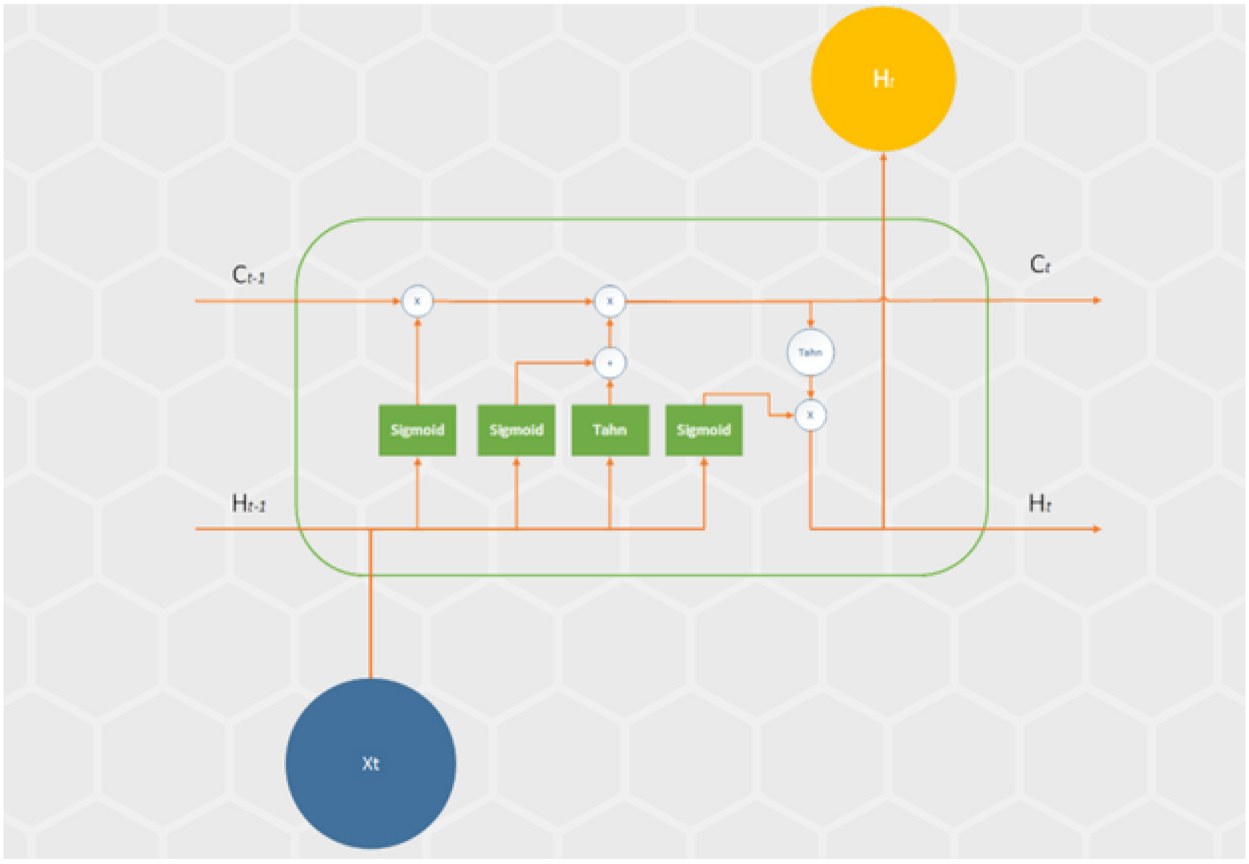
RUBIK'S CODE Freedom. Wisdom. Excellence.

Two Ways to Implement LSTM Network using Python – with TensorFlow and Keras

MARCH 26, 2018 — 1 COMMENT

In the **previous article**, we talked about the way that powerful type of Recurrent Neural Networks – Long Short-Term Memory (LSTM) Networks function. They are not keeping just propagating output information to the next time step, but they are also storing and propagating the state of the so-called LSTM cell. This cell is holding four neural networks inside – gates, which are used to decide which information will be stored in cell state and pushed to output. So, the output of the network at one time step is not depending only on the previous time step but depends on *n* previous time steps.



Ok, that is enough to get us up to speed with theory, and prepare us for the practical part – implementation of this kind of networks. If however, you want to learn more about Long Short-Term Memory Networks, you can do it **here**. In this article, we will consider two similar language modeling problems and solve them using two different APIs. Firstly we will create the network that will be able to predict words based on the provided peace of text and for this purpose, we will use **TensorFlow**. In second implementation we will be classifying reviews from the IMDB dataset using the **Keras**.

