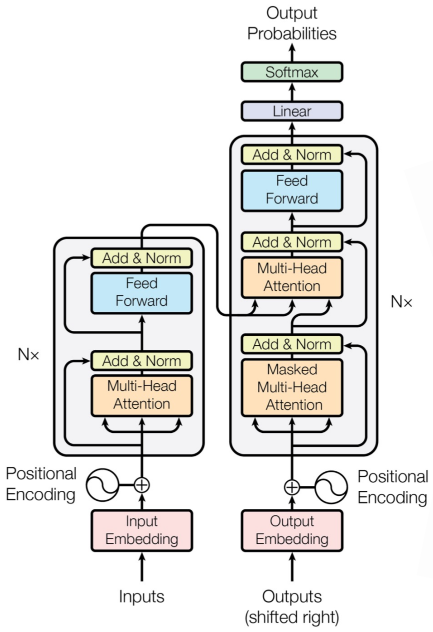
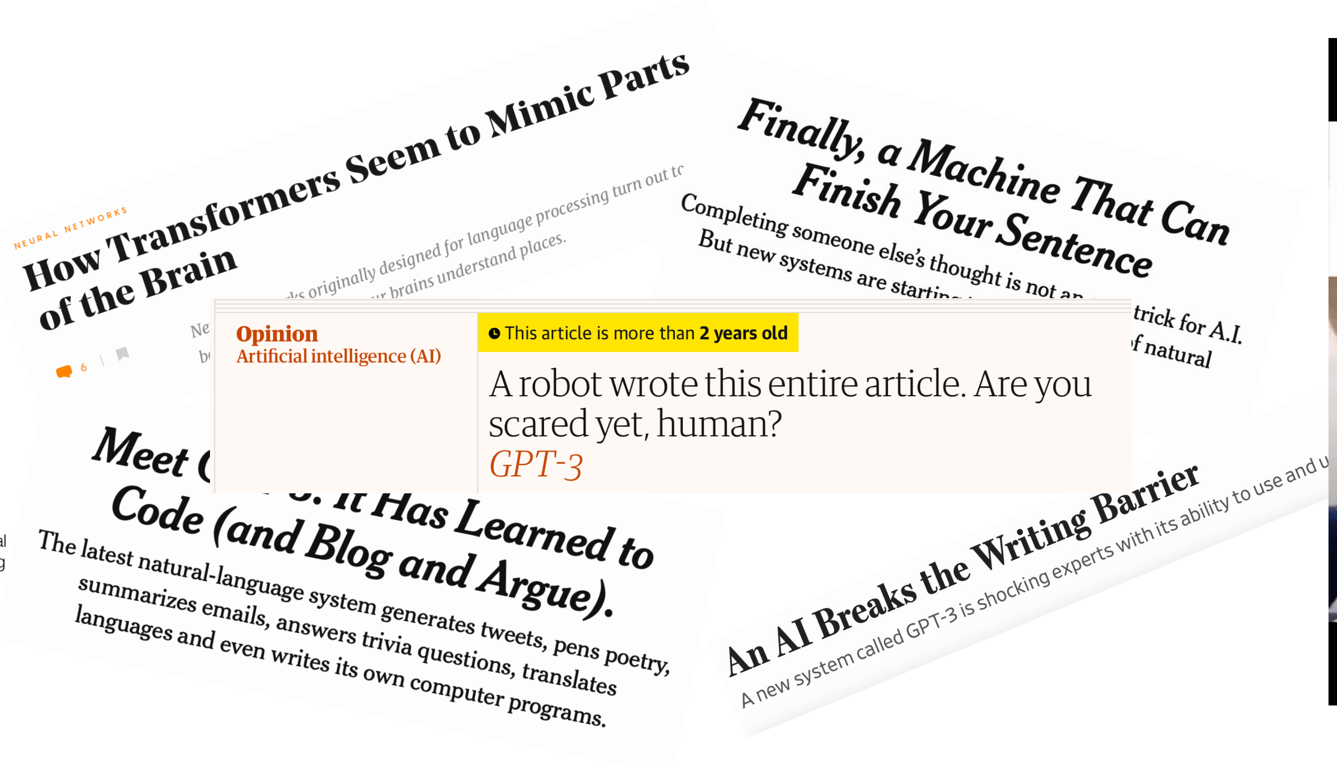
Vaswani et al. (2017) still being state of the art

obligatory Michael Bay Transformer

Dr. Vaswani or: How I learned to stop using LSTMs and Love Attention

Vaswani et al. 2017



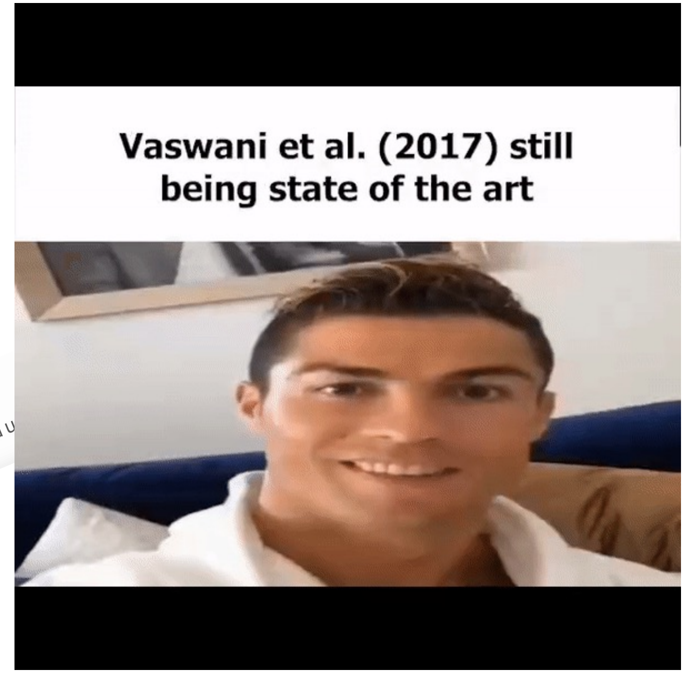

NEURAL NETWORKS
How Transformers Seem to Mimic Parts of the Brain

Finally, a Machine That Can Finish Your Sentence
Completing someone else's thought is not an
But new systems are startin
trick for A.I.
f natural

Opinion
Artificial intelligence (AI)
This article is more than **2 years old**
A robot wrote this entire article. Are you scared yet, human?
GPT-3

Meet (
It Has Learned to Code (and Blog and Argue).
The latest natural-language system generates tweets, pens poetry, summarizes emails, answers trivia questions, translates languages and even writes its own computer programs.

An AI Breaks the Writing Barrier
A new system called GPT-3 is shocking experts with its ability to use and u



UBC Machine Learning Reading Group – Fall 2022
Alan Milligan
alanmil@student.ubc.ca

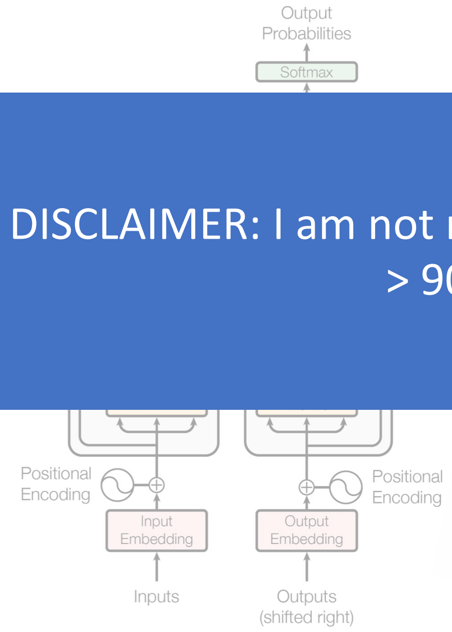
me after making slides!

