Encoder/Decoder architecture

Two RNN's

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

RNN's process sequences using the "loop architecture"

Consider the task of

- constructing the *next* element $\hat{\mathbf{y}}_{(t)}$ of sequence $\mathbf{y}$
- conditioned on some input sequence $\mathbf{x} = \mathbf{x}_{(1)} \dots \mathbf{x}_{(t')}$

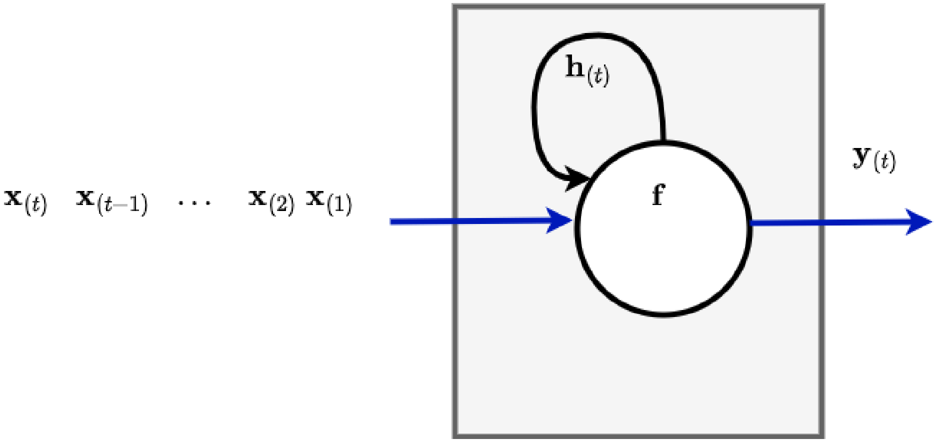
$$p(\hat{\mathbf{y}}_{(t)} | \mathbf{x}_{(1)} \dots \mathbf{x}_{(t)})$$

RNN Loop architecture

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

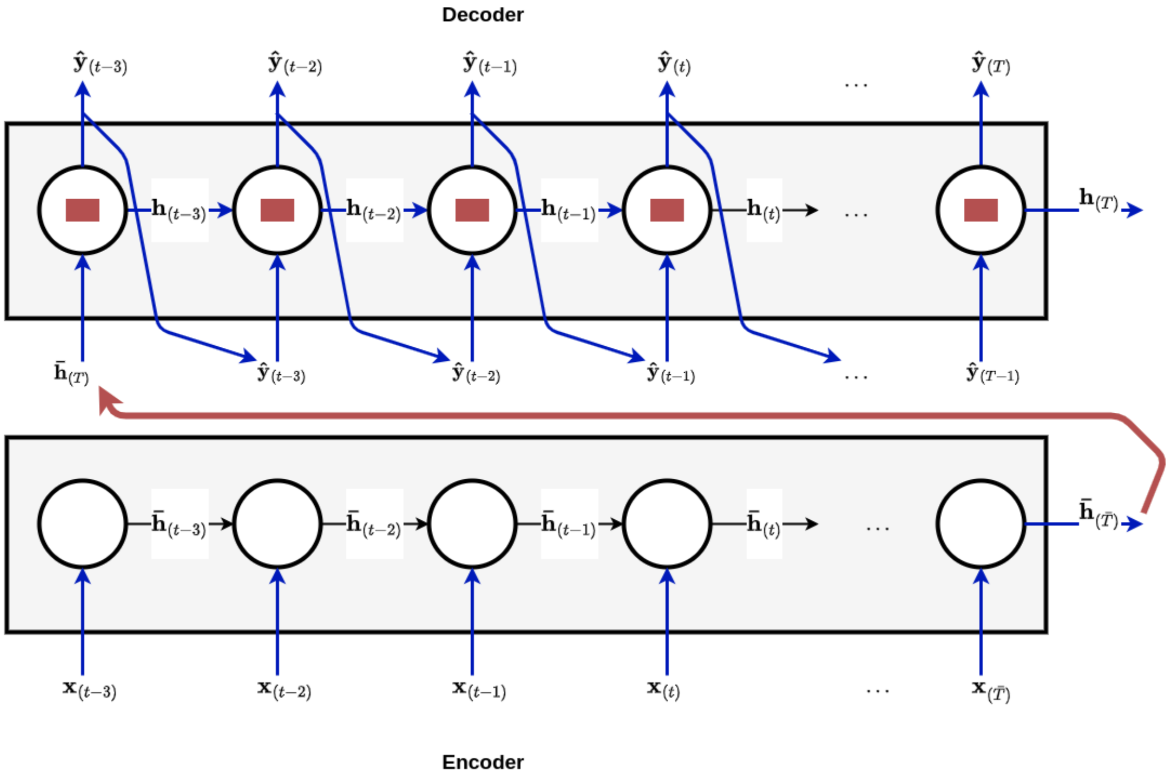
$$
\begin{array}{ll}
p(\mathbf{h}_{(t)} | \mathbf{x}_{(t)}, \mathbf{h}_{(t-1)}) & \text{latent variable } \mathbf{h}_{(t)} \text{encodes } [\mathbf{x}_{(1)} \ldots \mathbf{x}_{(t)}] \\
p(\hat{\mathbf{y}}_{(t)} | \mathbf{h}_{(t)}) & \text{prediction contingent on latent variable}
\end{array}
$$

