Entailment with TensorFlow

March 22, 2018

We'll start by doing all necessary imports, and we'll let our Jupyter Notebook know that it should display graphs and images in the notebook itself.

```
In [2]: %matplotlib inline

        import tensorflow as tf
        import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib.ticker as ticker
        import urllib
        import sys
        import os
        import zipfile
```

The files we're about to use may take five minutes or more to download, so if you're following along by running the program in the corresponding notebook, feel free to start running the next few cells. In the meantime, let's explore textual entailment in further detail.

```
In [3]: glove_zip_file = "glove.6B.zip"
        glove_vectors_file = "glove.6B.50d.txt"

        snli_zip_file = "snli_1.0.zip"
        snli_dev_file = "snli_1.0_dev.txt"
        snli_full_dataset_file = "snli_1.0_train.txt"
```

```
In [4]: from six.moves.urllib.request import urlretrieve

        #large file - 862 MB
        if (not os.path.isfile(glove_zip_file) and
            not os.path.isfile(glove_vectors_file)):
            urlretrieve ("http://nlp.stanford.edu/data/glove.6B.zip",
                         glove_zip_file)

        #medium-sized file - 94.6 MB
        if (not os.path.isfile(snli_zip_file) and
            not os.path.isfile(snli_dev_file)):
            urlretrieve ("https://nlp.stanford.edu/projects/snli/snli_1.0.zip",
                         snli_zip_file)
```

```
In [5]: def unzip_single_file(zip_file_name, output_file_name):
            """
                If the outFile is already created, don't recreate
                If the outFile does not exist, create it from the zipFile
            """
            if not os.path.isfile(output_file_name):
                with open(output_file_name, 'wb') as out_file:
                    with zipfile.ZipFile(zip_file_name) as zipped:
                        for info in zipped.infolist():
                            if output_file_name in info.filename:
                                with zipped.open(info) as requested_file:
                                    out_file.write(requested_file.read())
                                    return

        unzip_single_file(glove_zip_file, glove_vectors_file)
        unzip_single_file(snli_zip_file, snli_dev_file)
        # unzip_single_file(snli_zip_file, snli_full_dataset_file)
```

Now that we have our GloVe vectors downloaded, we can load them into memory, deserializing the space separated format into a Python dictionary:

```
In [6]: glove_wordmap = {}
        with open(glove_vectors_file, "r") as glove:
            for line in glove:
                name, vector = tuple(line.split(" ", 1))
                glove_wordmap[name] = np.fromstring(vector, sep=" ")
```

Once we have our words, we need our input to contain entire sentences and process it through a neural network. Let's start with making the sequence:

```
In [7]: def sentence2sequence(sentence):
            """

            - Turns an input sentence into an (n,d) matrix,
                where n is the number of tokens in the sentence
                and d is the number of dimensions each word vector has.

              Tensorflow doesn't need to be used here, as simply
              turning the sentence into a sequence based off our
              mapping does not need the computational power that
              Tensorflow provides. Normal Python suffices for this task.
            """
            tokens = sentence.lower().split(" ")
            rows = []
            words = []
```

```
        #Greedy search for tokens
        for token in tokens:
            i = len(token)
            while len(token) > 0 and i > 0:
                word = token[:i]
                if word in glove_wordmap:
                    rows.append(glove_wordmap[word])
                    words.append(word)
                    token = token[i:]
                    i = len(token)
                else:
                    i = i-1
        return rows, words
```

