CSL446
Neural Network and Deep Learning

Module 3

Dr. Snehal B Shinde

Computer Science and Engineering

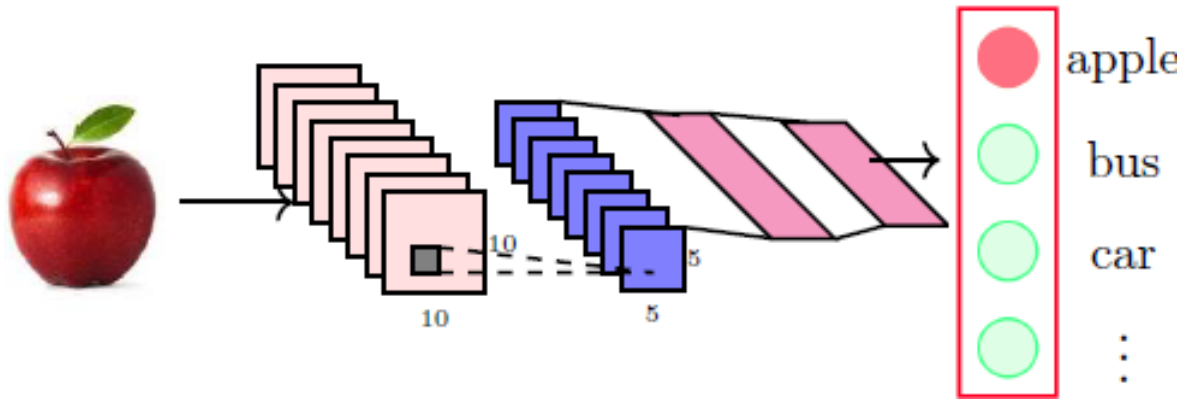
Indian Institute of Information Technology, Nagpur.

Sequence Models

- Sequence Models
- RNN
- BPTT
- LSTM & GRU
- Word Embeddings
- Attention Mechanism

Sequence Models

- In feedforward and convolutional neural networks the size of the input was always fixed.
- For example, we fed fixed size (32*32) images to convolutional neural networks for image classification.
- Further, each input to the network was independent of the previous or future inputs.
- For example, the computations, outputs and decisions for two successive images are completely independent of each other.

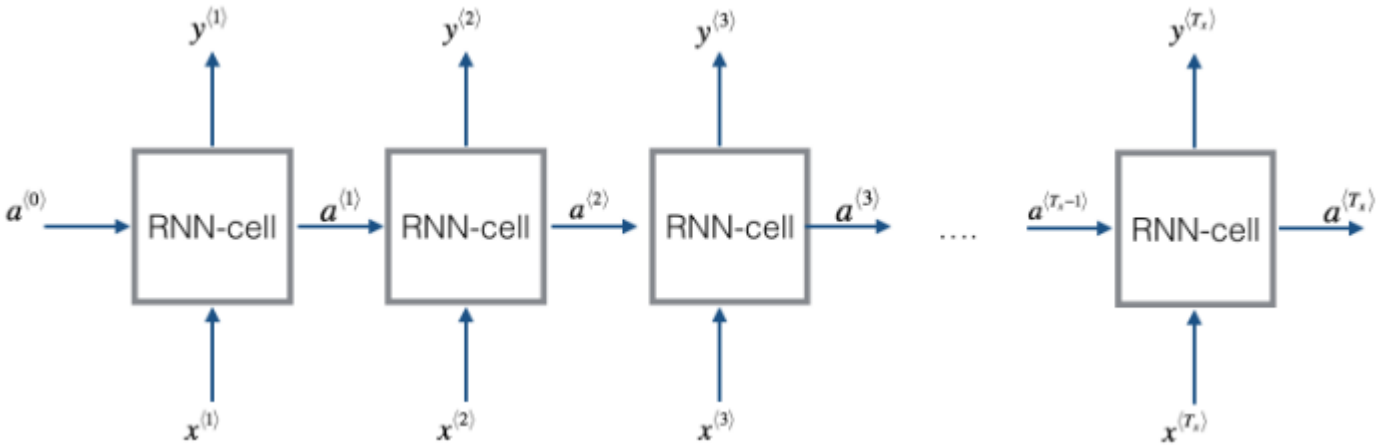


Problems with feedforward n/w and CNN

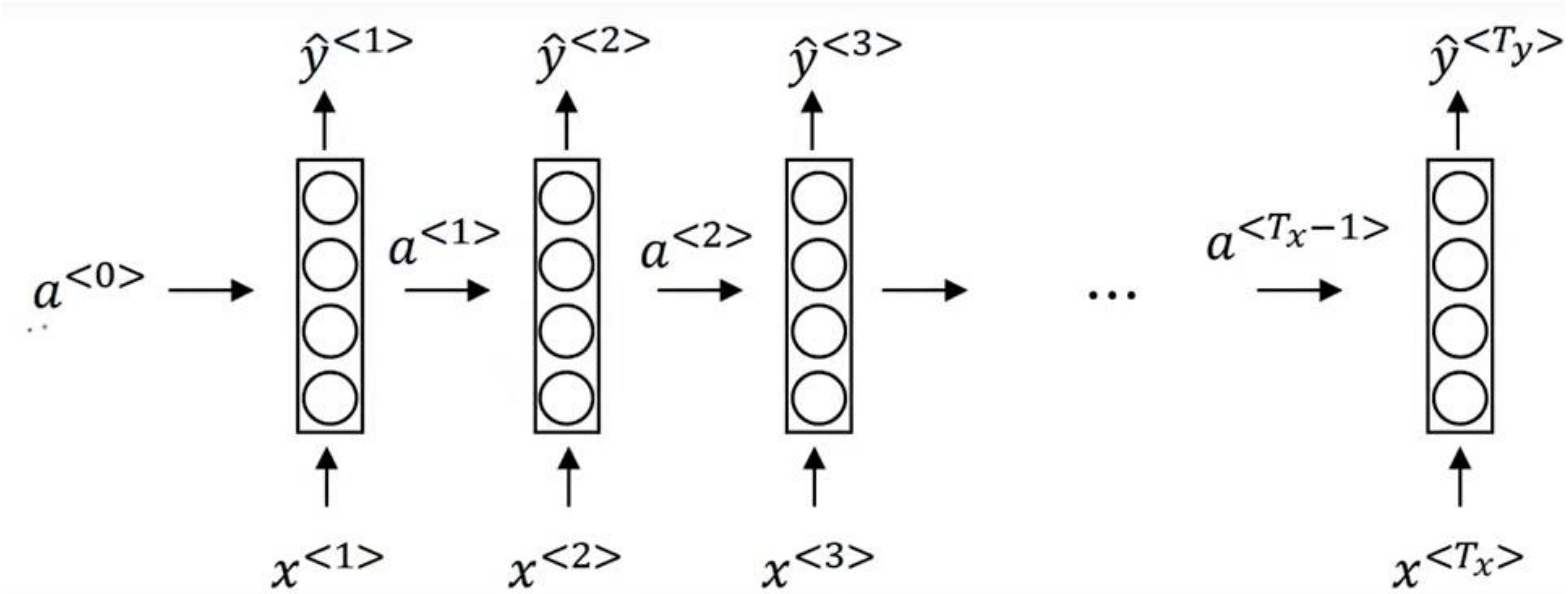
- Present predictions don’t have persistence of previous predictions.
- A feed-forward neural network can perform simple classification, regression, or recognition tasks, but it can’t remember the previous input that it has processed.
- RNNs are designed to capture long-term dependencies in sequential data
- Due to absence of memory cells, it cannot store information observed in previous.
- The Obstacles:
- Input, output can be of different length in different examples.
- Doesn’t share features learned across different position of input.

Sequence Models

- Sequence models are the machine learning models that input or output sequences of data.
- Sequential data includes text streams, audio clips, video clips, time-series data and etc.
- Recurrent Neural Networks (RNNs) is a popular algorithm used in sequence models.
- Recurrent Neural Networks introduce a mechanism where the output from one step is fed back as input to the next, allowing them to retain information from previous inputs. This design makes RNNs well-suited for tasks where context from earlier steps is essential, such as predicting the next word in a sentence.



Forward Propagation



$a^{\langle 0 \rangle} = \vec{0}$.

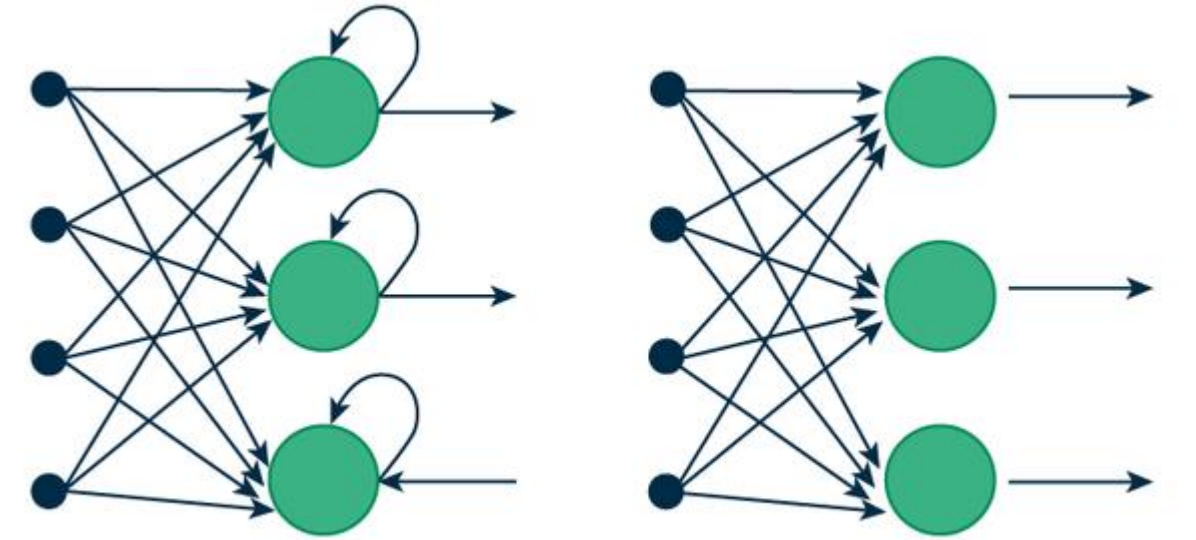
$$a^{\langle 1 \rangle} = g_1(W_{aa} a^{\langle 0 \rangle} + W_{ax} x^{\langle 1 \rangle} + b_a) \leftarrow \tanh / \text{ReLU}$$

$$\hat{y}^{\langle 1 \rangle} = g_2(W_{ya} a^{\langle 1 \rangle} + b_y) \leftarrow \text{sigmoid}$$

$$a^{\langle t \rangle} = g(W_{aa} a^{\langle t-1 \rangle} + W_{ax} x^{\langle t \rangle} + b_a)$$

$$\hat{y}^{\langle t \rangle} = g(W_{ya} a^{\langle t \rangle} + b_y)$$

How RNN Differs from Feedforward Neural Networks

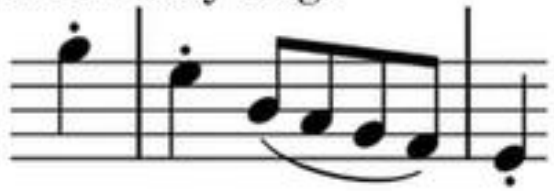


(a) Recurrent Neural Network (b) Feed-Forward Neural Network

- RNN solve this by incorporating loops that allow information from previous steps to be fed back into the network.
- This feedback enables RNNs to remember prior inputs, making them ideal for tasks where context is important.

- Feedforward Neural Networks (FNNs) process data in one direction, from input to output, without retaining information from previous inputs.
- This makes them suitable for tasks with independent inputs, like image classification.
- However, FNNs struggle with sequential data since they lack memory.

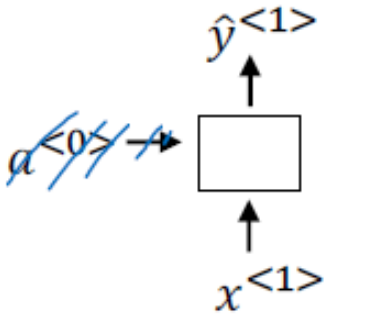
Human Vision - many layers of abstraction - Deep learning

Speech recognition		speech into text →	"The quick brown fox jumped over the lazy dog."
Music generation		generates music →	
Sentiment classification	"There is nothing to like in this movie."	sentiment of the text →	★☆☆☆☆
DNA sequence analysis	AGCCCCTGTGAGGAACTAG	mutations, or classifications →	AGCCCCTGTGAGGAACTAG
Machine translation	Voulez-vous chanter avec moi?	translates the text →	Do you want to sing with me?
Video activity recognition		classifies the activity →	Running
Name entity recognition	Yesterday, Harry Potter met Hermione Granger.	classifies proper nouns →	Yesterday, Harry Potter met Hermione Granger.

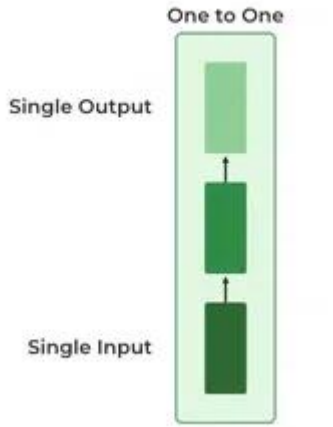
Types Of Recurrent Neural Networks

One-to-One RNN behaves as the Vanilla Neural Network, is the simplest type of neural network architecture.

In this setup, there is a single input and a single output. Commonly used for straightforward classification tasks where input data points do not depend on previous elements.



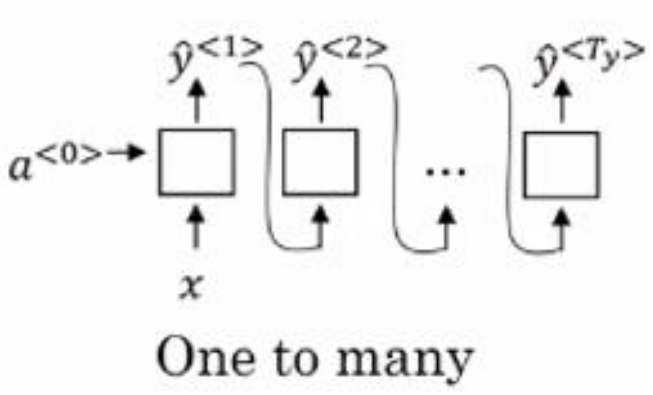
One to one

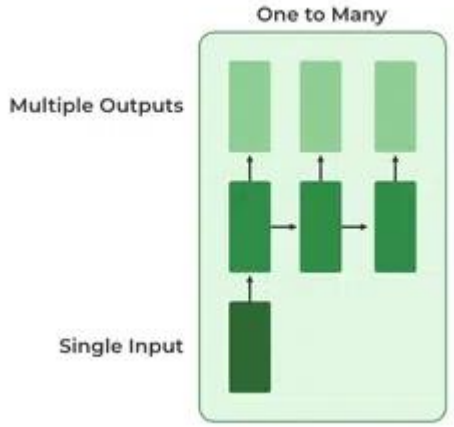


One-to-Many RNN

In a **One-to-Many RNN**, the network processes a single input to produce multiple outputs over time. This setup is beneficial when a single input element should generate a sequence of predictions.

