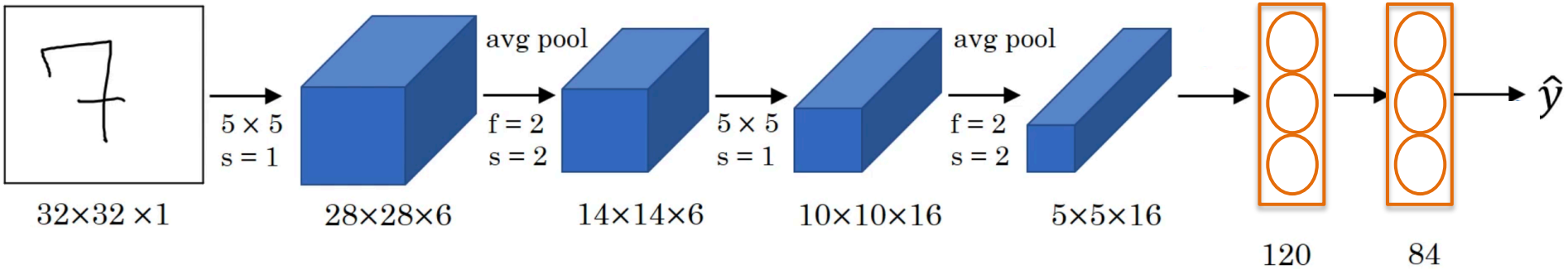
Lecture 10 Recap

LeNet

- Digit recognition: 10 classes

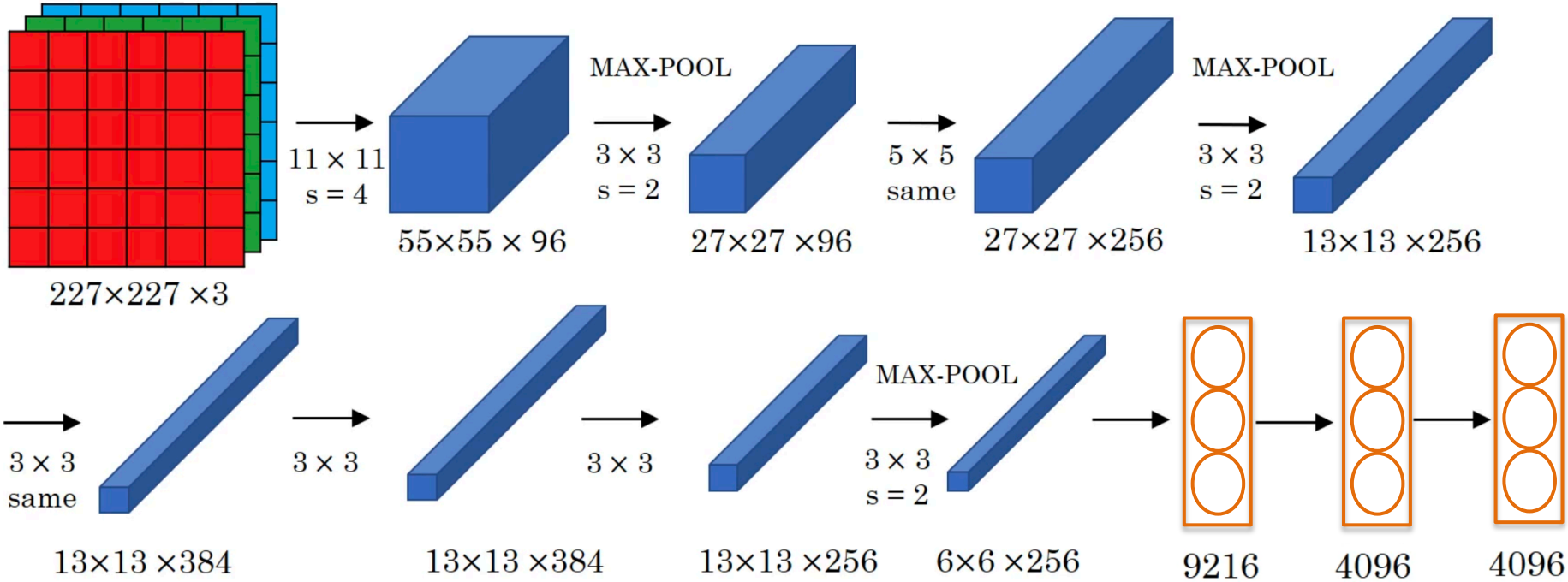
60k parameters



- Conv -> Pool -> Conv -> Pool -> Conv -> FC

As we go deeper: Width, Height ↓ Number of Filters ↑

AlexNet



- Softmax for 1000 classes

[Krizhevsky et al., ANIPS'12] AlexNet

VGGNet

- Striving for simplicity
  - Conv -> Pool -> Conv -> Pool -> Conv -> FC
  - Conv=3x3, s=1, same; Maxpool=2x2, s=2
- As we go deeper: Width, Height ↓ Number of Filters ↑
- Called VGG-16: 16 layers that have weights

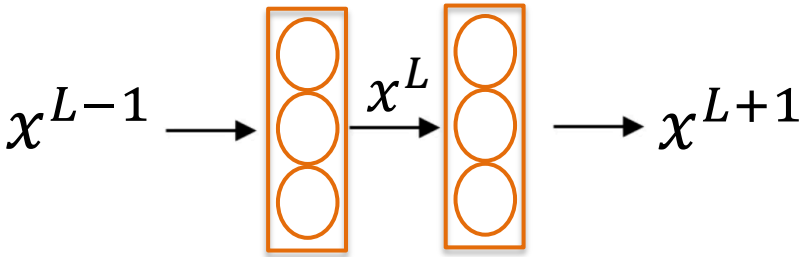
  138M parameters
- Large but simplicity makes it appealing

