**Sequence Classification with LSTM Recurrent Neural Networks in Python with Keras**

The problem that you will use to demonstrate sequence learning in this tutorial is the IMDB movie review sentiment classification problem. Each movie review is a variable sequence of words, and the sentiment of each movie review must be classified.

The Large Movie Review Dataset (often referred to as the IMDB dataset) contains 25,000 highly polar movie reviews (good or bad) for training and the same amount again for testing. The problem is to determine whether a given movie review has a positive or negative sentiment.

The data was collected by [Stanford researchers and used in a 2011 paper](http://ai.stanford.edu/~amaas/papers/wvSent_acl2011.pdf) where a 50/50 split of the data was used for training and testing. An accuracy of 88.89% was achieved.

Keras provides built-in access to the IMDB dataset. The **imdb.load\_data()** function allows you to load the dataset in a format ready for use in neural networks and deep learning models.

The words have been replaced by integers that indicate the ordered frequency of each word in the dataset. The sentences in each review are therefore comprised of a sequence of integers.

### **Word Embedding**

You will map each movie review into a real vector domain, a popular technique when working with text—called word embedding. This is a technique where words are encoded as real-valued vectors in a high dimensional space, where the similarity between words in terms of meaning translates to closeness in the vector space.

Keras provides a convenient way to convert positive integer representations of words into a word embedding by an Embedding layer.

You will map each word onto a 32-length real valued vector. You will also limit the total number of words that you are interested in modeling to the 5000 most frequent words and zero out the rest. Finally, the sequence length (number of words) in each review varies, so you will constrain each review to be 500 words, truncating long reviews and padding the shorter reviews with zero values.

Now that you have defined your problem and how the data will be prepared and modeled, you are ready to develop an LSTM model to classify the sentiment of movie reviews.

## **Simple LSTM for Sequence Classification**

You can quickly develop a small LSTM for the IMDB problem and achieve good accuracy.

Let’s start by importing the classes and functions required for this model and initializing the random number generator to a constant value to ensure you can easily reproduce the results.

|  |
| --- |
|  |
| import tensorflow as tffrom tensorflow.keras.datasets import imdbfrom tensorflow.keras.models import Sequentialfrom tensorflow.keras.layers import Densefrom tensorflow.keras.layers import LSTMfrom tensorflow.keras.layers import Embeddingfrom tensorflow.keras.preprocessing import sequence# fix random seed for reproducibilitytf.random.set\_seed(7) |

You need to load the IMDB dataset. You are constraining the dataset to the top 5,000 words. You will also split the dataset into train (50%) and test (50%) sets.

|  |
| --- |
|  |
| # load the dataset but only keep the top n words, zero the resttop\_words = 5000(X\_train, y\_train), (X\_test, y\_test) = imdb.load\_data(num\_words=top\_words) |

Next, you need to truncate and pad the input sequences, so they are all the same length for modeling. The model will learn that the zero values carry no information. The sequences are not the same length in terms of content, but same-length vectors are required to perform the computation in Keras.

|  |
| --- |
|  |
| # truncate and pad input sequencesmax\_review\_length = 500X\_train = sequence.pad\_sequences(X\_train, maxlen=max\_review\_length)X\_test = sequence.pad\_sequences(X\_test, maxlen=max\_review\_length) |

You can now define, compile and fit your LSTM model.

The first layer is the Embedded layer that uses 32-length vectors to represent each word. The next layer is the LSTM layer with 100 memory units (smart neurons). Finally, because this is a classification problem, you will use a Dense output layer with a single neuron and a sigmoid activation function to make 0 or 1 predictions for the two classes (good and bad) in the problem.

Because it is a binary classification problem, log loss is used as the loss function (**binary\_crossentropy** in Keras). The efficient ADAM optimization algorithm is used. The model is fit for only two epochs because it quickly overfits the problem. A large batch size of 64 reviews is used to space out weight updates.

|  |
| --- |
|  |
| # create the modelembedding\_vecor\_length = 32model = Sequential()model.add(Embedding(top\_words, embedding\_vecor\_length, input\_length=max\_review\_length))model.add(LSTM(100))model.add(Dense(1, activation='sigmoid'))model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])print(model.summary())model.fit(X\_train, y\_train, validation\_data=(X\_test, y\_test), epochs=3, batch\_size=64) |

Once fit, you can estimate the performance of the model on unseen reviews.

|  |
| --- |
|  |
| # Final evaluation of the modelscores = model.evaluate(X\_test, y\_test, verbose=0)print("Accuracy: %.2f%%" % (scores[1]\*100)) |

For completeness, here is the full code listing for this LSTM network on the IMDB dataset.

