Dealing with Sequences: Recurrent Neural Network (RNN) layer

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

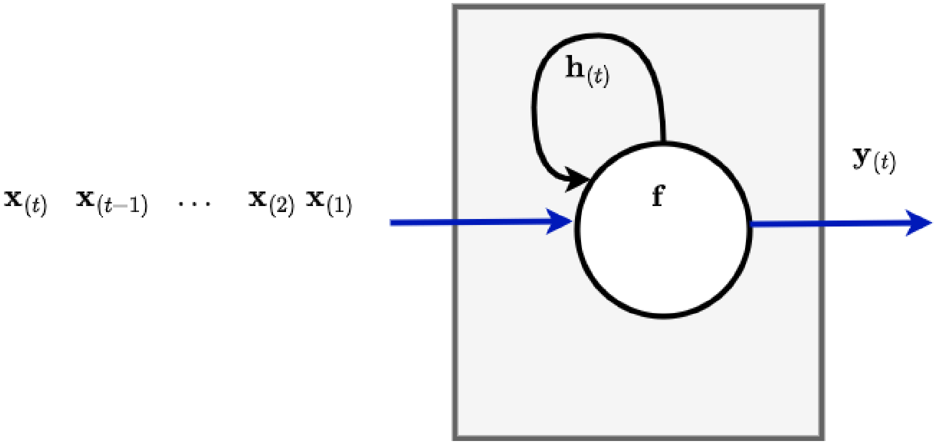
The RNN implements the function as a "loop"

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

$\backslash\mathrm{pr}\backslash\mathrm{h}_{\backslash\mathrm{tp}} | \backslash\mathrm{x}_{\backslash\mathrm{tp}}, \backslash\mathrm{h}_{(-1)}$ latent variable $\backslash\mathrm{h}_{\backslash\mathrm{tp}}$encodes $[\backslash\mathrm{x}_{(1)} \ldots \backslash\mathrm{x}_{\backslash\mathrm{tp}}]$

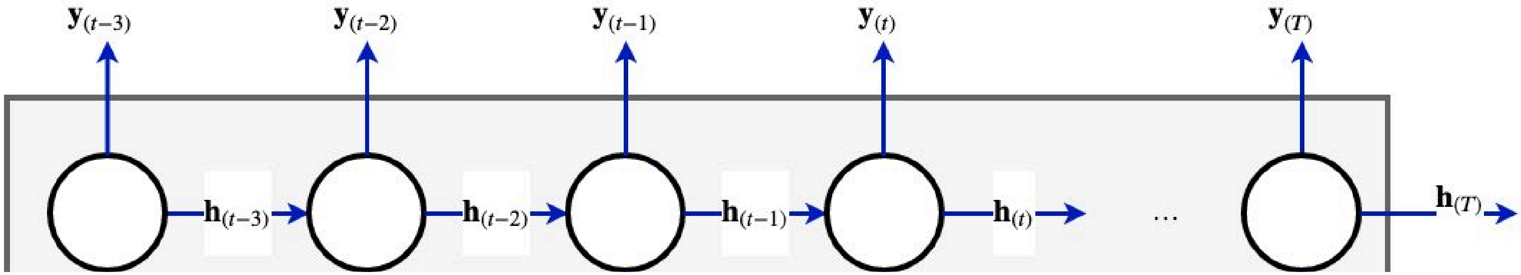
$\backslash\mathrm{pr}\backslash\mathrm{y}_{\backslash\mathrm{tp}} | \backslash\mathrm{h}_{\backslash\mathrm{tp}}$ prediction contingent on latent variable

Loop with latent state


$\mathbf{x}_{(t)}$ $\mathbf{x}_{(t-1)}$ $\ldots$ $\mathbf{x}_{(2)}$ $\mathbf{x}_{(1)}$
$\mathbf{h}_{(t)}$
$\mathbf{f}$
$\mathbf{y}_{(t)}$

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

RNN unrolled


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

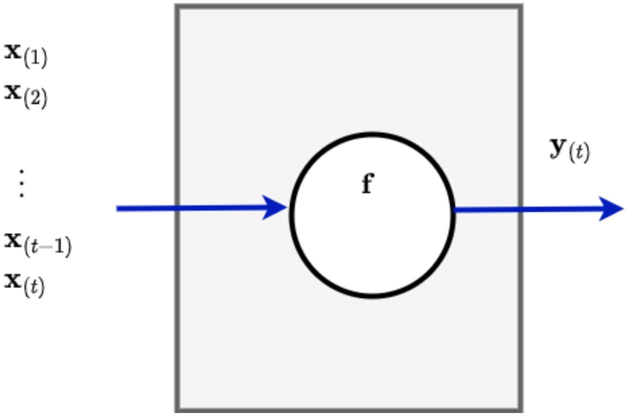
Transformer variants

Encoder style

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

- Taking a **sequence** $\backslash\mathbf{x}_{(1...)}$ as input
- Outputting $\backslash\mathbf{y}_{\backslash\mathrm{tp}}$

