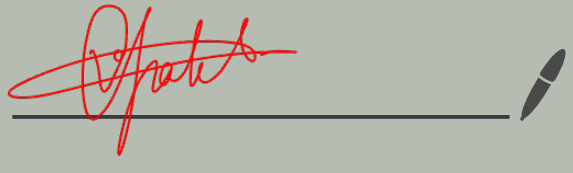
RNN



RNN : Recurrent Neural Network

RNN or recurrent neural networks are designed for sequential data, where outputs from previous steps feed back as inputs to capture temporal dependencies.

Why do we use RNN over ANN?

ANN treat each input independently, struggling with sequences since they can't retain prior context.

RNN's address this exact issue.

RNN's are used for time series data, text data and any data that requires past context.

Why we cannot use ANN's for certain types of data?

1) text inputs are not fixed size

2) This totally disregards the sequential information

In RNN, the data that we usually pass is in a very specific format.

[time stamps, Input features]

Main Intuition of RNN

Forward Propagation

In forward propagation, an RNN takes an input sequence and step by step computes hidden states and outputs using the current weights (no learning happens here, just prediction)

At each time step t the RNN

- reads the input x_t.
- Combines it with hidden state h_{t-1}
- Produces a new hidden state h_t and an output y_t.

So, lets say we have 5 rows of text data. The first being "Hello, how are you" → you convert this to vector first

[1000], [0100], [0010], [0001]

Now, we pass each of this input for one given row one by one.

So, firstly we take one row

text	y
Hello, how are you	
How is your day?	
I am Vraj	
Good Morning	

Now, we have these 4 interpretations for each word

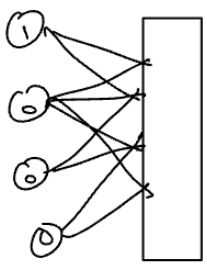
[1000] [0100] [0010] [0001]

t = 1

We take each of this one by one

lets firstly take [1 000]

This value will be passed to each perceptron



And it will multiply with the weight and add bias to produce the output O_1

t = 2

Now, we do the same process with [0100] where we multiply each of this values with the weights and add bias But, now we will also have the output of O_1 (which is 1D vector) and multiply O_1 with hidden weights as well. and after that we add the new O_1 output and the [0100]'s output together to get O_2.

This Process continues until we have the last value of our row's embedding ([0001]) and it finally gives us the output by applying a sigmoid at the final perceptron

Note here that when it's t=1, we do not have any previous's hidden layer weights because it t=1, so there was nothing before this, and here just to keep the convention, we take a pre initialized vector with either random numbers or zero vector.

Types of RNN

The main type of rnn's are usually grouped in two ways: by architecture pattern (how sequences map from input to output) and by cell variant (vanilla, LSTM, GRU, etc.)

By Input-Output Mapping

One-to-one: Single input, single output, no real sequence processing (essentially a standard feedforward net). Used as a conceptual baseline.

One-to-many: Single input, sequence output. Example: give an image, generate a caption word by word (image captioning).

Many-to-one: Sequence input, single output. Example: sentiment analysis over a sentence, spam vs. not-spam for an email.

Many-to-many (same length): Sequence in, same-length sequence out at each time step. Example: POS tagging, per-timestep time-series forecasting.

Many-to-many (different length, encoder-decoder): Input and output sequence lengths differ; implemented with encoder-decoder RNNs (often with attention) for tasks like machine translation or speech-to-text.

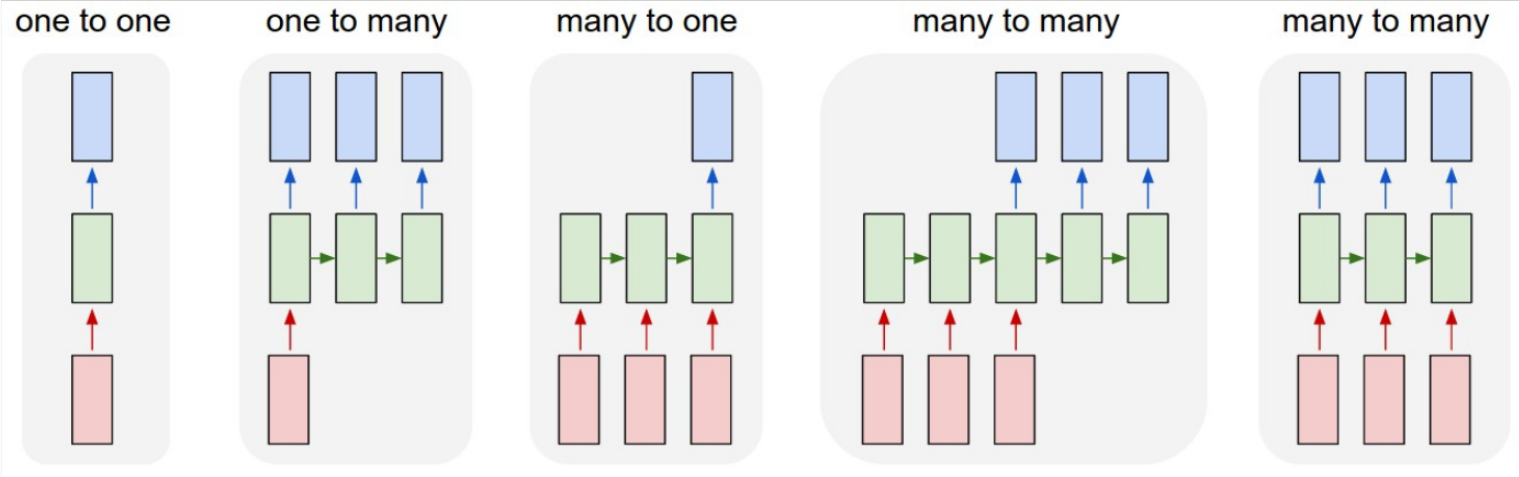
By RNN Cell Architecture

Vanilla (simple) RNN: Basic recurrent cell with a simple state update (e.g., Elman network). Easy to implement but suffers badly from vanishing/exploding gradients on long sequences.

LSTM (Long Short-Term Memory): Adds input, forget, and output gates plus a cell state to handle long-term dependencies and mitigate vanishing gradients; widely used in NLP, speech, and time-series.

GRU (Gated Recurrent Unit): Simplified gated architecture with fewer gates than LSTM (typically update and reset), often similar performance with faster training.

Bidirectional RNN (BiRNN/BiLSTM/BiGRU): Runs one RNN forward in time and another backward, then combines their states, so each timestep sees past and future context (useful in tagging, sequence labeling).



Back Propagation through Time (BPTT)

lets take a normal RNN (many to one)

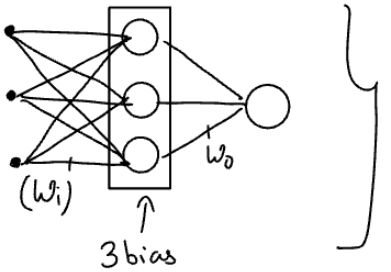
data

			output
hey	hello	hello	1
hello	hi	hey	1
hi	hi	hey	0

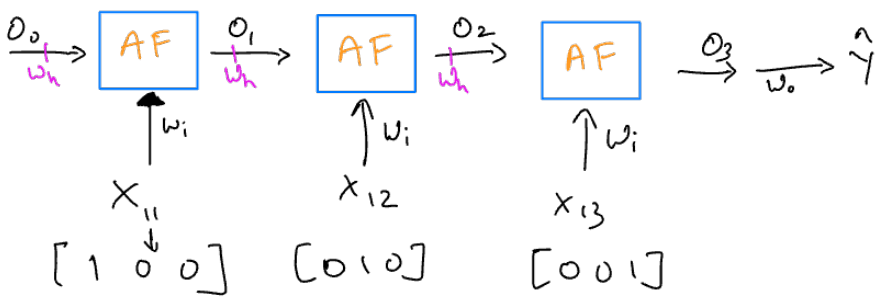
3 unique words — hey, hello, hi

[1 0 0] [0 1 0] [0 0 1]

lets just briefly see how will data forward propagate first



if we consider this whole as a block



activation functions (AF)

$O_1 = f(X_{11} W_i + O_0 W_h + b_h)$

$O_2 = f(X_{12} W_i + O_1 W_h + b_h)$

$O_3 = f(X_{13} W_i + O_2 W_h + b_h)$

$\hat{y} = \sigma(O_3 W_o)$

W_o — Output weight

We have to understand that there are total of only two biases here. b_h and b_o

b_o is for the whole raw level while b_h is for that recursive layer.

