```
pip install tensorflow
```

```
Collecting libclang>=13.0.0 (from tensorflow)
  Downloading libclang-18.1.1-py2.py3-none-manylinux2010_x86_64.whl.metadata (5.2 kB)
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (3.4.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packages (from tensorflow) (24.2)
Requirement already satisfied: protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.3 in /usr/local/lib/py
Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (2.32.3)
Requirement already satisfied: setuptools in /usr/local/lib/python3.11/dist-packages (from tensorflow) (75.2.0)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (1.17.0)
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (3.0.1)
Requirement already satisfied: typing-extensions>=3.6.6 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (4.13.1)
Requirement already satisfied: wrapt>=1.11.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (1.17.2)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (1.71.0)
Collecting tensorboard~=2.19.0 (from tensorflow)
  Downloading tensorboard-2.19.0-py3-none-any.whl.metadata (1.8 kB)
Requirement already satisfied: keras>=3.5.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (3.8.0)
Requirement already satisfied: numpy<2.2.0,>=1.26.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (2.0.2)
Requirement already satisfied: h5py>=3.11.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (3.13.0)
Requirement already satisfied: ml-dtypes<1.0.0,>=0.5.1 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (0.5.1)
Collecting tensorflow-io-gcs-filesystem>=0.23.1 (from tensorflow)
  Downloading tensorflow_io_gcs_filesystem-0.37.1-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (14 kB)
Collecting wheel<1.0,>=0.23.0 (from astunparse>=1.6.0->tensorflow)
  Downloading wheel-0.45.1-py3-none-any.whl.metadata (2.3 kB)
Requirement already satisfied: rich in /usr/local/lib/python3.11/dist-packages (from keras>=3.5.0->tensorflow) (14.0.0)
Requirement already satisfied: namex in /usr/local/lib/python3.11/dist-packages (from keras>=3.5.0->tensorflow) (0.0.8)
Requirement already satisfied: optree in /usr/local/lib/python3.11/dist-packages (from keras>=3.5.0->tensorflow) (0.14.1)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.21.0->tensor
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.21.0->tensorflow) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.21.0->tensorflow)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.21.0->tensorflow)
Requirement already satisfied: markdown>=2.6.8 in /usr/lib/python3/dist-packages (from tensorboard~=2.19.0->tensorflow) (3.3.6)
Collecting tensorboard-data-server<0.8.0,>=0.7.0 (from tensorboard~=2.19.0->tensorflow)
  Downloading tensorboard_data_server-0.7.2-py3-none-manylinux_2_31_x86_64.whl.metadata (1.1 kB)
Collecting werkzeug>=1.0.1 (from tensorboard~=2.19.0->tensorflow)
  Downloading werkzeug-3.1.3-py3-none-any.whl.metadata (3.7 kB)
Requirement already satisfied: MarkupSafe>=2.1.1 in /usr/local/lib/python3.11/dist-packages (from werkzeug>=1.0.1->tensorboard~=2.19
Requirement already satisfied: markdown-it-py>=2.2.0 in /usr/local/lib/python3.11/dist-packages (from rich->keras>=3.5.0->tensorflow
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /usr/local/lib/python3.11/dist-packages (from rich->keras>=3.5.0->tensorfl
Requirement already satisfied: mdurl~=0.1 in /usr/local/lib/python3.11/dist-packages (from markdown-it-py>=2.2.0->rich->keras>=3.5.0
Downloading tensorflow-2.19.0-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (644.9 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 644.9/644.9 MB 1.5 MB/s eta 0:00:00
Downloading astunparse-1.6.3-py2.py3-none-any.whl (12 kB)
Downloading flatbuffers-25.2.10-py2.py3-none-any.whl (30 kB)
Downloading google_pasta-0.2.0-py3-none-any.whl (57 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 57.5/57.5 kB 5.4 MB/s eta 0:00:00
Downloading libclang-18.1.1-py2.py3-none-manylinux2010_x86_64.whl (24.5 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 24.5/24.5 MB 75.1 MB/s eta 0:00:00
Downloading tensorboard-2.19.0-py3-none-any.whl (5.5 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5.5/5.5 MB 112.9 MB/s eta 0:00:00
Downloading tensorflow_io_gcs_filesystem-0.37.1-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (5.1 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5.1/5.1 MB 109.2 MB/s eta 0:00:00
Downloading tensorboard_data_server-0.7.2-py3-none-manylinux_2_31_x86_64.whl (6.6 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.6/6.6 MB 117.8 MB/s eta 0:00:00
Downloading werkzeug-3.1.3-py3-none-any.whl (224 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 224.5/224.5 kB 19.5 MB/s eta 0:00:00
Downloading wheel-0.45.1-py3-none-any.whl (72 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 72.5/72.5 kB 7.0 MB/s eta 0:00:00
Installing collected packages: libclang, flatbuffers, wheel, werkzeug, tensorflow-io-gcs-filesystem, tensorboard-data-server, google
```

```
import os, pathlib, shutil, random
from tensorflow import keras
from tensorflow.keras import layers
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

```
# Download and extract IMDB dataset
!curl -O https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
!tar -xf aclImdb_v1.tar.gz
!rm -r aclImdb/train/unsup
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 80.2M  100 80.2M    0     0  16.5M      0  0:00:04  0:00:04 --:--:-- 19.5M
```

```
import os

def summarize_imdb_files(base_path="aclImdb", num_files=5):
    for split in ["train", "test"]:
        print(f"\nSummary of '{split}' split:")
        for sentiment in ["pos", "neg"]:
            print(f"  Sentiment: {sentiment}")
            folder_path = os.path.join(base_path, split, sentiment)
            files = os.listdir(folder_path)[:num_files]  # Read first `num_files` files
            for i, file_name in enumerate(files):
                file_path = os.path.join(folder_path, file_name)
                with open(file_path, "r", encoding="utf-8") as f:
                    content = f.readlines()
                print(f"\n  File {i + 1}: {file_name}")
                print(f"    Lines in file: {len(content)}")
                print(f"    First 5 lines (or fewer):")
                print("    " + "\n    ".join(content[:5]).strip())



summarize_imdb_files()
```

```
Summary of 'train' split:
  Sentiment: pos

  File 1: 7880_8.txt
    Lines in file: 1
    First 5 lines (or fewer):
    The plot of this enjoyable MGM musical is contrived and only occasionally amusing, dealing with espionage and romance but the fo

