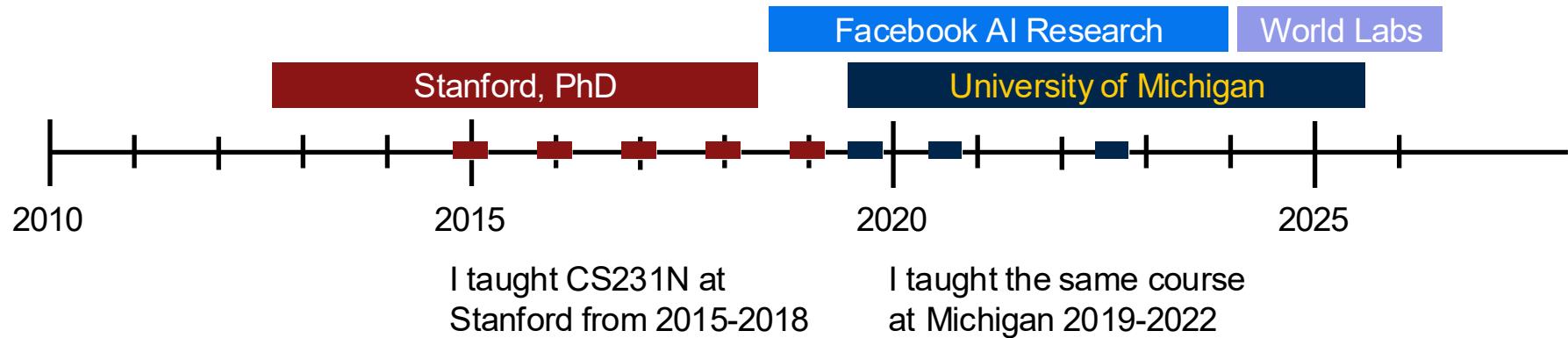
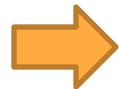


Lecture 5: Image Classification with CNNs

Hi I'm Justin

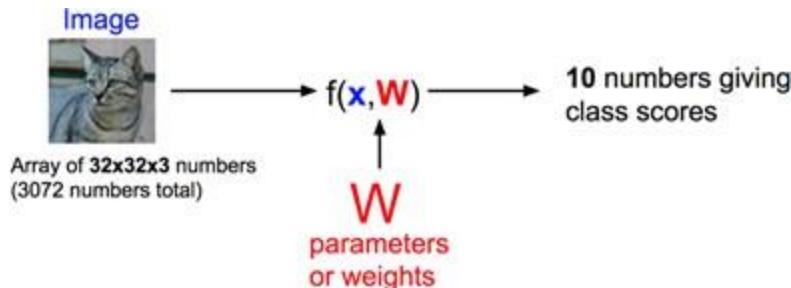


XCS231n: Deep Learning for Computer Vision



- Deep Learning Basics (Lecture 2–4)
- Perceiving and Understanding the Visual World (Lecture 5–12)
- Generative and Interactive Visual Intelligence (Lecture 13–16)
- Human-Centered Applications and Implications (Lecture 17–18)

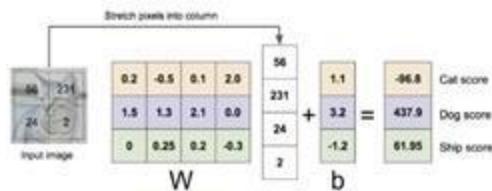
Recap: Image Classification with Linear Classifier



$$f(x, W) = Wx + b$$

Algebraic Viewpoint

$$f(x, W) = Wx$$



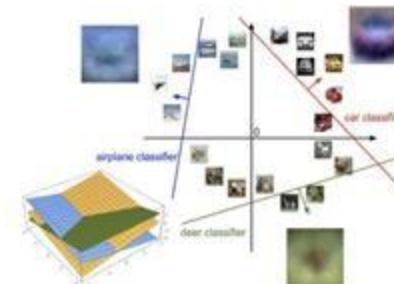
Visual Viewpoint

One template per class



Geometric Viewpoint

Hyperplanes cutting up space



Recap: Loss Function

- We have some dataset of (x, y)
- We have a score function:
- We have a loss function:

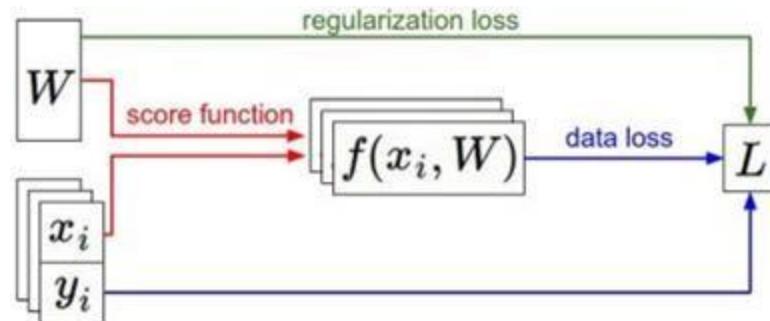
$$s = f(x; W) = Wx$$

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{sj}}\right)$$

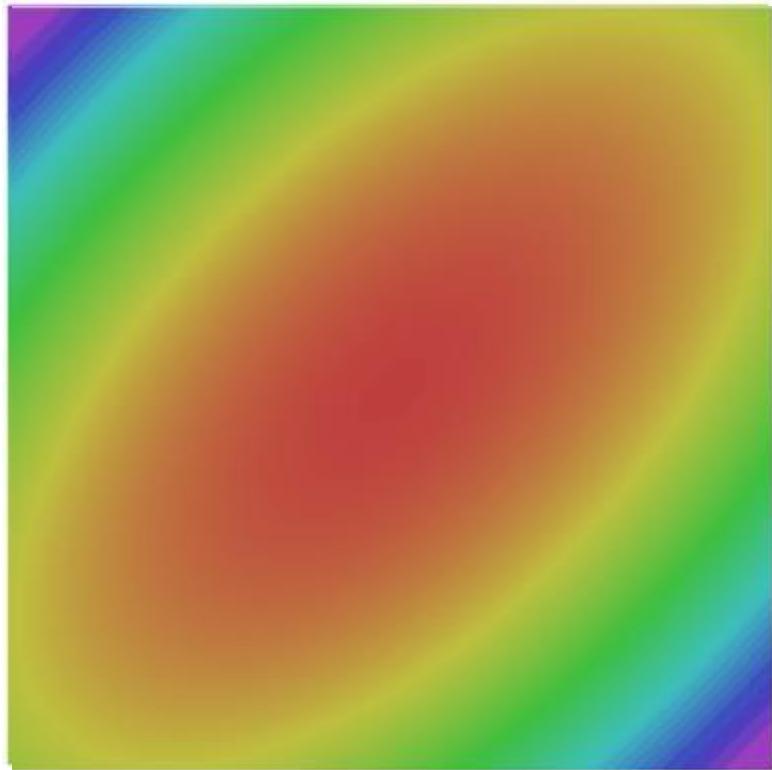
Softmax

$$L = \frac{1}{N} \sum_{i=1}^N L_i + R(W)$$

Full loss



Recap: Optimization



- SGD
- SGD+Momentum
- RMSProp
- Adam

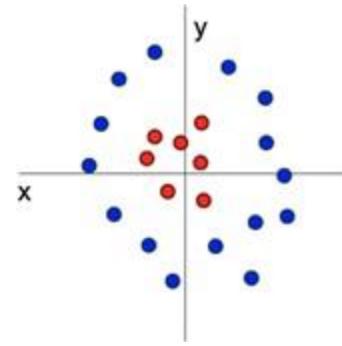
Problem: Linear Classifiers are not very powerful

Visual Viewpoint



Linear classifiers learn
one template per class

Geometric Viewpoint



Linear classifiers can
only draw linear
decision boundaries

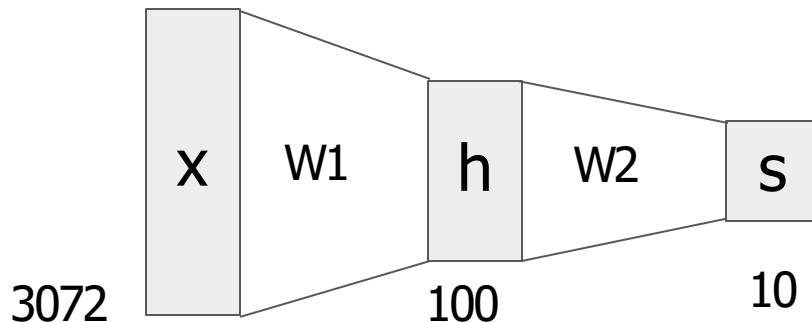
Last time: Neural Networks

Linear score function:

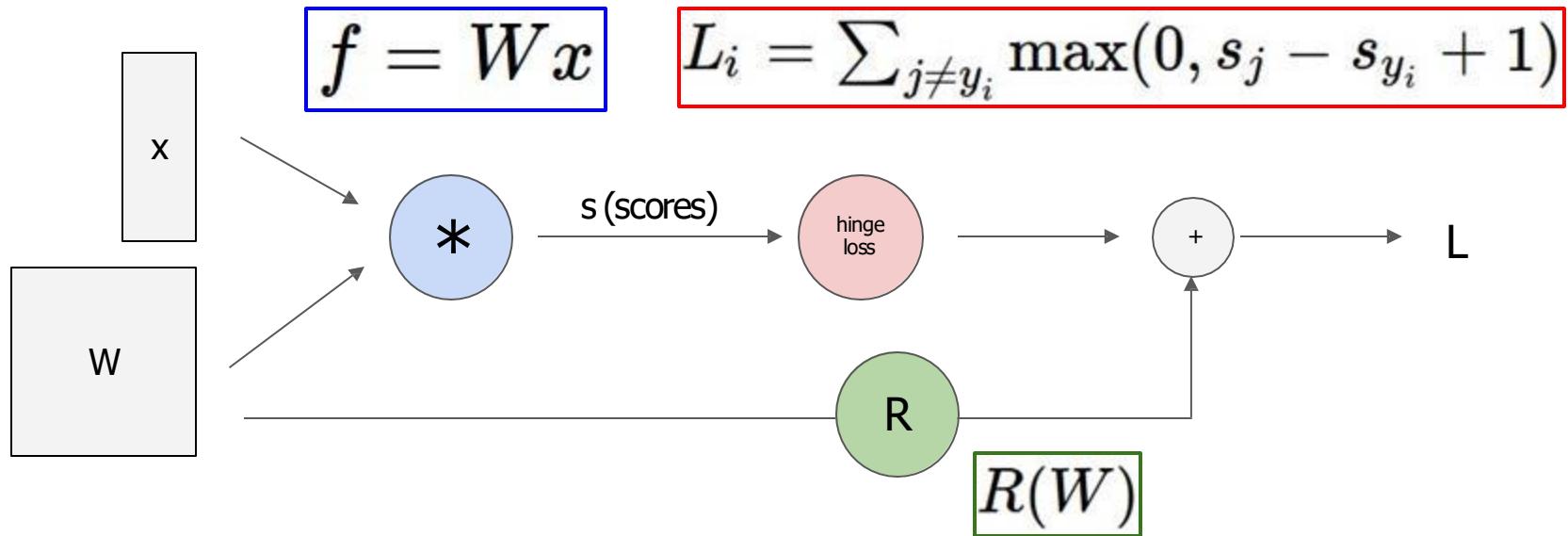
$$f = Wx$$

2-layer Neural Network

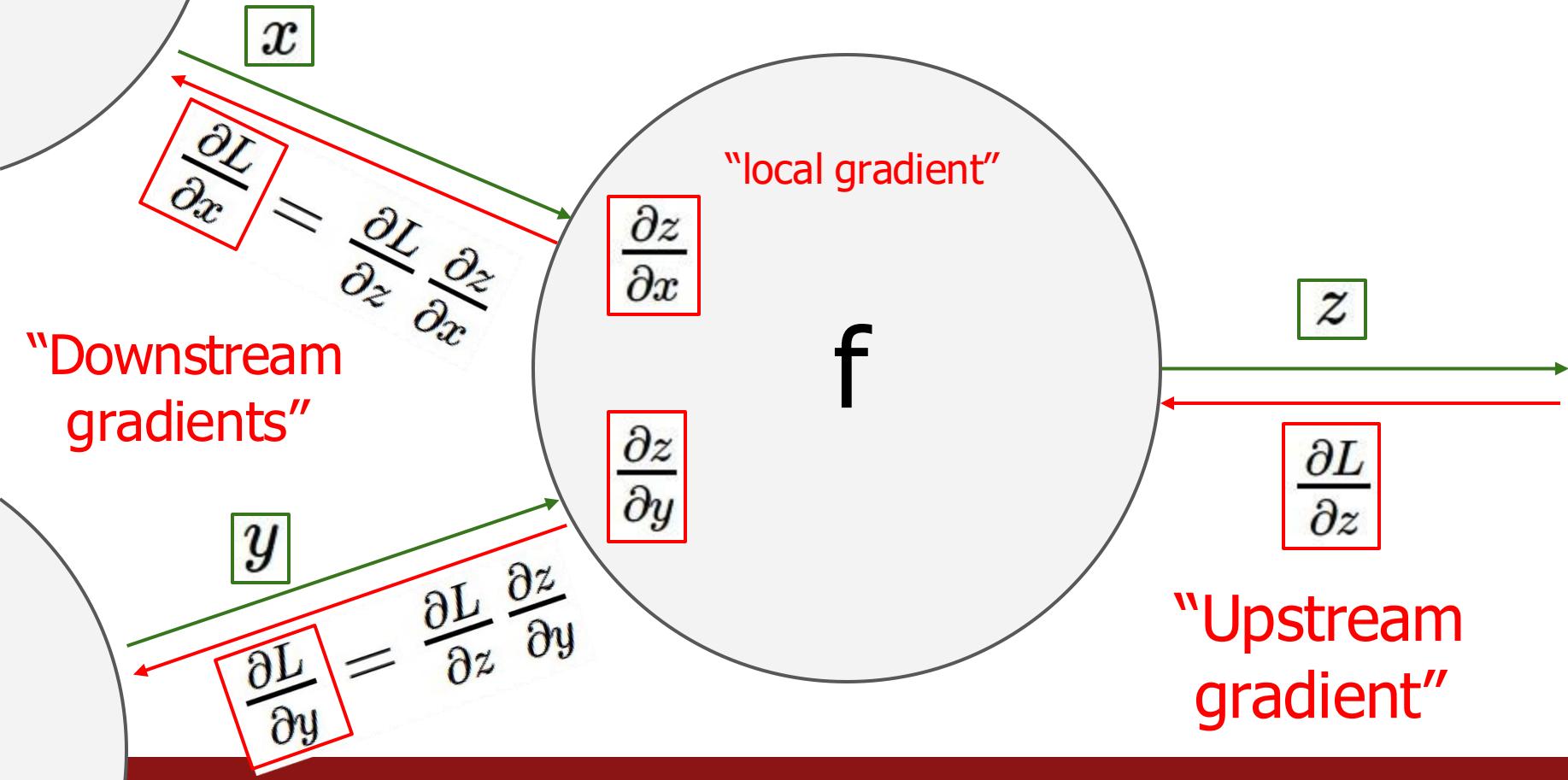
$$f = W_2 \max(0, W_1 x)$$



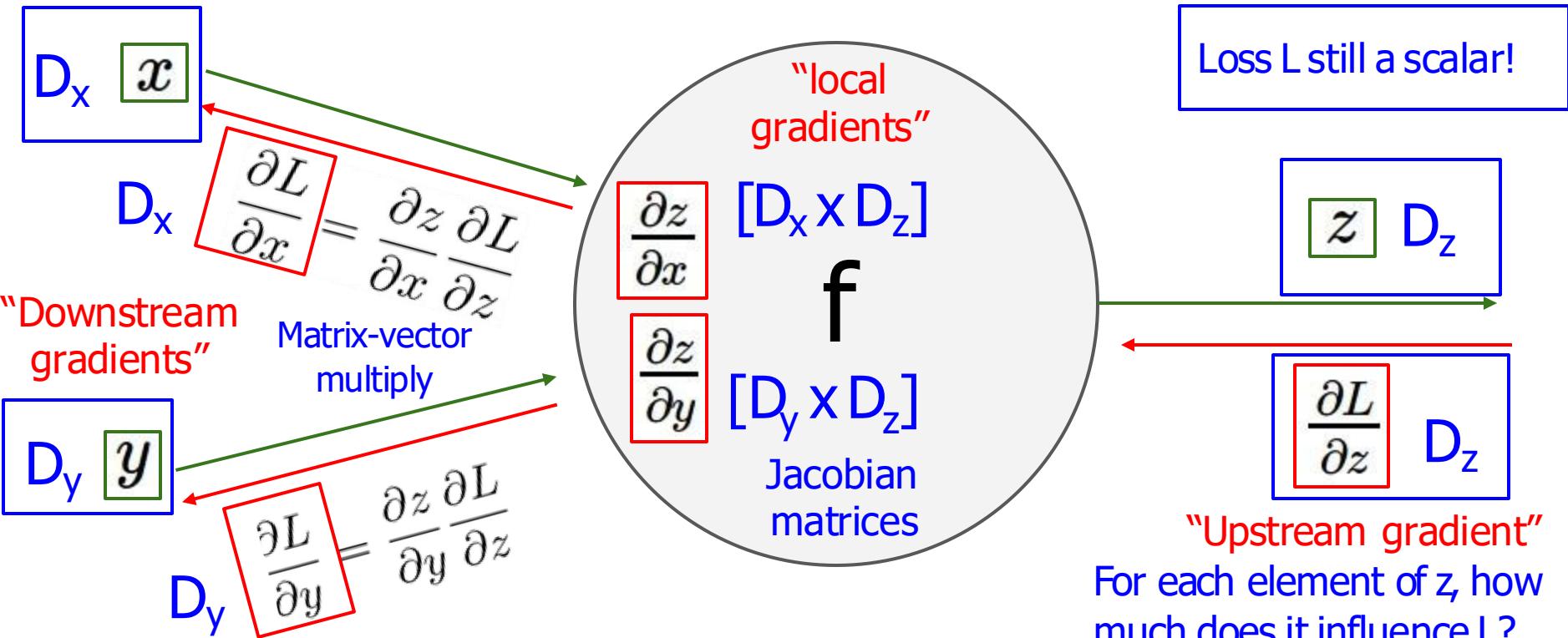
Last time: Computation Graph



Last time: Backpropagation

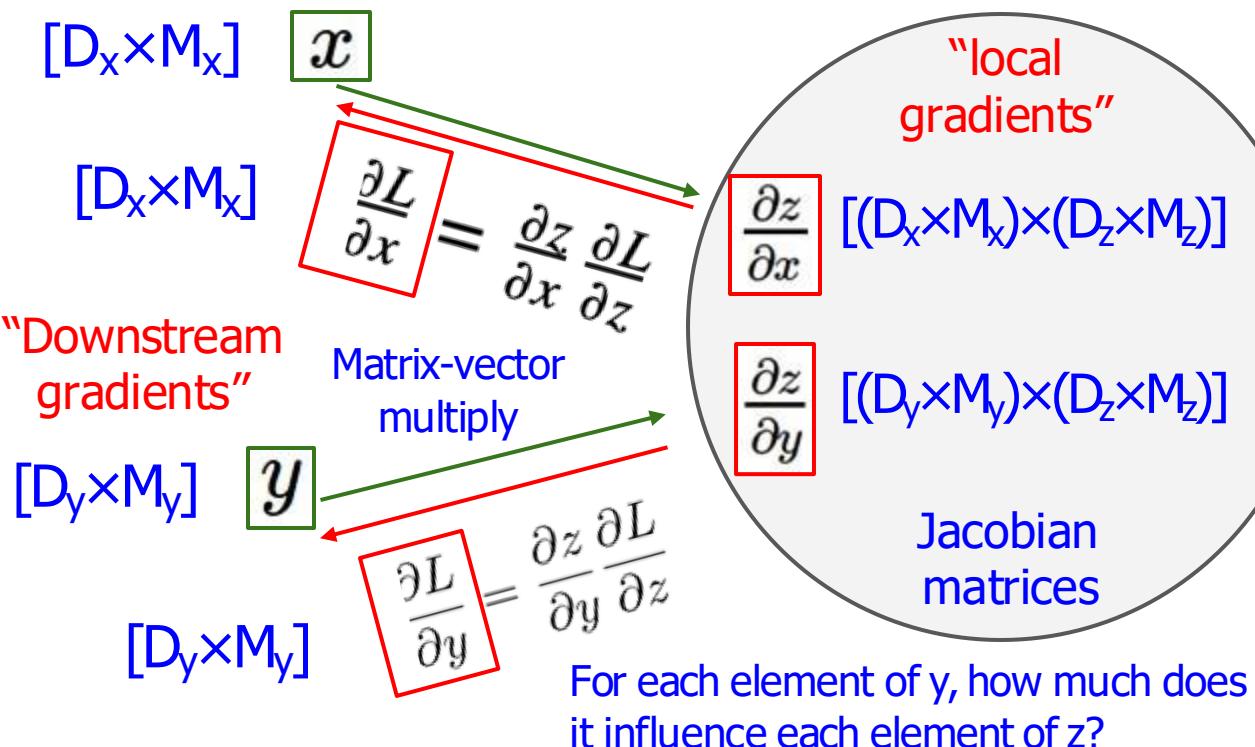


Backprop with Vectors



Backprop with Matrices (or Tensors)

Loss L still a scalar!



dL/dx always has the same shape as x !

"Upstream gradient"
For each element of z , how much does it influence L ?

CS231n: Deep Learning for Computer Vision



- Deep Learning Basics (Lecture 2–4)
- Perceiving and Understanding the Visual World (Lecture 5–12)
- Generative and Interactive Visual Intelligence (Lecture 13–16)
- Human-Centered Applications and Implications (Lecture 17–18)

CS231n: Deep Learning for Computer Vision

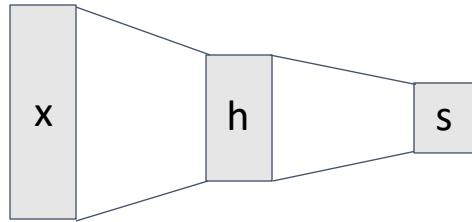
- Deep Learning Basics (Lecture 2–4)
- Perceiving and Understanding the Visual World (Lecture 5–12)

- Generative and Interactive Visual Intelligence (Lecture 13–16)
- Human-Centered Applications and Implications (Lecture 17–18)

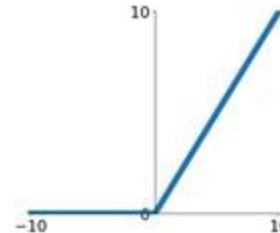
Today: Convolutional Networks

We have
already
seen these

Fully-Connected Layer



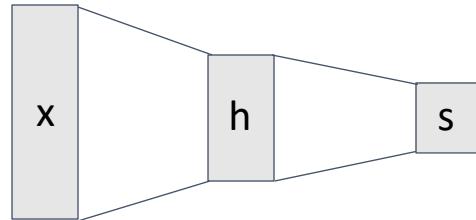
Activation Function



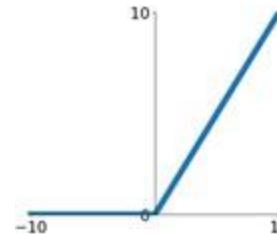
Today: Convolutional Networks

We have
already
seen these

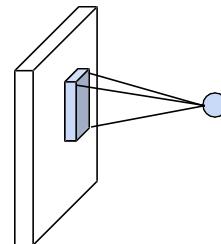
Fully-Connected Layer



Activation Function



Convolution Layer



Today: Image-specific operators

Pooling Layer

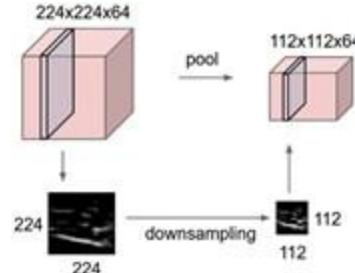


Image Classification: A core task in Computer Vision



(assume given a set of labels)
{dog, cat, truck, plane, ...}



cat
dog
bird
deer
truck

This image by [Nikita](#) is
licensed under [CC-BY 2.0](#).

Pixel space

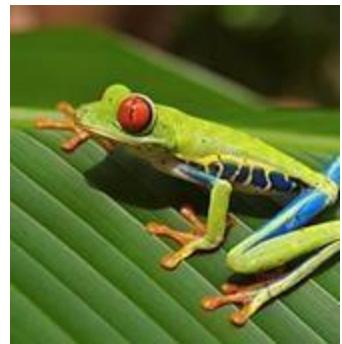


$$f(x) = Wx$$

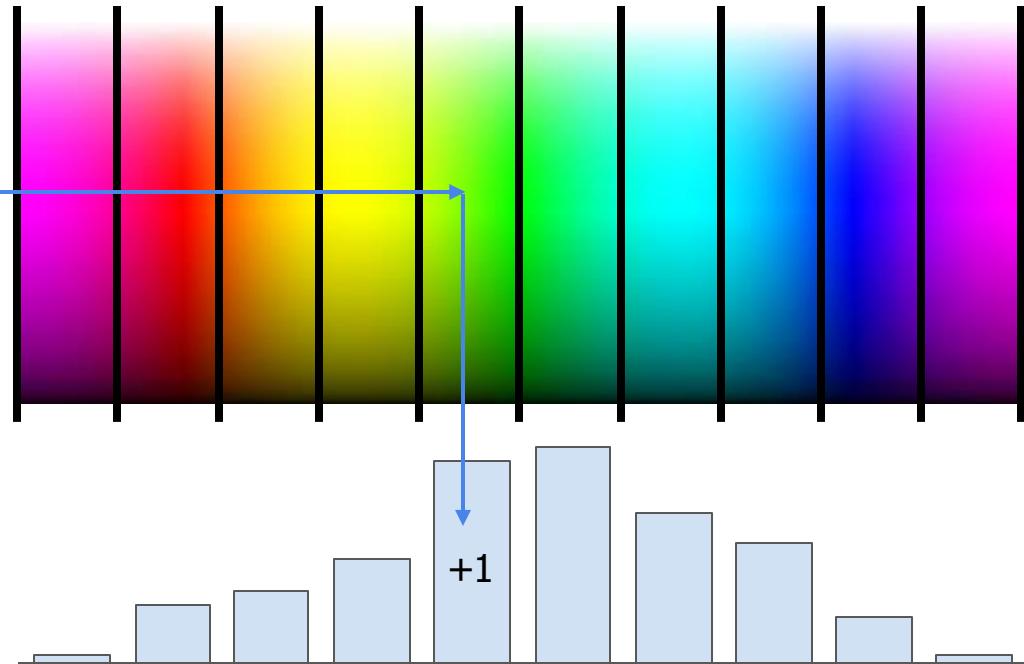
Class scores



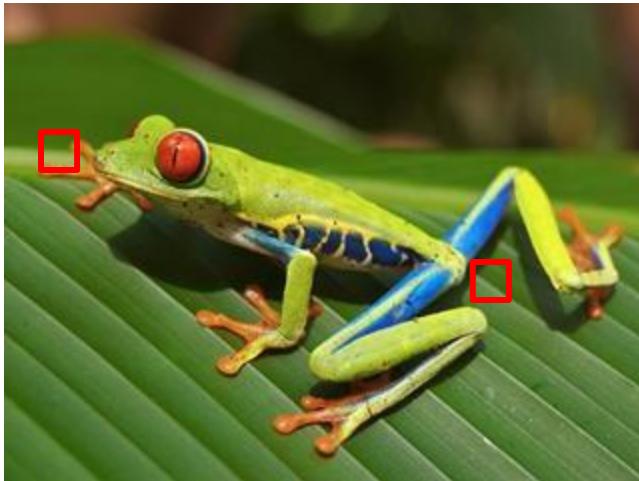
Image features



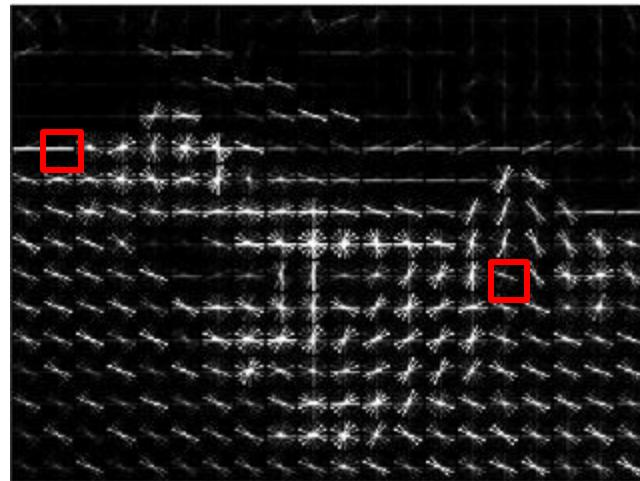
Example: Color Histogram



Example: Histogram of Oriented Gradients (HoG)



Divide image into 8x8 pixel regions
Within each region quantize edge
direction into 9 bins



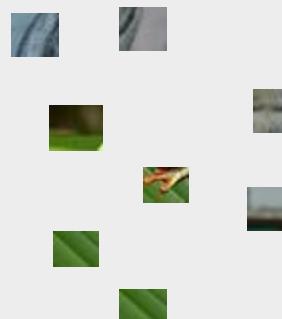
Example: 320x240 image gets divided
into 40x30 bins; in each bin there are 9
numbers so feature vector has $30 \times 40 \times 9 =$
10,800 numbers

Example: Bag of Words

Step 1: Build codebook



Extract random patches



Cluster patches to form “codebook” of “visual words”



Step 2: Encode images

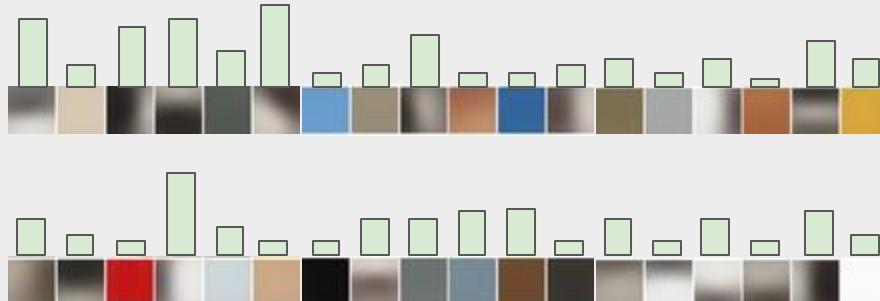


Image Features

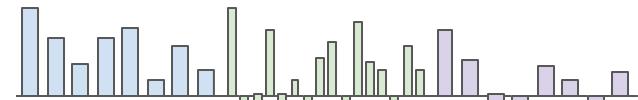
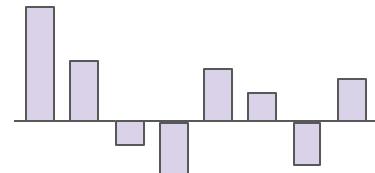
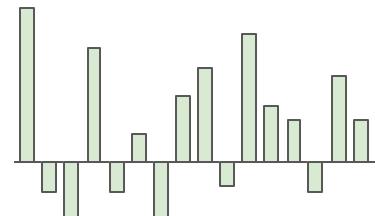
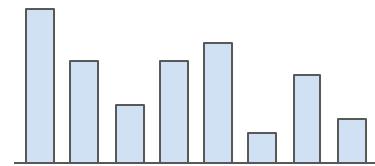
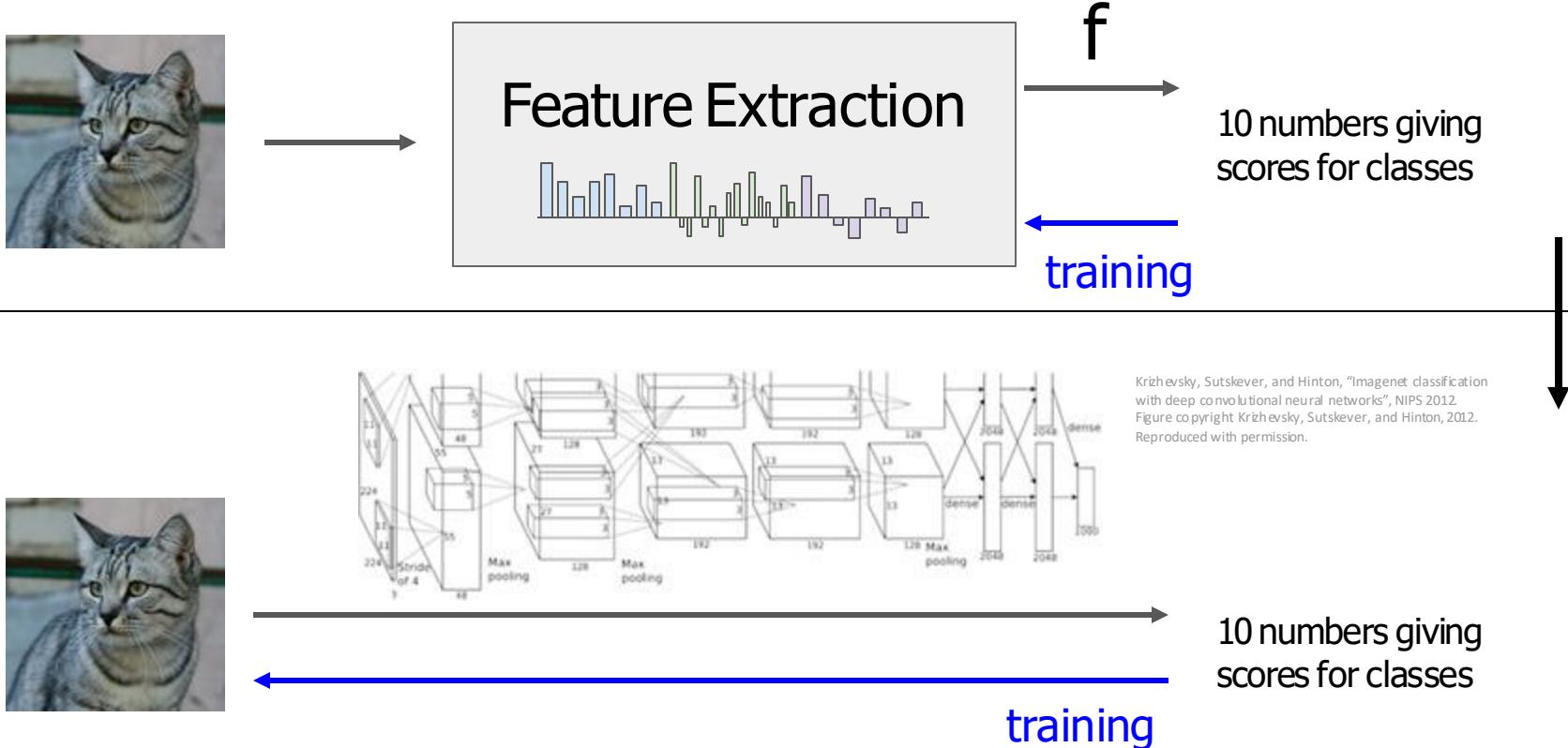


Image features vs. ConvNets



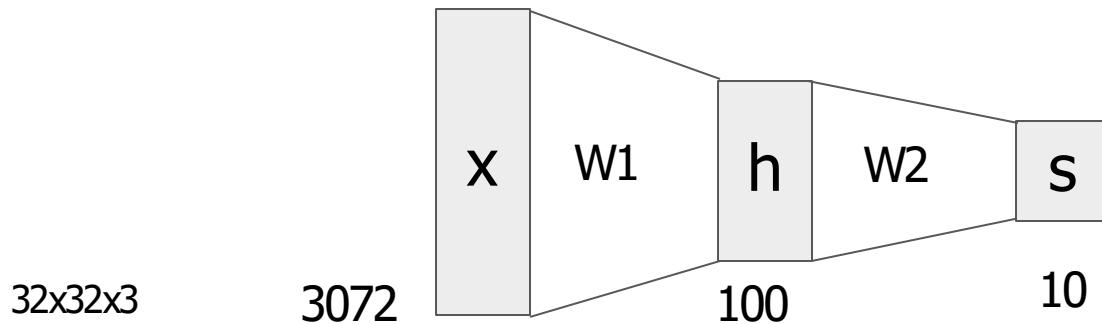
Last Time: Neural Networks

Linear score function:

$$f = Wx$$

2-layer Neural Network

$$f = W_2 \max(0, W_1 x)$$



Last Time: Neural Networks

Linear score function:

$$f = Wx$$

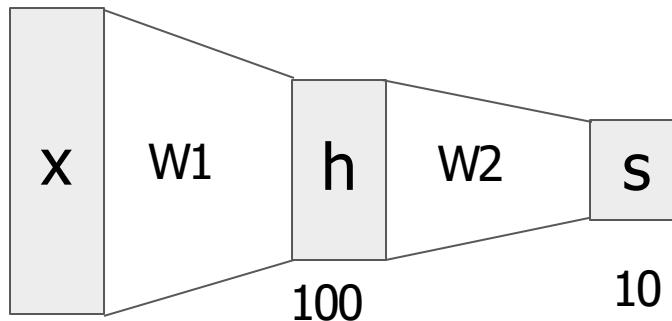
2-layer Neural Network

$$f = W_2 \max(0, W_1 x)$$

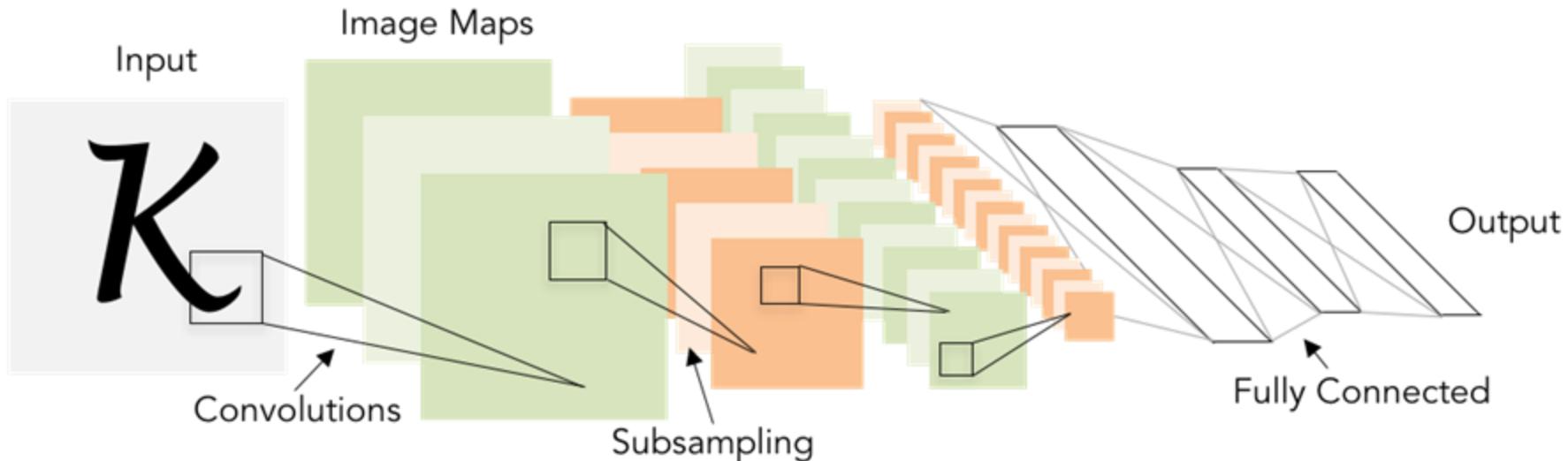
Problem: The spatial
structure of images
is destroyed!

32x32x3

3072

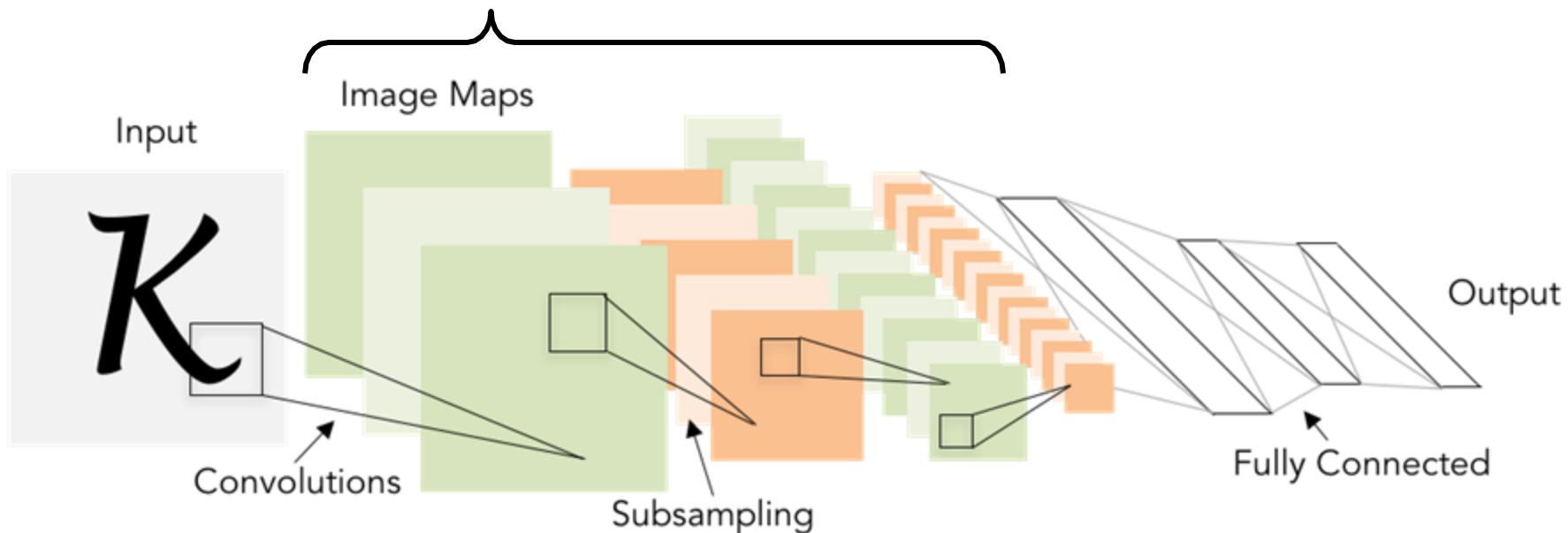


Next: Convolutional Neural Networks



Next: Convolutional Neural Networks

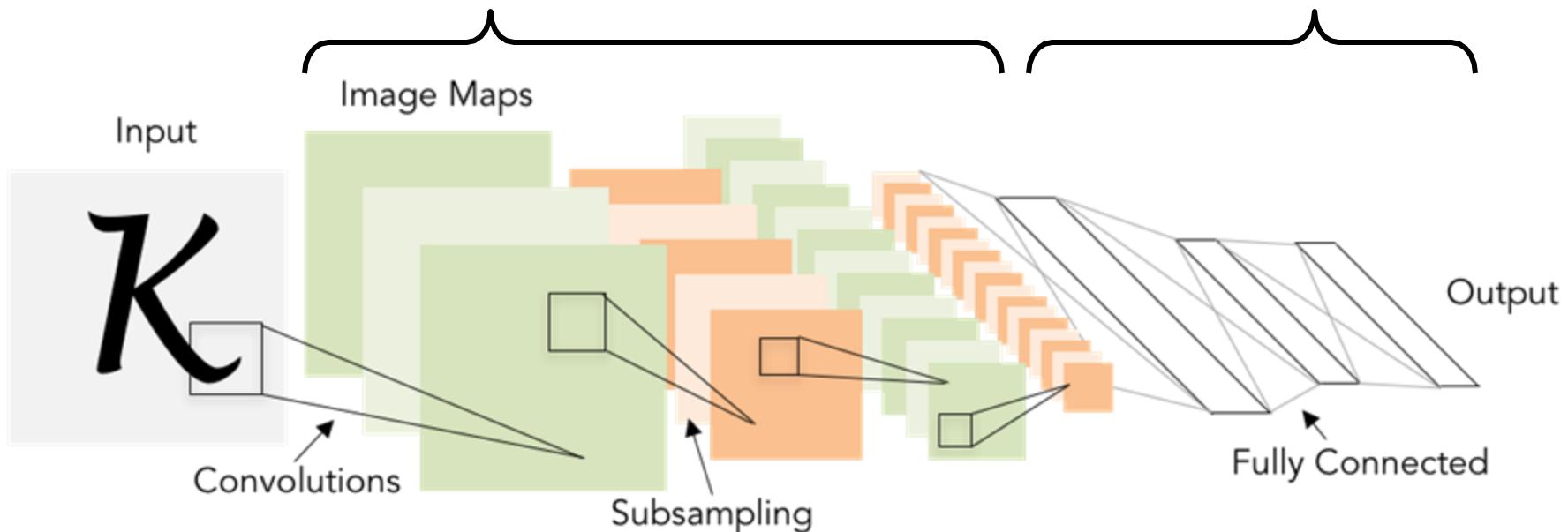
Convolution and **pooling** operators extract features while respecting 2D image structure



Next: Convolutional Neural Networks

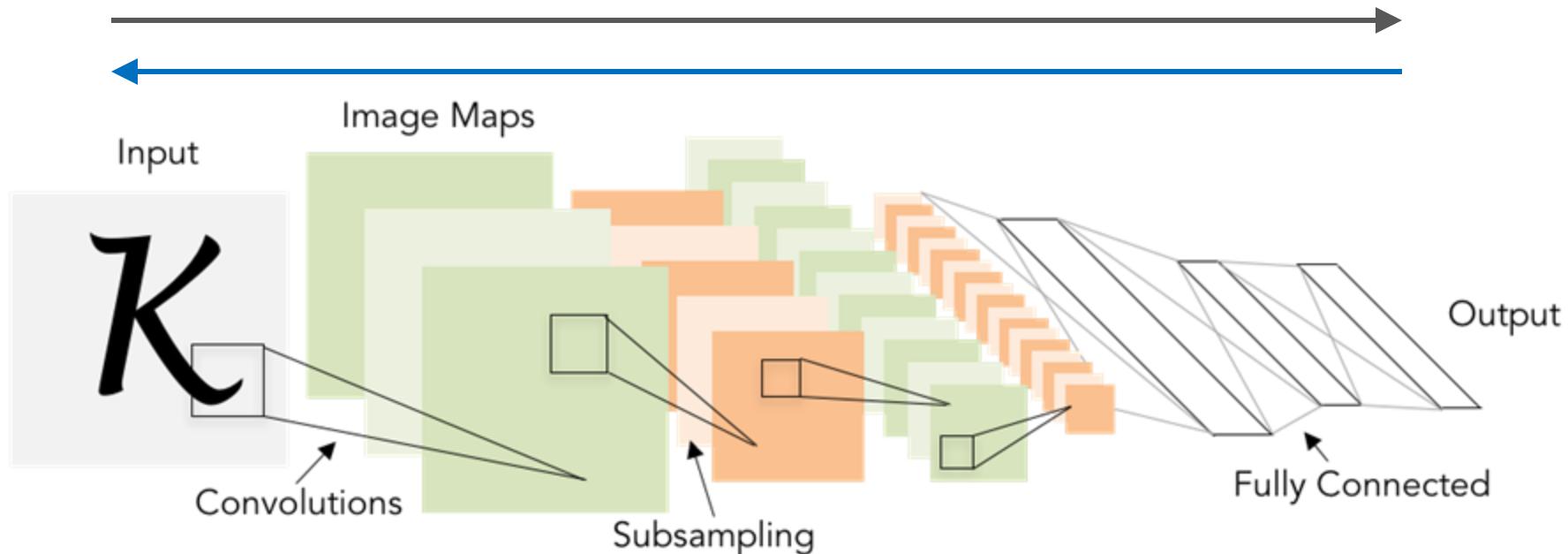
Convolution and **pooling** operators extract features while respecting 2D image structure

Fully-Connected layers form an MLP at the end to predict scores



Next: Convolutional Neural Networks

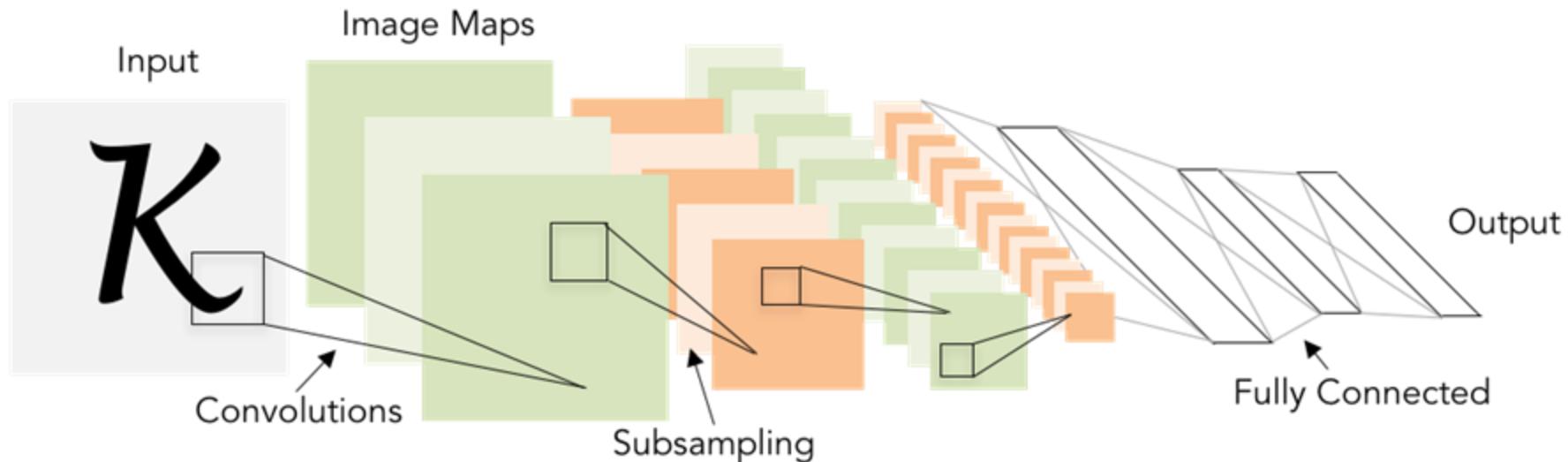
Trained end-to-end with backprop + gradient descent



A bit of history:

Gradient-based learning applied to
document recognition

[LeCun, Bottou, Bengio, Haffner 1998]



A bit of history:

ImageNet Classification with Deep Convolutional Neural Networks

[Krizhevsky, Sutskever, Hinton, 2012]

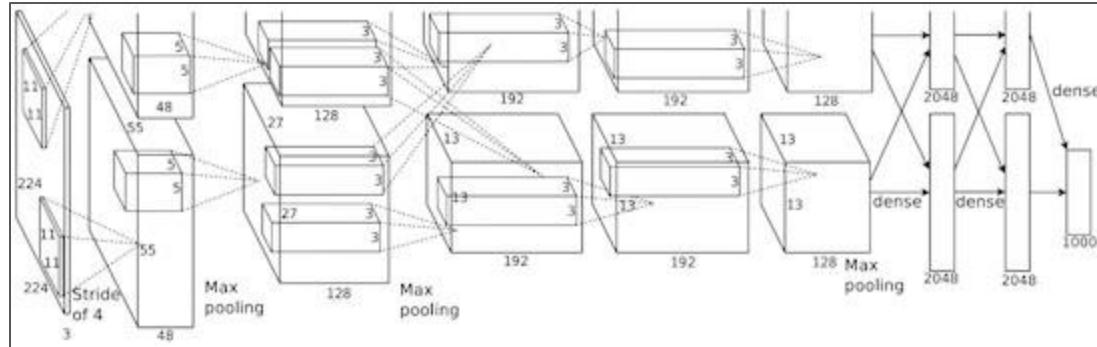
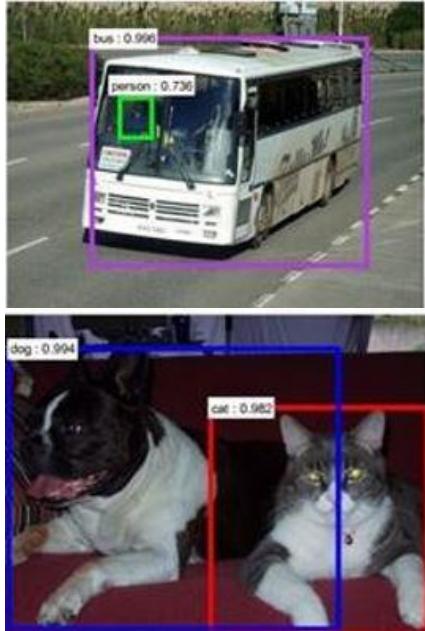


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

“AlexNet”

~2012 – 2020: ConvNets dominate all vision tasks

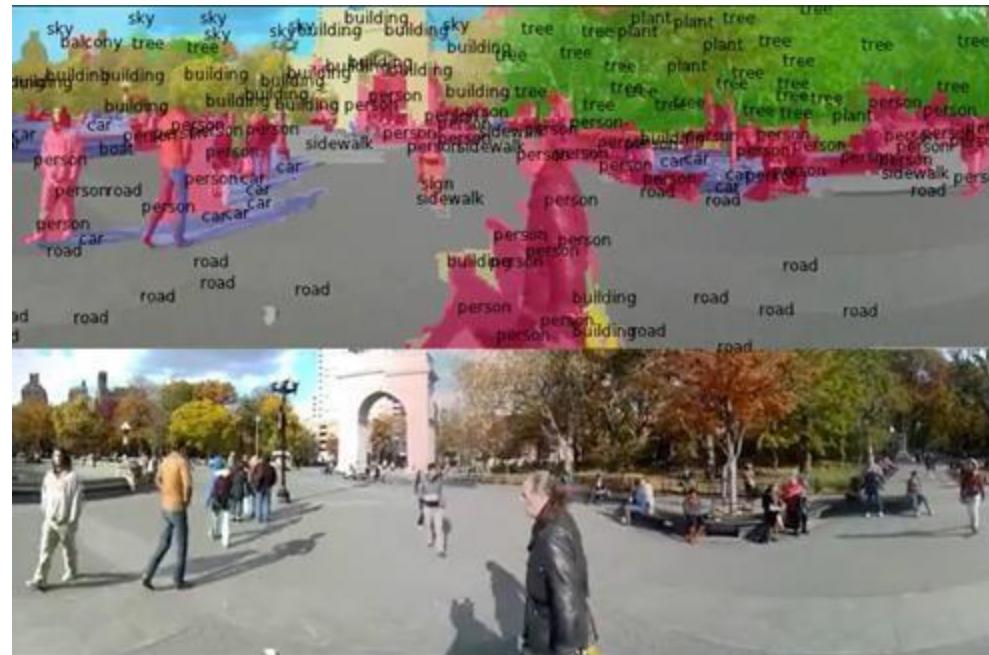
Detection



Figures copyright Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun, 2015. Reproduced with permission.
Images adapted from above source

[Faster R-CNN: Ren, He, Girshick, Sun 2015]

Segmentation



Figures copyright Clement Farabet, 2012. Reproduced with permission.

[Farabet et al., 2012]

~2012 – 2020: ConvNets dominate all vision tasks

Image Captioning



A white teddy bear sitting in the grass



A man in a baseball uniform throwing a ball



A woman is holding a cat in her hand



A man riding a wave on top of a surfboard



A cat sitting on a suitcase on the floor



A woman standing on a beach holding a surfboard

All images are CC0 Public domain:
<https://pixabay.com/en/luggage-antique-cat-1643010/>
<https://pixabay.com/en/teddy-plush-bears-cute-teddy-bear-1623436/>
<https://pixabay.com/en/surf-wave-summer-sport-litoral-1668716/>
<https://pixabay.com/en/woman-female-model-portrait-adult-983967/>
<https://pixabay.com/en/handstand-lake-meditation-496008/>
<https://pixabay.com/en/baseball-player-shortstop-infield-1045263/>

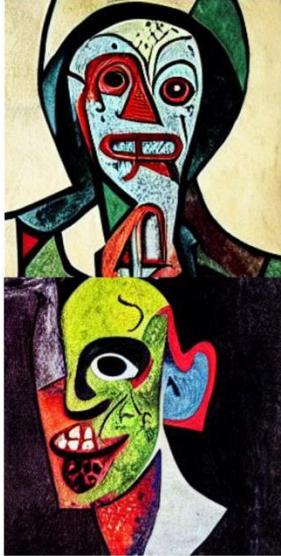
Captions generated by Justin Johnson using [Neuraltalk2](#)

[Vinyals et al., 2015]
[Karpathy and Fei-Fei, 2015]

~2012 – 2020: ConvNets dominate all vision tasks

Text-to-Image Generation

Rombach et al, "High-Resolution Image Synthesis with Latent Diffusion Models", CVPR 2022



A zombie in the style of Picasso



An image of a half mouse half octopus



A painting of a squirrel eating a burger



A watercolor painting of a chair that looks like an octopus



A shirt with the inscription: "I love generative models!"

~2012 – 2020: ConvNets dominate all vision tasks

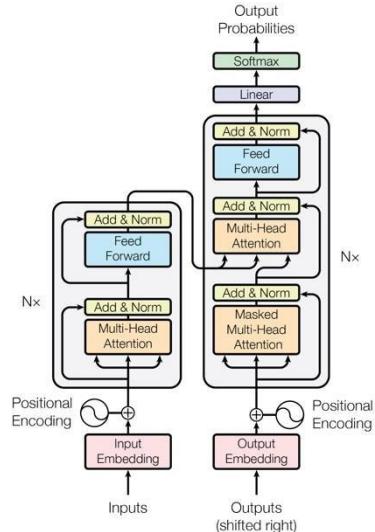
CS231n: Convolutional Neural Networks for Visual Recognition



This class used to be focused on ConvNets!

2021 - Present: Transformers have taken over

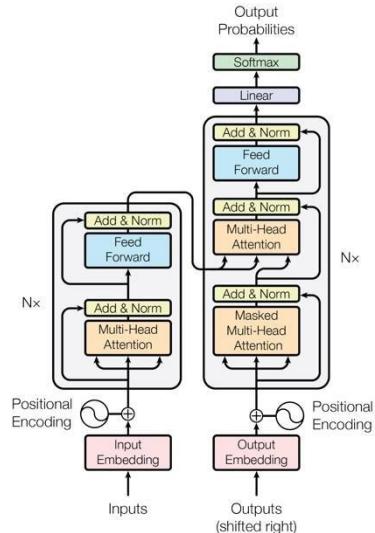
2017: Transformers
for language tasks



Vaswani et al, "Attention is
all you need", NeurIPS 2017

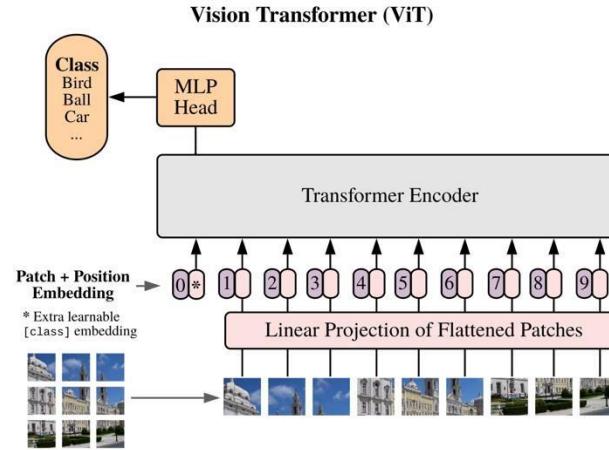
2021 - Present: Transformers have taken over

2017: Transformers
for language tasks

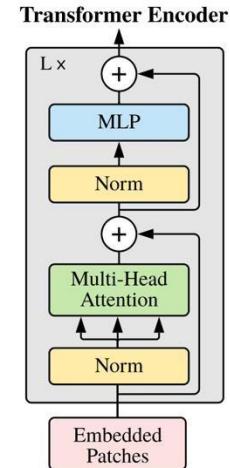


Vaswani et al, "Attention is all you need", NeurIPS 2017

2021: Transformers
for vision tasks

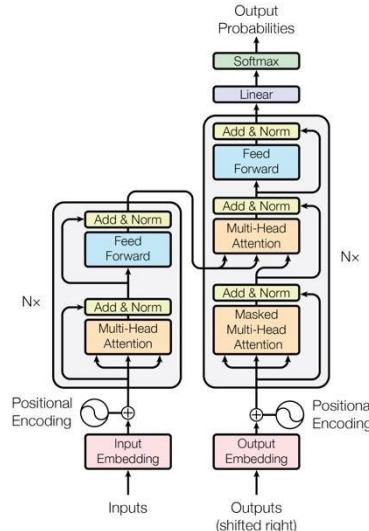


Dosovitskiy et al, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021



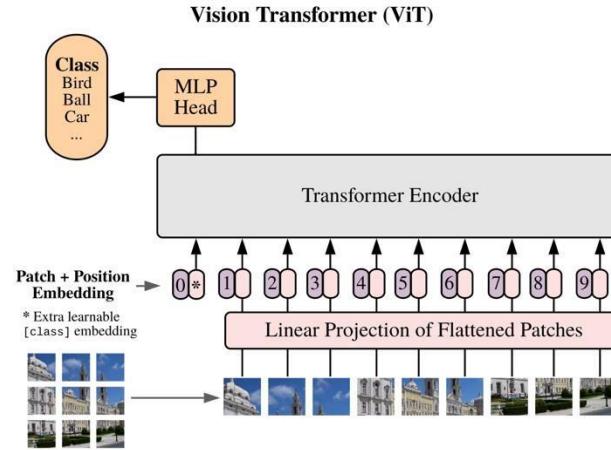
2021 - Present: Transformers have taken over

2017: Transformers
for language tasks



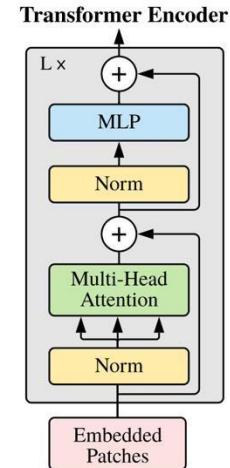
Vaswani et al, "Attention is all you need", NeurIPS 2017

2021: Transformers
for vision tasks



Dosovitskiy et al, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

Wait until
Lecture 8!

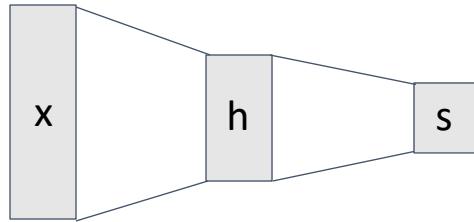


Convolutional Neural Networks

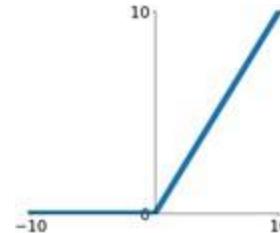
Today: Convolutional Networks

We have
already
seen these

Fully-Connected Layer



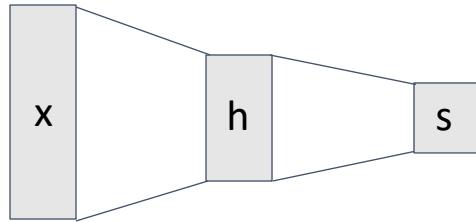
Activation Function



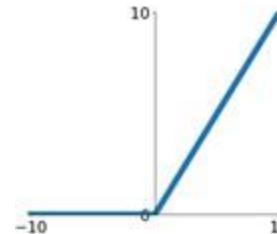
Today: Convolutional Networks

We have
already
seen these

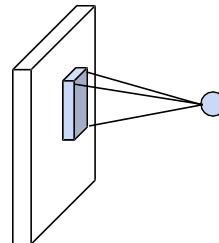
Fully-Connected Layer



Activation Function

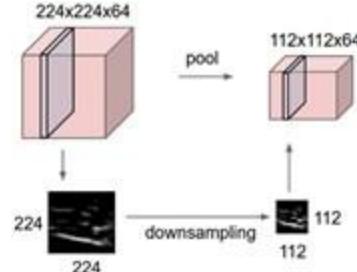


Convolution Layer



Today: Image-specific operators

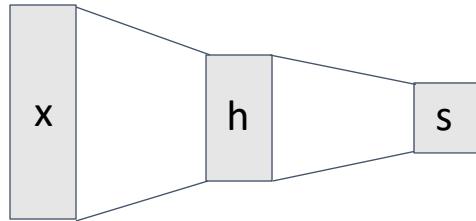
Pooling Layer



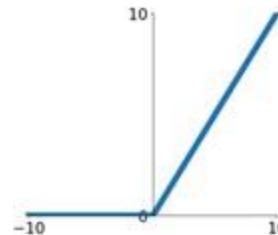
Today: Convolutional Networks

We have
already
seen these

Fully-Connected Layer

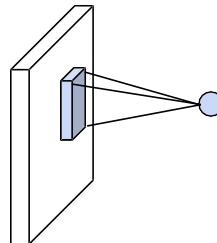


Activation Function

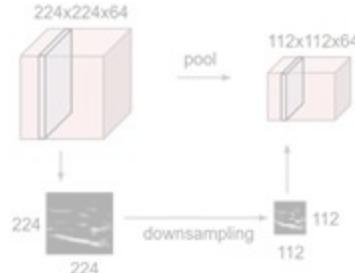


Convolution Layer

Today: Image-specific operators

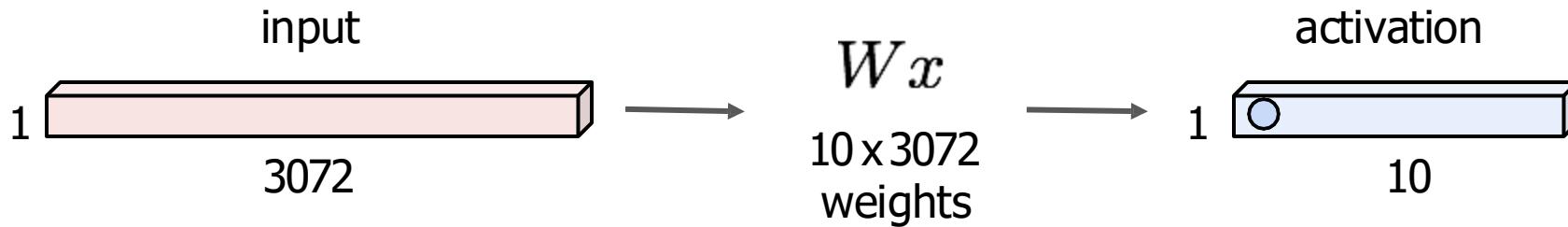


Pooling Layer



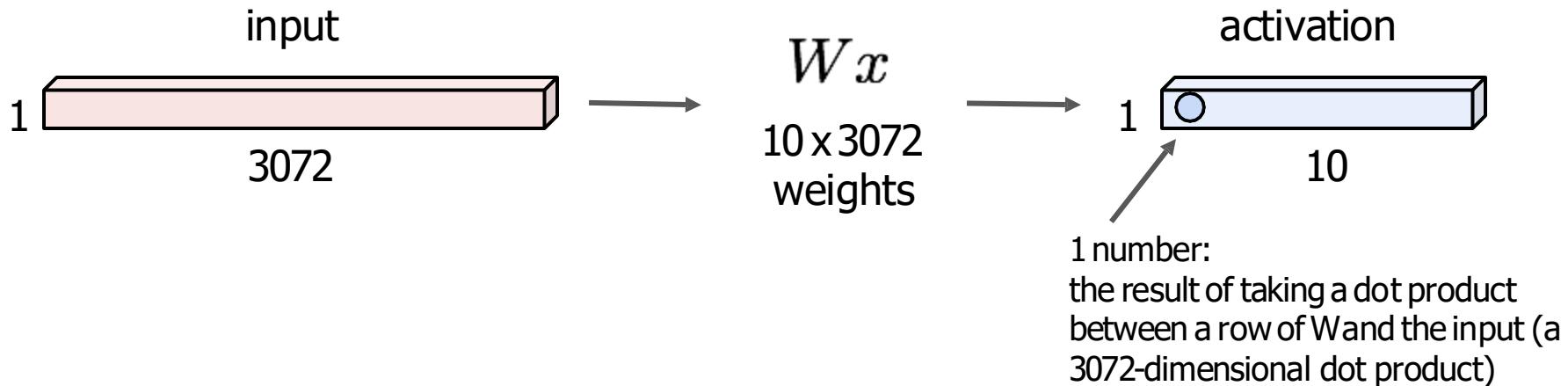
Recap: Fully Connected Layer

32x32x3 image ->stretch to 3072 x 1



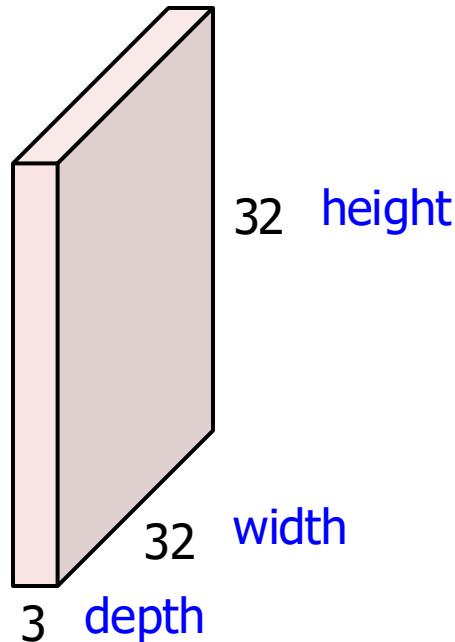
Fully Connected Layer

32x32x3 image ->stretch to 3072×1



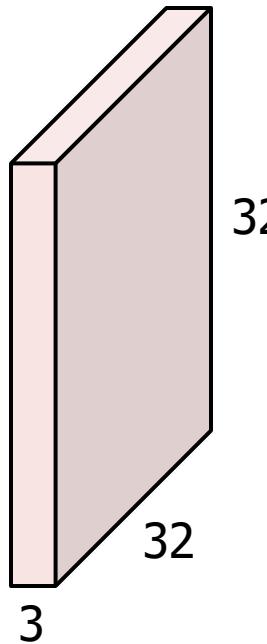
Convolution Layer

32x32x3 image ->preserve spatial structure



Convolution Layer

32x32x3 image



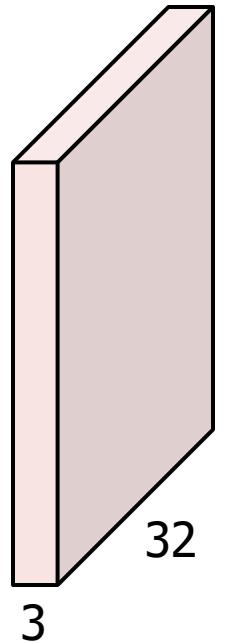
5x5x3 filter



Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

Convolution Layer

32x32x3 image



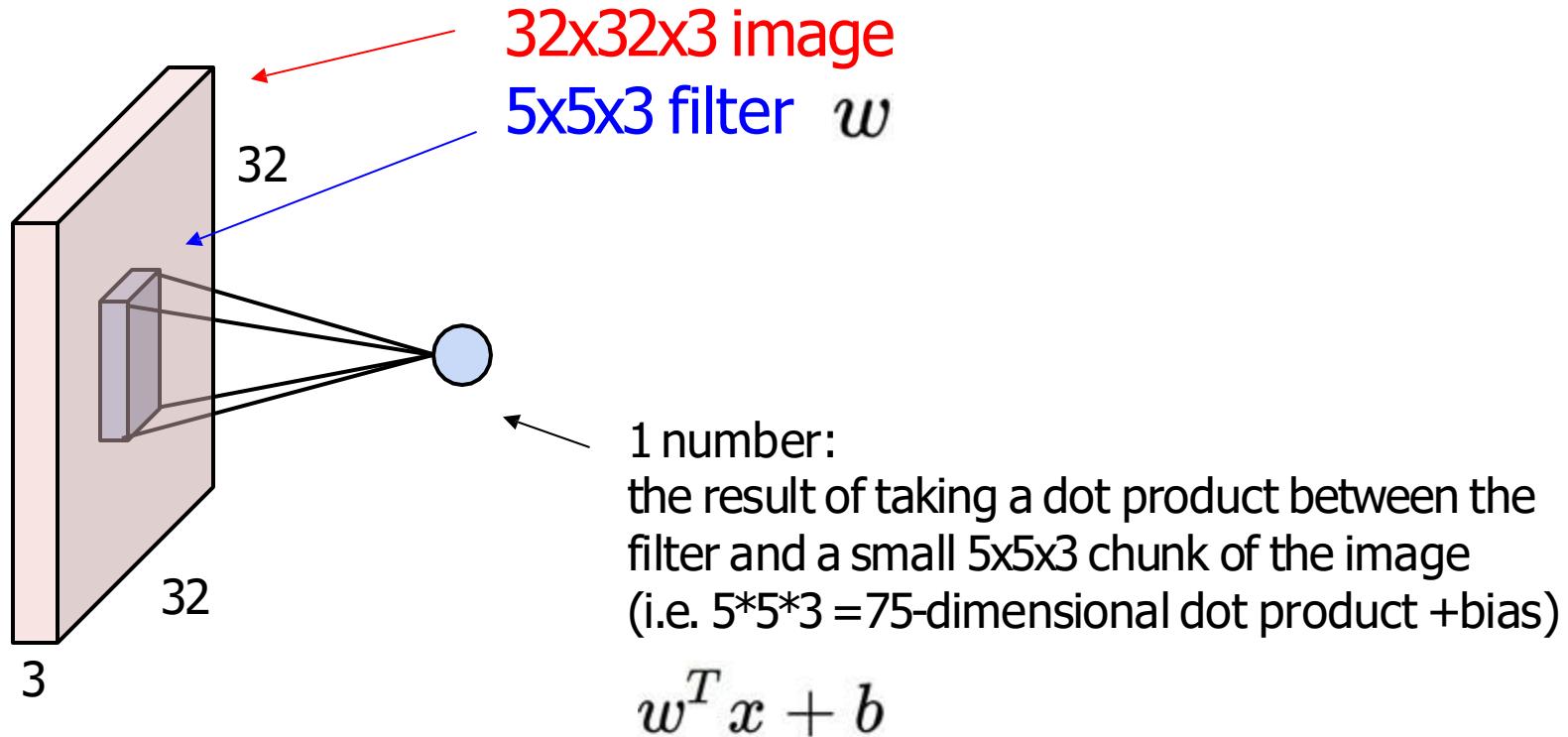
5x5x3 filter



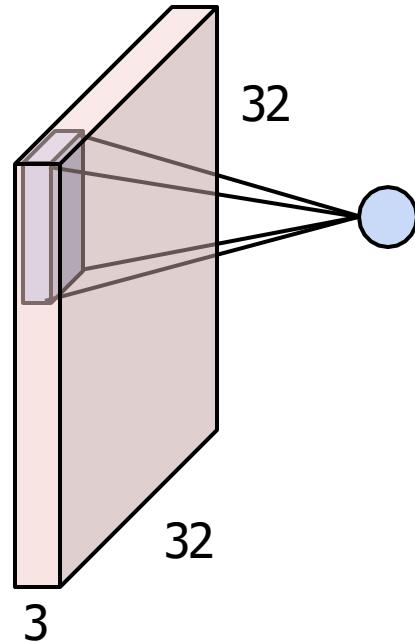
Filters always extend the full depth of the input volume

Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

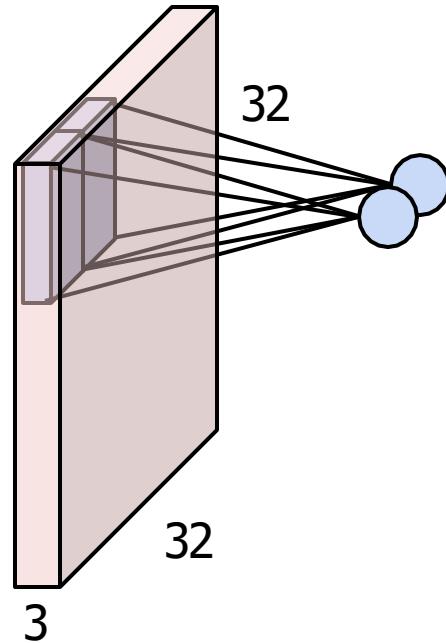
Convolution Layer



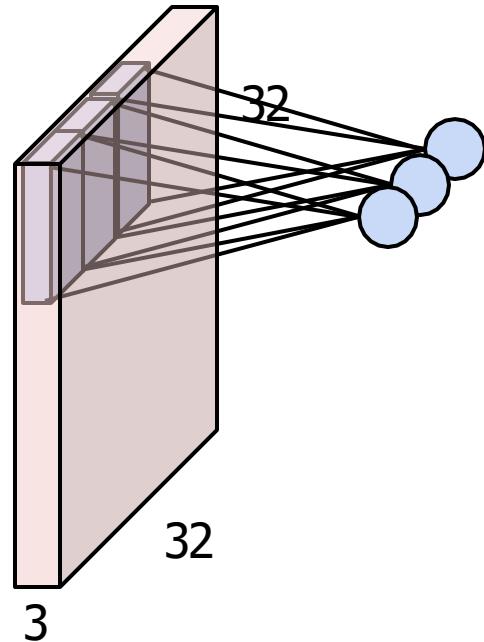
Convolution Layer



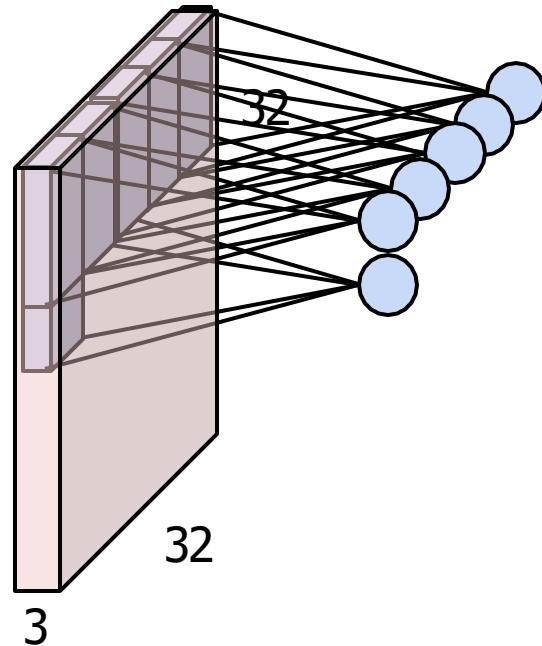
Convolution Layer



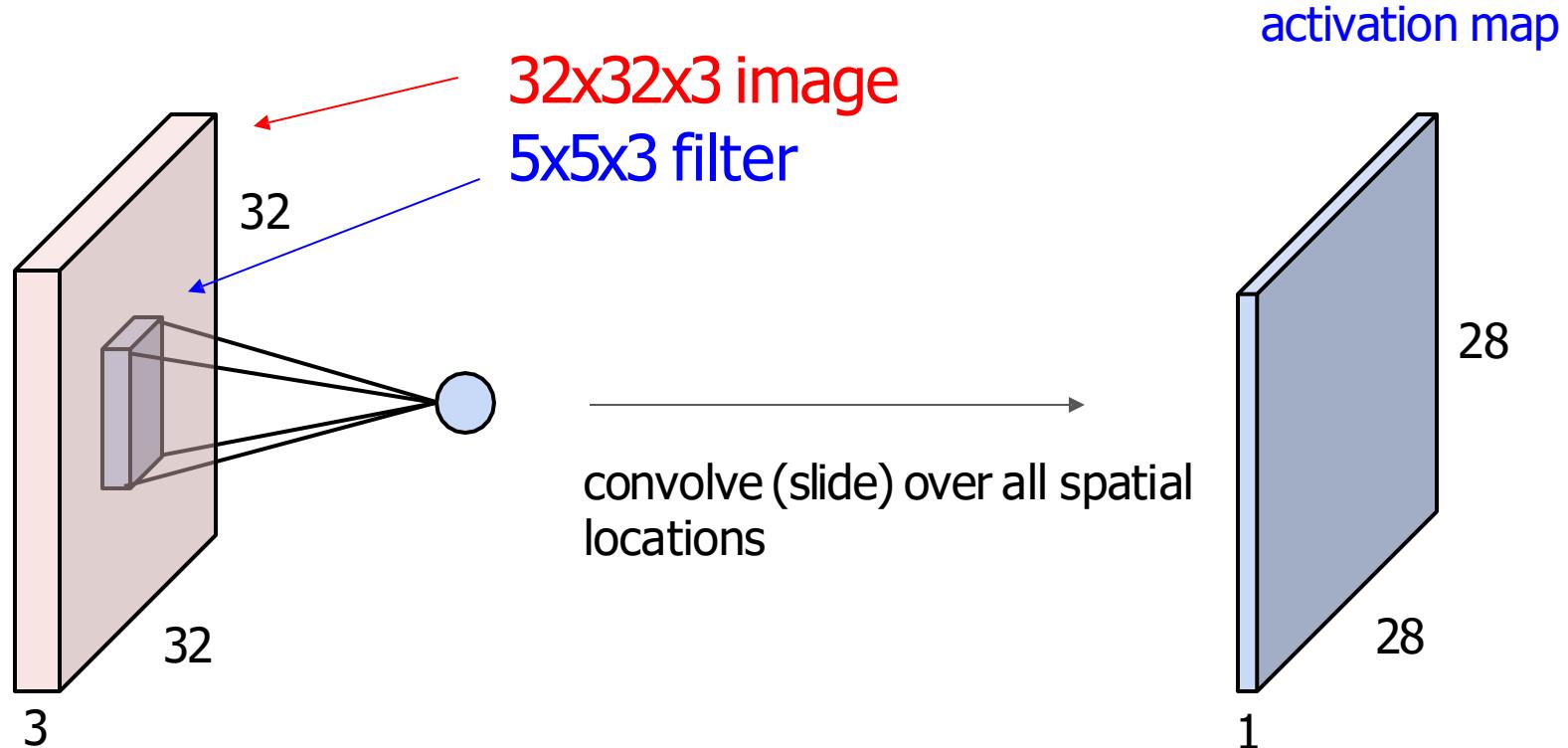
Convolution Layer



Convolution Layer

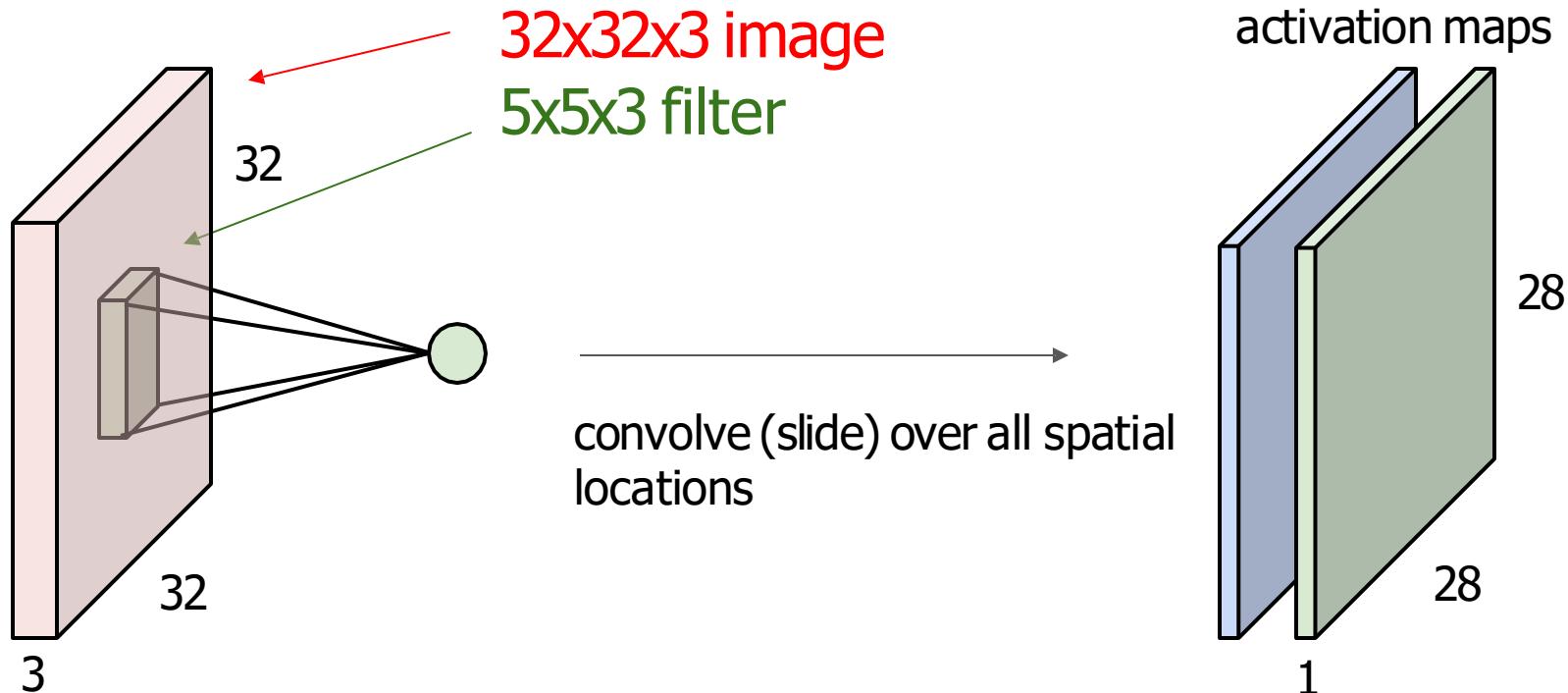


Convolution Layer



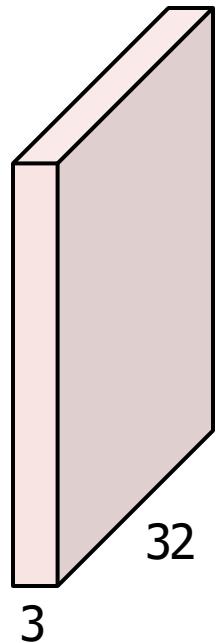
Convolution Layer

consider a second, green filter

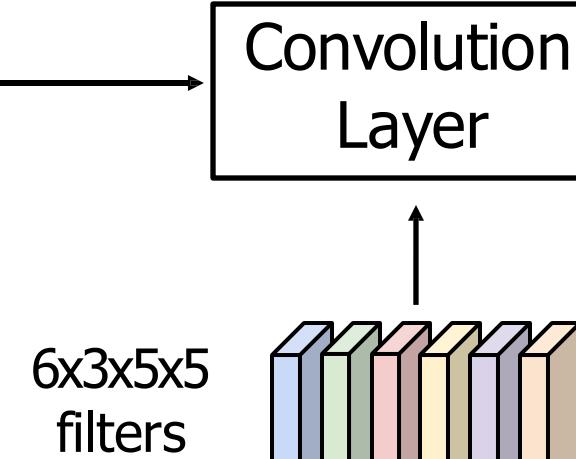


Convolution Layer

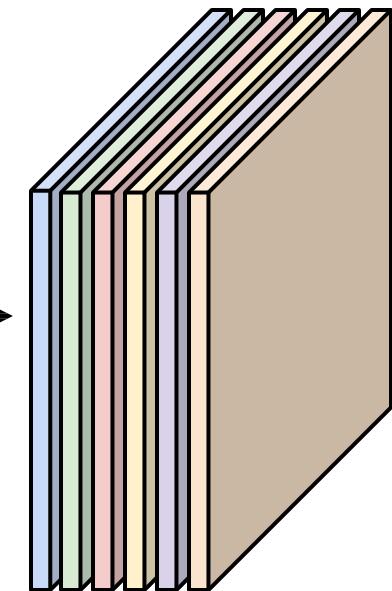
3x32x32 image



Consider 6 filters,
each 3x5x5



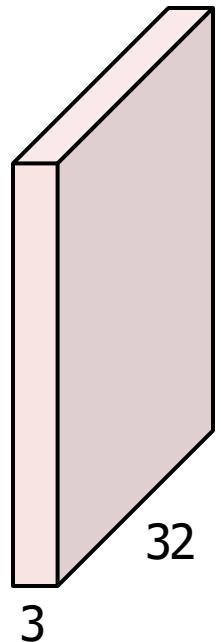
6 activation maps,
each 1x28x28



Stack activations to get a
6x28x28 output image!