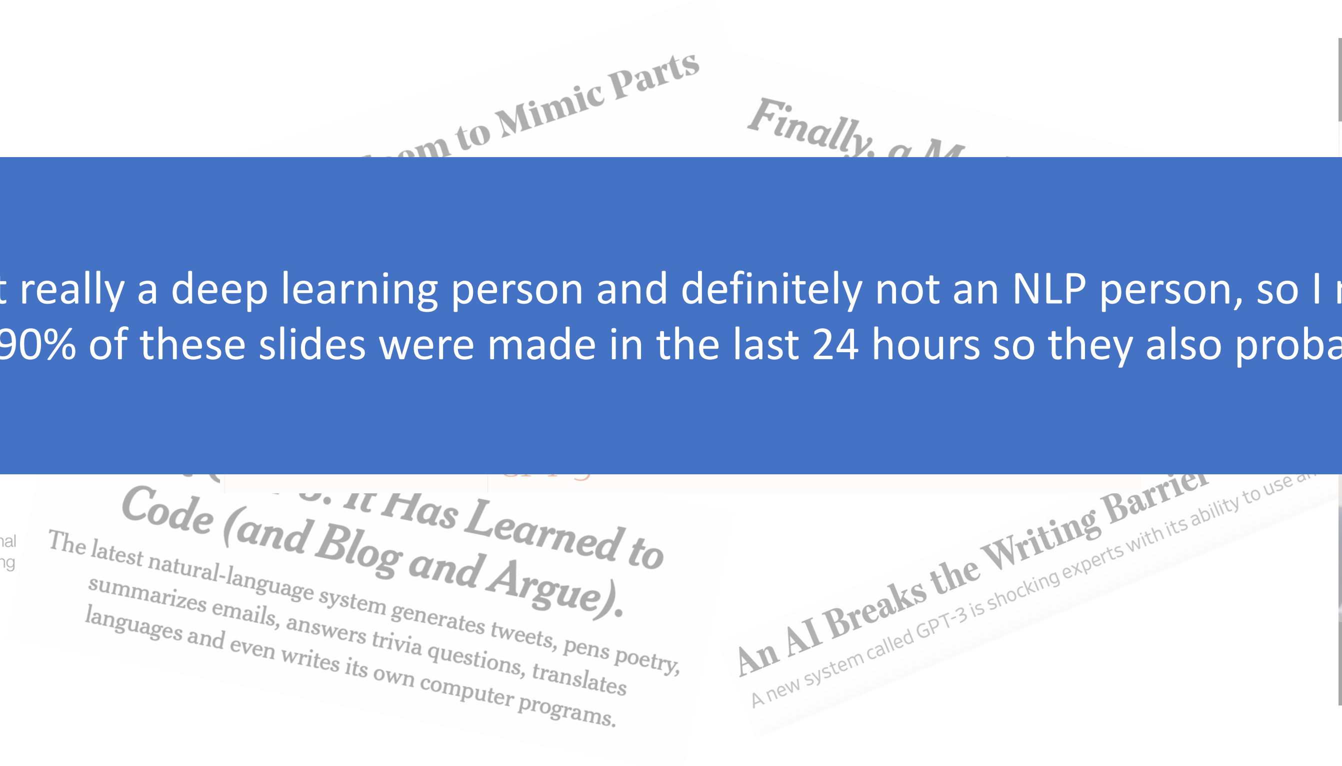
Not this Vaswani

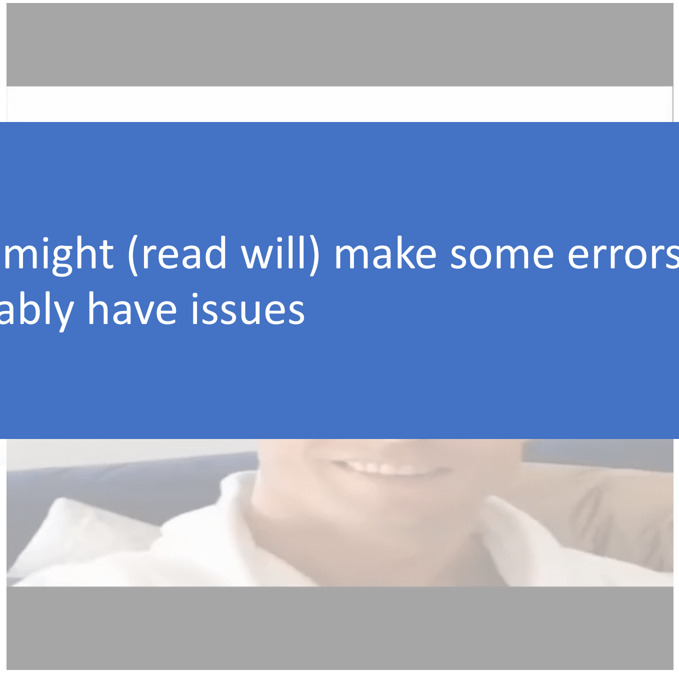
obligatory Michael Bay Transformer

Dr. Vaswani or: How I learned to stop using LSTMs and Love Attention

Vaswani et al. 2017







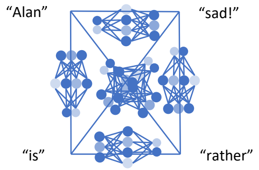
DISCLAIMER: I am not really a deep learning person and definitely not an NLP person, so I might (read will) make some errors

> 90% of these slides were made in the last 24 hours so they also probably have issues

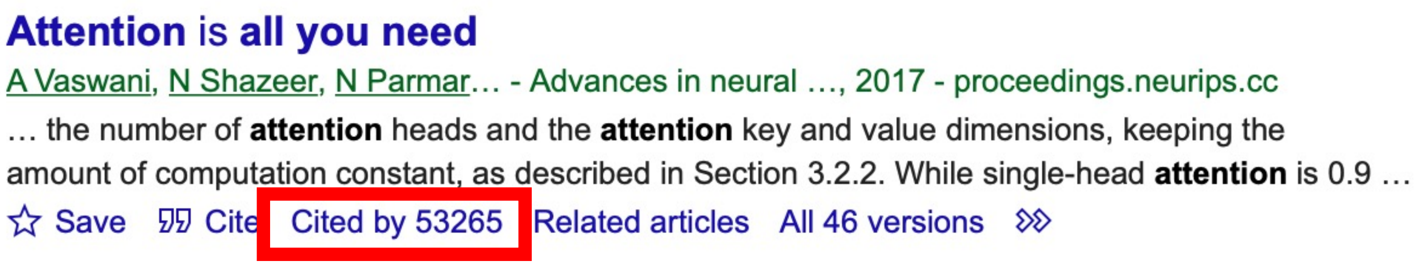
UBC Machine Learning Reading Group – Fall 2022
Alan Milligan
alanmil@student.ubc.ca

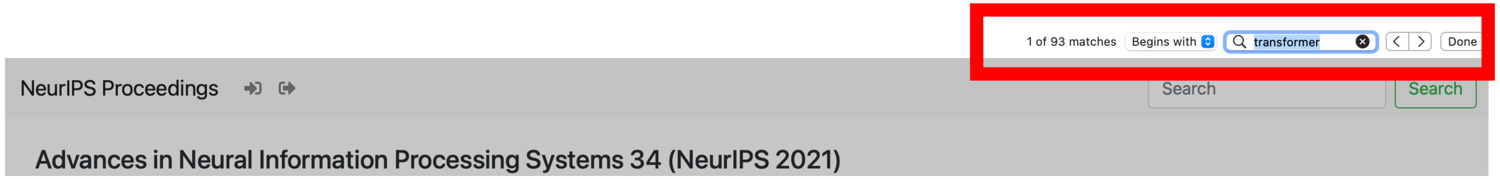
me after making slides!

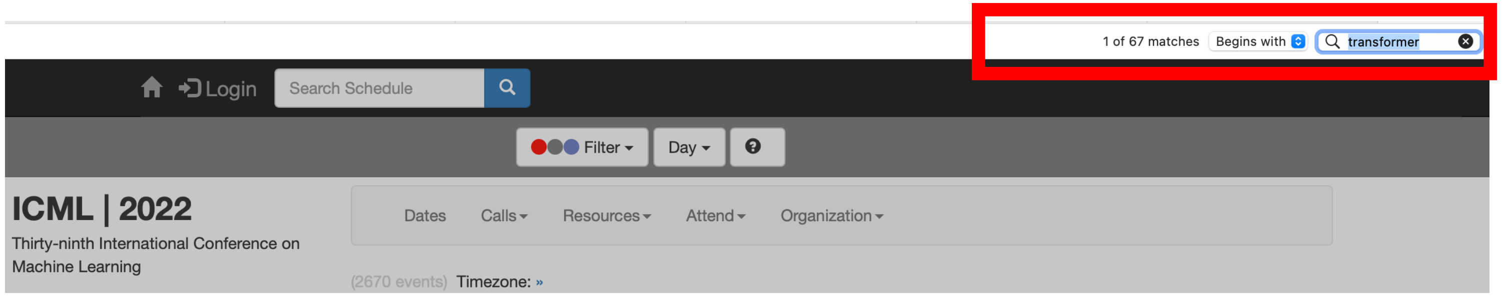
Not this Vaswani



We live in the age of transformers

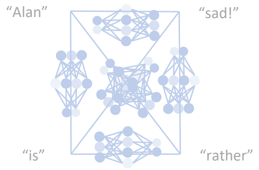






- In the way AlexNet and CNNs revolution computer vision, the advent of the transformer has revolutionized NLP (and several other fields)
- GPT-x, BERT, AlphaFold2, SWITCH-C, CLIP, DALL-E and many other famous models are all based on transformers
- If you want publications and have big (like really big) computers, you may want to consider training giant transformers

Machine Translation → LSTMs → Attention → Transformers



We live in the age of transformers

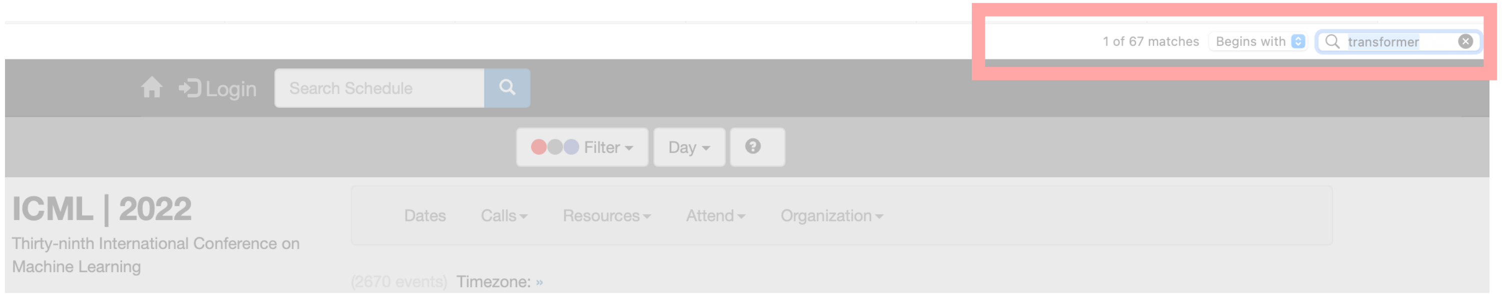
Attention is **all you need**

A Vaswani, N Shazeer, N Parmar… - Advances in neural …, 2017 - proceedings.neurips.cc

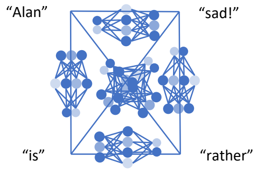
… the number of **attention** heads and the **attention** key and value dimensions, keeping the amount of computation constant, as described in Section 3.2.2. While single-head **attention** is 0.9 …

- In the way AlexNet and CNNs revolution computer vision, the advent of the transformer has revolutionized NLP (and several other fields)

TLDR: All those fancy models by Google/OpenAI/Deepmind/Meta are often just really big transformers



- If you want publications and have big (like really big) computers, you may want to consider training giant transformers.



Problem: I Don't Speak German (or anything else)

Suppose Fred and I go to his favourite German restaurant and I want to look (sound?) cool

"A blast from the past"

Rule Based Methods

- We can try to hard code language rules
- This incorporates lots of domain knowledge from the source and target language
- These were the original methods developed in the 1970s

Hard Coding?

Statistical Methods

- We look at frequencies across large corpora
- Modeling probabilities of translated phrases conditioned on original phrases
- Used things like Hidden Markov Models and context free grammars, much closer to "machine learning" and used in the mid 2000s

$$\tilde{e} = arg\max_{e \in e^*} p(e|f) = arg\max_{e \in e^*} p(f|e)p(e).$$

Machine Translation → LSTMs → Attention → Transformers

Problem: I Don’t Speak German (or anything else)

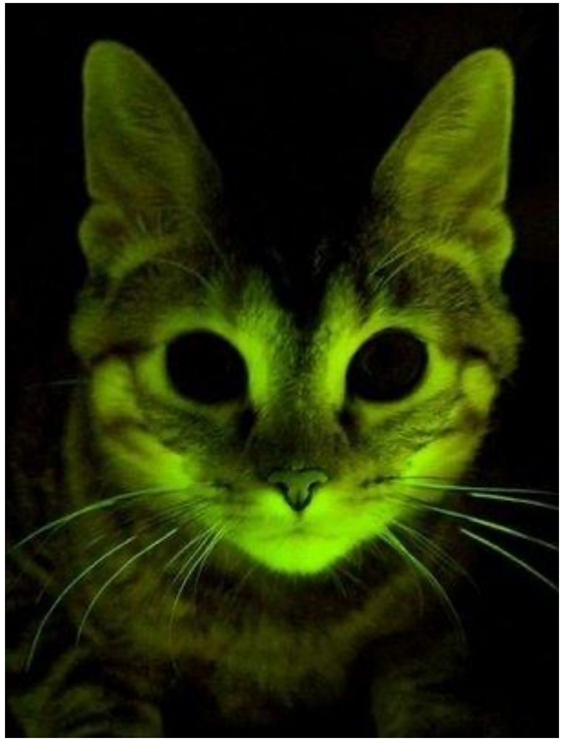
Suppose Fred and I go to his favourite German restaurant and I want to look (sound?) cool

I speak zero German but ϵ French so I will use this example

“I like the green cat.”

