

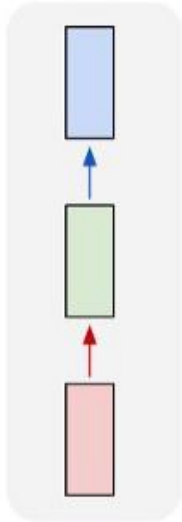
Attention & Transformer

Gone are the days of strict sequential processing

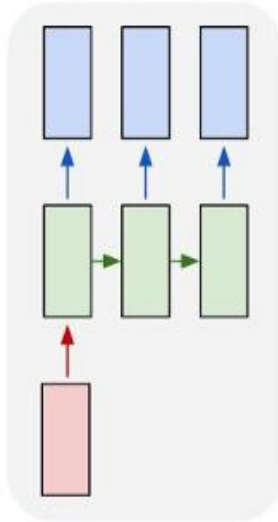
**What did we have
earlier ?**

Different types of RNN's

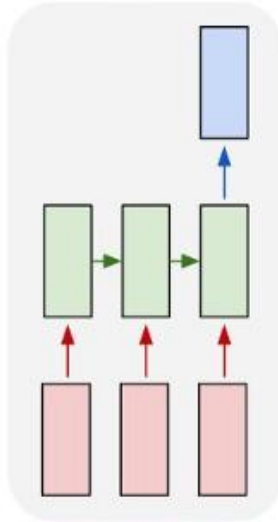
one to one



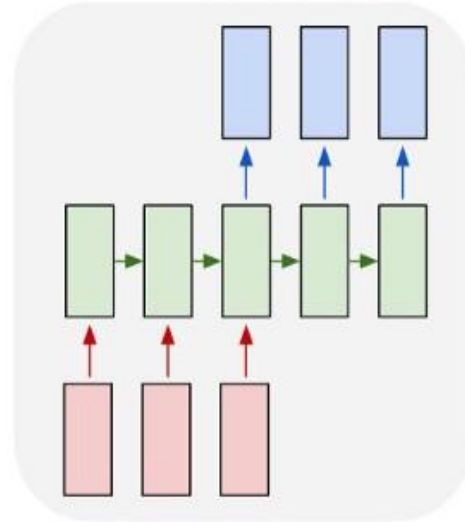
one to many



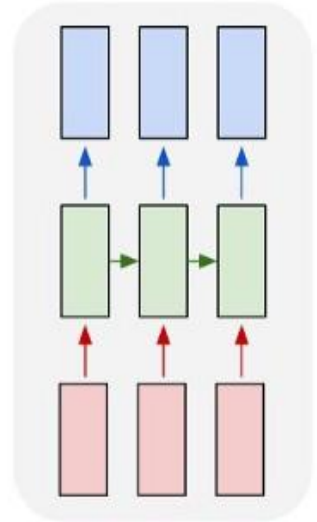
many to one



many to many



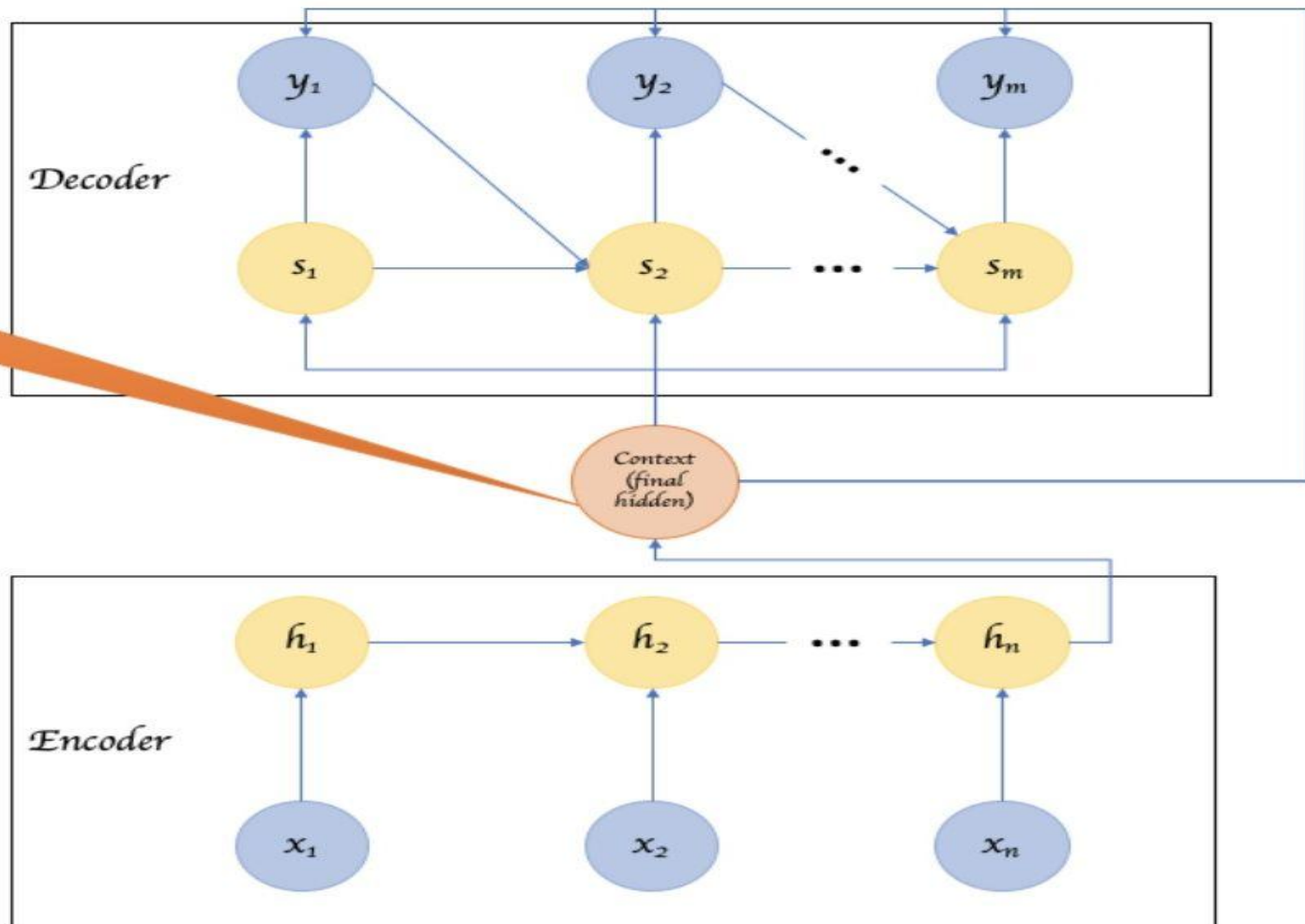
many to many



Seq2Seq Models

1. A sequence-to-sequence model is a model that takes a sequence of items (words, letters, features of an images...etc) and outputs another sequence of items.
2. The model is composed of an encoder and a decoder. The encoder processes each item in the input sequence, it compiles the information it captures into a vector (called the context). After processing the entire input sequence, the encoder sends the context over to the decoder, which begins producing the output sequence item by item.
3. Sequence-to-sequence models are deep learning models that have achieved a lot of success in tasks like machine translation, text summarization, and image captioning.

Everything's
crammed into
this vector



The Drawbacks

First Drawback

But is this sequential nature of processing important or does it put us at a disadvantage? There are languages where word order doesn't strictly matter like Polish and Hungarian. Or even in English, where we can change word order depending on what we want to emphasize.

Intuitively speaking, this strict order of processing is perhaps akin to flattening a 2-D image, i.e. converting from a matrix into a vector and using a vanilla feed-forward network to process it. It is much less efficient compared to CNN architectures that preserves the natural spatial relationship in the matrix representation.

This strict sequential nature of processing is perhaps the **first drawback**.

