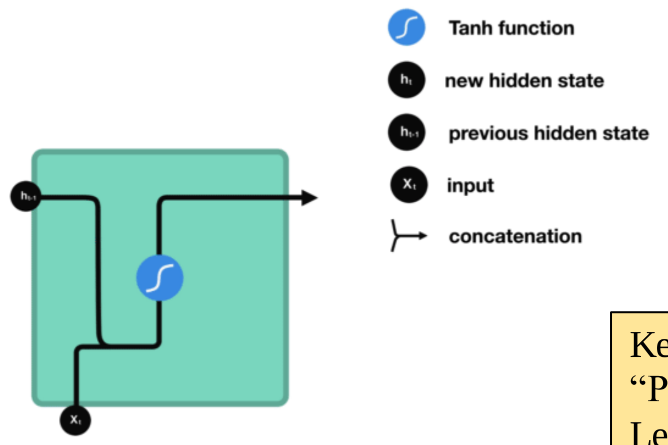
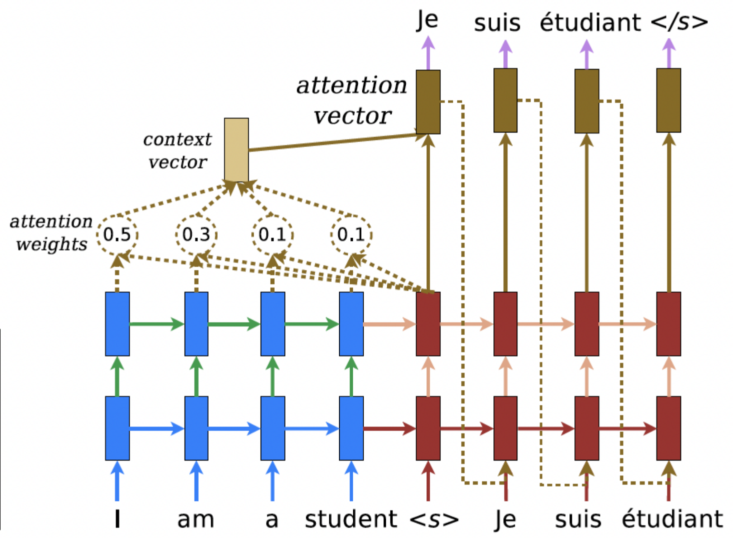
Sequence Learning

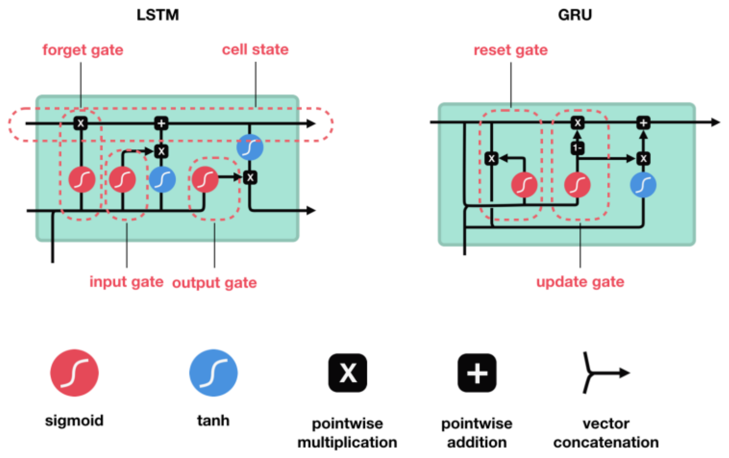
Souvik Chakraborty
Indian Institute of Technology Delhi
E-mail: souvik@am.iitd.ac.in
Website: https://www.csccm.in/

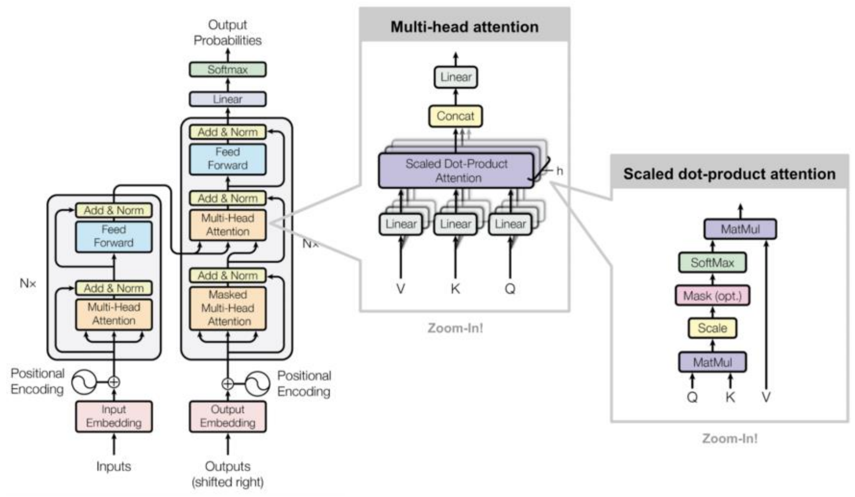
Overview



Kevin Murphy, "Probabilistic Machine Learning: An Introduction", 2022







Recurrent neural networks

- Recurrent neural networks, or RNNs, are a family of neural networks for processing sequential data.

- Much as a convolutional network is a neural network that is specialized for processing a grid of values X such as an image, a recurrent neural network is a neural network that is specialized for processing a sequence of values $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}$

- Consider the classical form of a dynamical system:

$$\boxed{\boldsymbol{s}^{(t)} = f\left(\boldsymbol{s}^{(t-1)}; \boldsymbol{\theta}\right)}$$

- where $\boldsymbol{s}^{(t)}$ is called the state of the system. Above equation is recurrent because the definition of s at time t refers back to the same definition at time $t-1$

Recurrent neural networks

$$\boxed{\boldsymbol{s}^{(t)} = f\left(\boldsymbol{s}^{(t-1)};\boldsymbol{\theta}\right)}$$

- For a finite number of time steps τ, the graph can be unfolded by applying the definition $\tau-1$ times. For example, if we unfold above equation for $\tau=3$ time steps, we obtain

$$\begin{aligned}\boldsymbol{s}^{(3)} &= f\left(\boldsymbol{s}^{(2)};\boldsymbol{\theta}\right)\\ &= f\left(f\left(\boldsymbol{s}^{(1)};\boldsymbol{\theta}\right);\boldsymbol{\theta}\right)\end{aligned}$$

- Unfolding the equation by repeatedly applying the definition in this way has yielded an expression that does not involve recurrence.
- Such an expression can now be represented by a traditional directed acyclic computational graph.

Recurrent neural networks

- Let us consider a dynamical system driven by an external signal $\boldsymbol{x}^{(t)}$

$$\boldsymbol{s}^{(t)} = f\left(\boldsymbol{s}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}\right)$$

where we see that the state now contains information about the whole past sequence.

- Recurrent neural networks can be built in many different ways.
- Much as almost any function can be considered a feedforward neural network, essentially any function involving recurrence can be considered a recurrent neural network.

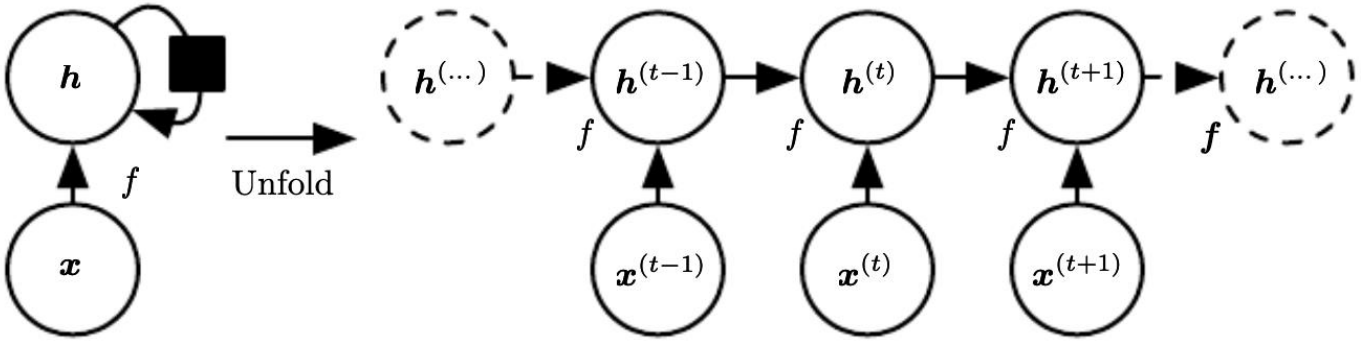
Recurrent neural networks

$$\boxed{\boldsymbol{s}^{(t)} = f\left(\boldsymbol{s}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}\right)}$$

- Many recurrent neural networks use the below equation to define the values of their hidden units. To indicate that the state is the hidden units of the network, we now rewrite the above equation using the variable $\boldsymbol{h}$ to represent the state,

$$\boxed{\boldsymbol{h}^{(t)} = f\left(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}\right)}$$

- Typical RNNs will add extra architectural features such as output layers that read information out of the state $\boldsymbol{h}$ to make predictions.



