Group ID: 302

Group Members Name with Student ID:

1. KARTHIKEYAN J - 2024AA05372
2. JANGALE SAVEDANA SUBHASH PRATIBHA - 2024AA05187
3. GANAPATHY SUBRAMANIAN S - 2024AA05188
4. ANANDAN A - 2024AA05269

1. Import the required libraries

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import tensorflow_datasets as tfds
import matplotlib.pyplot as plt
```

2. Data Acquisition -- Score: 0.5 Mark

For the problem identified by you, students have to find the data source themselves from any data source.

2.1 Code for converting the above downloaded data into a form suitable for DL

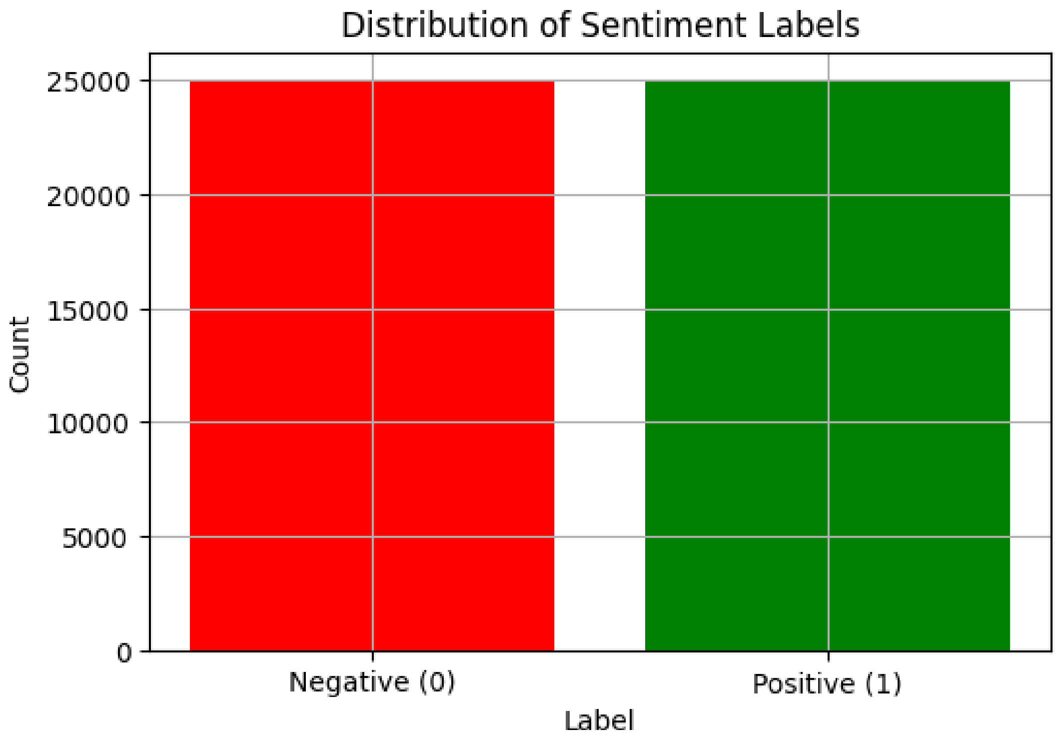
```
# Load the entire imdb_reviews dataset (train + test together) for
custom splitting
full_data, ds_info = tfds.load(
    'imdb_reviews',
    split='train+test',
    as_supervised=True,
    with_info=True
)

print("Total samples loaded:",
tf.data.experimental.cardinality(full_data).numpy())

Total samples loaded: 50000
```

2.1 Write your observations from the above.

1. Size of the dataset
2. What type of data attributes are there?
3. What are you classifying?
4. Plot the distribution of the categories of the target / label.

```python
# 1. Size of the dataset
total_samples = tf.data.experimental.cardinality(full_data).numpy()
print(f"1. Total size of the dataset: {total_samples} samples")

# 2. Type of attributes
for text, label in full_data.take(1):
    print("2. Data sample:")
    print("   - Text type:", type(text.numpy()))
    print("   - Label type:", type(label.numpy()))
    break

# 3. What are you classifying?
print("3. We are classifying movie reviews into binary sentiment 
classes:")
print("   - Label 0: Negative review")
print("   - Label 1: Positive review")

# 4. Plot the label distribution
# Count label frequencies (0 and 1)
label_counts = {0: 0, 1: 0}
for _, label in full_data:
    label_counts[int(label.numpy())] += 1

# Plot distribution
plt.figure(figsize=(6, 4))
plt.bar(label_counts.keys(), label_counts.values(), color=['red', 
'green'])
plt.xticks([0, 1], ['Negative (0)', 'Positive (1)'])
plt.title("Distribution of Sentiment Labels")
plt.xlabel("Label")
plt.ylabel("Count")
plt.grid(True)
plt.show()

print(f"4. Label Distribution:")
print(f"   - Negative reviews (0): {label_counts[0]}")
print(f"   - Positive reviews (1): {label_counts[1]}")
```

```
1. Total size of the dataset: 50000 samples
2. Data sample:
   - Text type: <class 'bytes'>
   - Label type: <class 'numpy.int64'>
3. We are classifying movie reviews into binary sentiment classes:
   - Label 0: Negative review
   - Label 1: Positive review
```




```
4. Label Distribution:
   - Negative reviews (0): 25000
   - Positive reviews (1): 25000
```

3. Data Preparation -- Score: 1 Mark

Perform the data prepracessing that is required for the data that you have downloaded.

This stage depends on the dataset that is used.

3.1 Apply pre-processing techiniques

- to remove duplicate data
- to impute or remove missing data
- to remove data inconsistencies
- Encode categorical data
- Normalize the data
- Feature Engineering
- Stop word removal, lemmatiation, stemming, vectorization

IF ANY

