

ADVANCED MACHINE LEARNING LAB WITH PYTHON

Course Code: B24DS05PC
BSc Data Science and Analytics
Practical Core Course
Self Learning Material



SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

SREENARAYANAGURU OPEN UNIVERSITY

Vision

To increase access of potential learners of all categories to higher education, research and training, and ensure equity through delivery of high quality processes and outcomes fostering inclusive educational empowerment for social advancement.

Mission

To be benchmarked as a model for conservation and dissemination of knowledge and skill on blended and virtual mode in education, training and research for normal, continuing, and adult learners.

Pathway

Access and Quality define Equity.

Advanced Machine Learning Lab with Python

Course Code: B24DS05PC

Semester - V

Practical Core Course
Undergraduate Programme
BSc Data Science and Analytics
Self Learning Material
(With Model Question Paper Sets)



SREENARAYANAGURU
OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala



SREENARAYANAGURU
OPEN UNIVERSITY

ADVANCED MACHINE LEARNING LAB WITH PYTHON

Course Code: B24DS05PC

Semester- V

Practical Core Course

BSc Data Science and Analytics

Academic Committee

Dr. T. K. Manoj
Dr. Smitha Dharan,
Dr. Satheesh S.
Dr. Vinod Chandra S.S.
Dr. Hari V. S.
Dr. Sharon Susan Jacob
Dr. Ajith Kumar R.
Dr. Smiju I.S.
Dr. Nimitha Aboobaker

Development of the Content

Dr. Chinnu Jacob

Review

Dr. Gopakumar C.

Edit

Dr. Gopakumar C.

Proofreading & Scrutiny

Greeshma P.P.
Dr. Kanitha D.K.
Sreerekha V.K.
Shamin S.
Sub Priya Laxmi S.B.N.
Aswathy V.S.

Cover Design

Jobin J.

Co-ordination

Director (IC), MDDC:
Dr. Sajeekumar G.
Coordinator, Development:
Dr. Anfal M.
Coordinator, Distribution:
Dr. Abdul Razak Kunnathodi



Scan this QR Code for reading the SLM
on a digital device.

Edition
September 2026

Copyright
© Sreenarayanaguru Open University

ISBN 978-81-180900-0-7



All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from Sreenarayanaguru Open University. Printed and published on behalf of Sreenarayanaguru Open University by Registrar, SGOU, Kollam.

www.sgou.ac.in



Visit and Subscribe our Social Media Platforms

Dear learner,

I extend my heartfelt greetings and profound enthusiasm as I warmly welcome you to Sreenarayanaguru Open University. Established in September 2020 as a state-led endeavour to promote higher education through open and distance learning modes, our institution was shaped by the guiding principle that access and quality are the cornerstones of equity. We have firmly resolved to uphold the highest standards of education, setting the benchmark and charting the course.

The courses offered by the Sreenarayanaguru Open University aim to strike a quality balance, ensuring students are equipped for both personal growth and professional excellence. The University embraces the widely acclaimed "blended format," a practical framework that harmoniously integrates Self-Learning Materials, Classroom Counseling, and Virtual modes, fostering a dynamic and enriching experience for both learners and instructors.

The University is committed to providing an engaging and dynamic educational environment that encourages active learning. The Study and Learning Material (SLM) is specifically designed to offer you a comprehensive and integrated learning experience, fostering a strong interest in exploring advancements in information technology (IT). The curriculum has been carefully structured to ensure a logical progression of topics, allowing you to develop a clear understanding of the evolution of the discipline. It is thoughtfully curated to equip you with the knowledge and skills to navigate current trends in IT, while fostering critical thinking and analytical capabilities. The Self-Learning Material has been meticulously crafted, incorporating relevant examples to facilitate better comprehension.

Rest assured, the university's student support services will be at your disposal throughout your academic journey, readily available to address any concerns or grievances you may encounter. We encourage you to reach out to us freely regarding any matter about your academic programme. It is our sincere wish that you achieve the utmost success.



Regards,
Dr. Jagathy Raj V. P.

01-09-2026

Contents

Introduction		2
CYCLE I	Neural Networks Fundamentals and Deep Learning Techniques	12
Experiment 1	Implement a Single Layer Perceptron for binary classification	13
Experiment 2	Implement a Multi-Layer Perceptron (MLP) using a Simple Dataset. Train an MLP with Backpropagation and Analyze its Performance	17
Experiment 3	Study the Effects of Overfitting in Neural Networks using Various Regularization Techniques	21
Experiment 4	Implement a Deep Feedforward Network and Explore Different Optimization Techniques	27
Experiment 5	Experiment with Gradient Descent (GD) and its Variants on a Complex Dataset	33
Experiment 6	Apply regularization techniques (L1, L2, Early Stopping) to a deep learning model	38
CYCLE II	Convolutional and Recurrent Neural Networks	43
Experiment 1	Build a Convolutional Neural Network (CNN) and evaluate its performance on image classification	44
Experiment 2	Implement a Recurrent Neural Network (RNN) for sequence prediction	48
Experiment 3	Compare the performance of LSTM and GRU networks on a time-series dataset	53
Experiment 4	Experiment with a simple Generative Adversarial Network (GAN) for image generation	58

```
#include "KMotionDef.h"
```

```
int main()
```

```
{
```

```
    ch0->Amp = 250;
```

```
    ch0->output_mode=MICROSTEP_MODE;
```

```
    ch0->Vel=70.0f;
```

```
    ch0->Accel=500.0f;
```

```
    ch0->Jerk =2000f;
```

```
    ch0->Lead=0.0f;
```

```
    EnableAxisDest(0,0);
```

```
    ch1->Amp = 250;
```

```
    ch1->output_mode=MICROSTEP_MODE;
```

```
    ch1->Vel=70.0f;
```

```
    ch1->Accel=500.0f;
```

```
    ch1->Jerk =2000f;
```

```
    ch1->Lead=0.0f;
```

```
    EnableAxisDest(1,0);
```

```
    DefineCoordSystem(0,1,-1,-1);
```

```
    return 0;
```

```
}
```

Lab Manual

Introduction

The Advanced Machine Learning Lab with Python supports the Advanced Machine Learning course through a sequence of laboratory experiments on neural networks and deep learning models.

The laboratory emphasizes implementation, model training, performance evaluation, and interpretation of experimental results using Python.

The laboratory consists of two cycles.

- ◆ Cycle I covers the implementation of feedforward neural networks, backpropagation, optimization methods, gradient descent algorithms, overfitting, and regularization techniques. These experiments introduce neural network training and the factors that influence model performance.
- ◆ Cycle II covers neural network architectures developed for specific learning tasks. The experiments include Convolutional Neural Networks for image classification, Recurrent Neural Networks for sequence modelling, Long Short-Term Memory and Gated Recurrent Unit networks for time-series analysis, and Generative Adversarial Networks for image generation. Students examine the architecture, training procedure, and evaluation of each model through practical implementation.

Python is the implementation platform for all experiments. The laboratory uses standard scientific computing and deep learning libraries for dataset preparation, model development, visualization, and performance analysis. Each experiment concludes with an analysis of the results obtained during model training and testing.

Cycle I: Neural Networks Fundamentals and Deep Learning Techniques

Cycle I introduces the basic concepts of neural networks and deep learning. The experiments begin with the Single Layer Perceptron for binary classification. The next experiment is the Multi-Layer Perceptron, where students learn the backpropagation algorithm and the training process of a neural network.

The remaining experiments focus on model training and performance. Students study overfitting, optimization techniques, gradient descent algorithms, and regularization methods. They also implement a deep feedforward neural network and compare different optimization methods during training.

