Attention: motivation

Let us consider a task familiar to all of us who have taken standardized exams: Question Answering.

Input consists of two pieces of text

- a paragraph (called the *Context*)
- a Question whose answer can be found in the paragraph

Output is a piece of text that answers the question.

$$
\x = \begin{cases} \text{Context:} & \text{The FRE Dept offers many Spring classes. The students are gı} \\ & \vdots \\ & \text{Professor Perry taught them Machine Learning. The studen} \\ & \vdots \\ & \text{Professor Blecherman led a class in ...} \\ & \vdots \\ \text{Question:} & \text{What did Professor Perry do ?} \end{cases}
$$

$$
\y = \text{Answer: He taught them Machine Learning}
$$

This is an example of a *Sequence to Sequence* task

- briefly encountered when we defined RNN's

Language translation is another common Sequence to Sequence task.

Let us hypothesize how a model might learn to solve this task

- it is only an hypothesis: we don't really know

The model might have generalized

- from seeing many training examples of disparate questions and their answers
- that there is a parameterized *template* for both the Question and the Answer

Question Template

What did Professor `<PROPER NOUN>` *teach in the Spring ?*

Answer Template:

```
<PRONOUN> <VERB> <INIDRECT OBJECT> <OBJECT>
```

where `<PROPER NOUN>, <PRONOUN>, <VERB>` , etc. are *pattern place-holders* parameters.

By using a parameterized template, the model captures

- commonality
- in many different types of questions

In order to produce the answer the model needs to

- Generate the tokens of the Answer Template in order
- Substituting in concrete values for the place-holders
    - by performing a Lookup in the Context in order to obtain these values

We will examine how the Lookup might be performed

- first: by using mechanisms that we have already studied
- subsequently: via a new mechanism called *Attention*

Using an RNN without Attention

Encoder-Decoder architecture: review

For the Question Answering task

- both the Input and Output are sequences
- thus, the task is a Sequence to Sequence task
    - just like: Language Translation

We learned that Recurrent architectures are best-suited for processing sequences.

These architectures

- operate in a "loop"
    - processing one Input or Output token at a time
- utilize **memory** (latent state)
    - necessary because Input/Output sequence lengths are unbounded
    - after processing the token at position
        - the latent state is finite representation of the prefix of the sequence of length

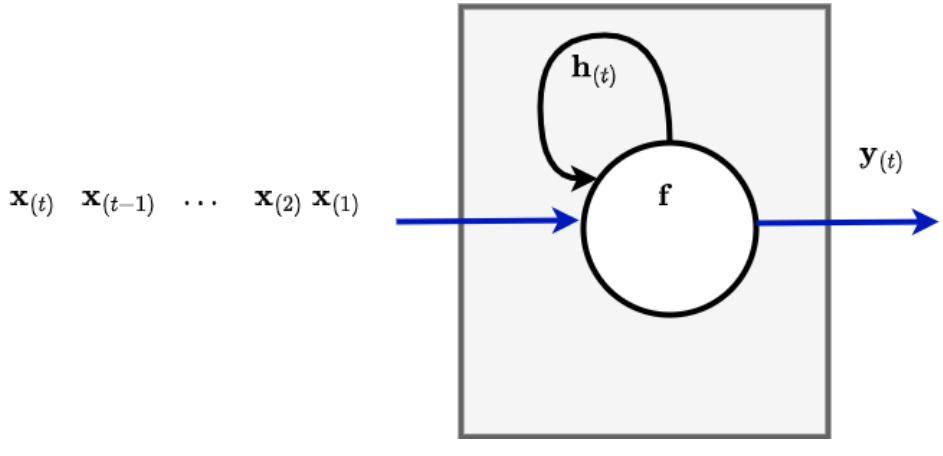
Loop architecture

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

$\pr\h_\tp|\x_\tp, \h_{(-1)}$ latent variable $\h_\tp$ encodes $[\x_{(1)} \dots \x_\tp]$

