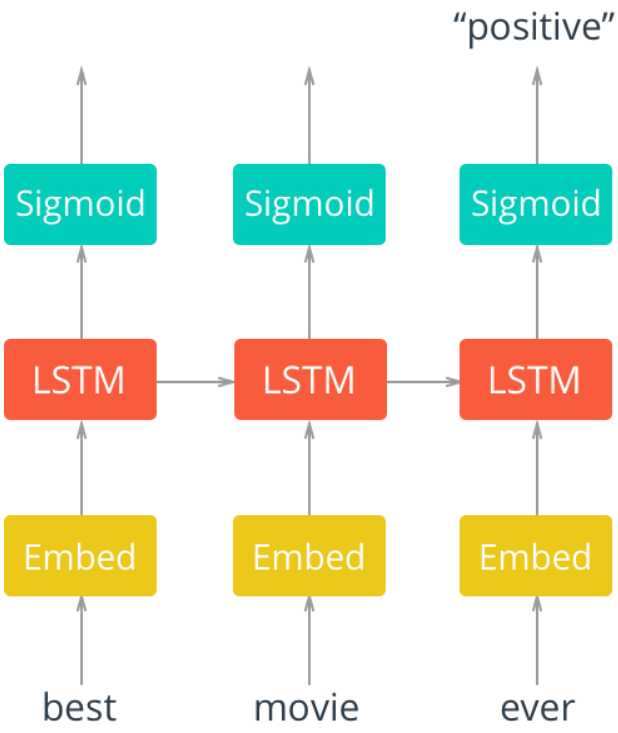
Sentiment Analysis with an RNN

In this notebook, you'll implement a recurrent neural network that performs sentiment analysis. Using an RNN rather than a feedfoward network is more accurate since we can include information about the *sequence* of words. Here we'll use a dataset of movie reviews, accompanied by labels.

The architecture for this network is shown below.



Here, we'll pass in words to an embedding layer. We need an embedding layer because we have tens of thousands of words, so we'll need a more efficient representation for our input data than one-hot encoded vectors. You should have seen this before from the word2vec lesson. You can actually train up an embedding with word2vec and use it here. But it's good enough to just have an embedding layer and let the network learn the embedding table on it's own.

From the embedding layer, the new representations will be passed to LSTM cells. These will add recurrent connections to the network so we can include information about the sequence of words in the data. Finally, the LSTM cells will go to a sigmoid output layer here. We're using the sigmoid because we're trying to predict if this text has positive or negative sentiment. The output layer will just be a single unit then, with a sigmoid activation function.

We don't care about the sigmoid outputs except for the very last one, we can ignore the rest. We'll calculate the cost from the output of the last step and the training label.

In [29]:

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

In [30]:

```
with open('../sentiment-network/reviews.txt', 'r') as f:
    reviews = f.read()
with open('../sentiment-network/labels.txt', 'r') as f:
    labels = f.read()
```

In [31]:

```
reviews[:2000]
```

Out[31]:

```
'bromwell high is a cartoon comedy . it ran at the same time as some other
programs about school life  such as  teachers  . my  years in the teaching
profession lead me to believe that bromwell high  s satire is much closer to
……
years but never a plan to help those on the street that were once considered
human who did everything from going to school  work  or vote for the matter .
most people think of the homeless as just a lost cause while worrying about
things such as racism  the war on iraq  pressuring kids to succeed  technology
the elections  inflation  or worrying if they  ll be next to end up on the
streets .  br  br  but what if y'
```

Data preprocessing

The first step when building a neural network model is getting your data into the proper form to feed into the network. Since we're using embedding layers, we'll need to encode each word with an integer. We'll also want to clean it up a bit.

You can see an example of the reviews data above. We'll want to get rid of those periods. Also, you might notice that the reviews are delimited with newlines `\n`. To deal with those, I'm going to split the text into each review using `\n` as the delimiter. Then I can combined all the reviews back together into one big string.

First, let's remove all punctuation. Then get all the text without the newlines and split it into individual words.