By the end of this cycle, students should be able to:

- ◆ Implement a Single Layer Perceptron for binary classification.
- ◆ Train a Multi-Layer Perceptron using backpropagation.
- ◆ Compare different gradient descent algorithms.
- ◆ Apply optimization techniques during model training.

- ◆ Identify overfitting in a neural network.
- ◆ Apply L1 Regularization, L2 Regularization, and Early Stopping.
- ◆ Evaluate the performance of deep learning models using suitable performance measures.

Cycle II: Convolutional and Recurrent Neural Networks

Cycle II covers neural network models used for image data and sequence data. The first experiment is based on Convolutional Neural Networks (CNNs). Students build a CNN model and use it for image classification. The experiment includes convolution, pooling, activation functions, and fully connected layers.

The next experiments cover Recurrent Neural Networks (RNNs). Students implement an RNN model for sequence prediction. They also compare the performance of Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) models using a time-series dataset. The comparison helps students understand the difference between these recurrent models.

The last experiment introduces a simple Generative Adversarial Network (GAN). Students train the generator and discriminator models together and observe the generated images after training. The experiment includes a basic evaluation of the generated images.

At the end of Cycle II, students should be able to:

- ◆ Build a Convolutional Neural Network for image classification.
- ◆ Implement a Recurrent Neural Network for sequence prediction.
- ◆ Compare the performance of LSTM and GRU models.
- ◆ Train a simple Generative Adversarial Network.
- ◆ Evaluate the performance of CNN, RNN, LSTM, GRU, and GAN models using suitable performance measures.
- ◆ Interpret the experimental results obtained from different deep learning models.

Objectives of the Advanced Machine Learning Lab

The objectives of the Advanced Machine Learning Lab are to:

- ◆ Understand the architecture and working of artificial neural networks.
- ◆ Implement deep learning models using Python and standard deep learning libraries.
- ◆ Train Single Layer Perceptron and Multi-Layer Perceptron models for classification problems.



- ◆ Apply backpropagation for neural network training.
- ◆ Compare gradient descent and optimization algorithms used in deep learning.
- ◆ Study the effect of overfitting and apply regularization techniques to improve model performance.
- ◆ Build Convolutional Neural Network models for image classification.
- ◆ Implement Recurrent Neural Network, Long Short-Term Memory, and Gated Recurrent Unit models for sequence and time-series data.
- ◆ Develop a simple Generative Adversarial Network and study its training process.
- ◆ Evaluate deep learning models using suitable performance measures and interpret the experimental results.
- ◆ Develop programming and analytical skills through laboratory experiments.

Tools and Software Used

The Advanced Machine Learning Lab uses Python and standard deep learning frameworks for model development, training, and evaluation. The following software and libraries are used in the laboratory.

Programming Language

◆ Integrated Development Environments (IDEs)

- Jupyter Notebook : Interactive environment for writing and testing deep learning programs.
- Google Colab : Cloud-based notebook with optional GPU support.
- Visual Studio Code (VS Code) : Code editor for Python development and project management.
- PyCharm : IDE for developing and debugging Python applications.

◆ Deep Learning Frameworks

- TensorFlow : Framework used for building and training deep learning models.
- Keras : High-level API for developing neural network models with TensorFlow.
- PyTorch : Framework used for research, experimentation, and custom neural network implementation.

◆ Python Libraries

- NumPy : Numerical computing and array operations.

- Pandas : Dataset loading and data preprocessing.
- Matplotlib : Plotting graphs and learning curves.
- Scikit-learn : Dataset preparation, preprocessing, and model evaluation.
- OpenCV : Image processing for computer vision experiments.

◆ **Hardware Support**

- GPU (Optional) : Used to reduce the training time of deep learning models.

◆ **Version Control**

- Git : Version control for source code management.
- GitHub / GitLab : Repository hosting and collaboration.

Scope of the Lab

The Advanced Machine Learning Lab is based on the implementation of deep learning models using Python. The laboratory covers neural networks, optimization methods, regularization techniques, Convolutional Neural Networks, Recurrent Neural Networks, Long Short-Term Memory networks, Gated Recurrent Unit networks, and Generative Adversarial Networks. The experiments use standard datasets for model training and testing. Students perform model implementation, training, performance evaluation, and result analysis as part of each experiment. The laboratory is intended to support the Advanced Machine Learning course through practical work.

Expectations from Students

- ◆ Actively participate in all lab sessions
- ◆ Prepare for lab sessions by studying the relevant theory
- ◆ Complete and submit lab records and assignments on time
- ◆ Maintain originality and follow ethical coding practices
- ◆ Collaborate constructively with peers while ensuring individual understanding
- ◆ Demonstrate conceptual and practical knowledge during viva examinations

Lab Guidelines and Code of Conduct

- ◆ Follow instructions provided in the lab manual and by the instructor
- ◆ Use only authorized software tools and datasets
- ◆ Avoid plagiarism and submit original work
- ◆ Maintain discipline and proper lab conduct



- ◆ Save and back up programs regularly

Violation of lab rules may result in:

- ◆ Warnings
- ◆ Deduction of marks
- ◆ Suspension from lab sessions

Students should show respect to instructors, lab assistants, and peers, avoid disturbing others, seek help when needed, and focus on developing strong analytical and problem solving skills.

Installation and Configuration of Python

Step 1: Install Python

Download Python from the official Python website.

<https://www.python.org>

- ◆ Install a Python version compatible with the TensorFlow version used in the laboratory.
- ◆ Select Add Python to PATH during installation.

Step 2: Verify Installation

Open Command Prompt / Terminal and run:

```
python --version
```

The installed Python version is displayed if the installation is complete.

Step 3: Install required Libraries

Install the libraries used in the laboratory.

```
pip install numpy pandas matplotlib scikit-learn tensorflow keras torch  
torchvision jupyter
```

If OpenCV is required, install it using:

```
pip install opencv-python
```

Step 4: Verify the Libraries

Open Python and import the required packages.

```
Import numpy
import pandas
import matplotlib
import sklearn
import tensorflow
import keras
import torch
```

If no error message is displayed, the installation is complete.

Common IDEs for Deep Learning Development

The laboratory programs can be developed using any one of the following software environments.

◆ Jupyter Notebook

- Interactive environment for writing and executing Python programs.
- Suitable for data preprocessing, model training, visualization, and result analysis.

◆ Google Colab

- Cloud-based Jupyter Notebook environment.
- Supports GPU acceleration for deep learning experiments.
- No local installation is required.

◆ Visual Studio Code (VS Code)

- Code editor with Python support.
- Suitable for developing and debugging deep learning programs.
- Supports virtual environments and extensions.

◆ PyCharm

- Integrated Development Environment (IDE) for Python.
- Suitable for developing large projects and debugging applications.

Datasets used in the Advanced Machine Learning Lab

Datasets are an essential part of deep learning experiments. The performance of a neural network depends on the quality of the dataset used for training and testing. The laboratory uses standard datasets for implementing and evaluating different neural network models.

Types of Datasets

- ◆ Image datasets
- ◆ Time-series datasets
- ◆ Tabular datasets

Commonly used Datasets

- ◆ MNIST Dataset : Handwritten digit classification and image generation.
- ◆ Fashion-MNIST Dataset : Clothing image classification.
- ◆ CIFAR-10 Dataset : Object image classification.
- ◆ Digits Dataset : Handwritten digit classification using a small image dataset.
- ◆ AirPassengers Dataset : Monthly passenger-count prediction using recurrent neural networks.
- ◆ Daily Minimum Temperatures Dataset : Time-series forecasting using LSTM and GRU networks.
- ◆ Breast Cancer Wisconsin Dataset : Binary classification using neural networks.