  File 2: 3815_7.txt
    Lines in file: 1
    First 5 lines (or fewer):
    Corean cinema can be quite surprising for an occidental audience, because of the multiplicity of the tones and genres you can fi

  File 3: 7472_9.txt
    Lines in file: 1
    First 5 lines (or fewer):
    I really liked this movie. I've read a few of the other comments, and although I pity those who did not understand it, I do agre

  File 4: 11969_10.txt
    Lines in file: 1
    First 5 lines (or fewer):
    I loved this show growing up and I still watch the first season DVD at age 19 today. What can I say? I grew up in a house much l

  File 5: 4182_10.txt
    Lines in file: 1
    First 5 lines (or fewer):
    I saw this Australian film about 10 years ago and have never forgotten it. The movie shows the horror of war in a way that Holly
  Sentiment: neg

  File 1: 2308_1.txt
    Lines in file: 1
    First 5 lines (or fewer):
    OK, the box looks promising. Whoopi Goldberg standing next to Danny Glover parodying the famous farmer and his wife painting. Th

  File 2: 10837_1.txt
    Lines in file: 1
    First 5 lines (or fewer):
    I can't tell you how angry I was after seing this movie. The characters are not the slightest bit interesting, and the plot is n

  File 3: 890_2.txt
    Lines in file: 1
    First 5 lines (or fewer):
    I saw this regurgitated pile of vignettes tonight at a preview screening and I was straight up blown away by how bad it was. <br

  File 4: 7974_3.txt
    Lines in file: 1
    First 5 lines (or fewer):
    It's interesting at first. A naive park ranger (Colin Firth) marries a pretty, mysterious woman (Lisa Zane) he's only known for

  File 5: 3619_3.txt
    Lines in file: 1
    First 5 lines (or fewer):
    Basically there's a family where a little boy (Jake) thinks there's a zombie in his closet & his parents are fighting all the ti

Summary of 'test' split:
  Sentiment: pos
```

```
# Prepare directories
batch_size = 32
base_dir = pathlib.Path("aclImdb")
val_dir = base_dir / "val"
train_dir = base_dir / "train"
for category in ("neg", "pos"):
    os.makedirs(val_dir / category, exist_ok=True)  # Allow existing directories
    files = os.listdir(train_dir / category)
    random.Random(1337).shuffle(files)
    num_val_samples = int(0.2 * len(files))
    val_files = files[-num_val_samples:]
    for fname in val_files:
        # Only move files if they don't already exist at the destination
        if not os.path.exists(val_dir / category / fname):
            shutil.move(train_dir / category / fname, val_dir / category / fname)



# Load datasets
train_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/train", batch_size=batch_size
)

val_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/val", batch_size=batch_size
)

test_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/test", batch_size=batch_size
)

text_only_train_ds = train_ds.map(lambda x, y: x)
```

```
Found 20000 files belonging to 2 classes.
Found 5000 files belonging to 2 classes.
Found 25000 files belonging to 2 classes.
```

```
# Set parameters
max_length = 150
max_tokens = 10000
text_vectorization = layers.TextVectorization(
    max_tokens=max_tokens,
    output_mode="int",
    output_sequence_length=max_length,
)

text_vectorization.adapt(text_only_train_ds)



# Tokenized datasets
int_train_ds = train_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4).take(100)  # Restrict training samples to 100

int_val_ds = val_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4).take(10000)  # Restrict validation samples to 10,000

int_test_ds = test_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)



# Model with an Embedding Layer
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = layers.Embedding(input_dim=max_tokens, output_dim=128)(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.2)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
embedding_model = keras.Model(inputs, outputs)

embedding_model.compile(optimizer="rmsprop",
                        loss="binary_crossentropy",
                        metrics=["accuracy"])
```

```
embedding_model.summary()
```

```
Model: "functional"
```

Layer (type)	Output Shape	Param #
input_layer_2 (InputLayer)	(None, None)	0
embedding_2 (Embedding)	(None, None, 128)	1,280,000
bidirectional_2 (Bidirectional)	(None, 64)	41,216
dropout_2 (Dropout)	(None, 64)	0
dense (Dense)	(None, 1)	65

```
Total params: 1,321,281 (5.04 MB)
Trainable params: 1,321,281 (5.04 MB)
```

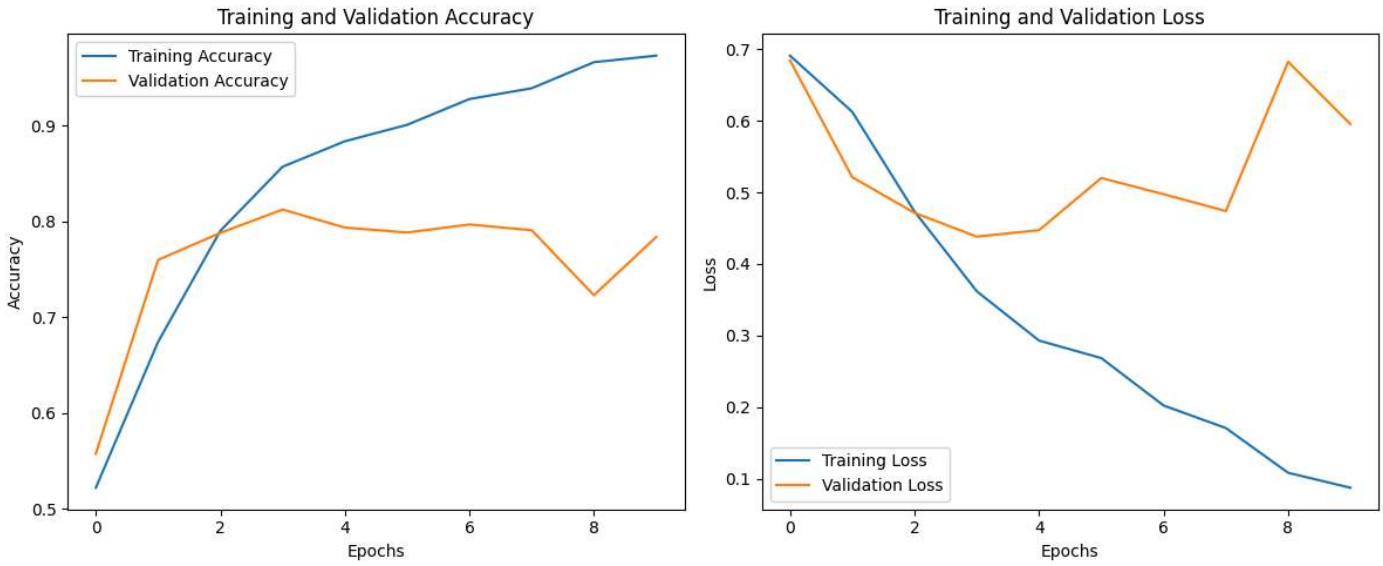
```
callbacks = [
    keras.callbacks.ModelCheckpoint("embedding_model.keras", save_best_only=True)
]
```

