Assignment 3 - RNN_IMDB-Q

July 15, 2024

0.1 Using RNNs to classify sentiment on IMDB data

In this assignment,you will train three types of RNNs: “vanilla” RNN, LSTM and GRU to predict the sentiment on IMDB reviews.

Keras provides a convenient interface to load the data and immediately encode the words into integers (based on the most common words). This will save you a lot of the drudgery that is usually involved when working with raw text.

The IMDB is data consists of 25000 training sequences and 25000 test sequences. The outcome is binary (positive/negative) and both outcomes are equally represented in both the training and the test set.

Walk through the followinng steps to prepare the data and the building of an RNN model.

1- Use the `imdb.load_data()` to load in the data

2- Specify the maximum length of a sequence to 30 words and the pick the 2000 most common words.

[30]:
```
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Set maximum length parameters
max_words = 2000
max_len = 30

# Load the data
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_words)
```

3- Check that the number of sequences in train and test datasets are equal (default split):

Expected output: - `x_train = 25000 train sequences`

- `x_test = 25000 test sequences`

[31]:
```
# Print the shape of the data to see the number of sequences in train and test␣
↪datasets
print(f'Training data shape: {x_train.shape}')
print(f'Test data shape: {x_test.shape}')
```

```
Training data shape: (25000,)
Test data shape: (25000,)
```

4- Pad (or truncate) the sequences so that they are of the maximum length

[37]:
```
# Set parameters
max_len = 30  # Maximum length of a sequence

# Pad or truncate the sequences to the specified maximum length
x_train_padded = pad_sequences(x_train, maxlen=max_len, padding='post',␣
 ↪truncating='post')
x_test_padded = pad_sequences(x_test, maxlen=max_len, padding='post',␣
 ↪truncating='post')
```

5- After padding or truncating, check the dimensionality of x_train and x_test.

Expected output: - `x_train shape: (25000, 30)` - `x_test shape: (25000, 30)`

[33]:
```
# Print the shape of the padded data to check the dimensionality of x_train and␣
 ↪x_test
print(f'Padded training data shape: {x_train_padded.shape}')
print(f'Padded test data shape: {x_test_padded.shape}')
```

```
Padded training data shape: (25000, 30)
Padded test data shape: (25000, 30)
```

[ ]:
```

```

0.2 Keras layers for (Vanilla) RNNs

In this step, you will not use pre-trained word vectors, Instead you will learn an embedding as part of the the Vanilla) RNNs network Neural Network.

In the Keras API documentation, the Embedding Layer and the SimpleRNN Layer have the following syntax:

0.2.1 Embedding Layer

`keras.layers.embeddings.Embedding(input_dim, output_dim, embeddings_initializer='uniform', embeddings_regularizer=None, activity_regularizer=None, embeddings_constraint=None, mask_zero=False, input_length=None)`

- This layer maps each integer into a distinct (dense) word vector of length `output_dim`.
- Can think of this as learning a word vector embedding "on the fly" rather than using an existing mapping (like GloVe)
- The `input_dim` should be the size of the vocabulary.
- The `input_length` specifies the length of the sequences that the network expects.

0.2.2 SimpleRNN Layer

`keras.layers.recurrent.SimpleRNN(units, activation='tanh', use_bias=True, kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal', bias_initializer='zeros', kernel_regularizer=None, recurrent_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, recurrent_constraint=None, bias_constraint=None, dropout=0.0, recurrent_dropout=0.0)`

- This is the basic RNN, where the output is also fed back as the "hidden state" to the next iteration.
- The parameter `units` gives the dimensionality of the output (and therefore the hidden state). Note that typically there will be another layer after the RNN mapping the (RNN) output to the network output. So we should think of this value as the desired dimensionality of the hidden state and not necessarily the desired output of the network.
- Recall that there are two sets of weights, one for the "recurrent" phase and the other for the "kernel" phase. These can be configured separately in terms of their initialization, regularization, etc.

