Introduction

Recurrent models typically do an update rule of the form $h_t = f(h_{t-1}, x_t)$

- they are sequential by nature so you cannot parallelize processing
  - parallelization becomes especially important in long sequences where memory availablity becomes a constraint
  - some work has achieved improvements through factorization tricks and conditional computation
  - but the fundamental problem remains

Until now, attention mechanims were used in sequence modelling and transduction models, usually in conjunction with a recurrent network(s)
But can we eliminate the use of recurrent networks entirely? The Transformer architecture is an attempt at this.

Background

Other attempts at reducing the reliance on sequential computation include egs that use CNNs as basic building blocks, computing hidden representations in parallel for all input and output positions

- Q: How? What does this mean?
  - presumably they only look at local neighbourhoods for signal and not the entire sequence?
- The number of operations needed to relate signal from two arbitrary input and output positions grows in the distance between positions
  - which still makes it difficult to gather signal from distant positions
- The transformer reduces this to a constant number of ops
  - at the cost of reduced resolution though: it averages agttention-weighted positions
  - but the multi head attention counteracts this downside (somehow)

Self attention

- attention mechanism that relates different positions of a single sequence to learn a representation of the sequence
- The intuition is: can you use self-attention alone to compute represnetations of the input and outpuf of a model eithough using sequence aligned RNNs or convolution?

Model architecture

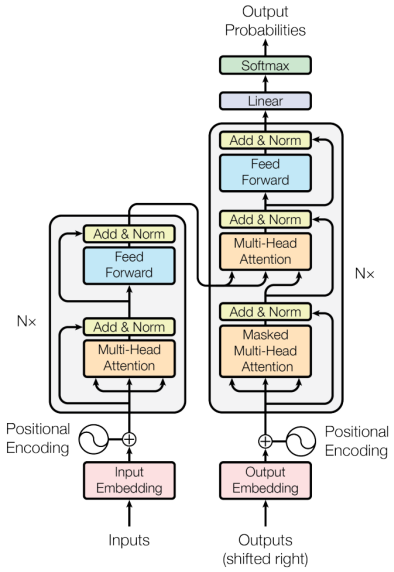

Figure 1: The Transformer - model architecture.

The Transformer follows the general encoder/decoder structure

- encoder maps input sequence of symbol representations $(x_1, x_2, ..., x_n)$ to a sequence of continuous representations $(z_1, z_2, ..., z_n)$
- Given **z,** the decoder generates output $(y_1, y_2, ..., y_n)$ of symbols, one element at a time
- at each tstep the model is auto-regressive, consuming a previously generated symbol as additional input when generating the next

Transformer encoder

- comprises n=6 identical layers, each layer has two sub;ayers
- Sublayer 1: multi head self-attention mechanism
  - Q: what is self attention vs attention?
    * Does it just mean that that the attention mechanism is trained on itself?
    * is it trained on positions x_{t+n} or is it causal? the causal piece perhaps doesn't matter since we just want to capture context
- Sublayer 2: position wise fully connected feed-forward network
  - between each of the d=512 dims? does this capure some intra-embedding context? Seems a bit strange?
- Employ a residual connection around each of the two sub layers, followed by layer normalization
  - Q: what does residual connection mean?
  - https://arxiv.org/pdf/1512.03385.pdf

- The output of each sub layer is $LayerNorm(x + sublayer(x))$ where sublayer() is the function implemented by the sublayer itself
- The embedding layer + every sublayer in the model produces outputs of dim=512
- Q: does the length of the output sequence need to match the length of the input sequence?

Transformer decoder

- comprises n=6 identical layers
- in addition to the two sublayers in each decoder layer, there is a third sublayer which performs multihead attention over the output of the encoder stack
- So there is 1. multi head self attention 2. multi head input attention 3. dense layer
- Q: ??
  - I think this is saying: $y_t = f(y_{t-1}, g(z_t))$ and $g(z_t)$ = multihead attention
    * I suppose this is during train time. Use dependencies from the previous output, and from the current input
  - but is there multi head attention (presumably with regards to y_{t+n} before it has been generated? presumably outputs need to be 'causal' since you have not generated future outputs yet?
- The self attention layer in the decoder stack does not attend to subsequent positions - hence the 'mask'
- This, combined with the fact that output embeddings are offset by one position (Q: what does this mean) means that predictions for position i can only depend on known outputs for positions < i

Q: assuming input sequence "This is a bird."
Encoder
→ embedding layer converts each word into a d=512 vector
so now input has shape 4x512?
→ does the self attention layer spit out one output per element? or one output for the whole sequence?
i.e. is the output here 4x512 or 1x512?