|  |
| --- |
|  |
| # LSTM for sequence classification in the IMDB datasetimport tensorflow as tffrom tensorflow.keras.datasets import imdbfrom tensorflow.keras.models import Sequentialfrom tensorflow.keras.layers import Densefrom tensorflow.keras.layers import LSTMfrom tensorflow.keras.layers import Embeddingfrom tensorflow.keras.preprocessing import sequence# fix random seed for reproducibilitytf.random.set\_seed(7)# load the dataset but only keep the top n words, zero the resttop\_words = 5000(X\_train, y\_train), (X\_test, y\_test) = imdb.load\_data(num\_words=top\_words)# truncate and pad input sequencesmax\_review\_length = 500X\_train = sequence.pad\_sequences(X\_train, maxlen=max\_review\_length)X\_test = sequence.pad\_sequences(X\_test, maxlen=max\_review\_length)# create the modelembedding\_vecor\_length = 32model = Sequential()model.add(Embedding(top\_words, embedding\_vecor\_length, input\_length=max\_review\_length))model.add(LSTM(100))model.add(Dense(1, activation='sigmoid'))model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])print(model.summary())model.fit(X\_train, y\_train, epochs=3, batch\_size=64)# Final evaluation of the modelscores = model.evaluate(X\_test, y\_test, verbose=0)print("Accuracy: %.2f%%" % (scores[1]\*100)) |

**Note**: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running this example produces the following output.

|  |
| --- |
|  |
| Epoch 1/3391/391 [==============================] - 124s 316ms/step - loss: 0.4525 - accuracy: 0.7794Epoch 2/3391/391 [==============================] - 124s 318ms/step - loss: 0.3117 - accuracy: 0.8706Epoch 3/3391/391 [==============================] - 126s 323ms/step - loss: 0.2526 - accuracy: 0.9003Accuracy: 86.83% |

You can see that this simple LSTM with little tuning achieves near state-of-the-art results on the IMDB problem. Importantly, this is a template that you can use to apply LSTM networks to your own sequence classification problems.

Now, let’s look at some extensions of this simple model that you may also want to bring to your own problems

## **LSTM for Sequence Classification with Dropout**

Recurrent neural networks like LSTM generally have the problem of overfitting.

Dropout can be applied between layers using the Dropout Keras layer. You can do this easily by adding new Dropout layers between the Embedding and LSTM layers and the LSTM and Dense output layers. For example:

|  |
| --- |
|  |
| model = Sequential()model.add(Embedding(top\_words, embedding\_vecor\_length, input\_length=max\_review\_length))model.add(Dropout(0.2))model.add(LSTM(100))model.add(Dropout(0.2))model.add(Dense(1, activation='sigmoid')) |

The full code listing example above with the addition of Dropout layers is as follows:

|  |
| --- |
|  |
| # LSTM with Dropout for sequence classification in the IMDB datasetimport tensorflow as tffrom tensorflow.keras.datasets import imdbfrom tensorflow.keras.models import Sequentialfrom tensorflow.keras.layers import Densefrom tensorflow.keras.layers import LSTMfrom tensorflow.keras.layers import Dropoutfrom tensorflow.keras.layers import Embeddingfrom tensorflow.keras.preprocessing import sequence# fix random seed for reproducibilitytf.random.set\_seed(7)# load the dataset but only keep the top n words, zero the resttop\_words = 5000(X\_train, y\_train), (X\_test, y\_test) = imdb.load\_data(num\_words=top\_words)# truncate and pad input sequencesmax\_review\_length = 500X\_train = sequence.pad\_sequences(X\_train, maxlen=max\_review\_length)X\_test = sequence.pad\_sequences(X\_test, maxlen=max\_review\_length)# create the modelembedding\_vecor\_length = 32model = Sequential()model.add(Embedding(top\_words, embedding\_vecor\_length, input\_length=max\_review\_length))model.add(Dropout(0.2))model.add(LSTM(100))model.add(Dropout(0.2))model.add(Dense(1, activation='sigmoid'))model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])print(model.summary())model.fit(X\_train, y\_train, epochs=3, batch\_size=64)# Final evaluation of the modelscores = model.evaluate(X\_test, y\_test, verbose=0)print("Accuracy: %.2f%%" % (scores[1]\*100)) |

**Note**: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running this example provides the following output.

|  |
| --- |
|  |
| Epoch 1/3391/391 [==============================] - 117s 297ms/step - loss: 0.4721 - accuracy: 0.7664Epoch 2/3391/391 [==============================] - 125s 319ms/step - loss: 0.2840 - accuracy: 0.8864Epoch 3/3391/391 [==============================] - 135s 346ms/step - loss: 0.3022 - accuracy: 0.8772Accuracy: 85.66% |

You can see dropout having the desired impact on training with a slightly slower trend in convergence and, in this case, a lower final accuracy. The model could probably use a few more epochs of training and may achieve a higher skill (try it and see).

Alternately, dropout can be applied to the input and recurrent connections of the memory units with the LSTM precisely and separately.