[Simonyan et al., ICLR'15] VGGNet

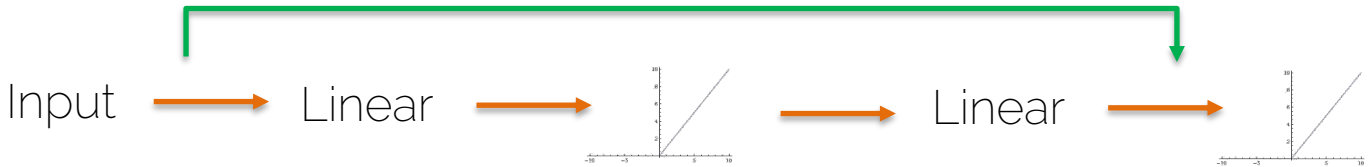
Residual Block

being useful in backprop as well

- Two layers

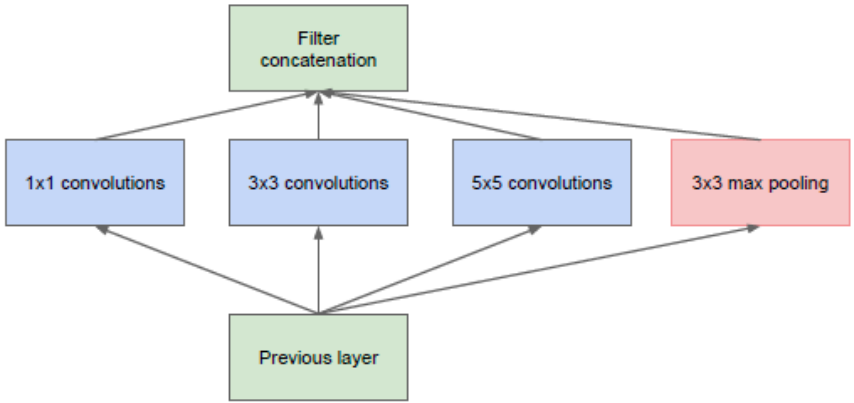


$$x^{L+1} = f(W^{L+1}x^L + b^{L+1} + x^{L-1})$$



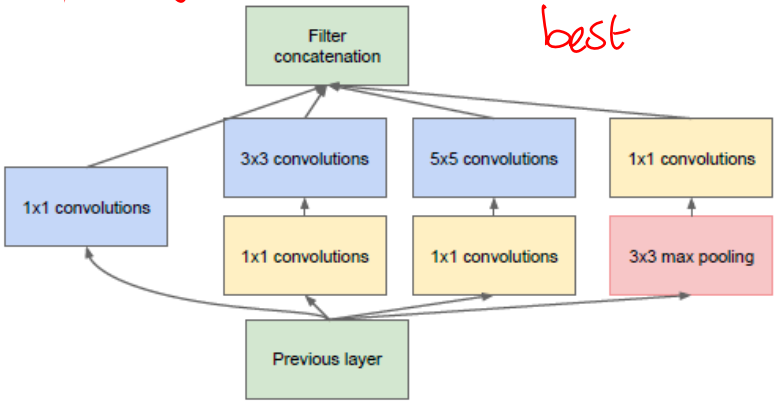
$$x^{L+1} = f(W^{L+1}x^L + b^{L+1})$$

Inception Layer



(a) Inception module, naïve version

⇒ applying all, let the net learn what's best



(b) Inception module with dimensionality reduction

1X1 conv → to reduce feature map i.e depth i.e # multiplications

Lecture 11

* Transfer Learning
* RNN
* LSTM

Transfer Learning

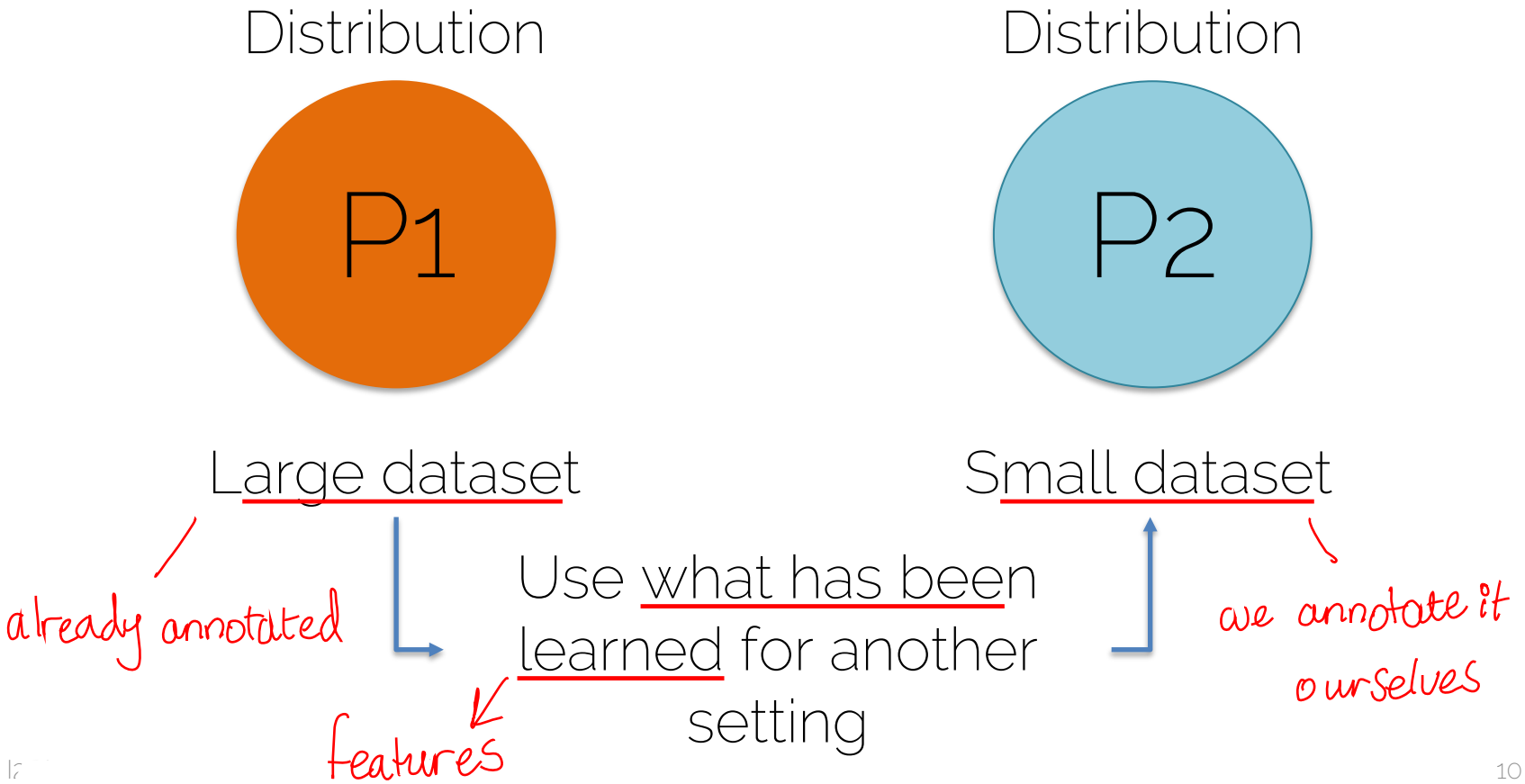
Transfer Learning

- Training your own model can be difficult with limited data and other resources

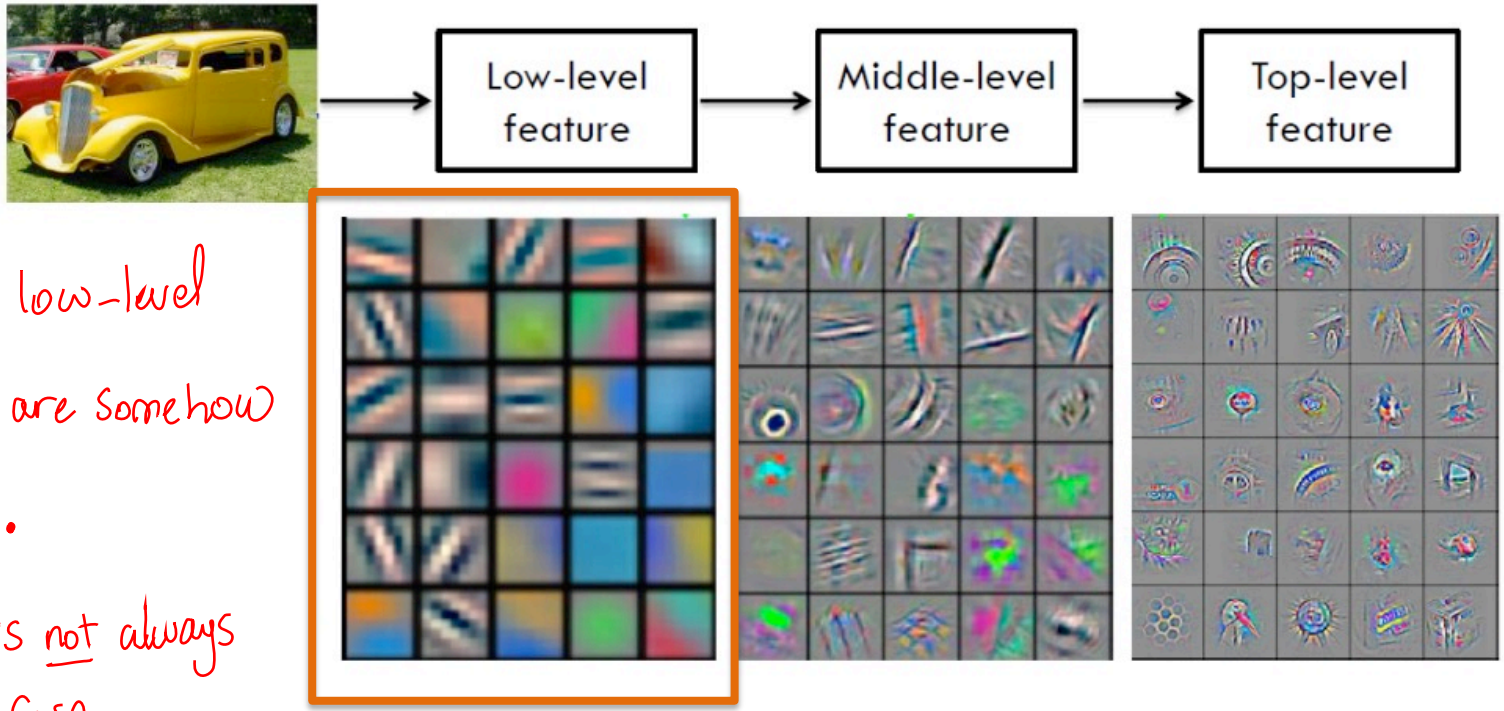
e.g.,

- It is a laborious task to manually annotate your own training dataset

→ Why not reuse already pre-trained models?


Distribution
P1
Large dataset
already annotated
Distribution
P2
Small dataset
we annotate it ourselves
Use what has been learned for another setting
features

Transfer Learning for Images



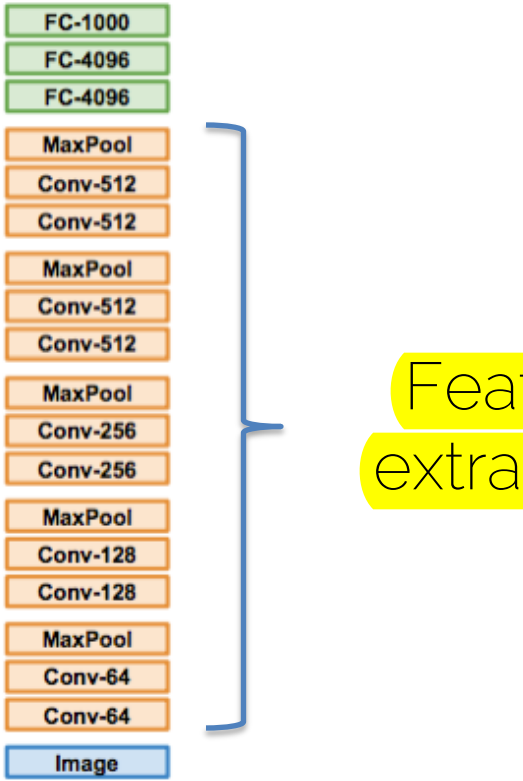
Usually low-level features are somehow the same.

Yet this is not always the case

[Zeiler al., ECCV'14] Visualizing and Understanding Convolutional Networks

Transfer Learning

Trained on ImageNet



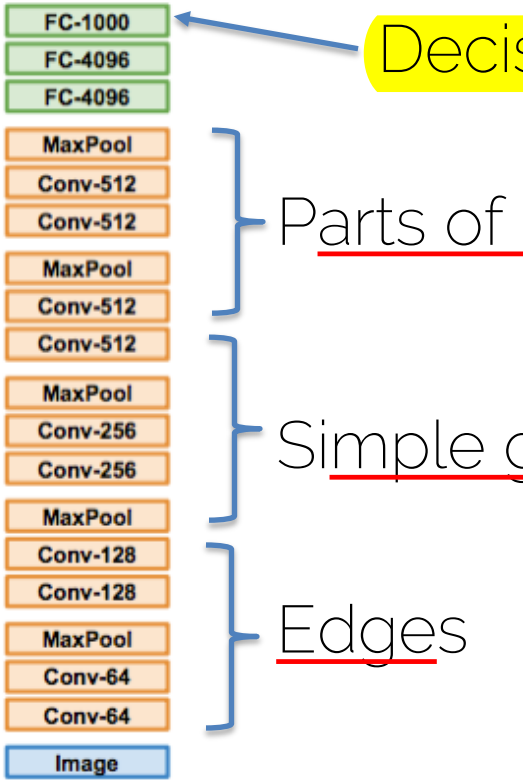
we are interested in those layers

Feature extraction

[Donahue et al., ICML'14] DeCAF,
[Razavian et al., CVPRW'14] CNN Features off-the-shelf

Trained on ImageNet

Understand the flow of feature capturing



Decision layers

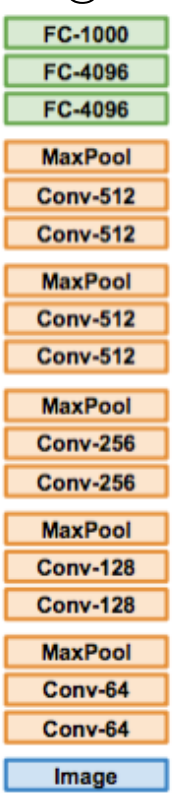
shall be changed to adapt for diff tasks

Parts of an object (wheel, window) "high-level"

Simple geometrical shapes (circles, etc)

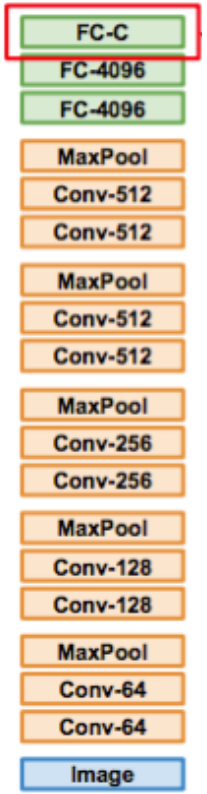
Edges "low-level"

Trained on ImageNet



TRAIN

Just



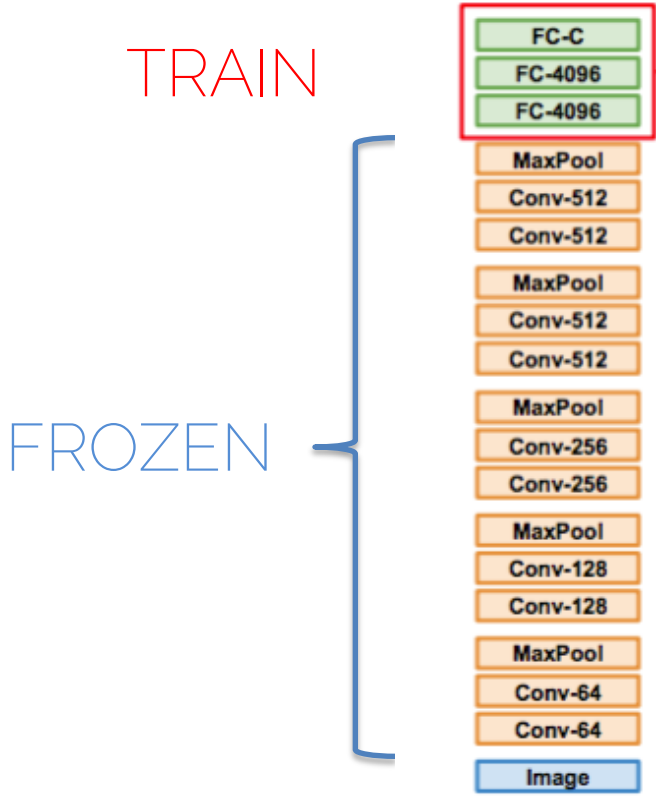
FROZEN

Assuming that

↳ New dataset with C classes

train the last layer, with high learning rate

If the dataset is big enough train more layers with a low learning rate



When Transfer Learning makes Sense

- When task T1 and T2 have the same input (e.g. an RGB image)

- When you have more data for task T1 than for task T2

- When the low-level features for T1 could be useful to learn T2

e.g RGB is diff from CTscan

Now you are:

- Ready to perform image classification on any dataset
- Ready to design your own architecture
- Ready to deal with other problems such as semantic segmentation (Fully Convolutional Network)

Recurrent Neural Networks

Processing Sequences

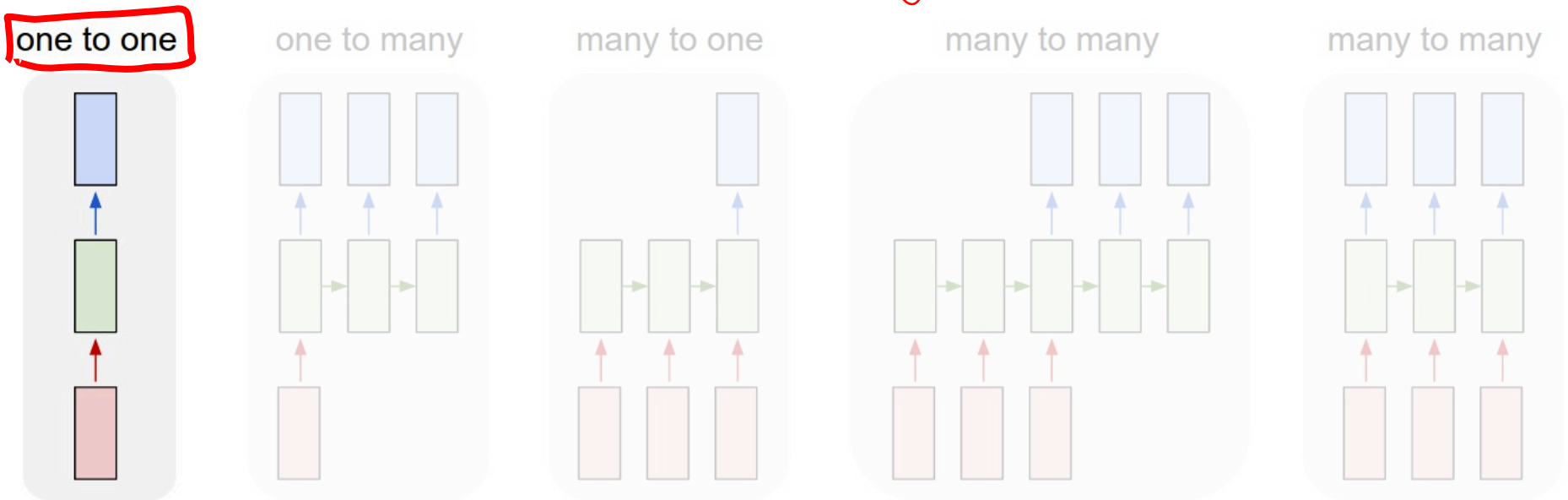
- Recurrent neural networks process sequence data

- Input/output can be sequences

* Design Criteria :

1. handle variable-length sequences.
2. track long-term dependencies.
3. maintain info about order.
4. share parameters across the sequence.