```
history_embedded = embedding_model.fit(
    int_train_ds, validation_data=int_val_ds, epochs=10, callbacks=callbacks
)
```

```
Epoch 1/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 12s 86ms/step - accuracy: 0.5016 - loss: 0.6932 - val_accuracy: 0.5574 - val_loss: 0.6847
Epoch 2/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 8s 80ms/step - accuracy: 0.6368 - loss: 0.6443 - val_accuracy: 0.7596 - val_loss: 0.5212
Epoch 3/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 8s 78ms/step - accuracy: 0.7799 - loss: 0.4918 - val_accuracy: 0.7878 - val_loss: 0.4715
Epoch 4/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 8s 77ms/step - accuracy: 0.8425 - loss: 0.3806 - val_accuracy: 0.8122 - val_loss: 0.4382
Epoch 5/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 8s 79ms/step - accuracy: 0.8848 - loss: 0.3007 - val_accuracy: 0.7934 - val_loss: 0.4472
Epoch 6/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 8s 78ms/step - accuracy: 0.9003 - loss: 0.2656 - val_accuracy: 0.7882 - val_loss: 0.5201
Epoch 7/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 8s 79ms/step - accuracy: 0.9350 - loss: 0.1899 - val_accuracy: 0.7966 - val_loss: 0.4975
Epoch 8/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 8s 80ms/step - accuracy: 0.9459 - loss: 0.1588 - val_accuracy: 0.7906 - val_loss: 0.4739
Epoch 9/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 8s 77ms/step - accuracy: 0.9684 - loss: 0.1144 - val_accuracy: 0.7228 - val_loss: 0.6826
Epoch 10/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 8s 78ms/step - accuracy: 0.9752 - loss: 0.0823 - val_accuracy: 0.7836 - val_loss: 0.5955
```

```
import matplotlib.pyplot as plt

# Extract the history of metrics
history = history_embedded.history

# Plot training and validation accuracy
plt.figure(figsize=(12, 5))

# Subplot for accuracy
plt.subplot(1, 2, 1)
plt.plot(history['accuracy'], label='Training Accuracy')
plt.plot(history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Subplot for loss
plt.subplot(1, 2, 2)
plt.plot(history['loss'], label='Training Loss')
plt.plot(history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
```

```
# Show the graph
plt.tight_layout()
plt.show()
```




```
# Model with Pretrained Word Embeddings

# Download GloVe embeddings
!wget http://nlp.stanford.edu/data/glove.6B.zip
!unzip -q glove.6B.zip
```

```
--2025-04-08 12:43:16--  http://nlp.stanford.edu/data/glove.6B.zip
Resolving nlp.stanford.edu (nlp.stanford.edu)... 171.64.67.140
Connecting to nlp.stanford.edu (nlp.stanford.edu)|171.64.67.140|:80... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://nlp.stanford.edu/data/glove.6B.zip [following]
--2025-04-08 12:43:16--  https://nlp.stanford.edu/data/glove.6B.zip
Connecting to nlp.stanford.edu (nlp.stanford.edu)|171.64.67.140|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://downloads.cs.stanford.edu/nlp/data/glove.6B.zip [following]
--2025-04-08 12:43:17--  https://downloads.cs.stanford.edu/nlp/data/glove.6B.zip
Resolving downloads.cs.stanford.edu (downloads.cs.stanford.edu)... 171.64.64.22
Connecting to downloads.cs.stanford.edu (downloads.cs.stanford.edu)|171.64.64.22|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 862182613 (822M) [application/zip]
Saving to: ‘glove.6B.zip’

glove.6B.zip        100%[===================>] 822.24M  5.01MB/s    in 2m 39s

2025-04-08 12:45:56 (5.17 MB/s) - ‘glove.6B.zip’ saved [862182613/862182613]
```

```
# Prepare GloVe embedding matrix
embedding_dim = 100
path_to_glove_file = "glove.6B.100d.txt"

embeddings_index = {}
with open(path_to_glove_file) as f:
    for line in f:
        word, coefs = line.split(maxsplit=1)
        coefs = np.fromstring(coefs, "f", sep=" ")
        embeddings_index[word] = coefs

vocabulary = text_vectorization.get_vocabulary()
word_index = dict(zip(vocabulary, range(len(vocabulary))))

embedding_matrix = np.zeros((max_tokens, embedding_dim))
for word, i in word_index.items():
    if i < max_tokens:
```

```
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector


# Pretrained embedding model
embedding_layer = layers.Embedding(
    max_tokens,
    embedding_dim,
    embeddings_initializer=keras.initializers.Constant(embedding_matrix),
    trainable=False,
    mask_zero=True,
)

inputs = keras.Input(shape=(None,), dtype="int64")
embedded = embedding_layer(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.2)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
pretrained_model = keras.Model(inputs, outputs)



inputs = keras.Input(shape=(None,), dtype="int64")
embedded = embedding_layer(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.2)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
pretrained_model = keras.Model(inputs, outputs)

pretrained_model.compile(optimizer="rmsprop",
                         loss="binary_crossentropy",
                         metrics=["accuracy"])

