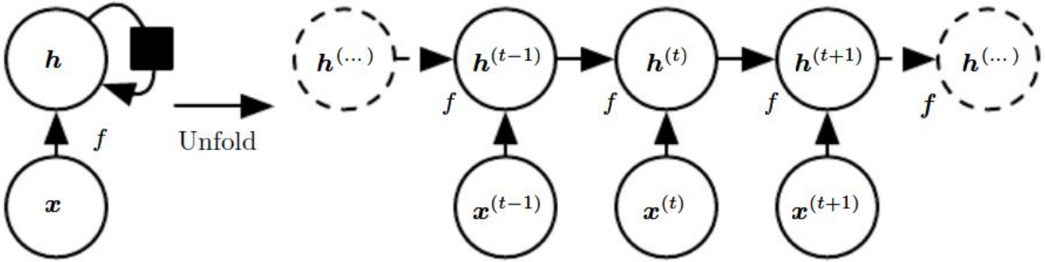
RNNs

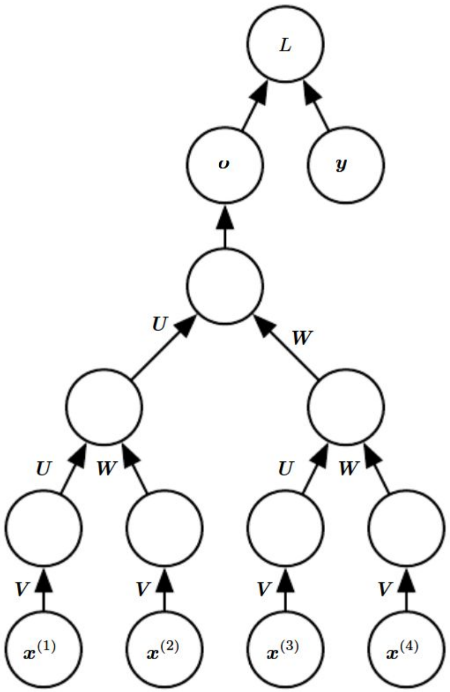
What is an RNN?

- Remember what the ‘R’ means
  - *Recurrent*, not recursive
- An RNN defines a computational chain
- In each step of the chain, the hidden response or *state* is computed from:
  - The previous state
  - The current input
  - Sometimes the previous output



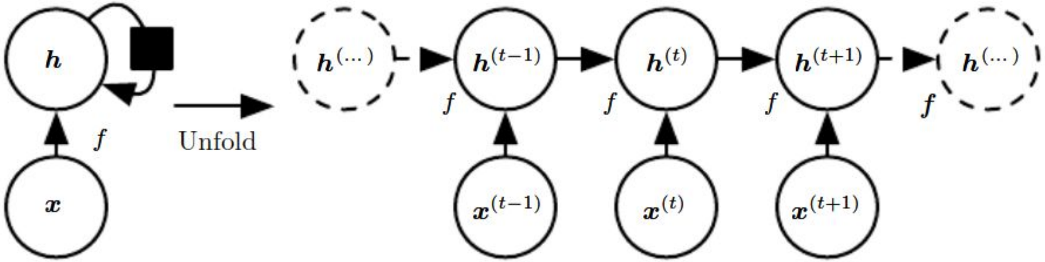
Recursive NNs

- A topic on its own
- Recursively apply the same weights upwards in a tree, as opposed to the chain-like structure of an RNN
- Less commonly used than RNNs



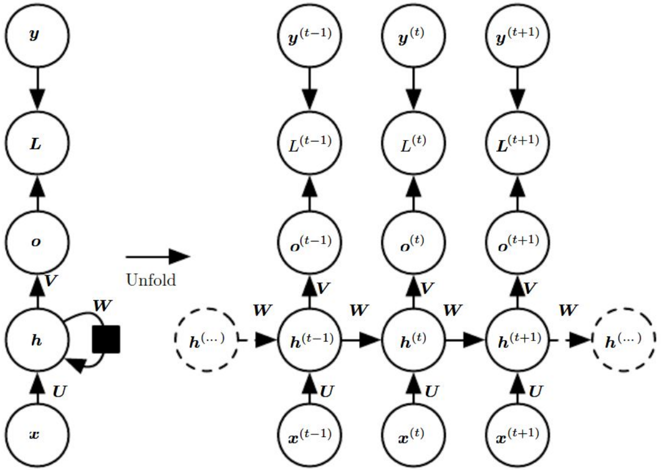
Back to RNNs

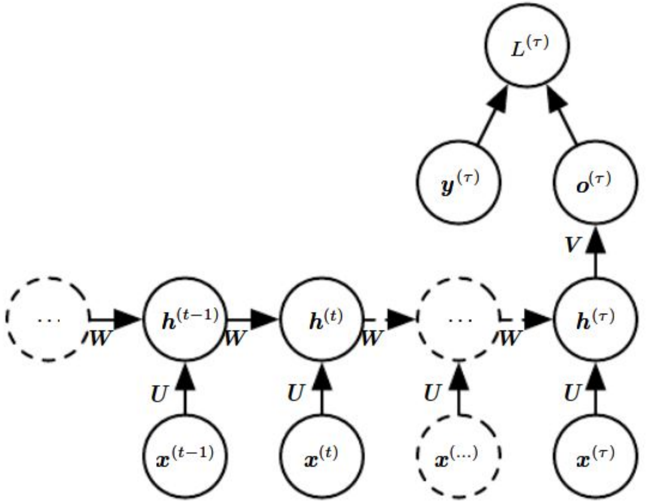
- Remember that, just like in e.g. signal processing, the “time” axis could mean something else
- Think of convolutions
  - Filters move along spatial dimension(s)
- RNNs and CNNs could be used on both kinds of data (time/space)
  - Depends on the task and the kind of data you have



Two basic output modes

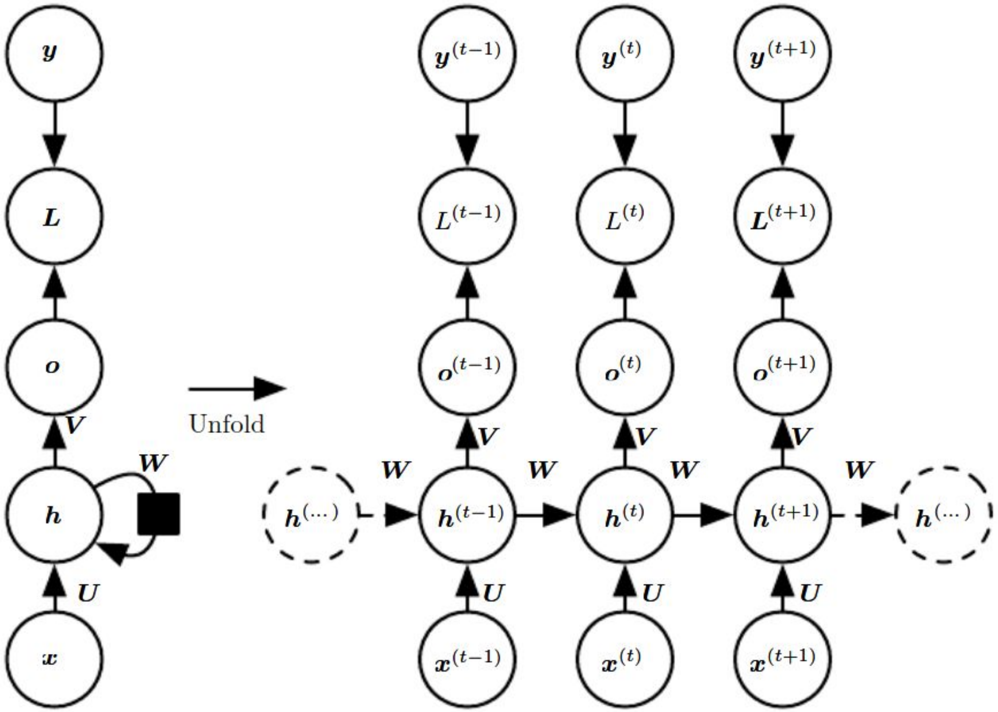
- Sequence to sequence
  - The input **x** and the output **o** are both sequences
- Sequence to single output
  - Input **x** is a sequence, output **o** is a single tensor
  - Think of **o** as the "summary" of the input sequence
- Single input to sequence
  - Later





