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.

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

$$
\mathbf{y} = \begin{array}{ll} \text{Answer:} & \text{He taught them Machine Learning} \end{array}
$$

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 NOUNE>, <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 t
        - the latent state is finite representation of the prefix of the sequence of length t

The use of latent state/memory evolved over the models we studied

- RNN
    - latent state encodes
        - input representation
        - "control" state
            - guiding how the model processes the data: state transitions
- LSTM
    - latent state partitioned into
        - Short Term memory: control state
        - Long Term memory

Both these models processed the input sequence **once**

- so input-specific representation needs to be part of memory

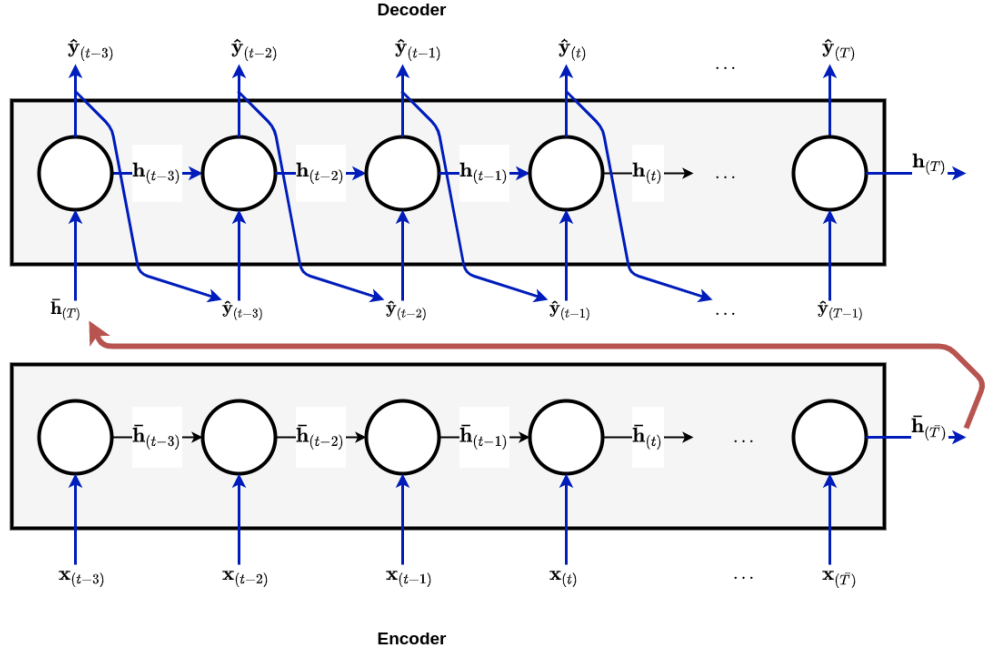
A common architecture for Sequence to Sequence tasks is the Encoder-Decoder:

- The Encoder is an RNN
  - Acts on input sequence $\left[\mathbf{x}_{(1)} \ldots \mathbf{x}_{(\bar{T})}\right]$
  - Producing a sequence of latent states $\left[\bar{\mathbf{h}}_{(1)}, \ldots, \bar{\mathbf{h}}_{(\bar{T})}\right]$
    - latent state $\bar{\mathbf{h}}_t$ is a summary of $\left[\mathbf{x}_{(1)} \ldots \mathbf{x}_{(t)}\right]$

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 $\mathbf{h}_{(t)}$ is response for generating output token $\hat{\mathbf{y}}_{(t)}$
  - Thus outputting a sequence $[\hat{\mathbf{y}}_{(1)}, \ldots, \hat{\mathbf{y}}_{(T)}]$
- Often feeding step t output $\hat{\mathbf{y}}_{(t)}$ as Encoder input at step $(t+1)$