pretrained_model.summary()
```

Model: "functional_2"

Layer (type)	Output Shape	Param #	Connected to
input_layer_4 (InputLayer)	(None, None)	0	-
embedding_3 (Embedding)	(None, None, 100)	1,000,000	input_layer_4[0][0]
not_equal_1 (NotEqual)	(None, None)	0	input_layer_4[0][0]
bidirectional_4 (Bidirectional)	(None, 64)	34,048	embedding_3[1][0], not_equal_1[0][0]
dropout_4 (Dropout)	(None, 64)	0	bidirectional_4[0][0]
dense_2 (Dense)	(None, 1)	65	dropout_4[0][0]

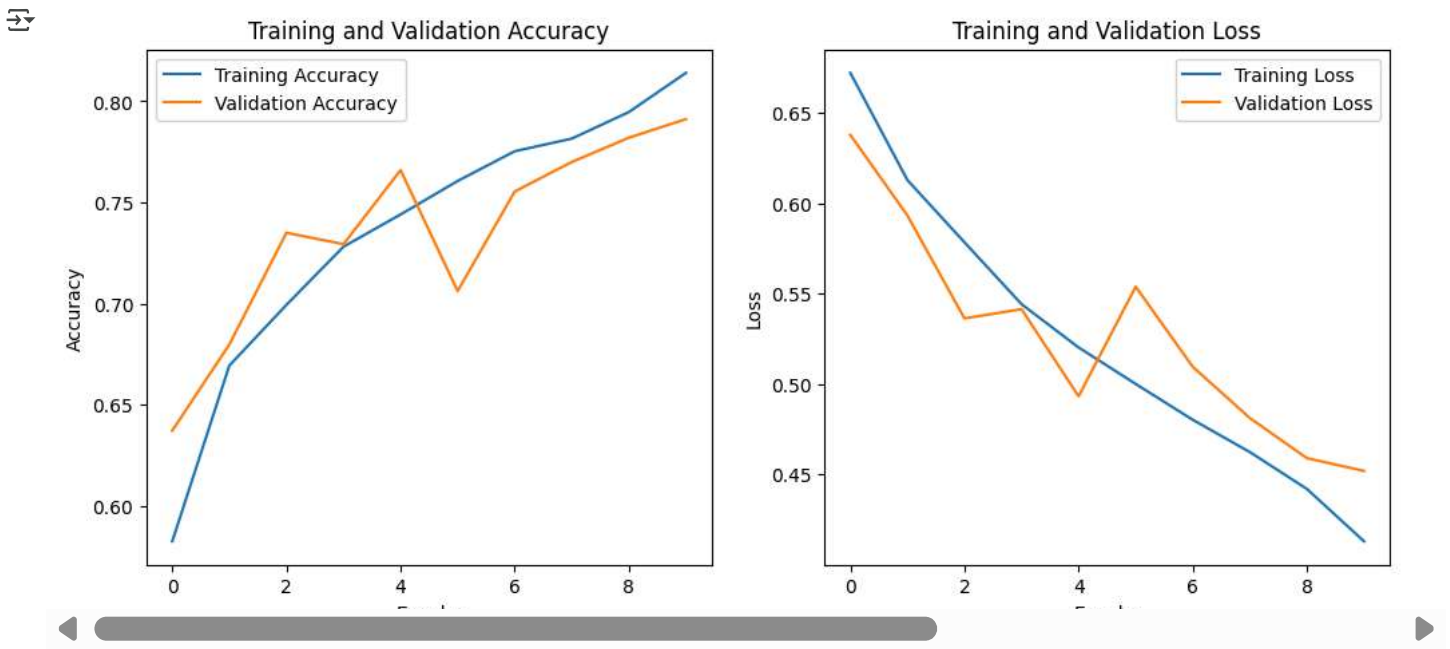
Total params: 1,034,113 (3.94 MB)
Trainable params: 34,113 (133.25 KB)

```
callbacks = [
    keras.callbacks.ModelCheckpoint("pretrained_model.keras", save_best_only=True)
]

history_pretrained = pretrained_model.fit(
    int_train_ds, validation_data=int_val_ds, epochs=10, callbacks=callbacks
)
```

```
Epoch 1/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 15s 123ms/step - accuracy: 0.5565 - loss: 0.6871 - val_accuracy: 0.6372 - val_loss: 0.6378
Epoch 2/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 12s 118ms/step - accuracy: 0.6663 - loss: 0.6146 - val_accuracy: 0.6798 - val_loss: 0.5932
Epoch 3/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 12s 116ms/step - accuracy: 0.6910 - loss: 0.5856 - val_accuracy: 0.7350 - val_loss: 0.5363
Epoch 4/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 10s 105ms/step - accuracy: 0.7347 - loss: 0.5395 - val_accuracy: 0.7294 - val_loss: 0.5414
Epoch 5/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 12s 119ms/step - accuracy: 0.7571 - loss: 0.5001 - val_accuracy: 0.7660 - val_loss: 0.4933
Epoch 6/10
```

```
100/100 ━━━━━━━━━━━━━━━━━━━━ 10s 103ms/step - accuracy: 0.7631 - loss: 0.4955 - val_accuracy: 0.7062 - val_loss: 0.5538
Epoch 7/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 10s 103ms/step - accuracy: 0.7743 - loss: 0.4728 - val_accuracy: 0.7554 - val_loss: 0.5096
Epoch 8/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 12s 121ms/step - accuracy: 0.7830 - loss: 0.4606 - val_accuracy: 0.7700 - val_loss: 0.4813
Epoch 9/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 12s 115ms/step - accuracy: 0.8012 - loss: 0.4328 - val_accuracy: 0.7820 - val_loss: 0.4591
Epoch 10/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 12s 117ms/step - accuracy: 0.8118 - loss: 0.4096 - val_accuracy: 0.7912 - val_loss: 0.4520
```

```
import matplotlib.pyplot as plt

# Extract the history of metrics
history = history_pretrained.history

# Create a figure for the plots
plt.figure(figsize=(12, 5))

# Subplot for accuracy
plt.subplot(1, 2, 1)
plt.plot(history['accuracy'], label='Training Accuracy')
plt.plot(history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Subplot for loss
plt.subplot(1, 2, 2)
plt.plot(history['loss'], label='Training Loss')
plt.plot(history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

