The importance of selectively forgetting

The RNN state $\mathbf{h}_{(t)}$ is updated as a function of the next element $\mathbf{x}_{(t)}$ of the input sequence.

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

- as a vector of features
- each feature is an independent function of all previous inputs $\mathbf{x}_{(t')}$ for $t' \leq t$.

But is each and every input $\mathbf{x}_{(t)}$ relevant to all the features in $\mathbf{h}_{(t)}$?

Probably not.

So it would be very powerful

- to be able to *selectively update* elements of $\mathbf{h}_{(t)}$ while leaving others unchanged.

There are even cases where we might want to reset an element of $\mathbf{h}_{(t)}$ to 0 (forget)

- a new piece of information invalidates prior assumptions

We will show how to use math to implement "conditional" statements like "if" and "switch".

This will allow us to construct powerful variants of RNN's

- that allow selective, independent modification of features in $\mathbf{h}_{(t)}$.

From Math to Program

Early layers (FC, CNN) looked like purely mathematical objects: functions.

The RNN is still a (recursively defined) function, but looks more like a program (loop).

Later layer types will continue the trend of looking more like programs than mathematical functions

- binary conditionals ("if" statements)
    - gates
- multi-case conditionals ("case", "switch" statements)
    - attention

An important requirement:

- the "program" must be differentiable in order for the layer to be used in Back Propagation !

As we will see, this requirement results in "soft" versions of otherwise hard ("if" or "else") decisions.

”If” statements - Gates

The sigmoid function range is $[0, 1]$.

This makes it attractive to use as a "gate" with which to implement an "if" statement

Suppose we need to compute a $\mathbf{y}$ that takes on value T if some condition g is `True` and F otherwise.

The following product (almost) does the trick

$$\begin{aligned} \text{\textbackslash g} &= \sigma(\ldots) \\ \mathbf{y} &= \text{\textbackslash g} \otimes \mathbf{T} + (1 - g) \otimes \mathbf{F} \end{aligned}$$

$\otimes$ denotes *element-wise* multiplication (Hadamard product)

This subtle difference is actually very important:

- being element-wise means components of the vectors are independent of one another.
- gradients don't interact

Note that $\mathbf{y}$, \g, $\mathbf{T}$, $\mathbf{F}$ are all vectors, not scalars.

Essentially: we are making a conditional choice for *each element* of $\mathbf{y}$, independently.

If $\g$ took on only the values 0 or 1, this would exactly replicate a conditional.

But $\g$ wouldn't be differentiable.

We use a continous (soft) decision $\g$

- not hard (exactly 0 or 1), but in range
- hope that $\mathbf{y}$, which is a mixture of $\mathbf{T}$ and $\mathbf{F}$, will be "mostly" of the correct type.

$\mathbf{y}$ being a vector of features means that our pseudo-conditional can update features independently.

”Switch” statements

Let's generalize the binary conditional to multiple choice: a "switch" or "case" statement.

Suppose we need to set $\mathbf{y}$ to one value from among multiple choices in $\mathbf{C}$

$$\backslash\mathrm{g} = \sigma(\ldots)$$
$$\mathbf{y} = \backslash\mathrm{g} \otimes \mathbf{C}$$

In the case that \g were like a OHE (single element with value 1, all other elements with value 0)

- this would replicate a "switch" or "case" statement.

Analagous with the "if" section, this is a soft rather than a hard choice.

The "if" statement could be derived from the "switch" by making

$$\mathbf{C} = \begin{bmatrix} \mathbf{T} \\ \mathbf{F} \end{bmatrix}$$

We refer to \g as a *mask* for $\mathbf{C}$.

Motivation for LSTM

The vanilla RNN was powerful but limited

- gradients were prone to vanishing/exploding
- they suffered from "short term" memory:
    - could not capture dependencies that were too far separated by time

LSTM's are one attempt at mitigating these two issues.

