Context Sensitive Memory

A Context Sensitive Memory is like a Python dict:

- data is stored via (key, value) pairs
- a "query" matches a key and returns the associated value

The difference from a Python dict

- the query is compared to every key, and a "weight" indicating strength of match is returned
    - match can be approximate
- the value returned is the weighted sum of all values
    - if there is an exact match of one and only one key, this is equivalent to a Python dict.

Context Sensitive Memory

- a collect of key/value pairs, like a Python dict
$$M = \{(k_t, v_t | 1 \leq t \leq T\}$$
- lookup: pass in a "query", get a value-like output

As we learned in studying gates: the lookup needs to make soft choices rather than hard choices to be differentiable.

Normalized scores

$$\alpha(q,k) = \frac{\exp(\text{score}(q,k))}{\sum_{k' \in \text{keys}(M)} \exp(\text{score}(q,k')}$$

Soft lookup

$$\mathbf{c} = \text{lookup}(q, M) = \sum_{(k,v) \in M} \alpha(q,k) * v$$

Scoring functions

Redefine using generic k,v rather than h_t

There are several choices for the scoring function

$$\text{score}(\mathbf{h}_{(t)}, \bar{\mathbf{h}}_{(t')}) = \begin{cases} \mathbf{h}_{(t)}^T \cdot \bar{\mathbf{h}}_{(t')} & \text{dot product, cosine similarity} \\ \mathbf{h}_{(t)}^T \mathbf{W}_\alpha \bar{\mathbf{h}}_{(t')} & \text{general} \\ \mathbf{v}_\alpha^T \tanh(\mathbf{W}_\alpha [\mathbf{h}_{(t)}; \bar{\mathbf{h}}_{(t')}]) & \text{concat} \end{cases}$$

Note

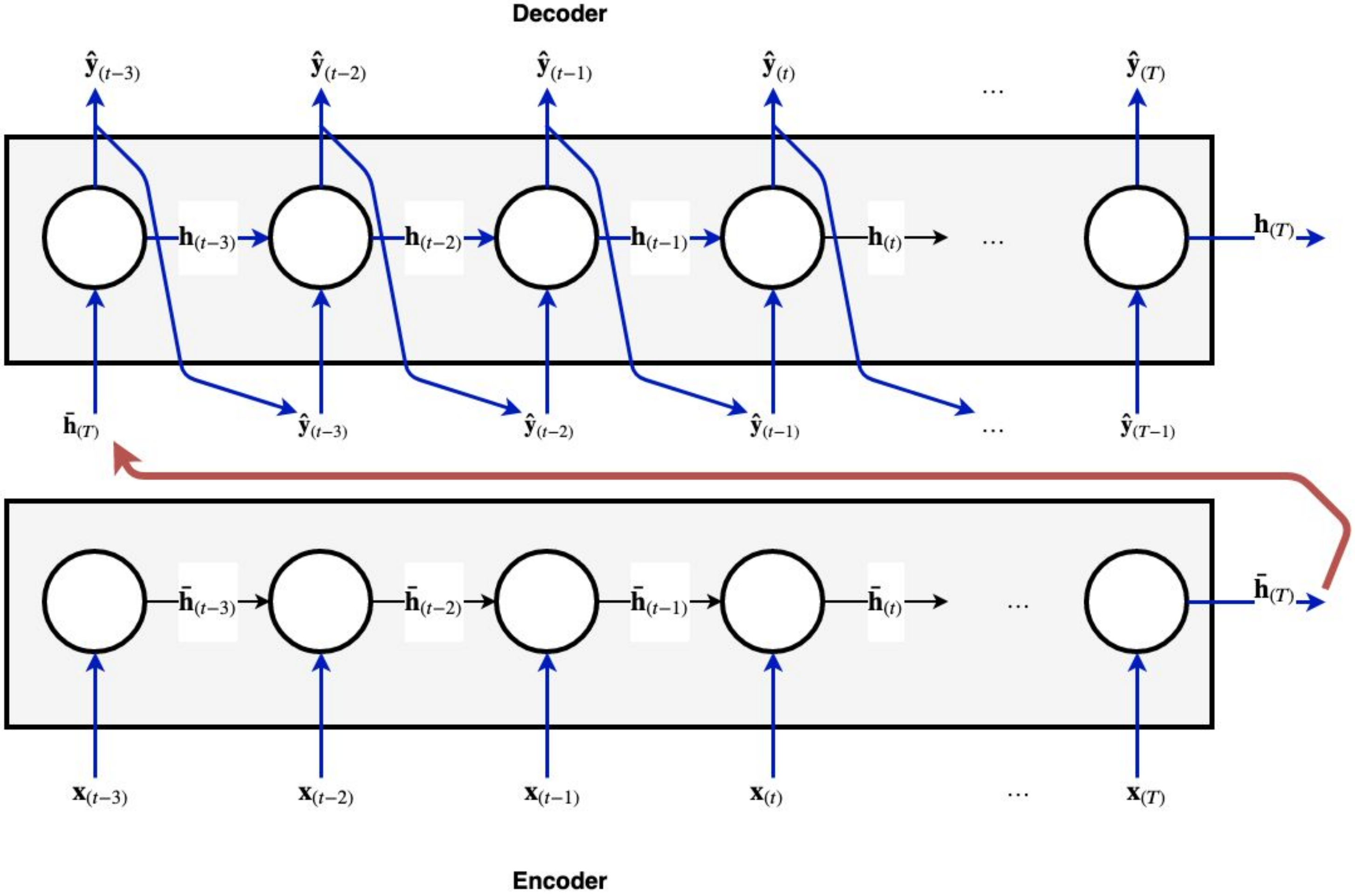
What is $\mathbf{v}_\alpha^T$?

