05_sentiment_analysis_imdb

September 29, 2021

1 LSTM & Word Embeddings for Sentiment Classification

RNNs are commonly applied to various natural language processing tasks. We've already encountered sentiment analysis using text data in part three of this book.

We are now going to illustrate how to apply an RNN model to text data to detect positive or negative sentiment (which can easily be extended to a finer-grained sentiment scale). We are going to use word embeddings to represent the tokens in the documents. We covered word embeddings in Chapter 15, Word Embeddings. They are an excellent technique to convert text into a continuous vector representation such that the relative location of words in the latent space encodes useful semantic aspects based on the words' usage in context.

We saw in the previous RNN example that Keras has a built-in embedding layer that allows us to train vector representations specific to the task at hand. Alternatively, we can use pretrained vectors.

1.1 Imports & Settings

[1]:
```
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
%matplotlib inline

from pathlib import Path

import numpy as np
import pandas as pd
from sklearn.metrics import roc_auc_score

import tensorflow as tf
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
from tensorflow.keras.datasets import imdb
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, GRU, Embedding
from tensorflow.keras.preprocessing.sequence import pad_sequences
import tensorflow.keras.backend as K

import matplotlib.pyplot as plt
import seaborn as sns
```

[3]:
```
gpu_devices = tf.config.experimental.list_physical_devices('GPU')
if gpu_devices:
    print('Using GPU')
    tf.config.experimental.set_memory_growth(gpu_devices[0], True)
else:
    print('Using CPU')
```

```
Using CPU
```

[4]:
```
sns.set_style('whitegrid')
np.random.seed(42)
```

[5]:
```
results_path = Path('results', 'sentiment_imdb')
if not results_path.exists():
    results_path.mkdir(parents=True)
```

1.2 Load Reviews

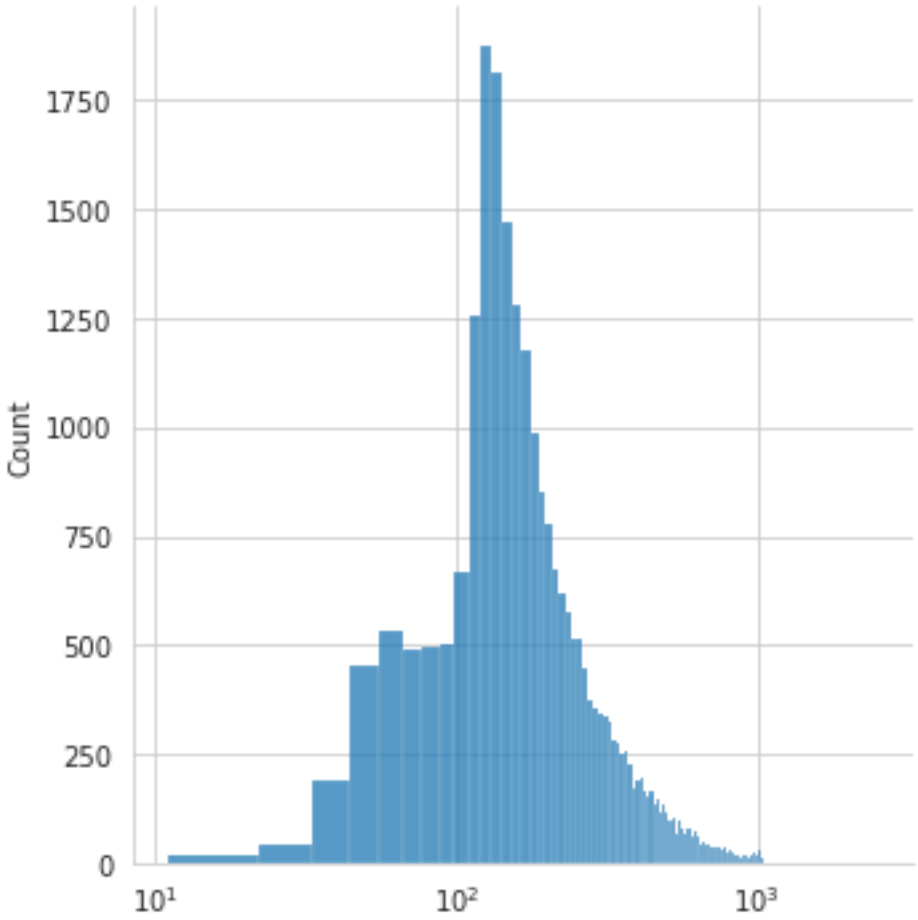
To keep the data manageable, we will illustrate this use case with the IMDB reviews dataset, which contains 50,000 positive and negative movie reviews evenly split into a train and a test set, and with balanced labels in each dataset. The vocabulary consists of 88,586 tokens.