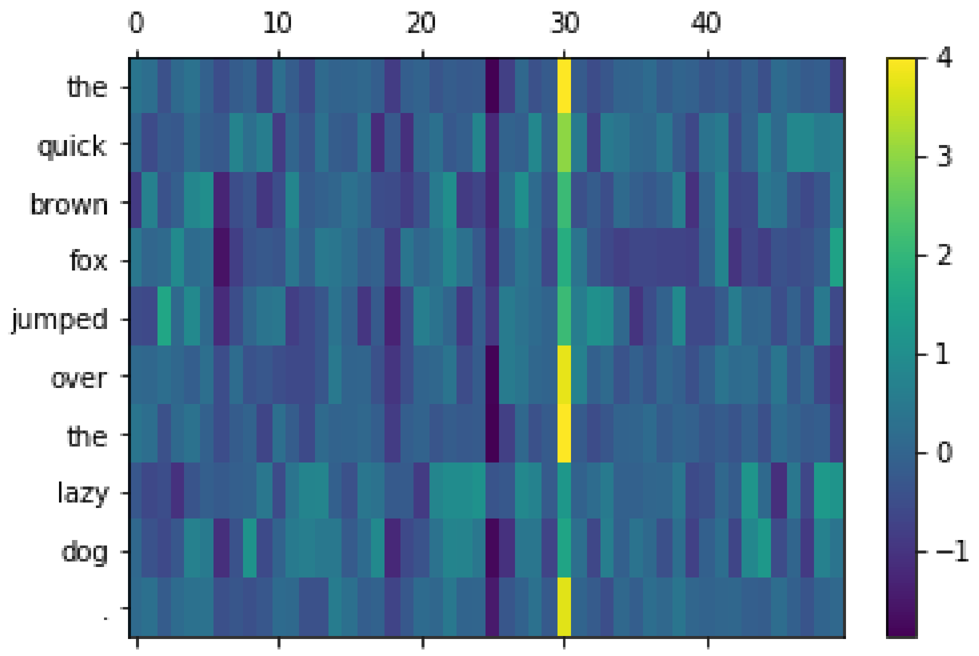
To better visualize the word vectorization process, and to see what the computer sees when it looks at a sentence, we can represent the vectors as images. Each row represents a single word, and the columns represent individual dimensions of the vectorized word. The vectorizations are trained in terms of relationships to other words, and so what the representations actually mean is ambiguous. The computer can understand this vector language, and that's the most important part to us. Generally speaking, two vectors that contain similar colors in the same positions represent words that are similar in meaning.

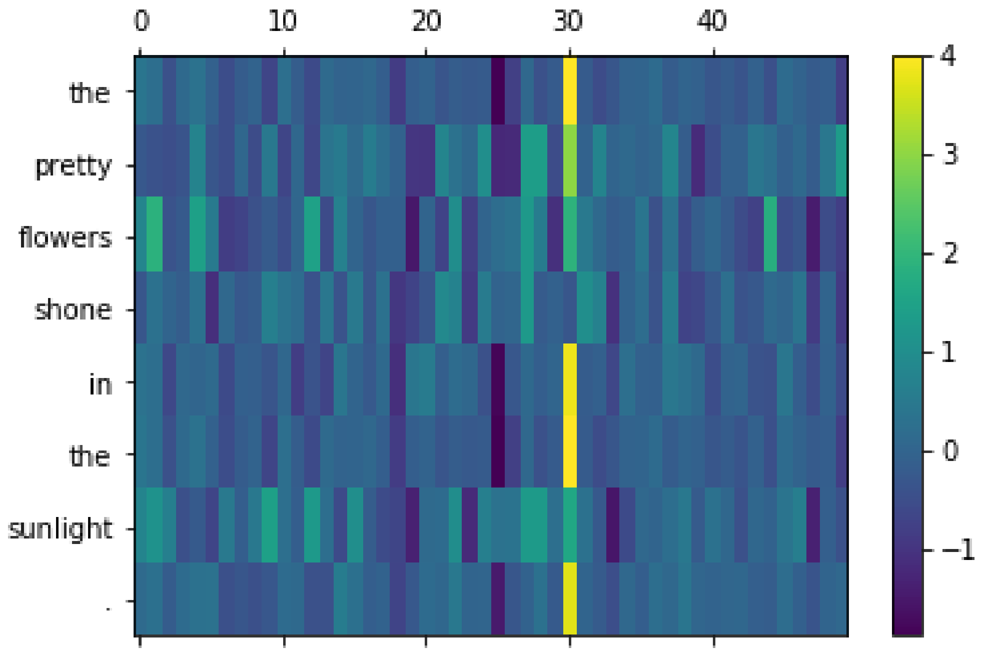
```
In [8]: def visualize(sentence):
            rows, words = sentence2sequence(sentence)
            mat = np.vstack(rows)

            fig = plt.figure()
            ax = fig.add_subplot(111)
            shown = ax.matshow(mat, aspect="auto")
            ax.yaxis.set_major_locator(ticker.MultipleLocator(1))
            fig.colorbar(shown)

            ax.set_yticklabels([""]+words)
            plt.show()

        visualize("The quick brown fox jumped over the lazy dog.")
        visualize("The pretty flowers shone in the sunlight.")
```