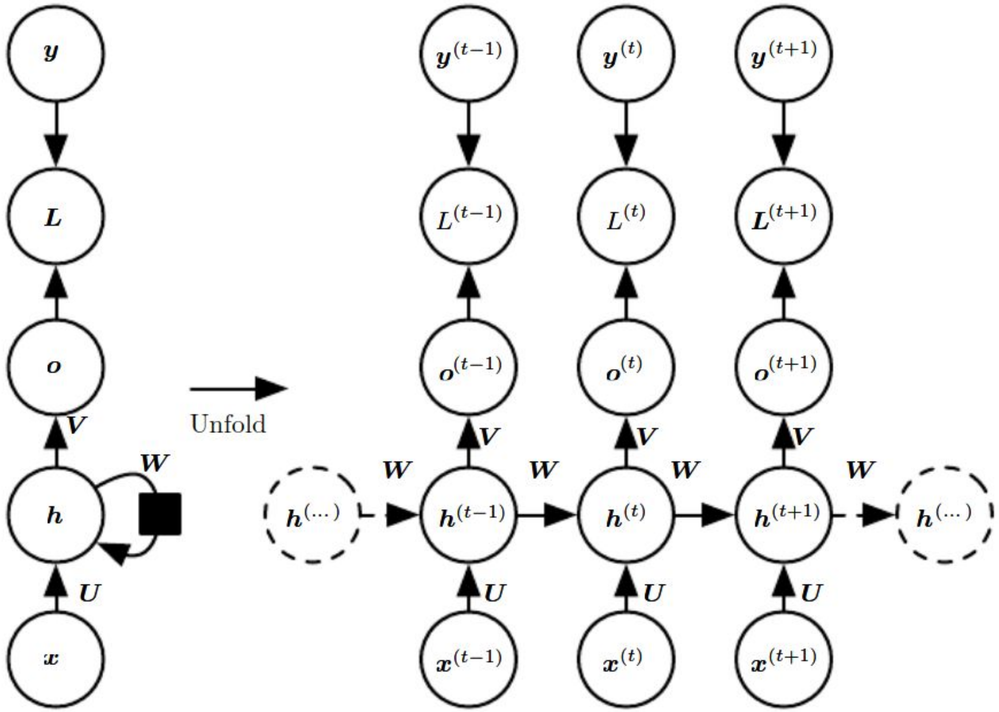
A prototypical RNN

- Now we have different (trainable) weights for the different *transitions*
  - **U:** Input to hidden, just like feedforward FCNs or CNNs



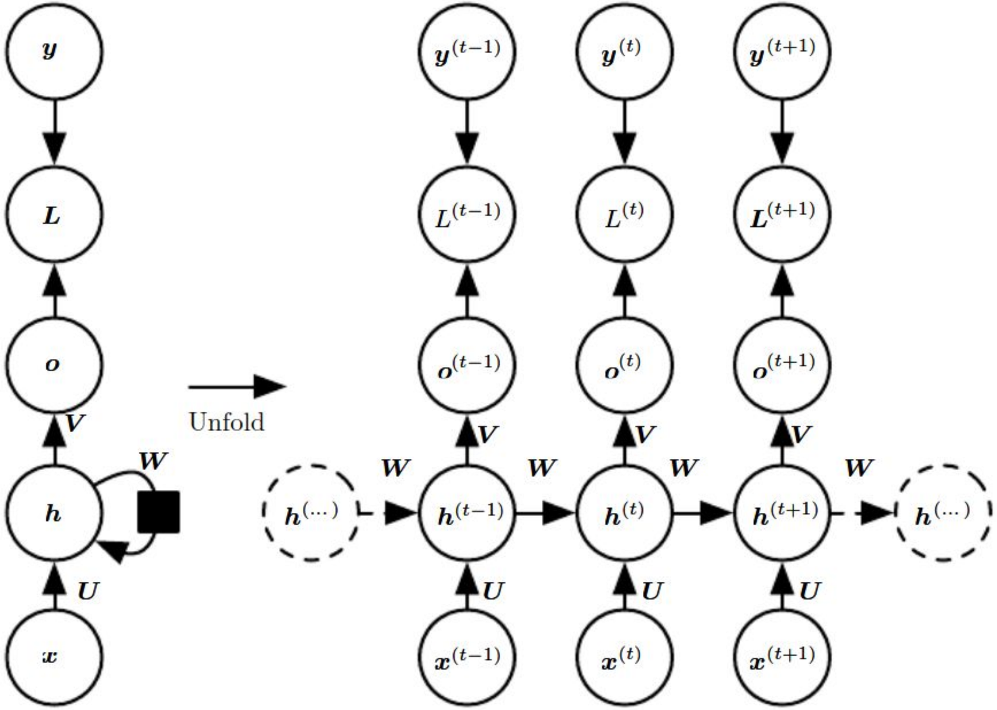
A prototypical RNN

- Now we have different (trainable) weights for the different *transitions*
    - **W:** Hidden to hidden, defining a transition to the next state



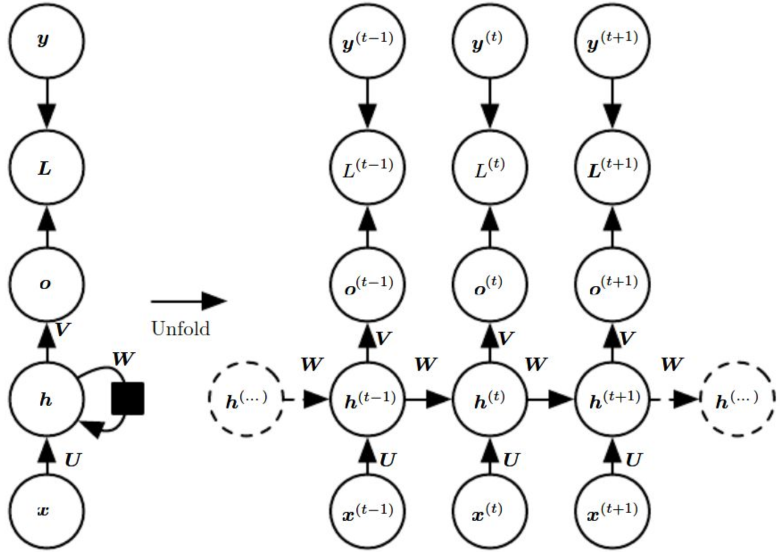
A prototypical RNN

- Now we have different (trainable) weights for the different *transitions*
    - **V:** Hidden to output, again like regular NNs, e.g. used to transform from low-/high-dimensional codes to the correct output type (e.g. classification/regression)



RNN equations

- First the pre-nonlinear activation:
  $$\mathbf{a}^t = \mathbf{W}\mathbf{h}^{t-1} + \mathbf{U}\mathbf{x}^t + \mathbf{b}$$
- Then the non-linearity to get the state:
  $$\mathbf{h}^t = \tanh(\mathbf{a}^t)$$
- Then the transition to the output:
  $$\mathbf{o}^t = \mathbf{V}\mathbf{h}^t + \mathbf{c}$$
- If we're e.g. building a classifier, we get:
  $$\hat{\mathbf{y}}^t = \text{softmax}(\mathbf{o}^t)$$

