Chapter 5: Deep Learning for Sequential Data

This chapter introduces recurrent neural networks, starting with the basic model and moving on to newer recurrent layers that can handle internal memory learning to remember, or forget, certain patterns found in datasets. We will begin by showing that recurrent networks are powerful in the case of inferring patterns that are temporal or sequential, and then we will introduce an improvement on the traditional paradigm for a model that has internal memory, which can be applied in both directions in the temporal space.

We will approach the learning task by looking at a sentiment analysis problem as a sequence-to-vector application, and then we will focus on an autoencoder as a vector-to-sequence and sequence-to-sequence model at the same time. By the end of this chapter, you will be able to explain why a long short-term memory model is better than the traditional dense approach. You will be able to describe how a bi-directional long short-term memory model might represent an advantage over the single directional approach. You will be able to implement your own recurrent networks and apply them to NLP problems or to image-related applications, including sequence-to-vector, vector-to-sequence, and sequence-to-sequence modeling.

5.1 Introduction to Recurrent Neural Networks

Recurrent neural networks (RNNs) are based on the early work of Rumelhart (Rumelhart, D. E., et al. (1986)), who was a psychologist who worked closely with Hinton, whom we have already mentioned here several times. The concept is simple, but revolutionary in the area of pattern recognition that uses sequences of data. The concept of recurrence in RNNs can be illustrated as shown in the following diagram. If you think of a dense layer of neural units, these can be stimulated using some input at different time steps, *t*. Figures 5.1 (b) and (c) show an RNN with five time steps, *t=5*. We can see in Figures 5.1 (b) and (c) how the input is accessible to the different time steps, but more importantly, the output of the neural units is also available to the next layer of neurons. The ability of an RNN to see how the previous layer of neurons is stimulated helps the network to interpret sequences much better than without that additional piece of information. However, this comes at a cost: there will be more parameters to be calculated in comparison to a traditional dense layer due to the fact that there are weights associated with the input x_t and the previous output o_{t-1}. In Keras, we can create a simple RNN with five time steps and 10 neural units (see Figure 5.1) as follows:

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import SimpleRNN

n_units = 10
t_steps = 5
```

```
inpt_ftrs=2
model = Sequential()
model.add(SimpleRNN(n_units, input_shape=(t_steps, inpt_ftrs)))
model.summary()
```

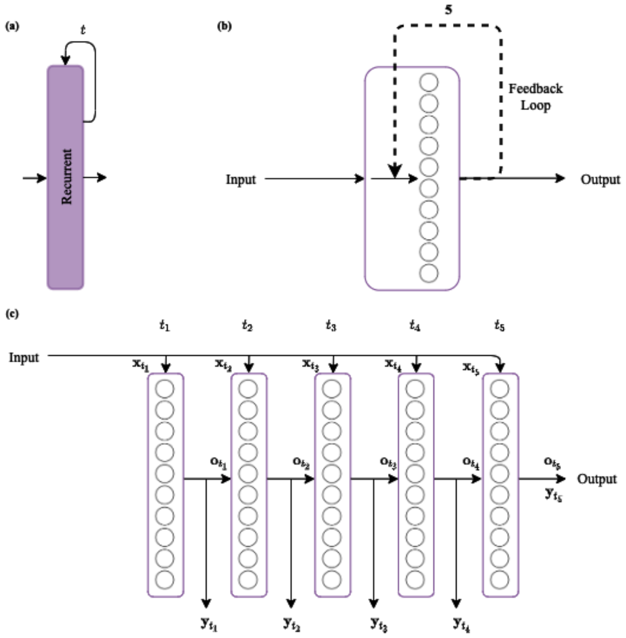

Figure 5.1. Different representations of recurrent layers: (a) will be the preferred use in this book; (b) depicts the neural units and the feedback loop; and (c) is the expanded version of (b), showing what really happens during training

This gives the following summary:

```
Model: "sequential"
_________________________________________________________________
Layer (type)          Output Shape  Param #
=================================================================
simple_rnn (SimpleRNN)  (None, 10)    130
=================================================================
Total params: 130
Trainable params: 130
Non-trainable params: 0
```

The preceding sample code assumes that the number of **features in the input** would be just **two**; for example, we can have sequential data in two dimensions. These types of RNNs are called simple because they resemble the simplicity of dense networks with tanh activations and a recurrence aspect to it.

RNNs are usually tied to embedding layers, which we discuss next.

5.1.1 Embedding layers

An embedding layer is usually paired with RNNs when there are sequences that require additional processing in order to make RNNs more robust. Consider the case when you have the sentence "This is a small vector", and you want to train an RNN to detect when sentences are correctly written or poorly written. You can train an RNN with all the sentences of length five that you can think of, including "This is a small vector". For this, you will have to figure out a way to transform a sentence into something that the RNN can understand. Embedding layers come to the rescue.

There is a technique called word embedding, which is tasked with converting a word into a vector. There are several successful approaches out there, such as Word2Vec (Mikolov, T., et al. (2013)) or GloVe (Pennington, J., et al. (2014)). However, we will focus on a simple technique that is readily available. We will do this in steps:

1. Determine the length of the sentences you want to work on. This will become the dimensionality of the input for the RNN layer. This step is not necessary for the design of the embedding layer, but you will need it for the RNN layer very soon, and it is important that you decide this early on.
2. Determine the number of different words in your dataset and assign a number to them, creating a dictionary: word-to-index. This is known as a vocabulary.
3. Substitute the words in all the sentences of the dataset with their corresponding index.
4. Determine the dimensionality of the word embedding and train an embedding layer to map from the numerical index into a real-valued vector with the desired dimensions.

Note:

Most people will determine the vocabulary and then calculate the frequency of each word to rank the words in the vocabulary so as to have the index 0 corresponding to the most common word in the dataset, and the last index corresponding to the most uncommon word. This can be helpful if you want to ignore the most common words or the most uncommon words, for example.

Look at the example in Figure 5.2. If we take the word This, whose given index is 7, some trained embedding layer can map that number into a vector of size 10, as you can see in Figure 5.2 (b). That is the word embedding process. You can repeat this process for the complete sentence "This is a small vector", which can be mapped to a sequence of indices [7, 0, 6, 1, 28], and it will produce for you a sequence of vectors; see Figure 5.2 (c). In other words, it will produce a sequence of word embeddings. The RNN can easily process these sequences and determine whether the sentence that these sequences represent is a correct sentence. However, we must say that determining whether a sentence is correct is a challenging and interesting problem (Rivas, P. et al. (2019)):

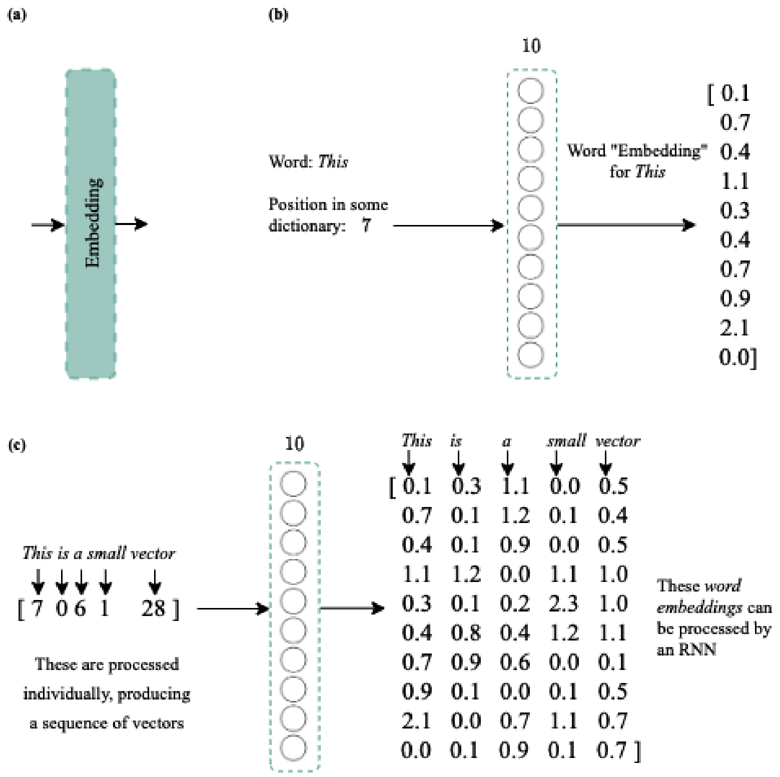

Figure 5.2. Embedding layer: (a) will be the preferred use in this book; (b) shows an example of a word embedding; and (c) shows a sequence of words and its corresponding matrix of word embeddings

Based on the model shown in Figure 5.2, an embedding layer in Keras can be created as follows:

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Embedding

vocab_size = 30
embddng_dim = 10
seqnc_lngth = 5

model = Sequential()
model.add(Embedding(vocab_size, embddng_dim, input_length=seqnc_lngth))
model.summary()
```

— hyper length

This produces the following summary:

```
Model: "sequential"
_________________________________________________________________
Layer (type)            Output Shape      Param #
=================================================================
embedding (Embedding)    (None, 5, 10)    300
=================================================================
Total params: 300
Trainable params: 300
Non-trainable params: 0
```

Note, however, that the vocabulary size is usually in the order of thousands for typical NLP tasks in most common languages. Just think of your good old-fashioned dictionary ... How many entries does it have? Several thousand, usually.

