Recurrent Neural Network (RNN)

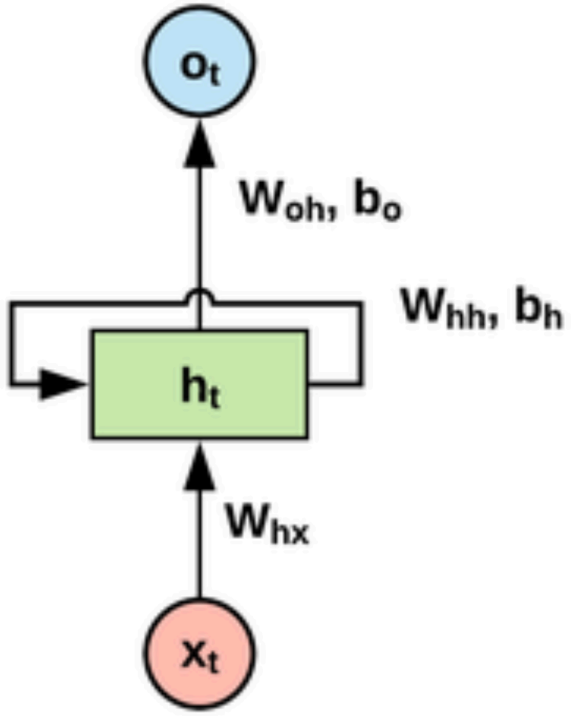
Motivation

- Humans don’t start their thinking from scratch every second.
- Vanilla neural network does not work with **sequential data**.

Opening or closing the door?

Introduction

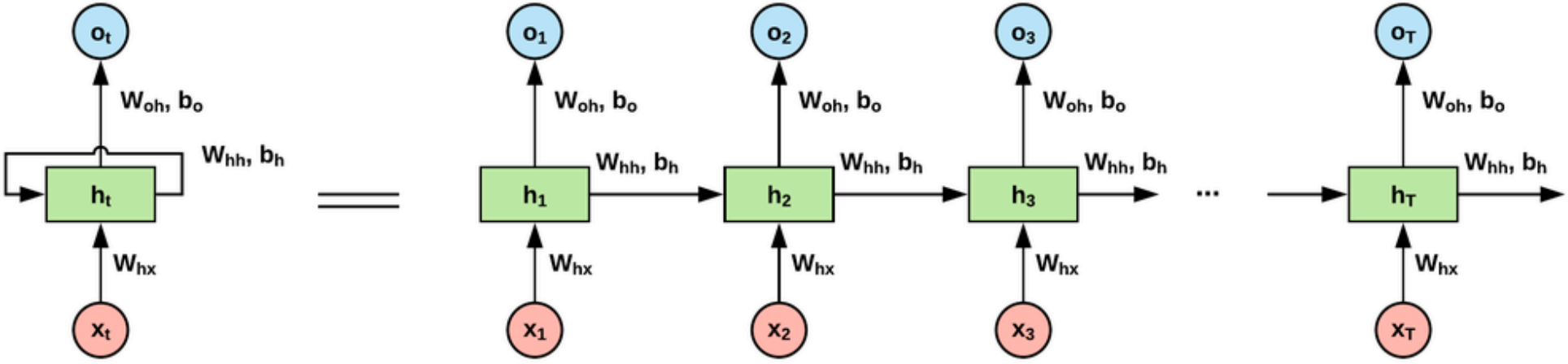
Recurrent neural network (RNN) can solve the issue: it has **memory** to **remember what it has seen** from sequential data.



RNN architecture

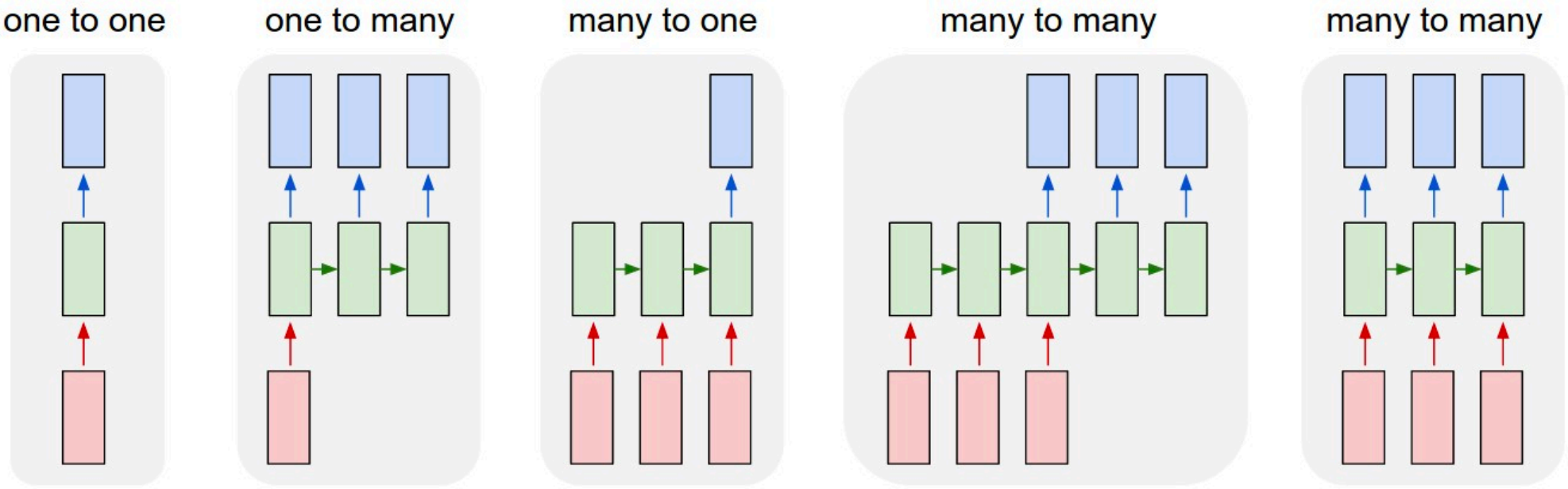
Introduction

RNN is able to do: next word prediction, music composition, image captioning, speech recognition, time series anomaly detection, stock market prediction, etc.



