SEQ2SEQ

(research paper)

"the main technical contribution of this work.. we use minimum innovation for maximum results"
- Ilya

- - Can skip to "Working" part - -

Despite being very powerful, DNNs cannot map sequences to sequences. They can only map vectors to vectors.

Learning to map sequences to sequence is very important in tasks like

Machine translation
Speech recognition
Image caption generation and many other tasks.

But we know that RNNs can work with sequences then the problem ?
In this u have sequence of inputs and sequence of outputs but both of them have the same alignment.
So they have one-to-One correspondence between them.
And because of that they have long term dependency problems due to exploding and vanishing gradient problem.

LSTMs are similar to RNNs but here we are less likely to get vanishing gradient problem
Key difference between LSTMs and RNNs ->

RNN -> overwrites the hidden state
LSTM -> Adds to the hidden state

This small difference is very significant because final hidden state is sum of many little Δ and because of this
Every Δ gets a gradient and doesnt vanish because of the sum.
And since it is a sum how does it know about order ?
Because LSTM decides what to add based on the sequence.

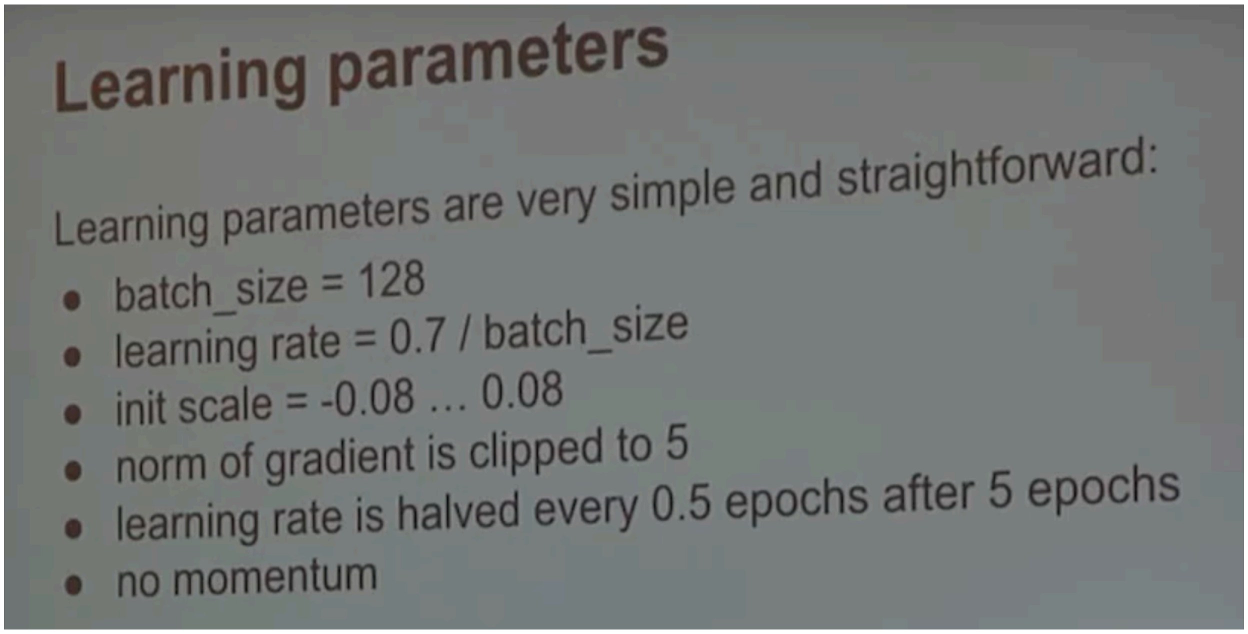
Main idea of seq2seq ->
Have LSTM first read the input sequence
And then produce the output sequence
Learning will take care of the rest.

