Recurrent Neural Networks (1)

Erdem Koyuncu

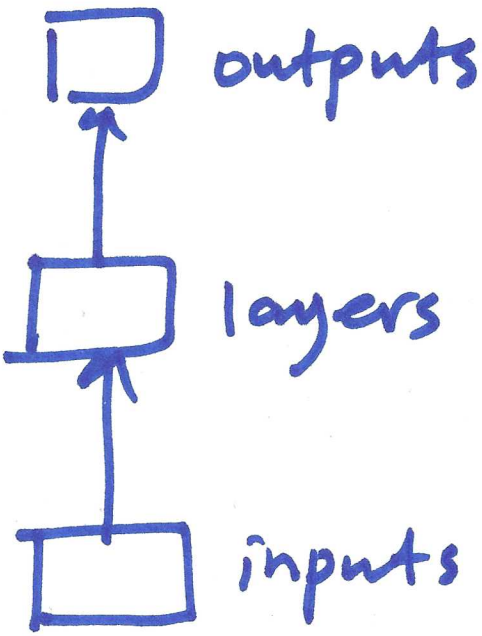
RNNs are used to process sequences (a sequence of images, words, letters, etc) in a way that takes into account the dependencies between the individual elements (or, the memory) of the sequence.

In regular (feedforward) neural networks, we provide an input to the network and observe the output after possibly many hidden layers. There is no feedback of the outputs back to the input.

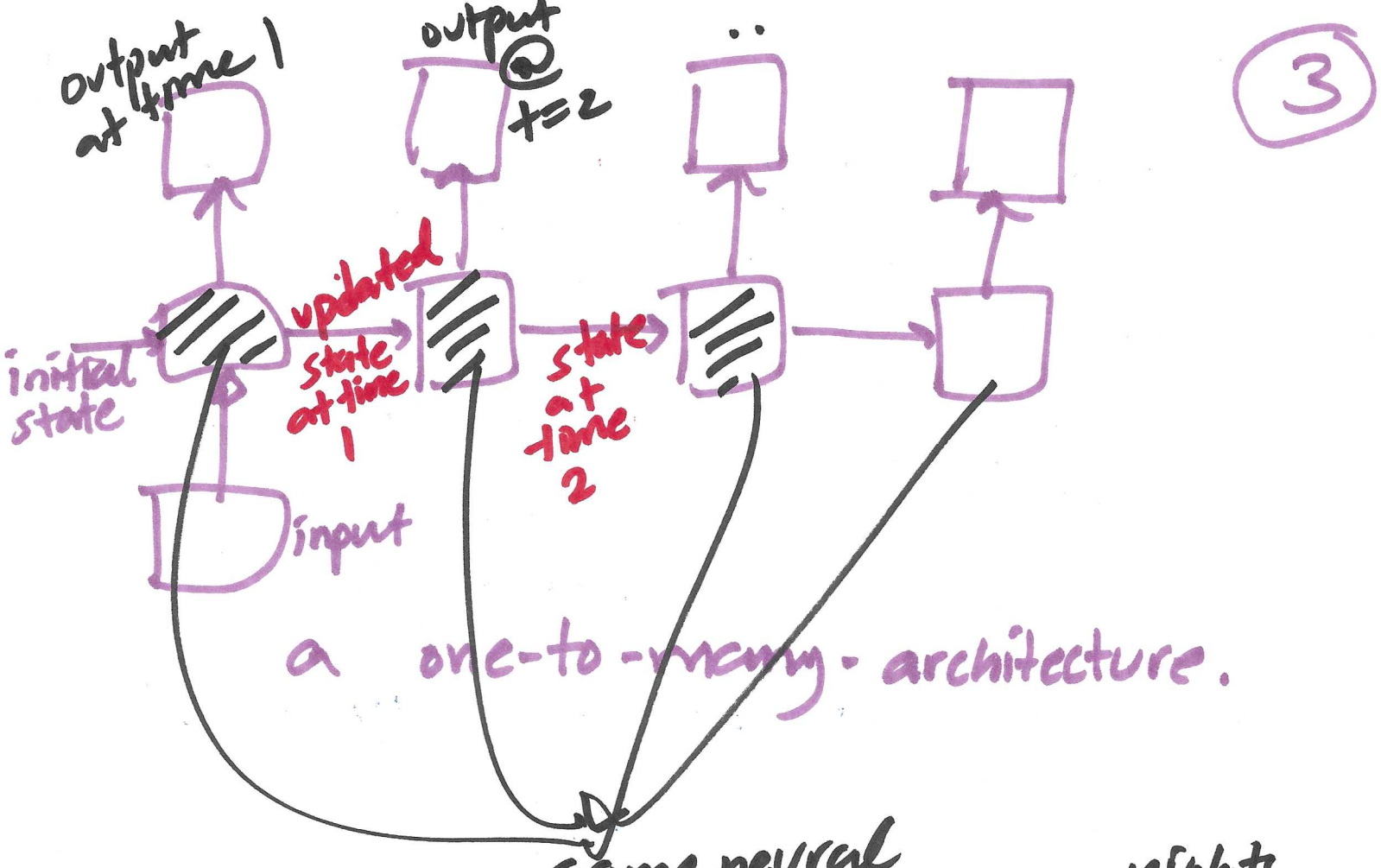
In RNNs, in general, we feed the input sequence one element at a time. Each input updates an internal state of the neural network,

which also serves as the RNN's memory. The output at any given time depends on the input at that time as well as the memory. One can imagine that the network feeds the internal states back to the network.