RNNs have a single state vector $\mathbf{h}$ that serves a dual purpose

- a form of memory
- a form of transition control: deciding what action to take on the current input $\mathbf{x}_{(t)}$

LSTM's separate these two roles:

- a separate "long-term" memory vector $\mathbf{c}$
    - can decide at step t what to remember independent of action/output at t
- a separate "short term" memory vector $\mathbf{h}$, used for transition control
    - sole responsibility is deciding on current action/output at t
    - derived from $\mathbf{c}$

In addition, the update equations for $\mathbf{c}$ (and $\mathbf{h}$ since it is derived from $\mathbf{c}$) are subtly crafted

- to diminish the issue of vanishing/exploding gradients.

The LSTM will seem complicated at first, because it has many small pieces that inter-connect.

The following Blog post by E. Chen (http://blog.echen.me/2017/05/30/exploring-lstms/) is one of the best intuitive explanations that I've seen.

Our presentation will follow the "classical" derivation (based on the original paper and Geron's book), but with a healthy assist from the blog.

State: Long and short term

Memory/state is divided within the LSTM into two parts with different roles:

- short term state $\mathbf{h}$ used for "transition control"
    - analogy: RAM, working memory
- long term state: $\mathbf{c}$
    - analogy: disk

The LSTM "API"

During one time step of computation, the LSTM computes 3 values

- new short term state $\mathbf{h}_{(t)}$
- new long term state $\mathbf{c}_{(t)}$
- output $\mathbf{y}_{(t)}$ (sometimes simply taken to be same as short term state)

The three separate computations are functions of

- the previous short term state $\mathbf{h}_{(t-1)}$,
- previous long term state $\mathbf{c}_{(t-1)}$
- and the current input $x_{(t)}$.

$$\mathbf{y}_{(t)}, \mathbf{h}_{(t)}, \mathbf{c}_{(t)} = f(\mathbf{x}_{(t)}, \mathbf{h}_{(t-1)}, \mathbf{c}_{(t-1)})$$

Note the recursive aspect of the computation of $\mathbf{h}_{(t)}$, $\mathbf{c}_{(t)}$: they implicitly depend on the values of the states at all previous time steps $t' < t$.

Rather than produce a distinct output $y_{(t)}$, we identify the output with the short-term state

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

Geron page 493, last equation

Updating long term and short term state

Overview

The basic processing step of an LSTM is

- update the long term state
- derive a new short term state from the updated long term state

There are quite a few moving parts to keep track of.

Each part is represented by an update equation.

The equations differ only in

- each has it's own unique weights and bias matrices
- the activation functions for gates are σ: keeps selectors in range $[0, 1]$
- the activation functions for memory is $\tanh$; keeps memory in range $[-1, +1]$

Gates

The long term state $\mathbf{c}$ is a vector of "features"

- potentially very long
- whose elements "remember" concepts that prove useful for solving the problem.

On each step, it would seem reasonable to

- be able to *selectively* update parts of the long state vectors, rather than the entire vector.

An LSTM associates several gate vectors with the long term state vector.

- Each gate vector serves as a mask for the long term state vector, weighting each element.
- Thus, it serves as a (continuous) way of selecting sub-parts of the long vectors.

There is more than one gate vector because different masks need to be applied during the update.

Update long term state

The first step in updating the long term state $\mathbf{c}_{(t)}$ is straight forward:

- produce a new candidate vector $\mathbf{c}'_{(t)}$ to replace the long term state.
- "candidate" in that it may or may not replace $\mathbf{c}_{(t)}$

The candidate is a function of the prior short term state $\mathbf{h}_{(t-1)}$ and the current input $\mathbf{x}_{(t)}$:

$$\mathbf{c}'_{(t)} = s(\mathbf{x}_{(t)}; \mathbf{h}_{(t-1)}) = \tanh(\mathbf{W}_{x,c} x_{(t)} + \mathbf{W}_{h,c}\mathbf{h}_{(t-1)} + \mathbf{b}_c)$$

This is very much like the RNN state update equation, without the recursion (since LHS is $\mathbf{c}'$).

Why choose $\tanh$ as the activation ?

- range is $[-1, +1]$
- will serve to increment/decrement a counter ($\mathbf{c}_{(t)}$, as we will see)

We may not want the candidate vector to replace *all* elements of the long term state.

