RECURRENT NEURAL NETWORKS:

Feedforward Network

- A trained feed forward network can be exposed to any random collection of inputs, therefore not following any order.
- In feedforward networks, the relationship between the input and the output can be expressed as a static function.
- Output at time *t* is independent of output at time *t*-1.
- But sometimes predicting an output requires its previous output.
- Understanding a sentence needs understanding of its previous sentence(s).

Recurrent Neural Network(RNN)

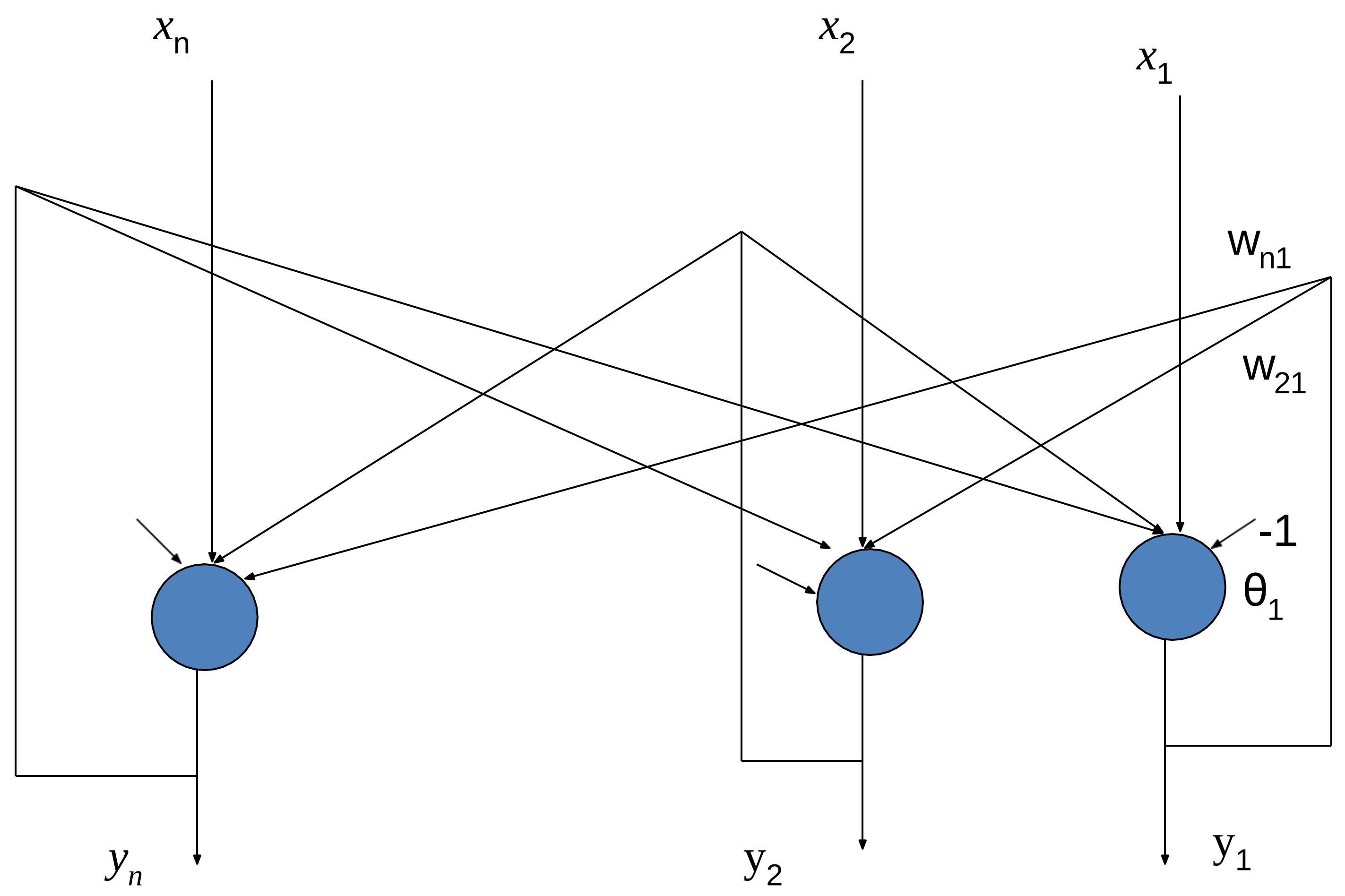
- The network we have encountered so far are known as feedforward networks since they have no loops in them.
- But loops are more common in the brain.
- Hopfield network is a simple neural network model that has feedback connections and present a model of human memory, known as an associative memory.
- RNN is a type of Neural Network where the output from previous step are fed as input to the current step.

RECURRENT

- Unlike feedforward neural networks, RNNs can use their internal state (memory) to process sequences of inputs.
- In RNN, self-loops and backward connections between the nodes are allowed.
- Single layer RNN and multilayer RNN.

HOPFIELD NETWORK

- A **Hopfield network** is a single layer recurrent ANN popularized by John Hopfield in 1982.
- The recurrent network performs sequential updating process.
- Hopfield nets serve as CAM systems with binary threshold nodes.
- Two versions of the network: *Discrete* and *Continuous* Hopfield
- Hopfield networks classify binary pattern vectors.



Every pair of units *i* and *j* in a Hopfield network have a connection described by the weight w_{ij}.

- No self feedback, $w_{ii}=0$ and weights are symmetric, $w_{ij} = w_{ji}$
- Each node has an external input and threshold θ_j, $j = 1\ldots n$

How to "train" a Hopfield network

- The constraint that weights be symmetric guarantees that the energy function decreases monotonically.
- Calculate the values of the weights without any training.
- Suppose we wish to store the set of patterns V^s, s = 1, ..., n.
- $W_{ij} = \sum_{s=1}^{n} (2V_i^s - 1)(2V_j^s - 1)$

Updating Rule

- The values of neurons *i* and *j* will converge if the weight between them is positive. Similarly, they will diverge if the weight is negative.

- **Asynchronous**: Only one unit is updated at a time. This unit can be picked at random, or a pre-defined order can be imposed from the very beginning.

- **Synchronous**: All units are updated at the same time. This requires a central clock to the system in order to maintain synchronization.

– Each neuron receives a weighted sum of the inputs from other neurons:

$$y_{\mathrm{i}} = \sum_{\substack{j=1 \\ j \neq i}}^{n} x_j w_{ij}$$; If y_i is positive the state of the neuron will be 1, otherwise 0:

- Update rule: $y_i^{(k+1)} = \mathrm{sgn}\left(\sum_{\substack{j=1 \\ j \neq i}}^{n} w_{ij} y_{\mathrm{j}}^{(k)} + x_i - \theta_{\mathrm{i}} \right)$; $i = 1 \ldots n$
- The connection weight $w_{ij} > 0$ implies:

  that $y_j = 1$ the contribution of j in the weighted sum is positive.

- Thus, y_i is pulled by j towards its value 1 and vice versa.

- Consider a two node discrete Hopfield network with $w_{12} = w_{21} = -1$, $w_{11} = w_{22} = 0$, $x_1 = x_2 = 0$ and $\theta_1 = \theta_2 = 0$ and the initial output vector $\boldsymbol{y}^{(0)} = [-1\ -1]^T$

- According to the asynchronous update rule, only one node is considered for updating at a time.

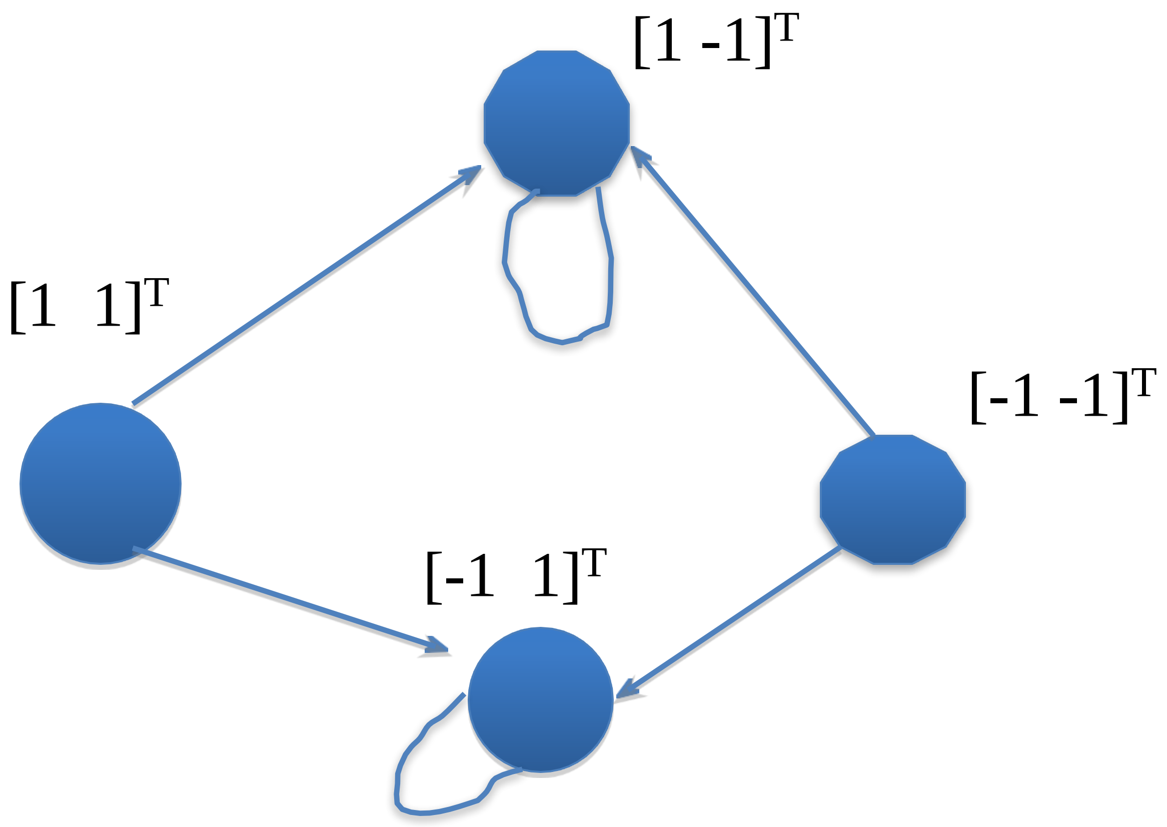
- Assume that the first node is chosen for update.
- So $y_1^{(1)} = \text{sgn}\ (w_{12}y_2^{(0)}) = \text{sgn}\ [\ (-1)\ (-1)] = 1$ and $\boldsymbol{y}^{(1)} = [1\ -1]^T$

- $y_2^{(2)} = \text{sgn}\ (w_{21}y_1^{(1)}) = \text{sgn}\ [\ (-1)\ (1)] = -\ 1$ and $\boldsymbol{y}^{(1)} = [1\ -1]^T$

- No further output state changes will occur and that the state $[1\ -1]^T$ is an equilibrium state of the network.

Asynchronous Update

- Using different initial outputs, we can obtain the state transition diagram, in which the vectors $[1\ -1]^T$ and $[-1\ 1]^T$ are the two equilibria of the system.



Synchronous Update

- For the same initial vector $[-1 \; -1]^T$

•we have $\mathbf{y}(^{1)} = [\,1 \; 1]^T$ and $\mathbf{y}(^{2)} = [\,-1 \quad -1]^T$

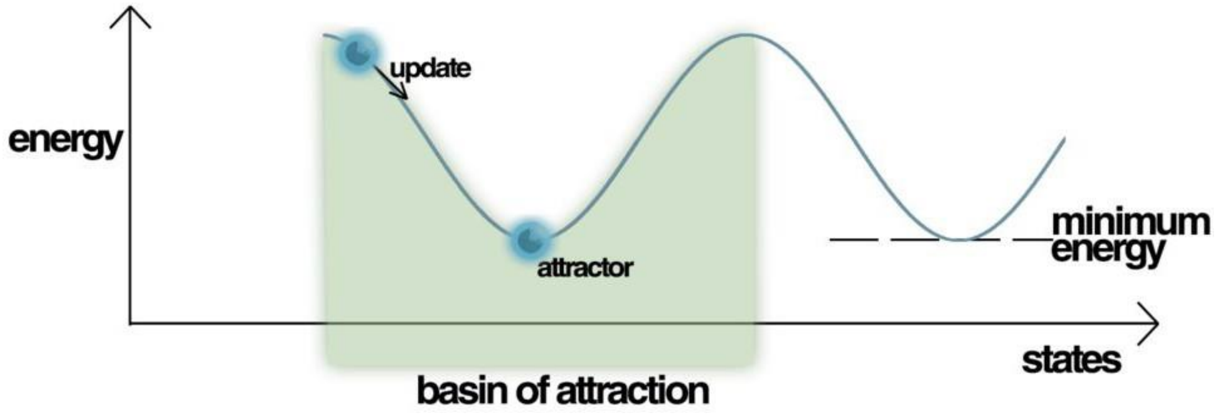
Same as initial output vector $\mathbf{y}^{(0)}$

- The synchronous update produces a cycle of two states rather than a single equilibrium state.

- Synchronous update may cause the networks to converge either to fixed points or limit cycles.

Energy

- The network is characterized by an energy function to evaluate the stability property of a discrete Hopfield network.
- Energy Landscape of a Hopfield Network, highlighting the current state of the network (up the hill), an attractor state to which it will eventually converge, i.e. a minimum energy level and a basin of attraction shaded in green.
- Hopfield nets have a scalar value associated with each state of the network referred to as the "energy", E, of the network.



STABILITY PROPERTY

- Energy function to characterize the behavior of the network:

$$E = -\tfrac{1}{2} \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} w_{ij}\, y_i\, y_j - \sum_{i=1}^{n} x_i\, y_i + \sum_{i=1}^{n} \theta_i\, y_i$$

- In stable network the energy function decreases during change of state of any node.
- Assume that node *i* changes state from $y_i^{(k)}$ to $y_i^{(k+1)}$.
- So its output changes from +1 to –1 or vice versa.

- Change in energy:

$$\Delta E = E(y_i^{(k+1)}) - E(y_i^{(k)}) = -\left(\sum_{j=1,\ j\neq i}^{n} w_{ij} y_j^{(k)} + x_i - \theta_i\right)(y_i^{(k+1)} - y_i^{(k)})$$

$\Delta E = -(\text{net}_i)\, \Delta y_i$ where $\Delta y_i = y_i^{(k+1)} - y_i^{(k)}$; $y_j^{(k+1)} = y_j^{(k)}$; $w_{ij} = w_{ji}$ and $w_{ii} = 0$

Case I: If $y_i^{(k)}$ has changed from -1 to +1; i.e. ($\Delta y_i = +2$), then net_i must be positive and **ΔE** will be negative.

Case II: If $y_i^{(k)}$ has changed from +1 to -1; i.e. ($\Delta y_i = -2$), then net_i must have been negative and **ΔE** will be again negative.

Case III: If $y_i^{(k)}$ has not changed ($\Delta y_i = 0$), so **ΔE** = 0. Thus **ΔE** ≤ 0 always.

Why RNN?

- The idea behind RNNs is to make use of sequential information, present in the input, output, or in most general case both.

- RNNs accept an input vector **x** and give an output vector **y**.

- Crucially this output vector's contents are influenced not only by the input you just fed in, but also on the entire history of inputs you've fed in the past.

- In theory RNNs can make use of information in arbitrarily long sequences, but in practice they are limited to looking back only a few steps.

Why RNN?

- RNNs are called *recurrent* because they perform the same task for every element of a sequence, with the output being depended on the previous computations.

- RNNs have a “memory” which captures information about what has been calculated so far.

