2
Building Your First RNN with TensorFlow

In this chapter, you will gain a hands-on experience of building a **recurrent neural network** (**RNN**). First, you will be introduced to the most widely used machine learning library—TensorFlow. From learning the basics to advancing into some fundamental techniques, you will obtain a reasonable understanding of how to apply this powerful library to your applications. Then, you will take on a fairly simple task of building an actual model. The process will show you how to prepare your data, train the network, and make predictions.

In summary, the topics of this chapter include the following:

- **What are you going to build?**: Introduction of your task
- **Introduction to TensorFlow**: Taking first steps into learning the TensorFlow framework
- **Coding the RNN**: You will go through the process of writing your first neural network using TensorFlow. This includes all steps required for a finished solution

The prerequisites for this chapter are basic Python programming knowledge and decent understanding of recurrent neural networks captured in the `Chapter 1`, *Introducing Recurrent Neural Networks*. After reading this chapter, you should have a full understanding of how to use TensorFlow with Python and how easy and straightforward it is to build a neural network.

What are you going to build?

Your first steps into the practical world of recurrent neural networks will be to build a simple model which determines the parity (`http://mathworld.wolfram.com/Parity.html`) of a bit sequence . This is a warm-up exercise released by OpenAI in January 2018 (`https://blog.openai.com/requests-for-research-2/`). The task can be explained as follows:

Given a binary string of a length of `50`, determine whether there is an even or odd number of ones. If that number is even, output `0`, otherwise `1`.

Later in this chapter, we will give a detailed explanation of the solution, together with addressing the difficult parts and how to tackle them.

Introduction to TensorFlow

TensorFlow is an open source library built by Google, which aims to assist developers in creating machine learning models of any kind. The recent improvements in the deep learning space created the need for an easy and fast way of building neural networks. TensorFlow addresses this problem in an excellent fashion, by providing a wide range of APIs and tools to help developers focus on their specific problem, rather than dealing with mathematical equations and scalability issues.

TensorFlow offers two main ways of programming a model:

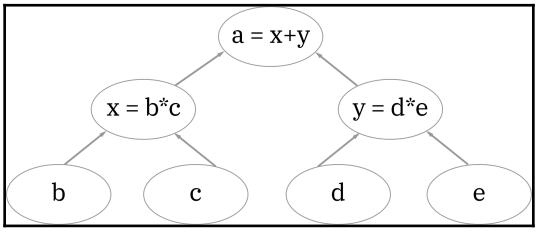
- Graph-based execution
- Eager execution

Graph-based execution

Graph-based execution is an alternative way of representing mathematical equations and functions. Considering the expression *a = (b*c) + (d*e)*, we can use a graph representation as follows:

1. Separate the expression into the following:
    - *x = b*c*
    - *y = d*e*
    - *a = x+y*

2. Build the following graph:



As you can see from the previous example, using graphs lets compute two equations in parallel. This way, the code can be distributed among multiple CPUs/GPUs.

More complex variants of that example are used in TensorFlow for training heavy models. Following that technique, TensorFlow graph-based execution requires a two-step approach when building your neural network. One should first construct the graph architecture and then execute it to receive results.

This approach makes your application run faster and it will be distributed across multiple CPUs, GPUs, and so on. Unfortunately, some complexity comes along with it. Understanding how this way of programming work, and the inability to debug your code in the already familiar way (for example, printing values at any point in your program) makes the graph-based execution (see for more details `http://smahesh.com/blog/2017/07/10/understanding-tensorflow-graph/`) a bit challenging for beginners.

Even though this technique may introduce a new way of programming, our examples will be based upon it. The reason behind this decision lies in the fact that there are many more resources out there and almost every TensorFlow example you come across is graph-based. In addition, I believe it is of vital importance to understand the fundamentals, even if they introduce unfamiliar techniques.

Eager execution

Eager execution is an approach, recently introduced by Google, which, as stated in the documentation (`https://www.tensorflow.org/guide/eager`), uses the following:

> *An imperative programming environment that evaluates operations immediately, without building graphs: operations return concrete values instead of constructing a computational graph to run later. This makes it easy to get started with TensorFlow and debug models, and it reduces boilerplate as well.*

As you can see, there is no overhead to learning the new programming technique and debugging is seamless. For a better understanding, I recommend checking this tutorial from the TensorFlow Conference 2018 (`https://www.youtube.com/watch?v=T8AW0fKP0Hs`).

I must state that, once you learn how to manipulate the TF API, building models becomes really easy on both graph-based and eager execution. Don't panic if the former seems complicated at first—I can assure you that it is worth investing the time to understand it properly.

