Transformers

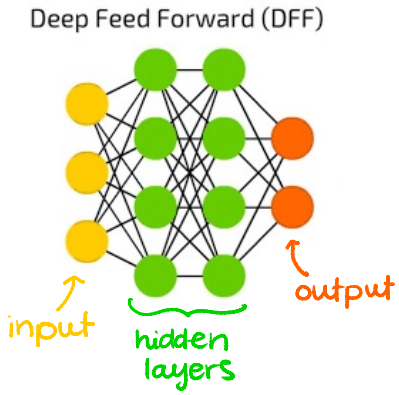
by whom?

- Deep Machine learning model introduced in 2017 used primarily in the field of NLP.
- Designed to handle ordered sequences of data, such as natural language, for various tasks as machine translation and text summarization.

But, wait a minute! Aren't RNN's supposed to do that as well? and they exists since the 80's

Let's see!

1. Feed Forward Neural Networks (quizá no poner esto).



* mostly used in classif ???

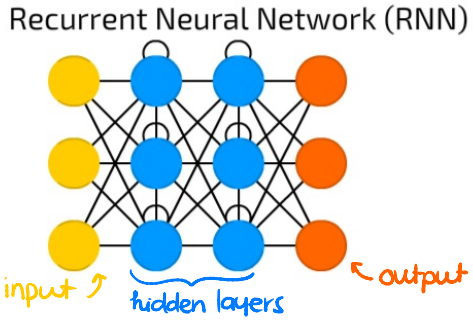
- universal approximator
- input ⇒ hidden layers ⇒ output. Flows in forward direction, no back loops, fully connected
- mapping one vector space into another vector space.
- no notion of order in time, only input it considers is the current example it has been exposed to.
- only remember the formative moments of training.

source: https://towardsdatascience.com/diy-ai-an-old-school-matrix-nn-401a00021a55

source: nn_intro paper saved here on goodnotes.

source: https://skymind.ai/wiki/lstm

2. Recurrent Neural Networks



- Designed to recognize patterns in sequences of data (text, handwriting, spoken words, time series data)
- Temporal dimension: take time and sequence into account
- 2 sources of input: the present and the recent past. The decision at time step t-1 affects the decision at time t.
- Finds correlations between events separated by many moments ("long-term dependencies")

- The process of carrying memory:

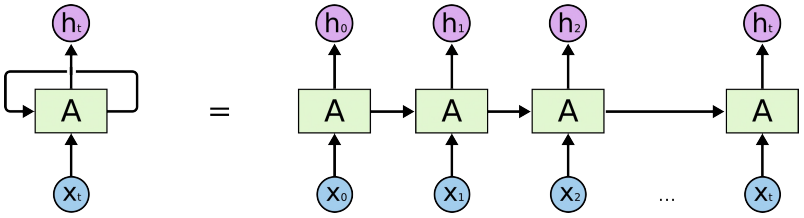
$$h_t = \phi(Wx_t + Uh_{t-1})$$

h_t: hidden state at time t
ϕ: activation function (sigmoid, tanh, etc...)
W: weight matrix
x_t: input at time t
U: weight matrix
h_{t-1}: hidden state at time t-1

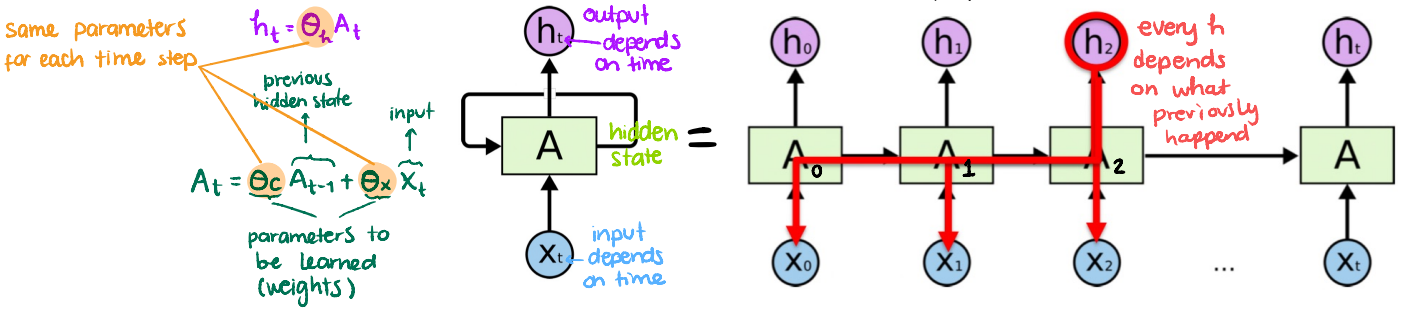
The weight matrices are filters that determine the importance of the present input and the past hidden state

source: https://skymind.ai/wiki/lstm

- RNN's are networks with loops in them, allowing information to persist.



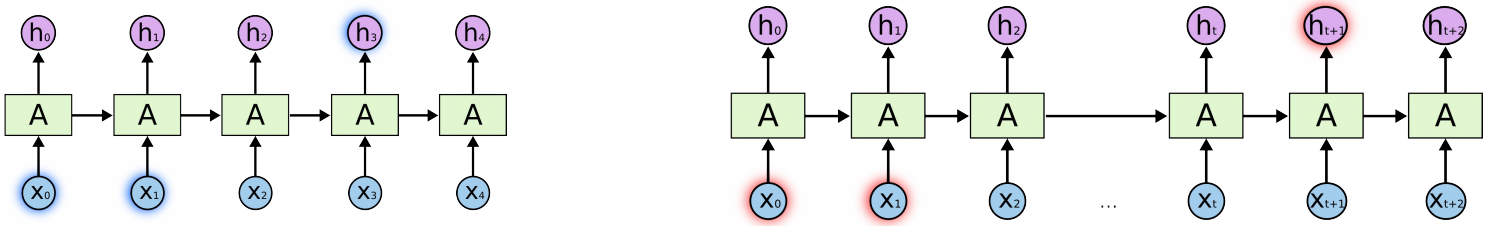
- RNN's can be thought of as multiple copies of the same network, each passing a message to a successor.
- Natural architecture for sequential(?) data.



source: Deep Learning Course TUM

- Variety of problems: speech recognition, language modeling, translation, image captioning

The problem of Long-Term Dependencies:



As the gap grows, RNN's become unable to learn to connect the information

↳ LSTM source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Because the layers and time steps relate to each other through multiplication, derivatives are very susceptible to vanishing or exploding.

To help preserve the error that can be backpropagated through time and layers

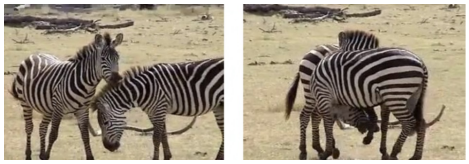
⇒ LSTM's source: https://skymind.ai/wiki/lstm

5. Sequence to Sequence Models

Machine Translation

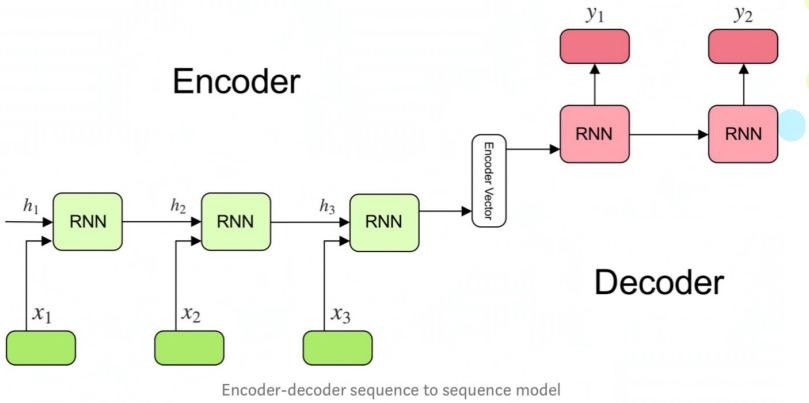
Speech Recognition

Video captioning (descriptions)



- Introduced in 2014 by Google
- Maps sequences of different lengths to each other.

Ex. English to Chinese maps words with symbols.



Encoder-decoder sequence to sequence model

Encoder: A stack of several recurrent units, where each accepts a single element of the input sequence, collects information for that element & propagates it forward.

hidden states: $h_t = f(W^{(hh)} h_{t-1} + W^{(hx)} X_t)$

weights; previous hidden state; word (position t)

Encoder vector:

- Final hidden state produced from the encoder part of the model (calculated using the formula above)
- Encapsulates the information for all input elements.
- Initial hidden state of the decoder part of the model.