Dataset Collection and Preparation

Steps in Dataset Preparation

- ◆ Select a suitable dataset.
- ◆ Verify the completeness of the dataset.
- ◆ Check the format of input and output data.

Data Preprocessing Techniques

- ◆ Handling missing values.
- ◆ Encoding categorical data.
- ◆ Feature scaling or normalization.
- ◆ Image resizing and pixel normalization.
- ◆ Splitting the dataset into training, validation, and testing sets.

Structure of Deep Learning Programs

Each laboratory program should follow a common structure. This helps in organizing the implementation and makes the results easier to verify and document.

General Program Structure

1. Import the required Python libraries.
2. Load the dataset.
3. Preprocess the dataset.
4. Prepare the training, validation, and test data as required.
5. Build the neural network model.
6. Configure the loss function and optimizer.
7. Train the model.
8. Test the trained model, where applicable.
9. Evaluate the model using suitable performance measures.
10. Display the results using appropriate output or visualizations.
11. Record the observations and output.

Program Development Guidelines

- ◆ Use meaningful variable names.
- ◆ Keep the program organized into logical sections.
- ◆ Add comments where necessary.
- ◆ Save the trained model, if required.
- ◆ Verify the output before submission.
- ◆ Record the parameters used for model training.

Result Documentation

Each experiment should include the following:

- ◆ Experiment title.
- ◆ Objective.

- ◆ Dataset used.
- ◆ Python program.
- ◆ Output.
- ◆ Result.
- ◆ Observation.

Use of Jupyter Notebook

Jupyter Notebook is the primary environment for the Advanced Machine Learning laboratory experiments.

Advantages

- ◆ Interactive code execution
- ◆ Step-by-step visualization of outputs
- ◆ Easy debugging and experimentation
- ◆ Markdown support for documentation
- ◆ Suitable for data preprocessing, model training, and result visualization

Each notebook should include:

- ◆ Experiment title
- ◆ Objective
- ◆ Dataset description
- ◆ Importing required libraries
- ◆ Data preprocessing
- ◆ Model implementation
- ◆ Output and result analysis

Use of Google Colab Notebook

Google Colab (Collaboratory) is a cloud-based Jupyter Notebook environment that allows users to write and execute Python programs through a web browser. It can be used as an alternative to a local Python environment for the laboratory experiments.

Advantages of Google Colab

- ◆ No installation required
- ◆ Accessible from any device with an internet connection

- ◆ Pre-installed Python and deep learning libraries
- ◆ Supports GPU and TPU acceleration
- ◆ Easy sharing and collaboration
- ◆ Automatic saving to Google Drive

Structure of a Google Colab Notebook

- ◆ Title of the experiment
- ◆ Objective
- ◆ Dataset description
- ◆ Importing required libraries
- ◆ Dataset loading and preprocessing
- ◆ Model implementation
- ◆ Model training
- ◆ Output and visualization
- ◆ Result and observation

Dataset Handling in Google Colab

- ◆ Upload datasets from the local system
- ◆ Load datasets from Google Drive
- ◆ Download datasets from online repositories

If additional Python libraries are required, they can be installed using:

```
pip install library_name
```

```
#include "KMotionDef.h"
```

```
int main()
```

```
{
```

```
ch0->Amp = 250;
```

```
ch0->output_mode=MICROSTEP_MODE;
```

```
ch0->Vel=70.0f;
```

```
ch0->Amp=250;
```

```
ch0->Vel=70.0f;
```

```
ch0->L_x=0.0f;
```

```
EnableAxisDest(0,0);
```

```
ch1->Amp = 250;
```

```
ch1->output_mode=MICROSTEP_MODE;
```

```
ch1->Vel=70.0f;
```

```
ch1->Amp=250;
```

```
ch1->Vel=70.0f;
```

```
ch1->L_x=0.0f;
```

```
ch1->L_y=0.0f;
```

```
EnableAxisDest(0,0);
```

```
DefineCoordSystem(0,1,1,-1);
```

```
return 0;
```

```
}
```

CYCLE I

Neural Networks Fundamentals and Deep Learning Techniques

Experiment 1

Implement a Single Layer Perceptron for binary classification

Objectives

- ◆ Implement a Single Layer Perceptron using Python.
- ◆ Train the perceptron for a binary classification problem.
- ◆ Predict the class label of unseen samples.
- ◆ Evaluate the performance of the trained model.
- ◆ Interpret the classification results.

1.1.1 Theory

A Single Layer Perceptron is a neural network model used for binary classification. It contains one input layer and one output neuron. Each input has a corresponding weight. A bias value is added to the weighted sum before the output is calculated.

The output depends on the weighted sum of the inputs. A step activation function is used to assign the output to one of the two classes. The weights are updated based on the difference between the target output and the predicted output.

A Single Layer Perceptron works only with linearly separable data. It cannot classify data that requires a non-linear decision boundary. Such problems are solved using Multi-Layer Perceptron models.

For an input vector $X=(x_1,x_2,\dots,x_n)$, the net input is

$$z = \sum_{i=1}^n w_i x_i + b$$

where

- ◆ x_i is the input value.
- ◆ w_i is the corresponding weight.
- ◆ b is the bias.

The output is obtained using the step activation function.

$$y = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases}$$

The weights are updated using

$$w_i = w_i + \eta(t - y) x_i$$

where

- ◆ w_i is the corresponding weight.
- ◆ η is the learning rate.
- ◆ t is the target output.
- ◆ y is the predicted output.
- ◆ x_i is the i th input feature value.

The updated weights are used in the next training iteration.

1.1.2 Implementation

1. Implement a Single Layer Perceptron using the Breast Cancer Wisconsin dataset. Split the dataset into training and testing sets. Train the model and evaluate its performance using classification accuracy and a confusion matrix.

```
from sklearn.datasets import load_breast_cancer

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.linear_model import Perceptron

from sklearn.metrics import accuracy_score, confusion_matrix

# Load dataset

data = load_breast_cancer()

X = data.data

y = data.target

# Split dataset

X_train, X_test, y_train, y_test = train_test_split(
```

```
X, y, test_size=0.2, random_state=42
)

# Feature scaling

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)

X_test = scaler.transform(X_test)

# Train Perceptron model

model = Perceptron(max_iter=1000, random_state=42)

model.fit(X_train, y_train)

# Prediction

y_pred = model.predict(X_test)

# Evaluation

print("Accuracy :", accuracy_score(y_test, y_pred))

print("\nConfusion Matrix")

print(confusion_matrix(y_test, y_pred))
```

OUTPUT

Accuracy : 0.9736842105263158

Confusion Matrix

[[42 1]

[2 69]]

1.1.3 Lab Practice Questions

1. Load the Breast Cancer Wisconsin dataset and:
 - a. Split the dataset into training and testing sets.

- b. Train a Single Layer Perceptron model.
 - c. Display the classification accuracy.
 - d. Display the confusion matrix.
2. Using the trained Perceptron model:
- a. Predict the class labels for the test data.
 - b. Display the first five predicted class labels.
 - c. Compare the predicted labels with the corresponding actual labels.

SGOU

Experiment 2

Implement a Multi-Layer Perceptron (MLP) using a Simple Dataset. Train an MLP with Backpropagation and Analyze its Performance

Objectives

- ◆ Implement a Multi-Layer Perceptron using Python.
- ◆ Train an MLP using the backpropagation algorithm.
- ◆ Classify data using a neural network.
- ◆ Evaluate the performance of the trained model.
- ◆ Interpret the classification results.

1.2.1 Theory

A Multi-Layer Perceptron (MLP) is a neural network with one or more hidden layers. It is used for classification and prediction problems where a Single Layer Perceptron is not sufficient.

