Recurrent Neural Network-(RNN)

What is Temporal Data?

Temporal data refers to data that is **associated with time**, meaning it has a time dimension. This type of data represents how something **changes over time**, and each data point is timestamped.

- **Time-stamped:** Each data point is associated with a specific time. This can be a precise timestamp (e.g., 2024-06-07 12:34:56) or a more general period (e.g., daily, monthly, yearly).

- **Sequential:** Temporal data is ordered by time. The sequence of data points is crucial for understanding trends, patterns, and relationships over time.

- **Dynamic:** It reflects changes over time, allowing for the analysis of how a phenomenon evolves.

Examples of Temporal Data for RNNs

Recurrent Neural Networks (RNNs) are particularly **well-suited for handling temporal data**, which involves sequences where the order and context of data points are important.

- Stock Prices
- Weather Data
- Energy Consumption
- Sensor Data
- Text Sequences
- Speech Recognition
- Transaction Records
- Loan Payment Histories
- Patient Monitoring
- Medical Records
- Video Frames

Applications of RNNs

Application of RNNs in Natural Language Processing (NLP) and Finance.

- **Natural Language Processing (NLP):**
  - **Language Modeling:** Generating text based on previous words.
  - **Machine Translation:** Translating text from one language to another.
  - **Speech Recognition:** Converting spoken language into text.
  - **Sentiment Analysis:** Analyzing text to determine sentiment.
  - **Named Entity Recognition (NER):** Identifying and classifying proper nouns in text.

- **Finance:**
  - **Stock Market Prediction:** Forecasting future stock prices based on historical data.
  - **Risk Management:** Evaluating the risk level of loans or investments over time.
  - **Behavioral Data:** Patterns of user activity on websites over days, months, or years.

Types of Recurrent Neural Networks (RNNs):

1. Basic RNNs
2. Long Short-Term Memory (LSTM)
3. Gated Recurrent Units (GRUs)
4. Bidirectional RNNs (BiRNNs)
5. Deep (Stacked) RNNs
6. Echo State Networks (ESNs)
7. Peephole LSTM
8. Independently Recurrent Neural Network (IndRNN)
9. Attention Mechanisms and Transformers

Recurrent Neural Networks

- **Basic RNN (Vanilla RNN):** The simplest form of RNNs that use recurrent connections to process sequential data by maintaining a hidden state that is updated at each time step.
  - **Use Cases:** Simple sequential data like time series prediction, and basic text generation.
  - **Limitations:** Struggles with long-term dependencies due to vanishing and exploding gradient problems.

- **Long Short-Term Memory (LSTM):** Includes memory cells and gates (input, forget, and output gates) to control the flow of information.
  - **Use Cases:** Tasks requiring long-term memory, such as language modeling, speech recognition, and machine translation.
  - **Advantages:** Better at capturing long-term dependencies compared to basic RNNs.

- **Gated Recurrent Unit (GRU):** Similar to LSTM but with a simpler architecture using only two gates (reset and update gates).
  - **Use Cases:** Similar to LSTM but more efficient due to fewer parameters.
  - **Advantages:** Computationally more efficient than LSTMs, with comparable performance.

- **Bidirectional RNN (BiRNN):** Consists of two RNNs running in parallel, one processing the sequence from start to end and the other from end to start.
  - **Use Cases:** Natural language processing tasks where context from both past and future is important, such as named entity recognition and machine translation.
  - **Advantages:** Provides context from both directions, improving performance on tasks requiring understanding of the entire sequence.

Recurrent Neural Networks

- **Deep (Stacked) RNN:** Multiple RNN layers stacked on top of each other, allowing the model to learn more complex representations.
  - **Use Cases:** Complex sequential tasks like advanced language models, and video analysis.
  - **Advantages:** Increased capacity to learn hierarchical features and patterns in data.

- **Echo State Networks (ESNs):** Consists of a large, fixed recurrent layer (reservoir) with randomly initialized weights. Only the output layer is trained.
  - **Use Cases:** Time series prediction, dynamical systems modeling.
  - **Advantages:** Simpler training process since only the output weights are learned.

Recurrent Neural Networks

- **Peephole LSTM:** A variant of LSTM where the gates also receive information from the cell state.
  - **Use Cases:** Similar to LSTM but potentially more accurate for certain tasks requiring fine-grained temporal information.
  - **Advantages:** Allows the network to access the cell state more directly, potentially improving performance.

- **Independently Recurrent Neural Network (IndRNN):** Each neuron in a layer has its recurrent connection, allowing for better handling of long sequences.
  - **Use Cases:** Tasks requiring very long-term dependencies.
  - **Advantages:** Better handling of long sequences and mitigating gradient issues.

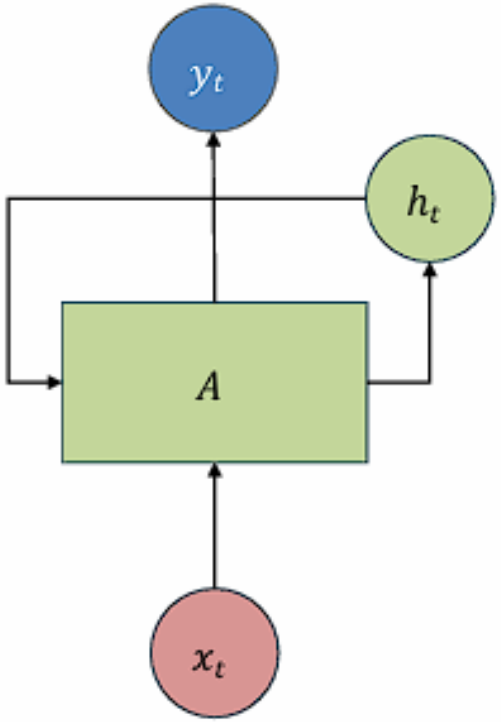
- **Attention Mechanisms and Transformers:** Not traditional RNNs, but worth mentioning as they handle sequence data using attention mechanisms to capture dependencies without recurrence.
  - **Use Cases:** Machine translation, text summarization, and various NLP tasks.
  - **Advantages:** Overcomes limitations of RNNs in handling long-term dependencies and parallelizes computation.

Attention Is All You Need

Basic Architecture of RNNs

Basic Architecture of RNNs

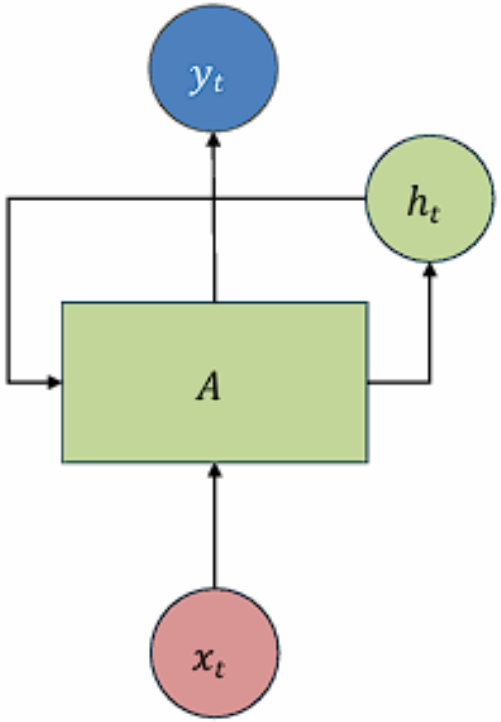
An RNN, or Recurrent Neural Network, is a type of artificial neural network designed to **recognize patterns** in **sequences of data** such as text, genomes, handwriting, or spoken words. RNNs have a unique feature: they use their internal state (memory) to process sequences of inputs.



1. x_t (Input):
   - This is the input at time step t. It represents the current data point being fed into the RNN unit.
2. h_t (Hidden State):
   - This is the hidden state at time step t. It captures information from previous time steps and helps in maintaining context over the sequence. The hidden state h_t is computed based on the current input x_t and the previous hidden state h_{t-1}.

Basic Architecture of RNNs

An RNN, or Recurrent Neural Network, is a type of artificial neural network designed to **recognize patterns** in **sequences of data** such as text, genomes, handwriting, or spoken words. RNNs have a unique feature: they use their internal state (memory) to process sequences of inputs.



3. y_t (Output):

   - This is the output at time step t. It is generated based on the current hidden state h_t.

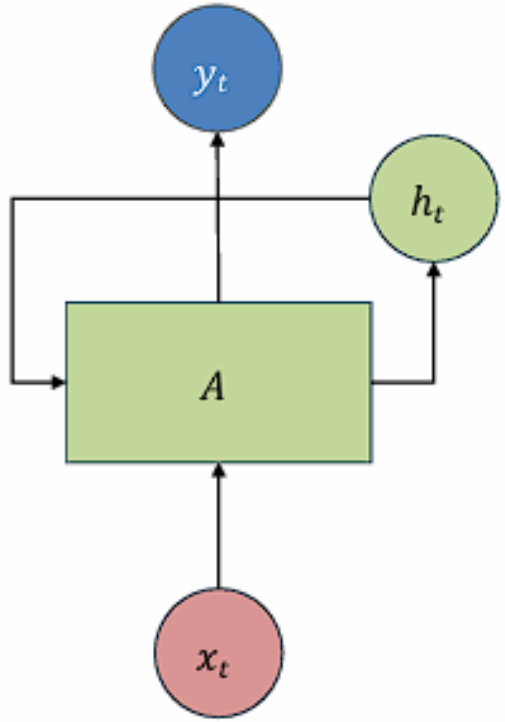
4. A (RNN Cell):

   - The RNN cell, denoted by A, performs the computations to update the hidden state and generate the output. It takes the current input x_t and the previous hidden state h_{t-1} to produce the new hidden state h_t and the output y_t.

Steps: Recurrent Neural Networks

Main Steps in Training an RNN

1. **Initialization:** Initialize weights, biases, and hidden states.
2. **Forward Pass:** Compute hidden states and outputs for each time step.
3. **Loss Calculation:** Calculate the loss over the entire sequence.
4. **Backward Pass (BPTT):** Unroll the RNN, and compute gradients with respect to all parameters.
5. **Parameter Update:** Update weights and biases using the computed gradients.
6. **Iteration:** Repeat the process for multiple epochs.



Difference Between ANNs & RNNs

Feature	ANNs	RNNs
Architecture	Layers with unidirectional flow, no cycles.	Loops in network, maintaining internal state.
Data Handling	Processes inputs independently, no memory.	Maintains memory for sequential data processing.
Task Suitability	Good for independent data points like classification.	Ideal for sequential tasks like speech recognition.
Training Complexity	Simpler and faster to train.	More complex due to temporal dependencies.

Unfolding: Recurrent Neural Networks

- "Unfolded" RNN unit: sequence of copies of the **same** unit (= same weights)
- Each unit passes hidden state as additional input to successor

➔ Previous input can influence current output

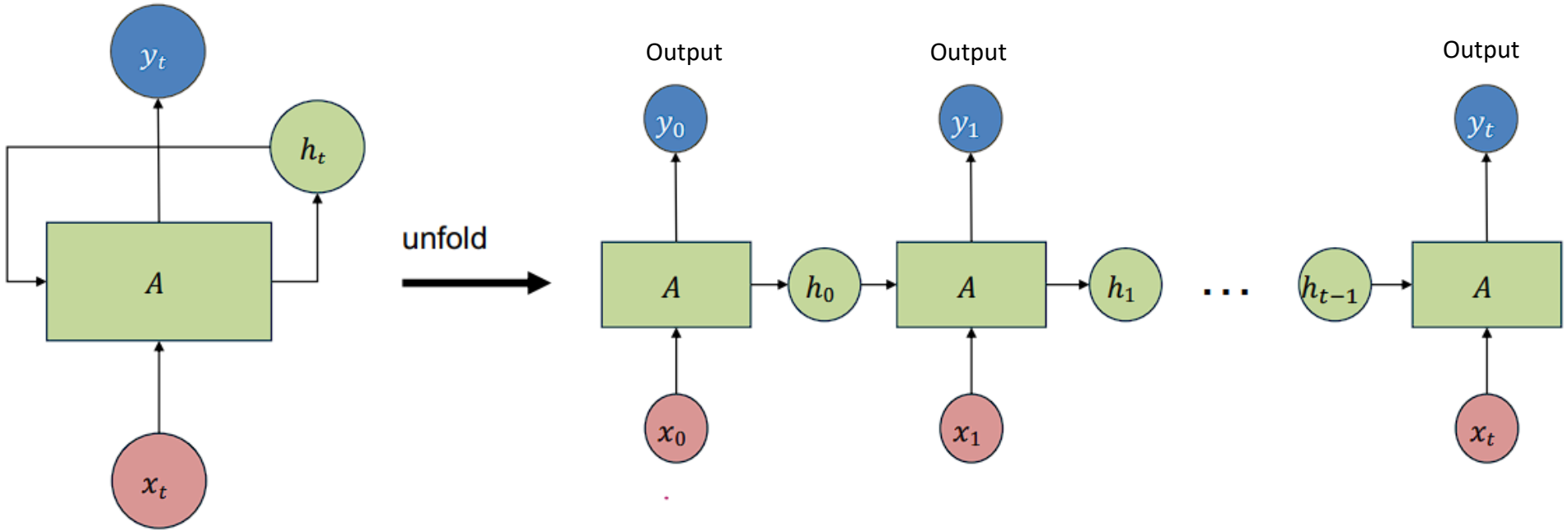


Fig: The cyclic structure of the RNN from the left side to the right side.

Recurrent Neural Networks

RNN: Unrolling

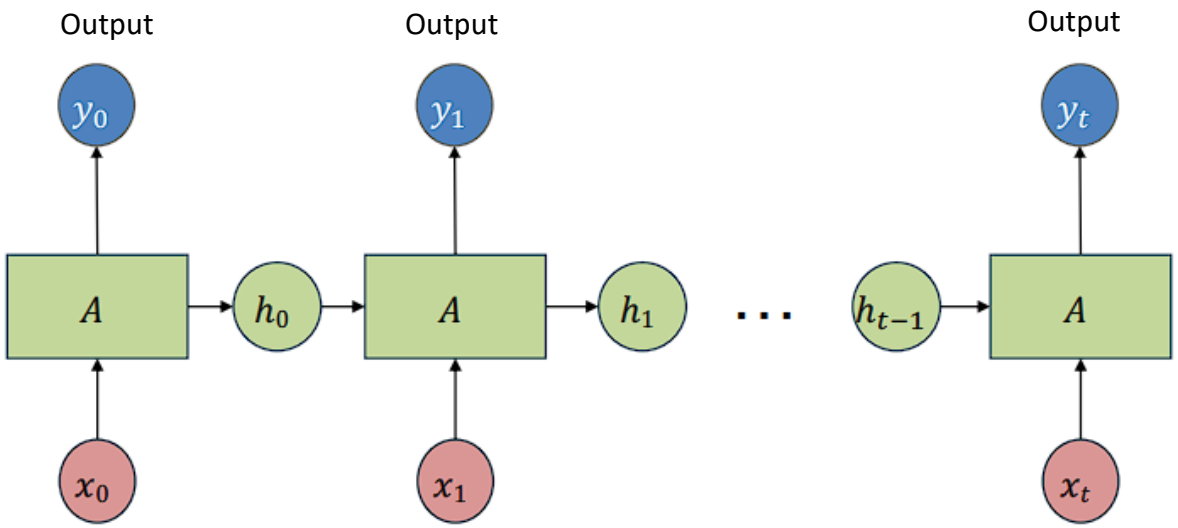
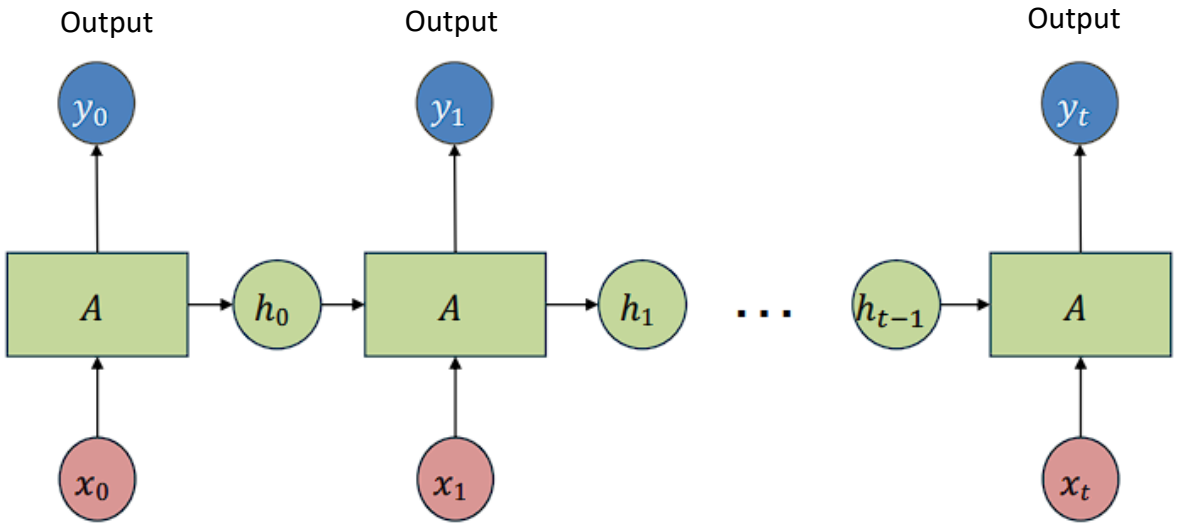


Fig: The cyclic structure of the RNN.

1. **Unfolded Representation**:
   - An "unfolded" RNN shows the network structure across multiple time steps, making it easier to understand how data flows through the network over time.
   - Each unit A in the unfolded sequence is essentially the same RNN cell, repeated at each time step with shared weights.
2. **Components**:
   - **Input** x_t: The input at each time step t.
   - **Hidden State** h_t: The hidden state at each time step t.
   - **Output** y_t: The output at each time step t.
   - **RNN Cell** A: The same RNN cell is used at each time step with the same weights.

