Last time..

CapsNets

Sequence Models / Recurrent Neural Networks

(Vanilla) RNN, LSTM, GRU

Some references / Links

- MIT 6.S191: Recurrent Neural Networks (video)
  - MiT 6.S191: Introduction to Deep Learning (course)
- Stanford: CS231n: Recurrent Neural Networks (video)
  - Stanford: CS231n: Recurrent Neural Networks (course)
- RNNs Course (Deep Learning, Andrew Ng)

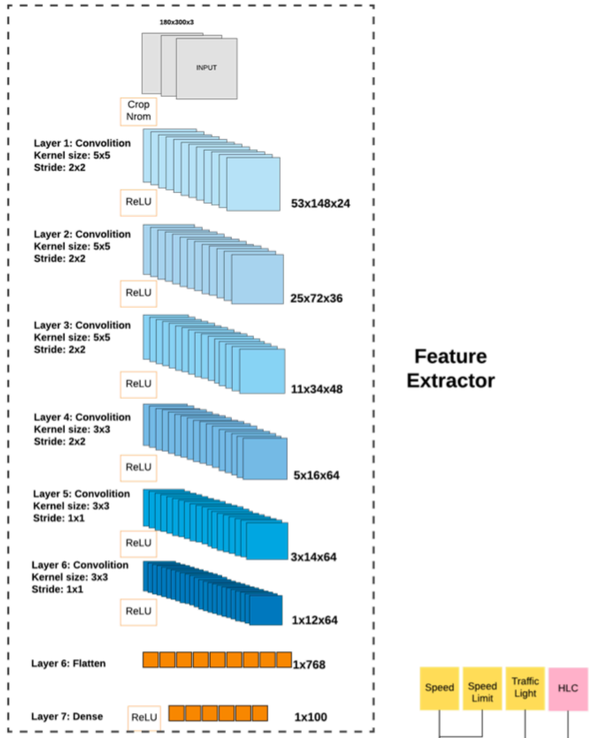
Sequence Models

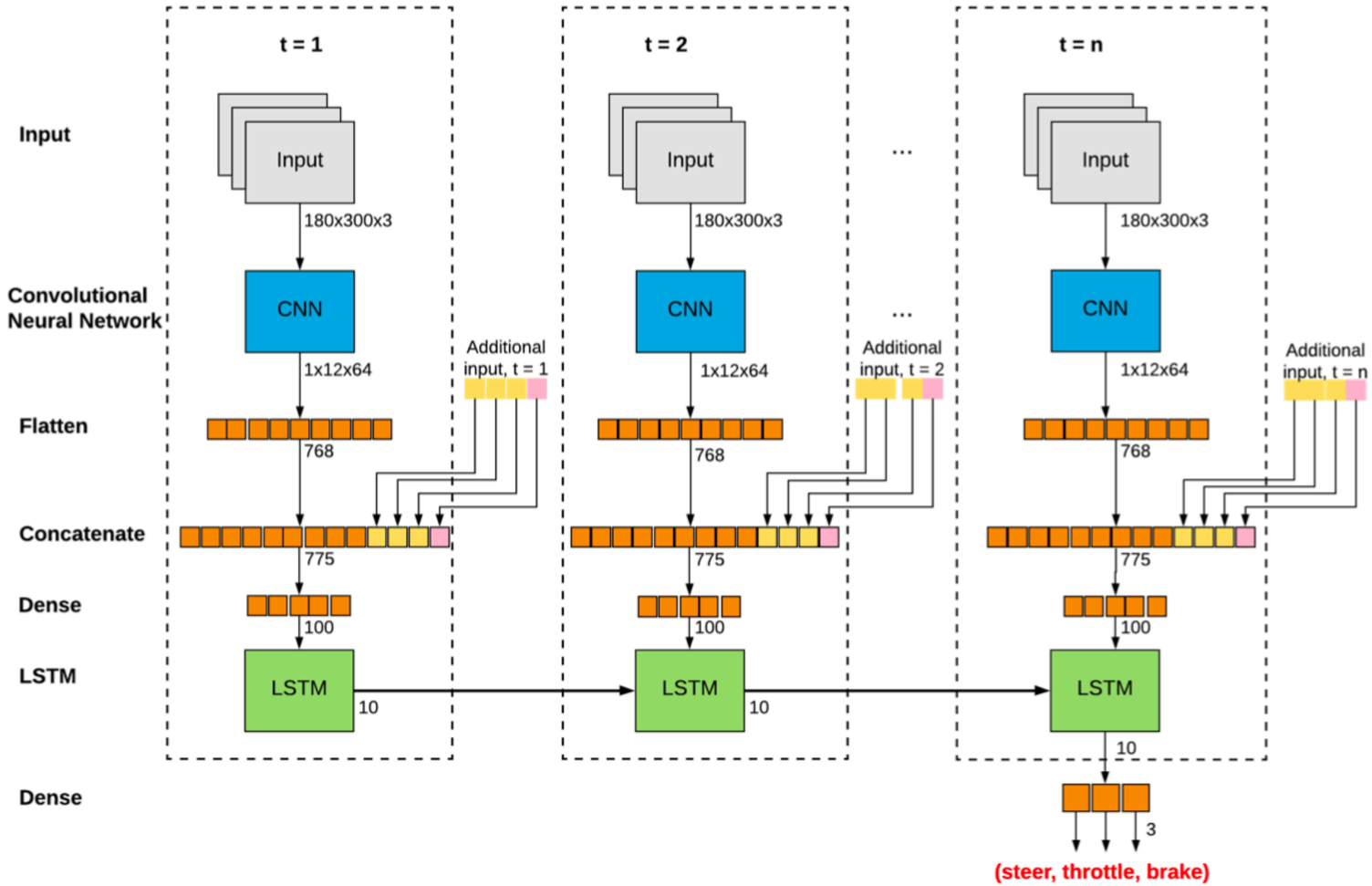
High-level description

RNN: Applications

- CV: for the most part CNNs in combinations with RNNs
  - End-to-end learning for AVs
  - Image Captioning
  - Visual Question Answering

End-to-end learning for AVs





Autonomous Driving Using a CNN-LSTM: An Example of End-to-end Learning

Image Captioning: Example Results

Captions generated using neuraltalk2
All images are CC0 Public domain:
cat suitcase, cat tree, dog, bear,
surfers, tennis, giraffe, motorcycle

A cat sitting on a suitcase on the floor

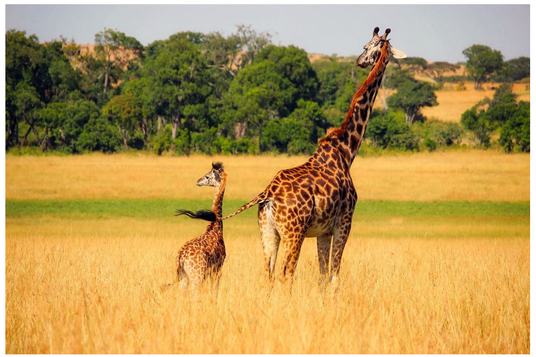
A cat is sitting on a tree branch

A dog is running in the grass with a frisbee

A white teddy bear sitting in the grass

Two people walking on the beach with surfboards

A tennis player in action on the court



Two giraffes standing in a grassy field

A man riding a dirt bike on a dirt track

Visual Question Answering

Q: What endangered animal is featured on the truck?

A: **A bald eagle.**
A: A sparrow.
A: A humming bird.
A: A raven.

Q: Where will the driver go if turning right?

A: **Onto 24 ¾ Rd.**
A: Onto 25 ¾ Rd.
A: Onto 23 ¾ Rd.
A: Onto Main Street.

Q: When was the picture taken?

A: **During a wedding.**
A: During a bar mitzvah.
A: During a funeral.
A: During a Sunday church service

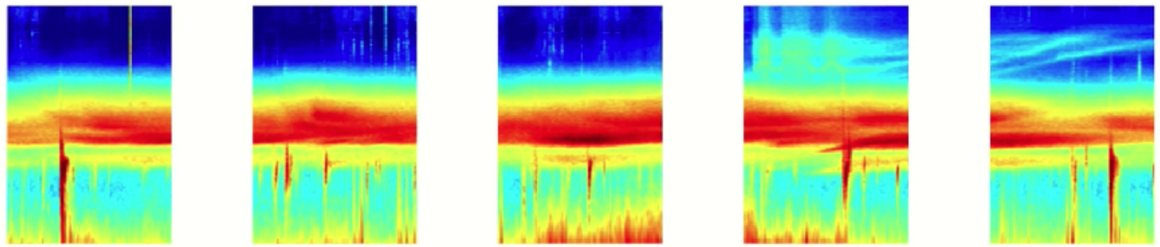
Q: Who is under the umbrella?

A: **Two women.**
A: A child.
A: An old man.
A: A husband and a wife.

Agrawal et al, “VQA: Visual Question Answering”, ICCV 2015
Zhu et al, “Visual 7W: Grounded Question Answering in Images”, CVPR 2016

RNN: Sequence Data

- Understanding: thought experiment:
  - moving ball, predict direction?
  - record one **snapshot:** guess / random (Without knowledge of where the ball has been, you wouldn't have enough data to predict where it's going)
  - record **many** snapshots: enough information to make a good prediction
- Sequence data comes in many forms:
  - Text: sequence of words (seq. of chars)
  - Audio: chop up the spectrogram
  - Video: sequence of frames



RNN: Sequential Memory

- Intuition:
  - say the alphabet in your head:
    - **easy**, if you were taught this specific sequence
  - try saying the alphabet backward:
    - **harder**, unless you've practiced
  - what happens if you start from «F»:
    - **struggle** first, and then the rest will come **naturally**
  - Sequential memory is a mechanism that makes it easier for your brain to **recognize sequence patterns**

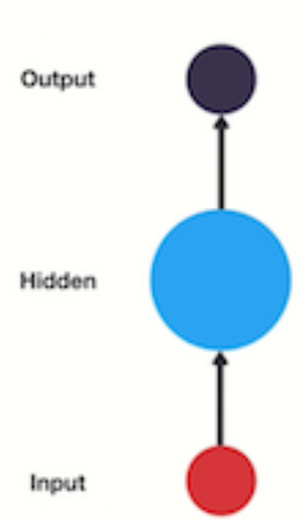
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Z Y X W V U T S R Q P O N M L K J I H G F E D C B A

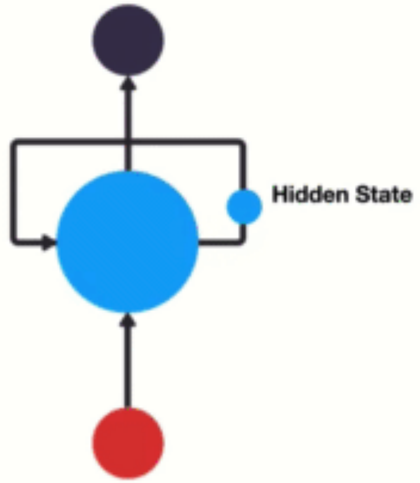
F

RNN: Architecture

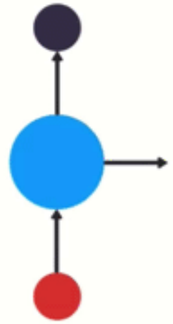
- FF-NN: input, hidden, output layer
- How to use previous information to effect future predictions.
- Add a loop in the network that can pass prior information forward = RNN
- An RNN has a **looping** mechanism that acts as a highway to allow information to flow from one step to the next: **hidden state**, which is a representation of previous inputs.
- An RNN can be unrolled for better understanding.



Feed Forward Neural Network

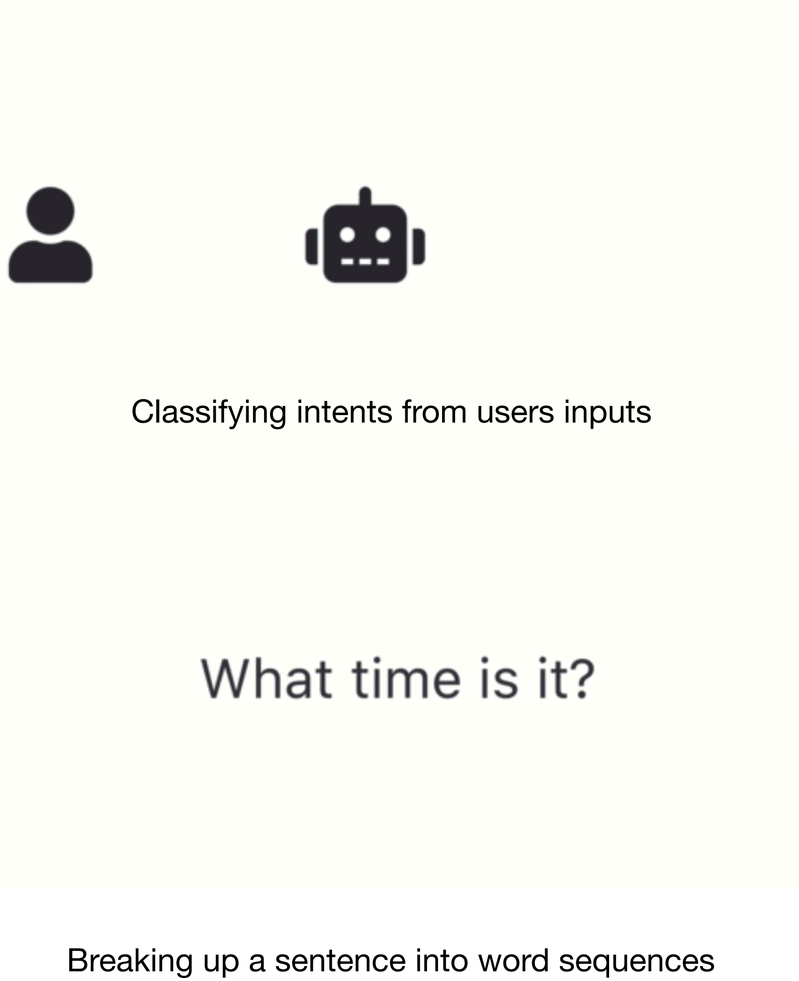


Recurrent Neural Network



RNN: Example: Chatbot

- Classify **intentions** from the users inputted text.
- First, **encode** the sequence of text using an RNN
- Then, **feed** the RNN output into a feed-forward neural network which will classify the intents
- User types: «What time is it?»
  - Break up the sentence into individual words

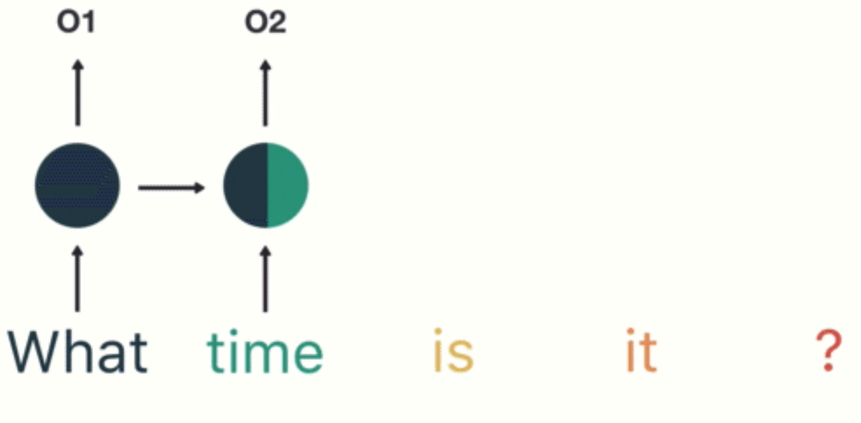


Classifying intents from users inputs

Breaking up a sentence into word sequences

RNN: Example: Chatbot (2)

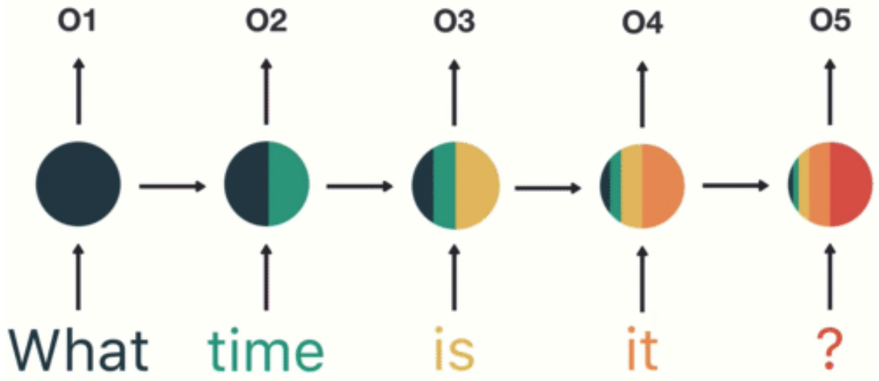
- RNN's work sequentially so we feed it one word at a time
  - First step is to **feed** "What" into the RNN. The RNN **encodes** "What" and produces an **output**.
  - Next step, we feed the word "time" and the hidden state from the previous step. The RNN now has information on both the word "What" and "time."