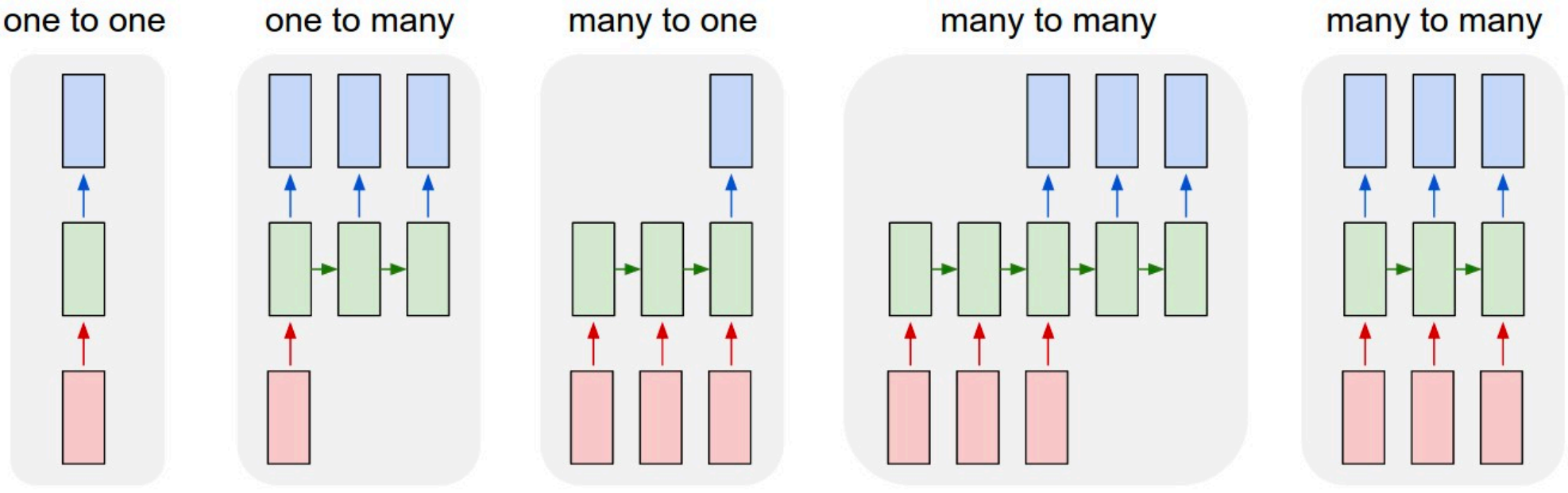
Unfold RNN

Introduction



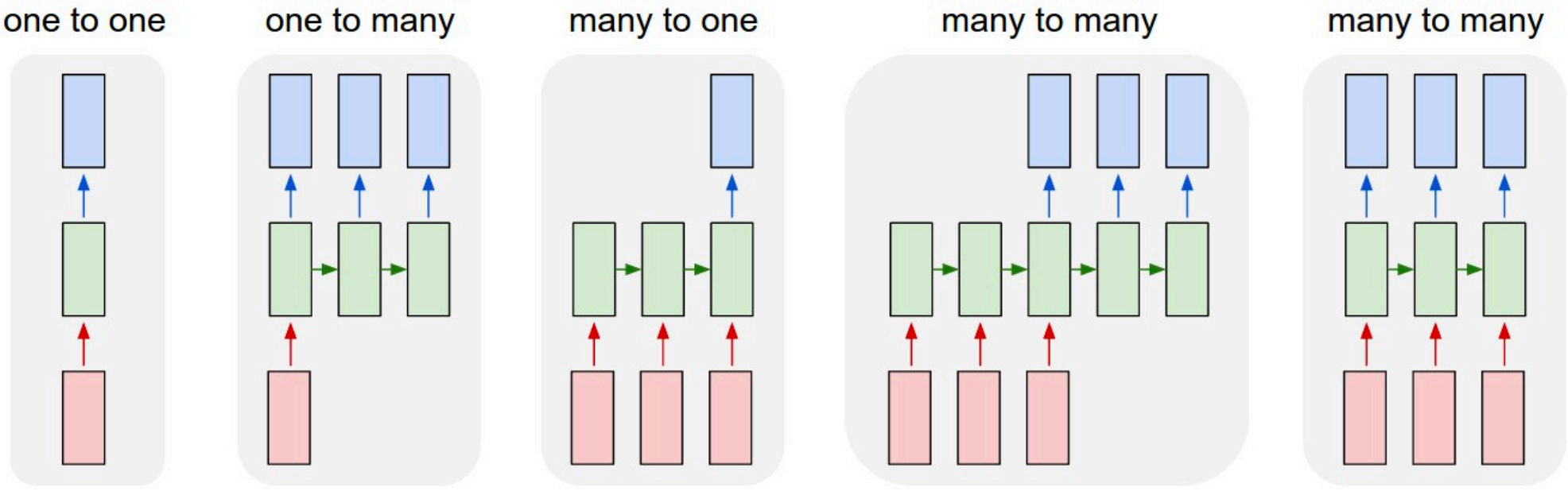
Different types of neural networks

Introduction



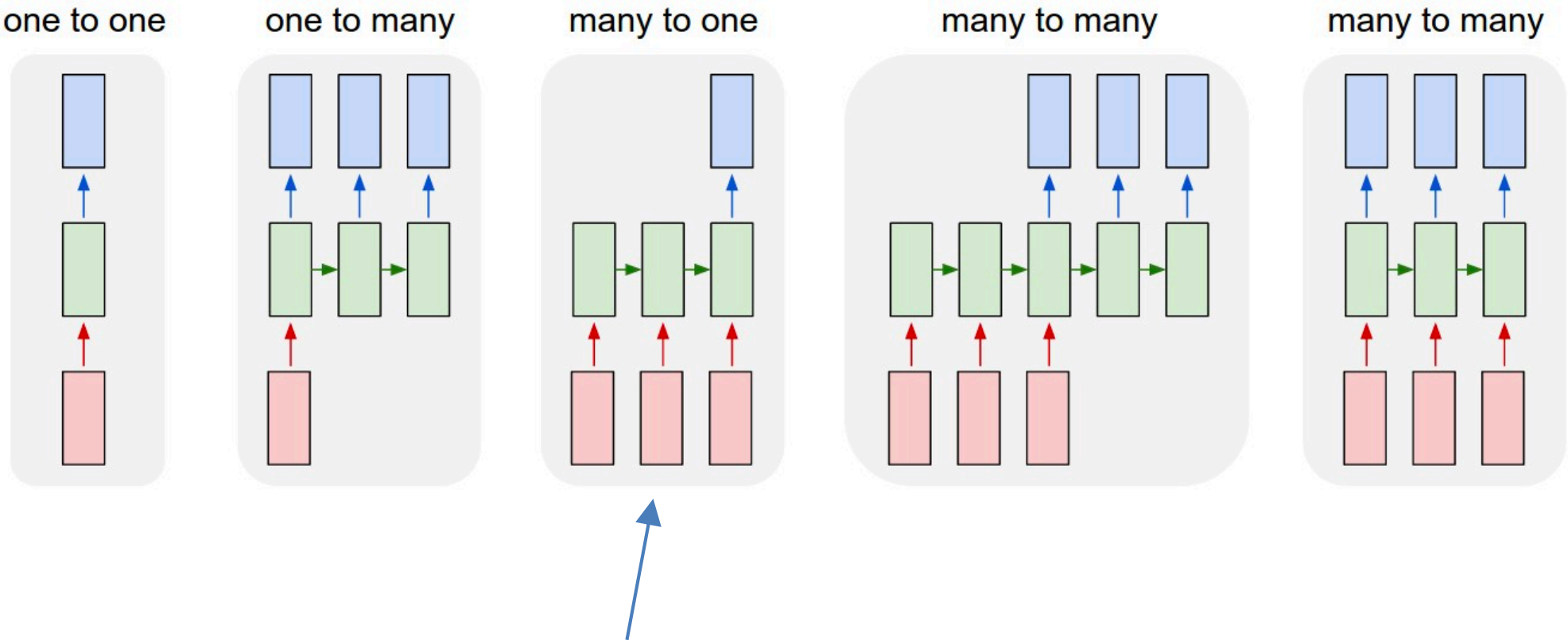
Vanilla Neural Networks

Introduction



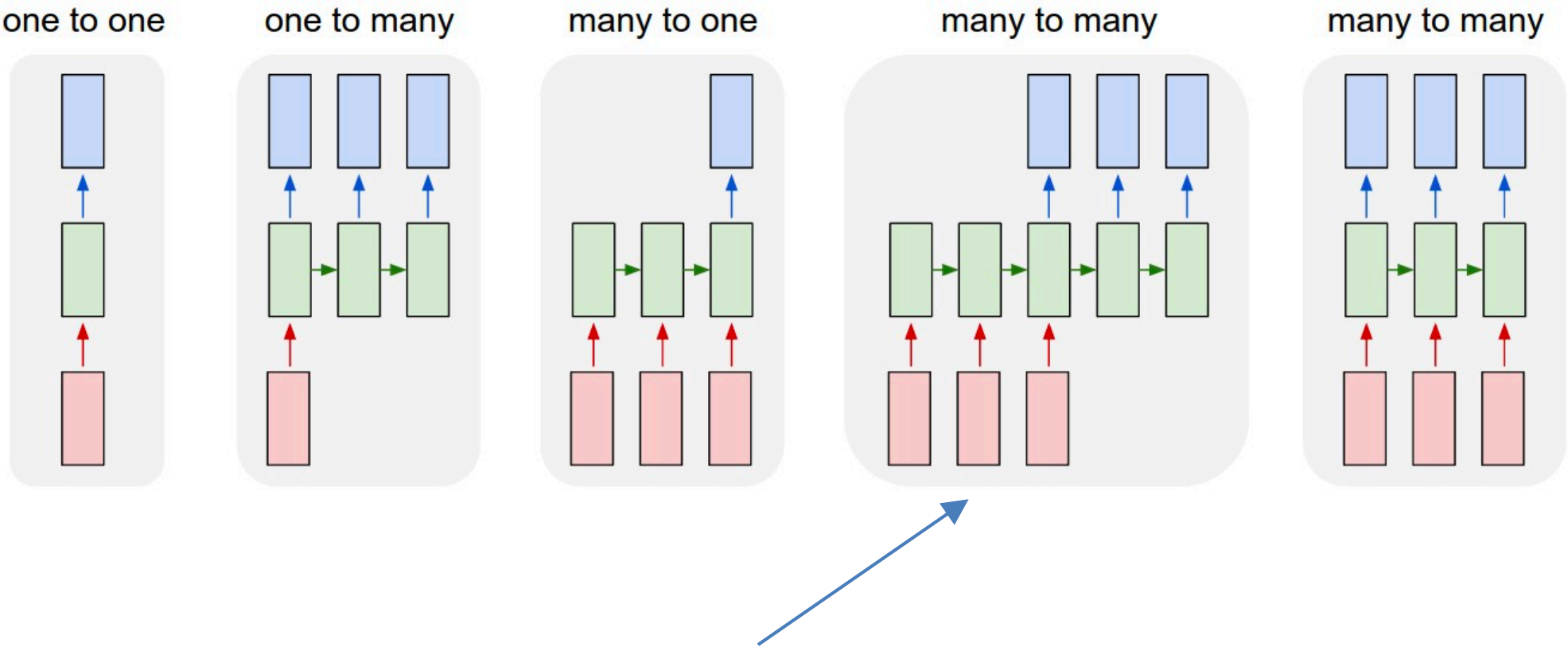
e.g., Image Captioning (image → sequence of words)