Coding the recurrent neural network

As mentioned before, the aim of our task is to build a recurrent neural network that predicts the parity of a bit sequence. We will approach this problem in a slightly different way. Since the parity of a sequence depends on the number of ones, we will sum up the elements of the sequence and find whether the result is even or not. If it is even, we will output `0`, otherwise, `1`.

This section of the chapter includes code samples and goes through the following steps:

- Generating data to train the model
- Building the TensorFlow graph (using TensorFlow's built-in functions for recurrent neural networks)
- Training the neural network with the generated data
- Evaluating the model and determining its accuracy

Generating data

Let's revisit the OpenAI's task (`https://blog.openai.com/requests-for-research-2/`). As stated there, we need to generate a dataset of random 100,000 binary strings of length 50. In other words, our training set will be formed of 100,000 examples and the recurrent neural network will accept 50 time steps. The result of the last time step would be counted as the model prediction.

The task of determining the sum of a sequence can be viewed as a classification problem where the result can be any of the classes from `0` to `50`. A standard practice in machine learning is to encode the data into an easily decodable numeric way. But why is that? Most machine learning algorithms cannot accept anything apart from numeric data, so we need to always encode our input/output. This means that, our predictions will also come out in an encoding format. Thus, it is vital to understand the actual value behind these predictions. This means that we need to be able to easily decode them into a human understandable format. A popular way of encoding data for classification problems is one-hot encoding.

Here is an example of that technique.

Imagine the predicted output for a specific sequence is `30`. We can encode this number by introducing a `1x50` array where all numbers, except the one in the 30th position, are `0s - [0, 0,..., 0, 1, 0, ..., 0, 0, 0]`.

Before preparing the actual data, we need to import all of the necessary libraries. To do that, follow this link (`https://www.python.org/downloads/`) to install Python on your machine. In your command-line/Terminal window install the following packages:

```
pip3 install tensorflow
```

After you have done that, create a new file called `ch2_task.py` and import the following libraries:

```
import tensorflow as tf
import random
```

Preparing the data requires an input and output value. The input value is a three-dimensional array of a size of `[100000, 50, 1]`, with `100000` items, each one containing `50` one-element arrays (either `0` or `1`), and is shown in the following example:

```
[ [ [1], [0], [1], [1], …, [0], [1] ]
[ [0], [1], [0], [1], …, [0], [1] ]
[ [1], [1], [1], [0], …, [0], [0] ]
[ [1], [0], [0], [0], …, [1], [1] ] ]
```

The following example shows the implementation:

```
num_examples = 100000
num_classes = 50

def input_values():
    multiple_values = [map(int, '{0:050b}'.format(i)) for i in
range(2**20)]
    random.shuffle(multiple_values)
```

```
    final_values = []
    for value in multiple_values[:num_examples]:
        temp = []
        for number in value:
            temp.append([number])
        final_values.append(temp)
    return final_values
```

Here, `num_classes` is the number of time steps in our RNN (`50`, in this example). The preceding code returns a list with the `100000` binary sequences. The style is not very Pythonic but, written in this way, it makes it easy to follow and understand.

First, we start with initializing the `multiple_values` variable. It contains a binary representation of the first 2^{20} `= 1,048,576` numbers, where each binary number is padded with zeros to accommodate the length of `50`. Obtaining so many examples minimizes the chance of similarity between any two of them. We use the `map` function together with `int` in order to convert the produced string into a number.

Here is a quick example of how this works. We want to represent the number `2` inside the `multiple_values` array. The binary version of `2` is `'10'`, so the string produced after `'{0:050b}'.format(i)` where `i = 2`, is `'00000000000000000000000000000000000000000000000010'` (48 zeros at the front to accommodate a length of `50`). Finally, the `map` function makes the previous string into a number without removing the zeros at the front.

Then, we shuffle the `multiple_values` array, assuring difference between neighboring elements. This is important during backpropagation when the network is trained, because we are iteratively looping throughout the array and training the network at each step using a single example. Having similar values next to each other inside the array may produce biased results and incorrect future predictions.

Finally, we enter a loop, which traverses over all of the binary elements and builds an array similar to the one we saw previously. An important thing to note is the usage of `num_examples`, which slices the array, so we pick only the first `100,000` values.

