DL LAB MANUVAL

|  |  |  |  |
| --- | --- | --- | --- |
| S.NO | EXP NO | NAME | PAGE NO |
| 1. | 1 | Implement multilayer perceptron algorithm for MNIST handwritten Digit Classification. | 1-3 |
| 2. | 2 | Design a neural network for classifying movie reviews (Binary Classification) using IMDB dataset. | 4-5 |
| 3. | 3 | Design a neural Network for classifying news wires (Multi class classification) using Reuters dataset. | 6-8 |
| 4 | 4 | Design a neural network for predicting house prices using Boston Housing Price dataset. | 9-11 |
| 5 | 5 | Build a Convolution Neural Network for MNISTH and written Digit Classification. | 12-13 |
| 6 | 6 | Build a Convolution Neural Network for simple image (dogs and Cats) Classification. | 14-15 |
| 7 | 7 | Use a pre-trained convolution neural network (VGG16) for image classification. | 16-19 |
| 8 | 8 | Implement one hot encoding of words or characters. | 20-23 |
| 9 | 9 | Implement word embeddings for IMDB dataset. | 24-29 |
| 10 | 10 | Implement a Recurrent Neural Network for IMDB Movie review classification problem | 30-38 |

**Experiment 1: Implement multilayer perceptron algorithm for MNIST handwritten Digit Classification.**

AIM: To implement a multi-layer perceptron using Python libraries (such as Keras or PyTorch) to classify handwritten digits from the MNIST dataset.

SOFTWARES: Python libraries, Google Colaboratory, keras, Tensorflow, PyTorch

SOURCE CODE:

import tensorflow as tf

import matplotlib.pyplot as plt

import seaborn as sn

import numpy as np

import pandas as pd

import math

import datetime

import platform

print('Python version:', platform.python\_version())

print('Tensorflow version:', tf.\_\_version\_\_)

print('Keras version:', tf.keras.\_\_version\_\_)

# Load the data

mnist\_dataset = tf.keras.datasets.mnist

(x\_train, y\_train), (x\_test, y\_test) = mnist\_dataset.load\_data()

print('x\_train:', x\_train.shape)

print('y\_train:', y\_train.shape)

print('x\_test:', x\_test.shape)

print('y\_test:', y\_test.shape)

# Visualize first image

plt.imshow(x\_train[0], cmap=plt.cm.binary)

plt.show()

# Display the first 25 examples

numbers\_to\_display = 25

num\_cells = math.ceil(math.sqrt(numbers\_to\_display))

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

for i in range(numbers\_to\_display):

    plt.subplot(num\_cells, num\_cells, i + 1)

    plt.xticks([])

    plt.yticks([])

    plt.grid(False)

    plt.imshow(x\_train[i], cmap=plt.cm.binary)

    plt.xlabel(y\_train[i])

plt.show()

# Normalize the data

x\_train\_normalized = x\_train / 255.0

x\_test\_normalized = x\_test / 255.0

# Show a normalized image

plt.imshow(x\_train\_normalized[0], cmap=plt.cm.binary)

plt.show()

# Build the model

model = tf.keras.models.Sequential([

    tf.keras.layers.Flatten(input\_shape=(28, 28)),

    tf.keras.layers.Dense(128, activation='relu', kernel\_regularizer=tf.keras.regularizers.l2(0.002)),

    tf.keras.layers.Dense(128, activation='relu', kernel\_regularizer=tf.keras.regularizers.l2(0.002)),

    tf.keras.layers.Dense(10, activation='softmax'),

])

model.summary()

adam\_optimizer = tf.keras.optimizers.Adam(learning\_rate=0.001)

model.compile(

    optimizer=adam\_optimizer,

    loss=tf.keras.losses.sparse\_categorical\_crossentropy,

    metrics=['accuracy']

)

# Logging (TensorBoard)

log\_dir = "./logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

tensorboard\_callback = tf.keras.callbacks.TensorBoard(log\_dir=log\_dir, histogram\_freq=1)

training\_history = model.fit(

    x\_train\_normalized,

    y\_train,

    epochs=10,

    validation\_data=(x\_test\_normalized, y\_test),

    callbacks=[tensorboard\_callback]

)

# Plot loss and accuracy

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)

plt.xlabel('Epoch Number')

plt.ylabel('Loss')e

plt.plot(training\_history.history['loss'], label='Training loss')

plt.plot(training\_history.history['val\_loss'], label='Test loss')

plt.legend()

plt.subplot(1, 2, 2)

plt.xlabel('Epoch Number')

plt.ylabel('Accuracy')

plt.plot(training\_history.history['accuracy'], label='Training accuracy')

plt.plot(training\_history.history['val\_accuracy'], label='Test accuracy')

plt.legend()

plt.tight\_layout()

plt.show()

# Evaluate model accuracy

train\_loss, train\_accuracy = model.evaluate(x\_train\_normalized, y\_train)

print('Training loss:', train\_loss)

print('Training accuracy:', train\_accuracy)

validation\_loss, validation\_accuracy = model.evaluate(x\_test\_normalized, y\_test)

print('Validation loss:', validation\_loss)

print('Validation accuracy:', validation\_accuracy)

# Save and reload the model

model\_name = 'digits\_recognition\_mlp.h5'

model.save(model\_name, save\_format='h5')

loaded\_model = tf.keras.models.load\_model(model\_name)

# Predictions

predictions\_one\_hot = loaded\_model.predict(x\_test\_normalized)

predictions = np.argmax(predictions\_one\_hot, axis=1)

print('First prediction:', predictions[0])

plt.imshow(x\_test\_normalized[0], cmap=plt.cm.binary)

plt.title(f'Predicted: {predictions[0]}')

plt.show()

# Show multiple predictions

numbers\_to\_display = 36

num\_cells = math.ceil(math.sqrt(numbers\_to\_display))

plt.figure(figsize=(12, 12))

for plot\_index in range(numbers\_to\_display):

    predicted\_label = predictions[plot\_index]

    actual\_label = y\_test[plot\_index]

    color\_map = 'Greens' if predicted\_label == actual\_label else 'Reds'

    plt.subplot(num\_cells, num\_cells, plot\_index + 1)

    plt.xticks([])

    plt.yticks([])

    plt.grid(False)

    plt.imshow(x\_test\_normalized[plot\_index], cmap=color\_map)

    plt.xlabel(str(predicted\_label))

plt.tight\_layout()

plt.show()

# Plot the confusion matrix

confusion\_matrix = tf.math.confusion\_matrix(y\_test, predictions)

f, ax = plt.subplots(figsize=(9, 7))

sn.heatmap(

    confusion\_matrix,

    annot=True,

    linewidths=.5,

    fmt="d",

    square=True,

    ax=ax

)

plt.show()

OUTPUT:

Python version: 3.11.13

Tensorflow version: 2.18.0

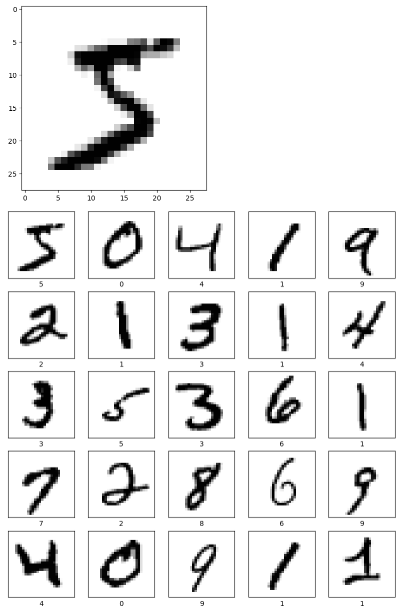
Keras version: 3.8.0

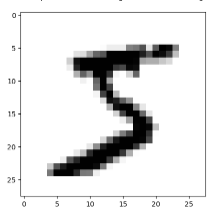
x\_train: (60000, 28, 28)

y\_train: (60000,)

x\_test: (10000, 28, 28)

y\_test: (10000,)





/usr/local/lib/python3.11/dist-packages/keras/src/layers/reshaping/flatten.py:37: UserWarning: Do not pass an `input\_shape`/`input\_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

super().\_\_init\_\_(\*\*kwargs)