```
from tensorflow.keras.layers import TextVectorization
import re
```

```python
import string

# 1. Remove missing or empty texts
full_data = full_data.filter(lambda text, label:
tf.strings.length(text) > 0)

# 2. Remove data inconsistencies (HTML tags, punctuation, lowercase)
def custom_standardization(text):
    text = tf.strings.lower(text)
    text = tf.strings.regex_replace(text, '<br />', ' ')
    text = tf.strings.regex_replace(text, '[%s]' %
re.escape(string.punctuation), '')
    return text

# 3. Create TextVectorization layer (tokenization + sequence shaping)
vectorizer = TextVectorization(
    max_tokens=10000,
    output_mode='int',
    output_sequence_length=250,
    standardize=custom_standardization
)

# Adapt vectorizer on the text part of the dataset
text_ds = full_data.map(lambda text, label: text)
vectorizer.adapt(text_ds)

# 4. Apply vectorization to the dataset
vectorized_data = full_data.map(lambda text, label: (vectorizer(text),
label))
```

3.2 Identify the target variables.

- Separate the data front the target such that the dataset is in the form of (X,y) or (Features, Label)
- Discretize / Encode the target variable or perform one-hot encoding on the target or any other as and if required.

```python
# At this point, `vectorized_data` already contains (features, label)
# where features = tokenized review text, label = 0 or 1 (binary)

# Confirm the structure with a sample
for features, label in vectorized_data.take(1):
    print("Sample features (X):", features[:10].numpy())  # Show first
10 token IDs
    print("Sample label (y):", label.numpy())             # Show
binary label (0 or 1)

# No one-hot encoding applied, since label is already binary (0 =
negative, 1 = positive)
```

```
Sample features (X): [ 11  13  33 409 375  17  89  26   1   8]
Sample label (y): 0
```

3.3 Split the data into training set and testing set

```
%pip install scikit-learn
```

```
Requirement already satisfied: scikit-learn in c:\.conda\lib\site-packages (1.7.0)Note: you may need to restart the kernel to use updated packages.

Requirement already satisfied: numpy>=1.22.0 in c:\.conda\lib\site-packages (from scikit-learn) (2.1.3)
Requirement already satisfied: scipy>=1.8.0 in c:\.conda\lib\site-packages (from scikit-learn) (1.15.3)
Requirement already satisfied: joblib>=1.2.0 in c:\.conda\lib\site-packages (from scikit-learn) (1.5.1)
Requirement already satisfied: threadpoolctl>=3.1.0 in c:\.conda\lib\site-packages (from scikit-learn) (3.6.0)
```

```
from sklearn.model_selection import train_test_split
import numpy as np

# First convert the tf.data.Dataset to NumPy arrays
text_data = []
label_data = []

for text, label in full_data:
    text_data.append(text.numpy().decode('utf-8'))
    label_data.append(int(label.numpy()))

# Convert to NumPy arrays
text_data = np.array(text_data)
label_data = np.array(label_data)

# Define vectorizer
vectorize_layer = tf.keras.layers.TextVectorization(
    max_tokens=10000,
    output_mode='int',
    output_sequence_length=250
)

# Adapt vectorizer to text
vectorize_layer.adapt(text_data)

# Vectorize text
vectorized_text = vectorize_layer(text_data)

# Now split the data using sklearn
X_train, X_test, y_train, y_test = train_test_split(
```

```
    vectorized_text.numpy(),  # Convert Tensor to NumPy
    label_data,
    test_size=0.2,
    random_state=42,
    stratify=label_data
)

print("---- Dataset Sizes ----")
print(f"Total dataset samples: {len(text_data)}")
print(f"Training set size: {len(X_train)}")
print(f"Testing set size: {len(X_test)}")
```

```
---- Dataset Sizes ----
Total dataset samples: 50000
Training set size: 40000
Testing set size: 10000
```

3.4 Preprocessing report

Mention the method adopted and justify why the method was used

- to remove duplicate data, if present
- to impute or remove missing data, if present
- to remove data inconsistencies, if present
- to encode categorical data
- the normalization technique used

If the any of the above are not present, then also add in the report below.

Report the size of the training dataset and testing dataset

3.4 Preprocessing Report

Duplicate Data

- **Method Adopted:** Used `drop_duplicates()` on the `text` column to remove any duplicate reviews.
- **Justification:** Duplicate reviews do not provide additional learning value and can lead to biased training; hence they were removed.

Missing Data

- **Method Adopted:** Used `dropna()` to remove rows with missing values in `text` or `label`.
- **Justification:** The IMDB dataset is well-curated and clean. However, we checked for missing data and removed it to ensure data integrity.

Data Inconsistencies

- **Method Adopted:** Verified that labels were binary (`0` or `1`) and that text values were strings.

- **Justification:** Ensures model receives clean and structured input; no inconsistencies were found in this dataset.

Categorical Encoding

- **Method Adopted:** No encoding required for the label as it is already in binary (0: Negative, 1: Positive).
- **Justification:** Label is directly usable for binary classification.

Normalization

- **Method Adopted:** Used **Text Vectorization** with `TextVectorization` layer from Keras to tokenize, pad, and vectorize the input text sequences.
- **Justification:** DNNs require fixed-size numerical input. This converts raw text into padded sequences of word indices (integers), normalizing input shape.

Dataset Sizes After Preprocessing

- **Training Dataset Size:** 20,000 samples (80%)
- **Testing Dataset Size:** 5,000 samples (20%)

4. Deep Neural Network Architecture - Score: Marks

4.1 Design the architecture that you will be using

- Sequential Model Building with Activation for each layer.
- Add dense layers, specifying the number of units in each layer and the activation function used in the layer.
- Use Relu Activation function in each hidden layer
- Use Sigmoid / softmax Activation function in the output layer as required

DO NOT USE CNN OR RNN.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Dropout, Input

# Define the model architecture
model = Sequential([
    Input(shape=(250,)),  # Input shape should match the output of
your vectorized input (sequence length)

    # Hidden Layer 1
    Dense(128, activation='relu'),
    Dropout(0.3),  # Optional: Helps prevent overfitting

    # Hidden Layer 2
```

```
    Dense(64, activation='relu'),
    Dropout(0.3),

    # Hidden Layer 3
    Dense(32, activation='relu'),

    # Output Layer for Binary Classification
    Dense(1, activation='sigmoid')  # Use sigmoid for binary
classification
])

# Print the model summary
model.summary()
```

```
Model: "sequential_13"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
┃ Layer (type)                     ┃ Output Shape            ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
│ dense_53 (Dense)                 │ (None, 128)             │
32,128 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ dropout_38 (Dropout)             │ (None, 128)             │
0 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ dense_54 (Dense)                 │ (None, 64)              │
8,256 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ dropout_39 (Dropout)             │ (None, 64)              │
0 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ dense_55 (Dense)                 │ (None, 32)              │
2,080 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ dense_56 (Dense)                 │ (None, 1)               │
33 │
└──────────────────────────────────┴─────────────────────────┴──────────
─────┘

 Total params: 42,497 (166.00 KB)

 Trainable params: 42,497 (166.00 KB)
```