The Solution

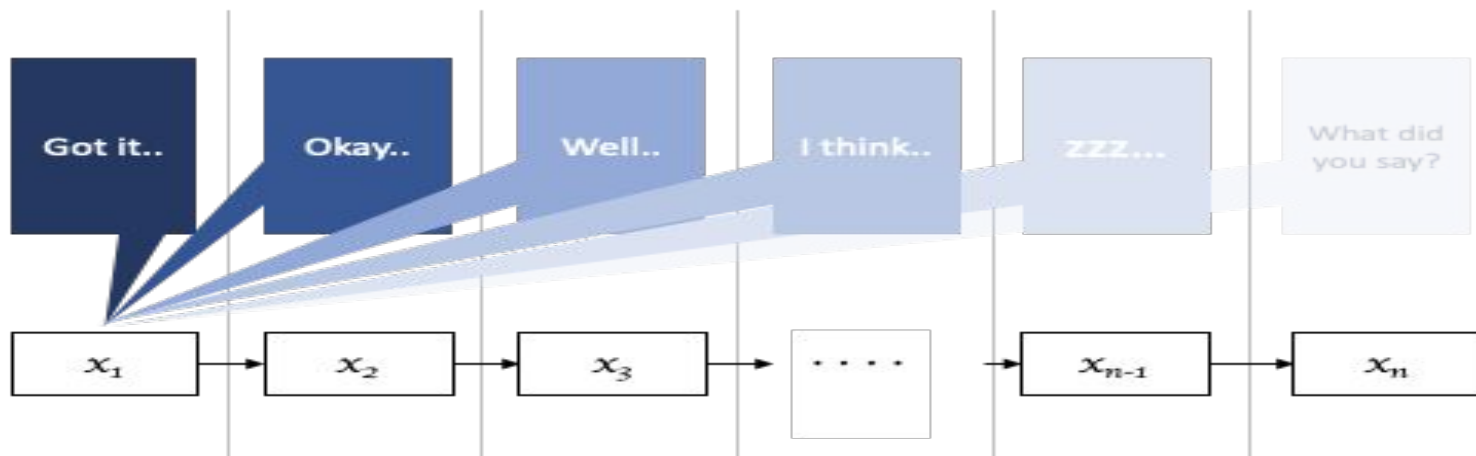
This is where LSTMs and GRUs helped in a big way by providing a way to carry only relevant information from one step to the next through various cell level innovations like forget gate, reset gate, update gate etc.


Bidirectional RNNs provided a mechanism to look at not just the prior but also the subsequent inputs before generating an output at a time step. Such developments addressed the “strictly sequential” problem.

Second Drawback

The longer the input sequence length (i.e. sentence length in NLP) the more difficult it is for the hidden vector to capture the context. This drawback makes sense intuitively; the more updates are made to the same vector, the higher the chances are the earlier inputs and updates are lost.

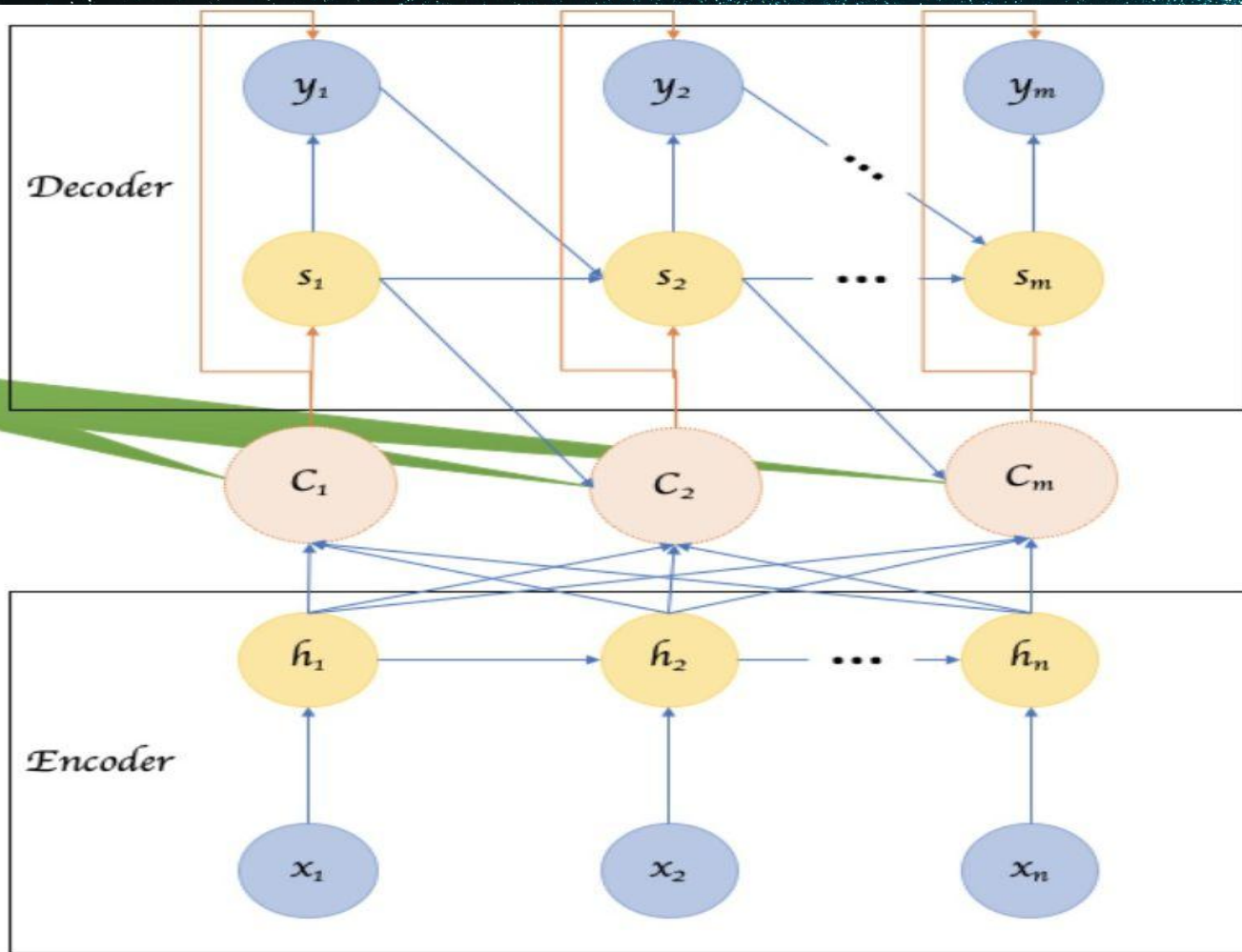
Influence of x_1 weakens in hidden state vector as it gets updated over and over in longer sequences...





**Attention
to the rescue**

Well, then
let's look at
everything



The Solution

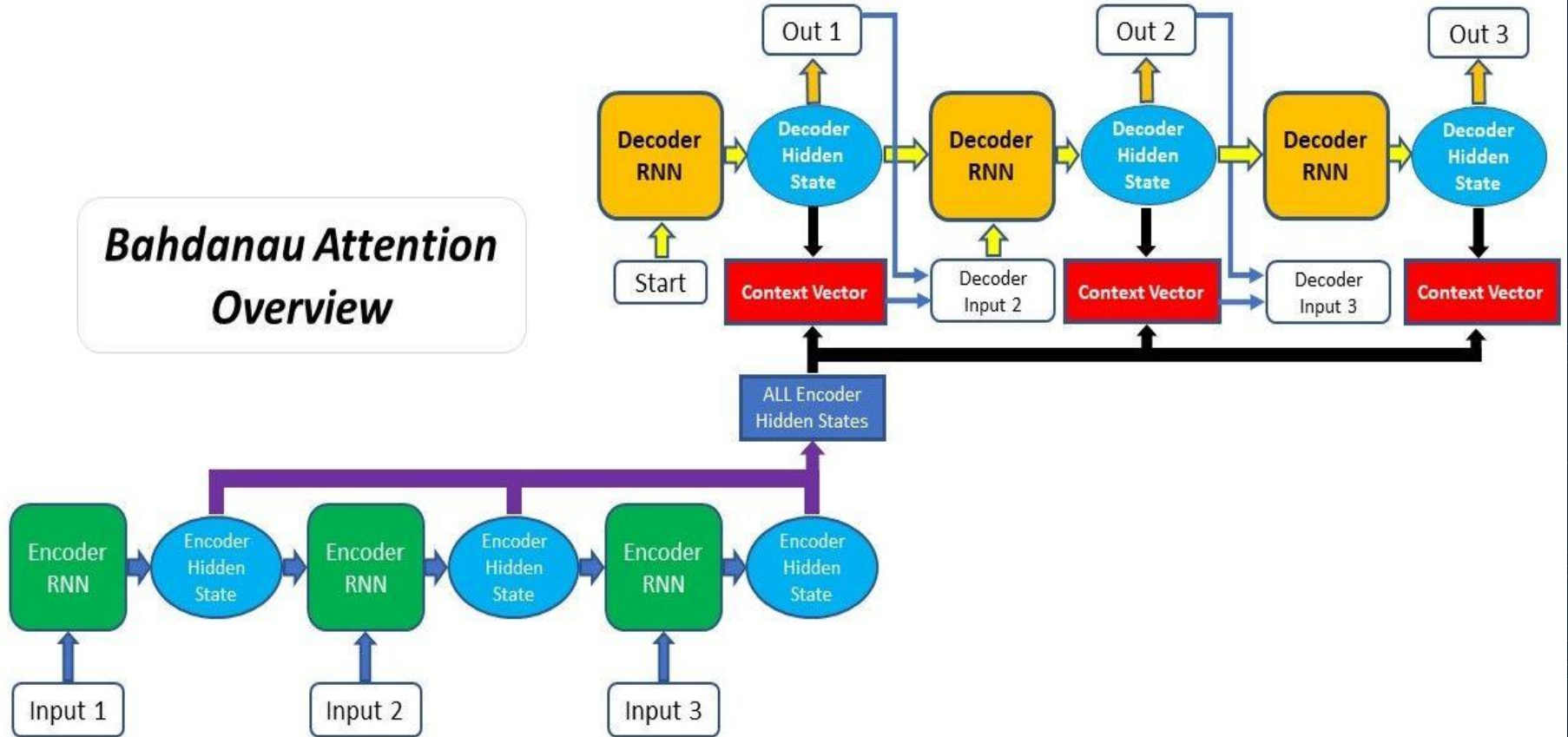
Perhaps if we get rid of using just the last hidden state as a proxy for the entire sentence and instead build an architecture that consumes all hidden states, then we won't have to deal with the weakening context.

