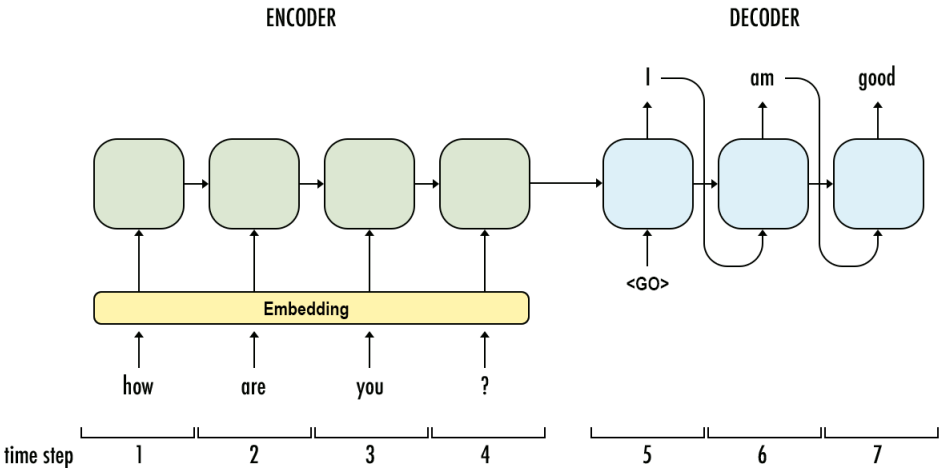
Encoder-Decoder Models in NMT

Joseph Rejive

March 2019

1 Introduction

Encoder-Decoder models are popular models for sequence to sequence tasks. They have many applications such as image captioning, neural machine translation, and creating chatbots. At their core, they consist of an encoder RNN which generates a 'context' vector from the input sequence and a decoder RNN which generates an output sequence based off the context.



2 Encoder

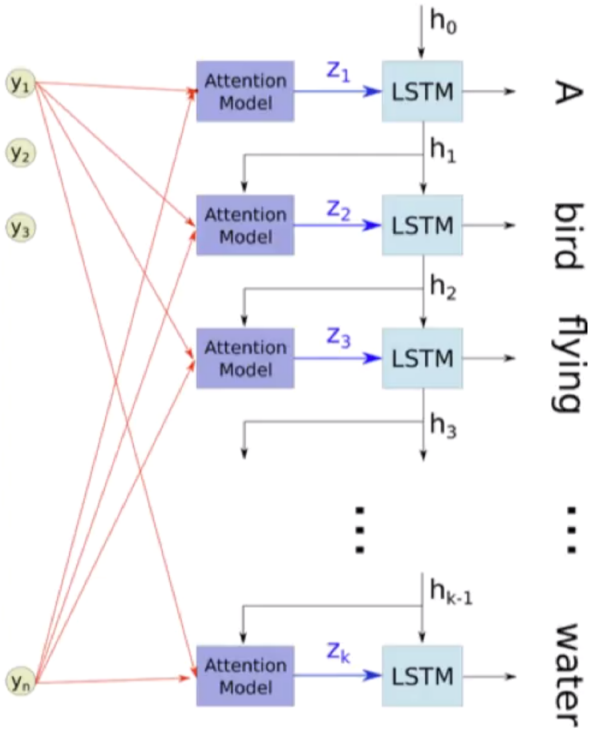
The encoder is just an RNN, typically an LSTM or GRU, which generates a 'context' vector. In the encoder RNN, we do not care about the outputs. Rather, we want to preserve the internal state of the RNN. The final internal state is what we refer to as the 'context' vector. This is what the decoder uses to return an output sequence.

3 Decoder

The decoder is another RNN which uses the 'context' vector (the final hidden state of the encoder RNN) to generate an output. In training, a start of sequence token signals the decoder to begin the translation. It then uses the output and the hidden states of the current timestep to produce the output and hidden states of the next timestep.

4 Problems With Long Term Dependencies

Encoder-Decoder models struggle to handle long input sequences. Although in theory, LSTMs were designed to handle long term dependencies, they does not work perfectly in practice. For example, say the encoder input is an English sentence that has 50 words. The encoder has to represent all this information into a fixed size vector. The decoder then has to use this vector and remember the information from 50 timesteps ago. To fix this issue, a strategy called 'Attention' is used.



In the picture above, 'y' is the encoder output at each timestep, 'LSTM' is the decoder RNN, 'h' is the decoder hidden state, and 'Z' is the output of the attention mechanism. The attention model takes each encoder output and the decoder states of the previous timestep to compute the decoder states of the current timestep. By looking at every encoder output, the decoder is able to take into account the context of the entire sentence and produce a more accurate translation.