The second part of this section shows how to generate the expected output (the sum of all the elements in each list from the input set). These outputs are used to evaluate the model and tune the `weight`/`biases` during backpropagation. The following example shows the implementation:

```
def output_values(inputs):
    final_values = []
    for value in inputs:
        output_values = [0 for _ in range(num_classes)]
```

```
            count = 0
            for i in value:
                count += i[0]
            if count < num_classes:
                output_values[count] = 1
            final_values.append(output_values)
        return final_values
```

The `inputs` parameter is a result of `input_values()` that we declared earlier. The `output_values()` function returns a list of one-hot encoded representations of each member in `inputs`. If the sum of all of the elements in the `[[0], [1], [1], [1], [0], ..., [0], [1]]` sequence is `48`, then its corresponding value inside `output_values` is `[0, 0, 0, ..., 1, 0, 0]` where `1` is at position `48`.

Finally, we use the `generate_data()` function to obtain the final values for the network's input and output, as shown in the following example:

```
def generate_data():
    inputs = input_values()
    return inputs, output_values(inputs)
```

We use the previous function to create these two new variables: `input_values`, and `output_values = generate_data()`. One thing to pay attention to is the dimensions of these lists:

- `input_values` is of a size of `[num_examples, num_classes, 1]`
- `output_values` is of a size of `[num_examples, num_classes]`

Where `num_examples = 100000` and `num_classes = 50`.

Building the TensorFlow graph

Constructing the TensorFlow graph is probably the most complex part of building a neural network. We will precisely examine all of the steps so you can obtain a full understanding.

The TensorFlow graph can be viewed as a direct implementation of the recurrent neural network model, including all equations and algorithms introduced in `Chapter 1`, *Introducing Recurrent Neural Networks*.

First, we start with setting the parameters of the model, as shown in the following example:

```
X = tf.placeholder(tf.float32, shape=[None, num_classes, 1])
Y = tf.placeholder(tf.float32, shape=[None, num_classes])
num_hidden_units = 24
weights = tf.Variable(tf.truncated_normal([num_hidden_units, num_classes]))
biases = tf.Variable(tf.truncated_normal([num_classes]))
```

`X` and `Y` are declared as `tf.placeholder`, which inserts a placeholder (inside the graph) for a tensor that will be always fed. Placeholders are used for variables that expect data when training the network. They often hold values for the training input and expected output of the network. You might be surprised why one of the dimensions is `None`. The reason is that we have trained the network using batches. These are collections of several elements from our training data stacked together. When specifying the dimension as None, we let the tensor decide this dimension, calculating it using the other two values.

According to the TensorFlow documentation: A tensor is a generalization of vectors and matrices to potentially higher dimensions. Internally, TensorFlow represents tensors as n-dimensional arrays of base datatypes.

When performing training using batches, we split the training data into several smaller arrays of size—`batch_size`. Then, instead of training the network with all examples at once, we use one batch at a time.

The advantages of this are less memory is required and faster learning is achieved.

The `weight` and `biases` are declared as `tf.Variable`, which holds a certain value during training. This value can be modified. When a variable is first introduced, one should specify an initial value, type, and shape. The type and shape remain constant and cannot be changed.

Next, let's build the RNN cell. If you recall from `Chapter 1`, *Introducing Recurrent Neural Networks*, an input at time step, *t*, is plugged into an RNN cell to produce an output, y_t, and a hidden state, h_t. Then, the hidden state and the new input at time step (`t+1`) are plugged into a new RNN cell (which shares the same weights and biases as the previous). It produces its own output, y_{t+1}, and hidden state, h_{t+1}. This pattern is repeated for every time step.

With TensorFlow, the previous operation is just a single line:

```
rnn_cell = tf.contrib.rnn.BasicRNNCell(num_units=num_hidden_units)
```

As you already know, each cell requires an activation function that is applied to the hidden state. By default, TensorFlow chooses **tanh** (perfect for our use case) but you can specify any that you wish. Just add an additional parameter called `activation`.

Both in `weights` and in `rnn_cell`, you can see a parameter called `num_hidden_units`. As stated here (`https://stackoverflow.com/questions/37901047/what-is-num-units-in-tensorflow-basiclstmcell`), the `num_hidden_units` is a direct representation of the learning capacity of a neural network. It determines the dimensionality of both the memory state, h_t, and the output, y_t.

The next step is to produce the output of the network. This can also be implemented with a single line:

```
outputs, state = tf.nn.dynamic_rnn(rnn_cell, inputs=X,
dtype=tf.float32)
```

Since `X` is a batch of input sequences, then `outputs` represents a batch of outputs at every time step in all sequences. To evaluate the prediction, we need the value of the last time step for every output in the batch. This happens in three steps, explained in the following bulleted examples:

- We get the values from the last time step: `outputs = tf.transpose(outputs, [1, 0, 2])`

This would reshape the output's tensor from (`1000, 50, 24`) to (`50, 1,000, 24`) so that the outputs from the last time step in every sequence are accessible to be gathered using the following: `last_output = tf.gather(outputs, int(outputs.get_shape()[0]) - 1)`.