decode_init(enc_states): h = 0 self.s = enc_states return h, s def decode_step(h, x, s): """ h: hidden state (t-1) x: input t teacher forcing: x == y_t inference : x = hat{y}_{(t-1)} s: array of encoder hidden states """ # Update hidden state, based on input x h, out = RNN(h, x) # Compute attention weights # query == h (new state of decoder) # key == value == s att_weights = ATT(h, s) # Compute context c = att_weights * s # hat{y} is function of h and c (rather than just h as in NN w/o attention) y = g([h,c]) return h, y def decode(enc_states, y=None): h, s = decode_init(enc_states) for t in range(1, t): h, y_hat = decode_step(h, x, s) # Create next input as output of this time step if y is not None: # Training: teacher forces output to be correct answer x = y[t] else: # Test: output (which becomes next input) x = y_hat

Attention

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

RNN Encoder/Decoder



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 ($t' < t$).

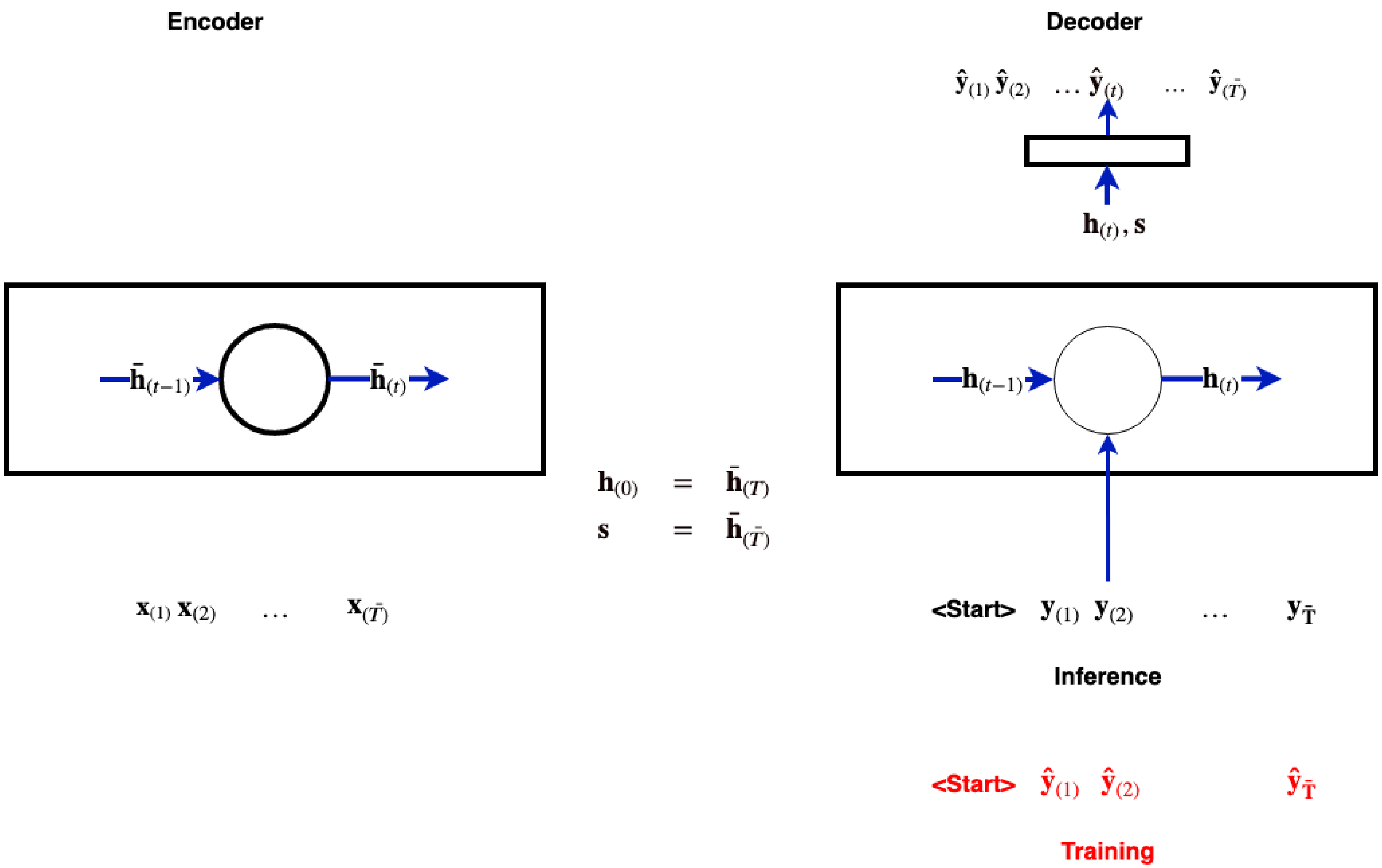
A similar thing happens when translating between languages.