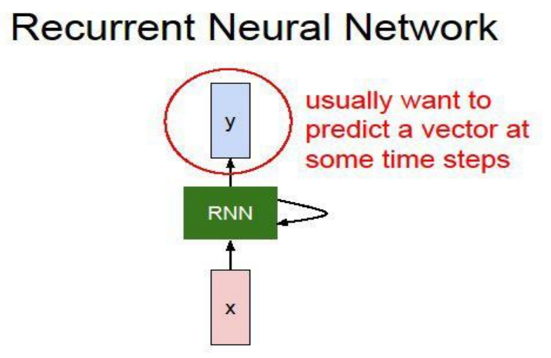
- To maintain word order
- To share parameters across the sequence
- To keep track of long term dependencies

RNNs as sequence Modeling

- Speech Recognition: Input X is a speech signal transforms to a sentence, i.e. output Y is sequence of words.
- Music generation: X may be an empty set, Y is a sequence of tunes of music.
- Sentiment classification: X is the sentences, Y is review of a movie.
- Video activity recognition: X is Frame of videos, Y is Interpretation
- Name entity recognition: X is a sequence of words as sentence, Y is identification of people

Recurrent Neural Network

- Humans don’t start their thinking from scratch every second because thoughts have persistence.
- In a traditional neural network we assume that all inputs (and outputs) are independent of each other.
- RNNs address this issue with loops in them, allowing information to persist.



- Supervised learning as label data is to be provided

- when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words.
- Thus RNN came into existence, which solved this issue with the help of a Hidden Layer.
- The main and most important feature of RNN is **Hidden state**, which remembers some information about a sequence.
- h_t is current state, h_{t-1} is previous state and x_t is input state.

$$h_t = f(h_{t-1}, x_t)$$

W_{hh} is weight at current neuron

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

W_{xh} is weight at input neuron

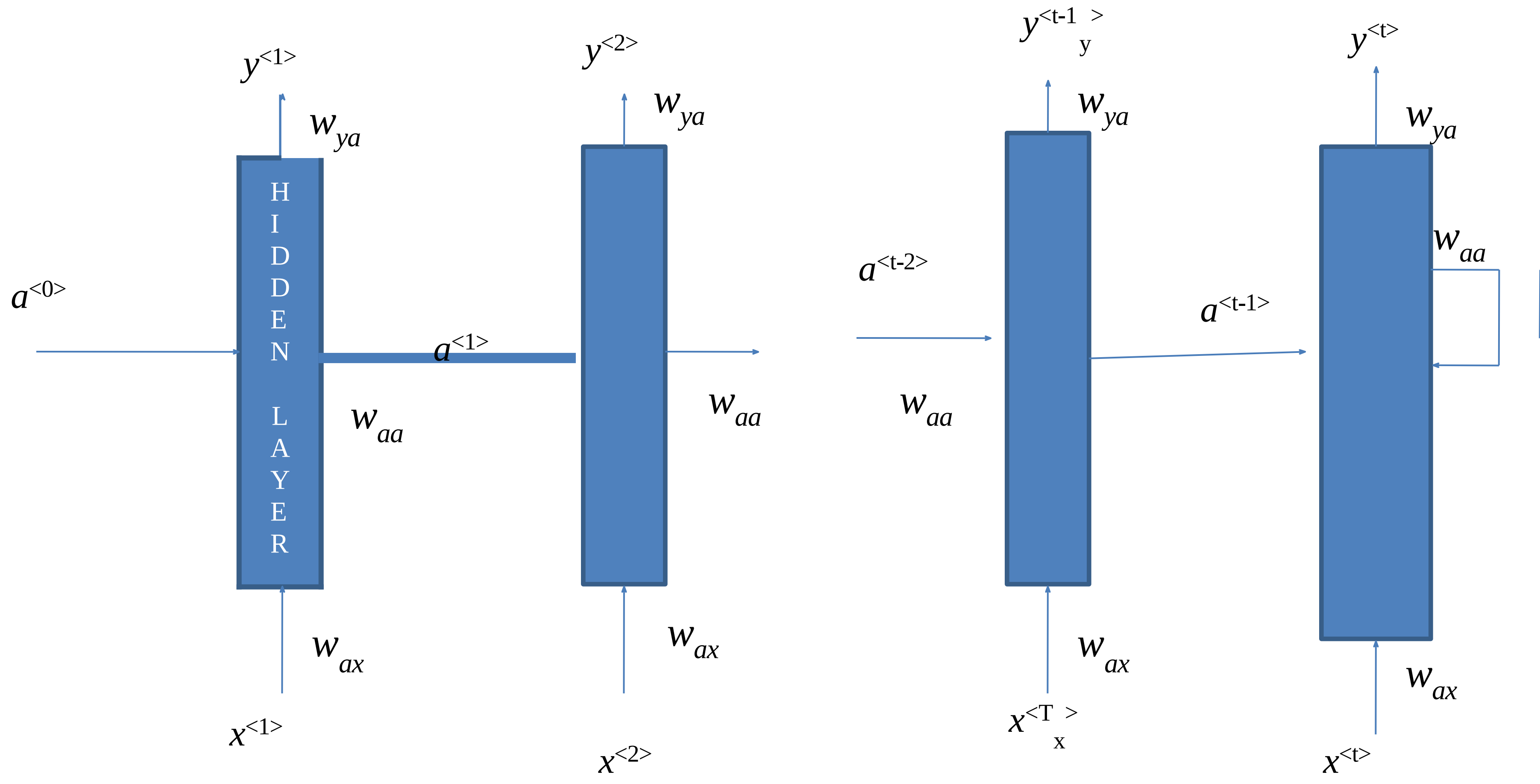
$$y_t = W_{hy}h_t$$

y_t is output and W_{hy} is weight at output layer

Representing words

- As a vocabulary or Dictionary of words of finite size.
- One vector representation for each word, i.e. $X^{<i>}$ of dimension say, 10,000
- Provide input-output pair, but for each input, output dimension may be different.
- One layer in the RNN for each word of a sentence.
- There is many more sentences.

Forward Propagation

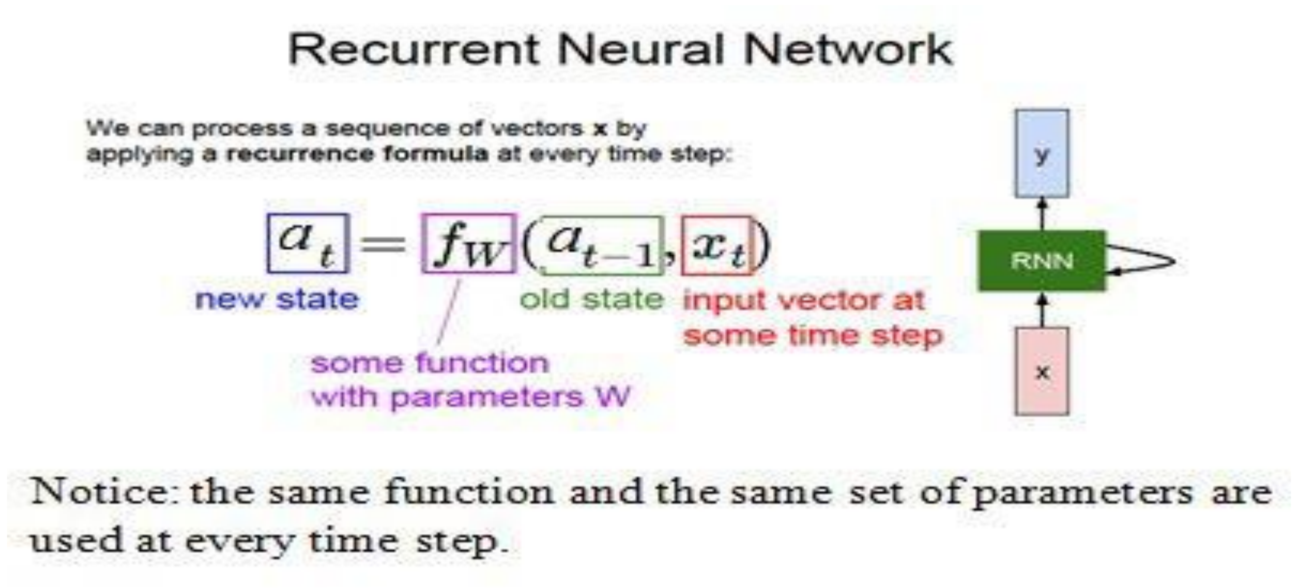


- Activation value passes on next layer $a^{<1>}$
- The parameters at each time step are shared

Forward Propagation

- For each time step t, the activation $a^{<t>}$ and the output $y^{<t>}$ are expressed as:
- $a^{<0>} = 0$, $a^{<1>} = f_w\,(w_{aa}\, a^{<0>} + w_{ax}\, x^{<1>})$: tanh/ReLU
- $y^{<1>} = h\,(w_{ya}\, a^{<1>})$: sigmoid/softmax
- $a^{<t>} = f_w\,(w_{aa}\, a^{<t-1>} + w_{ax}\, x^{<t>})$
- $y^{<t>} = h\,(w_{ya}\, a^{<t>})$

- where W_{ax}, W_{aa}, W_{ya} are coefficients that are shared temporally and f_w, h activation functions.


Recurrent Neural Network

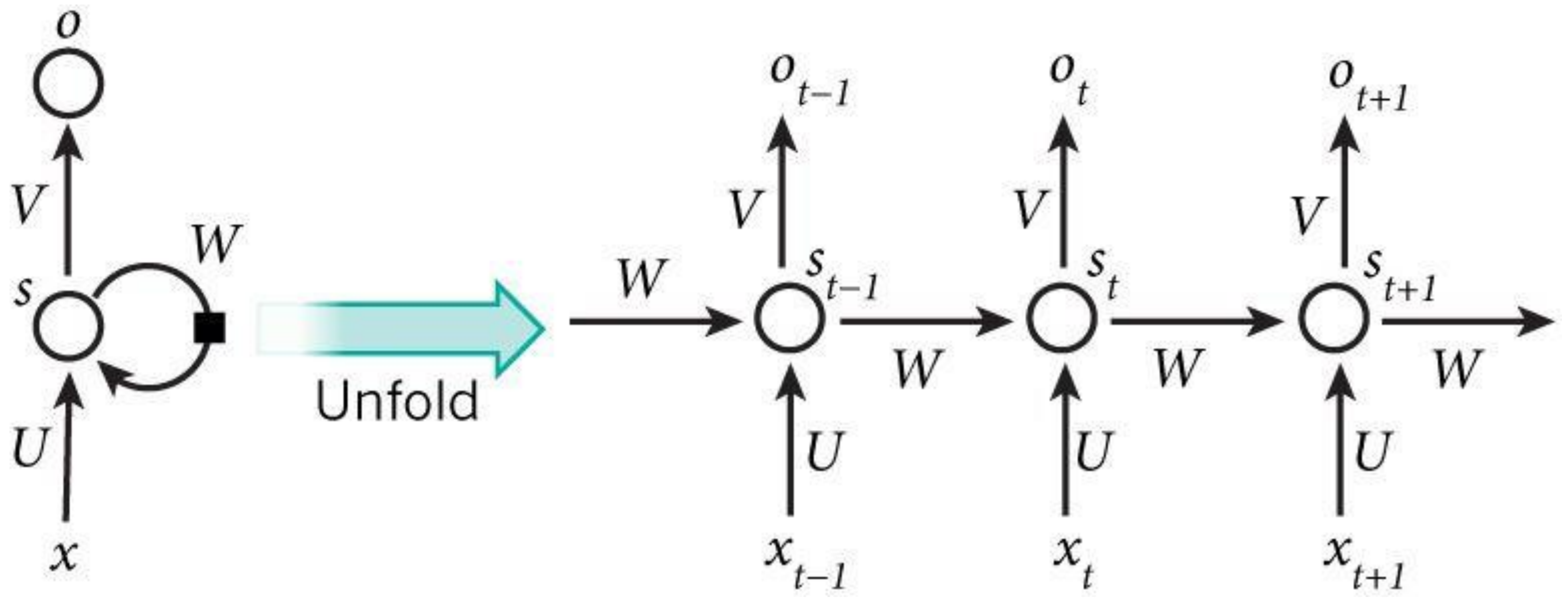
We can process a sequence of vectors **x** by applying a **recurrence formula** at every time step:

$$\boxed{a_t} = \boxed{f_W}(\boxed{a_{t-1}}, \boxed{x_t})$$

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

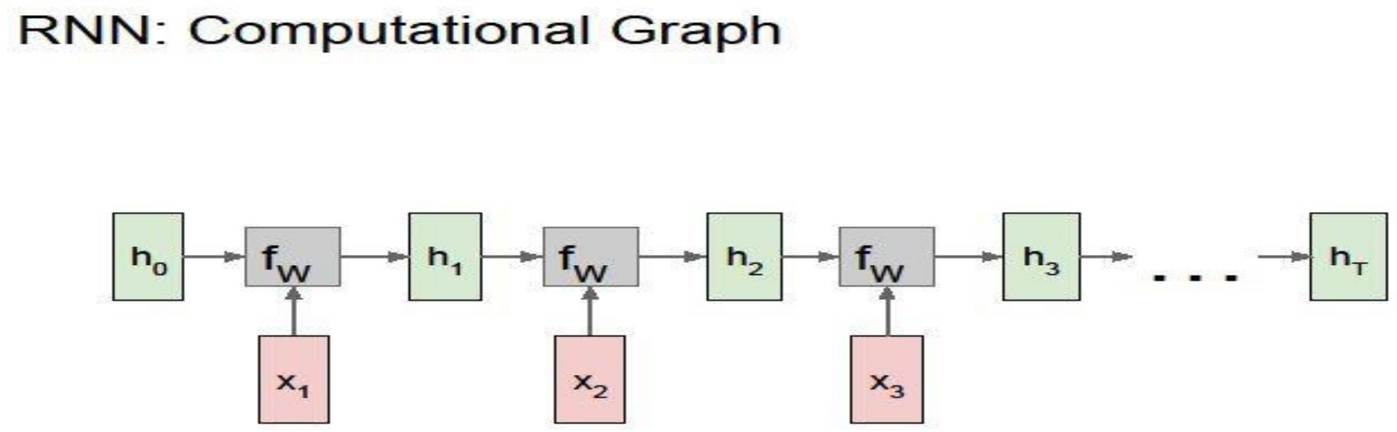
- The weight matrices are filters that determine how much importance to accord to both the present input and the past hidden state.
- The error they generate will return via backpropagation and be used to adjust their weights until error is not reduced further.

The network for the complete sequence



- A recurrent neural network and the unfolding in time of the computation involved in its forward computation.
- If the sequence we care about is a sentence of 5 words, the network would be unrolled into a 5-layer neural network, one layer for each word.

