A noob's guide to implementing RNN-LSTM using Tensorflow

machine learning *June 20, 2016*

The purpose of this tutorial is to help anybody write their first RNN LSTM model without much background in Artificial Neural Networks or Machine Learning. The discussion is not centered around the theory or working of such networks but on writing code for solving a particular problem. We will understand how neural networks let us solve some problems effortlessly, and how they can be applied to a multitude of other problems.

What are RNNs?

Simple multi-layered neural networks are classifiers which when given a certain input, tag the input as belonging to one of the many classes. They are trained using the existing backpropagation algorithms. These networks are great at what they do but they are not capable of handling inputs which come in a sequence. For example, for a neural net to identify the nouns in a sentence, having just the word as input is not helpful at all. A lot of information is present in the context of the word which can only be determined by looking at the words near the given word. The entire sequence is to be studied to determine the output. This is where Recurrent Neural Networks (RNNs) find their use. As the RNN traverses the input sequence, output for every input also becomes a part of the input for the next item of the sequence. You can read more about the utility of RNNs in Andrej Karpathy's brilliant blog post. It is helpful to note the 'recurrent' property of the network, where the previous output for an input item becomes a part of the current input which comprises the current item in the sequence and the last output. When done over and over, the last output would be the result of all the previous inputs and the last input.

What is LSTM?

RNNs are very apt for sequence classification problems and the reason they're so good at this is that they're able to retain important data from the previous inputs and use that information to modify the current output. If the sequences are quite long, the gradients (values calculated to tune the network) computed during their training (backpropagation) either vanish (multiplication of many 0 < values < 1) or explode (multiplication of many large values) causing it to train very slowly.

Long Short Term Memory is a RNN architecture which addresses the problem of training over long sequences and retaining memory. LSTMs solve the gradient problem by introducing a few more gates that control access to the cell state. You could refer to Colah's blog post which is a great place to understand the working of LSTMs. If you didn't get what is being discussed, that's fine and you can safely move to the next part.

The task

Given a binary string (a string with just 0s and 1s) of length 20, we need to determine the count of 1s in a binary string. For example, "01010010011011100110" has 11 ones. So the input for our program will be a string of length twenty that contains 0s and 1s and the output must be a single number between 0 and 20 which represents the number of ones in the string. Here is a link to the complete gist, in case you just want to jump at the code.

Even an amateur programmer can't help but giggle at the task definition. It won't take anybody more than a minute to execute this program and get the correct output on every input (0% error).

```python
count = 0
for i in input_string:
    if i == '1':
        count+=1
```

Anybody in their right mind would wonder, if it is so easy, why the hell can't a computer figure it out by itself? Computers aren't that smart without a human instructor. Computers need to be given precise instructions and the 'thinking' has to be done by the human issuing the commands. Machines can repeat the most complicated calculations a gazillion times over but they still fail miserably at things humans do painlessly, like recognizing cats in a picture.

What we plan to do is to feed neural network enough input data and tell it the correct output values for those inputs. Post that, we will give it input that it has not seen before and we will see how many of those does the program get right.

Generating the training input data

Each input is a binary string of length twenty. The way we will represent it will be as a python list of 0s and 1s. The test input to be used for training will contain many such lists.

```python
import numpy as np
from random import shuffle

train_input = ['{0:020b}'.format(i) for i in range(2**20)]
shuffle(train_input)
train_input = [map(int,i) for i in train_input]
ti  = []
for i in train_input:
    temp_list = []
    for j in i:
            temp_list.append([j])
    ti.append(np.array(temp_list))
train_input = ti
```

There can be a total of 2^{20} ~ 10^{6} combinations of 1s and 0s in a string of length 20. We generate a list of all the 2^{20} numbers, convert it to their binary string and shuffle the entire list. Each binary string is then converted to a list of 0s and 1s. Tensorflow requires input as a tensor (a Tensorflow variable) of the dimensions [batch_size, sequence_length, input_dimension] (a 3d variable). In our case, batch_size is something we'll determine later but sequence_length is fixed at 20 and input_dimension is 1 (i.e each individual bit of the string). Each bit will actually be represented as a list containing just that bit. A list of 20 such lists will form a sequence which we convert to a numpy array. A list of all such sequences is the value of train_input that we're trying to compute. If you print the first few values of train_input, it would look like

```
[
 array([[0],[0],[1],[0],[0],[1],[0],[1],[1],[0],[0],[0],[1],[1],[1],[1],[1],
 array([[1],[1],[0],[0],[0],[0],[1],[1],[1],[1],[1],[0],[0],[1],[0],[0],[0],
 .....
]
```

Don't worry about the values if they don't match yours because they will be different as they are in random order.

