

# **18AIC301J: DEEP LEARNING TECHNIQUES**

**B. Tech in ARTIFICIAL INTELLIGENCE, 5th semester**

Faculty: **Dr. Athira Nambiar**

Section: A, slot:D

Venue: TP 804

Academic Year: 2022-22

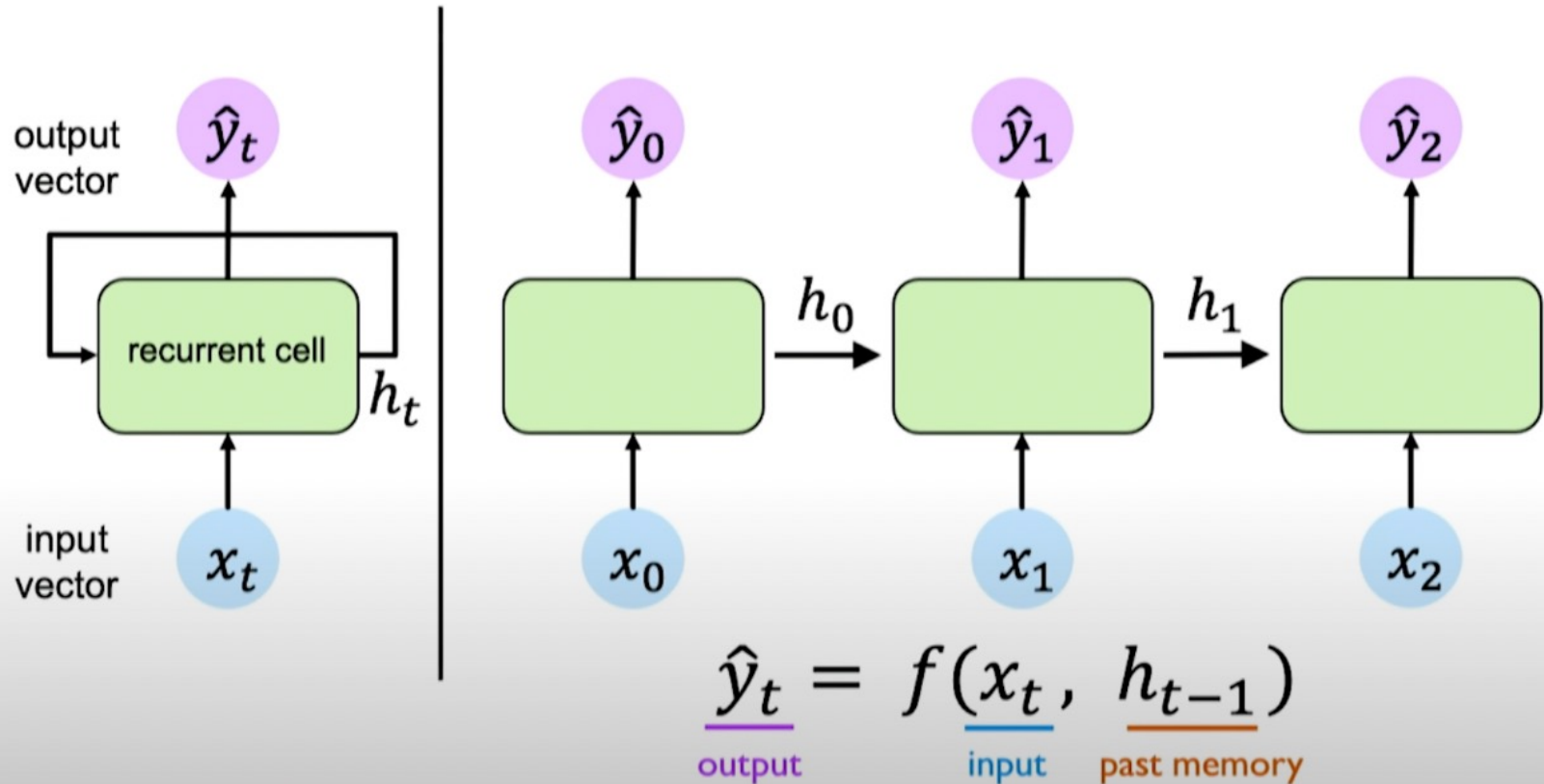
# UNIT-4

DenseNet Architecture, Transfer Learning
Need for Transfer Learning, Deep Transfer Learning, Types of Deep Transfer learning, Applications of Transfer learning
Transfer learning implementation using VGG16 model to classify images
Sequence Learning Problems, Recurrent Neural Networks
Backpropagation through time, Unfolded RNN, The problem of exploding and vanishing Gradients, Seq to Seq Models
Building a RNN to perform Character level language modeling.
How gates help to solve the problem of vanishing gradients, Long-Short Term Memory architectures
Dealing with exploding gradients, Gated Recurrent Units, Introduction to Encoder Decoder Models, Applications of Encoder Decoder Models
Build a LSTM network for Named Entity recognition.