Recurrent Neural Networks (also known as RNNs) are a sequence-learning tool for neural networks. This type of neural network has only one layer's worth of hidden inputs, which is re-used for each input from the sequence, along with a "memory" that's passed ahead to the next input's calculations. These are calculated using matrix multiplication where the matrix indices are trained weights, just like they are in a fully-connected layer.

The same calculations are repeated for each input in the sequence, meaning that a single "layer" of a recurrent neural network can be unrolled into many layers. In fact, there will be as many layers as there are inputs in the sequence. This allows the network to process a very complex sentence. TensorFlow includes its own implementation of a vanilla RNN cell, BasicRNNCell, which can be added to your TensorFlow graph as follows:

```
In [9]: rnn_size = 64
        rnn = tf.contrib.rnn.BasicRNNCell(rnn_size)
```

---

1 Defining the constants for our network

Since we aren't going to use a vanilla RNN layer in our network, let's clear out the graph and add an LSTM layer, which TensorFlow also includes by default. Since this is going to be the first part of our actual network, let's also define all the constants we'll need for the network, which we'll talk about as they come up:

```
In [10]: #Constants setup
         max_hypothesis_length, max_evidence_length = 30, 30
         batch_size, vector_size, hidden_size = 128, 50, 64

         lstm_size = hidden_size

         weight_decay = 0.0001

         learning_rate = 1

         input_p, output_p = 0.5, 0.5

         training_iterations_count = 100000

         display_step = 10

         def score_setup(row):
             convert_dict = {
               'entailment': 0,
               'neutral': 1,
               'contradiction': 2
             }
             score = np.zeros((3,))
             for x in range(1,6):
                 tag = row["label"+str(x)]
                 if tag in convert_dict: score[convert_dict[tag]] += 1
             return score / (1.0*np.sum(score))
```

```
        def fit_to_size(matrix, shape):
            res = np.zeros(shape)
            slices = [slice(0,min(dim,shape[e])) for e, dim in enumerate(matrix.shape)]
            res[slices] = matrix[slices]
            return res
```

```
In [11]: def split_data_into_scores():
             import csv
             with open("snli_1.0_dev.txt","r") as data:
                 train = csv.DictReader(data, delimiter='\t')
                 evi_sentences = []
                 hyp_sentences = []
                 labels = []
                 scores = []
                 for row in train:
                     hyp_sentences.append(np.vstack(
                             sentence2sequence(row["sentence1"].lower())[0]))
                     evi_sentences.append(np.vstack(
                             sentence2sequence(row["sentence2"].lower())[0]))
                     labels.append(row["gold_label"])
                     scores.append(score_setup(row))

                 hyp_sentences = np.stack([fit_to_size(x, (max_hypothesis_length, vector_size))
                                   for x in hyp_sentences])
                 evi_sentences = np.stack([fit_to_size(x, (max_evidence_length, vector_size))
                                   for x in evi_sentences])

                 return (hyp_sentences, evi_sentences), labels, np.array(scores)

         data_feature_list, correct_values, correct_scores = split_data_into_scores()

         l_h, l_e = max_hypothesis_length, max_evidence_length
         N, D, H = batch_size, vector_size, hidden_size
         l_seq = l_h + l_e
```