Limitations ->
Very long sequences (above 70)
Out of vocab words

Dataset used -> WMT'14 (eng to french translation)
340m french words
303m english words
Trained on only 30% of training data

Model for large experiments ->
160k input words, 80k output words
4 layers of 1000D LSTM

384m parameters



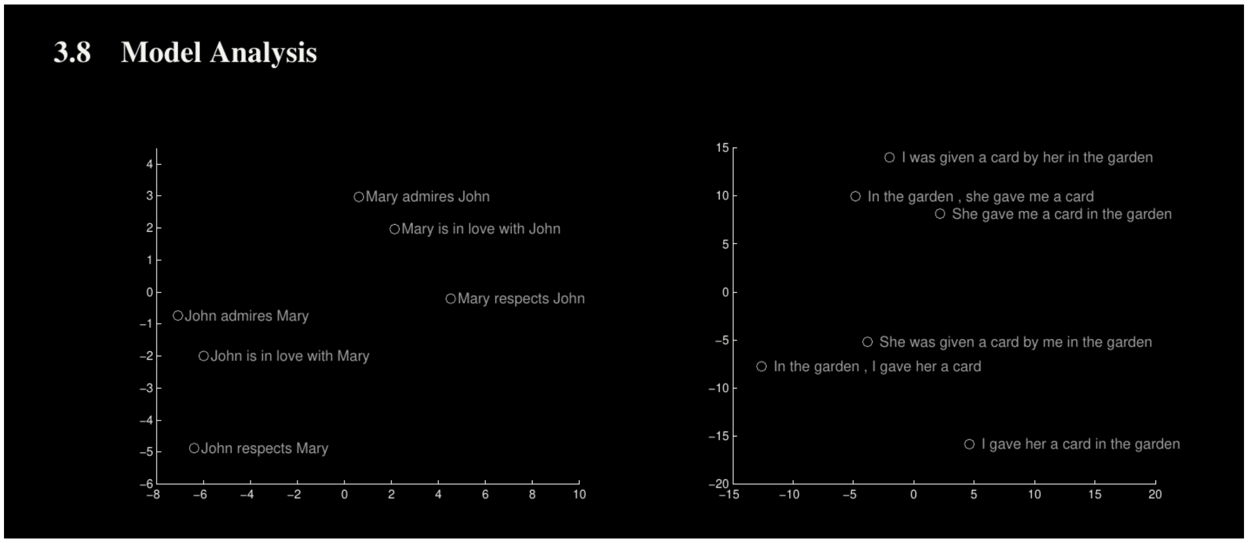
Learning parameters

Learning parameters are very simple and straightforward:

- batch_size = 128
- learning rate = 0.7 / batch_size
- init scale = -0.08 … 0.08
- norm of gradient is clipped to 5
- learning rate is halved every 0.5 epochs after 5 epochs
- no momentum

Uses multi-layer LSTM cells to map the input
sequence to the vector of fixed dimension
(Encoder).
And then another deep LSTM to decode the target
sequence from the vector (Decoder).

Note that the LSTM reads the
input sentence in reverse, because doing so
introduces many short term dependencies in the data
that make the
optimization problem much easier.



Sentences with similar meaning are close to each other (PCA projection)

Working

Encoder

1. LSTM cell is there
2. After sending sequence we get final cell state(c_t) and hidden state(h_t). That is the final representation of the sentence (contect vector)
3. Then we pass this context vector to the decoder.

