```
# Importing the IMDB dataset from TensorFlow:
from tensorflow.keras.datasets import imdb

# Loading the dataset containing the 10,000 most common words:
(A_train, B_train), (A_test, B_test) = imdb.load_data(num_words=10000)

# Displaying the first training sample:
A_train[0]
```

```
 104,
 88,
 4,
 381,
 15,
 297,
 98,
 32,
 2071,
 56,
 26,
 141,
 6,
 194,
 7486,
 18,
 4,
 226,
 22,
 21,
 134,
 476,
 26,
 480,
 5,
 144,
 30,
 5535,
 18,
 51,
 36,
 28,
 224,
 92,
 25,
 104,
 4,
 226,
 65,
 16,
 38,
 1334,
 88,
 12,
 16,
 283,
 5,
 16,
 4472,
 113,
 103,
 32,
 15,
 16,
 5345,
 19,
 178,
 32]
```

```
# Accessing the first output value in the training data:
B_train[0]
```

```
np.int64(1)
```

```
# Identifying the highest word index in the dataset:
max([max(seq) for seq in A_train])
```

```
9999
```

```
# Transforming numerical sequences back into words:
word_to_id = imdb.get_word_index()

# Creating a lookup table that converts indices into words:
id_to_word = {index: word for word, index in word_to_id.items()}

# Converting index values of the first review into readable words:
review_text = " ".join([id_to_word.get(i - 3, "?") for i in A_train[0]])
```

```
# Data preprocessing:
# Transforming integer sequences into multi-hot vectors:
import numpy as np

def multi_hot_encode(sequences, vocab_size=10000):
    encoded_matrix = np.zeros((len(sequences), vocab_size))
    for idx, seq in enumerate(sequences):
        for token in seq:
            encoded_matrix[idx, token] = 1.
    return encoded_matrix

A_train_encoded = multi_hot_encode(A_train)
A_test_encoded = multi_hot_encode(A_test)
```

```
# Showing the first training example after applying multi-hot encoding:
A_train_encoded[0]
```

```
array([0., 1., 1., ..., 0., 0., 0.])
```

```
# Now, Converting labels to float32 arrays:
B_train = np.asarray(y_train).astype("float32")
B_test = np.asarray(y_test).astype("float32")
```

```
# Building a neural network model:
# It has 3 layers, each with 64 neurons (nodes):
# We use the "tanh" activation function because it works well with both positive and negative numbers:
# L2 regularization is added to reduce overfitting
# Dropout layers are included to help the model generalize better:

from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras import regularizers

neural_net = keras.Sequential([
    layers.Dense(64, kernel_regularizer=regularizers.l2(0.005), activation="tanh"),
    layers.Dense(64, activation="tanh"),
    layers.Dropout(0.5),
    layers.Dense(64, activation="tanh"),
    layers.Dropout(0.5),
    layers.Dense(1, activation="sigmoid")
])
```

```
# Compiling the model:
neural_net.compile(optimizer="rmsprop",
                   loss="mse",
                   metrics=["accuracy"])
```

```
# Checking if our model works well using validation data and We make a validation set by splitting part of the training data:
A_val = A_train_encoded[:10000]
A_train_partial = A_train_encoded[10000:]
B_val = B_train[:10000]
B_train_partial = B_train[10000:]
```

```
# Now, Training the neural network model:
# We use 512 samples in each training batch and train for 20 rounds (epochs)
# The validation set made earlier is used to check how well the model is learning:
```