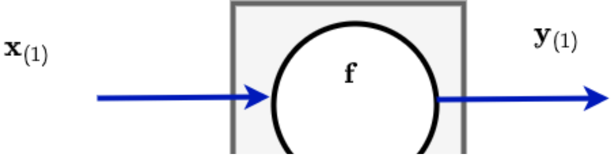
Direct function



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

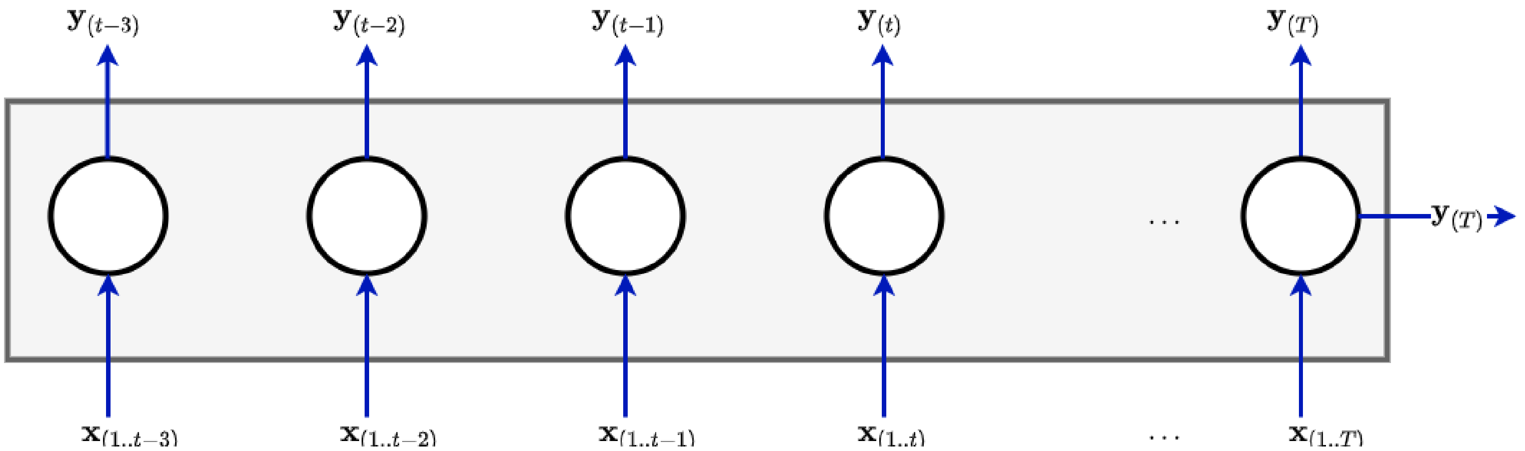
- computes each $\backslash\mathbf{y}_{\backslash\mathrm{tp}}$ in **parallel**, not sequentially as in the loop

Direct function, in parallel (masked input)


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

The parallel units constitute a *Transformer Encoder*

Transformer Encoder (causal masked input)



Compared to the unrolled RNN, the Transformer Encoder

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

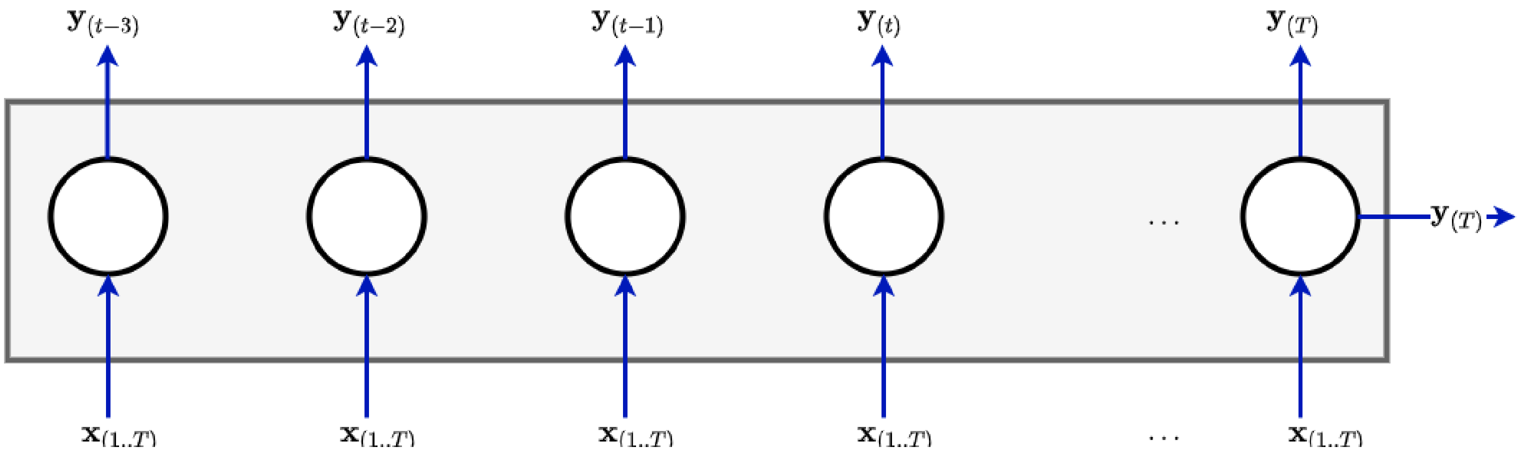
With this architecture, we can compute more general functions than the RNN

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

Direct function, in parallel (un-masked input)

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

Transformer Encoder (unmasked input)



With *unmasked* Self Attention

- output $\backslash\hat{\mathbf{y}}_{\backslash\mathrm{tp}}$ at position
- is a function of **all** inputs $\backslash\mathbf{y}_{(1..T)}$
    - including positions after

This is useful, for example, when the meaning of a word depends on its *entire* context.