The dataset is bundled into Keras and can be loaded so that each review is represented as an integer-encoded sequence. We can limit the vocabulary to num_words while filtering out frequent and likely less informative words using skip_top, as well as sentences longer than maxlen. We can also choose oov_char, which represents tokens we chose to exclude from the vocabulary on frequency grounds, as follows:

[6]:
```
vocab_size = 20000
```

[7]:
```
(X_train, y_train), (X_test, y_test) = imdb.load_data(seed=42,
                                                      skip_top=0,
                                                      maxlen=None,
                                                      oov_char=2,
                                                      index_from=3,
                                                      num_words=vocab_size)
```

[8]:
```
ax = sns.displot([len(review) for review in X_train])
ax.set(xscale='log');
```



1.3 Prepare Data

In the second step, convert the lists of integers into fixed-size arrays that we can stack and provide as input to our RNN. The pad_sequence function produces arrays of equal length, truncated, and padded to conform to maxlen, as follows:

[9]:
```
maxlen = 100
```

[10]:
```
X_train_padded = pad_sequences(X_train,
                         truncating='pre',
                         padding='pre',
                         maxlen=maxlen)

X_test_padded = pad_sequences(X_test,
                        truncating='pre',
                        padding='pre',
                        maxlen=maxlen)
```

[11]:
```
X_train_padded.shape, X_test_padded.shape
```

[11]:
```
((25000, 100), (25000, 100))
```

1.4 Define Model Architecture

Now we can define our RNN architecture. The first layer learns the word embeddings. We define the embedding dimension as previously using the input_dim keyword to set the number of tokens that we need to embed, the output_dim keyword, which defines the size of each embedding, and how long each input sequence is going to be.

[12]:
```
K.clear_session()
```

1.4.1 Custom Loss Metric

[13]:
```
embedding_size = 100
```

Note that we are using GRUs this time, which train faster and perform better on smaller data. We are also using dropout for regularization, as follows:

[14]:
```
rnn = Sequential([
    Embedding(input_dim=vocab_size,
              output_dim= embedding_size,
              input_length=maxlen),
    GRU(units=32,
        dropout=0.2, # comment out to use optimized GPU implementation
        recurrent_dropout=0.2),
    Dense(1, activation='sigmoid')
])
rnn.summary()
```

```
WARNING:tensorflow:Layer gru will not use cuDNN kernel since it doesn't meet the
cuDNN kernel criteria. It will use generic GPU kernel as fallback when running
on GPU
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 100, 100)          2000000
_________________________________________________________________
gru (GRU)                    (None, 32)                12864
_________________________________________________________________
dense (Dense)                (None, 1)                 33
=================================================================
Total params: 2,012,897
Trainable params: 2,012,897
Non-trainable params: 0
_________________________________________________________________
```

The resulting model has over 2 million parameters.

