Dealing with Sequences: Recurrent Neural Network (RNN) layer

For a function that takes sequence $\mathbf{x}^{(\mathbf{i})}$ as input and creates sequence $\mathbf{y}$ as output we had two choices for implementing the function.

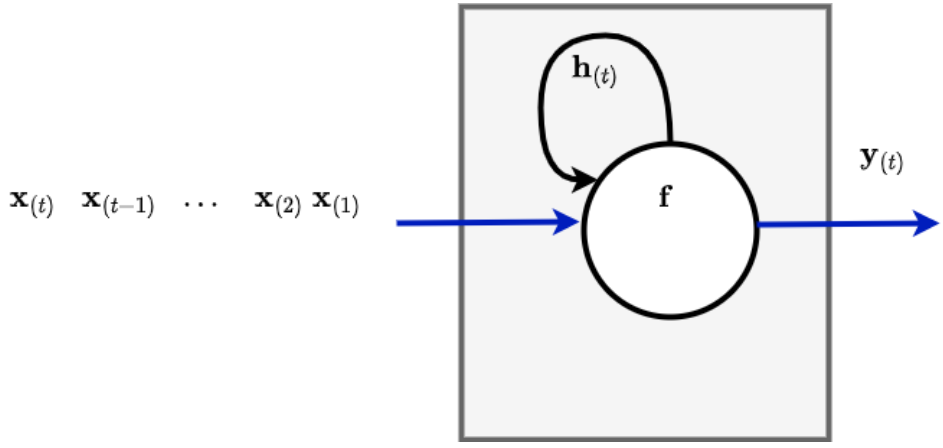
The RNN implements the function as a "loop"

- A function that taking **a single** $\mathbf{x}_{(t)}$ as input a time
- Outputting $\mathbf{y}_{(t)}$
- Using a "latent state" $\mathbf{h}_{(t)}$ to summarize the prefix $\mathbf{x}_{(1...t)}$
- Repeat in a loop over t

$p(\mathbf{h}_{(t)}|\mathbf{x}_{(t)}, \mathbf{h}_{(t-1)})$ latent variable $\mathbf{h}_{(t)}$ encodes $[\mathbf{x}_{(1)} \dots \mathbf{x}_{(t)}]$

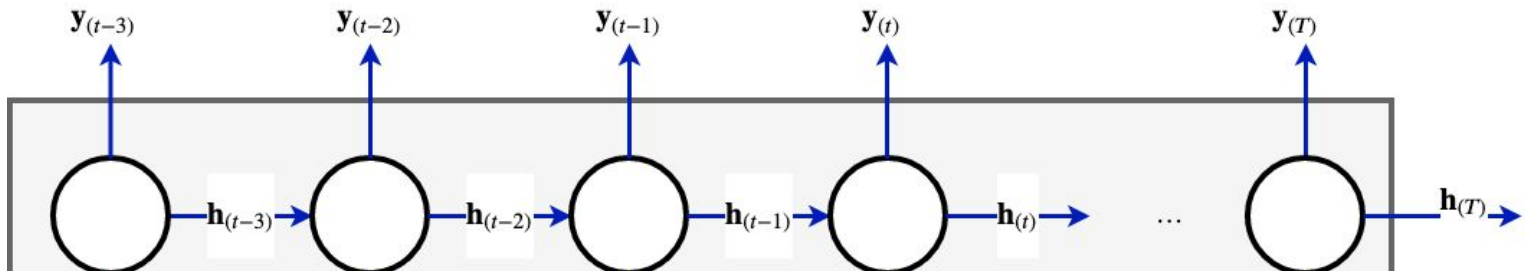
$p(\mathbf{y}_{(t)}|\mathbf{h}_{(t)})$ prediction contingent on latent variable

Loop with latent state



"Unrolling" the loop makes it equivalent to a multi-layer network

RNN unrolled

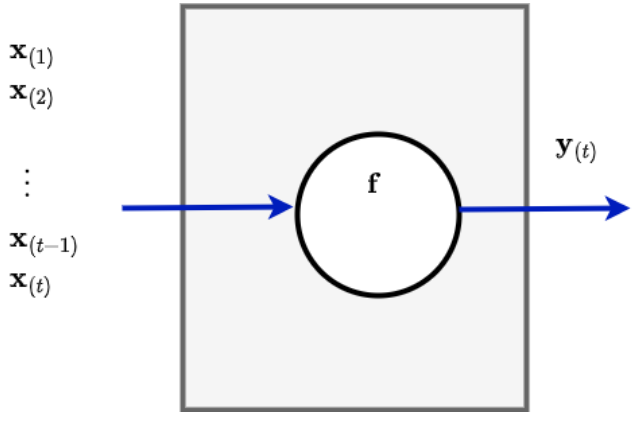

$\mathbf{y}_{(t-3)}$
$\mathbf{y}_{(t-2)}$
$\mathbf{y}_{(t-1)}$
$\mathbf{y}_{(t)}$
$\mathbf{y}_{(T)}$
$\mathbf{h}_{(t-3)}$
$\mathbf{h}_{(t-2)}$
$\mathbf{h}_{(t-1)}$
$\mathbf{h}_{(t)}$
...
$\mathbf{h}_{(T)}$

Transformer: Encoder-style

The alternative to the loop was to create a "direct function"

- Taking a **sequence** $\mathbf{x}_{(1\ldots t)}$ as input
- Outputting $\mathbf{y}_{(t)}$

