# Encoder/Decoder architecture

Two RNN's

- Encoder: takes input sequence $\backslash\mathbf{x}$
- Decoder: creates output sequence $\backslash\hat{\mathbf{y}}$

RNN's process sequences using the "loop architecture"

Consider the task of

- constructing the *next* element $\backslash \hat{\mathbf{y}}_{\backslash \mathrm{tp}}$ of sequence $\backslash \mathbf{y}$
- conditioned on some input sequence $\backslash \mathbf{x} = \backslash \mathbf{x}_{(1)} \ldots \backslash \mathbf{x}_{(')}$

$$\backslash \mathbf{pr} \backslash \hat{\mathbf{y}}_{\backslash \mathrm{tp}} | \backslash \mathbf{x}_{(1)} \ldots \backslash \mathbf{x}_{\backslash \mathrm{tp}}$$

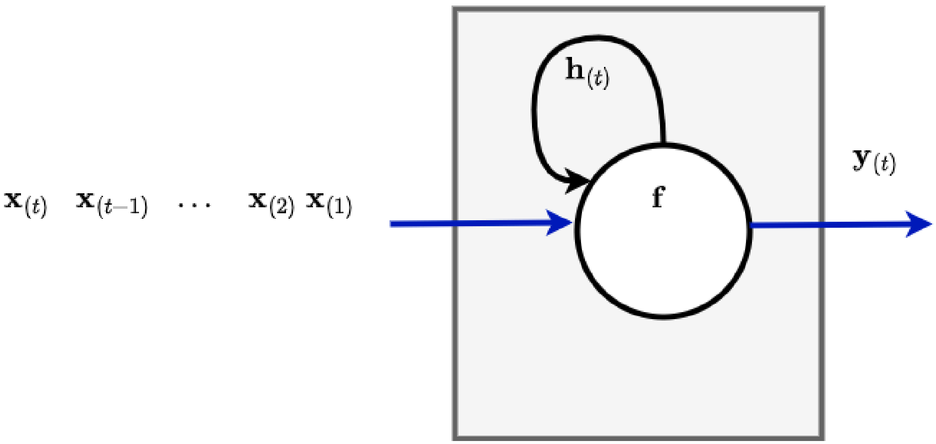
# RNN Loop architecture

- Uses a "latent state" that is updated with each element of the sequence, then predict the output

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

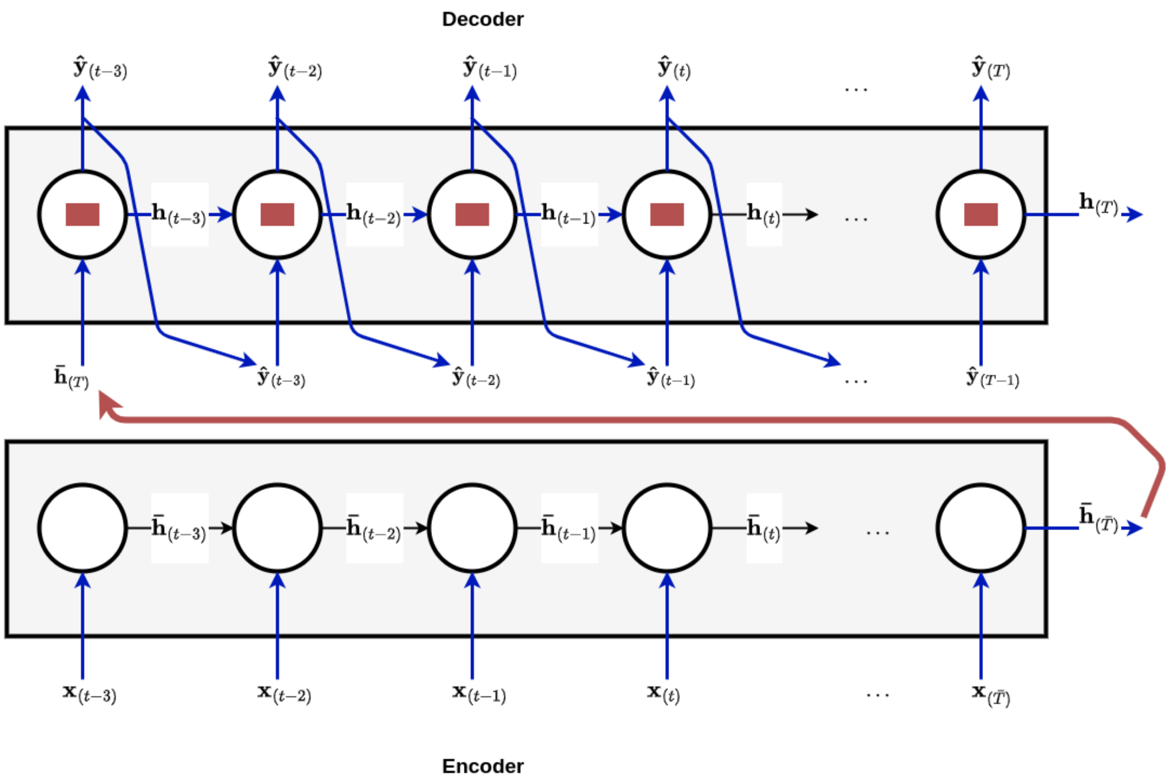
$$\backslash\mathrm{pr}\backslash\hat{\mathrm{y}}_{\backslash\mathrm{tp}} | \backslash\mathrm{h}_{\backslash\mathrm{tp}} \quad \text{prediction contingent on latent variable}$$

