In [1]:

```
import tensorflow as tf
import numpy as np
import os, re
from tensorflow.python.layers.core import Dense
```

`load_sentences` reads the text data into a list, where each list element is a sentence, cropped at `MAX_CHAR_PER_LINE` characters

In [2]:

```
MAX_CHAR_PER_LINE = 20

def load_sentences(path):
    with open(path, 'r', encoding="ISO-8859-1") as f:
        data_raw = f.read().encode('ascii', 'ignore').decode('UTF-8').lower()
        data_alpha = re.sub('[^a-z\n]+', ' ', data_raw)
        data = []
        for line in data_alpha.split('\n'):
            data.append(line[:MAX_CHAR_PER_LINE])
    return data
```

`extract_character_vocab` converts a list of of sentences to vocabularies which are `python` dictionaries of `int_to_symbol` {index:character} and `symbol_to_int` {character:index}.

The list comprehension `[character for line in data for character in line]` works like this (probably): `[subListItem for ListItem in List for subListItem in ListItem]`

The creation of the vocabularies work using a dictionary comprehension: `dict = {key:item for key, item in enumerate(list)}`

In [3]:

```python
def extract_character_vocab(data):
    special_symbols = ['<PAD>', '<UNK>', '<GO>',  '<EOS>']
    # extract unique characters from all sentences in data
    set_symbols = set([character for line in data for character in line])
    # add special symbols
    all_symbols = special_symbols + list(set_symbols)
    # create vocabularies that match symbol to index, and index to symbol
    int_to_symbol = {word_i: word for word_i, word in enumerate(all_symbols)}
    symbol_to_int = {word: word_i for word_i, word in int_to_symbol.items()}
    return int_to_symbol, symbol_to_int

input_sentences = load_sentences('data/words_input.txt')
output_sentences = load_sentences('data/words_output.txt')

input_int_to_symbol, input_symbol_to_int = extract_character_vocab(input_sentences)
output_int_to_symbol, output_symbol_to_int = extract_character_vocab(output_sentences)
```

In [4]:
```
1 input_int_to_symbol
```

Out[4]:
```
{0: '<PAD>',
 1: '<UNK>',
 2: '<GO>',
 3: '<EOS>',
 4: 'a',
 5: 'm',
 6: 'q',
 7: 'r',
 8: 'u',
 9: 'l',
 10: 'y',
 11: 'e',
 12: 't',
 13: 'v',
 14: 'z',
 15: 'd',
 16: 'h',
 17: 'f',
 18: 'k',
 19: 's',
 20: 'o',
 21: 'n',
 22: 'p',
 23: 'b',
 24: 'i',
 25: 'g',
 26: 'j',
 27: 'w',
 28: 'x',
 29: 'c',
 30: ' '}
```

In [*]:
```
1 output_int_to_symbol
```

Out[5]:
```
{0: '<PAD>',
 1: '<UNK>',
 2: '<GO>',
 3: '<EOS>',
 4: 'a',
 5: 'm',
 6: 'r',
 7: 'q',
 8: 'u',
 9: 'l',
 10: 'y',
 11: 'e',
 12: 't',
 13: 'v',
 14: 'z',
 15: 'd',
 16: 'h',
 17: 'f',
 18: 'k',
 19: 's',
 20: 'o',
 21: 'n',
 22: 'p',
 23: 'b',
 24: 'g',
 25: 'i',
 26: 'j',
 27: 'w',
 28: 'x',
 29: 'c',
 30: ' '}
```

In [*]:

```
NUM_EPOCS = 300
RNN_STATE_DIM = 512
RNN_NUM_LAYERS = 2
ENCODER_EMBEDDING_DIM = DECODER_EMBEDDING_DIM = 64

BATCH_SIZE = int(32)
LEARNING_RATE = 0.0003

INPUT_NUM_VOCAB = len(input_symbol_to_int)
OUTPUT_NUM_VOCAB = len(output_symbol_to_int)
```

In [*]:

```
# Encoder placeholders
encoder_input_seq = tf.placeholder(
    tf.int32,
    [None, None],
    name='encoder_input_seq'
)

encoder_seq_len = tf.placeholder(
    tf.int32,
    (None,),
    name='encoder_seq_len'
)

# Decoder placeholders
decoder_output_seq = tf.placeholder(
    tf.int32,
    [None, None],
    name='decoder_output_seq'
)

decoder_seq_len = tf.placeholder(
    tf.int32,
    (None,),
    name='decoder_seq_len'
)

max_decoder_seq_len = tf.reduce_max(
    decoder_seq_len,
    name='max_decoder_seq_len'
)
```