Loop with latent state



Original Encoder/Decoder architecture

RNN Encoder/Decoder without Attention
Bottleneck



Critique

- bottleneck
    - *all* information about input $\mathbf{x}$ 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

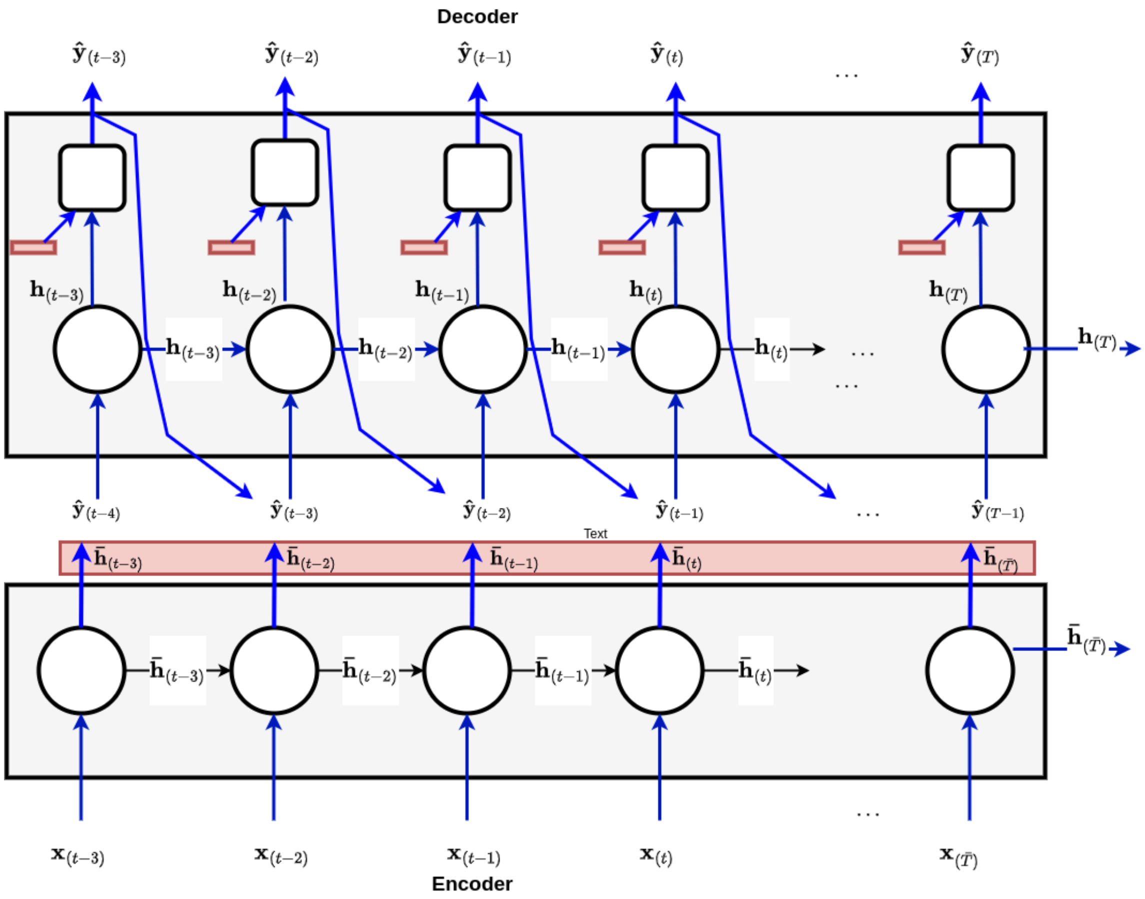
We removed the bottleneck via *Cross Attention*

