
(https://www.bigdatauniversity.com)

RECURRENT NETWORKS and LSTM IN DEEP LEARNING

Applying Recurrent Neural Networks/LSTM for Language Modeling

Hello and welcome to this part. In this notebook, we will go over the topic of Language Modelling, and create a Recurrent Neural Network model based on the Long Short-Term Memory unit to train and benchmark on the Penn Treebank dataset. By the end of this notebook, you should be able to understand how TensorFlow builds and executes a RNN model for Language Modelling.

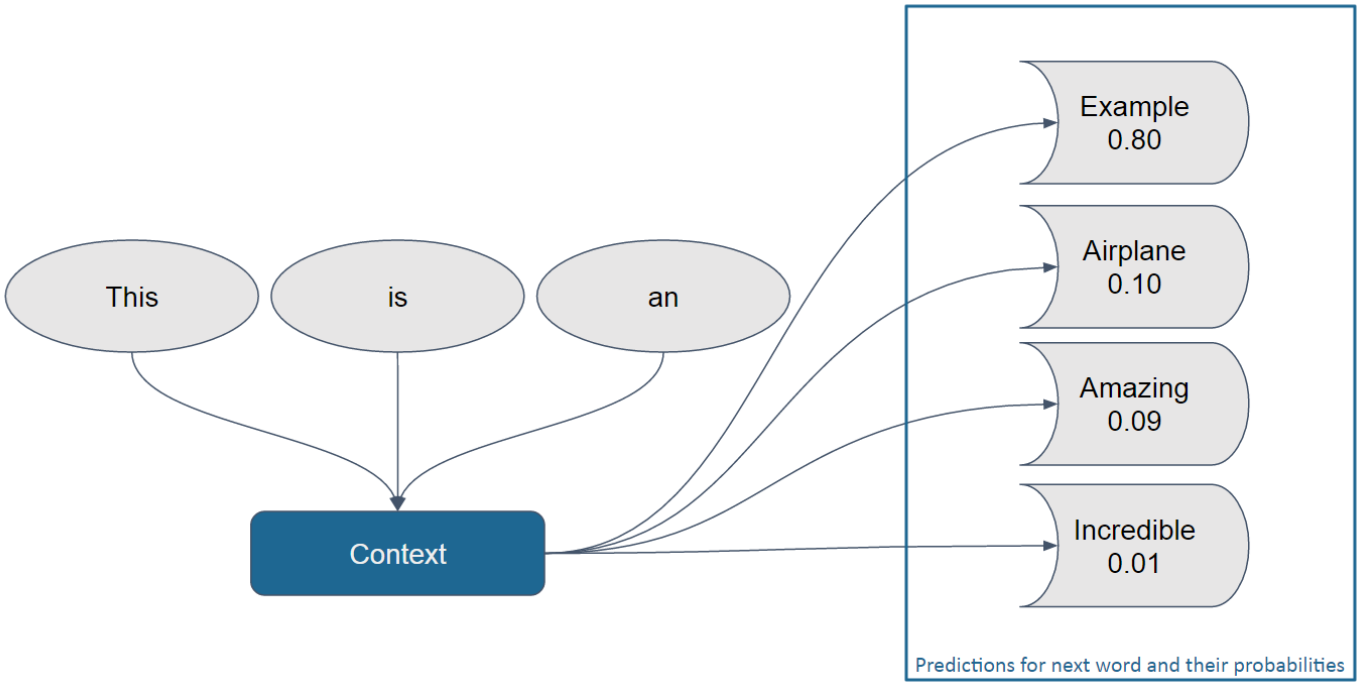
The Objective

By now, you should have an understanding of how Recurrent Networks work -- a specialized model to process sequential data by keeping track of the "state" or context. In this notebook, we go over a TensorFlow code snippet for creating a model focused on **Language Modelling** -- a very relevant task that is the cornerstone of many different linguistic problems such as **Speech Recognition, Machine Translation and Image Captioning**. For this, we will be using the Penn Treebank dataset, which is an often-used dataset for benchmarking Language Modelling models.

Table of Contents

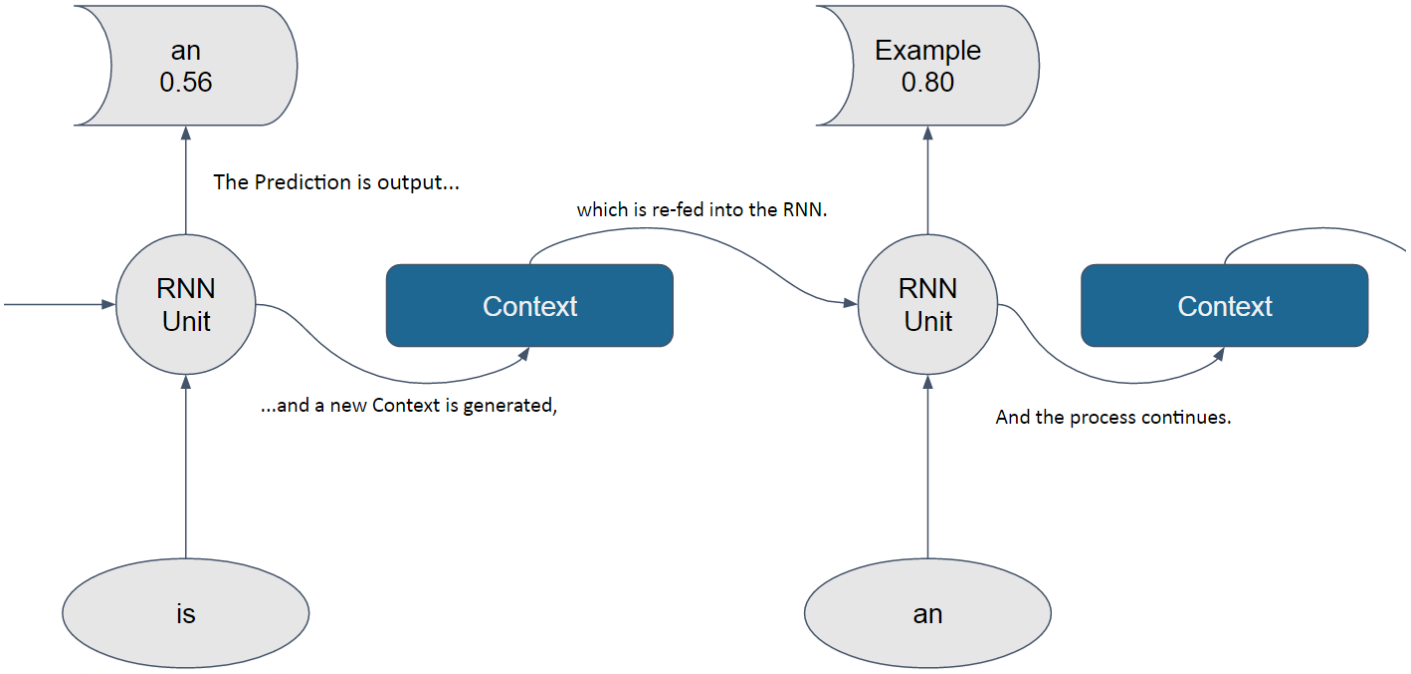
What exactly is Language Modelling?

Language Modelling, to put it simply, **is the task of assigning probabilities to sequences of words**. This means that, given a context of one or a sequence of words in the language the model was trained on, the model should provide the next most probable words or sequence of words that follows from the given sequence of words the sentence. Language Modelling is one of the most important tasks in Natural Language Processing.



Example of a sentence being predicted

In this example, one can see the predictions for the next word of a sentence, given the context "This is an". As you can see, this boils down to a sequential data analysis task -- you are given a word or a sequence of words (the input data), and, given the context (the state), you need to find out what is the next word (the prediction). This kind of analysis is very important for language-related tasks such as **Speech Recognition, Machine Translation, Image Captioning, Text Correction** and many other very relevant problems.


an
0.56
Example
0.80
The Prediction is output...
which is re-fed into the RNN.
RNN
Unit
Context
RNN
Unit
Context
...and a new Context is generated,
And the process continues.
is
an

The Penn Treebank dataset

Historically, datasets big enough for Natural Language Processing are hard to come by. This is in part due to the necessity of the sentences to be broken down and tagged with a certain degree of correctness -- or else the models trained on it won't be able to be correct at all. This means that we need a **large amount of data, annotated by or at least corrected by humans**. This is, of course, not an easy task at all.

The Penn Treebank, or PTB for short, is a dataset maintained by the University of Pennsylvania. It is *huge* -- there are over **four million and eight hundred thousand** annotated words in it, all corrected by humans. It is composed of many different sources, from abstracts of Department of Energy papers to texts from the Library of America. Since it is verifiably correct and of such a huge size, the Penn Treebank is commonly used as a benchmark dataset for Language Modelling.

The dataset is divided in different kinds of annotations, such as Piece-of-Speech, Syntactic and Semantic skeletons. For this example, we will simply use a sample of clean, non-annotated words (with the exception of one tag -- `<unk>` , which is used for rare words such as uncommon proper nouns) for our model. This means that we just want to predict what the next words would be, not what they mean in context or their classes on a given sentence.

