Sentiment Classification & How To "Frame Problems" for a Neural Network

by Andrew Trask

- **Twitter**: @iamtrask
- **Blog**: http://iamtrask.github.io

What You Should Already Know

- neural networks, forward and back-propagation
- stochastic gradient descent
- mean squared error
- and train/test splits

Where to Get Help if You Need it

- Re-watch previous Udacity Lectures
- Leverage the recommended Course Reading Material - Grokking Deep Learning (Check inside your classroom for a discount code)
- Shoot me a tweet @iamtrask

Tutorial Outline:

- Intro: The Importance of "Framing a Problem" (this lesson)

- Curate a Dataset
- Developing a "Predictive Theory"
- **PROJECT 1**: Quick Theory Validation

- Transforming Text to Numbers
- **PROJECT 2**: Creating the Input/Output Data

- Putting it all together in a Neural Network (video only - nothing in notebook)
- **PROJECT 3**: Building our Neural Network

- Understanding Neural Noise
- **PROJECT 4**: Making Learning Faster by Reducing Noise

- Analyzing Inefficiencies in our Network
- **PROJECT 5**: Making our Network Train and Run Faster

- Further Noise Reduction
- **PROJECT 6**: Reducing Noise by Strategically Reducing the Vocabulary

- Analysis: What's going on in the weights?

Lesson: Curate a Dataset

In [1]:

```
def pretty_print_review_and_label(i):
    print(labels[i] + "\t:\t" + reviews[i][:80] + "...")

g = open('reviews.txt','r') # What we know!
reviews = list(map(lambda x:x[:-1],g.readlines()))
g.close()

g = open('labels.txt','r') # What we WANT to know!
labels = list(map(lambda x:x[:-1].upper(),g.readlines()))
g.close()
```

Note: The data in `reviews.txt` we're using has already been preprocessed a bit and contains only lower case characters. If we were working from raw data, where we didn't know it was all lower case, we would want to add a step here to convert it. That's so we treat different variations of the same word, like `The`, `the`, and `THE`, all the same way.

In [2]:

```
len(reviews)
```

Out[2]:

```
25000
```

In [3]:

```
reviews[0]
```

Out[3]:

```
'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 reality than is  teachers  . the scramble to survive financially  the insightful students who can see right through their pathetic teachers  pomp  the pettiness of the whole situation  all remind me of the schools i knew and their students . when i saw the episode in which a student repeatedly tried to burn down the school  i immediately recalled . . . . . . . . . at . . . . . . . . . . high . a classic line inspector i  m here to sack one of your teachers . student welcome to bromwell high . i expect that many adults of my age think that bromwell high is far fetched . what a pity that it isn  t   '
```

In [4]:

```
labels[0]
```

Out[4]:

```
'POSITIVE'
```

Lesson: Develop a Predictive Theory

In [5]:

```
print("labels.txt \t : \t reviews.txt\n")
pretty_print_review_and_label(2137)
pretty_print_review_and_label(12816)
pretty_print_review_and_label(6267)
pretty_print_review_and_label(21934)
pretty_print_review_and_label(5297)
pretty_print_review_and_label(4998)
```

```
labels.txt 	 : 	 reviews.txt

NEGATIVE	:	this movie is terrible but it has some good effects .  ...
POSITIVE	:	adrian pasdar is excellent is this film . he makes a fascinating woman .  ...
NEGATIVE	:	comment this movie is impossible . is terrible  very improbable  bad interpretat...
POSITIVE	:	excellent episode movie ala pulp fiction .  days   suicides . it doesnt get more...
NEGATIVE	:	if you haven  t seen this  it  s terrible . it is pure trash . i saw this about ...
POSITIVE	:	this schiffer guy is a real genius  the movie is of excellent quality and both e...
```

Project 1: Quick Theory Validation

There are multiple ways to implement these projects, but in order to get your code closer to what Andrew shows in his solutions, we've provided some hints and starter code throughout this notebook.

You'll find the Counter class to be useful in this exercise, as well as the numpy library.

In [6]:

```
from collections import Counter
import numpy as np
```

We'll create three `Counter` objects, one for words from postive reviews, one for words from negative reviews, and one for all the words.

In [7]:

```
# Create three Counter objects to store positive, negative and total counts
positive_counts = Counter()
negative_counts = Counter()
total_counts = Counter()
```

TODO: Examine all the reviews. For each word in a positive review, increase the count for that word in both your positive counter and the total words counter; likewise, for each word in a negative review, increase the count for that word in both your negative counter and the total words counter.

Note: Throughout these projects, you should use `split(' ')` to divide a piece of text (such as a review) into individual words. If you use `split()` instead, you'll get slightly different results than what the videos and solutions show.

In [8]:

```
# Loop over all the words in all the reviews and increment the counts in the appropriate counter objects
for i in range(len(reviews)):
    if(labels[i] == 'POSITIVE'):
        for word in reviews[i].split(" "):
            positive_counts[word] += 1
            total_counts[word] += 1
    else:
        for word in reviews[i].split(" "):
            negative_counts[word] += 1
            total_counts[word] += 1
```

Run the following two cells to list the words used in positive reviews and negative reviews, respectively, ordered from most to least commonly used.

In [9]:

```
# Examine the counts of the most common words in positive reviews
positive_counts.most_common()
```

Out[9]:

```
[('', 550468),
 ('the', 173324),
 ('.', 159654),
……
('waiting', 288),
 ...]
```

As you can see, common words like "the" appear very often in both positive and negative reviews. Instead of finding the most common words in positive or negative reviews, what you really want are the words found in positive reviews more often than in negative reviews, and vice versa. To accomplish this, you'll need to calculate the **ratios** of word usage between positive and negative reviews.

TODO: Check all the words you've seen and calculate the ratio of postive to negative uses and store that ratio in `pos_neg_ratios`.
Hint: the positive-to-negative ratio for a given word can be calculated with `positive_counts[word] / float(negative_counts[word]+1)`. Notice the `+1` in the denominator – that ensures we don't divide by zero for words that are only seen in positive reviews.

In [11]:

```
pos_neg_ratios = Counter()

# Calculate the ratios of positive and negative uses of the most common words
# Consider words to be "common" if they've been used at least 100 times
for term,cnt in list(total_counts.most_common()):
    if(cnt > 100):
        pos_neg_ratio = positive_counts[term] / float(negative_counts[term]+1)
        pos_neg_ratios[term] = pos_neg_ratio
```

Examine the ratios you've calculated for a few words:

In [12]:

```
print("Pos-to-neg ratio for 'the' = {}".format(pos_neg_ratios["the"]))
print("Pos-to-neg ratio for 'amazing' = {}".format(pos_neg_ratios["amazing"]))
print("Pos-to-neg ratio for 'terrible' = 
{}".format(pos_neg_ratios["terrible"]))
```

```
Pos-to-neg ratio for 'the' = 1.0607993145235326
Pos-to-neg ratio for 'amazing' = 4.022813688212928
Pos-to-neg ratio for 'terrible' = 0.17744252873563218
```

Looking closely at the values you just calculated, we see the following:

- Words that you would expect to see more often in positive reviews – like "amazing" – have a ratio greater than 1. The more skewed a word is toward postive, the farther from 1 its positive-to-negative ratio will be.
- Words that you would expect to see more often in negative reviews – like "terrible" – have positive values that are less than 1. The more skewed a word is toward negative, the closer to zero its positive-to-negative ratio will be.
- Neutral words, which don't really convey any sentiment because you would expect to see them in all sorts of reviews – like "the" – have values very close to 1. A perfectly neutral word – one that was used in exactly the same number of positive reviews as negative reviews – would be almost exactly 1. The `+1` we suggested you add to the denominator slightly biases words toward negative, but it won't matter because it will be a tiny bias and later we'll be ignoring words that are too close to neutral anyway.

Ok, the ratios tell us which words are used more often in postive or negative reviews, but the specific values we've calculated are a bit difficult to work with. A very positive word like "amazing" has a value above 4, whereas a very negative word like "terrible" has a value around 0.18. Those values aren't easy to compare for a couple of reasons:

- Right now, 1 is considered neutral, but the absolute value of the postive-to-negative rations of very postive words is larger than the absolute value of the ratios for the very negative words. So there is no way to directly compare two numbers and see if one word conveys the same magnitude of positive sentiment as another word conveys negative sentiment. So we should center all the values around netural so the absolute value fro neutral of the postive-to-negative ratio for a word would indicate how much sentiment (positive or negative) that word conveys.
- When comparing absolute values it's easier to do that around zero than one.

To fix these issues, we'll convert all of our ratios to new values using logarithms.

TODO: Go through all the ratios you calculated and convert their values using the following formulas:

- For any postive words, convert the ratio using `np.log(ratio)`
- For any negative words, convert the ratio using `-np.log(1/(ratio + 0.01))`

That second equation may look strange, but what it's doing is dividing one by a very small number, which will produce a larger positive number. Then, it takes the `log` of that, which produces numbers similar to the ones for the postive words. Finally, we negate the values by adding that minus sign up front. In the end, extremely positive and extremely negative words will have positive-to-negative ratios with similar magnitudes but oppositite signs.

