In [ ]:

```
#********************************************************************************
#
#                Project : Language Translation from English to French
#
#********************************************************************************
# Helper Functions

import os
import pickle
import copy
import numpy as np
import tensorflow as tf

# Codes set for Padding, End of Sentence,Unknown word and Go for Starting the new Sentence
CODES = {'<PAD>': 0, '<EOS>': 1, '<UNK>': 2, '<GO>': 3 }


# Loading the Data from the File
def load_data(path):
    """
    Load Dataset from File
    """
    input_file = os.path.join(path)
    with open(input_file, 'r', encoding='utf-8') as f:
        data = f.read()


    return data


# Function to Load the Data from the Source Language File , Target Language File,
# Preprocessing the Data by making it Lowercase, and converting it to numeric
# Saving the Data in Preprocess File
def preprocess_and_save_data(source_path, target_path, text_to_ids):
    """
    Preprocess Text Data.  Save to to file.
    """
    # Preprocess
    source_text = load_data(source_path)
    target_text = load_data(target_path)
    print('-'*80,"\n Loading the data from files \n")

    source_text = source_text.lower()
    target_text = target_text.lower()

    source_vocab_to_int, source_int_to_vocab = create_lookup_tables(source_text)
    target_vocab_to_int, target_int_to_vocab = create_lookup_tables(target_text)

    print("Creating the Lookup for Source and Target Files\n")

    source_text, target_text = text_to_ids(source_text, target_text, source_vocab_to_int, target_vocab_to_int)

    # Save Data
    pickle.dump((
        (source_text, target_text),
        (source_vocab_to_int, target_vocab_to_int),
        (source_int_to_vocab, target_int_to_vocab)), open('preprocess.p', 'wb'))


def load_preprocess():
    """
    Load the Preprocessed Training data and return them in batches of <batch_size> or less
    """
    return pickle.load(open('preprocess.p', mode='rb'))


def create_lookup_tables(text):
    """
    Create lookup tables for vocabulary
    """
    vocab = set(text.split())
    vocab_to_int = copy.copy(CODES)


    for v_i, v in enumerate(vocab, len(CODES)):
        vocab_to_int[v] = v_i

    int_to_vocab = {v_i: v for v, v_i in vocab_to_int.items()}

    return vocab_to_int, int_to_vocab


def save_params(params):
```

```python
    """
    Save parameters to file
    """
    pickle.dump(params, open('params.p', 'wb'))


def load_params():
    """
    Load parameters from file
    """
    return pickle.load(open('params.p', mode='rb'))


def batch_data(source, target, batch_size):
    """
    Batch source and target together
    """
    for batch_i in range(0, len(source)//batch_size):
        start_i = batch_i * batch_size
        source_batch = source[start_i:start_i + batch_size]
        target_batch = target[start_i:start_i + batch_size]
        yield np.array(pad_sentence_batch(source_batch)), np.array(pad_sentence_batch(target_batch))


def pad_sentence_batch(sentence_batch):
    """
    Pad sentence with <PAD> id
    """
    max_sentence = max([len(sentence) for sentence in sentence_batch])
    return [sentence + [CODES['<PAD>']] * (max_sentence - len(sentence))
            for sentence in sentence_batch]



```

In [ ]:

```python
# Load data, from the Source File (english)and the Target file (French)

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

Explore the Data

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

In [ ]:

```python
view_sentence_range = (0, 7)

import numpy as np

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

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]]))
```

Solution

Implement Preprocessing Function

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 each sentence from `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 [ ]:

```
# Defining the Function to convert the Text in the source and target files into Id
# In this function , for each line in the file , we replace the word with integer

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)
    """
    # list for Source Text sentences. This will be 2 D list
    source_id_text = []
    for i, line in enumerate(source_text.split('\n')):
        source_id_text.append([])
        for word in line.split():
            source_id_text[i].append(source_vocab_to_int[word])
    print(" The Number of Lines in Source File : ",(i+1))

# adding the EOS word id after each target Sentence
    target_id_text = []
    for i, line in enumerate(target_text.split('\n')):
        target_id_text.append([])
        for word in line.split():
            target_id_text[i].append(target_vocab_to_int[word])
        target_id_text[i].append(target_vocab_to_int['<EOS>'])

    print(" The Number of Lines in Target File : ",(i+1))
    print(" Replacing the Words with Integers in the Files\n")
    return (source_id_text, target_id_text)



```

Preprocess all the data and save it

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

In [ ]:

```
# Perprocessing the Data from the source and target files and storing the numeric version
# to file with parameters
preprocess_and_save_data(source_path,target_path,text_to_ids)
```

In [ ]:

```
# Loading the numeric version of source and output file from the preprocessed the file

import numpy as np


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

In [ ]:

```
# the Unique words in Source and Target Files
print(" The Unique Words in Source and Target Files \n",'-'*80)
print(len(source_vocab_to_int))
print(len(target_vocab_to_int))

# The total sentences in Source  and Taget Files
print(" The sentences in Source and Target Files \n",'-'*80)
print(len(source_int_text))
print(len(target_int_text))
```

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_decoding_input`
- `encoding_layer`
- `decoding_layer_train`
- `decoding_layer_infer`
- `decoding_layer`
- `seq2seq_model`

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.

