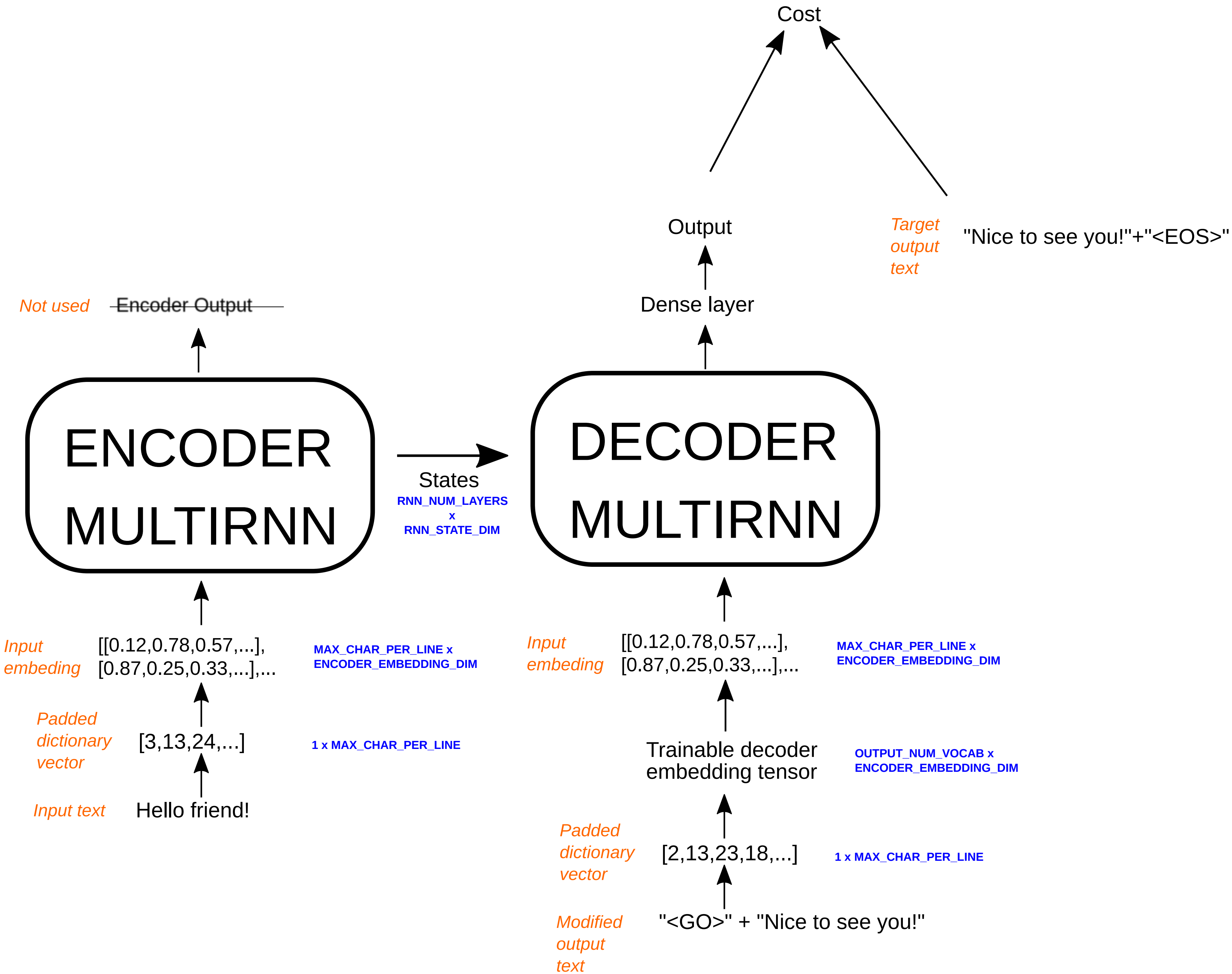
Step by step example

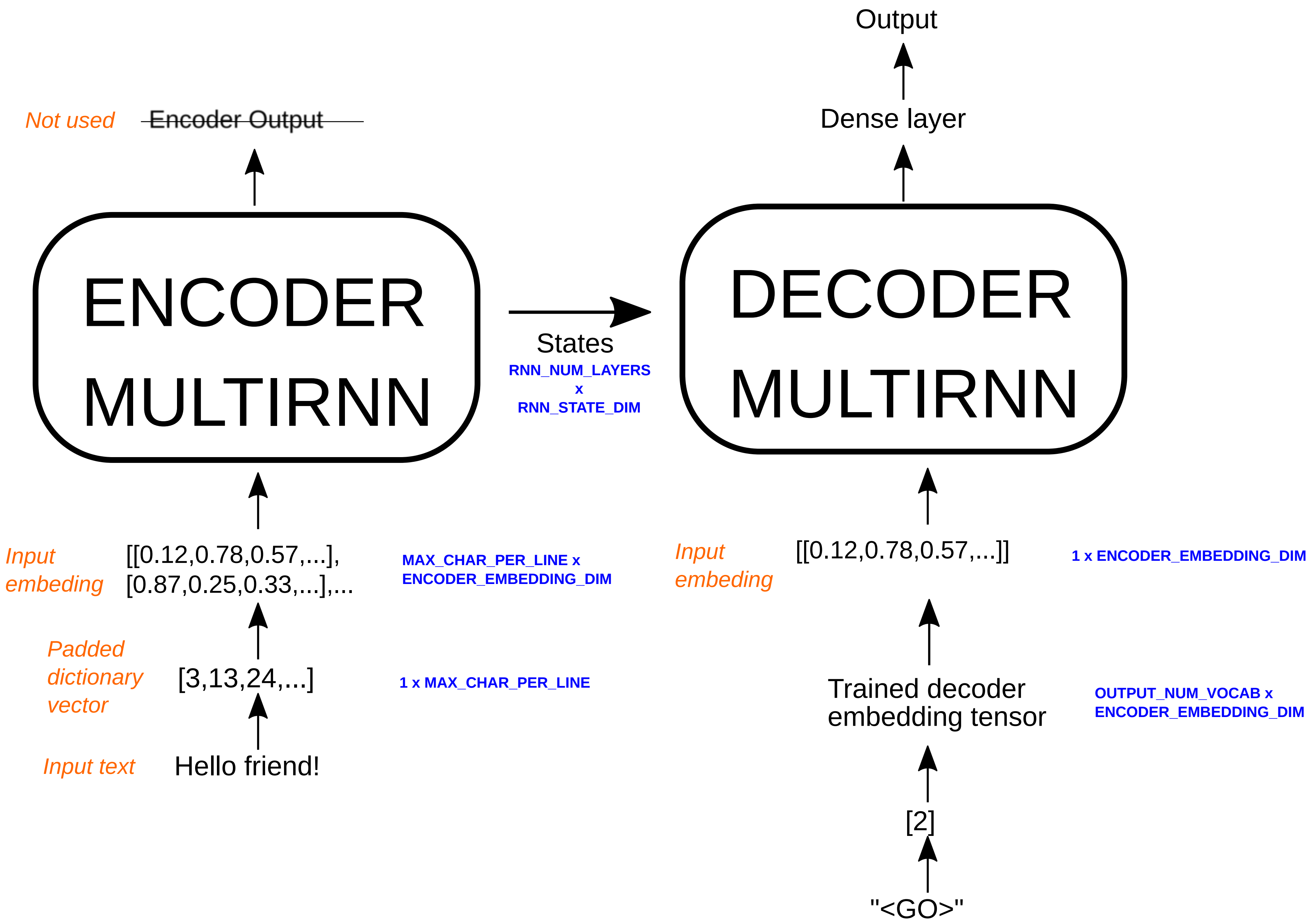
Practical explanation

Overview

Training


Cost
Output
Target output text
"Nice to see you!"+"<EOS>"
Dense layer
Not used
~~Encoder Output~~
ENCODER MULTIRNN
States
RNN_NUM_LAYERS x RNN_STATE_DIM
DECODER MULTIRNN
Input embeding
[[0.12,0.78,0.57,...], [0.87,0.25,0.33,...],...
MAX_CHAR_PER_LINE x ENCODER_EMBEDDING_DIM
Padded dictionary vector
[3,13,24,...]
1 x MAX_CHAR_PER_LINE
Input text
Hello friend!
Input embeding
[[0.12,0.78,0.57,...], [0.87,0.25,0.33,...],...
MAX_CHAR_PER_LINE x ENCODER_EMBEDDING_DIM
Trainable decoder embedding tensor
OUTPUT_NUM_VOCAB x ENCODER_EMBEDDING_DIM
Padded dictionary vector
[2,13,23,18,...]
1 x MAX_CHAR_PER_LINE
Modified output text
"<GO>" + "Nice to see you!"

Inference



Inputs/Outputs

When we train a seq2seq model we pass input and output sentences. These sentences will need to be croped to a maximum length limit (`MAX_CHAR_PER_LINE`) and also each character assigned to a dictionary index.

This is achieved by the functions `load_sentences` and `extract_character_vocab` .

For example if we have sentence 'some sentence' it will become a vector representation with the dictionary indices `[2, 3, 4, 5, 6, ...]` , this vector will have a maximum size of `MAX_CHAR_PER_LINE` .