5 Implementing an Encoder-Decoder Model in Keras

Here's the overall process for training the model:
1) Create 3 numpy arrays: 'encoder_input_data', 'decoder_input_data', and 'decoder_target_data'
'encoder_input_data' is an array of shape (num_samples, max_english_sentence_length, num_english_chars) which contains the one hot encoded English sentences.
'decoder_input_data' is an array of shape (num_samples, max_spanish_sentence_length, num_spanish_chars) which contains the one hot encoded Spanish sentences.
'decoder_target_data' is the same as 'decoder_input_data' but offset by one timestep. This means decoder_target_data[t] equals decoder_input_data[t+1]
2) Create an encoder LSTM which takes in 'encoder_input_data' and returns its final states.
3) Create a decoder LSTM which accepts the encoder's final states and 'decoder_input_data' and returns a character prediction.

5.1 Training the Model

```
import numpy as np
from keras.layers import LSTM, Dense, Input
from keras.models import Model

#hyperparameters
epochs = 100
latent_dim = 512
batch_size = 250
num_samples = 1000

#get data and vectorize it
data = open('spa.txt', 'r').read().splitlines()[:num_samples]
english = [i.split('\t')[0].lower() for i in data]
spanish = ['\t' + i.split('\t')[1].lower() + '\n' for i in data]
english_vocab = sorted({letter for word in english for letter in word})
spanish_vocab = sorted({letter for word in spanish for letter in word})

num_english_features = len(english_vocab)
num_spanish_features = len(spanish_vocab)
max_encoder_size = max([len(i) for i in english])
max_decoder_size = max([len(i) for i in spanish])

eng_c2i = {c:i for i,c in enumerate(english_vocab)}
spa_c2i = {c:i for i,c in enumerate(spanish_vocab)}
eng_i2c = {i:c for c,i in eng_c2i.items()}
spa_i2c = {i:c for c,i in spa_c2i.items()}
```

In this example, `\t` is the start of sequence token while `\n` is the end of sequence token. The 'c2i' dictionaries map characters to integers while the 'i2c' dictionaries map integers to characters. The c2i dictionaries will be used to vectorize a sentence while the i2c dictionaries will be used to obtain a character from a vector.

```
encoder_input_data = np.zeros(shape = (num_samples, max_encoder_size, num_english_features))
decoder_input_data = np.zeros(shape = (num_samples, max_decoder_size, num_spanish_features))
decoder_target_data = np.zeros(shape = (num_samples, max_decoder_size, num_spanish_features))

for index in range(num_samples):
    for i, c in enumerate(english[index]):
        encoder_input_data[index, i, eng_c2i[c]] = 1
    for i, c in enumerate(spanish[index]):
        decoder_input_data[index, i, spa_c2i[c]] = 1
        if i > 0:
            decoder_target_data[index, i-1, spa_c2i[c]] = 1
```

'encoder_input_data' is the one hot encoded vector which will be fed into the encoder LSTM. 'decoder_input_data' is the one hot vector which is fed into the decoder LSTM. The only difference between 'decoder_input_data' and 'decoder_target_data' is that 'decoder_target_data' is one timestep ahead of 'decoder_input_data'. This means 'decoder_target_data' will be compared to the output of the decoder LSTM at each timestep to compute the loss.

```
encoder_input = Input(shape = (None, num_english_features))
encoder_lstm = LSTM(latent_dim, return_state = True)
_, encoder_state_h, encoder_state_c = encoder_lstm(encoder_input)
encoder_states = [encoder_state_h, encoder_state_c]

decoder_input = Input(shape = (None, num_spanish_features))
```