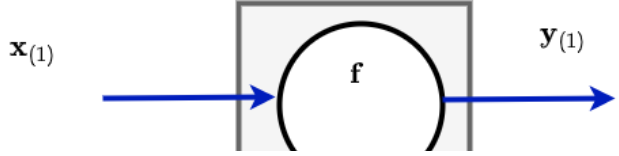
Direct function



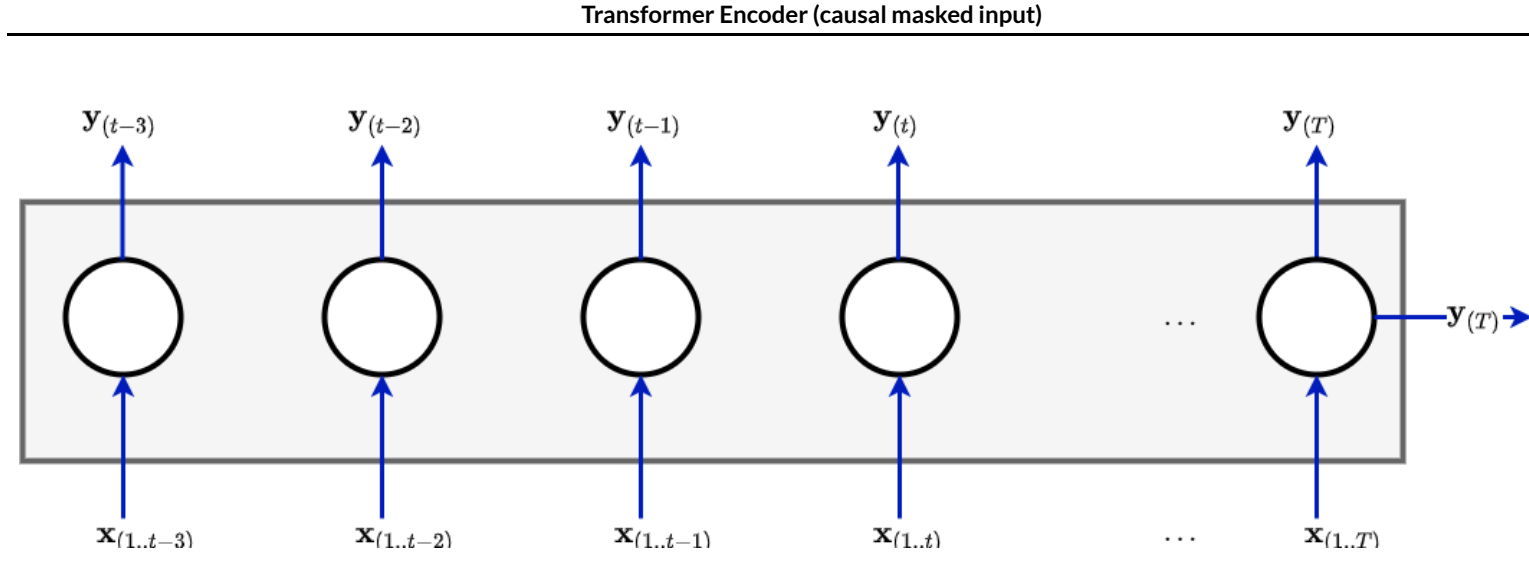
In order to output the sequence $\mathbf{y}_{(1)} \ldots \mathbf{y}_{(T)}$ we create T copies of the function (one for each $\mathbf{y}_{(t)}$)

- computes each $\mathbf{y}_{(t)}$ in **parallel**, not sequentially as in the loop

Direct function, in parallel (masked input)


$\mathbf{x}_{(1)}$
$\mathbf{f}$
$\mathbf{y}_{(1)}$

The parallel units constitute a *Transformer Encoder*



Compared to the unrolled RNN, the Transformer Decoder

- Takes a **sequence** $\mathbf{x}_{(1..t)}$ as input
    - Because $\mathbf{y}_{(t)}$ is computed as a *direct* function of the prefix $\mathbf{x}_{(1..t)}$ rather than recursively
- Has **no** latent state: output is a direct function of the input sequence
- Has **no** data (e.g., $\mathbf{h}_{(t)}$) passing from the computation between time steps (e.g., from t to $(t+1)$)
- Outputs generated in parallel, not sequentially
- No gradients flowing backward over time

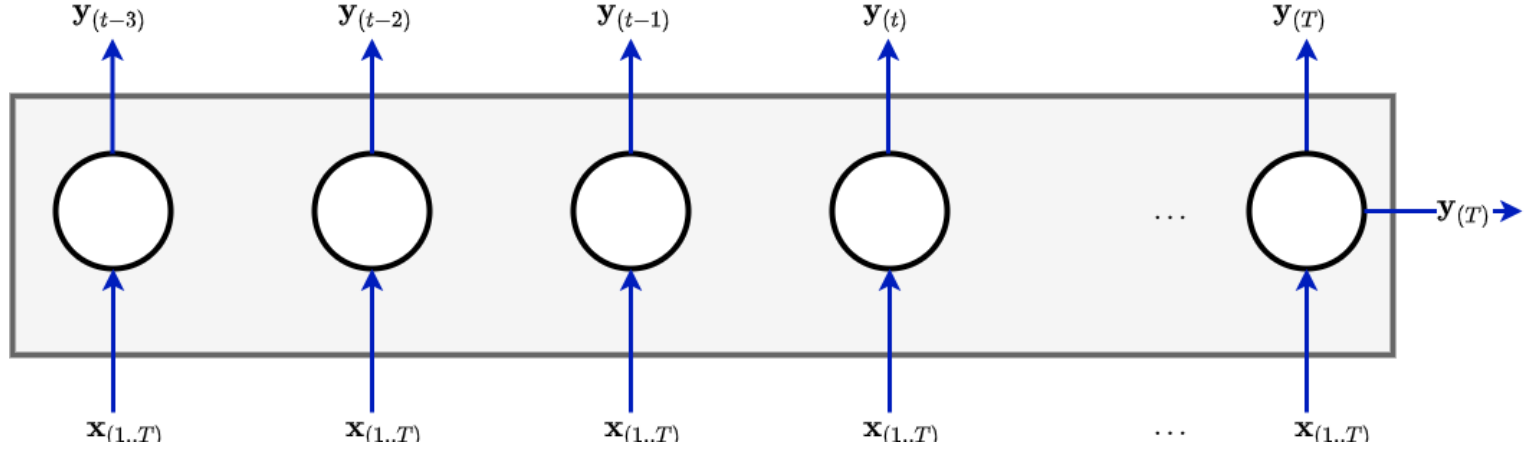
With this architecture, we can compute more general functions than the RNN