If sentences are shorter than `MAX_CHAR_PER_LINE` the remainder of the vector representation is filled with the dictionary index for the character `<PAD>` , using the function `pad` .

In the code example below the two sentences are

```
Input sentence = 'she okay'
vector: [20, 30, 22, 5, 23, 12, 7, 19, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Output sentence = 'i hope so'
vector: [13, 5, 30, 23, 27, 22, 5, 20, 23, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Encoder input embeddings

The input at this point is a sequence of integers. Our model will benefit from a denser representation of the input using embeddings compared to one-hot representation of each of the integer.

In the code this embeding is achieved by

```
encoder_input_embedded = tf.contrib.layers.embed_sequence(
ids=padded_symbols_input,  # current input sequence of numbers
vocab_size=INPUT_NUM_VOCAB,  # rows of embedding matrix from dictionary size
embed_dim=ENCODER_EMBEDDING_DIM  # cols of embedding matrix from user
)
```

For the input in the example above it becomes:

```
[[ 0.15645272  0.20985416 -0.10991608 ...  0.09195071  0.05068645 -0.15631753]
 [-0.05581941  0.16376457 -0.12799817 ... -0.14128818 -0.07264952 -0.17910267]
 [-0.1667774  -0.04544044  0.13017449 ...  0.06774965  0.02339229 -0.10449551]
 ...
 [ 0.16037011  0.19128785  0.05270347 ...  0.23134735  0.1546481 -0.18897504]
 [ 0.16037011  0.19128785  0.05270347 ...  0.23134735  0.1546481 -0.18897504]
 [ 0.16037011  0.19128785  0.05270347 ...  0.23134735  0.1546481 -0.18897504]]
```

where each row is a vector of size `ENCODER_EMBEDDING_DIM` representing each integer in

```
[14, 29, 25, 13, 6, 30, 17, 12, 13, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

See https://towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526 (https://towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526) for more

Encoder

The encoder is a multiRNN as explained in `Chapter_11_Section_1_MultiRNN.ipynb` . The functions used to create this multiRNN are `make_cell` , `make_multi_cell`

The output of the encoder is obtained by running the embedded input through `dynamic_rnn` tensorflow function:

```
encoder_output, encoder_state = tf.nn.dynamic_rnn(encoder_multi_cell,
                                    encoder_input_embedded,
                                    sequence_length=(len(padded_symbols_input),),
                                    dtype=tf.float32)
```

For more see `Chapter_11_Section_1_MultiRNN.ipynb`

The encoder output is deleted (`del encoder_output`) and the state is passed into the decoder.