A recurrent network with no outputs.

Recurrent neural networks

- When the recurrent network is trained to perform a task that requires predicting the future from the past, the network typically learns to use $\boldsymbol{h}^{(t)}$ as a kind of lossy summary of the task-relevant aspects of the past sequence of inputs up to t.

- This summary is in general necessarily lossy, since it maps an arbitrary length sequence $\left(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \dots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}\right)$ to a fixed length vector $\boldsymbol{h}^{(t)}$.

- Depending on the training criterion, this summary might selectively keep some aspects of the past sequence with more precision than other aspects.

- For example, if the RNN is used in statistical language modeling, typically to predict the next word given previous words, storing all the information in the input sequence up to time t may not be necessary; storing only enough information to predict the rest of the sentence is sufficient.

Recurrent neural networks

Some examples of important design patterns for recurrent neural networks include the following:

- Recurrent networks that produce an output at each time step and have recurrent connections between hidden units

- Recurrent networks that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step

- Recurrent networks with recurrent connections between hidden units, that read an entire sequence and then produce a single output

Recurrent neural networks

- We now develop the forward propagation equations for the RNN.
- Here, we assume that the output is discrete, as if the RNN is used to predict words or characters.
- A natural way to represent discrete variables is to regard the output $\boldsymbol{O}$ as giving the unnormalized log probabilities of each possible value of the discrete variable.
- We can then apply the softmax operation as a post-processing step to obtain a vector $\hat{\boldsymbol{y}}$ of normalized probabilities over the output.

Recurrent neural networks

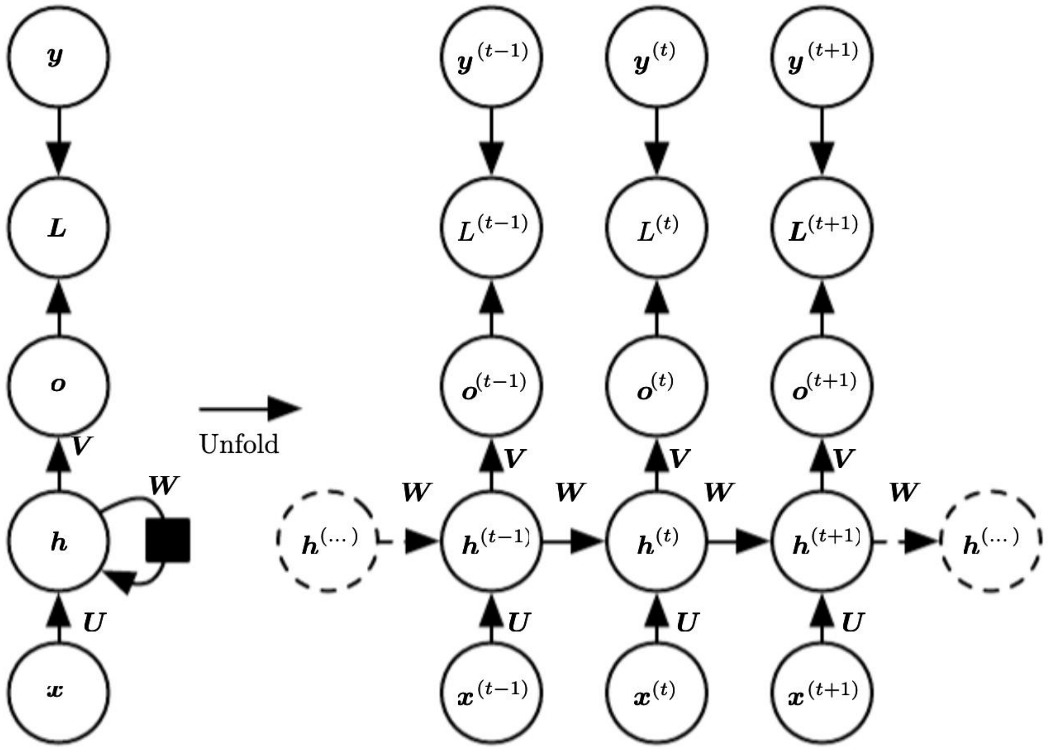
- Forward propagation begins with a specification of the initial state $\boldsymbol{h}^{(0)}$. Then, for each time step from $t = 1$ to $t = \tau$, we apply the following update equations:

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

where the parameters are the bias vectors $\boldsymbol{b}$ and $\boldsymbol{c}$ along with the weight matrices $\boldsymbol{U}, \boldsymbol{V}$ and $\boldsymbol{W}$, respectively, for input-to-hidden, hidden-to-output and hidden to-hidden connections.

- This is an example of a recurrent network that maps an input sequence to an output sequence of the same length.

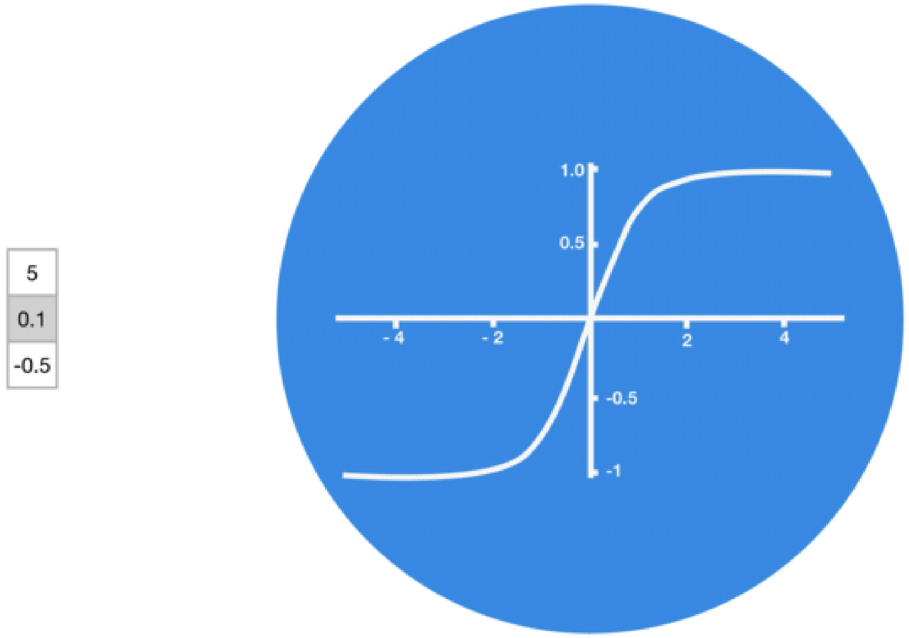
Recurrent neural networks



The computational graph to compute the training loss of a recurrent network that maps an input sequence of $\boldsymbol{x}$ *values to a corresponding sequence of output* $\boldsymbol{o}$ *values. A loss L measures how far each* $\boldsymbol{o}$ *is from the corresponding training target* $\boldsymbol{y}$*. When using softmax outputs, we assume o is the unnormalized log probabilities. The loss L internally computes* $\hat{\boldsymbol{y}} = softmax(\boldsymbol{o})$ *and compares this to the target* $\boldsymbol{y}$*. The RNN has input to hidden connections parametrized by a weight matrix* $\boldsymbol{U}$*, hidden-to-hidden recurrent connections parametrized by a weight matrix* $\boldsymbol{W}$*, and hidden-to-output connections parametrized by a weight matrix* $\boldsymbol{V}$*.*

Recurrent neural networks: Tanh activation function

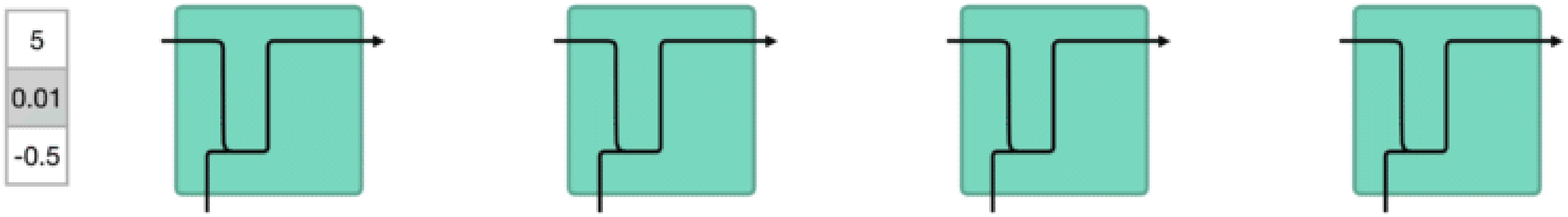
- The tanh activation is used to help regulate the values flowing through the network. The tanh function squishes values to always be between -1 and 1.