6- Build the RNN with three layers: - The SimpleRNN layer with 5 neurons and initialize its kernel with stddev=0.001

- The Embedding layer and initialize it by setting the word embedding dimension to 50. This means that this layer takes each integer in the sequence and embeds it in a 50-dimensional vector.

- The output layer has the sigmoid activation function.

[34]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense
from tensorflow.keras.initializers import RandomNormal

# Set parameters
vocab_size = 2000  # Size of the vocabulary
embedding_dim = 50  # the word embedding dimension
max_len = 30  # Maximum length of a sequence

# Build the Vanilla RNN model
vanilla_rnn_model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=embedding_dim,␣
 ↪input_length=max_len),
    SimpleRNN(units=5, kernel_initializer=RandomNormal(stddev=0.001)),
    Dense(1, activation='sigmoid')
])

# Compile the model
vanilla_rnn_model.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])

# Train the model
```

```
history_vanilla_rnn = vanilla_rnn_model.fit(x_train_padded, y_train, epochs=5,␣
↪batch_size=64, validation_split=0.2)

# Evaluate the model
#vanilla_rnn_test_loss, vanilla_rnn_test_acc = vanilla_rnn_model.
↪evaluate(x_test_padded, y_test)
#print(f'Vanilla RNN Test Accuracy: {vanilla_rnn_test_acc:.4f}')
```

```
Epoch 1/5
313/313              2s 4ms/step -
accuracy: 0.6124 - loss: 0.6476 - val_accuracy: 0.7182 - val_loss: 0.5709
Epoch 2/5
313/313              1s 3ms/step -
accuracy: 0.7455 - loss: 0.5368 - val_accuracy: 0.7142 - val_loss: 0.5646
Epoch 3/5
313/313              1s 3ms/step -
accuracy: 0.7622 - loss: 0.5086 - val_accuracy: 0.7104 - val_loss: 0.5628
Epoch 4/5
313/313              1s 3ms/step -
accuracy: 0.7736 - loss: 0.4915 - val_accuracy: 0.7204 - val_loss: 0.5613
Epoch 5/5
313/313              1s 3ms/step -
accuracy: 0.7886 - loss: 0.4708 - val_accuracy: 0.7082 - val_loss: 0.5811
```

7- How many parameters have the embedding layer?

[36]:
```
# Print the model summary to see the number of parameters
vanilla_rnn_model.summary()
```

```
Model: "sequential_18"


 Layer (type)                       Output Shape                  Param #

 embedding_18 (Embedding)           (None, 30, 50)                100,000

 simple_rnn_16 (SimpleRNN)          (None, 5)                         280

 dense_18 (Dense)                   (None, 1)                           6


Total params: 300,860 (1.15 MB)

Trainable params: 100,286 (391.74 KB)

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

```
 Optimizer params: 200,574 (783.50 KB)
```

Embedding layer has 100,000 parameters. *2000 X 50

8- Train the network with the RMSprop with learning rate of .0001 and epochs=10.

[38]:
```python
# Import necessary libraries
from tensorflow.keras.optimizers import RMSprop
```

