Sumarizzing

Programming Club

Introduction to NLP(3/3)

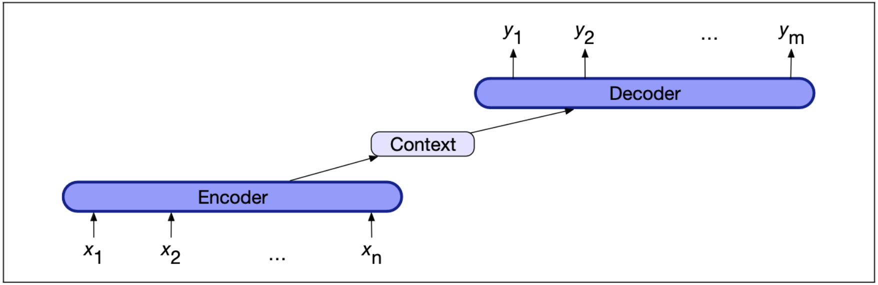
Content to be covered...

Content to be covered..

Encoder Decoder Architecture with RNNs

- The model is used when the input sequence is taken and is being (say) translated to an output sequence that is of a different length, and doesn't align with it in a word-to-word way.
- Encoder-decoder networks have been applied to a very wide range of applications including summarization, question answering, and dialogue, but they are particularly popular for machine translation.
- The key idea underlying these networks is the use of an **encoder** network that takes an input sequence and creates a contextualized representation of it, often called the **context**. This representation is then passed to a **decoder** which generates a task-specific output sequence.

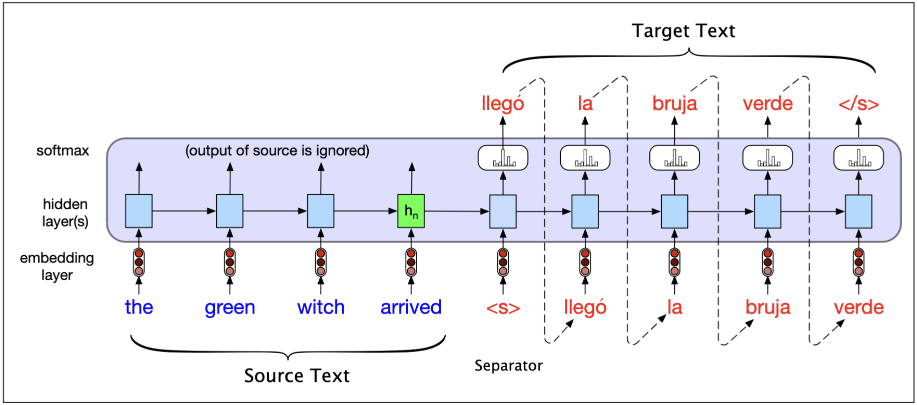
Encoder Decoder Architecture with RNNs



The architecture thus includes the following three components:

- An encoder that accepts an input sequence, x_1^n, and generates a corresponding sequence of contextualized representations, h_1^n. LSTMs, convolutional net-works, and Transformers can all be employed as encoders
- A context vector, c, which is a function of h_1^n, and conveys the essence of the input to the decoder.
- A decoder, which accepts c as input and generates an arbitrary length sequence of hidden states h_1^m , from which a corresponding sequence of output states y_1^m , can be obtained. Just as with encoders, decoders can be realized by any kind of sequence architecture.

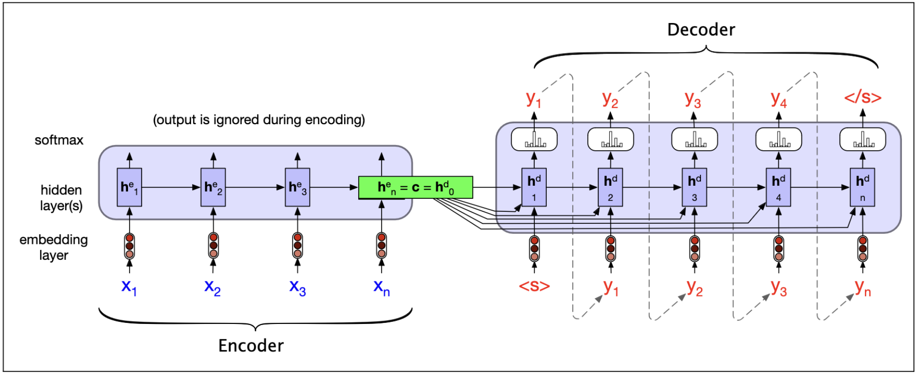
- The aim is to build a encoder-decoder model by starting with the conditional RNN language model $p(y)$, the probability of the sequence y.
  $p(y) = p(y_1)p(y_2|y_1)...p(y_m|y_1, y_2, ..., y_{m-1})$
- In RNN language modeling, at a particular time t, we pass the prefix of $t-1$ tokens through the language model, using forward inference to produce a sequence of hidden states, ending with the hidden state corresponding to the last word of the prefix. We then use the final hidden state of the prefix as our starting point to generate the next token.
- One slight change to turn this language model with autoregressive generation into an encoder-decoder model that is a translation model that can translate from a source text in one language to a target text in a second: add a sentence separation marker at the end of the source text, and then simply concatenate the target text.
  seq2seq paper
- Thus an encoder-decoder model tries to computes the probability
  $p(y|x) = p(y_1|x)p(y_2|y_1, x)...p(y_m|y_1, y_2, ..., y_{m-1}, x)$