In the proposed model, each generated output word is not just a function of just the final hidden state but rather a function of ALL hidden states. And, it's not just a simple operation that combines all hidden state — if it was, then we are still giving the same context to every output step, so it has to be different!

It is not a simple concatenation or dot product, but an “attention” operation that, for every decoder output step, produces a distinct vector representing all encoder hidden states but giving different weights to different encoder hidden state.

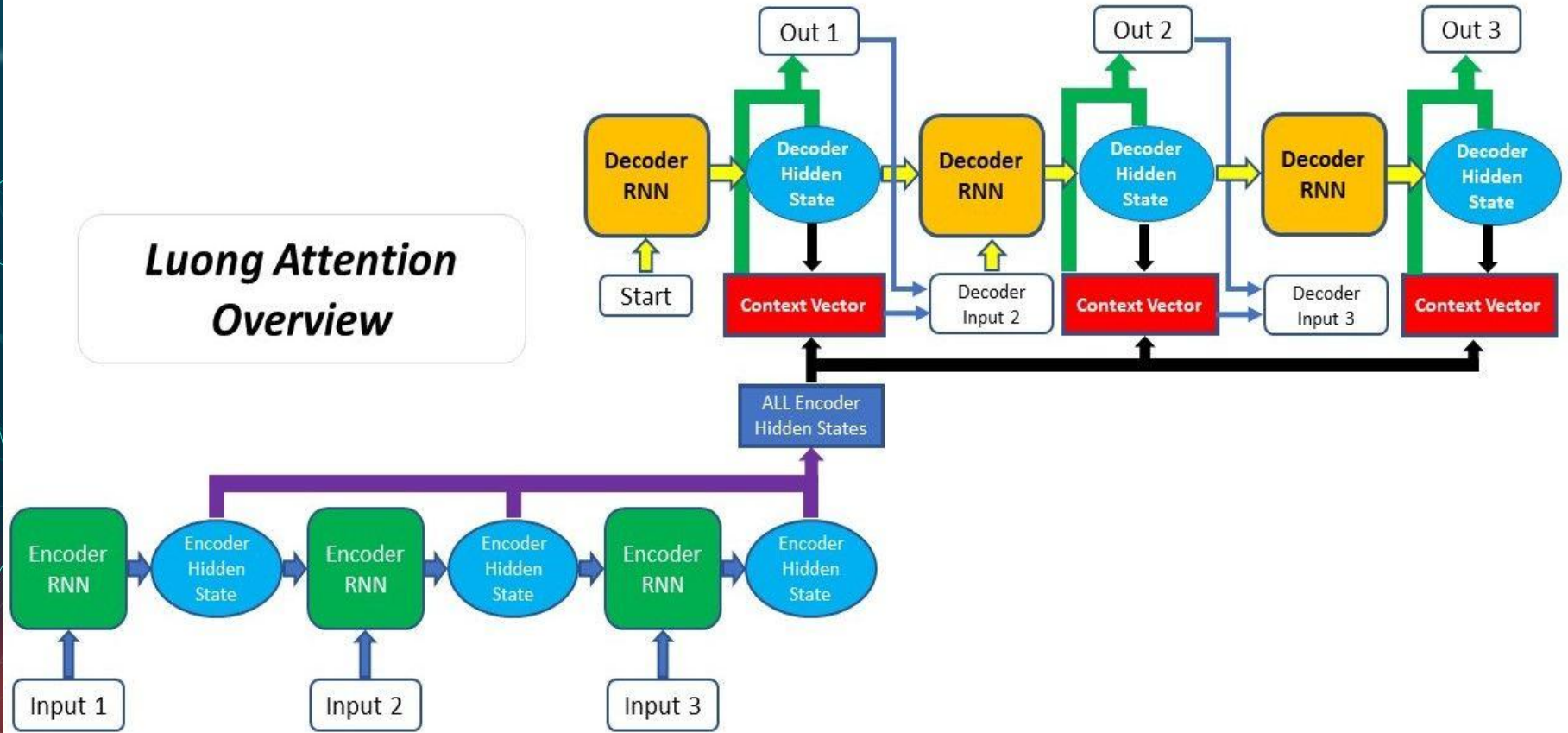
Bahdanau Attention

Bahdanau Attention Overview



Luong Attention

Luong Attention Overview



Differences between the two

Bahdanau

It is commonly referred to as Additive Attention

The major steps in this process are -

- Producing the Encoder Hidden States - Encoder produces hidden states of each element in the input sequence.
- Calculating the Context Vector
- Decoding the Output - the context vector is concatenated with the previous decoder output and fed into the Decoder RNN for that time step along with the previous decoder hidden state to produce a new output

Luong

It is commonly referred to as Multiplicative Attention

The major steps in this process are -

- Producing the Encoder Hidden States - Encoder produces hidden states of each element in the input sequence
- Decoder RNN - the previous decoder hidden state and decoder output is passed through the Decoder RNN to generate a new hidden state for that time step
- Calculating the Context Vector
- Producing the Final Output - the context vector is concatenated with the decoder hidden state and passed through a fully connected layer to produce a new output

Context vector computed as weighted sum of all encoder hidden state vectors

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$

The attention weight is computed as Softmax of alignment scores

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$$

And the alignment score "e" is a function of prior decoder hidden state and encoder hidden state

$$e_{ij} = a(s_{i-1}, h_j)$$

The alignment function "a", in this paper was

a = feed forward network jointly trained with the model



Alignment score calculation

- Bahdanau

$$score_{alignment} = W_{combined} \cdot \tanh(W_{decoder} \cdot H_{decoder} + W_{encoder} \cdot H_{encoder})$$

- Luong

Dot

$$score_{alignment} = H_{encoder} \cdot H_{decoder}$$

General

$$score_{alignment} = W(H_{encoder} \cdot H_{decoder})$$

Concat

$$score_{alignment} = W \cdot \tanh(W_{combined}(H_{encoder} + H_{decoder}))$$

Drawbacks

while this solution seems to have addressed the problem of single context vector, it has made the model really big. There are a lot of computations involved when you try to prepare a separate context vector for every output step.

In addition, there is yet another problem with computational complexity that wasn't introduced by this solution, but existed even in the basic RNN.

Given the sequential nature of the operations, if the input sequence is of length "n", it requires "n" sequential operations to arrive at the final hidden state (i.e. calculate h_1 , h_2 etc till h_n). We cannot perform these operations in parallel as h_1 is a prerequisite to calculate h_2 .

This lack of parallelization within a sequence cannot be offset by adding more samples within a training batch either, as loading and optimizing weights for different samples increases memory needs which will limit the number of samples that can be used.

**The solution ?
Self-Attention**

The intuition

When thinking about sequential processing, one might wonder, given that we are replacing the one final hidden state with a context vector generated for every output step — do we need the “ h ” states at all?

After all, attention alignment is supposed to define which part of the input the given output step should focus on, and “ h ” is only an indirect representation of “ x ”. It represents the context of all input steps until “ x ” and not just “ x ” alone.

Wouldn't using “ x ” directly make more sense?