Let's Think -



- How do we update the hidden state, h_t?
- How do we combine the input and hidden state to calculate the output, y_t?

Recurrent Neural Networks

Mathematical Formulation

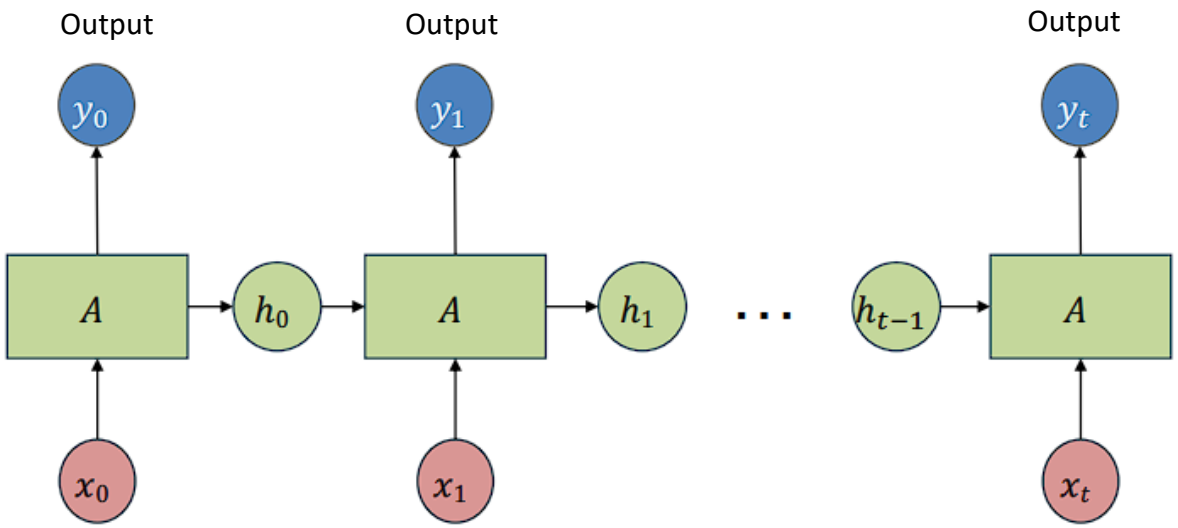


Fig: The cyclic structure of the RNN.

1. **Forward Pass:**

- At each time step t, the RNN cell A processes the input x_t and the previous hidden state h_{t-1} to produce the new hidden state h_t and the output y_t.
- The equations are:

Update Hidden State, $h_t = f(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$

Here, f is the activation function (ReLU/tanh), and g is another activation function (sigmoid/softmax for classification).

Output at Time Step t, $y_t = g(W_{hy}h_t + b_y)$

Here,

- W_{xh}: Input to hidden state
- W_{hh}: Hidden state to hidden state
- W_{hy}: Hidden state to output

Recurrent Neural Networks

Mathematical Formulation

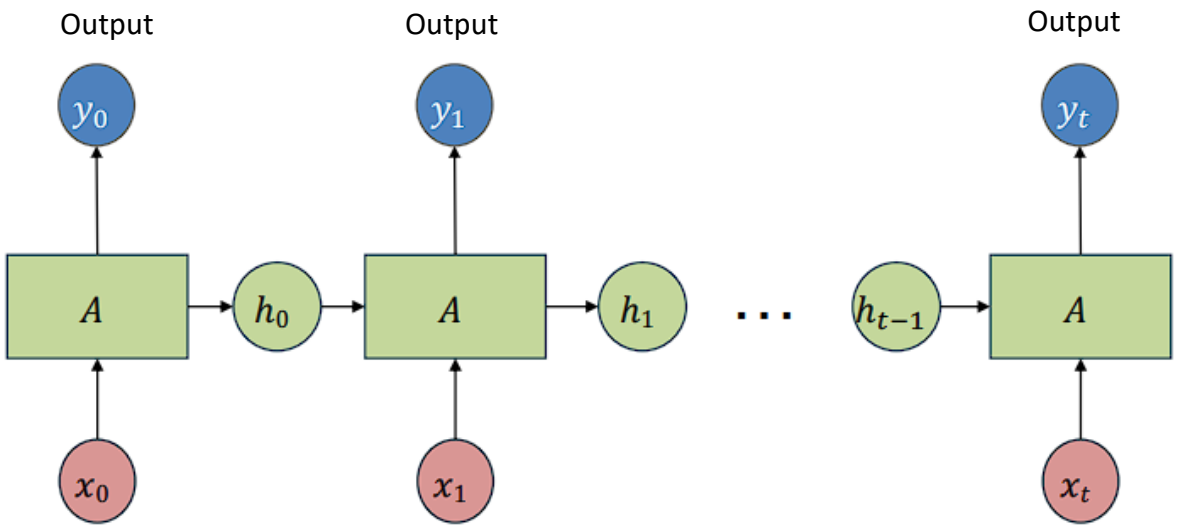


Fig: The cyclic structure of the RNN.

1. **Forward Pass:**

   - At each time step t, the RNN cell A processes the input x_t and the previous hidden state h_{t-1} to produce the new hidden state h_t and the output y_t.
   - The equations are:

   Update Hidden State, $h_t = f(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$

   Here, f is the activation function (ReLU/tanh), and g is another activation function (sigmoid/softmax for classification).

   Output at Time Step t, $y_t = g(W_{hy}h_t + b_y)$

2. **Sharing Weights:**

   - **Same Weights Across Time Steps**: The weights W_{xh}, W_{hh}, and W_{hy} are shared across all time steps. This means the same weight matrices and biases are used for each unit A at each time step.

Recurrent Neural Networks

Mathematical Formulation

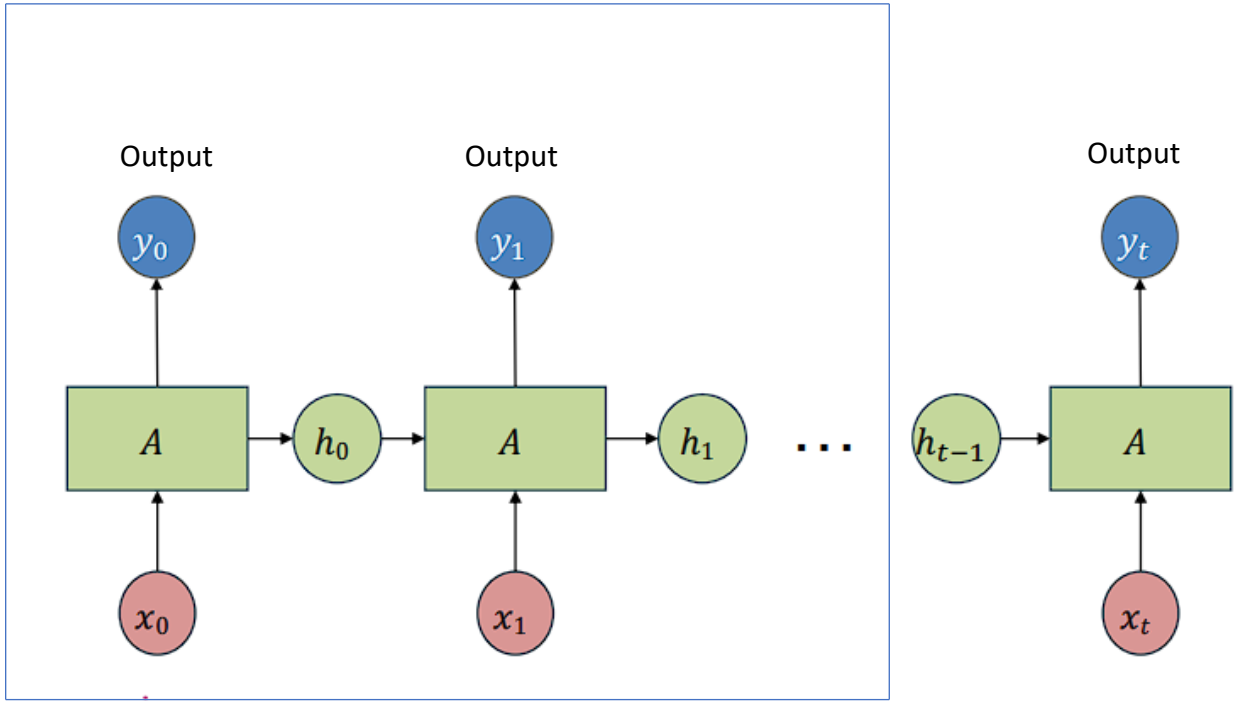


Fig: The cyclic structure of the RNN.

Time Step $t = 0$:

1. Compute the hidden state:
   This equation does not include the recurrent weight matrix W_{hh}

$$h_0 = f(W_{xh}x_0 + b_h)$$

2. Compute the output:

$$y_0 = g(W_{hy}h_0 + b_y)$$

Time Step $t = 1$:

1. Compute the hidden state:

$$h_1 = f(W_{xh}x_1 + W_{hh}h_0 + b_h)$$

2. Compute the output:

$$y_1 = g(W_{hy}h_1 + b_y)$$

Recurrent Neural Networks

Mathematical Formulation

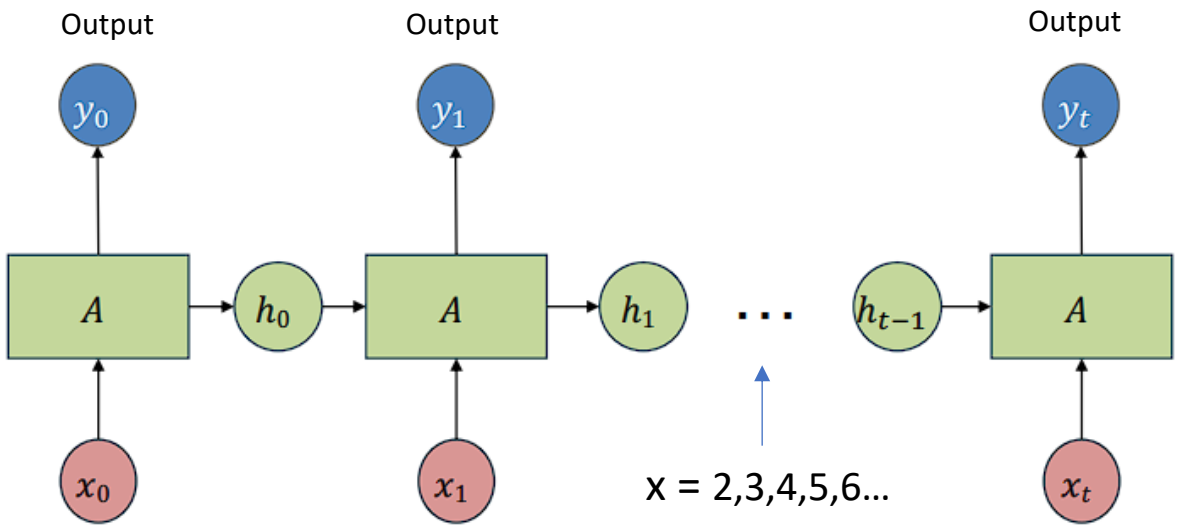


Fig: The cyclic structure of the RNN.

Time Step $t = 2$:

1. Compute the hidden state:

$$h_2 = f(W_{xh}x_2 + W_{hh}h_1 + b_h)$$

2. Compute the output:

$$y_2 = g(W_{hy}h_2 + b_y)$$

Time Step $t = 3$:

1. Compute the hidden state:

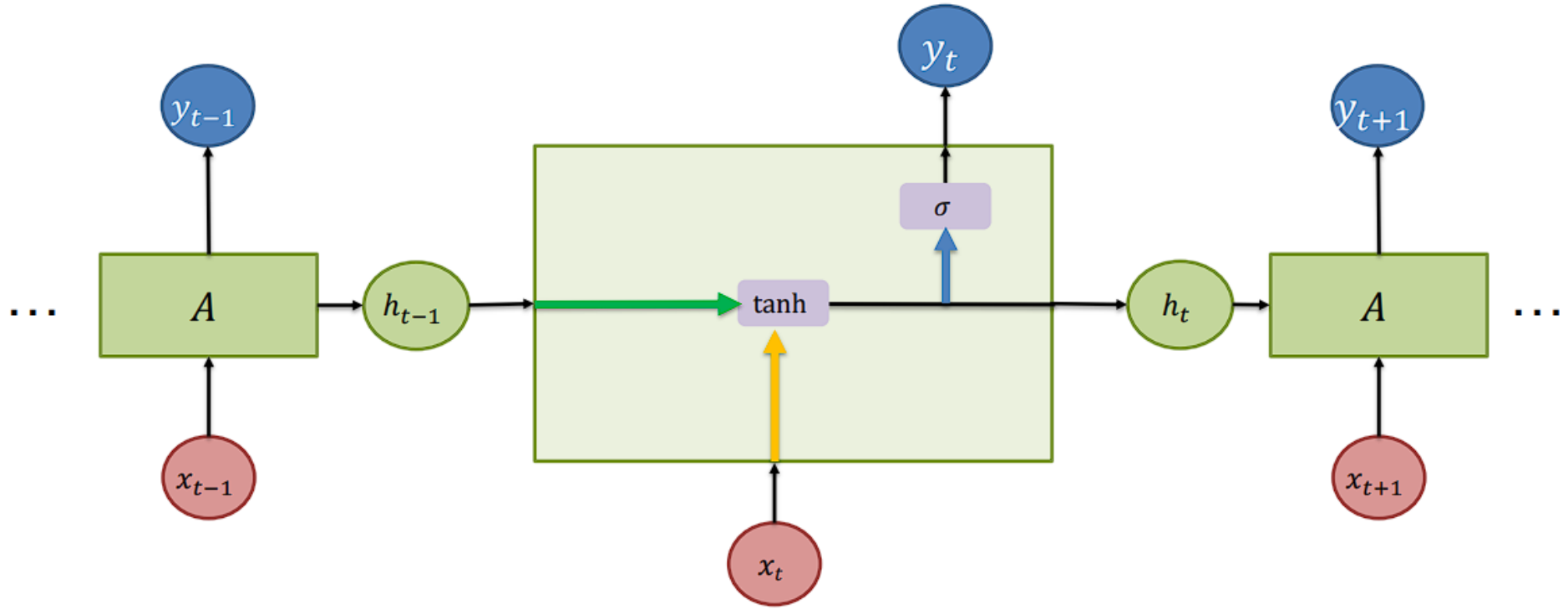
$$h_3 = f(W_{xh}x_3 + W_{hh}h_2 + b_h)$$

2. Compute the output:

$$y_3 = g(W_{hy}h_3 + b_y)$$

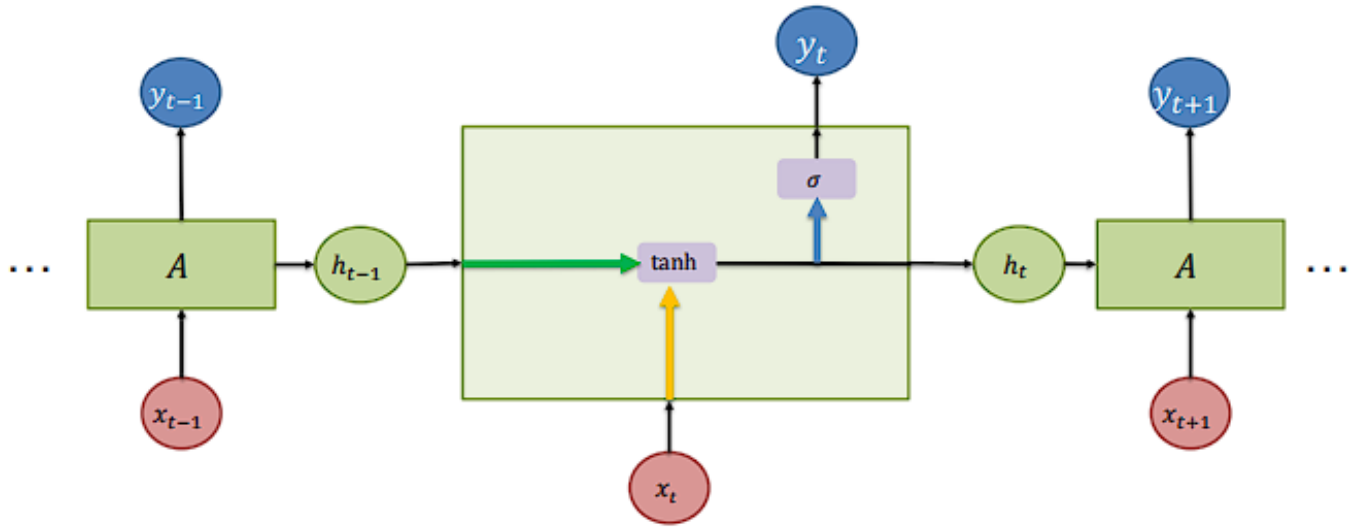
Recurrent Neural Networks

Mathematical Formulation



Recurrent Neural Networks

Mathematical Formulation



The behaviour of the RNN can be described as a **dynamical system** by the pair of non-linear matrix equations:

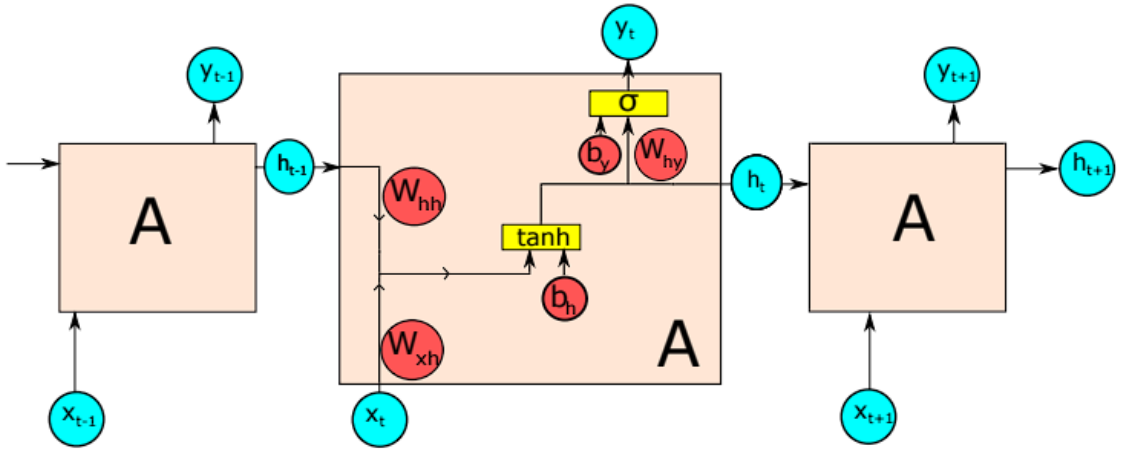
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = \sigma(W_{hy}h_t)$$

The order of the dynamical system corresponds to the dimensionality of the state h_t.