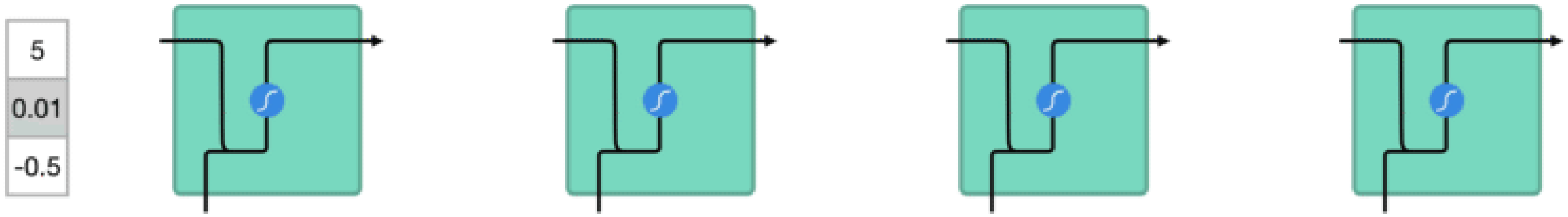
Animation taken from:
https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

Recurrent neural networks: tanh activation function

- When vectors are flowing through a neural network, it undergoes many transformations due to various math operations. So imagine a value that continues to be multiplied by let's say 3. One can see how some values can explode and high, causing other values to seem insignificant.



- A tanh function ensures that the values stay between -1 and 1, thus regulating the output of the neural network. One can see how the same values from above remain between the boundaries allowed by the tanh function



Animation taken from:
https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

Recurrent neural networks

- The total loss for a given sequence of $\boldsymbol{x}$ values paired with a sequence of $\boldsymbol{y}$ values would then be just the sum of the losses over all the time steps. For example, if $L^{(t)}$ is the negative log-likelihood of $y^{(t)}$ given $\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)}$, then

  $$L\left(\left\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(\tau)}\right\}, \left\{\boldsymbol{y}^{(1)}, \dots, \boldsymbol{y}^{(\tau)}\right\}\right) = \sum_t L^{(t)}$$

  $$L\left(\left\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(\tau)}\right\}, \left\{\boldsymbol{y}^{(1)}, \dots, \boldsymbol{y}^{(\tau)}\right\}\right) = -\sum_t \log p_{\text{model}}\left(y^{(t)} \mid \left\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)}\right\}\right)$$

  where $p_{\text{model}}\left(y^{(t)} \mid \left\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)}\right\}\right)$ is given by reading the entry for $y^{(t)}$ from the model's output vector $\hat{\boldsymbol{y}}^{(t)}$.

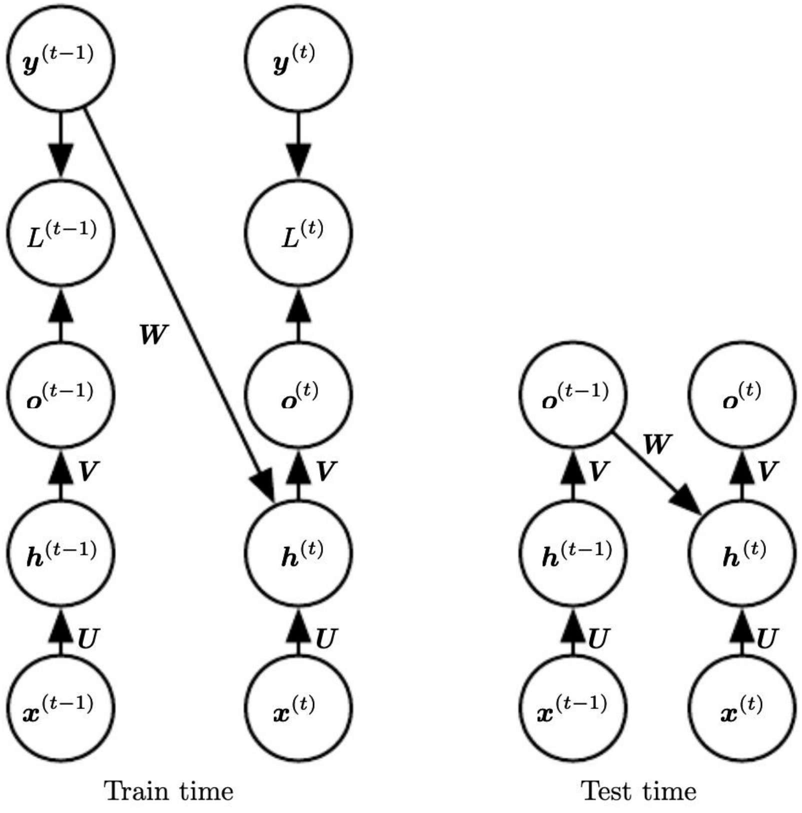
- Computing the gradient of this loss function with respect to the parameters is an expensive operation. The gradient computation involves performing a forward propagation pass moving left to right, followed by a backward propagation pass moving right to left through the graph.

Teacher Forcing and Networks with Output Recurrence

- The network with recurrent connections only from the output at one time step to the hidden units at the next time step is strictly less powerful because it lacks hidden-to-hidden recurrent connections.

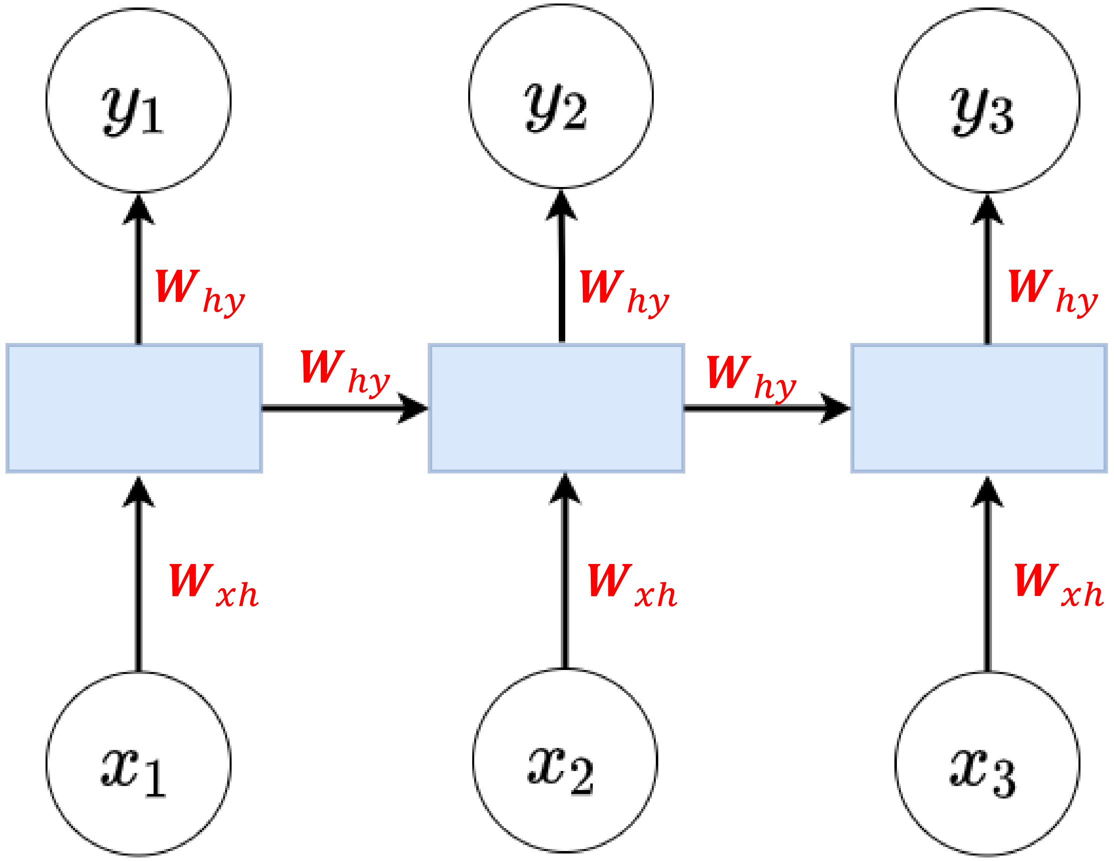
- Because this network lacks hidden-to-hidden recurrence, it requires that the output units capture all the information about the past that the network will use to predict the future.