In [13]:

```
# Convert ratios to logs
for word,ratio in pos_neg_ratios.most_common():
    if(ratio > 1):
        pos_neg_ratios[word] = np.log(ratio)
    else:
        pos_neg_ratios[word] = -np.log((1 / (ratio+0.01)))
```

Examine the new ratios you've calculated for the same words from before:

In [14]:

```
print("Pos-to-neg ratio for 'the' = {}".format(pos_neg_ratios["the"]))
print("Pos-to-neg ratio for 'amazing' = {}".format(pos_neg_ratios["amazing"]))
print("Pos-to-neg ratio for 'terrible' =
{}".format(pos_neg_ratios["terrible"]))
```

```
Pos-to-neg ratio for 'the' = 0.05902269426102881
Pos-to-neg ratio for 'amazing' = 1.3919815802404802
Pos-to-neg ratio for 'terrible' = -1.6742829939664696
```

If everything worked, now you should see neutral words with values close to zero. In this case, "the" is near zero but slightly positive, so it was probably used in more positive reviews than negative reviews. But look at "amazing"'s ratio - it's above `1`, showing it is clearly a word with positive sentiment. And "terrible" has a similar score, but in the opposite direction, so it's below `-1`. It's now clear that both of these words are associated with specific, opposing sentiments.

Now run the following cells to see more ratios.

The first cell displays all the words, ordered by how associated they are with postive reviews. (Your notebook will most likely truncate the output so you won't actually see *all* the words in the list.)

The second cell displays the 30 words most associated with negative reviews by reversing the order of the first list and then looking at the first 30 words. (If you want the second cell to display all the words, ordered by how associated they are with negative reviews, you could just write `reversed(pos_neg_ratios.most_common())`.)

You should continue to see values similar to the earlier ones we checked – neutral words will be close to `0`, words will get more positive as their ratios approach and go above `1`, and words will get more negative as their ratios approach and go below `-1`. That's why we decided to use the logs instead of the raw ratios.

In [15]:

```
# words most frequently seen in a review with a "POSITIVE" label
pos_neg_ratios.most_common()
```

Out[15]:

```
[('edie', 4.6913478822291435),
 ('paulie', 4.0775374439057197),
........
('lame', -1.9117232884159072),
 ('insult', -1.9085323769376259)]
```

End of Project 1.

Watch the next video to continue with Andrew's next lesson.

Transforming Text into Numbers

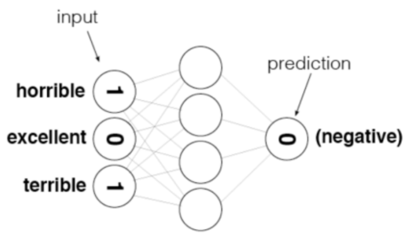
In [17]:

```
from IPython.display import Image

review = "This was a horrible, terrible movie."

Image(filename='sentiment_network.png')
```

Out[17]:

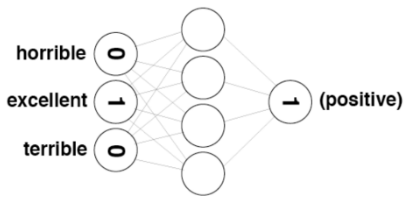


In [18]:

```
review = "The movie was excellent"

Image(filename='sentiment_network_pos.png')
```

Out[18]:



Project 2: Creating the Input/Output Data

TODO: Create a set named `vocab` that contains every word in the vocabulary.

In [19]:

```
vocab = set(total_counts.keys())
```

Run the following cell to check your vocabulary size. If everything worked correctly, it should print **74074**

In [20]:

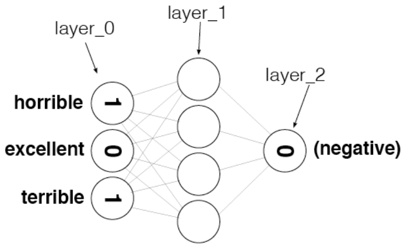
```
vocab_size = len(vocab)
print(vocab_size)
```

```
74074
```

Take a look at the following image. It represents the layers of the neural network you'll be building throughout this notebook. `layer_0` is the input layer, `layer_1` is a hidden layer, and `layer_2` is the output layer.

In [1]:

```
from IPython.display import Image
Image(filename='sentiment_network_2.png')
```



TODO: Create a numpy array called `layer_0` and initialize it to all zeros. You will find the zeros function particularly helpful here. Be sure you create `layer_0` as a 2-dimensional matrix with 1 row and `vocab_size` columns.

In [21]:

```
layer_0 = np.zeros((1,vocab_size))
```

Run the following cell. It should display `(1, 74074)`

In [22]:

```
layer_0.shape
```

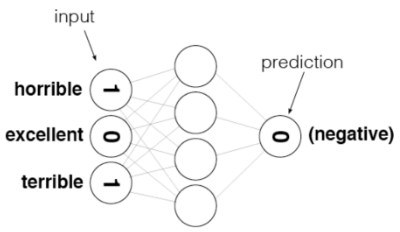
Out[22]:

```
(1, 74074)
```

In [23]:

```
from IPython.display import Image
Image(filename='sentiment_network.png')
```

Out[23]:



`layer_0` contains one entry for every word in the vocabulary, as shown in the above image. We need to make sure we know the index of each word, so run the following cell to create a lookup table that stores the index of every word.

In [24]:

```
# Create a dictionary of words in the vocabulary mapped to index positions
# (to be used in layer_0)
word2index = {}
for i,word in enumerate(vocab):
    word2index[word] = i

# display the map of words to indices
word2index
```

Out[24]:

```
{'': 0,
 'newmail': 1,
 'reaganism': 2,
.......
'kont': 999,
 ...}
```

TODO: Complete the implementation of `update_input_layer`. It should count how many times each word is used in the given review, and then store those counts at the appropriate indices inside `layer_0`.

In [25]:

```
def update_input_layer(review):
    """ Modify the global layer_0 to represent the vector form of review.
    The element at a given index of layer_0 should represent
    how many times the given word occurs in the review.
    Args:
        review(string) - the string of the review
    Returns:
```

```
        None
    """

    global layer_0

    # clear out previous state, reset the layer to be all 0s
    layer_0 *= 0

    # count how many times each word is used in the given review and store the results in layer_0
    for word in review.split(" "):
        layer_0[0][word2index[word]] += 1
```

Run the following cell to test updating the input layer with the first review. The indices assigned may not be the same as in the solution, but hopefully you'll see some non-zero values in `layer_0`.

In [26]:

```
update_input_layer(reviews[0])
layer_0
```

Out[26]:

```
array([[ 18.,   0.,   0., ...,   0.,   0.,   0.]])
```

TODO: Complete the implementation of `get_target_for_labels`. It should return `0` or `1`, depending on whether the given label is `NEGATIVE` or `POSITIVE`, respectively.

In [27]:

```
def get_target_for_label(label):
    """Convert a label to `0` or `1`.
    Args:
        label(string) - Either "POSITIVE" or "NEGATIVE".
    Returns:
        `0` or `1`.
    """
    if(label == 'POSITIVE'):
        return 1
    else:
        return 0
```

Run the following two cells. They should print out`'POSITIVE'` and `1`, respectively.

In [28]:

```
labels[0]
```

Out[28]:

```
'POSITIVE'
```

In [29]:

```
get_target_for_label(labels[0])
```

Out[29]:

```
1
```

Run the following two cells. They should print out `'NEGATIVE'` and `0`, respectively.

In [30]:

```
labels[1]
```

Out[30]:

```
'NEGATIVE'
```

In [31]:

```
get_target_for_label(labels[1])
```

Out[31]:

```
0
```

End of Project 2 solution.

Watch the next video to continue with Andrew's next lesson.

Project 3: Building a Neural Network

TODO: We've included the framework of a class called `SentimentNetork`. Implement all of the items marked `TODO` in the code. These include doing the following:

- Create a basic neural network much like the networks you've seen in earlier lessons and in Project 1, with an input layer, a hidden layer, and an output layer.
- Do **not** add a non-linearity in the hidden layer. That is, do not use an activation function when calculating the hidden layer outputs.
- Re-use the code from earlier in this notebook to create the training data (see `TODO`s in the code)
- Implement the `pre_process_data` function to create the vocabulary for our training data generating functions
- Ensure `train` trains over the entire corpus

Where to Get Help if You Need it

- Re-watch previous week's Udacity Lectures
- Chapters 3-5 - Grokking Deep Learning - (Check inside your classroom for a discount code)