- image captioning task, a single image as input, the model predicts a sequence of words as a caption.

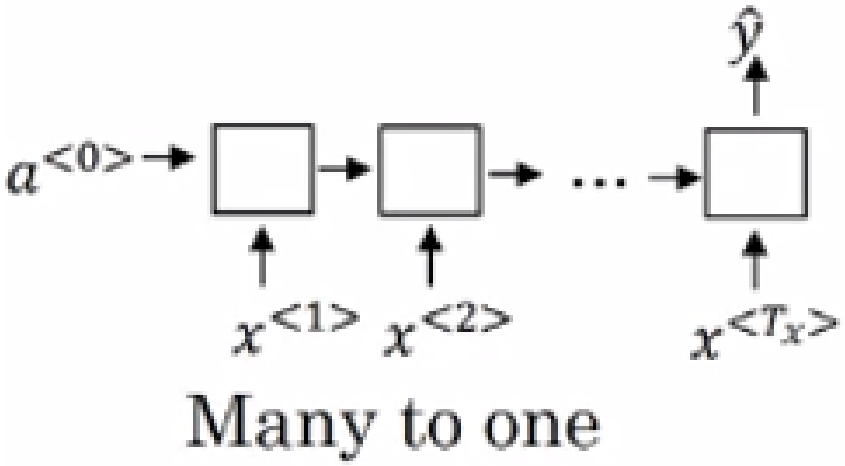


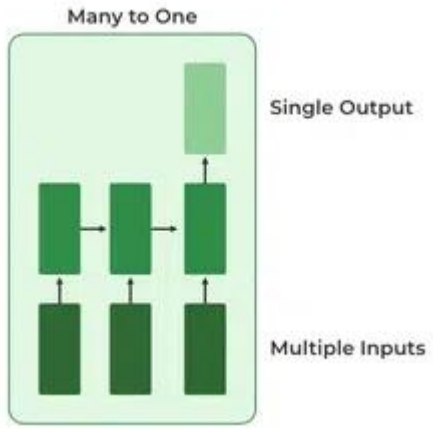


Many-to-One RNN

The **Many-to-One RNN** receives a sequence of inputs and generates a single output. This type is useful when the overall context of the input sequence is needed to make one prediction.

In sentiment analysis, the model receives a sequence of words (like a sentence) and produces a single output, which is the sentiment of the sentence (positive, negative, or neutral).

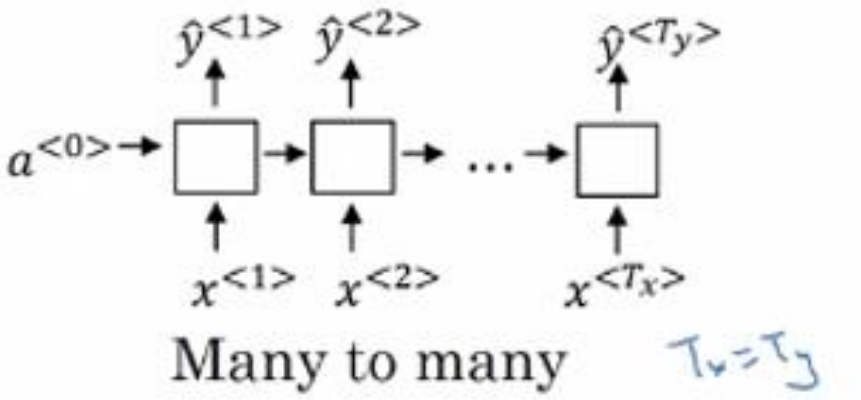


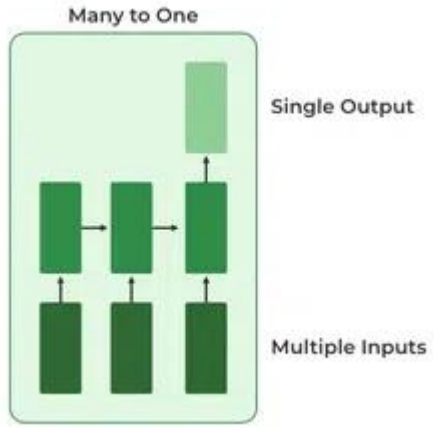


Many-to-Many RNN

The **Many-to-Many RNN** type processes a sequence of inputs and generates a sequence of outputs.

In language translation task, a sequence of words in one language is given as input, and a corresponding sequence in another language is generated as output.





What is the Need for Sequence Modeling?

- Sequence modeling is essential for tasks where the order of data points matters.
- Many real-world problems involve sequential dependencies where past information influences the future.
- Sequence models help capture these dependencies, making them crucial in various domains.

Applications of Sequence Modeling

- **Natural Language Processing (NLP)**
    1. Machine Translation (e.g., Google Translate)
    2. Speech-to-Text (e.g., Siri, Google Assistant)
    3. Chatbots and Conversational AI

- **Time-Series Analysis**
    1. Stock Market Forecasting
    2. Weather Prediction
    3. Energy Demand Forecasting

- **Healthcare & Bioinformatics**
    1. ECG & EEG Signal Analysis
    2. DNA Sequence Analysis
    3. Medical Diagnosis Based on Patient History

- **Autonomous Systems**
    1. Self-Driving Cars (analyzing sensor data over time)
    2. Robotics (learning from sequential actions)

- **Reinforcement Learning**
    1. Gaming AI (learning strategies from sequences of actions)
    2. Robotics and Control Systems

Sequence Modeling Techniques

Traditional Methods:

1. Markov Models, (future state depends only on the current state (not on past states))
2. Hidden Markov Models (HMMs) (**hidden states**, where we observe only their effects.)

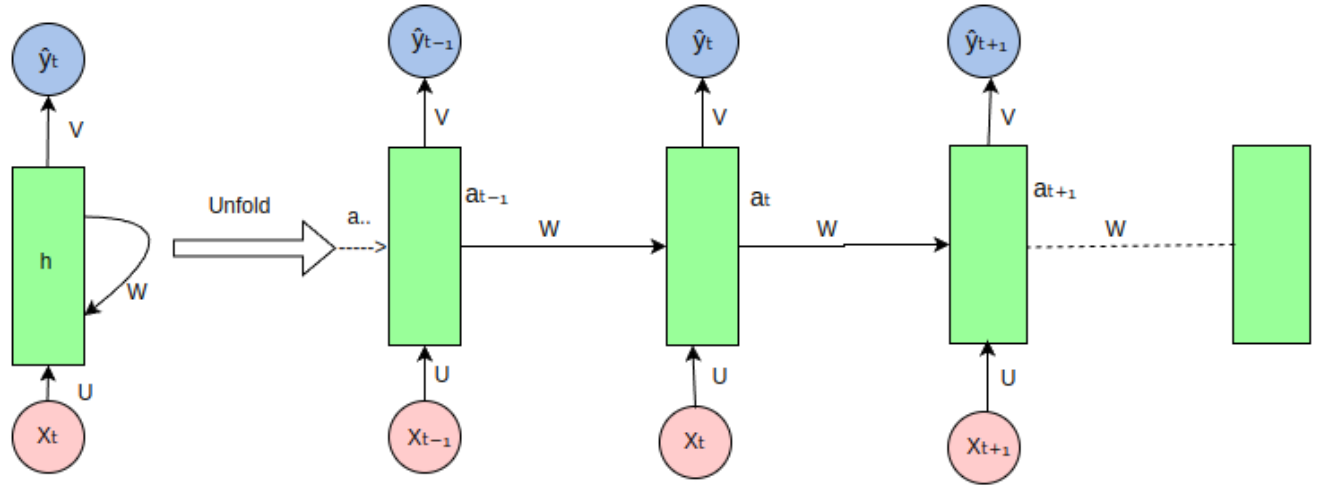
Deep Learning Approaches:

1. **Recurrent Neural Networks (RNNs)** – Good for sequential dependencies

2. **Long Short-Term Memory (LSTMs) & Gated Recurrent Units (GRUs)** – Handle long-term dependencies

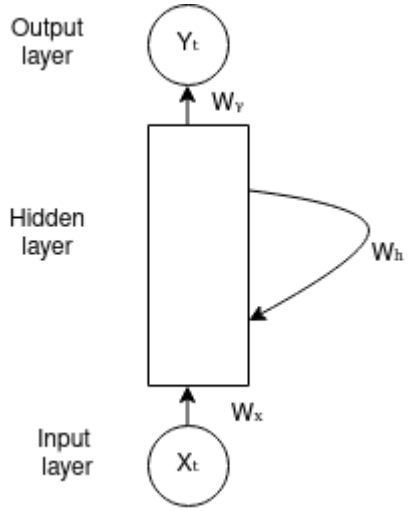
3. **Transformers (e.g., BERT, GPT, T5)** – More efficient, capturing global dependencies with self-attention

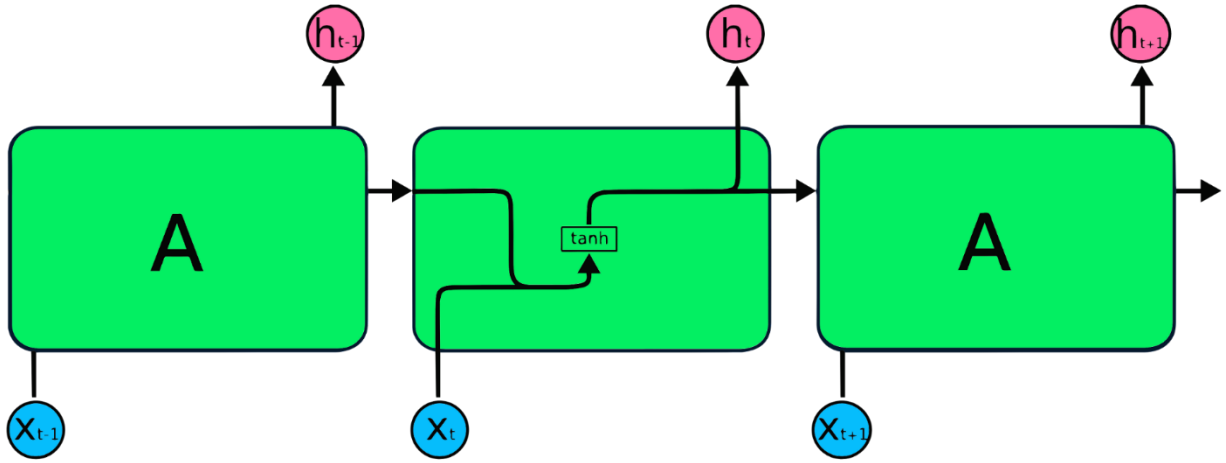
Simple RNN Cell

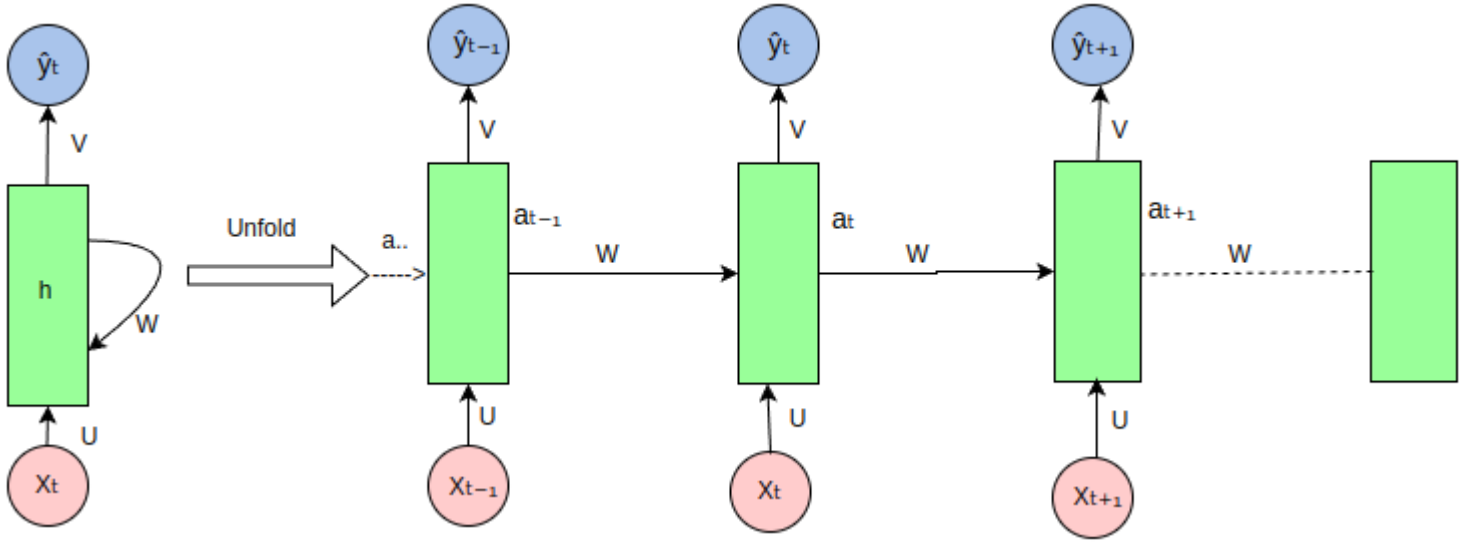


- The RNN takes an input vector X and the network generates an output vector y by scanning the data sequentially from left to right, with each time step updating the hidden state and producing an output.
- It shares the same parameters (U, V, W) across all time steps.
  - U represents the weight parameter governing the connection **from input layer X to the hidden layer h** ,
  - W represents the weight associated with the connection **between hidden layers**,
  - and V for the connection from **hidden layer h to output layer y**.
- This sharing of parameters allows the RNN to effectively **capture temporal dependencies** and process **sequential data more efficiently** by retaining the information from previous input in its current hidden state.

Simple RNN Cell







RNN

- Ex: Trying to predict the last word “sky” in “the clouds are in the sky”, you should have information of previous words.
- You don’t throw everything away and start thinking from scratch again.
- Your thoughts have persistence.

