

dlnd_language_translation

July 25, 2017

1 Language Translation

In this project, you're going to take a peek into the realm of neural network machine translation. You'll be training a sequence to sequence model on a dataset of English and French sentences that can translate new sentences from English to French. ## Get the Data Since translating the whole language of English to French will take lots of time to train, we have provided you with a small portion of the English corpus.

```
In [1]: """
        DON'T MODIFY ANYTHING IN THIS CELL
        """

import helper
import problem_unittests as tests

source_path = 'data/small_vocab_en'
target_path = 'data/small_vocab_fr'
source_text = helper.load_data(source_path)
target_text = helper.load_data(target_path)
```

1.1 Explore the Data

Play around with `view_sentence_range` to view different parts of the data.

```
In [2]: view_sentence_range = (0, 10)

        """
        DON'T MODIFY ANYTHING IN THIS CELL
        """

import numpy as np

print('Dataset Stats')
print('Roughly the number of unique words: {}'.format(len({word: None for word in source_text.split() if word != None})))

sentences = source_text.split('\n')
word_counts = [len(sentence.split()) for sentence in sentences]
print('Number of sentences: {}'.format(len(sentences)))
print('Average number of words in a sentence: {}'.format(np.average(word_counts)))
```

```

print()
print('English sentences {} to {}'.format(*view_sentence_range))
print('\n'.join(source_text.split('\n')[view_sentence_range[0]:view_sentence_range[1]]))
print()
print('French sentences {} to {}'.format(*view_sentence_range))
print('\n'.join(target_text.split('\n')[view_sentence_range[0]:view_sentence_range[1]]))

```

Dataset Stats

Roughly the number of unique words: 227

Number of sentences: 137861

Average number of words in a sentence: 13.225277634719028

English sentences 0 to 10:

```

new jersey is sometimes quiet during autumn , and it is snowy in april .
the united states is usually chilly during july , and it is usually freezing in november .
california is usually quiet during march , and it is usually hot in june .
the united states is sometimes mild during june , and it is cold in september .
your least liked fruit is the grape , but my least liked is the apple .
his favorite fruit is the orange , but my favorite is the grape .
paris is relaxing during december , but it is usually chilly in july .
new jersey is busy during spring , and it is never hot in march .
our least liked fruit is the lemon , but my least liked is the grape .
the united states is sometimes busy during january , and it is sometimes warm in november .

```

French sentences 0 to 10:

```

new jersey est parfois calme pendant l' automne , et il est neigeux en avril .
les états-unis est généralement froid en juillet , et il gèle habituellement en novembre .
california est généralement calme en mars , et il est généralement chaud en juin .
les états-unis est parfois légère en juin , et il fait froid en septembre .
votre moins aimé fruit est le raisin , mais mon moins aimé est la pomme .
son fruit préféré est l'orange , mais mon préféré est le raisin .
paris est relaxant en décembre , mais il est généralement froid en juillet .
new jersey est occupé au printemps , et il est jamais chaude en mars .
notre fruit est moins aimé le citron , mais mon moins aimé est le raisin .
les états-unis est parfois occupé en janvier , et il est parfois chaud en novembre .

```

1.2 Implement Preprocessing Function

1.2.1 Text to Word Ids

As you did with other RNNs, you must turn the text into a number so the computer can understand it. In the function `text_to_ids()`, you'll turn `source_text` and `target_text` from words to ids. However, you need to add the `<EOS>` word id at the end of `target_text`. This will help the neural network predict when the sentence should end.

You can get the `<EOS>` word id by doing:

```
target_vocab_to_int['<EOS>']
```

You can get other word ids using `source_vocab_to_int` and `target_vocab_to_int`.

```
In [3]: def text_to_ids(source_text, target_text, source_vocab_to_int, target_vocab_to_int):
        """
        Convert source and target text to proper word ids
        :param source_text: String that contains all the source text.
        :param target_text: String that contains all the target text.
        :param source_vocab_to_int: Dictionary to go from the source words to an id
        :param target_vocab_to_int: Dictionary to go from the target words to an id
        :return: A tuple of lists (source_id_text, target_id_text)
        """

        source_id_text = list()
        for line in source_text.split('\n'):
            source_id_text.append([source_vocab_to_int[word] for word in line.split()])

        end_of_sequence = target_vocab_to_int['<EOS>']
        target_id_text = list()
        for line in target_text.split('\n'):
            target_id_text.append([target_vocab_to_int[word] for word in line.split()] + [end_of_sequence])

        return (source_id_text, target_id_text)

        """
        DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
        """

        tests.test_text_to_ids(text_to_ids)
```

Tests Passed

1.2.2 Preprocess all the data and save it

Running the code cell below will preprocess all the data and save it to file.

```
In [4]: """
        DON'T MODIFY ANYTHING IN THIS CELL
        """

        helper.preprocess_and_save_data(source_path, target_path, text_to_ids)
```

2 Check Point

This is your first checkpoint. If you ever decide to come back to this notebook or have to restart the notebook, you can start from here. The preprocessed data has been saved to disk.

```
In [5]: """
        DON'T MODIFY ANYTHING IN THIS CELL
        """
```

```
import numpy as np
import helper
```

```
(source_int_text, target_int_text), (source_vocab_to_int, target_vocab_to_int), _ = help
```

2.0.1 Check the Version of TensorFlow and Access to GPU

This will check to make sure you have the correct version of TensorFlow and access to a GPU

```
In [6]: """
        DON'T MODIFY ANYTHING IN THIS CELL
        """

        from distutils.version import LooseVersion
        import warnings
        import tensorflow as tf
        from tensorflow.python.layers.core import Dense

        # Check TensorFlow Version
        assert LooseVersion(tf.__version__) >= LooseVersion('1.1'), 'Please use TensorFlow versi
        print('TensorFlow Version: {}'.format(tf.__version__))

        # Check for a GPU
        if not tf.test.gpu_device_name():
            warnings.warn('No GPU found. Please use a GPU to train your neural network.')
        else:
            print('Default GPU Device: {}'.format(tf.test.gpu_device_name()))
```

TensorFlow Version: 1.1.0

/home/david/anaconda3/envs/dlnd/lib/python3.5/site-packages/ipykernel/__main__.py:15: UserWarnin

2.1 Build the Neural Network

You'll build the components necessary to build a Sequence-to-Sequence model by implementing the following functions below: - model_inputs - process_decoder_input - encoding_layer - decoding_layer_train - decoding_layer_infer - decoding_layer - seq2seq_model

2.1.1 Input

Implement the model_inputs() function to create TF Placeholders for the Neural Network. It should create the following placeholders:

- Input text placeholder named "input" using the TF Placeholder name parameter with rank 2.
- Targets placeholder with rank 2.
- Learning rate placeholder with rank 0.