Similarly, sentences are usually longer than five words, so you should expect to have longer sequences than in the preceding example.

Finally, the embedding dimension depends on how rich you want your model to be in the embedding space, or on your model space constraints. If you want a smaller model, consider having embeddings of 50 dimensions for example. But if space is not a problem and you have an excellent dataset with millions of entries, and you have unlimited GPU power, you should try embedding dimensions of 500, 700, or even 1000+ dimensions.

Now, let's try to put the pieces together with a real-life example.

5.1.2 Word embedding and RNNs on IMDb

The IMDb dataset was explained in previous chapters, but to keep things brief, we will say that it has movie reviews based on text and a positive (1) or negative (0) review associated with every entry. Keras lets you have access to this dataset and gives a couple of nice features to optimize time when designing a model. For example, the dataset is already processed according to the frequency of each word such that the smallest index is associated with frequent words and vice versa. With this in mind, you can also exclude the most common words in the English language, say 10 or 20. And you can even limit the size of the vocabulary to, say, 5,000 or 10,000 words.

Before we go further, we will have to justify some things you are about see:

- A vocabulary size of 10,000. We can make an argument in favor of keeping a vocabulary size of 10,000 since the task here is to determine whether a review is positive or negative. That is, we do not need an overly complex vocabulary to determine this.
- Eliminating the top 20 words. The most common words in English include words such as "a" or "the"; words like these are probably not very important in determining whether a movie review is positive or negative. So, eliminating the 20 most common should be OK.
- Sentence length of 128 words. Having smaller sentences, such as 5-word sentences, might be lacking enough content, and it would not make a lot of sense having longer sentences, such as 300-word sentences, since we can probably sense the tone of a review in fewer words than that. The choice of 128 words is completely arbitrary, but justified in the sense explained.

With such considerations, we can easily load the dataset as follows:

```
from keras.datasets import imdb
from keras.preprocessing import sequence

inpt_dim = 128
index_from = 3

(x_train, y_train),(x_test, y_test)=imdb.load_data(num_words=10000,
                                                   start_char=1,
                                                   oov_char=2,
                                                   index_from=index_from,
                                                   skip_top=20)
x_train = sequence.pad_sequences(x_train,
                               maxlen=inpt_dim).astype('float32')
x_test = sequence.pad_sequences(x_test,
maxlen=inpt_dim).astype('float32')

# let's print the shapes
print('x_train shape:', x_train.shape)
print('x_test shape:', x_test.shape)
```

We can also print some data for verification purposes like this:

```
# let's print the indices of sample #7
print(' '.join(str(int(id)) for id in x_train[7]))

# let's print the actual words of sample #7
wrd2id = imdb.get_word_index()
wrd2id = {k:(v+index_from) for k,v in wrd2id.items()}
wrd2id["<PAD>"] = 0
wrd2id["<START>"] = 1
wrd2id["<UNK>"] = 2
wrd2id["<UNUSED>"] = 3

id2wrd = {value:key for key,value in wrd2id.items()}
print(' '.join(id2wrd[id] for id in x_train[7] ))
```

This will output the following:

```
x_train shape: (25000, 128)
x_test shape: (25000, 128)

 55   655    707    6371    956    225    1456    841     42 1310   225
2 ...
very middle class suburban setting there's zero atmosphere or mood
there's <UNK> ...
```

The first part of the preceding code shows how to load the dataset split into training and test sets, x_train and y_train, x_test and y_test, respectively. The remaining part is simply to display the shape of the dataset (dimensionality) for purposes of verification, and also for verification, we can print out sample #7 in its original form (the indices) and also its corresponding word. Such a portion of the code is a little bit strange if you have not used IMDb before. But the major points are that we need to reserve certain indices for special tokens: beginning of the sentence <START>, unused index <UNUSED>, unknown word index <UNK>, and zero padding index <PAD>. One we have made a special allocation for these tokens, we can easily map from the indices back to words. These indices will be learned by the RNN, and it will learn how to handle them, either by ignoring those, or by giving specific weights to them.

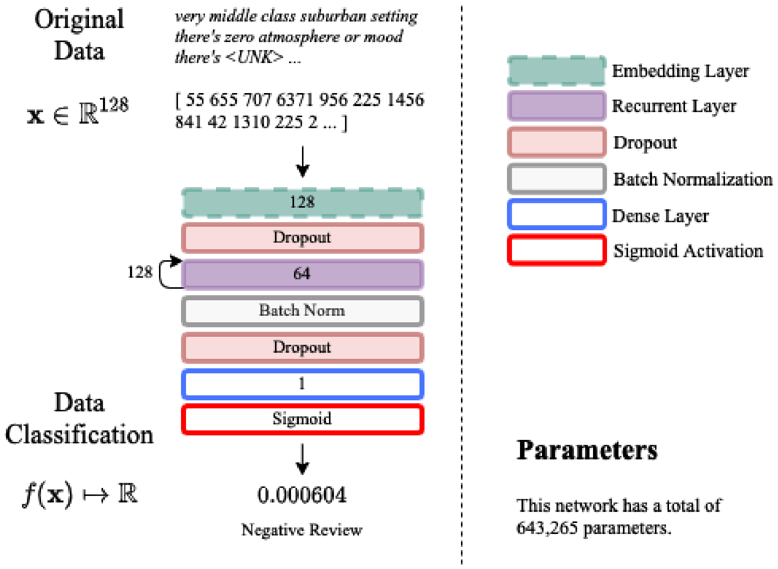

Figure 5.3. An RNN architecture for the IMDb dataset

The diagram shows the same example (#7 from the training set) that is associated with a negative review. The architecture depicted in the diagram along with the code that loads the data is the following:

```
from keras.datasets import imdb
from keras.preprocessing import sequence
from tensorflow.keras.models import Model
from tensorflow.keras.layers import SimpleRNN, Embedding,
BatchNormalization
from tensorflow.keras.layers import Dense, Activation, Input, Dropout

seqnc_lngth = 128
embddng_dim = 64
vocab_size = 10000

(x_train, y_train), (x_test, y_test) =
imdb.load_data(num_words=vocab_size,
                                             skip_top=20)
x_train = sequence.pad_sequences(x_train,
                           maxlen=seqnc_lngth).astype('float32')
x_test = sequence.pad_sequences(x_test,
                          maxlen=seqnc_lngth).astype('float32')
```

The layers of the model are defined as follows:

```
inpt_vec = Input(shape=(seqnc_lngth,))
l1 = Embedding(vocab_size, embddng_dim,
input_length=seqnc_lngth)(inpt_vec)
l2 = Dropout(0.3)(l1)
l3 = SimpleRNN(32)(l2)
l4 = BatchNormalization()(l3)
l5 = Dropout(0.2)(l4)
output = Dense(1, activation='sigmoid')(l5)

rnn = Model(inpt_vec, output)

rnn.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
rnn.summary()
```

This model uses the standard loss and optimizer that we have used before, and the summary produced is the following:

```
Model: "functional"
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
input_1 (InputLayer) [(None, 128)] 0
_________________________________________________________________
embedding (Embedding) (None, 128, 64) 640000
_________________________________________________________________
dropout_1 (Dropout) (None, 128, 64) 0
_________________________________________________________________
simple_rnn (SimpleRNN) (None, 32) 3104
_________________________________________________________________
batch_normalization (BatchNo (None, 32) 128
_________________________________________________________________
dropout_2 (Dropout) (None, 32) 0
_________________________________________________________________
dense (Dense) (None, 1) 33
=================================================================
Total params: 643,265
Trainable params: 643,201
Non-trainable params: 64
```