The network has an input layer, one or more hidden layers, and an output layer. Neurons in one layer are connected to the next layer through weights. A bias value is also used in the computation.

Each neuron calculates a weighted sum of its inputs. The result is passed through an activation function. The output from one layer becomes the input to the next layer.

The network is trained using backpropagation. The error between the predicted output and the expected output is calculated, and the gradients are propagated backward through the network to update the weights.

ReLU, Sigmoid, and Tanh are commonly used activation functions in MLP models. The activation function is selected based on the problem being solved.

For an input vector $X=(x_1,x_2,\dots,x_n)$, the net input is

$$z = \sum_{i=1}^n w_i x_i + b$$

where

- ◆ w_i is the corresponding weight.



- ◆ x_i is the input value.
- ◆ b is the bias.

The output of the neuron is

$$y = f(z)$$

where f is the activation function and z is the weighted sum of inputs plus bias.

The weights are updated after each training step. The same process is repeated until training is completed.

1.2.2 Implementation

1. Implement a Multi-Layer Perceptron (MLP) using the Breast Cancer Wisconsin dataset. Train the model using backpropagation and evaluate its performance using classification accuracy.

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Dense

# Load dataset
data = load_breast_cancer()

X = data.data

y = data.target

# Split dataset
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Feature scaling
```

```

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)

X_test = scaler.transform(X_test)

# Build MLP model

model = Sequential([

    Input(shape=(X_train.shape[1],)),

    Dense(16, activation="relu"),

    Dense(8, activation="relu"),

    Dense(1, activation="sigmoid")

])

# Compile model

model.compile(

    optimizer="adam",

    loss="binary_crossentropy",

    metrics=["accuracy"]

)

# Train model

model.fit(

    X_train,

    y_train,

    epochs=50,

    batch_size=16,

    verbose=0

)

```

```
# Prediction

y_pred = (model.predict(X_test, verbose=0) > 0.5).astype(int)

# Evaluation

accuracy = accuracy_score(y_test, y_pred)

print("Accuracy :", accuracy)
```

OUTPUT

```
Accuracy : 0.9824561403508771
```

1.2.3 Lab Practice Questions

1. Load the Breast Cancer Wisconsin dataset and:

- Split the dataset into training and testing sets.
- Scale the input features.
- Build a Multi-Layer Perceptron model.
- Train the model using backpropagation.
- Display the classification accuracy.

2. Using the trained MLP model:

- Predict the class labels for the test data.
- Display the first five predicted class labels.
- Compare the predicted labels with the corresponding actual labels.

Experiment 3

Study the Effects of Overfitting in Neural Networks using Various Regularization Techniques

Objectives

- ◆ Identify overfitting in a neural network.
- ◆ Compare training and validation performance.
- ◆ Apply regularization techniques to reduce overfitting.
- ◆ Evaluate the performance of the regularized model.
- ◆ Compare the results before and after regularization.

1.3.1 Theory

Overfitting is a common problem in neural networks. A model affected by overfitting gives high accuracy on the training data but performs poorly on unseen data. The model learns the training samples very closely and fails to generalize well.

Training accuracy and validation accuracy are used to identify overfitting. When the training accuracy continues to increase while the validation accuracy remains unchanged or starts to decrease, the model is likely to be overfitted.

Regularization methods are used to control overfitting. These methods reduce unnecessary learning and improve the performance of the model on new data.

Some of the commonly used regularization methods are:

1. Dropout - Removes a fraction of neurons during training.
2. L1 Regularization - Applies a penalty to the absolute value of the weights.
3. L2 Regularization - Applies a penalty to the squared value of the weights.
4. Early Stopping - Stops the training process when the validation performance no longer improves.

In this experiment, the performance of the network is observed before and after applying regularization. The comparison is made using training accuracy and validation accuracy.

1.3.2 Implementation

1. Build a Multi-Layer Perceptron (MLP) using the Fashion-MNIST dataset. Train the model without regularization and compare its performance with a model using Dropout. Compare the training accuracy and validation accuracy of both models.

```
import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Input, Dense, Dropout, Flatten

from sklearn.model_selection import train_test_split

# Load Fashion-MNIST dataset

(X, y), (X_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()

# Split training data into training and validation sets

X_train, X_val, y_train, y_val = train_test_split(

    X, y,

    test_size=0.2,

    random_state=42

)

# Normalize pixel values

X_train = X_train / 255.0

X_val = X_val / 255.0

# Model without regularization

base_model = Sequential([

    Input(shape=(28, 28)),

    Flatten(),

    Dense(256, activation="relu"),

    Dense(128, activation="relu"),
```

```

    Dense(10, activation="softmax")
])

base_model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)

# Model with Dropout

dropout_model = Sequential([
    Input(shape=(28, 28)),
    Flatten(),
    Dense(256, activation="relu"),
    Dropout(0.5),
    Dense(128, activation="relu"),
    Dense(10, activation="softmax")
])

dropout_model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)

# Train models

base_model.fit(
    X_train, y_train,

```

```

    epochs=30,

    batch_size=128,

    verbose=0
)

dropout_model.fit(

    X_train, y_train,

    epochs=30,

    batch_size=128,

    verbose=0
)

# Evaluate base model

base_train_loss, base_train_acc = base_model.evaluate(

    X_train, y_train, verbose=0
)

base_val_loss, base_val_acc = base_model.evaluate(

    X_val, y_val, verbose=0
)

# Evaluate dropout model

drop_train_loss, drop_train_acc = dropout_model.evaluate(

    X_train, y_train, verbose=0
)

drop_val_loss, drop_val_acc = dropout_model.evaluate(

    X_val, y_val, verbose=0
)

```

```
# Display results

print("\nModel Performance\n")

print("Without Regularization")

print("Training Accuracy : {:.2f}%".format(base_train_acc * 100))

print("Validation Accuracy : {:.2f}%".format(base_val_acc * 100))

print("\nDropout")

print("Training Accuracy : {:.2f}%".format(drop_train_acc * 100))

print("Validation Accuracy : {:.2f}%".format(drop_val_acc * 100))
```

OUTPUT

```
Model Performance

Without Regularization

Training Accuracy : 96.19%

Validation Accuracy : 89.45%

Dropout

Training Accuracy : 92.48%

Validation Accuracy : 89.47%
```

1.3.3 Lab Practice Questions

1. **Build a Multi-Layer Perceptron (MLP) using the Fashion-MNIST dataset and:**
 - a. Train the model without regularization.
 - b. Display the training accuracy and validation accuracy.
 - c. Identify whether the model shows signs of overfitting.

2. Modify the MLP model by applying the following regularization techniques:

- a. L1 Regularization
- b. L2 Regularization
- c. Compare the training and validation accuracy with the model without regularization and identify which method reduces the training-validation accuracy gap.

SGOU

Experiment 4

Implement a Deep Feedforward Network and Explore Different Optimization Techniques

Objectives

- ◆ Build a deep feedforward neural network using Python.
- ◆ Train the network using different optimization techniques.
- ◆ Compare the performance of different optimizers.
- ◆ Evaluate the trained models using classification accuracy.
- ◆ Select an appropriate optimizer based on the obtained results.

1.4.1 Theory

A deep feedforward network is a neural network with multiple hidden layers. The data moves from the input layer to the output layer through the hidden layers. There are no feedback connections between the layers.

The performance of a neural network depends on the optimizer used during training. The optimizer updates the weights after each iteration and helps reduce the loss function.

Gradient Descent is the basic optimization method. Several improved optimizers are available for deep learning models. These optimizers differ in the way the weights are updated during training.

Some of the commonly used optimization techniques are:

- ◆ **SGD** updates the weights using the gradients calculated from training batches.
- ◆ **RMSprop** adjusts the learning rate using the moving average of squared gradients.
- ◆ **Adam** combines momentum and adaptive learning rates during weight updates.

In this experiment, the same deep feedforward network is trained using different optimizers. The performance of each model is compared using classification accuracy.

1.4.2 Implementation

1. Implement a deep feedforward neural network using the MNIST dataset. Train the model using different optimization techniques and compare the classification accuracy.