Convolution Layer

3x32x32 image

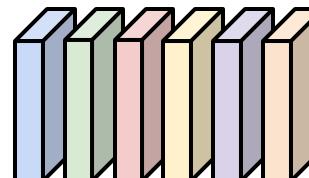


Also 6-dim bias vector:

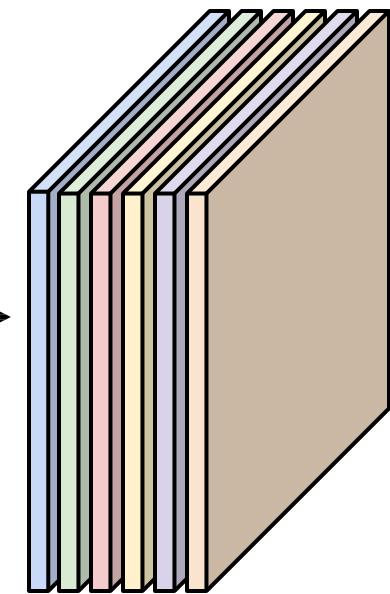


Convolution
Layer

6x3x5x5
filters



6 activation maps,
each 1x28x28

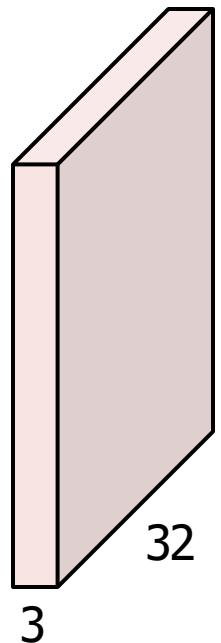


Stack activations to get a
6x28x28 output image!

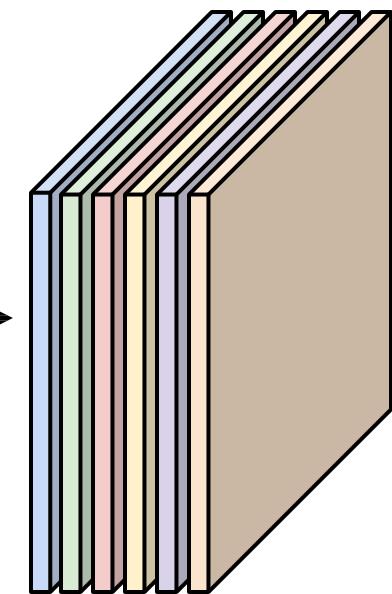
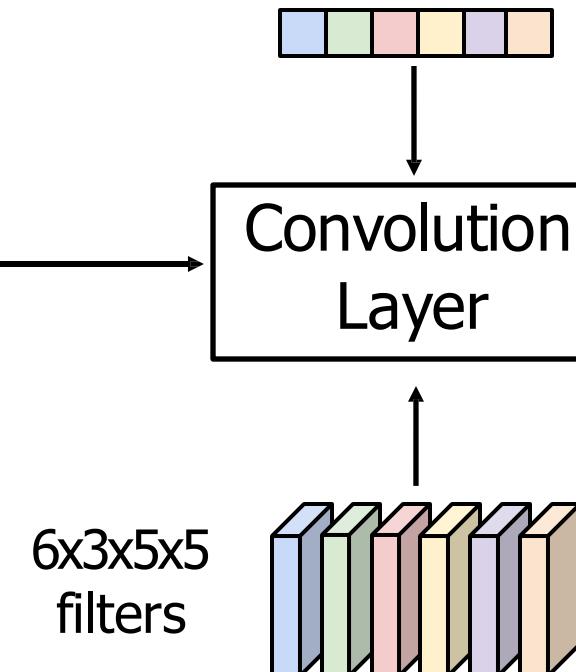
Convolution Layer

28x28 grid, at each point a 6-dim vector

3x32x32 image



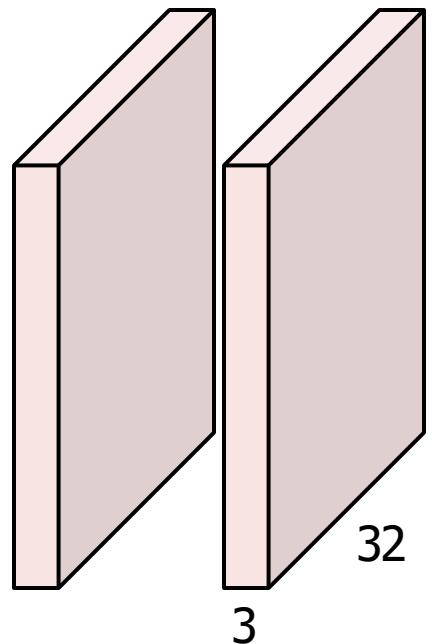
Also 6-dim bias vector:



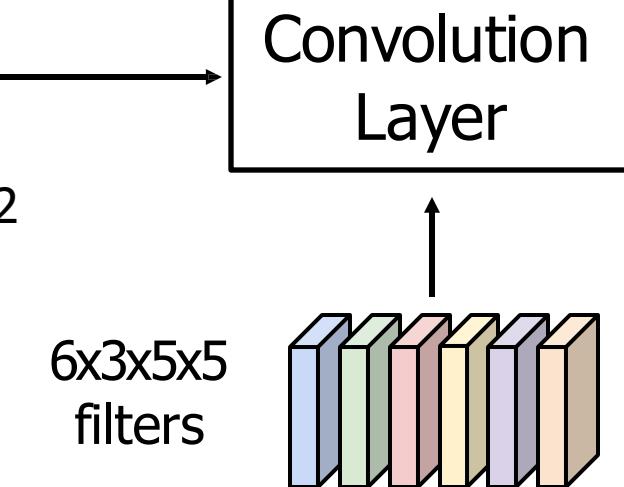
Stack activations to get a 6x28x28 output image!

Convolution Layer

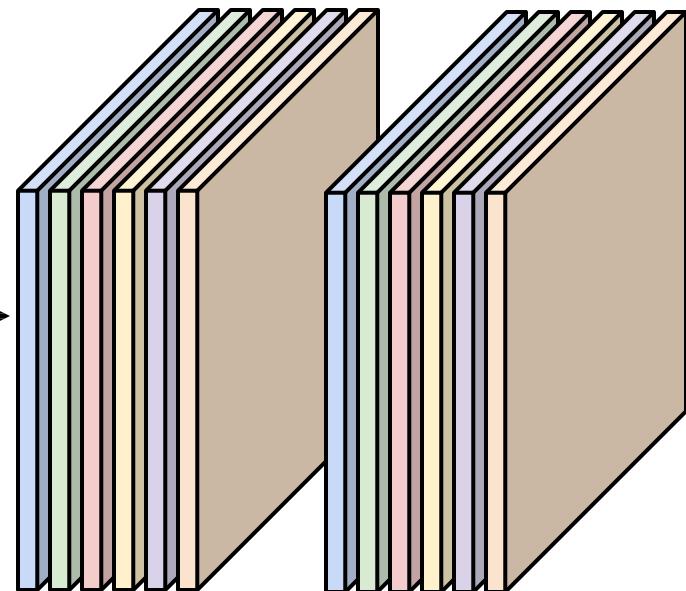
$2 \times 3 \times 32 \times 32$
Batch of images



Also 6-dim bias vector:



$2 \times 6 \times 28 \times 28$
Batch of outputs



Convolution Layer

$N \times C_{in} \times H \times W$

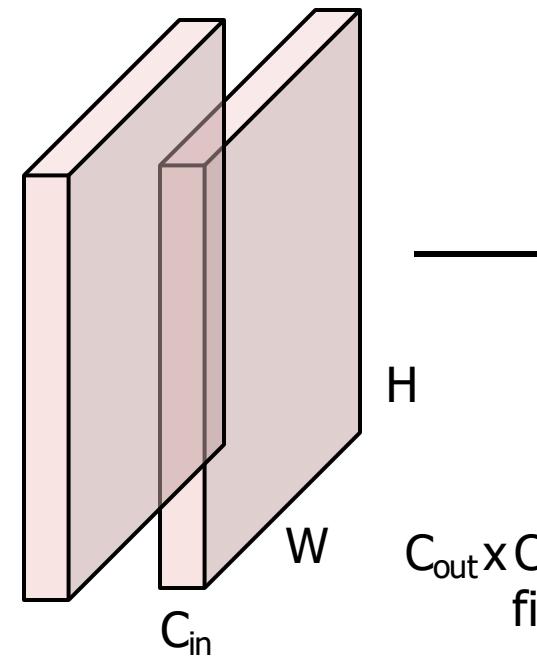
Batch of images

$N \times C_{out} \times H' \times W'$
Batch of outputs

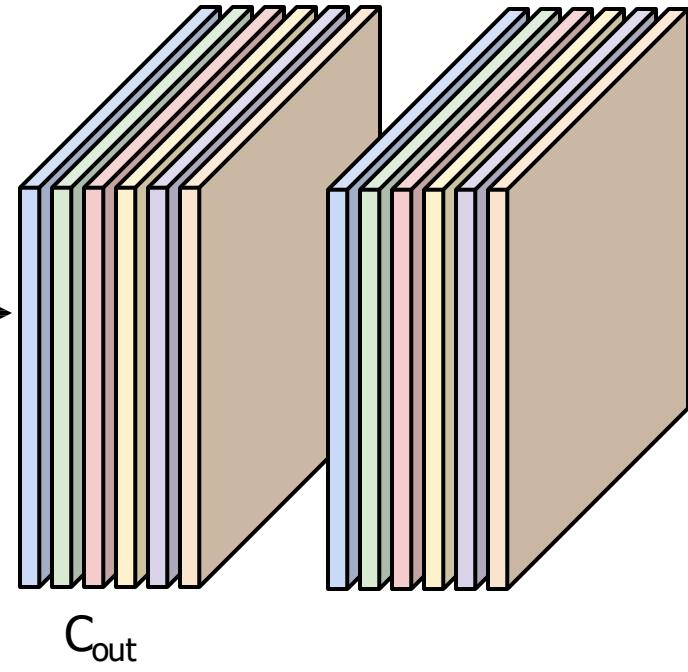
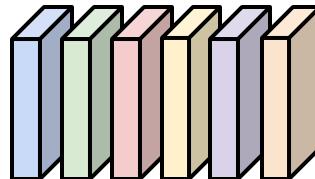
Also C_{out} -dim bias vector:



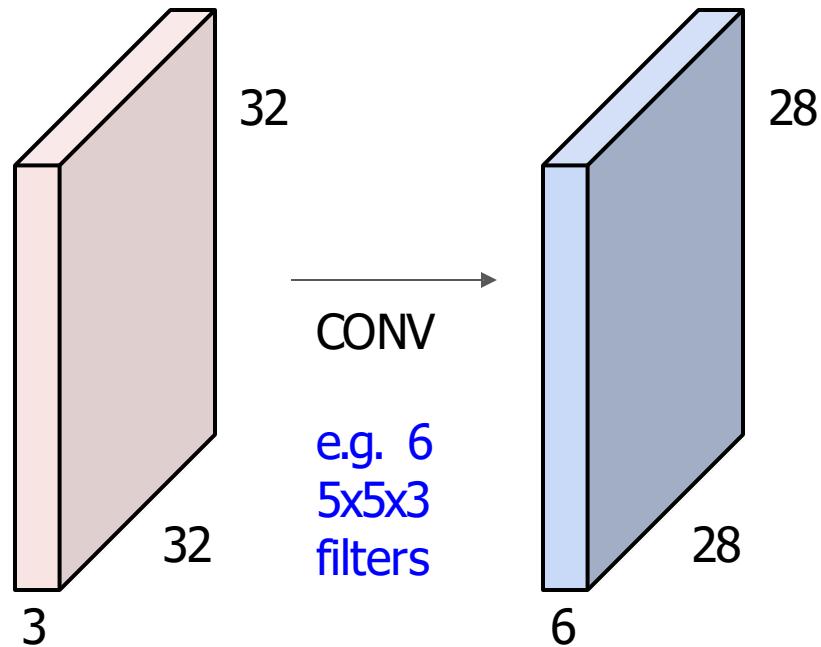
Convolution
Layer



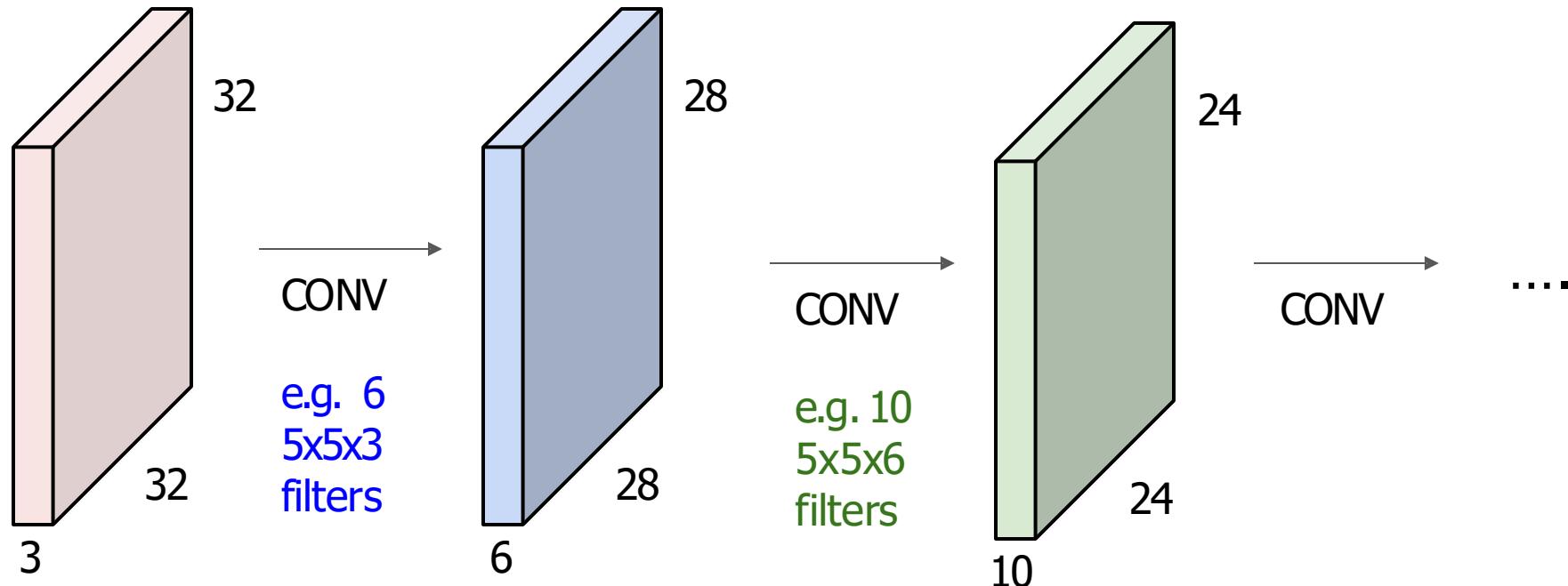
$C_{out} \times C_{in} \times K_w \times K_h$
filters



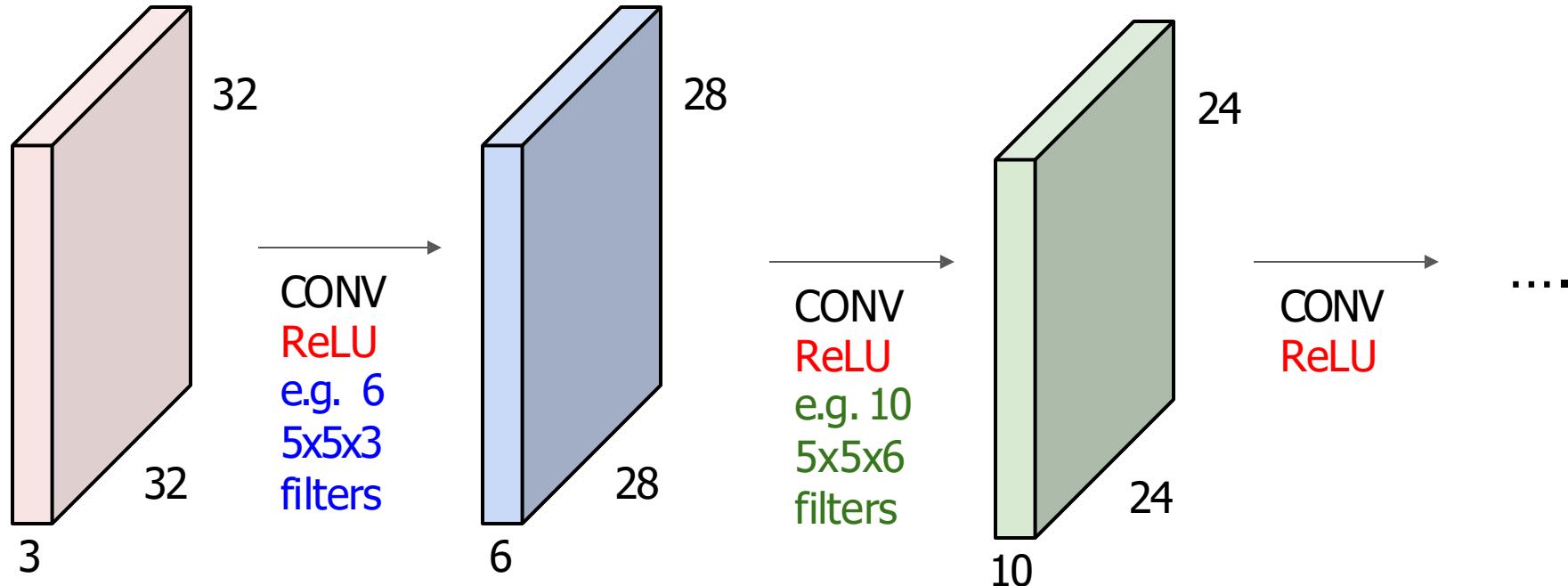
A ConvNet is a neural network with Conv layers



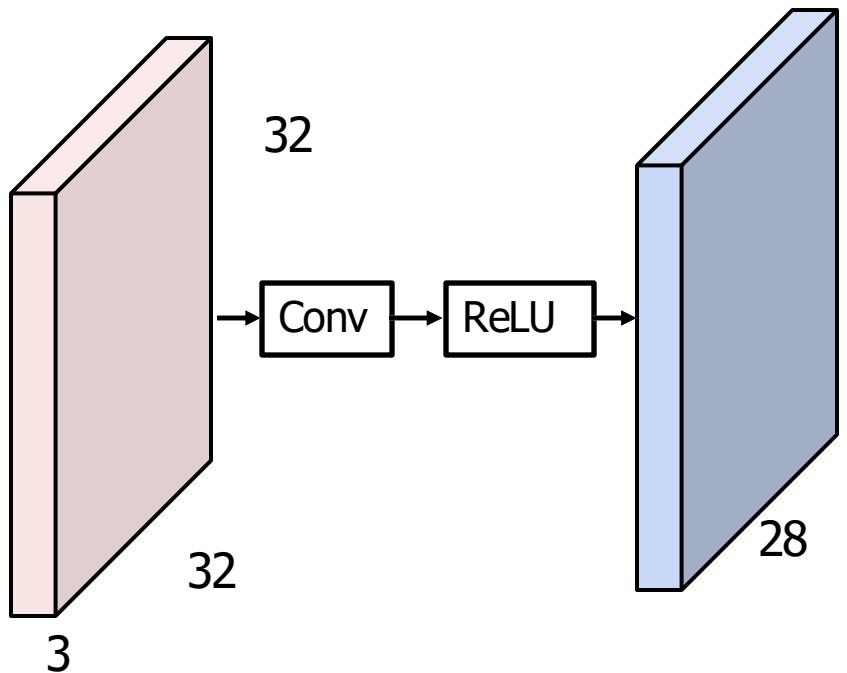
A ConvNet is a neural network with Conv layers



A ConvNet is a neural network with Conv layers with activation functions!



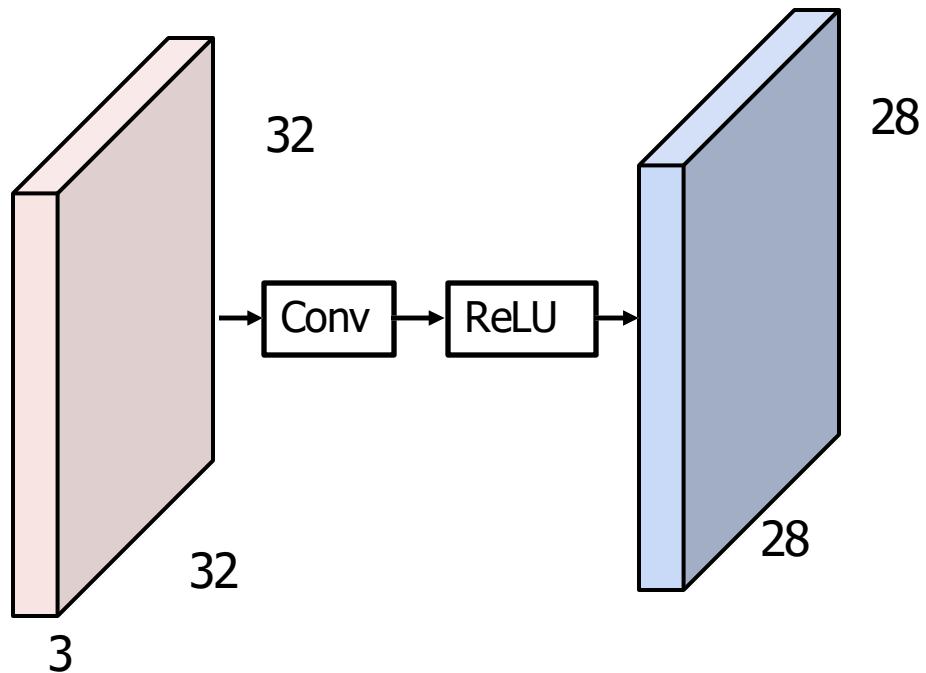
What do Conv filters learn?



Linear classifier: One template per class



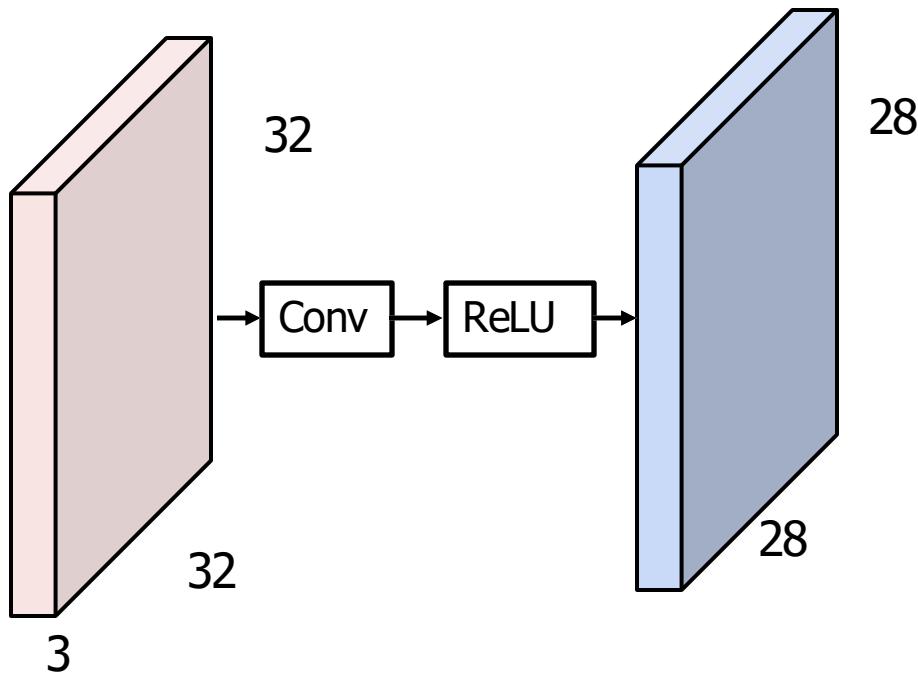
What do Conv filters learn?



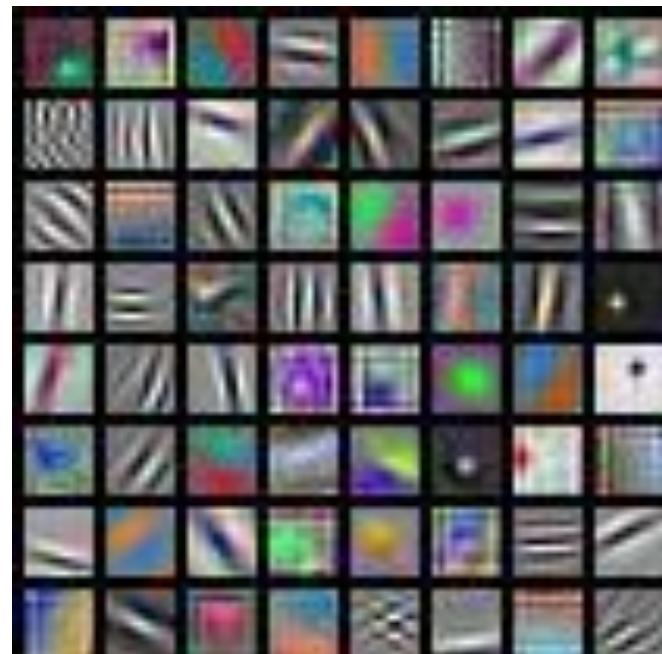
MLP: Bank of whole-image templates



What do Conv filters learn?

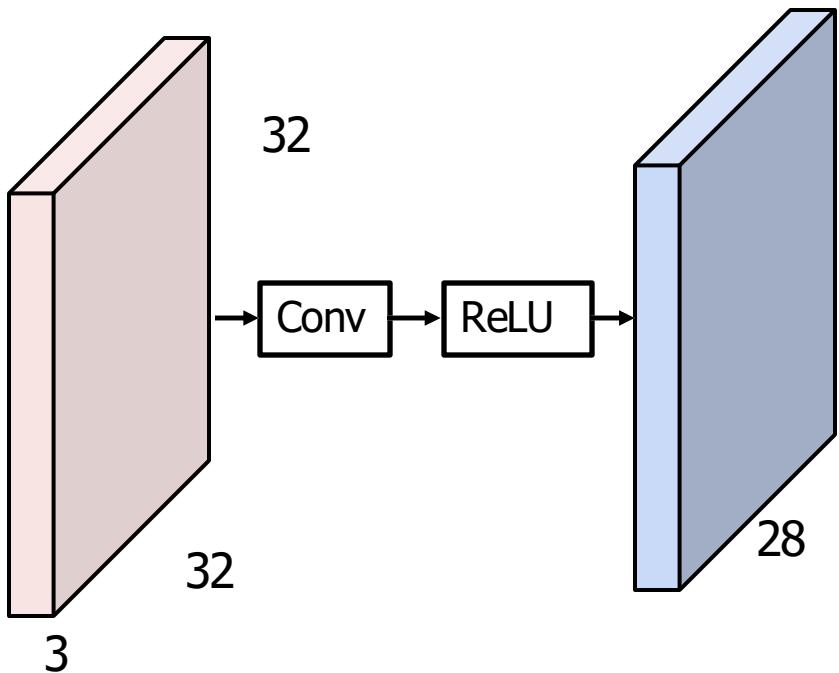


First-layer conv filters: local image templates
(Often learns oriented edges, opposing colors)

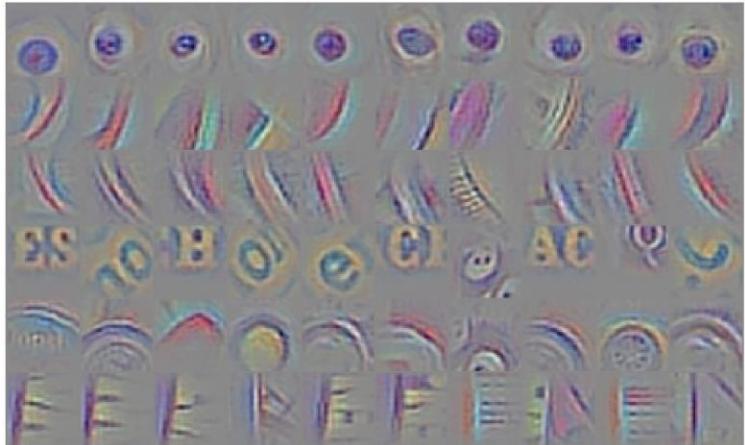


AlexNet: 64 filters, each 3x11x11

What do Conv filters learn?



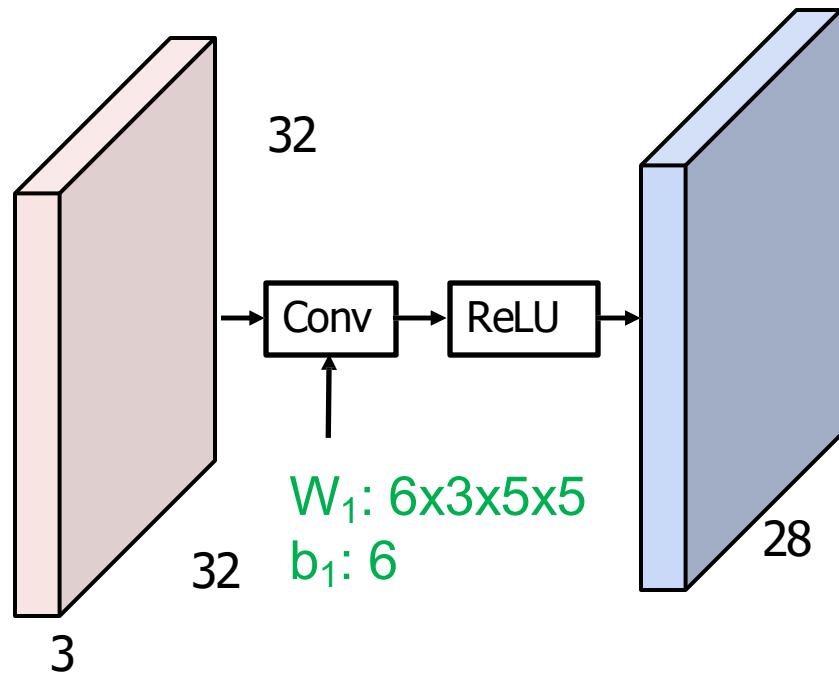
Deeper conv layers: Harder to visualize
Tend to learn larger structures e.g. eyes, letters



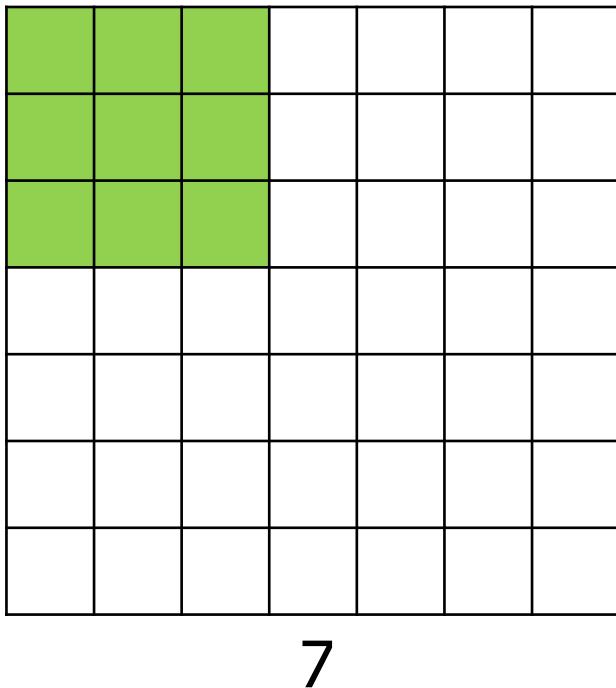
6th layer conv layer from an ImageNet model

Visualization from [Springenberg et al, ICLR 2015]

Convolution: Spatial Dimensions

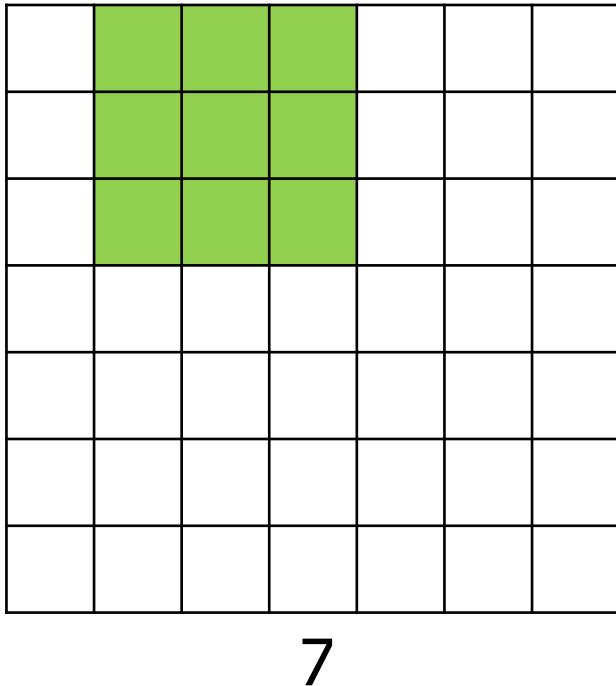


Convolution: Spatial Dimensions



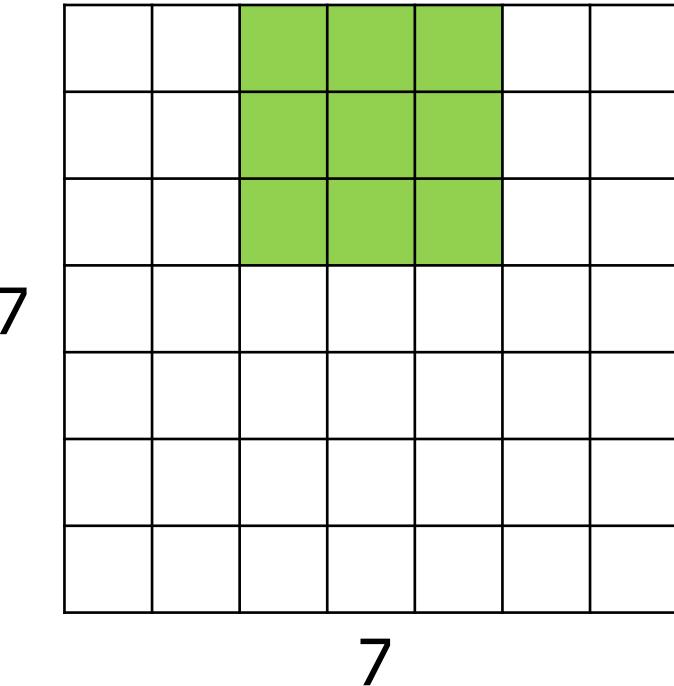
Input: 7x7
Filter: 3x3

Convolution: Spatial Dimensions



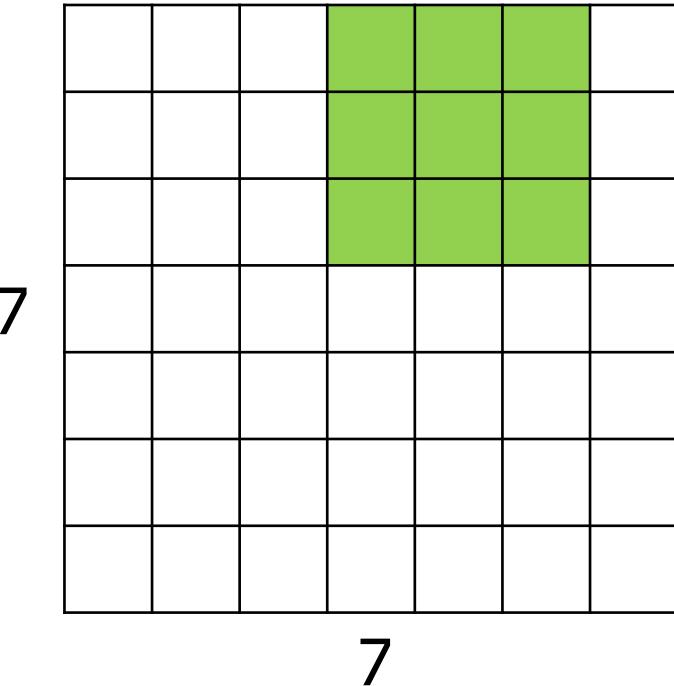
Input: 7x7
Filter: 3x3

Convolution: Spatial Dimensions



Input: 7x7
Filter: 3x3

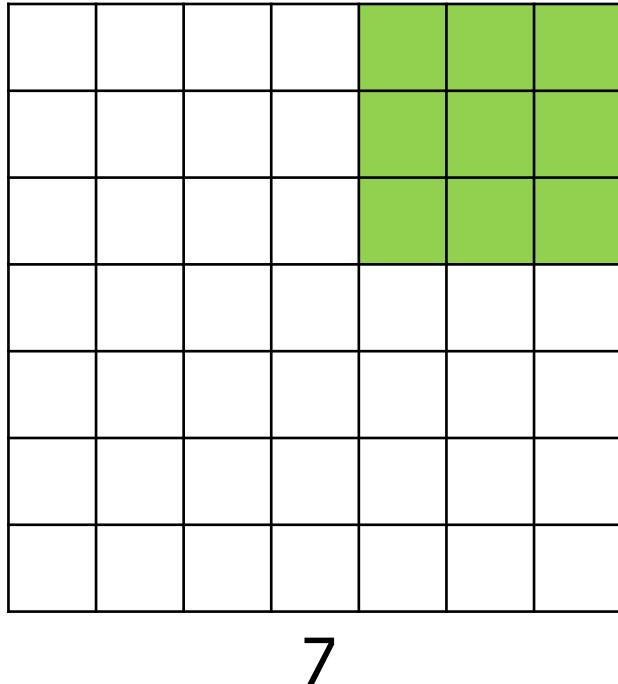
Convolution: Spatial Dimensions



Input: 7x7
Filter: 3x3

Convolution: Spatial Dimensions

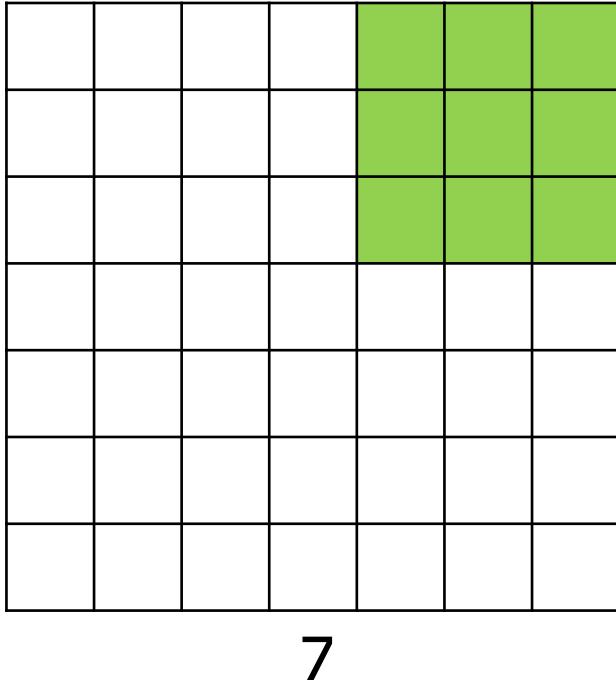
7



Input: 7x7
Filter: 3x3
Output: 5x5

Convolution: Spatial Dimensions

7

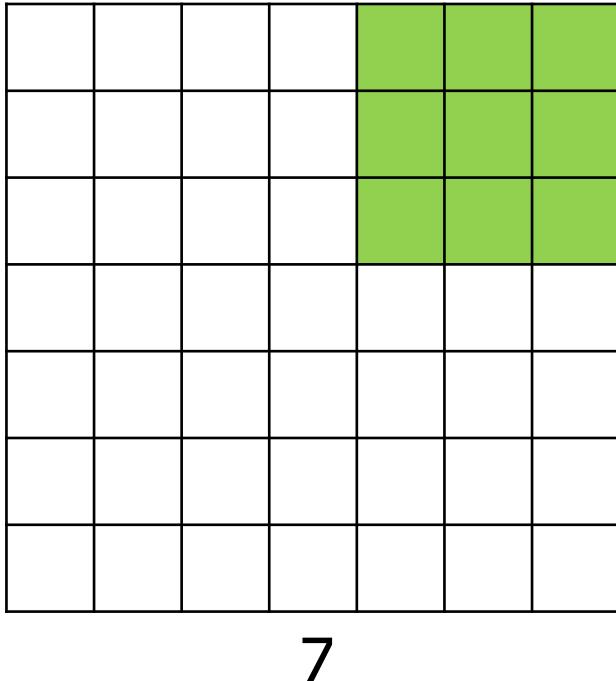


Input: 7x7
Filter: 3x3
Output: 5x5

In general
Input: W
Filter: K
Output: $W - K + 1$

Convolution: Spatial Dimensions

7



Input: 7x7
Filter: 3x3
Output: 5x5

Problem: Feature maps shrink with each layer!

In general
Input: W
Filter: K
Output: $W - K + 1$

Convolution: Spatial Dimensions

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

Input: 7x7

Filter: 3x3

Output: 5x5

In general

Input: W

Filter: K

Padding: P

Output: $W - K + 1 + 2P$

Problem: Feature maps shrink with each layer!

Solution: Add padding around the input before sliding the filter

Convolution: Spatial Dimensions

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

Input: 7x7

Filter: 3x3

Output: 5x5

In general

Input: W

Filter: K

Padding: P

Output: $W - K + 1 + 2P$

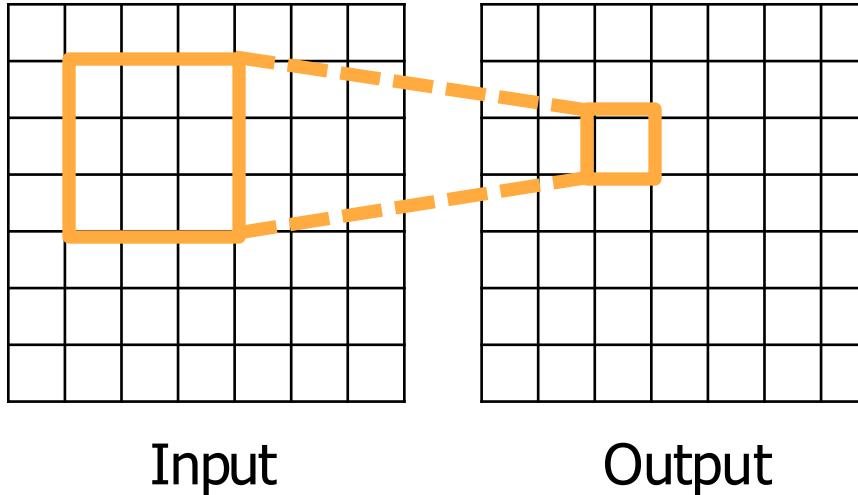
Common setting:

$$P = (K - 1) / 2$$

Means output has
same size as input

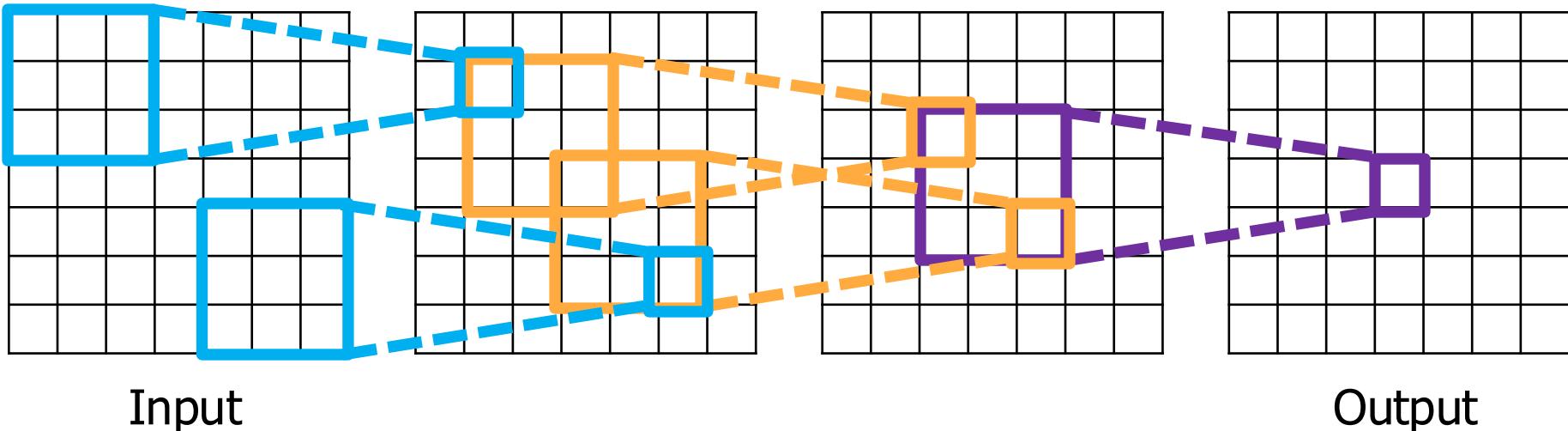
Receptive Fields

For convolution with **kernel size K**, each element in the output depends on a $K \times K$ receptive field in the input



Receptive Fields

Each successive convolution adds $K - 1$ to the receptive field size
With L layers the receptive field size is $1 + L * (K - 1)$



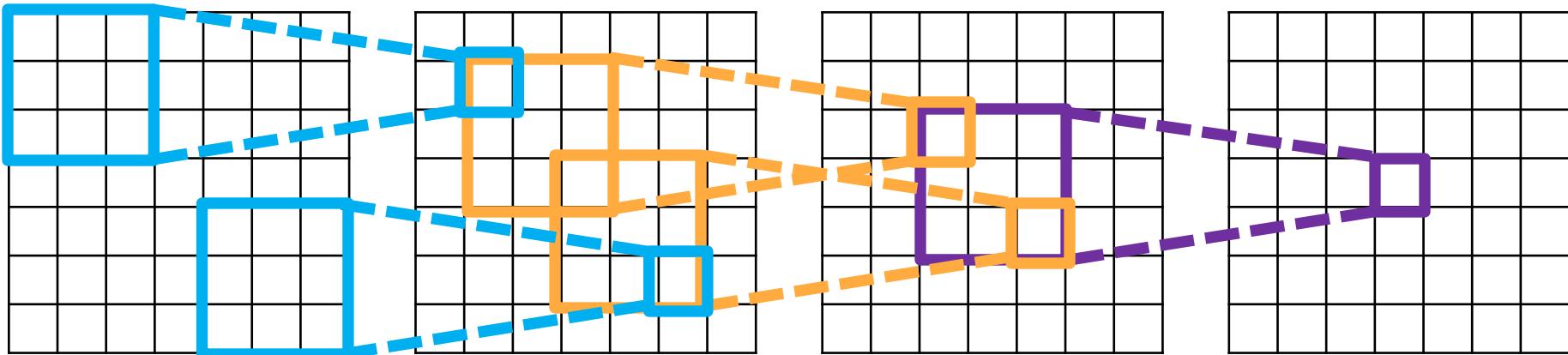
Input

Output

Be careful – “receptive field in the input” vs. “receptive field in the previous layer”

Receptive Fields

Each successive convolution adds $K - 1$ to the receptive field size
With L layers the receptive field size is $1 + L * (K - 1)$



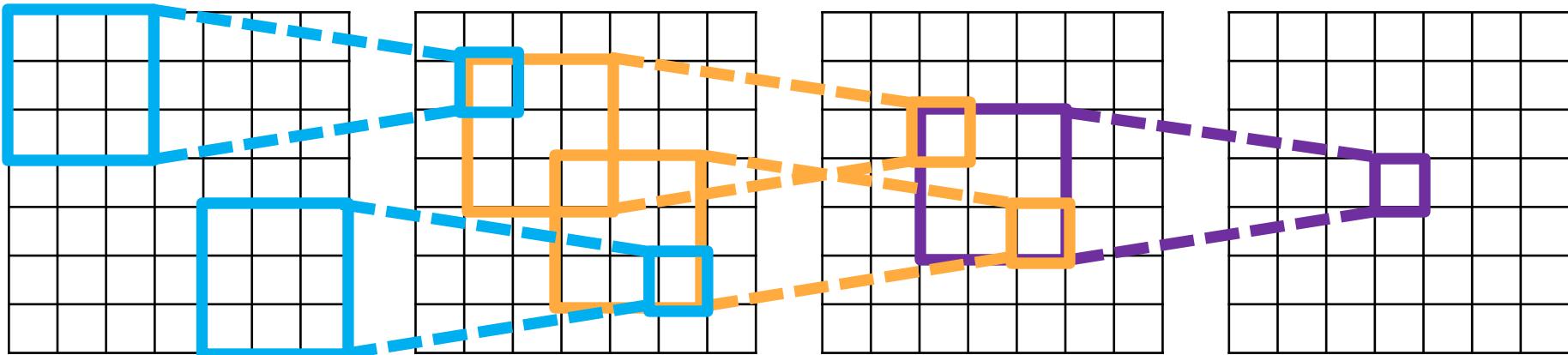
Input

Problem: For large images we need many layers for each output to “see” the whole image

Output

Receptive Fields

Each successive convolution adds $K - 1$ to the receptive field size
With L layers the receptive field size is $1 + L * (K - 1)$



Input

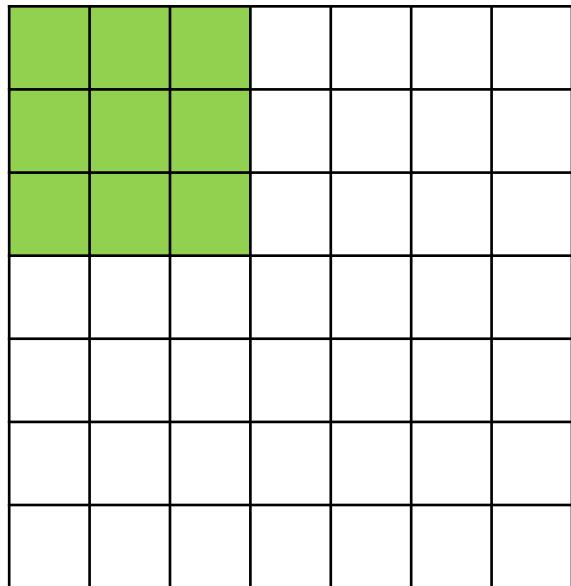
Problem: For large images we need many layers for each output to "see" the whole image

Output

Solution: Downsample inside the network

Strided Convolution

7



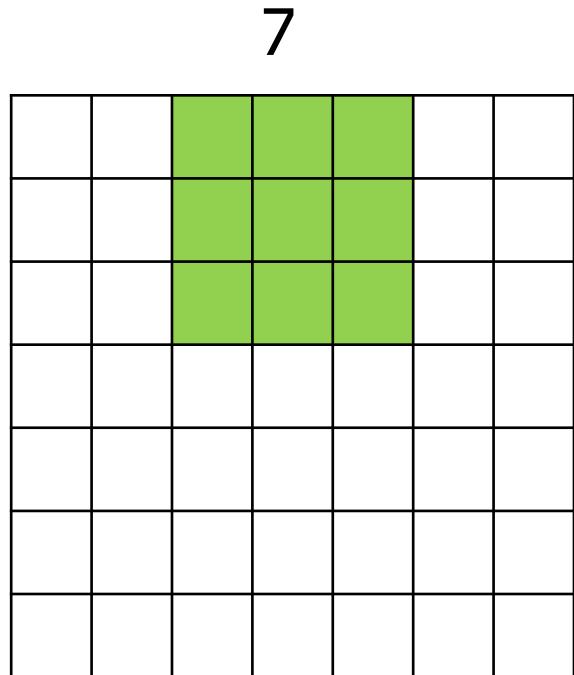
7

Input: 7x7

Filter: 3x3

Stride: 2

Strided Convolution



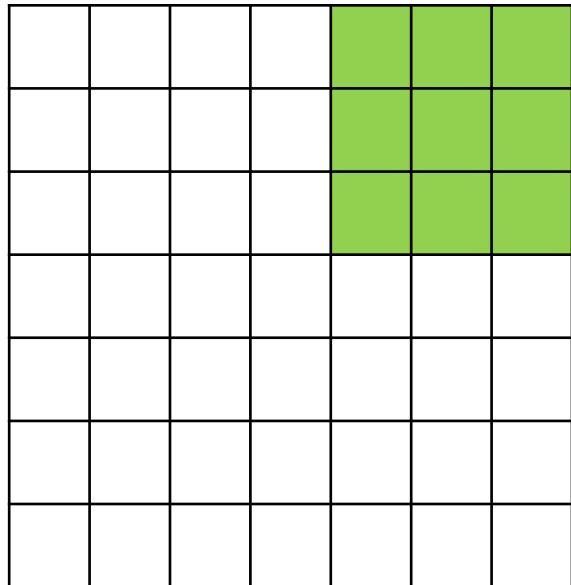
Input: 7x7

Filter: 3x3

Stride: 2

Strided Convolution

7



7

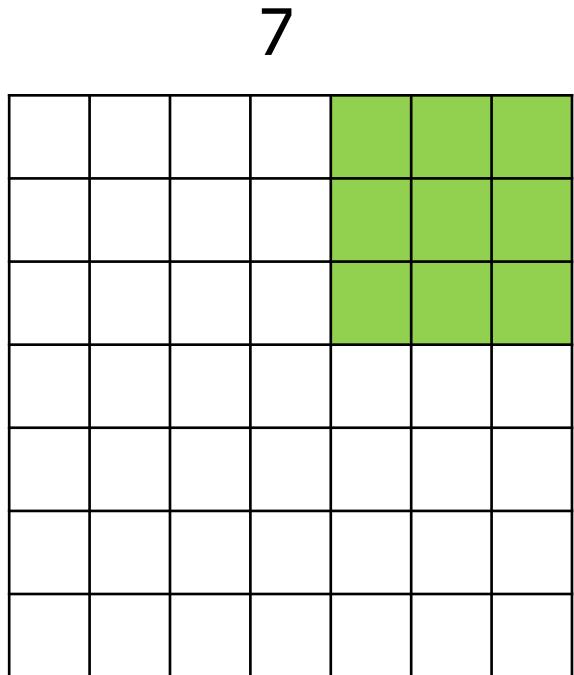
Input: 7x7

Filter: 3x3

Stride: 2

Output: 3x3

Strided Convolution



Input: 7x7

Filter: 3x3

Stride: 2

Output: 3x3

In general:

Input: W

Filter: K

Padding: P

Stride: S

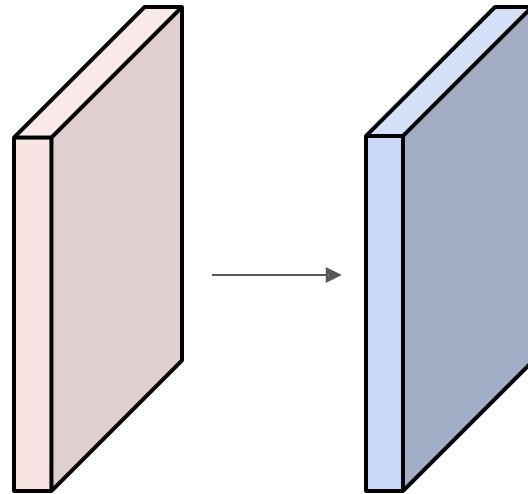
→ Output:
 $(W - K + 2P) / S + 1$

Convolution Example

Input volume: $3 \times 32 \times 32$

10 5x5 filters with stride 1, pad 2

Output volume size: ?



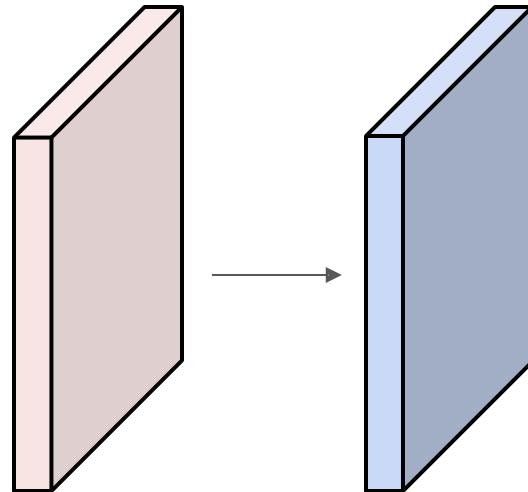
Convolution Example

Input volume: $3 \times 32 \times 32$

$10 \text{ } 5 \times 5$ filters with stride 1 , pad 2

Output volume size: $10 \times 32 \times 32$

$$32 = (32 + 2 * 2 - 5) / 1 + 1$$



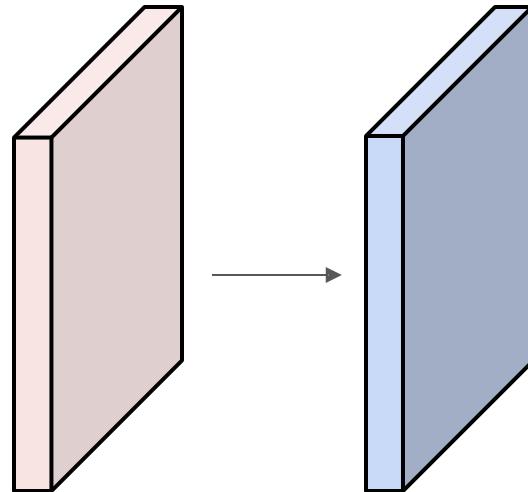
Convolution Example

Input volume: $3 \times 32 \times 32$

10 5x5 filters with stride 1, pad 2

Output volume size: $10 \times 32 \times 32$

Number of learnable parameters: ?



Convolution Example

Input volume: **3** x 32 x 32

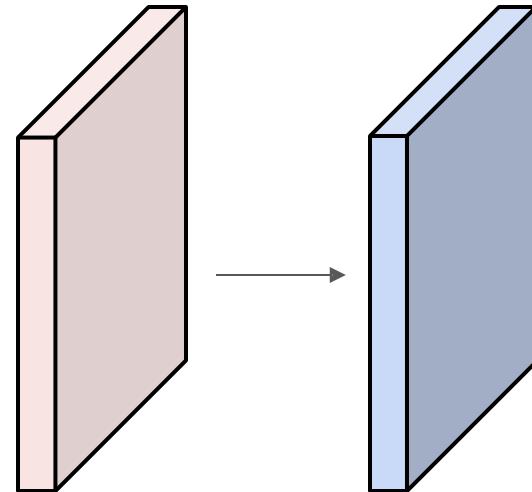
10 5x5 filters with stride 1, pad 2

Output volume size: 10 x 32 x 32

Number of learnable parameters: 760

Parameters per filter: **3*5*5 + 1** (for bias) = **76**

10 filters, so total is **10 * 76 = 760**



Convolution Example

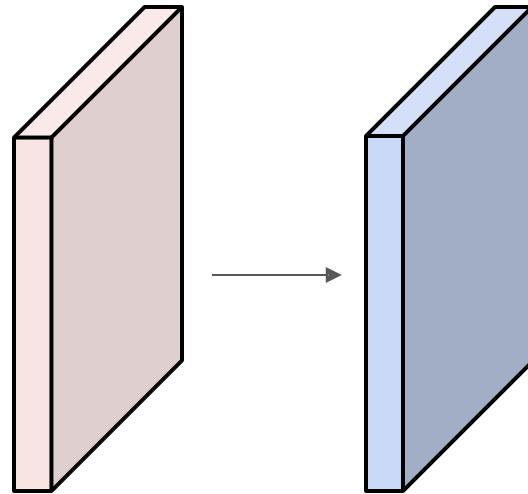
Input volume: $3 \times 32 \times 32$

10 5x5 filters with stride 1, pad 2

Output volume size: $10 \times 32 \times 32$

Number of learnable parameters: 760

Number of multiply-add operations?



Convolution Example

Input volume: **3x 32 x 32**

10 **5x5** filters with stride 1, pad 2

Output volume size: **10x 32x 32**

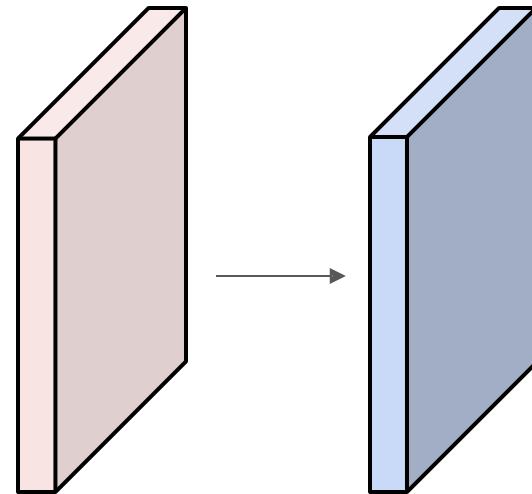
Number of learnable parameters: 760

Number of multiply-add operations: **768,000**

10*32*32 = 10,240 outputs

Each output is the inner product of two **3x5x5** tensors (75 elems)

Total = $75 * 10240 = \mathbf{768K}$



Convolution Summary

Input: $C_{in} \times H \times W$

Hyperparameters:

- **Kernel size:** $K_H \times K_W$
- **Number filters:** C_{out}
- **Padding:** P
- **Stride:** S

Weight matrix: $C_{out} \times C_{in} \times K_H \times K_W$

giving C_{out} filters of size $C_{in} \times K_H \times K_W$

Bias vector: C_{out}

Output size: $C_{out} \times H' \times W'$ where:

- $H' = (H - K + 2P) / S + 1$
- $W' = (W - K + 2P) / S + 1$

Common settings:

$K_H = K_W$ (Small square filters)

$P = (K - 1) / 2$ ("Same" padding)

$C_{in}, C_{out} = 32, 64, 128, 256$ (powers of 2)

$K = 3, P = 1, S = 1$ (3x3 conv)

$K = 5, P = 2, S = 1$ (5x5 conv)

$K = 1, P = 0, S = 1$ (1x1 conv)

$K = 3, P = 1, S = 2$ (Downsample by 2)

PyTorch Convolution Layer

```
CLASS torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0,  
dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None) [SOURCE]
```

Applies a 2D convolution over an input signal composed of several input planes.

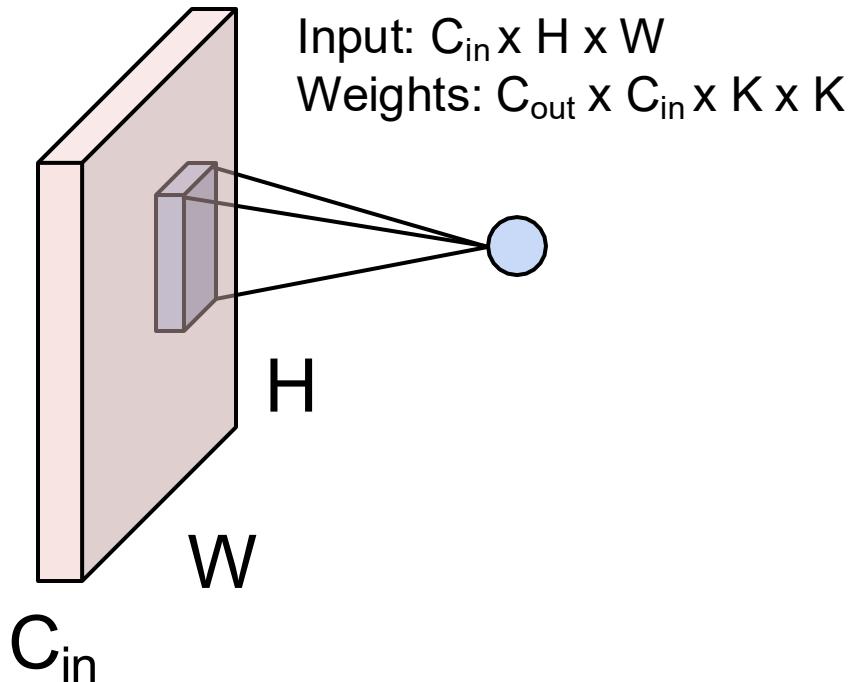
In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

We didn't talk about groups or dilation...

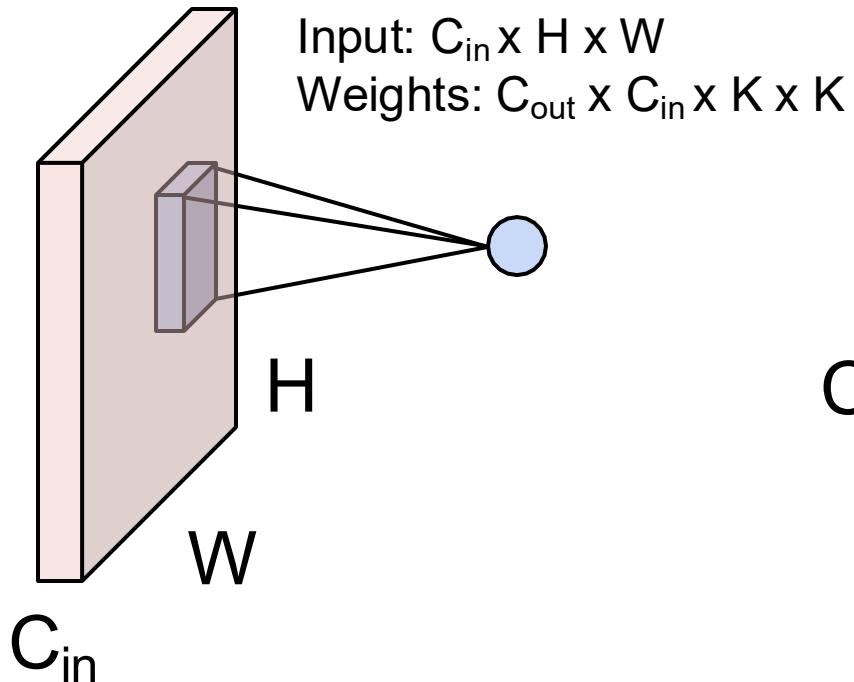
Other Types of Convolution

So far: 2D Convolution

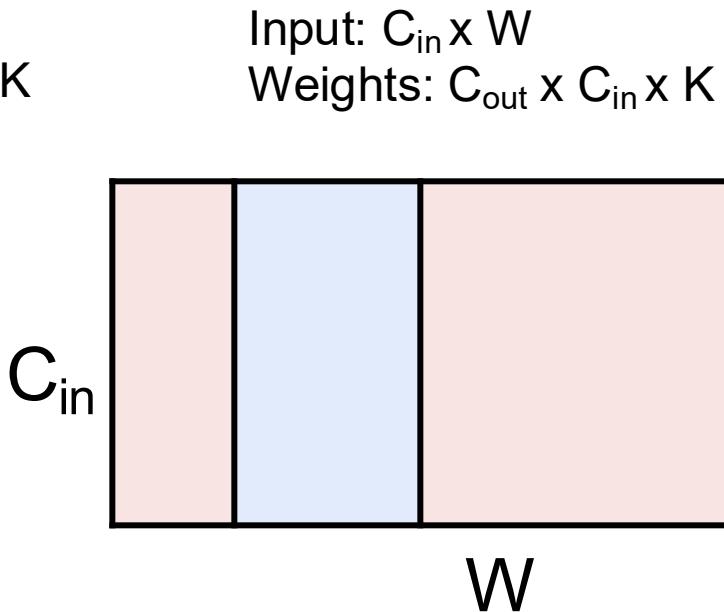


Other Types of Convolution

So far: 2D Convolution

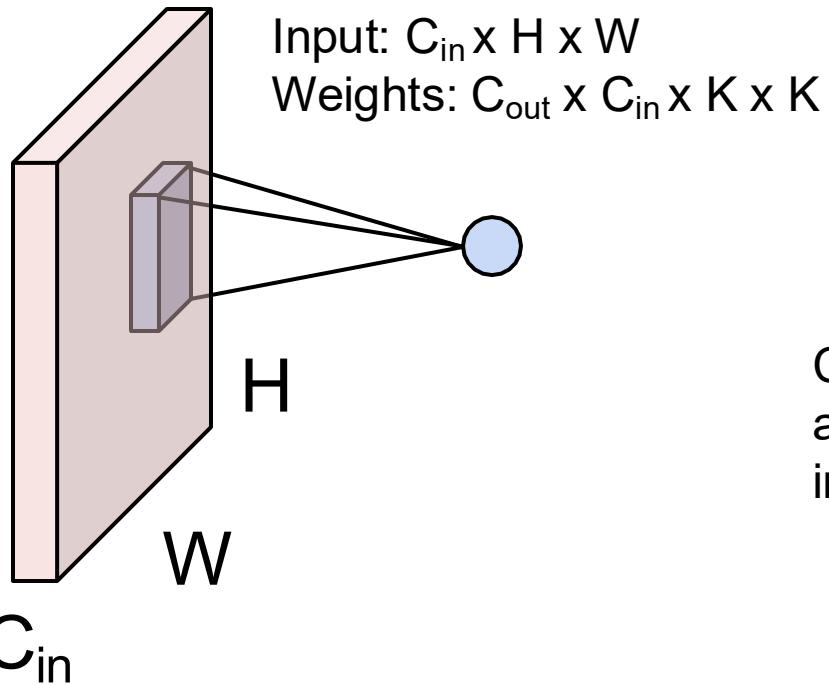


1D Convolution



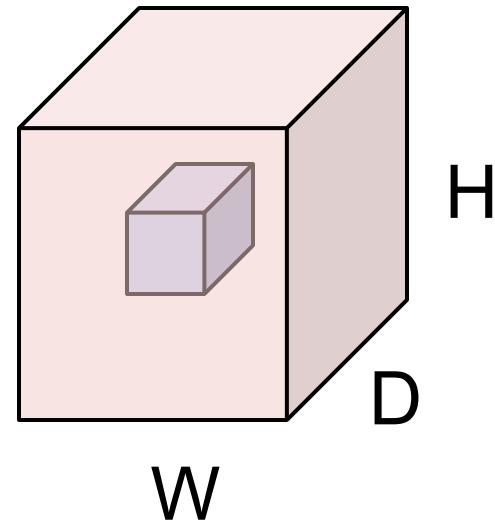
Other Types of Convolution

So far: 2D Convolution



3D Convolution

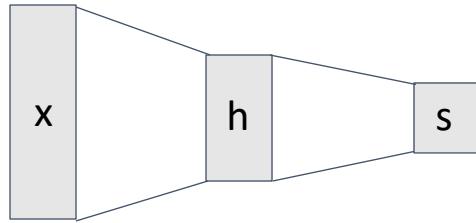
Input: $C_{in} \times H \times W \times D$
Weights: $C_{out} \times C_{in} \times K \times K \times K$



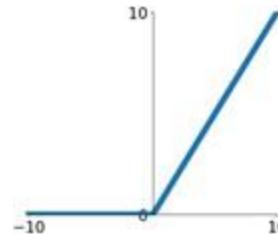
Convolutional Networks

Fully-Connected Layer

We have
already
seen these

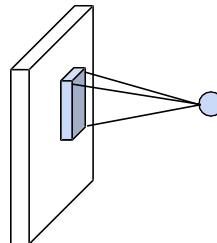


Activation Function

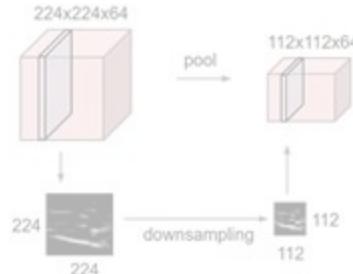


Convolution Layer

Today: Image-specific operators



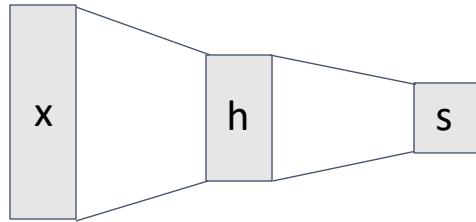
Pooling Layer



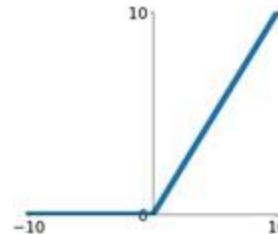
Convolutional Networks

Fully-Connected Layer

We have
already
seen these

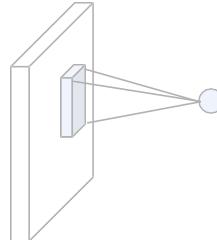


Activation Function

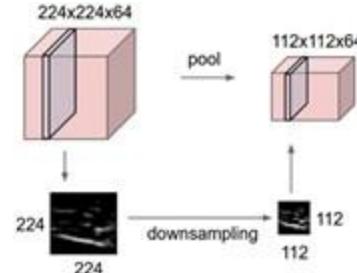


Convolution Layer

Today: Image-specific operators

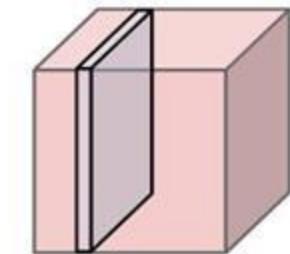


Pooling Layer



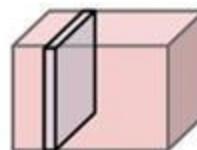
Pooling Layers: Another way to downsample

$64 \times 224 \times 224$

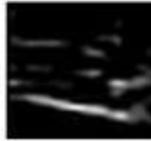


pool

$64 \times 112 \times 112$



224



224

downsampling

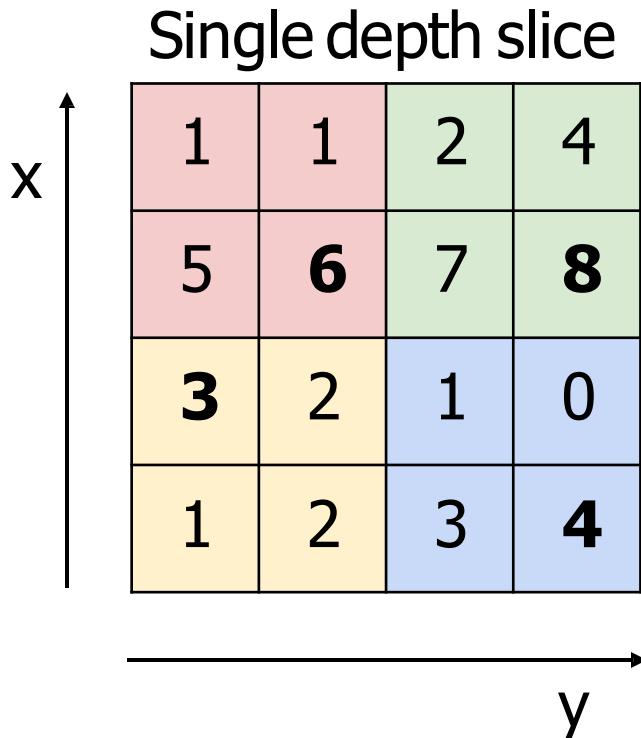


112

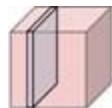
Given an input $C \times H \times W$,
downsample each $1 \times H \times W$ plane

Hyperparameters:
Kernel Size
Stride
Pooling function

Pooling Layers: Another way to downsample



64 x 224 x 224



Max pooling with 2x2 kernel size and stride 2

6	8
3	4

Gives **invariance** to small spatial shifts. No learnable parameters.

Pooling Summary

Input: $C \times H \times W$

Hyperparameters:

- **Kernel size:** K
- **Stride:** S
- **Pooling function:** max, avg

Output size: $C \times H' \times W'$ where:

- $H' = (H - K) / S + 1$
- $W' = (W - K) / S + 1$

No learnable parameters

Common setting:
max, $K=2, S=2 \Rightarrow$ Gives 2x downsampling

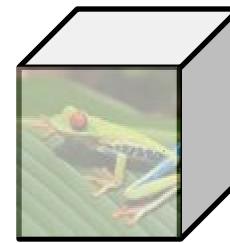
Convolution and Pooling: Translation Equivariance

$H \times W \times C$

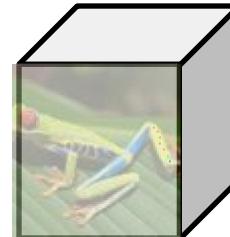


Conv or Pool

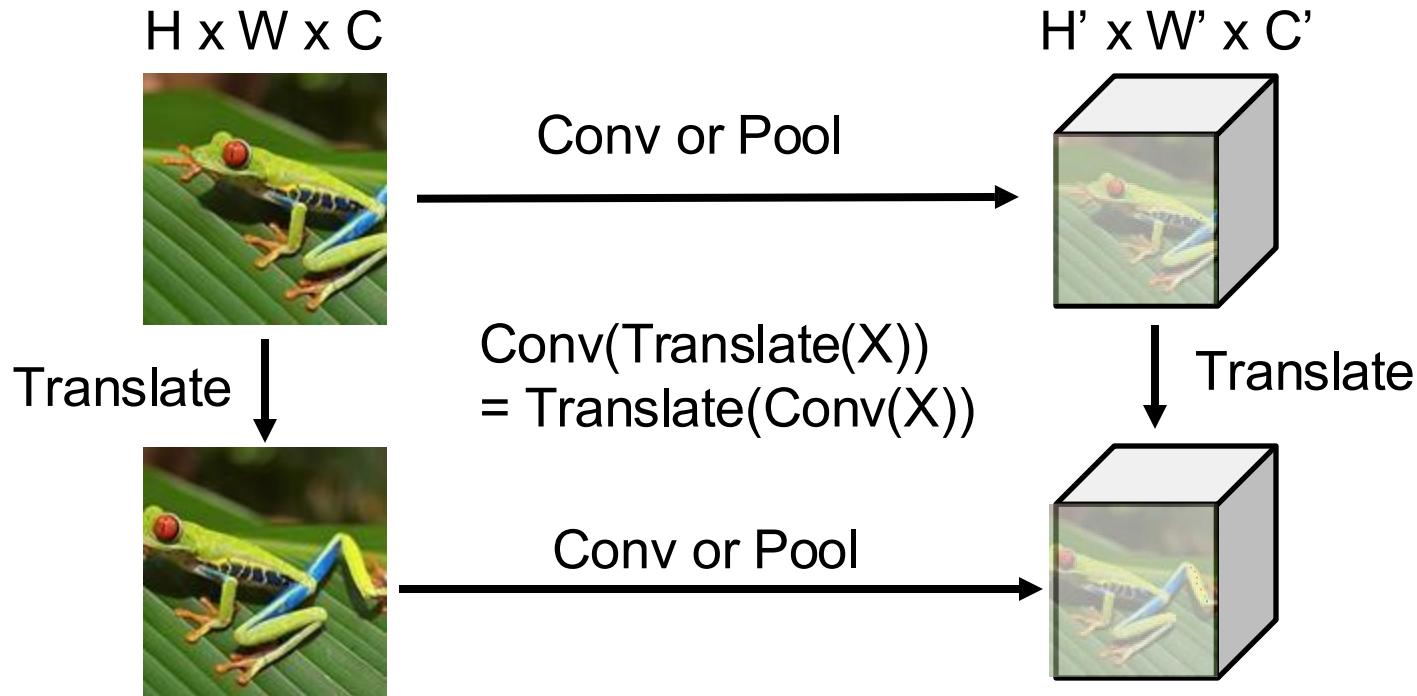
$H' \times W' \times C'$



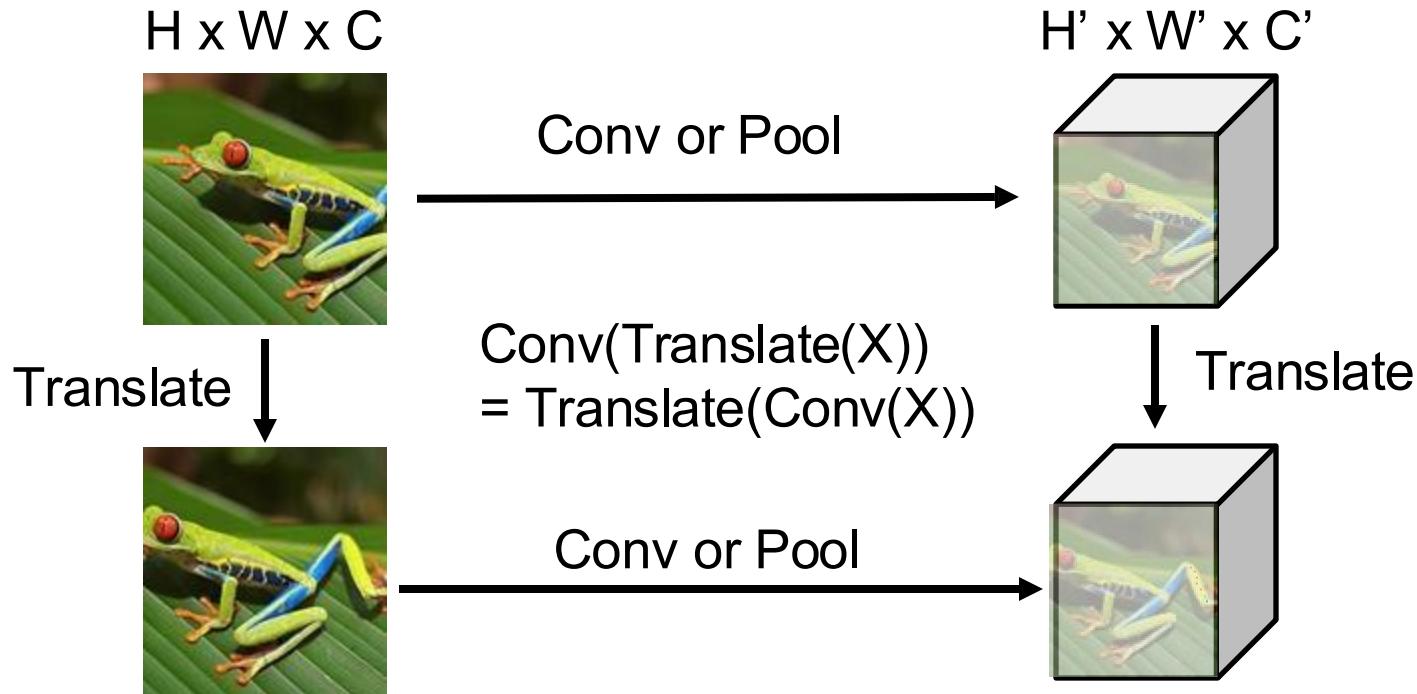
Translate



Convolution and Pooling: Translation Equivariance



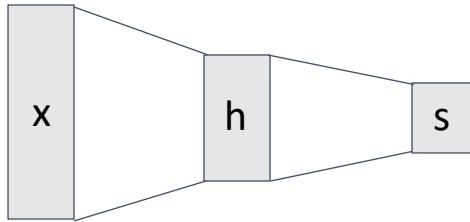
Convolution and Pooling: Translation Equivariance



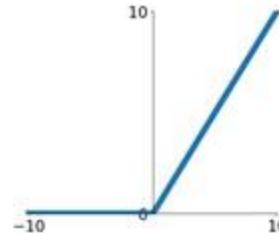
Intuition: Features of images don't depend on their location in the image

Summary: Convolutional Networks

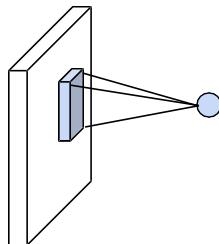
Fully-Connected Layer



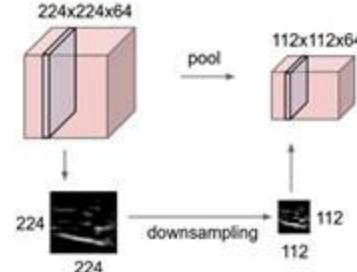
Activation Function



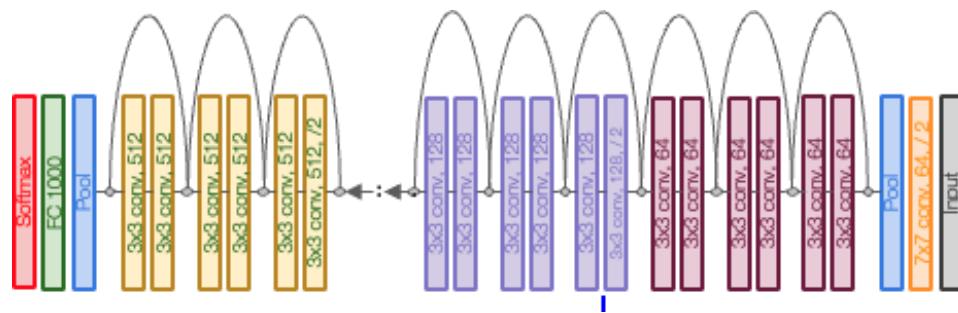
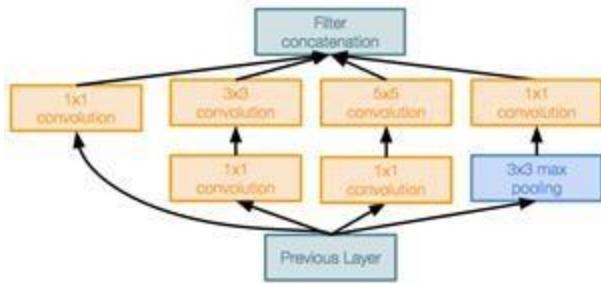
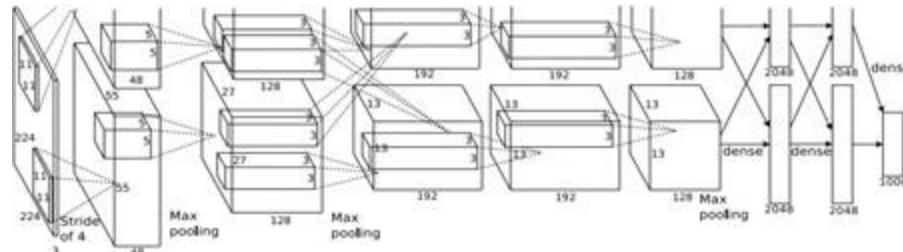
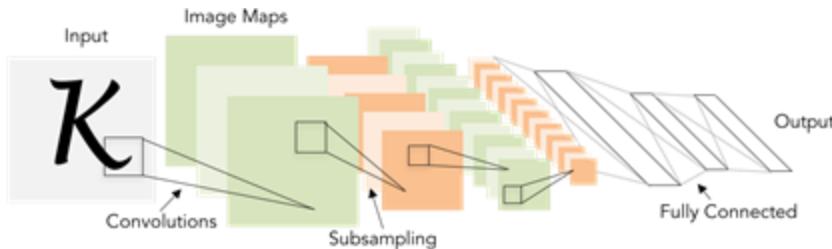
Convolution Layer



Pooling Layer



Next time: CNN Architectures



Lecture 6: Training CNNs and CNN Architectures



Lecture Overview – Two Broad Sets of Topics

How to build CNNs?