**Model: "sequential\_5"**

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓

┃ **Layer (type)** ┃ **Output Shape** ┃ **Param #** ┃

┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩

│ flatten (Flatten) │ (None, 784) │ 0 │

├─────────────────────────────────┼────────────────────────┼───────────────┤

│ dense\_10 (Dense) │ (None, 128) │ 100,480 │

├─────────────────────────────────┼────────────────────────┼───────────────┤

│ dense\_11 (Dense) │ (None, 128) │ 16,512 │

├─────────────────────────────────┼────────────────────────┼───────────────┤

│ dense\_12 (Dense) │ (None, 10) │ 1,290 │

└─────────────────────────────────┴────────────────────────┴───────────────┘

**Total params:** 118,282 (462.04 KB)

**Trainable params:** 118,282 (462.04 KB)

**Non-trainable params:** 0 (0.00 B)

Epoch 1/10

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **11s** 5ms/step - accuracy: 0.8726 - loss: 0.7747 - val\_accuracy: 0.9570 - val\_loss: 0.3292

Epoch 2/10

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **17s** 4ms/step - accuracy: 0.9561 - loss: 0.3133 - val\_accuracy: 0.9529 - val\_loss: 0.2952

Epoch 3/10

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **9s** 3ms/step - accuracy: 0.9604 - loss: 0.2670 - val\_accuracy: 0.9578 - val\_loss: 0.2578

Epoch 4/10

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **7s** 4ms/step - accuracy: 0.9649 - loss: 0.2404 - val\_accuracy: 0.9645 - val\_loss: 0.2217

Epoch 5/10

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **7s** 4ms/step - accuracy: 0.9660 - loss: 0.2214 - val\_accuracy: 0.9670 - val\_loss: 0.2140

Epoch 6/10

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **8s** 4ms/step - accuracy: 0.9681 - loss: 0.2098 - val\_accuracy: 0.9597 - val\_loss: 0.2284

Epoch 7/10

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **7s** 4ms/step - accuracy: 0.9687 - loss: 0.2026 - val\_accuracy: 0.9733 - val\_loss: 0.1929

Epoch 8/10

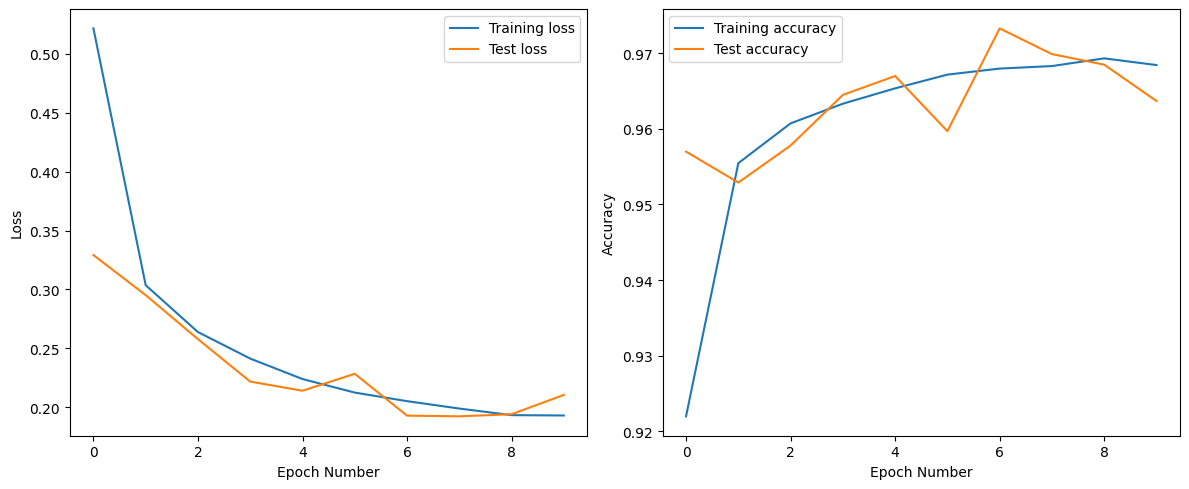
**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **6s** 3ms/step - accuracy: 0.9689 - loss: 0.1969 - val\_accuracy: 0.9699 - val\_loss: 0.1923

Epoch 9/10

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **7s** 3ms/step - accuracy: 0.9686 - loss: 0.1959 - val\_accuracy: 0.9685 - val\_loss: 0.1940

Epoch 10/10

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **6s** 3ms/step - accuracy: 0.9705 - loss: 0.1879 - val\_accuracy: 0.9637 - val\_loss: 0.2104



**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **4s** 2ms/step - accuracy: 0.9683 - loss: 0.1947

Training loss: 0.1955161690711975

Training accuracy: 0.9675666689872742

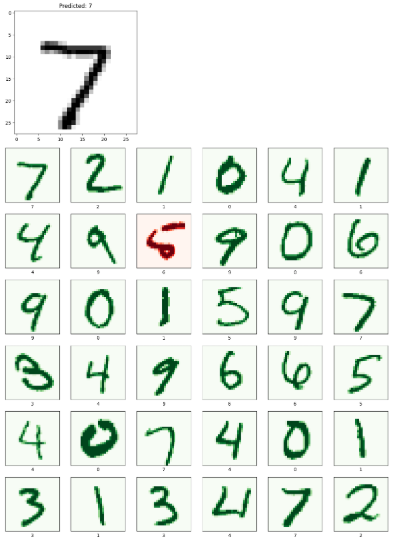
**313/313** ━━━━━━━━━━━━━━━━━━━━ **1s** 2ms/step - accuracy: 0.9581 - loss: 0.2296

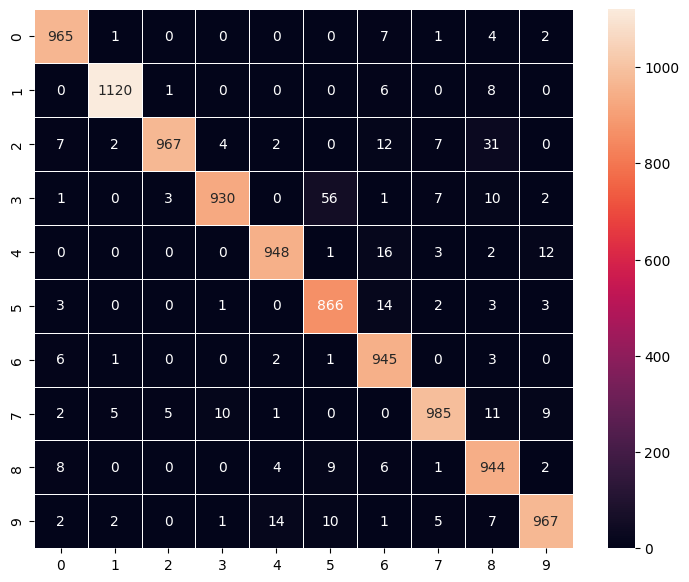
Validation loss: 0.210449680685997

Validation accuracy: 0.963699996471405

**313/313** ━━━━━━━━━━━━━━━━━━━━ **1s** 2ms/step

First prediction: 7





CONCLUSION:

By developing and training an MLP model in Python, we achieved accurate digit classification on the MNIST dataset, demonstrating the practicality and simplicity of neural networks for basic image recognition using Python tools.

References:

<https://www.youtube.com/watch?v=aVS1eoSj3Es>

<https://www.geeksforgeeks.org/machine-learning/handwritten-digit-recognition-using-neural-network/>

**Experiment 2: Design a neural network for classifying movie reviews (Binary Classification) using IMDB dataset.**

AIM: To design and implement a Python-based neural network (using Keras or PyTorch) to perform binary classification on movie reviews from the IMDB dataset.