Example of text from the dataset we are going to use, **ptb.train**

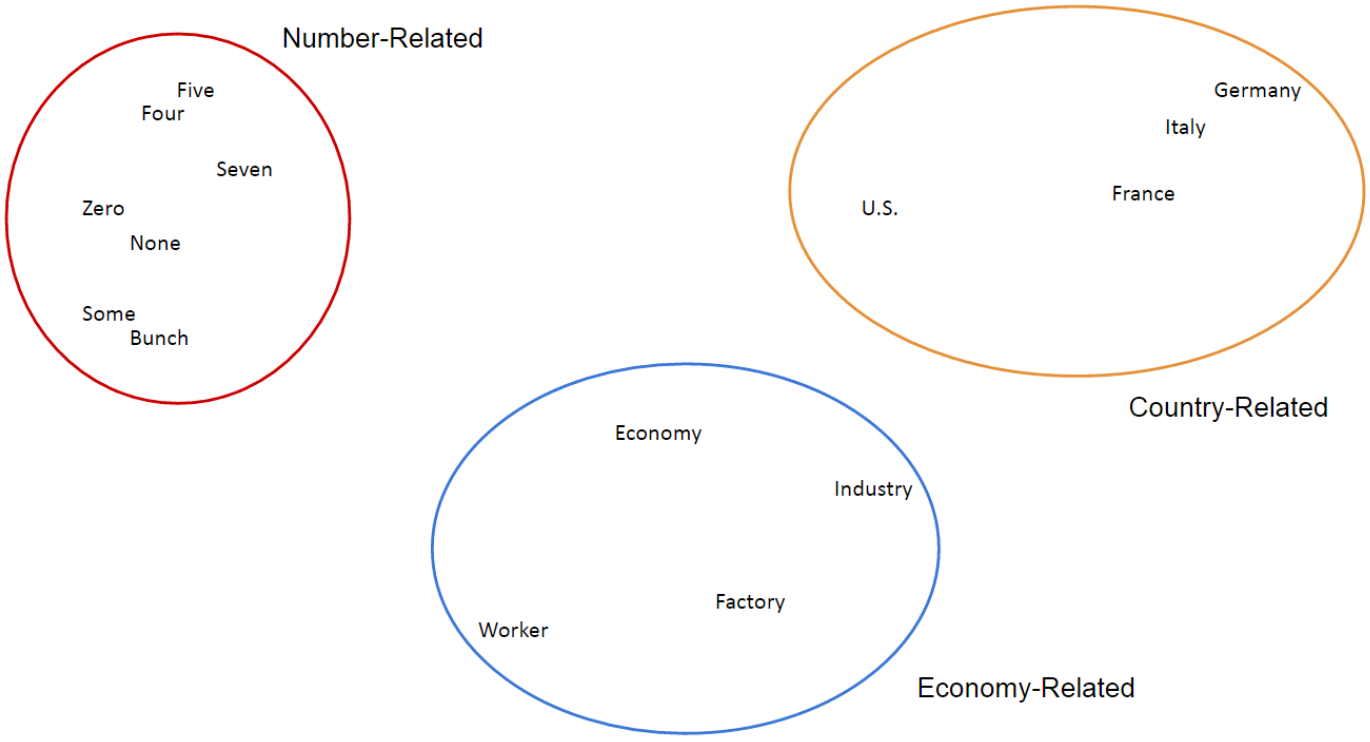
> the percentage of lung cancer deaths among the workers at the west `<unk>` mass. paper factory appears to be the highest for any asbestos workers studied in western industrialized countries he said the plant which is owned by `<unk>` & `<unk>` co. was under contract with `<unk>` to make the cigarette filters the finding probably will support those who argue that the U.S. should regulate the class of asbestos including `<unk>` more `<unk>` than the common kind of asbestos `<unk>` found in most schools and other buildings dr. `<unk>` said

Word Embeddings

For better processing, in this example, we will make use of **word embeddings** (https://www.tensorflow.org/tutorials/word2vec/), which is **a way of representing sentence structures or words as n-dimensional vectors (where n is a reasonably high number, such as 200 or 500) of real numbers**. Basically, we will assign each word a randomly-initialized vector, and input those into the network to be processed. After a number of iterations, these vectors are expected to assume values that help the network to correctly predict what it needs to -- in our case, the probable next word in the sentence. This is shown to be a very effective task in Natural Language Processing, and is a commonplace practice.

$$Vec("Example") = [0.02, 0.00, 0.00, 0.92, 0.30, \ldots]$$

Word Embedding tends to group up similarly used words *reasonably* close together in the vectorial space. For example, if we use T-SNE (a dimensional reduction visualization algorithm) to flatten the dimensions of our vectors into a 2-dimensional space and plot these words in a 2-dimensional space, we might see something like this:



T-SNE Mockup with clusters marked for easier visualization

As you can see, words that are frequently used together, in place of each other, or in the same places as them tend to be grouped together -- being closer together the higher they are correlated. For example, "None" is pretty semantically close to "Zero", while a phrase that uses "Italy", you could probably also fit "Germany" in it, with little damage to the sentence structure. The vectorial "closeness" for similar words like this is a great indicator of a well-built model.

We need to import the necessary modules for our code. We need **`numpy`** and **`tensorflow`**, obviously. Additionally, we can import directly the **`tensorflow.models.rnn`** model, which includes the function for building RNNs, and **`tensorflow.models.rnn.ptb.reader`** which is the helper module for getting the input data from the dataset we just downloaded.

If you want to learn more take a look at
https://github.com/tensorflow/models/blob/master/tutorials/rnn/ptb/reader.py
(https://github.com/tensorflow/models/blob/master/tutorials/rnn/ptb/reader.py)

In [1]:

```
import time
import numpy as np
import tensorflow as tf
```

```
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:519: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:520: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:521: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:522: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:523: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:528: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
```

In [2]:

```
!mkdir data
!wget -q -O data/ptb.zip https://ibm.box.com/shared/static/z2yvmhbskc45xd2a9a4kkn6h
g4g4kj5r.zip
!unzip -o data/ptb.zip -d data
!cp data/ptb/reader.py .

import reader
```

```
Archive:  data/ptb.zip
   creating: data/ptb/
  inflating: data/ptb/reader.py
   creating: data/__MACOSX/
   creating: data/__MACOSX/ptb/
  inflating: data/__MACOSX/ptb/._reader.py
  inflating: data/__MACOSX/._ptb
```

Building the LSTM model for Language Modeling

Now that we know exactly what we are doing, we can start building our model using TensorFlow. The very first thing we need to do is download and extract the `simple-examples` dataset, which can be done by executing the code cell below.

In [3]:

```
!wget http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz
!tar xzf simple-examples.tgz -C data/
```

```
--2020-06-08 06:09:48--  http://www.fit.vutbr.cz/~imikolov/rnnlm/simple
-examples.tgz
Resolving www.fit.vutbr.cz (www.fit.vutbr.cz)... 147.229.9.23, 2001:67
c:1220:809::93e5:917
Connecting to www.fit.vutbr.cz (www.fit.vutbr.cz)|147.229.9.23|:80... c
onnected.
HTTP request sent, awaiting response... 200 OK
Length: 34869662 (33M) [application/x-gtar]
Saving to: ‘simple-examples.tgz’

simple-examples.tgz 100%[===================>]  33.25M  3.94MB/s    in
9.4s

2020-06-08 06:09:58 (3.54 MB/s) - ‘simple-examples.tgz’ saved [3486966
2/34869662]
```

Additionally, for the sake of making it easy to play around with the model's hyperparameters, we can declare them beforehand. Feel free to change these -- you will see a difference in performance each time you change those!

In [4]:

```
#Initial weight scale
init_scale = 0.1
#Initial learning rate
learning_rate = 1.0
#Maximum permissible norm for the gradient (For gradient clipping -- another measur
e against Exploding Gradients)
max_grad_norm = 5
#The number of layers in our model
num_layers = 2
#The total number of recurrence steps, also known as the number of layers when our
 RNN is "unfolded"
num_steps = 20
#The number of processing units (neurons) in the hidden layers
hidden_size_l1 = 256
hidden_size_l2 = 128
#The maximum number of epochs trained with the initial learning rate
max_epoch_decay_lr = 4
#The total number of epochs in training
max_epoch = 15
#The probability for keeping data in the Dropout Layer (This is an optimization, bu
t is outside our scope for this notebook!)
#At 1, we ignore the Dropout Layer wrapping.
keep_prob = 1
#The decay for the learning rate
decay = 0.5
#The size for each batch of data
batch_size = 60
#The size of our vocabulary
vocab_size = 10000
embeding_vector_size = 200
#Training flag to separate training from testing
is_training = 1
#Data directory for our dataset
data_dir = "data/simple-examples/data/"
```

Some clarifications for LSTM architecture based on the arguments:

Network structure:

- In this network, the number of LSTM cells are 2. To give the model more expressive power, we can add multiple layers of LSTMs to process the data. The output of the first layer will become the input of the second and so on.
- The recurrence steps is 20, that is, when our RNN is "Unfolded", the recurrence step is 20.
- the structure is like:
  - 200 input units -> [200x200] Weight -> 200 Hidden units (first layer) -> [200x200] Weight matrix -> 200 Hidden units (second layer) -> [200] weight Matrix -> 200 unit output

  Input layer:

  - The network has 200 input units.
  - Suppose each word is represented by an embedding vector of dimensionality e=200. The input layer of each cell will have 200 linear units. These e=200 linear units are connected to each of the h=200 LSTM units in the hidden layer (assuming there is only one hidden layer, though our case has 2 layers).
  - The input shape is [batch_size, num_steps], that is [30x20]. It will turn into [30x20x200] after embedding, and then 20x[30x200]

  Hidden layer:

  - Each LSTM has 200 hidden units which is equivalent to the dimensionality of the embedding words and output.