- Keep probability placeholder named "keep_prob" using the TF Placeholder name parameter with rank 0.
- Target sequence length placeholder named "target_sequence_length" with rank 1
- Max target sequence length tensor named "max_target_len" getting its value from applying `tf.reduce_max` on the `target_sequence_length` placeholder. Rank 0.
- Source sequence length placeholder named "source_sequence_length" with rank 1

Return the placeholders in the following the tuple (input, targets, learning rate, keep probability, target sequence length, max target sequence length, source sequence length)

```
In [7]: def model_inputs():
        """
        Create TF Placeholders for input, targets, learning rate, and lengths of source and
        :return: Tuple (input, targets, learning rate, keep probability, target sequence length,
        max target sequence length, source sequence length)
        """
        inputs = tf.placeholder(tf.int32, [None, None], name="input")
        targets = tf.placeholder(tf.int32, [None, None], name="targets")
        learning_rate = tf.placeholder(tf.float32, name="learning_rate")
        keep_prob = tf.placeholder(tf.float32, name="keep_prob")

        target_sequence_length = tf.placeholder(tf.int32, shape=[None], name="target_sequence_length")
        max_target_length = tf.reduce_max(target_sequence_length, name="max_target_len")
        source_sequence_length = tf.placeholder(tf.int32, shape=[None], name="source_sequence_length")
        return inputs, targets, learning_rate, keep_prob, target_sequence_length, max_target_length, source_sequence_length

        """
        DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
        """
        tests.test_model_inputs(model_inputs)
```

Tests Passed

2.1.2 Process Decoder Input

Implement `process_decoder_input` by removing the last word id from each batch in `target_data` and concat the GO ID to the beginning of each batch.

```
In [8]: def process_decoder_input(target_data, target_vocab_to_int, batch_size):
        """
        Preprocess target data for encoding
        :param target_data: Target Placeholder
        :param target_vocab_to_int: Dictionary to go from the target words to an id
        :param batch_size: Batch Size
        :return: Preprocessed target data
        """
        removed_end = tf.strided_slice(target_data, [0, 0], [batch_size, -1], [1, 1])
```

```

        target_batch = tf.concat([tf.fill([batch_size, 1], target_vocab_to_int['<GO>']), rem
        return target_batch

    """
    DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
    """

    tests.test_process_encoding_input(process_decoder_input)

```

Tests Passed

2.1.3 Encoding

Implement `encoding_layer()` to create a Encoder RNN layer: * Embed the encoder input using `tf.contrib.layers.embed_sequence` * Construct a `stacked tf.contrib.rnn.LSTMCell` wrapped in a `tf.contrib.rnn.DropoutWrapper` * Pass cell and embedded input to `tf.nn.dynamic_rnn()`

```

In [9]: from imp import reload
        reload(tests)

def encoding_layer(rnn_inputs, rnn_size, num_layers, keep_prob,
                  source_sequence_length, source_vocab_size,
                  encoding_embedding_size):
    """
    Create encoding layer
    :param rnn_inputs: Inputs for the RNN
    :param rnn_size: RNN Size
    :param num_layers: Number of layers
    :param keep_prob: Dropout keep probability
    :param source_sequence_length: a list of the lengths of each sequence in the batch
    :param source_vocab_size: vocabulary size of source data
    :param encoding_embedding_size: embedding size of source data
    :return: tuple (RNN output, RNN state)
    """
    enc_embed_input = tf.contrib.layers.embed_sequence(rnn_inputs, source_vocab_size, en

    # Stacked LSTMs
    def make_cell(rnn_size):
        lstm = tf.contrib.rnn.LSTMCell(rnn_size,
                                       initializer=tf.random_uniform_initializer(-0.
        drop = tf.contrib.rnn.DropoutWrapper(lstm, output_keep_prob=keep_prob)
        return drop

    cell = tf.contrib.rnn.MultiRNNCell([make_cell(rnn_size) for _ in range(num_layers)])

    # Get output and state
    enc_output, enc_state = tf.nn.dynamic_rnn(cell, enc_embed_input,
                                              sequence_length=source_sequence_length, dt

```

```

    return enc_output, enc_state

    """
    DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
    """

    tests.test_encoding_layer(encoding_layer)

```

Tests Passed

2.1.4 Decoding - Training

Create a training decoding layer: * Create a `tf.contrib.seq2seq.TrainingHelper`
 * Create a `tf.contrib.seq2seq.BasicDecoder` * Obtain the decoder outputs from
`tf.contrib.seq2seq.dynamic_decode`

```

In [10]: def decoding_layer_train(encoder_state, dec_cell, dec_embed_input,
                                target_sequence_length, max_summary_length,
                                output_layer, keep_prob):

    """
    Create a decoding layer for training
    :param encoder_state: Encoder State
    :param dec_cell: Decoder RNN Cell
    :param dec_embed_input: Decoder embedded input
    :param target_sequence_length: The lengths of each sequence in the target batch
    :param max_summary_length: The length of the longest sequence in the batch
    :param output_layer: Function to apply the output layer
    :param keep_prob: Dropout keep probability
    :return: BasicDecoderOutput containing training logits and sample_id
    """

    training_helper = tf.contrib.seq2seq.TrainingHelper(inputs=dec_embed_input,
                                                         sequence_length=target_sequence_length,
                                                         time_major=False)

    training_decoder = tf.contrib.seq2seq.BasicDecoder(dec_cell,
                                                         training_helper,
                                                         encoder_state,
                                                         output_layer)

    training_decoder_output, _ = tf.contrib.seq2seq.dynamic_decode(training_decoder,
                                                                    impute_finished=True,
                                                                    maximum_iterations=max_summary_length)

    return training_decoder_output

```

```

"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""

tests.test_decoding_layer_train(decoding_layer_train)

```

Tests Passed

2.1.5 Decoding - Inference

Create inference decoder: * Create a `tf.contrib.seq2seq.GreedyEmbeddingHelper`
 * Create a `tf.contrib.seq2seq.BasicDecoder` * Obtain the decoder outputs from
`tf.contrib.seq2seq.dynamic_decode`

```

In [11]: def decoding_layer_infer(encoder_state, dec_cell, dec_embeddings, start_of_sequence_id,
                                end_of_sequence_id, max_target_sequence_length,
                                vocab_size, output_layer, batch_size, keep_prob):

    """
    Create a decoding layer for inference
    :param encoder_state: Encoder state
    :param dec_cell: Decoder RNN Cell
    :param dec_embeddings: Decoder embeddings
    :param start_of_sequence_id: GO ID
    :param end_of_sequence_id: EOS Id
    :param max_target_sequence_length: Maximum length of target sequences
    :param vocab_size: Size of decoder/target vocabulary
    :param decoding_scope: TenorFlow Variable Scope for decoding
    :param output_layer: Function to apply the output layer
    :param batch_size: Batch size
    :param keep_prob: Dropout keep probability
    :return: BasicDecoderOutput containing inference logits and sample_id
    """

    start_tokens = tf.tile(tf.constant([start_of_sequence_id], dtype=tf.int32),
                            [batch_size], name='start_tokens')

    inference_helper = tf.contrib.seq2seq.GreedyEmbeddingHelper(dec_embeddings,
                                                                start_tokens,
                                                                end_of_sequence_id)

    inference_decoder = tf.contrib.seq2seq.BasicDecoder(dec_cell,
                                                         inference_helper,
                                                         encoder_state,
                                                         output_layer)

    inference_decoder_output, _ = tf.contrib.seq2seq.dynamic_decode(inference_decoder,
                                                                    impute_finished=True,
                                                                    maximum_iterations=

```



```

return inference_decoder_output

"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""

tests.test_decoding_layer_infer(decoding_layer_infer)

```

Tests Passed

2.1.6 Build the Decoding Layer

Implement `decoding_layer()` to create a Decoder RNN layer.

- Embed the target sequences
- Construct the decoder LSTM cell (just like you constructed the encoder cell above)
- Create an output layer to map the outputs of the decoder to the elements of our vocabulary
- Use your `decoding_layer_train(encoder_state, dec_cell, dec_embed_input, target_sequence_length, max_target_sequence_length, output_layer, keep_prob)` function to get the training logits.
- Use your `decoding_layer_infer(encoder_state, dec_cell, dec_embeddings, start_of_sequence_id, end_of_sequence_id, max_target_sequence_length, vocab_size, output_layer, batch_size, keep_prob)` function to get the inference logits.

