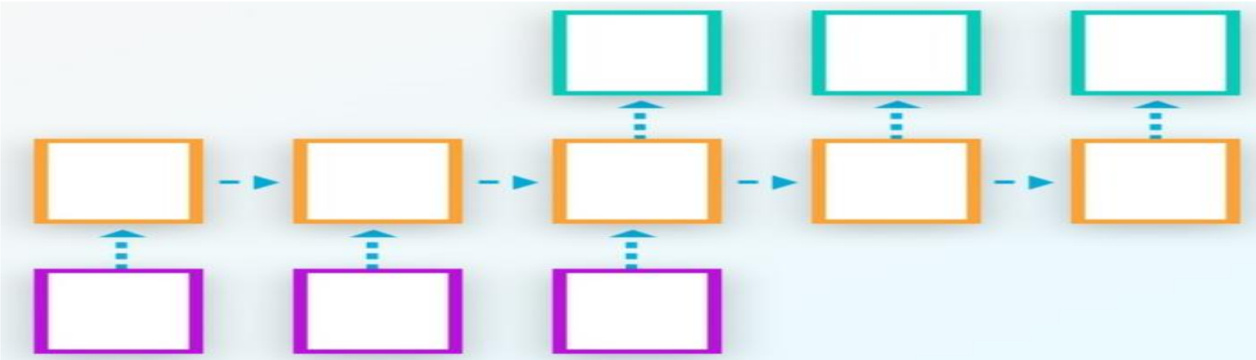
Character Sequence to Sequence

In this notebook, we'll build a model that takes in a sequence of letters, and outputs a sorted version of that sequence. We'll do that using what we've learned so far about Sequence to Sequence models.



Dataset

The dataset lives in the /data/ folder. At the moment, it is made up of the following files:

- **letters_source.txt**: The list of input letter sequences. Each sequence is its own line.
- **letters_target.txt**: The list of target sequences we'll use in the training process. Each sequence here is a response to the input sequence in letters_source.txt with the same line number.

In [1]:

```
import helper

source_path = 'data/letters_source.txt'
target_path = 'data/letters_target.txt'

source_sentences = helper.load_data(source_path)
target_sentences = helper.load_data(target_path)
```

Let's start by examining the current state of the dataset. `source_sentences` contains the entire input sequence file as text delimited by newline symbols.

In [2]:

```
source_sentences[:50].split('\n')
```

Out[2]:

```
['bsaqq',
 'npy',
 'lbwuj',
 'bqv',
 'kial',
 'tddam',
 'edxpjpg',
 'nspv',
 'huloz',
 '']
```

`target_sentences` contains the entire output sequence file as text delimited by newline symbols. Each line corresponds to the line from `source_sentences`. `target_sentences` contains a sorted characters of the line.

In [3]:

```
target_sentences[:50].split('\n')
```

Out[3]:

```
['abqqs',
 'npy',
 'bjluw',
 'bqv',
 'aikl',
 'addmt',
 'degjppx',
 'npsv',
 'hlouz',
 '']
```

Preprocess

To do anything useful with it, we'll need to turn the characters into a list of integers:

In [4]:

```
def extract_character_vocab(data):
    special_words = ['<pad>', '<unk>', '<s>',  '<\s>']

    set_words = set([character for line in data.split('\n') for character in 
line])
    int_to_vocab = {word_i: word for word_i, word in enumerate(special_words + 
list(set_words))}
    vocab_to_int = {word: word_i for word_i, word in int_to_vocab.items()}

    return int_to_vocab, vocab_to_int

# Build int2letter and letter2int dicts
source_int_to_letter, source_letter_to_int = 
extract_character_vocab(source_sentences)
target_int_to_letter, target_letter_to_int = 
extract_character_vocab(target_sentences)

# Convert characters to ids
source_letter_ids = [[source_letter_to_int.get(letter, 
source_letter_to_int['<unk>']) for letter in line] for line in 
source_sentences.split('\n')]
target_letter_ids = [[target_letter_to_int.get(letter, 
target_letter_to_int['<unk>']) for letter in line] for line in 
target_sentences.split('\n')]

print("Example source sequence")
print(source_letter_ids[:3])
print("\n")
print("Example target sequence")
print(target_letter_ids[:3])
```

```
Example source sequence
[[14, 4, 5, 26, 26], [22, 6, 17], [28, 14, 11, 27, 19]]
Example target sequence
[[5, 14, 26, 26, 4], [22, 6, 17], [14, 19, 28, 27, 11]]
```