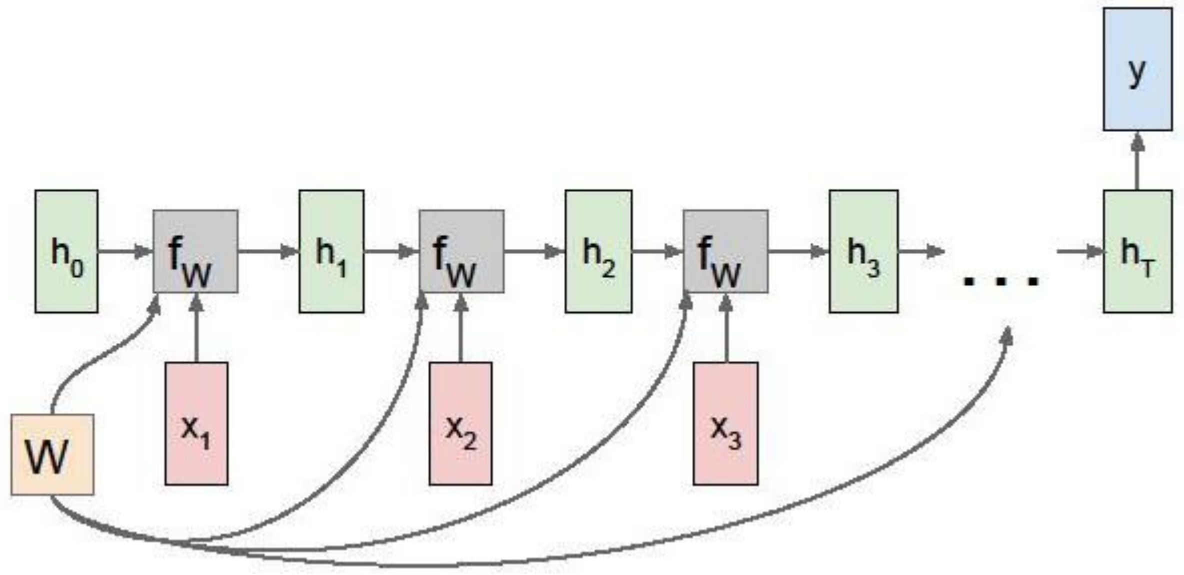
An unrolled Recurrent Neural Network



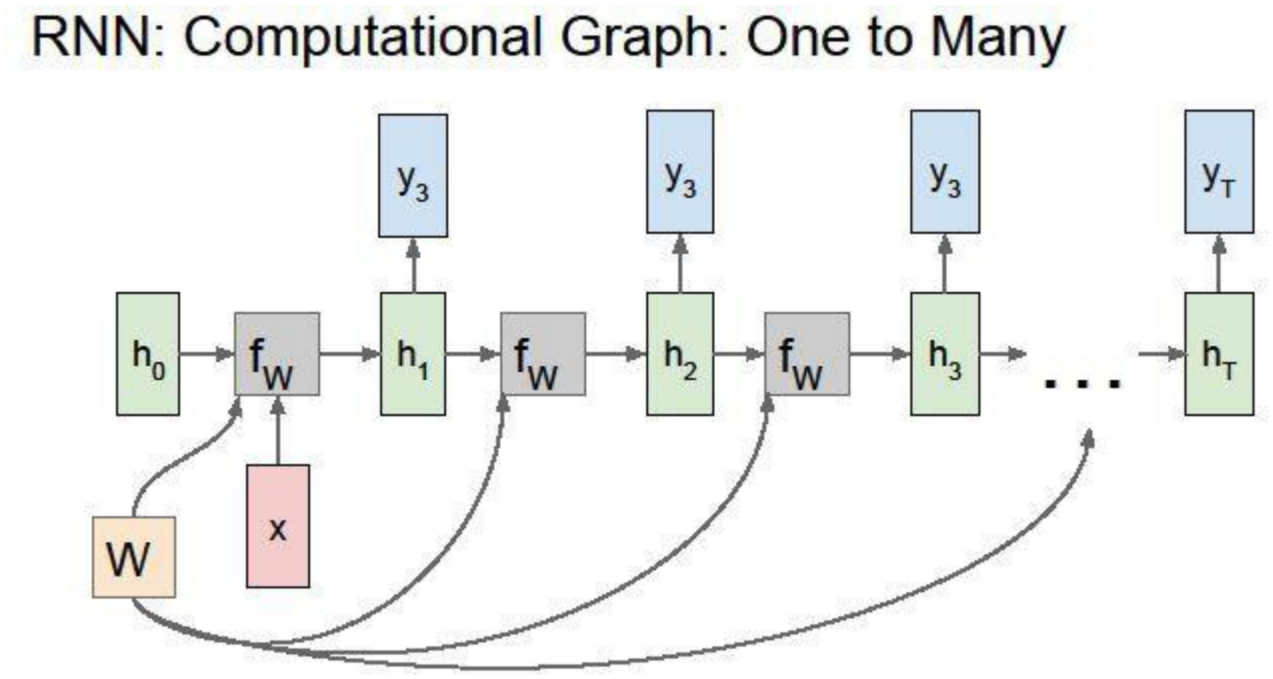
- A chunk of neural network, A, looks at some input x_t and outputs a value h_t
- A loop allows information to be passed from one step of the network to the next.
- A RNN can be thought of as multiple copies of the same network, each passing a message to a successor.
- This chain-like nature reveals that RNNs are intimately related to sequences and lists and the natural architecture of neural network to use for such data.

Sentiment classification

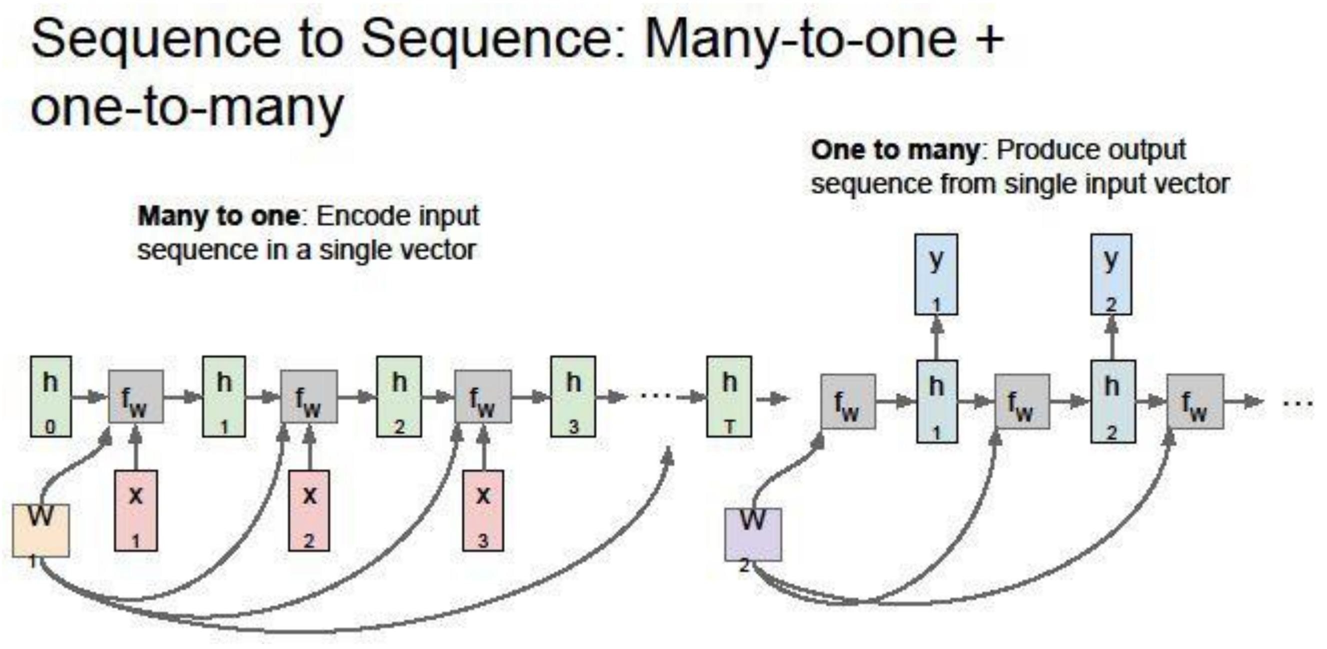
RNN: Computational Graph: Many to One



Music Generation



Name Entity Recognition

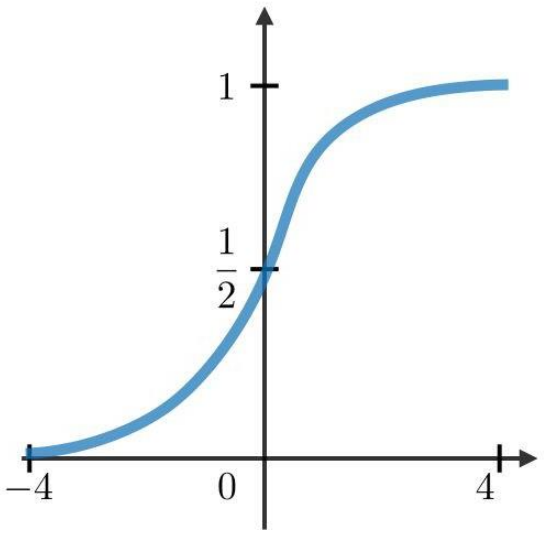

Sequence to Sequence: Many-to-one + one-to-many
Many to one: Encode input sequence in a single vector
One to many: Produce output sequence from single input vector
h_0
f_W
h_1
h_2
h_3
h_T
x_1
x_2
x_3
W_1
W_2
y_1
y_2
...

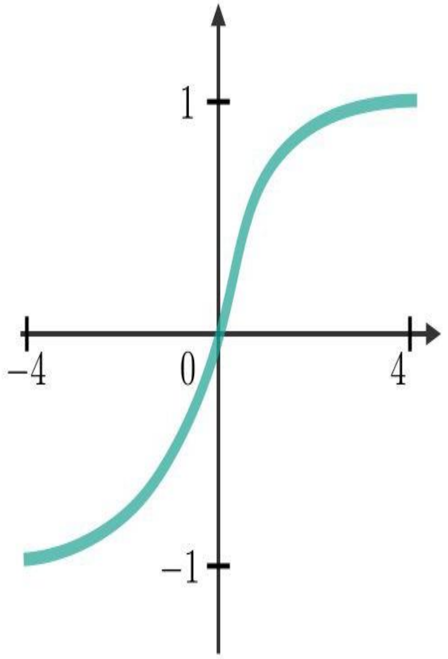
Loss function

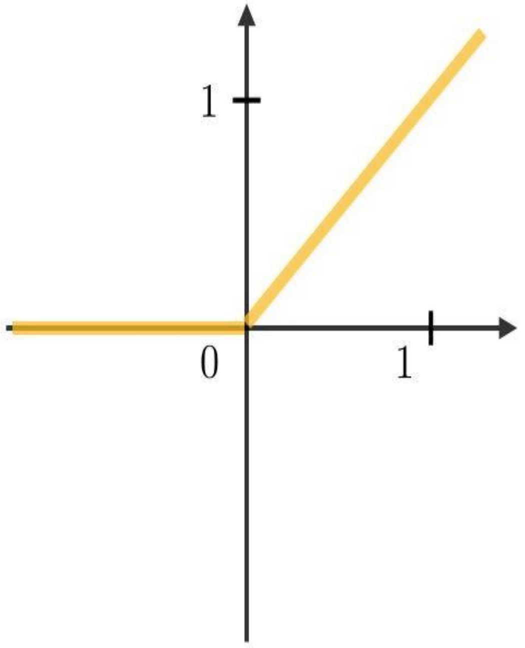
- In the case of a recurrent neural network, the loss function L of all time steps is defined based on the loss at every time step.
- $L(y', y) = \sum_{t=1 \,..\, T_y} [L(y'^{<t>}, y^{<t>})]$
- **Backpropagation through time** is done at each point in time. At time step *T*, the derivative of the loss *L* with respect to weight matrix *W* is expressed as:

$$\frac{\partial L^{(T)}}{\partial W} = \sum_{t=1}^{T} \frac{\partial L^{(T)}}{\partial W} \Big|_{(t)}$$

Activation functions







Sigmoid:

$$g(z) = \frac{1}{1 + e^{-z}}$$

Tanh

$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

ReLU

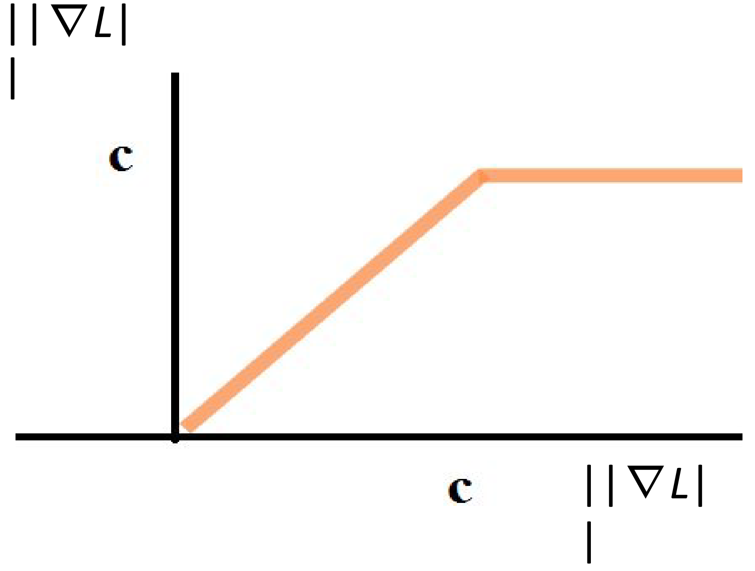
$$g(z) = \max(0, z)$$

Issues with Backpropagation:

- it is difficult to capture long term dependencies because of multiplicative gradient that exponentially decreasing/increasing with number of layers.

- **Gradient clipping** is a technique used to cope with the exploding gradient problem.

•By capping the maximum value for the gradient, this phenomenon is controlled in practice.



Training a RNN

- RNN uses *backpropagation* algorithm applied at every time stamp, called *backpropagation* through time (BTT).
- x_t is the input at time step t. For example, x_1 could be a one-hot vector corresponding to the second word of a sentence.
- s_t is the hidden state at time step t.
- It is the "memory" of the network, captures information about what happened in all the previous time steps.
- s_t is calculated based on the previous hidden state and the input at the current step: $s_t = g(\mathrm{U}x_t + \mathrm{W}s_{t-1})$.

Training of RNN

- The function *g* usually is a nonlinearity such as tanh or ReLU,
- s_{t-1} required to calculate the first hidden state, typically initialized to all zeroes.
- o_t is the output at step *t*. If we want to predict the next word in a sentence it would be a vector of probabilities across our vocabulary: $o_t = \text{softmax}(Vs_t)$
- The output at step o_t is calculated solely based on the memory at time t.

Training of RNN

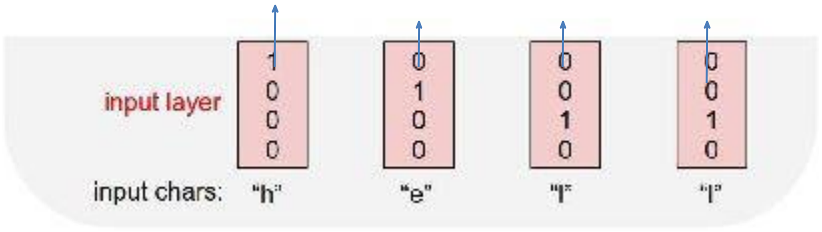
- Unlike a traditional neural network, which uses different parameters at each layer, a RNN shares the same parameters (above) across all steps.
- This reflects the fact that the same task is performed at each step, just with different inputs reduces the total number of parameters we need to learn.
- Outputs may not be available at each time step, but depending on the task this may not be necessary.