The encoder state has dimensions $\mathrm{RNN\_NUM\_LAYERS} \times \mathrm{RNN\_STATE\_DIM}$ and is printed out in the code example with

```
for i, encoder_state_ in enumerate(encoder_state):
    print('Layer {}, hidden state shape {}'.format(i, encoder_state_[0].shape))
    print('Layer {}, activation state shape {}'.format(i, encoder_state_[1].shape))
```

Decoder

Training input

The training input to the decoder is a modified version of the ouput

```
output_sentence = 'i hope so'+padding
vector: [13, 5, 30, 23, 27, 22, 5, 20, 23, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Where we add the `'<GO>'` vocabulary represenation at the beggining and clip to the `MAX_CHAR_PER_LINE`

```
Input sentence = '<GO>' + clipped_output_sentence
vector: [2, 13, 5, 30, 23, 27, 22, 5, 20, 23, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Training output

For training notice that we attach a `<EOS>` at the end of each sentence.

```
# output_sentence = 'i hope so'+'<EOS>'+padding
symbols_output[-1]=input_symbol_to_int['<EOS>']
```

Inference input

For training it's ok to use a modified version of the output to assist with training (since we have the ground truth) but for inference (final predictions) the input to the decoder will just be the vocabulary representation of `<GO>` . The code to produce the inference inputs is:

```
start_tokens = tf.tile(tf.constant([output_symbol_to_int['<GO>']],
                dtype=tf.int32),[BATCH_SIZE],name='start_tokens')
```

Decoder embedding

The decoder embedding is a trainable tensor. The aim is to learn good embedding representations that based on the encoder state, and the inference input `<GO>` it will produce reasonable outputs for a conversation.

The first dimension of the `decoder_embedding` variable tensor represents the number of entries in our output vocabulary and it's size is `OUTPUT_NUM_VOCAB` , the second dimension is user defined and in the example code is defined in variable `DECODER_EMBEDDING_DIM` . The definition of the encoder embedding in the code is initialized by:

```
decoder_embedding = tf.Variable(tf.random_uniform([OUTPUT_NUM_VOCAB,DECODER_EMBEDDING_DIM]))
```

For a specific input to the decoder, for example for training this would be the vocabulary sequence of `'<GO>' + clipped_output_sentence` , we can extract it's embedding using:

```
decoder_input_embedded = tf.nn.embedding_lookup(decoder_embedding, decoder_input_seq)
```

where `decoder_input_seq` is the vocabulary sequence of `'<GO>' + clipped_output_sentence`

Training Decoder MultiRNN

The decoder is constructed in a more complicated way than the encoder multiRNN. Instead of using `tf.nn.dynamic_rnn` we use `tf.contrib.seq2seq.BasicDecoder`. This is done because in the case of the decoder we need to handle the input states from the encoder, we also need to use a fully connected `Dense` layer to transform the output of the decoder multiRNN to a one-hot representation of the output vocabulary and also `tf.contrib.seq2seq.BasicDecoder` accepts a `tf.contrib.seq2seq.TrainingHelper` which will handle the input to the decoder for us.

multiRNN cell

Same as with the encoder

```
decoder_multi_cell = make_multi_cell(RNN_STATE_DIM, RNN_NUM_LAYERS)
```

Fully connected Dense layer

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

Input training helper

The `tf.contrib.seq2seq.TrainingHelper` will handle the input to the decoder for us

```
training_helper = tf.contrib.seq2seq.TrainingHelper(inputs=decoder_input_embedded,
                sequence_length=(len(padded_symbols_output),), time_major=False)
```

Decoder main

The `tf.contrib.seq2seq.BasicDecoder` is essentially what `tf.nn.dynamic_rnn` was for the encoder. It accepted the multiRNN cells, the `training_helper` that handles the input, the encoder state and the `Dense` output_layer.

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

Decoder outputs

The `tf.contrib.seq2seq.dynamic_decode` produces the outputs of the decoder as `[final_outputs, final_state, final_sequence_lengths]`

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

We can access the ouput from the decoder using `training_decoder_output_seq.rnn_output` which is a tensor with dimensions `[batchSize, length_current_output, size_output_dictionary]`. `length_current_output` is defined by `MAX_CHAR_PER_LINE`.