The last step in the preprocessing stage is to determine the the longest sequence size in the dataset we'll be using, then pad all the sequences to that length.

In [5]:

```
def pad_id_sequences(source_ids, source_letter_to_int, target_ids,
target_letter_to_int, sequence_length):
    new_source_ids = [sentence + [source_letter_to_int['<pad>']] *
(sequence_length - len(sentence)) \
                      for sentence in source_ids]
    new_target_ids = [sentence + [target_letter_to_int['<pad>']] *
(sequence_length - len(sentence)) \
                      for sentence in target_ids]

    return new_source_ids, new_target_ids


# Use the longest sequence as sequence length
sequence_length = max(
        [len(sentence) for sentence in source_letter_ids] + [len(sentence) for
sentence in target_letter_ids])

# Pad all sequences up to sequence length
source_ids, target_ids = pad_id_sequences(source_letter_ids,
source_letter_to_int,
                                          target_letter_ids,
target_letter_to_int, sequence_length)

print("Sequence Length")
print(sequence_length)
print("\n")
print("Input sequence example")
print(source_ids[:3])
print("\n")
print("Target sequence example")
print(target_ids[:3])
```

```
Sequence Length
7


Input sequence example
[[14, 4, 5, 26, 26, 0, 0], [22, 6, 17, 0, 0, 0, 0], [28, 14, 11, 27, 19, 0, 0]]


Target sequence example
[[5, 14, 26, 26, 4, 0, 0], [22, 6, 17, 0, 0, 0, 0], [14, 19, 28, 27, 11, 0, 0]]
```

This is the final shape we need them to be in. We can now proceed to building the model.

Model

Check the Version of TensorFlow

This will check to make sure you have the correct version of TensorFlow

In [6]:

```
from distutils.version import LooseVersion
import tensorflow as tf
```

```
# Check TensorFlow Version
assert LooseVersion(tf.__version__) >= LooseVersion('1.0'), 'Please use
TensorFlow version 1.0 or newer'
print('TensorFlow Version: {}'.format(tf.__version__))
TensorFlow Version: 1.0.0
```

Hyperparameters

In [7]:

```
# Number of Epochs
epochs = 60
# Batch Size
batch_size = 128
# RNN Size
rnn_size = 50
# Number of Layers
num_layers = 2
# Embedding Size
encoding_embedding_size = 13
decoding_embedding_size = 13
# Learning Rate
learning_rate = 0.001
```

Input

In [8]:

```
input_data = tf.placeholder(tf.int32, [batch_size, sequence_length])
targets = tf.placeholder(tf.int32, [batch_size, sequence_length])
lr = tf.placeholder(tf.float32)
```