RNNs are ~~x~~ extremely Flexible



Classic Neural Networks for Image Classification

RNNs are Flexible

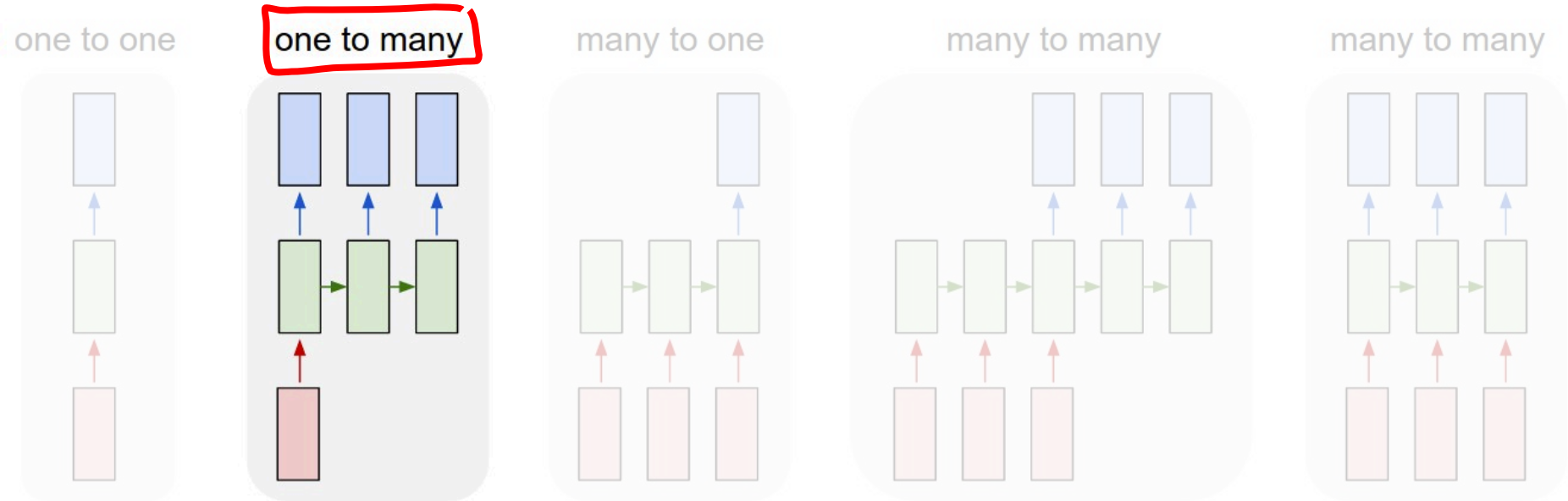
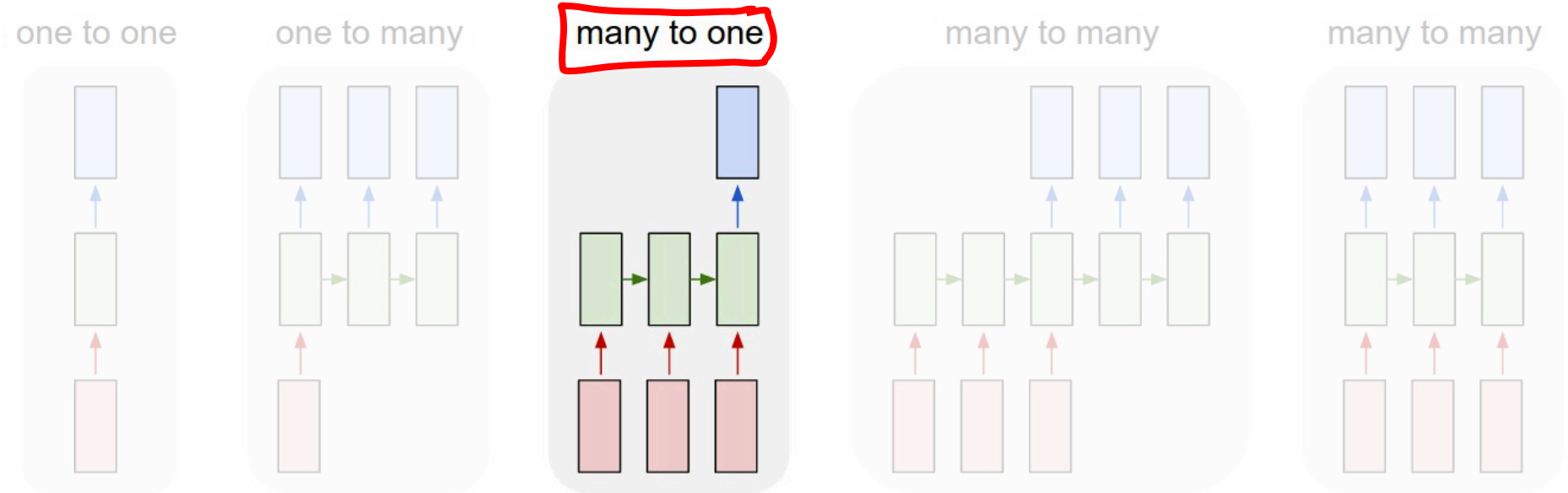


Image captioning

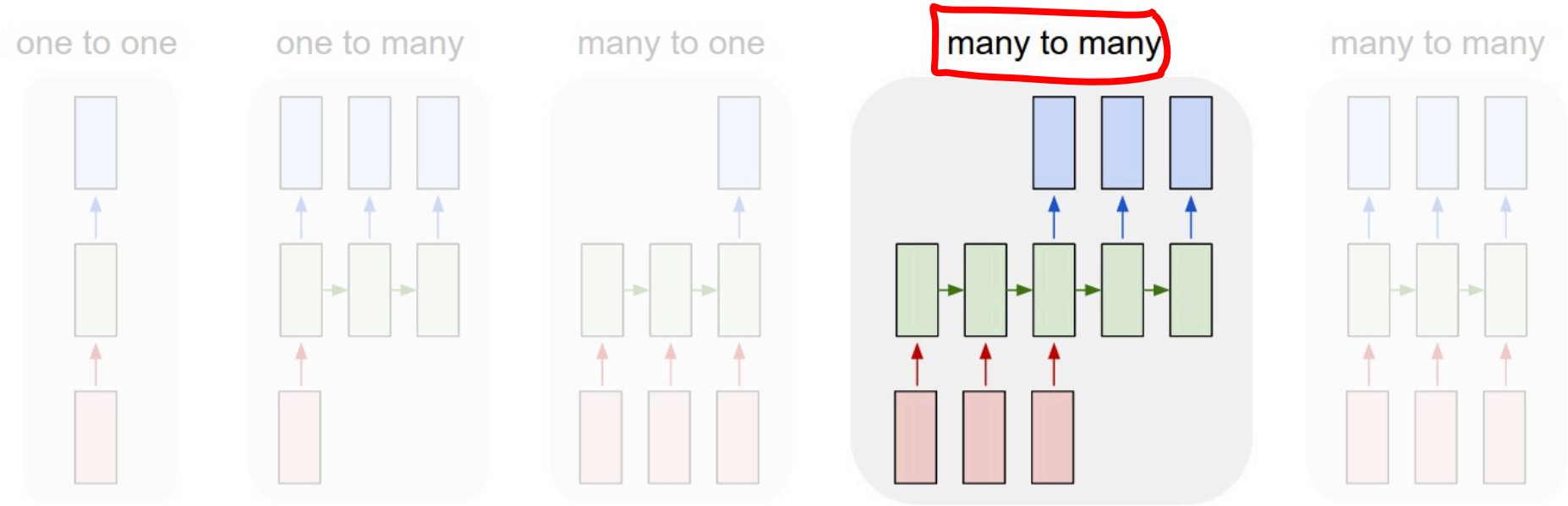
Sentence explaining the image

RNNs are Flexible



Language recognition

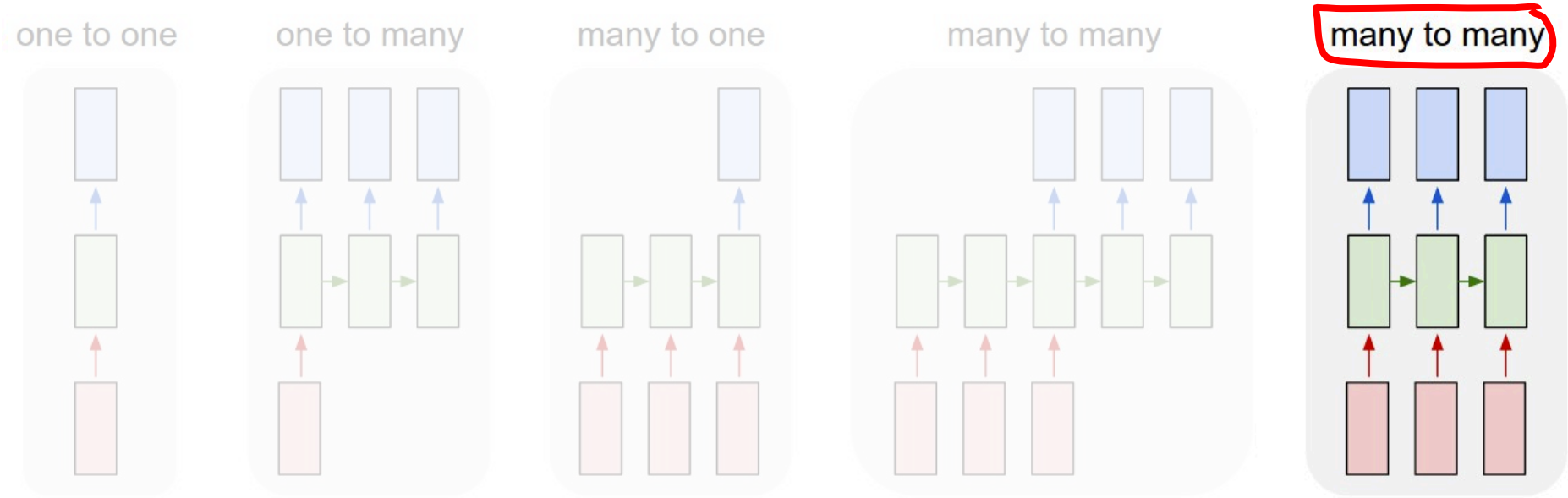
RNNs are Flexible



reading a little bit first ..