In [32]:

```
from string import punctuation
all_text = ''.join([c for c in reviews if c not in punctuation])
reviews = all_text.split('\n')

all_text = ' '.join(reviews)
words = all_text.split()
```

In [33]:

```
all_text[:2000]
```

Out[33]:

```
'bromwell high is a cartoon comedy  it ran at the same time as some other
programs about school life  such as  teachers   my   years in the teaching
profession lead me to believe that bromwell high  s satire is much closer to
……
of the homeless as just a lost cause while worrying about things such as racism
the war on iraq  pressuring kids to succeed  technology  the elections
inflation  or worrying if they  ll be next to end up on the streets   br    br
but what if you were given a bet to live on the st'
```

In [34]:

```
words[:100]
```

Out[34]:

```
['bromwell',
 'high',
 'is',
 'a',
 'cartoon',
……
'immediately',
 'recalled',
 'at',
 'high']
```

Encoding the words

The embedding lookup requires that we pass in integers to our network. The easiest way to do this is to create dictionaries that map the words in the vocabulary to integers. Then we can convert each of our reviews into integers so they can be passed into the network.
Exercise: Now you're going to encode the words with integers. Build a dictionary that maps words to integers. Later we're going to pad our input vectors with zeros, so make sure the integers **start at 1, not 0**. Also, convert the reviews to integers and store the reviews in a new list called `reviews_ints`.

In [35]:
```
from collections import Counter
counts = Counter(words)
vocab = sorted(counts, key=counts.get, reverse=True)
vocab_to_int = {word: ii for ii, word in enumerate(vocab, 1)}

reviews_ints = []
for each in reviews:
    reviews_ints.append([vocab_to_int[word] for word in each.split()])
```

Encoding the labels

Our labels are "positive" or "negative". To use these labels in our network, we need to convert them to 0 and 1.
Exercise: Convert labels from `positive` and `negative` to 1 and 0, respectively.

In [36]:
```
labels = labels.split('\n')
labels = np.array([1 if each == 'positive' else 0 for each in labels])
```

In [37]:
```
review_lens = Counter([len(x) for x in reviews_ints])
print("Zero-length reviews: {}".format(review_lens[0]))
print("Maximum review length: {}".format(max(review_lens)))
```

```
Zero-length reviews: 1
Maximum review length: 2514
```

Okay, a couple issues here. We seem to have one review with zero length. And, the maximum review length is way too many steps for our RNN. Let's truncate to 200 steps. For reviews shorter than 200, we'll pad with 0s. For reviews longer than 200, we can truncate them to the first 200 characters.
Exercise: First, remove the review with zero length from the `reviews_ints` list.

In [38]:
```
non_zero_idx = [ii for ii, review in enumerate(reviews_ints) if len(review) != 0]
len(non_zero_idx)
```

Out[38]:
```
25000
```

In [41]:
```
reviews_ints[-1]
```

Out[41]:
```
[]
```

Turns out its the final review that has zero length. But that might not always be the case, so let's make it more general.

In [42]:
```
reviews_ints = [reviews_ints[ii] for ii in non_zero_idx]
labels = np.array([labels[ii] for ii in non_zero_idx])
```

Exercise: Now, create an array `features` that contains the data we'll pass to the network. The data should come from `review_ints`, since we want to feed integers to the network. Each row should be 200 elements long. For reviews shorter than 200 words, left pad with 0s. That is, if the review is `['best', 'movie', 'ever']`, `[117, 18, 128]` as integers, the row will look like `[0, 0, 0, ..., 0, 117, 18, 128]`. For reviews longer than 200, use on the first 200 words as the feature vector.

This isn't trivial and there are a bunch of ways to do this. But, if you're going to be building your own deep learning networks, you're going to have to get used to preparing your data.

In [46]:

```
seq_len = 200
features = np.zeros((len(reviews_ints), seq_len), dtype=int)
for i, row in enumerate(reviews_ints):
    features[i, -len(row):] = np.array(row)[:seq_len]
```

In [47]:

```
features[:10,:100]
```

Out[47]:

```
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,
            0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0, 21025,   308,     6,
            3,  1050,   207,     8,  2138,    32,     1,   171,    57,
           15,    49,    81,  5785,    44,   382,   110,   140,    15,
         5194,    60,   154,     9,     1,  4975,  5852,   475,    71,
            5,   260,    12, 21025,   308,    13,  1978,     6,    74,
         2395],
.....
[   54,    10,    14,   116,    60,   798,   552,    71,   364,
            5,     1,   730,     5,    66,  8057,     8,    14,    30,
            4,   109,    99,    10,   293,    17,    60,   798,    19,
           11,    14,     1,    64,    30,    69,  2500,    45,     4,
          234,    93,    10,    68,   114,   108,  8057,   363,    43,
         1009,     2,    10,    97,    28,  1431,    45,     1,   357,
            4,    60,   110,   205,     8,    48,     3,  1929, 10880,
            2,  2124,   354,   412,     4,    13,  6609,     2,  2974,
         5148,  2125,  1366,     6,    30,     4,    60,   502,   876,
           19,  8057,     6,    34,   227,     1,   247,   412,     4,
          582,     4,    27,   599,     9,     1, 13586,   396,     4,
        14047]])
```