Decoder

- A stack of several recurrent units where each predicts an output y_t at a time step t.
- Each recurrent unit accepts a hidden state from the previous unit and produces an output and its own hidden state.

$$h_t = f(W^{(hh)} h_{t-1}) \qquad y_t = \underbrace{\text{softmax}}_{\text{used to create a probability vector}}(W^{S} h_t)$$

- The output sequence is a collection of all words from the answer. Each word is represented as y_i, where i is the order of that word.

https://towardsdatascience.com/understanding-encoder-decoder-sequence-to-sequence-model-679e04af4346

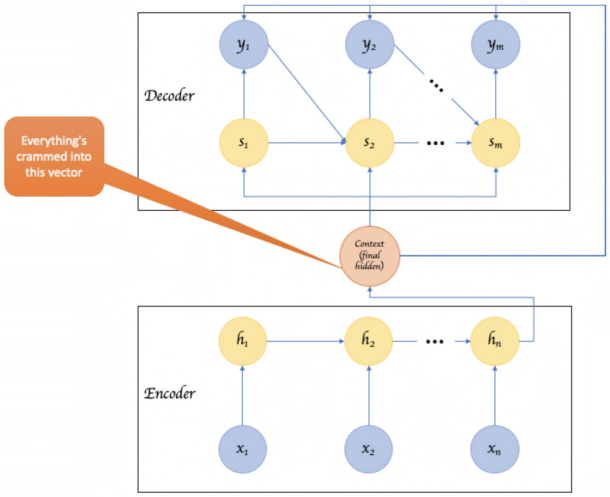


Fig 2: RNNs in Seq to Seq Encoder Decoder model

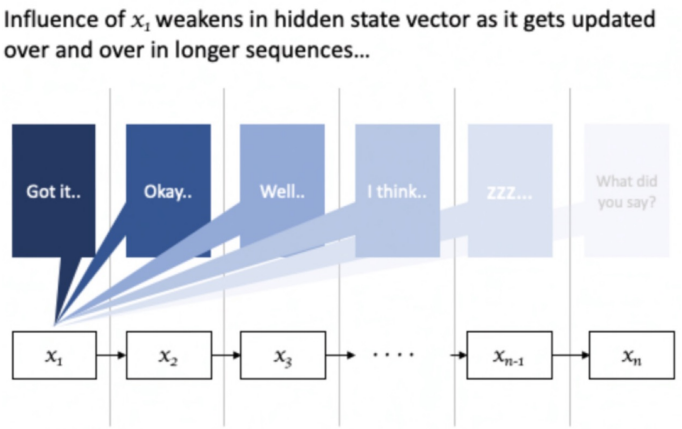


Fig 3: Context becomes weak with longer sentences

Strict sequential order of processing can be a drawback.
The longer the input sequence length, the more difficult it is to capture context.

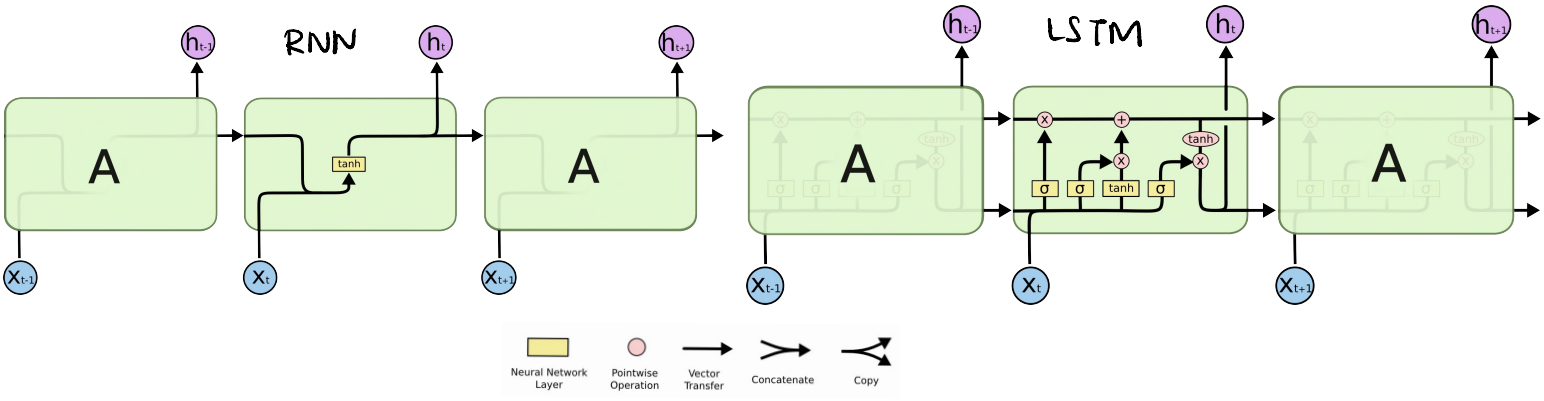
⇒ LSTM's, GRU's provide a way to carry only relevant information. (improves the performance but does not solves the problem)

Source:
→ https://towardsdatascience.com/an-introduction-to-attention-transformers-and-bert-part-1-da0e838c7cda

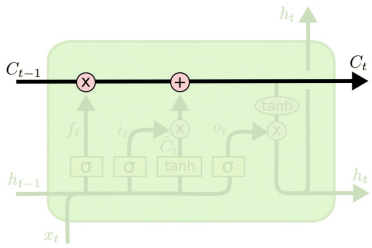
3. Long Short Term Memory ← Skip Detail

- Variation of RNN introduced in the mid-90's by German researchers Sepp Hochreiter & Juergen Schmidhuber as a solution to the vanishing gradient problem
- By maintaining a more constant error, they allow recurrent nets to continue to learn over many time steps (over 1000).
- The cell makes decisions about what to store, and when to allow reads, writes and erasures, via gates that open & close. (remember & forget things that are important & not so important)
- The gates are implemented with element-wise multiplication by sigmoids (range [0,1]), they are differentiable & therefore suitable for backpropagation.
- Instead of determining the subsequent cell state by multiplying its current state with new input, they add the two. This helps to preserve the error during backpropagation. The forget gate still relies on multiplication.

source:
https://skymind.ai/wiki/lstm

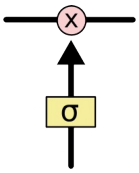


- Capable of learning long-term dependencies
- The repeating module in a LSTM contains 4 interactive layers, instead of 1 (RNN's).



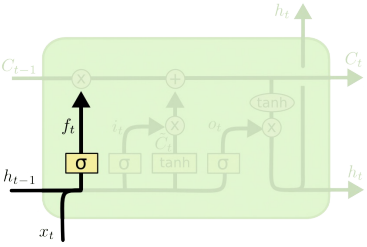
← The key to LSTMs is the cell state (horizontal line running through the top of the diagram). It's very easy for information to just flow along it unchanged.

- LSTM has the ability to remove or add information to the cell state, carefully regulated by gates.
- Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer & a pointwise multiplication operation.



- The sigmoid layer outputs numbers $\in [0,1]$.
  - 0: "let nothing through"
  - 1: "let everything through"

A LSTM has 3 of these gates, to protect & control the cell state.

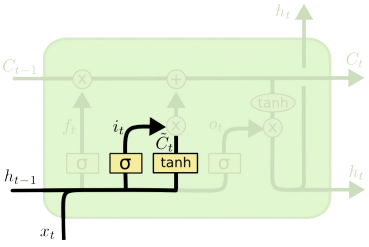


- 1st step (gate): Forget gate.
- Decides what information we're going to throw away from the cell state
  - 1 = "completely keep this"
  - 0 = "completely get rid of this"

previous output, input, bias

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$$

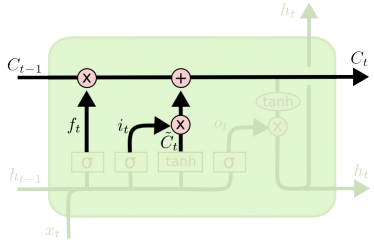
just differ on W & b.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right)$$

$$\tilde{C}_t = \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right)$$

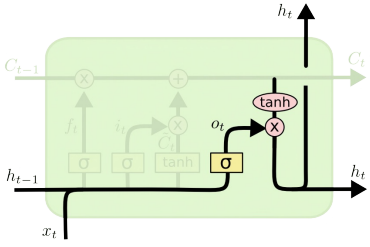
- 2nd step: what new information we're going to store in the cell state?
  - 2.1 : input gate: decides which values we'll update
  - Next, a tanh layer creates a vector of new candidate values, $\tilde{C}t$, that could be added to the state.
  - Then create the updated state C_t



things we want to forget; new candidate values

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

how much we decided to update each state value.