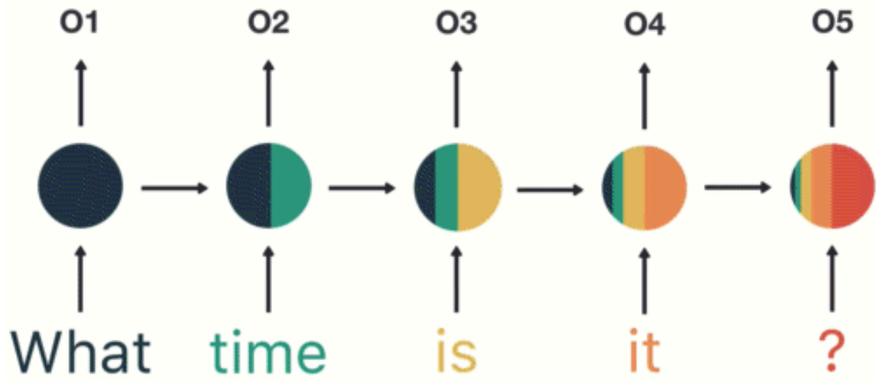


RNN: Example: Chatbot (3)

- We **repeat** this process until the final step. You can see by the final step that the RNN has **encoded** information from all the words in previous steps (color).

- Since the final output was **created from the rest** of the sequence, we should be able to take the **final output** and pass it to the feed-forward layer to **classify** an intent.





RNN: Example: Chatbot (4)

- Python **code** showcasing the control flow (**forward pass**)
  - **Initialize** your network layers and the initial hidden state
  - **loop** through your inputs, pass the word and hidden state into the RNN
  - RNN **returns** the output and a modified hidden state.
  - continue to loop until you're out of words
  - pass the last output to the feedforward layer, and it returns a **prediction**

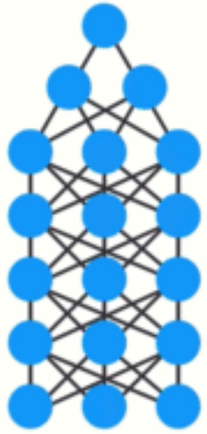
```
rnn = RNN()
ff = FeedForwardNN()
hidden_state =[0.0, 0.0, 0.0, 0.0]

for word in input:
    output, hidden_state = rnn(word, hidden_state)

prediction = ff(output)
```

Pseudo code for RNN control flow

RNN: Vanishing Gradient

- **Short-term memory**
  - Troubles **retaining** information from **previous** steps (distribution of colors in the hidden states: **information** from the word “what” and “time” is almost non-existent at the final time step)
  - Short-Term memory and the vanishing gradient is due to the nature of **back-propagation**
- **Training** a neural network has three major steps:
  - First, it does a **forward pass** and makes a **prediction**
  - Second, it **compares** the prediction to the ground truth using a **loss** function (the loss function outputs an **error** value which is an estimate of how poorly the network is performing)
  - Last, it uses that error value to do **back propagation** which calculates the **gradients for each weight** in the network

Final hidden state of the RNN



Forward / Backward pass

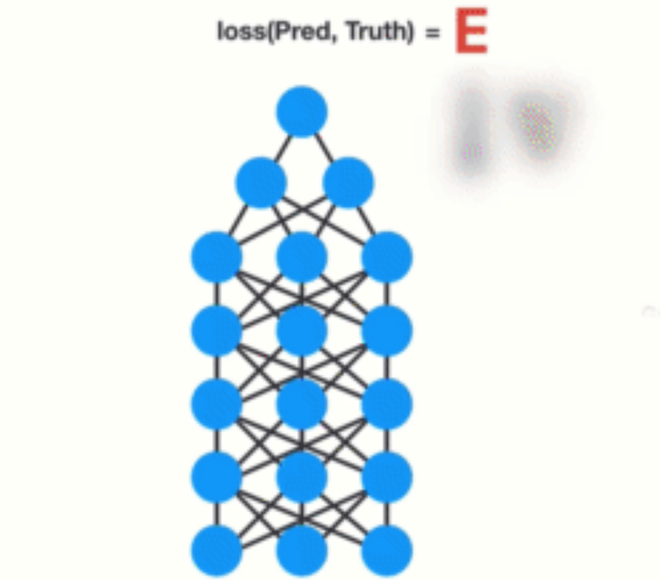
RNN: Vanishing Gradient (2)

- The **gradient** is the value used to adjust the networks internal **weights**, allowing the network to **learn**. The bigger the gradient, the bigger the **adjustments** and vice versa.

- **Problem**: When doing back propagation, each node in a layer calculates it's gradient with respect to the effects of the gradients in the layer before it. So if the adjustments to the layers before it is small, then adjustments to the current layer will be even smaller.

- That causes gradients to exponentially **shrink** as it back propagates down. The earlier layers **fail to do any learning** as the internal weights are barely being adjusted due to **extremely small gradients**. And that's the vanishing gradient problem.

- RNN: think of each **time step** in a recurrent neural network as a **layer** and **back-propagation through time**. The gradient values will exponentially **shrink** as it propagates through each time step. Again, the gradient is used to make **adjustments** in the neural networks **weights** thus allowing it to **learn**. Small gradients mean small adjustments. That causes the early layers not to learn.



Gradients shrink as it back-propagates down

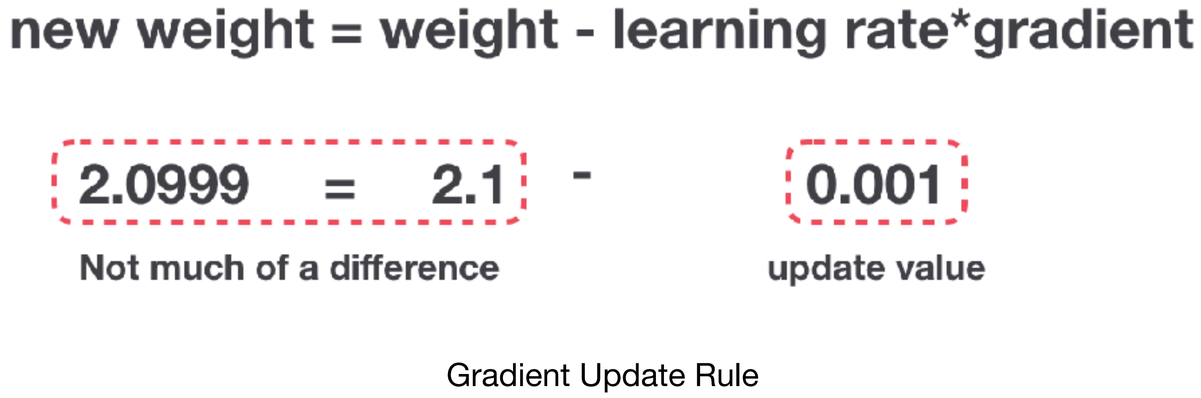
Gradients shrink as it back-propagates through time

RNN: Vanishing Gradient (3)

- Recurrent Neural Networks suffer from **short-term memory**. If a sequence is **long** enough, they'll have a hard time carrying information from **earlier** time steps to later ones. So 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, recurrent neural networks suffer from the **vanishing** gradient problem. Gradients are values used to **update** a neural networks weights. The vanishing gradient problem is when the gradient **shrinks** as it back propagates through time. If a gradient value becomes extremely small, it doesn't contribute too much **learning**.

- So in recurrent neural networks, **layers** that get a **small gradient** update **stops learning**. Those are usually the **earlier** layers. So because these layers don't learn, RNN's can **forget** what it seen in longer sequences, thus having a short-term memory.



Gradient Update Rule

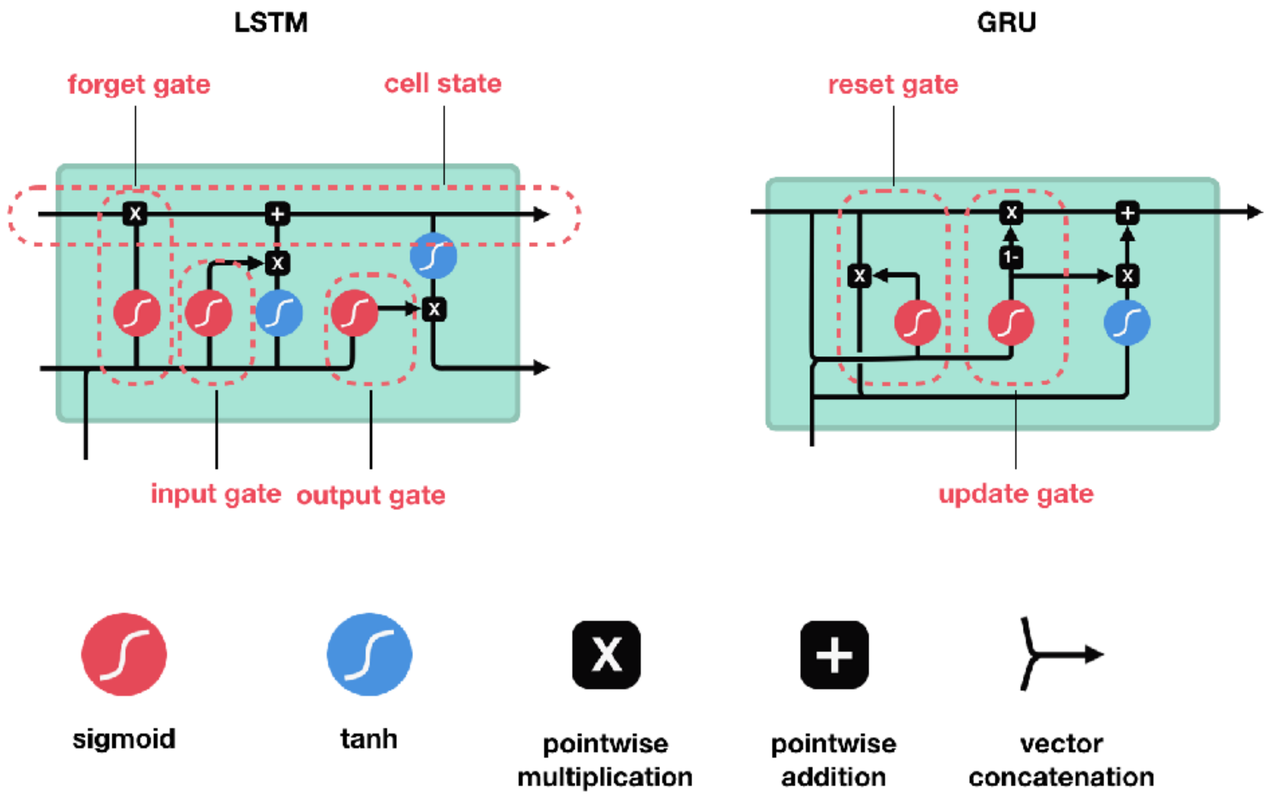
LSTM / GRU: Solution

- LSTMs (Long Short Term Memory) and GRUs (Gated Recurrent Units) were created as the **solution** to the short-term memory problem (they are good at processing long sequences)

- They have internal mechanisms called **gates** that can regulate the flow of information. Also: **Cell state** (hidden state).

- These gates can **learn** which data in a sequence is important to keep or throw away (passes on relevant information).

- **SotA** RNN uses LSTMs and GRUs (speech recognition, speech synthesis, text generation and video captioning)



LSTM / GRU: Intuition

- Looking at **reviews**: Buy «Life cereal» or not?
  - Given review: **good or bad**?
  - The **brain** only remembers important **keywords** (other words will fade away from memory).
  - LSTMs or GRUs: **learn** to **keep** only **relevant** information to make predictions, and **forget** non relevant data.

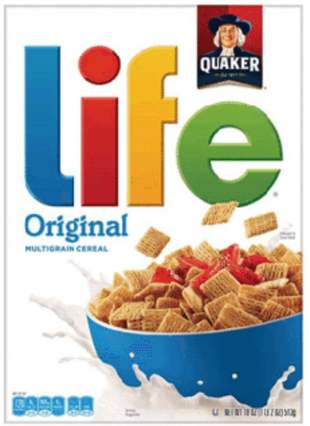
Customers Review 2,491

Thanos

September 2018

Verified Purchase

Amazing! This box of cereal gave me a perfectly balanced breakfast, as all things should be. I only ate half of it but will definitely be buying again!

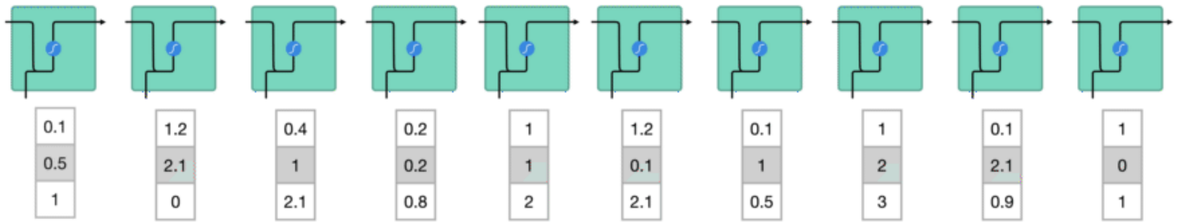


A Box of Cereal

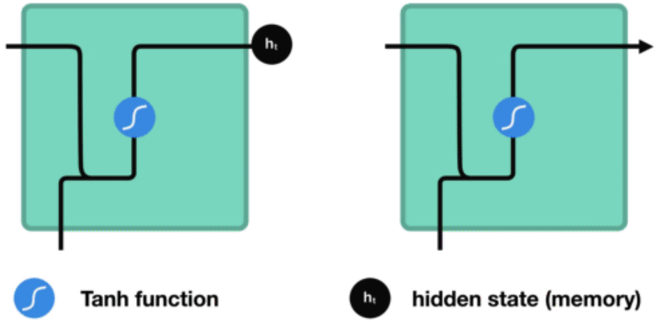
$3.99

LSTM / GRU: RNN review

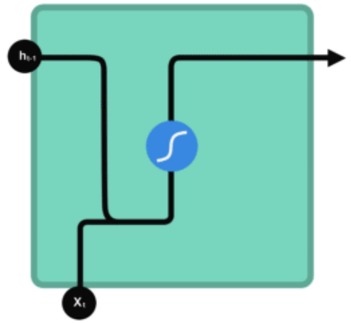
- Review RNNs to **understand** LSTMs / GRUs
- First **words** get transformed into machine-readable **vectors**. Then the RNN processes the **sequence** of vectors one by one.
- While processing, it passes the **previous hidden state** to the **next step** of the sequence. The hidden state acts as the neural networks **memory**. It holds information on **previous** data the network has **seen** before.
- RNN **cell**:
  - First, the input and previous hidden state are **combined** to form a vector. That vector now has information on the current input and previous inputs.
  - The vector goes through the **tanh activation**, and the **output** is the **new hidden state**, or the memory of the network.

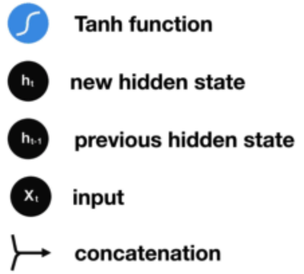


Processing sequence one by one



Passing hidden state to next time step

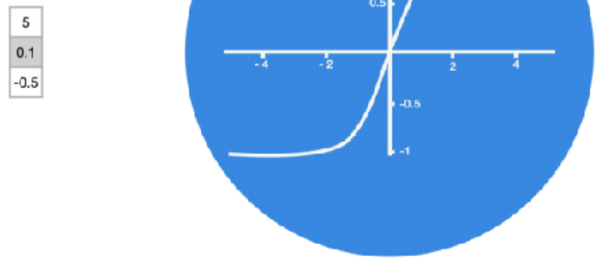


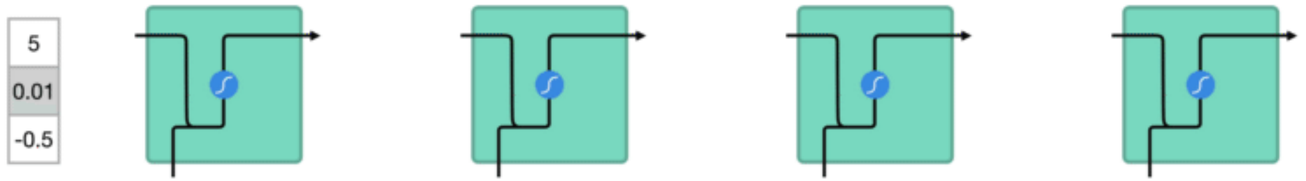


RNN Cell

LSTM / GRU: RNN review (2)

- The tanh activation is used to help regulate the values flowing through the network. The tanh function **squishes** values to always be between -1 and 1.
- When vectors are flowing through a neural network, it undergoes many transformations due to various math operations.
  - So imagine a value that continues to be multiplied by let's say 3. You can see how some values can **explode** and become astronomical, causing other values to seem **insignificant**.
  - A tanh function ensures that the values stay between -1 and 1, thus **regulating** the output of the neural network.
- An RNN has very **few operations** internally but works pretty **well for short** sequences and uses a lot **less computational resources** than it's evolved variants, LSTMs and GRUs.