Target Text
llegó
la
bruja
verde
</s>
softmax
(output of source is ignored)
hidden layer(s)
h_n
embedding layer
the
green
witch
arrived
<s>
llegó
la
bruja
verde
Source Text
Separator

- The encoder can comprise of stacked RNNs, biRNNs, LSTMs, biLSTMs, etc. The entire purpose of the encoder is to generate a contextualized representation of the input. This representation is embodied in the final hidden state of the encoder h_n^e. This representation is also called c or the context.
- the first decoder RNN cell uses c as its prior hidden state h_0^d. The decoder autoregressively generates a sequence of outputs, an element at a time, until an end-of-sequence marker is generated. Often the context vector is made available at each step of the decoding process.

$$\begin{aligned} c &= h_n^e \\ h_0^d &= c \\ h_t^d &= g(\hat{y}_{t-1}, h_{t-1}^d, c) \\ z_t &= f(h_t^d) \\ y_t &= softmax(z_t) \end{aligned}$$


Decoder
(output is ignored during encoding)
softmax
hidden layer(s)
embedding layer
h^e_1
h^e_2
h^e_3
$h^e_n = c = h^d_0$
h^d_1
h^d_2
h^d_3
h^d_4
h^d_n
y_1
y_2
y_3
y_4
</s>
x_1
x_2
x_3
x_n
<s>
y_1
y_2
y_3
y_n
Encoder

Content to be covered..

Attention

- The last hidden state is a **bottleneck** as it must represent absolutely everything about the source text, since it is the only thing the decoder knows about the source text is what is in the context vector. Information at the beginning of the sentence, especially for long sentences, may not be equally well represented in the context vector.
- The attention mechanism is a solution to the bottleneck problem, a way of allowing the decoder to get information from all the hidden states of the encoder, not just the last hidden state.
- The idea of attention is instead to create the single fixed-length vector c by taking a weighted sum of all the encoder hidden states. The weights focus on ('attend to') a particular part of the source text that is relevant for the token the decoder is currently producing.

- This context vector c_i is generated anew with each decoding step i and takes all the encoder hidden states into account in its derivation. We then make this context available during decoding by conditioning the computation of the current decoder hidden state on it. $h_i^d = g(\hat{y}_{t-1}, h_{i-1}^d, c_i)$.
- To capture relevance of each encoder state to the decoder state captured in h_{i-1}^d, one has to measure how similar the decoder hidden state is to an encoder hidden state. Thus **dot-product attention** implements this relevance by similarity.

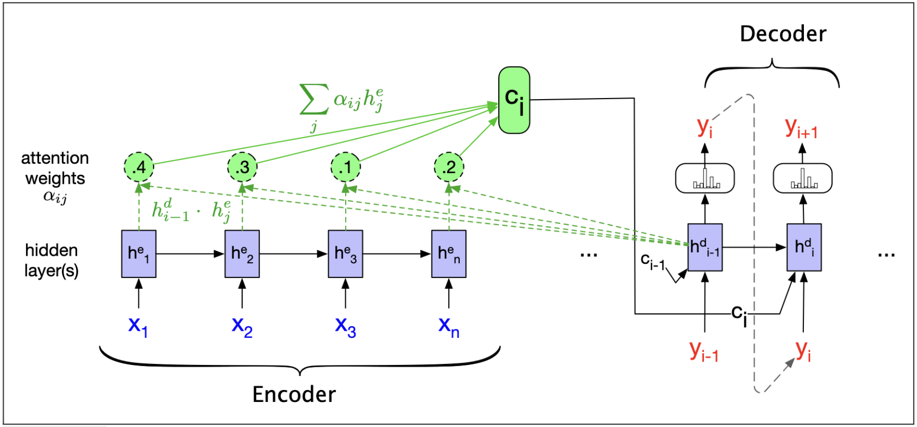
  $$score(h_{i-1}^d, h_j^e) = h_{i-1}^d . h_j^e$$

  Then these scores are normalized to get weights to arrive at a weighted sum.

  $$\alpha_{ij} = softmax(score(h_{i-1}^d, h_j^e) \forall j \in e)$$

  Thus a context vector becomes the follows:

  $$c_i = \sum_j \alpha_{ij} h_j^e$$



There can however be even more sophisticated attention scores.