- where each $\mathbf{y}_{(t)}$ depends on the entire $\mathbf{x}_{(1...T)}$ rather than a prefix $\mathbf{x}_{(1...t)}$

Direct function, in parallel (un-masked input)


$\mathbf{x}_{(1\ldots T)}$
$\mathbf{y}_{(1)}$

Transformer Decoder (unmasked input)



An example of such a general function is the "meaning" of a word, in context.

Sentence	Meaning of "it"

The animal didn't cross the street because **it** was too tired | the animal The animal didn't cross the street because **it** was too wide | the road

The meaning of the word "it" is determined by a word that follows it ("tired" or "wide")

So even though the Transformer output at each position is a function of the entire sequence $\mathbf{x}_{(1...T)}$

- the output is different for each position t

We can control whether the input to Transformer element at position t is prefix $\mathbf{x}_{(1...t)}$ or the entire sequence $\mathbf{x}_{(1...T)}$ by **masking** the input to element t

- no masking: the entire sequence is visible
- *casual masking*: only the prefix up to t is visible: $\mathbf{x}_{(1...t)}$

Technical clarifications

Shared Transformer blocks across positions

The transformer blocks ("circles" in the diagram)

- are **shared** across all positions
- that is: the same computation (with shared parameters) is performed in parallel
- Thus, the number of parameters is **not** a function of sequence length T

Identifying $\mathbf{y}_{(t)}$ with $\mathbf{h}_{(t)}$

When we introduced the RNN, at each step t of the loop, we defined two outputs $\mathbf{h}_{(t)}$ and $\mathbf{y}_{(t)}$.

In general, we only need to output $\mathbf{h}_{(t)}$

- $\mathbf{y}_{(t)}$ can be defined as a further processing of $\mathbf{h}_{(t)}$

Henceforth, we will assume the style of a single output $\mathbf{h}_{(t)}$

The reason for doing this:

- We can "stack" N Transformer layers (just as we can stack RNN layers)
- The output of the non-top layer j is $\mathbf{h}_{(t)}^{[j]}$, not the final $\mathbf{y}_{(t)}$
- We identify $\mathbf{y}_{(t)}$ as the output of the top layer $\mathbf{h}_{(t)}^{[N]}$
    - perhaps after a further processing

Furthermore:

Since the Encoder part is no longer a "loop"

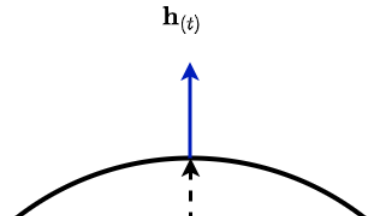
- It is inaccurate to refer to the Encoder output $\bar{\mathbf{h}}_{(t)}$ as a "latent" state
- However, $\bar{\mathbf{h}}_{(t)}$ *is still* a summary of the input sequence
    - a summary of $\mathbf{x}_{(1\ldots t)}$ when casual attention is used
    - a summary of $\mathbf{x}_{(1\ldots \bar{T})}$ otherwise
- Out of **bad habit** we may continue to erroneously refer to $\bar{\mathbf{h}}$ and $\mathbf{h}$ as "latent" states

Inside the Transformer Encoder: Self Attention

If we look inside the box computing the direct function, we will find several layers

- An Attention layer
    - To influence which elements of the input sequence $\mathbf{x}$ to attend/focus when outputting $\mathbf{y}_{(t)}$
- A Feed Forward Network (FF) layer to compute the function

Transformer Layer (Encoder)


$\mathbf{h}_{(t)}$

An Attention layer that attends/focus on its inputs implements what is called *Self-attention*

- We will soon see the possibility of attending to other values

If the function for $\mathbf{h}_{(t)}$ is restricted to prefix $\mathbf{x}_{(1...t)}$ the Attention layer can use causal masking of the input sequence.

This is referred to as *Masked Self-Attention*.

The Feed Forward Network computes the function, given the elements of the input sequence that are being attend to.

Advantages of a Transformer compared to an RNN

As we will demonstrate in detail below

- The Transformer's operations can be performed in parallel versus sequentially for the RNN
- Gradients less likely to vanish or explode

We can **leverage** these advantages in complexity by

- By making a Transformer model bigger (e.g., more stacked Transformer layers)
- Making the sequence lengths longer
- Increasing the number of examples of training data

So, for the same time "cost" as an RNN, we can use a bigger Transformer on more data

- **Hence: we can learn more complex functions for similar time cost**

Moreover: the path length from the output to the input is constant in an Transformer, compared to T in the RNN.

Recall that gradients tend to vanish or explode as the path length during Back Propagation increases.

So **Transformers are better able to capture long-range dependencies than an RNN**

This gives them an advantage in learning as well.

The price we pay for this is that the number of parameters of a Transformer is greater than a similar RNN

- By a factor of T
- The parameters for each time step of a Transformer are *independent*
- The parameters for each time step of an RNN are *shared*

We give the detailed math below.

Number of sequential steps

The most obvious advantage of the "direct function" as opposed to the "loop" is that outputs are computed in parallel versus sequentially.

