Word Level English to Marathi Neural Machine Translation using Encoder-Decoder Model

A Guide to build Sequence to sequence models using LSTM

Harshall Lamba Feb 9, 2019 · 13 min read

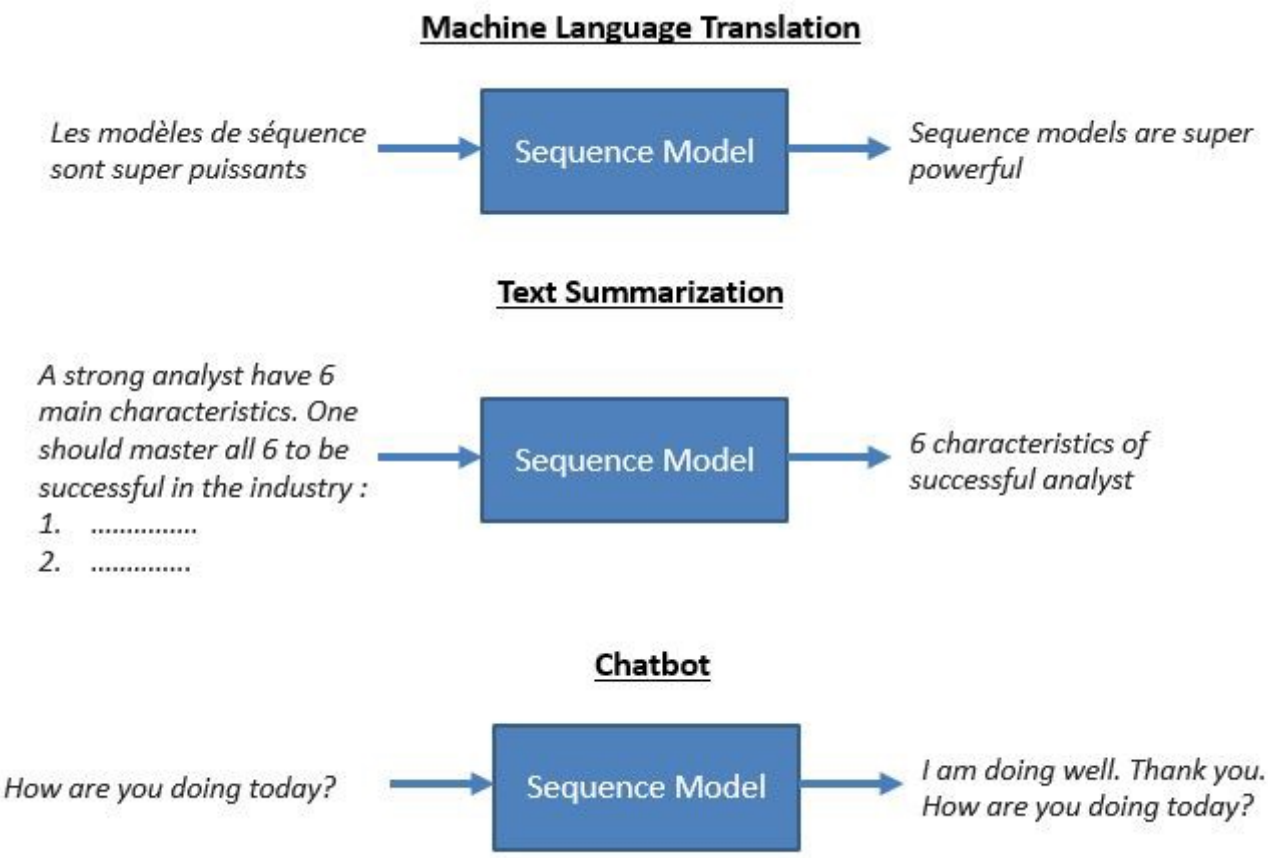
Table of Contents

1. Introduction

Recurrent Neural Networks (or more precisely LSTM/GRU) have been found to be very effective in solving complex sequence related problems given a large amount of data. They have real time applications in speech recognition, Natural Language Processing (NLP) problems, time series forecasting, etc. This *blog* nicely explains some of these applications.

Sequence to Sequence (often abbreviated to seq2seq) models are a special class of Recurrent Neural Network architectures typically used (but not restricted) to solve complex Language related problems like Machine

Translation, Question Answering, creating Chat-bots, Text Summarization, etc.



Source https://www.analyticsvidhya.com/blog/2018/04/sequence-modelling-an-introduction-with-practical-use-cases/

The purpose of this blog post is to give a detailed explanation on how sequence to sequence models are built and to give an intuitive understanding of how they solve these complex tasks.

We will take the problem of Machine Translation (translating a text from one language to another, in our case from English to Marathi) as the running example in this blog. However the technical details apply to any sequence to sequence problem in general.

Since we are using Neural Networks to perform Machine Translation, more commonly it is called as Neural Machine translation (NMT).

2. Prerequisites

This post assumes that you:

a. Know Fundamental concepts in Machine Learning and Neural Networks

b. Know High School Linear Algebra and Probability

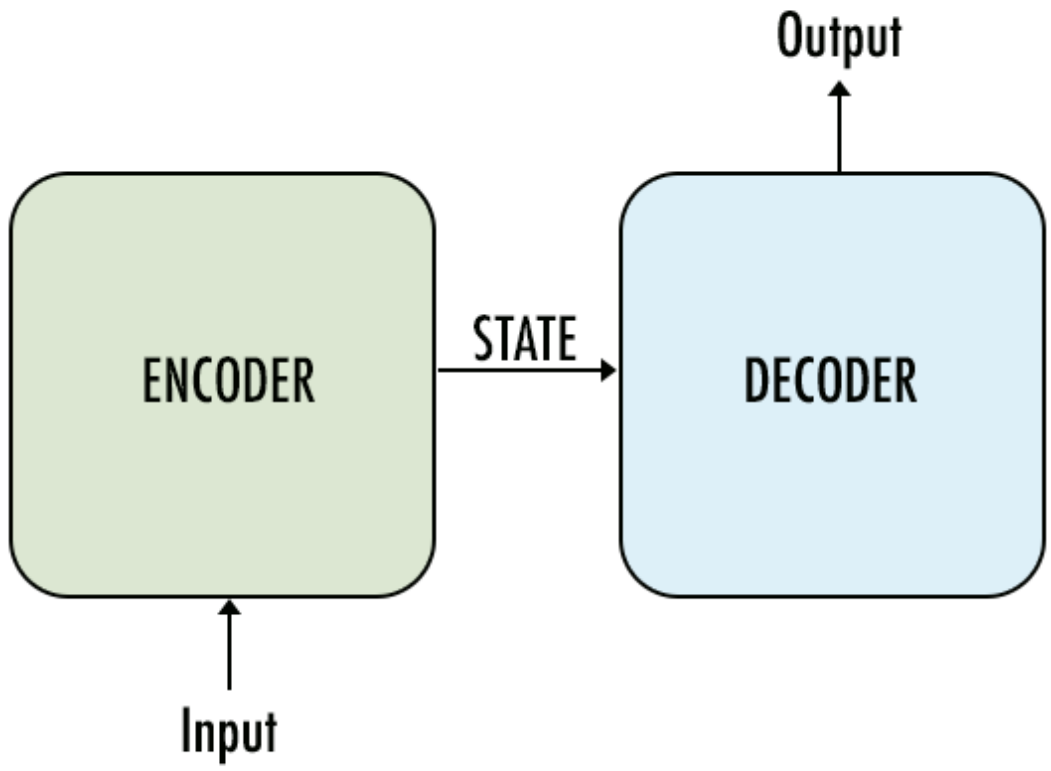
c. Have working knowledge of LSTM networks in Python and Keras

The Unreasonable Effectiveness of Recurrent Neural Networks (explains how RNNs can be used to build language models) and *Understanding LSTM Networks* (explains the working of LSTMs with solid intuition) are two

brilliant blogs that I strongly suggest to go through if you haven't. The concepts explained in these blogs are extensively used in my post.

3. Encoder — Decoder Architecture

The most common architecture used to build Seq2Seq models is the Encoder Decoder architecture. This is the one we will use for this post. Below is a very high level view of this architecture.



Source: https://towardsdatascience.com/sequence-to-sequence-model-introduction-and-concepts-44d9b41cd42d

Points to note:

a. Both encoder and the decoder are typically LSTM models (or sometimes GRU models)

b. Encoder reads the input sequence and summarizes the information in something called as the internal state vectors (in case of LSTM these are called as the hidden state and cell state vectors). We discard the outputs of the encoder and only preserve the internal states.

c. Decoder is an LSTM whose initial states are initialized to the final states of the Encoder LSTM. Using these initial states, decoder starts generating the output sequence.

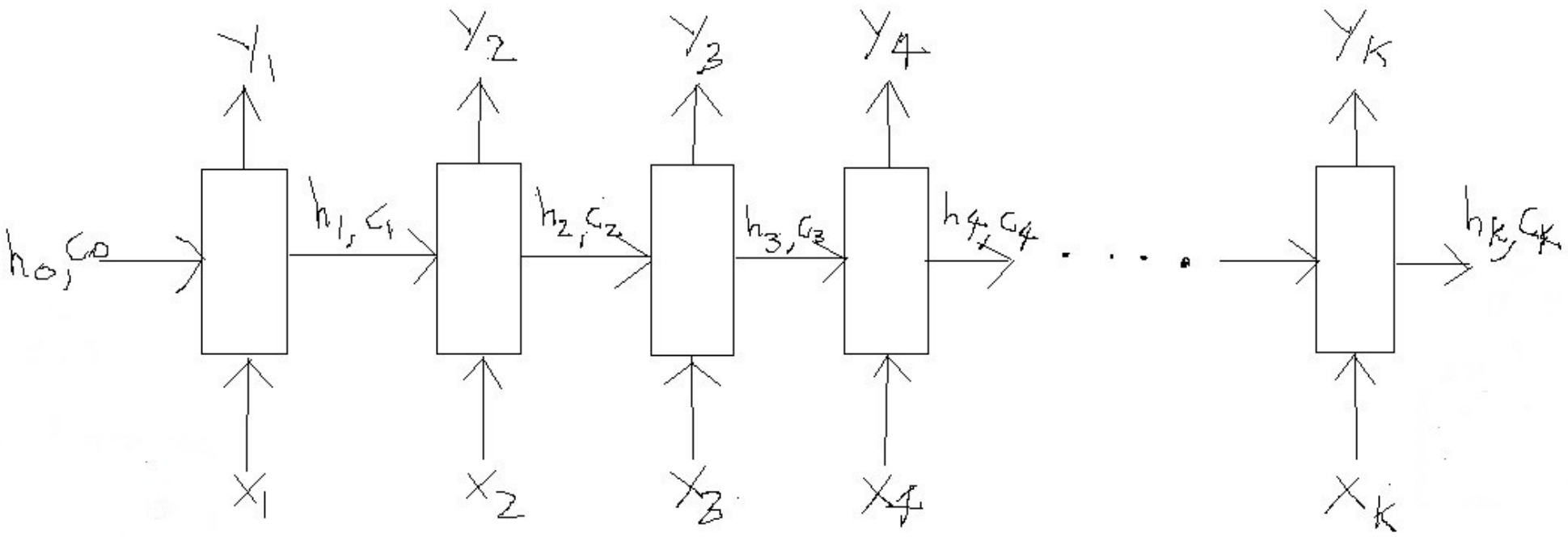
d. The decoder behaves a bit differently during the training and inference procedure. During the training, we use a technique call teacher forcing which helps to train the decoder faster. During inference, the input to the decoder at each time step is the output from the previous time step.

e. Intuitively, the encoder summarizes the input sequence into state vectors (sometimes also called as Thought vectors), which are then fed to the decoder which starts generating the output sequence given the Thought vectors. The decoder is just a language model conditioned on the initial states.

Now we will understand all the above steps in detail by considering the example of translating an English sentence (input sequence) into its equivalent Marathi sentence (output sequence).

4. Encoder LSTM

This section provides a brief overview of the main components of the Encoder LSTM. I will keep this intuitive without going into the Mathematical arena. So here's what they are:



LSTM processing an input sequence of length 'k'

The LSTM reads the data one sequence after the other. Thus if the input is a sequence of length 'k', we say that LSTM reads it in 'k' time steps (think of this as a for loop with 'k' iterations).