Layers in CNNs
Activation Functions
CNN Architectures
Weight Initialization

How to train CNNs?

Data Preprocessing
Data augmentation
Transfer Learning
Hyperparameter Selection

Lecture Overview – Two Broad Sets of Topics

How to build CNNs?

Layers in CNNs
Activation Functions
CNN Architectures
Weight Initialization

How to train CNNs?

Data Preprocessing
Data augmentation
Transfer Learning
Hyperparameter Selection

Lecture Overview – Two Broad Sets of Topics

How to build CNNs?

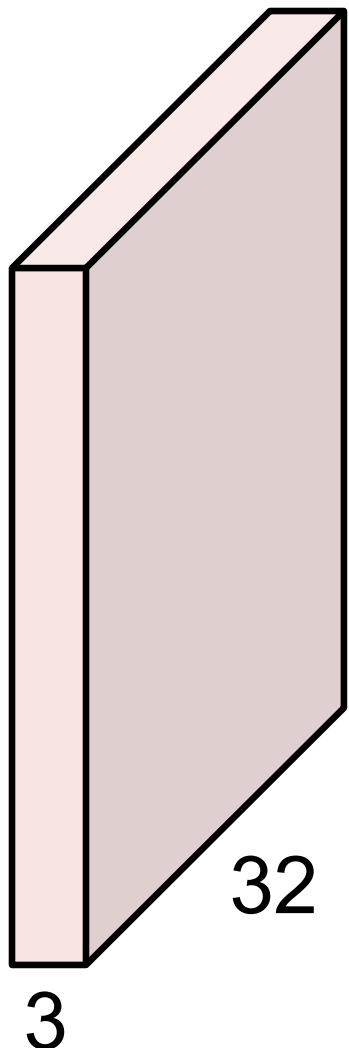
{ **Layers in CNNs**
Activation Functions
CNN Architectures
Weight Initialization

How to train CNNs?

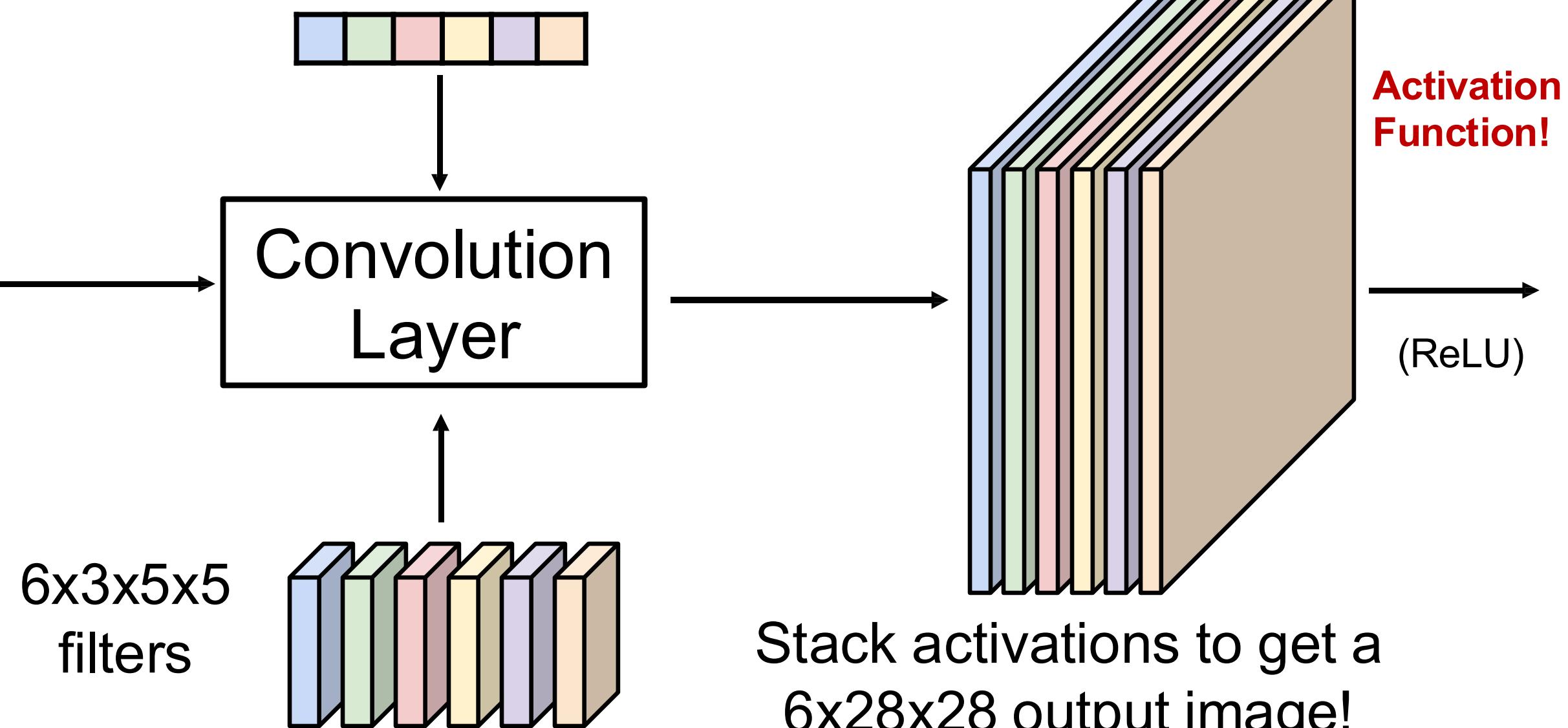
{ **Data Preprocessing**
Data augmentation
Transfer Learning
Hyperparameter Selection

Recap: Convolution Layer

3x32x32 image

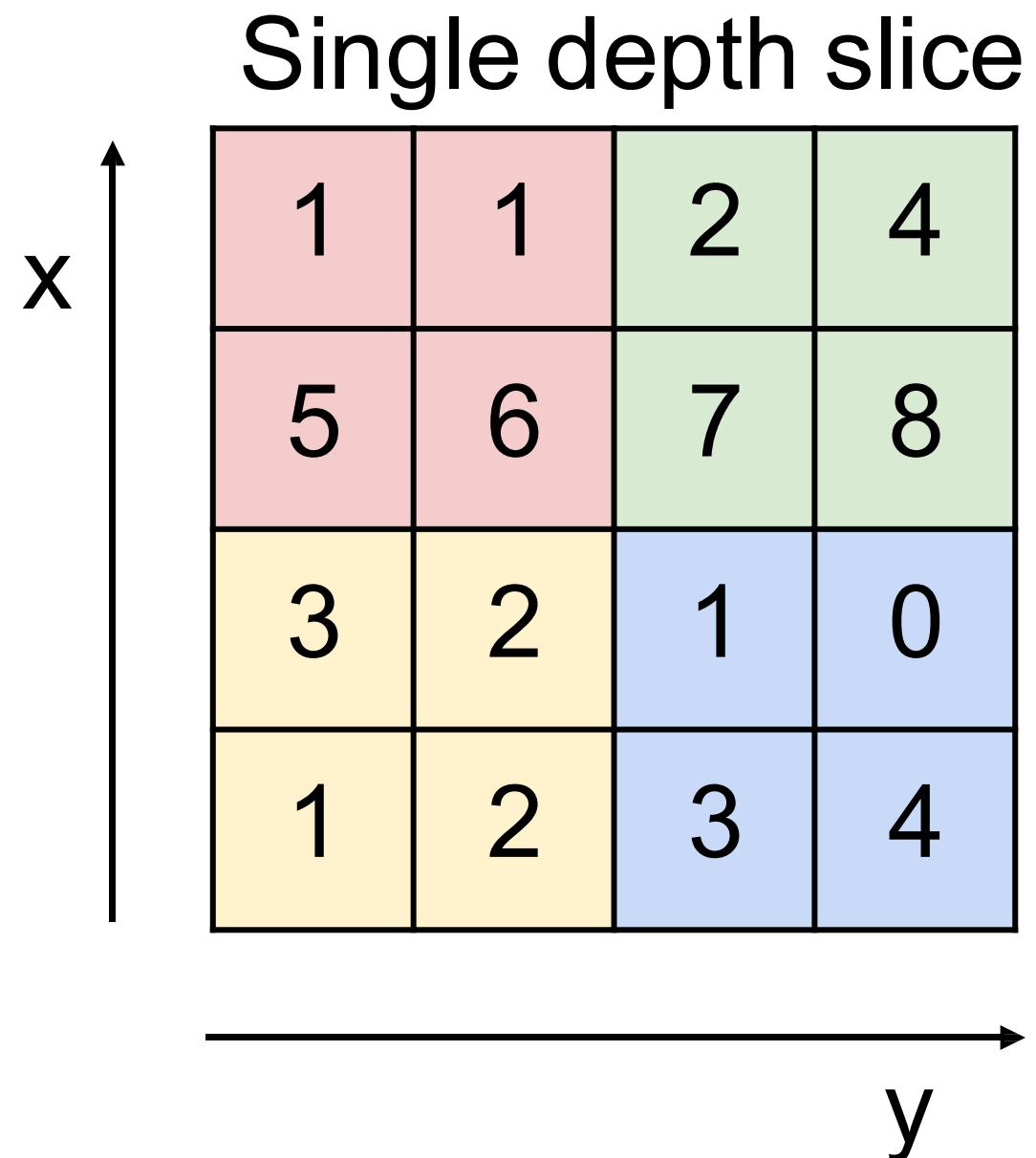


Don't forget bias terms!



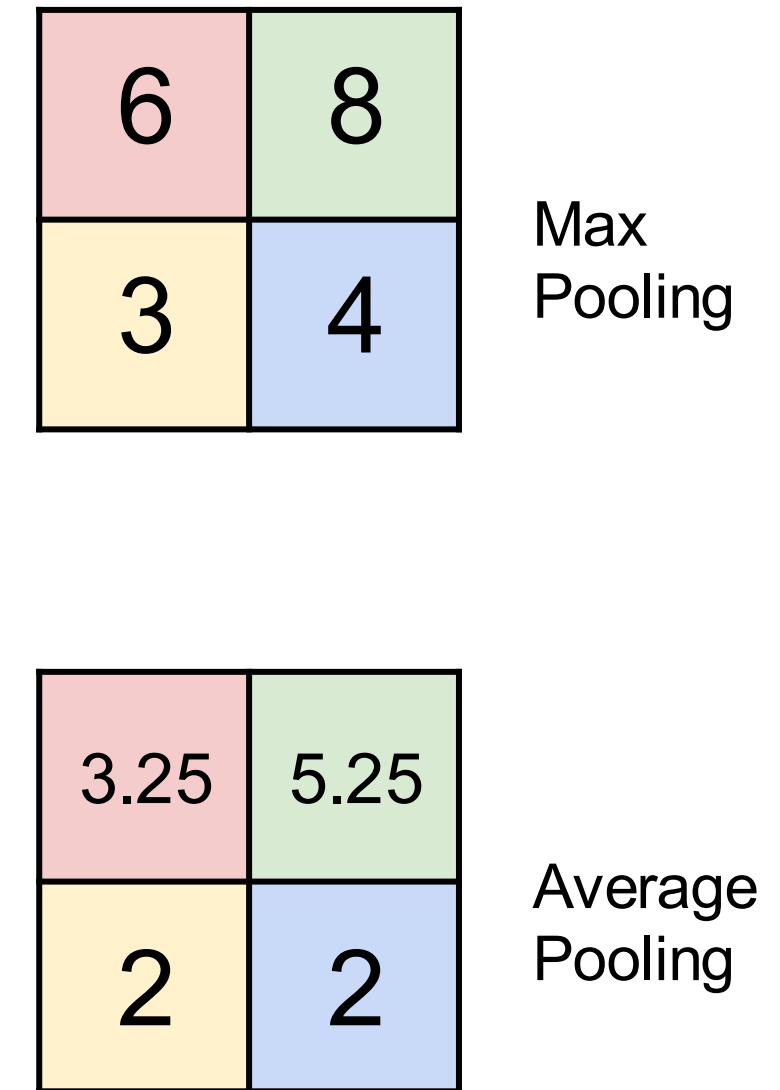
Slide inspiration: Justin Johnson

Recap: Pooling Layer



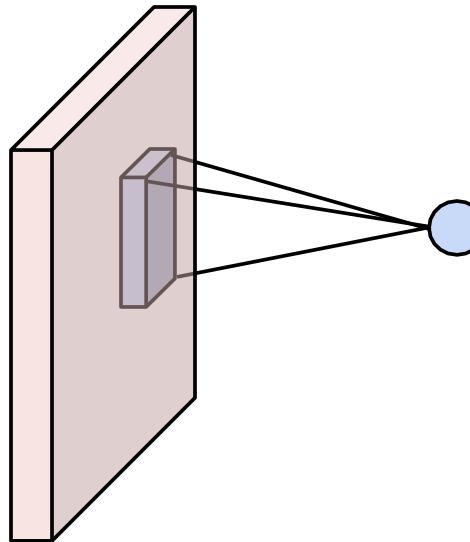
pool with 2x2 filters and
stride 2

→

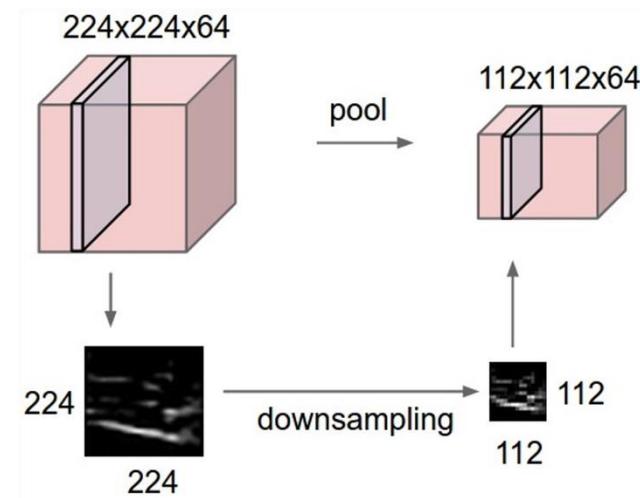


Components of CNNs

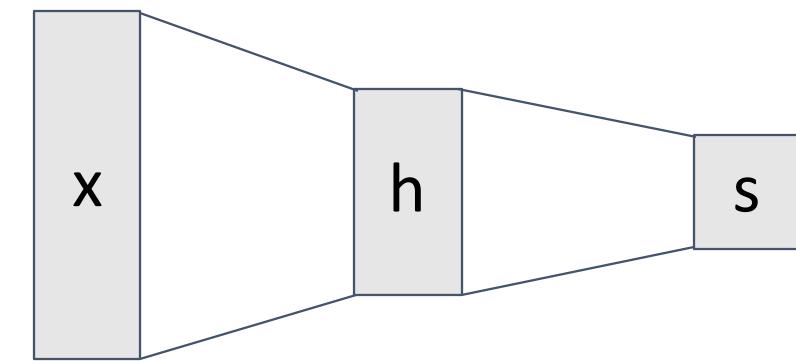
Convolution Layers



Pooling Layers



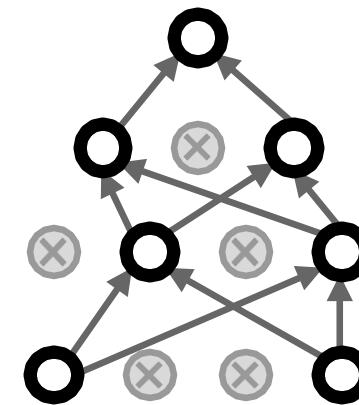
Fully-Connected Layers



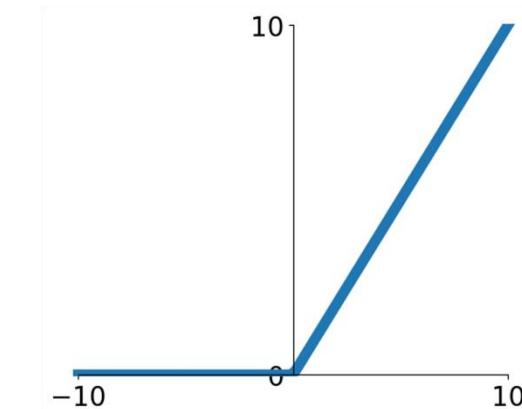
Normalization Layers

$$x_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Dropout (sometimes)

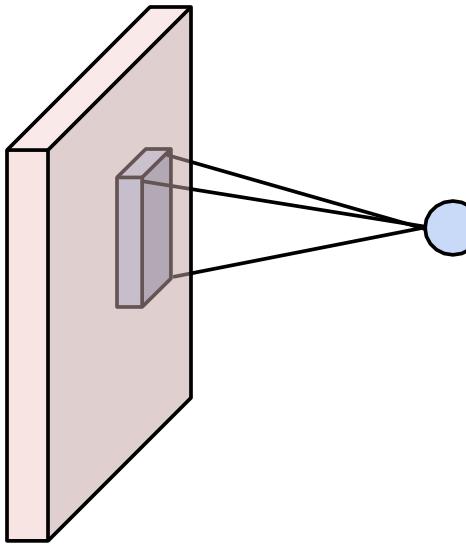


Activation Functions

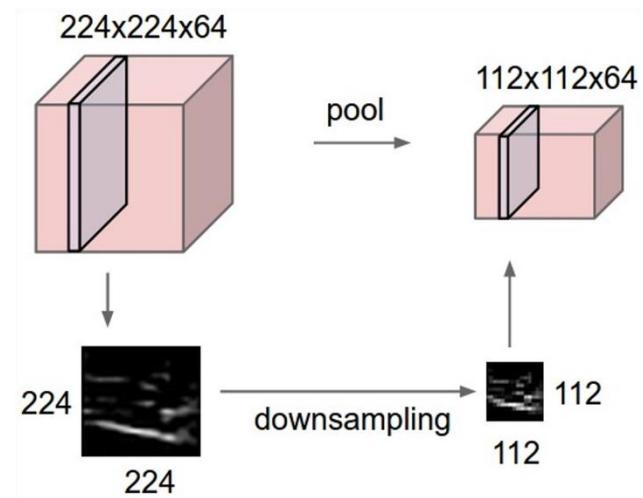


Components of CNNs

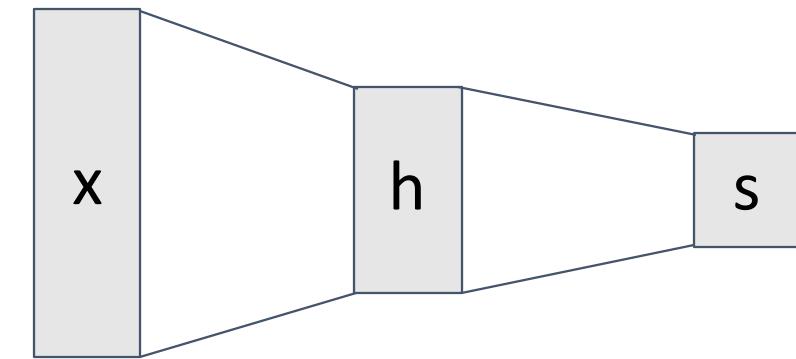
Convolution Layers



Pooling Layers



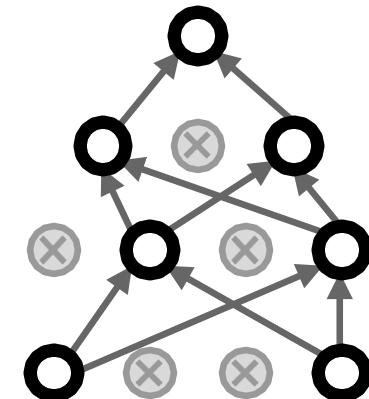
Fully-Connected Layers



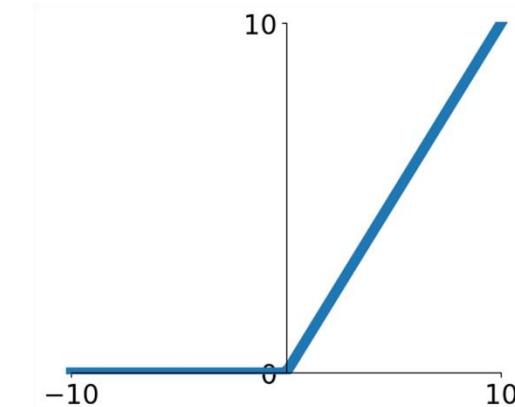
Normalization Layers

$$x_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Dropout (sometimes)



Activation Functions



Example Normalization Layer: LayerNorm

High-level Idea: Learn parameters that let us **scale / shift the input data**

1. Normalize input data
2. Scale / shift using learned parameters

Ba, Kiros, and Hinton, “Layer Normalization”, arXiv 2016

Example Normalization Layer: LayerNorm

High-level Idea: Learn parameters that let us **scale / shift the input data**

1. Normalize input data
2. Scale / shift using learned parameters

Statistics calculated per batch →

Learned parameters applied to each sample →

$$\mathbf{x} : N \times D$$

Normalize

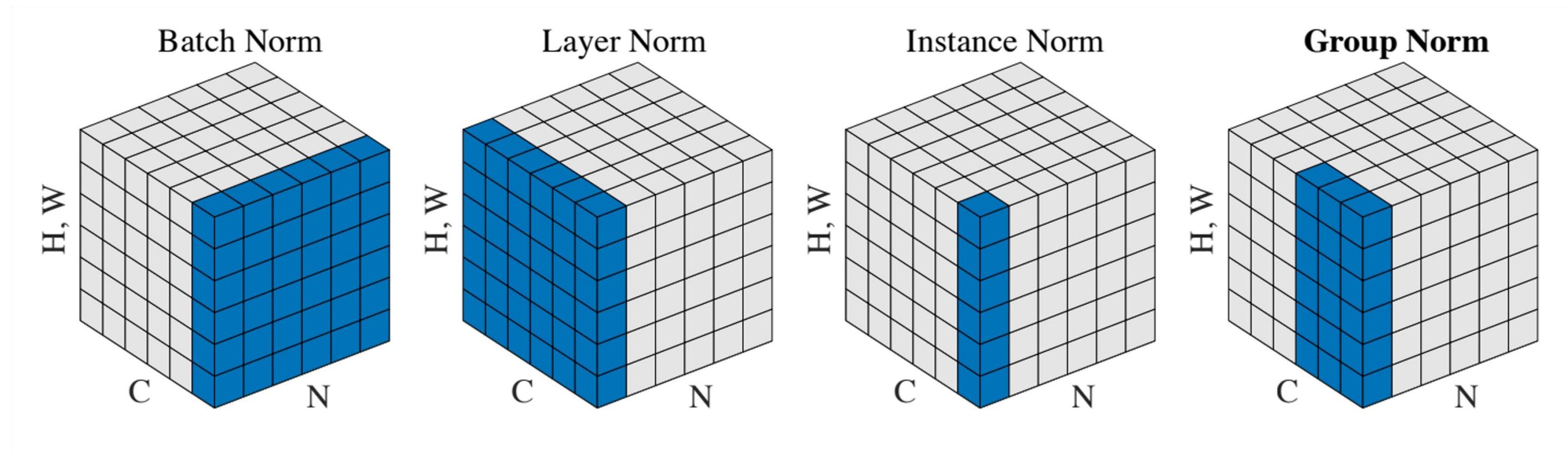
$$\mu, \sigma : N \times 1$$

$$\gamma, \beta : 1 \times D$$

$$y = \gamma(\mathbf{x} - \mu) / \sigma + \beta$$

Ba, Kiros, and Hinton, “Layer Normalization”, arXiv 2016

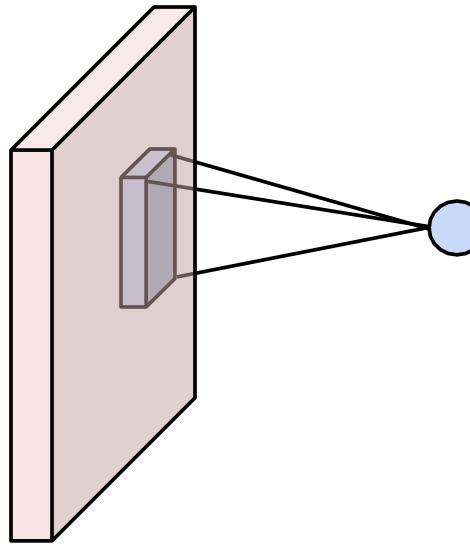
Other Normalization Layers



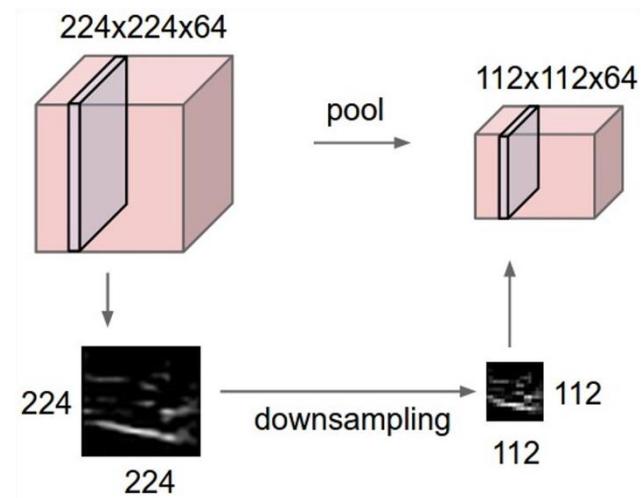
Wu and He, "Group Normalization", ECCV 2018

Components of CNNs

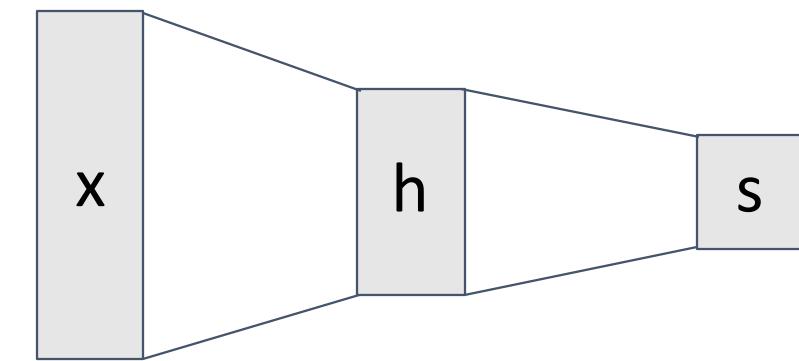
Convolution Layers



Pooling Layers



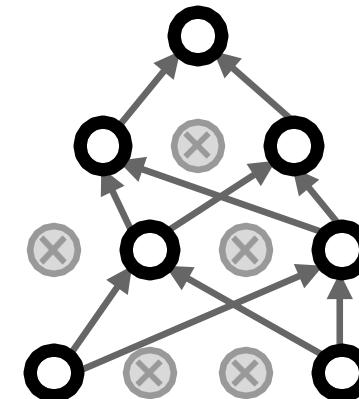
Fully-Connected Layers



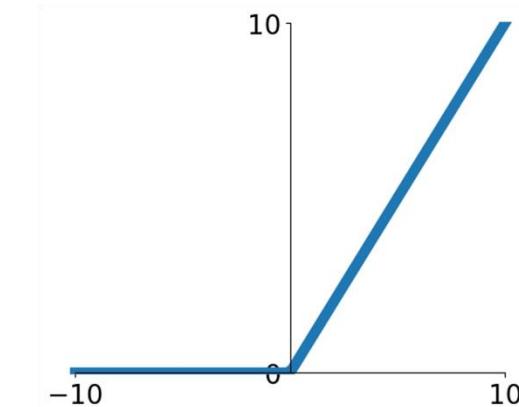
Normalization Layers

$$x_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Dropout (sometimes)

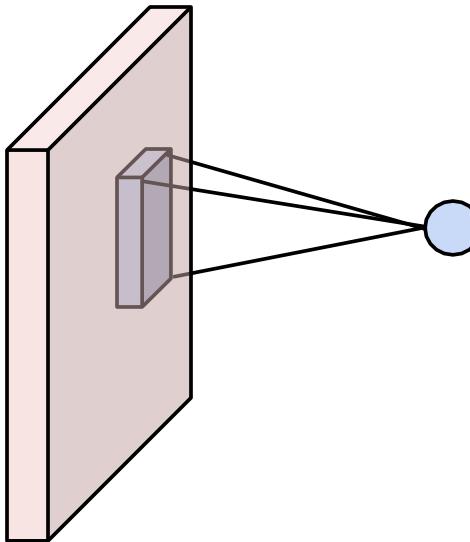


Activation Functions

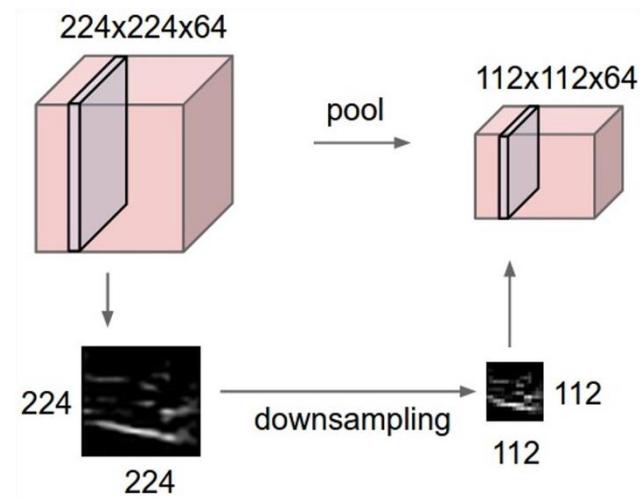


Components of CNNs

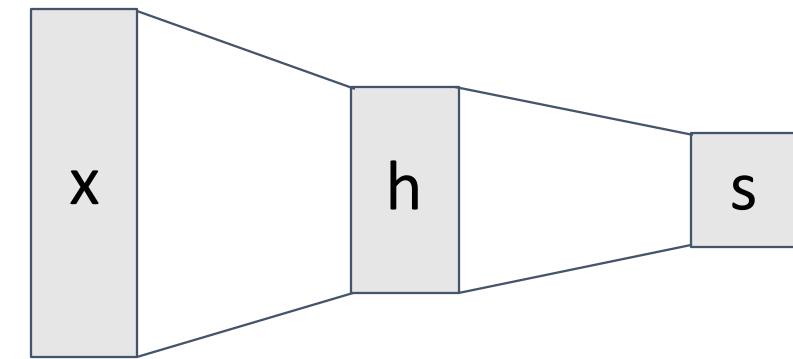
Convolution Layers



Pooling Layers



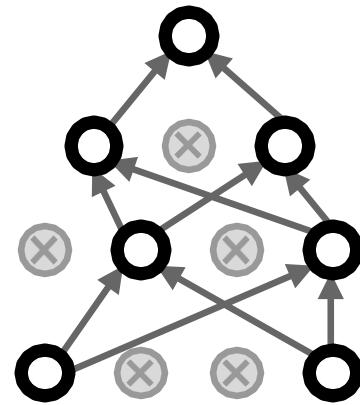
Fully-Connected Layers



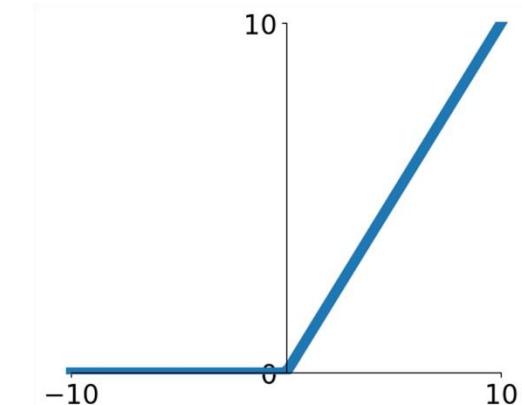
Normalization Layers

$$x_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Dropout (sometimes)



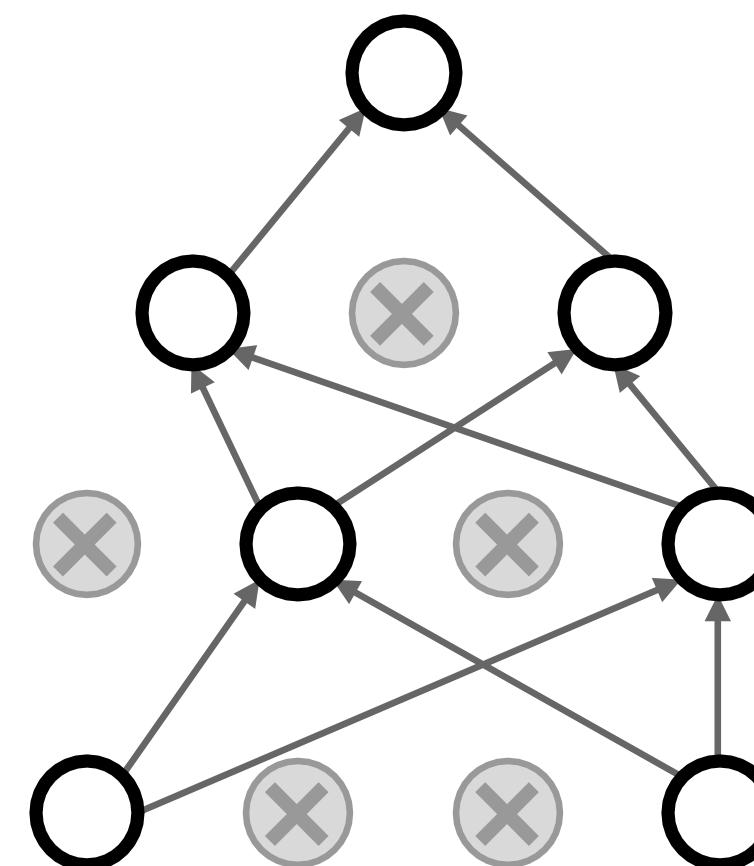
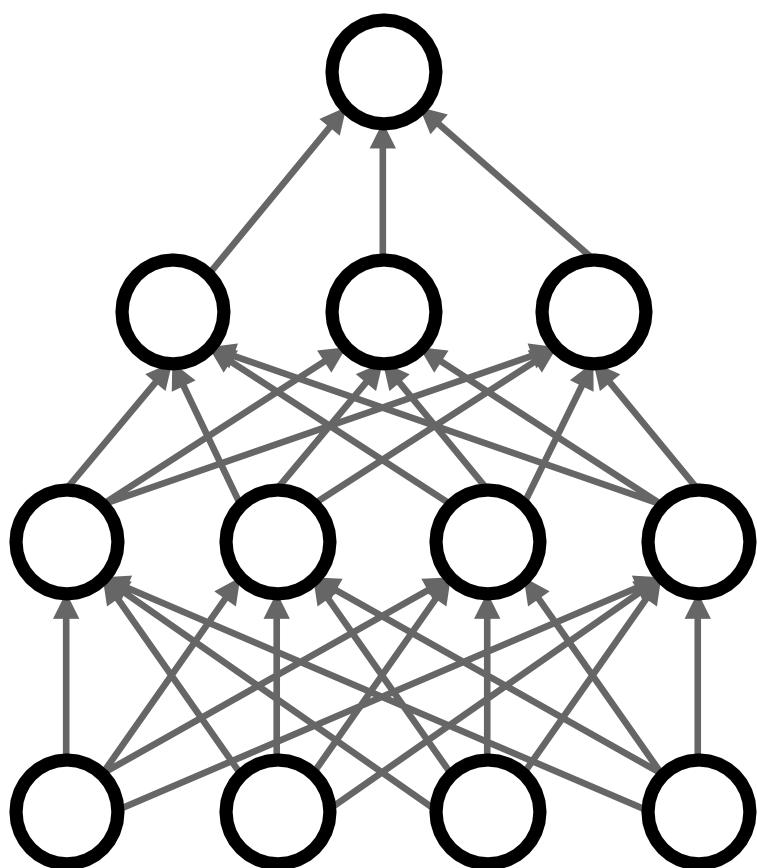
Activation Functions



Regularization: Dropout

In each forward pass, randomly set some neurons to zero

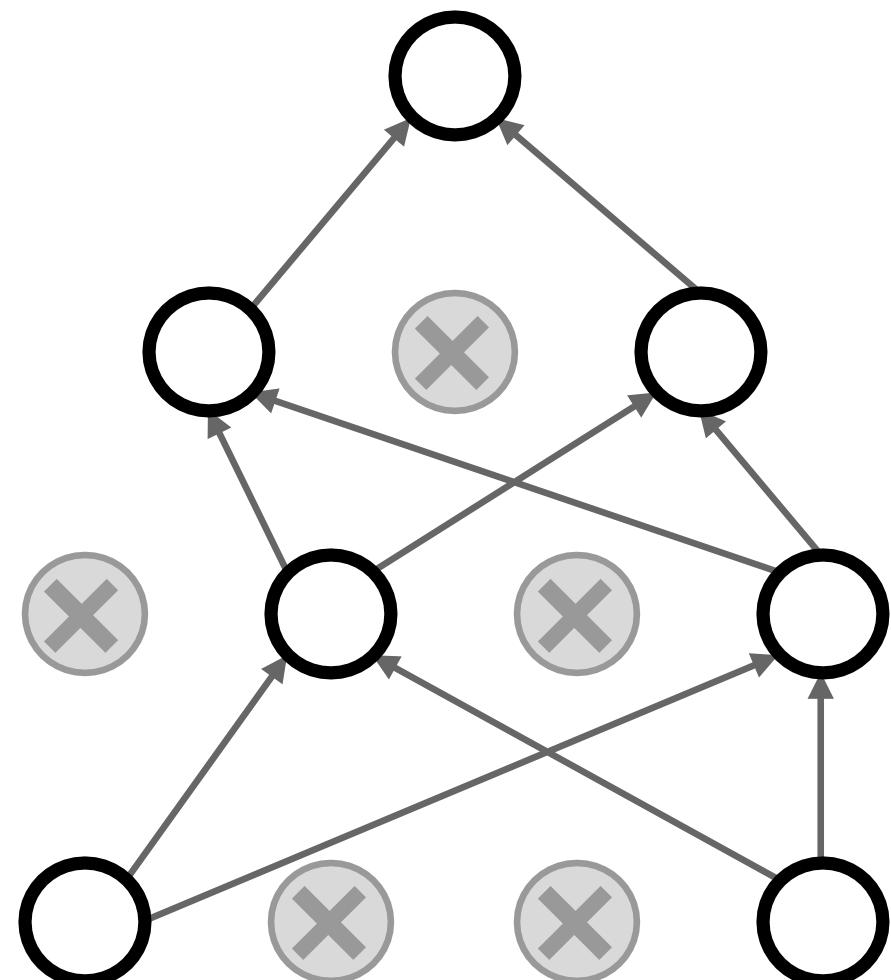
Probability of dropping is a hyperparameter; 0.5 is common



Srivastava et al, "Dropout: A simple way to prevent neural networks from overfitting", JMLR 2014

Regularization: Dropout

How can this possibly be a good idea?

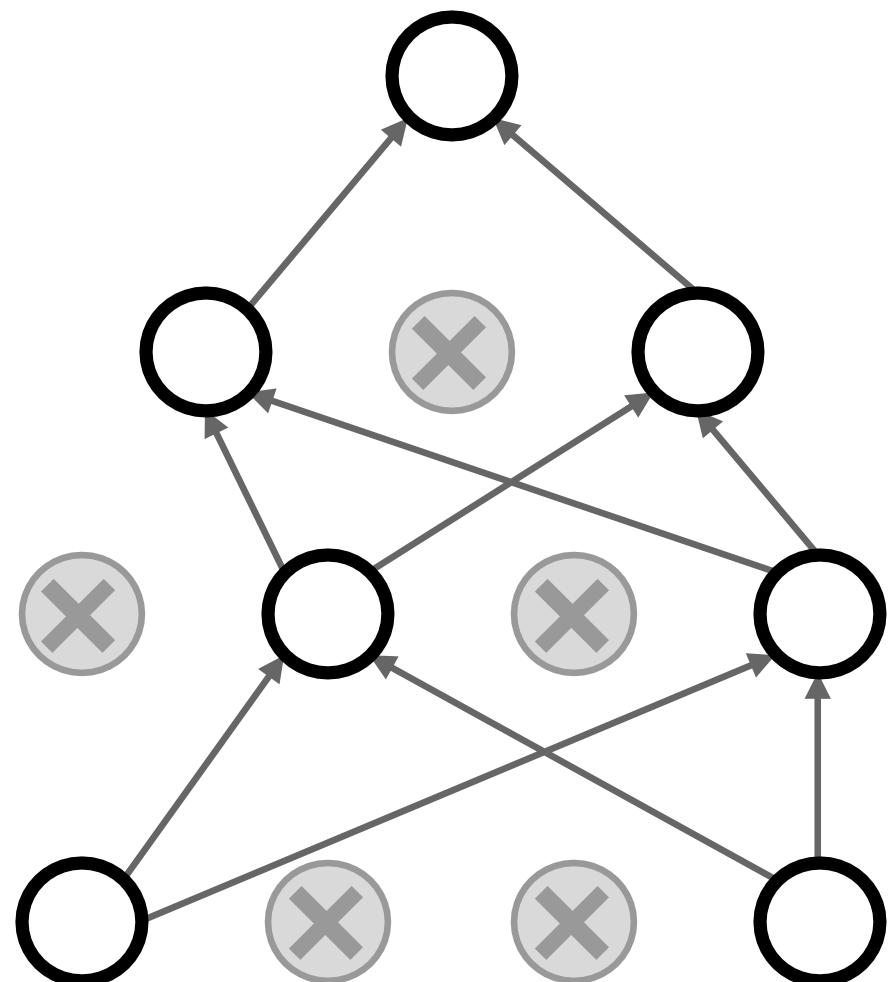


Forces the network to have a redundant representation;
Prevents co-adaptation of features



Regularization: Dropout

How can this possibly be a good idea?



Another interpretation:

Dropout is training a large **ensemble** of models (that share parameters).

Each binary mask is one model

An FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!

Only $\sim 10^{82}$ atoms in the universe...