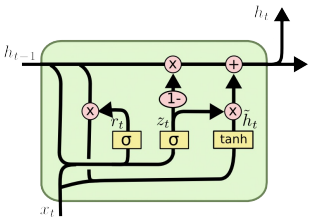


previous output, input

$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

- 3rd step: output gate
  - Run sigmoid layer which decides what parts of the cell state we're going to output.
  - for h_t, we put the cell state through tanh (to push the values to be between -1 & 1) and multiply this by the output of the sigmoid gate. This is to output only the parts we decided to. (for example, it might output whether the subject is singular or plural, so that we know what form of a verb should be conjugated into if that's what follows next)



$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$ → forget gate

$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$ → input gate

$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t])$

$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$ ← update gate

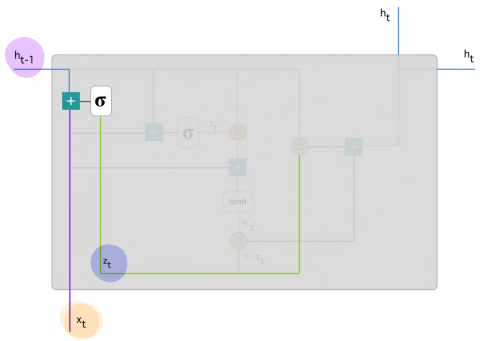
There are a lot of LSTM's variations. The most popular is "Gated Recurrent Unit", GRU.

- combines forget & input gates into a single "update gate".
- merges the cell state & hidden state
- simpler than standard LSTM

source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

4. GRU's (Gated Recurrent Unit) ← Skip Detail

- aims to solve the vanishing gradient problem since the model is not washing out the new input every single time, but keeps the relevant information and passes it down to the next time steps of the network.
- considered as a variant of LSTM's, improved version of RNN's
- uses update / reset gate

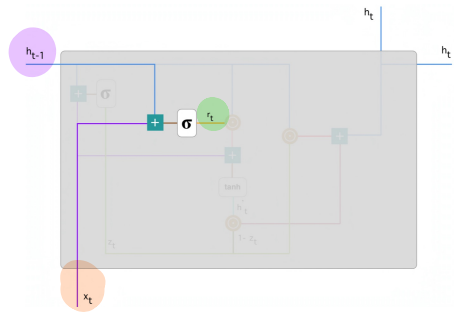


Update gate z_t for time step t

$$z_t = \sigma(W^{(z)} x_t + U^{(z)} h_{t-1})$$

$W^{(z)} x_t$: input; $U^{(z)} h_{t-1}$: info for the previous t-1 units; $W^{(z)}$, $U^{(z)}$: weight matrices

Determines how much of the past information needs to be passed along to the future. The model can decide to copy information from the past.

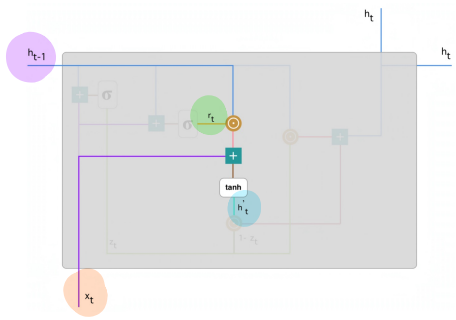


Reset gate

$$r_t = \sigma(W^{(r)} x_t + U^{(r)} h_{t-1})$$

weight matrices; $W^{(r)} x_t$: input; $U^{(r)} h_{t-1}$: info for the previous t-1 units

Determines how much of the past information to forget.
Same as update gate, just the weights are different.



Current memory content

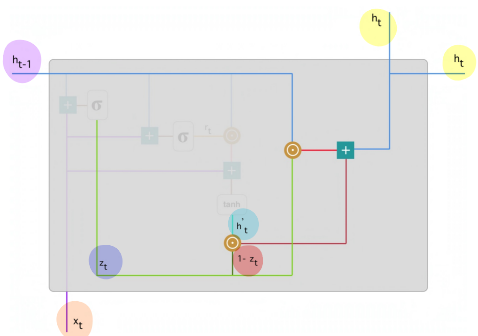
$$h'_t = \tanh(W x_t + [r_t \odot U h_{t-1}])$$

weights; $W x_t$: input; r_t: reset gate; $U h_{t-1}$: info for the previous t-1 units

- uses reset gate to store the relevant information from the past.
- Hadamard product will determine what to remove from the previous time steps.

For example, if we have a sentiment analysis problem for determining one's opinion about a book from a review he wrote. The text starts with "This is a fantasy book which illustrates..." and after a couple of paragraphs ends with "I didn't quite enjoy the book because I think it captures too many details". To determine the overall level of satisfaction from the book we only need the last part of the review.

In that case, as the neural network approaches to the end of the text it will learn to assign $r_t \approx 0$, washing out the past and focusing only on the last sentences.



Final memory at current time step

$$h_t = (z_t \odot h_{t-1}) + [(1 - z_t) \odot h'_t]$$

h_{t-1}: info from previous units; z_t: update gate; h'_t: current memory content

h_t holds information for the current unit and passes it down to the network.

h_t uses the update gate z_t to determine what to collect from the current memory content (h'_t) and what from the previous steps (h_{t-1}).

Let's bring up the example about the book review. This time, the most relevant information is positioned in the beginning of the text. The model can learn to set the vector $z_t \approx 1$ and keep a majority of the previous information. Since $z_t \approx 1 \Rightarrow 1 - z_t \approx 0$, which will ignore big portion of the current content which is irrelevant for our prediction.

source: https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be

The problem with LSTM's: (and GRU's)

1. Sequential computation inhibits parallelization. (since you have to process word by word)
2. No explicit modeling of long and short range dependencies
3. Distance between positions is linear. (the probability of keeping the context from a word that is far away from the current word being processed decreases exponentially with the distance from it).

same problem as before. LSTM's just improves the performance.

How can we solve this? Attention !!

source: https://towardsdatascience.com/transformers-141e32e69591

4. Attention

idea: build an architecture that consumes all hidden states and we don't have to deal with weakening context.

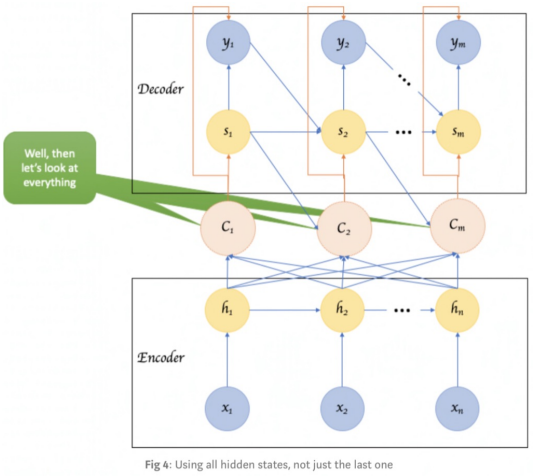


Fig 4: Using all hidden states, not just the last one

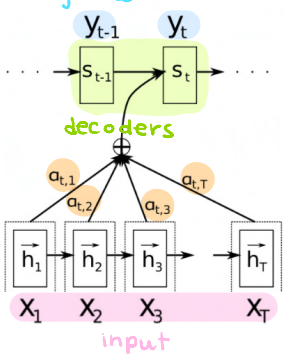
what to retrieve from memory

Don't encode the full sentence into a vector, instead allow the decoder to "attend" to different parts of the source sentence at each step of the output generation.

Model learns what to attend to based on: input, what it has produced so far.

- Languages pretty well aligned (like English & German), the decoder will attend things in sequence.

translated words produced by the decoder



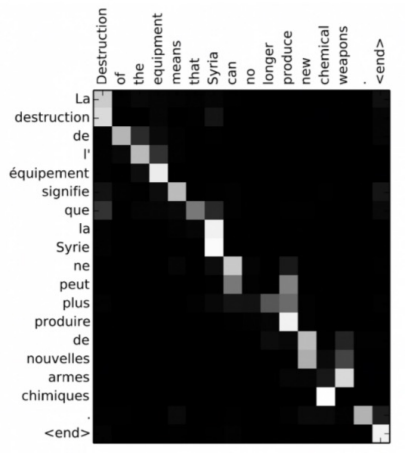
- outputs (y_i's) depend on a weighted combination of all the input states
- the a's are weights that define in how much of each input state should be considered for each output.