We compile the model to use our custom AUC metric, which we introduced previously, and train with early stopping:

[15]:
```
rnn.compile(loss='binary_crossentropy',
            optimizer='RMSProp',
            metrics=['accuracy',
                     tf.keras.metrics.AUC(name='AUC')])
```

[16]:
```
rnn_path = (results_path / 'lstm.h5').as_posix()

checkpointer = ModelCheckpoint(filepath=rnn_path,
                               verbose=1,
                               monitor='val_AUC',
                               mode='max',
                               save_best_only=True)
```

[17]:
```
early_stopping = EarlyStopping(monitor='val_AUC',
                               mode='max',
                              patience=5,
                              restore_best_weights=True)
```

Training stops after eight epochs and we recover the weights for the best models to find a high test AUC of 0.9346:

[18]:
```
training = rnn.fit(X_train_padded,
                   y_train,
                   batch_size=32,
                   epochs=100,
                   validation_data=(X_test_padded, y_test),
                   callbacks=[early_stopping, checkpointer],
                   verbose=1)
```

```
Epoch 1/100
782/782 [==============================] - ETA: 0s - loss: 0.4345 - accuracy:
0.7942 - AUC: 0.8801
Epoch 00001: val_AUC improved from -inf to 0.93268, saving model to
results/sentiment_imdb/lstm.h5
782/782 [==============================] - 117s 150ms/step - loss: 0.4345 -
accuracy: 0.7942 - AUC: 0.8801 - val_loss: 0.3455 - val_accuracy: 0.8501 -
val_AUC: 0.9327
Epoch 2/100
782/782 [==============================] - ETA: 0s - loss: 0.2887 - accuracy:
0.8803 - AUC: 0.9492
Epoch 00002: val_AUC improved from 0.93268 to 0.93450, saving model to
results/sentiment_imdb/lstm.h5
782/782 [==============================] - 116s 148ms/step - loss: 0.2887 -
accuracy: 0.8803 - AUC: 0.9492 - val_loss: 0.3317 - val_accuracy: 0.8568 -
```