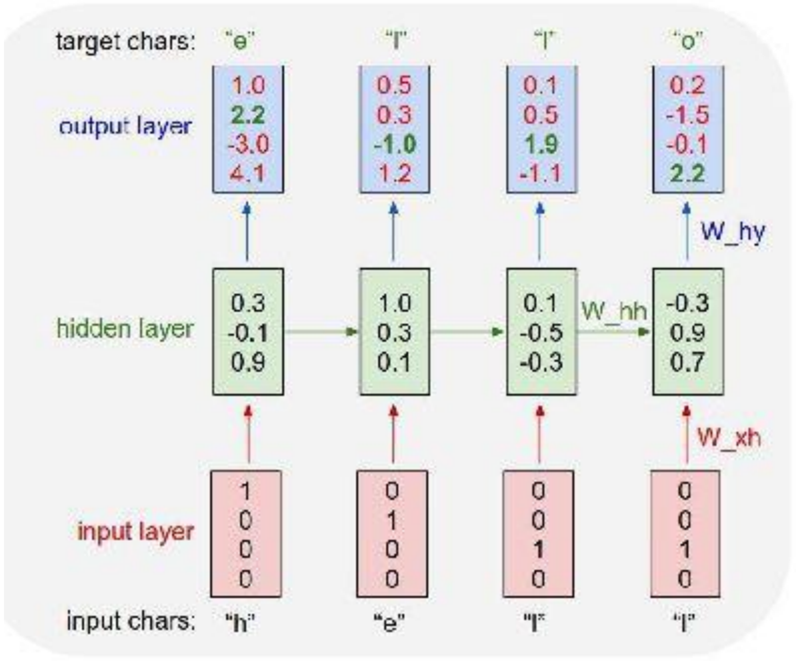
- For example, when predicting the sentiment of a sentence we may only care about the final output, not the sentiment after each word.
- Similarly, we may not need inputs at each time step.
- The main feature of an RNN is its **hidden state**, which captures some information about a sequence.

Example: Character-level Language Model

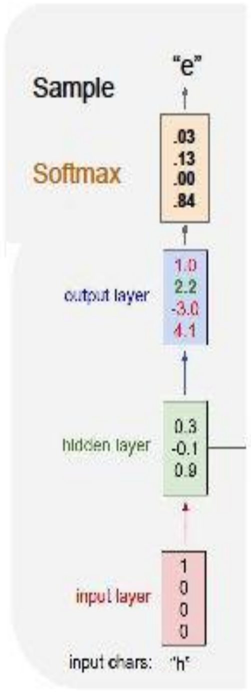
Vocabulary: [h,e,l,o]

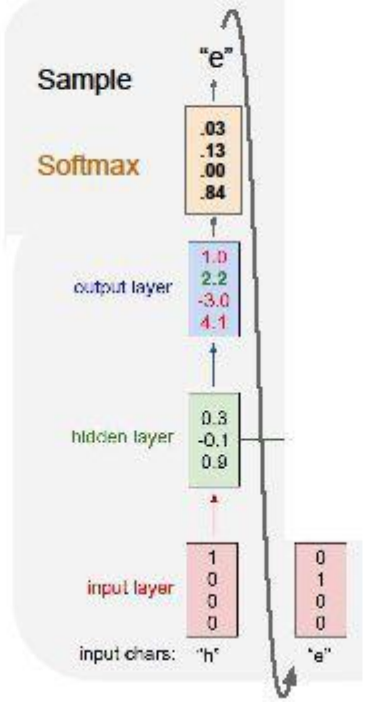
Example training sequence: **“hello”**

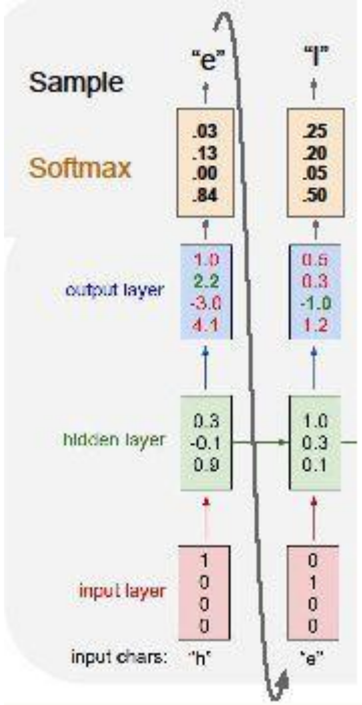


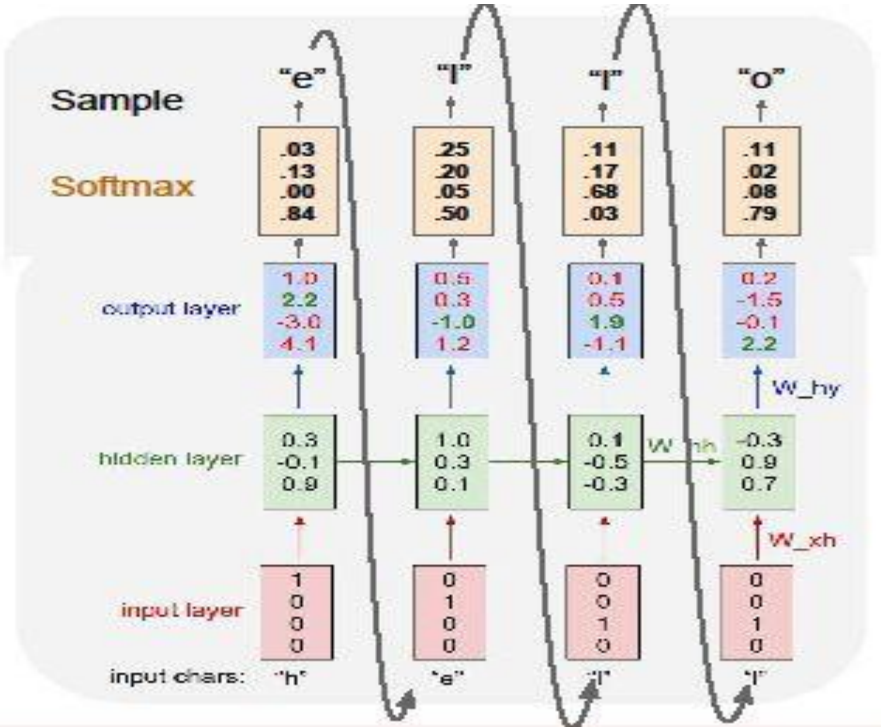


At test-time sample characters one at a time, feed back to model



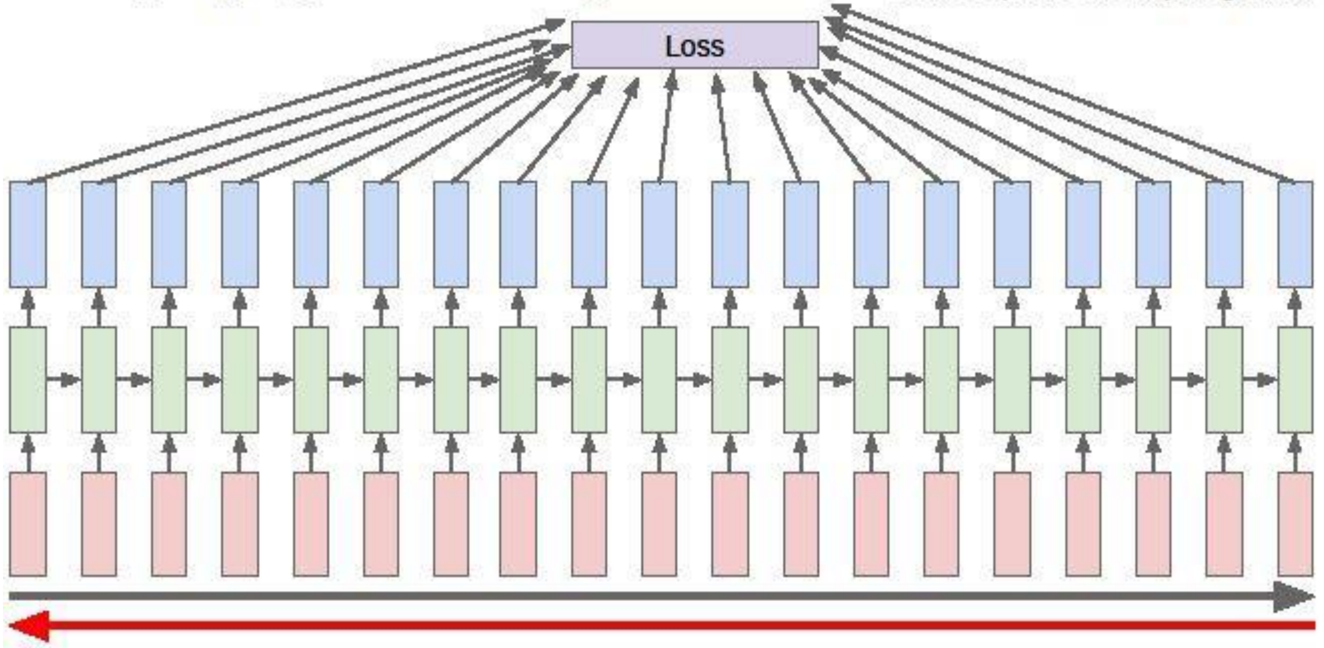






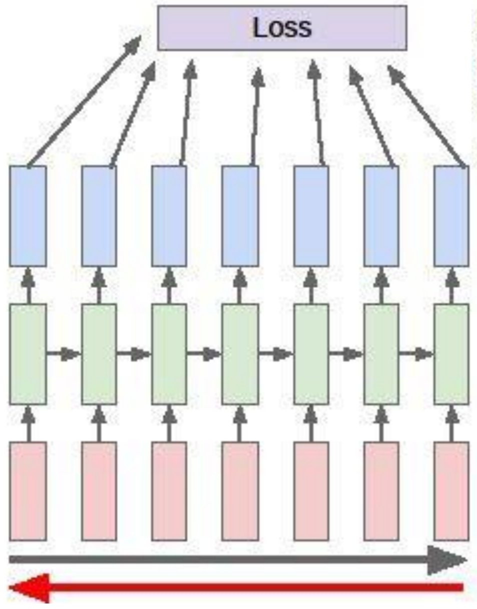
Backpropagation through time

Forward through entire sequence to compute loss, then backward through entire sequence to compute gradient

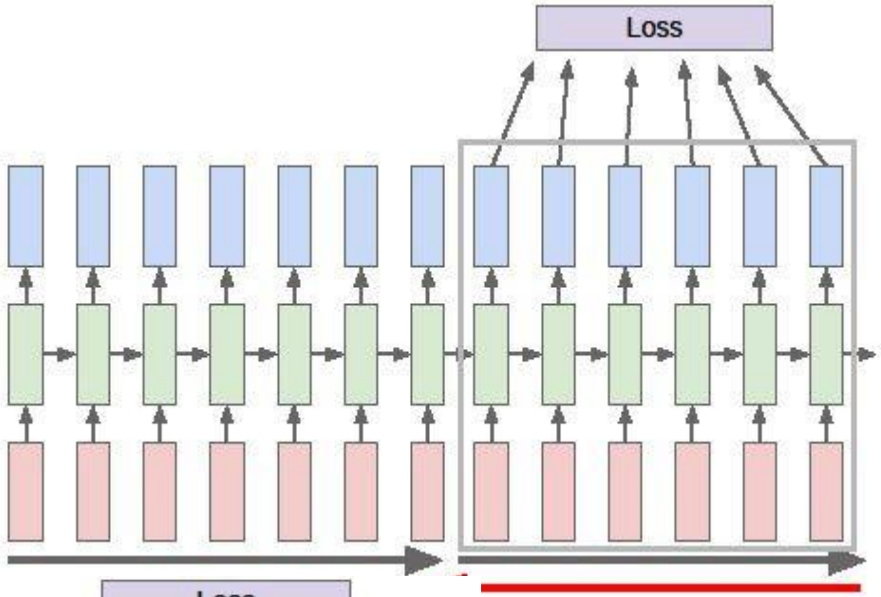


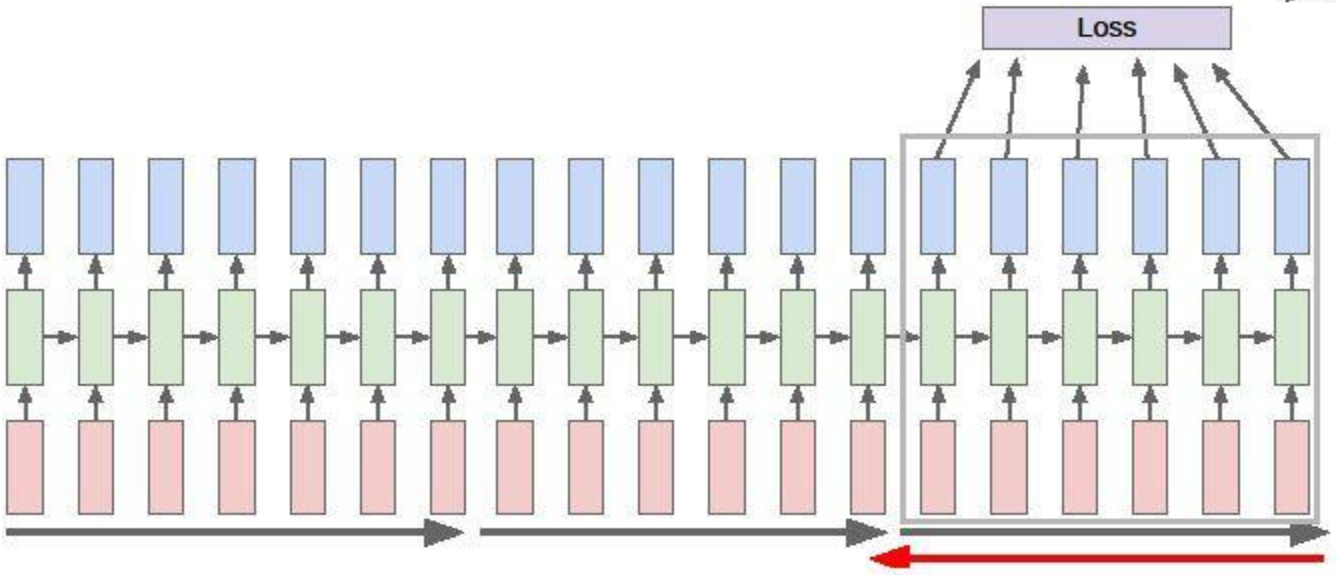
Truncated Backpropagation through time

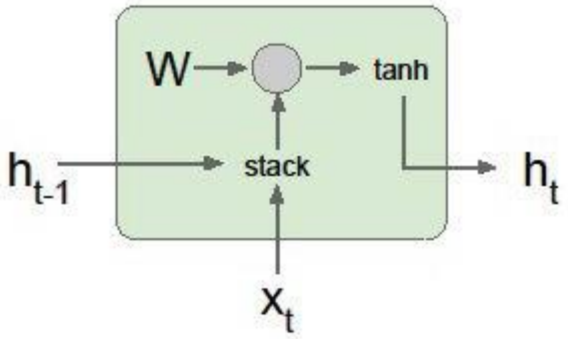
Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps



Run forward and backwarc through chunks of the sequence instead of whole sequence





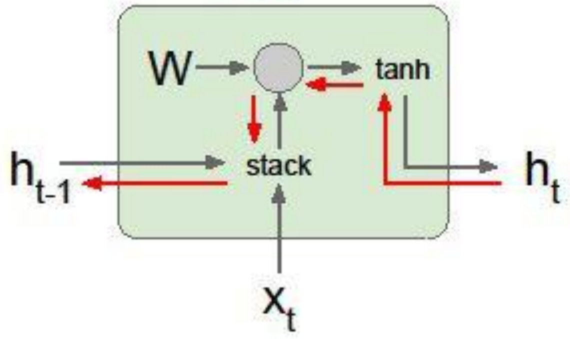


$$
\begin{aligned}
h_t &= \tanh(W_{hh} h_{t-1} + W_{xh} x_t) \\
&= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right) \\
&= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)
\end{aligned}
$$