Referring to the above diagram, below are the 3 main components of an LSTM:

a. Xi => Input sequence at time step i

b. hi and ci => LSTM maintains two states ('h' for hidden state and 'c' for cell state) at each time step. Combined together these are internal state of the LSTM at time step i.

c. Yi => Output sequence at time step i

Important: Technically all of these components (Xi, hi, ci and Yi) are actually vectors of floating point numbers (explained below)

Let's try to map all of these in the context of our problem. Recall that our problem is to translate an English sentence to its Marathi equivalent. For the purpose of this blog we will consider the below example. Say, we have the following sentence

Input sentence (English) => "Rahul is a good boy"

Output sentence (Marathi) => "राहुल चांगला मुलगा आहे"

For now just focus on the input i.e. the English sentence

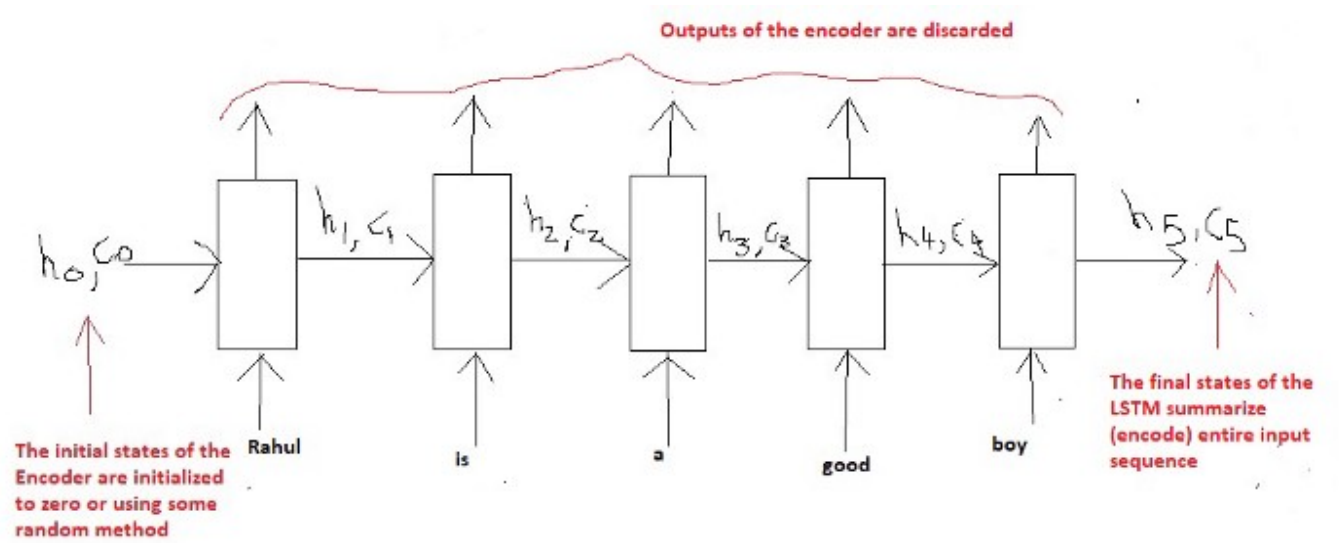
Explanation for Xi:

Now a sentence can be seen as a sequence of either words or characters. For example in case of words, the above English sentence can be thought of as a sequence of 5 words ('Rahul', 'is', 'a', 'good', 'boy'). And in case of characters, it can be thought of as a sequence of 19 characters ('R', 'a', 'h', 'u', 'l', ' ', ……, 'y').

We will break the sentence by words as this scheme is more common in real world applications. Hence the name 'Word Level NMT'. So, referring to the diagram above, we have the following input:

X1 = 'Rahul', X2 = 'is', X3 = 'a', X4 = 'good, X5 = 'boy'.

The LSTM will read this sentence word by word in 5 time steps as follows



Encoder LSTM

But one question that we must answer is how to represent each Xi (each word) as a vector?

There are various word embedding techniques which map (embed) a word into a fixed length vector. I will assume the reader to be familiar with the concept of word embeddings and won't cover this topic in detail. However,

we will use the built-in Embedding Layer of the Keras API to map each word into a fixed length vector.

Explanation for hi and ci:

The next question is what is the role of the internal states (hi and ci) at each time step?

In very simple terms, they remember what the LSTM has read (learned) till now. For example:

h3, c3 =>These two vectors will remember that the network has read "Rahul is a" till now. Basically its the summary of information till time step 3 which is stored in the vectors h3 and c3 (thus called the states at time step 3).

Similarly, we can thus say that h5, c5 will contain the summary of the entire input sentence, since this is where the sentence ends (at time step 5). These states coming out of the last time step are also called as the "**Thought vectors**" as they summarize the entire sequence in a vector form.

Then what about h0,c0? These vectors are typically initialized to zero as the model has not yet started to read the input.

Note: The size of both of these vectors is equal to number of units (neurons) used in the LSTM cell.

Explanation for Yi:

Finally, what about Yi at each time step? These are the output (predictions) of the LSTM model at each time step.

But what type of a vector is Yi? More specifically in case of word level language models each Yi is actually a probability distribution over the entire vocabulary which is generated by using a softmax activation. Thus each Yi is a vector of size "vocab_size" representing a probability distribution.

Depending on the context of the problem they might sometimes be used or sometimes be discarded.

In our case we have nothing to output unless we have read the entire English sentence. Because we will start generating the output sequence (equivalent Marathi sentence) once we have read the entire English sentence. Thus we will discard the Yi of the Encoder for our problem.

Summary of the encoder:

We will read the input sequence (English sentence) word by word and preserve the internal states of the LSTM network generated after the last time step hk, ck (assuming the sentence has 'k' words). These vectors (states hk and ck) are called as the **encoding** of the input sequence, as they encode (summarize) the entire input in a vector form. Since we will start generating the output once we have read the entire sequence, outputs (Yi) of the Encoder at each time step are discarded.

Moreover you must also understand what type of vectors are Xi, hi, ci and Yi. What are their sizes (shapes) and what do they represent. If you have any confusion understanding this part, then you need to first strengthen your understanding of LSTM and language models.

5. Decoder LSTM — Training Mode

Unlike the Encoder LSTM which has the same role to play in both the training phase as well as in the inference phase, the Decoder LSTM has a slightly different role to play in both of these phases.

