ML0120EN-3.1-Reveiw-LSTM-basics

January 18, 2020

RECURRENT NETWORKS IN DEEP LEARNING

Hello and welcome to this notebook. In this notebook, we will go over concepts of the Long Short-Term Memory (LSTM) model, a refinement of the original Recurrent Neural Network model. By the end of this notebook, you should be able to understand the Long Short-Term Memory model, the benefits and problems it solves, and its inner workings and calculations.

Table of Contents

Introduction

Recurrent Neural Networks are Deep Learning models with simple structures and a feedback mechanism built-in, or in different words, the output of a layer is added to the next input and fed back to the same layer.

The Recurrent Neural Network is a specialized type of Neural Network that solves the issue of **maintaining context for Sequential data** – such as Weather data, Stocks, Genes, etc. At each iterative step, the processing unit takes in an input and the current state of the network, and produces an output and a new state that is re-fed into the network.

Representation of a Recurrent Neural Network

However, this model has some problems. It's very computationally expensive to maintain the state for a large amount of units, even more so over a long amount of time. Additionally, Recurrent Networks are very sensitive to changes in their parameters. As such, they are prone to different problems with their Gradient Descent optimizer – they either grow exponentially (Exploding Gradient) or drop down to near zero and stabilize (Vanishing Gradient), both problems that greatly harm a model's learning capability.

To solve these problems, Hochreiter and Schmidhuber published a paper in 1997 describing a way to keep information over long periods of time and additionally solve the oversensitivity to parameter changes, i.e., make backpropagating through the Recurrent Networks more viable. This proposed method is called Long Short-Term Memory (LSTM).

(In this notebook, we will cover only LSTM and its implementation using TensorFlow)

Long Short-Term Memory Model

The Long Short-Term Memory, as it was called, was an abstraction of how computer memory works. It is "bundled" with whatever processing unit is implemented in the Recurrent Network, although outside of its flow, and is responsible for keeping, reading, and outputting information for the model. The way it works is simple: you have a linear unit, which is the information cell itself, surrounded by three logistic gates responsible for maintaining the data. One gate is for inputting data into the information cell, one is for outputting data from the input cell, and the last one is to keep or forget data depending on the needs of the network.

Thanks to that, it not only solves the problem of keeping states, because the network can choose to forget data whenever information is not needed, it also solves the gradient problems, since the Logistic Gates have a very nice derivative.

Long Short-Term Memory Architecture

The Long Short-Term Memory is composed of a linear unit surrounded by three logistic gates. The name for these gates vary from place to place, but the most usual names for them are:

the "Input" or "Write" Gate, which handles the writing of data into the information cell

the "Output" or "Read" Gate, which handles the sending of data back onto the Recurrent Network

the "Keep" or "Forget" Gate, which handles the maintaining and modification of the data stored in the information cell

Diagram of the Long Short-Term Memory Unit

The three gates are the centerpiece of the LSTM unit. The gates, when activated by the network, perform their respective functions. For example, the Input Gate will write whatever data it is passed into the information cell, the Output Gate will return whatever data is in the information cell, and the Keep Gate will maintain the data in the information cell. These gates are analog and multiplicative, and as such, can modify the data based on the signal they are sent.

For example, an usual flow of operations for the LSTM unit is as such: First off, the Keep Gate has to decide whether to keep or forget the data currently stored in memory. It receives both the input and the state of the Recurrent Network, and passes it through its Sigmoid activation. If K_t has value of 1 means that the LSTM unit should keep the data stored perfectly and if K_t a value of 0 means that it should forget it entirely. Consider S_{t-1} as the incoming (previous) state, x_t as the incoming input, and W_k, B_k as the weight and bias for the Keep Gate. Additionally, consider Old_{t-1} as the data previously in memory. What happens can be summarized by this equation:

$$K_t = \sigma(W_k \times [S_{t-1}, x_t] + B_k)$$

$$Old_t = K_t \times Old_{t-1}$$

As you can see, Old_{t-1} was multiplied by value was returned by the Keep Gate(K_t) – this value is written in the memory cell.