Then we can train the network using the callbacks that we have used before: a) early stopping, and b) automatic learning rate reduction. The learning can be executed as follows:

```
from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping
import matplotlib.pyplot as plt

#callbacks
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5,
patience=3,
min_delta=1e-4, mode='min', verbose=1)

stop_alg = EarlyStopping(monitor='val_loss', patience=7,
restore_best_weights=True, verbose=1)
```

```
#training
hist = rnn.fit(x_train, y_train, batch_size=100, epochs=1000,
callbacks=[stop_alg, reduce_lr], shuffle=True,
validation_data=(x_test, y_test))
```

Then we save the model and display the loss like so:

```
# save and plot training process
rnn.save_weights("rnn.hdf5")

fig = plt.figure(figsize=(10,6))
plt.plot(hist.history['loss'], color='#785ef0')
plt.plot(hist.history['val_loss'], color='#dc267f')
plt.title('Model Loss Progress')
plt.ylabel('Brinary Cross-Entropy Loss')
plt.xlabel('Epoch')
plt.legend(['Training Set', 'Test Set'], loc='upper right')
plt.show()
```

The preceding code produces the plot shown in the following diagram, which indicates that the network starts to overfit after epoch #3:

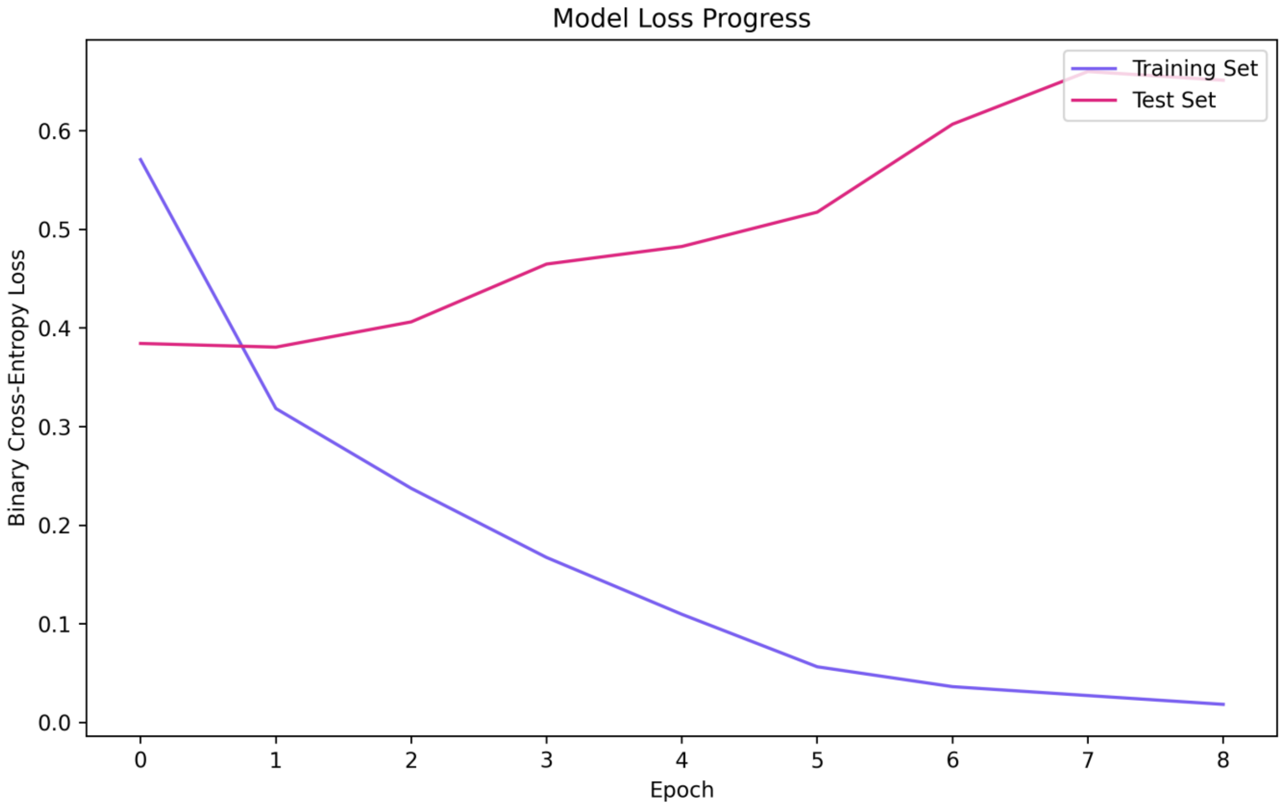

Figure 5.4. RNN loss during training.

Overfitting is quite common in recurrent networks, and you should not be surprised by this behavior. As of today, with the current algorithms, this happens a lot. However, one interesting fact about RNNs is that they also converge really fast compared to other traditional models. As you can see, convergence after three epochs is not too bad.

```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import balanced_accuracy_score
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt
import numpy as np

y_hat = rnn.predict(x_test)

# gets the ROC
fpr, tpr, thresholds = roc_curve(y_test, y_hat)
roc_auc = auc(fpr, tpr)

# plots ROC
fig = plt.figure(figsize=(10,6))
plt.plot(fpr, tpr, color='#785ef0',
label='ROC curve (AUC = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='#dc267f', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic Curve')
plt.legend(loc="lower right")
plt.show()

# finds optimal threshold and gets ACC and CM
optimal_idx = np.argmax(tpr - fpr)
optimal_threshold = thresholds[optimal_idx]
print("Threshold value is:", optimal_threshold)
y_pred = np.where(y_hat>=optimal_threshold, 1, 0)
print(balanced_accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
```

First, let's analyze the plot produced here, which is shown in Figure 5.5:

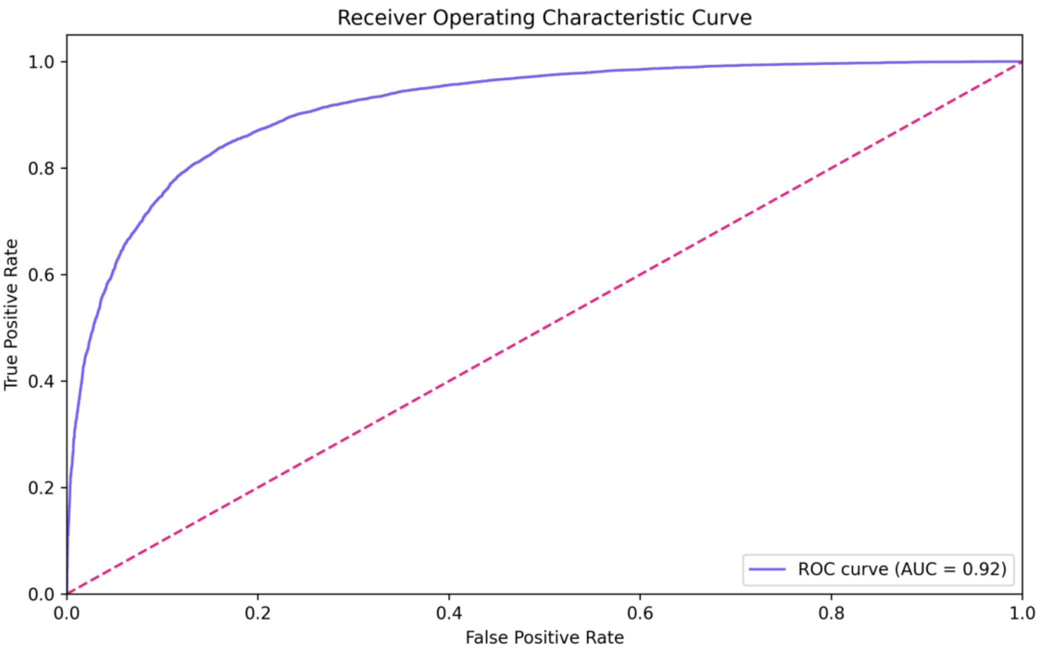

Figure 5.5. ROC and AUC of the RNN model calculated in the test set

The diagram shows a good combination of True Positive Rates (TPR) and False Positive Rates (FPR), although it is not ideal: we would like to see a sharper step-like curve. The AUC is 0.92, which again is good, but the ideal would be an AUC of 1.0.