$a_{i,j}$: weight (influence) of input j into output i.

the larger the weight, the stronger the attention

a's typically normalized to sum to 1.

Attention gives us the ability to interpret and visualize what the model is doing.



Attention is counterintuitive.

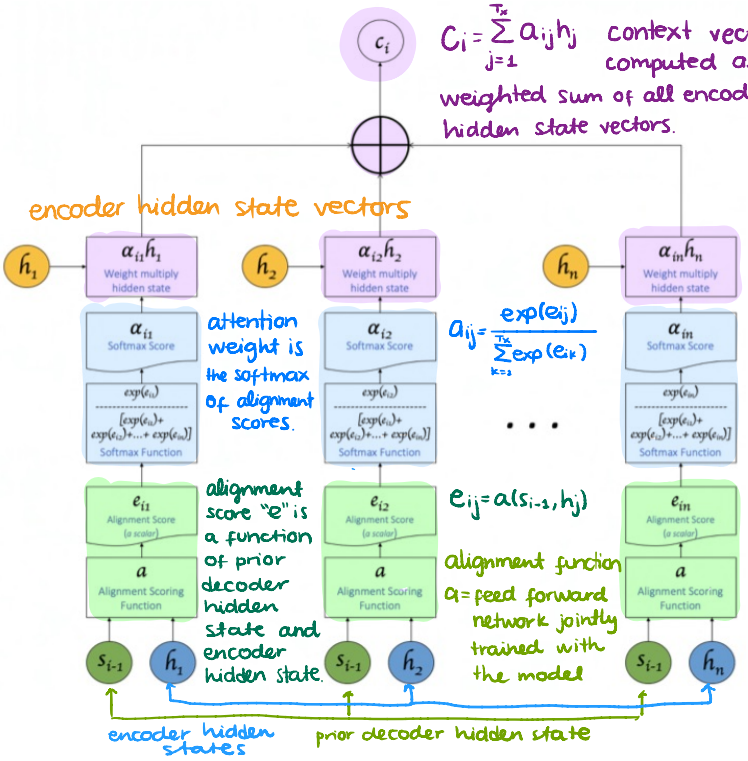
it's not focusing on one thing, it looks at everything in detail before deciding what to focus on.

We need to calculate an attention value for each combination of input and output. (can become expensive).

it performs well on many tasks.

Attention is easy to train using backpropagation.

http://www.wildml.com/2016/01/attention-and-memory-in-deep-learning-and-nlp/



- The distinct context vector for an output step is a sum product of attention weights and all input hidden states.
- The attention weights for every single output will be different
  ⇒ the sum of the weighted hidden vectors is distinct for each output step.

Pros:
- addresses the problem of having one single context vector.

Cons:
- the model is really big (a lot of computations are involved. (the number of samples in a batch is limited).
- lack of parallelization: given the sequential nature of the operations, if the input sequence is of length "n", it requires "n" sequential operations to arive at the final hidden state (i.e. calculate $h_1, h_2, \dots, h_n$). we can't perform these operations in parallel as h_1 is a prerequisite to calculate h_2.

Attention

- learned weighting per input element x_j: $y_i = \left(\frac{1}{C(x)}\right) \sum_{j=1}^{M} a(x_i, x_j) f(x_j)$ (normalization factor; attention weight)
- attention a can be modelled in several ways, e.g.:
  1. Dot product: $a(x_i, x_j) = \theta(x_i)^T \varphi(x_j)$, with some functions θ and φ, and $C(x) = M$.
  2. query-key-value: Query $q(x_i) = W_q x_i$, key $k(x_j) = W_k x_j$, value $f(x_j)$. With $a(x_i, x_j) = e^{q(x_i)^T k(x_j)}$ and $C(x) = \sum_j a(x_i, x_j)$. This becomes equivalent to a Softmax: $y_i = \sum_j \text{softmax}(q(x_i)^T k(x_j)) f(x_j)$
- attention allows to use information from larger distances and weight parts of the input differently.
- self-attention denotes attention across all inputs, including the element x_i itself (i and j iterate over the same elements).

source: TUM Machine Learning Course

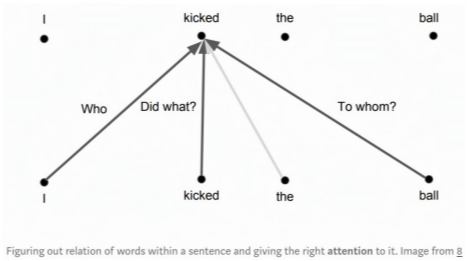
* Attention improved RNN's, but didn't solve the problem of parallelization.

From here: Transformers

Transformer is the first transduction model relying entirely on self-attention to compute representations of its input & output without using sequence-aligned RNNs or convolution.
Self-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence.

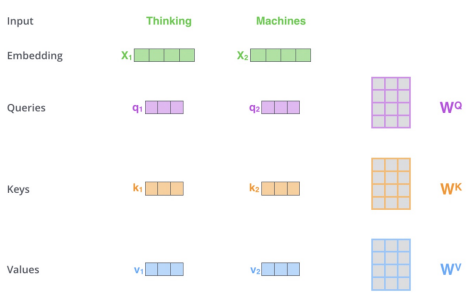
https://arxiv.org/pdf/1706.03762.pdf

5. Self-Attention



Figuring out relation of words within a sentence and giving the right attention to it. Image from 8

1. Create 3 vectors from each of the encoder's input vectors.
   Query vector, Key vector & Value vector.
   These vectors are created by multiplying the embedding by three matrices that we trained during the training process.
   (first randomly initialized, then learned during training)

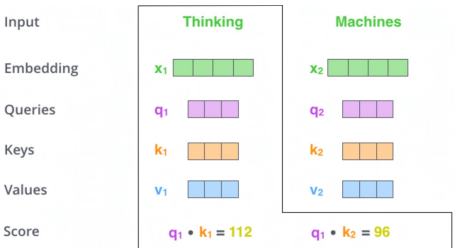


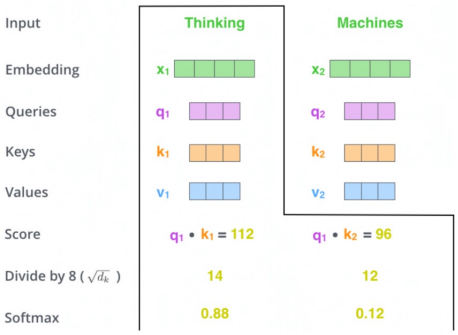
2. Calculate a score.

The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

Self-attention for the word in position #1: score 1 = $q_1 \cdot k_1$, score 2 = $q_1 \cdot k_2$.

Self-attention for the word in position #2: score 1 = $q_2 \cdot k_1$, score 2 = $q_2 \cdot k_2$.





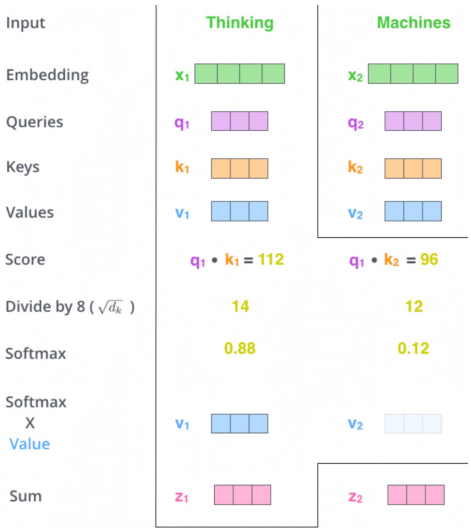
3. Divide the scores by the square root of the dimension of the key vectors to have more stable gradients.

4. Pass the result through a softmax operation. The softmax score determines how much each word will be expressed at this position. Clearly the word at this position will have the highest softmax score, but sometimes it's useful to attend to another word that is relevant to the current word.

5. Multiply each value vector by the softmax score.

The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words.