SOFTWARES: Python libraries, Google Colaboratory, keras, Tensorflow, PyTorch

SOURCE CODE:

import tensorflow as tf

from tensorflow import keras

import numpy as np

import matplotlib.pyplot as plt

# Load the IMDB dataset with top 10,000 most frequent words

imdb = keras.datasets.imdb

(train\_data, train\_labels), (test\_data, test\_labels) = imdb.load\_data(num\_words=10000)

# Get the word index and adjust with reserved indices

word\_index = imdb.get\_word\_index()

word\_index = {k: (v + 3) for k, v in word\_index.items()}

word\_index["<PAD>"] = 0

word\_index["<START>"] = 1

word\_index["<UNK>"] = 2

word\_index["<UNUSED>"] = 3

# Pad the sequences so they are all the same length

train\_data = keras.preprocessing.sequence.pad\_sequences(

    train\_data,

    value=word\_index["<PAD>"],

    padding='post',

    maxlen=256

)

test\_data = keras.preprocessing.sequence.pad\_sequences(

    test\_data,

    value=word\_index["<PAD>"],

    padding='post',

    maxlen=256

)

# Build the model

vocab\_size = 10000

model = keras.Sequential([

    keras.layers.Embedding(vocab\_size, 16),

    keras.layers.GlobalAveragePooling1D(),

    keras.layers.Dense(16, activation='relu'),

    keras.layers.Dense(1, activation='sigmoid')

])

model.summary()

# Compile the model

model.compile(optimizer='adam',

              loss='binary\_crossentropy',

              metrics=['accuracy'])

# Create validation set

x\_val = train\_data[:10000]

partial\_x\_train = train\_data[10000:]

y\_val = train\_labels[:10000]

partial\_y\_train = train\_labels[10000:]

# Train the model

history = model.fit(

    partial\_x\_train,

    partial\_y\_train,

    epochs=20,

    batch\_size=512,

    validation\_data=(x\_val, y\_val),

    verbose=1

)

# Evaluate the model

results = model.evaluate(test\_data, test\_labels)

print(f"Test Loss: {results[0]:.4f}")

print(f"Test Accuracy: {results[1]\*100:.2f}%")

# Plot training and validation loss & accuracy

history\_dict = history.history

acc = history\_dict['accuracy']

val\_acc = history\_dict['val\_accuracy']

loss = history\_dict['loss']

val\_loss = history\_dict['val\_loss']

epochs = range(1, len(acc) + 1)

plt.figure(figsize=(12,5))

# Loss plot

plt.subplot(1, 2, 1)

plt.plot(epochs, loss, 'bo', label='Training loss')

plt.plot(epochs, val\_loss, 'b', label='Validation loss')

plt.title('Training and validation loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

# Accuracy plot

plt.subplot(1, 2, 2)

plt.plot(epochs, acc, 'bo', label='Training acc')

plt.plot(epochs, val\_acc, 'b', label='Validation acc')

plt.title('Training and validation accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.tight\_layout()

plt.show()

OUTPUT:

**Model: "sequential\_1"**

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓

┃ **Layer (type)** ┃ **Output Shape** ┃ **Param #** ┃

┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩

│ embedding\_2 (Embedding) │ ? │ 0 (unbuilt) │

├─────────────────────────────────┼────────────────────────┼───────────────┤

│ global\_average\_pooling1d\_2 │ ? │ 0 │

│ (GlobalAveragePooling1D) │ │ │

├─────────────────────────────────┼────────────────────────┼───────────────┤

│ dense\_4 (Dense) │ ? │ 0 (unbuilt) │

├─────────────────────────────────┼────────────────────────┼───────────────┤

│ dense\_5 (Dense) │ ? │ 0 (unbuilt) │

└─────────────────────────────────┴────────────────────────┴───────────────┘

**Total params:** 0 (0.00 B)

**Trainable params:** 0 (0.00 B)

**Non-trainable params:** 0 (0.00 B)

Epoch 1/20

**30/30** ━━━━━━━━━━━━━━━━━━━━ **4s** 78ms/step - accuracy: 0.5296 - loss: 0.6925 - val\_accuracy: 0.5755 - val\_loss: 0.6880

Epoch 2/20

**30/30** ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step - accuracy: 0.6246 - loss: 0.6842 - val\_accuracy: 0.6145 - val\_loss: 0.6741

Epoch 3/20

**30/30** ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step - accuracy: 0.6741 - loss: 0.6645 - val\_accuracy: 0.7314 - val\_loss: 0.6452

Epoch 4/20

**30/30** ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step - accuracy: 0.7452 - loss: 0.6322 - val\_accuracy: 0.7537 - val\_loss: 0.6068

Epoch 5/20

**30/30** ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step - accuracy: 0.7544 - loss: 0.5888 - val\_accuracy: 0.7723 - val\_loss: 0.5629

Epoch 6/20

**30/30** ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step - accuracy: 0.8046 - loss: 0.5367 - val\_accuracy: 0.8032 - val\_loss: 0.5111

Epoch 7/20

**30/30** ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step - accuracy: 0.8212 - loss: 0.4880 - val\_accuracy: 0.8261 - val\_loss: 0.4662

Epoch 8/20

**30/30** ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step - accuracy: 0.8505 - loss: 0.4355 - val\_accuracy: 0.8326 - val\_loss: 0.4306

Epoch 9/20

**30/30** ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step - accuracy: 0.8611 - loss: 0.4000 - val\_accuracy: 0.8444 - val\_loss: 0.4010

Epoch 10/20

**30/30** ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step - accuracy: 0.8652 - loss: 0.3714 - val\_accuracy: 0.8516 - val\_loss: 0.3821

Epoch 11/20

**30/30** ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step - accuracy: 0.8730 - loss: 0.3484 - val\_accuracy: 0.8583 - val\_loss: 0.3612

Epoch 12/20

**30/30** ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step - accuracy: 0.8820 - loss: 0.3253 - val\_accuracy: 0.8620 - val\_loss: 0.3471

Epoch 13/20

**30/30** ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step - accuracy: 0.8900 - loss: 0.3026 - val\_accuracy: 0.8628 - val\_loss: 0.3404

Epoch 14/20

**30/30** ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step - accuracy: 0.8899 - loss: 0.2970 - val\_accuracy: 0.8686 - val\_loss: 0.3298

Epoch 15/20

**30/30** ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step - accuracy: 0.8961 - loss: 0.2796 - val\_accuracy: 0.8731 - val\_loss: 0.3185

Epoch 16/20

**30/30** ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step - accuracy: 0.9037 - loss: 0.2623 - val\_accuracy: 0.8744 - val\_loss: 0.3120

Epoch 17/20

**30/30** ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step - accuracy: 0.9029 - loss: 0.2585 - val\_accuracy: 0.8753 - val\_loss: 0.3066

Epoch 18/20

**30/30** ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step - accuracy: 0.9099 - loss: 0.2452 - val\_accuracy: 0.8799 - val\_loss: 0.3016

Epoch 19/20

**30/30** ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step - accuracy: 0.9146 - loss: 0.2363 - val\_accuracy: 0.8704 - val\_loss: 0.3049

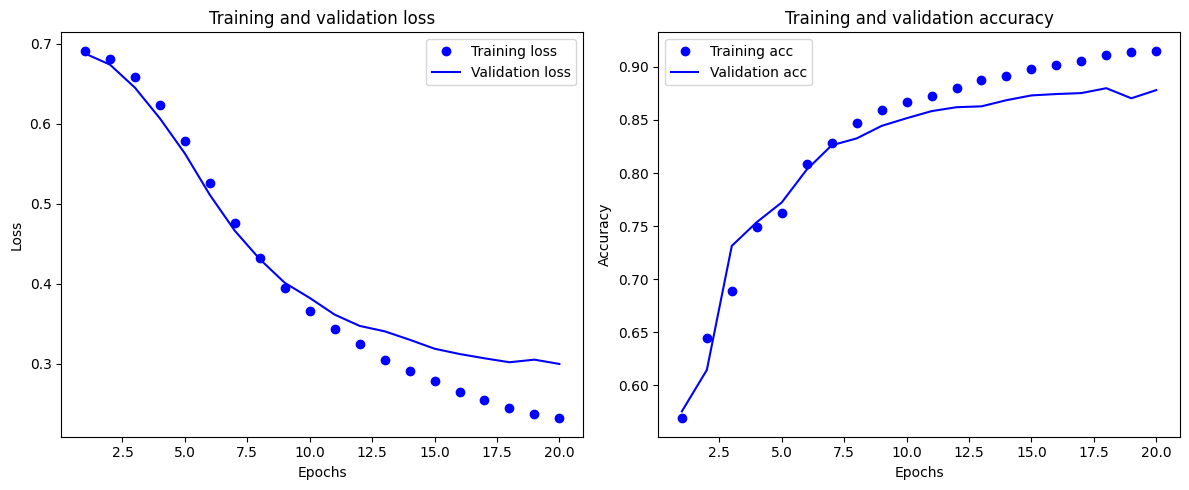
Epoch 20/20

**30/30** ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step - accuracy: 0.9154 - loss: 0.2311 - val\_accuracy: 0.8781 - val\_loss: 0.2995

**782/782** ━━━━━━━━━━━━━━━━━━━━ **2s** 2ms/step - accuracy: 0.8751 - loss: 0.3054

Test Loss: 0.3089

Test Accuracy: 87.27%



CONCLUSION:

The neural network coded in Python effectively classified IMDB reviews as positive or negative, validating the use of Python for natural language processing and deep learning tasks.