Similarly, the code produces the balanced accuracy and confusion matrix, which would look something like this:

```
Threshold value is: 0.81700134

0.8382000000000001

[[10273 2227]
[ 1818 10682]]
```

First of all, we calculate here the optimal threshold value as a function of the TPR and FPR. We want to choose the threshold that will give us the maximum TPR and minimum FPR. The threshold and results shown here will vary depending on the initial state of the network; however, the accuracy should typically be around a very similar value.

Once the optimal threshold is calculated, we can use NumPy's `np.where()` method to threshold the entire predictions, mapping them to `{0, 1}`. After this, the balanced accuracy is calculated to be 83.82%, which again is not too bad, but also not ideal.

One of the possible ways to improve on the RNN model shown in Figure 5.3 would be to somehow give the recurrent layer the ability to remember or forget specific words across layers and have them continue to stimulate neural units across the sequence. The next section will introduce a type of RNN with such capabilities.

5.2 Long short-term memory (LSTM) models

Long Short-Term Memory Models (LSTMs) gained traction as an improved version of recurrent models since 1997. LSTMs promised to alleviate the following problems associated with traditional RNNs:

- Vanishing gradients
- Exploding gradients
- The inability to remember or forget certain aspects of the input sequences

The following diagram shows a very simplified version of an LSTM. In (b), we can see the additional self-loop that is attached to some memory, and in (c), we can observe what the network looks like when unfolded or expanded:

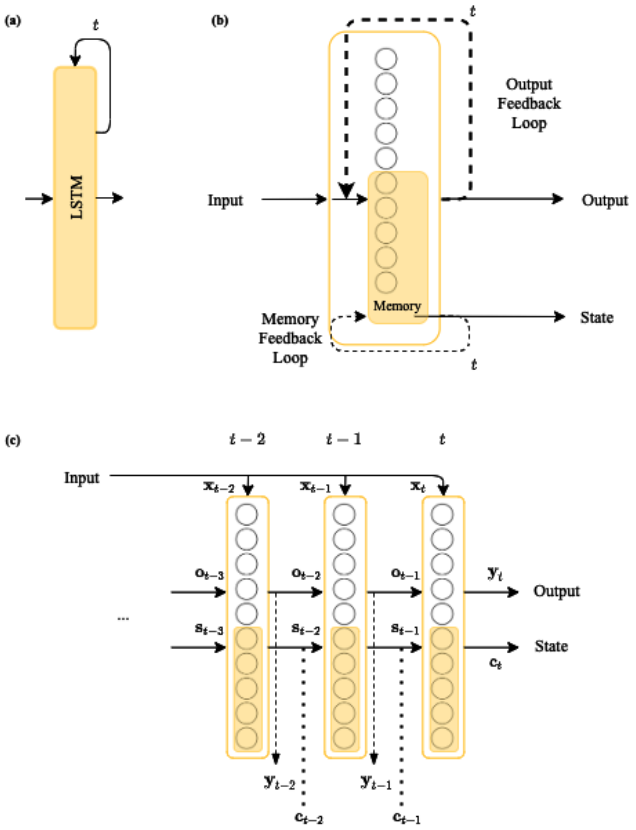

Figure 5.6. Simplified representation of an LSTM

There is much more to the model, but the most essential elements are shown in Figure 5.6. Observe how an LSTM layer receives from the previous time step not only the previous output, but also something called state, which acts as a type of memory. In the diagram, you can see that while the current output and state are available to the next layer, these are also available to use at any point if they are needed.

Some of the things that we are not showing in Figure 5.6 include the mechanisms by which the LSTM remembers or forgets. These can be complex to explain in this lecture for beginners. However, all you need to know at this point is that there are three major mechanisms:

- **Output control**: How much an output neuron is stimulated by the previous output and the current state
- **Memory control**: How much of the previous state will be forgotten in the current state
- **Input control**: How much of the previous output and new state (memory) will be considered to determine the new current state

These mechanisms are trainable and optimized for each and every single dataset of sequences. But to show the advantages of using an LSTM as our recurrent layer, we will repeat the exact same code as before, only changing the RNN by an LSTM.

The code to load the dataset and build the model is the following:

```
from keras.datasets import imdb
from keras.preprocessing import sequence
from tensorflow.keras.models import Model
from tensorflow.keras.layers import LSTM, Embedding, BatchNormalization
from tensorflow.keras.layers import Dense, Activation, Input, Dropout

seqnc_lngth = 128
embddng_dim = 64
vocab_size = 10000

(x_train, y_train), (x_test, y_test) =
imdb.load_data(num_words=vocab_size,skip_top=20)
x_train =
sequence.pad_sequences(x_train,maxlen=seqnc_lngth).astype('float32')
x_test =
sequence.pad_sequences(x_test,maxlen=seqnc_lngth).astype('float32')
```

The model can be specified as follows:

```
inpt_vec = Input(shape=(seqnc_lngth,))
l1 = Embedding(vocab_size, embddng_dim,
input_length=seqnc_lngth)(inpt_vec)
l2 = Dropout(0.3)(l1)
l3 = LSTM(32)(l2)
l4 = BatchNormalization()(l3)
l5 = Dropout(0.2)(l4)
output = Dense(1, activation='sigmoid')(l5)

lstm = Model(inpt_vec, output)

lstm.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
lstm.summary()
```

This produces the following output:

```
Model: "functional"
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
input (InputLayer) [(None, 128)] 0
_________________________________________________________________
embedding (Embedding) (None, 128, 64) 640000
_________________________________________________________________
dropout_1 (Dropout) (None, 128, 64) 0
_________________________________________________________________
lstm (LSTM) (None, 32) 12416
_________________________________________________________________
batch_normalization (Batch (None, 32) 128
_________________________________________________________________
dropout_2 (Dropout) (None, 32) 0
_________________________________________________________________
dense (Dense) (None, 1) 33
=================================================================
Total params: 652,577
Trainable params: 652,513
Non-trainable params: 64
```

This essentially replicates the model shown in the following diagram:

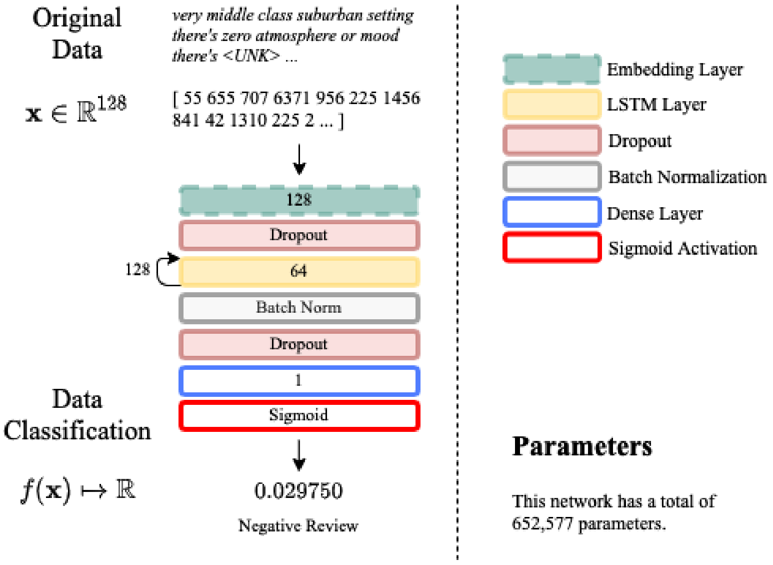

Figure 5.7. LSTM-based neural architecture for the IMDb dataset

Notice that this model has nearly 10,000 more parameters than the simple RNN approach. However, the premise is that this increase in parameters should also result in an increase in performance.