Generating the training output data

For every sequence, the result can be anything between 0 and 20. So we have 21 choices per sequence. Very clearly, our task is a sequence classification problem. Each sequence belongs to the class number which is the same as the count of ones in the sequence. The representation of the output would be a list of the length of 21 with zeros at all positions except a one at the index of the class to which the sequence belongs.

```
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20
This is a sample output for a sequence which belongs to 4th class i.e has 4
```

More formally, this is called the one hot encoded representation.

```
train_output = []

for i in train_input:
    count = 0
    for j in i:
        if j[0] == 1:
            count+=1
    temp_list = ([0]*21)
    temp_list[count]=1
    train_output.append(temp_list)
```

For every training input sequence, we generate an equivalent one hot encoded output representation.

Generating the test data

For any supervised machine learning task, we need some data as training data to teach our program to identify the correct outputs and some data as test data to check how our program performs on inputs that it hasn't seen before. Letting test and training data overlap is self-defeating because, if you had already practiced the questions that were to come in your exam, you would most definitely ace it. Currently in our train_input and train_output, we have 2^{20} (1,048,576) unique examples. We will split those into two sets, one for training and the other for testing. We will take 10,000 examples (0.9% of the entire data) from the dataset and use it as training data and use the rest of the 1,038,576 examples as test data.

```
NUM_EXAMPLES = 10000
test_input = train_input[NUM_EXAMPLES:]
test_output = train_output[NUM_EXAMPLES:] #everything beyond 10

train_input = train_input[:NUM_EXAMPLES]
train_output = train_output[:NUM_EXAMPLES] #till 10,000
```

Designing the model

This is the most important part of the tutorial. Tensorflow and various other libraries (Theano, Torch, PyBrain) provide tools for users to design the model without getting into the nitty-gritty of implementing the neural network, the optimization or the backpropagation algorithm.

Danijar outlines a great way to organize Tensorflow models which you might want to use later to organize tidy up your code. For the purpose of this tutorial, we will skip that and focus on writing code that just works.

Import the required packages to begin with. If you haven't already installed Tensorflow, follow the instructions on this page and then continue.

```
import tensorflow as tf
```

After importing the tensorflow, we will define two variables which will hold the input data and the target data.

```
data = tf.placeholder(tf.float32, [None, 20,1])
target = tf.placeholder(tf.float32, [None, 21])
```

The dimensions for data are [Batch Size, Sequence Length, Input Dimension]. We let the batch size be unknown and to be determined at runtime. Target will hold the training output data which are the correct results that we desire. We've made Tensorflow placeholders which are basically just what they are, placeholders that will be supplied with data later.

Now we will create the RNN cell. Tensorflow provides support for LSTM, GRU (slightly different architecture than LSTM) and simple RNN cells. We're going to use LSTM for this task.

```
num_hidden = 24
cell = tf.nn.rnn_cell.LSTMCell(num_hidden,state_is_tuple=True)
```

For each LSTM cell that we initialise, we need to supply a value for the hidden dimension, or as some people like to call it, the number of units in the LSTM cell. The value of it is it up to you, too high a value may lead to overfitting or a very low value may yield extremely poor results. As many experts have put it, selecting the right parameters is more of an art than science.

Before we write any more code, it is imperative to understand how Tensorflow computation graphs work. From a hacker perspective, it is enough to think of it as having two phases.

The **first phase is building the computation graph where you define all the calculations and functions** that you will execute during runtime. The **second phase is the execution phase where a Tensorflow session is created and the graph that was defined earlier is executed** with the data we supply.

```
val, state = tf.nn.dynamic_rnn(cell, data, dtype=tf.float32)
```

We unroll the network and pass the data to it and store the output in val. We also get the state at the end of the dynamic run as a return value but we discard it because every time we look at a new sequence, the state becomes irrelevant for us. **Please note, writing this line of code doesn't mean it is executed. We're still in the first phase of designing the model. Think of these as functions that are stored in variables which will be invoked when we start a session.**

```
val = tf.transpose(val, [1, 0, 2])
last = tf.gather(val, int(val.get_shape()[0]) - 1)
```

We transpose the output to switch batch size with sequence size. After that we take the values of outputs only at sequence's last input, which means in a string of 20 we're only interested in the output we got at the 20th character and the rest of the output for previous characters is irrelevant here.

```
weight = tf.Variable(tf.truncated_normal([num_hidden, int(targe
bias = tf.Variable(tf.constant(0.1, shape=[target.get_shape()[1
```

What we want to do is apply the final transformation to the outputs of the LSTM and map it to the 21 output classes. We define weights and biases, and multiply the output with the weights and add the bias values to it. The dimension of the weights will be num_hidden X number_of_classes. Thus on multiplication with the output (val), the resulting dimension will be batch_size X number_of_classes which is what we are looking for.

```
prediction = tf.nn.softmax(tf.matmul(last, weight) + bias)
```

After multiplying the output with the weights and adding the bias, we will have a matrix with a variety of different values for each class. What we are interested in is the probability score for each class i.e the chance that the sequence belongs to a particular class. We then calculate the softmax activation to give us the probability scores.