Question 01: How do we Update the Hidden State?

Question 01 - Solution



Update hidden state:

$$\mathbf{h}_t = \tanh\left(\mathbf{W}_{hh} \cdot \mathbf{h}_{t-1} + \mathbf{W}_{xh} \cdot \mathbf{x}_t + \mathbf{b}_h\right)$$

$\mathbf{W}_{hh}$: Weight matrix for previous hidden state $\mathbf{h}_{t-1}$

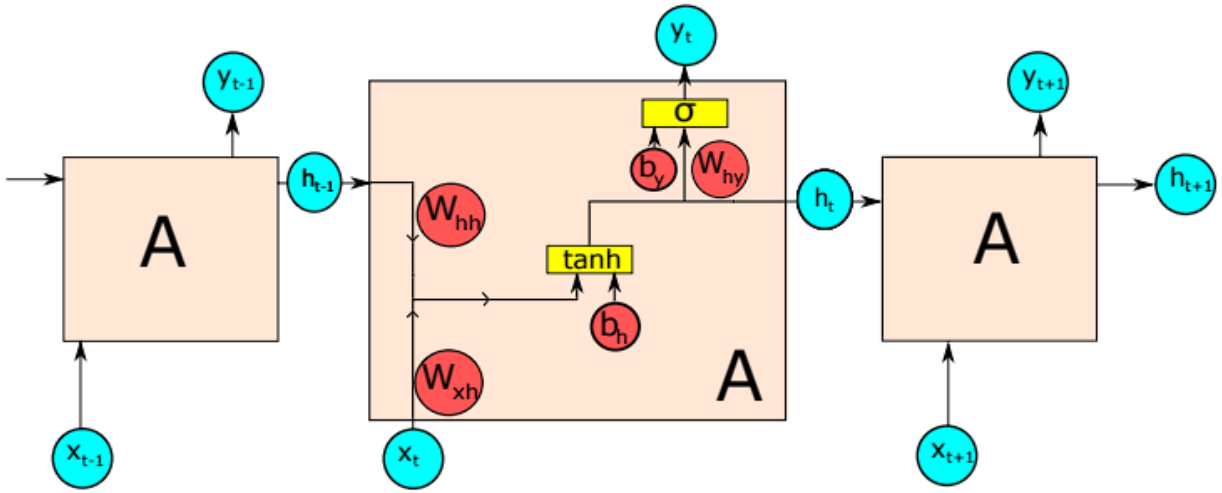
$\mathbf{W}_{xh}$: Weight matrix for current input $\mathbf{x}_t$

$\mathbf{b}_h$: Update bias

Note: The provided diagram and equations depict a vanilla RNN.

Question 02: How to Calculate the Output?

Question 01 - Solution



Output formula:

$$\mathbf{y}_t = \sigma\left(\mathbf{W}_{hy} \cdot \mathbf{h}_t + \mathbf{b}_y\right)$$

$\mathbf{W}_{hy}$: Weight matrix for current hidden state $\mathbf{h}_t$

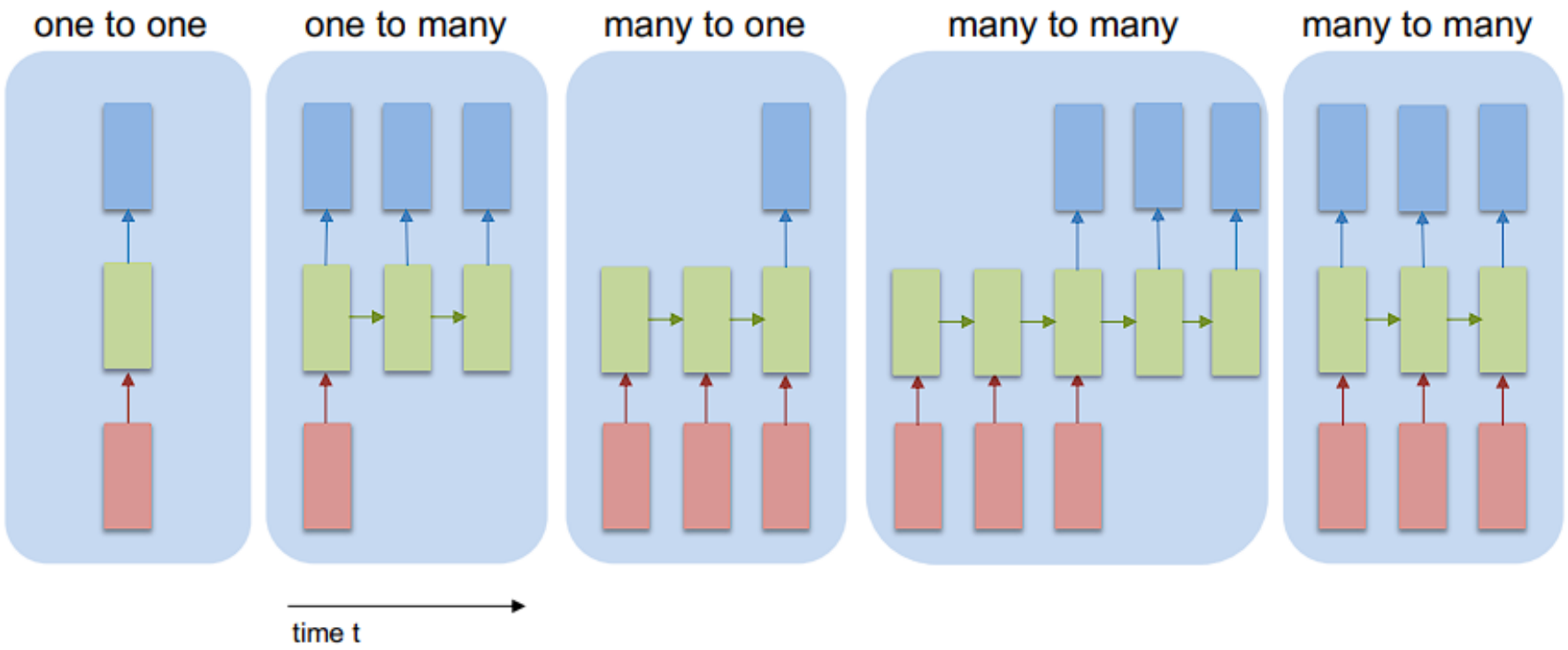
$\mathbf{b}_h$: Output bias

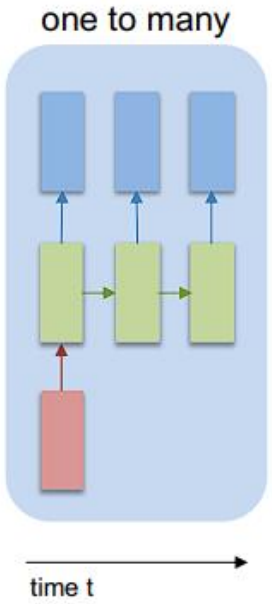
Note: The provided diagram and equations depict a vanilla RNN.

Types of Sequential Relationships and Corresponding RNN Architectures

Recurrent Neural Networks

Here are the main types of RNN architectures based on the type of sequential relationship:





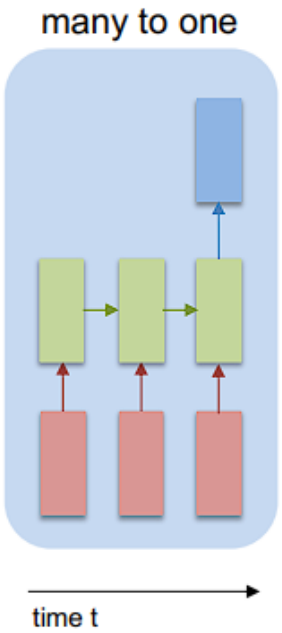
A typical example of a one to many problem is that of **image captioning**.

Input:

Output:

A cat playing with a ball

Recurrent Neural Networks



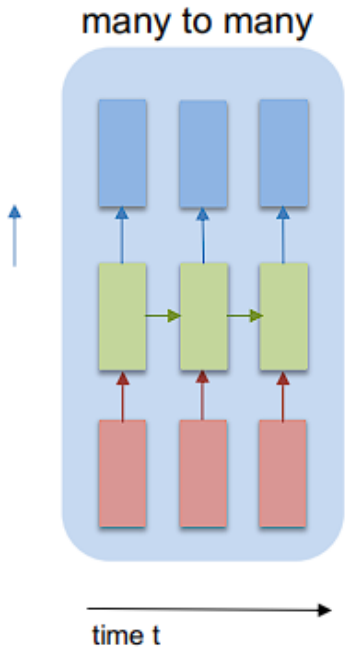
A typical example of a many to one problem is that of **sentiment analysis.**

Input:

Horrible	service	the	room	was	dirty

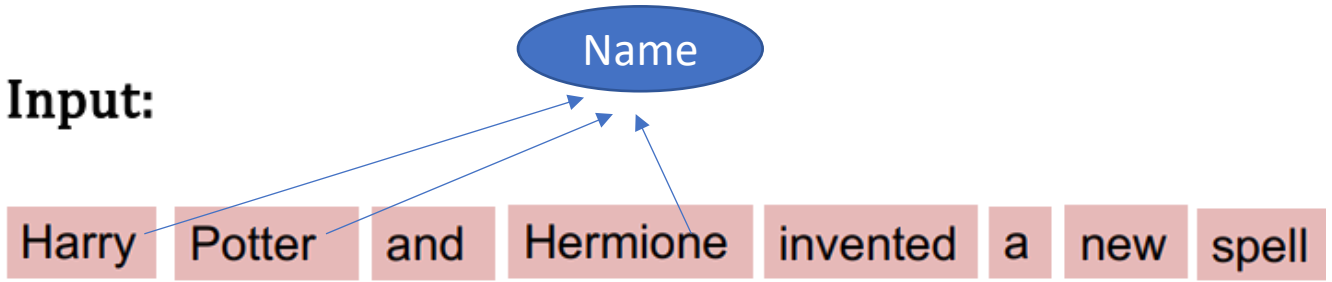
Output:

("negative")



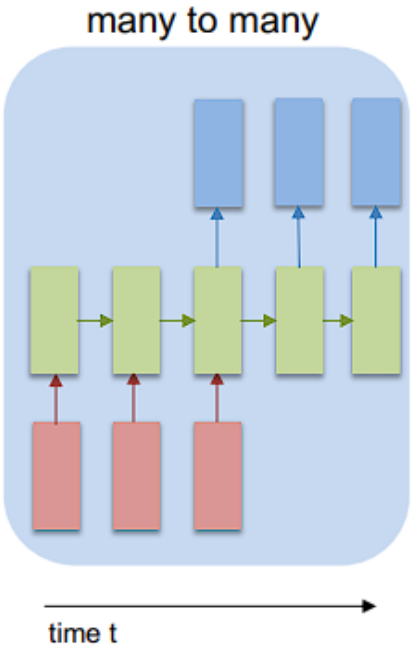
A typical example of a many to many problem is that of name entity recognition.

Input:



Output:

1	1	0	1	0	0	0	0



Another example of a many to many problem is that of **machine translation.**

Input: English

Horrible	service	the	room	was	dirty

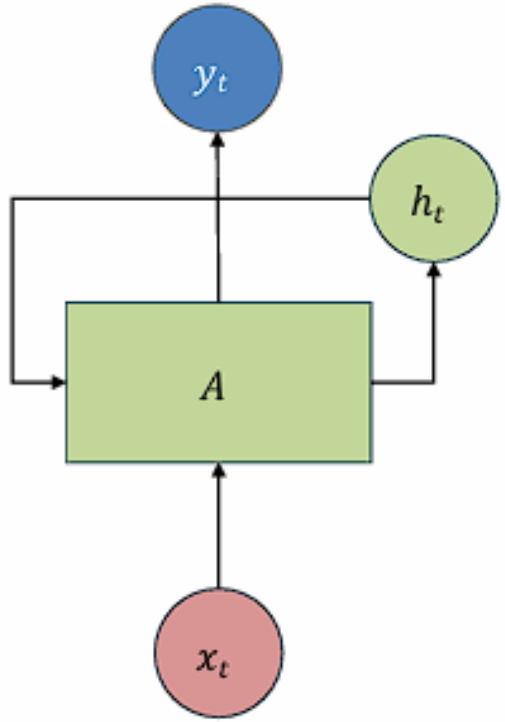
Output: Italian

Un	servizio	orribile	la	camera	era	sporca

Calculating Loss/Cost &
Backpropagation Through Time (BPTT)

Main Steps in Training an RNN

1. **Initialization:** Initialize weights, biases, and hidden states.
2. **Forward Pass:** Compute hidden states and outputs for each time step.
3. **Loss Calculation:** Calculate the loss over the entire sequence.
4. **Backward Pass (BPTT):** Unroll the RNN, and compute gradients with respect to all parameters.
5. **Parameter Update:** Update weights and biases using the computed gradients.
6. **Iteration:** Repeat the process for multiple epochs.



Loss/Cost Calculation

Compute Loss: Calculate the loss L over the entire sequence.

$$L = \sum_{t=1}^{T} L(y_t, \hat{y}_t)$$

where $\hat{y}_t$ is the true output at time step t and L is the loss function (e.g., mean squared error, cross-entropy).

Loss/Cost Calculation

- **Mean Squared Error (MSE)** for regression tasks:

$$L_t = \frac{1}{2}(\hat{y}_t - y_t)^2$$

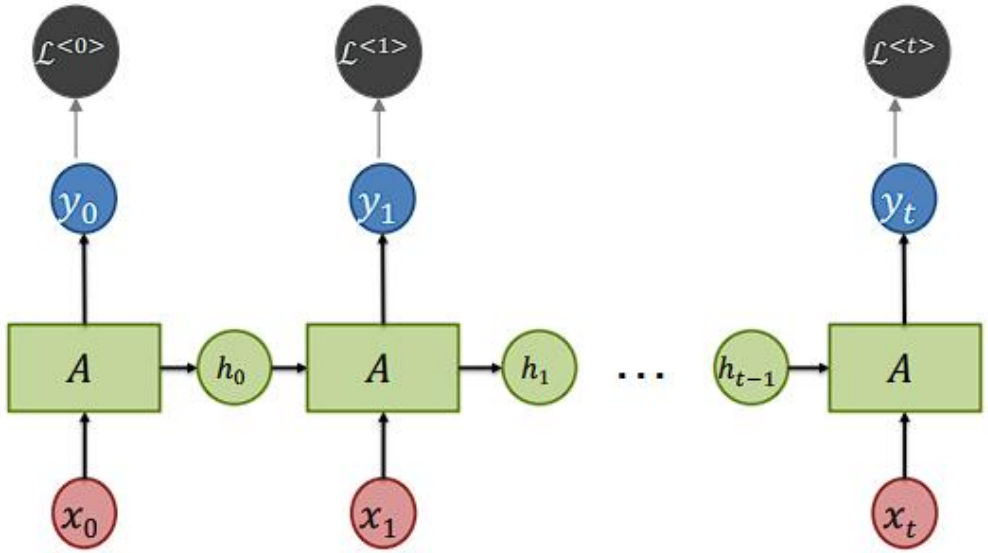
- **Cross-Entropy Loss** for classification tasks:

$$L_t = -\sum_{i=1}^{C} \hat{y}_t{}^{(i)} \log(y_t^{(i)})$$

where C is the number of classes, $\hat{y}_t{}^{(i)}$ is the true label for class i, and $y_t^{(i)}$ is the predicted probability for class i.

Loss/Cost Calculation

We compute the overall loss of our prediction $\hat{y}$ w.r.t. the true sequence y.



Loss/Cost Calculation

Sum of Individual Losses

- Sum the individual losses over all time steps to obtain the total loss for the sequence.

