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)
```

Explore the Data

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

In [10]:

```
view_sentence_range = (1100, 1110)

"""
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()})))

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_ran
ge[1]]))
print()
print('French sentences {} to {}:'.format(*view_sentence_range))
print('\n'.join(target_text.split('\n')[view_sentence_range[0]:view_sentence_ran
ge[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 1100 to 1110:
he likes grapes , mangoes , and bananas.

French sentences 1100 to 1110:
il aime les raisins , les mangues et les bananes .
```

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 [13]:

```
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)
    """
    # TODO: Implement Function
    source_id_text = [[source_vocab_to_int[word] 
                       for word in sentence.split()]
                       for sentence in source_text.split('\n')]
    target_id_text = [[target_vocab_to_int[word] for word in sentence.split()] +
                      [target_vocab_to_int['<EOS>'] ]
                       for sentence in target_text.split('\n')]
    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
```

Preprocess all the data and save it

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

In [14]:

```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
helper.preprocess_and_save_data(source_path, target_path, text_to_ids)
```

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 [15]:

```
"""
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),
_ = helper.load_preprocess()
```

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 [16]:

```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
from distutils.version import LooseVersion
import warnings
import tensorflow as tf

# Check TensorFlow Version
assert LooseVersion(tf.__version__) in [LooseVersion('1.0.0'),
LooseVersion('1.0.1')], 'This project requires TensorFlow version 1.0  You are
using {}'.format(tf.__version__)
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: 1.0.0
C:\Users\VadymSerpak\Anaconda3\envs\Python35\lib\site-
packages\ipykernel_launcher.py:14: UserWarning: No GPU found. Please use a GPU to
train your neural network.
```

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.

Return the placeholders in the following the tuple (Input, Targets, Learing Rate, Keep Probability)

In [17]:

```
def model_inputs():
    """
    Create TF Placeholders for input, targets, and learning rate.
    :return: Tuple (input, targets, learning rate, keep probability)
    """
    # TODO: Implement Function

    inputs = tf.placeholder(tf.int32, shape=(None, None), name='input') # Perhaps inputs would have been better name??
    targets = tf.placeholder(tf.int32, shape=(None, None))
    learning_rate = tf.placeholder(tf.float32)
    keep_prob = tf.placeholder(tf.float32, name='keep_prob')
    return (inputs, targets, learning_rate, keep_prob)

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

```
Tests Passed
```

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 beginning of each batch.

In [18]:

```
def process_decoding_input(target_data, target_vocab_to_int, batch_size):
    """
    Preprocess target data for dencoding
    :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
    """
    # TODO: Implement Function
    # Remove last word id
    stride_slice = tf.strided_slice(target_data,
                                    begin=[0, 0],
                                    end=[batch_size, -1],
                                    strides=[1, 1])
     # Concat GO id
    concat = tf.concat([tf.fill([batch_size, 1],
                                target_vocab_to_int['<GO>']),
                        stride_slice], 1)

    return concat

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

```
Tests Passed
```

Encoding

Implement `encoding_layer()` to create a Encoder RNN layer using `tf.nn.dynamic_rnn()`.

In [19]:

```
def encoding_layer(rnn_inputs, rnn_size, num_layers, keep_prob):
    """
    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
    :return: RNN state
    """
    # TODO: Implement Function
    cell =
tf.contrib.rnn.MultiRNNCell([tf.contrib.rnn.BasicLSTMCell(rnn_size)]*num_layers)
    drop = tf.contrib.rnn.DropoutWrapper(cell, input_keep_prob=1,
output_keep_prob=keep_prob)
    rnn_out, rnn_state = tf.nn.dynamic_rnn(cell=drop, inputs=rnn_inputs,
dtype=tf.float32)
    return rnn_state
```

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

```
Tests Passed
```

Decoding - Training

Create training logits using `tf.contrib.seq2seq.simple_decoder_fn_train()` and `tf.contrib.seq2seq.dynamic_rnn_decoder()`. Apply the `output_fn` to the `tf.contrib.seq2seq.dynamic_rnn_decoder()` outputs.

In [22]:

```
def decoding_layer_train(encoder_state, dec_cell, dec_embed_input,
sequence_length, decoding_scope,
                         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 sequence_length: Sequence Length
    :param decoding_scope: TenorFlow Variable Scope for decoding
    :param output_fn: Function to apply the output layer
    :param keep_prob: Dropout keep probability
    :return: Train Logits
    """
    # TODO: Implement Function
    train_decoder_fn = tf.contrib.seq2seq.simple_decoder_fn_train(encoder_state)
    train_pred, _, _ = tf.contrib.seq2seq.dynamic_rnn_decoder(cell=dec_cell,

decoder_fn=train_decoder_fn,

inputs=dec_embed_input,

sequence_length=sequence_length,

scope=decoding_scope)
    train_logits = output_fn(train_pred)
    return train_logits

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

```
Tests Passed
```

Decoding - Inference

Create inference logits using `tf.contrib.seq2seq.simple_decoder_fn_inference()` and `tf.contrib.seq2seq.dynamic_rnn_decoder()`.