Introduction



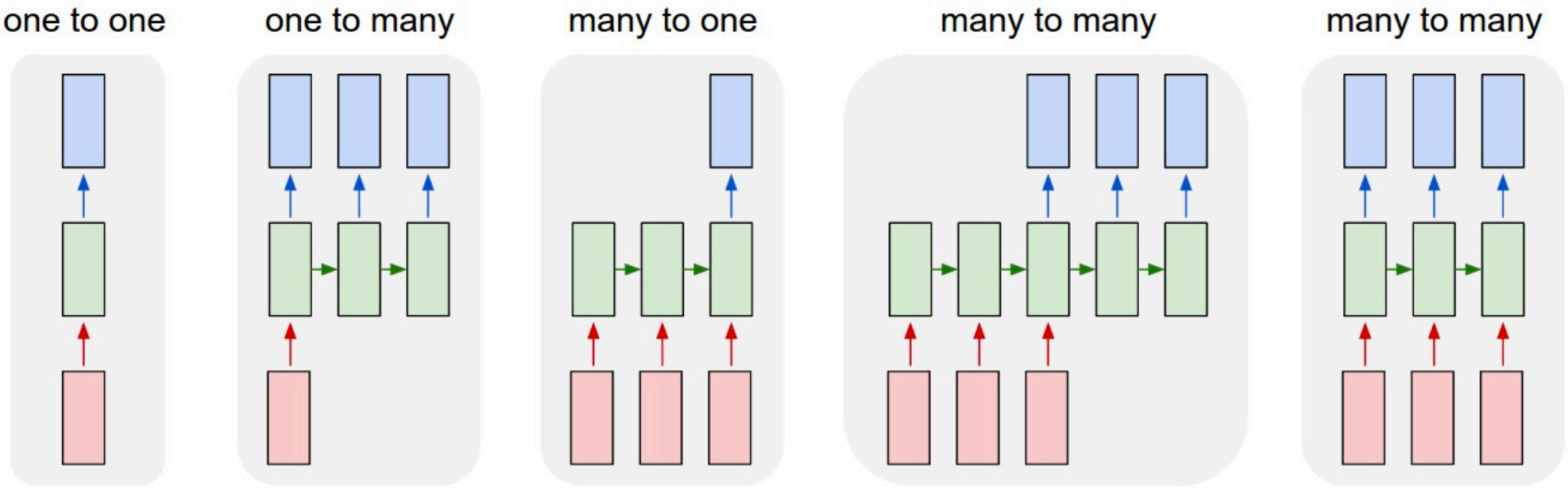
e.g., Sentiment Classification (sequence of words → sentiment)

Introduction



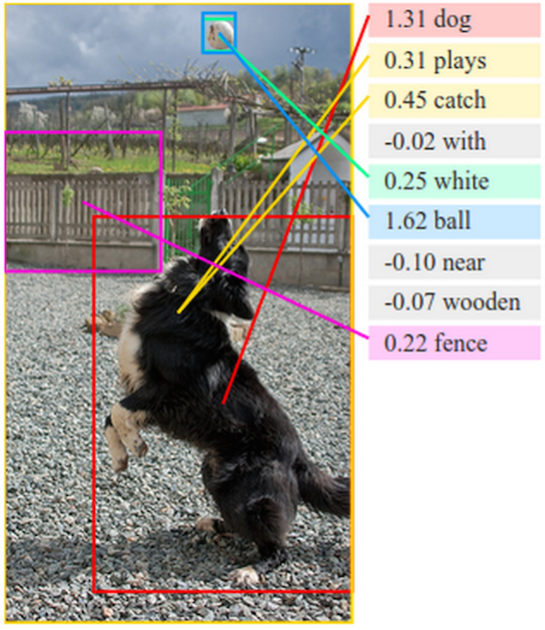
e.g., Machine Translation (seq of words → seq of words)

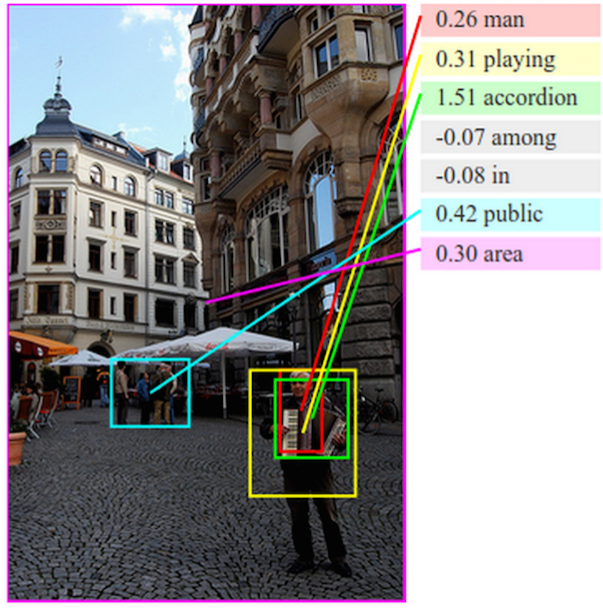
Introduction

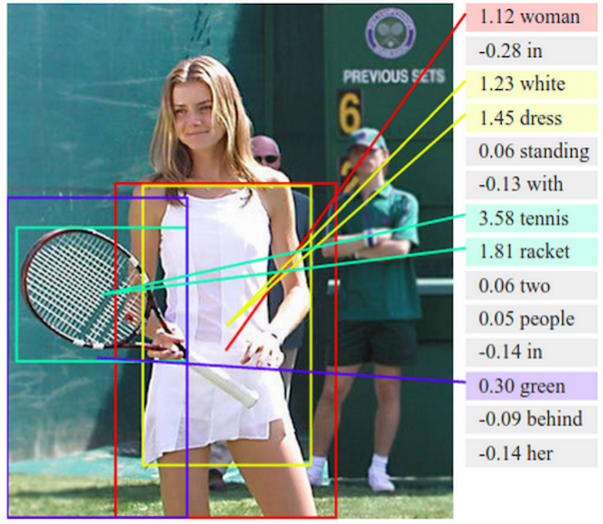


e.g., Video Classification on frame level

RNN







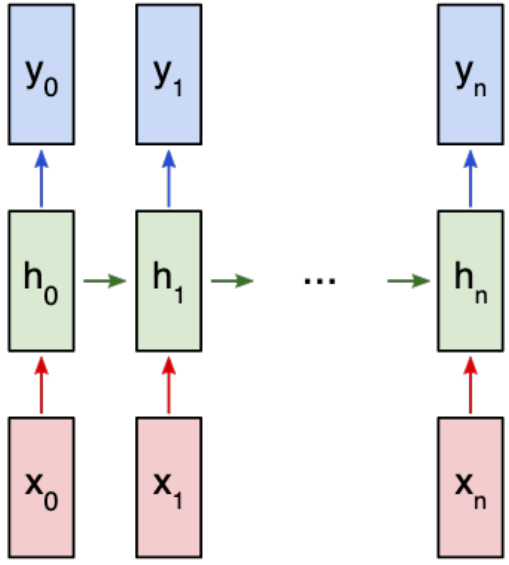
Deep Visual-Semantic Alignments for Generating Image Descriptions

RNN

Let's consider a "many to many" RNN with inputs $x_0, x_1, \ldots x_n$ that wants to produce outputs $y_0, y_1, \ldots y_n$. These x_i and y_i are **vectors** and can have arbitrary dimensions.

RNNs work by iteratively updating a hidden state h, which is a vector that can also have arbitrary dimension. At any given step t,

1. The next hidden state h_t is calculated using the previous hidden state h_{t-1} and the next input x_t.
2. The next output y_t is calculated using h_t.



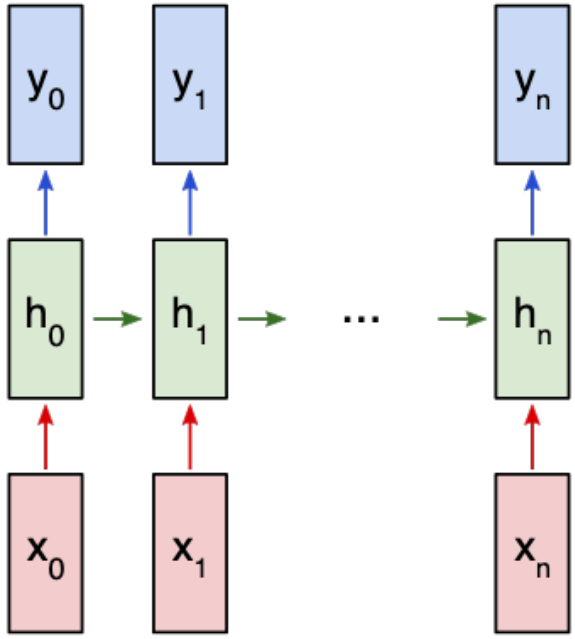
RNN

Here's what makes a RNN *recurrent*: **it uses the same weights for each step**.
More specifically, a typical vanilla RNN uses only 3 sets of weights to perform its calculations:

- W_{xh}, used for all $x_t \rightarrow h_t$ links.
- W_{hh}, used for all $h_{t-1} \rightarrow h_t$ links.
- W_{hy}, used for all $h_t \rightarrow y_t$ links.

We'll also use two biases for our RNN:

- b_h, added when calculating h_t.
- b_y, added when calculating y_t.