$\pr\hat{\y}_\tp|\h_\tp$ prediction contingent on latent variable

Loop with latent state

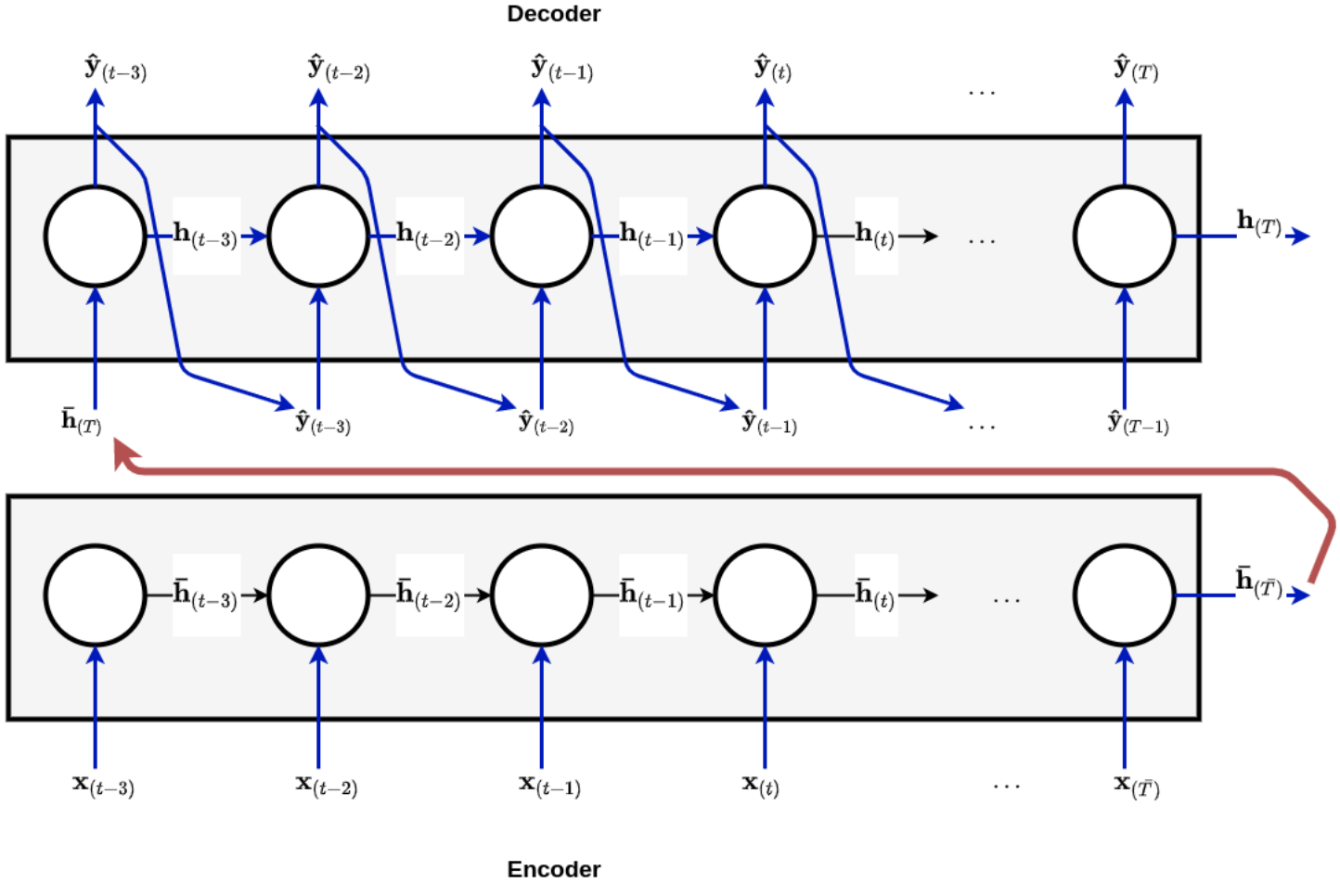


A common architecture for Sequence to Sequence tasks is the Encoder-Decoder:

- The Encoder is an RNN
  - Acts on input sequence $\left[\x_{(1)} \dots \x_{(\bar{T})}\right]$
  - Producing a sequence of latent states $\left[\bar{\h}_{(1)}, \dots, \bar{\h}_{(\bar{T})}\right]$
    - latent state $\boxed{\bar{\h}_\tt}$ is a summary of $\left[\x_{(1)} \dots \x_{\tp}\right]$

The Decoder

- Acts on the *final* Encoder latent state $\bar{\mathbf{h}}_{(\bar{T})}$
  - which summarizes the entire input sequence $\mathbf{x}$
  - Producing a sequence of latent states $[\mathbf{h}_{(1)}, \ldots, \mathbf{h}_{(T)}]$
    - latent state $\mathbf{h}_{\text{tp}}$ is response for generating output token $\hat{\mathbf{y}}_{\text{tp}}$
  - Thus outputting a sequence $[\hat{\mathbf{y}}_{(1)}, \ldots, \hat{\mathbf{y}}_{(T)}]$


Decoder
$\hat{\mathbf{y}}_{(t-3)}$
$\hat{\mathbf{y}}_{(t-2)}$
$\hat{\mathbf{y}}_{(t-1)}$
$\hat{\mathbf{y}}_{(t)}$
$\hat{\mathbf{y}}_{(T)}$
$\mathbf{h}_{(t-3)}$
$\mathbf{h}_{(t-2)}$
$\mathbf{h}_{(t-1)}$
$\mathbf{h}_{(t)}$
$\mathbf{h}_{(T)}$
$\bar{\mathbf{h}}_{(T)}$
$\hat{\mathbf{y}}_{(t-3)}$
$\hat{\mathbf{y}}_{(t-2)}$
$\hat{\mathbf{y}}_{(t-1)}$
$\hat{\mathbf{y}}_{(T-1)}$
$\bar{\mathbf{h}}_{(t-3)}$
$\bar{\mathbf{h}}_{(t-2)}$
$\bar{\mathbf{h}}_{(t-1)}$
$\bar{\mathbf{h}}_{(t)}$
$\bar{\mathbf{h}}_{(\bar{T})}$
$\mathbf{x}_{(t-3)}$
$\mathbf{x}_{(t-2)}$
$\mathbf{x}_{(t-1)}$
$\mathbf{x}_{(t)}$
$\mathbf{x}_{(\bar{T})}$
Encoder

Auto-regressive Decoder

The Decoder usually works in an *Auto-Regressive* manner

- output $\hat{\mathbf{y}}_{\text{tp}}$ of step
- becomes part of the input $\hat{\mathbf{y}}_{(1:)}$ of step **+1**
- see illustration above

The final output sequence is thus constructed

- single element at a time
- with a sequential dependency on all prior elements of the output sequence

Decoder output $\hat{\mathbf{y}}_{\tp}$

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

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

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

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

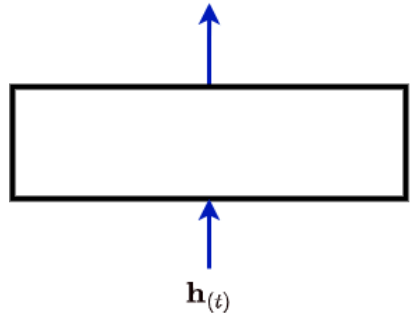
For clarity: we will omit this additional NN from our diagrams until it becomes necessary

Here is what the additional NN looks like:

Decoder output transformation: No attention

Decoder

$\hat{\mathbf{y}}_{(1)}, \hat{\mathbf{y}}_{(2)}, \cdots, \hat{\mathbf{y}}_{(T)}$



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

How does the Decoder perform Lookup (without Attention) ?

Suppose the Decoder has already output

$$\hat{\mathbf{y}}_{([1:3])} = \text{He taught them}$$

It must subsequently output

$$\hat{\mathbf{y}}_{([4:5])} = \text{Machine Learning}$$

In order to do this

- it must Lookup "Machine Learning" in the Context, resulting in

$$\begin{aligned} D(\mathbf{h}_{(4)}) &= \text{Machine} \\ D(\mathbf{h}_{(5)}) &= \text{Learning} \end{aligned}$$

But D is conditioned on the single input $\mathbf{h}_{\text{tp}}$.

Thus, in order for $D(\mathbf{h}_{(4)})$ to be equal to "Machine"

- this information must somehow be encoded in $\mathbf{h}_{(4)}$

How did it get there ?