There is a lot to be done and a ton of information to process at the same time, so go over this code slowly. It may seem complex at first, but if you try to apply what you just learned about language modelling to the code you see, you should be able to understand it.

This code is adapted from the PTBModel (https://github.com/tensorflow/models) example bundled with the TensorFlow source code.

Train data
The story starts from data:

- Train data is a list of words, of size 929589, represented by numbers, e.g. [9971, 9972, 9974, 9975,...]
- We read data as mini-batch of size b=30. Assume the size of each sentence is 20 words (num_steps = 20). Then it will take $$floor(\frac{N}{b \times h})+1=1548$$ iterations for the learner to go through all sentences once. Where N is the size of the list of words, b is batch size, andh is size of each sentence. So, the number of iterators is 1548
- Each batch data is read from train dataset of size 600, and shape of [30x20]

First we start an interactive session:

In [5]:

```
session = tf.InteractiveSession()
```

In [6]:

```
# Reads the data and separates it into training data, validation data and testing d
ata
raw_data = reader.ptb_raw_data(data_dir)
train_data, valid_data, test_data, vocab, word_to_id = raw_data
```

In [7]:

```
len(train_data)
```

Out[7]:

```
929589
```

In [8]:

```
def id_to_word(id_list):
    line = []
    for w in id_list:
        for word, wid in word_to_id.items():
            if wid == w:
                line.append(word)
    return line


print(id_to_word(train_data[0:100]))
```

```
['aer', 'banknote', 'berlitz', 'calloway', 'centrust', 'cluett', 'froms
tein', 'gitano', 'guterman', 'hydro-quebec', 'ipo', 'kia', 'memotec',
'mlx', 'nahb', 'punts', 'rake', 'regatta', 'rubens', 'sim', 'snack-foo
d', 'ssangyong', 'swapo', 'wachter', '<eos>', 'pierre', '<unk>', 'N',
'years', 'old', 'will', 'join', 'the', 'board', 'as', 'a', 'nonexecutiv
e', 'director', 'nov.', 'N', '<eos>', 'mr.', '<unk>', 'is', 'chairman',
'of', '<unk>', 'n.v.', 'the', 'dutch', 'publishing', 'group', '<eos>',
'rudolph', '<unk>', 'N', 'years', 'old', 'and', 'former', 'chairman',
'of', 'consolidated', 'gold', 'fields', 'plc', 'was', 'named', 'a', 'no
nexecutive', 'director', 'of', 'this', 'british', 'industrial', 'conglo
merate', '<eos>', 'a', 'form', 'of', 'asbestos', 'once', 'used', 'to',
'make', 'kent', 'cigarette', 'filters', 'has', 'caused', 'a', 'high',
'percentage', 'of', 'cancer', 'deaths', 'among', 'a', 'group', 'of']
```

Lets just read one mini-batch now and feed our network:

In [9]:

```
itera = reader.ptb_iterator(train_data, batch_size, num_steps)
first_touple = itera.__next__()
x = first_touple[0]
y = first_touple[1]
```

In [10]:

```
x.shape
```

Out[10]:

```
(60, 20)
```

Lets look at 3 sentences of our input x:

In [11]:

```
x[0:3]
```

Out[11]:

```
array([[9970, 9971, 9972, 9974, 9975, 9976, 9980, 9981, 9982, 9983, 998
4,
        9986, 9987, 9988, 9989, 9991, 9992, 9993, 9994, 9995],
       [ 901,   33, 3361,    8, 1279,  437,  597,    6,  261, 4276, 108
9,
           8, 2836,    2,  269,    4, 5526,  241,   13, 2420],
       [2654,    6,  334, 2886,    4,    1,  233,  711,  834,   11,  13
0,
         123,    7,  514,    2,   63,   10,  514,    8,  605]],
      dtype=int32)
```

we define 2 place holders to feed them with mini-batchs, that is x and y:

In [12]:

```
_input_data = tf.placeholder(tf.int32, [batch_size, num_steps]) #[30#20]
_targets = tf.placeholder(tf.int32, [batch_size, num_steps]) #[30#20]
```

Lets define a dictionary, and use it later to feed the placeholders with our first mini-batch:

In [13]:

```
feed_dict = {_input_data:x, _targets:y}
```

For example, we can use it to feed _input_data :

In [14]:

```
session.run(_input_data, feed_dict)
```

Out[14]:

```
array([[9970, 9971, 9972, ..., 9993, 9994, 9995],
       [ 901,   33, 3361, ...,  241,   13, 2420],
       [2654,    6,  334, ...,  514,    8,  605],
       ...,
       [7831,   36, 1678, ...,    4, 4558,  157],
       [  59, 2070, 2433, ...,  400,    1, 1173],
       [2097,    3,    2, ..., 2043,   23,    1]], dtype=int32)
```

In this step, we create the stacked LSTM, which is a 2 layer LSTM network:

In [15]:

```
lstm_cell_l1 = tf.contrib.rnn.BasicLSTMCell(hidden_size_l1, forget_bias=0.0)
lstm_cell_l2 = tf.contrib.rnn.BasicLSTMCell(hidden_size_l2, forget_bias=0.0)
stacked_lstm = tf.contrib.rnn.MultiRNNCell([lstm_cell_l1, lstm_cell_l2])
```

Also, we initialize the states of the nework:

_initial_state

For each LCTM, there are 2 state matrices, c_state and m_state. c_state and m_state represent "Memory State" and "Cell State". Each hidden layer, has a vector of size 30, which keeps the states. so, for 200 hidden units in each LSTM, we have a matrix of size [30x200]

In [16]:

```
_initial_state = stacked_lstm.zero_state(batch_size, tf.float32)
_initial_state
```

Out[16]:

```
(LSTMStateTuple(c=<tf.Tensor 'MultiRNNCellZeroState/BasicLSTMCellZeroSt
ate/zeros:0' shape=(60, 256) dtype=float32>, h=<tf.Tensor 'MultiRNNCell
ZeroState/BasicLSTMCellZeroState/zeros_1:0' shape=(60, 256) dtype=float
32>),
 LSTMStateTuple(c=<tf.Tensor 'MultiRNNCellZeroState/BasicLSTMCellZeroSt
ate_1/zeros:0' shape=(60, 128) dtype=float32>, h=<tf.Tensor 'MultiRNNCe
llZeroState/BasicLSTMCellZeroState_1/zeros_1:0' shape=(60, 128) dtype=f
loat32>))
```

Lets look at the states, though they are all zero for now:

In [17]:

```
session.run(_initial_state, feed_dict)
```

Out[17]:

```
(LSTMStateTuple(c=array([[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]], dtype=float32), h=array([[0.,
0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)),
 LSTMStateTuple(c=array([[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]], dtype=float32), h=array([[0.,
0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)))
```

Embeddings

We have to convert the words in our dataset to vectors of numbers. The traditional approach is to use one-hot encoding method that is usually used for converting categorical values to numerical values. However, One-hot encoded vectors are high-dimensional, sparse and in a big dataset, computationally inefficient. So, we use word2vec approach. It is, in fact, a layer in our LSTM network, where the word IDs will be represented as a dense representation before feeding to the LSTM.

The embedded vectors also get updated during the training process of the deep neural network. We create the embeddings for our input data. **embedding_vocab** is matrix of [10000x200] for all 10000 unique words.

In [18]:

```
embedding_vocab = tf.get_variable("embedding_vocab", [vocab_size, embeding_vector_s
ize])  #[10000x200]
```

Lets initialize the `embedding_words` variable with random values.

In [19]:

```
session.run(tf.global_variables_initializer())
session.run(embedding_vocab)
```

Out[19]:

```
array([[-0.00902626, -0.01734165, -0.00033544, ...,  0.00134134,
         0.00113114,  0.02395696],
       [ 0.00678663, -0.01678014, -0.0035552 , ...,  0.0191374 ,
        -0.00066233, -0.0121508 ],
       [-0.01157477, -0.00222771, -0.00549181, ...,  0.00282263,
         0.01272673, -0.00118867],
       ...,
       [-0.01173353,  0.01133806, -0.0113985 , ...,  0.01417069,
         0.00370108,  0.00238578],
       [-0.0065457 , -0.02346947,  0.01306494, ..., -0.0132121 ,
         0.00608167, -0.00072341],
       [ 0.01207733,  0.00166364,  0.02398165, ...,  0.01180564,
        -0.00096454,  0.01444308]], dtype=float32)
```

embedding_lookup() finds the embedded values for our batch of 30x20 words. It goes to each row of `input_data`, and for each word in the row/sentence, finds the correspond vector in `embedding_dic`.

It creates a [30x20x200] tensor, so, the first element of **inputs** (the first sentence), is a matrix of 20x200, which each row of it, is vector representing a word in the sentence.