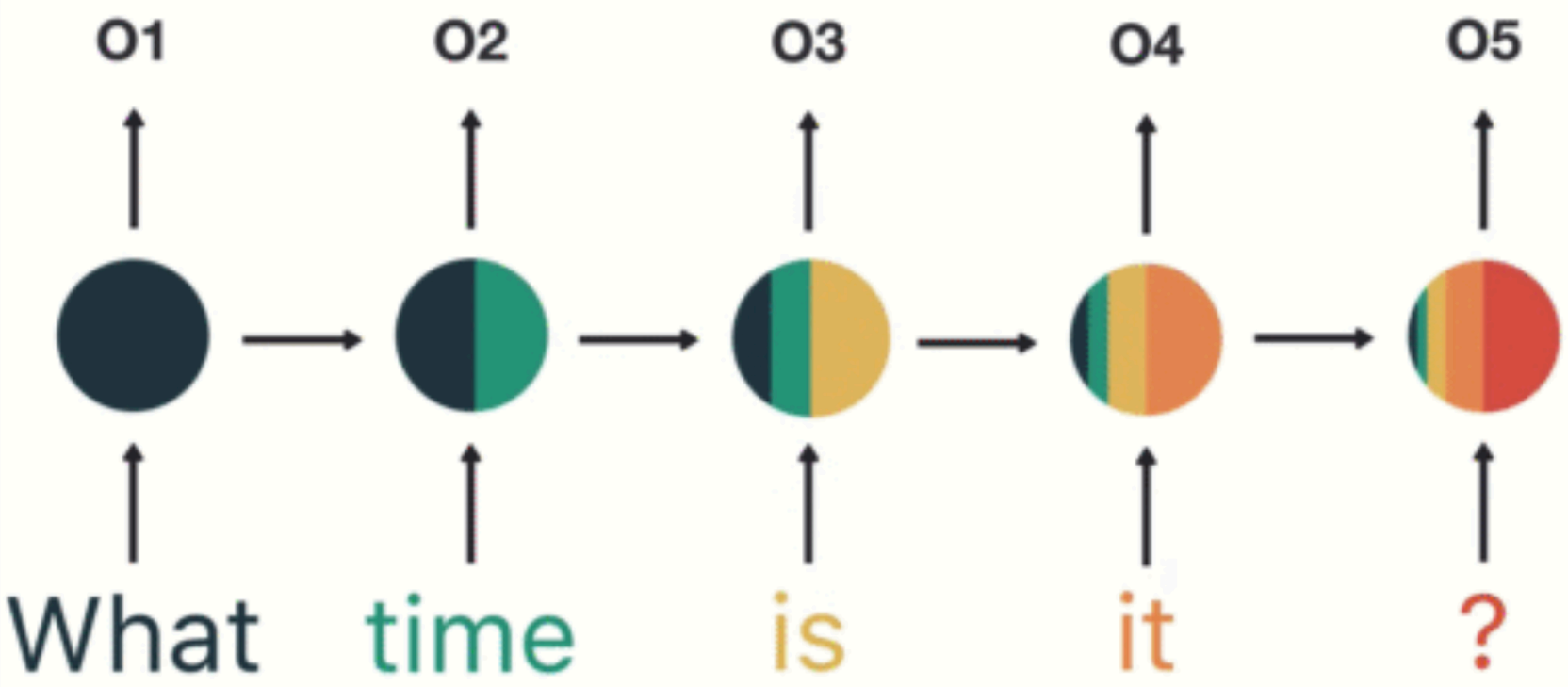
$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$y_t = W_{hy}h_t + b_y$$

Problem with RNN

Analyze the sentence “What time is it ?”

Asking for time



Problem with RNN

Analyze the sentence “What time is it ?”

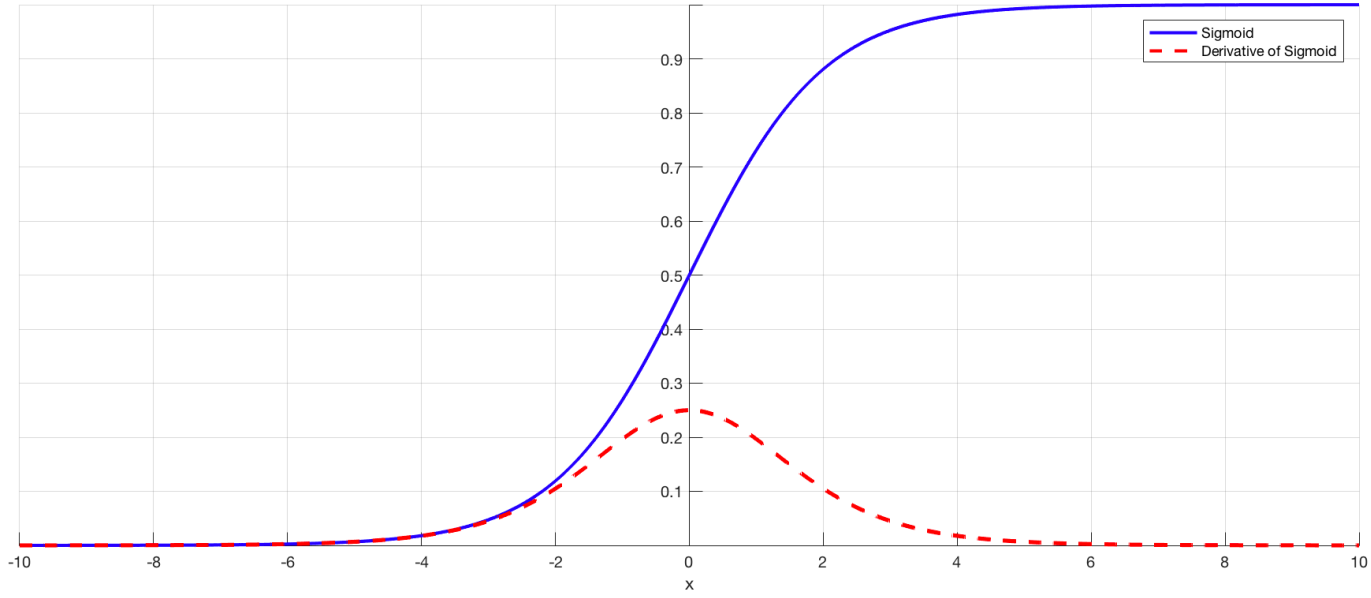
Final hidden state of the RNN

The prediction is made mostly by “is it ?” → RNN is known as **short-term memory**

Vanishing Gradient

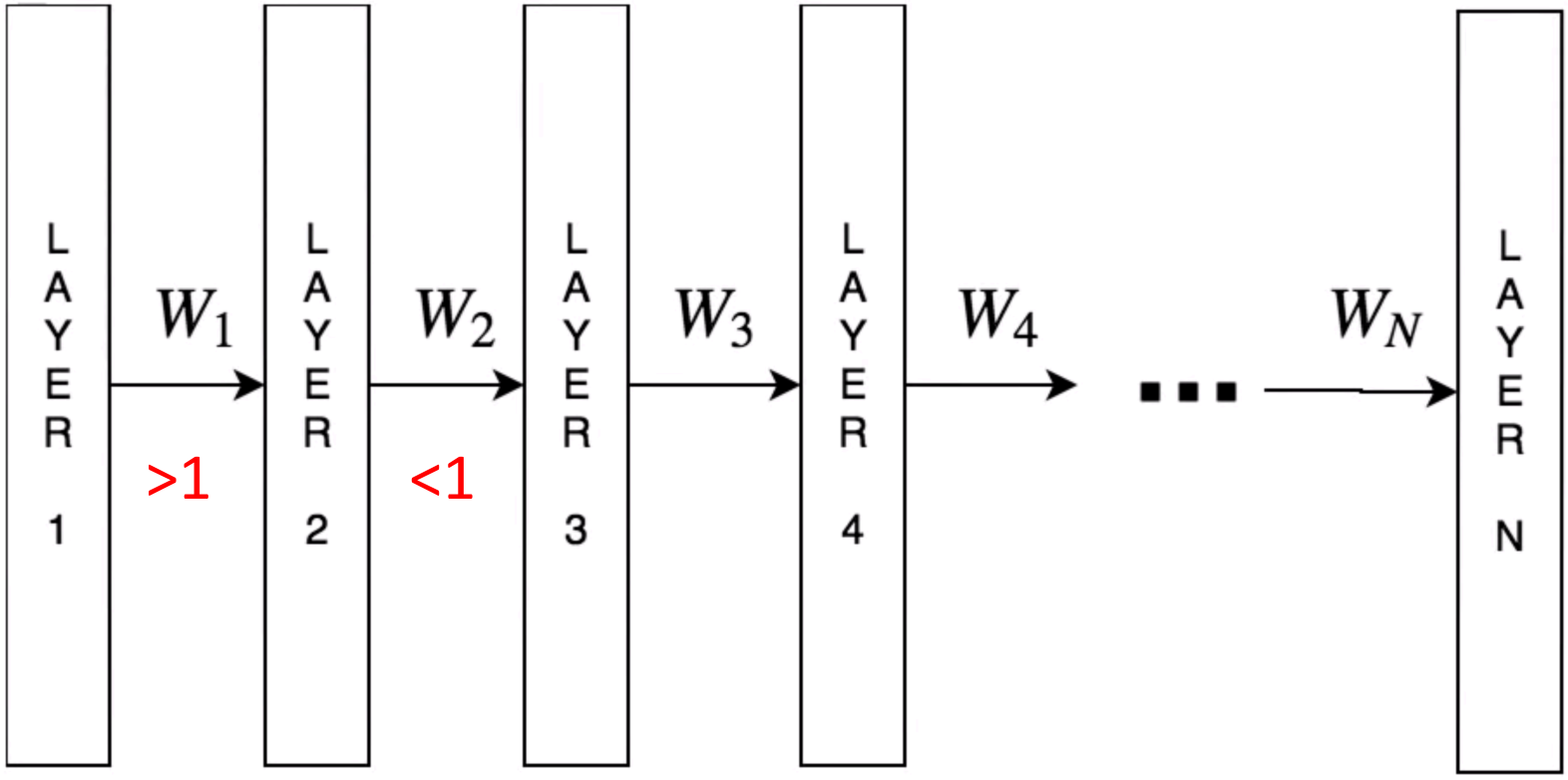
The short-term memory is caused by **Vanishing Gradient**, which also happens in other types of neural networks.