What is this function and why are we using it?

$$\frac{e^{x_j}}{\sum_K^{k=1} e^{x_k}}$$

This function takes in a vector of values and returns a probability distribution for each index depending upon its value. This function returns a probability scores (sum of all the values equate to one) which is the final output that we need. If you want to learn more about softmax, head over to this link.

```
cross_entropy = -tf.reduce_sum(target * tf.log(tf.clip_by_value
```

The next step is to calculate the loss or in less technical words, our degree of incorrectness. We calculate the cross entropy loss (more details here) and use that as our cost function. The cost function will help us determine how poorly or how well our predictions stack against the actual results. This is the function that we are trying to minimize. If you don't want to delve into the technical details, it is okay to just understand what cross entropy loss is calculating. The log term helps us measure the degree to which the network got it right or wrong. Say for example, if the target was 1 and the prediction is close to one, our loss would not be much because the values of -log(x) where x nears 1 is almost 0. For the same target, if the prediction was 0, the cost would increase by a huge amount because -log(x) is very high when x is close to zero. Adding the log term helps in penalizing the model more if it is terribly wrong and very little when the prediction is close to the target. The last step in model design is to prepare the optimization function.

```
optimizer = tf.train.AdamOptimizer()
minimize = optimizer.minimize(cross_entropy)
```

Tensorflow has a few optimization functions like RMSPropOptimizer, AdaGradOptimizer, etc. We choose AdamOptimzer and we set minimize to the function that shall minimize the cross_entropy loss that we calculated previously.

Calculating the error on test data

```
mistakes = tf.not_equal(tf.argmax(target, 1), tf.argmax(predict
error = tf.reduce_mean(tf.cast(mistakes, tf.float32))
```

This error is a count of how many sequences in the test dataset were classified incorrectly. This gives us an idea of the correctness of the model on the test dataset.

Execution of the graph

We're done with designing the model. Now the model is to be executed!

```
init_op = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init_op)
```

We start a session and initialize all the variables that we've defined. After that, we begin our training process.

```
batch_size = 1000
no_of_batches = int(len(train_input)/batch_size)
epoch = 5000
for i in range(epoch):
    ptr = 0
    for j in range(no_of_batches):
        inp, out = train_input[ptr:ptr+batch_size], train_outp
        ptr+=batch_size
        sess.run(minimize,{data: inp, target: out})
    print "Epoch - ",str(i)
incorrect = sess.run(error,{data: test_input, target: test_out
print('Epoch {:2d} error {:3.1f}%'.format(i + 1, 100 * incorre
sess.close()
```

We decide the batch size and divide the training data accordingly. I've fixed the batch size at 1000 but you would want to experiment by changing it to see how it impacts your results and training time.

If you are familiar with stochastic gradient descent, this idea would seem fairly simple. Instead of updating the values after running it through all the training samples, we break the training set into smaller batches and run it for those. After processing each batch, the values of the network are tuned. So every few steps, the network weights are adjusted. Stochastic optimization methods are known to perform better than their counterparts for certain functions. This is because the stochastic methods converge much faster but this may not always be the case.

For every batch, we get the input and output data and we run minimize, the optimizer function to minimize the cost. All the calculation of prediction, cost and backpropagation is done by tensorflow. We pass the feed_dict in sess.run along with the function. The feed_dict is a way of assigning data to tensorflow variables in that frame. So we pass the input data along with target (correct) outputs. The functions that we wrote above, are now being executed.

That's all. We've made our toy LSTM-RNN that learns to count just by looking at correct examples! This wasn't very intuitive to me when I trained it for the first time, so I added this line of code below the error calculation that would print the result for a particular example.

```
print sess.run(model.prediction,{data: [[[1],[0],[0],[1],[1],[0
```

So as the model trains, you will notice how the probability score at the correct index in the list gradually increases. Here's a link to the complete gist of the code.

Concerns regarding the training data

Many would ask, why use a training data set which is just 1% of the all the data. Well, to be able to train it on a CPU with a single core, a higher number would increase the time exponentially. You could of course adjust the batch size to still keep the number of updates same but the final decision is always up to the model designer. Despite everything, you will be surprised with the results when you realize that 1% of the data was enough to let the network achieve stellar results!