In [*]:

```
def make_cell(state_dim):
    lstm_initializer = tf.random_uniform_initializer(-0.1, 0.1)
    return tf.contrib.rnn.LSTMCell(state_dim, initializer=lstm_initializer)

def make_multi_cell(state_dim, num_layers):
    cells = [make_cell(state_dim) for _ in range(num_layers)]
    return tf.contrib.rnn.MultiRNNCell(cells)
```

In [*]:

```
# Encoder embedding

encoder_input_embedded = tf.contrib.layers.embed_sequence(
    encoder_input_seq,
    INPUT_NUM_VOCAB,
    ENCODER_EMBEDDING_DIM
)


# Encoder output

encoder_multi_cell = make_multi_cell(RNN_STATE_DIM, RNN_NUM_LAYERS)

encoder_output, encoder_state = tf.nn.dynamic_rnn(
    encoder_multi_cell,
    encoder_input_embedded,
    sequence_length=encoder_seq_len,
    dtype=tf.float32
)

del(encoder_output)
```

In [*]:

```
decoder_raw_seq = decoder_output_seq[:, :-1]
go_prefixes = tf.fill([BATCH_SIZE, 1], output_symbol_to_int['<GO>'])
decoder_input_seq = tf.concat([go_prefixes, decoder_raw_seq], 1)
```

In [*]:

```
decoder_embedding = tf.Variable(tf.random_uniform([OUTPUT_NUM_VOCAB,
                                                   DECODER_EMBEDDING_DIM]))
decoder_input_embedded = tf.nn.embedding_lookup(decoder_embedding,
                                                decoder_input_seq)

decoder_multi_cell = make_multi_cell(RNN_STATE_DIM, RNN_NUM_LAYERS)

output_layer_kernel_initializer = tf.truncated_normal_initializer(mean=0.0, stddev=0.1)
output_layer = Dense(
    OUTPUT_NUM_VOCAB,
    kernel_initializer = output_layer_kernel_initializer
)
```

In [*]:

```
with tf.variable_scope("decode"):

    training_helper = tf.contrib.seq2seq.TrainingHelper(
        inputs=decoder_input_embedded,
        sequence_length=decoder_seq_len,
        time_major=False
    )

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

    training_decoder_output_seq, _, _ = tf.contrib.seq2seq.dynamic_decode(
        training_decoder,
        impute_finished=True,
        maximum_iterations=max_decoder_seq_len
    )
```

In [*]:

```
with tf.variable_scope("decode", reuse=True):
    start_tokens = tf.tile(
        tf.constant([output_symbol_to_int['<GO>']],
                    dtype=tf.int32),
        [BATCH_SIZE],
        name='start_tokens')

    # Helper for the inference process.
    inference_helper = tf.contrib.seq2seq.GreedyEmbeddingHelper(
        embedding=decoder_embedding,
        start_tokens=start_tokens,
        end_token=output_symbol_to_int['<EOS>']
    )

    # Basic decoder
    inference_decoder = tf.contrib.seq2seq.BasicDecoder(
        decoder_multi_cell,
        inference_helper,
        encoder_state,
        output_layer
    )

    # Perform dynamic decoding using the decoder
    inference_decoder_output_seq, _, _ = tf.contrib.seq2seq.dynamic_decode(
        inference_decoder,
        impute_finished=True,
        maximum_iterations=max_decoder_seq_len
    )
```

In [*]:

```
# rename the tensor for our convenience
training_logits = tf.identity(training_decoder_output_seq.rnn_output, name='logits')
inference_logits = tf.identity(inference_decoder_output_seq.sample_id, name='predictions')

# Create the weights for sequence_loss
masks = tf.sequence_mask(
    decoder_seq_len,
    max_decoder_seq_len,
    dtype=tf.float32,
    name='masks'
)

cost = tf.contrib.seq2seq.sequence_loss(
    training_logits,
    decoder_output_seq,
    masks
)
```

In [*]:

```
optimizer = tf.train.AdamOptimizer(LEARNING_RATE)

gradients = optimizer.compute_gradients(cost)
capped_gradients = [(tf.clip_by_value(grad, -5., 5.), var)
                        for grad, var in gradients if grad is not None]
train_op = optimizer.apply_gradients(capped_gradients)
```