Note: You'll need to use `tf.variable_scope` to share variables between training and inference.

```

In [12]: def decoding_layer(dec_input, encoder_state,
                           target_sequence_length, max_target_sequence_length,
                           rnn_size,
                           num_layers, target_vocab_to_int, target_vocab_size,
                           batch_size, keep_prob, decoding_embedding_size):

    """
    Create decoding layer
    :param dec_input: Decoder input
    :param encoder_state: Encoder state
    :param target_sequence_length: The lengths of each sequence in the target batch
    :param max_target_sequence_length: Maximum length of target sequences
    :param rnn_size: RNN Size
    :param num_layers: Number of layers
    :param target_vocab_to_int: Dictionary to go from the target words to an id
    :param target_vocab_size: Size of target vocabulary
    :param batch_size: The size of the batch
    :param keep_prob: Dropout keep probability
    :return: Tuple of (Training BasicDecoderOutput, Inference BasicDecoderOutput)
    """

```

```

dec_embeddings = tf.Variable(tf.random_uniform([target_vocab_size, decoding_embeddings_size], -0.1, 0.1))
dec_embed_input = tf.nn.embedding_lookup(dec_embeddings, dec_input)

def make_cell(rnn_size):
    lstm = tf.contrib.rnn.LSTMCell(rnn_size,
                                   initializer=tf.random_uniform_initializer(-0.1, 0.1, seed=1234))
    drop = tf.contrib.rnn.DropoutWrapper(lstm, output_keep_prob=keep_prob)
    return drop

dec_cell = tf.contrib.rnn.MultiRNNCell([make_cell(rnn_size) for _ in range(num_layers)])

output_layer = Dense(target_vocab_size,
                     kernel_initializer = tf.truncated_normal_initializer(mean = 0.0, stddev = 0.1, seed=1234))

with tf.variable_scope("decode"):
    training_decoder_output = decoding_layer_train(encoder_state, dec_cell,
                                                    dec_embed_input, target_sequence,
                                                    max_target_sequence_length, output_layer,
                                                    keep_prob)

    with tf.variable_scope("decode", reuse=True):
        start_of_sequence_id = target_vocab_to_int['<GO>']
        end_of_sequence_id = target_vocab_to_int['<EOS>']
        inference_decoder_output = decoding_layer_infer(encoder_state, dec_cell,
                                                         dec_embeddings, start_of_sequence_id,
                                                         end_of_sequence_id,
                                                         max_target_sequence_length, target_vocab_to_int,
                                                         output_layer, batch_size, keep_prob)

    return training_decoder_output, inference_decoder_output

"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_decoding_layer(decoding_layer)

```

Tests Passed

2.1.7 Build the Neural Network

Apply the functions you implemented above to:

- Apply embedding to the input data for the encoder.

- Encode the input using your `encoding_layer(rnn_inputs, rnn_size, num_layers, keep_prob, source_sequence_length, source_vocab_size, encoding_embedding_size)`.
- Process target data using your `process_decoder_input(target_data, target_vocab_to_int, batch_size)` function.
- Apply embedding to the target data for the decoder.
- Decode the encoded input using your `decoding_layer(dec_input, enc_state, target_sequence_length, max_target_sentence_length, rnn_size, num_layers, target_vocab_to_int, target_vocab_size, batch_size, keep_prob, dec_embedding_size)` function.

```
In [13]: def seq2seq_model(input_data, target_data, keep_prob, batch_size,
                           source_sequence_length, target_sequence_length,
                           max_target_sentence_length,
                           source_vocab_size, target_vocab_size,
                           enc_embedding_size, dec_embedding_size,
                           rnn_size, num_layers, target_vocab_to_int):
    """
    Build the Sequence-to-Sequence part of the neural network
    :param input_data: Input placeholder
    :param target_data: Target placeholder
    :param keep_prob: Dropout keep probability placeholder
    :param batch_size: Batch Size
    :param source_sequence_length: Sequence Lengths of source sequences in the batch
    :param target_sequence_length: Sequence Lengths of target sequences in the batch
    :param source_vocab_size: Source vocabulary size
    :param target_vocab_size: Target vocabulary size
    :param enc_embedding_size: Decoder embedding size
    :param dec_embedding_size: Encoder embedding size
    :param rnn_size: RNN Size
    :param num_layers: Number of layers
    :param target_vocab_to_int: Dictionary to go from the target words to an id
    :return: Tuple of (Training BasicDecoderOutput, Inference BasicDecoderOutput)
    """
    _, enc_state = encoding_layer(input_data, rnn_size, num_layers, keep_prob,
                                  source_sequence_length, source_vocab_size,
                                  enc_embedding_size)

    dec_input = process_decoder_input(target_data, target_vocab_to_int, batch_size)

    training_decoder_output, inference_decoder_output = decoding_layer(dec_input, enc_s
                                                                    target_sequence
                                                                    max_target_sente
                                                                    rnn_size, num_la
                                                                    target_vocab_to_
                                                                    target_vocab_siz
                                                                    batch_size, keep
                                                                    dec_embedding_si
```

```

        return training_decoder_output, inference_decoder_output

    """
    DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
    """
    tests.test_seq2seq_model(seq2seq_model)

```

Tests Passed

2.2 Neural Network Training

2.2.1 Hyperparameters

Tune the following parameters:

- Set `epochs` to the number of epochs.
- Set `batch_size` to the batch size.
- Set `rnn_size` to the size of the RNNs.
- Set `num_layers` to the number of layers.
- Set `encoding_embedding_size` to the size of the embedding for the encoder.
- Set `decoding_embedding_size` to the size of the embedding for the decoder.
- Set `learning_rate` to the learning rate.
- Set `keep_probability` to the Dropout keep probability
- Set `display_step` to state how many steps between each debug output statement

```

In [5]: # Number of Epochs
epochs = 20
# Batch Size
batch_size = 128
# RNN Size
rnn_size = 200
# Number of Layers
num_layers = 5
# Embedding Size
encoding_embedding_size = 64
decoding_embedding_size = 64
# Learning Rate
learning_rate = 0.001
# Dropout Keep Probability
keep_probability = 0.6
display_step = 100

```

2.2.2 Build the Graph

Build the graph using the neural network you implemented.

```

In [15]: """
DON'T MODIFY ANYTHING IN THIS CELL
"""

save_path = 'checkpoints/dev'
(source_int_text, target_int_text), (source_vocab_to_int, target_vocab_to_int), _ = helper.load(source_vocab_to_int, target_vocab_to_int)
max_target_sentence_length = max([len(sentence) for sentence in source_int_text])

train_graph = tf.Graph()
with train_graph.as_default():
    input_data, targets, lr, keep_prob, target_sequence_length, max_target_sequence_length = tf.placeholder_with_default(max_target_sentence_length, None, name='target_sequence_length')
    input_shape = tf.shape(input_data)

    train_logits, inference_logits = seq2seq_model(tf.reverse(input_data, [-1]),
                                                    targets,
                                                    keep_prob,
                                                    batch_size,
                                                    source_sequence_length,
                                                    target_sequence_length,
                                                    max_target_sequence_length,
                                                    len(source_vocab_to_int),
                                                    len(target_vocab_to_int),
                                                    encoding_embedding_size,
                                                    decoding_embedding_size,
                                                    rnn_size,
                                                    num_layers,
                                                    target_vocab_to_int)

    training_logits = tf.identity(train_logits.rnn_output, name='logits')
    inference_logits = tf.identity(inference_logits.sample_id, name='predictions')

    masks = tf.sequence_mask(target_sequence_length, max_target_sequence_length, dtype=tf.float32, name='masks')

    with tf.name_scope("optimization"):
        # Loss function
        cost = tf.contrib.seq2seq.sequence_loss(
            training_logits,
            targets,
            masks)

        # Optimizer
        optimizer = tf.train.AdamOptimizer(lr)

        # Gradient Clipping
        gradients = optimizer.compute_gradients(cost)
        capped_gradients = [(tf.clip_by_value(grad, -1., 1.), var) for grad, var in gradients]