How can we go about translating this (simple) sentence?

“J’aime le chat vert.”



Problem: I Don’t Speak German (or anything else)

Suppose Fred and I go to his favourite German restaurant and I want to look (sound?) cool

I speak zero German but ϵ French so I will use this example

“I like the green cat.”

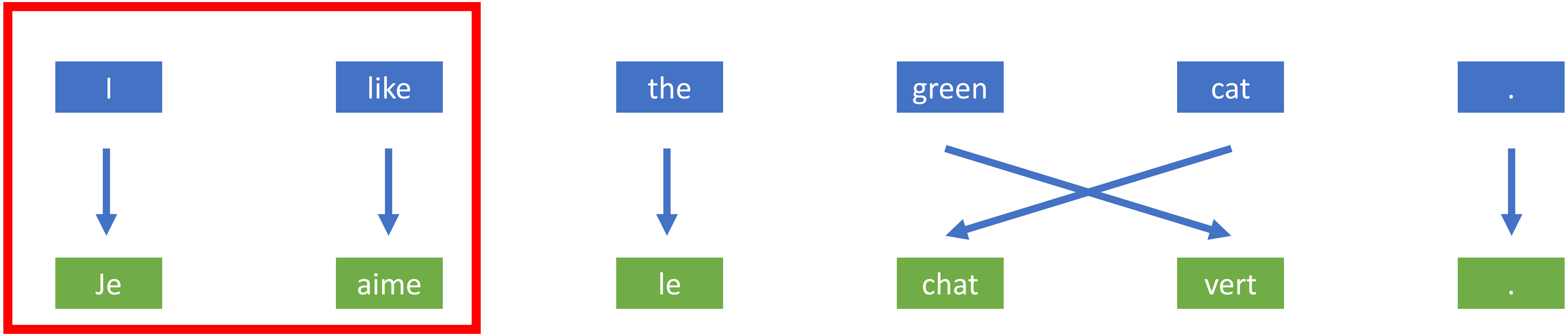
“J’aime le chat vert.”

Problem: I Don’t Speak German (or anything else)

Suppose Fred and I go to his favourite German restaurant and I want to look (sound?) cool

I speak zero German but ϵ French so I will use this example

“I like the green cat.”



How do we handle cross-language contractions?

“J’aime le chat vert.”

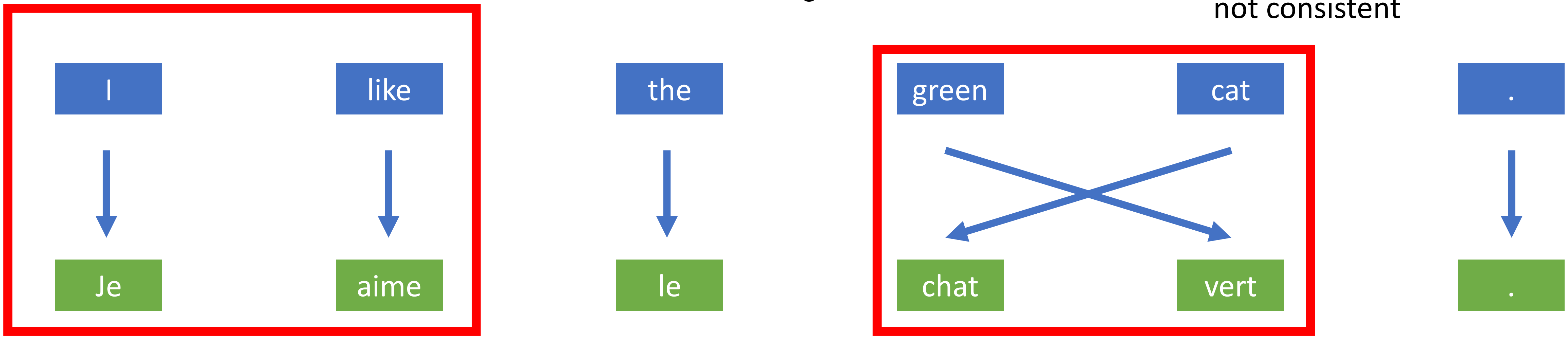
Problem: I Don’t Speak German (or anything else)

Suppose Fred and I go to his favourite German restaurant and I want to look (sound?) cool

I speak zero German but ϵ French so I will use this example

“I like the green cat.”

Language “order” is often not consistent



How do we handle cross-language contractions?

“J’aime le chat vert.”

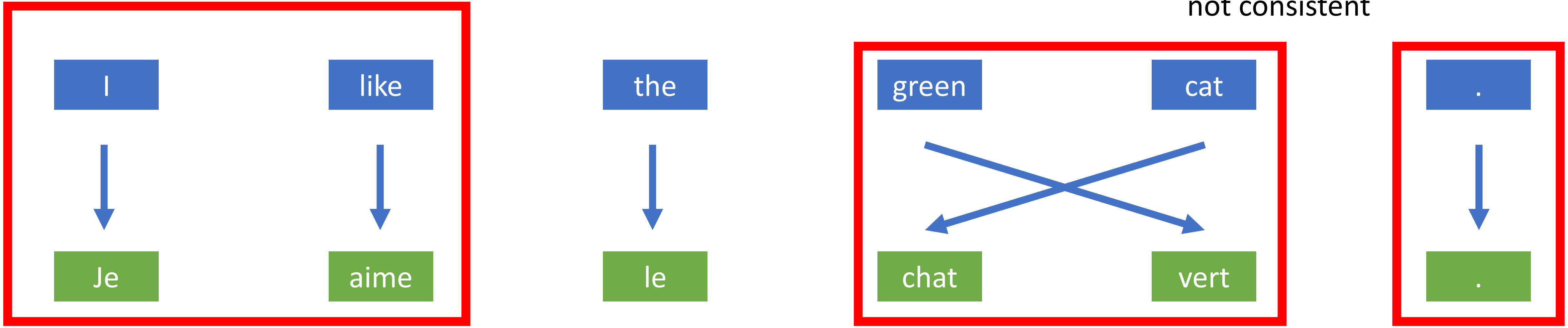
Problem: I Don’t Speak German (or anything else)

Suppose Fred and I go to his favourite German restaurant and I want to look (sound?) cool

I speak zero German but ϵ French so I will use this example

“I like the green cat.”

Language “order” is often not consistent



How do we handle cross-language contractions?

“J’aime le chat vert.”

How do we handle the end (and start) of sentences?
What about variable lengths?

Machine Translation → LSTMs → Attention → Transformers

Problem: I Don’t Speak German (or anything else)

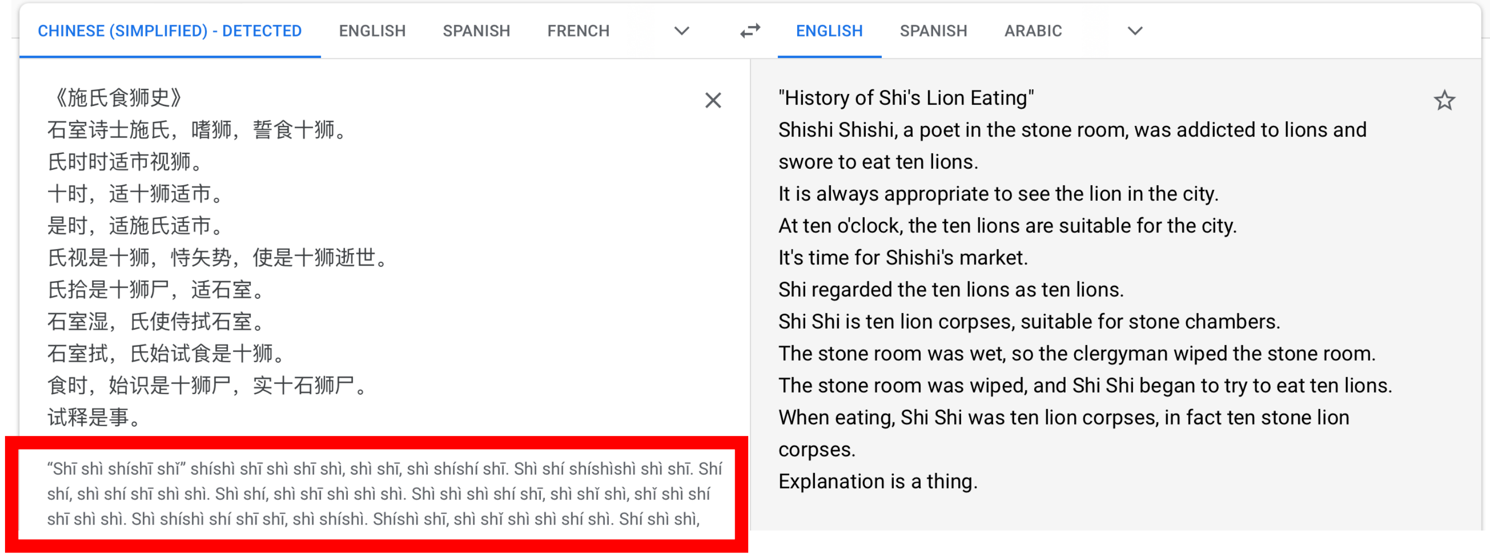
Suppose Fred and I go to his favourite German restaurant and I want to look (sound?) cool

“I like the green cat.”

↓

“J’aime le chat vert.”

Good rule based or statistical models could probably handle this fine but things get harder…



Oh hey it’s the mid 2010s and deep learning just happened! What can we do!

"Alan" "sad!" "is" "rather"

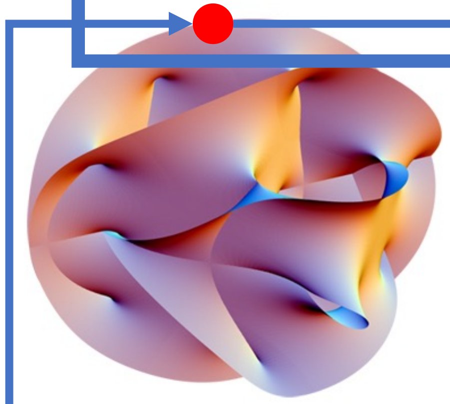
Possibly flawed intuition: The Manifold of Meaning

- Suppose there exists some continuous space of "meaning", where every sentence in every language is represented the same

- Could we build a function that takes a language into meaning land?