- I imagine it is 4x512 -- each position gets a position specific attention vector?
- how is this vector generated/trained? is it an autoencoder trained to predict the input sequence?
  - the whole point of this paper is to stop using recurrent networks

→ then output norm(x + selfattention(x))
→ then run a dense layer? for what? is this per element? So the result is 4x512?
→ then output norm(norm(x + selfattention(x)) + dense(norm(x + selfattention(x))))
→ then re run (for n = 6 times) the same process?
→ then feed as input to the decoder...?

I don't understand

- what is the input to a given layer
- what is the output from a given layer

Attention

Attention: Q, (K, V) → O, where all of Q, K, V, and O are all vectors
output = weighted sum of the values V
the weight for each value is a compatibility function of the query with the corresponding key

Editorial: is it
o = None
for each k, v in K, V:
w_k ← compatibility(Q, k)
o ← o + w_k*v

Q: What is this Q, K, V? Where do they come from??

- Why isnt the key the same as the value? Assuming both are supposed to encode something about what the vector contains?

<u>Scaled dot-product attention</u>
Queries and keys have to share the same dimension, d_k. Values have dimension d_v
compute $q_i^T k_i / \sqrt{d_k}$ for all i, and apply softmax to obtain the weights on the values
→ can just do $softmax\left(QK^T / \sqrt{d_k}\right) * V$

Additive attention uses a feed-forward network with a single layer

- Q: what is the inner product between here?
- I think additive attention is the weighted sum of hidden states?
-

Multi head attention

Instead of a single attention function with d_{model} dimensional K, V, Q -- can you instead linearly project the queries, keys, and valyes h times with different, learned linear projects? to $d_k, d_k, and\ d_v$ dimensions?
On each of these projections of q, k, v → perform the attention function in parallel, yielding $d_v$dimensional values (h of them presumably)?
then concat these values and project again -- Q: onto what?

- Q: Are these K, V, Q matrices learned?

This allows the model to jointly attend to information from differente representation subspaces at differente positions → Q?

$$MultiHead(Q,\ K,\ V)\ =\ Concat(head_1,\ head_2,\ ...,\ head_h)W^O$$
$$where\ head_i\ =\ Attnention\left(QW_i^Q,\ KW_i^K,\ VW_i^V\right)$$

- each head gets its own linear projection matrix
- $W_i^Q\ \in\ R^{d_{model}xd_k},\ W_i^K\ \in\ R^{d_{model}xd_k},\ W_i^V\ \in\ R^{d_{model}xd_v}$
- the ultimate d_v dimensional results can be way smaller than the original - if for eg you do $d_v\ =\ d_{model}/h$ then the total computational cost across the heads will be similar to that of the single head attention with full dim

The transformer uses attention in three ways
1. encoder-decoder attention: queries from previous decoder layer, memory keys and values from the output of the encoder
- this allows every position in the decoder to attent over all positions in the input sequence
    - this mimics the typical encoder-decoder attenghbtion mechanism in seq-to-seq models

2. encoder: contains self-attention layers. all the keys, values and queries come from the previous layer of the encoder. Each position in the encoder attends to all positions in the previous layer of hte encoder
3. decoder: contains self-attentoin layers. allows each postiion to attend to all positions in the decoder up to and including that position. (prevent leftward information flow to preserve auto regressive property)
    - implement this by masking out all values (-inf) in the input of the softmax that correspond to illegal connections

Position wise feed forward networks
Each layer in the encoder and decoer contains a FC feed forward netowrk, which is applued to each position identically and separately.
$$FFN(x)\ =\ \max(0,\ xW_1\ +\ b_1)W_2\ +\ b_2$$

- each layer uses different parameters
- analogous to 2 convolutions with kernel size 1 -→ Q?? What does this mean?
- input and putput dim = $d_{model}\ =\ 512$ in this paper
- inner layer has dimensionality $d_{ff}\ =\ 2048$
    - Q: what is this inner dimesionality? presumably the dimenstionality of xW_1

Embeddings

- use learned embeddings to convert input tokens to vectors of dimension d_{model}
  - Q: how are these embeddings learned?
- use the usual learned linear transformation and softmax to convert decoder output to next token prob
- share the same weight matrix across the input and output embedding layers

Note that this model has no recurrent and no convolution! So how we do encode ordering?