[39]:
```python
# Set parameters
vocab_size = 2000  # Size of the vocabulary
embedding_dim = 50  # Dimension of the embedding vector
max_len = 30  # Maximum length of a sequence

# Build the Vanilla RNN model
vanilla_rnn_model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=embedding_dim,␣
 ↪input_length=max_len),
    SimpleRNN(units=5, kernel_initializer=RandomNormal(stddev=0.001)),
    Dense(1, activation='sigmoid')
])

# Set the RMSprop optimizer with a learning rate of 0.0001
optimizer = RMSprop(learning_rate=0.0001)

# Compile the model with the RMSprop optimizer
vanilla_rnn_model.compile(optimizer=optimizer, loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])

# Train the model
history_vanilla_rnn = vanilla_rnn_model.fit(x_train_padded, y_train, epochs=10,␣
 ↪batch_size=64, validation_split=0.2)
```

```
Epoch 1/10
313/313             2s 4ms/step -
accuracy: 0.5129 - loss: 0.6930 - val_accuracy: 0.5258 - val_loss: 0.6925
Epoch 2/10
313/313             1s 3ms/step -
accuracy: 0.5414 - loss: 0.6911 - val_accuracy: 0.5486 - val_loss: 0.6852
Epoch 3/10
313/313             1s 3ms/step -
accuracy: 0.5910 - loss: 0.6738 - val_accuracy: 0.5974 - val_loss: 0.6611
Epoch 4/10
313/313             1s 3ms/step -
accuracy: 0.6309 - loss: 0.6382 - val_accuracy: 0.6324 - val_loss: 0.6401
Epoch 5/10
```

```
313/313              1s 3ms/step -
accuracy: 0.6739 - loss: 0.5993 - val_accuracy: 0.6354 - val_loss: 0.6358
Epoch 6/10
313/313              1s 3ms/step -
accuracy: 0.6939 - loss: 0.5823 - val_accuracy: 0.6446 - val_loss: 0.6287
Epoch 7/10
313/313              1s 3ms/step -
accuracy: 0.6994 - loss: 0.5737 - val_accuracy: 0.6478 - val_loss: 0.6304
Epoch 8/10
313/313              1s 3ms/step -
accuracy: 0.7082 - loss: 0.5593 - val_accuracy: 0.6404 - val_loss: 0.6377
Epoch 9/10
313/313              1s 3ms/step -
accuracy: 0.7174 - loss: 0.5546 - val_accuracy: 0.6476 - val_loss: 0.6300
Epoch 10/10
313/313              1s 3ms/step -
accuracy: 0.7198 - loss: 0.5514 - val_accuracy: 0.6508 - val_loss: 0.6344
```

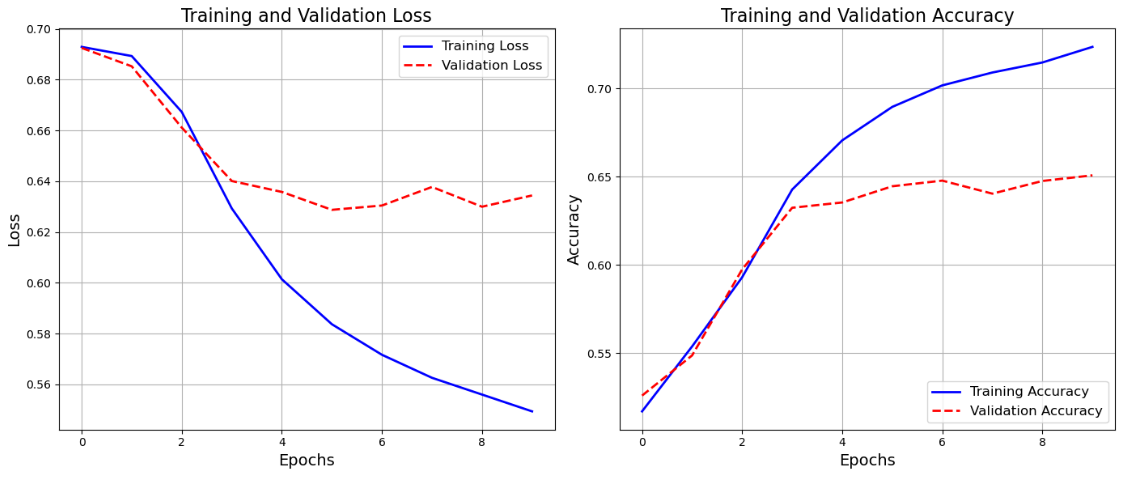
9- PLot the loss and accuracy metrics during the training and interpret the result.