Training, Validation, Test

With our data in nice shape, we'll split it into training, validation, and test sets.

> **Exercise:** Create the training, validation, and test sets here. You'll need to create sets for the features and the labels, `train_x` and `train_y` for example. Define a split fraction, `split_frac` as the fraction of data to keep in the training set. Usually this is set to 0.8 or 0.9. The rest of the data will be split in half to create the validation and testing data.

In [48]:

```
split_frac = 0.8
split_idx = int(len(features)*0.8)
train_x, val_x = features[:split_idx], features[split_idx:]
train_y, val_y = labels[:split_idx], labels[split_idx:]

test_idx = int(len(val_x)*0.5)
val_x, test_x = val_x[:test_idx], val_x[test_idx:]
val_y, test_y = val_y[:test_idx], val_y[test_idx:]

print("\t\t\tFeature Shapes:")
print("Train set: \t\t{}".format(train_x.shape),
    "\nValidation set: \t{}".format(val_x.shape),
    "\nTest set: \t\t{}".format(test_x.shape))
```

```
                      Feature Shapes:
Train set:            (20000, 200)
Validation set:       (2500, 200)
Test set:             (2500, 200)
```

With train, validation, and text fractions of 0.8, 0.1, 0.1, the final shapes should look like:

```
                      Feature Shapes:
    Train set:        (20000, 200)
    Validation set:   (2500, 200)
    Test set:         (2500, 200)
```

Build the graph

Here, we'll build the graph. First up, defining the hyperparameters.

- `lstm_size`: Number of units in the hidden layers in the LSTM cells. Usually larger is better performance wise. Common values are 128, 256, 512, etc.
- `lstm_layers`: Number of LSTM layers in the network. I'd start with 1, then add more if I'm underfitting.
- `batch_size`: The number of reviews to feed the network in one training pass. Typically this should be set as high as you can go without running out of memory.
- `learning_rate`: Learning rate

In [31]:

```
lstm_size = 256
lstm_layers = 1
batch_size = 500
learning_rate = 0.001
```

For the network itself, we'll be passing in our 200 element long review vectors. Each batch will be `batch_size` vectors. We'll also be using dropout on the LSTM layer, so we'll make a placeholder for the keep probability.

Exercise: Create the `inputs_`, `labels_`, and drop out `keep_prob` placeholders using `tf.placeholder`. `labels_` needs to be two-dimensional to work with some functions later. Since `keep_prob` is a scalar (a 0-dimensional tensor), you shouldn't provide a size to `tf.placeholder`.

In [32]:

```
n_words = len(vocab_to_int)

# Create the graph object
graph = tf.Graph()
# Add nodes to the graph
with graph.as_default():
    inputs_ = tf.placeholder(tf.int32, [None, None], name='inputs')
    labels_ = tf.placeholder(tf.int32, [None, None], name='labels')
    keep_prob = tf.placeholder(tf.float32, name='keep_prob')
```

Embedding

Now we'll add an embedding layer. We need to do this because there are 74000 words in our vocabulary. It is massively inefficient to one-hot encode our classes here. You should remember dealing with this problem from the word2vec lesson. Instead of one-hot encoding, we can have an embedding layer and use that layer as a lookup table. You could train an embedding layer using word2vec, then load it here. But, it's fine to just make a new layer and let the network learn the weights.

Exercise: Create the embedding lookup matrix as a `tf.Variable`. Use that embedding matrix to get the embedded vectors to pass to the LSTM cell with `tf.nn.embedding_lookup`. This function takes the embedding matrix and an input tensor, such as the review vectors. Then, it'll return another tensor with the embedded vectors. So, if the embedding layer as 200 units, the function will return a tensor with size [batch_size, 200].