```
training_history = neural_net.fit(
    A_train_partial,
    B_train_partial,
    epochs=20,
    batch_size=512,
    validation_data=(A_val, B_val)
)
```

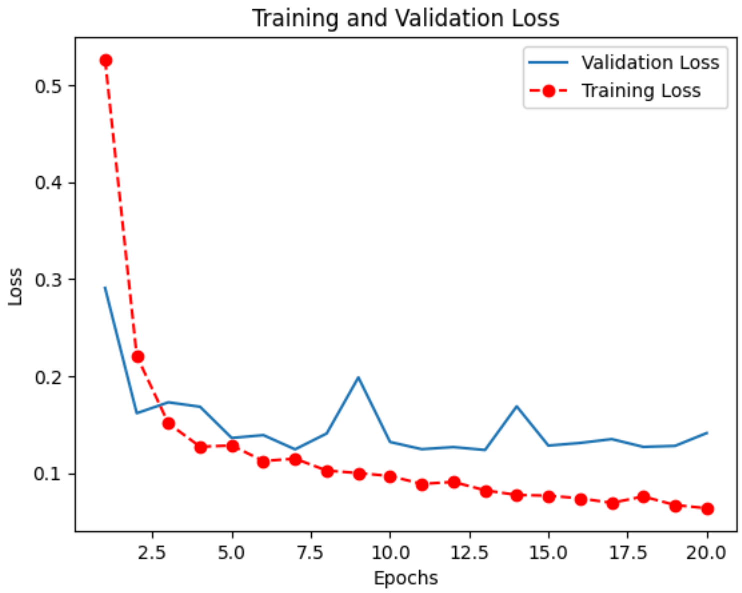
```
Epoch 1/20
30/30 ━━━━━━━━━━━━━━━━━━━━ 9s 239ms/step - accuracy: 0.6484 - loss: 0.6736 - val_accuracy: 0.8287 - val_loss: 0.2910
Epoch 2/20
30/30 ━━━━━━━━━━━━━━━━━━━━ 3s 103ms/step - accuracy: 0.8414 - loss: 0.2489 - val_accuracy: 0.8813 - val_loss: 0.1619
Epoch 3/20
30/30 ━━━━━━━━━━━━━━━━━━━━ 2s 73ms/step - accuracy: 0.8843 - loss: 0.1524 - val_accuracy: 0.8325 - val_loss: 0.1732
Epoch 4/20
30/30 ━━━━━━━━━━━━━━━━━━━━ 4s 144ms/step - accuracy: 0.8905 - loss: 0.1322 - val_accuracy: 0.8273 - val_loss: 0.1686
Epoch 5/20
30/30 ━━━━━━━━━━━━━━━━━━━━ 7s 216ms/step - accuracy: 0.8798 - loss: 0.1321 - val_accuracy: 0.8728 - val_loss: 0.1364
Epoch 6/20
30/30 ━━━━━━━━━━━━━━━━━━━━ 7s 112ms/step - accuracy: 0.9151 - loss: 0.1079 - val_accuracy: 0.8647 - val_loss: 0.1393
Epoch 7/20
30/30 ━━━━━━━━━━━━━━━━━━━━ 4s 80ms/step - accuracy: 0.8960 - loss: 0.1174 - val_accuracy: 0.8814 - val_loss: 0.1247
Epoch 8/20
30/30 ━━━━━━━━━━━━━━━━━━━━ 2s 75ms/step - accuracy: 0.9253 - loss: 0.0961 - val_accuracy: 0.8593 - val_loss: 0.1411
Epoch 9/20
30/30 ━━━━━━━━━━━━━━━━━━━━ 2s 76ms/step - accuracy: 0.9252 - loss: 0.0942 - val_accuracy: 0.7948 - val_loss: 0.1988
Epoch 10/20
30/30 ━━━━━━━━━━━━━━━━━━━━ 2s 81ms/step - accuracy: 0.9159 - loss: 0.0989 - val_accuracy: 0.8709 - val_loss: 0.1323
Epoch 11/20
30/30 ━━━━━━━━━━━━━━━━━━━━ 4s 134ms/step - accuracy: 0.9305 - loss: 0.0884 - val_accuracy: 0.8763 - val_loss: 0.1248
Epoch 12/20
30/30 ━━━━━━━━━━━━━━━━━━━━ 2s 72ms/step - accuracy: 0.9227 - loss: 0.0933 - val_accuracy: 0.8740 - val_loss: 0.1270
Epoch 13/20
30/30 ━━━━━━━━━━━━━━━━━━━━ 3s 79ms/step - accuracy: 0.9416 - loss: 0.0780 - val_accuracy: 0.8760 - val_loss: 0.1240
Epoch 14/20
30/30 ━━━━━━━━━━━━━━━━━━━━ 2s 74ms/step - accuracy: 0.9474 - loss: 0.0736 - val_accuracy: 0.8265 - val_loss: 0.1690
Epoch 15/20
30/30 ━━━━━━━━━━━━━━━━━━━━ 3s 86ms/step - accuracy: 0.9380 - loss: 0.0789 - val_accuracy: 0.8704 - val_loss: 0.1286
Epoch 16/20
30/30 ━━━━━━━━━━━━━━━━━━━━ 3s 99ms/step - accuracy: 0.9485 - loss: 0.0711 - val_accuracy: 0.8690 - val_loss: 0.1313
Epoch 17/20
30/30 ━━━━━━━━━━━━━━━━━━━━ 4s 74ms/step - accuracy: 0.9541 - loss: 0.0660 - val_accuracy: 0.8628 - val_loss: 0.1352
Epoch 18/20
30/30 ━━━━━━━━━━━━━━━━━━━━ 2s 71ms/step - accuracy: 0.9361 - loss: 0.0778 - val_accuracy: 0.8755 - val_loss: 0.1272
Epoch 19/20
30/30 ━━━━━━━━━━━━━━━━━━━━ 2s 69ms/step - accuracy: 0.9588 - loss: 0.0620 - val_accuracy: 0.8734 - val_loss: 0.1282
Epoch 20/20
30/30 ━━━━━━━━━━━━━━━━━━━━ 4s 130ms/step - accuracy: 0.9612 - loss: 0.0595 - val_accuracy: 0.8603 - val_loss: 0.1415
```

```
# Extracting training history data:
training_history_dictionary = training_history.history

# Displaying available keys in the history dictionary:
training_history_dictionary.keys()
```

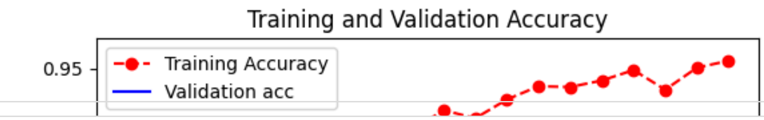
```
dict_keys(['accuracy', 'loss', 'val_accuracy', 'val_loss'])
```

```