"Meaning land" – language independent

"J'aime le chat vert."
"I like the green cat."



- And then another function to take it into a different language?

"I like the green cat."

Magic

Magic 2

"J'aime le chat vert."

Deep learning is magic (alchemy?) so let's try that!

Machine Translation → LSTMs → Attention → Transformers

How do we make sentences continuous?

“I like the green cat.”

↓

[“I”, “like”, “the”, “green”, “cat”, “.”]

↓

[23, 796, 4012, 8923, 4850, 42]

↓

[-2.51, 0.727, -0.943, -0.935, -0.072, 0.573],
[-0.4, -2.73, 0.906, 0.026, 0.216, 0.571],
[0.169, -1.13, 0.181, 0.086, 0.242, -2.35],
[0.279, 1.11, -0.896, 0.593, -0.211, -1.21],
.
.
.
[-2.03, 0.799, 0.448, 1.34, 1.14, -0.639],
[-1.71, -0.163, 0.863, -1.01, 1.21, 0.73],
[-0.932, -0.599, 0.558, -1.13, 1.54, 1.34],
[0.782, -0.542, -0.00264, -0.99, -1.9, 0.399]

d_{embed}

Tokenization

- A sentence is broken up into “tokens”
- These could be words, word parts, or characters
- There also special tokens like “<SOS>”, “<EOS>”, “<OOV>”, and “<PAD>”

Convert to token IDs

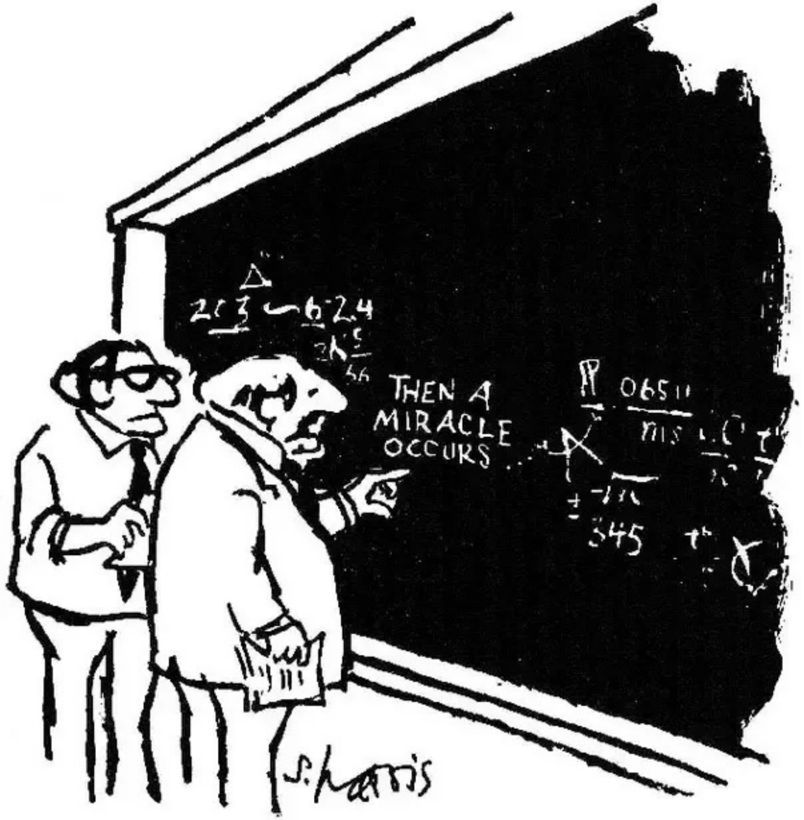
- Each token is assigned an ID from a predefined lookup table
- You could think of this like a one hot vector but its usually a dictionary

Token Embedding

- Each token ID is assigned a vector in a fixed dimension (512 in Vaswani)
- These vectors are initialized with Gaussian noise
- Over the course of training, the embedding vector is treated as a parameter and gradients are propagated into the vector to train

$x_{embed} \xleftarrow{\nabla}$ Model $\xleftarrow{\nabla}$ Loss

Warning: Entering hand wavey zone



"I think you should be more explicit here in step two."

Recurrent Neural Network TLDR

How about we have a neural network eat these vectors one by one, plus the output of the previous step

Encoding

Do this for each of the n_{src} input embeddings

$$h_t = f(x_t, h_{t-1})$$

$\rightarrow h_{encoding} \rightarrow$

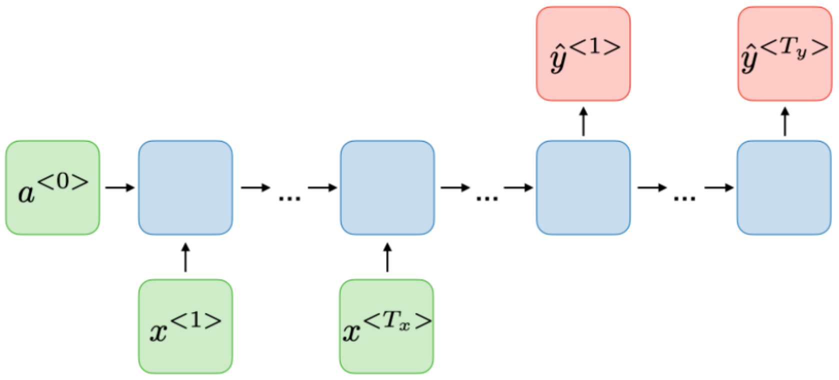
Decoding

Do this until the network produces an end token

$$y_{t'} = f(h_{t'-1}, [y_{t'-1}])$$

x_t = source embedding vectors
h_t = latent state vectors
y_t = predicted token
f = neural network

Visually, (using some Stanford guy's notation)



PROBLEMS:

- It's hard to model long range dependencies because the model sees x_n much later than x_0
- $h_{encoding}$ is of fixed size, so it can only hold so much information (related to the first point)
- Optimization is hard because propagating gradients backwards in time involves taking matrices to high powers, leading to vanishing or exploding gradient

Recurrent Neural Network TLDR

How about we have a neural network eat these vectors one by one, plus the output of the previous step

Encoding

Do this for each of the n_{src} input embeddings

$\rightarrow$ $h_{encoding}$ $\rightarrow$

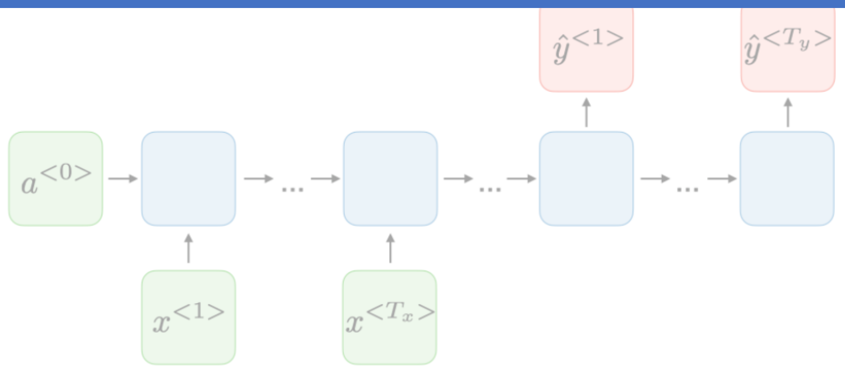
Decoding

Do this until the network produces an end token

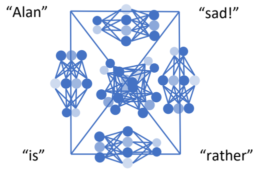
x_t = source embedding vectors
h_t = latent state vectors
y_t = predicted token

Note: I am skipping a lot of details like what the inside of that neural network actually looks like, but hopefully this gives an intuition

model sees x_n much later than x_0



- $h_{encoding}$ is of fixed size, so it can only hold so much information (related to the first point)
- Optimization is hard because propagating gradients backwards in time involves taking matrices to high powers, leading to vanishing or exploding gradient



Long Short-Term Memory TLDR ("fancy" RNNs)

Instead of just storing h_t as a function of x_t and h_{t-1}, we can also store another hidden state c_t

Possibly Flawed Intuition

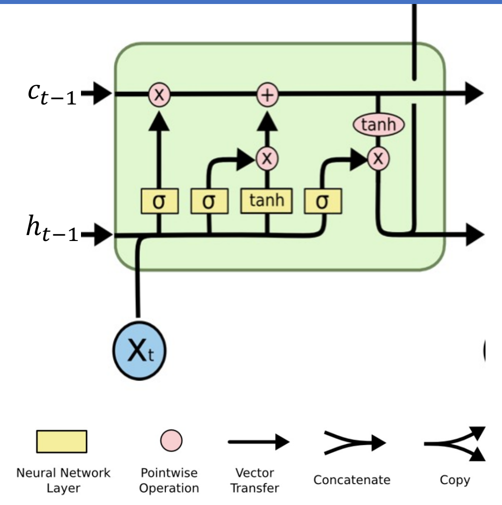
- Let h_t be the "short term memory," updated by a network output of each new input and previous short-term memory

- Let c_t be the "long term memory," updated by a learned linear combination of previous memory and the input

Unreadable Math

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \quad \text{(forget)}$$
$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \quad \text{(input)}$$
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \quad \text{(output)}$$
$$\tilde{c}_t = \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \quad \text{(candidate memory)}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \quad \text{(updated long memory)}$$
$$h_t = o_t \odot \sigma_h(c_t) \quad \text{(updated short memory)}$$

LSTM Cell



Benefits

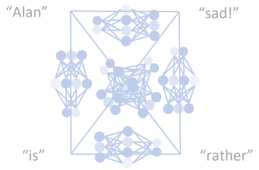
- Long term memory acts like residual connections which allow for much better gradient flow during optimization

- (ideally) the network can learn to remember important stuff in c_t

Remaining Issues

- The amount of information that can be propagated forward is still fixed and long term dependencies can still be forgotten

- Vanishing/Exploding gradient can still happen albeit less



Long Short-Term Memory TLDR ("fancy" RNNs)

Instead of just storing h_t as a function of x_t and h_{t-1}, we can also store another hidden state c_t

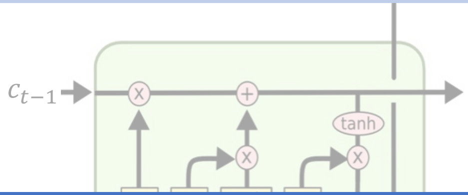
Possibly Flawed Intuition

- Let h_t be the "short term memory," updated by a network output of each new input and previous short-term

Unreadable Math

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \text{ (forget)}$$
$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \text{ (input)}$$