Ordinary (feedforward) NN:



a one-to-one architecture.



a one-to-many architecture.

same neural network with same weights.

Ex: image description/captionning

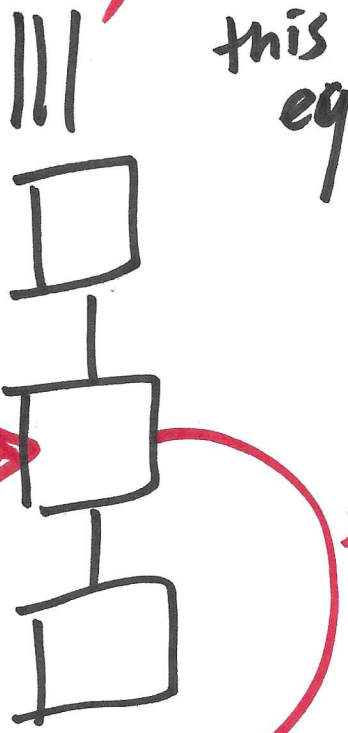
Input can be an image:

Output can be a sequence of words describing the image

"person holding an apple"

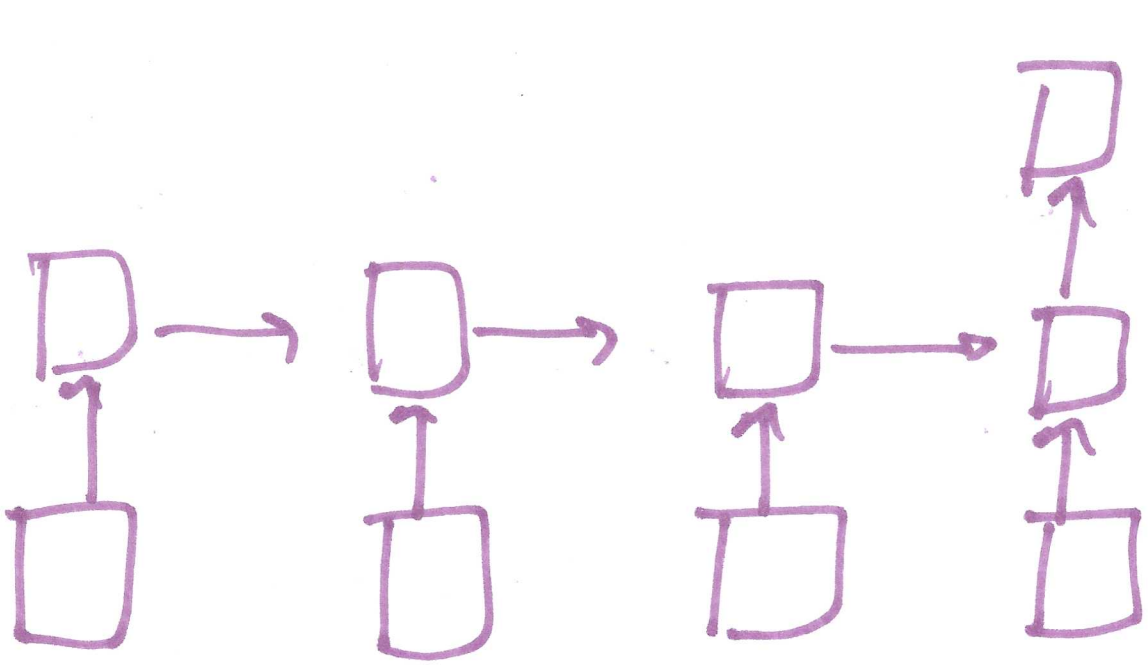
outputs at different times will be different words.

this is equivalent to



unrolled 3 times.

state feedback

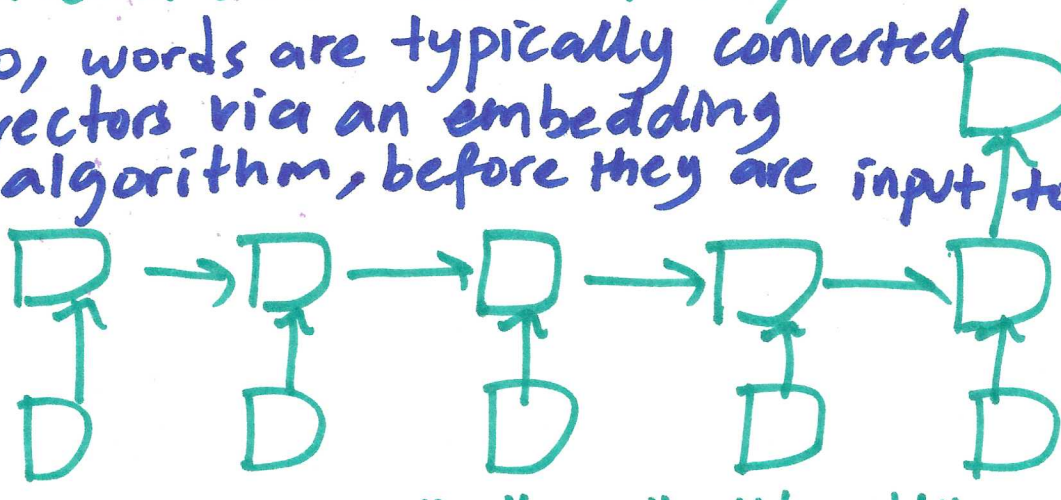


many-to-one architecture.