- Because the output units are explicitly trained to match the training set targets, they are unlikely to capture the necessary information about the past history of the input, unless the user knows how to describe the full state of the system and provides it as part of the training set targets.



BPTT

- For training such networks, we use good old backpropagation but with a slight twist.
  - We don't independently train the system at a specific time "t". We train it at a specific time "t" as well as all that has happened before time "t" like t-1, t-2, t-3.



$$\boxed{\begin{aligned} h_t &= \sigma_1\left(W_{xh}^T x_t + W_{hh}^T h_{t-1}\right) \\ y_t &= \sigma_2\left(W_{hy}^T h_t\right) \end{aligned}}$$

BPTT

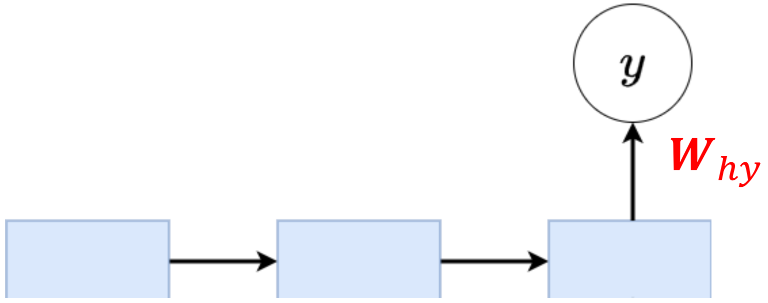
- Let us now attempt to backpropagate at $t = 3$

$$E_3 = (t_3 - y_3)^2$$

- To perform back propagation, we have to adjust the weights associated with inputs, the memory units and the outputs

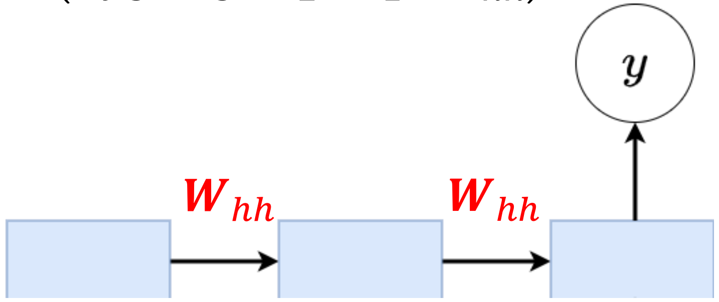
- **Adjusting $\boldsymbol{W_{hy}}$:**

$$\frac{\partial E_3}{\partial W_{hy}} = \frac{\partial E_3}{\partial y_3}\frac{\partial y_3}{\partial W_{hy}}$$



- **Adjusting $\boldsymbol{W_{hh}}$:**

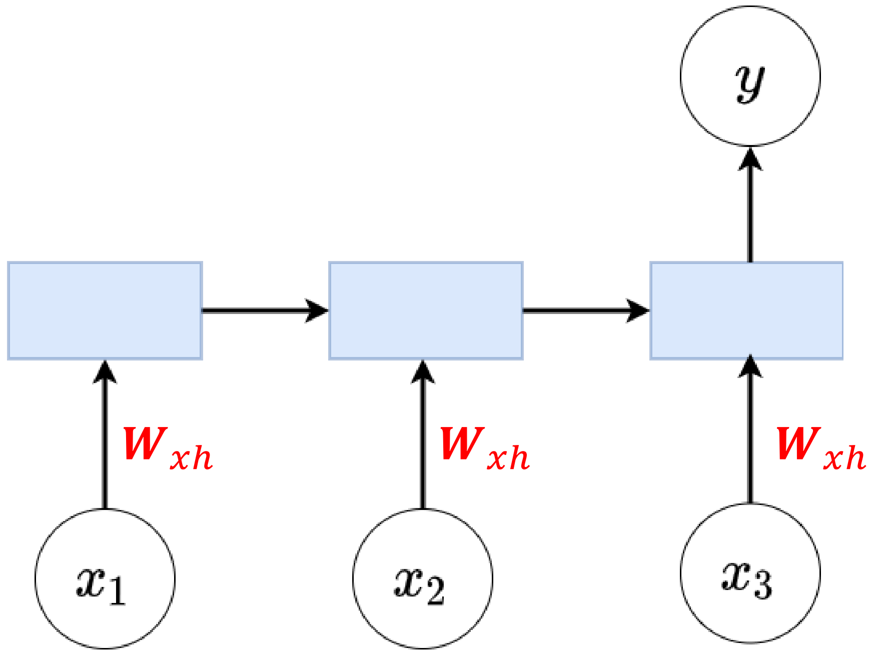
$$\frac{\partial E_3}{\partial W_{hh}} = \left(\frac{\partial E_3}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial W_{hh}}\right) + \left(\frac{\partial E_3}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial W_{hh}}\right) + \left(\frac{\partial E_3}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial W_{hh}}\right)$$



BPTT

- **Adjusting W_{xh}:**

$$\frac{\partial E_3}{\partial W_{xh}} = \left(\frac{\partial E_3}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial W_{xh}}\right) + \left(\frac{\partial E_3}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial W_{xh}}\right) + \left(\frac{\partial E_3}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial W_{xh}}\right)$$



BPTT

Limitations:

- This method of Back Propagation through time (BPTT) can be used up to a limited number of time steps like 8 or 10
- If we back propagate further, the gradient δ becomes too small. This problem is called the "Vanishing gradient" problem
- The problem is that the contribution of information decays geometrically over time. So, if the number of time steps is >10 (Let's say), that information will effectively be discarded.

One of the famous solutions to this problem is by using what is called Long Short-Term Memory (LSTM for short) cells instead of the traditional RNN cells.

But there might arise yet another problem here, called the exploding gradient problem, where the gradient grows uncontrollably large.

BPTT

Limitations:

- This method of Back Propagation through time (BPTT) can be used up to a limited number of time steps like 8 or 10
- If we back propagate further, the gradient δ becomes too small. This problem is called the “Vanishing gradient” problem
- The problem is that the contribution of information decays geometrically over time. So, if the number of time steps is >10 (Let’s say), that information will effectively be discarded.

Computing the Gradient in a Recurrent Neural Network

- The nodes of our computational graph include the parameters $\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W}, \boldsymbol{b}$ and $\boldsymbol{c}$ as well as the sequence of nodes indexed by t for $\boldsymbol{x}^{(t)}, \boldsymbol{h}^{(t)}, \boldsymbol{o}^{(t)}$ and $L^{(t)}$.

- For each node $\mathbf{N}$ we need to compute the gradient $\nabla_{\mathrm{N}} L$ recursively, based on the gradient computed at nodes that follow it in the graph. We start the recursion with the nodes immediately preceding the final loss:

$$\frac{\partial L}{\partial L^{(t)}} = 1$$

- In this derivation we assume that the outputs $\boldsymbol{o}^{(t)}$ are used as the argument to the softmax function to obtain the vector $\hat{\boldsymbol{y}}$ of probabilities over the output. We also assume that the loss is the negative log-likelihood of the true target $y^{(t)}$ given the input so far. The gradient $\nabla_{\boldsymbol{o}^{(t)}} L$ on the outputs at time step t, for all i, t, is as follows:

$$\left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - 1_{i=y^{(t)}}$$

Computing the Gradient in a Recurrent Neural Network

- We work our way backward, starting from the end of the sequence. At the final time step $\tau, \boldsymbol{h}^{(\tau)}$ only has $\boldsymbol{o}^{(\tau)}$ as a descendent, so its gradient is simple:

$$\nabla_{\boldsymbol{h}^{(\tau)}} L = \boldsymbol{V}^{\top} \nabla_{\boldsymbol{o}^{(\tau)}} L$$

- We can then iterate backward in time to back-propagate gradients through time, from $t = \tau - 1$ down to $t = 1$, noting that $\boldsymbol{h}^{(t)}$(for $t < \tau$) has as descendants both $\boldsymbol{o}^{(t)}$ and $\boldsymbol{h}^{(t+1)}$. Its gradient is thus given by

$$\begin{aligned}\nabla_{\boldsymbol{h}^{(t)}} L &= \left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}}\right)^{\top} \left(\nabla_{\boldsymbol{h}^{(t+1)}} L\right) + \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}}\right)^{\top} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \\ &= \boldsymbol{W}^{\top} \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t+1)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t+1)}} L\right) + \boldsymbol{V}^{\top} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)\end{aligned}$$

where $\operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t+1)}\right)^2\right)$ indicates the diagonal matrix containing the elements $1 - \left(h_i^{(t+1)}\right)^2$. This is the Jacobian of the hyperbolic tangent associated with the hidden unit i at time $t + 1$