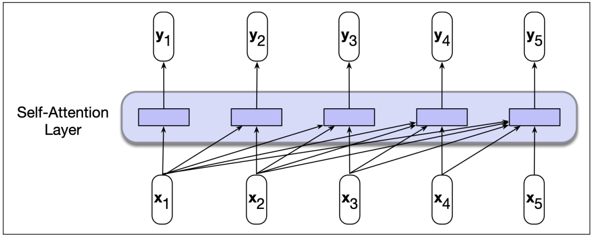
Content to be covered..

Pre-trained models and Self-Attention

- The crucial insight of the distributional hypothesis is that the knowledge that we acquire through this process can be brought to bear during language processing long after its initial acquisition in novel contexts.
- Pre-pretraining: the process of learning some sort of representation of meaning for words or sentences by processing very large amounts of texts. Such models are called **pre-trained language models**.
- **Transformer** are architecture, which like LSTM can handle distant information. Transformers map input vectors $(x_1, x_2, ..., x_n)$ to a sequence of output vectors $(y_1, y_2, ..., y_n)$ of the same length.
- Transformers are made up of stack of transformer block each of which is a multilayer network made of combining simple linear layers, feedforward networks and **self-attention** layers.

Self-attention

- Self-attention allows network to directly extract and use information from arbitrarily large contexts without the need to pass it through intermediate recurrent connections as in RNNs.



Information flow in a causal (or masked) self-attention mode

- When processing each item in the input, the model has access to all of the inputs up to and including the one under consideration, but no access to information about inputs beyond the current one. In addition, the computation performed for each item is independent of all the other computations.

The computation of y_n is dependent on the comparisons of between the input x_n and its preceding elements.

- The simplest form of comparison between elements in a self-attention layer is a dot product. As seen previously a weighted sum of input sequences based on attention weights can be used to arrive at the output tokens. However transformer use more sophisticated ways of representing how words can contribute to representation of longer inputs.

Consider the three different roles that each input embedding plays during the course of the attention process.

- As the *current focus of attention*: **query**.
- In its role as a *preceding input*: **key**.
- As the **value** used to compute the output for the current focus.

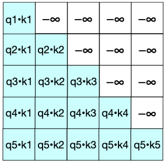
Introduction of three weight matrices: W^Q, W^K, W^V. Thus each input vector x_i is projected onto its role as key, query, and value.

$$q_i = W^Q x_i, k_i = W^K x_i, v_i = W^V x_i$$

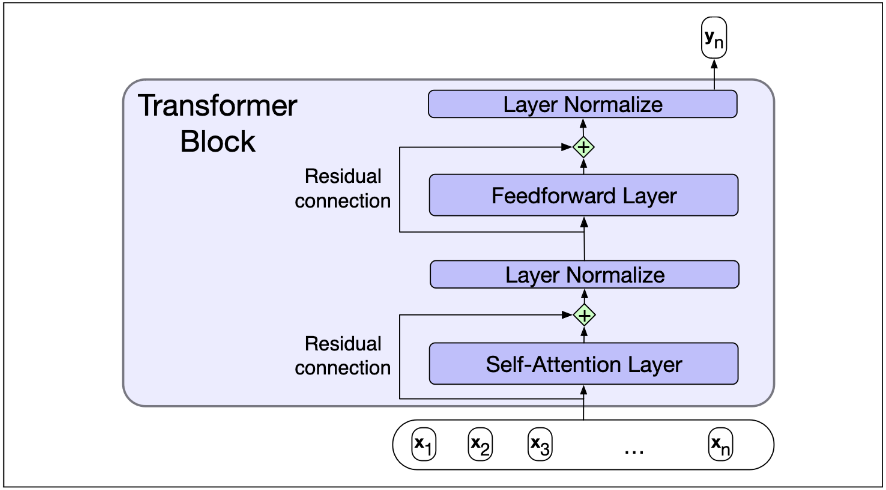
Given these projections the score would now be: $score(x_i, x_j) = q_i.k_j$. And therefore the after the calculation of α_{ij} using softmax, the output calculations are $y_i = \sum_{j \leq i} \alpha_{ij} v_j$.
Note: normalization of the dot product is typically done by dividing by square root of the query or key value dimensions.
Thus, finalizing:

$$Q = XW^Q; K = XW^K; V = XW^V$$
$$SelfAttention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Unfortunately, this process goes a bit too far since the calculation of the comparisons in QK^T, results in a score for each query value to every key value, including those that follow the query. This is inappropriate in the setting of language modeling since guessing the next word is pretty simple if you already know it. We introduce a mask therefore.