Then, the input and state are passed on to the Input Gate, in which there is another Sigmoid activation applied. Concurrently, the input is processed as normal by whatever processing unit is implemented in the network, and then multiplied by the Sigmoid activation's result I_t, much like

the Keep Gate. Consider W_i and B_i as the weight and bias for the Input Gate, and C_t the result of the processing of the inputs by the Recurrent Network.

$$I_t = \sigma(W_i \times [S_{t-1}, x_t] + B_i)$$

$$New_t = I_t \times C_t$$

New_t is the new data to be input into the memory cell. This is then added to whatever value is still stored in memory.

$$Cell_t = Old_t + New_t$$

We now have the candidate data which is to be kept in the memory cell. The conjunction of the Keep and Input gates work in an analog manner, making it so that it is possible to keep part of the old data and add only part of the new data. Consider however, what would happen if the Forget Gate was set to 0 and the Input Gate was set to 1:

$$Old_t = 0 \times Old_{t-1}$$

$$New_t = 1 \times C_t$$

$$Cell_t = C_t$$

The old data would be totally forgotten and the new data would overwrite it completely.

The Output Gate functions in a similar manner. To decide what we should output, we take the input data and state and pass it through a Sigmoid function as usual. The contents of our memory cell, however, are pushed onto a Tanh function to bind them between a value of -1 to 1. Consider W_o and B_o as the weight and bias for the Output Gate.

$$O_t = \sigma(W_o \times [S_{t-1}, x_t] + B_o)$$

$$Output_t = O_t \times tanh(Cell_t)$$

And that $Output_t$ is what is output into the Recurrent Network.

The Logistic Function plotted

As mentioned many times, all three gates are logistic. The reason for this is because it is very easy to backpropagate through them, and as such, it is possible for the model to learn exactly *how* it is supposed to use this structure. This is one of the reasons for which LSTM is a very strong structure. Additionally, this solves the gradient problems by being able to manipulate values through the gates themselves – by passing the inputs and outputs through the gates, we have now a easily derivable function modifying our inputs.

In regards to the problem of storing many states over a long period of time, LSTM handles this perfectly by only keeping whatever information is necessary and forgetting it whenever it is not needed anymore. Therefore, LSTMs are a very elegant solution to both problems.

LSTM

Lets first create a tiny LSTM network sample to understand the architecture of LSTM networks.

We need to import the necessary modules for our code. We need numpy and tensorflow, obviously. Additionally, we can import directly the tensorflow.contrib.rnn model, which includes the function for building RNNs.

[1]:
```
import numpy as np
import tensorflow as tf
sess = tf.Session()
```

We want to create a network that has only one LSTM cell. We have to pass 2 elements to LSTM, the prv_output and prv_state, so called, h and c. Therefore, we initialize a state vector, state. Here, state is a tuple with 2 elements, each one is of size [1 x 4], one for passing prv_output to next time step, and another for passing the prv_state to next time stamp.

[2]:
```
LSTM_CELL_SIZE = 4  # output size (dimension), which is same as hidden size in␣
 ↪the cell

lstm_cell = tf.contrib.rnn.BasicLSTMCell(LSTM_CELL_SIZE, state_is_tuple=True)
state = (tf.zeros([1,LSTM_CELL_SIZE]),)*2
state
```

[2]:
```
(<tf.Tensor 'zeros:0' shape=(1, 4) dtype=float32>,
 <tf.Tensor 'zeros:0' shape=(1, 4) dtype=float32>)
```

Let define a sample input. In this example, batch_size = 1, and seq_len = 6:

[3]:
```
sample_input = tf.constant([[3,2,2,2,2,2]],dtype=tf.float32)
print (sess.run(sample_input))
```

```
[[3. 2. 2. 2. 2. 2.]]
```

Now, we can pass the input to lstm_cell, and check the new state:

[4]:
```
with tf.variable_scope("LSTM_sample1"):
    output, state_new = lstm_cell(sample_input, state)
sess.run(tf.global_variables_initializer())
print (sess.run(state_new))
```

```
LSTMStateTuple(c=array([[ 0.4020096 , -0.48979944, -0.43417126, -0.7463889 ]],
      dtype=float32), h=array([[ 0.32737035, -0.12205502, -0.15654875,
-0.5282989 ]],
      dtype=float32))
```

As we can see, the states has 2 parts, the new state c, and also the output h. Lets check the output again:

[5]:
```
print (sess.run(output))
```

```
[[ 0.32737035 -0.12205502 -0.15654875 -0.5282989 ]]
```

Stacked LSTM

What about if we want to have a RNN with stacked LSTM? For example, a 2-layer LSTM. In this case, the output of the first layer will become the input of the second.