  For a sequence of length T:

$$L = \sum_{t=1}^{T} L_t$$

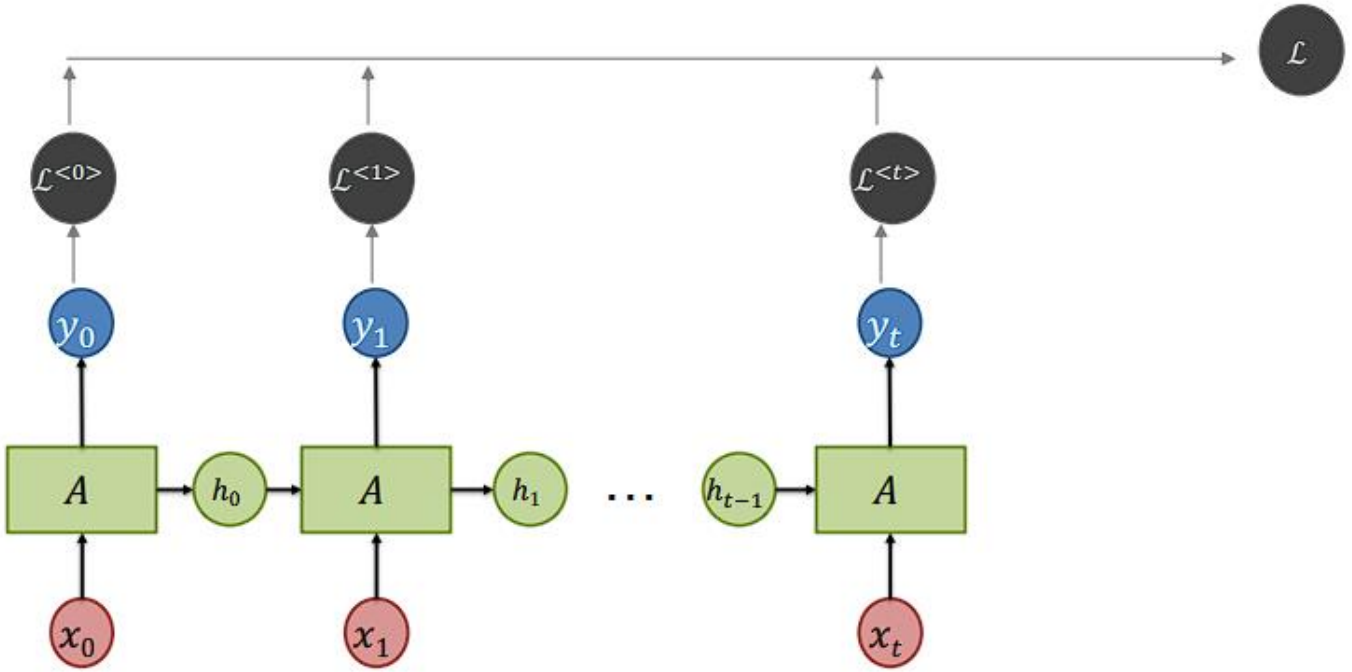
Suppose this is a regression problem, So total MSE =

$$L = \frac{1}{2}(\hat{y_1} - y_1)^2 + \frac{1}{2}(\hat{y_2} - y_2)^2 + \frac{1}{2}(\hat{y_3} - y_3)^2 \text{ + ...}$$

Loss/Cost Calculation

Sum Individual Losses

We compute the overall loss of our prediction $\hat{y}$ w.r.t. the true sequence y.



Backpropagation Through Time (BPTT)

Backpropagation Through Time (BPTT)

Backpropagation Through Time (BPTT) is a method specifically designed for training types of neural networks that have connections across **time steps**, such as Recurrent Neural Networks (RNNs) and their variants, including Long Short-Term Memory networks (LSTMs) and Gated Recurrent Units (GRUs). These architectures are *primarily used for sequence processing tasks where the output at a given time step depends on previous inputs.*

- Standard RNNs
- Long Short-Term Memory (LSTMs)
- Gated Recurrent Units (GRUs)
- Other Sequence Models

Backpropagation Through Time (BPTT)

- **Unroll the RNN**: Represent the RNN across all time steps as a computational graph.
- **Compute Gradients**: For each time step t from T to 1:
  - **Gradients for Output Weights**:

$$\frac{\partial L}{\partial W_{hy}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial W_{hy}}$$

  - **Gradients for Hidden Weights**:

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_{hh}}$$

  - **Gradients for Input Weights**:

$$\frac{\partial L}{\partial W_{xh}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_{xh}}$$

Here,

- W_{xh}: Input to hidden state
- W_{hh}: Hidden state to hidden state
- W_{hy}: Hidden state to output

Backpropagation Through Time (BPTT)

Gradients for Output Weights

- **Gradient of Loss**: The term $\frac{\partial L_t}{\partial y_t}$ measures how much the loss L_t would change if the output y_t changed. This is a direct sensitivity measure.

- **Gradient of Output**: The term $\frac{\partial y_t}{\partial W_{hy}}$ measures how much the output y_t would change if the weights W_{hy} changed. This is a structural dependency based on the network's architecture.

Backpropagation Through Time (BPTT)

- **Unroll the RNN**: Represent the RNN across all time steps as a computational graph.
- **Compute Gradients**: For each time step t from T to 1:
  - **Gradients for Output Weights**:

$$\frac{\partial L}{\partial W_{hy}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial W_{hy}}$$

  - **Gradients for Hidden Weights**:

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_{hh}}$$

  - **Gradients for Input Weights**:

$$\frac{\partial L}{\partial W_{xh}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_{xh}}$$

Here,

- W_{xh}: Input to hidden state
- W_{hh}: Hidden state to hidden state
- W_{hy}: Hidden state to output

Backpropagation Through Time (BPTT)

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_{hh}}$$

- **Gradient of Loss with respect to hidden state:** Measures how sensitive the loss is to changes in the hidden state.
- **Gradient of Hidden State with respect to hidden-to-hidden weight:** Measures how sensitive the hidden state is to changes in the recurrent weights.
- **Chain Rule:** Used to combine these sensitivities, giving the overall gradient of the loss with respect to the recurrent weights.

Backpropagation Through Time (BPTT)

- **Unroll the RNN**: Represent the RNN across all time steps as a computational graph.
- **Compute Gradients**: For each time step t from T to 1:
  - **Gradients for Output Weights**:

$$\frac{\partial L}{\partial W_{hy}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial W_{hy}}$$

  - **Gradients for Hidden Weights**:

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_{hh}}$$

  - **Gradients for Input Weights**:

$$\frac{\partial L}{\partial W_{xh}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_{xh}}$$

Here,

- W_{xh}: Input to hidden state
- W_{hh}: Hidden state to hidden state
- W_{hy}: Hidden state to output

Backpropagation Through Time (BPTT)

$$\frac{\partial L}{\partial W_{xh}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_{xh}}$$

This equation can be broken down into several components:

1. **Gradient of Loss with Respect to Hidden State ($\frac{\partial L_t}{\partial h_t}$):**
   - This part represents how the loss at a specific time step t changes with respect to the hidden state h_t.
   - It measures the sensitivity of the loss to changes in the hidden state at each time step.
2. **Gradient of Hidden State with Respect to Input-to-Hidden Weight ($\frac{\partial h_t}{\partial W_{xh}}$):**
   - This part represents how the hidden state h_t at time step t changes with respect to the input-to-hidden weight matrix W_{xh}.
   - It captures how a small change in the input-to-hidden weights affects the hidden state.

Now: Step 05 & Step 06

Backpropagation Through Time (BPTT)

5. Update Parameters

- **Apply Gradients**: Update the weights and biases using an optimization algorithm like Stochastic Gradient Descent (SGD) or Adam:

$$W \leftarrow W - \eta \frac{\partial L}{\partial W}$$

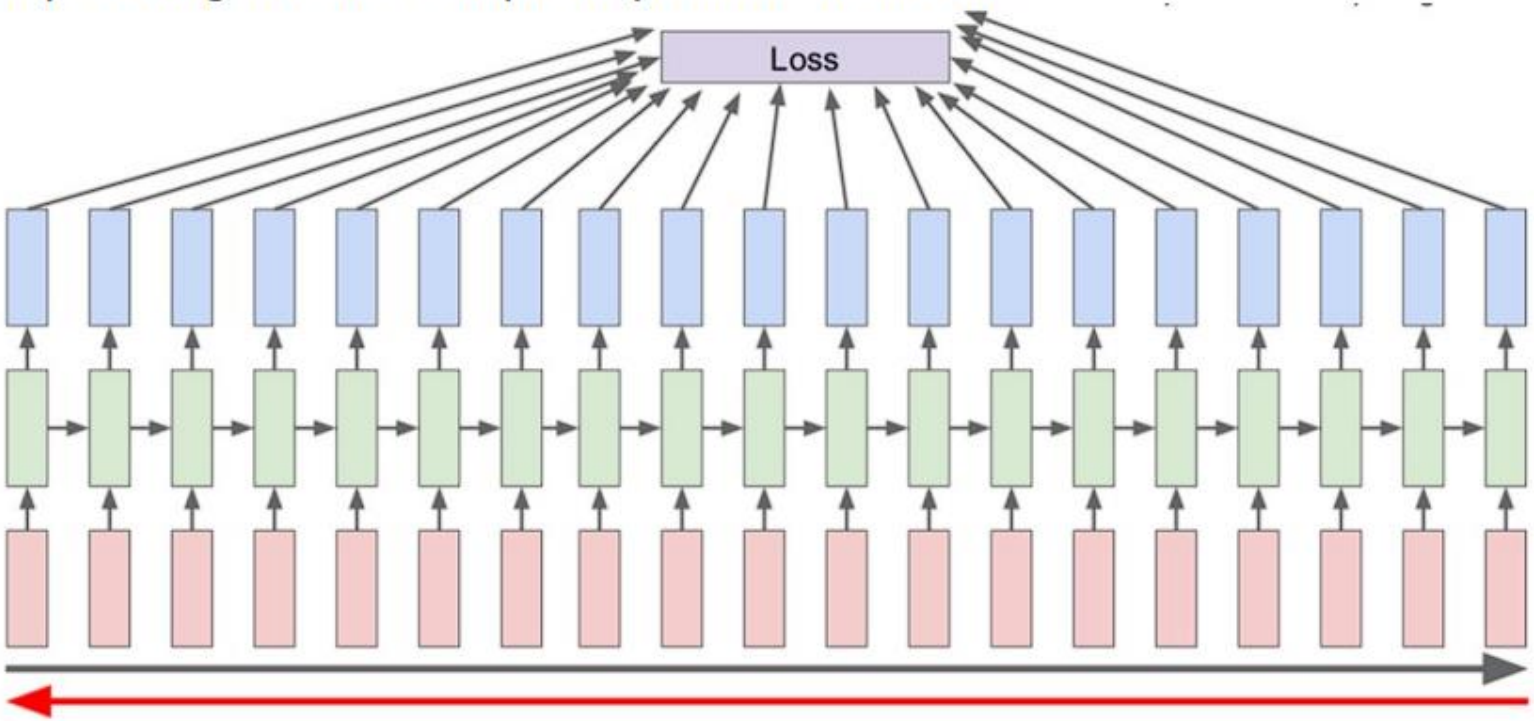
where η is the learning rate.

6. Repeat

- **Iterate**: Repeat the forward pass, loss calculation, backward pass, and parameter update steps for multiple epochs until the model converges.

Limitation of BPTT

BPTT can be computationally very expensive as a lot of partial derivatives have to be computed, depending on the complexity of the network.

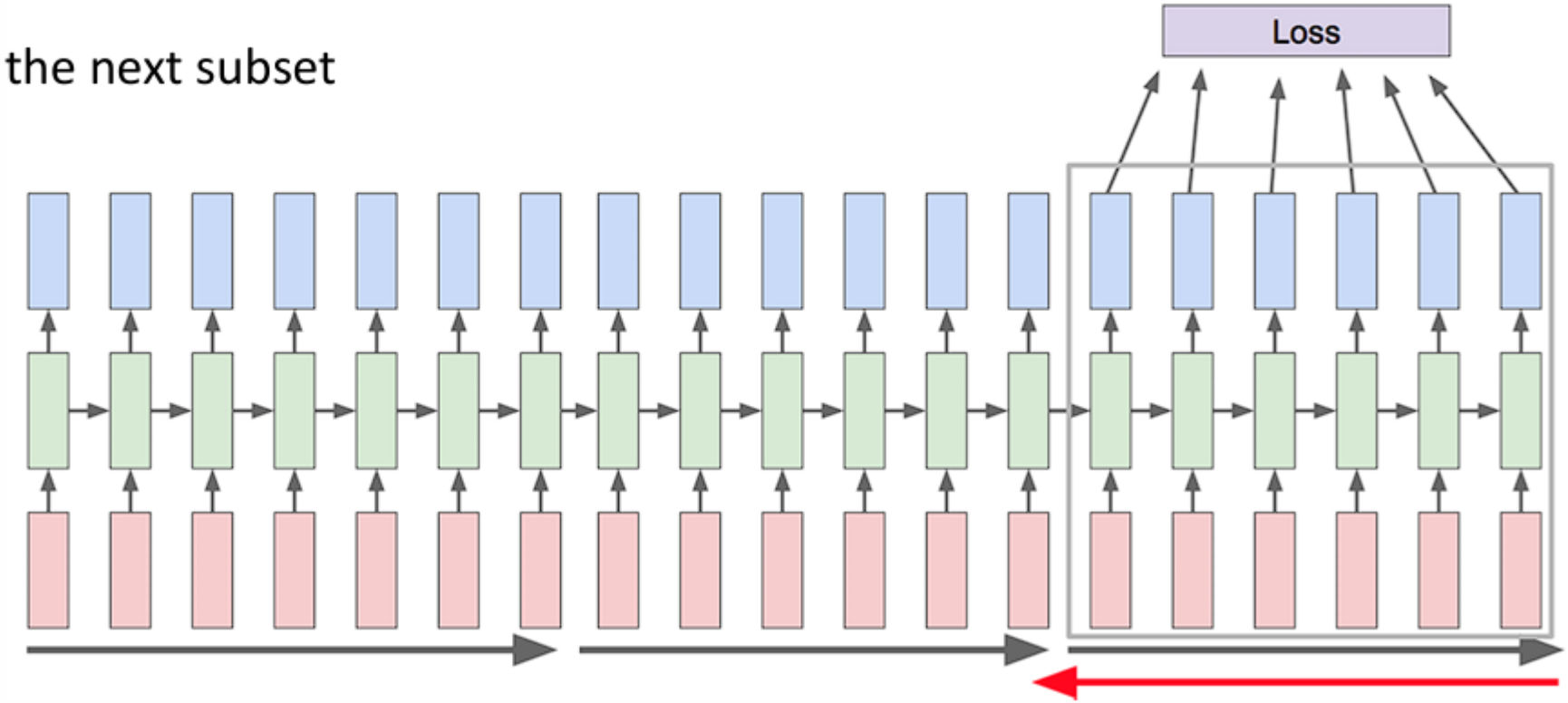


What is the Solution?

Truncated Backpropagation Through Time (Trunc-BPTT)

Truncated Backpropagation Through Time (Trunc-BPTT)

- Instead of passing through the complete sequence, forward pass through a subset
- Backpropagate through the subset
- Continue with the next subset



Major Problems in Recurrent Neural Networks (RNNs)

Main Problems in RNNs

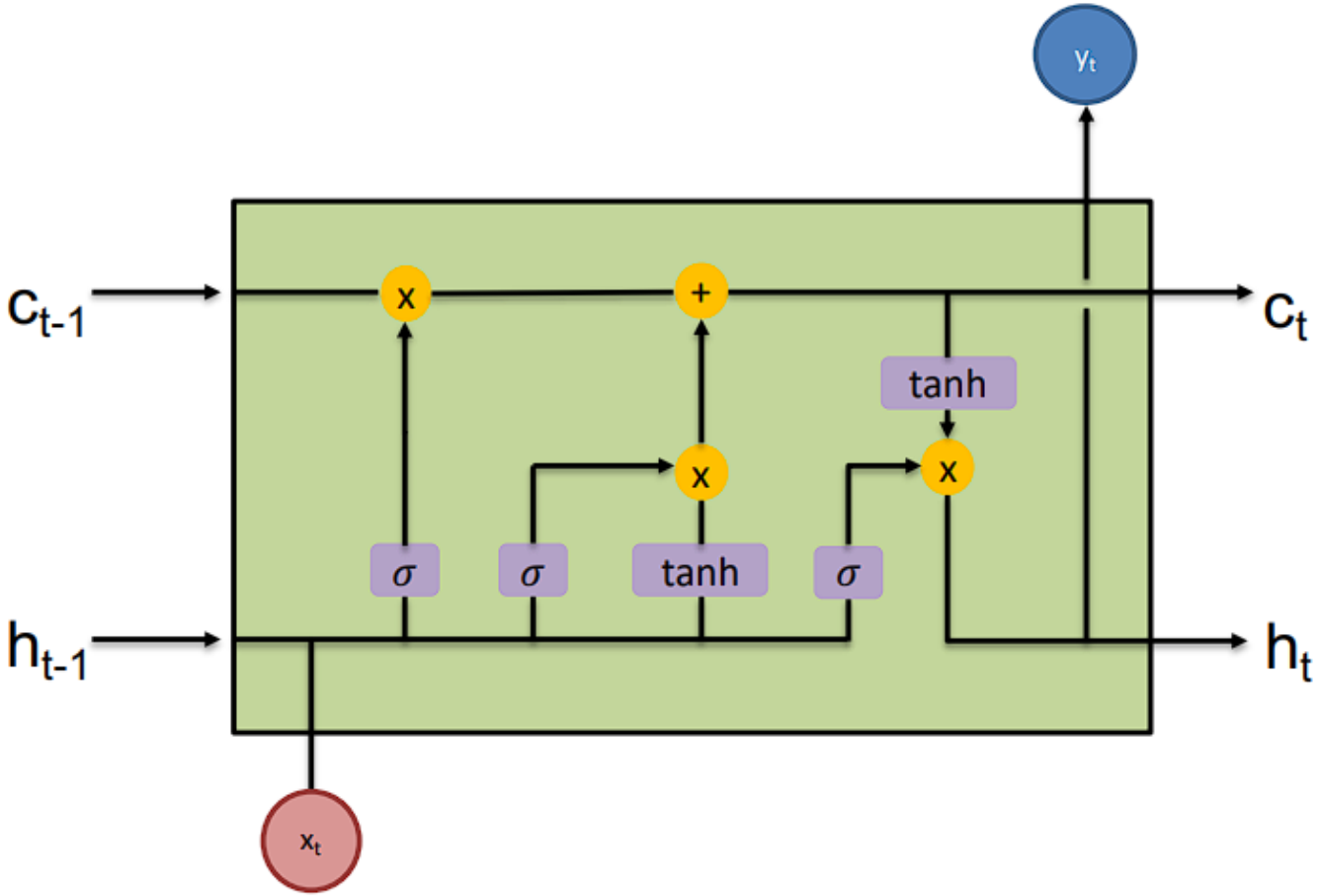
- **Vanishing Gradient Problem:** Gradients become very small during backpropagation, causing slow learning and difficulty in capturing long-term dependencies.
- **Exploding Gradient Problem:** Gradients become very large, leading to unstable training and potential divergence of the model.
- **Short-term Memory:** Difficulty in maintaining information over long sequences, causing poor performance on tasks requiring long-term context retention.
- **Handling Long Sequences:** Performance degrades with increasing sequence length, making RNNs less suitable for tasks involving long sequences.
- **Gradient Clipping Requirement:** To manage exploding gradients, gradient clipping is needed, adding complexity to the training process.