In [20]:

```
# Define where to get the data for our embeddings from
inputs = tf.nn.embedding_lookup(embedding_vocab, _input_data)  #shape=(30, 20, 200)
inputs
```

Out[20]:

```
<tf.Tensor 'embedding_lookup:0' shape=(60, 20, 200) dtype=float32>
```

In [21]:

```
session.run(inputs[0], feed_dict)
```

Out[21]:

```
array([[-0.0188488 , -0.02403067, -0.00066788, ...,  0.01974649,
         0.01159596, -0.01773928],
       [ 0.00789267, -0.0013746 , -0.01649964, ...,  0.00114487,
         0.00557627,  0.00228115],
       [ 0.00171588,  0.0079329 ,  0.00801094, ...,  0.01252862,
        -0.01008555, -0.02204007],
       ...,
       [ 0.01181928,  0.00866624, -0.00700261, ...,  0.00892083,
        -0.01091242, -0.00548531],
       [-0.02101085,  0.00215198, -0.02127971, ..., -0.01380314,
        -0.02237165,  0.01724135],
       [ 0.0124624 , -0.00685833,  0.01710434, ...,  0.0101826 ,
        -0.0042201 , -0.02325571]], dtype=float32)
```

Constructing Recurrent Neural Networks

tf.nn.dynamic_rnn() creates a recurrent neural network using **stacked_lstm.**

The input should be a Tensor of shape: [batch_size, max_time, embedding_vector_size], in our case it would be (30, 20, 200)

This method, returns a pair (outputs, new_state) where:

- **outputs:** is a length T list of outputs (one for each input), or a nested tuple of such elements.
- **new_state:** is the final state.

In [22]:

```
outputs, new_state =  tf.nn.dynamic_rnn(stacked_lstm, inputs, initial_state=_initial_state)
```

so, lets look at the outputs. The output of the stackedLSTM comes from 200 hidden_layer, and in each time step(=20), one of them get activated. we use the linear activation to map the 200 hidden layer to a [?x10 matrix]

In [23]:

```
outputs
```

Out[23]:

```
<tf.Tensor 'rnn/transpose_1:0' shape=(60, 20, 128) dtype=float32>
```

In [24]:

```
session.run(tf.global_variables_initializer())
session.run(outputs[0], feed_dict)
```

Out[24]:

```
array([[-1.7726891e-04, -1.0701056e-04,  3.6677535e-04, ...,
        -1.7957884e-04, -3.0365729e-04,  4.2002602e-04],
       [-5.9756736e-04, -2.0404220e-04, -3.6607325e-05, ...,
        -2.0660809e-04, -6.8695022e-04,  6.2485581e-04],
       [-1.7328798e-04,  4.5081713e-05, -1.9513239e-04, ...,
         1.7447531e-04, -1.1427420e-03,  4.6506320e-04],
       ...,
       [ 1.5934210e-04,  3.3947386e-04, -6.2543608e-04, ...,
         3.1514876e-04,  6.3632790e-04,  3.4656539e-04],
       [ 4.5750348e-05,  6.9912616e-04, -6.5783592e-04, ...,
         1.2224721e-03,  6.8250322e-04,  6.6462328e-04],
       [-9.5398558e-05,  1.2786839e-04, -1.2960391e-03, ...,
         1.1640216e-03, -1.6724439e-04,  1.2024270e-03]], dtype=float32)
```

we need to flatten the outputs to be able to connect it softmax layer. Lets reshape the output tensor from [30 x 20 x 200] to [600 x 200].

Notice: Imagine our output is 3-d tensor as following (of course each sen_x_word_y is a an embedded vector by itself):

- sentence 1: [[sen1word1], [sen1word2], [sen1word3], ..., [sen1word20]]
- sentence 2: [[sen2word1], [sen2word2], [sen2word3], ..., [sen2word20]]
- sentence 3: [[sen3word1], [sen3word2], [sen3word3], ..., [sen3word20]]
- ...
- sentence 30: [[sen30word1], [sen30word2], [sen30word3], ..., [sen30word20]]

Now, the flatten would convert this 3-dim tensor to:

[[sen1word1], [sen1word2], [sen1word3], ..., [sen1word20],[sen2word1], [sen2word2], [sen2word3], ..., [sen2word20], ..., [sen30word20]]

In [25]:

```
output = tf.reshape(outputs, [-1, hidden_size_l2])
output
```

Out[25]:

```
<tf.Tensor 'Reshape:0' shape=(1200, 128) dtype=float32>
```

logistic unit

Now, we create a logistic unit to return the probability of the output word in our vocabulary with 1000 words.

$$Softmax = [600 \times 200] * [200 \times 1000] + [1 \times 1000] \Longrightarrow [600 \times 1000]$$

In [26]:

```
softmax_w = tf.get_variable("softmax_w", [hidden_size_l2, vocab_size]) #[200x1000]
softmax_b = tf.get_variable("softmax_b", [vocab_size]) #[1x1000]
logits = tf.matmul(output, softmax_w) + softmax_b
prob = tf.nn.softmax(logits)
```

Lets look at the probability of observing words for t=0 to t=20:

In [27]:

```
session.run(tf.global_variables_initializer())
output_words_prob = session.run(prob, feed_dict)
print("shape of the output: ", output_words_prob.shape)
print("The probability of observing words in t=0 to t=20", output_words_prob[0:20])
```

```
shape of the output:  (1200, 10000)
The probability of observing words in t=0 to t=20 [[1.01029553e-04 9.95522787e-05 1.01206802e
-04 ... 1.01532933e-04
  1.01608828e-04 1.01440601e-04]
 [1.01029749e-04 9.95510345e-05 1.01205071e-04 ... 1.01531652e-04
  1.01611797e-04 1.01442041e-04]
 [1.01025107e-04 9.95457303e-05 1.01206242e-04 ... 1.01534431e-04
  1.01610131e-04 1.01444093e-04]
 ...
 [1.01031335e-04 9.95724986e-05 1.01201775e-04 ... 1.01539612e-04
  1.01604594e-04 1.01449667e-04]
 [1.01039768e-04 9.95605005e-05 1.01201120e-04 ... 1.01539845e-04
  1.01607453e-04 1.01452351e-04]
 [1.01040889e-04 9.95547962e-05 1.01200298e-04 ... 1.01546764e-04
  1.01612044e-04 1.01452279e-04]]
```

Prediction

What is the word correspond to the probability output? Lets use the maximum probability:

In [28]:

```
np.argmax(output_words_prob[0:20], axis=1)
```

Out[28]:

```
array([1103, 1103, 1103, 9132, 8670, 9132, 6758, 6758,  244, 7364, 7364,
       8223, 7474, 7981, 7364, 7364, 6577, 6577, 6577, 6577])
```

So, what is the ground truth for the first word of first sentence?

In [29]:

```
y[0]
```

Out[29]:

```
array([9971, 9972, 9974, 9975, 9976, 9980, 9981, 9982, 9983, 9984, 9986,
       9987, 9988, 9989, 9991, 9992, 9993, 9994, 9995, 9996], dtype=int32)
```

Also, you can get it from target tensor, if you want to find the embedding vector:

In [30]:

```
targ = session.run(_targets, feed_dict)
targ[0]
```

Out[30]:

```
array([9971, 9972, 9974, 9975, 9976, 9980, 9981, 9982, 9983, 9984, 9986,
       9987, 9988, 9989, 9991, 9992, 9993, 9994, 9995, 9996], dtype=int32)
```

How similar the predicted words are to the target words?

Objective function

Now we have to define our objective function, to calculate the similarity of predicted values to ground truth, and then, penalize the model with the error. Our objective is to minimize loss function, that is, to minimize the average negative log probability of the target words:

$$\text{loss} = -\frac{1}{N}\sum_{i=1}^{N} \ln p_{\text{target}_i}$$

This function is already implemented and available in TensorFlow through **sequence_loss_by_example**. It calculates the weighted cross-entropy loss for **logits** and the **target** sequence.

The arguments of this function are:

- logits: List of 2D Tensors of shape [batch_size x num_decoder_symbols].
- targets: List of 1D batch-sized int32 Tensors of the same length as logits.
- weights: List of 1D batch-sized float-Tensors of the same length as logits.

In [31]:

```
loss = tf.contrib.legacy_seq2seq.sequence_loss_by_example([logits], [tf.reshape(_targets, [-1])],[tf.on
es([batch_size * num_steps])])
```

loss is a 1D batch-sized float Tensor [600x1]: The log-perplexity for each sequence. Lets look at the first 10 values of loss:

In [32]:

```
session.run(loss, feed_dict)[:10]
```

Out[32]:

```
array([9.200306, 9.208275, 9.213675, 9.222969, 9.20124 , 9.219968,
       9.215756, 9.205656, 9.20363 , 9.217363], dtype=float32)
```

Now, we define loss as average of the losses:

In [33]:

```
cost = tf.reduce_sum(loss) / batch_size
session.run(tf.global_variables_initializer())
session.run(cost, feed_dict)
```

Out[33]:

```
184.19876
```

Training

To do training for our network, we have to take the following steps:

1. Define the optimizer.

2. Extract variables that are trainable.

3. Calculate the gradients based on the loss function.

4. Apply the optimizer to the variables/gradients tuple.

1. Define Optimizer

GradientDescentOptimizer constructs a new gradient descent optimizer. Later, we use constructed **optimizer** to compute gradients for a loss and apply gradients to variables.