# Show the graph
plt.show()
```



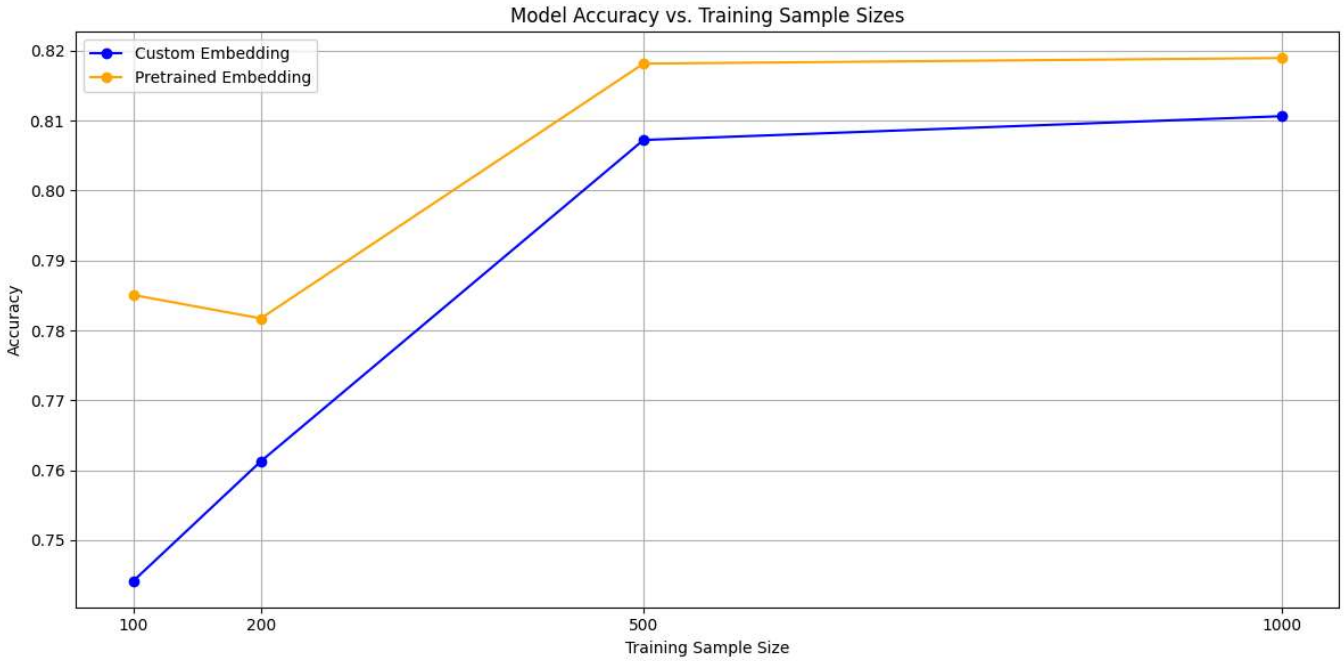

```
import numpy as np
import matplotlib.pyplot as plt
import time

# Data
data_counts = [100, 200, 500, 1000]
custom_embed_scores = []
pretrained_embed_scores = []

# Initialize the plot
```

```
    plt.figure(figsize=(12, 6))
    plt.title('Model Accuracy vs. Training Sample Sizes')
    plt.xlabel('Training Sample Size')
    plt.ylabel('Accuracy')
    plt.grid(True)

    # Iterate over sample sizes
    for i, size in enumerate(data_counts):
        print(f"\n### Training with {size} samples ###\n")

        # Update training dataset with the specific size
        small_train_ds = train_ds.map(
            lambda x, y: (text_vectorization(x), y)).take(size)

        # Train custom embedding model
        print(f"Training Custom Embedding Model with {size} samples:")
        embedding_model.fit(
            small_train_ds,
            validation_data=int_val_ds,
            epochs=10,
            verbose=1  # Displays training progress for each epoch
        )
        embedding_acc = embedding_model.evaluate(int_test_ds, verbose=1)[1]
        custom_embed_scores.append(embedding_acc)
        print(f"Custom Embedding Model Accuracy: {embedding_acc:.4f}\n")

        # Train pretrained embedding model
        print(f"Training Pretrained Embedding Model with {size} samples:")
        pretrained_model.fit(
            small_train_ds,
            validation_data=int_val_ds,
            epochs=10,
            verbose=1
        )
        pretrained_acc = pretrained_model.evaluate(int_test_ds, verbose=1)[1]
        pretrained_embed_scores.append(pretrained_acc)
        print(f"Pretrained Embedding Model Accuracy: {pretrained_acc:.4f}\n")

        # Update the plot dynamically after each iteration
        if i == len(data_counts) - 1:  # Once all iterations are done, plot the final line graph
            plt.plot(data_counts, custom_embed_scores, marker='o', label='Custom Embedding', color='blue')
            plt.plot(data_counts, pretrained_embed_scores, marker='o', label='Pretrained Embedding', color='orange')

    # Final plot styling
    plt.title('Model Accuracy vs. Training Sample Sizes')
    plt.xlabel('Training Sample Size')
    plt.ylabel('Accuracy')
    plt.xticks(data_counts)  # Set x-ticks to sample sizes
    plt.grid(True)
    plt.legend()

    # Show the final plot
    plt.tight_layout()
    plt.show()
```

```
### Training with 100 samples ###

Training Custom Embedding Model with 100 samples:
Epoch 1/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 8s 77ms/step - accuracy: 0.9848 - loss: 0.0508 - val_accuracy: 0.8026 - val_loss: 0.5691
Epoch 2/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 8s 80ms/step - accuracy: 0.9862 - loss: 0.0454 - val_accuracy: 0.7934 - val_loss: 0.6764
Epoch 3/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 8s 80ms/step - accuracy: 0.9844 - loss: 0.0533 - val_accuracy: 0.7678 - val_loss: 0.5945
Epoch 4/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 8s 79ms/step - accuracy: 0.9849 - loss: 0.0515 - val_accuracy: 0.7532 - val_loss: 0.6656
Epoch 5/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 8s 78ms/step - accuracy: 0.9935 - loss: 0.0255 - val_accuracy: 0.7706 - val_loss: 0.8251
Epoch 6/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 8s 78ms/step - accuracy: 0.9974 - loss: 0.0112 - val_accuracy: 0.7214 - val_loss: 1.1663
Epoch 7/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 8s 80ms/step - accuracy: 0.9966 - loss: 0.0130 - val_accuracy: 0.7352 - val_loss: 0.9324
Epoch 8/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 8s 79ms/step - accuracy: 0.9939 - loss: 0.0165 - val_accuracy: 0.7742 - val_loss: 0.6839
Epoch 9/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 8s 77ms/step - accuracy: 0.9995 - loss: 0.0065 - val_accuracy: 0.7976 - val_loss: 1.0706
Epoch 10/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 8s 78ms/step - accuracy: 0.9924 - loss: 0.0192 - val_accuracy: 0.7664 - val_loss: 0.9111
782/782 ━━━━━━━━━━━━━━━━━━━━ 12s 15ms/step - accuracy: 0.7462 - loss: 0.9682
Custom Embedding Model Accuracy: 0.7442