In [24]:

```
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):
    """
    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 maximum_length: The maximum allowed time steps to decode
    :param vocab_size: Size of vocabulary
    :param decoding_scope: TensorFlow Variable Scope for decoding
    :param output_fn: Function to apply the output layer
    :param keep_prob: Dropout keep probability
    :return: Inference Logits
    """
    # TODO: Implement Function
    infer_decoder_fn = tf.contrib.seq2seq.simple_decoder_fn_inference(output_fn,

encoder_state,

dec_embeddings,

start_of_sequence_id,

end_of_sequence_id,

maximum_length,

vocab_size)

    inference_logits, _, _ = tf.contrib.seq2seq.dynamic_rnn_decoder(dec_cell,

infer_decoder_fn,

scope=decoding_scope)

    return inference_logits

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

```
Tests Passed
```

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` 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 to share variables between training and inference.

In [27]:

```
def decoding_layer(dec_embed_input, dec_embeddings, encoder_state, vocab_size,
sequence_length, rnn_size,
                   num_layers, target_vocab_to_int, keep_prob):
    """
    Create decoding layer
    :param dec_embed_input: Decoder embedded input
    :param dec_embeddings: Decoder embeddings
    :param encoder_state: The encoded state
    :param vocab_size: Size of vocabulary
    :param sequence_length: Sequence Length
    :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 keep_prob: Dropout keep probability
    :return: Tuple of (Training Logits, Inference Logits)
    """
    # TODO: Implement Function
    cell = tf.contrib.rnn.BasicLSTMCell(rnn_size)
    dec_cell = tf.contrib.rnn.MultiRNNCell([cell] * num_layers)

    with tf.variable_scope("decoding") as decoding_scope:
        output_fn = lambda x: tf.contrib.layers.fully_connected(x,
                                                                  vocab_size,
                                                                  None,
scope=decoding_scope)

        train_output = decoding_layer_train(encoder_state,
                                            dec_cell,
                                            dec_embed_input,
                                            sequence_length,
                                            decoding_scope,
                                            output_fn,
                                            keep_prob)

    with tf.variable_scope("decoding", reuse=True) as decoding_scope:
        inf_output = decoding_layer_infer(encoder_state,
                                                      dec_cell,
                                                      dec_embeddings,
                                                      helper.CODES['<GO>'],
                                                      helper.CODES['<EOS>'],
                                                      sequence_length,
                                                      vocab_size,
                                                      decoding_scope,
                                                      output_fn,
                                                      keep_prob)
    return (train_output, inf_output)

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

```
Tests Passed
```

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)`.
- 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 [28]:

```
def seq2seq_model(input_data, target_data, keep_prob, batch_size,
sequence_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 sequence_length: Sequence Length
    :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 Logits, Inference Logits)
    """
    # TODO: Implement Function
    enc_embed_input = tf.contrib.layers.embed_sequence(ids=input_data,
vocab_size=source_vocab_size,

embed_dim=enc_embedding_size)
    encode_state = encoding_layer(rnn_inputs=enc_embed_input,
                                  rnn_size=rnn_size,
                                  num_layers=num_layers,
                                  keep_prob=keep_prob)

    proc_target_data = process_decoding_input(target_data=target_data,

target_vocab_to_int=target_vocab_to_int, batch_size=batch_size)

    dec_embeddings = tf.Variable(tf.random_uniform([target_vocab_size,
dec_embedding_size]))

    dec_embed_input = tf.nn.embedding_lookup(dec_embeddings, proc_target_data)

    train_logits, infer_logits = decoding_layer(dec_embed_input=dec_embed_input,
                                                dec_embeddings=dec_embeddings,
                                                encoder_state=encode_state,
                                                vocab_size=target_vocab_size,
                                                sequence_length=sequence_length,
                                                rnn_size=rnn_size,
                                                num_layers=num_layers,
                                      target_vocab_to_int=target_vocab_to_int,
                                                keep_prob=keep_prob)

    return train_logits, infer_logits

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

