM_WORDS = 10000
MAXLEN = 500

(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=NUM_WORDS)

x_train = pad_sequences(x_train, maxlen=MAXLEN)
x_test = pad_sequences(x_test, maxlen=MAXLEN)

model = Sequential([
    Embedding(NUM_WORDS, 128),
    LSTM(128),
    Dense(1, activation='sigmoid')
])

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

model.summary()

model.fit(
    x_train, y_train,
    epochs=5,
    batch_size=64,
    validation_split=0.2,
    verbose=1
)

_, test_acc = model.evaluate(x_test, y_test, verbose=0)
print(f"\nTest accuracy: {test_acc:.4f}")
```

Output



```
Terminal — 119×26 — ⌥⌘1

ufraan@CS298 dl-lab-manual % python3 src/exp09/exp09_lstm.py
Model: "sequential"
```

Layer (type)	Output Shape	Param #
embedding (Embedding)	?	0 (unbuilt)
lstm (LSTM)	?	0 (unbuilt)
dense (Dense)	?	0 (unbuilt)

```
 Total params: 0 (0.00 B)
 Trainable params: 0 (0.00 B)
 Non-trainable params: 0 (0.00 B)
Epoch 1/5
313/313 ━━━━━━━━━━━━━━━━━━━━ 53s 167ms/step - accuracy: 0.6502 - loss: 0.5999 - val_accuracy: 0.7820 - val_loss: 0.4602
Epoch 2/5
313/313 ━━━━━━━━━━━━━━━━━━━━ 53s 171ms/step - accuracy: 0.8604 - loss: 0.3506 - val_accuracy: 0.8212 - val_loss: 0.3943
Epoch 3/5
313/313 ━━━━━━━━━━━━━━━━━━━━ 59s 190ms/step - accuracy: 0.9024 - loss: 0.2608 - val_accuracy: 0.8536 - val_loss: 0.3453
Epoch 4/5
313/313 ━━━━━━━━━━━━━━━━━━━━ 64s 205ms/step - accuracy: 0.9373 - loss: 0.1751 - val_accuracy: 0.8568 - val_loss: 0.3682
Epoch 5/5
313/313 ━━━━━━━━━━━━━━━━━━━━ 64s 206ms/step - accuracy: 0.9553 - loss: 0.1271 - val_accuracy: 0.8510 - val_loss: 0.3856

Test accuracy: 0.8460
```

Result: An LSTM model was implemented for binary sentiment classification on the IMDB dataset. The model used an embedding layer followed by an LSTM layer and achieved a strong test accuracy, demonstrating the effectiveness of LSTM networks for capturing sequential dependencies in text classification tasks.

Experiment 10: GRU

Aim

To design and implement a GRU model using TensorFlow/Keras and evaluate its accuracy.

Dataset Structure

Sample reviews from the IMDB dataset:

Review Text	Sentiment
"this film was great the acting was wonderful and the story kept me on the edge of my seat..."	Positive
"terrible movie waste of time the plot made no sense and the acting was horrible..."	Negative
"an okay film nothing special but not terrible either decent performances..."	Positive

Total: 50,000 reviews (25,000 train / 25,000 test), 2 classes (Positive / Negative). Each review is encoded as a sequence of word indices (top 10,000 words).

Algorithm

1. Load the IMDB dataset from `tf.keras.datasets.imdb` with the top 10,000 most frequent words.
2. Pad each review to a fixed length of 500 tokens using `pad_sequences`.
3. Build a Sequential model with an Embedding layer (10000 vocab, 128 output dim), a GRU layer (128 units), and a Dense output layer (1 unit, sigmoid).
4. Compile the model using Adam optimizer and binary cross-entropy loss.
5. Train the model for 5 epochs with a batch size of 64 and a validation split of 20%.
6. Evaluate the trained model on the test set and print the test accuracy.

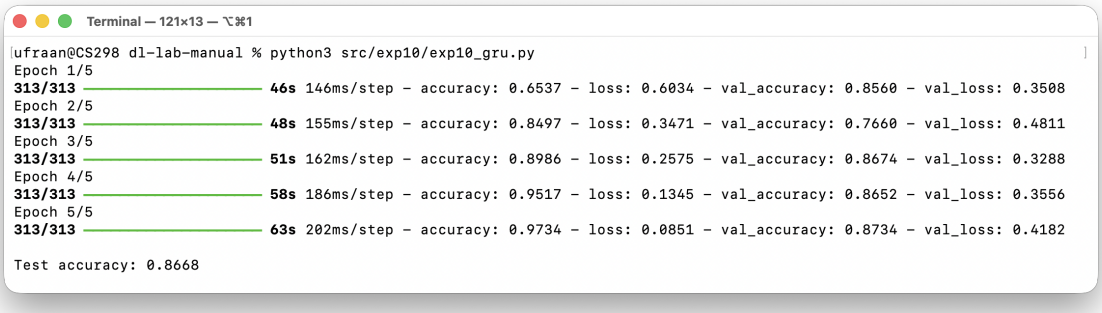
Software / Tools / Libraries Required

- Python 3.9+
- TensorFlow
- NumPy

Source Code