Tinkering with the model

You can try changing the parameter values to see how it affects the performance and training time. You can also try adding multiple layers to the RNN to make your model more complex and enable it to learn more features. An important feature you can implement is to add the ability to save the model values after every few iterations and retrieve those values to perform predictions in future. You could also change the cell from LSTM to GRU or a simple RNN cell and compare the performance.

Results

Training the model with 10,000 sequences, batch size of 1,000 and 5000 epochs on a MacbookPro/8GB/2.4Ghz/i5 and no GPU took me about 3-4 hours. And now the answer to the question, everybody is waiting for. How well did it perform?

```
Epoch 5000 error 0.1%
```

For the final epoch, the error rate is 0.1% across the entire (almost so because our test data is 99% of all possible combinations) dataset! This is pretty close to what somebody with the least programming skills would have been able to achieve (0% error). But, our neural network figured that out by itself! We did not instruct it to perform any of the counting operations.

If you want to speed up the process, you could try reducing the length of the binary string and adjusting the values elsewhere in the code to make it work.

What can you do now?

Now that you've implemented your LSTM model, what else is there that you can do? Sequence classification can be applied to a lot of different problems, like handwritten digit recognition or even autonomous car driving! Think of the rows of the image as individual steps or inputs and the entire image to be the sequence. You must classify the image as belonging to one of the classes which could be to halt, accelerate, turn left, turn right or continue at same speed. Training data could be a stopper but hell, you could even generate it yourself. There is so much more waiting to be done!

*The post has been updated to be compatible with Tensorflow version 0.9 and above.

Share this:

Related

Solving the Auto-Rickshaw Problem in Mumbai
August 1, 2010
In "thoughts"

Jnana : Hyperfocus Education
December 24, 2012
In "projects"

A quick guide to getting a Django - Uwsgi - Nginx server up on Ubuntu 16.04 (AWS EC2)
July 26, 2017
In "development"

58 Comments

58 Comments

Jay
June 20, 2016

Reply

Could you explain why you only use the last output of the sequence?

Monik

Reply

June 20, 2016

We use the last output of the sequence for calculating the cost because the last output accounts for all the outputs due to previous items in the sequence.

For example, given a sequence "10100100110010101101", the RNN would treat it like this :

Step 1:
Input : 1,None (Previous output goes here)
Output : x1

Step 2:
Input: 0,x1
Output : x2

Step 3:
Input 1,x2
Output : x3

… at the last step ..

Step 20:
Input 1,x19
Output : x20

So the final output has accounted for all the input items in the sequence and thus it is enough to only use the last output of the sequence when computing the cost. We want to classify the entire sequence and we only have correct values for entire sequence (class to which it belongs) to be able to compare it with.

Although, there would be a case where we would be interested in using all the outputs of the sequence. Such a problem would be a sequence labelling problem, such as that of tagging nouns in a sentence where you would have an output for each and every word.

MSummers
June 16, 2017

Reply

Looks like she took the sample from Aymeric Damien like most of the other guys!?!! Of course you need to do softmax at several output nodes to predict sequences. @Monik, please go through Andrej's RNN example and try to update your code.

Danijar Hafner
June 25, 2016

Reply

Very comprehensive and well-written tutorial! Btw, in `no_of_batches = int(len(train_input)) / batch_size` you probably meant to cast the result rather than the length which is already int.

Monik
June 25, 2016

Reply

Yes, that's correct. Thanks for the correction!

Karanl
June 28, 2016

Reply

One the est tutorial on Tensorflow, Rnn for practical application!! amazing

Pavel
August 9, 2016

Reply

Looks like you are ignoring state that is returned from dynamic_rnn.. so how is this network learning in such case?

Monik
August 12, 2016

Reply

dynamic_rnn returns a state after the entire input sequence is traversed and not the state after each input bit (which constitutes the sequence) is iterated. Thus, it makes sense to discard it because we do not want to pass inter-sequence state over different batches. Tensorflow takes care of the back propagation and tuning of the state variables in this case.

ankuj
August 16, 2016

Reply

Very nicely compiled and explained article. Helped me better understand the LSTM implementation on tensorflow.org 🙂

Shalom
September 5, 2016

Reply

Tensorflow has been updated – `from tensorflow.models.rnn import rnn_cell` now throws an ImportError.

Instead, `cell = rnn_cell.LSTMCell(num_hidden)` should be `cell = tf.nn.rnn_cell.LSTMCell(num_hidden)

Monik
September 6, 2016

Reply

Thanks Shalom. The post and the code has been updated to be compatible with the newer version of tensorflow.