As more layers using certain activation functions (e.g., sigmoid) are added to neural networks, the **gradients of the loss function approaches zero**, making the network hard to train.



Vanishing Gradient

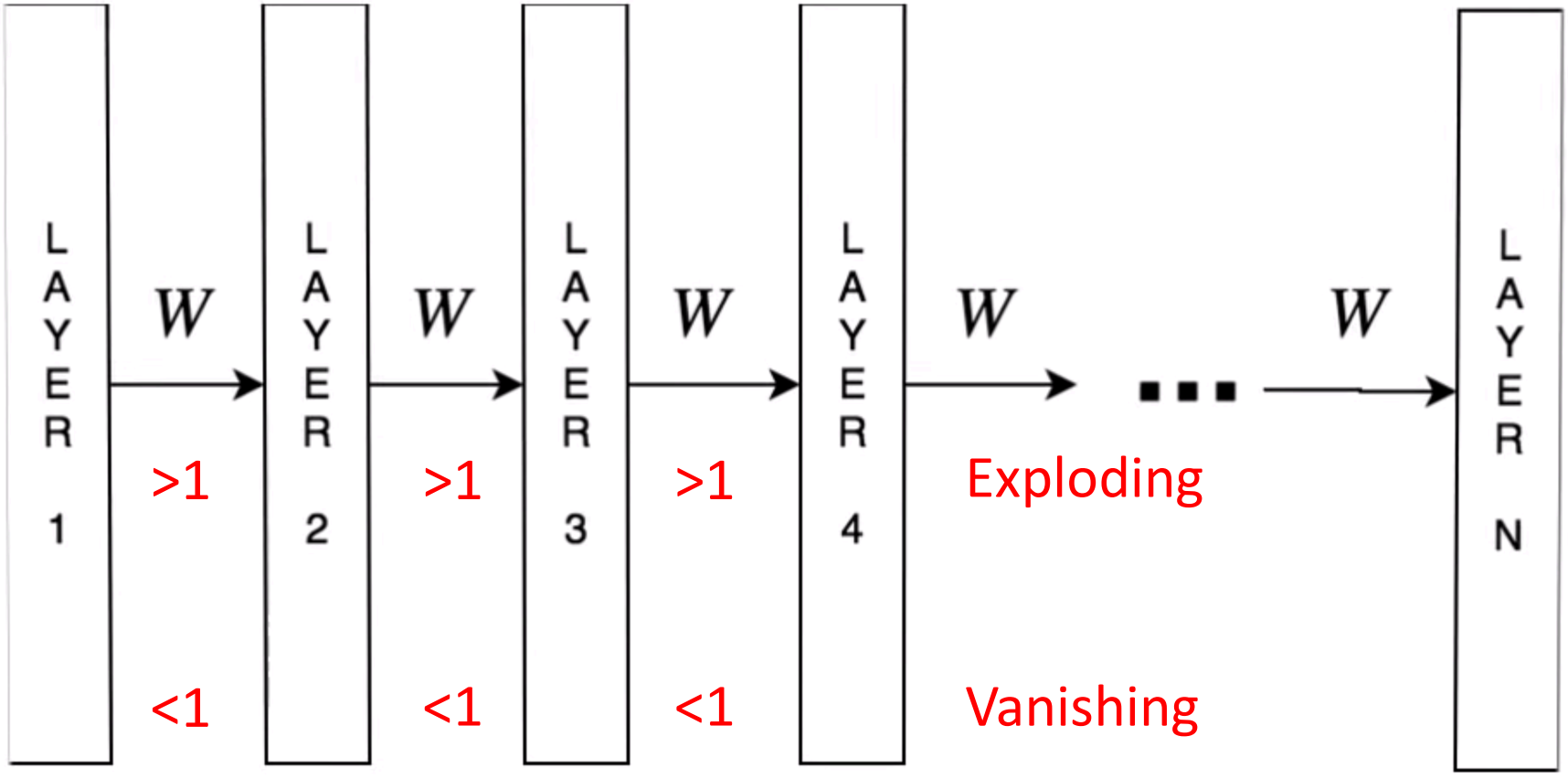
Vanishing gradient also happens in DNN, but much worse in RNN.



DNN - Heterogeneous weights W

Vanishing Gradient

Vanishing gradient also happens in DNN, but much worse in RNN.

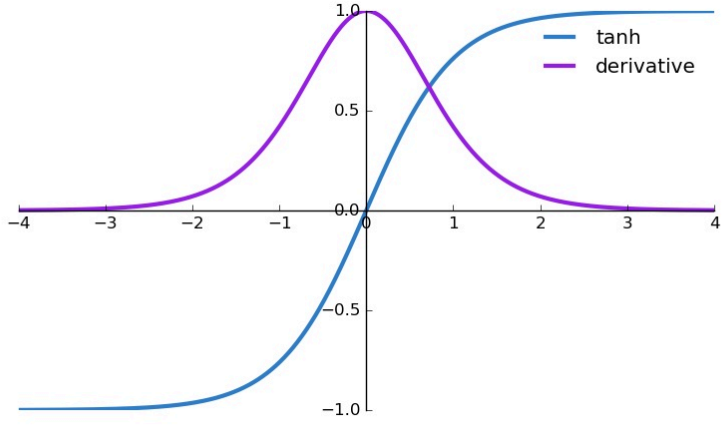


RNN - Homogeneous weights W

Vanishing Gradient

For shallow network with only a few layers that use these activations, this isn't a big problem. However, when more layers are used, it can cause the gradient to be too small for training to work effectively.

However, when n hidden layers use an activation like the sigmoid function, n small derivatives are multiplied together. Thus, the gradient decreases exponentially as we propagate down to the initial layers.

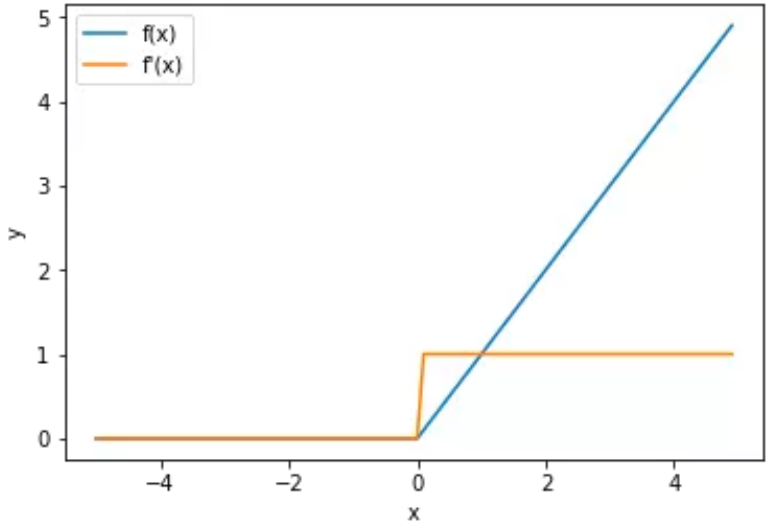


tanh distributes gradient better than sigmoid

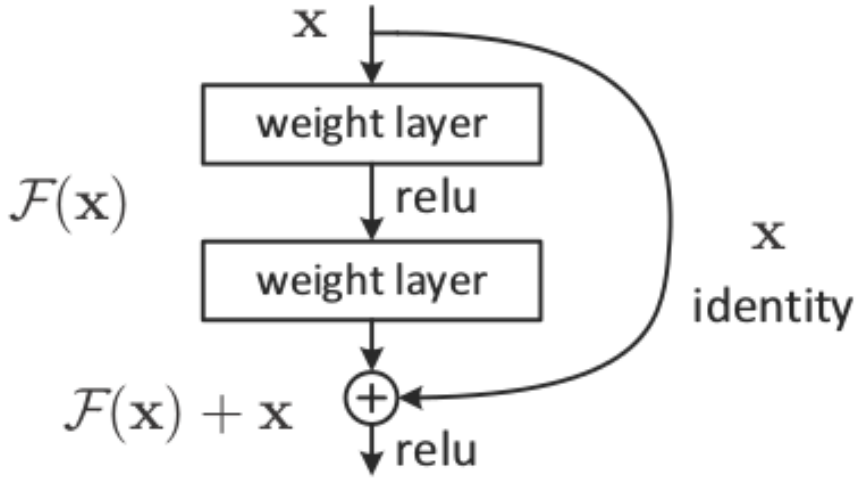
Vanishing Gradient Solutions

To solve the vanishing gradient issue:

- Use other activation functions, such as ReLU, which doesn't cause a small derivative
- Use residual connections as they provide connections straight to earlier layers.
- LSTM, GRU: use gating (pass or block / 1 or 0) function, not activation function → able to let the information go through



Derivative of ReLU



Residual block

Exploding Gradient

Exploding gradient is a problem where large error gradients **accumulate** and result in very **large updates** to neural network model weights during training.