In [32]:

```python
import time
import sys
import numpy as np

# Encapsulate our neural network in a class
class SentimentNetwork:
    def __init__(self, reviews,labels,hidden_nodes = 10, learning_rate = 0.1):
        """Create a SentimenNetwork with the given settings
        Args:
            reviews(list) - List of reviews used for training
            labels(list) - List of POSITIVE/NEGATIVE labels associated with the given reviews
            hidden_nodes(int) - Number of nodes to create in the hidden layer
            learning_rate(float) - Learning rate to use while training
        """
        # Assign a seed to our random number generator to ensure we get
        # reproducable results during development
        np.random.seed(1)

        # process the reviews and their associated labels so that everything
        # is ready for training
        self.pre_process_data(reviews, labels)

        # Build the network to have the number of hidden nodes and learning rate that
        # were passed into this initializer. Make the same number of input nodes as
        # there are vocabulary words and create a single output node.
        self.init_network(len(self.review_vocab),hidden_nodes, 1,learning_rate)
```

```
    def pre_process_data(self, reviews, labels):

        # populate review_vocab with all of the words in the given reviews
        review_vocab = set()
        for review in reviews:
            for word in review.split(" "):
                review_vocab.add(word)

        # Convert the vocabulary set to a list so we can access words via indices
        self.review_vocab = list(review_vocab)

        # populate label_vocab with all of the words in the given labels.
        label_vocab = set()
        for label in labels:
            label_vocab.add(label)

        # Convert the label vocabulary set to a list so we can access labels via indices
        self.label_vocab = list(label_vocab)

        # Store the sizes of the review and label vocabularies.
        self.review_vocab_size = len(self.review_vocab)
        self.label_vocab_size = len(self.label_vocab)

        # Create a dictionary of words in the vocabulary mapped to index positions
        self.word2index = {}
        for i, word in enumerate(self.review_vocab):
            self.word2index[word] = i

        # Create a dictionary of labels mapped to index positions
        self.label2index = {}
        for i, label in enumerate(self.label_vocab):
            self.label2index[label] = i

    def init_network(self, input_nodes, hidden_nodes, output_nodes, learning_rate):
        # Set number of nodes in input, hidden and output layers.
        self.input_nodes = input_nodes
        self.hidden_nodes = hidden_nodes
        self.output_nodes = output_nodes

        # Store the learning rate
        self.learning_rate = learning_rate

        # Initialize weights

        # These are the weights between the input layer and the hidden layer.
        self.weights_0_1 = np.zeros((self.input_nodes,self.hidden_nodes))

        # These are the weights between the hidden layer and the output layer.
        self.weights_1_2 = np.random.normal(0.0, self.output_nodes**-0.5,
                                                (self.hidden_nodes,
self.output_nodes))
```

```
        # The input layer, a two-dimensional matrix with shape 1 x input_nodes
        self.layer_0 = np.zeros((1,input_nodes))

    def update_input_layer(self,review):

        # clear out previous state, reset the layer to be all 0s
        self.layer_0 *= 0

        for word in review.split(" "):
          # NOTE: This if-check was not in the version of this method created in Project 2,
          #       and it appears in Andrew's Project 3 solution without explanation.
          #       It simply ensures the word is actually a key in word2index before
          #       accessing it, which is important because accessing an invalid key
          #       with raise an exception in Python. This allows us to ignore unknown
          #       words encountered in new reviews.
            if(word in self.word2index.keys()):
                self.layer_0[0][self.word2index[word]] += 1

    def get_target_for_label(self,label):
        if(label == 'POSITIVE'):
            return 1
        else:
            return 0

    def sigmoid(self,x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_output_2_derivative(self,output):
        return output * (1 - output)

    def train(self, training_reviews, training_labels):

        # make sure out we have a matching number of reviews and labels
        assert(len(training_reviews) == len(training_labels))

        # Keep track of correct predictions to display accuracy during training
        correct_so_far = 0

        # Remember when we started for printing time statistics
        start = time.time()

        # loop through all the given reviews and run a forward and backward pass,
        # updating weights for every item
        for i in range(len(training_reviews)):

            # Get the next review and its correct label
            review = training_reviews[i]
            label = training_labels[i]

            #### Implement the forward pass here ####
            ### Forward pass ###
```

```
            # Input Layer
            self.update_input_layer(review)

            # Hidden layer
            layer_1 = self.layer_0.dot(self.weights_0_1)

            # Output layer
            layer_2 = self.sigmoid(layer_1.dot(self.weights_1_2))

            #### Implement the backward pass here ####
            ### Backward pass ###

            # Output error
            layer_2_error = layer_2 - self.get_target_for_label(label) # Output
layer error is the difference between desired target and actual output.
            layer_2_delta = layer_2_error * self.sigmoid_output_2_derivative(layer_2)

            # Backpropagated error
            layer_1_error = layer_2_delta.dot(self.weights_1_2.T) # errors
propagated to the hidden layer
            layer_1_delta = layer_1_error # hidden layer gradients - no
nonlinearity so it's the same as the error

            # Update the weights
            self.weights_1_2 -= layer_1.T.dot(layer_2_delta) * self.learning_rate #
update hidden-to-output weights with gradient descent step
            self.weights_0_1 -= self.layer_0.T.dot(layer_1_delta) * self.learning_rate
# update input-to-hidden weights with gradient descent step

            # Keep track of correct predictions.
            if(layer_2 >= 0.5 and label == 'POSITIVE'):
                correct_so_far += 1
            elif(layer_2 < 0.5 and label == 'NEGATIVE'):
                correct_so_far += 1

            # For debug purposes, print out our prediction accuracy and speed
            # throughout the training process.
            elapsed_time = float(time.time() - start)
            reviews_per_second = i / elapsed_time if elapsed_time > 0 else 0

            sys.stdout.write("\rProgress:" + str(100 *
i/float(len(training_reviews)))[:4] \
                             + "% Speed(reviews/sec):" +
str(reviews_per_second)[0:5] \
                             + " #Correct:" + str(correct_so_far) + "
#Trained:" + str(i+1) \
                             + " Training Accuracy:" + str(correct_so_far * 100
/ float(i+1))[:4] + "%")
            if(i % 2500 == 0):
                print("")
```

```
    def test(self, testing_reviews, testing_labels):
        """
        Attempts to predict the labels for the given testing_reviews,
        and uses the test_labels to calculate the accuracy of those predictions.
        """
        # keep track of how many correct predictions we make
        correct = 0

        # we'll time how many predictions per second we make
        start = time.time()

        # Loop through each of the given reviews and call run to predict
        # its label.
        for i in range(len(testing_reviews)):
            pred = self.run(testing_reviews[i])
            if(pred == testing_labels[i]):
                correct += 1

            # For debug purposes, print out our prediction accuracy and speed
            # throughout the prediction process.

            elapsed_time = float(time.time() - start)
            reviews_per_second = i / elapsed_time if elapsed_time > 0 else 0

            sys.stdout.write("\rProgress:" + str(100 *
i/float(len(testing_reviews)))[:4] \
                             + "% Speed(reviews/sec):" + str(reviews_per_second)[0:5] \
                             + " #Correct:" + str(correct) + " #Tested:" + str(i+1) \
                             + " Testing Accuracy:" + str(correct * 100 /
float(i+1))[:4] + "%")

    def run(self, review):
        """
        Returns a POSITIVE or NEGATIVE prediction for the given review.
        """
        # Run a forward pass through the network, like in the "train" function.

        # Input Layer
        self.update_input_layer(review.lower())

        # Hidden layer
        layer_1 = self.layer_0.dot(self.weights_0_1)

        # Output layer
        layer_2 = self.sigmoid(layer_1.dot(self.weights_1_2))

        # Return POSITIVE for values above greater-than-or-equal-to 0.5 in the output layer;
        # return NEGATIVE for other values
        if(layer_2[0] >= 0.5):
            return "POSITIVE"
        else:
            return "NEGATIVE"
```

Run the following cell to create a `SentimentNetwork` that will train on all but the last 1000 reviews (we're saving those for testing). Here we use a learning rate of `0.1`.

In [33]:

```
mlp = SentimentNetwork(reviews[:-1000],labels[:-1000], learning_rate=0.1)
```

Run the following cell to test the network's performance against the last 1000 reviews (the ones we held out from our training set).

We have not trained the model yet, so the results should be about 50% as it will just be guessing and there are only two possible values to choose from.

In [34]:

```
mlp.test(reviews[-1000:],labels[-1000:])
```

```
Progress:99.9% Speed(reviews/sec):1065. #Correct:500 #Tested:1000 Testing Accuracy:50.0%
```

Run the following cell to actually train the network. During training, it will display the model's accuracy repeatedly as it trains so you can see how well it's doing.

In [35]:

```
mlp.train(reviews[:-1000],labels[:-1000])
```

```
Progress:0.0% Speed(reviews/sec):0.0 #Correct:1 #Trained:1 Training Accuracy:100.%
Progress:10.4% Speed(reviews/sec):227.4 #Correct:1251 #Trained:2501 Training Accuracy:50.0%
......
Progress:93.7% Speed(reviews/sec):219.2 #Correct:11251 #Trained:22501 Training Accuracy:50.0%
Progress:99.9% Speed(reviews/sec):219.0 #Correct:12000 #Trained:24000 Training Accuracy:50.0%
```

That most likely didn't train very well. Part of the reason may be because the learning rate is too high. Run the following cell to recreate the network with a smaller learning rate, `0.01`, and then train the new network.

In [36]:

```
mlp = SentimentNetwork(reviews[:-1000],labels[:-1000], learning_rate=0.01)
mlp.train(reviews[:-1000],labels[:-1000])
```

```
Progress:0.0% Speed(reviews/sec):0.0 #Correct:1 #Trained:1 Training Accuracy:100.%
.....
Progress:99.9% Speed(reviews/sec):161.3 #Correct:11990 #Trained:24000 Training Accuracy:49.9%
```

That probably wasn't much different. Run the following cell to recreate the network one more time with an even smaller learning rate, `0.001`, and then train the new network.

In [37]:

```
mlp = SentimentNetwork(reviews[:-1000],labels[:-1000], learning_rate=0.001)
mlp.train(reviews[:-1000],labels[:-1000])
```

```
Progress:0.0% Speed(reviews/sec):0.0 #Correct:1 #Trained:1 Training Accuracy:100.%
Progress:10.4% Speed(reviews/sec):170.6 #Correct:1256 #Trained:2501 Training Accuracy:50.2%
```

With a learning rate of `0.001`, the network should finall have started to improve during training. It's still not very good, but it shows that this solution has potential. We will improve it in the next lesson.

End of Project 3.

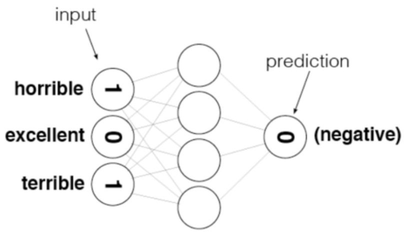
Watch the next video to continue with Andrew's next lesson.