For certain tasks, full visibility of all inputs is not permissible

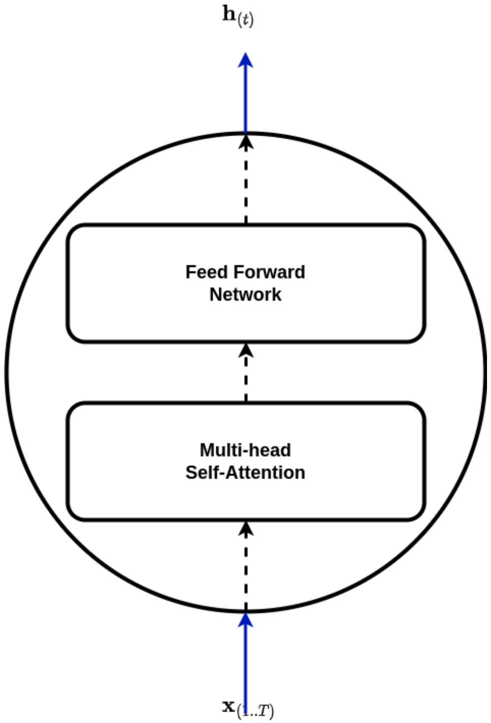
- "looking into the future"
    - e.g., predict stock return based only on **past** information

In this case, we use *masked* Self Attention

- output $\backslash\hat{\mathbf{y}}_{\backslash\mathrm{tp}}$ at position
- is a function of **preceding** inputs $\backslash\mathbf{y}_{(1..-1)}$

Here is a diagram of an Encoder style Transformer block.

Transformer layer: Encoder style



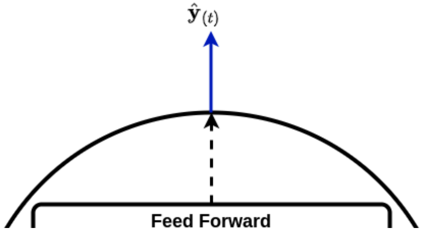
Decoder Style (stand-alone)

Although we have introduced the Decoder as part of an Encoder/Decoder pair, the decoder can also be used independently from an Encoder.

This style decoder is similar to an Encoder

- **but uses masked Self-Attention rather than unrestricted Attention**
- **this is because it is often**
    - **trained with Teacher Forcing**
    - **see the section on Auto Regressive Decoder behavior**

Transformer layer: Decoder style


$\hat{\mathbf{y}}_{(t)}$
Feed Forward

Auto regressive Decoder behavior

A Decoder usually operates in an *auto regressive* manner

- it operates in a "loop"
- feeding the output of iteration
- back as the input for iteration **+1**

It builds the final output sequence

$$\backslash\hat{\mathbf{y}}_{(1:T)}$$

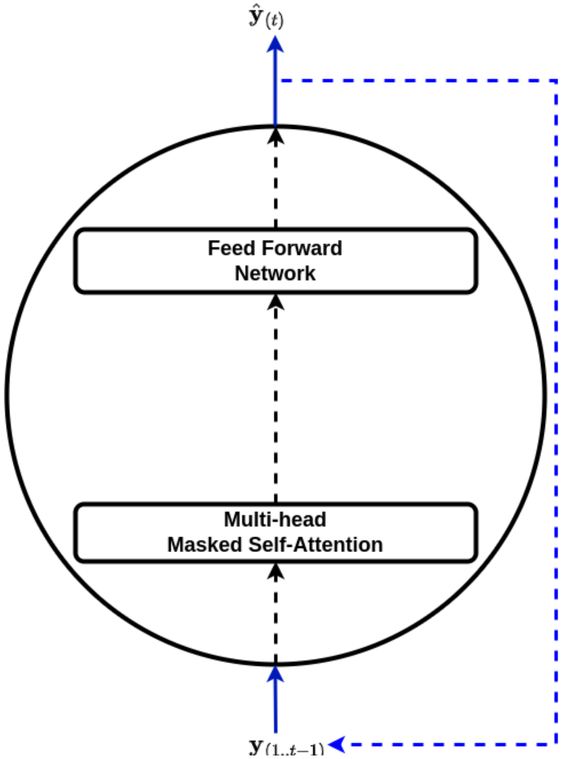
one element at a time.

This is called a *generative* process.

This is accomplished by

- appending the output $\backslash\hat{\mathbf{y}}_{\backslash\mathbf{tp}}$ of iteration
- to it's input $\backslash\hat{\mathbf{y}}_{(1:-1)}$
- and using the newly lengthened sequence
- as input for iteration $\mathbf{+1}$.

Transformer layer: Auto-Regressive Decoder style



Confusion alert

The *internal* behavior of the Decoder is "loop-free"

- it processes the input sequence $\backslash\hat{\mathbf{y}}_{(1:-1)}$
- in parallel (all positions at once)
    - direct function computation rather than a loop

The *external* behavior is to use one-step of the Decoder

- producing a single output $\backslash\hat{\mathbf{y}}_{\backslash \mathbf{tp}}$
- in an *loop* that calls the single-step Decoder

Notice that the Decoder has no internal latent state

- iteration **`+1`** re-processes the entire new input sequence $\backslash\hat{\mathbf{y}}_{(1:)}$
- rather than incrementally updating the result ("latent state") of having previously processed prefix $\backslash\hat{\mathbf{y}}_{(1:-1)}$

because it uses the direct function approach.