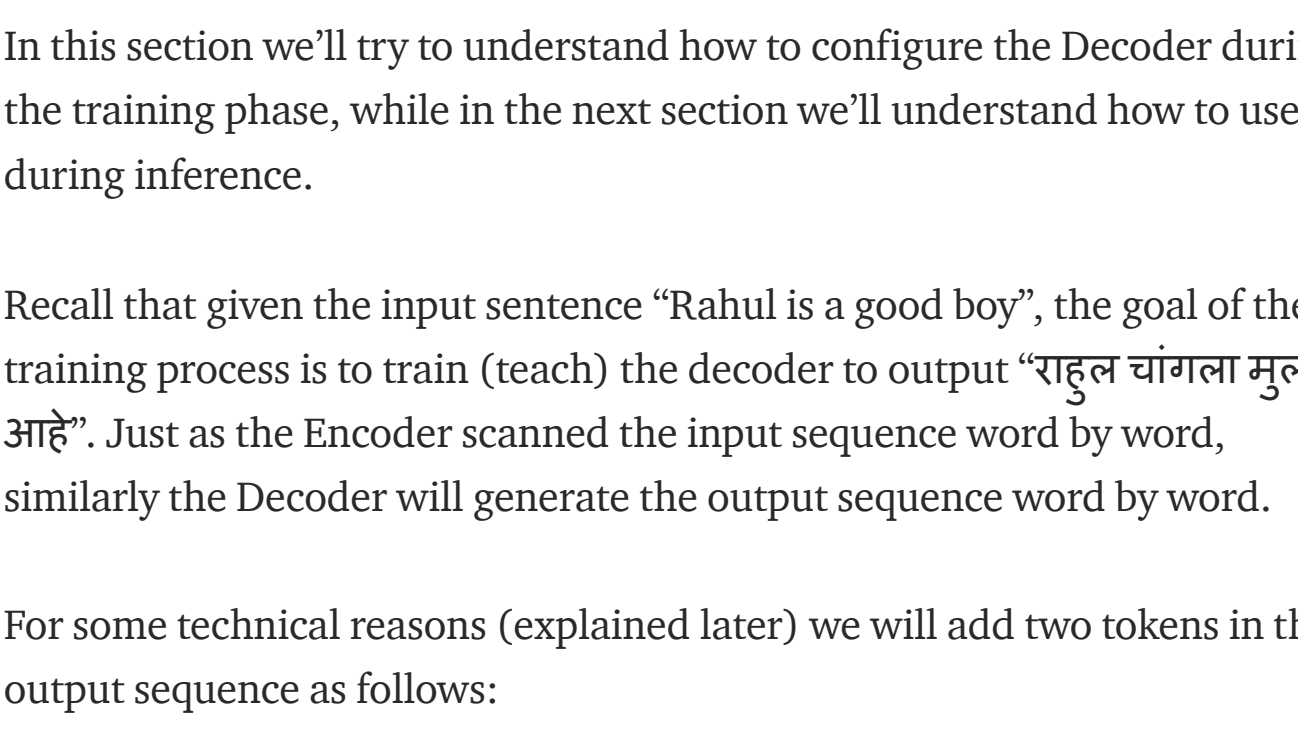
In this section we'll try to understand how to configure the Decoder during the training phase, while in the next section we'll understand how to use it during inference.

Recall that given the input sentence "Rahul is a good boy", the goal of the training process is to train (teach) the decoder to output "राहुल चांगला मुलगा आहे". Just as the Encoder scanned the input sequence word by word, similarly the Decoder will generate the output sequence word by word.

For some technical reasons (explained later) we will add two tokens in the output sequence as follows:

Output sequence => "START_ राहुल चांगला मुलगा आहे _END"

Now consider the diagram below:



Decoder LSTM — Training Mode

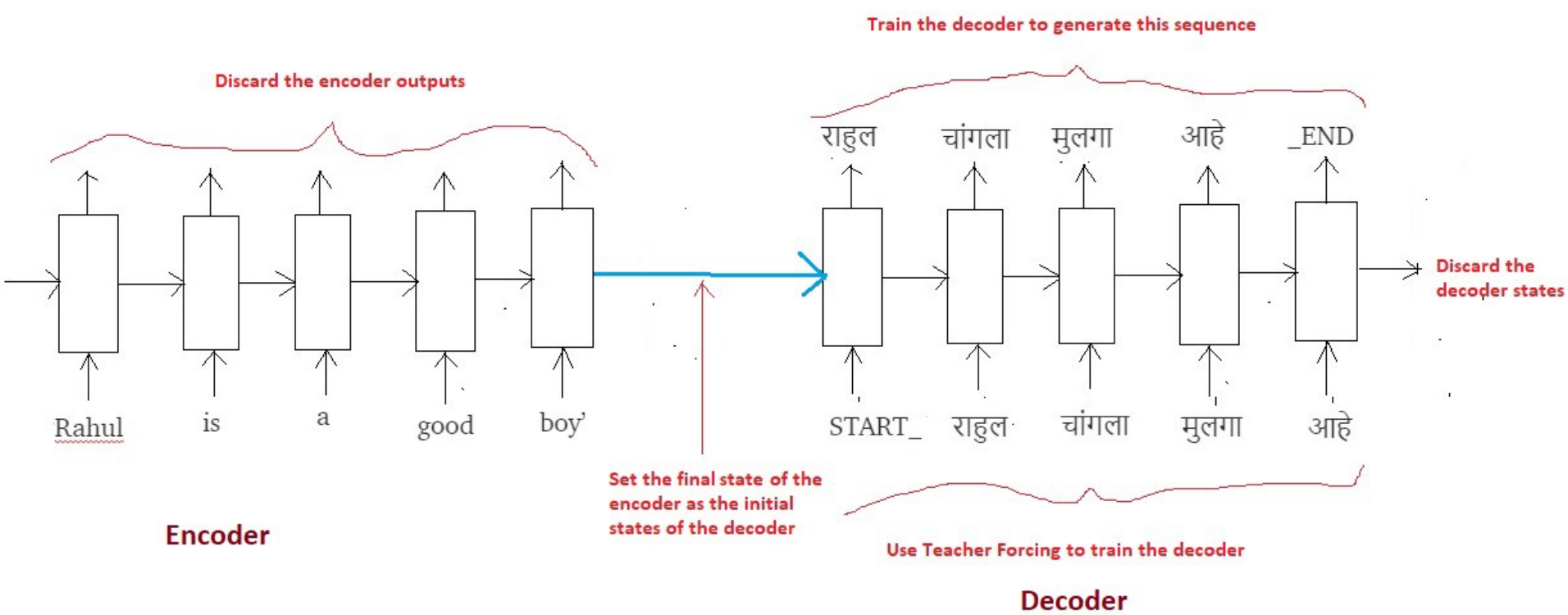
The most important point is that the initial states (h0, c0) of the decoder are set to the final states of the encoder. This intuitively means that the decoder is trained to start generating the output sequence depending on the information encoded by the encoder. Obviously the translated Marathi sentence must depend on the given English sentence.