RNN Encoder/Decoder



Decoder output $\hat{\mathbf{y}}_{(t)}$

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

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

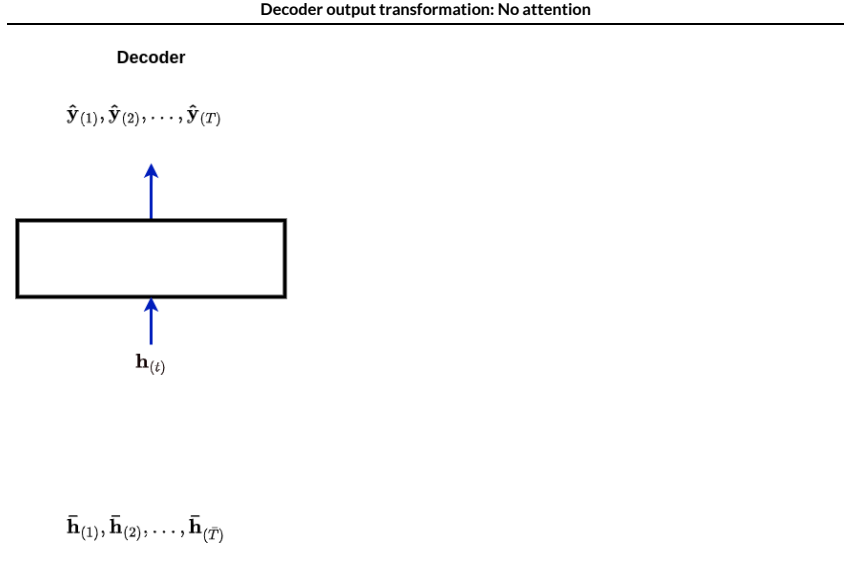
It is easy to add another NN to transform $\mathbf{h}_{(t)}$ into a $\hat{\mathbf{y}}_{(t)}$ that is different.

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

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

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

Here is what the additional NN looks like:



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

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 $\bar{\mathbf{h}}_{(\bar{T})}$
- which in turn was encoded in all Encoder states $\bar{\mathbf{h}}_{(\bar{t}')}$ $\text{for } t' \geq \bar{p}$
    - where $\bar{p}$ is the position withing sequence $\mathbf{x}$ 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]

$\bar{\mathbf{h}}_{(1)}, \ldots, \bar{\mathbf{h}}_{(\bar{T})} =$

- **Subject:** Professor Perry
- **Pronoun:** he
- **Object:** Machine Learning
- **Indirect Object:** them
- **Verb:** taught

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 $[\mathbf{h}_{(1)}, \ldots, \mathbf{h}_{(4)}]$
- 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 $\mathbf{h}_{(t)}$
    - 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 t

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

Query performing a Lookup in the Dictionary

Question: What did Professor Perry do ?

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

Queries:

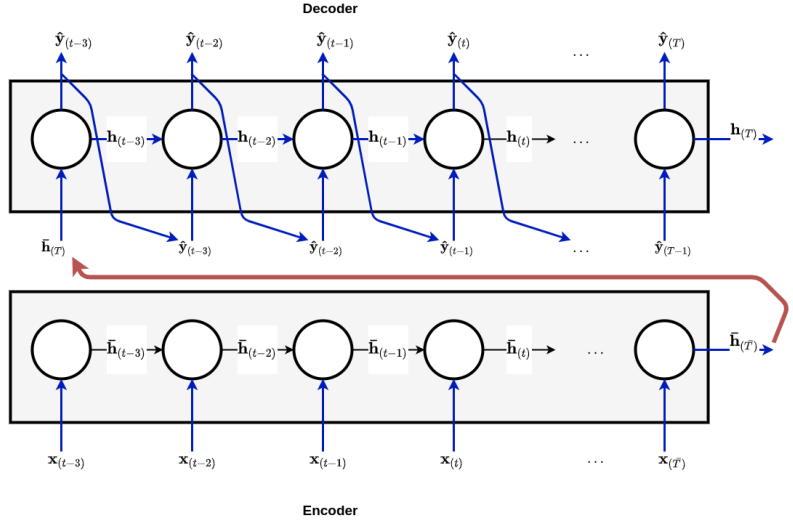
$\mathbf{h}_{(\mathbf{Pronoun})}$ = {Pronoun: ?}

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

Answer: He taught them Machine Learning [CLS]

Here is a picture describing this hypothetical functioning.