Note on the diagram

We show the input for iteration

- as target

  $$\backslash\mathbf{y}_{(1:-1)}$$

  rather than the generated

  $$\backslash\hat{\mathbf{y}}_{(1:-1)}$$

as an illustration of Teacher Forcing as training time.

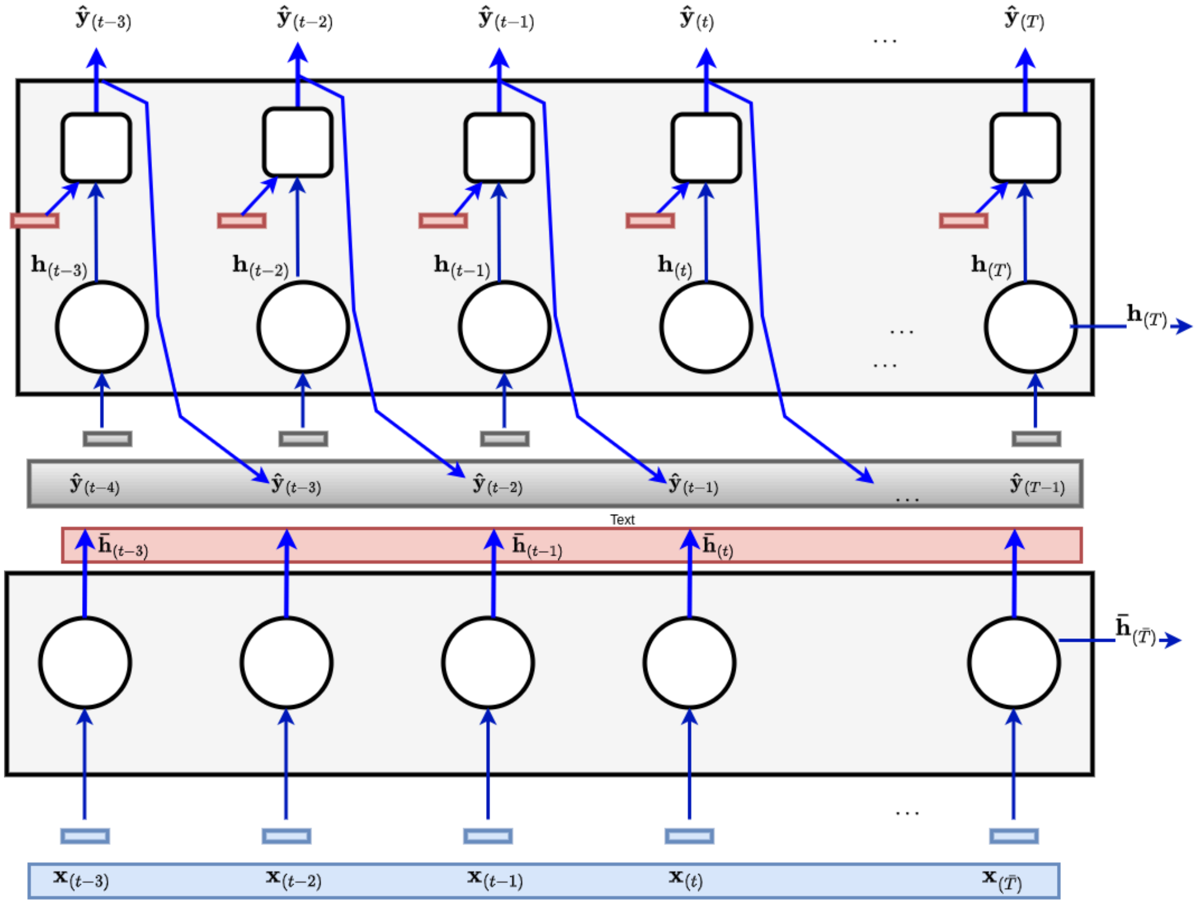
Decoder style within Encoder/Decoder

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

Refer back to our [Attention module (Intro_to_Attention.ipynb#Attention)](Intro_to_Attention.ipynb#Attention)

- used to motivate Attention
- through several steps
    - we modified a pair of RNN's (Encoder and Decoder)
    - into a pair of direct function modules
    - which form the basis for the Transformer

Encoder/Decoder with Cross Attention and Self Attention (Encoder/Decoder)



The Encoder part of the pair

- has full visibility to the input sequence $\backslash\mathbf{x}_{(1\ldots\bar{T})}$
    - via un-masked Self-Attention
- it transformers the input into sequence $\backslash\bar{\mathbf{h}}_{(1\ldots\bar{T})}$
    - which is made available to the Decoder when generated each element of the Output $\backslash\hat{\mathbf{y}}_{\backslash\mathbf{tp}}$

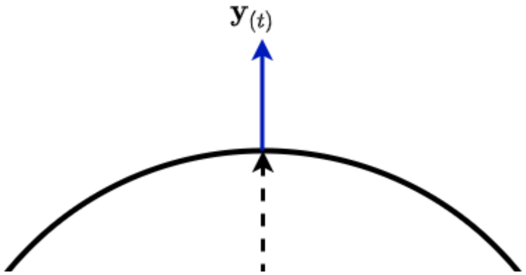
The Decoder part of the pair

- can attend to the Encoder Input $\backslash\bar{\mathbf{h}}_{(1...\bar{T})}$ when generating any Output $\backslash\hat{\mathbf{y}}_{\backslash\mathbf{tp}}$
- the Output is produced auto-regressively
    - so Output at position is a function of
        - $\backslash\hat{\mathbf{y}}_{(1..-1)}$: the Output generated thus far
        - the encoded input $\backslash\bar{\mathbf{h}}_{(1...\bar{T})}$