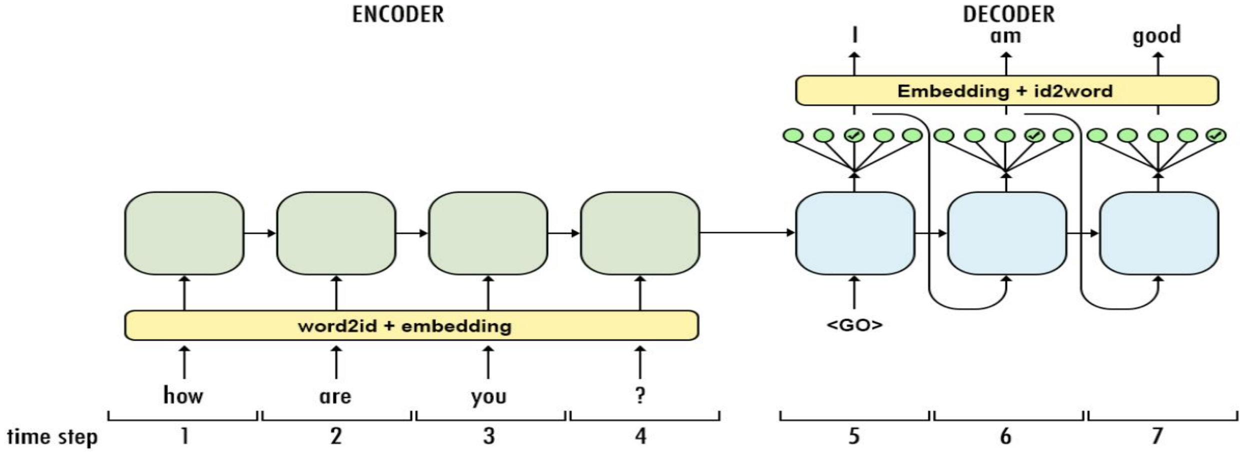
Sequence to Sequence

The decoder is probably the most complex part of this model. We need to declare a decoder for the training phase, and a decoder for the inference/prediction phase. These two decoders will share their parameters (so that all the weights and biases that are set during the training phase can be used when we deploy the model).

First, we'll need to define the type of cell we'll be using for our decoder RNNs. We opted for LSTM.

Then, we'll need to hookup a fully connected layer to the output of decoder. The output of this layer tells us which word the RNN is choosing to output at each time step.

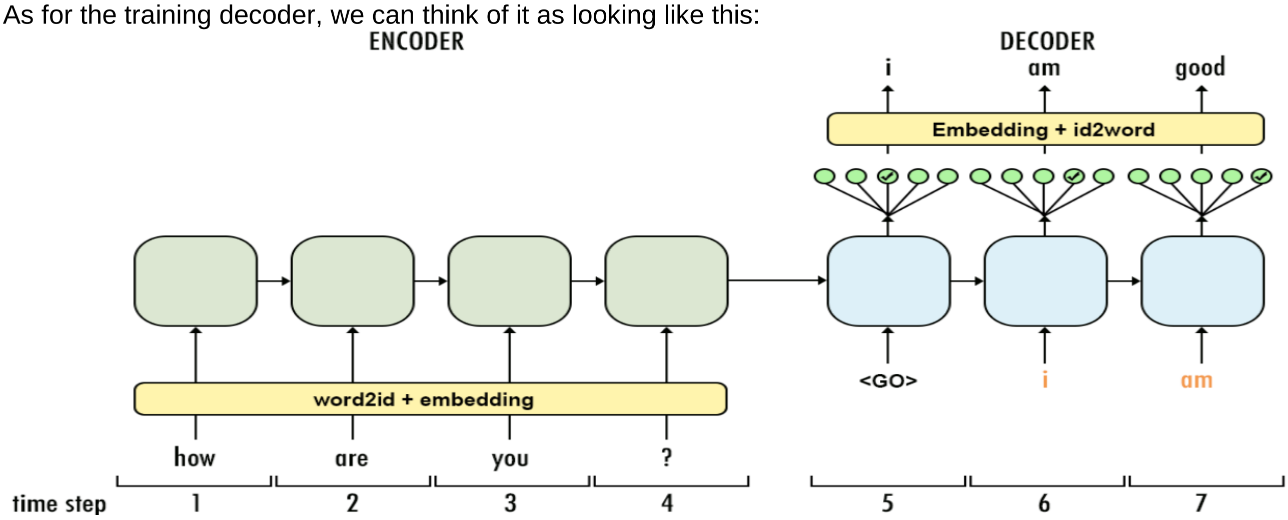
Let's first look at the inference/prediction decoder. It is the one we'll use when we deploy our chatbot to the wild (even though it comes second in the actual code).