Understanding Neural Noise

In [38]:

```
from IPython.display import Image
Image(filename='sentiment_network.png')
```

Out[38]:



In [39]:

```
def update_input_layer(review):

    global layer_0

    # clear out previous state, reset the layer to be all 0s
    layer_0 *= 0
    for word in review.split(" "):
        layer_0[0][word2index[word]] += 1

update_input_layer(reviews[0])
```

In [40]:

```
layer_0
```

Out[40]:

```
array([[ 18.,   0.,   0., ...,   0.,   0.,   0.]])
```

In [41]:

```
review_counter = Counter()
```

In [42]:

```
for word in reviews[0].split(" "):
    review_counter[word] += 1
```

In [43]:

```
review_counter.most_common()
```

Out[43]:

```
[('.', 27),
 ('', 18),
 ('the', 9),
.....
('isn', 1),
 ('t', 1)]
```

Project 4: Reducing Noise in Our Input Data

TODO: Attempt to reduce the noise in the input data like Andrew did in the previous video. Specifically, do the following:

- Copy the `SentimentNetwork` class you created earlier into the following cell.
- Modify `update_input_layer` so it does not count how many times each word is used, but rather just stores whether or not a word was used.

The following code is the same as the previous project, with project-specific changes marked with "`New for Project 4`"

In [44]:

```
import time
import sys
import numpy as np

# Encapsulate our neural network in a class
class SentimentNetwork:
    def __init__(self, reviews,labels,hidden_nodes = 10, learning_rate = 0.1):
        """Create a SentimenNetwork with the given settings
        Args:
            reviews(list) - List of reviews used for training
            labels(list) - List of POSITIVE/NEGATIVE labels associated with the given reviews
            hidden_nodes(int) - Number of nodes to create in the hidden layer
            learning_rate(float) - Learning rate to use while training
        """
        # Assign a seed to our random number generator to ensure we get
        # reproducable results during development
        np.random.seed(1)

        # process the reviews and their associated labels so that everything
        # is ready for training
        self.pre_process_data(reviews, labels)

        # Build the network to have the number of hidden nodes and the learning rate that
        # were passed into this initializer. Make the same number of input nodes as
        # there are vocabulary words and create a single output node.
        self.init_network(len(self.review_vocab),hidden_nodes,1, learning_rate)

    def pre_process_data(self, reviews, labels):

        # populate review_vocab with all of the words in the given reviews
        review_vocab = set()
        for review in reviews:
            for word in review.split(" "):
                review_vocab.add(word)

        # Convert the vocabulary set to a list so we can access words via indices
        self.review_vocab = list(review_vocab)

        # populate label_vocab with all of the words in the given labels.
        label_vocab = set()
        for label in labels:
            label_vocab.add(label)

        # Convert the label vocabulary set to a list so we can access labels via indices
        self.label_vocab = list(label_vocab)

        # Store the sizes of the review and label vocabularies.
        self.review_vocab_size = len(self.review_vocab)
        self.label_vocab_size = len(self.label_vocab)
```

```
        # Create a dictionary of words in the vocabulary mapped to index positions
        self.word2index = {}
        for i, word in enumerate(self.review_vocab):
            self.word2index[word] = i

        # Create a dictionary of labels mapped to index positions
        self.label2index = {}
        for i, label in enumerate(self.label_vocab):
            self.label2index[label] = i

    def init_network(self,input_nodes,hidden_nodes,output_nodes,learning_rate):
        # Set number of nodes in input, hidden and output layers.
        self.input_nodes = input_nodes
        self.hidden_nodes = hidden_nodes
        self.output_nodes = output_nodes

        # Store the learning rate
        self.learning_rate = learning_rate

        # Initialize weights

        # These are the weights between the input layer and the hidden layer.
        self.weights_0_1 = np.zeros((self.input_nodes,self.hidden_nodes))

        # These are the weights between the hidden layer and the output layer.
        self.weights_1_2 = np.random.normal(0.0, self.output_nodes**-0.5,
                                                (self.hidden_nodes, self.output_nodes))

        # The input layer, a two-dimensional matrix with shape 1 x input_nodes
        self.layer_0 = np.zeros((1,input_nodes))

    def update_input_layer(self,review):

        # clear out previous state, reset the layer to be all 0s
        self.layer_0 *= 0

        for word in review.split(" "):
          # NOTE: This if-check was not in the version of this method created in Project 2,
          #       and it appears in Andrew's Project 3 solution without explanation.
          #       It simply ensures the word is actually a key in word2index before
          #       accessing it, which is important because accessing an invalid key
          #       with raise an exception in Python. This allows us to ignore unknown
          #       words encountered in new reviews.
            if(word in self.word2index.keys()):
                ## New for Project 4: changed to set to 1 instead of add 1
                self.layer_0[0][self.word2index[word]] = 1

    def get_target_for_label(self,label):
        if(label == 'POSITIVE'):
            return 1
        else:
            return 0
```

```
    def sigmoid(self,x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_output_2_derivative(self,output):
        return output * (1 - output)

    def train(self, training_reviews, training_labels):

        # make sure out we have a matching number of reviews and labels
        assert(len(training_reviews) == len(training_labels))

        # Keep track of correct predictions to display accuracy during training
        correct_so_far = 0

        # Remember when we started for printing time statistics
        start = time.time()

        # loop through all the given reviews and run a forward and backward pass,
        # updating weights for every item
        for i in range(len(training_reviews)):

            # Get the next review and its correct label
            review = training_reviews[i]
            label = training_labels[i]

            #### Implement the forward pass here ####
            ### Forward pass ###

            # Input Layer
            self.update_input_layer(review)

            # Hidden layer
            layer_1 = self.layer_0.dot(self.weights_0_1)

            # Output layer
            layer_2 = self.sigmoid(layer_1.dot(self.weights_1_2))

            #### Implement the backward pass here ####
            ### Backward pass ###

            # Output error
            layer_2_error = layer_2 - self.get_target_for_label(label) # Output
layer error is the difference between desired target and actual output.
            layer_2_delta = layer_2_error *
self.sigmoid_output_2_derivative(layer_2)

            # Backpropagated error
            layer_1_error = layer_2_delta.dot(self.weights_1_2.T) # errors
propagated to the hidden layer
            layer_1_delta = layer_1_error # hidden layer gradients - no
nonlinearity so it's the same as the error
```

```
            # Update the weights
            self.weights_1_2 -= layer_1.T.dot(layer_2_delta) * self.learning_rate
# update hidden-to-output weights with gradient descent step
            self.weights_0_1 -= self.layer_0.T.dot(layer_1_delta) * self.learning_rate
# update input-to-hidden weights with gradient descent step

            # Keep track of correct predictions.
            if(layer_2 >= 0.5 and label == 'POSITIVE'):
                correct_so_far += 1
            elif(layer_2 < 0.5 and label == 'NEGATIVE'):
                correct_so_far += 1

            # For debug purposes, print out our prediction accuracy and speed
            # throughout the training process.
            elapsed_time = float(time.time() - start)
            reviews_per_second = i / elapsed_time if elapsed_time > 0 else 0

           sys.stdout.write("\rProgress:" + str(100 *
i/float(len(training_reviews)))[:4] \
                             + "% Speed(reviews/sec):" + str(reviews_per_second)[0:5] \
                             + " #Correct:" + str(correct_so_far) + " #Trained:" +
str(i+1) \
                             + " Training Accuracy:" + str(correct_so_far * 100 /
float(i+1))[:4] + "%")
            if(i % 2500 == 0):
                print("")

    def test(self, testing_reviews, testing_labels):
        """
        Attempts to predict the labels for the given testing_reviews,
        and uses the test_labels to calculate the accuracy of those predictions.
        """

        # keep track of how many correct predictions we make
        correct = 0

        # we'll time how many predictions per second we make
        start = time.time()

        # Loop through each of the given reviews and call run to predict
        # its label.
        for i in range(len(testing_reviews)):
            pred = self.run(testing_reviews[i])
            if(pred == testing_labels[i]):
                correct += 1

            # For debug purposes, print out our prediction accuracy and speed
            # throughout the prediction process.

            elapsed_time = float(time.time() - start)
            reviews_per_second = i / elapsed_time if elapsed_time > 0 else 0
```

```
            sys.stdout.write("\rProgress:"+str(100*i/float(len(testing_reviews)))[:4] \
                             + "% Speed(reviews/sec):" + str(reviews_per_second)[0:5] \
                             + " #Correct:" + str(correct) + " #Tested:" + str(i+1) \
                             + " Testing Accuracy:"+str(correct*100/float(i+1))[:4]+"%")

    def run(self, review):
        """
        Returns a POSITIVE or NEGATIVE prediction for the given review.
        """
        # Run a forward pass through the network, like in the "train" function.

        # Input Layer
        self.update_input_layer(review.lower())

        # Hidden layer
        layer_1 = self.layer_0.dot(self.weights_0_1)

        # Output layer
        layer_2 = self.sigmoid(layer_1.dot(self.weights_1_2))

        # Return POSITIVE for values above greater-than-or-equal-to 0.5 in the output layer;
        # return NEGATIVE for other values
        if(layer_2[0] >= 0.5):
            return "POSITIVE"
        else:
            return "NEGATIVE"
```

Run the following cell to recreate the network and train it. Notice we've gone back to the higher learning rate of `0.1`.