As Tensorflow verion 1.12 needs more variables , Returning the placeholders in the following the tuple (Input, Targets, Learing Rate, Keep Probability,Target Seq Length, max Target Length,Source Sequence Length)

In [ ]:

```
# Defining a function to create Placeholders for Input,Output
# and other hyperparameters(learning rate, Keep Probability)
# Also The APIs with 1.12 version of tensorflow , needs additional parameters like
# target_squence_length, max_target_length and source_sequence_length , hence creating Placeholders
# for them

def model_inputs():
    """
    Create TF Placeholders for input, targets, learning rate,keep Probability
    return: Tuple (input, targets, learning rate, keep probability,target_sequence_length
    max_target_len,source_sequence_length)
    """
    # Placeholder for inputs,targets,learning rate and Keep probability
    inputs = tf.placeholder(tf.int32, shape=[None,None], name= "input")
    targets = tf.placeholder(tf.int32, shape=[None,None], name= "targets")
    learn_rate = tf.placeholder(tf.float32, name= "learning_rate")
    keep_prob = tf.placeholder(tf.float32, name= "keep_prob")


    target_seq_lenth = tf.placeholder(tf.int32, shape=[None], name= "target_sequence_length")
    max_target_len = tf.reduce_max(target_seq_lenth, name= 'max_target_len')
    source_seq_length = tf.placeholder(tf.int32, shape=[None], name= "source_sequence_length")

    return (inputs, targets, learn_rate, keep_prob,target_seq_lenth,max_target_len,source_seq_length)

```

Process Decoding Input

Implement `process_decoding_input` using TensorFlow to remove the last word id from each batch in `target_data` and concat the GO ID to the begining of each batch.

In [ ]:

```
# function to process the Decoding Input , where in the last eOS word is removed
# and GO Id is added at the begining of the sentence

def process_decoding_input(target_data, target_vocab_to_int, batch_size):
    """
    Preprocessing the target data for encoding
    :param target_data: Target Placehoder
    :param target_vocab_to_int: Dictionary to go from the target words to an id
    :param batch_size: Batch Size
    :return: Preprocessed target data
    """
    # Create a constant tensor with the 'go id'.

    go_id = tf.constant(target_vocab_to_int['<GO>'], shape=(batch_size,1), dtype=tf.int32)

    # Concatenate the vector (without the last word id <EOS> )  with the go ids vector
    processed_input = tf.concat([go_id,target_data[:,:-1]],1)

    return processed_input
```

Encoding

Implement `encoding_layer()` to create a Encoder RNN layer using [`tf.nn.dynamic_rnn()` (https://www.tensorflow.org/api_docs/python/tf/nn/dynamic_rnn)](https://www.tensorflow.org/api_docs/python/tf/nn/dynamic_rnn).

In [ ]:

```python
# Function to create an Encoder RNN

def encoding_layer(rnn_inputs, rnn_size, num_layers, keep_prob,
                   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_vocab_size: vocabulary size of source data
    :param encoding_embedding_size: embedding size of source data
    """
    # function to Build the lstm cell based on the size, and then wrap in dropout
    def build_cell(rnn_size, keep_prob):
        lstm = tf.contrib.rnn.LSTMCell(rnn_size)
        lstm_drop = tf.contrib.rnn.DropoutWrapper(lstm, output_keep_prob=keep_prob)
        return lstm_drop

    # Stack the LSTM cells to create a stacked lstm
    stacked_lstm = tf.contrib.rnn.MultiRNNCell([build_cell(rnn_size, keep_prob) for _ in range(num_layers)])

    # Create embedding layer.
    embed_encoder = tf.contrib.layers.embed_sequence(rnn_inputs, vocab_size = source_vocab_size, embed_dim = enc

    # calling the dynamic RNN
    # If we don't have an initial zero state, provide a dtype.
    output, state = tf.nn.dynamic_rnn(stacked_lstm, embed_encoder,  dtype=tf.float32)
    return (output, state)
```

Decoding - Training

Create training logits using `tf.contrib.seq2seq.simple_decoder_fn_train()` (https://www.tensorflow.org/api_docs/python/tf/contrib/seq2seq/simple_decoder_fn_train) and `tf.contrib.seq2seq.dynamic_rnn_decoder()` (https://www.tensorflow.org/api_docs/python/tf/contrib/seq2seq/dynamic_rnn_decoder). Apply the `output_fn` to the `tf.contrib.seq2seq.dynamic_rnn_decoder()` (https://www.tensorflow.org/api_docs/python/tf/contrib/seq2seq/dynamic_rnn_decoder) outputs.