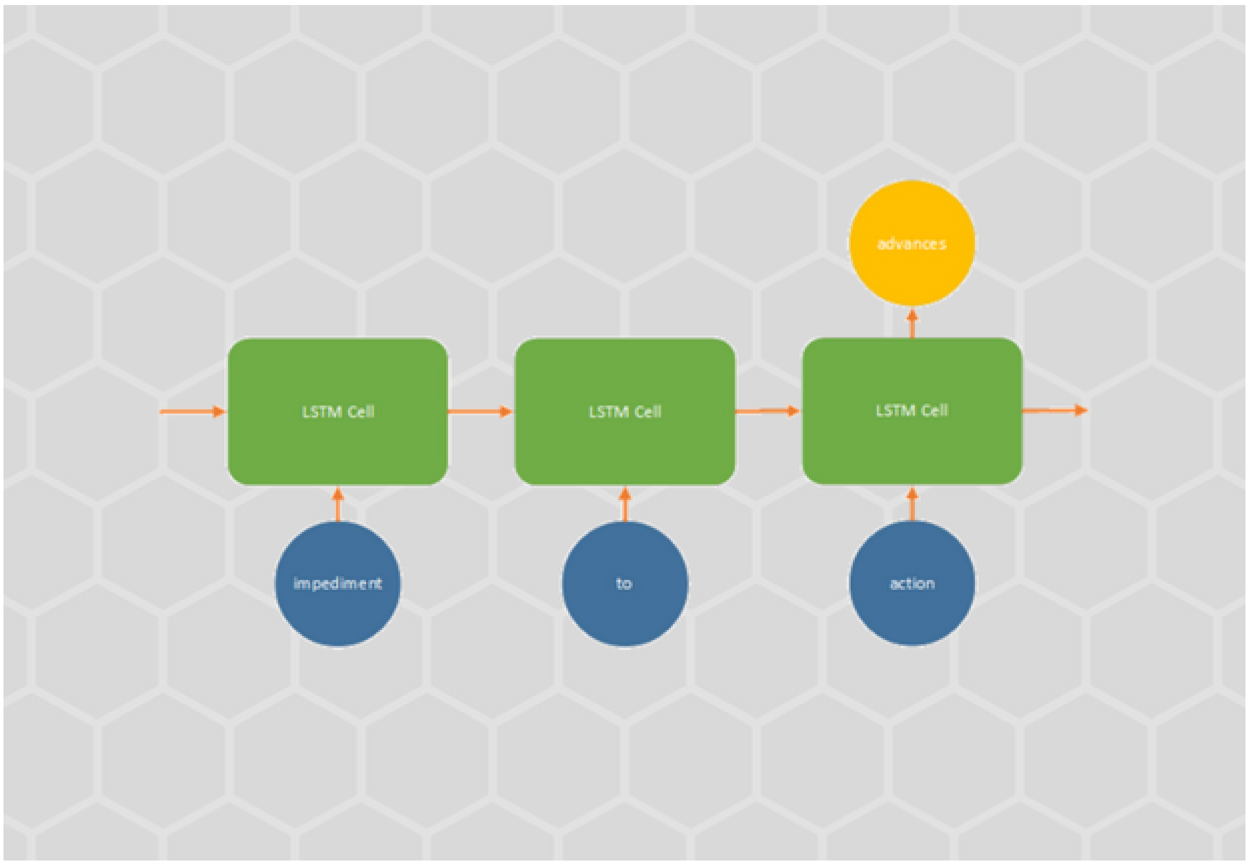
Before we wander off into the problem we are solving and the code itself make sure to setup your environment. As in all previous articles from this **series**, I will be using Python 3.6. Also, I am using Anaconda and Spyder, but you can use any IDE that you prefer. However, the important thing to do is to install Tensorflow and Keras. Instructions for installing and using TensorFlow can be found **here**, while instructions for installing and using Keras are **here**.

The Problem for Tensorflow Implementation

Let's say that we want to train one LSTM to predict the next word using a sample text. Simple text in our example will be one of the favorite sections of mine from **Marcus Aurelius – Meditations**:

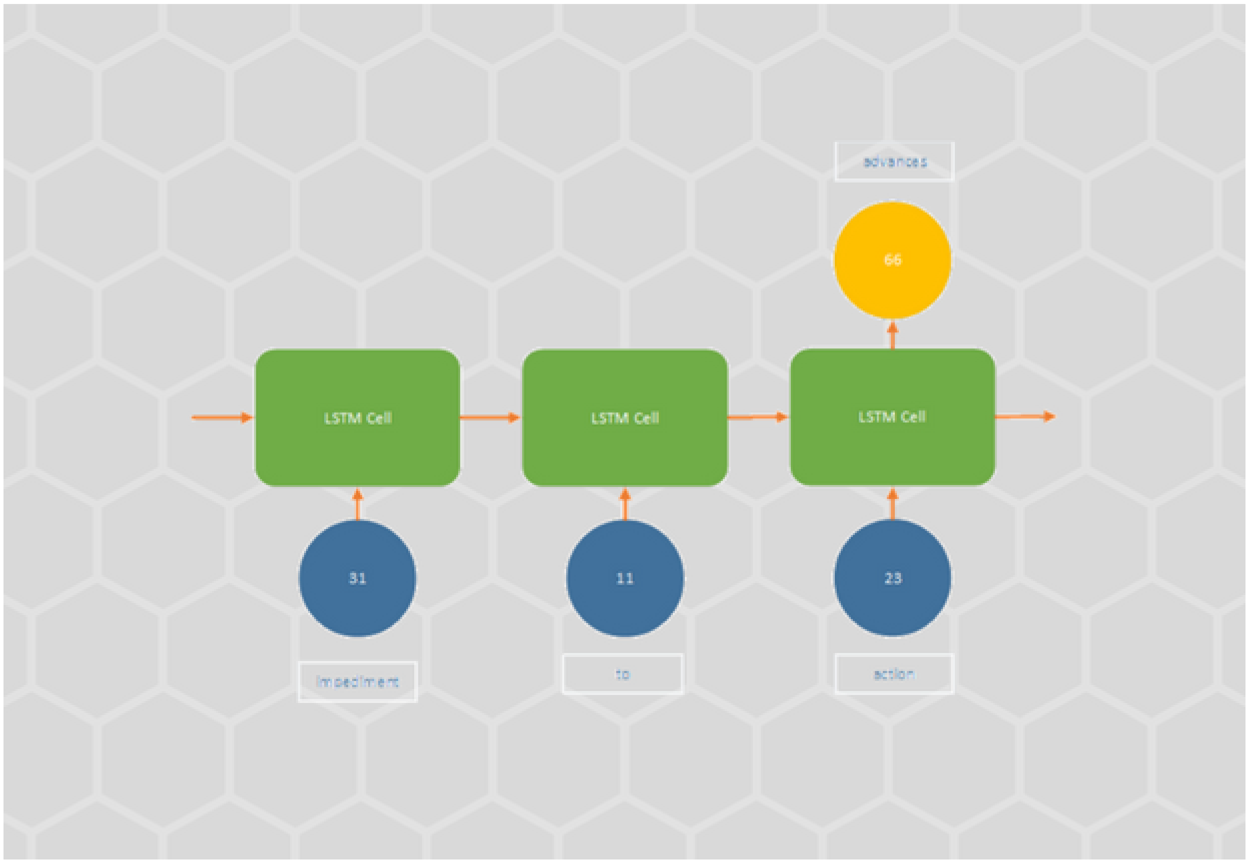
> *In a sense , people are our proper occupation . Our job is to do them good and put up with them . But when they obstruct our proper tasks , they become irrelevant to us—like sun , wind, animals . Our actions may be impeded by them , but there can be no impeding our intentions or our dispositions . Because we can accommodate and adapt . The mind adapts and converts to its own purposes the obstacle to our acting . The impediment to action advances action . What stands in the way becomes the way .*