So for each output element we produce a vector of length `size_output_dictionary` (`[:,i,:]`), from that we select the highest (?) activation to be the vocabulary character to output.

Inference Decoder MultiRNN

The inference decoder is build in a similar way as the training decoder.

```
# Helper for the inference process. It takes the whole of decoder embeddings
# and the integer representation of the <EOS> symbol, which is the `end_token`
# This helper will handle the inputs to the decoder for inference.
inference_helper = tf.contrib.seq2seq.GreedyEmbeddingHelper(embedding=decoder_embedding,
        start_tokens=start_tokens,end_token=output_symbol_to_int['<EOS>'])
# Basic decoder for inference. It uses the decoder MultiRNN `decoder_multi_cell`
# inference_helper takes care of the input (input only the <GO> symbol), the encoder state
# which takes the sentence for which we want to have a reply from our chatbot
# and the output layer that turns the output into a one-hot representation corresponding to
# our vocabulary
inference_decoder = tf.contrib.seq2seq.BasicDecoder(decoder_multi_cell,
    inference_helper,encoder_state,output_layer)
# Perform dynamic decoding using the decoder
# this is the output, same as in the case of the train decoder
inference_decoder_output_seq, _, _ = tf.contrib.seq2seq.dynamic_decode(inference_decoder,
    impute_finished=True,maximum_iterations=MAX_CHAR_PER_LINE)
```

Cost

The ground truth is compared to the output of the model. For this we are going to use `tf.contrib.seq2seq.sequence_loss` which takes the output of the training decoder `training_decoder_output_seq.rnn_output` , the tensor with the ground truth and a mask that controls which parts of the output/ground truth sequences to compare. For example we might want to use the mask hide the `<PAD>` symbol so the model doesn't learn to pad a sentence.

The code to calculate cost of the example is:

```
outputGroundTruth=tf.convert_to_tensor(padded_symbols_output,dtype=tf.int32)
# add batch dummy index
batchGroundTruth=tf.expand_dims(outputGroundTruth,axis=0)
# mask is used to clip the output sequence to its maximum allows size
masks = tf.sequence_mask(tf.constant([len(symbols_output)]),MAX_CHAR_PER_LINE,
                     dtype=tf.float32,name='masks')
cost = tf.contrib.seq2seq.sequence_loss(training_decoder_output_seq.rnn_output,
                                        batchGroundTruth,masks)
```

Code

Import libraries

In [1]:
```
import tensorflow as tf
import numpy as np
import os, re
from tensorflow.python.layers.core import Dense
```

In the example code we are just going to run through a simple input/output example and calculate the output. So first load the data and create a simple example

In [2]:
```
# Functions to load data and create vocabularies
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

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
    # breakdown of syntax -> dict = {key:item for key, item in enumerate(list)}
    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

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

Load data and create vocabularies

In [3]:

```
MAX_CHAR_PER_LINE = 20
# read text as two lists `input_sentences` and `output_sentences`
# the two lists have the same length and they have 1:1 correspondence
# i.e. `input_sentences[0]`->`output_sentences[0]`, `input_sentences[1]`->`output_sentences[1]`
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)
```

Pick two sentences to create an example of input/output

In [4]:

```
# input sentece
# for i in range(100):
#     print('{}, In: {}, out: {}'.format(i, input_sentences[i],output_sentences[i]))
exampleIndex=42
symbols_input = [input_symbol_to_int[symbol] for symbol in input_sentences[exampleIndex]]
padded_symbols_input = pad(symbols_input, MAX_CHAR_PER_LINE, input_symbol_to_int['<PAD>'])
print('Input sentence: {}, vocabulary representation: {}'.format(input_sentences[exampleIndex]
                                                                  , padded_symbols_input))
symbols_output = [input_symbol_to_int[symbol] for symbol in output_sentences[exampleIndex]]
# for training an <EOS> is attached to the end of the output
symbols_output[-1]=input_symbol_to_int['<EOS>']
padded_symbols_output = pad(symbols_output, MAX_CHAR_PER_LINE, input_symbol_to_int['<PAD>'])
print('Output sentence vocabulary representation without padding: {}'.format(symbols_output))
print('Output sentence: {}, vocabulary representation: {}'.format(output_sentences[exampleIndex]
                                                                  , padded_symbols_output))

```