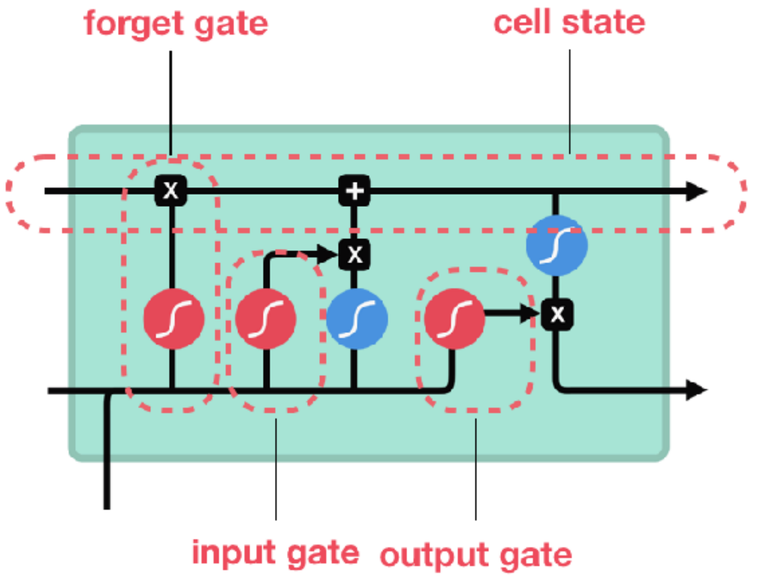
LSTM

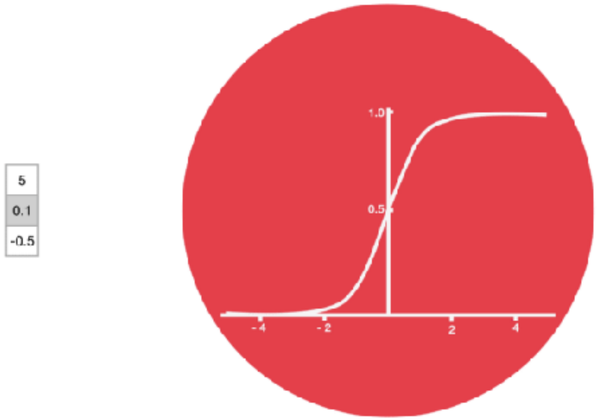
- The core concept of LSTM **cells** are the **cell state**, and it's various **gates**.

- The cell state act as a transport **highway** that transfers relevant information all the way down the sequence chain. You can think of it as the "**memory**" of the network. The cell state can carry relevant information throughout the processing of the sequence. So even information from the **earlier** time steps can make it's way to later time steps, reducing the effects of short-term memory.

- As the cell state goes on its journey, information get's **removed or added** to the cell state via gates. The gates are different NNs that decide which information is allowed on the cell state. The gates can learn what information is relevant to forget or keep during training.

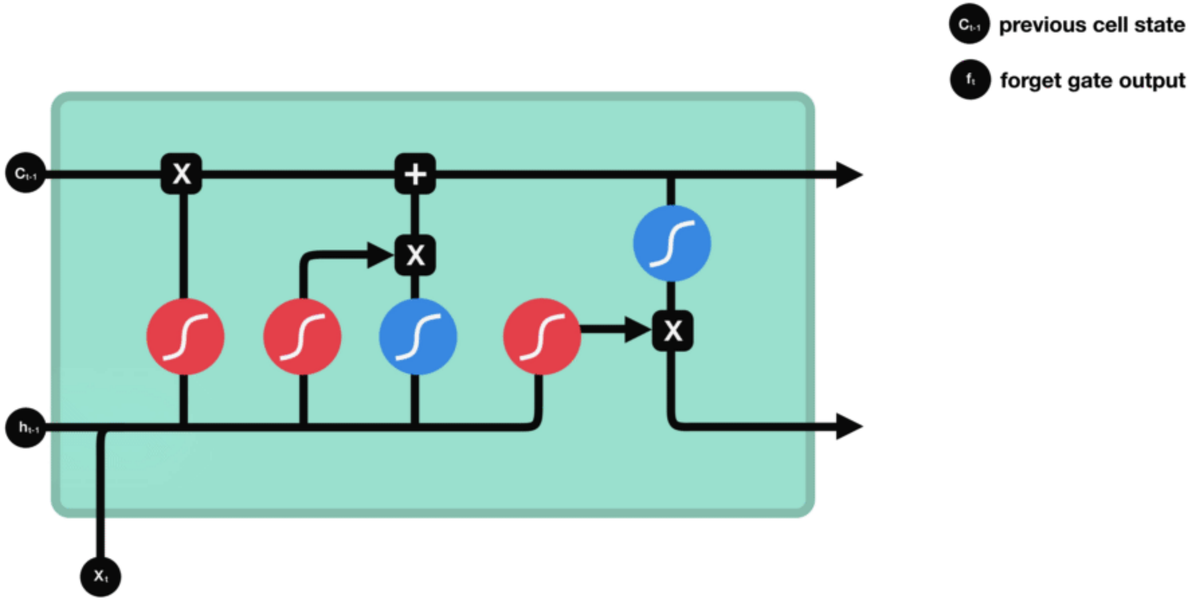
- Gates contains **sigmoid** activations that **squishes** values between 0 and 1. **Helpful** to **forget** or **update** data because any number getting multiplied by by **0** is 0, causing values to disappear or be "**forgotten.**" Likewise, any number getting multiplied by **1** is 1, therefore that value stay's the same or is "**kept.**"





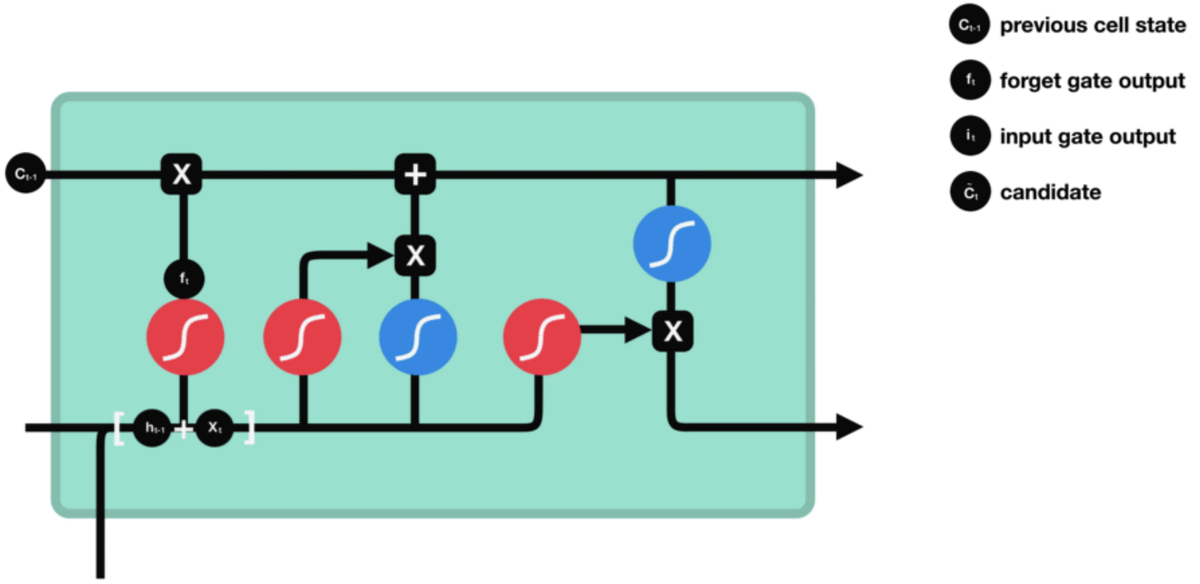
LSTM: Forget gate

- This gate decides what information should be **thrown** away or **kept**.
- Values come out between 0 and 1. The closer to 0 means to **forget**, and the closer to 1 means to **keep**.



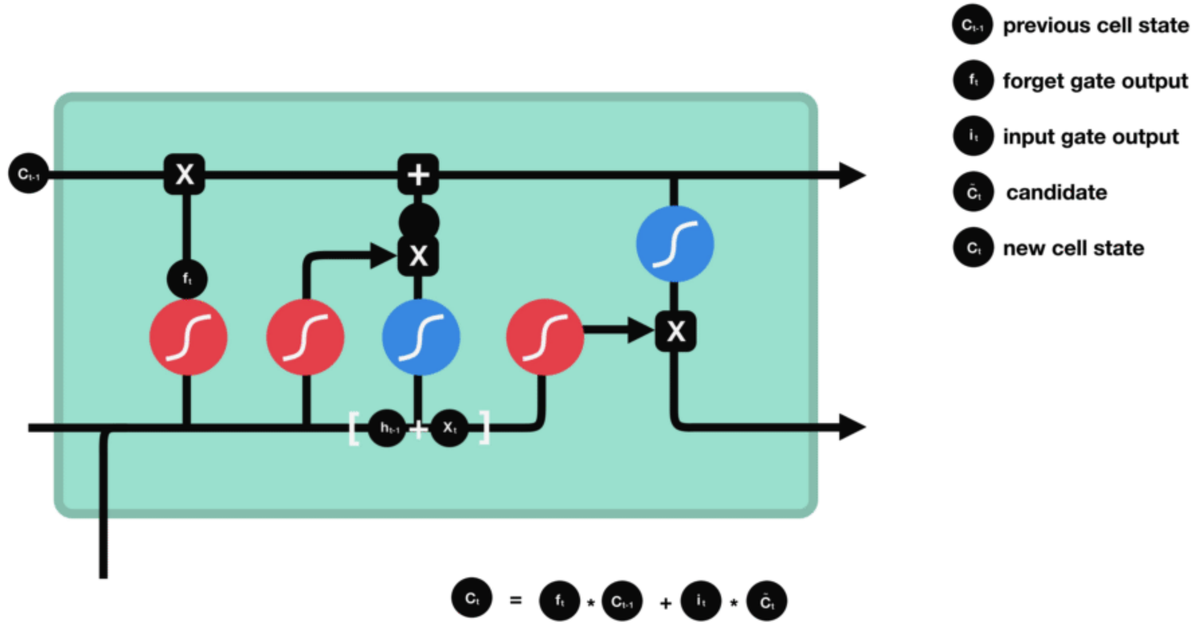
LSTM: Input Gate

- To **update** the cell state, we have the **input** gate.
- 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.



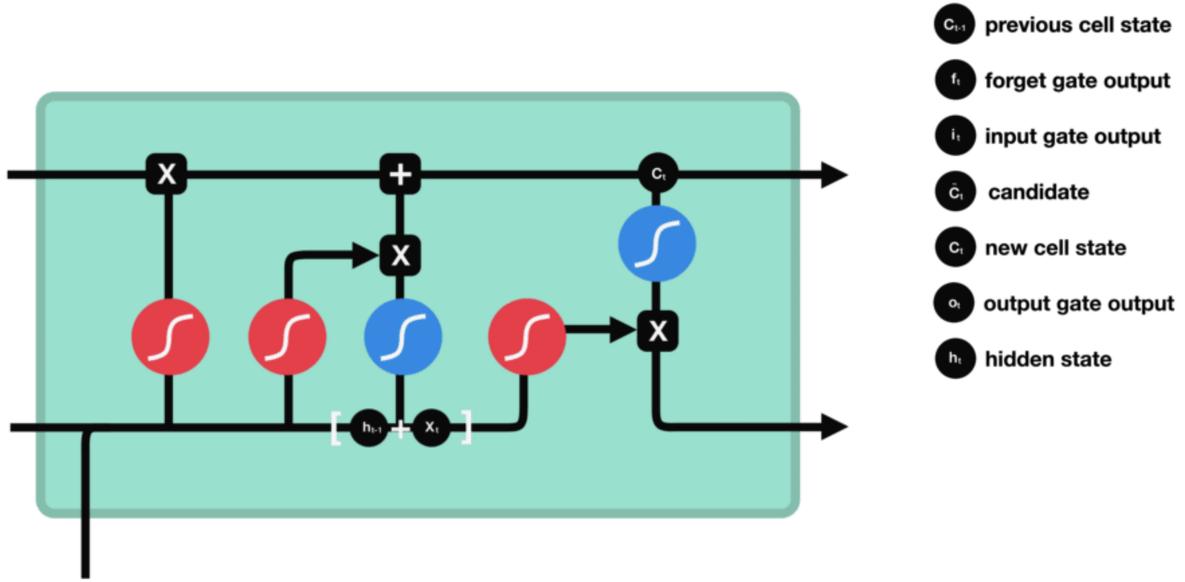
LSTM: Cell State

- Now we should have enough information to calculate the cell state.
- First, the cell state gets pointwise **multiplied** by the forget vector.
- This has a possibility of dropping values in the cell state if it gets multiplied by values near 0.
- Then we take the output from the input gate and do a pointwise **addition** which updates the cell state to new values that the neural network finds relevant.
- That gives us our new cell state.



LSTM: Output Gate

- The output gate decides what the next **hidden** state should be.
- Remember that the hidden state contains information on previous inputs. The hidden state is also used for **predictions**.
- First, we pass the previous hidden state and the current input into a **sigmoid** function.
- Then we pass the newly modified **cell state** to the tanh function.
- We multiply the tanh output with the sigmoid output to decide what information the hidden state should carry.
- The output is the hidden state. The new cell state and the new hidden is then carried over to the **next time** step.
- To review, the Forget gate decides what is relevant to **keep** from prior steps. The input gate decides what information is relevant to **add** from the current step. The output gate determines what the next **hidden** state should be.



LSTM: Code

- The control flow of an LSTM network are a few tensor operations and a for loop. You can use the hidden states for predictions. Combining all those mechanisms, an LSTM can choose which information is relevant to remember or forget during sequence processing.

  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.

  3. A candidate layer is created using combine. The candidate holds possible values to add to the cell state.

  4. 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.

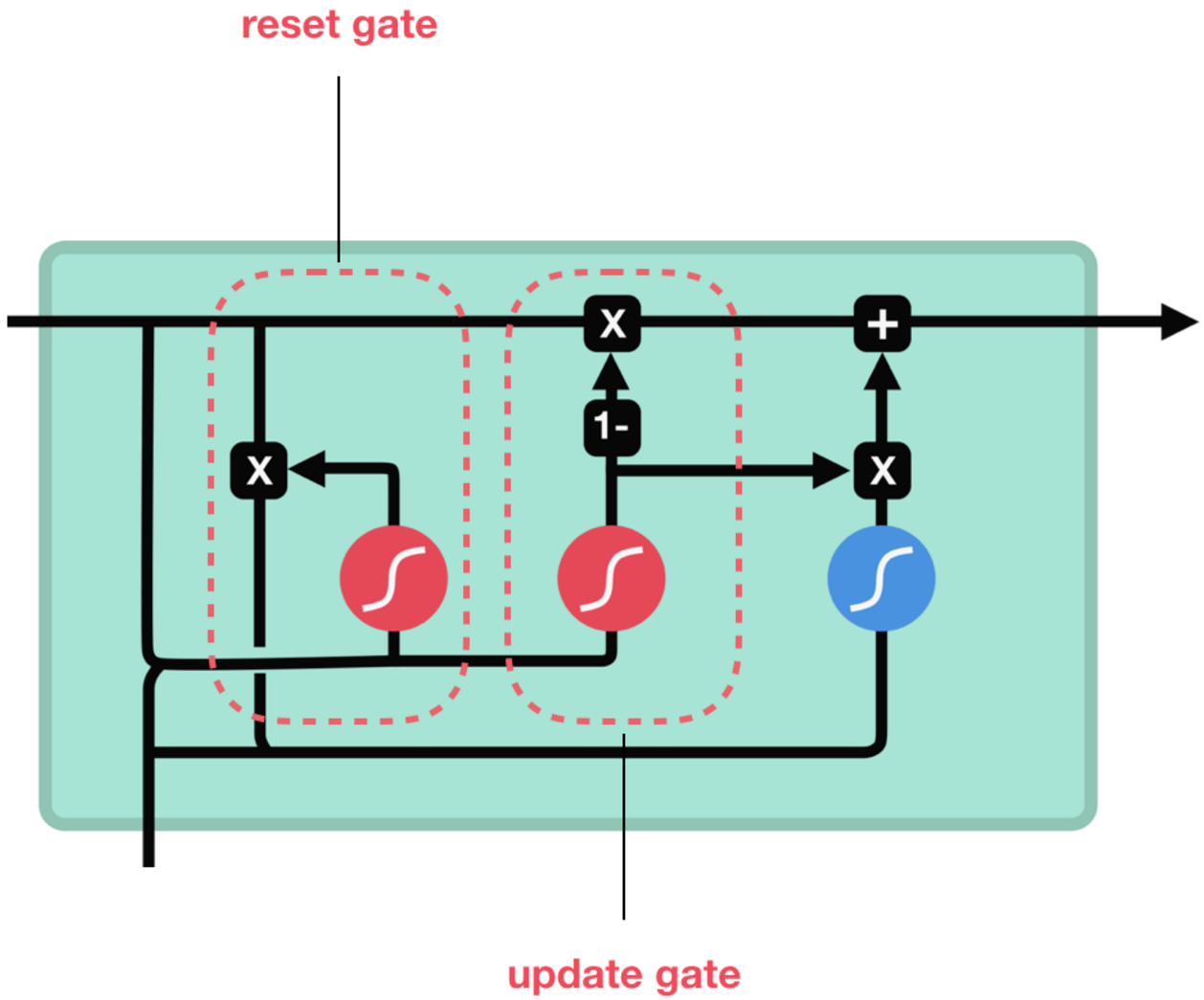
  6. The output is then computed.

