Creating a Chatbot Using Sequence to Sequence Model

We have covered the theory behind sequence to sequence architecture. Now is the time to see a real world application of sequence to sequence modeling.

In this section we will be using Python's Keras library to develop a sequence to sequence chatbot. The code template for the sequence to sequence model is available at Keras official blog. This code can be used to create any sequence to sequence model. We will use this code to create our Chabot. So, let's start.

Step1: Importing the Library

As always, the first step is to import libraries required to implement the sequence to sequence model. Execute the following script:

```
import os, sys
import re
from keras.models import Model
from keras.layers import Input, LSTM, GRU, Dense, Embedding
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical
import numpy as np
import matplotlib.pyplot as plt
```

Step 2: Configuration Settings

```
BATCH_SIZE = 4
EPOCHS = 7
LSTM_NODES = 256
MAX_SENTENCE_LENGTH = 100
VOCABULARY_SIZE = 5000
VECTOR_DIM = 100
```

Here MAX_SENTENCE_LENGTH will be the maximum length of the sequence. VOCABULAR_SIZE is the number of words that we will use to create embedding layer. The VECTOR_DIM will be the number of features per word in the sentence. Our batch size will be 4 and we will have 7 epochs. Batch size refers to the number of records after which the weights are updated or

backpropagation takes place. An epoch refers to training on whole dataset. Here 7 epochs mean that we will train our algorithm 7 times on the whole dataset. LSTM_NODES are the number of nodes or neurons in our LSTM.

Step 3: Importing and Preprocessing the Dataset

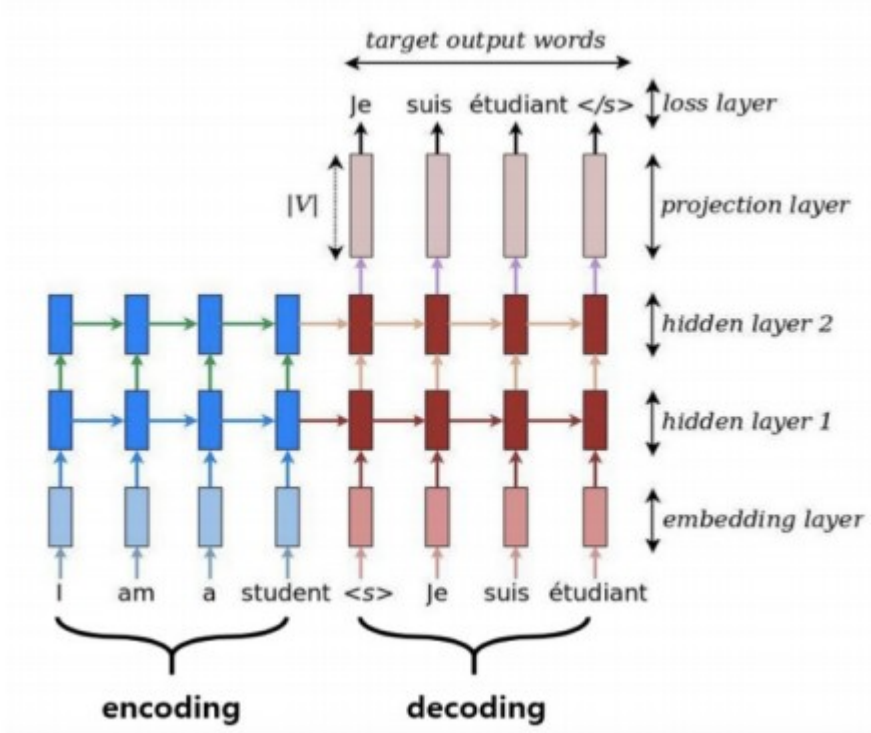
Now is the time to load and preprocess the data. The dataset that we are going to use is chat conversation between two humans. Where one of the humans play the role of a robot. The files for this section are available on moodle by the name “human_text.txt” and “robot_text.txt”. The dataset can be downloaded online from this kaggle link as well.

