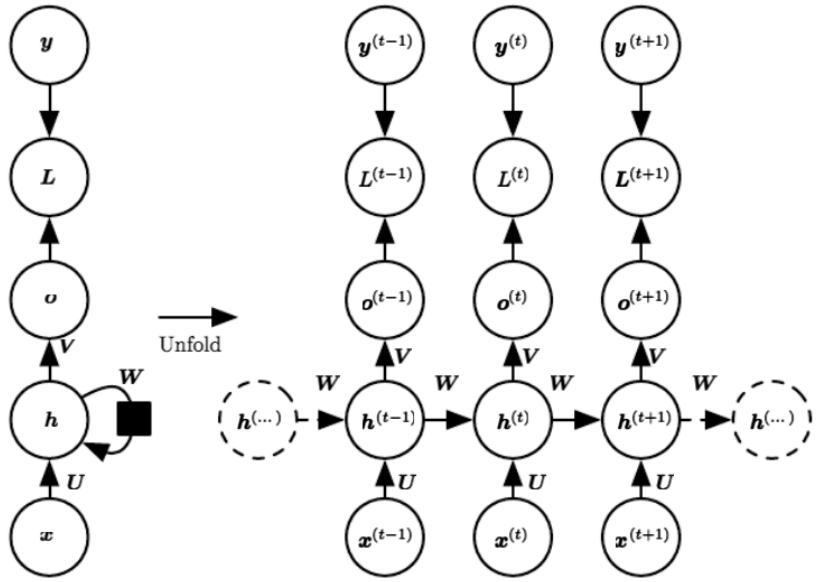
Recurrent Neural Networks(RNNs)

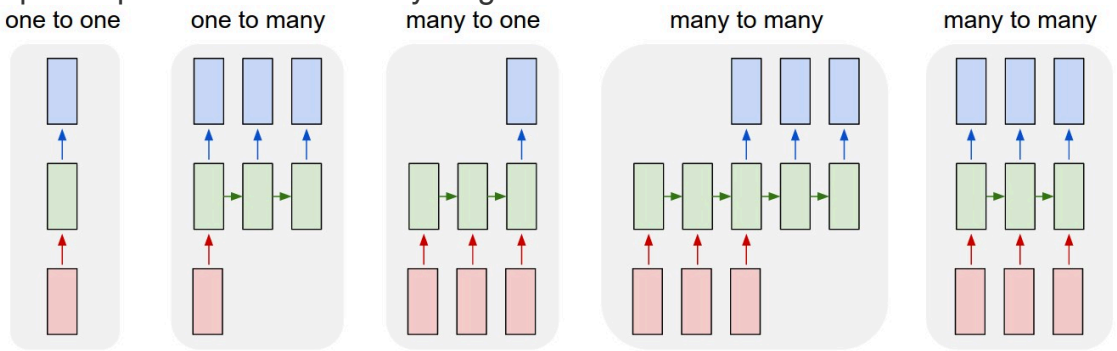
- RNNs are neural networks used to model data in the form of a sequence



- Inputs x(t), hidden states h(t), output o(t), weight matrices U,V,W.

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

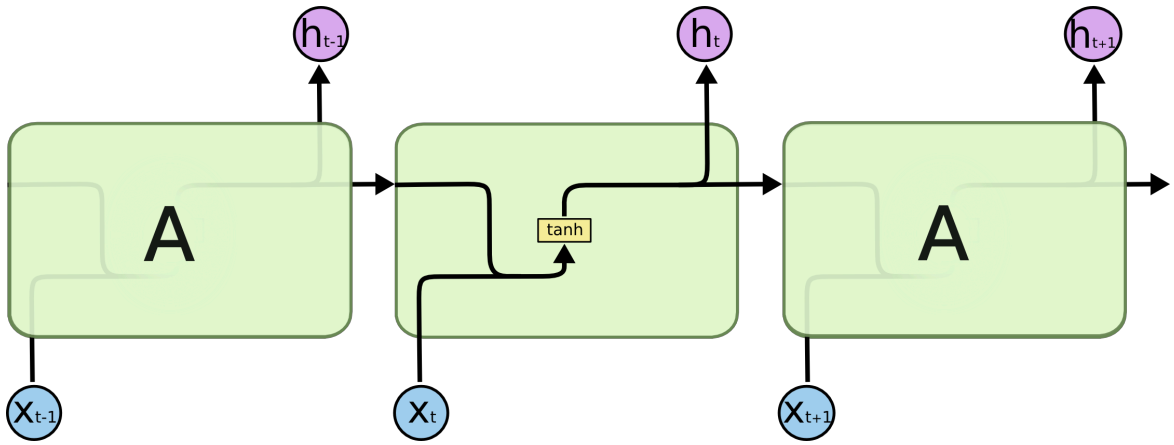
- RNNs accept an input vector x and give you an output vector y_hat.
- Training is done by minimising loss L = L(1) + L(2) + … + L(T) by backprop.
  (need to cut off at length T, typically ~ 50)
- Crucially this output is influenced by both the input you just fed in, and the entire history of inputs you've fed in in the past
- One benefit of RNNs: generalises naturally to new lengths not seen during training, and input/output can have arbitrary length:



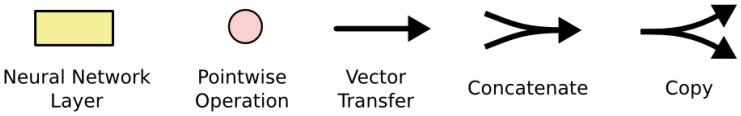
1. Image classification (image -> label)
2. Image captioning (image -> sentence)
3. Sentiment analysis (sentence -> sentiment)
4. Machine translation (sentence -> sentence)
5. Video classification (frame -> label)

- However RNNs struggle to learn long-term dependencies with gradient descent, due to vanishing gradients. (RNN is similar to a deep NN with shared weights)

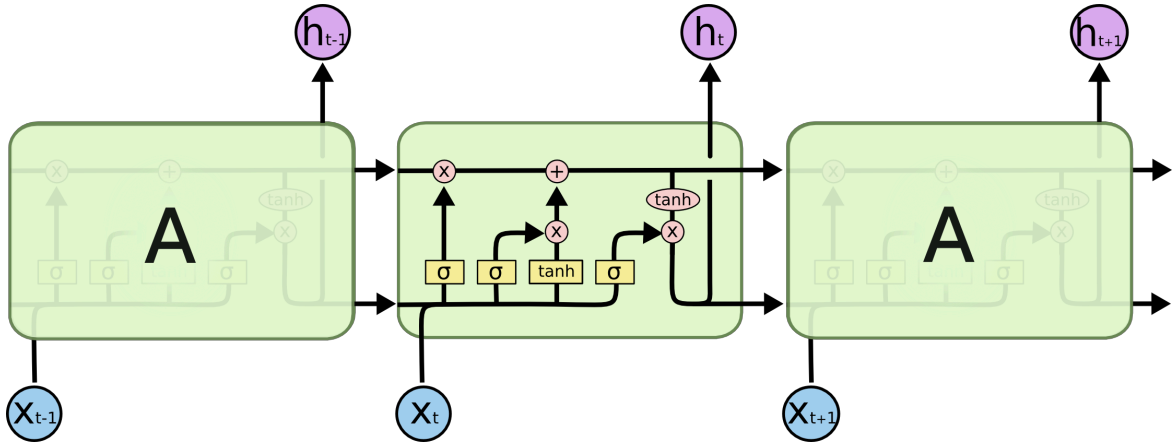
- Long Short Term Memory Networks (LSTM) are capable of learning such long-term dependencies
- Vanilla RNN



Key:

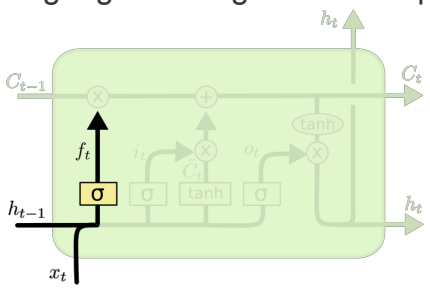


- LSTM



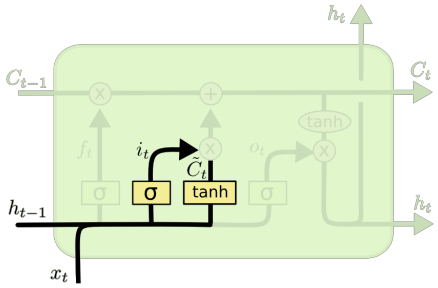
  - The key to LSTMs is the cell state C_t (horizontal line running through top of diagram)
    - Like a conveyer belt - runs down entire chain, with only minor modifications
    - So it's easy for info to flow unchanged
    - Acts as 'memory' to store info from long ago and hence learn long-term dependencies
  - LSTM has ability to remove or add info to the cell state, regulated by structures called gates
  - These gates are composed of a sigmoid NN layer and a pointwise multiplication
  - Sigmoid layer outputs numbers in [0,1], describing how much of each component should be let through

- There are three gates:
  1. Forget gate – forget info from previous cell state



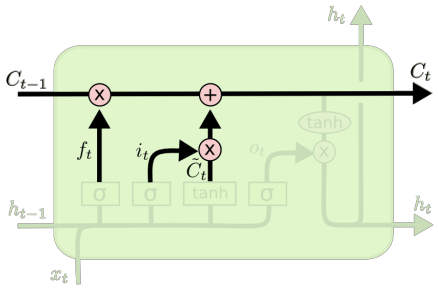
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