```
Input sentence: she okay , vocabulary representation: [23, 13, 19, 28, 25, 4, 11, 30, 28, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0]
Output sentence vocabulary representation without padding: [14, 28, 13, 25, 24, 19, 28, 23, 25, 3]
Output sentence: i hope so , vocabulary representation: [14, 28, 13, 25, 24, 19, 28, 23, 25, 3, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0]
```

Some constants to be used by our seq2seq model

In [5]:
```python
# number of neurons of LSTM cell
# if we feed a sequence of [batch_size, sequence_length, features] -> [1, 3, 1]
# we will get back [batch_size, sequence_length, RNN_STATE_DIM] -> [1, 3, 512]
# of course if we are interested only in the last output we can get it with [:, -1, :]
RNN_STATE_DIM = 512
# number of layers of the multiRNN cell, see `Concept01_multi_rnn_DC.ipynb`
RNN_NUM_LAYERS = 2
# number of dimensions of embedding layer
ENCODER_EMBEDDING_DIM = DECODER_EMBEDDING_DIM = 64
# sizes of input/output vocabularies
INPUT_NUM_VOCAB = len(input_symbol_to_int)
OUTPUT_NUM_VOCAB = len(output_symbol_to_int)
```

Helper functions to create MultiRNNCell, see Concept01_multi_rnn_DC.ipynb

In [6]:
```python
# create LSTM cell
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)