If you are using Google Colaboratory (Which I recommend that you use), you will need to upload your file to your google drive and then run this script to access your google drive:

```
from google.colab import drive
drive.mount('/content/drive')
```

Else, you can simply store the dataset on your local file system.

In case of sequence to sequence learning, we need three separate lists as shown in the following figure:



The first one is the input text list that contains the input to the encoder LSTM (I am a student) , the second one is the target text list which will be the output of the decoder LSTM (Je suis etudiante <s>) and third one is the target text input list which will act as input to the decoder LSTM (<s> Je suis etudiant). Let's create these three lists:

```
input_texts = []
target_texts = []
target_texts_inputs = []
```

Next, we will read both the "human_text.txt" and "robot_text.txt", line by line. Remember that each line in the "robot_text.txt" is a response to the corresponding line in the "human_text.txt".

We will read the files line by line, remove all the punctuation and extra spaces from the lines. We will then check if the number of words in the input line is greater than the MAX_SENTENCE_LENGTH, simply ignore the sentence since we do not want very big sentences in our document.

Finally, we will insert lines from "human_text.txt" in the input_texts lists, and the lines from "robot_text.txt" will be inserted in the target_texts lists. We prefix all the lines in target text with the "eos" (end of sentence tag). The lines from the "robot_text.txt", will also be inserted to the target_texts_inputs list, however they will be prefixed with "sos" tag since the input to the decoder LSTM will be same as the target output, however it will be offset by one using the "sos" tag.

Take a look at the following script to perform the data import and preprocessing tasks. You will need to change the path as per your directory structure.

```
with open("/content/drive/My
Drive/Seq-seq-Chatbot/human_text.txt", encoding="utf8") as
textfile1, open("/content/drive/My
Drive/Seq-seq-Chatbot/robot_text.txt", encoding="utf8") as
textfile2:
  for human_text, robot_text in zip(textfile1, textfile2):


    human_text = human_text.strip()
    robot_text = robot_text.strip()

    human_text = re.sub(r'\W', ' ', str(human_text))
    robot_text = re.sub(r'\W', ' ', str(robot_text))
```

```
    human_text = re.sub(r'\s+', ' ', human_text, flags=re.I)
    robot_text = re.sub(r'\s+', ' ', robot_text, flags=re.I)



    input_text = human_text
    target_text = robot_text + ' <eos>'
    target_text_input = '<sos> ' + robot_text

    if len(input_text.split()) > MAX_SENTENCE_LENGTH:
      break

    input_texts.append(input_text)
    target_texts.append(target_text)
    target_texts_inputs.append(target_text_input)



print("num samples:", len(input_texts))
```

Let's print the number of samples in our dataset. Execute the following script to do so:

```
print("num samples:", len(input_texts))
```

The output will display 579. Our sample size is really small but you will see that our dataset will be able to answer greetings.

Let's now randomly choose sentences from three lists to see how they look:

```
print(input_texts[2])
print(target_texts[2])
print(target_texts_inputs[2])
```

Here is the output:

```
how do you feel today tell me something about yourself
my name is rdany but you can call me dany the r means robot i
hope we can be virtual friends  <eos>
<sos> my name is rdany but you can call me dany the r means robot
i hope we can be virtual friends
```

Challenge: Try to print some other sentence in the three lists and see if you can differentiate between the encoder inputs, decoder inputs and decoder outputs.

Step 4: Tokenizing and Padding the Sentences

Before we can feed our sentences into LSTM, we need to convert them into word vectors. For Input, we will use glove word embeddings for our data but before that, we need to convert sentences into integers so that the integers can be used to index the embedding matrix, which contains the vector representation of the word. For the target text, we will use Keras built in embedding, which we will see in a later section. For now, let's tokenize our input and target data. Execute the following script to tokenize the input:

```
tokenizer_inputs = Tokenizer(num_words=VOCABULARY_SIZE )
tokenizer_inputs.fit_on_texts(input_texts)
input_sequences =
tokenizer_inputs.texts_to_sequences(input_texts)
word2idx_inputs = tokenizer_inputs.word_index
max_len_input = max(len(s) for s in input_sequences)
```

In the script above we, tokenize in the inputs, find the number of unique tokens in the input data and the length of the longest sentence.