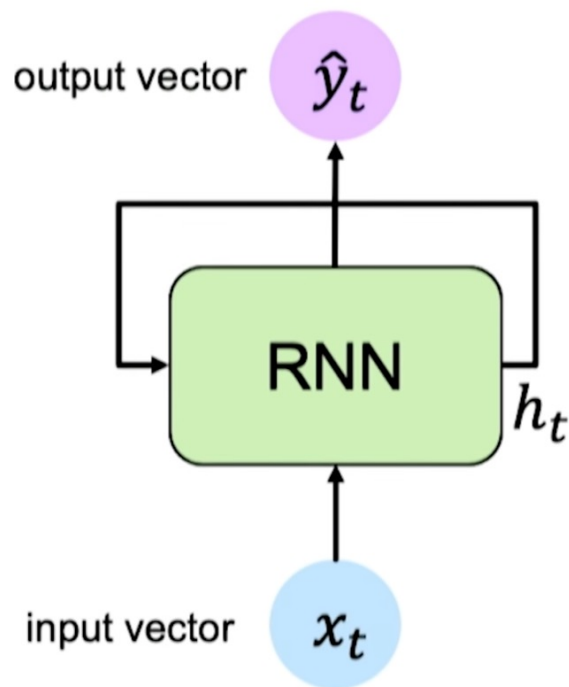
# UNIT-4

DenseNet Architecture, Transfer Learning
Need for Transfer Learning, Deep Transfer Learning, Types of Deep Transfer learning, Applications of Transfer learning
Transfer learning implementation using VGG16 model to classify images
Sequence Learning Problems, Recurrent Neural Networks
Backpropagation through time, Unfolded RNN, The problem of exploding and vanishing Gradients, Seq to Seq Models
Building a RNN to perform Character level language modeling.
How gates help to solve the problem of vanishing gradients, Long-Short Term Memory architectures
Dealing with exploding gradients, Gated Recurrent Units, Introduction to Encoder Decoder Models, Applications of Encoder Decoder Models
Build a LSTM network for Named Entity recognition.

# Neurons with Recurrence



# Recurrent Neural Networks (RNNs)



Apply a **recurrence relation** at every time step to process a sequence:

$$\boxed{h_t} = \boxed{f_W}(\boxed{x_t}, \boxed{h_{t-1}})$$

cell state      function with weights  $W$       input      old state

Note: the same function and set of parameters are used at every time step

RNNs have a **state**,  $h_t$ , that is updated **at each time step** as a sequence is processed

# Weight matrices in a single-hidden layer RNN

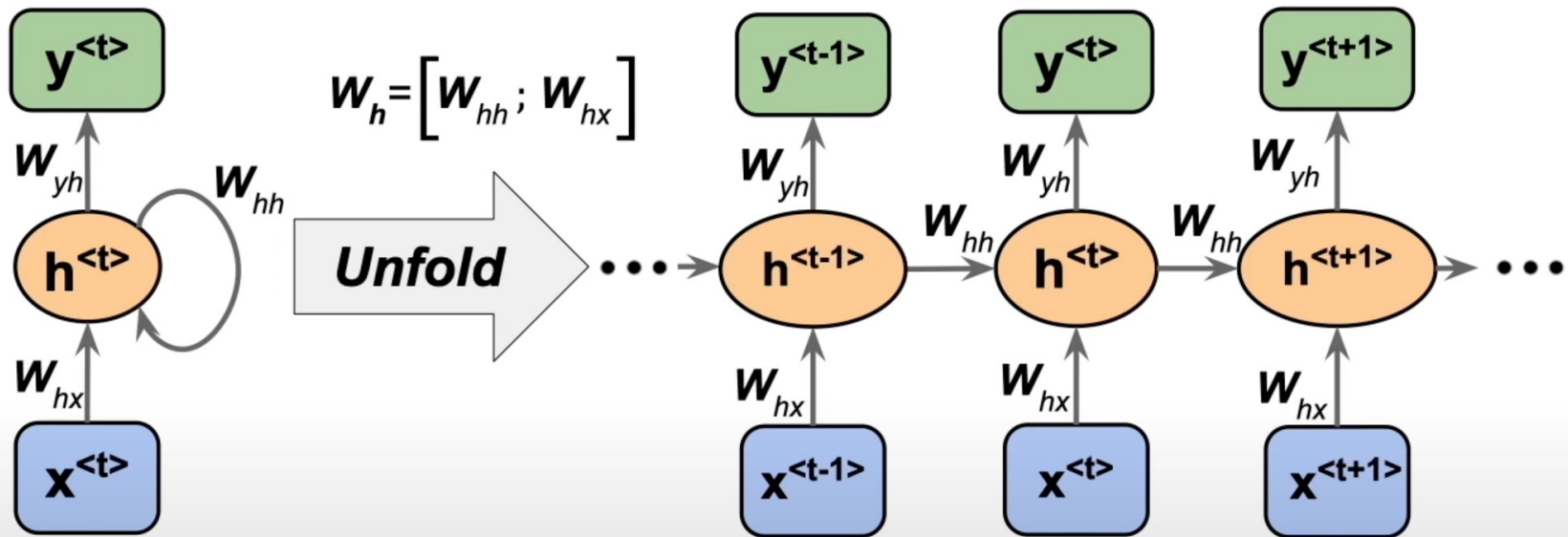
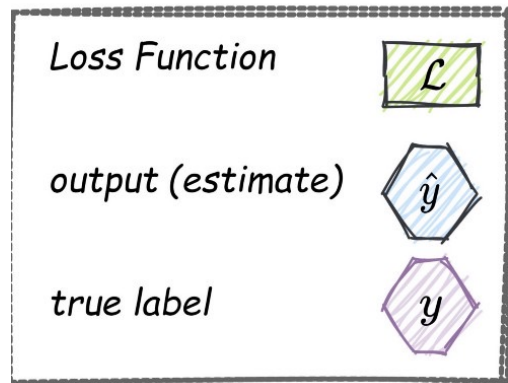


Image source: Sebastian Raschka, Vahid Mirjalili. *Python Machine Learning*. 3rd Edition. Packt, 2019

## **Backpropagation Through Time (BPTT)**

Training an RNN is done by defining a loss function ( $L$ ) that measures the error between the true label and the output, and minimizes it by using forward pass and backward pass. The following simple RNN architecture summarizes the entire backpropagation through time idea.