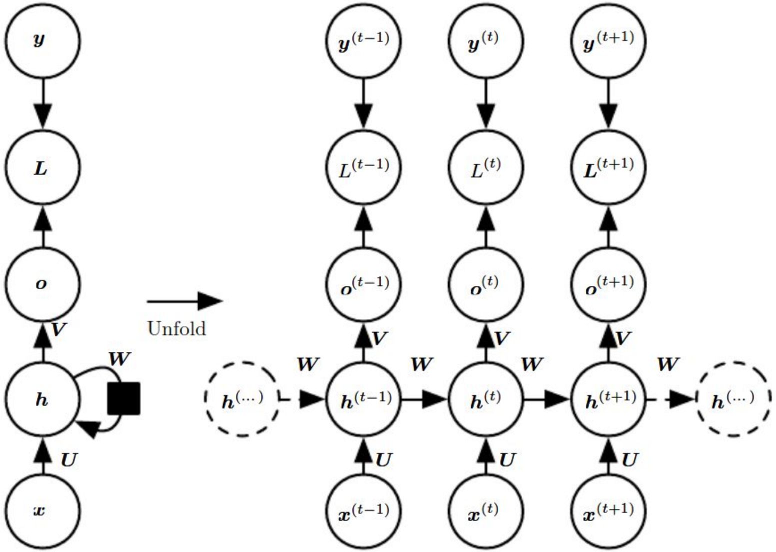


RNN equations

- Note that the update equations:

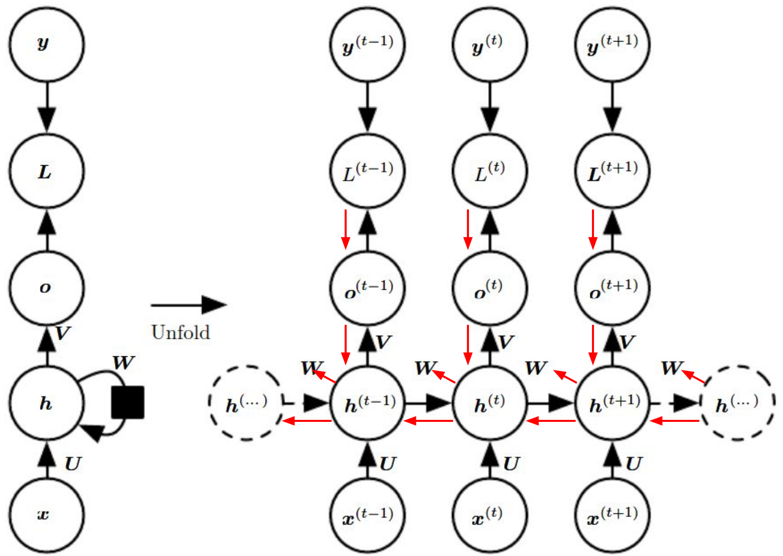
$$
\begin{aligned}
\mathbf{a}^t &= \mathbf{W}\mathbf{h}^{t-1} + \mathbf{U}\mathbf{x}^t + \mathbf{b} \\
\mathbf{h}^t &= \tanh(\mathbf{a}^t) \\
\mathbf{o}^t &= \mathbf{V}\mathbf{h}^t + \mathbf{c} \\
\hat{\mathbf{y}}^t &= \mathrm{softmax}\left(\mathbf{o}^t\right)
\end{aligned}
$$

- are:
  - differentiable (backprop),
  - applied at every time step *t*



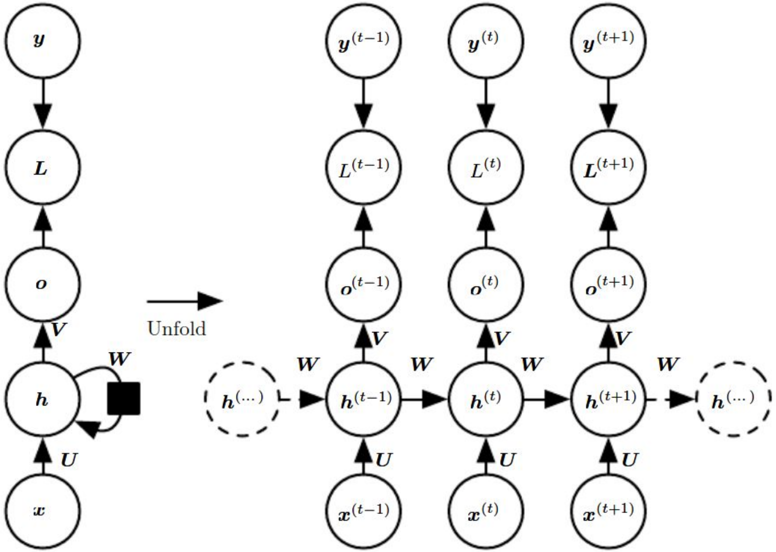
BPTT - Backpropagation Through Time

- Backpropagation for RNNs starts like usual, namely with the delta at the output **o**:
  $\nabla_{\mathbf{o}^t} L = \hat{\mathbf{y}}^t - \mathbf{y}^t$
- Note that the book uses the component-wise notation in Eq. (10.18)
- I use the assumption that the **y**'s are one-hot encoded already, like we did in lecture 4



BPTT

- Now we go back to the gradient for the hiddens
- We start at the final step, called τ
- From last slide, but with τ:
  $\nabla_{\mathbf{o}^\tau} = \hat{\mathbf{y}}^\tau - \mathbf{y}^\tau$
- And now one step back in the chain (not in time), exactly like in lecture 4:
  $\nabla_{\mathbf{h}^\tau} L = \mathbf{V}^\top \nabla_{\mathbf{o}^\tau} L$



BPTT

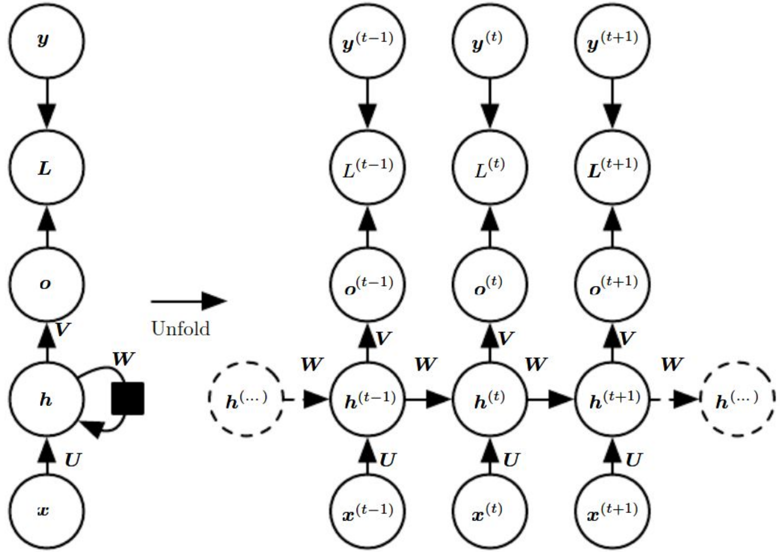
- Using the final hiddens gradient:

  $$\nabla_{\mathbf{o}^\tau} = \hat{\mathbf{y}}^\tau - \mathbf{y}^\tau$$

  $$\nabla_{\mathbf{h}^\tau} L = \mathbf{V}^\top \nabla_{\mathbf{o}^\tau} L$$

- we can now go “back in time” to earlier times $t < \tau$
- The derivations are similar to regular backprop and we end up with:

  $$\nabla_{\mathbf{h}^t} L = \mathbf{W}^\top f'(\mathbf{h}^{t+1}) \nabla_{\mathbf{h}^{t+1}} L + \mathbf{V}^\top \nabla_{\mathbf{o}^t} L$$