We'll also reset the graph to not include the RNN cell we added earlier, since we won't be using that for this network:

```
In [12]: tf.reset_default_graph()
```

With both those out of the way, we can define our LSTM using TensorFlow as follows:

```
In [13]: lstm = tf.contrib.rnn.BasicLSTMCell(lstm_size)
```

The loss of certain pieces of crucial memory means that complicated relationships required for first order logic have a harder time forming with dropout, and so for our LSTM layer we'll skip using dropout on internal gates, instead using it on everything else. Thankfully, this is the default implementation of Tensorflow's DropoutWrapper for recurrent layers:

```
In [14]: lstm_drop =  tf.contrib.rnn.DropoutWrapper(lstm, input_p, output_p)
```

With all the explanations out of the way, we can finish up our model. The first step is tokenizing and using our GloVe dictionary to turn the two input sentences into a single sequence of vectors. Since we can't effectively use dropout on information that gets passed within an LSTM, we'll use dropout on features from words, and on final output instead -- effectively using dropout on the first and last layers from the unrolled LSTM network portions.

The final output from the LSTMs will be passed into a set of fully connected layers, and then from that we'll get a single real-valued score that indicates how strong each of the kinds of entailment are, which we use to select our final result and our confidence in that result.

```
In [15]: # N: The number of elements in each of our batches,
         #   which we use to train subsets of data for efficiency's sake.
         # l_h: The maximum length of a hypothesis, or the second sentence.  This is
         #   used because training an RNN is extraordinarily difficult without
         #   rolling it out to a fixed length.
         # l_e: The maximum length of evidence, the first sentence.  This is used
         #   because training an RNN is extraordinarily difficult without
         #   rolling it out to a fixed length.
         # D: The size of our used GloVe or other vectors.
         hyp = tf.placeholder(tf.float32, [N, l_h, D], 'hypothesis')
         evi = tf.placeholder(tf.float32, [N, l_e, D], 'evidence')
         y = tf.placeholder(tf.float32, [N, 3], 'label')
         # hyp: Where the hypotheses will be stored during training.
         # evi: Where the evidences will be stored during training.
         # y: Where correct scores will be stored during training.

         # lstm_size: the size of the gates in the LSTM,
         #    as in the first LSTM layer's initialization.
         lstm_back = tf.contrib.rnn.BasicLSTMCell(lstm_size)
         # lstm_back:  The LSTM used for looking backwards
         #   through the sentences, similar to lstm.

         # input_p: the probability that inputs to the LSTM will be retained at each
         #   iteration of dropout.
         # output_p: the probability that outputs from the LSTM will be retained at
         #   each iteration of dropout.
         lstm_drop_back = tf.contrib.rnn.DropoutWrapper(lstm_back, input_p, output_p)
         # lstm_drop_back:  A dropout wrapper for lstm_back, like lstm_drop.


         fc_initializer = tf.random_normal_initializer(stddev=0.1)
         # fc_initializer: initial values for the fully connected layer's weights.
         # hidden_size: the size of the outputs from each lstm layer.
         #   Multiplied by 2 to account for the two LSTMs.
         fc_weight = tf.get_variable('fc_weight', [2*hidden_size, 3],
                                     initializer = fc_initializer)
         # fc_weight: Storage for the fully connected layer's weights.
         fc_bias = tf.get_variable('bias', [3])
         # fc_bias: Storage for the fully connected layer's bias.

         # tf.GraphKeys.REGULARIZATION_LOSSES:  A key to a collection in the graph
         #   designated for losses due to regularization.
         #   In this case, this portion of loss is regularization on the weights
         #   for the fully connected layer.
         tf.add_to_collection(tf.GraphKeys.REGULARIZATION_LOSSES,
                              tf.nn.l2_loss(fc_weight))

         x = tf.concat([hyp, evi], 1) # N, (Lh+Le), d
         # Permuting batch_size and n_steps
         x = tf.transpose(x, [1, 0, 2]) # (Le+Lh), N, d
         # Reshaping to (n_steps*batch_size, n_input)
         x = tf.reshape(x, [-1, vector_size]) # (Le+Lh)*N, d
         # Split to get a list of 'n_steps' tensors of shape (batch_size, n_input)
         x = tf.split(x, l_seq,)

         # x: the inputs to the bidirectional_rnn


         # tf.contrib.rnn.static_bidirectional_rnn: Runs the input through
         #   two recurrent networks, one that runs the inputs forward and one
         #   that runs the inputs in reversed order, combining the outputs.
         rnn_outputs, _, _ = tf.contrib.rnn.static_bidirectional_rnn(lstm, lstm_back,
                                                                     x, dtype=tf.float32)
         # rnn_outputs: the list of LSTM outputs, as a list.
         #   What we want is the latest output, rnn_outputs[-1]

         classification_scores = tf.matmul(rnn_outputs[-1], fc_weight) + fc_bias
         # The scores are relative certainties for how likely the output matches
         #   a certain entailment:
         #     0: Positive entailment
         #     1: Neutral entailment
         #     2: Negative entailment
```