In [45]:

```
mlp = SentimentNetwork(reviews[:-1000],labels[:-1000], learning_rate=0.1)
mlp.train(reviews[:-1000],labels[:-1000])
```

```
Progress:0.0% Speed(reviews/sec):0.0 #Correct:1 #Trained:1 Training
Accuracy:100.%
Progress:10.4% Speed(reviews/sec):170.0 #Correct:1796 #Trained:2501 Training
Accuracy:71.8%
……
Progress:72.9% Speed(reviews/sec):160.3 #Correct:14394 #Trained:17501 Training
Accuracy:82.2%
……
Progress:99.9% Speed(reviews/sec):160.3 #Correct:20084 #Trained:24000 Training
Accuracy:83.6%
```

In [46]:

```
# evaluate our model before training (just to show how horrible it is)
mlp.test(reviews[-1000:],labels[-1000:])
```

```
Progress:99.9% Speed(reviews/sec):1280. #Correct:857 #Tested:1000 Testing
Accuracy:85.7%
```

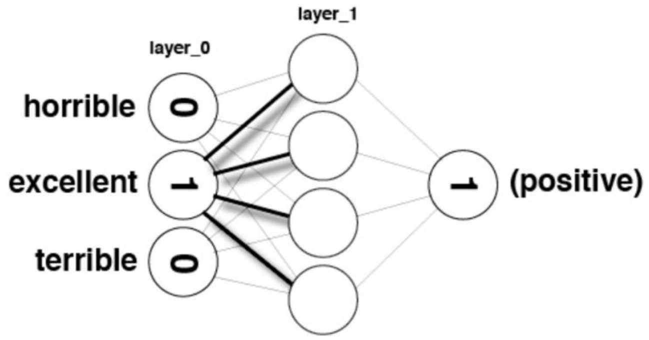
End of Project 4 solution.

Watch the next video to continue with Andrew's next lesson.

Analyzing Inefficiencies in our Network

```
Image(filename='sentiment_network_sparse.png')
```

Out[47]:



In [48]:

```
layer_0 = np.zeros(10)
```

In [49]:

```
layer_0
```

Out[49]:

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

In [50]:

```
layer_0[4] = 1
layer_0[9] = 1
```

In [51]:

```
layer_0
```

Out[51]:

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

In [52]:

```
weights_0_1 = np.random.randn(10,5)
```

In [53]:

```
layer_0.dot(weights_0_1)
```

Out[53]:

```
array([-0.10503756,  0.44222989,  0.24392938, -0.55961832,  0.21389503])
```

In [54]:

```
indices = [4,9]
```

In [55]:

```
layer_1 = np.zeros(5)
```

In [56]:

```
for index in indices:
    layer_1 += (1 * weights_0_1[index])
```

In [57]:

```
layer_1
```

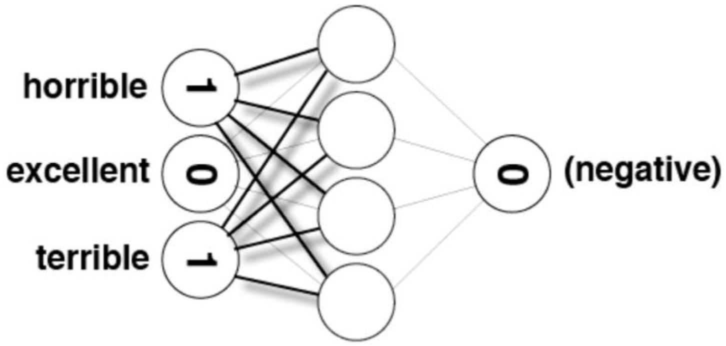
Out[57]:

```
array([-0.10503756,  0.44222989,  0.24392938, -0.55961832,  0.21389503])
```

In [58]:

```
Image(filename='sentiment_network_sparse_2.png')
```

Out[58]:



In [59]:

```
layer_1 = np.zeros(5)
```

In [60]:

```
for index in indices:
    layer_1 += (weights_0_1[index])
```

In [61]:

```
layer_1
```

Out[61]:

```
array([-0.10503756,  0.44222989,  0.24392938, -0.55961832,  0.21389503])
```

Project 5: Making our Network More Efficient

TODO: Make the `SentimentNetwork` class more efficient by eliminating unnecessary multiplications and additions that occur during forward and backward propagation. To do that, you can do the following:

- Copy the `SentimentNetwork` class from the previous project into the following cell.
- Remove the `update_input_layer` function - you will not need it in this version.
- Modify `init_network`:
  - You no longer need a separate input layer, so remove any mention of `self.layer_0`
  - You will be dealing with the old hidden layer more directly, so create `self.layer_1`, a two-dimensional matrix with shape 1 x hidden_nodes, with all values initialized to zero
- Modify `train`:
  - Change the name of the input parameter `training_reviews` to `training_reviews_raw`. This will help with the next step.
  - At the beginning of the function, you'll want to preprocess your reviews to convert them to a list of indices (from `word2index`) that are actually used in the review. This is equivalent to what you saw in the video when Andrew set specific indices to 1. Your code should create a local `list` variable named `training_reviews` that should contain a `list` for each review in `training_reviews_raw`. Those lists should contain the indices for words found in the review.
  - Remove call to `update_input_layer`
  - Use `self`'s `layer_1` instead of a local `layer_1` object.
  - In the forward pass, replace the code that updates `layer_1` with new logic that only adds the weights for the indices used in the review.
  - When updating `weights_0_1`, only update the individual weights that were used in the forward pass.
- Modify `run`:
  - Remove call to `update_input_layer`
  - Use `self`'s `layer_1` instead of a local `layer_1` object.
  - Much like you did in `train`, you will need to pre-process the `review` so you can work with word indices, then update `layer_1` by adding weights for the indices used in the review.

The following code is the same as the previous project, with project-specific changes marked with "`New for Project 5`"