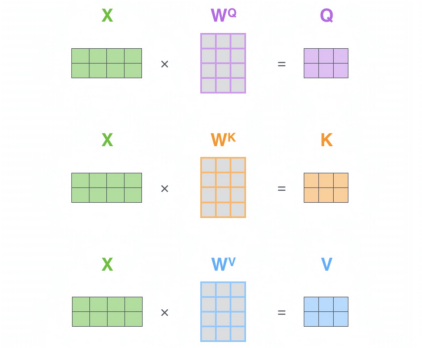
6. Sum up the weighted value vectors. This produces the output of the self-attention layer at this position.



The resulting vector is sent through a feed forward neural network (when using the transformer).

https://towardsdatascience.com/transformers-141e32e69591

Matrix Calculation of Self-Attention



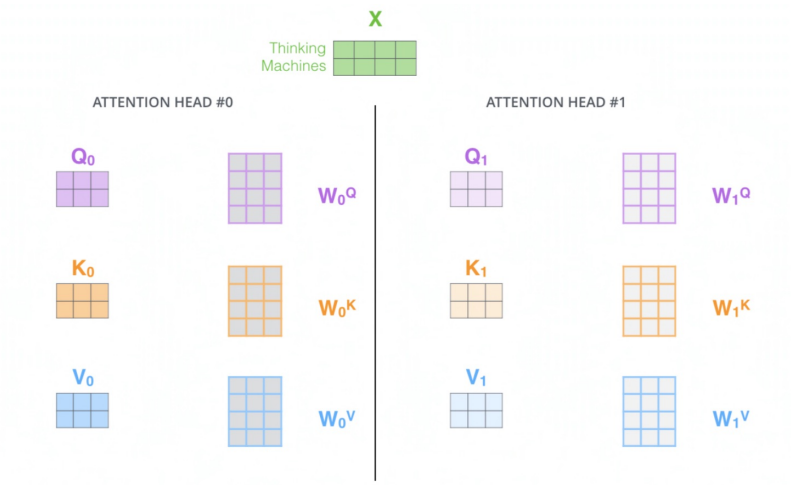
Every row in the X matrix corresponds to a word in the input sentence. We again see the difference in size of the embedding vector (512, or 4 boxes in the figure), and the q/k/v vectors (64, or 3 boxes in the figure)

← First, pack the embeddings into a matrix X and multiply it by the three matrices that we trained during the training process to obtain the Query, Key & Value matrices.

Steps 2-6 can be summarized by this formula:

$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) V = Z$$

6. Multihead Attention



With multi-headed attention, we maintain separate Q/K/V weight matrices for each head resulting in different Q/K/V matrices. As we did before, we multiply X by the WQ/WK/WV matrices to produce Q/K/V matrices.

Improves attention layer (in transformer) in 2 ways:

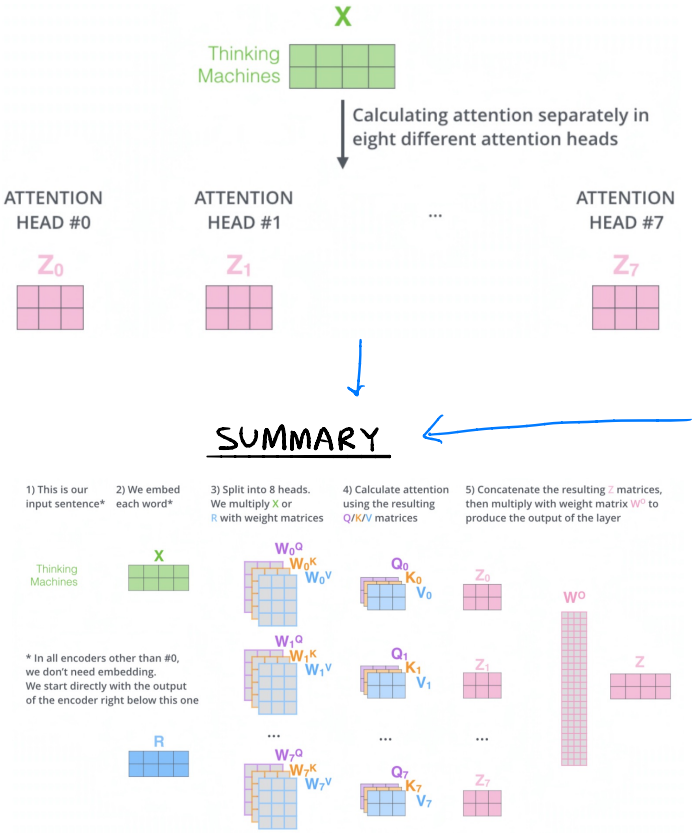
1. Expands the model's ability to focus on different positions. → self-attention example

In the example above, z_1 contains a little bit of every other encoding, but it could be dominated by the actual word itself. It would be useful if we're translating a sentence like "The animal didn't cross the street because it was too tired", we would want to know which word "it" refers to.

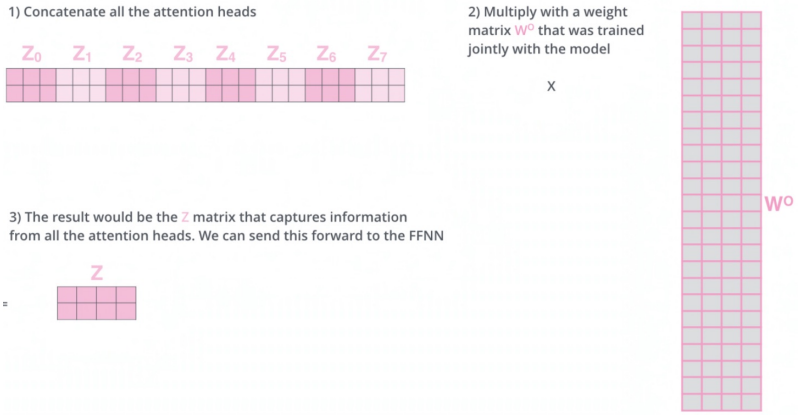
2. It gives the attention layer multiple "representation subspaces".

We'll have multiple sets of Query/key/Value weight matrices (Transformer has 8). Each

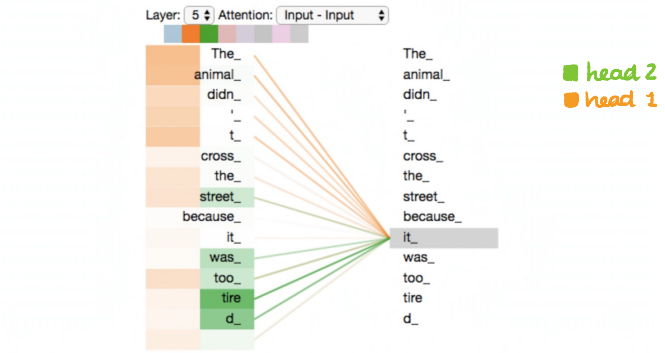
of these sets is randomly initialized. Then, after training, each set is used to project the input embeddings into a different representation subspace.



if we do the same self-attention calculation we outlined above, 8 different times with different weight matrices, we end up with 8 different Z matrices, but the feed forward neural network just need one, then (for the transformer)

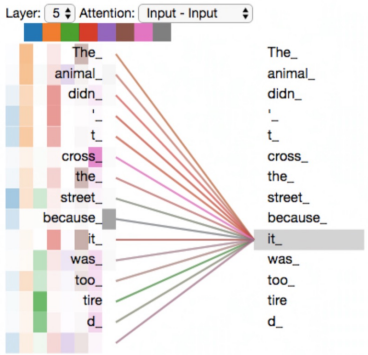


Example:

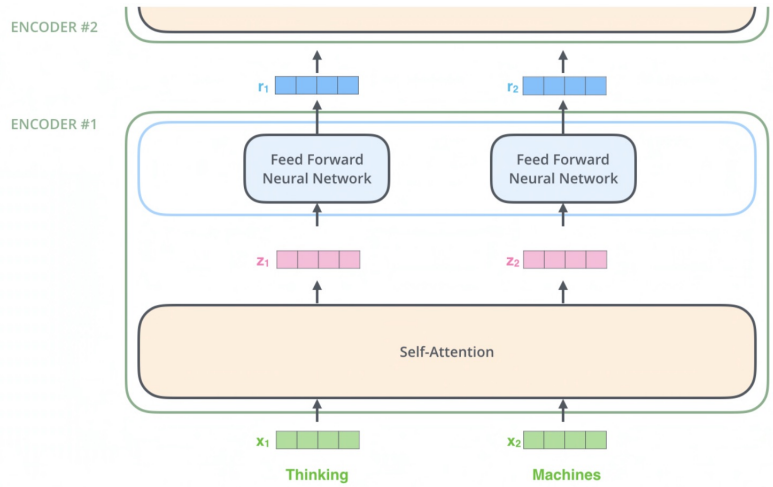


As we encode the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired" -- in a sense, the model's representation of the word "it" bakes in some of the representation of both "animal" and "tired".