[42]:
```python
import matplotlib.pyplot as plt

# Plot the training and validation loss and accuracy
plt.figure(figsize=(14, 6))

# Plot the training and validation loss
plt.subplot(1, 2, 1)
plt.plot(history_vanilla_rnn.history['loss'], label='Training Loss',␣
 ↪color='blue', linestyle='-', linewidth=2)
plt.plot(history_vanilla_rnn.history['val_loss'], label='Validation Loss',␣
 ↪color='red', linestyle='--', linewidth=2)
plt.title('Training and Validation Loss', fontsize=16)
plt.xlabel('Epochs', fontsize=14)
plt.ylabel('Loss', fontsize=14)
plt.legend(loc='upper right', fontsize=12)
plt.grid(True)

# Plot the training and validation accuracy
plt.subplot(1, 2, 2)
plt.plot(history_vanilla_rnn.history['accuracy'], label='Training Accuracy',␣
 ↪color='blue', linestyle='-', linewidth=2)
plt.plot(history_vanilla_rnn.history['val_accuracy'], label='Validation␣
 ↪Accuracy', color='red', linestyle='--', linewidth=2)
plt.title('Training and Validation Accuracy', fontsize=16)
plt.xlabel('Epochs', fontsize=14)
plt.ylabel('Accuracy', fontsize=14)
plt.legend(loc='lower right', fontsize=12)
plt.grid(True)
plt.tight_layout()
plt.show()
```



Conclusion:

- Training and Validation Loss: the training loss decreases but the validation loss slightly flattens and starts to increase after a certain number of epochs. This indicates overfitting.
- Training and Validation Accuracy: the training accuracy increases but the validation accuracy stagnates suggesting overfitting.

** Therefore, the model appears to be overfitted. The model performs well on training data but not on unseen data.

10- Check the accuracy and the loss of your models on the test dataset.

[16]:
```
# Evaluate the model on the test dataset
vanilla_rnn_test_loss, vanilla_rnn_test_acc = vanilla_rnn_model.
 ↪evaluate(x_test_padded, y_test)
print(f'Vanilla RNN Test Accuracy: {vanilla_rnn_test_acc:.4f}')
```

```
782/782              0s 570us/step -
accuracy: 0.5658 - loss: 0.6894
Vanilla RNN Test Accuracy: 0.5672
```

Conclusion:

- The accuracy of 0.5672 means that the model correctly predicts the sentiment of IMDB reviews approximately 56.72% of the time. As this is a binary classification problem, an accuracy slightly above 50% indicates that the model is only slightly better than random guessing.
- The loss value of 0.6894 is relatively high for a binary classification problem using binary cross-entropy loss.

- A high loss value, combined with low accuracy, suggests that the model is not learning the underlying patterns in the data well. The model needs improvement.

0.3 Tuning The Vanilla RNN Network

11- Prepare the data to use sequences of length 80 rather than length 30 and retrain your model. Did it improve the performance?

12- Try different values of the maximum length of a sequence ("max_features"). Can you improve the performance?

13- Try smaller and larger sizes of the RNN hidden dimension. How does it affect the model performance? How does it affect the run time?

[44]:
```
# Set new sequence length
max_len_80 = 80

# Pad or truncate the sequences to the new maximum length
x_train_padded_80 = pad_sequences(x_train, maxlen=max_len_80, padding='post',␣
 ↪truncating='post')
x_test_padded_80 = pad_sequences(x_test, maxlen=max_len_80, padding='post',␣
 ↪truncating='post')

# Print the shape of the padded data
print(f'Padded training data shape (max_len=80): {x_train_padded_80.shape}')
print(f'Padded test data shape (max_len=80): {x_test_padded_80.shape}')
```

```
Padded training data shape (max_len=80): (25000, 80)
Padded test data shape (max_len=80): (25000, 80)
```