In [62]:

```
import time
import sys
import numpy as np

# Encapsulate our neural network in a class
class SentimentNetwork:
    def __init__(self, reviews,labels,hidden_nodes = 10, learning_rate = 0.1):
        """Create a SentimenNetwork with the given settings
        Args:
            reviews(list) - List of reviews used for training
            labels(list) - List of POSITIVE/NEGATIVE labels associated with the given reviews
            hidden_nodes(int) - Number of nodes to create in the hidden layer
            learning_rate(float) - Learning rate to use while training
        """

        # Assign a seed to our random number generator to ensure we get
        # reproducable results during development
        np.random.seed(1)

        # process the reviews and their associated labels so that everything
        # is ready for training
        self.pre_process_data(reviews, labels)

        # Build the network to have the number of hidden nodes and the learning rate that
        # were passed into this initializer. Make the same number of input nodes as
        # there are vocabulary words and create a single output node.
        self.init_network(len(self.review_vocab),hidden_nodes,1, learning_rate)

    def pre_process_data(self, reviews, labels):

        # populate review_vocab with all of the words in the given reviews
        review_vocab = set()
        for review in reviews:
            for word in review.split(" "):
                review_vocab.add(word)

        # Convert the vocabulary set to a list so we can access words via indices
        self.review_vocab = list(review_vocab)

        # populate label_vocab with all of the words in the given labels.
        label_vocab = set()
        for label in labels:
            label_vocab.add(label)

        # Convert the label vocabulary set to a list so we can access labels via indices
        self.label_vocab = list(label_vocab)

        # Store the sizes of the review and label vocabularies.
        self.review_vocab_size = len(self.review_vocab)
        self.label_vocab_size = len(self.label_vocab)
```

```
        # Create a dictionary of words in the vocabulary mapped to index positions
        self.word2index = {}
        for i, word in enumerate(self.review_vocab):
            self.word2index[word] = i

        # Create a dictionary of labels mapped to index positions
        self.label2index = {}
        for i, label in enumerate(self.label_vocab):
            self.label2index[label] = i

    def init_network(self,input_nodes,hidden_nodes,output_nodes,learning_rate):
        # Set number of nodes in input, hidden and output layers.
        self.input_nodes = input_nodes
        self.hidden_nodes = hidden_nodes
        self.output_nodes = output_nodes

        # Store the learning rate
        self.learning_rate = learning_rate

        # Initialize weights

        # These are the weights between the input layer and the hidden layer.
        self.weights_0_1 = np.zeros((self.input_nodes,self.hidden_nodes))

        # These are the weights between the hidden layer and the output layer.
        self.weights_1_2 = np.random.normal(0.0, self.output_nodes**-0.5,
                                                (self.hidden_nodes, self.output_nodes))

        ## New for Project 5: Removed self.layer_0; added self.layer_1
        # The input layer, a two-dimensional matrix with shape 1 x hidden_nodes
        self.layer_1 = np.zeros((1,hidden_nodes))

    ## New for Project 5: Removed update_input_layer function

    def get_target_for_label(self,label):
        if(label == 'POSITIVE'):
            return 1
        else:
            return 0

    def sigmoid(self,x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_output_2_derivative(self,output):
        return output * (1 - output)

    ## New for Project 5: changed name of first parameter form 'training_reviews'
    #                     to 'training_reviews_raw'
    def train(self, training_reviews_raw, training_labels):

        ## New for Project 5: pre-process training reviews so we can deal
        #                     directly with the indices of non-zero inputs
```

```
        training_reviews = list()
        for review in training_reviews_raw:
            indices = set()
            for word in review.split(" "):
                if(word in self.word2index.keys()):
                    indices.add(self.word2index[word])
            training_reviews.append(list(indices))

        # make sure out we have a matching number of reviews and labels
        assert(len(training_reviews) == len(training_labels))

        # Keep track of correct predictions to display accuracy during training
        correct_so_far = 0

        # Remember when we started for printing time statistics
        start = time.time()

        # loop through all the given reviews and run a forward and backward
pass,
        # updating weights for every item
        for i in range(len(training_reviews)):

            # Get the next review and its correct label
            review = training_reviews[i]
            label = training_labels[i]

            #### Implement the forward pass here ####
            ### Forward pass ###

            ## New for Project 5: Removed call to 'update_input_layer' function
            #                     because 'layer_0' is no longer used

            # Hidden layer
            ## New for Project 5: Add in only the weights for non-zero items
            self.layer_1 *= 0
            for index in review:
                self.layer_1 += self.weights_0_1[index]

            # Output layer
            ## New for Project 5: changed to use 'self.layer_1' instead of
'local layer_1'
            layer_2 = self.sigmoid(self.layer_1.dot(self.weights_1_2))

            #### Implement the backward pass here ####
            ### Backward pass ###

            # Output error
            layer_2_error = layer_2 - self.get_target_for_label(label) # Output
layer error is the difference between desired target and actual output.
            layer_2_delta = layer_2_error *
self.sigmoid_output_2_derivative(layer_2)
```

```
            # Backpropagated error
            layer_1_error = layer_2_delta.dot(self.weights_1_2.T) # errors propagated to the hidden layer
            layer_1_delta = layer_1_error # hidden layer gradients - no nonlinearity so it's the same as the error

            # Update the weights
            ## New for Project 5: changed to use 'self.layer_1' instead of local 'layer_1'
            self.weights_1_2 -= self.layer_1.T.dot(layer_2_delta) * self.learning_rate # update hidden-to-output weights with gradient descent step

            ## New for Project 5: Only update the weights that were used in the forward pass
            for index in review:
                self.weights_0_1[index] -= layer_1_delta[0] * self.learning_rate # update input-to-hidden weights with gradient descent step

            # Keep track of correct predictions.
            if(layer_2 >= 0.5 and label == 'POSITIVE'):
                correct_so_far += 1
            elif(layer_2 < 0.5 and label == 'NEGATIVE'):
                correct_so_far += 1

            # For debug purposes, print out our prediction accuracy and speed
            # throughout the training process.
            elapsed_time = float(time.time() - start)
            reviews_per_second = i / elapsed_time if elapsed_time > 0 else 0

            sys.stdout.write("\rProgress:" + str(100 * i/float(len(training_reviews)))[:4] \
                             + "% Speed(reviews/sec):" + str(reviews_per_second)[0:5] \
                             + " #Correct:" + str(correct_so_far) + " #Trained:" + str(i+1) \
                             + " Training Accuracy:"+str(correct_so_far*100/float(i+1))[:4] + "%")
            if(i % 2500 == 0):
                print("")

    def test(self, testing_reviews, testing_labels):
        """
        Attempts to predict the labels for the given testing_reviews,
        and uses the test_labels to calculate the accuracy of those predictions.
        """
        # keep track of how many correct predictions we make
        correct = 0

        # we'll time how many predictions per second we make
        start = time.time()

        # Loop through each of the given reviews and call run to predict
        # its label.
        for i in range(len(testing_reviews)):
            pred = self.run(testing_reviews[i])
            if(pred == testing_labels[i]):
                correct += 1
```

```
            # For debug purposes, print out our prediction accuracy and speed
            # throughout the prediction process.

            elapsed_time = float(time.time() - start)
            reviews_per_second = i / elapsed_time if elapsed_time > 0 else 0

            sys.stdout.write("\rProgress:" + str(100 * i/float(len(testing_reviews)))[:4] \
                             + "% Speed(reviews/sec):" + str(reviews_per_second)[0:5] \
                             + " #Correct:" + str(correct) + " #Tested:" + str(i+1) \
                             + " Testing Accuracy:" + str(correct * 100 / float(i+1))[:4] + "%")

    def run(self, review):
        """
        Returns a POSITIVE or NEGATIVE prediction for the given review.
        """
        # Run a forward pass through the network, like in the "train" function.

        ## New for Project 5: Removed call to update_input_layer function
        #                     because layer_0 is no longer used

        # Hidden layer
        ## New for Project 5: Identify the indices used in the review and then add
        #                     just those weights to layer_1
        self.layer_1 *= 0
        unique_indices = set()
        for word in review.lower().split(" "):
            if word in self.word2index.keys():
                unique_indices.add(self.word2index[word])
        for index in unique_indices:
            self.layer_1 += self.weights_0_1[index]

        # Output layer
        ## New for Project 5: changed to use self.layer_1 instead of local layer_1
        layer_2 = self.sigmoid(self.layer_1.dot(self.weights_1_2))

        # Return POSITIVE for values above greater-than-or-equal-to 0.5 in the output layer;
        # return NEGATIVE for other values
        if(layer_2[0] >= 0.5):
            return "POSITIVE"
        else:
            return "NEGATIVE"
```

Run the following cell to recreate the network and train it once again.

In [63]:

```
mlp = SentimentNetwork(reviews[:-1000],labels[:-1000], learning_rate=0.1)
mlp.train(reviews[:-1000],labels[:-1000])
```

```
Progress:0.0% Speed(reviews/sec):0.0 #Correct:1 #Trained:1 Training
Accuracy:100.%
…..
Progress:52.0% Speed(reviews/sec):1145. #Correct:10014 #Trained:12501 Training
Accuracy:80.1%
```

```
…..
Progress:99.9% Speed(reviews/sec):1130. #Correct:19945 #Trained:24000 Training
Accuracy:83.1%
```

That should have trained much better than the earlier attempts. Run the following cell to test your model with 1000 predictions.

In [64]:

```
mlp.test(reviews[-1000:],labels[-1000:])
```

```
Progress:99.9% Speed(reviews/sec):1623. #Correct:846 #Tested:1000 Testing
Accuracy:84.6%
```

End of Project 5 solution.

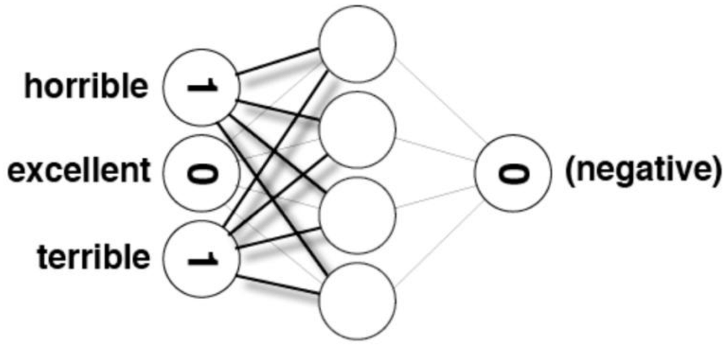
Watch the next video to continue with Andrew's next lesson.

Further Noise Reduction

In [65]:

```
Image(filename='sentiment_network_sparse_2.png')
```

Out[65]:



In [66]:

```
# words most frequently seen in a review with a "POSITIVE" label
pos_neg_ratios.most_common()
```

Out[66]:

```
[('edie', 4.6913478822291435),
 ('paulie', 4.0775374439057197),
…..
('miss', 0.43006426712153217),
 ('movement', 0.42956265968722499),
 ...]
```

In [67]:

```
# words most frequently seen in a review with a "NEGATIVE" label
list(reversed(pos_neg_ratios.most_common()))[0:30]
```

Out[67]:

```
[('boll', -4.0778152602708904),
 ('uwe', -3.9218753018711578),
…..
('lame', -1.9117232884159072),
 ('insult', -1.9085323769376259)]
```

In [68]:

```
from bokeh.models import ColumnDataSource, LabelSet
from bokeh.plotting import figure, show, output_file
from bokeh.io import output_notebook
```

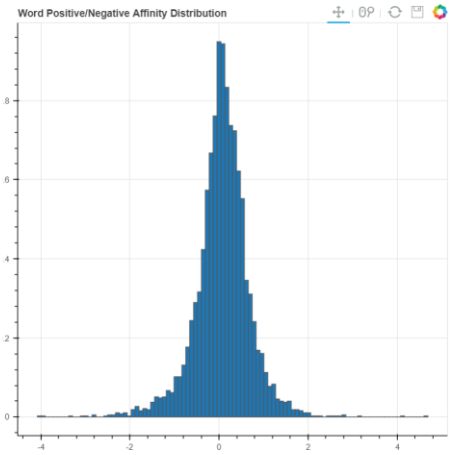
```
output_notebook()
```

```
 BokehJS successfully loaded.
```

In [69]:

```
hist, edges = np.histogram(list(map(lambda x:x[1],pos_neg_ratios.most_common())), density=True, bins=100, normed=True)

p = figure(tools="pan,wheel_zoom,reset,save",
           toolbar_location="above",
           title="Word Positive/Negative Affinity Distribution")
p.quad(top=hist, bottom=0, left=edges[:-1], right=edges[1:], line_color="#555555")
show(p)
```



In [70]:

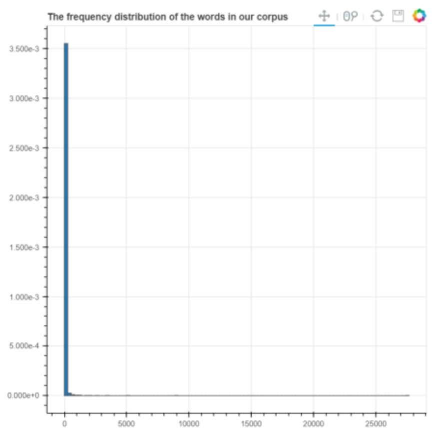
```
frequency_frequency = Counter()

for word, cnt in total_counts.most_common():
    frequency_frequency[cnt] += 1
```