Why gradient explodes:

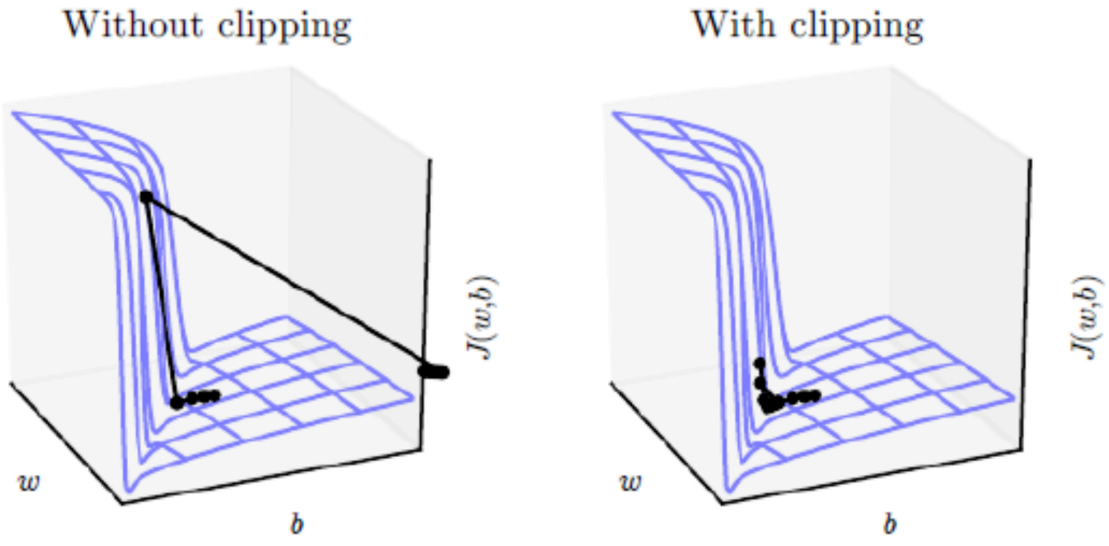
- Poor choice of **learning rate** results in large weight updates.
- Poor choice of **loss function**, allowing the calculation of large error values.
- Poor **initialization of weight matrices** with very large values.

How to solve:

- Smaller learning rate
- Weight regularization
- Gradient scaling: normalizing the error gradient vector such that vector norm (magnitude) equals a defined value, such as 1.0.
- Gradient clipping: forcing the gradient values (element-wise) to a specific minimum or maximum value if the gradient exceeded an expected range.

→ only **change the magnitude**, **not the direction** of the gradient vector.

Exploding Gradient – Gradient clipping



$$\text{if } \|g\| > \ threshold$$

$$g \leftarrow \frac{threshold \times g}{\|g\|}$$

where: g is the gradient and

$\|g\|$ *is the norm of the gradient*

Tensorflow: tf.clip_by_norm

Pytorch: torch.nn.utils.clip_grad

LSTM (Long Short-Term Memory)

Problems with RNN

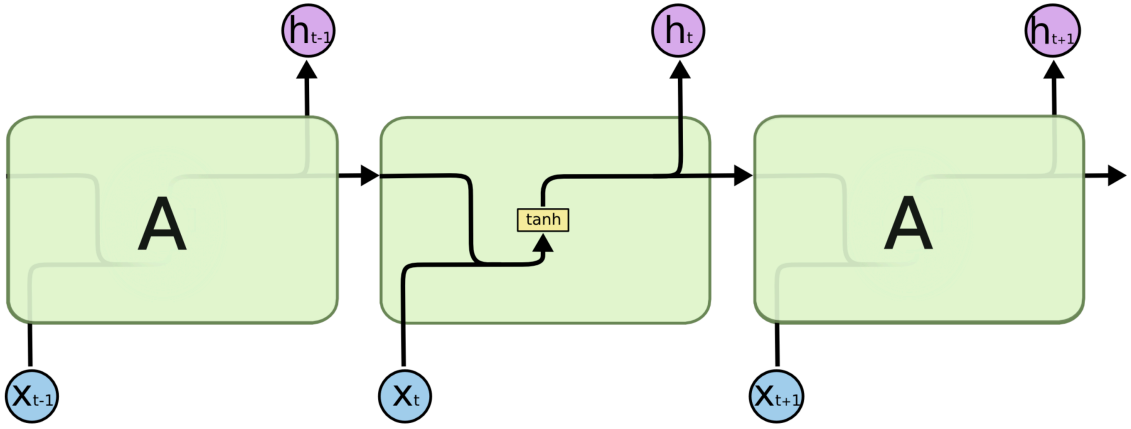
- Short-term memory
- Not easy to train (vanishing gradient)

→ Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU)

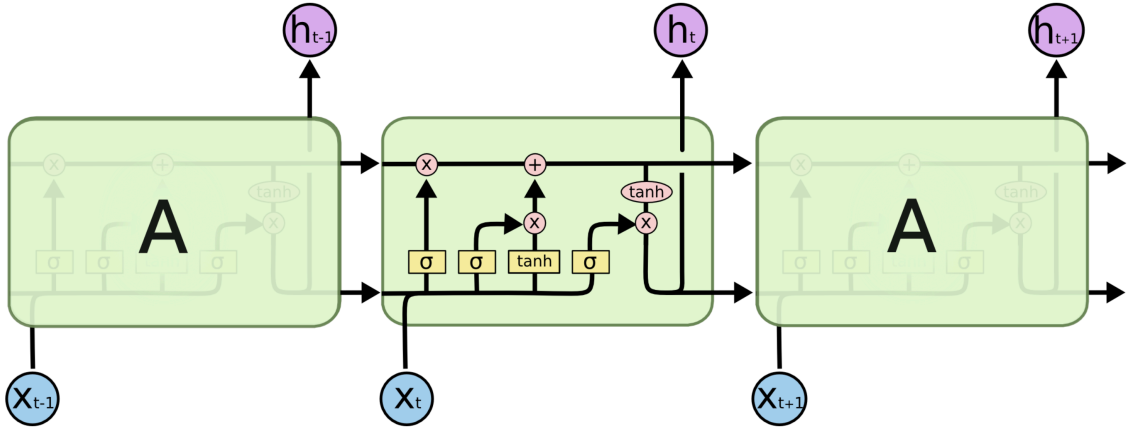
What is LSTM:

- Special kind of RNN.
- Able to learn long-term dependencies.
- But LTSM does not learn everything. It is capable of **selectively learning**!

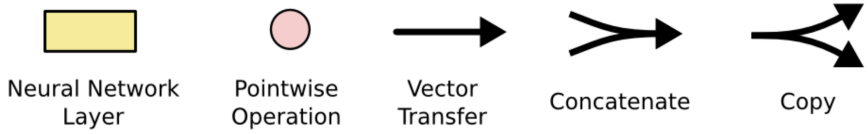
LSTM (Long Short-Term Memory)