The Decoder, when paired with an Encoder, uses two types of Attention

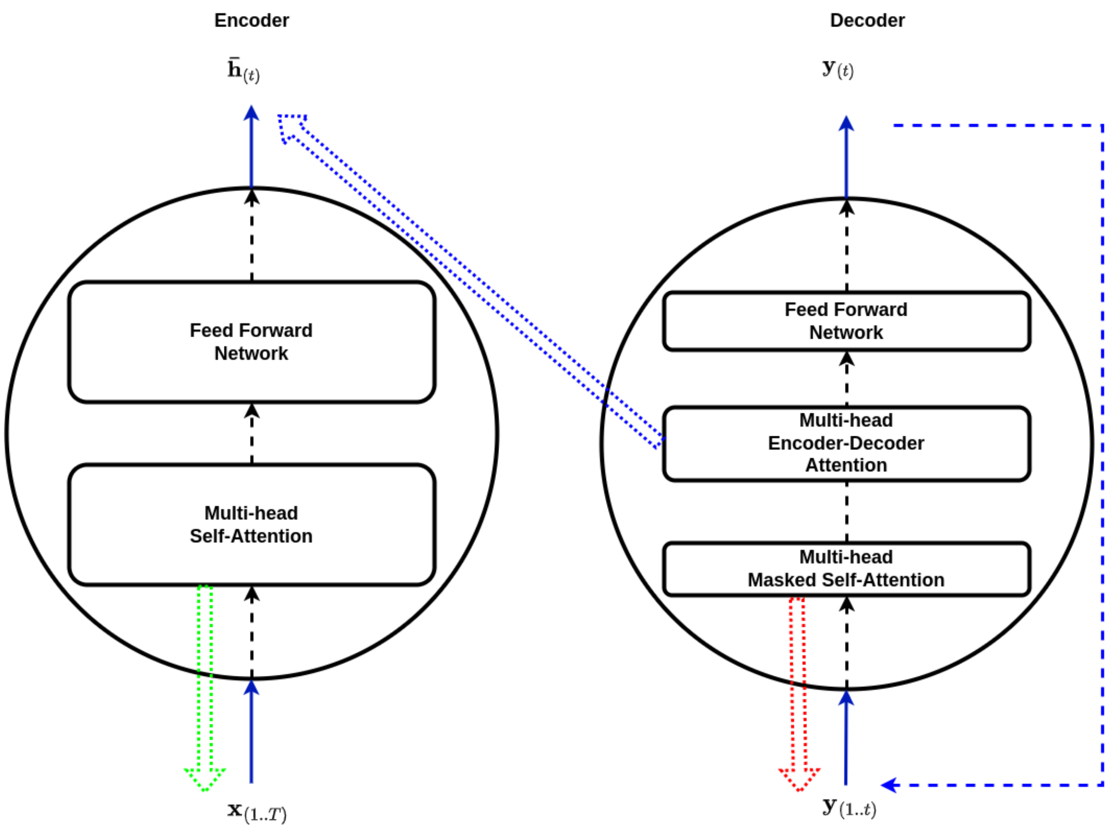
- Attention to the Encoder output $\backslash\bar{\mathbf{h}}_{(1...\bar{T})}$: Encoder/Decoder Cross Attention
- Self-Attention to the Output generated thus far $\backslash\hat{\mathbf{y}}_{(1..-1)}$
  - Masked Self-Attention
    - Self-Attention: attends to its own input
    - Masked: access only to prefix $\backslash\hat{\mathbf{y}}_{(1..-1)}$ rather than full $\backslash\hat{\mathbf{y}}_{(1..T)}$

Transformer Layer (Decoder)


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

The combinded Encoder-Decoder Transformer diagram looks like this

Transformer Layer (Encoder/Decoder)



Explanation of diagram

- The Encoder uses Self-attention (wide Green arrow) to attend to input sequence $\backslash\mathbf{x}$
- The Decoder uses Masked Self-attention (wide Red arrow) to attend to its input
  - It's input is the prefix of the output sequence $\backslash\mathbf{y}$
  - Limited to prefix of length by masking
- The Decoder uses Cross Attention (between Encoder and Decoder) (wide Blue arrow)
  - To enable Decoder to focus on which Encoder latent state $\backslash\bar{\mathbf{h}}_{\backslash\mathbf{tp}}$ to atttend to
- The dotted (thin Blue arrow) indicates that the output $\backslash\hat{\mathbf{y}}_{\backslash\mathbf{tp}}$ is appended to the input that is available when generating $\backslash\hat{\mathbf{y}}_{(+1)}$

Note that the Decoder is Auto Regressive

- it generates a single output at a time

The diagram illustrates its input

- as would be used at Training time
- with Teacher Forcing

Thus the need for *masked* self-attention.

Use cases for each variant of 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.

Encoder/Decoder uses

The Encoder/Decoder acts as a function

- from Input, processed by the Encoder
- to Target, processed by the Decoder

This is a natural architecture for Sequence to Sequence tasks.