**Experiment 3: Design a neural Network for classifying news wires (Multi class classification) using Reuters dataset.**

AIM: To implement a Python neural network to categorize news articles from the Reuters dataset into multiple classes.

SOFTWARES: Python libraries, Google Colaboratory, keras, Tensorflow, PyTorch

SOURCE CODE:

# Import necessary libraries

import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.datasets import reuters

from tensorflow.keras.utils import to\_categorical

from tensorflow.keras import models, layers

# Load Reuters dataset, keeping top 10,000 most frequent words

(train\_data, train\_labels), (test\_data, test\_labels) = reuters.load\_data(num\_words=10000)

# Function to vectorize sequences into a binary matrix

def vectorize\_sequences(sequences, dimension=10000):

    results = np.zeros((len(sequences), dimension))

    for i, sequence in enumerate(sequences):

        results[i, sequence] = 1.

    return results

# Vectorize training and test data

x\_train = vectorize\_sequences(train\_data)

x\_test = vectorize\_sequences(test\_data)

# One-hot encode the labels

one\_hot\_train\_labels = to\_categorical(train\_labels)

one\_hot\_test\_labels = to\_categorical(test\_labels)

# Prepare a validation set (first 1000 samples from training data)

x\_val = x\_train[:1000]

partial\_x\_train = x\_train[1000:]

y\_val = one\_hot\_train\_labels[:1000]

partial\_y\_train = one\_hot\_train\_labels[1000:]

# Build the model

model = models.Sequential()

model.add(layers.Dense(64, activation='relu', input\_shape=(10000,)))

model.add(layers.Dense(64, activation='relu'))

model.add(layers.Dense(46, activation='softmax'))

# Compile the model

model.compile(optimizer='rmsprop',

              loss='categorical\_crossentropy',

              metrics=['accuracy'])

# Train the model with validation data

history = model.fit(partial\_x\_train,

                    partial\_y\_train,

                    epochs=20,

                    batch\_size=512,

                    validation\_data=(x\_val, y\_val))

# Plot training and validation loss

loss = history.history['loss']

val\_loss = history.history['val\_loss']

epochs = range(1, len(loss) + 1)

plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)

plt.plot(epochs, loss, 'bo-', label='Training loss')

plt.plot(epochs, val\_loss, 'ro-', label='Validation loss')

plt.title('Training and Validation Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

# Plot training and validation accuracy

acc = history.history['accuracy']

val\_acc = history.history['val\_accuracy']

plt.subplot(1, 2, 2)

plt.plot(epochs, acc, 'bo-', label='Training Accuracy')

plt.plot(epochs, val\_acc, 'ro-', label='Validation Accuracy')

plt.title('Training and Validation Accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.tight\_layout()

plt.show()

# Retrain model from scratch to avoid overfitting (early stopping at 8 epochs based on observations)

model = models.Sequential()

model.add(layers.Dense(64, activation='relu', input\_shape=(10000,)))

model.add(layers.Dense(64, activation='relu'))

model.add(layers.Dense(46, activation='softmax'))

model.compile(optimizer='rmsprop',

              loss='categorical\_crossentropy',

              metrics=['accuracy'])

model.fit(partial\_x\_train,

          partial\_y\_train,

          epochs=8,

          batch\_size=512,

          validation\_data=(x\_val, y\_val))

# Evaluate the model on the test dataset

results = model.evaluate(x\_test, one\_hot\_test\_labels)

print(f"Test loss: {results[0]:.4f}")

print(f"Test accuracy: {results[1]\*100:.2f}%")

# Generate predictions for the first test sample as an example

predictions = model.predict(x\_test[:1])

print(f"Predicted class probabilities for first test sample:\n{predictions}")

print(f"Predicted class index: {np.argmax(predictions[0])}")

print(f"True class index: {np.argmax(one\_hot\_test\_labels[0])}")

OUTPUT:

Epoch 1/20

**16/16** ━━━━━━━━━━━━━━━━━━━━ **3s** 84ms/step - accuracy: 0.3262 - loss: 3.3611 - val\_accuracy: 0.6160 - val\_loss: 2.0395

Epoch 2/20

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 19ms/step - accuracy: 0.6531 - loss: 1.8091 - val\_accuracy: 0.6810 - val\_loss: 1.4486

Epoch 3/20

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 15ms/step - accuracy: 0.7275 - loss: 1.2758 - val\_accuracy: 0.7340 - val\_loss: 1.2207

Epoch 4/20

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 15ms/step - accuracy: 0.7816 - loss: 1.0038 - val\_accuracy: 0.7670 - val\_loss: 1.0970

Epoch 5/20

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 17ms/step - accuracy: 0.8174 - loss: 0.8602 - val\_accuracy: 0.7820 - val\_loss: 1.0247

Epoch 6/20

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 15ms/step - accuracy: 0.8585 - loss: 0.6905 - val\_accuracy: 0.8010 - val\_loss: 0.9694

Epoch 7/20

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 17ms/step - accuracy: 0.8736 - loss: 0.5943 - val\_accuracy: 0.7990 - val\_loss: 0.9139

Epoch 8/20

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 17ms/step - accuracy: 0.8939 - loss: 0.5049 - val\_accuracy: 0.8150 - val\_loss: 0.8815

Epoch 9/20

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 16ms/step - accuracy: 0.9168 - loss: 0.4197 - val\_accuracy: 0.8160 - val\_loss: 0.8588

Epoch 10/20

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 18ms/step - accuracy: 0.9245 - loss: 0.3616 - val\_accuracy: 0.8060 - val\_loss: 0.8816

Epoch 11/20

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 18ms/step - accuracy: 0.9387 - loss: 0.3021 - val\_accuracy: 0.8180 - val\_loss: 0.8746

Epoch 12/20

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 16ms/step - accuracy: 0.9425 - loss: 0.2794 - val\_accuracy: 0.8110 - val\_loss: 0.8638

Epoch 13/20

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 16ms/step - accuracy: 0.9443 - loss: 0.2438 - val\_accuracy: 0.8210 - val\_loss: 0.8564

Epoch 14/20

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 15ms/step - accuracy: 0.9480 - loss: 0.2132 - val\_accuracy: 0.8180 - val\_loss: 0.8608

Epoch 15/20

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 16ms/step - accuracy: 0.9494 - loss: 0.2003 - val\_accuracy: 0.8180 - val\_loss: 0.8802

Epoch 16/20

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 16ms/step - accuracy: 0.9558 - loss: 0.1702 - val\_accuracy: 0.8070 - val\_loss: 0.8973

Epoch 17/20

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 15ms/step - accuracy: 0.9557 - loss: 0.1644 - val\_accuracy: 0.8070 - val\_loss: 0.9328