All "knowledge" from the Context must be transfered from Encoder to Decoder

- through final Encoder state $\bar{\mathrm{h}}_{(\bar{T})}$
- which in turn was encoded in all Encoder states $\bar{\mathrm{h}}_{(\bar{\ })}$ $\mathrm{for}\ ' \geq \bar{\mathrm{p}}$
    - where $\bar{p}$ is the position withing sequence x of the word "Machine"

We can hypothesize that the final Encoder latent state $\bar{\mathbf{h}}_{\bar{T}}$

- encodes a Dictionary (key/value pairs)
- mapping Place-holder names to Concrete values
- the dictionary is built incrementally by prior latent states of the Encoder

Answering questions using Attention

Input Tokens: Professor Perry taught them Machine Learning [CLS]

$$\bar{\mathbf{h}}_{(1)}, \ldots, \bar{\mathbf{h}}_{(\bar{T})} = \left\{ \begin{array}{l} \text{Subject: Professor Perry} \\ \text{Pronoun: he} \\ \text{Object: Machine Learning} \\ \text{Indirect Object: them} \\ \text{Verb: taught} \end{array} \right\}$$

This dictionary is passed to the Decoder via the single connection from Encoder to Decoder

- and must be carried forward by the Decoder
- through Decoder states $[\backslash \mathbf{h}_{(1)}, \ldots, \backslash \mathbf{h}_{(4)}]$
- in order to make the dictionary available to subsequent latent states of the Decoder

We further hypothesize that the Decoder

- performs Lookups
- by using the Decoder latent state $\backslash \mathrm{h}_{\backslash \mathrm{tp}}$
  - as a *query* that matches against the keys of the Dictionary
  - in order to obtain the Concrete value required to produced output token at position

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

Query performing a Lookup in the Dictionary

Question: What did Professor Perry do ?

Answer template: <PRONOUN> <VERB> <INDIRECT OBJECT> <OBJECT> [CLS]

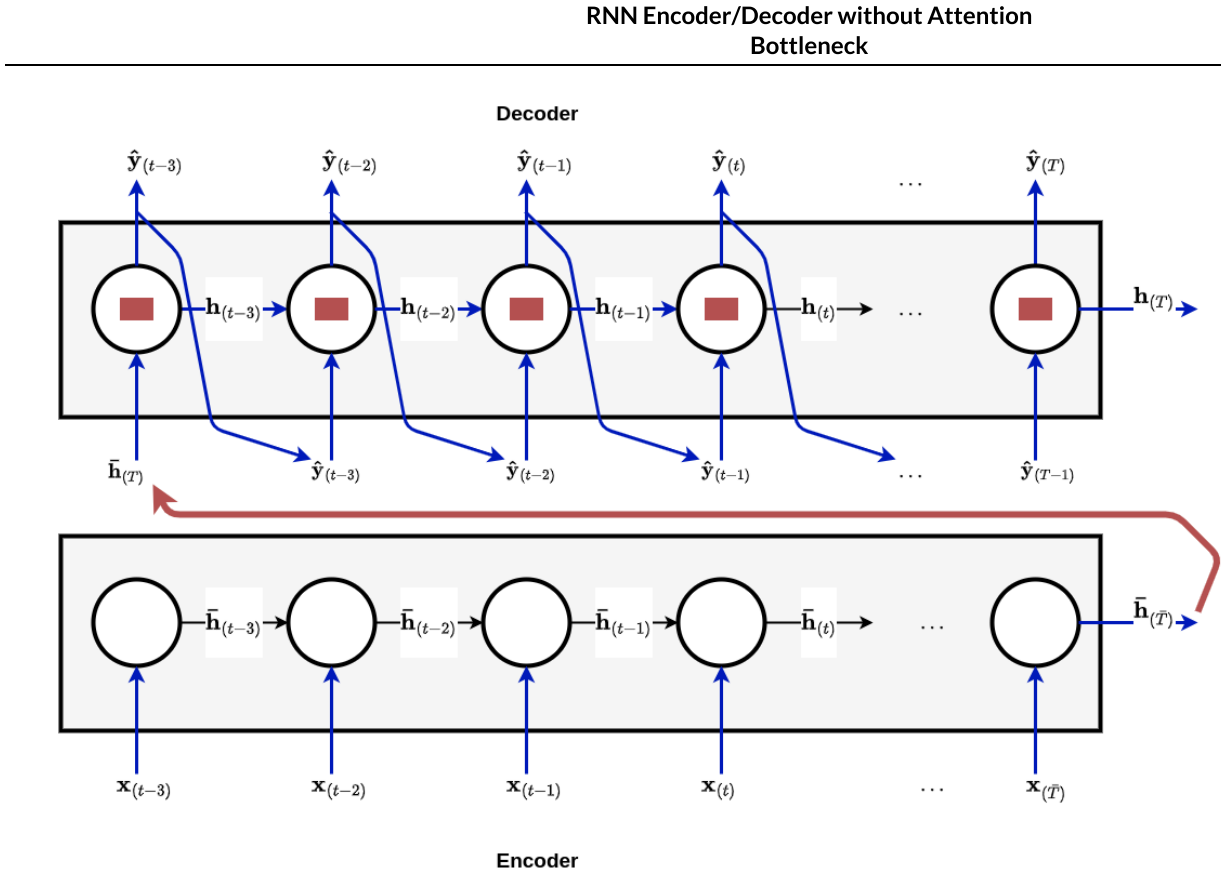
Queries:

$\mathbf{h}_{(\mathbf{Pronoun})}$ = {Pronoun: ?}

$\mathbf{h}_{(\mathbf{Verb})}$ = { Verb: ? }

Answer: He taught them Machine Learning [CLS]

Here is a picture describing this hypothetical functioning.