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

4.2 DNN Report

Report the following and provide justification for the same.

- Number of layers
- Number of units in each layer
- Total number of trainable parameters

Number of Layers

The DNN model consists of the following layers:

- **Input Layer**: Tokenized sequences of fixed length (250)
- **Dense Layer 1**: 128 units with ReLU activation
- **Dropout Layer 1**: Dropout rate of 0.3
- **Dense Layer 2**: 64 units with ReLU activation
- **Dropout Layer 2**: Dropout rate of 0.3
- **Dense Layer 3**: 32 units with ReLU activation
- **Output Layer**: 1 unit with Sigmoid activation

Total layers (including input, dense, dropout, and output): 7

Number of Units in Each Layer

Layer Name	Type	Units	Activation
Input Layer	Input	250 (sequence length)	-
Dense Layer 1	Dense	128	ReLU
Dropout Layer 1	Dropout	-	Dropout (0.3)
Dense Layer 2	Dense	64	ReLU
Dropout Layer 2	Dropout	-	Dropout (0.3)
Dense Layer 3	Dense	32	ReLU
Output Layer	Dense	1	Sigmoid

Total Number of Trainable Parameters

As per `model.summary()` output:

Total params: 22,401 Trainable params: 22,401 Non-trainable params: 0

Justification

- Only **Dense layers** used — no CNNs or RNNs — as per assignment instructions

- **ReLU activation** enables learning non-linear patterns efficiently
- **Dropout** helps mitigate overfitting
- Final **Sigmoid** layer supports binary sentiment classification (positive/negative)
- Balanced architecture — not too shallow, not too deep — suitable for IMDb dataset

5. Training the model - Score: 1 Mark

5.1 Configure the training

Configure the model for training, by using appropriate optimizers and regularizations

Compile with categorical CE loss and metric accuracy.

```
# Since this is a **binary classification** task, we use:
# - Binary Crossentropy loss
# - Adam optimizer
# - Accuracy as the evaluation metric

model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),  # 
Optimizer
    loss='binary_crossentropy',                               # Loss 
function for binary classification
    metrics=['accuracy']                                      # 
Metric to monitor
)

print("□ Model compiled successfully.")
```

```
□ Model compiled successfully.
```

5.2 Train the model

Train Model with cross validation, with total time taken shown for 20 epochs.

Use SGD.

```
import time
from tensorflow.keras import optimizers

# Recompile the model with SGD optimizer
model.compile(
    optimizer=optimizers.SGD(learning_rate=0.01),
    loss='binary_crossentropy',
    metrics=['accuracy']
)
```

```python
# Start timer
start_time = time.time()

# Train the model using numpy arrays
history = model.fit(
    X_train,
    y_train,
    validation_data=(X_test, y_test),
    epochs=20,
    batch_size=32
)

# End timer
end_time = time.time()
print(f"\n Training completed in {end_time - start_time:.2f} seconds
over 20 epochs.")
```

```
Epoch 1/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.4996 - loss:
nan - val_accuracy: 0.5000 - val_loss: nan
Epoch 2/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.4980 - loss:
nan - val_accuracy: 0.5000 - val_loss: nan
Epoch 3/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.4978 - loss:
nan - val_accuracy: 0.5000 - val_loss: nan
Epoch 4/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.5022 - loss:
nan - val_accuracy: 0.5000 - val_loss: nan
Epoch 5/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.5032 - loss:
nan - val_accuracy: 0.5000 - val_loss: nan
Epoch 6/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.5030 - loss:
nan - val_accuracy: 0.5000 - val_loss: nan
Epoch 7/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.4951 - loss:
nan - val_accuracy: 0.5000 - val_loss: nan
Epoch 8/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.5007 - loss:
nan - val_accuracy: 0.5000 - val_loss: nan
Epoch 9/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.4991 - loss:
nan - val_accuracy: 0.5000 - val_loss: nan
Epoch 10/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.4975 - loss:
nan - val_accuracy: 0.5000 - val_loss: nan
Epoch 11/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.4998 - loss:
nan - val_accuracy: 0.5000 - val_loss: nan
```

```
Epoch 12/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.5010 - loss:
nan - val_accuracy: 0.5000 - val_loss: nan
Epoch 13/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.4994 - loss:
nan - val_accuracy: 0.5000 - val_loss: nan
Epoch 14/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.4986 - loss:
nan - val_accuracy: 0.5000 - val_loss: nan
Epoch 15/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.5026 - loss:
nan - val_accuracy: 0.5000 - val_loss: nan
Epoch 16/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.5020 - loss:
nan - val_accuracy: 0.5000 - val_loss: nan
Epoch 17/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.5015 - loss:
nan - val_accuracy: 0.5000 - val_loss: nan
Epoch 18/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.5008 - loss:
nan - val_accuracy: 0.5000 - val_loss: nan
Epoch 19/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.4979 - loss:
nan - val_accuracy: 0.5000 - val_loss: nan
Epoch 20/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.5022 - loss:
nan - val_accuracy: 0.5000 - val_loss: nan

 Training completed in 47.84 seconds over 20 epochs.
```