# Importing the required library for visualization:
import matplotlib.pyplot as plt

# Extracting the model’s training history for analysis:
history_dictionary = training_history.history
loss_values = history_dict["loss"]
val_loss_values = history_dict["val_loss"]
epochs = range(1, len(loss_values) + 1)

# Plotting training loss with a dashed red line and markers:
plt.plot(epochs, val_loss_values, label='Validation Loss')

# Plotting the training loss curve using a red dashed line with point markers:
plt.plot(epochs, loss_values, "ro--", label="Training Loss")

# Adding title and labels:
plt.title("Training and Validation Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")

# Displaying legend:
plt.legend()
```

```
# Showing the plot:
plt.show()
```




```
# Clearing the previous plot:
plt.clf()

# Extracting accuracy metrics from the training history:
acc = training_history_dict["accuracy"]
val_acc = training_history_dict["val_accuracy"]

# # Visualizing model accuracy (training) with red dashed line and circles:
plt.plot(epochs, acc, "ro--", label="Training Accuracy")

# # Visualizing validation accuracy trend using green solid line and squares:
plt.plot(epochs, val_acc, "b", label="Validation acc")

# Adding title and labels:
plt.title("Training and Validation Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")

# Displaying the plot legend for better readability:
plt.legend()

# Visualizing the plot:
plt.show()
```




```
# Measuring model performance on the test set:
test_results = neural_net.evaluate(A_test_encoded, B_test)
test_results
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - accuracy: 0.8471 - loss: 0.1494
[0.14833103120326996, 0.8489999771118164]
```

```
# Checking how stable the model is when we add more nodes and Training a new model with more layers and using MSE as the loss functic

from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras import regularizers

# Defining a new model with dropout layers:
New_model = keras.Sequential([
    layers.Dense(64, kernel_regularizer=regularizers.l2(0.005), activation="tanh"),
    layers.Dropout(0.5),
    layers.Dense(64, activation="tanh"),
    layers.Dropout(0.5),
    layers.Dense(64, activation="tanh"),
    layers.Dropout(0.5),
    layers.Dense(1, activation="sigmoid")
])

# Compiling the model with Adam optimizer:
New_model.compile(optimizer="adam",
                  loss="mse",
                  metrics=["accuracy"])

# Training the model for 4 epochs with a batch size of 512
New_model.fit(A_train_encoded, B_train, epochs=4, batch_size=512)

# Evaluating the model on the test set
New_results = new_model.evaluate(A_test_encoded, B_test)

# Displaying the test evaluation results
New_results
```

```
Epoch 1/4
49/49 ━━━━━━━━━━━━━━━━━━━━ 5s 56ms/step - accuracy: 0.6907 - loss: 0.5361
Epoch 2/4
49/49 ━━━━━━━━━━━━━━━━━━━━ 5s 64ms/step - accuracy: 0.8938 - loss: 0.1541
Epoch 3/4
49/49 ━━━━━━━━━━━━━━━━━━━━ 5s 60ms/step - accuracy: 0.9039 - loss: 0.1421
Epoch 4/4
49/49 ━━━━━━━━━━━━━━━━━━━━ 5s 57ms/step - accuracy: 0.9092 - loss: 0.1398
782/782 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - accuracy: 0.8760 - loss: 0.1635
[0.16295744478702545, 0.8762800097465515]
```