Positional encoding

- need to somehow inject information about hte relative or absolute position of the tokens in the sequence
- These have the same dimension d_{model}as the embeddings, so the two can be summed
- Each dimension of a positional encoding corresponds to a sinusoid → Q?
- The wavelenghts form a gp from $2\pi \ to \ 10000.2\pi$
  - $\sin(2\pi f x) \ = \ \sin(2\pi.1/\lambda.x)$
- Q: I don't understand this at all
- $PE_{(pos,\ 2i)} \ = \ \sin\left(pos/10000^{2i/d_{model}}\right)$
- $PE_{(pos,\ 2i+1)} \ = \cos\left(pos/10000^{2i/d_{model}}\right)$
- since d = d_{model}, $2i/d_{model} \ = \ 1$ at some point
- for any fixed offset k, PE_{pos+k} is a linear function of PE_{pos} → how?
  - Q: and why does this help/matter?
- Q: So these positional encoding are not learned? They are just fixed a priori? Then added to the learned word embeddings?

Why self attention

Three criteria
1. total computational complexity per layer
2. parallelization
3. path length between long range dependencies -- how many hops do forward and backward signals need to traverse in the network? the shorter this path length, the easier it is to learn long-range dependencies

A self attentoin layer connects all positions with a constant number of sequentially executed operations, while a recurrent layer needs $O(n)$ sequential operations

http://nlp.seas.harvard.edu/2018/04/01/attention.html

https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial6/Transformers_and_MHAttention.html

If we do not scale down the variance back to , the softmax over the logits will already saturate to for one random element and for all others. The gradients through the softmax will be close to zero so that we can't learn the parameters appropriately.

- I dont understand this scaling thing
- If there is large spread in values to which softmax is applied, then the small values get really small softmax values and this causing vanish gradients
-

```
>>> l_2 = [a/3 for a in l]
>>> l_2
[1.3333333333333333, 1.6666666666666667, 2.0]
>>> softmax(l_2)
array([0.23023722, 0.32132192, 0.44844086])
>>> softmax(l)
array([0.09003057, 0.24472847, 0.66524096])
>>>
```

-

One crucial characteristic of the multi-head attention is that it is permutation-equivariant with respect to its inputs. This means that if we switch two input elements in the sequence, e.g. (neglecting the batch dimension for now), the output is exactly the same besides the elements 1 and 2 switched. Hence, the multi-head attention is actually looking at the input not as a sequence, but as a set of elements. This property makes the multi-head attention block and the Transformer architecture so powerful and widely applicable! But what if the order of the input is actually important for solving the task, like language modeling? The answer is to encode the position in the input features, which we will take a closer look at later (topic Positional encodings below).

- Strangely the transfomer itself is a 'bag of words' style approach

The residual connection is crucial in the Transformer architecture for two reasons:

Similar to ResNets, Transformers are designed to be very deep. Some models contain more than 24 blocks in the encoder. Hence, the residual connections are crucial for enabling a smooth gradient flow through the model.

Without the residual connection, the information about the original sequence is lost. Remember that the Multi-Head Attention layer ignores the position of elements in a sequence, and can only learn it based on the input features. Removing the residual

connections would mean that this information is lost after the first attention layer (after initialization), and with a randomly initialized query and key vector, the output vectors for position has no relation to its original input. All outputs of the attention are likely to represent similar/same information, and there is no chance for the model to distinguish which information came from which input element. An alternative option to residual connection would be to fix at least one head to focus on its original input, but this is very inefficient and does not have the benefit of the improved gradient flow.