The gate vector $\text{\save}_{(t)}$

- a mask to select only those parts of the candidate that will replace their counterparts.

We may also decide to "forget" parts of the long term state as they may no longer be relevant.

The gate vector $\text{\remember}_{(t)}$

- is a mask to select those parts of the *prior long term state vector* which should **not** be forgotten.

The update of $\mathbf{c}$ combines the old parts to remember with the new parts to include:

$$\mathbf{c}_{(t)} = \text{\remember}_{(t)} \otimes \mathbf{c}_{(t-1)} + \text{\save}_{(t)} \otimes \mathbf{c}'_{(t)}$$

Note

$\mathbf{c}_{(t)}$ has not been squashed by an activation function, so we haven't limited its range.

But we **have** squashed the candidate (via $\tanh$) so that it's range is $[-1, +1]$

So

$$\text{\save}_{(t)} \otimes \mathbf{c}'_{(t)}$$

acts like an increment/decrement to $\mathbf{c}_{(t)}$.

That is: $\mathbf{c}_{(t)}$ acts like a simple counter.

Cell state as counter

You can imagine how a counter might be used in a text sequence

- Count nesting level
    - balanced open/close delimiters
    - itemized list counter
- Binary counts to determine conditions
    - inside/outside a quote
    - inside/outside a URL
- Count length of input
    - end of sentence marker more likely as sentence length increases

Update short term state

The short term state update

- selects which parts of the newly-updated long term state
- to make part of the control program (RAM)

$$\mathbf{h}_{(t)} = \text{\focus}_t \otimes \tanh(\mathbf{c}_{(t)})$$

That is, the new short term state $h_{(t)}$ are (squashed) elements of the new long term state

We will also choose to output $\mathbf{h}_{(t)}$:

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

The gate equations

All of the gates are updated via similar equations; each has it's own unique weight and bias matrices.

Historically $\text{\remember}_{(t)}, \text{\save}_{(t)}, \text{\focus}_{(t)}$ have been denoted

$$f_{(t)}, i_{(t)}, o_{(t)}$$

for "forget" (although it really means "don't forget"!), "input", "output".

$$\begin{aligned}
\text{\remember}_{(t)} &= f_{(t)} &= \sigma(\mathbf{W}_{x,f}\mathbf{x}_{(t)} + \mathbf{W}_{h,f}\mathbf{h}_{(t-1)} + \mathbf{b}_f) \\
\text{\save}_{(t)} &= i_{(t)} &= \sigma(\mathbf{W}_{x,i}\mathbf{x}_{(t)} + \mathbf{W}_{h,i}\mathbf{h}_{(t-1)} + \mathbf{b}_i) \\
\text{\focus}_{(t)} &= o_{(t)} &= \sigma(\mathbf{W}_{x,o}\mathbf{x}_{(t)} + \mathbf{W}_{h,o}\mathbf{h}_{(t-1)} + \mathbf{b}_o)
\end{aligned}$$

These equations are identical except for having their own unique weights and biases.

They differ from the "candidate" update equation

- latter uses a $\tanh$ activation to squash the long term memory candidate to the range $[-1, +1]$
- gates use the sigmoid σ to keep the gates in the $[0, 1]$ range.

LSTM as gated residual connections

We have previously described how residual (or skip) connections

- address the problem of vanishing/exploding gradients
- allowing the output of layer $l-1$ to "skip over" the computation in layer l.

This applies to the gradients as well, which flow backwards and can skip a layer.

Examine the update equation for the long term state:

$$\mathbf{c}_{(t)} = \text{\remember}_{(t)} \otimes \mathbf{c}_{(t-1)} + \text{\save}_{(t)} \otimes \mathbf{c}'_{(t)}$$

and consider element i of the vector.

If

$$\text{\remember}_{(t),i} = 1, \text{\save}_{(t),i} = 0$$

then $\mathbf{c}_{(t),i} = \mathbf{c}_{(t-1),i}$.