Dropout: Test time

```
def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

At test time all neurons are active always

=> We must scale the activations so that for each neuron:

output at test time = expected output at training time

Dropout Summary

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

drop in train time

scale at test time

Lecture Overview – Two Broad Sets of Topics

How to build CNNs?

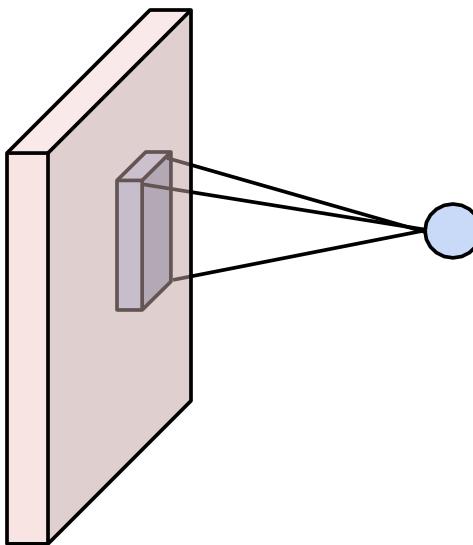
Layers in CNNs
Activation Functions
CNN Architectures
Weight Initialization

How to train CNNs?

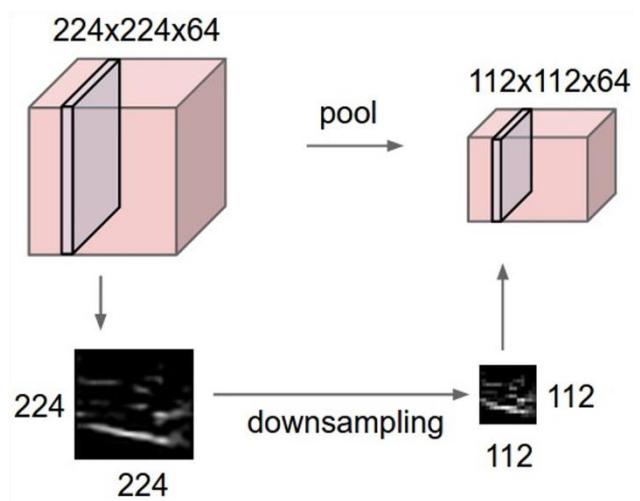
Data Preprocessing
Data augmentation
Transfer Learning
Hyperparameter Selection

Components of CNNs

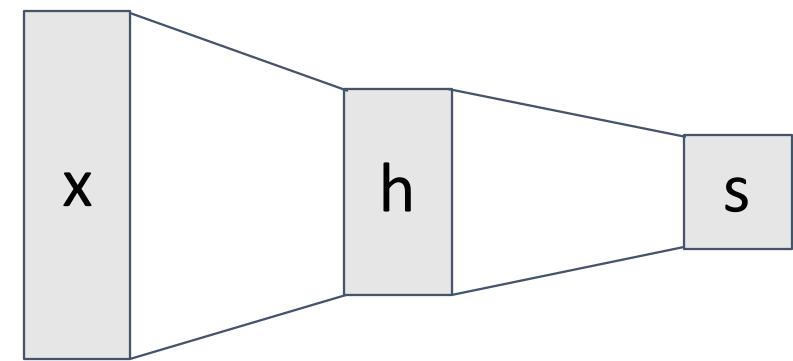
Convolution Layers



Pooling Layers



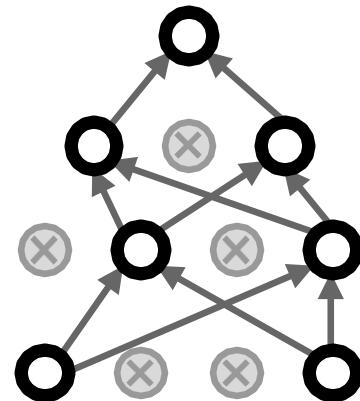
Fully-Connected Layers



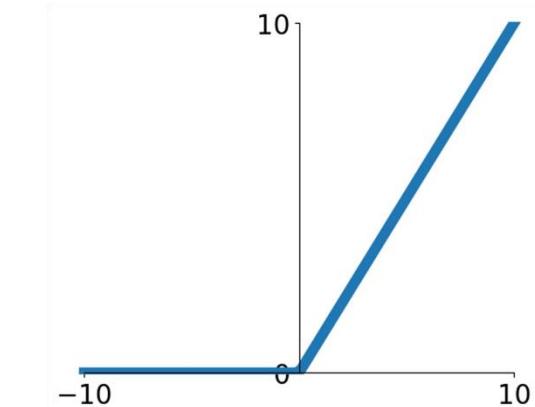
Normalization Layers

$$x_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Dropout (sometimes)

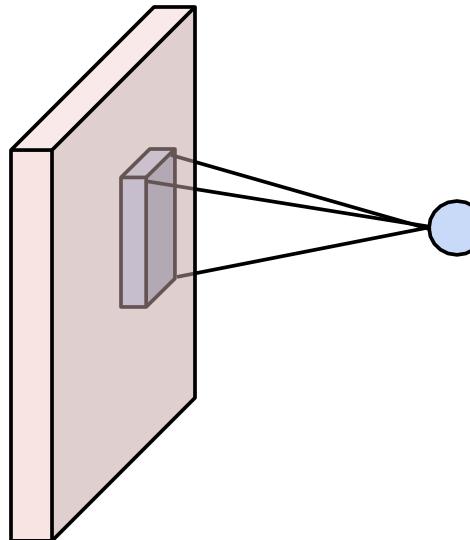


Activation Functions

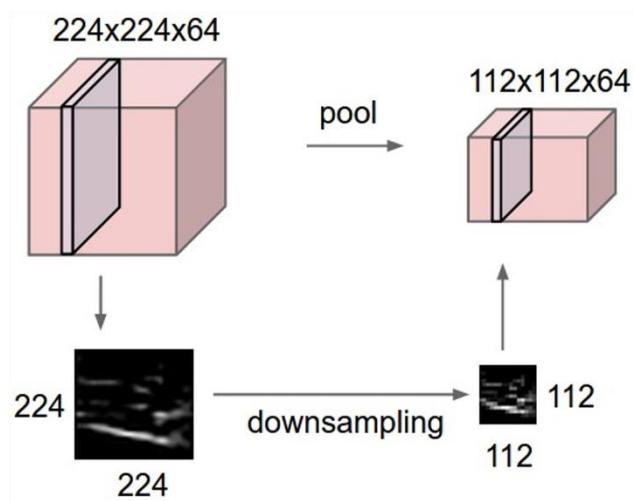


Components of CNNs

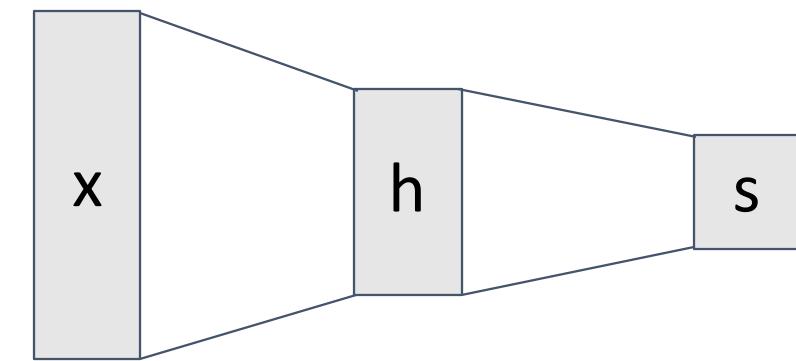
Convolution Layers



Pooling Layers



Fully-Connected Layers



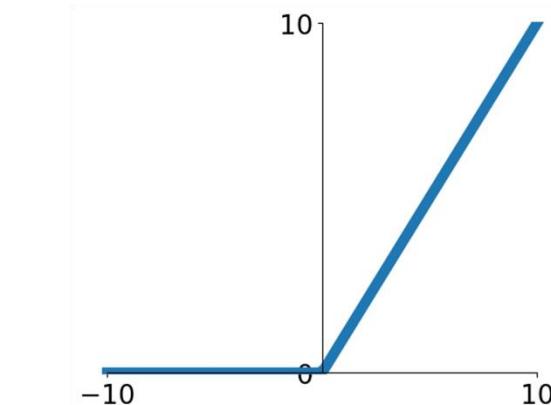
Normalization Layers

$$x_{ij} = \frac{x_{ij}}{\sigma_j} + \varepsilon$$

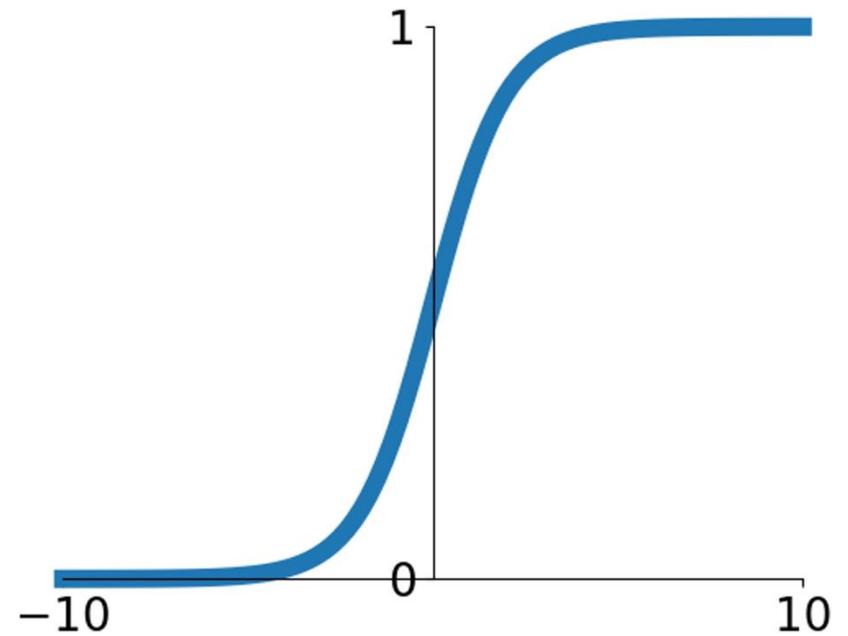
Dropout (sometimes)

Goal: Introduce non-linearities to our model!

Activation Functions



Activation Functions



Sigmoid

$$\sigma(x) = 1/(1 + e^{-x})$$

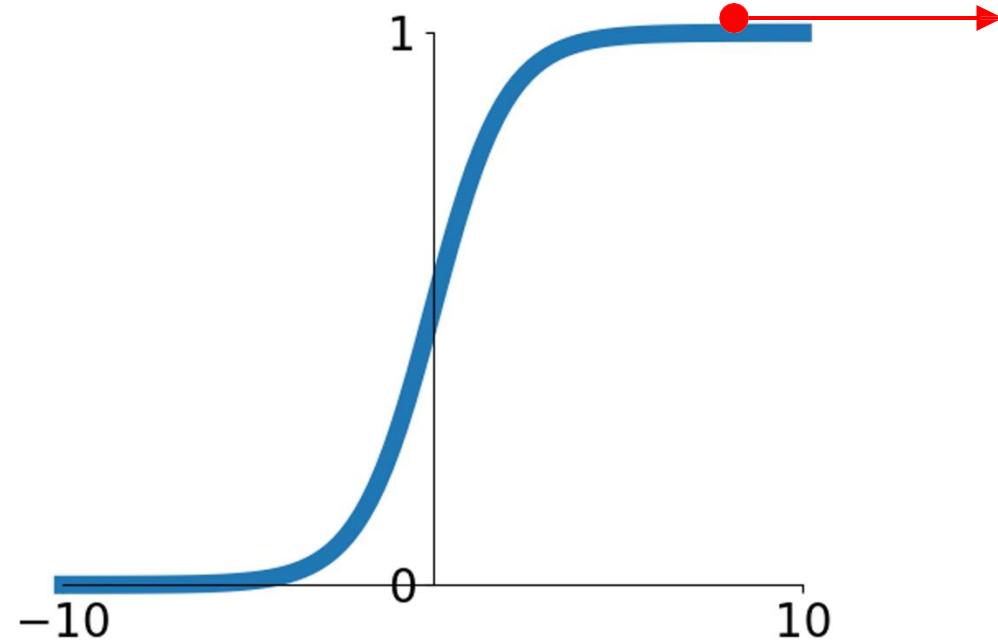
- Squashes numbers to range [0, 1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

Key problem:

Many layers of sigmoids → smaller and smaller gradients.

Q: In which regions does sigmoid have a small gradient?

Activation Functions



Sigmoid

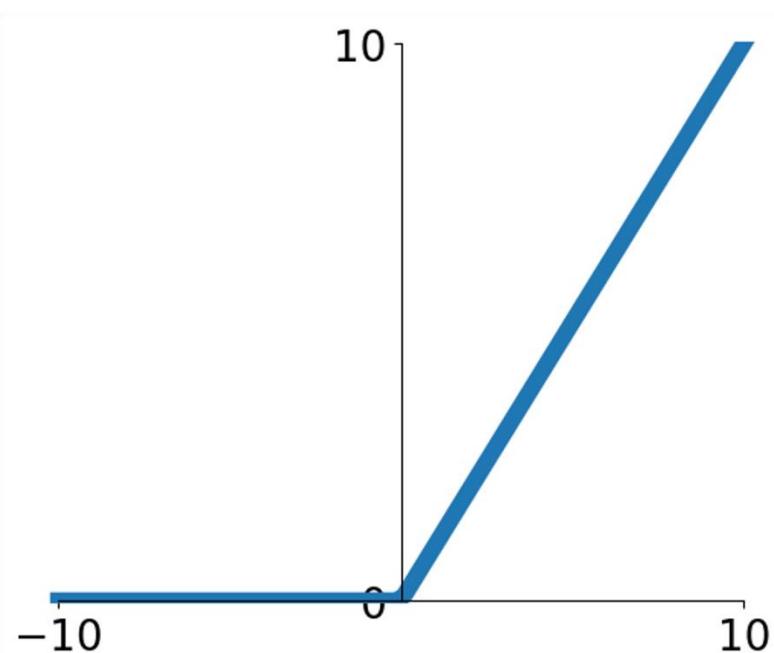
$$\sigma(x) = 1/(1 + e^{-x})$$

- Squashes numbers to range [0, 1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

Key problem:

Large positive or negative values can “kill” the gradients. Many layers of sigmoids → smaller and smaller gradients in practice

Activation Functions

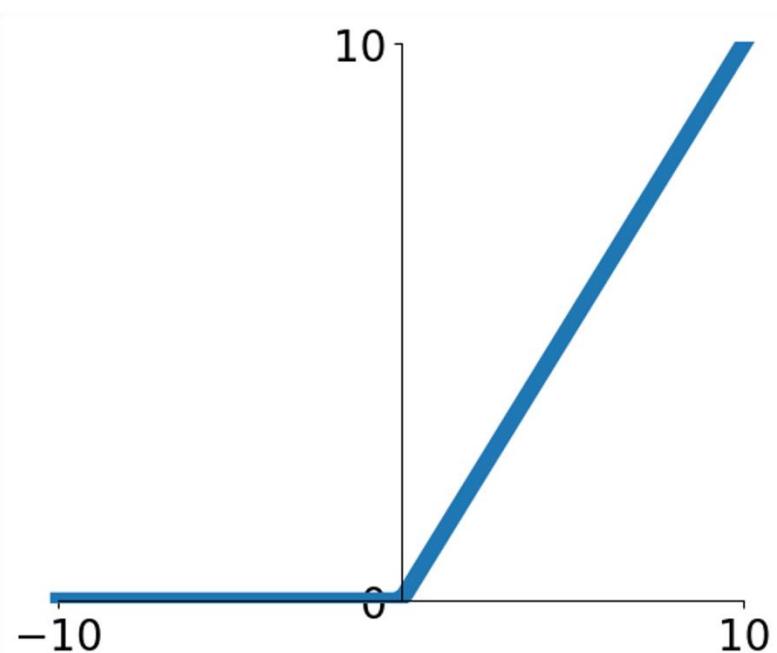


- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid in practice (e.g. 6x)

ReLU
(Rectified Linear Unit)

[Krizhevsky et al., 2012]

Activation Functions



ReLU
(Rectified Linear Unit)

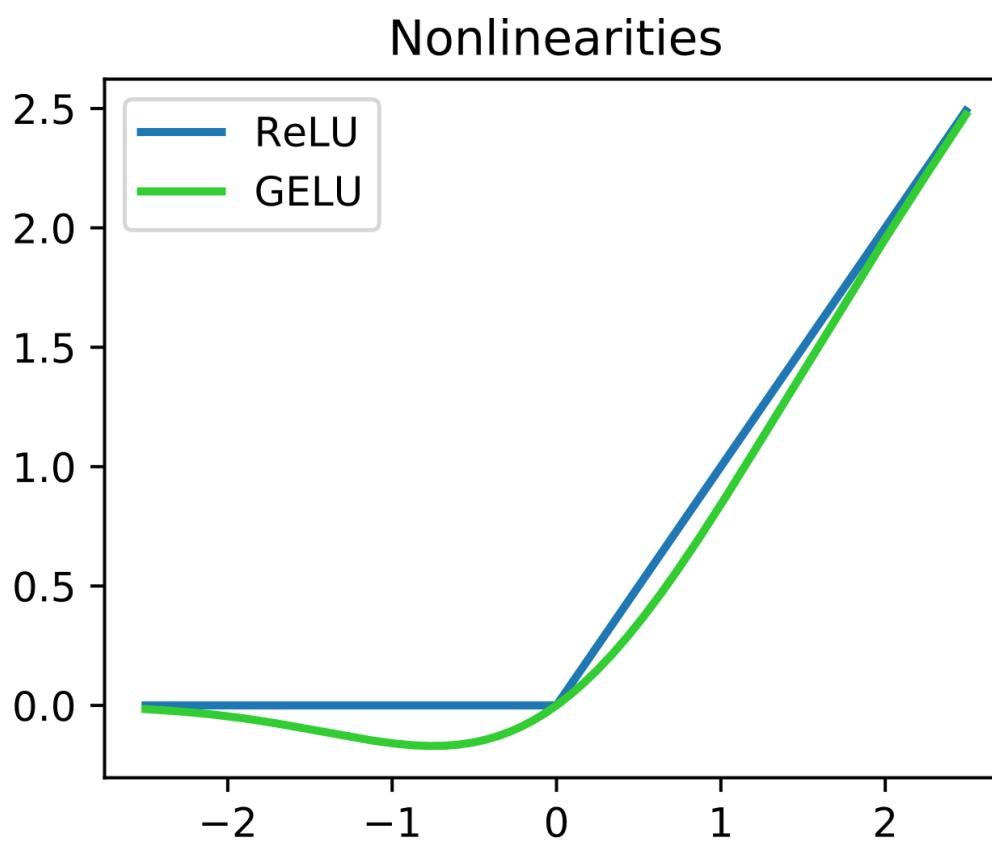
- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid in practice (e.g. 6x)

- Not zero-centered output
- An annoyance:

Dead ReLUs when $x < 0$!

Activation Functions

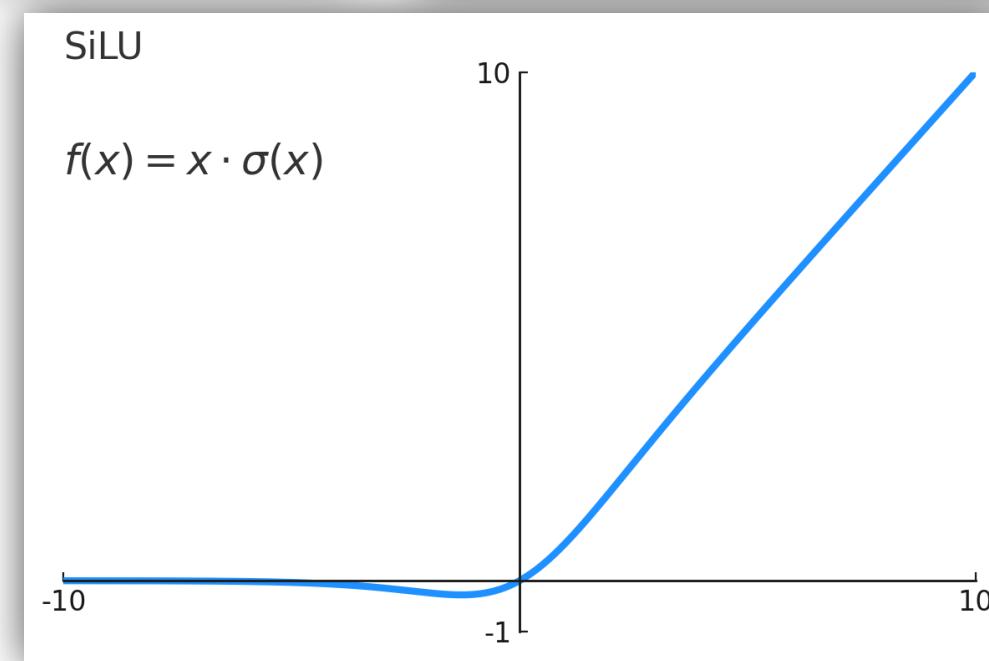
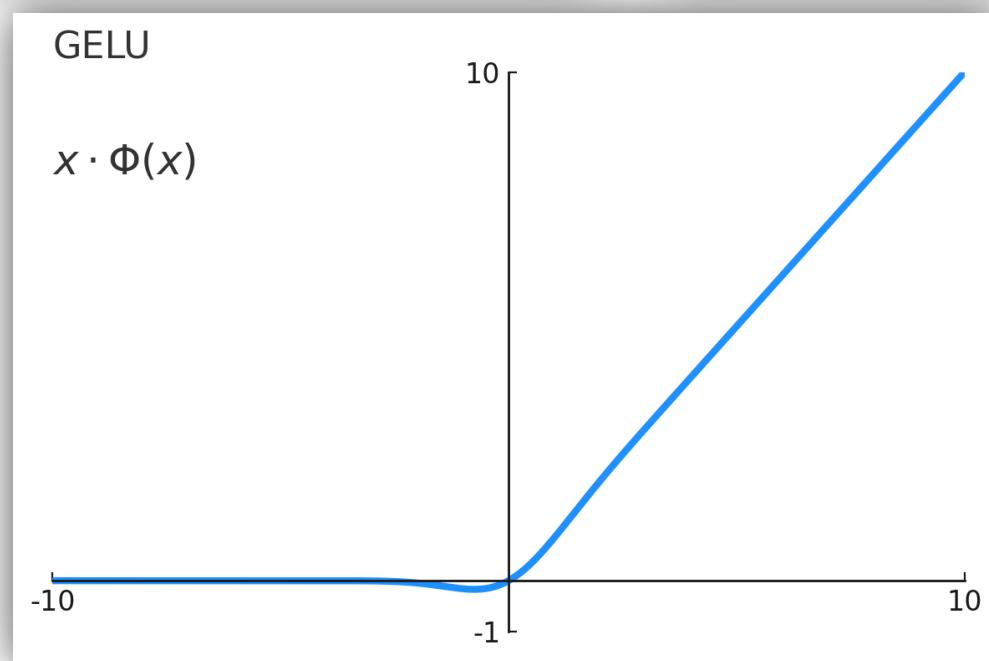
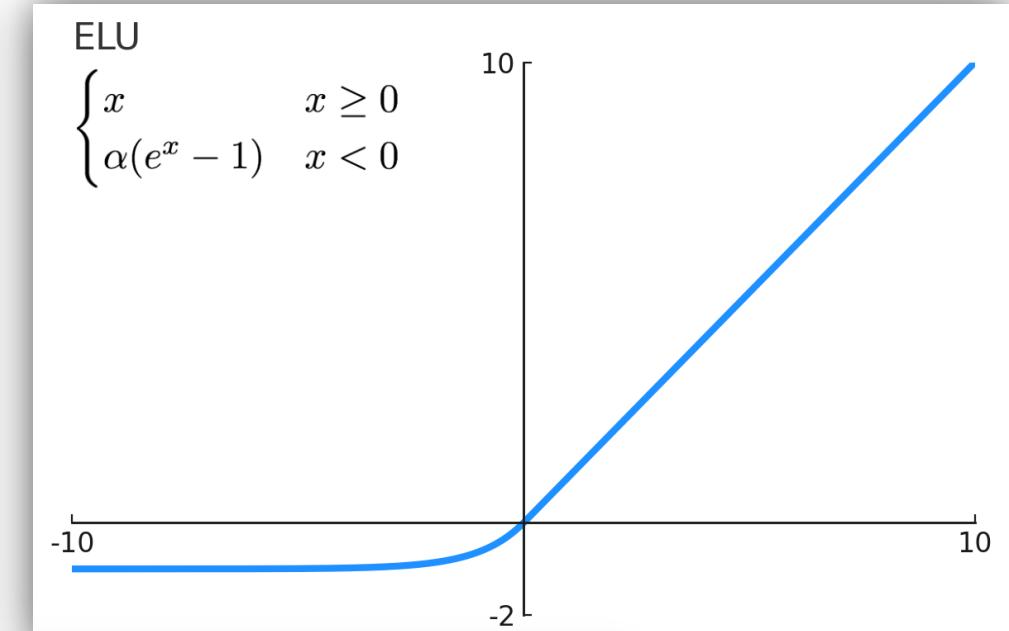
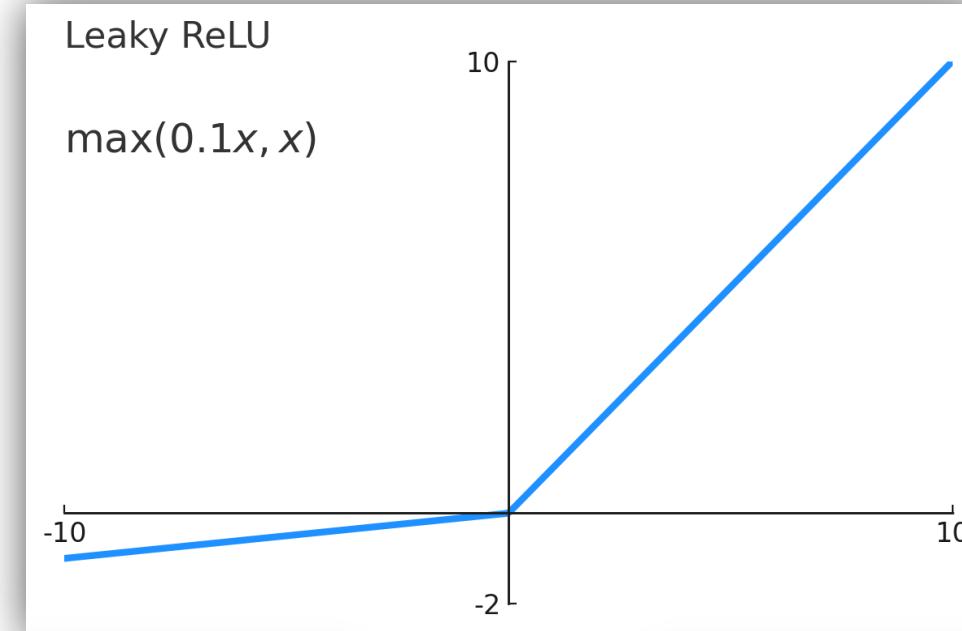
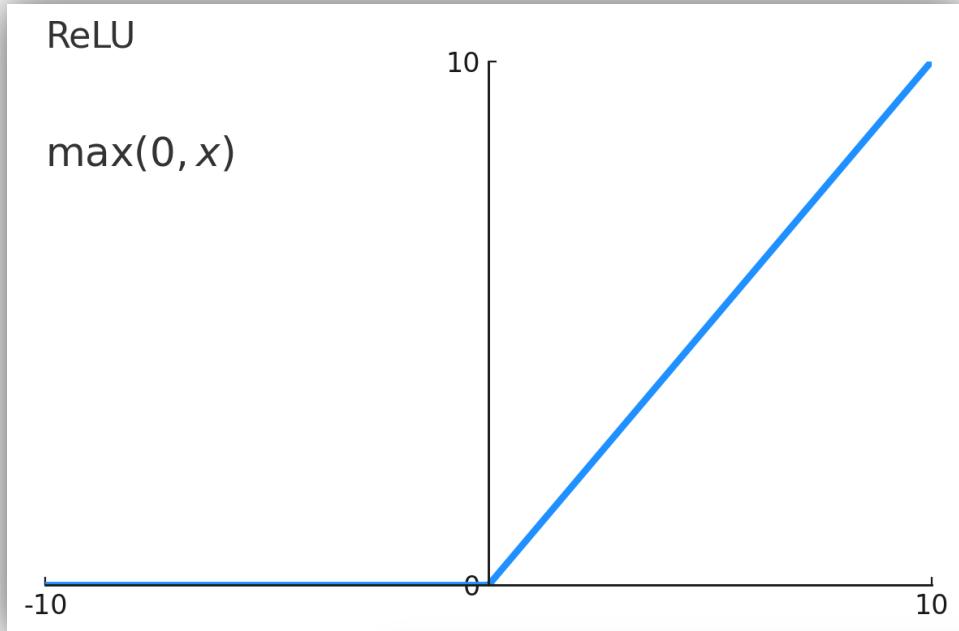
[Hendrycks et al., 2016]



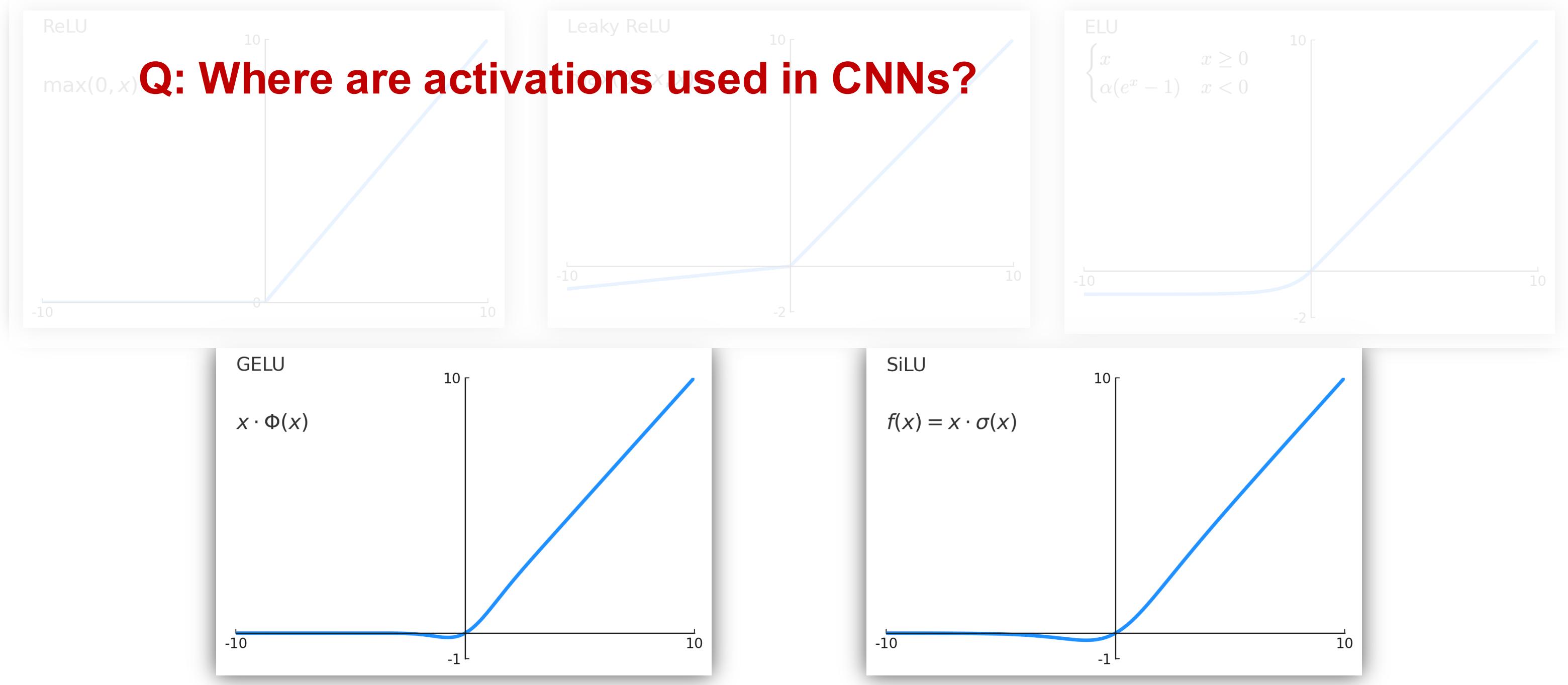
- Computes $f(x) = x^* \Phi(x)$
- Very nice behavior around 0
- Smoothness facilitates training in practice
- Higher computational cost than ReLU
- Large negative values can still have gradient $\rightarrow 0$

GELU
(Gaussian Error
Linear Unit)

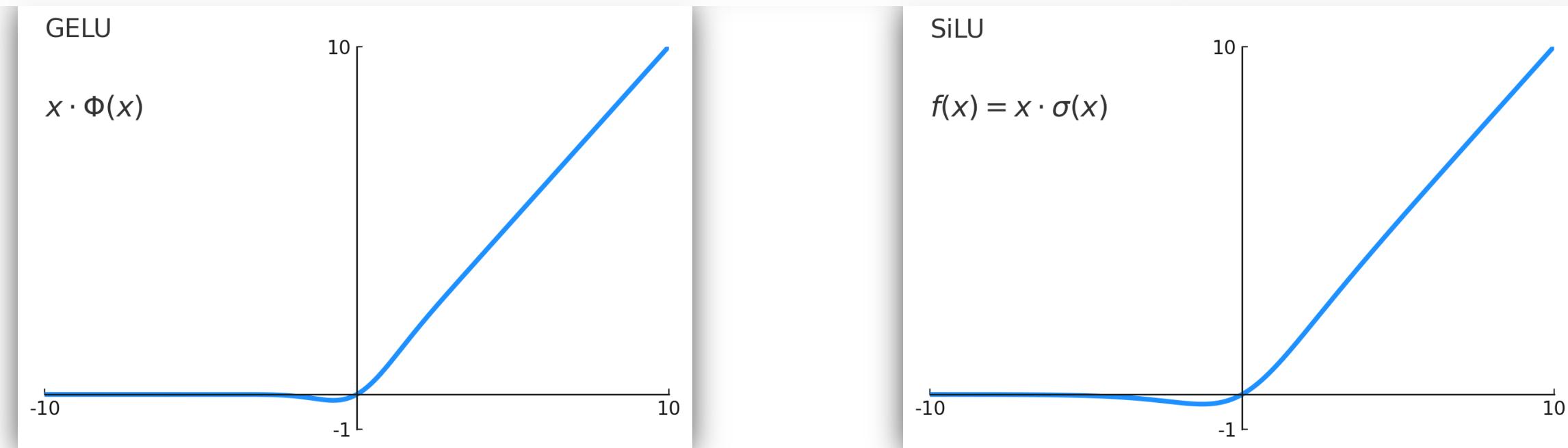
Activation Function Zoo



Activation Function Zoo



Activation Function Zoo



Lecture Overview – Two Broad Sets of Topics

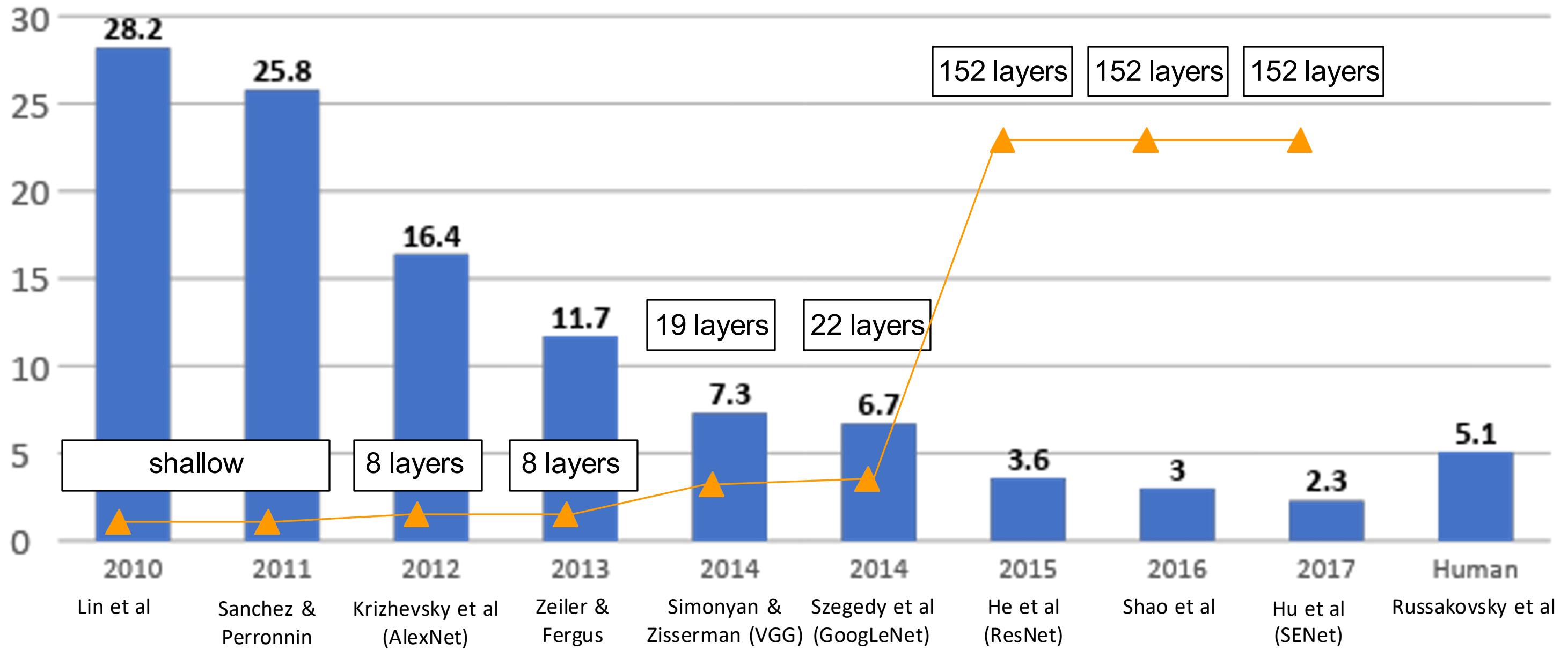
How to build CNNs?

Layers in CNNs
Activation Functions
CNN Architectures
Weight Initialization

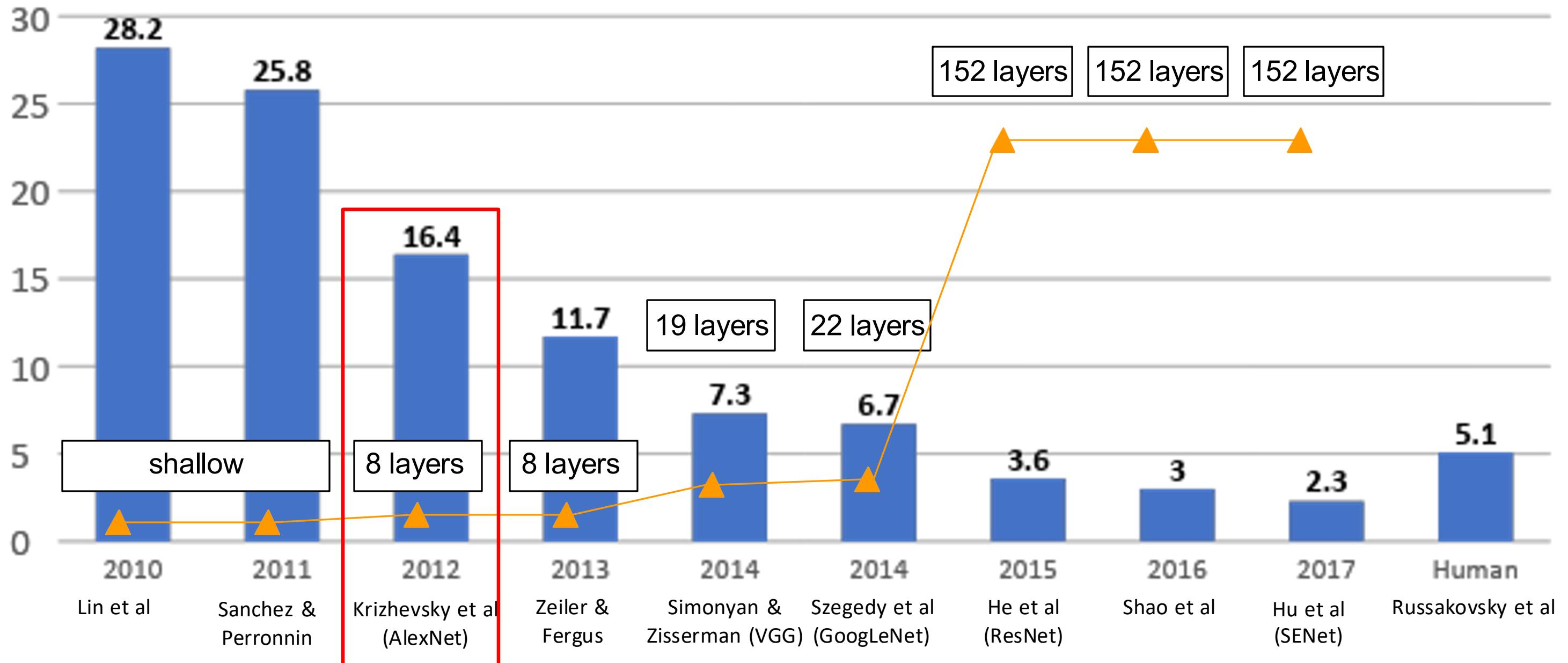
How to train CNNs?

Data Preprocessing
Data augmentation
Transfer Learning
Hyperparameter Selection

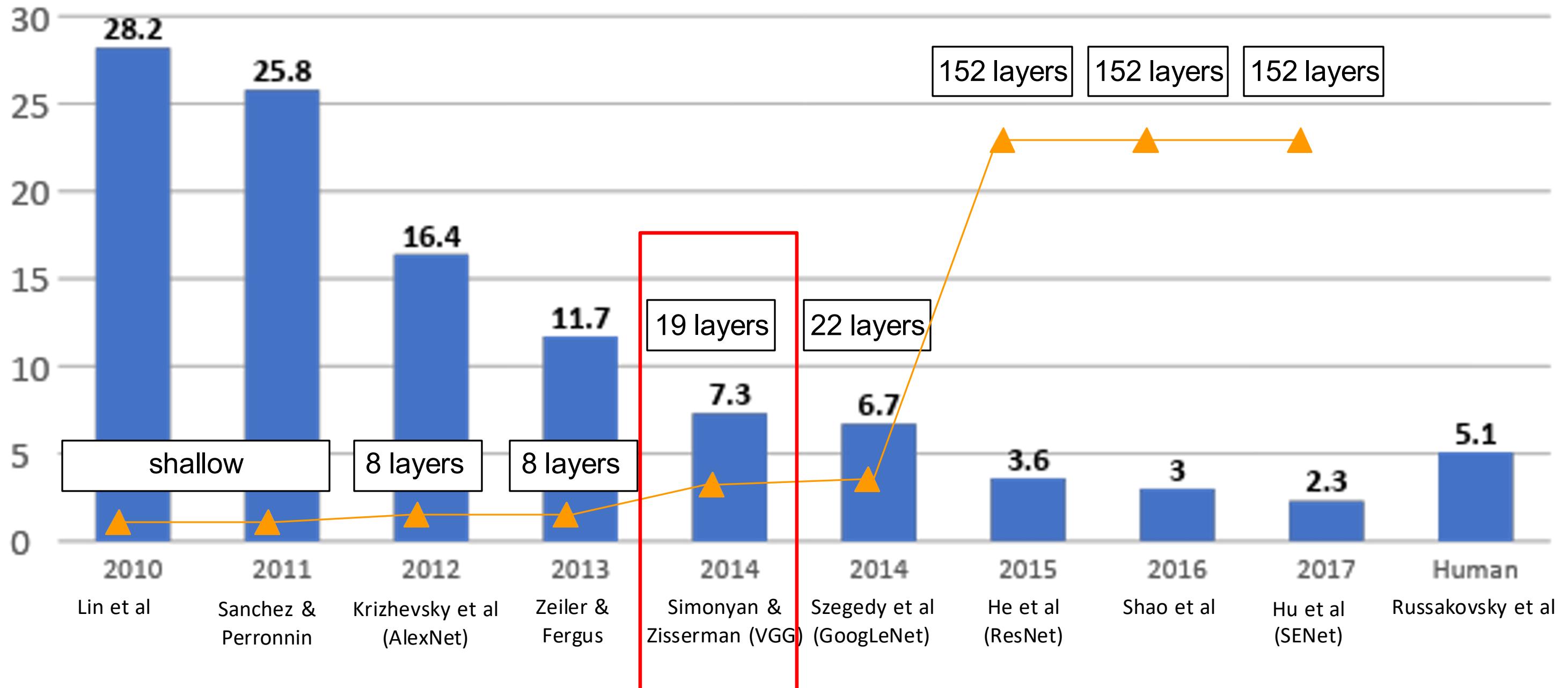
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Small filters, Deeper networks

8 layers (AlexNet)

-> 16 - 19 layers (VGG16Net)

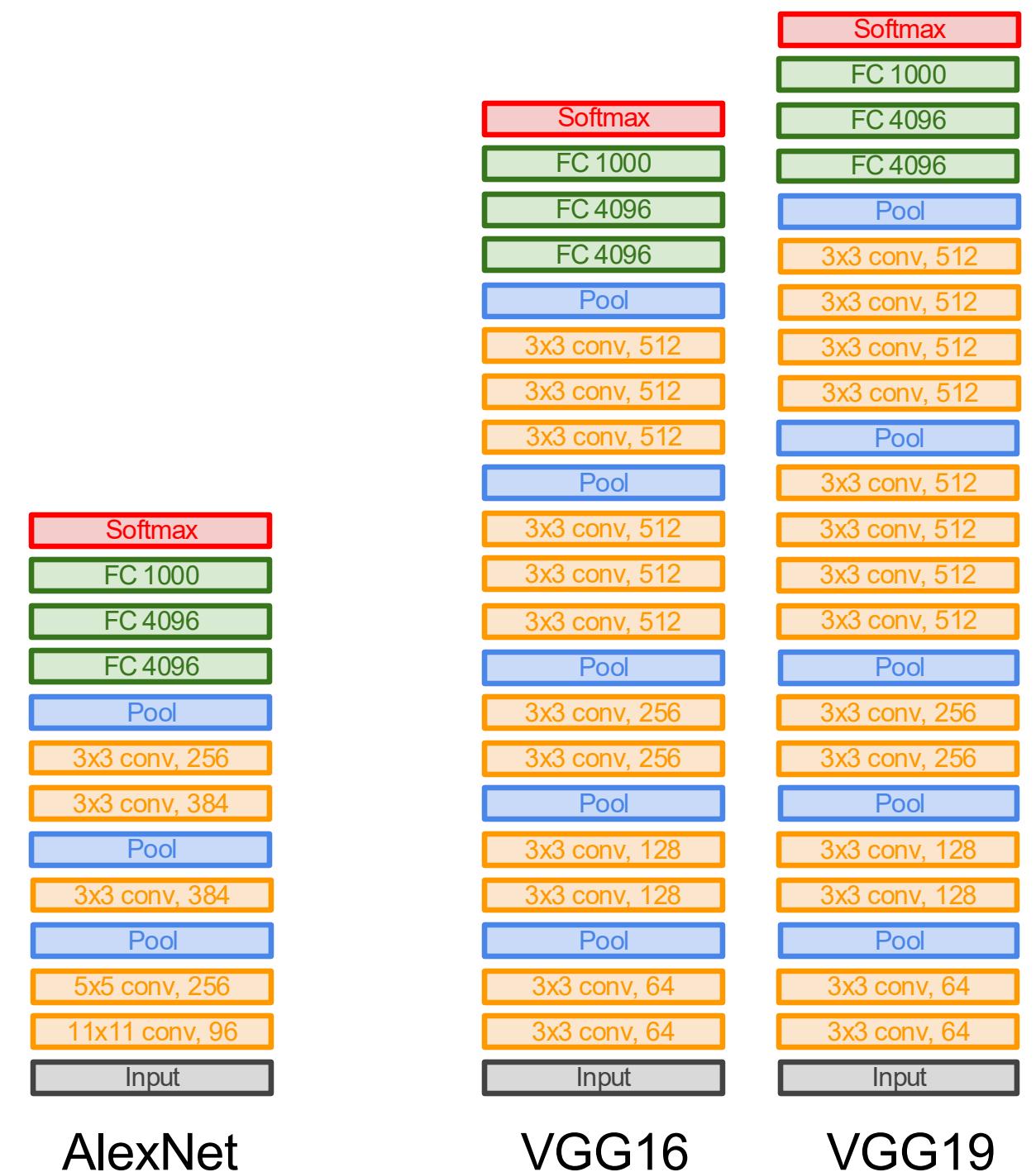
Only 3x3 CONV stride 1, pad 1

and 2x2 MAX POOL stride 2

11.7% top 5 error in ILSVRC'13

(ZFNet)

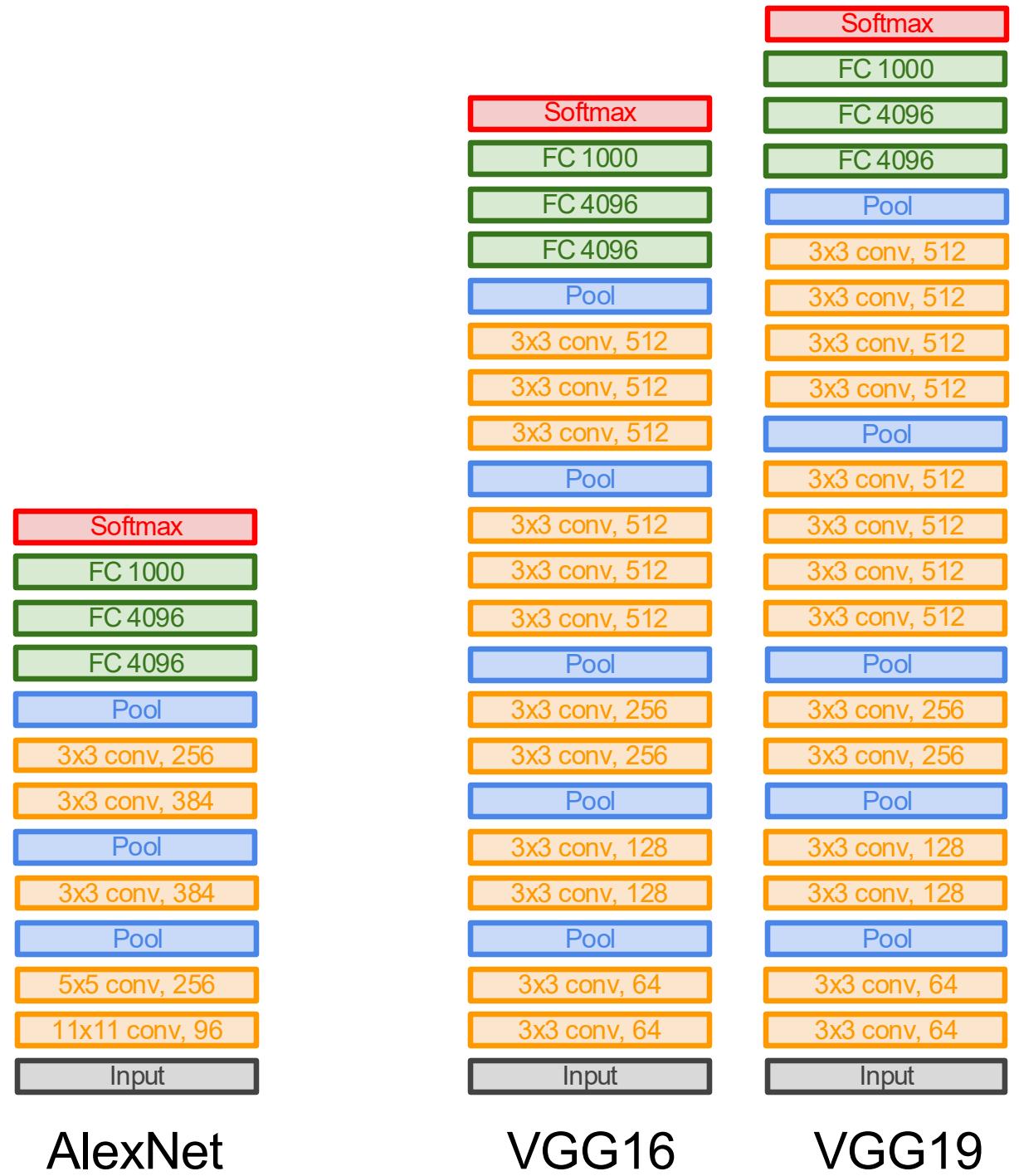
-> 7.3% top 5 error in ILSVRC'14



Case Study: VGGNet

[Simonyan and Zisserman, 2014]

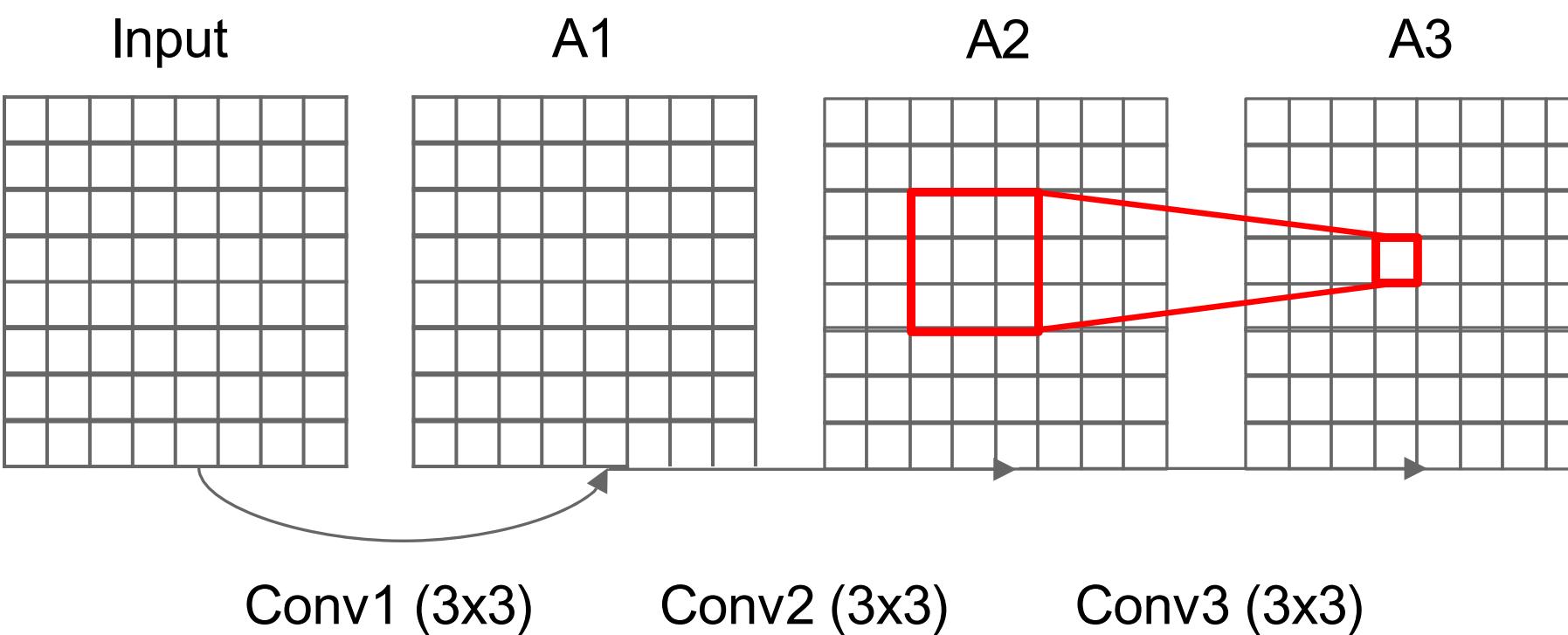
Q: Why use smaller filters? (3x3 conv)



Case Study: VGGNet

[Simonyan and Zisserman, 2014]

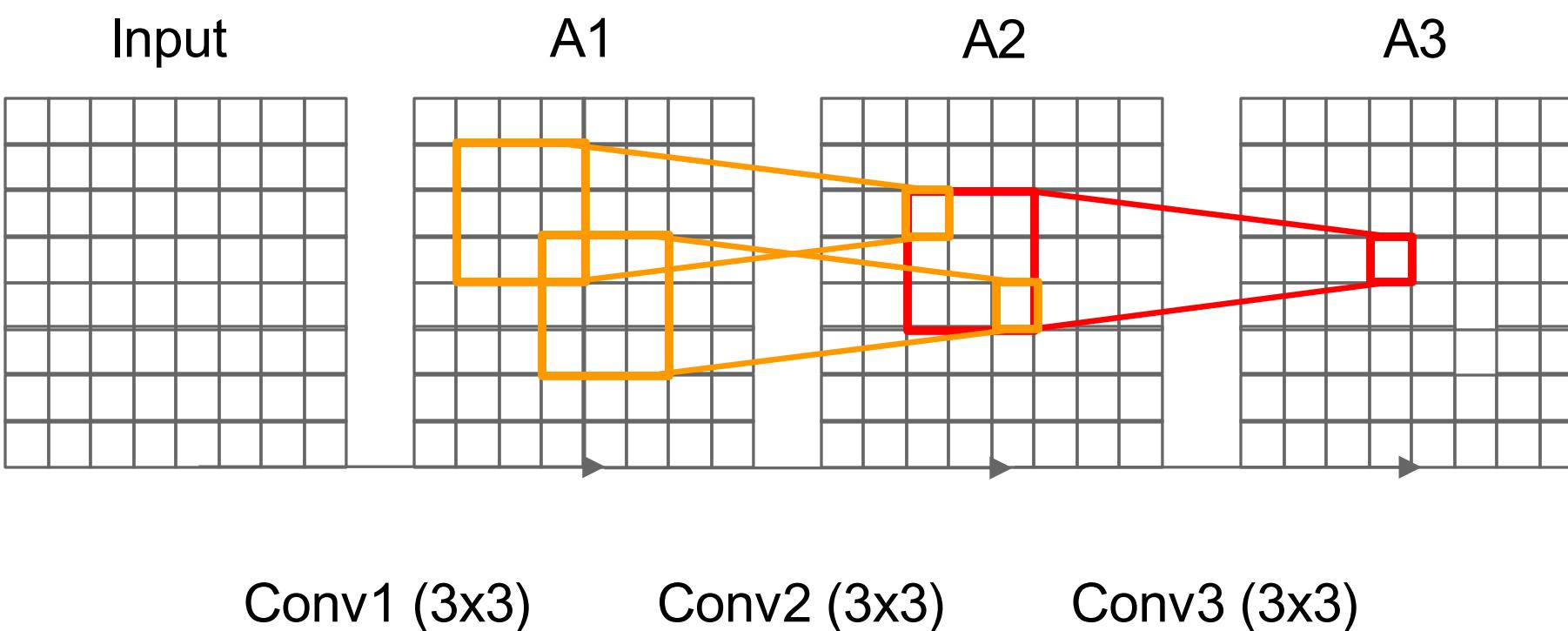
Q: What is the effective receptive field of three 3x3 conv (stride 1) layers?



Case Study: VGGNet

[Simonyan and Zisserman, 2014]

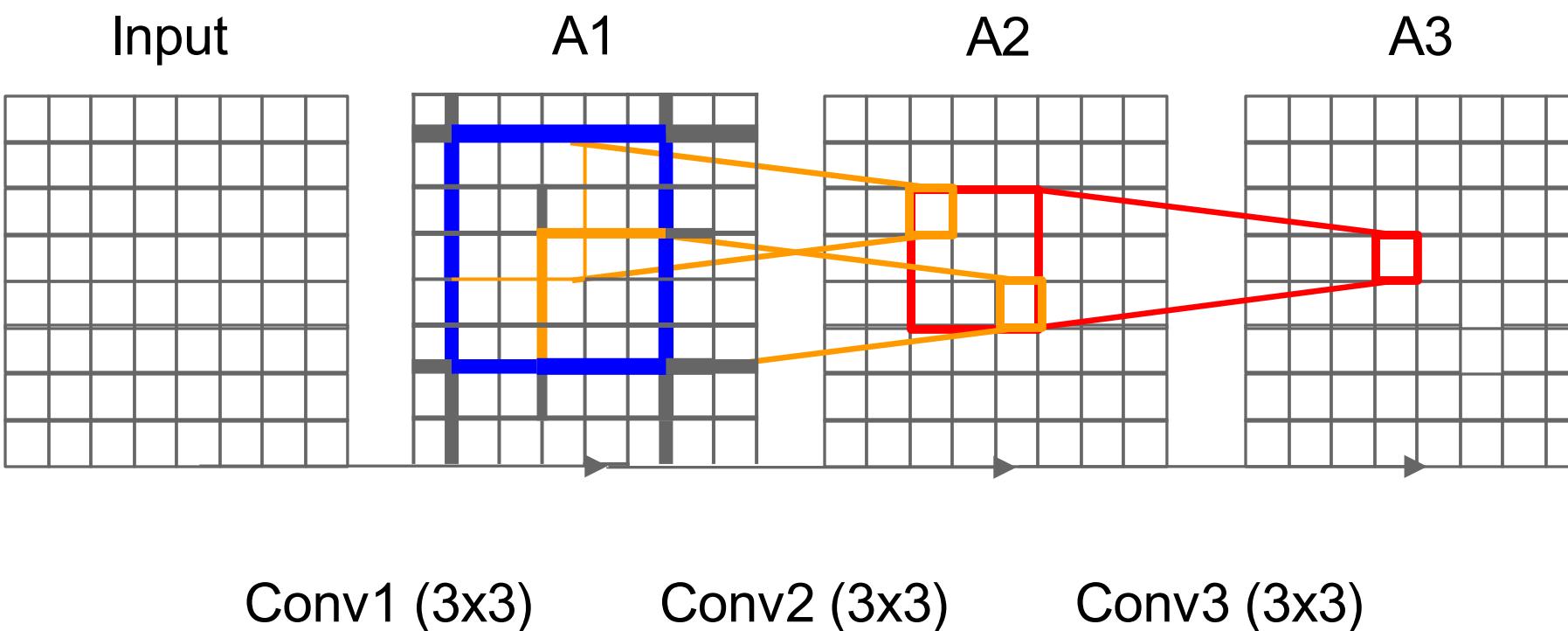
Q: What is the effective receptive field
of three 3x3 conv (stride 1) layers?



Case Study: VGGNet

[Simonyan and Zisserman, 2014]

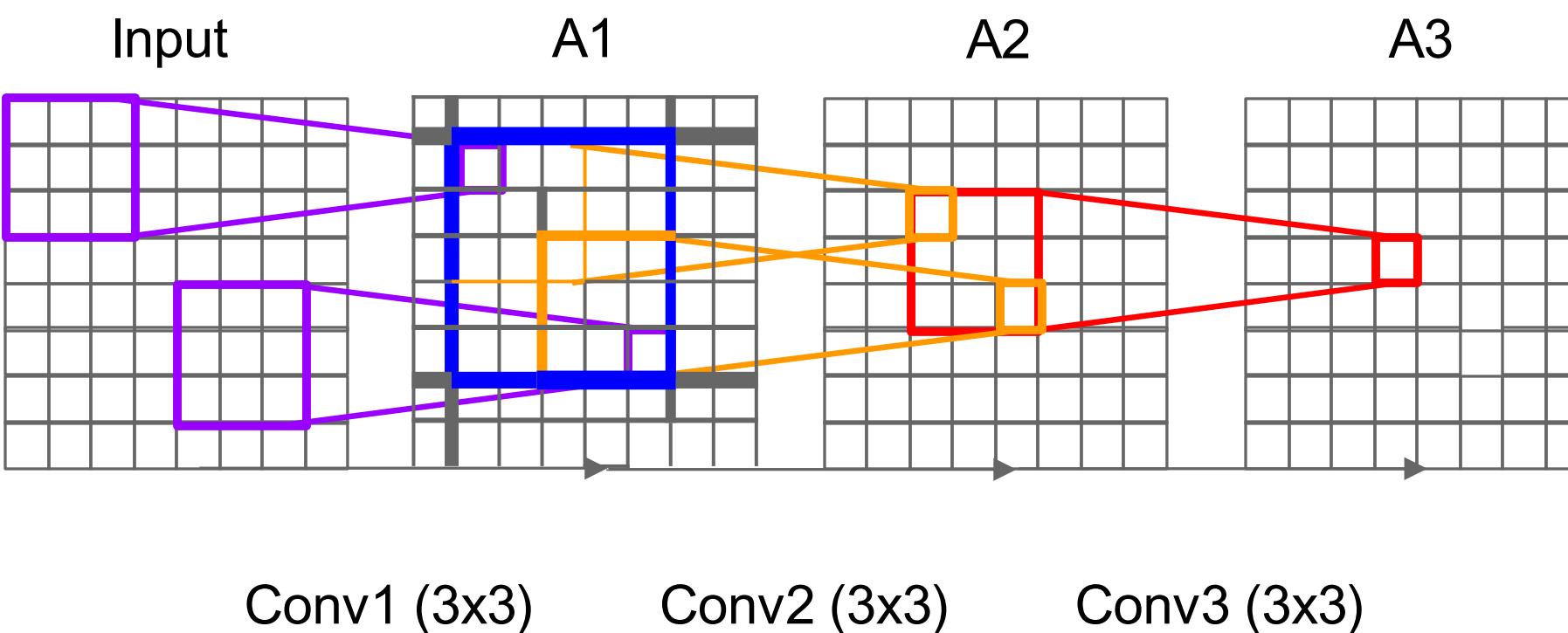
Q: What is the effective receptive field
of three 3x3 conv (stride 1) layers?



Case Study: VGGNet

[Simonyan and Zisserman, 2014]

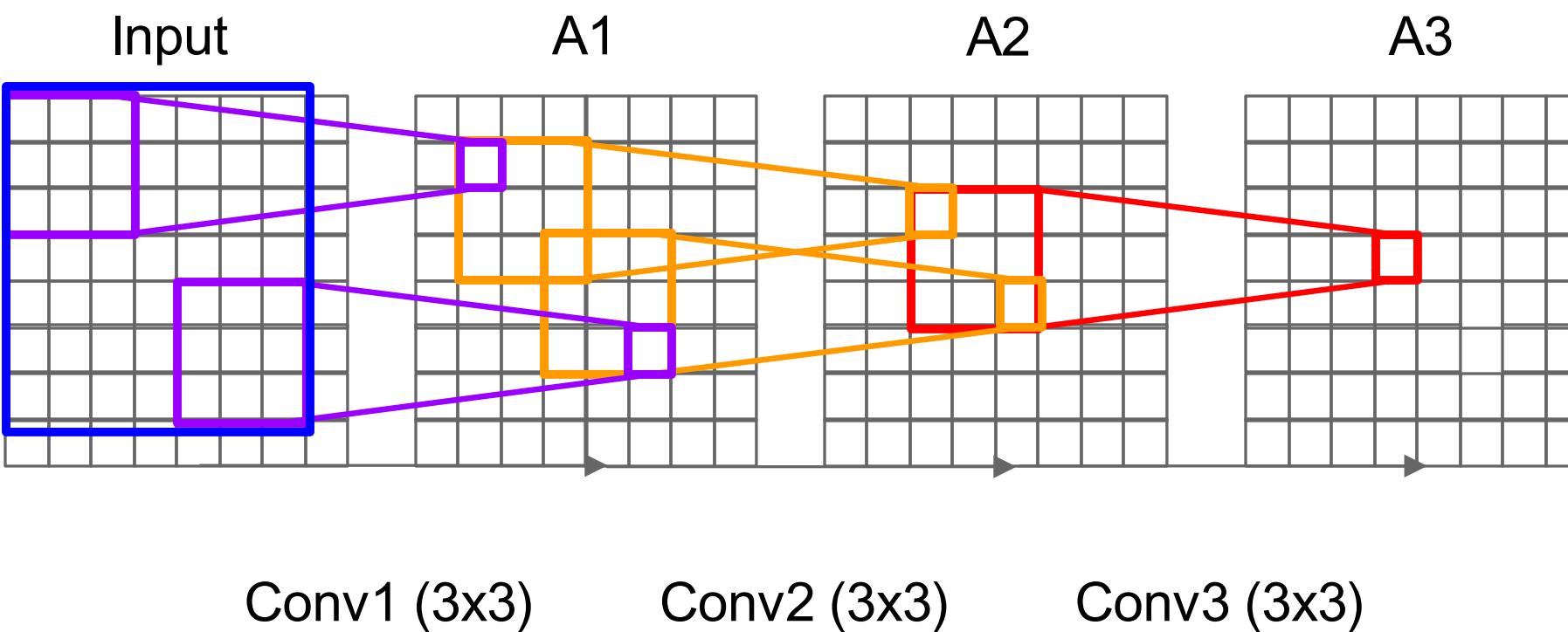
Q: What is the effective receptive field
of three 3x3 conv (stride 1) layers?



Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: What is the effective receptive field
of three 3x3 conv (stride 1) layers?



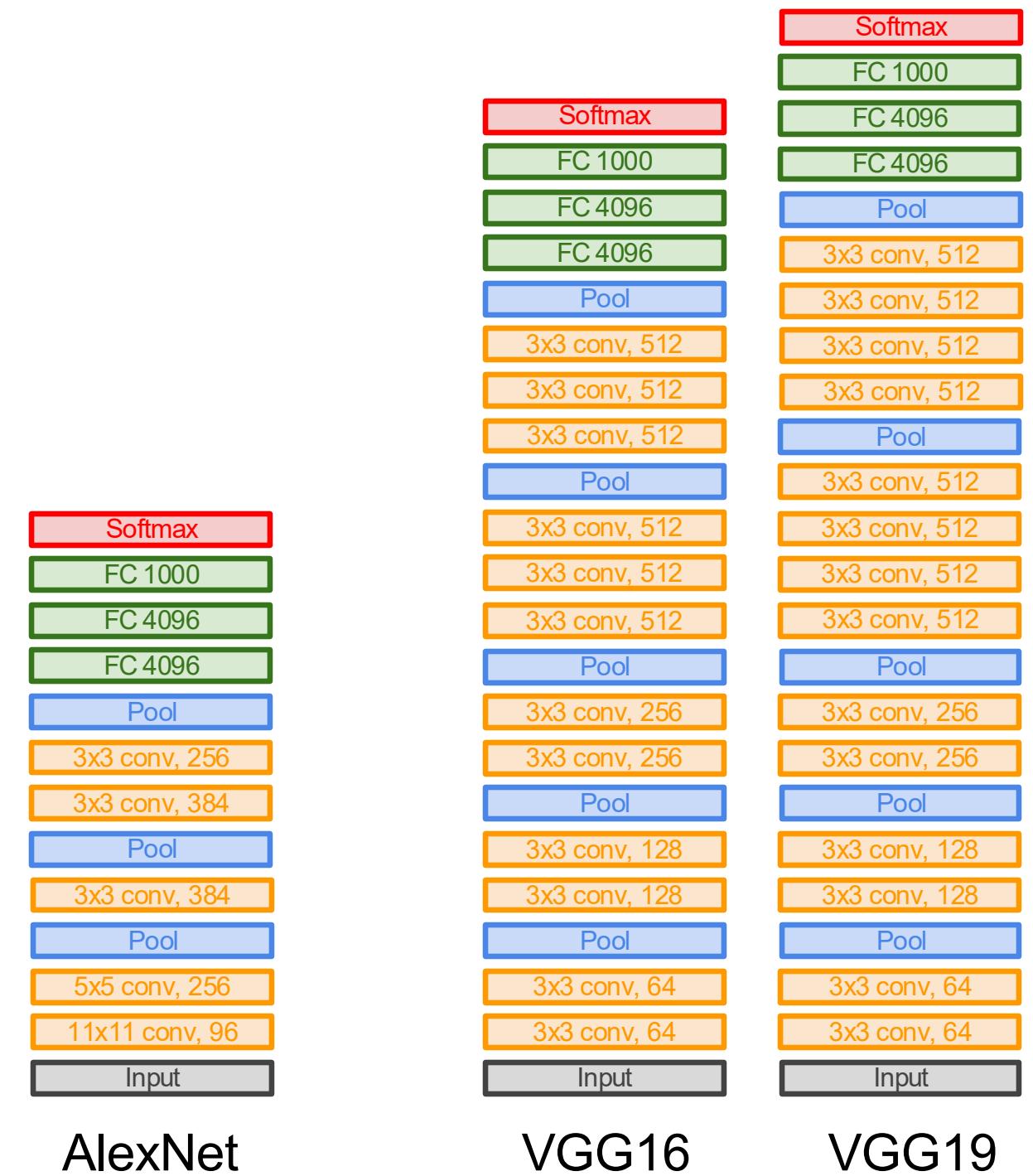
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

Stack of three 3x3 conv (stride 1) layers
has same **effective receptive field** as
one 7x7 conv layer

[7x7]



Case Study: VGGNet

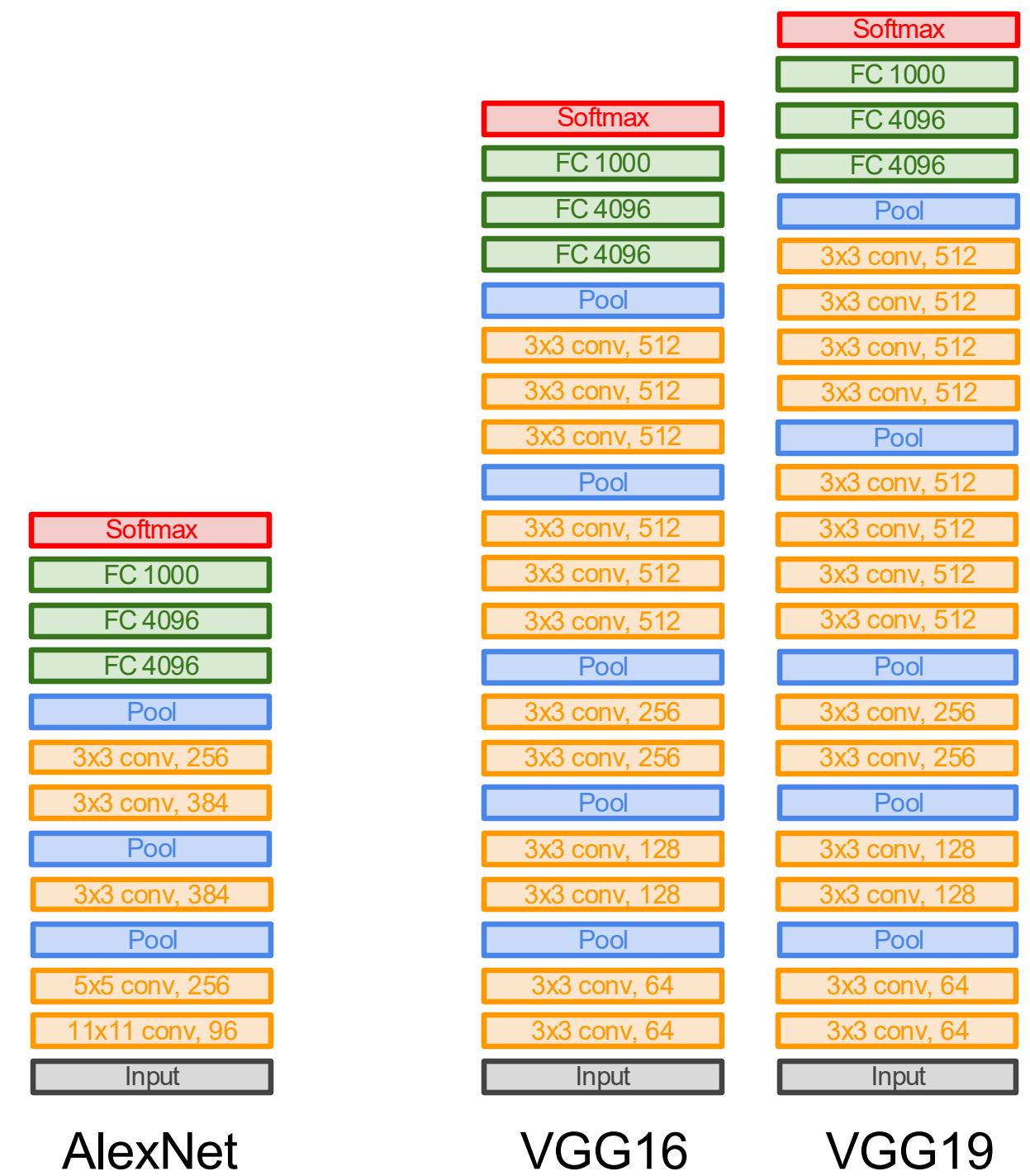
[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

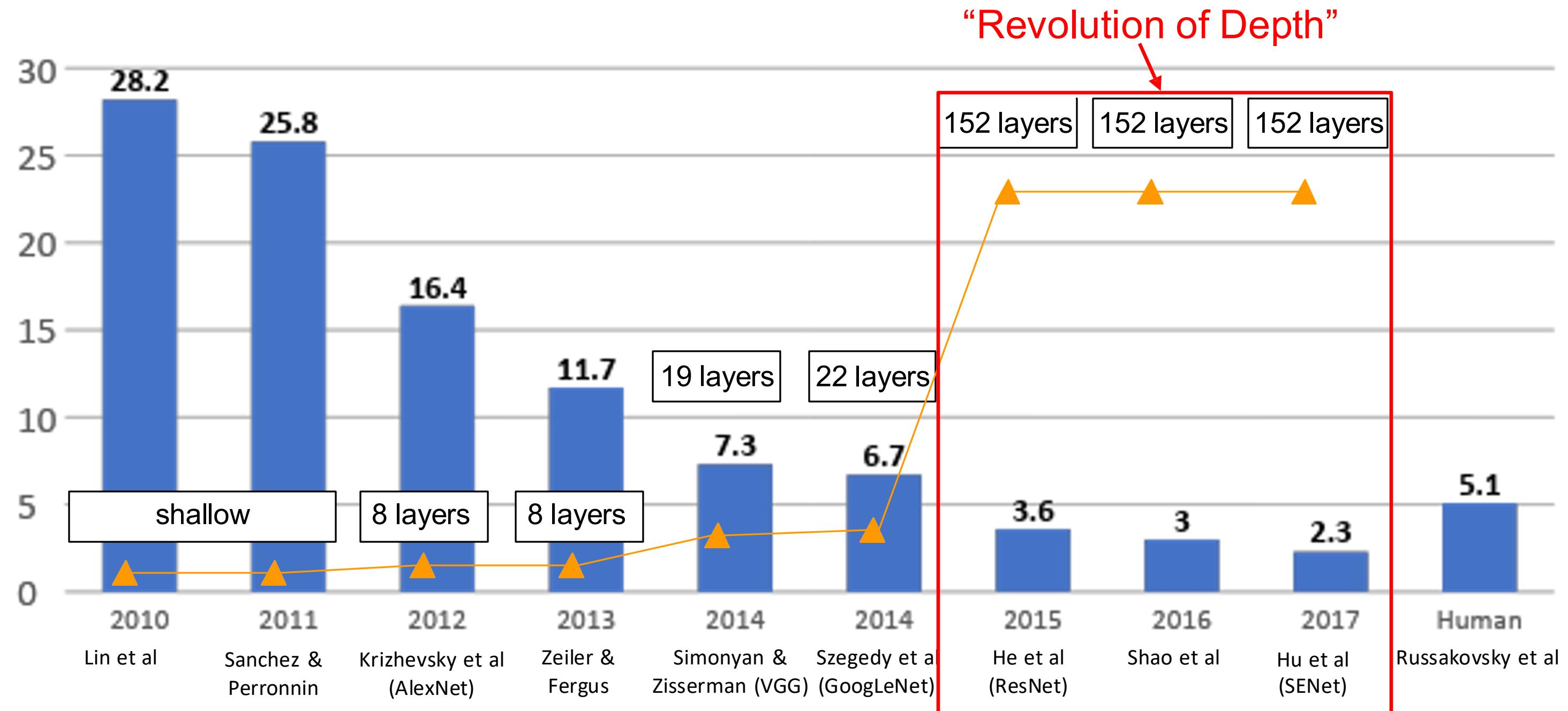
Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer

But deeper, more non-linearities

And fewer parameters: $3 * (3^2 C^2)$ vs. $7^2 C^2$ for C channels per layer



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Case Study: ResNet

[He et al., 2015]

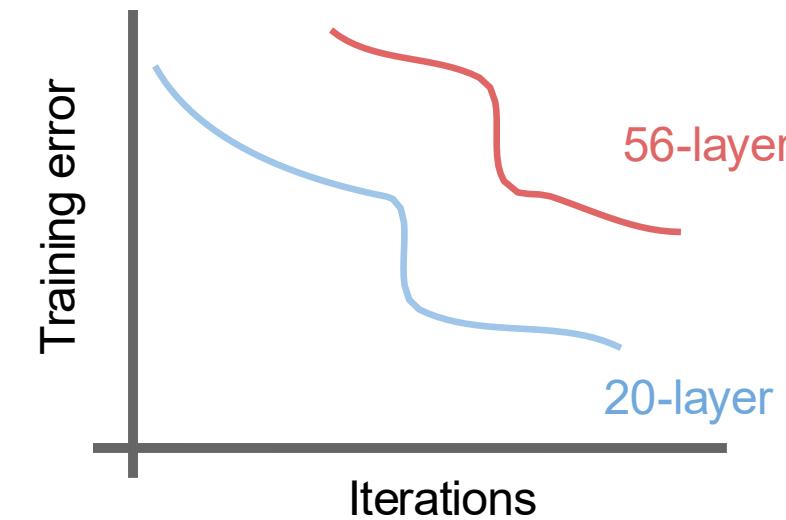
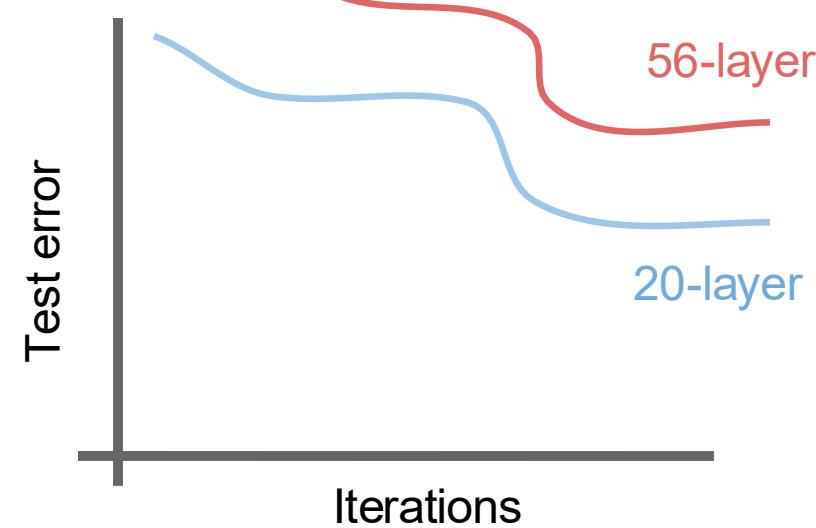
What happens when we continue stacking deeper layers on a “plain” convolutional neural network?



Case Study: ResNet

[He et al., 2015]

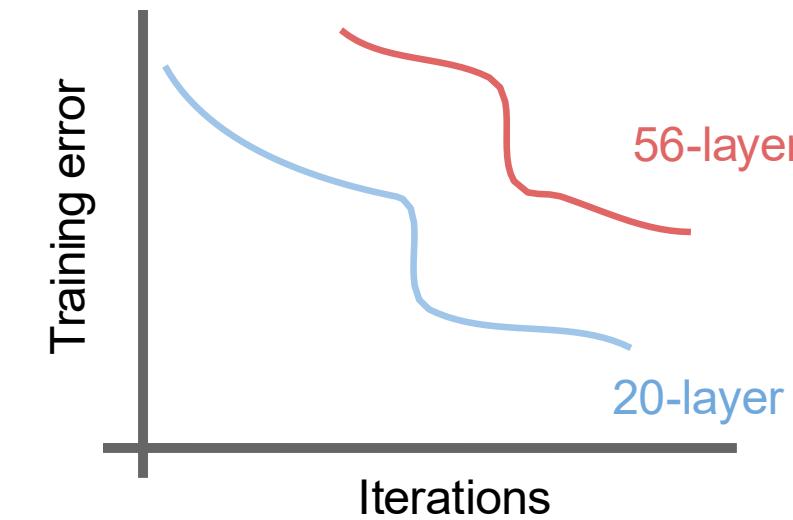
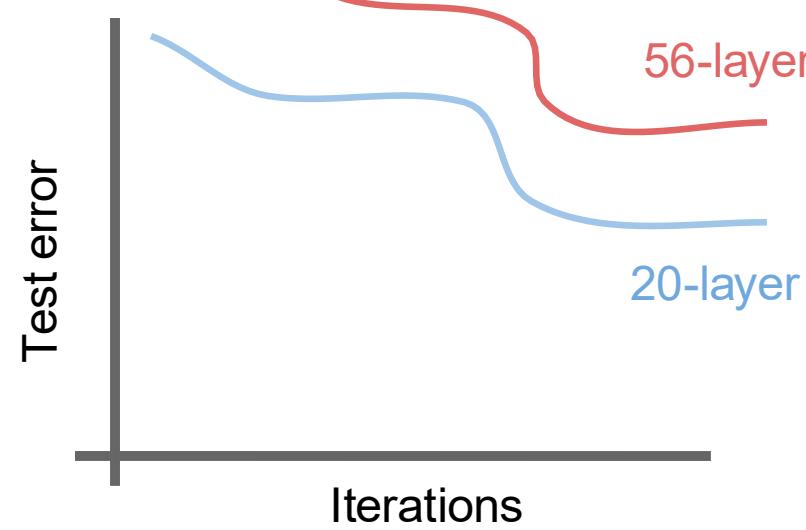
What happens when we continue stacking deeper layers on a “plain” convolutional neural network?



Case Study: ResNet

[He et al., 2015]

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?



56-layer model performs worse on both test and training error
-> The deeper model performs worse, but it's not caused by overfitting!

Case Study: ResNet

[He et al., 2015]

Fact: Deep models have more representation power
(more parameters) than shallower models.

Hypothesis: the problem is an *optimization* problem,
deeper models are harder to optimize



Case Study: ResNet

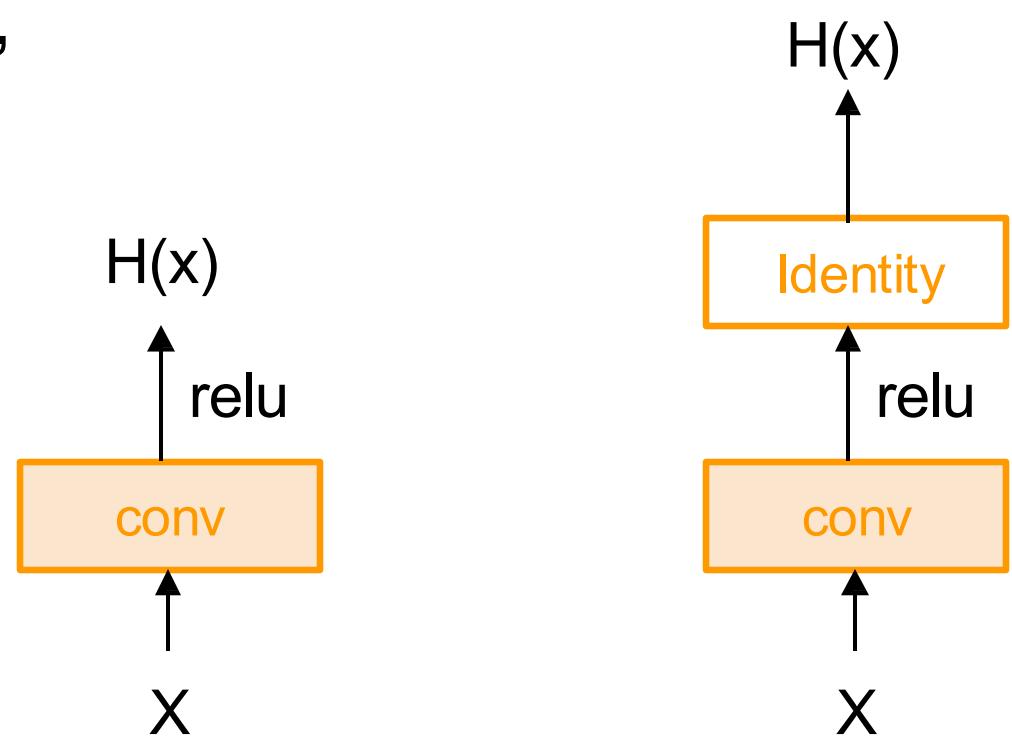
[He et al., 2015]

Fact: Deep models have more representation power (more parameters) than shallower models.

Hypothesis: the problem is an *optimization* problem, deeper models are harder to optimize

What should the deeper model learn to be at least as good as the shallower model?

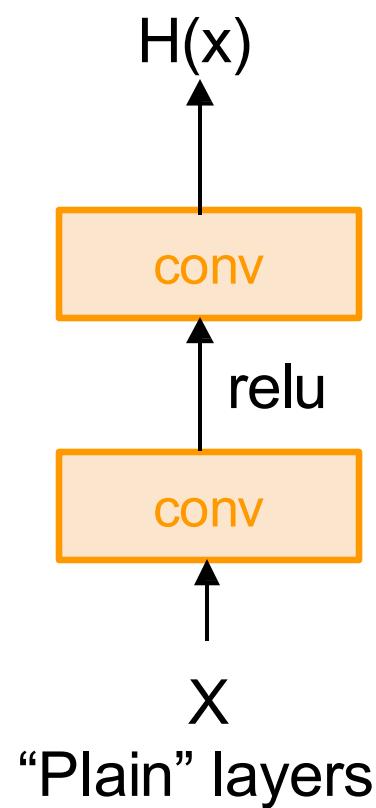
A solution by construction is copying the learned layers from the shallower model and setting additional layers to identity mapping.



Case Study: ResNet

[He et al., 2015]

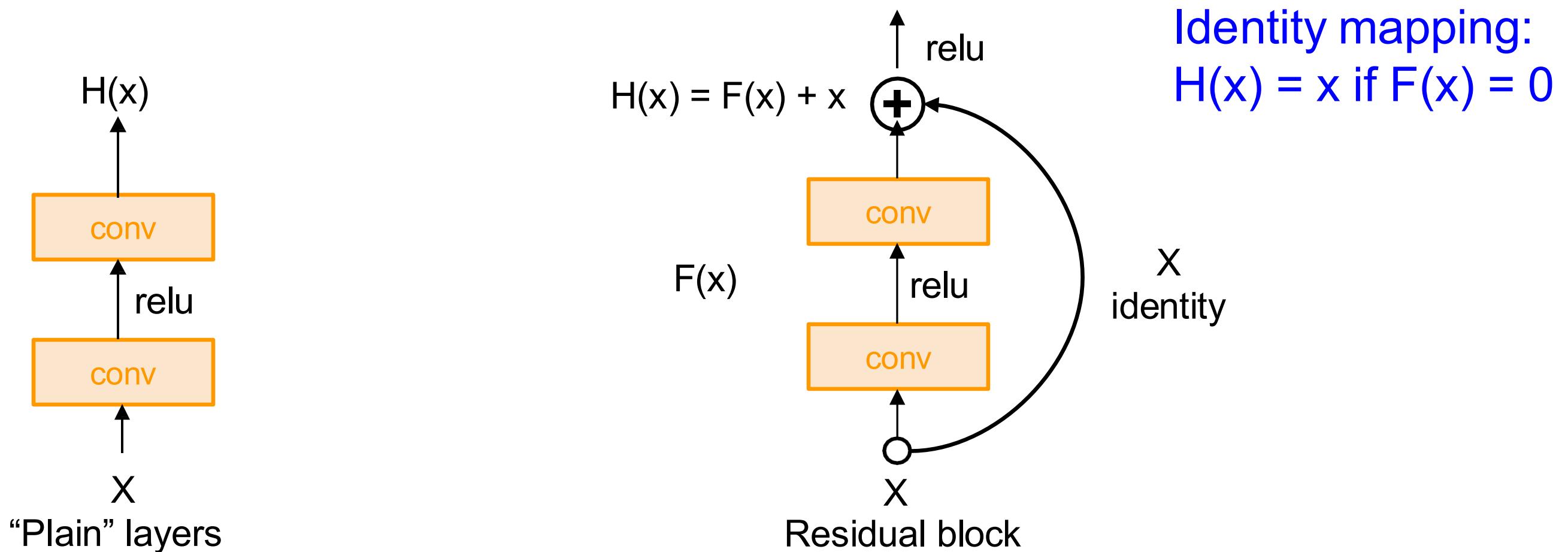
Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping



Case Study: ResNet

[He et al., 2015]

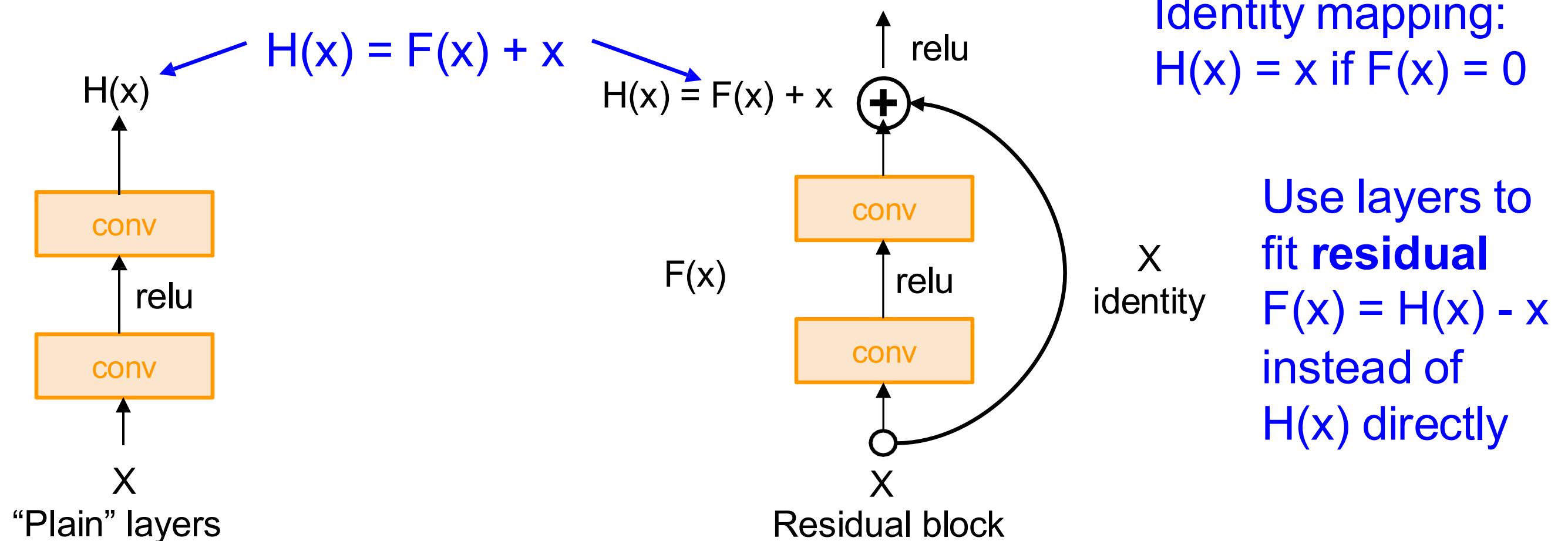
Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping



Case Study: ResNet

[He et al., 2015]

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping

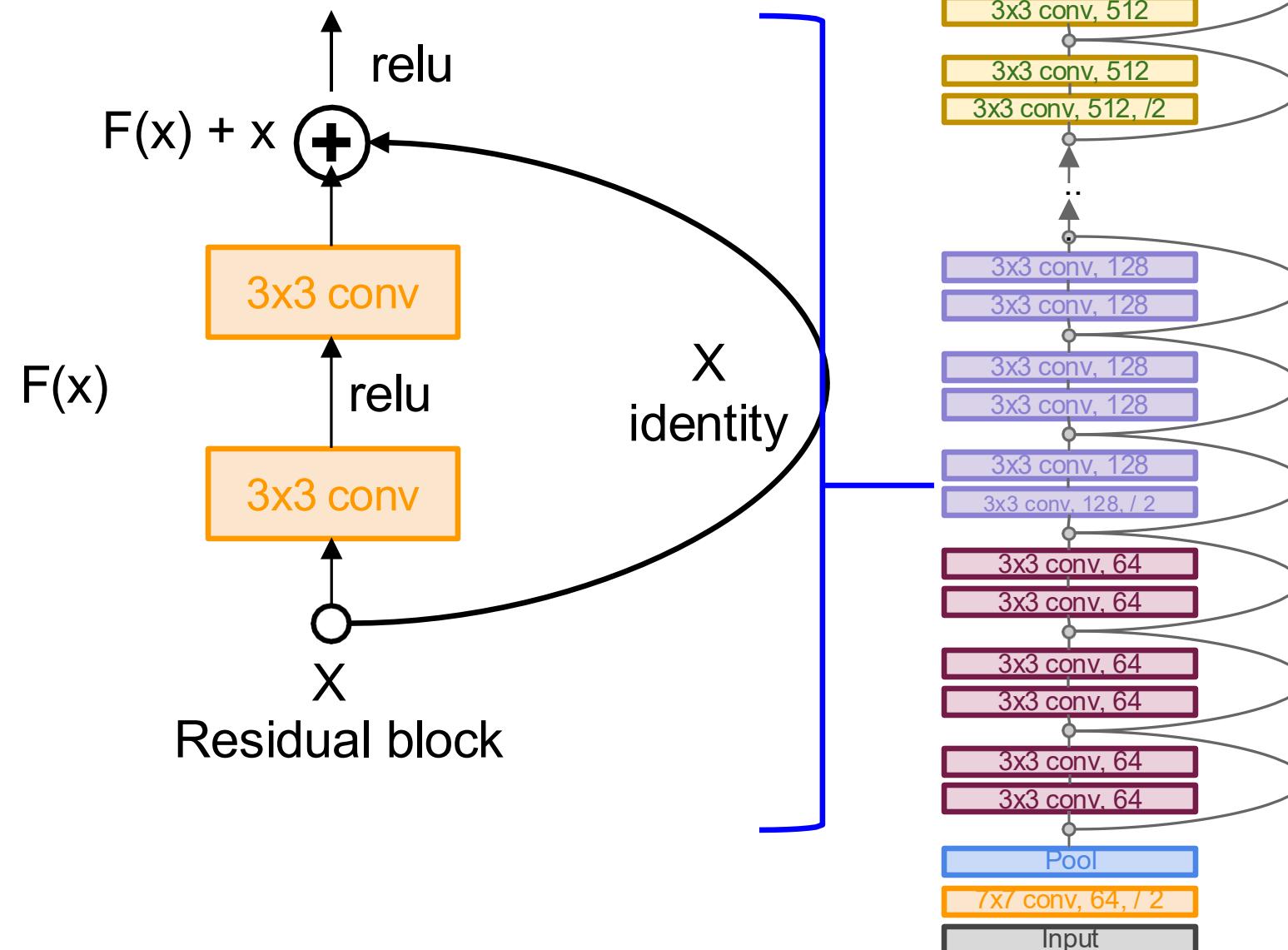


Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

- Stack residual blocks
- Every residual block has two 3x3 conv layers

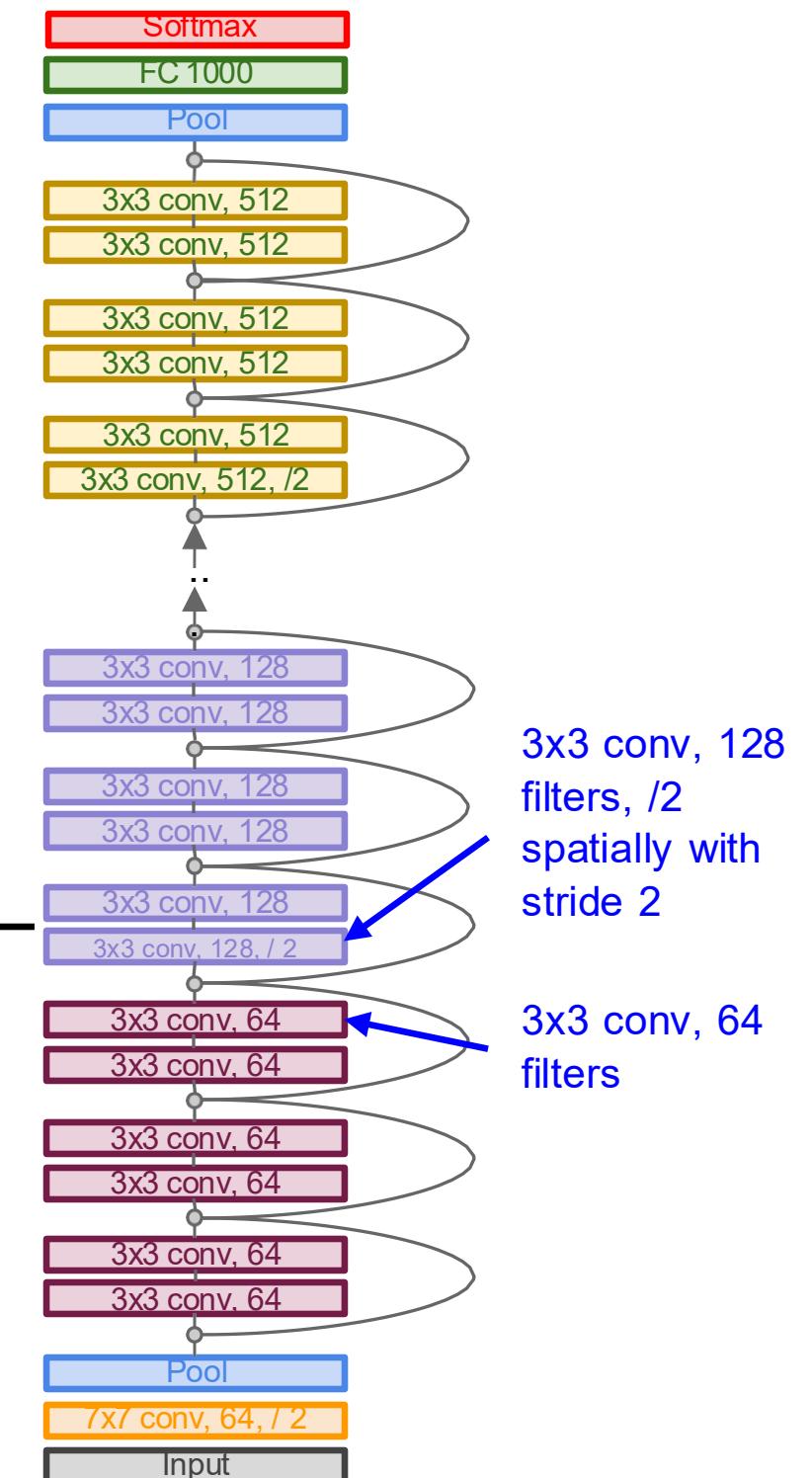
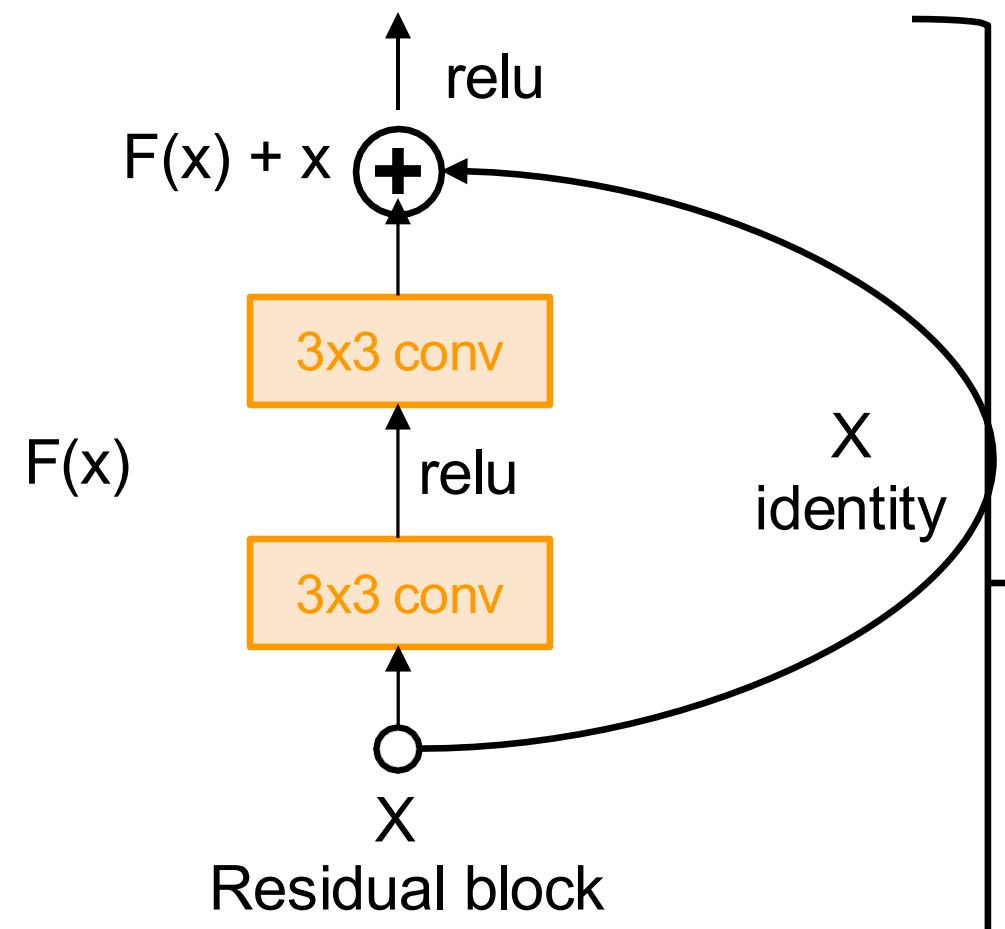


Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
Reduce the activation volume by half.

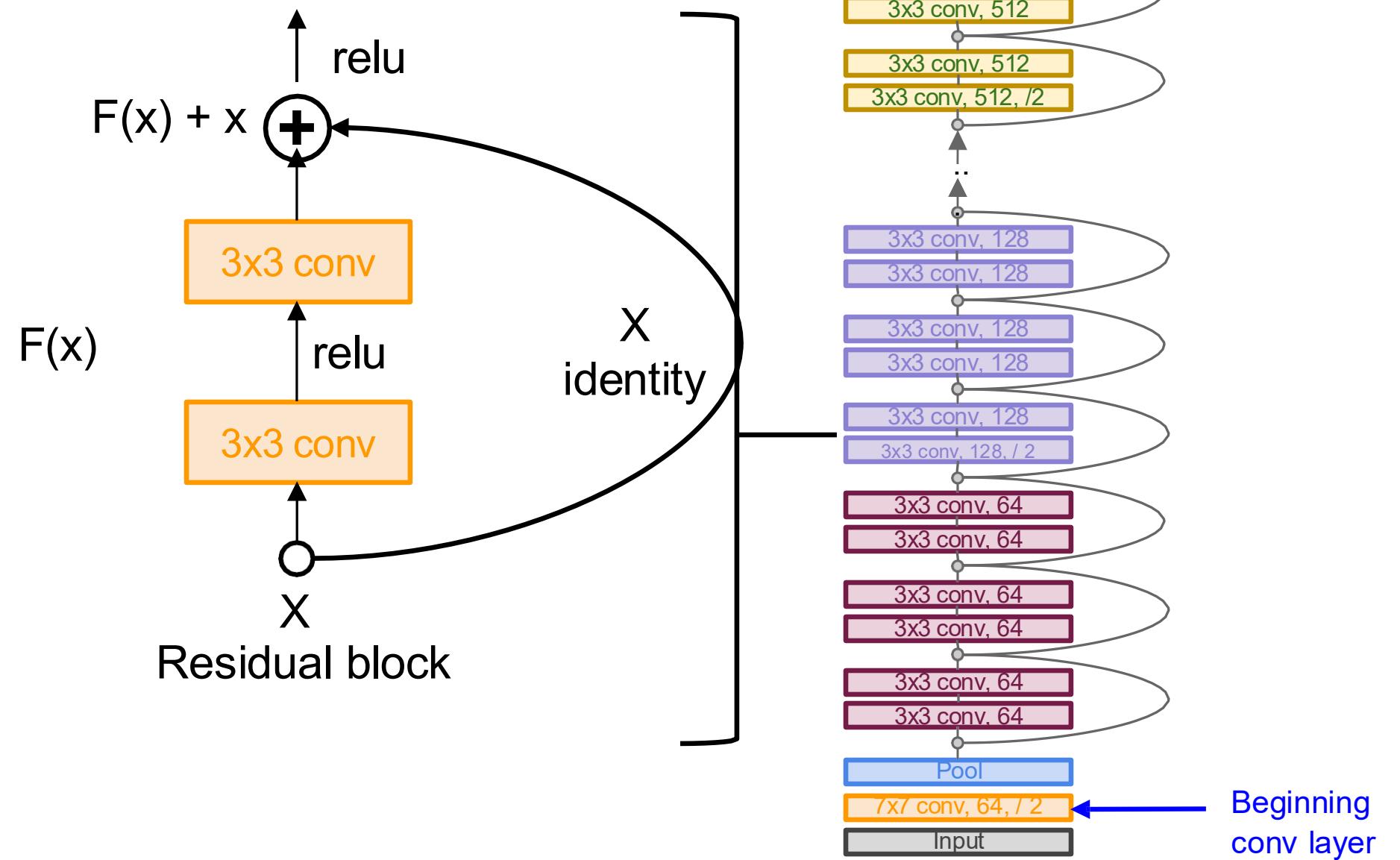


Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

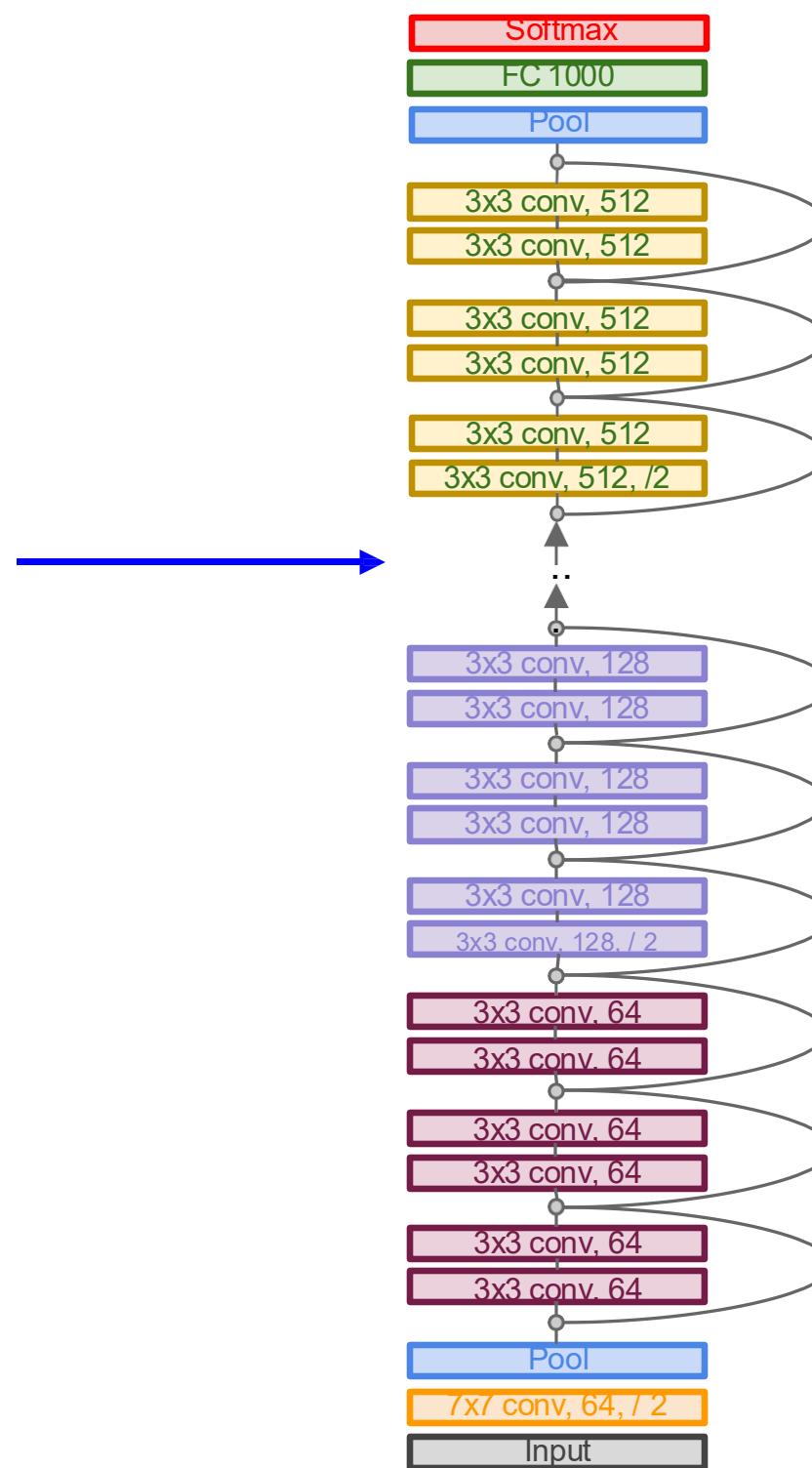
- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
- Additional conv layer at the beginning (stem)



Case Study: ResNet

[He et al., 2015]

Total depths of 18, 34, 50,
101, or 152 layers for
ImageNet

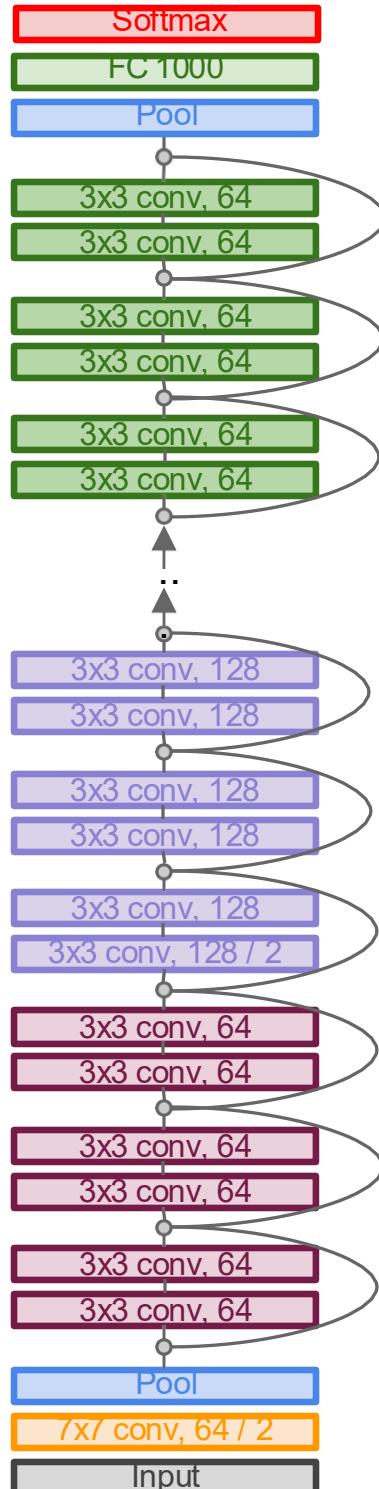
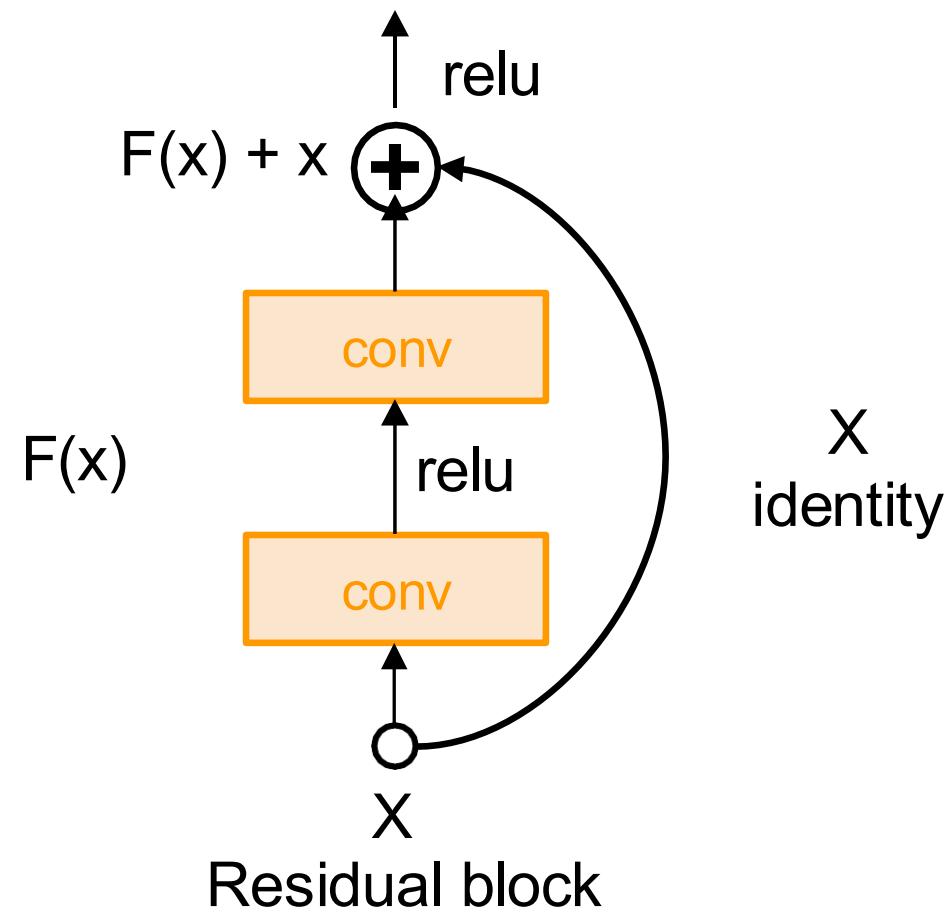


Case Study: ResNet

[He et al., 2015]

Very deep networks using residual connections

- 152-layer model for ImageNet
- ILSVRC'15 classification winner (3.57% top 5 error)
- Swept all classification and detection competitions in ILSVRC'15 and COCO'15!



Lecture Overview – Two Broad Sets of Topics

How to build CNNs?

Layers in CNNs
Activation Functions
CNN Architectures
Weight Initialization

How to train CNNs?

Data Preprocessing
Data augmentation
Transfer Learning
Hyperparameter Selection

How to initialize weights in neural network layers?



Weight Initialization Case: Values too small

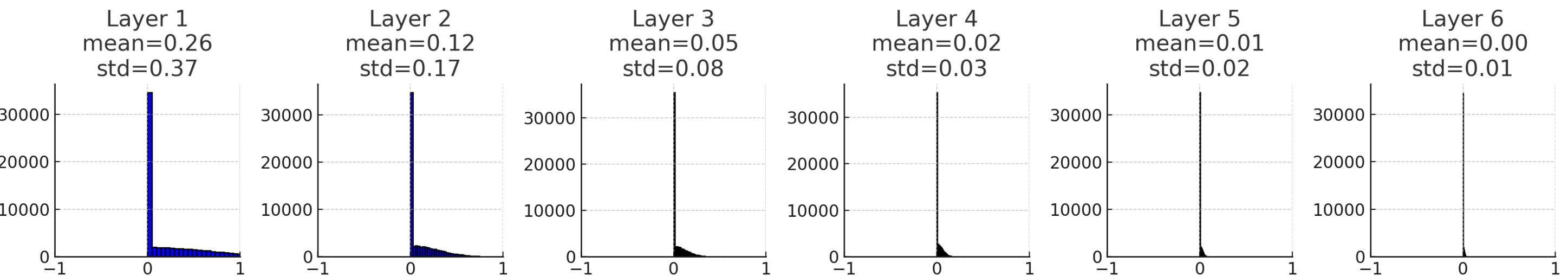
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
# Forward pass with ReLU activation
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout) # Small weight init
    x = np.maximum(0, x.dot(W)) # ReLU activation
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

Weight Initialization Case: Values too small

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
# Forward pass with ReLU activation
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout) # Small weight init
    x = np.maximum(0, x.dot(W)) # ReLU activation
    hs.append(x)
```

All activations tend to zero
for deeper network layers



Weight Initialization Case: Values too large

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
# Forward pass with ReLU activation
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout) # Small weight init
    x = np.maximum(0, x.dot(W)) # ReLU activation
    hs.append(x)
```

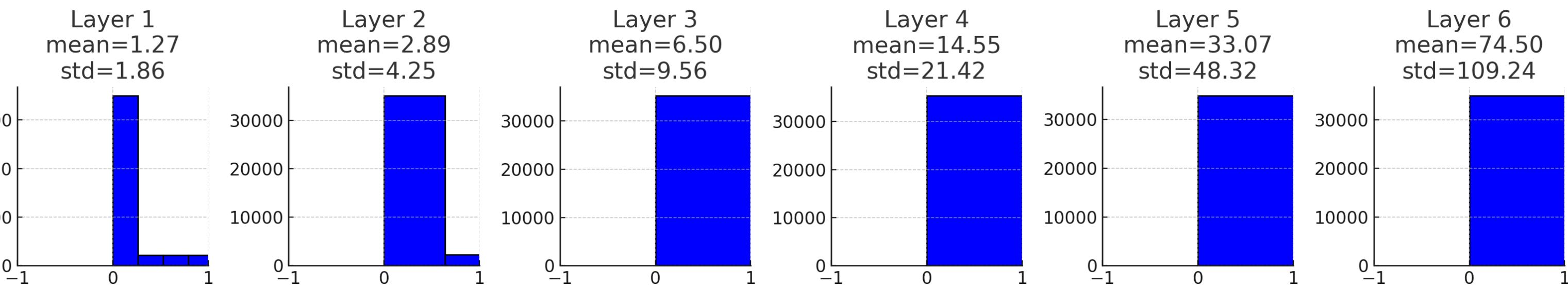
Increase std of initial weights from 0.01 to 0.05

Weight Initialization Case: Values too large

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
# Forward pass with ReLU activation
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout) # Small weight init
    x = np.maximum(0, x.dot(W)) # ReLU activation
    hs.append(x)
```