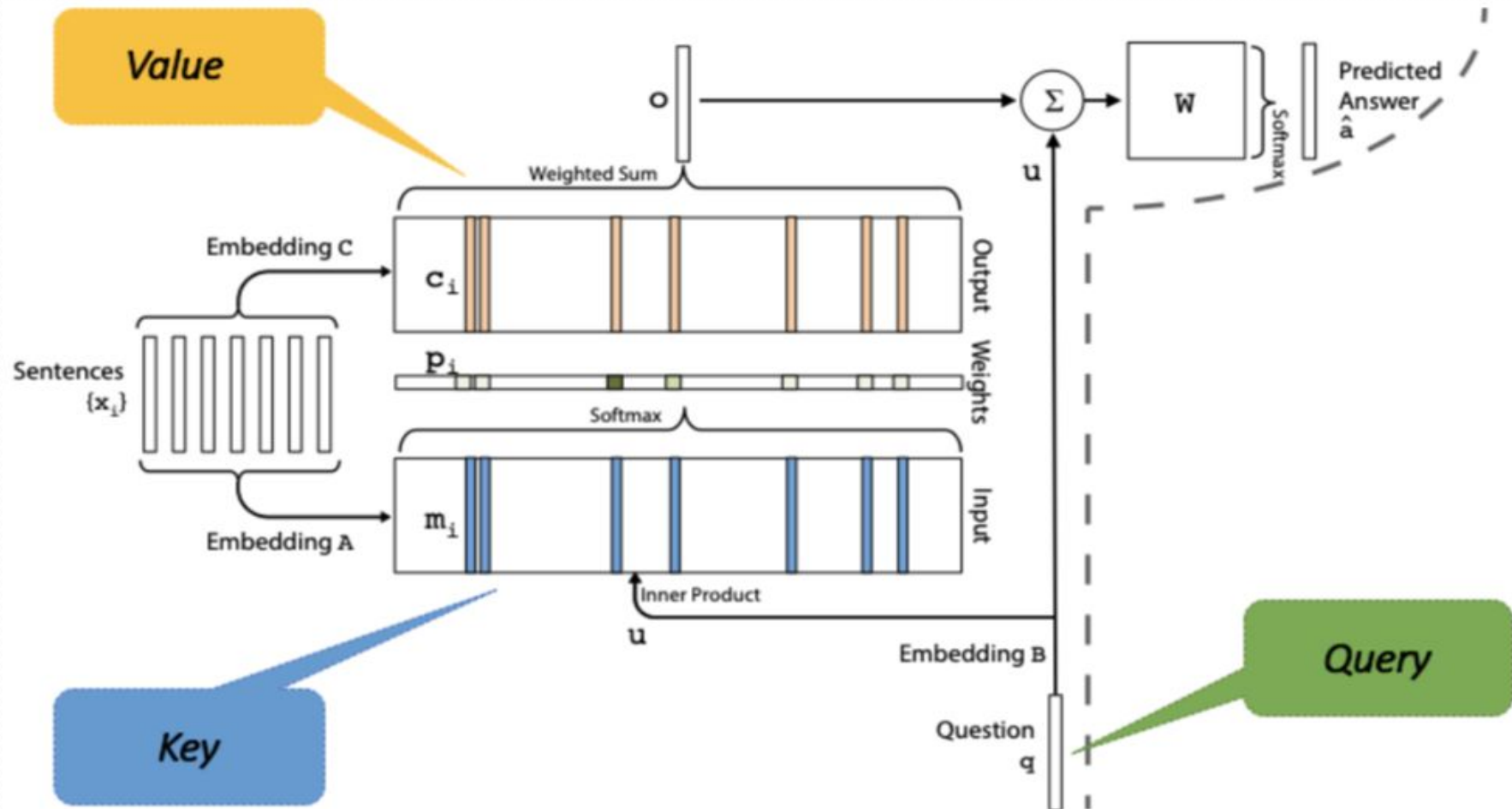
Turns out it does.

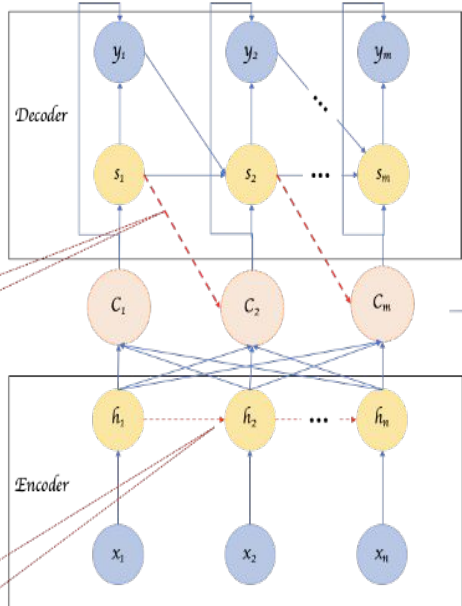
End to End Memory Networks

The proposed model has “input memory” or “key” vectors representing all inputs, a “query” vector to which the model needs to respond to (like the previous decoder hidden state) and “value” or “output memory” vectors — again a representation of the inputs.

The inner product between “query” and “keys” give the “match” (akin to attention) probability. The sum of “value” vectors weighted by the probability gives the final response.

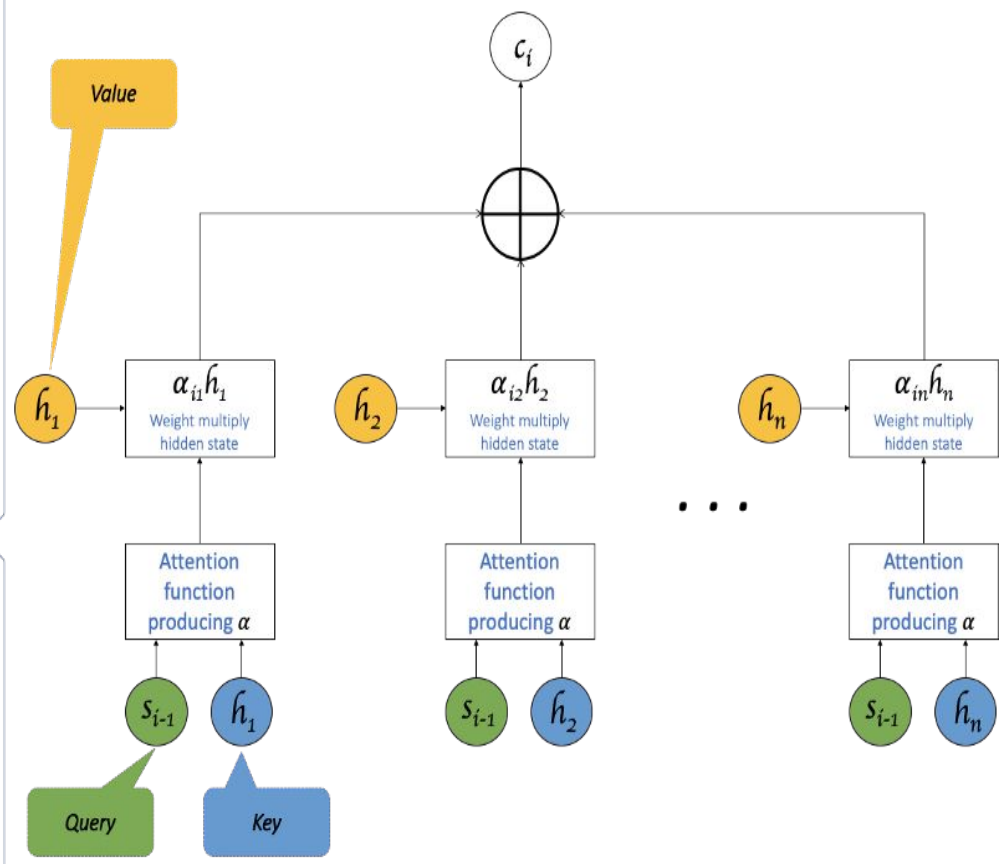
While producing good results, this eliminated sequential processing of the inputs and replaced it with a “memory query” paradigm.





2 Figure out a way to eliminate this sequential nature of the operation.

1 Figure out a way to eliminate this sequential nature of the operation.



The problems

The above figure highlights the two challenges we would like to resolve.

For challenge 1, we could perhaps just replace the hidden state (h) acting as keys with the inputs (x) directly. But this wouldn't be a rich representation - if we directly use word embeddings. The end-to-end memory network used different embedding matrices for input and output memory representations, which is better but they are still independent representations of the word.

Compare this with the hidden state (h) which represents not just the word, but the word in context of the given sentence.