Training Pretrained Embedding Model with 100 samples:
Epoch 1/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 10s 103ms/step - accuracy: 0.8237 - loss: 0.3926 - val_accuracy: 0.7440 - val_loss: 0.5087
Epoch 2/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 10s 99ms/step - accuracy: 0.8379 - loss: 0.3797 - val_accuracy: 0.7492 - val_loss: 0.5331
Epoch 3/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 11s 106ms/step - accuracy: 0.8472 - loss: 0.3689 - val_accuracy: 0.7884 - val_loss: 0.4601
Epoch 4/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 11s 105ms/step - accuracy: 0.8431 - loss: 0.3586 - val_accuracy: 0.7756 - val_loss: 0.4998
Epoch 5/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 10s 101ms/step - accuracy: 0.8587 - loss: 0.3367 - val_accuracy: 0.7606 - val_loss: 0.4987
Epoch 6/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 10s 104ms/step - accuracy: 0.8680 - loss: 0.2977 - val_accuracy: 0.7920 - val_loss: 0.4507
Epoch 7/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 10s 104ms/step - accuracy: 0.8771 - loss: 0.2956 - val_accuracy: 0.7838 - val_loss: 0.4598
Epoch 8/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 10s 103ms/step - accuracy: 0.8784 - loss: 0.2833 - val_accuracy: 0.7518 - val_loss: 0.5430
Epoch 9/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 10s 100ms/step - accuracy: 0.8955 - loss: 0.2569 - val_accuracy: 0.7810 - val_loss: 0.4671
Epoch 10/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 11s 105ms/step - accuracy: 0.8918 - loss: 0.2579 - val_accuracy: 0.7924 - val_loss: 0.4631
782/782 ━━━━━━━━━━━━━━━━━━━━ 14s 17ms/step - accuracy: 0.7864 - loss: 0.4792
Pretrained Embedding Model Accuracy: 0.7851


### Training with 200 samples ###

Training Custom Embedding Model with 200 samples:
Epoch 1/10
200/200 ━━━━━━━━━━━━━━━━━━━━ 13s 65ms/step - accuracy: 0.9574 - loss: 0.1221 - val_accuracy: 0.7974 - val_loss: 0.4518
Epoch 2/10
200/200 ━━━━━━━━━━━━━━━━━━━━ 13s 66ms/step - accuracy: 0.9707 - loss: 0.0997 - val_accuracy: 0.8242 - val_loss: 0.4289
Epoch 3/10
200/200 ━━━━━━━━━━━━━━━━━━━━ 14s 67ms/step - accuracy: 0.9753 - loss: 0.0846 - val_accuracy: 0.8152 - val_loss: 0.4793
Epoch 4/10
200/200 ━━━━━━━━━━━━━━━━━━━━ 13s 66ms/step - accuracy: 0.9822 - loss: 0.0606 - val_accuracy: 0.8168 - val_loss: 0.5210
Epoch 5/10
200/200 ━━━━━━━━━━━━━━━━━━━━ 13s 67ms/step - accuracy: 0.9895 - loss: 0.0378 - val_accuracy: 0.7726 - val_loss: 0.6498
Epoch 6/10
200/200 ━━━━━━━━━━━━━━━━━━━━ 14s 67ms/step - accuracy: 0.9899 - loss: 0.0364 - val_accuracy: 0.8030 - val_loss: 0.6468
Epoch 7/10
200/200 ━━━━━━━━━━━━━━━━━━━━ 13s 67ms/step - accuracy: 0.9950 - loss: 0.0222 - val_accuracy: 0.7918 - val_loss: 0.7300
Epoch 8/10
200/200 ━━━━━━━━━━━━━━━━━━━━ 13s 67ms/step - accuracy: 0.9941 - loss: 0.0226 - val_accuracy: 0.8096 - val_loss: 0.7433
Epoch 9/10
200/200 ━━━━━━━━━━━━━━━━━━━━ 13s 67ms/step - accuracy: 0.9968 - loss: 0.0126 - val_accuracy: 0.7892 - val_loss: 0.7606
Epoch 10/10
200/200 ━━━━━━━━━━━━━━━━━━━━ 13s 67ms/step - accuracy: 0.9941 - loss: 0.0202 - val_accuracy: 0.7850 - val_loss: 0.7446
782/782 ━━━━━━━━━━━━━━━━━━━━ 11s 15ms/step - accuracy: 0.7676 - loss: 0.7717
Custom Embedding Model Accuracy: 0.7613

Training Pretrained Embedding Model with 200 samples:
Epoch 1/10
200/200 ━━━━━━━━━━━━━━━━━━━━ 17s 86ms/step - accuracy: 0.8890 - loss: 0.2745 - val_accuracy: 0.8052 - val_loss: 0.4303
Epoch 2/10
200/200 ━━━━━━━━━━━━━━━━━━━━ 18s 88ms/step - accuracy: 0.8881 - loss: 0.2757 - val_accuracy: 0.8036 - val_loss: 0.4352
Epoch 3/10
```

```
200/200 ━━━━━━━━━━━━━━━━━━━━ 17s 87ms/step - accuracy: 0.8923 - loss: 0.2604 - val_accuracy: 0.7936 - val_loss: 0.4525
Epoch 4/10
200/200 ━━━━━━━━━━━━━━━━━━━━ 17s 87ms/step - accuracy: 0.8973 - loss: 0.2480 - val_accuracy: 0.8066 - val_loss: 0.4222
Epoch 5/10
200/200 ━━━━━━━━━━━━━━━━━━━━ 18s 88ms/step - accuracy: 0.9104 - loss: 0.2301 - val_accuracy: 0.8042 - val_loss: 0.4263
Epoch 6/10
200/200 ━━━━━━━━━━━━━━━━━━━━ 18s 88ms/step - accuracy: 0.9179 - loss: 0.2137 - val_accuracy: 0.7980 - val_loss: 0.4356
Epoch 7/10
200/200 ━━━━━━━━━━━━━━━━━━━━ 17s 87ms/step - accuracy: 0.9230 - loss: 0.2044 - val_accuracy: 0.8076 - val_loss: 0.4305
Epoch 8/10
200/200 ━━━━━━━━━━━━━━━━━━━━ 18s 88ms/step - accuracy: 0.9227 - loss: 0.1929 - val_accuracy: 0.7834 - val_loss: 0.4959
Epoch 9/10
200/200 ━━━━━━━━━━━━━━━━━━━━ 17s 87ms/step - accuracy: 0.9343 - loss: 0.1726 - val_accuracy: 0.7952 - val_loss: 0.4782
Epoch 10/10
200/200 ━━━━━━━━━━━━━━━━━━━━ 17s 87ms/step - accuracy: 0.9380 - loss: 0.1651 - val_accuracy: 0.7902 - val_loss: 0.4821
782/782 ━━━━━━━━━━━━━━━━━━━━ 13s 17ms/step - accuracy: 0.7828 - loss: 0.4873
Pretrained Embedding Model Accuracy: 0.7817