RNN Encoder/Decoder without Attention



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

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

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

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)

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

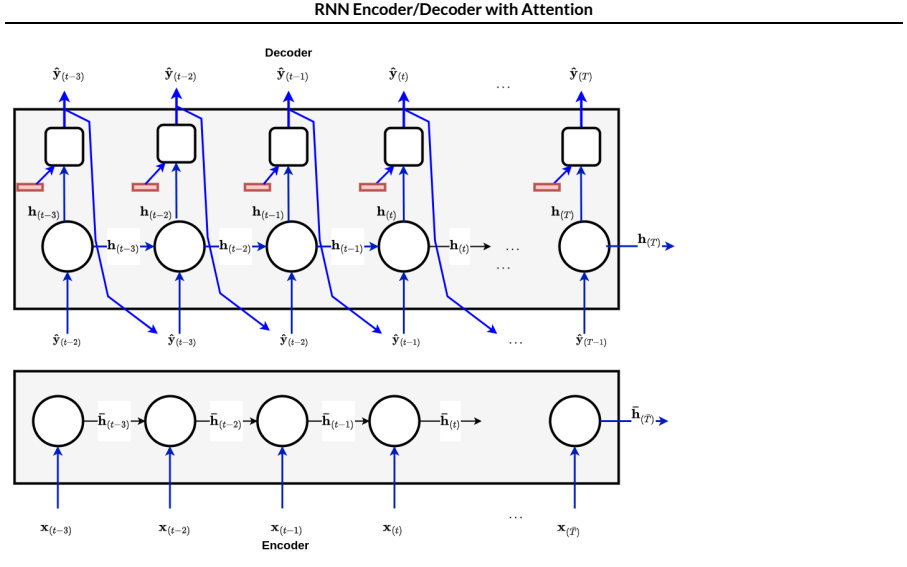
- to directly access all of the Encoder latent states $\bar{\mathbf{h}}_{(1)} \ldots \mathbf{h}_{(\bar{T})}$
- at each time step of the Decoder

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



$\bar{\mathbf{h}}_{(\bar{T})} =$

- Subject: Professor Perry
- Pronoun: he
- Object: Machine Learning
- Indirect Object: them
- Verb: taught

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 ?

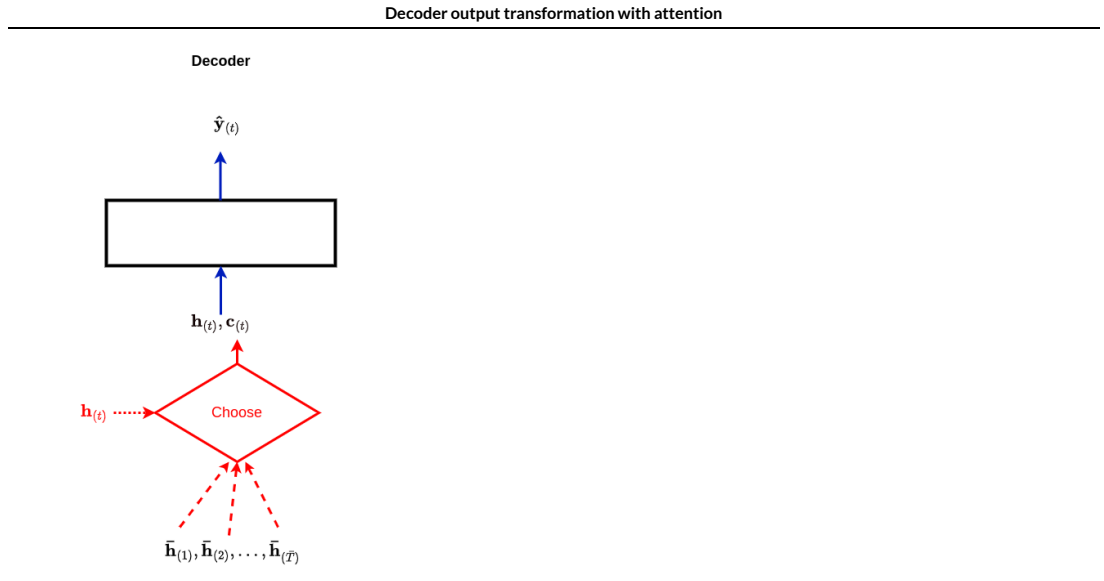
The box's input at step t

- the Decoder latent state $\mathbf{h}_{(t)}$
- 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 $\hat{\mathbf{y}}_{(t)}$ that is a function of both $\mathbf{h}_{(t)}$ and $\bar{\mathbf{h}}_{(1)} \ldots \mathbf{h}_{(\bar{T})}$