Ex: sentiment analysis
Input is a sequence of words
Output is whether that describes a good or bad sentiment (or any other class that we are interested in)

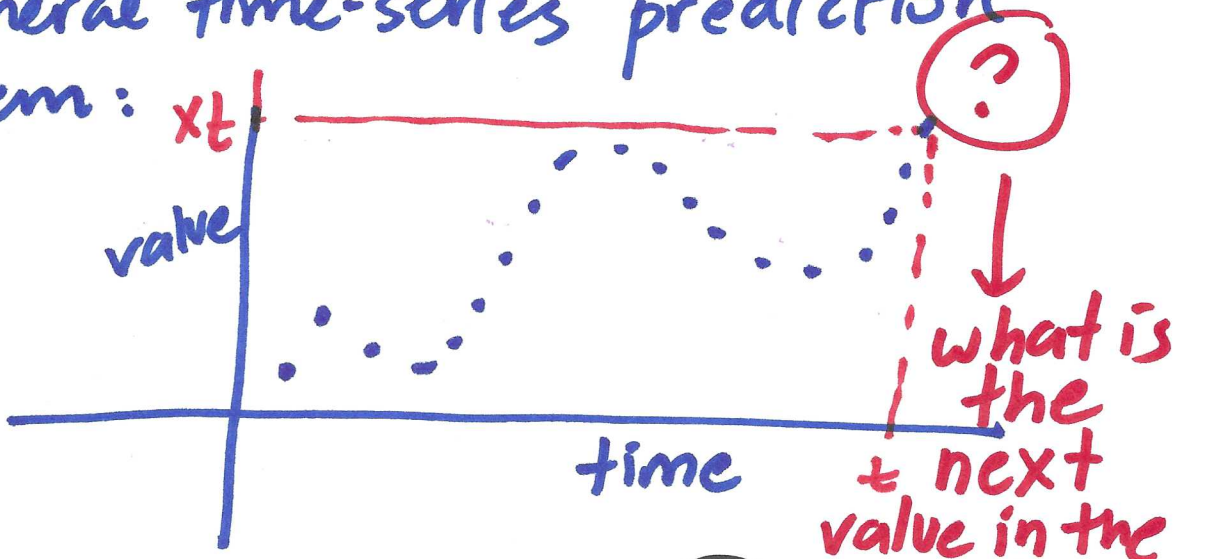
In most of these problems we will need an "END of sequence" word EOS.

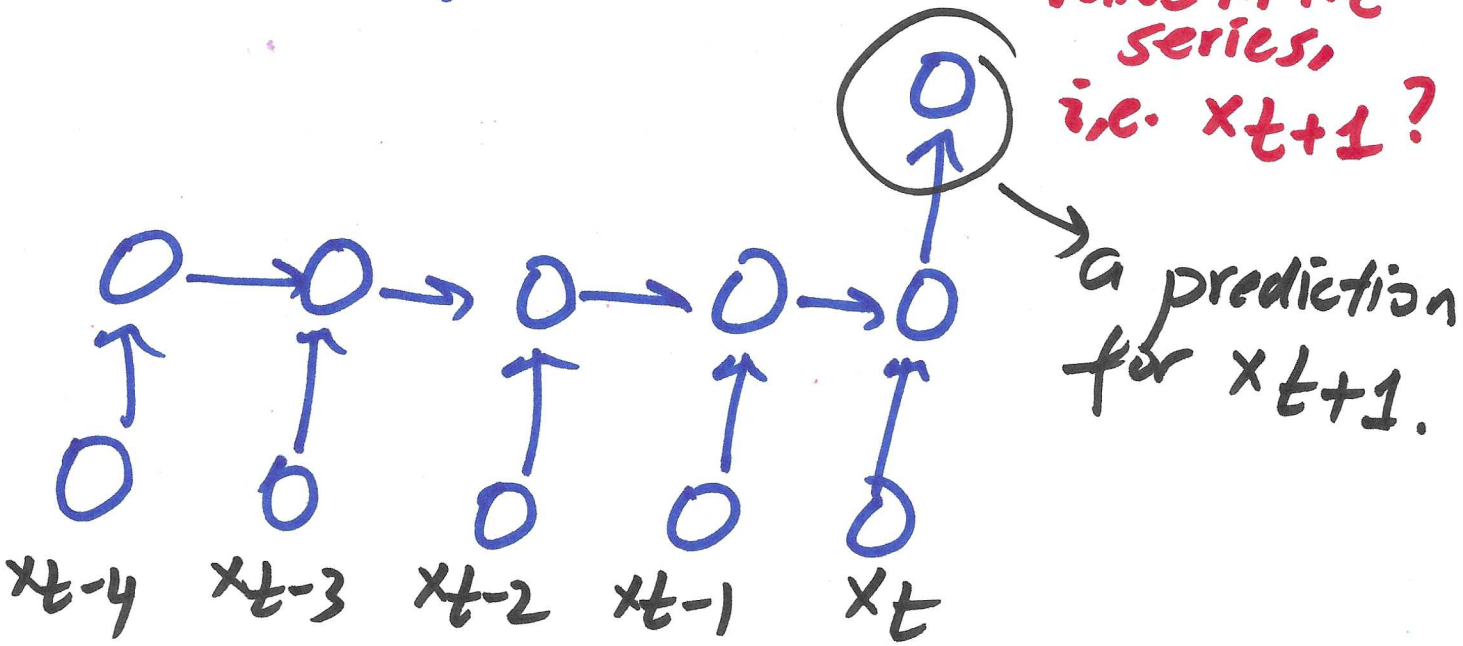
Also, words are typically converted to vectors via an embedding algorithm, before they are input to the RNN.