An advantage of this architecture

- it decouples the Input and Output sequences
- which may be different lengths
- and may not be in one-to-one correspondence
    - e.g., in some languages: adjectives precede nouns; it is reversed in others

Encoder only uses

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

- Self-attention without causal masking

Thus the Encoder output at *each* position is a function of the input at *all* positions.

This is valuable 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

It is often the case that special tokens are added to the input of an Encoder style transformer.

- Designating a special role for this token, compared to the other tokens in the sequence
- For example `<CLS>` ("Classification") is the single token used as input to a subsequent Classifier layer

Thus Encoder style Transformers are usually used as the first "layer" of a multi-layer network

- creating a context-sensitive "understanding" of
    - each token
    - the entire sequence
- which can then be manipulated by later layers
    - e.g., task-specific Classification head

Decoder only uses

A Decoder style Transformer

- looks like the Decoder side of the Encoder-Decoder
- *without* Cross-Attention, since there is no Encoder

One notable aspect of the Decoder is its auto-regressive behavior

- Initial input is empty
- Output $\backslash\hat{\mathbf{y}}_{(-1)}$ is appended to the Decoder inputs available at step .
- step input: $\backslash\hat{\mathbf{y}}_{([0:-1])}$

Thus, a Decoder only Transformer is useful for completely *generative* task

- create sequence output
- from no input

One can modify a Decoder only Transformer to implement a function from Input to Target

- just like an Encoder/Decoder
- by initializing the Decoder input to the function input sequence $\backslash\mathbf{x}_{(0..\bar{T})}$

Thus, a Decoder only Transformer become similar in function to an Encoder/Decoder Transformer.

- but with half the parameters (since no Encoder)

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

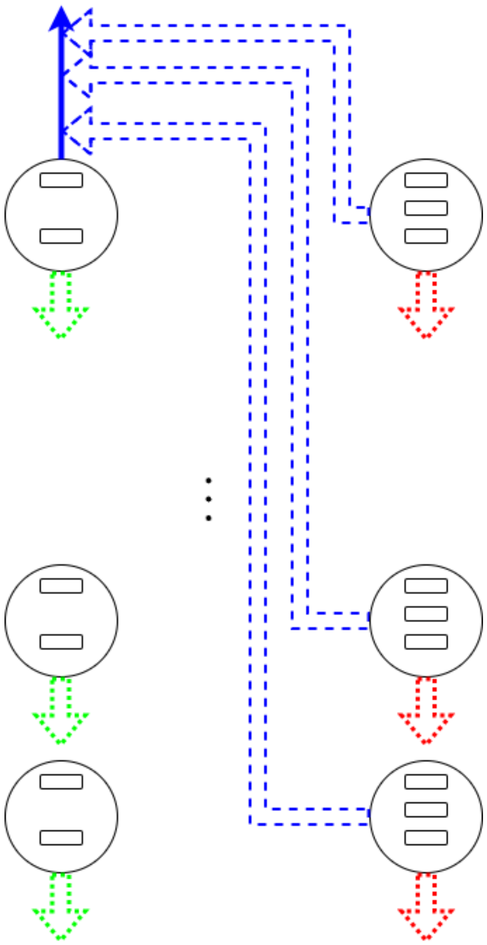
In fact, stacking $N > 1$ Transformer layers is typical.

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

Note that this is still an Encoder/Decoder

- so the *final* output of the Encoder is attended to by *each* layer of the Decoder.

Stacked Transformer Layers (Encoder/Decoder)



Advantages of a Transformer compared to an RNN (w/o Attention)

We will illustrate some of the advantages that Attention brings to the Transformer

- high degree of parallelism possible
- facilitates longer sequences
    - Vanishing Gradient no longer an issue
- facilitates more complex computation (via deeper stack of layers)

The price for this

- potentially higher number of operations

We compare a Decoder style (Auto-regressive) Transformer to an RNN without Attention.

Number of sequential steps

Let's consider the output sequence $\backslash\hat{\mathbf{y}}_{(1:T)}$ produced by an Auto-regressive Decoder.

During training, we use Teacher Forcing

- so that $\backslash\hat{\mathbf{y}}_{\backslash\mathbf{tp}}$ depends on true target $\backslash\mathbf{y}_{(\mathbf{1}:-\mathbf{1})}$
- rather than $\backslash\hat{\mathbf{y}}_{(\mathbf{1}:-\mathbf{1})}$

Note

Even though $\backslash\hat{\mathbf{y}}_{\backslash\mathbf{tp}}$ is not fed into the input of the next step

- it still must be computed
- as the Loss for step depends on it

The RNN (without Attention) must produce each output $\backslash\hat{\mathbf{y}}_{\backslash\mathbf{tp}}$ sequentially

- even though the full target sequence $\backslash\mathbf{y}_{(1:T)}$ is available in the training examples
- because $\backslash\hat{\mathbf{y}}_{\backslash\mathbf{tp}}$ depends on the previous latent state $\backslash\mathbf{h}_{(-1)}$

Thus an RNN takes T sequential steps to process an entire sequence of length T.

- spanning T training examples

But, because of Attention, a Transformer

- can produce the output $\backslash\hat{\mathbf{y}}_{\backslash\mathbf{tp}}$ of all T training examples $1 \leq\leq \mathtt{T}$
- *simultaneously*
- given the available of parallel compute

That is, to produce T outputs

- the Transformer takes 1 step
- the RNN takes T steps

This means that

- training time using a Transformer
- can be reduced compared to an RNN
- via parallel processing
    - multiple GPU's
    - because we can compute the loss of all T examples in a sequence
    - in one parallel time step

We can leverage this time advantage by

- Increasing the number of Transformer layers in the stack
- So each time step now takes longer
    - the blocks in the stack are processed sequentially
    - but this is a constant (finite number of layers)
- Trading off increased depth for a more powerful (deeper) stack

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 \OrderOf$1$.