In [34]:

```
# Create a variable for the learning rate
lr = tf.Variable(0.0, trainable=False)
# Create the gradient descent optimizer with our learning rate
optimizer = tf.train.GradientDescentOptimizer(lr)
```

2. Trainable Variables

Defining a variable, if you passed *trainable=True*, the variable constructor automatically adds new variables to the graph collection **GraphKeys.TRAINABLE_VARIABLES.** Now, using *tf.trainable_variables()* you can get all variables created with **trainable=True.**

In [35]:

```
# Get all TensorFlow variables marked as "trainable" (i.e. all of them except _lr, which we just create
d)
tvars = tf.trainable_variables()
tvars
```

Out[35]:

```
[<tf.Variable 'embedding_vocab:0' shape=(10000, 200) dtype=float32_ref>,
 <tf.Variable 'rnn/multi_rnn_cell/cell_0/basic_lstm_cell/kernel:0' shape=(456, 1024) dtype=fl
oat32_ref>,
 <tf.Variable 'rnn/multi_rnn_cell/cell_0/basic_lstm_cell/bias:0' shape=(1024,) dtype=float32_
ref>,
 <tf.Variable 'rnn/multi_rnn_cell/cell_1/basic_lstm_cell/kernel:0' shape=(384, 512) dtype=flo
at32_ref>,
 <tf.Variable 'rnn/multi_rnn_cell/cell_1/basic_lstm_cell/bias:0' shape=(512,) dtype=float32_r
ef>,
 <tf.Variable 'softmax_w:0' shape=(128, 10000) dtype=float32_ref>,
 <tf.Variable 'softmax_b:0' shape=(10000,) dtype=float32_ref>]
```

Note: we can find the name and scope of all variables:

In [36]:

```
[v.name for v in tvars]
```

Out[36]:

```
['embedding_vocab:0',
 'rnn/multi_rnn_cell/cell_0/basic_lstm_cell/kernel:0',
 'rnn/multi_rnn_cell/cell_0/basic_lstm_cell/bias:0',
 'rnn/multi_rnn_cell/cell_1/basic_lstm_cell/kernel:0',
 'rnn/multi_rnn_cell/cell_1/basic_lstm_cell/bias:0',
 'softmax_w:0',
 'softmax_b:0']
```

3. Calculate the gradients based on the loss function

Gradient
:
The gradient of a function is the slope of its derivative (line), or in other words, the rate of change of a function. It's a vector (a direction to move) that points in the direction of greatest increase of the function, and calculated by the **derivative** operation.

First lets recall the gradient function using an toy example:
$$ z = \left(2x^2 + 3xy\right)$$

In [37]:

```
var_x = tf.placeholder(tf.float32)
var_y = tf.placeholder(tf.float32)
func_test = 2.0 * var_x * var_x + 3.0 * var_x * var_y
session.run(tf.global_variables_initializer())
session.run(func_test, {var_x:1.0,var_y:2.0})
```

Out[37]:

```
8.0
```

The **tf.gradients()** function allows you to compute the symbolic gradient of one tensor with respect to one or more other tensors—including variables. **tf.gradients(func, xs)** constructs symbolic partial derivatives of sum of **func** w.r.t. *x* in **xs.**

Now, lets look at the derivitive w.r.t. **var_x:**

$$ \frac{\partial \:}{\partial \:x}\left(2x^2 + 3xy\right) = 4x + 3y $$

In [38]:

```
var_grad = tf.gradients(func_test, [var_x])
session.run(var_grad, {var_x:1.0,var_y:2.0})
```

Out[38]:

```
[10.0]
```

the derivative w.r.t. **var_y:**

$$ \frac{\partial \:}{\partial \:x}\left(2x^2 + 3xy\right) = 3x $$

In [39]:

```
var_grad = tf.gradients(func_test, [var_y])
session.run(var_grad, {var_x:1.0, var_y:2.0})
```

Out[39]:

```
[3.0]
```

Now, we can look at gradients w.r.t all variables:

In [40]:

```
tf.gradients(cost, tvars)
```

Out[40]:

```
[<tensorflow.python.framework.ops.IndexedSlices at 0x7f1fa1868c18>,
 <tf.Tensor 'gradients_2/rnn/while/rnn/multi_rnn_cell/cell_0/basic_lstm_cell/MatMul/Enter_gra
d/b_acc_3:0' shape=(456, 1024) dtype=float32>,
 <tf.Tensor 'gradients_2/rnn/while/rnn/multi_rnn_cell/cell_0/basic_lstm_cell/BiasAdd/Enter_gr
ad/b_acc_3:0' shape=(1024,) dtype=float32>,
 <tf.Tensor 'gradients_2/rnn/while/rnn/multi_rnn_cell/cell_1/basic_lstm_cell/MatMul/Enter_gra
d/b_acc_3:0' shape=(384, 512) dtype=float32>,
 <tf.Tensor 'gradients_2/rnn/while/rnn/multi_rnn_cell/cell_1/basic_lstm_cell/BiasAdd/Enter_gr
ad/b_acc_3:0' shape=(512,) dtype=float32>,
 <tf.Tensor 'gradients_2/MatMul_grad/MatMul_1:0' shape=(128, 10000) dtype=float32>,
 <tf.Tensor 'gradients_2/add_grad/Reshape_1:0' shape=(10000,) dtype=float32>]
```

In [41]:

```
grad_t_list = tf.gradients(cost, tvars)
#sess.run(grad_t_list,feed_dict)
```

now, we have a list of tensors, t-list. We can use it to find clipped tensors. **clip_by_global_norm** clips values of multiple tensors by the ratio of the sum of their norms.

clip_by_global_norm get *t-list* as input and returns 2 things:

- a list of clipped tensors, so called *list_clipped*
- the global norm (global_norm) of all tensors in t_list

In [42]:

```
# Define the gradient clipping threshold
grads, _ = tf.clip_by_global_norm(grad_t_list, max_grad_norm)
grads
```

Out[42]:

```
[<tensorflow.python.framework.ops.IndexedSlices at 0x7f1fa180d128>,
 <tf.Tensor 'clip_by_global_norm/clip_by_global_norm/_1:0' shape=(456, 1024) dtype=float32>,
 <tf.Tensor 'clip_by_global_norm/clip_by_global_norm/_2:0' shape=(1024,) dtype=float32>,
 <tf.Tensor 'clip_by_global_norm/clip_by_global_norm/_3:0' shape=(384, 512) dtype=float32>,
 <tf.Tensor 'clip_by_global_norm/clip_by_global_norm/_4:0' shape=(512,) dtype=float32>,
 <tf.Tensor 'clip_by_global_norm/clip_by_global_norm/_5:0' shape=(128, 10000) dtype=float32>,
 <tf.Tensor 'clip_by_global_norm/clip_by_global_norm/_6:0' shape=(10000,) dtype=float32>]
```

In [43]:

```
session.run(grads, feed_dict)
```

Out[43]:

```
[IndexedSlicesValue(values=array([[-1.4761964e-06,  2.8837335e-06, -8.7622493e-06, ...,
         -7.6902579e-06,  1.6152146e-05, -5.2746168e-06],
        [-1.0676466e-05, -2.7942906e-06, -9.7363609e-06, ...,
         -6.9873317e-06,  1.7574257e-05, -1.6554552e-06],
        [-1.0419625e-05, -2.0251307e-06, -1.2877392e-05, ...,
         -7.5318981e-06,  1.4133769e-05,  5.9265672e-06],
        ...,
        [ 2.6873317e-06, -5.0842627e-06,  6.8582581e-06, ...,
         -2.4063370e-06, -2.9662348e-07,  5.8100920e-07],
        [-2.0702328e-06, -3.2271414e-06,  4.8046707e-07, ...,
          4.1231187e-06, -3.7911750e-06,  3.8252897e-06],
        [-2.3829889e-06, -5.4837223e-08, -6.9595359e-07, ...,
          1.3674828e-06, -1.0461699e-06,  8.3080204e-06]], dtype=float32), indices=array([997
0, 9971, 9972, ..., 2043,   23,    1], dtype=int32), dense_shape=array([10000,   200], dtype=
int32)),
 array([[-1.8405798e-09, -3.1994631e-08,  2.3946001e-08, ...,
          1.3354203e-08, -2.3405757e-08, -4.1589594e-09],
        [-9.1852186e-08,  2.0004784e-08, -4.5353577e-08, ...,
          3.4887663e-08,  7.8337647e-09,  3.7139747e-09],
        [ 5.0924370e-08,  1.9195475e-08, -5.2159741e-08, ...,
         -3.2702914e-08,  7.4668605e-10, -1.8803103e-08],
        ...,
        [ 2.7028721e-09, -1.1007504e-09, -1.9411790e-09, ...,
         -1.1699623e-08,  1.0719061e-09,  2.8957781e-09],
        [-4.2527697e-09,  2.3546840e-09, -4.9742872e-09, ...,
         -2.0641178e-09,  2.8362528e-09, -2.4926088e-09],
        [ 5.0224935e-09, -5.5672800e-11, -1.8447400e-09, ...,
         -4.7100823e-10, -7.3696627e-10,  4.9655546e-09]], dtype=float32),
 array([-1.48190225e-08, -3.60064837e-06,  4.01541365e-06, ...,
         3.12504380e-06, -2.83452550e-06, -5.31231592e-07], dtype=float32),
 array([[ 1.7050820e-09, -3.1845979e-09, -4.6471778e-09, ...,
         -1.1726372e-08,  1.3059895e-09, -1.4103302e-08],
        [ 2.2695042e-08, -2.2669648e-09, -1.6026396e-09, ...,
          5.7543668e-09, -7.2102013e-09,  1.2096042e-09],
        [-6.2019643e-09,  4.4931161e-09, -1.6989468e-10, ...,
          4.6474473e-09,  7.4162121e-09, -3.9567039e-09],
        ...,
        [ 2.8151453e-10, -1.4020337e-09,  3.6845825e-11, ...,
          7.4824040e-09,  1.2538369e-09,  1.5315813e-09],
        [-2.7005107e-09,  2.9668457e-09,  1.1963031e-09, ...,
         -1.6570390e-09,  6.0684346e-10, -2.3887081e-09],
        [ 3.1488343e-09, -1.1875978e-09,  3.1008143e-10, ...,
         -1.3370233e-10,  2.2005258e-09,  5.5456533e-09]], dtype=float32),
 array([ 7.43911096e-06, -3.24106145e-06,  2.68317717e-06,  1.31919967e-06,
        -5.20503090e-06, -3.86957845e-06,  2.35273774e-06, -1.25092538e-06,
         1.71901377e-06,  4.38427287e-06,  2.31960712e-06,  2.29801094e-06,
        -7.37425125e-06, -6.00161400e-07,  3.73991929e-06,  4.68583858e-06,
        -2.87669877e-06,  1.02666272e-05,  2.90786238e-06, -1.06980406e-06,
         1.35618791e-06, -6.08193602e-07, -3.12980956e-06,  2.15545651e-06,
        -6.19616713e-08, -1.17665263e-06,  8.46665330e-07,  1.73706121e-06,
         5.22365190e-06, -7.24673441e-07, -3.09295842e-06,  4.10257144e-06,
        -8.91804859e-07, -1.65411632e-06, -3.80559345e-06, -1.37984921e-06,
        -2.51358256e-06, -4.25294076e-07, -7.78778656e-07,  2.61721516e-06,
        -3.29546037e-06, -1.79091546e-06,  2.49253304e-07,  3.62961600e-06,
         6.91698688e-06, -5.77540141e-06,  4.12573263e-06, -2.85256192e-07,
         1.66888447e-06,  3.10909172e-07, -1.59013496e-08,  7.36990216e-07,
        -3.00673651e-06,  1.01763953e-06, -4.61160744e-06,  1.96367409e-06,
        -1.77317844e-07, -4.59097737e-06,  1.20431878e-06,  2.17668003e-06,
         2.03560558e-06,  5.39294035e-07,  1.92743096e-06,  7.16975535e-07,
         7.25315385e-06, -7.66227004e-07, -7.14647854e-07,  2.37523432e-06,
         2.33094170e-06,  5.72697309e-06, -9.12741962e-06,  4.90171033e-07,
        -3.60989998e-06,  5.99487691e-07, -5.60587478e-06, -1.07131189e-06,
         1.05421141e-05,  4.93334392e-06, -2.31131207e-06,  1.48849097e-06,
        -2.92086020e-06,  6.12656606e-07, -3.20661252e-06, -2.58245291e-06,
         1.47602361e-06, -3.26423060e-06,  3.91438306e-07, -3.26919371e-06,
        -1.13363649e-06, -2.41446696e-06, -1.03426146e-05, -1.07886181e-05,
        -2.52007931e-06,  1.60326783e-06, -1.48260619e-06, -2.26519035e-07,
        -3.74991373e-07, -1.65603342e-06, -2.06724781e-06, -6.26060182e-06,
        -2.53121186e-07, -8.22982997e-07,  2.69592738e-06, -1.17796969e-06,
         1.60603076e-06, -2.54494097e-08,  6.20139554e-06,  3.02610232e-07,
         6.72309216e-06,  4.70306031e-06,  3.37450251e-06,  1.28835222e-06,
         2.35714225e-07, -1.53841540e-06,  1.22401150e-06,  2.08258325e-06,
        -4.49750951e-06, -5.62618334e-06,  4.95368613e-06,  1.49532480e-06,
        -9.31265913e-06,  3.14439774e-07, -9.29366024e-07, -2.72829857e-06,
        -9.14493967e-07,  4.03452805e-06,  3.70021303e-06,  7.21215088e-09,
```