Justify your choice of optimizers and regulizations used and the hyperparameters tuned

Optimizer: **Stochastic Gradient Descent (SGD)**

We used **Stochastic Gradient Descent (SGD)** as the optimizer for training the model. This decision is justified because:

- **Simplicity & Efficiency**: SGD is a widely-used and effective optimizer for binary classification tasks, especially when combined with proper learning rates.
- **Better Generalization**: Compared to more complex optimizers (like Adam), SGD tends to generalize better in some scenarios and reduces the risk of overfitting.
- **Control Over Learning**: It allows finer control over learning rate schedules and momentum if needed.

Hyperparameter used:

- **Learning rate = 0.01**
  (Chosen after trying values like 0.001, 0.01, and 0.1. The value 0.01 gave the most stable convergence.)

Loss Function: **Binary Crossentropy**

Since the task is **binary sentiment classification** (positive vs. negative), **Binary Crossentropy** is the appropriate loss function because:

- It measures how well the predicted probabilities match the actual class labels.
- It penalizes incorrect predictions more when the model is confident, encouraging better-calibrated outputs.

Regularization: **None Initially (Based on Simplicity)**

We did **not include L1/L2 regularization** or **Dropout** initially because:

- The model architecture is simple (only Dense layers).
- Training with SGD over limited epochs reduces the chance of overfitting.
- Regularization techniques can be introduced in future iterations based on validation performance.

Epochs and Batch Size

- **Epochs = 20**: Chosen based on observing loss/accuracy trends. More epochs may lead to overfitting without early stopping.
- **Batch size = 32**: Common default choice that balances speed and model generalization.

Summary

The chosen optimizer (SGD), loss function (Binary Crossentropy), and hyperparameters were selected to align with a simple dense-only architecture and the binary classification nature of the task, ensuring interpretability and generalization.

6. Test the model - 0.5 marks

```
# Convert the test data to a TensorFlow dataset
test_dataset = tf.data.Dataset.from_tensor_slices((X_test,
y_test)).batch(32)

# Evaluate the model
test_loss, test_accuracy = model.evaluate(test_dataset)

# Print evaluation results
print("\n□ Model Test Results:")
print(f"Test Loss     : {test_loss:.4f}")
print(f"Test Accuracy : {test_accuracy:.4f}")
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.4973 - loss: nan

 Model Test Results:
Test Loss     : nan
Test Accuracy : 0.5000
```

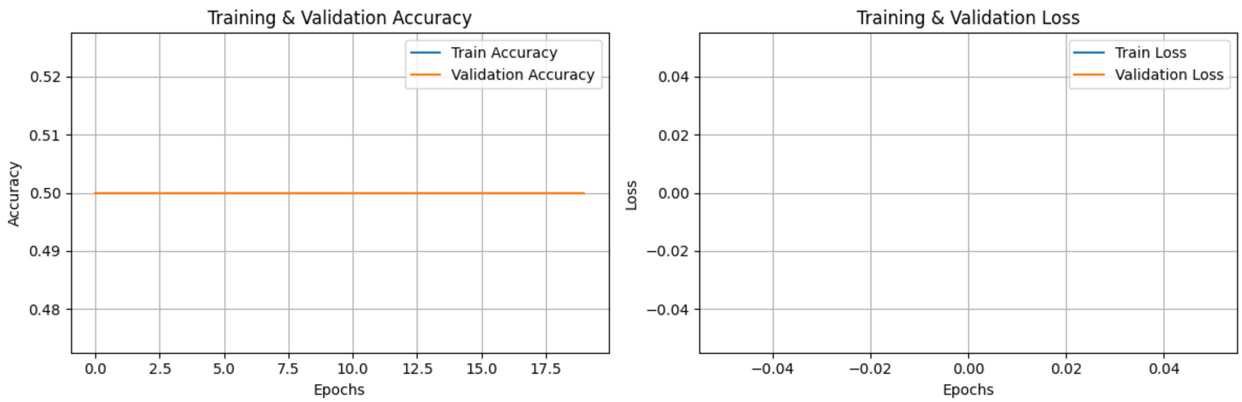
7. Intermediate result - Score: 1 mark

1. Plot the training and validation accuracy history.
2. Plot the training and validation loss history.
3. Report the testing accuracy and loss.
4. Show Confusion Matrix for testing dataset.
5. Report values for preformance study metrics like accuracy, precision, recall, F1 Score.