Here there are two layers. That one recursive layer and one is output layer

Now lets go to the back propagation part

Once we forward propagate, we calculate the loss and using gradient descent we back propagate.

$$L = -y_i \log \hat{y}_i - (1-y_i) \log (1-\hat{y}_i)$$

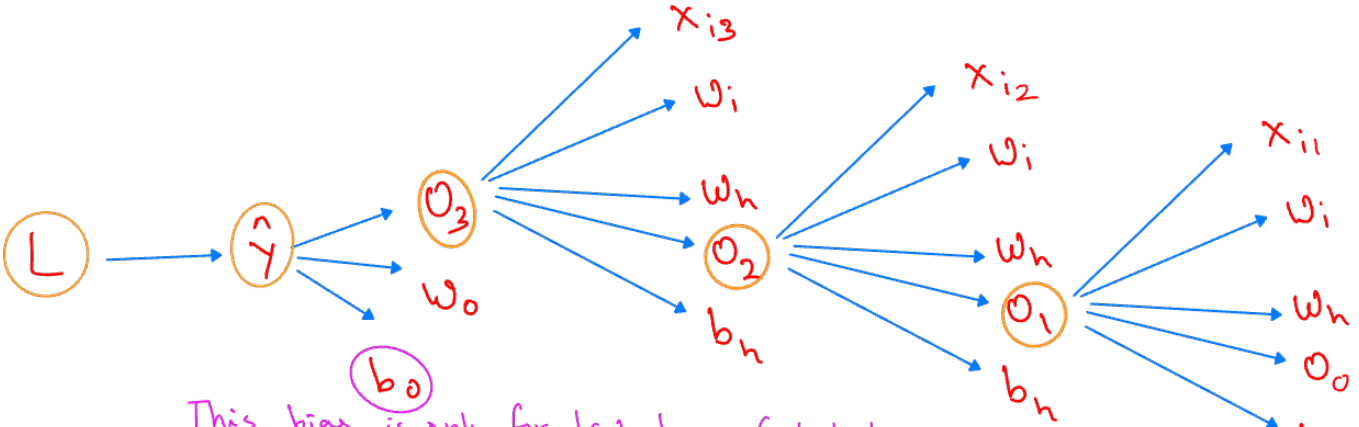
Now, we know that in order to find the minimum loss, we need to reduce these three terms :

$$w_i \; , \; w_h \; , \; w_o \; , \; b_h$$

$$w_i = w_i - \frac{\partial L}{\partial w_i} \qquad w_o = w_o - \frac{\partial L}{\partial w_o} \qquad b_o = b_o - \frac{\partial L}{\partial b_o}$$

$$w_h = w_h - \frac{\partial L}{\partial w_h} \qquad b_h = b_h - \frac{\partial L}{\partial b_h}$$

lets draw the dependency graph

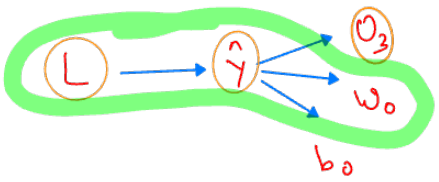


This bias is only for last layer (output layer) (b_h is different, so not considered)

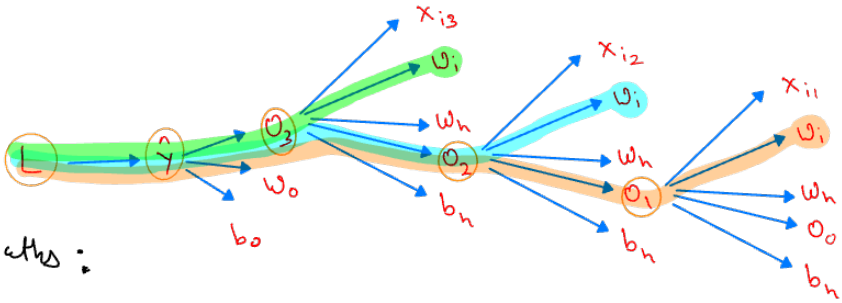
lets derivate each of the above mentioned derivation equation ....

① $$w_o = w_o - \frac{\partial L}{\partial w_o}$$

$$\boxed{\frac{\partial L}{\partial w_o} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_o}}$$



(2) $w_i = w_i - \frac{\partial L}{\partial w_i}$



We notice it will take 3 paths:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial O_3} \frac{\partial O_3}{\partial w_i} + \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial O_3} \frac{\partial O_3}{\partial O_2} \frac{\partial O_2}{\partial w_i} + \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial O_3} \frac{\partial O_3}{\partial O_2} \frac{\partial O_2}{\partial O_1} \frac{\partial O_1}{\partial w_i}$$

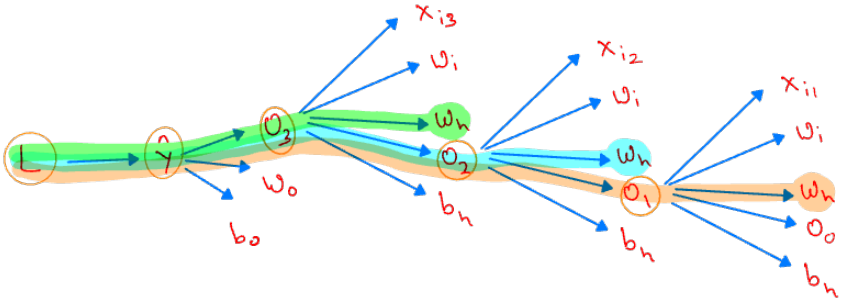
Now, ofc we cannot calculate this much for just one term, so we compress it:

$$\frac{\partial L}{\partial w_i} = \sum_{j=0}^{3} \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial O_j} \frac{\partial O_j}{\partial w_i}$$

Therefore, we finally get:

$$\boxed{\frac{\partial L}{\partial w_i} = \sum_{j=0}^{n} \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial O_j} \frac{\partial O_j}{\partial w_i}}$$

(3) $w_h = w_h - \frac{\partial L}{\partial w_h}$



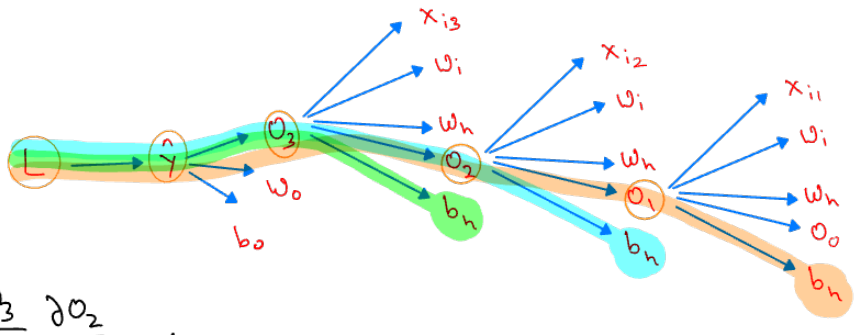
Again 3 possibilities

$$\frac{\partial L}{\partial w_h} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial O_3} \frac{\partial O_3}{\partial w_h} + \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial O_3} \frac{\partial O_3}{\partial O_2} \frac{\partial O_2}{\partial w_h} + \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial O_3} \frac{\partial O_3}{\partial O_2} \frac{\partial O_2}{\partial O_1} \frac{\partial O_1}{\partial w_h}$$

Generalizing the equation:

$$\frac{\partial L}{\partial w_h} = \sum_{j=1}^{n} \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial O_j} \frac{\partial O_j}{\partial w_h}$$