Note that this text is a bit modified. Every punctuation mark is surrounded by space characters, ergo every word and every punctuation mark are considered as a symbol. To demonstrate how LSTM Networks work, we will use simplified process. We will push sequences of three symbols as inputs and one output. By repeating this process, the network will learn how to predict next word based on three previous ones.



Tensorflow Implementation

So, we have our plan of attack: provide a sequence of three symbols and one output to the LSTM Network and learn it to predict that output. However, since we are using mathematical models first thing we need to do is to prepare this data (text) for any kind of operation. These networks understand numbers, not strings as we have it in our text, so we need to convert these symbols into numbers. We will do this by assigning a unique integer to each symbol based the on the frequency of occurrence.

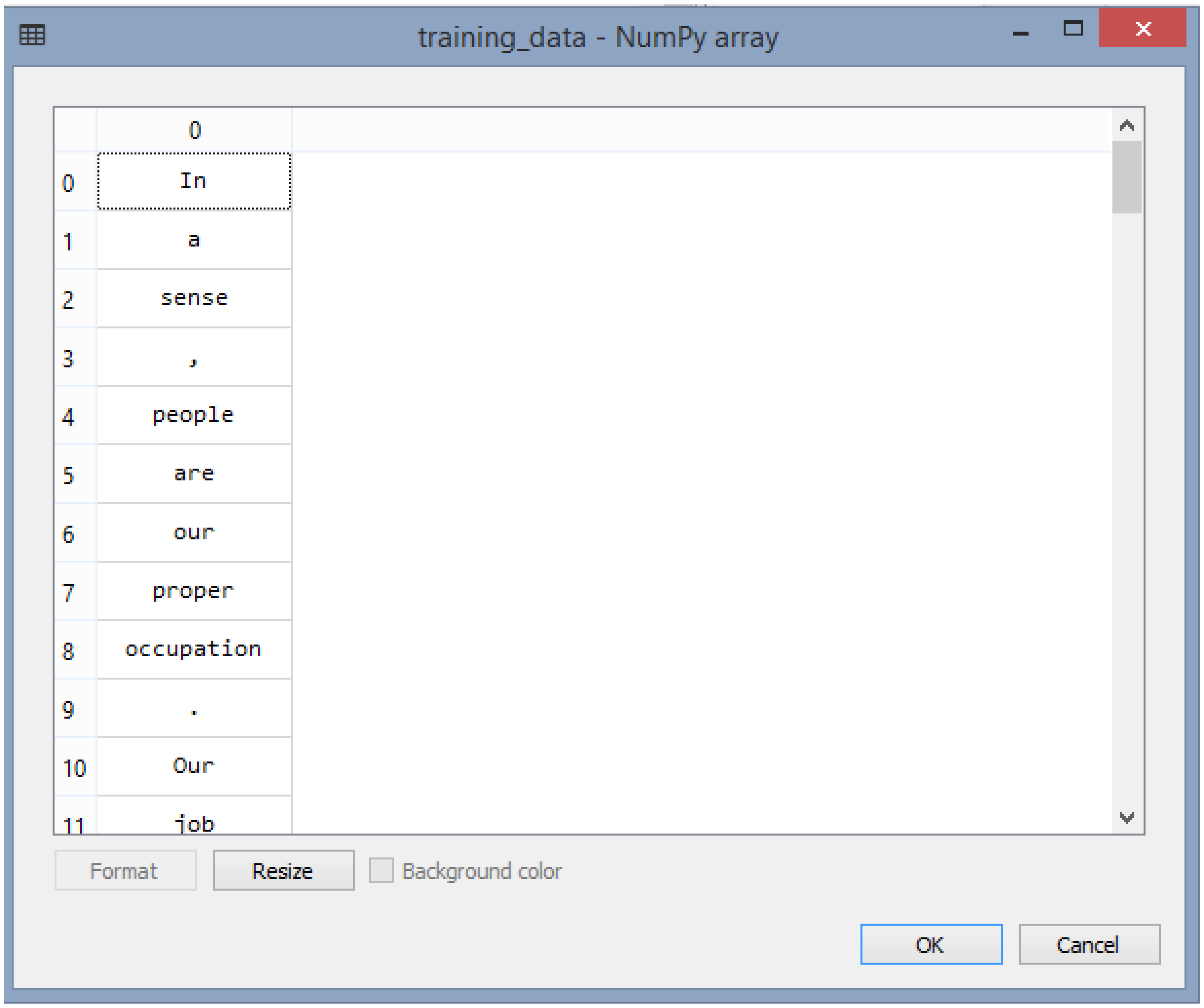


For this purpose, we will use *DataHandler* class. This class has two purposes, to load the data from the file, and to assign a number to each symbol. Here is the code:

```
import numpy as np
import collections

class DataHandler:
    def read_data(self, fname):
        with open(fname) as f:
            content = f.readlines()
        content = [x.strip() for x in content]
        content = [content[i].split() for i in range(len(content))]
        content = np.array(content)
        content = np.reshape(content, [-1, ])
        return content

    def build_datasets(self, words):
        count = collections.Counter(words).most_common()
        dictionary = dict()