Machine translation

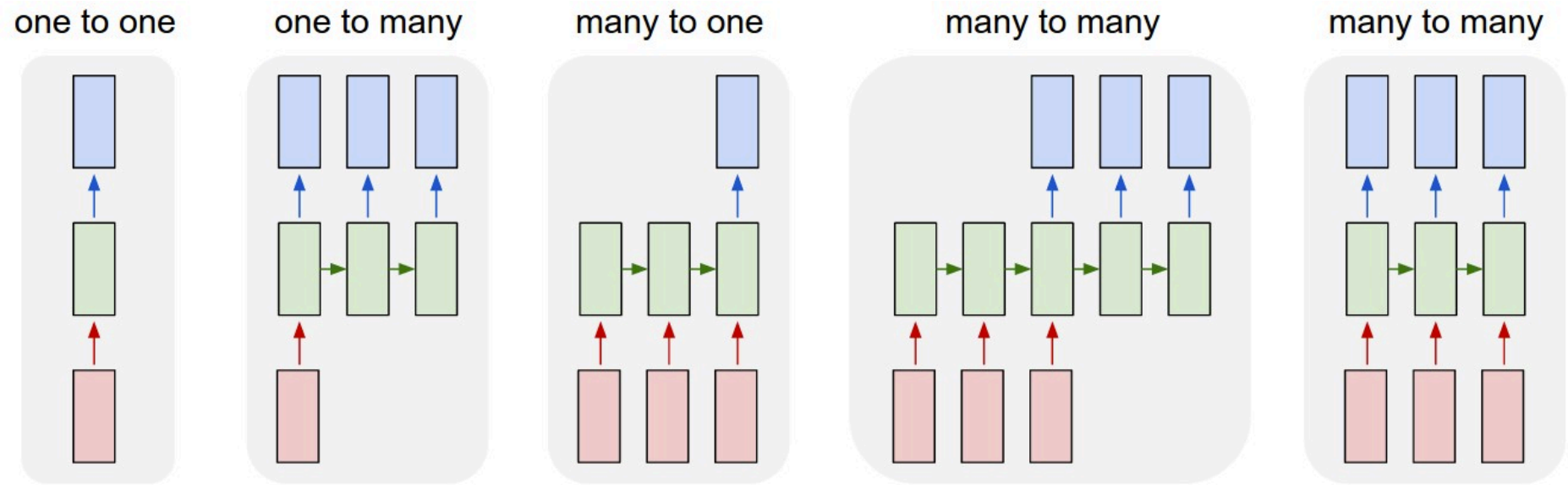
RNNs are Flexible



immediate action is required

Event classification

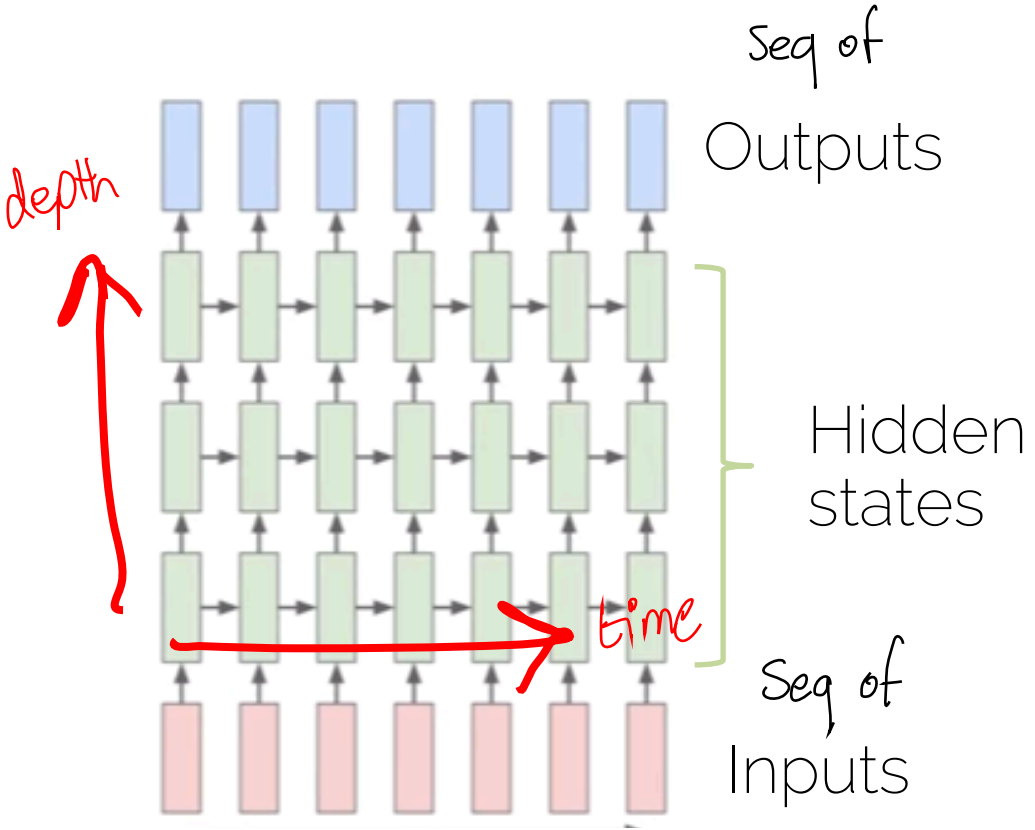
RNNs are Flexible



Source: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Basic Structure of an RNN

- Multi-layer RNN



Basic Structure of an RNN

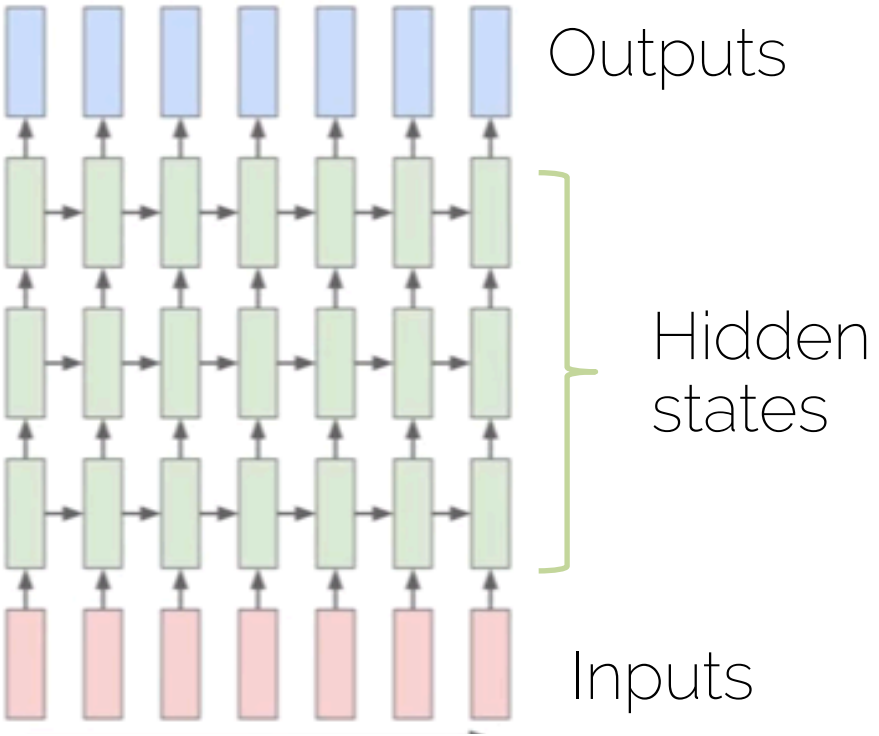
- Multi-layer RNN

The hidden state will have its own internal dynamics

↓

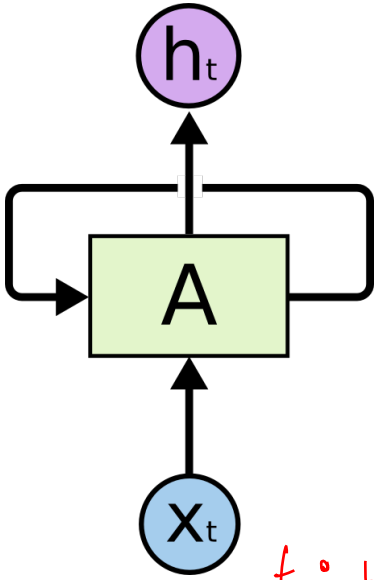
More expressive model!

definitely expensive as well



* We want to have notion of "time" or "sequence"

Hidden state



t : time step

- $\boldsymbol{A_t = \theta_c A_{t-1} + \theta_x x_t}$

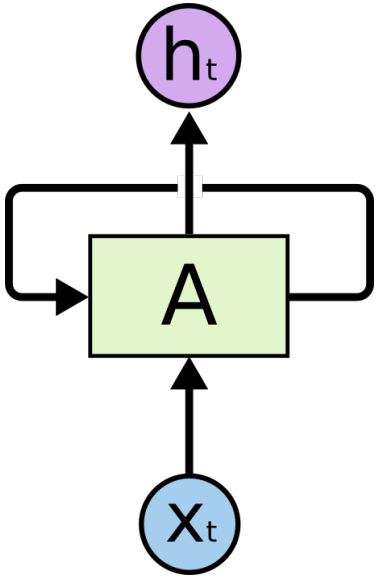
Previous hidden state

input

Our sources of info

[Olah, https://colah.github.io '15] Understanding LSTMs

Hidden state



- $\boldsymbol{A_t = \theta_c A_{t-1} + \theta_x x_t}$

Parameters to be learned

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

internal cell state

f_W parameterized by W

prev hidden state

input vec at time step t

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

"hidden state update"

- $y_t = W_{hy}^T h_t$

"output vec"

- $\boldsymbol{A_t = \theta_c A_{t-1} + \theta_x x_t}$
- $\boldsymbol{h_t = \theta_h A_t}$

Note: non-linearities ignored for now

they exist, of course

the main idea in RNN

* All prev hidden states, processed with θ_x, are processed via θ_c.

* Having A_t, we need to predict.

  So all output prediction will be created through θ_h.

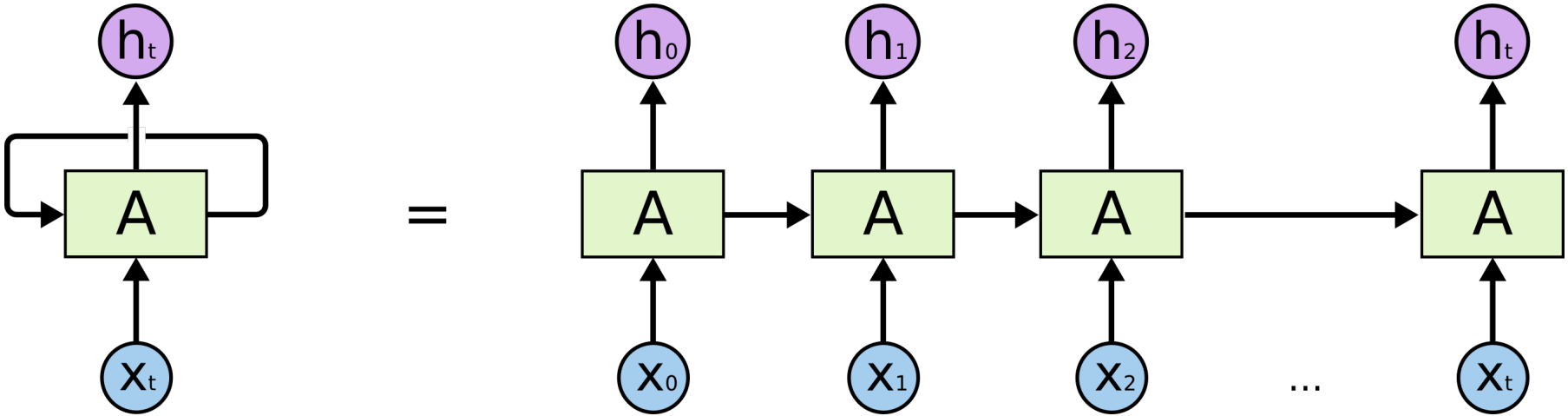
$$A_t = \theta_c A_{t-1} + \theta_x x_t$$

$$h_t = \theta_h A_t$$

Same parameters for each time step = generalization!