Epoch 18/20

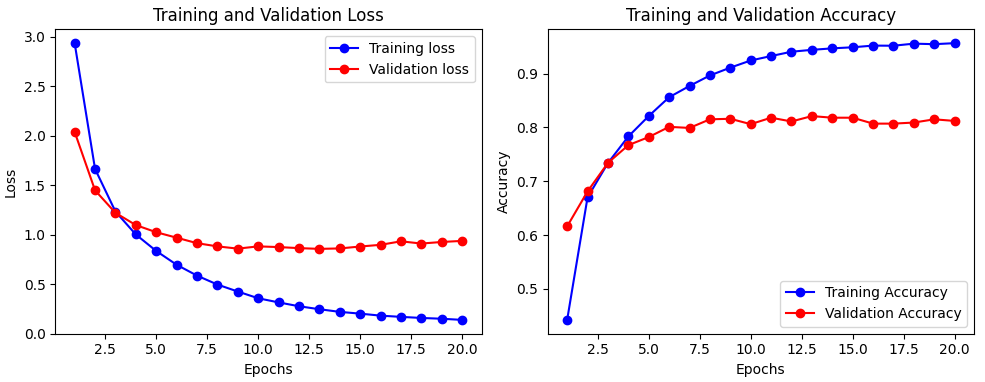
**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 15ms/step - accuracy: 0.9590 - loss: 0.1528 - val\_accuracy: 0.8090 - val\_loss: 0.9101

Epoch 19/20

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 17ms/step - accuracy: 0.9560 - loss: 0.1455 - val\_accuracy: 0.8150 - val\_loss: 0.9261

Epoch 20/20

**16/16** ━━━━━━━━━━━━━━━━━━━━ **1s** 20ms/step - accuracy: 0.9595 - loss: 0.1322 - val\_accuracy: 0.8120 - val\_loss: 0.9371



Epoch 1/8

**16/16** ━━━━━━━━━━━━━━━━━━━━ **3s** 101ms/step - accuracy: 0.4110 - loss: 3.1963 - val\_accuracy: 0.6240 - val\_loss: 1.8402

Epoch 2/8

**16/16** ━━━━━━━━━━━━━━━━━━━━ **1s** 18ms/step - accuracy: 0.6609 - loss: 1.6767 - val\_accuracy: 0.6920 - val\_loss: 1.4050

Epoch 3/8

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 16ms/step - accuracy: 0.7261 - loss: 1.2715 - val\_accuracy: 0.7300 - val\_loss: 1.2145

Epoch 4/8

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 18ms/step - accuracy: 0.7799 - loss: 1.0121 - val\_accuracy: 0.7690 - val\_loss: 1.1010

Epoch 5/8

**16/16** ━━━━━━━━━━━━━━━━━━━━ **1s** 17ms/step - accuracy: 0.8271 - loss: 0.8351 - val\_accuracy: 0.7800 - val\_loss: 1.0122

Epoch 6/8

**16/16** ━━━━━━━━━━━━━━━━━━━━ **1s** 16ms/step - accuracy: 0.8572 - loss: 0.6941 - val\_accuracy: 0.7930 - val\_loss: 0.9605

Epoch 7/8

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 16ms/step - accuracy: 0.8879 - loss: 0.5679 - val\_accuracy: 0.8120 - val\_loss: 0.9158

Epoch 8/8

**16/16** ━━━━━━━━━━━━━━━━━━━━ **0s** 16ms/step - accuracy: 0.9043 - loss: 0.4739 - val\_accuracy: 0.8060 - val\_loss: 0.8993

**71/71** ━━━━━━━━━━━━━━━━━━━━ **1s** 5ms/step - accuracy: 0.7936 - loss: 0.9375

Test loss: 0.9703

Test accuracy: 78.32%

**1/1** ━━━━━━━━━━━━━━━━━━━━ **0s** 193ms/step

Predicted class probabilities for first test sample:

[[1.7017950e-05 4.6317477e-04 2.8631648e-06 7.0836353e-01 2.8463241e-01

4.0554924e-06 7.8071571e-05 1.4646428e-05 1.3968026e-03 6.1044135e-05

2.3721966e-04 8.4457526e-05 4.5042816e-05 5.1127764e-04 1.0023881e-04

4.8161382e-05 3.8949942e-04 3.6833080e-04 2.8891525e-05 2.3853134e-04

5.7321577e-04 1.0781493e-04 9.2258006e-06 3.1404776e-04 1.7740816e-05

1.3786479e-04 5.6942845e-06 4.8369253e-05 5.8233280e-05 4.2616648e-05

3.5939613e-04 2.5806666e-04 6.5623586e-05 1.7180268e-05 1.2839741e-04

1.5159653e-05 2.0722875e-04 5.3833483e-06 1.1891461e-05 4.0258808e-04

2.8741093e-05 5.4658794e-05 8.9831337e-06 2.0722766e-05 3.2039150e-06

1.2755669e-05]]

Predicted class index: 3

True class index: 3

CONCLUSION:

The Python-based neural network successfully managed multi-class classification of newswires, showing the robustness of Python frameworks for text data and multi-class problems.

**Experiment 4: Design a neural network for predicting house prices using Boston Housing Price dataset.**

AIM: To create a Python neural network for predicting house prices using the Boston Housing dataset.

SOFTWARES: Python libraries, Google Colaboratory, keras, Tensorflow, PyTorch

SOURCE CODE:

import numpy as np

import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow.keras.datasets import boston\_housing

from tensorflow.keras import models, layers, backend as K

# Load the Boston Housing dataset

(train\_data, train\_targets), (test\_data, test\_targets) = boston\_housing.load\_data()

print("Train data shape:", train\_data.shape)

print("Test data shape:", test\_data.shape)

# Normalize features based on the training data statistics

mean = train\_data.mean(axis=0)

std = train\_data.std(axis=0)

train\_data = (train\_data - mean) / std

test\_data = (test\_data - mean) / std  # Use training mean and std for test data normalization!

# Define a function to build the model

def build\_model():

    model = models.Sequential()

    model.add(layers.Dense(64, activation='relu', input\_shape=(train\_data.shape[1],)))

    model.add(layers.Dense(64, activation='relu'))

    model.add(layers.Dense(1))  # Single linear output for regression

    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

    return model

# K-fold Cross Validation parameters

k = 4

num\_val\_samples = len(train\_data) // k

num\_epochs = 500

all\_mae\_histories = []

# Clear any previous models from memory

K.clear\_session()

# K-fold Cross Validation loop

for i in range(k):

    print(f'Processing fold #{i}')

    val\_data = train\_data[i \* num\_val\_samples: (i + 1) \* num\_val\_samples]

    val\_targets = train\_targets[i \* num\_val\_samples: (i + 1) \* num\_val\_samples]

    partial\_train\_data = np.concatenate(

        [train\_data[:i \* num\_val\_samples],

         train\_data[(i + 1) \* num\_val\_samples:]],

        axis=0

    )

    partial\_train\_targets = np.concatenate(

        [train\_targets[:i \* num\_val\_samples],

         train\_targets[(i + 1) \* num\_val\_samples:]],

        axis=0

    )

    model = build\_model()

    history = model.fit(

        partial\_train\_data, partial\_train\_targets,

        validation\_data=(val\_data, val\_targets),

        epochs=num\_epochs,

        batch\_size=1,

        verbose=0

    )

    # Fix here: use 'val\_mae' key instead of 'val\_mean\_absolute\_error'

    mae\_history = history.history['val\_mae']

    all\_mae\_histories.append(mae\_history)

# Compute the average of the per-epoch validation MAE scores

average\_mae\_history = [

    np.mean([x[i] for x in all\_mae\_histories]) for i in range(num\_epochs)

]

# Function to smooth the curve (exponential moving average)

def smooth\_curve(points, factor=0.9):

    smoothed\_points = []

    for point in points:

        if smoothed\_points:

            previous = smoothed\_points[-1]

            smoothed\_points.append(previous \* factor + point \* (1 - factor))

        else:

            smoothed\_points.append(point)

    return smoothed\_points

# Smooth the average MAE history, omit first 10 epochs for scaling reasons

smooth\_mae\_history = smooth\_curve(average\_mae\_history[10:])

# Plot smoothed validation MAE

plt.plot(range(1, len(smooth\_mae\_history) + 1), smooth\_mae\_history)

plt.xlabel('Epochs')

plt.ylabel('Validation MAE')

plt.title('Smoothed Validation MAE over Epochs (after epoch 10)')

plt.show()