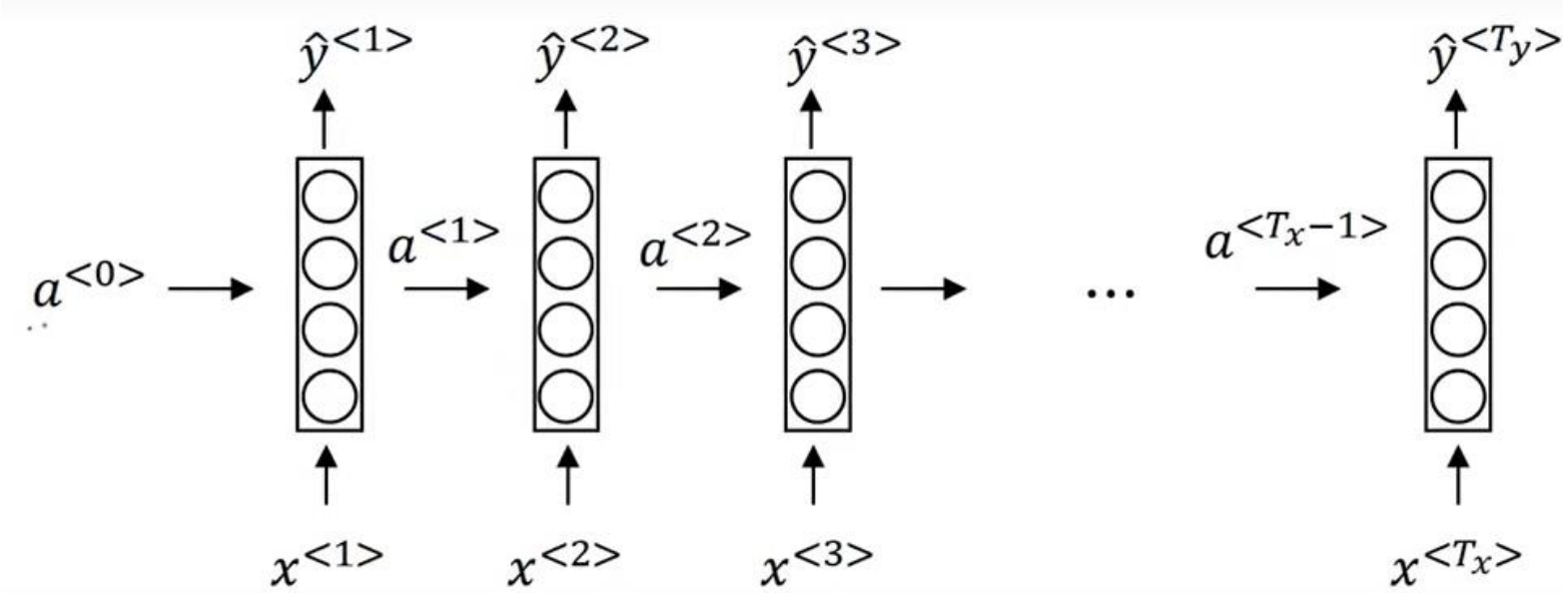
Steps:

- Define the architecture of RNN.
- first initializes the weights and biases.
- Train the NN using Gradient Descent-loss is minimized.

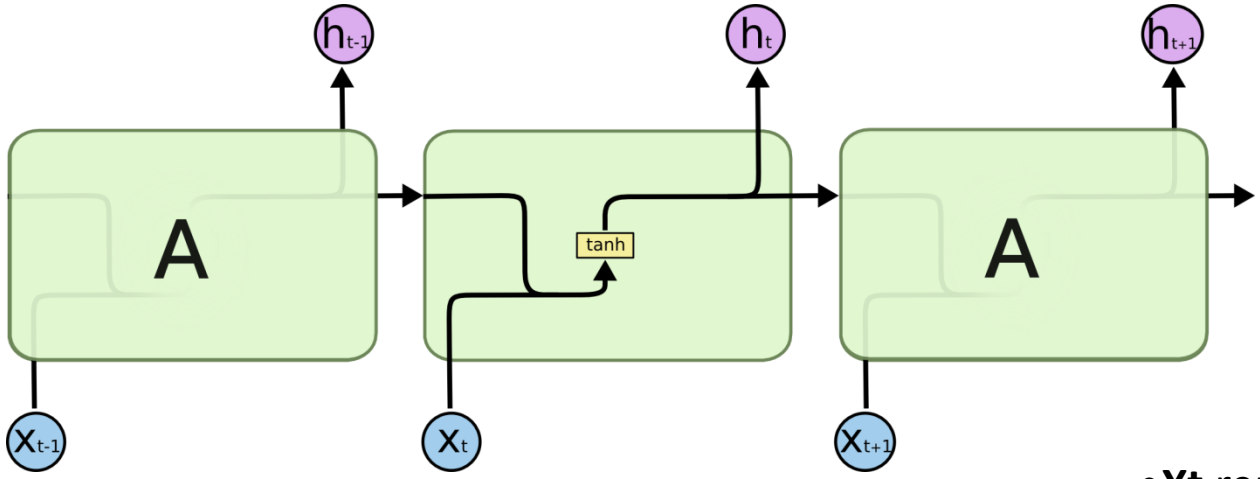
$$L(\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}\}, \{\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(\tau)}\}) = \sum_t L^{(t)} = \sum_t -\log \hat{y}^{(t)}_{y^{(t)}}$$

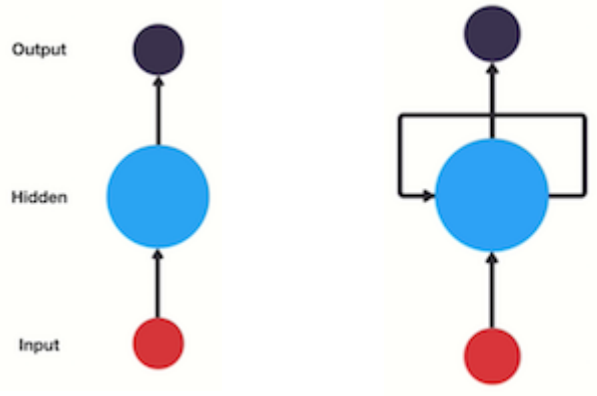
Architectural Design Many to Many

- Input and Output have the same length



RNN Architectures

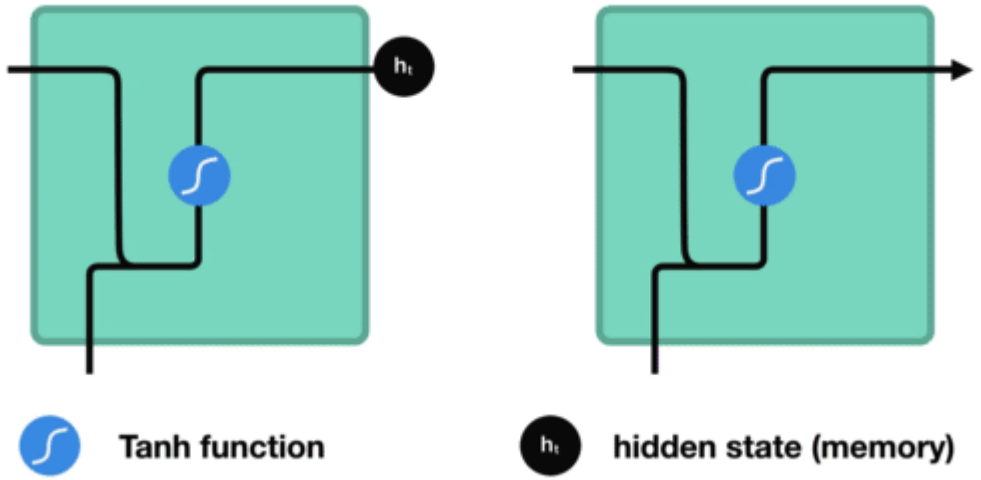




- **Xt** represents the input at time step **t**.
- **ht** represents the hidden state that **retains memory**, which helps in learning temporal dependencies at time step **t**.
- The **arrows show how information flows through time**.
- The activation function used in the hidden state update is **tanh**, which helps in capturing non-linearity.
- This is an **unrolled** view of an RNN, meaning that a single recurrent cell is repeated for each time step.
- **Unrolling** shows how an RNN operates over multiple time steps.
- The **feedback loop** differentiates RNNs from standard feedforward neural networks.

RNN Architectures

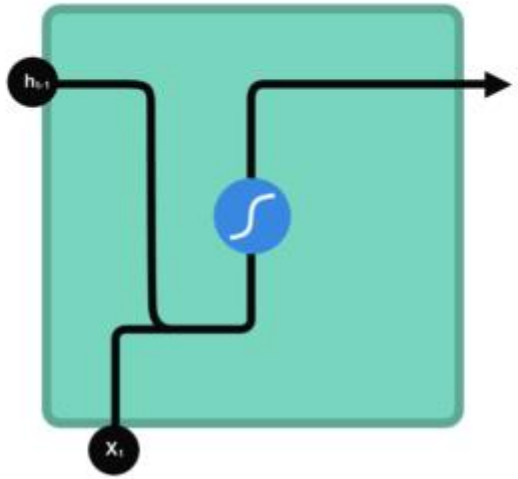
- Passing Hidden state to next time step

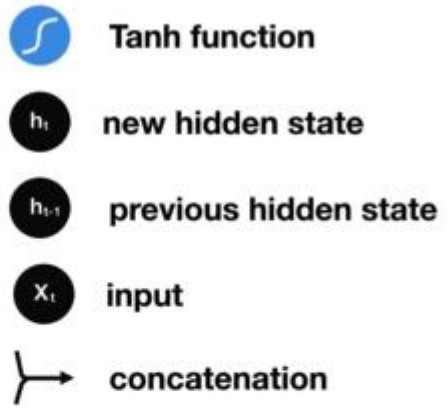


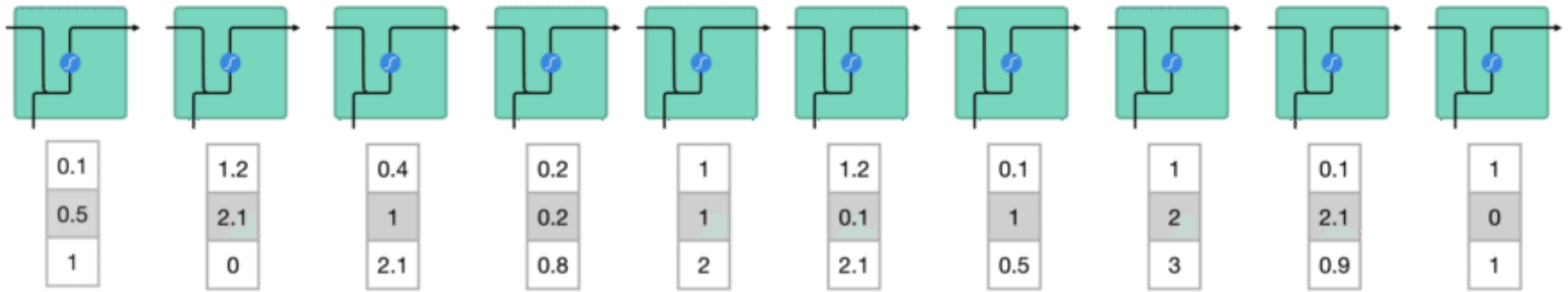
The tanh activation function is applied to transform the input and previous hidden state before passing it forward.

Represents the memory of the RNN, which carries information from previous time steps. Updated at each step using the previous hidden state and the current input.

RNN

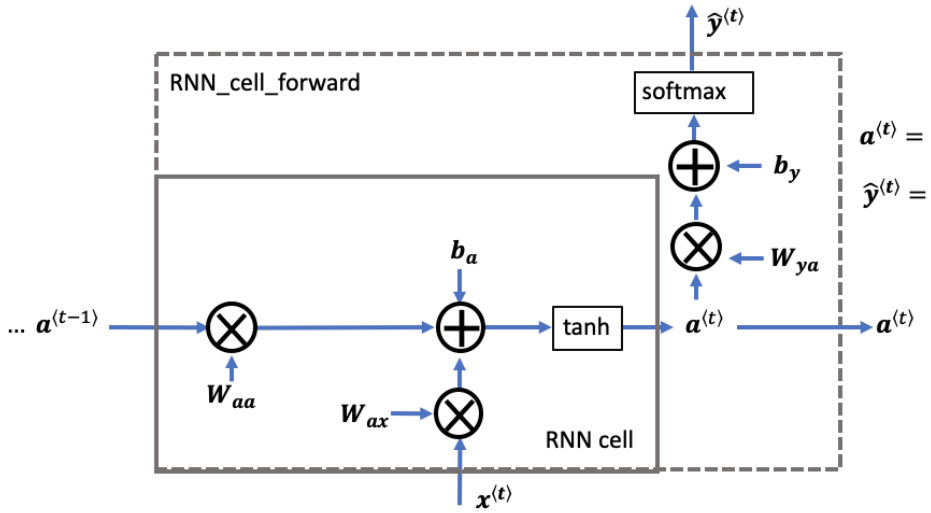

h_{t-1}
x_t


Tanh function
h_t
new hidden state
h_{t-1}
previous hidden state
x_t
input
concatenation


0.1
0.5
1
1.2
2.1
0
0.4
1
2.1
0.2
0.2
0.8
1
1
2
1.2
0.1
2.1
0.1
1
0.5
1
2
3
0.1
2.1
0.9
1
0
1

RNN

Basic RNN cell. Takes as input x⟨t⟩ (current input) and a⟨t−1⟩ (previous hidden state containing information from the past), and outputs a⟨t⟩ which is given to the next RNN cell and also used to predict ŷ ⟨t⟩



$$a^{\langle t \rangle} = \tanh(W_{ax}\, x^{\langle t \rangle} + W_{aa}\, a^{\langle t-1 \rangle} + b_a)$$

$$\hat{y}^{\langle t \rangle} = softmax(W_{ya} a^{\langle t \rangle} + b_y)$$

Step 1: Compute the New Hidden State $a^{(t)}$

$$a^{(t)} = \tanh(W_{ax}x^{(t)} + W_{aa}a^{(t-1)} + b_a)$$

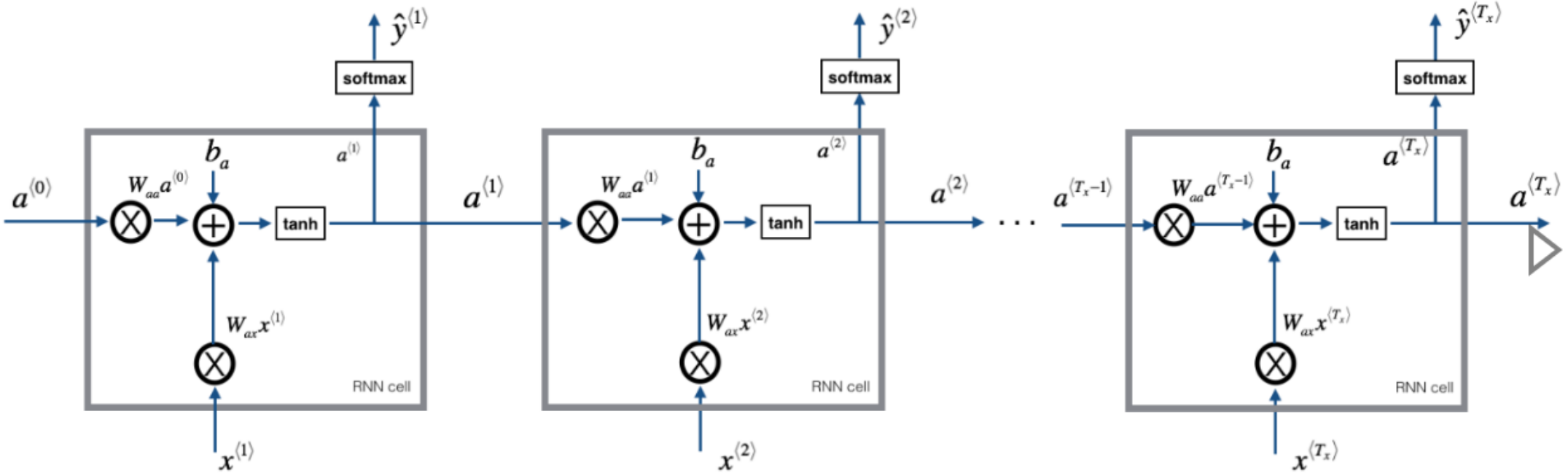
- W_{ax}: Weight matrix for the input $x^{(t)}$.
- W_{aa}: Weight matrix for the previous hidden state $a^{(t-1)}$.
- b_a: Bias term.
- **tanh**: Activation function to introduce non-linearity.