For an input sequence of length T:

- The loop requires T steps
- The direct function requires 1 step

Path length

The *Path length* is the distance that the Loss Gradient needs to travel backwards during Back Propagation.

At each step, the gradient is subject to being diminished or increased (Vanishing/Exploding gradients).

Since the Transformer operates in parallel across positions, this is $\mathcal{O}\left(1\right)$.

It is $\mathcal{O}\left(T\right)$ for the RNN due to the sequential computation.

The constant path length is critical to the success of the Transformer

- The query used for the input at position t can access **all** prior positions $t' \leq t$ at the same cost
    - Gradient not diminished
    - RNN
        - Gradient signal diminished for position $t' << t$
        - Truncated Back Propagation may kill the gradient flow from position t back to t' beyond truncation window

A key strength of the Transformer is that it enables learning long-range dependencies.

Number of operations

What about the number of operations ?

Let d denote the length of the output of a Transformer

- i.e., $d = |\mathbf{h}_{(t)}|$

When we examine the internals of the Transformer in precise detail

- We will discover additional layers
- The size of the output of each layer is also d

The Self Attention layer attend to (transformed) inputs

- each element assumed size of d

The keys and values of the CSM implementing Attention are the size d input elements.

- Each attention lookup (dot product of query with a key) requires d multiplications.
- There are T key/value pairs in the CSM
- There are T attention units (one for each position, outputting $\mathbf{h}_{(t)}$)

Thus: $\mathcal{O}\left(T^2 * d\right)$ multiplications.

What about the number of operations for an RNN computing the same function ?

The RNN outputs $\mathbf{h}_{(t)}$ of size d (same as Transformer).

- In the RNN $\mathbf{h}_{(t)}$ is also the latent state

The RNN "loops" for T steps.

Each step updates latent state $\mathbf{h}_{(t)}$ via the equation

$$\mathbf{h}_{(t)} = \phi(\mathbf{W}_{xh}\mathbf{x}_{(t)} + \mathbf{W}_{hh}\mathbf{h}_{(t-1)} + \mathbf{b}_h)$$

- $\mathbf{x}_{(t)}$ is also size d (same assumption as for Transformer)
- The weight matrices

  $$\mathbf{W}_{xh} \text{ and } \mathbf{W}_{hh}$$

  are of size

  $$\mathcal{O}\left(d \times d\right)$$

So each step involves d^2 multiplications.

For T steps: $\mathcal{O}\left(T * d^2\right)$ multiplications.

Transformer number of operations: $\mathcal{O}\left(T^2 * d\right)$

RNN number of operations $\mathcal{O}\left(T * d^2\right)$

When $T < d$, the Transformer uses fewer operations compared to the RNN.

Typical values

- $d \geq 768$
- $T < d$ in typical RNN
    - remember: TBPTT divides the input sequence into shorter segments
- **but** $T > d$ in the most recent Transformer modeles
    - Path length is constant, so able to increase T without fear of vanishing/exploding gradients
    - Can capture very long-term dependencies

Number of parameters

The number of parameters is dominated by matrices used in implementing Attention.

As described in the module on [implementing Attention (Attention_Lookup.ipynb#Attention-lookup:-in-practice)]

- the raw inputs at each position (size d)
- are transformed into a different "internal attention size" $d_{\text{attn}} \leq d$
    - for simplicity: we will assume $d_{\text{attn}} = d$
- in parallel across all T positions via matrix multiplication

out		left		right
Q	=	$\mathbf{X}$	*	$\mathbf{W}_Q$
K	=	$\mathbf{X}$	*	$\mathbf{W}_K$
V	=	$\mathbf{X}$	*	$\mathbf{W}_V$
$(T \times d)$		$(T \times d)$		$(d \times d)$

The matrices $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V$ are of size $\mathcal{O}\left(d^2\right)$.

The FFN layer (assuming a single Fully Connected layer)

- mapping attention outputs of size d_{attn} back to d

would also have d^2 parameters.

Total number of parameters in the combined Attention + FFN layers: $\mathcal{O}\left(d^2\right)$.

Complexity: summary

We also throw in a CNN for comparison

The detailed CNN math is given in a following section.

Type	Parameters	Operations	Sequential steps	Path length
CNN	$\mathcal{O}\left(k * d^2\right)$	$\mathcal{O}\left(T * k * d^2\right)$	$\mathcal{O}(T)$	$\mathcal{O}(T)$
RNN	$\mathcal{O}\left(d^2\right)$	$\mathcal{O}\left(T * d^2\right)$	$\mathcal{O}(T)$	$\mathcal{O}(T)$
Self-attention	$\mathcal{O}\left(d^2\right)$	$\mathcal{O}\left(T^2 * d\right)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Reference:

- Transformer Scaling paper (https://arxiv.org/pdf/2001.08361.pdf#page=6)
- Table 1 of Attention paper (https://arxiv.org/pdf/1706.03762.pdf#page=6)
- See Stack overflow (https://stackoverflow.com/questions/65703260/computational-complexity-of-self-attention-in-the-transformer-model) for correction of the number Operations calculated in paper

Here's the details of the math for the CNN

- path length T
  - each kernel multiplication connects only k elements of $\mathbf{x}$
  - since kernels overlap inputs, can't parallelize, hence $\mathcal{O}\left(T/k\right)$ path length
    - can reduce to $\log(T)$ with tree structure
- Parameters
  - kernel size k
  - number of input channels = number of output channels = d
  - $k * d$ parameters for kernel of one channel
  - $\mathcal{O}\left(k * d^2\right)$ parameters for kernel for all d output channels
- Operations
  - for a single output channel: k per input channel
    - There are d input channels, so $k * d$ for each dot product of *one* output channel
    - There are d output channels, so $k * d^2$ per time step
  - T time steps so $\mathcal{O}\left(T * k * d^2\right)$ number of operations

A free lunch ? Almost !