We then train our model the same as before, like so:

```
from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping
import matplotlib.pyplot as plt

#callbacks
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5,
patience=3,
min_delta=1e-4, mode='min', verbose=1)

stop_alg = EarlyStopping(monitor='val_loss', patience=7,
restore_best_weights=True, verbose=1)

#training
hist = lstm.fit(x_train, y_train, batch_size=100, epochs=1000,
callbacks=[stop_alg, reduce_lr], shuffle=True,
validation_data=(x_test, y_test))
```

Next we save the model and display its performance as follows:

```
# save and plot training process
lstm.save_weights("lstm.hdf5")

fig = plt.figure(figsize=(10,6))
plt.plot(hist.history['loss'], color='#785ef0')
plt.plot(hist.history['val_loss'], color='#dc267f')
plt.title('Model Loss Progress')
plt.ylabel('Brinary Cross-Entropy Loss')
plt.xlabel('Epoch')
plt.legend(['Training Set', 'Test Set'], loc='upper right')
plt.show()
```

This code will produce the plot shown in the following diagram:

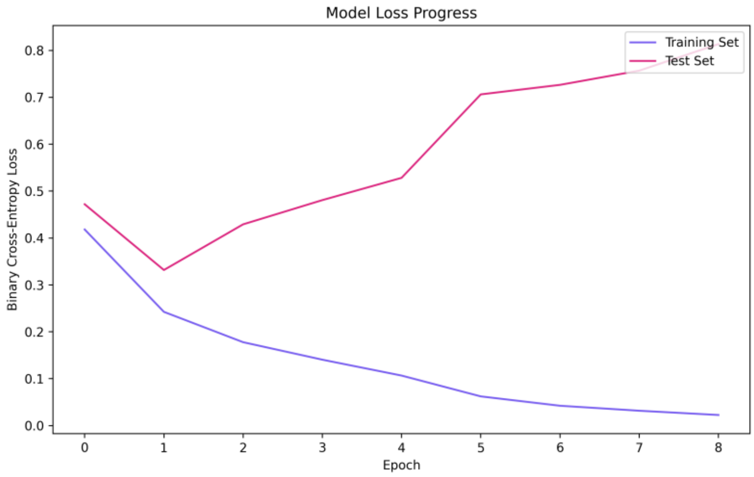

Figure 5.8. Loss across epochs of training an LSTM

Notice from the diagram that the model begins to overfit after one epoch. Using the trained model at the best point, we can calculate the actual performance as follows:

```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import balanced_accuracy_score
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt
import numpy as np

y_hat = lstm.predict(x_test)

# gets the ROC
fpr, tpr, thresholds = roc_curve(y_test, y_hat)
roc_auc = auc(fpr, tpr)

# plots ROC
fig = plt.figure(figsize=(10,6))
plt.plot(fpr, tpr, color='#785ef0',
label='ROC curve (AUC = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='#dc267f', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic Curve')
plt.legend(loc="lower right")
plt.show()

# finds optimal threshold and gets ACC and CM
optimal_idx = np.argmax(tpr - fpr)
optimal_threshold = thresholds[optimal_idx]
print("Threshold value is:", optimal_threshold)
y_pred = np.where(y_hat>=optimal_threshold, 1, 0)
print(balanced_accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
```

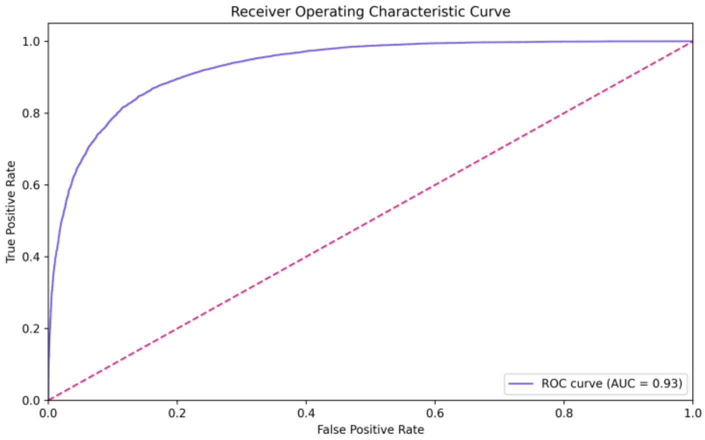

Figure 5.9. ROC curve of an LSTM-based architecture

From the plot, we can see that there is a slight gain in the model, producing an AUC of 0.93 when the simple RNN model had an AUC of 0.92. When looking at the balanced accuracy and the confusion matrix, which was produced by the preceding code, this shows numbers like these:

```
Threshold value is: 0.44251397
```

```
0.8544400000000001
[[10459 2041]
[ 1598 10902]]
```

Here, we can appreciate that the accuracy was of 85.44%, which is a gain of about 2% over the simple RNN. We undertook this experiment simply to show that by switching the RNN models, we can easily see improvements. Of course there are other ways to improve the models, such as the following:

- Increase/reduce the vocabulary size
- Increase/reduce the sequence length
- Increase/reduce the embedding dimension
- Increase/reduce the neural units in recurrent layers

And there may be others besides. So far, you have seen how to take text representations (movie reviews), which is a common NLP task, and find a way to represent those in a space where you can classify them into negative or positive reviews. We did this through embedding and LSTM layers, but at the end of this, there is a dense layer with one neuron that gives the final output. We can think of this as mapping from the text space into a one-dimensional space where we can perform classification. We say this because there are three main ways in which to consider these mappings:

1. **Sequence-to-vector**: Just like the example covered here, mapping sequences to an n-dimensional space.
2. **Vector-to-sequence**: This goes the opposite way, from an n-dimensional space to a sequence.
3. **Sequence-to-sequence**: This maps from a sequence to a sequence, usually going through an n-dimensional mapping in the middle.
4. **Transformer**: A seq2seq model with complex self-attention mechanism is the state of the art model to learn from complex data.

5.3 Sequence-to-vector models

In the previous section, you technically saw a sequence-to-vector model, which took a sequence (of numbers representing words) and mapped to a vector (of one dimension corresponding to a movie review). However, to appreciate these models further, we will move back to MNIST as the source of input to build a model that will take one MNIST numeral and map it to a latent vector.

Let's work in the autoencoder architecture shown in the following diagram. We have studied autoencoders before and now we will use them again since we learned that they are powerful in finding vectorial representations (latent spaces) that are robust and driven by unsupervised learning:

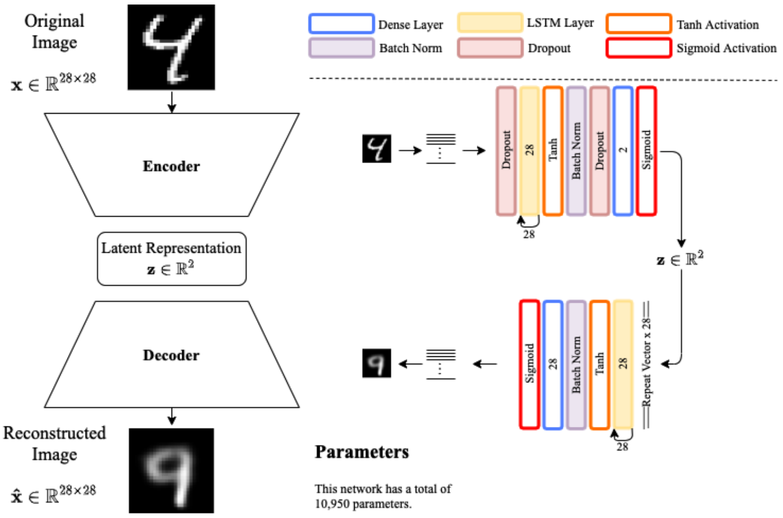

Figure 5.10. LSTM-based autoencoder architecture for MNIST

The goal here is to take an image and find its latent representation, which, in the example of Figure 5.10, would be two dimensions. However, you might be wondering: how can an image be a sequence?

We can interpret an image as a sequence of rows or as a sequence of columns. Let's say that we interpret a two-dimensional image, `28x28` pixels, as a sequence of rows; we can look at every row from top to bottom as a sequence of `28` vectors whose dimensions are each `1x28`. In this way, we can use an LSTM to process those sequences, taking advantage of the LSTM's ability to understand temporal relationships in sequences. By this, we mean that, for example in the case of MNIST, the chances that a particular row in an image will look like the previous or next row are very high.

Notice further that the model proposed in Figure 5.10 does not require an embedding layer as we did before when processing text. Recall that when processing text, we need to embed (vectorize) every single word into a sequence of vectors. However, with images, they already are sequences of vectors, which eliminates the need for an embedding layer.