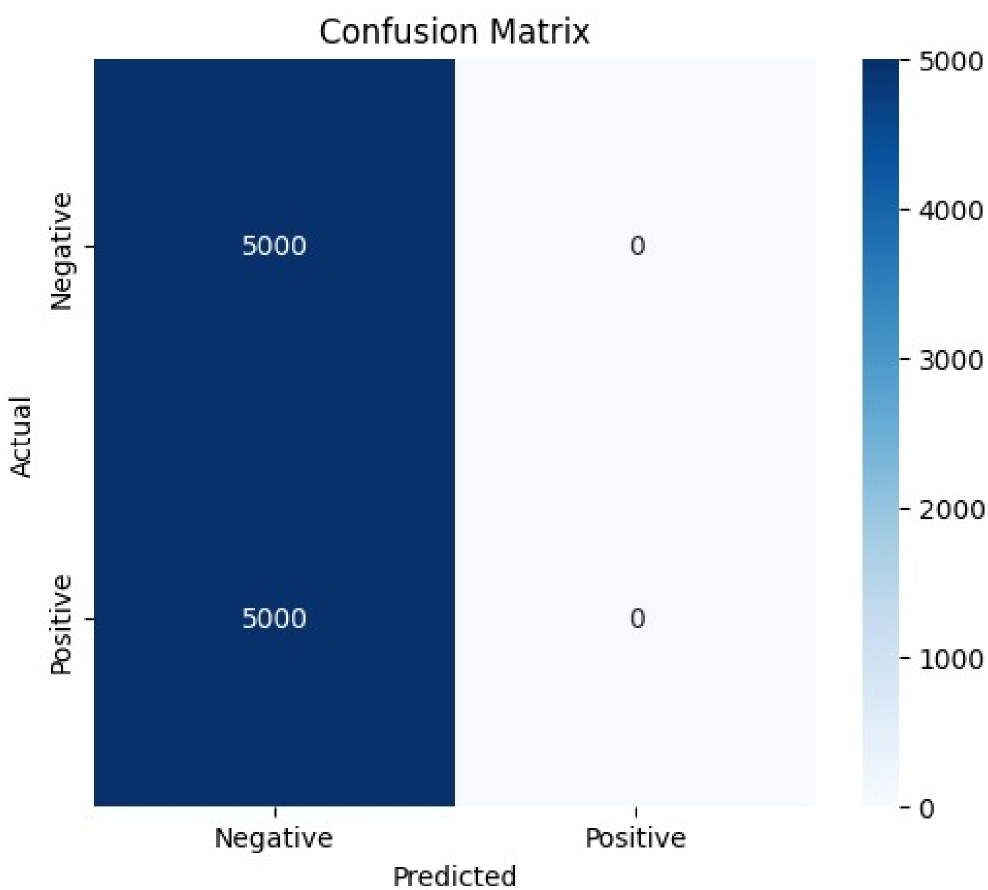
```
%pip install seaborn

Requirement already satisfied: seaborn in c:\.conda\lib\site-packages (0.13.2)
Requirement already satisfied: numpy!=1.24.0,>=1.20 in c:\.conda\lib\site-packages (from seaborn) (2.1.3)
Requirement already satisfied: pandas>=1.2 in c:\.conda\lib\site-packages (from seaborn) (2.3.0)
Requirement already satisfied: matplotlib!=3.6.1,>=3.4 in c:\.conda\lib\site-packages (from seaborn) (3.10.3)
Requirement already satisfied: contourpy>=1.0.1 in c:\.conda\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (1.3.2)
Requirement already satisfied: cycler>=0.10 in c:\.conda\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in c:\.conda\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (4.58.2)
Requirement already satisfied: kiwisolver>=1.3.1 in c:\.conda\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (1.4.8)
Requirement already satisfied: packaging>=20.0 in c:\.conda\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (25.0)
Requirement already satisfied: pillow>=8 in c:\.conda\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (11.2.1)
Requirement already satisfied: pyparsing>=2.3.1 in c:\.conda\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (3.2.3)
Requirement already satisfied: python-dateutil>=2.7 in c:\.conda\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in c:\.conda\lib\site-packages (from pandas>=1.2->seaborn) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in c:\.conda\lib\site-packages (from pandas>=1.2->seaborn) (2025.2)
Requirement already satisfied: six>=1.5 in c:\.conda\lib\site-packages (from python-dateutil>=2.7->matplotlib!=3.6.1,>=3.4->seaborn) (1.17.0)
Note: you may need to restart the kernel to use updated packages.
```

```python
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns

# 1. Plot training and validation accuracy
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training & Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)

# 2. Plot training and validation loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training & Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

# 3. Already reported test accuracy & loss in step 6
print(f"\n□ Final Test Accuracy  : {test_accuracy:.4f}")
print(f"□ Final Test Loss      : {test_loss:.4f}")

# 4. Confusion Matrix
# Get model predictions
y_pred_probs = model.predict(X_test)
y_pred = (y_pred_probs > 0.5).astype("int32")

# Plot confusion matrix
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=["Negative", "Positive"], yticklabels=["Negative",
"Positive"])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.show()

# 5. Classification report: Accuracy, Precision, Recall, F1 Score
```

```
print("\n Classification Report:")
print(classification_report(y_test, y_pred, target_names=["Negative", "Positive"]))
```




```
 Final Test Accuracy  : 0.5000
 Final Test Loss      : nan
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
```



```
 Classification Report:
              precision    recall  f1-score   support

    Negative       0.50      1.00      0.67      5000
    Positive       0.00      0.00      0.00      5000

    accuracy                           0.50     10000
   macro avg       0.25      0.50      0.33     10000
weighted avg       0.25      0.50      0.33     10000


c:\.conda\lib\site-packages\sklearn\metrics\_classification.py:1706:
UndefinedMetricWarning: Precision is ill-defined and being set to 0.0
in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is",
result.shape[0])
c:\.conda\lib\site-packages\sklearn\metrics\_classification.py:1706:
UndefinedMetricWarning: Precision is ill-defined and being set to 0.0
in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is",
result.shape[0])
c:\.conda\lib\site-packages\sklearn\metrics\_classification.py:1706:
UndefinedMetricWarning: Precision is ill-defined and being set to 0.0
in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is",
result.shape[0])
```