q1•k1	$-\infty$	$-\infty$	$-\infty$	$-\infty$
q2•k1	q2•k2	$-\infty$	$-\infty$	$-\infty$
q3•k1	q3•k2	q3•k3	$-\infty$	$-\infty$
q4•k1	q4•k2	q4•k3	q4•k4	$-\infty$
q5•k1	q5•k2	q5•k3	q5•k4	q5•k5



y_n
Transformer Block
Layer Normalize
Residual connection
Feedforward Layer
Layer Normalize
Residual connection
Self-Attention Layer
x_1
x_2
x_3
…
x_n

Residual connections are connections that pass information from a lower layer to a higher layer without going through the intermediate layer. Allowing information from the activation going forward and the gradient going backwards to skip a layer improves learning and gives higher level layers direct access to information from lower layers.

$$z = LayerNorm(x + SelfAttention(x))$$
$$y = LayerNorm(z + FFN(z))$$

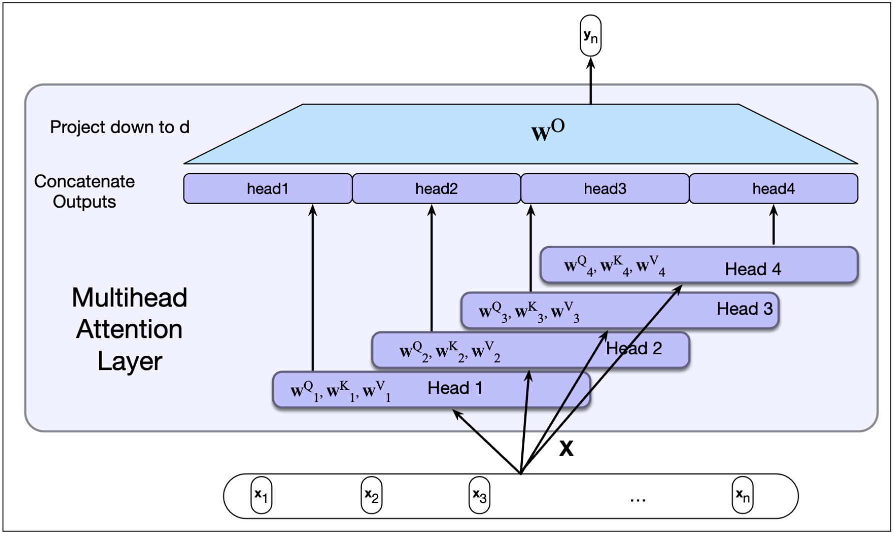
Layer norm is a variation of the standard score, or z-score, from statistics applied to a single hidden layer. In the standard implementation of layer normalization, two learnable parameters γ and β are introduced and thus $LayerNorm = \gamma\hat{x} + \beta$ where $\hat{x}$ is the normalized vector x.

MultiHead Attention

- It would be difficult for a single transformer block to learn to capture all of the different kinds of parallel relations among its inputs. Transformers address this issue with multihead self-attention layers.
- These are sets of self-attention layers, called heads, that reside in multihead self-attention layers parallel layers at the same depth in a model, each with its own set of parameters. Each head can learn different aspects of the relationships that exist among inputs at the same level of abstraction.

$$MultiHeadAttention(X) = (head_1 \oplus head_2 ... \oplus head_n)W^O$$
$$Q = XW_i^Q; K = XW_i^K; V = XW_i^V$$
$$head_i = SelfAttention(Q, K, V)$$

Dimensions of the various matrices: $W_i^Q, W_i^K \in \mathbb{R}^{d \times d_k}$, $W_i^V \in \mathbb{R}^{d \times d_v}$, and $W^O \in \mathbb{R}^{hd_v \times d}$.