        for word, _ in count:
            dictionary[word] = len(dictionary)
        reverse_dictionary = dict(zip(dictionary.values(), dictionary.keys()))
        return dictionary, reverse_dictionary
```

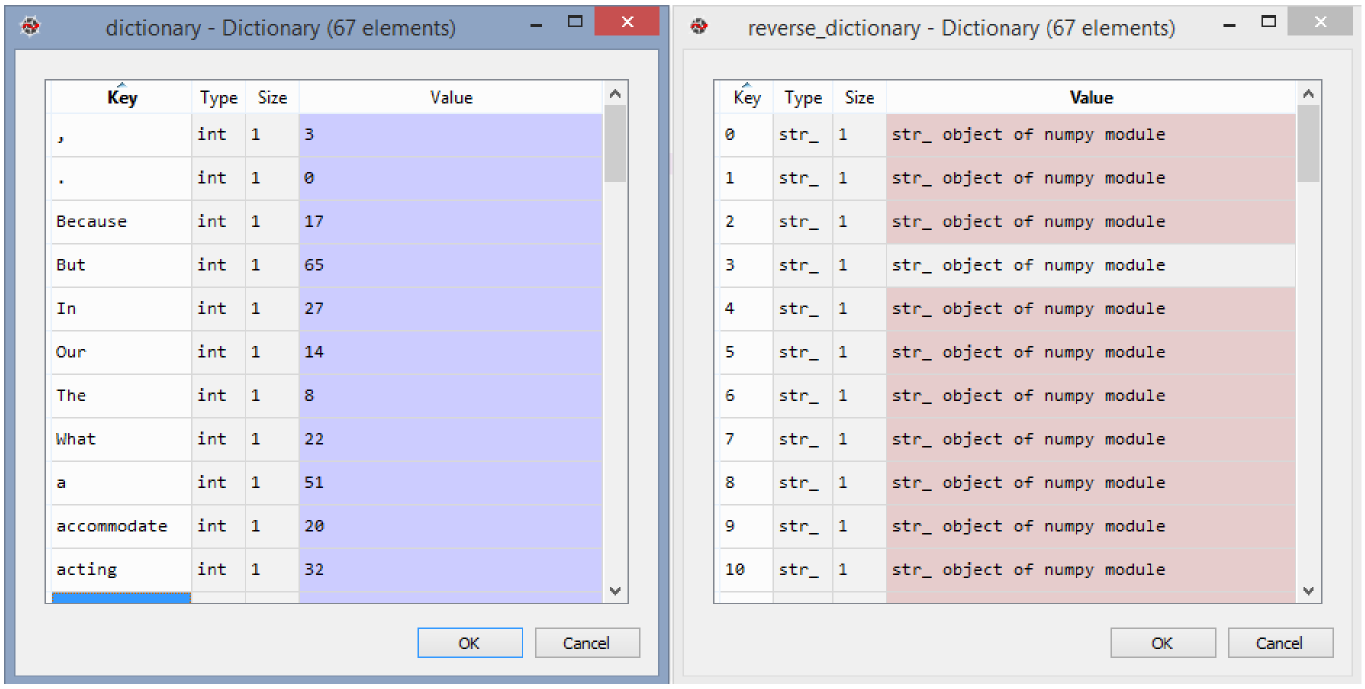
data_handler.py hosted with ❤ by GitHub view raw

The first method of this class *read_data* is used to read text from the defined file and create an array of symbols. Here is how that looks like once called on the sample text:



	0
0	In
1	a
2	sense
3	,
4	people
5	are
6	our
7	proper
8	occupation
9	.
10	Our
11	job

The second method *build_datasets* is used for creating two dictionaries. The first dictionary labeled as just *dictionary* contains symbols as keys and their corresponding number as a value. Second dictionary *reverse_dictionary* contains the same information, just keys are numbers and values are the symbols themselves. This is how they will look like created using the sample text we are using in this example:


dictionary - Dictionary (67 elements)

Key	Type	Size	Value
,	int	1	3
.	int	1	0
Because	int	1	17
But	int	1	65
In	int	1	27
Our	int	1	14
The	int	1	8
What	int	1	22
a	int	1	51
accommodate	int	1	20
acting	int	1	32

reverse_dictionary - Dictionary (67 elements)

Key	Type	Size	Value
0	str_	1	str_ object of numpy module
1	str_	1	str_ object of numpy module
2	str_	1	str_ object of numpy module
3	str_	1	str_ object of numpy module
4	str_	1	str_ object of numpy module
5	str_	1	str_ object of numpy module
6	str_	1	str_ object of numpy module
7	str_	1	str_ object of numpy module
8	str_	1	str_ object of numpy module
9	str_	1	str_ object of numpy module
10	str_	1	str_ object of numpy module