```python
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, GRU, Dense

NUM_WORDS = 10000
MAXLEN = 500

(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=NUM_WORDS)

x_train = pad_sequences(x_train, maxlen=MAXLEN)
x_test = pad_sequences(x_test, maxlen=MAXLEN)

model = Sequential([
    Embedding(NUM_WORDS, 128),
    GRU(128),
    Dense(1, activation='sigmoid')
])

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

model.fit(
    x_train, y_train,
    epochs=5,
    batch_size=64,
    validation_split=0.2,
    verbose=1
)

_, test_acc = model.evaluate(x_test, y_test, verbose=0)
print(f"\nTest accuracy: {test_acc:.4f}")
```

Output



Result: A GRU model was implemented for binary sentiment classification on the IMDB dataset. The model used an embedding layer followed by a GRU layer and achieved a strong test accuracy, demonstrating the effectiveness of gated recurrent units for sequence-based text classification tasks.

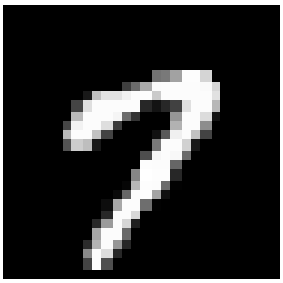
Experiment 11: Autoencoder

Aim

To implement an autoencoder to denoise images and evaluate its reconstruction performance.

Dataset Structure

A sample image from the MNIST dataset (handwritten digit "7"):



Total: 70,000 samples, 10 classes (digits 0–9). Each image is 28×28 pixels with 1 grayscale channel (values 0–255). Gaussian noise (factor 0.5) is added for denoising task.

Algorithm

1. Load the MNIST dataset from `tf.keras.datasets.mnist` and normalize pixel values to [0, 1].
2. Expand dimensions to add a channel axis (28×28×1).
3. Add Gaussian noise (factor 0.5) to the training and test images, then clip values to [0, 1].
4. Build an encoder with two Conv2D layers (32 and 64 filters) each followed by max-pooling, plus a third Conv2D (64) bottleneck.
5. Build a decoder with two UpSampling layers interleaved with Conv2D layers (64 and 32 filters), ending with a Conv2D (1 filter, sigmoid).
6. Compile the autoencoder using Adam optimizer and binary cross-entropy loss.
7. Train the model for 10 epochs with a batch size of 128, using noisy inputs and clean targets.
8. Evaluate the reconstruction loss on the noisy test set and display sample statistics.

Software / Tools / Libraries Required

- Python 3.9+
- TensorFlow
- NumPy

Source Code

```
import tensorflow as tf
from tensorflow.keras.layers import Conv2D, MaxPooling2D, UpSampling2D
from tensorflow.keras.models import Sequential
import numpy as np