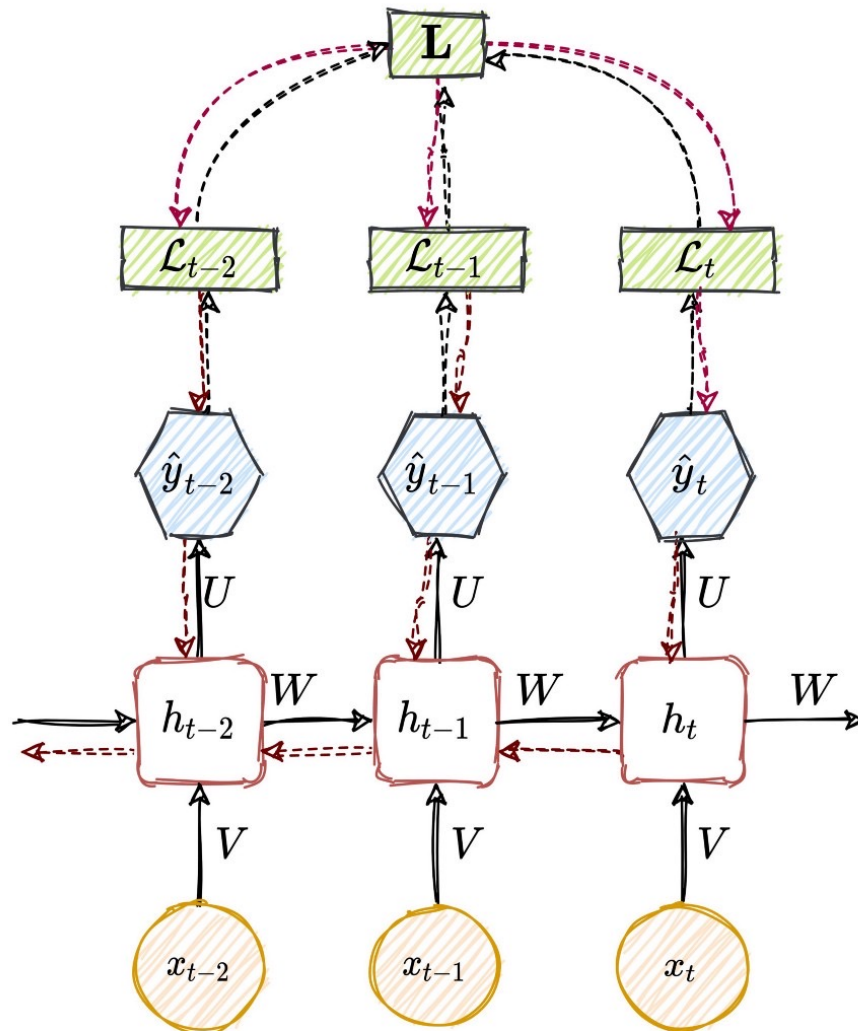
For a single time step, the following procedure is done: first, the input arrives, then it processes through a hidden layer/state, and the estimated label is calculated. In this phase, the loss function is computed to evaluate the difference between the true label and the estimated label. The total loss function,  $L$ , is computed, and by that, the forward pass is finished. The second part is the backward pass, where the various derivatives are calculated.



$$\mathbf{L} = \sum_i \mathcal{L}_i(\hat{y}_t, y_t)$$

Forward Pass:  
 $h_t, \hat{y}_t, \mathcal{L}_t, \mathbf{L}$

Backward Pass:  
 $\frac{\partial \mathbf{L}}{\partial U}, \frac{\partial \mathbf{L}}{\partial V}, \frac{\partial \mathbf{L}}{\partial W}, \frac{\partial \mathbf{L}}{\partial b_h}, \frac{\partial \mathbf{L}}{\partial b_y}$





The training of RNN is not trivial, as we backpropagate gradients **through layers** and also **through time**. Hence, in each time step we have to sum up all the previous contributions until the current one, as given in the equation:

$$\frac{\partial \mathbf{L}}{\partial W} = \sum_{i=0}^T \frac{\partial \mathcal{L}_i}{\partial W} \propto \sum_{i=0}^T \left( \prod_{i=k+1}^y \frac{\partial h_i}{\partial h_{i-1}} \right) \frac{\partial h_k}{\partial W}$$


Image by Author

In this equation, the contribution of a state at time step  $k$  to the gradient of the entire loss function  $\mathbf{L}$ , at time step  $t=T$  is calculated. The challenge during the training is in the ratio of the hidden state:

$$\frac{\partial \mathbf{L}}{\partial W} \propto \sum_{i=0}^T \left( \prod_{i=k+1}^y \frac{\partial h_i}{\partial h_{i-1}} \right) \frac{\partial h_k}{\partial W}$$

# The Vanishing and Exploding Gradients Problem

RNNs suffer from the problem of vanishing gradients, which hampers learning of long data sequences. The gradients carry information used in the RNN parameter update and when the gradient becomes smaller and smaller, the parameter updates become insignificant which means no real learning is done.



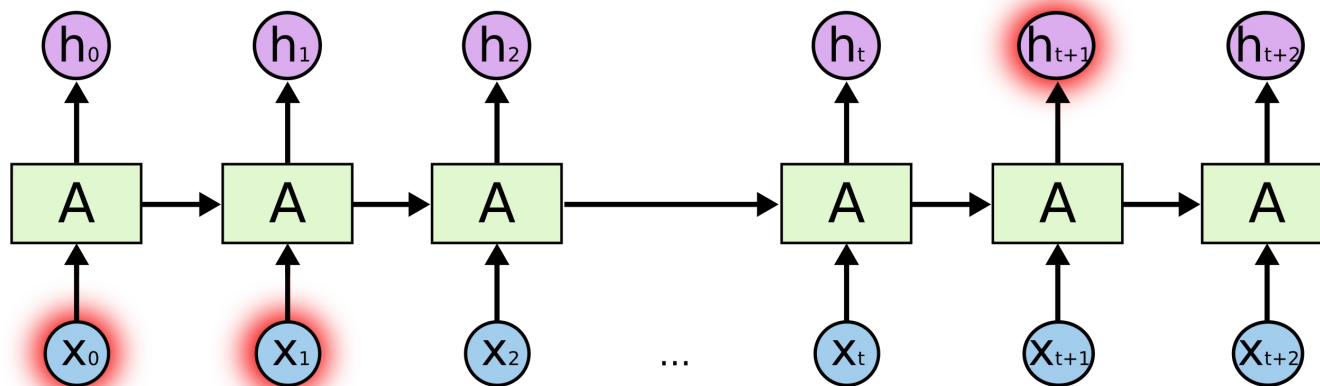
1. <i>Vanishing gradient</i>	$\left\  \frac{\partial h_i}{\partial h_{i-1}} \right\ _2 < 1$
2. <i>Exploding gradient</i>	$\left\  \frac{\partial h_i}{\partial h_{i-1}} \right\ _2 > 1$

In the first case, the term goes to zero exponentially fast, which makes it difficult to learn some long period dependencies. This problem is called the *vanishing gradient*.