Step 2: Compute the Output $\hat{y}^{(t)}$

$$\hat{y}^{(t)} = \text{softmax}(W_{ya}a^{(t)} + b_y)$$

- W_{ya}: Weight matrix mapping hidden state $a^{(t)}$ to output $\hat{y}^{(t)}$.
- b_y: Bias term.
- **softmax**: Ensures the output is a probability distribution (used in classification tasks).

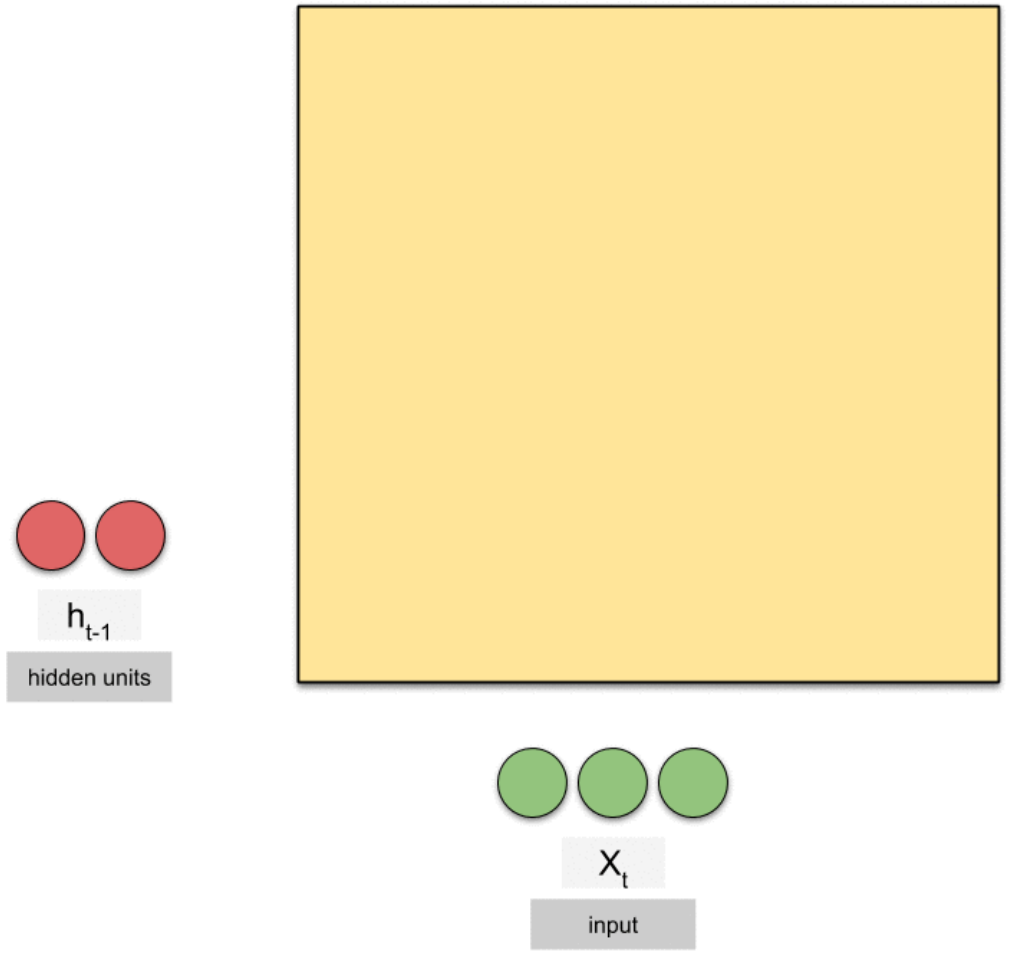
Parameter Sharing & f/w pass Computation



$$a^{\langle t \rangle} = \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b_a)$$

$$\hat{y}^{\langle t \rangle} = soft\max(W_{ya}a^{\langle t \rangle} + b_y)$$

RNN

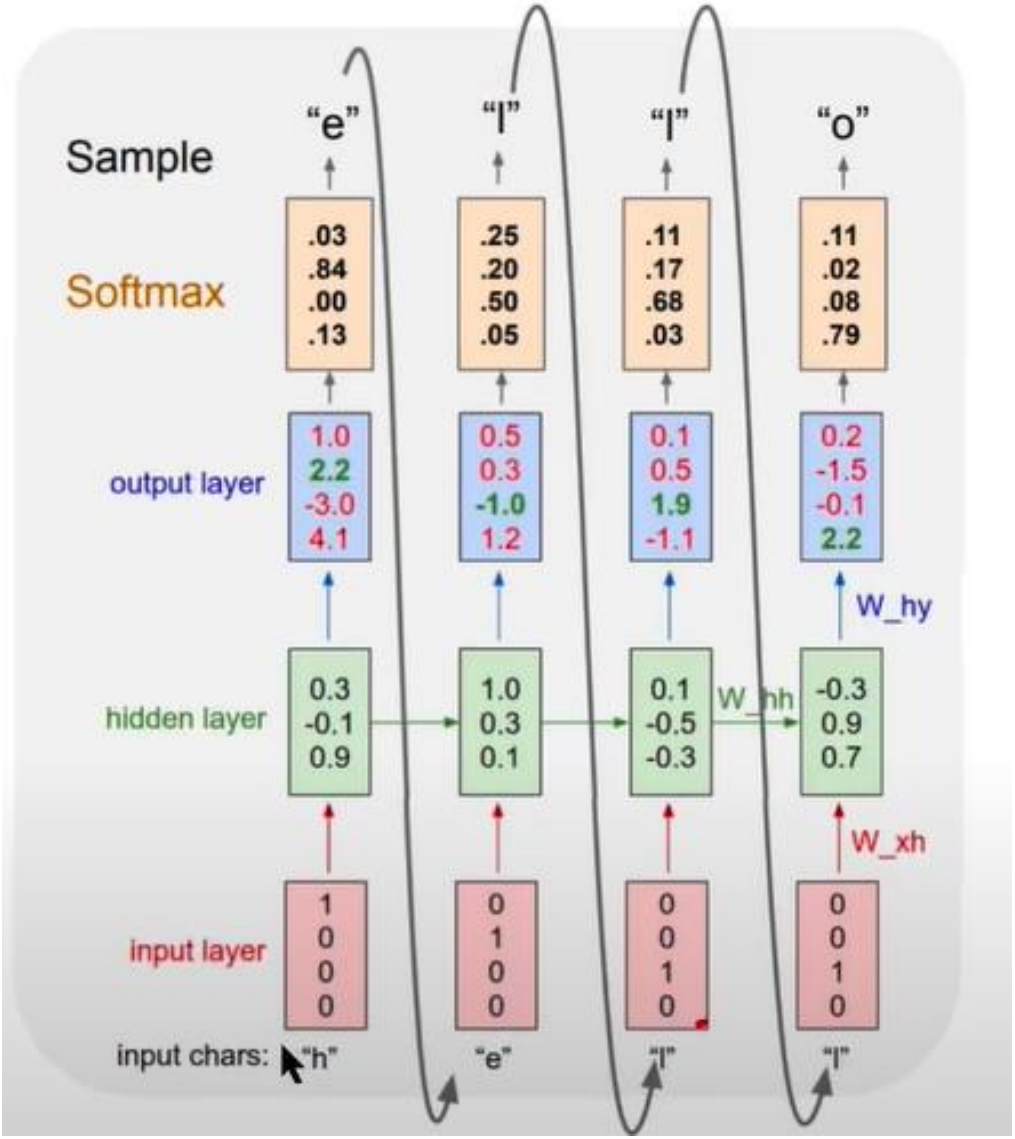


Parameter Setup

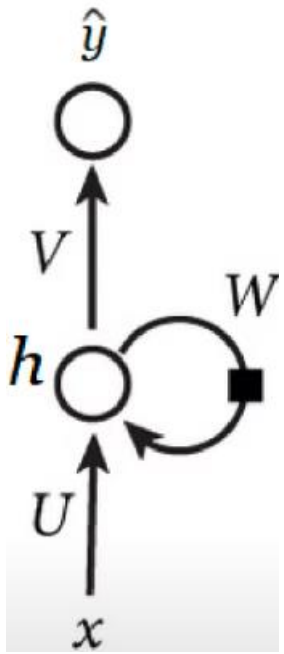
- Decide the dimensions of various weight matrices and biases.
- Initialize all the parameters.
- The weight matrices **are filters that determine** how much importance to accord to both the present input and the past hidden state.
- Decide all the hyperparameters.

Character Level Language Model

- Vocabulary:[h,e,l,o]



Backprop Through Time (BPTT)



- Forward pass equations:

$$h_t = \tanh(Ux_t + Wh_{t-1})$$
$$\hat{y}_t = \text{softmax}(Vh_t)$$

- Loss function e.g., Cross Entropy loss:

$$E_t(y_t, \hat{y}_t) = -y_t \log \hat{y}_t$$
$$E(y_t, \hat{y}_t) = \sum_t E_t(y_t, \hat{y}_t)$$
$$= -\sum_t y_t \log \hat{y}_t$$

- Backpropagation involves adjusting the model's parameters (weights and biases) based on the error between predicted output and the actual target value.
- During training, the error is back propagated through the unfolded network, and the weights are updated using gradient descent.
- This allows the network to learn to predict the output at each time step based on the input at that time step as well as the previous time steps.

Backprop Through Time (BPTT) Unrolled RNN

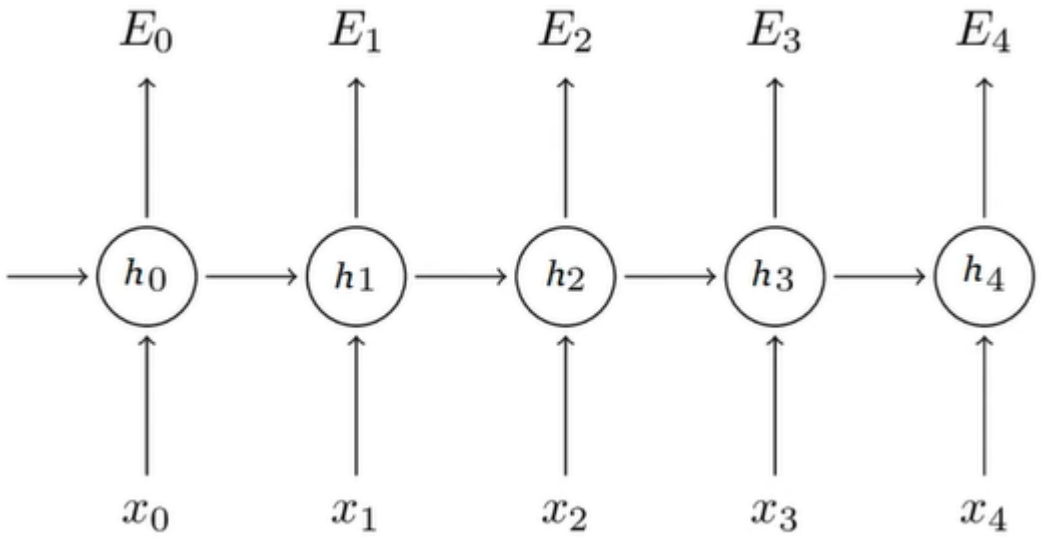
Summarization of BPTT

1. Present a sequence of timesteps of input and output pairs to the network.
2. Unroll the network then calculate and accumulate errors across each timestep.
3. Roll-up the network and update weights.
4. Repeat.
5. Summarization of BPTT

computationally expensive as the number of timesteps increases.

W is used in every step up to the output, we need to back propagate gradients from t=t through the network all the way to t=0.