④ $b_h = b_h - \frac{\partial L}{\partial b_h}$

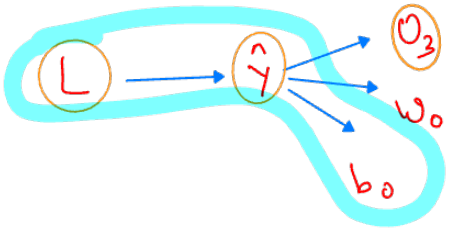


$$\frac{\partial L}{\partial b_h} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial o_3} \frac{\partial o_3}{\partial b_h} + \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial o_3} \frac{\partial o_3}{\partial o_2} \frac{\partial o_2}{\partial b_h} + \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial o_3} \frac{\partial o_3}{\partial o_2} \frac{\partial o_2}{\partial o_1} \frac{\partial o_1}{\partial b_h}$$

Therefore

$$\boxed{\frac{\partial L}{\partial b_h} = \sum_{j=1}^{n} \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial o_j} \frac{\partial o_j}{\partial b_h}}$$

⑤ $b_0 = b_0 - \frac{\partial L}{\partial b_0}$



$$\boxed{\frac{\partial L}{\partial b_0} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b_0}}$$

Therefore, we get these equations

$$\boxed{\begin{aligned}
w_i &= w_i - \frac{\partial L}{\partial w_i} = w_i - \sum_{j=1}^{n} \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial o_j} \frac{\partial o_j}{\partial w_i} \\
w_h &= w_h - \frac{\partial L}{\partial w_h} = w_h - \sum_{j=1}^{n} \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial o_j} \frac{\partial o_j}{\partial w_h} \\
w_0 &= w_0 - \frac{\partial L}{\partial w_0} = w_0 - \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_0} \\
b_h &= b_h - \frac{\partial L}{\partial b_h} = b_h - \sum_{j=1}^{n} \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial o_j} \frac{\partial o_j}{\partial b_h} \\
b_0 &= b_0 - \frac{\partial L}{\partial b_0} = b_0 - \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b_0}
\end{aligned}}$$

Problems with RNN

There are two major problems with RNN's

1) Vanishing Gradients (long term dependency)
2) Unstable training (Exploding Gradients)

① long term Dependency

lets say we have a very long input scentence and the final answer (word to be predicted) depends on the first two words of input.

Now, as we keep on having longer inputs, the model is not able to remember anything from the long term (first 2 words)

Therefore, while back propagating, the long term inputs will have very very less impact on the answer and short term memory will have very big impact which will lead to wrong answer.

lets say an input has 100 words

$$\frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial O_{100}} \frac{\partial O_{100}}{\partial O_{99}} \frac{\partial O_{99}}{\partial O_{98}} \cdots \cdots \frac{\partial O_2}{\partial O_1} \frac{\partial O_1}{\partial W_i}$$

Very less impact

↓

Because we have tanh function which results in values from 0 - 1.

So if we have alot of values in range 0 - 1, the overall value will be very close to 0.

So, it will have very less impact on the output

While on the other hand, if we consider the recent values, since their derivatives are not that big, the value will be bigger.

So, Recent values (short term) will contribute more!

Solutions to vanishing Gradients

- Different Activation Functions
- Better weight initialization (the hidden weight's vector)
- Using LSTM
- Skip Connections

(2) Exploding Gradients

If we change the activation function to ReLU, $\max(0, x)$, the gradients value will be very large and sometimes to infinity which will stop the model's training completely.

Solutions to Exploding Gradient

- Probably use LSTM
- Gradient Clipping.

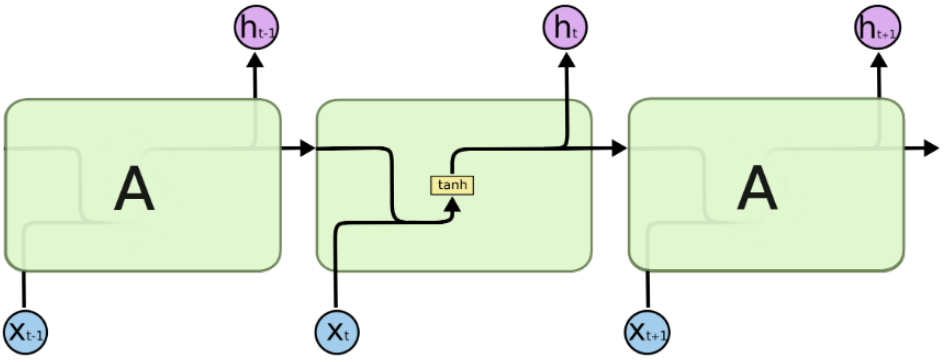
Long Short Term Memory (LSTM)

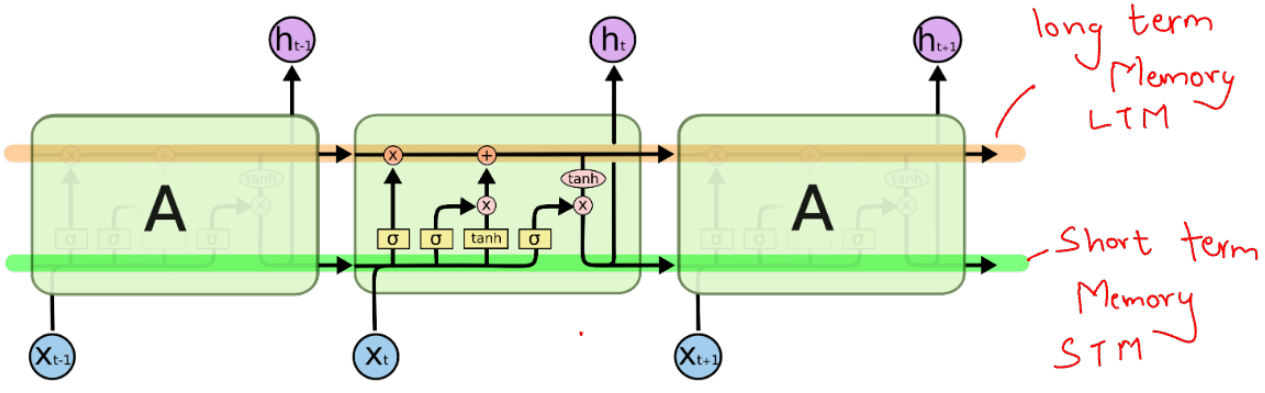
We know that there is a problem with RNN because of inability to handle long term memory.

LSTM is a special type of RNN designed to learn long term dependencies in sequential data. Introduced by Hochreiter and Schmidhuber in 1997, LSTM's are widely used in tasks like language modelling, time series prediction, speech recognition, and machine translation.