```

```
train_op = optimizer.apply_gradients(capped_gradients)
```

Batch and pad the source and target sequences

```
In [16]: """
DON'T MODIFY ANYTHING IN THIS CELL
"""

def pad_sentence_batch(sentence_batch, pad_int):
    """Pad sentences with <PAD> so that each sentence of a batch has the same length"""
    max_sentence = max([len(sentence) for sentence in sentence_batch])
    return [sentence + [pad_int] * (max_sentence - len(sentence)) for sentence in sentence_batch]

def get_batches(sources, targets, batch_size, source_pad_int, target_pad_int):
    """Batch targets, sources, and the lengths of their sentences together"""
    for batch_i in range(0, len(sources)//batch_size):
        start_i = batch_i * batch_size

        # Slice the right amount for the batch
        sources_batch = sources[start_i:start_i + batch_size]
        targets_batch = targets[start_i:start_i + batch_size]

        # Pad
        pad_sources_batch = np.array(pad_sentence_batch(sources_batch, source_pad_int))
        pad_targets_batch = np.array(pad_sentence_batch(targets_batch, target_pad_int))

        # Need the lengths for the _lengths parameters
        pad_targets_lengths = []
        for target in pad_targets_batch:
            pad_targets_lengths.append(len(target))

        pad_source_lengths = []
        for source in pad_sources_batch:
            pad_source_lengths.append(len(source))

        yield pad_sources_batch, pad_targets_batch, pad_source_lengths, pad_targets_lengths
```

2.2.3 Train

Train the neural network on the preprocessed data. If you have a hard time getting a good loss, check the forms to see if anyone is having the same problem.

```
In [17]: """
DON'T MODIFY ANYTHING IN THIS CELL
"""

def get_accuracy(target, logits):
    """
    Calculate accuracy
    """
```

```

max_seq = max(target.shape[1], logits.shape[1])
if max_seq - target.shape[1]:
    target = np.pad(
        target,
        [(0,0),(0,max_seq - target.shape[1])],
        'constant')
if max_seq - logits.shape[1]:
    logits = np.pad(
        logits,
        [(0,0),(0,max_seq - logits.shape[1])],
        'constant')

return np.mean(np.equal(target, logits))

# Split data to training and validation sets
train_source = source_int_text[batch_size:]
train_target = target_int_text[batch_size:]
valid_source = source_int_text[:batch_size]
valid_target = target_int_text[:batch_size]
(valid_sources_batch, valid_targets_batch, valid_sources_lengths, valid_targets_lengths)

with tf.Session(graph=train_graph) as sess:
    sess.run(tf.global_variables_initializer())

    for epoch_i in range(epochs):
        for batch_i, (source_batch, target_batch, sources_lengths, targets_lengths) in
            get_batches(train_source, train_target, batch_size,
                        source_vocab_to_int['<PAD>'],
                        target_vocab_to_int['<PAD>']):

            _, loss = sess.run(
                [train_op, cost],
                {input_data: source_batch,
                 targets: target_batch,
                 lr: learning_rate,
                 target_sequence_length: targets_lengths,
                 source_sequence_length: sources_lengths,
                 keep_prob: keep_probability})

            if batch_i % display_step == 0 and batch_i > 0:

                batch_train_logits = sess.run(
                    inference_logits,

```

```

        {input_data: source_batch,
         source_sequence_length: sources_lengths,
         target_sequence_length: targets_lengths,
         keep_prob: 1.0})

    batch_valid_logits = sess.run(
        inference_logits,
        {input_data: valid_sources_batch,
         source_sequence_length: valid_sources_lengths,
         target_sequence_length: valid_targets_lengths,
         keep_prob: 1.0})

    train_acc = get_accuracy(target_batch, batch_train_logits)

    valid_acc = get_accuracy(valid_targets_batch, batch_valid_logits)

    print('Epoch {:>3} Batch {:>4}/{:} - Train Accuracy: {:>6.4f}, Validation
          .format(epoch_i, batch_i, len(source_int_text) // batch_size, tra

    # Save Model
    saver = tf.train.Saver()
    saver.save(sess, save_path)
    print('Model Trained and Saved')

```

```

Epoch 0 Batch 100/1077 - Train Accuracy: 0.4414, Validation Accuracy: 0.4975, Loss: 2.3390
Epoch 0 Batch 200/1077 - Train Accuracy: 0.4789, Validation Accuracy: 0.5305, Loss: 1.8094
Epoch 0 Batch 300/1077 - Train Accuracy: 0.4823, Validation Accuracy: 0.5522, Loss: 1.5551
Epoch 0 Batch 400/1077 - Train Accuracy: 0.5020, Validation Accuracy: 0.5455, Loss: 1.3674
Epoch 0 Batch 500/1077 - Train Accuracy: 0.5437, Validation Accuracy: 0.5703, Loss: 1.2048
Epoch 0 Batch 600/1077 - Train Accuracy: 0.5547, Validation Accuracy: 0.5629, Loss: 1.0586
Epoch 0 Batch 700/1077 - Train Accuracy: 0.5270, Validation Accuracy: 0.5700, Loss: 0.9206
Epoch 0 Batch 800/1077 - Train Accuracy: 0.5312, Validation Accuracy: 0.5803, Loss: 0.9009
Epoch 0 Batch 900/1077 - Train Accuracy: 0.5797, Validation Accuracy: 0.5547, Loss: 0.9050
Epoch 0 Batch 1000/1077 - Train Accuracy: 0.6146, Validation Accuracy: 0.6151, Loss: 0.7551
Epoch 1 Batch 100/1077 - Train Accuracy: 0.6039, Validation Accuracy: 0.6026, Loss: 0.7284
Epoch 1 Batch 200/1077 - Train Accuracy: 0.5676, Validation Accuracy: 0.6236, Loss: 0.7317
Epoch 1 Batch 300/1077 - Train Accuracy: 0.5835, Validation Accuracy: 0.6094, Loss: 0.6770
Epoch 1 Batch 400/1077 - Train Accuracy: 0.6512, Validation Accuracy: 0.6172, Loss: 0.6555
Epoch 1 Batch 500/1077 - Train Accuracy: 0.6133, Validation Accuracy: 0.6371, Loss: 0.6108
Epoch 1 Batch 600/1077 - Train Accuracy: 0.6555, Validation Accuracy: 0.6388, Loss: 0.5428
Epoch 1 Batch 700/1077 - Train Accuracy: 0.6098, Validation Accuracy: 0.6495, Loss: 0.5479
Epoch 1 Batch 800/1077 - Train Accuracy: 0.5789, Validation Accuracy: 0.6257, Loss: 0.5858
Epoch 1 Batch 900/1077 - Train Accuracy: 0.6570, Validation Accuracy: 0.6616, Loss: 0.5771
Epoch 1 Batch 1000/1077 - Train Accuracy: 0.7113, Validation Accuracy: 0.6562, Loss: 0.4755
Epoch 2 Batch 100/1077 - Train Accuracy: 0.6730, Validation Accuracy: 0.6879, Loss: 0.4673
Epoch 2 Batch 200/1077 - Train Accuracy: 0.6773, Validation Accuracy: 0.7106, Loss: 0.4947
Epoch 2 Batch 300/1077 - Train Accuracy: 0.6793, Validation Accuracy: 0.6882, Loss: 0.4511