Transformers sound almost too good to be true

- Faster compute (through reduced number of Sequential steps)
- Constant Path Length
    - Better able to capture long range dependencies

Is there really such a thing as a free lunch ?

Almost.

In order to achieve the full benefit of reduced path length

- the operations across all T positions must be computed in parallel
- this involves a tremendous amount of simultaneous compute power
    - very expensive in hardware and power costs

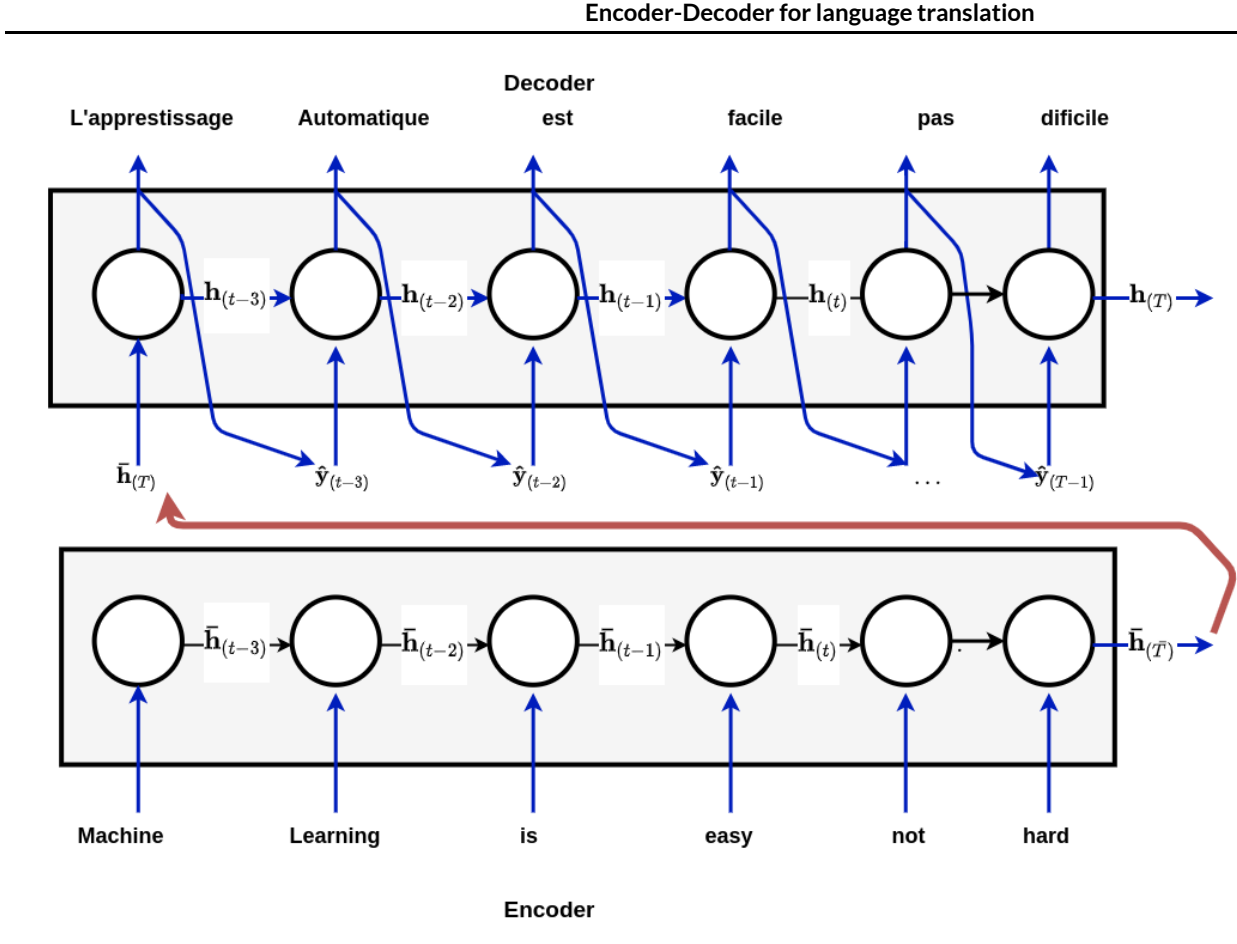
In addition, *positional encoding* needs to be preserved at each layer

- to maintain relative ordering (e.g., for causal attention)
- more complicated than an RNN

Transformer: Decoder style

It is common to use two Transformers in an Encoder-Decoder configuration.

Recall the Encoder-Decoder architecture (using RNN's rather than Transformers in the diagram)



The Decoder in the Encoder-Decoder architecture is *generative*

- Outputs $\hat{\mathbf{y}}_{(t)}$ for a single t at a time
- Appending output $\hat{\mathbf{y}}_{(t)}$ to the input available to output the next $\hat{\mathbf{y}}_{(t+1)}$

The Encoder in the Encoder-Decoder architecture creates a latent state $\bar{\mathbf{h}}_{(t)}$ which summarizes the input prefix $\mathbf{x}_{(1 \ldots t)}$.