In the second case, the term goes to infinity exponentially fast, and their value becomes a NaN due to the unstable process. This problem is called the *exploding gradient*.

# The Vanishing and Exploding Gradients Problem

Consider trying to predict the last word in the text “I grew up in France... I speak fluent *French*.” Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It’s entirely possible for the gap between the relevant information and the point where it is needed to become very large.



In theory, RNNs are absolutely capable of handling such “long-term dependencies.” A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs don’t seem to be able to learn them.

Thankfully, LSTMs don’t have this problem!

# Long-Short Term Memory architectures

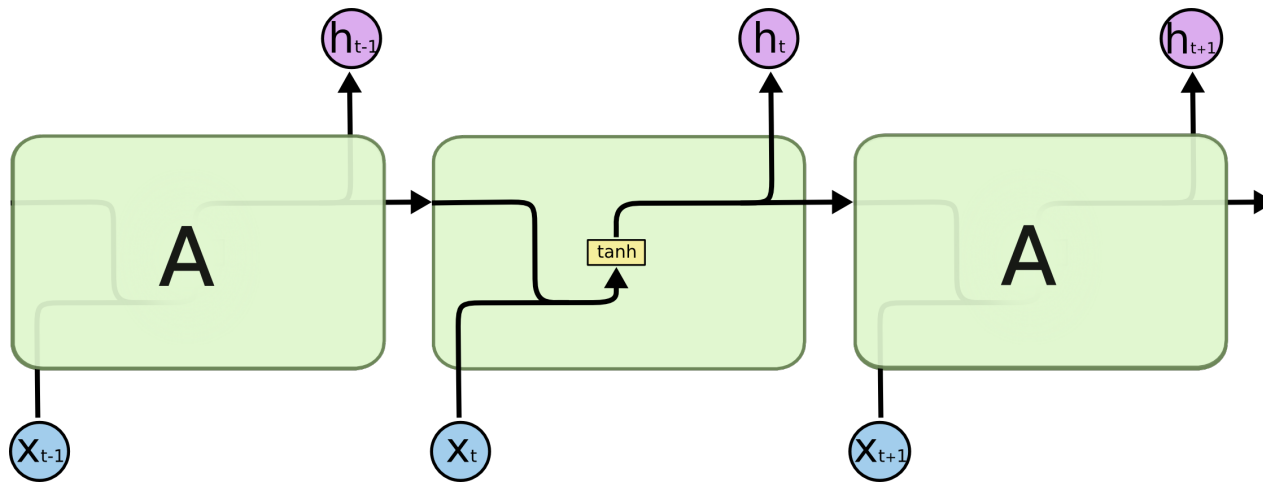
- Long Short Term Memory networks – usually just called “LSTMs” – are a special kind of RNN, capable of learning long-term dependencies.
- They were introduced by [Hochreiter & Schmidhuber \(1997\)](#), and were refined and popularized by many people in following work.
- LSTMs are explicitly designed to avoid the long-term dependency problem.
- Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

<https://medium.datadriveninvestor.com/how-do-lstm-networks-solve-the-problem-of-vanishing-gradients-a6784971a577>

<https://colah.github.io/posts/2015-08-Understanding-LSTMs/>

# Long-Short Term Memory architectures

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.



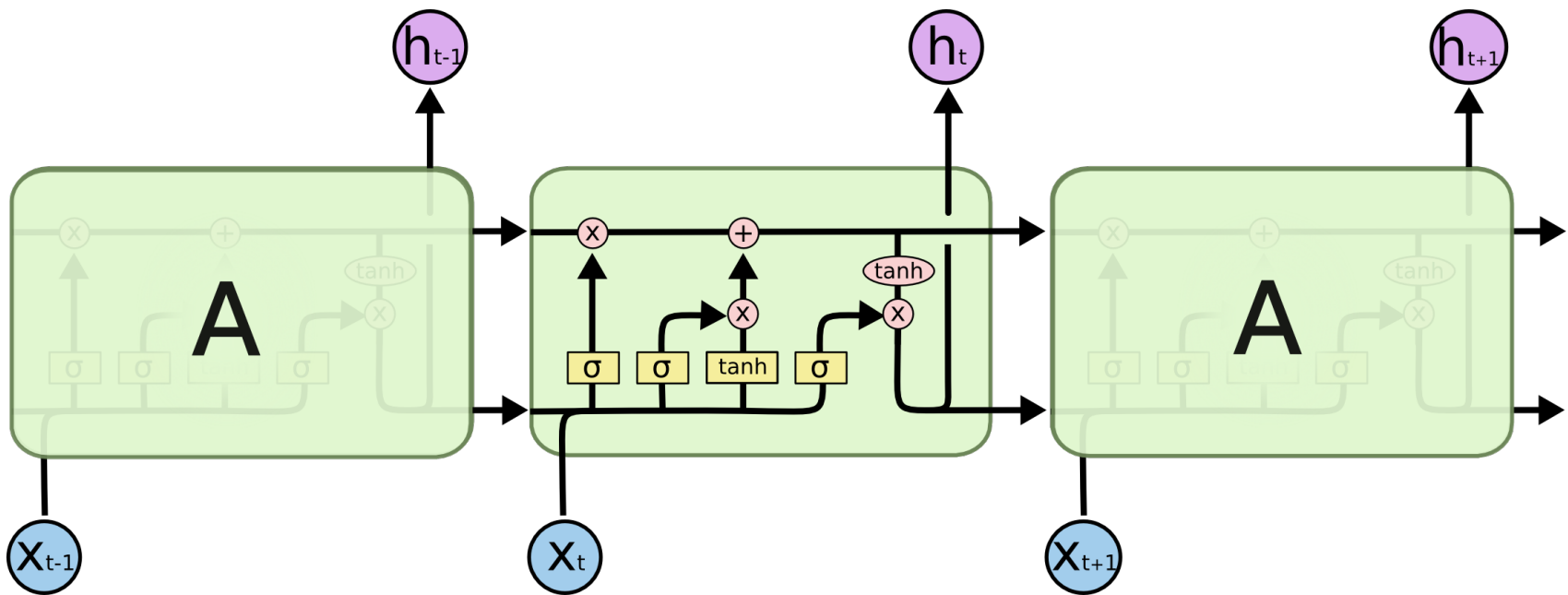
The repeating module in a standard RNN contains a single layer.