Challenge: Try to tokenize the target_text and target_texts_inputs list yourself.

```
tokenizer_outputs =
Tokenizer(num_words=VOCABULARY_SIZE ,filters='')
tokenizer_outputs.fit_on_texts(target_texts +
target_texts_inputs) # inefficient, oh well
target_sequences =
tokenizer_outputs.texts_to_sequences(target_texts)
target_sequences_inputs =
tokenizer_outputs.texts_to_sequences(target_texts_inputs)
word2idx_outputs = tokenizer_outputs.word_index
num_words_output = len(word2idx_outputs) + 1
max_len_target = max(len(s) for s in target_sequences)
```

Let's now print the two sentences from the tokenized `input_sequence` list:

```
print(input_sequences[2])
```

```
print(input_sequences[3])
```

Here is the output:

```
[15, 4, 1, 142, 86, 54, 18, 58, 23, 107]
[15, 168, 311, 221, 9, 1, 222]
```

Both our inputs and output sentences are not of same size. We will pad the inputs according to the longest sentence in the input text. In padding, zeros are added to the indexes that are empty when the size of a sentence is increased to the size of the longest sentence. Similarly, the target text and target text input will be padded as per the longest sentence in the output data. Execute the following script to pad input and output data:

```
encoder_inputs = pad_sequences(input_sequences,
maxlen=max_len_input)
```

Challenge: Try to apply padding to `target_sequence` and `target_sequence_inputs` list to create `decoder_target` and `decoder_inputs` list respectively. Apply post padding, to do so you can use "padding = 'post'" attribute.

```
decoder_inputs = pad_sequences(target_sequences_inputs,
maxlen=max_len_target, padding='post')


decoder_targets = pad_sequences(target_sequences,
maxlen=max_len_target, padding='post')
```

Step 5: Loading Word Embeddings and Creating Word Embedding Matrix

We will use Glove embeddings to achieve vector representation of the word. You need to change the path of the Glove file as per your runtime setup. Execute the following script to load pertained word vectors:

```
print('Loading Word Vectors')
embedding_dic = {}
with open ('/content/drive/My Drive/datasets/glove.6B.100d.txt',
encoding= "utf8")as vector_list:
```

```
  for line in vector_list:
    dimensions = line.split()
    word = dimensions[0]
    vector = np.asarray(dimensions[1:], dtype='float32')
    embedding_dic[word] = vector
```

The following script creates word embeddings for the input text:

```
num_words = min(VOCABULARY_SIZE, len(word2idx_inputs) + 1)
embedding_matrix = np.zeros((num_words, VECTOR_DIM ))
for word, i in word2idx_inputs.items():
  if i < num_words:
    embedding_vector = embedding_dic.get(word)
    if embedding_vector is not None:
      embedding_matrix[i] = embedding_vector
```

Challenge: Try to create Word Embeddings for the decoder input and output:

```
num_words_out = min(VOCABULARY_SIZE, len(word2idx_outputs) + 1)
embedding_matrix_out = np.zeros((num_words_out, VECTOR_DIM ))
for word, i in word2idx_outputs.items():
  if i < num_words_out:
    embedding_vector_out = embedding_dic.get(word)
    if embedding_vector_out is not None:
      embedding_matrix_out[i] = embedding_vector_out
```

Step 6: Creating Input and Output Embedding Layers and Output One Hot Encoded Layer