Activations blow up quickly

Increase std of initial weights from 0.01 to 0.05



How to fix this? Depends on the size of the layer

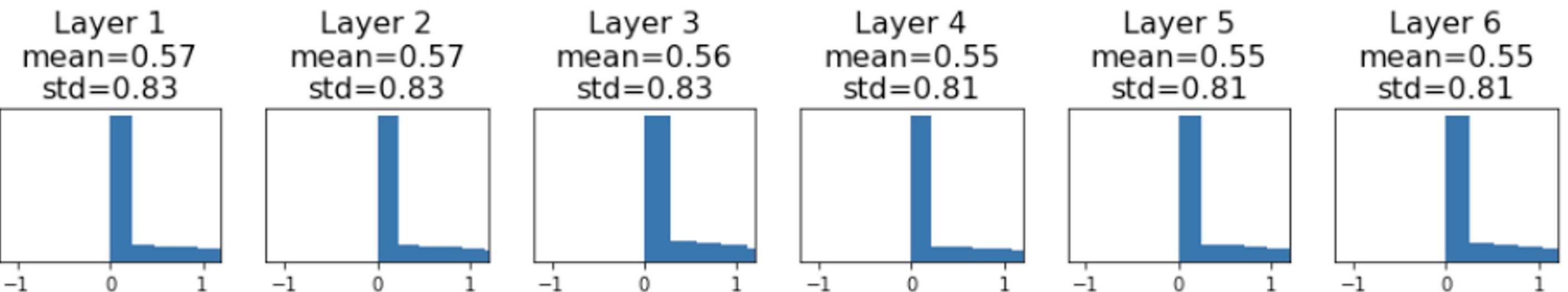
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) * np.sqrt(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

He et al, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, ICCV 2015

One solution: Kaiming / MSRA Initialization

```
dims = [4096] * 7  #ReLU correction: std = sqrt(2 / Din)
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) * np.sqrt(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!



He et al, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, ICCV 2015

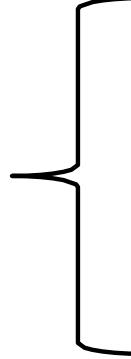
Lecture Overview – Two Broad Sets of Topics

How to build CNNs?



- Layers in CNNs
- Activation Functions
- CNN Architectures
- Weight Initialization

How to train CNNs?



- Data Preprocessing**
- Data augmentation
- Transfer Learning
- Hyperparameter Selection

TLDR for Image Normalization: center and scale for each channel

- Subtract per-channel mean and Divide by per-channel std (almost all modern models)
(stats along each channel = 3 numbers)
- Requires pre-computing means and std for each pixel channel (given your dataset)

```
norm_pixel[i,j,c] = (pixel[i,j,c] - np.mean(pixel[:, :, c])) / np.std(pixel[:, :, c])
```

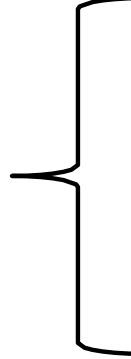
Lecture Overview – Two Broad Sets of Topics

How to build CNNs?



- Layers in CNNs
- Activation Functions
- CNN Architectures
- Weight Initialization

How to train CNNs?



- Data Preprocessing
- Data augmentation**
- Transfer Learning
- Hyperparameter Selection

Regularization: A common pattern

Training: Add some kind of randomness

$$y = f_W(x, z)$$

Testing: Average out randomness (sometimes approximate)

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Regularization: A common pattern

Training: Add some kind of randomness

$$y = f_W(x, z)$$

Testing: Average out randomness (sometimes approximate)

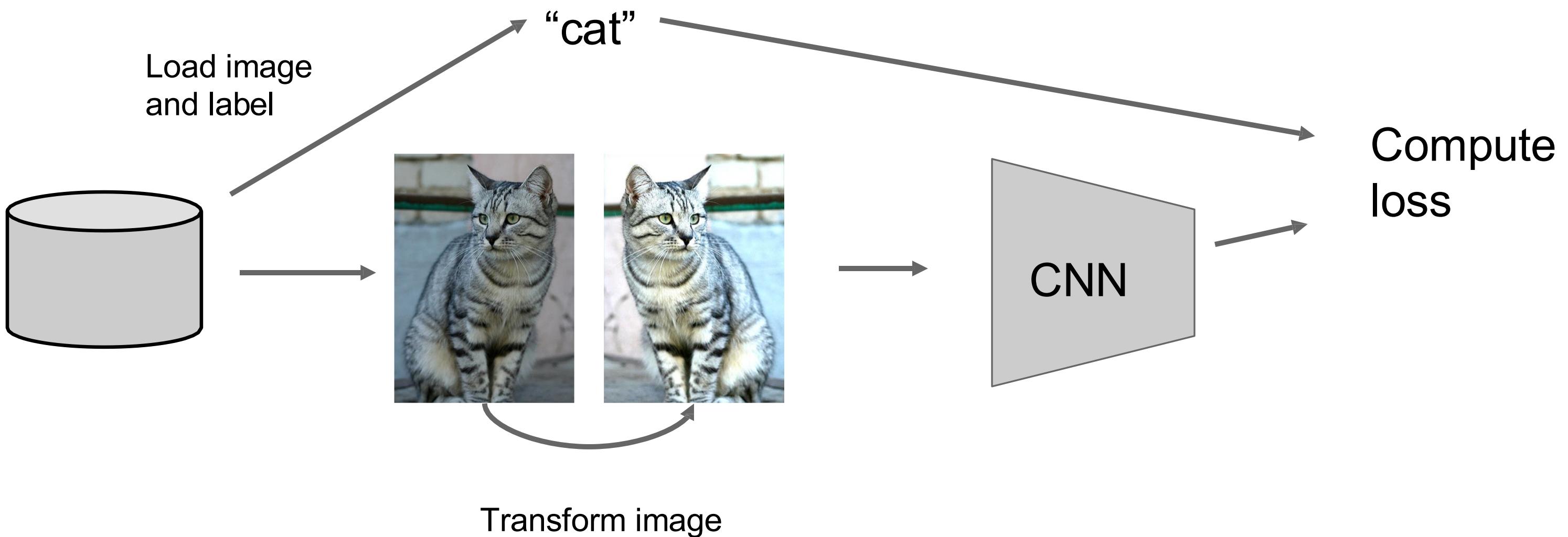
$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Example: Dropout

Training:
Randomly drop activations

Testing: Use all activations and average values with p

Regularization: Data Augmentation



Data Augmentation

Horizontal Flips



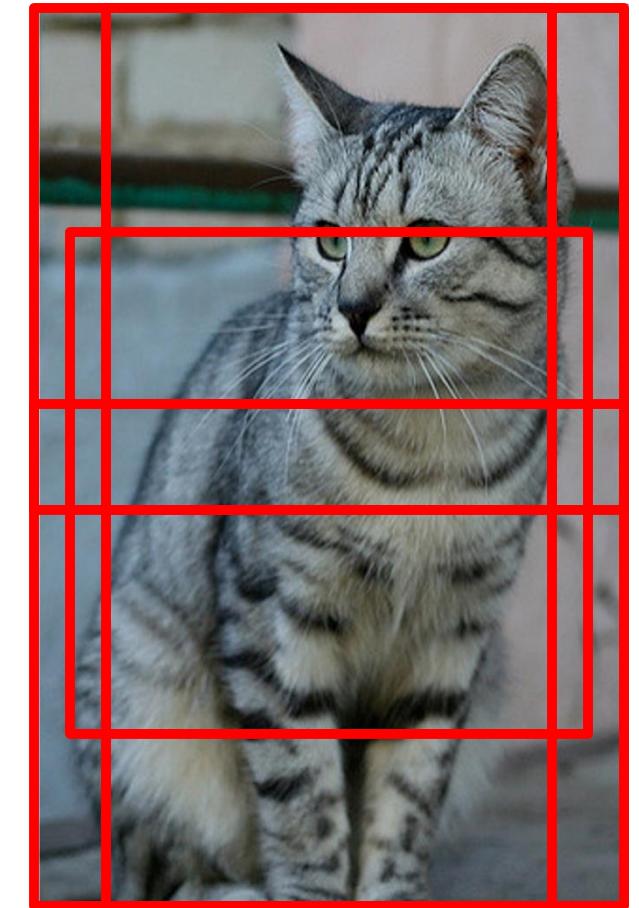
Data Augmentation

Random crops and scales

Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224×224 patch



Test Time Augmentation: average a fixed set of crops

ResNet:

1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224×224 crops: 4 corners + center, + flips

Data Augmentation

Color Jitter

Simple: Randomize
contrast and brightness



Regularization: Cutout

Training: Set random image regions to zero

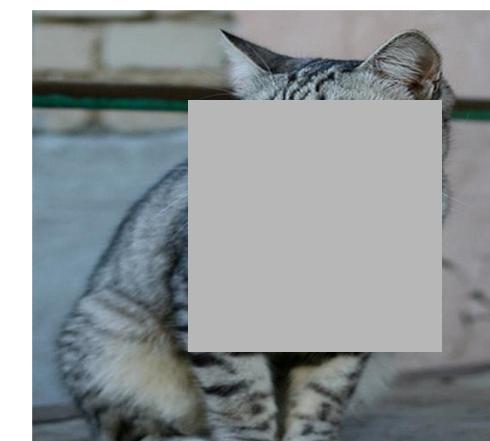
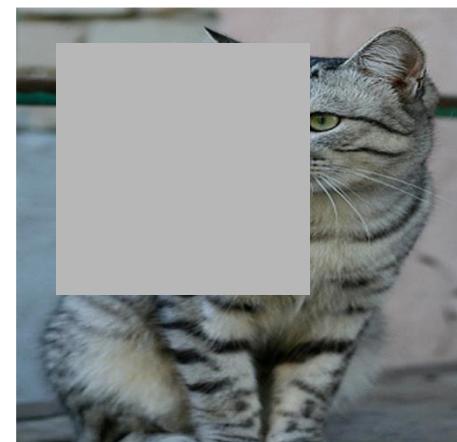
Testing: Use full image

Examples:

Dropout

Data Augmentation

Cutout / Random Crop



Works very well for small datasets like CIFAR,
less common for large datasets like ImageNet

DeVries and Taylor, "Improved Regularization of
Convolutional Neural Networks with Cutout", arXiv 2017

Lecture Overview – Two Broad Sets of Topics

How to build CNNs?

Layers in CNNs
Activation Functions
CNN Architectures
Weight Initialization

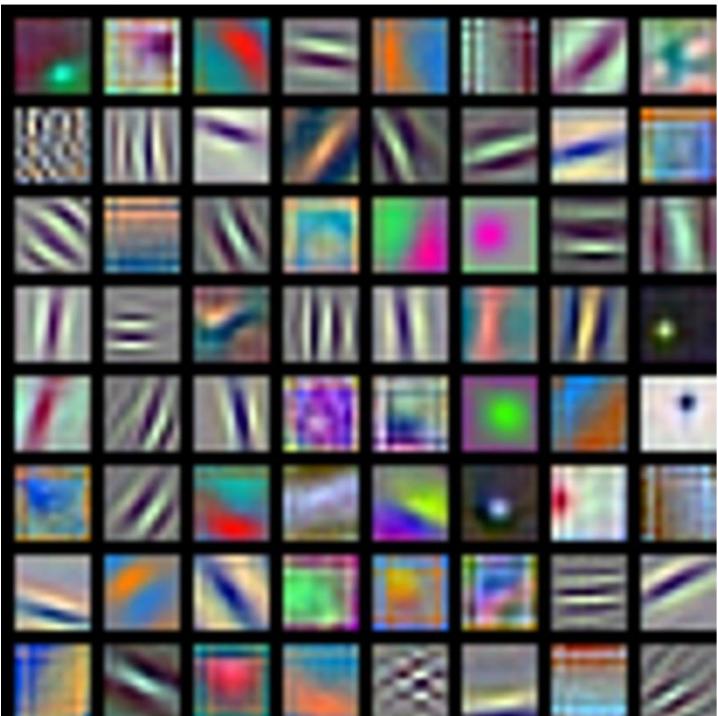
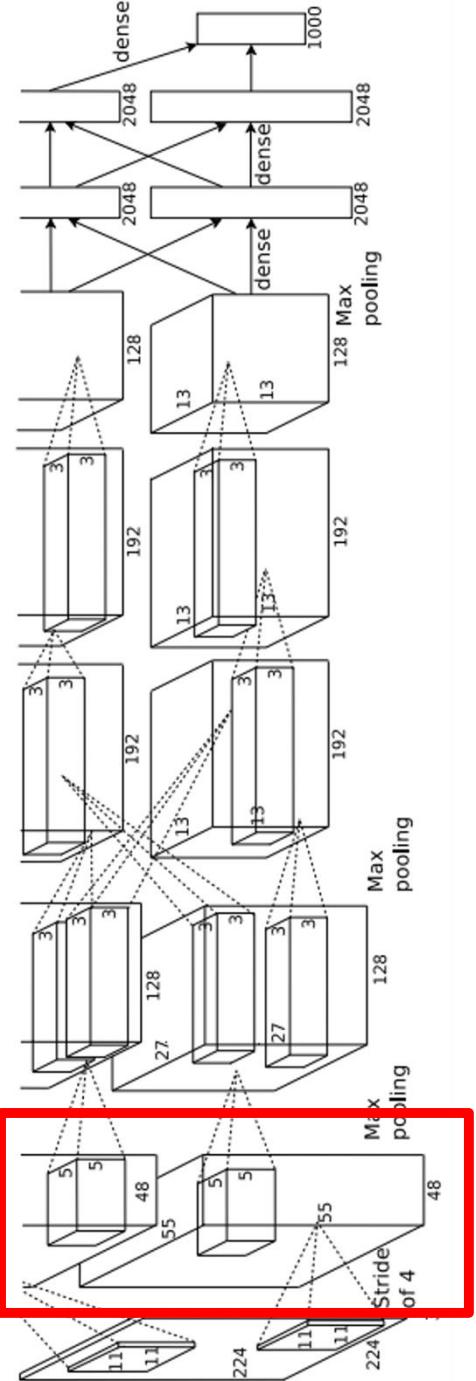
How to train CNNs?

Data Preprocessing
Data augmentation
Transfer Learning
Hyperparameter Selection

What if you **don't** have a lot of data? Can you still train CNNs?



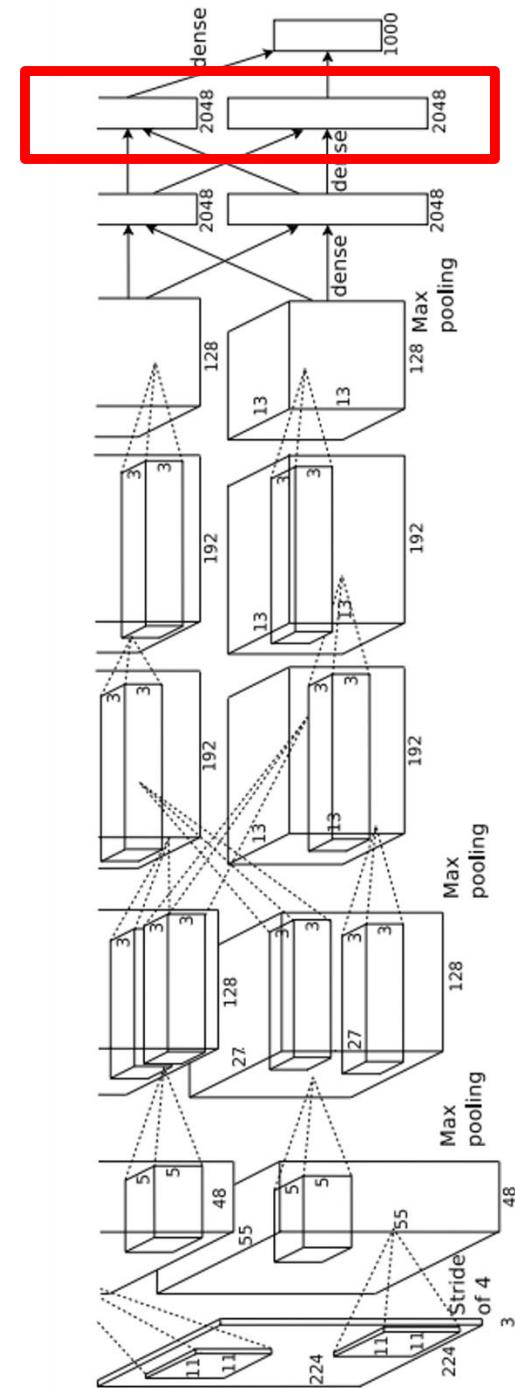
Transfer Learning with CNNs



AlexNet:
64 x 3 x 11 x 11

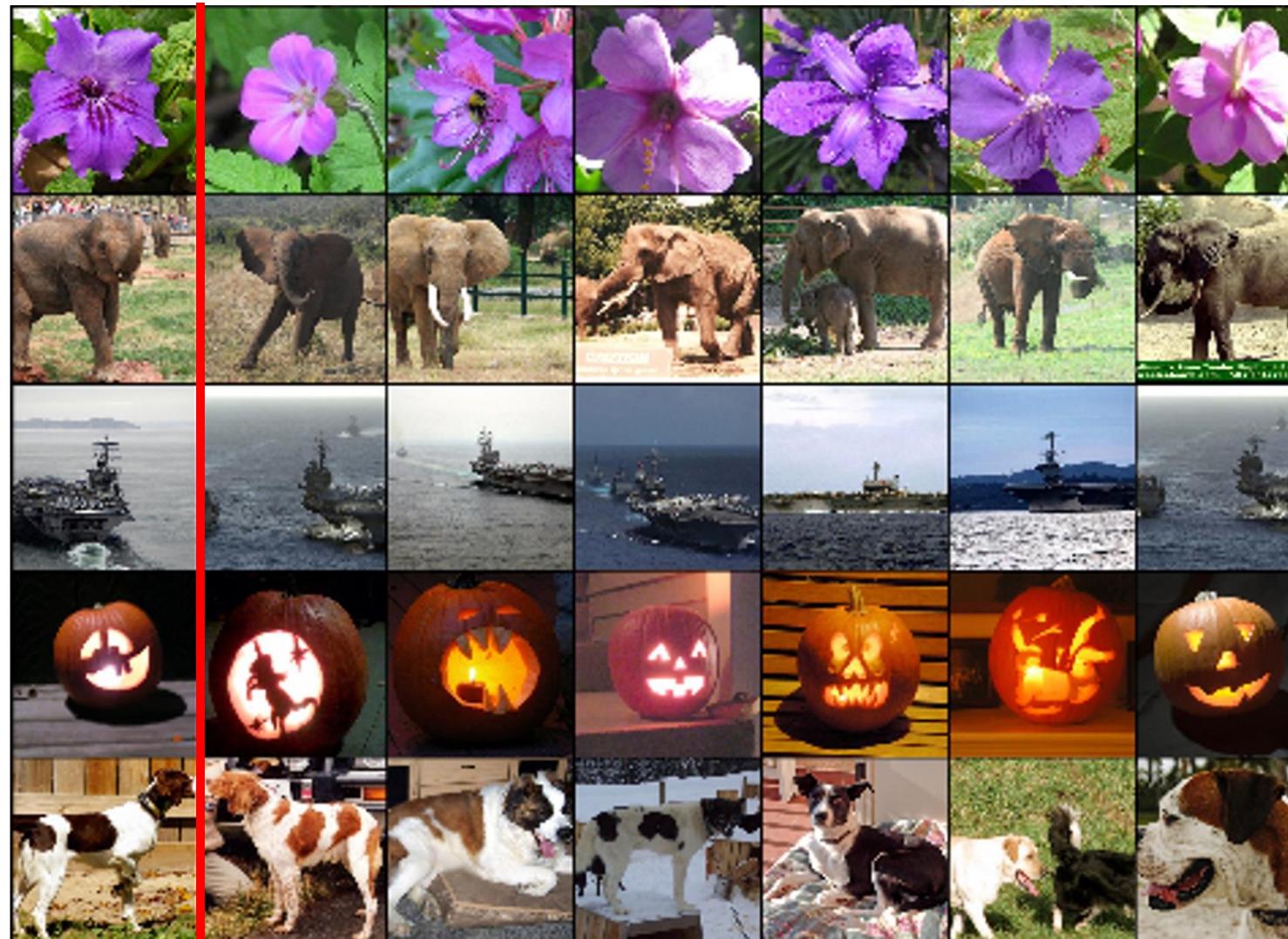
(More on this in Lecture 13)

Transfer Learning with CNNs



Test image

L2 Nearest neighbors in feature space

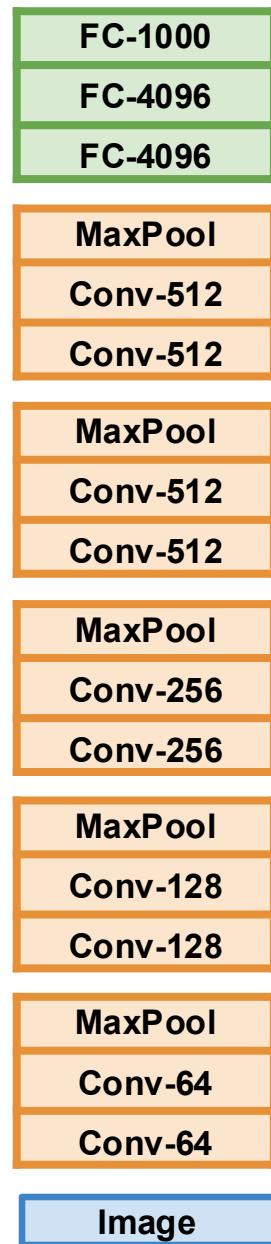


(More on this in Lecture 13)

Transfer Learning with CNNs

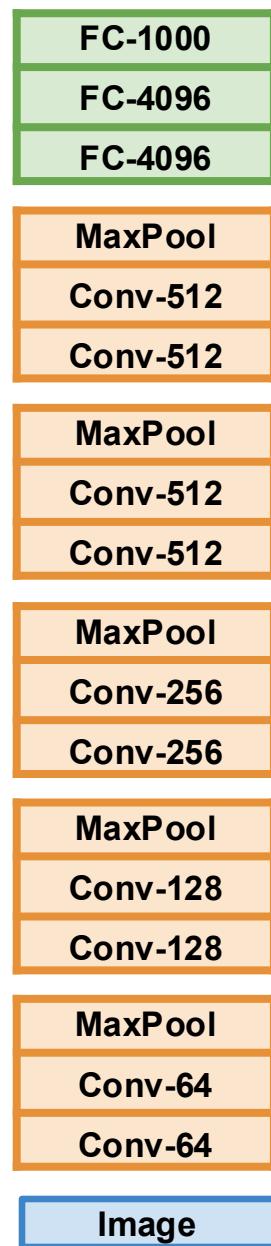
Donahue et al, “DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition”, ICML 2014
Razavian et al, “CNN Features Off-the-Shelf: An Astounding Baseline for Recognition”, CVPR Workshops 2014

1. Train on Imagenet (or internet scale data)

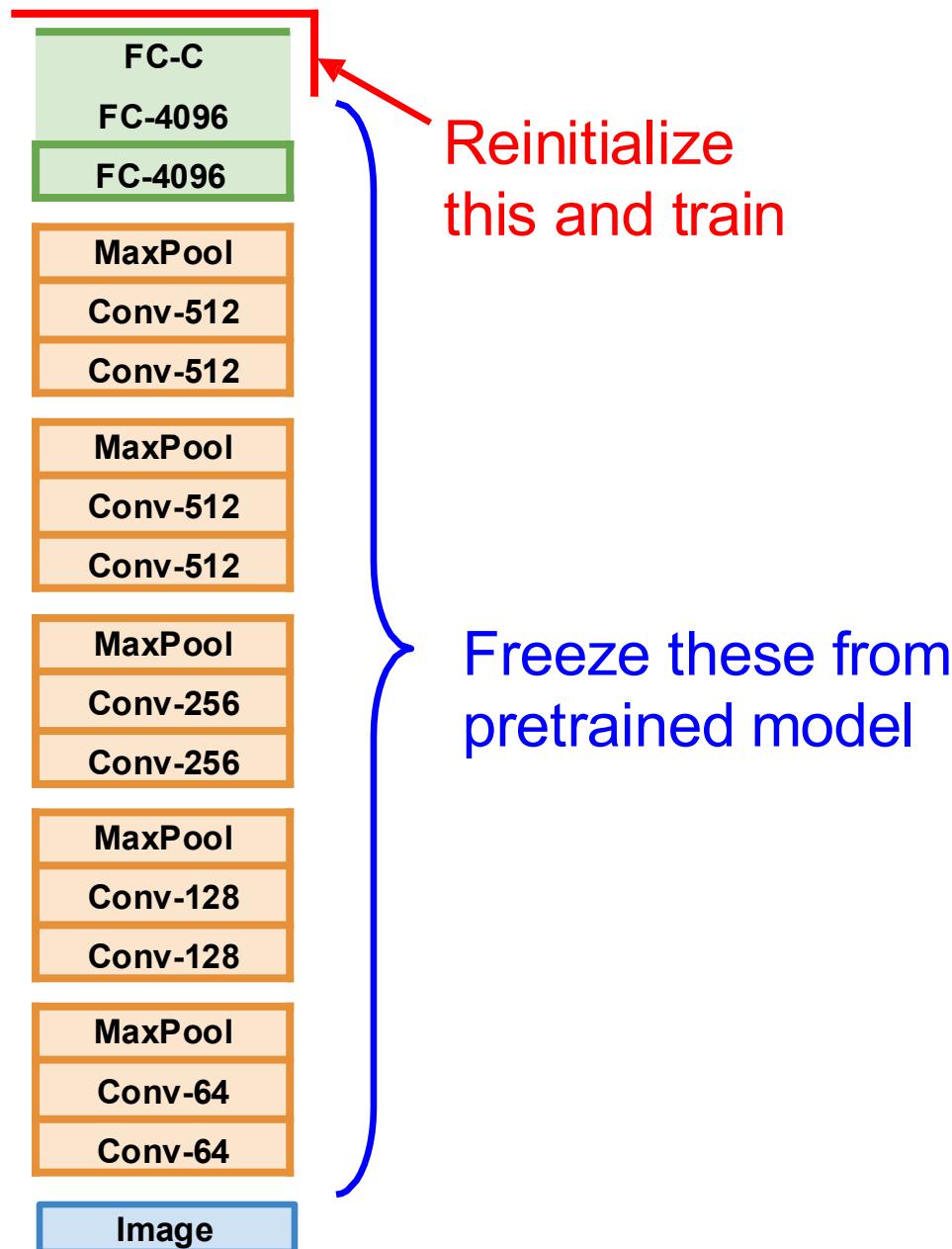


Transfer Learning with CNNs

1. Train on Imagenet



2. Small Dataset (C classes)

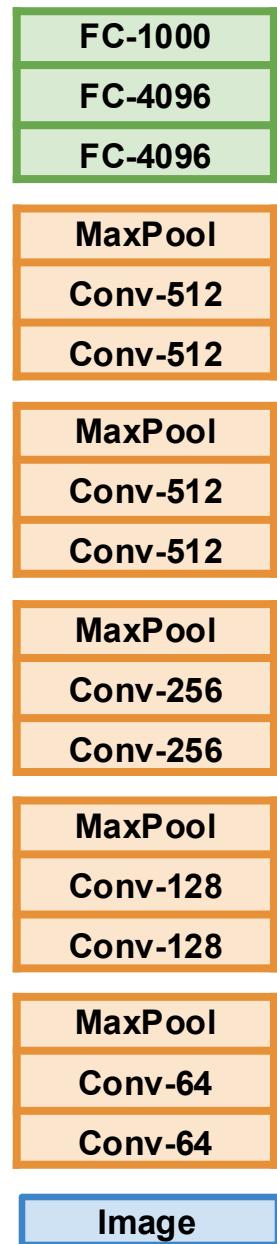


Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

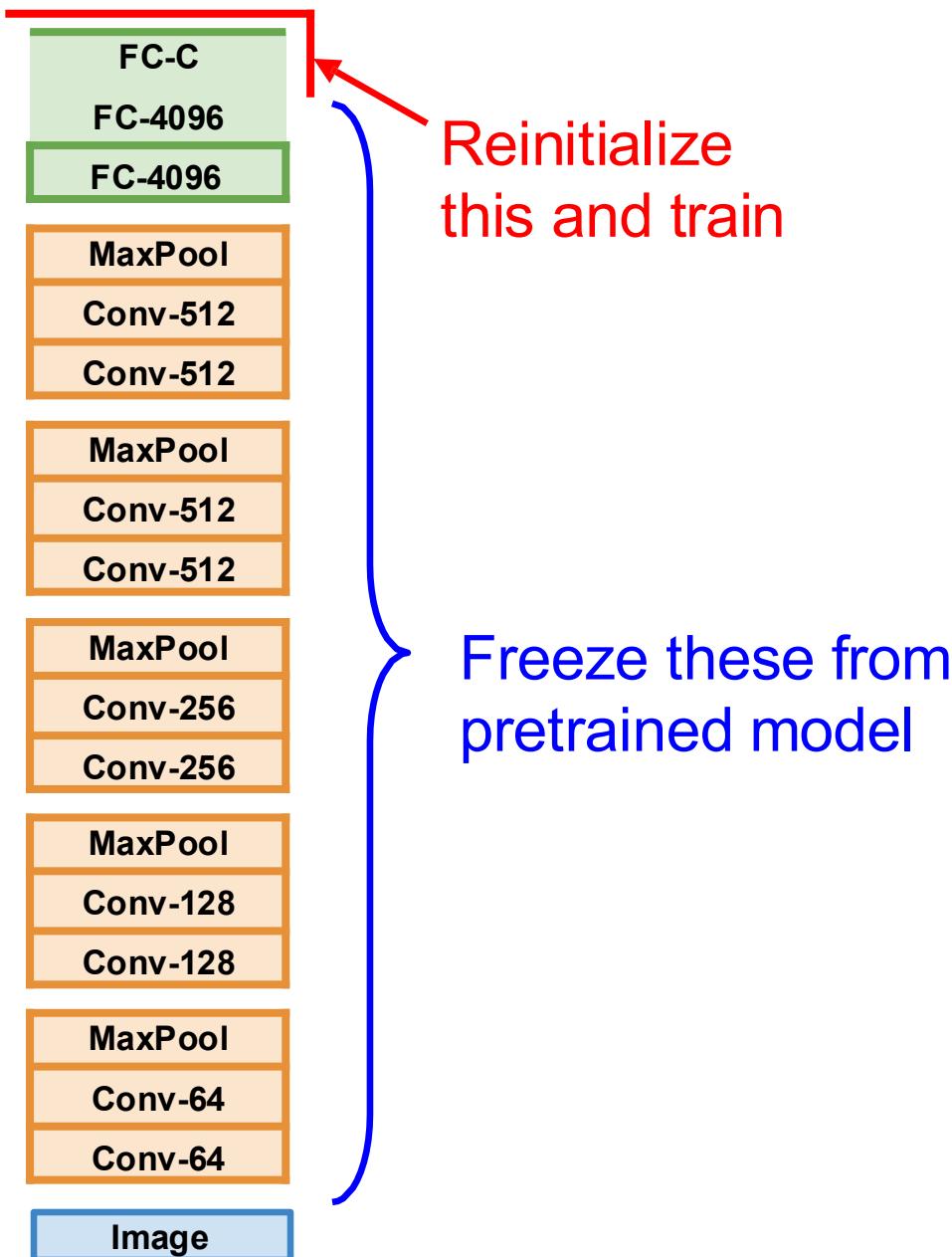
Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

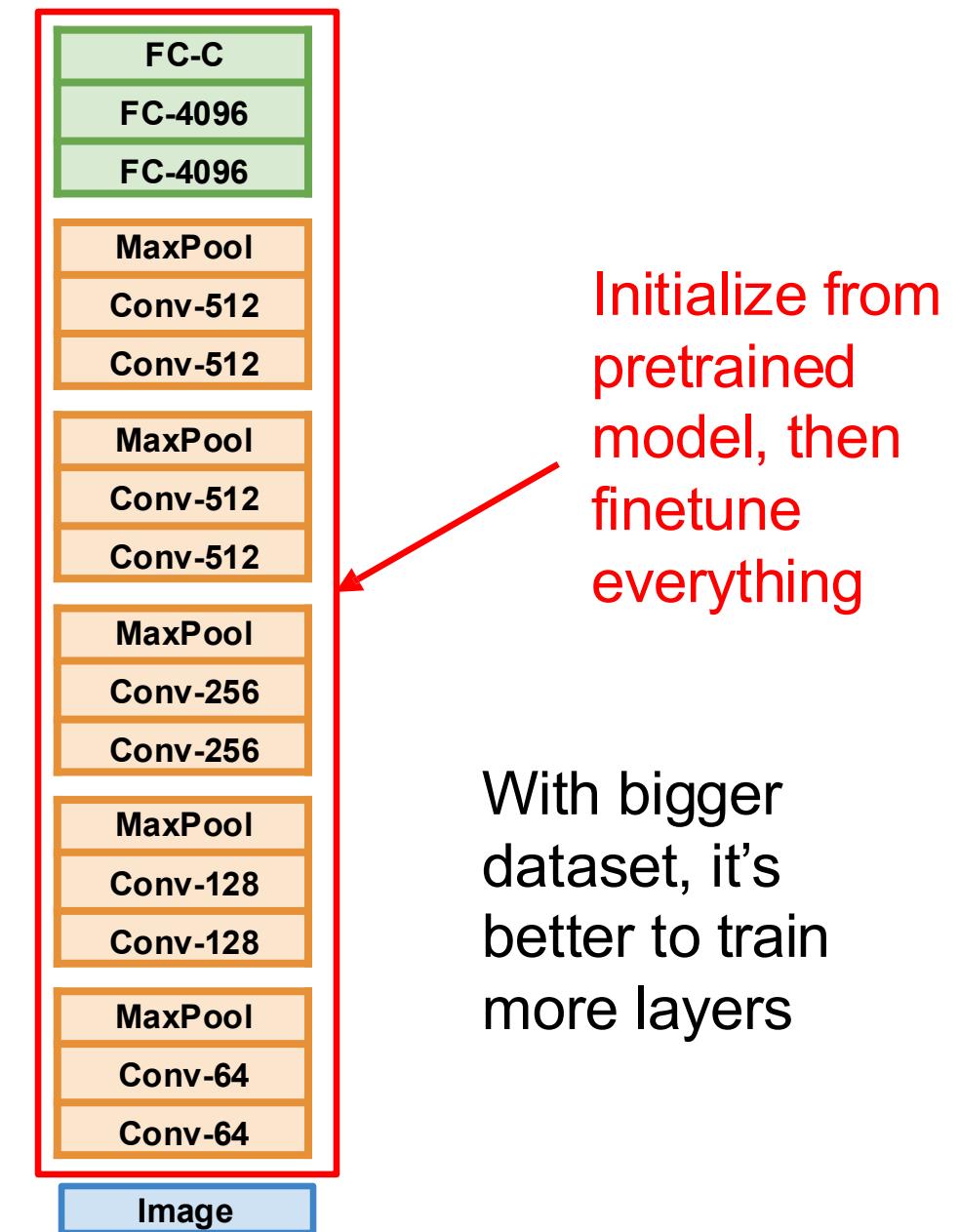
1. Train on Imagenet



2. Small Dataset (C classes)



3. Bigger dataset



FC-1000
FC-4096
FC-4096

MaxPool
Conv-512
Conv-512

MaxPool
Conv-512
Conv-512

MaxPool
Conv-256
Conv-256

MaxPool
Conv-128
Conv-128

MaxPool
Conv-64
Conv-64

Image

More specific

More generic

	very similar dataset	very different dataset
very little data	?	?
quite a lot of data	?	?

FC-1000
FC-4096
FC-4096

MaxPool
Conv-512
Conv-512

MaxPool
Conv-512
Conv-512

MaxPool
Conv-256
Conv-256

MaxPool
Conv-128
Conv-128

MaxPool
Conv-64
Conv-64

Image

More specific

More generic

	very similar dataset	very different dataset
very little data	Use Linear Classifier on final layer	?
quite a lot of data	Finetune all model layers	?

FC-1000
FC-4096
FC-4096

MaxPool
Conv-512
Conv-512

MaxPool
Conv-512
Conv-512

MaxPool
Conv-256
Conv-256

MaxPool
Conv-128
Conv-128

MaxPool
Conv-64
Conv-64

Image

More specific

More generic

	very similar dataset	very different dataset
very little data	Use Linear Classifier on final layer	Try another model or collect more data 😞
quite a lot of data	Finetune all model layers	Either finetune all model layers or train from scratch!

Have some dataset of interest but it has < ~1M images?

1. Find a very large dataset that has similar data, train a big model there
2. Transfer learn to your dataset

Deep learning frameworks provide a “Model Zoo” of pretrained models so you don’t need to train your own

PyTorch: <https://github.com/pytorch/vision>

Huggingface: <https://github.com/huggingface/pytorch-image-models>

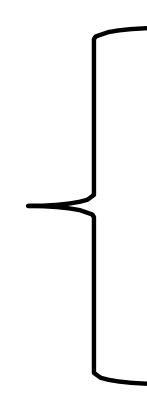
Lecture Overview – Two Broad Sets of Topics

How to build CNNs?



- Layers in CNNs
- Activation Functions
- CNN Architectures
- Weight Initialization

How to train CNNs?



- Data Preprocessing
- Data augmentation
- Transfer Learning
- Hyperparameter Selection

Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within ~100 iterations

Good learning rates to try: 1e-1, 1e-2, 1e-3, 1e-4, 1e-5

Choosing Hyperparameters

Step 1: Check initial loss

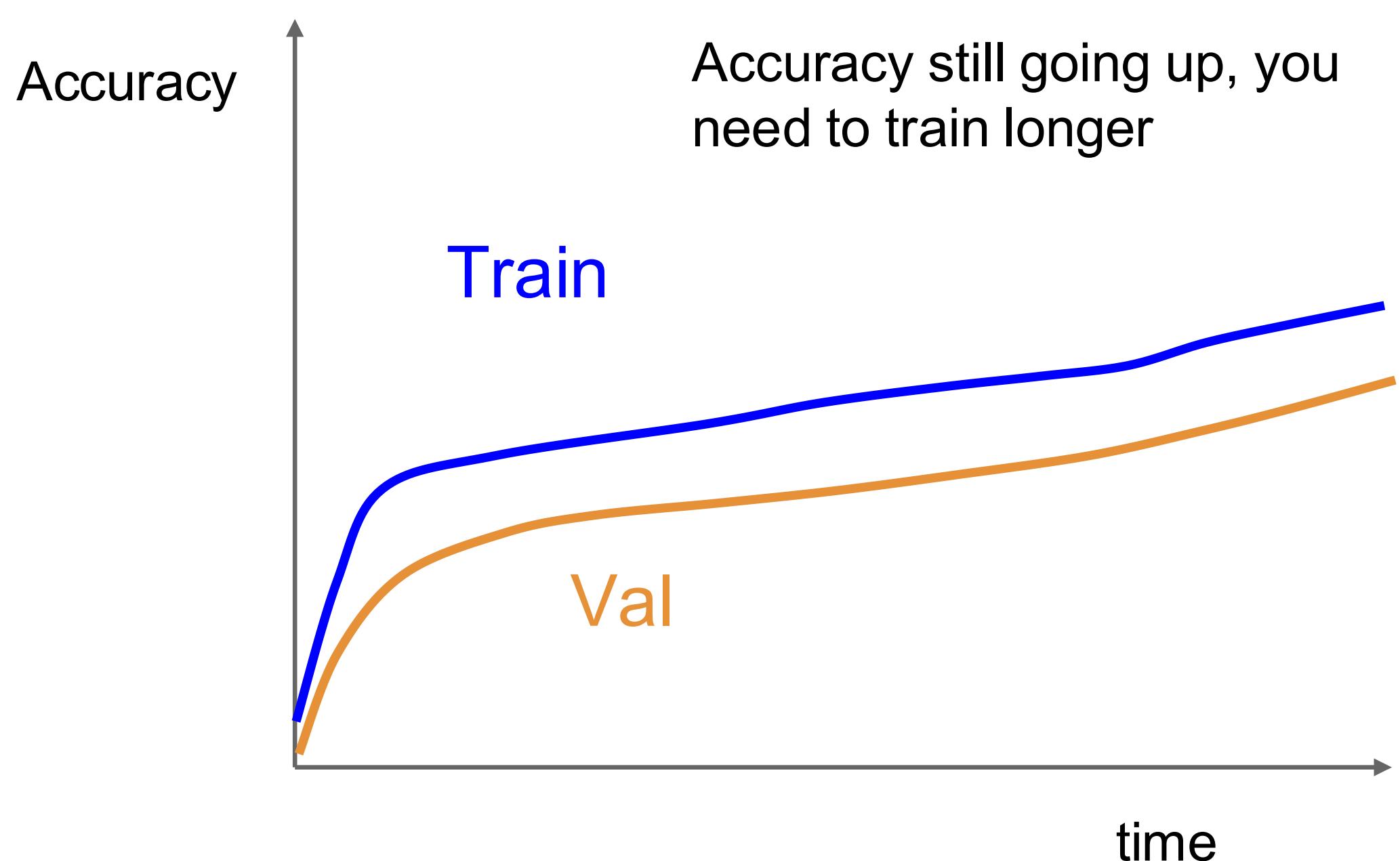
Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Step 4: Coarse grid of hyperparams, train for ~1-5 epochs

Step 5: Refine grid, train longer

Step 6: Look at loss and accuracy curves (next slides)



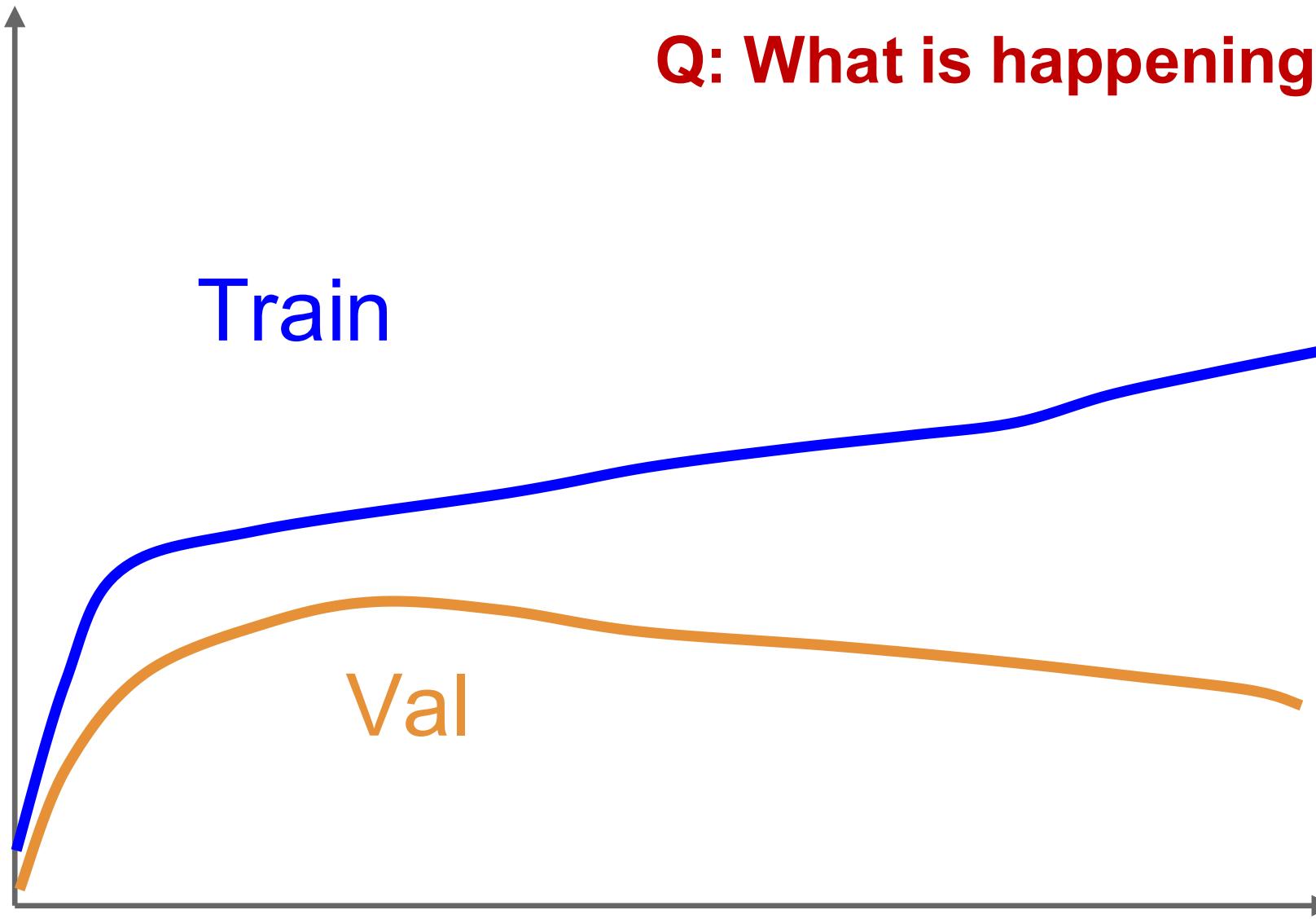
Accuracy

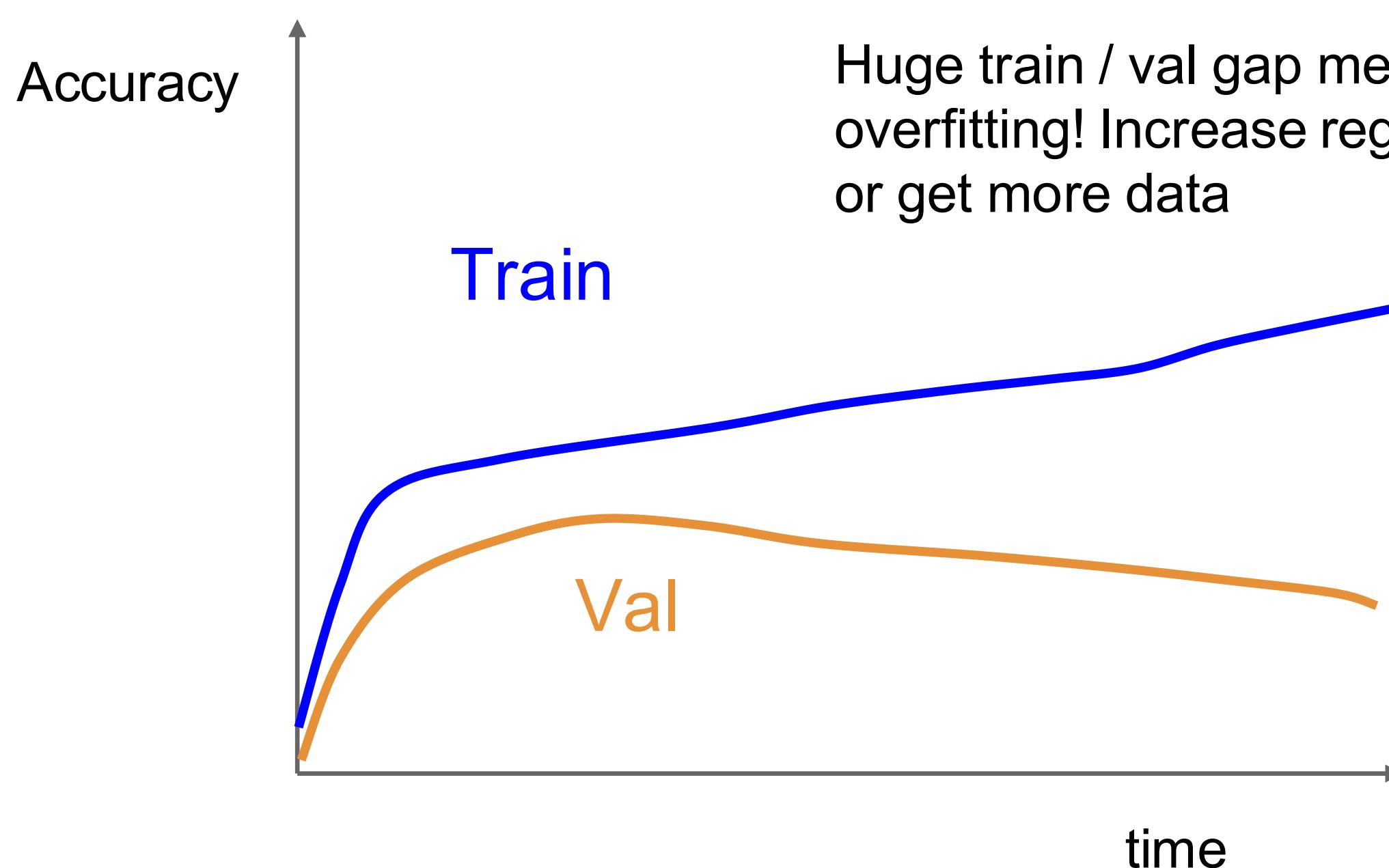
Q: What is happening here?

Train

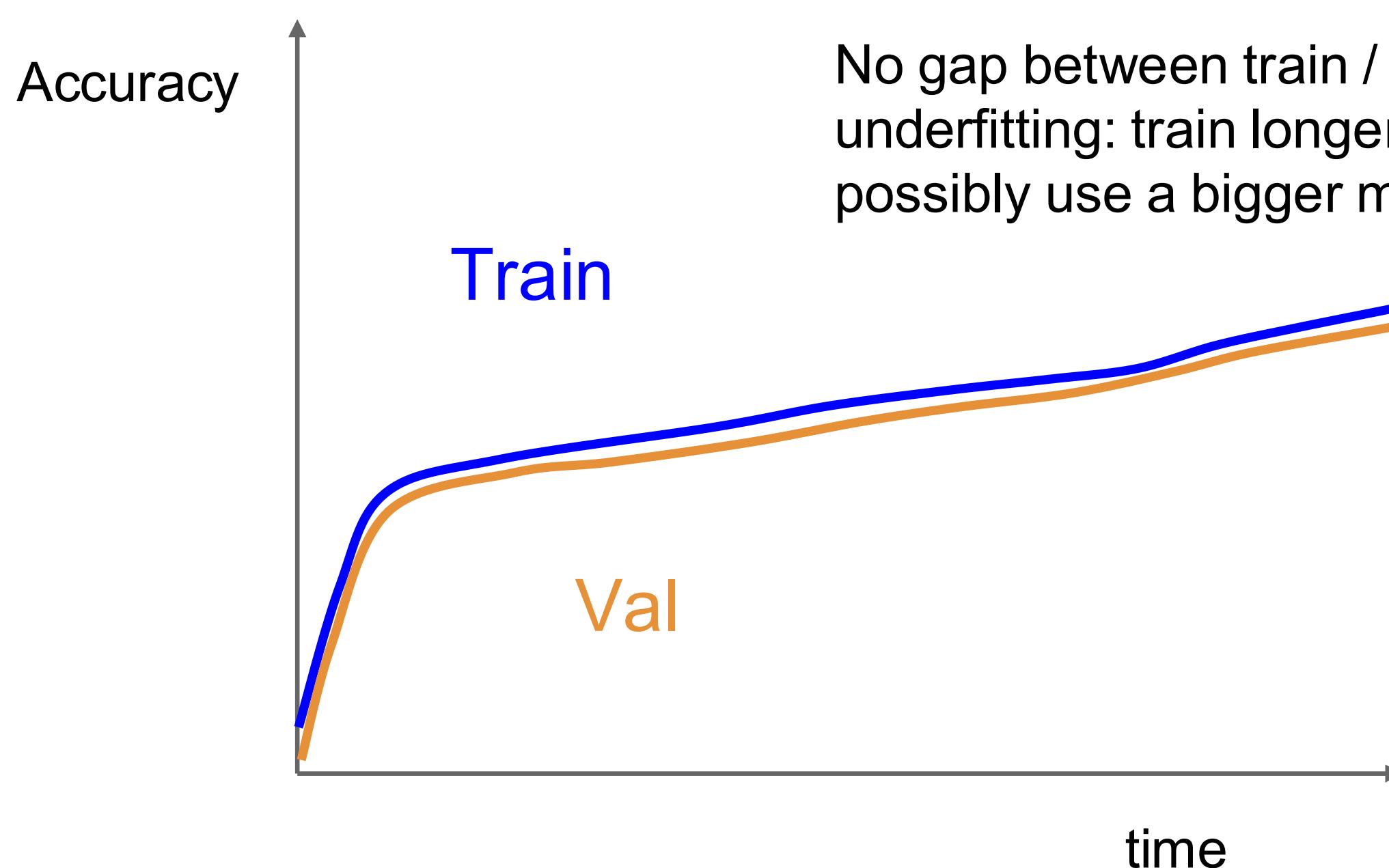
Val

time





Huge train / val gap means overfitting! Increase regularization or get more data



No gap between train / val means underfitting: train longer, can possibly use a bigger model

Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Step 4: Coarse grid, train for ~1-5 epochs

Step 5: Refine grid, train longer

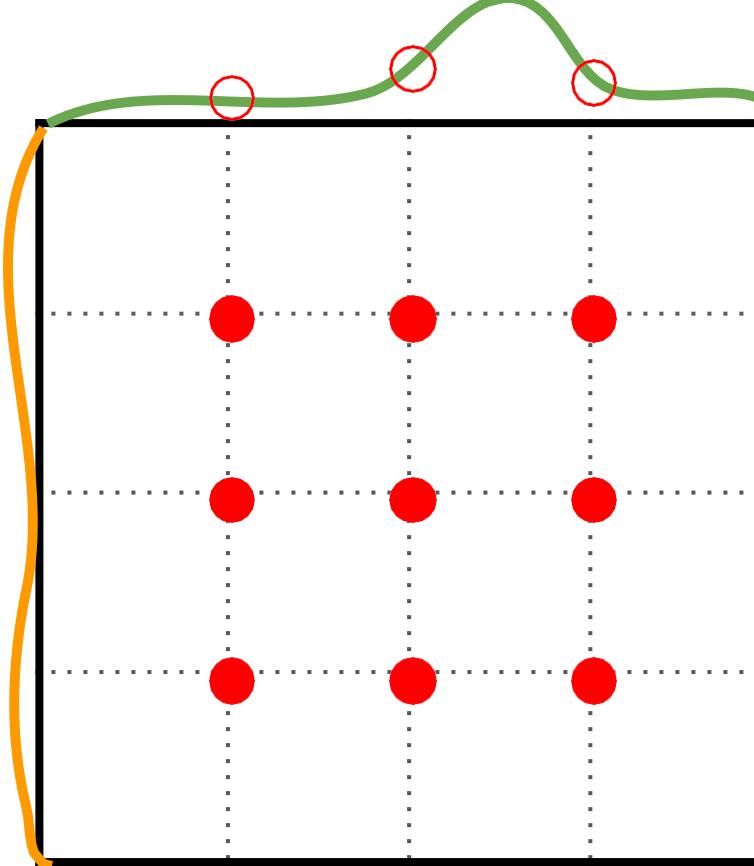
Step 6: Look at loss and accuracy curves

Step 7: **GOTO step 5**

Random Search vs. Grid Search

Random Search for Hyper-Parameter Optimization
Bergstra and Bengio, 2012

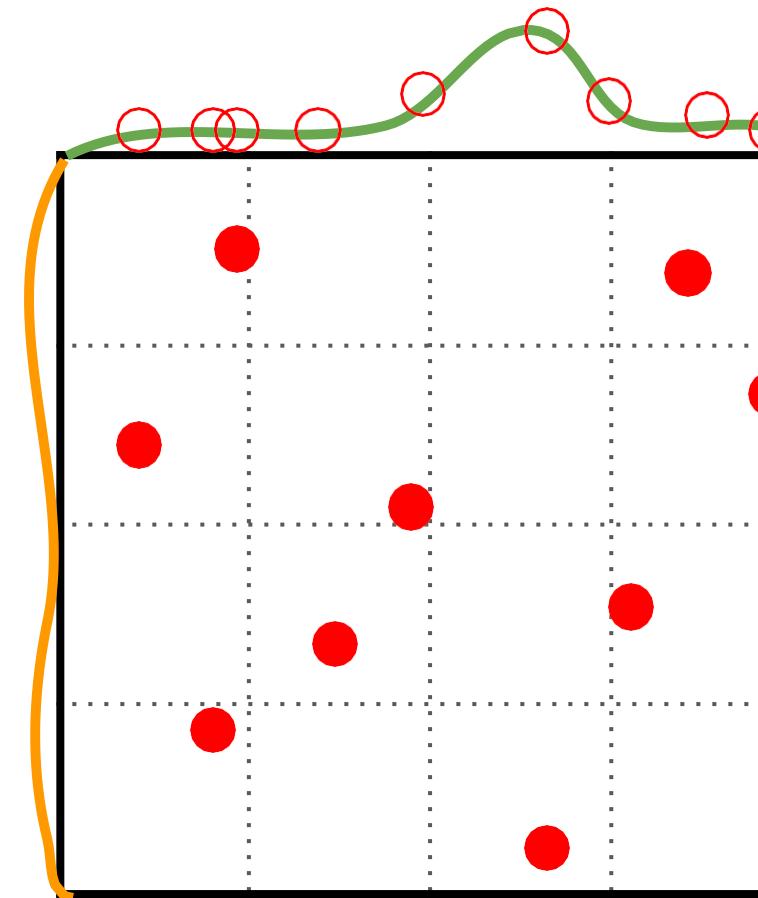
Grid Layout



Important Parameter

Unimportant Parameter

Random Layout



Important Parameter

Unimportant Parameter

Illustration of Bergstra et al., 2012 by Shayne
Longpre, copyright CS231n 2017

Summary

We reviewed 8 topics at a high level:

1. Layers in CNNs (Conv, FC, Norm, Dropout)
2. Activation Functions in NNs (ReLU, GELU, etc.)
3. CNN Architectures (VGG, ResNets)
4. Weight Initialization (Maintain Activation Distribution)

Summary

We reviewed 8 topics at a high level:

5. Data Preprocessing (subtract mean, divide std)
6. Data augmentation (cropping, jitter)
7. Transfer Learning (train on ImageNet first)
8. Hyperparameter (Checking Losses + Random Search)

Lecture 7: Recurrent Neural Networks



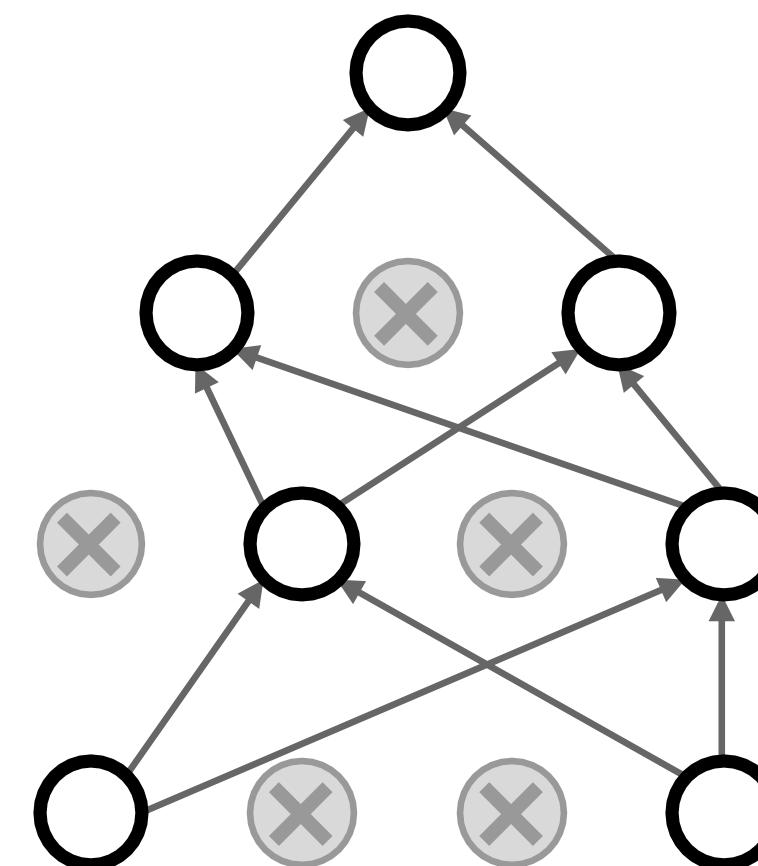
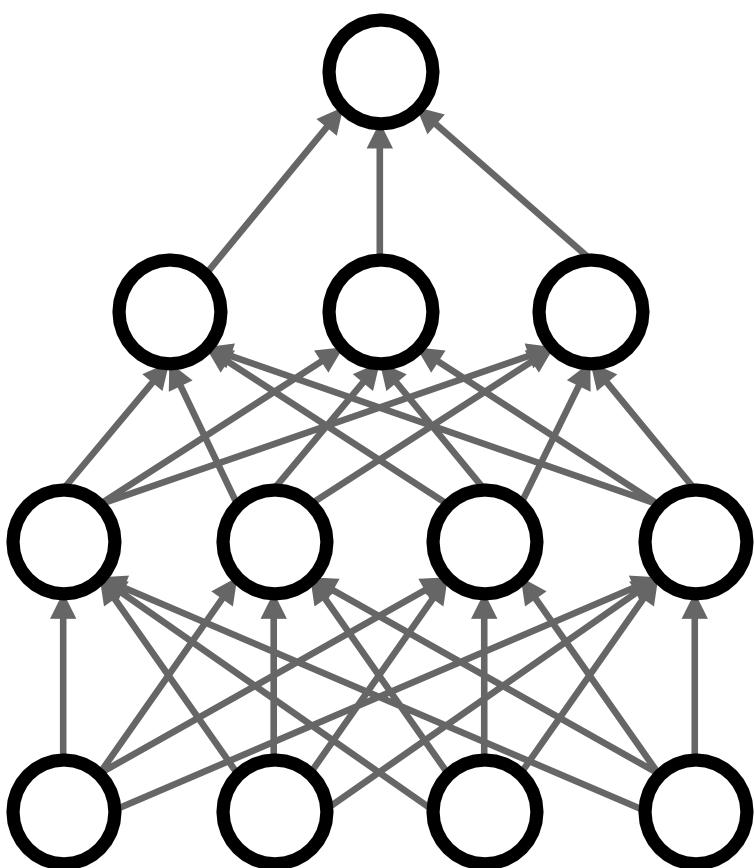
Clarifications from Last Time

- Dropout, how to scale probabilities at test-time
- Question in class about normalization vs weight initialization

Regularization: Dropout

In each forward pass, randomly set some neurons to zero

Probability of dropping is a hyperparameter; 0.5 is common



Srivastava et al, "Dropout: A simple way to prevent neural networks from overfitting", JMLR 2014

Dropout: Test time

```
def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

At test time all neurons are active always

=> We must scale the activations so that for each neuron:

output at test time = expected output at training time

Dropout Summary

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

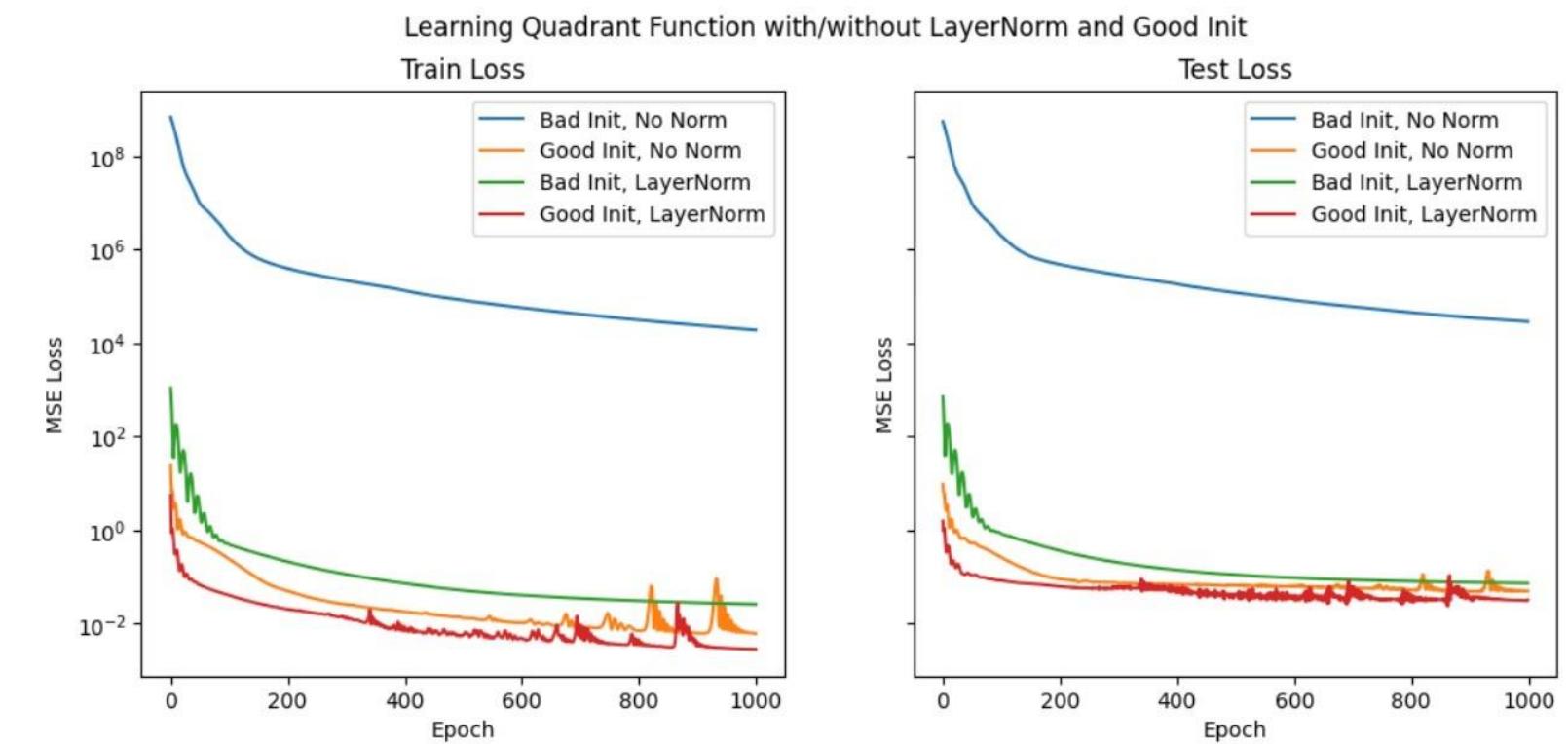
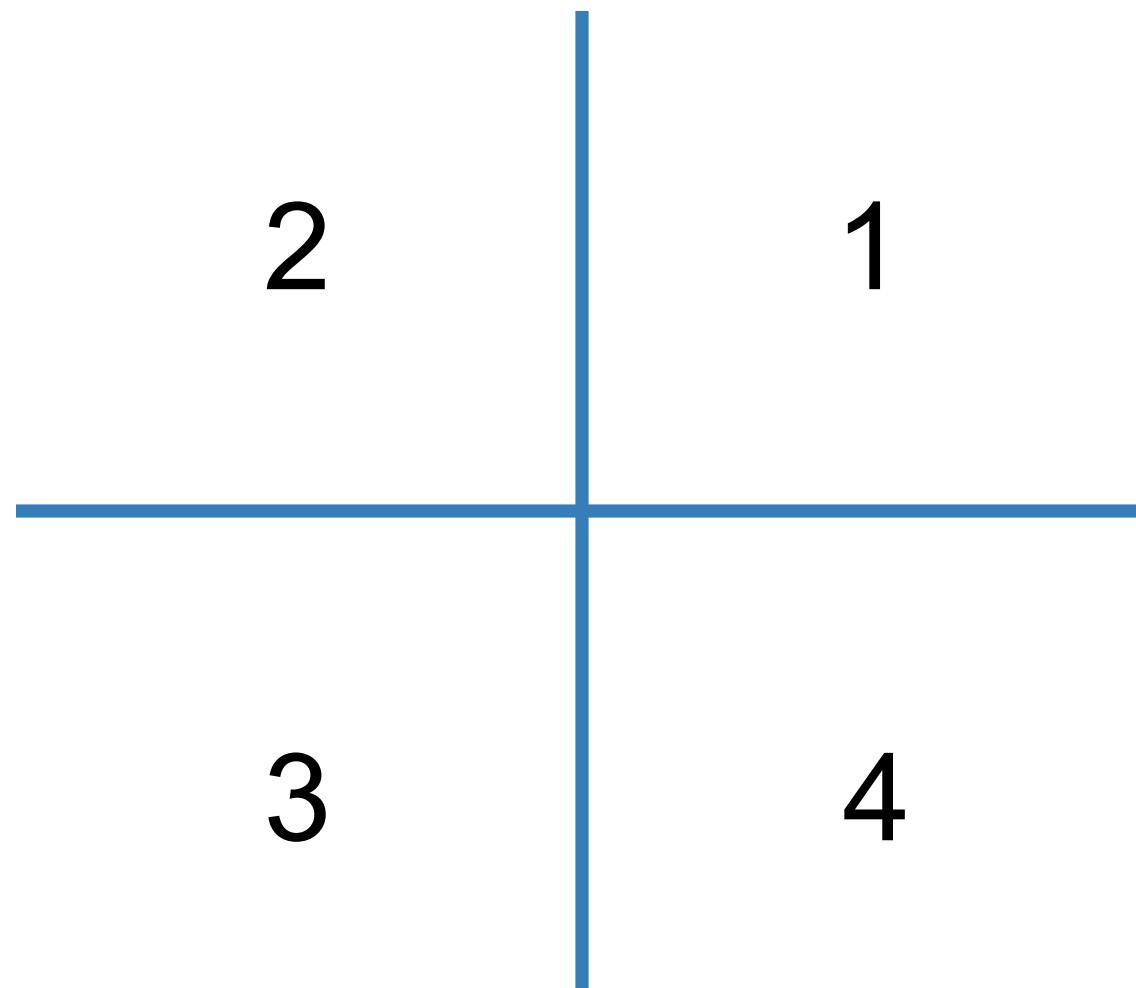
def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

drop in train time

scale at test time

Question in class: can normalization resolve the issues that arise with having weights initialized incorrectly?

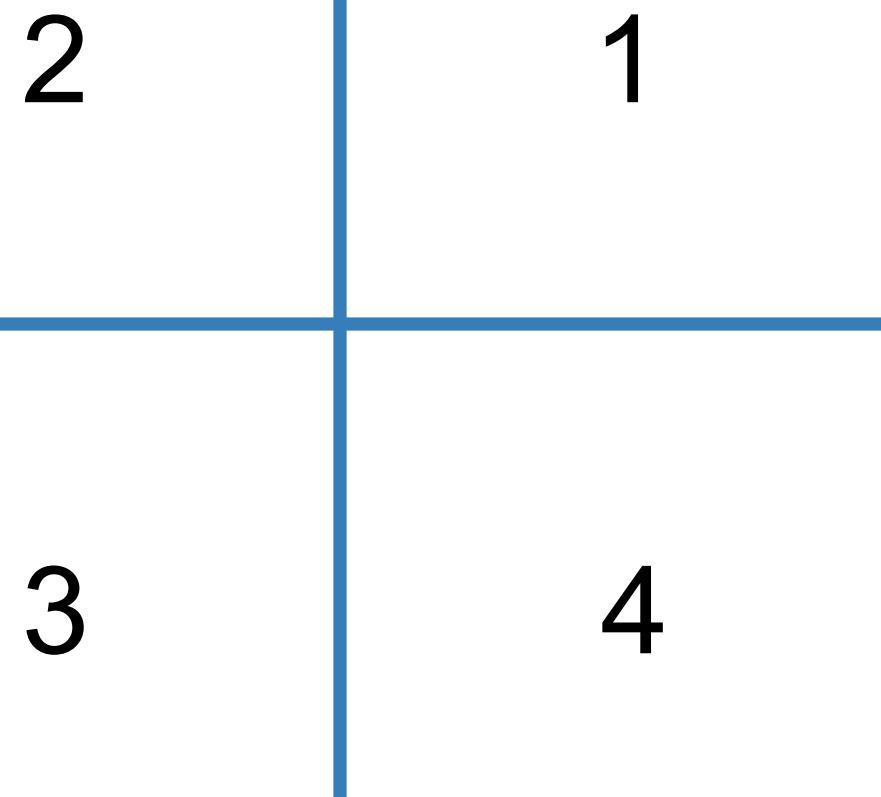
Toy setting, 2d input and 2-layer NN w/ ReLU



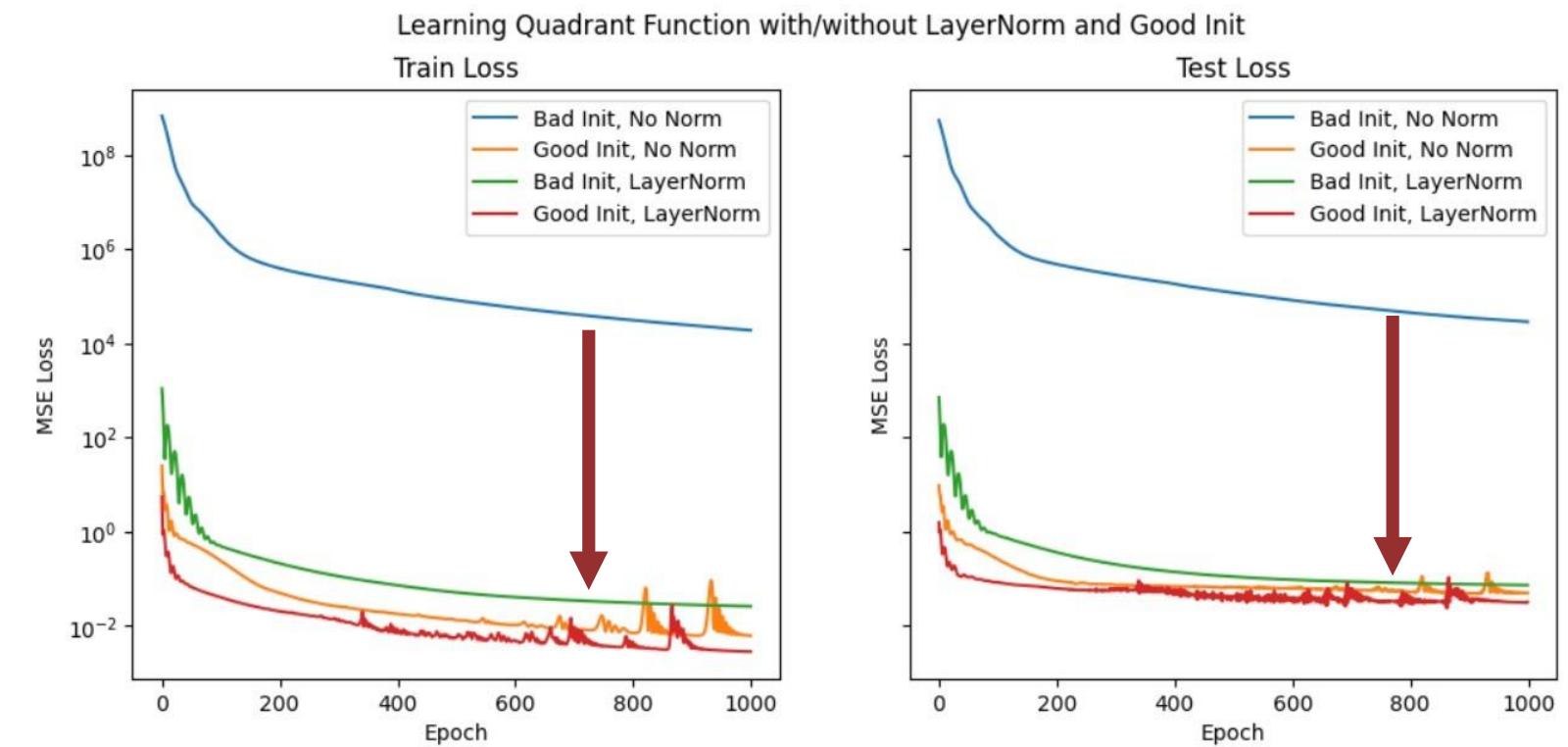
[Link to code \(colab\)](#)

Question in class: can normalization resolve the issues that arise with having weights initialized incorrectly?

Toy setting, 2d input and 2-layer NN w/ ReLU



Significant reduction in error!



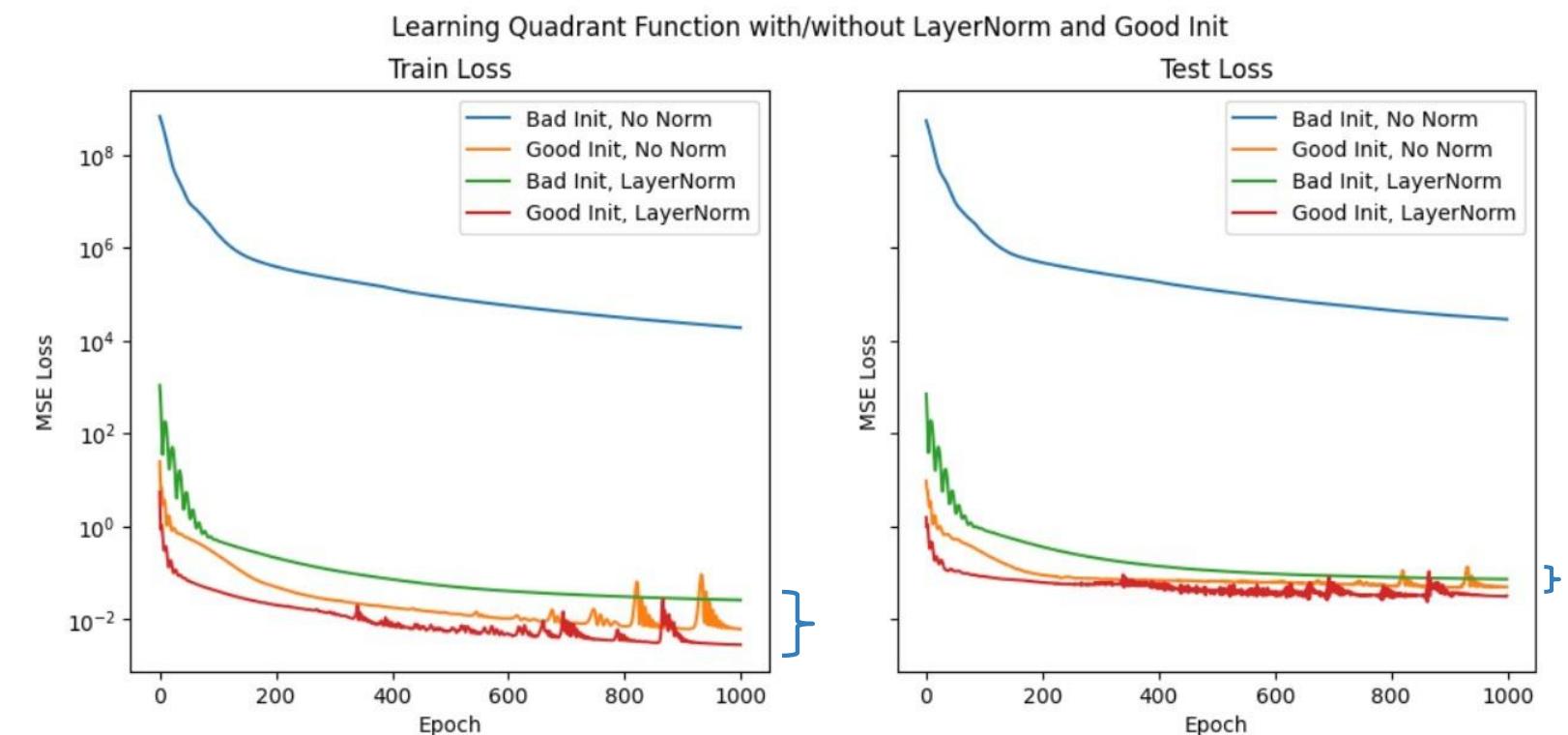
[Link to code \(colab\)](#)

Question in class: can normalization resolve the issues that arise with having weights initialized incorrectly?

Toy setting, 2d input and 2-layer NN w/ ReLU



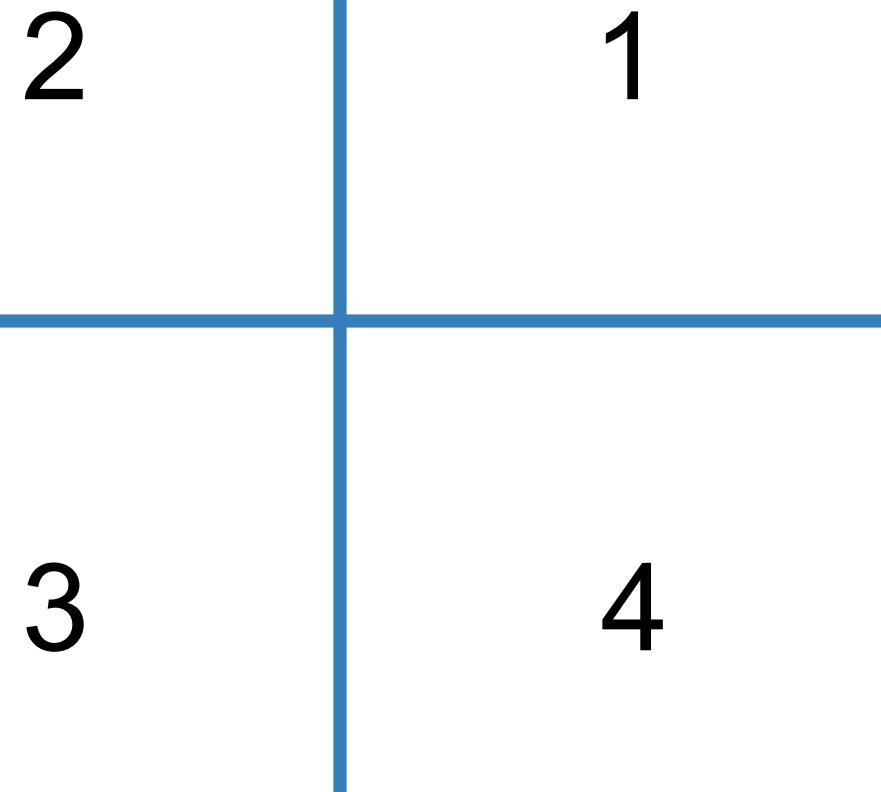
Performance gap still exists, does not resolve issues entirely, still optimization issues



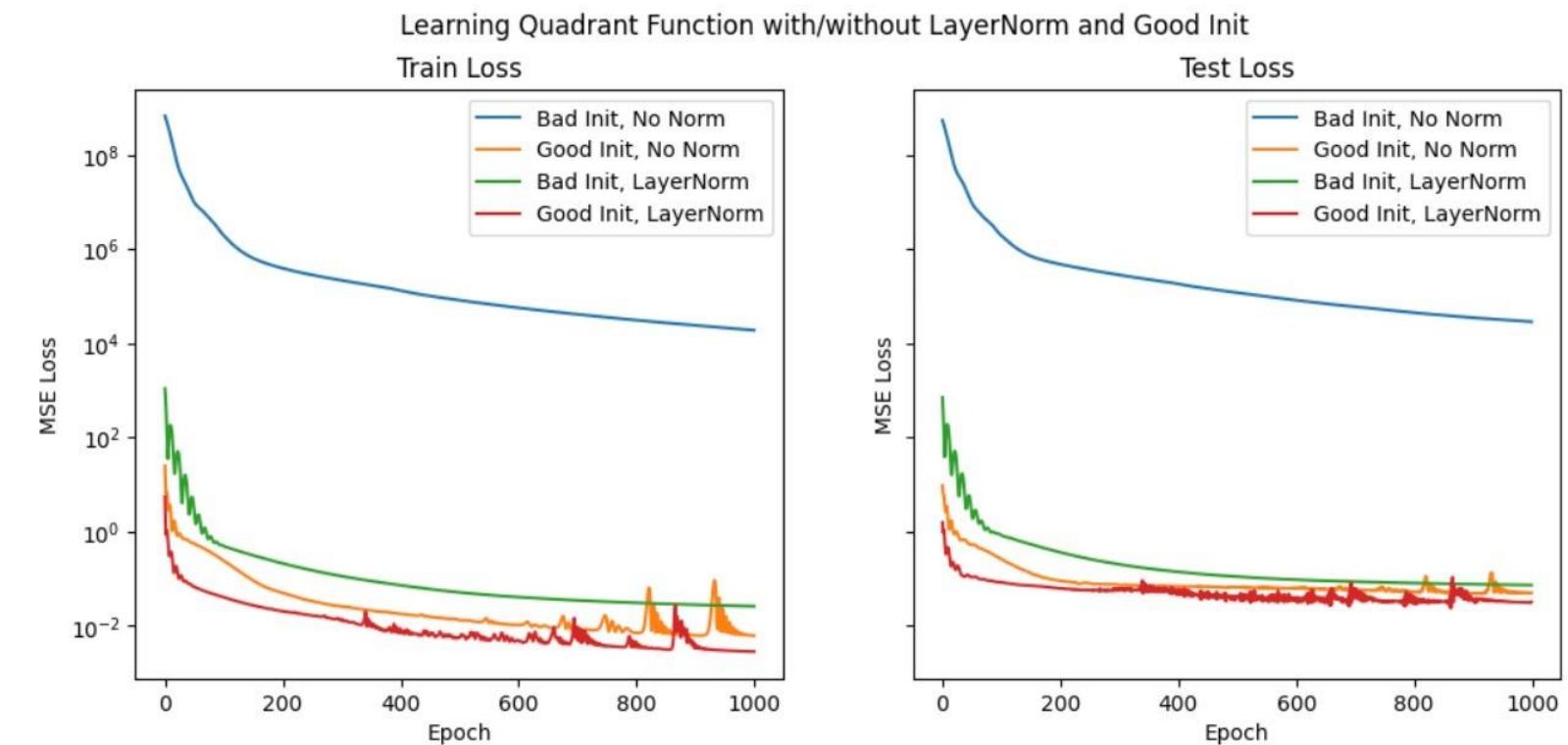
[Link to code \(colab\)](#)

Question in class: can normalization resolve the issues that arise with having weights initialized incorrectly?