Differences: RNNs Vs. LSTMs

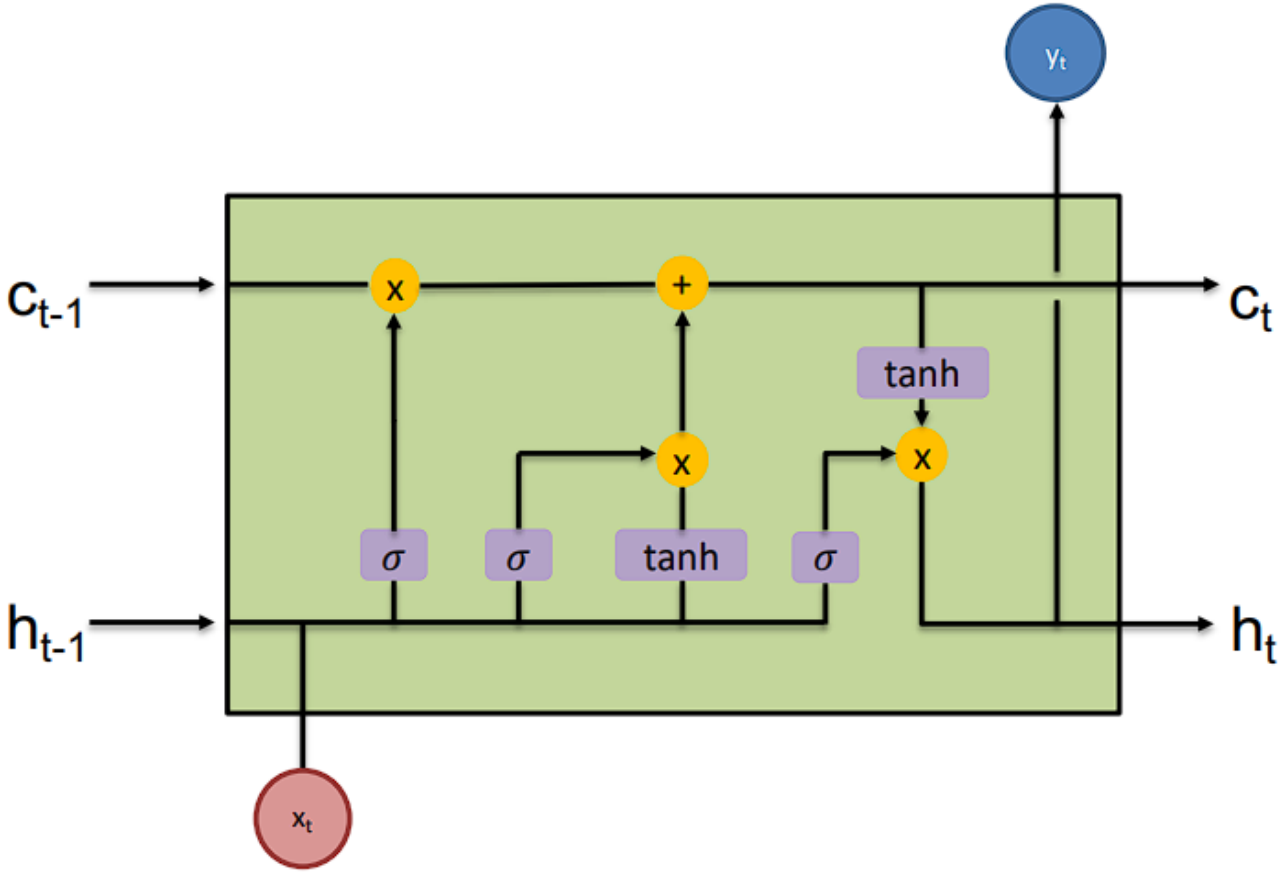
Feature	RNNs	LSTMs
Vanishing Gradient Problem	Common	Mitigated
Exploding Gradient Problem	Common	Mitigated
Memory	Short-term	Long-term
Sequence Length Handling	Poor	Better
Internal Mechanisms	Simple recurrent connections	Cell state, input, forget, output gates
Training Stability	Less stable	More stable
Task Performance	Weaker for long sequences	Stronger for long sequences

Long Short-Term Memory (LSTMs)

Long Short-Term Memory (LSTM)



LSTM: Architecture

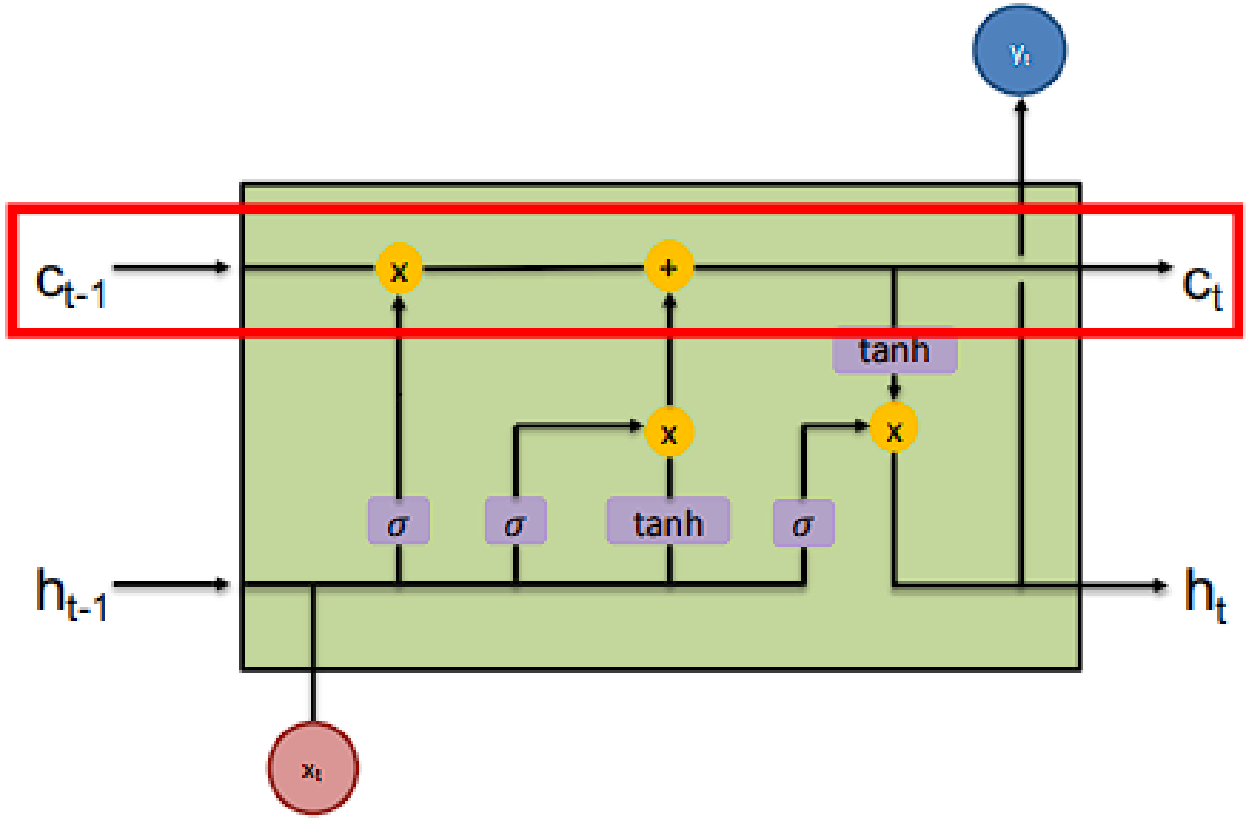


- Compared to RNNs, additionally to the **hidden state** h_t, we have another state, c_t, said the **cell state**.
- The information flow is regulated by **three gates**: the forget gate, the input gate and the output gate.

LSTM: Main Components

- The **Forget Gate** f_t decides which information to discard from the cell state.
- The **Input Gate** i_t decides which new information to add to the cell state.
- The **Candidate Cell State** $\tilde{C}_t$ represents new information that could be added to the cell state.
- The **Cell State Update** C_t combines the retained and new information to update the cell state.
- The **Output Gate** o_t decides the information that will be output from the LSTM.
- The **Hidden State Update** h_t combines the output gate decision with the updated cell state to produce the new hidden state.

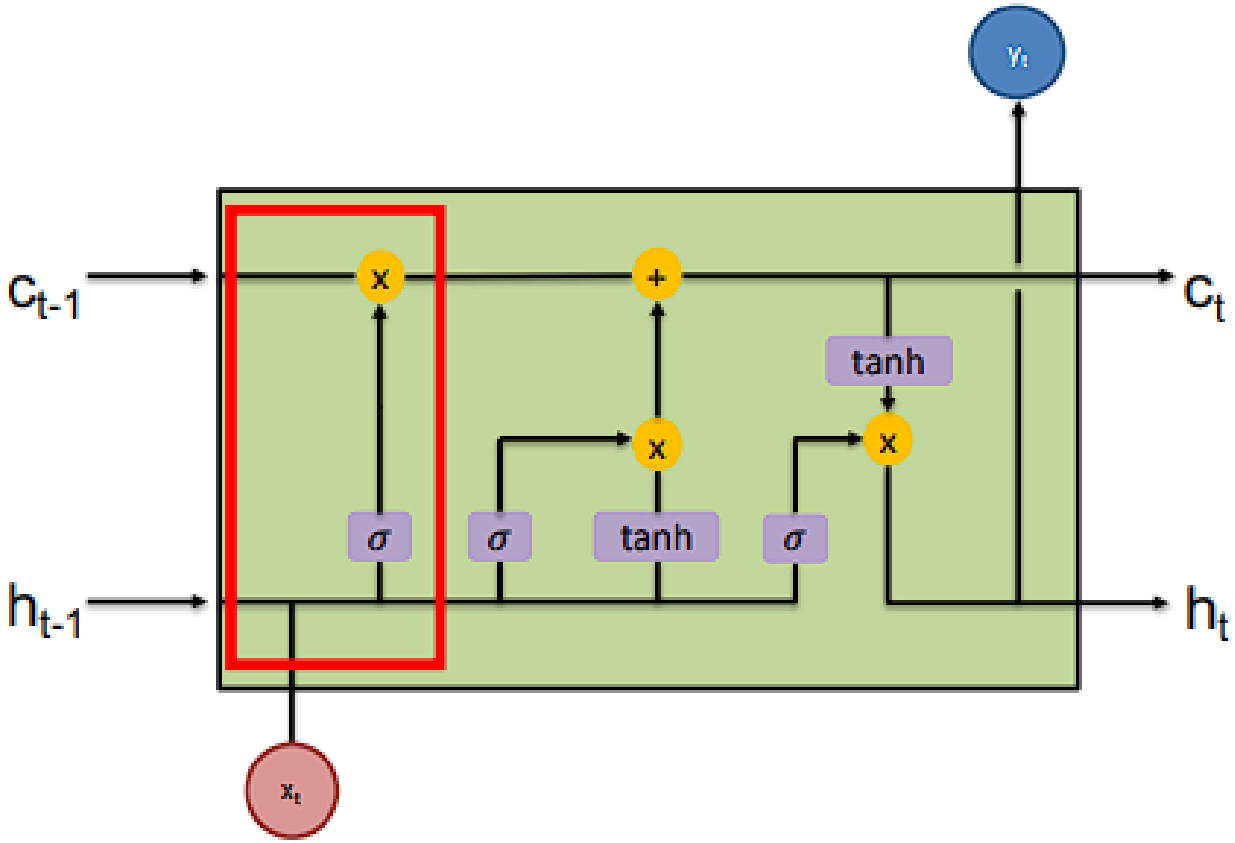
LSTM: Cell State



The cell state has:

- Only minor interactions
- Simple information flow
- Other gates regulates weather it is preserved/not-preserved or updated.

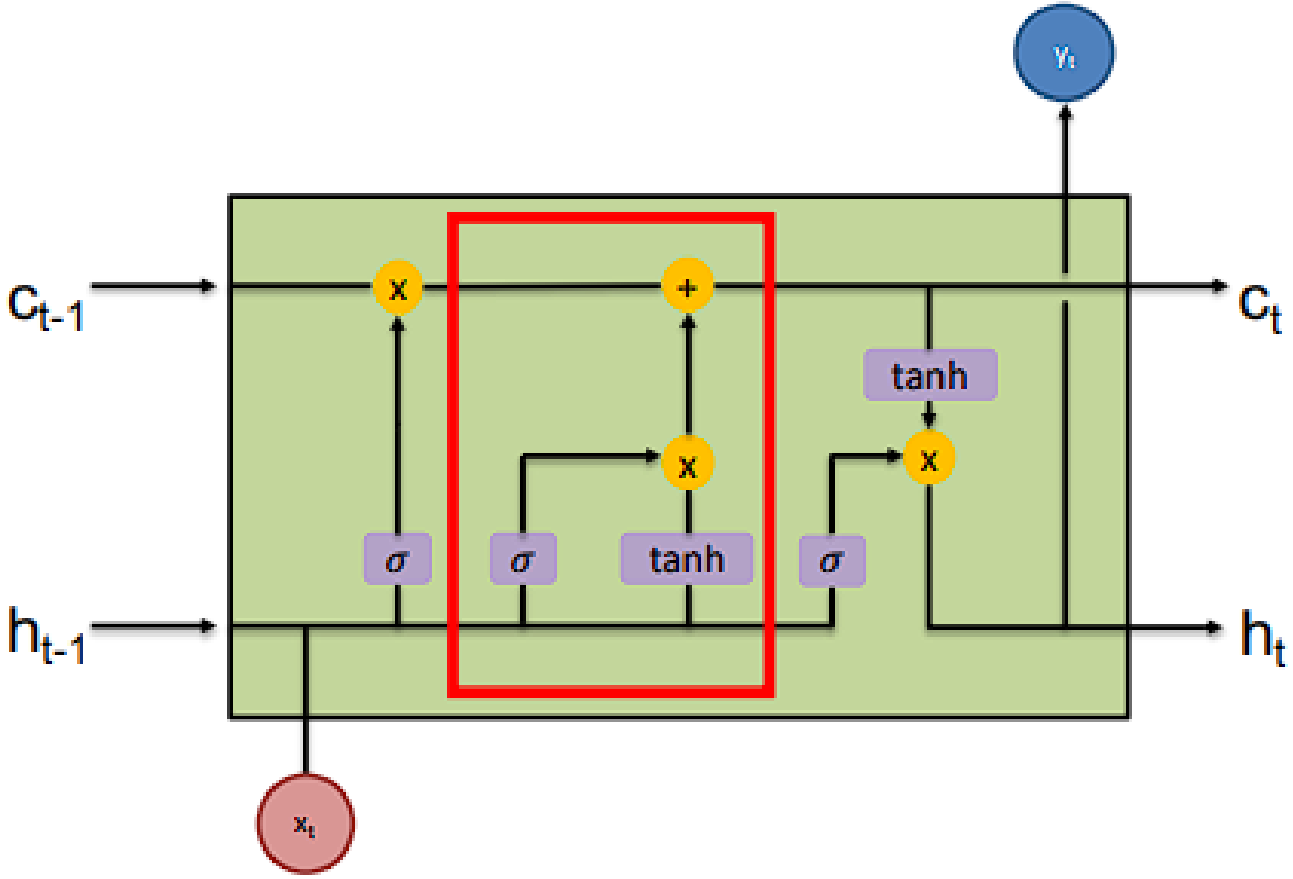
LSTM: Forget Gate



The **forget gate** decides how much information to retain from the previous cell state.