Benecoder
March 3, 2017

Reply

For the TF 1.0 Release tnn_cell got temporarly moved to contrib. So it tf.contrib.rnn.BasicLSTMCell(is the right object now, but that will no last for long. I don't know wether you bother to change that.

Mentioned here: https://github.com/tensorflow/tensorflow/issues/7664

Vishal
September 6, 2016

Reply

I changed your cell definition as follows:

num_layers=2
cell = tf.nn.rnn_cell.LSTMCell(num_hidden,state_is_tuple=True)
cell = tf.nn.rnn_cell.MultiRNNCell([cell] * num_layers, state_is_tuple=True)

and the performance for 200 epochs went from 4% to 0.7%

Monik
September 6, 2016

Reply

If you introduce another layer, you are also increasing the number of parameters to be trained to get the certain results. Thus the performance may go down as more layers will require more training to do the job.

Raisa
September 7, 2016

Reply

Very well explained especially helpful for beginners. I recently started learning about RNNs. I have one question. Is it possible to use more that string as the input? And giving a class label to each string. (something like a sequence of sequences)

Monik
September 11, 2016

Reply

If I get what you're saying, https://www.tensorflow.org/versions/r0.10/tutorials/seq2seq/index.html which is Google's seq2seq model might be the answer

Sukrit Gupta
September 11, 2016

Reply

The tutorial was really helpful. But the doubt that I am having is that, I have an energy load prediction dataset that predicts load on hourly bases. The dataset has the hourly load requirements of a region for 24 hours of a day. The dataset that I am referring is available here: https://www.kaggle.com/c/global-energy-forecasting-competition-2012-load-forecasting/data
Now in this case will using only the last output of the sequence work? If not, Then what should be the desired change in the code snippet?

Mad Wombat
September 13, 2016

Reply

Nice explanation. I wonder, why did you pick tf.nn.dynamic_rnn? How is it different from tf.nn.rnn or other variants provided by tf?

Chirag Jain
October 7, 2016

Reply

excerpt from http://www.wildml.com/2016/08/rnns-in-tensorflow-a-practical-guide-and-undocumented-features/

"Internally, tf.nn.rnn creates an unrolled graph for a fixed RNN length. That means, if you call tf.nn.rnn with inputs having 200 time steps you are creating a static graph with 200 RNN steps. First, graph creation is slow. Second, you're unable to pass in longer sequences (> 200) than you've originally specified.

tf.nn.dynamic_rnn solves this. It uses a tf.While loop to dynamically construct the graph when it is executed. That means graph creation is faster and you can feed batches of variable size"

Baki
September 17, 2016

Reply

Thanks! Very nice explanation. I'm going to try this implementation to predict ranking of movie reviews.

Kim
September 22, 2016

Reply

Thank you for your great explanation! I've been looking for an easy, explanatory, working example. And this is definitely the one. Thank you again.

Eudie
October 24, 2016

Reply

Thanks Monik, very helpful for beginners like me. It would be very helpful if you can clarify my couple of doubts :
1. What are the dimensions of val(output), and what they are representing?
2. Are we using state(output), between the sequence(of length 20)?

Monik
November 2, 2016

Reply

Hello Eudie,

1. The dimensions of the output are 1 X class_size+1 which is the maximum no of set bits in our case. They represent the output value till that point in the sequence. We would be interested in these outputs if it were a sequence tagging problem where each item in the sequence is to be trained for a particular output. But we are only interested in the final output of the entire sequence so we discard all the preliminary outputs.

2. Yes, the state is being used internally by the rnn cell as it unloops itself in time. The state the function returns is at the end of traversing the entire sequence.

Som

Reply

November 4, 2016

Could you expand on why the following line is necessary:
val = tf.transpose(val, [1, 0, 2])

Thanks

Monik
November 13, 2016
Reply

From Tensorflow's docstring here
(https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/ops/rnn.py),

The outputs of dynamic_rnn called before the transpose are,
If time_major == False (default), this will be a Tensor shaped: [batch_size, max_time, cell.output_size].
If time_major == True, this will be a Tensor shaped: [max_time, batch_size, cell.output_size].

We didn't pass a value for time_major while calling the function so it defaults to false in which case the returned tensor dimensions are [batch_size, max_time, cell.output_size].

Tf.transpose transposes the tensor to change to [max_time, batch_size, cell.output_size] ([0 1 2] —> [1 0 2]). For our experiment, we need the last time step of each batch. The next function call, tf.gather simply gets us the last element in the first dimension, which will be the last time step for all the batches. Say, if A is transposed 3 dim array, [L X M X N], with gather we get A[L]

Trideep Rath
January 30, 2017
Reply

I am unable to understand this. I guess the batch_size is None(n), max_time = 20, output_size = 21. So this makes value of dimension n * 20 * 21 . What i understand is the requirement is that, the last output value of every sequence. Can you please elaborate.
Thanks

John Skibicki III
June 15, 2017
Reply

That was insanely confusing for me as well. I had to re-read that a few time. From what I understand is that the order of the elements in the array is in the wrong arrangement. The transpose function changes their positions. So if val's elements are ordered val = [A,B,C] running it through transpose would change val to [B,A,C].