Please note :This API's were available with version 1.0 , but with the tensorflow version 1.12 , these API's have been changed and hence new API's implemented instead

In [ ]:

```
# creating a Decoding Function
# Please Note : The simple_Decoder_fn_train function was in 1.0 TF version , but has been replaced
# by BasicDecoder, so hence used here

def decoding_layer_train(encoder_state, dec_cell, dec_embed_input,
                         target_sequence_length, max_summary_length,
                         output_fn, 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_fn: Function to apply the output fn
    :param keep_prob: Dropout keep probability
    :return: BasicDecoderOutput containing training logits and sample_id
    """
    # TODO: Implement Function
    trainig_helper = tf.contrib.seq2seq.TrainingHelper(dec_embed_input, target_sequence_length)
    basic_decoder = tf.contrib.seq2seq.BasicDecoder(dec_cell, trainig_helper, encoder_state, output_fn)
    train_logits, _, _ = tf.contrib.seq2seq.dynamic_decode(basic_decoder,maximum_iterations=max_summary_length)
    return train_logits


```

Decoding - Inference

Create inference logits using `tf.contrib.seq2seq.simple_decoder_fn_inference()` (https://www.tensorflow.org/api_docs/python/tf/contrib/seq2seq/simple_decoder_fn_inference) and `tf.contrib.seq2seq.dynamic_rnn_decoder()` (https://www.tensorflow.org/api_docs/python/tf/contrib/seq2seq/dynamic_rnn_decoder).

Please Note : The above API's not available with version 1.12 , so instead latest API's used for this code

In [ ]:

```python
# function to create Decoding Inference

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
    """
    # Convert the start_ids to be a vector with batch size (the go id repeated batch size times)
    start_ids = tf.tile([start_of_sequence_id], [batch_size])
    # Create the embedding helper.
    embedding_helper = tf.contrib.seq2seq.GreedyEmbeddingHelper(dec_embeddings, start_ids, end_of_sequence_id)
    basic_decoder = tf.contrib.seq2seq.BasicDecoder(dec_cell, embedding_helper, encoder_state, output_layer)
    Inference_logits, _, _ = tf.contrib.seq2seq.dynamic_decode(basic_decoder,maximum_iterations=max_target_seque

    return Inference_logits

```

Build the Decoding Layer

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

- Create RNN cell for decoding using `rnn_size` and `num_layers` .
- Create the output fuction using `lambda` (https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions) to transform it's input, logits, to class logits.

- Use the your `decoding_layer_train(encoder_state, dec_cell, dec_embed_input, sequence_length, decoding_scope, output_fn, 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, maximum_length, vocab_size, decoding_scope, output_fn, keep_prob)` function to get the inference logits.

Note: You'll need to use tf.variable_scope (https://www.tensorflow.org/api_docs/python/tf/variable_scope) to share variables between training and inference.

Please note: new API's used as the above APi's available with tensorflow version 1.0 , hence following new steps

- Created a RNN MultiCell with rnn_size and num_layers.
- Used Decoding Lookup Tables to get back the Decoded embedded input
- Used Decoding_layer_train to get training logits
- Used DEcoding_layer_infer to get Inference Logits
- Used scope.reuse_Variables() to reuse the variables between trainign and inference