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

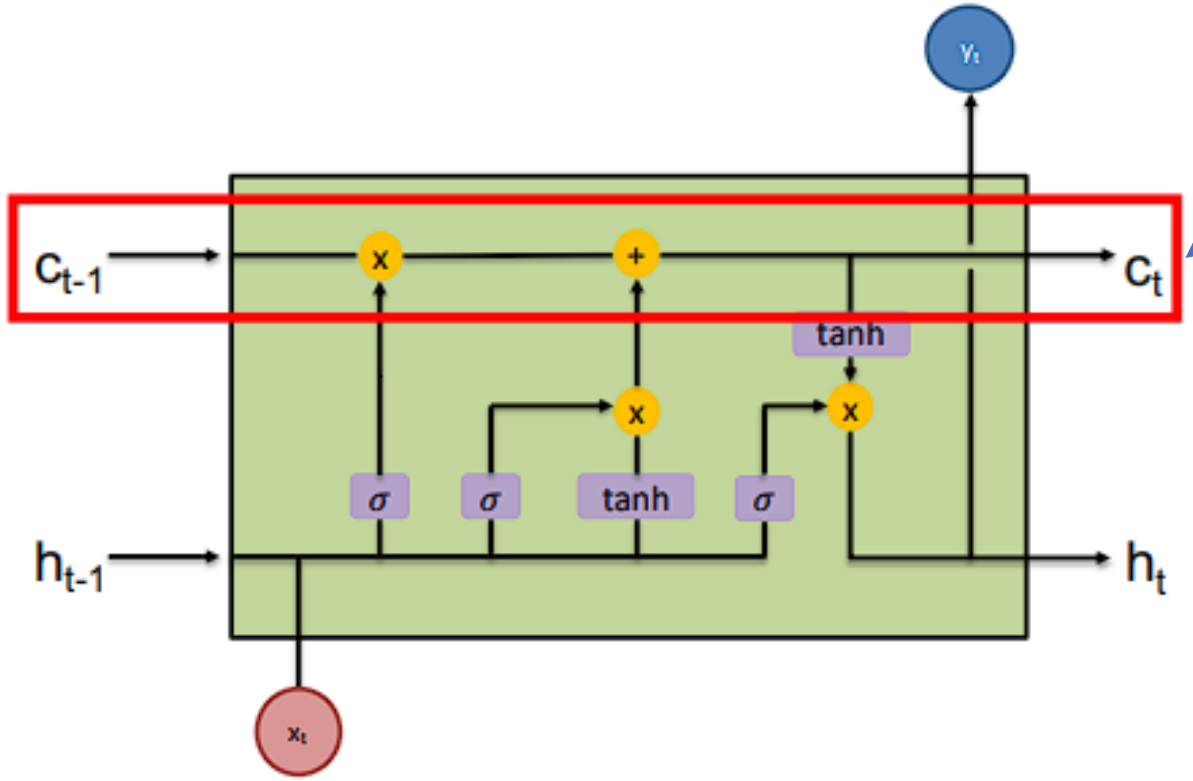
LSTM: Input Gate



The **input gate** decides the information to be added to the cell state, based on the current input.

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

LSTM: Cell State Update

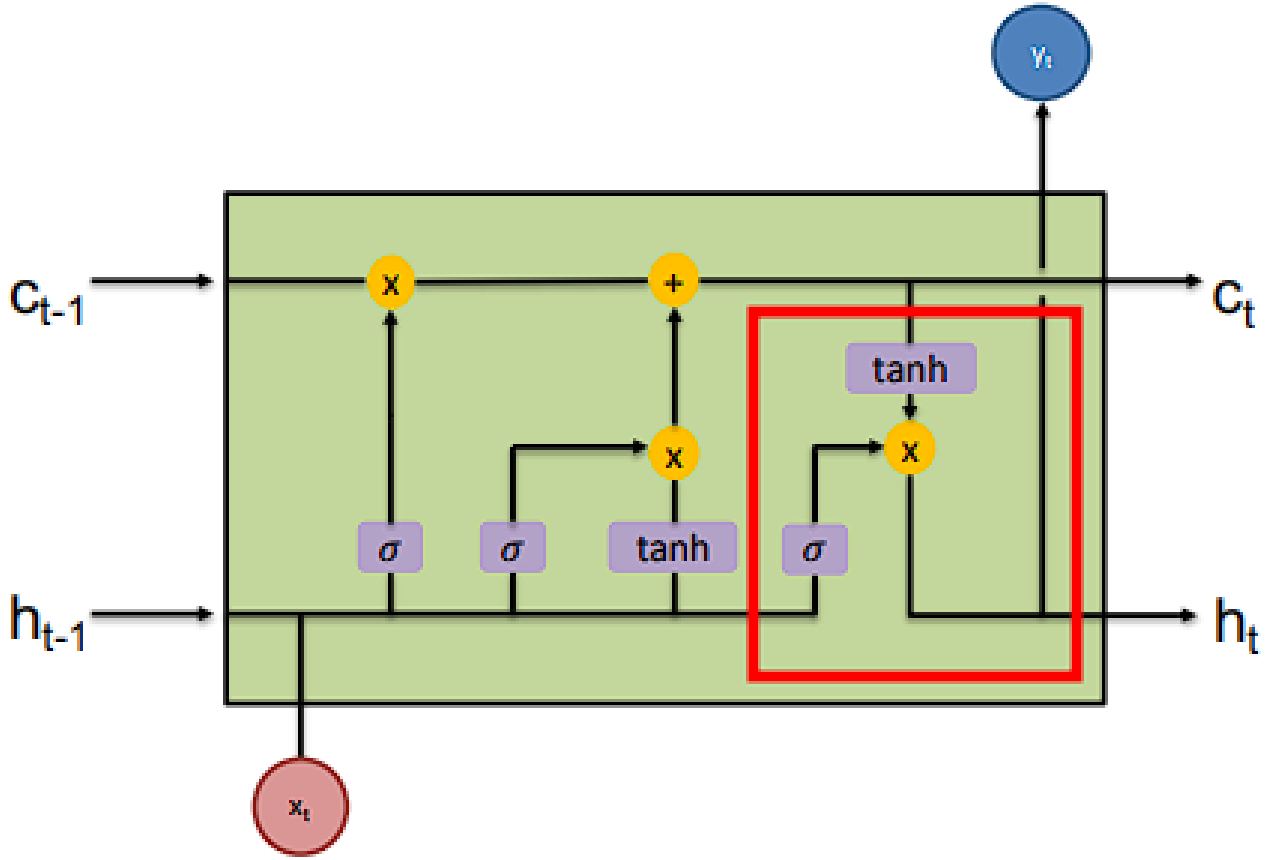


Update: Cell State

The values can be combined to update the cell state

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

LSTM: Output Gate

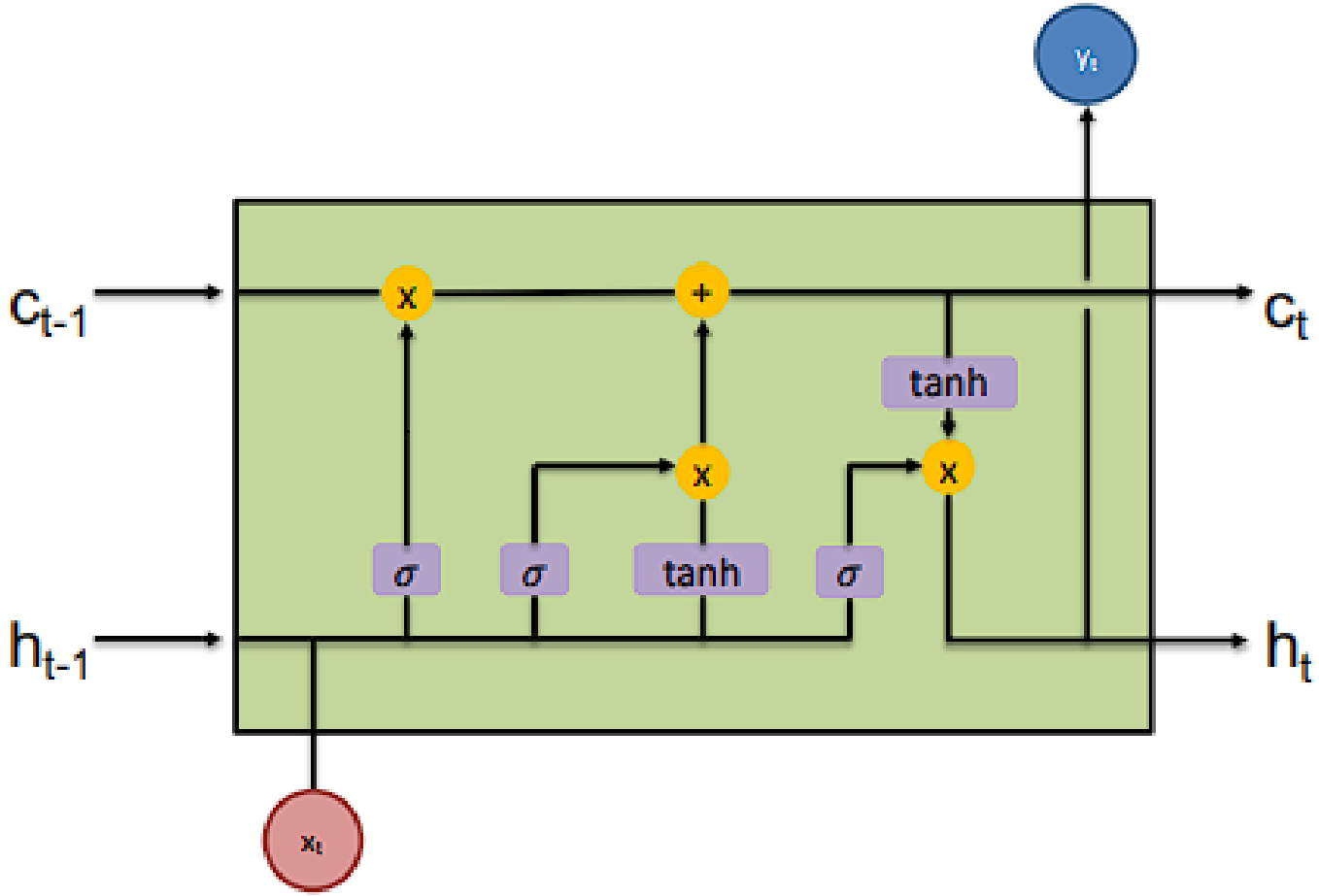


The **output gate** generates the output of the current LSTM cell, based on the current input, the previous output, and the updated cell state.

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

Let’s Summarize, the LSTM Cell is Described by the Following Equations!

Long Short-Term Memory (LSTM)

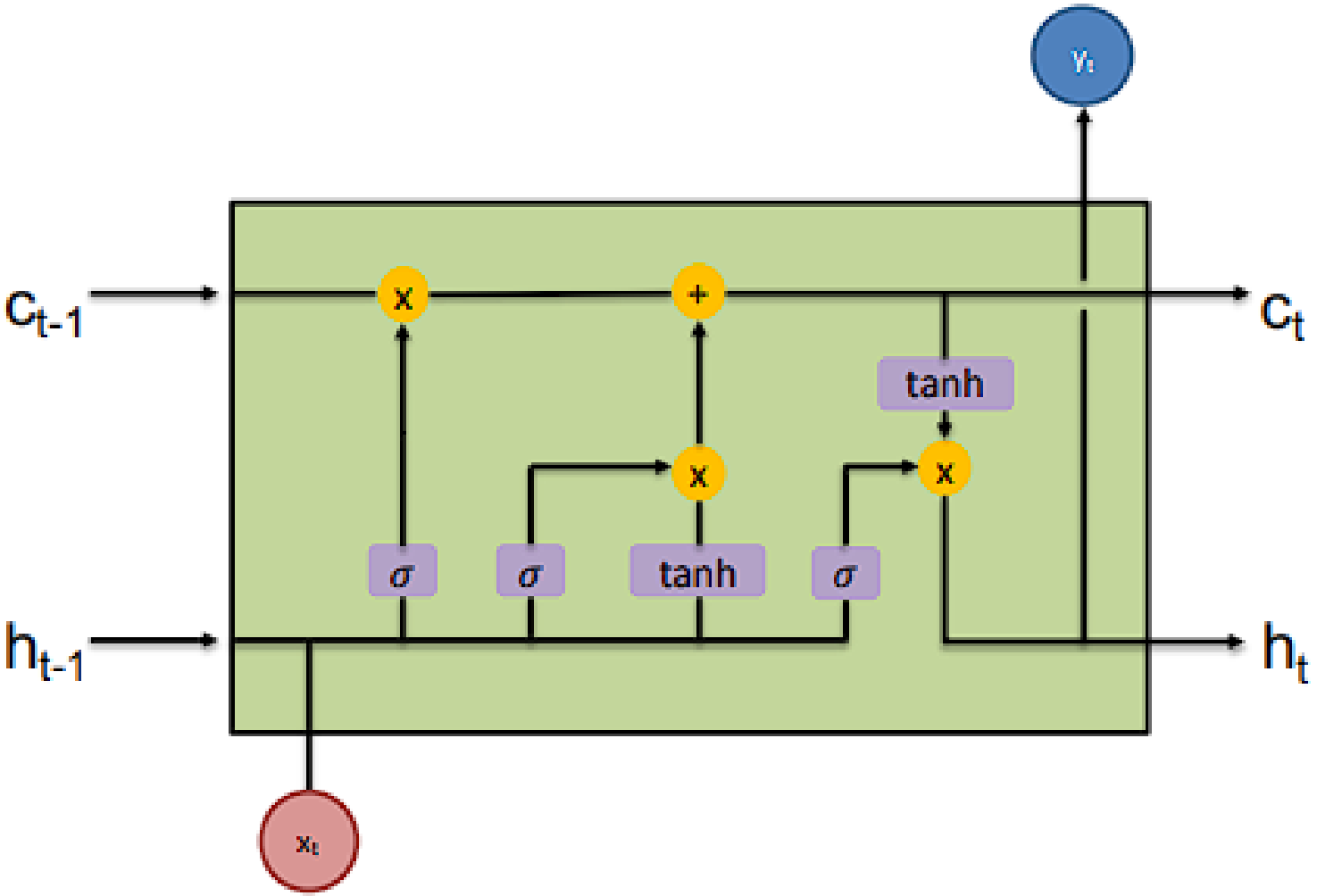


Long Short-Term Memory (LSTM)

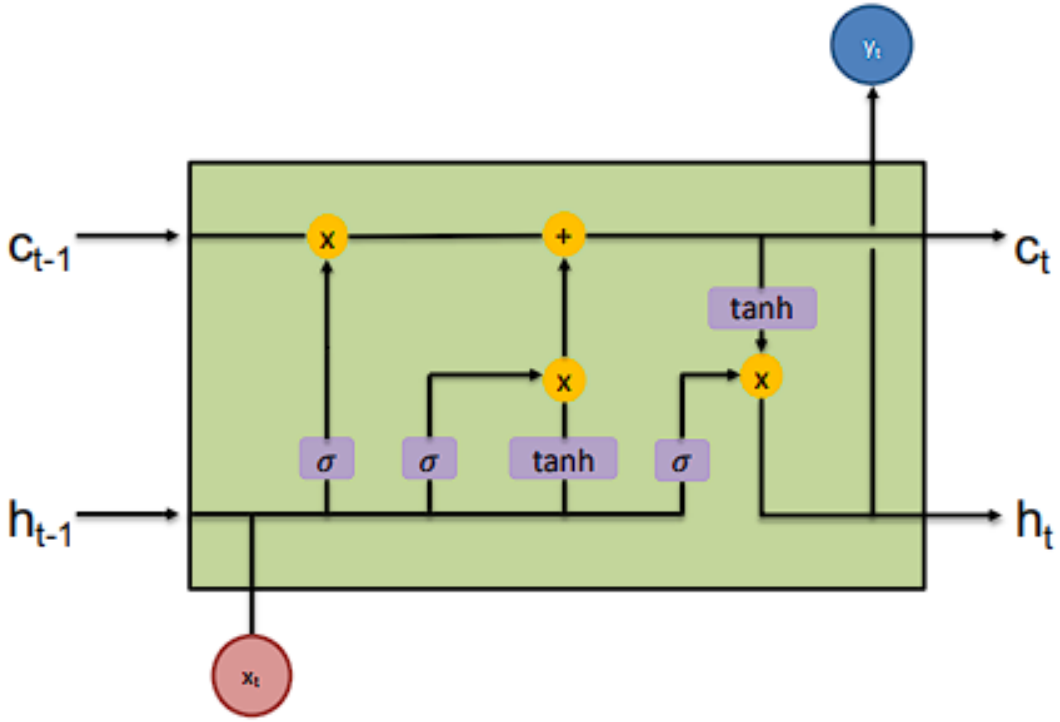
Component	Equation
Forget Gate (f_t)	$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$
Input Gate (i_t)	$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$
Candidate Cell State ($\tilde{C}_t$)	$\tilde{C}_t = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$
Update Cell State (C_t)	$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$
Output Gate (o_t)	$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$
Update Hidden State (h_t)	$h_t = o_t \cdot \tanh(C_t)$
Updated Output (y_t)	$y_t = \text{softmax}(W_y h_t + b_y)$

Implementing LSTM Cells in Python

Long Short-Term Memory (LSTM)



Long Short-Term Memory (LSTM)



All Equations Together:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$g_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(c_t)$$

Long Short-Term Memory (LSTM)

Component	Equation
Forget Gate (f_t)	$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$
Input Gate (i_t)	$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$
Candidate Cell State ($\tilde{C}_t$)	$\tilde{C}_t = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$
Update Cell State (C_t)	$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$
Output Gate (o_t)	$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$
Update Hidden State (h_t)	$h_t = o_t \cdot \tanh(C_t)$
Updated Output (y_t)	$y_t = \text{softmax}(W_y h_t + b_y)$

Long Short-Term Memory (LSTM)

Step	Implementation
Concatenate	`concat = np.concatenate([h_prev, xt], axis=0)`
Forget Gate (f_t)	`ft = sigmoid(Wf @ concat + bf)`
Input Gate (i_t)	`it = sigmoid(np.dot(Wi, concat) + bi)`
Candidate Cell State ($\tilde{C}_t$)	`gt = np.tanh(np.dot(Wc, concat) + bc)`
Update Cell State (C_t)	`c_next = ft * c_prev + it * gt`
Output Gate (o_t)	`ot = sigmoid(np.dot(Wo, concat) + bo)`
Update Hidden State (h_t)	`h_next = ot * np.tanh(c_next)`
Output (y_t)	`yt_pred = softmax(np.dot(Wy, h_next) + by)`

Long Short-Term Memory (LSTM)

- `np.dot`: General dot product function. For 1-D arrays, it calculates the inner product. For 2-D arrays, it performs matrix multiplication.
- `*`: Element-wise multiplication. Multiplies corresponding elements of two arrays of the same shape.
- `@`: Matrix multiplication operator. Used for matrix multiplication of 2-D arrays and works with higher-dimensional arrays.