-3.14392149e-02, -9.10963584e-03, -7.39855366e-03, 2.38143001e-03,
-2.78733037e-02, 1.94265880e-02, 1.50416791e-02, -1.75229530e-03,
 2.64715590e-02, -9.12279706e-04, 4.68383823e-03, 5.35370298e-02,
-1.43111423e-02, 1.96135622e-02, 1.75334024e-03, 3.83477584e-02,
-8.83468520e-03, -2.59343106e-02, 3.64173227e-03, 6.47144997e-03,
 7.64409918e-03, 2.33436413e-02, 7.47586554e-03, -2.14189552e-02,
 6.00448158e-03, 3.41934385e-03, -1.67969242e-02, 2.45301351e-02,
 1.15759000e-02, 9.64337680e-03, -2.92204297e-03, 5.94378635e-03,
-1.18295532e-02, -6.10278174e-03, 1.28217915e-03, 1.72044616e-03,
-6.76902104e-03, 3.39535065e-03, 8.71006958e-03, -3.08221322e-03,
 2.39534900e-02, -2.43782438e-02, -3.62582086e-03, 1.17021473e-02,
 1.34731142e-03, 1.05172256e-02, -1.59193482e-02, 1.18921865e-02,
 7.53655331e-03, 3.28934379e-02, 1.52126048e-02, -3.10551934e-02,
 4.91389120e-03, 9.11924522e-03, 2.50246637e-02, -2.94572543e-02,
 1.37154628e-02, -1.76510587e-02, 1.02501921e-02, -9.73187480e-03,
 6.76038535e-03, 5.20824827e-03, 2.58607278e-03, 1.56388190e-02,
 5.17142471e-03, 2.01467369e-02, 5.63172670e-03, -1.52983610e-02,
 7.87441526e-03, 3.30442712e-02, 2.23832484e-02, 1.69605426e-02,
-4.81336983e-03, -5.24539920e-03, 1.81866679e-02, -5.75028360e-03,
-2.00913642e-02, 8.64040293e-03, -2.41388176e-02, -1.29309972e-03,
 8.74661468e-03, 4.57775034e-03, 2.78850406e-04, 2.34826151e-02,
 1.12993401e-02, -1.04678993e-03, -3.87797202e-03, -6.83886837e-03,
-5.94671210e-03, 2.21266411e-03, -1.73163414e-02, -1.87826362e-02,
 2.41795601e-03, 2.04040930e-02, -1.16233826e-02, 5.40584419e-03,
-2.67027970e-02, -1.28349545e-03, 1.50024280e-05, -2.23553311e-02,
 1.83391944e-03, -9.08307824e-03, 2.51460616e-02, -2.76363809e-02,
-2.13584080e-02, 1.49537614e-02, 1.41843222e-02, 2.61052605e-03,
 1.63381901e-02, 2.72136182e-02, -1.17925797e-02, -4.49977769e-03,
-8.00076593e-03, 7.85257854e-03, 5.86397201e-03, -3.13691609e-02,
 2.54362356e-02, 5.56563074e-03, 1.59323830e-02, -1.63381658e-02,
 2.58750543e-02, 4.40080278e-03, 9.88699682e-03, 5.14265709e-03,
-6.28657872e-05, 2.01562475e-02, 1.50575740e-02, 1.51166311e-02,
 5.94640733e-06, -6.82590496e-07, 2.80427503e-06, 1.49979223e-06,
-5.50311279e-06, -3.72471845e-06, -7.01276349e-07, -1.15190198e-06,
 1.38643099e-06, 9.37325240e-07, 1.65022880e-06, 5.09915571e-06,
-7.77998684e-06, -4.08819869e-06, 2.43273848e-06, 9.90303533e-07,
-6.09470362e-07, 8.27403073e-06, 3.14168688e-06, 6.66154961e-07,
 3.00657530e-06, -6.45923080e-07, -1.53470751e-06, 3.25303654e-06,
 1.43036218e-07, -1.47128208e-06, -3.43832028e-07, 2.10360781e-06,
 5.26093709e-06, 1.29167205e-07, -2.56433827e-06, 2.62397498e-06,
 1.39421093e-06, -8.89711316e-07, -2.68016311e-06, -8.31823627e-07,
-2.06072264e-06, -1.46923855e-06, -1.92061884e-06, 1.99390320e-06,
-2.24589803e-06, -2.20186848e-06, 2.06880509e-06, 3.47852438e-06,
 3.01531713e-06, -3.76408343e-06, 2.64736877e-06, -3.30093076e-06,
 1.81108271e-06, 2.20167237e-07, -5.43364365e-07, 1.64575340e-06,
-7.60741784e-07, -5.25585733e-07, -4.11210522e-06, 8.75207775e-07,
-1.74005237e-08, -2.50262269e-06, -1.58797275e-06, 4.78889024e-06,
 3.94514882e-06, 2.95423251e-06, 2.56485578e-07, 2.25412373e-06,
 4.99646467e-06, 8.91065156e-07, 2.92760888e-06, 1.82211284e-06,
 3.11509694e-07, 5.86461192e-06, -3.95149482e-06, 3.13707392e-06,
-2.87708167e-06, -9.42764160e-08, -3.61694606e-06, -2.52908558e-06,
 1.02508620e-05, 2.17028696e-06, -2.27924284e-06, 8.46567843e-07,
-1.46987088e-06, -4.13462260e-08, -9.12419978e-07, -3.33771595e-06,
 5.88663056e-07, -9.57749762e-07, -7.17417834e-07, -1.55829196e-06,
-4.11318894e-07, -1.91968365e-06, -7.23132143e-06, -5.90137597e-06,
-4.03393869e-07, 3.72035811e-06, -7.47734077e-07, 5.39226789e-07,
 1.88116474e-06, -1.82829751e-06, -4.07900501e-07, -3.92738593e-06,
-1.14376712e-06, -3.82762977e-07, 3.41601799e-06, 1.73529963e-06,
 3.18591538e-06, -1.84331145e-06, 4.53867233e-06, -5.43115164e-07,
 5.86216947e-06, 6.10044890e-06, 2.43084946e-06, 6.59909063e-07,
 3.01188993e-06, -6.37133212e-07, 1.25398822e-06, 2.28062072e-06,
-2.71385329e-06, -1.35053074e-06, 1.96509109e-06, 5.94651510e-07,
-6.19892126e-06, -1.18855638e-07, 1.05221613e-06, -9.17227680e-07,
-1.12798659e-07, 5.40683413e-06, 2.07964604e-06, -1.72583407e-06,
 7.43371766e-06, -3.24104917e-06, 2.68018016e-06, 1.31542288e-06,
-5.21207858e-06, -3.87699811e-06, 2.35295101e-06, -1.25008455e-06,
 1.72180626e-06, 4.37849303e-06, 2.31894342e-06, 2.30359819e-06,
-7.37268192e-06, -5.92136416e-07, 3.74462047e-06, 4.68636108e-06,
-2.87370449e-06, 1.02776621e-05, 2.91140304e-06, -1.06941593e-06,
 1.34789991e-06, -6.05273897e-07, -3.13168061e-06, 2.15296427e-06,
-6.77641836e-08, -1.17652405e-06, 8.51674145e-07, 1.74643390e-06,
 5.22201572e-06, -7.19267518e-07, -3.09157394e-06, 4.09995073e-06,
-8.90188687e-07, -1.65731490e-06, -3.80199890e-06, -1.38107293e-06,
-2.50723770e-06, -4.31207297e-07, -7.77500702e-07, 2.61989771e-06,
-3.30696116e-06, -1.78935670e-06, 2.48968377e-07, 3.62953188e-06,

```
         6.91728201e-06, -5.77339688e-06,  4.13245698e-06, -2.87809002e-07,
         1.66709367e-06,  3.09546863e-07, -1.39195890e-08,  7.38480480e-07,
        -3.00843271e-06,  1.01179569e-06, -4.61403079e-06,  1.96547444e-06,
        -1.79495260e-07, -4.58724071e-06,  1.20792834e-06,  2.17566298e-06,
         2.03149148e-06,  5.40567385e-07,  1.92684024e-06,  7.14254725e-07,
         7.25170048e-06, -7.69282508e-07, -7.12372128e-07,  2.37492850e-06,
         2.33443689e-06,  5.71866485e-06, -9.12271571e-06,  4.91158517e-07,
        -3.60836657e-06,  5.97442295e-07, -5.60640046e-06, -1.07388291e-06,
         1.05424479e-05,  4.92863683e-06, -2.30141495e-06,  1.49322284e-06,
        -2.92214577e-06,  6.11597443e-07, -3.20553590e-06, -2.58823820e-06,
         1.47800245e-06, -3.26483064e-06,  3.95305165e-07, -3.26755548e-06,
        -1.13625458e-06, -2.41981343e-06, -1.03425973e-05, -1.07671294e-05,
        -2.52141876e-06,  1.59884621e-06, -1.48668744e-06, -2.23500606e-07,
        -3.74041349e-07, -1.65170070e-06, -2.06345067e-06, -6.25837993e-06,
        -2.55892331e-07, -8.17402281e-07,  2.69009070e-06, -1.17690695e-06,
         1.60710636e-06, -1.90682385e-08,  6.20065066e-06,  3.04107374e-07,
         6.71916587e-06,  4.70324358e-06,  3.37128563e-06,  1.28660270e-06,
         2.35376945e-07, -1.53968449e-06,  1.22244046e-06,  2.09136761e-06,
        -4.50214293e-06, -5.62112700e-06,  4.95244194e-06,  1.49418224e-06,
        -9.30930219e-06,  3.17065172e-07, -9.29828502e-07, -2.72897341e-06,
        -9.13749773e-07,  4.03737340e-06,  3.70067391e-06,  5.98555516e-09],
       dtype=float32),
 array([[ 1.73404114e-04,  2.58801592e-04,  1.02787730e-04, ...,
         -3.70277547e-07, -3.77359129e-07, -3.74582243e-07],
        [-1.04275634e-04, -1.81319352e-04,  7.19394448e-05, ...,
          2.54231225e-07,  2.59042480e-07,  2.57163407e-07],
        [-9.09706869e-05,  4.29514112e-05, -1.03729981e-04, ...,
         -7.80411469e-08, -7.95605644e-08, -7.89469183e-08],
        ...,
        [-2.13911862e-05, -3.66407796e-04, -5.05774951e-05, ...,
          2.66888037e-07,  2.71990672e-07,  2.69988192e-07],
        [ 4.79268674e-05,  9.09635273e-05,  4.20610522e-05, ...,
         -2.15053262e-07, -2.19167035e-07, -2.17581373e-07],
        [-8.50783108e-05, -1.02979975e-04,  7.47508966e-06, ...,
          1.58913195e-07,  1.61989092e-07,  1.60784253e-07]], dtype=float32),
 array([-0.78131574, -1.0979856 , -0.9813465 , ...,  0.00198156,
         0.00201933,  0.00200458], dtype=float32)]
```