In the first time step we provide the START_ token so that the decoder starts generating the next token (the actual first word of Marathi sentence). And after the last word in the Marathi sentence, we make the decoder learn to predict the _END token. This will be used as the stopping condition during the inference procedure, basically it will denote the end of the translated sentence and we will stop the inference loop (more on this later).

We use a technique called “Teacher Forcing” wherein the input at each time step is given as the **actual** output (and not the predicted output) from the previous time step. This helps in more faster and efficient training of the network. To understand more about teacher forcing, refer this *blog*.

Finally the loss is calculated on the predicted outputs from each time step and the errors are back propagated through time in order to update the parameters of the network. Training the network over longer period with sufficiently large amount of data results in pretty good predictions (translations) as we’ll see later.

The entire training process (Encoder + Decoder) can be summarized in the below diagram:



Summary of the training process

6. Decoder LSTM — Inference Mode

Let's now try to understand the setup required for inference. As already stated the Encoder LSTM plays the same role of reading the input sequence (English sentence) and generating the thought vectors (hk, ck).

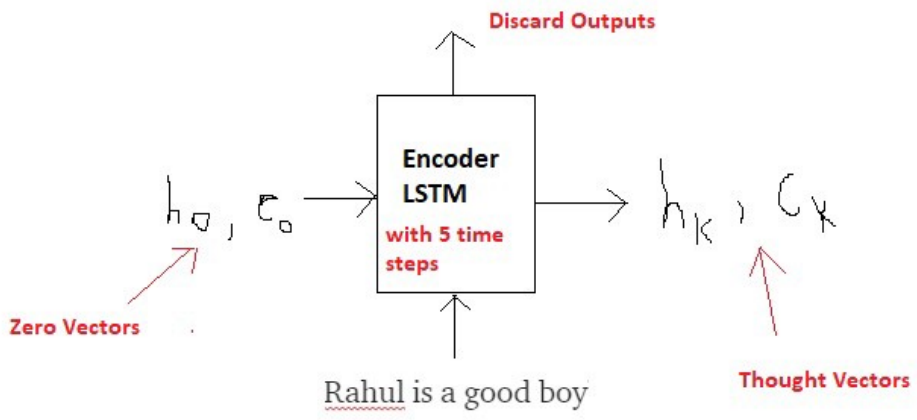
However, the decoder now has to predict the entire output sequence (Marathi sentence) given these thought vectors.

Let's try to visually understand by taking the same example.

Input sequence => "Rahul is a good boy"

(Expected) Output Sequence => "राहुल चांगला मुलगा आहे"

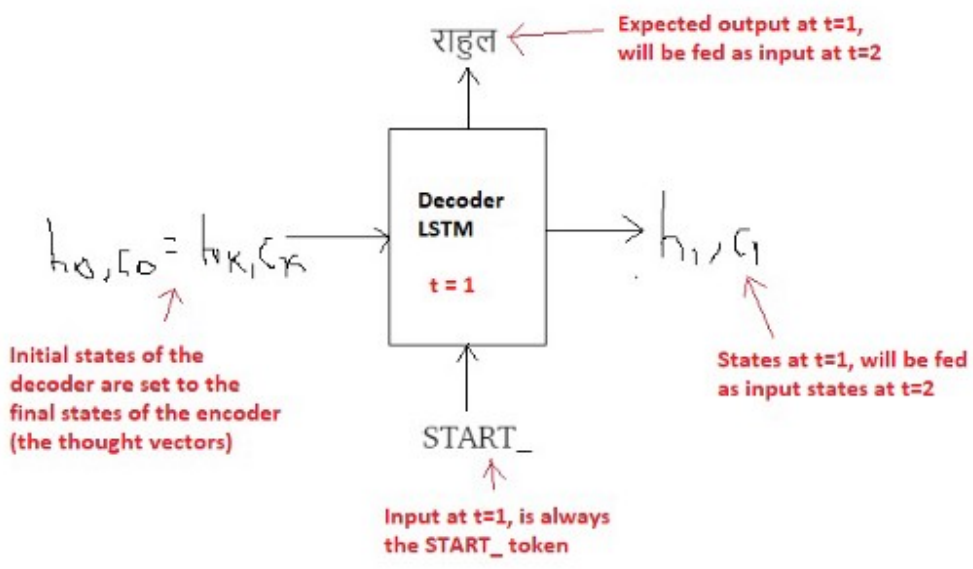
Step 1: Encode the input sequence into the Thought Vectors:



Encode the input sequence into thought vectors

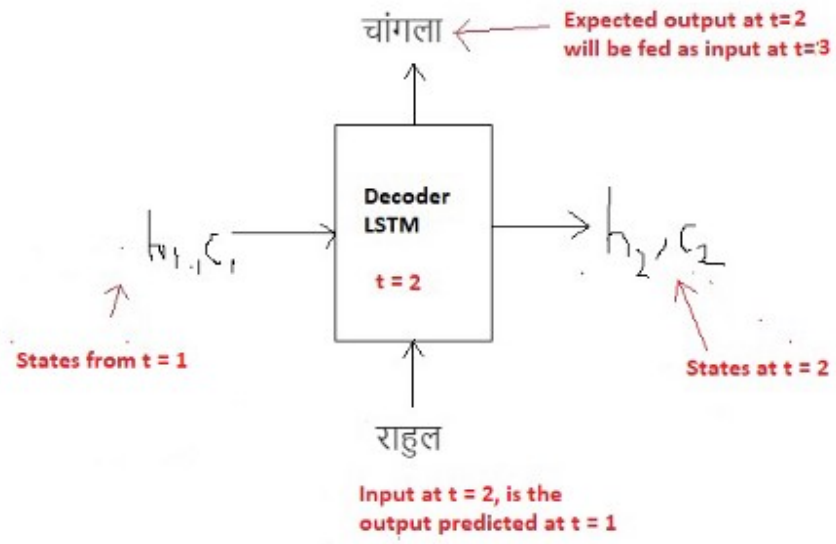
Step 2: Start generating the output sequence in a loop, word by word:

At t = 1



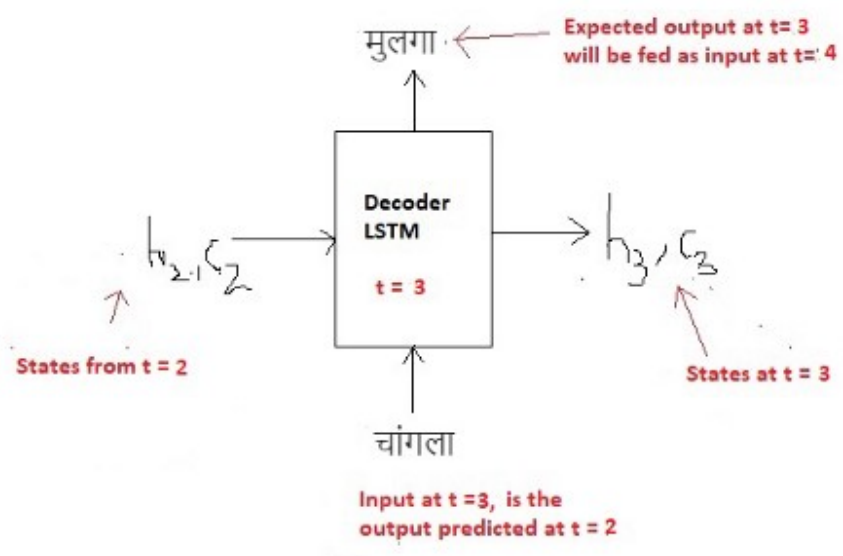
Decoder at t=1

At t = 2



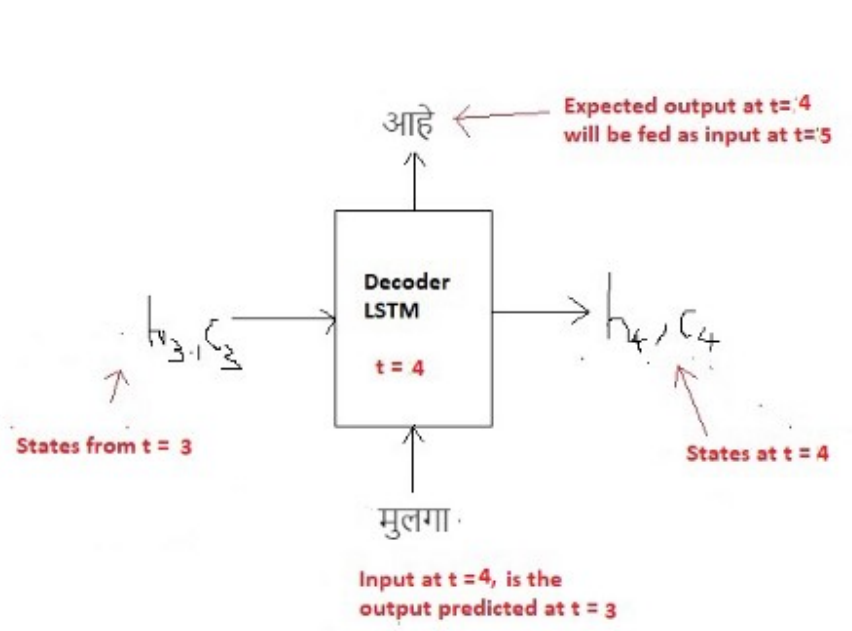
Decoder at t = 2

At t = 3



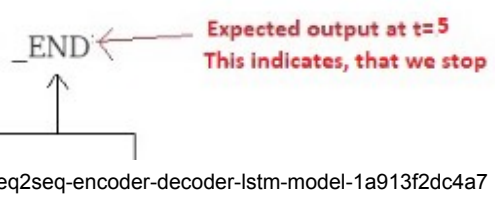
Decoder at t = 3

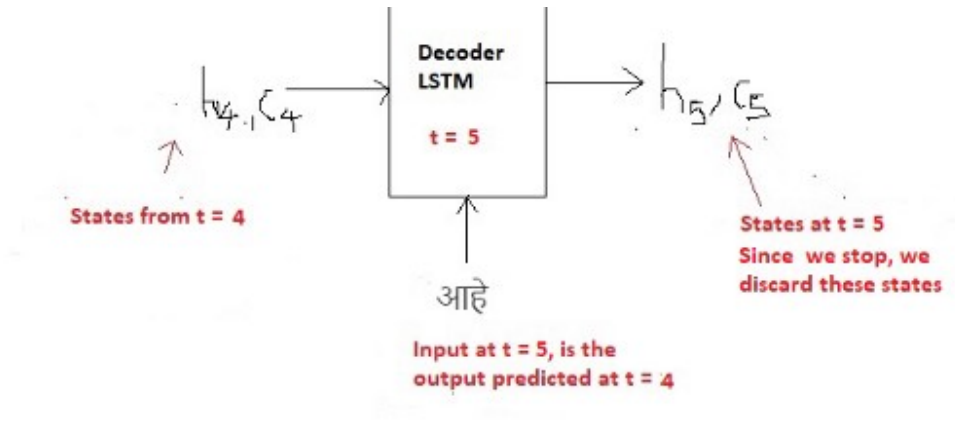
At t = 4



Decoder at t = 4

At t = 5





Decoder at t = 5

Inference Algorithm:

a. During inference, we generate one word at a time. Thus the Decoder LSTM is called in a loop, every time processing only one time step.

b. The initial states of the decoder are set to the final states of the encoder.