Before we create our model, we need to create the input embedding layer and the one hot encoded output layer. Since we are using Softmax function at the output node, our output should be one hot encoded. The following script creates embedding layer and one hot encoded output vector:

```
embedding_layer = Embedding(
 num_words,
 VECTOR_DIM,
 weights=[embedding_matrix],
 input_length=max_len_input,
 # trainable=True
)

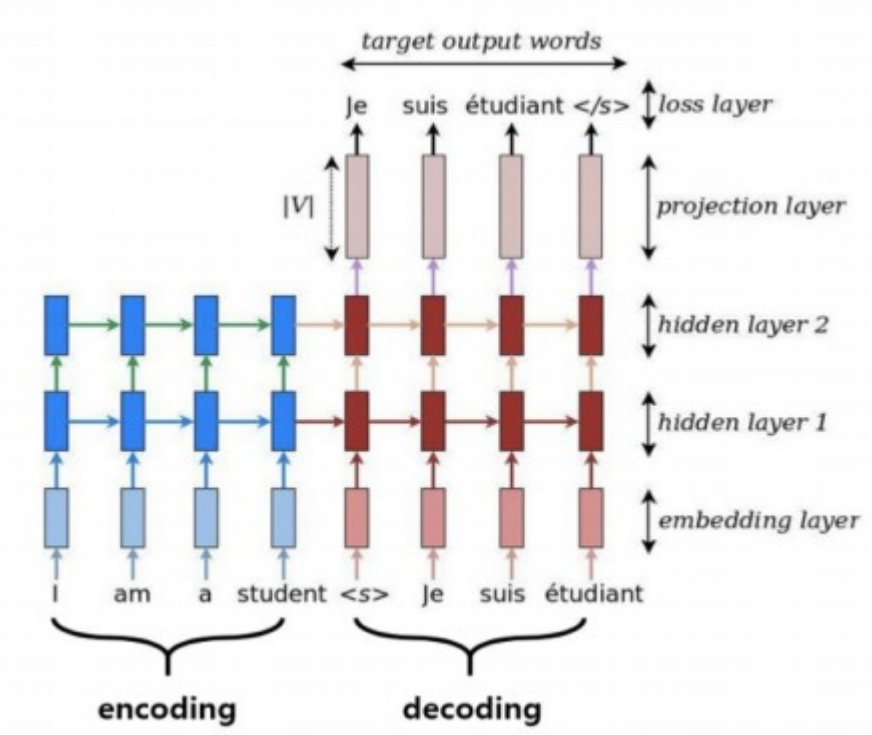
embedding_layer_out = Embedding(
 num_words_out,
 VECTOR_DIM,
 weights=[embedding_matrix_out],
 input_length=max_len_target,
 # trainable=True
)

decoder_targets_one_hot = np.zeros(
 (
 len(input_texts),
 max_len_target,
 num_words_output
 ),
 dtype='float32'
)
for i, d in enumerate(decoder_targets):
  for t, word in enumerate(d):
    decoder_targets_one_hot[i, t, word] = 1
    decoder_targets_one_hot[i, t, word] = 1
```

Step 7: Creating the Model

We have preprocessed all the data needed to create our model. Now is the time to create our model.

As we said earlier, we need to create two LSTMs: an encoder LSTM and a decoder LSTM. During training, our model will work as explained in the following figure:



The input will be passed to encoder, the encoder will generate the hidden and cell states which will be then passed to the decoder. The input to the decoder will be the hidden and cell state from the encoder, along with the target sentence input with <"sos"> at the beginning. The output will be predicted response.

Let's first define the input, embedding layer and the output for the encoder. Execute the following script:

```
encoder_inputs_placeholder = Input(shape=(max_len_input,))
x = embedding_layer(encoder_inputs_placeholder)
encoder = LSTM(
  LSTM_NODES,
  return_state=True,
)
encoder_outputs, h, c = encoder(x)
encoder_states = [h, c]
```

In the script above, we create input place holders for the encoder input. We know that encoder receives a sequence of input sentences. Therefore, the shape of the input is the "max_len_input" parameter.

Next, we pass our input through the embedding layer. Finally we create our encoder and set its return state to two. The input passed through the embedding layer is passed to the encoder which returns output, hidden state and cell state. From the encoder, we only need the cell state and hidden state, which we capture in the "encoder_states" list.