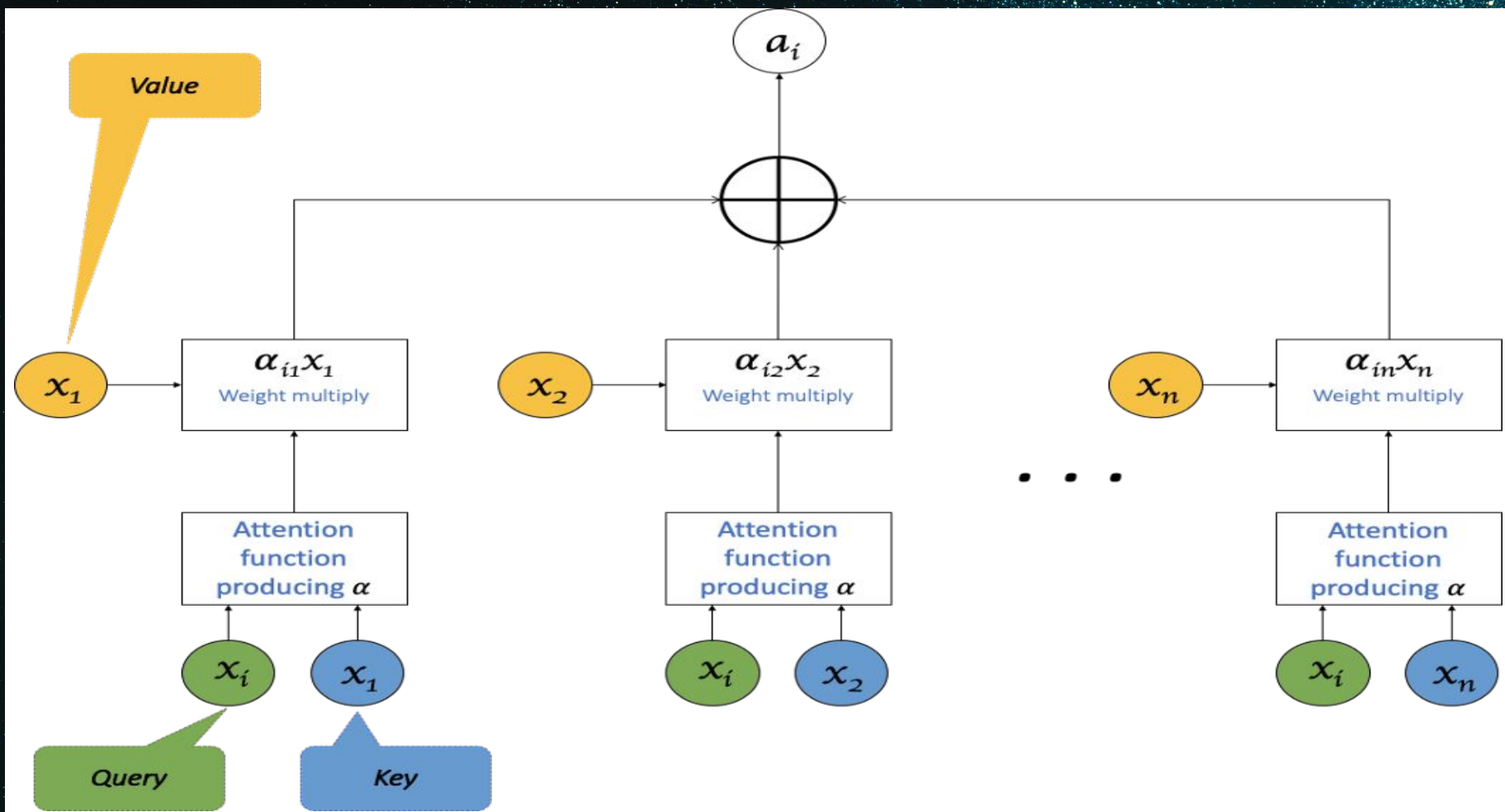
Is there a way to eliminate the sequential nature of generating hidden states, but still produce a richer, context representing vector?

The solution

What if, instead of using attention to connect encoder and decoder, we use attention within encoder and decoder respectively?

Attention, after all, is a rich representation — as it considers all keys and values.

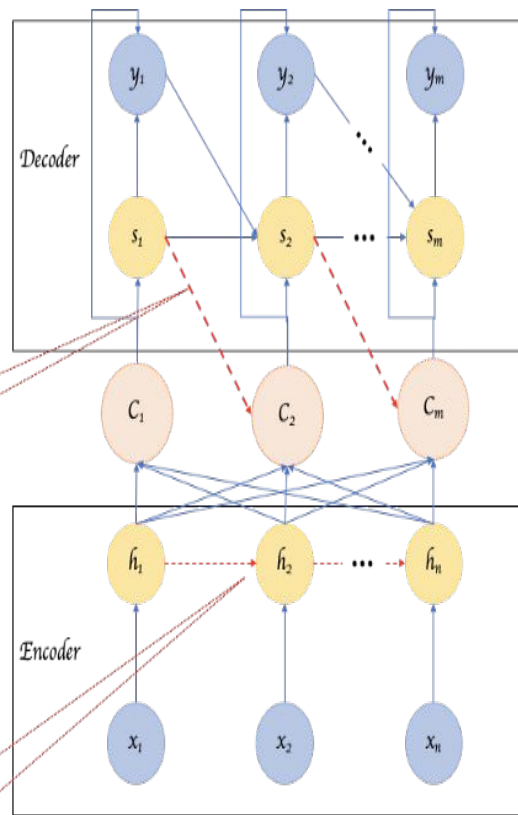
So instead of deriving hidden states using a RNN, we can use an attention based replacement where inputs (x) are used as “Keys” and “Values”. i.e. “ h ”s are replaced by “ x ”s.



Culminating towards the final structure

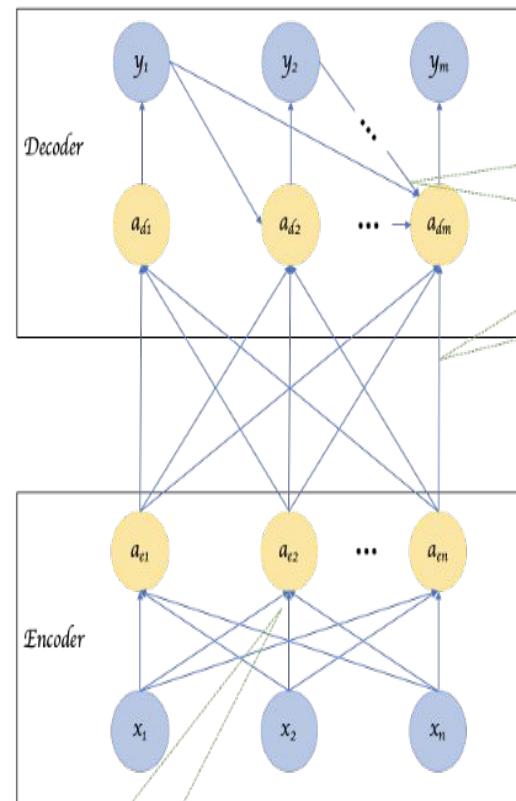
On the encoder side, we can use self attention to generate a richer representation of a given input step x_i , with respect to all other items in the input $x_1, x_2 \dots x_n$. This can be done for all input steps in parallel, unlike hidden state generation in a RNN based encoder. We are basically moving the criss-crossed lines on the left half of the below figure downwards as seen in the right half, thereby eliminating the dashed red lines between the representative vectors.

On the decoder side, we can do something similar. We replace a RNN based decoder to an attention based decoder. i.e. there are no hidden states anymore and no computation of a separate context vector for every decoder step. Instead, we do self attention on all outputs generated so far and along with it consume the entirety of encoder output. In other words, we are applying attention to whatever we know so far.



2 Figure out a way to eliminate this sequential nature of the operation.

1 Figure out a way to eliminate this sequential nature of the operation.



2 All encoder outputs (self attention on all input steps) are used along with decoder outputs until now (self attention on all generated outputs till current decoder step)

1 Self Attention replaces sequential hidden states.

**So everything resolved.
What about encodings ?**

Lost sense of Order

While getting rid of the sequential nature was helpful in many ways, like resulting in massive speed-up in the training time and ability to capture longer dependencies in a sentence.

It took off one key advantage — of knowing the order of words in the input sequence. Without it, the same word occurring in different positions within the same sentence might end up with the same output representation (since it will have the same key, value etc).

One possible solution to give the model some sense of order is to add a piece of information to each word about its position in the sentence. We call this “piece of information”, the positional encoding.

Possible ways

The first idea that might come to mind is to assign a number to each time-step within the $[0, 1]$ range in which 0 means the first word and 1 is the last time-step. One of the problems it will introduce is that you can't figure out how many words are present within a specific range. In other words, time-step delta doesn't have consistent meaning across different sentences.

Another idea is to assign a number to each time-step linearly. That is, the first word is given "1", the second word is given "2", and so on. The problem with this approach is that not only the values could get quite large, but also our model can face sentences longer than the ones in training. In addition, our model may not see any sample with one specific length which would hurt generalization of our model.

Ideally, the following criteria should be satisfied:

1. It should output a unique encoding for each time-step (word's position in a sentence)
2. Distance between any two time-steps should be consistent across sentences with different lengths.
3. Our model should generalize to longer sentences without any efforts. Its values should be bounded.

Positional Encoding

The encoding proposed satisfies all of the above criteria. Further, It has 2 important features -

1. First of all, it isn't a single number. Instead, it's a d -dimensional vector that contains information about a specific position in a sentence.
2. Secondly, this encoding is not integrated into the model itself. Instead, this vector is used to equip each word with information about its position in a sentence. In other words, we enhance the model's input to inject the order of words.

