Chapter 9

How to Develop Encoder-Decoder LSTMs

9.0.1 Lesson Goal

The goal of this lesson is to learn how to develop encoder-decoder LSTM models. After completing this lesson, you will know:

- The Encoder-Decoder LSTM architecture and how to implement it in Keras.
- The addition sequence-to-sequence prediction problem.
- How to develop an Encoder-Decoder LSTM for the addition sequence-to-sequence prediction problem.

9.1 Lesson Overview

This lesson is divided into 7 parts; they are:

1. The Encoder-Decoder LSTM.
2. Addition Prediction Problem.
3. Define and Compile the Model.
4. Fit the Model.
5. Evaluate the Model.
6. Make Predictions With the Model.
7. Complete Example.

Let's get started.

9.2 The Encoder-Decoder LSTM

9.2.1 Sequence-to-Sequence Prediction Problems

Sequence prediction often involves forecasting the next value in a real valued sequence or outputting a class label for an input sequence. This is often framed as a sequence of one input time step to one output time step (e.g. one-to-one) or multiple input time steps to one output time step (many-to-one) type sequence prediction problem.

There is a more challenging type of sequence prediction problem that takes a sequence as input and requires a sequence prediction as output. These are called sequence-to-sequence prediction problems, or seq2seq for short. One modeling concern that makes these problems challenging is that the length of the input and output sequences may vary. Given that there are multiple input time steps and multiple output time steps, this form of problem is referred to as many-to-many type sequence prediction problem.

9.2.2 Architecture

One approach to seq2seq prediction problems that has proven very effective is called the Encoder-Decoder LSTM. This architecture is comprised of two models: one for reading the input sequence and encoding it into a fixed-length vector, and a second for decoding the fixed-length vector and outputting the predicted sequence. The use of the models in concert gives the architecture its name of Encoder-Decoder LSTM designed specifically for seq2seq problems.

> ... RNN Encoder-Decoder, consists of two recurrent neural networks (RNN) that act as an encoder and a decoder pair. The encoder maps a variable-length source sequence to a fixed-length vector, and the decoder maps the vector representation back to a variable-length target sequence.

— *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*, 2014.

The Encoder-Decoder LSTM was developed for natural language processing problems where it demonstrated state-of-the-art performance, specifically in the area of text translation called statistical machine translation. The innovation of this architecture is the use of a fixed-sized internal representation in the heart of the model that input sequences are read to and output sequences are read from. For this reason, the method may be referred to as sequence embedding.

In one of the first applications of the architecture to English-to-French translation, the internal representation of the encoded English phrases was visualized. The plots revealed a qualitatively meaningful learned structure of the phrases harnessed for the translation task.

> The proposed RNN Encoder-Decoder naturally generates a continuous-space representation of a phrase. [...] From the visualization, it is clear that the RNN Encoder-Decoder captures both semantic and syntactic structures of the phrases

— *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*, 2014.

On the task of translation, the model was found to be more effective when the input sequence was reversed. Further, the model was shown to be effective even on very long input sequences.

> We were able to do well on long sentences because we reversed the order of words in the source sentence but not the target sentences in the training and test set. By doing so, we introduced many short term dependencies that made the optimization problem much simpler. ... The simple trick of reversing the words in the source sentence is one of the key technical contributions of this work

— *Sequence to Sequence Learning with Neural Networks*, 2014.

This approach has also been used with image inputs where a Convolutional Neural Network is used as a feature extractor on input images, which is then read by a decoder LSTM.

> ... we propose to follow this elegant recipe, replacing the encoder RNN by a deep convolution neural network (CNN). [...] it is natural to use a CNN as an image "encoder", by first pre-training it for an image classification task and using the last hidden layer as an input to the RNN decoder that generates sentences

— *Show and Tell: A Neural Image Caption Generator*, 2014.

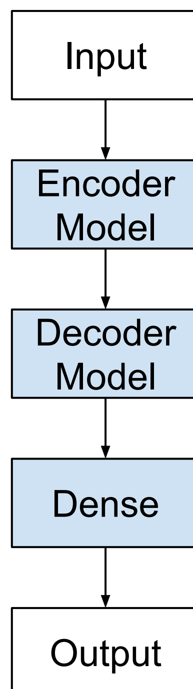

Figure 9.1: Encoder-decoder LSTM Architecture.

9.2.3 Applications

The list below highlights some interesting applications of the Encoder-Decoder LSTM architecture.

- **Machine Translation**, e.g. English to French translation of phrases.

- **Learning to Execute**, e.g. calculate the outcome of small programs.
- **Image Captioning**, e.g. generating a text description for images.
- **Conversational Modeling**, e.g. generating answers to textual questions.
- **Movement Classification**, e.g. generating a sequence of commands from a sequence of gestures.