Similarly, let's create input and output for the decoder. Execute the following script:

```
decoder_inputs_placeholder = Input(shape=(max_len_target,))
#decoder_embedding = Embedding(num_words_output, LSTM_NODES)
decoder_inputs_x = embedding_layer_out
(decoder_inputs_placeholder)
decoder_lstm = LSTM(
 LSTM_NODES,
 return_sequences=True,
 return_state=True,
)
decoder_outputs, _, _ = decoder_lstm(
 decoder_inputs_x,
 initial_state=encoder_states
)
```

In the script above, we first define the input place holder for the decoder. Notice the length of the input shape for the input place holder is "max_len_target" which is the length of the longest sentence in target_text list. Next we pass the input placeholder through embedding layer. We then create an LSTM with return_sequences = True since we need output at every point in the decoder. Notice that the initial_state of the decoder_lstm is set to encoder_states which consists of the hidden state and cell state of the encoder at the output unit.

Next, we need to create a dense layer and pass the decoder output to the dense layer. Take a look at the following script:

```
decoder_dense = Dense(num_words_output, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)
```

Next, we need to create the model and compile it as shown below:

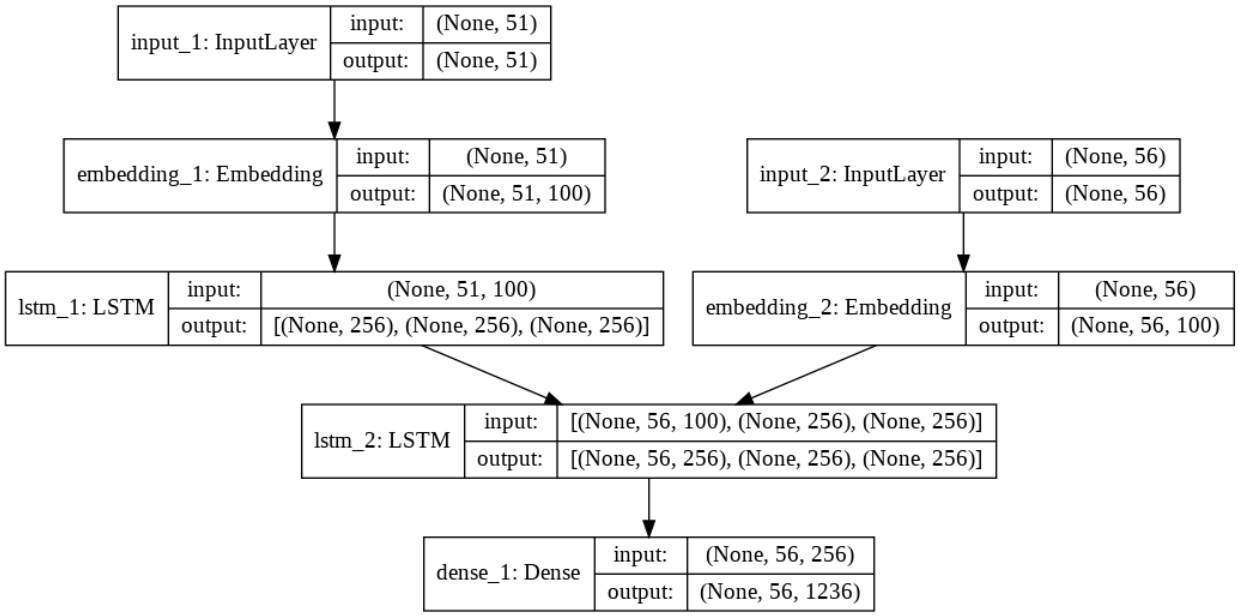
```
model = Model([encoder_inputs_placeholder,
decoder_inputs_placeholder], decoder_outputs)


model.compile(
  optimizer='rmsprop',
  loss='categorical_crossentropy',
  metrics=['accuracy']
  )
```

The following script prints the model: (this script works only in Google Colab, and requires Pydot library with graphic configurations to be installed if you want to run this on your local machines)

```
from keras.utils import plot_model
plot_model(model, to_file='model_plot_enc.png', show_shapes=True,
show_layer_names=True)
```

Here is the output:



Here the `lstm_1` is the encoder LSTM and the `lstm_2 ` is the decoder LSTM, the input to decoder LSTM is the hidden and cell state from the encoder LSTM and the embedded input sentence.