We'll hand our encoder hidden state to the inference decoder and have it process its output. TensorFlow handles most of the logic for us. We just have to use `tf.contrib.seq2seq.simple_decoder_fn_inference` and `tf.contrib.seq2seq.dynamic_rnn_decoder` and supply them with the appropriate inputs.

Notice that the inference decoder feeds the output of each time step as an input to the next.

As for the training decoder, we can think of it as looking like this:



The training decoder **does not** feed the output of each time step to the next. Rather, the inputs to the decoder time steps are the target sequence from the training dataset (the orange letters).

Encoding

- Embed the input data using `tf.contrib.layers.embed_sequence`
- Pass the embedded input into a stack of RNNs. Save the RNN state and ignore the output.

In [9]:

```
source_vocab_size = len(source_letter_to_int)

# Encoder embedding
enc_embed_input = tf.contrib.layers.embed_sequence(input_data,
source_vocab_size, encoding_embedding_size)

# Encoder
enc_cell = tf.contrib.rnn.MultiRNNCell([tf.contrib.rnn.BasicLSTMCell(rnn_size)]
* num_layers)
_, enc_state = tf.nn.dynamic_rnn(enc_cell, enc_embed_input, dtype=tf.float32)
```

Process Decoding Input

In [10]:

```
import numpy as np

# Process the input we'll feed to the decoder
ending = tf.strided_slice(targets, [0, 0], [batch_size, -1], [1, 1])
dec_input = tf.concat([tf.fill([batch_size, 1], target_letter_to_int['<s>']),
ending], 1)

demonstration_outputs = np.reshape(range(batch_size * sequence_length),
(batch_size, sequence_length))
sess = tf.InteractiveSession()
```

```
print("Targets")
print(demonstration_outputs[:2])
print("\n")
print("Processed Decoding Input")
print(sess.run(dec_input, {targets: demonstration_outputs})[:2])
```

```
Targets
[[ 0  1  2  3  4  5  6]
 [ 7  8  9 10 11 12 13]]


Processed Decoding Input
[[ 2  0  1  2  3  4  5]
 [ 2  7  8  9 10 11 12]]
```

Decoding

- Embed the decoding input
- Build the decoding RNNs
- Build the output layer in the decoding scope, so the weight and bias can be shared between the training and inference decoders.

In [11]:

```
target_vocab_size = len(target_letter_to_int)

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

# Decoder RNNs
dec_cell = tf.contrib.rnn.MultiRNNCell([tf.contrib.rnn.BasicLSTMCell(rnn_size)]
* num_layers)

with tf.variable_scope("decoding") as decoding_scope:
    # Output Layer
    output_fn = lambda x: tf.contrib.layers.fully_connected(x,
target_vocab_size, None, scope=decoding_scope)
```

Decoder During Training

- Build the training decoder using `tf.contrib.seq2seq.simple_decoder_fn_train` and `tf.contrib.seq2seq.dynamic_rnn_decoder`.
- Apply the output layer to the output of the training decoder

In [12]:

```
with tf.variable_scope("decoding") as decoding_scope:
    # Training Decoder
    train_decoder_fn = tf.contrib.seq2seq.simple_decoder_fn_train(enc_state)
    train_pred, _, _ = tf.contrib.seq2seq.dynamic_rnn_decoder(
        dec_cell, train_decoder_fn, dec_embed_input, sequence_length,
scope=decoding_scope)

    # Apply output function
    train_logits =  output_fn(train_pred)
```

Decoder During Inference

- Reuse the weights the biases from the training decoder using `tf.variable_scope("decoding", reuse=True)`
- Build the inference decoder using `tf.contrib.seq2seq.simple_decoder_fn_inference` and `tf.contrib.seq2seq.dynamic_rnn_decoder`.
- The output function is applied to the output in this step