In [33]:
```
# Size of the embedding vectors (number of units in the embedding layer)
embed_size = 300

with graph.as_default():
    embedding = tf.Variable(tf.random_uniform((n_words, embed_size), -1, 1))
    embed = tf.nn.embedding_lookup(embedding, inputs_)
```

LSTM cell

Next, we'll create our LSTM cells to use in the recurrent network (TensorFlow documentation). Here we are just defining what the cells look like. This isn't actually building the graph, just defining the type of cells we want in our graph.

To create a basic LSTM cell for the graph, you'll want to use `tf.contrib.rnn.BasicLSTMCell`. Looking at the function documentation:

```
tf.contrib.rnn.BasicLSTMCell(num_units, forget_bias=1.0,
input_size=None, state_is_tuple=True, activation=<function tanh at
0x109f1ef28>)
```

you can see it takes a parameter called `num_units`, the number of units in the cell, called `lstm_size` in this code. So then, you can write something like

```
lstm = tf.contrib.rnn.BasicLSTMCell(num_units)
```

to create an LSTM cell with `num_units`. Next, you can add dropout to the cell with `tf.contrib.rnn.DropoutWrapper`. This just wraps the cell in another cell, but with dropout added to the inputs and/or outputs. It's a really convenient way to make your network better with almost no effort! So you'd do something like

```
drop = tf.contrib.rnn.DropoutWrapper(cell, output_keep_prob=keep_prob)
```

Most of the time, you're network will have better performance with more layers. That's sort of the magic of deep learning, adding more layers allows the network to learn really complex relationships. Again, there is a simple way to create multiple layers of LSTM cells with `tf.contrib.rnn.MultiRNNCell`:

```
cell = tf.contrib.rnn.MultiRNNCell([drop] * lstm_layers)
```

Here, `[drop] * lstm_layers` creates a list of cells (`drop`) that is `lstm_layers` long.
The `MultiRNNCell` wrapper builds this into multiple layers of RNN cells, one for each cell in the list.

So the final cell you're using in the network is actually multiple (or just one) LSTM cells with dropout. But it all works the same from an achitectural viewpoint, just a more complicated graph in the cell.
Exercise: Below, use `tf.contrib.rnn.BasicLSTMCell` to create an LSTM cell. Then, add drop out to it with `tf.contrib.rnn.DropoutWrapper`. Finally, create multiple LSTM layers with `tf.contrib.rnn.MultiRNNCell`.

Here is a tutorial on building RNNs that will help you out.

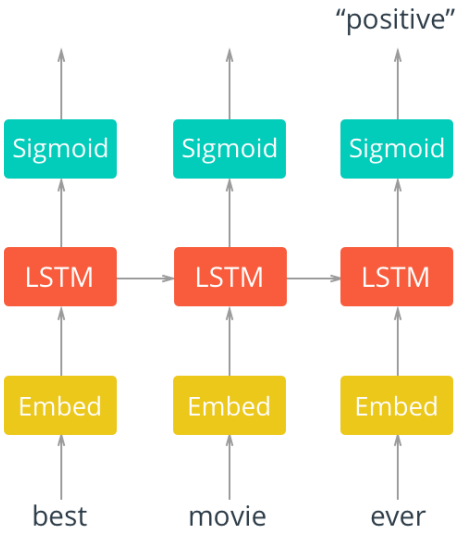
In [34]:
```
with graph.as_default():
    # Your basic LSTM cell
    lstm = tf.contrib.rnn.BasicLSTMCell(lstm_size)

    # Add dropout to the cell
    drop = tf.contrib.rnn.DropoutWrapper(lstm, output_keep_prob=keep_prob)
```

```
    # Stack up multiple LSTM layers, for deep learning
    cell = tf.contrib.rnn.MultiRNNCell([drop] * lstm_layers)

    # Getting an initial state of all zeros
    initial_state = cell.zero_state(batch_size, tf.float32)
```

RNN forward pass



Now we need to actually run the data through the RNN nodes. You can use `tf.nn.dynamic_rnn` to do this. You'd pass in the RNN cell you created (our multiple layered LSTM `cell` for instance), and the inputs to the network.

```
outputs, final_state = tf.nn.dynamic_rnn(cell,
inputs, initial_state=initial_state)
```

Above I created an initial state, `initial_state`, to pass to the RNN. This is the cell state that is passed between the hidden layers in successive time steps. `tf.nn.dynamic_rnn` takes care of most of the work for us. We pass in our cell and the input to the cell, then it does the unrolling and everything else for us. It returns outputs for each time step and the final_state of the hidden layer.
Exercise: Use `tf.nn.dynamic_rnn` to add the forward pass through the RNN. Remember that we're actually passing in vectors from the embedding layer, `embed`.