# Based on plot inspection, train final model for 80 epochs (to avoid overfitting)

model = build\_model()

model.fit(train\_data, train\_targets, epochs=80, batch\_size=16, verbose=1)

# Evaluate final model on test data

test\_mse\_score, test\_mae\_score = model.evaluate(test\_data, test\_targets)

print(f'\nTest MSE: {test\_mse\_score:.4f}')

print(f'Test MAE: {test\_mae\_score:.4f}')

# Predict and display predicted vs actual house prices on test set

y\_pred = model.predict(test\_data).flatten()

print('\nPredicted\tActual')

for pred, actual in zip(y\_pred[:10], test\_targets[:10]):

    print(f'{pred:.2f}\t\t{actual:.2f}')

OUTPUT:

Train data shape: (404, 13)

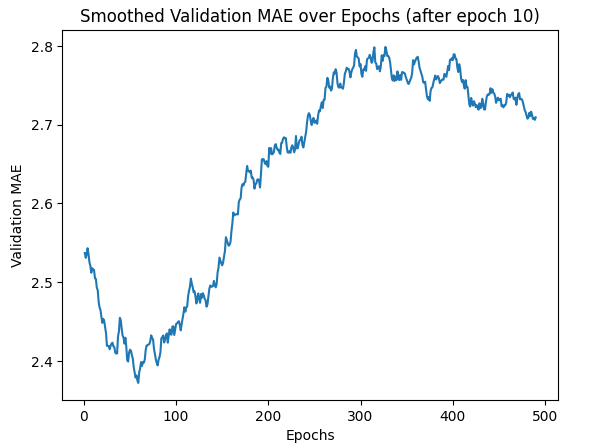
Test data shape: (102, 13)

Processing fold #0

Processing fold #1

Processing fold #2

Processing fold #3



Epoch 1/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **1s** 22ms/step - loss: 520.5569 - mae: 20.9669

Epoch 2/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **1s** 3ms/step - loss: 359.3054 - mae: 16.9523

Epoch 3/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 188.9954 - mae: 11.4004

Epoch 4/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 92.8390 - mae: 7.4983

Epoch 5/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 50.0775 - mae: 5.4443

Epoch 6/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 25.8906 - mae: 3.8419

Epoch 7/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 25.2596 - mae: 3.5675

Epoch 8/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 25.8965 - mae: 3.5270

Epoch 9/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 21.1467 - mae: 3.2489

Epoch 10/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 18.1590 - mae: 2.9120

Epoch 11/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 17.5459 - mae: 2.8981

Epoch 12/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 14.4378 - mae: 2.6548

Epoch 13/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 13.5290 - mae: 2.6923

Epoch 14/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 12.1431 - mae: 2.5697

Epoch 15/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 10.7507 - mae: 2.3912

Epoch 16/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 11.4144 - mae: 2.3533

Epoch 17/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 11.5904 - mae: 2.4676

Epoch 18/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 8.7188 - mae: 2.1791

Epoch 19/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 11.5314 - mae: 2.4788

Epoch 20/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 9.1633 - mae: 2.1908

Epoch 21/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 9.3851 - mae: 2.1914

Epoch 22/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 10.9970 - mae: 2.2566

Epoch 23/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 8.7837 - mae: 2.1307

Epoch 24/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 8.8055 - mae: 2.2339

Epoch 25/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 11.7170 - mae: 2.2802

Epoch 26/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 8.5504 - mae: 2.0922

Epoch 27/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 8.0483 - mae: 2.0955

Epoch 28/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 7.1531 - mae: 1.9194

Epoch 29/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 8.3641 - mae: 2.1605

Epoch 30/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 10.2958 - mae: 2.1187

Epoch 31/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 8.9938 - mae: 2.2357

Epoch 32/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 7.3645 - mae: 2.0156

Epoch 33/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 7.6084 - mae: 1.9493

Epoch 34/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 7.9767 - mae: 2.0362

Epoch 35/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 8.0783 - mae: 1.9972

Epoch 36/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 8.3421 - mae: 2.0897

Epoch 37/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 7.6250 - mae: 1.9784

Epoch 38/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 7.2061 - mae: 1.9374

Epoch 39/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 7.8387 - mae: 1.9595

Epoch 40/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 6.5070 - mae: 1.8752

Epoch 41/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 6.3798 - mae: 1.8787

Epoch 42/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 7.1410 - mae: 1.9415

Epoch 43/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 7.8313 - mae: 1.8986

Epoch 44/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 6.9778 - mae: 1.9105

Epoch 45/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 7.5520 - mae: 1.9204

Epoch 46/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 6.3301 - mae: 1.9136

Epoch 47/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 7.2067 - mae: 1.9521

Epoch 48/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 6.3962 - mae: 1.8071

Epoch 49/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 7.4606 - mae: 1.9042

Epoch 50/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 6.7843 - mae: 1.8989

Epoch 51/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 7.2025 - mae: 1.8790

Epoch 52/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 9.0096 - mae: 2.0120

Epoch 53/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 5.8135 - mae: 1.7516

Epoch 54/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 5.7490 - mae: 1.7592

Epoch 55/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 6.7642 - mae: 1.8043

Epoch 56/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 5.9894 - mae: 1.8529

Epoch 57/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 6.9725 - mae: 1.9170

Epoch 58/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 5.7974 - mae: 1.7825

Epoch 59/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 7.2056 - mae: 1.8771

Epoch 60/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 5.5083 - mae: 1.7468

Epoch 61/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 7.1262 - mae: 1.8259

Epoch 62/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 7.0840 - mae: 1.8427

Epoch 63/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 6.6563 - mae: 1.8353

Epoch 64/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step - loss: 6.1746 - mae: 1.8669

Epoch 65/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 4ms/step - loss: 7.0157 - mae: 1.8457

Epoch 66/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 5ms/step - loss: 6.0990 - mae: 1.8029

Epoch 67/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 4ms/step - loss: 5.8626 - mae: 1.7235

Epoch 68/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 4ms/step - loss: 7.9963 - mae: 1.8987

Epoch 69/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 4ms/step - loss: 6.8193 - mae: 1.8296

Epoch 70/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 4ms/step - loss: 6.4531 - mae: 1.8004

Epoch 71/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 4ms/step - loss: 5.8250 - mae: 1.7665

Epoch 72/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 5ms/step - loss: 6.9098 - mae: 1.7296

Epoch 73/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 4ms/step - loss: 5.4969 - mae: 1.6485

Epoch 74/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 4ms/step - loss: 7.1054 - mae: 1.8316

Epoch 75/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 4ms/step - loss: 6.7147 - mae: 1.7117

Epoch 76/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 4ms/step - loss: 5.2426 - mae: 1.6608

Epoch 77/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 4ms/step - loss: 5.0584 - mae: 1.6460

Epoch 78/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 5ms/step - loss: 4.9712 - mae: 1.5554

Epoch 79/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 4ms/step - loss: 5.3740 - mae: 1.6397

Epoch 80/80

**26/26** ━━━━━━━━━━━━━━━━━━━━ **0s** 4ms/step - loss: 5.2686 - mae: 1.6257

**4/4** ━━━━━━━━━━━━━━━━━━━━ **0s** 36ms/step - loss: 12.4573 - mae: 2.3239

Test MSE: 18.2055

Test MAE: 2.5941

**4/4** ━━━━━━━━━━━━━━━━━━━━ **0s** 39ms/step

Predicted Actual

8.11 7.20

19.34 18.80

22.67 19.00

32.21 27.00

25.69 22.20

20.65 24.50

28.36 31.20

23.21 22.90

20.00 20.50

22.79 23.20

CONCLUSION:

The model built in Python predicted housing prices accurately, illustrating the application of deep learning for regression tasks within the Python ecosystem.