Finally, we need to fit our model on our data that we created in Step 1 to Step 6. Execute the following script to do so:

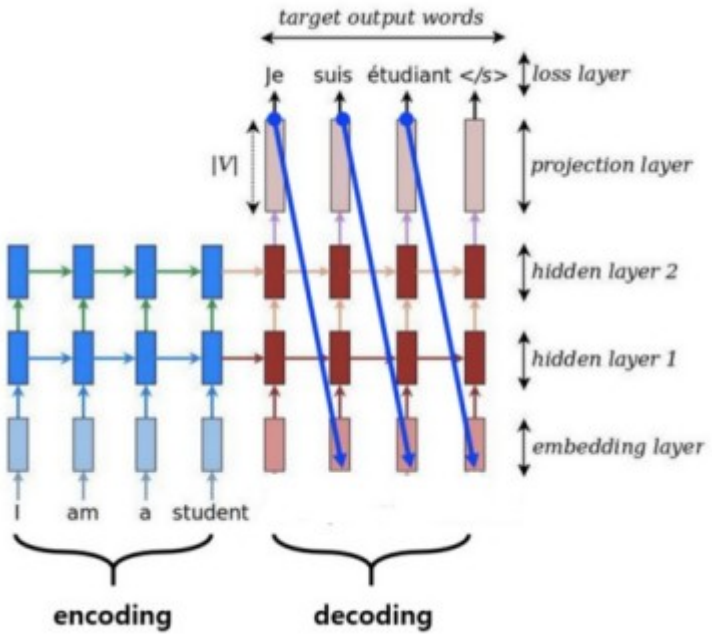
```
r = model.fit(
  [encoder_inputs, decoder_inputs], decoder_targets_one_hot,
  batch_size=BATCH_SIZE,
  epochs=EPOCHS,
  validation_split=0.2,
)
```

You will see that our model will start training. At the end of 7th epochs, you should see an accuracy of around 88.08% as shown in the following figure:

```
Train on 463 samples, validate on 116 samples
Epoch 1/7
463/463 [==============================] - 22s 49ms/step - loss: 1.6676 - acc: 0.8142 - val_loss: 0.9217 - val_acc: 0.8618
Epoch 2/7
463/463 [==============================] - 21s 46ms/step - loss: 1.0481 - acc: 0.8328 - val_loss: 0.8949 - val_acc: 0.8651
Epoch 3/7
463/463 [==============================] - 21s 45ms/step - loss: 0.9946 - acc: 0.8377 - val_loss: 0.8663 - val_acc: 0.8695
Epoch 4/7
463/463 [==============================] - 21s 45ms/step - loss: 0.9485 - acc: 0.8414 - val_loss: 0.8475 - val_acc: 0.8718
Epoch 5/7
463/463 [==============================] - 21s 46ms/step - loss: 0.9071 - acc: 0.8455 - val_loss: 0.8253 - val_acc: 0.8761
Epoch 6/7
463/463 [==============================] - 21s 46ms/step - loss: 0.8658 - acc: 0.8487 - val_loss: 0.8161 - val_acc: 0.8778
Epoch 7/7
463/463 [==============================] - 21s 46ms/step - loss: 0.8265 - acc: 0.8522 - val_loss: 0.8022 - val_acc: 0.8808
```

Step 8: Making Predictions

We have trained our model; the next step is to make predictions. In the prediction phase we will again have encoder-decoder architecture. However, we will train the encoders and decoders since we have trained them already. All we need to do is to change the input parameters. Here is what happens while making predictions:



The input will be passed to encoder, the encoder will generate the hidden and cell states which will be then passed to the decoder. The input to the decoder will be the hidden and cell state from the encoder, along with single word <"sos"> at the beginning. In the next iteration, the word predicted from the previous decoder LSTM will be used as input along with the hidden and cell state of the decoder. At the end, all the predicted words are joined together to create final response.

First let's discuss how our encoder will look like for making predictions. Take a look at the following script:

```
encoder_model = Model(encoder_inputs_placeholder, encoder_states)
```

Here the encoder will be stand alone. Remember, we will not train the encoding model, since we have already trained everything (Basically, in training the weights are updated). The weights have already been updated. The only job of the encoder is to take sequence as input and output the final hidden state and cell state of the LSTM.