Toy setting, 2d input and 2-layer NN w/ ReLU



Normalization may not always make sense! In this case, easy to see why it's helpful (LayerNorm does not change quadrant of inputs)

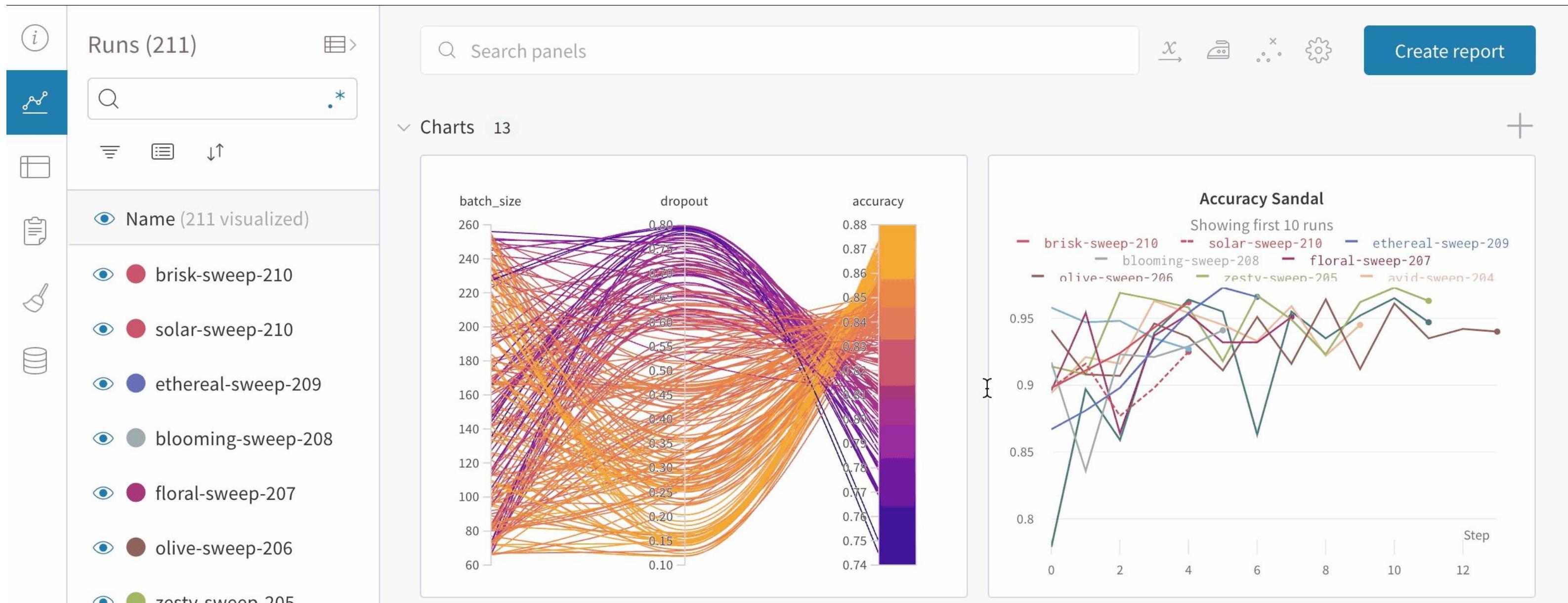


[Link to code \(colab\)](#)

Training Non-Recurrent Neural Networks

- 1. One time setup:** activation functions, preprocessing, weight initialization, normalization, transfer learning
- 2. Training dynamics:** babysitting the learning process, parameter updates, hyperparameter optimization
- 3. Evaluation:** validation performance, test-time augmentation

Evaluate models and tune hyperparameters



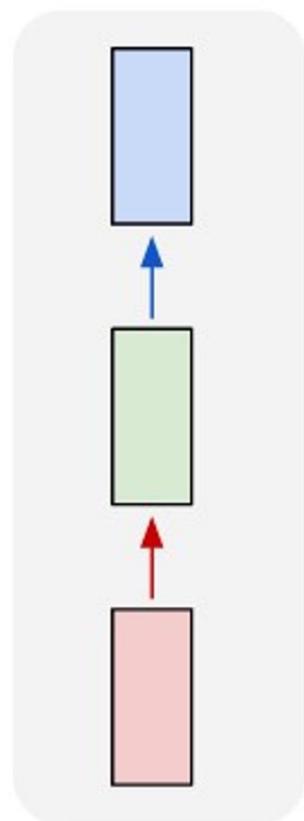
<https://docs.wandb.ai/guides/track/app>

Rest of Today's Lecture

- Discuss sequence modeling (assumed fixed-length inputs so far)
- Simple models commonly used before the era of transformers
 - RNNs and some variants
- Relation to modern state-space models (e.g. Mamba)

“Vanilla” Neural Network

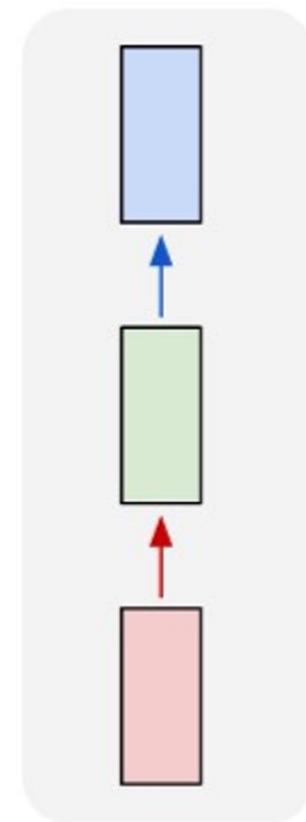
one to one



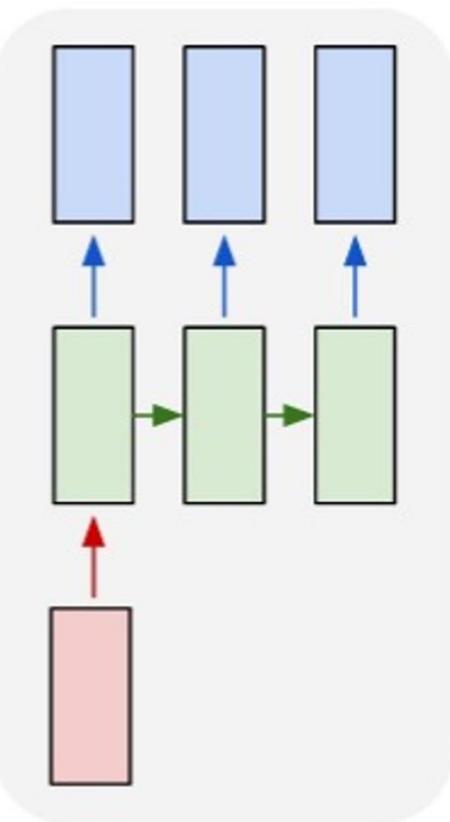
→ Vanilla Neural Networks

Recurrent Neural Networks: Process Sequences

one to one



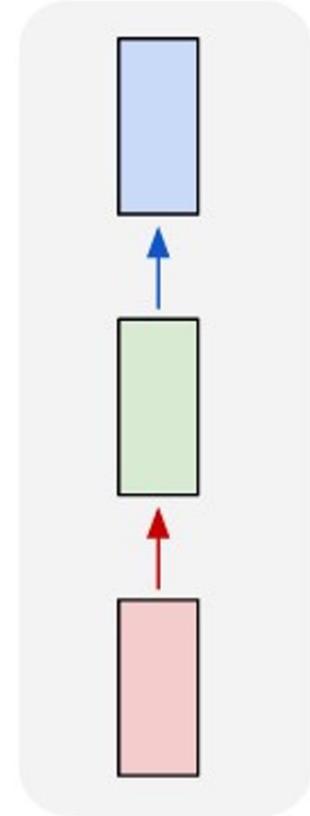
one to many



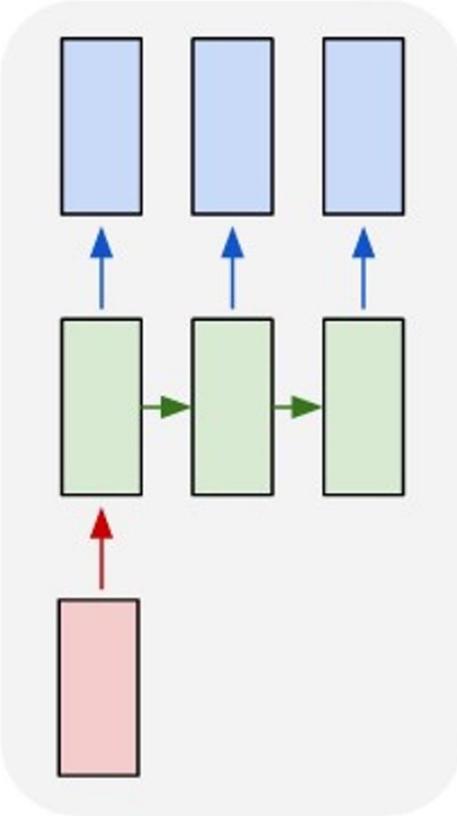
e.g. Image Captioning
image ->sequence of words

Recurrent Neural Networks: Process Sequences

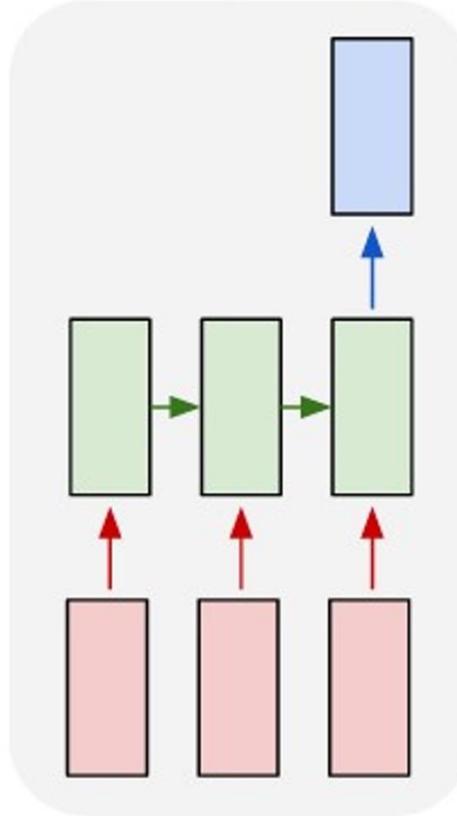
one to one



one to many



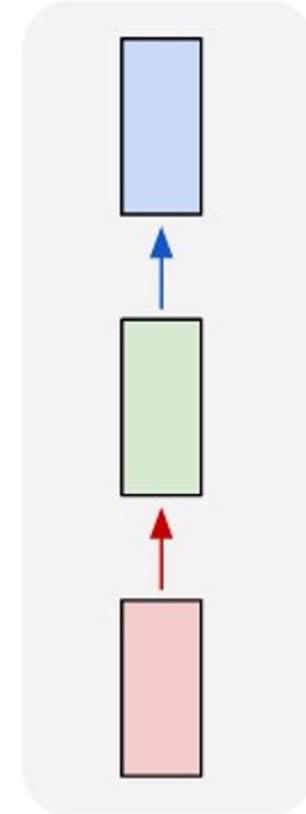
many to one



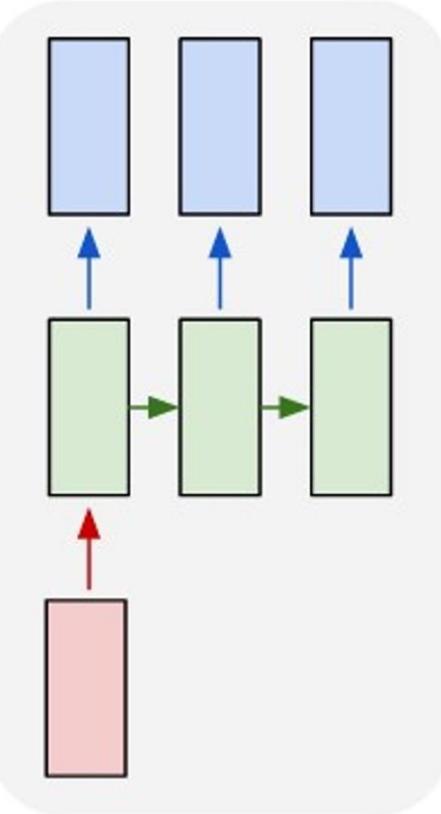
e.g. action prediction
sequence of video frames ->action class

Recurrent Neural Networks: Process Sequences

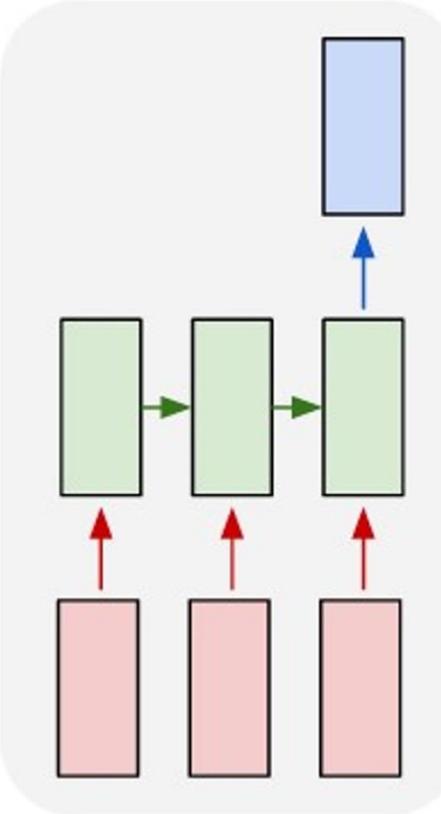
one to one



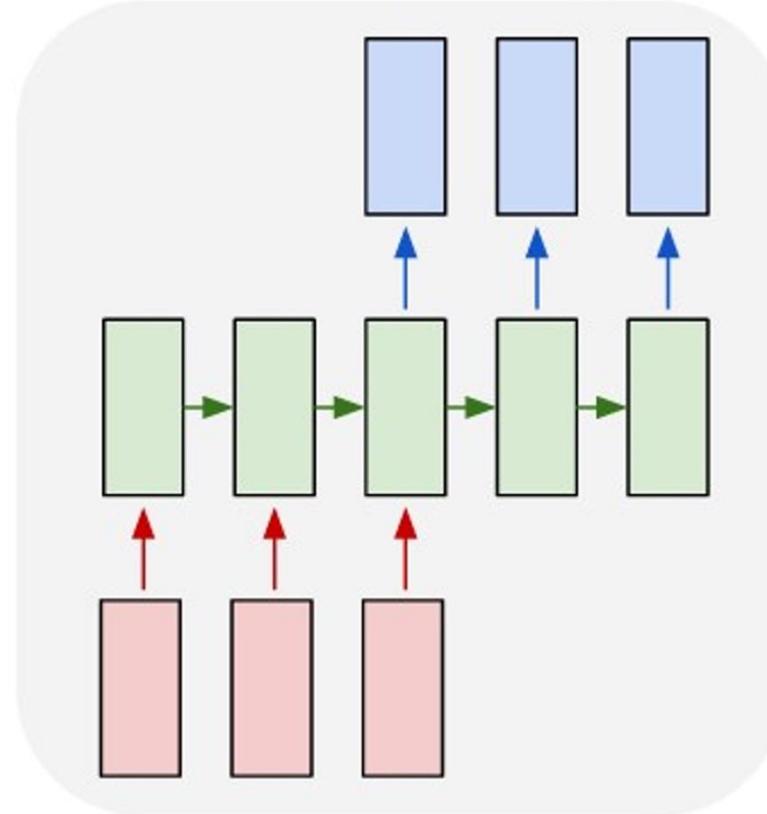
one to many



many to one



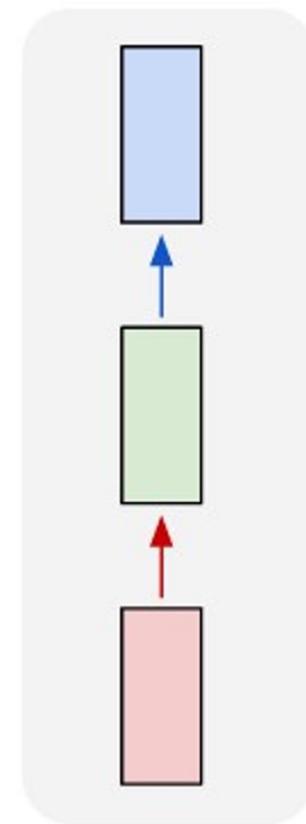
many to many



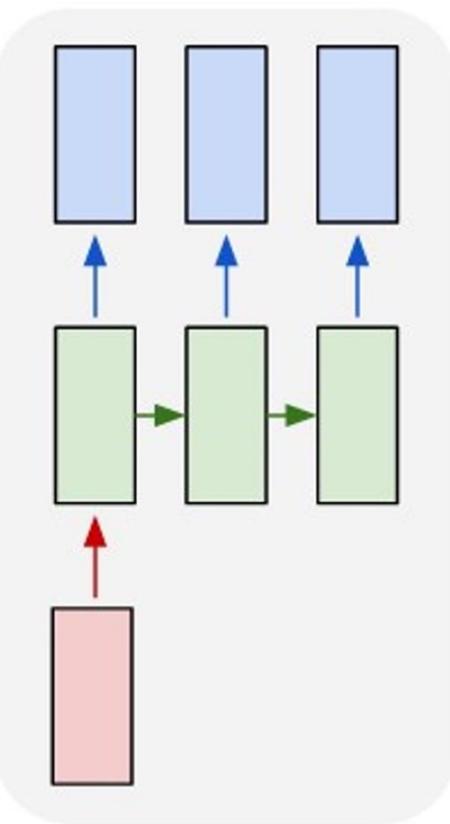
E.g. Video Captioning
Sequence of video frames ->caption

Recurrent Neural Networks: Process Sequences

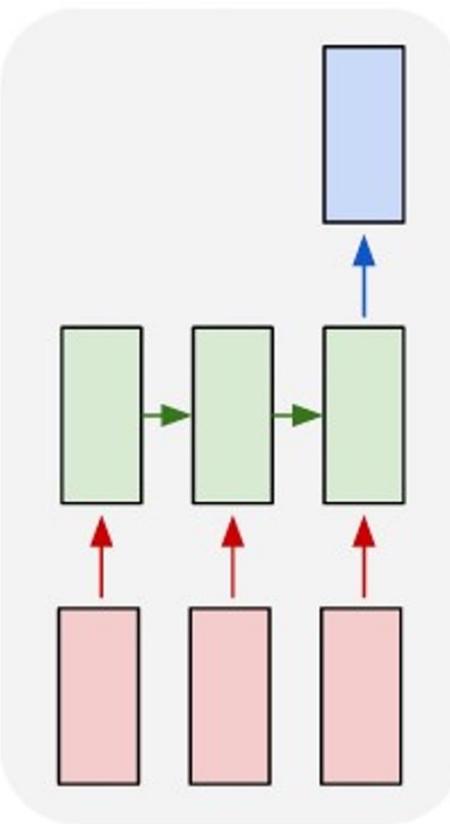
one to one



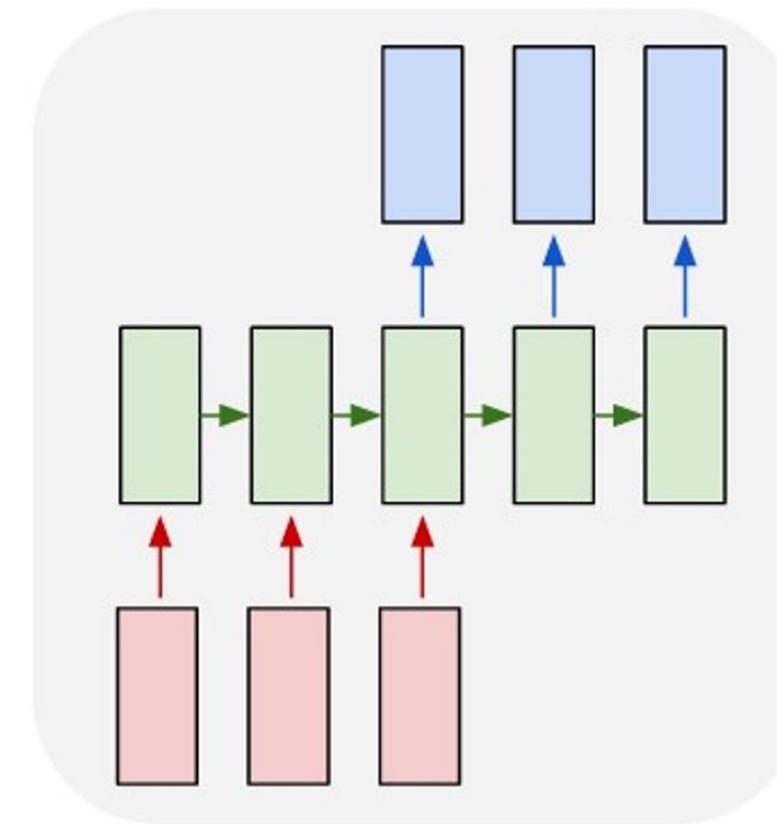
one to many



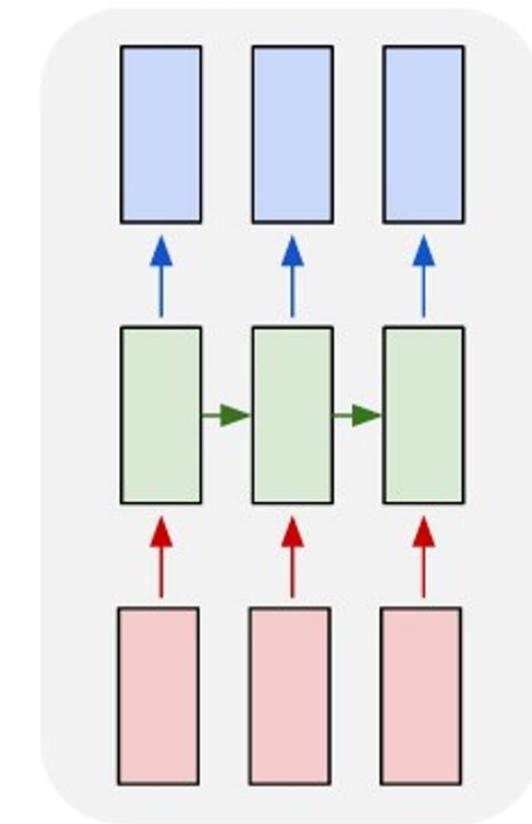
many to one



many to many

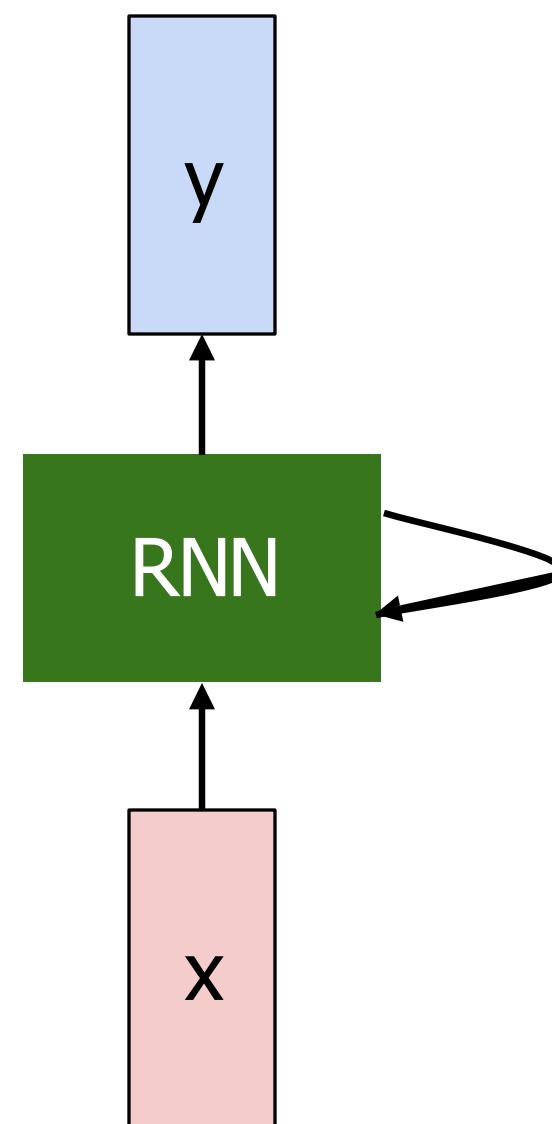


many to many

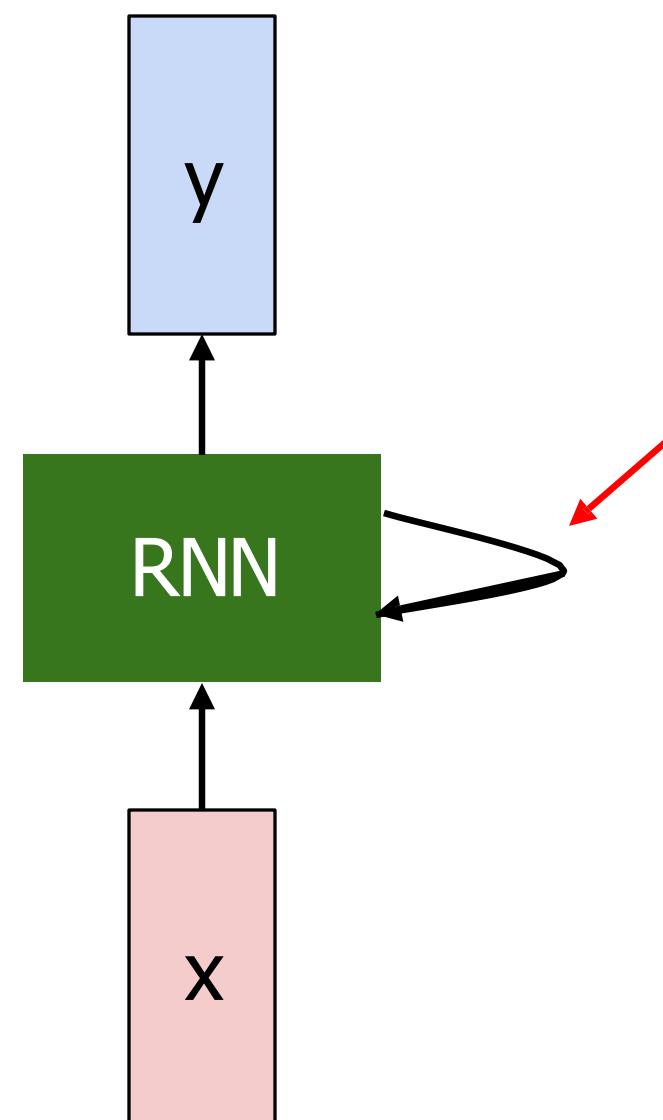


e.g. Video classification on frame level

Recurrent Neural Network

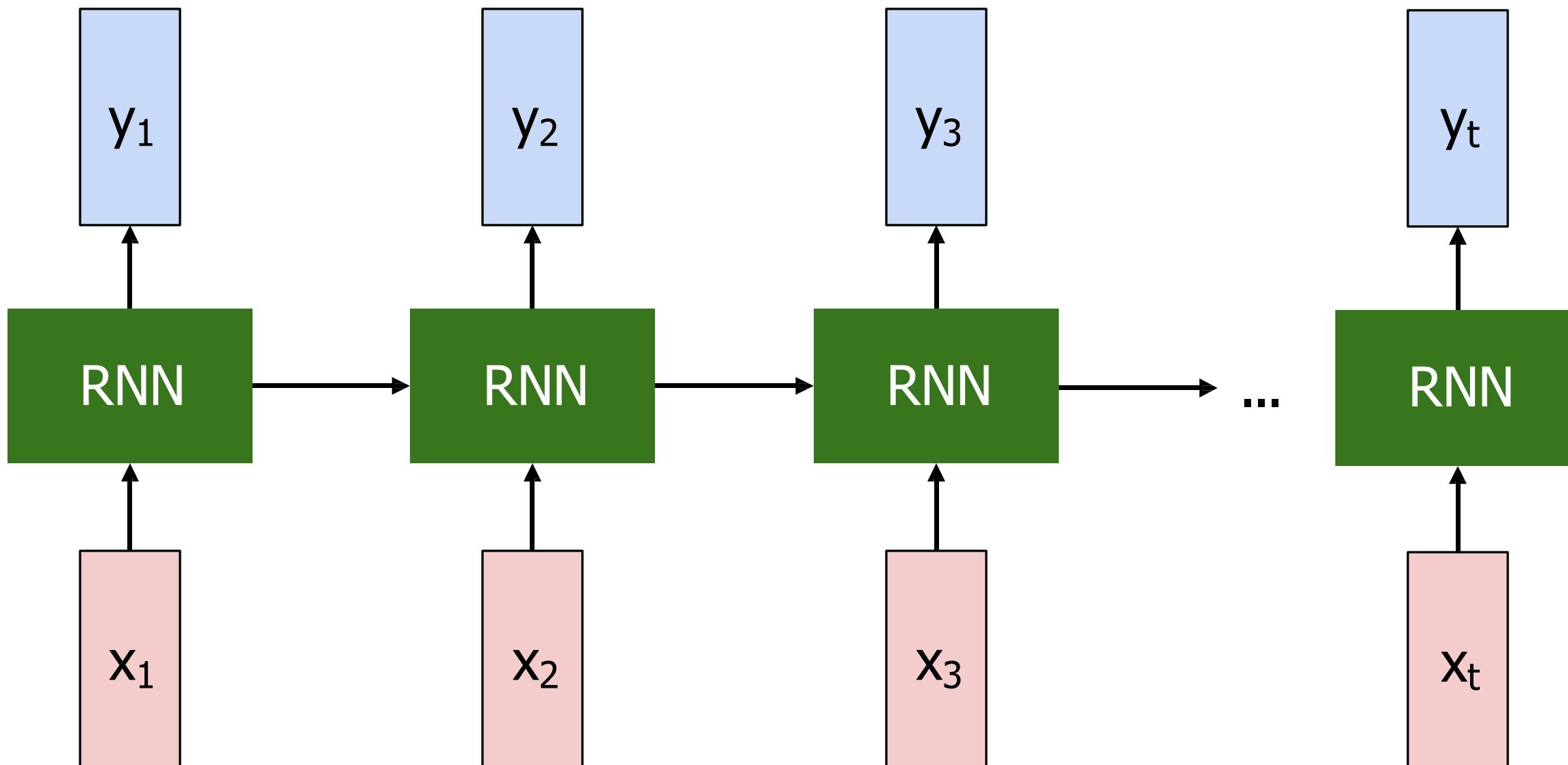


Recurrent Neural Network



Key idea: RNNs have an “internal state” that is updated as a sequence is processed

Unrolled RNN

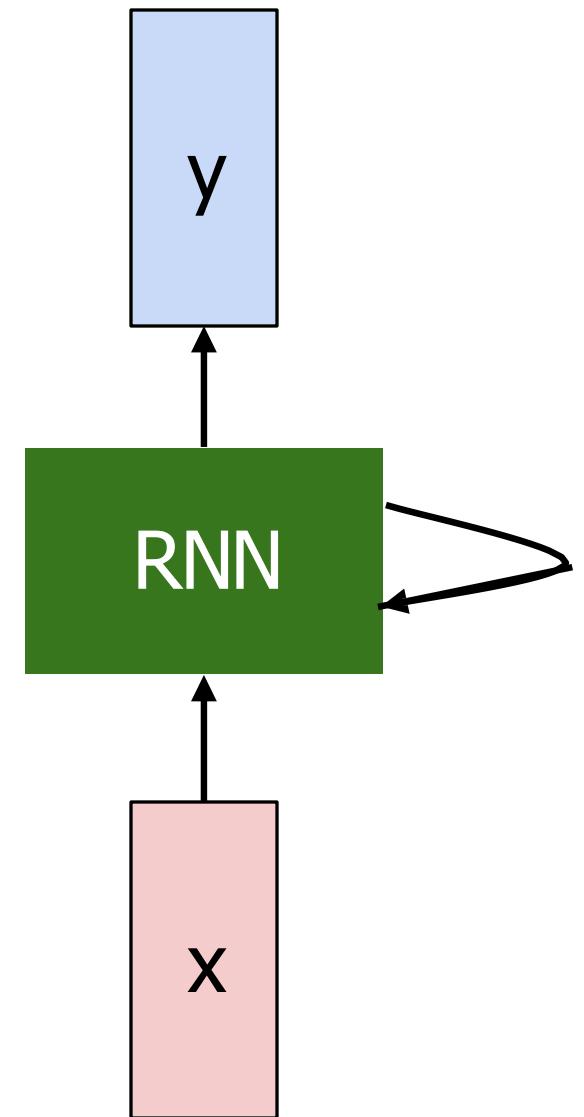


RNN hidden state update

We can process a sequence of vectors x by applying a recurrence formula at every time step:

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

new state old state input vector at
some function with parameters W some time step

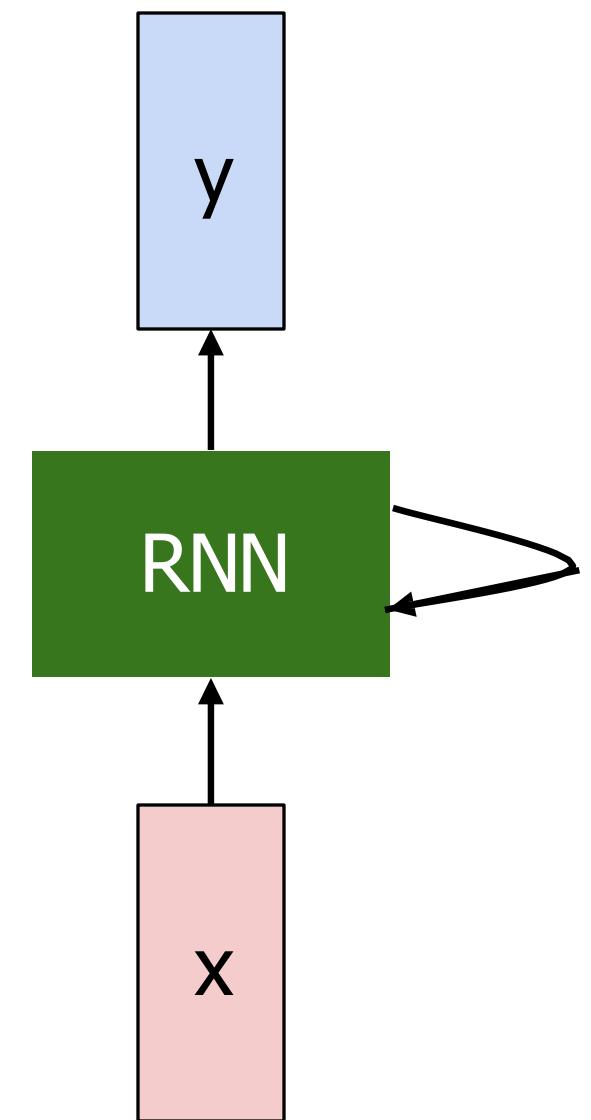


RNN output generation

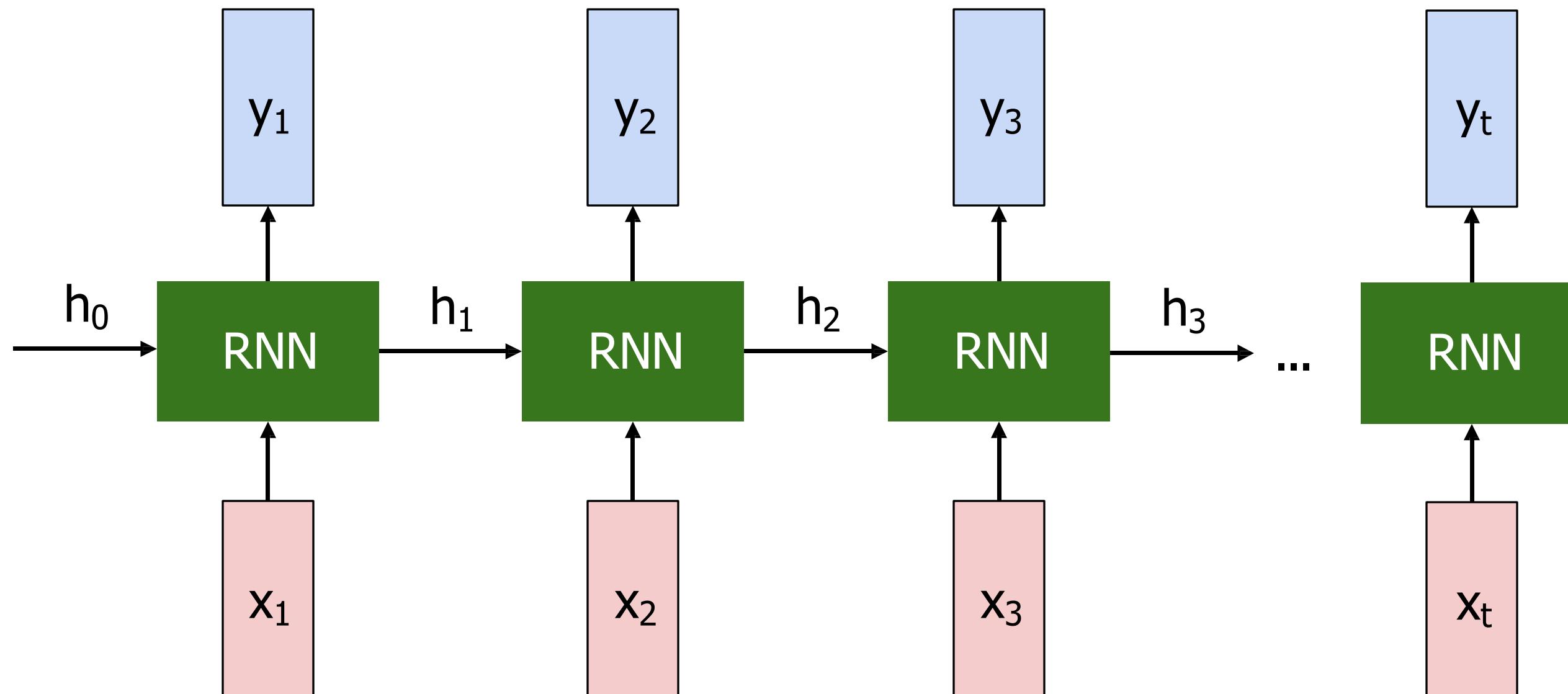
We can process a sequence of vectors x by applying a recurrence formula at every time step:

$$y_t = f_{W_{hy}}(h_t)$$

output new state
another function
with parameters W_{hy}



Recurrent Neural Network

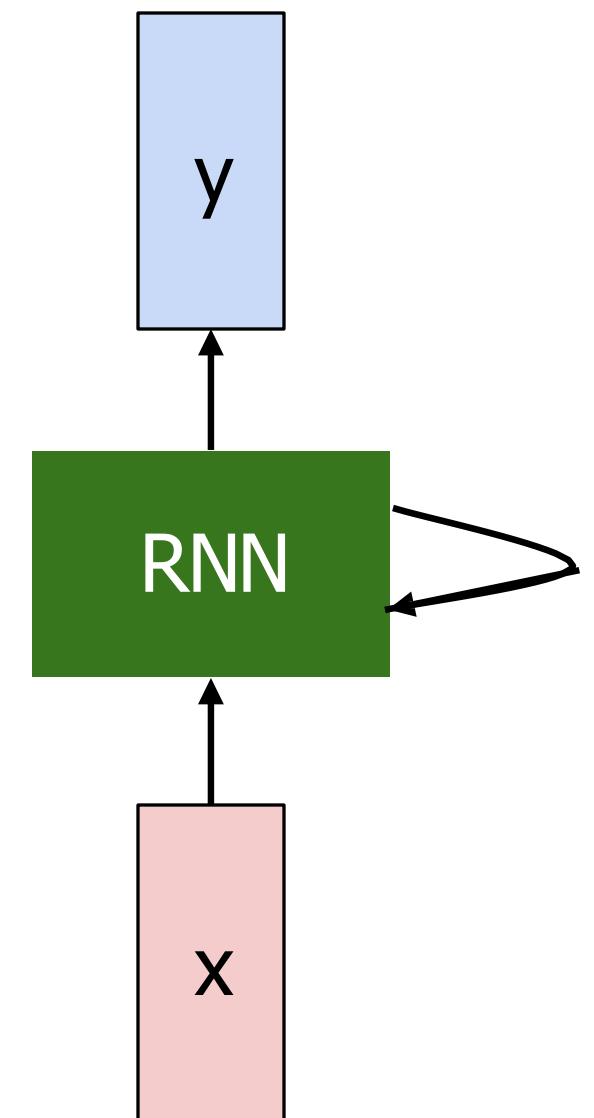


Recurrent Neural Network

We can process a sequence of vectors x by applying a recurrence formula at every time step:

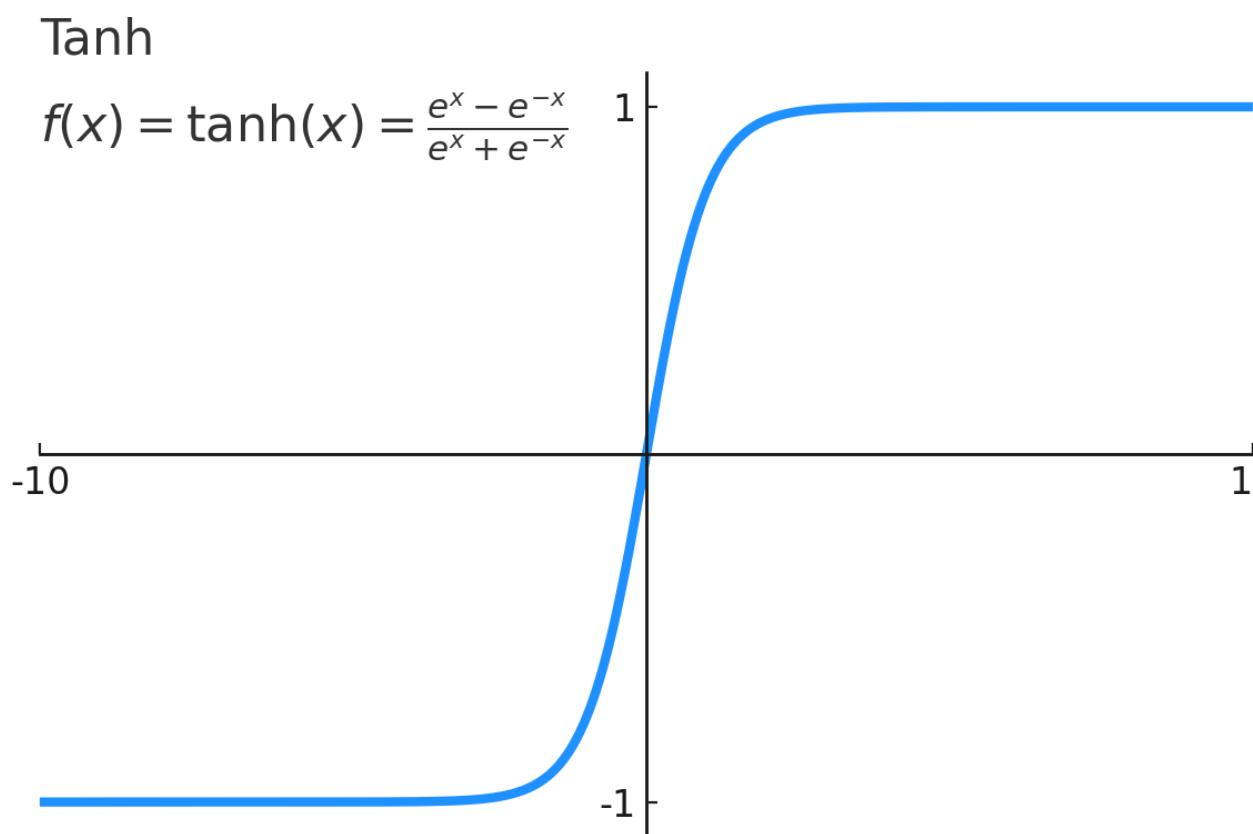
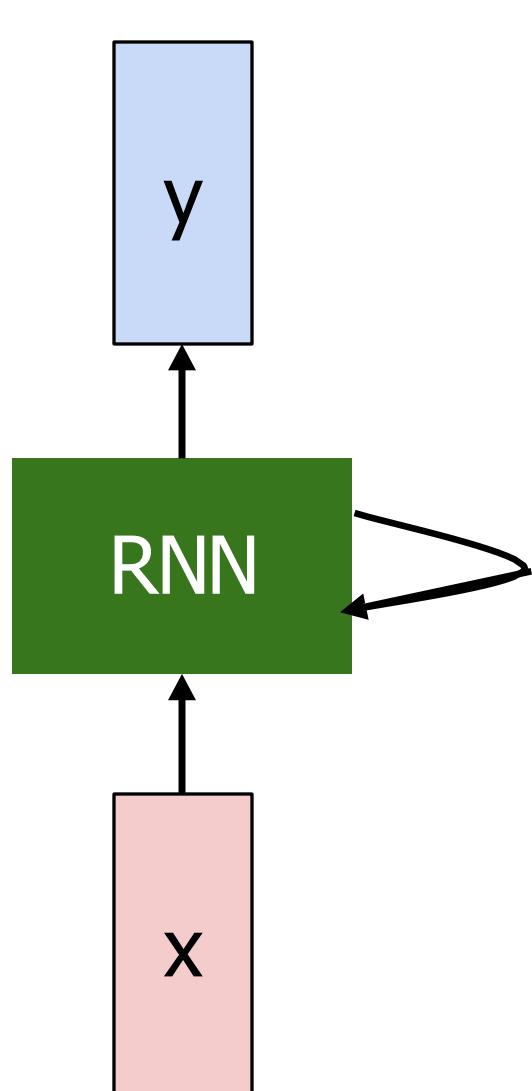
$$h_t = f_W(h_{t-1}, x_t)$$

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



(Vanilla) Recurrent Neural Network

The state consists of a single “hidden” vector h :



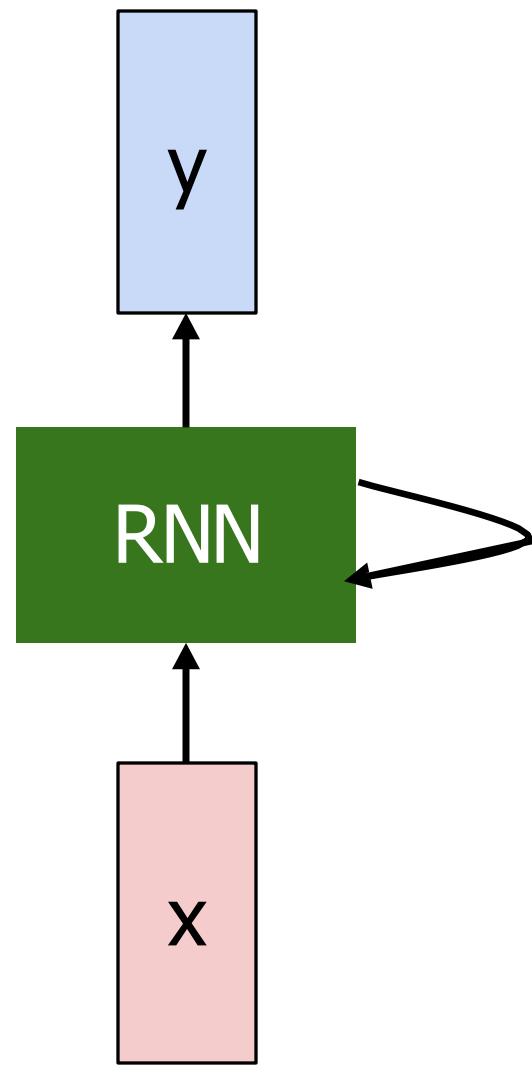
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \quad y_t = W_{hy}h_t$$

Sometimes called a “Vanilla RNN” or an “Elman RNN” after Prof. Jeffrey Elman

Often, we also have an output function: f_y

Vanilla(-ish) RNN: Concrete Example

Manually creating a recurrent network for detecting repeated 1s



X	RNN	Y
0		0
1		0
0		0
1		0
1		1
1		1
1		1
0		0
1		0
1		1

“Many to many” sequence modeling task

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

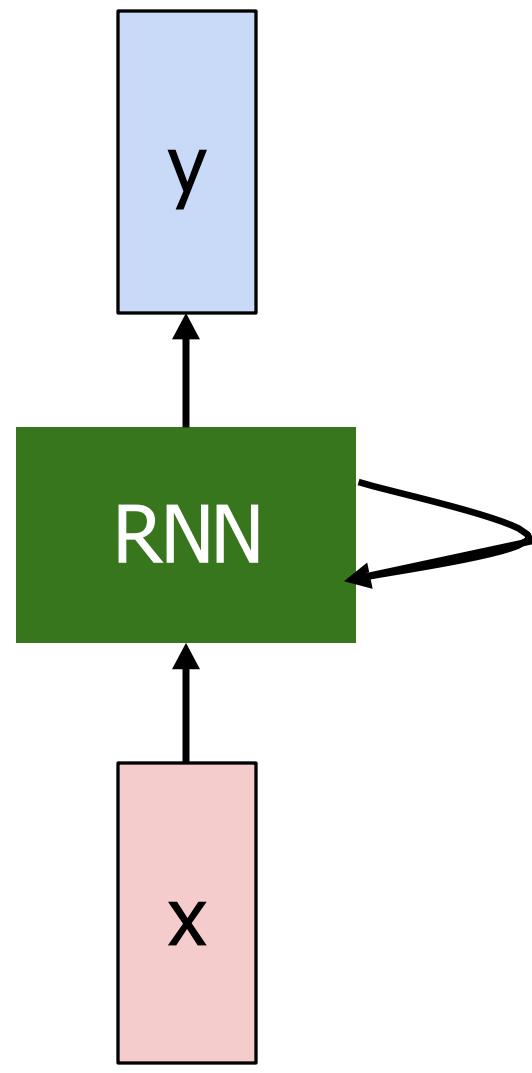
$$y_t = f_y(W_{hy}h_t)$$

How could we create an RNN to do this?

Q: What information should be captured in the “hidden” state?

Vanilla(-ish) RNN: Concrete Example

Manually creating a recurrent network for detecting repeated 1s



X	RNN	Y
0		0
1		0
0		0
1		0
1		1
1		1
1		1
0		0
1		0
1		1

“Many to many” sequence modeling task

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

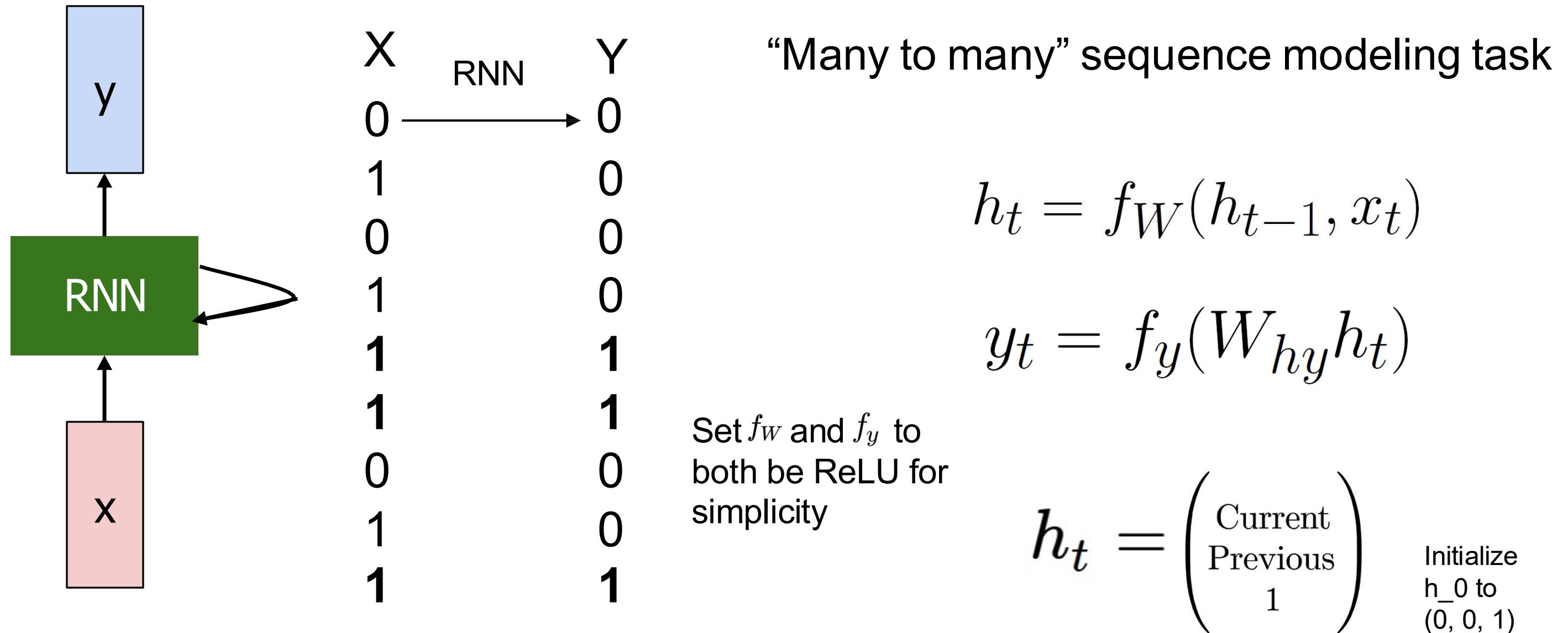
$$y_t = f_y(W_{hy}h_t)$$

How could we create an RNN to do this?

A: Previous input and current value for x

Vanilla(-ish) RNN: Concrete Example

Manually creating a recurrent network for detecting repeated 1s



Vanilla(-ish) RNN: Concrete Example

Manually creating a recurrent network for detecting repeated 1s

```
w_xh = np.array([[1], [0], [0]])  
w_hh = np.array([[0, 0, 0],  
                 [1, 0, 0],  
                 [0, 0, 1]])  
  
w_yh = np.array([1, 1, -1])  
  
x_seq = [0, 1, 0, 1, 1, 1, 0, 1, 1]  
  
h_t_prev = np.array([[0], [0], [1]])  
  
for t, x in enumerate(x_seq):  
    h_t = relu(w_hh @ h_t_prev + (w_xh @ x))  
    y_t = relu(w_yh @ h_t)  
    h_t_prev = h_t
```

$$h_t = \text{ReLU}(W_{hh}h_{t-1} + W_{xh}x_t) \quad h_t = \begin{pmatrix} \text{Current} \\ \text{Previous} \\ 1 \end{pmatrix}$$
$$y_t = \text{ReLU}(W_{yh}h_t)$$

X	Y
0	0
1	0
0	0
1	0
1	1
1	1
0	0
1	0
1	1

* Code is missing parts, for full code see [here](#)

Vanilla(-ish) RNN: Concrete Example

Manually creating a recurrent network for detecting repeated 1s

```
w_xh = np.array([[1], [0], [0]])
```

```
w_hh = np.array([[0, 0, 0],  
                 [1, 0, 0],  
                 [0, 0, 1]])
```

```
w_yh = np.array([1, 1, -1])
```

```
x_seq = [0, 1, 0, 1, 1, 1, 0, 1, 1]
```

```
h_t_prev = np.array([[0], [0], [1]])
```

```
for t, x in enumerate(x_seq):  
    h_t = relu(w_hh @ h_t_prev + (w_xh @ x))  
    y_t = relu(w_yh @ h_t)  
    h_t_prev = h_t
```

* Code is missing parts, for full code see [here](#)

$$h_t = \text{ReLU}(W_{hh}h_{t-1} + W_{xh}x_t) \quad h_t = \begin{pmatrix} \text{Current} \\ \text{Previous} \\ 1 \end{pmatrix}$$
$$y_t = \text{ReLU}(W_{yh}h_t)$$

Right hand term
 $x=0 \rightarrow [0, 0, 0]$
 $X=1 \rightarrow [1, 0, 0]$

X	Y
0	0
1	0
0	0
1	0
1	1
0	0
1	0
1	1

Vanilla(-ish) RNN: Concrete Example

Manually creating a recurrent network for detecting repeated 1s

```
w_xh = np.array([[1], [0], [0]])  
w_hh = np.array([[0, 0, 0],  
                 [1, 0, 0],  
                 [0, 0, 1]])  
w_yh = np.array([1, 1, -1])  
x_seq = [0, 1, 0, 1, 1, 1, 0, 1, 1]  
  
h_t_prev = np.array([[0], [0], [1]])  
  
for t, x in enumerate(x_seq):  
    h_t = relu(w_hh @ h_t_prev + (w_xh @ x))  
    y_t = relu(w_yh @ h_t)  
    h_t_prev = h_t
```

0 for top row of left term (only use value from right term)

$$h_t = \text{ReLU}(W_{hh}h_{t-1} + W_{xh}x_t) \quad h_t = \begin{pmatrix} \text{Current} \\ \text{Previous} \\ 1 \end{pmatrix}$$
$$y_t = \text{ReLU}(W_{yh}h_t)$$

X	Y
0	0
1	0
0	0
1	0
1	1
1	1
0	0
1	0
1	1

* Code is missing parts, for full code see [here](#)

Vanilla(-ish) RNN: Concrete Example

Manually creating a recurrent network for detecting repeated 1s

```
w_xh = np.array([[1], [0], [0]])  
w_hh = np.array([[0, 0, 0],  
                 [1, 0, 0],  
                 [0, 0, 1]])  
  
w_yh = np.array([1, 1, -1])  
  
x_seq = [0, 1, 0, 1, 1, 1, 0, 1, 1]  
  
h_t_prev = np.array([[0], [0], [1]])  
  
for t, x in enumerate(x_seq):  
    h_t = relu(w_hh @ h_t_prev + (w_xh @ x))  
    y_t = relu(w_yh @ h_t)  
    h_t_prev = h_t
```

Copy over “current” value from previous hidden state to be “previous”

$$h_t = \text{ReLU}(W_{hh}h_{t-1} + W_{xh}x_t) \quad h_t = \begin{pmatrix} \text{Current} \\ \text{Previous} \\ 1 \end{pmatrix}$$
$$y_t = \text{ReLU}(W_{yh}h_t)$$

X	Y
0	0
1	0
0	0
1	0
1	1
1	1
0	0
1	0
1	1

* Code is missing parts, for full code see [here](#)

Vanilla(-ish) RNN: Concrete Example

Manually creating a recurrent network for detecting repeated 1s

```
w_xh = np.array([[1], [0], [0]])  
w_hh = np.array([[0, 0, 0],  
                 [1, 0, 0],  
                 [0, 0, 1]])  
w_yh = np.array([1, 1, -1])  
  
x_seq = [0, 1, 0, 1, 1, 1, 0, 1, 1]  
  
h_t_prev = np.array([[0], [0], [1]])  
  
for t, x in enumerate(x_seq):  
    h_t = relu(w_hh @ h_t_prev + (w_xh @ x))  
    y_t = relu(w_yh @ h_t)  
    h_t_prev = h_t
```

Keep 1 on the bottom
(helpful for output)

$$h_t = \text{ReLU}(W_{hh}h_{t-1} + W_{xh}x_t)$$
$$y_t = \text{ReLU}(W_{yh}h_t)$$

$$h_t = \begin{pmatrix} \text{Current} \\ \text{Previous} \\ 1 \end{pmatrix}$$

X	Y
0	0
1	0
0	0
1	0
1	1
1	1
0	0
1	0
1	1

* Code is missing parts, for full code see [here](#)

Vanilla(-ish) RNN: Concrete Example

Manually creating a recurrent network for detecting repeated 1s

```
w_xh = np.array([[1], [0], [0]])
```

```
w_hh = np.array([[0, 0, 0],  
                 [1, 0, 0],  
                 [0, 0, 1]])
```

```
w_yh = np.array([1, 1, -1])
```

```
x_seq = [0, 1, 0, 1, 1, 1, 0, 1, 1]
```

```
h_t_prev = np.array([[0], [0], [1]])
```

```
for t, x in enumerate(x_seq):  
    h_t = relu(w_hh @ h_t_prev + (w_xh @ x))  
    y_t = relu(w_yh @ h_t)  
    h_t_prev = h_t
```

* Code is missing parts, for full code see [here](#)

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

$$y_t = \boxed{\text{ReLU}(W_{yh}h_t)}$$

$$h_t = \begin{pmatrix} \text{Current} \\ \text{Previous} \\ 1 \end{pmatrix}$$

Max(Current + Previous – 1, 0)

X	Y
0	0
1	0
0	0
1	0
1	1
1	1
0	0
1	0
1	1



Vanilla(-ish) RNN: Concrete Example

Manually creating a recurrent network for detecting repeated 1s

```
w_xh = np.array([[1], [0], [0]])
```

```
w_hh = np.array([[0, 0, 0],  
                 [1, 0, 0],  
                 [0, 0, 1]])
```

```
w_yh = np.array([1, 1, -1])
```

```
x_seq = [0, 1, 0, 1, 1, 1, 0, 1, 1]
```

```
h_t_prev = np.array([[0], [0], [1]])
```

```
for t, x in enumerate(x_seq):  
    h_t = relu(w_hh @ h_t_prev + (w_xh @ x))  
    y_t = relu(w_yh @ h_t)  
    h_t_prev = h_t
```

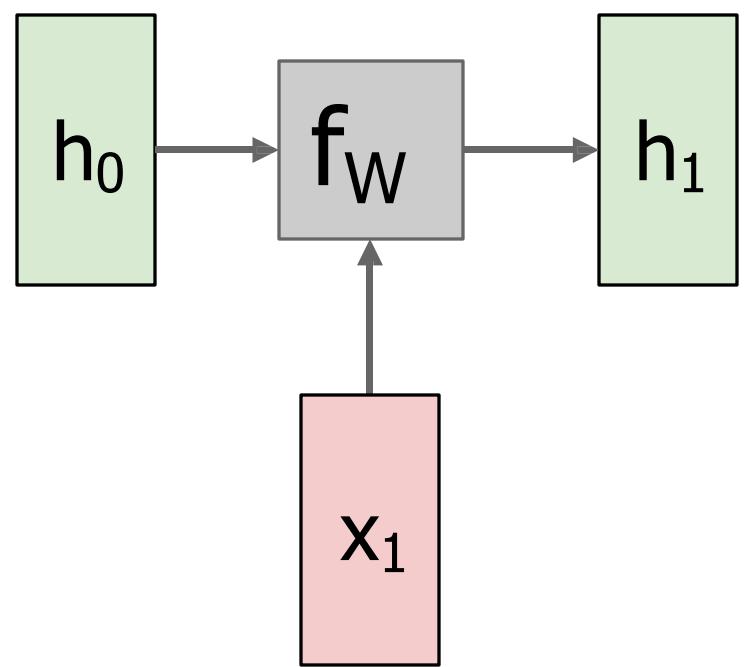
* Code is missing parts, for full code see [here](#)

$$h_t = \text{ReLU}(W_{hh}h_{t-1} + W_{xh}x_t) \quad h_t = \begin{pmatrix} \text{Current} \\ \text{Previous} \\ 1 \end{pmatrix}$$
$$y_t = \text{ReLU}(W_{yh}h_t)$$

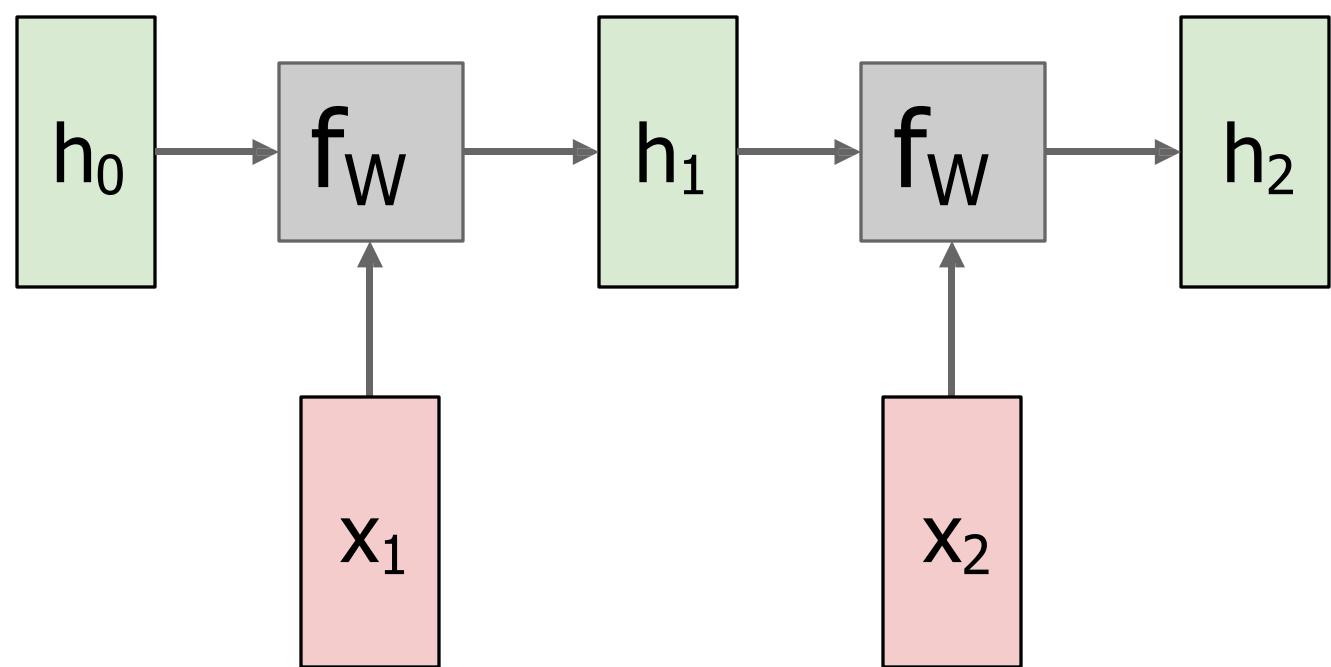
And it just works! But
how do we find the Ws
in practice?

X	Y
0	0
1	0
0	0
1	0
1	1
1	1
0	0
1	0
1	1

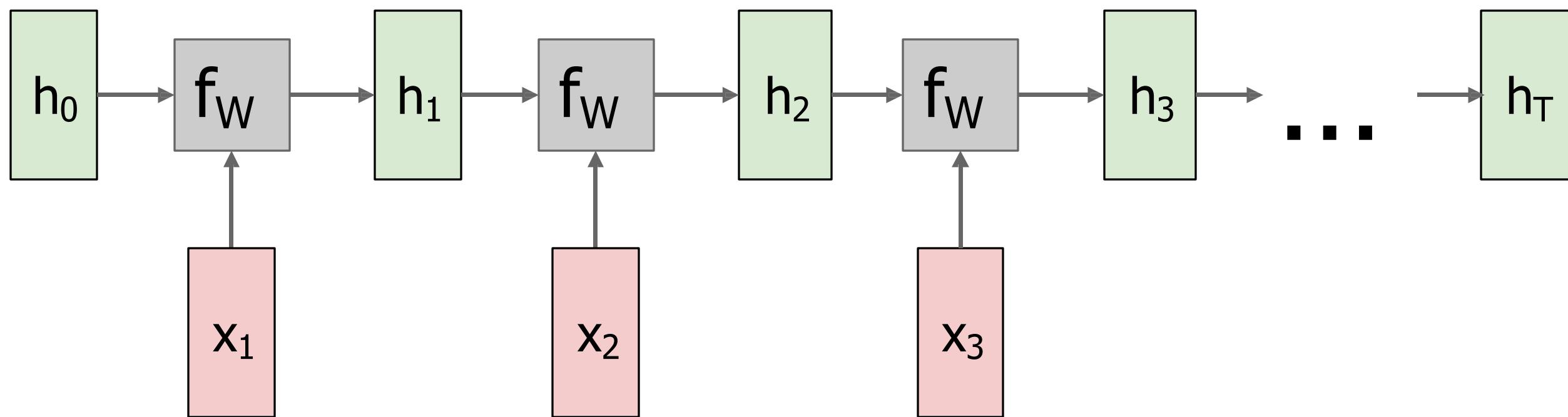
RNN: Computational Graph



RNN: Computational Graph

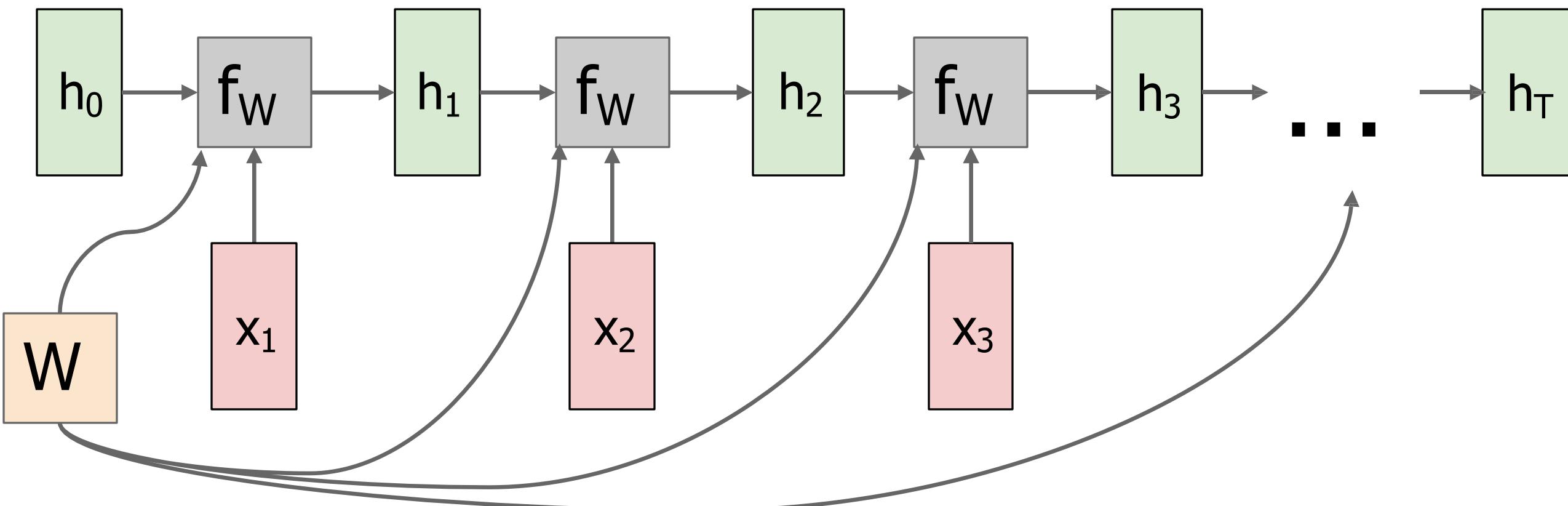


RNN: Computational Graph

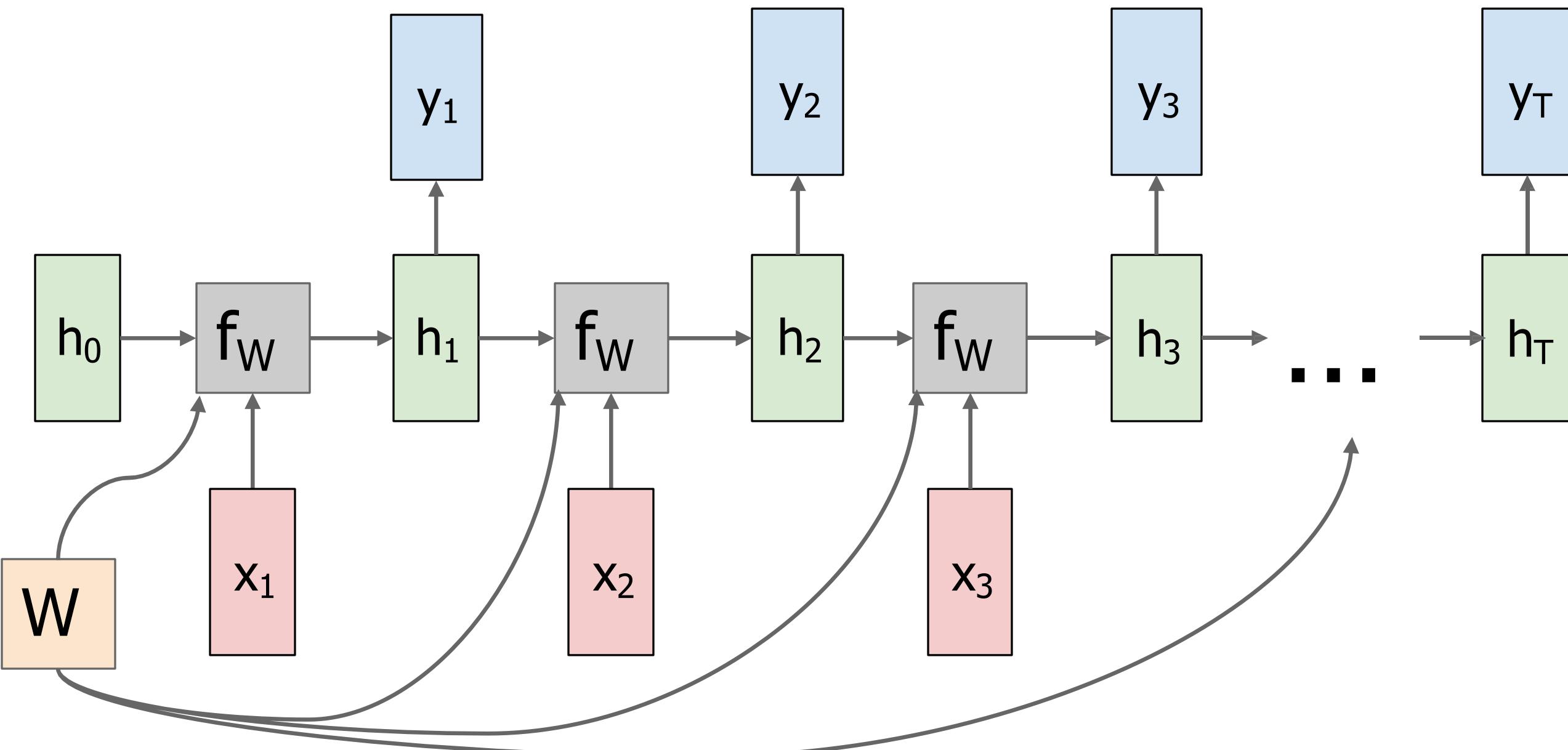


RNN: Computational Graph

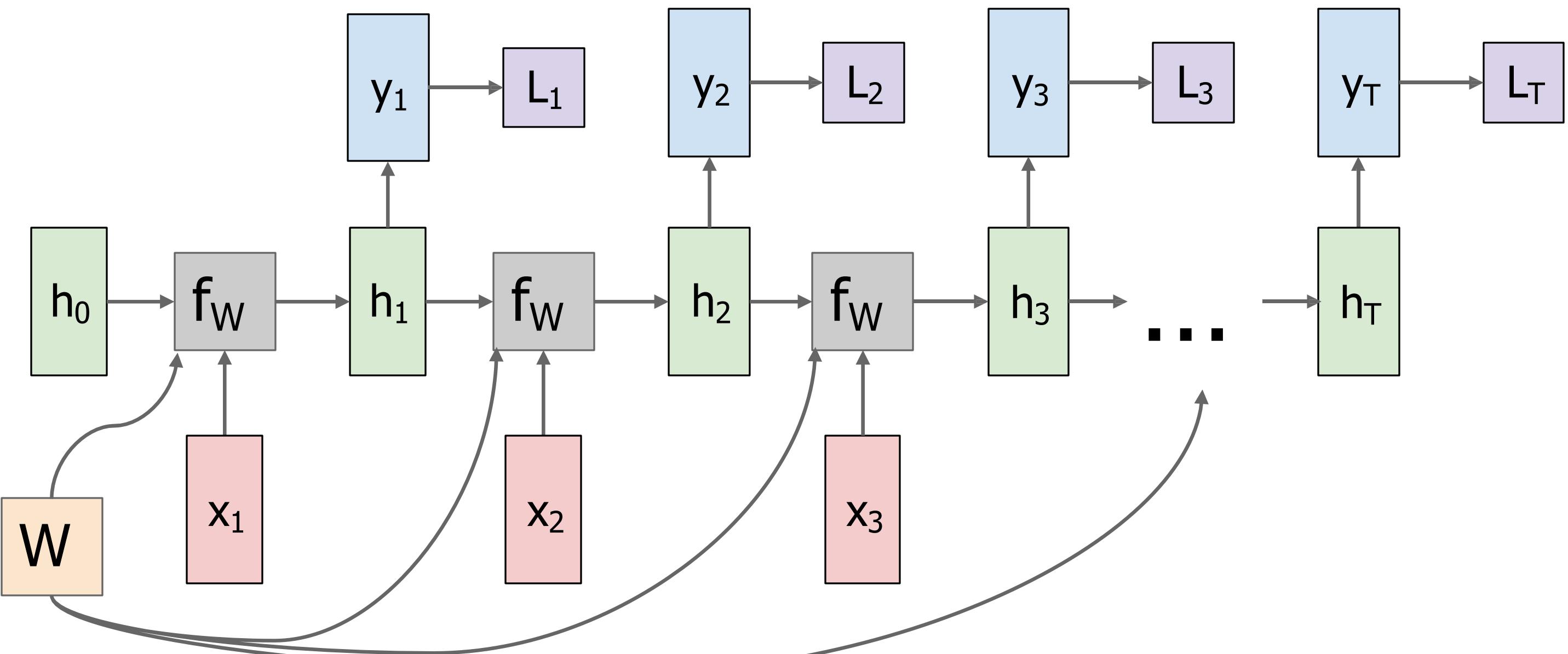
Re-use the same weight matrix at every time-step