Update Hidden State

$$h_t = \tanh(W_{hh}^T h_{t-1} + W_{xh}^T x_t)$$

# Long-Short Term Memory architectures

LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.

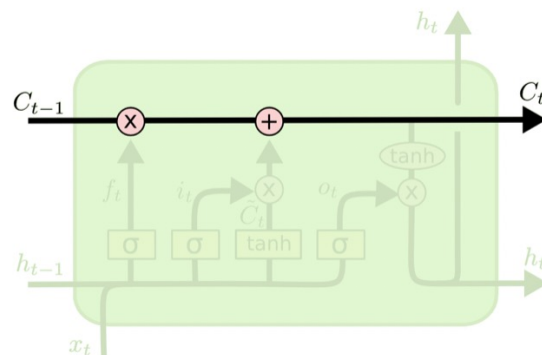


The repeating module in an LSTM contains four interacting layers.

# The Core Idea Behind LSTMs

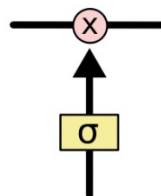
The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.

The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.



The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

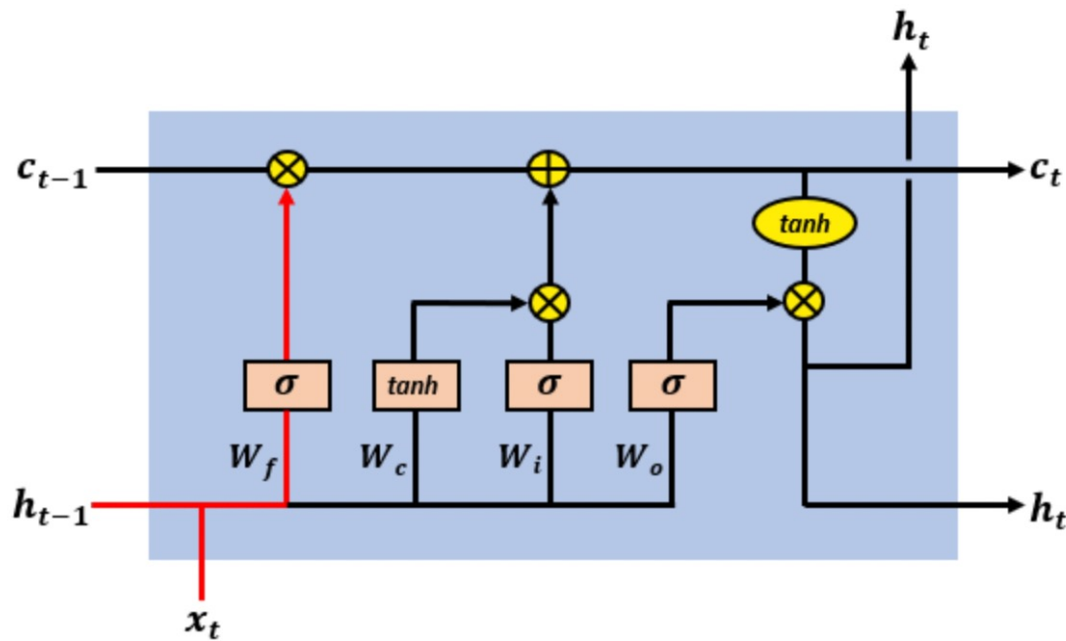
Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.



The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means “let nothing through,” while a value of one means “let everything through!”

An LSTM has three of these gates, to protect and control the cell state.

The forget gate controls what information in the cell state to forget, given new information than entered the network.



The LSTM forget gate update of the cell state

The forget gate's output is given by:

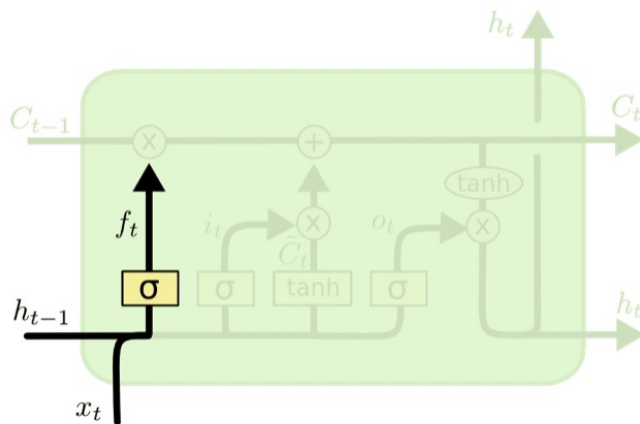
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t])$$



# Step-by-Step LSTM Walk Through

The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the “forget gate layer.” It looks at  $h_{t-1}$  and  $x_t$ , and outputs a number between 0 and 1 for each number in the cell state  $C_{t-1}$ . A 1 represents “completely keep this” while a 0 represents “completely get rid of this.”

Let's go back to our example of a language model trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.