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



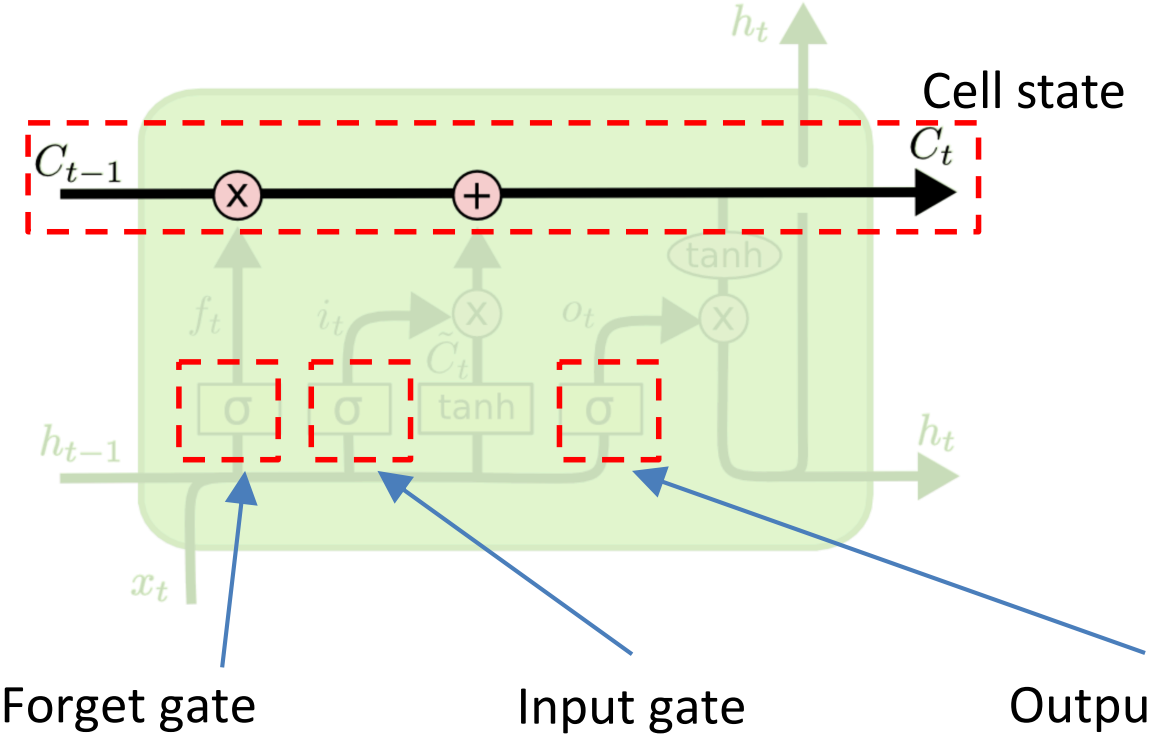
The repeating module in an LSTM contains four interacting layers.

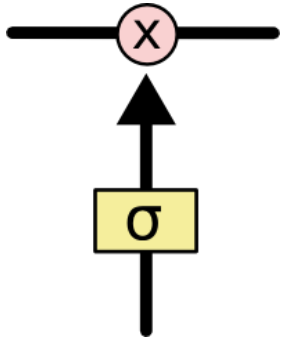


LSTM – Core ideas

The key to LSTM is the **cell state**, the horizontal line running through the top of the diagram → information can flow through (long term memory)

Sigmoid activation function represent gates. Gates are a way to **optionally let information through**.





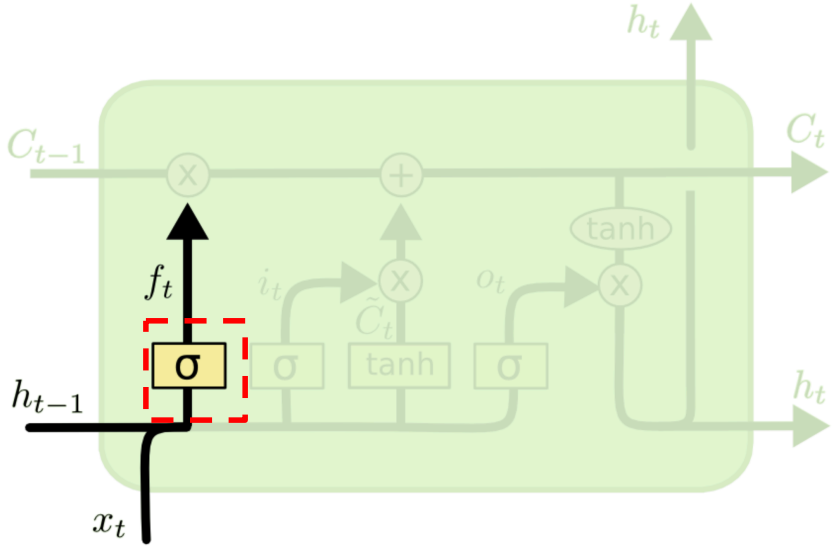
LSTM – Forget gate

It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}. A **1** represents "completely keep this" while a **0** represents "completely get rid of this." → **decide what to keep and what to omit**

Example: In cell state: Tom is a boy. The sky is blue.

Input x_t: He

→ cell state now selectively remembers "Tom is a boy". "The sky is blue" is omitted.



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

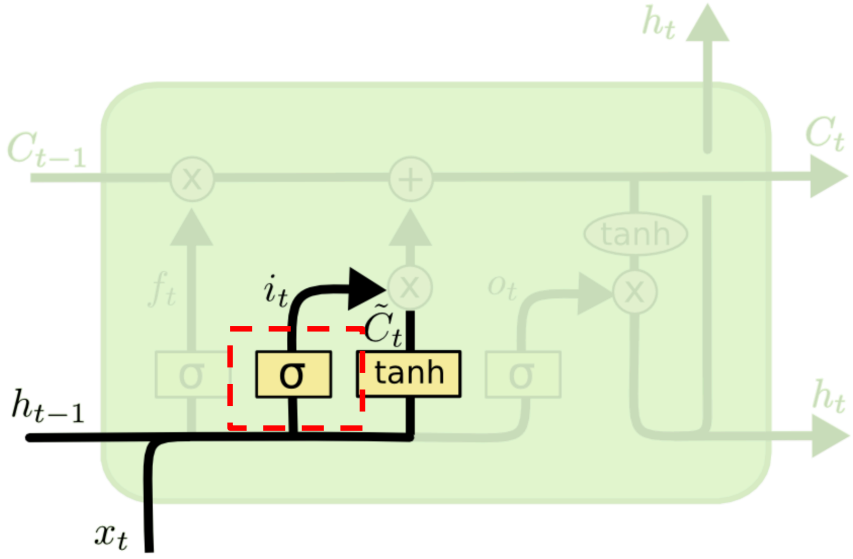
LSTM – Input gate

It looks at h_{t-1} and x_t, get them through a sigmoid function and a tanh function to output i_t and $\tilde{C}_t$ to selectively add information to the cell sate, then update the old cell state C_{t-1} to C_t → **decide what to add (sigmoid) and how to add (tanh → regularize [-1,1]) to cell state.**

Example: In cell state: Tom is a boy.

Input x_t: He

→ cell state C_{t-1} is added the pronoun information He, so that it knows the next word prediction will be "is", and becomes cell state C_t



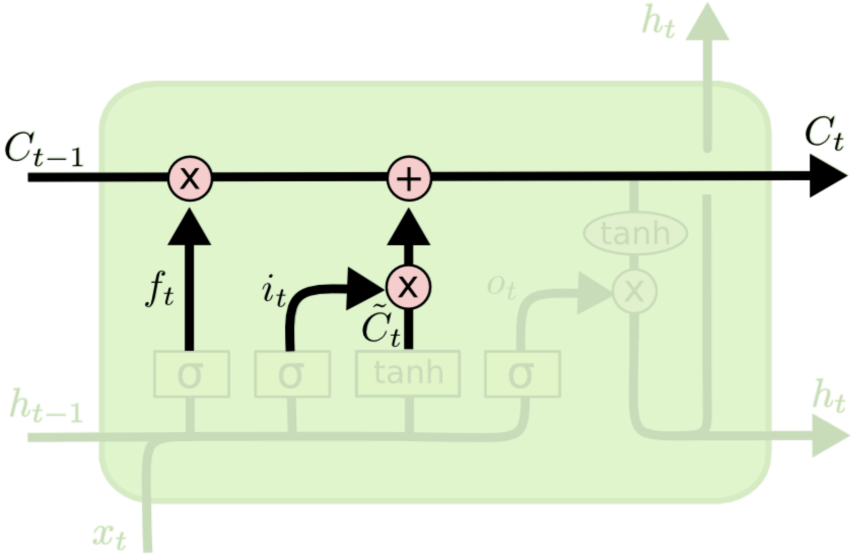
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

LSTM – Update cell state

Update the old cell state C_{t-1} to C_t

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

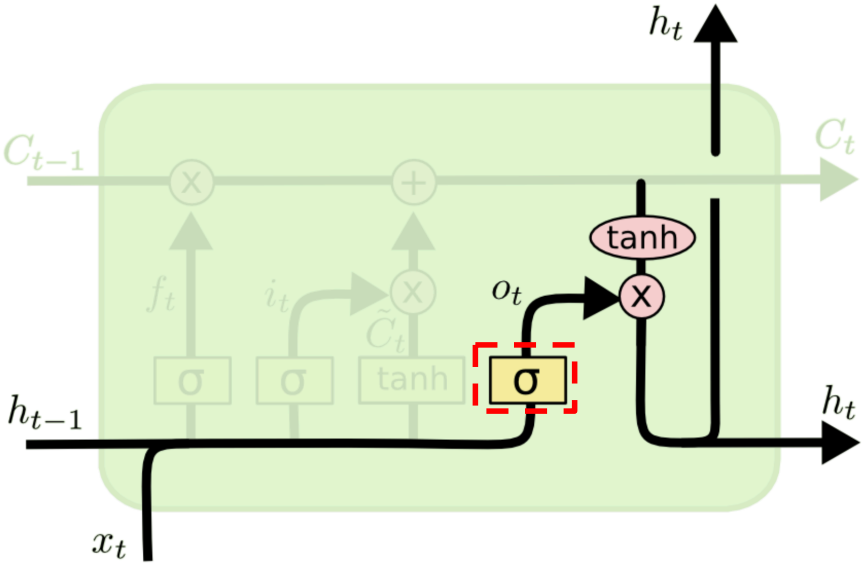


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

LSTM – Output gate

The output (hidden state h_t) will be based on the cell state, but will be a filtered version. Sigmoid layer decides what part of the cell state we're going to output, then regularize it using tanh ([-1,1]) → decides what to output