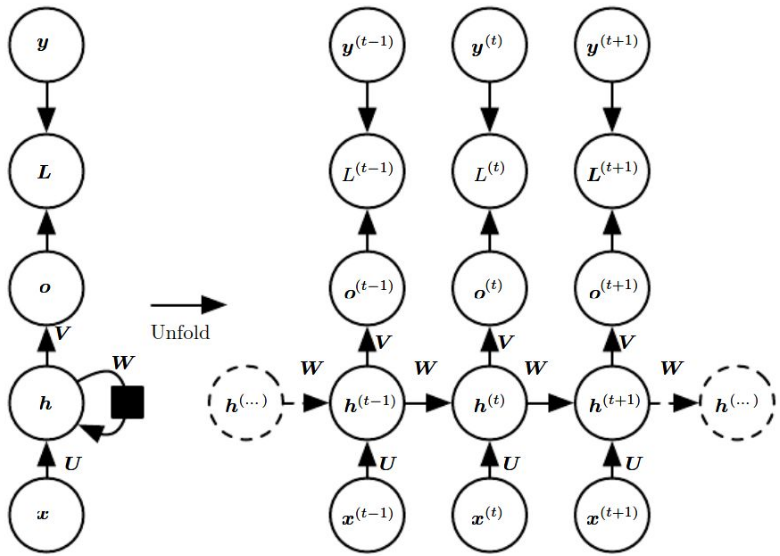


BPTT

- To get all the parameter gradients, we sum up over all time steps
- First the output and hidden biases:

$$\nabla_{\mathbf{c}} L = \sum_t \nabla_{\mathbf{o}^t} L$$

$$\nabla_{\mathbf{b}} L = \sum_t f'(\mathbf{h}^t) \nabla_{\mathbf{h}^t} L$$



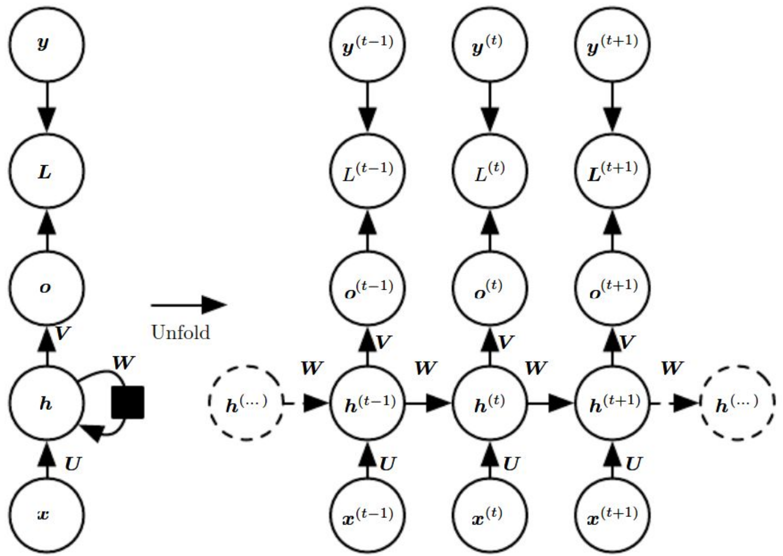
BPTT

- To get all the parameter gradients, we sum up over all time steps
- And now the weight matrices:

$$\nabla_{\mathbf{V}} L = \sum_t \nabla_{\mathbf{o}^t} L \cdot (\mathbf{h}^t)^\top$$

$$\nabla_{\mathbf{W}} L = \sum_t f'(\mathbf{h}^t) \cdot \nabla_{\mathbf{h}^t} L \cdot (\mathbf{h}^{t-1})^\top$$

$$\nabla_{\mathbf{U}} L = \sum_t f'(\mathbf{h}^t) \cdot \nabla_{\mathbf{h}^t} L \cdot \mathbf{x}^t$$



BPTT

- All together now:

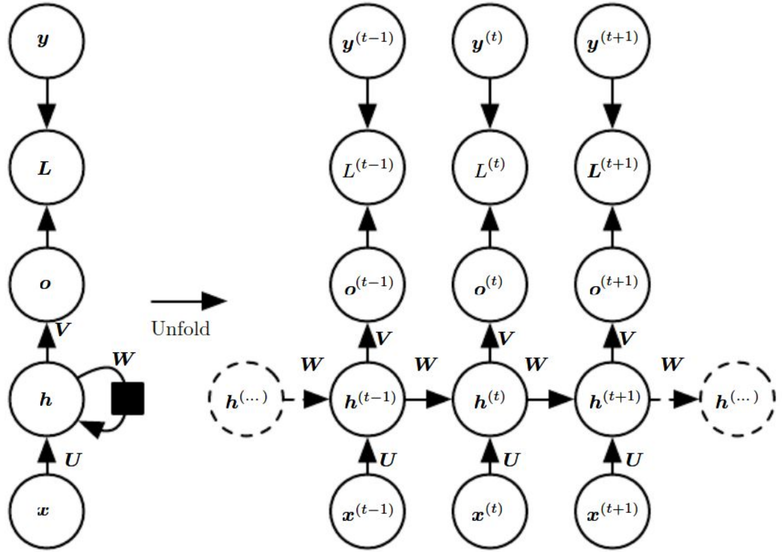
$$\nabla_{\mathbf{c}} L = \sum_t \nabla_{\mathbf{o}^t} L$$

$$\nabla_{\mathbf{b}} L = \sum_t f'(\mathbf{h}^t) \nabla_{\mathbf{h}^t} L$$

$$\nabla_{\mathbf{V}} L = \sum_t \nabla_{\mathbf{o}^t} L \cdot (\mathbf{h}^t)^\top$$

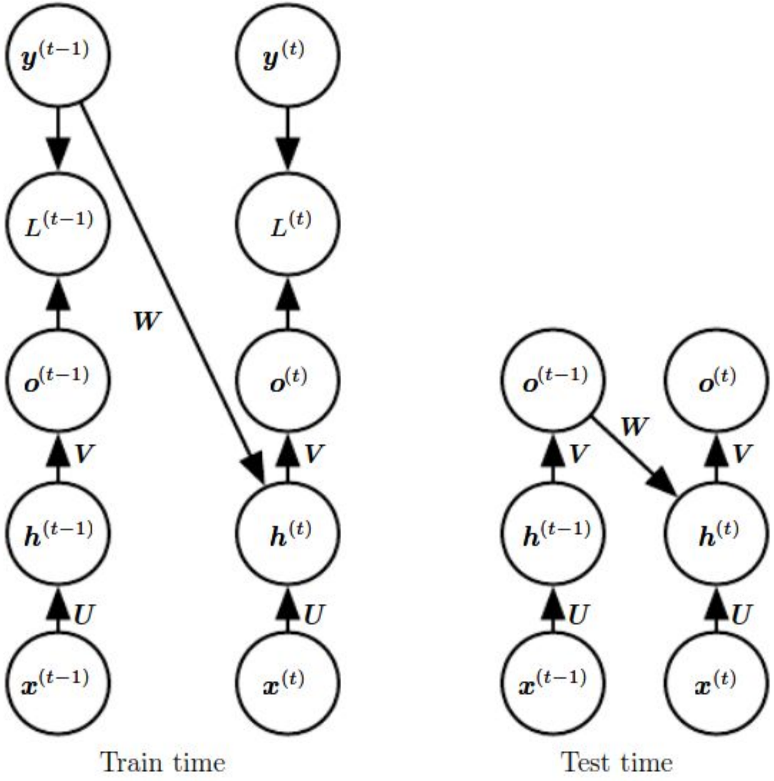
$$\nabla_{\mathbf{W}} L = \sum_t f'(\mathbf{h}^t) \cdot \nabla_{\mathbf{h}^t} L \cdot (\mathbf{h}^{t-1})^\top$$

$$\nabla_{\mathbf{U}} L = \sum_t f'(\mathbf{h}^t) \cdot \nabla_{\mathbf{h}^t} L \cdot \mathbf{x}^t$$