```


Epoch	2	Batch	400/1077	-	Train Accuracy:	0.7355,	Validation Accuracy:	0.7116,	Loss:	0.4439
Epoch	2	Batch	500/1077	-	Train Accuracy:	0.7176,	Validation Accuracy:	0.7088,	Loss:	0.4062
Epoch	2	Batch	600/1077	-	Train Accuracy:	0.7597,	Validation Accuracy:	0.7085,	Loss:	0.3707
Epoch	2	Batch	700/1077	-	Train Accuracy:	0.7227,	Validation Accuracy:	0.7223,	Loss:	0.3769
Epoch	2	Batch	800/1077	-	Train Accuracy:	0.7383,	Validation Accuracy:	0.7337,	Loss:	0.3710
Epoch	2	Batch	900/1077	-	Train Accuracy:	0.7594,	Validation Accuracy:	0.7145,	Loss:	0.3764
Epoch	2	Batch	1000/1077	-	Train Accuracy:	0.7935,	Validation Accuracy:	0.7415,	Loss:	0.3287
Epoch	3	Batch	100/1077	-	Train Accuracy:	0.7871,	Validation Accuracy:	0.7773,	Loss:	0.3142
Epoch	3	Batch	200/1077	-	Train Accuracy:	0.7516,	Validation Accuracy:	0.7859,	Loss:	0.3352
Epoch	3	Batch	300/1077	-	Train Accuracy:	0.8035,	Validation Accuracy:	0.7546,	Loss:	0.2892
Epoch	3	Batch	400/1077	-	Train Accuracy:	0.8148,	Validation Accuracy:	0.7837,	Loss:	0.3038
Epoch	3	Batch	500/1077	-	Train Accuracy:	0.8086,	Validation Accuracy:	0.7514,	Loss:	0.2682
Epoch	3	Batch	600/1077	-	Train Accuracy:	0.8304,	Validation Accuracy:	0.7869,	Loss:	0.2375
Epoch	3	Batch	700/1077	-	Train Accuracy:	0.7566,	Validation Accuracy:	0.7827,	Loss:	0.2337
Epoch	3	Batch	800/1077	-	Train Accuracy:	0.8297,	Validation Accuracy:	0.8061,	Loss:	0.2417
Epoch	3	Batch	900/1077	-	Train Accuracy:	0.8203,	Validation Accuracy:	0.8136,	Loss:	0.2556
Epoch	3	Batch	1000/1077	-	Train Accuracy:	0.8557,	Validation Accuracy:	0.8114,	Loss:	0.2153
Epoch	4	Batch	100/1077	-	Train Accuracy:	0.8379,	Validation Accuracy:	0.8200,	Loss:	0.2065
Epoch	4	Batch	200/1077	-	Train Accuracy:	0.7953,	Validation Accuracy:	0.8303,	Loss:	0.2485
Epoch	4	Batch	300/1077	-	Train Accuracy:	0.8709,	Validation Accuracy:	0.8271,	Loss:	0.2101
Epoch	4	Batch	400/1077	-	Train Accuracy:	0.8496,	Validation Accuracy:	0.8182,	Loss:	0.2222
Epoch	4	Batch	500/1077	-	Train Accuracy:	0.8824,	Validation Accuracy:	0.8263,	Loss:	0.1962
Epoch	4	Batch	600/1077	-	Train Accuracy:	0.8594,	Validation Accuracy:	0.8349,	Loss:	0.1765
Epoch	4	Batch	700/1077	-	Train Accuracy:	0.8516,	Validation Accuracy:	0.8232,	Loss:	0.2232
Epoch	4	Batch	800/1077	-	Train Accuracy:	0.8699,	Validation Accuracy:	0.8555,	Loss:	0.1690
Epoch	4	Batch	900/1077	-	Train Accuracy:	0.9023,	Validation Accuracy:	0.8548,	Loss:	0.1737
Epoch	4	Batch	1000/1077	-	Train Accuracy:	0.8638,	Validation Accuracy:	0.8533,	Loss:	0.1551
Epoch	5	Batch	100/1077	-	Train Accuracy:	0.8691,	Validation Accuracy:	0.8619,	Loss:	0.1637
Epoch	5	Batch	200/1077	-	Train Accuracy:	0.8246,	Validation Accuracy:	0.8615,	Loss:	0.1700
Epoch	5	Batch	300/1077	-	Train Accuracy:	0.9132,	Validation Accuracy:	0.8583,	Loss:	0.1444
Epoch	5	Batch	400/1077	-	Train Accuracy:	0.8828,	Validation Accuracy:	0.8587,	Loss:	0.1409
Epoch	5	Batch	500/1077	-	Train Accuracy:	0.8898,	Validation Accuracy:	0.8576,	Loss:	0.1229
Epoch	5	Batch	600/1077	-	Train Accuracy:	0.8895,	Validation Accuracy:	0.8420,	Loss:	0.1337
Epoch	5	Batch	700/1077	-	Train Accuracy:	0.9176,	Validation Accuracy:	0.8516,	Loss:	0.1192
Epoch	5	Batch	800/1077	-	Train Accuracy:	0.8883,	Validation Accuracy:	0.8540,	Loss:	0.1343
Epoch	5	Batch	900/1077	-	Train Accuracy:	0.9125,	Validation Accuracy:	0.8761,	Loss:	0.1387
Epoch	5	Batch	1000/1077	-	Train Accuracy:	0.8802,	Validation Accuracy:	0.8825,	Loss:	0.1210
Epoch	6	Batch	100/1077	-	Train Accuracy:	0.8715,	Validation Accuracy:	0.8761,	Loss:	0.1204
Epoch	6	Batch	200/1077	-	Train Accuracy:	0.8426,	Validation Accuracy:	0.8736,	Loss:	0.1212
Epoch	6	Batch	300/1077	-	Train Accuracy:	0.9268,	Validation Accuracy:	0.8796,	Loss:	0.1157
Epoch	6	Batch	400/1077	-	Train Accuracy:	0.8922,	Validation Accuracy:	0.8757,	Loss:	0.1279
Epoch	6	Batch	500/1077	-	Train Accuracy:	0.9094,	Validation Accuracy:	0.8651,	Loss:	0.1058
Epoch	6	Batch	600/1077	-	Train Accuracy:	0.9003,	Validation Accuracy:	0.8729,	Loss:	0.1178
Epoch	6	Batch	700/1077	-	Train Accuracy:	0.9262,	Validation Accuracy:	0.8643,	Loss:	0.0912
Epoch	6	Batch	800/1077	-	Train Accuracy:	0.9000,	Validation Accuracy:	0.8732,	Loss:	0.1045
Epoch	6	Batch	900/1077	-	Train Accuracy:	0.9246,	Validation Accuracy:	0.8640,	Loss:	0.1093
Epoch	6	Batch	1000/1077	-	Train Accuracy:	0.9070,	Validation Accuracy:	0.8835,	Loss:	0.0909
Epoch	7	Batch	100/1077	-	Train Accuracy:	0.9090,	Validation Accuracy:	0.8945,	Loss:	0.0891