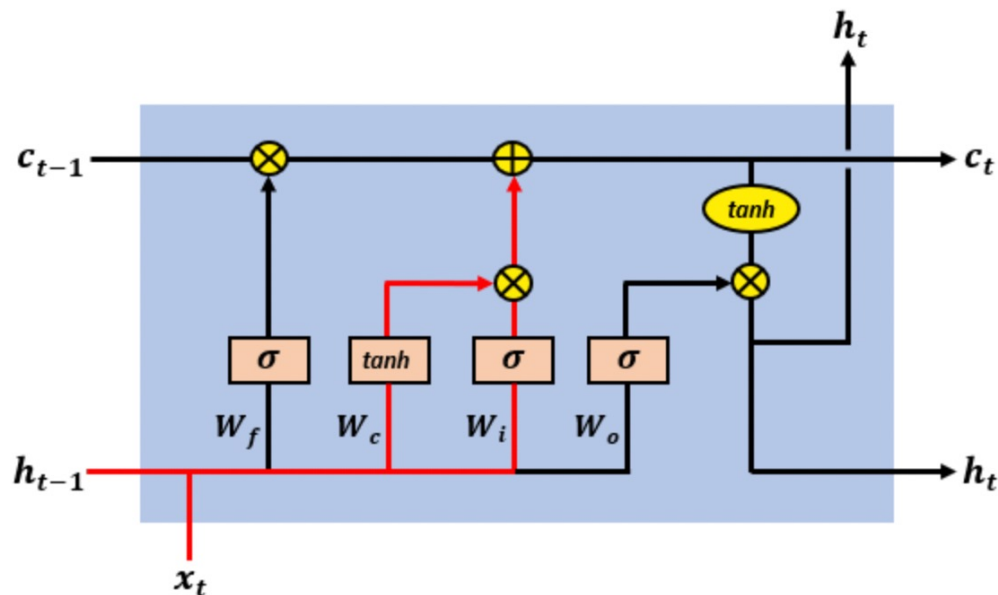


$$f_t = \sigma (W_f \cdot [h_{t-1}, x_t] + b_f)$$

# How gates help to solve the problem of vanishing gradients?

- The gates **regulate/control the flow of the information in LSTMs**.
- During forward propagation, gates control the flow of the information.
- They prevent any irrelevant information from being written to the state.
- Similarly, during backward propagation, they control the flow of the gradients.

The input gate controls what new information will be encoded into the cell state, given the new input information.



The LSTM input gate update of the cell state

The input gate's output has the form:

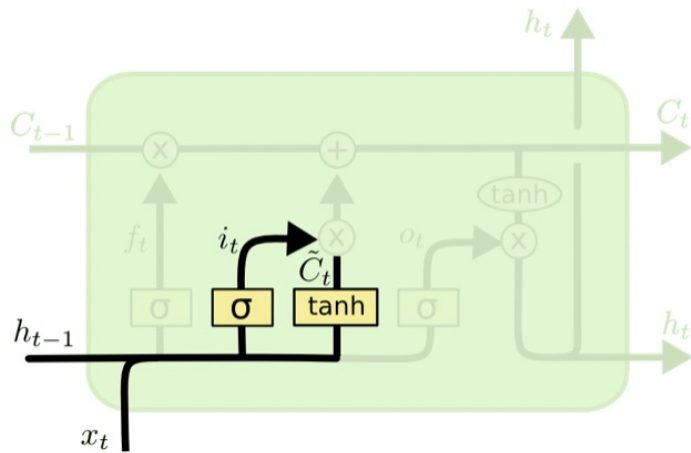
$$\tanh(W_c \cdot [h_{t-1}, x_t]) \otimes \sigma(W_i \cdot [h_{t-1}, x_t])$$

is equal to the element-wise product of the outputs of the two fully connected layers

# Step-by-Step LSTM Walk Through

The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values,  $\tilde{C}_t$ , that could be added to the state. In the next step, we'll combine these two to create an update to the state.

In the example of our language model, we'd want to add the gender of the new subject to the cell state, to replace the old one we're forgetting.



$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

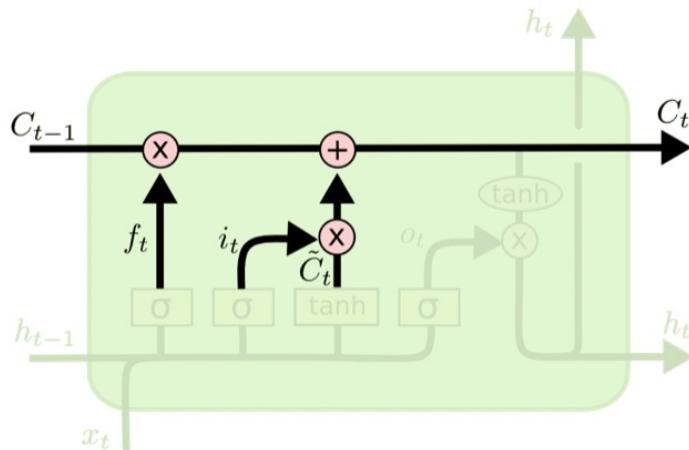
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

# Step-by-Step LSTM Walk Through

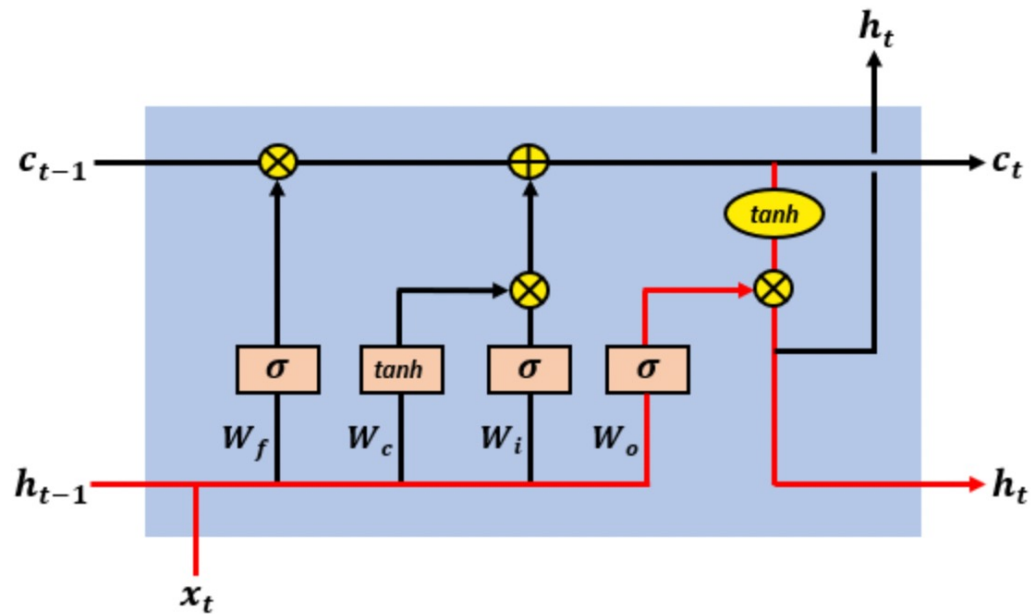
It's now time to update the old cell state,  $C_{t-1}$ , into the new cell state  $C_t$ . The previous steps already decided what to do, we just need to actually do it.

