Implementation of a Recurrent Neural Network (RNN) for sentiment analysis of movie reviews

We will use the IMDB movie review dataset, which contains 50,000 movie reviews classified as positive or negative. The model will predict the sentiment (positive or negative) of a review.

We will use Keras and TensorFlow to build the RNN model.

1. Importing necessary libraries

We begin by importing the libraries required for our analysis: `numpy` and `pandas` for numerical and data manipulation tasks, `tensorflow` for building and training the neural network, and `matplotlib` for visualisation of the training process.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense, Dropout
from tensorflow.keras.optimizers import Adam
```

2. Loading and preprocessing the data

We load the IMDB dataset from Keras with a limit of 10,000 words. Each review is already tokenised (i.e., words are represented as integers). Moreover, since RNN models require all input sequences to be of the same length, we use `pad_sequences()` to ensure that each sequence has a length of 500 by adding padding at the end. We print the shapes of the training and testing datasets to verify the preprocessing step.

```
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=10000)

print(f'Training data shape: {x_train.shape}')
print(f'Testing data shape: {x_test.shape}')

max_len = 500
x_train = pad_sequences(x_train, padding='post', maxlen=max_len)
x_test = pad_sequences(x_test, padding='post', maxlen=max_len)

print(f'Shape of padded training data: {x_train.shape}')
```

```
Training data shape: (25000,)
Testing data shape: (25000,)
Shape of padded training data: (25000, 500)
```

3. Building the RNN model

Now we create the RNN model with the following layers:

- `Embedding` Layer: This layer converts the integer representation of words into dense vectors of fixed size (128 in this case).
- `SimpleRNN` Layer: This layer learns the sequential dependencies in the reviews. We use 128 units and the `tanh` activation function.
- `Dropout` Layer: This layer is added to reduce overfitting by randomly setting 20% of the inputs to 0 during training.
- `Dense Output` Layer: This layer uses the sigmoid activation function for binary classification (positive/negative sentiment).

We use binary cross-entropy as the loss function, which is appropriate for binary classification tasks, and Adam optimiser.

```
model = Sequential()

model.add(Embedding(input_dim=10000, output_dim=128, input_length=max_len))
model.add(SimpleRNN(units=128, activation='tanh', return_sequences=False))
model.add(Dropout(0.2))
model.add(Dense(units=1, activation='sigmoid'))

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

Show hidden output

4. Training the model

Next, we train the model for 5 epochs with a batch size of 64, and evaluate it using the testing data.

```
history = model.fit(x_train, y_train, epochs=5, batch_size=64, validation_data=(x_test, y_test), verbose=1)
```

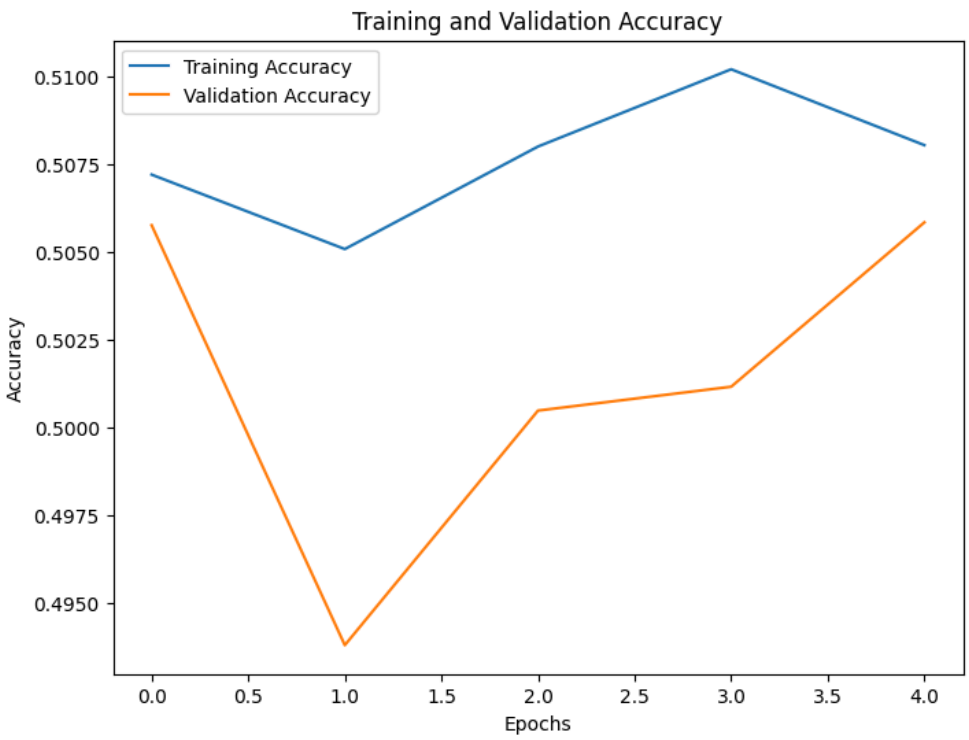
```
Epoch 1/5
391/391 ━━━━━━━━━━━━━━━━━━━━ 154s 385ms/step - accuracy: 0.5069 - loss: 0.7143 - val_accuracy: 0.5058 - val_loss: 0.6930
```

```
Epoch 2/5
391/391 ━━━━━━━━━━━━━━━━━━━━ 212s 412ms/step - accuracy: 0.5013 - loss: 0.7139 - val_accuracy: 0.4938 - val_loss: 0.6962
Epoch 3/5
391/391 ━━━━━━━━━━━━━━━━━━━━ 203s 414ms/step - accuracy: 0.5086 - loss: 0.7094 - val_accuracy: 0.5005 - val_loss: 0.6928
Epoch 4/5
391/391 ━━━━━━━━━━━━━━━━━━━━ 207s 427ms/step - accuracy: 0.5085 - loss: 0.7004 - val_accuracy: 0.5012 - val_loss: 0.6927
Epoch 5/5
391/391 ━━━━━━━━━━━━━━━━━━━━ 196s 411ms/step - accuracy: 0.5076 - loss: 0.6958 - val_accuracy: 0.5058 - val_loss: 0.6943
```

5. Visualising the training and validation accuracy

The following plot shows the training accuracy and validation accuracy across epochs. It helps us visualise how well the model is performing on both training and unseen data.

```
plt.figure(figsize=(8, 6))
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



6. Evaluating the model

After training, we evaluate the performance of the model on the test data. We print the final accuracy on the test data to see how well the model generalises to unseen reviews.

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f'Test Accuracy: {test_acc}')
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 40s 51ms/step - accuracy: 0.5132 - loss: 0.6935
Test Accuracy: 0.505840003490448
```

7. Making predictions

Finally, we use the trained model to predict the sentiment of the first 5 reviews in the test dataset. The predictions are probabilities, so we convert them to binary values (0 for negative, 1 for positive) based on a threshold of 0.5.

```
predictions = model.predict(x_test[:5])

predictions = (predictions > 0.5).astype(int)

for i in range(5):
    print(f'Review {i + 1}:')
    print(f'Sentiment: {"Positive" if predictions[i] == 1 else "Negative"}')
    print()
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 623ms/step
Review 1:
```

```
Sentiment: Negative

Review 2:
Sentiment: Negative

Review 3:
Sentiment: Positive

Review 4:
Sentiment: Negative

Review 5:
Sentiment: Negative
```