Subject: Professor Perry

Connecting the Encoder and Decoder through the "bottleneck" of $\bar{\mathbf{h}}_{(\bar{T})}$ thus burdens the

- Encoder: passing knowledge forward to the bottleneck
- Decoder: passing knowledge from the bottleneck
    - to all subsequent positions (red box within each circle)

This results in an inefficient use of the model's latent state variable

- In addition to
    - the Encoder and Decoder allocating some of the model's latent state for "control"
    - guiding the loop that processes the Input, or generates the output positions in the template
- It must **also** allocate some of the model's latent state for "knowledge storage"
    - in order to Lookup the concrete value corresponding to a place-holder in the Output template

Cross Attention

Reference

[Neural Machine Translation by Jointly Learning To Align and Translate (https://arxiv.org/pdf/1409.0473.pdf) paper that introduced Attention (https://arxiv.org/pdf/1409.0473.pdf)](https://arxiv.org/pdf/1409.0473.pdf)

The flaw in the Encoder-Decoder without Attention is

- the input $\mathbf{x}$ is processed *only once*
- by the Encoder
- which has to summarize it in $\bar{\mathbf{h}}_{(\bar{T})}$

We will introduce a mechanism called *Attention*

- that allows the input sequence to be *re-visited* at each time step of Output generation

This will result in a cleaner separation between control memory and input memory

Attention allows the Decoder

- to directly access all of the Encoder latent states $\bar{\mathbf{h}}_{(1)} \ldots \bar{\mathbf{h}}_{(\bar{T})}$
- at each time step of the Decoder

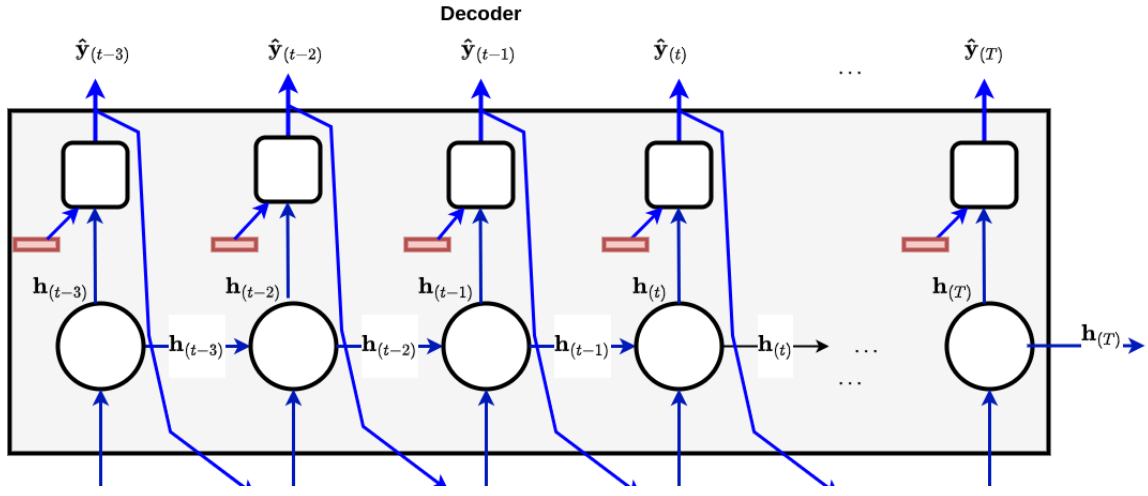
Thus, there is no need

- for an Encoder to create a full dictionary as the final Encoder latent state $\bar{\mathbf{h}}_{(\bar{T})}$
- for the Decoder to keep the dictionary in all it's latent states $\mathbf{h}_{(1)} \ldots \mathbf{h}_{(T)}$

Here is a picture of an Encoder/Decoder augmented with Attention

- we have add an additional box to the diagram for the NN that implements the function D
    - that maps $\backslash \mathbf{h}_{\backslash \mathrm{tp}}$ to $\backslash \mathbf{y}_{\backslash \mathrm{tp}}$

RNN Encoder/Decoder with Attention


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

This is one "flavor" of Attention known as *Cross Attention*

- one component (the Decoder) uses as input (*attends to*) the output of another component (the Encoder)

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

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

Notice that the final Encoder latent state $\bar{\mathbf{h}}_{(\bar{T})}$ is **no longer** connected to the Decoder.

What is going on inside the "box" implementing function D that we added at each time step ?

The box's input at step

- the Decoder latent state $\mathbf{h}_{\text{tp}}$
- the collection of Encoder latent states $\bar{\mathbf{h}}_{(1)} \ldots \mathbf{h}_{(\bar{T})}$
    - the red box in the above diagram

That is, it is computing a $\hat{\mathbf{y}}_{\text{tp}}$ that is a function of both $\mathbf{h}_{\text{tp}}$ and $\bar{\mathbf{h}}_{(1)} \ldots \mathbf{h}_{(\bar{T})}$

$$\hat{\mathbf{y}}_{\text{tp}} = D(\mathbf{h}_{\text{tp}}, [\bar{\mathbf{h}}_{(1)} \ldots \mathbf{h}_{(\bar{T})}])$$

Self-Attention: removing the Encoder loop

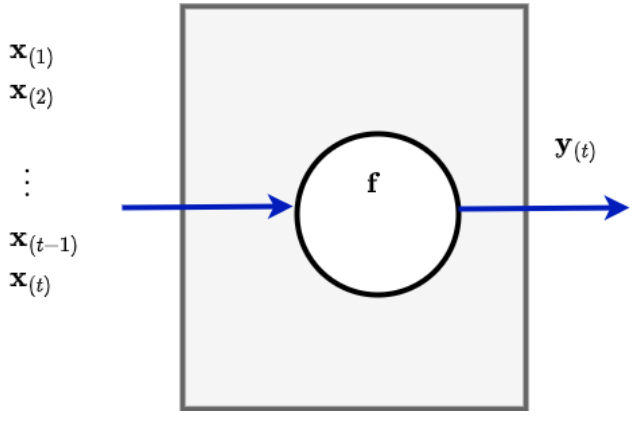
There is an alternative to the loop architecture for processing sequences

- the direct function approach

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

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

Direct function



Note that output at position t *no longer depend* on latent state from position $(t-1)$

- we have removed the loop !