In the above diagram the Decoder only has access to $\bar{\mathbf{h}}_{(\bar{T})}$, the final latent state

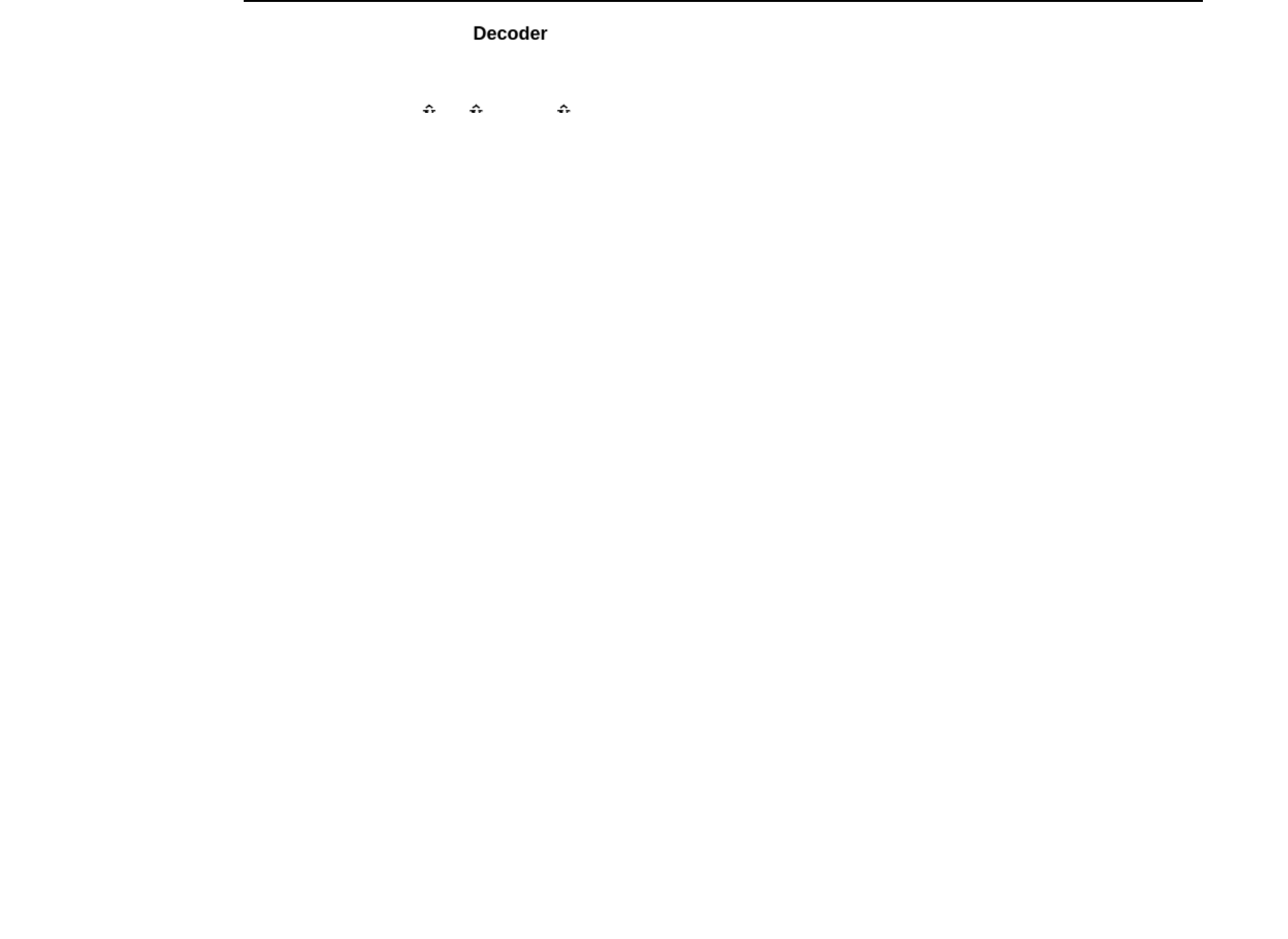
- summarizing the entire input sequence

This is very restrictive, forcing $\bar{\mathbf{h}}_{(\bar{T})}$ to encode a lot of information.

But we motivated Attention by suggesting that the Decoder have access to *each* $\bar{\mathbf{h}}_{(t)}$ for $1 \leq t \leq \bar{T}$.

- and use the Attention mechanism to decide which $\bar{\mathbf{h}}_{(t)}$ to focus on when generating $\hat{\mathbf{y}}_{(t)}$