Gated Recurrent Units (GRUs)

Gated Recurrent Units (GRUs)

Main components of Gated Recurrent Unit (GRU):

1. Reset Gate (r_t)
2. Update Gate (z_t)
3. Candidate Hidden State ($\tilde{h}_t$)

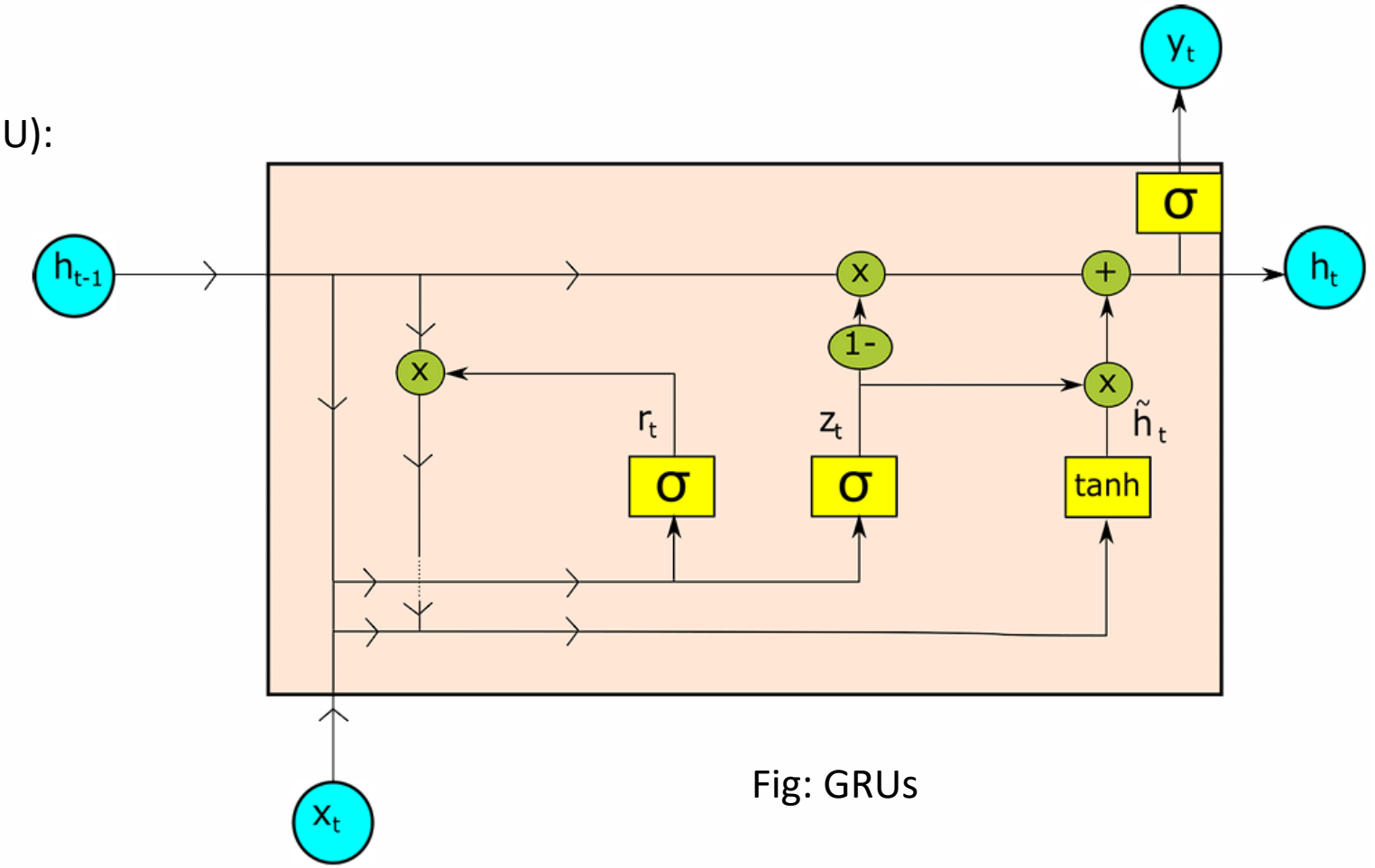


Fig: GRUs

Long Short-Term Memory (LSTM)

Main components of Long Short-Term Memory (LSTM):

1. Forget Gate (f_t)
2. Input Gate (i_t)
3. Output Gate (o_t)
4. Cell State (C_t)

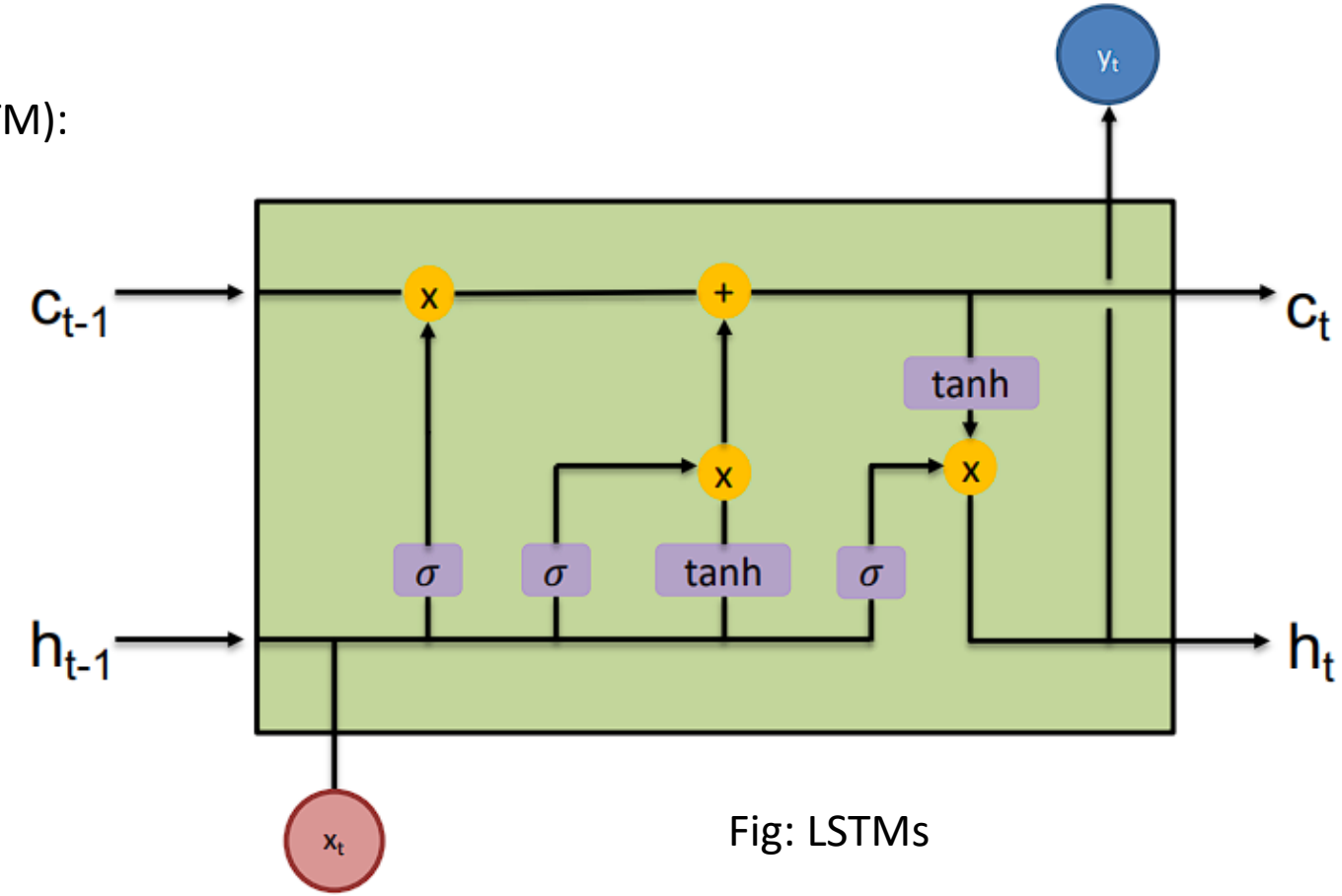


Fig: LSTMs

More Gates:

- LSTMs have **three gates**: input gate, forget gate, and output gate.
- Each gate requires its own set of weights and biases.
- This results in a **larger number of parameters** (weights and biases) that need to be stored and updated during training.

Cell State:

- LSTMs maintain a **separate cell state** in addition to the hidden state.
- The cell state requires **additional memory** to store its value for **each time step** during the forward and backward passes.

Parameters: LSTMs Vs. GRUs

- **LSTM Parameter Count:** For an LSTM cell, the parameters include weights and biases for the input, forget, and output gates, as well as the candidate cell state.
  - Total Parameters=4 × ((input_size + hidden_size) × hidden_size + hidden_size)

- **GRU Parameter Count:** For a GRU cell, the parameters include weights and biases for the reset gate, update gate, and candidate hidden state.
  - Total Parameters=3 × ((input_size + hidden_size) × hidden_size + hidden_size)

Parameters: LSTMs Vs. GRUs

Suppose we have:

- Input size $i = 100$
- Hidden state size $h = 50$

For an LSTM:

- Number of parameters: $4 \times ((100 + 50) \times 50 + 50) = 4 \times (150 \times 50 + 50) = 4 \times (7500 + 50) = 4 \times 7550 = \boxed{30200}$

For a GRU:

- Number of parameters: $3 \times ((100 + 50) \times 50 + 50) = 3 \times (150 \times 50 + 50) = 3 \times (7500 + 50) = 3 \times 7550 = \boxed{22650}$

Step by Step: Gated Recurrent Units (GRUs)

Gated Recurrent Units (GRUs)

Main components of Gated Recurrent Unit (GRU):

1. Reset Gate (r_t)
2. Update Gate (z_t)
3. Candidate Hidden State ($\tilde{h}_t$)

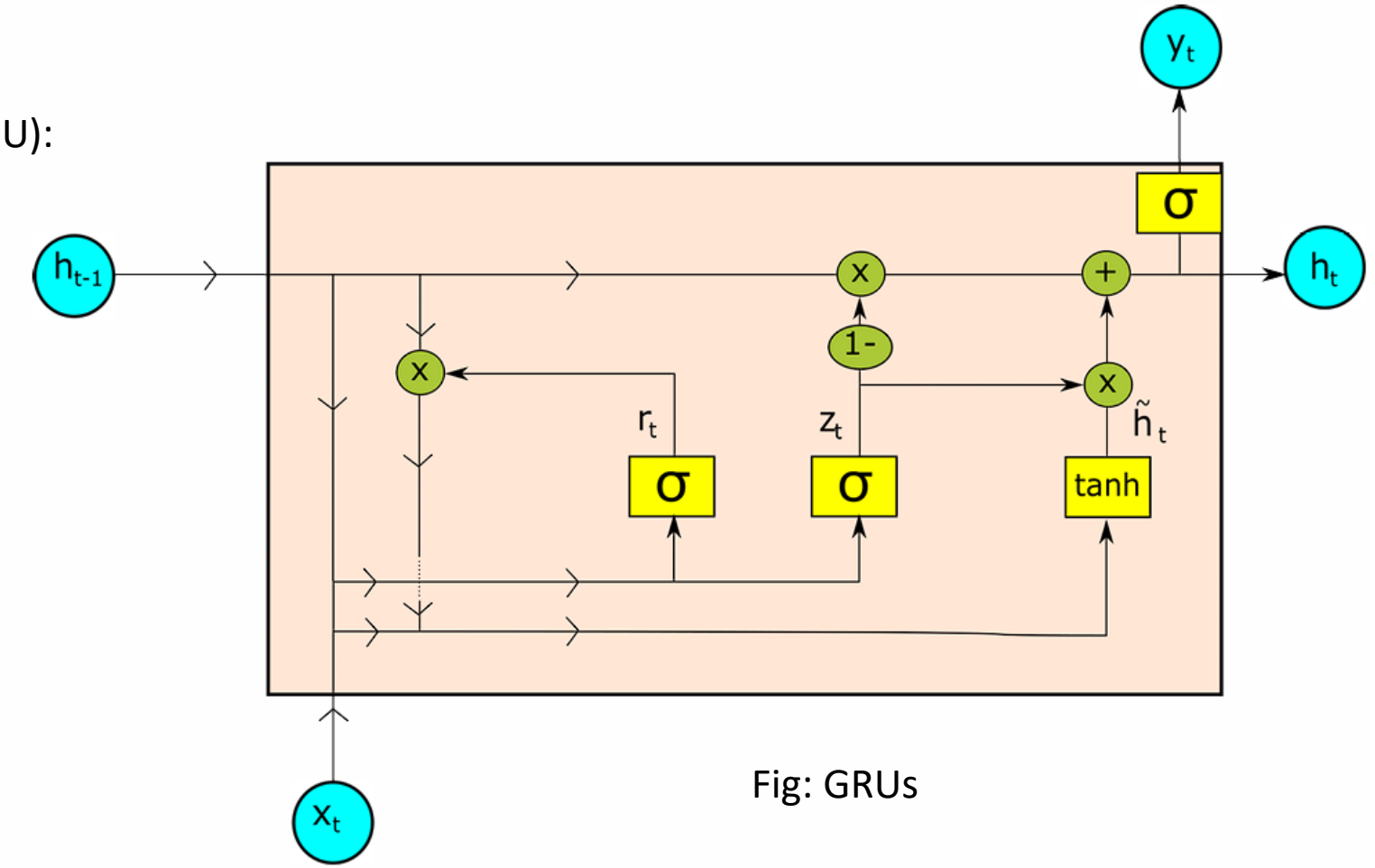


Fig: GRUs

1. **Reset Gate** controls how much past information to forget.
2. **Update Gate** controls how much past information to retain.
3. **Candidate Hidden State** is computed using the reset gate's influence on the previous hidden state.
4. **Final Hidden State** is a weighted combination of the **previous hidden state** and the **candidate hidden state**, controlled by the **update gate**.

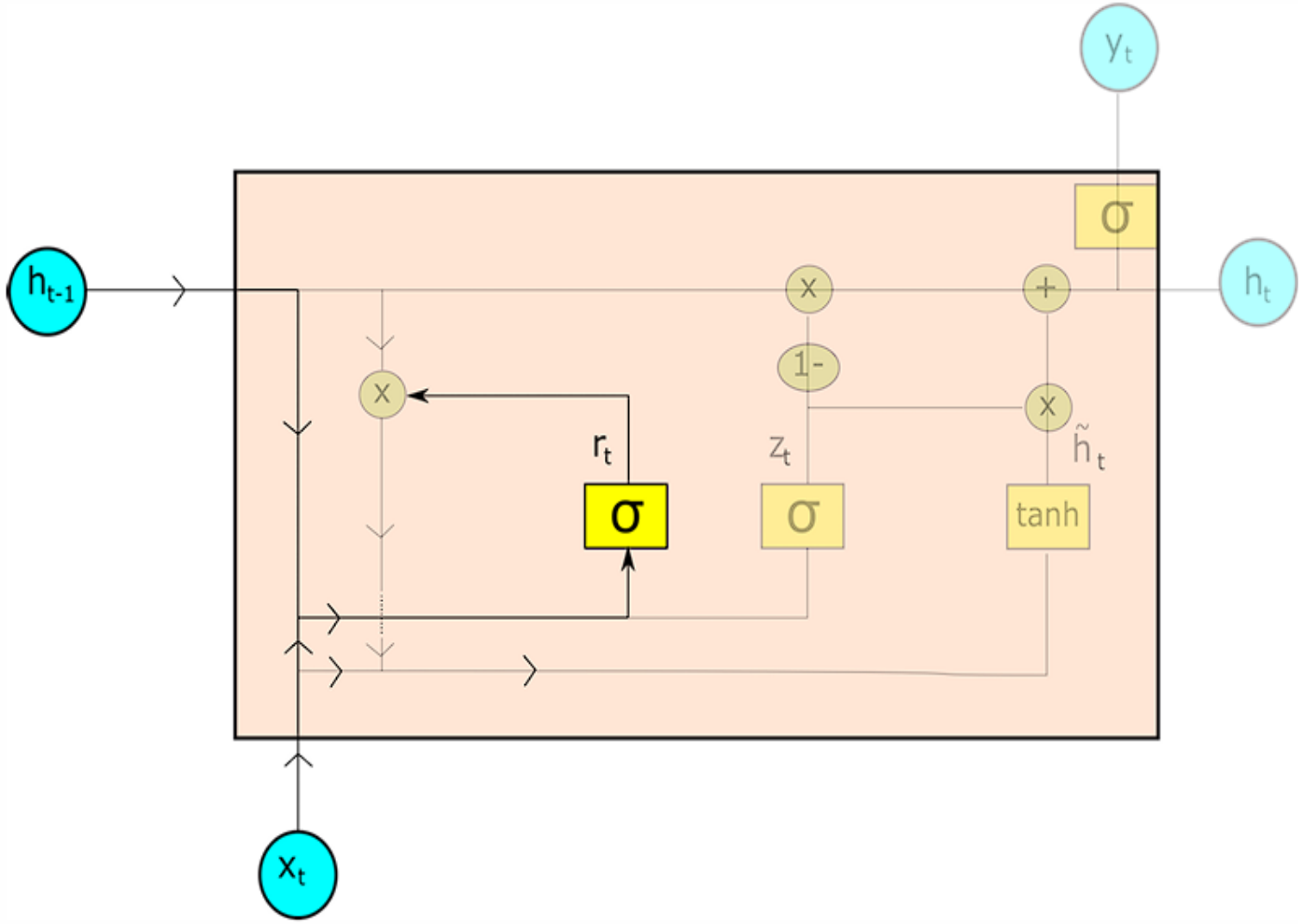
Gated Recurrent Units (GRUs)

Reset Gate

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

Here,

- W_r: Weight matrix for the reset gate.
- h_{t-1}: Previous hidden state.
- x_t: Current input.
- b_r: Bias term for the reset gate.