If we add all the attention heads to the picture, however, things can be harder to interpret:



In the transformer:



The word at each position passes through a self-attention process. Then, they each pass through a feed-forward neural network -- the exact same network with each vector flowing through it separately.

http://jalammar.github.io/illustrated-transformer/

Why Self-attention?

computations that can be parallelized

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

self-attention layers are faster than recurrent layers when the sequence length n is smaller than the representation dimensionality d, which is most often the case with sentence representation.

Learning long-range dependencies is a key challenge in many sequence transduction tasks. One key factor affecting the ability to learn such dependencies is the length of the paths forward and backward signals have to traverse in the network. The shorter these paths between any combination of positions in the input and output sequences, the easier is to learn long-range dependencies.

Transformer Architecture

https://arxiv.org/pdf/1706.03762.pdf

http://jalammar.github.io/illustrated-transformer/

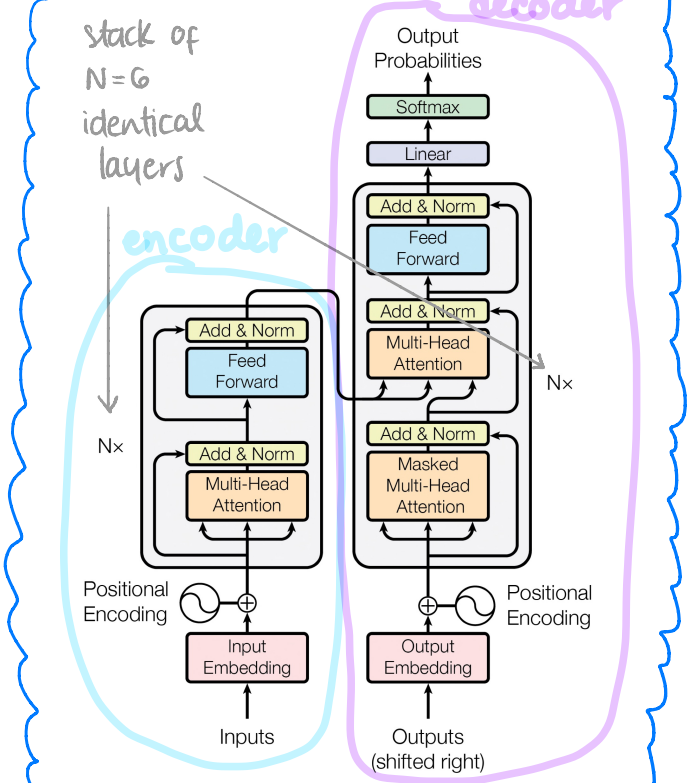
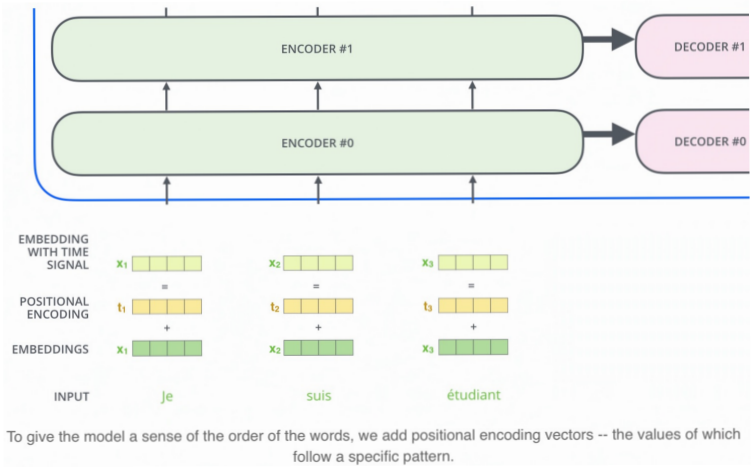


Figure 1: The Transformer - model architecture.

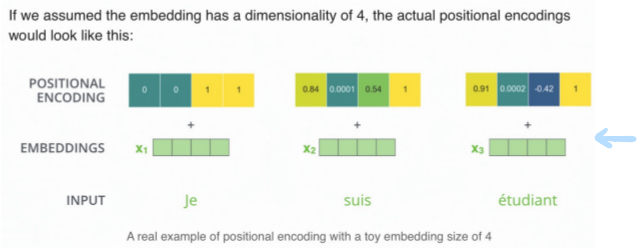
7. Positional Encoding

The transformer adds a vector to each input embedding as a way to account for the order of the words in the input sequence.

These vectors follow a specific pattern that the model learns, which helps it determine the position of each word, or the distances between different words in the sequence. Adding these values to the embeddings provides meaningful distances between the embedding vectors once they're projected into Q/K/V vectors and during dot-product attention.



To give the model a sense of the order of the words, we add positional encoding vectors -- the values of which follow a specific pattern.



A real example of positional encoding with a toy embedding size of 4

positional encodings have the same dimension d_{model} as the embeddings, so that the two can be summed.

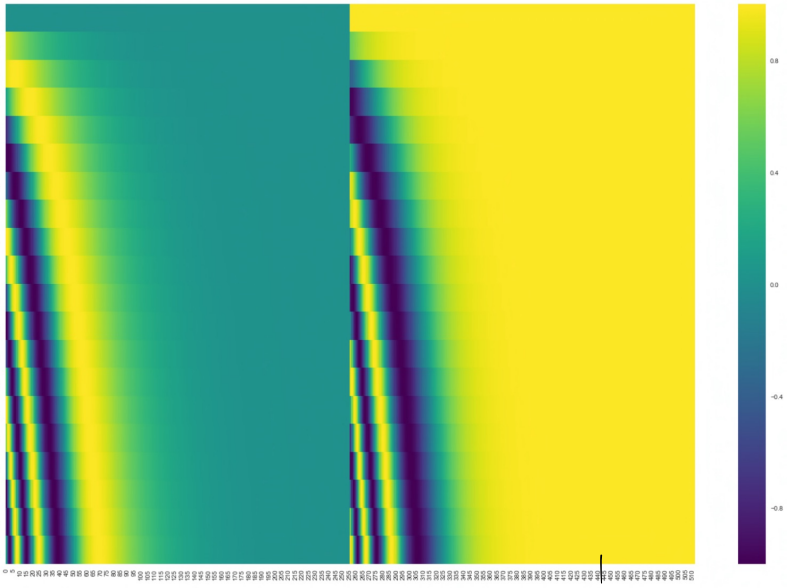
https://arxiv.org/pdf/1706.03762.pdf

In this work, we use sine and cosine functions of different frequencies:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

where pos is the position and i is the dimension. That is, each dimension of the positional encoding corresponds to a sinusoid. The wavelengths form a geometric progression from 2π to $10000 \cdot 2\pi$. We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.

We also experimented with using learned positional embeddings [9] instead, and found that the two versions produced nearly identical results (see Table 3 row (E)). We chose the sinusoidal version because it may allow the model to extrapolate to sequence lengths longer than the ones encountered during training.

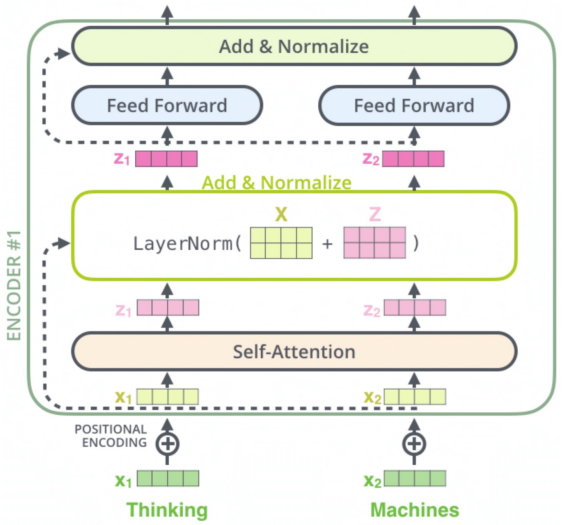


A real example of positional encoding for 20 words (rows) with an embedding size of 512 (columns). You can see that it appears split in half down the center. That's because the values of the left half are generated by one function (which uses sine), and the right half is generated by another function (which uses cosine). They're then concatenated to form each of the positional encoding vectors.