Any way you slice it I don't know that I would have understood to do that in any of the tensorflow documentation. Regardless, Great Article!

Malhar Thakkar
November 8, 2016

Reply

First, I get a warning saying, "UserWarning: Converting sparse IndexedSlices to a dense Tensor of unknown shape. This may consume a large amount of memory.
"Converting sparse IndexedSlices to a dense Tensor of unknown shape. """

Additionally, I am getting an error at the following line in the code:
sess.run(minimize,{data: inp, target: out})

It says, "ValueError: setting an array element with a sequence."

Monik
November 13, 2016

Reply

The warning is because of the 'None' in
data = tf.placeholder(tf.float32, [None, 20,1])

We don't indicate the size to allocate and thus make it variable which may slow it down. This is done so that we can vary the batch size during training as we wish. If you are certain about the size, you may replace the None with the size and the warning should disappear.

Regarding the other error, the dimensions of the inp variable may be incorrect. It is likely that not all elements in inp have the same length which must be the case.

XiangHu
November 18, 2016

Reply

is there something wrong about the code in
https://gist.github.com/monikkinom/e97d518fe02a79177b081c028a83ec1c?
i just cope it and run it,after a while,it come out that:
Epoch 4997
Epoch 4998
Epoch 4999

[[nan nan nan nan nan nan nan nan nan nan nan nan nan nan
nan nan nan nan nan nan nan]]
Epoch 5000 error 100.0%
please let me know what' wrong about it! thanks a lot!

XiangHu
November 18, 2016

Reply

??

Monik
December 2, 2016

Reply

The error is because of this line :

cross_entropy = -tf.reduce_sum(target * tf.log(prediction))

because for certain classes the probability scores may be very small (<10^-8) so the product function seems to mess up.

Solution to it is replacing the above line with

cross_entropy = -tf.reduce_sum(target * tf.log(tf.clip_by_value(prediction,1e-10,1.0)))

A more detailed explanation in this stackoverflow answer: http://stackoverflow.com/a/33713196/2950515

David
November 24, 2016

Reply

Great overview, but I'm still not clear on a few things related to actually coding LSTMs with TensorFlow. Where it says "the number of units in the LSTM cell", it isn't clear what a "unit" is. What is the distinction between a "unit" and a "cell"? Are "units" the rolled-across-time instances of an LSTM that hold the sequence steps for a given input data item? And a cell is the combination of all those units for 1 data item? If so, why is num_hidden (i.e. number of units) so big (24), given that the sequence length is just 20? It would seem that 20 units should be ample to preserve sufficient state information during presentation of the sequence – in fact, it would seem 5 units would be sufficient to count to 20.

A diagram following each line of TensorFlow code would be also be extremely helpful because it would help clarify what the graph looks like after tf.nn.rnn_cell.LSTMCell, and after tf.nn.dynamic_rnn. (It is almost impossible to find diagrams on the web of LSTM architectures that directly correspond to TensorFlow code. There are lots of abstract LSTM diagrams, and lots of standalone TensorFlow code for LSTMs, but for some reason they never seem to be both shown for the same problem.)

Thanks!

Monik
December 2, 2016

Reply

Comparing it to the usual multi layered network, you can say that a cell represents a layer and each unit represents an individual perceptron. It has nothing to do with rolling across time. The number of units decide the dimensions of the tensors that are used in calculation because it affects the size of the state tensor (more the units, higher the dimension) and the weight tensor. Each individual 'unit' in a cell performs the same function and each unit is gated similarly.

It is difficult to say, what size would be perfect because it is hard to guess where the minima may lie (for different state sizes). If you reduce the state size, the number of epochs required will definitely jump up which is why i've kept it sufficiently high to be training it on a normal CPU. It isn't that obvious to pick a size of 5 because although 5 bit positions are enough to reflect the count, it may not be enough the do the 'thinking' or store the intermediate computation that lead to the answer. To put it in a human way, although we see 5 bits to determine what number in decimal it may represent, but internally we will be multiplying bits with powers of two and summing them up so all the intermediate values being generated during the conversion must also be accounted for somewhere.