Thus, the Encoder can output *all* elements of sequence $\bar{\mathbf{h}}$ *simultaneously*

- n.b. the Encoder output sequence will still be denoted

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

even though the outputs are no longer *latent states*

- 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*

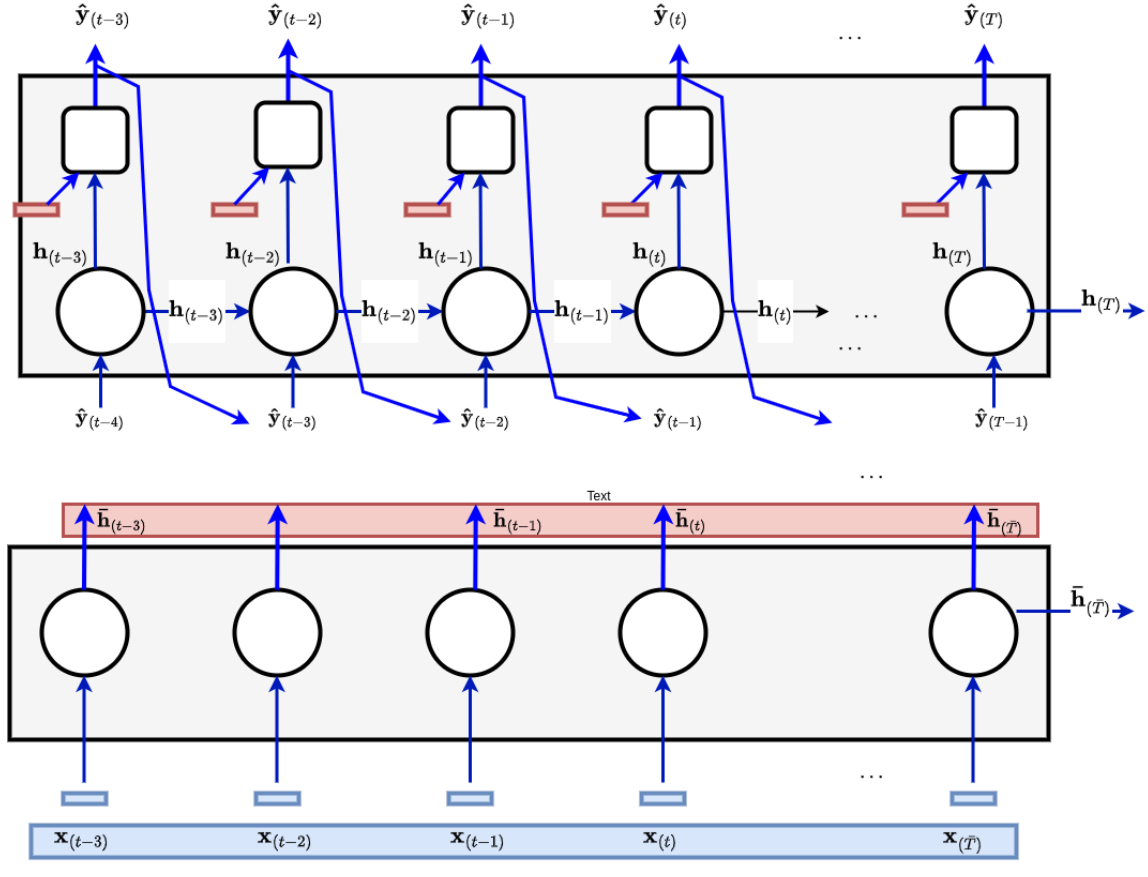
The blue box represents the *entire* input sequence

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

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

- no more loop !

RNN Encoder/Decoder with Cross Attention/Decoder Self Attention



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 $\bar{\mathrm{h}}_{\mathrm{tp}}$ at position
- is a function only of **preceding** inputs $\mathrm{x}_{(1:-1)}$

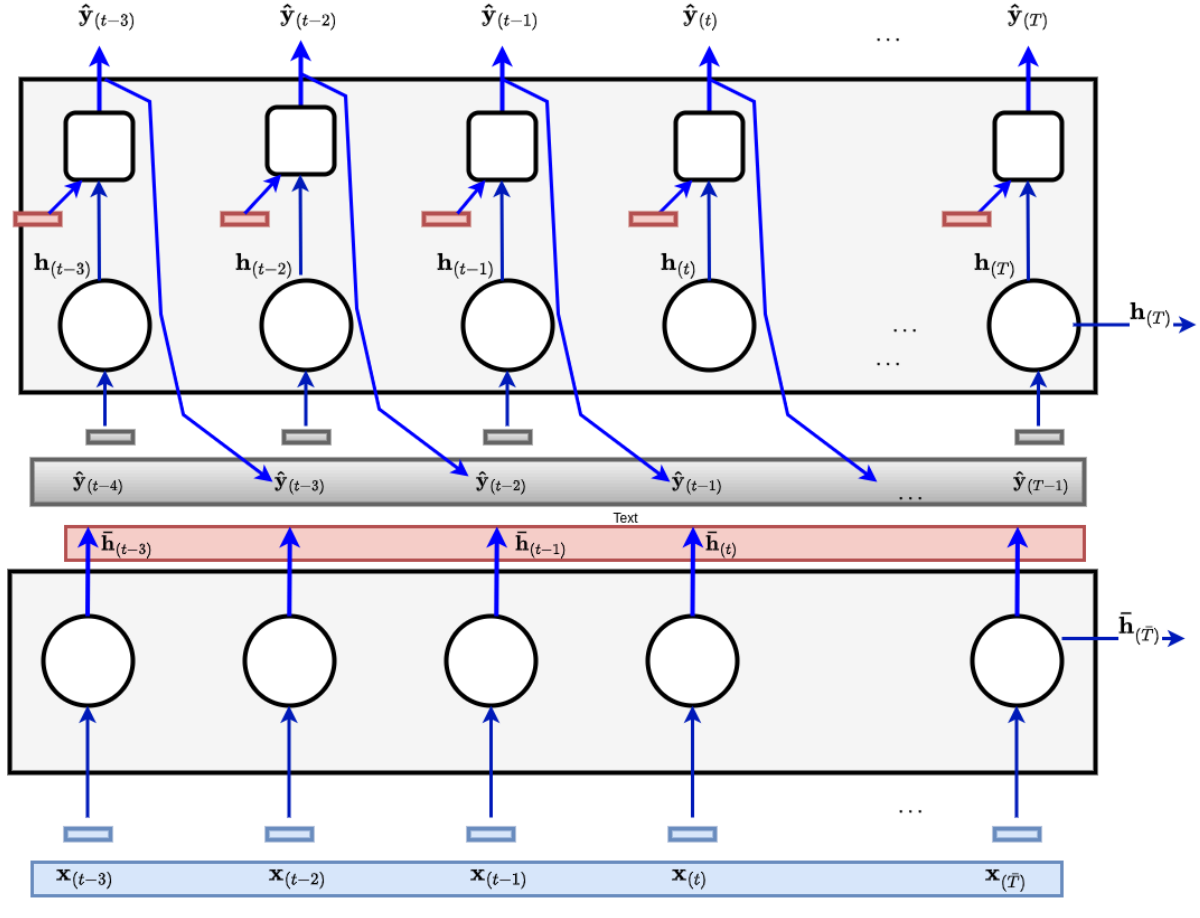
We will see the use of masking in the next section.