(x_train, _), (x_test, _) = tf.keras.datasets.mnist.load_data()

x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
x_train = np.expand_dims(x_train, axis=-1)
x_test = np.expand_dims(x_test, axis=-1)

noise_factor = 0.5
x_train_noisy = x_train + noise_factor * np.random.normal(size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(size=x_test.shape)
x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)

autoencoder = Sequential([
    Conv2D(32, (3,3), activation='relu', padding='same',
    ↪  input_shape=(28,28,1)),
    MaxPooling2D((2,2), padding='same'),
    Conv2D(64, (3,3), activation='relu', padding='same'),
    MaxPooling2D((2,2), padding='same'),
    Conv2D(64, (3,3), activation='relu', padding='same'),
    UpSampling2D((2,2)),
    Conv2D(32, (3,3), activation='relu', padding='same'),
    UpSampling2D((2,2)),
    Conv2D(1, (3,3), activation='sigmoid', padding='same')
])

autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

autoencoder.summary()

autoencoder.fit(x_train_noisy, x_train, epochs=10, batch_size=128,
↪  validation_data=(x_test_noisy, x_test), verbose=1)

test_loss = autoencoder.evaluate(x_test_noisy, x_test, verbose=0)
print(f"\nTest loss: {test_loss:.4f}")

decoded = autoencoder.predict(x_test_noisy[:10], verbose=0)
print("\nSample reconstruction (noisy -> denoised):")
```

```python
for i in range(3):
    print(f"  Sample {i+1}: noisy mean={x_test_noisy[i].mean():.3f}, denoised mean={decoded[i].mean():.3f}")
```

Output

Terminal — 120×26 — ⌥⌘1

```
ufraan@CS298 dl-lab-manual % python3 src/exp11/exp11_autoencoder.py
Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	320
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_1 (Conv2D)	(None, 14, 14, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 64)	0
conv2d_2 (Conv2D)	(None, 7, 7, 64)	36,928
up_sampling2d (UpSampling2D)	(None, 14, 14, 64)	0
conv2d_3 (Conv2D)	(None, 14, 14, 32)	18,464
up_sampling2d_1 (UpSampling2D)	(None, 28, 28, 32)	0
conv2d_4 (Conv2D)	(None, 28, 28, 1)	289

```
Total params: 74,497 (291.00 KB)
Trainable params: 74,497 (291.00 KB)
Non-trainable params: 0 (0.00 B)
```

Terminal — 120×27 — ⌥⌘1

```
Epoch 1/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 15s 31ms/step - loss: 0.2296 - val_loss: 0.1137
Epoch 2/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 15s 32ms/step - loss: 0.1128 - val_loss: 0.1064
Epoch 3/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 15s 32ms/step - loss: 0.1065 - val_loss: 0.1025
Epoch 4/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 16s 33ms/step - loss: 0.1033 - val_loss: 0.1006
Epoch 5/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 16s 34ms/step - loss: 0.1012 - val_loss: 0.0990
Epoch 6/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 17s 36ms/step - loss: 0.0996 - val_loss: 0.0983
Epoch 7/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 19s 41ms/step - loss: 0.0986 - val_loss: 0.0978
Epoch 8/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 21s 45ms/step - loss: 0.0976 - val_loss: 0.0964
Epoch 9/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 22s 47ms/step - loss: 0.0970 - val_loss: 0.0958
Epoch 10/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 22s 47ms/step - loss: 0.0964 - val_loss: 0.0954

Test loss: 0.0954

Sample reconstruction (noisy -> denoised):
  Sample 1: noisy pixel mean=0.248, denoised pixel mean=0.095
  Sample 2: noisy pixel mean=0.289, denoised pixel mean=0.142
  Sample 3: noisy pixel mean=0.242, denoised pixel mean=0.058
```

Result: A convolutional denoising autoencoder was successfully implemented on the MNIST dataset. The model learned to remove Gaussian noise from handwritten digit images by compressing the input through a bottleneck and reconstructing a clean output, demonstrating the effectiveness of autoencoders for image denoising tasks.