- 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

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

RNN Encoder/Decoder with Cross Attention



Encoder Self-Attention: removing the Encoder loop

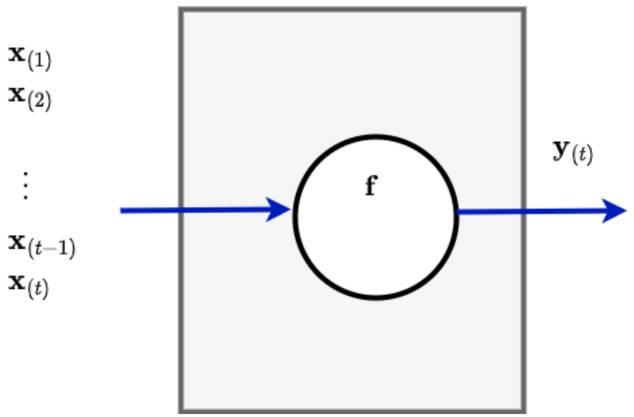
There is an alternative to the loop architecture for processing sequences

- the direct function approach

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

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

Direct function



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

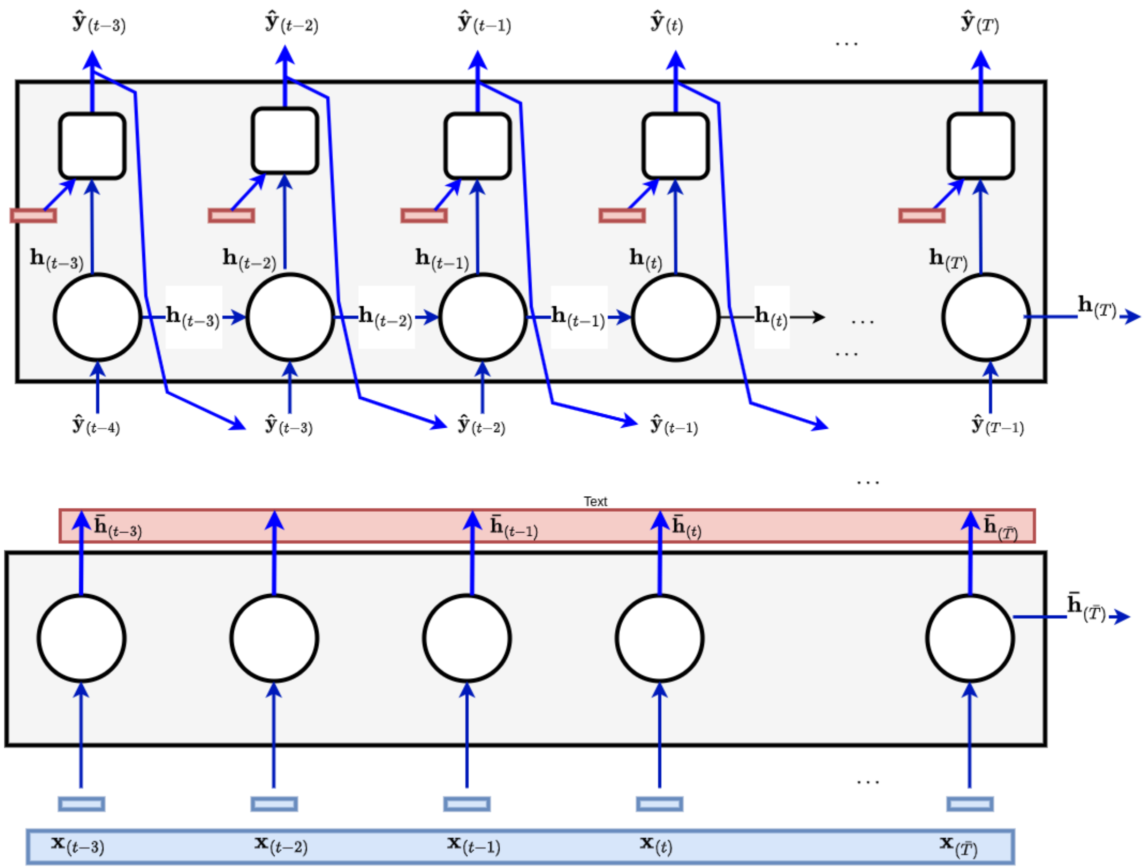
The blue box represents the *entire* input sequence

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

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

- no more loop !