[45]:
```
from tensorflow.keras.optimizers import RMSprop

# Build the Vanilla RNN model with sequence length 80
vanilla_rnn_model_80 = Sequential([
    Embedding(input_dim=vocab_size, output_dim=embedding_dim,␣
 ↪input_length=max_len_80),
    SimpleRNN(units=5, kernel_initializer=RandomNormal(stddev=0.001)),
    Dense(1, activation='sigmoid')
])

# Recreate the RMSprop optimizer with the specified learning rate
optimizer_80 = RMSprop(learning_rate=0.0001)

# Compile the model with RMSprop optimizer
vanilla_rnn_model_80.compile(optimizer=optimizer_80,␣
 ↪loss='binary_crossentropy', metrics=['accuracy'])

# Train the model
```

```python
history_vanilla_rnn_80 = vanilla_rnn_model_80.fit(x_train_padded_80, y_train,␣
↪epochs=10, batch_size=64, validation_split=0.2)

# Evaluate the model
vanilla_rnn_test_loss_80, vanilla_rnn_test_acc_80 = vanilla_rnn_model_80.
↪evaluate(x_test_padded_80, y_test)
print(f'Vanilla RNN (max_len=80) Test Loss: {vanilla_rnn_test_loss_80:.4f}')
print(f'Vanilla RNN (max_len=80) Test Accuracy: {vanilla_rnn_test_acc_80:.4f}')
```

```
Epoch 1/10
313/313              3s 7ms/step -
accuracy: 0.5037 - loss: 0.6931 - val_accuracy: 0.5120 - val_loss: 0.6928
Epoch 2/10
313/313              2s 7ms/step -
accuracy: 0.5319 - loss: 0.6915 - val_accuracy: 0.5292 - val_loss: 0.6913
Epoch 3/10
313/313              2s 7ms/step -
accuracy: 0.5563 - loss: 0.6839 - val_accuracy: 0.5348 - val_loss: 0.6908
Epoch 4/10
313/313              2s 7ms/step -
accuracy: 0.5944 - loss: 0.6717 - val_accuracy: 0.5544 - val_loss: 0.6895
Epoch 5/10
313/313              2s 7ms/step -
accuracy: 0.6173 - loss: 0.6593 - val_accuracy: 0.5560 - val_loss: 0.6917
Epoch 6/10
313/313              2s 7ms/step -
accuracy: 0.6362 - loss: 0.6488 - val_accuracy: 0.5562 - val_loss: 0.6916
Epoch 7/10
313/313              2s 7ms/step -
accuracy: 0.6496 - loss: 0.6391 - val_accuracy: 0.5536 - val_loss: 0.6965
Epoch 8/10
313/313              2s 7ms/step -
accuracy: 0.6678 - loss: 0.6285 - val_accuracy: 0.5496 - val_loss: 0.6999
Epoch 9/10
313/313              2s 7ms/step -
accuracy: 0.6721 - loss: 0.6235 - val_accuracy: 0.5576 - val_loss: 0.6988
Epoch 10/10
313/313              2s 7ms/step -
accuracy: 0.6770 - loss: 0.6170 - val_accuracy: 0.5614 - val_loss: 0.6999
782/782              1s 1ms/step -
accuracy: 0.5600 - loss: 0.6993
Vanilla RNN (max_len=80) Test Loss: 0.6965
Vanilla RNN (max_len=80) Test Accuracy: 0.5636
```

Conclusion: - Using sequences of length 80 rather than length 30 and did not improve the model's performance.

Step 12: Experiment with Different Sequence Lengths ("max_features")