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.

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



The grey box represents the *entire* output sequence

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

From this diagram: it appears that

- the Encoder/Decoder can produce output $\hat{\mathbf{y}}_{\mathrm{tp}}$
- while attending to outputs *that have not yet been generated* at the start of step

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

- "looking into the future"

That is, it is computing

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

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 $\hat{\mathbf{y}}_{\text{tp}}$

- 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**

Visualizing Attention

We can illustrate the behavior of Neural Networks that have been augmented with Attention through diagrams.

- at a particular output position
- we can display the amount of "attention"
- that each position in the input receives

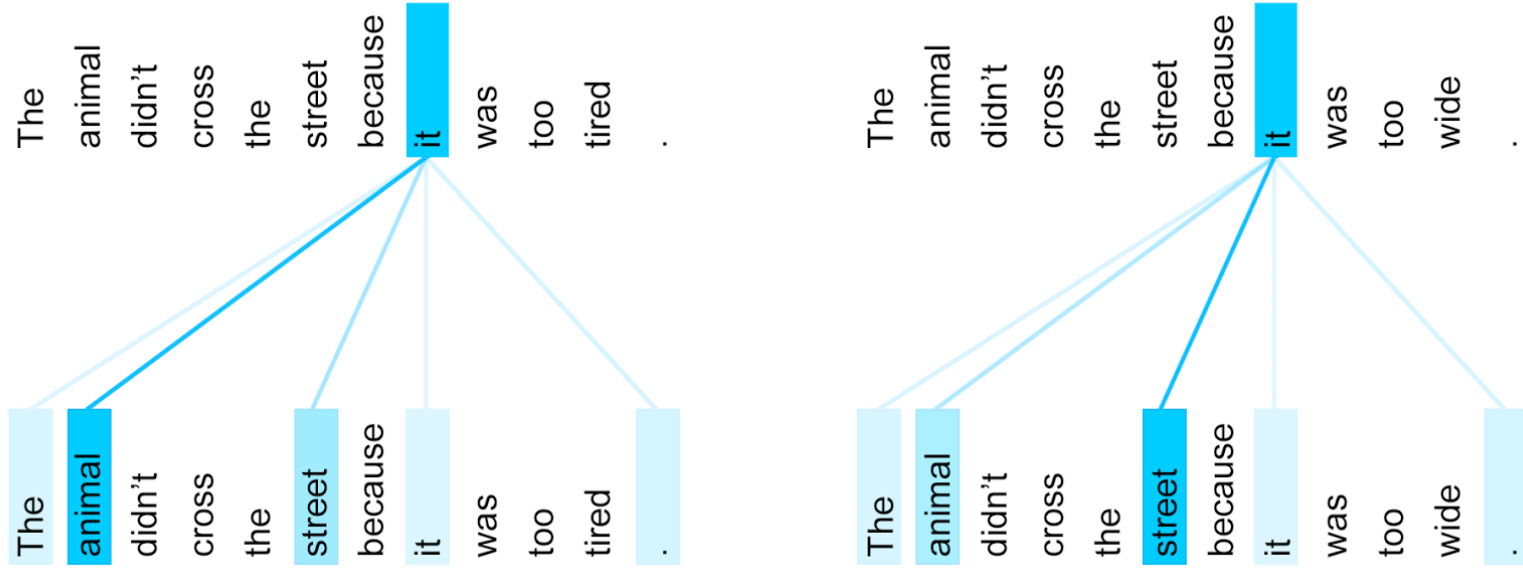
Visualizing Self-Attention

Self Attention can be used to create a Context Sensitive Encoding of words

- The meaning of a word may change depending on the rest of the sentence

We can illustrate this with an example: how the meaning of the word "it" changes

- The thickness of the blue line indicates the attention weight that is given in processing the word "it".



Much of the recent advances in NLP may be attributed to these improved, context sensitive embeddings.

We note that simple Word Embeddings

- also capture "meaning"
- but are *not* sensitive to context

Visualizing Cross Attention

The Entailment task

The *Entailment* task is a binary classification task based on two sentence

- first sentence (the *premise*)
- second sentence (the *hypothesis*)

Classify: Does the Hypothesis logically follow (is *entailed* by) the Premise ?

Here is an illustration of what part of the Premise is attended to as we encounter the Hypothesis