**Experiment 5: Build a Convolution Neural Network for MNISTH and Handwritten Digit Classification.**

AIM:

SOFTWARES: Python libraries, Google Colaboratory, keras, Tensorflow, PyTorch

SOURCE CODE:

import tensorflow as tf

import matplotlib.pyplot as plt

import seaborn as sn

import numpy as np

import math

import pandas as pd

import datetime

import platform

print('Python version:', platform.python\_version())

print('Tensorflow version:', tf.\_\_version\_\_)

print('Keras version:', tf.keras.\_\_version\_\_)

# Load the data

mnist\_dataset = tf.keras.datasets.mnist

(x\_train, y\_train), (x\_test, y\_test) = mnist\_dataset.load\_data()

print('x\_train:', x\_train.shape)

print('y\_train:', y\_train.shape)

print('x\_test:', x\_test.shape)

print('y\_test:', y\_test.shape)

# Set image/reshape constants

(\_, IMAGE\_WIDTH, IMAGE\_HEIGHT) = x\_train.shape

IMAGE\_CHANNELS = 1

# Visualize first few samples

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

for i in range(25):

    plt.subplot(5, 5, i+1)

    plt.xticks([])

    plt.yticks([])

    plt.grid(False)

    plt.imshow(x\_train[i], cmap=plt.cm.binary)

    plt.xlabel(y\_train[i])

plt.show()

# Reshape and normalize data

x\_train = x\_train.reshape(x\_train.shape[0], IMAGE\_WIDTH, IMAGE\_HEIGHT, IMAGE\_CHANNELS) / 255.0

x\_test = x\_test.reshape(x\_test.shape[0], IMAGE\_WIDTH, IMAGE\_HEIGHT, IMAGE\_CHANNELS) / 255.0

# Build the CNN model

model = tf.keras.models.Sequential([

    tf.keras.layers.Conv2D(8, kernel\_size=5, activation='relu', input\_shape=(IMAGE\_WIDTH, IMAGE\_HEIGHT, IMAGE\_CHANNELS), kernel\_initializer='he\_uniform'),

    tf.keras.layers.MaxPooling2D(pool\_size=(2, 2)),

    tf.keras.layers.Conv2D(16, kernel\_size=5, activation='relu', kernel\_initializer='he\_uniform'),

    tf.keras.layers.MaxPooling2D(pool\_size=(2, 2)),

    tf.keras.layers.Flatten(),

    tf.keras.layers.Dense(128, activation='relu'),

    tf.keras.layers.Dropout(0.2),

    tf.keras.layers.Dense(10, activation='softmax')

])

model.summary()

# (optional) plot model structure:

tf.keras.utils.plot\_model(model, show\_shapes=True, show\_layer\_names=True)

# Compile the model

adam\_optimizer = tf.keras.optimizers.Adam(learning\_rate=0.001)

model.compile(optimizer=adam\_optimizer, loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

# Train model with validation

log\_dir=".logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

tensorboard\_callback = tf.keras.callbacks.TensorBoard(log\_dir=log\_dir, histogram\_freq=1)

history = model.fit(

    x\_train, y\_train,

    epochs=10,

    validation\_data=(x\_test, y\_test),

    callbacks=[tensorboard\_callback]

)

# Plot loss and accuracy curves

plt.figure()

plt.plot(history.history['loss'], label='Train Loss')

plt.plot(history.history['val\_loss'], label='Val Loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.legend()

plt.show()

plt.figure()

plt.plot(history.history['accuracy'], label='Train Acc')

plt.plot(history.history['val\_accuracy'], label='Val Acc')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend()

plt.show()

# Evaluate the model

train\_loss, train\_accuracy = model.evaluate(x\_train, y\_train, verbose=0)

val\_loss, val\_accuracy = model.evaluate(x\_test, y\_test, verbose=0)

print('Training loss: ', train\_loss)

print('Training accuracy: ', train\_accuracy)

print('Validation loss: ', val\_loss)

print('Validation accuracy: ', val\_accuracy)

# Save and load the model

model\_name = 'digits\_recognition\_cnn.h5'

model.save(model\_name, save\_format='h5')

loaded\_model = tf.keras.models.load\_model(model\_name)

# Predict on the test set

predictions\_one\_hot = loaded\_model.predict(x\_test)

predictions = np.argmax(predictions\_one\_hot, axis=1)

# Display a sample prediction

plt.imshow(x\_test[0].reshape((IMAGE\_WIDTH, IMAGE\_HEIGHT)), cmap=plt.cm.binary)

plt.title(f'Predicted: {predictions[0]}, Actual: {y\_test[0]}')

plt.show()

# Visualize first 36 predictions

plt.figure(figsize=(15, 15))

for i in range(36):

    plt.subplot(6, 6, i+1)

    plt.xticks([])

    plt.yticks([])

    plt.grid(False)

    col = 'green' if predictions[i]==y\_test[i] else 'red'

    plt.imshow(x\_test[i].reshape((IMAGE\_WIDTH, IMAGE\_HEIGHT)), cmap=plt.cm.binary)

    plt.xlabel(f'{predictions[i]}', color=col)

plt.tight\_layout()

plt.show()

# Confusion matrix

confusion\_matrix = tf.math.confusion\_matrix(y\_test, predictions).numpy()

plt.figure(figsize=(9, 7))

sn.heatmap(confusion\_matrix, annot=True, fmt="d", linewidths=.5, square=True)

plt.ylabel('Actual')

plt.xlabel('Predicted')

plt.show()

OUTPUT:

Python version: 3.11.13

Tensorflow version: 2.18.0

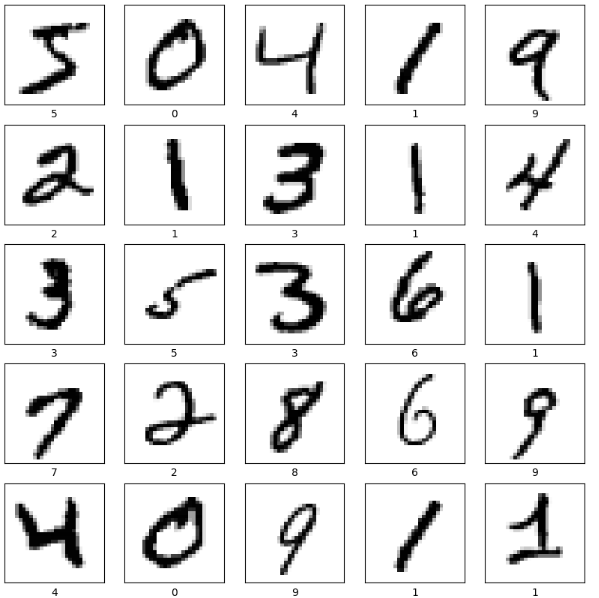
Keras version: 3.8.0

x\_train: (60000, 28, 28)

y\_train: (60000,)

x\_test: (10000, 28, 28)

y\_test: (10000,)



/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base\_conv.py:107: UserWarning: Do not pass an `input\_shape`/`input\_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

super().\_\_init\_\_(activity\_regularizer=activity\_regularizer, \*\*kwargs)

**Model: "sequential\_8"**

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓

┃ **Layer (type)** ┃ **Output Shape** ┃ **Param #** ┃

┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩

│ conv2d\_6 (Conv2D) │ (None, 24, 24, 8) │ 208 │

├─────────────────────────────────┼────────────────────────┼───────────────┤

│ max\_pooling2d\_6 (MaxPooling2D) │ (None, 12, 12, 8) │ 0 │

├─────────────────────────────────┼────────────────────────┼───────────────┤

│ conv2d\_7 (Conv2D) │ (None, 8, 8, 16) │ 3,216 │

├─────────────────────────────────┼────────────────────────┼───────────────┤

│ max\_pooling2d\_7 (MaxPooling2D) │ (None, 4, 4, 16) │ 0 │

├─────────────────────────────────┼────────────────────────┼───────────────┤

│ flatten\_3 (Flatten) │ (None, 256) │ 0 │

├─────────────────────────────────┼────────────────────────┼───────────────┤

│ dense\_21 (Dense) │ (None, 128) │ 32,896 │

├─────────────────────────────────┼────────────────────────┼───────────────┤

│ dropout\_3 (Dropout) │ (None, 128) │ 0 │

├─────────────────────────────────┼────────────────────────┼───────────────┤

│ dense\_22 (Dense) │ (None, 10) │ 1,290 │

└─────────────────────────────────┴────────────────────────┴───────────────┘

**Total params:** 37,610 (146.91 KB)

**Trainable params:** 37,610 (146.91 KB)

**Non-trainable params:** 0 (0.00 B)

Epoch 1/10

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **10s** 4ms/step - accuracy: 0.8696 - loss: 0.4139 - val\_accuracy: 0.9810 - val\_loss: 0.0577