Keras provides this capability with parameters on the LSTM layer, the **dropout** for configuring the input dropout, and **recurrent\_dropout** for configuring the recurrent dropout. For example, you can modify the first example to add dropout to the input and recurrent connections as follows:

|  |
| --- |
|  |
| model = Sequential()model.add(Embedding(top\_words, embedding\_vecor\_length, input\_length=max\_review\_length))model.add(LSTM(100, dropout=0.2, recurrent\_dropout=0.2))model.add(Dense(1, activation='sigmoid')) |

The full code listing with more precise LSTM dropout is listed below for completeness.

|  |
| --- |
|  |
| # LSTM with dropout for sequence classification in the IMDB datasetimport tensorflow as tffrom tensorflow.keras.datasets import imdbfrom tensorflow.keras.models import Sequentialfrom tensorflow.keras.layers import Densefrom tensorflow.keras.layers import LSTMfrom tensorflow.keras.layers import Embeddingfrom tensorflow.keras.preprocessing import sequence# fix random seed for reproducibilitytf.random.set\_seed(7)# load the dataset but only keep the top n words, zero the resttop\_words = 5000(X\_train, y\_train), (X\_test, y\_test) = imdb.load\_data(num\_words=top\_words)# truncate and pad input sequencesmax\_review\_length = 500X\_train = sequence.pad\_sequences(X\_train, maxlen=max\_review\_length)X\_test = sequence.pad\_sequences(X\_test, maxlen=max\_review\_length)# create the modelembedding\_vecor\_length = 32model = Sequential()model.add(Embedding(top\_words, embedding\_vecor\_length, input\_length=max\_review\_length))model.add(LSTM(100, dropout=0.2, recurrent\_dropout=0.2))model.add(Dense(1, activation='sigmoid'))model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])print(model.summary())model.fit(X\_train, y\_train, epochs=3, batch\_size=64)# Final evaluation of the modelscores = model.evaluate(X\_test, y\_test, verbose=0)print("Accuracy: %.2f%%" % (scores[1]\*100)) |

**Note**: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running this example provides the following output.

|  |
| --- |
|  |
| Epoch 1/3391/391 [==============================] - 220s 560ms/step - loss: 0.4605 - accuracy: 0.7784Epoch 2/3391/391 [==============================] - 219s 560ms/step - loss: 0.3158 - accuracy: 0.8773Epoch 3/3391/391 [==============================] - 219s 559ms/step - loss: 0.2734 - accuracy: 0.8930Accuracy: 86.78% |

You can see that the LSTM-specific dropout has a more pronounced effect on the convergence of the network than the layer-wise dropout. Like above, the number of epochs was kept constant and could be increased to see if the skill of the model could be further lifted.

Dropout is a powerful technique for combating overfitting in your LSTM models, and it is a good idea to try both methods. Still, you may get better results with the gate-specific dropout provided in Keras.

## **Bidirectional LSTM for Sequence Classification**

Sometimes, a sequence is better used in reversed order. In those cases, you can simply reverse a vector x using the Python syntax x[::-1] before using it to train your LSTM network.

Sometimes, neither the forward nor the reversed order works perfectly, but combining them will give better results. In this case, you will need a **bidirectional LSTM network**.

A bidirectional LSTM network is simply two separate LSTM networks; one feeds with a forward sequence and another with reversed sequence. Then the output of the two LSTM networks is concatenated together before being fed to the subsequent layers of the network. In Keras, you have the function Bidirectional() to clone an LSTM layer for forward-backward input and concatenate their output. For example,

|  |
| --- |
|  |
| model = Sequential()model.add(Embedding(top\_words, embedding\_vecor\_length, input\_length=max\_review\_length))model.add(Bidirectional(LSTM(100, dropout=0.2, recurrent\_dropout=0.2)))model.add(Dense(1, activation='sigmoid')) |

Since you created not one, but two LSTMs with 100 units each, this network will take twice the amount of time to train. Depending on the problem, this additional cost may be justified.

The full code listing with adding the bidirectional LSTM to the last example is listed below for completeness.

|  |
| --- |
|  |
| # LSTM with dropout for sequence classification in the IMDB datasetimport tensorflow as tffrom tensorflow.keras.datasets import imdbfrom tensorflow.keras.models import Sequentialfrom tensorflow.keras.layers import Densefrom tensorflow.keras.layers import LSTMfrom tensorflow.keras.layers import Bidirectionalfrom tensorflow.keras.layers import Embeddingfrom tensorflow.keras.preprocessing import sequence# fix random seed for reproducibilitytf.random.set\_seed(7)# load the dataset but only keep the top n words, zero the resttop\_words = 5000(X\_train, y\_train), (X\_test, y\_test) = imdb.load\_data(num\_words=top\_words)# truncate and pad input sequencesmax\_review\_length = 500X\_train = sequence.pad\_sequences(X\_train, maxlen=max\_review\_length)X\_test = sequence.pad\_sequences(X\_test, maxlen=max\_review\_length)# create the modelembedding\_vecor\_length = 32model = Sequential()model.add(Embedding(top\_words, embedding\_vecor\_length, input\_length=max\_review\_length))model.add(Bidirectional(LSTM(100, dropout=0.2, recurrent\_dropout=0.2)))model.add(Dense(1, activation='sigmoid'))model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])print(model.summary())model.fit(X\_train, y\_train, epochs=3, batch\_size=64)# Final evaluation of the modelscores = model.evaluate(X\_test, y\_test, verbose=0)print("Accuracy: %.2f%%" % (scores[1]\*100)) |