Attention: Entailment

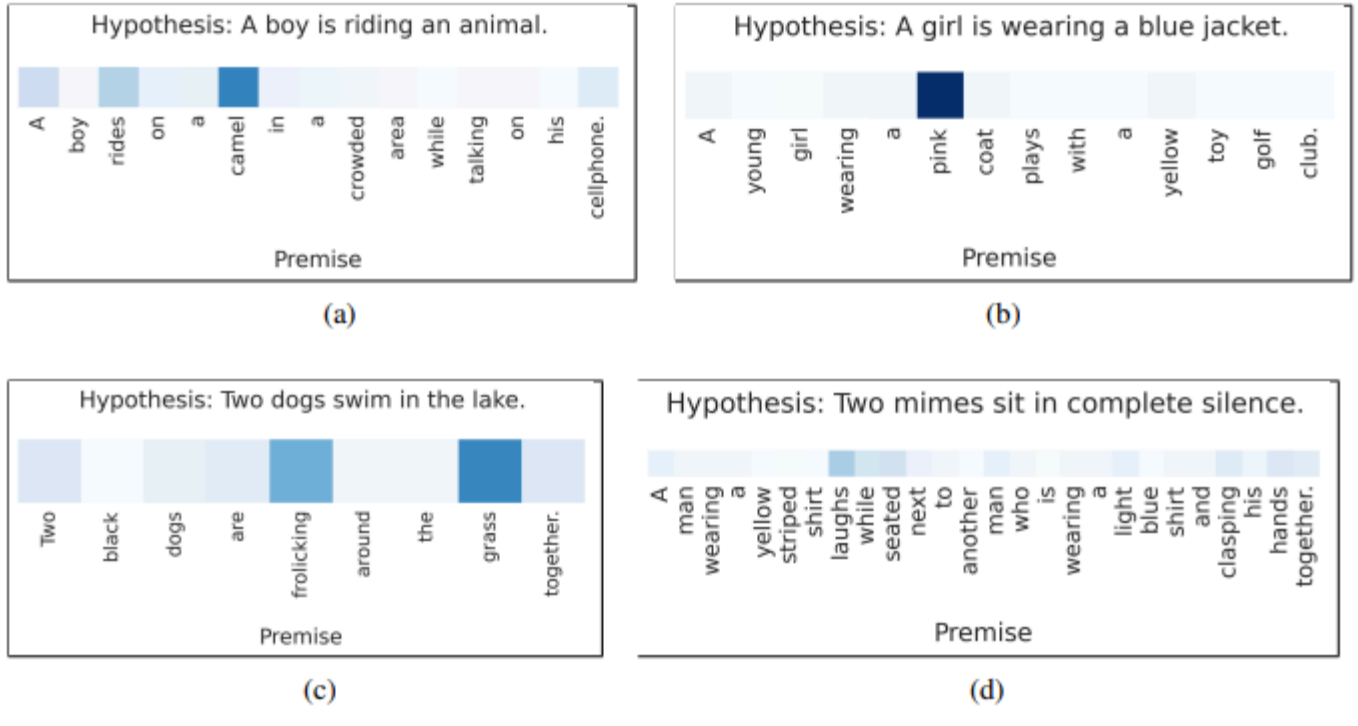


Figure 2: Attention visualizations.

Does the Premise logically entail the Hypothesis.

Attribution: https://arxiv.org/pdf/1509.06664.pdf#page=6 (https://arxiv.org/pdf/1509.06664.pdf#page=6)"

The Date Normalization task

- Source: Dates in free-form: "Saturday 09 May 2018"
- Target: Dates in normalized form: "2018-05-09"

link (https://github.com/datalogue/keras-attention#example-visualizations)

The Image captioning task

- Source: Image
- Target: Caption: "A woman is throwing a **frisbee** in a park."
- Attending over *pixels* **not** sequence

Visual attention

A woman is throwing a **frisbee** in a park.

Attribution: https://arxiv.org/pdf/1502.03044.pdf (https://arxiv.org/pdf/1502.03044.pdf)

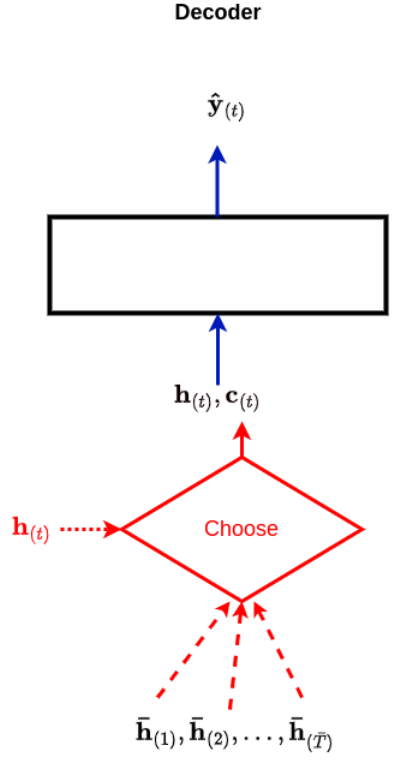
Implementing Attention (Preview): Inside the NN for D

Recall the Decoder conditions output $\hat{\mathbf{y}}_{\tp}$ on

- Decoder state $\mathbf{h}_{\tp}$
- Encoder output sequence $\bar{\mathbf{h}}_{(1:\bar{T})}$

Here is a high-level view of the inner workings of the NN for D:

Decoder output transformation with attention



Inside the box:

- the Decoder latent state $\mathbf{h}_{\text{tp}}$ is used as a *query*
- which is matched against each of the Encoder outputs
- resulting in one Encoder output being chosen as $\mathbf{c}_{\text{tp}}$

The chosen Encoder output $\mathbf{c}_{\text{tp}}$ and Decoder latent state $\mathbf{h}_{\text{tp}}$

- are input to another Neural Network
- which produces output $\hat{\mathbf{y}}_{\text{tp}}$

The "Choose" box implements an *Attention* mechanism, which allows the Decoder

- to **attend to** the part of Input $\mathbf{x}$ (represented via some Encoder latent state $\bar{\mathbf{h}}_{(\bar{\ })}$)
- that is *relevant* for producing $\hat{\mathbf{y}}_{\mathrm{tp}}$
- exactly when it is needed

This seems very natural to a human

- rather than memorizing details (e.g., the big dictionary $\bar{\mathbf{h}}_{(\bar{T})}$ in the architecture without Attention)
- we refer back to the context
- focusing of only the part that is immediately needed

The discussion of the **implementation** of Attention will be deferred to a later module Attention lookup (Attention_Lookup.ipynb).