Computing the Gradient in a Recurrent Neural Network

- The gradient on the remaining parameters is given by

$$\nabla_{\boldsymbol{c}} L = \sum_t \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{c}}\right)^{\top} \nabla_{\boldsymbol{o}^{(t)}} L = \sum_t \nabla_{\boldsymbol{o}^{(t)}} L$$

$$\nabla_{\boldsymbol{b}} L = \sum_t \left(\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{b}^{(t)}}\right)^{\top} \nabla_{\boldsymbol{h}^{(t)}} L = \sum_t diag\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \nabla_{\boldsymbol{h}^{(t)}} L,$$

$$\nabla_{\boldsymbol{V}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}}\right) \nabla_{\boldsymbol{V}^{(t)}} o_i^{(t)} = \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \boldsymbol{h}^{(t)^T}$$

$$\nabla_{\boldsymbol{W}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{\boldsymbol{W}^{(t)}} h_i^{(t)}$$

$$= \sum_t diag\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \boldsymbol{h}^{(t-\mathbf{1})^T}$$

$$\nabla_{\boldsymbol{U}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{\boldsymbol{U}^{(t)}} h_i^{(t)}$$

$$= \sum_t diag\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \boldsymbol{x}^{(t)^T}$$

Bidirectional RNNs

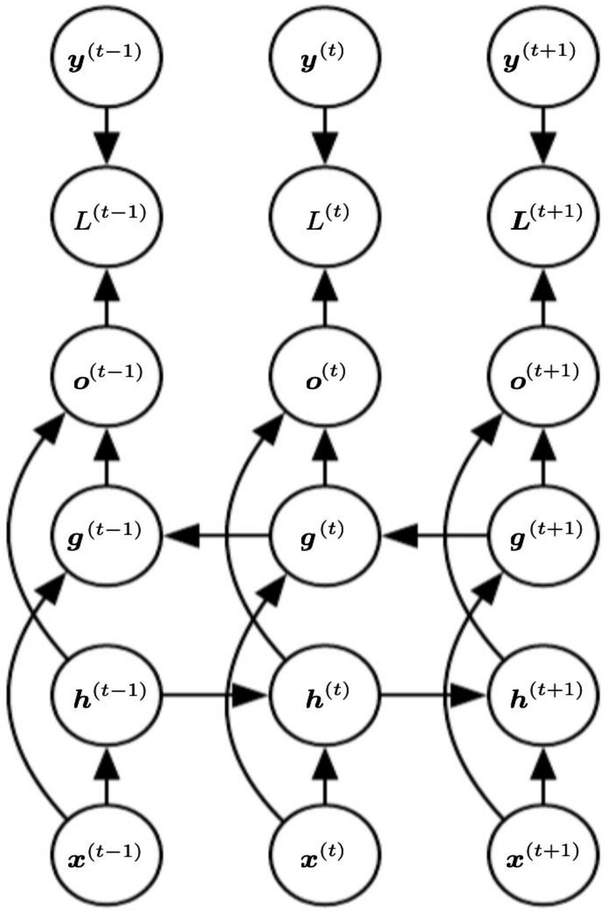
- In many applications, we want to output a prediction of $\boldsymbol{y}^{(t)}$ that may depend on the whole input sequence.

- For example, in speech recognition, the correct interpretation of the current sound as a phoneme may depend on the next few phonemes because of co-articulation and may even depend on the next few words because of the linguistic dependencies between nearby words: if there are two interpretations of the current word that are both acoustically plausible, we may have to look far into the future (and the past) to disambiguate them.

- This is also true of handwriting recognition and many other sequence-to-sequence learning tasks. Example
  - I am ___
  - I am ___ hungry
  - I am ___ hungry, and I can eat only salads.

- A. Zhang, Z. Lipton, M. Li, and A. Smola. Dive into deep learning. 2019.

Bidirectional RNNs



Computation of a typical bidirectional recurrent neural network, meant to learn to map input sequences $\boldsymbol{x}$ to target sequences $\boldsymbol{y}$, with loss $L^{(t)}$ at each step t The $\boldsymbol{h}$ recurrence propagates information forward in time (toward the right), while the $\boldsymbol{g}$ recurrence propagates information backward in time (toward the left). Thus at each point t, the output units $\boldsymbol{o}^{(t)}$ can benefit from a relevant summary of the past in its $\boldsymbol{h}^{(t)}$ input and from a relevant summary of the future in its $\boldsymbol{g}^{(t)}$ input.

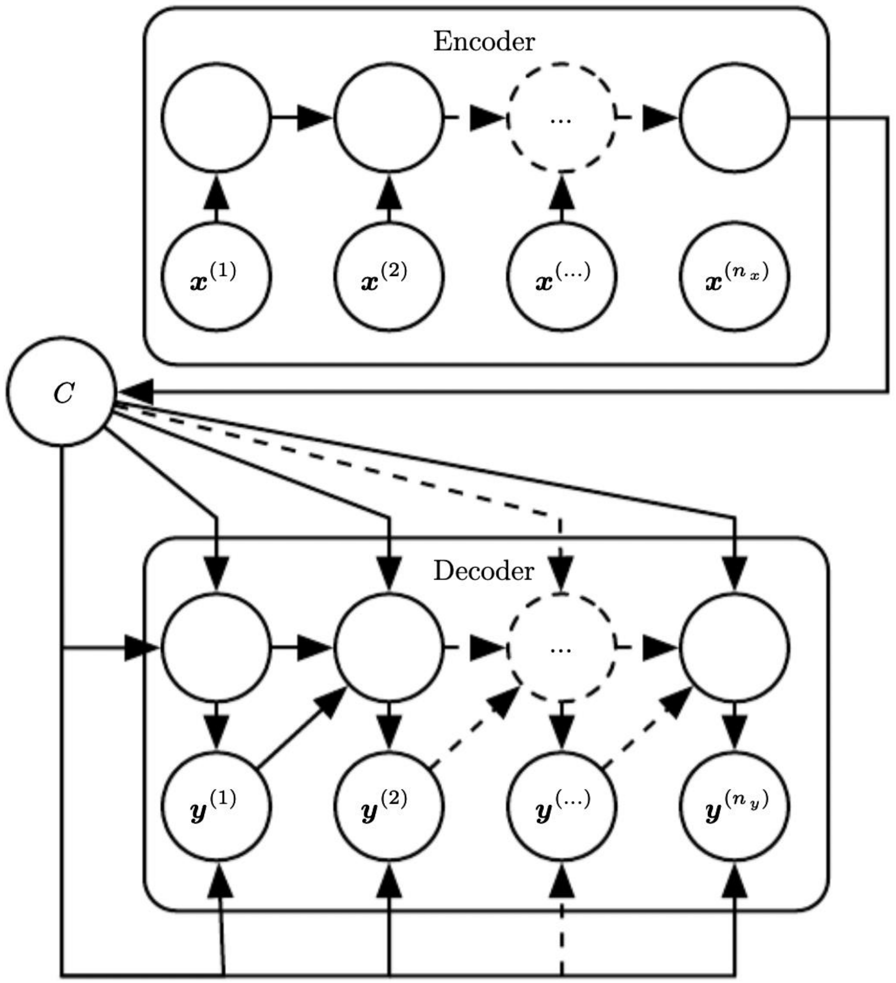
Encoder-Decoder Sequence-to-Sequence Architectures

- Now we discuss how an RNN can be trained to map an input sequence to an output sequence which is not necessarily of the same length. This comes up in many applications, such as speech recognition, machine translation and question answering, where the input and output sequences in the training set are generally not of the same length (although their lengths might be related).

- We often call the input to the RNN the "context." We want to produce a representation of this context, C. The context C might be a vector or sequence of vectors that summarize the input sequence $\boldsymbol{X} = \left(\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(n_x)}\right)$.

- The idea is very simple:
  - (1) An encoder or reader or input RNN processes the input sequence. The encoder emits the context C, usually as a simple function of its final hidden state.
  - (2) A decoder or writer or output RNN is conditioned on that fixed-length to generate the output sequence $\boldsymbol{Y} = \left(\boldsymbol{y}^{(1)}, \dots, \boldsymbol{y}^{(n_y)}\right)$.