**Note**: Your [results may vary](https://machinelearningmastery.com/different-results-each-time-in-machine-learning/) given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running this example provides the following output.

|  |
| --- |
|  |
| Epoch 1/3391/391 [==============================] - 405s 1s/step - loss: 0.4960 - accuracy: 0.7532Epoch 2/3391/391 [==============================] - 439s 1s/step - loss: 0.3075 - accuracy: 0.8744Epoch 3/3391/391 [==============================] - 430s 1s/step - loss: 0.2551 - accuracy: 0.9014Accuracy: 87.69% |

It seems you can only get a slight improvement but with a significantly longer training time.

## **LSTM and Convolutional Neural Network for Sequence Classification**

Convolutional neural networks excel at learning the spatial structure in input data.

The IMDB review data does have a one-dimensional spatial structure in the sequence of words in reviews, and the CNN may be able to pick out invariant features for the good and bad sentiment. This learned spatial feature may then be learned as sequences by an LSTM layer.

You can easily add a one-dimensional CNN and [max pooling layers](https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/) after the Embedding layer, which then feeds the consolidated features to the LSTM. You can use a smallish set of 32 features with a small filter length of 3. The pooling layer can use the standard length of 2 to halve the feature map size.

For example, you would create the model as follows:

|  |
| --- |
|  |
| model = Sequential()model.add(Embedding(top\_words, embedding\_vecor\_length, input\_length=max\_review\_length))model.add(Conv1D(filters=32, kernel\_size=3, padding='same', activation='relu'))model.add(MaxPooling1D(pool\_size=2))model.add(LSTM(100))model.add(Dense(1, activation='sigmoid')) |

The full code listing with CNN and LSTM layers is listed below for completeness.

|  |
| --- |
|  |
| # LSTM and CNN for sequence classification in the IMDB datasetimport tensorflow as tffrom tensorflow.keras.datasets import imdbfrom tensorflow.keras.models import Sequentialfrom tensorflow.keras.layers import Densefrom tensorflow.keras.layers import LSTMfrom tensorflow.keras.layers import Conv1Dfrom tensorflow.keras.layers import MaxPooling1Dfrom tensorflow.keras.layers import Embeddingfrom tensorflow.keras.preprocessing import sequence# fix random seed for reproducibilitytf.random.set\_seed(7)# load the dataset but only keep the top n words, zero the resttop\_words = 5000(X\_train, y\_train), (X\_test, y\_test) = imdb.load\_data(num\_words=top\_words)# truncate and pad input sequencesmax\_review\_length = 500X\_train = sequence.pad\_sequences(X\_train, maxlen=max\_review\_length)X\_test = sequence.pad\_sequences(X\_test, maxlen=max\_review\_length)# create the modelembedding\_vecor\_length = 32model = Sequential()model.add(Embedding(top\_words, embedding\_vecor\_length, input\_length=max\_review\_length))model.add(Conv1D(filters=32, kernel\_size=3, padding='same', activation='relu'))model.add(MaxPooling1D(pool\_size=2))model.add(LSTM(100))model.add(Dense(1, activation='sigmoid'))model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])print(model.summary())model.fit(X\_train, y\_train, epochs=3, batch\_size=64)# Final evaluation of the modelscores = model.evaluate(X\_test, y\_test, verbose=0)print("Accuracy: %.2f%%" % (scores[1]\*100)) |

**Note**: Your [results may vary](https://machinelearningmastery.com/different-results-each-time-in-machine-learning/) given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running this example provides the following output.

|  |
| --- |
|  |
| Epoch 1/3391/391 [==============================] - 65s 163ms/step - loss: 0.4213 - accuracy: 0.7950Epoch 2/3391/391 [==============================] - 66s 168ms/step - loss: 0.2490 - accuracy: 0.9026Epoch 3/3391/391 [==============================] - 73s 188ms/step - loss: 0.1979 - accuracy: 0.9261Accuracy: 88.45% |

You can see that you achieve slightly better results than the first example, although with fewer weights and faster training time.

You might expect that even better results could be achieved if this example was further extended to use dropout.