That is, the LSTM has the ability to

- flow $\mathbf{c}_{(t-1),i}$ forward unchanged
- and for its derivative to flow backward unchanged

This was the key motivation of the skip connection.

If $\text{\focus}_{(t),i}$ is also equal to 1

- the i^{th} component of short term state $\mathbf{h}_{(t),i}$ also "skips" interacting at t
- since

$$\mathbf{h}_{(t)} = \text{\focus}_t \otimes \tanh(\mathbf{c}_{(t)})$$

So part of the power of the LSTM is

- its combination of gates and skip connections to avoid exploding/vanishing gradients.

Initial bias to ”not forget”

The update equation for the long term state:

$$\mathbf{c}_{(t)} = \text{\remember}_{(t)} \otimes \mathbf{c}_{(t-1)} + \text{\save}_{(t)} \otimes \mathbf{c}'_{(t)}$$

is not a true "skip" connection as it is gated by $\text{\remember}_{(t)}$.

In practice; we want

$$\text{\remember}_{(t)} \approx +1$$

in early *epochs* of training.

This will speed up learning

- by allowing gradients to flow backwards unmodulated
- during the time weights need to be adjusted most (from initial, uninformed values)

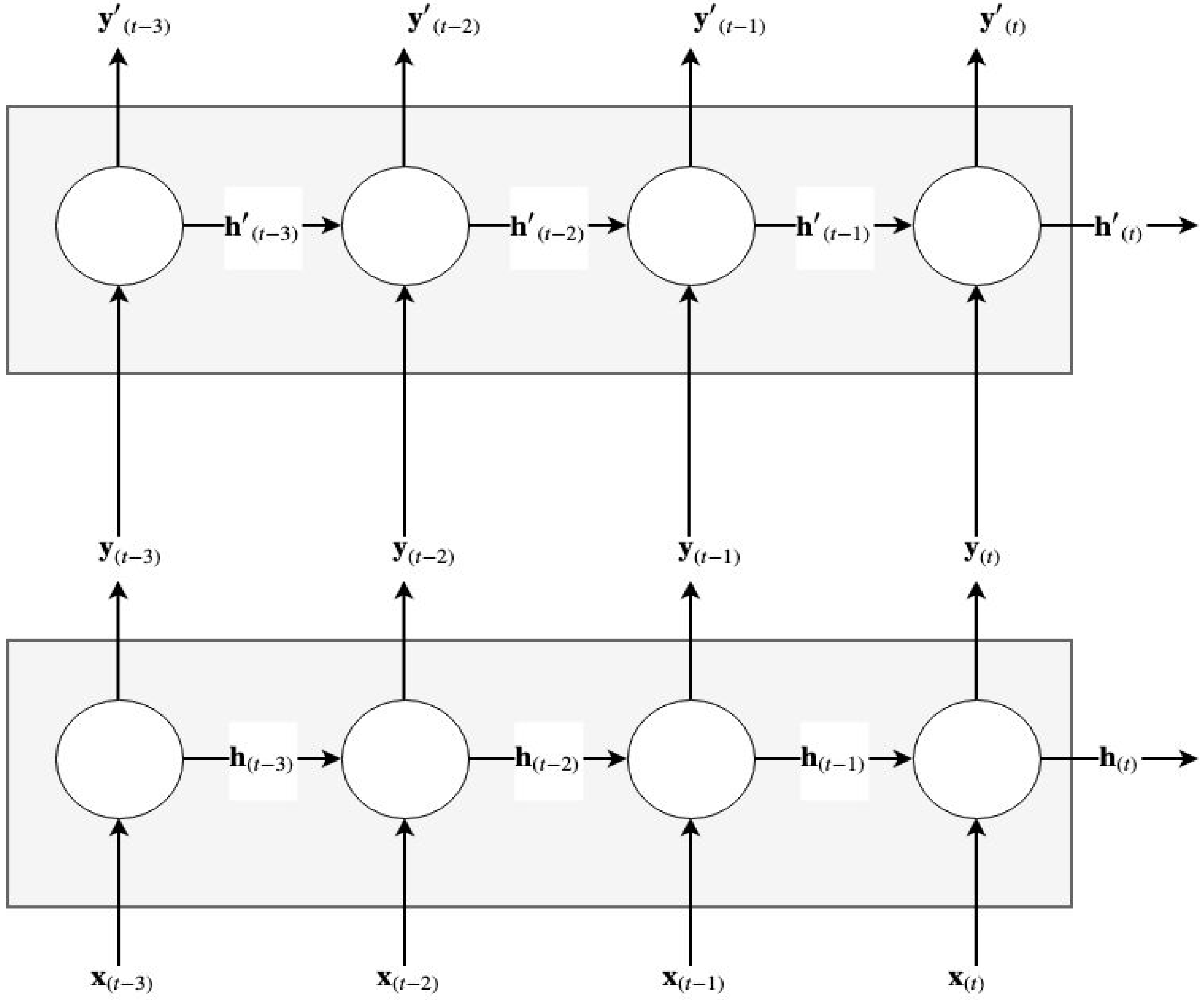
This is done by setting $\mathbf{b}_f$ to a large value in the equation