LSTM Cell



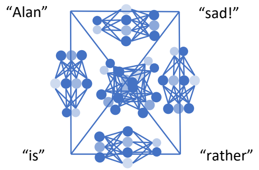
Note: I am skipping a lot of details again

Benefits

- Long term memory acts like residual connections which allow for much better gradient flow during optimization
- (ideally) the network can learn to remember important stuff in c_t

Remaining Issues

- The amount of information that can be propagated forward is still fixed and long term dependencies can still be forgotten
- Vanishing/Exploding gradient can still happen albeit less



Summary: RNNs are problematic

Fixed Context Information

c_t and h_t have predetermined size, so in problems with large inputs it will be challenging to squeeze all information into these vectors

Long Term Dependency Issues

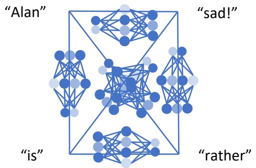
The sequential processing of inputs means that inputs early in the sequence can be forgotten in very long sequences, even in LSTMs

Optimization Issues

Backpropagation through time means exponentiating matrices many times, leading to exploding or vanishing gradients depending on the eigenvalues

Parallelization Issues

RNNs are fundamentally ***sequential***, meaning it is impossible to parallelize processing of a sequence, slowing down training and inference



Summary: RNNs are problematic

Fixed Context Information

c_t and h_t have predetermined size, so in problems with large inputs it will be challenging to squeeze all information into these vectors

Long Term Dependency Issues

The sequential processing of inputs means that inputs early in the sequence can be forgotten in very long sequences, even in LSTMs

Optimization Issues

Backpropagation through time means exponentiating matrices many times, leading to exploding or vanishing gradients depending on the eigenvalues

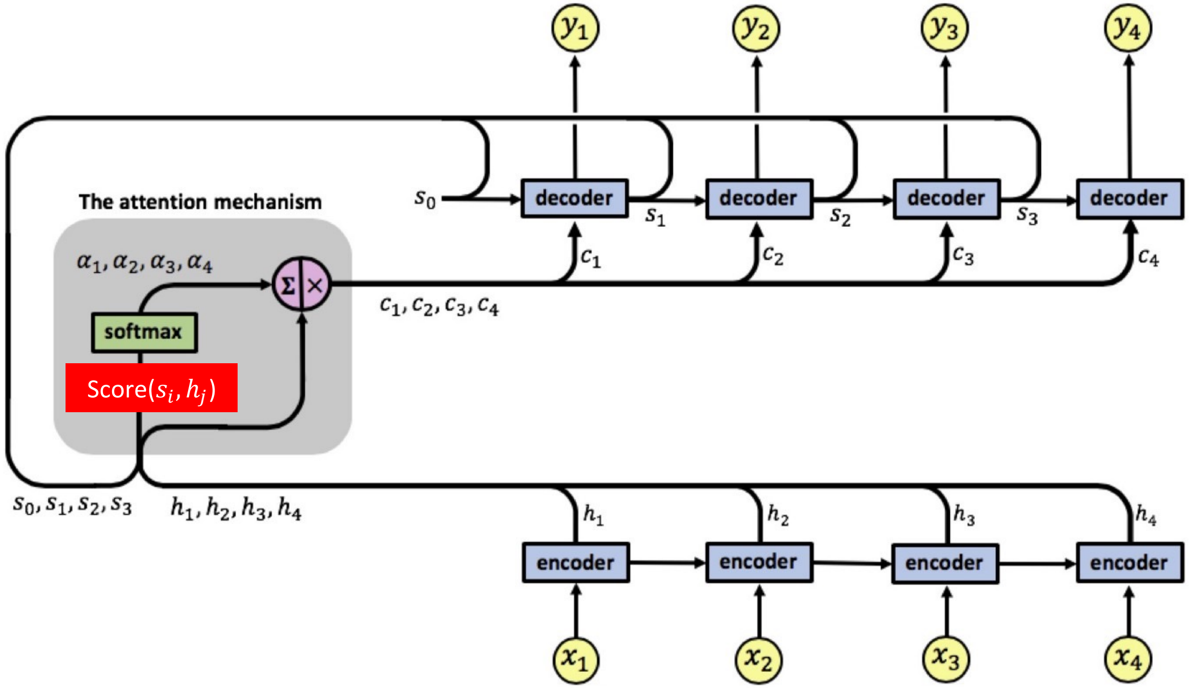
Parallelization Issues

RNNs are fundamentally ***sequential***, meaning it is impossible to parallelize processing of a sequence, slowing down training and inference

Finally, pay Attention (in an RNN)

How can we better model long range dependencies?

Idea: In the decoding phase, use a weighted combination of all h_t so that we “pay attention” to the more important parts of the h_t



Note this diagram happens in sequence not all at once

Step 1

Generate all h_t in the encoding phase

Step 2

Repeat for until <EOS> token

Step 2.1

Compute Score(s_i, h_j) for current decoder state s_i and all encoder states h_j

Step 2.2

Compute attention weights as Softmax(scores)

Step 2.3

Compute context vector c_i as attention weighted sum of all h_j

Step 2.4

Decode using s_i and c_i as decoder input

Step 3

Profit

What is score though?

Sounds reasonable, but in order to compute the attention weights, we need some sort of scoring function

Options from Lilian Weng's blog

Name	Alignment score function	Citation
Content-base attention	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \text{cosine}[\boldsymbol{s}_t, \boldsymbol{h}_i]$	Graves2014
Additive(*)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\boldsymbol{s}_t; \boldsymbol{h}_i])$	Bahdanau2015
Location-Base	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \boldsymbol{s}_t)$ Note: This simplifies the softmax alignment to only depend on the target position.	Luong2015
General	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \mathbf{W}_a \boldsymbol{h}_i$ where $\mathbf{W}_a$ is a trainable weight matrix in the attention layer.	Luong2015
Dot-Product	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \boldsymbol{h}_i$	Luong2015
Scaled Dot-Product(^)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state.	Vaswani2017

Some authors use score functions with learned parameters

We focus on scaled dot-product attention as it is used in Vaswani et al.

$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$$

Recall from math that a dot product is a measure of similarity in a vector space

What is score though?

Sounds reasonable, but in order to compute the attention weights, we need some sort of scoring function

Options from Lilian Weng's blog

Name	Alignment score function	Citation
Content-base		
	where W_a is a trainable weight matrix in the attention layer.	
Dot-Product	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \boldsymbol{h}_i$	Luong2015
Scaled Dot-Product(^)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state.	Vaswani2017

Some authors use score functions with learned parameters

TLDR: Attention is a method of deciding which inputs to care about

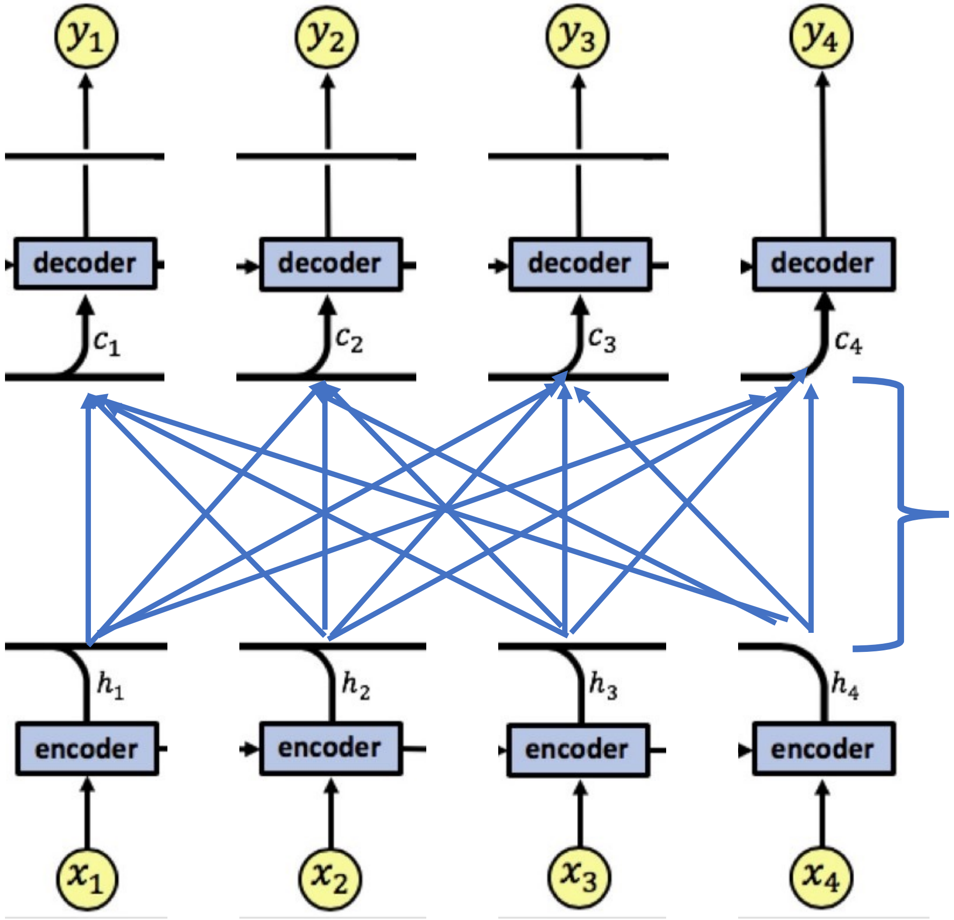
in Vaswani et al.

$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$$

Recall from math that a dot product is a measure of similarity in a vector space

Enter Michael Bay: The Transformer

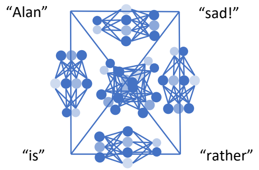
In an RNN with attention we are using all h_t, why don't we just ditch the recurrent part



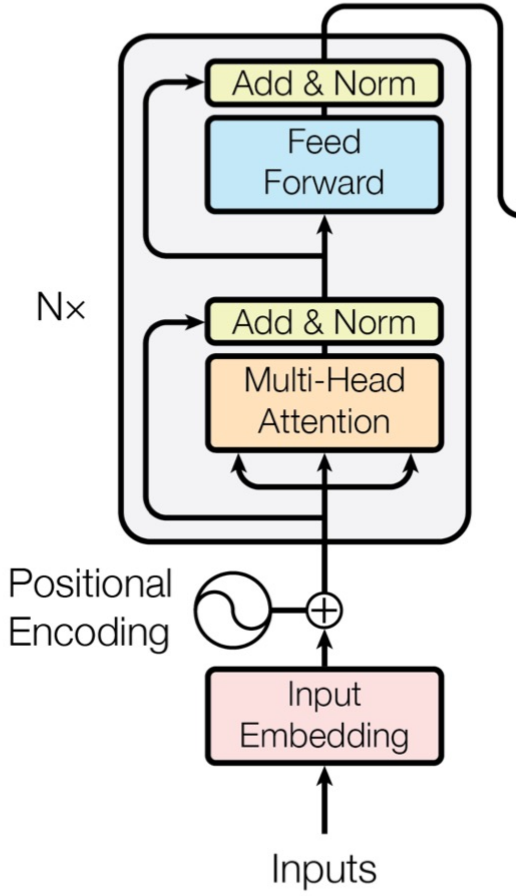
Every arrow here is an attention computation

Vaswani et al. proposes: "Attention is all you need"

- Have an encoder where all input embeddings pay attention to all other input embeddings
- Add "positional encodings" to input embeddings so that the sequential structure is retained
- Have a decoder that pays attention to all input embeddings as well as the already decoded embeddings



Step 1: Self-Attention Encoder



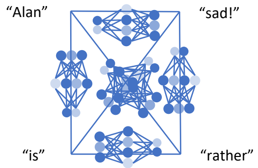
Lets build a Transformer Encoder!