Let t be the desired position in an input sentence, $\vec{p}_t \in \mathbb{R}^d$ be its corresponding encoding, and d be the encoding dimension. Then $f : \mathbb{N} \rightarrow \mathbb{R}^d$ will be the function that produces the output vector \vec{p}_t and it is defined as follows:

$$\vec{p}_t^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k \cdot t), & \text{if } i = 2k \\ \cos(\omega_k \cdot t), & \text{if } i = 2k + 1 \end{cases}$$

where

$$\omega_k = \frac{1}{10000^{2k/d}}$$

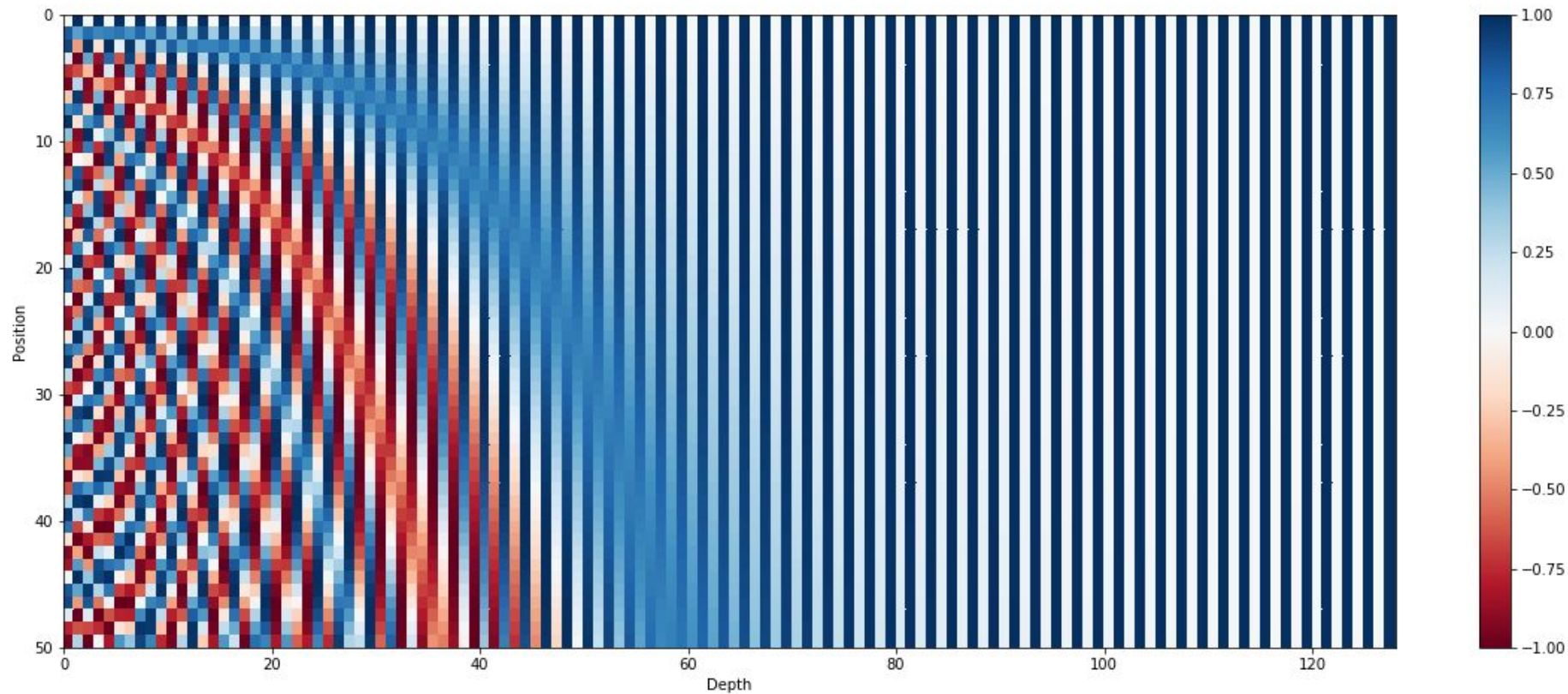
How does this even work ?

You may wonder how this combination of sines and cosines could ever represent a position/order? It is actually quite simple, Suppose you want to represent a number in binary format, how will that be?

You can spot the rate of change between different bits. The LSB bit is alternating on every number, the second-lowest bit is rotating on every two numbers, and so on.

But using binary values would be a waste of space in the world of floats. So instead, we can use their float continuous counterparts - Sinusoidal functions. Indeed, they are the equivalent to alternating bits.

0 :	0	0	0	0	8 :	1	0	0	0
1 :	0	0	0	1	9 :	1	0	0	1
2 :	0	0	1	0	2 :	1	0	1	0
3 :	0	0	1	1	11 :	1	0	1	1
4 :	0	1	0	0	12 :	1	1	0	0
5 :	0	1	0	1	13 :	1	1	0	1
6 :	0	1	1	0	14 :	1	1	1	0
7 :	0	1	1	1	15 :	1	1	1	1



128-dimensional positional encoding for a sentence with the maximum length of 50.
Each row represents the embedding vector.

Applying Positional Encoding

Positional embeddings are used to equip the input words with their positional information. But how is it done? In fact, the original paper added the positional encoding on top of the actual embeddings. That is for every word w_t in a sentence $[w_1, \dots, w_n]$, Calculating the corresponding embedding which is fed to the model is as follows:

$$\psi'(w_t) = \psi(w_t) + \vec{p}_t$$

To make this summation possible, we keep the positional embeddings' dimension equal to the word embeddings' dimension i.e.

$$d_{\text{word embedding}} = d_{\text{positional embedding}}$$

Relative Positioning

Another characteristic of sinusoidal positional encoding is that it allows the model to attend relative positions effortlessly. Here is a quote from the original paper:

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} .

But why does this statement hold ? To prove this, we consider -

For every sine-cosine pair corresponding to frequency ω_k , there is a linear transformation $M \in \mathbb{R}^{2 \times 2}$ (independent of t) where the following equation holds:

$$M \cdot \begin{bmatrix} \sin(\omega_k \cdot t) \\ \cos(\omega_k \cdot t) \end{bmatrix} = \begin{bmatrix} \sin(\omega_k \cdot (t + \phi)) \\ \cos(\omega_k \cdot (t + \phi)) \end{bmatrix}$$

...continued

Solving the previous equation we get -

$$M_{\phi,k} = \begin{bmatrix} \cos(\omega_k \cdot \phi) & \sin(\omega_k \cdot \phi) \\ -\sin(\omega_k \cdot \phi) & \cos(\omega_k \cdot \phi) \end{bmatrix}$$

As you can see, the final transformation does not depend on t . Note that one can find the matrix M very similar to the rotation matrix.

Similarly, we can find M for other sine-cosine pairs, which eventually allows us to represent $p_{t+\phi}$ as a linear function of p_t for any fixed offset ϕ . This property, makes it easy for the model to learn to attend by relative positions.