Decoder: Attention

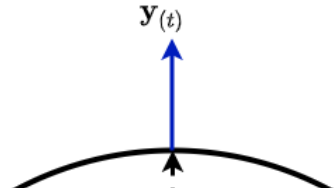

Decoder

The *Decoder Transformer*

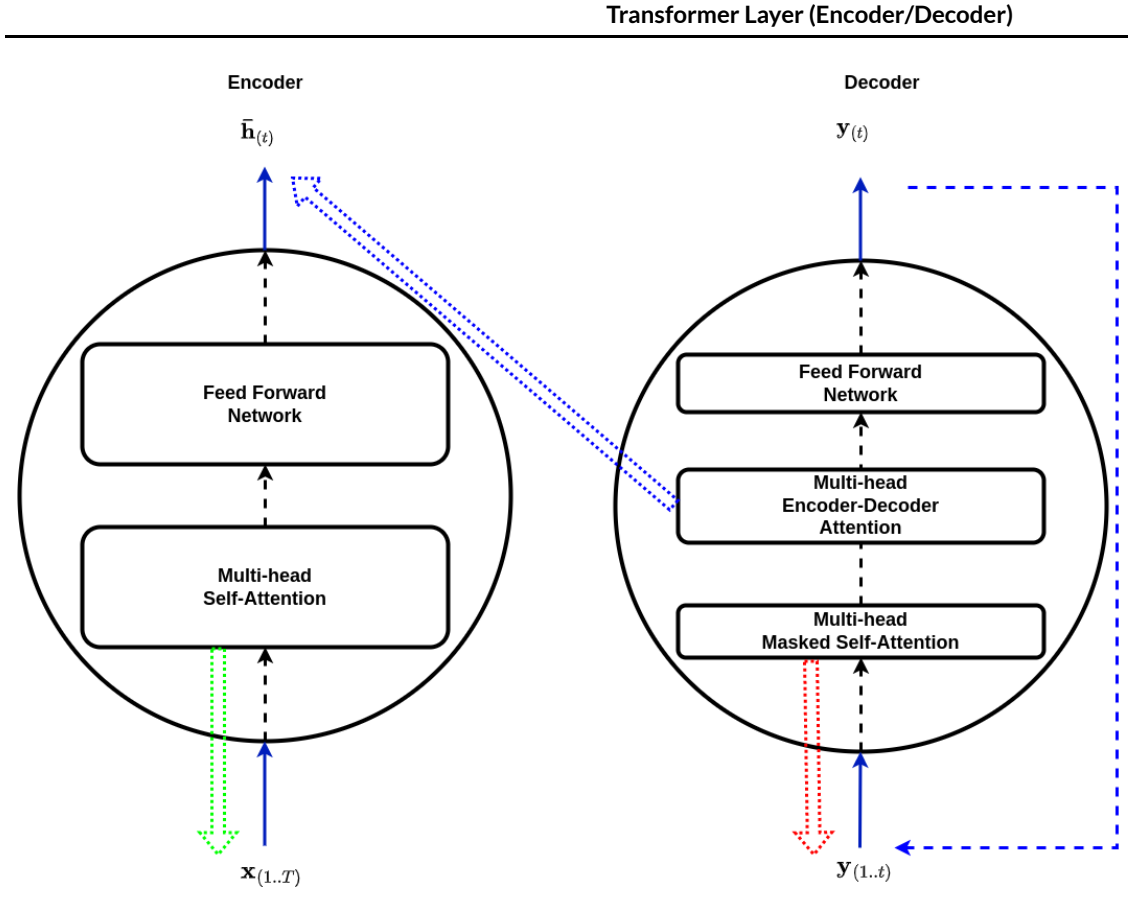
- is similar to the Encoder Transformer in that both use Self-Attention to their own inputs
- differs in that it can also attend to the output of the Encoder.

Attending to the output of another model (e.g., Decoder attending to Encoder output) is called *Cross Attention* (Encoder-Decoder Attention).

Transformer Layer (Decoder)


$\mathbf{y}_{(t)}$

The combinded Encoder-Decoder Transformer diagram looks like this



Explanation of diagram

- The Encoder uses Self-attention (wide Green arrow) to attend to input sequence $\mathbf{x}$
- The Decoder uses Masked Self-attention (wide Red arrow) to attend to its input
    - It's input is the prefix of the output sequence $\mathbf{y}$
    - Limited to prefix of length t by **masking**
- The Decoder uses Cross Attention (between Encoder and Decoder) (wide Blue arrow)
    - To enable Decoder to focus on which Encoder latent state $\bar{\mathbf{h}}_{(t)}$ to atttend to
- The dotted (thin Blue arrow) indicates that the output $\hat{\mathbf{y}}_{(t)}$ is appended to the input that is available when generating $\hat{\mathbf{y}}_{(t+1)}$

Note that the Decoder is recurrent (generative)

- it generates a single output at a time
- unlike the Encoder, which generates all outputs (i.e., "encodings") in parallel

Functional versus Sequential architecture

The architecture diagram is more complex than we have seen thus far.

In particular: data no longer strictly flows forward in a layer-wise arrangement !

- There are two independent sub-networks (Encoder and Decoder)
- Connection from the Encoder output to the middle of the Decoder (Cross-Attention)

Each of the Encoder and Decoder is an independent Functional model.

- not our familiar Sequential modles

The Encoder-Decoder pair combination is also constructed as a Functional model.

Since we have not yet addressed Functional Models, you may not be prepared to completely grasp the totality.

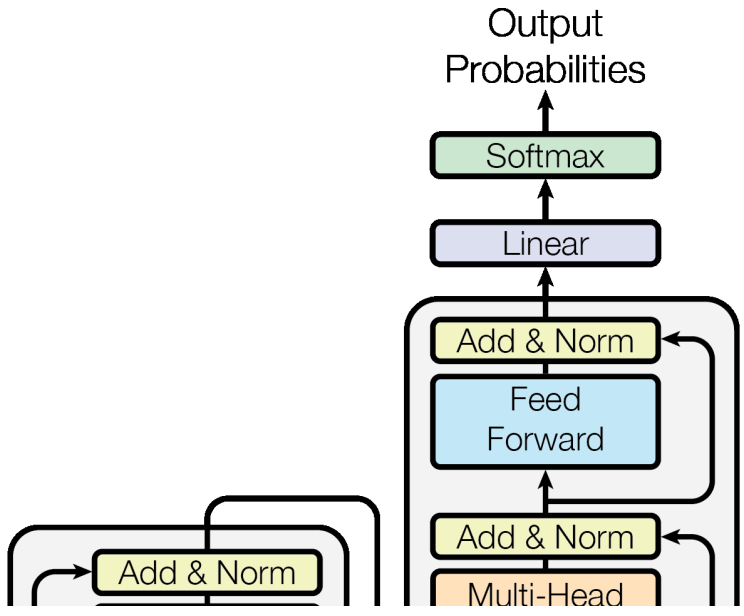
But hopefully you can absorb the concepts even without fully understanding the details.

Detailed Encoder-Decoder Transformer architecture

There are other components of the Encoder and Decoder that we have yet to describe.

We will do so briefly.

(The Transformer was introduced in the paper Attention is all you Need (https://arxiv.org/pdf/1706.03762.pdf)


Output
Probabilities
Softmax
Linear
Add & Norm
Feed
Forward
Add & Norm
Multi-Head
Add & Norm

Embedding layers

We will motivate and describe Embeddings in the NLP module.