9.2.4 Implementation

The Encoder-Decoder LSTM can be implemented directly in Keras. We can think of the model as being comprised of two key parts: the encoder and the decoder. First, the input sequence is shown to the network one encoded character at a time. We need an encoding level to learn the relationship between the steps in the input sequence and develop an internal representation of these relationships.

One or more LSTM layers can be used to implement the encoder model. The output of this model is a fixed-size vector that represents the internal representation of the input sequence. The number of memory cells in this layer defines the length of this fixed-sized vector.

```
model = Sequential()
model.add(LSTM(..., input_shape=(...)))
```

Listing 9.1: Example of a Vanilla LSTM model.

The decoder must transform the learned internal representation of the input sequence into the correct output sequence. One or more LSTM layers can also be used to implement the decoder model. This model reads from the fixed sized output from the encoder model. As with the Vanilla LSTM, a `Dense` layer is used as the output for the network. The same weights can be used to output each time step in the output sequence by wrapping the `Dense` layer in a `TimeDistributed` wrapper.

```
model.add(LSTM(..., return_sequences=True))
model.add(TimeDistributed(Dense(...)))
```

Listing 9.2: Example of a LSTM model with `TimeDistributed` wrapped `Dense` layer.

There's a problem though. We must connect the encoder to the decoder, and they do not fit. That is, the encoder will produce a 2-dimensional matrix of outputs, where the length is defined by the number of memory cells in the layer. The decoder is an `LSTM` layer that expects a 3D input of [samples, time steps, features] in order to produce a decoded sequence of some different length defined by the problem.

If you try to force these pieces together, you get an error indicating that the output of the decoder is 2D and 3D input to the decoder is required. We can solve this using a `RepeatVector` layer. This layer simply repeats the provided 2D input multiple times to create a 3D output.

The `RepeatVector` layer can be used like an adapter to fit the encoder and decoder parts of the network together. We can configure the `RepeatVector` to repeat the fixed length vector one time for each time step in the output sequence.

```
model.add(RepeatVector(...))
```

Listing 9.3: Example of a `RepeatVector` layer.

Putting this together, we have:

```
model = Sequential()
model.add(LSTM(..., input_shape=(...)))
model.add(RepeatVector(...))
model.add(LSTM(..., return_sequences=True))
model.add(TimeDistributed(Dense(...)))
```

Listing 9.4: Example of an Encoder-Decoder model.

To summarize, the `RepeatVector` is used as an adapter to fit the fixed-sized 2D output of the encoder to the differing length and 3D input expected by the decoder. The `TimeDistributed` wrapper allows the same output layer to be reused for each element in the output sequence.

9.3 Addition Prediction Problem

The addition problem is a sequence-to-sequence, or seq2seq, prediction problem. It was used by Wojciech Zaremba and Ilya Sutskever in their 2014 paper exploring the capabilities of the Encoder-Decoder LSTM titled "*Learning to Execute*" where the architecture was demonstrated learning to calculate the output of small programs.

The problem is defined as calculating the sum output of two input numbers. This is challenging as each digit and mathematical symbol is provided as a character and the expected output is also expected as characters. For example, the input `10+6` with the output `16` would be represented by the sequences:

```
Input: ['1', '0', '+', '6']
Output: ['1', '6']
```

Listing 9.5: Example of input and output sequences for the addition problem.

The model must learn not only the integer nature of the characters, but also the nature of the mathematical operation to perform. Notice how sequence is now important, and that randomly shuffling the input will create a nonsense sequence that could not be related to the output sequence. Also notice how the number of digits could vary in both the input and output sequences. Technically this makes the addition prediction problem a sequence-to-sequence problem that requires a many-to-many model to address.

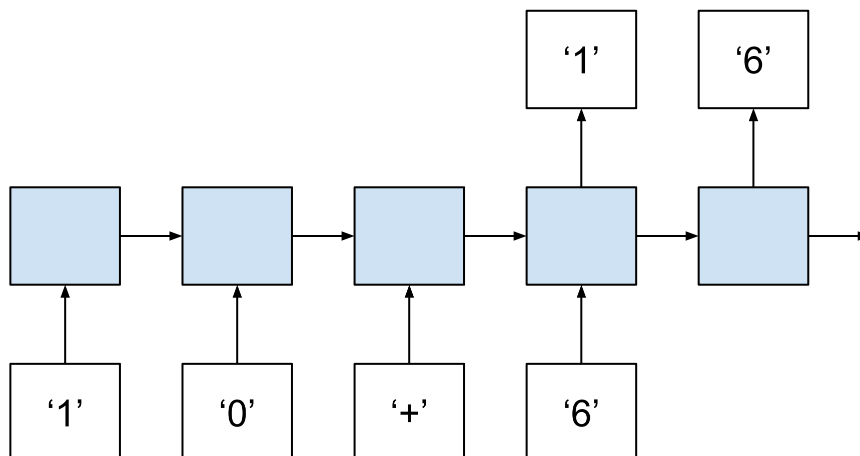

Figure 9.2: Addition prediction problem framed with a many-to-many prediction model.