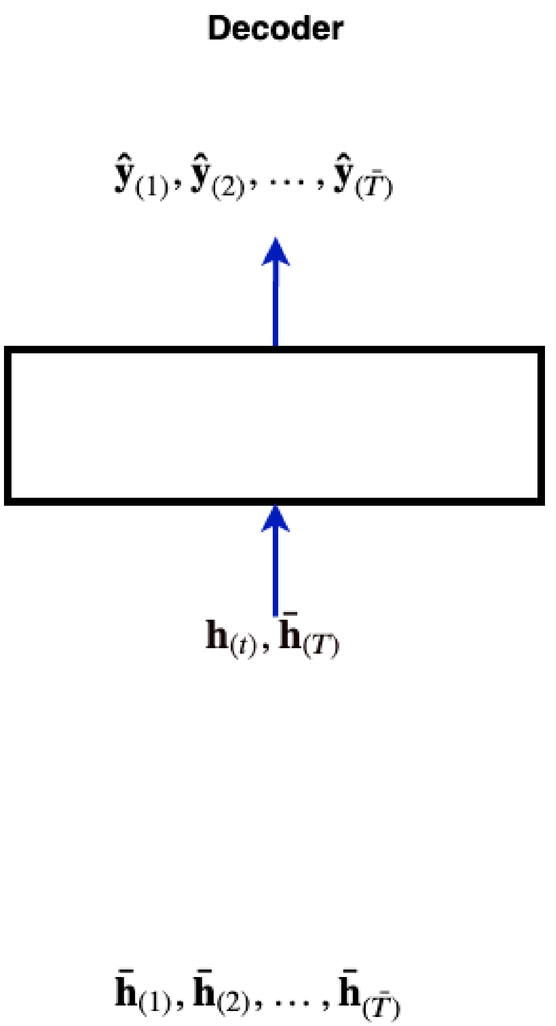
There is not necessarily a correspondence between output t and input t.

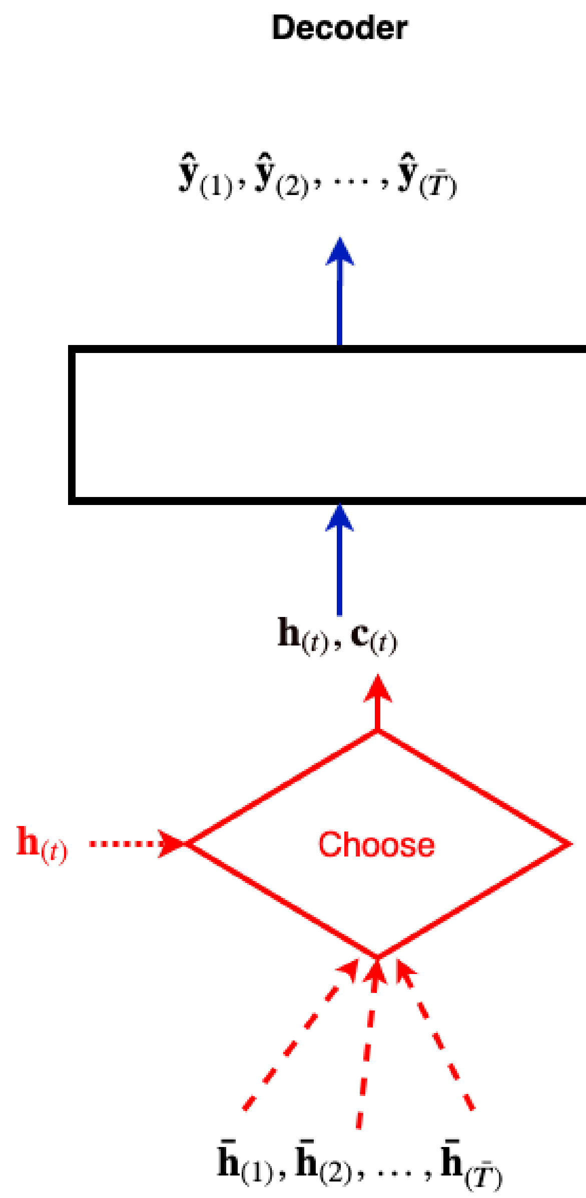
So an LSTM needs to decide which part of the past to "attend" (pay attention) to.