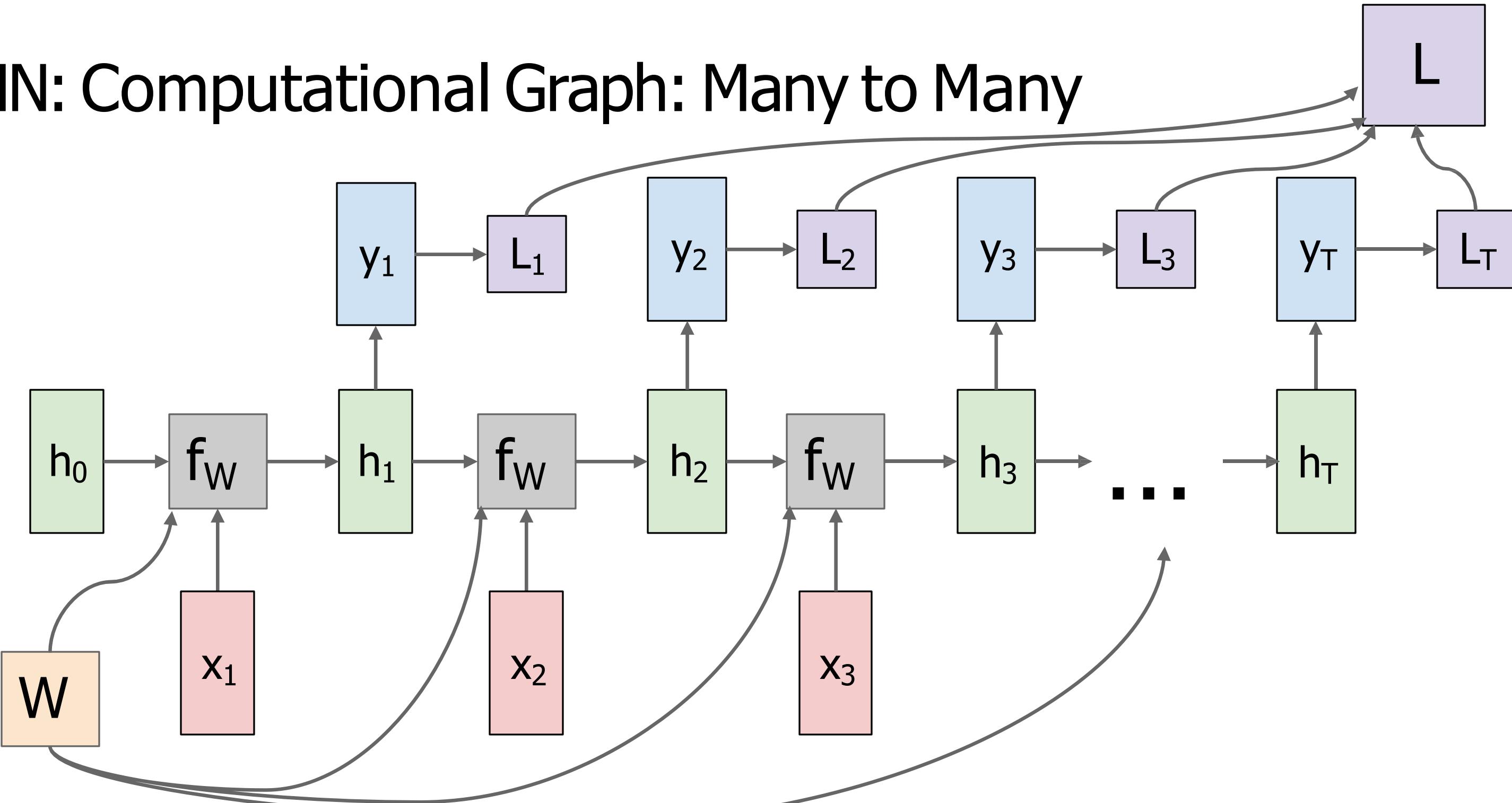
RNN: Computational Graph: Many to Many



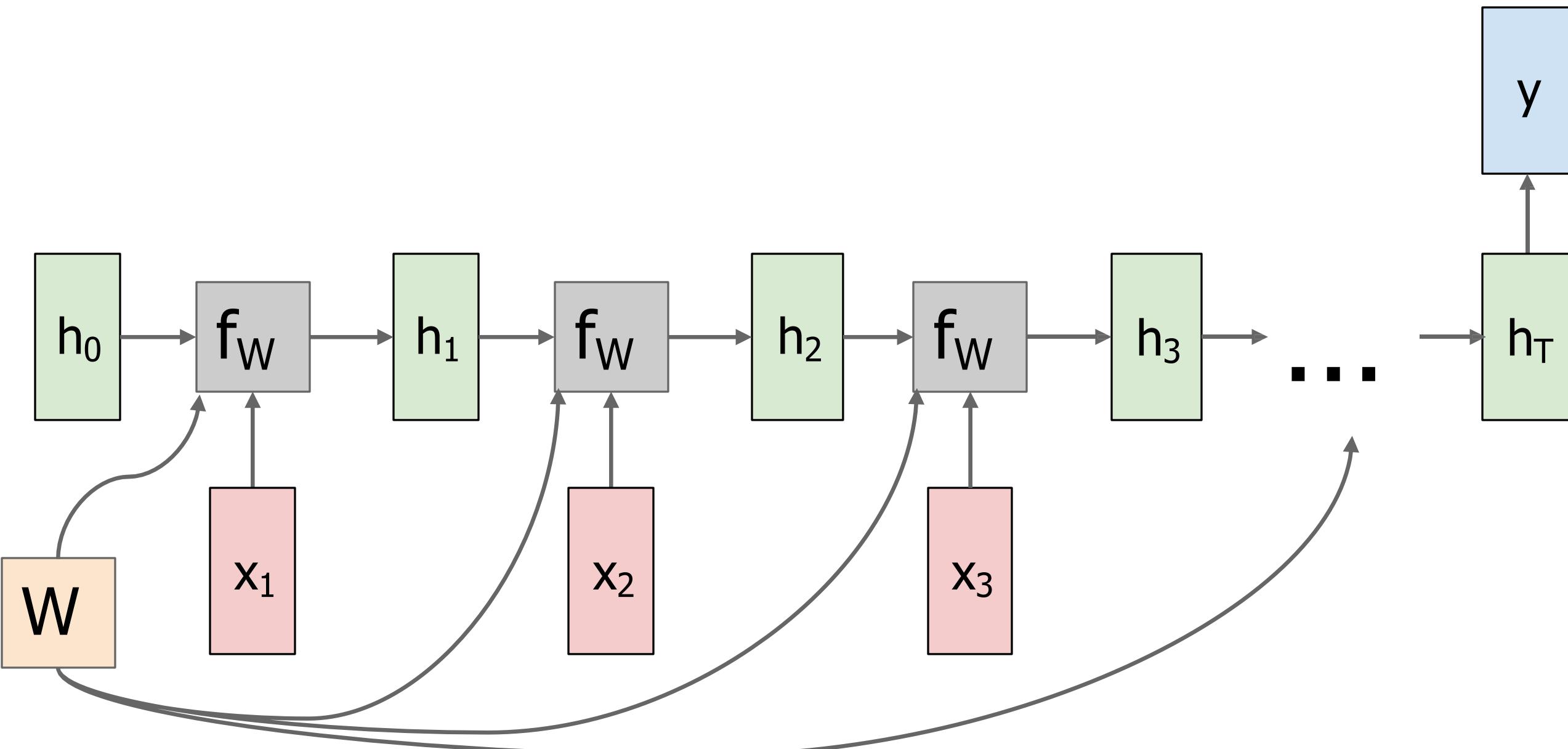
RNN: Computational Graph: Many to Many



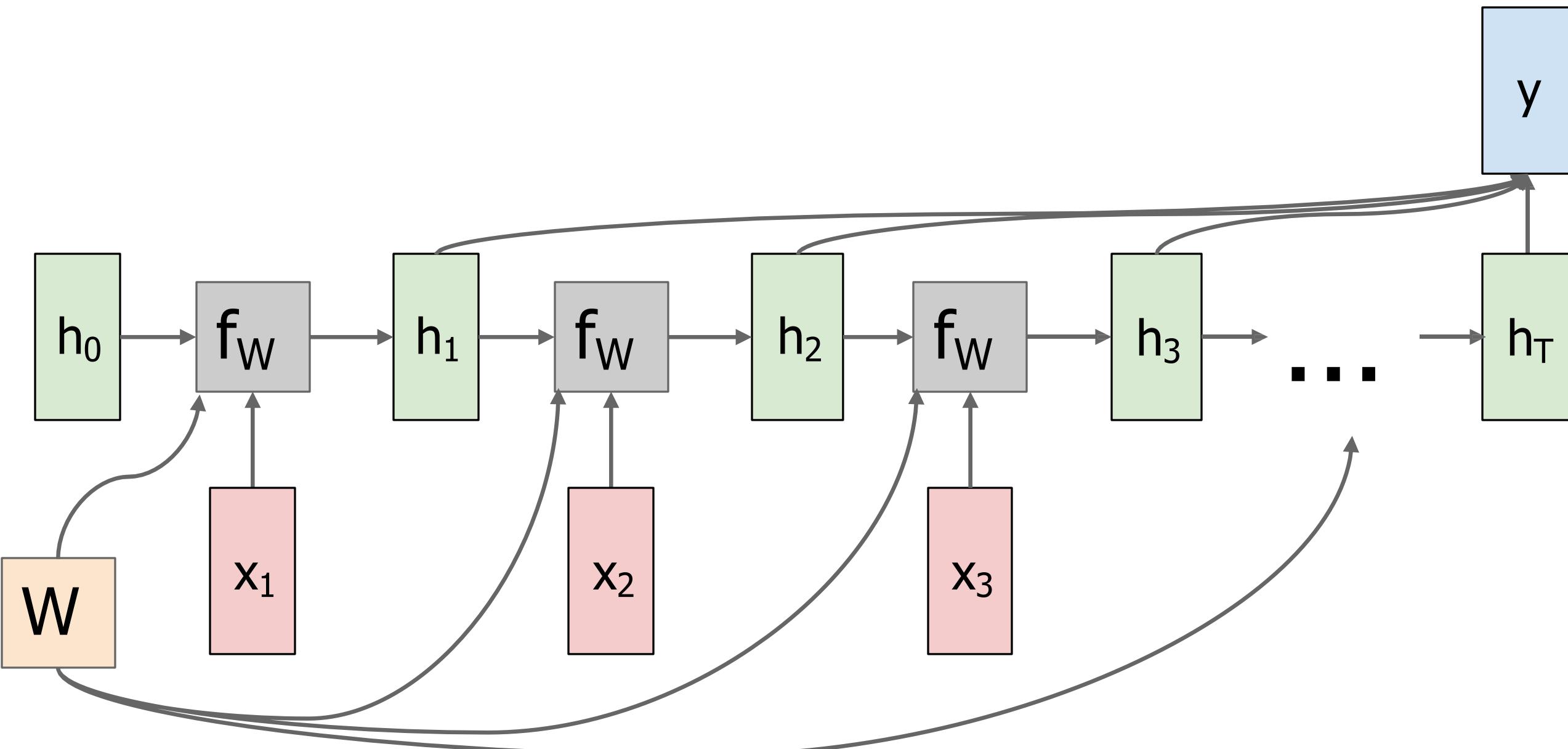
RNN: Computational Graph: Many to Many



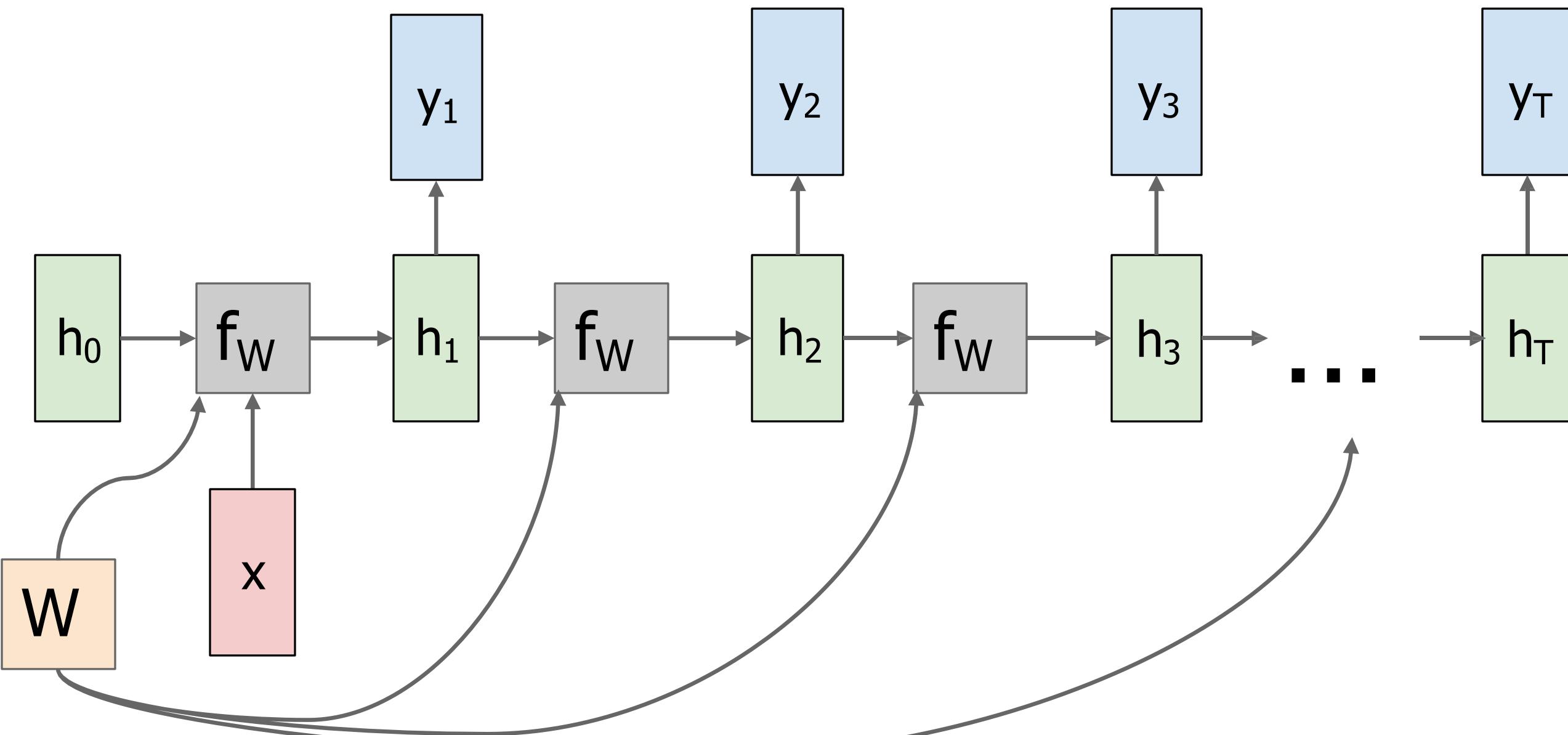
RNN: Computational Graph: Many to One



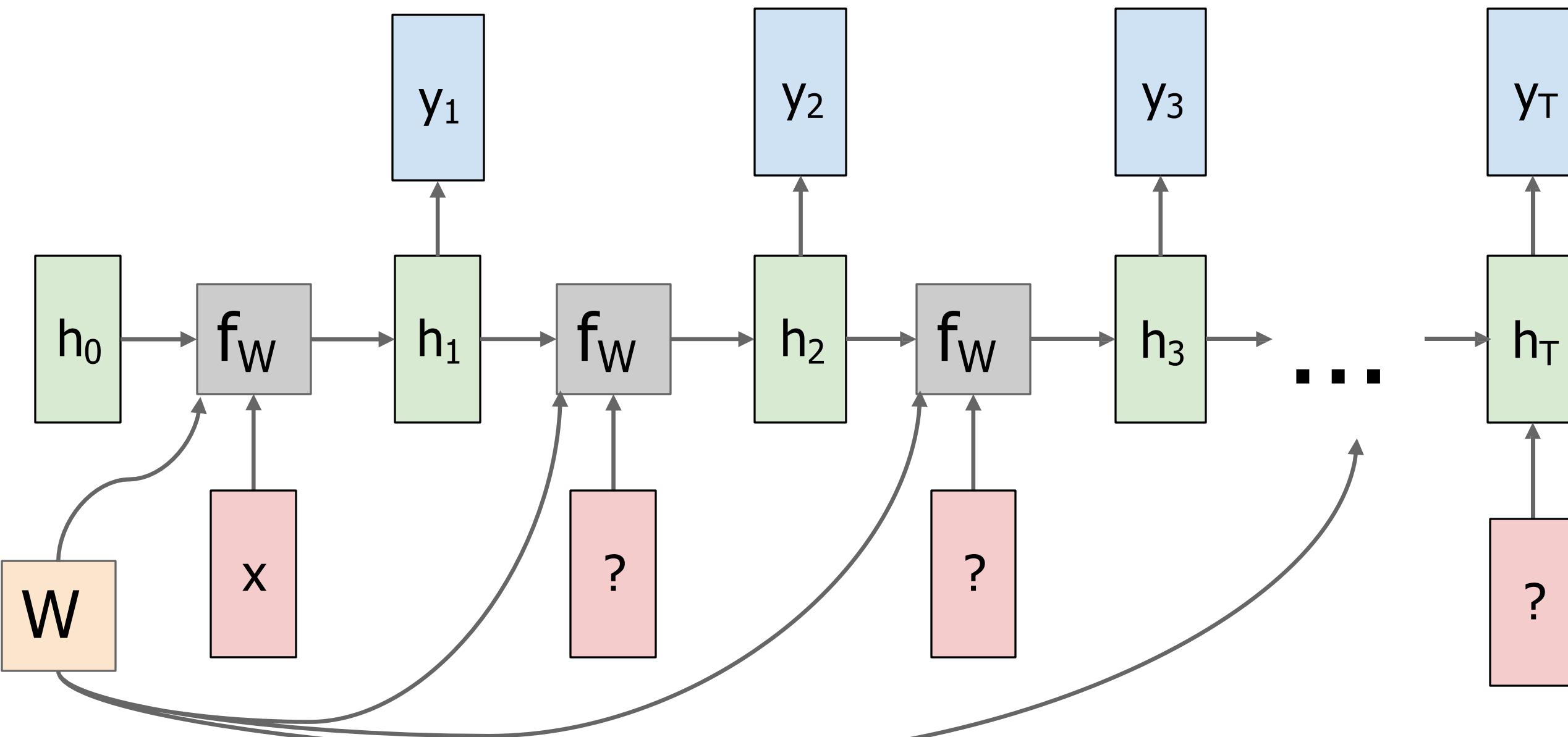
RNN: Computational Graph: Many to One



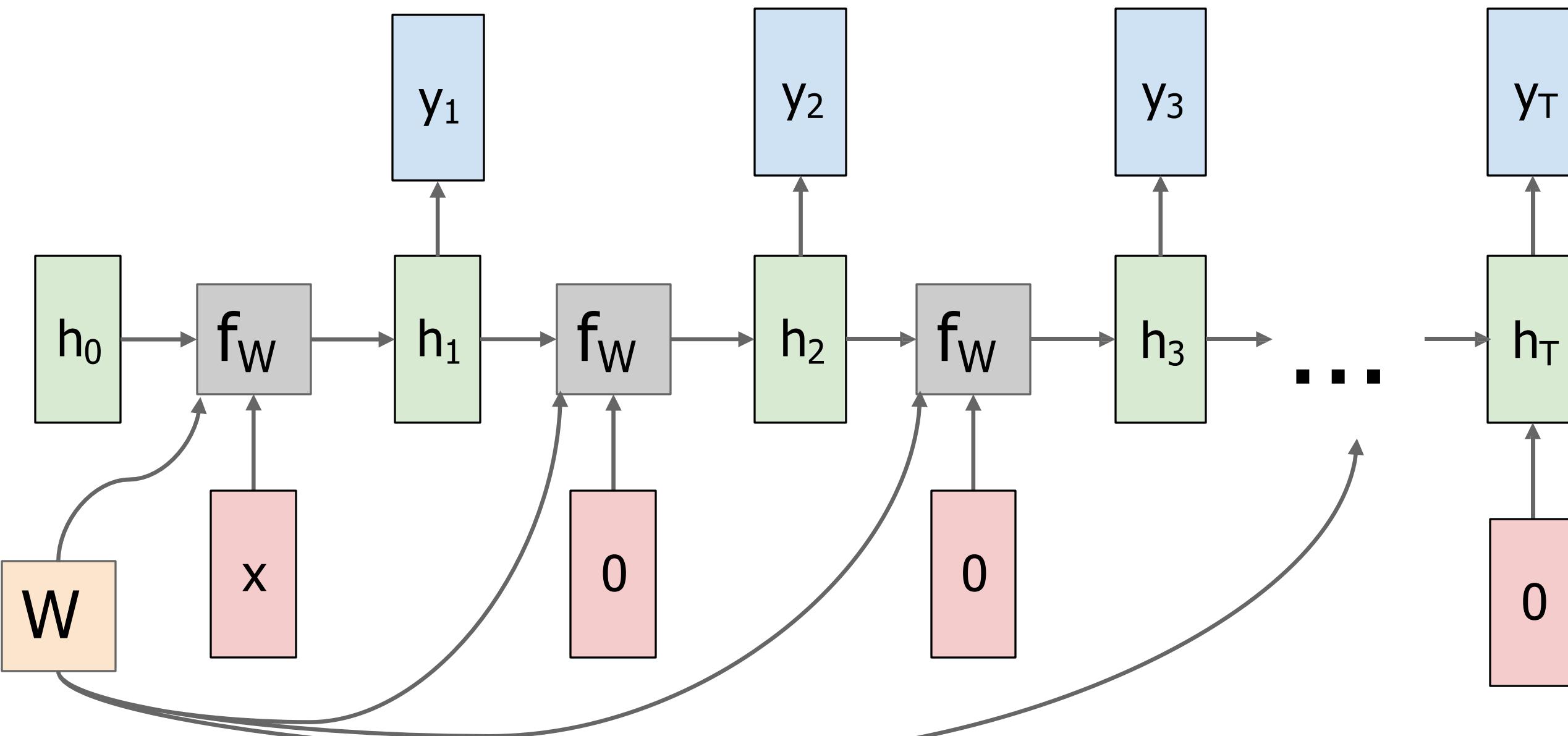
RNN: Computational Graph: One to Many



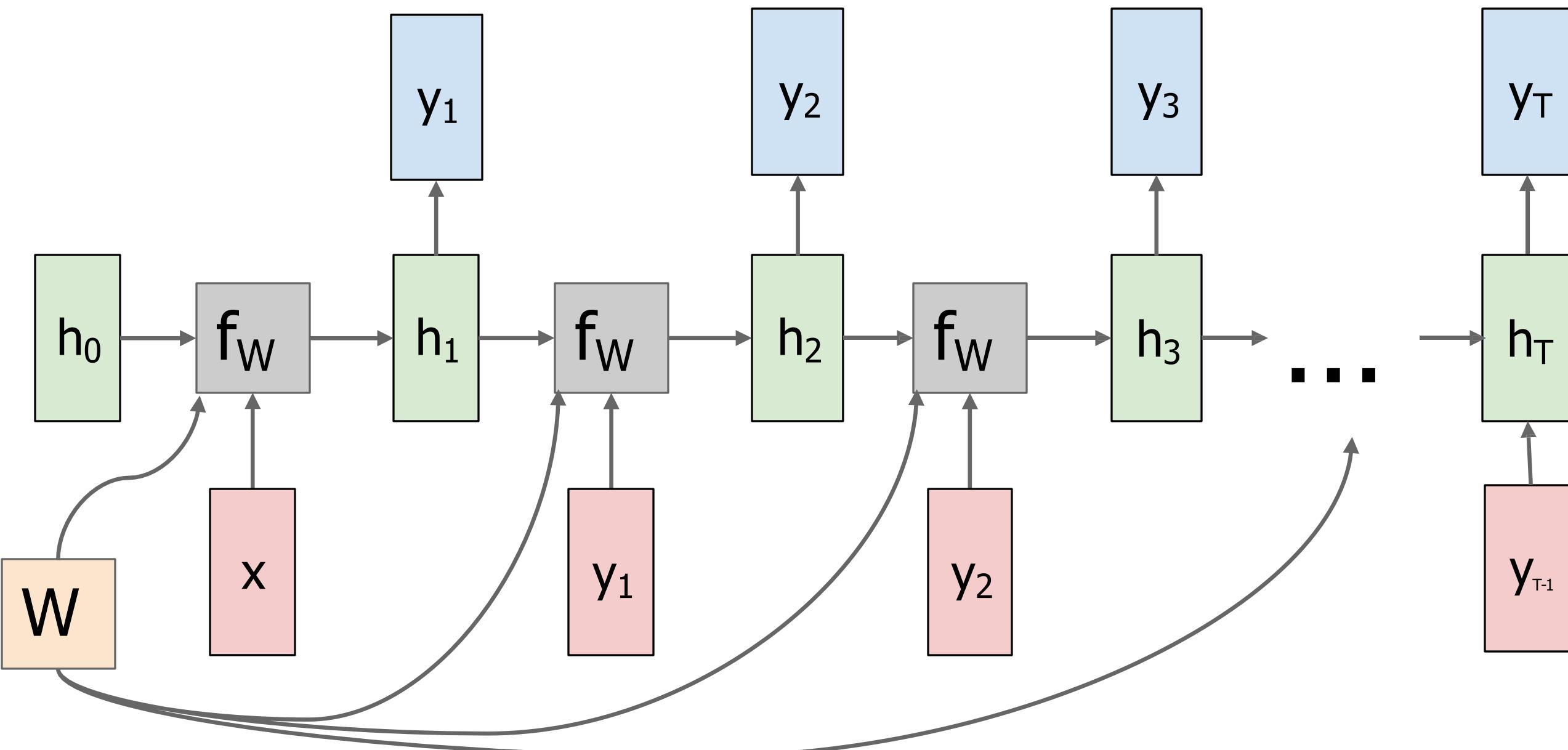
RNN: Computational Graph: One to Many



RNN: Computational Graph: One to Many

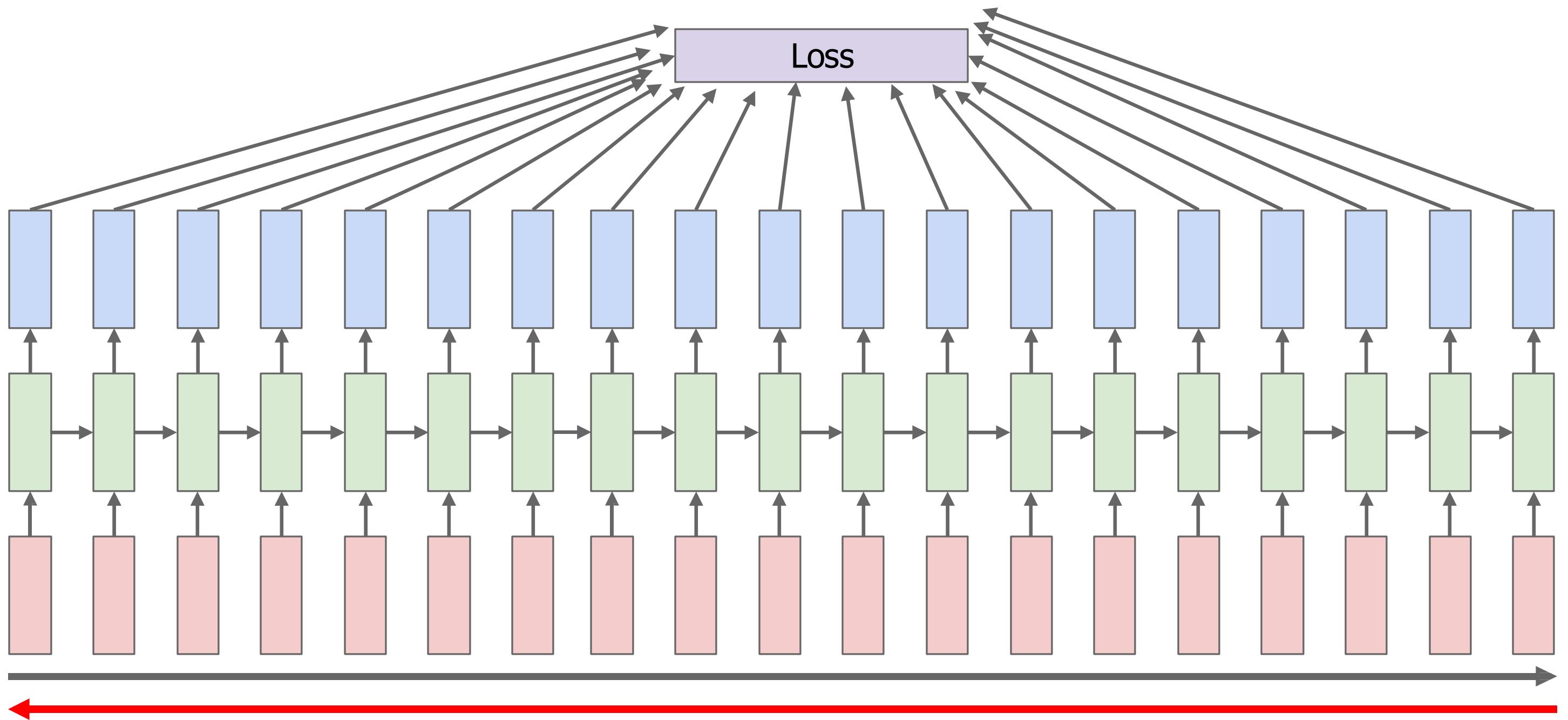


RNN: Computational Graph: One to Many

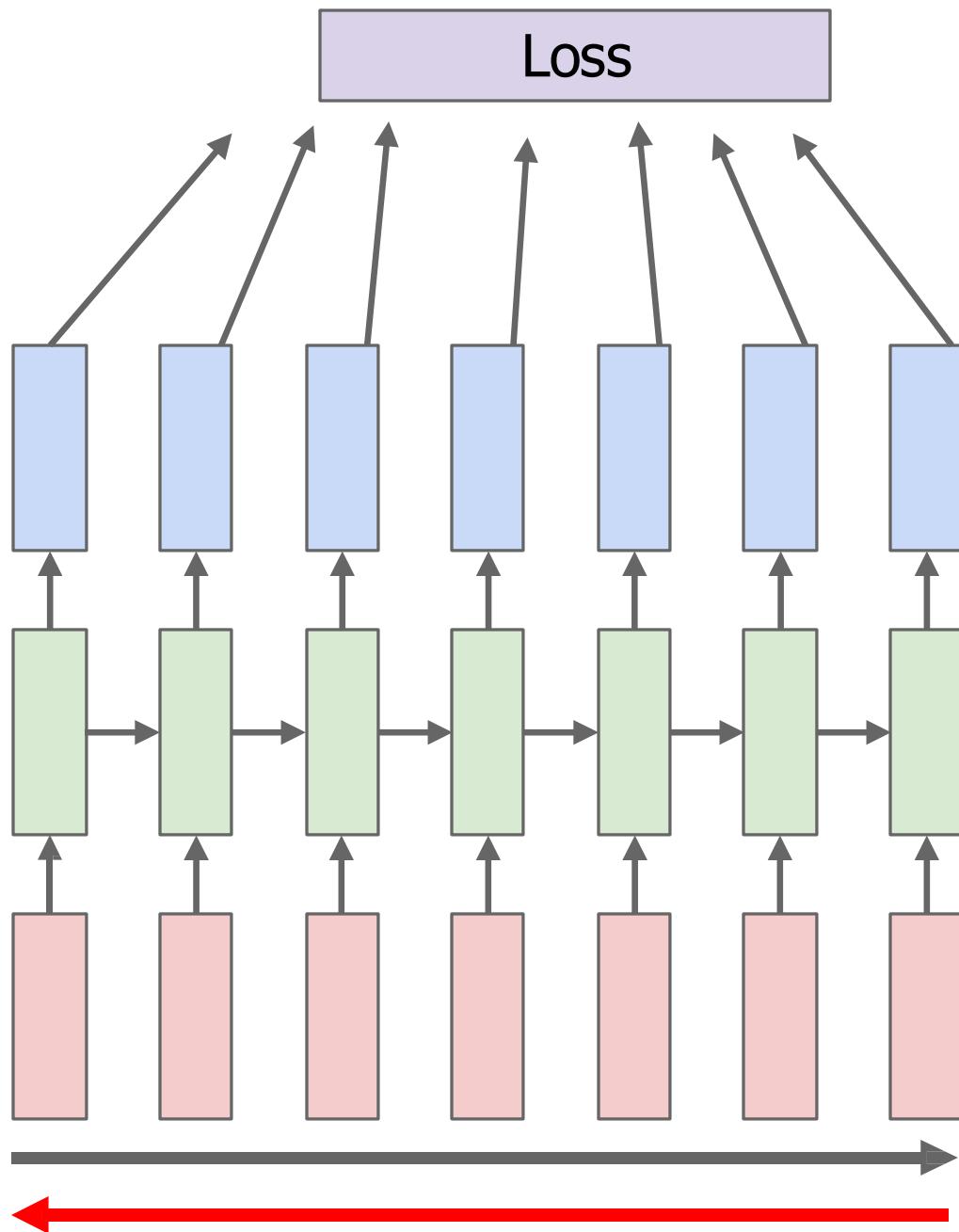


Backpropagation through time

Forward through entire sequence to compute loss, then backward through entire sequence to compute gradient

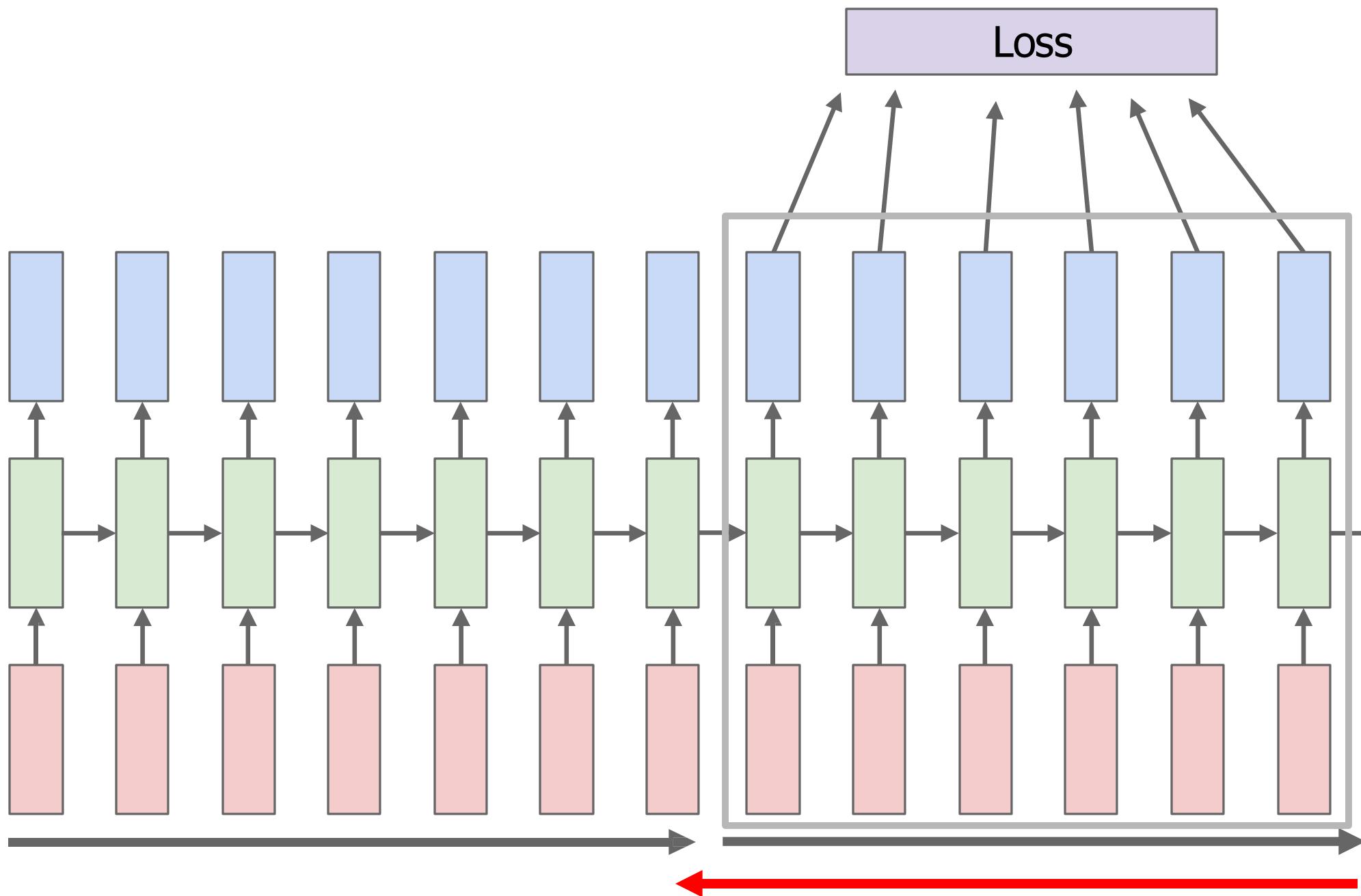


Truncated Backpropagation through time



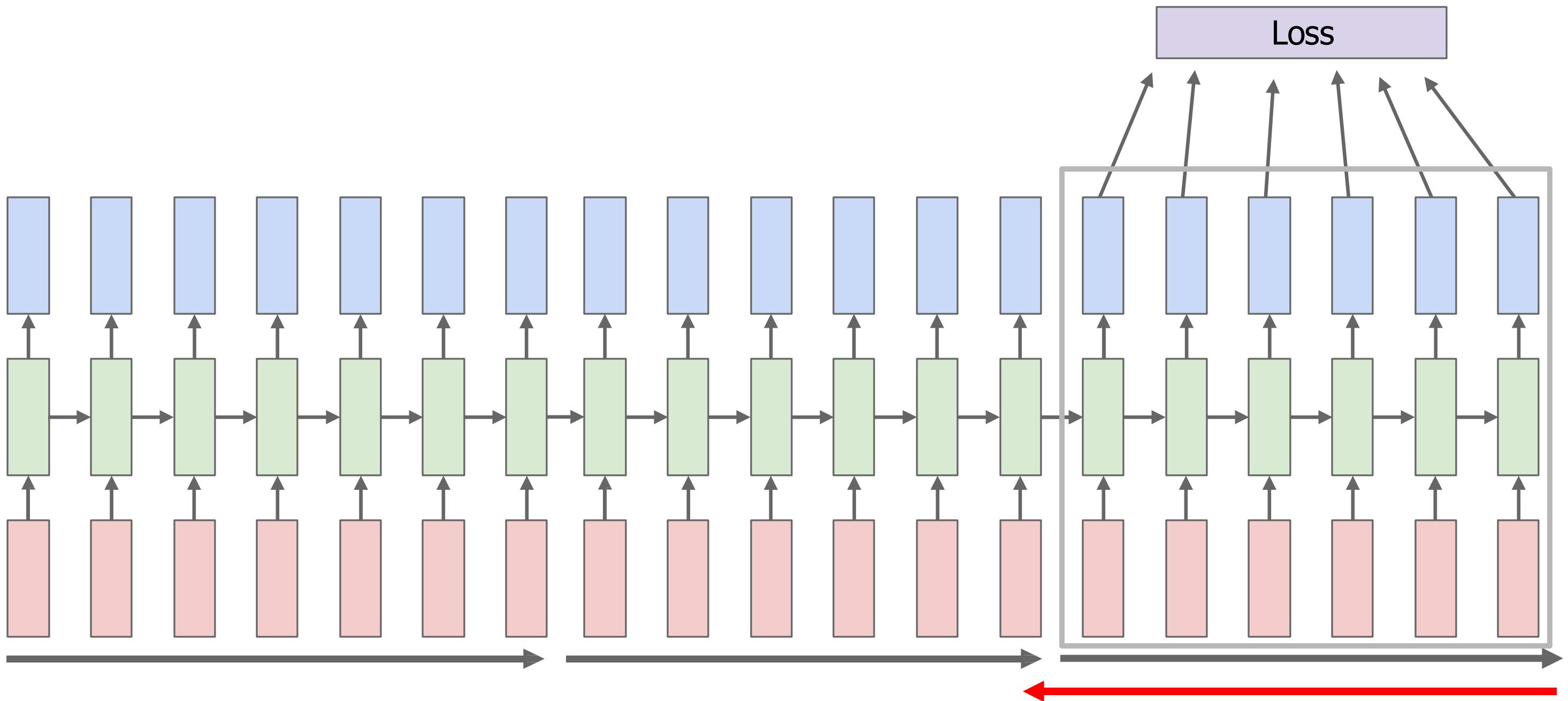
Run forward and backward
through chunks of the
sequence instead of whole
sequence

Truncated Backpropagation through time



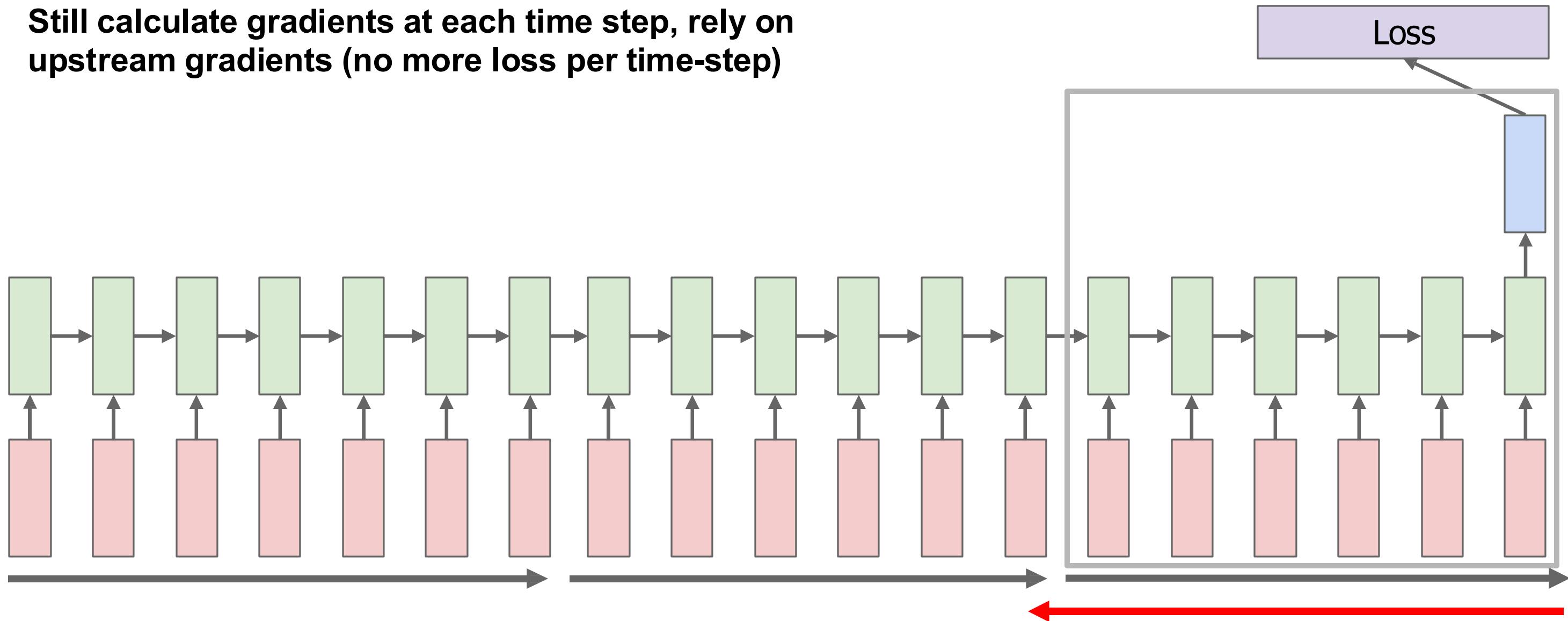
Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps

Truncated Backpropagation through time



Truncated BPTT: Single Output

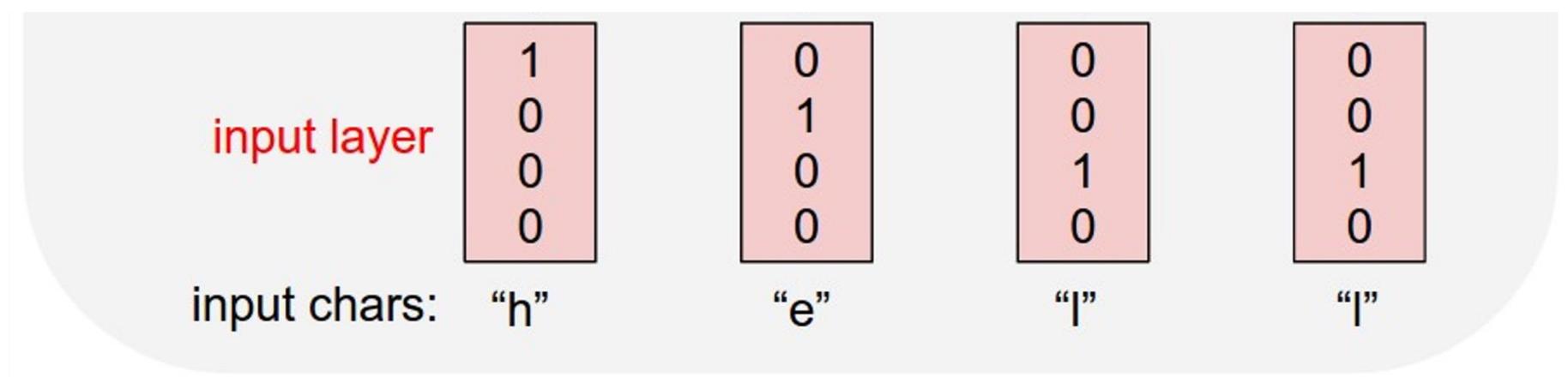
Still calculate gradients at each time step, rely on upstream gradients (no more loss per time-step)



A more practical example: Character-level **Language Model**

Vocabulary:
[h,e,l,o]

Example training sequence:
“hello”

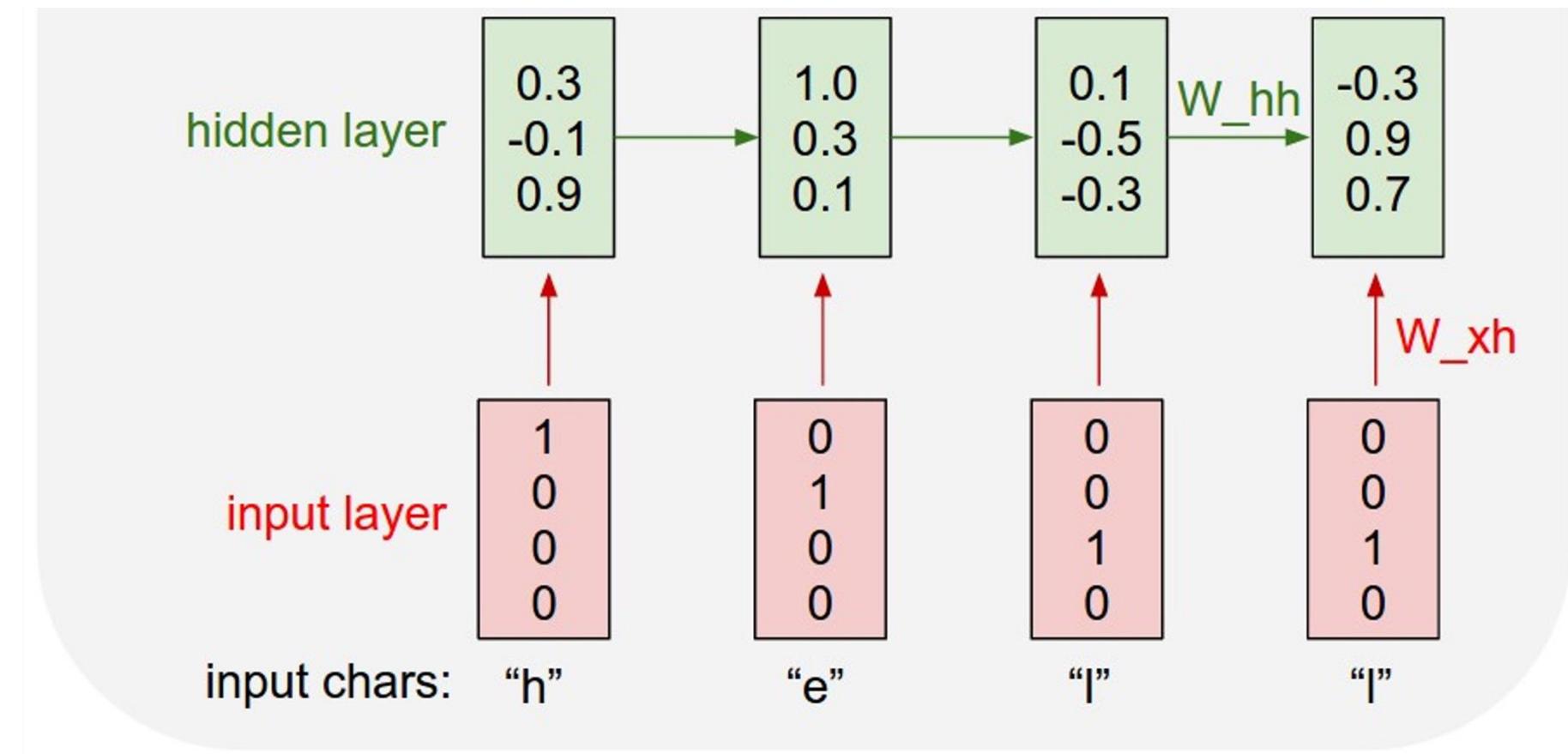


A more practical example:
Character-level
Language Model

Vocabulary:
[h,e,l,o]

Example training sequence:
“hello”

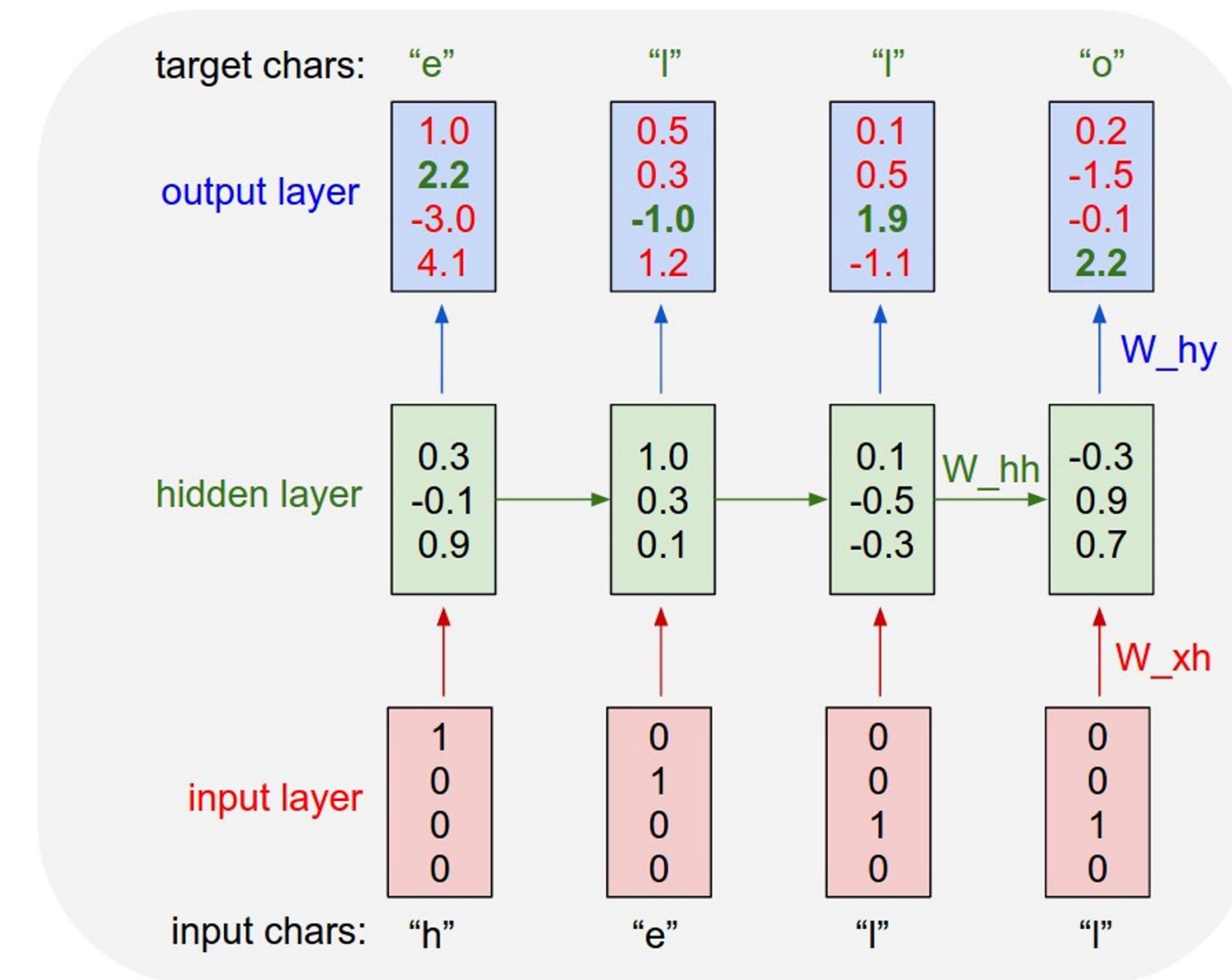
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$



A more practical example: Character-level **Language Model**

Vocabulary:
[h,e,l,o]

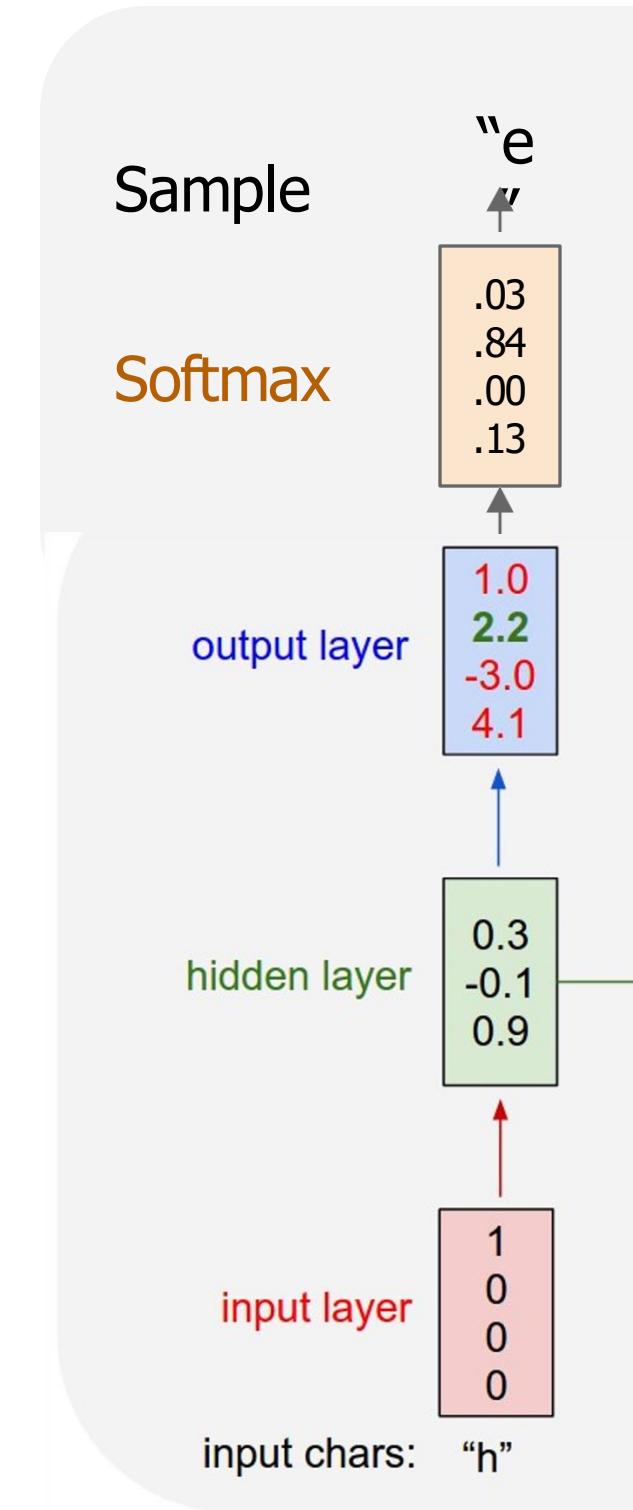
Example training
sequence:
“hello”



Example: Character-level Language Model **Sampling**

Vocabulary:
[h,e,l,o]

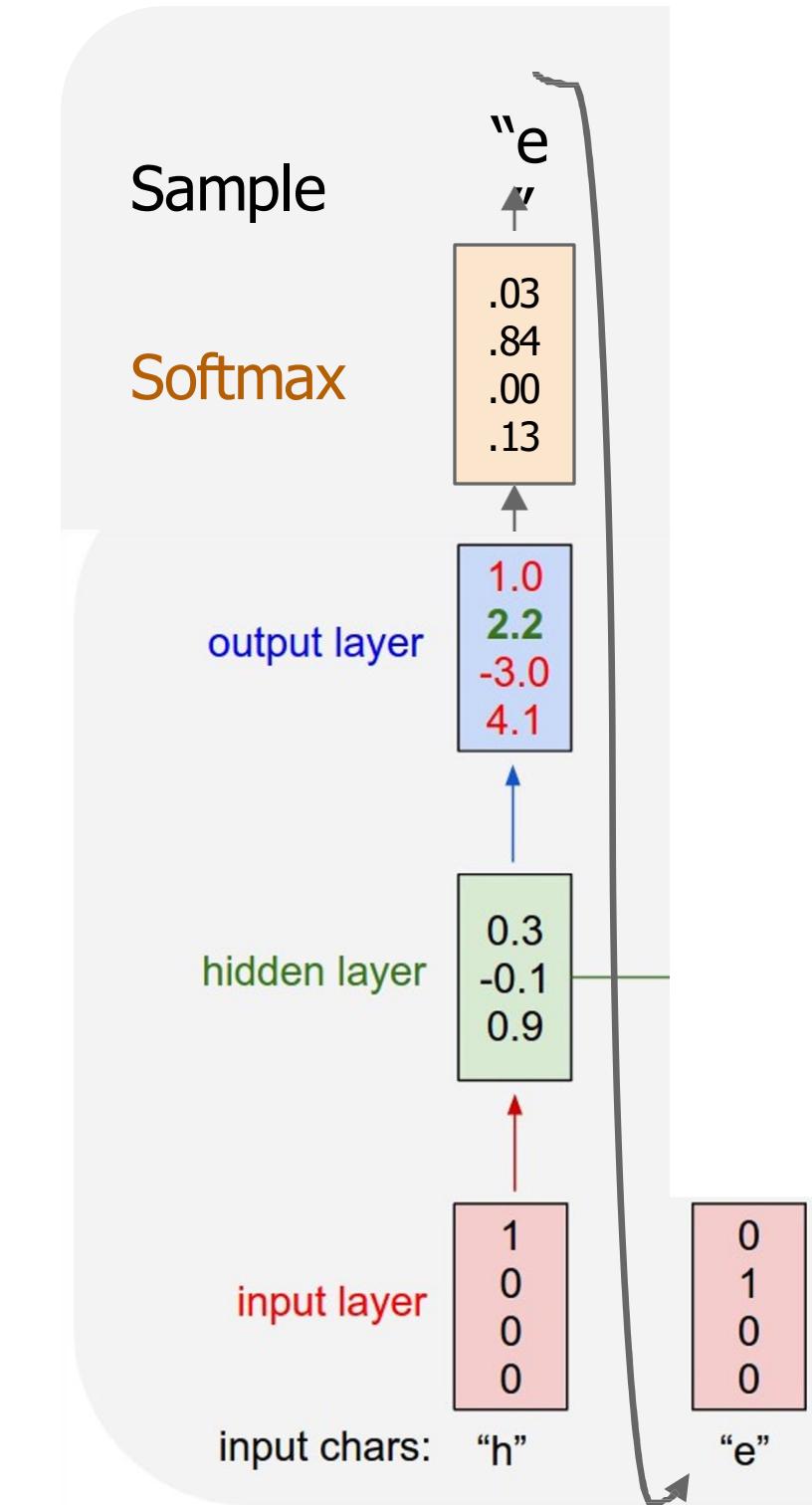
At test-time sample characters
one at a time, feed back to
model



Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

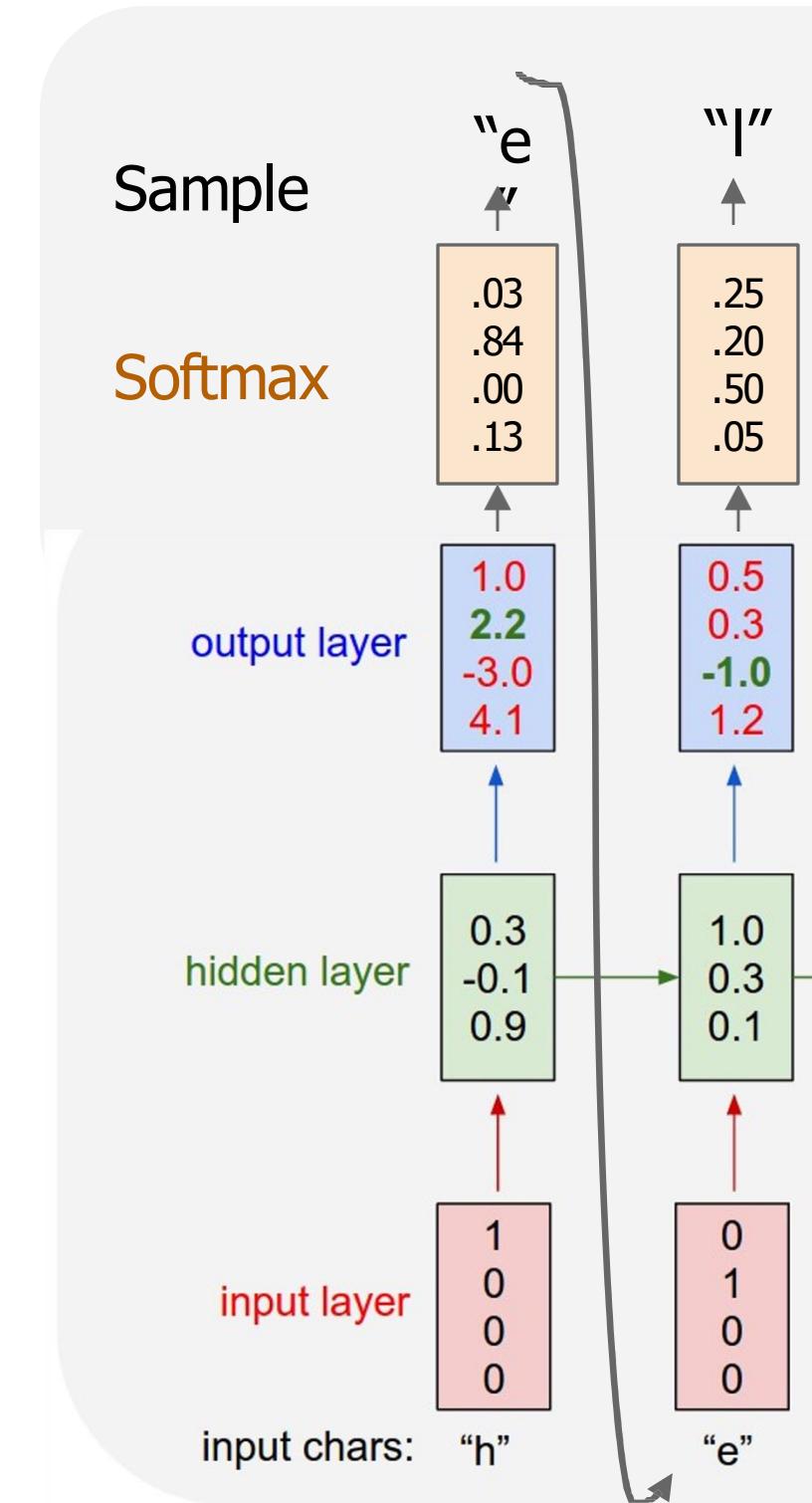
At test-time sample characters one at a time, feed back to model



Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

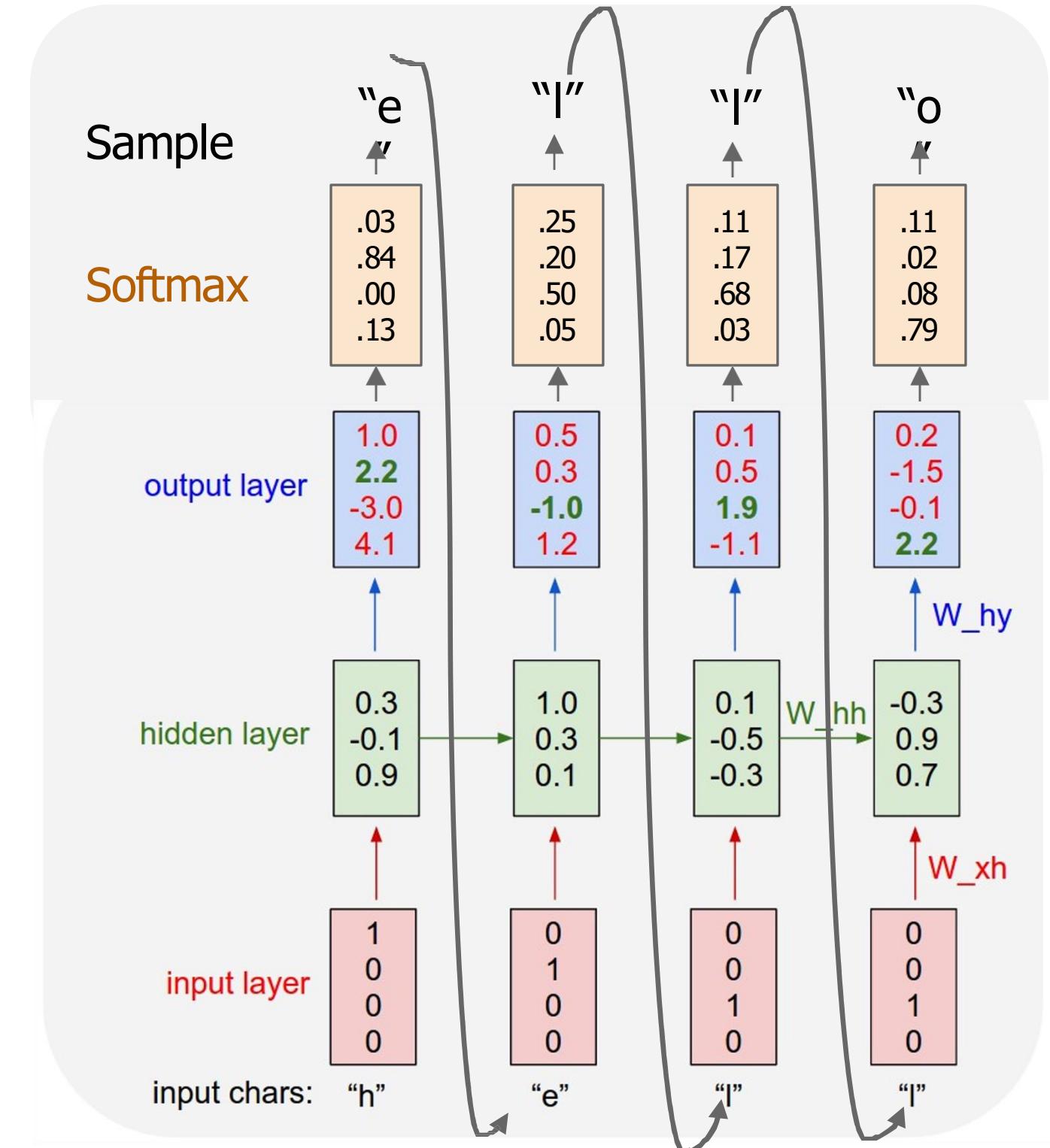
At test-time sample characters one at a time, feed back to model



Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

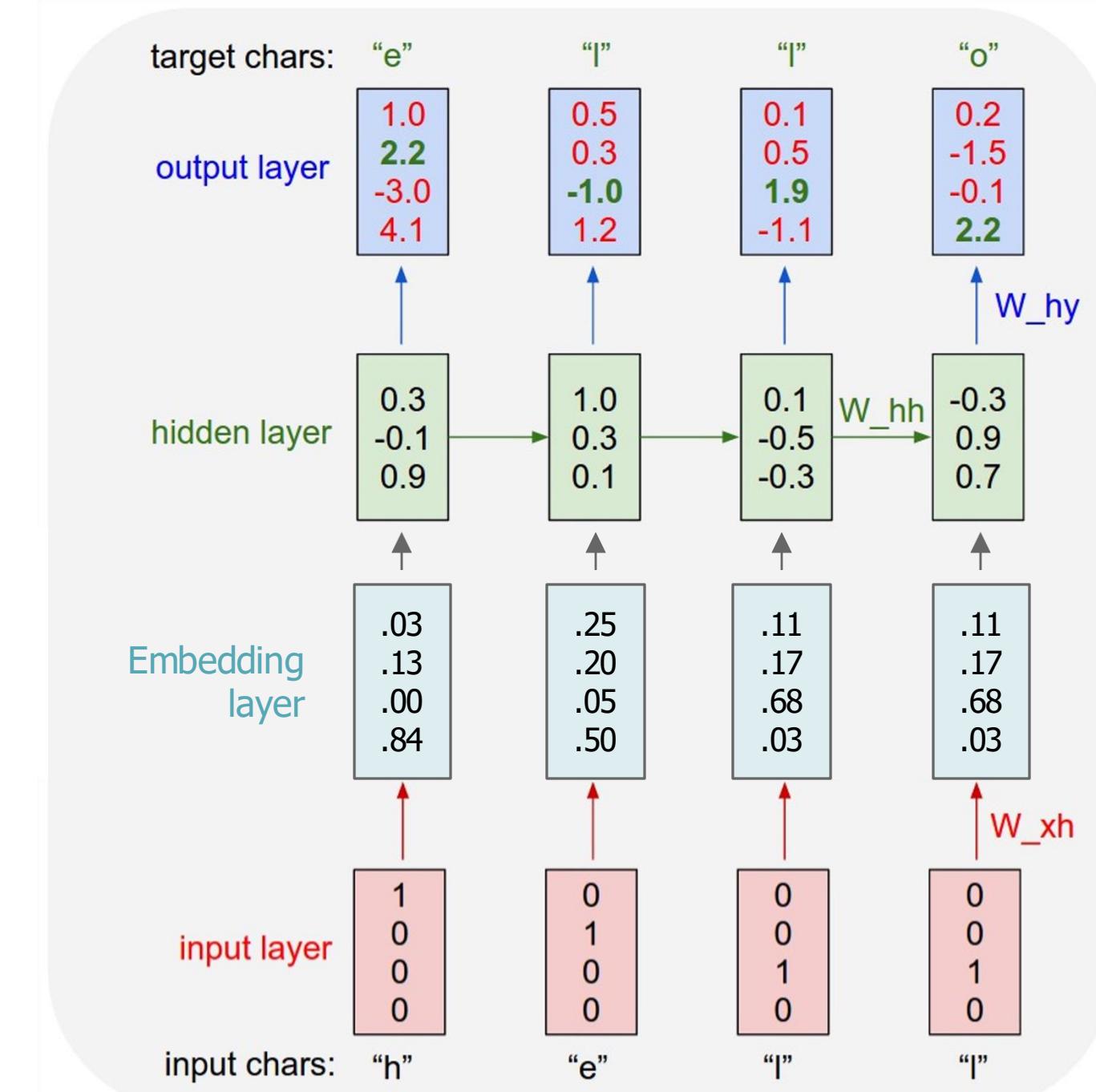
At test-time sample characters one at a time, feed back to model



Example: Character-level Language Model Sampling

$$\begin{aligned}
 [W_{11} & W_{12} & W_{13} & W_{14}] [1] &= [W_{11}] \\
 [W_{21} & W_{22} & W_{23} & W_{14}] [0] &= [W_{21}] \\
 [W_{31} & W_{32} & W_{33} & W_{14}] [0] &= [W_{31}] \\
 [W_{41} & W_{42} & W_{43} & W_{44}] [0] &= [W_{41}]
 \end{aligned}$$

Matrix multiplication with a one-hot vector just extracts a column from the weight matrix. We often put a separate embedding layer between the input and hidden layers.



min-char-rnn.py gist: 112 lines of Python

```
"""
Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
BSD License
"""

import numpy as np

# data I/O
data = open('input.txt', 'r').read() # should be simple plain text file
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print 'data has %d characters, %d unique.' % (data_size, vocab_size)
char_to_ix = { ch:i for i,ch in enumerate(chars) }
ix_to_char = { i:ch for i,ch in enumerate(chars) }

# hyperparameters
hidden_size = 100 # size of hidden layer of neurons
seq_length = 25 # number of steps to unroll the RNN for
learning_rate = 1e-1

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
bh = np.zeros((hidden_size, 1)) # hidden bias
by = np.zeros((vocab_size, 1)) # output bias

def lossFun(inputs, targets, hprev):
    """
    inputs,targets are both list of integers.
    hprev is Hx1 array of initial hidden state
    returns the loss, gradients on model parameters, and last hidden state
    """
    xs, hs, ys, ps = {}, {}, {}, {}
    hs[-1] = np.copy(hprev)
    loss = 0
    # forward pass
    for t in xrange(len(inputs)):
        xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
        xs[t][inputs[t]] = 1
        hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
        ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
        ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
        loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
    # backward pass: compute gradients going backwards
    dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
    dbh, dby = np.zeros_like(bh), np.zeros_like(by)
    dhnext = np.zeros_like(hs[0])
    for t in reversed(xrange(len(inputs))):
        dy = np.copy(ps[t])
        dy[targets[t]] -= 1 # backprop into y
        dWhy += np.dot(dy, hs[t].T)
        dby += dy
        dh = np.dot(Why.T, dy) + dhnext # backprop into h
        ddraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
        dbh += ddraw
        dWxh += np.dot(ddraw, xs[t].T)
        dWhh += np.dot(ddraw, hs[t-1].T)
        dhnext = np.dot(Whh.T, ddraw)
    for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
        np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
    return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]

def sample(h, seed_ix, n):
    """
    sample a sequence of integers from the model
    h is memory state, seed_ix is seed letter for first time step
    """
    x = np.zeros((vocab_size, 1))
    x[seed_ix] = 1
    ixes = []
    for t in xrange(n):
        h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
        y = np.dot(Why, h) + by
        p = np.exp(y) / np.sum(np.exp(y))
        ix = np.random.choice(range(vocab_size), p=p.ravel())
        x[ix] = 1
        ixes.append(ix)
    return ixes

n, p = 0, 0
mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
while True:
    # prepare inputs (we're sweeping from left to right in steps seq_length long)
    if p+seq_length+1 >= len(data) or n == 0:
        hprev = np.zeros((hidden_size,1)) # reset RNN memory
        p = 0 # go from start of data
    inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
    targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]

    # sample from the model now and then
    if n % 100 == 0:
        sample_ix = sample(hprev, inputs[0], 200)
        txt = ''.join(ix_to_char[ix] for ix in sample_ix)
        print '----\n%s\n----' % (txt,)

    # forward seq_length characters through the net and fetch gradient
    loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
    smooth_loss = smooth_loss * 0.999 + loss * 0.001
    if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress

    # perform parameter update with Adagrad
    for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
                                   [dWxh, dWhh, dWhy, dbh, dby],
                                   [mWxh, mWhh, mWhy, mbh, mby]):
        mem += dparam * dparam
        param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update

    p += seq_length # move data pointer
    n += 1 # iteration counter
```

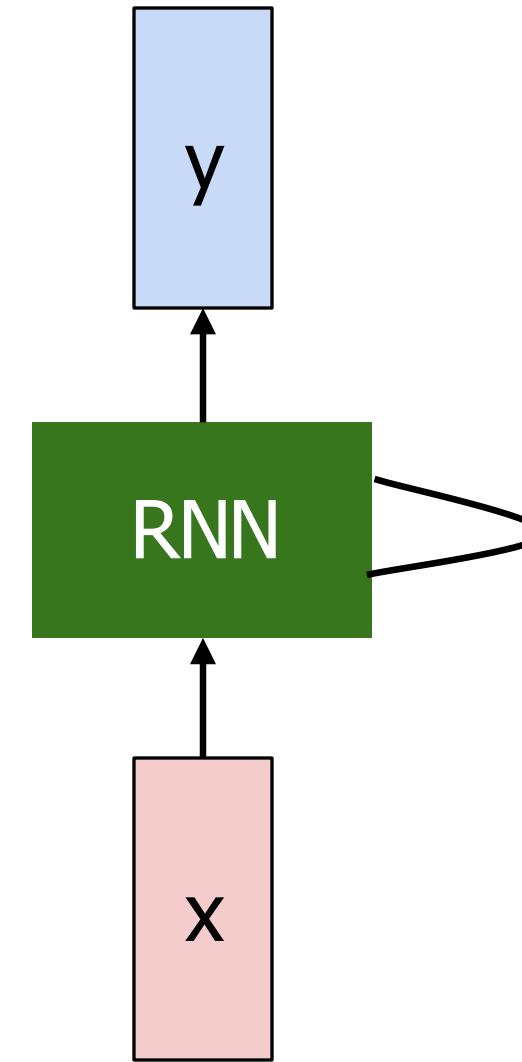
(<https://gist.github.com/karpathy/d4dee66867f8291f086>)

THE SONNETS

by William Shakespeare

From fairest creatures we desire increase,
That thereby beauty's rose might never die,
But as the riper should by time decease,
His tender heir might bear his memory:
But thou, contracted to thine own bright eyes,
Feed'st thy light's flame with self-substantial fuel,
Making a famine where abundance lies,
Thyself thy foe, to thy sweet self too cruel:
Thou that art now the world's fresh ornament,
And only herald to the gaudy spring,
Within thine own bud buriest thy content,
And tender churl mak'st waste in niggarding:
Pity the world, or else this glutton be,
To eat the world's due, by the grave and thee.

When forty winters shall besiege thy brow,
And dig deep trenches in thy beauty's field,
Thy youth's proud livery so gazed on now,
Will be a tatter'd weed of small worth held:
Then being asked, where all thy beauty lies,
Where all the treasure of thy lusty days;
To say, within thine own deep sunken eyes,
Were an all-eating shame, and thriftless praise.
How much more praise deserv'd thy beauty's use,
If thou couldst answer 'This fair child of mine
Shall sum my count, and make my old excuse,'
Proving his beauty by succession thine!
This were to be new made when thou art old,
And see thy blood warm when thou feel'st it cold.



[Blogpost from Andrey Karpathy back in 2015!](#)

at first:

tyntd-iafhatawiaoahrdemot lytdws e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e
plia tklrgd t o idoe ns,smtt h ne etie h,hregtrs nigtike,aoaenns lng

train more

"Tmont thithey" fomesscerliund
Keushey. Thom here
sheulke, anmerenith ol sivh I lalterthend Bleipile shuwyl on aseterlome
coaniogennc Phe lism thond hon at. MeiDimorotion in ther thize."

train more

Aftair fall unsuch that the hall for Prince Velzonski's that me of
her hearly, and behs to so arwage fiving were to it beloge, pavu say falling misfort
how, and Gogition is so overelical and ofter.

train more

"Why do what that day," replied Natasha, and wishing to himself the fact the
princess, Princess Mary was easier, fed in had oftened him.
Pierre aking his soul came to the packs and drove up his father-in-law women.

PANDARUS:

Alas, I think he shall be come approached and the day
When little strain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:

They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:

Well, your wit is in the care of side and that.

Second Lord:

They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:

Come, sir, I will make did behold your worship.

VIOLA:

I'll drink it.

VIOLA:

Why, Salisbury must find his flesh and thought
That which I am not aps, not a man and in fire,
To show the reining of the raven and the wars
To grace my hand reproach within, and not a fair are hand,
That Caesar and my goodly father's world;
When I was heaven of presence and our fleets,
We spare with hours, but cut thy council I am great,
Murdered and by thy master's ready there
My power to give thee but so much as hell:
Some service in the noble bondman here,
Would show him to her wine.

KING LEAR:

O, if you were a feeble sight, the courtesy of your law,
Your sight and several breath, will wear the gods
With his heads, and my hands are wonder'd at the deeds,
So drop upon your lordship's head, and your opinion
Shall be against your honour.

This repository Search

Explore Gist Blog Help

 karpathy + - ⌂ ⚙ ⌁

 torvalds / linux

Watch 3,711 Star 23,054 Fork 9,141

Linux kernel source tree

520,037 commits 1 branch 420 releases 5,039 contributors

 branch: master / [linux](#) / +

Merge branch 'drm-fixes' of git://people.freedesktop.org/~airlied/linux ...

 torvalds authored 9 hours ago latest commit 4b1706927d

Category	Commit Message	Date
Documentation	Merge git://git.kernel.org/pub/scm/linux/kernel/git/nab/target-pending	6 days ago
arch	Merge branch 'x86-urgent-for-linus' of git://git.kernel.org/pub/scm/l...	a day ago
block	block: discard bdi_unregister() in favour of bdi_destroy()	9 days ago
crypto	Merge git://git.kernel.org/pub/scm/linux/kernel/git/herbert/crypto-2.6	10 days ago
drivers	Merge branch 'drm-fixes' of git://people.freedesktop.org/~airlied/linux	9 hours ago
firmware	firmware/ihex2fw.c: restore missing default in switch statement	2 months ago
fs	vfs: read file_handle only once in handle_to_path	4 days ago
include	Merge branch 'perf-urgent-for-linus' of git://git.kernel.org/pub/scm/...	a day ago
init	init: fix regression by supporting devices with major:minor:offset fo...	a month ago
ipc	Merge branch 'perf-urgent-for-linus' of git://git.kernel.org/pub/scm/...	a month ago

 Code

74 Pull requests

 Pulse

 Graphs

HTTPS clone URL

<https://github.com/torvalds/linux> 

You can clone with [HTTPS](#), [SSH](#), or [Subversion](#). 

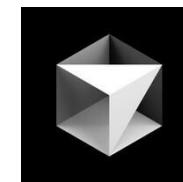
 Clone in Desktop

 Download ZIP

```
static void do_command(struct seq_file *m, void *v)
{
    int column = 32 << (cmd[2] & 0x80);
    if (state)
        cmd = (int)(int_state ^ (in_8(&ch->ch_flags) & Cmd) ? 2 : 1);
    else
        seq = 1;
    for (i = 0; i < 16; i++) {
        if (k & (1 << i))
            pipe = (in_use & UMXTHREAD_UNCCA) +
                ((count & 0x00000000fffffff8) & 0x000000f) << 8;
        if (count == 0)
            sub(pid, ppc_md.kexec_handle, 0x20000000);
        pipe_set_bytes(i, 0);
    }
    /* Free our user pages pointer to place camera if all dash */
    subsystem_info = &of_changes[PAGE_SIZE];
    rek_controls(offset, idx, &soffset);
    /* Now we want to deliberately put it to device */
    control_check_polarity(&context, val, 0);
    for (i = 0; i < COUNTER; i++)
        seq_puts(s, "policy ");
}
```

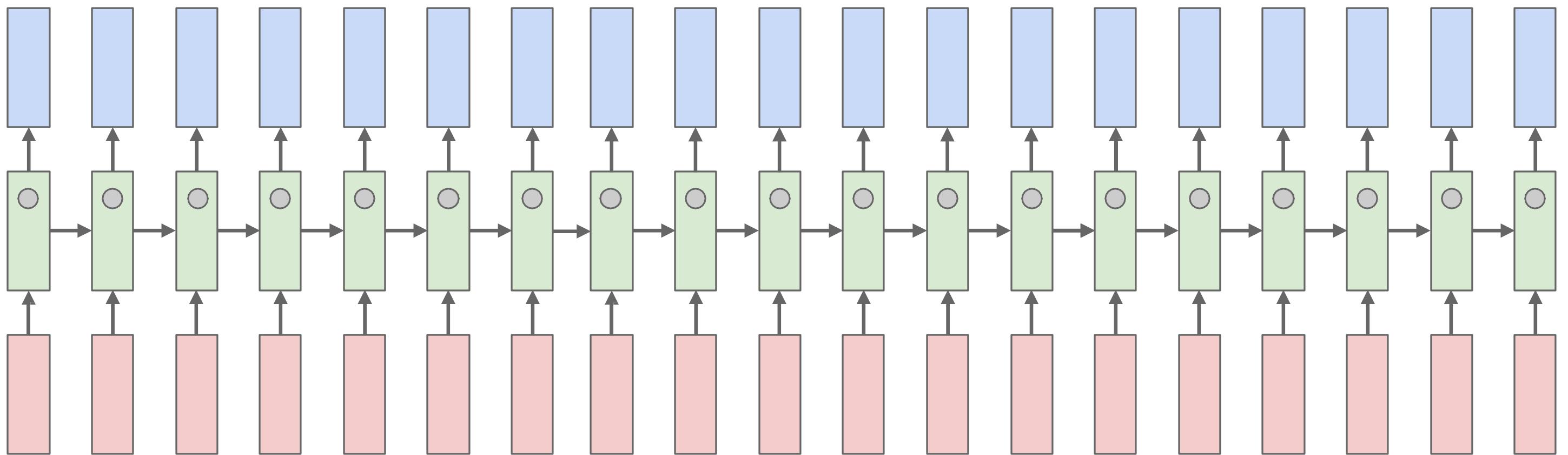
Generated C code

OpenAI Codex, GitHub Copilot, Cursor IDE



<https://openai.com/blog/openai-codex/>

Searching for interpretable cells



Searching for interpretable cells

```
/* Unpack a filter field's string representation from user-space
 * buffer. */
char *audit_unpack_string(void **bufp, size_t *remain, size_t len)
{
    char *str;
    if (!*bufp || (len == 0) || (len > *remain))
        return ERR_PTR(-EINVAL);
    /* Of the currently implemented string fields, PATH_MAX
     * defines the longest valid length.
    */
```

Searching for interpretable cells

"You mean to imply that I have nothing to eat out of.... On the contrary, I can supply you with everything even if you want to give dinner parties," warmly replied Chichagov, who tried by every word he spoke to prove his own rectitude and therefore imagined Kutuzov to be animated by the same desire.

Kutuzov, shrugging his shoulders, replied with his subtle penetrating smile: "I meant merely to say what I said."

quote detection cell

Karpathy, Johnson, and Fei-Fei: Visualizing and Understanding Recurrent Networks, ICLR Workshop 2016
Figures copyright Karpathy, Johnson, and Fei-Fei, 2015; reproduced with permission

Searching for interpretable cells

Cell sensitive to position in line:

The sole importance of the crossing of the Berezina lies in the fact that it plainly and indubitably proved the fallacy of all the plans for cutting off the enemy's retreat and the soundness of the only possible line of action--the one Kutuzov and the general mass of the army demanded--namely, simply to follow the enemy up. The French crowd fled at a continually increasing speed and all its energy was directed to reaching its goal. It fled like a wounded animal and it was impossible to block its path. This was shown not so much by the arrangements it made for crossing as by what took place at the bridges. When the bridges broke down, unarmed soldiers, people from Moscow and women with children who were with the French transport, all--carried on by vis inertiae--pressed forward into boats and into the ice-covered water and did not, surrender.

line length tracking cell

Karpathy, Johnson, and Fei-Fei: Visualizing and Understanding Recurrent Networks, ICLR Workshop 2016
Figures copyright Karpathy, Johnson, and Fei-Fei, 2015; reproduced with permission

Searching for interpretable cells

```
static int __dequeue_signal(struct sigpending *pending, sigset_t *mask,
    siginfo_t *info)
{
    int sig = next_signal(pending, mask);
    if (sig) {
        if (current->notifier) {
            if (sigismember(current->notifier_mask, sig)) {
                if (!(current->notifier)(current->notifier_data)) {
                    clear_thread_flag(TIF_SIGPENDING);
                    return 0;
                }
            }
        }
        collect_signal(sig, pending, info);
    }
    return sig;
}
```

if statement cell

Searching for interpretable cells

Cell that turns on inside comments and quotes:

```
/* Duplicate LSM field information. The lsm_rule is opaque, so
 * re-initialized. */
static inline int audit_dupe_lsm_field(struct audit_field *df,
    struct audit_field *sf)
{
    int ret = 0;
    char *lsm_str;
    /* our own copy of lsm_str */
    lsm_str = kstrdup(sf->lsm_str, GFP_KERNEL);
    if (unlikely(!lsm_str))
        return -ENOMEM;
    df->lsm_str = lsm_str;
    /* our own (refreshed) copy of lsm_rule */
    ret = security_audit_rule_init(df->type, df->op, df->lsm_str,
        (void **)&df->lsm_rule);
    /* Keep currently invalid fields around in case they
     * become valid after a policy reload. */
    if (ret == -EINVAL) {
        pr_warn("audit rule for LSM \\'%s\\' is invalid\n",
            df->lsm_str);
        ret = 0;
    }
    return ret;
}
```

quote/comment cell

Searching for interpretable cells

```
#ifdef CONFIG_AUDITSYSCALL
static inline int audit_match_class_bits(int class, u32 *mask)
{
    int i;
    if (classes[class]) {
        for (i = 0; i < AUDIT_BITMASK_SIZE; i++)
            if (mask[i] & classes[class][i])
                return 0;
    }
    return 1;
}
```

code depth cell

RNN tradeoffs

RNN Advantages:

- Can process any length of the input (no context length)
- Computation for step t can (in theory) use information from many steps back
- Model size does not increase for longer input
- The same weights are applied on every timestep, so there is symmetry in how inputs are processed.