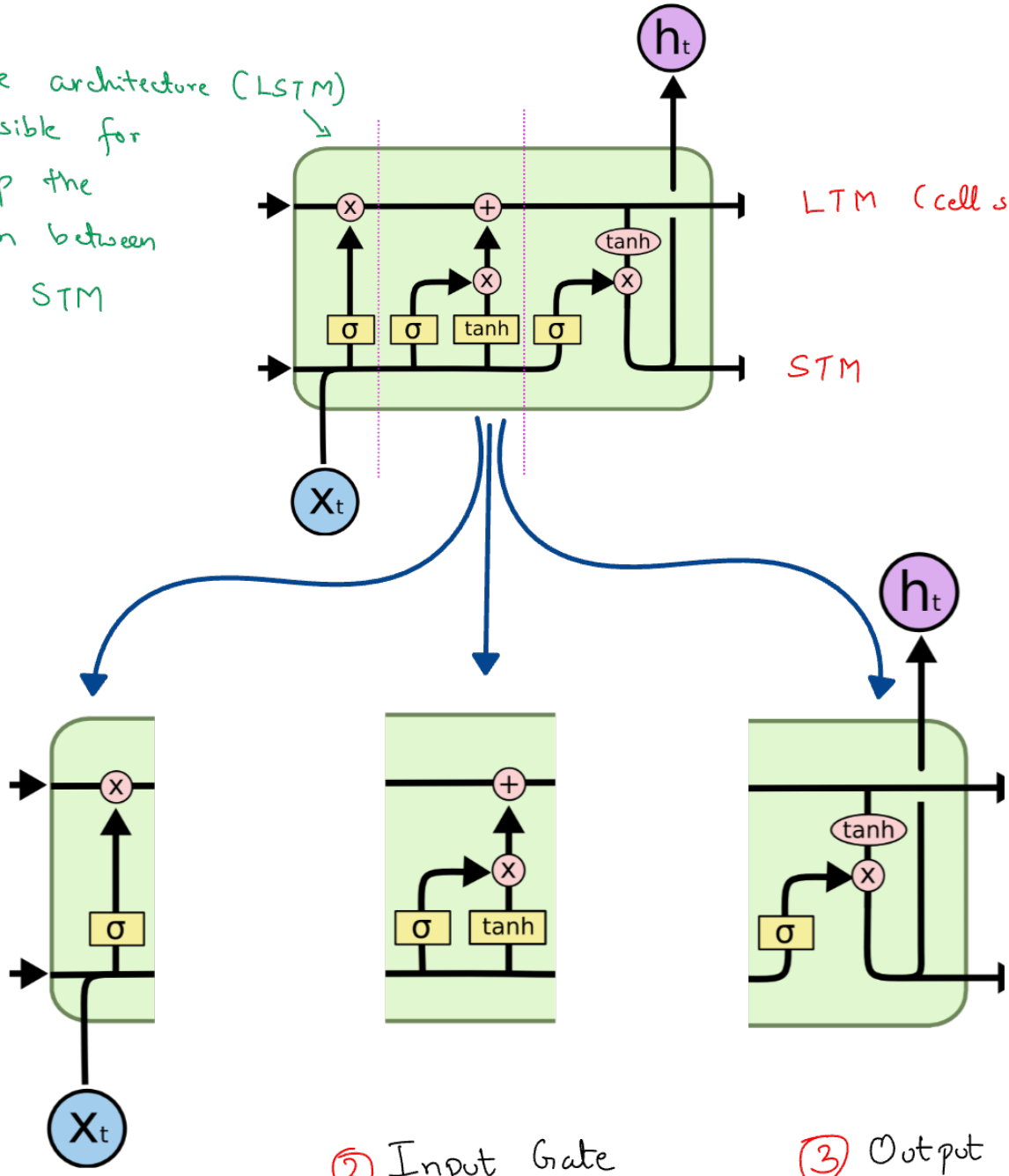
LSTM are designed specifically to solve the RNN issue where RNN are not able to retain long term context due to vanishing/exploding gradient. The key change was to introducing a "cell state" - a seperate memory highway that runs alongside the hidden state. Instead of overwriting memory at everystep like a vanilla RNN, an LSTM uses learnable gates to control what information gets added, kept or discarded. This gating mechanism allows gradients to flow more stably over long sequences, preventing relevant context across hundreds of time steps if needed.

In short: RNN's have only one state and one path whereas LSTM's have two states : hidden state and cell state





This whole architecture (LSTM) is responsible for setting up the connection between LTM and STM



① Forget Gate ② Input Gate ③ Output Gate

① Forget Gate : This gate decides what information from the previous cell state should be thrown away. It looks at the current input and the previous hidden state, and outputs a value between 0 (forget) and 1 (keep) for each piece of info in cell state.

② Input Gate : Decides what new information to write into memory. It selects which values to update and generates new candidate values to potentially add to the cell state.

③ Output Gate : Decides what to output. It filters the cell state and uses it to produce the next hidden state, which carries forward into the next time step.

Overview

An LSTM architecture takes 3 inputs and produces 2 outputs with internal processing happening in between.

Inputs:

- Previous cell state (C_{t-1}) — long term memory
- Previous hidden state (h_{t-1}) — short term memory
- Current input (X_t) — Input for the current timestamp

Outputs:

- Current cell state (C_t)
- Current hidden state (h_t)

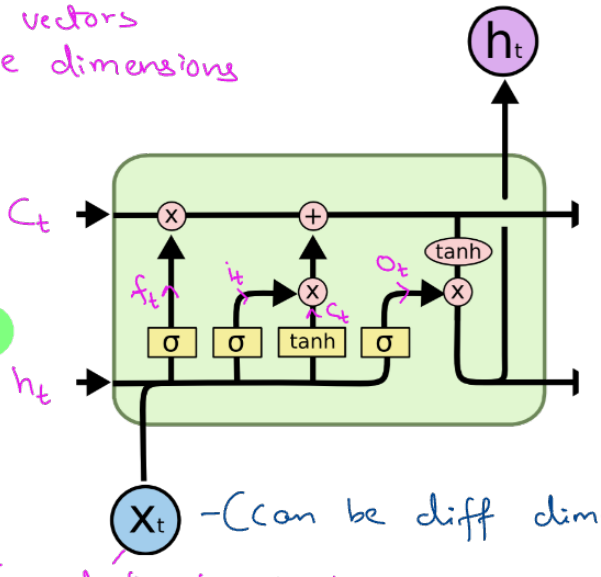
The internal processing has two main steps:

1. Update cell state (involves 2 sub steps)
   - i. Forget Gate : Removes something from the cell state
   - ii. Input Gate : Adds something to the cell state

So together, these two sub steps are simply updating the cell state.

2. Update the Hidden state
   - Output Gate : Updates the hidden state (STM) using the updated cell state

C_t, H_t : Both are vectors with same dimensions

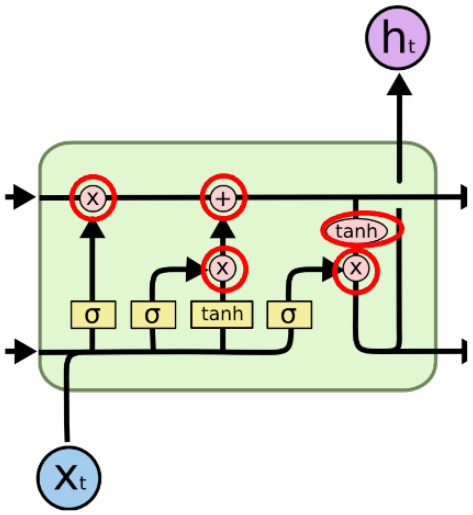


f_t, i_t, C_t, O_t are forget input output gate vectors. All have the same dimensions

No. of nodes and C_{t-1} and h_{t-1} are same dimension

X_t — (can be diff dimension)

(current timestamp input)



Pointwise Operations

⊗ – Pointwise Multiplication

$$\begin{bmatrix} 3 & 8 & 6 \\ 2 & 1 & 7 \end{bmatrix} \xrightarrow{\otimes} \begin{bmatrix} 6 & 8 & 42 \end{bmatrix}$$

⊕ – Pointwise Multiplication

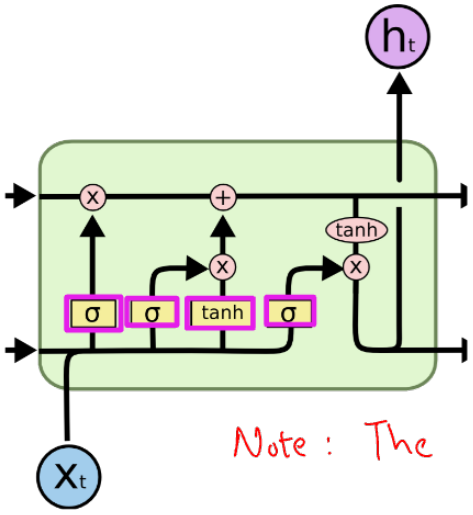
$$\begin{bmatrix} 3 & 8 & 6 \\ 2 & 1 & 7 \end{bmatrix} \xrightarrow{\oplus} \begin{bmatrix} 5 & 9 & 13 \end{bmatrix}$$

tanh – Pointwise tanh

Applying activation function element wise

$$\begin{bmatrix} 3 & 8 & 7 \end{bmatrix} \xrightarrow{\tanh} \begin{bmatrix} 0.99 & 1 & 1 \end{bmatrix}$$

Neural Network layers



These highlighted are a ANN.