In [13]:

```
with tf.variable_scope("decoding", reuse=True) as decoding_scope:
    # Inference Decoder
    infer_decoder_fn = tf.contrib.seq2seq.simple_decoder_fn_inference(
        output_fn, enc_state, dec_embeddings, target_letter_to_int['<s>'],
target_letter_to_int['<\s>'],
        sequence_length - 1, target_vocab_size)
    inference_logits, _, _ = tf.contrib.seq2seq.dynamic_rnn_decoder(dec_cell,
infer_decoder_fn, scope=decoding_scope)
```

Optimization

Our loss function is `tf.contrib.seq2seq.sequence_loss` provided by the tensor flow seq2seq module. It calculates a weighted cross-entropy loss for the output logits.

In [14]:

```
# Loss function
cost = tf.contrib.seq2seq.sequence_loss(
    train_logits,
    targets,
    tf.ones([batch_size, 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

We're now ready to train our model. If you run into OOM (out of memory) issues during training, try to decrease the batch_size.

In [15]:

```
import numpy as np

train_source = source_ids[batch_size:]
train_target = target_ids[batch_size:]

valid_source = source_ids[:batch_size]
valid_target = target_ids[:batch_size]

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)):
        _, loss = sess.run(
            [train_op, cost],
            {input_data: source_batch, targets: target_batch, lr:
learning_rate})
        batch_train_logits = sess.run(
            inference_logits,
            {input_data: source_batch})
        batch_valid_logits = sess.run(
            inference_logits,
            {input_data: valid_source})

        train_acc = np.mean(np.equal(target_batch,
np.argmax(batch_train_logits, 2)))
        valid_acc = np.mean(np.equal(valid_target,
np.argmax(batch_valid_logits, 2)))
        print('Epoch {:>3} Batch {:>4}/{} - Train Accuracy: {:>6.3f},
Validation Accuracy: {:>6.3f}, Loss: {:>6.3f}'
              .format(epoch_i, batch_i, len(source_ids) // batch_size,
train_acc, valid_acc, loss))
```

```
Epoch   0 Batch    0/78 - Train Accuracy:  0.018, Validation Accuracy:  0.018,
Loss:  3.415
Epoch   0 Batch    1/78 - Train Accuracy:  0.374, Validation Accuracy:  0.398,
Loss:  3.387
……..
Epoch  59 Batch   75/78 - Train Accuracy:  1.000, Validation Accuracy:  0.990,
Loss:  0.005
Epoch  59 Batch   76/78 - Train Accuracy:  1.000, Validation Accuracy:  0.996,
Loss:  0.007
```

Prediction

In [16]:

```
input_sentence = 'hello'

input_sentence = [source_letter_to_int.get(word, source_letter_to_int['<unk>'])
for word in input_sentence.lower()]
input_sentence = input_sentence + [0] * (sequence_length - len(input_sentence))
batch_shell = np.zeros((batch_size, sequence_length))
batch_shell[0] = input_sentence
chatbot_logits = sess.run(inference_logits, {input_data: batch_shell})[0]

print('Input')
print('  Word Ids:      {}'.format([i for i in input_sentence]))
print('  Input Words: {}'.format([source_int_to_letter[i] for i in
                                                          input_sentence]))
print('\nPrediction')
print('  Word Ids:      {}'.format([i for i in np.argmax(chatbot_logits, 1)]))
print('  Chatbot Answer Words: {}'.format([target_int_to_letter[i] for i in
np.argmax(chatbot_logits, 1)]))
```

Character Sequence to Sequence

```
Input
  Word Ids:       [20, 18, 28, 28, 10, 0, 0]
  Input Words: ['h', 'e', 'l', 'l', 'o', '<pad>', '<pad>']

Prediction
  Word Ids:       [18, 20, 28, 28, 10, 0, 0]
  Chatbot Answer Words: ['e', 'h', 'l', 'l', 'o', '<pad>', '<pad>']
```