```
# Import required libraries
import tensorflow as tf

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Input
from tensorflow.keras.optimizers import SGD, RMSprop, Adam

# Load the MNIST dataset
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()

# Normalize the pixel values
X_train = X_train / 255.0
X_test = X_test / 255.0

# Model using SGD Optimizer
sgd_model = Sequential([
    Input(shape=(28, 28)),
    Flatten(),
    Dense(256, activation='relu'),
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])

sgd_model.compile(
    optimizer=SGD(),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

```

# Model using RMSprop Optimizer
rmsprop_model = Sequential([
    Input(shape=(28, 28)),
    Flatten(),
    Dense(256, activation='relu'),
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
rmsprop_model.compile(
    optimizer=RMSprop(),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# Model using Adam Optimizer
adam_model = Sequential([
    Input(shape=(28, 28)),
    Flatten(),
    Dense(256, activation='relu'),
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
adam_model.compile(
    optimizer=Adam(),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

```

```

# Train the model using SGD Optimizer
sgd_model.fit(
    X_train,
    y_train,
    epochs=10,
    batch_size=128,
    verbose=0
)

# Train the model using RMSprop Optimizer
rmsprop_model.fit(
    X_train,
    y_train,
    epochs=10,
    batch_size=128,
    verbose=0
)

# Train the model using Adam Optimizer
adam_model.fit(
    X_train,
    y_train,
    epochs=10,
    batch_size=128,
    verbose=0
)

# Evaluate the models
sgd_loss, sgd_accuracy = sgd_model.evaluate(
    X_test,
    y_test,

```

```

    verbose=0
)
rmsprop_loss, rmsprop_accuracy = rmsprop_model.evaluate(
    X_test,
    y_test,
    verbose=0
)
adam_loss, adam_accuracy = adam_model.evaluate(
    X_test,
    y_test,
    verbose=0
)
# Display the results
print("Model Performance\n")
print("SGD Optimizer")
print("Accuracy : {:.2f}%".format(sgd_accuracy * 100))
print("\nRMSprop Optimizer")
print("Accuracy : {:.2f}%".format(rmsprop_accuracy * 100))
print("\nAdam Optimizer")
print("Accuracy : {:.2f}%".format(adam_accuracy * 100))

```

OUTPUT

```

Model Performance
SGD Optimizer
Accuracy : 94.48%
RMSprop Optimizer
Accuracy : 97.97%
Adam Optimizer
Accuracy : 97.77%

```

1.4.3 Lab Practice Questions

1. Implement the deep feedforward network using the Adagrad optimizer and evaluate its classification accuracy.
2. Modify the SGD optimizer by adding momentum and compare its classification accuracy with the standard SGD optimizer.

SGOU

Experiment 5

Experiment with Gradient Descent (GD) and its Variants on a Complex Dataset

Objectives

- ◆ Implement a deep learning model using Gradient Descent.
- ◆ Train the model using different Gradient Descent variants.
- ◆ Compare the performance of the trained models.
- ◆ Evaluate the classification accuracy.
- ◆ Analyze the effect of different Gradient Descent variants.

1.5.1 Theory

Gradient Descent is an optimization method used to train neural networks. It updates the model weights in the direction that reduces the loss function.

The main variants of Gradient Descent differ in the number of training samples used to calculate each weight update.

Some commonly used variants are:

- ◆ Batch Gradient Descent uses the entire training dataset to calculate each weight update.
- ◆ Mini-Batch Gradient Descent uses a small batch of training samples for each weight update.
- ◆ Stochastic Gradient Descent (SGD) uses one training sample for each weight update.

In this experiment, the same neural network is trained using Batch Gradient Descent, Mini-Batch Gradient Descent, and Stochastic Gradient Descent. The performance of each method is compared using classification accuracy.

1.5.2 Implementation

1. Implement a deep feedforward neural network using the Digits dataset. Train the model using Batch Gradient Descent, Mini-Batch Gradient Descent, and Stochastic Gradient Descent. Compare the classification accuracy obtained for each method.

```

from sklearn.datasets import load_digits

from sklearn.model_selection import train_test_split

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Input, Flatten, Dense

from tensorflow.keras.optimizers import SGD

# Load Digits dataset

data = load_digits()

X = data.images

y = data.target

# Normalize pixel values

X = X / 16.0

# Split the dataset

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

# ----- Batch Gradient Descent -----

batch_model = Sequential([
    Input(shape=(8, 8)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])

batch_model.compile(
    optimizer=SGD(learning_rate=0.01),

```

```

        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )
    batch_model.fit(
        X_train,
        y_train,
        epochs=20,
        batch_size=len(X_train),
        verbose=0
    )

# ----- Mini-Batch Gradient Descent -----
mini_batch_model = Sequential([
    Input(shape=(8, 8)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
mini_batch_model.compile(
    optimizer=SGD(learning_rate=0.01),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
mini_batch_model.fit(
    X_train,
    y_train,
    epochs=20,
    batch_size=32,

```

```

    verbose=0
)
# ----- Stochastic Gradient Descent -----
sgd_model = Sequential([
    Input(shape=(8, 8)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
sgd_model.compile(
    optimizer=SGD(learning_rate=0.01),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
sgd_model.fit(
    X_train,
    y_train,
    epochs=20,
    batch_size=1,
    verbose=0
)
# ----- Evaluation -----
batch_loss, batch_accuracy = batch_model.evaluate(
    X_test, y_test, verbose=0
)
mini_batch_loss, mini_batch_accuracy = mini_batch_model.evaluate(
    X_test, y_test, verbose=0
)

```

```

)
sgd_loss, sgd_accuracy = sgd_model.evaluate(
    X_test, y_test, verbose=0
)
# Display results
print("Model Performance\n")
print("Batch Gradient Descent")
print("Accuracy :", round(batch_accuracy * 100, 2), "%")
print("\nMini-Batch Gradient Descent")
print("Accuracy :", round(mini_batch_accuracy * 100, 2), "%")
print("\nStochastic Gradient Descent")
print("Accuracy :", round(sgd_accuracy * 100, 2), "%")

```

OUTPUT

```

Model Performance

Batch Gradient Descent

Accuracy : 18.06 %

Mini-Batch Gradient Descent

Accuracy : 91.39 %

Stochastic Gradient Descent

Accuracy : 98.06 %

```

1.5.3 Lab Practice Questions

1. Change the batch size used for Mini-Batch Gradient Descent and observe the classification accuracy.
2. Change the learning rate and compare the classification accuracy of the three Gradient Descent variants.

Experiment 6

Apply regularization techniques (L1, L2, Early Stopping) to a deep learning model

Objectives

- ◆ To apply L1 and L2 regularization to a deep learning model and compare their effect on model performance.
- ◆ To implement early stopping and see how it prevents unnecessary overfitting.
- ◆ To compare a baseline model against L1, L2, and early stopping variants.

1.6.1 Theory

L1 regularization adds the sum of the absolute values of the weights to the loss function. It tends to push some weights exactly to zero, which has the side effect of making the model somewhat sparse.

$$Loss_{\{L1\}} = Loss + \lambda \sum_{i=1}^n |w_i|$$

where

- ◆ Loss is the original loss function.
- ◆ λ is the regularization strength.
- ◆ w represents the model weights.