For now:

- an embedding is an encoding of a categorical value that is shorter than OHE

It is used in the Transformer to

- encode the input sequence of words
- encode the output sequence of words

Positional Encoding

The inputs are ordered (i.e., sequences) and thus describe a relative ordering relationship between elements.

But inputs to most layer types (e.g., Fully Connected) are unordered.

The Positional Encoding is a way of encoding the the relative ordering of elements.

To represent the relative position of each element in the sequence,

- we can pair the input element with an encoding of its position in the sequence.

$$\langle \mathbf{x}_{(t)}, \mathrm{encode}(t) \rangle$$

The box labeled "Positional Encoding" creates $\mathrm{encode}(t)$.

The "+" concatenates the Input Embedding and Positional Encoding to create $\langle \mathbf{x}_{(t)}, \mathrm{encode}(t) \rangle$.

If relative position is important, the NN can learn the values of $\mathrm{encode}(t)$.

The encoding is subtle.

A fuller explanation is given in this module (Transformer_PositionalEmbedding.ipynb).

Self Attention layers (Encoder and Decoder)

The 3 arrows flowing into the Multi-Head Attention box

- are identical
- are the inputs (after being Embedded and having Positional Encoding added)

The Self-Attention layers for the Encoder and Decoder

- **differ in that the Decoder uses Causal Masking versus no-masking for the Encoder**
- Decoder can't "look ahead" at output $\mathbf{y}_{t'}$ for $t' \geq t$
    - it hasn't been generated yet at test time step t
    - it **is** available at training time (via Teacher Forcing)
        - but shouldn't look at it during training time, in order for training to be similar to test time

Cross Attention layer (Decoder)

The two arrows flowing from the Encoder output are the keys and values of the CSM

The arrow flowing from the Self Attention layer is the query

- The output of the Self Attention layer is the **query** used in Cross Attention

Add and Norm

We have seen each of these layer types before

- Norm: Batch (or other) Normalization layers
- Add: the part of the residual network that joins outputs of multiple previous layers

The diagram shows an Encoder-Decoder pair.

You will notice that each element of the pair is different.

- It is possible to use each element independently as well.
- But first we need to understand the source of the differences and their implications.

How is the direct function computed ?

The Encoder uses self-attention

- So the keys and values of the CSM are derived directly from input sequence $x_{(1...T)}$

During training, the Encoder

- learns a query, derived from input sequence $\mathbf{x}_{(1...T)}$
- learns weights for the Feed Forward Network

The Attention output

- is equal to a weighted combination of CSM values
    - i.e., weighted sum of input elements

The Feed Forward Network transforms the Attention output into Encoder output $\bar{\mathbf{h}}_{(t)}$.

Similarly for the Decoder.

The Self-Attention layer CSM has keys and values that are incrementally constructed from the outputs $\mathbf{y}_{(1\ldots,t)}$ that have been created from the first t steps.

The Cross-Attention layer CSM has keys and values that are outputs $\bar{\mathbf{h}}_{(t)}$ of the Encoder.

During training, the Self-Attention layer outputs **the query** that is used for Cross Attention.

The query is created by self-attention to the inputs.

The Decoder **learns (from training)**

- the Self-Attention query
- the Cross Attention query
- the weights of the Feed Forward Network

Stacked Transformer

Just as with many other layer types (e.g., RNN), we may stack Transformer layers.

- Each layer creating alternate representations of the input of increasing complexity

In fact, stacking $N > 1$ Transformer layers is typical.

$N = 6$ was the choice of the original paper.

Stacked Transformer Layers (Encoder/Decoder)

Uses of an Encoder-style Transformer

The Transformer for the Encoder and Decoder of an Encoder-Decoder Transformer are slightly different.

They can also be used individually as well as in pairs.

It's important to understand the differences in order to know when to use each individually.

The Encoder side of the pair **does not** restrict the order in which it's inputs are accessed.

- Self-attention **without** causal masking

So the Encoder is appropriate for tasks that require a context-sensitive representation of each input element.

For example: the meaning of the word "**it**" changes with a small change to a subsequent word in the following sentences:

- "The animal didn't cross the road because **it** was too tired"
- "The animal didn't cross the road because **it** was too wide"

Some tasks with this characteristic are

- Sentiment
- Masked Language Modeling: fill-in the masked word
- Semantic Search
    - compare a summary of the sequence that is the context-sensitive representation of
        - query sentence
        - document sentences
    - Each summary is a kind of **sentence embedding**
    - Summary
        - pooling over each word
        - final token

Uses of a Decoder-style Transformer

One notable aspect of the Decoder is its recurrent (generative) architecture

- Output $\mathbf{y}_{(t-1)}$ is appended to the Decoder inputs available at step t.
  - The Decoder inputs are $\mathbf{y}_{(1..T)}$, where T is the full length of the Decoder output
  - **But** Causal Masking ensures that only $\mathbf{y}_{(1..t)}$ is *available* at step t.

Thus, the Decoder is appropriate for *generative* tasks

- Text generation
- Predict the next word in a sentence

Conclusion

The Transformer architecture has come to dominate tasks with long sequences (e.g., NLP).

The operations of a Transformer occur in parallel for each position.

This allows us to leverage the compute time

- Use many stacked Transformer layers
- At time cost still less than a sequential RNN layer

Moreover, the constant path length means the gradients are less likely to vanish/explode for long sequences

- No need to truncate Back Propagation as in an RNN
- Long term dependencies between positions become feasible.

We pay for these advantages in terms of increasing

- number of operations
    - but they occur in parallel, so no increase in elapsed time
- number of weights

Thus, Transformer training is both compute and memory intensive.

- This limits the number of individuals/organizations able to train very large models.

In [2]:
```
print("Done")
```

```
Done
```