  7. Pointwise multiplying the output and the new cell state gives us the new hidden state.

```
def LSTMCELL(prev_ct, prev_ht, input):
    combine = prev_ht + input
    ft = forget_layer(combine)
    candidate = candidate_layer(combine)
    it = input_layer(combine)
    Ct = prev_ct * ft + candidate * it
    ot = output_layer(combine)
    ht = ot * tanh(Ct)
    return ht, Ct



ct = [0, 0, 0]
ht = [0, 0, 0]

for input in inputs:
    ct, ht = LSTMCELL(ct, ht, input)
```

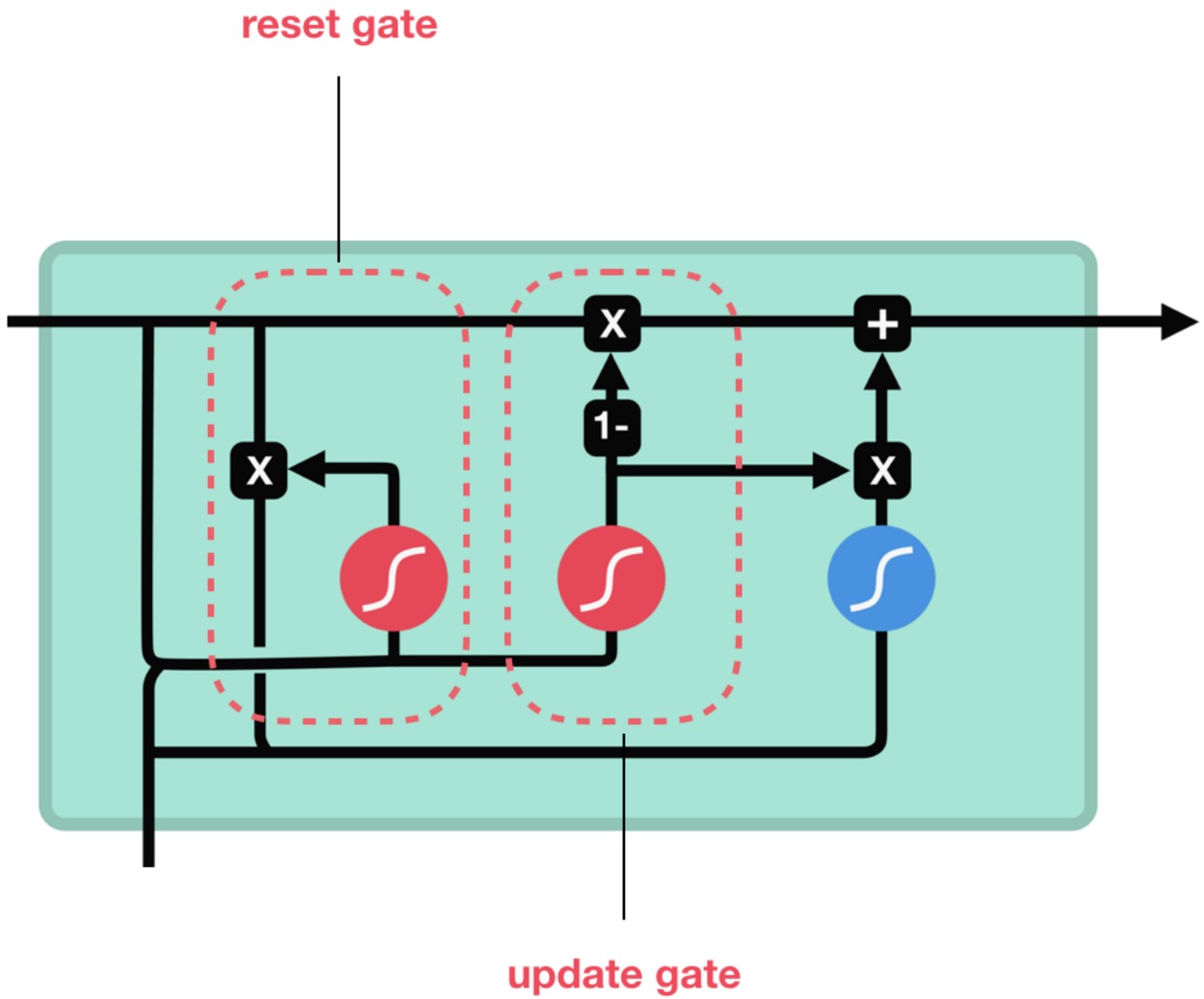
GRU (Gated Recurrent Units)

- The GRU is the newer generation of Recurrent Neural networks and is pretty similar to an LSTM.
- GRU's got rid of the cell state and used the **hidden** state to transfer information.
- Only has two gates, a **reset** gate and **update** gate.



GRU (2)

- Reset Gate:
  - The reset gate is used to decide how much past information to **forget**.
- 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**.
- GRU's has fewer tensor operations; therefore, they are a little speedier to train then LSTM's. There isn't a clear winner which one is better. Researchers and engineers usually try both to determine which one works better for their use case.

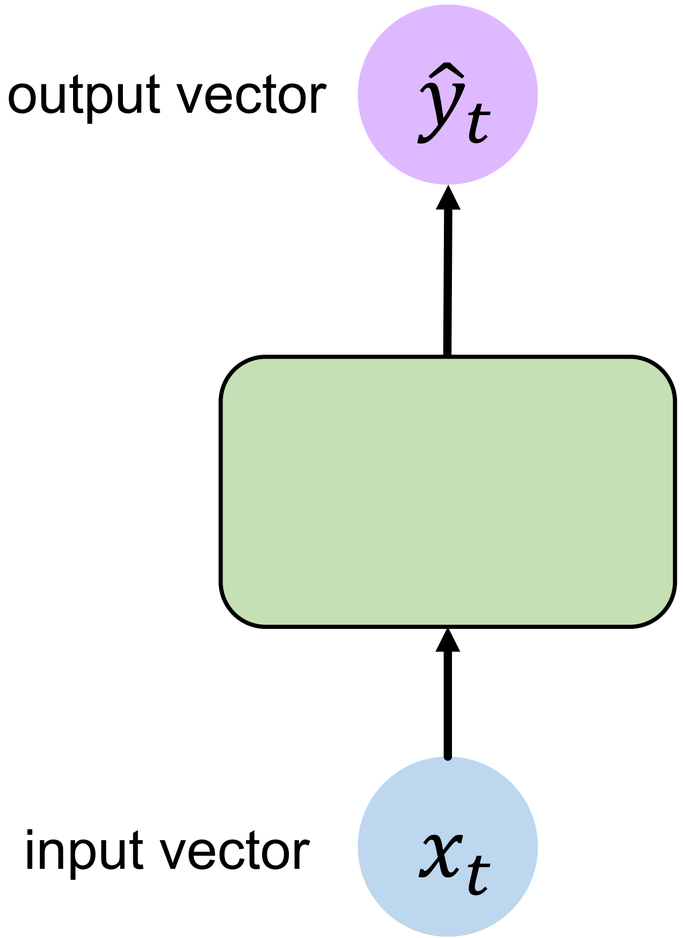


MIT - 6S19
Sequence Models

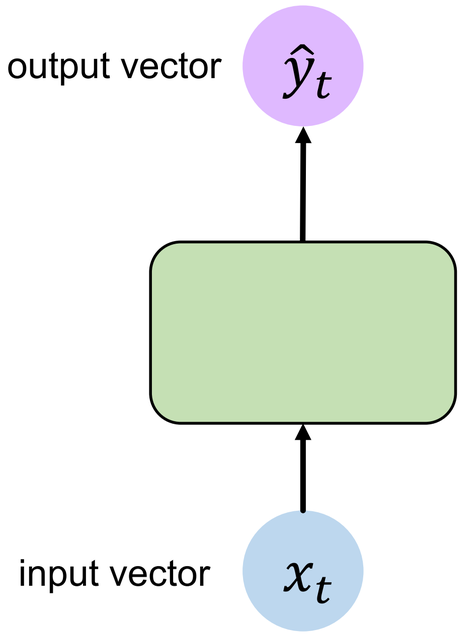
Some more details
Summary

Motivation:
Predict the Next Word

using a FF-NN



A sequence modeling problem: predict the next word



"This morning I took my cat for a walk."

given these words

predict the next word

A standard «vanilla» FF-NN

Idea #1: use a fixed window

"This morning I took my cat for a walk."

given these two words

predict the next word

One-hot feature encoding: tells us what each word is

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

for

a

prediction

Problem #1: can't model long-term dependencies

"**France** is where I grew up, but I now live in Boston. I speak fluent ___."

We need information from **the distant past** to accurately predict the correct word.

Idea #2: use entire sequence as set of counts

"This morning I took my cat for a"

"bag of words"

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

prediction

Problem #2: counts don't preserve order

The food was good, not bad at all.

vs.

The food was bad, not good at all.

Idea #3: use a really big fixed window

"This morning I took my cat for a walk."

given these words

predict the next word

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

morning I took this cat

prediction

Problem #3: no parameter sharing

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

this morning took the cat

Each of these inputs has a **separate parameter:**

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

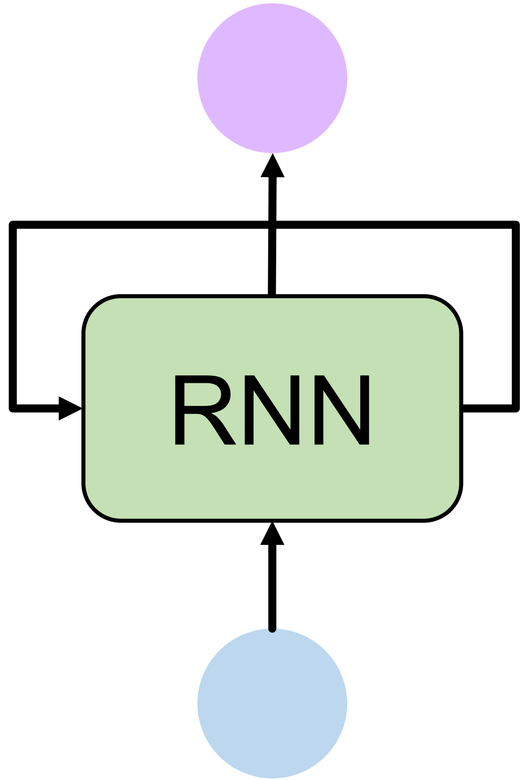
this morning

Things we learn about the sequence **won't transfer** if they appear **elsewhere** in the sequence.

Sequence modeling: design criteria

To model sequences, we need to:

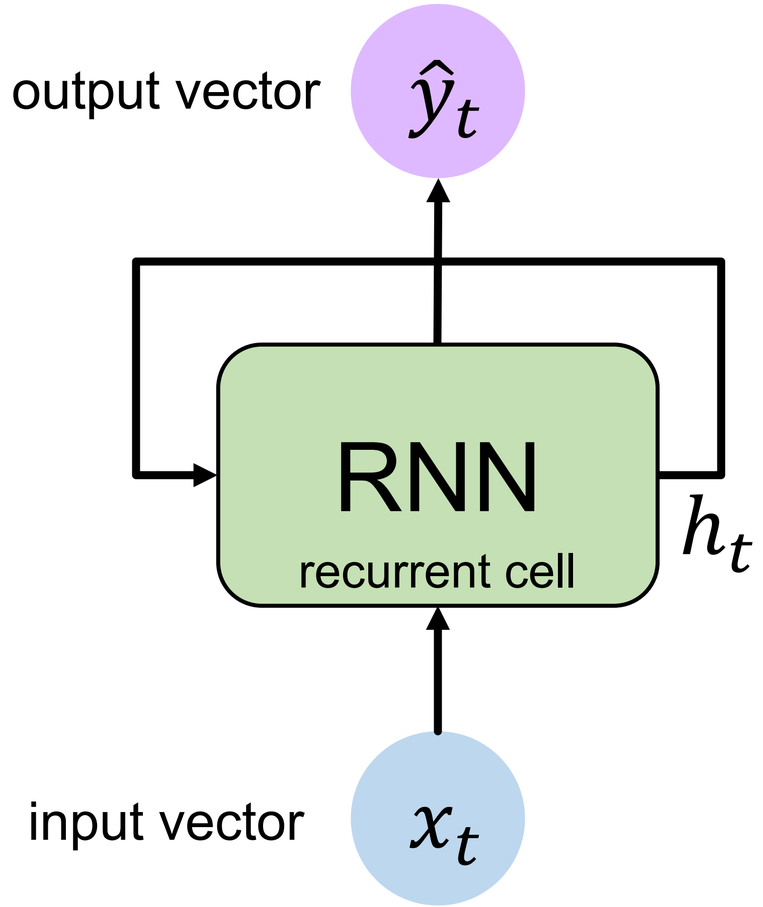
1. Handle **variable-length** sequences
2. Track **long-term** dependencies
3. Maintain information about **order**
4. **Share parameters** across the sequence



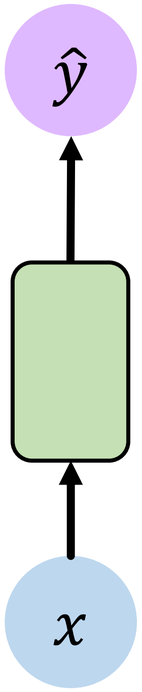
Today: **Recurrent Neural Networks (RNNs)** as an approach to sequence modeling problems

Recurrent Neural Networks

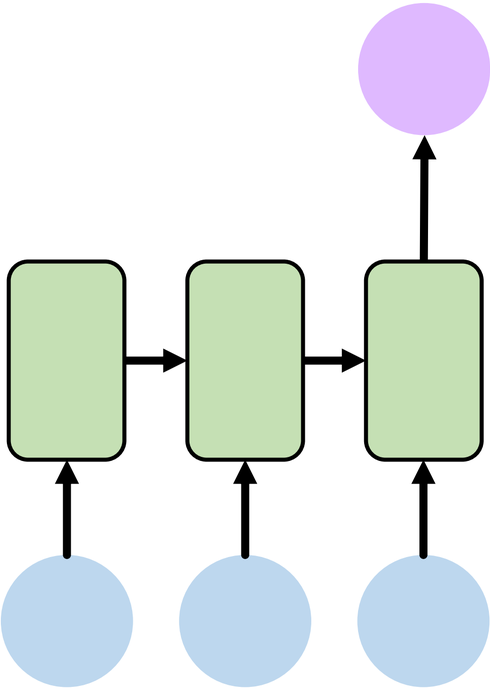
(RNNs)



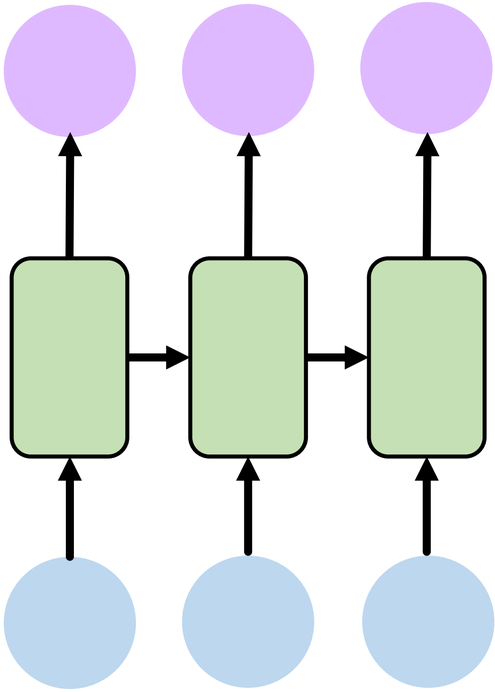
Recurrent neural networks: sequence modeling