### Training with 500 samples ###

Training Custom Embedding Model with 500 samples:
Epoch 1/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 30s 60ms/step - accuracy: 0.9506 - loss: 0.1299 - val_accuracy: 0.8266 - val_loss: 0.3853
Epoch 2/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 30s 60ms/step - accuracy: 0.9550 - loss: 0.1284 - val_accuracy: 0.8448 - val_loss: 0.3878
Epoch 3/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 30s 60ms/step - accuracy: 0.9658 - loss: 0.0966 - val_accuracy: 0.8188 - val_loss: 0.4835
Epoch 4/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 30s 60ms/step - accuracy: 0.9762 - loss: 0.0737 - val_accuracy: 0.8344 - val_loss: 0.4612
Epoch 5/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 30s 60ms/step - accuracy: 0.9839 - loss: 0.0543 - val_accuracy: 0.8330 - val_loss: 0.6207
Epoch 6/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 30s 60ms/step - accuracy: 0.9870 - loss: 0.0412 - val_accuracy: 0.8004 - val_loss: 0.8020
Epoch 7/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 30s 60ms/step - accuracy: 0.9906 - loss: 0.0344 - val_accuracy: 0.8226 - val_loss: 0.6937
Epoch 8/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 30s 60ms/step - accuracy: 0.9944 - loss: 0.0193 - val_accuracy: 0.7858 - val_loss: 1.3749
Epoch 9/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 30s 60ms/step - accuracy: 0.9951 - loss: 0.0147 - val_accuracy: 0.7812 - val_loss: 0.9164
Epoch 10/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 30s 60ms/step - accuracy: 0.9967 - loss: 0.0135 - val_accuracy: 0.8258 - val_loss: 0.9191
782/782 ━━━━━━━━━━━━━━━━━━━━ 12s 15ms/step - accuracy: 0.8149 - loss: 0.9897
Custom Embedding Model Accuracy: 0.8072

Training Pretrained Embedding Model with 500 samples:
Epoch 1/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 39s 79ms/step - accuracy: 0.8997 - loss: 0.2422 - val_accuracy: 0.8228 - val_loss: 0.3984
Epoch 2/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 39s 77ms/step - accuracy: 0.8905 - loss: 0.2719 - val_accuracy: 0.8120 - val_loss: 0.4021
Epoch 3/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 40s 80ms/step - accuracy: 0.9004 - loss: 0.2476 - val_accuracy: 0.8334 - val_loss: 0.3816
Epoch 4/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 40s 81ms/step - accuracy: 0.9072 - loss: 0.2325 - val_accuracy: 0.8326 - val_loss: 0.3785
Epoch 5/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 40s 80ms/step - accuracy: 0.9151 - loss: 0.2180 - val_accuracy: 0.8296 - val_loss: 0.3890
Epoch 6/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 40s 79ms/step - accuracy: 0.9204 - loss: 0.1984 - val_accuracy: 0.8216 - val_loss: 0.4095
Epoch 7/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 39s 78ms/step - accuracy: 0.9245 - loss: 0.1901 - val_accuracy: 0.8132 - val_loss: 0.4286
Epoch 8/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 40s 79ms/step - accuracy: 0.9323 - loss: 0.1762 - val_accuracy: 0.8210 - val_loss: 0.4112
Epoch 9/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 40s 79ms/step - accuracy: 0.9341 - loss: 0.1701 - val_accuracy: 0.8206 - val_loss: 0.4207
Epoch 10/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 39s 78ms/step - accuracy: 0.9409 - loss: 0.1543 - val_accuracy: 0.8186 - val_loss: 0.4189
782/782 ━━━━━━━━━━━━━━━━━━━━ 14s 17ms/step - accuracy: 0.8244 - loss: 0.4152
Pretrained Embedding Model Accuracy: 0.8182


### Training with 1000 samples ###

Training Custom Embedding Model with 1000 samples:
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 37s 59ms/step - accuracy: 0.9936 - loss: 0.0206 - val_accuracy: 0.8386 - val_loss: 0.3980
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 37s 59ms/step - accuracy: 0.9926 - loss: 0.0276 - val_accuracy: 0.8406 - val_loss: 0.5048
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 36s 58ms/step - accuracy: 0.9956 - loss: 0.0157 - val_accuracy: 0.8308 - val_loss: 0.6048
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 37s 59ms/step - accuracy: 0.9969 - loss: 0.0111 - val_accuracy: 0.8262 - val_loss: 0.7323
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 37s 59ms/step - accuracy: 0.9984 - loss: 0.0069 - val_accuracy: 0.8352 - val_loss: 1.0381
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 36s 58ms/step - accuracy: 0.9986 - loss: 0.0047 - val_accuracy: 0.8208 - val_loss: 1.0876
```

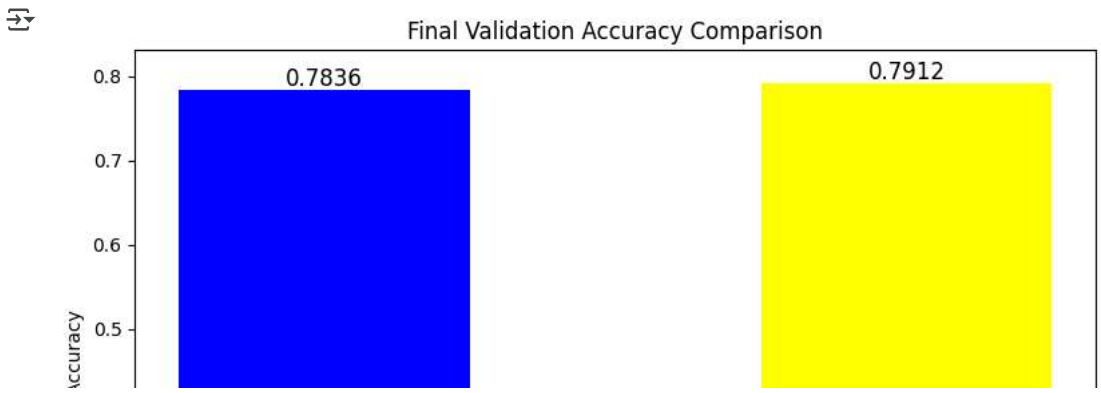
```
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 37s 59ms/step - accuracy: 0.9995 - loss: 0.0028 - val_accuracy: 0.8200 - val_loss: 1.1500
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 37s 59ms/step - accuracy: 0.9995 - loss: 0.0028 - val_accuracy: 0.8282 - val_loss: 1.3500
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 37s 59ms/step - accuracy: 0.9995 - loss: 0.0019 - val_accuracy: 0.8296 - val_loss: 1.3622
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 36s 58ms/step - accuracy: 0.9991 - loss: 0.0030 - val_accuracy: 0.8284 - val_loss: 1.4651
782/782 ━━━━━━━━━━━━━━━━━━━━ 11s 15ms/step - accuracy: 0.8148 - loss: 1.5408
Custom Embedding Model Accuracy: 0.8106