σ – A neural network with σ as AF

tanh – A neural network with tanh as Af

Note: The number of units (nodes) are flexible.

The no. of nodes are decided by a hyperparameter.

But also know that the no. of nodes set by the hyperparameter will be the same for all other Neural networks across the entire block.

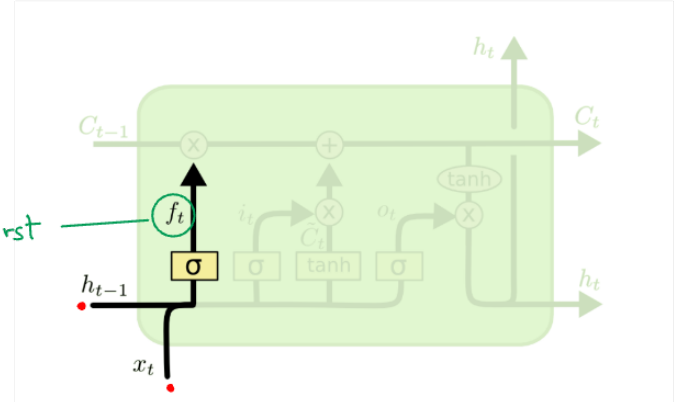
We cannot hyper tune no. of nodes individually. It will only follow the global value set for no. of nodes.

① Forget Gate

The forget gate decides what to remove from the cell state (long term memory). f_t outputs a vector of values between 0 and 1, where 0 means "completly forget" and 1 means "compeletly keep"

f_t has 2 inputs h_{t-1} and x_t

We calculate value of f_t first



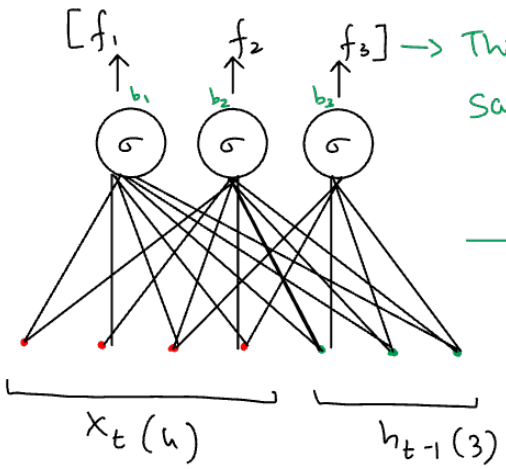
for example, lets take h_{t-1} as 3 dimensional vector : $[h_{t_1} \; h_{t_2} \; h_{t_3}]$

lets take x_t as 4 dimensional vector : $[x_{i1} \; x_{i2} \; x_{i3} \; x_{i4}]$

This vector can be different

Now, we calculate the output vector by a FCC (fully connected layer)

$n=3$ as h_{t-1} is 3 dims



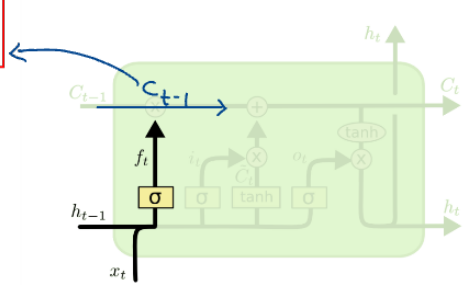
→ This f_t vector will be the same dimensions as h_{t-1} / No. of nodes (n)

⟶ f_t will have $(7 \times 3) = 21$ weights + 3 biases (b)

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

Now at the end, we remove some values from C_{t-1} (using pointwise multiplication)

$f_t \otimes C_{t-1}$

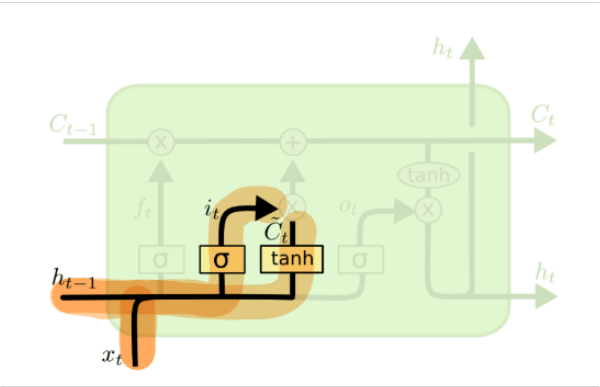


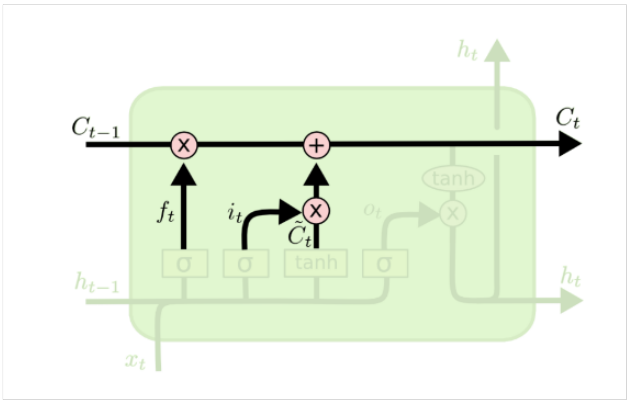
② The input Gate

The input gate decides what new information to add to the cell state (long term memory). Unlike forget one, this one has two parts working together.

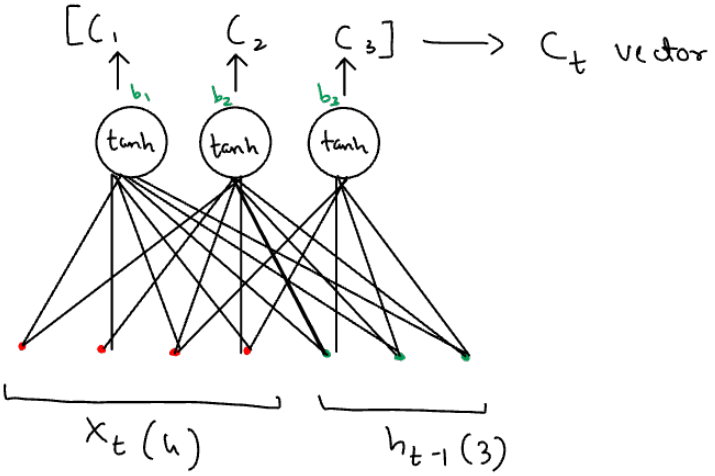
Part ① Candidate Gate (tanh): Decides what the new values actually are. Outputs values between -1 and 1.

Part ② Input Gate (Sigmoid): Decides which values to update. Outputs values between 0 and 1 — acting as filter on whats worth writing



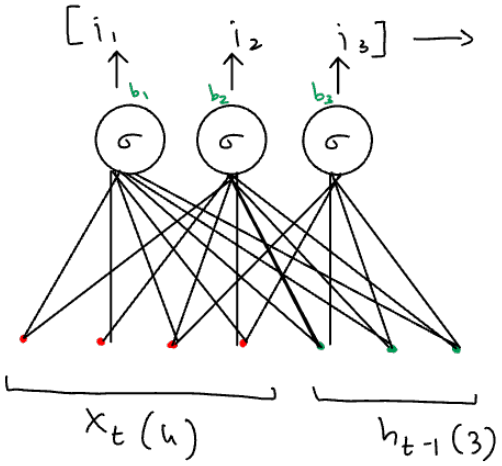


So, simply first we calculate the C_t which is candidate State where we apply a FCC on the inputs (x_t, h_{t-1}) that has tanh as Activation function



$$C'_t = \tanh\left(W_c[x_t, h_{t-1}] + b_c\right)$$

Now, once we have C'_t, we calculate it with the same process but using Sigmoid FCC