So, we are not inputting "words" of course but their embedded versions.

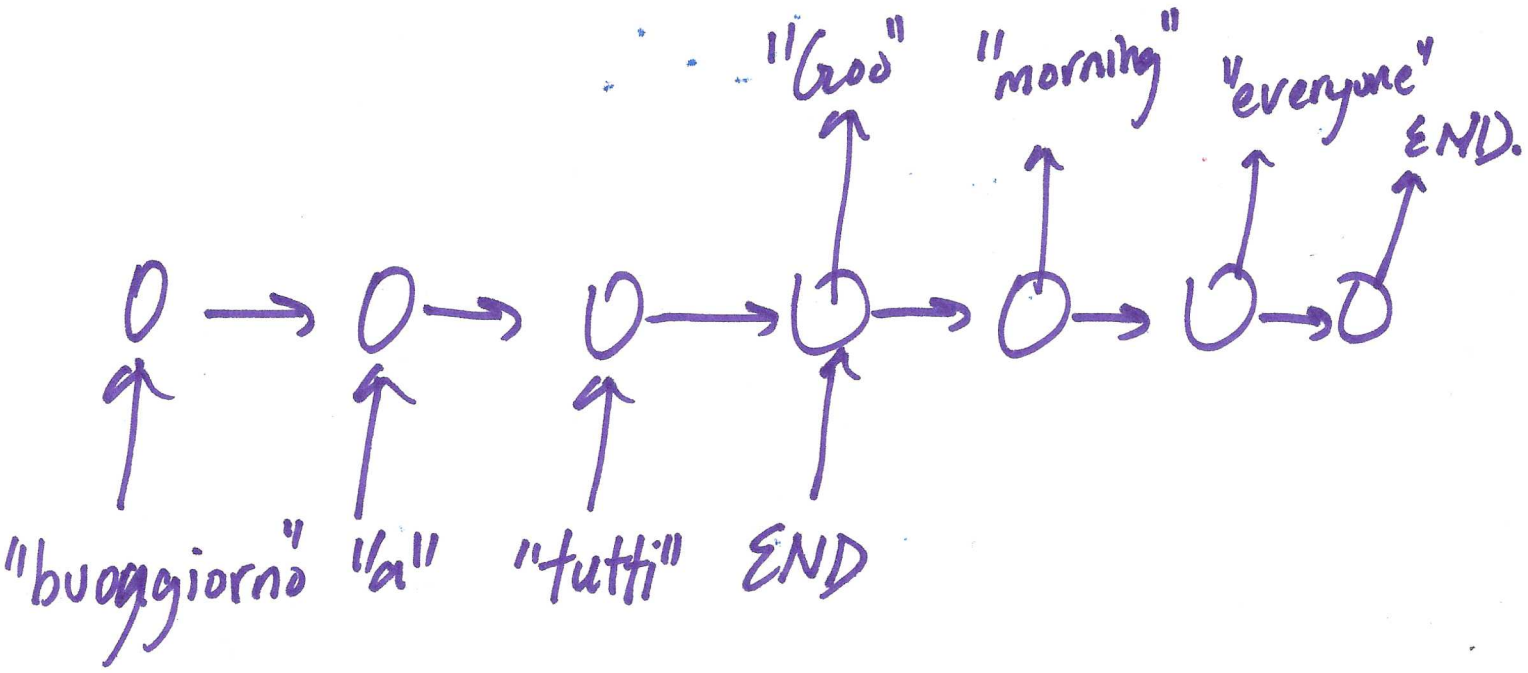
Another application of the many to one architecture could be a general time-series prediction problem:



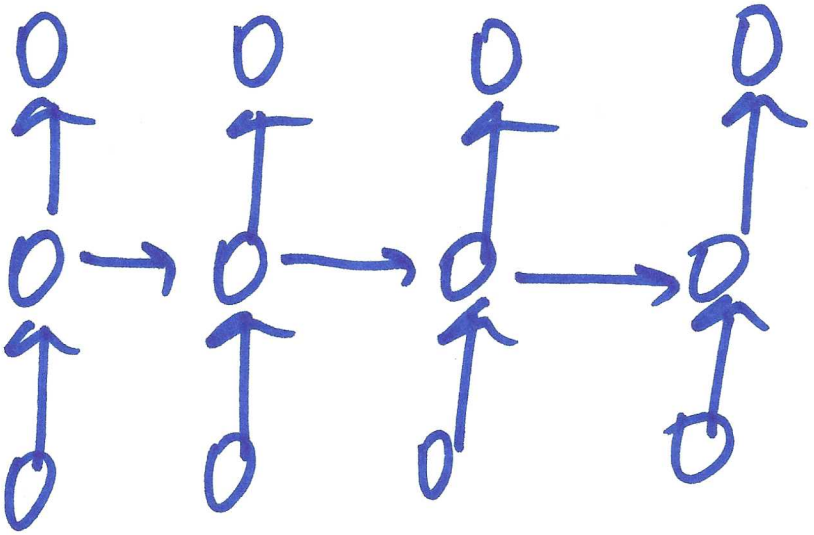


Many-to-many architecture.

Example: Machine translation



Synch. many-to-many architecture.



e.g. frame-by-frame video classification. etc.

many other variants...

Simple RNN architecture

x_t: Input at time $t \in \mathbb{R}^{d \times 1}$

s_t: State at time $t \in \mathbb{R}^{N \times 1}$

y_t: Output at time $t \in \mathbb{R}^{d \times 1}$.

$$s_t = \tanh(Ux_t + Ws_{t-1})$$

(s_{t-1}: recurrence)

$W, U \in \mathbb{R}^{N \times d}$ are trainable weights.

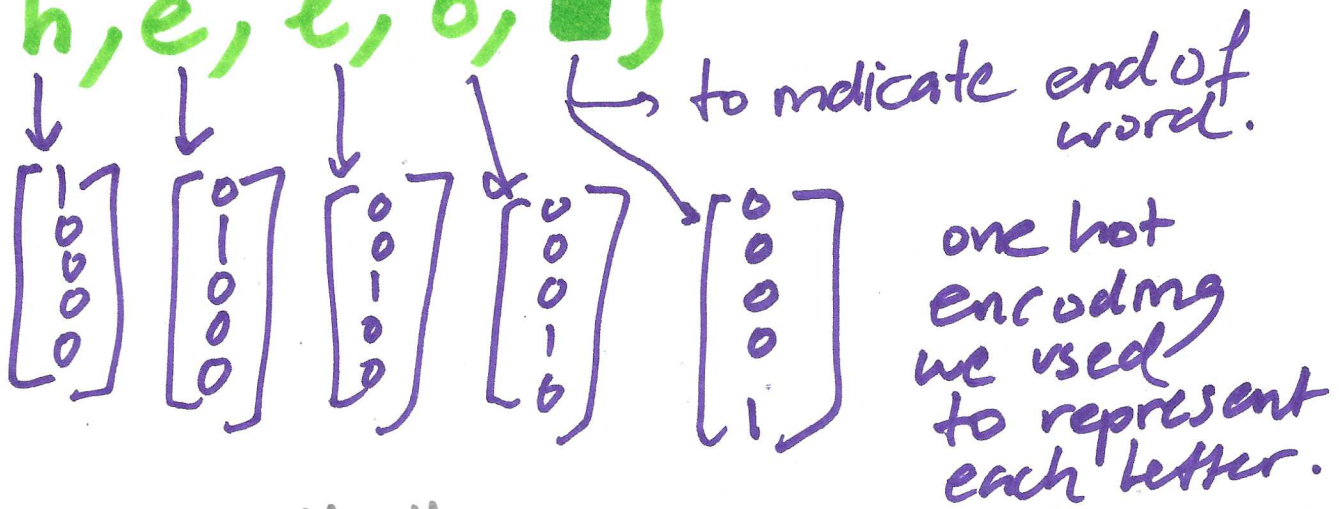
$$y_t = Vs_t$$

(one can also do sigmoid, hard decisions (step func.) etc.

e.g. $y_t = \sigma(Vs_t)$ where σ is the sigmoid act. function.

Ex: An RNN trained to generate words over the alphabet

$\{h, e, l, o, \blacksquare\}$



E.g: $x_1 =$ "e", $x_2 =$ "e"

(First two letters of the word are "ee" what will come after is likely "l", indicating eel (a certain kind of fish))