Vanilla RNN Gradient Flow

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013

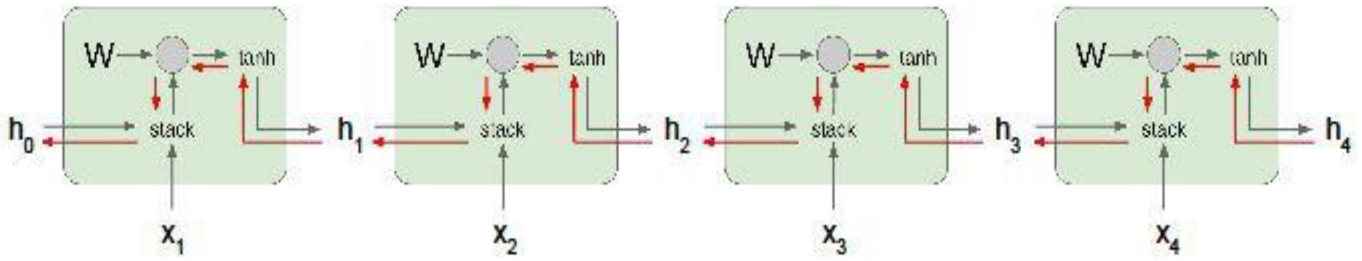
Backpropagation from h_t to h_{t-1} multiplies by W (actually W_{hh}^T)



$$
\begin{aligned}
h_t &= \tanh(W_{hh} h_{t-1} + W_{xh} x_t) \\
&= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right) \\
&= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)
\end{aligned}
$$

Vanilla RNN Gradient Flow

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013



Computing gradient of h_0 involves many factors of W (and repeated tanh)

Largest singular value > 1: **Exploding gradients** → **Gradient clipping**: Scale gradient if its norm is too big

```
grad_norm = np.sum(grad * grad)
if grad_norm > threshold:
  grad *= (threshold / grad_norm)
```

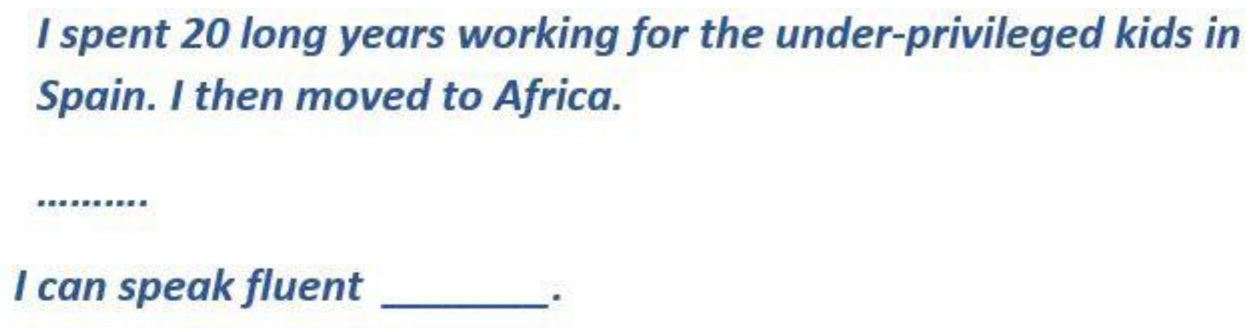
Largest singular value < 1: **Vanishing gradients** → Change RNN architecture

- Sequential information is preserved in the recurrent network's hidden state.
- Correlations between events separated by many moments, and these correlations are called "long-term dependencies".
- An event downstream in time depends upon, and is a function of, one or more events that came before.

Limitation of RNN

- Recurrent Neural Networks work just fine when dealing with short-term dependencies.
- The colour of milk is ---
- Nothing to do with the context of the statement.
- The RNN need not to remember what was said before this, or what was its meaning.
- Vanilla RNNs fail to understand the context behind an input.
- Something that was said long before, cannot be recalled when making predictions in the present.

Problem of RNN



I spent 20 long years working for the under-privileged kids in Spain. I then moved to Africa.

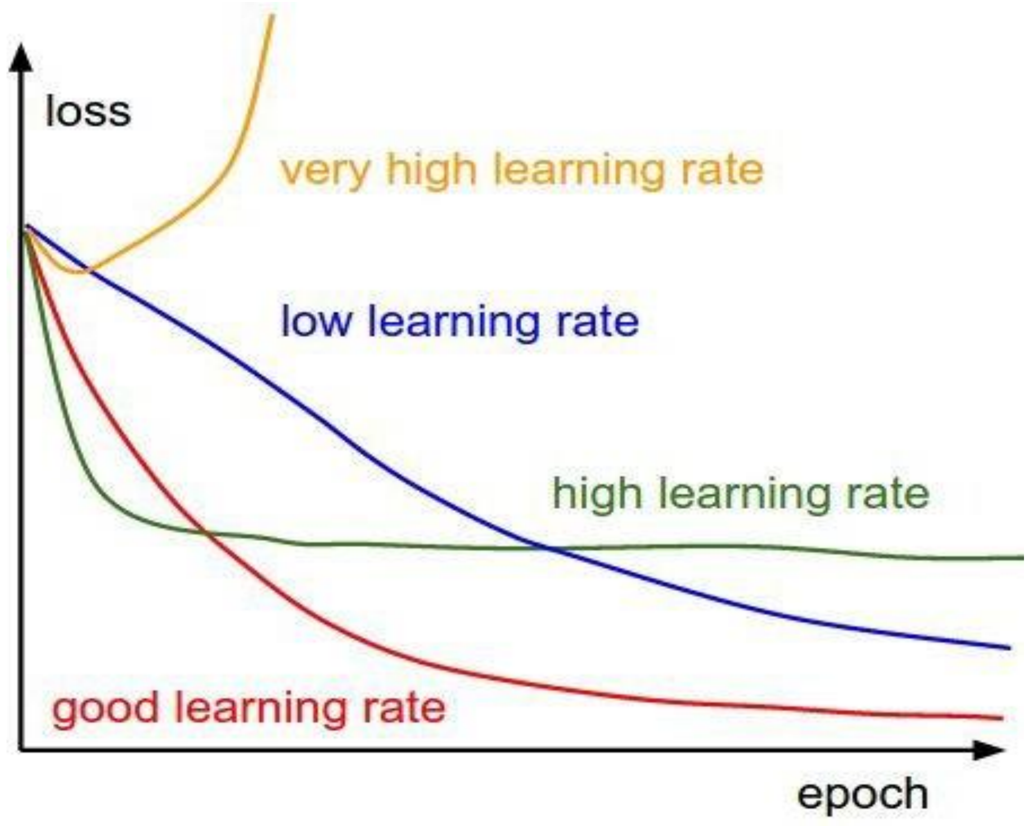
……….

I can speak fluent ________.

- To make a proper prediction, the RNN needs to remember this context.
- The relevant information may be separated from the point where it is needed, from the irrelevant data.
- The reason behind this is the problem of **Vanishing Gradient.**

Variants of Gradient Descent algorithm- Vanilla Gradient Descent

Learning rate is a hyper-parameter



Long-Term Dependencies

- RNN remembers things for just small durations of time, i.e. if we need the information after a small time it may be reproducible, but once a lot of words are fed in, this information gets lost somewhere.
- In RNN there is no consideration for *‘important’* information and *‘not so important’* information.
- This issue can be resolved using Long Short-Term Memory (LSTM) Networks.
- LSTMs on the other hand, make small modifications to the information

Long Short Term Memory as Gated Cell

- Information can be stored in, or read from a cell, much like data in a computer’s memory.
- The cell makes decisions about what to store, and when to allow reads, writes and erases, via gates that open and close.
- The core concept of LSTMs are the cell state, and its various gates.
- The cell state act as a transport highway that transfers relative information all the way down the sequence chain.

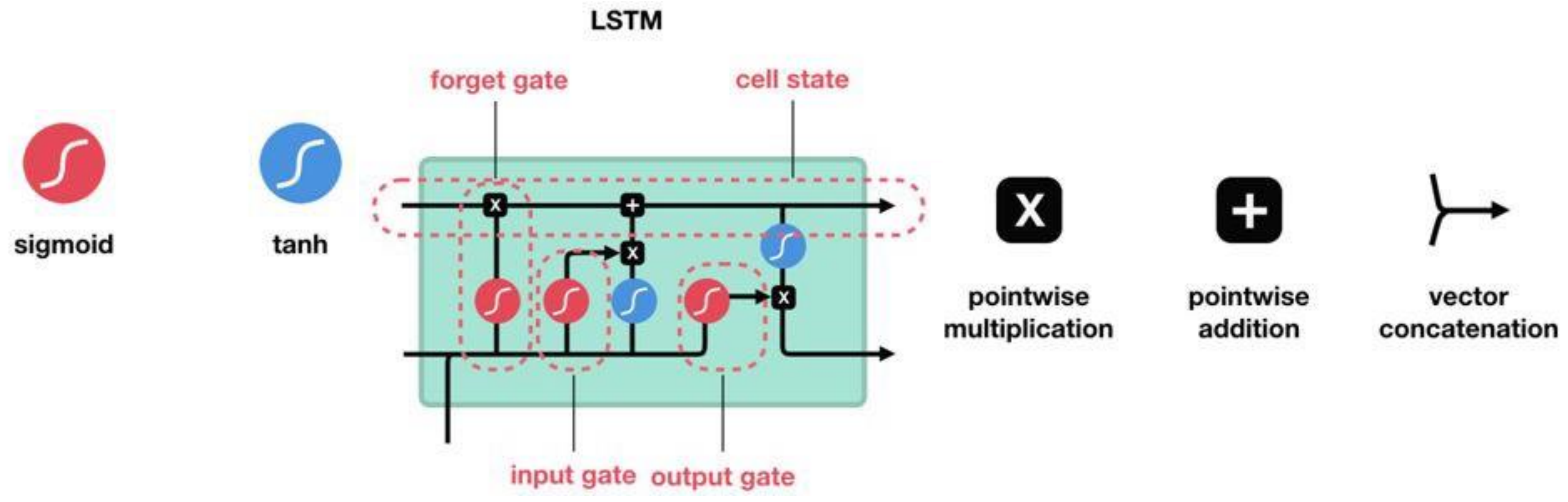
LSTM (Long Short Term Memory)

- LSTM (Long Short Term Memory) networks improve on this simple transformation and introduces additional gates and a cell state, such that it fundamentally addresses the problem of keeping or resetting context, across sentences and regardless of the distance between such context resets.

- There are variants of LSTMs including GRUs that utilize the gates in different manners to address the problem of long term dependencies.

Cell State

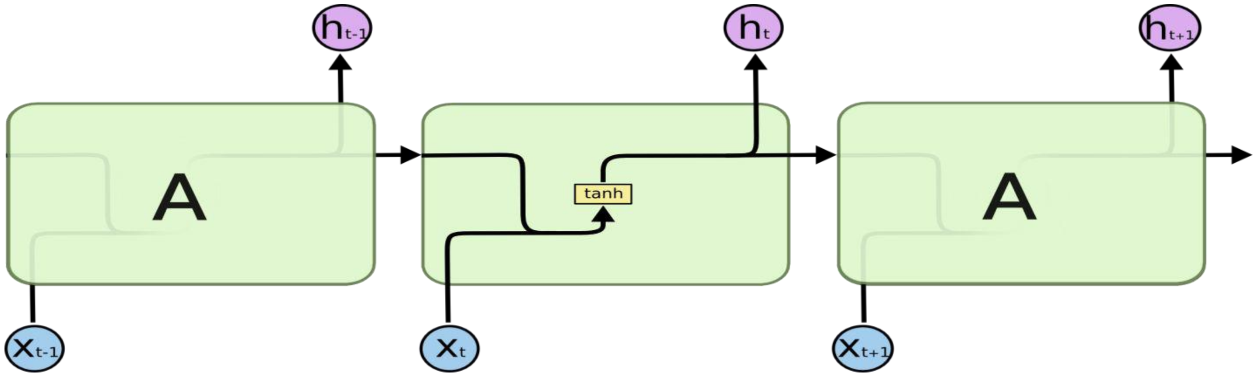
- Unlike the digital storage on computers, however, these gates are analog, implemented with element-wise multiplication by sigmoids, which are all in the range of 0-1.
- Analog has the advantage over digital of being differentiable, and therefore suitable for backpropagation.



LSTMs

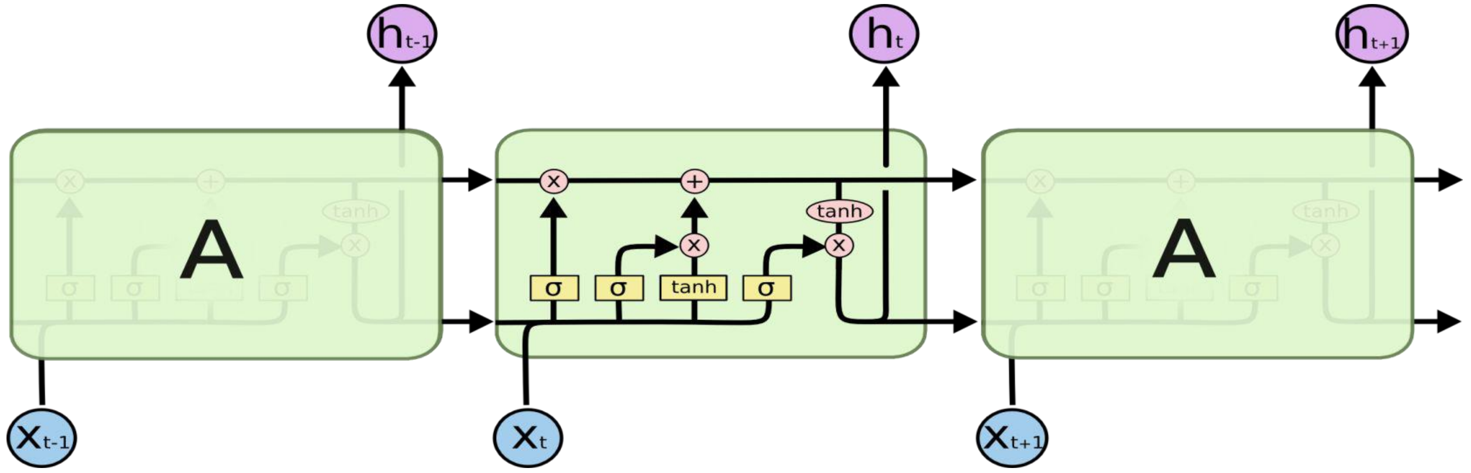
- The cells learn when to allow data to enter, leave or be deleted through the iterative process of making guesses, backpropagating error, and adjusting weights via gradient descent.
- LSTMs can selectively remember or forget things,composed of a **cell**, an **input gate**, an **output gate** and a **forget gate**.
- The information at a particular cell state has three different dependencies.
- *The previous cell state*
- *The previous hidden state*
- *The input at the current time step*
- **Addition, modification or removal** of information as it flows through the different layers, thus able to *forget* and *remember* things selectively.

- In standard RNNs:



The repeating module in an LSTM

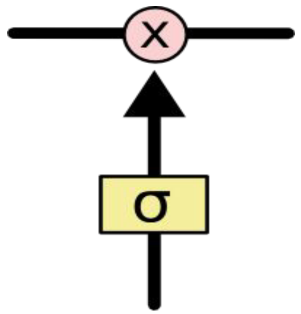
- LSTMs mechanisms: Forget, Input and Output.



•A typical LSTM network is comprised of different memory blocks called **cells,** responsible for remembering things**.**

•The expression *long short-term* refers to the fact that LSTM is a model for the *short-term memory* which can last for a *long* period of time.

Long short-term memory



- The sigmoid layer outputs values 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!”
- The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called **gates**.