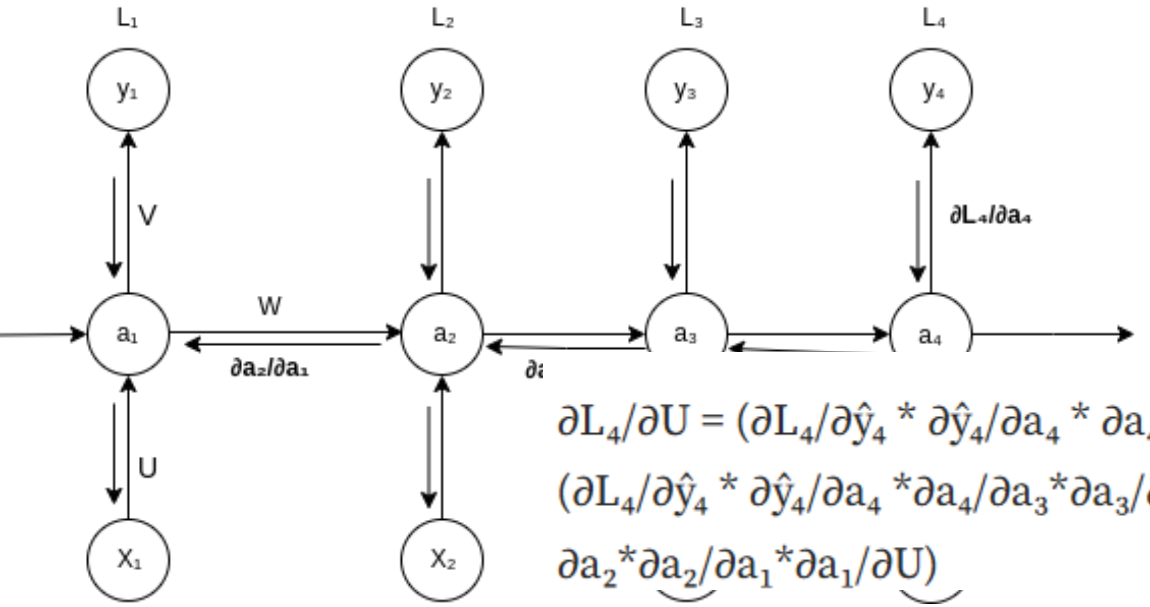
Backpropogation Through Time (BPTT) Unrolled RNN



- Calculate error gradients w.r.t. U, V and W
- Learn weights using SGD
- Just like we sum up errors, we also sum up gradients at each time step for one training example

Backpropagation Through Time (BPTT): We sum up gradients at each time step for one training example: $\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$

backpropagation Through Time (BPTT) Unrolled RNN



Derivative of loss L w.r.t W

$$\partial L_4/\partial W = (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial W) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial W) +$$
$$(\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial a_2 * \partial a_2/\partial W) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/$$

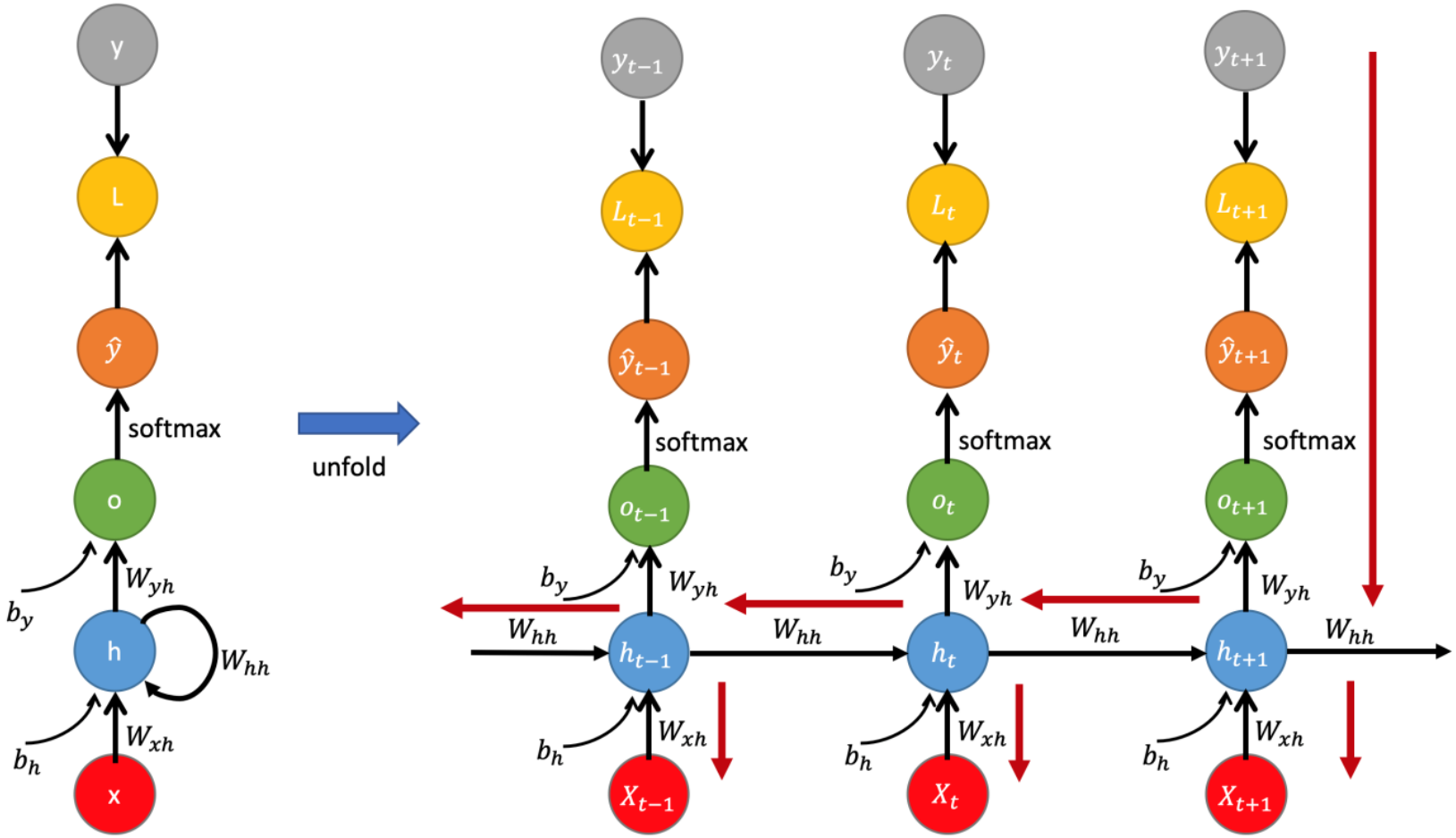
$$\partial L_4/\partial U = (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial U) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial U) +$$
$$(\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial a_2 * \partial a_2/\partial U) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/$$
$$\partial a_2 * \partial a_2/\partial a_1 * \partial a_1/\partial U)$$

- Do you see any problem?
- Sequences (sentences) can be quite long, perhaps 20 words or more need to back-propagate through many layers!

$$\partial L_4/\partial U = (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial U) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial U) +$$
$$(\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial a_2 * \partial a_2/\partial U) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/$$
$$\partial a_2 * \partial a_2/\partial a_1 * \partial a_1/\partial U)$$

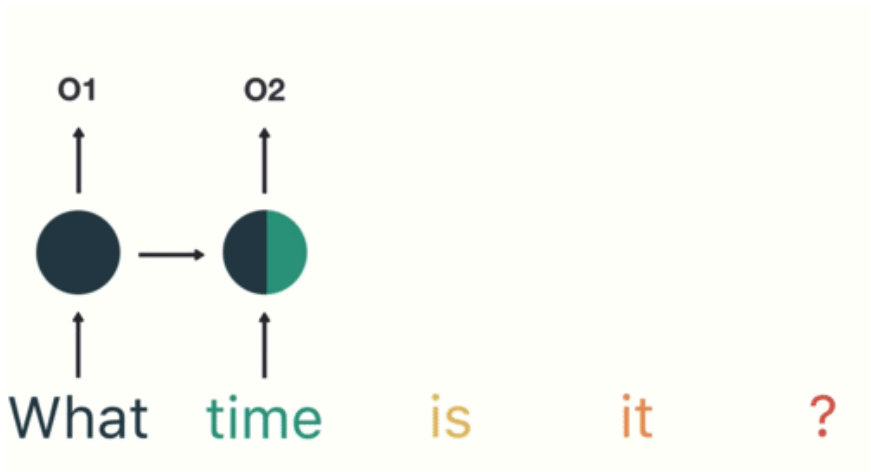
Vanishing gradient problem

backpropagation Through Time (BPTT) Unrolled RNN



Vanishing Gradient Problem with RNN

- Long term dependency problem and example
- Locality influence-Short term memory
- If you are trying to process a paragraph of text to do predictions, RNN's may leave out important information from the beginning.
- During back propagation, RNN suffer from the vanishing gradient problem.



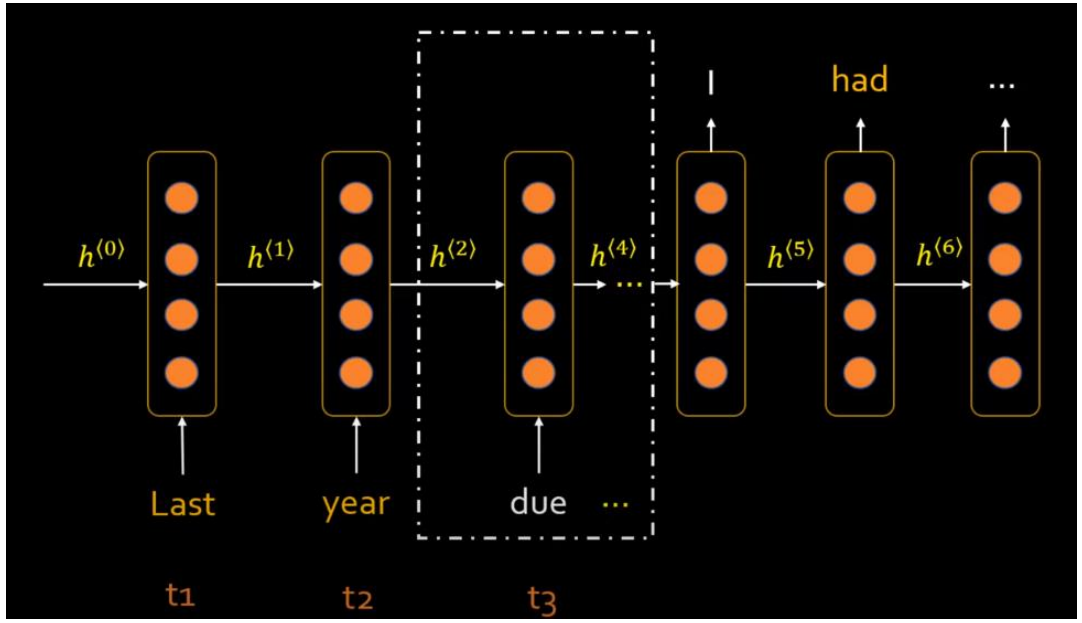
Long Short Term Memory

- **Limitations of BPTT**
  1. Vanishing Gradients
  2. Exploding Gradients

- **How to overcome**
  1. LSTMs (1997)
  2. Recent Variants including GRUs (2014)

Long Short Term Memory

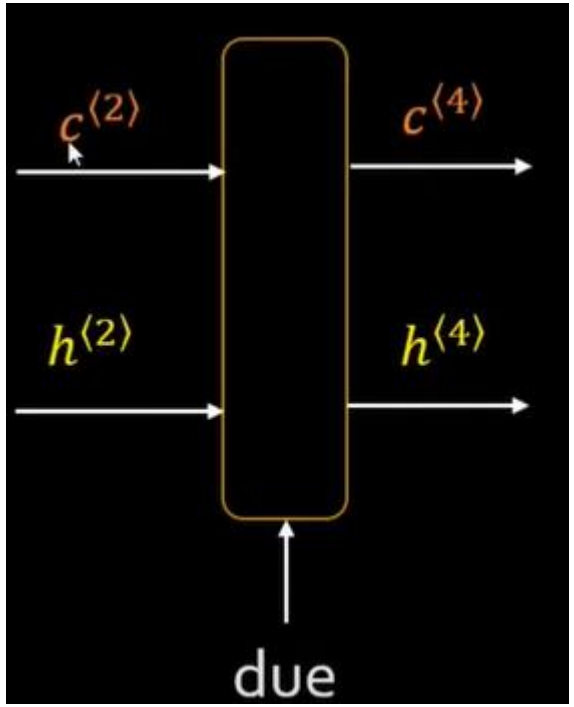
- Capable of learning long-term dependencies using mechanisms called “gates.”
- Previous hidden state is also passed to the next step of the sequence.
- Cell state acts a memory.



"Last year, due to my current job situation and family conditions, I had to take a loan."

Long Short Term Memory

- Capable of learning long-term dependencies using mechanisms called “gates.”
- Previous hidden state is also passed to the next step of the sequence.
- Cell state acts a memory.