In [35]:

```
with graph.as_default():
    outputs, final_state = tf.nn.dynamic_rnn(cell, embed,
                                             initial_state=initial_state)
```

Output

We only care about the final output, we'll be using that as our sentiment prediction. So we need to grab the last output with `outputs[:, -1]`, the calculate the cost from that and `labels_`.

In [36]:

```
with graph.as_default():
    predictions = tf.contrib.layers.fully_connected(outputs[:, -1], 1,
activation_fn=tf.sigmoid)
    cost = tf.losses.mean_squared_error(labels_, predictions)

    optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)
```

Validation accuracy

Here we can add a few nodes to calculate the accuracy which we'll use in the validation pass.

In [37]:

```
with graph.as_default():
    correct_pred = tf.equal(tf.cast(tf.round(predictions), tf.int32), labels_)
    accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

Batching

This is a simple function for returning batches from our data. First it removes data such that we only have full batches. Then it iterates through the `x` and `y` arrays and returns slices out of those arrays with size `[batch_size]`.

In [38]:

```
def get_batches(x, y, batch_size=100):

    n_batches = len(x)//batch_size
    x, y = x[:n_batches*batch_size], y[:n_batches*batch_size]
    for ii in range(0, len(x), batch_size):
        yield x[ii:ii+batch_size], y[ii:ii+batch_size]
```

Training

Below is the typical training code. If you want to do this yourself, feel free to delete all this code and implement it yourself. Before you run this, make sure the `checkpoints` directory exists.

In [43]:

```
epochs = 10

with graph.as_default():
    saver = tf.train.Saver()

with tf.Session(graph=graph) as sess:
    sess.run(tf.global_variables_initializer())
    iteration = 1
    for e in range(epochs):
        state = sess.run(initial_state)

        for ii, (x, y) in enumerate(get_batches(train_x,train_y,batch_size),1):
            feed = {inputs_: x,
                    labels_: y[:, None],
                    keep_prob: 0.5,
                    initial_state: state}
            loss, state, _ = sess.run([cost, final_state, optimizer],
feed_dict=feed)

            if iteration%5==0:
                print("Epoch: {}/{}".format(e, epochs),
                      "Iteration: {}".format(iteration),
                      "Train loss: {:.3f}".format(loss))

            if iteration%25==0:
                val_acc = []
                val_state = sess.run(cell.zero_state(batch_size, tf.float32))
                for x, y in get_batches(val_x, val_y, batch_size):
                    feed = {inputs_: x,
                            labels_: y[:, None],
                            keep_prob: 1,
                            initial_state: val_state}
```

```
                    batch_acc, val_state = sess.run([accuracy, final_state],
feed_dict=feed)
                    val_acc.append(batch_acc)
                print("Val acc: {:.3f}".format(np.mean(val_acc)))
            iteration +=1
    saver.save(sess, "checkpoints/sentiment.ckpt")
```

```
Epoch: 0/10 Iteration: 5 Train loss: 0.244
Epoch: 0/10 Iteration: 10 Train loss: 0.237
……
Epoch: 9/10 Iteration: 395 Train loss: 0.051
Epoch: 9/10 Iteration: 400 Train loss: 0.075
Val acc: 0.826
```

Testing

In [47]:

```
test_acc = []
with tf.Session(graph=graph) as sess:
    saver.restore(sess, tf.train.latest_checkpoint('checkpoints'))
    test_state = sess.run(cell.zero_state(batch_size, tf.float32))
    for ii, (x, y) in enumerate(get_batches(test_x, test_y, batch_size), 1):
        feed = {inputs_: x,
                labels_: y[:, None],
                keep_prob: 1,
                initial_state: test_state}
        batch_acc, test_state = sess.run([accuracy, final_state],
feed_dict=feed)
        test_acc.append(batch_acc)
    print("Test accuracy: {:.3f}".format(np.mean(test_acc)))
```

```
Test accuracy: 0.830
```