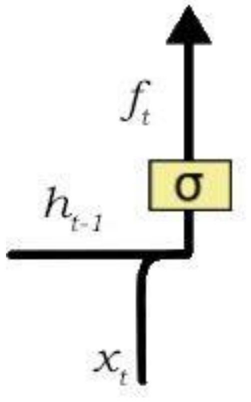
Forget Gate

- An LSTM has three of these gates, to protect and control the cell state.
- Gates are a way to let information through.

Forget Gate:

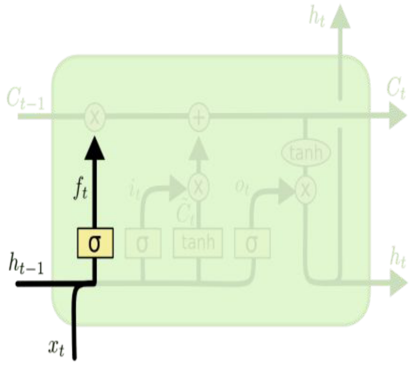
An LSTM is fed in, the following sentence:

Mohan is a nice person. Tapas is an evil.



A forget gate is responsible for removing less or no longer required information from the cell state, via multiplication of a filter.

- The first step in 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 [0, 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."

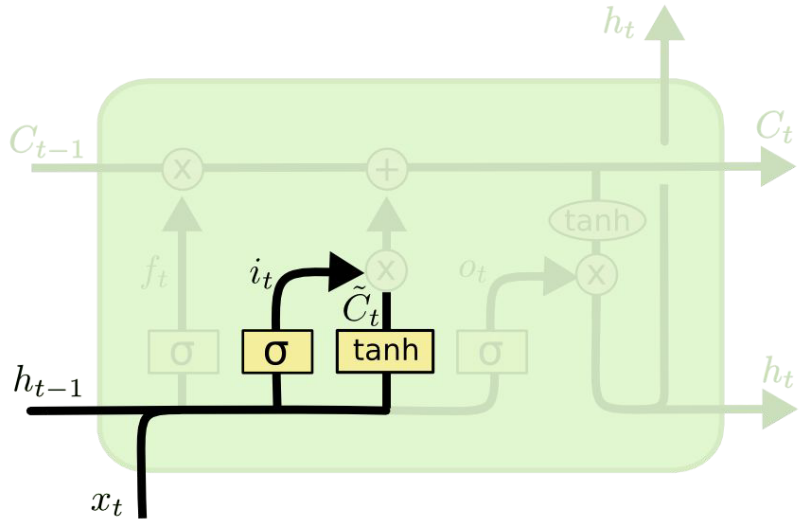


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$$

Input Gate

- 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, 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.

Input Gate

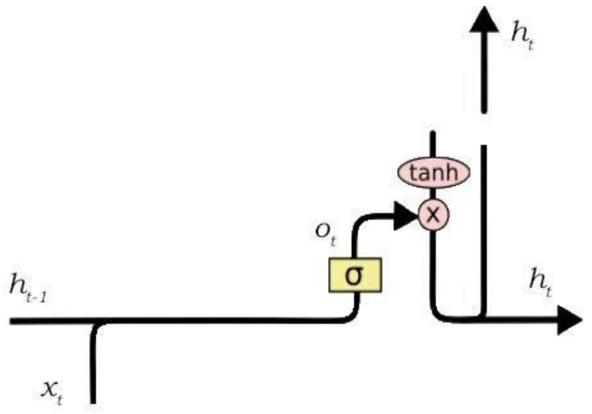


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

- It’s now time to update the old cell state, C_{t-1}, into the new cell state C_t.

Output Gate

- Its job is to selecting useful information from the current cell state and showing it out as an output via the output gate.

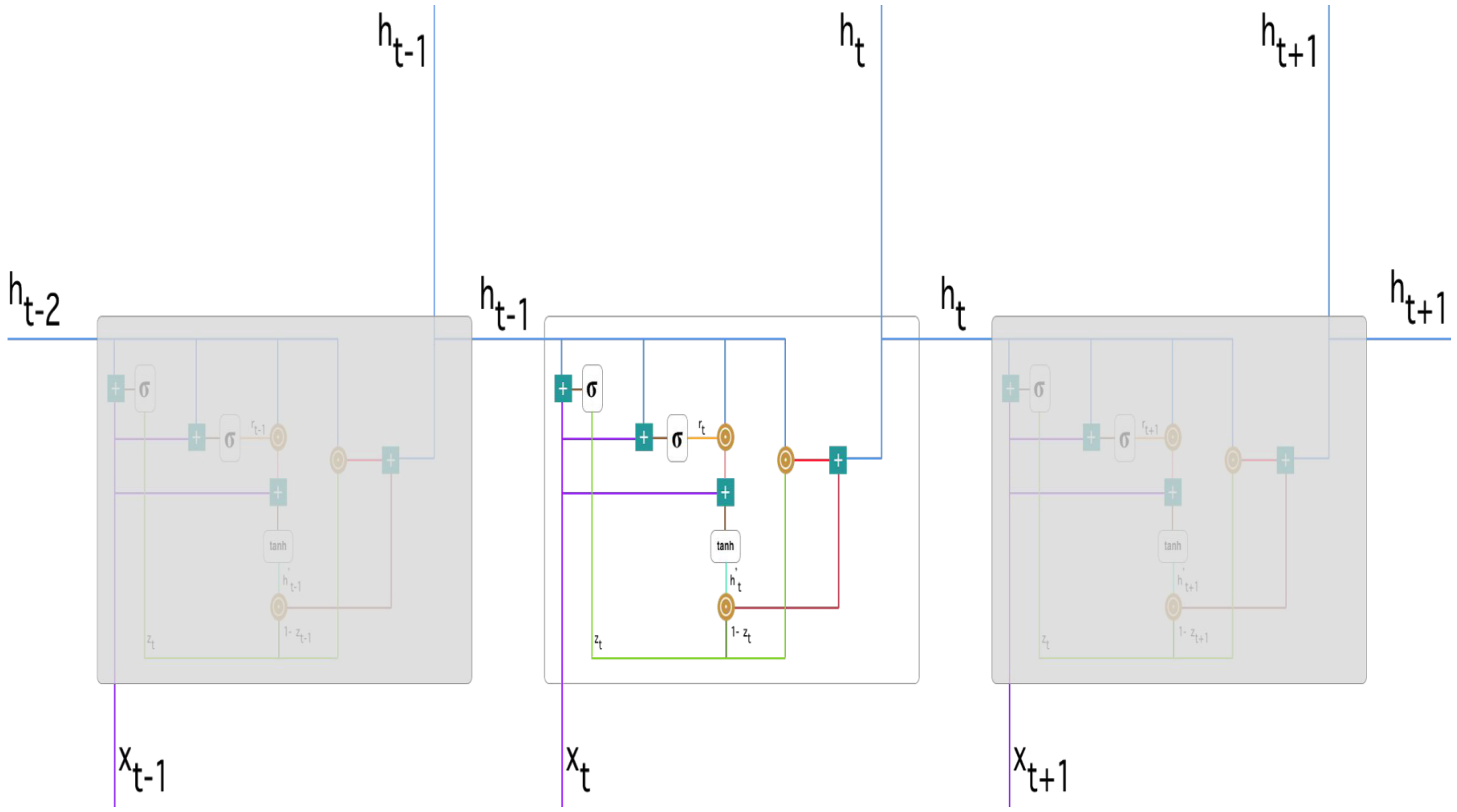


- Creating a vector after applying **tanh** function to the cell state, thereby scaling the values to the range -1 to +1.
- Making a filter using the values of h_{t-1} and x_t, such that it can regulate the values that need to be output from the created vector. This filter again employs a sigmoid function.
- Multiplying the value of this regulatory filter to the vector created in step 1, and sending it out as a output and also to the hidden state of the next cell.

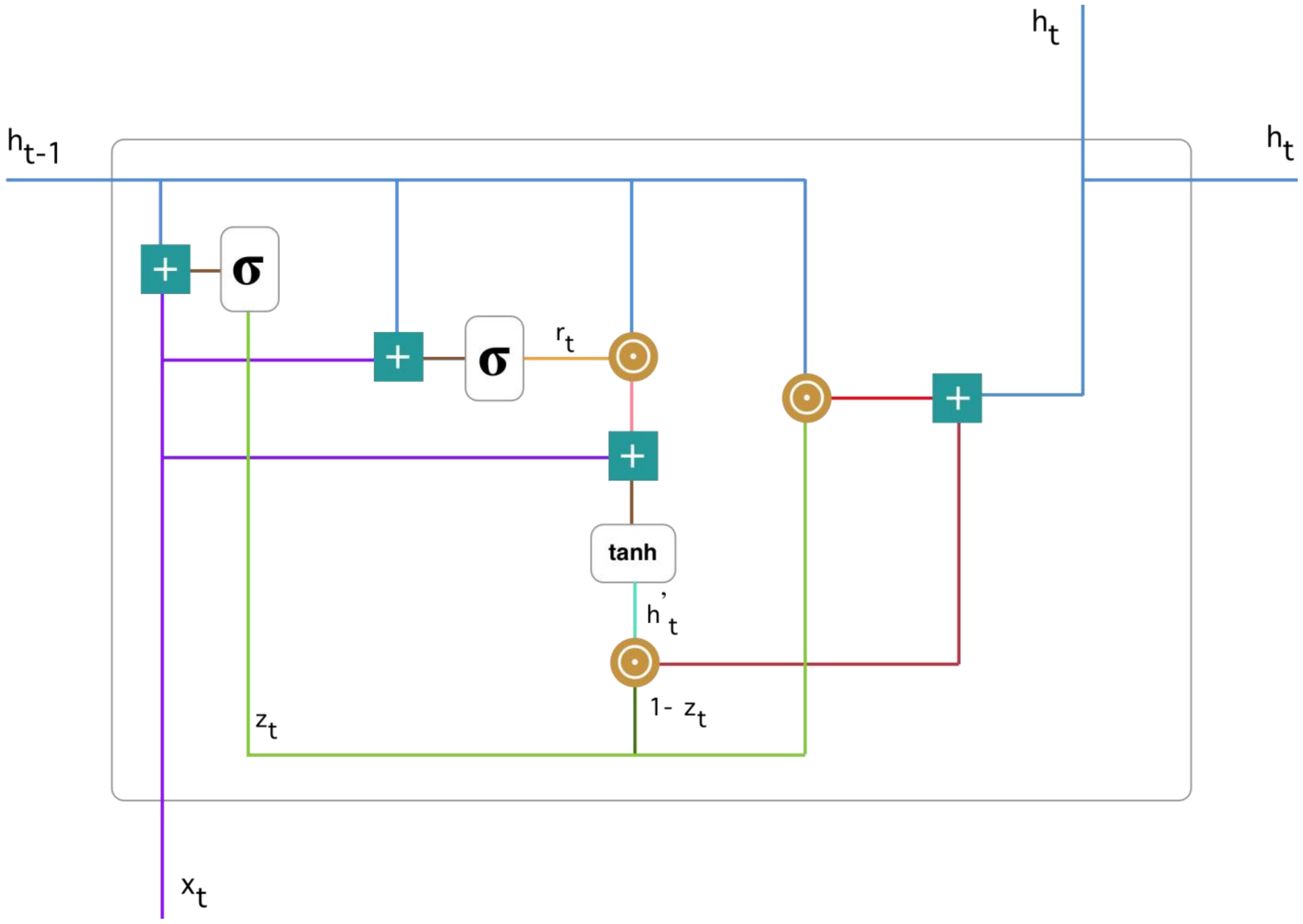
GRU

- GRU can also be considered as a variation on the LSTM
- To solve the vanishing gradient problem of a standard RNN, GRU uses, so called, **update gate and reset gate**.
- These are two vectors which decide what information should be passed to the output.
- The special thing about them is that they can be trained to keep information from long ago, without washing it through time or remove information which is irrelevant to the prediction.

A single Unit



Gated Recurrent Unit



Update Gate

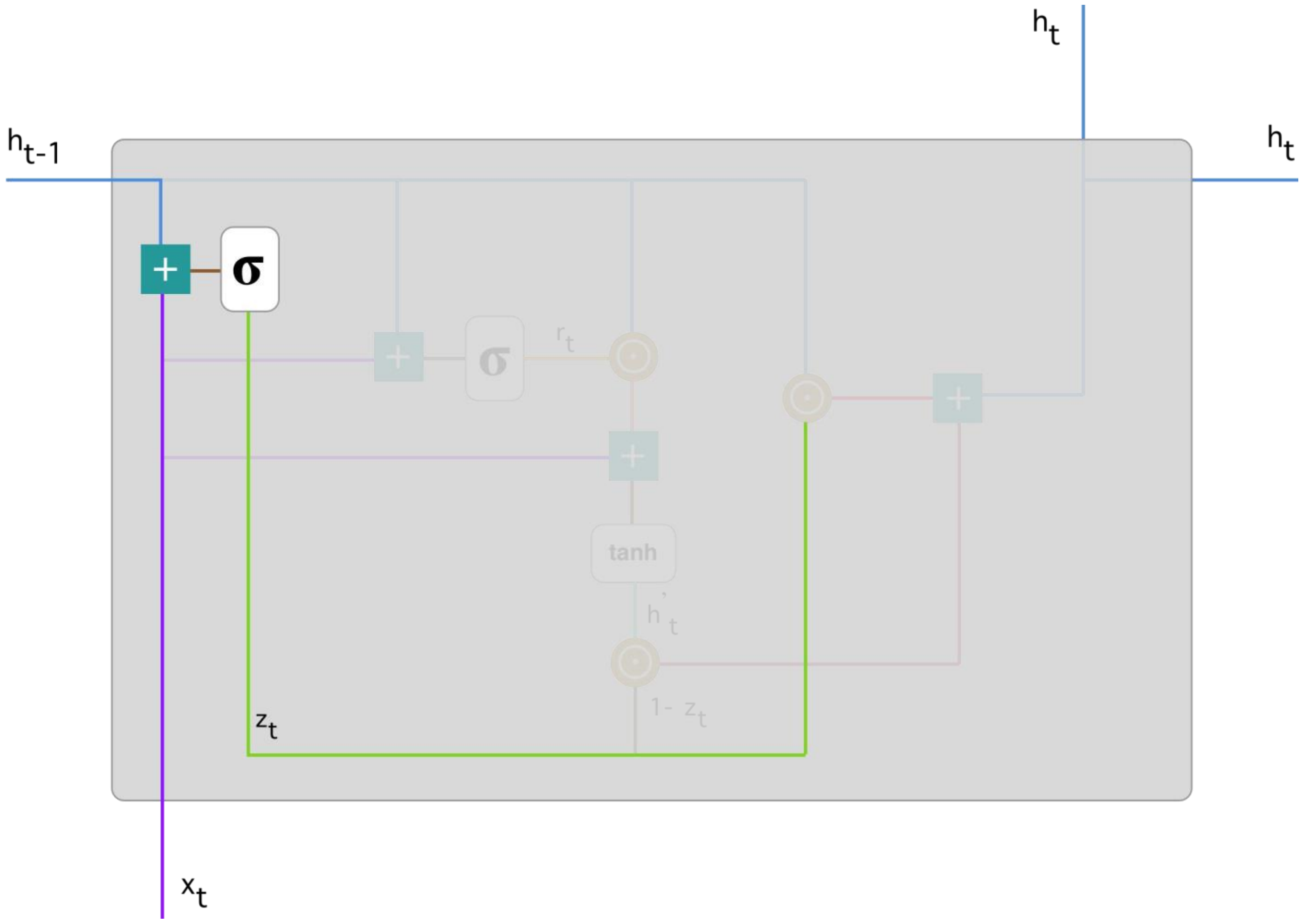
- Calculating the **update gate z_t for time step t** using the formula:

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1})$$

- The update gate helps the model to determine how much of the past information (from previous time steps) needs to be passed along to the future.