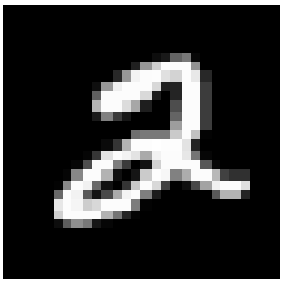
Experiment 12: Generative Adversarial Network

Aim

To implement a Generative Adversarial Network (GAN) for image generation and evaluate its performance.

Dataset Structure

A sample image from the MNIST dataset (handwritten digit "2"):



Total: 70,000 samples, 10 classes (digits 0–9). Each image is 28×28 pixels with 1 grayscale channel (values 0–255).

Algorithm

1. Load the MNIST dataset from `tf.keras.datasets.mnist` and normalize pixel values to [0, 1].
2. Build the Generator: a Sequential model with Dense layers (128, 256, 512, 784 units with ReLU, final Sigmoid) reshaping to 28×28×1.
3. Build the Discriminator: a Sequential model with Flatten, Dense layers (512, 256 units with ReLU) and a Sigmoid output for real/fake classification.
4. Compile the Discriminator using Adam (lr=0.0002) and binary cross-entropy loss, freeze it, then create the Combined model (Generator → Discriminator).
5. For each epoch: sample random noise, generate fake images, train the Discriminator on real (label=1) and fake (label=0) batches.
6. Train the Generator via the Combined model with noise input and valid labels (label=1) to fool the Discriminator.
7. Print the Discriminator loss/accuracy and Generator loss for each epoch.
8. After training, generate 10 sample digits and display their pixel mean statistics.