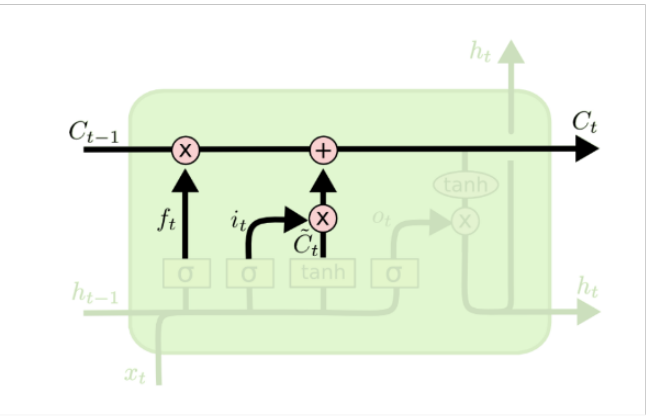
(This is also known as filter)

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

Now that we also have i_t, (i_t is called as a filter layer)

We do pointwise multiplication of i_t and C'_t where i_t acts as a filter and filters values from vector C'_t

$$\bar{C}_t = i_t \otimes C_t$$



Now, we simply again do pointwise addition of $\bar{C}_t$ and the C_{t-1} that was updated by the forget gate.

$$C_t = C_{t-1} \oplus \bar{C}_t$$

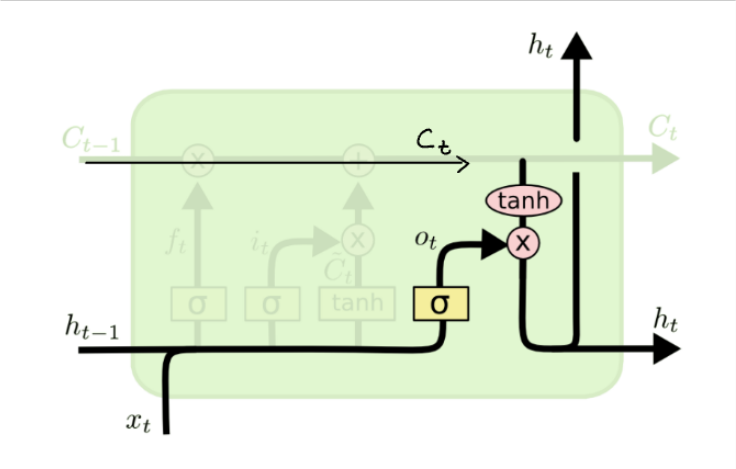
Or

$$C_t = f_t \otimes C_{t-1} \oplus i_t \otimes C'_t$$

Therefore, Think of C'_t as a draft of new information and it as the editor (filter) deciding how much of that draft makes it into long term memory C_t. Together they write the new relevant information into the cell state.

③ The Output Gate

The output gate decides what to output as the new hidden state (short term memory) based on the updated cell state (C_t).



We calculate O_t.

We use FCC with sigmoid as AF with (h_{t-1}, x_t) as input to obtain "O_t"

then we apply a FCC with tanh as AF with C_t as input which we can represent temporarily as $\tanh(C_t)$

Now we do pointwise multiplication on O_t and $\tanh(C_t)$ to get the final output h_t (hidden state)

$$h_t = \underbrace{O_t}_{\sigma(W_o[x_t, h_{t-1}] + b_o)} \otimes \underbrace{\tanh(C_t)}_{\text{same}}$$

Intuition: the cell state at this point has already been cleaned (forget gate) and updated (input gate).

The output gate's job is to decide which parts of that long term memory are relevant right now to carry forward as short term memory.

Not everything in the cell state needs to be exposed - just whats useful for the current state

Gated Recurrent Unit (GRU)

GRU are simplified variant of LSTM that combines the forget gate and input gates into a single update gate, and merges the cell state and the hidden state – making it computationally more efficient while achieving comparable performance.

GRU have two gates

1. Reset gate (r_t)

Determines how much of past information to forget

$$r_t = \sigma(W_r[h_{t-1}, x_t] + b_r)$$

When $r_t \approx 0$: Ignore previous hidden state (act like reading fresh input)

When $r_t \approx 1$: fully retain previous hidden state

This allows the model to drop irrelevant past information

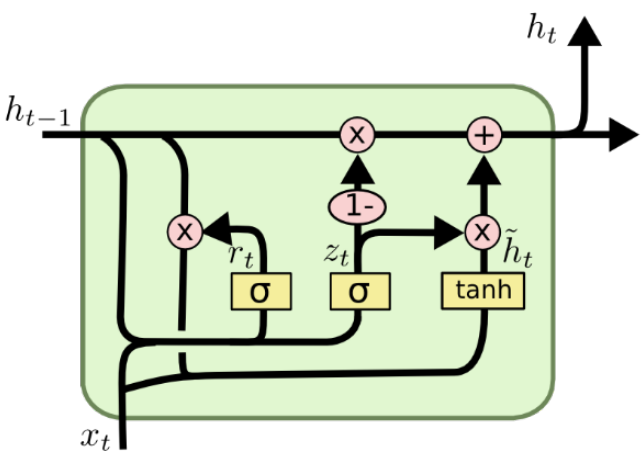
2. Update Gate (z_t)

Determines how much of the past information to carry forward (acts like combined forget + input gate from LSTM)

$$z_t = \sigma(W_z[h_{t-1}, x_t] + b_z)$$

When $z_t \approx 1$: mostly keep old hidden state (skip current input)

When $z_t \approx 0$: mostly use new candidate hidden state



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

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

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

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

Step by Step Computation

Before that, let's understand the symbols

x_t ⟶ Current Input

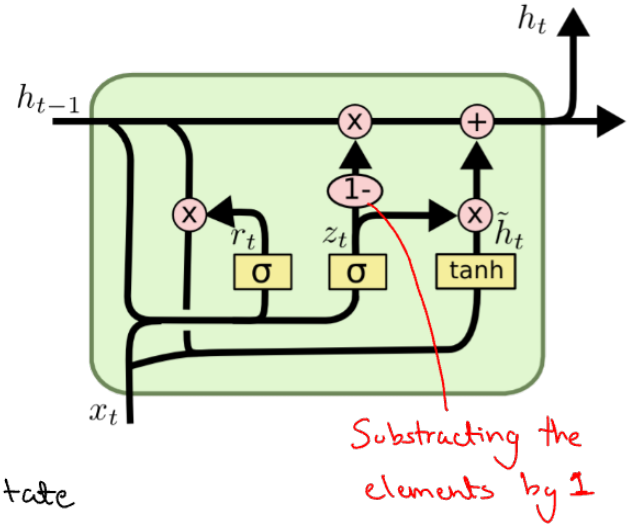
h_{t-1} ⟶ previous hidden state

h_t ⟶ Current hidden state

r_t ⟶ Reset gate

z_t ⟶ Update gate

$\tilde{h}_t$ ⟶ Candidate Hidden State



Now that we know the notations, lets explore step by step implem.