One to One
"Vanilla" neural network

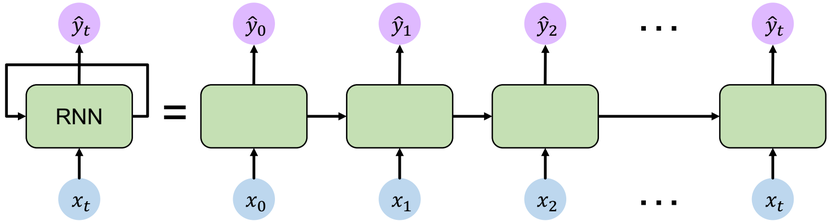


Many to One
Sentiment Classification

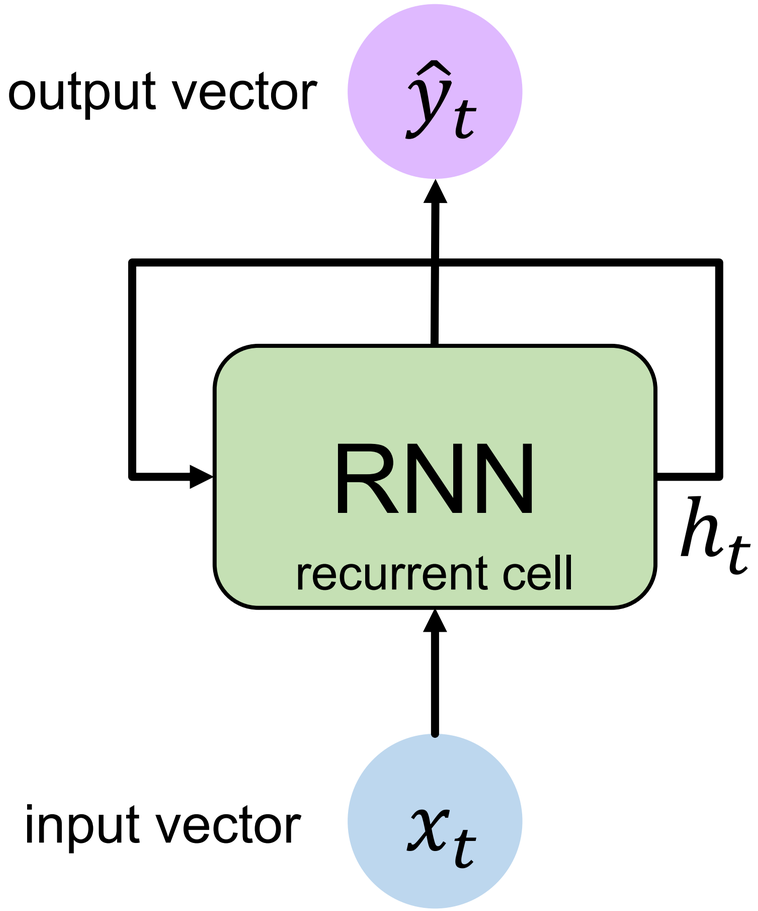


Many to Many
Music Generation

… and many other architectures and applications



A recurrent neural network (RNN)



Apply a **recurrence relation** at every time step to process a sequence:

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

new state

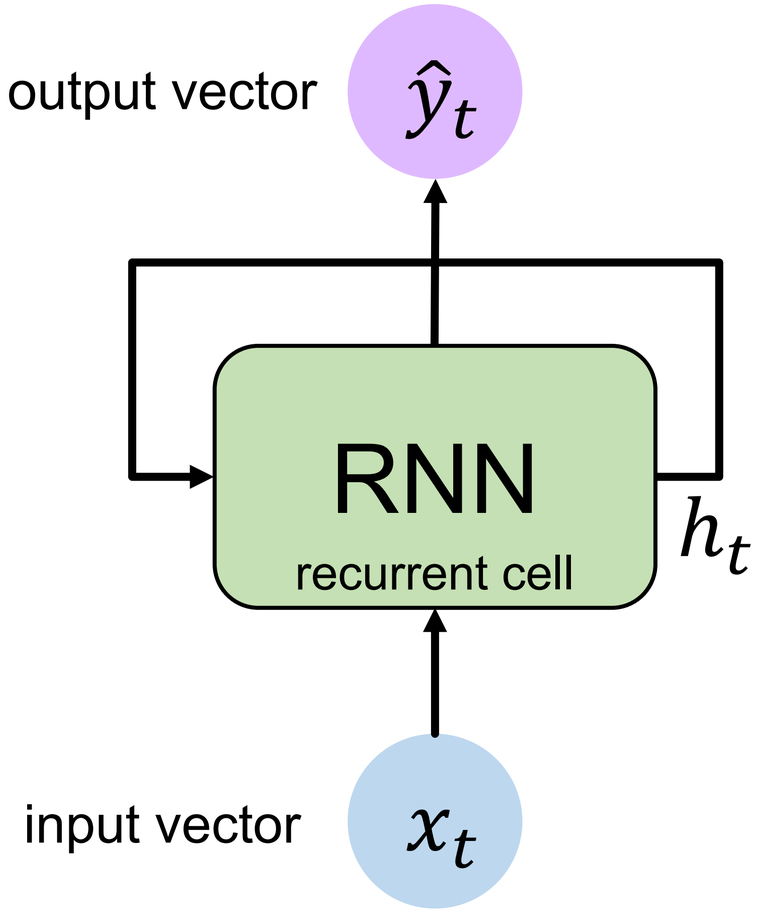
function parameterized by W

old state

input vector at time step *t*

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

RNN state update and output



Output Vector

$$\hat{y}_t = \boldsymbol{W_{hy}} h_t$$

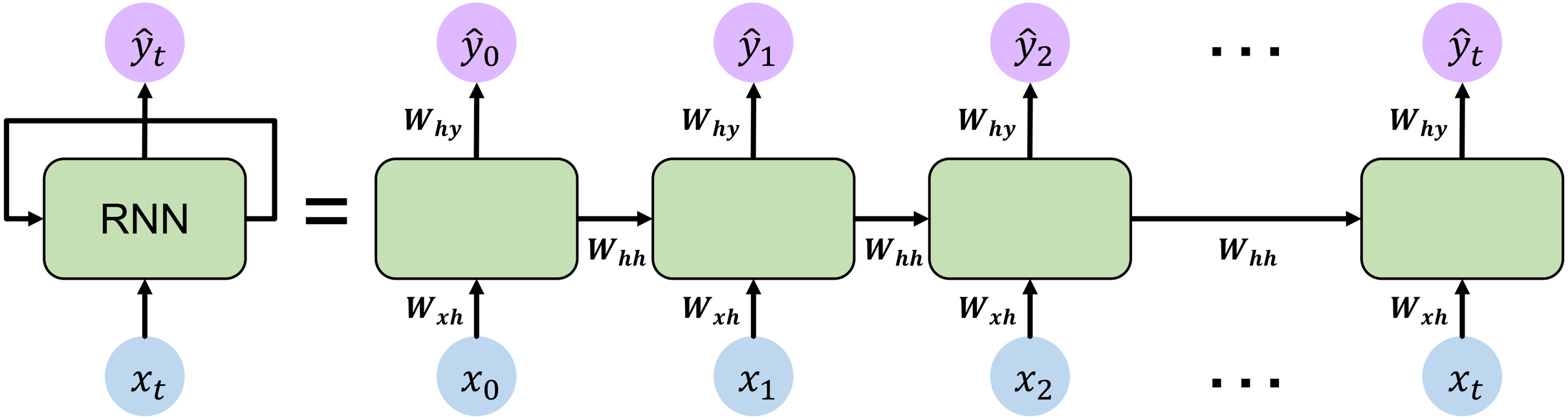
Update Hidden State

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

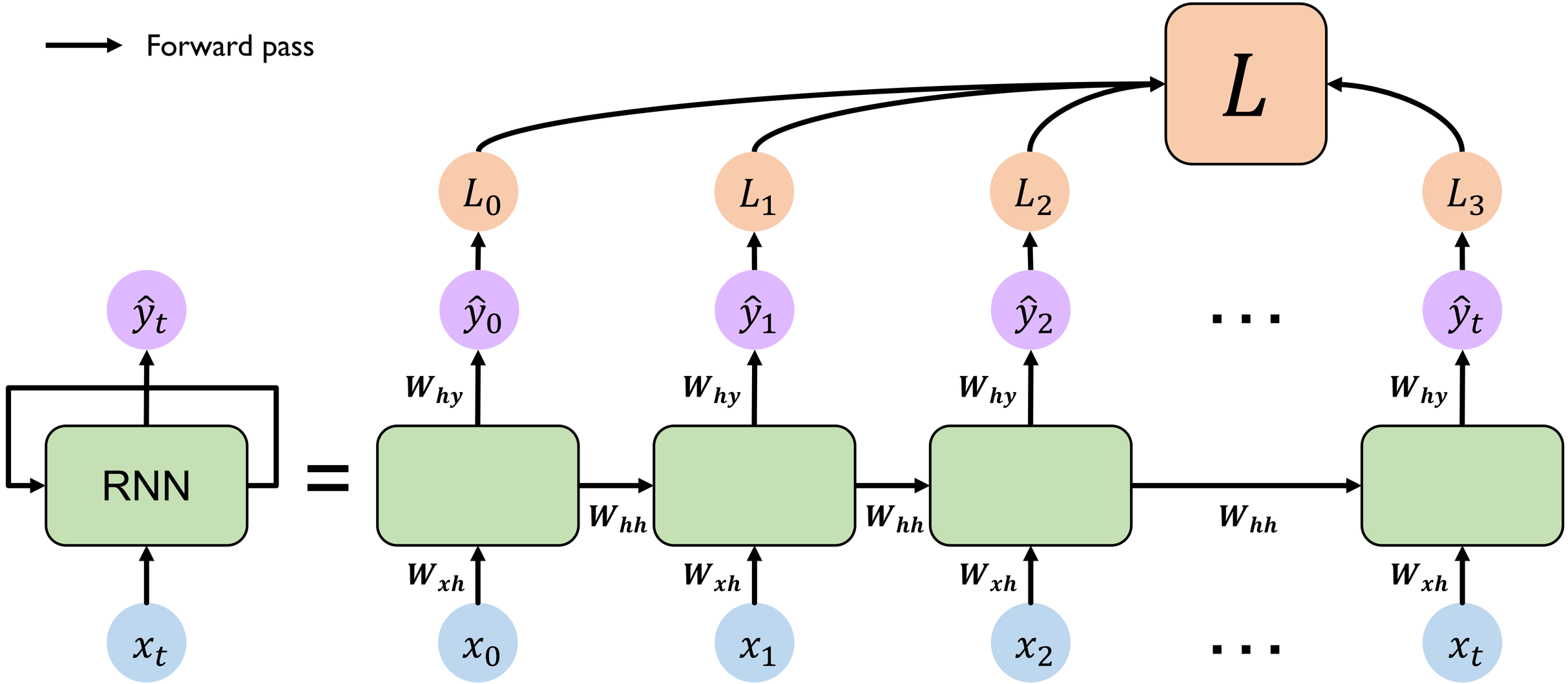
Input Vector

RNNs: computational graph across time

Re-use the **same weight matrices** at every time step

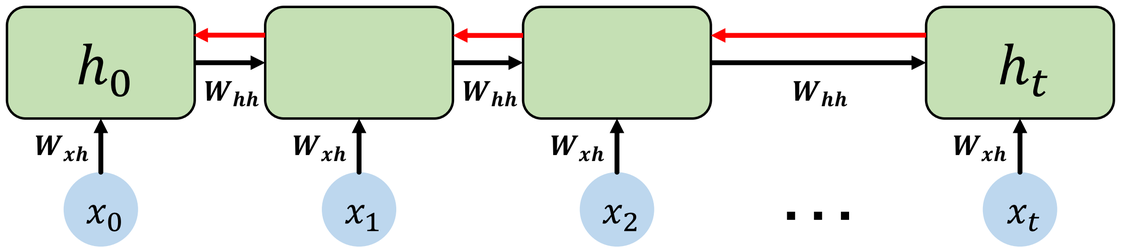


RNNs: computational graph across time

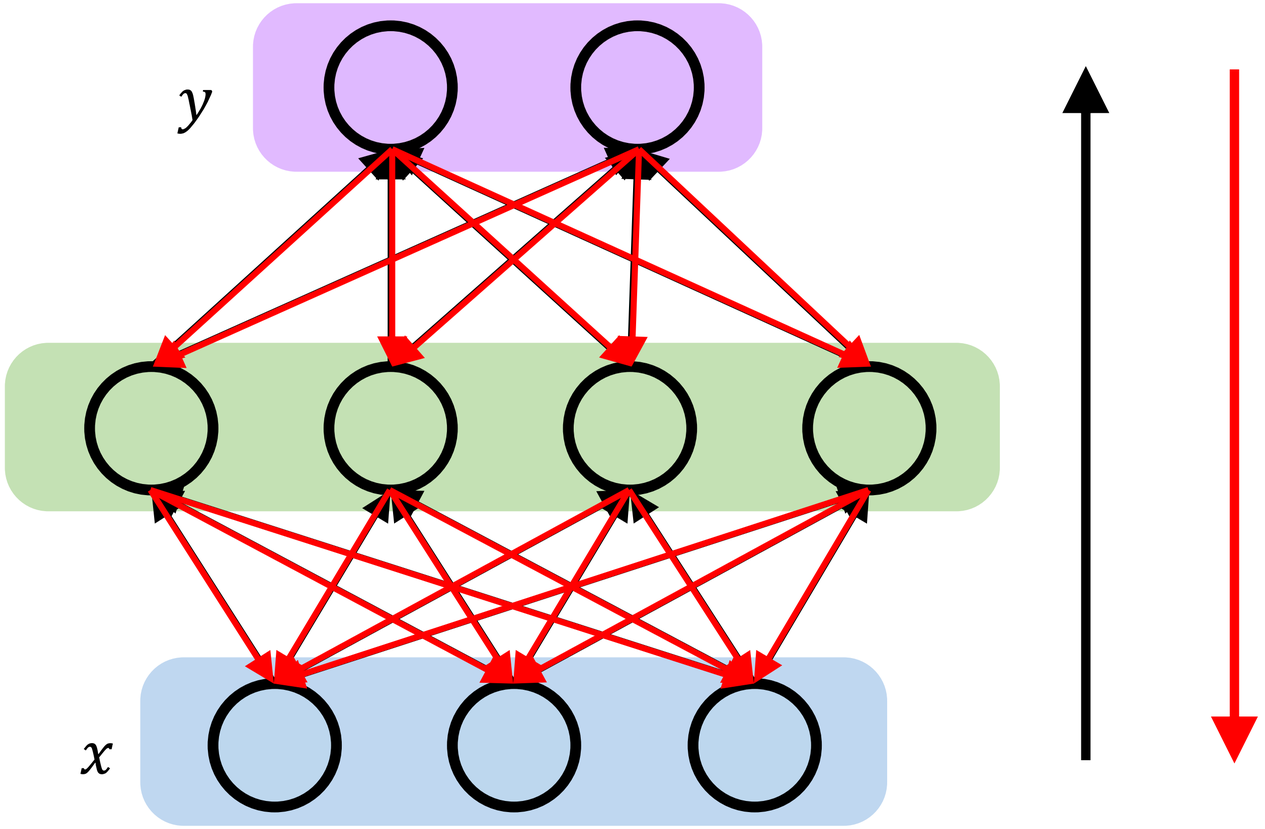


Backpropagation Through Time

(BPTT)