Thank you for the suggestion and I'll definitely try adding the diagrams when I get time. In the mean time, you could try generating it for yourself with some effort of naming the variables and mentioning scope. Tensorboard, tensorflow's visualisation library will come handy :
https://www.tensorflow.org/versions/r0.12/how_tos/graph_viz/index.html

Niklas
January 31, 2017

Reply

Hey!

I have to agree with the common opinon and thank you for a great article!

Concerning the number of hidden units, as i understand it it can be interpolated as the memory of LSTM cells. But i still have a hard time to get my head around why we have a larger memory than sequence. If the memory is larger than the input sequence it should always be possible to store the how sequence in the memory. Is this correct?

Daniel
December 1, 2016

Reply

Hello.
Great tutorial.
I've tried it with my own data and it worked great.
The problem is when I try to include a validation set.
I'm trying to run the trained cell on my validation set and I get the following error:
“Variable RNN/BasicLSTMCell/Linear/Matrix already exists, disallowed. Did you mean to set reuse=True in VarScope? Originally defined at:”
Here is my code:

```
# Variables.
embeddings = tf.Variable(
tf.random_uniform([aa_num, embedding_size], -1.0, 1.0))
cell = tf.nn.rnn_cell.BasicLSTMCell(num_hidden,state_is_tuple=True)
weight = tf.get_variable(initializer=tf.truncated_normal([num_hidden, num_labels], stddev=0.1),
name = “W”)
bias = tf.get_variable(initializer=tf.constant(0.1, shape=[num_labels]), name = “B”)

#Model calculation.
def model(data):
embed = tf.nn.embedding_lookup(embeddings, data)
val, state = tf.nn.dynamic_rnn(cell, embed, dtype=tf.float32)
val = tf.transpose(val, [1, 0, 2])
last = tf.gather(val, int(val.get_shape()[0]) – 1)
return tf.matmul(last, weight) + bias

# Training computation.
logits = model(tf_train_dataset)
loss = tf.reduce_mean(
tf.nn.softmax_cross_entropy_with_logits(logits, tf_train_labels)) + beta

# Optimizer.
learning_rate = tf.train.exponential_decay(0.001, global_step, 1000, 1.0)
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

# Predictions for the training, validation, and test data.
train_prediction = tf.nn.softmax(logits)
valid_prediction = tf.nn.softmax(model(tf_valid_dataset))
```

Thanks a lot for any help.

Monik
December 2, 2016

Reply

It is a scope related problem. You are redeclaring the cells where you must not be doing it. The portion of the code dealing the the scope and the session may give a better idea as to what is going wrong.

sunny
December 7, 2016
Reply

seems unable to open the " complete gist of the code".

Grant
December 7, 2016
Reply

Thank you so much for this wonderfully illustrated example! Two thumbs up!!

Could you please help me understand what would need to change in order to adapt this from Classification to Time Series Regression? I am attempting to artificially generate training data from a simple sine wave function and hoping to predict future points of a sine wave when given some sine wave test data.

Later, however, I would need to supply more complex multivariate independent variable inputs (ex: Inputs: Earth's temperature historical data per each major city,continent and or oceans – Output: Earths average temperature).

I am very noob at machine learning and TensorFlow and I apologize that I have to ask. And if this question is outside of the scope for this guide I completely understand. Thanks again!

Amer
December 16, 2016
Reply

Hi Grant,

I am looking for something very similar. I haven't been able to find something useful so far.
one of the comments in this issue has a simple example

https://github.com/tensorflow/tensorflow/issues/3703

let me know please if you come across something interesting.

Thanks

ray
December 18, 2016
Reply

Nice example that demonstrates how neural network figured out the pattern through supervised learning. But isn't RNN is more on the input that sequence matter? I think the sequence of 1001 and 0011 are both have result = 2 so it is count of 1 but not the sequence that matters. If the example is to like given a sequence and you predict you is next bit will be. It is more fit for RNN. Am I missing something here? thanks!

---

tien
December 22, 2016

Reply

Thank you for your great post. I have a question.
Suppose I don't always use 20 bits for a number then the sequence length may vary. Suppose I split the training input into batches having different sizes but the sequence length of items in a batch is the same, how do I customize your code?
For example:
batch 1: [[[0]] , [[1]]]
batch 2: [[[1] , [0]] , [[1] , [1]]]
batch 3: [[[1] , [0], [0]] , [[1] , [0] , [1]] , [[1] , [1] , [0]] , [[1] , [1], [1]]]

---

ad
March 17, 2017

Reply

For this task you could pad shorter sequences with zeros to make all training input sequences have length 20.