Epoch	7	Batch	200/1077	-	Train Accuracy:	0.8727,	Validation Accuracy:	0.8924,	Loss:	0.1068
Epoch	7	Batch	300/1077	-	Train Accuracy:	0.9597,	Validation Accuracy:	0.8899,	Loss:	0.0797
Epoch	7	Batch	400/1077	-	Train Accuracy:	0.9078,	Validation Accuracy:	0.8913,	Loss:	0.1014
Epoch	7	Batch	500/1077	-	Train Accuracy:	0.9164,	Validation Accuracy:	0.8899,	Loss:	0.0728
Epoch	7	Batch	600/1077	-	Train Accuracy:	0.9252,	Validation Accuracy:	0.8999,	Loss:	0.0777
Epoch	7	Batch	700/1077	-	Train Accuracy:	0.9352,	Validation Accuracy:	0.8878,	Loss:	0.0660
Epoch	7	Batch	800/1077	-	Train Accuracy:	0.9332,	Validation Accuracy:	0.8949,	Loss:	0.0770
Epoch	7	Batch	900/1077	-	Train Accuracy:	0.9465,	Validation Accuracy:	0.8853,	Loss:	0.0889
Epoch	7	Batch	1000/1077	-	Train Accuracy:	0.9115,	Validation Accuracy:	0.9023,	Loss:	0.0749
Epoch	8	Batch	100/1077	-	Train Accuracy:	0.9152,	Validation Accuracy:	0.9144,	Loss:	0.0735
Epoch	8	Batch	200/1077	-	Train Accuracy:	0.8855,	Validation Accuracy:	0.9016,	Loss:	0.0902
Epoch	8	Batch	300/1077	-	Train Accuracy:	0.9437,	Validation Accuracy:	0.9048,	Loss:	0.0628
Epoch	8	Batch	400/1077	-	Train Accuracy:	0.9203,	Validation Accuracy:	0.8991,	Loss:	0.0738
Epoch	8	Batch	500/1077	-	Train Accuracy:	0.9266,	Validation Accuracy:	0.9062,	Loss:	0.0528
Epoch	8	Batch	600/1077	-	Train Accuracy:	0.9483,	Validation Accuracy:	0.9102,	Loss:	0.0622
Epoch	8	Batch	700/1077	-	Train Accuracy:	0.9344,	Validation Accuracy:	0.9151,	Loss:	0.0582
Epoch	8	Batch	800/1077	-	Train Accuracy:	0.9270,	Validation Accuracy:	0.9087,	Loss:	0.0809
Epoch	8	Batch	900/1077	-	Train Accuracy:	0.9457,	Validation Accuracy:	0.9041,	Loss:	0.0711
Epoch	8	Batch	1000/1077	-	Train Accuracy:	0.9297,	Validation Accuracy:	0.9123,	Loss:	0.0588
Epoch	9	Batch	100/1077	-	Train Accuracy:	0.9223,	Validation Accuracy:	0.8949,	Loss:	0.0608
Epoch	9	Batch	200/1077	-	Train Accuracy:	0.9238,	Validation Accuracy:	0.9222,	Loss:	0.0662
Epoch	9	Batch	300/1077	-	Train Accuracy:	0.9433,	Validation Accuracy:	0.9105,	Loss:	0.0499
Epoch	9	Batch	400/1077	-	Train Accuracy:	0.9199,	Validation Accuracy:	0.9339,	Loss:	0.0529
Epoch	9	Batch	500/1077	-	Train Accuracy:	0.9277,	Validation Accuracy:	0.9141,	Loss:	0.0503
Epoch	9	Batch	600/1077	-	Train Accuracy:	0.9613,	Validation Accuracy:	0.9173,	Loss:	0.0565
Epoch	9	Batch	700/1077	-	Train Accuracy:	0.9281,	Validation Accuracy:	0.9055,	Loss:	0.0483
Epoch	9	Batch	800/1077	-	Train Accuracy:	0.9383,	Validation Accuracy:	0.9098,	Loss:	0.0601
Epoch	9	Batch	900/1077	-	Train Accuracy:	0.9195,	Validation Accuracy:	0.9102,	Loss:	0.0662
Epoch	9	Batch	1000/1077	-	Train Accuracy:	0.9379,	Validation Accuracy:	0.9183,	Loss:	0.0545
Epoch	10	Batch	100/1077	-	Train Accuracy:	0.9375,	Validation Accuracy:	0.9215,	Loss:	0.0560
Epoch	10	Batch	200/1077	-	Train Accuracy:	0.9320,	Validation Accuracy:	0.9297,	Loss:	0.0575
Epoch	10	Batch	300/1077	-	Train Accuracy:	0.9576,	Validation Accuracy:	0.9244,	Loss:	0.0478
Epoch	10	Batch	400/1077	-	Train Accuracy:	0.9348,	Validation Accuracy:	0.9396,	Loss:	0.0601
Epoch	10	Batch	500/1077	-	Train Accuracy:	0.9449,	Validation Accuracy:	0.9212,	Loss:	0.0431
Epoch	10	Batch	600/1077	-	Train Accuracy:	0.9635,	Validation Accuracy:	0.9084,	Loss:	0.0566
Epoch	10	Batch	700/1077	-	Train Accuracy:	0.9527,	Validation Accuracy:	0.9222,	Loss:	0.0499
Epoch	10	Batch	800/1077	-	Train Accuracy:	0.9391,	Validation Accuracy:	0.9237,	Loss:	0.0518
Epoch	10	Batch	900/1077	-	Train Accuracy:	0.9371,	Validation Accuracy:	0.9197,	Loss:	0.0605
Epoch	10	Batch	1000/1077	-	Train Accuracy:	0.9375,	Validation Accuracy:	0.9215,	Loss:	0.0521
Epoch	11	Batch	100/1077	-	Train Accuracy:	0.9141,	Validation Accuracy:	0.9308,	Loss:	0.0581
Epoch	11	Batch	200/1077	-	Train Accuracy:	0.9285,	Validation Accuracy:	0.9169,	Loss:	0.0612
Epoch	11	Batch	300/1077	-	Train Accuracy:	0.9531,	Validation Accuracy:	0.9339,	Loss:	0.0397
Epoch	11	Batch	400/1077	-	Train Accuracy:	0.9480,	Validation Accuracy:	0.9283,	Loss:	0.0536
Epoch	11	Batch	500/1077	-	Train Accuracy:	0.9531,	Validation Accuracy:	0.9308,	Loss:	0.0401
Epoch	11	Batch	600/1077	-	Train Accuracy:	0.9624,	Validation Accuracy:	0.9325,	Loss:	0.0436
Epoch	11	Batch	700/1077	-	Train Accuracy:	0.9437,	Validation Accuracy:	0.9304,	Loss:	0.0401
Epoch	11	Batch	800/1077	-	Train Accuracy:	0.9254,	Validation Accuracy:	0.9322,	Loss:	0.0636
Epoch	11	Batch	900/1077	-	Train Accuracy:	0.9500,	Validation Accuracy:	0.9247,	Loss:	0.0558