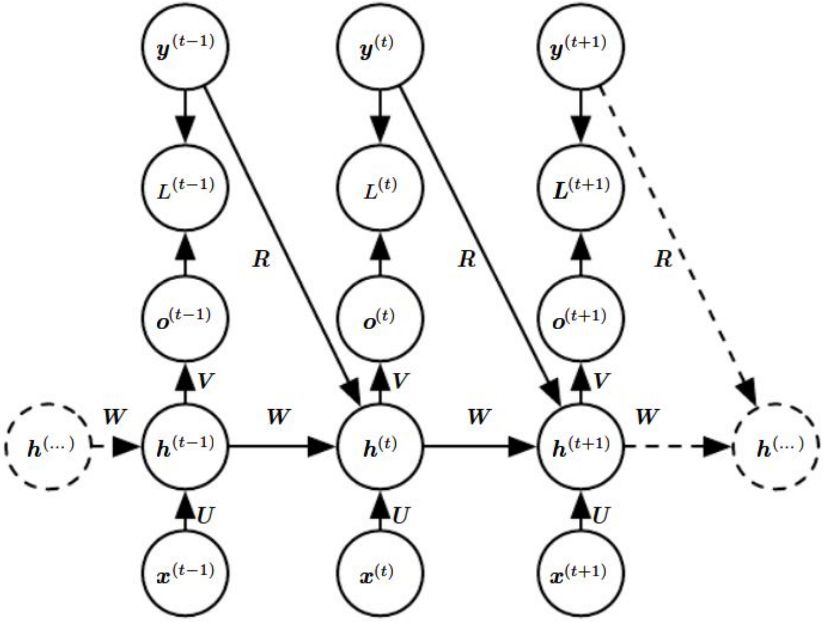
From sequence to sequence

- For some tasks, it is beneficial to know the previous output when predicting the next



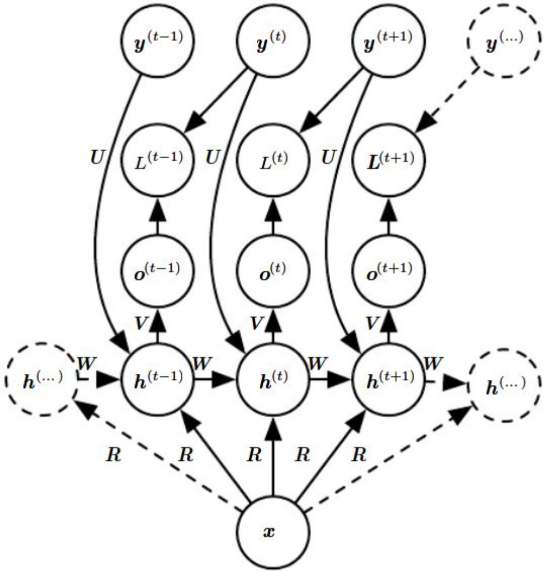
From sequence to sequence

- And we can of course add hidden connections also:



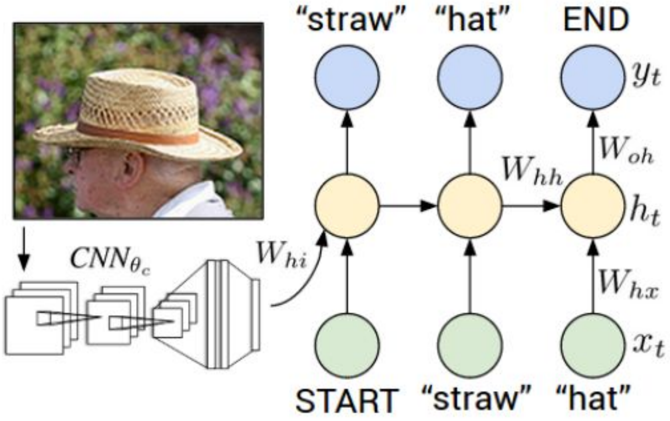
From single input to sequence

- The general structure is as follows:
    - Some earlier process has produced a "summary" or a *context* **x** that you want to transform into a sequence
    - In general, **x** can couple to all hidden states, but often only at the first time step
- During training, learn to predict the *next* output
- During testing, you have only the **o**'s, which you use instead of the **y**'s to drive the transitions



From context to sequence

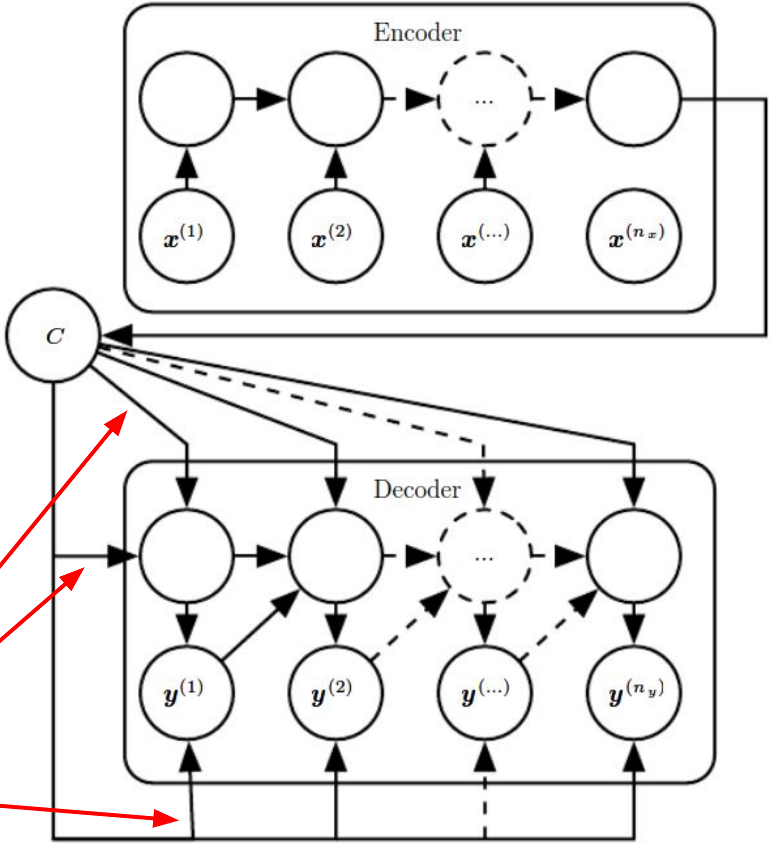
- Here’s a specific example where x is only fed into the first hidden state:



https://cs.stanford.edu/people/karpathy/deepimagesent

Encoder-decoder RNNs

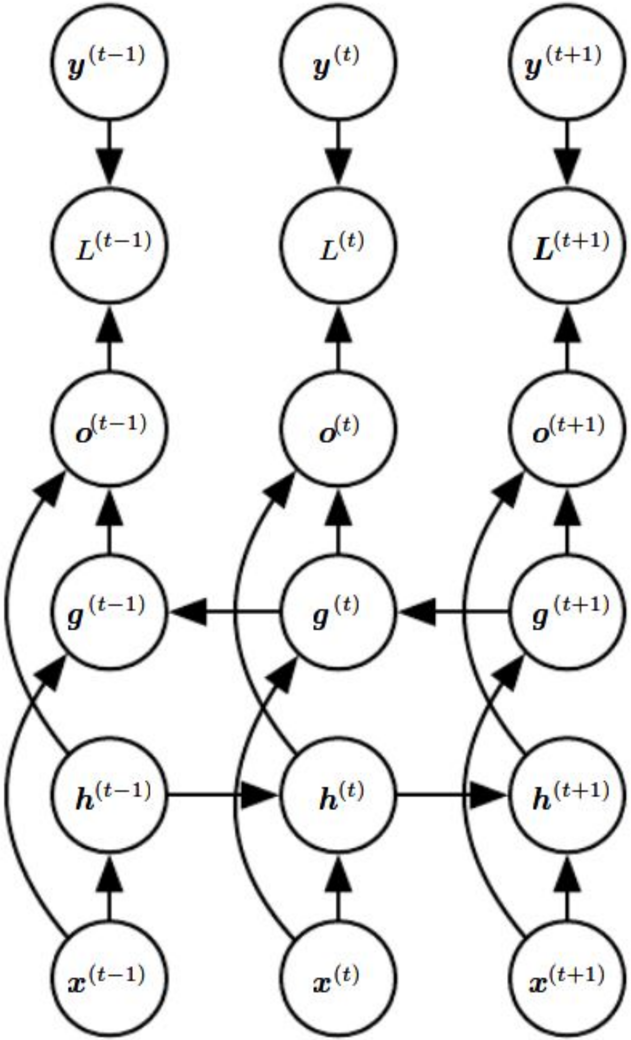
- We have now seen RNNs for:
  - Sequence to sequence
  - Sequence to single output
  - Single input (context) to sequence
- What if both input/output sequences can have variable and different lengths?
  - In general, this structure is called an encoder-decoder RNN
  - These RNNs also work by summarizing the input sequence in one/multiple *context* vectors(s) **C**



These are not necessarily all active!

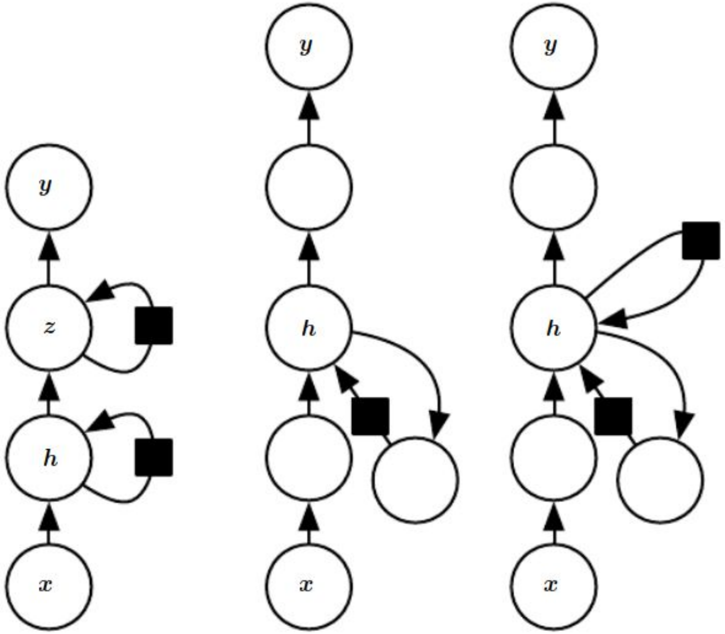
Bidirectional RNNs

- Uses both a forward and a backward “stream” of hidden states **h** and **g**
- The input **x** maps to both hidden streams
- Both hidden streams map to the output
- This allows for a better “holistic” understanding of the input sequence
    - Contrast this by only knowing about the past (**h**)



Deeper RNNs

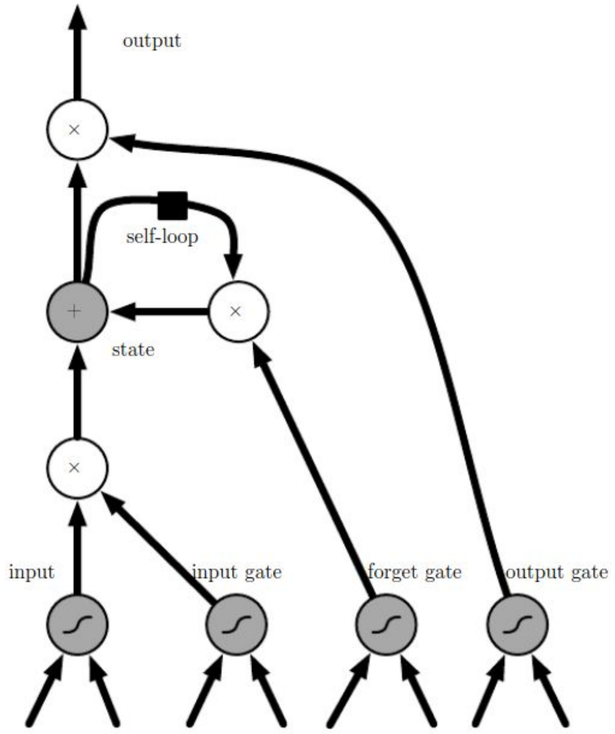
- It's natural to consider whether an RNN could be improved by making it deeper
- Three strategies:
    - Add more hidden states that connect to each other over time (a)
    - Add non-recurrent, regular layers (b)
    - In (c), the structure in (b) is augmented with a *skip connection*, because (b) causes even more problems for BPTT



(a) (b) (c)

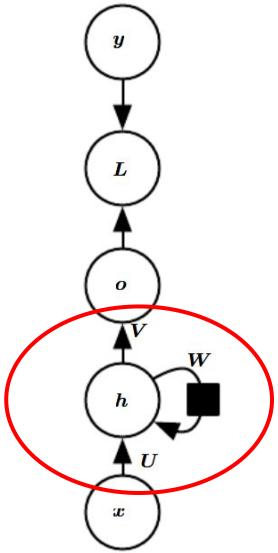
LSTMs

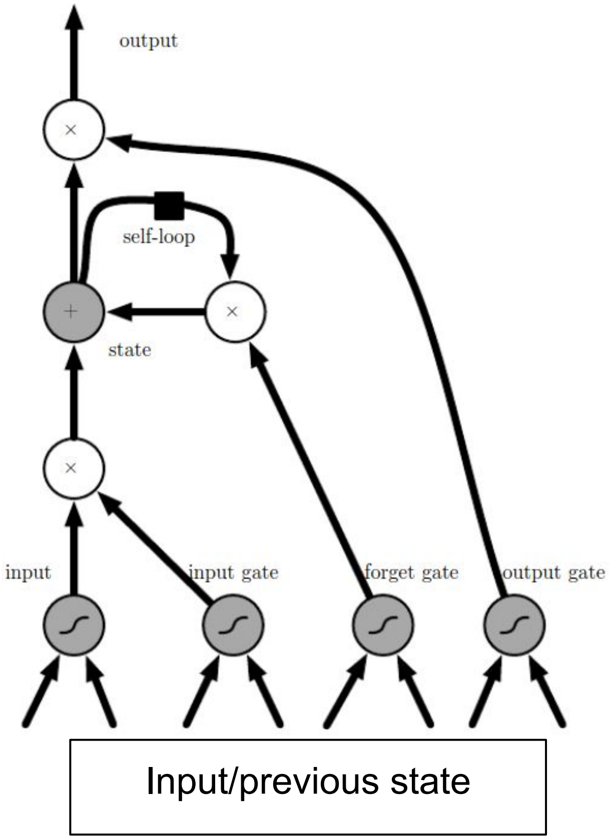
- Long short-term memory “cells” are a special type of hidden units for RNNs
- LSTM and similar *gated* units have proven much better when running the same computations through multiple time steps