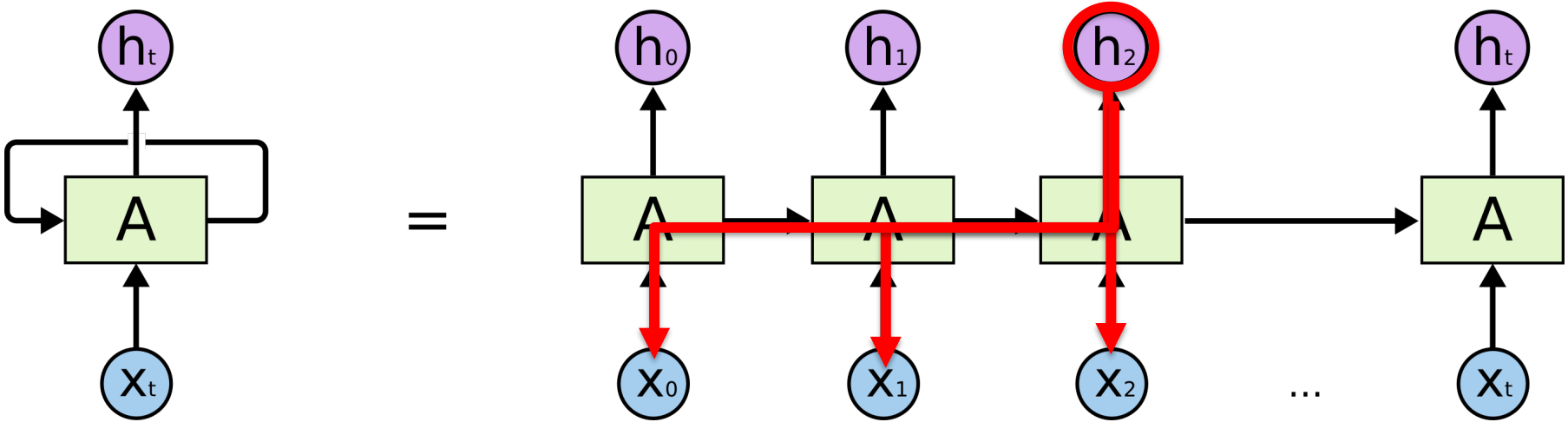
- Unrolling RNNs

Same function for the hidden layers



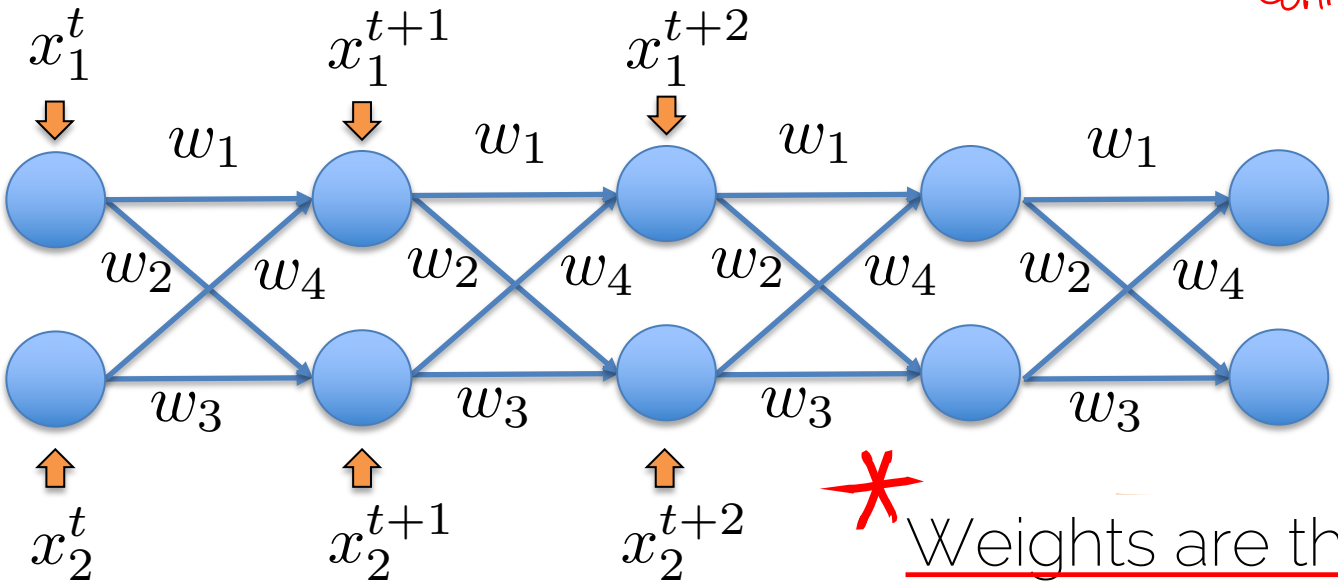
Long Dependencies

* h_2 does not to only depend on X_2, but also X_1 & X_0.



idea : Connection bet t & t+1 is the same as Connection bet t-1 & t

- Unrolling RNNs as feedforward nets



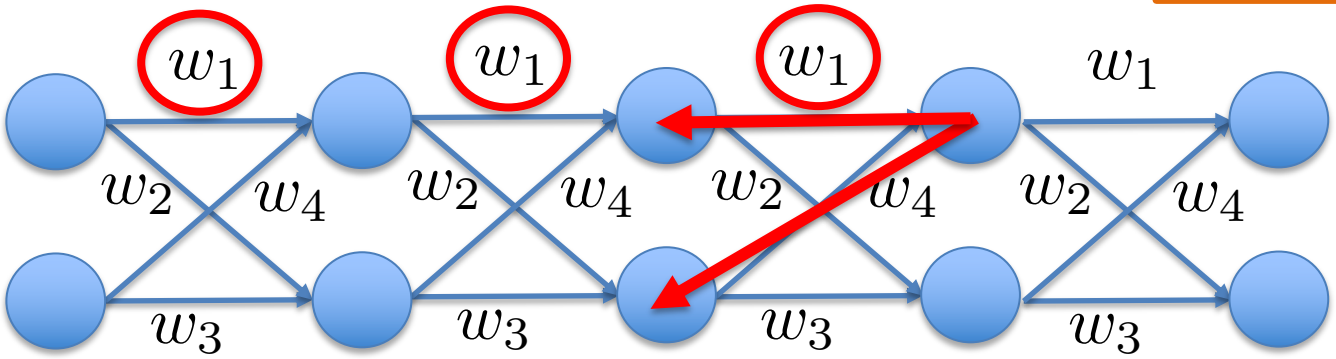
* Weights are the same!

regardless of the input

Backprop through an RNN

- Unrolling RNNs as feedforward nets

Chain rule



weights are Combined in such high-dim hidden states

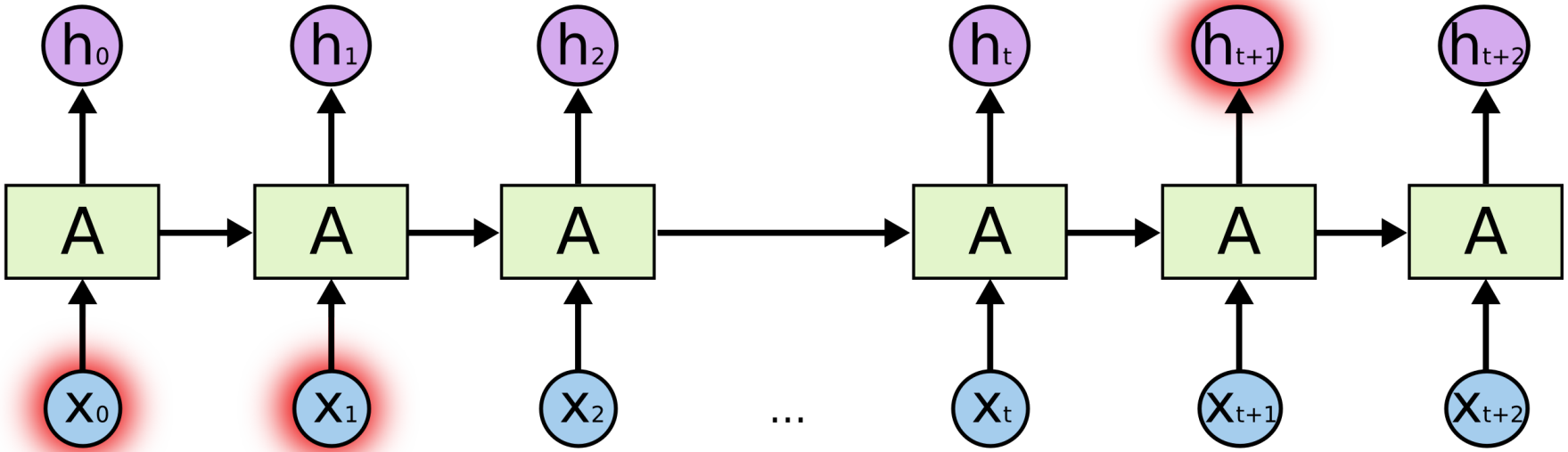
All the way to $t = 0$

* Add the derivatives at different times for each weight

Long-term Dependencies

it's challenging to keep key info from early stages

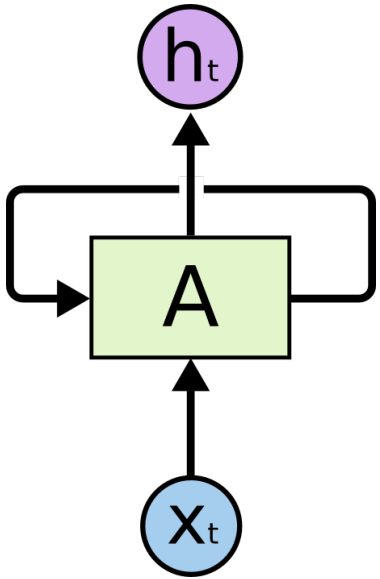
more on slide #40



I moved to Germany ... so I speak German fluently.

- Simple recurrence $\boldsymbol{A_t = \theta_c A_{t-1} + \theta_x x_t}$

- Let us forget the input $\boldsymbol{A_t = \theta_c{}^t A_0}$



Same weights are multiplied over and over again

- Simple recurrence $\boldsymbol{A_t = \theta_c{}^t A_0}$

In case of scalar weights

What happens to small weights?

Vanishing gradient

to solve → act. fn
→ init weights
→ change archit-ecture

What happens to large weights?

Exploding gradient

to solve → grad clipping

- Simple recurrence $\boldsymbol{A_t = \theta_c{}^t A_0}$

- If $\boldsymbol{\theta}$ admits eigendecomposition

weights is matrix

$$\boldsymbol{\theta = Q \Lambda Q^T}$$

Matrix of eigenvectors

Diagonal of this matrix are the eigenvalues

To solve vanishing grad:

1. Activation fn → using ReLU, prevents f' from shrinking only when $x > 0$ though
2. weight initialization
3. Gated cells → more complex recurrent units with gates

- Orthogonal $\boldsymbol{\theta}$ allows us to simplify the recurrence
  - $\boldsymbol{A_t = Q\Lambda^t Q^T A_0}$

In case of matrix weights

What happens to eigenvalues with magnitude less than one?

Vanishing gradient

needless to say Vanishing is worst

What happens to eigenvalues with magnitude larger than one?

Exploding gradient

Gradient clipping

- Simple recurrence $\boldsymbol{A_t = \theta_c{}^t A_0}$

Hence, the obvious choice left will be:

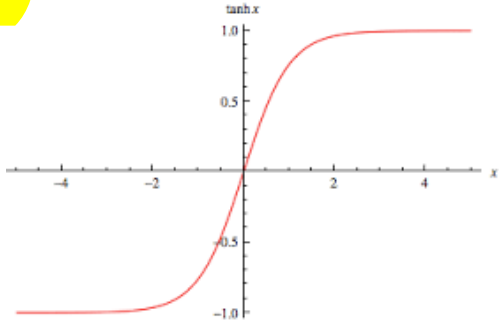
* Let us just make a matrix with eigenvalues = 1

meaning → Allow the **cell** to maintain its "*state*"

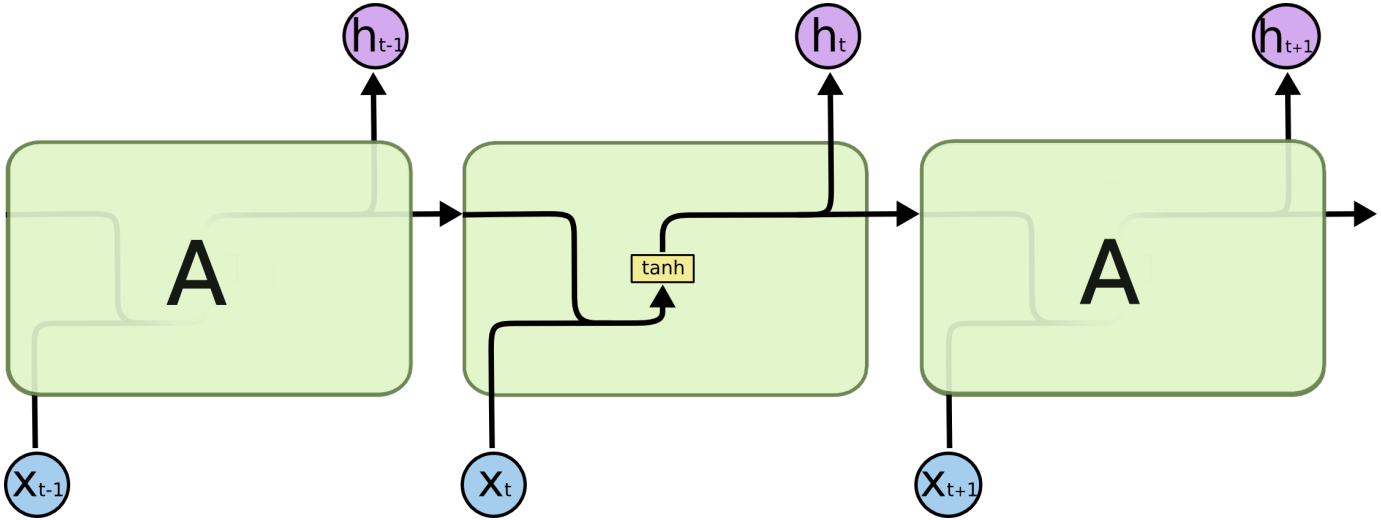
Vanishing Gradient

comes from :