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

Model

In [7]:

```python
config = tf.ConfigProto(allow_soft_placement = True)
sess = tf.Session(config = config)
with sess:
    # initialise
    sess.run(tf.global_variables_initializer())
    sess.run(tf.local_variables_initializer())
    # Create embeddings from input of sequence of integers
    encoder_input_embedded = tf.contrib.layers.embed_sequence(
    ids=padded_symbols_input,  # input seq of numbers (row indices)
    vocab_size=INPUT_NUM_VOCAB,  # rows of embedding matrix
    embed_dim=ENCODER_EMBEDDING_DIM  # cols of embedding matrix
    )
    sess.run(tf.global_variables_initializer())
    sess.run(tf.local_variables_initializer())
    # **** INPUT EMBEDDING *****
    print('**********************Input****************************************')
    print('Length of input sequence: {}'.format(len(padded_symbols_input)))
    print('*************************Embeded Input*************************************')
    print('Shape of embeded input -> rows=lengthInputSequence={}, \
    columns=userDefined=`ENCODER_EMBEDDING_DIM`={}'.format(encoder_input_embedded.shape[0],
                                    encoder_input_embedded.shape[1]))
    # add degenarate dimension to emulate batch size
    encoder_input_embedded  = tf.expand_dims(encoder_input_embedded, axis=0)
    print('Reshaped embedded input -> [batch_size, length_of_sequence, \
    embedding_dimensions]=[{}, {}, {}]'.format(encoder_input_embedded.shape[0],
    encoder_input_embedded.shape[1], encoder_input_embedded.shape[2]))
    # **** ENCODER *****
    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=(len(padded_symbols_input),),
                                              dtype=tf.float32)
    print('*********************Output-States of Encoder*****************************************')
    for i, temp_encoder_state in enumerate(encoder_state):
        print('Layer {}, hidden state shape {}'.format(i, temp_encoder_state[0].shape))
        print('Layer {}, activation state shape {}'.format(i, temp_encoder_state[1].shape))
    # we don't need the encoder output only the state
    del encoder_output
    # **** DECODER *****
    # For the decoder we need to pass:
    # 1) the states from the encoder
    # 2) the input will be of two kinds: a) one for training that will be the same as the output
    # sequence except the last sequence item will be removed the first sequence item will be a <GO> symbol
    # in front so that the decoder will start producing the output without knowing the output
    #     b)the other kind of input is for the inference stage,this will not  have any information
    #     about the output so it will only be the <GO> symbol
    # remove last symbol from input (which for training is a modified output)
```

```
    decoder_raw_seq = padded_symbols_output[:-1]
    decoder_input_seq = [output_symbol_to_int['<GO>']]+decoder_raw_seq
    print('Unmodified output to decoder (training): {}'.format(padded_symbols_output))
    print('Modified output for input to decoder (training): {}'.format(decoder_input_seq))
    print('********************Trainable decoder embeddings*****************************************')
    # initialize embedding vector representations of the input to the decoder
    # this is initialized to random numbers and will be trained by the seq2seq model
    # it represents every symbol in out vocabulary (row) and it's vector representation (column)
    decoder_embedding = tf.Variable(tf.random_uniform([OUTPUT_NUM_VOCAB,  # number of rows -> total number of output sy
                                                       # in output dictionary
                                                       DECODER_EMBEDDING_DIM # the length of the embedding vectors (hyp
                                                       ]))
    print('Shape of decoder embedings are: row=sizeOutputVocabulary={}, \
    columns=userDefined=lengthOfEmbeddingVectors={}'.format(decoder_embedding.shape[0],
                                                           decoder_embedding.shape[1]))
    # this returns the embedding vectors of the current input defined by `decoder_input_seq`
    # (which is actually the output with <GO> at the front)
    decoder_input_embedded = tf.nn.embedding_lookup(decoder_embedding, decoder_input_seq)
    print('Shape of embeddings for current input sequence: rows=inputSequenceLength={}, \
          columns=userDefined=lengthOfEmbeddingVectors={}'\
          .format(decoder_input_embedded.shape[0], decoder_input_embedded.shape[1]))
    # add degenarate dimension to emulate batch size
    decoder_input_embedded  = tf.expand_dims(decoder_input_embedded, axis=0)
    # Output multiRNN
    decoder_multi_cell = make_multi_cell(RNN_STATE_DIM, RNN_NUM_LAYERS)
    # The output of the decoder will need to be mapped to a one-hot representation of the vocabulary
    # this will also be trained
    # BUT: I'm not sure how the Dense layer output is used since not activation is defined
    # maybe it's just a linear weighted sum... (?)
    output_layer = Dense(OUTPUT_NUM_VOCAB,  # number of symbols in the output dictionary
        kernel_initializer=tf.truncated_normal_initializer(mean=0.0, stddev=0.1))
    # **** TRAINING DECODER *****
    # this function manages the input to the decoder for us
    training_helper = tf.contrib.seq2seq.TrainingHelper(inputs=decoder_input_embedded,
                        sequence_length=(len(padded_symbols_output),), time_major=False)