LSTMs

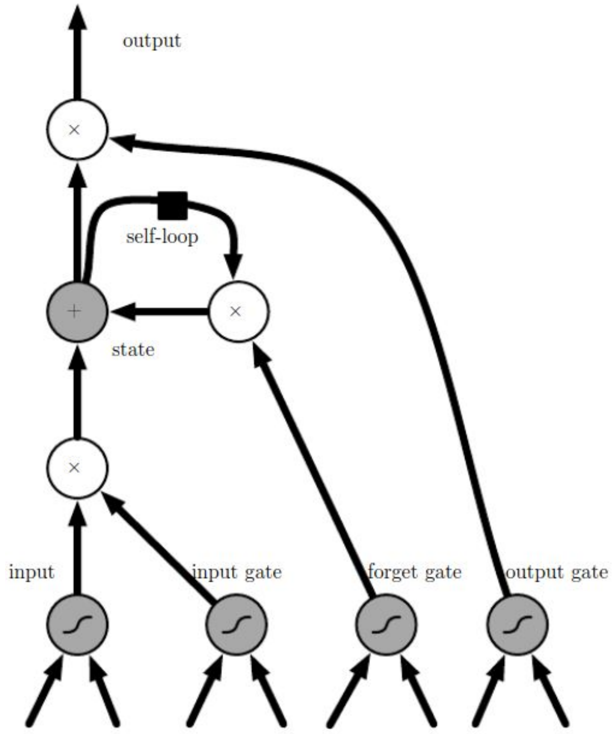
- The basic principles - compared to classical RNN cells - is the use of *gates*





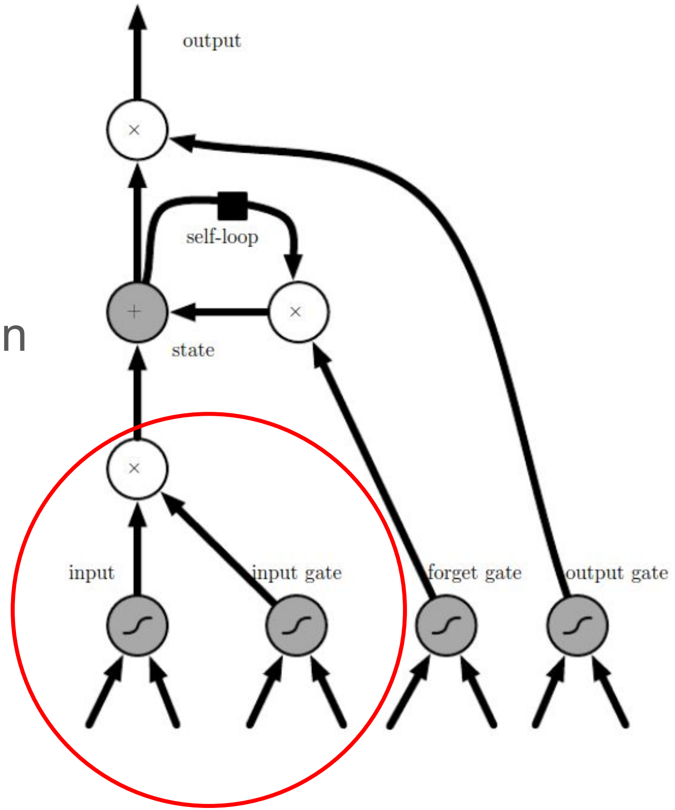
LSTMs

- The input is like before
  - The tanh function is standard, but any non-linearity can be used
- After all the important operations, the gates are used to control how much of the information is "passed through the gate"
  - Therefore, they are always sigmoids [0,1]



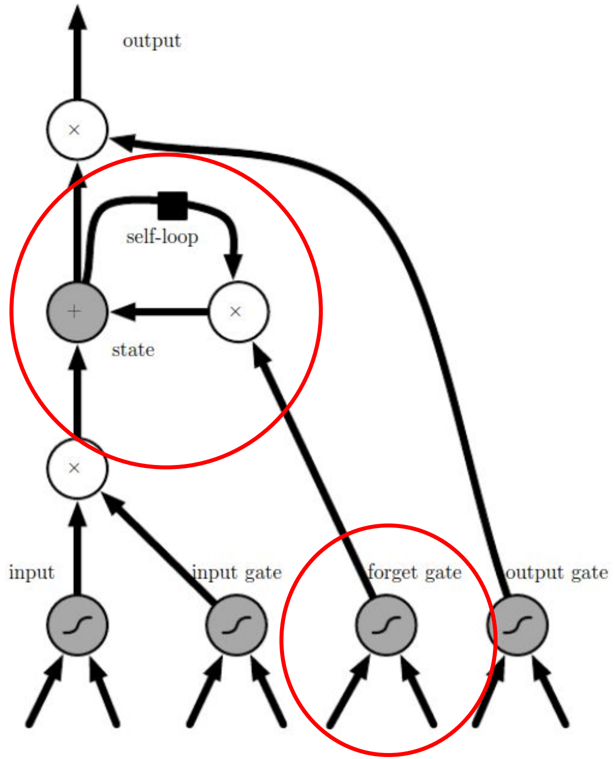
LSTMs

- The input gets the current input **x** and the previous state **h**
- These are squashed, often using tanh
- The input gate uses a sigmoid to weight the influence of this input for the state computation



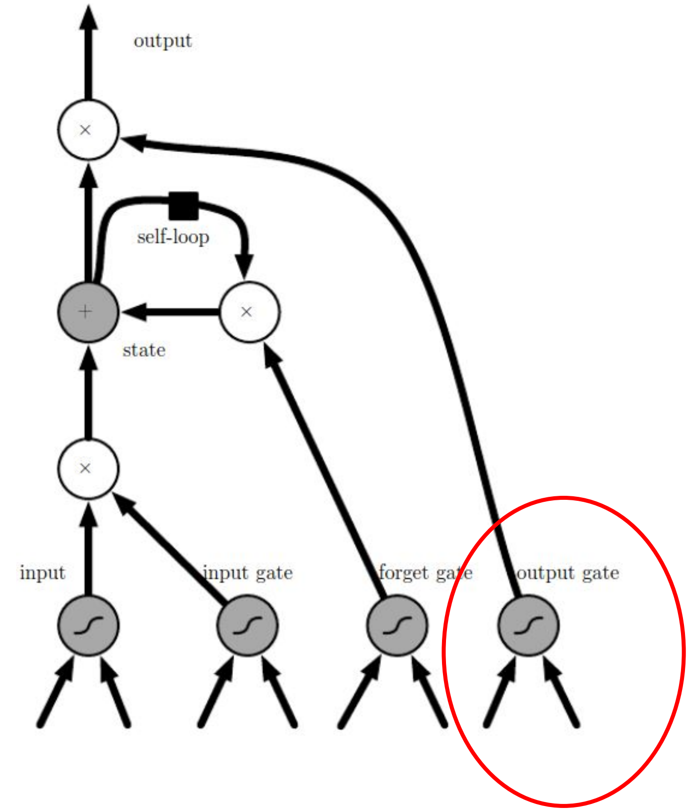
LSTMs

- The state is “leaky” and is a function of the previous state
- This is much like momentum, where the current value is accumulated as a running average of the past history
- In LSTMs, however, the momentum coefficient is dynamic and implemented in the forget gate