L2 regularization adds the sum of the squared weights to the loss instead, which shrinks weights toward zero without necessarily making them exactly zero.

$$Loss_{\{L2\}} = Loss + \lambda \sum_{i=1}^n w_i^2$$

where

- ◆ Loss is the original loss function.
- ◆ λ is the regularization strength.
- ◆ w represents the model weights.

Early stopping is a different kind of regularization: instead of changing the loss function, training is stopped once the validation loss stops improving for a set number of epochs (the patience). This avoids the extra training that would otherwise push the model into overfitting.

1.6.2 Implementation

1. Train a baseline model, an L1-regularized model, an L2-regularized model, and a model with early stopping, and compare test accuracy.

```
import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Input, Flatten, Dense

from tensorflow.keras.regularizers import l1, l2

from tensorflow.keras.callbacks import EarlyStopping

# Load MNIST dataset
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()

# Normalize pixel values
X_train = X_train / 255.0
X_test = X_test / 255.0

# Build the model
def build_model(reg=None):
    return Sequential([
        Input(shape=(28, 28)),
        Flatten(),
        Dense(256, activation='relu', kernel_regularizer=reg),
        Dense(128, activation='relu', kernel_regularizer=reg),
        Dense(10, activation='softmax')
    ])

# Baseline model
baseline = build_model()

baseline.compile(
```

```

optimizer='adam',
loss='sparse_categorical_crossentropy',
metrics=['accuracy']
)
baseline.fit(
    X_train, y_train,
    epochs=20,
    validation_split=0.1,
    verbose=0
)
# L1 regularization
l1_model = build_model(reg=l1(0.0001))
l1_model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
l1_model.fit(
    X_train, y_train,
    epochs=20,
    validation_split=0.1,
    verbose=0
)
# L2 regularization
l2_model = build_model(reg=l2(0.0001))
l2_model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',

```

```

    metrics=['accuracy']
)
l2_model.fit(
    X_train, y_train,
    epochs=20,
    validation_split=0.1,
    verbose=0
)
# Early stopping
es_model = build_model()
es_model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
early_stop = EarlyStopping(
    monitor='val_loss',
    patience=3,
    restore_best_weights=True
)
es_model.fit(
    X_train, y_train,
    epochs=20,
    validation_split=0.1,
    callbacks=[early_stop],
    verbose=0
)
# Evaluate all models

```

```
print("Model Performance\n")  
for name, model in [  
    ("Baseline", baseline),  
    ("L1", l1_model),  
    ("L2", l2_model),  
    ("EarlyStopping", es_model)  
]:  
    _, accuracy = model.evaluate(X_test, y_test, verbose=0)  
    print(name, "Test Accuracy:", round(accuracy, 4))
```

OUTPUT

```
Model Performance  
Baseline Test Accuracy: 0.9813  
L1 Test Accuracy: 0.9798  
L2 Test Accuracy: 0.9745  
EarlyStopping Test Accuracy: 0.9786
```

1.6.3 Lab Practice Questions

1. Change the value of lambda for L1 and L2 regularization and study how increasing the regularization strength affects the model's test accuracy.
2. Train the models with different values of patience in early stopping and determine how the choice of patience affects the training duration and model performance.

```
#include "KMotionDef.h"
```

```
int main()
```

```
{
```

```
    ch0->Amp = 250;
```

```
    ch0->output_mode=MICROSTEP_MODE;
```

```
    ch0->Vel=70.0f;
```

```
    ch0->Amp=250;
```

```
    ch0->Amp=250;
```

```
    ch0->Amp=250;
```

```
    EnableAxisDest(0,0);
```

```
    ch1->Amp = 250;
```

```
    ch1->output_mode=MICROSTEP_MODE;
```

```
    ch1->Vel=70.0f;
```

```
    ch1->Amp=250;
```

```
    ch1->Amp=250;
```

```
    ch1->Amp=250;
```

```
    ch1->Amp=250;
```

```
    EnableAxisDest(0,0);
```

```
    DefineCoordSystem(0,1,-1,-1);
```

```
    return 0;
```

```
}
```

CYCLE II

Convolutional and Recurrent Neural Networks

Experiment 1

Build a Convolutional Neural Network (CNN) and evaluate its performance on image classification

Objectives

- ◆ To understand the building blocks of a CNN: convolution, pooling, and fully connected layers.
- ◆ To build and train a CNN for an image classification task.
- ◆ To evaluate the trained CNN using accuracy and a confusion matrix.

2.1.1 Theory

A Convolutional Neural Network (CNN) is built around convolutional layers, which slide a small filter across the input image and produce a feature map that responds to patterns such as edges, textures, or shapes. Stacking several convolutional layers lets the network build up from simple features in the early layers to more complex, task-specific features in the deeper layers.

Pooling layers (usually max pooling) are placed between convolutional layers to reduce the spatial size of the feature maps, which cuts down the number of parameters and makes the network somewhat robust to small shifts in the input. After the convolutional and pooling layers, the feature maps are flattened and passed through one or more fully connected layers that produce the final classification.

Compared to a plain MLP applied directly to pixel values, a CNN uses far fewer parameters for image data because the same filter weights are reused across the whole image (weight sharing), and this generally gives better accuracy on image classification tasks.

2.1.2 Implementation

1. Build and train a CNN on the CIFAR-10 dataset and evaluate its test accuracy using a confusion matrix.

```

from tensorflow.keras.datasets import cifar10

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Flatten,
Dense

from tensorflow.keras.utils import to_categorical

from sklearn.metrics import confusion_matrix

# Load CIFAR-10 dataset
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

# Normalize pixel values
X_train = X_train / 255.0
X_test = X_test / 255.0

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

# Build CNN
model = Sequential([
    Input(shape=(32, 32, 3)),
    Conv2D(32, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])

# Compile model
model.compile(

```

```

optimizer='adam',
loss='categorical_crossentropy',
metrics=['accuracy']
)
# Train model
history = model.fit(
    X_train,
    y_train_cat,
    epochs=10,
    validation_split=0.1,
    batch_size=64,
    verbose=0
)
# Evaluate model
test_loss, test_acc = model.evaluate(
    X_test,
    y_test_cat,
    verbose=0
)
print("Test Accuracy:", round(test_acc, 4))
# Generate predictions
y_pred = model.predict(X_test, verbose=0)
y_pred_classes = y_pred.argmax(axis=1)
# Confusion matrix
cm = confusion_matrix(y_test.flatten(), y_pred_classes)
print("\nConfusion Matrix:")
print(cm)

```

OUTPUT

Test Accuracy: 0.7013

Confusion Matrix:

```
[[795 12 35 21 6 7 7 8 48 61]
 [49 729 2 11 4 7 9 4 36 149]
 [94 7 589 79 63 70 49 26 9 14]
 [38 12 69 595 45 142 42 20 15 22]
 [39 2 96 93 592 29 38 93 10 8]
 [22 1 49 251 30 568 15 41 6 17]
 [16 3 60 83 29 23 754 6 7 19]
 [27 4 28 58 42 56 5 751 5 24]
 [96 20 5 22 4 8 3 5 800 37]
 [37 42 8 18 3 11 5 8 28 840]]
```

2.1.3 Lab Practice Questions

1. Add a Dropout layer after the flatten step and check whether it improves test accuracy.
2. Compare the trained CNN against a plain MLP on the same dataset and note the difference in accuracy and parameter count.
3. Change the number of filters in the convolutional layers and compare the effect on test accuracy.

Experiment 2

Implement a Recurrent Neural Network (RNN) for sequence prediction

Objectives

- ◆ To understand the working of an RNN with sequential data.
- ◆ To implement an RNN for sequence prediction.
- ◆ To measure prediction error using MAE.