In order to test the accuracy and begin to add in optimization constraints, we need to show TensorFlow how to calculate the accuracy, or -- the percentage of correctly predicted labels.

We also need to determine a loss, to show how poorly the network is doing. Since we have both classification scores and optimal scores, the choice here is using a variation on softmax loss from Tensorflow: tf.nn.softmax_cross_entropy_with_logits. We add in regularization losses to help with overfitting, and then prepare an optimizer to learn how to reduce the loss.

```
In [16]: with tf.variable_scope('Accuracy'):
             predicts = tf.cast(tf.argmax(classification_scores, 1), 'int32')
             y_label = tf.cast(tf.argmax(y, 1), 'int32')
```

```
        corrects = tf.equal(predicts, y_label)
        num_corrects = tf.reduce_sum(tf.cast(corrects, tf.float32))
        accuracy = tf.reduce_mean(tf.cast(corrects, tf.float32))

    with tf.variable_scope("loss"):
        cross_entropy = tf.nn.softmax_cross_entropy_with_logits(
            logits = classification_scores, labels = y)
        loss = tf.reduce_mean(cross_entropy)
        total_loss = loss + weight_decay * tf.add_n(
            tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES))

    optimizer = tf.train.GradientDescentOptimizer(learning_rate)

    opt_op = optimizer.minimize(total_loss)
```