Let's review the following diagram to understand how this `last_output` is obtained:

The previous diagram shows how one input example of `50` steps is plugged into the network. This operation should be done `1,000` times for each individual example having `50` steps but, for the sake of simplicity, we are showing only one example.

After iteratively going through each time step, we produce `50` outputs, each one having the dimensions (`24, 1`). So, for one example of `50` input time steps, we produce `50` output steps. Presenting all of the outputs mathematically results in a (`1,000, 50, 24`) matrix. The height of the matrix is `1,000`—the number of individual examples. The width of the matrix is `50`—the number of time steps for each example. The depth of the matrix is `24`—the dimension of each element.

To make a prediction, we only care about `output_last` at each example, and since the number of examples is `1,000`, we only need 1,000 output values. As seen in the previous example, we transpose the matrix (`1000, 50, 24`) into (`50, 1000, 24`), which will make it easier to get `output_last` from each example. Then, we use `tf.gather` to obtain the `last_output` tensor which has size of (`1000, 24, 1`).

Final lines of building our graph include:

- We predict the output of the particular sequence:

```
prediction = tf.matmul(last_output, weights) + biases
```

Using the newly obtained tensor, `last_output` , we can calculate a prediction using the weights and biases.

- We evaluate the output based on the expected value:

```
loss = tf.nn.softmax_cross_entropy_with_logits_v2(labels=Y,
logits=prediction)
total_loss = tf.reduce_mean(loss)
```

We can use the popular cross entropy loss function in a combination with `softmax`. If you recall from `Chapter 1`, *Introducing Recurrent Neural Networks*, the `softmax` function transforms a tensor to emphasize the largest values and suppress values that are significantly below the maximum value. This is done by normalizing the values from the initial array to ones that add up to `1`. For example, the input `[0.1, 0.2, 0.3, 0.4, 0.1, 0.2, 0.3]` becomes [`0.125, 0.138, 0.153, 0.169, 0.125, 0.138, 0.153`]. The cross entropy is a loss function that computes the difference between the `label` (expected values) and `logits` (predicted values).

Since `tf.nn.softmax_cross_entropy_with_logits_v2` returns a 1-D tensor of a length of `batch_size` (declared below), we use `tf.reduce_mean` to compute the mean of all elements in that tensor.

As a final step, we will see how TensorFlow makes it easy for us to optimize the weights and biases. Once we have obtained the loss function, we need to perform a backpropagation algorithm, adjusting the weights and biases to minimize the loss. This can be done in the following way:

```
learning_rate = 0.001
optimizer =
tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(loss=total_los
s)
```

`learning_rate` is one of the model's hyperparameters and is used when optimizing the loss function. Tuning this value is essential for better performance, so feel free to adjust it and evaluate the results.

Minimizing the error of the loss function is done using an Adam optimizer. Here (`https://stats.stackexchange.com/questions/184448/difference-between-gradientdescentoptimizer-and-adamoptimizer-tensorflow`) is a good explanation of why it is preferred over the Gradient descent.

We have just built the architecture of our recurrent neural network. Let's put everything together, as shown in the following example:

```
X = tf.placeholder(tf.float32, shape=[None, num_classes, 1])
Y = tf.placeholder(tf.float32, shape=[None, num_classes])

num_hidden_units = 24

weights = tf.Variable(tf.truncated_normal([num_hidden_units, num_classes]))
biases = tf.Variable(tf.truncated_normal([num_classes]))

rnn_cell = tf.contrib.rnn.BasicRNNCell(num_units=num_hidden_units,
activation=tf.nn.relu)
outputs1, state = tf.nn.dynamic_rnn(rnn_cell, inputs=X, dtype=tf.float32)
outputs = tf.transpose(outputs1, [1, 0, 2])

last_output = tf.gather(outputs, int(outputs.get_shape()[0]) - 1)
prediction = tf.matmul(last_output, weights) + biases

loss = tf.nn.softmax_cross_entropy_with_logits_v2(labels=Y,
logits=prediction)
total_loss = tf.reduce_mean(loss)

learning_rate = 0.001
optimizer =
tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(loss=total_los
s)
```

The next task is to train the neural network using the TensorFlow graph in combination with the previously generated data.

Training the RNN

In this section, we will go through the second part of a TensorFlow program—executing the graph with a predefined data. For this to happen, we will use the `Session` object, which encapsulates an environment in which the tensor objects are executed.