Lets start with a new session:

[6]:
```
sess = tf.Session()
```

[7]:
```
input_dim = 6
```

Lets create the stacked LSTM cell:

[8]:
```
cells = []
```

Creating the first layer LTSM cell.

[9]:
```
LSTM_CELL_SIZE_1 = 4 #4 hidden nodes
cell1 = tf.contrib.rnn.LSTMCell(LSTM_CELL_SIZE_1)
cells.append(cell1)
```

Creating the second layer LTSM cell.

[10]:
```
LSTM_CELL_SIZE_2 = 5 #5 hidden nodes
cell2 = tf.contrib.rnn.LSTMCell(LSTM_CELL_SIZE_2)
cells.append(cell2)
```

To create a multi-layer LTSM we use the tf.contrib.rnnMultiRNNCell function, it takes in multiple single layer LTSM cells to create a multilayer stacked LTSM model.

[11]:
```
stacked_lstm = tf.contrib.rnn.MultiRNNCell(cells)
```

Now we can create the RNN from stacked_lstm:

[12]:
```
# Batch size x time steps x features.
data = tf.placeholder(tf.float32, [None, None, input_dim])
output, state = tf.nn.dynamic_rnn(stacked_lstm, data, dtype=tf.float32)
```

Lets say the input sequence length is 3, and the dimensionality of the inputs is 6. The input should be a Tensor of shape: [batch_size, max_time, dimension], in our case it would be (2, 3, 6)

[13]:
```
#Batch size x time steps x features.
sample_input = [[[1,2,3,4,3,2],␣
↪[1,2,1,1,1,2],[1,2,2,2,2,2]],[[1,2,3,4,3,2],[3,2,2,1,1,2],[0,0,0,0,3,2]]]
```

```
sample_input
```

[13]:
```
[[[1, 2, 3, 4, 3, 2], [1, 2, 1, 1, 1, 2], [1, 2, 2, 2, 2, 2]],
 [[1, 2, 3, 4, 3, 2], [3, 2, 2, 1, 1, 2], [0, 0, 0, 0, 3, 2]]]
```

we can now send our input to network, and check the output:

[14]:
```
output
```

[14]:
```
<tf.Tensor 'rnn/transpose_1:0' shape=(?, ?, 5) dtype=float32>
```

[15]:
```
sess.run(tf.global_variables_initializer())
sess.run(output, feed_dict={data: sample_input})
```

[15]:
```
array([[[-0.00814136,  0.05396684,  0.0069079 ,  0.01966399,
          0.02080007],
        [-0.01359569,  0.08686274,  0.01117458,  0.03448345,
          0.01553203],
        [-0.02453815,  0.13257378,  0.01510709,  0.05805094,
          0.02027102]],

       [[-0.00814136,  0.05396684,  0.0069079 ,  0.01966399,
          0.02080007],
        [-0.00281993,  0.0948989 , -0.00875902,  0.04722311,
          0.03249784],
        [-0.01348863,  0.11981951, -0.01278677,  0.09910364,
          0.04303816]]], dtype=float32)
```

As you see, the output is of shape (2, 3, 5), which corresponds to our 2 batches, 3 elements in our sequence, and the dimensionality of the output which is 5.

0.1 Want to learn more?

Running deep learning programs usually needs a high performance platform. **PowerAI** speeds up deep learning and AI. Built on IBM's Power Systems, **PowerAI** is a scalable software platform that accelerates deep learning and AI with blazing performance for individual users or enterprises. The **PowerAI** platform supports popular machine learning libraries and dependencies including TensorFlow, Caffe, Torch, and Theano. You can use PowerAI on IMB Cloud.

Also, you can use **Watson Studio** to run these notebooks faster with bigger datasets.___Watson Studio___ is IBM's leading cloud solution for data scientists, built by data scientists. With Jupyter notebooks, RStudio, Apache Spark and popular libraries pre-packaged in the cloud, **Watson Studio** enables data scientists to collaborate on their projects without having to install anything. Join the fast-growing community of **Watson Studio** users today with a free account at Watson Studio.This is the end of this lesson. Thank you for reading this notebook, and good luck on your studies.

0.1.1 Thanks for completing this lesson!

Notebook created by: Saeed Aghabozorgi , Walter Gomes de Amorim Junior