We can keep things simple with addition of two numbers, but we can see how this may be scaled to a variable number of terms and mathematical operations that could be given as input for the model to learn and generalize. This problem can be implemented in Python. We will divide this into the following steps:

1. Generate Sum Pairs.
2. Integers to Padded Strings.
3. Integer Encoded Sequences.
4. One Hot Encoded Sequences.
5. Sequence Generation Pipeline.
6. Decode Sequences.

9.3.1 Generate Sum Pairs

The first step is to generate sequences of random integers and their sum. We can put this in a function named `random_sum_pairs()`, as follows.

```
from random import seed
from random import randint

# generate lists of random integers and their sum
def random_sum_pairs(n_examples, n_numbers, largest):
  X, y = list(), list()
  for _ in range(n_examples):
    in_pattern = [randint(1,largest) for _ in range(n_numbers)]
    out_pattern = sum(in_pattern)
    X.append(in_pattern)
    y.append(out_pattern)
  return X, y
```

```
seed(1)
n_samples = 1
n_numbers = 2
largest = 10
# generate pairs
X, y = random_sum_pairs(n_samples, n_numbers, largest)
print(X, y)
```

Listing 9.6: Example of generating random a sequence pair.

Running just this function prints a single example of adding two random integers between 1 and 10.

```
[[3, 10]] [13]
```

Listing 9.7: Example of output of generating a random sequence pair.

9.3.2 Integers to Padded Strings

The next step is to convert the integers to strings. The input string will have the format '10+10' and the output string will have the format '20'. Key to this function is the padding of numbers to ensure that each input and output sequence has the same number of characters. A padding character should be different from the data so the model can learn to ignore them. In this case, we use the space character for padding(' ') and pad the string on the left, keeping the information on the far right.

There are other ways to pad, such as padding each term individually. Try it and see if it results in better performance. Padding requires we know how long the longest sequence may be. We can calculate this easily by taking the `log10()` of the largest integer we can generate and the ceiling of that number to get an idea of how many chars are needed for each number. We add 1 to the largest number to ensure we expect 3 chars instead of 2 chars for the case of a round largest number, like 200 and take the ceiling of the result (e.g. `ceil(log10(largest+1))`). We then need to add the right number of plus symbols (e.g. `n_numbers - 1`).

```
max_length = int(n_numbers * ceil(log10(largest+1)) + n_numbers - 1)
```

Listing 9.8: Example of calculating the maximum length of input sequences.

We can make this concrete with a worked example where the total number of terms (`n_numbers`) is 3 and the largest value (`largest`) is 10.

```
max_length = n_numbers * ceil(log10(largest+1)) + n_numbers - 1
max_length = 3 * ceil(log10(10+1)) + 3 - 1
max_length = 3 * ceil(1.0413926851582251) + 3 - 1
max_length = 3 * 2 + 3 - 1
max_length = 6 + 3 - 1
max_length = 8
```

Listing 9.9: Worked example of maximum input sequence length.

Intuitively, we would expect 2 spaces for each term (e.g. `['1','0']`) multiplied by 3 terms, or a maximum length of input sequences of 6 spaces with two more spaces for the addition symbols (e.g. `['1','0','+','1','0','+','1','0']`) making the largest possible sequence 8 characters in length. This is what we see in the worked example.

A similar process is repeated on the output sequence, without the plus symbols, of course.

```
max_length = int(ceil(log10(n_numbers * (largest+1))))
```

Listing 9.10: Example of calculating the length of output sequences.

Again, we can make this concrete by calculating the expected maximum output sequence length for the above example with the total number of terms (`n_numbers`) is 3 and the largest value (`largest`) is 10.

```
max_length = ceil(log10(n_numbers * (largest+1)))
max_length = ceil(log10(3 * (10+1)))
max_length = ceil(log10(33))
max_length = ceil(1.5185139398778875)
max_length = 2
```

Listing 9.11: Worked example of maximum output sequence length.