It is \OrderOf$T$ 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 can access all prior positions$'$ $\le$ at the same cost**
    - **Gradient not diminished**
    - **RNN**
        - **Gradient signal diminished for position$'$ <<**
        - **Truncated Back Propagation may kill the gradient flow from position back to$'$ beyond truncation window**

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

Number of parameters

In the Transformer, the Q, K, V matrices are first projected through $(d \times d)$ matrices, $\backslash\mathbf{W}_k, \backslash\mathbf{W}_Q, \backslash\mathbf{W}_V$

out		left		right
Q	=	Q	*	$\backslash\mathrm{W}_Q$
$(T \times d)$		$(T \times d)$		$(d \times d)$

K | = | K| | $\backslash W_K$ | V | = | V| |$\backslash\mathbf{W}_V$ | $(\bar{T} \times d)$ | | $(\bar{T} \times d)$ | | $(d \times d)$

Each of the matrices, $\backslash\mathbf{W}_k, \backslash\mathbf{W}_Q, \backslash\mathbf{W}_V$, is $\backslash\mathrm{OrderOf}d^2$ parameters.

The Feed Forward Network is usually implemented by 2 `Dense` layers.

the first takes the length d attention output

- and creates d_{ffn} new features
- by custom: $d_{\text{ffn}} = 4 * d$

The second takes the length d_{ffn} output of the first and creates the final length d_{model} output.

Thus, each `Dense` layer has \OrderOf$d^2$ weights.

Number of operations

What about the number of operations ?

The Attention Lookup is computed via matrix multiplication

$$Q * K^T * V$$

$Q * K^T$ has $(T \times \bar{T})$ elements, each the result of d multiplications.

Thus: \OrderOf$T^2 * d$ multiplications.

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 $\backslash\mathbf{h}_{\backslash\mathbf{tp}}$)

Thus: $\backslash\mathrm{OrderOf}T^2 * d$ multiplications.

RNN calculations

Let's examine the RNN's number of operations and weights.

The RNN inputs $\backslash\mathbf{x}_{\backslash\mathbf{tp}}$ and outputs $\backslash\mathbf{h}_{\backslash\mathbf{tp}}$ of size d (same as Transformer).

- In the RNN $\backslash\mathbf{h}_{\backslash\mathbf{tp}}$ is also the latent state

Each step of the RNN updates the latent state $\backslash\mathbf{h}_{\backslash\mathbf{tp}}$ via the equation

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

The weight matrices

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

are of size

$$\backslash\mathrm{OrderOf} d \times d$$

- transforming length d vectors ($\backslash\mathbf{x}_{\backslash\mathbf{tp}}$, $\backslash\mathbf{h}_{(-1)}$) into a length d vector $\backslash\mathbf{h}_{\backslash\mathbf{tp}}$

The multiplication of $(d \times d)$ weights matrices times a vector of length d

- requires d multiplications per element
- there are d elements in $\backslash\mathbf{h}_{\backslash\mathbf{tp}}$

Thus $\backslash\mathrm{OrderOf}d^2$ operations per time step.

There are T *sequential* time-steps

- $\backslash\mathrm{OrderOf}T * d^2$ total operations
- involving T sequential steps
  - steps are computed sequentially in the RNN, versus in parallel in the Transformer
- path length T as gradient flows backward through each of the T time steps

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	\OrderOf$k * d^2$	\OrderOf$T * k * d^2$	\OrderOf$T$	\OrderOf$T$
RNN	\OrderOf$d^2$	\OrderOf$T * d^2$	\OrderOf$T$	\OrderOf$T$
Self-attention	\OrderOf$d^2$	\OrderOf$T^2 * d$	\OrderOf$1$	\OrderOf$1$

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

Transformer main point of comparison to the RNN

- fewer Sequential Steps: $\backslash\mathrm{OrderOf}1$ versus $\backslash\mathrm{OrderOf}T$
- operations: $\backslash\mathrm{OrderOf}T^2 * d$ versus $\backslash\mathrm{OrderOf}T * d^2$
    - more when sequences are long, i.e., $T > d$

But: because of the reduced number of sequential steps, Transformers

- can stack *many* (i.e., n_{layers}) blocks, each taking $\backslash\text{OrderOf}1$ time
    - $\backslash\text{OrderOf}n_{\text{layers}}$ Sequential Steps total
- and still be less than the $\backslash\text{OrderOf}T$ Sequential Steps of an RNN
- at the cost of increasing number of operations and parameters by $\backslash\text{OrderOf}n_{\text{layers}}$

Transformers consume larger number of parameters and operations through this factor of n_{layers} blocks.