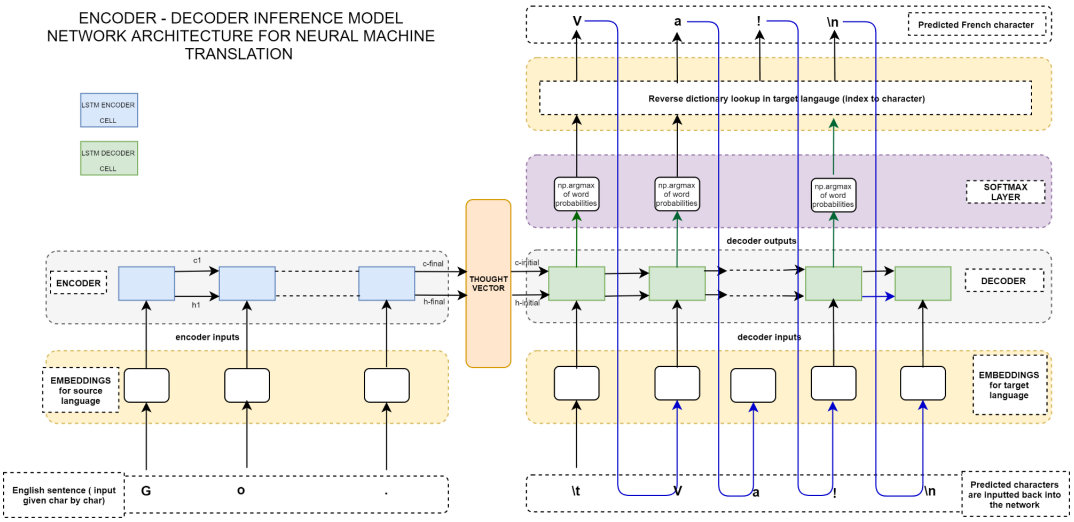
```
decoder_lstm = LSTM(latent_dim, return_sequences = True, return_state = True)
decoder_outputs, _, _ = decoder_lstm(decoder_input, initial_state = encoder_states)
decoder_dense = Dense(num_spanish_features, activation = 'softmax')
dense_output = decoder_dense(decoder_outputs)

model = Model([encoder_input, decoder_input], dense_output)
model.compile(optimizer = 'rmsprop', loss = 'categorical_crossentropy')
model.fit([encoder_input_data, decoder_input_data], decoder_target_data, epochs = epochs,
    batch_size = batch_size, validation_split = .1)
```

On the encoder side, we feed 'encoder_input_data' into the 'encoder_input' variable which is then fed into the LSTM. 'return_states = True' will return the final hidden states of the encoder LSTM, which are then used as the initial states of the decoder LSTM. Finally, a Dense layer with a softmax activation is used to create our predictions, which are then compared with 'decoder_target_data' to calculate our loss.

5.2 Testing the Model (Inference)

To actually test the model we trained, we need a slightly different setup. We'll create an encoder model which will accept a vectorized sentence as the input and output the LSTM's final hidden states. The decoder model will accept 2 different inputs: the hidden states of the decoder at the previous timestep (at the first timestep it will accept the encoder's hidden states) and the output of the decoder LSTM at the previous timestep (at the first timestep it will accept the start of sentence token, \t).




```
encoder_model = Model(encoder_input, encoder_states)

decoder_input_h = Input(shape = (latent_dim,))
decoder_input_c = Input(shape = (latent_dim,))
decoder_state_inputs = [decoder_input_h, decoder_input_c]
decoder_outputs, state_h, state_c = decoder_lstm(decoder_input, initial_state =
    decoder_state_inputs)
decoder_state_outputs = [state_h, state_c]
decoder_outputs = decoder_dense(decoder_outputs)
decoder_model = Model(decoder_state_inputs + [decoder_input], [decoder_outputs] +
    decoder_state_outputs)
```

Here, we create an encoder model which returns its internal states. The decoder model will then use the decoder's internal states from the previous timestep and the decoder's output from the previous timestep to

compute the output and hidden states of the current timestep.

```
def decode_sequence(input_seq):
    states_value = encoder_model.predict(input_seq)

    target_seq = np.zeros((1, 1, num_spanish_features))
    target_seq[0, 0, spa_c2i['\t']] = 1.

    stop_condition = False
    decoded_sentence = ''
    while not stop_condition:
        output_tokens, h, c = decoder_model.predict(states_value+
            [target_seq])
        sampled_token_index = np.argmax(output_tokens[0, 0, :])
        sampled_char = spa_i2c[sampled_token_index]
        decoded_sentence += sampled_char

        if (sampled_char == '\n' or
           len(decoded_sentence) > max_decoder_size):
            stop_condition = True

        target_seq = np.zeros((1, 1, num_spanish_features))
        target_seq[0, 0, sampled_token_index] = 1.

        states_value = [h, c]

    return decoded_sentence
```

First, we get the encoder's final hidden states by feeding in the vectorized English sentence. Then we feed the decoder the start of sequence token and the encoder's final hidden states. The decoder then outputs the vectorized form of the output character along with the decoder's hidden states. We take 'output_tokens' and convert that one_hot encoded vector into its respective character by using the i2c dictionary. This character, along with the decoder's hidden states of the current timestep, is then fed back into the decoder. This process keeps on going until we hit the end of sequence token (\n).

```
for seq_index in range(10):
    input_seq = encoder_input_data[seq_index: seq_index + 1]
    decoded_sentence = decode_sequence(input_seq)
    print('Input sentence:', english[seq_index])
    print('Decoded sentence:', decoded_sentence)
```

After testing the model on sentences from the training data, here is the output:

```
Input sentence: i work.
Decoded sentence: estoy trabajando.

Input sentence: im 19.
Decoded sentence: tengo diecinueve.

Input sentence: im up.
Decoded sentence: estoy levantado.
```

The model works well on these examples, but that's to be expected as it's from the training data. Further improvements could include a word level model with an attention mechanism.