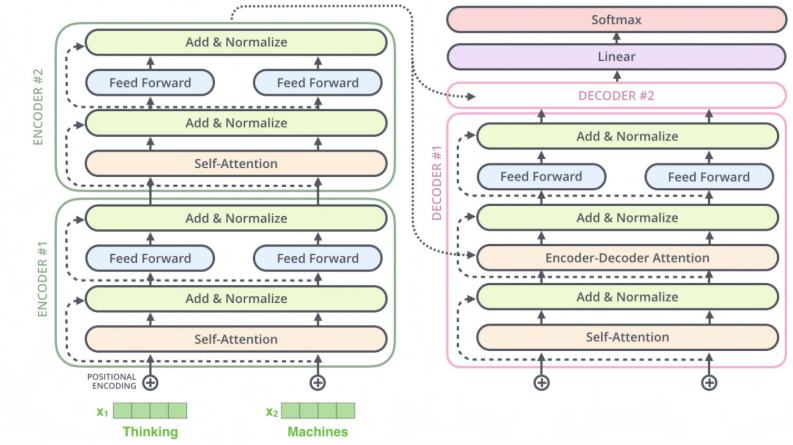
In the figure, each row corresponds to a positional encoding of a vector. So, the first row would be the vector we'd add to the embedding of the first word in an input sequence.

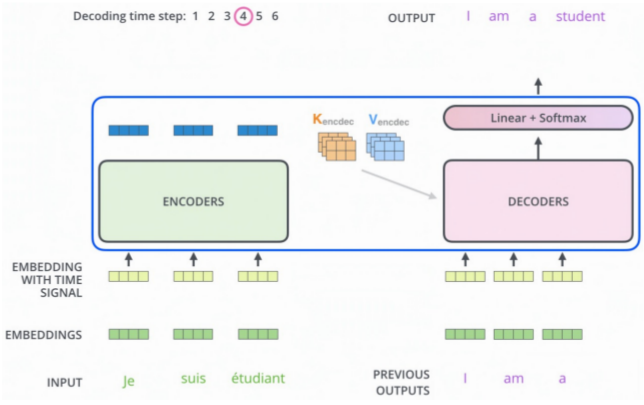
Each row contains 512 values - each with a value between 1 & -1. It's color-coded so the pattern is visible.

8. Encoder / Decoder Architecture



This goes for the sub-layers of the decoder as well. If we're to think of a Transformer of 2 stacked encoders and decoders, it would look something like this:





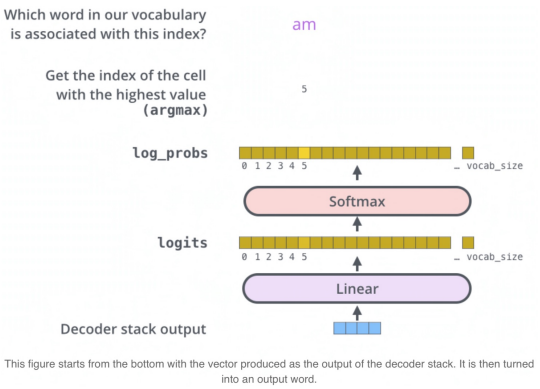
- The encoder starts by processing the input sequence.
- The output of the top encoder is then transformed into a set of attention vectors k & V. These are to be used by each decoder in its "encoder-decoder attention" layer which helps the decoder focus on appropriate places in the input sequence.
- The following steps repeat the process until a special symbol is reached indicating the transformer decoder has completed its output:

* The output of each step is fed to the bottom decoder in the next time step, and the decoders bubble up their decoding results just like the encoders did. And just like we did with the encoder inputs, we embed and add positional encoding to those decoder inputs to indicate the position of each word.

The self-attention layers in the decoder operate in a slightly different way than the one in the encoder: In the decoder, the self-attention layer is only allowed to attend the earlier positions in the output sequence. This is done by masking future positions (setting them to -inf) before the softmax step in the self-attention calculation.

The "Encoder-Decoder Attention" layer works just like multiheaded self-attention, except it creates its Queries matrix from the layer below it, and takes the Keys and Values matrix from the output of the encoder stack.

9. Final Linear & Softmax Layer



This figure starts from the bottom with the vector produced as the output of the decoder stack. It is then turned into an output word.

The Linear & Softmax Layer turn the vector of floats, that is given as an output of the decoder stack, into a word.

- Linear layer: is a simple fully connected neural network that projects the vector produced by the stack of decoders, into a much much larger vector called "logits vector".
  Let's assume that our model knows 10,000 unique English words (our model's "output vocabulary") that it's learned from its training dataset. This would make the logits vector 10,000 cells wide - each cell corresponding to the score of a unique word. That is how we interpret the output of the model followed by the Linear layer.
- Softmax layer: turns the scores of the "logits vector" into probabilities (all positive, all add up to 1). The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.

10. Intuition - Training the Transformer.

During training, an untrained model would go through the exact same forward pass. But since we are training it on a labeled training dataset, we can compare its output with the actual correct output.
To visualize this, let's assume our output vocabulary only contains six words ("a", "am", "I", "thanks", "student", and "<eos>" (short for "end of sentence").

Output Vocabulary

WORD	a	am	I	thanks	student	<eos>
INDEX	0	1	2	3	4	5

One-hot encoding of the word "am"

Example: one-hot encoding of our output vocabulary

Once we define our output vocabulary, we can use a vector of the same width to indicate each word in our vocabulary. This is also known as one-hot encoding. So for example, we can indicate the word "am" using the vector:

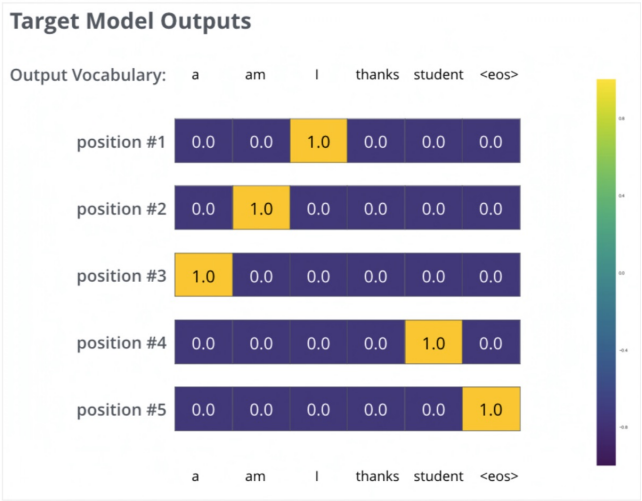
1. Loss Function.

What this really means, is that we want our model to successively output probability distributions where:

- Each probability distribution is represented by a vector of width vocab_size (6 in our toy example, but more realistically a number like 3,000 or 10,000)
- The first probability distribution has the highest probability at the cell associated with the word "i"
- The second probability distribution has the highest probability at the cell associated with the word "am"
- And so on, until the fifth output distribution indicates '<end of sentence>' symbol, which also has a cell associated with it from the 10,000 element vocabulary.

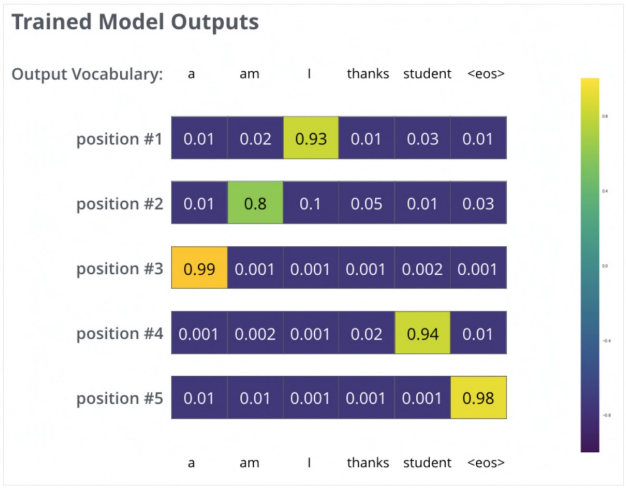
Cross Entropy

Kullback-Leibler divergence



The targeted probability distributions we'll train our model against in the training example for one sample sentence.

After training the model for enough time on a large enough dataset, we would hope the produced probability distributions would look like this:



Hopefully upon training, the model would output the right translation we expect. Of course it's no real indication if this phrase was part of the training dataset (see: cross validation). Notice that every position gets a little bit of probability even if it's unlikely to be the output of that time step -- that's a very useful property of softmax which helps the training process.