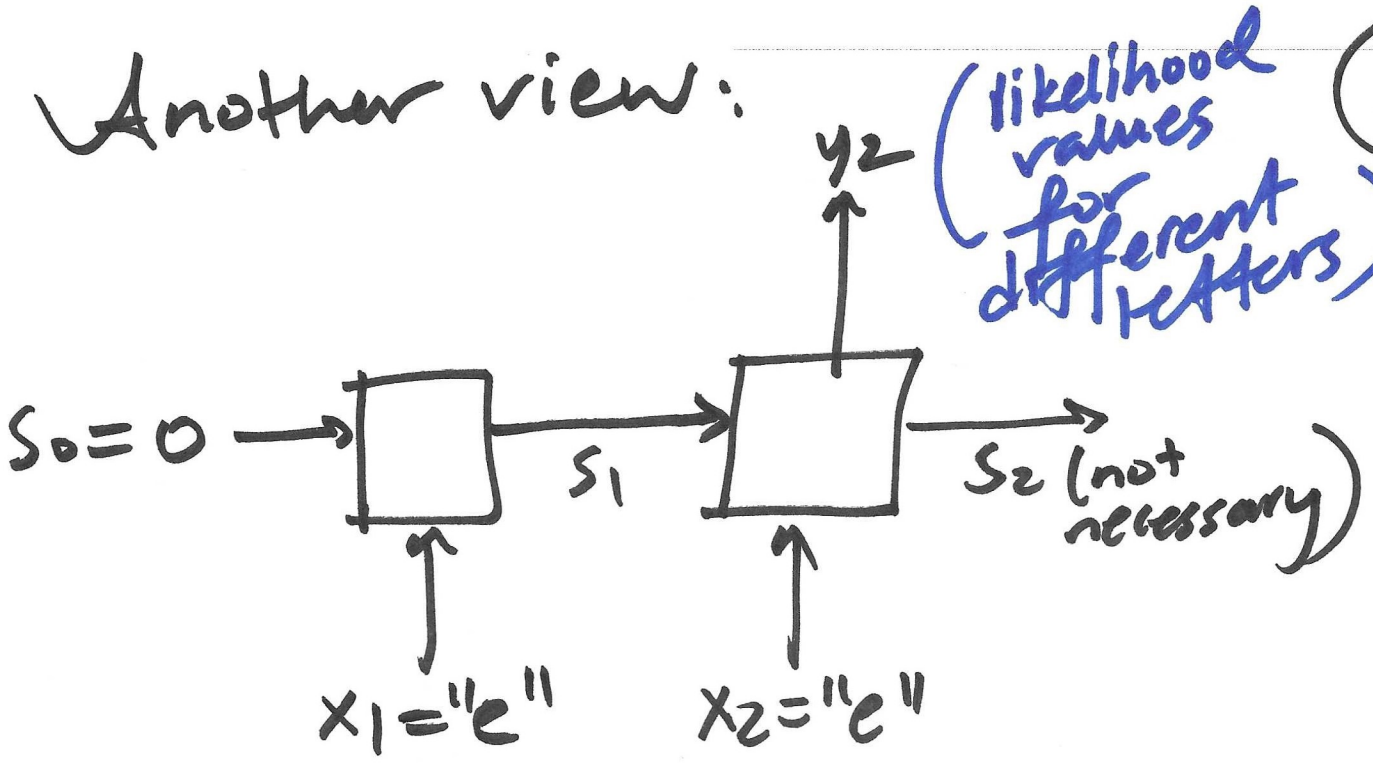
We are interested in predicting what comes after "ee", which we determine through y_2.

$$y_2 = g(Vs_2) = g(V \tanh(Ux_2 + Ws_1))$$

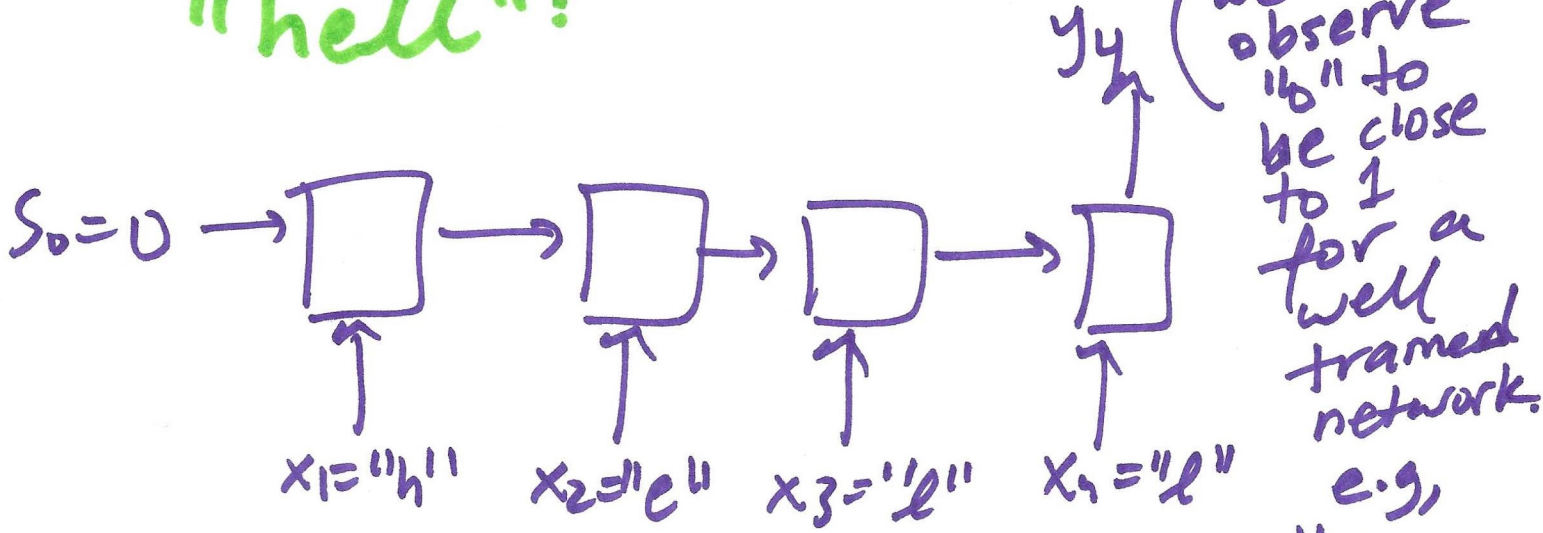
$$s_1 = \tanh(Ux_1 + Ws_0)$$

$s_0 =$ some fixed 5-dim vector (e.g. all zero vector).