Again, intuitively, we would expect the largest possible addition to be `10+10+10` or the value of `30`. This would require a maximum length of 2, and this is what we see in the worked example. The example below adds the `to_string()` function and demonstrates its usage with a single input/output pair.

```
from random import seed
from random import randint
from math import ceil
from math import log10

# generate lists of random integers and their sum
def random_sum_pairs(n_examples, n_numbers, largest):
  X, y = list(), list()
  for _ in range(n_examples):
    in_pattern = [randint(1,largest) for _ in range(n_numbers)]
    out_pattern = sum(in_pattern)
    X.append(in_pattern)
    y.append(out_pattern)
  return X, y

# convert data to strings
def to_string(X, y, n_numbers, largest):
  max_length = int(n_numbers * ceil(log10(largest+1)) + n_numbers - 1)
  Xstr = list()
  for pattern in X:
    strp = '+'.join([str(n) for n in pattern])
    strp = ''.join([' ' for _ in range(max_length-len(strp))]) + strp
    Xstr.append(strp)
  max_length = int(ceil(log10(n_numbers * (largest+1))))
  ystr = list()
  for pattern in y:
    strp = str(pattern)
    strp = ''.join([' ' for _ in range(max_length-len(strp))]) + strp
    ystr.append(strp)
  return Xstr, ystr

seed(1)
n_samples = 1
n_numbers = 2
largest = 10
```

```
# generate pairs
X, y = random_sum_pairs(n_samples, n_numbers, largest)
print(X, y)
# convert to strings
X, y = to_string(X, y, n_numbers, largest)
print(X, y)
```

Listing 9.12: Example of converting a sequence pair to padded characters.

Running this example first prints the integer sequence and the padded string representation of the same sequence.

```
[[3, 10]] [13]
[' 3+10'] ['13']
```

Listing 9.13: Example of output of converting a sequence pair to padded characters.

9.3.3 Integer Encoded Sequences

Next, we need to encode each character in the string as an integer value. We have to work with numbers in neural networks after all, not characters. Integer encoding transforms the problem into a classification problem where the output sequence may be considered class outputs with 11 possible values each. This just so happens to be integers with some ordinal relationship (the first 10 class values). To perform this encoding, we must define the full alphabet of symbols that may appear in the string encoding, as follows:

```
alphabet = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', ' ']
```

Listing 9.14: Example of defining a character alphabet.

Integer encoding then becomes a simple process of building a lookup table of character-to-integer offset and converting each char of each string, one by one. The example below provides the `integer_encode()` function for integer encoding and demonstrates how to use it.

```
from random import seed
from random import randint
from math import ceil
from math import log10

# generate lists of random integers and their sum
def random_sum_pairs(n_examples, n_numbers, largest):
  X, y = list(), list()
  for _ in range(n_examples):
    in_pattern = [randint(1,largest) for _ in range(n_numbers)]
    out_pattern = sum(in_pattern)
    X.append(in_pattern)
    y.append(out_pattern)
  return X, y

# convert data to strings
def to_string(X, y, n_numbers, largest):
  max_length = int(n_numbers * ceil(log10(largest+1)) + n_numbers - 1)
  Xstr = list()
  for pattern in X:
    strp = '+'.join([str(n) for n in pattern])
```

```
    strp = ''.join([' ' for _ in range(max_length-len(strp))]) + strp
    Xstr.append(strp)
  max_length = int(ceil(log10(n_numbers * (largest+1))))
  ystr = list()
  for pattern in y:
    strp = str(pattern)
    strp = ''.join([' ' for _ in range(max_length-len(strp))]) + strp
    ystr.append(strp)
  return Xstr, ystr

# integer encode strings
def integer_encode(X, y, alphabet):
  char_to_int = dict((c, i) for i, c in enumerate(alphabet))
  Xenc = list()
  for pattern in X:
    integer_encoded = [char_to_int[char] for char in pattern]
    Xenc.append(integer_encoded)
  yenc = list()
  for pattern in y:
    integer_encoded = [char_to_int[char] for char in pattern]
    yenc.append(integer_encoded)
  return Xenc, yenc

seed(1)
n_samples = 1
n_numbers = 2
largest = 10
# generate pairs
X, y = random_sum_pairs(n_samples, n_numbers, largest)
print(X, y)
# convert to strings
X, y = to_string(X, y, n_numbers, largest)
print(X, y)
# integer encode
alphabet = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', ' ']
X, y = integer_encode(X, y, alphabet)
print(X, y)
```

Listing 9.15: Example of integer encoding padded sequences.

Running the example prints the integer encoded version of each string encoded pattern. We can see that the space character (' ') was encoded with 11 and the three character ('3') was encoded as 3, and so on.

```
[[3, 10]] [13]
[' 3+10'] ['13']
[[11, 3, 10, 1, 0]] [[1, 3]]
```

Listing 9.16: Example output from integer encoding input and output sequences.