In [ ]:

```
# To Implement the Decoding Layer to create Decoder RNN layer
from tensorflow.python.layers.core import Dense

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
    :param decoding_embedding_size: Decoding embedding size
    :return: Tuple of (Training BasicDecoderOutput, Inference BasicDecoderOutput)
    """
    # Using  the same proess as in the encoding layer.
    def build_cell(rnn_size, keep_prob):
        lstm = tf.contrib.rnn.LSTMCell(rnn_size)
        lstm_drop = tf.contrib.rnn.DropoutWrapper(lstm, output_keep_prob=keep_prob)
        return lstm_drop

    # Stack them all
    stacked_lstm = tf.contrib.rnn.MultiRNNCell([build_cell(rnn_size, keep_prob) for _ in range(num_layers)])

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

    dense_layer = Dense(target_vocab_size,
                         kernel_initializer = tf.truncated_normal_initializer(mean = 0.0, stddev=0.1))

    with tf.variable_scope("decode") as scope:
        tr_decoder_output = decoding_layer_train(
            encoder_state, stacked_lstm, dec_embed_input,
            target_sequence_length, max_target_sequence_length,
            dense_layer, keep_prob)

    # reusing the Variables being shared between training and inference phases
        scope.reuse_variables()
        inf_decoder_output = decoding_layer_infer(
            encoder_state, stacked_lstm, dec_embeddings,
            target_vocab_to_int['<GO>'], target_vocab_to_int['<EOS>'],
            max_target_sequence_length, target_vocab_size,
            dense_layer, batch_size, keep_prob)

    return tr_decoder_output, inf_decoder_output
```

Build the Neural Network

Apply the functions you implemented above to:

- Encode the input using your `encoding_layer(rnn_inputs, rnn_size, num_layers, keep_prob)` .
- Process target data using your `process_decoding_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_embed_input, dec_embeddings, encoder_state, vocab_size, sequence_length, rnn_size, num_layers, target_vocab_to_int, keep_prob)` .

In [ ]:

```
# building the sequence to Sequence Model

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)
    """
    output, state = encoding_layer(input_data, rnn_size, num_layers, keep_prob,
                                   source_vocab_size,enc_embedding_size)

    processed_input = process_decoding_input(target_data, target_vocab_to_int, batch_size)

    tr_decoder_output, inf_decoder_output = decoding_layer(processed_input, 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)

    return tr_decoder_output, inf_decoder_output

```

Neural Network Training

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

In [ ]:

```
# setting the Hyperparameters

# Number of Epochs
epochs = 10

# Batch Size
batch_size = 512

# RNN Size
rnn_size = 128

# Number of Layers
num_layers = 2

# Embedding Size
encoding_embedding_size = 128
decoding_embedding_size = 128


# Learning Rate
learning_rate = 0.01

# Dropout Keep Probability
keep_probability = 0.5
display_step = True
```

Build the Graph

Build the graph using the neural network you implemented.

In [ ]:

```
import tensorflow as tf

# Path for saving the checkpoints

save_path = 'checkpoints/dev'

# Loading the preprocessed source, targetfiles
(source_int_text, target_int_text), (source_vocab_to_int, target_vocab_to_int), _ = load_preprocess()

max_target_sentence_length = max([len(sentence) for sentence in source_int_text])

train_graph = tf.Graph()
with train_graph.as_default():
    # getting the inputs
    input_data, targets, lr, keep_prob,target_sequence_length, max_target_sequence_length, source_sequence_lengt

    #sequence_length = tf.placeholder_with_default(max_target_sentence_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,
                                                   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 if grad is not Non
        train_op = optimizer.apply_gradients(capped_gradients)
```

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 [ ]:

```
# Training the model

# funtion to get if prediction is accurate
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))

# Splitting the 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 ) =  next(batch_data(valid_source,valid_target,batch_size))
# getting the target and source lengths for validation set
valid_targets_lengths = []
for sen in valid_targets_batch:
    valid_targets_lengths.append(len(sen))

valid_sources_lengths = []
for sen in valid_sources_batch:
    valid_sources_lengths.append(len(sen))



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) in enumerate(batch_data(train_source, train_target, batch_size
            # Need the lengths for the _lengths parameters
            targets_lengths = []
            for sen in target_batch:
                targets_lengths.append(len(sen))

            sources_lengths = []
            for sen in source_batch:
                sources_lengths.append(len(sen))


            _, 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 Accuracy: {:>6.4f}, Los
                      .format(epoch_i, batch_i, len(source_int_text) // batch_size, train_acc, valid_acc, loss))

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

Save Parameters

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

In [ ]:

```
# save the parameters to be used later
save_params(save_path)
```

In [ ]:

```
# Loading the parameters from the checkpoint

import tensorflow as tf
import numpy as np

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

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 [ ]:

```python
# Function to preprocess the sentence in English Language

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
    """
    lower_case_words = [word.lower() for word in sentence.split()]

    word_id = [vocab_to_int.get(word, vocab_to_int['<UNK>']) for word in lower_case_words]

    return word_id
```

Translate

This will translate `translate_sentence` from English to French.

In [ ]:

```python
# To check if the model is translating the sentence
translate_sentence = 'it is cold in winter but summer somtimes hot .'


"""
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)*2]*batch_size,
                                         source_sequence_length: [len(translate_sentence)]*batch_size,
                                         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])))
```