RNN Encoder/Decoder with Cross Attention/Decoder Self Attention



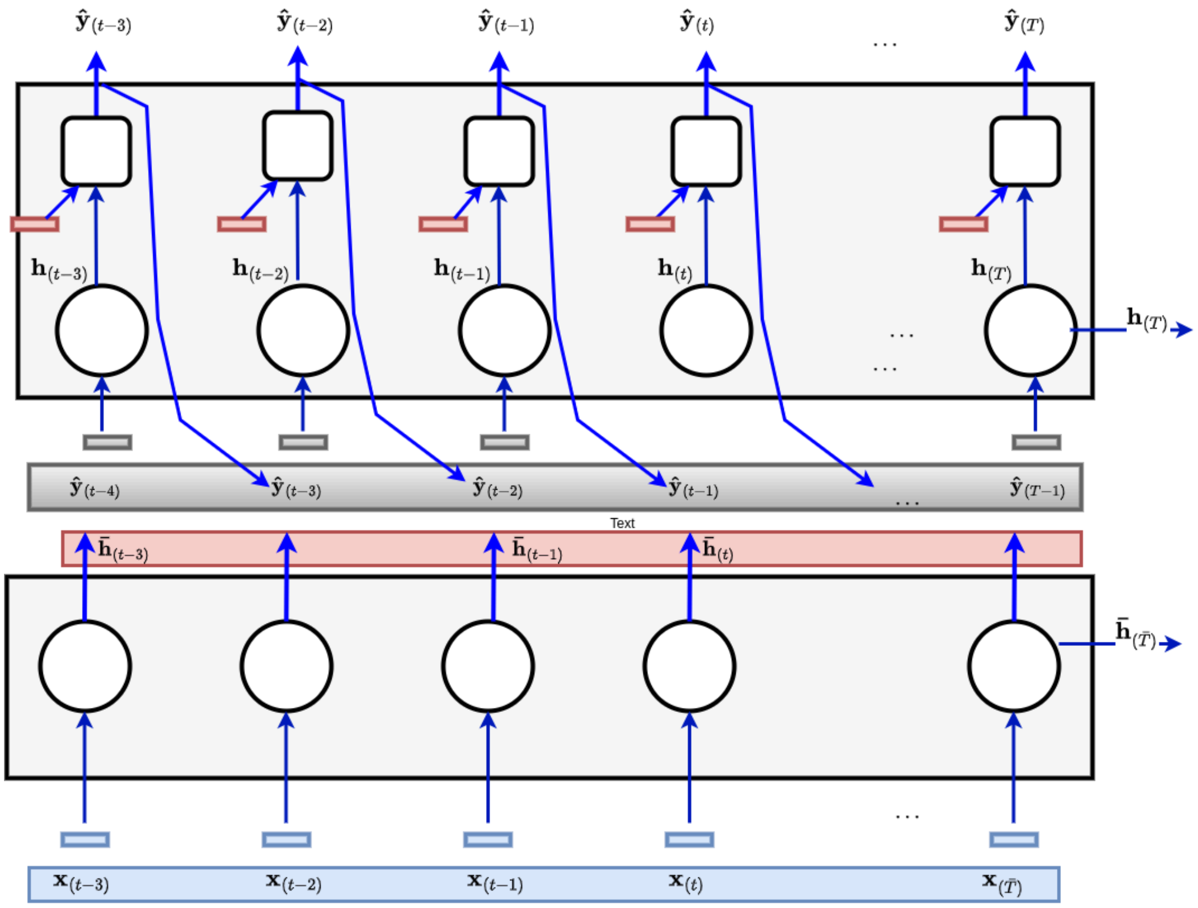
Decoder Masked Self Attention: removing the Decoder loop

Finally we remove the loop architecture for the Decoder as well using Self-Attention

The grey box represents the *entire* output sequence

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

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



Now

- the output sequence $\hat{\mathbf{y}}$ is built iteratively (auto-regressively)
- units work in parallel
- each iteration outputs *all* positions
$$\hat{\mathbf{y}}_{(1:T)}$$
    - including ones whose full inputs have not been defined yet!
    - $\hat{\mathbf{y}}_{(t)}$ is not defined until iteration t

This is confusing !

The point is we don't output position t to the user until iteration t

We certainly don't want $\hat{\mathbf{y}}_{(t)}$ to change on iterations $t' > t$

- don't want future outputs $\hat{\mathbf{y}}_{(t')}$ for $t' \ge t$ to affect $\hat{\mathbf{y}}_{(t)}$
- $\hat{\mathbf{y}}_{(t)}$ depends *only* on $\hat{\mathbf{y}}_{(1:t-1)}$

We can ensure this by using **Masked Self Attention**

- position t can only access positions $t' < t$

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

This means that outputs after iteration t *can't effect* $\hat{\mathbf{y}}_{(t)}$

In [2]:

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

```
Done
```