Another view:



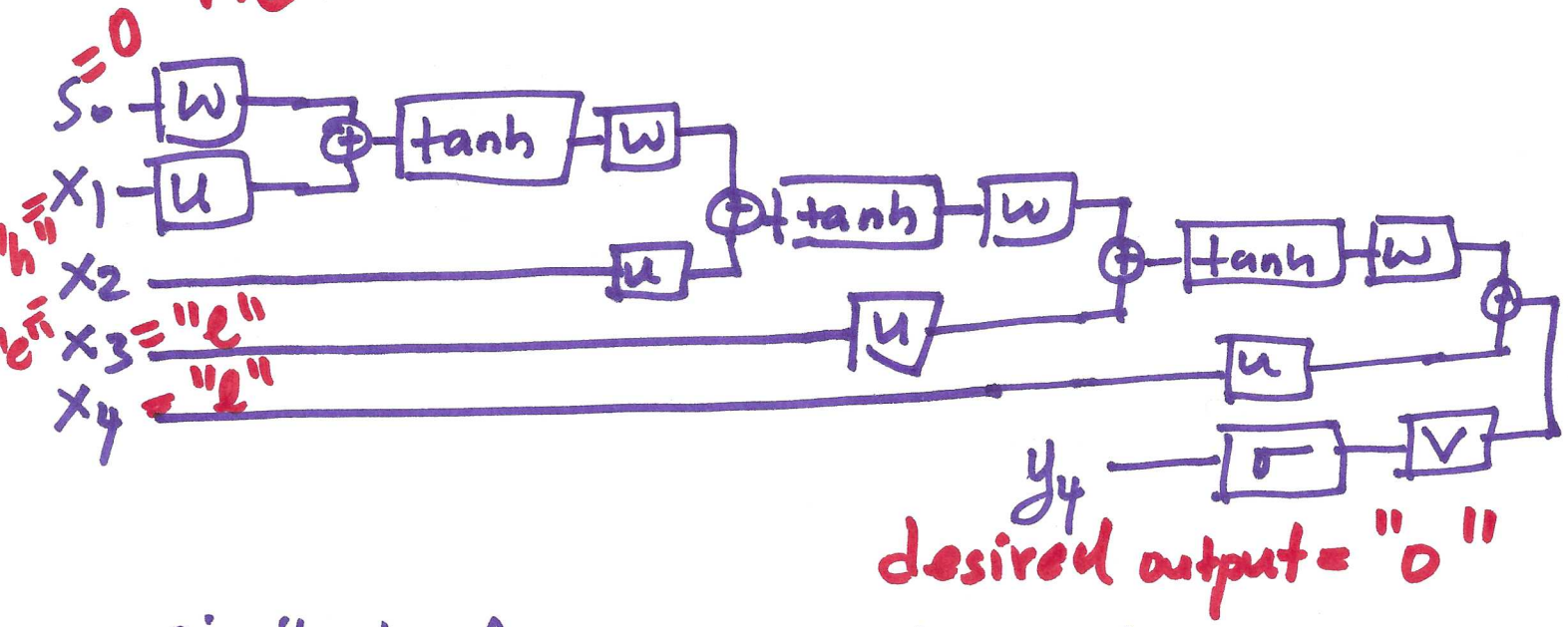
Same network for initial sequence "hell":



Note that "o" = $\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ 0 \end{bmatrix}$

Training RNNs

View each example above as one instance of an ordinary feedforward NN. For example, for the "hello" prediction, the network is:



Similarly for any such ~~desired~~ inputs - desired output pairs the RNN can be unrolled any number of times and trained using standard backpropagation.

The resulting algorithm is referred to as backpropagation through time (BPTT).

Long-Short Term Memory

As we consider longer sequences, we face the vanishing & exploding gradient problems in the standard RNN structure.

LSTMs alleviate (though not entirely solve) the vanishing & exploding ∇ problems of RNNs.

We simply replace the basic recurrence block with another block. Define

x_t: Input at time t

y_t: Output at time t

s_t: State at time t

As usual. LSTMs define several new "gates":