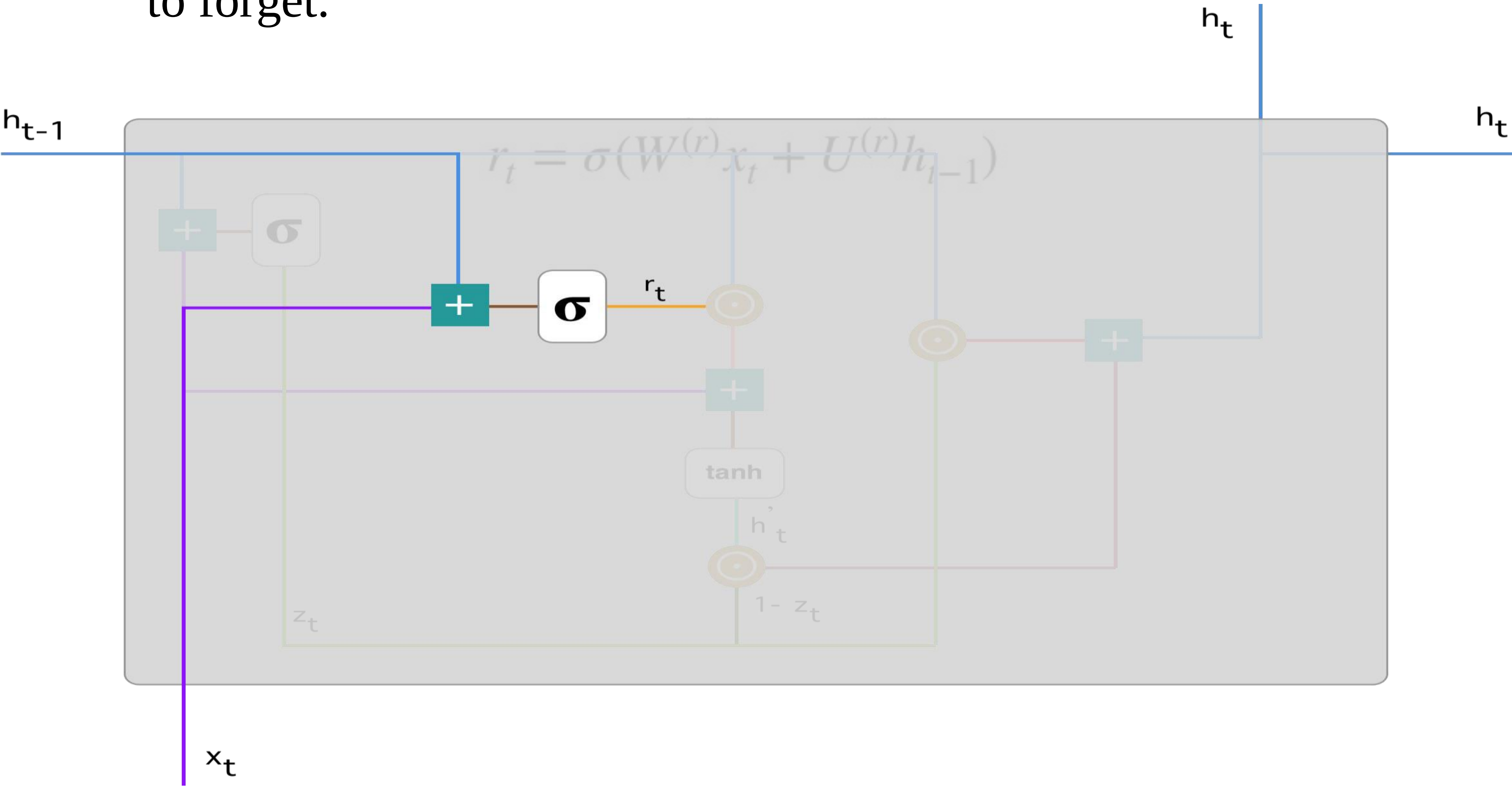
- The model can decide to copy all the information from the past and eliminate the risk of vanishing gradient problem.

Update Gate



Reset gate

- This gate is used to decide how much of the past information to forget.



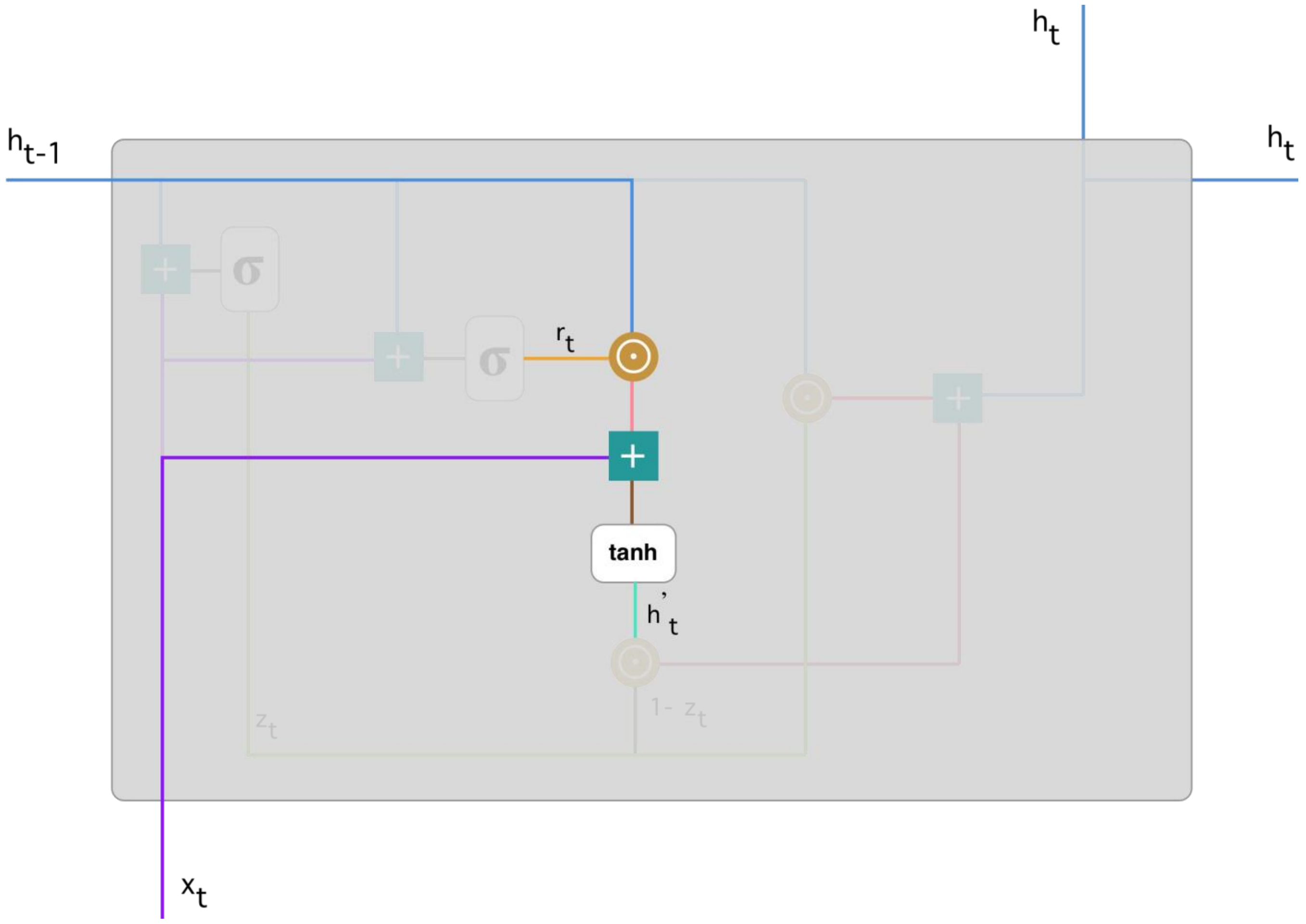
Current memory content

- New memory content which will use the reset gate to store the relevant information from the past.

$$h_t' = \tanh(Wx_t + r_t \odot Uh_{t-1})$$

Hadamard product between the reset gate r_t and $Uh_{(t-1).}$ determine what to remove from the previous time steps.

Current memory content

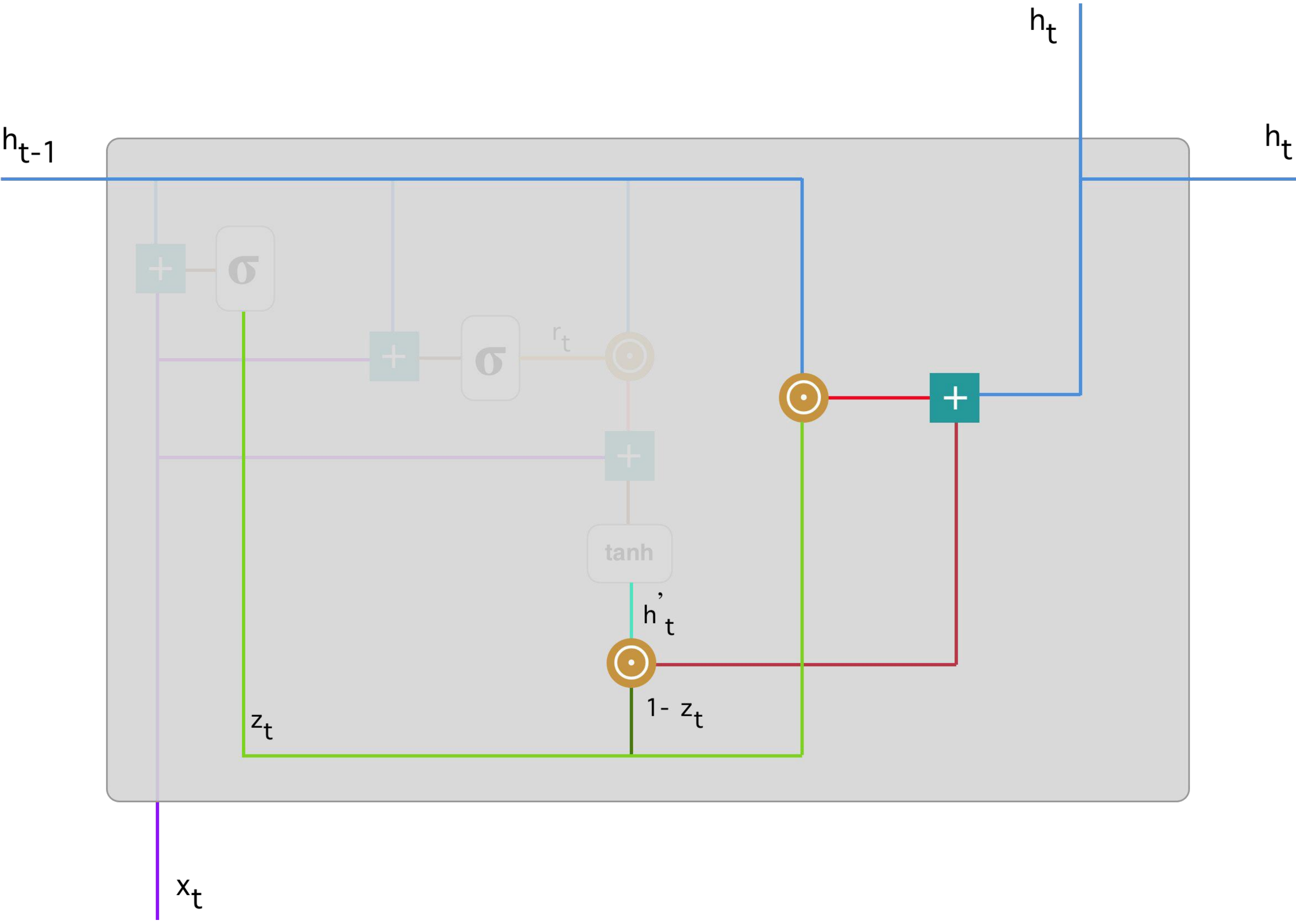


Final memory at current time step

- As a last step, the network needs to calculate h_t — vector which holds information for the current unit and passes it down to the network.

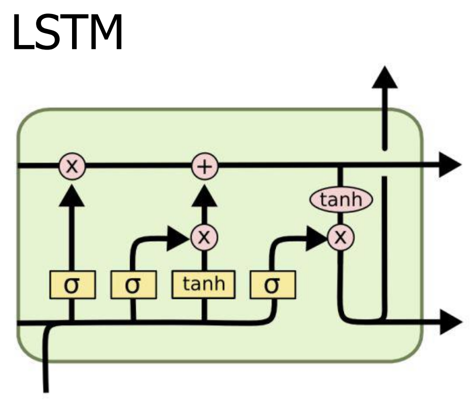
- In order to do that the update gate is needed, which determines what to collect from the current memory content — h'_t and what from the previous steps — $h_{(t-1)}$

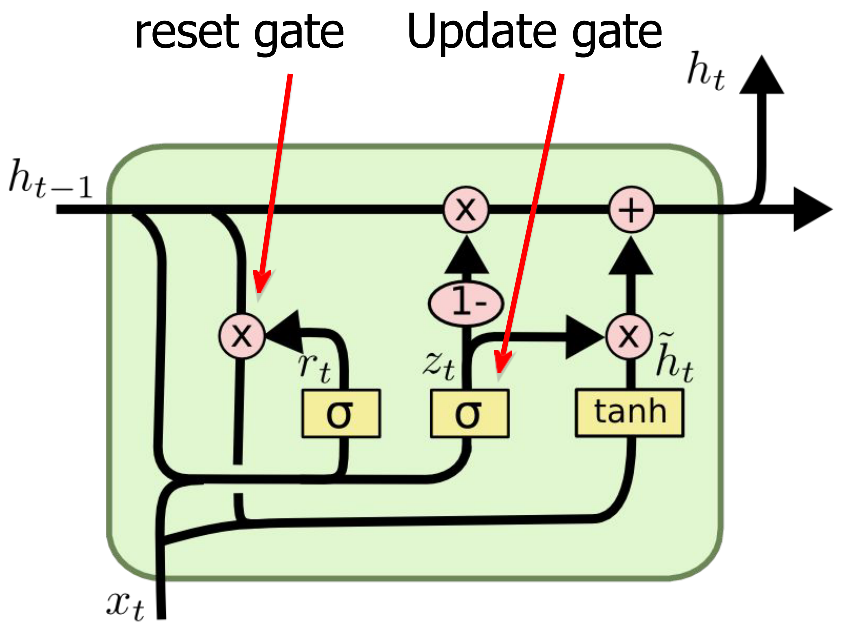
$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h'_t$$


h_t
h_{t-1}
h_t
σ
σ
r_t
tanh
h'_t
$1 - z_t$
z_t
x_t

GRU – gated recurrent unit

(more compression)





