In this assignment we use a PC with the below Hardware configuration:

* CPU: i7-4700HQ @ 2,4GHz
* RAM: 16GB
* GPU: NVIDIA GeForce GT 750M @ 967 MHz

we use the Tensorflow version 1 for the needs of code

the code was run normally without any errors or exceptions.

On the other hand, to run the code as much possible, we the set of the Hyperparameters as the below:

epochs = 1

batch\_size = 1

rnn\_size = 1

num\_layers = 1

encoding\_embedding\_size = 1

decoding\_embedding\_size = 1

learning\_rate = 1

learning\_rate\_decay = 1

min\_learning\_rate = 1

keep\_probability = 1

unfortunately, and after plenty of different configurations, is not able to fully run the code.

Specifically, the code felt in dead kernel in the training questions process and before the question preparation of the model.

Below is following the description of the code.

**Chatbot building**

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

First step, create placeholders for our model’s inputs. You might have noticed that learning\_rate and keep\_prob do not have a shape parameter. This is because the default shape is None, which is what we want, so we can just leave it blank to keep our code concise.

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

tf.strided\_slice() will remove the final word from each batch. Appended to the start of each batch will be the token <GO> . This formatting is necessary for creating the embeddings for our decoding layer.

def encoding\_layer(rnn\_inputs, rnn\_size, num\_layers, keep\_prob,

sequence\_length, attn\_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

This will encode our input data.

* From what I have read, LSTM cells typically outperform GRU cells for seq2seq tasks, such as this one.
* Making the encoder bidirectional proved to be much more effective than a simple feed forward network.
* We return only the encoder’s state because it is the input for our decoding layer. Simply put, the weights of the encoding cells are what interest us.

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 in our decoding layers reduces the loss of our model by about 20% and increases the training time by about 20%. I’d say that it’s a fair trade-off. Some notes to make:

* The model performs best when the attention states are set with zeros.
* The two attention options are bahdanau and luong. Bahdanau is less computationally expensive and better results were achieved with it.

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

decoding\_layer\_infer() is very similar to decoding\_layer\_train(). The main difference is the extra parameters added to attention\_decoder\_fn\_inference() compared to attention\_decoder\_fn\_train(). These extra parameters are necessary to help the model create accurate responses for your input sentences.

There is also no dropout in this function. This is because we are using it to create our responses during testing (aka making predictions), and we want to be using our full network for that.

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

Here we are using the previous two functions, a decoding cell, and a fully connected layer to create our training and inference logits. We are using tf.variable\_scope() to reuse the variables from training for making predictions.

By initializing the weights with a truncated normal distribution and a small standard deviation, this can improve the performance of the model.

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

This is where we tie everything together and generate the outputs for our model.

* Similar to initializing weights and biases, I find it best to initialize my embeddings as well. Rather than using a truncated normal distribution, a random uniform distribution is more appropriate. If you want, you can read more about embeddings from TensorFlow’s tutorial.
* Since we do not have to process our encoding’s inputs, we can use tf.contrib.layers.embed\_sequence() to simplify the code a little.
* If you want to shorten your code a little, you could return decoding\_layer() rather than creating train\_logits & infer\_logits and returning them. I wrote it this way to be more explicit.

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

Here are the parameters that I used. A larger network could produce better results, but given the number of iterations that I performed, I didn’t want to rack up my bill on FloydHub.

Using learning rate decay is always something you should consider. As your model tries to find the optimal weights, it needs to update these values with smaller increments, so a shrinking learning rate is beneficial.

To help you build and improve your model, I highly recommend that you read this research paper. It will provide you with some great insights about how to set your hyperparameters’ values and what the size of your network should be.

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)

I chose to use an interactive session to provide a little more flexibility when building this model, but you can use whatever session type you wish.

Sequence length will be the max line length for each batch. I sorted my inputs by length to reduce the amount of padding when creating the batches. This helped to speed up training.

If you are unfamiliar with seq2seq models, the input is often reversed. This helps a model to produce better outputs because when the input data is being fed into the model, the start of the sequence will now become closer to the start of the output sequence.

Although I have clipped my gradients at ±5, I didn’t notice much of a difference with ±1.

I’m going to skip over creating the batches, padding the batches, and training the model since it’s pretty standard stuff. Below you’ll see how to make predictions with this model.

#input\_question = 'How are you?'

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(' Word Ids: {}'.format(

[i for i in input\_question if i != pad\_q]))

print(' Input Words: {}'.format(

[questions\_int\_to\_vocab[i] for i in input\_question if i != pad\_q]))

print('\nAnswer')

print('Word Ids: {}'.format(

[i for i in np.argmax(answer\_logits, 1) if i != pad\_a]))

print('Response Words: {}'.format(

[answers\_int\_to\_vocab[i] for i in \

np.argmax(answer\_logits, 1) if i != pad\_a]))

I provided the optionality to either input your own questions or use one from the data. I didn’t find the model to be any better at answering a question from either type of input.

For the input question to be used by the model, it needs to be formatted like the training data. This is why padding was added and batch\_shell was created.

If you test out this model, and expand it, or do anything else cool with it, could you please make a comment about it below. I’ll be looking for ways to improve this model and how best to apply it to other seq2seq tasks, and it would be great to see what you come up with!

**References**

David Currie, How to Build Your First Chatbot, 2017, <https://tutorials.botsfloor.com/how-to-build-your-first-chatbot-c84495d4622d>

David Currie, Chatbot-from-Movie-Dialogue, 2017, <https://github.com/Currie32/Chatbot-from-Movie-Dialogue>