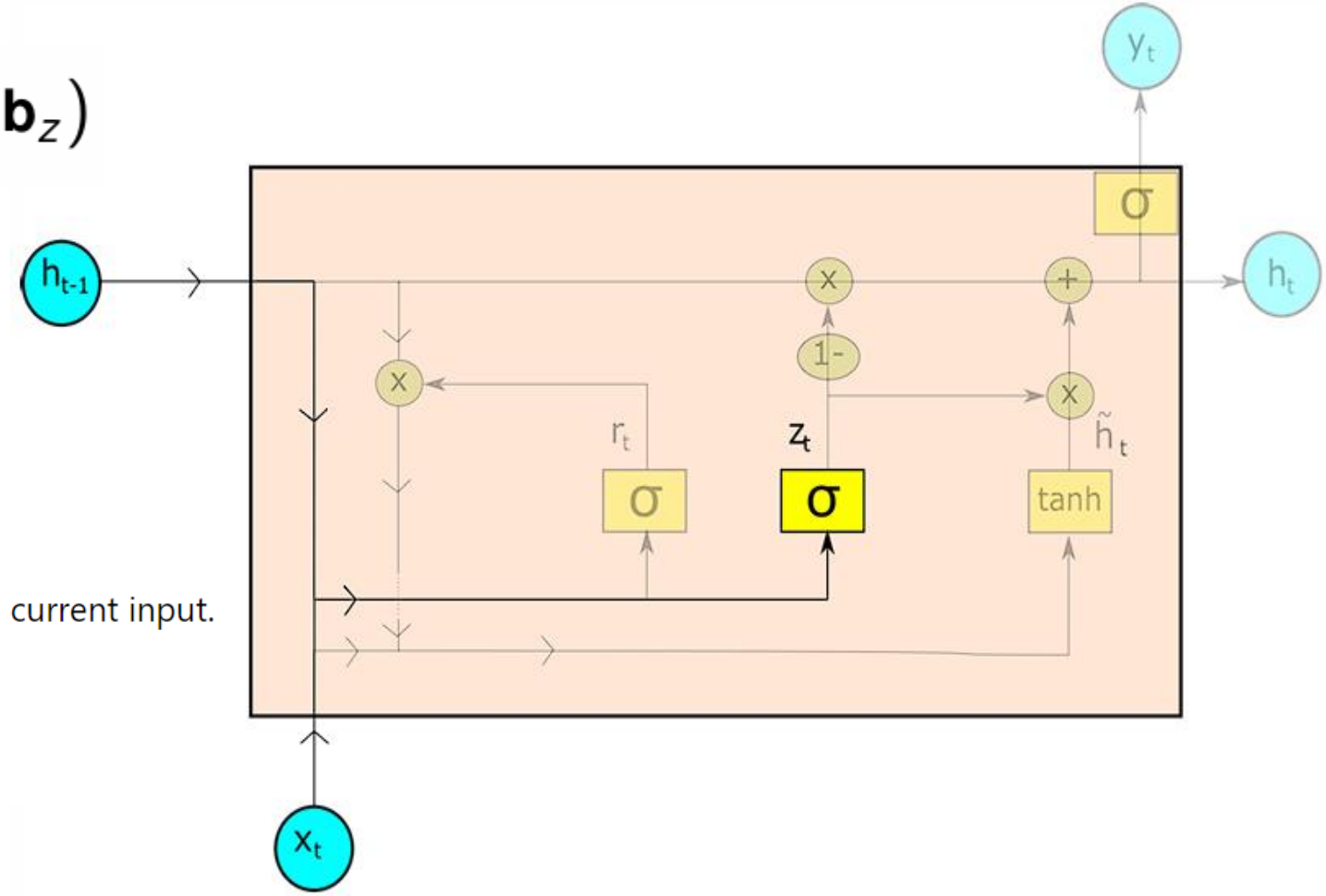
Gated Recurrent Units (GRUs)

Update Gate

Formula: $\mathbf{z}_t = \sigma\left(\mathbf{W}_z \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_z\right)$

Here,

- W_z: Weight matrix for the update gate.
- h_{t-1}: Previous hidden state.
- x_t: Current input.
- h_{t-1}, x_t: Concatenation of previous hidden state and current input.
- b_z: Bias term for the update gate.
- σ: Sigmoid activation function.



Gated Recurrent Units (GRUs)

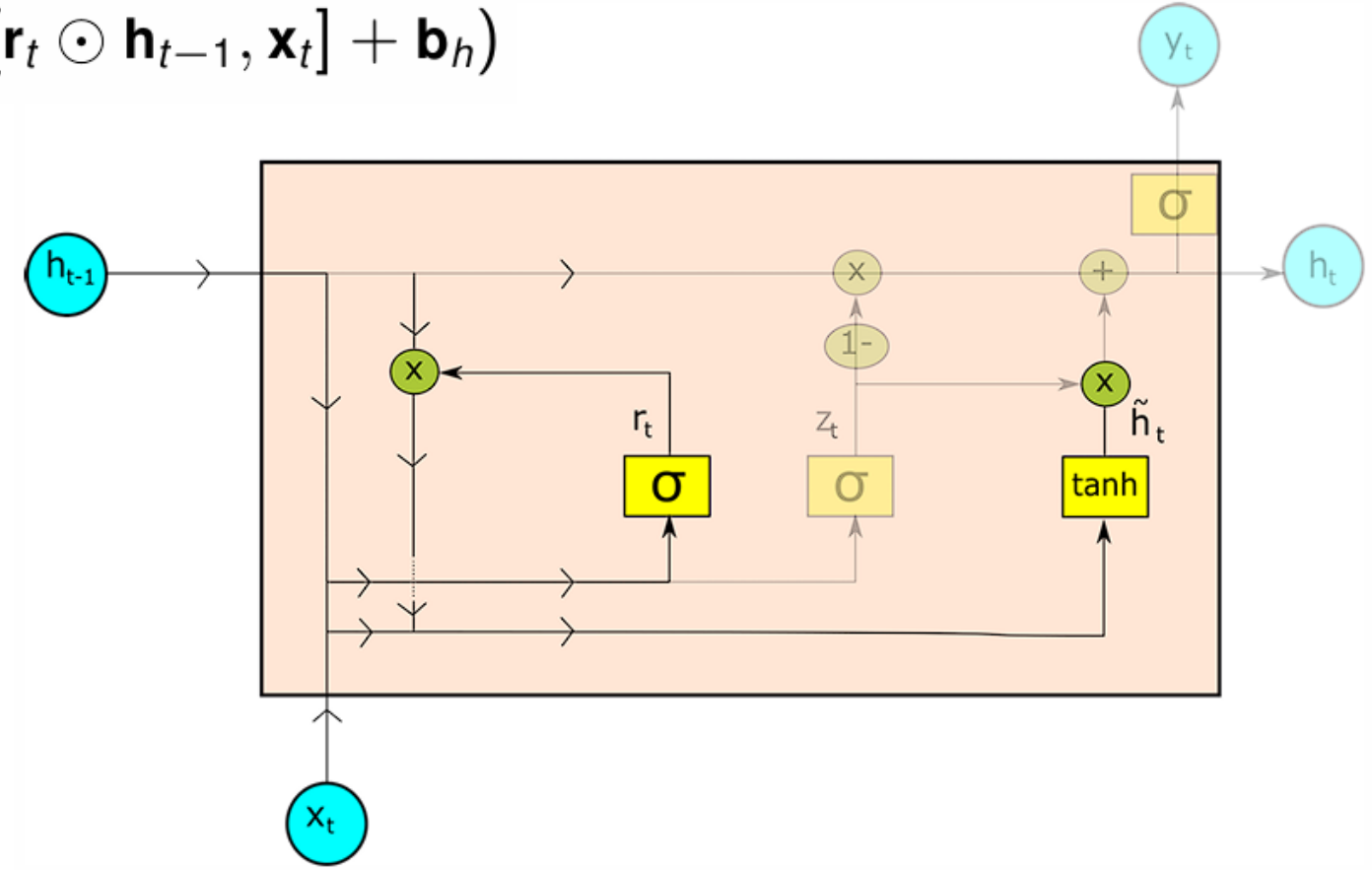
Candidate Hidden State

- Combination of input and "reset" hidden state.
- If $\mathbf{r}_t$ is close to 0 ➔ low influence of previous hidden state

$$\tilde{\mathbf{h}}_t = tanh\left(\mathbf{W}_h \cdot [\mathbf{r}_t \odot \mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_h\right)$$

The Reset Gate Controls the influence of the previous hidden state

- **Low** r_t: Less influence of h_{t-1}.
- **High** r_t: More influence of h_{t-1}.



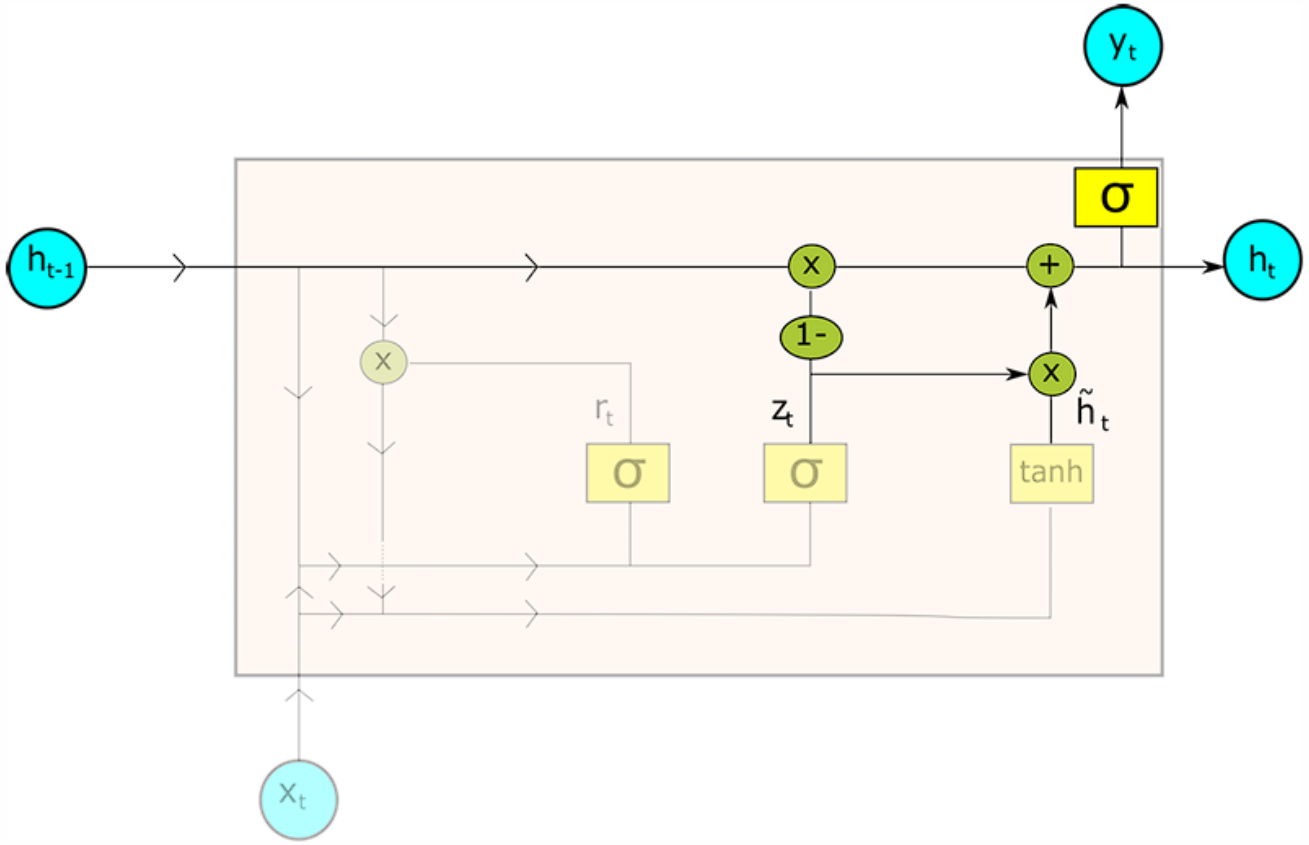
Gated Recurrent Units (GRUs)

Updated Hidden State

- **Update gate** controls combination of old state and proposed update

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

- Node output: $\hat{\mathbf{y}}_t = \sigma(\mathbf{h}_t)$



Gated Recurrent Units (GRUs)

Main components of Gated Recurrent Unit (GRU):

1. Reset Gate (r_t)
2. Update Gate (z_t)
3. Candidate Hidden State ($\tilde{h}_t$)

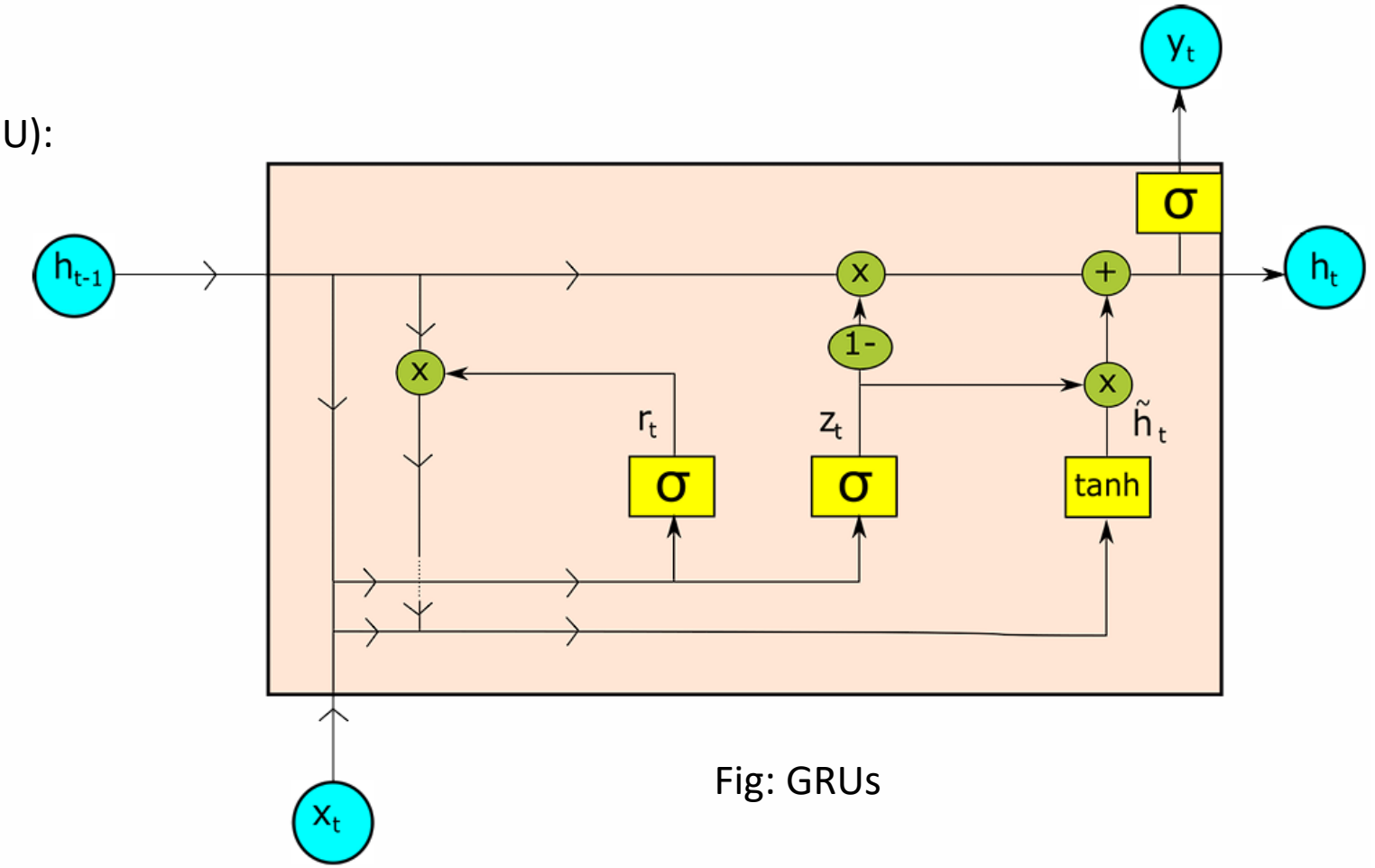


Fig: GRUs

Gated Recurrent Units (GRUs)

Equations for GRU

Component	Equation
Reset Gate (r_t)	$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$
Update Gate (z_t)	$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$
Candidate Hidden State ($\tilde{h}_t$)	$\tilde{h}_t = \tanh(W_h \cdot [r_t \circ h_{t-1}, x_t] + b_h)$
Final Hidden State (h_t)	$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t$
Updated Output (y_t)	$y_t = \text{softmax}(W_y \cdot h_t + b_y)$

LSTMs Vs. GRUs

Component	LSTM	GRU
Forget Gate	$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$	Not present
Input Gate	$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$	Combined with the update gate
Output Gate	$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$	Not present
Cell State	$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$	Not present
Reset Gate	Not present	$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$
Update Gate	Not present	$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$
Candidate Cell State	$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$	$\tilde{h}_t = \tanh(W_h \cdot [r_t \circ h_{t-1}, x_t] + b_h)$
Hidden State	$h_t = o_t \cdot \tanh(C_t)$	$h_t = z_t \cdot h_{t-1} + (1 - z_t) \cdot \tilde{h}_t$

Thank you!