Encoder-Decoder Sequence-to-Sequence Architectures

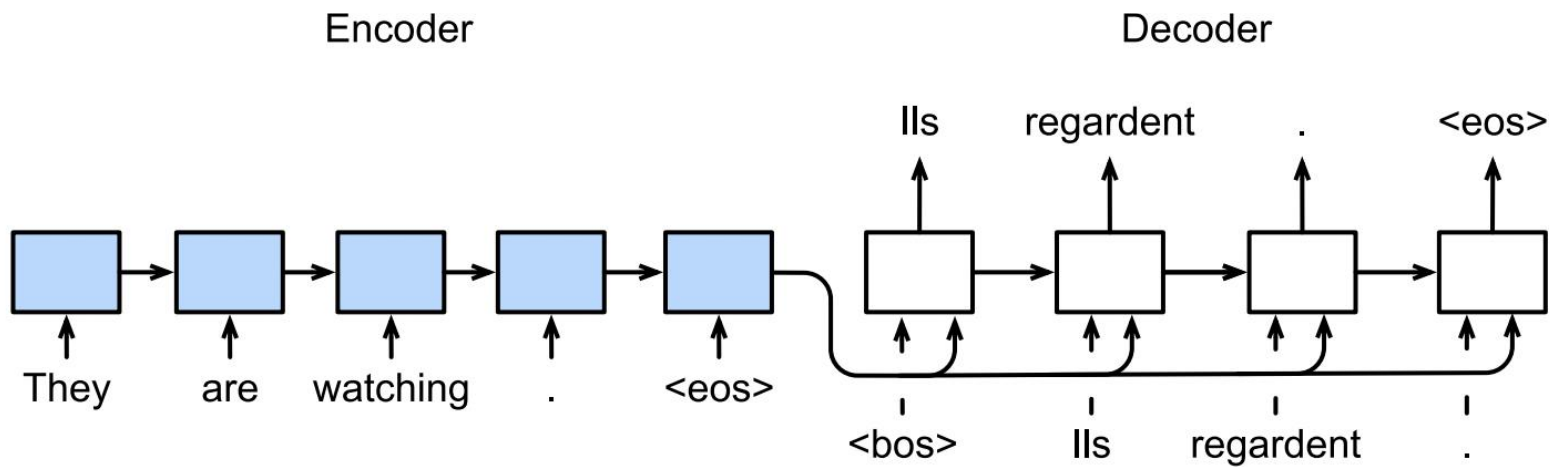


- Example of an encoder-decoder or sequence-to-sequence RNN architecture, for learning to generate an output sequence $\left(\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(n_y)}\right)$ given an input sequence $\left(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(n_x)}\right)$.

- It is composed of an encoder RNN that reads the input sequence as well as a decoder RNN that generates the output sequence (or computes the probability of a given output sequence).

- The final hidden state of the encoder RNN is used to compute a generally fixed-size context variable C, which represents a semantic summary of the input sequence and is given as input to the decoder RNN.

Encoder-Decoder Sequence-to-Sequence Architectures



Sequence to sequence learning with an RNN encoder and an RNN decoder.
The special “<eos>” token marks the end of the sequence. The model can stop making predictions once this token is generated. At the initial time step of the RNN decoder, there are two special design decisions. First, the special beginning-of-sequence “<bos>” token is an input.

- A. Zhang, Z. Lipton, M. Li, and A. Smola. Dive into deep learning. 2019.

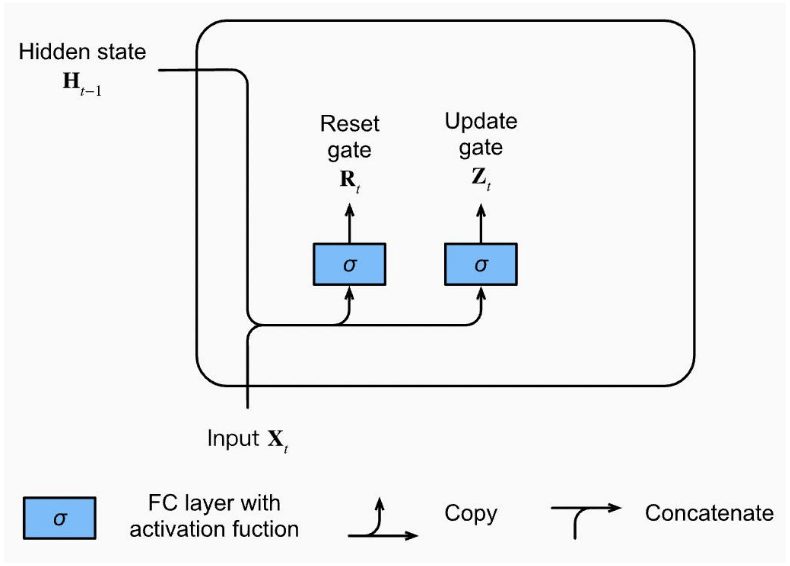
Gated Recurrent Units (GRU)

- The key distinction between vanilla RNNs and GRUs is that the latter support gating of the hidden state.
- This means that we have dedicated mechanisms for when a hidden state should be updated and also when it should be reset.
- These mechanisms are learned and they address the vanishing or exploding gradients concern commonly observed in RNNs.

- A. Zhang, Z. Lipton, M. Li, and A. Smola. Dive into deep learning. 2019.

GRU: Reset Gate and Update Gate

- A reset gate would allow us to control how much of the previous state we might still want to remember.
- Likewise, an update gate would allow us to control how much of the new state is just a copy of the old state.
- The outputs of two gates are given by two fully-connected layers with a sigmoid activation function.



GRU: Reset Gate and Update Gate

- Mathematically, for a given time step t, suppose that the input is a minibatch $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ (number of examples: n number of inputs: d) and the hidden state of the previous time step is $\mathbf{H}_{t-1} \in \mathbb{R}^{n\times h}$ (number of hidden units: h).

- Then, the reset gate $\mathbf{R}_t \in \mathbb{R}^{n\times h}$ and update gate $\mathbf{Z}_t \in \mathbb{R}^{n\times h}$ are computed as follows:

$$\mathbf{R}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xr} + \mathbf{H}_{t-1}\mathbf{W}_{hr} + \mathbf{b}_r)$$
$$\mathbf{Z}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xz} + \mathbf{H}_{t-1}\mathbf{W}_{hz} + \mathbf{b}_z)$$

where $\mathbf{W}_{xr}, \mathbf{W}_{xz} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{hr}, \mathbf{W}_{hz} \in \mathbb{R}^{h\times h}$ are weight parameters and $\mathbf{b}_r, \mathbf{b}_z \in \mathbb{R}^{1\times h}$ are biases. Note that broadcasting is triggered during the summation. We use sigmoid functions to transform input values to the interval (0,1) .

GRU: Candidate Hidden State

- Next, let us integrate the reset gate $\mathbf{R}_t$ with the regular latent state updating mechanism.

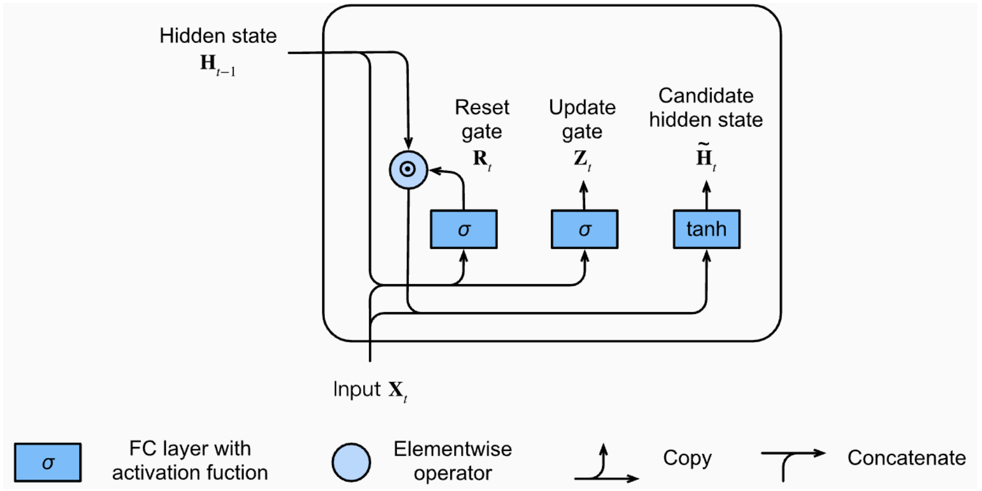
- It leads to the following candidate hidden state $\tilde{\mathbf{H}}_t \in \mathbb{R}^{n\times h}$ at time step t:

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1})\mathbf{W}_{hh} + \mathbf{b}_h)$$

where $\mathbf{W}_{xh} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{hh} \in \mathbb{R}^{h\times h}$ are weight parameters, $\mathbf{b}_h \in \mathbb{R}^{1\times h}$ is the bias, and the symbol $\odot$ is the Hadamard (elementwise) product operator. Here we use a nonlinearity in the form of tanh to ensure that the values in the candidate hidden state remain in the interval (-1,1) .