9.3.4 One Hot Encoded Sequences

The next step is to binary encode the integer encoding sequences. This involves converting each integer to a binary vector with the same length as the alphabet and marking the specific integer with a 1. For example, a 0 integer represents the `'0'` character and would be encoded as a

binary vector with a 1 in the 0th position of an 11 element vector: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]. The example below defines the one_hot_encode() function for binary encoding and demonstrates how to use it.

```
from random import seed
from random import randint
from math import ceil
from math import log10

# generate lists of random integers and their sum
def random_sum_pairs(n_examples, n_numbers, largest):
  X, y = list(), list()
  for _ in range(n_examples):
    in_pattern = [randint(1,largest) for _ in range(n_numbers)]
    out_pattern = sum(in_pattern)
    X.append(in_pattern)
    y.append(out_pattern)
  return X, y

# convert data to strings
def to_string(X, y, n_numbers, largest):
  max_length = int(n_numbers * ceil(log10(largest+1)) + n_numbers - 1)
  Xstr = list()
  for pattern in X:
    strp = '+'.join([str(n) for n in pattern])
    strp = ''.join([' ' for _ in range(max_length-len(strp))]) + strp
    Xstr.append(strp)
  max_length = int(ceil(log10(n_numbers * (largest+1))))
  ystr = list()
  for pattern in y:
    strp = str(pattern)
    strp = ''.join([' ' for _ in range(max_length-len(strp))]) + strp
    ystr.append(strp)
  return Xstr, ystr

# integer encode strings
def integer_encode(X, y, alphabet):
  char_to_int = dict((c, i) for i, c in enumerate(alphabet))
  Xenc = list()
  for pattern in X:
    integer_encoded = [char_to_int[char] for char in pattern]
    Xenc.append(integer_encoded)
  yenc = list()
  for pattern in y:
    integer_encoded = [char_to_int[char] for char in pattern]
    yenc.append(integer_encoded)
  return Xenc, yenc

# one hot encode
def one_hot_encode(X, y, max_int):
  Xenc = list()
  for seq in X:
    pattern = list()
    for index in seq:
      vector = [0 for _ in range(max_int)]
      vector[index] = 1
```

```
      pattern.append(vector)
    Xenc.append(pattern)
  yenc = list()
  for seq in y:
    pattern = list()
    for index in seq:
      vector = [0 for _ in range(max_int)]
      vector[index] = 1
      pattern.append(vector)
    yenc.append(pattern)
  return Xenc, yenc

seed(1)
n_samples = 1
n_numbers = 2
largest = 10
# generate pairs
X, y = random_sum_pairs(n_samples, n_numbers, largest)
print(X, y)
# convert to strings
X, y = to_string(X, y, n_numbers, largest)
print(X, y)
# integer encode
alphabet = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', ' ']
X, y = integer_encode(X, y, alphabet)
print(X, y)
# one hot encode
X, y = one_hot_encode(X, y, len(alphabet))
print(X, y)
```

Listing 9.17: Example of one hot encoding an integer encoded sequences.

Running the example prints the binary encoded sequence for each integer encoding. I've added some new lines to make the input and output binary encodings clearer. You can see that a single sum pattern becomes a sequence of 5 binary encoded vectors, each with 11 elements. The output, or sum, becomes a sequence of 2 binary encoded vectors, again each with 11 elements.

```
[[3, 10]] [13]
[' 3+10'] ['13']
[[11, 3, 10, 1, 0]] [[1, 3]]
[[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
  [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
  [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
  [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]]
 [[[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]]]
```

Listing 9.18: Example output from one hot encoding an integer encoded sequences.

9.3.5 Sequence Generation Pipeline

We can tie all of these steps together into a function called `generate_data()`, listed below. Given a designed number of samples, number of terms, the largest value of each term, and the alphabet of possible characters, the function will generate a set of input and output sequences.

```
# generate an encoded dataset
def generate_data(n_samples, n_numbers, largest, alphabet):
  # generate pairs
  X, y = random_sum_pairs(n_samples, n_numbers, largest)
  # convert to strings
  X, y = to_string(X, y, n_numbers, largest)
  # integer encode
  X, y = integer_encode(X, y, alphabet)
  # one hot encode
  X, y = one_hot_encode(X, y, len(alphabet))
  # return as NumPy arrays
  X, y = array(X), array(y)
  return X, y
```

Listing 9.19: Function for generating a sequence, encoding and reshaping it for an LSTM model.

9.3.6 Decode Sequences

Finally, we need to invert the encoding to convert the output vectors back into numbers so we can compare expected output integers to predicted integers. The `invert()` function below performs this operation. Key is first converting the binary encoding back into an integer using the `argmax()` function, then converting the integer back into a character using a reverse mapping of the integers to chars from the alphabet.

```
# invert encoding
def invert(seq, alphabet):
  int_to_char = dict((i, c) for i, c in enumerate(alphabet))
  strings = list()
  for pattern in seq:
    string = int_to_char[argmax(pattern)]
    strings.append(string)
  return ''.join(strings)
```

Listing 9.20: Function for deciding an encoded input or output sequence.