4. Apply the optimizer to the variables / gradients tuple.

In [44]:

```
# Create the training TensorFlow Operation through our optimizer
train_op = optimizer.apply_gradients(zip(grads, tvars))
```

In [45]:

```
session.run(tf.global_variables_initializer())
session.run(train_op, feed_dict)
```

LSTM

We learned how the model is build step by step. Noe, let's then create a Class that represents our model. This class needs a few things:

- We have to create the model in accordance with our defined hyperparameters
- We have to create the placeholders for our input data and expected outputs (the real data)
- We have to create the LSTM cell structure and connect them with our RNN structure
- We have to create the word embeddings and point them to the input data
- We have to create the input structure for our RNN
- We have to instantiate our RNN model and retrieve the variable in which we should expect our outputs to appear
- We need to create a logistic structure to return the probability of our words
- We need to create the loss and cost functions for our optimizer to work, and then create the optimizer
- And finally, we need to create a training operation that can be run to actually train our model

In [46]:

```
hidden_size_l1
```

Out[46]:

256

In [47]:

```
class PTBModel(object):

    def __init__(self, action_type):
        ###################################
        # Setting parameters for ease of use #
        ###################################
        self.batch_size = batch_size
        self.num_steps = num_steps
        self.hidden_size_l1 = hidden_size_l1
        self.hidden_size_l2 = hidden_size_l2
        self.vocab_size = vocab_size
        self.embeding_vector_size = embeding_vector_size
        ###############################################################################
        # Creating placeholders for our input data and expected outputs (target data) #
        ###############################################################################
        self._input_data = tf.placeholder(tf.int32, [batch_size, num_steps]) #[30#20]
        self._targets = tf.placeholder(tf.int32, [batch_size, num_steps]) #[30#20]

        ######################################################################
        # Creating the LSTM cell structure and connect it with the RNN structure #
        ######################################################################
        # Create the LSTM unit.
        # This creates only the structure for the LSTM and has to be associated with a RNN unit still.
        # The argument n_hidden(size=200) of BasicLSTMCell is size of hidden layer, that is, the number
of hidden units of the LSTM (inside A).
        # Size is the same as the size of our hidden layer, and no bias is added to the Forget Gate.
        # LSTM cell processes one word at a time and computes probabilities of the possible continuatio
ns of the sentence.
        lstm_cell_l1 = tf.contrib.rnn.BasicLSTMCell(self.hidden_size_l1, forget_bias=0.0)
        lstm_cell_l2 = tf.contrib.rnn.BasicLSTMCell(self.hidden_size_l2, forget_bias=0.0)

        # Unless you changed keep_prob, this won't actually execute -- this is a dropout wrapper for ou
r LSTM unit
        # This is an optimization of the LSTM output, but is not needed at all
        if action_type == "is_training" and keep_prob < 1:
            lstm_cell_l1 = tf.contrib.rnn.DropoutWrapper(lstm_cell_l1, output_keep_prob=keep_prob)
            lstm_cell_l2 = tf.contrib.rnn.DropoutWrapper(lstm_cell_l2, output_keep_prob=keep_prob)

        # By taking in the LSTM cells as parameters, the MultiRNNCell function junctions the LSTM units
to the RNN units.
        # RNN cell composed sequentially of multiple simple cells.
        stacked_lstm = tf.contrib.rnn.MultiRNNCell([lstm_cell_l1, lstm_cell_l2])

        # Define the initial state, i.e., the model state for the very first data point
        # It initialize the state of the LSTM memory. The memory state of the network is initialized wi
th a vector of zeros and gets updated after reading each word.
        self._initial_state = stacked_lstm.zero_state(batch_size, tf.float32)

        ######################################################################
        # Creating the word embeddings and pointing them to the input data #
        ######################################################################
        with tf.device("/cpu:0"):
            # Create the embeddings for our input data. Size is hidden size.
            embedding = tf.get_variable("embedding", [vocab_size, self.embeding_vector_size])  #[10000x
200]
            # Define where to get the data for our embeddings from
            inputs = tf.nn.embedding_lookup(embedding, self._input_data)

        # Unless you changed keep_prob, this won't actually execute -- this is a dropout addition for o
ur inputs
        # This is an optimization of the input processing and is not needed at all
        if action_type == "is_training" and keep_prob < 1:
            inputs = tf.nn.dropout(inputs, keep_prob)

        ##############################################
        # Creating the input structure for our RNN #
        ##############################################
        # Input structure is 20x[30x200]
        # Considering each word is represended by a 200 dimentional vector, and we have 30 batchs, we c
reate 30 word-vectors of size [30xx2000]
        # inputs = [tf.squeeze(input_, [1]) for input_ in tf.split(1, num_steps, inputs)]
        # The input structure is fed from the embeddings, which are filled in by the input data
        # Feeding a batch of b sentences to a RNN:
        # In step 1,  first word of each of the b sentences (in a batch) is input in parallel.
        # In step 2,  second word of each of the b sentences is input in parallel.
```

```
        # The parallelism is only for efficiency.
        # Each sentence in a batch is handled in parallel, but the network sees one word of a sentence
 at a time and does the computations accordingly.
        # All the computations involving the words of all sentences in a batch at a given time step are
done in parallel.

        ################################################################################################
#####
        # Instantiating our RNN model and retrieving the structure for returning the outputs and the st
ate #
        ################################################################################################
#####

        outputs, state = tf.nn.dynamic_rnn(stacked_lstm, inputs, initial_state=self._initial_state)
        ########################################################################
        # Creating a logistic unit to return the probability of the output word #
        ########################################################################
        output = tf.reshape(outputs, [-1, self.hidden_size_l2])
        softmax_w = tf.get_variable("softmax_w", [self.hidden_size_l2, vocab_size]) #[200x1000]
        softmax_b = tf.get_variable("softmax_b", [vocab_size]) #[1x1000]
        logits = tf.matmul(output, softmax_w) + softmax_b
        logits = tf.reshape(logits, [self.batch_size, self.num_steps, vocab_size])
        prob = tf.nn.softmax(logits)
        out_words = tf.argmax(prob, axis=2)
        self._output_words = out_words
        ########################################################################
        # Defining the loss and cost functions for the model's learning to work #
        ########################################################################


        # Use the contrib sequence loss and average over the batches
        loss = tf.contrib.seq2seq.sequence_loss(
            logits,
            self.targets,
            tf.ones([batch_size, num_steps], dtype=tf.float32),
            average_across_timesteps=False,
            average_across_batch=True)

#         loss = tf.contrib.legacy_seq2seq.sequence_loss_by_example([logits], [tf.reshape(self._target
s, [-1])],
#                                                                   [tf.ones([batch_size * num_steps])])
        self._cost = tf.reduce_sum(loss)

        # Store the final state
        self._final_state = state

        #Everything after this point is relevant only for training
        if action_type != "is_training":
            return

        ################################################
        # Creating the Training Operation for our Model #
        ################################################
        # Create a variable for the learning rate
        self._lr = tf.Variable(0.0, trainable=False)
        # Get all TensorFlow variables marked as "trainable" (i.e. all of them except _lr, which we jus
t created)
        tvars = tf.trainable_variables()
        # Define the gradient clipping threshold
        grads, _ = tf.clip_by_global_norm(tf.gradients(self._cost, tvars), max_grad_norm)
        # Create the gradient descent optimizer with our learning rate
        optimizer = tf.train.GradientDescentOptimizer(self.lr)
        # Create the training TensorFlow Operation through our optimizer
        self._train_op = optimizer.apply_gradients(zip(grads, tvars))

    # Helper functions for our LSTM RNN class

    # Assign the learning rate for this model
    def assign_lr(self, session, lr_value):
        session.run(tf.assign(self.lr, lr_value))

    # Returns the input data for this model at a point in time
    @property
    def input_data(self):
        return self._input_data
```

```
    # Returns the targets for this model at a point in time
    @property
    def targets(self):
        return self._targets

    # Returns the initial state for this model
    @property
    def initial_state(self):
        return self._initial_state

    # Returns the defined Cost
    @property
    def cost(self):
        return self._cost

    # Returns the final state for this model
    @property
    def final_state(self):
        return self._final_state

    # Returns the final output words for this model
    @property
    def final_output_words(self):
        return self._output_words

    # Returns the current learning rate for this model
    @property
    def lr(self):
        return self._lr

    # Returns the training operation defined for this model
    @property
    def train_op(self):
        return self._train_op
```