Encoder Objective

Create an "interesting" learned representation of the inputs useful for the Decoder (next)

Ingredients

1. Embeddings + Positional Encodings
2. Multi-Head Attention
3. Residual Connection and Layer Norm
4. Fully Connected Layer



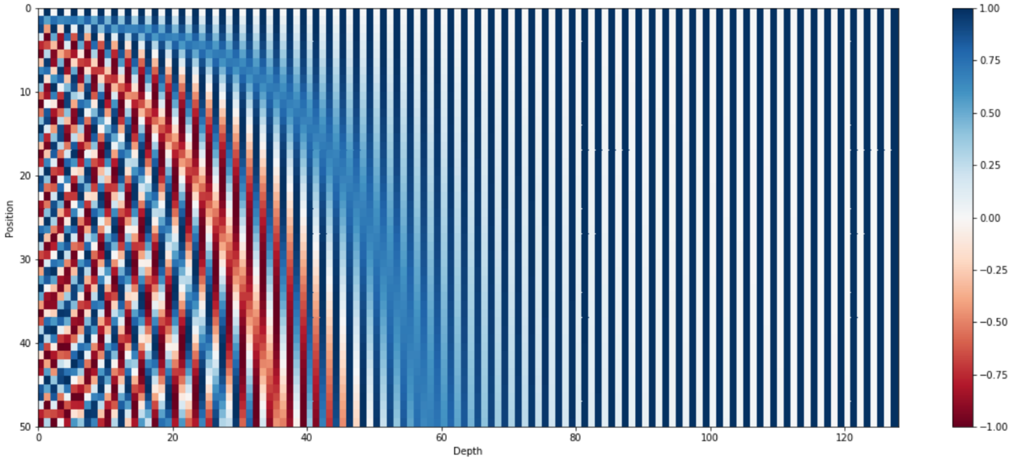
Step 1.1: Positional Encodings

Word Embedding

Same as earlier, Token IDs mapped into real vectors that are learned during training

Positional Encoding

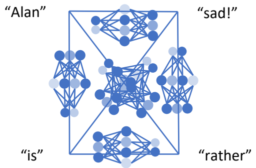
$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$



- Vectors defined by this formula are added to each word embedding vector to add “relative position” between embeddings

- Not intuitive to me, but this formula allows for easy relative position learning via some trigonometric addition identities

- You can also just add torch.nn.Embedding style absolute positional encodings in the same way as word embedding and learn via backpropagation (Vaswani et al. tested this with similar results)



Step 1.2: Multi-Head Self-Attention

Here is where the real magic happens

Recall Scaled Dot-Product Attention

$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$$

$$\text{Attention}(\mathbf{s}_t, \mathbf{h}_i) = \frac{e^{\text{score}(\mathbf{s}_t, \mathbf{h}_i)}}{\sum_{j=1}^{n} e^{\text{score}(\mathbf{s}_t, \mathbf{h}_j)}}$$

- In Vaswani et al, we do a learned projection of the input to produce matrices Q, K, which are analogous to s, h above

- The columns of the non-linear matrix product are akin to the attention weights in the RNN example, we use these weights on another learned projection of the input V

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

More is Better

- Rather than doing attention once on the input, lets do it N (= 6) times

- N different copies of projection matrices are learned, attention is run N times, and then all outputs are projected back to d_{model}

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

- More heads mean different heads can learn different things like grammar and vocabulary

- Also ensemble good

Step 1.3: Layer Norm and Residual Connections

Layer Normalization

- Given the activations of a layer, we compute the mean and standard deviation

- We subtract and divide by these values respectively, then multiply and add by learned parameters (so that the identity can be learned as in Batch Norm)

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$$

$$\text{LayerNorm}(x) = \frac{\gamma}{\sigma^l}(x - \mu^l) + \beta$$

something something “reduce internal covariate shift”

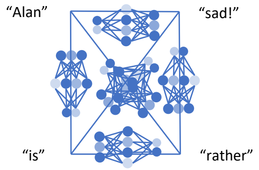
Residual Connections

- As with ResNets from vision, instead of directly transforming the input, we learn a residual, then apply layer norm

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

Where Sublayer(x) is either Multi-Head Attention or a feed forward network

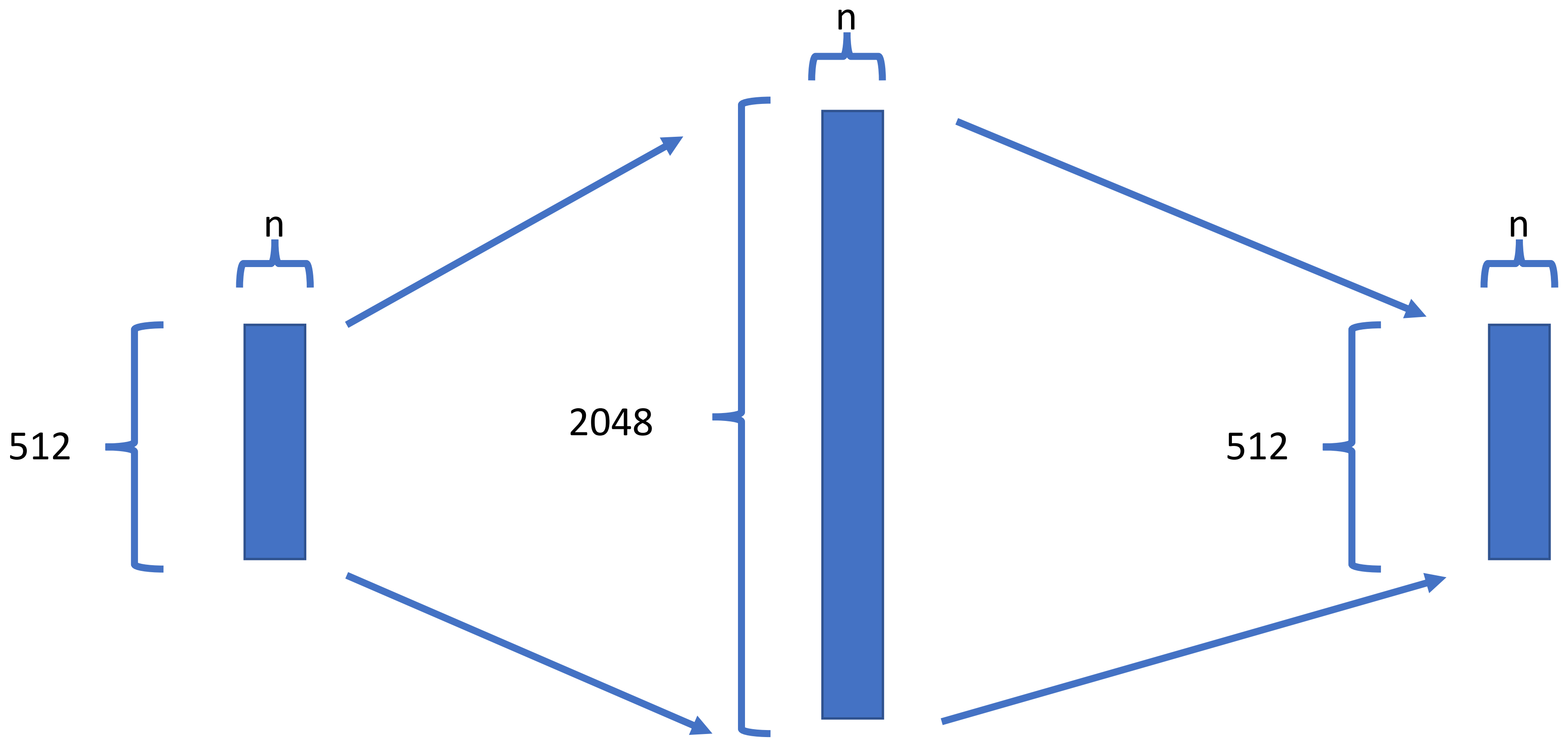
- Residual Connections have been shown to make optimization easier in cases where transformations should be close to identity maps



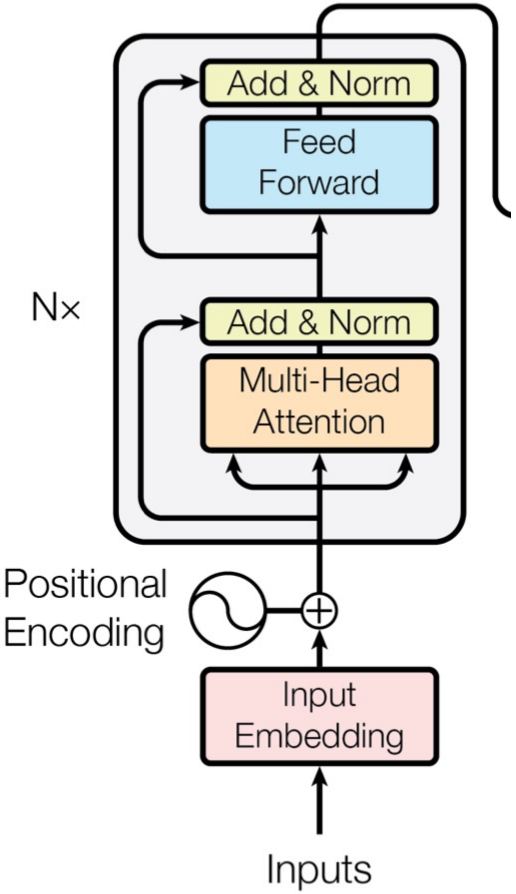
Step 1.4: Position-wise Feed Forward Network

Here we have a simple two layer ReLU network that acts on each embedding individually

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$



Self-Attention Encoder Assembled!