In [71]:

```
hist, edges = np.histogram(list(map(lambda x:x[1],frequency_frequency.most_common())), density=True, bins=100, normed=True)

p = figure(tools="pan,wheel_zoom,reset,save",
           toolbar_location="above",
           title="The frequency distribution of the words in our corpus")
p.quad(top=hist, bottom=0, left=edges[:-1], right=edges[1:], line_color="#555555")
show(p)
```



Project 6: Reducing Noise by Strategically Reducing the Vocabulary

TODO: Improve `SentimentNetwork`'s performance by reducing more noise in the vocabulary. Specifically, do the following:

- Copy the `SentimentNetwork` class from the previous project into the following cell.
- Modify `pre_process_data`:
  - Add two additional parameters: `min_count` and `polarity_cutoff`
  - Calculate the positive-to-negative ratios of words used in the reviews. (You can use code you've written elsewhere in the notebook, but we are moving it into the class like we did with other helper code earlier.)
  - Andrew's solution only calculates a postive-to-negative ratio for words that occur at least 50 times. This keeps the network from attributing too much sentiment to rarer words. You can choose to add this to your solution if you would like.
  - Change so words are only added to the vocabulary if they occur in the vocabulary more than `min_count` times.
  - Change so words are only added to the vocabulary if the absolute value of their postive-to-negative ratio is at least `polarity_cutoff`
- Modify `__init__`:
  - Add the same two parameters (`min_count` and `polarity_cutoff`) and use them when you call `pre_process_data`

The following code is the same as the previous project, with project-specific changes marked with `"New for Project 6"`

In [72]:

```
import time
import sys
import numpy as np

# Encapsulate our neural network in a class
class SentimentNetwork:
    ## New for Project 6: added min_count and polarity_cutoff parameters
    def __init__(self, reviews,labels,min_count = 10,polarity_cutoff = 
0.1,hidden_nodes = 10, learning_rate = 0.1):
        """Create a SentimenNetwork with the given settings
        Args:
            reviews(list) - List of reviews used for training
            labels(list) - List of POSITIVE/NEGATIVE labels associated with the given reviews
            min_count(int) - Words should only be added to the vocabulary 
                             if they occur more than this many times
            polarity_cutoff(float) - The absolute value of a word's positive-to-negative
                                     ratio must be at least this big to be considered.
            hidden_nodes(int) - Number of nodes to create in the hidden layer
            learning_rate(float) - Learning rate to use while training
        """
        # Assign a seed to our random number generator to ensure we get
        # reproducable results during development
        np.random.seed(1)

        # process the reviews and their associated labels so that everything
        # is ready for training
        ## New for Project 6: added min_count and polarity_cutoff arguments to 
pre_process_data call
        self.pre_process_data(reviews, labels, polarity_cutoff, min_count)
```

```
        # Build the network to have the number of hidden nodes and the learning rate that
        # were passed into this initializer. Make the same number of input nodes as
        # there are vocabulary words and create a single output node.
        self.init_network(len(self.review_vocab),hidden_nodes, 1,
learning_rate)

    ## New for Project 6: added min_count and polarity_cutoff parameters
    def pre_process_data(self, reviews, labels, polarity_cutoff, min_count):

        ## ----------------------------------------
        ## New for Project 6: Calculate positive-to-negative ratios for words before
        #                     building vocabulary
        positive_counts = Counter()
        negative_counts = Counter()
        total_counts = Counter()

        for i in range(len(reviews)):
            if(labels[i] == 'POSITIVE'):
                for word in reviews[i].split(" "):
                    positive_counts[word] += 1
                    total_counts[word] += 1
            else:
                for word in reviews[i].split(" "):
                    negative_counts[word] += 1
                    total_counts[word] += 1

        pos_neg_ratios = Counter()

        for term,cnt in list(total_counts.most_common()):
            if(cnt >= 50):
                pos_neg_ratio = positive_counts[term] /
float(negative_counts[term]+1)
                pos_neg_ratios[term] = pos_neg_ratio

        for word,ratio in pos_neg_ratios.most_common():
            if(ratio > 1):
                pos_neg_ratios[word] = np.log(ratio)
            else:
                pos_neg_ratios[word] = -np.log((1 / (ratio + 0.01)))
        #
        ## end New for Project 6
        ## ----------------------------------------

        # populate review_vocab with all of the words in the given reviews
        review_vocab = set()
        for review in reviews:
            for word in review.split(" "):
                ## New for Project 6: only add words that occur at least min_count times
                #                     and for words with pos/neg ratios, only add words
                #                     that meet the polarity_cutoff
                if(total_counts[word] > min_count):
                    if(word in pos_neg_ratios.keys()):
```

```
                        if((pos_neg_ratios[word] >= polarity_cutoff) or
(pos_neg_ratios[word] <= -polarity_cutoff)):
                            review_vocab.add(word)
                    else:
                        review_vocab.add(word)

        # Convert the vocabulary set to a list so we can access words via indices
        self.review_vocab = list(review_vocab)

        # populate label_vocab with all of the words in the given labels.
        label_vocab = set()
        for label in labels:
            label_vocab.add(label)

        # Convert the label vocabulary set to a list so we can access labels via indices
        self.label_vocab = list(label_vocab)

        # Store the sizes of the review and label vocabularies.
        self.review_vocab_size = len(self.review_vocab)
        self.label_vocab_size = len(self.label_vocab)

        # Create a dictionary of words in the vocabulary mapped to index positions
        self.word2index = {}
        for i, word in enumerate(self.review_vocab):
            self.word2index[word] = i

        # Create a dictionary of labels mapped to index positions
        self.label2index = {}
        for i, label in enumerate(self.label_vocab):
            self.label2index[label] = i

    def init_network(self,input_nodes,hidden_nodes,output_nodes,learning_rate):
        # Set number of nodes in input, hidden and output layers.
        self.input_nodes = input_nodes
        self.hidden_nodes = hidden_nodes
        self.output_nodes = output_nodes

        # Store the learning rate
        self.learning_rate = learning_rate

        # Initialize weights

        # These are the weights between the input layer and the hidden layer.
        self.weights_0_1 = np.zeros((self.input_nodes,self.hidden_nodes))

        # These are the weights between the hidden layer and the output layer.
        self.weights_1_2 = np.random.normal(0.0, self.output_nodes**-0.5,
                                                (self.hidden_nodes, self.output_nodes))

        ## New for Project 5: Removed self.layer_0; added self.layer_1
        # The input layer, a two-dimensional matrix with shape 1 x hidden_nodes
        self.layer_1 = np.zeros((1,hidden_nodes))
```

```
    ## New for Project 5: Removed update_input_layer function

    def get_target_for_label(self,label):
        if(label == 'POSITIVE'):
            return 1
        else:
            return 0

    def sigmoid(self,x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_output_2_derivative(self,output):
        return output * (1 - output)

    ## New for Project 5: changed name of first parameter form 'training_reviews'
    #                     to 'training_reviews_raw'
    def train(self, training_reviews_raw, training_labels):

        ## New for Project 5: pre-process training reviews so we can deal
        #                     directly with the indices of non-zero inputs
        training_reviews = list()
        for review in training_reviews_raw:
            indices = set()
            for word in review.split(" "):
                if(word in self.word2index.keys()):
                    indices.add(self.word2index[word])
            training_reviews.append(list(indices))

        # make sure out we have a matching number of reviews and labels
        assert(len(training_reviews) == len(training_labels))

        # Keep track of correct predictions to display accuracy during training
        correct_so_far = 0

        # Remember when we started for printing time statistics
        start = time.time()

        # loop through all the given reviews and run a forward and backward pass,
        # updating weights for every item
        for i in range(len(training_reviews)):

            # Get the next review and its correct label
            review = training_reviews[i]
            label = training_labels[i]

            #### Implement the forward pass here ####
            ### Forward pass ###

            ## New for Project 5: Removed call to 'update_input_layer' function
            #                     because 'layer_0' is no longer used
```

```
            # Hidden layer
            ## New for Project 5: Add in only the weights for non-zero items
            self.layer_1 *= 0
            for index in review:
                self.layer_1 += self.weights_0_1[index]

            # Output layer
            ## New for Project 5: changed to use 'self.layer_1' instead of 'local layer_1'
            layer_2 = self.sigmoid(self.layer_1.dot(self.weights_1_2))

            #### Implement the backward pass here ####
            ### Backward pass ###

            # Output error
            layer_2_error = layer_2 - self.get_target_for_label(label) # Output
layer error is the difference between desired target and actual output.
            layer_2_delta = layer_2_error *
self.sigmoid_output_2_derivative(layer_2)

            # Backpropagated error
            layer_1_error = layer_2_delta.dot(self.weights_1_2.T) # errors
propagated to the hidden layer
            layer_1_delta = layer_1_error # hidden layer gradients - no
nonlinearity so it's the same as the error

            # Update the weights
            ## New for Project 5: changed to use 'self.layer_1' instead of local 'layer_1'
            self.weights_1_2 -= self.layer_1.T.dot(layer_2_delta) *
self.learning_rate # update hidden-to-output weights with gradient descent step

            ## New for Project 5: Only update the weights that were used in the forward pass
            for index in review:
                self.weights_0_1[index] -= layer_1_delta[0] *
self.learning_rate # update input-to-hidden weights with gradient descent step

            # Keep track of correct predictions.
            if(layer_2 >= 0.5 and label == 'POSITIVE'):
                correct_so_far += 1
            elif(layer_2 < 0.5 and label == 'NEGATIVE'):
                correct_so_far += 1

            # For debug purposes, print out our prediction accuracy and speed
            # throughout the training process.
            elapsed_time = float(time.time() - start)
            reviews_per_second = i / elapsed_time if elapsed_time > 0 else 0

            sys.stdout.write("\rProgress:" + str(100 * i/float(len(training_reviews)))[:4] \
                             + "% Speed(reviews/sec):" + str(reviews_per_second)[0:5] \
                             + " #Correct:" + str(correct_so_far) + " #Trained:" + str(i+1) \
                             + " Training Accuracy:" +str(correct_so_far*100/float(i+1))[:4] + "%")
            if(i % 2500 == 0):
                print("")
```

```
    def test(self, testing_reviews, testing_labels):
        """
        Attempts to predict the labels for the given testing_reviews,
        and uses the test_labels to calculate the accuracy of those predictions.
        """

        # keep track of how many correct predictions we make
        correct = 0

        # we'll time how many predictions per second we make
        start = time.time()

        # Loop through each of the given reviews and call run to predict
        # its label.
        for i in range(len(testing_reviews)):
            pred = self.run(testing_reviews[i])
            if(pred == testing_labels[i]):
                correct += 1

            # For debug purposes, print out our prediction accuracy and speed
            # throughout the prediction process.

            elapsed_time = float(time.time() - start)
            reviews_per_second = i / elapsed_time if elapsed_time > 0 else 0

            sys.stdout.write("\rProgress:" + str(100 * i/float(len(testing_reviews)))[:4] \
                             + "% Speed(reviews/sec):" + str(reviews_per_second)[0:5] \
                             + " #Correct:" + str(correct) + " #Tested:" + str(i+1) \
                             + " Testing Accuracy:" + str(correct * 100 / float(i+1))[:4] + "%")

    def run(self, review):
        """
        Returns a POSITIVE or NEGATIVE prediction for the given review.
        """
        # Run a forward pass through the network, like in the "train" function.

        ## New for Project 5: Removed call to update_input_layer function
        #                     because layer_0 is no longer used

        # Hidden layer
        ## New for Project 5: Identify the indices used in the review and then add
        #                     just those weights to layer_1
        self.layer_1 *= 0
        unique_indices = set()
        for word in review.lower().split(" "):
            if word in self.word2index.keys():
                unique_indices.add(self.word2index[word])
        for index in unique_indices:
            self.layer_1 += self.weights_0_1[index]

        # Output layer
        ## New for Project 5: changed to use self.layer_1 instead of local layer_1
        layer_2 = self.sigmoid(self.layer_1.dot(self.weights_1_2))
```

```
        # Return POSITIVE for values above greater-than-or-equal-to 0.5 in the output layer;
        # return NEGATIVE for other values
        if(layer_2[0] >= 0.5):
            return "POSITIVE"
        else:
            return "NEGATIVE"
```

