# Project: Building a Chatbot with TensorFlow using Cornell Movie Dialog Corpus  
  
In this project, we will use conversations from Cornell University’s Movie Dialogue Corpus to build a simple chatbot. The implementation is done in Python, using TensorFlow for building the model.  
  
The focus will be on building the sequence-to-sequence model. For the full project, you can visit the project's [GitHub page](#). Preparing the dataset for the model is a bit of work, so if you're unsure how to do it, or would like suggestions, refer to the code on GitHub.  
  
---  
  
### Why Sequence-to-Sequence Models?  
  
Sequence-to-sequence models have a variety of applications beyond chatbots, such as language translation, text summarization, and text generation.  
  
---  
  
## Model Inputs  
  
The first step is to create placeholders for the model’s inputs:  
  
```python  
def model\_inputs():  
 input\_data = tf.placeholder(tf.int32, [None, None], name='input')  
 targets = tf.placeholder(tf.int32, [None, None], name='targets')  
 lr = tf.placeholder(tf.float32, name='learning\_rate')  
 keep\_prob = tf.placeholder(tf.float32, name='keep\_prob')  
 return input\_data, targets, lr, keep\_prob

We do not specify the shape for learning\_rate and keep\_prob because the default is None.

## Processing Encoding Input

We need to format the encoding inputs for the model. The following function removes the final word from each batch and adds the <GO> token at the start of each batch.

def process\_encoding\_input(target\_data, vocab\_to\_int, batch\_size):  
 ending = tf.strided\_slice(target\_data, [0, 0], [batch\_size, -1], [1, 1])  
 dec\_input = tf.concat([tf.fill([batch\_size, 1], vocab\_to\_int['<GO>']), ending], 1)  
 return dec\_input

## Encoding Layer

This layer encodes the input data using LSTM cells wrapped with dropout. We use a bidirectional LSTM for more effective encoding.

def encoding\_layer(rnn\_inputs, rnn\_size, num\_layers, keep\_prob, sequence\_length):  
 lstm = tf.contrib.rnn.BasicLSTMCell(rnn\_size)  
 drop = tf.contrib.rnn.DropoutWrapper(lstm, input\_keep\_prob=keep\_prob)  
 enc\_cell = tf.contrib.rnn.MultiRNNCell([drop] \* num\_layers)  
 \_, enc\_state = tf.nn.bidirectional\_dynamic\_rnn(cell\_fw=enc\_cell, cell\_bw=enc\_cell, sequence\_length=sequence\_length, inputs=rnn\_inputs, dtype=tf.float32)  
 return enc\_state

## Decoding Layer for Training

For the training process, we use attention mechanisms to improve performance by focusing on the relevant parts of the input.

def decoding\_layer\_train(encoder\_state, dec\_cell, dec\_embed\_input, sequence\_length, decoding\_scope, output\_fn, keep\_prob, batch\_size):  
 attention\_states = tf.zeros([batch\_size, 1, dec\_cell.output\_size])  
 att\_keys, att\_vals, att\_score\_fn, att\_construct\_fn = tf.contrib.seq2seq.prepare\_attention(attention\_states, attention\_option="bahdanau", num\_units=dec\_cell.output\_size)  
 train\_decoder\_fn = tf.contrib.seq2seq.attention\_decoder\_fn\_train(encoder\_state[0], att\_keys, att\_vals, att\_score\_fn, att\_construct\_fn, name="attn\_dec\_train")  
 train\_pred, \_, \_ = tf.contrib.seq2seq.dynamic\_rnn\_decoder(dec\_cell, train\_decoder\_fn, dec\_embed\_input, sequence\_length, scope=decoding\_scope)  
 train\_pred\_drop = tf.nn.dropout(train\_pred, keep\_prob)  
 return output\_fn(train\_pred\_drop)

Using attention reduced our model’s loss by 20% while increasing training time by the same amount, which is a fair trade-off.

## Decoding Layer for Inference

For inference, we use a similar structure but without dropout and with additional parameters for generating predictions.

def decoding\_layer\_infer(encoder\_state, dec\_cell, dec\_embeddings, start\_of\_sequence\_id, end\_of\_sequence\_id, maximum\_length, vocab\_size, decoding\_scope, output\_fn, keep\_prob, batch\_size):  
 attention\_states = tf.zeros([batch\_size, 1, dec\_cell.output\_size])  
 att\_keys, att\_vals, att\_score\_fn, att\_construct\_fn = tf.contrib.seq2seq.prepare\_attention(attention\_states, attention\_option="bahdanau", num\_units=dec\_cell.output\_size)  
 infer\_decoder\_fn = tf.contrib.seq2seq.attention\_decoder\_fn\_inference(output\_fn, encoder\_state[0], att\_keys, att\_vals, att\_score\_fn, att\_construct\_fn, dec\_embeddings, start\_of\_sequence\_id, end\_of\_sequence\_id, maximum\_length, vocab\_size, name="attn\_dec\_inf")  
 infer\_logits, \_, \_ = tf.contrib.seq2seq.dynamic\_rnn\_decoder(dec\_cell, infer\_decoder\_fn, scope=decoding\_scope)  
 return infer\_logits

## Complete Decoding Layer

This combines both the training and inference parts to form the complete decoding process.

def decoding\_layer(dec\_embed\_input, dec\_embeddings, encoder\_state, vocab\_size, sequence\_length, rnn\_size, num\_layers, vocab\_to\_int, keep\_prob, batch\_size):  
 with tf.variable\_scope("decoding") as decoding\_scope:  
 lstm = tf.contrib.rnn.BasicLSTMCell(rnn\_size)  