Awesome! Our data is ready for use. Off to the fun part, the creation of the model. For this purpose, we will create a new class that will be able to generate LSTM network based on the passed parameters. Take a look:

```python
import tensorflow as tf
from tensorflow.contrib import rnn

class RNNGenerator:
    def create_LSTM(self, inputs, weights, biases, seq_size, num_units):
        # Reshape input to [1, sequence_size] and split it into sequences
        inputs = tf.reshape(inputs, [-1, seq_size])
        inputs = tf.split(inputs, seq_size, 1)

        # LSTM with 2 layers
        rnn_model = rnn.MultiRNNCell([rnn.BasicLSTMCell(num_units),rnn.BasicLSTMCell(num_unit

        # Generate prediction
        outputs, states = rnn.static_rnn(rnn_model, inputs, dtype=tf.float32)

        return tf.matmul(outputs[-1], weights['out']) + biases['out']
```

rnn_generator.py hosted with ❤ by **GitHub** **view raw**

We imported some important classes there: TensorFlow itself and *rnn* class form *tensorflow.contrib*. Since our LSTM Network is a subtype of RNNs we will use this to create our model. Firstly, we reshaped our input and then split it into sequences of three symbols. Then we created the model itself.

We created two LSTM layers using *BasicLSTMCell* method. Each of these layers has a number of units defined by the parameter *num_units*. Apart from that, we use *MultiRNNCell* to combine these two layers in one network. Then we used *static_rnn* method to construct the network and generate the predictions.

In the end, we will use *SessionRunner* class. This class contains environment in which our model will be run and evaluated. Here is how the code looks like:

```
import tensorflow as tf
import random
import numpy as np

class SessionRunner():
    training_iters = 50000

    def __init__(self, optimizer, accuracy, cost, lstm, initilizer, writer):
        self.optimizer = optimizer
        self.accuracy = accuracy
        self.cost = cost
        self.lstm = lstm
        self.initilizer = initilizer
        self.writer = writer

    def run_session(self, x, y, n_input, dictionary, reverse_dictionary, training_data):

        with tf.Session() as session:
            session.run(self.initilizer)
            step = 0
            offset = random.randint(0, n_input + 1)
            acc_total = 0

            self.writer.add_graph(session.graph)

            while step < self.training_iters:
                if offset > (len(training_data) - n_input - 1):
                    offset = random.randint(0, n_input+1)

                sym_in_keys = [ [dictionary[ str(training_data[i])]] for i in range(offset, o
                sym_in_keys = np.reshape(np.array(sym_in_keys), [-1, n_input, 1])

                sym_out_onehot = np.zeros([len(dictionary)], dtype=float)
                sym_out_onehot[dictionary[str(training_data[offset+n_input])]] = 1.0
                sym_out_onehot = np.reshape(sym_out_onehot,[1,-1])
```

```

                    _, acc, loss, onehot_pred = session.run([self.optimizer, self.accuracy, self.
                    acc_total += acc

                    if (step + 1) % 1000 == 0:
                        print("Iteration = " + str(step + 1) + ", Average Accuracy= " + "{:.2f}%"
                        acc_total = 0
                    step += 1
                    offset += (n_input+1)
```

session_runner.py hosted with ❤ by GitHub view raw

50000 iterations are being run using our model. We injected model, optimizer, loss function, etc. in the constructor, so the class has this information available. Of course, the first thing we need to do is slice up the data in the provided dictionary, and make encoded outputs (*sym_in_keys* and *sym_out_onehot*, respectively). Also, we are pushing random sequences in the model so we avoid overfitting. This is handled by *offset* variable. Finally, we will run the session and get accuracy. Don't get confused by final if statement in the code it is used just for display purposes (at every 1000 iterations present the average accuracy).

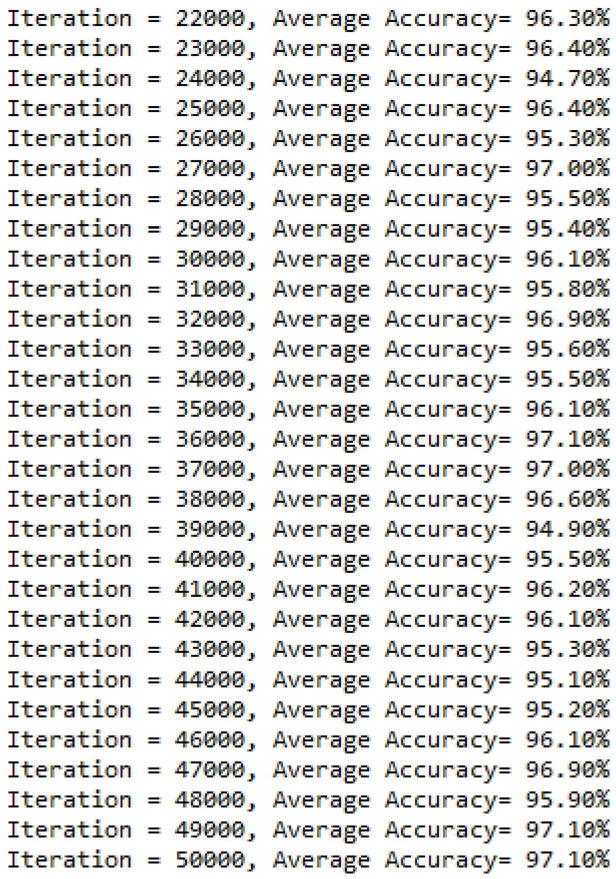
Our main script combines all this into one, and here is how it looks like:

```
import tensorflow as tf
from DataHandler import DataHandler
from RNN_generator import RNNGenerator
from session_runner import SessionRunner

log_path = '/output/tensorflow/'
writer = tf.summary.FileWriter(log_path)

# Load and prepare data
data_handler = DataHandler()

training_data =  data_handler.read_data('meditations.txt')

dictionary, reverse_dictionary = data_handler.build_datasets(training_data)

# TensorFlow Graph input
n_input = 3
n_units = 512

x = tf.placeholder("float", [None, n_input, 1])
y = tf.placeholder("float", [None, len(dictionary)])
```

```

# RNN output weights and biases
weights = {
    'out': tf.Variable(tf.random_normal([n_units, len(dictionary)]))
}
biases = {
    'out': tf.Variable(tf.random_normal([len(dictionary)]))
}

rnn_generator = RNNGenerator()
lstm = rnn_generator.create_LSTM(x, weights, biases, n_input, n_units)

# Loss and optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=lstm, labels=y))
optimizer = tf.train.RMSPropOptimizer(learning_rate=0.001).minimize(cost)

# Model evaluation
correct_pred = tf.equal(tf.argmax(lstm,1), tf.argmax(y,1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initializing the variables
initilizer = tf.global_variables_initializer()

session_runner = SessionRunner(optimizer, accuracy, cost, lstm, initilizer, writer)
session_runner.run_session(x, y, n_input, dictionary, reverse_dictionary, training_data)
```