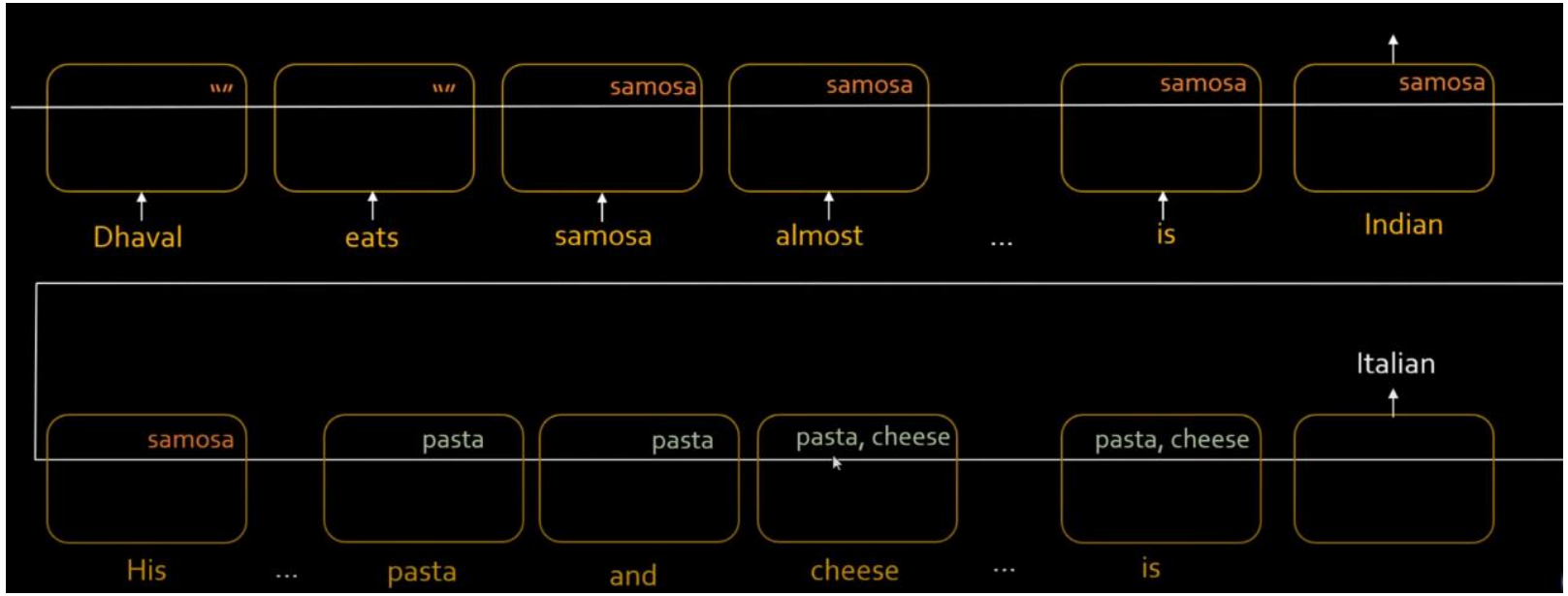


Long Term Memory

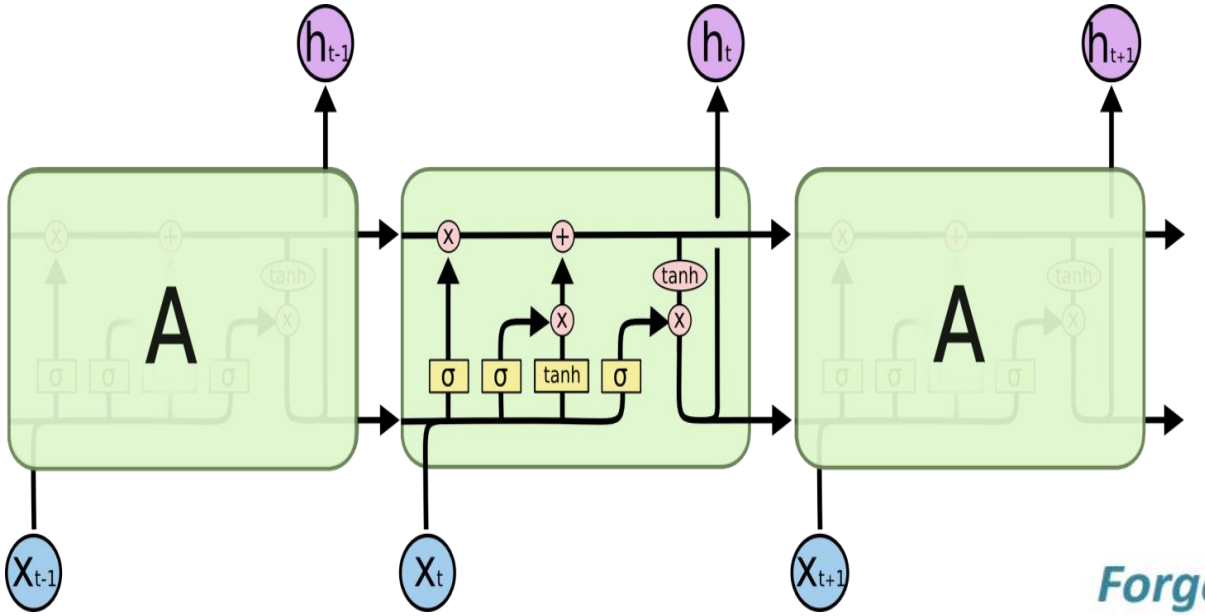
Short Term Memory

Long Short Term Memory

- Dhaval eats samosa almost everyday, it shouldn't be hard to guess that his favorite cuisine is Indian. His brother Bhavin however is a lover of pasta and cheese that means Bhavin's favorite cuisine is Italian

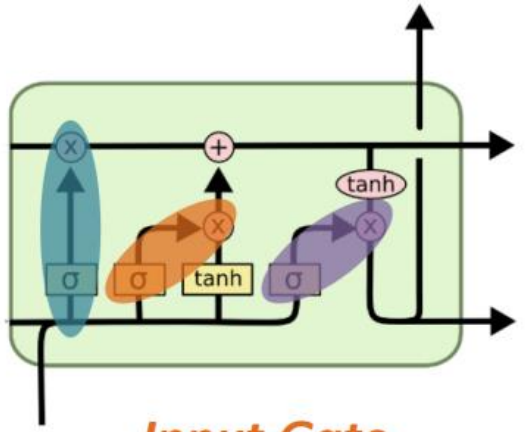


Squashing Functions used in LSTM



- Sigmoid-helpful to update or forget data.
- Tanh-used to help regulate the values flowing through the network.

Forget gate

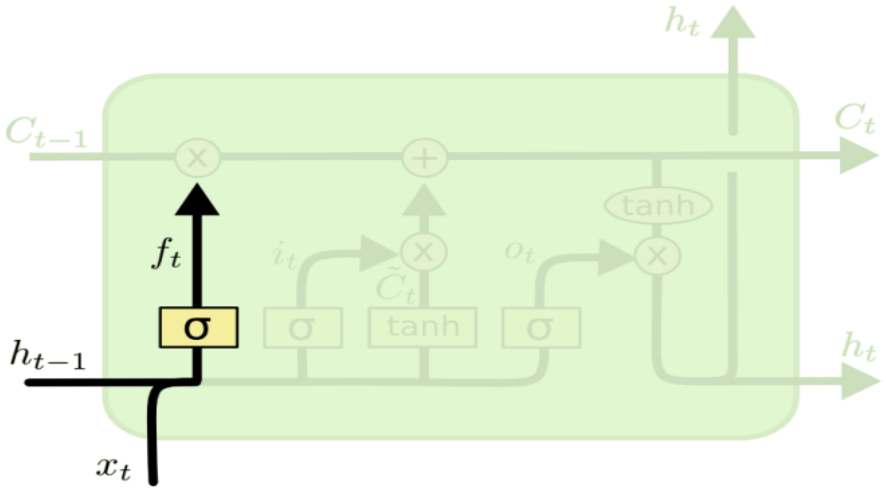


Output Gate

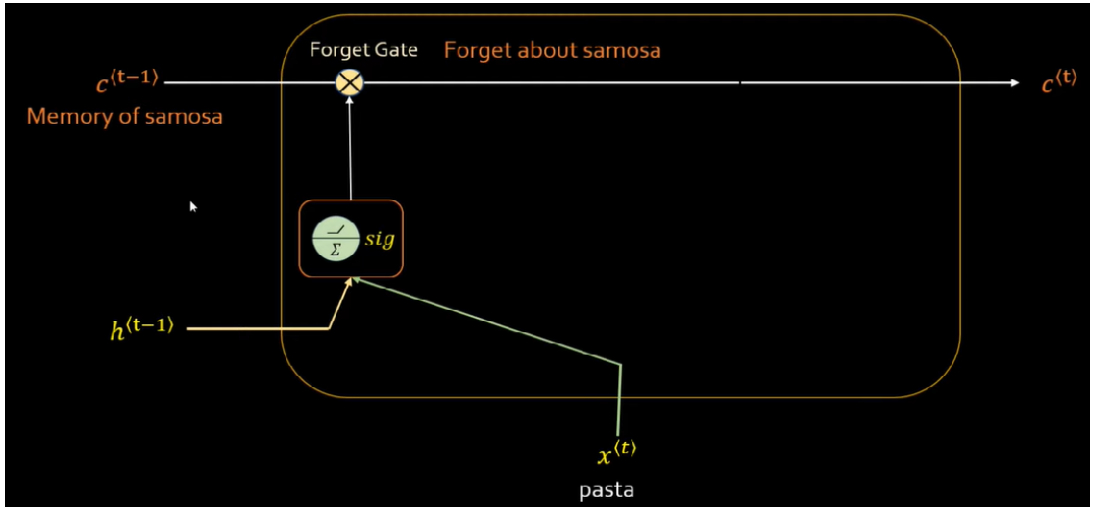
Input Gate

Gates in LSTM

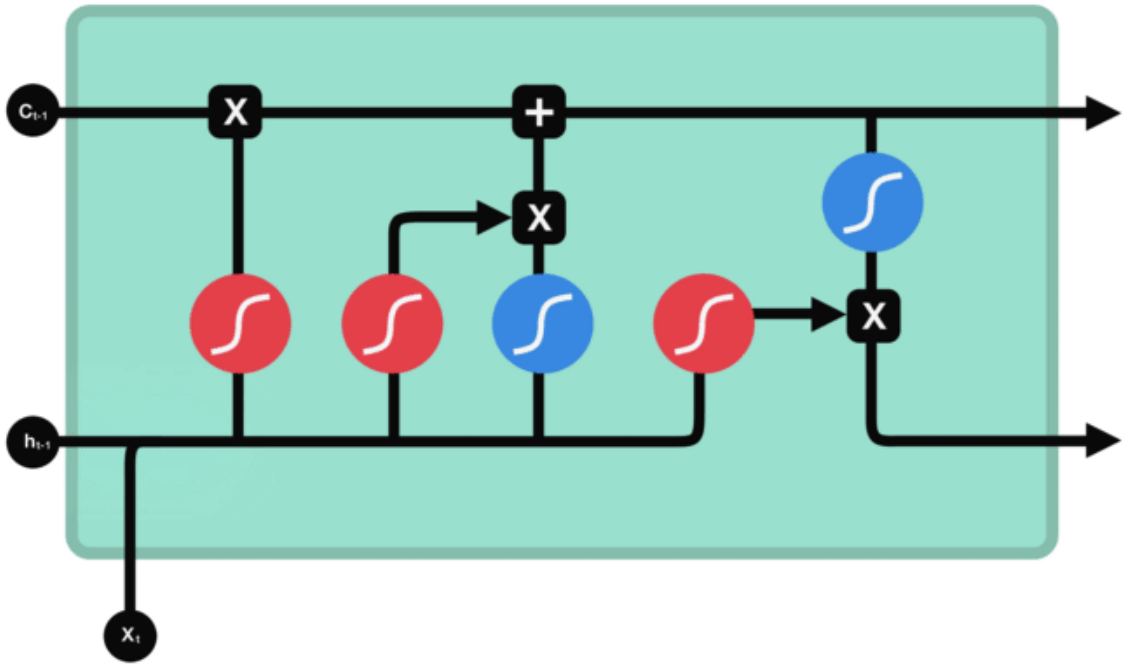
- Forget gate: decides what information should be thrown away or kept.
- The closer to 0 means to forget, and the closer to 1 means to keep.



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



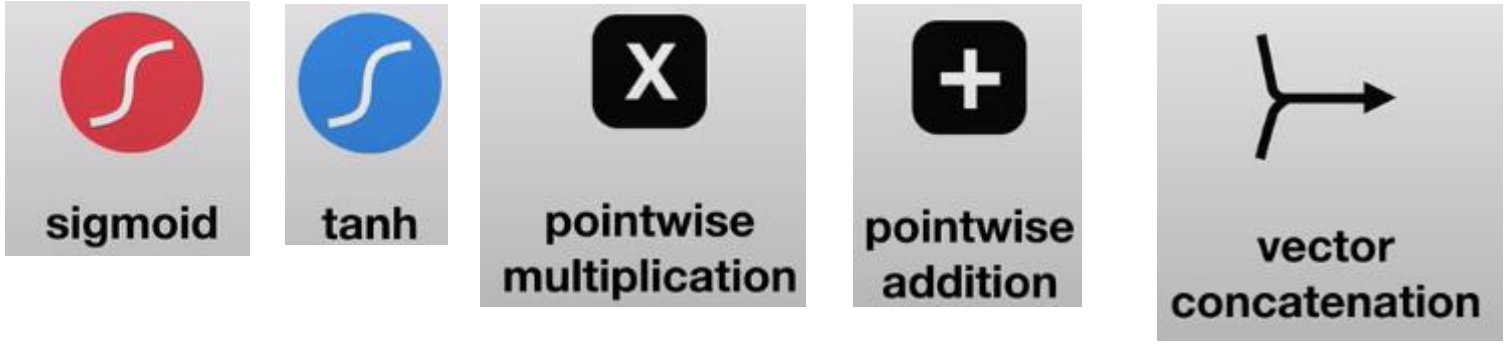
Forget Gate



C_{t-1} previous cell state

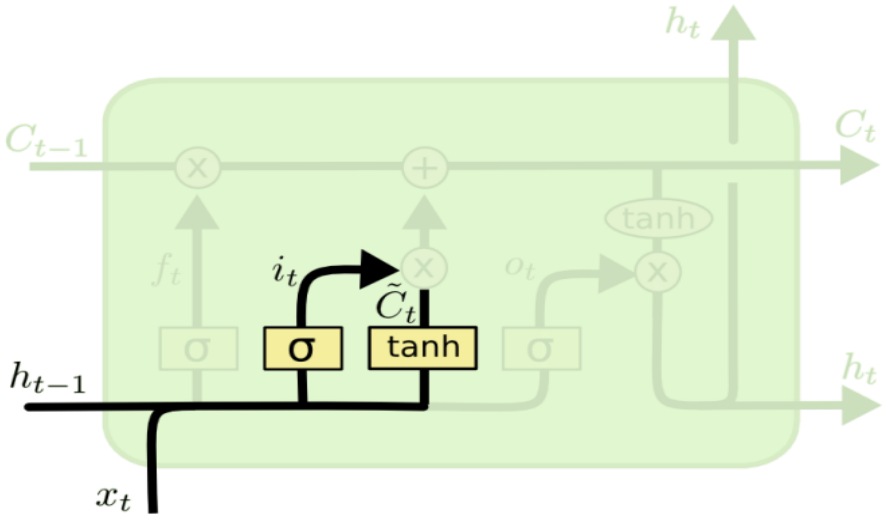
f_t forget gate output

- This gate decides what information should be thrown away or kept.
- Information from the previous hidden state and information from the current input is passed through the sigmoid function.
- Values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep.



Input Gate

- Used to compute the current cell state



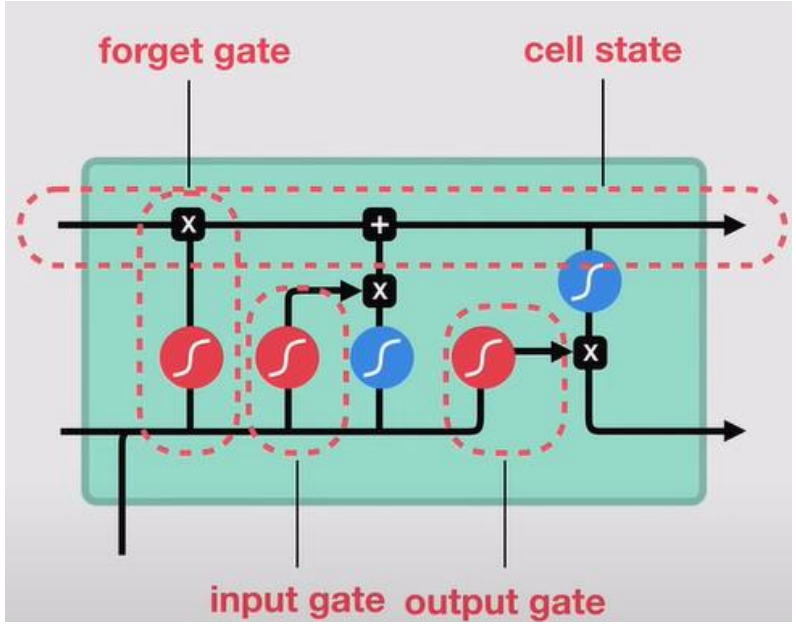
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

- First, we pass the previous hidden state and current input into a sigmoid function. That decides which values will be updated by transforming the values to be between 0 and 1. 0 means not important, and 1 means important.