[47]:
```python
from tensorflow.keras.optimizers import RMSprop

# Define sequence lengths to try
sequence_lengths = [50, 100, 150, 200]

results = []

for max_len in sequence_lengths:
    # Pad or truncate the sequences to the new maximum length
    x_train_padded = pad_sequences(x_train, maxlen=max_len, padding='post',␣
 ↪truncating='post')
    x_test_padded = pad_sequences(x_test, maxlen=max_len, padding='post',␣
 ↪truncating='post')

    # Build the Vanilla RNN model with the new sequence length
    vanilla_rnn_model = Sequential([
        Embedding(input_dim=vocab_size, output_dim=embedding_dim,␣
 ↪input_length=max_len),
        SimpleRNN(units=5, kernel_initializer=RandomNormal(stddev=0.001)),
        Dense(1, activation='sigmoid')
    ])

    # Recreate the RMSprop optimizer with the specified learning rate
    optimizer = RMSprop(learning_rate=0.0001)

    # Compile the model with RMSprop optimizer
    vanilla_rnn_model.compile(optimizer=optimizer, loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])

    # Train the model
    history_vanilla_rnn = vanilla_rnn_model.fit(x_train_padded, y_train,␣
 ↪epochs=10, batch_size=64, validation_split=0.2, verbose=0)

    # Evaluate the model
    test_loss, test_acc = vanilla_rnn_model.evaluate(x_test_padded, y_test,␣
 ↪verbose=0)
    results.append((max_len, test_loss, test_acc))

# Print the results
for max_len, test_loss, test_acc in results:
    print(f'Sequence Length {max_len} -> Test Loss: {test_loss:.4f}, Test␣
 ↪Accuracy: {test_acc:.4f}')
```

```
Sequence Length 50 -> Test Loss: 0.7015, Test Accuracy: 0.5268
Sequence Length 100 -> Test Loss: 0.7118, Test Accuracy: 0.5363
Sequence Length 150 -> Test Loss: 0.6588, Test Accuracy: 0.6382
Sequence Length 200 -> Test Loss: 0.6456, Test Accuracy: 0.6632
```

Conclusion:

- Increasing the Sequence Lengths using RMSprop optimizer, increased the model performance.

Step 13: Experiment with Different RNN Hidden Dimensions

[28]:
```python
from tensorflow.keras.optimizers import RMSprop

# Define hidden dimensions to experiment with
hidden_dimensions = [2, 10, 20]

results_hidden_dim = []

for hidden_dim in hidden_dimensions:
    # Build the Vanilla RNN model with the new hidden dimension
    vanilla_rnn_model = Sequential([
        Embedding(input_dim=vocab_size, output_dim=embedding_dim,␣
 ↪input_length=max_len),
        SimpleRNN(units=hidden_dim, kernel_initializer=RandomNormal(stddev=0.
 ↪001)),
        Dense(1, activation='sigmoid')
    ])

    # Recreate the RMSprop optimizer with the specified learning rate
    optimizer = RMSprop(learning_rate=0.0001)

    # Compile the model with RMSprop optimizer
    vanilla_rnn_model.compile(optimizer=optimizer, loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])

    # Train the model
    history_vanilla_rnn = vanilla_rnn_model.fit(x_train_padded, y_train,␣
 ↪epochs=10, batch_size=64, validation_split=0.2, verbose=0)

    # Evaluate the model
    test_loss, test_acc = vanilla_rnn_model.evaluate(x_test_padded, y_test,␣
 ↪verbose=0)
    results_hidden_dim.append((hidden_dim, test_loss, test_acc))

# Print the results
for hidden_dim, test_loss, test_acc in results_hidden_dim:
    print(f'Hidden Dimension {hidden_dim} -> Test Loss: {test_loss:.4f}, Test␣
 ↪Accuracy: {test_acc:.4f}')
```

```
Hidden Dimension 2 -> Test Loss: 0.6631, Test Accuracy: 0.6385
Hidden Dimension 10 -> Test Loss: 0.5874, Test Accuracy: 0.7156
Hidden Dimension 20 -> Test Loss: 0.7106, Test Accuracy: 0.5489
```

Conclusion: - The model with 10 hidden units in the RNN layer performs the best, achieving the highest test accuracy and the lowest test loss. This configuration strikes the right balance between model complexity and the ability to generalize to new, unseen data. - The model with 2 hidden units is too simple and underfits the data, resulting in moderate accuracy. - The model with 20 hidden units is too complex, leading to overfitting and poor generalization performance.