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

Here are the inner workings of the NN for D:



Inside the box:

- the Decoder latent state $\mathbf{h}_{(t)}$ is used as a *query*
- which is matched against each of the Encoder latent states
- resulting in one Encoder latent state being chosen as $\mathbf{c}_{(t)}$

The chosen Encoder latent state $\mathbf{c}_{(t)}$ and Decoder latent state $\mathbf{h}_{(t)}$

- are input to another Neural Network
- which produces output $\hat{\mathbf{y}}_{(t)}$

Essentially we change D so that it is conditioned on

- $\mathbf{h}_{(t)}$: the query (Decoder state)
- **and** summary of the Context $\bar{\mathbf{h}}_{([1:\bar{T}])}$

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 $\bar{\mathbf{h}}_{(\bar{t})}$)
- that is *relevant* for producing $\hat{\mathbf{y}}_{(t)}$
- 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

A big advantage of this approach:

- the latent state of the Decoder is solely for "control" (e.g., creating the output according to the Answer template)
- and not for storing "knowledge" (dictionary)

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: $\langle \bar{\mathbf{h}}_{(\bar{t})}, \bar{\mathbf{h}}_{(\bar{t})} \rangle$
  - Key is equal to value; they are latent states of the Encoder
- But with *soft* lookup
  - The current Decoder state $\mathbf{h}_{(t)}$ is presented to the CSM
    - Called the *query*
    - Is matched across each key of the dict (i.e., a latent state $\bar{\mathbf{h}}_{(\bar{t})}$)
  - 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`

Have we seen this before ?

If you recall the architecture of the LSTM

- *short term* (control) memory
- was separated from *long term* memory
- elements of long term memory are moved to short term memory *as needed*

This is partly similar to the advantages of Attention.

But

- all factual information from input $\mathbf{x}$ has to flow through the bottleneck $\bar{\mathbf{h}}_{(\bar{T})}$ of the Encoder output

Visualizing Attention

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

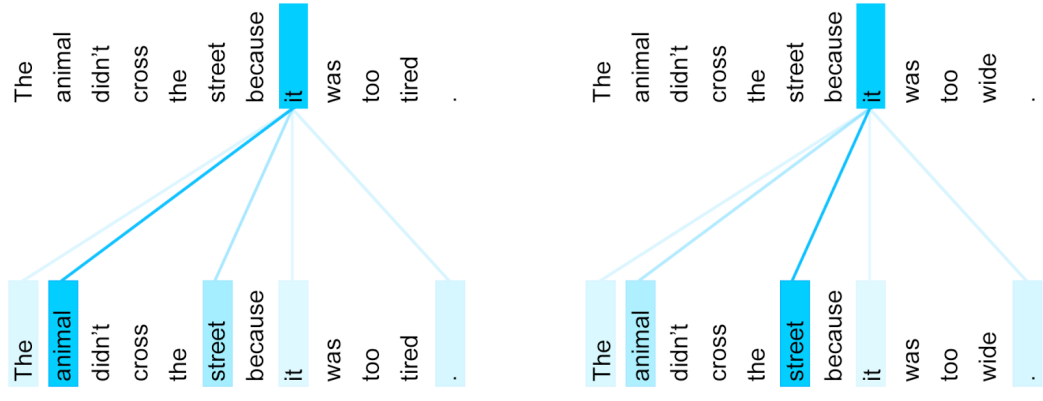
- at a particular output position t
- we can display the amount of "attention"
- that each position in the input receives

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

Entailment: Does the "hypothesis" logically follow from the "premise"