    # the training decoder will receive both the state from the encoder `encoder_state`
    # and the decoder inputs `decoder_input_embedded` via the training helper
    # this defines the decoder RNN as in the case of `tf.nn.dynamic_rnn`
    # but this is a bit more complicated because of the input/output relationship that is handled by `training_helper`
    # and that we need to pass an `encoder_state` to it, as well as the output Dense layer
    training_decoder = tf.contrib.seq2seq.BasicDecoder(decoder_multi_cell,
        training_helper,encoder_state,output_layer)
    # produces the output of the decoder
    # Specifically it returns [final_outputs, final_state, final_sequence_lengths]
    training_decoder_output_seq, _, _ = tf.contrib.seq2seq.dynamic_decode(training_decoder,
        impute_finished=True, maximum_iterations=len(padded_symbols_output))
```

```
    sess.run(tf.global_variables_initializer())
    sess.run(tf.local_variables_initializer())
    # output of training decoder has shape [shape,numberOfOutputCharacters,sizeOfOutputDictionary]
    # so for each outputCharacter there is a sequence of numbers of
    # size=sizeOfOutputDictionary=OUTPUT_NUM_VOCAB that represent each of the possible
    # characters from the output dictionary
    print('Shape of training decoder -> [batchSize={}, currentOutputCharacters={}, \
          sizeOfOutputDictionary={}]'.format(training_decoder_output_seq.rnn_output.eval().shape[0],
                                            training_decoder_output_seq.rnn_output.eval().shape[1],
                                            training_decoder_output_seq.rnn_output.eval().shape[2]))
    # **** INFERENCE DECODER *****
    # Since we don't have an input to the decoder in INFERENCE we will just input the
    # <GO> token to get the sequence started and start producing the proper output
    BATCH_SIZE=1
    start_tokens = tf.tile(tf.constant([output_symbol_to_int['<GO>']],
                    dtype=tf.int32),[BATCH_SIZE],name='start_tokens')
    # Helper for the inference process. It takes the whole of decoder embeddings
    # and the integer representation of the <EOS> symbol, which is the `end_token`
    # This helper will handle the inputs to the decoder during INFER.
    inference_helper = tf.contrib.seq2seq.GreedyEmbeddingHelper(embedding=decoder_embedding,
            start_tokens=start_tokens,end_token=output_symbol_to_int['<EOS>'])
    # Basic decoder for INFERENCE. It uses the decoder MultiRNN `decoder_multi_cell`
    # inference_helper takes care of the input (input only the <GO> symbol), the encoder state
    # which takes the sentence for which we want to have a reply from our chatbot
    # and the output layer that turns the output into a one-hot representation corresponding to
    # our vocabulary
    inference_decoder = tf.contrib.seq2seq.BasicDecoder(decoder_multi_cell,
        inference_helper,encoder_state,output_layer)
    # Perform dynamic decoding using the decoder
    # this is the output, same as in the case of the TRAIN decoder
    inference_decoder_output_seq, _, _ = tf.contrib.seq2seq.dynamic_decode(inference_decoder,
        impute_finished=True,maximum_iterations=MAX_CHAR_PER_LINE)
    # **** TRAINING COST ****
    # convert output ground truth to tensor
    outputGroundTruth=tf.convert_to_tensor(padded_symbols_output,dtype=tf.int32)
    # add batch dummy index
    batchGroundTruth=tf.expand_dims(outputGroundTruth,axis=0)
    # mask is used to clip the output sequence to it's maximum allows size
    # (tf.reduce_max(tf.constant([MAX_CHAR_PER_LINE])))
    masks = tf.sequence_mask(tf.constant([len(symbols_output)]),MAX_CHAR_PER_LINE,
                             dtype=tf.float32,name='masks')
    print('Mask for current output: {}'.format(masks.eval()))
    cost = tf.contrib.seq2seq.sequence_loss(training_decoder_output_seq.rnn_output,
                                            batchGroundTruth,masks)
    print('Cost between model output estimation and grounf truth = {}'.format(cost.eval()))
```

```
**********************Input****************************************
Length of input sequence: 20
*************************Embeded Input*************************************
Shape of embeded input -> rows=lengthInputSequence=20,     columns=userDefined=`ENCODER_EMBEDDING_DIM`=64
Reshaped embedded input -> [batch_size, length_of_sequence,     embedding_dimensions]=[1, 20, 64]
*********************Output-States of Encoder*****************************************
Layer 0, hidden state shape (1, 512)
Layer 0, activation state shape (1, 512)
Layer 1, hidden state shape (1, 512)
Layer 1, activation state shape (1, 512)
Unmodified output to decoder (training): [14, 28, 13, 25, 24, 19, 28, 23, 25, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Modified output for input to decoder (training): [2, 14, 28, 13, 25, 24, 19, 28, 23, 25, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0]
*********************Trainable decoder embeddings*****************************************
Shape of decoder embedings are: row=sizeOutputVocabulary=31,     columns=userDefined=lengthOfEmbeddingVectors=64
Shape of embeddings for current input sequence: rows=inputSequenceLength=20,          columns=userDefined=lengthOfEmbed
dingVectors=64
Shape of training decoder -> [batchSize=1, currentOutputCharacters=20,          sizeOfOutputDictionary=31]
Mask for current output: [[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
Cost between model output estimation and grounf truth = 3.3847382068634033
```