Epoch 11 Batch 1000/1077 - Train Accuracy: 0.9364, Validation Accuracy: 0.9293, Loss: 0.0542
 Epoch 12 Batch 100/1077 - Train Accuracy: 0.9445, Validation Accuracy: 0.9435, Loss: 0.0373
 Epoch 12 Batch 200/1077 - Train Accuracy: 0.9461, Validation Accuracy: 0.9332, Loss: 0.0500
 Epoch 12 Batch 300/1077 - Train Accuracy: 0.9585, Validation Accuracy: 0.9325, Loss: 0.0666
 Epoch 12 Batch 400/1077 - Train Accuracy: 0.9492, Validation Accuracy: 0.9446, Loss: 0.0564
 Epoch 12 Batch 500/1077 - Train Accuracy: 0.9457, Validation Accuracy: 0.9400, Loss: 0.0381
 Epoch 12 Batch 600/1077 - Train Accuracy: 0.9676, Validation Accuracy: 0.9432, Loss: 0.0429
 Epoch 12 Batch 700/1077 - Train Accuracy: 0.9594, Validation Accuracy: 0.9421, Loss: 0.0357
 Epoch 12 Batch 800/1077 - Train Accuracy: 0.9574, Validation Accuracy: 0.9347, Loss: 0.0348
 Epoch 12 Batch 900/1077 - Train Accuracy: 0.9477, Validation Accuracy: 0.9581, Loss: 0.0420
 Epoch 12 Batch 1000/1077 - Train Accuracy: 0.9289, Validation Accuracy: 0.9350, Loss: 0.0469
 Epoch 13 Batch 100/1077 - Train Accuracy: 0.9422, Validation Accuracy: 0.9485, Loss: 0.0424
 Epoch 13 Batch 200/1077 - Train Accuracy: 0.9625, Validation Accuracy: 0.9375, Loss: 0.0440
 Epoch 13 Batch 300/1077 - Train Accuracy: 0.9667, Validation Accuracy: 0.9318, Loss: 0.0318
 Epoch 13 Batch 400/1077 - Train Accuracy: 0.9504, Validation Accuracy: 0.9442, Loss: 0.0477
 Epoch 13 Batch 500/1077 - Train Accuracy: 0.9652, Validation Accuracy: 0.9364, Loss: 0.0379
 Epoch 13 Batch 600/1077 - Train Accuracy: 0.9762, Validation Accuracy: 0.9549, Loss: 0.0395
 Epoch 13 Batch 700/1077 - Train Accuracy: 0.9719, Validation Accuracy: 0.9506, Loss: 0.0337
 Epoch 13 Batch 800/1077 - Train Accuracy: 0.9594, Validation Accuracy: 0.9638, Loss: 0.0327
 Epoch 13 Batch 900/1077 - Train Accuracy: 0.9570, Validation Accuracy: 0.9528, Loss: 0.0438
 Epoch 13 Batch 1000/1077 - Train Accuracy: 0.9449, Validation Accuracy: 0.9450, Loss: 0.0396
 Epoch 14 Batch 100/1077 - Train Accuracy: 0.9512, Validation Accuracy: 0.9709, Loss: 0.0332
 Epoch 14 Batch 200/1077 - Train Accuracy: 0.9812, Validation Accuracy: 0.9499, Loss: 0.0360
 Epoch 14 Batch 300/1077 - Train Accuracy: 0.9696, Validation Accuracy: 0.9549, Loss: 0.0292
 Epoch 14 Batch 400/1077 - Train Accuracy: 0.9688, Validation Accuracy: 0.9698, Loss: 0.0430
 Epoch 14 Batch 500/1077 - Train Accuracy: 0.9527, Validation Accuracy: 0.9556, Loss: 0.0307
 Epoch 14 Batch 600/1077 - Train Accuracy: 0.9803, Validation Accuracy: 0.9670, Loss: 0.0387
 Epoch 14 Batch 700/1077 - Train Accuracy: 0.9641, Validation Accuracy: 0.9535, Loss: 0.0358
 Epoch 14 Batch 800/1077 - Train Accuracy: 0.9695, Validation Accuracy: 0.9602, Loss: 0.0363
 Epoch 14 Batch 900/1077 - Train Accuracy: 0.9563, Validation Accuracy: 0.9545, Loss: 0.0446
 Epoch 14 Batch 1000/1077 - Train Accuracy: 0.9513, Validation Accuracy: 0.9411, Loss: 0.0414
 Epoch 15 Batch 100/1077 - Train Accuracy: 0.9590, Validation Accuracy: 0.9709, Loss: 0.0397
 Epoch 15 Batch 200/1077 - Train Accuracy: 0.9633, Validation Accuracy: 0.9627, Loss: 0.0369
 Epoch 15 Batch 300/1077 - Train Accuracy: 0.9618, Validation Accuracy: 0.9780, Loss: 0.0313
 Epoch 15 Batch 400/1077 - Train Accuracy: 0.9629, Validation Accuracy: 0.9631, Loss: 0.0438
 Epoch 15 Batch 500/1077 - Train Accuracy: 0.9613, Validation Accuracy: 0.9624, Loss: 0.0346
 Epoch 15 Batch 600/1077 - Train Accuracy: 0.9658, Validation Accuracy: 0.9570, Loss: 0.0392
 Epoch 15 Batch 700/1077 - Train Accuracy: 0.9594, Validation Accuracy: 0.9538, Loss: 0.0428
 Epoch 15 Batch 800/1077 - Train Accuracy: 0.9637, Validation Accuracy: 0.9709, Loss: 0.0303
 Epoch 15 Batch 900/1077 - Train Accuracy: 0.9590, Validation Accuracy: 0.9638, Loss: 0.0442
 Epoch 15 Batch 1000/1077 - Train Accuracy: 0.9554, Validation Accuracy: 0.9556, Loss: 0.0358
 Epoch 16 Batch 100/1077 - Train Accuracy: 0.9621, Validation Accuracy: 0.9762, Loss: 0.0322
 Epoch 16 Batch 200/1077 - Train Accuracy: 0.9797, Validation Accuracy: 0.9648, Loss: 0.0340
 Epoch 16 Batch 300/1077 - Train Accuracy: 0.9683, Validation Accuracy: 0.9581, Loss: 0.0287
 Epoch 16 Batch 400/1077 - Train Accuracy: 0.9680, Validation Accuracy: 0.9613, Loss: 0.0369
 Epoch 16 Batch 500/1077 - Train Accuracy: 0.9547, Validation Accuracy: 0.9549, Loss: 0.0283
 Epoch 16 Batch 600/1077 - Train Accuracy: 0.9762, Validation Accuracy: 0.9737, Loss: 0.0351
 Epoch 16 Batch 700/1077 - Train Accuracy: 0.9750, Validation Accuracy: 0.9659, Loss: 0.0287

```

Epoch 16 Batch 800/1077 - Train Accuracy: 0.9688, Validation Accuracy: 0.9645, Loss: 0.0299
Epoch 16 Batch 900/1077 - Train Accuracy: 0.9684, Validation Accuracy: 0.9691, Loss: 0.0363
Epoch 16 Batch 1000/1077 - Train Accuracy: 0.9431, Validation Accuracy: 0.9670, Loss: 0.0325
Epoch 17 Batch 100/1077 - Train Accuracy: 0.9652, Validation Accuracy: 0.9648, Loss: 0.0283
Epoch 17 Batch 200/1077 - Train Accuracy: 0.9809, Validation Accuracy: 0.9652, Loss: 0.0298
Epoch 17 Batch 300/1077 - Train Accuracy: 0.9708, Validation Accuracy: 0.9673, Loss: 0.0241
Epoch 17 Batch 400/1077 - Train Accuracy: 0.9766, Validation Accuracy: 0.9627, Loss: 0.0349
Epoch 17 Batch 500/1077 - Train Accuracy: 0.9637, Validation Accuracy: 0.9670, Loss: 0.0236
Epoch 17 Batch 600/1077 - Train Accuracy: 0.9766, Validation Accuracy: 0.9695, Loss: 0.0333
Epoch 17 Batch 700/1077 - Train Accuracy: 0.9727, Validation Accuracy: 0.9581, Loss: 0.0209
Epoch 17 Batch 800/1077 - Train Accuracy: 0.9637, Validation Accuracy: 0.9638, Loss: 0.0235
Epoch 17 Batch 900/1077 - Train Accuracy: 0.9586, Validation Accuracy: 0.9652, Loss: 0.0348
Epoch 17 Batch 1000/1077 - Train Accuracy: 0.9621, Validation Accuracy: 0.9549, Loss: 0.0314
Epoch 18 Batch 100/1077 - Train Accuracy: 0.9660, Validation Accuracy: 0.9684, Loss: 0.0231
Epoch 18 Batch 200/1077 - Train Accuracy: 0.9805, Validation Accuracy: 0.9734, Loss: 0.0418
Epoch 18 Batch 300/1077 - Train Accuracy: 0.9720, Validation Accuracy: 0.9759, Loss: 0.0238
Epoch 18 Batch 400/1077 - Train Accuracy: 0.9750, Validation Accuracy: 0.9741, Loss: 0.0319
Epoch 18 Batch 500/1077 - Train Accuracy: 0.9656, Validation Accuracy: 0.9695, Loss: 0.0218
Epoch 18 Batch 600/1077 - Train Accuracy: 0.9736, Validation Accuracy: 0.9624, Loss: 0.0312
Epoch 18 Batch 700/1077 - Train Accuracy: 0.9734, Validation Accuracy: 0.9709, Loss: 0.0272
Epoch 18 Batch 800/1077 - Train Accuracy: 0.9742, Validation Accuracy: 0.9695, Loss: 0.0440
Epoch 18 Batch 900/1077 - Train Accuracy: 0.9781, Validation Accuracy: 0.9766, Loss: 0.0357
Epoch 18 Batch 1000/1077 - Train Accuracy: 0.9501, Validation Accuracy: 0.9627, Loss: 0.0369
Epoch 19 Batch 100/1077 - Train Accuracy: 0.9613, Validation Accuracy: 0.9691, Loss: 0.0273
Epoch 19 Batch 200/1077 - Train Accuracy: 0.9840, Validation Accuracy: 0.9648, Loss: 0.0226
Epoch 19 Batch 300/1077 - Train Accuracy: 0.9692, Validation Accuracy: 0.9741, Loss: 0.0235
Epoch 19 Batch 400/1077 - Train Accuracy: 0.9738, Validation Accuracy: 0.9801, Loss: 0.0362
Epoch 19 Batch 500/1077 - Train Accuracy: 0.9691, Validation Accuracy: 0.9702, Loss: 0.0215
Epoch 19 Batch 600/1077 - Train Accuracy: 0.9751, Validation Accuracy: 0.9691, Loss: 0.0259
Epoch 19 Batch 700/1077 - Train Accuracy: 0.9812, Validation Accuracy: 0.9698, Loss: 0.0212
Epoch 19 Batch 800/1077 - Train Accuracy: 0.9762, Validation Accuracy: 0.9769, Loss: 0.0236
Epoch 19 Batch 900/1077 - Train Accuracy: 0.9766, Validation Accuracy: 0.9719, Loss: 0.0301
Epoch 19 Batch 1000/1077 - Train Accuracy: 0.9658, Validation Accuracy: 0.9762, Loss: 0.0324
Model Trained and Saved