**Loop with latent state**



# Original Encoder/Decoder architecture

RNN Encoder/Decoder without Attention
Bottleneck



Critique

- bottleneck
    - *all* information about input $\backslash\mathbf{x}$ passes through out of Encoder (red line)
    - and must be carried over to every iteration of the Decoder loop (red box)
- loop architecture for Encoder and Decoder
    - dependency: horizontal line carrying latent state across time

# Cross-Attention: removing the bottleneck

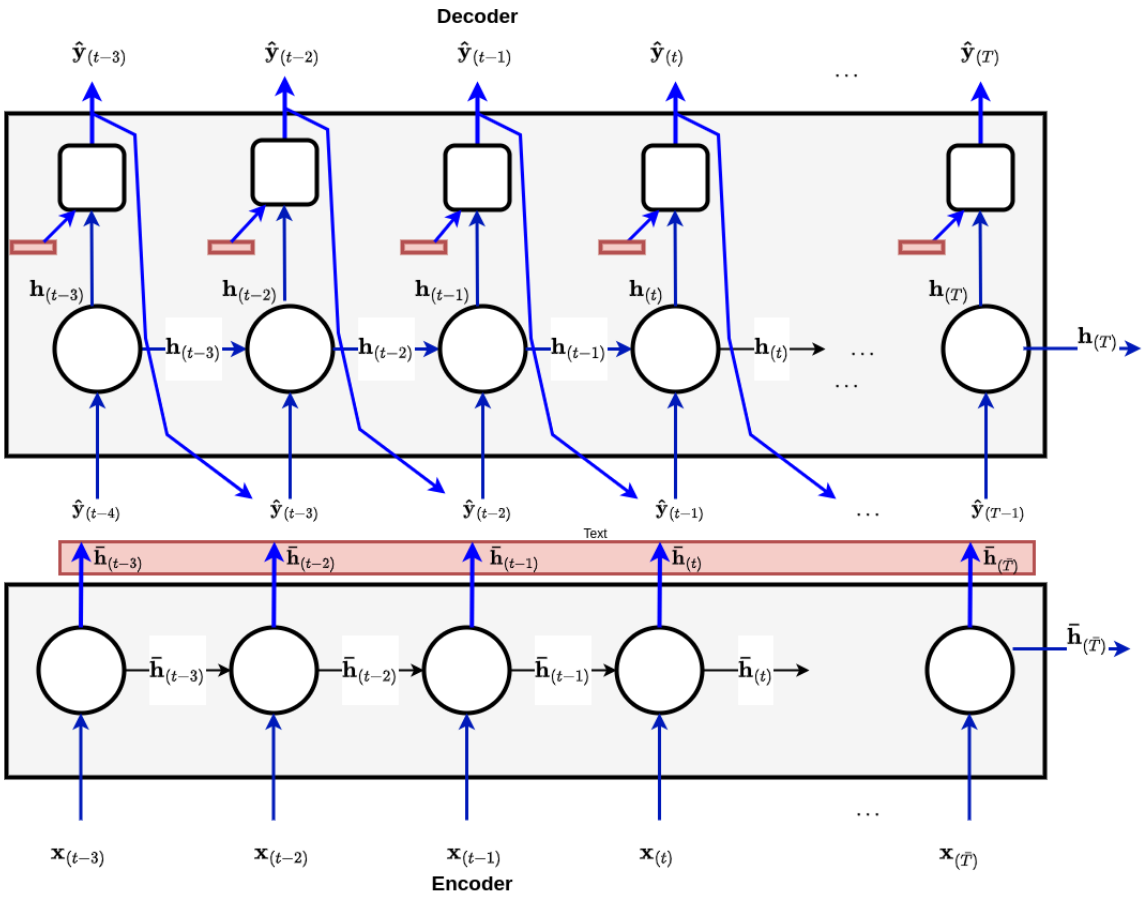
We removed the bottleneck via *Cross Attention*