With that, the actual structure of our Recurrent Neural Network with Long Short-Term Memory is finished. What remains for us to do is to actually create the methods to run through time -- that is, the `run_epoch` method to be run at each epoch and a `main` script which ties all of this together.

What our `run_epoch` method should do is take our input data and feed it to the relevant operations. This will return at the very least the current result for the cost function.

In [48]:

```
##############################################################################################################
##################
# run_one_epoch takes as parameters the current session, the model instance, the data to be fed, and th
e operation to be run #
##############################################################################################################
##################
def run_one_epoch(session, m, data, eval_op, verbose=False):

    #Define the epoch size based on the length of the data, batch size and the number of steps
    epoch_size = ((len(data) // m.batch_size) - 1) // m.num_steps
    start_time = time.time()
    costs = 0.0
    iters = 0

    state = session.run(m.initial_state)

    #For each step and data point
    for step, (x, y) in enumerate(reader.ptb_iterator(data, m.batch_size, m.num_steps)):

        #Evaluate and return cost, state by running cost, final_state and the function passed as parame
ter
        cost, state, out_words, _ = session.run([m.cost, m.final_state, m.final_output_words, eval_op],
                                     {m.input_data: x,
                                      m.targets: y,
                                      m.initial_state: state})

        #Add returned cost to costs (which keeps track of the total costs for this epoch)
        costs += cost

        #Add number of steps to iteration counter
        iters += m.num_steps

        if verbose and step % (epoch_size // 10) == 10:
            print("Itr %d of %d, perplexity: %.3f speed: %.0f wps" % (step , epoch_size, np.exp(costs /
iters), iters * m.batch_size / (time.time() - start_time)))

    # Returns the Perplexity rating for us to keep track of how the model is evolving
    return np.exp(costs / iters)
```

Now, we create the main method to tie everything together. The code here reads the data from the directory, using the reader helper module, and then trains and evaluates the model on both a testing and a validating subset of data.

In [49]:

```
# Reads the data and separates it into training data, validation data and testing data
raw_data = reader.ptb_raw_data(data_dir)
train_data, valid_data, test_data, _, _ = raw_data
```

In [ ]:

```
# Initializes the Execution Graph and the Session
with tf.Graph().as_default(), tf.Session() as session:
    initializer = tf.random_uniform_initializer(-init_scale, init_scale)

    # Instantiates the model for training
    # tf.variable_scope add a prefix to the variables created with tf.get_variable
    with tf.variable_scope("model", reuse=None, initializer=initializer):
        m = PTBModel("is_training")

    # Reuses the trained parameters for the validation and testing models
    # They are different instances but use the same variables for weights and biases, they just don't c
hange when data is input
    with tf.variable_scope("model", reuse=True, initializer=initializer):
        mvalid = PTBModel("is_validating")
        mtest = PTBModel("is_testing")

    #Initialize all variables
    tf.global_variables_initializer().run()

    for i in range(max_epoch):
        # Define the decay for this epoch
        lr_decay = decay ** max(i - max_epoch_decay_lr, 0.0)

        # Set the decayed learning rate as the learning rate for this epoch
        m.assign_lr(session, learning_rate * lr_decay)

        print("Epoch %d : Learning rate: %.3f" % (i + 1, session.run(m.lr)))

        # Run the loop for this epoch in the training model
        train_perplexity = run_one_epoch(session, m, train_data, m.train_op, verbose=True)
        print("Epoch %d : Train Perplexity: %.3f" % (i + 1, train_perplexity))

        # Run the loop for this epoch in the validation model
        valid_perplexity = run_one_epoch(session, mvalid, valid_data, tf.no_op())
        print("Epoch %d : Valid Perplexity: %.3f" % (i + 1, valid_perplexity))

    # Run the loop in the testing model to see how effective was our training
    test_perplexity = run_one_epoch(session, mtest, test_data, tf.no_op())

    print("Test Perplexity: %.3f" % test_perplexity)
```

```
Epoch 1 : Learning rate: 1.000
Itr 10 of 774, perplexity: 4121.075 speed: 257 wps
Itr 87 of 774, perplexity: 1265.477 speed: 262 wps
Itr 164 of 774, perplexity: 976.483 speed: 261 wps
Itr 241 of 774, perplexity: 815.197 speed: 262 wps
Itr 318 of 774, perplexity: 721.811 speed: 260 wps
Itr 395 of 774, perplexity: 646.499 speed: 249 wps
Itr 472 of 774, perplexity: 586.276 speed: 230 wps
Itr 549 of 774, perplexity: 532.302 speed: 217 wps
Itr 626 of 774, perplexity: 488.858 speed: 213 wps
Itr 703 of 774, perplexity: 454.560 speed: 211 wps
Epoch 1 : Train Perplexity: 430.328
Epoch 1 : Valid Perplexity: 233.651
Epoch 2 : Learning rate: 1.000
Itr 10 of 774, perplexity: 272.895 speed: 191 wps
Itr 87 of 774, perplexity: 237.084 speed: 187 wps
```

As you can see, the model's perplexity rating drops very quickly after a few iterations. As was elaborated before, **lower Perplexity means that the model is more certain about its prediction.** As such, we can be sure that this model is performing well!

This is the end of the **Applying Recurrent Neural Networks to Text Processing** notebook. Hopefully you now have a better understanding of Recurrent Neural Networks and how to implement one utilizing TensorFlow. Thank you for reading this notebook, and good luck on your studies.

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 (https://cocl.us/ML0120EN_PAI).

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 (https://cocl.us/ML0120EN_DSX).This is the end of this lesson. Thank you for reading this notebook, and good luck on your studies.

Thanks for completing this lesson!

Notebook created by Walter Gomes de Amorim Junior (https://br.linkedin.com/in/walter-gomes-de-amorim-junior-624726121), Saeed Aghabozorgi (https://linkedin.com/in/saeedaghabozorgi)</h4>