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

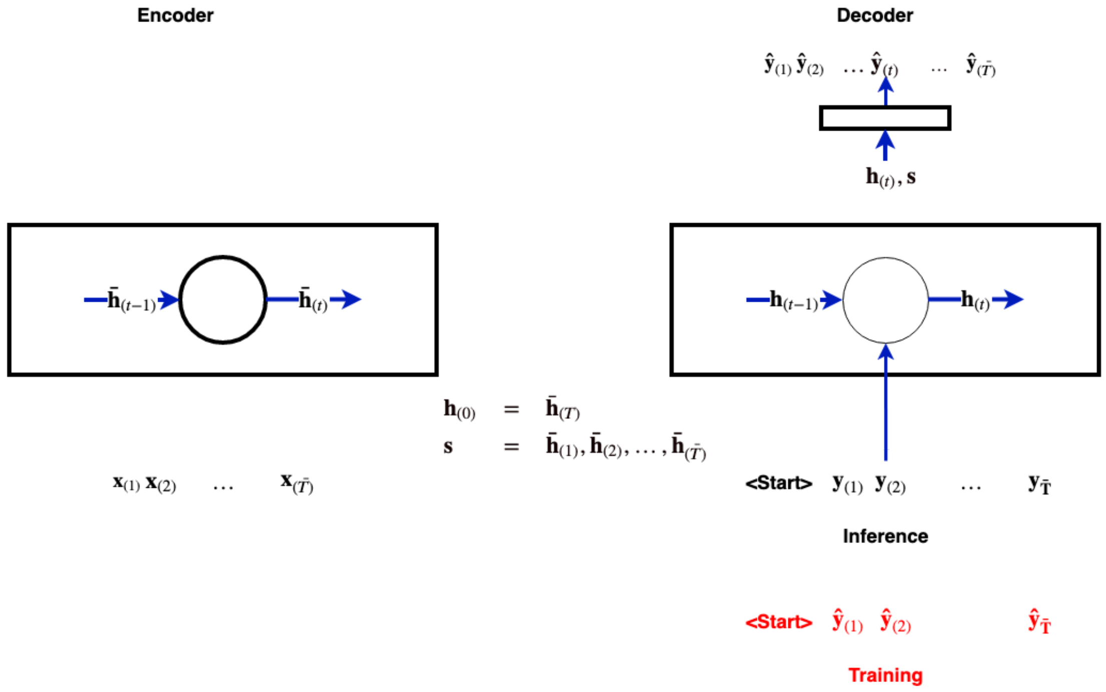
Sequence to Sequence: training (teacher forcing) + inference: No attention




Decoder
$\hat{\mathbf{y}}_{(1)}, \hat{\mathbf{y}}_{(2)}, \ldots, \hat{\mathbf{y}}_{(\bar{T})}$
$\mathbf{h}_{(t)}, \bar{\mathbf{h}}_{(T)}$
$\bar{\mathbf{h}}_{(1)}, \bar{\mathbf{h}}_{(2)}, \ldots, \bar{\mathbf{h}}_{(\bar{T})}$


Decoder
$\hat{\mathbf{y}}_{(1)}, \hat{\mathbf{y}}_{(2)}, \dots, \hat{\mathbf{y}}_{(\bar{T})}$
$\mathbf{h}_{(t)}, \mathbf{c}_{(t)}$
$\mathbf{h}_{(t)}$
Choose
$\bar{\mathbf{h}}_{(1)}, \bar{\mathbf{h}}_{(2)}, \dots, \bar{\mathbf{h}}_{(\bar{T})}$

Sequence to Sequence: attention



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 [ ]:
```

In [3]:

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

```
Done
```