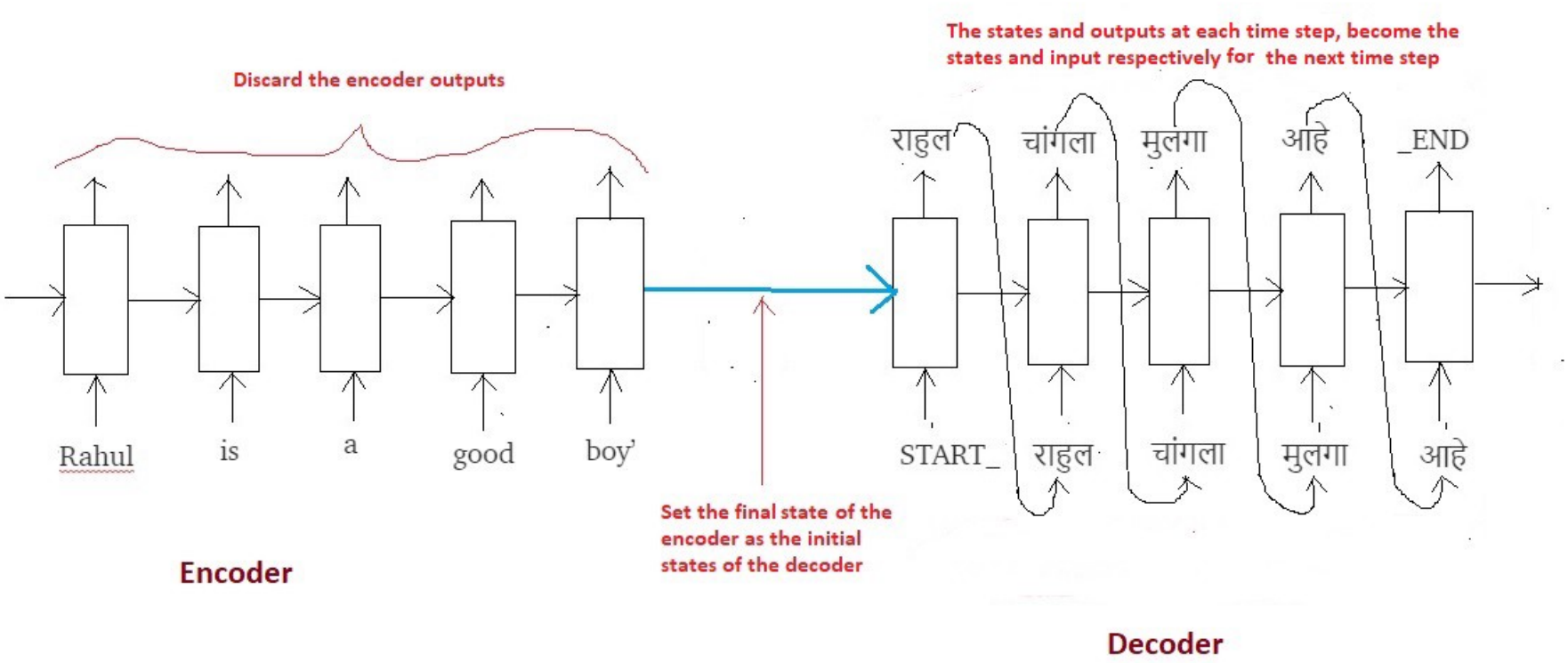
c. The initial input to the decoder is always the START_ token.

d. At each time step, we preserve the states of the decoder and set them as initial states for the next time step.

e. At each time step, the predicted output is fed as input in the next time step.

f. We break the loop when the decoder predicts the END_ token.

The entire inference procedure can be summarized in the below diagram:



Summary of the inference process

7. Code Walk through

Nothing beats the understanding developed when we actually implement the code, no matter how much efforts are put in to understand the theory (that does not however mean that we do not discuss any theory, but what I mean to say is theory must always be followed by implementation).

Dataset:

Download and unzip mar-eng.zip file from *here*.

Before we start building the models, we need to perform some data cleaning and preparation. Without going into too much details, I will assume the reader to understand the below (self explanatory) steps that are usually a part of any language processing project.

```
lines= pd.read_table('mar.txt', names=['eng', 'mar'])

# Lowercase all characters
lines.eng=lines.eng.apply(lambda x: x.lower())
lines.mar=lines.mar.apply(lambda x: x.lower())

# Remove quotes
lines.eng=lines.eng.apply(lambda x: re.sub("'", '', x))
lines.mar=lines.mar.apply(lambda x: re.sub("'", '', x))
exclude = set(string.punctuation) # Set of all special characters

# Remove all the special characters
lines.eng=lines.eng.apply(lambda x: ''.join(ch for ch in x if ch not in exclude))
lines.mar=lines.mar.apply(lambda x: ''.join(ch for ch in x if ch not in exclude))

# Remove all numbers from text
remove_digits = str.maketrans('', '', digits)
lines.eng=lines.eng.apply(lambda x: x.translate(remove_digits))
lines.mar = lines.mar.apply(lambda x: re.sub("[२३०८१५७९४६]", "", x))

# Remove extra spaces
lines.eng=lines.eng.apply(lambda x: x.strip())
lines.mar=lines.mar.apply(lambda x: x.strip())
lines.eng=lines.eng.apply(lambda x: re.sub(" +", " ", x))
lines.mar=lines.mar.apply(lambda x: re.sub(" +", " ", x))

# Add start and end tokens to target sequences
lines.mar = lines.mar.apply(lambda x : 'START_ '+ x + ' _END')
```

data_cleaning.py hosted with ♥ by GitHub view raw

Code to perform Data Cleaning

Below we compute the vocabulary for both English and Marathi. We also compute the vocabulary sizes and the length of maximum sequence for both the languages. Finally we create 4 Python dictionaries (two for each language) to convert a given token into an integer index and vice-versa.