We multiply the old state by  $f_t$ , forgetting the things we decided to forget earlier. Then we add  $i_t * \tilde{C}_t$ . This is the new candidate values, scaled by how much we decided to update each state value.

In the case of the language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in the previous steps.



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



The LSTM output gate's action on the cell state

The output gate's activations are given by:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t])$$

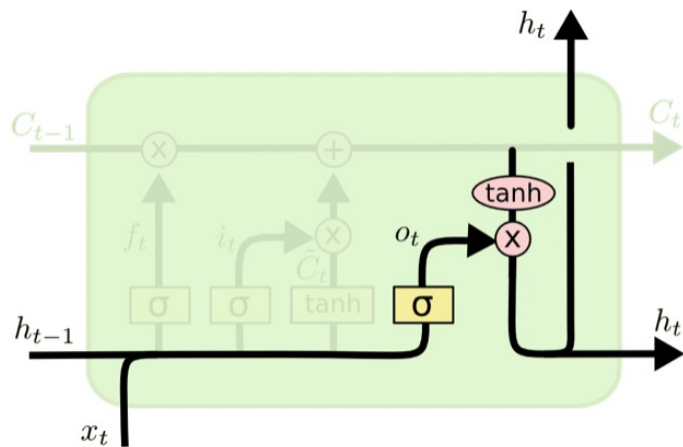
and the cell's output vector is given by:

$$h_t = o_t \otimes \tanh(c_t)$$

# Step-by-Step LSTM Walk Through

Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through **tanh** (to push the values to be between  $-1$  and  $1$ ) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.



$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

# How gates help to solve the problem of vanishing gradients?

- The gates **regulate/control** the flow of the information in **LSTMs**.
- During forward propagation, gates control the flow of the information. They prevent any irrelevant information from being written to the state.
- Similarly, during backward propagation, they control the flow of the gradients.

## The LSTM cell state

The long term dependencies and relations are encoded in the cell state vectors and it's the cell state derivative that can prevent the LSTM gradients from vanishing. The LSTM cell state has the form:

$$c_t = c_{t-1} \otimes f_t \oplus \tilde{c}_t \otimes i_t$$



# How gates help to solve the problem of vanishing gradients?

## Preventing Vanishing Gradients with LSTMs

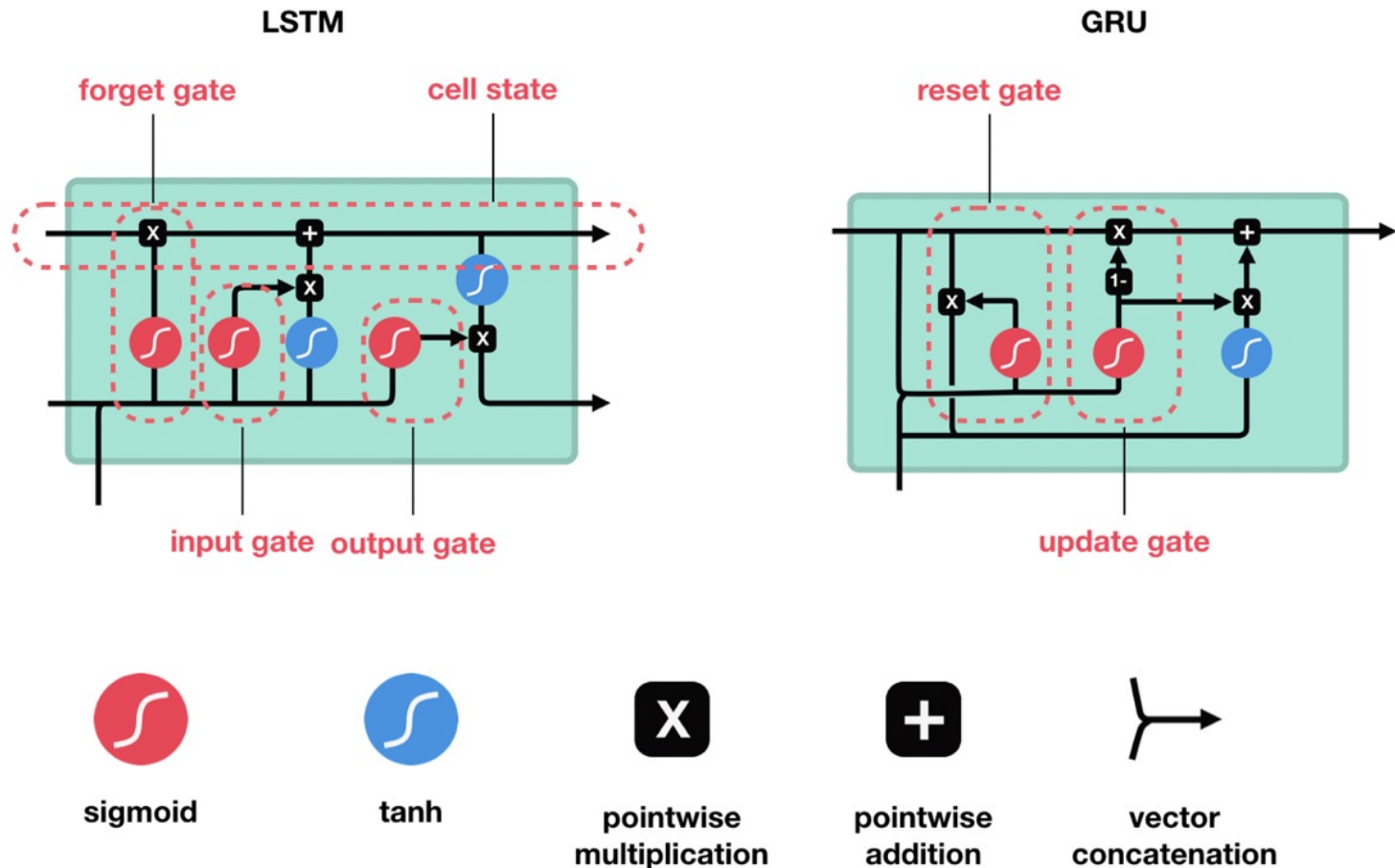
The biggest culprit in causing our gradients to vanish is that dastardly recursive derivative we need to compute:

If only this derivative was 'well behaved' (that is, it doesn't go to 0 or infinity as we backpropagate through layers) then we could learn long term dependencies!

<https://weberna.github.io/blog/2017/11/15/LSTM-Vanishing-Gradients.html>

## LSTM and GRU

- LSTM's and GRU's were created as the solution to short-term memory. They have internal mechanisms called gates that can regulate the flow of information.

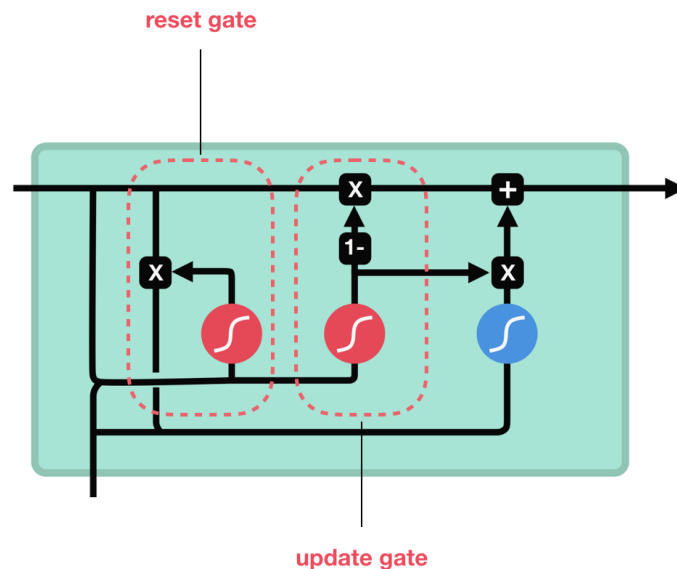


# LSTM and GRU

- These gates can learn which data in a sequence is important to keep or throw away.
- By doing that, it can pass relevant information down the long chain of sequences to make predictions.
- Almost all state of the art results based on recurrent neural networks are achieved with these two networks.
- LSTM's and GRU's can be found in speech recognition, speech synthesis, and text generation. You can even use them to generate captions for videos.

# LSTM and GRU

- The GRU is the newer generation of Recurrent Neural networks and is pretty similar to an LSTM.
- GRU's got rid of the cell state and used the hidden state to transfer information. It also only has two gates, a reset gate and update gate.



# LSTM and GRU

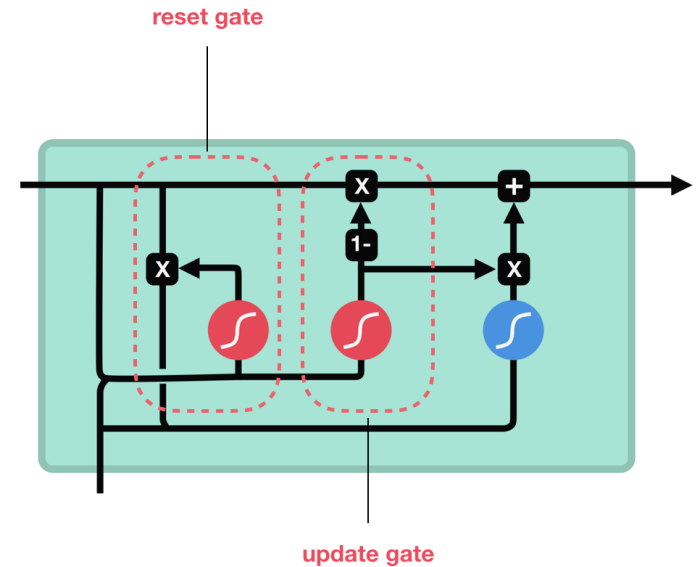
## Update Gate

The update gate acts similar to the forget and input gate of an LSTM. It decides what information to throw away and what new information to add.

## Reset Gate

The reset gate is another gate is used to decide how much past information to forget.

GRU's has fewer tensor operations; therefore, they are a little speedier to train than LSTM's. There isn't a clear winner which one is better. Researchers and engineers usually try both to determine which one works better for their use case.



# Learning Resources

- Charu C. Aggarwal, Neural Networks and Deep Learning, Springer, 2018.
- Eugene Charniak, Introduction to Deep Learning, MIT Press, 2018.
- Ian Goodfellow, Yoshua Bengio, Aaron Courville, Deep Learning, MIT Press, 2016.
- Michael Nielsen, Neural Networks and Deep Learning, Determination Press, 2015.
- Deng & Yu, Deep Learning: Methods and Applications, Now Publishers, 2013.

<https://medium.datadriveninvestor.com/how-do-lstm-networks-solve-the-problem-of-vanishing-gradients-a6784971a577>

<https://colah.github.io/posts/2015-08-Understanding-LSTMs/>

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

***Thank you***