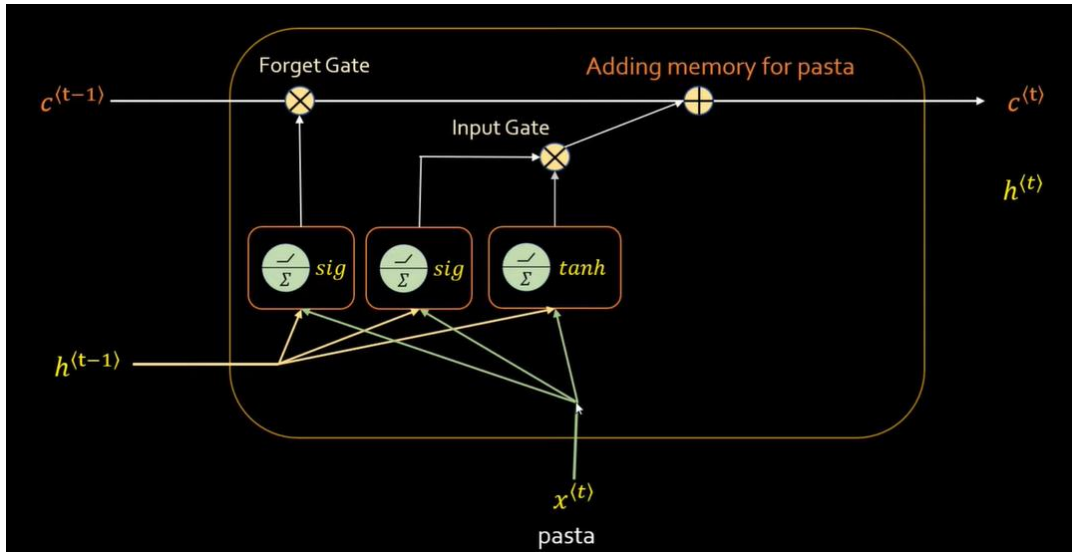
- You also pass the hidden state and current input into the tanh function to squish values between -1 and 1 to help regulate the network. Then you multiply the tanh output with the sigmoid output. The sigmoid output will decide which information is important to keep from the tanh output.

Input Gate

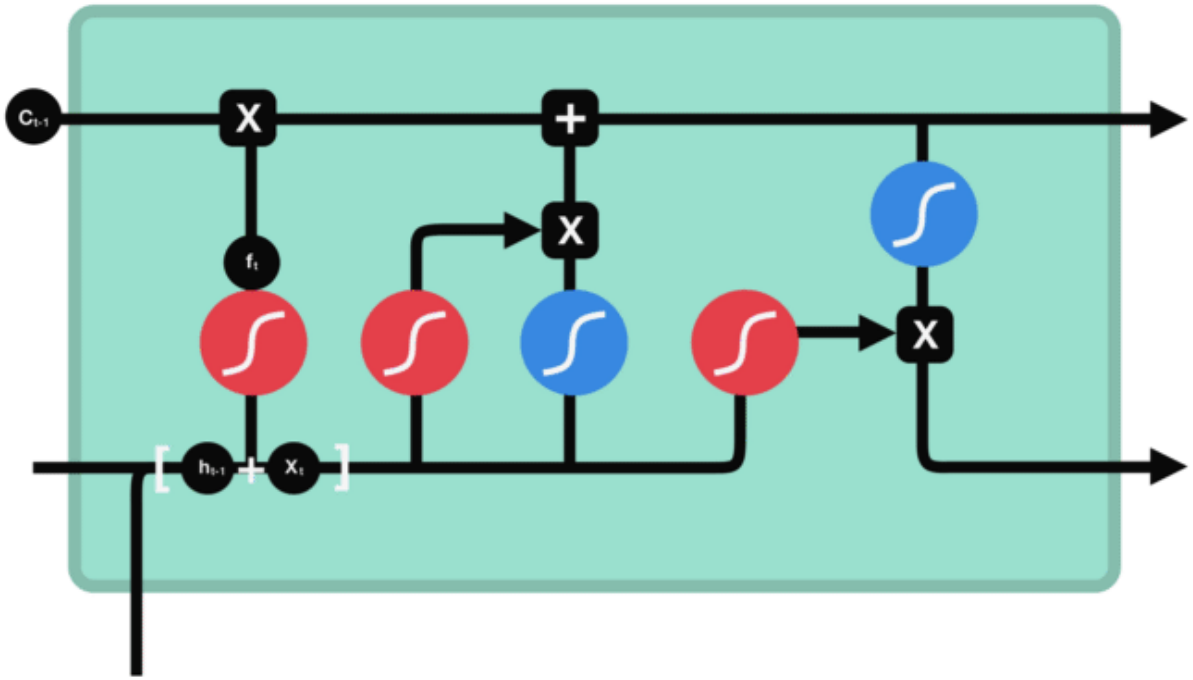


a. A sigmoid layer (the “input gate layer”) that decides which values to update.

b. A tanh layer (which creates a vector of new candidate values to add to the cell state).



Input Gate



C_{t-1} previous cell state

f_t forget gate output

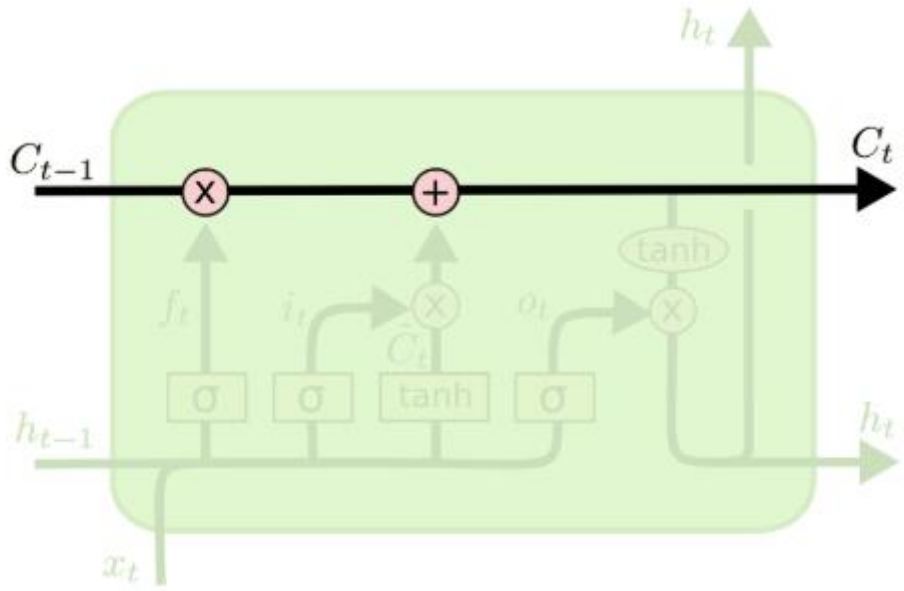
i_t input gate output

$\tilde{C}_t$ candidate

Updating Cell State

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

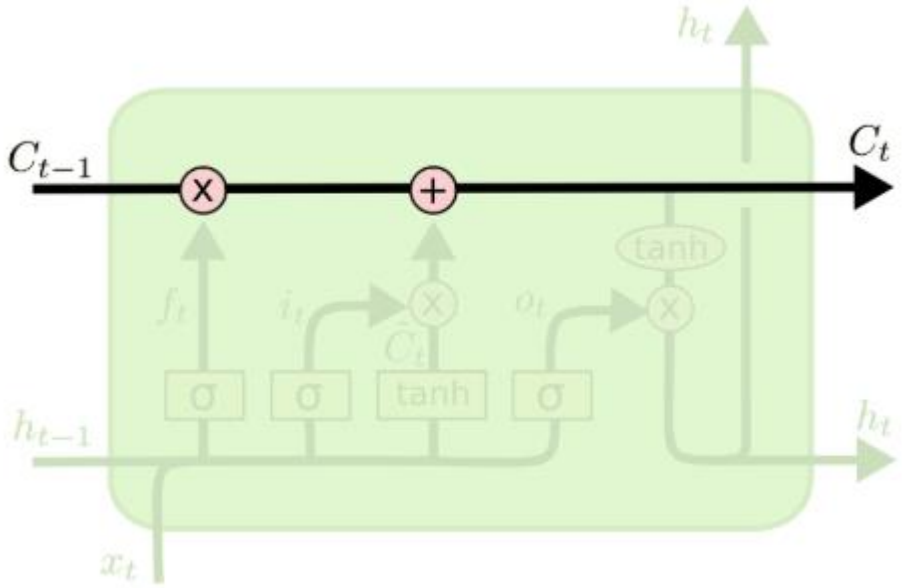
- The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.



Updating Cell State

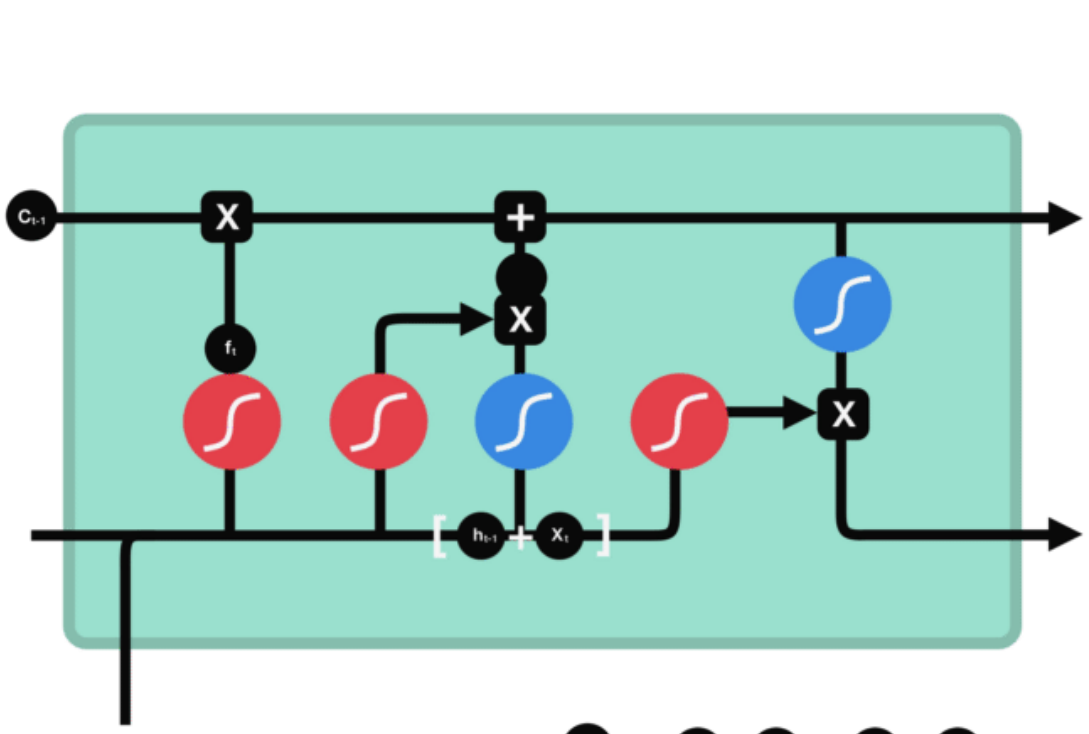
- First, the cell state gets pointwise multiplied by the forget vector.

- updates the cell state to new values that the neural network finds relevant.



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

Updating Cell State



C_{t-1} previous cell state

f_t forget gate output

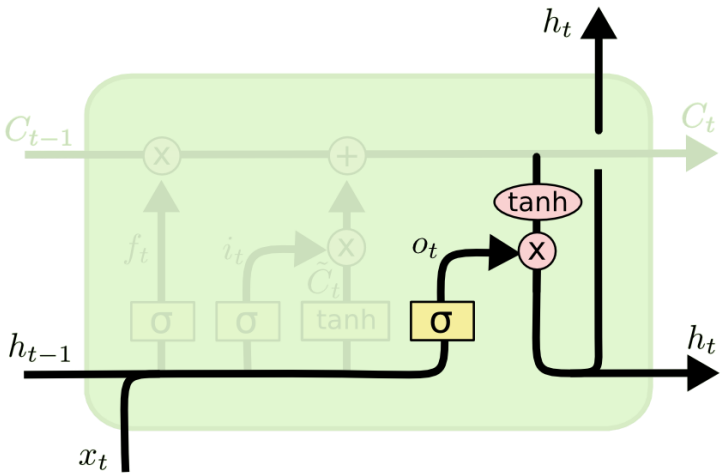
i_t input gate output

$\tilde{C}_t$ candidate

C_t new cell state

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

Output Gate

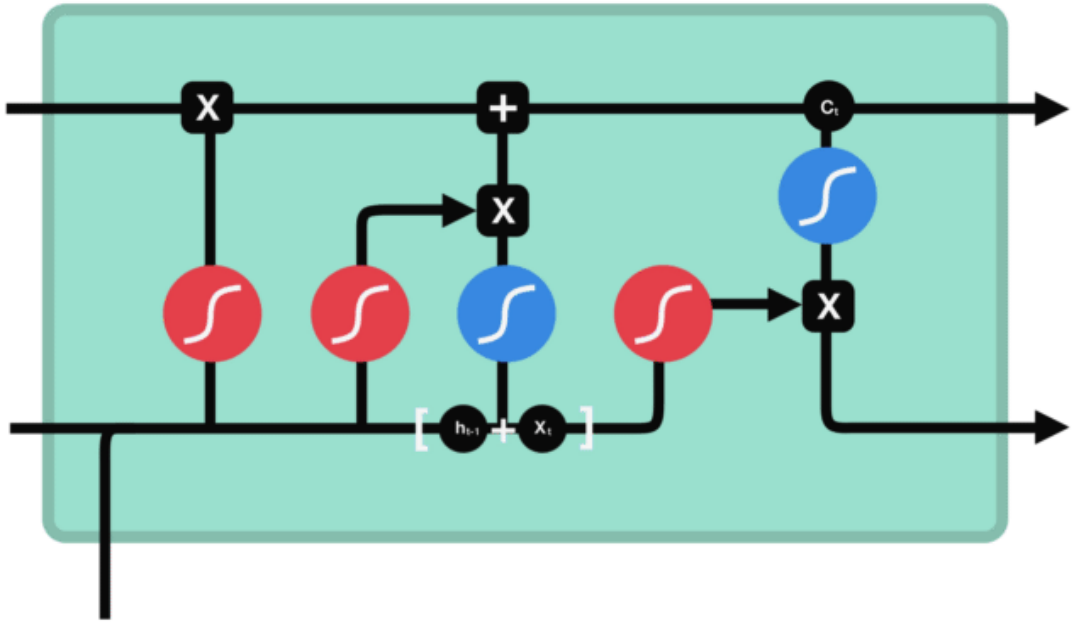


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

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

- 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.