- 1. From the weights $\boldsymbol{A_t = \theta_c{}^t A_0}$



- 2. From the activation functions ($\boldsymbol{tanh}$)

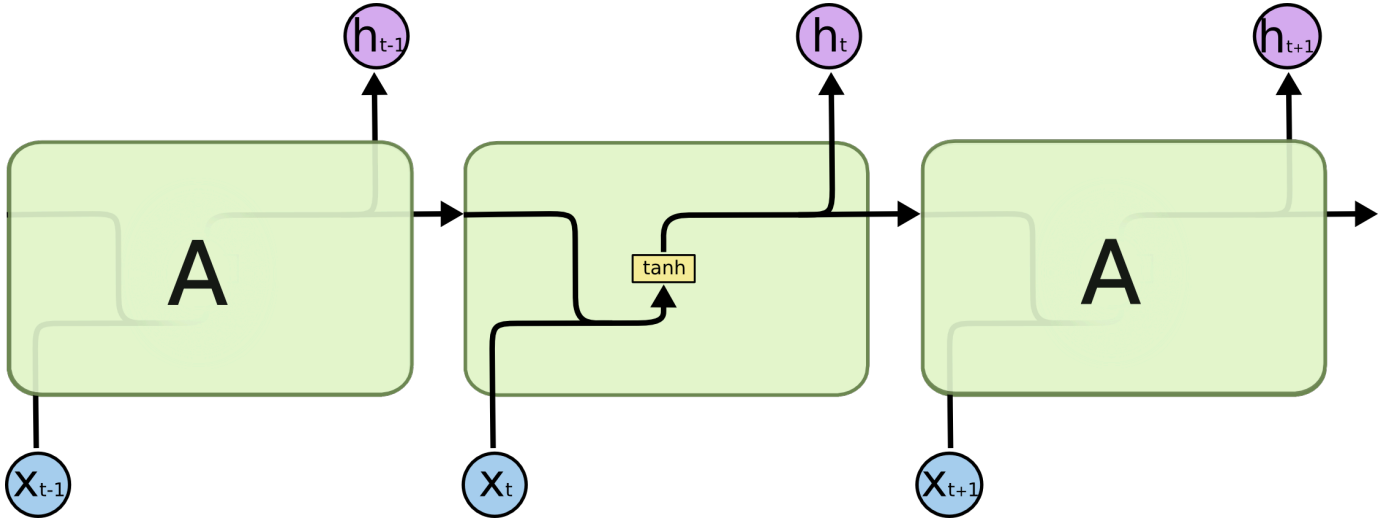


Our soln:

- 1. From the weights $A_t = \theta^t A_0$

1

- 2. From the activation functions ($tanh$) ← RNN typically use



LSTMs: modules contain computational blocks that control info flow, by tracking info through time.

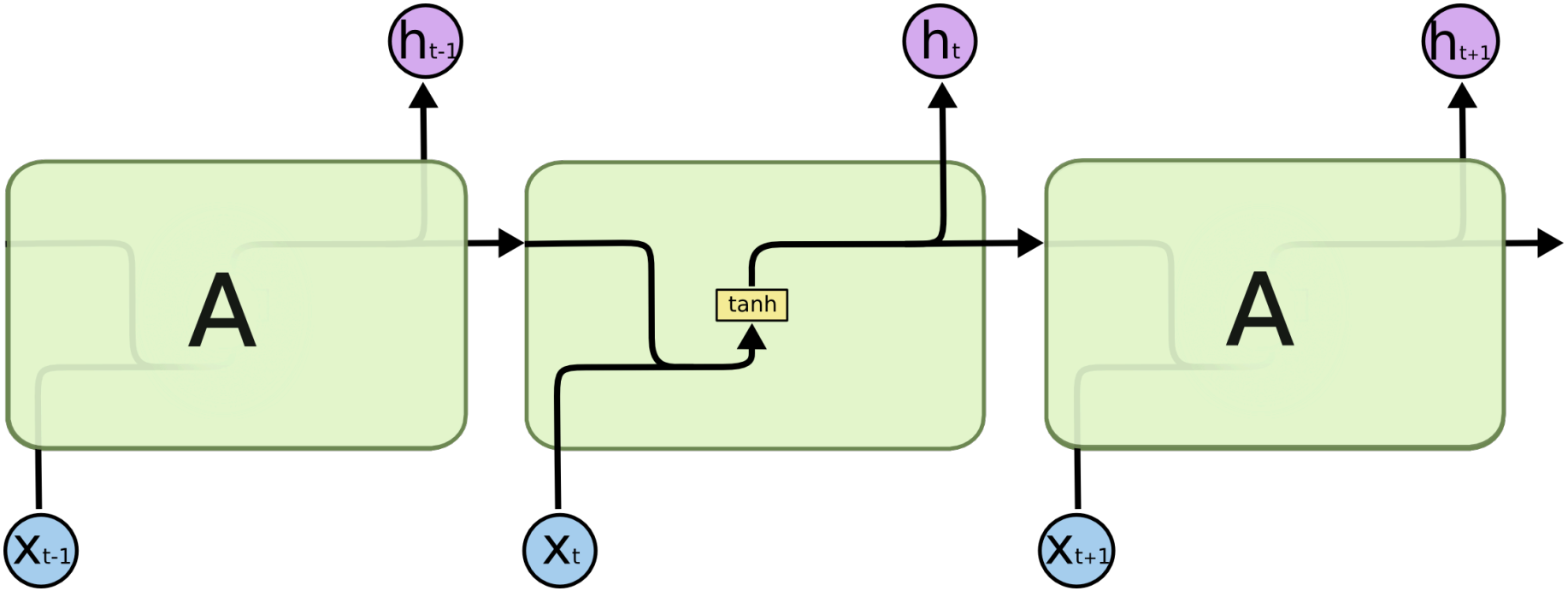
Long Short Term Memory

they set $\lambda = 1$, avoiding vanishing gradients & allow longer seq to train

[Hochreiter et al., Neural Computation'97] Long Short-Term Memory

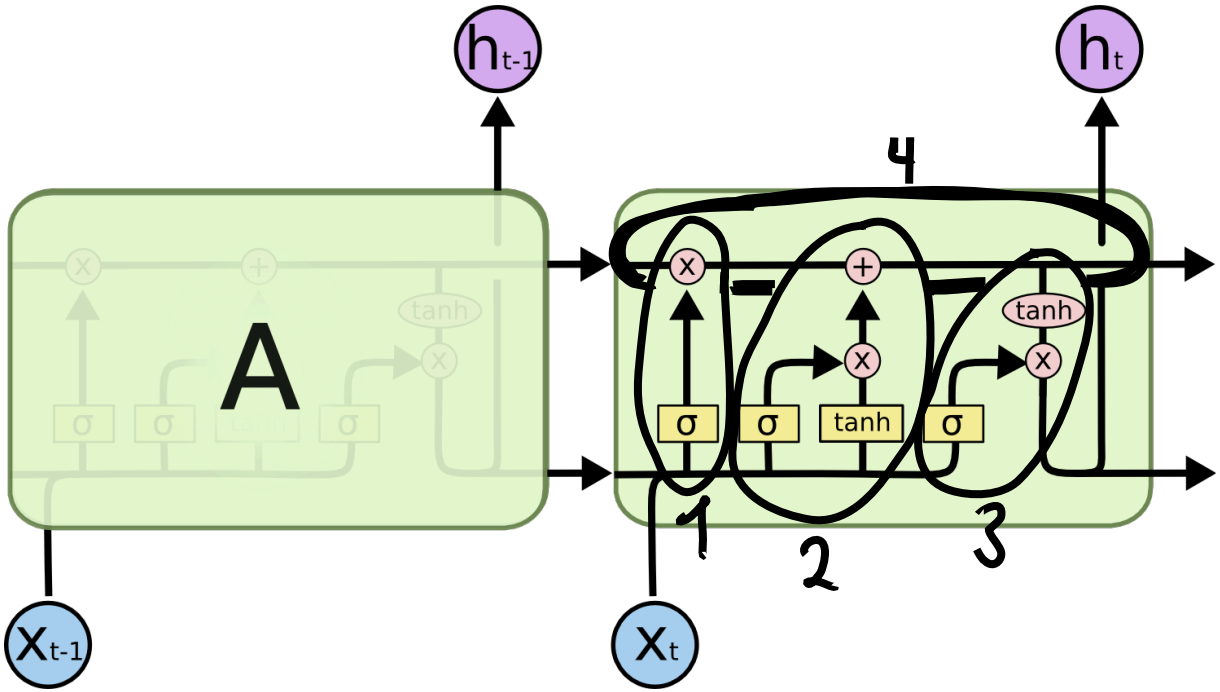
Long-Short Term Memory Units

Simple RNN has **tanh** as non-linearity



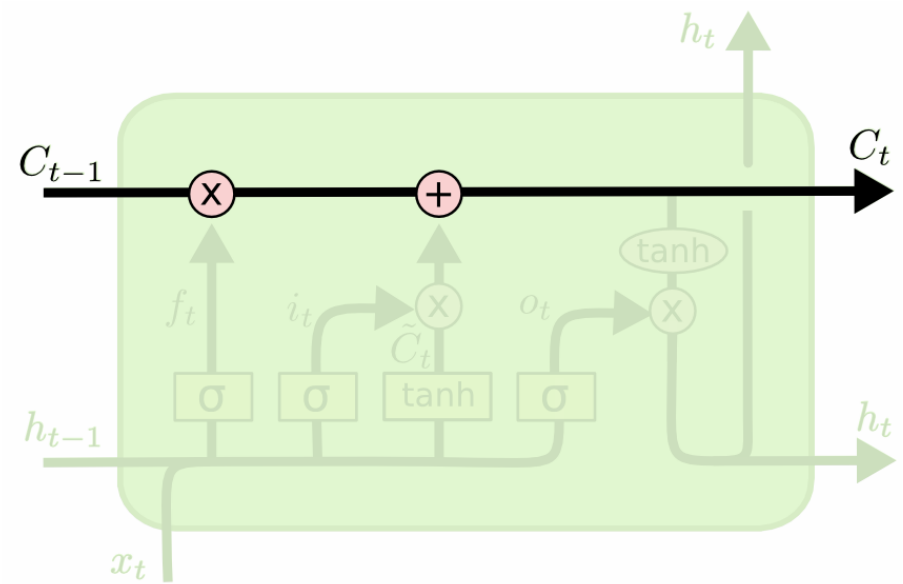
[Olah, https://colah.github.io '15] Understanding LSTMs