- Decoder has *direct access* to **all** outputs (i.e., Latent sates) of the Encoder
    - each Encoder output is proxy for a prefix of the input

The pink box is the sequent of Encoder outputs

$$\backslash\bar{\mathbf{h}}_{(1:\bar{T})}$$

## RNN Encoder/Decoder with Cross Attention



# Encoder Self-Attention: removing the Encoder loop

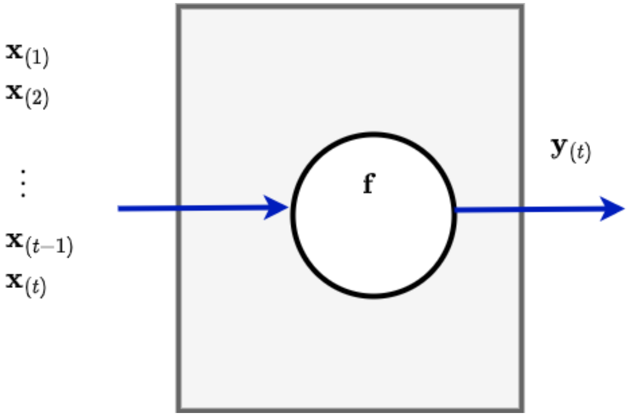
There is an alternative to the loop architecture for processing sequences

- the direct function approach

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

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

**Direct function**



Can output *all* elements of sequence $\backslash\hat{\mathbf{y}}$ *simultaneously*

- each output position is independent of previous output
- only dependent on input

We removed the "loop" architecture of the Encoder by using the direct function approach

- the mechanism enabling each position of the Encoder output to *attend* to the entire sequence $x$ is called *Self-Attention*
    - Notice: no dependency arrow between circles in the Encoder
- Encoder output is a direct function of **all** positions in the input
    - all Encoder output positions can be computed *in parallel*