① Calculate Reset gate (r_t)

$$r_t = \sigma(W_r [h_{t-1}, x_t] + b_r)$$

② Compute update gate (z_t)

$$z_t = \sigma(W_z [h_{t-1}, x_t] + b_z)$$

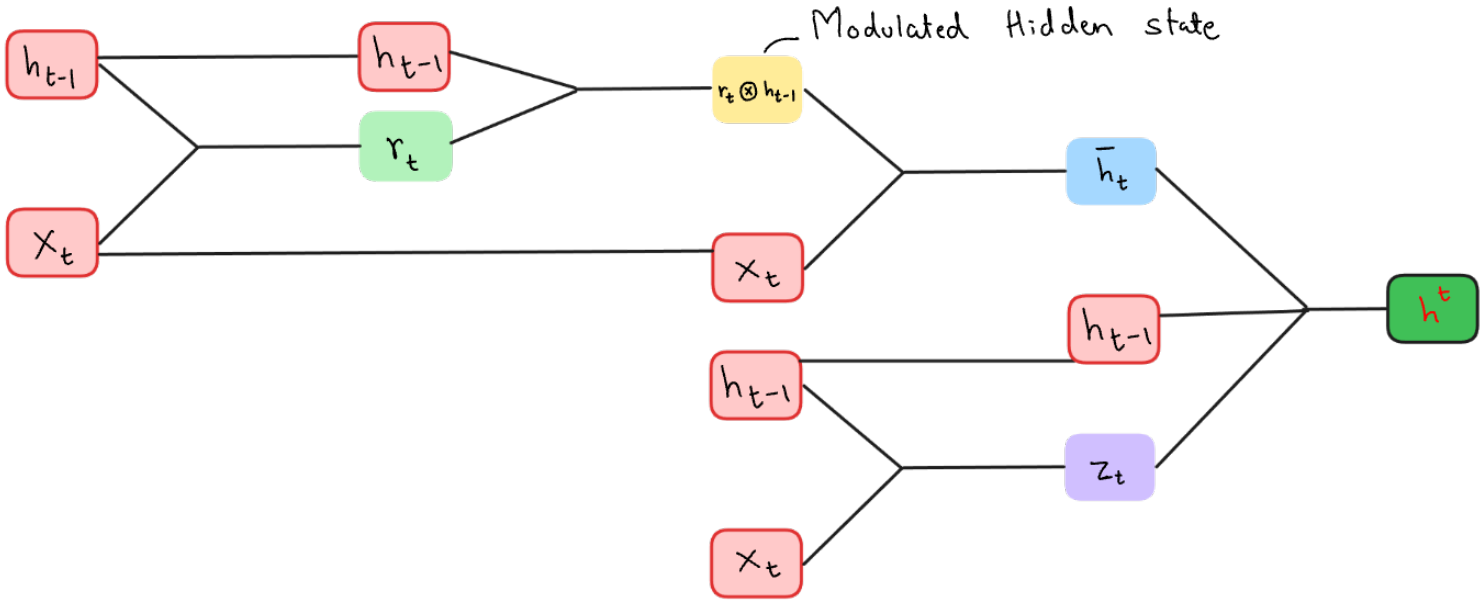
③ Compute Candidate State ($\tilde{h}_t$)

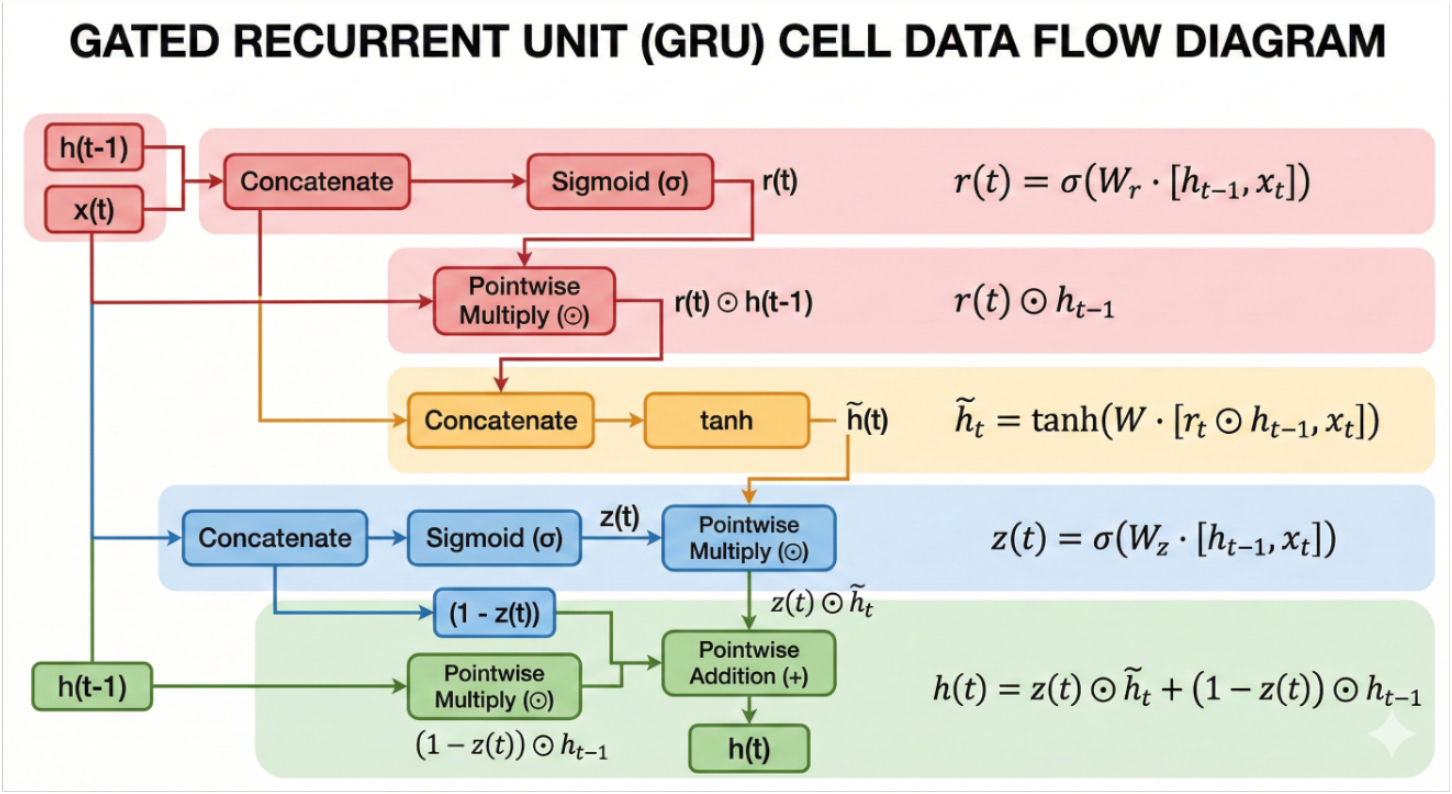
$$\tilde{h}_t = \tanh(W_h [r_t \otimes h_t, x_t] + b_h)$$

④ Compute Final Hidden State (h_t)

h_{t-1}
x_t

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





GRU — Step-by-Step Data Flow

Computation flow through a single Gated Recurrent Unit cell

Step 1: Reset Gate (rₜ)

Concatenate the previous hidden state **h(t-1)** and current input **x(t)**, then pass through **sigmoid (σ)** to get the reset gate **r(t)**. It outputs values in [0, 1] — deciding how much past memory to forget before computing the new candidate. Values near 0 mean "ignore the past", near 1 mean "keep the past."

```
r(t) = σ( W_r · [h(t-1), x(t)] + b_r )
```

Step 2: Modulated (Reset) Hidden State

Pointwise multiply **r(t)** ⊙ **h(t-1)** to produce the **modulated hidden state** — a filtered version of old memory. The reset gate acts as a mask: where r(t) ≈ 0, that part of h(t-1) gets zeroed out (forgotten). This is an intermediate result, not a gate output.

```
modulated state = r(t) ⊙ h(t-1)
```

Step 3: Candidate Hidden State (h̃(t))

Concatenate the modulated state with **x(t)** again and pass through **tanh** to produce the **candidate hidden state h̃(t)** — the proposed new memory. Tanh squashes values to [-1, 1]. This represents what the hidden state *could be* if we fully updated. It's a "proposal" that gets blended with the old state in Step 5.

```
h̃(t) = tanh( W_h · [r(t) ⊙ h(t-1), x(t)] + b_h )
```

Step 4: Update Gate (zₜ)

Separately (can run in parallel with Steps 1–3), concatenate **h(t-1)** and **x(t)** and pass through **sigmoid (σ)** to get the **update gate z(t)**. This controls the balance: how much old memory to keep vs. how much new candidate to use. Similar to the combined forget + input gate in LSTM.

```
z(t) = σ( W_z · [h(t-1), x(t)] + b_z )
```