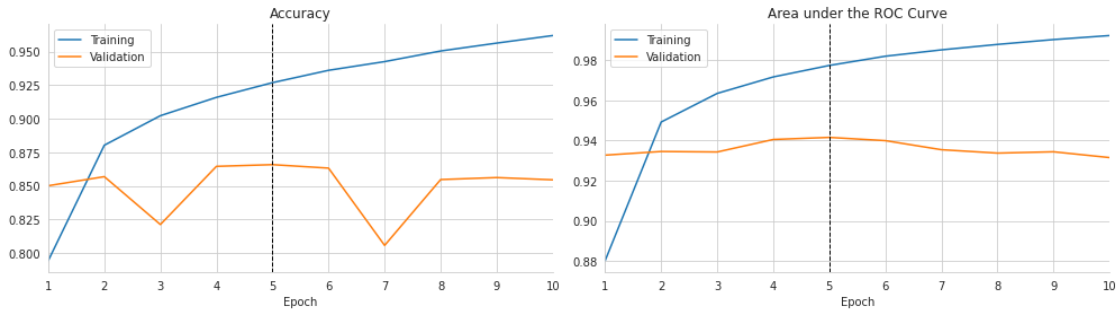
```
val_AUC: 0.9345
Epoch 3/100
782/782 [==============================] - ETA: 0s - loss: 0.2442 - accuracy:
0.9023 - AUC: 0.9634
Epoch 00003: val_AUC did not improve from 0.93450
782/782 [==============================] - 117s 149ms/step - loss: 0.2442 -
accuracy: 0.9023 - AUC: 0.9634 - val_loss: 0.4815 - val_accuracy: 0.8212 -
val_AUC: 0.9343
Epoch 4/100
782/782 [==============================] - ETA: 0s - loss: 0.2143 - accuracy:
0.9160 - AUC: 0.9716
Epoch 00004: val_AUC improved from 0.93450 to 0.94048, saving model to
results/sentiment_imdb/lstm.h5
782/782 [==============================] - 116s 148ms/step - loss: 0.2143 -
accuracy: 0.9160 - AUC: 0.9716 - val_loss: 0.3312 - val_accuracy: 0.8645 -
val_AUC: 0.9405
Epoch 5/100
782/782 [==============================] - ETA: 0s - loss: 0.1901 - accuracy:
0.9269 - AUC: 0.9774
Epoch 00005: val_AUC improved from 0.94048 to 0.94152, saving model to
results/sentiment_imdb/lstm.h5
782/782 [==============================] - 116s 148ms/step - loss: 0.1901 -
accuracy: 0.9269 - AUC: 0.9774 - val_loss: 0.3367 - val_accuracy: 0.8658 -
val_AUC: 0.9415
Epoch 6/100
782/782 [==============================] - ETA: 0s - loss: 0.1693 - accuracy:
0.9361 - AUC: 0.9819
Epoch 00006: val_AUC did not improve from 0.94152
782/782 [==============================] - 116s 148ms/step - loss: 0.1693 -
accuracy: 0.9361 - AUC: 0.9819 - val_loss: 0.3186 - val_accuracy: 0.8632 -
val_AUC: 0.9399
Epoch 7/100
782/782 [==============================] - ETA: 0s - loss: 0.1519 - accuracy:
0.9426 - AUC: 0.9851
Epoch 00007: val_AUC did not improve from 0.94152
782/782 [==============================] - 115s 147ms/step - loss: 0.1519 -
accuracy: 0.9426 - AUC: 0.9851 - val_loss: 0.5009 - val_accuracy: 0.8056 -
val_AUC: 0.9354
Epoch 8/100
782/782 [==============================] - ETA: 0s - loss: 0.1364 - accuracy:
0.9505 - AUC: 0.9878
Epoch 00008: val_AUC did not improve from 0.94152
782/782 [==============================] - 117s 150ms/step - loss: 0.1364 -
accuracy: 0.9505 - AUC: 0.9878 - val_loss: 0.3860 - val_accuracy: 0.8547 -
val_AUC: 0.9337
Epoch 9/100
782/782 [==============================] - ETA: 0s - loss: 0.1206 - accuracy:
0.9564 - AUC: 0.9902
```

```
Epoch 00009: val_AUC did not improve from 0.94152
782/782 [==============================] - 119s 152ms/step - loss: 0.1206 -
accuracy: 0.9564 - AUC: 0.9902 - val_loss: 0.3833 - val_accuracy: 0.8562 -
val_AUC: 0.9343
Epoch 10/100
782/782 [==============================] - ETA: 0s - loss: 0.1061 - accuracy:
0.9620 - AUC: 0.9922
Epoch 00010: val_AUC did not improve from 0.94152
782/782 [==============================] - 118s 151ms/step - loss: 0.1061 -
accuracy: 0.9620 - AUC: 0.9922 - val_loss: 0.3820 - val_accuracy: 0.8545 -
val_AUC: 0.9315
```

1.5 Evaluate Results

[19]:
```
history = pd.DataFrame(training.history)
history.index += 1
```

[23]:
```
fig, axes = plt.subplots(ncols=2, figsize=(14, 4))
df1 = (history[['accuracy', 'val_accuracy']]
       .rename(columns={'accuracy': 'Training',
                        'val_accuracy': 'Validation'}))
df1.plot(ax=axes[0], title='Accuracy', xlim=(1, len(history)))

axes[0].axvline(df1.Validation.idxmax(), ls='--', lw=1, c='k')

df2 = (history[['AUC', 'val_AUC']]
       .rename(columns={'AUC': 'Training',
                        'val_AUC': 'Validation'}))
df2.plot(ax=axes[1], title='Area under the ROC Curve', xlim=(1, len(history)))

axes[1].axvline(df2.Validation.idxmax(), ls='--', lw=1, c='k')

for i in [0, 1]:
    axes[i].set_xlabel('Epoch')

sns.despine()
fig.tight_layout()
fig.savefig(results_path / 'rnn_imdb_cv', dpi=300)
```



[24]:
```
y_score = rnn.predict(X_test_padded)
y_score.shape
```

[24]: (25000, 1)

[25]:
```
roc_auc_score(y_score=y_score.squeeze(), y_true=y_test)
```

[25]: 0.941730672