main.py hosted with ❤ by GitHub view raw

Once we run this code we get the accuracy of 97.10%:



```
Iteration = 22000, Average Accuracy= 96.30%
Iteration = 23000, Average Accuracy= 96.40%
Iteration = 24000, Average Accuracy= 94.70%
Iteration = 25000, Average Accuracy= 96.40%
Iteration = 26000, Average Accuracy= 95.30%
Iteration = 27000, Average Accuracy= 97.00%
Iteration = 28000, Average Accuracy= 95.50%
Iteration = 29000, Average Accuracy= 95.40%
Iteration = 30000, Average Accuracy= 96.10%
Iteration = 31000, Average Accuracy= 95.80%
Iteration = 32000, Average Accuracy= 96.90%
Iteration = 33000, Average Accuracy= 95.60%
Iteration = 34000, Average Accuracy= 95.50%
Iteration = 35000, Average Accuracy= 96.10%
Iteration = 36000, Average Accuracy= 97.10%
Iteration = 37000, Average Accuracy= 97.00%
Iteration = 38000, Average Accuracy= 96.60%
Iteration = 39000, Average Accuracy= 94.90%
Iteration = 40000, Average Accuracy= 95.50%
Iteration = 41000, Average Accuracy= 96.20%
Iteration = 42000, Average Accuracy= 96.10%
Iteration = 43000, Average Accuracy= 95.30%
Iteration = 44000, Average Accuracy= 95.10%
Iteration = 45000, Average Accuracy= 95.20%
Iteration = 46000, Average Accuracy= 96.10%
Iteration = 47000, Average Accuracy= 96.90%
Iteration = 48000, Average Accuracy= 95.90%
Iteration = 49000, Average Accuracy= 97.10%
Iteration = 50000, Average Accuracy= 97.10%
```

The problem for Keras Implementation

This example with TensorFlow was pretty straightforward, and simple. We used the small amount of data and network was able to learn this rather quickly. What if we have a more complex problem? For example, let's say that we want to classify sentiment of each movie review on some site. Lucky for us there is already a dataset, dedicated to this problem – **The Large Movie Review Dataset** (often referred to as the IMDB dataset).

This dataset was collected by Stanford researchers back in 2011. It contains 25000 movie reviews (good or bad) for training and the same amount of reviews for testing. Our goal is to create a network that will be able to determine which of these reviews are positive and which are negative. We will use Keras API which has this dataset built in.

Keras Implementation

The power of Keras is that it abstracts a lot of things we had to take care while we were using TensorFlow. However, it is giving us a less flexibility. Of course, everything is a trade-off. So, let's start this implementation by importing necessary classes and libraries.

```
from keras.preprocessing import sequence
```

```python
from keras.models import Sequential
from keras.layers import Dense, Dropout, Embedding, LSTM
from keras.datasets import imdb
```

import.py hosted with ❤ by GitHub view raw

As you can see there are is a little difference in imports from examples where we implemented **standard ANN** or when we implemented **Convolutional Neural Network**. We imported *Sequential, Dense* and *Dropout*. Still, we can see a couple new imports. We used *Embedding* as well as *LSTM* from the *keras.layers.* As you can imagine *LSTM* is used for creating LSTM layers in the networks. *Embedding*, on the other hand, is used to provide a dense representation of words.

This is one cool technique that will map each movie review into a real vector domain. Words are encoded as real-valued vectors in a high dimensional space, where the similarity between words in terms of meaning translates to closeness in the vector space. We can notice that we imported *imdb* dataset that is provided in the *keras.datasets*. However, to load data from that dataset we need to do this:

```python
num_words = 1000
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=num_words)
```