GRU: Candidate Hidden State

- The result is a candidate since we still need to incorporate the action of the update gate.
- Now the influence of the previous states can be reduced with the elementwise multiplication of $\mathbf{R}_t$ and $\mathbf{H}_{t-1}$.
- Whenever the entries in the reset gate $\mathbf{R}_t$ are close to 1, we recover a vanilla RNN.
- For all entries of the reset gate $\mathbf{R}_t$ that are close to 0, the candidate hidden state is the result of an MLP with $\mathbf{X}_t$ as the input. Any preexisting hidden state is thus reset to defaults.


Hidden state $\mathbf{H}_{t-1}$
Reset gate $\mathbf{R}_t$
Update gate $\mathbf{Z}_t$
Candidate hidden state $\tilde{\mathbf{H}}_t$
⊙
σ
σ
tanh
Input $\mathbf{X}_t$
σ
FC layer with activation fuction
Elementwise operator
Copy
Concatenate

GRU: Hidden State

- Finally, we need to incorporate the effect of the update gate $\mathbf{Z}_t$.
- This determines the extent to which the new hidden state $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ is just the old state $\mathbf{H}_{t-1}$ and by how much the new candidate state $\tilde{\mathbf{H}}_t$ is used.
- This leads to the final update equation for the GRU:

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t$$

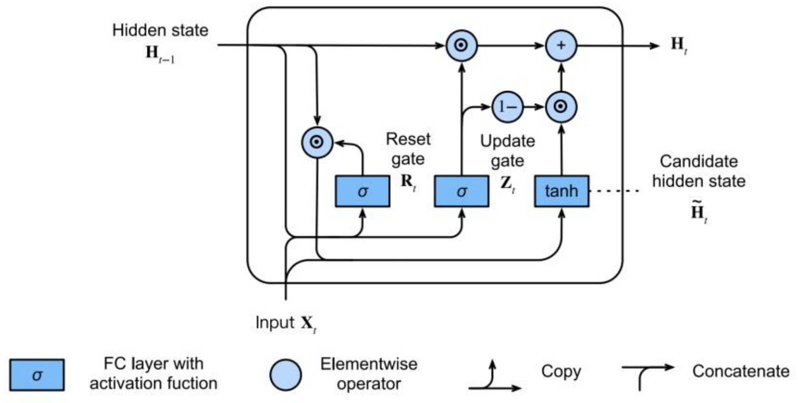
GRU

- Whenever the update gate $\mathbf{Z}_t$ is close to 1 , we simply retain the old state.

- In this case the information from $\mathbf{X}_t$ is essentially ignored, effectively skipping time step t in the dependency chain. In contrast, whenever $\mathbf{Z}_t$ is close to 0, the new latent state $\mathbf{H}_t$ approaches the candidate latent state $\tilde{\mathbf{H}}_t$.

- These designs can help us cope with the vanishing gradient problem in RNNs and better capture dependencies for sequences with large time step distances.

- For instance, if the update gate has been close to 1 for all the time steps of an entire subsequence, the old hidden state at the time step of its beginning will be easily retained and passed to its end, regardless of the length of the subsequence.



Hidden state $\mathbf{H}_{t-1}$
$\mathbf{H}_t$
Reset gate $\mathbf{R}_t$
Update gate $\mathbf{Z}_t$
1−
σ
tanh
Candidate hidden state $\tilde{\mathbf{H}}_t$
Input $\mathbf{X}_t$
FC layer with activation fuction
Elementwise operator
Copy
Concatenate

Kevin Murphy, "Probabilistic Machine Learning: An Introduction"

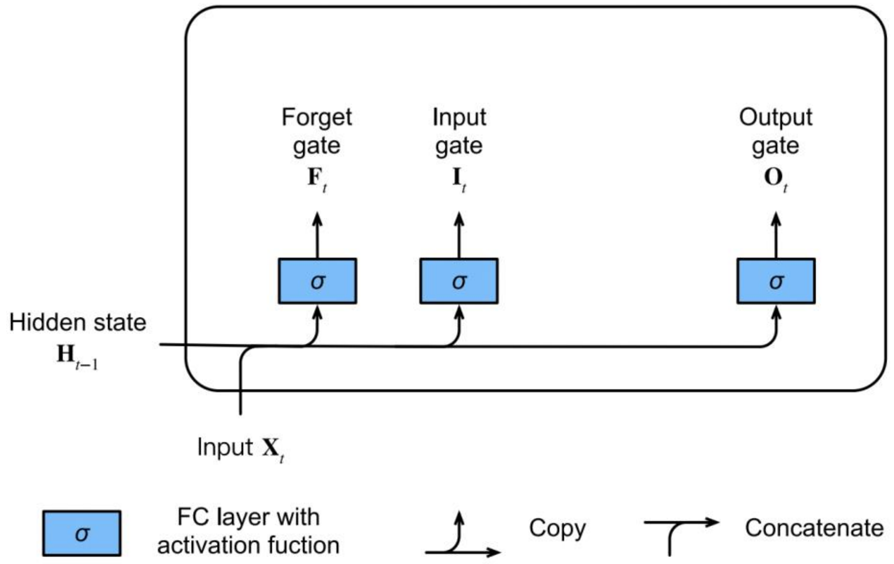
Long Short-Term Memory (LSTM)

- LSTM introduces a memory cell (or cell for short) that has the same shape as the hidden state (some literatures consider the memory cell as a special type of the hidden state), engineered to record additional information.
- To control the memory cell we need a number of gates.
- One gate is needed to read out the entries from the cell. We will refer to this as the output gate.
- A second gate is needed to decide when to read data into the cell. We refer to this as the input gate.
- Last, we need a mechanism to reset the content of the cell, governed by a forget gate

Long Short-Term Memory (LSTM)

- The data feeding into the LSTM gates are the input at the current time step and the hidden state of the previous time step.

- They are processed by three fully connected layers with a sigmoid activation function to compute the values of the input, forget. and output gates. As a result, values of the three gates are in the range of (0, 1).


Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Output gate $\mathbf{O}_t$
σ
σ
σ
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$
σ
FC layer with activation fuction
Copy
Concatenate

LSTM: Input Gate, Forget Gate, and Output Gate

- Mathematically, suppose that there are h hidden units, the batch size is n, and the number of inputs is d. Thus, the input is $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ and the hidden state of the previous time step is $\mathbf{H}_{t-1} \in \mathbb{R}^{n\times h}$. Correspondingly, the gates at time step t are defined as follows: the input gate is $\mathbf{I}_t \in \mathbb{R}^{n\times h}$, the forget gate is $\mathbf{F}_t \in \mathbb{R}^{n\times h}$, and the output gate is $\mathbf{O}_t \in \mathbb{R}^{n\times h}$. They are calculated as follows:

$$\begin{aligned}
\mathbf{I}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i) \\
\mathbf{F}_t &= \sigma\left(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f\right) \\
0_t &= \sigma(\mathbf{X}_t\mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o)
\end{aligned}$$

where $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h\times h}$ are weight parameters and $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1\times h}$ are bias parameters.

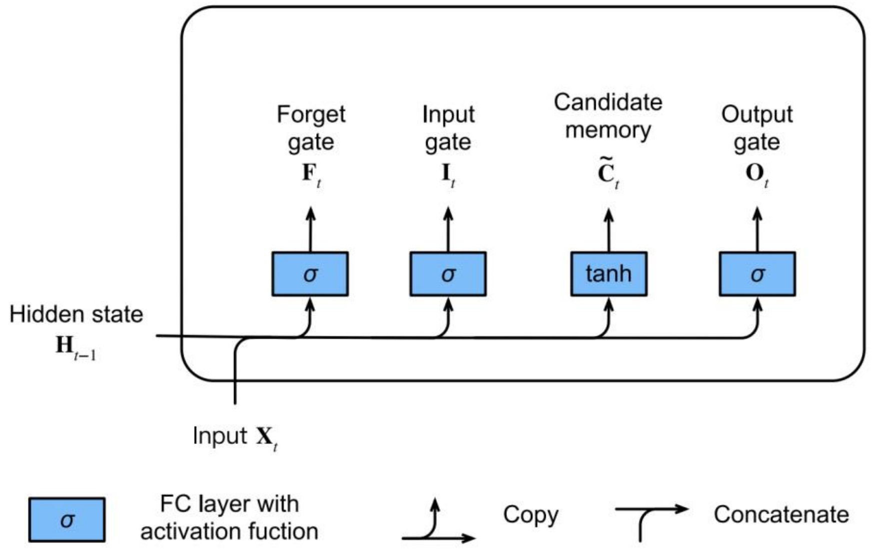
LSTM: Candidate Memory Cell

- Since we have not specified the action of the various gates yet, we first introduce the candidate memory cell $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n\times h}$.