Putting everything together, The Transformer

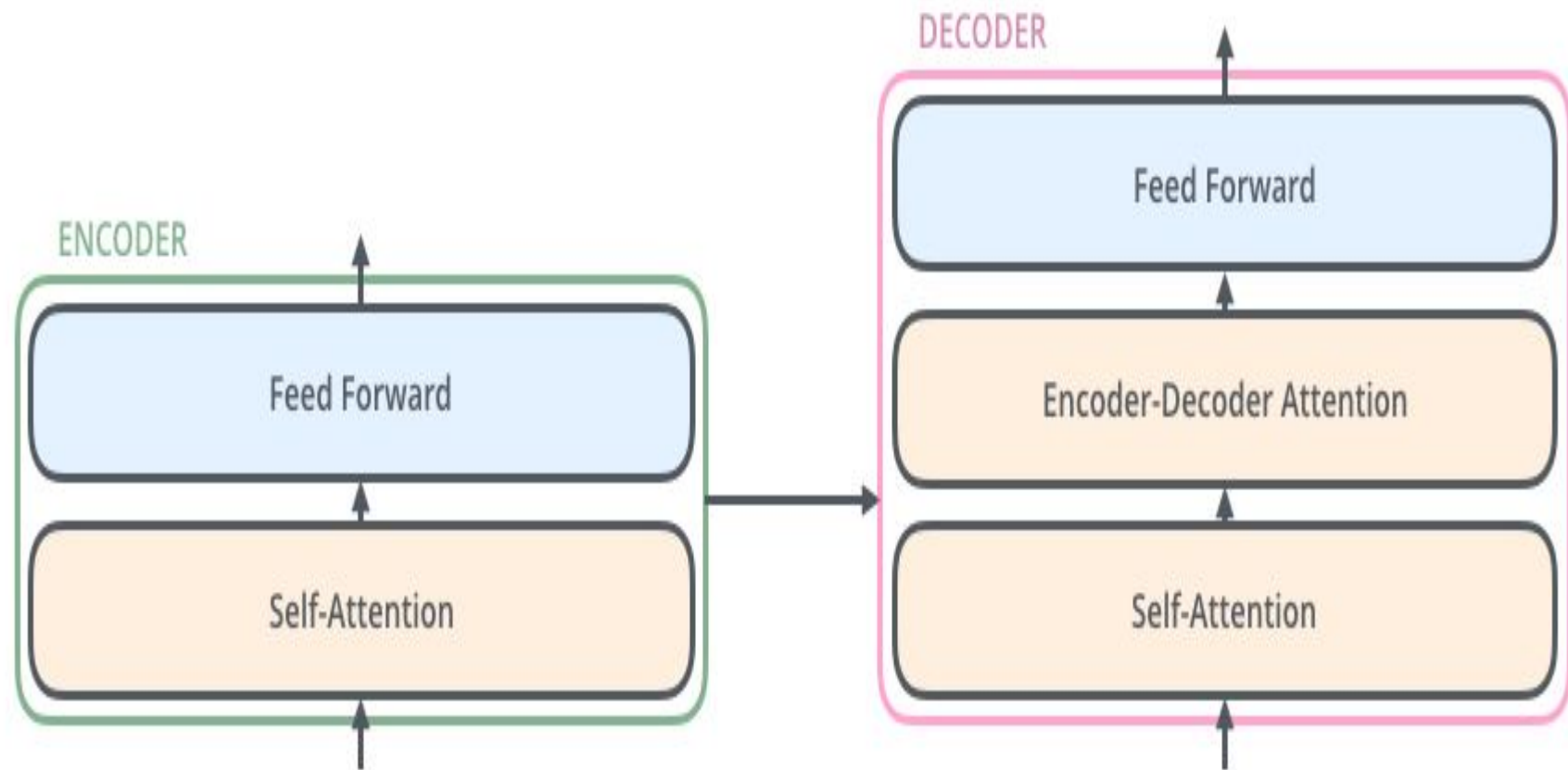
The raw structure

The encoding component is a stack of encoders (the paper stacks $N=6$ of them on top of each other). The decoding component is a stack of decoders of the same number.

The encoder inputs first flow through a self-attention layer – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word.

The outputs of the self-attention layer are fed to a feed-forward neural network. The exact same feed-forward network is independently applied to each position.

The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence (similar to what attention does in seq2seq models).

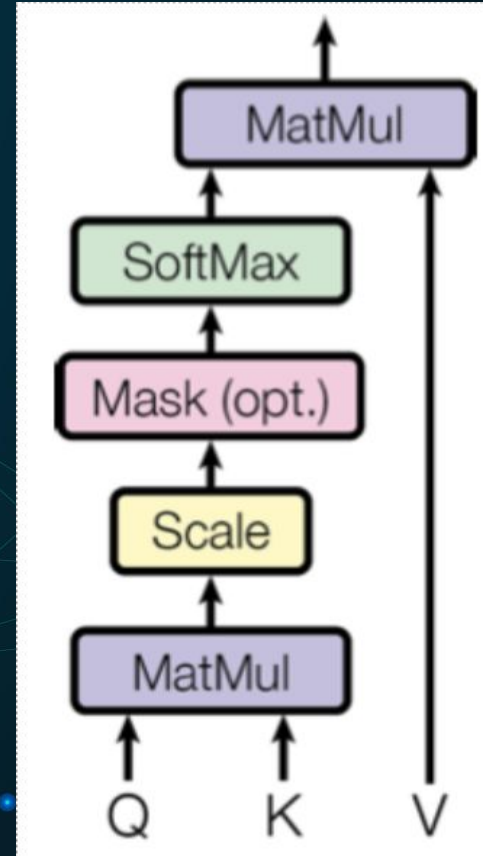


Scaled Dot Product Attention

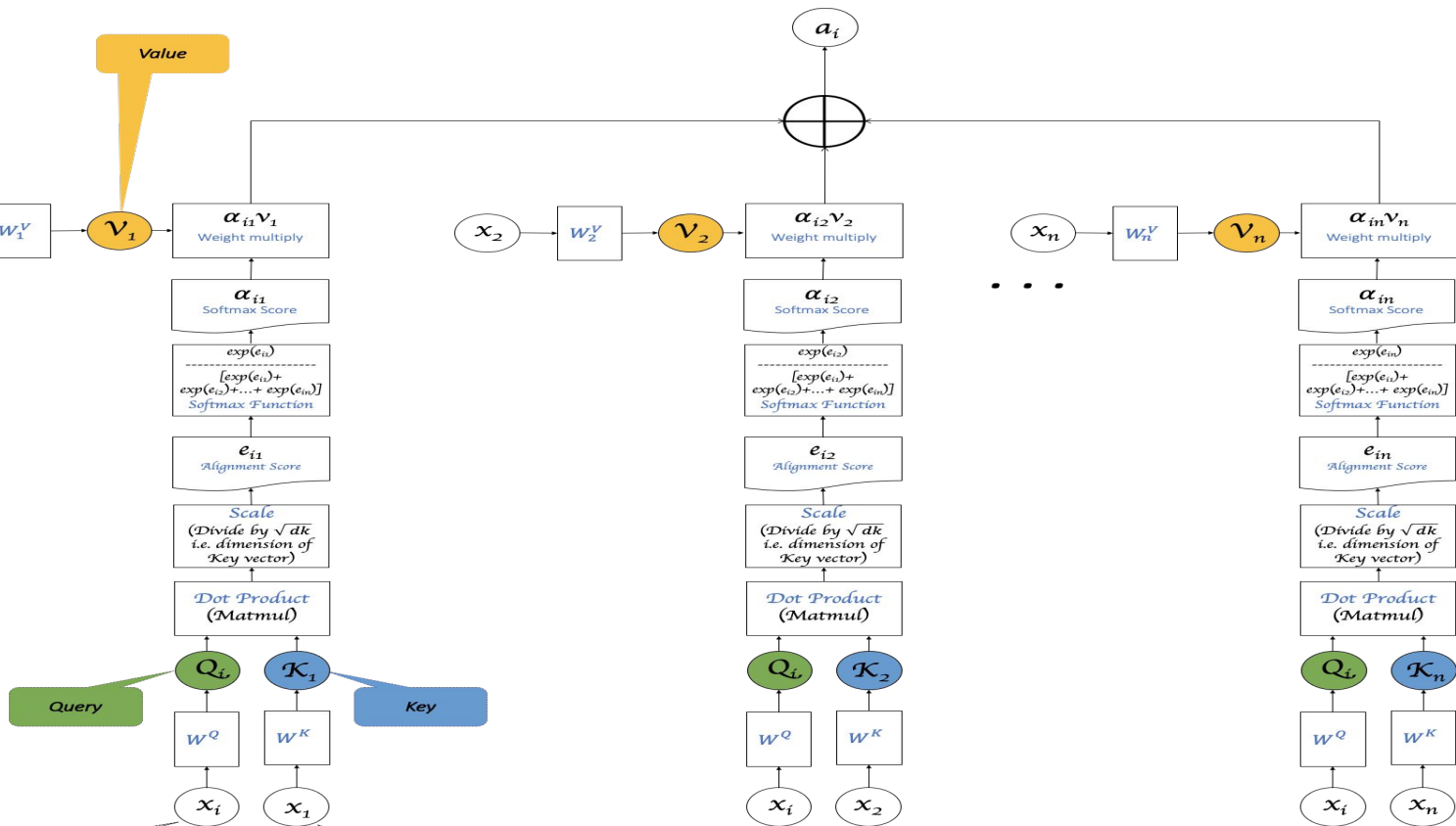
The input consists of queries and keys of dimension d_k , and values of dimension d_v . We compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix Q . The keys and values are also packed together into matrices K and V . We compute the matrix of outputs as:

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



1. Use x (i.e. encoder inputs) as shown for encoder self attention
2. Use y (i.e. decoder outputs) for decoder self attention
3. Use a_i (i.e. encoder outputs) for encoder-decoder attention



1. Use x (i.e. encoder inputs) as shown for encoder self attention
2. Use y (i.e. decoder outputs) for decoder self attention
3. Use a_i (i.e. decoder self attention outputs) for encoder-decoder attention

1. Use x (i.e. encoder inputs) as shown for encoder self attention
2. Use y (i.e. decoder outputs) for decoder self attention
3. Use a_i (i.e. encoder outputs) for encoder-decoder attention

- In "encoder-decoder attention" layers, the queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence.
- The encoder contains self-attention layers. In a self-attention layer all of the keys, values and queries come from the same place, in this case, the output of the previous layer in the encoder. Each position in the encoder can attend to all positions in the previous layer of the encoder.
- Similarly, self-attention layers in the decoder allow each position in the decoder to attend to all positions in the decoder up to and including that position. We need to prevent leftward information flow in the decoder to preserve the autoregressive property. We implement this inside of scaled dot-product attention by masking out (setting to $-\infty$) all values in the input of the softmax which correspond to illegal connections