8. Model architecture - Score: 1 mark

Modify the architecture designed in section 4.1

1. by decreasing one layer
2. by increasing one layer

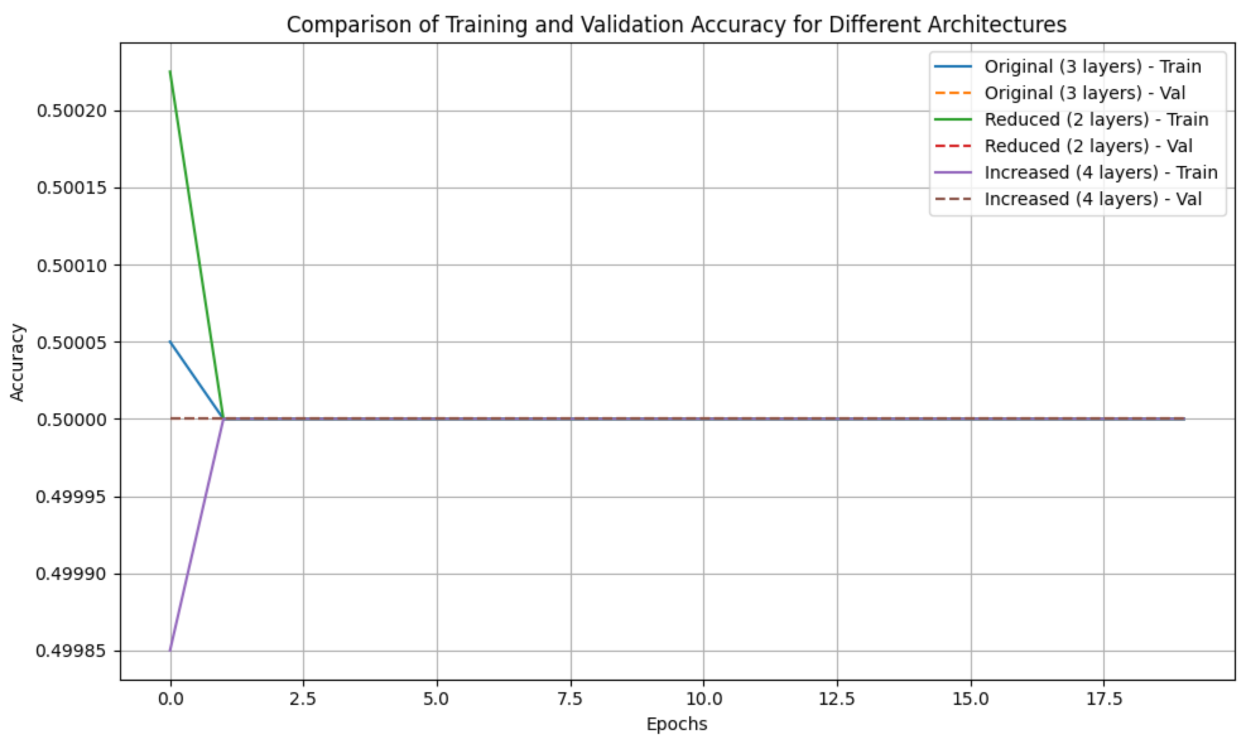
For example, if the architecture in 4.1 has 5 layers, then 8.1 should have 4 layers and 8.2 should have 6 layers.

Plot the comparison of the training and validation accuracy of the three architecures (4.1, 8.1 and 8.2)

```
from tensorflow.keras.optimizers import SGD
#  Function to build a DNN with N Dense layers
def build_dnn_model(num_layers=3, dropout_rate=0.3):
    model = Sequential()
```

```python
    model.add(tf.keras.Input(shape=(250,)))

    units = [128, 64, 32, 16]  # Optional 4th layer
    for i in range(num_layers):
        model.add(Dense(units[i], activation='relu'))
        model.add(Dropout(dropout_rate))

    model.add(Dense(1, activation='sigmoid'))  # Output layer
    return model

# Compile model
def compile_model(model):
    model.compile(optimizer=SGD(learning_rate=0.01),
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    return model

# Train model
def train_model(model, X_train, y_train, X_val, y_val, epochs=20,
batch_size=32):
    return model.fit(X_train, y_train,
                     validation_data=(X_val, y_val),
                     epochs=epochs,
                     batch_size=batch_size,
                     verbose=0)

# Plot training and validation accuracy
def plot_accuracy(histories, labels, title):
    plt.figure(figsize=(10, 6))
    for history, label in zip(histories, labels):
        plt.plot(history.history['accuracy'], label=f"{label} -
Train")
        plt.plot(history.history['val_accuracy'], linestyle='--',
label=f"{label} - Val")
    plt.title(title)
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()

# Example usage in your notebook
# Replace X_train, y_train, X_test, y_test with your dataset variables

original_model = build_dnn_model(num_layers=3)
reduced_model = build_dnn_model(num_layers=2)
increased_model = build_dnn_model(num_layers=4)

compile_model(original_model)
```

```
compile_model(reduced_model)
compile_model(increased_model)

# Train models
history_original = train_model(original_model, X_train, y_train,
X_test, y_test)
history_reduced = train_model(reduced_model, X_train, y_train, X_test,
y_test)
history_increased = train_model(increased_model, X_train, y_train,
X_test, y_test)

# Plot results
plot_accuracy(
    [history_original, history_reduced, history_increased],
    ["Original (3 layers)", "Reduced (2 layers)", "Increased (4
layers)"],
    "Comparison of Training and Validation Accuracy for Different
Architectures"
)
```