- Its computation is similar to that of the three gates described above, but using a tanh function with a value range for (-1,1) as the activation function. This leads to the following equation at time step t :

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c)$$

where $\mathbf{W}_{xc} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{hc} \in \mathbb{R}^{h\times h}$ are weight parameters and $\mathbf{b}_c \in \mathbb{R}^{1\times h}$ is a bias parameter.



Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Candidate memory $\tilde{\mathbf{C}}_t$
Output gate $\mathbf{O}_t$
σ
σ
tanh
σ
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$
σ
FC layer with activation fuction
Copy
Concatenate

- A. Zhang, Z. Lipton, M. Li, and A. Smola. Dive into deep learning. 2019.

LSTM: Memory Cell

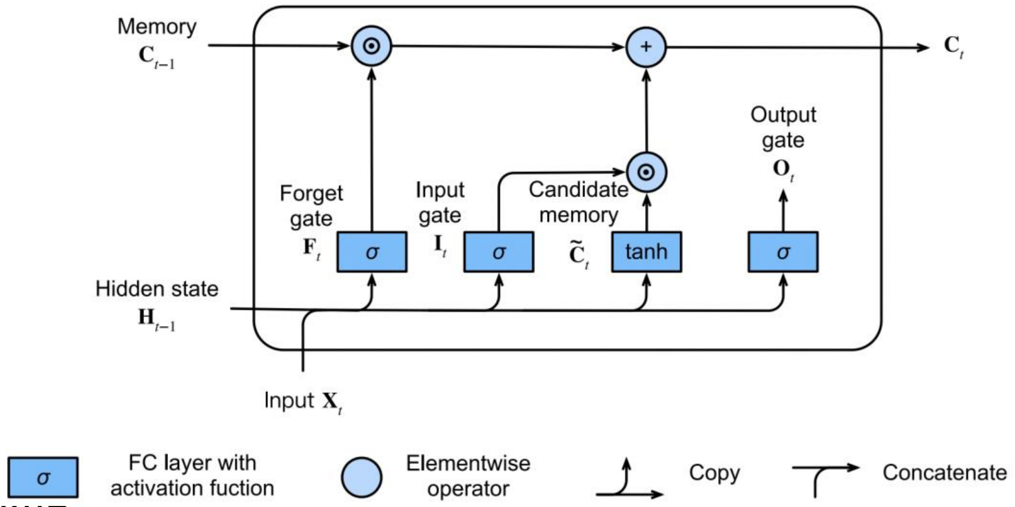
- In LSTMs, we have two dedicated gates for such purposes: the input gate $\mathbf{I}_t$ governs how much we take new data into account via $\tilde{\mathbf{C}}_t$ and the forget gate $\mathbf{F}_t$ addresses how much of the old memory cell content $\mathbf{C}_{t-1} \in \mathbb{R}^{n \times h}$ we retain. Using the same pointwise multiplication trick as before, we arrive at the following update equation:

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t$$

LSTM: Memory Cell

- If the forget gate is alw
approximately 1 and the input
is always approximately 0 ,
past memory cells $\mathbf{C}_{t-1}$ will
saved over time and passed to
current time step.

- This design is introduced
alleviate the vanishing grad
problem and to better capture l
range dependencies within
sequences.


Memory $\mathbf{C}_{t-1}$
$\mathbf{C}_t$
Output gate $\mathbf{O}_t$
Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Candidate memory $\tilde{\mathbf{C}}_t$
σ
σ
tanh
σ
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$
σ
FC layer with activation fuction
Elementwise operator
Copy
Concatenate

- A. Zhang, Z. Lipton, M. Li, and A. Smola. Dive into deep learning. 2019.

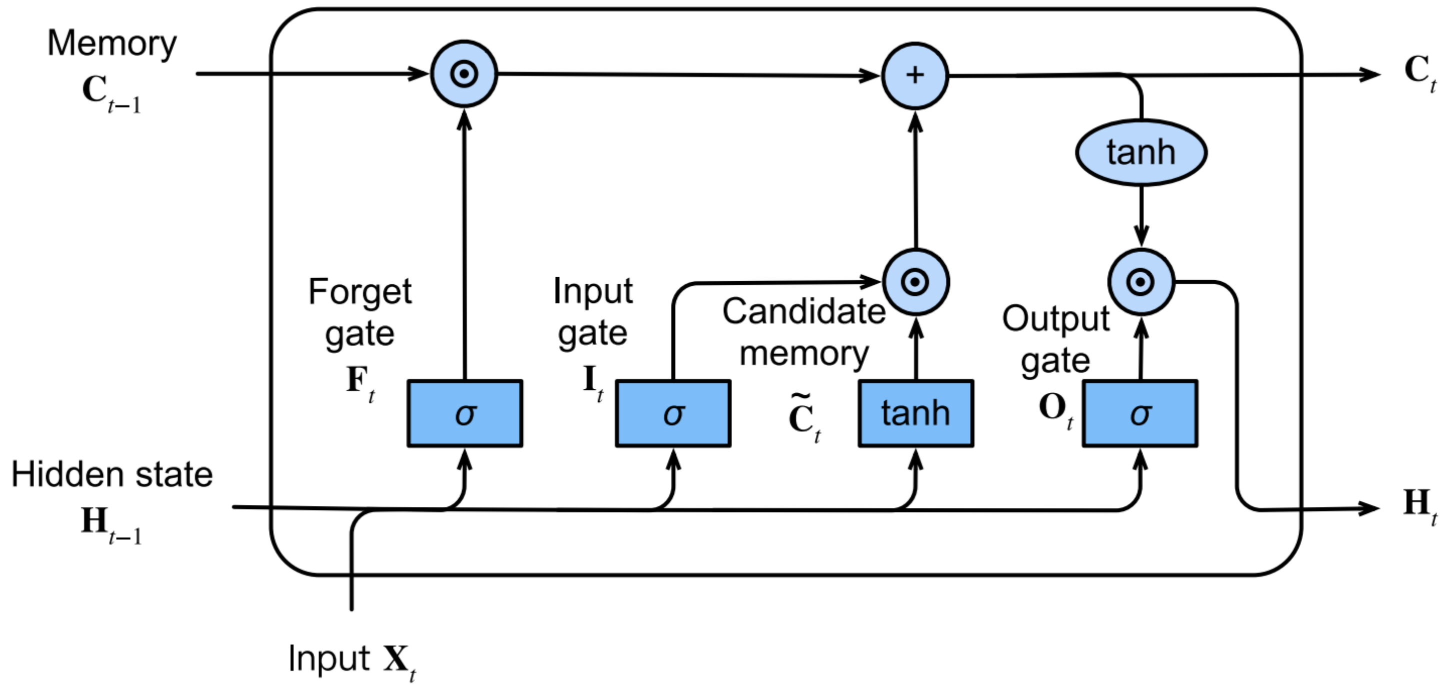
LSTM: Hidden State

- Last, we need to define how to compute the hidden state $\mathbf{H}_t \in \mathbb{R}^{n\times h}$. This is where the output gate comes into play. In LSTM it is simply a gated version of the tanh of the memory cell. This ensures that the values of $\mathbf{H}_t$ are always in the interval (-1,1) .

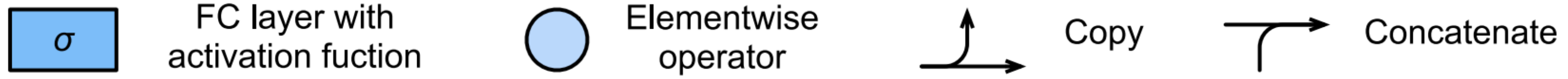
$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t)$$

- Whenever the output gate approximates 1 we effectively pass all memory information through to the predictor, whereas for the output gate close to 0 we retain all the information only within the memory cell and perform no further processing.

LSTM



Memory $\mathbf{C}_{t-1}$
$\mathbf{C}_t$
tanh
Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Candidate memory $\tilde{\mathbf{C}}_t$
Output gate $\mathbf{O}_t$
σ
tanh
Hidden state $\mathbf{H}_{t-1}$
$\mathbf{H}_t$
Input $\mathbf{X}_t$



σ
FC layer with activation fuction
Elementwise operator
Copy
Concatenate