We now have everything we need to prepare data for this example.

9.4 Define and Compile the Model

The first step is to define the specifications of the sequence prediction problem. We must specify 3 parameters as input to the `generate_data()` function (above) for generating samples of input-output sequences:

- `n_terms`: The number of terms in the equation, (e.g. 2 for `10+10`).
- `largest`: The largest numerical value for each term (e.g. 10 for values between 1-10).
- `alphabet`: The symbols used to encode the input and output sequences (e.g. `0-9`, `+` and ‘ ’)

We will use a configuration of the problem that has a modest complexity. Each instance will be comprised of 3 terms with the maximum value of 10 per term. The alphabet remains fixed regardless of configuration with the values `0-9`, `'+'`, and `' '`.

```
# number of math terms
n_terms = 3
# largest value for any single input digit
largest = 10
# scope of possible symbols for each input or output time step
alphabet = [str(x) for x in range(10)] + ['+', ' ']
```

Listing 9.21: Example of configuring the problem instance.

The network needs three configuration values defined by the specification of the addition problem:

- `n_chars`: The size of the alphabet for a single time step (e.g. 12 for 0-9, '+' and ' ').
- `n_in_seq_length`: The number of time steps of encoded input sequences (e.g. 8 for '10+10+10').
- `n_out_seq_length`: The number of time steps of an encoded output sequence (e.g. 2 for '30')

The `n_chars` variable is used to define the number of features in the input layer and the number of features in the output layer for each input and output time step. The `n_in_seq_length` variable is used to define the number of time steps for the input layer of the network. The `n_out_seq_length` variable is used to define the number of times to repeat the encoded input in the `RepeatVector` that in turn defines the length of the sequence fed to the decoder for creating the output sequence. The definition of `n_in_seq_length` and `n_out_seq_length` uses the same code from the `to_string()` function used to map the integer sequence to strings.

```
# size of alphabet: (12 for 0-9, + and ' ')
n_chars = len(alphabet)
# length of encoded input sequence (8 for '10+10+10)
n_in_seq_length = int(n_terms * ceil(log10(largest+1)) + n_terms - 1)
# length of encoded output sequence (2 for '30')
n_out_seq_length = int(ceil(log10(n_terms * (largest+1))))
```

Listing 9.22: Example of defining network configuration based on the problem instance.

We are now ready to define the Encoder-Decoder LSTM. We will use a single `LSTM` layer for the encoder and another single layer for the decoder. The encoder is defined with 75 memory cells and the decoder with 50 memory cells. The number of memory cells was found with a little trial and error. The asymmetry in layer sizes in the encoder and decoder seems like a natural organization given that input sequences are relatively longer than output sequences.

The output layer uses the categorical log loss for the 12 possible *classes* that may be predicted. The efficient Adam implementation of gradient descent is used and accuracy will be calculated during training and model evaluation.

```
# define LSTM
model = Sequential()
model.add(LSTM(75, input_shape=(n_in_seq_length, n_chars)))
model.add(RepeatVector(n_out_seq_length))
```

```
model.add(LSTM(50, return_sequences=True))
model.add(TimeDistributed(Dense(n_chars, activation='softmax')))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model.summary())
```

Listing 9.23: Example of defining and compiling the Encoder-Decoder LSTM.

Running the example prints a summary of the network structure. We can see that the Encoder will output a fixed size vector with the length of 75 for a given input sequence. This sequence is repeated 2 times to provide a sequence of 2 time steps of 75 *features* to the decoder. The decoder outputs two time steps of 50 features to the `Dense` output layer that processes these one at a time via the `TimeDistributed` wrapper to output one encoded character at a time.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 75)                26400
_________________________________________________________________
repeat_vector_1 (RepeatVecto (None, 2, 75)             0
_________________________________________________________________
lstm_2 (LSTM)                (None, 2, 50)             25200
_________________________________________________________________
time_distributed_1 (TimeDist (None, 2, 12)             612
=================================================================
Total params: 52,212
Trainable params: 52,212
Non-trainable params: 0
_________________________________________________________________
```

Listing 9.24: Example output from defining and compiling the Encoder-Decoder LSTM.

9.5 Fit the Model

The model is fit on a single epoch of 75,000 randomly generated instances of input-output pairs. The number of sequences is a proxy for the number of training epochs. The total of 75,000 and a batch size of 32 were found with a little trial and error and are by no means an optimal configuration.