Run the following cell to train your network with a small polarity cutoff.

In [73]:

```
mlp = SentimentNetwork(reviews[:-1000],labels[:-1000],min_count=20,polarity_cutoff=0.05,learning_rate=0.01)
mlp.train(reviews[:-1000],labels[:-1000])
```

```
Progress:0.0% Speed(reviews/sec):0.0 #Correct:1 #Trained:1 Training Accuracy:100.%
Progress:10.4% Speed(reviews/sec):1307. #Correct:1994 #Trained:2501 Training Accuracy:79.7%
......
Progress:99.9% Speed(reviews/sec):1275. #Correct:20461 #Trained:24000 Training Accuracy:85.2%
```

And run the following cell to test it's performance.

In [74]:

```
mlp.test(reviews[-1000:],labels[-1000:])
```

```
Progress:99.9% Speed(reviews/sec):1903. #Correct:859 #Tested:1000 Testing Accuracy:85.9%
```

Run the following cell to train your network with a much larger polarity cutoff.

In [75]:

```
mlp = SentimentNetwork(reviews[:-1000],labels[:-1000],min_count=20,polarity_cutoff=0.8,learning_rate=0.01)
mlp.train(reviews[:-1000],labels[:-1000])
```

```
Progress:0.0% Speed(reviews/sec):0.0 #Correct:1 #Trained:1 Training Accuracy:100.%
Progress:10.4% Speed(reviews/sec):6770. #Correct:2114 #Trained:2501 Training Accuracy:84.5%
Progress:20.8% Speed(reviews/sec):6416. #Correct:4235 #Trained:5001 Training Accuracy:84.6%
......
Progress:93.7% Speed(reviews/sec):6403. #Correct:19258 #Trained:22501 Training Accuracy:85.5%
Progress:99.9% Speed(reviews/sec):6424. #Correct:20552 #Trained:24000 Training Accuracy:85.6%
```

And run the following cell to test it's performance.

In [76]:

```
mlp.test(reviews[-1000:],labels[-1000:])
```

```
Progress:99.9% Speed(reviews/sec):6031. #Correct:822 #Tested:1000 Testing Accuracy:82.2%
```

End of Project 6 solution.

Watch the next video to continue with Andrew's next lesson.

Analysis: What's Going on in the Weights?

In [77]:

```
mlp_full = SentimentNetwork(reviews[:-1000],labels[:-
1000],min_count=0,polarity_cutoff=0,learning_rate=0.01)
```

In [78]:

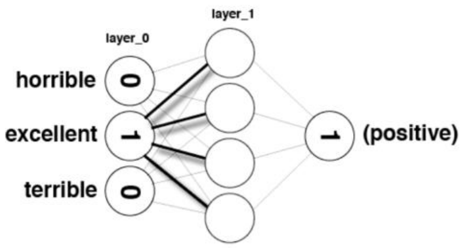
```
mlp_full.train(reviews[:-1000],labels[:-1000])
```

```
Progress:0.0% Speed(reviews/sec):0.0 #Correct:1 #Trained:1 Training
Accuracy:100.%
Progress:10.4% Speed(reviews/sec):1063. #Correct:1962 #Trained:2501 Training
Accuracy:78.4%
……
Progress:99.9% Speed(reviews/sec):1039. #Correct:20335 #Trained:24000 Training
Accuracy:84.7%
```

In [79]:

```
Image(filename='sentiment_network_sparse.png')
```

Out[79]:



In [80]:

```
def get_most_similar_words(focus = "horrible"):
    most_similar = Counter()

    for word in mlp_full.word2index.keys():
        most_similar[word] =
np.dot(mlp_full.weights_0_1[mlp_full.word2index[word]],mlp_full.weights_0_1[mlp
_full.word2index[focus]])

    return most_similar.most_common()
```

In [81]:

```
get_most_similar_words("excellent")
```

Out[81]:

```
[('excellent', 0.13672950757352467),
 ('perfect', 0.12548286087225943),
…….
('overwhelmed', 0.012306633138869476),
 ...]
```

In [82]:

```
get_most_similar_words("terrible")
```

Out[82]:

```
[('worst', 0.16966107259049848),
 ('awful', 0.12026847019691246),
……
('graces', 0.0092620944963291325),
 ...]
```

In [83]:

```
import matplotlib.colors as colors

words_to_visualize = list()
for word, ratio in pos_neg_ratios.most_common(500):
    if(word in mlp_full.word2index.keys()):
        words_to_visualize.append(word)

for word, ratio in list(reversed(pos_neg_ratios.most_common()))[0:500]:
    if(word in mlp_full.word2index.keys()):
        words_to_visualize.append(word)
```

In [84]:

```
pos = 0
neg = 0

colors_list = list()
vectors_list = list()
for word in words_to_visualize:
    if word in pos_neg_ratios.keys():
        vectors_list.append(mlp_full.weights_0_1[mlp_full.word2index[word]])
        if(pos_neg_ratios[word] > 0):
            pos+=1
            colors_list.append("#00ff00")
        else:
            neg+=1
            colors_list.append("#000000")
```

In [85]:

```
from sklearn.manifold import TSNE
tsne = TSNE(n_components=2, random_state=0)
words_top_ted_tsne = tsne.fit_transform(vectors_list)
```

In [87]:

```
p = figure(tools="pan,wheel_zoom,reset,save",
           toolbar_location="above",
           title="vector T-SNE for most polarized words")

source = ColumnDataSource(data=dict(x1=words_top_ted_tsne[:,0],
                                    x2=words_top_ted_tsne[:,1],
                                    names=words_to_visualize))

p.scatter(x="x1", y="x2", size=8, source=source,color=colors_list)

word_labels = LabelSet(x="x1", y="x2", text="names", y_offset=6,
                  text_font_size="8pt", text_color="#555555",
                  source=source, text_align='center')
p.add_layout(word_labels)

show(p)

# green indicates positive words, black indicates negative words
```

```
/anaconda3/envs/sentiment/lib/python3.6/site-
packages/bokeh/util/deprecation.py:34: BokehDeprecationWarning:
Supplying a user-defined data source AND iterable values to glyph methods is
deprecated.

See https://github.com/bokeh/bokeh/issues/2056 for more information.

  warn(message)
/anaconda3/envs/sentiment/lib/python3.6/site-
packages/bokeh/util/deprecation.py:34: BokehDeprecationWarning:
Supplying a user-defined data source AND iterable values to glyph methods is
deprecated.

See https://github.com/bokeh/bokeh/issues/2056 for more information.

  warn(message)
```