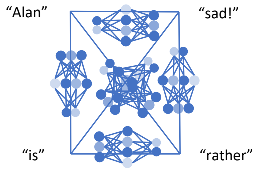


- With these pieces, we can now create a latent representation of an input sentences where each vector has applied self attention N times across h heads

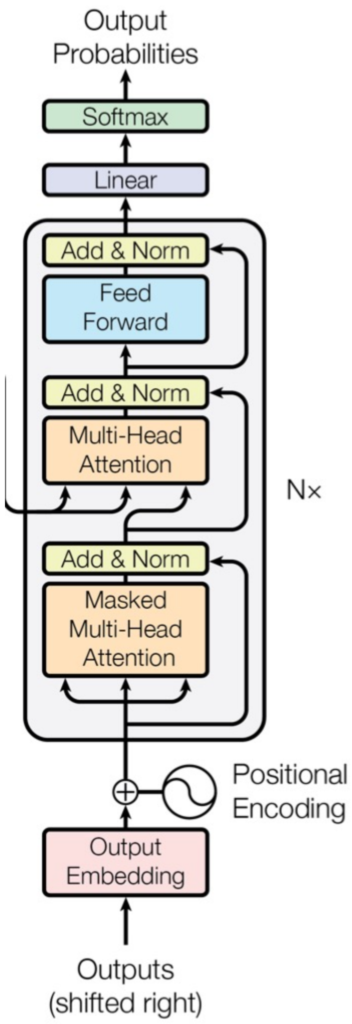
- In Vaswani et al, N = 6 and h = 8

- The output is a matrix of embedding vectors in $\mathbb{R}^{n \times d_{model}}$ (n x 512)

- We can now use this in step 2: the Decoder



Step 2: Masked and Cross Attention Decoder

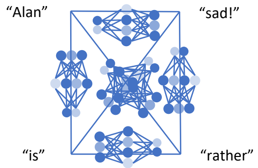


Decoder Objective

Given Encoder("I like the green cat. <EOS>") and Decoder("<SOS> J'aime le chat "), predict "vert".

Ingredients

1. Embeddings + Positional Encodings **[Done]**
2. Masked Self Attention
3. Residual Connection and Layer Norm **[Done]**
4. Encoder-Decoder Cross Attention
5. Fully Connected Layer **[Done]**
6. Output Layer



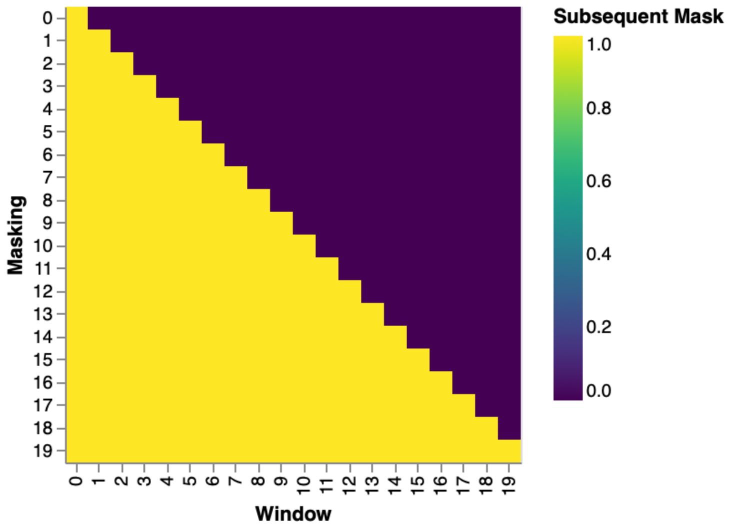
Step 2.2: Masked Self Attention

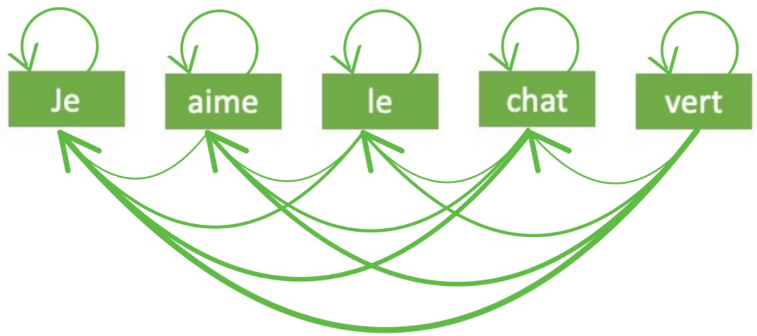
- Vaswani et al. wanted to "preserve [the] auto-regressive property" of the model, meaning that no word should be able to attend to words decoded after it

- This is accomplished with "masking," which essentially sets the score of later entries to zero

Visualizing Legal Attention Connections

This triangular mask represents which position each position can attend to





As math,

$$\text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}} \odot M\right)V$$

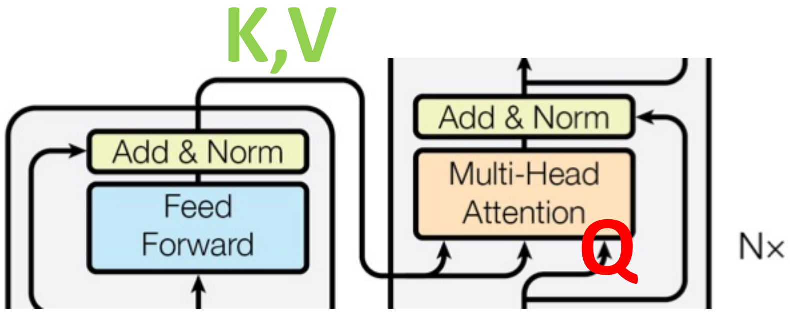
Where M is the lower triangular matrix on the left

Step 2.4: Encoder-Decoder Cross Attention

- We want the decoder to be able to use the "interesting" representation learned by the encoder
- This is done by letting the decoder embeddings attend to the keys and values of the Encoder

Projected Decoder Output   Projected Encoder Output

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



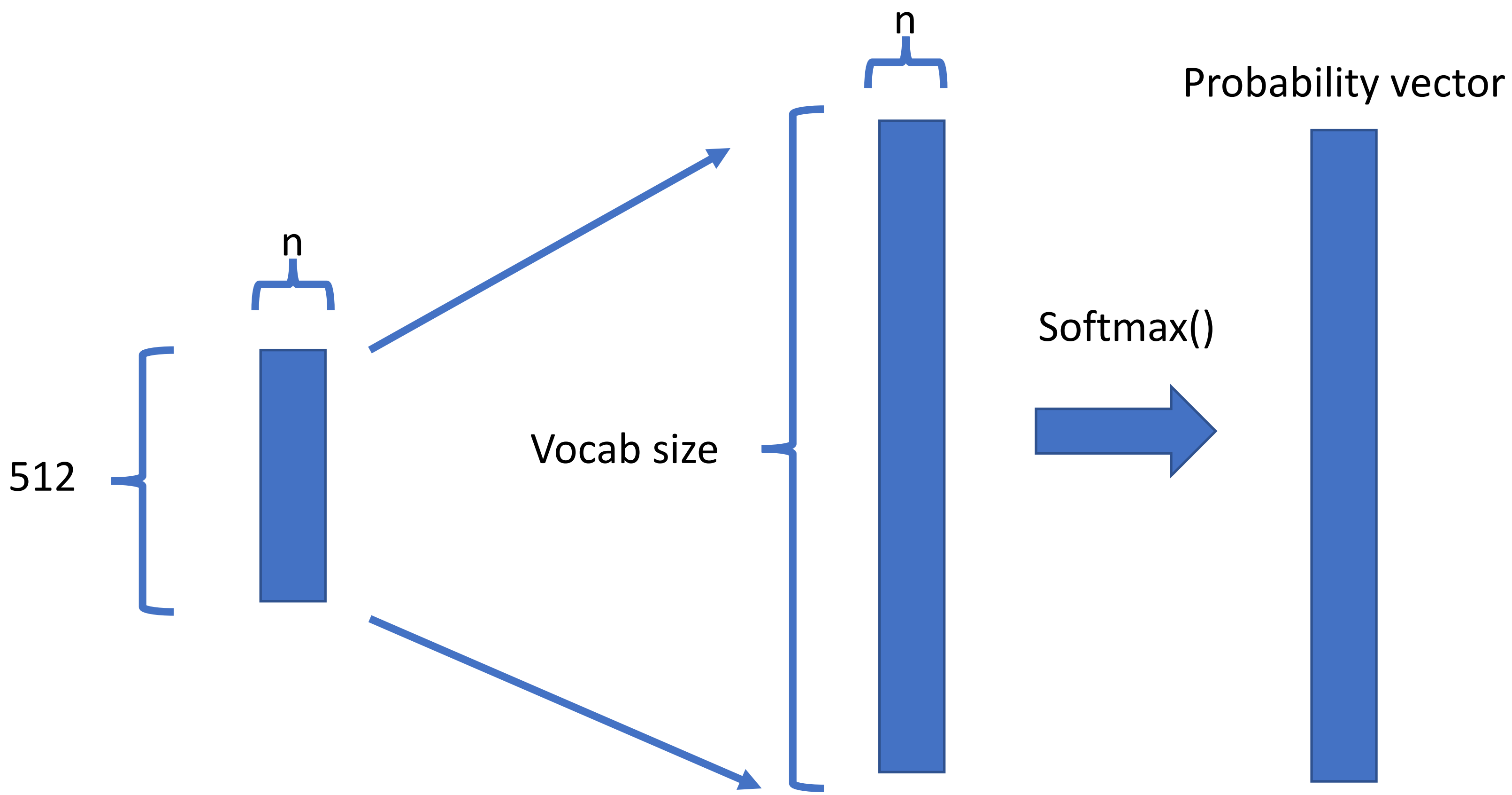
As before, this is really multi-head cross attention

Decoder Output   Encoder Output

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$
$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

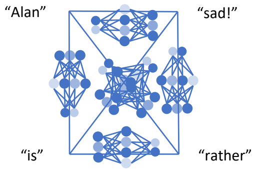
Step 2.6: Output Projection

After N decoder layers, we project up to the dimension of the target vocabulary and softmax for predictions