```
# fit LSTM
X, y = generate_data(75000, n_terms, largest, alphabet)
model.fit(X, y, epochs=1, batch_size=32)
```

Listing 9.25: Example of fitting the defined Encoder-Decoder LSTM.

Fitting the model provides a progress bar that shows the loss and accuracy of the model at the end of each batch. The model does not take long to fit on the CPU. If the progress bar interferes with your development environment, you can turn it off by setting `verbose=0` in the call to the `fit()` function.

```
75000/75000 [==============================] - 37s - loss: 0.6982 - acc: 0.7943
```

Listing 9.26: Example output from fitting the defined Encoder-Decoder LSTM.

9.6 Evaluate the Model

We can evaluate the model by generating predictions on 100 different randomly generated input-output pairs. The result will give an estimate of the model skill on randomly generated examples in general.

```
# evaluate LSTM
X, y = generate_data(100, n_terms, largest, alphabet)
loss, acc = model.evaluate(X, y, verbose=0)
print('Loss: %f, Accuracy: %f' % (loss, acc*100))
```

Listing 9.27: Example of evaluating the fit Encoder-Decoder LSTM.

Running the example prints both the log loss and accuracy of the model. Your specific values may differ because of the stochastic nature of neural networks, but the model accuracy should be in the high 90s.

```
Loss: 0.128379, Accuracy: 100.000000
```

Listing 9.28: Example output from evaluating the fit Encoder-Decoder LSTM.

9.7 Make Predictions with the Model

We can make predictions using the fit model. We will demonstrate making one prediction at a time and provide a summary of the decoded input, expected output, and predicted output. Printing the decoded output gives us a more concrete connection to the problem and model performance. Here, we generate 10 new random input-output sequence pairs, make a prediction using the fit model for each, decode all the sequences involved, and print them to the screen.

```
# predict
for _ in range(10):
  # generate an input-output pair
  X, y = generate_data(1, n_terms, largest, alphabet)
  # make prediction
  yhat = model.predict(X, verbose=0)
  # decode input, expected and predicted
  in_seq = invert(X[0], alphabet)
  out_seq = invert(y[0], alphabet)
  predicted = invert(yhat[0], alphabet)
  print('%s = %s (expect %s)' % (in_seq, predicted, out_seq))
```

Listing 9.29: Example of making predictions with the fit Encoder-Decoder LSTM.

Running the example shows that the model gets most of the sequences correct. The specific sequences you generate and the skill of the model on just ten examples will vary. Try running the prediction piece a few times to get a good feel for the model behavior.

```
 9+10+9 = 27 (expect 28)
  9+6+9 = 24 (expect 24)
 8+9+10 = 27 (expect 27)
 9+9+10 = 28 (expect 28)
  2+4+5 = 11 (expect 11)
  2+9+7 = 18 (expect 18)
  7+3+2 = 12 (expect 12)
```

```
4+1+4 = 9 (expect 9)
8+6+7 = 21 (expect 21)
5+2+7 = 14 (expect 14)
```

Listing 9.30: Example output from making predictions with the fit Encoder-Decoder LSTM.

9.8 Complete Example

For completeness, the full code listing is provided below for your reference.

```
from random import randint
from numpy import array
from math import ceil
from math import log10
from numpy import argmax
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import TimeDistributed
from keras.layers import RepeatVector

# generate lists of random integers and their sum
def random_sum_pairs(n_examples, n_numbers, largest):
  X, y = list(), list()
  for _ in range(n_examples):
    in_pattern = [randint(1,largest) for _ in range(n_numbers)]
    out_pattern = sum(in_pattern)
    X.append(in_pattern)
    y.append(out_pattern)
  return X, y

# convert data to strings
def to_string(X, y, n_numbers, largest):
  max_length = int(n_numbers * ceil(log10(largest+1)) + n_numbers - 1)
  Xstr = list()
  for pattern in X:
    strp = '+'.join([str(n) for n in pattern])
    strp = ''.join([' ' for _ in range(max_length-len(strp))]) + strp
    Xstr.append(strp)
  max_length = int(ceil(log10(n_numbers * (largest+1))))
  ystr = list()
  for pattern in y:
    strp = str(pattern)
    strp = ''.join([' ' for _ in range(max_length-len(strp))]) + strp
    ystr.append(strp)
  return Xstr, ystr

# integer encode strings
def integer_encode(X, y, alphabet):
  char_to_int = dict((c, i) for i, c in enumerate(alphabet))
  Xenc = list()
  for pattern in X:
    integer_encoded = [char_to_int[char] for char in pattern]
    Xenc.append(integer_encoded)
```

```
  yenc = list()
  for pattern in y:
    integer_encoded = [char_to_int[char] for char in pattern]
    yenc.append(integer_encoded)
  return Xenc, yenc