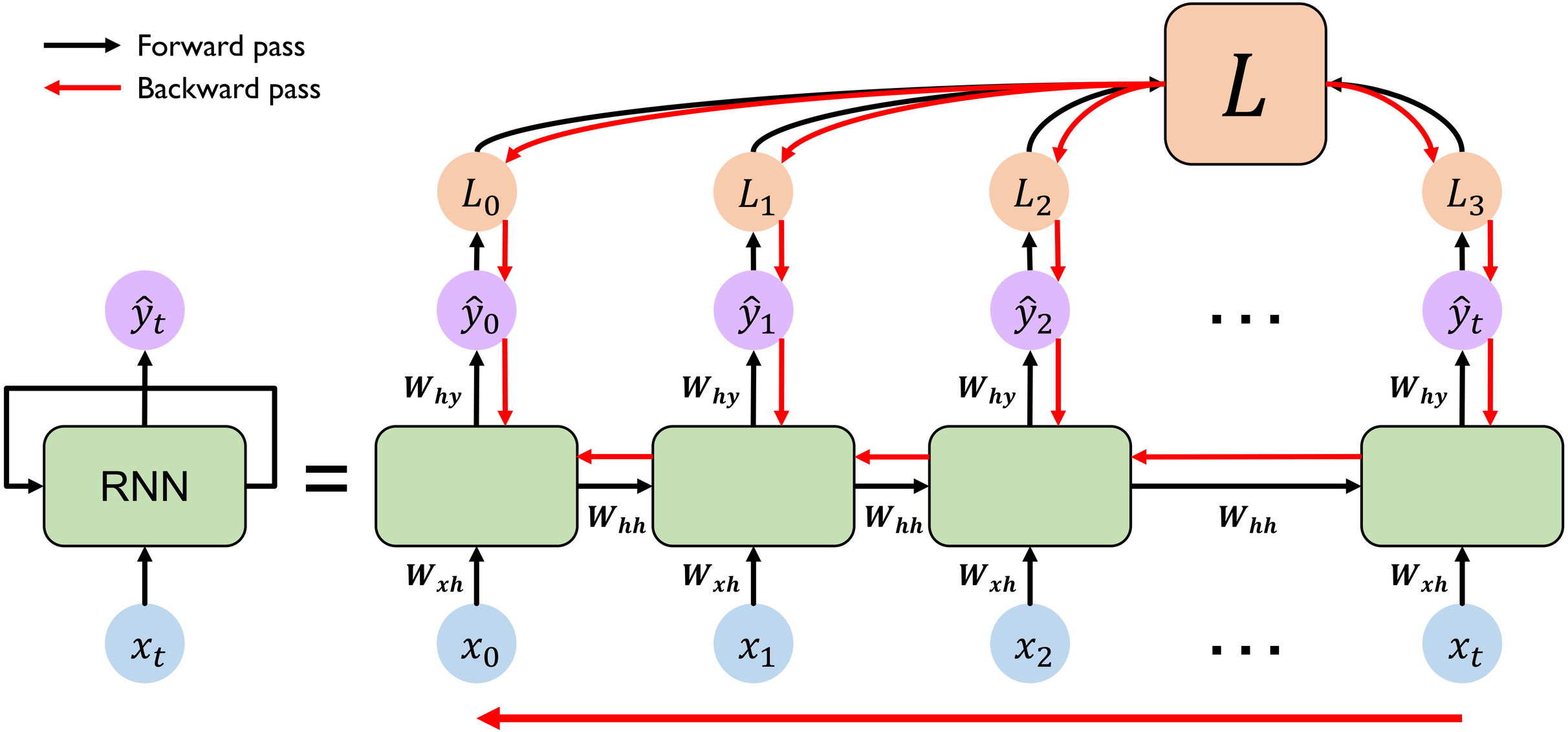
Recall: backpropagation in feed forward models



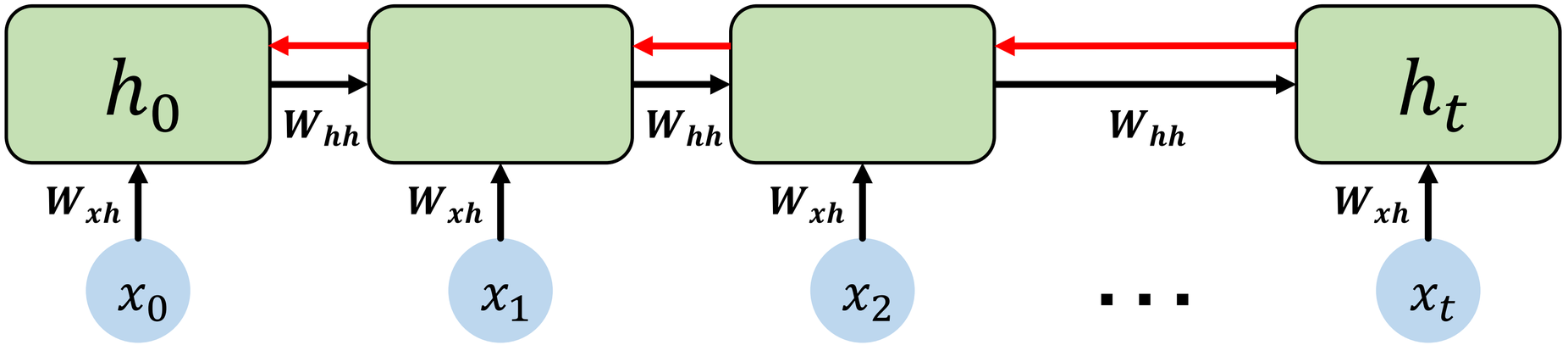
Backpropagation algorithm:

1. Take the derivative (gradient) of the loss with respect to each parameter
2. Shift parameters in order to minimize loss

RNNs: backpropagation through time



Standard RNN gradient flow: exploding gradients

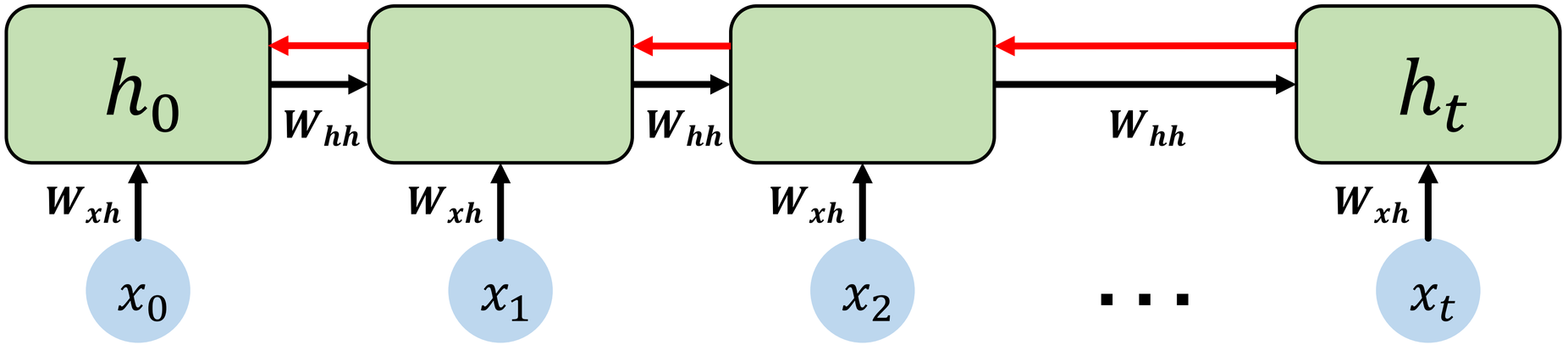


Computing the gradient wrt h_0 involves **many factors of W_{hh}** (and repeated f'!)

Many values > 1:
exploding gradients

Gradient clipping to
scale big gradients

Standard RNN gradient flow: vanishing gradients



Computing the gradient wrt h_0 involves **many factors of W_{hh}** (and repeated f'!)

Largest singular value > 1:
exploding gradients

Gradient clipping to
scale big gradients

Largest singular value < 1:
vanishing gradients

1. Activation function
2. Weight initialization
3. Network architecture

The problem of long-term dependencies

Why are vanishing gradients a problem?

Multiply many **small numbers** together

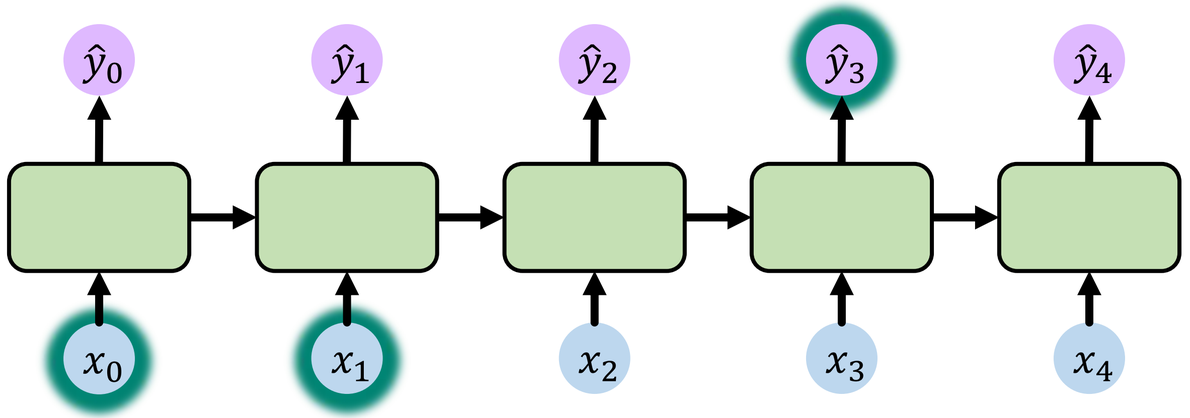
↓

Errors due to further back time steps have smaller and smaller gradients

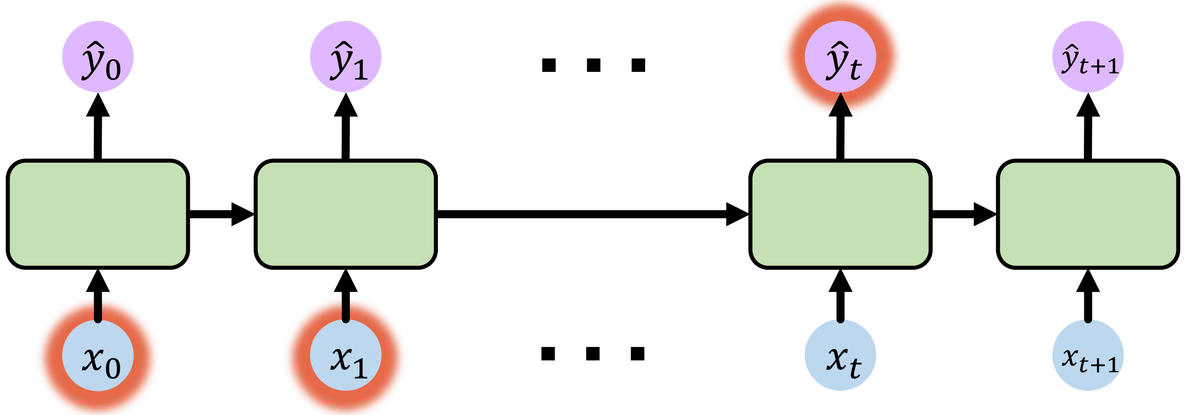
↓

Bias parameters to capture short-term dependencies

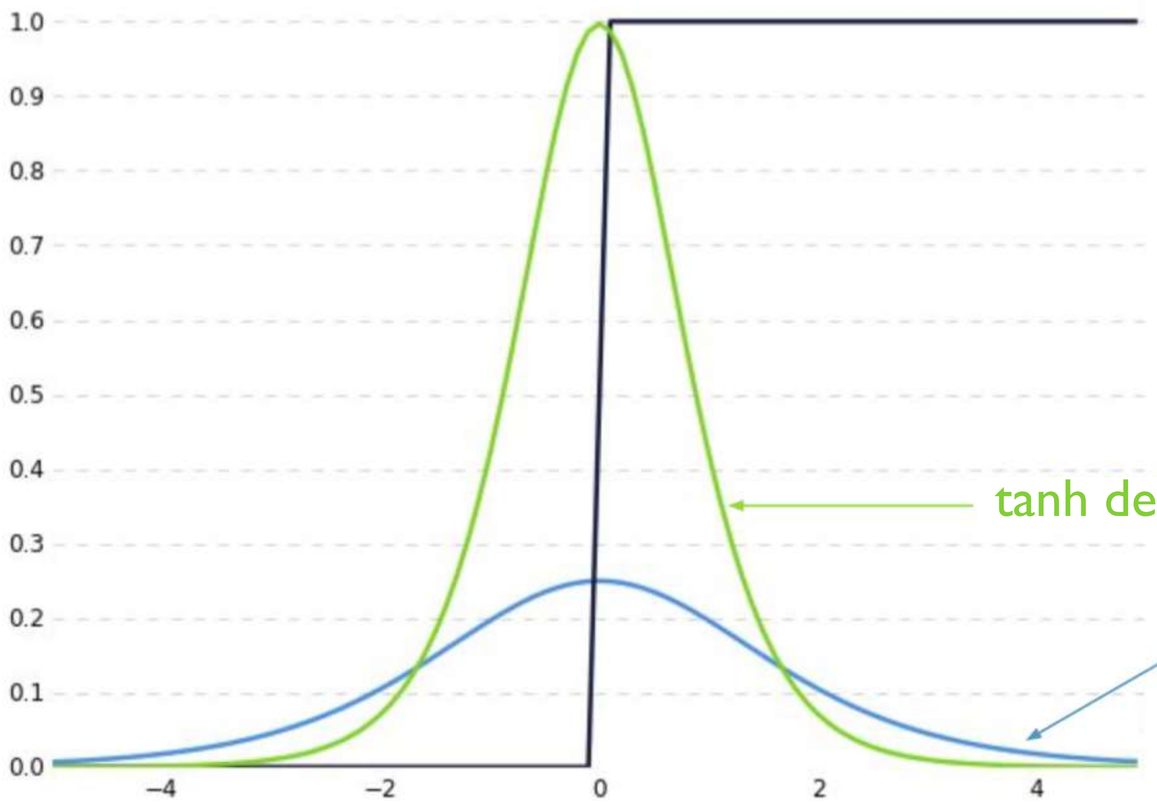
"The clouds are in the ___"



"I grew up in France, … and I I speak fluent___ "



Trick #1: activation functions



ReLU derivative

tanh derivative

sigmoid derivative

Using ReLU prevents f' from shrinking the gradients when $x > 0$

Trick #2: parameter initialization

Initialize **weights** to identity matrix

Initialize **biases** to zero

$$I_n = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

This helps prevent the weights from shrinking to zero.

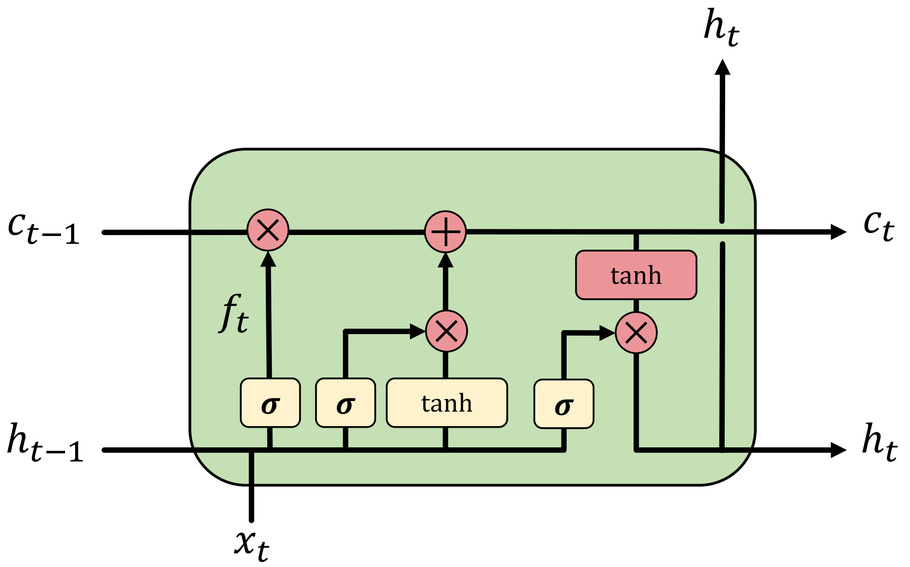
Solution #3: gated cells

Idea: use a more **complex recurrent unit with gates** to control what information is passed through

Long Short Term Memory (LSTMs) networks rely on a gated cell to track information throughout many time steps.