```

2.2.4 Save Parameters

Save the `batch_size` and `save_path` parameters for inference.

```

In [21]: """
          DON'T MODIFY ANYTHING IN THIS CELL
          """
          # Save parameters for checkpoint
          helper.save_params(save_path)

```

3 Checkpoint

```
In [2]: """
        DON'T MODIFY ANYTHING IN THIS CELL
        """

import tensorflow as tf
import numpy as np
import helper
import problem_unittests as tests

_, (source_vocab_to_int, target_vocab_to_int), (source_int_to_vocab, target_int_to_vocab)
load_path = helper.load_params()
```

3.1 Sentence to Sequence

To feed a sentence into the model for translation, you first need to preprocess it. Implement the function `sentence_to_seq()` to preprocess new sentences.

- Convert the sentence to lowercase
- Convert words into ids using `vocab_to_int`
- Convert words not in the vocabulary, to the <UNK> word id.

```
In [3]: def sentence_to_seq(sentence, vocab_to_int):
        """
        Convert a sentence to a sequence of ids
        :param sentence: String
        :param vocab_to_int: Dictionary to go from the words to an id
        :return: List of word ids
        """

        return [vocab_to_int.get(word, vocab_to_int['<UNK>']) for word in sentence.split()]
        """
        DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
        """

        tests.test_sentence_to_seq(sentence_to_seq)
```

Tests Passed

3.2 Translate

This will translate `translate_sentence` from English to French.

```
In [6]: translate_sentence = 'he saw a old yellow truck .'

        """
        DON'T MODIFY ANYTHING IN THIS CELL
        """

        translate_sentence = sentence_to_seq(translate_sentence, source_vocab_to_int)
```

```

loaded_graph = tf.Graph()
with tf.Session(graph=loaded_graph) as sess:
    # Load saved model
    loader = tf.train.import_meta_graph(load_path + '.meta')
    loader.restore(sess, load_path)

    input_data = loaded_graph.get_tensor_by_name('input:0')
    logits = loaded_graph.get_tensor_by_name('predictions:0')
    target_sequence_length = loaded_graph.get_tensor_by_name('target_sequence_length:0')
    source_sequence_length = loaded_graph.get_tensor_by_name('source_sequence_length:0')
    keep_prob = loaded_graph.get_tensor_by_name('keep_prob:0')

    translate_logits = sess.run(logits, {input_data: [translate_sentence]*batch_size,
                                           target_sequence_length: [len(translate_sentence)],
                                           source_sequence_length: [len(translate_sentence)],
                                           keep_prob: 1.0})[0]

    print('Input')
    print('  Word Ids:      {}'.format([i for i in translate_sentence]))
    print('  English Words: {}'.format([source_int_to_vocab[i] for i in translate_sentence]))

    print('\nPrediction')
    print('  Word Ids:      {}'.format([i for i in translate_logits]))
    print('  French Words: {}'.format(" ".join([target_int_to_vocab[i] for i in translate_logits])))

```

INFO:tensorflow:Restoring parameters from checkpoints/dev

Input

Word Ids: [104, 18, 140, 56, 76, 43, 47]

English Words: ['he', 'saw', 'a', 'old', 'yellow', 'truck', '.']

Prediction

Word Ids: [127, 319, 163, 104, 217, 44, 254, 99, 1]

French Words: il a vu une vieille voiture jaune . <EOS>

3.3 Imperfect Translation

You might notice that some sentences translate better than others. Since the dataset you're using only has a vocabulary of 227 English words of the thousands that you use, you're only going to see good results using these words. For this project, you don't need a perfect translation. However, if you want to create a better translation model, you'll need better data.

You can train on the [WMT10 French-English corpus](#). This dataset has more vocabulary and is richer in topics discussed. However, this will take you days to train, so make sure you've a GPU and the neural network is performing well on dataset we provided. Just make sure you play with the WMT10 corpus after you've submitted this project. ## Submitting This Project When submitting this project, make sure to run all the cells before saving the notebook. Save the notebook file as "dlnd_language_translation.ipynb" and save it as a HTML file under "File" -> "Download as".

Include the "helper.py" and "problem_unittests.py" files in your submission.