     - f_t is a vector with entries in [0,1], determining which bits of the cell state one should remember
     - h_t is the hidden state (which confusingly also happens to be the output from previous step), different to the cell state
  2. Input gate – add new info to current cell state



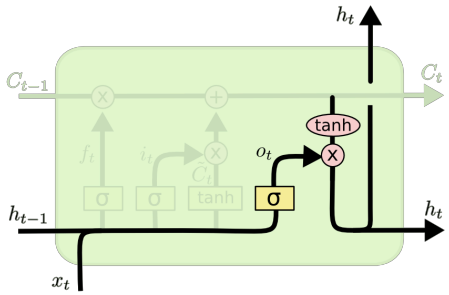
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

     - C_tilde is the new info which we want to add to the cell state
     - i_t determines which bits of this new info we should add
     - So forgetting & input done by:



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

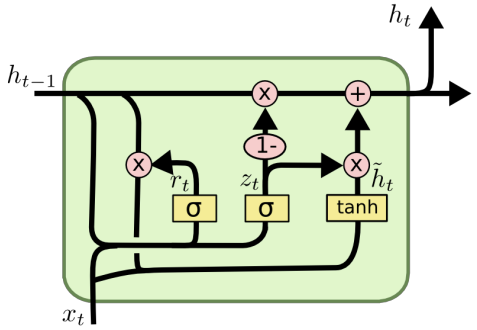
  3. Output gate – decide what to output as hidden state (output) h_t



$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

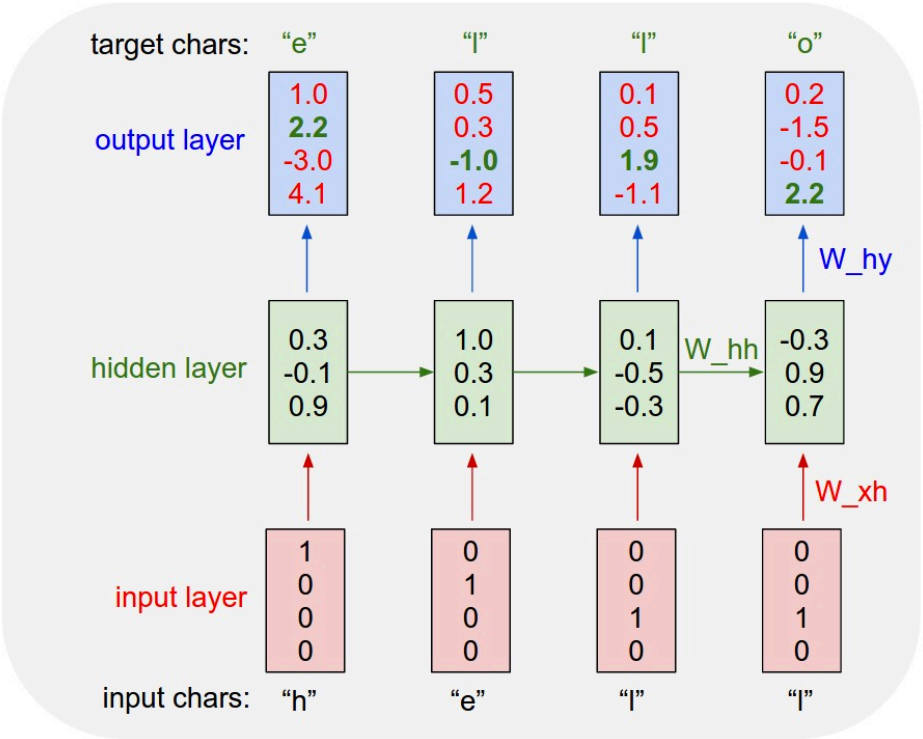
     - First run a sigmoid layer o_t to decide which bits of C_t we would like to output.
     - Then multiply o_t by tanh(C_t) to form output h_t

- All this may seem needlessly complicated. A simpler version of LSTM is the Gated Recurrent Unit (GRU)
- GRU combines the forget & input gate into a single update gate, and also merges the cell state and hidden state



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

- There are lots of other variants of LSTMs, but only LSTM and GRU seems to be notable. Others are either more complicated and show similar performance, or do not show better performance.
- Application of LSTMs: Character-level language model
  - Give LSTM a big chunk of text and ask it to model the probability distribution of the next character in the sequence given a sequence of previous characters. This will then allow us to generate new text one character at a time



  - Want to increase confidence in observed letters (green) and decrease the confidence of all other letters (red).
  - Most common approach is to use a cross-entropy loss function, which corresponds to maximum likelihood with softmax classifier on every output vector, with the correct class being the index of the next character in the sequence

References (for text & figures):

- http://karpathy.github.io/2015/05/21/rnn-effectiveness/
- http://www.deeplearningbook.org/contents/rnn.html
- http://colah.github.io/posts/2015-08-Understanding-LSTMs/