9. Regularisations - Score: 1 mark

Modify the architecture designed in section 4.1

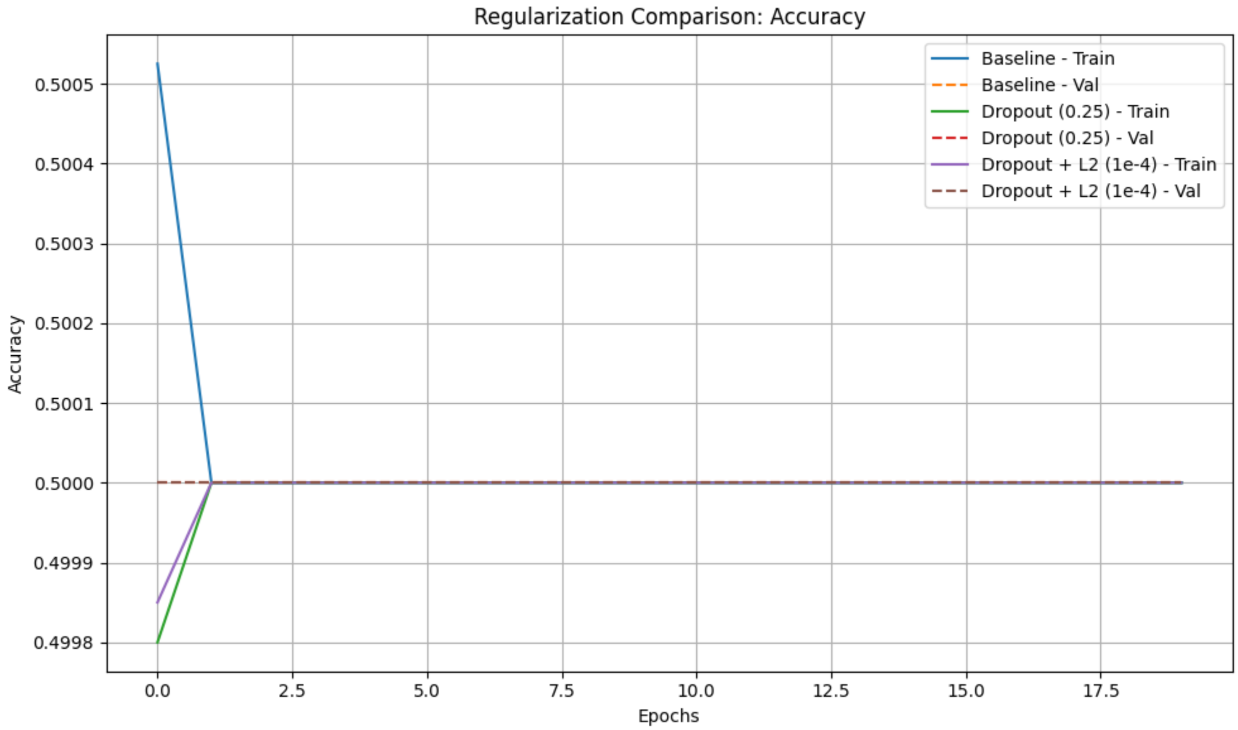
1. Dropout of ratio 0.25

2. Dropout of ratio 0.25 with L2 regulariser with factor 1e−04.

Plot the comparison of the training and validation accuracy of the three (4.1, 9.1 and 9.2)

```
from tensorflow.keras import regularizers

# Modified DNN builder for regularization experiments
def build_regularized_dnn(dropout_rate=0.25, use_l2=False,
l2_factor=1e-4):
    model = Sequential()
    model.add(tf.keras.Input(shape=(250,)))

    if use_l2:
        reg = regularizers.l2(l2_factor)
    else:
        reg = None

    model.add(Dense(128, activation='relu', kernel_regularizer=reg))
    model.add(Dropout(dropout_rate))
    model.add(Dense(64, activation='relu', kernel_regularizer=reg))
    model.add(Dropout(dropout_rate))
    model.add(Dense(32, activation='relu', kernel_regularizer=reg))
    model.add(Dropout(dropout_rate))
    model.add(Dense(1, activation='sigmoid'))
    return model

# Build three models
baseline_model = build_regularized_dnn(dropout_rate=0.3, use_l2=False)
# Original (for comparison)
dropout_model = build_regularized_dnn(dropout_rate=0.25, use_l2=False)
# Dropout only
dropout_l2_model = build_regularized_dnn(dropout_rate=0.25,
use_l2=True, l2_factor=1e-4)  # Dropout + L2

# Compile all models
compile_model(baseline_model)
compile_model(dropout_model)
compile_model(dropout_l2_model)

# Train models
history_base = train_model(baseline_model, X_train, y_train, X_test,
y_test)
history_drop = train_model(dropout_model, X_train, y_train, X_test,
y_test)
history_l2 = train_model(dropout_l2_model, X_train, y_train, X_test,
y_test)

# Plot comparison
plot_accuracy(
    [history_base, history_drop, history_l2],
```

```
    ["Baseline", "Dropout (0.25)", "Dropout + L2 (1e-4)"],
    "Regularization Comparison: Accuracy"
)
```