$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

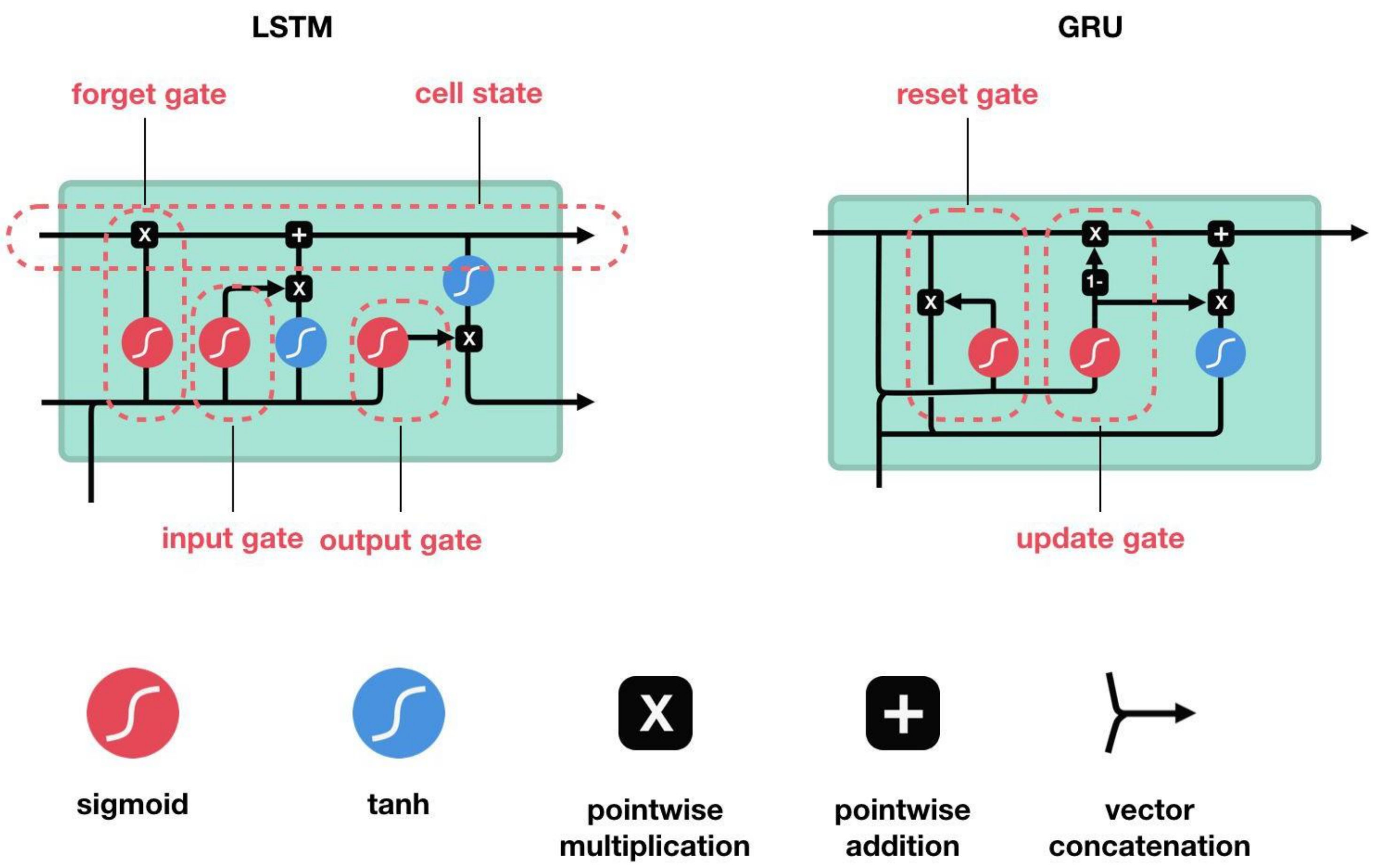
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

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

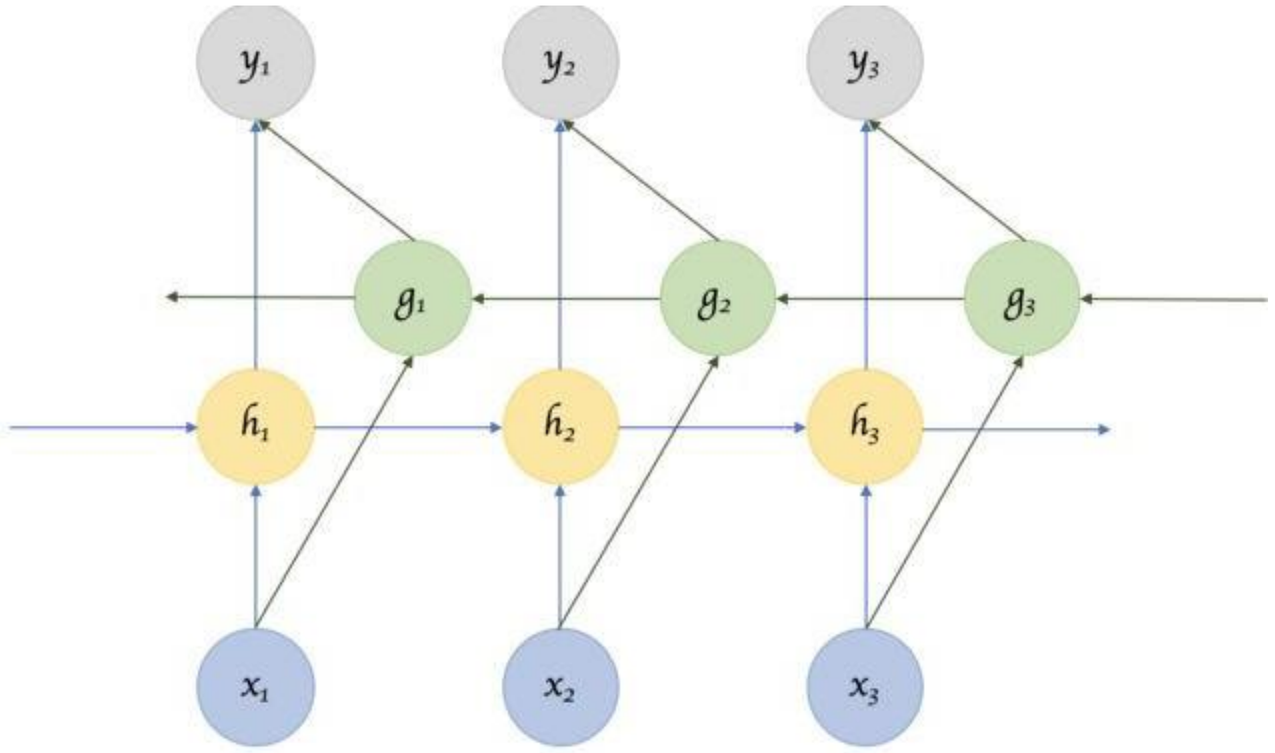
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

The model is not washing out the new input every single time but keeps the relevant information and passes it down to the next time steps of the network.

X,*: element-wise


LSTM
GRU
forget gate
cell state
input gate
output gate
reset gate
update gate
X
+
1-
sigmoid
tanh
pointwise multiplication
pointwise addition
vector concatenation

Variants of RNNs



Bidirectional (BRNN)

Bidirectional (BRNN)

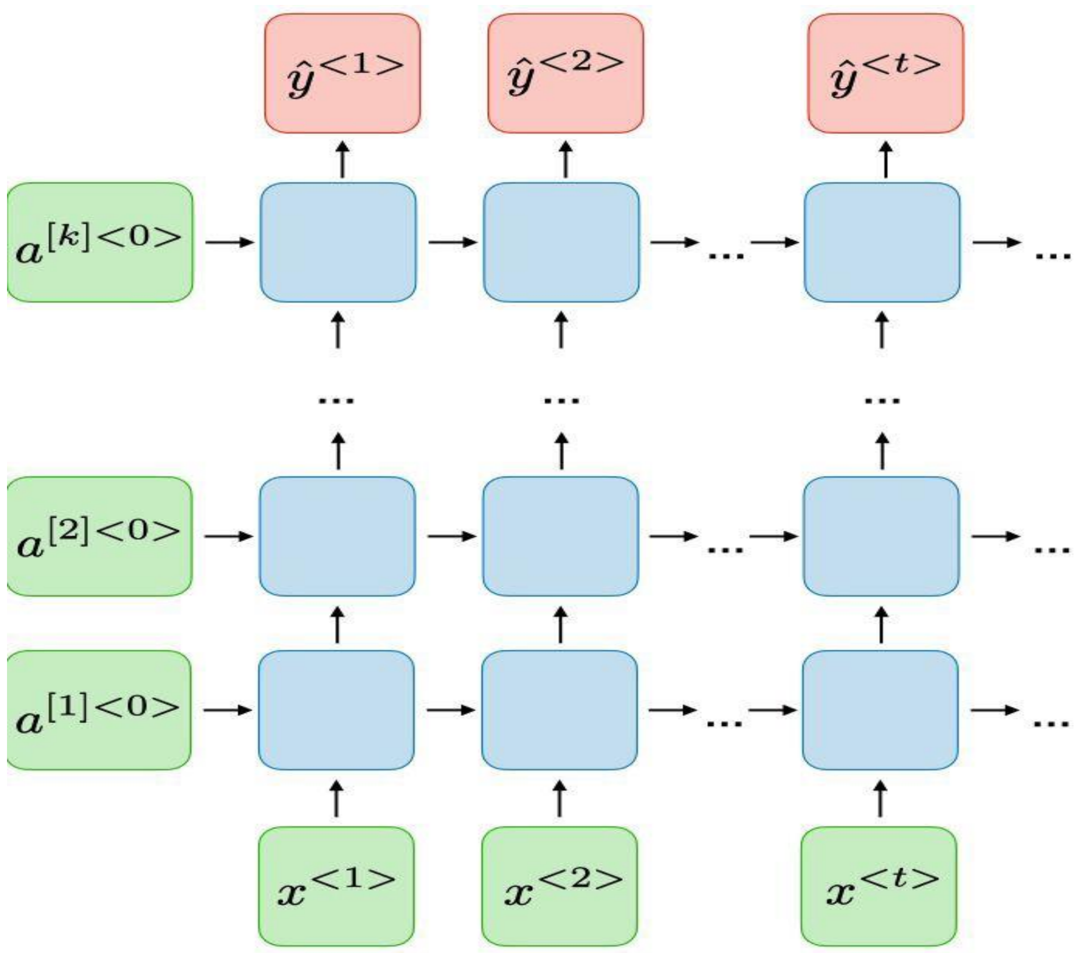
- Sometimes it’s not just about learning from the past to predict the future, but we also need to look into the future to fix the past.
- In speech recognition and handwriting recognition tasks, where there could be considerable ambiguity given just one part of the input.
- We often need to know what’s coming next to better understand the context and detect the present.
- The obvious challenge of how much into the future we need to look into, because if we have to wait to see all inputs then the entire operation will become costly.

Word Embeddings

- **Word2vec** is a framework aimed at learning word embeddings by estimating the likelihood that a given word is surrounded by other words.
- Popular models include skip-gram, negative sampling and CBOW.
- **Skip-gram:** The skip-gram word2vec model is a supervised learning task that learns word embeddings by assessing the likelihood of any given target word *t* happening with a context word *c*.

- ***t*-SNE** (*t*-distributed Stochastic Neighbor Embedding) is a technique aimed at reducing high-dimensional embeddings into a lower dimensional space.
- In practice, it is commonly used to visualize word vectors in the 2D space.

Deep (DRNN)

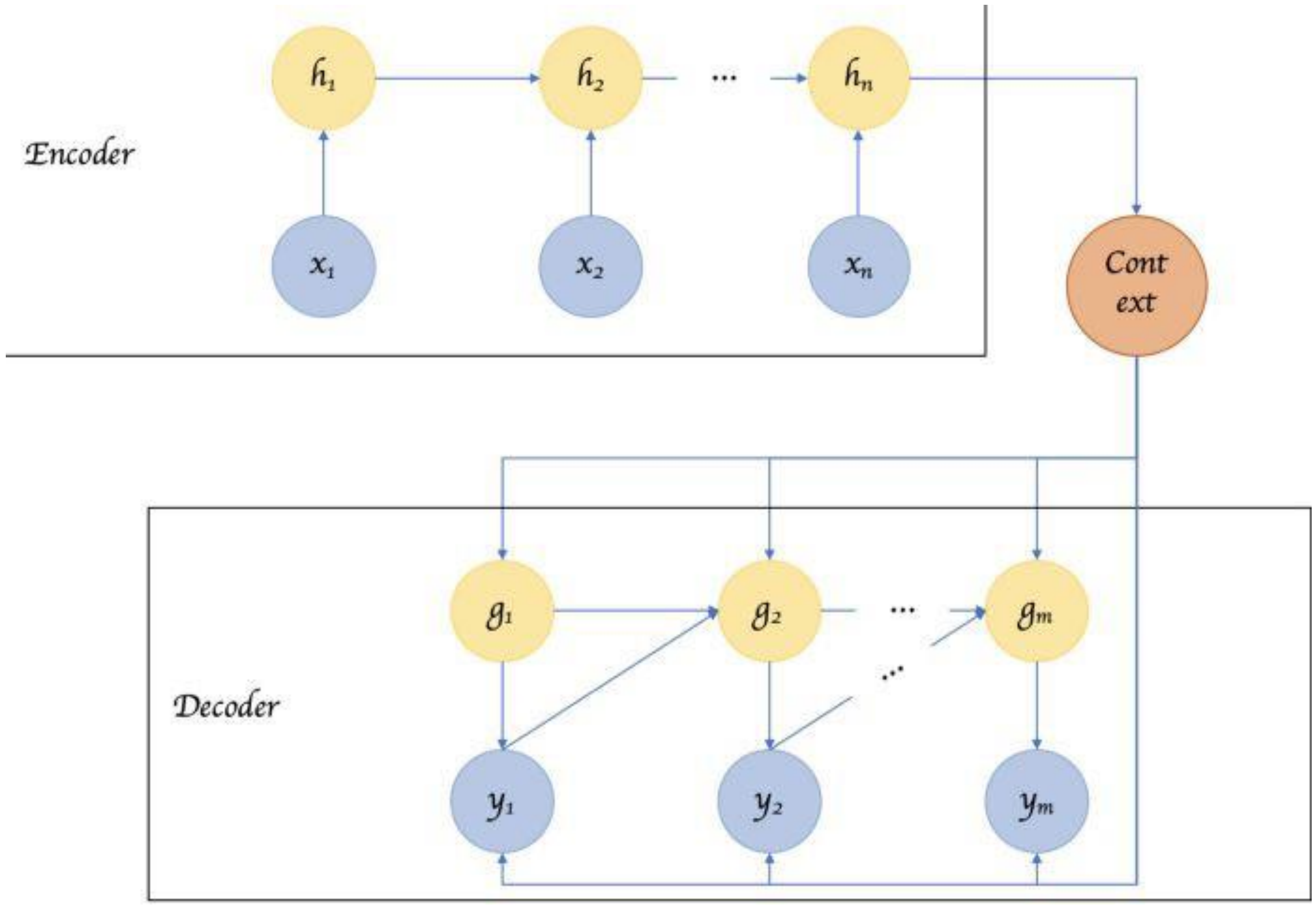


Attention model

- This model allows an RNN to pay attention to specific parts of the input that is considered as being important, which improves the performance of the resulting model in practice.

Encoder Decoder Sequence to Sequence RNNs

- The basic idea is that there are two RNNs, one an encoder that keeps updating its hidden state and produces a final single “Context” output.
- This is then fed to the decoder, which translates this context to a sequence of outputs.
- The length of the input sequence and the length of the output sequence need not necessarily be the same.


Encoder
h_1
h_2
h_n
x_1
x_2
x_n
Cont ext
Decoder
g_1
g_2
g_m
y_1
y_2
y_m

RNNs in NLP: Language Modeling and Generating Text

1. Given a sequence of words we want to predict the probability of each word given the previous words.
2. Language Models allow us to measure how likely a sentence is, an important input for Machine Translation.
3. Language Modeling being able to predict the next word as a *generative* model, which allows us to generate new text by sampling from the output probabilities.

Machine Translation

- Machine Translation is similar to language modeling in that our input is a sequence of words in our source language (e.g. German).
- We want to output a sequence of words in our target language (e.g. English).
- A key difference is that our output only starts after we have seen the complete input, because the first word of our translated sentences may require information captured from the complete input sequence.

THANK YOU