Finally, we can train the network! If you installed TQDM, you can use it to keep track of progress as the network trains.

```
In [17]: # Initialize variables
         init = tf.global_variables_initializer()

         # Use TQDM if installed
         tqdm_installed = False
         try:
             from tqdm import tqdm
             tqdm_installed = True
         except:
             pass

         # Launch the Tensorflow session
         sess = tf.Session()
         sess.run(init)

         # training_iterations_count: The number of data pieces to train on in total
         # batch_size: The number of data pieces per batch
         training_iterations = range(0,training_iterations_count,batch_size)
         if tqdm_installed:
             # Add a progress bar if TQDM is installed
             training_iterations = tqdm(training_iterations)

         for i in training_iterations:

             # Select indices for a random data subset
             batch = np.random.randint(data_feature_list[0].shape[0], size=batch_size)

             # Use the selected subset indices to initialize the graph's
             #   placeholder values
             hyps, evis, ys = (data_feature_list[0][batch,:],
                               data_feature_list[1][batch,:],
                               correct_scores[batch])

             # Run the optimization with these initialized values
             sess.run([opt_op], feed_dict={hyp: hyps, evi: evis, y: ys})
             # display_step: how often the accuracy and loss should
             #   be tested and displayed.
             if (i/batch_size) % display_step == 0:
                 # Calculate batch accuracy
                 acc = sess.run(accuracy, feed_dict={hyp: hyps, evi: evis, y: ys})
                 # Calculate batch loss
                 tmp_loss = sess.run(loss, feed_dict={hyp: hyps, evi: evis, y: ys})
                 # Display results
                 print("Iter " + str(i/batch_size) + ", Minibatch Loss= " + \
                       "{:.6f}".format(tmp_loss) + ", Training Accuracy= " + \
                       "{:.5f}".format(acc))
```

```
  1%|          | 7/782 [00:02<04:39,  2.78it/s]

Iter 0.0, Minibatch Loss= 1.105667, Training Accuracy= 0.37500
Iter 10.0, Minibatch Loss= 1.086592, Training Accuracy= 0.39062


  4%|         | 32/782 [00:02<01:09, 10.79it/s]

Iter 20.0, Minibatch Loss= 1.083417, Training Accuracy= 0.37500
Iter 30.0, Minibatch Loss= 1.085873, Training Accuracy= 0.41406


  7%|         | 51/782 [00:03<00:47, 15.40it/s]

Iter 40.0, Minibatch Loss= 1.083049, Training Accuracy= 0.44531
Iter 50.0, Minibatch Loss= 1.095345, Training Accuracy= 0.35938


  9%|         | 71/782 [00:03<00:36, 19.39it/s]

Iter 60.0, Minibatch Loss= 1.090816, Training Accuracy= 0.41406
Iter 70.0, Minibatch Loss= 1.074096, Training Accuracy= 0.43750


 12%|        | 91/782 [00:04<00:30, 22.66it/s]

Iter 80.0, Minibatch Loss= 1.082164, Training Accuracy= 0.44531
Iter 90.0, Minibatch Loss= 1.060746, Training Accuracy= 0.52344


 14%|        | 111/782 [00:04<00:26, 25.42it/s]
```

```
Iter 100.0, Minibatch Loss= 1.085269, Training Accuracy= 0.45312
Iter 110.0, Minibatch Loss= 1.076293, Training Accuracy= 0.41406


 17%|        | 131/782 [00:04<00:23, 27.79it/s]

Iter 120.0, Minibatch Loss= 1.076493, Training Accuracy= 0.32031
Iter 130.0, Minibatch Loss= 1.070149, Training Accuracy= 0.47656


 19%|        | 151/782 [00:05<00:21, 29.87it/s]

Iter 140.0, Minibatch Loss= 1.076749, Training Accuracy= 0.46875