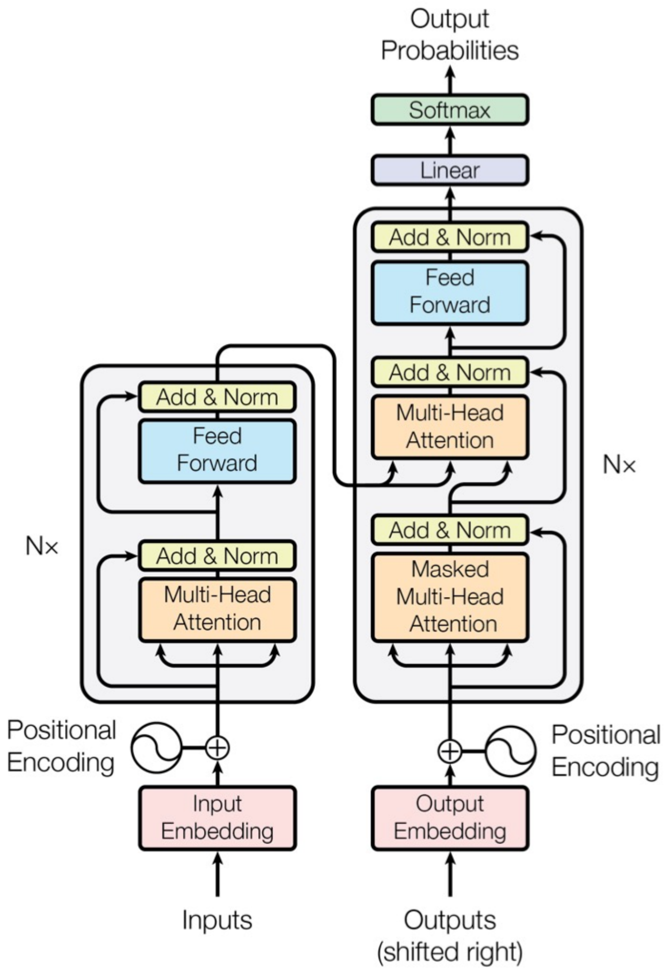


Important Note:

During training/testing, we feed in the whole target sentence shifted by <SOS> since the self attention mask will make it seem like you are doing a one step prediction at every position. This was non-obvious to me.



We have assembled the full Transformer!



- That was a lot of deep learning jargon that I don't expect everyone to understand

- It took me over 10 attempts to grasp all of this and I still have questions

Transformer TLDR:

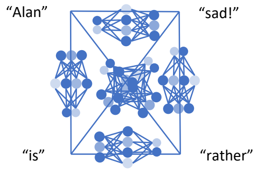
- Embed source words with some learnable vector plus positional encodings

- Run a few rounds of scaled dot product self attention plus a layer normalized feedforward network for your source embeddings

- Embed known target words (or <SOS>) with some learnable vector plus positional encodings

- Run a few rounds of forward masked self attention, cross attention with the encoded source sentence, layer normalization, and a feedforward network

- Project and softmax the output, profit



How to train your transformer

We now know the architecture, but there are still some training details

Regularization

1. Despite no mention in the paper, all implementations I've seen use "weight decay," also known as L2 regularization

2. Dropout is applied to both attention and feed forward layers as well as the embeddings

3. "Label Smoothing" which punishes incorrect softmax outputs slightly less

Loss Function

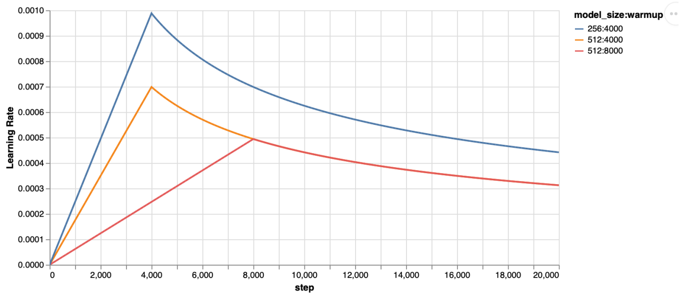
- Cross Entropy loss is applied between the prediction vector and (smoothed) label vector

- The loss is computed independently for each prediction in a forward pass, recalling that we make several predictions concurrently

Optimizer

- Everyone's good friend Adam is used for optimization with the following odd learning rate schedule using 4000 warmup steps (no justification provided)

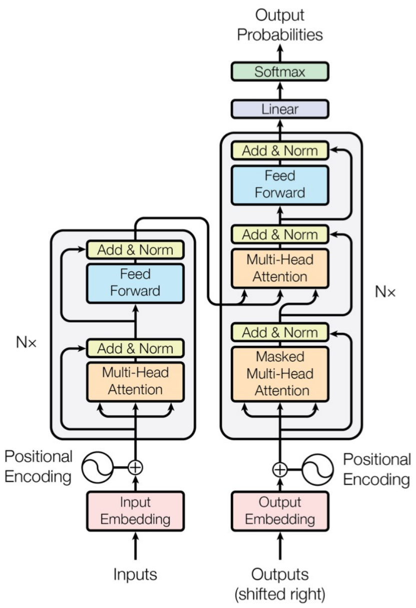
$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$



How to inference your transformer

Forward Pass

You get a probability vector for the next word



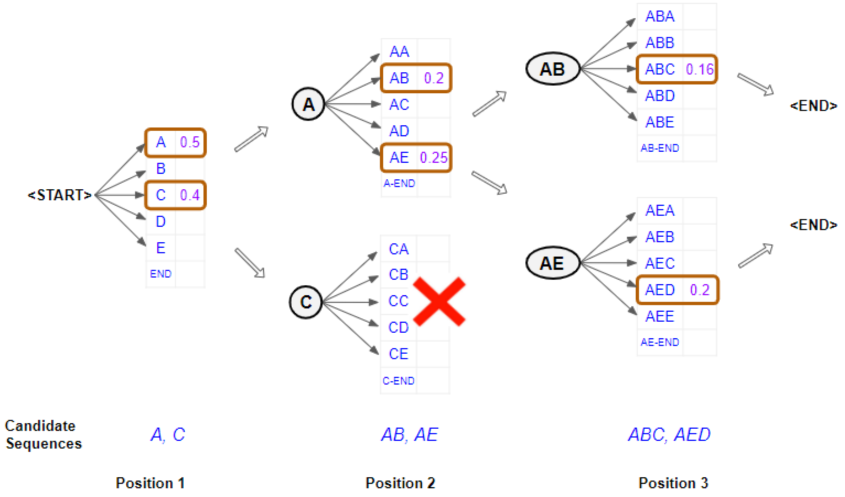
"I like the green cat."

"<SOS>"

Beam Search

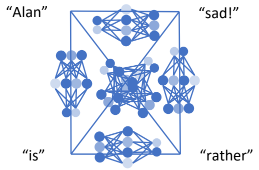
- To get the better predictions, Vaswani et al. (and NLP in general) will use *BeamSearch(b)*, a process where we autoregressively predict *b* copies until all reach <EOS>

- Each beam is ranked by total probability, and we only propagate the top *b* at any given time (the others die)



In this example *b* = 2

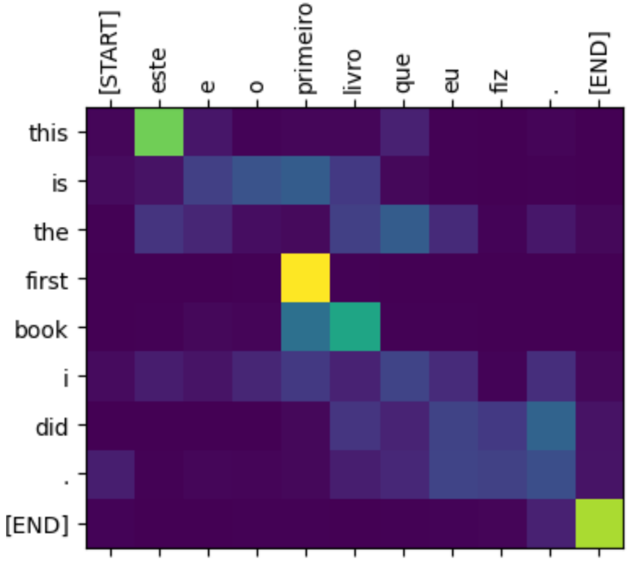
Machine Translation — → LSTMs — → Attention — → Transformers



Transformer Benefits

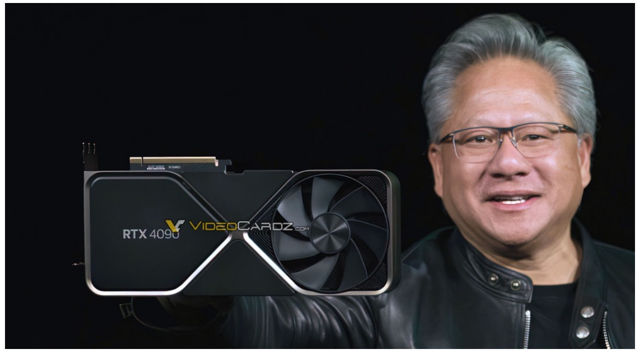
Long Range Dependency

- Self-attention can model arbitrarily long sequences in constant distance
- This completely removes the issue RNNs face about forgetting the start (or middle for bidirectional) of the sequence
- Below are visualized attention weights during translation



Computational Efficiency

- Since we no longer have to process sentences token by token, Transformers are extremely parallelizable and GPU friendly
- Self attention masking means each training sequence of length n gives us n gradients from one forward pass
- Every attention had can be on a on a different machine, every layer can be on a different machine, the encoder and decoder can be on different machines, etc.
- Some argue the true reason transformers perform is simply that we are able to scale them to levels that would be impossible for other methods



Machine Translation → LSTMs → Attention → Transformers

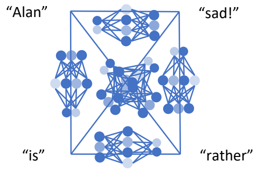
Generic Results Table

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on th
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	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

I have elected to leave BLEU out of the talk, but it's a
measure of how good translation is and more is better

Also note the FLOPs difference relative to performance



My unsolicited comments

Good stuff

- If you read machine learning literature, you know these models have revolutionized several fields
- There are some very cool tricks and innovations that I have struggled to highlight in this paper, notable how self attention masking can give you multiple independent gradients for in a single pass
- I like that "*X is all you need*" has become a meme title
- Transformer based models generate the best news headlines

ART IS IN THE AI OF THE BEHOLDER —

AI wins state fair art contest, annoys humans

Stealth win for AI-generated art inspires heated ethics debate on social media.

Not good stuff

- Training is great if you have a DGX A100 server lying around, but it took 2 days on my laptop to get through 20 epochs of 1000 before my laptop threw some OS error and killed it
- Self-attention across all inputs is an $O(n^2)$ operation (all inputs attend to all other inputs), which can very extremely costly when you deal with things like images
- There is much in trying to solve the above problem such as the Performer, Linformer, and Reformer (great original names guys)
- Despite the technical innovations and contributions, the quality of Attention is all you need as a paper is f REDACTED REDACTED.....

Machine Translation → LSTMs → Attention → Transformers

🥳

Thanks for listening !

Feedback appreciated I am very inexperience at presenting technical content

see there's a typo!