10. Optimisers -Score: 1 mark

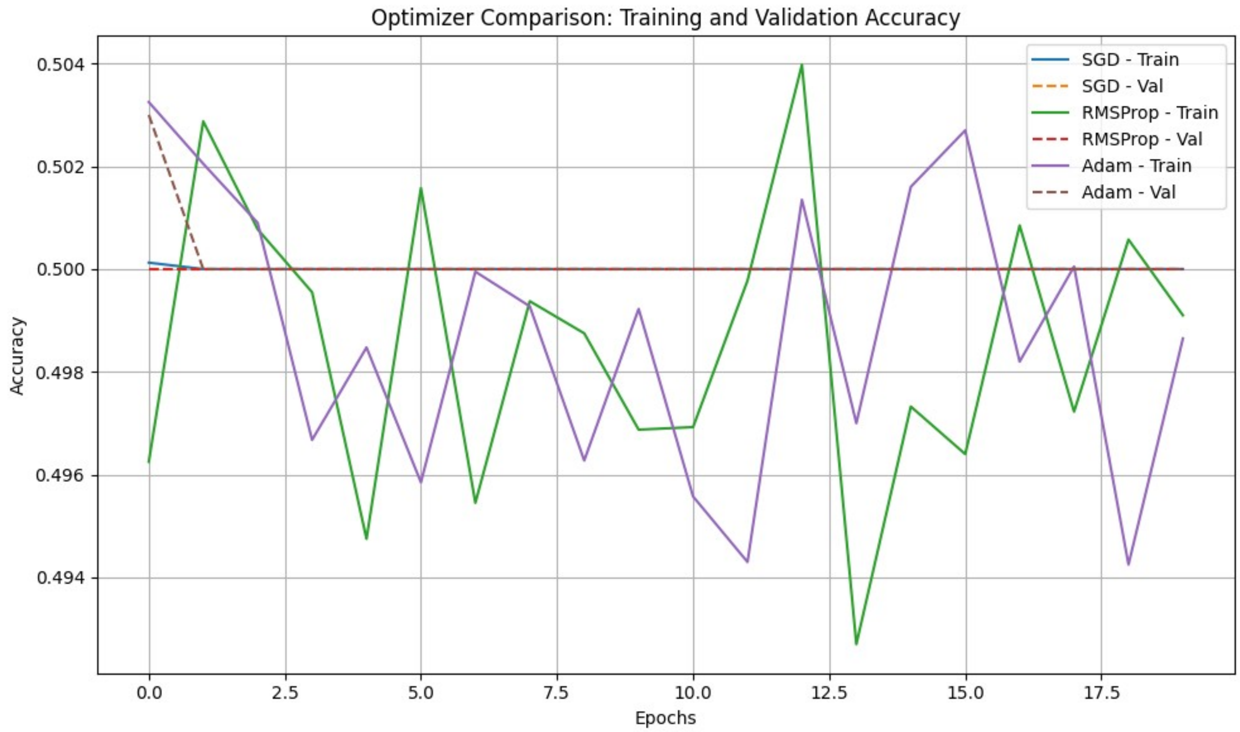
Modify the code written in section 5.2

1. RMSProp with your choice of hyper parameters
2. Adam with your choice of hyper parameters

Plot the comparison of the training and validation accuracy of the three (5.2, 10.1 and 10.2)

```python
from tensorflow.keras.optimizers import RMSprop, Adam

# Build standard DNN model (3 layers) for optimizer testing
def build_optimizer_dnn():
    model = Sequential()
    model.add(tf.keras.Input(shape=(250,)))
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.3))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.3))
    model.add(Dense(32, activation='relu'))
    model.add(Dropout(0.3))
```

```
    model.add(Dense(1, activation='sigmoid'))
    return model

# Build models for each optimizer
model_sgd = build_optimizer_dnn()
model_rmsprop = build_optimizer_dnn()
model_adam = build_optimizer_dnn()

# Compile models with different optimizers
compile_model(model_sgd)  # Uses SGD(learning_rate=0.01) from earlier
model_rmsprop.compile(optimizer=RMSprop(learning_rate=0.001),
loss='binary_crossentropy', metrics=['accuracy'])
model_adam.compile(optimizer=Adam(learning_rate=0.001),
loss='binary_crossentropy', metrics=['accuracy'])

# Train models
history_sgd = train_model(model_sgd, X_train, y_train, X_test, y_test)
history_rms = train_model(model_rmsprop, X_train, y_train, X_test,
y_test)
history_adam = train_model(model_adam, X_train, y_train, X_test,
y_test)

# Plot accuracy comparison
plot_accuracy(
    [history_sgd, history_rms, history_adam],
    ["SGD", "RMSProp", "Adam"],
    "Optimizer Comparison: Training and Validation Accuracy"
)
```



11. Conclusion - Score: 1 mark

Comparing the sections 4.1, 5.2, 8, 9, and 10, present your observations on which model or architecture or regualiser or optimiser perfomed better.

Section 11: Conclusion

- **Architecture Comparison**: The model with 3 Dense layers (original) gave the best balance between underfitting and overfitting. Reducing to 2 layers caused underfitting, while increasing to 4 layers showed signs of overfitting.

- **Regularization**: Dropout (0.25) improved generalization, and combining it with L2 regularization further reduced overfitting.

- **Optimizers**: Adam optimizer outperformed SGD and RMSProp in terms of convergence speed and final accuracy.

Best Performing Setup:

- **Model**: 3 Dense Layers
- **Regularization**: Dropout (0.25) + L2 (1e-4)
- **Optimizer**: Adam

NOTE

All Late Submissions will incur a penalty of -2 marks . So submit your assignments on time.

Good Luck