Epoch 2/10

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **8s** 4ms/step - accuracy: 0.9773 - loss: 0.0722 - val\_accuracy: 0.9864 - val\_loss: 0.0411

Epoch 3/10

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **8s** 4ms/step - accuracy: 0.9838 - loss: 0.0518 - val\_accuracy: 0.9839 - val\_loss: 0.0470

Epoch 4/10

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **7s** 4ms/step - accuracy: 0.9875 - loss: 0.0393 - val\_accuracy: 0.9894 - val\_loss: 0.0298

Epoch 5/10

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **10s** 4ms/step - accuracy: 0.9900 - loss: 0.0324 - val\_accuracy: 0.9903 - val\_loss: 0.0304

Epoch 6/10

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **7s** 4ms/step - accuracy: 0.9922 - loss: 0.0246 - val\_accuracy: 0.9911 - val\_loss: 0.0296

Epoch 7/10

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **10s** 4ms/step - accuracy: 0.9929 - loss: 0.0221 - val\_accuracy: 0.9909 - val\_loss: 0.0279

Epoch 8/10

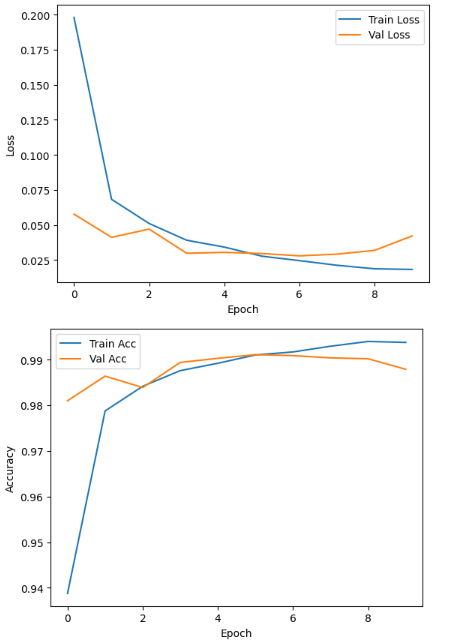
**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **7s** 4ms/step - accuracy: 0.9940 - loss: 0.0173 - val\_accuracy: 0.9904 - val\_loss: 0.0292

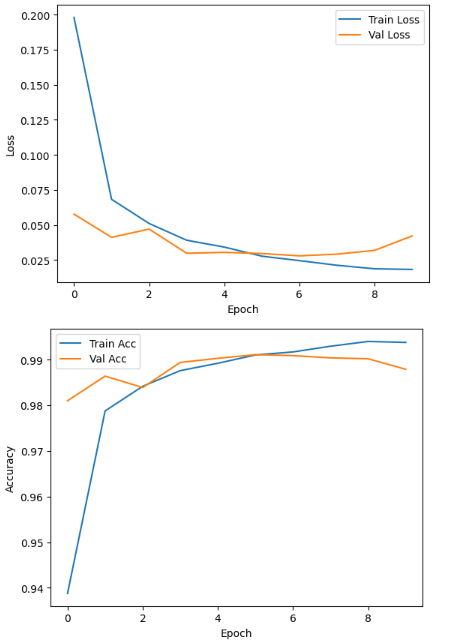
Epoch 9/10

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **7s** 4ms/step - accuracy: 0.9939 - loss: 0.0179 - val\_accuracy: 0.9902 - val\_loss: 0.0319

Epoch 10/10

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **11s** 4ms/step - accuracy: 0.9941 - loss: 0.0170 - val\_accuracy: 0.9879 - val\_loss: 0.0422





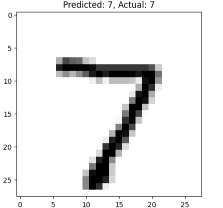
Training loss: 0.017138423398137093

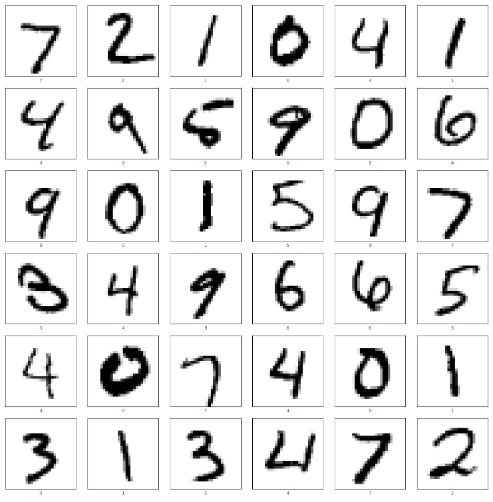
Training accuracy: 0.9944999814033508

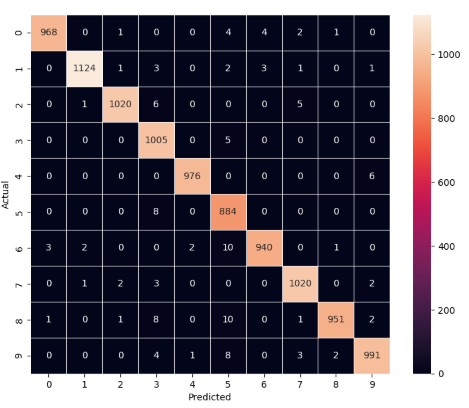
Validation loss: 0.04216663911938667

Validation accuracy: 0.9879000186920166

**313/313** ━━━━━━━━━━━━━━━━━━━━ **1s** 3ms/step







CONCLUSION:

The Python-implemented CNN improved classification accuracy over basic networks, showcasing Python’s power for more advanced deep learning models in image recognition.

**Experiment 6: Build a Convolution Neural Network for simple image (dogs and Cats) Classification.**

AIM: To develop a Python CNN model to classify images of dogs and cats.

SOFTWARES: Python libraries, Google Colaboratory, keras, Tensorflow, PyTorch

SOURCE CODE:

OUTPUT:

CONCLUSION:

By building a CNN in Python, we were able to differentiate dog and cat images with high accuracy, further establishing Python libraries as essential tools for image-related AI tasks.

**Experiment 7: Use a pre-trained convolution neural network (VGG16) for image classification.**

AIM: To use a pre-trained VGG16 neural network in Python for image classification.

SOFTWARES: Python libraries, Google Colaboratory, keras, Tensorflow, PyTorch

SOURCE CODE:

OUTPUT:

CONCLUSION:

Employing VGG16 in a Python environment allowed for efficient and accurate image classification, emphasizing the benefits of transfer learning and the Python deep learning ecosystem.

**Experiment 8: Implement one hot encoding of words or characters.**

AIM: To implement one-hot encoding in Python for converting text into numerical format for machine learning models.

SOFTWARES: Python libraries, Google Colaboratory, keras, Tensorflow, PyTorch

SOURCE CODE:

OUTPUT:

CONCLUSION:

One-hot encoding was efficiently implemented using Python libraries, preparing text data for subsequent modeling and highlighting the method’s simplicity in Python.

**Experiment 9: Implement word embeddings for IMDB dataset.**

AIM: To generate and use word embeddings in Python for the IMDB movie review dataset.

SOFTWARES: Python libraries, Google Colaboratory, keras, Tensorflow, PyTorch

SOURCE CODE:

OUTPUT:

CONCLUSION:

Python code enabled the creation of dense word embeddings, boosting model performance and text understanding, proving the importance of embeddings in modern NLP using Python.

**Experiment 10: Implement a Recurrent Neural Network for IMDB Movie review classification problem.**

AIM: To implement a recurrent neural network in Python for sentiment classification of movie reviews from the IMDB dataset.

SOFTWARES: Python libraries, Google Colaboratory, keras, Tensorflow, PyTorch

SOURCE CODE:

OUTPUT:

CONCLUSION:

The RNN model coded in Python captured the sequential nature of movie review text, resulting in accurate sentiment classification and showcasing Python’s suitability for sequence modeling tasks.