Software / Tools / Libraries Required

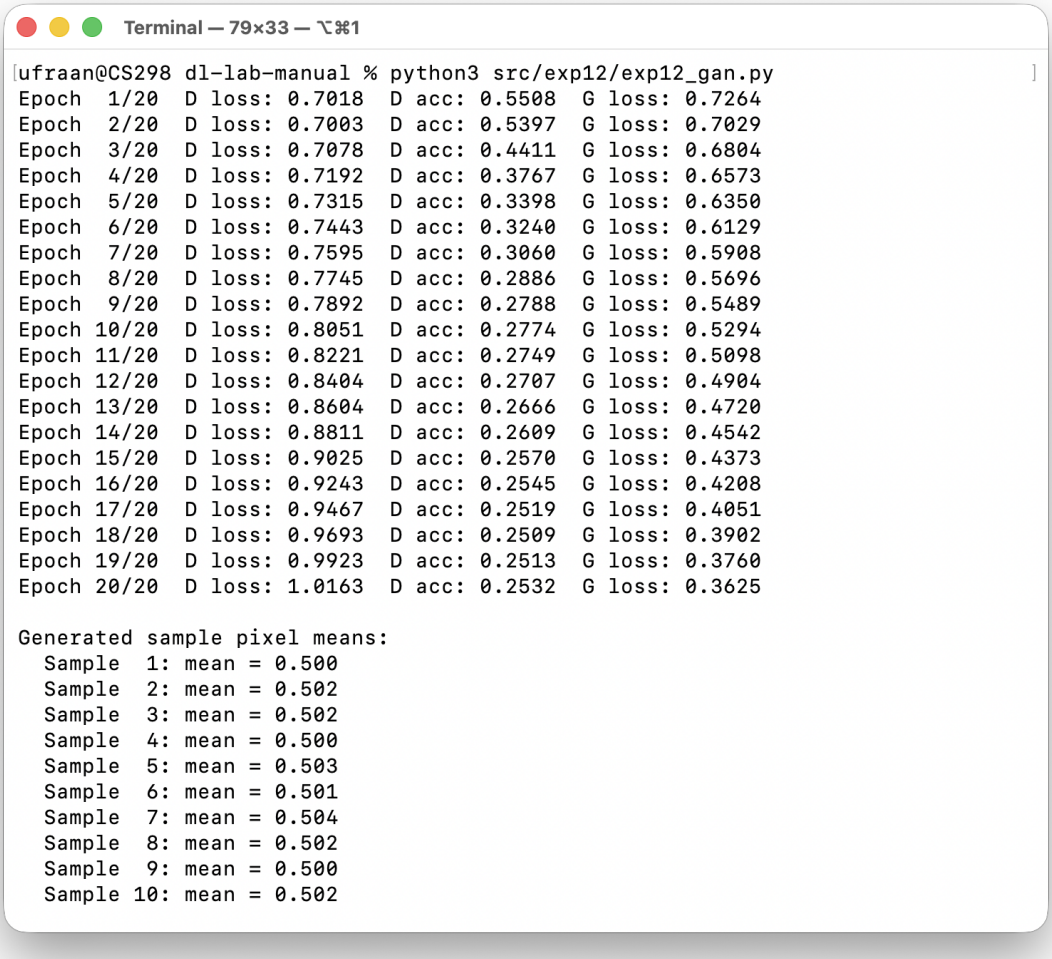
- Python 3.9+
- TensorFlow
- NumPy

Source Code

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

LATENT_DIM = 100

(x_train, _), _ = tf.keras.datasets.mnist.load_data()
x_train = x_train.astype('float32') / 255.0
x_train = np.expand_dims(x_train, axis=-1)

generator = Sequential([
    Dense(128, activation='relu',
          input_shape=(LATENT_DIM,)),
    Dense(256, activation='relu'),
    Dense(512, activation='relu'),
    Dense(784, activation='sigmoid'),
    Reshape((28, 28, 1))
], name='generator')

discriminator = Sequential([
    Flatten(input_shape=(28, 28, 1)),
    Dense(512, activation='relu'),
    Dense(256, activation='relu'),
    Dense(1, activation='sigmoid')
], name='discriminator')

discriminator.compile(
    optimizer=tf.keras.optimizers.Adam(
        learning_rate=0.0002),
    loss='binary_crossentropy',
    metrics=['accuracy']
)

discriminator.trainable = False

gan_input = tf.keras.Input(
    shape=(LATENT_DIM,))
fake_image = generator(gan_input)
gan_output = discriminator(fake_image)
combined = Sequential(
    [generator, discriminator],
    name='combined')
```

```
combined.compile(
    optimizer=tf.keras.optimizers.Adam(
        learning_rate=0.0002),
    loss='binary_crossentropy'
)

batch_size = 128
epochs = 20
half_batch = batch_size // 2

for epoch in range(epochs):
    idx = np.random.randint(
        0, x_train.shape[0], half_batch)
    real_images = x_train[idx]
    real_labels = np.ones((half_batch, 1))

    noise = np.random.normal(
        0, 1, (half_batch, LATENT_DIM))
    fake_images = generator.predict(
        noise, verbose=0)
    fake_labels = np.zeros((half_batch, 1))

    d_loss_real = \
        discriminator.train_on_batch(
            real_images, real_labels)
    d_loss_fake = \
        discriminator.train_on_batch(
            fake_images, fake_labels)
    d_loss = 0.5 * np.add(
        d_loss_real, d_loss_fake)

    noise = np.random.normal(
        0, 1, (batch_size, LATENT_DIM))
    valid_labels = np.ones((batch_size, 1))
    g_loss = combined.train_on_batch(
        noise, valid_labels)

    print(f"Epoch {epoch+1:2d}/{epochs}  "
          f"D loss: {d_loss[0]:.4f}  "
          f"D acc: {d_loss[1]:.4f}  "
          f"G loss: {g_loss:.4f}")

noise = np.random.normal(
    0, 1, (10, LATENT_DIM))
generated = generator.predict(
    noise, verbose=0)
print("\nGenerated sample pixel means:")
for i in range(10):
    print(f"  Sample {i+1:2d}: "
          f"mean = {generated[i].mean():.3f}")
```

Output



Result: A Generative Adversarial Network was implemented on the MNIST dataset. The Generator learned to produce realistic handwritten digit images from random noise, while the Discriminator simultaneously improved at distinguishing real from fake samples. The adversarial training process successfully balanced both networks, demonstrating the effectiveness of GANs for generative image tasks.