# one hot encode
def one_hot_encode(X, y, max_int):
  Xenc = list()
  for seq in X:
    pattern = list()
    for index in seq:
      vector = [0 for _ in range(max_int)]
      vector[index] = 1
      pattern.append(vector)
    Xenc.append(pattern)
  yenc = list()
  for seq in y:
    pattern = list()
    for index in seq:
      vector = [0 for _ in range(max_int)]
      vector[index] = 1
      pattern.append(vector)
    yenc.append(pattern)
  return Xenc, yenc

# generate an encoded dataset
def generate_data(n_samples, n_numbers, largest, alphabet):
  # generate pairs
  X, y = random_sum_pairs(n_samples, n_numbers, largest)
  # convert to strings
  X, y = to_string(X, y, n_numbers, largest)
  # integer encode
  X, y = integer_encode(X, y, alphabet)
  # one hot encode
  X, y = one_hot_encode(X, y, len(alphabet))
  # return as numpy arrays
  X, y = array(X), array(y)
  return X, y

# invert encoding
def invert(seq, alphabet):
  int_to_char = dict((i, c) for i, c in enumerate(alphabet))
  strings = list()
  for pattern in seq:
    string = int_to_char[argmax(pattern)]
    strings.append(string)
  return ''.join(strings)

# configure problem

# number of math terms
n_terms = 3
# largest value for any single input digit
largest = 10
# scope of possible symbols for each input or output time step
```

```
alphabet = [str(x) for x in range(10)] + ['+', ' ']

# size of alphabet: (12 for 0-9, + and ' ')
n_chars = len(alphabet)
# length of encoded input sequence (8 for '10+10+10)
n_in_seq_length = int(n_terms * ceil(log10(largest+1)) + n_terms - 1)
# length of encoded output sequence (2 for '30')
n_out_seq_length = int(ceil(log10(n_terms * (largest+1))))

# define LSTM
model = Sequential()
model.add(LSTM(75, input_shape=(n_in_seq_length, n_chars)))
model.add(RepeatVector(n_out_seq_length))
model.add(LSTM(50, return_sequences=True))
model.add(TimeDistributed(Dense(n_chars, activation='softmax')))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model.summary())

# fit LSTM
X, y = generate_data(75000, n_terms, largest, alphabet)
model.fit(X, y, epochs=1, batch_size=32)

# evaluate LSTM
X, y = generate_data(100, n_terms, largest, alphabet)
loss, acc = model.evaluate(X, y, verbose=0)
print('Loss: %f, Accuracy: %f' % (loss, acc*100))

# predict
for _ in range(10):
  # generate an input-output pair
  X, y = generate_data(1, n_terms, largest, alphabet)
  # make prediction
  yhat = model.predict(X, verbose=0)
  # decode input, expected and predicted
  in_seq = invert(X[0], alphabet)
  out_seq = invert(y[0], alphabet)
  predicted = invert(yhat[0], alphabet)
  print('%s = %s (expect %s)' % (in_seq, predicted, out_seq))
```

Listing 9.31: Complete examples of the Encoder-Decoder LSTM model on the Addition Prediction prediction problem.

9.9 Further Reading

This section provides some resources for further reading.

9.9.1 Papers on Encoder-Decoder LSTM

- *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*, 2014.
  https://arxiv.org/abs/1406.1078

- *Sequence to Sequence Learning with Neural Networks*, 2014.
  https://arxiv.org/abs/1409.3215
- *Show and Tell: A Neural Image Caption Generator*, 2014.
  https://arxiv.org/abs/1411.4555
- *Learning to Execute*, 2015.
  http://arxiv.org/abs/1410.4615
- *A Neural Conversational Model*, 2015.
  https://arxiv.org/abs/1506.05869

9.9.2 Keras API

- `RepeatVector` Keras API.
  https://keras.io/layers/core/#repeatvector
- `TimeDistributed` Keras API.
  https://keras.io/layers/wrappers/#timedistributed

9.10 Extensions

Do you want to dive deeper into Encoder-Decoder LSTMs? This section lists some challenging extensions to this lesson.

- List 10 sequence-to-sequence prediction problems that may benefit from the Encoder-Decoder LSTM architecture.
- Increase the number of terms or number of digits and tune the model to get 100% accuracy.
- Design a study to compare model size to problem complexity (terms and/or digits).
- Update the example to support a variable number of terms in a given instance and tune a model to get 100% accuracy.
- Add support for other math operations like subtraction, division, and multiplication.

Post your extensions online and share the link with me; I'd love to see what you come up with!

9.11 Summary

In this lesson, you discovered how to develop an Encoder-Decoder LSTM model. Specifically, you learned:

- The Encoder-Decoder LSTM architecture and how to implement it in Keras.
- The addition sequence-to-sequence prediction problem.