For now, think of the "Choose" box as a Context Sensitive Memory (as described in the module on Neural Programming (Neural_Programming.ipynb#Soft-Lookup))

- Like a Python `dict`
  - Collection of key/value pairs: $\langle \bar{\mathbf{h}}_{(-)}, \bar{\mathbf{h}}_{(-)} \rangle$
  - Key is equal to value; they are latent states of the Encoder
- But with *soft* lookup
  - The current Decoder state $\mathbf{h}_{tp}$ is presented to the CSM
    - Called the *query*
    - Is matched across each key of the dict (i.e., a latent state $\bar{\mathbf{h}}_{(-)}$)
  - The CSM returns an approximate match of the query to a *key* of the `dict`
    - The distance between the query and each key in the CSM is computed
    - The Soft Lookup returns a *weighted* (by inverse distance) sum of the *values* in the CSM `dict`

Multi-head attention: two heads are better than one

Perhaps when generating the output for position of the output sequence

- we need to attend to *more than one* position of the sequence being attended to
    - need to know both gender and plurality of subject
- that is: we want an Attention layer to output multiple items.

We can attend to n positions

- by creating n separate Attention mechanisms
- each one called a *head*

This behavior is referred to as *Multi-head attention*

This type of behavior is common to many layer types in a Neural Network

- a Dense layer l may produce a vector $\mathrm{y}_{\llp}$ where $n_{\llp} > 1$
- a Convolutional layer l may produce outputs (for each spatial location) for many channels

We have referred to this as layer $\ll$ producing $n_{\llp}$ *features*.

It would be natural for an Attention layer to output many "features" to enable attention to many positions.

In practice, this is sometimes (always ?) not done

- Model architectures (e.g., the Transformer) are simplified when the inputs/outputs of each sub-component
- have the same length
- often denoted as d or d_{model} in the Transformer

When a Transformer needs to attend to n positions

- it uses n Attention heads
- each outputting a vector of length $\frac{d}{n}$
- which are concatenated together to produce a single output of length d

In essence

- we create n "mini-attention" heads
    - we uses queries
    - and input sequences -which are a *fraction* of the original lengths
- the outputs of the n mini-heads can be combined into an output of length d

So multi-head attention is compatible (in terms of shape of input and output) with single head attention.

When we have n heads

- Rather than having one Attention head operating on vectors of length d
    - producing an output of length d (weighted sum of values in the CSM)
- We create n Attention heads operating on vectors (keys, values, queries) of length $\frac{d}{n}$.
    - Output of these smaller heads are values, and hence also of length $\frac{d}{n}$
- The final output concatenates these n outputs into a single output of length d
    - identical in length to the single head
- we project each of these length d vector into vectors of length $\frac{d}{n}$

The picture shows n Attention heads.

Note that each head is working on vectors of length $\frac{d}{n}$ rather than original dimensions d.

- variables with superscript (j) are of fractional length

Details are deferred to the module Attention lookup (Attention_Lookup.ipynb).

Each head j uniquely transforms the query $\mathbf{h}_{\text{tp}}$ and the key/value pairs $\bar{\mathbf{h}}_{(1)} \ldots \bar{\mathbf{h}}_{(\bar{T})}$ being queried.

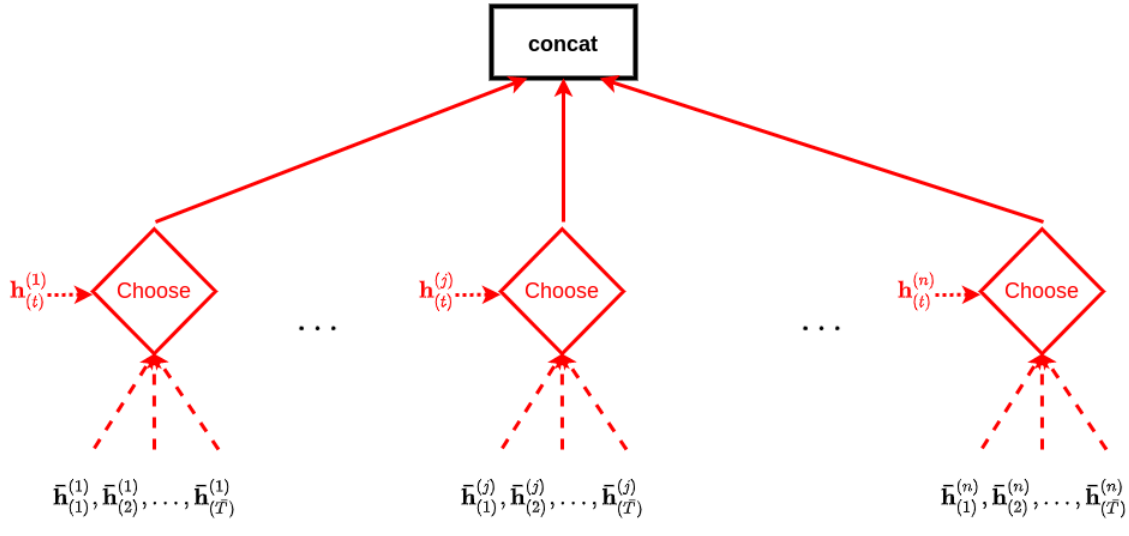
- into $\mathbf{h}^{(j)}_{\text{tp}}$ and the key/value pairs $\bar{\mathbf{h}}^{(j)}_{(1)} \ldots \bar{\mathbf{h}}^{(j)}_{(\bar{T})}$
- all vectors/vector elements with superscripts are of length $\frac{d}{n}$ rather than the original length d

Decoder Multi-head Attention

Per-head query and value

$$\mathbf{h}^{(j)}_{(t)} = \mathbf{W}^{(j)}_{\text{query}} \mathbf{h}_{(t)}$$

$$\bar{\mathbf{h}}^{(j)}_{(t)} = \mathbf{W}^{(j)}_{\text{value}} \bar{\mathbf{h}}_{(t)}$$



Transformers (preview)

Using various flavor of Attention

- we have replaced the Encoder and Decoder RNN's in an Encoder/Decoder architecture
- with a new layer type
    - direct function approach to sequences

This new architecture is the basis for the *Transformer* layer

- a key advance in modern Deep Learning, particularly for NLP
- which we will study in more depth in a subsequent module

Conclusion

We recognized that the Decoder function responsible for generating Decoder output $\hat{\mathbf{y}}_{\tp}$

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

was quite rigid when it ignored argument $\mathbf{s}$.

This rigidity forced Decoder latent state $\mathbf{h}_{\tp}$ to assume the additional responsibility of including Encoder context.

Attention was presented as a way to obtain Encoder context through argument $\mathbf{s}$.

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

```
Done
```