2.2.1 Theory

An RNN is used when the data comes in a sequence. The network takes the current value along with information from the previous step. This information is carried through the hidden state.

$$h_t = \tanh(W_x x_t + W_h h_{t-1} + b)$$

where:

- ◆ h_t — current hidden state
- ◆ x_t — current input at time step t
- ◆ h_{t-1} — hidden state from the previous time step
- ◆ W_x — weights for the current input
- ◆ W_h — weights for the previous hidden state
- ◆ b — bias
- ◆ $\tanh$ — activation function

For sequence prediction, earlier values are used to find the next value. In the AirPassengers dataset, the passenger counts of the previous 12 months are given to the RNN to predict the passenger count for the following month.

A simple RNN is suitable for short sequences. When the sequence becomes longer, information from earlier time steps may not be retained properly. Vanishing and exploding gradients are common reasons for this problem.

2.2.2 Implementation

1. Use the AirPassengers dataset to train an RNN for predicting the next month's passenger count from the previous 12 months.

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, SimpleRNN, Dense
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_absolute_error

# Load AirPassengers dataset

url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers.csv"

data = pd.read_csv(url)

# Select passenger counts
values = data["Passengers"].values.reshape(-1, 1)

# Split into training and testing data
split = int(len(values) * 0.8)
train_data = values[:split]
test_data = values[split:]

# Scale the data
scaler = MinMaxScaler()

train_scaled = scaler.fit_transform(train_data)
test_scaled = scaler.transform(test_data)

# Use previous 12 months to predict the next month
sequence_length = 12

X_train = []
y_train = []

for i in range(sequence_length, len(train_scaled)):
```

```

X_train.append(train_scaled[i-sequence_length:i])
y_train.append(train_scaled[i])
X_train = np.array(X_train)
y_train = np.array(y_train)
# Prepare test sequences
combined = np.concatenate(
    (train_scaled[-sequence_length:], test_scaled)
)
X_test = []
y_test = []
for i in range(sequence_length, len(combined)):
    X_test.append(combined[i-sequence_length:i])
    y_test.append(combined[i])
X_test = np.array(X_test)
y_test = np.array(y_test)
# Build RNN model
model = Sequential([
    Input(shape=(sequence_length, 1)),
    SimpleRNN(32, activation="tanh"),
    Dense(1)
])
# Compile model
model.compile(
    optimizer="adam",
    loss="mse"
)
# Train model
history = model.fit(

```

```

X_train,
y_train,
epochs=50,
batch_size=16,
validation_split=0.1,
verbose=1
)
# Predict test values
predicted = model.predict(X_test, verbose=0)
# Convert predictions to original scale
predicted = scaler.inverse_transform(predicted)
actual = scaler.inverse_transform(y_test)
# Calculate Mean Absolute Error
mae = mean_absolute_error(actual, predicted)
print("\nTest MAE:", round(mae, 2))
# Display predictions
print("\nActual vs Predicted")
for i in range(5):
    print(
        "Actual:", round(actual[i][0], 2),
        "Predicted:", round(predicted[i][0], 2)
    )

```

OUTPUT

```

Test MAE: 32.65
Actual vs Predicted
Actual: 505.0 Predicted: 484.91
Actual: 404.0 Predicted: 450.27
Actual: 359.0 Predicted: 379.26
Actual: 310.0 Predicted: 336.67
Actual: 337.0 Predicted: 345.18

```

2.2.3 Lab Practice Questions

1. Try using 6 months of previous passenger data instead of 12 months. What difference do you observe in the prediction?
2. Run the RNN with 32 units and then with 64 units. Which setting gives better results?

SGOU

Experiment 3

Compare the performance of LSTM and GRU networks on a time-series dataset

Objectives

- ◆ To understand the working of LSTM and GRU networks for time-series data.
- ◆ To train LSTM and GRU models on the same time-series dataset and compare their prediction performance.
- ◆ To compare the two models based on RMSE, training time, and number of parameters.

2.3.1 Theory

LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit) are both extensions of the plain RNN that use gates to control how information flows through the hidden state, which helps them retain useful information over longer sequences and largely avoids the vanishing gradient problem.

LSTM uses three gates - input, forget, and output - along with a separate cell state that carries long-term information across timesteps. The input gate controls how much new information is added to the cell state, the forget gate controls which existing information is discarded, and the output gate controls how much information from the cell state is passed to the hidden state.

GRU simplifies this structure to two gates - reset and update - and does not use a separate cell state. The update gate controls how much of the previous hidden state is retained and how much new information is incorporated, while the reset gate controls how much of the previous hidden state is considered when computing the new candidate state. This simpler structure gives GRU fewer parameters and generally makes it a little faster to train, often with comparable accuracy to LSTM.

This experiment trains both architectures on the same time-series forecasting problem so their behavior can be compared directly rather than going by general claims about which one is 'better'.

2.3.2 Implementation

1. Train LSTM and GRU models on a time-series dataset using the previous 12 observations to predict the next value. Compare the models using RMSE, training time, and number of parameters.

```

import numpy as np
import pandas as pd
import time

from sklearn.preprocessing import MinMaxScaler

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Input, LSTM, GRU, Dense

# Load the dataset

url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/daily-min-temperatures.csv"

df = pd.read_csv(url)

data = df["Temp"].values.reshape(-1, 1)

# Split the data

split = int(len(data) * 0.8)

train_data = data[:split]

test_data = data[split:]

# Scale the data using training data

scaler = MinMaxScaler()

train_scaled = scaler.fit_transform(train_data)

test_scaled = scaler.transform(test_data)

# Create sequences

window = 12

X_train = []

y_train = []

for i in range(window, len(train_scaled)):

    X_train.append(train_scaled[i-window:i])

    y_train.append(train_scaled[i])

X_test = []

y_test = []

```

```

combined = np.concatenate((train_scaled[-window:], test_scaled))
for i in range(window, len(combined)):
    X_test.append(combined[i-window:i])
    y_test.append(combined[i])
X_train = np.array(X_train)
y_train = np.array(y_train)
X_test = np.array(X_test)
y_test = np.array(y_test)
# Function to build and train a model
def build_and_train(layer):
    model = Sequential([
        Input(shape=(window, 1)),
        layer(32),
        Dense(1)
    ])
    model.compile(
        optimizer="adam",
        loss="mse"
    )
    start_time = time.time()
    model.fit(
        X_train,
        y_train,
        epochs=20,
        batch_size=32,
        verbose=0
    )
    training_time = time.time() - start_time

```

```

predictions = model.predict(X_test, verbose=0)
predictions = scaler.inverse_transform(predictions)
actual = scaler.inverse_transform(y_test)
rmse = np.sqrt(np.mean((predictions - actual) ** 2))
return rmse, training_time, model.count_params()

# Train LSTM and GRU
lstm_rmse, lstm_time, lstm_params = build_and_train(LSTM)
gru_rmse, gru_time, gru_params = build_and_train(GRU)

# Display results
print("LSTM")
print("RMSE:", round(lstm_rmse, 2))
print("Training Time:", round(lstm_time, 2), "seconds")
print("Parameters:", lstm_params)
print("\nGRU")
print("RMSE:", round(gru_rmse, 2))
print("Training Time:", round(gru_time, 2), "seconds")
print("Parameters:", gru_params)

```

OUTPUT

```

LSTM

RMSE: 2.38

Training Time: 21.42 seconds

Parameters: 4385

GRU

RMSE: 2.22

Training Time: 16.67 seconds

Parameters: 3393

```

2.3.3 Lab Practice Questions

1. Try a different number of previous time steps as input and check how the predictions change.
2. Run the models with a different number of units in the LSTM and GRU layers. Which model gives better results?
3. Add one more recurrent layer to both models and compare the results with the single-layer models.

SGOU

Experiment 4

Experiment with a simple Generative Adversarial Network (GAN) for image generation

Objectives

- ◆ To understand the role of the generator and discriminator in a GAN.
- ◆ To build and train a basic GAN for generating images using the MNIST dataset.
- ◆ To observe the generated images after training and study the changes in their quality.

2.4.1 Theory

A Generative Adversarial Network (GAN) consists of two networks: a generator and a discriminator. The generator takes random noise as input and creates images. The discriminator receives both real images from the training data and generated images and tries to distinguish between them.

During training, the discriminator learns to identify generated images, while the generator learns to produce images that can fool the discriminator. This process is repeated throughout training. As training progresses, the generated images can become more similar to the images in the training dataset. GAN training can also be difficult to control, and the results may depend on factors such as the learning rate and network architecture.

In this experiment, a basic GAN is trained using the MNIST dataset. The generated images are examined after training to see how well the generator is able to produce handwritten digits.

2.4.2 Implementation

1. Build and train a basic GAN on the MNIST dataset and display the generated digits after training.