Iter 150.0, Minibatch Loss= 1.066136, Training Accuracy= 0.40625


 22%|       | 171/782 [00:05<00:19, 31.70it/s]

Iter 160.0, Minibatch Loss= 1.081707, Training Accuracy= 0.42969
Iter 170.0, Minibatch Loss= 1.068961, Training Accuracy= 0.42969


 24%|       | 191/782 [00:05<00:17, 33.31it/s]

Iter 180.0, Minibatch Loss= 1.077067, Training Accuracy= 0.42188
Iter 190.0, Minibatch Loss= 1.094476, Training Accuracy= 0.36719


 27%|       | 211/782 [00:06<00:16, 34.73it/s]

Iter 200.0, Minibatch Loss= 1.081669, Training Accuracy= 0.41406
Iter 210.0, Minibatch Loss= 1.092131, Training Accuracy= 0.35938


 30%|       | 231/782 [00:06<00:15, 36.02it/s]

Iter 220.0, Minibatch Loss= 1.038646, Training Accuracy= 0.56250
Iter 230.0, Minibatch Loss= 1.061287, Training Accuracy= 0.50000


 32%|      | 251/782 [00:06<00:14, 37.14it/s]

Iter 240.0, Minibatch Loss= 1.018567, Training Accuracy= 0.51562
Iter 250.0, Minibatch Loss= 1.069479, Training Accuracy= 0.43750


 35%|      | 271/782 [00:07<00:13, 38.19it/s]

Iter 260.0, Minibatch Loss= 1.044998, Training Accuracy= 0.50000
Iter 270.0, Minibatch Loss= 1.036998, Training Accuracy= 0.44531
```

```
 37%|      | 291/782 [00:07<00:12, 39.12it/s]

Iter 280.0, Minibatch Loss= 1.072192, Training Accuracy= 0.42969
Iter 290.0, Minibatch Loss= 1.057323, Training Accuracy= 0.44531


 40%|      | 311/782 [00:07<00:11, 39.98it/s]

Iter 300.0, Minibatch Loss= 1.027099, Training Accuracy= 0.49219
Iter 310.0, Minibatch Loss= 1.049477, Training Accuracy= 0.46875


 42%|     | 331/782 [00:08<00:11, 40.75it/s]

Iter 320.0, Minibatch Loss= 1.075351, Training Accuracy= 0.45312
Iter 330.0, Minibatch Loss= 1.068570, Training Accuracy= 0.39844


 45%|     | 351/782 [00:08<00:10, 41.45it/s]

Iter 340.0, Minibatch Loss= 1.059098, Training Accuracy= 0.42969
Iter 350.0, Minibatch Loss= 1.064130, Training Accuracy= 0.47656


 47%|     | 370/782 [00:08<00:09, 42.08it/s]

Iter 360.0, Minibatch Loss= 1.003851, Training Accuracy= 0.53906
Iter 370.0, Minibatch Loss= 1.020049, Training Accuracy= 0.47656


 50%|     | 389/782 [00:09<00:09, 42.67it/s]

Iter 380.0, Minibatch Loss= 1.029562, Training Accuracy= 0.50000
Iter 390.0, Minibatch Loss= 1.060648, Training Accuracy= 0.45312


 52%|    | 408/782 [00:09<00:08, 43.20it/s]

Iter 400.0, Minibatch Loss= 1.105812, Training Accuracy= 0.41406
Iter 410.0, Minibatch Loss= 1.035448, Training Accuracy= 0.52344


 55%|    | 428/782 [00:09<00:08, 43.74it/s]

Iter 420.0, Minibatch Loss= 1.042926, Training Accuracy= 0.53125
Iter 430.0, Minibatch Loss= 1.058548, Training Accuracy= 0.43750


 57%|    | 448/782 [00:10<00:07, 44.23it/s]
```

```
Iter 440.0, Minibatch Loss= 1.074499, Training Accuracy= 0.43750
Iter 450.0, Minibatch Loss= 1.014167, Training Accuracy= 0.50000


 60%|    | 468/782 [00:10<00:07, 44.69it/s]

Iter 460.0, Minibatch Loss= 1.030002, Training Accuracy= 0.48438
Iter 470.0, Minibatch Loss= 1.043793, Training Accuracy= 0.50000


 62%|   | 488/782 [00:10<00:06, 45.11it/s]

Iter 480.0, Minibatch Loss= 1.045442, Training Accuracy= 0.43750
Iter 490.0, Minibatch Loss= 1.055641, Training Accuracy= 0.48438


 65%|   | 508/782 [00:11<00:06, 45.53it/s]

Iter 500.0, Minibatch Loss= 1.043379, Training Accuracy= 0.51562
Iter 510.0, Minibatch Loss= 0.987236, Training Accuracy= 0.55469


 68%|   | 528/782 [00:11<00:05, 45.93it/s]

Iter 520.0, Minibatch Loss= 1.072384, Training Accuracy= 0.39844
Iter 530.0, Minibatch Loss= 1.034984, Training Accuracy= 0.49219


 70%|   | 548/782 [00:11<00:05, 46.31it/s]