Next, we need to define the inputs and outputs for our decoder. In the training phase, the inputs to the decoder were the hidden and cell state from the last LSTM unit of the input sequence and the whole target sequence offset by 1. The output was the corresponding target sequence.

In case of predictions, we do not have the whole target sequence since this is what we have to predict. The only thing we have is the hidden and cell state from the input sequence and the start of the word for the target sequence. The output of the decoder will be the corresponding

target word and the cell state and the hidden state. This cell state and the hidden state, along with the predicted word will be used as input to the next LSTM unit for the decoder.

Let's create inputs for the decoder LSTM:

```
decoder_state_input_h = Input(shape=(LSTM_NODES,))
decoder_state_input_c = Input(shape=(LSTM_NODES,))
decoder_states_inputs = [decoder_state_input_h,
decoder_state_input_c]
```

These are the hidden states and the cell state that each decoder LSTM will receive as input.

Next, we need to specify the input shape. Since this time, there will only be a one word as input, the shape of input will be 1. We need to create a new embedding layer which accepts only one word as shown below:

```
embedding_layer_out_single = Embedding(
 num_words_out,
 VECTOR_DIM,
 weights=[embedding_matrix_out],
 input_length=1,
 # trainable=True
)
```

The input is needed to be passed through, embedding layer. The following script does that:

```
decoder_inputs_single = Input(shape=(1,))
decoder_inputs_single_x =
embedding_layer_out_single(decoder_inputs_single)
```

Finally, we need to pass the decoder_states_inputs, and decoder_inputs_single_x as the single word to the decoder LSTM, that we trained during the training phase.

```
decoder_outputs, h, c = decoder_lstm(
  decoder_inputs_single_x,
  initial_state=decoder_states_inputs
)
```

The output of the LSTM will be the decoder_output and the hidden and cell state. The decoder output will be passed through a dense layer to produce the final output as shown in the final script:

```
decoder_states = [h, c]
decoder_outputs = decoder_dense(decoder_outputs)
```

Notice we store the hidden and cell state in a variable decoder_states since we want our model to output these values. The sampling model used for prediction can be created using the following script:

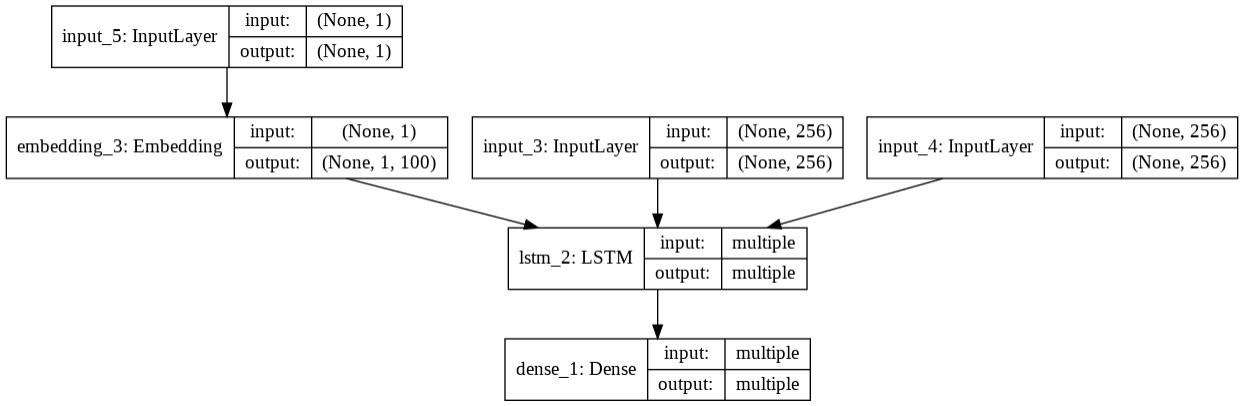
```
# The sampling model
# inputs: y(t-1), h(t-1), c(t-1)
# outputs: y(t), h(t), c(t)

decoder_model = Model(
  [decoder_inputs_single] + decoder_states_inputs,
  [decoder_outputs] + decoder_states
)
```

We can plot the decoder LSTM for predictions as follows:

```
from keras.utils import plot_model
plot_model(decoder_model, to_file='model_plot_dec.png',
show_shapes=True, show_layer_names=True)
```

Output:



The decoder here is `lstm_2`. The input to the decoder at time T will be the output at time T-1, along with the hidden and cell state at T-1.