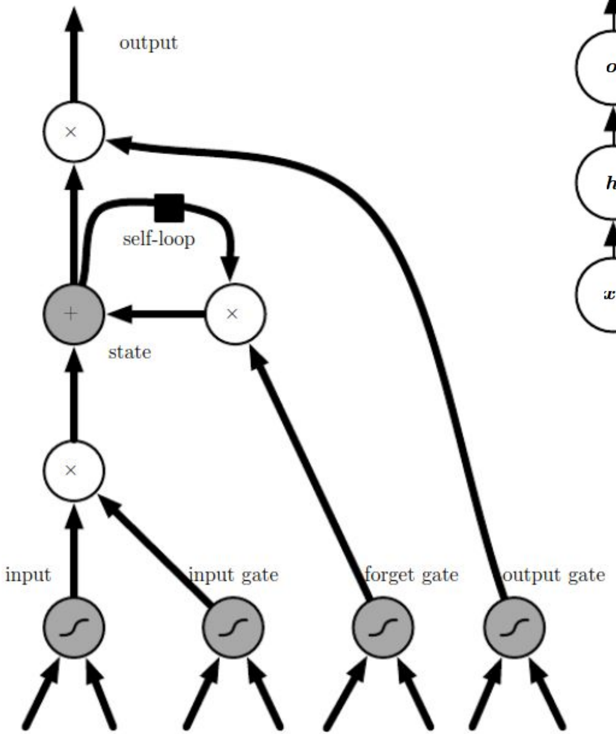
LSTMs

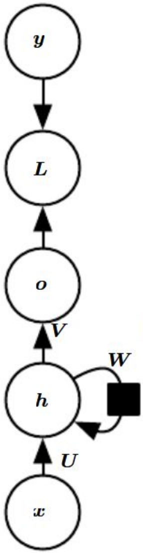
- The output gate is trivial
- It simply controls which parts of the state are important for the current output



LSTMs

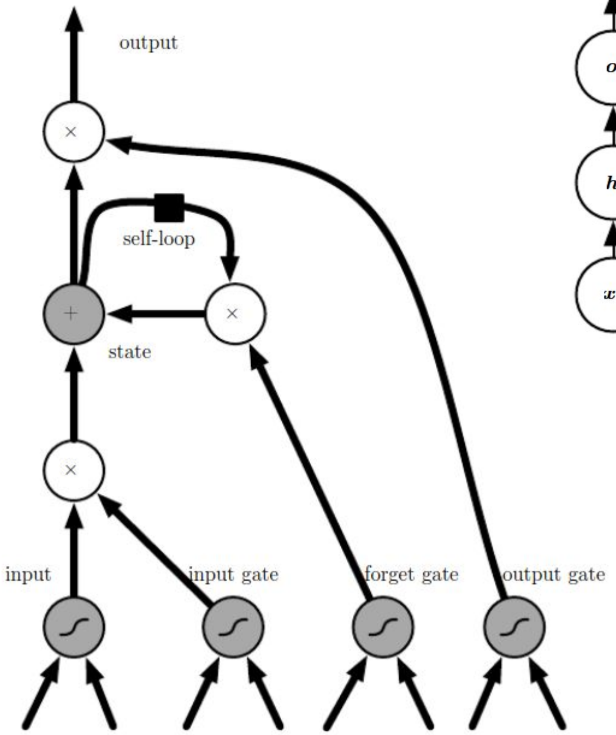
- On the number of parameters
- Each of the bottom blocks couples to current input and previous state with their own weights: $\mathbf{U}, \mathbf{W}, \mathbf{U}^g, \mathbf{W}^g, \mathbf{U}^f, \mathbf{W}^f, \mathbf{U}^o, \mathbf{W}^o$
- - and of course bias vectors: $\mathbf{b}, \mathbf{b}^g, \mathbf{b}^f, \mathbf{b}^o$
- Add to that the increased number of non-linearities
- This makes LSTMs significantly more computationally expensive - but it pays off!

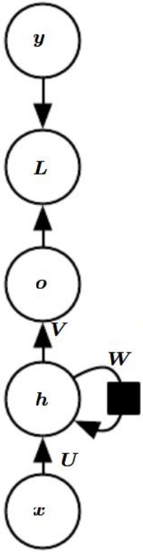




LSTMs

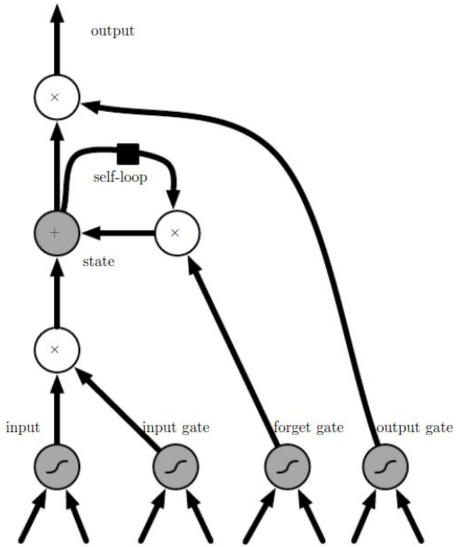
- Let's take a look at the LSTM layer in pytorch: https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html





Convolutional RNNs

- These layers allow for processing of sequences of images
- All the four weight matrix pairs are now replaced by pairs of filter banks
- Available directly in TF/Keras, but not in pytorch
  - Not hard to implement, though!

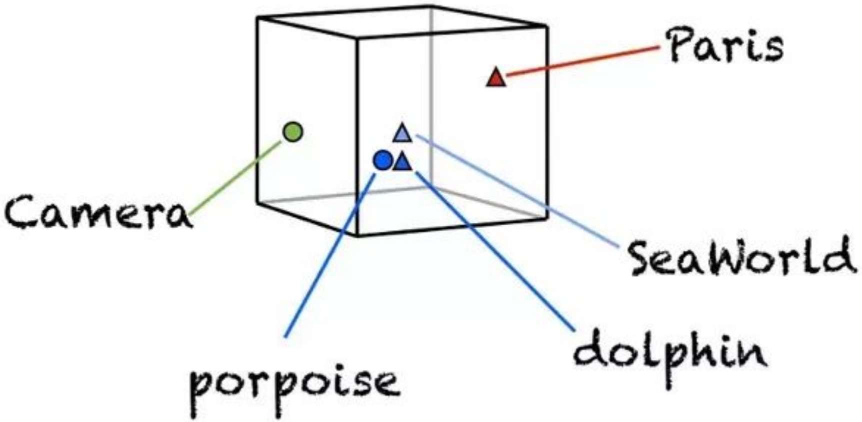


Embeddings

- A problem often encountered in RNNs is how to represent discrete inputs
- Say you want to process a sequence of words or characters
  - Example: given an input text, determine who wrote it
- Basic representation
  - Characters: raw ASCII code in [0,255]
  - Words: ???
- The solution: *embeddings*
  - First create an indexing of the N most frequent words (N being a high number)
  - Then map each of these integers in [0,N[to a high-dimensional real-valued vector
  - Neural nets like this kind of data much better than integers
  - In pytorch, this is achieved using the `Embedding` layer

Embeddings

- Say you have a vocabulary of 20k most frequent words
- You can now embed these words in a 128-dimensional vector space using simple indexing
- Just randomly initialize a 20k-by-128 matrix
- Use the input word index to get the corresponding row



https://www.kaggle.com/sbongo/do-pretrained-embeddings-give-you-the-extra-edge

Embeddings

- Let’s have a look: https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html
- Note also that the initial values for the rows may not be optimal representations of the words
- Therefore, the whole matrix is actually made trainable by default, in order to improve the embeddings during training
- Taking this to the next level, you can even download pre-trained word embedding models that give you even better word vectors for your process

Last three lectures

- Sparse reading
- Papers and a bit from the book
- Limited talk from my side
- **Project**

Challenge

- Email spam classifier
- Bonus: Number sorter