Vigenesh
December 24, 2016

Reply

Wonderful article 🙂

Elliott
February 2, 2017

Reply

Thanks for the article! I was struggling with a starting point for RNNs using Tensorflow and this was really helpful. I appreciate the notes with each code block

---

Wasim
February 21, 2017

Reply

Really impressed with your effort. I went through it and found it really helpful. I am a little confused about our output for the test data. Once I print the prediction tensor by feeding test data to placeholder inside a session. I get a tensor of 21 values. So can anyone explain these values?? I wonder whether these are probabilities, each index shows the number of 1's in the string or not?? Thank you!

Divyansh
April 16, 2017

Reply

Say the output for a particular input is 4, i.e. there are four 1 bits in the input string. So, in this case the output will have the 4th bit as one, to represent that the input belongs to the 4th class (i.e. input has 4 one digits). If the input has 6 one digits, then only the 6th bit will be one in the output, representing that input belonging to the 6th class. Note that there are 21 values in the output because an input string can have 0 ones, 1 ones, ….. , 20 ones. Hope this helps !

Yuan Lukito
February 24, 2017

Reply

Thanks a lot for the article. It helped me to implement LSTM for my project although I am not familiar with Python and Tensorflow. I wonder how to display current error rate (let say, every 50 epochs)?

Wasim
February 27, 2017

Reply

Add if condition inside the loop of EPOCHS, incase it becomes true, the current error will be calculated and displayed after every 50 epochs
for i in range(no_of_epochs):
if i % 50 == 0:
incorrect = sess.run(error,{data: test_input, target: test_output})
print('Epoch {:d} error {:3.1f}%'.format(i + 1, 100 * incorrect))

brebh
March 8, 2017

Reply

how can I find variables values after turning tensorflow code.

David

March 17, 2017

Reply

Hi, nice post, but Im not pretty sure if im reading this code in a proper way.
When we split into 10,000 samples, I guess that train_inputs should be 10,000 samples and train_output should be 10,000 * 20, since there are 20 vectors for each input….if not, please, forgive me. But imagine training input are:
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
by considering this code, the training out is just:
[1, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0]
Instead of:
[1, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0] –> For the 1st vector

[1, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0] –> For the second vector

Looking forward some help,

David

Konuko II
April 12, 2017

Reply

Great post, just wanted to give you a heads up:
The line of code: cell = tf.nn.rnn_cell.LSTMCell(num_hidden,state_is_tuple=True) will throw an error, it is now moved to cell = tf.contrib.rnn.LSTMCell(num_hidden,state_is_tuple=True).
Cheers!

Source: https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/LSTMCell

---

Malti
May 14, 2017

Reply

Good tutorial, but I have one question.
What needs to be changed to divide the network with 3 hidden layers of neurons (100, 50, 50)
thank you.

---

siddhadev
May 24, 2017

Reply

First of all, great post, thank you!!!!

Its worth trying with a smaller batch_size like 8 or even 1 – this should finish the training in just 10-20 epochs 😉
Its also advisable to split the data into three sets – train, validation and test/evaluation. To monitor training you could evaluate the performance on the validation set every few epochs – just to make sure you're not overfiting or training longer (run for more epochs) than required.

A.YULGHUN
May 29, 2017

Reply

cell = tf.contrib.rnn.core_rnn_cell.LSTMCell(num_hidden,state_is_tuple=True)
is working for me.
cell = tf.nn.rnn_cell.LSTMCell(num_hidden,state_is_tuple=True)
give error "model not include rnn_cell".

Yang
June 6, 2017

Reply

Hi,
Thank you so much for so comprehensive tutorial. I also would like to apply this to my problem.
My problem is I want to classify protein into 2 class.
My training example:
For ex:
ARTMCNLIPKHJWENMGFJMSMDDJ 1 #this mean that this protein sequence is of class 1
AAARTUTTNMLWCSFTGH 2 #this means that this protein sequence is of class 2

The problem is the length of the sequences are not equal (not like your example, all input have the same length which is 20)
So what should I do?
Thanks in advance!

Sequence Modelling using Deep Learning(RNNs) – Your data modelling helper
July 2, 2017

Reply

[…] http://monik.in/a-noobs-guide-to-implementing-rnn-lstm-using-tensorflow/ […]

P. Hopkinson
August 29, 2017

Reply

Very interesting example for lots of reasons. Have you noticed that it performs poorly in edge case scenarios where there are either very many or very few zeros? Presumably this is due to bias in the training set which is binomially distributed rather than uniformly distributed.

It is particularly poor at detecting "0 zeros" because it has probably never seen a training example containing no zeros (there is only one such number in 2^20). Similarly for 20 zeros.

Your email address will not be published. Required fields are marked *

Say something nice!

Name*	Email*	Website

Post Comment

☐ Notify me of follow-up comments by email.

☐ Notify me of new posts by email.