Step 5: Final Hidden State h(t) — Linear Interpolation

Compute the final output **h(t)** via **linear interpolation** between old and new, controlled by the update gate. When z(t) ≈ 1, output is mostly the new candidate (update memory). When z(t) ≈ 0, output is mostly the old state (preserve memory). This also helps solve the **vanishing gradient problem** — when z(t) ≈ 0, gradients flow directly from h(t) to h(t-1) like a skip connection.

```
h(t) = z(t) ⊙ h̃(t)  +  (1 - z(t)) ⊙ h(t-1)
```

Quick Summary

[h(t-1), x(t)] → σ → **r(t)** → **r(t)⊙h(t-1)** → concat with x(t) → tanh → **h̃(t)**

[h(t-1), x(t)] → σ → **z(t)** → **z(t)⊙h̃(t) + (1-z(t))⊙h(t-1) = h(t)**

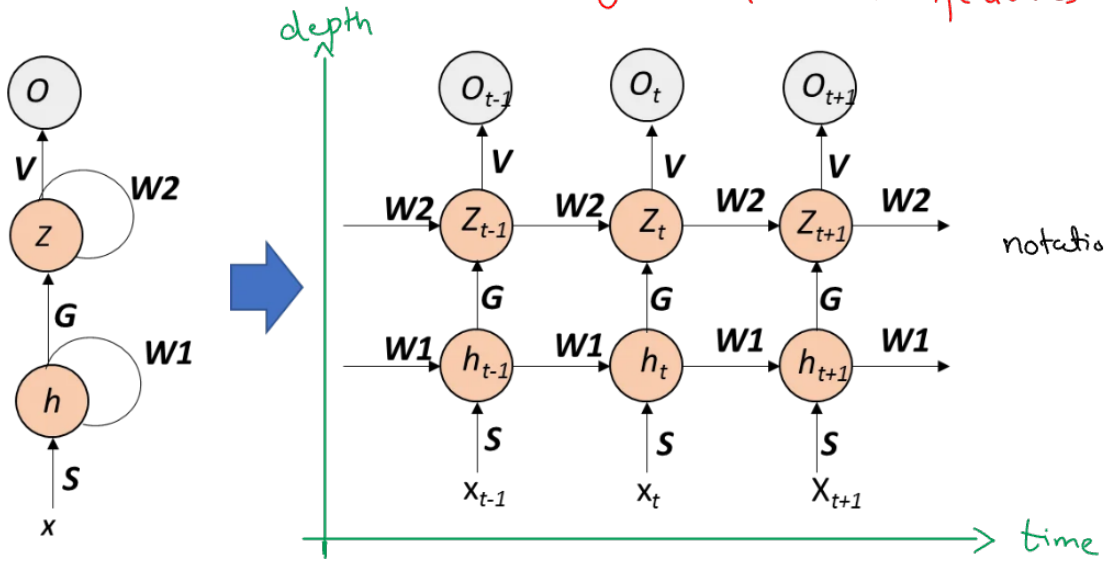
Deep RNN

Deep RNN is simply multiple RNN layers stacked on top of each other. Instead of having a single hidden layer, we stack multiple RNN layers vertically. The output (hidden state) of one RNN layer becomes the input to the next RNN layer above it.

A standard RNN (shallow) RNN has one hidden layer between input and output. A deep RNN has multiple hidden layers, allowing the network to learn hierarchical representations of the data - similar to how deep CNNs learn low-level to high-level features.

If we have multiple layers:

- few first layers will learn low level features (individual word patterns etc)
- few middle layers can learn mid level features (phrases, etc)
- last few layers can learn high level (abstract features (sentiment, topic etc)



notation $h^{(\text{layer no.})}_{(\text{time series})}$

a) 2-layer Recurrent Neural Network (RNN) b) Unfolded 2-layer Recurrent Neural Network (RNN)

The simple idea is that when we stack layers vertically, the initial first layer will get h_{t-1} and x_i as the input but the layer vertically (up direction) connected will get h_{t-1} from the bottom layer and it's own h_{t-1}.

Architecture

We have an input sequence $x_1, x_2, x_3 \ldots x_n$ and L stacked layers:

Layer 1 (bottom):

$$h_t^{(1)} = f\left(W^{(1)} \cdot [h_{t-1}^{(1)}, x^t] + b^{(1)}\right)$$

Takes the actual input x, and its own previous hidden state.

Layer 2:

$$h_t^{(2)} = f\left(W^{(2)} [h_{t-1}^{(2)}, h_t^{(1)}] + b^{(2)}\right)$$

Takes the hidden state output from layer 1 as input

Layer l (general form):

$$h_t^{(l)} = f\left(W^l [h_{t-1}^l, h_t^{l-1}] + b^l\right)$$

Each layer takes two things: its own previous hidden state h_{t-1}^l (horizontal flow, across time) and the output from the layer below $h_t^{(l-1)}$ (vertical flow, across depth)

Output: $$y_t = g\left(W_y h_t^{(L)} + b_y\right)$$

The final prediction comes from the topmost layer's hidden state.

How data flows:

There are two directions of information flow.

① Horizontal (across time): Within each layer, the hidden state flows from timestamp $t-1$ to t, this captures temporal / sequential patterns (same as regular RNN).

② Vertical (across depth): At each timestep, the hidden state flows from layer 1 up to layer L - this captures hierarchal / abstract representations.

So at every timestep t, data flows bottom-to-top through all layers. And across timesteps, each layer maintains its own horizontal chain of hidden states.

Deep RNN's are used in google's neural machine translation which uses 8 layer deep LSTM for speech recognition systems etc.

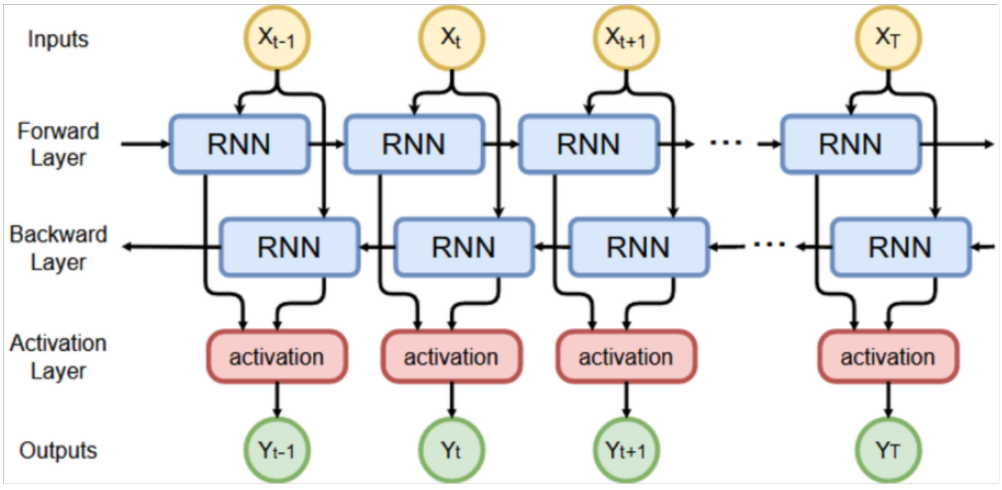
We can also replace those layers with LSTM or GRU blocks.

Generally, traditional single layer RNN are not used, and deep RNN, LSTM's are preffered instead.

Bidirectional RNN / LSTM

A bidirectional RNN processes the input scentence in two dimensions simultaneously — forward (left to right) and backward (right to left). It uses two seperate hidden states : one that reads the sequence from start to end, and another that reads from end to start. The outputs from both directions are then concatenated at each timestep to form the final output.

The idea is simple : In a standard RNN, the hidden state at timestep t only has information about the past (tokens before t). But in many tasks, the future context (tokens after t) is equally important for understanding current token.



We simply have another opposite RNN layer running along with the forward one. And at the end, both of the output from forward and backward layer are concatenated together to get the final output.

Pros:

- Access to full context
- Significantly better accuracy of tasks like NGR, sentiment analysis.
- Works with any cell type - Vanilla RNN, LSTM, GRU
- The two directions are independent, so they can be parallelized during computation.

Cons:

- Cannot be used in Real time
- Double parameters
- Higher computation cost
- Higher latency at inference
- Can overfit easily on small datasets due to increased parameters count.