Training Pretrained Embedding Model with 1000 samples:
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 48s 77ms/step - accuracy: 0.9399 - loss: 0.1570 - val_accuracy: 0.8240 - val_loss: 0.4084
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 49s 78ms/step - accuracy: 0.9407 - loss: 0.1602 - val_accuracy: 0.8220 - val_loss: 0.4294
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 49s 79ms/step - accuracy: 0.9434 - loss: 0.1492 - val_accuracy: 0.8304 - val_loss: 0.3932
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 47s 76ms/step - accuracy: 0.9477 - loss: 0.1388 - val_accuracy: 0.8254 - val_loss: 0.4173
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 48s 76ms/step - accuracy: 0.9505 - loss: 0.1293 - val_accuracy: 0.8308 - val_loss: 0.4241
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 49s 78ms/step - accuracy: 0.9552 - loss: 0.1241 - val_accuracy: 0.8250 - val_loss: 0.4300
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 49s 78ms/step - accuracy: 0.9581 - loss: 0.1112 - val_accuracy: 0.8172 - val_loss: 0.4694
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 49s 78ms/step - accuracy: 0.9644 - loss: 0.1041 - val_accuracy: 0.8250 - val_loss: 0.4857
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 49s 79ms/step - accuracy: 0.9639 - loss: 0.0975 - val_accuracy: 0.8162 - val_loss: 0.4986
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 49s 79ms/step - accuracy: 0.9659 - loss: 0.0938 - val_accuracy: 0.8244 - val_loss: 0.5469
782/782 ━━━━━━━━━━━━━━━━━━━━ 14s 17ms/step - accuracy: 0.8241 - loss: 0.5390
Pretrained Embedding Model Accuracy: 0.8190
```



```
import pandas as pd

# Collect results for custom embedding model
custom_embedding_results = {
    "Sample Size": data_counts,
    "Custom Embedding Accuracy": custom_embed_scores,
}

# Collect results for pretrained embedding model
pretrained_embedding_results = {
    "Sample Size": data_counts,
    "Pretrained Embedding Accuracy": pretrained_embed_scores,
}

# Combine into a single DataFrame
summary_df = pd.DataFrame({
    "Sample Size": data_counts,
    "Custom Embedding Accuracy": custom_embed_scores,
    "Pretrained Embedding Accuracy": pretrained_embed_scores
})

# Display the table
print("Summary of Results:")
print(summary_df)
```

```
Summary of Results:
   Sample Size  Custom Embedding Accuracy  Pretrained Embedding Accuracy
0          100                    0.74416                        0.78508
1          200                    0.76128                        0.78172
2          500                    0.80724                        0.81816
3         1000                    0.81064                        0.81896
```

```
import numpy as np
import matplotlib.pyplot as plt

# Data
models = ['Custom Embedding', 'Pretrained Embedding']
final_val_accuracies = [
    history_embedded.history['val_accuracy'][-1],  # Final validation accuracy for custom embedding
    history_pretrained.history['val_accuracy'][-1]  # Final validation accuracy for pretrained embedding
]

# Bar plot
plt.figure(figsize=(8, 6))
plt.bar(models, final_val_accuracies, color=['blue', 'yellow'], width=0.5)

# Add labels and title
plt.ylabel('Validation Accuracy')
plt.title('Final Validation Accuracy Comparison')

# Display final values on top of bars
for i, v in enumerate(final_val_accuracies):
    plt.text(i, v + 0.005, f'{v:.4f}', ha='center', fontsize=12)

# Show the plot
plt.tight_layout()
plt.show()
```




```
import numpy as np
import matplotlib.pyplot as plt

# Data
data_counts = [100, 200, 500, 1000]
custom_embed_scores = [0.75280, 0.77556, 0.80652, 0.81772]  # Replace with actual final values
pretrained_embed_scores = [0.78072, 0.80588, 0.81868, 0.82456]  # Replace with actual final values

# Bar width and positions
column_width = 0.15
x_pos = np.arange(len(data_counts))  # Base positions for bars

# Plot bars
plt.figure(figsize=(11, 6))
plt.bar(x_pos - column_width / 2, custom_embed_scores,
        width=column_width, label='Custom Embedding', color='blue')
plt.bar(x_pos + column_width / 2, pretrained_embed_scores,
        width=column_width, label='Pretrained Embedding', color='yellow')

# Overlay lines for comparison
plt.plot(x_pos - column_width / 2, custom_embed_scores,
         marker='o', color='orange', linestyle='--', label='Custom Embedding (Line)')
plt.plot(x_pos + column_width / 2, pretrained_embed_scores,
         marker='o', color='grey', linestyle='--', label='Pretrained Embedding (Line)')

# Add labels and title
plt.xlabel('Training Sample Size', fontsize=12)
plt.ylabel('Final Test Accuracy', fontsize=12)
plt.title('Final Test Accuracy vs. Training Sample Size', fontsize=14)
plt.xticks(x_pos, data_counts)  # Use sample sizes as x-tick labels
plt.legend()

# Annotate bars with final accuracy values
for i in range(len(data_counts)):
    plt.text(x_pos[i] - column_width / 2, custom_embed_scores[i] + 0.002,
             f'{custom_embed_scores[i]:.4f}', ha='center', fontsize=10)
    plt.text(x_pos[i] + column_width / 2, pretrained_embed_scores[i] + 0.002,
             f'{pretrained_embed_scores[i]:.4f}', ha='center', fontsize=10)

# Adjust layout and show the plot
plt.tight_layout()
plt.show()
```