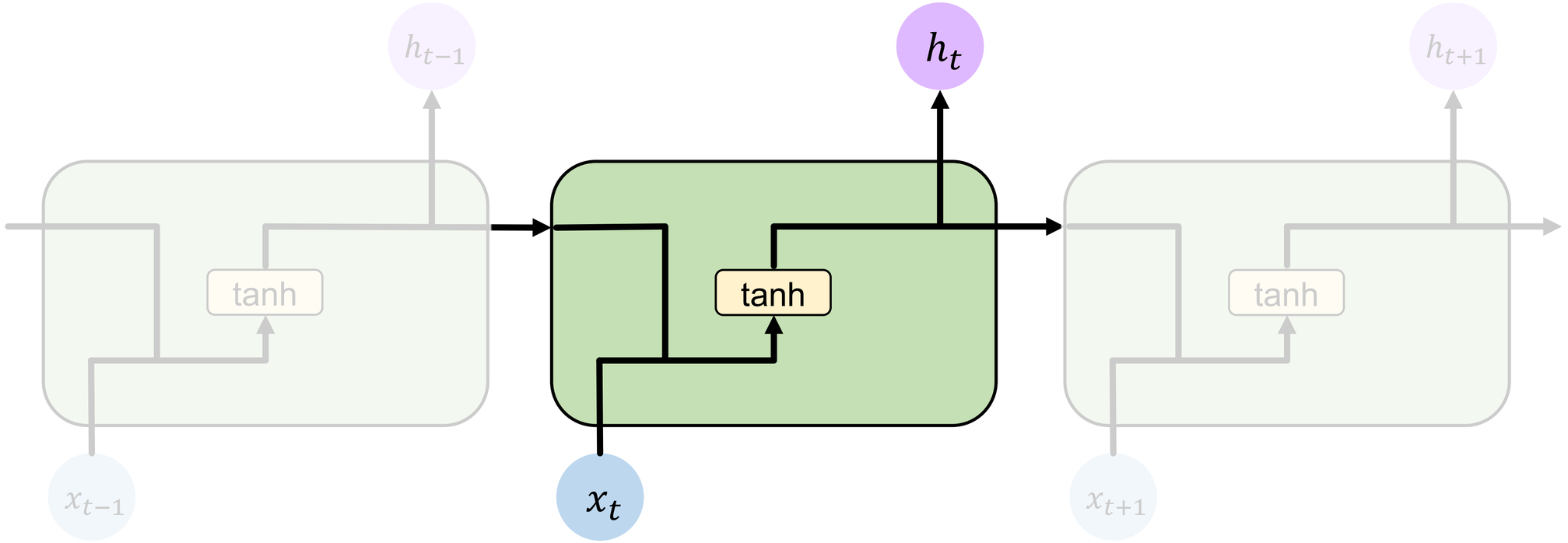
Long Short Term Memory

(LSTM)



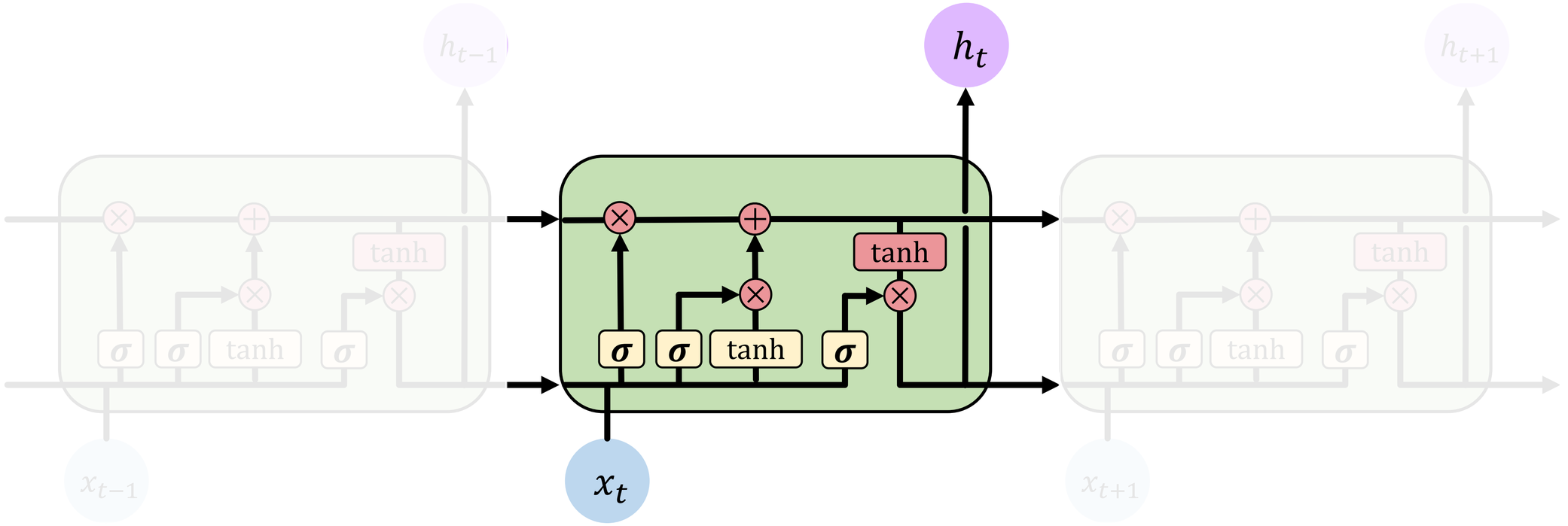
Standard RNN

In a standard RNN, repeating modules contain a **simple computation node**



Long Short Term Memory (LSTMs)

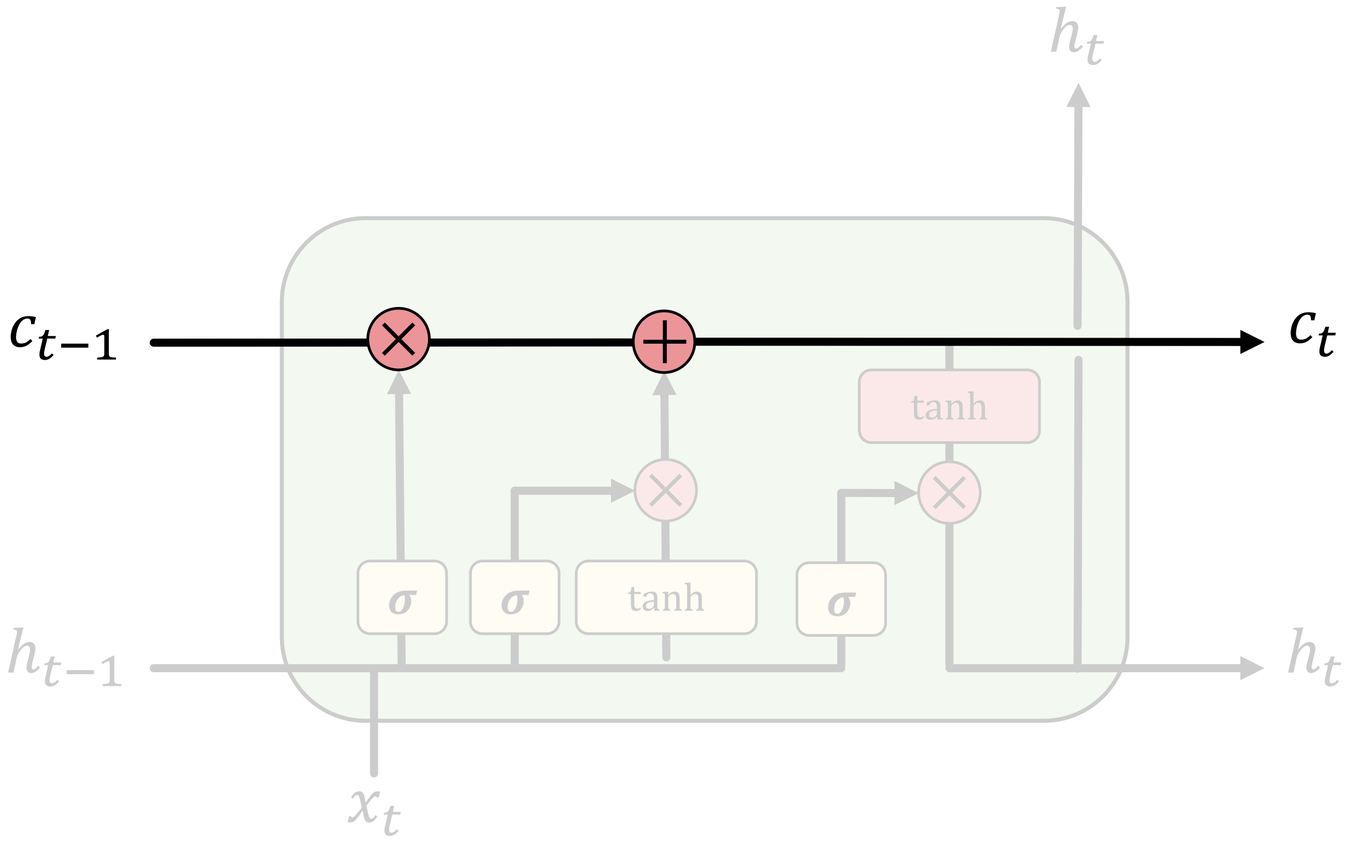
LSTM repeating modules contain **interacting layers** that **control information flow**



LSTM cells are able to track information throughout many timesteps

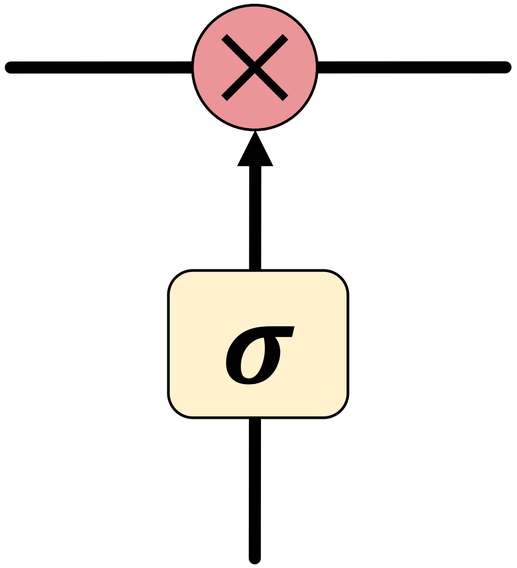
Long Short Term Memory (LSTMs)

LSTMs maintain a **cell state c_t** where it's easy for information to flow



Long Short Term Memory (LSTMs)

Information is **added** or **removed** to cell state through structures called **gates**

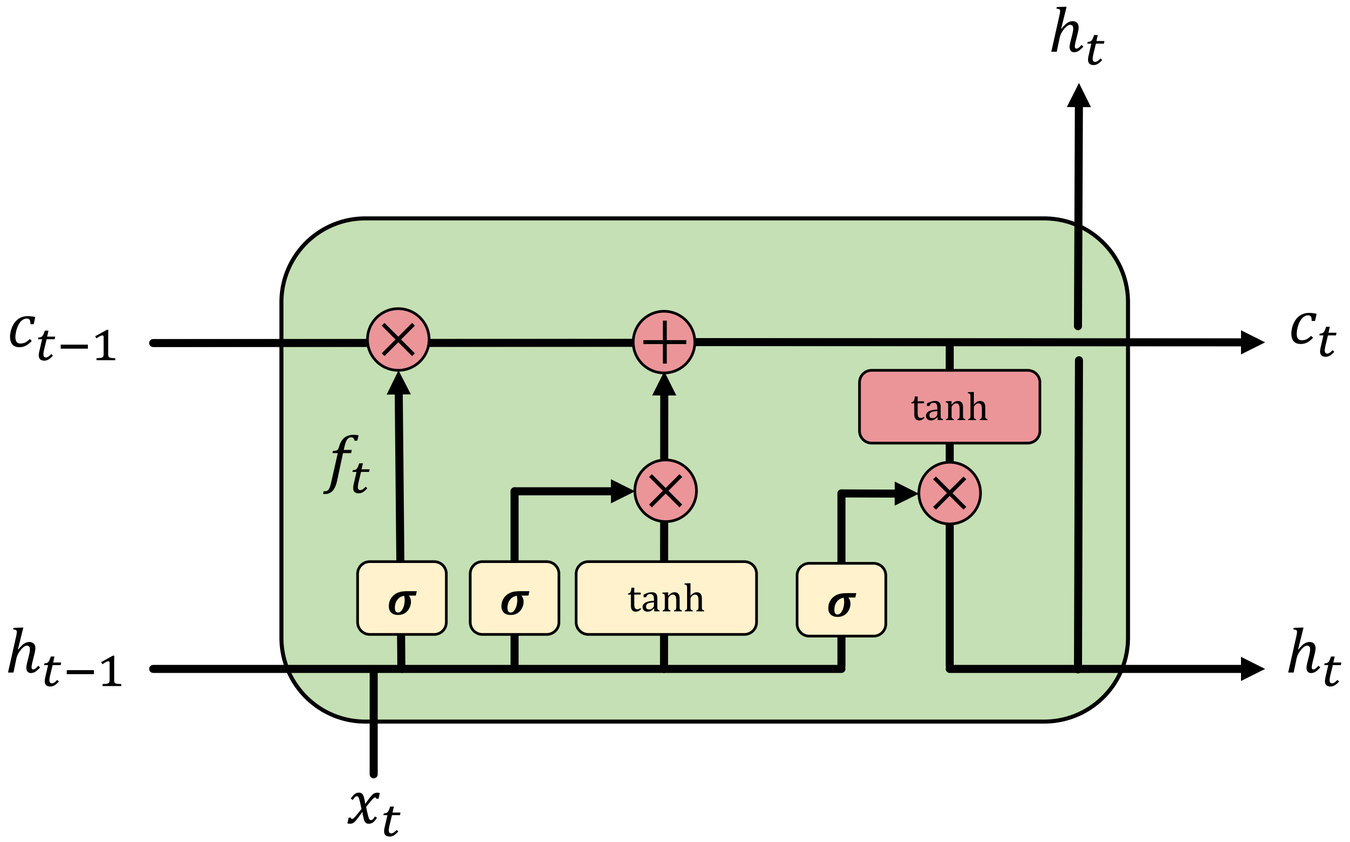


Gates optionally let information through, via a sigmoid neural net layer and pointwise multiplication

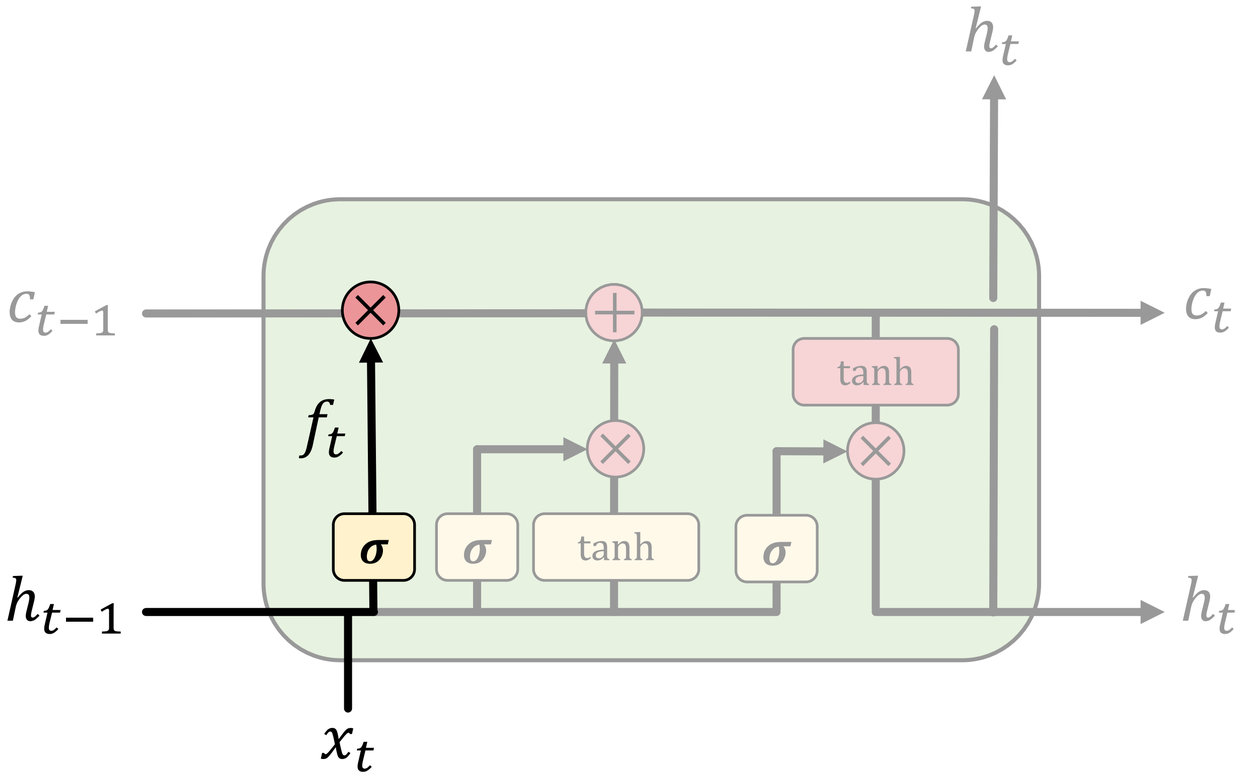
Long Short Term Memory (LSTMs)

How do LSTMs work?