After adding information He to Cell state C_{t-1} (now C_t), it knows the next word prediction may be "is" (output h_t)

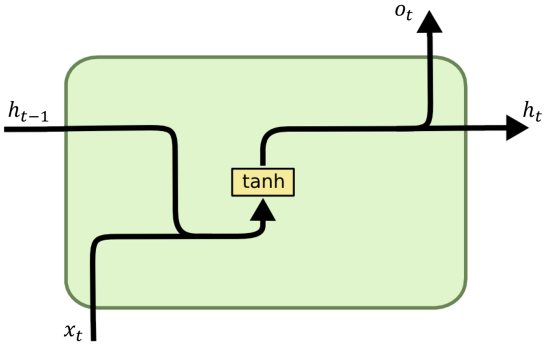


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

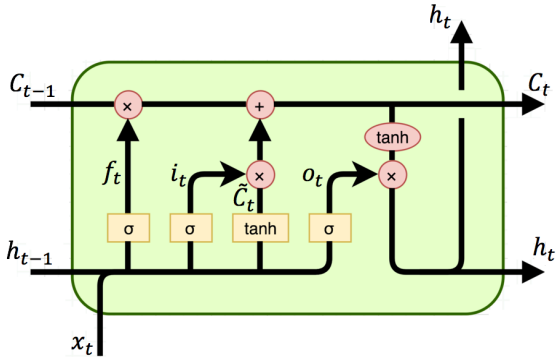
$$h_t = o_t * \tanh\left(C_t\right)$$

GRU (Gated Recurrent Unit)

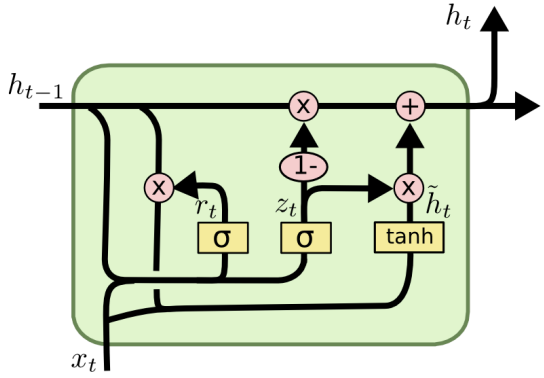
RNN

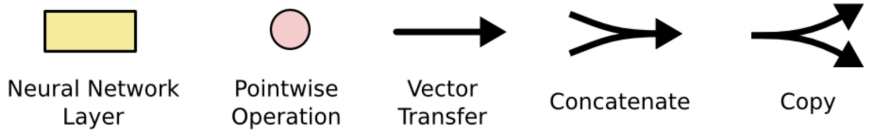


LSTM



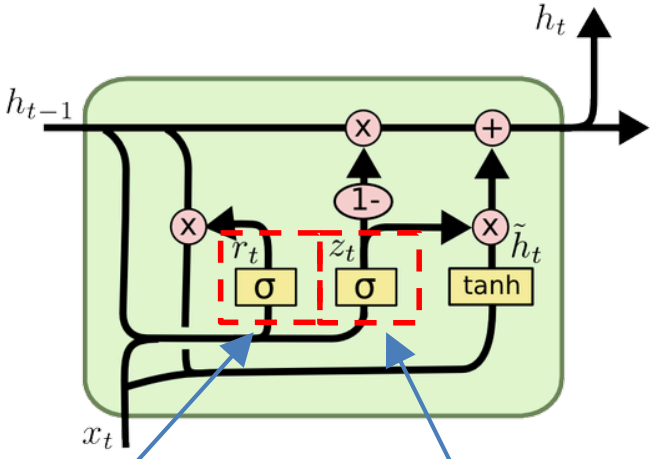
GRU





GRU

- **Sigmoid** activation function represents gate which allows a GRU to carry forward information over many time periods in order to influence a future time period.
  - **Reset gate (r_t):** decide **how much** past information to **forget**
  - **Update gate (z_t)**: decides what information to **throw away** and what new information to **add**.



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

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

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

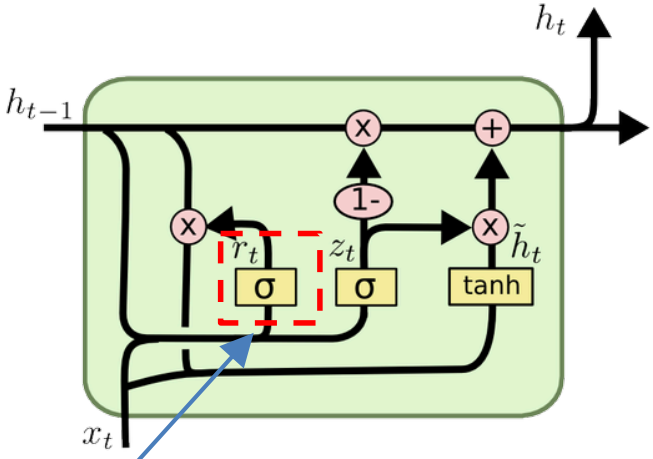
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Reset gate

Update gate

GRU– Reset gate

Reset gate (r_t): decide **how much** past information to **forget**. It calculates the reset gate r_t at time step t.



Reset gate

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

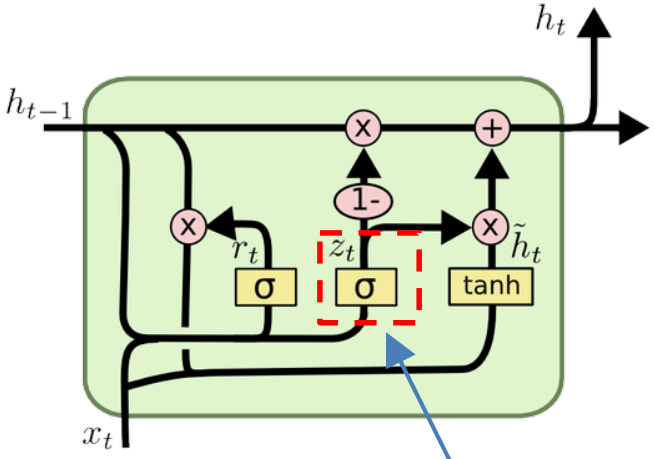
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

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

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

GRU – Update gate

- **Update gate (z_t)**: decides what information to **throw away** and what new information to **add**. This gate calculates the update gate z_t at time step t.
- Similar with the reset gate and differs only in the weights and the gate's usage.
- Given the update gate, the issue of the **vanishing gradient** is eliminated since the model on its own learn how much of the past information to pass to the future.



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

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

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

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

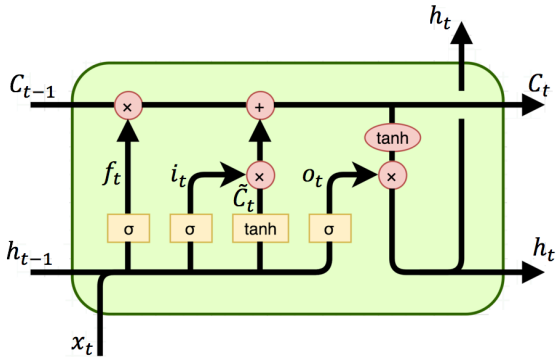
Update gate

GRU vs. LSTM

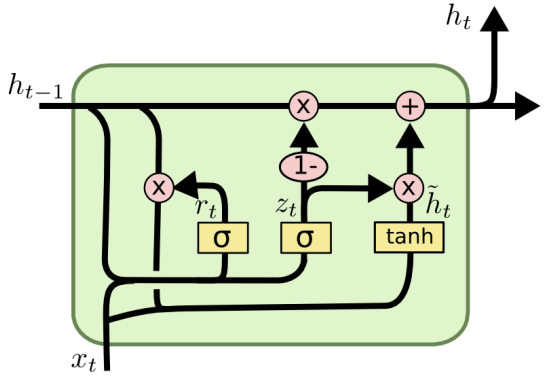
- GRU has **fewer tensor operations**, thus (little) **faster** to train then LSTM.
- GRU and LSTM have comparable performance and there is no simple way to recommend one or the other for a specific task.

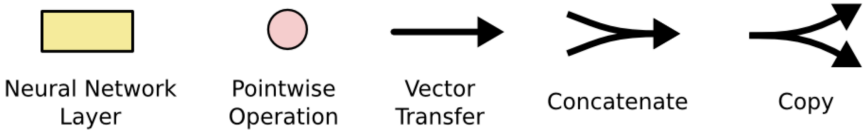
→ try both to determine which one works better for a specific use case

LSTM



GRU





Summary

- Different types of neural networks: 1-1, 1-N, N-M, N-N.
- Recurrent Neural Network (RNN)
- Vanishing Gradient
- Exploding Gradient
- Long Short-Term Memory
- Gated Recurrent Unit

Q&A