LSTM



1. forget
2. Store
3. update
4. output

- Key ingredients
- Cell = transports the information through the unit



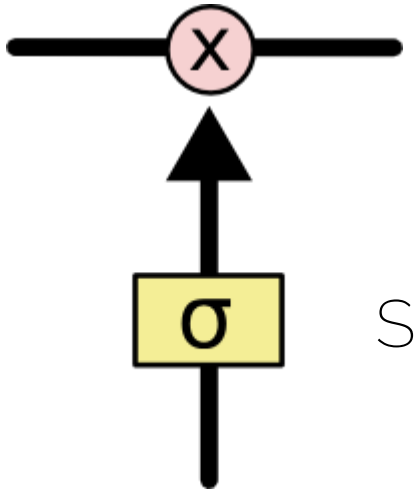
Cell

gate

will be changed according to current role at time step t

- Gate = remove or add information to the cell state

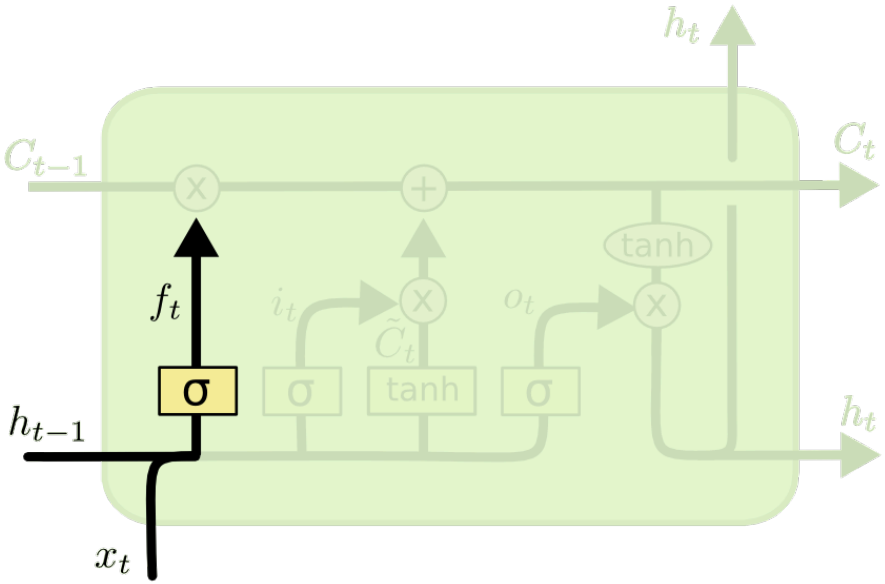
* All gates are governed by sigmoid



Sigmoid

LSTM: Step by Step

2.1 Forget gate $\boldsymbol{f_t} = sigm(\boldsymbol{\theta}_{xf}\boldsymbol{x}_t + \boldsymbol{\theta}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{b}_f)$

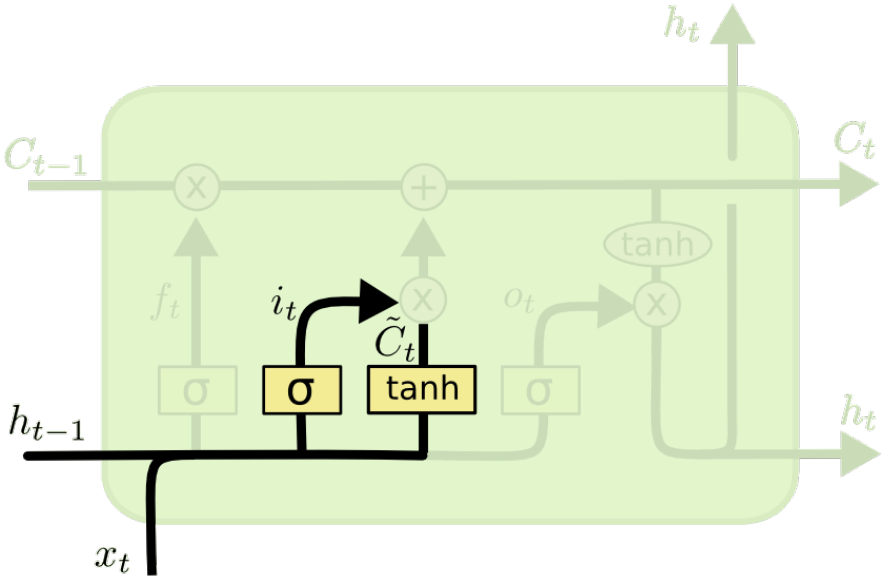


Decides when to erase the cell state

Sigmoid = output between **0** (forget) and **1** (keep)

feature decision

2.2 Input gate $i_t = sigm(\theta_{xi} x_t + \theta_{hi} h_{t-1} + b_i)$

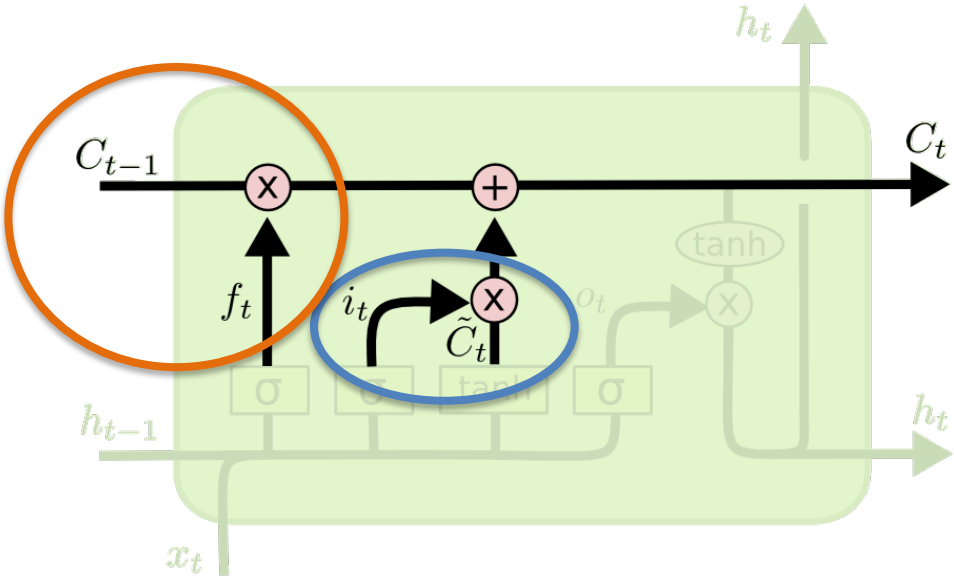


Decides which values will be updated i.e adding / subtracting info to the cell state

New cell state, output from a **tanh** $(-1,1)$

t & t-1

- Element-wise operations



$$C_t = f_t \odot C_{t-1} + i_t \odot g_t$$

Previous states

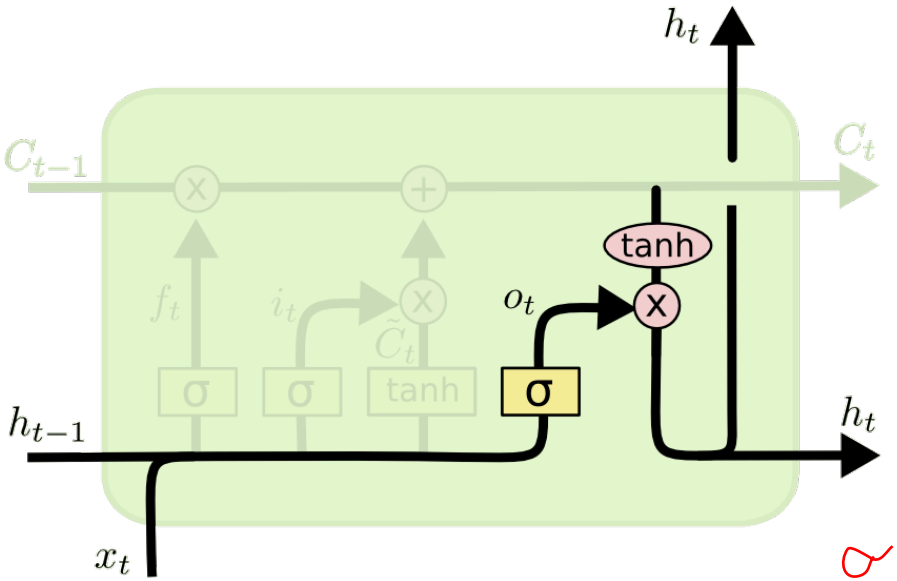
Current state

how f_t & i_t interact; in order to update the cell

actually predicting hidden state at t

passed to $t+1$; so it knows what computation happened prev

2.3 Output gate $\boldsymbol{h_t} = \boldsymbol{o_t} \odot \tanh(\boldsymbol{C_t})$



Decides which values will be outputted

Output from a **tanh** $(-1, 1)$

σ : which values to keep / erase

tanh : compressing C_t bet -1 & 1

Summary

- Forget gate $\boldsymbol{f}_t = sigm(\boldsymbol{\theta}_{xf}\boldsymbol{x}_t + \boldsymbol{\theta}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{b}_f)$
- Input gate $\boldsymbol{i}_t = sigm(\boldsymbol{\theta}_{xi}\boldsymbol{x}_t + \boldsymbol{\theta}_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{b}_i)$
- Output gate $\boldsymbol{o}_t = sigm(\boldsymbol{\theta}_{xo}\boldsymbol{x}_t + \boldsymbol{\theta}_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{b}_o)$
- Cell update $\boldsymbol{g}_t = tanh(\boldsymbol{\theta}_{xg}\boldsymbol{x}_t + \boldsymbol{\theta}_{hg}\boldsymbol{h}_{t-1} + \boldsymbol{b}_g)$
- Cell $\boldsymbol{C}_t = \boldsymbol{f}_t \odot \boldsymbol{C}_{t-1} + \boldsymbol{i}_t \odot \boldsymbol{g}_t$
- Output $\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{C}_t)$

- Forget gate $\boldsymbol{f}_t = sigm(\boldsymbol{\theta}_{xf}\boldsymbol{x}_t + \boldsymbol{\theta}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{b}_f)$
- Input gate $\boldsymbol{i}_t = sigm(\boldsymbol{\theta}_{xi}\boldsymbol{x}_t + \boldsymbol{\theta}_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{b}_i)$
- Output gate $\boldsymbol{o}_t = sigm(\boldsymbol{\theta}_{xo}\boldsymbol{x}_t + \boldsymbol{\theta}_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{b}_o)$
- Cell update $\boldsymbol{g}_t = tanh(\boldsymbol{\theta}_{xg}\boldsymbol{x}_t + \boldsymbol{\theta}_{hg}\boldsymbol{h}_{t-1} + \boldsymbol{b}_g)$
- Cell $\boldsymbol{C}_t = \boldsymbol{f}_t \odot \boldsymbol{C}_{t-1} + \boldsymbol{i}_t \odot \boldsymbol{g}_t$
- Output $\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{C}_t)$

NB

Learned through backpropagation

all weights will be the same for each time step

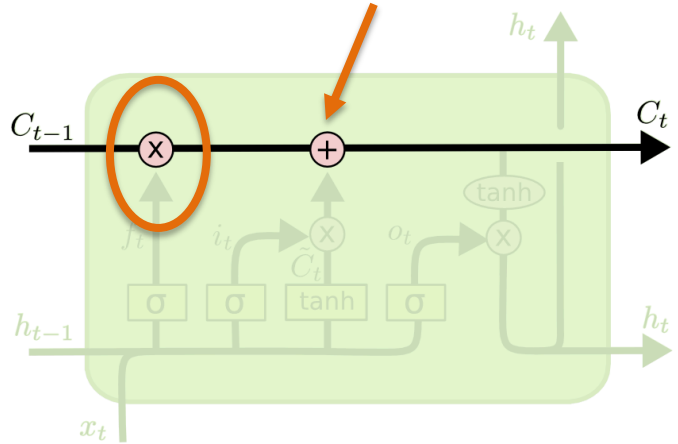
LSTM: Vanishing Gradients?

Did it Solve it?