In the preprocessing steps, we converted words to integers. The output of our model will be an integer. We need to convert the integer back to the word. Execute the following script to do so:

```
idx2word_input = {v:k for k, v in word2idx_inputs.items()}
idx2word_target = {v:k for k, v in word2idx_outputs.items()}
```

Here, idx2word_input and idw2word target are the dictionaries where key is the integer and the value is the word that corresponds to the integer.

Finally, we write a function that takes in the input sequence, and returns back the output sequence. The script for the function looks likes that:

```
def generate_response(input_seq):
  # Encode the input as state vectors.
  states_value = encoder_model.predict(input_seq)

  # Generate empty target sequence of length 1.
  target_seq = np.zeros((1, 1))

  # Populate the first character of target sequence with the
start character.
  # NOTE: tokenizer lower-cases all words
  target_seq[0, 0] = word2idx_outputs['<sos>']

  # if we get this we break
  eos = word2idx_outputs['<eos>']

  # Create the translation
  output_sentence = []
  for _ in range(max_len_target):
    output_tokens, h, c = decoder_model.predict(
      [target_seq] + states_value
    )

    # Get next word
```

```
    idx = np.argmax(output_tokens[0, 0, :])

    # End sentence of EOS
    if eos == idx:
      break

    word = ''
    if idx > 0:
      word = idx2word_target[idx]
      output_sentence.append(word)

    # Update the decoder input
    # which is just the word just generated
    target_seq[0, 0] = idx

    # Update states
    states_value = [h, c]
    # states_value = [h] # gru

  return ' '.join(output_sentence)
```

The “generate_response” function takes input sequence as the parameter. It first finds the hidden and cell state using the encoder model.

It then creates a new random variable of size 1 x 1 and initialize it with the index of “sos”. Remember, the first word of every sentence is “sos”. We then find the index of of “eos” since we want to end our target sequence generation when “eos” is predicted.

Next we execute the loop with the number of iterations equal to the maximum target sequence length. During each iteration we insert our input target plus the previous hidden state values to the decoder_model. In the output we receive probability distribution of all the possible words along with the hidden state and cell state.

We find the world with highest probability using the argmax function and store its index. If the index is equal to the index of eos, we exit our loop. Else if the index is greater than zero, we find

the corresponding word from the index and append it to the output_sentence list. Finally we join the individual words in the list and return it.

Step 8: Making Predictions

The final step is to see the magic. Take a look at the following script:

```
while True:
  # Do some test translations
  i = np.random.choice(len(input_texts))
  input_seq = encoder_inputs[i:i+1]
  translation = generate_response(input_seq)
  print('-')
  print('Input:', input_texts[i])
  print('Response:', translation)

  ans = input("Continue? [Y/n]")
  if ans and ans.lower().startswith('n'):
    break
```

In the script above we create an infinite loop. Inside the loop we randomly chose an input sequence from the list of input sequences and pass it to the generate_response function which returns the response message. The loop continues until you don't press "n". A snapshot of the output is here attached:

```
Continue? [Y/n]y
-
Input: nice do you use something like that
Response: mess mean ll you

Continue? [Y/n]y
-
Input: what do you think about golang can it replace python
Response: bit

Continue? [Y/n]y
-
Input: never paid attention to it can you recomend some films with that
Response: bit mean you

Continue? [Y/n]
```

The chatbots is not really efficient since I used only 581 samples which is very less compared to the dataset use to develop actual chatbots. However, the concept remains the same; you just need a bigger and better dataset to increase the efficiency of the chatbots.