Attention: Entailment

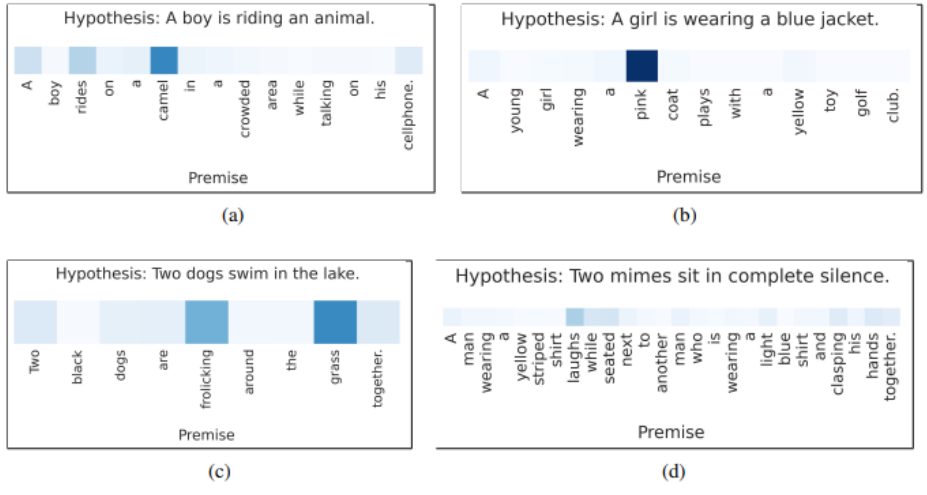


Figure 2: Attention visualizations.

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)"

Date normalization example

- 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)

Image captioning example

- 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)

Self-attention

We have illustrated the benefit of enabling the Decoder to attend to the Encoder.

This form of attention is called *Cross Attention*.

But we can further simplify the Decoder control by enabling it, when generating $\hat{\mathbf{y}}_{(t)}$

- to attend to all previously generated outputs $\hat{\mathbf{y}}_{([1:t-1])}$

This form of attention if called *Self Attention*

For example, suppose the Decoder is generating a long sentence

- in many languages, there needs to be agreement between the gender/plurality of a subject and verbs
- Self attention enables the Decoder to refer back to the previously generated subject of the sentence
- when generating the verb for each subsequent output position

It is common in an Encoder-Decoder architecture to have both

- Cross Attention from Decoder to Encoder
- Self Attention from Decoder to Decoder

We will see both forms used in the Transformer.

These mechanisms are attending to different sequences (Encoder states or Decoder outputs).

We will henceforth use the term *sequence being attended to* as a general term

- instead of specifically referring to the part of the network that produced it

Masked attention

As presented, the Attention mechanism can refer to an entire sequence

- e.g., the sequence of Encoder latent states

It is sometimes desirable to *limit* what may be attended to.

For example, consider a decision at time t that may depend *only on the past*

- positions $t' < t$
- for example, a trading decision at time t may depend only on *prior* information
    - typical of sequences that are timeseries

Restricting attention to the past is called *Causal Attention*.

- the next output depends only on things that could have caused it (the past), not the future

There is a mechanism to restrict what may be attended to in a general way

- create a "mask"
- a bit vector for each position of the sequence being attended to
- such that attention is limited to positions where the mask element if True.

This is called *Masked Attention*.

It is frequently used to enable a Decoder, when predicting output $\hat{\mathbf{y}}_{(t)}$

- to attend to **previously** generate outputs $\hat{\mathbf{y}}_{[1:t-1])}$
- but not **future** outputs $\hat{\mathbf{y}}_{(t')}$ for $t' \geq t$