```

import torch
from torch import nn
import math
import matplotlib.pyplot as plt
import torchvision
import torchvision.transforms as transforms

torch.manual_seed(111)

device = ""

if torch.cuda.is_available():
    device = torch.device("cuda")
else:
    device = torch.device("cpu")

transform = transforms.Compose(
    [transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))]
)

train_set = torchvision.datasets.MNIST(
    root=".", train=True, download=True, transform=transform
)

batch_size = 32

train_loader = torch.utils.data.DataLoader(
    train_set, batch_size=batch_size, shuffle=True
)

real_samples, mnist_labels = next(iter(train_loader))

for i in range(16):
    ax = plt.subplot(4, 4, i + 1)
    plt.imshow(real_samples[i].reshape(28, 28), cmap="gray_r")
    plt.xticks([])
    plt.yticks([])

```

```

class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(784, 1024),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(256, 1),
            nn.Sigmoid(),
        )
    def forward(self, x):
        x = x.view(x.size(0), 784)
        output = self.model(x)
        return output

discriminator = Discriminator().to(device=device)

```

```

class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(100, 256),
            nn.ReLU(),
            nn.Linear(256, 512),

```

```

        nn.ReLU(),
        nn.Linear(512, 1024),
        nn.ReLU(),
        nn.Linear(1024, 784),
        nn.Tanh(),
    )

    def forward(self, x):
        output = self.model(x)
        output = output.view(x.size(0), 1, 28, 28)
        return output

generator = Generator().to(device=device)
lr = 0.0001
num_epochs = 50
loss_function = nn.BCELoss()

optimizer_discriminator = torch.optim.Adam(discriminator.parameters(),
lr=lr)
optimizer_generator = torch.optim.Adam(generator.parameters(), lr=lr)
for epoch in range(num_epochs):
    for n, (real_samples, mnist_labels) in enumerate(train_loader):
        # Data for training the discriminator
        real_samples = real_samples.to(device=device)
        real_samples_labels = torch.ones((batch_size, 1)).to(
            device=device
        )
        latent_space_samples = torch.randn((batch_size, 100)).to(
            device=device
        )
        generated_samples = generator(latent_space_samples)

```

```

generated_samples_labels = torch.zeros((batch_size, 1)).to(
    device=device
)
all_samples = torch.cat((real_samples, generated_samples))
all_samples_labels = torch.cat(
    (real_samples_labels, generated_samples_labels)
)
# Training the discriminator
discriminator.zero_grad()
output_discriminator = discriminator(all_samples)
loss_discriminator = loss_function(
    output_discriminator, all_samples_labels
)
loss_discriminator.backward()
optimizer_discriminator.step()

# Data for training the generator
latent_space_samples = torch.randn((batch_size, 100)).to(
    device=device
)
# Training the generator
generator.zero_grad()
generated_samples = generator(latent_space_samples)
output_discriminator_generated = discriminator(generated_samples)
loss_generator = loss_function(
    output_discriminator_generated, real_samples_labels
)
loss_generator.backward()

```

```

optimizer_generator.step()

# Show loss

if n == batch_size - 1:

    print(f'Epoch: {epoch} Loss D.: {loss_discriminator}')

    print(f'Epoch: {epoch} Loss G.: {loss_generator}')

latent_space_samples = torch.randn(batch_size, 100).to(device=device)

generated_samples = generator(latent_space_samples)

generated_samples = generated_samples.cpu().detach()

for i in range(16):

    ax = plt.subplot(4, 4, i + 1)

    plt.imshow(generated_samples[i].reshape(28, 28), cmap="gray_r")

    plt.xticks([])

    plt.yticks([])

```

OUTPUT



2.4.3 Lab Practice Questions

1. Change the number of training epochs and compare the generated digits with the original output.
2. Change the batch size used during training and compare the generated images with the original output.
3. Change the learning rate of the generator and discriminator and observe its effect on the training process and generated images.

SGOU

സർവ്വകലാശാലാഗീതം

വിദ്യായാൽ സ്വതന്ത്രരാകണം
വിശ്വപൗരരായി മാറണം
ഗ്രഹപ്രസാദമായ് വിളങ്ങണം
ഗുരുപ്രകാശമേ നയിക്കണേ

കുതിരുട്ടിൽ നിന്നു ഞങ്ങളെ
സൂര്യവീഥിയിൽ തെളിക്കണം
സ്നേഹദീപ്തിയായ് വിളങ്ങണം
നീതിവൈജയന്തി പാറണം

ശാസ്ത്രവ്യാപ്തിയെന്നുമേകണം
ജാതിഭേദമാകെ മാറണം
ബോധരശ്മിയിൽ തിളങ്ങുവാൻ
ജ്ഞാനകേന്ദ്രമേ ജ്വലിക്കണേ

കുരിപ്പുഴ ശ്രീകുമാർ

SREENARAYANAGURU OPEN UNIVERSITY

Regional Centres

Kozhikode

Govt. Arts and Science College
Meenchantha, Kozhikode,
Kerala, Pin: 673002
Ph: 04952920228
email: rckdirector@sgou.ac.in

Thalassery

Govt. Brennen College
Dharmadam, Thalassery,
Kannur, Pin: 670106
Ph: 04902990494
email: rctdirector@sgou.ac.in

Tripunithura

Govt. College
Tripunithura, Ernakulam,
Kerala, Pin: 682301
Ph: 04842927436
email: rcedirector@sgou.ac.in

Pattambi

Sree Neelakanta Govt. Sanskrit College
Pattambi, Palakkad,
Kerala, Pin: 679303
Ph: 04662912009
email: rcpdirector@sgou.ac.in



BEFORE YOU EVEN THINK OF RAGGING



Humiliation

Blacklisting

Ruined Career

Download

**ANTI
RAGGING**

App

Don't just stand and watch. Stop Ragging! Show Character

Visit UGC Website i.e. www.ugc.Ac.in & www.antiragging.in to UGC Anti Ragging regulations.



Ministry of Education
Government of India



Are You Being Ragged ?

विश्वविद्यालय अनुदान आयोग
University Grants Commission
quality higher education for all

**DON'T LET IT
BE TOO LATE
SAY
NO
TO
DRUGS**

**LOVE YOURSELF
AND ALWAYS BE
HEALTHY**



SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

Advanced Machine Learning Lab with Python

COURSE CODE: B24DS05PC

SGOU



Sreenarayanaguru Open University

Kollam, Kerala Pin- 691601, email: info@sgou.ac.in, www.sgou.ac.in Ph: +91 474 2966841

ISBN 978-81-180900-0-7



9 788118 090007