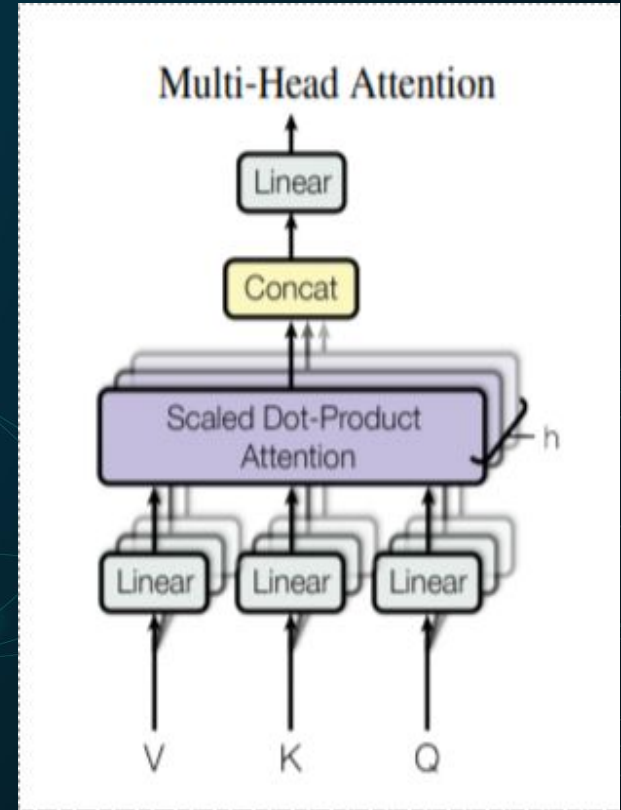
Multi-Head Attention

Instead of performing a single attention function with d_{model} -dimensional keys, values and queries, It was found beneficial to linearly project the queries, keys and values h times with different, learned linear projections to d_k , d_k and d_v dimensions, respectively.

On each of these projected versions of queries, keys and values, we then perform the attention function in parallel, yielding d_v -dimensional output values. These are concatenated and once again projected, resulting in the final values.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$



Advantages

It expands the model's ability to focus on different positions. The scaled dot product attention contains a little bit of every other encoding, but it could be dominated by the the actual word itself.

It gives the attention layer multiple “representation subspaces”. We have not only one, but multiple sets of Query/Key/Value weight matrices (the Transformer uses eight attention heads, so we end up with eight sets for each encoder/decoder).

Each of these sets is randomly initialized. Then, after training, each set is used to project the input embeddings (or vectors from lower encoders/decoders) into a different representation subspace akin to using different filters to create different features maps in a single layer in a CNN.

Positionwise FFNN, SubLayers and Residual connection

In addition to attention sub-layers, each of the layers in our encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between.

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

Each sub-layer (self-attention, ffnn) in each encoder has a residual connection around it, and is followed by a layer-normalization step. I.e. the output of each sub-layer is $\text{LayerNorm}(x + \text{Sublayer}(x))$ where $\text{Sublayer}(x)$ is the function implemented by the sub-layer itself consisting of positionwise ffnn and self attention layer.

Encoder and Decoder Stacks

Encoder:

The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position wise fully connected feed-forward network. We employ a residual connection around each of the two sub-layers, followed by layer normalization.. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Decoder:

The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i .

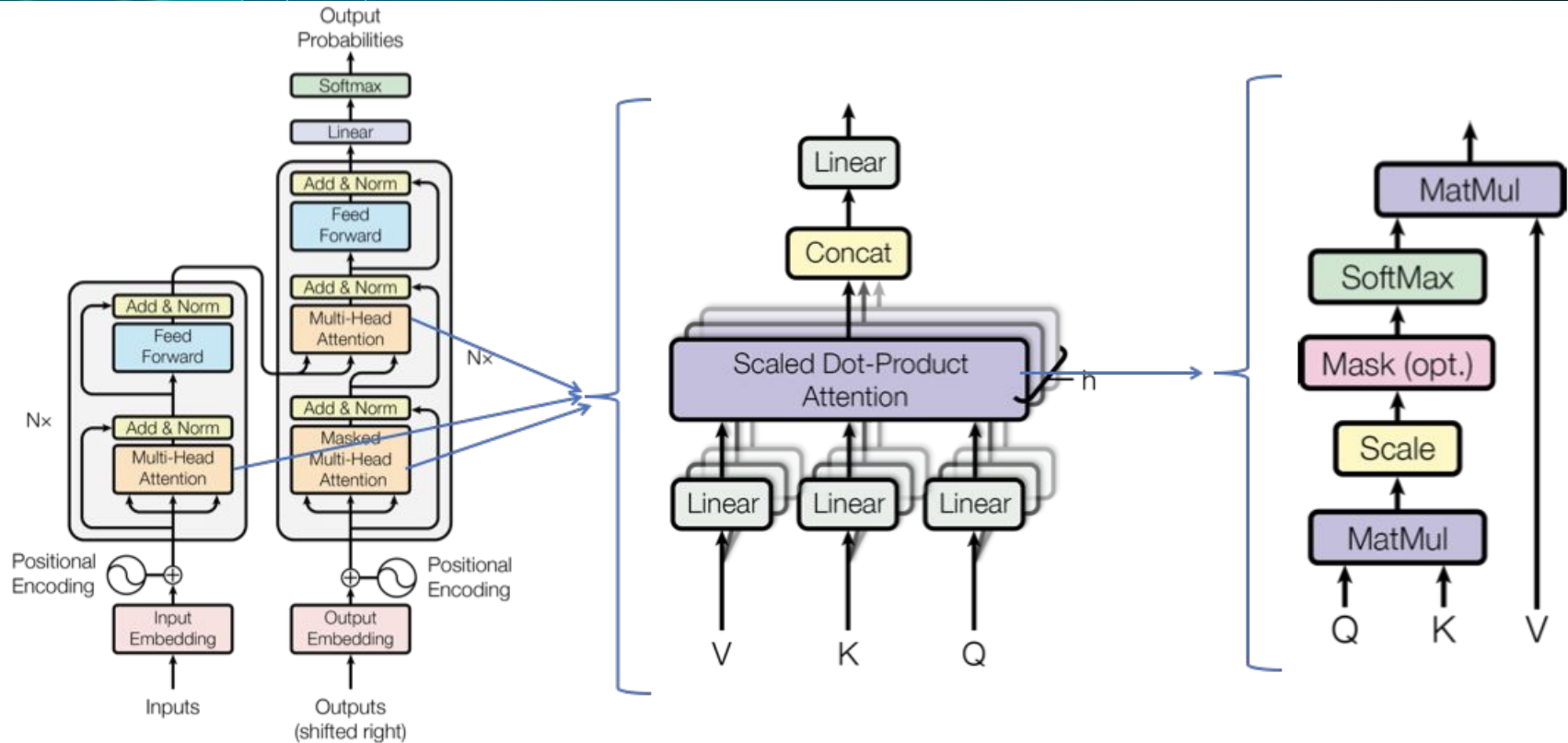
Final Linear and Softmax Layer

The decoder stack outputs a vector of floats. How do we turn that into a word? That's the job of the final Linear layer which is followed by a Softmax Layer.

The 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 a logits vector. Each cell in the logit vector corresponds to the score of a unique word.

The softmax layer then turns those scores into probabilities (all positive, all add up to 1.0). The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.

Complete Architecture



**Finally, Training and
getting Results**

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, 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. 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.

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, step time was 1.0 seconds. The big models were trained for 300,000 steps (3.5 days).

Optimizer -

We used the Adam optimizer 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:

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step_num^{-0.5}, step_num \cdot warmup_steps^{-1.5})$$

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.

Regularisation -

Residual Dropout

We apply dropout 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 $P_{\text{drop}} = 0.1$.

Label Smoothing

During training, we employed label smoothing of value $ls = 0.1$. This hurts perplexity, as the model learns to be more unsure, but improves accuracy and BLEU score.

Label Smoothing

Label smoothing replaces one-hot encoded label vector y_{hot} with a mixture of y_{hot} and the uniform distribution:

$$y_{\text{ls}} = (1 - \alpha) * y_{\text{hot}} + \alpha / K$$

where K is the number of label classes, and α is a hyperparameter that determines the amount of smoothing. If $\alpha = 0$, we obtain the original one-hot encoded y_{hot} . If $\alpha = 1$, we get the uniform distribution.

To reduce gradients using the cross entropy loss,

1. One-hot encoded labels encourages largest possible logit gaps to be fed into the softmax function. Intuitively, large logit gaps combined with the bounded gradient will make the model less adaptive and too confident about its predictions.
2. In contrast, smoothed labels encourages small logit gaps. It is shown in that this results in better model calibration and prevents overconfident predictions.

In the paper, label smoothing has been implemented using the KL div loss, instead of using a one-hot target distribution, we create a distribution that has confidence of the correct word and the rest of the smoothing mass distributed throughout the vocabulary.

BLEU 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 approach works by counting matching n-grams in the candidate translation to n-grams in the reference text, where 1-gram or unigram would be each token and a bigram comparison would be each word pair. The comparison is made regardless of word order.

Results

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	$2.3 \cdot 10^{19}$	

**The End,
Thank You**