Output Gate



C_{t-1} previous cell state

f_t forget gate output

i_t input gate output

$\tilde{C}_t$ candidate

C_t new cell state

o_t output gate output

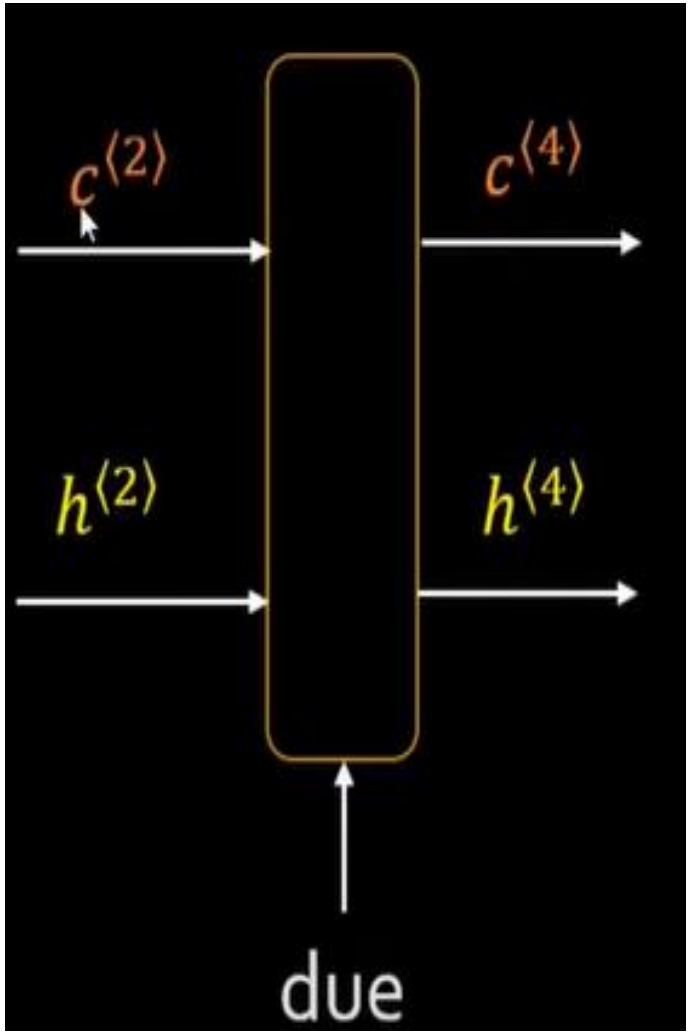
h_t hidden state

What is the difference between cell state and hidden state

1. First, the previous hidden state and the current input get concatenated. We'll call it *combine*.
2. *Combine* get's fed into the forget layer. This layer removes non-relevant data.
4. A candidate layer is created using *combine*. The candidate holds possible values to add to the cell state.
3. *Combine* also get's fed into the input layer. This layer decides what data from the candidate should be added to the new cell state.
5. After computing the forget layer, candidate layer, and the input layer, the cell state is calculated using those vectors and the previous cell state.
6. The output is then computed.
7. Pointwise multiplying the output and the new cell state gives us the new hidden state.

What is the difference between cell state and hidden state??????????

Long Short Term Memory

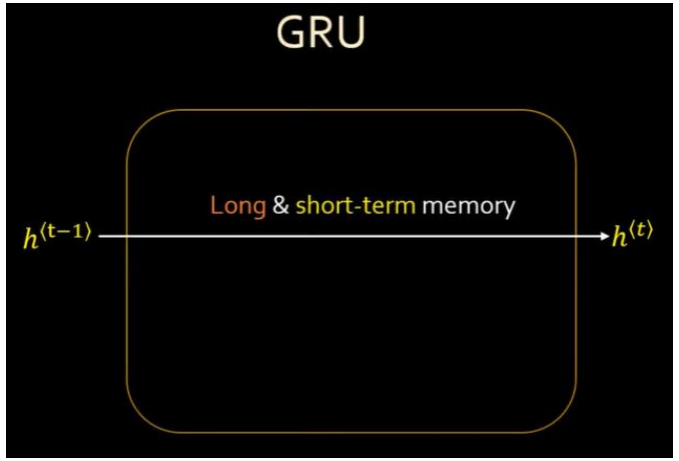


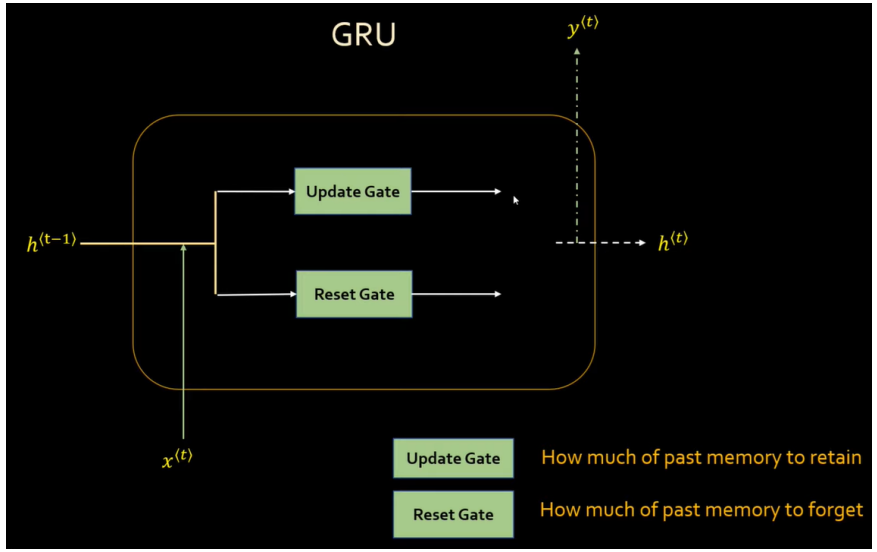
- The **cell state** is the **long-term memory** of the LSTM.
- It carries information across multiple time steps, allowing the network to remember important features over long sequences.
- The cell state is modified using **forget, input, and output gates**, which control what information should be retained, updated, or removed.
- It helps in mitigating the vanishing gradient problem by allowing gradients to flow more easily through time.

Short Term Memory

- The hidden state is the short-term memory of the LSTM.
- It is the output of the current time step and is used to make predictions.
- The hidden state is influenced by the cell state but is further processed through a tanh activation function to regulate its values.
- It acts as an interface between different time steps and is used for further computations in the network.

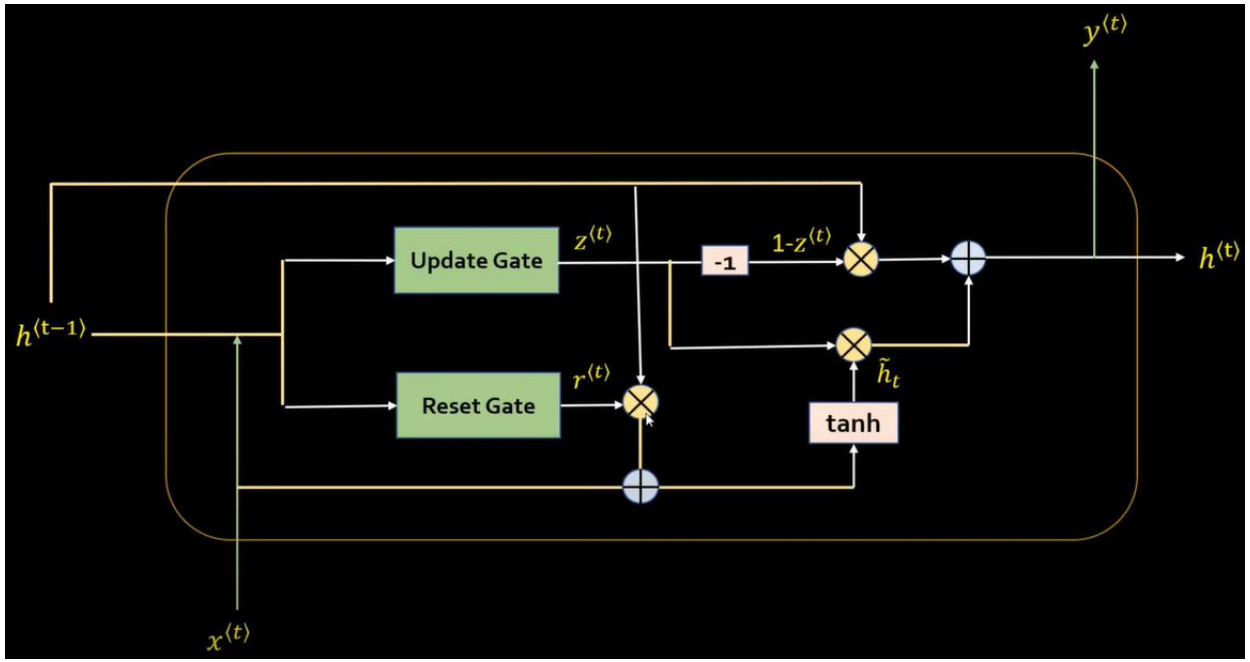
Gated Recurrent Unit





- Capable of learning long-term dependencies using mechanisms called “gates.”
- It only has two gates, a reset gate and 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.
- The reset gate is another gate is used to decide how much past information to forget.

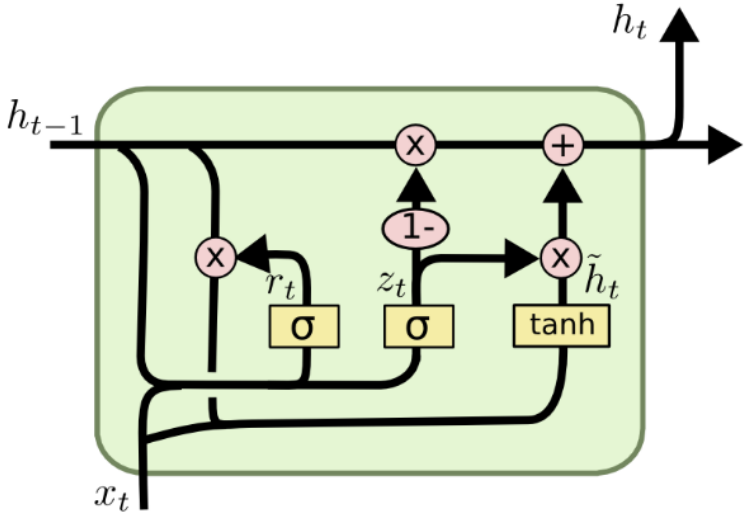
Gated Recurrent Unit



- Capable of learning long-term dependencies using mechanisms called “gates.”
- It only has two gates, a reset gate and 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.
- The reset gate is another gate is used to decide how much past information to forget.

Gated Recurrent Unit

- Capable of learning long-term dependencies using mechanisms called “gates.”
- It only has two gates, a reset gate and 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.
- The reset gate is another gate is used to decide how much past information to forget.



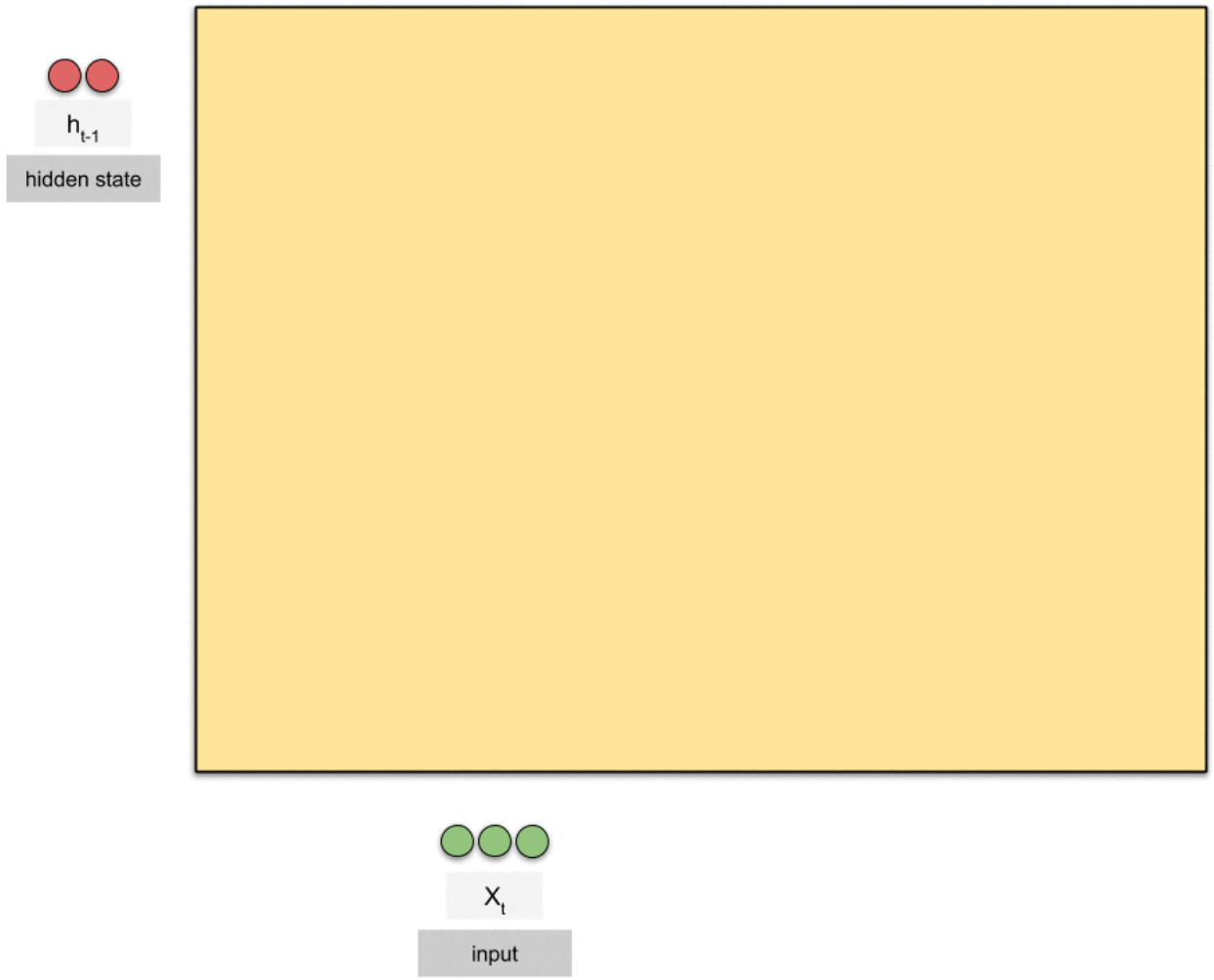
$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

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

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

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

Gated Recurrent Unit



Comparative Analysis of RNN, LSTM, GRU

	Simple RNN	LSTM	GRU
Model Complexity	Low	High	Moderate
Key Characteristics	Easier to train, Less computational Resources	Model long term dependency, Extraction of contextual Information	Model long term dependency, Extraction of contextual Information
Shortcomings	Vanishing gradient problem	High hidden layer complexity	Higher complexity than simple RNN