"Forget" Gate: (how much of the previous state to forget)

$$f_t = \sigma(U^f x_t + W^f y_{t-1} + b^f)$$

Input Gate: (how much of the new candidate state to keep)

$$i_t : \sigma(U^i x_t + W^i y_{t-1} + b^i)$$

Output Gate: (how much of the calculated output to expose)

$$o_t : \sigma(U^o x_t + W^o y_{t-1} + b^o)$$

New candidate state:

$$g = \tanh(U^g x_t + W^g y_{t-1} + b^g)$$

New state: → Hadamard (element-wise) product

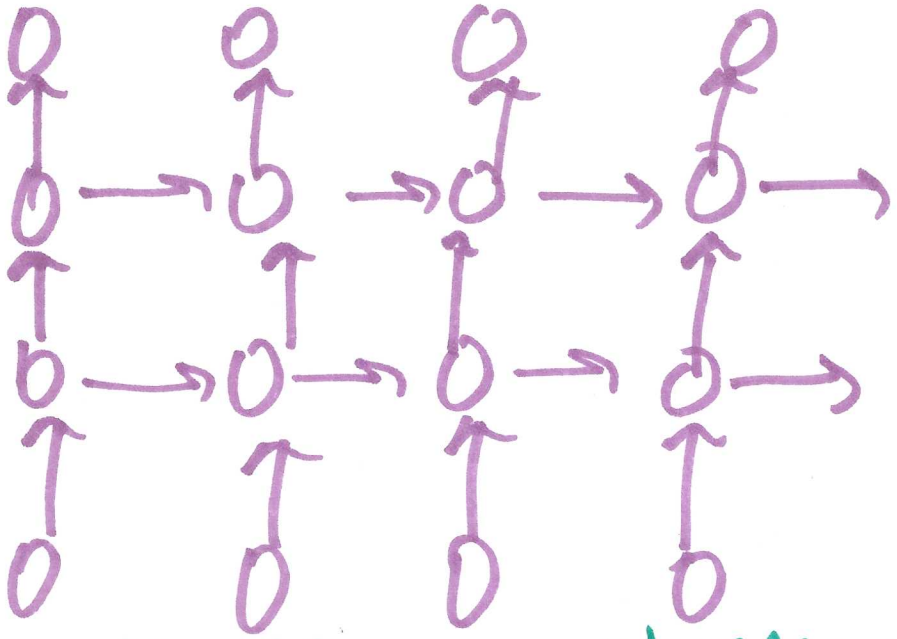
$$s_t = s_{t-1} \odot f_t + g \odot i_t$$

output:

$$y_t = \tanh(s_t) \odot o_t$$

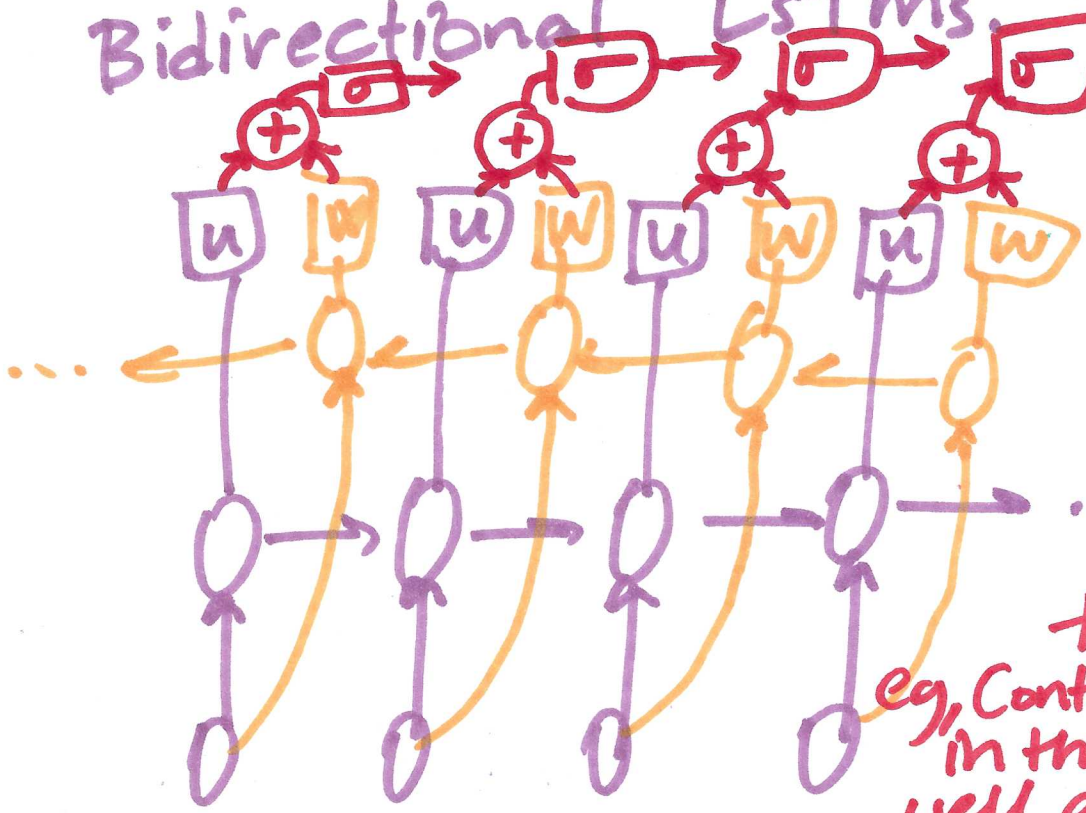
- All weights, biases are trainable.
- Peephole LSTMs include a trainable weight x previous state to the induced local fields.

Stacked LSTMs



* motivation: capturing more complex features (as in deep vs shallow NNs).
* Not much gains to have >3 stacks.

Bidirectional LSTMs



- Information flows in both negative & positive time directions.
- Useful in machine translation.

eg, Context can be hidden in the beginning as well as at the end of a sentence.