$$\text{\remember}_{(t)} = f_{(t)} = \sigma(\mathbf{W}_{x,f}\mathbf{x}_{(t)} + \mathbf{W}_{h,f}\mathbf{h}_{(t-1)} + \mathbf{b}_f)$$

Attention

Consider a many to many implementation of a Recurrent NN (RNN, LSTM, etc).

RNN Encoder/Decoder


$\mathbf{y'}_{(t-3)}$
$\mathbf{y'}_{(t-2)}$
$\mathbf{y'}_{(t-1)}$
$\mathbf{y'}_{(t)}$
$\mathbf{h'}_{(t-3)}$
$\mathbf{h'}_{(t-2)}$
$\mathbf{h'}_{(t-1)}$
$\mathbf{h'}_{(t)}$
$\mathbf{y}_{(t-3)}$
$\mathbf{y}_{(t-2)}$
$\mathbf{y}_{(t-1)}$
$\mathbf{y}_{(t)}$
$\mathbf{h}_{(t-3)}$
$\mathbf{h}_{(t-2)}$
$\mathbf{h}_{(t-1)}$
$\mathbf{h}_{(t)}$
$\mathbf{x}_{(t-3)}$
$\mathbf{x}_{(t-2)}$
$\mathbf{x}_{(t-1)}$
$\mathbf{x}_{(t)}$

An example might be a network that adds descriptions/captions to a stream of images (video)

- input sequence: a sequence of frames
- output sequence: a sequence of words

or that translates from one language to another

- input sequence: words in source language
- output sequence: words in target language

It is very possible that the next word (time step t) might refer to a much earlier frame ($t' < t$).