CNN calculations

Here's the details of the math for the CNN

- path length T
  - each kernel multiplication connects only k elements of \x
  - since kernels overlap inputs, can't parallelize, hence \OrderOf$T/k$ 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
  - \OrderOf$k * d^2$ 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 \OrderOf$T * k * d^2$ 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**

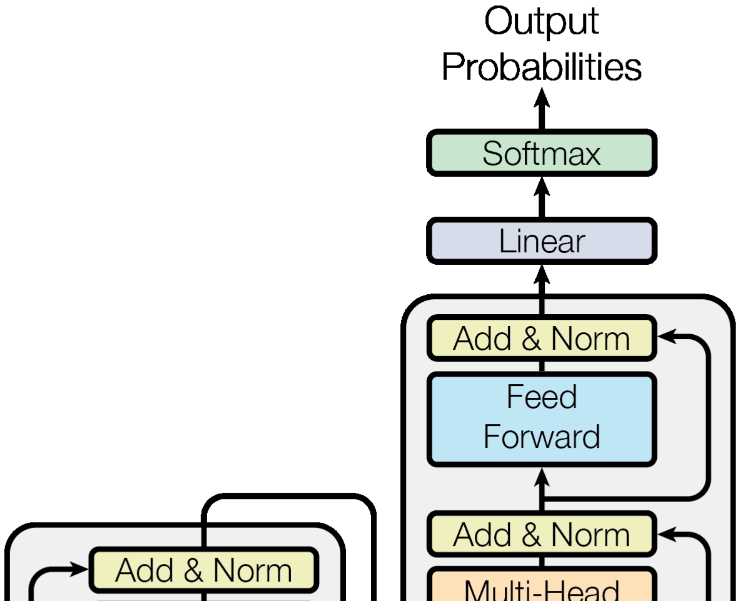
Detailed Encoder/Decoder Transformer architecture

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)](https://arxiv.org/pdf/1706.03762.pdf)

Transformer (Encoder/Decoder)


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.

$$\langle \backslash\mathbf{x}_{\backslash\mathbf{tp}}, \mathbf{encode}() \rangle$$

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

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

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

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 $\backslash\mathbf{y}$, for $' \geq$
    - it hasn't been generated yet at test time step
    - 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.**

What happens during training ?

Encoder

The Encoder uses self-attention

- So the keys and values of the CSM are derived directly from input sequence $\backslash\mathtt{x}_{(1...T)}$

During training, the Encoder

- learns a query, derived from input sequence $\backslash\mathtt{x}_{(1...T)}$
- 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 $\backslash\bar{\mathbf{h}}_{\backslash\mathbf{tp}}$.

Decoder

Similarly for the Decoder.

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

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

During training, the Self-Attention layer learns to construct

- The query that is used for Self Attention.
    - attention to the inputs.
- The output of Self-Attention
    - the weighted sum of input positions
    - becomes the query that is used for Cross Attention.
- The query used for Cross Attention
    - attention to the Encoder outputs
- The output of Cross Attention
    - the weighted sum of Encoder outputs
    - becomes the input to the Feed Forward Network
- The weights of the Feed Foward Network
    - this is where "world knowledge" from training data is stored

Technical clarifications

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.

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 $\backslash\hat{\mathbf{y}}_{\backslash\mathbf{tp}}$ with $\backslash\mathbf{h}_{\backslash\mathbf{tp}}$

The simplest RNN (corresponding to our diagrams) use the latent state $\backslash\mathbf{h}_{\backslash\mathbf{tp}}$ as the output $\backslash\hat{\mathbf{y}}_{\backslash\mathbf{tp}}$

$$\backslash\hat{\mathbf{y}}_{\backslash\mathbf{tp}} = \backslash\mathbf{h}_{\backslash\mathbf{tp}}$$

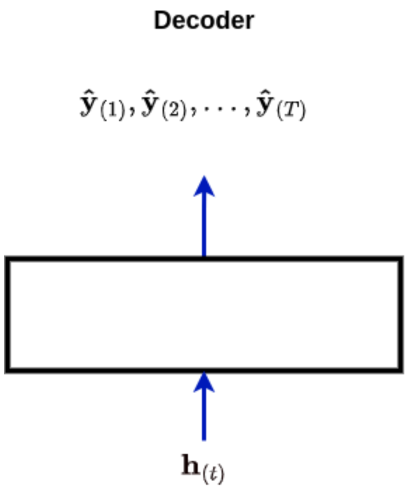
It is easy to add another NN to transform $\backslash\mathbf{h}_{\backslash\mathbf{tp}}$ into a $\backslash\hat{\mathbf{y}}_{\backslash\mathbf{tp}}$ that is different.

- We can add a NN to the Decoder RNN that implements a function D that transforms the latent state into an output.

$$\backslash\hat{\mathbf{y}}_{\backslash\mathbf{tp}} = D(\backslash\mathbf{h}_{\backslash\mathbf{tp}})$$

Here is what the additional NN looks like:

Decoder output transformation: No attention


Decoder
$\hat{\mathbf{y}}_{(1)}, \hat{\mathbf{y}}_{(2)}, \ldots, \hat{\mathbf{y}}_{(T)}$
$\mathbf{h}_{(t)}$

In the context of the Transformer:we will assume the style of a single output $\backslash\mathbf{h}_{\backslash\mathbf{tp}}$

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 $\backslash\mathbf{h}_{\backslash\mathbf{tp}}^{[j]}$, not the final $\backslash\mathbf{y}_{\backslash\mathbf{tp}}$
- We identify $\backslash\mathbf{y}_{\backslash\mathbf{tp}}$ as the output of the top layer $\backslash\mathbf{h}_{\backslash\mathbf{tp}}^{[N]}$
    - perhaps after a further processing

Furthermore:

Since the Encoder part is no longer a "loop"

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

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
```