When used in this manner, we refer to the behavior as *Masked Self Attention*

Aside

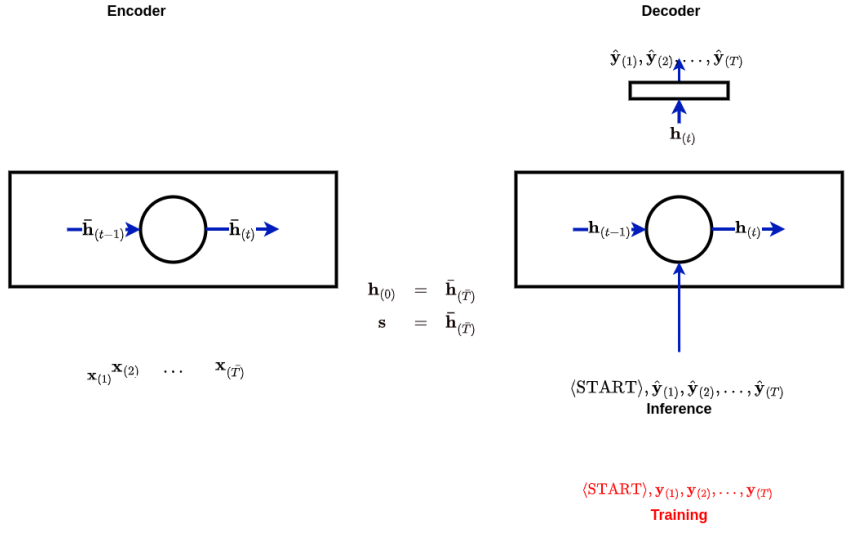
You may wonder how it is even practically possible for a Decoder to refer to the future.

When using *Teacher Forcing* for **training**

- the Decoder does not use the *predicted* target sequence $\hat{\mathbf{y}}_{(1:T)}$
- the Decoder uses the *actual* target sequence $\mathbf{y}_{(1:T)}$
    - hence, "future" positions $t' \geq t$ are available
- this prevents a single mis-prediction at position t from cascading and ruining all future output
    - facilitates training
- at inference time: the Decoder works on the *predicted* Target sequence.

In the diagram below, we illustrate (lower right) how the Decoder input changes between Training and Test/Inference time.

Sequence to Sequence: training (teacher forcing) + inference: No attention



Multi-head attention: two heads are better than one

Perhaps when generating the output for position t 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 $\mathbf{y}_{(l)}$ where $n_{(l)} > 1$
- a Convolutional layer l may produce outputs (for each spatial location) for many channels

We have referred to this as layer l producing $n_{(l)}$ *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

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 $\mathbf{h}_{(t)}$ and the key/value pairs $\bar{\mathbf{h}}_{(1)} \ldots \bar{\mathbf{h}}_{(\bar{T})}$ being queried.

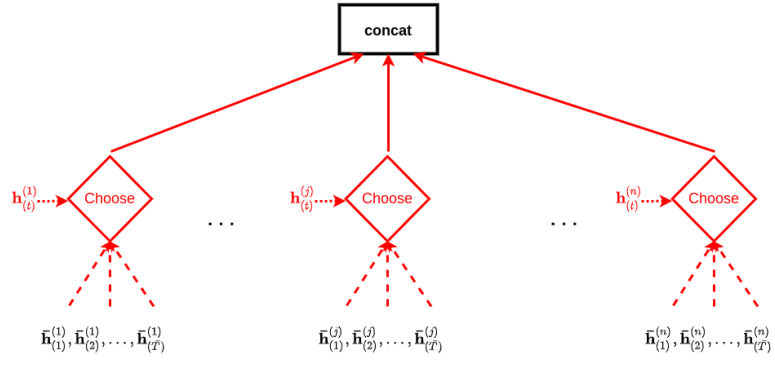
- into $\mathbf{h}^{(j)}_{(t)}$ and the key/value pairs $\bar{\mathbf{h}}^{(j)}_{(1)} \ldots \bar{\mathbf{h}}^{(j)}_{(\bar{T})}$
- Such that each head attends to a separate item

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

There is a new model (the Transformer) that processes sequences much faster than RNN's.

It is an Encoder/Decoder architecture that uses multiple forms of Attention

- Self Attention in the Encoder
    - to tell the Encoder the relevant parts of the input sequence $\mathbf{x}$ to attend to
- Decoder/Encoder attention
    - to tell the Decoder which Encoder state $\bar{\mathbf{h}}_{(t')}$ to attend to when outputting $\mathbf{y}_{(t)}$
- Masked Self-Attention in the Decoder
    - to prevent the Decoder from looking ahead into inputs that have not yet been generated

Conclusion

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

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

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

This rigidity forced Decoder latent state $\mathbf{h}_{(t)}$ 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
```