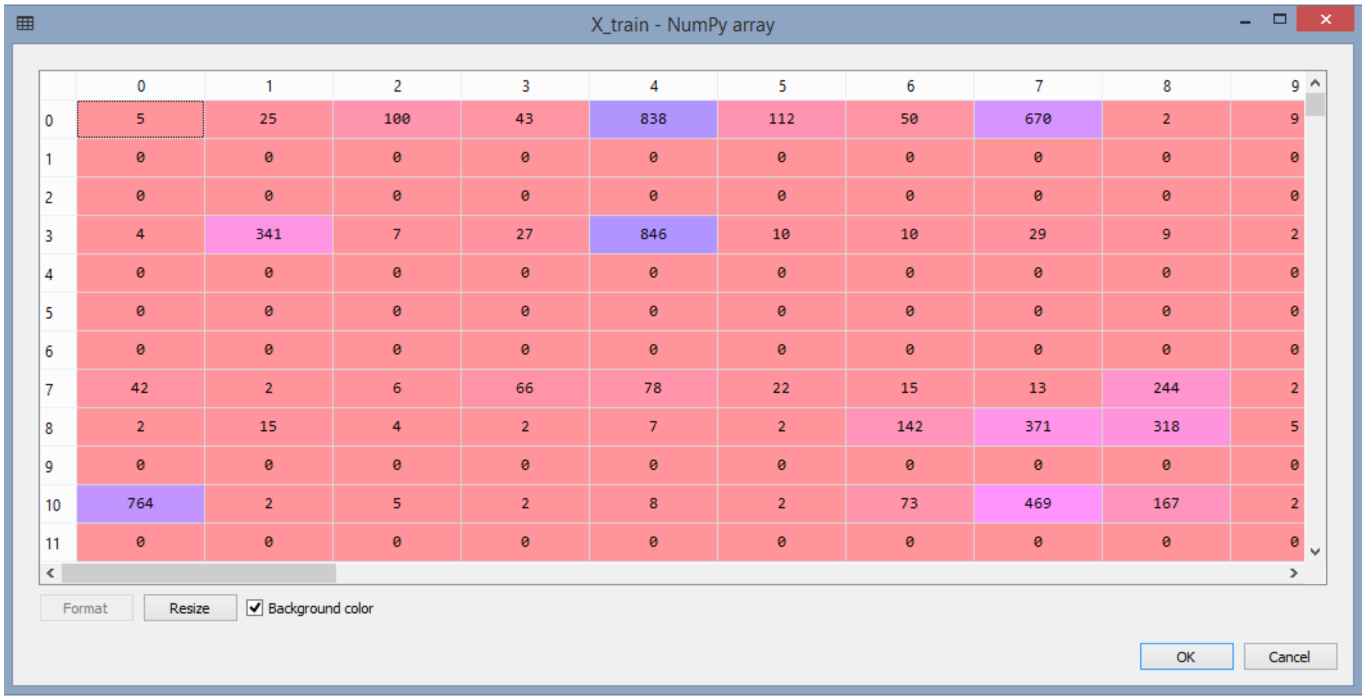
load.py hosted with ❤ by GitHub view raw

We are loading dataset of top 1000 words. After this, we need to divide this dataset and create and pad sequences. This is done by using *sequence* from *keras.preprocessing,* like this:

```python
X_train = sequence.pad_sequences(X_train, maxlen=200)
X_test = sequence.pad_sequences(X_test, maxlen=200)
```

pad_sequences.py hosted with ❤ by GitHub view raw

In the padding we used number 200, meaning that our sequences will be 200 words long. Here is how training input data is looking like after this:


X_train - NumPy array

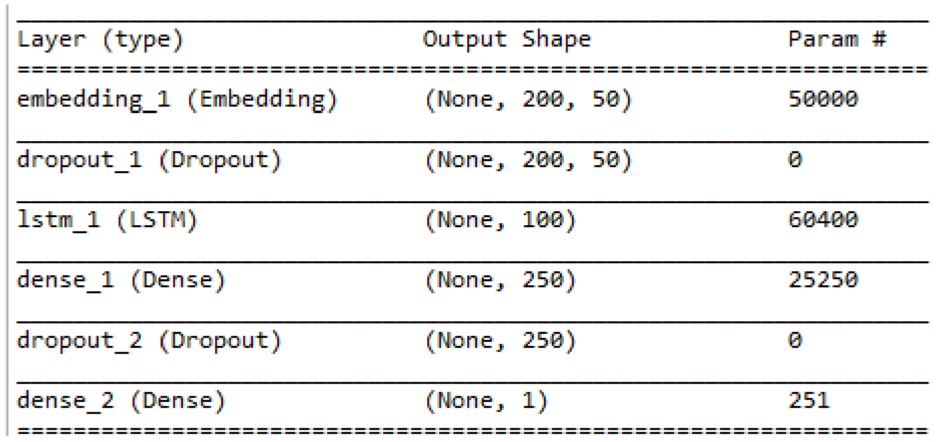
	0	1	2	3	4	5	6	7	8	9
0	5	25	100	43	838	112	50	670	2	9
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	4	341	7	27	846	10	10	29	9	2
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	42	2	6	66	78	22	15	13	244	2
8	2	15	4	2	7	2	142	371	318	5
9	0	0	0	0	0	0	0	0	0	0
10	764	2	5	2	8	2	73	469	167	2
11	0	0	0	0	0	0	0	0	0	0