```
Tests Passed
```

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 [29]:

```
# Number of Epochs
epochs = 20
# Batch Size
batch_size = 256
# RNN Size
rnn_size = 30
# Number of Layers
num_layers = 2
# Embedding Size
encoding_embedding_size = 25
decoding_embedding_size = 25
# Learning Rate
learning_rate = 0.005
# Dropout Keep Probability
keep_probability = 0.7
```

Build the Graph

Build the graph using the neural network you implemented.

In [30]:

```
"""
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_preprocess()
max_source_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 = model_inputs()
    sequence_length = tf.placeholder_with_default(max_source_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,
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)

    tf.identity(inference_logits, 'logits')
    with tf.name_scope("optimization"):
        # Loss function
        cost = tf.contrib.seq2seq.sequence_loss(
            train_logits,
            targets,
            tf.ones([input_shape[0], sequence_length]))
        # 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 None]
        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 [31]:

```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
import time

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]), (0,0)],
            'constant')

    return np.mean(np.equal(target, np.argmax(logits, 2)))

train_source = source_int_text[batch_size:]
train_target = target_int_text[batch_size:]

valid_source = helper.pad_sentence_batch(source_int_text[:batch_size])
valid_target = helper.pad_sentence_batch(target_int_text[:batch_size])

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(
                helper.batch_data(train_source, train_target, batch_size)):
            start_time = time.time()
            _, loss = sess.run(
                [train_op, cost],
                {input_data: source_batch,
                 targets: target_batch,
                 lr: learning_rate,
                 sequence_length: target_batch.shape[1],
                 keep_prob: keep_probability})

            batch_train_logits = sess.run(
                inference_logits,
                {input_data: source_batch, keep_prob: 1.0})
            batch_valid_logits = sess.run(
                inference_logits,
                {input_data: valid_source, keep_prob: 1.0})

            train_acc = get_accuracy(target_batch, batch_train_logits)
            valid_acc = get_accuracy(np.array(valid_target), batch_valid_logits)
            end_time = time.time()
            print('Epoch {:>3} Batch {:>4}/{} - Train Accuracy: {:>6.3f},
Validation Accuracy: {:>6.3f}, Loss: {:>6.3f}'
                  .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')
```

```
Epoch   0 Batch    0/538 - Train Accuracy:  0.209, Validation Accuracy:  0.291,
Loss:  5.889
Epoch   0 Batch    1/538 - Train Accuracy:  0.231, Validation Accuracy:  0.316,
Loss:  5.828
.....
Epoch  19 Batch  535/538 - Train Accuracy:  0.958, Validation Accuracy:  0.952,
Loss:  0.030
Epoch  19 Batch  536/538 - Train Accuracy:  0.944, Validation Accuracy:  0.953,
Loss:  0.038
Model Trained and Saved
```

Save Parameters

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

In [32]:

```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
# Save parameters for checkpoint
helper.save_params(save_path)
```

Checkpoint

In [33]:

```
"""
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) = helper.load_preprocess()
load_path = helper.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 [35]:

```
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
    """
    # TODO: Implement Function
    sentence = sentence.lower()
    words = sentence.split()
    word_id_list = [vocab_to_int.get(word, vocab_to_int['<UNK>'])
                    for word in words]

    return word_id_list

"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_sentence_to_seq(sentence_to_seq)
Tests Passed
```

Translate

This will translate `translate_sentence` from English to French.

In [37]:

```
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('logits:0')
    keep_prob = loaded_graph.get_tensor_by_name('keep_prob:0')

    translate_logits = sess.run(logits, {input_data: [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 np.argmax(translate_logits, 1)]))
print('  French Words: {}'.format([target_int_to_vocab[i] for i in
np.argmax(translate_logits, 1)]))
```

```
Input
  Word Ids:      [63, 168, 187, 146, 175, 148, 116]
  English Words: ['he', 'saw', 'a', 'old', 'yellow', 'truck', '.']

Prediction
  Word Ids:      [70, 332, 350, 228, 30, 180, 234, 126, 1]
  French Words: ['il', 'a', 'vu', 'un', 'vieille', 'voiture', 'jaune', '.',
'<EOS>']
```

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 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.

Helper.py

Language Translation

```
import os
import pickle
import copy
import numpy as np

CODES = {'<PAD>': 0, '<EOS>': 1, '<UNK>': 2, '<GO>': 3 }

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

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)

    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)

    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):
    """
    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]
```