Decoder

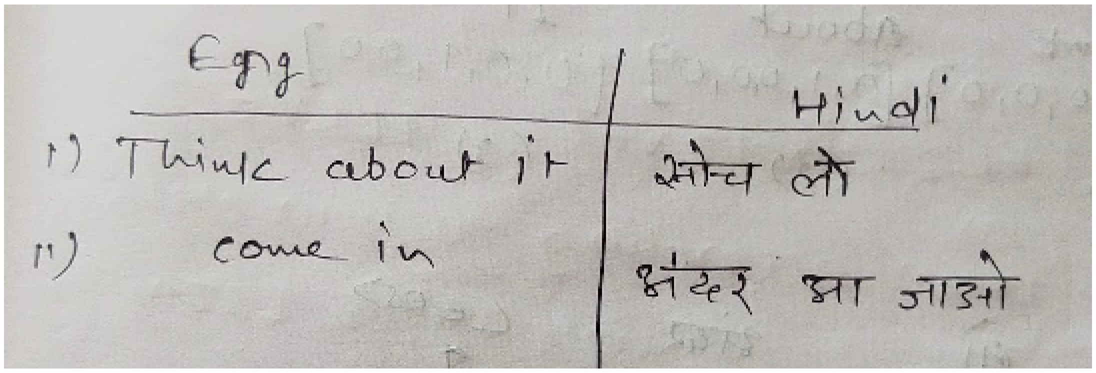
1. Contains LSTM
2. It will produce the o/p at each time step

3. Initial state of LSTM will be same as final state of encoder (context vector)
4. At time step = 1
   You send a special symbol for denoting that this is the start of the sentence (<sos>).
   Basically this tells decoder to start producing the o/p.
5. At time step = 2
   You pass the o/p of t=1 as i/p along with the internal states.
   Continue this till we get <eos> (end of sentence token) in the o/p.

Consider the example:

Taking the example of machine translation
For simplicity using one hot encoding but we generally use embeddings of the words
In case anyone is wondering how to generate embeddings of the words of a different language than english ?

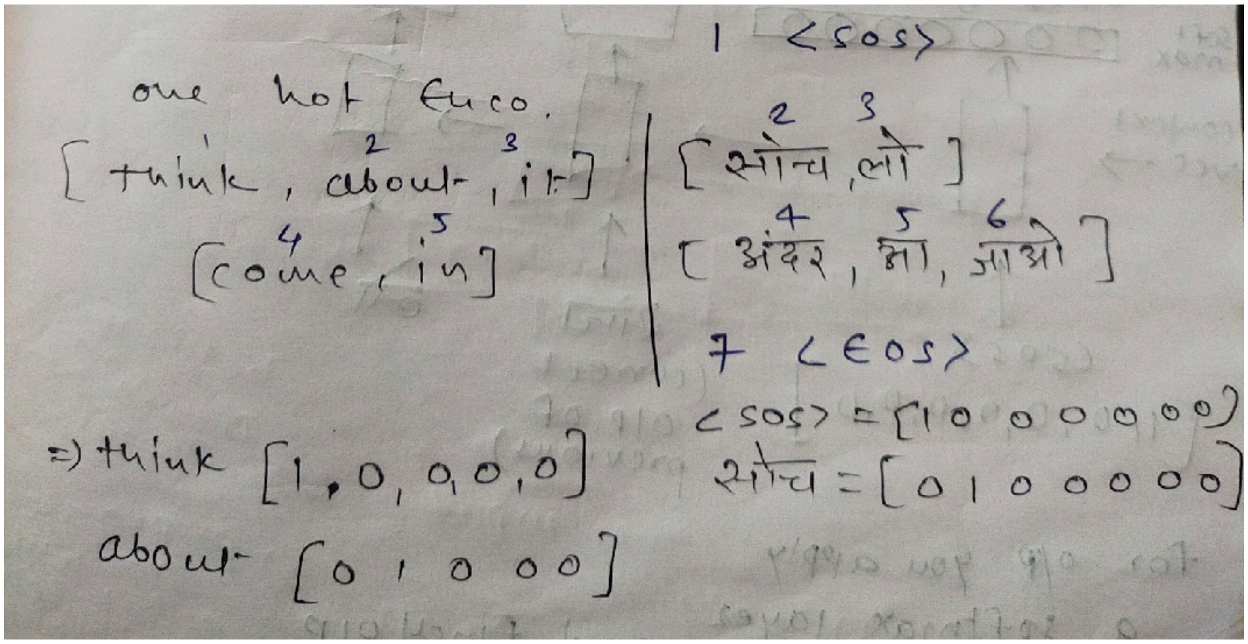
The process of generating embeddings for words in different languages doesn't fundamentally differ from English. The system does not "understand" the language itself but instead learns patterns and relationships between tokens (words, subwords, or characters) through numerical representations