Code for Data Preparation

Then we make a 90–10 train and test split and write a Python generator function to load the data in batches as follows:

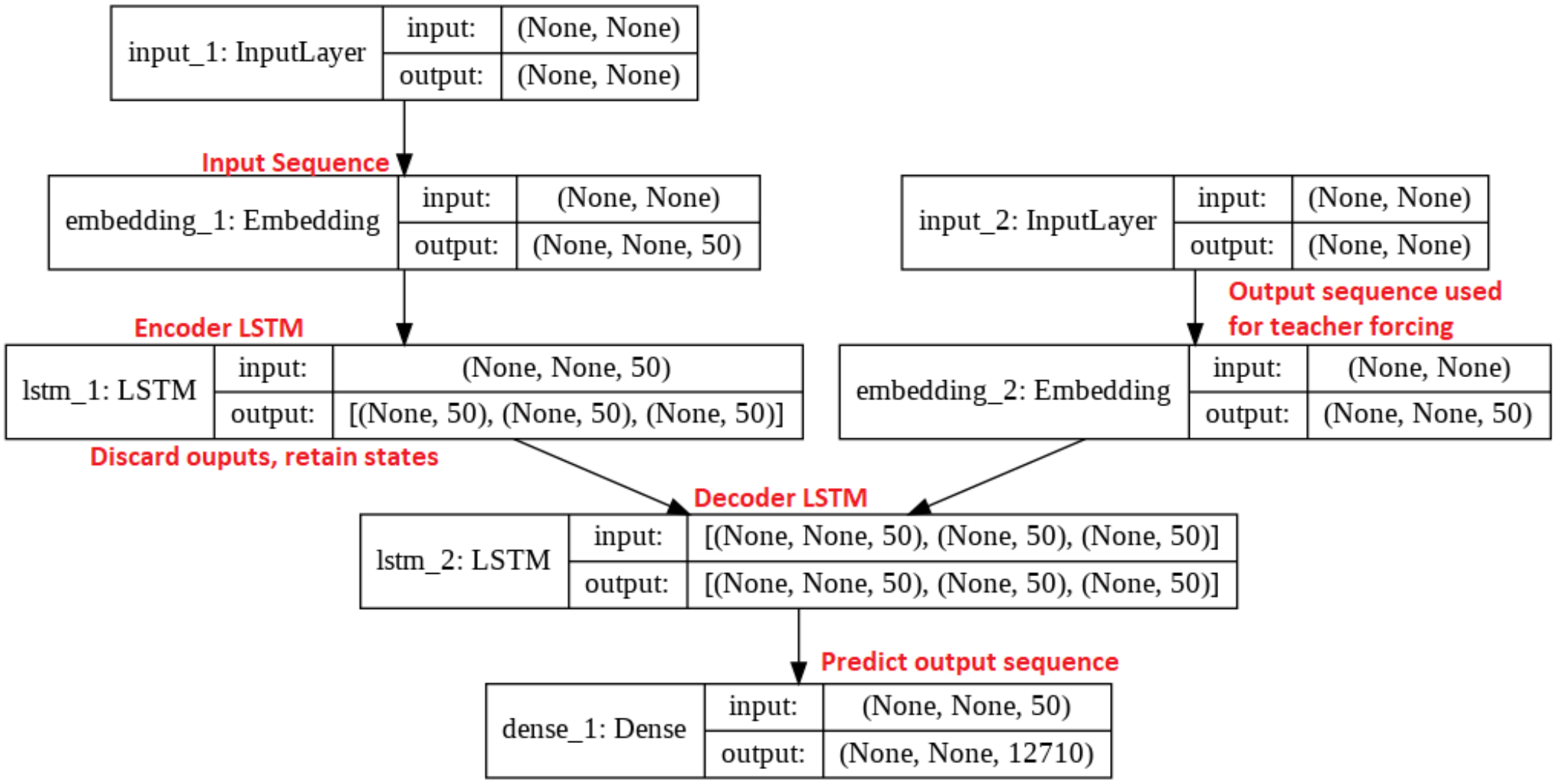
Code for loading Batches of Data

Then we define the model required for training as follows:

Code to define the Model to be trained

You should be able to conceptually connect each and every line with the explanation I have provided in sections 4 and 5 above.

Let’s look at the model architecture generated from the plot_model utility of the Keras.



Training the network

We train the network for 50 epochs with a batch size of 128. On a P4000 GPU, it takes slightly more than 2 hours to train.

Inference Setup:

Code for setting up Inference

Finally, we generate the output sequence by invoking the above setup in a loop as follows:

Code to decode the output sequence in a loop

At this point you must be able to conceptually connect each and every line of the code in the above two blocks with the explanation provided in section 6.

8. Results and Evaluation

The purpose of this blog post was to give an intuitive explanation on how to build basic level sequence to sequence models using LSTM and not to develop a top quality language translator. So keep in mind that the results are not world class (and you don't start comparing with google translate) for many reasons. The most important reason being is that the dataset size is very small, only 33000 pairs of sentences (yes these are too few). If you want to improve the quality of translations, I will list some suggestions towards the end of this blog. However for now, let's see some results generated from the above model (they are not too bad either).

Evaluation on train dataset:

Evaluation on Training Dataset

Evaluation on test dataset:

Evaluation on Test Dataset

What can we conclude?

Even though the results are not the best, they are not that bad as well. Certainly much better than what a randomly generated sequence would result in. In some sentences we can even note that the words predicted are not correct but they are semantically quite close to the correct words.

Also, another point to be noticed is that the results on training set are a bit better than the results on test set, which indicates that the model might be over-fitting a bit.

9. Future Work

If you are interested to improve the quality, you can try out below measures:

a. Get much more data. Top quality translators are trained on millions of sentence pairs.

b. Build more complex models like Attention.

c. Use dropout and other forms of regularization techniques to mitigate over-fitting.

d. Perform Hyper-parameter tuning. Play with learning rate, batch size, dropout rate, etc. Try using bidirectional Encoder LSTM. Try using multi-layered LSTMs.

e. Try using beam search instead of a greedy approach.

f. Try BLEU score to evaluate your model.

g. The list is never ending and goes on.

10. End Notes

If the article appeals you, do provide some comments, feedback, constructive criticism, etc.

Full code on my GitHub repo *here*.

If you like my explanations, you can follow me as I plan to release some more interesting blogs related to Deep Learning and AI.

You can connect on LinkedIn *here*

11. References

a. https://medium.com/@dev.elect.iitd/neural-machine-translation-using-word-level-seq2seq-model-47538cba8cd7

b. https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html

c. https://arxiv.org/abs/1409.3215

d. https://arxiv.org/abs/1406.1078

Machine Learning Deep Learning Lstm Seq2seq Machine Translation