The code for our training is shown in the following example:

```
batch_size = 1000
number_of_batches = int(num_examples/batch_size)
epoch = 100
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    X_train, y_train = generate_data()
    for epoch in range(epoch):
        iter = 0
        for _ in range(number_of_batches):
            training_x = X_train[iter:iter+batch_size]
            training_y = y_train[iter:iter+batch_size]
            iter += batch_size
            _, current_total_loss = sess.run([optimizer, total_loss],
            feed_dict={X: training_x, Y: training_y})
            print("Epoch:", epoch, "Iteration:", iter, "Loss",
current_total_loss)
            print("____________________")
```

First, we initialize the batch size. At each training step, the network is tuned, based on examples from the chosen batch. Then, we compute the number of batches as well as the number of epochs—this determines how many times our model should loop through the training set. `tf.Session()` encapsulates the code in a TensorFlow `Session` and `sess.run(tf.global_variables_initializer())` (`https://stackoverflow.com/questions/44433438/understanding-tf-global-variables-initializer`) makes sure all variables hold their values.

Then, we store an individual batch from the training set in `training_x` and `training_y` .

The last, and most, important part of training the network comes with the usage of `sess.run()` . By calling this function, you can compute the value of any tensor. In addition, one can specify as many arguments as you want by ordering them in a list—in our case, we have specified the optimizer and loss function. Remember how, while building the graph, we created placeholders for holding the values of the current batch? These values should be mentioned in the `feed_dict` parameter when running `Session`.

Training this network can take around four or five hours. You can verify that it is learning by examining the value of the loss function. If its value decreases, then the network is successfully modifying the weights and biases. If the value is not decreasing, you most likely need to make some additional changes to optimize the performance. These will be explained in `Chapter 6`, *Improving Your RNN Performance*.

Evaluating the predictions

Testing the model using a fresh new example can be accomplished in the following way:

```
prediction_result = sess.run(prediction, {X: test_example})
largest_number_index = prediction_result[0].argsort()[-1:][::-1]

print("Predicted sum: ", largest_number_index, "Actual sum:", 30)
print("The predicted sequence parity is ", largest_number_index % 2, " and
it should be: ", 0)
```

This is where `test_example` is an array of a size of (`1 x num_classes x 1`).

Let `test_example` be as follows:

```
[[[1],[0],[0],[1],[1],[0],[1],[1],[1],[0],[1],[0],[0],[1],[1],[0],[1],[1],[
1],[0],
[1],[0],[0],[1],[1],[0],[1],[1],[1],[0],[1],[0],[0],[1],[1],[0],[1],[1],[1]
,[0],
[1],[0],[0],[1],[1],[0],[1],[1],[1],[0]]]
```

The sum of all elements in the above array is equal to `30`. With the last line, `prediction_result[0].argsort()[-1:][::-1]`, we can find the index of the largest number. The index would tell us the sum of the sequence. As a last step, we need to find the remainder when this number is divided by `2`. This will give us the parity of the sequence.

Both training and evaluation are done together after you run `python3 ch2_task.py`. If you want to only do evaluation, comment out the lines between `70` and `91` from the program and run it again.

Summary

In this chapter, you explored how to build a simple recurrent neural network to solve the problem of identifying sequence parity. You obtained a brief understanding of the TensorFlow library and how it can be utilized for building deep learning models. I hope the study of this chapter leaves you more confident in your deep learning knowledge, as well as excited to learn and grow more in this field.

In the next chapter, you will go a step further by implementing a more sophisticated neural network for the task of generating text. You will gain both theoretical and practical experience. This will result in you learning about a new type of network, GRU, and understanding how to implement it in TensorFlow. In addition, you will face the challenge of formatting your input text correctly as well as using it for training the TensorFlow graph.

I can assure you that an exciting learning experience is coming, so I cannot wait for you to be part of it.

External links

- Parity: `http://mathworld.wolfram.com/Parity.html`
- Request for Research 2.0 by OpenAI: `https://blog.openai.com/requests-for-research-2/`
- Eager execution documentation: `https://www.tensorflow.org/guide/eager`
- Eager execution (TensorFlow Conference 2018): `https://www.youtube.com/watch?v=T8AW0fKP0Hs`
- Python installation: `https://www.python.org/downloads/`
- Understanding `num_hidden_units` : `https://stackoverflow.com/questions/37901047/what-is-num-units-in-tensorflow-basiclstmcell`
- Adam versus Gradient descent optimizer: `https://stats.stackexchange.com/questions/184448/difference-between-gradientdescentoptimizer-and-adamoptimizer-tensorflow`
- Understanding `sess.run(tf.global_variables_initializer()`: `https://stackoverflow.com/questions/44433438/understanding-tf-global-variables-initializer`