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Activation, Input
from tensorflow.keras.layers import BatchNormalization, Dropout
from tensorflow.keras.layers import Embedding, LSTM
from tensorflow.keras.layers import RepeatVector, TimeDistributed
from tensorflow.keras.datasets import mnist
from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping
import numpy as np

seqnc_lngth = 28 # length of the sequence; must be 28 for MNIST
ltnt_dim = 2 # latent space dimension; it can be anything reasonable

(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

```
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

print('x_train shape:', x_train.shape)
print('x_test shape:', x_test.shape)
```

The code that we will show here has nothing new to show except for two useful data manipulation tools:

- `RepeatVector()`: This will allow us to arbitrarily repeat a vector. It helps in the decoder (see Figure 5.10) to go from a vector to a sequence.
- `TimeDistributed()`: This will allow us to assign a specific type of layer to every element of a sequence.

After loading the data we can define the encoder part of the model as follows:

```
inpt_vec = Input(shape=(seqnc_lngth, seqnc_lngth,))
l1 = Dropout(0.1)(inpt_vec)
l2 = LSTM(seqnc_lngth, activation='tanh',
recurrent_activation='sigmoid')(l1)
l3 = BatchNormalization()(l2)
l4 = Dropout(0.1)(l3)
l5 = Dense(ltnt_dim, activation='sigmoid')(l4)

# model that takes input and encodes it into the latent space
encoder = Model(inpt_vec, l5)
```

Next, we can define the decoder part of the model as follows:

```
l6 = RepeatVector(seqnc_lngth)(l5)
l7 = LSTM(seqnc_lngth, activation='tanh',
recurrent_activation='sigmoid',
return_sequences=True)(l6)
l8 = BatchNormalization()(l7)
l9 = TimeDistributed(Dense(seqnc_lngth, activation='sigmoid'))(l8)

autoencoder = Model(inpt_vec, l9)
```

Finally, we compile and train the model like this:

```
autoencoder.compile(loss='binary_crossentropy', optimizer='adam')
autoencoder.summary()

reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5,
patience=5,
min_delta=1e-4, mode='min', verbose=1)

stop_alg = EarlyStopping(monitor='val_loss', patience=15,
restore_best_weights=True, verbose=1)

hist = autoencoder.fit(x_train, x_train, batch_size=100, epochs=1000,
callbacks=[stop_alg, reduce_lr], shuffle=True,
validation_data=(x_test, x_test))
```

The code should print the following output, corresponding to the dimensions of the dataset, a summary of the model parameters, followed by the training steps, which we omitted in order to save space:

```
x_train shape: (60000, 28, 28)
x_test shape: (10000, 28, 28)

Model: "functional"
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
input (InputLayer) [(None, 28, 28)] 0
_________________________________________________________________
dropout_1 (Dropout) (None, 28, 28) 0
_________________________________________________________________
lstm_1 (LSTM) (None, 28) 6384
_________________________________________________________________
batch_normalization_1 (Bat (None, 28) 112
_________________________________________________________________
.
.
.
time_distributed (TimeDist (None, 28, 28) 812
=================================================================
Total params: 10,950
Trainable params: 10,838
Non-trainable params: 112
_________________________________________________________________

Epoch 1/1000
600/600 [==============================] - 5s 8ms/step - loss: 0.3542 -
val_loss: 0.2461
```

The model will eventually converge to a valley where it is stopped automatically by the callback. After this, we can simply invoke the encoder model to literally convert any valid sequence (for example, MNIST images) into a vector, which we will do next.

We can invoke the encoder model to convert any valid sequence into a vector like so:

```
encoder.predict(x_test[0:1])
```

This will produce a two-dimensional vector with values corresponding to a vectorial representation of the `sequence x_test[0]`, which is the first image of the test set of MNIST. It might look something like this:

```
array([[3.8787320e-01, 4.8048562e-01]], dtype=float32)
```

However, remember that this model was trained without supervision, hence, the numbers shown here will be different for sure! The encoder model is literally our sequence-to-vector model. The rest of the autoencoder model is meant to do the reconstruction.

If you are curious about how the autoencoder model is able to reconstruct a 28x28 image from a vector of just two values, or if you are curious about how the entire test set of MNIST would look when projected in the learned two-dimensional space, you can run the following code:

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

x_hat = autoencoder.predict(x_test)

smp_idx = [3,2,1,18,4,8,11,0,61,9] # samples for 0,...,9 digits
plt.figure(figsize=(12,6))
for i, (img, y) in enumerate(zip(x_hat[smp_idx].reshape(10, 28, 28),
y_test[smp_idx])):
plt.subplot(2,5,i+1)
plt.imshow(img, cmap='gray')
plt.xticks([])
plt.yticks([])
plt.title(y)
plt.show()
```

Which displays samples of the original digits, as shown in Figure 5.11.

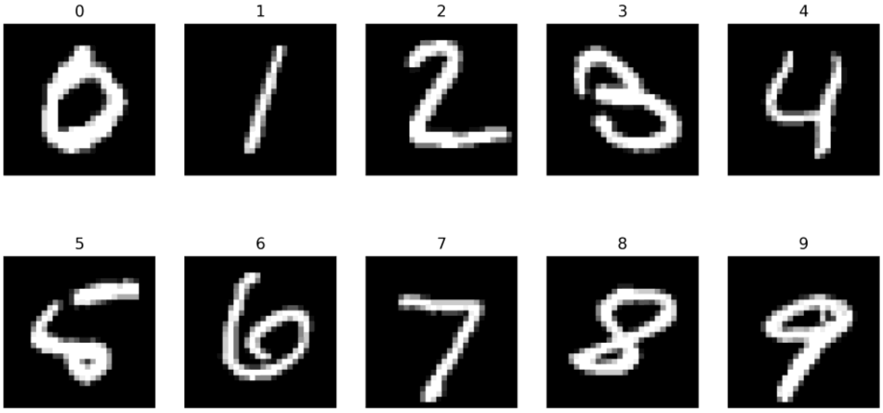

Figure 5.11. MNIST original digits 0-9

The following code produces samples of the reconstructed digits:

```
plt.figure(figsize=(12,6))
for i, (img, y) in enumerate(zip(x_test[smp_idx].reshape(10, 28, 28),
y_test[smp_idx])):
plt.subplot(2,5,i+1)
plt.imshow(img, cmap='gray')
plt.xticks([])
plt.yticks([])
plt.title(y)
plt.show()
```

The reconstructed digits appear as shown in Figure 5.12:

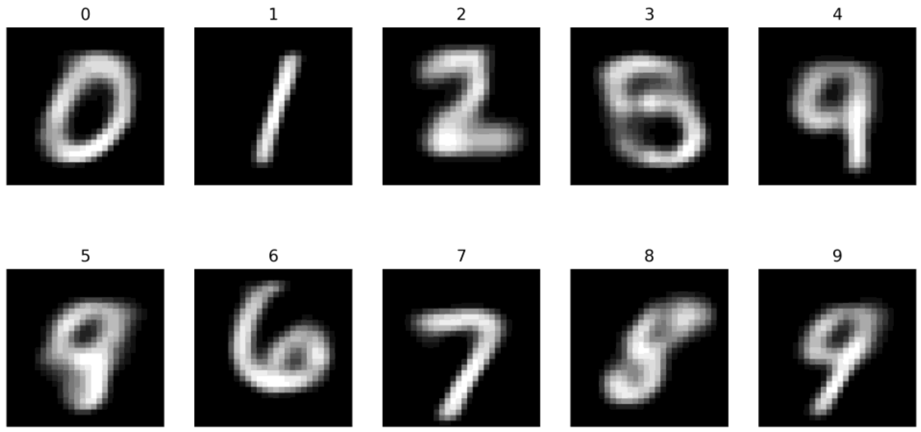

Figure 5.12. MNIST reconstructed digits 0-9 using an LSTM-based autoencoder

The next piece of code will display a scatter plot of the original data projected into the latent space, which is shown in Figure 5.13:

```
y_ = list(map(int, y_test))
X_ = encoder.predict(x_test)

plt.figure(figsize=(10,8))
plt.title('LSTM-based Encoder')
plt.scatter(X_[:,0], X_[:,1], s=5.0, c=y_, alpha=0.75, cmap='tab10')
plt.xlabel('First encoder dimension')
plt.ylabel('Second encoder dimension')
plt.colorbar()
```

Recall that these results may vary due to the unsupervised nature of the autoencoder. Similarly, the learned space can be visually conceived to look like the one shown in Figure 13, where every dot corresponds to a sequence (MNIST digit) that was made a vector of two dimensions:

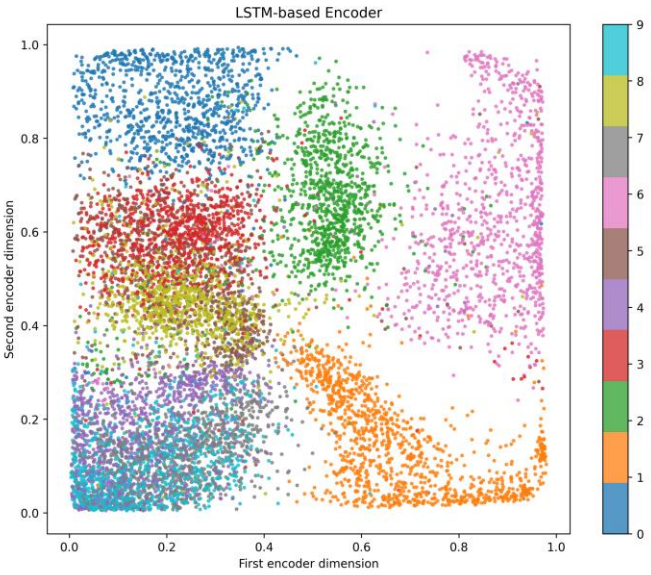

Figure 5.13. Learned vector space based on the MNIST dataset

From Figure 5.13, we can see that the sequence-to-vector model is working decently even when the reconstruction was based only in two-dimensional vectors. We will see larger representations in the next section. However, you need to know that sequence-to-vector models have been very useful in the last few years.

Another useful strategy is to create vector-to-sequence models, which is going from a vectorial representation to a sequential representation. In an autoencoder, this would correspond to the decoder part. Let's go ahead and discuss this next.

5.4 Vector-to-sequence models

If you look back at Figure 5.10, the vector-to-sequence model would correspond to the decoder funnel shape. The major philosophy is that most models usually can go from large inputs down to rich representations with no problems. However, it is only recently that the machine learning community regained traction in producing sequences from vectors very successfully (Goodfellow, I., et al. (2016)).

You can think of Figure 5.10 again and the model represented there, which will produce a sequence back from an original sequence. In this section, we will focus on that second part, the decoder, and use it as a vector-to-sequence model. However, before we go there, we will introduce another version of an RNN, a bi-directional LSTM.

5.4.1 Bi-directional LSTM

A Bi-directional LSTM (BiLSTM), simply put, is an LSTM that analyzes a sequence going forward and backward, as shown in Figure 5.14:

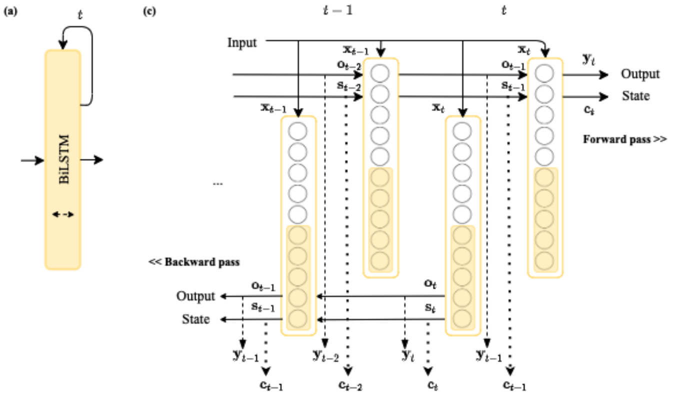

Figure 5.14. A bi-directional LSTM representation

Consider the following examples of sequences analyzed going forward and backward: (1)An audio sequence that is analyzed in natural sound, and then going backward (some people do this to look for subliminal messages). (2) A text sequence, like a sentence, that is analyzed for good style going forward, and also going backward since some patterns (at least in the English and Spanish languages) make reference backward; for example, a verb that makes reference to a subject that appears at the beginning of the sentence. (3) An image that has peculiar shapes going from top to bottom, or bottom to top, or from side to side and backwards; if you think of the number 9, going from top to bottom, a traditional LSTM might forget the round part at the top and remember the slim part at the bottom, but a BiLSTM might be able to recall both important aspects of the number by going top to bottom and bottom to top.

We can implement the bi-directional LSTM by simply invoking the `Bidirectional()` wrapper around a simple LSTM layer. We will then take the architecture in Figure 5.10 and modify it. The new architecture will look like Figure 5.15:

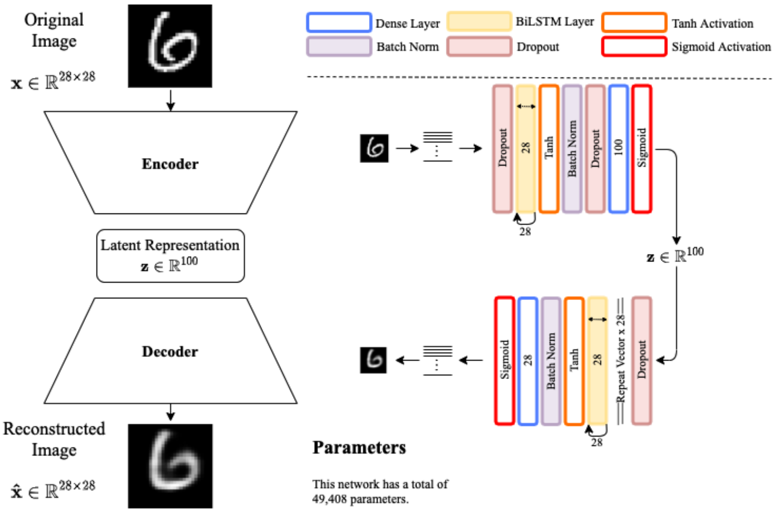

Figure 5. 15. Implementing BiLSTMs with a view to building a vector-to-sequence model

Recall that the most important point here is to make the latent space (the input to the vector-to-sequence model) as rich as possible in order to generate better sequences. We are trying to achieve this by increasing the latent space dimensionality and adding BiLSTMS. Let's go ahead and implement this and look a the results. The code to implement the architecture in Figure 5.15 is the following:

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Activation, Input
from tensorflow.keras.layers import BatchNormalization, Dropout
from tensorflow.keras.layers import Bidirectional, LSTM
from tensorflow.keras.layers import RepeatVector, TimeDistributed
from tensorflow.keras.datasets import mnist
from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping
import numpy as np

seqnc_lngth = 28
ltnt_dim = 100

(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

```
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
```

We define the encoder portion of the model as follows:

```
inpt_vec = Input(shape=(seqnc_lngth, seqnc_lngth,))
l1 = Dropout(0.5)(inpt_vec)
l2 = Bidirectional(LSTM(seqnc_lngth, activation='tanh',
recurrent_activation='sigmoid'))(l1)
l3 = BatchNormalization()(l2)
l4 = Dropout(0.5)(l3)
l5 = Dense(ltnt_dim, activation='sigmoid')(l4)

# sequence to vector model
encoder = Model(inpt_vec, l5, name='encoder')
```

The decoder portion of the model can be defined as follows:

```
ltnt_vec = Input(shape=(ltnt_dim,))
l6 = Dropout(0.1)(ltnt_vec)
l7 = RepeatVector(seqnc_lngth)(l6)
l8 = Bidirectional(LSTM(seqnc_lngth, activation='tanh',
recurrent_activation='sigmoid',
return_sequences=True))(l7)
l9 = BatchNormalization()(l8)
l10 = TimeDistributed(Dense(seqnc_lngth, activation='sigmoid'))(l9)

# vector to sequence model
decoder = Model(ltnt_vec, l10, name='decoder')
```

Next we compile the autoencoder and train it:

```
recon = decoder(encoder(inpt_vec))
autoencoder = Model(inpt_vec, recon, name='ae')

autoencoder.compile(loss='binary_crossentropy', optimizer='adam')
autoencoder.summary()

reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5,
patience=5,
min_delta=1e-4, mode='min', verbose=1)

stop_alg = EarlyStopping(monitor='val_loss', patience=15,
restore_best_weights=True, verbose=1)

hist = autoencoder.fit(x_train, x_train, batch_size=100, epochs=1000,
callbacks=[stop_alg, reduce_lr], shuffle=True,
validation_data=(x_test, x_test))
```

There is nothing new here, except for the `Bidirectional()` wrapper used that has been explained previously. The output should produce a summary of the full autoencoder model and the full training operation and will look something like this:

```
Model: "ae"
```

```
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
input (InputLayer) [(None, 28, 28)] 0
_________________________________________________________________
encoder (Functional) (None, 100) 18692
_________________________________________________________________
decoder (Functional) (None, 28, 28) 30716
=================================================================
Total params: 49,408
Trainable params: 49,184
Non-trainable params: 224
_________________________________________________________________
Epoch 1/1000
600/600 [==============================] - 9s 14ms/step - loss: 0.3150
- val_loss: 0.1927
.
.
.
```

Now, after a number of epochs of unsupervised learning, the training will stop automatically and we can use the `decoder` model as our vector-to-sequence model. But before we do that, we might want to quickly check the quality of the reconstructions by running the same code as before to produce the images shown in the following diagram:

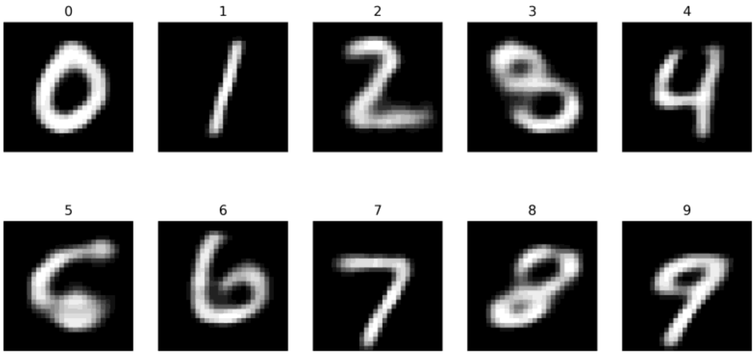

Figure 5.16. MNIST digits reconstructed with a BiLSTM autoencoder

If you compare Figure 5.11 with Figure 5.16, you will notice that the reconstructions are much better and the level of detail is better when compared to the previous model reconstructions in Figure 5
12.

Now we can call our vector-to-sequence model directly with any compatible vector as follows:

```
z = np.random.rand(1,100)
x_ = decoder.predict(z)
print(x_.shape)
plt.imshow(x_[0], cmap='gray')
```

This produces the following output and the plot in Figure 5.17:

```
(1, 28, 28)
```

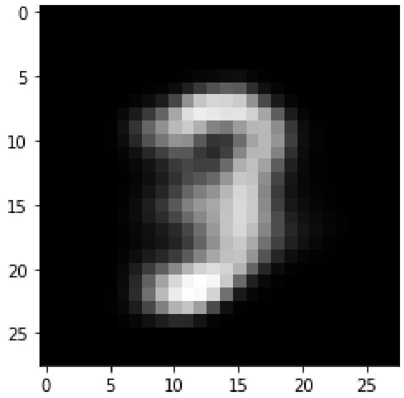

Figure 5.17. Sequence produced by a model from a random vector

You can generate as many random vectors as you wish and test your vector-to-sequence model. And another interesting thing to observe is a sequence-to-sequence model, which we will cover next.

5.5 Sequence-to-sequence models

Sequence-to-sequence learning (Seq2Seq) is about training models to convert sequences from one domain (e.g., sentences in English) to sequences in another domain (e.g., the same sentences translated to Chinese). Just think about the following sequence-to-sequence project ideas:

1. Document summarization. Input sequence: a document. Output sequence: an abstract.
2. Image super resolution. Input sequence: a low-resolution image. Output sequence: a high-resolution image.
3. Video subtitles. Input sequence: video. Output sequence: text captions.
4. Machine translation. Input sequence: text in source language. Output sequence: text in a target language.

These are exciting and extremely challenging applications. If you have used online translators, chances are you have used some type of sequence-to-sequence model.

In this section, to keep it simple, we will continue using the autoencoder in Figure 15 as our main focus, but just to make sure we are all on the same page with respect to the generality of sequence-to-sequence models, we will point out the following notes:

- Sequence-to-sequence models can map across domains; for example, video to text or text to audio.
- Sequence-to-sequence models can map in different dimensions; for example, a low-res image to high-res or vice versa for compression.
- Sequence-to-sequence models can use many different tools, such as dense layers, convolutional layers, and recurrent layers.

With this in mind, you can pretty much build a sequence-to-sequence model depending on your application. For now, we will come back to the model in Figure 5.15 and show that the autoencoder is a sequence-to-sequence model in the sense that it takes a sequence of rows of an image and produces a sequence of rows of another image. Since this is an autoencoder, the input and output dimensions must match.

We will limit our showcase of the previously trained sequence-to-sequence model (autoencoder) to the following short code snippet, which builds up from the code in the previous section:

```
plt.figure(figsize=(12,6))
for i in range(10):
plt.subplot(2,5,i+1)
rnd_vec = np.round(np.mean(x_test[y_test==i],axis=0)) #(a)
rnd_vec = np.reshape(rnd_vec, (1,28,28)) #(b)
z = encoder.predict(rnd_vec) #(c)
decdd = decoder.predict(z) #(d)
plt.imshow(decdd[0], cmap='gray')
plt.xticks([])
plt.yticks([])
plt.title(i)
plt.show()
```

Let's explain some of these steps. In (a), we calculate the average sequence for every single number; this is in response to the question: what can we use as our input sequence since doing random is so easy? Well, using the average sequences to form the test set sounds interesting enough.

Next, (b) is simply to make the input compatible with the encoder input dimensions. Then, (c) takes the average sequence and makes a vector out of it. Finally, (d) uses that vector to recreate the sequence, producing the plot shown in the following diagram:

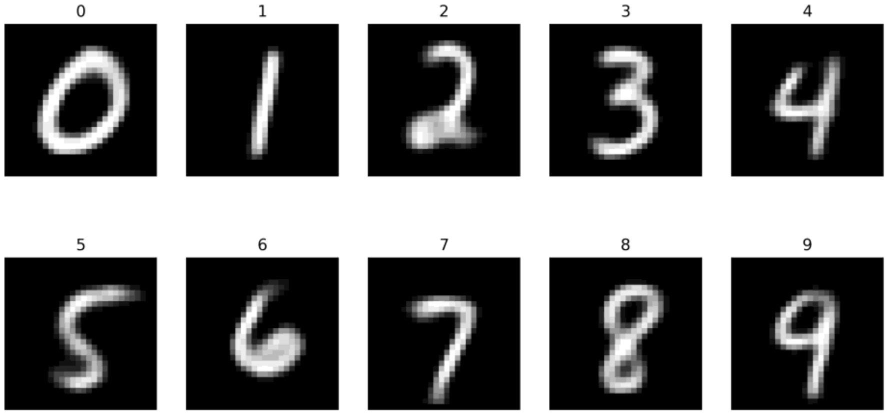

Figure 5.18. Sequence-to-sequence example outputs

From the diagram, you can easily observe well-defined patterns consistent with handwritten numbers, which are generated as sequences of rows by bi-directional LSTMs.

Summary

In this chapter, we showed how to create RNNs to learn from sequential data. RNN are known to be overfitting by extremely short memory. To overcome this problem, we introduced LSTM model and its bi-directional implementation, which is one of the most powerful approaches for sequences that can have distant temporal correlations. You also learned to create an LSTM-based sentiment analysis model for the classification of movie reviews.

At this point, you should feel confident explaining the motivation behind memory in RNNs founded in the need for more robust models. You should feel comfortable coding your own recurrent network using Keras/TensorFlow. Furthermore, you should feel confident implementing both supervised and unsupervised recurrent networks.

LSTMs are great in encoding highly correlated spatial information, such as images, or audio, or text, just like CNNs. However, both CNNs and LSTMs learn very specific latent spaces that may lack diversity. This can cause a problem if there is a malicious hacker that is trying to break your system; if your model is very specific to your data, it may create certain sensitivity to variations, leading to disastrous consequences in your outputs. The next chapter will present a way of overcoming the fragility of neural networks by attacking them and teaching them to be more robust. But before you go, quiz yourself with the following questions.