1) Forget 2) Update 3) Output



LSTMs: forget irrelevant information

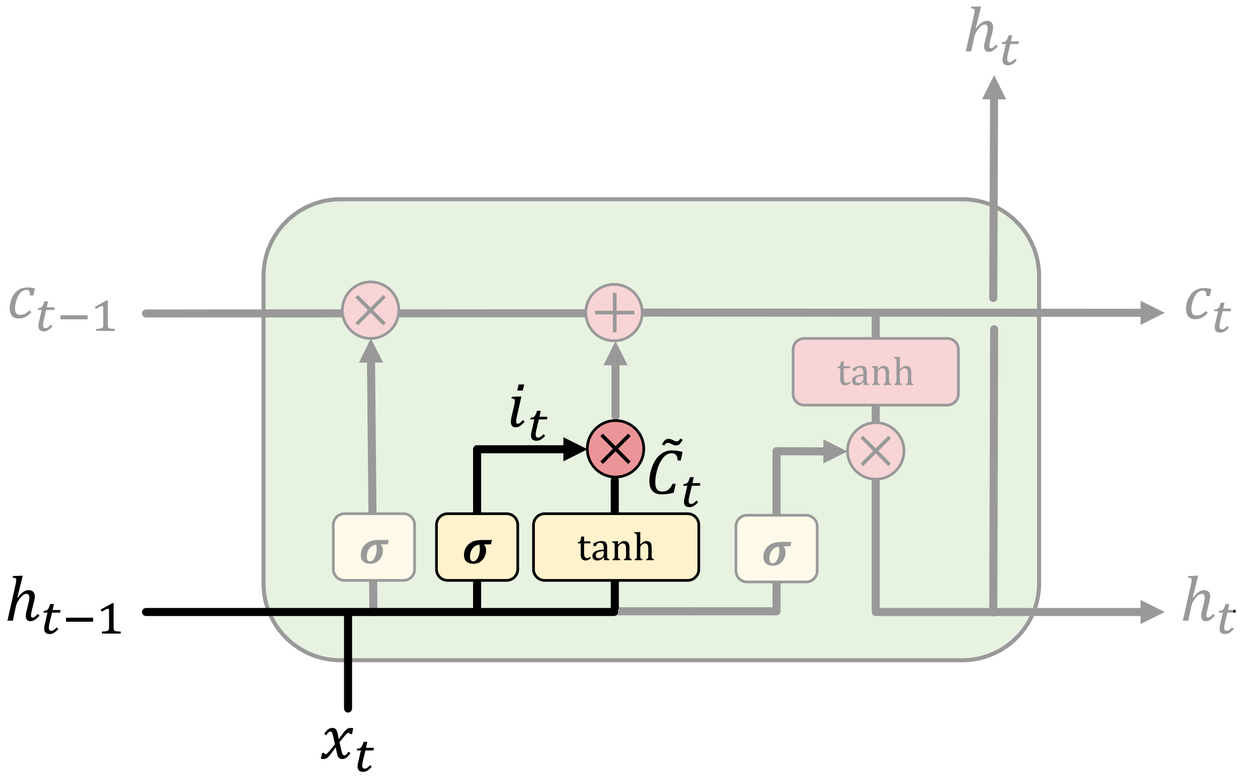


$$f_t = \sigma(\boldsymbol{W_i}[\, h_{t-1}, x_t \,] + b_f)$$

- Use previous cell output and input
- Sigmoid: value 0 and 1 – "completely forget" vs. "completely keep"

ex: Forget the gender pronoun of previous subject in sentence.

LSTMs: identify new information to be stored



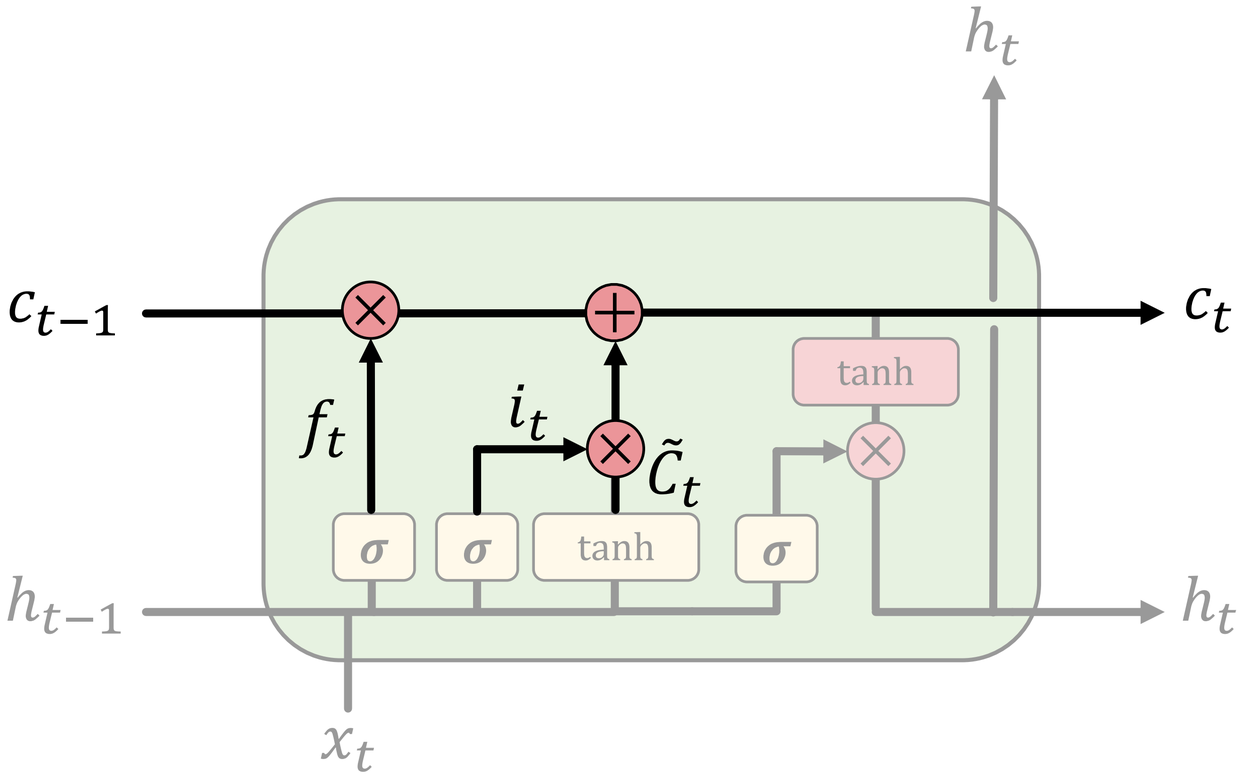
$$i_t = \sigma(\boldsymbol{W_i}[\, h_{t-1}, x_t \,] + b_i)$$

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

- Sigmoid layer: decide what values to update
- Tanh layer: generate new vector of "candidate values" that could be added to the state

ex: Add gender of new subject to replace that of old subject.

LSTMs: update cell state

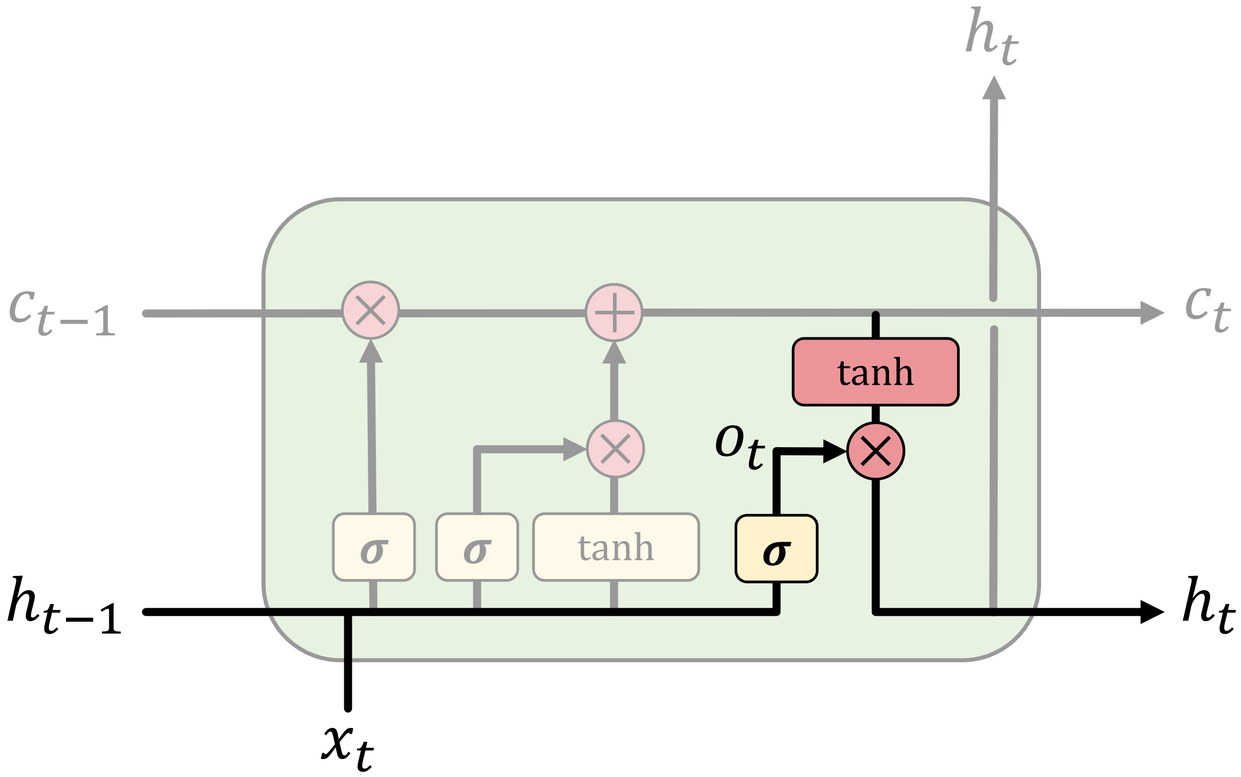


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

- Apply forget operation to previous internal cell state: $f_t * C_{t-1}$
- Add new candidate values, scaled by how much we decided to update: $i_t * \tilde{C}_t$

ex: Actually drop old information and add new information about subject's gender.

LSTMs: output filtered version of cell state



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

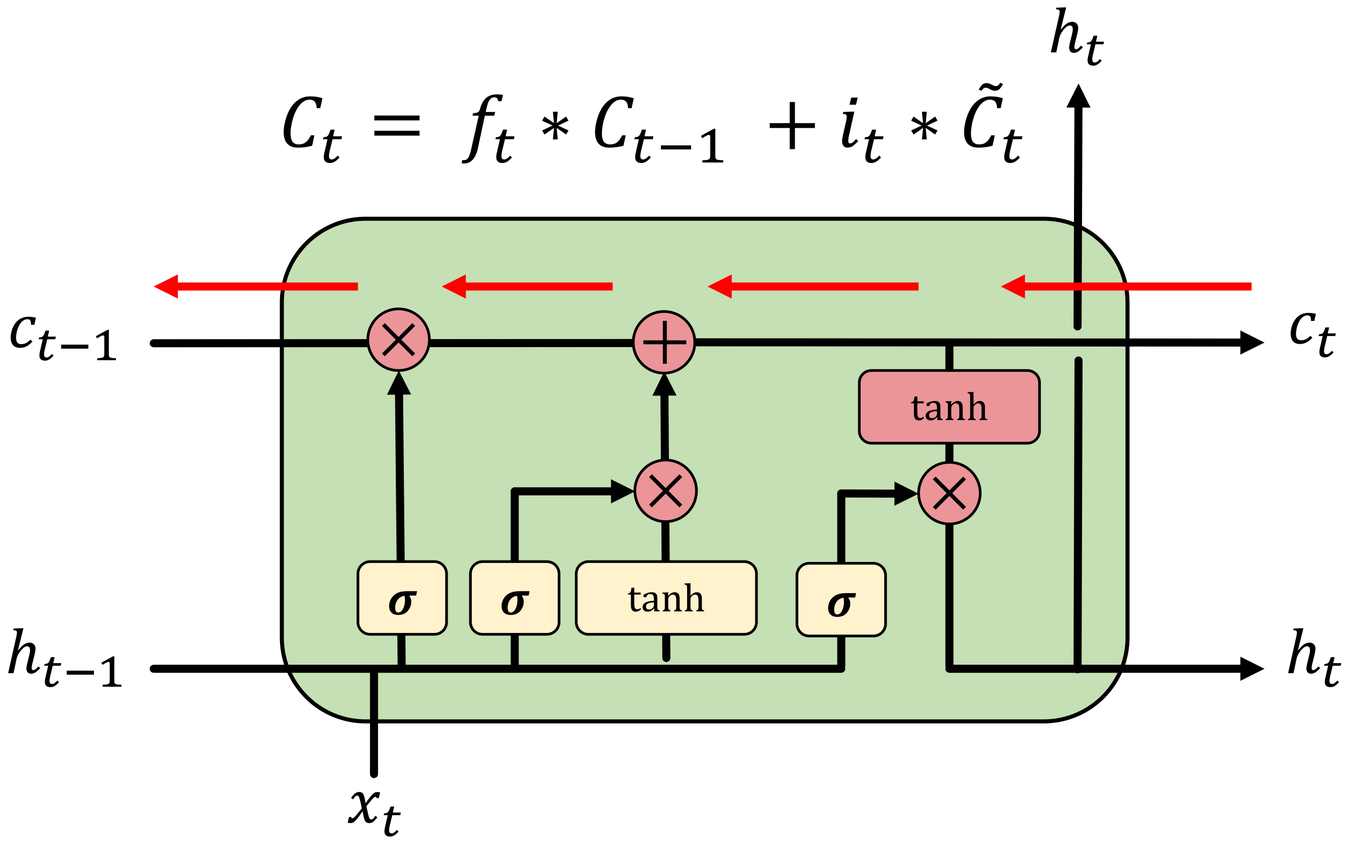
$$h_t = o_t * \tanh(C_t)$$

- Sigmoid layer: decide what parts of state to output
- Tanh layer: squash values between -1 and 1
- $o_t * \tanh(C_t)$: output filtered version of cell state

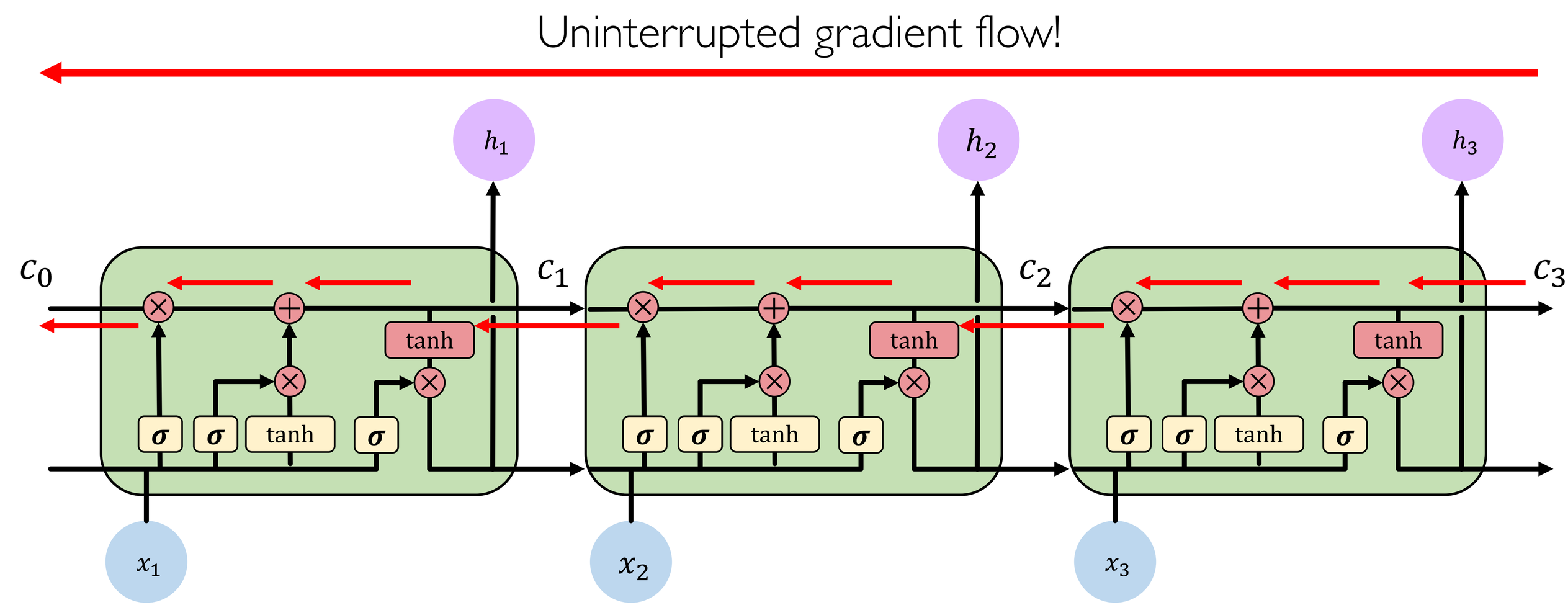
ex: Having seen a subject, may output information relating to a verb.

LSTM gradient flow

Backpropagation from C_t to C_{t-1} requires only elementwise multiplication!
No matrix multiplication → avoid vanishing gradient problem.



LSTM gradient flow



LSTMs: key concepts

1. Maintain a **separate cell state** from what is outputted
2. Use **gates** to control the **flow of information**
   - Forget gate gets rid of irrelevant information
   - Selectively update cell state
   - Output gate returns a filtered version of the cell state
3. Backpropagation from c_t to c_{t-1} doesn't require matrix multiplication: **uninterrupted gradient flow**

Next time..

Object Detection