Now, because the model produces the outputs one at a time, we can assume that the model is selecting the word with the highest probability from that probability distribution and throwing away the rest. That's one way to do it (called greedy decoding). Another way to do it would be to hold on to, say, the top two words (say, 'I' and 'a' for example), then in the next step, run the model twice: once assuming the first output position was the word 'I', and another time assuming the first output position was the word 'me', and whichever version produced less error considering both positions #1 and #2 is kept. We repeat this for positions #2 and #3… etc. This method is called "beam search", where in our example, beam_size was two (because we compared the results after calculating the beams for positions #1 and #2), and top_beams is also two (since we kept two words). These are both hyperparameters that you can experiment with.

http://jalammar.github.io/illustrated-transformer/

5.1 Training Data and Batching

We trained on the standard WMT 2014 English-German dataset consisting of about 4.5 million sentence pairs. Sentences were encoded using byte-pair encoding [3], which has a shared source-target vocabulary of about 37000 tokens. For English-French, we used the significantly larger WMT 2014 English-French dataset consisting of 36M sentences and split tokens into a 32000 word-piece vocabulary [38]. Sentence pairs were batched together by approximate sequence length. Each training batch contained a set of sentence pairs containing approximately 25000 source tokens and 25000 target tokens.

5.2 Hardware and Schedule

We trained our models on one machine with 8 NVIDIA P100 GPUs. For our base models using the hyperparameters described throughout the paper, each training step took about 0.4 seconds. We trained the base models for a total of 100,000 steps or 12 hours. For our big models,(described on the bottom line of table 3), step time was 1.0 seconds. The big models were trained for 300,000 steps (3.5 days).

https://arxiv.org/pdf/1706.03762.pdf

5.3 Optimizer

We used the Adam optimizer [20] with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$. We varied the learning rate over the course of training, according to the formula:

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5}) \quad (3)$$

This corresponds to increasing the learning rate linearly for the first $warmup\_steps$ training steps, and decreasing it thereafter proportionally to the inverse square root of the step number. We used $warmup\_steps = 4000$.

5.4 Regularization

We employ three types of regularization during training:

Residual Dropout We apply dropout [33] to the output of each sub-layer, before it is added to the sub-layer input and normalized. In addition, we apply dropout to the sums of the embeddings and the positional encodings in both the encoder and decoder stacks. For the base model, we use a rate of $P_{drop} = 0.1$.

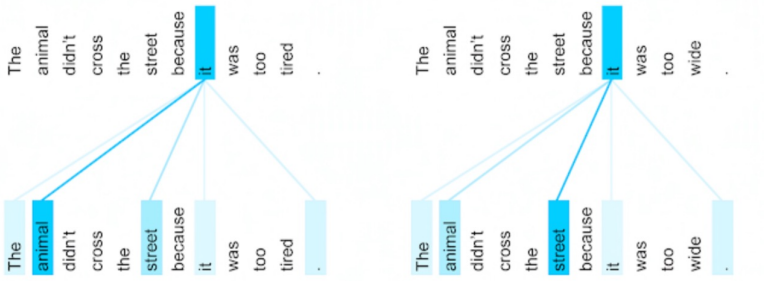
Label Smoothing During training, we employed label smoothing of value $\epsilon_{ls} = 0.1$ [36]. This hurts perplexity, as the model learns to be more unsure, but improves accuracy and BLEU score.

11. Transformer Performance

The animal didn't cross the street because it was too tired.
L'animal n'a pas traversé la rue parce qu'il était trop fatigué.

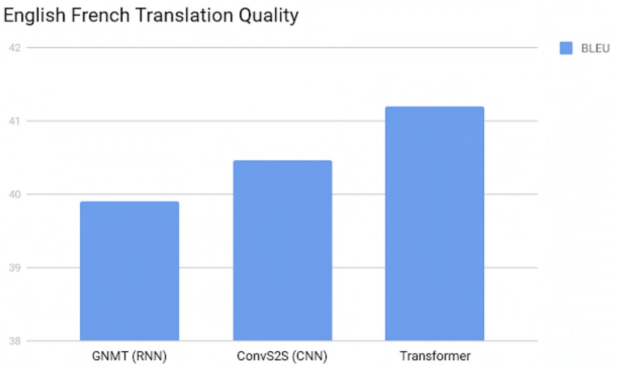
The animal didn't cross the street because it was too wide.
L'animal n'a pas traversé la rue parce qu'elle était trop large.

It is obvious to most that in the first sentence pair "it" refers to the animal, and in the second to the street. When translating these sentences to French or German, the translation for "it" depends on the gender of the noun it refers to - and in French "animal" and "street" have different genders. In contrast to the current Google Translate model, the Transformer translates both of these sentences to French correctly. Visualizing what words the encoder attended to when computing the final representation for the word "it" sheds some light on how the network made the decision. In one of its steps, the Transformer clearly identified the two nouns "it" could refer to and the respective amount of attention reflects its choice in the different contexts.



The encoder self-attention distribution for the word "it" from the 5th to the 6th layer of a Transformer trained on English to French translation (one of eight attention heads).

Given this insight, it might not be that surprising that the Transformer also performs very well on the classic language analysis task of syntactic constituency parsing, a task the natural language processing community has attacked with highly specialized systems for decades.
In fact, with little adaptation, the same network we used for English to German translation outperformed all but one of the previously proposed approaches to constituency parsing.



BLEU scores (higher is better) of single models on the standard WMT newstest2014 English to French translation benchmark.

https://machinelearningmastery.com/calculate-bleu-score-for-text-python/

Bilingual Evaluation Understudy Score

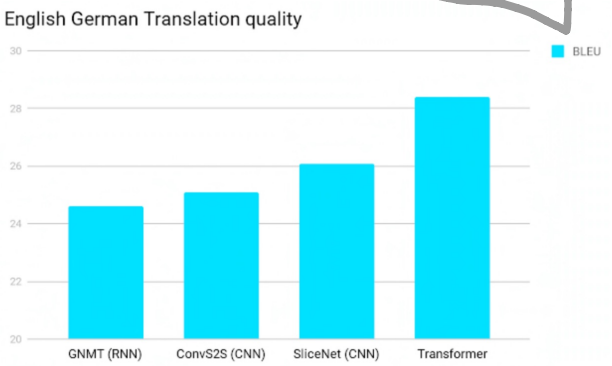
The Bilingual Evaluation Understudy Score, or BLEU for short, is a metric for evaluating a generated sentence to a reference sentence.

A perfect match results in a score of 1.0, whereas a perfect mismatch results in a score of 0.0.

The score was developed for evaluating the predictions made by automatic machine translation systems. It is not perfect, but does offer 5 compelling benefits:

- It is quick and inexpensive to calculate.
- It is easy to understand.
- It is language independent.
- It correlates highly with human evaluation.
- It has been widely adopted.

The BLEU score was proposed by Kishore Papineni, et al. in their 2002 paper "BLEU: a Method for Automatic Evaluation of Machine Translation".



BLEU scores (higher is better) of single models on the standard WMT newstest2014 English to German translation benchmark.

https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html

6.1 Machine Translation Results

On the WMT 2014 English-to-German translation task, the big transformer model (Transformer (big) in Table 2) outperforms the best previously reported models (including ensembles) by more than 2.0 BLEU, establishing a new state-of-the-art BLEU score of 28.4. The configuration of this model is listed in the bottom line of Table 3. Training took 3.5 days on 8 P100 GPUs. Even our base model surpasses all previously published models and ensembles, at a fraction of the training cost of any of the competitive models.

On the WMT 2014 English-to-French translation task, our big model achieves a BLEU score of 41.0, outperforming all of the previously published single models, at less than $1/4$ the training cost of the previous state-of-the-art model. The Transformer (big) model trained for English-to-French used dropout rate $P_{drop} = 0.1$, instead of 0.3.

For the base models, we used a single model obtained by averaging the last 5 checkpoints, which were written at 10-minute intervals. For the big models, we averaged the last 20 checkpoints. We used beam search with a beam size of 4 and length penalty $\alpha = 0.6$ [38]. These hyperparameters were chosen after experimentation on the development set. We set the maximum output length during inference to input length + 50, but terminate early when possible [38].

Table 2 summarizes our results and compares our translation quality and training costs to other model architectures from the literature. We estimate the number of floating point operations used to train a model by multiplying the training time, the number of GPUs used, and an estimate of the sustained single-precision floating-point capacity of each GPU [5].

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

https://arxiv.org/pdf/1706.03762.pdf