Now we can define, compile and fit LSTM model:

```
# Define network architecture and compile
model = Sequential()
model.add(Embedding(num_words, 50, input_length=200))
model.add(Dropout(0.2))
model.add(LSTM(100, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(250, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

model.py hosted with ❤ by **GitHub** **view raw**

As we have learned from the **previous articles**, *Sequential* is used for model composition. The first layer that is added to it is *Embedding* and we've explained its purpose in the previous chapter. After the word embedding is done we added one LSTM layer. In the end, since this is a classification problem, we are trying to figure out was the review good or bad, *Dense* layer with sigmoid function is added. Finally, the model is compiled and *binary_crossentorpy* and *Adam* optimizer are used. This is how our model is looking like:



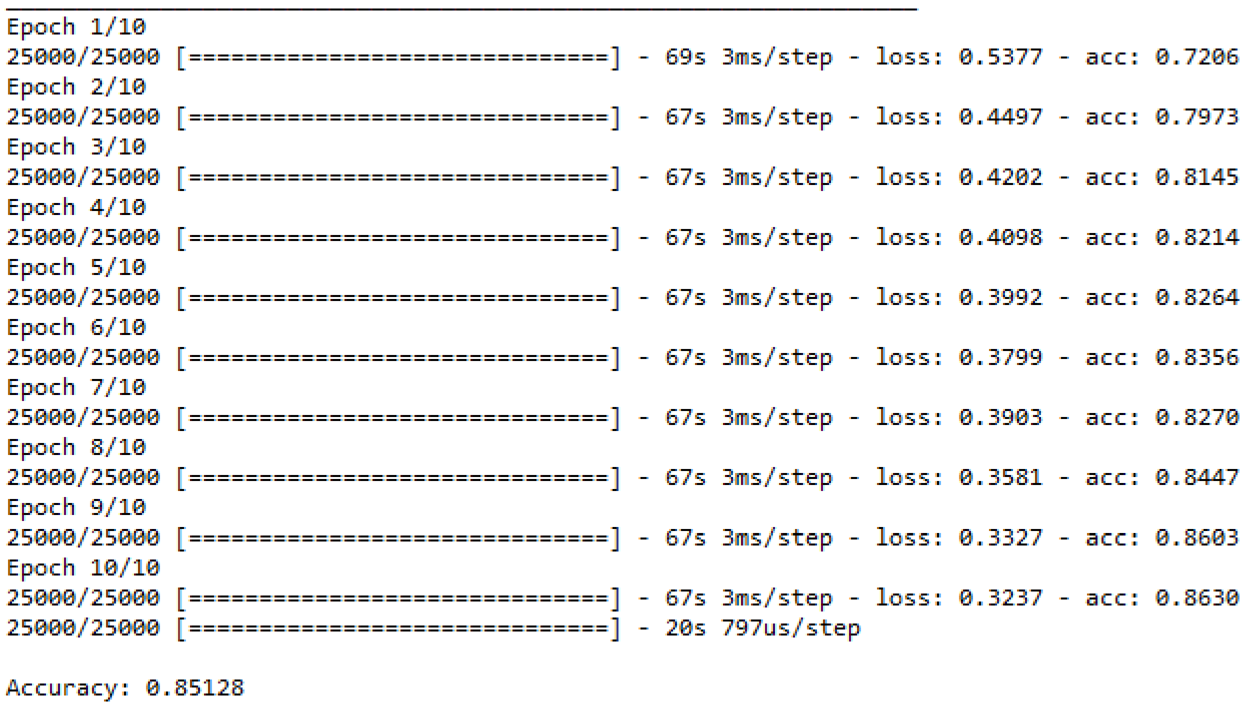
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 200, 50)           50000
_________________________________________________________________
dropout_1 (Dropout)          (None, 200, 50)           0
_________________________________________________________________
lstm_1 (LSTM)                (None, 100)               60400
_________________________________________________________________
dense_1 (Dense)              (None, 250)               25250
_________________________________________________________________
dropout_2 (Dropout)          (None, 250)               0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 251
=================================================================
```

Let's fit our data to the model and get the accuracy:

```
model.fit(X_train, y_train, batch_size=64, epochs=10)

print('\nAccuracy: {}'. format(model.evaluate(X_test, y_test)[1]))
```

fit.py hosted with ❤ by **GitHub** **view raw**

We got the accuracy of 85.12%.



```
_________________________________________________________________
Epoch 1/10
25000/25000 [==============================] - 69s 3ms/step - loss: 0.5377 - acc: 0.7206
Epoch 2/10
25000/25000 [==============================] - 67s 3ms/step - loss: 0.4497 - acc: 0.7973
Epoch 3/10
25000/25000 [==============================] - 67s 3ms/step - loss: 0.4202 - acc: 0.8145
Epoch 4/10
25000/25000 [==============================] - 67s 3ms/step - loss: 0.4098 - acc: 0.8214
Epoch 5/10
25000/25000 [==============================] - 67s 3ms/step - loss: 0.3992 - acc: 0.8264
Epoch 6/10
25000/25000 [==============================] - 67s 3ms/step - loss: 0.3799 - acc: 0.8356
Epoch 7/10
25000/25000 [==============================] - 67s 3ms/step - loss: 0.3903 - acc: 0.8270
Epoch 8/10
25000/25000 [==============================] - 67s 3ms/step - loss: 0.3581 - acc: 0.8447
Epoch 9/10
25000/25000 [==============================] - 67s 3ms/step - loss: 0.3327 - acc: 0.8603
Epoch 10/10
25000/25000 [==============================] - 67s 3ms/step - loss: 0.3237 - acc: 0.8630
25000/25000 [==============================] - 20s 797us/step

Accuracy: 0.85128
```

Conclusion

We saw two approaches when creating LSTM networks. Both approaches were dealing with simple problems and each was using a different API. This way one could see that TensorFlow is more detailed and flexible, however, you need to take care of lot more stuff than when you are using Keras. Keras is simpler and more straightforward but it doesn’t give us the flexibility and possibilities we have when we are using pure TensorFlow. Both of these examples provided ok results, but they could be even better. Especially the second example, for which we usually use a combination of CNN and RNN to get higher accuracy, but that is a topic for another article. Thanks for reading!

This article is a part of **Artificial Neural Networks Series**.

Read more posts from the author at **Rubik’s Code**.