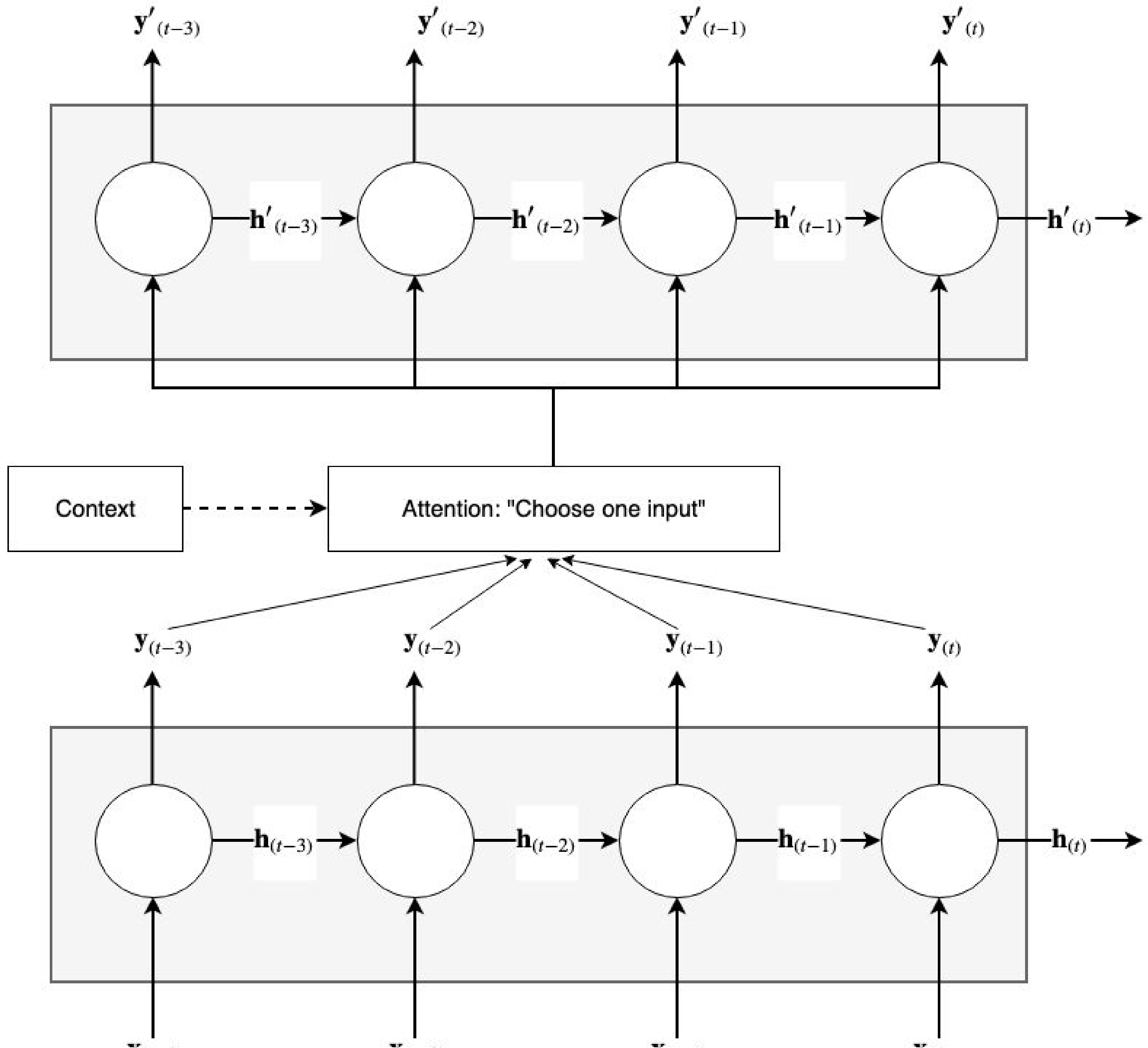
A similar thing happens when translating between languages.

There is not necessarily a correspondence between output t and input t.

So an LSTM needs to decide which part of the past to "attend" (pay attention) to.

We can help it via a mechanism know as "attention", which we sketch below.

RNN with Attention


$\mathbf{y}'_{(t-3)}$
$\mathbf{y}'_{(t-2)}$
$\mathbf{y}'_{(t-1)}$
$\mathbf{y}'_{(t)}$
$\mathbf{h}'_{(t-3)}$
$\mathbf{h}'_{(t-2)}$
$\mathbf{h}'_{(t-1)}$
$\mathbf{h}'_{(t)}$
Context
Attention: "Choose one input"
$\mathbf{y}_{(t-3)}$
$\mathbf{y}_{(t-2)}$
$\mathbf{y}_{(t-1)}$
$\mathbf{y}_{(t)}$
$\mathbf{h}_{(t-3)}$
$\mathbf{h}_{(t-2)}$
$\mathbf{h}_{(t-1)}$
$\mathbf{h}_{(t)}$

The decoder is able to "select one" of the prior states, rather than just the latest one.

Of course, by now, we understand that this is a "soft" select (case/switch)

- needs to be differentiable
- so it provides a weighted combination of all prior states
    - a mask that is almost OHE becomes a true "choose one"

How does the LSTM decide which of the past states to attend to ?

Same way as all Machine Learning:

- it is controlled by weights
- that are learned by training !

So Deep Learning layers are almost becoming little computers that learn their own programs !

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

```
Done
```