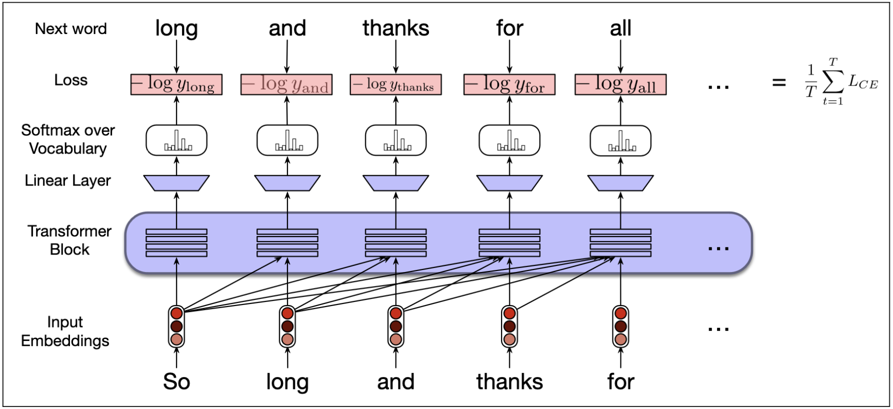


y_n
Project down to d
W^O
Concatenate Outputs
head1
head2
head3
head4
W^Q_4, W^K_4, W^V_4 Head 4
W^Q_3, W^K_3, W^V_3 Head 3
W^Q_2, W^K_2, W^V_2 Head 2
W^Q_1, W^K_1, W^V_1 Head 1
Multihead Attention Layer
X
x_1 x_2 x_3 … x_n

Positional embeddings

In order to incorporate position of word tokens in a sequence, we combine the input embeddings with **positional embeddings**.
Read more about it...

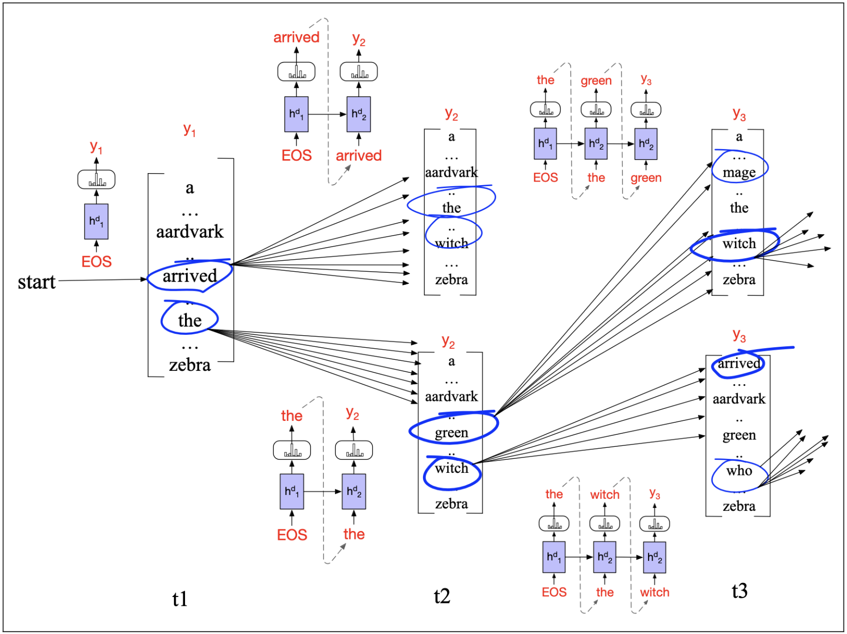
Transformers as Language Model



Autoregressive generation or Causal LM generation.

Beam Search

Greedy approach may not be optimal. A best choice may turn out to be the wrong one after some generations. We deploy a **search tree** based approach instead.
Instead, decoding in sequence generation problems generally uses a method called beam search. In beam search, instead of choosing the best token to generate at each timestep, we keep k possible tokens at each step. This fixed-size memory footprint k is called the **beam width**.



start
y_1
h^d_1
EOS
y_1
a
…
aardvark
..
arrived
..
the
…
zebra
arrived
y_2
h^d_1
h^d_2
EOS
arrived
y_2
a
…
aardvark
..
the
..
witch
…
zebra
the
green
y_3
h^d_1
h^d_2
h^d_2
EOS
the
green
y_3
a
…
mage
..
the
witch
…
zebra
the
y_2
h^d_1
h^d_2
EOS
the
y_2
a
…
aardvark
..
green
..
witch
zebra
y_3
arrived
…
aardvark
..
green
..
who
…
zebra
the
witch
y_3
h^d_1
h^d_2
h^d_2
EOS
the
witch
t1
t2
t3

Topics that are left include BERT, and using transformer models for other NLP applications. Will refer some text and resources.
Thank you!