In [*]:

```
def pad(xs, size, pad):
    return xs + [pad] * (size - len(xs))
```

In [*]:

```
input_seq = [
    [input_symbol_to_int.get(symbol, input_symbol_to_int['<UNK>'])
        for symbol in line]
    for line in input_sentences
]

output_seq = [
    [output_symbol_to_int.get(symbol, output_symbol_to_int['<UNK>'])
        for symbol in line] + [output_symbol_to_int['<EOS>']]
    for line in output_sentences
]

sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
saver = tf.train.Saver()

for epoch in range(NUM_EPOCS + 1):
    for batch_idx in range(len(input_sentences) // BATCH_SIZE):

        input_batch, input_lengths, output_batch, output_lengths = [], [], [], []
        for sentence in input_sentences[batch_idx:batch_idx + BATCH_SIZE]:
            symbol_sent = [input_symbol_to_int[symbol] for symbol in sentence]
            padded_symbol_sent = pad(symbol_sent, MAX_CHAR_PER_LINE, input_symbol_to_int['<PAD>'])
            input_batch.append(padded_symbol_sent)
            input_lengths.append(len(sentence))
        for sentence in output_sentences[batch_idx:batch_idx + BATCH_SIZE]:
            symbol_sent = [output_symbol_to_int[symbol] for symbol in sentence]
            padded_symbol_sent = pad(symbol_sent, MAX_CHAR_PER_LINE, output_symbol_to_int['<PAD>'])
            output_batch.append(padded_symbol_sent)
            output_lengths.append(len(sentence))

        _, cost_val = sess.run(
            [train_op, cost],
            feed_dict={
                encoder_input_seq: input_batch,
                encoder_seq_len: input_lengths,
                decoder_output_seq: output_batch,
                decoder_seq_len: output_lengths
            }
        )

```

```
        if batch_idx % 629 == 0:
            print('Epcoh {}. Batch {}/{}. Cost {}'.format(epoch, batch_idx, len(input_sentences) // BATCH_
    saver.save(sess, 'model.ckpt')
sess.close()
```

```
Epcoh 0. Batch 0/6919. Cost 3.582240104675293
Epcoh 0. Batch 629/6919. Cost 0.8955135941505432
Epcoh 0. Batch 1258/6919. Cost 0.7081497311592102
Epcoh 0. Batch 1887/6919. Cost 0.7692012786865234
Epcoh 0. Batch 2516/6919. Cost 0.8782562017440796
Epcoh 0. Batch 3145/6919. Cost 0.699303925037384
Epcoh 0. Batch 3774/6919. Cost 0.726224958896637
Epcoh 0. Batch 4403/6919. Cost 0.6380016207695007
Epcoh 0. Batch 5032/6919. Cost 0.6510648727416992
Epcoh 0. Batch 5661/6919. Cost 0.6080302596092224
Epcoh 0. Batch 6290/6919. Cost 0.5920670628547668
```

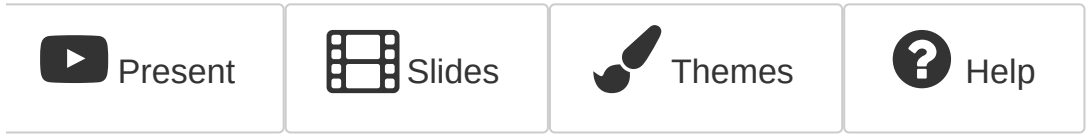
In [*]:

```
sess = tf.InteractiveSession()
saver.restore(sess, 'model.ckpt')

example_input_sent = "do you want to play games"
example_input_symb = [input_symbol_to_int[symbol] for symbol in example_input_sent]
example_input_batch = [pad(example_input_symb, MAX_CHAR_PER_LINE, input_symbol_to_int['<PAD>'])] * BATCH_S
example_input_lengths = [len(example_input_sent)] * BATCH_SIZE

output_ints = sess.run(inference_logits, feed_dict={
    encoder_input_seq: example_input_batch,
    encoder_seq_len: example_input_lengths,
    decoder_seq_len: example_input_lengths
})[0]

output_str = ''.join([output_int_to_symbol[i] for i in output_ints])
print(output_str)
```


Present
Slides
Themes
Help