 drop = tf.contrib.rnn.DropoutWrapper(lstm, input\_keep\_prob=keep\_prob)  
 dec\_cell = tf.contrib.rnn.MultiRNNCell([drop] \* num\_layers)  
 weights = tf.truncated\_normal\_initializer(stddev=0.1)  
 biases = tf.zeros\_initializer()  
 output\_fn = lambda x: tf.contrib.layers.fully\_connected(x, vocab\_size, None, scope=decoding\_scope, weights\_initializer=weights, biases\_initializer=biases)  
 train\_logits = decoding\_layer\_train(encoder\_state, dec\_cell, dec\_embed\_input, sequence\_length, decoding\_scope, output\_fn, keep\_prob, batch\_size)  
 decoding\_scope.reuse\_variables()  
 infer\_logits = decoding\_layer\_infer(encoder\_state, dec\_cell, dec\_embeddings, vocab\_to\_int['<GO>'], vocab\_to\_int['<EOS>'], sequence\_length - 1, vocab\_size, decoding\_scope, output\_fn, keep\_prob, batch\_size)  
 return train\_logits, infer\_logits

## Sequence-to-Sequence Model

This function ties together the encoding and decoding layers.

def seq2seq\_model(input\_data, target\_data, keep\_prob, batch\_size, sequence\_length, answers\_vocab\_size, questions\_vocab\_size, enc\_embedding\_size, dec\_embedding\_size, rnn\_size, num\_layers, questions\_vocab\_to\_int):  
 enc\_embed\_input = tf.contrib.layers.embed\_sequence(input\_data, answers\_vocab\_size+1, enc\_embedding\_size, initializer=tf.random\_uniform\_initializer(-1, 1))  
 enc\_state = encoding\_layer(enc\_embed\_input, rnn\_size, num\_layers, keep\_prob, sequence\_length)  
 dec\_input = process\_encoding\_input(target\_data, questions\_vocab\_to\_int, batch\_size)  
 dec\_embeddings = tf.Variable(tf.random\_uniform([questions\_vocab\_size+1, dec\_embedding\_size], -1, 1))  
 dec\_embed\_input = tf.nn.embedding\_lookup(dec\_embeddings, dec\_input)  
 train\_logits, infer\_logits = decoding\_layer(dec\_embed\_input, dec\_embeddings, enc\_state, questions\_vocab\_size, sequence\_length, rnn\_size, num\_layers, questions\_vocab\_to\_int, keep\_prob, batch\_size)  
 return train\_logits, infer\_logits

## Model Training and Optimization

We define the training process and optimization setup here:

epochs = 100  
batch\_size = 128  
rnn\_size = 512  
num\_layers = 2  
encoding\_embedding\_size = 512  
decoding\_embedding\_size = 512  
learning\_rate = 0.005  
learning\_rate\_decay = 0.9  
min\_learning\_rate = 0.0001  
keep\_probability = 0.75  
  
tf.reset\_default\_graph()  
sess = tf.InteractiveSession()  
  
input\_data, input\_length, targets, lr, keep\_prob = model\_inputs()  
sequence\_length = tf.placeholder\_with\_default(max\_line\_length, None, name='sequence\_length')  
input\_shape = tf.shape(input\_data)  
  
train\_logits, inference\_logits = seq2seq\_model(tf.reverse(input\_data, [-1]), targets, keep\_prob, batch\_size, sequence\_length, len(answers\_vocab\_to\_int), len(questions\_vocab\_to\_int), encoding\_embedding\_size, decoding\_embedding\_size, rnn\_size, num\_layers, questions\_vocab\_to\_int)  
  
with tf.name\_scope("optimization"):  
 cost = tf.contrib.seq2seq.sequence\_loss(train\_logits, targets, tf.ones([input\_shape[0], sequence\_length]))  
 optimizer = tf.train.AdamOptimizer(learning\_rate)  
 gradients = optimizer.compute\_gradients(cost)  
 capped\_gradients = [(tf.clip\_by\_value(grad, -5., 5.), var) for grad, var in gradients if grad is not None]  
 train\_op = optimizer.apply\_gradients(capped\_gradients)

## Making Predictions

Here’s how we generate predictions from the trained model:

random = np.random.choice(len(short\_questions))  
input\_question = short\_questions[random]  
input\_question = question\_to\_seq(input\_question, questions\_vocab\_to\_int)  
input\_question = input\_question + [questions\_vocab\_to\_int["<PAD>"]] \* (max\_line\_length - len(input\_question))  
  
batch\_shell = np.zeros((batch\_size, max\_line\_length))  
batch\_shell[0] = input\_question   
  
answer\_logits = sess.run(inference\_logits, {input\_data: batch\_shell, keep\_prob: 1.0})[0]  
pad\_q = questions\_vocab\_to\_int["<PAD>"]  
pad\_a = answers  
  
\_vocab\_to\_int["<PAD>"]  
  
print("Question:")  
print(" ".join([questions\_int\_to\_vocab[i] for i in input\_question if i != pad\_q]))  
  
print("\nAnswer:")  
print(" ".join([answers\_int\_to\_vocab[i] for i in np.argmax(answer\_logits, 1) if i != pad\_a]))

## Conclusion

This project demonstrates how sequence-to-sequence models, enhanced by attention mechanisms, can be used for building chatbots. Although the model has room for improvement, the performance achieved is reasonable given the size of the dataset and the complexity of the task.

You can experiment with different architectures, or tune the hyperparameters further to improve the model’s performance.

For further exploration and improvements, the full project code and dataset are available on [GitHub](#Xa39a3ee5e6b4b0d3255bfef95601890afd80709). ```

This markdown version is designed for a Jupyter notebook and retains all the essential code and explanations from the project report.