Eng	Hindi
i) Think about it	सोच लो
ii) come in	अंदर आ जाओ

Consider english to hindi

Here we will apply OHE
But the number of words in the vocabulary of the output will differ because we have to add start and end of sentence special tokens (sos, eos)



one hot Enco.

$[\overset{1}{\text{think}}, \overset{2}{\text{about}}, \overset{3}{\text{it}}]$

$[\overset{4}{\text{come}}, \overset{5}{\text{in}}]$

1 <SOS>

$[\overset{2}{\text{सोच}}, \overset{3}{\text{लो}}]$

$[\overset{4}{\text{अंदर}}, \overset{5}{\text{आ}}, \overset{6}{\text{जाओ}}]$

7 <EOS>

=> think [1, 0, 0, 0, 0]

about [0 1 0 0 0]

<SOS> = [1 0 0 0 0 0 0]

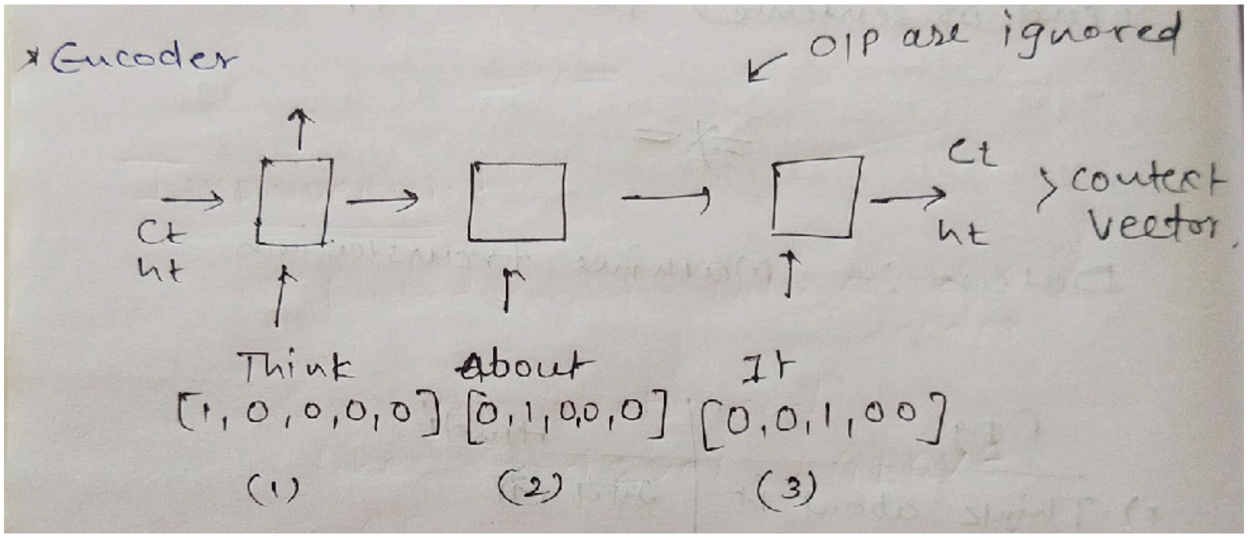
सोच = [0 1 0 0 0 0 0]

Training phase=>

Here we will train the model on our supervised data

Encoder ->

In encoder which is comprised of LSTM we pass our one hot encoded inputs at each time stamp
And at the end we get **context vector** with the help of cell state and hidden state (we concat c_t and h_t)