- 1. From the weights

by setting $\lambda = 1$

- 2. From the activation functions



1 for important information / 0 for non-important

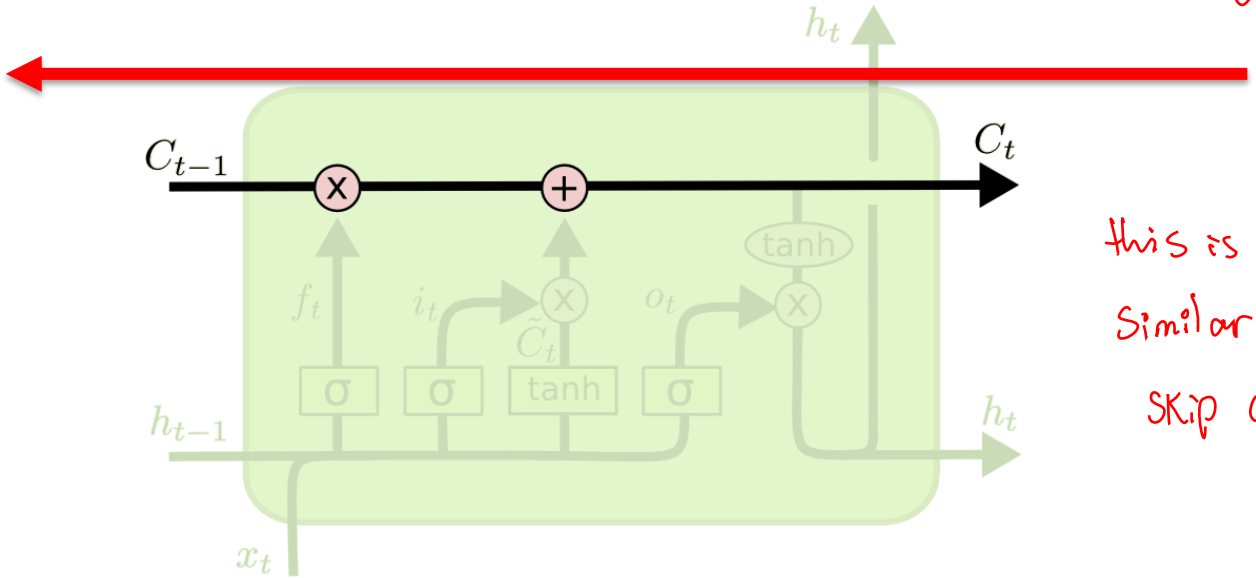
- Cell $\boldsymbol{C_t = f_t \odot C_{t-1} + i_t \odot g_t}$

multiplied over & over going from C_t to C_0

weights

Identity function

- Highway for the gradient to flow



* In case of vanishing grad, the summation will prevent its propagation

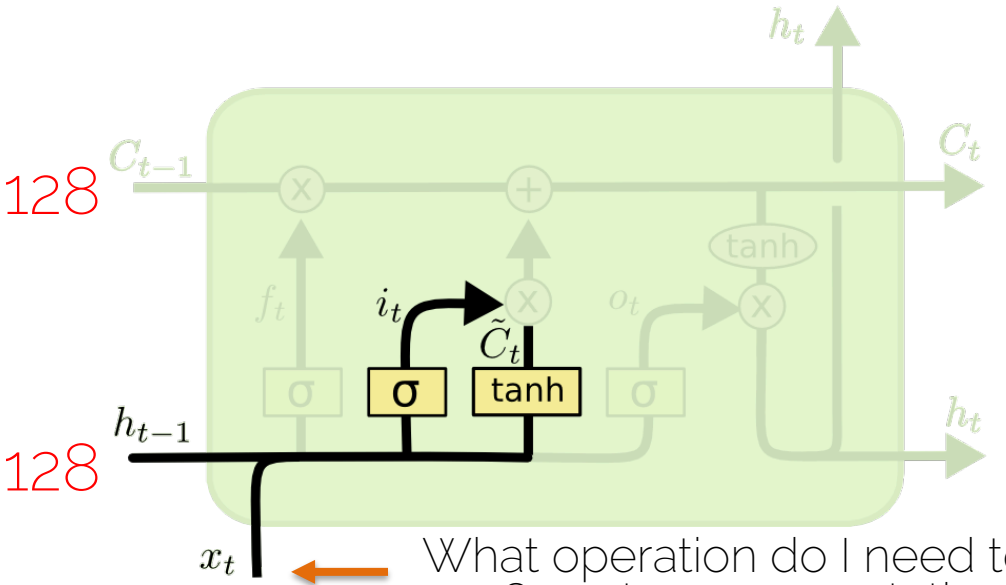
this is somehow similar to Skip connections

Remember

Dimensions

- Cell update

$$\overset{128}{\boldsymbol{g}_t} = tanh(\boldsymbol{\theta}_{xg}\overset{128}{\boldsymbol{x}_t} + \boldsymbol{\theta}_{hg}\overset{128}{\boldsymbol{h}_{t-1}} + \boldsymbol{b}_g)$$



When coding an LSTM, we have to define the size of the hidden state

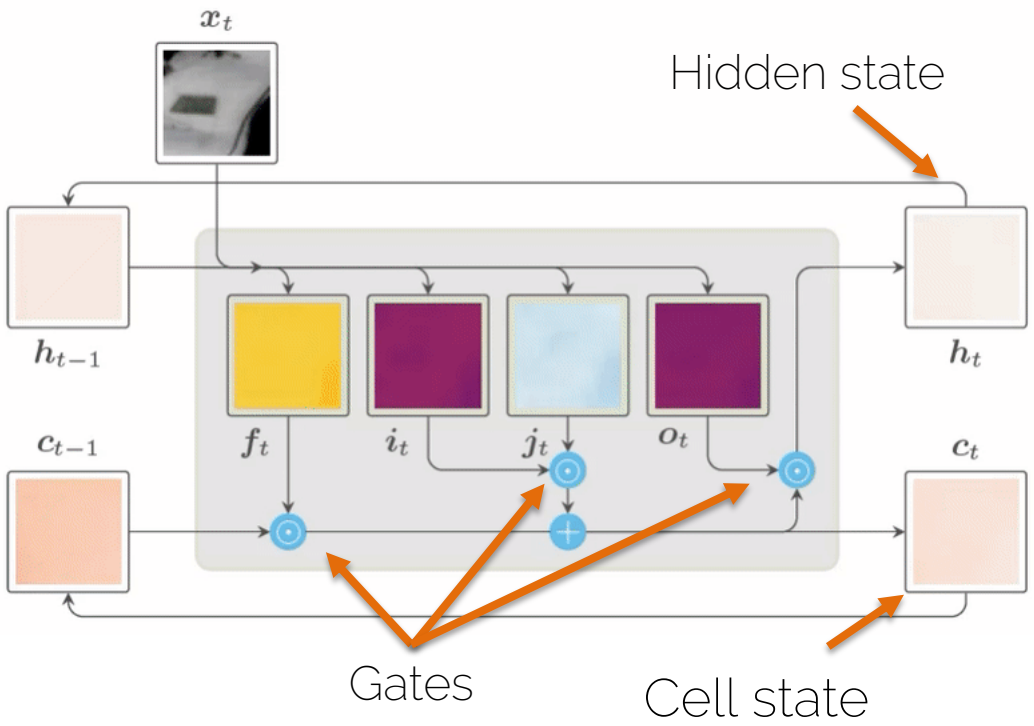
* Dimensions need to match

What operation do I need to do to my input to get a 128 vector representation?

General LSTM Units

- Input, states, and gates not limited to 1st-order tensors
- Gate functions can consist of FC and CNN layers

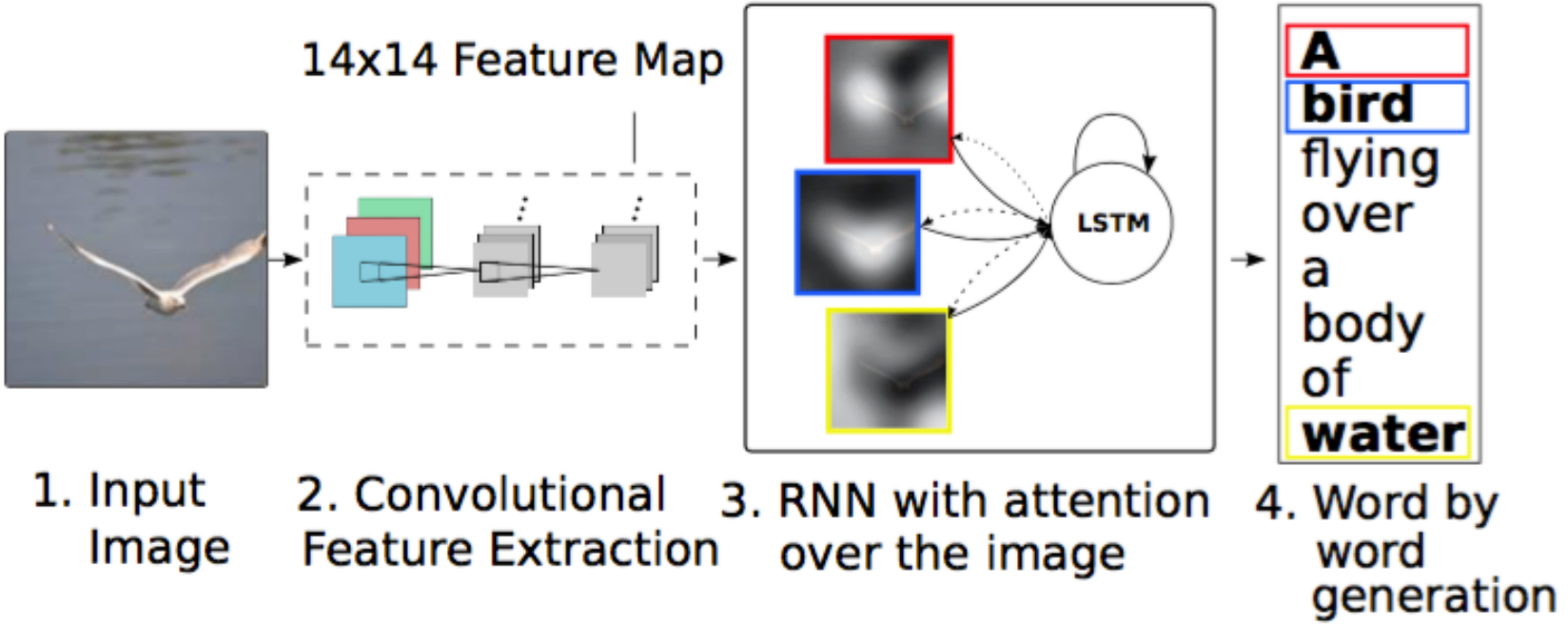
ConvLSTM for Video Sequences



- Input, hidden, and cell states are higher order tensors (i.e. images)
- Gates have CNN instead of FC layers

RNNs in Computer Vision

- Caption generation

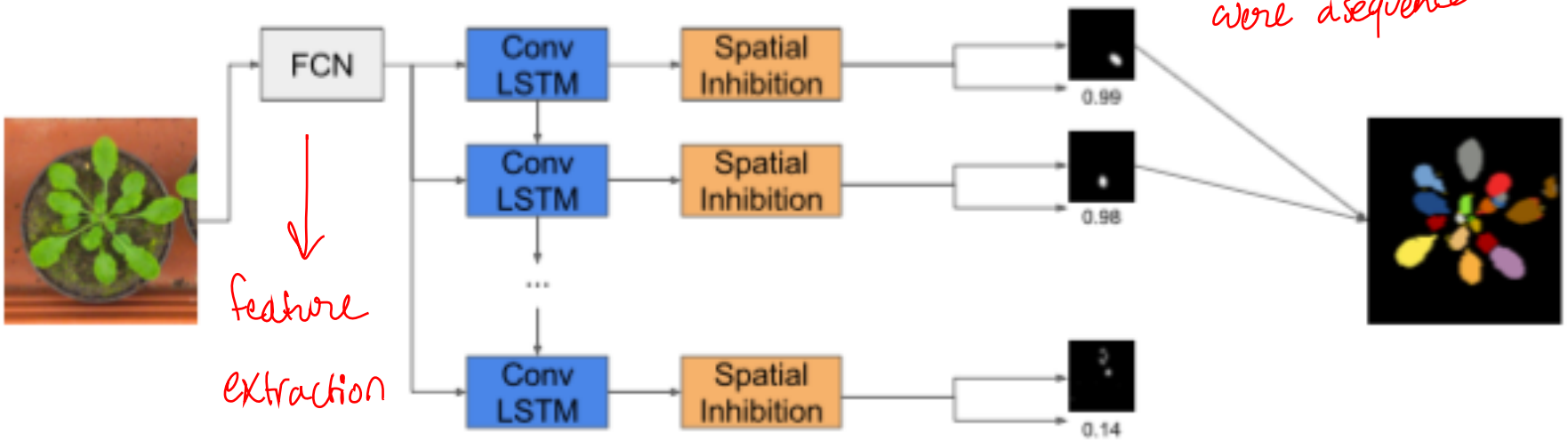


[Xu et al., PMLR'15] Neural Image Caption Generation

identifying diff instances of the same type of object

- Instance segmentation

predicting one leaf at a time as if it were a sequence



[Romera-Paredes et al., ECCV'16] Recurrent Instance Segmentation

Back prop through time

instead of backprop errors through a single feedforward net at a single time t, in RNN errors are backprop at each individual t, aggregated across all time steps.

i.e from where it currently is all the way to the beginning of the seq.