Iter 540.0, Minibatch Loss= 1.026031, Training Accuracy= 0.54688
Iter 550.0, Minibatch Loss= 1.067918, Training Accuracy= 0.42969


 73%|  | 568/782 [00:12<00:04, 46.64it/s]

Iter 560.0, Minibatch Loss= 1.029303, Training Accuracy= 0.54688
Iter 570.0, Minibatch Loss= 1.019768, Training Accuracy= 0.50000


 75%|  | 588/782 [00:12<00:04, 46.97it/s]

Iter 580.0, Minibatch Loss= 1.014870, Training Accuracy= 0.56250
Iter 590.0, Minibatch Loss= 1.063535, Training Accuracy= 0.46094


 78%|  | 608/782 [00:12<00:03, 47.28it/s]

Iter 600.0, Minibatch Loss= 1.043036, Training Accuracy= 0.53125
Iter 610.0, Minibatch Loss= 1.036003, Training Accuracy= 0.48438
```

```
 80%|  | 628/782 [00:13<00:03, 47.57it/s]

Iter 620.0, Minibatch Loss= 1.037401, Training Accuracy= 0.50781
Iter 630.0, Minibatch Loss= 0.974858, Training Accuracy= 0.56250


 83%| | 648/782 [00:13<00:02, 47.84it/s]

Iter 640.0, Minibatch Loss= 1.057583, Training Accuracy= 0.46094
Iter 650.0, Minibatch Loss= 1.049396, Training Accuracy= 0.44531


 85%| | 667/782 [00:13<00:02, 48.02it/s]

Iter 660.0, Minibatch Loss= 1.029728, Training Accuracy= 0.46875
Iter 670.0, Minibatch Loss= 1.015780, Training Accuracy= 0.53906


 88%| | 692/782 [00:14<00:01, 48.30it/s]

Iter 680.0, Minibatch Loss= 1.089828, Training Accuracy= 0.40625
Iter 690.0, Minibatch Loss= 0.982862, Training Accuracy= 0.57031


 91%| | 711/782 [00:14<00:01, 48.52it/s]

Iter 700.0, Minibatch Loss= 1.040684, Training Accuracy= 0.49219
Iter 710.0, Minibatch Loss= 1.009991, Training Accuracy= 0.57812


 93%|| 731/782 [00:14<00:01, 48.76it/s]

Iter 720.0, Minibatch Loss= 1.014683, Training Accuracy= 0.57031
Iter 730.0, Minibatch Loss= 1.032561, Training Accuracy= 0.51562


 96%|| 751/782 [00:15<00:00, 48.98it/s]

Iter 740.0, Minibatch Loss= 1.042345, Training Accuracy= 0.52344
Iter 750.0, Minibatch Loss= 1.018030, Training Accuracy= 0.50781


 98%|| 770/782 [00:15<00:00, 49.12it/s]

Iter 760.0, Minibatch Loss= 1.064371, Training Accuracy= 0.46094


100%|| 782/782 [00:15<00:00, 49.20it/s]

Iter 770.0, Minibatch Loss= 1.025236, Training Accuracy= 0.49219
Iter 780.0, Minibatch Loss= 1.001164, Training Accuracy= 0.52344
```

Your network is now trained! You should see accuracies around 50-55%, which can be improved by careful modification of hyperparameters and increasing the dataset size to include the entire training set. Usually, this will correspond with an increase in training time.

Feel free to modify the following code by inserting your own sentences:

```
In [18]: evidences = ["Maurita and Jade both were at the scene of the car crash."]

         hypotheses = ["Multiple people saw the accident."]

         sentence1 = [fit_to_size(np.vstack(sentence2sequence(evidence)[0]),
                                  (30, 50)) for evidence in evidences]

         sentence2 = [fit_to_size(np.vstack(sentence2sequence(hypothesis)[0]),
                                  (30,50)) for hypothesis in hypotheses]

         prediction = sess.run(classification_scores, feed_dict={hyp: (sentence1 * N),
                                                                 evi: (sentence2 * N),
                                                                 y: [[0,0,0]]*N})
         print(["Positive", "Neutral", "Negative"][np.argmax(prediction[0])]+
               " entailment")
```

```
Positive entailment
```

Finally, once we're done playing with our model, we'll close the session to free up system resources.

```
In [19]: sess.close()
```