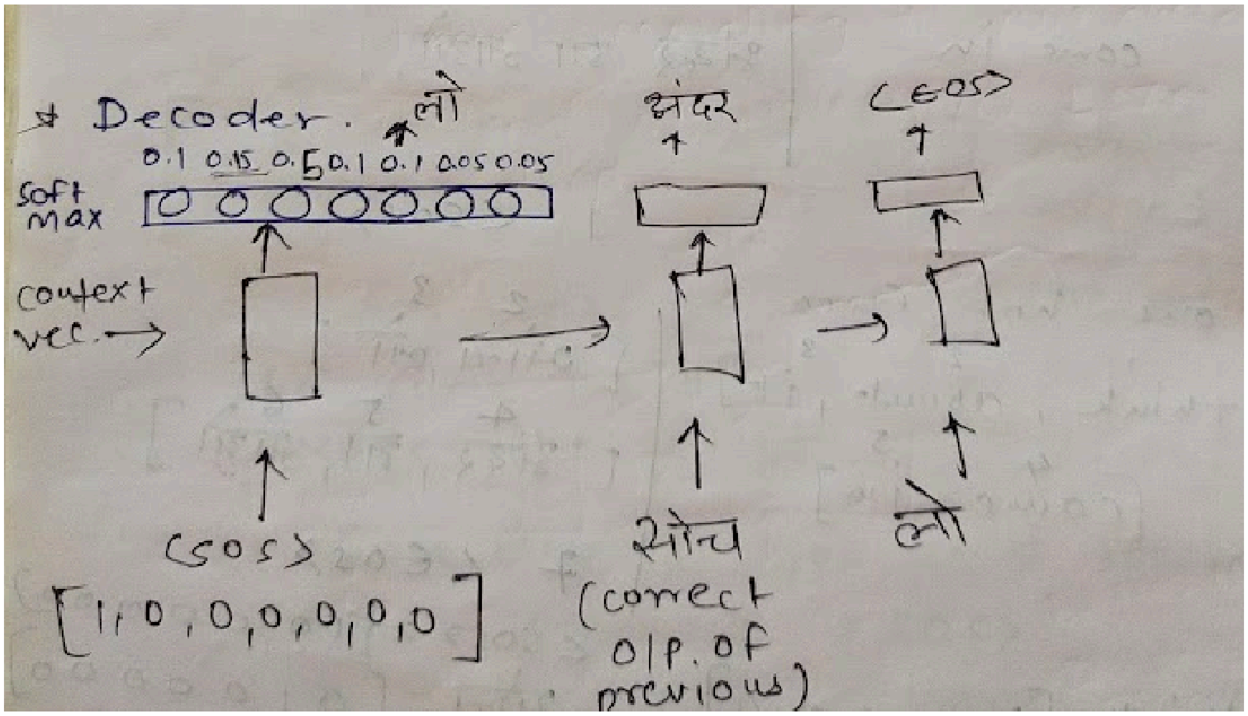
Decoder ->

<u>During training -></u>

Here decoder part is a bit tricky
It behaves differently during training than during inferencing(prediction).

During training
Decoder at each time step it is possible that you may get wrong output and we know that the output at t=1 will be used as input at t=2
So it affect it negatively if model predicts wrong at any time step

So instead of sending the o/p of previous time step we will be sending the correct output manually to the current timestamp
This concept is called **teacher forcing.**



So after one forward propagation you calculate the loss and try to minimize it through stochastic gradient descent.

<u>During prediction -></u>

During prediction obviously we don't know the correct o/p so we cannot do teacher forcing in the decoder.
So whatever o/p we get at a particular time stamp we pass it as the i/p of the next time stamp.
And we will stop when we get <eos>.

Improvements we can make:

1. Instead of OHE, use embeddings for both encoder and decoder
2. Instead of single layer LSTM use deep LSTM
3. In the encoder you can reverse the input.
   In the seq2seq paper they also found that reversing the i/p actually increased the accuracy
   They were not really sure about why this was the case but one possible reason is that it propagates the gradients a bit better.