0.4 Train LSTM and GRU networks

14- Build LSTM and GRU networks and compare their performance (accuracy and execution time) with the SimpleRNN. What is your conclusion?

[48]:
```
import time
from tensorflow.keras.layers import Embedding, SimpleRNN, LSTM, GRU, Dense

# Set parameters
vocab_size = 2000
embedding_dim = 50
max_len = 30

# Load the data
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=vocab_size)

# Pad sequences to ensure equal length
x_train_padded = pad_sequences(x_train, maxlen=max_len, padding='post',␣
 ↪truncating='post')
x_test_padded = pad_sequences(x_test, maxlen=max_len, padding='post',␣
 ↪truncating='post')

# Define a function to build, compile, train, and evaluate a model
def build_and_evaluate_rnn(model_type, units=5, max_len=30, epochs=10,␣
 ↪batch_size=64):
    # Build the model
    model = Sequential([
        Embedding(input_dim=vocab_size, output_dim=embedding_dim,␣
 ↪input_length=max_len)
    ])

    if model_type == 'SimpleRNN':
        model.add(SimpleRNN(units=units,␣
 ↪kernel_initializer=RandomNormal(stddev=0.001)))
    elif model_type == 'LSTM':
        model.add(LSTM(units=units, kernel_initializer=RandomNormal(stddev=0.
 ↪001)))
    elif model_type == 'GRU':
        model.add(GRU(units=units, kernel_initializer=RandomNormal(stddev=0.
 ↪001)))

    model.add(Dense(1, activation='sigmoid'))
    # Recreate the RMSprop optimizer
    optimizer = RMSprop(learning_rate=0.0001)

    # Compile the model
    model.compile(optimizer=optimizer, loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])

    # Train the model and measure the time taken
    start_time = time.time()
    history = model.fit(x_train_padded, y_train, epochs=10,␣
 ↪batch_size=batch_size, validation_split=0.2, verbose=0)
    training_time = time.time() - start_time

    # Evaluate the model
    test_loss, test_acc = model.evaluate(x_test_padded, y_test, verbose=0)

    return test_loss, test_acc, training_time

# Evaluate SimpleRNN
simple_rnn_loss, simple_rnn_acc, simple_rnn_time =␣
 ↪build_and_evaluate_rnn('SimpleRNN')

# Evaluate LSTM
lstm_loss, lstm_acc, lstm_time = build_and_evaluate_rnn('LSTM')

# Evaluate GRU
gru_loss, gru_acc, gru_time = build_and_evaluate_rnn('GRU')

# Print the results
print(f'SimpleRNN -> Test Loss: {simple_rnn_loss:.4f}, Test Accuracy:␣
 ↪{simple_rnn_acc:.4f}, Training Time: {simple_rnn_time:.2f} seconds')
print(f'LSTM -> Test Loss: {lstm_loss:.4f}, Test Accuracy: {lstm_acc:.4f},␣
 ↪Training Time: {lstm_time:.2f} seconds')
print(f'GRU -> Test Loss: {gru_loss:.4f}, Test Accuracy: {gru_acc:.4f},␣
 ↪Training Time: {gru_time:.2f} seconds')
```

```
SimpleRNN -> Test Loss: 0.5907, Test Accuracy: 0.6869, Training Time: 9.90
seconds
LSTM -> Test Loss: 0.6873, Test Accuracy: 0.5490, Training Time: 13.39 seconds
GRU -> Test Loss: 0.6916, Test Accuracy: 0.5272, Training Time: 15.92 seconds
```

Conclusion:

- SimpleRNN outperforms both LSTM and GRU in terms of test accuracy and test loss. It also has the shortest training time, making it the most efficient and effective model for this particular task.
- LSTM and GRU both perform worse than SimpleRNN, with GRU being the least effective

in terms of both accuracy and training time.

[ ]:

```

```