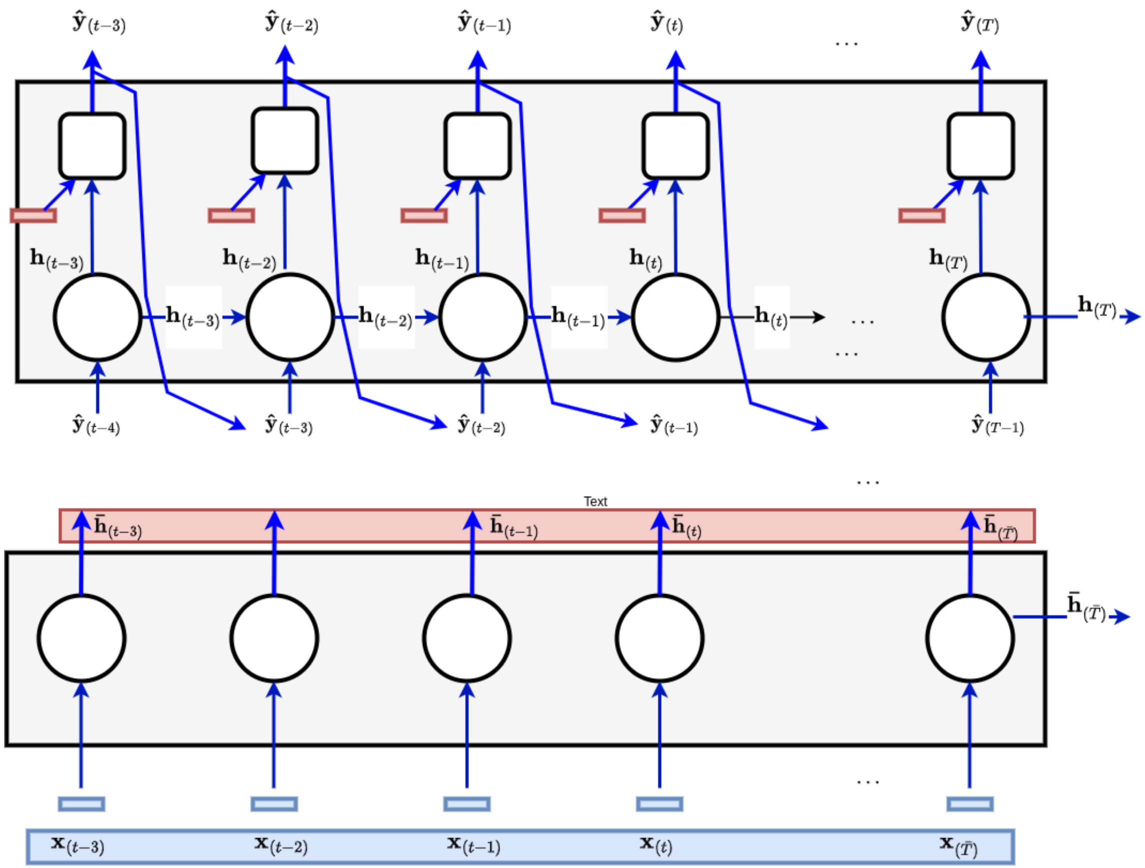
The blue box represents the *entire* input sequence

$$\backslash\mathbf{x}_{(1:\bar{T})}$$

We no longer refer to the Encoder output as a Latent state

- no more loop !

## RNN Encoder/Decoder with Cross Attention/Decoder Self Attention



Observe that

- by removing the looping architecture from the Encoder
- the Encoder is no longer called an RNN

# Masked Self Attention

With *unmasked* Self Attention

- Encoder output $\backslash\bar{\mathbf{h}}_{\backslash\mathrm{tp}}$ at position
- is a function of **all** inputs $\backslash\mathbf{x}_{(1:\bar{T})}$
  - including positions after

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

- as in our motivating example

For certain tasks (not so for our motivating example), 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

- we use a mask to hide inputs from position  onwards so that
- output $\backslash\bar{\mathbf{h}}_{\backslash\mathrm{tp}}$ at position
- is a function only of **preceding** inputs $\backslash\mathbf{x}_{(1:-1)}$

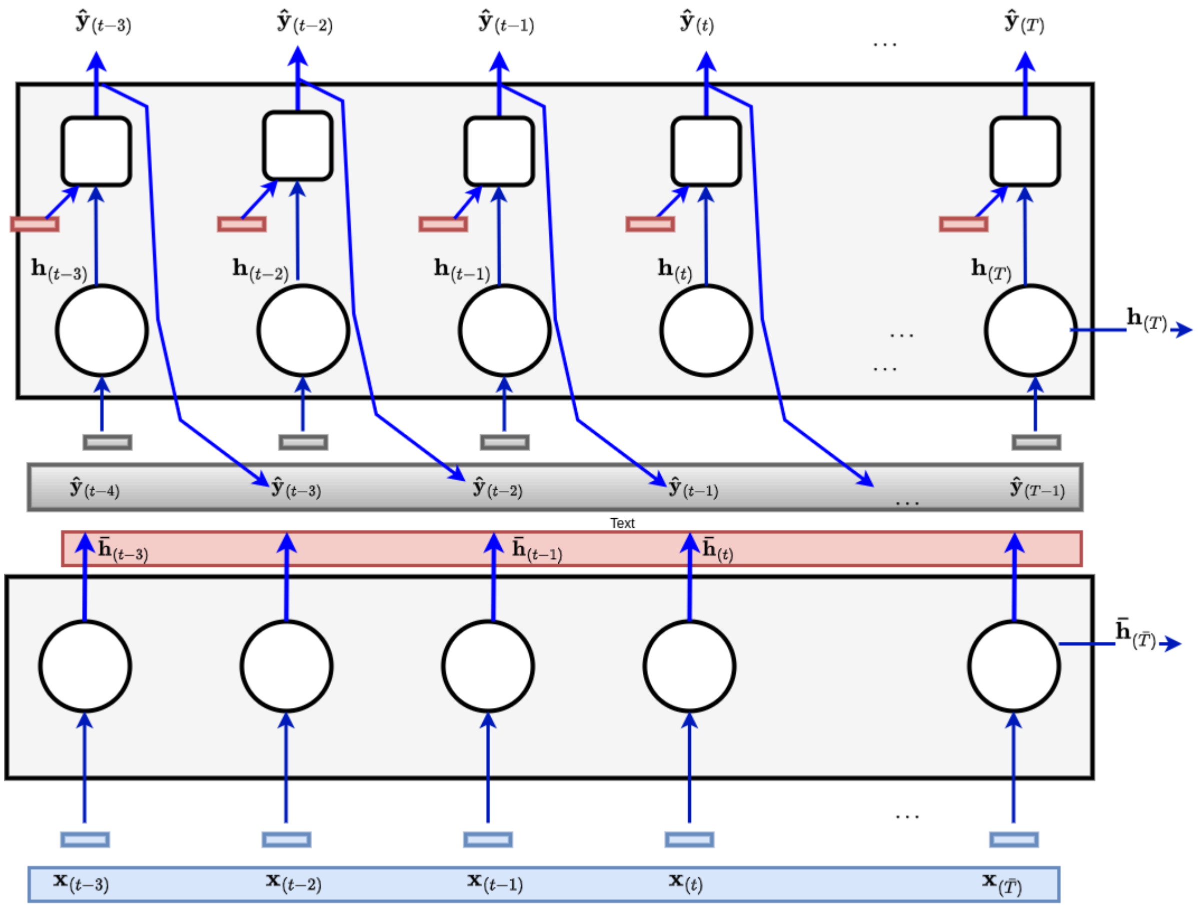
We will see the use of masking in the next section.

# Causal Masked Self Attention: removing the Decoder loop

Finally we remove the loop architecture for the Decoder as well using a different "flavor" of Self-Attention

- Masked Self-Attention.

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



The grey box represents the *entire* output sequence

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

From this diagram: it appears that

- the Encoder/Decoder can produce output $\backslash\hat{\mathrm{y}}_{\backslash\mathrm{tp}}$
- while attending to outputs *that have not yet been generated* at the start of step
$$\backslash\hat{\mathrm{y}}_{(\texttt{:T})}$$
- "looking into the future"

That is, it is computing

$$\backslash\mathrm{prc}\backslash\hat{\mathrm{y}}_{\backslash\mathrm{tp}}\,\backslash\hat{\mathrm{y}}_{(1:T)}$$

What is going on ?

# Teacher forcing at training time

An explanation of this strange behavior is that the behavior of the model is *different*

- at training time
- versus at test/inference time

*Teacher Forcing* alters the training behavior in order to improve the ability of a model to learn.

Let's examine Teacher Forcing (Teacher_Forcing.ipynb) in depth.

# Masked attention

Hopefully it is clear that, regardless of whether we are computing $\color{red}{\hat{\backslash\mathbf{y}}_{\backslash\mathrm{tp}}}$

- at training time
- at inference time

the computation should depend only on positions $1: -1$ of the output.

- can't peek into the future

To enforce this

- we *mask* the outputs
- so that only positions $1 : -1$ are visible when generating output position

The general mechanism of hiding some inputs is called

- **Masked Self-Attention**

The specific masking of only future positions is called

- **Causal Masked Self-Attention** of **Causal Self-Attention**

In [2]:

```
print("Done")
```

```
Done
```