RNN Disadvantages:

- Recurrent computation is slow
- In practice, difficult to access information from many steps back

Image Captioning

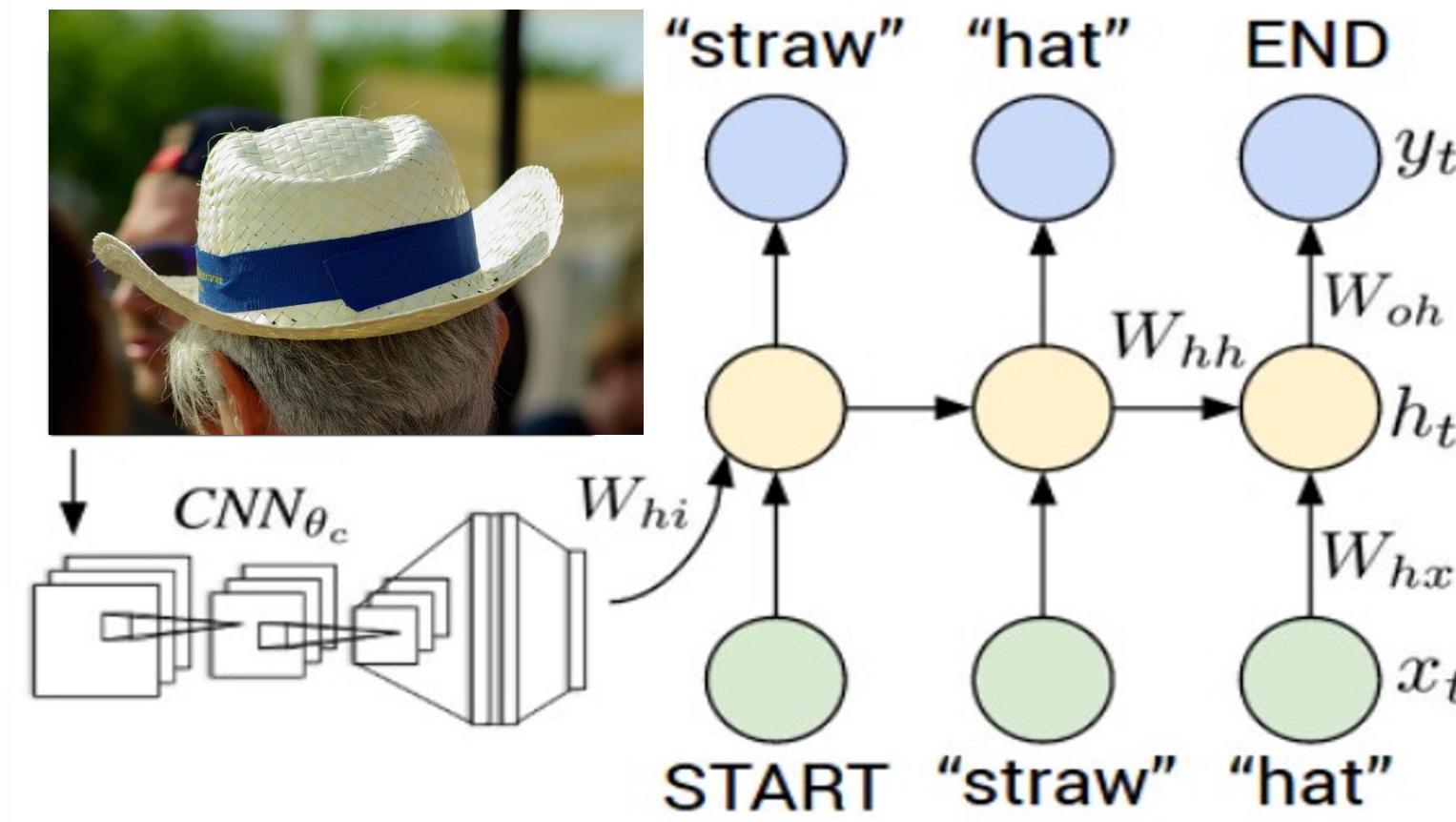


Figure from Karpathy et al, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015; figure copyright IEEE, 2015.
Reproduced for educational purposes.

Explain Images with Multimodal Recurrent Neural Networks, Mao et al.

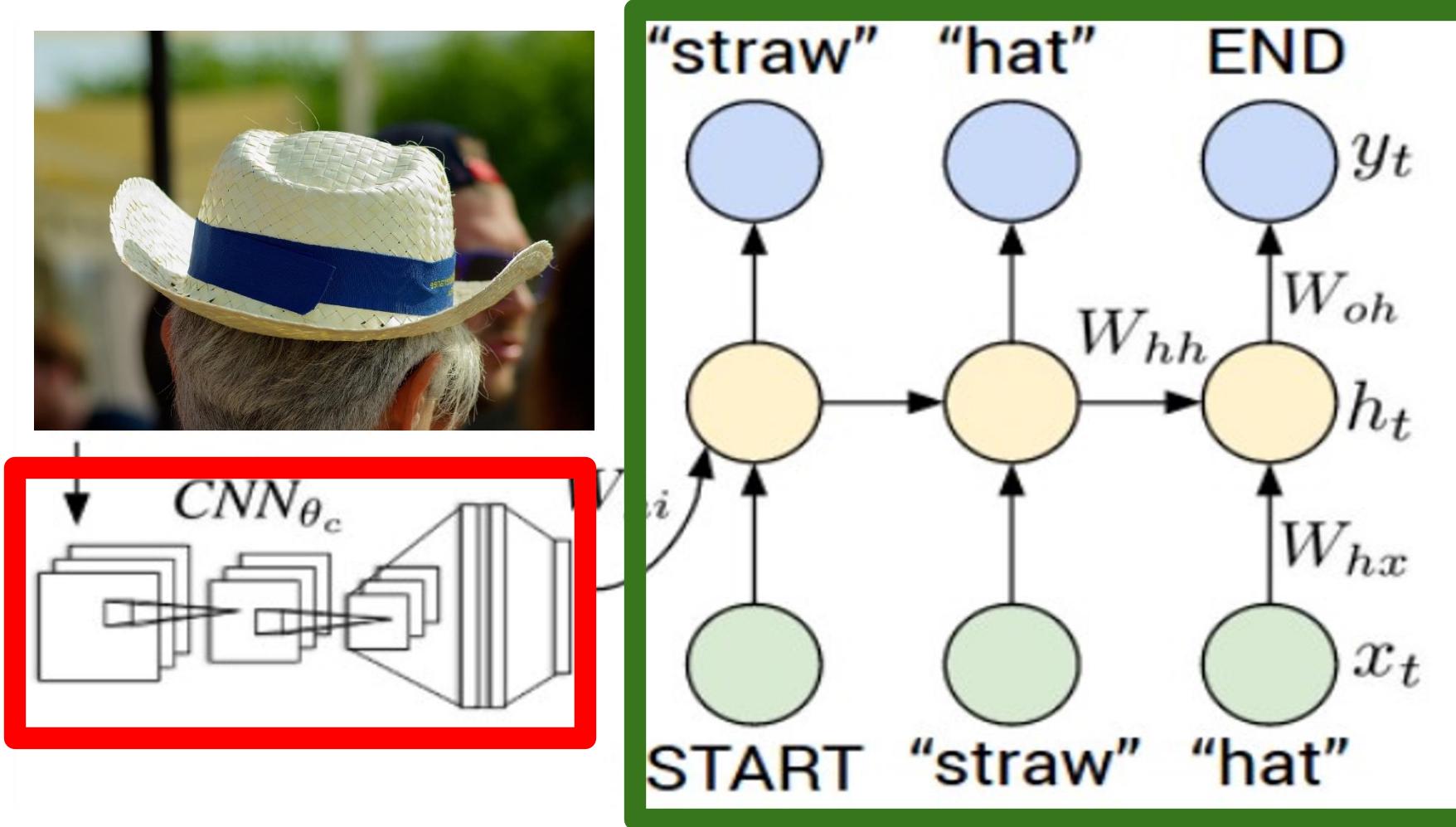
Deep Visual-Semantic Alignments for Generating Image Descriptions, Karpathy and Fei-Fei

Show and Tell: A Neural Image Caption Generator, Vinyals et al.

Long-term Recurrent Convolutional Networks for Visual Recognition and Description, Donahue et al.

Learning a Recurrent Visual Representation for Image Caption Generation, Chen and Zitnick

Recurrent Neural Network



Convolutional Neural Network

test image



This image is [CC0 public domain](#)

image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096

FC-1000

softmax



test image

image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096

FC-1000

softmax



test image

image



conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096

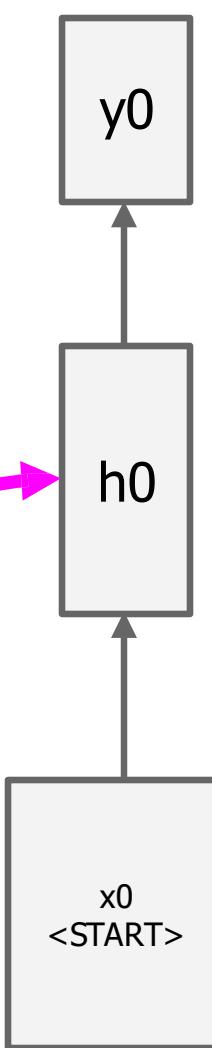


test image

x0
<START>



test image



before:

$$h = \tanh(W_{xh} * x + W_{hh} * h)$$

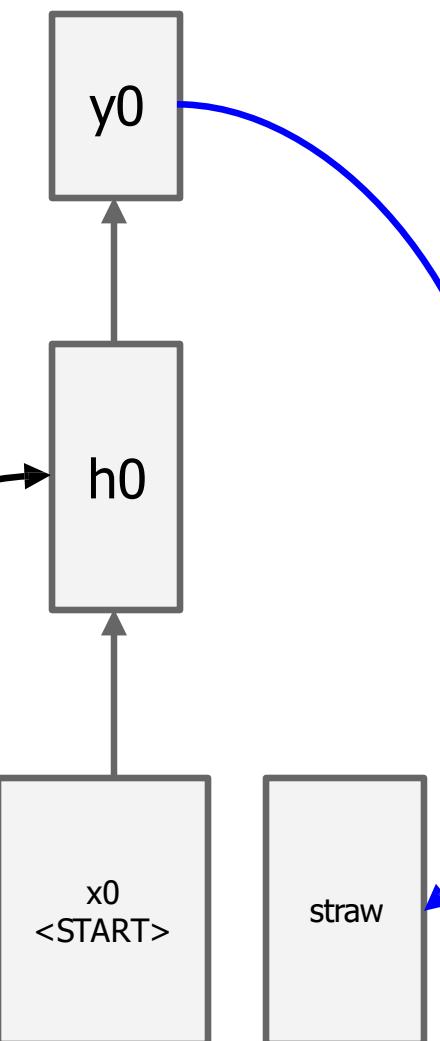
now:

$$h = \tanh(W_{xh} * x + W_{hh} * h + W_{ih} * v)$$



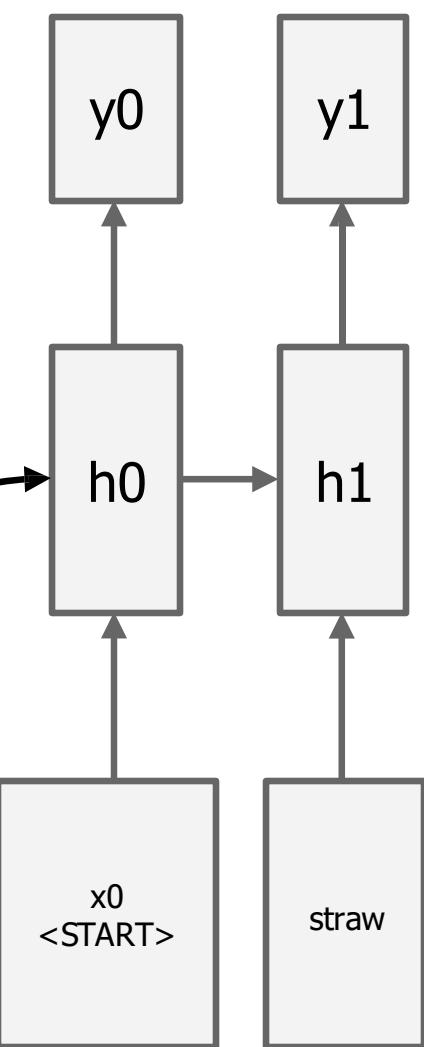
test image

sample!



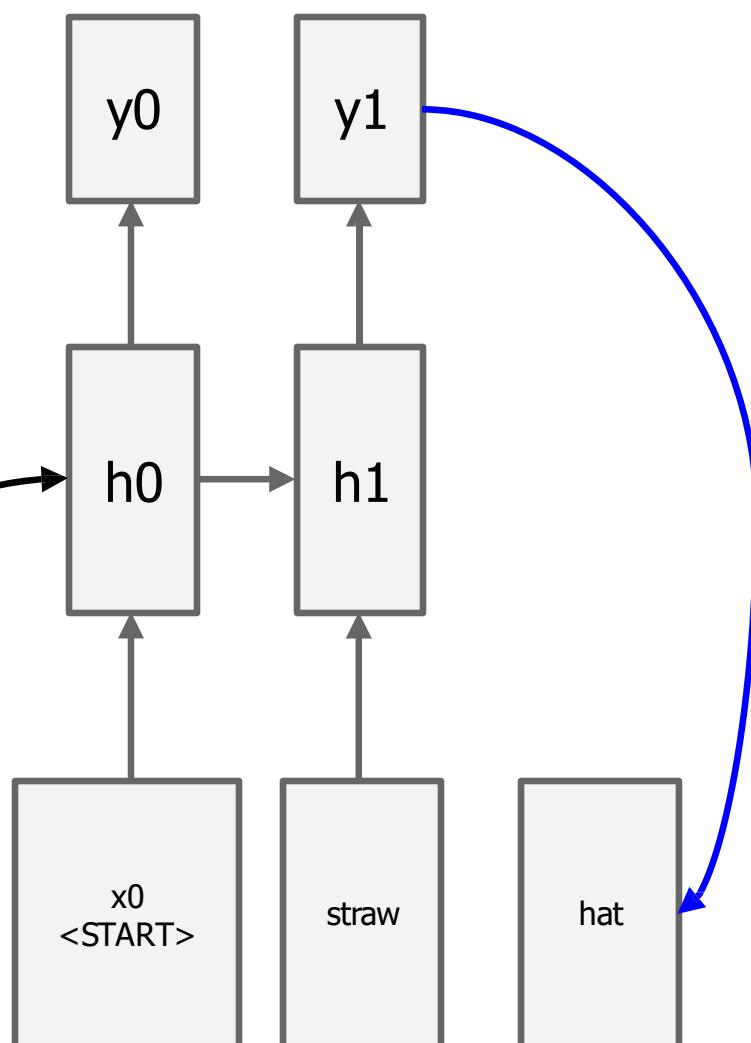


test image





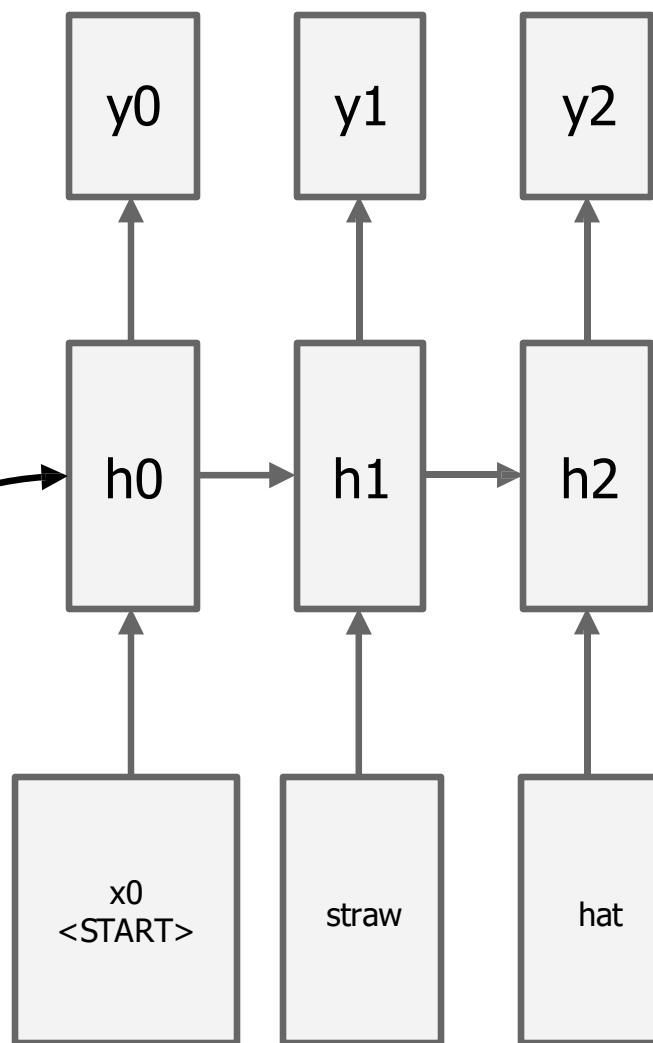
test image



sample!

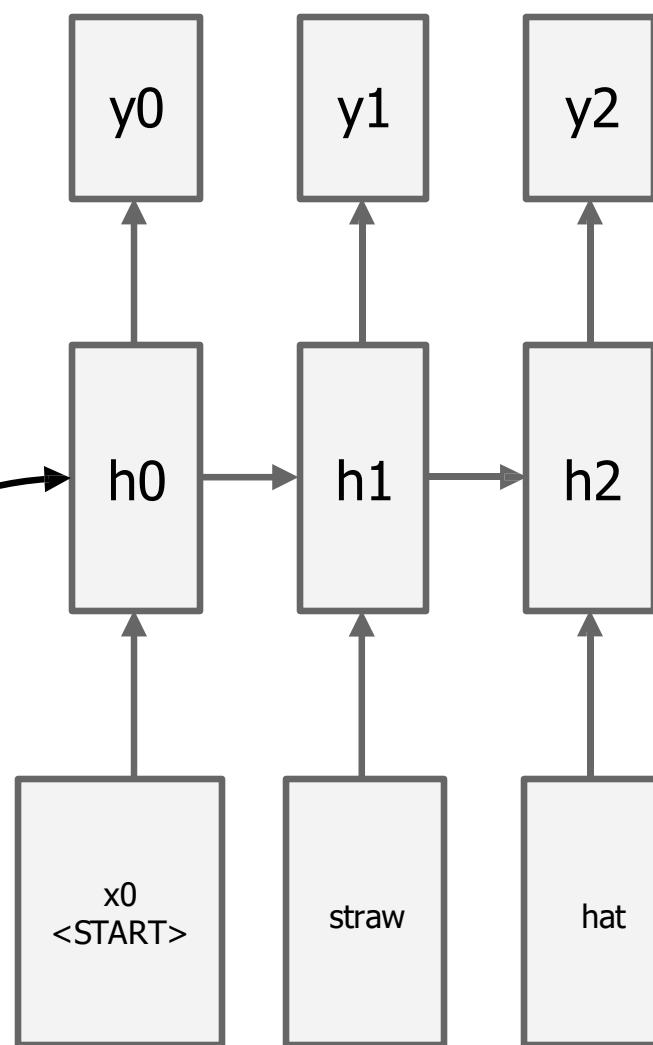


test image





test image



sample
<END> token
=> finish.

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

Image Captioning: Failure Cases

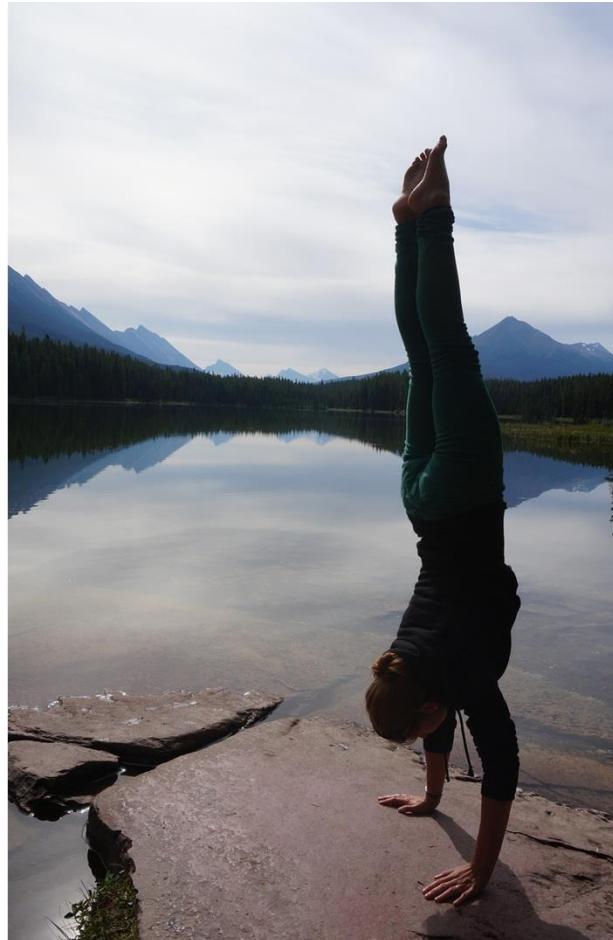
Captions generated using [neuraltalk2](#)
All images are CC0 Public domain: [fur coat](#),
[handstand](#), [spider web](#), [baseball](#)



A woman is holding a cat in her hand



A person holding a computer mouse on a desk



A woman standing on a beach holding a surfboard



A bird is perched on a tree branch



A man in a baseball uniform throwing a ball

Visual Question Answering (VQA)



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 1/4 Rd.
- A: Onto 25 1/4 Rd.
- A: Onto 23 1/4 Rd.
- A: Onto Main Street.

Agrawal et al, "VQA: Visual Question Answering", ICCV 2015

Zhu et al, "Visual 7W: Grounded Question Answering in Images", CVPR 2016

Figure from Zhu et al, copyright IEEE 2016. Reproduced for educational purposes.

Visual Dialog: Conversations about images



Visual Dialog

A cat drinking water out of a coffee mug.

White and red

No, something is there can't tell what it is

Yes, they are

What color is the mug?

Are there any pictures on it?

Is the mug and cat on a table?

Start typing question here ...

C >

Das et al, "Visual Dialog", CVPR 2017

Figures from Das et al, copyright IEEE 2017. Reproduced with permission.

Visual Language Navigation: Go to the living room

Agent encodes instructions in language and uses an RNN to generate a series of movements as the visual input changes after each move.

Instruction

Turn right and head towards the *kitchen*. Then turn left, pass a *table* and enter the *hallway*. Walk down the hallway and turn into the *entry way* to your right *without doors*. Stop in front of the *toilet*.



Initial Position



Target Position



Demonstration Path A

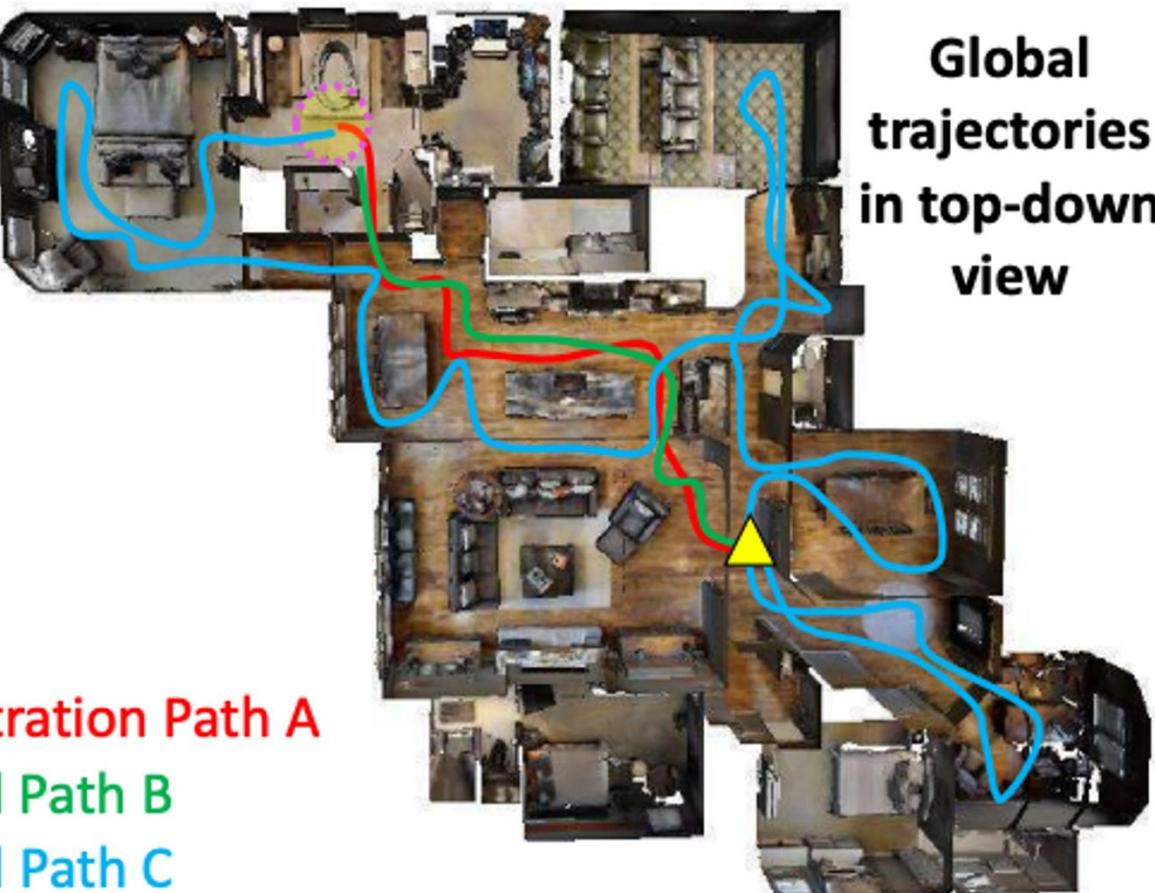


Executed Path B



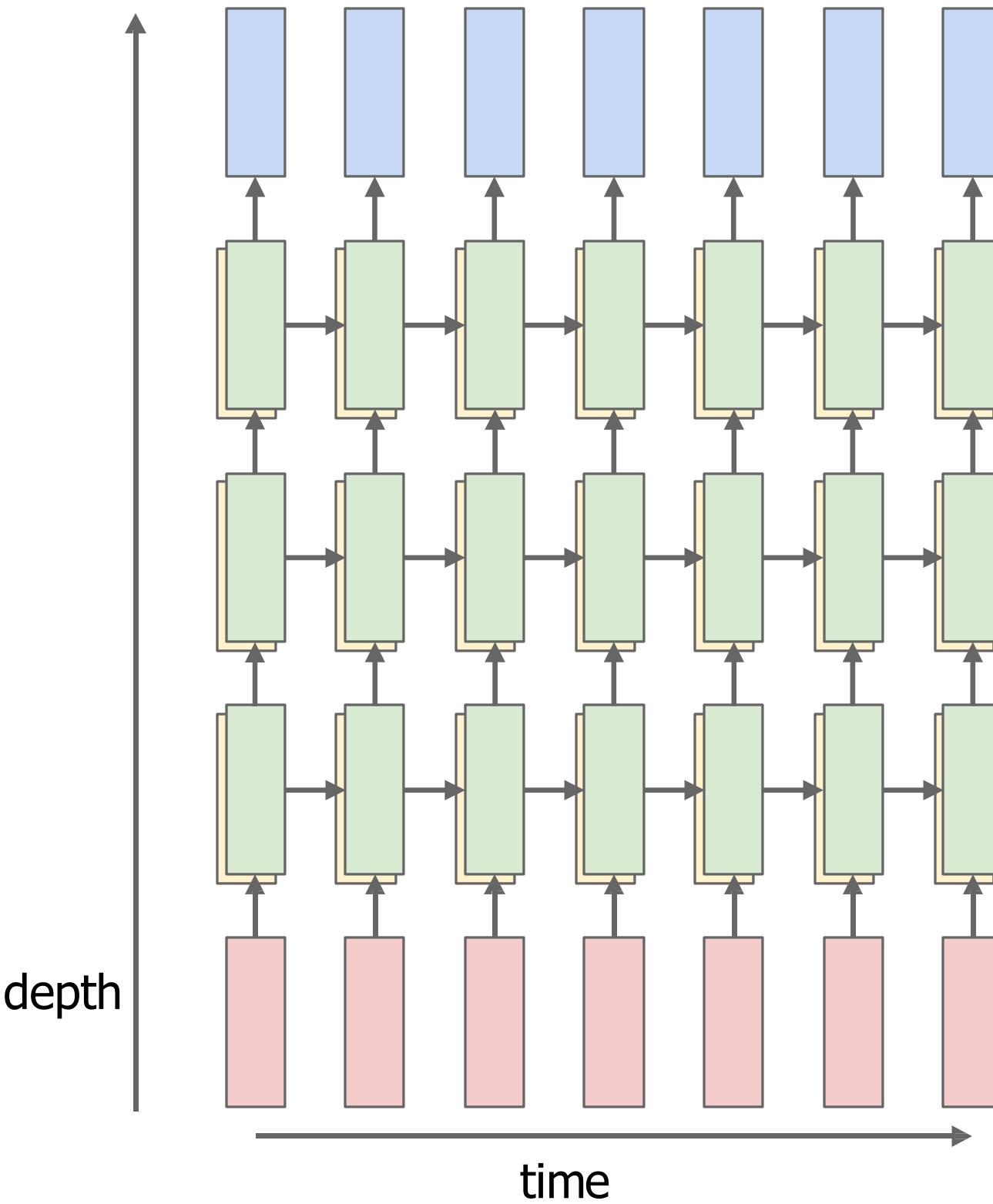
Executed Path C

Wang et al, "Reinforced Cross-Modal Matching and Self-Supervised Imitation Learning for Vision-Language Navigation", CVPR 2018
Figures from Wang et al, copyright IEEE 2017. Reproduced with permission.



Global
trajectories
in top-down
view

Multilayer RNNs



RNN Variants: Long Short Term Memory (LSTM)

Vanilla RNN

$$h_t = \tanh \left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)$$

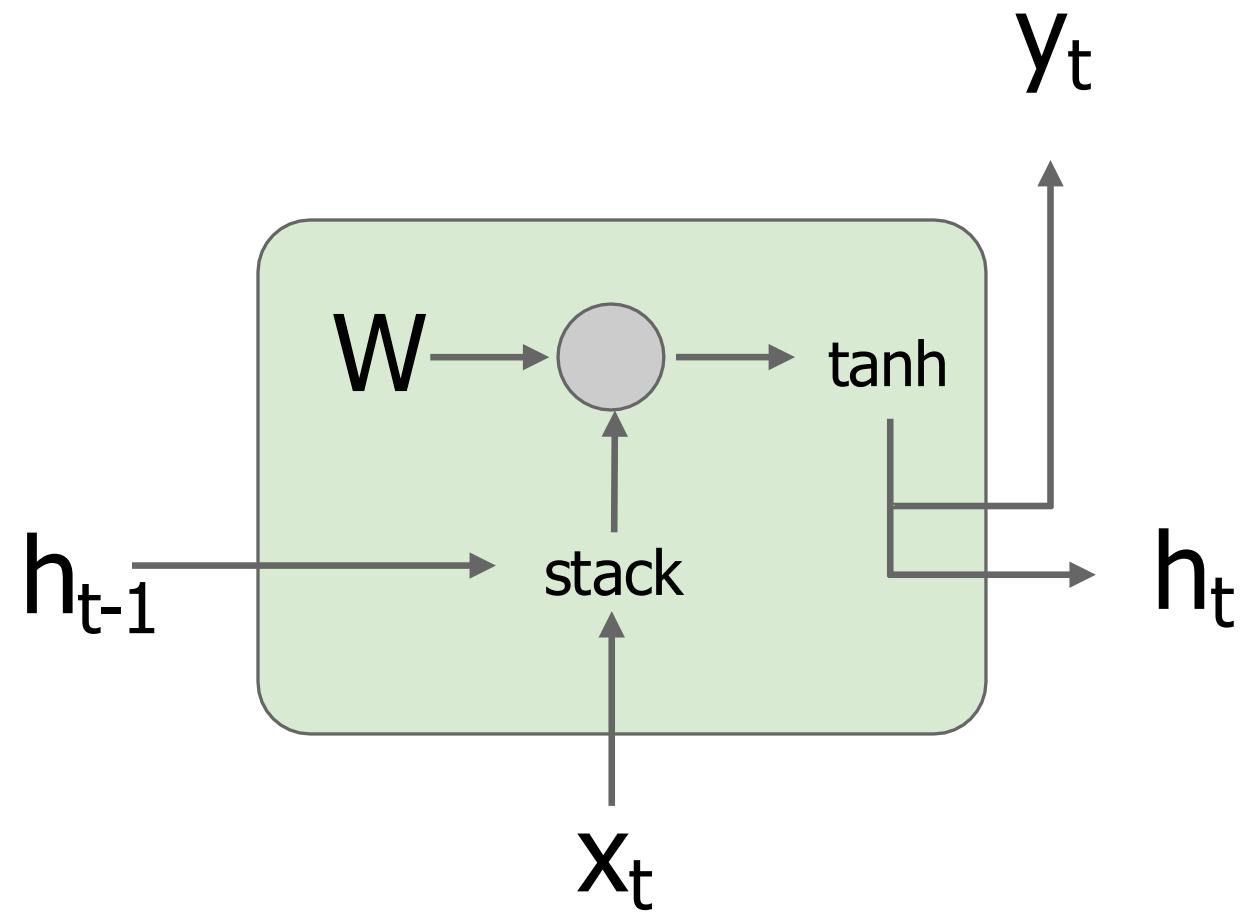
LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$
$$c_t = f \odot c_{t-1} + i \odot g$$
$$h_t = o \odot \tanh(c_t)$$

Hochreiter and Schmidhuber, "Long Short Term Memory", Neural Computation 1997

Vanilla RNN Gradient Flow

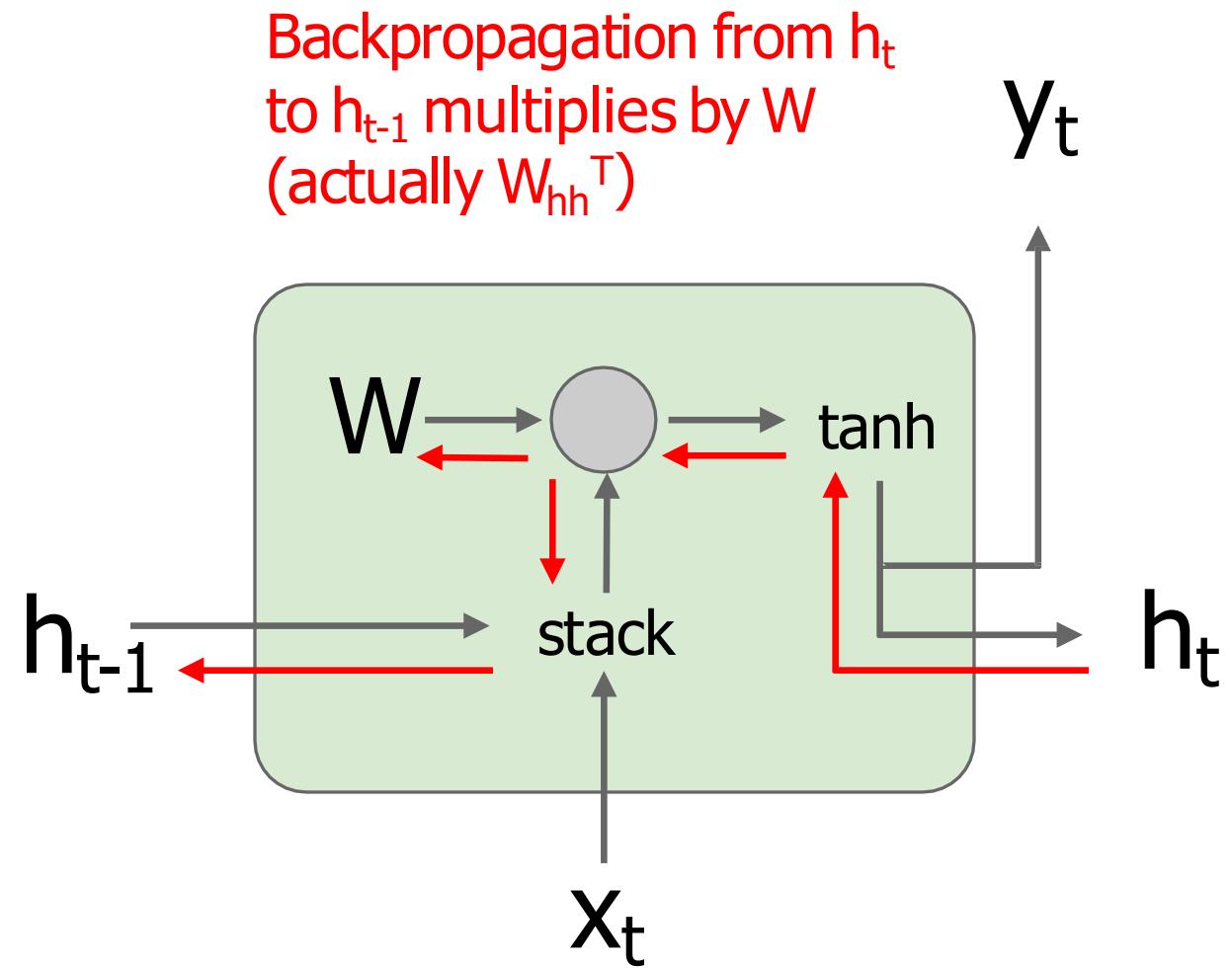
Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013



$$\begin{aligned} h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\ &= \tanh \left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right) \\ &= \tanh \left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right) \end{aligned}$$

Vanilla RNN Gradient Flow

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013

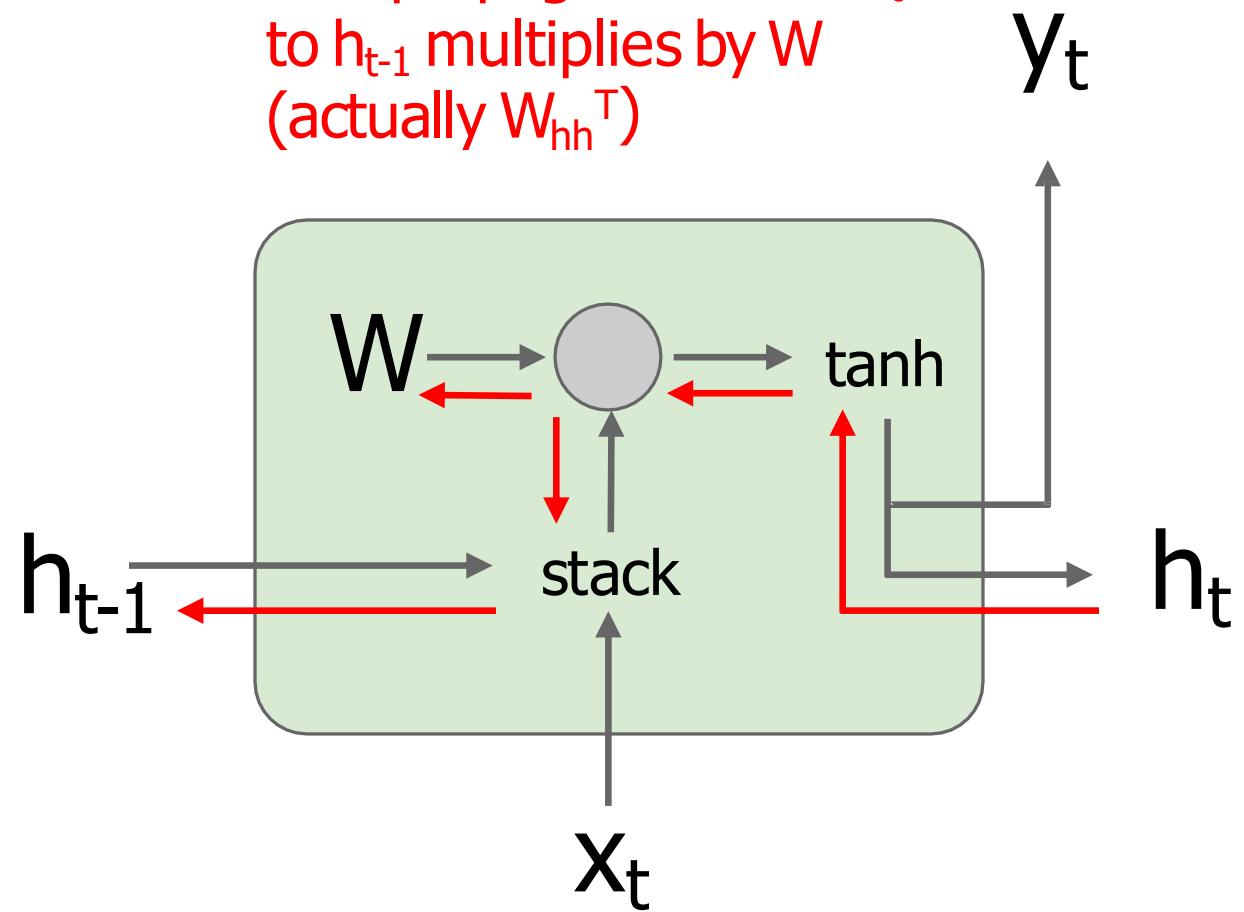


$$\begin{aligned} h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\ &= \tanh \left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right) \\ &= \tanh \left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right) \end{aligned}$$

Vanilla RNN Gradient Flow

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013

Backpropagation from h_t
to h_{t-1} multiplies by W
(actually W_{hh}^T)



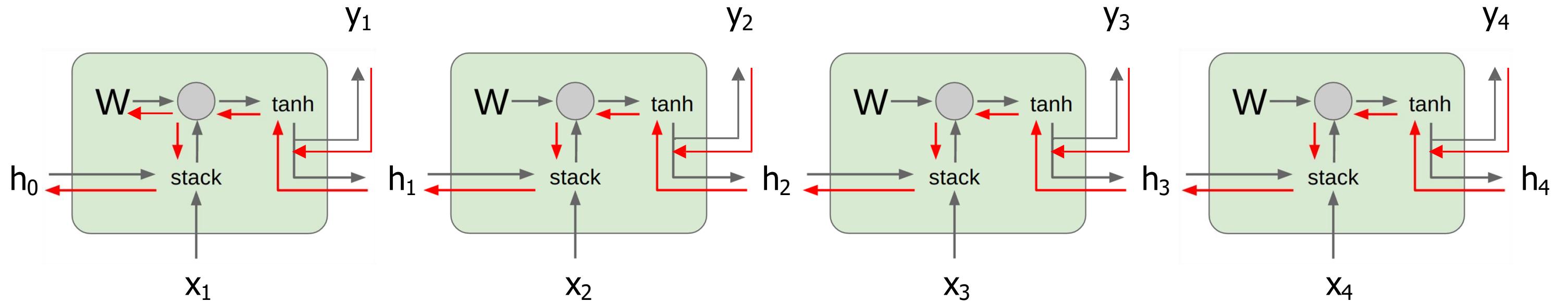
$$\begin{aligned} h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\ &= \tanh \left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right) \\ &= \tanh \left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right) \end{aligned}$$

$$\frac{\partial h_t}{\partial h_{t-1}} = \tanh'(W_{hh}h_{t-1} + W_{xh}x_t)W_{hh}$$

Vanilla RNN Gradient Flow

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994

Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013



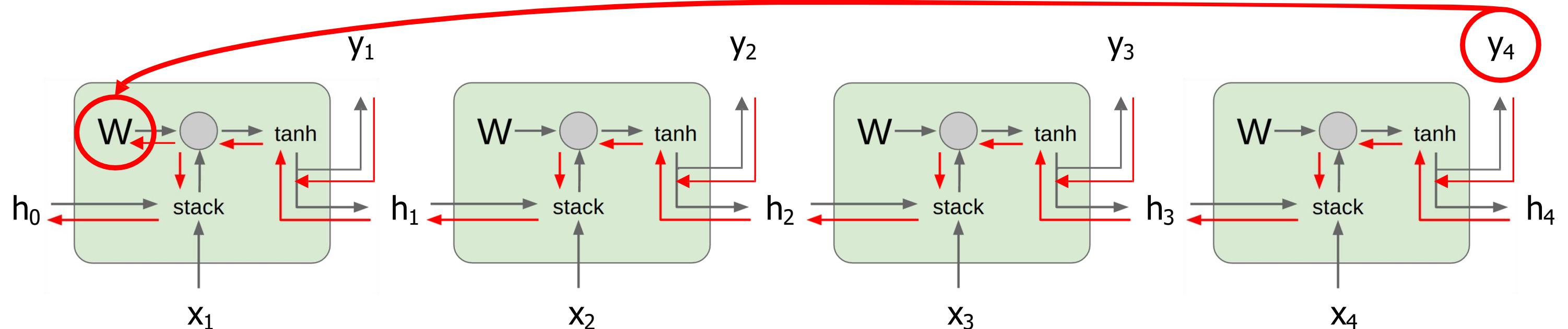
$$\frac{\partial L}{\partial W} = \sum_{t=1}^T \frac{\partial L_t}{\partial W}$$

Vanilla RNN Gradient Flow

Gradients over multiple time steps:

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994

Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013



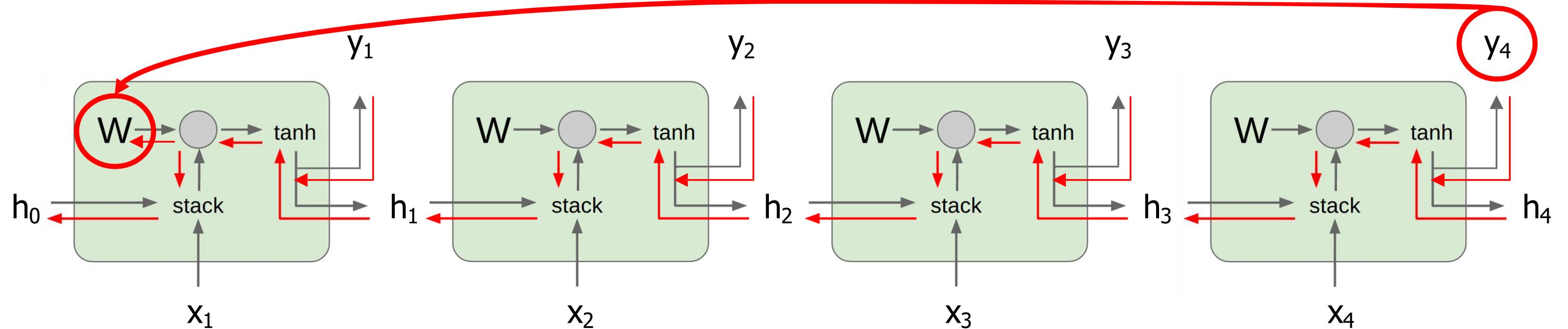
$$\frac{\partial L}{\partial W} = \sum_{t=1}^T \frac{\partial L_t}{\partial W}$$

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \frac{\partial h_t}{\partial h_{t-1}} \cdots \frac{\partial h_1}{\partial W}$$

Vanilla RNN Gradient Flow

Gradients over multiple time steps:

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013



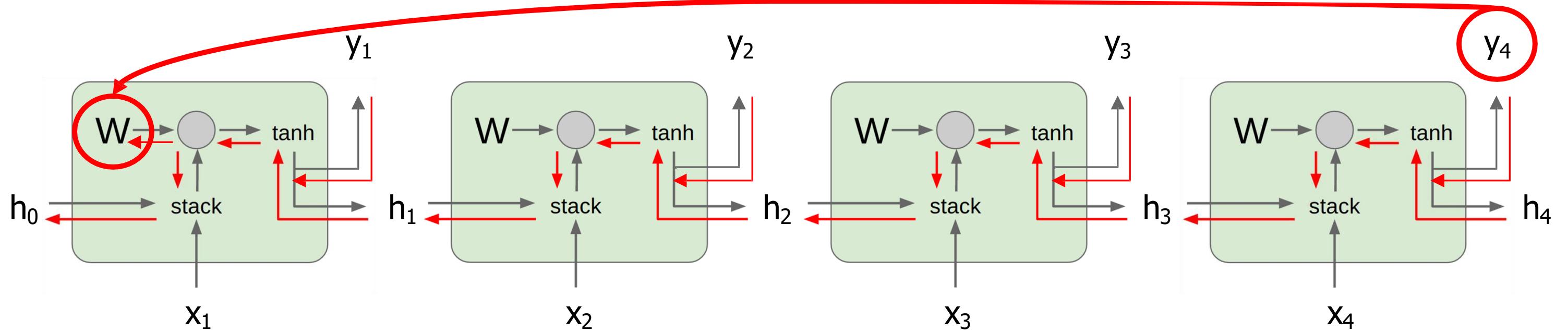
$$\frac{\partial L}{\partial W} = \sum_{t=1}^T \frac{\partial L_t}{\partial W}$$

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \frac{\partial h_t}{\partial h_{t-1}} \cdots \frac{\partial h_1}{\partial W} = \frac{\partial L_T}{\partial h_T} \left(\prod_{t=2}^T \frac{\partial h_t}{\partial h_{t-1}} \right) \frac{\partial h_1}{\partial W}$$

Vanilla RNN Gradient Flow

Gradients over multiple time steps:

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
 Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013



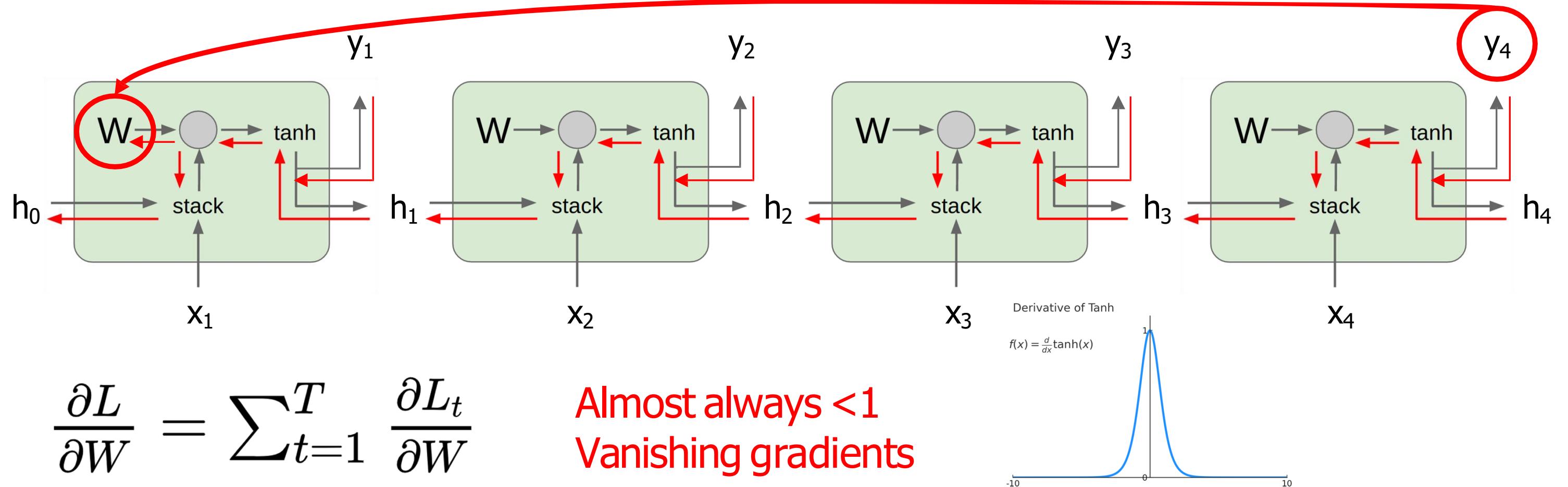
$$\frac{\partial L}{\partial W} = \sum_{t=1}^T \frac{\partial L_t}{\partial W} \quad \boxed{\frac{\partial h_t}{\partial h_{t-1}} = \tanh'(W_{hh}h_{t-1} + W_{xh}x_t)W_{hh}}$$

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \frac{\partial h_t}{\partial h_{t-1}} \cdot \dots \frac{\partial h_1}{\partial W} = \frac{\partial L_T}{\partial h_T} \left(\prod_{t=2}^T \boxed{\frac{\partial h_t}{\partial h_{t-1}}} \right) \frac{\partial h_1}{\partial W}$$

Vanilla RNN Gradient Flow

Gradients over multiple time steps:

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
 Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013

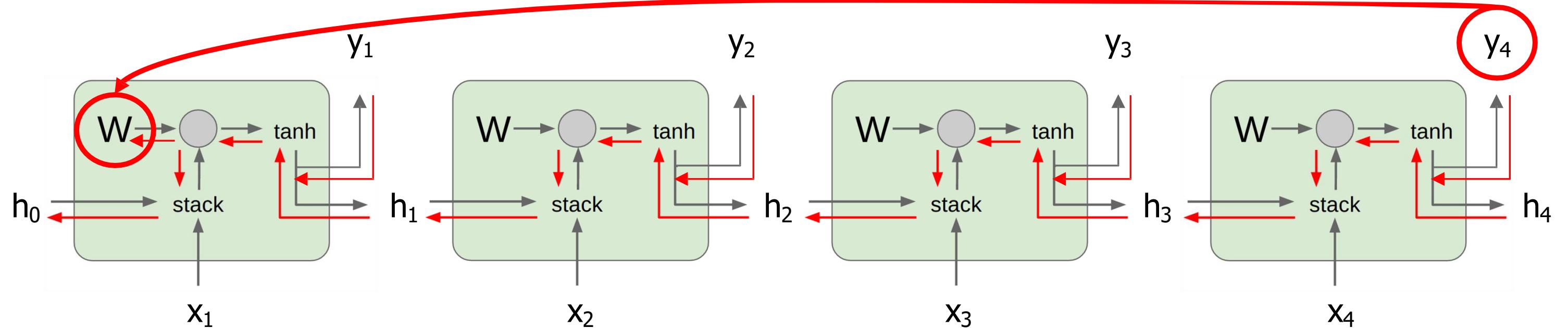


$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \left(\prod_{t=2}^T \boxed{\tanh'(W_{hh} h_{t-1} + W_{xh} x_t)} \right) W_{hh}^{T-1} \frac{\partial h_1}{\partial W}$$

Vanilla RNN Gradient Flow

Gradients over multiple time steps:

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013



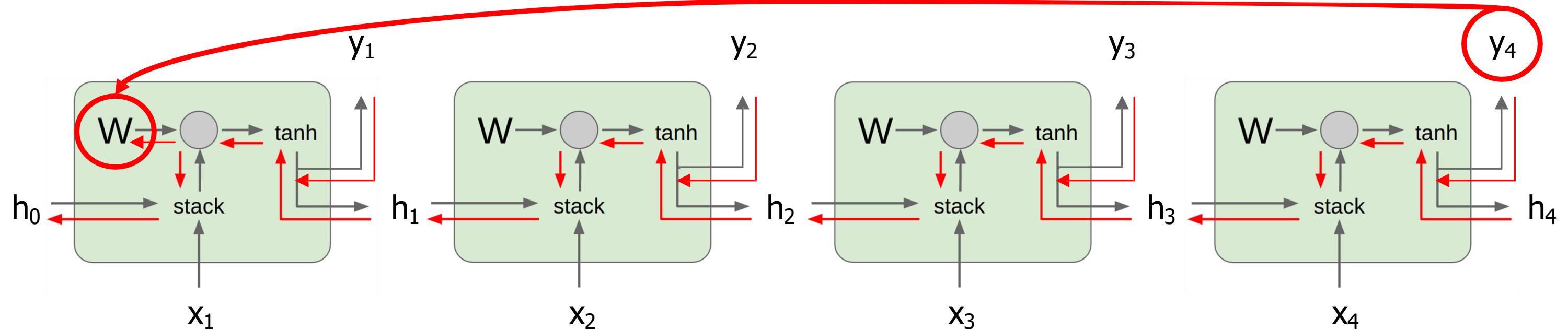
$$\frac{\partial L}{\partial W} = \sum_{t=1}^T \frac{\partial L_t}{\partial W}$$

What if we assumed no non-linearity?

Vanilla RNN Gradient Flow

Gradients over multiple time steps:

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013



What if we assumed no non-linearity?

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

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \boxed{W_{hh}^{T-1}} \frac{\partial h_1}{\partial W}$$

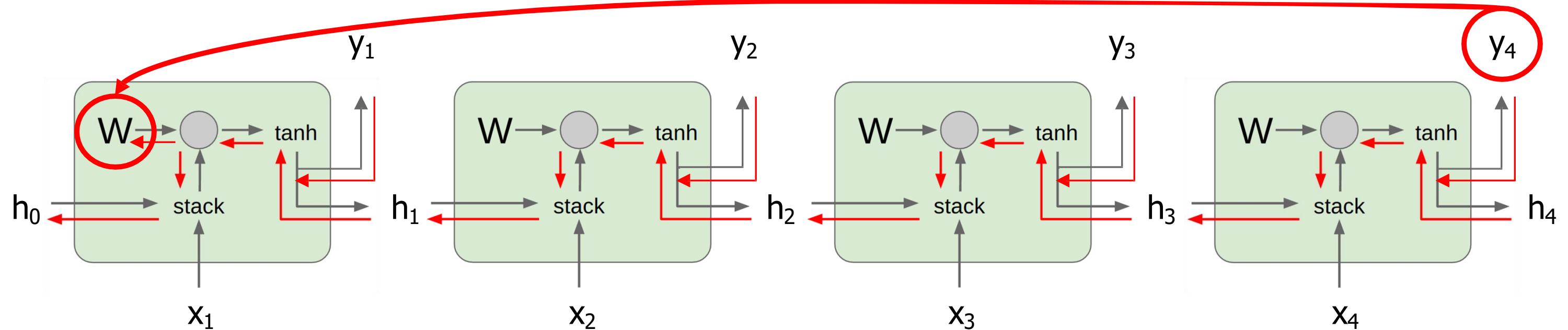
Largest singular value > 1 :
Exploding gradients

Largest singular value < 1 :
Vanishing gradients

Vanilla RNN Gradient Flow

Gradients over multiple time steps:

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013



What if we assumed no non-linearity?

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

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \boxed{W_{hh}^{T-1}} \frac{\partial h_1}{\partial W}$$

Largest singular value > 1 :
Exploding gradients

Largest singular value < 1 :
Vanishing gradients

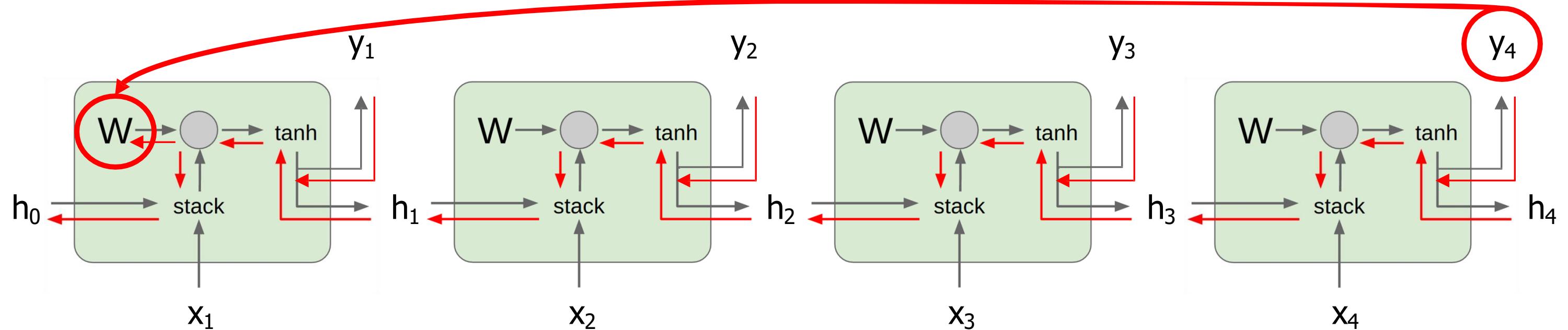
→ Gradient clipping: Scale gradient if its norm is too big

```
grad_norm = np.sum(grad * grad)
if grad_norm > threshold:
    grad *= (threshold / grad_norm)
```

Vanilla RNN Gradient Flow

Gradients over multiple time steps:

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013



What if we assumed no non-linearity?

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

Largest singular value > 1 :
Exploding gradients

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \boxed{W_{hh}^{T-1}} \frac{\partial h_1}{\partial W}$$

Largest singular value < 1 :
Vanishing gradients

→ Change RNN architecture

Long Short Term Memory (LSTM) – A Historical Note

Vanilla RNN

$$h_t = \tanh \left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)$$

LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Hochreiter and Schmidhuber, "Long Short Term Memory", Neural Computation 1997

Long Short Term Memory (LSTM)

Vanilla RNN

$$h_t = \tanh \left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)$$

LSTM

Four gates

Cell state

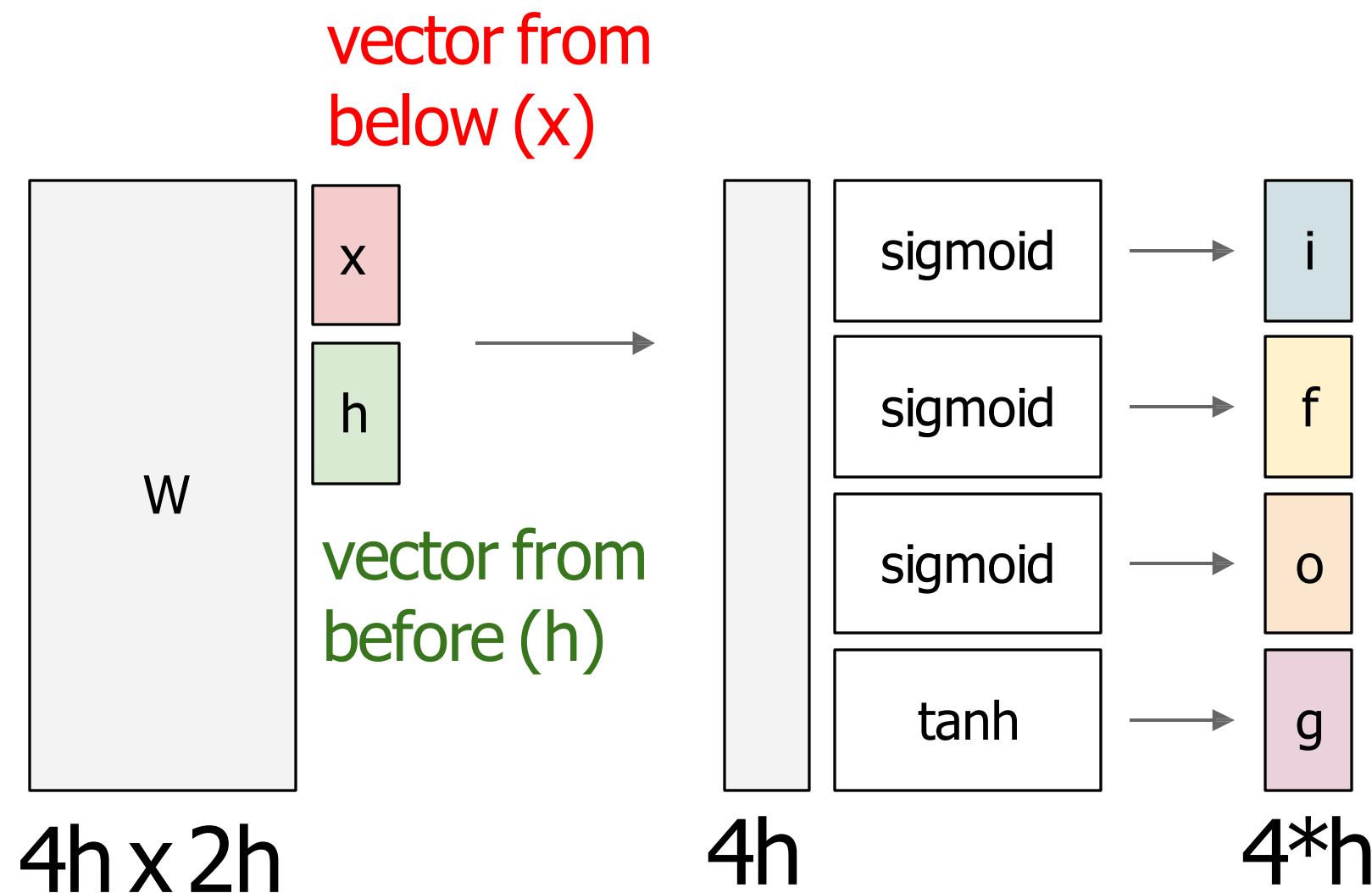
Hidden state

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$
$$c_t = f \odot c_{t-1} + i \odot g$$
$$h_t = o \odot \tanh(c_t)$$

Hochreiter and Schmidhuber, "Long Short Term Memory", Neural Computation 1997

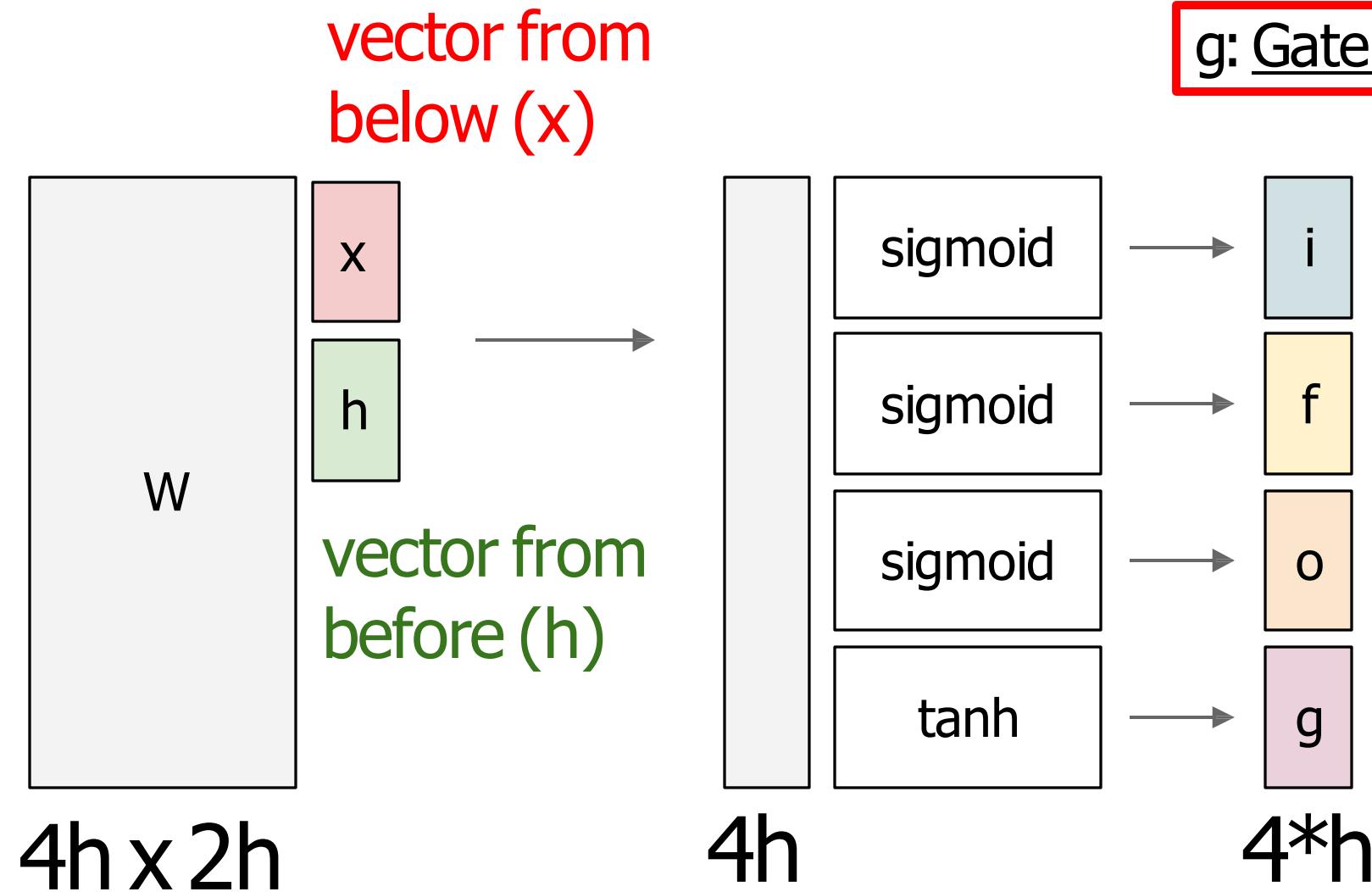
Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



g: Gate (?), How much to write to cell

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

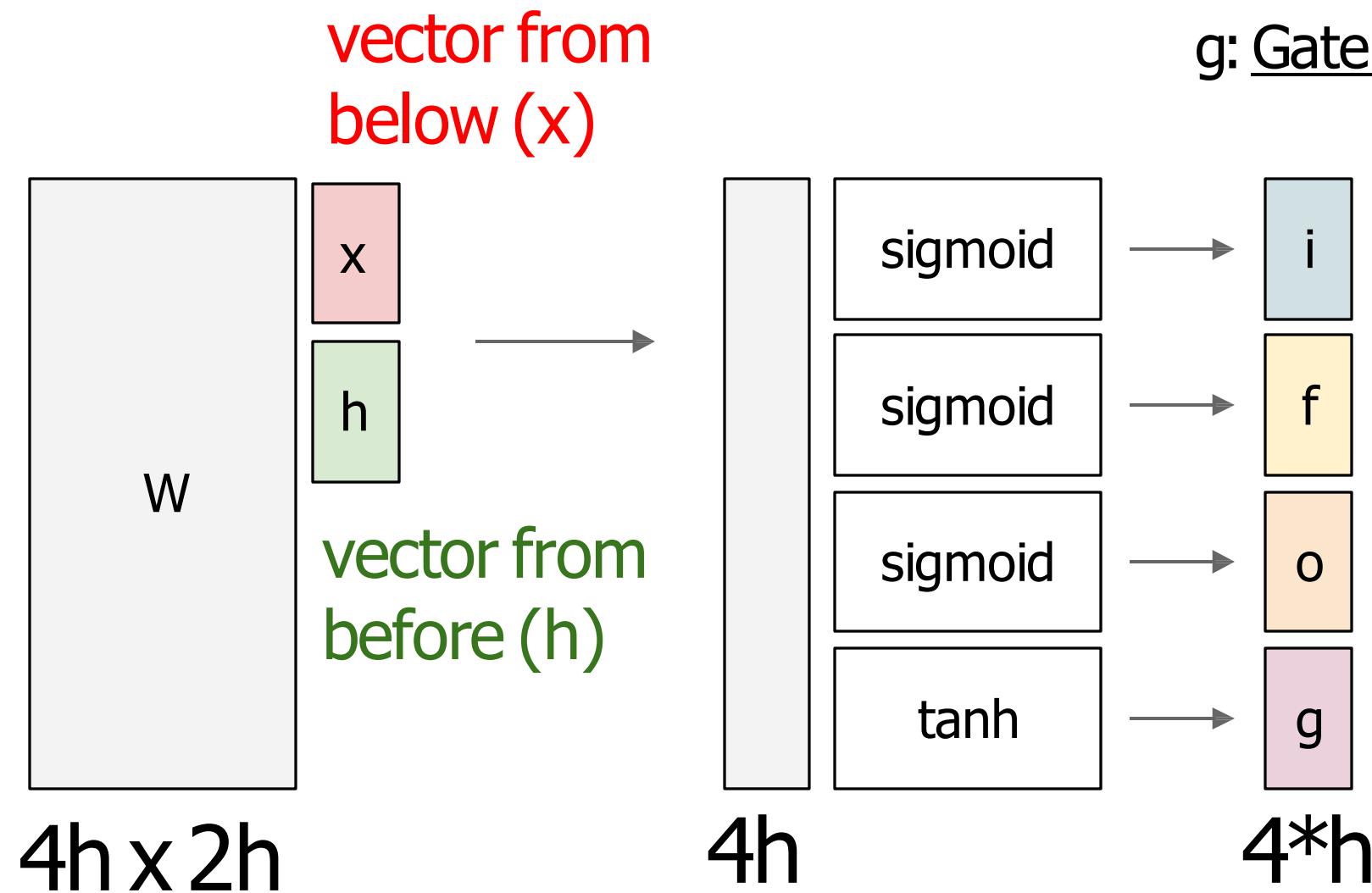
$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

i: Input gate, whether to write to cell



g: Gate gate (?), How much to write to cell

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

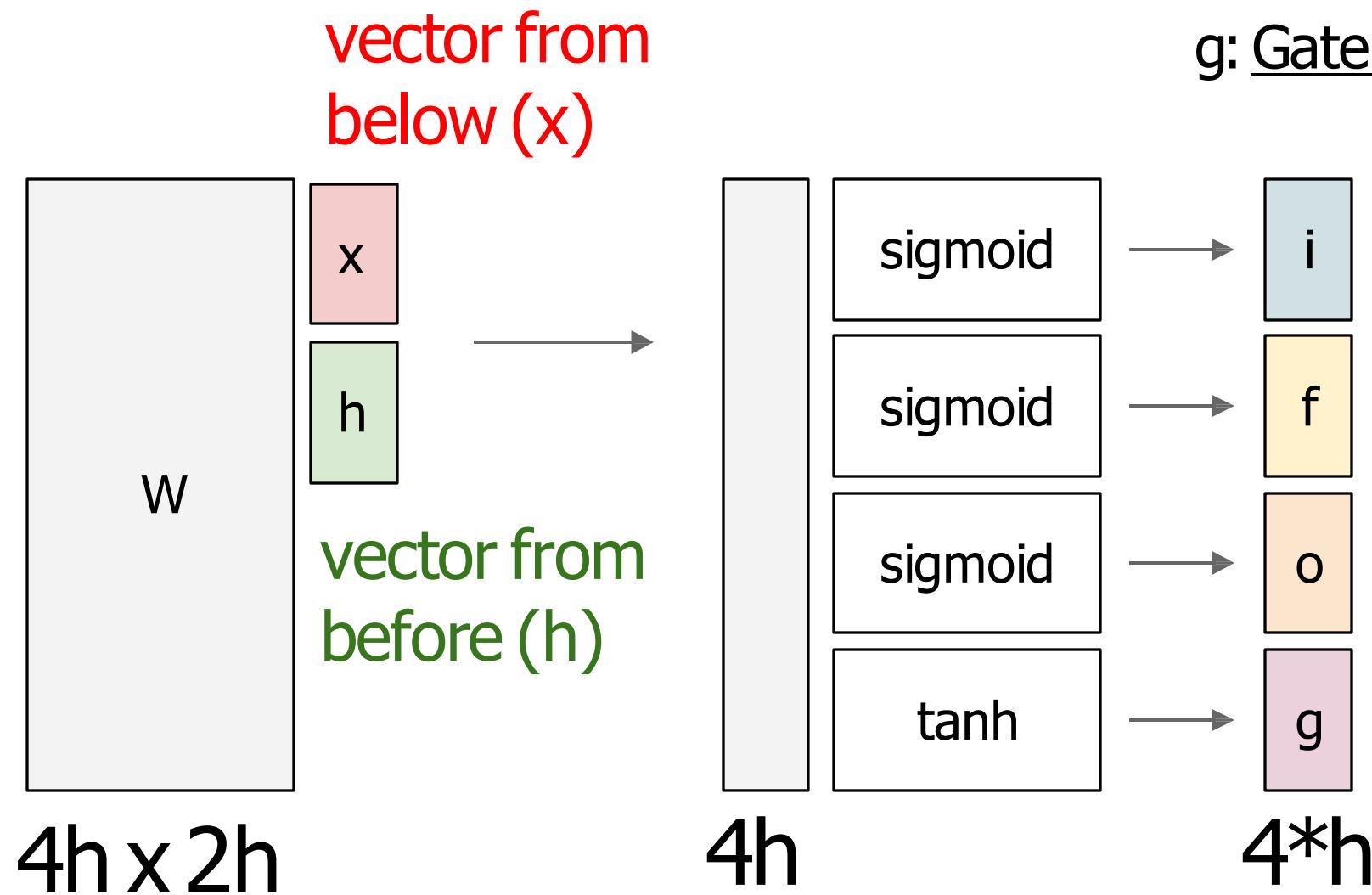
$$c_t = f \odot c_{t-1} + i \odot g$$
$$h_t = o \odot \tanh(c_t)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

i: Input gate, whether to write to cell

f: Forget gate, Whether to erase cell



g: Gate gate (?), How much to write to cell

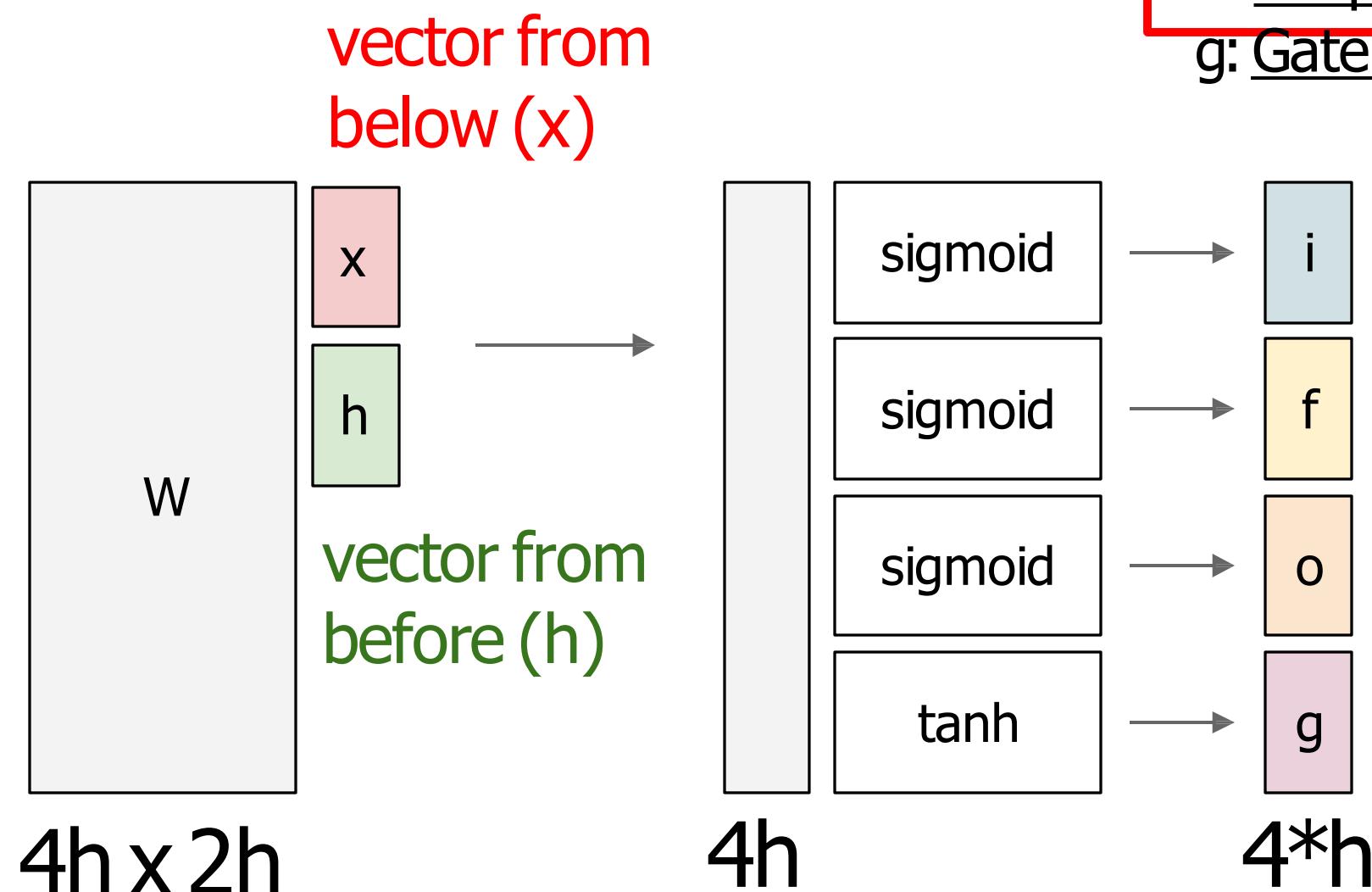
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



i: Input gate, whether to write to cell

f: Forget gate, Whether to erase cell

o: Output gate, How much to reveal cell

g: Gate gate (?), How much to write to cell

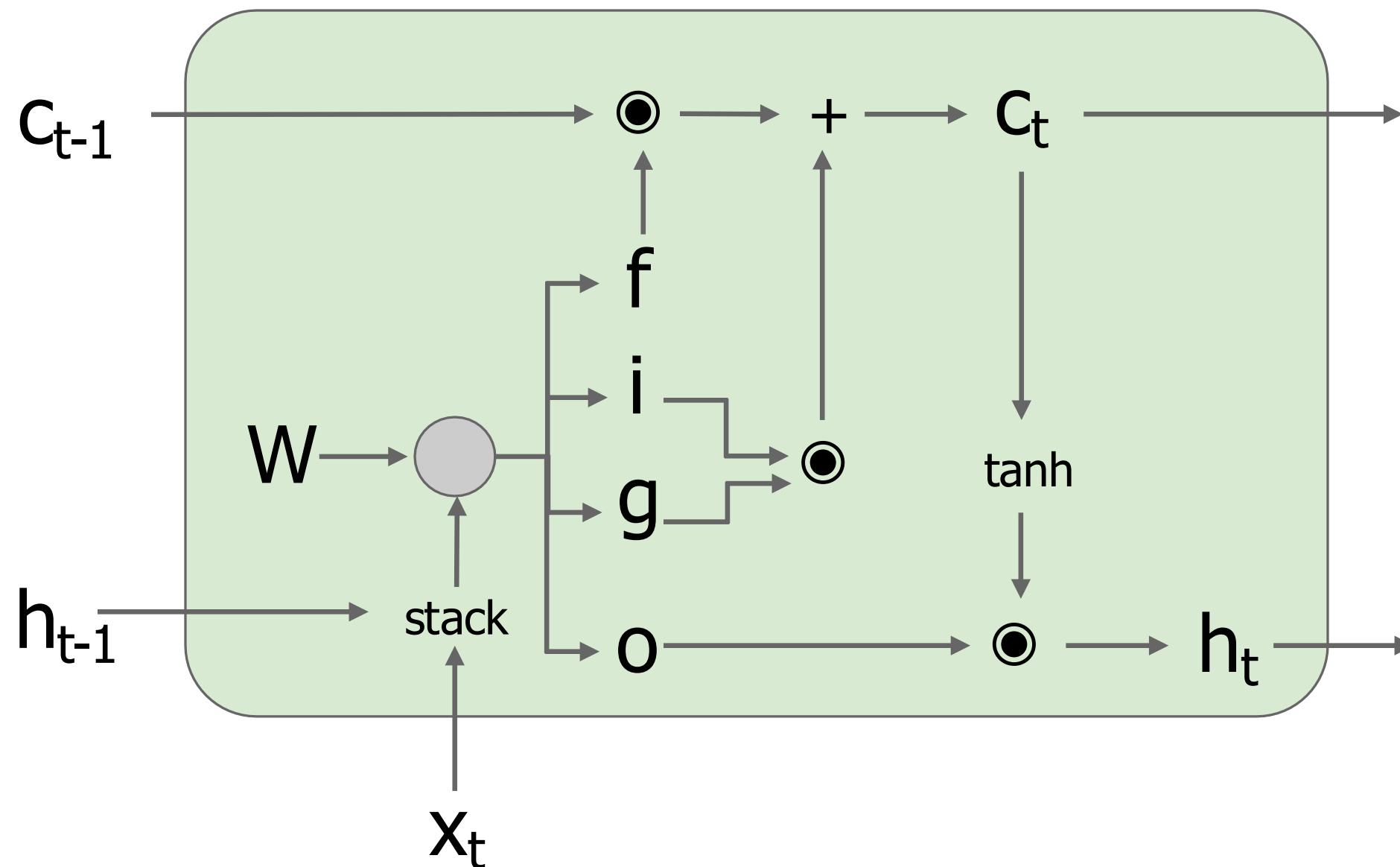
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



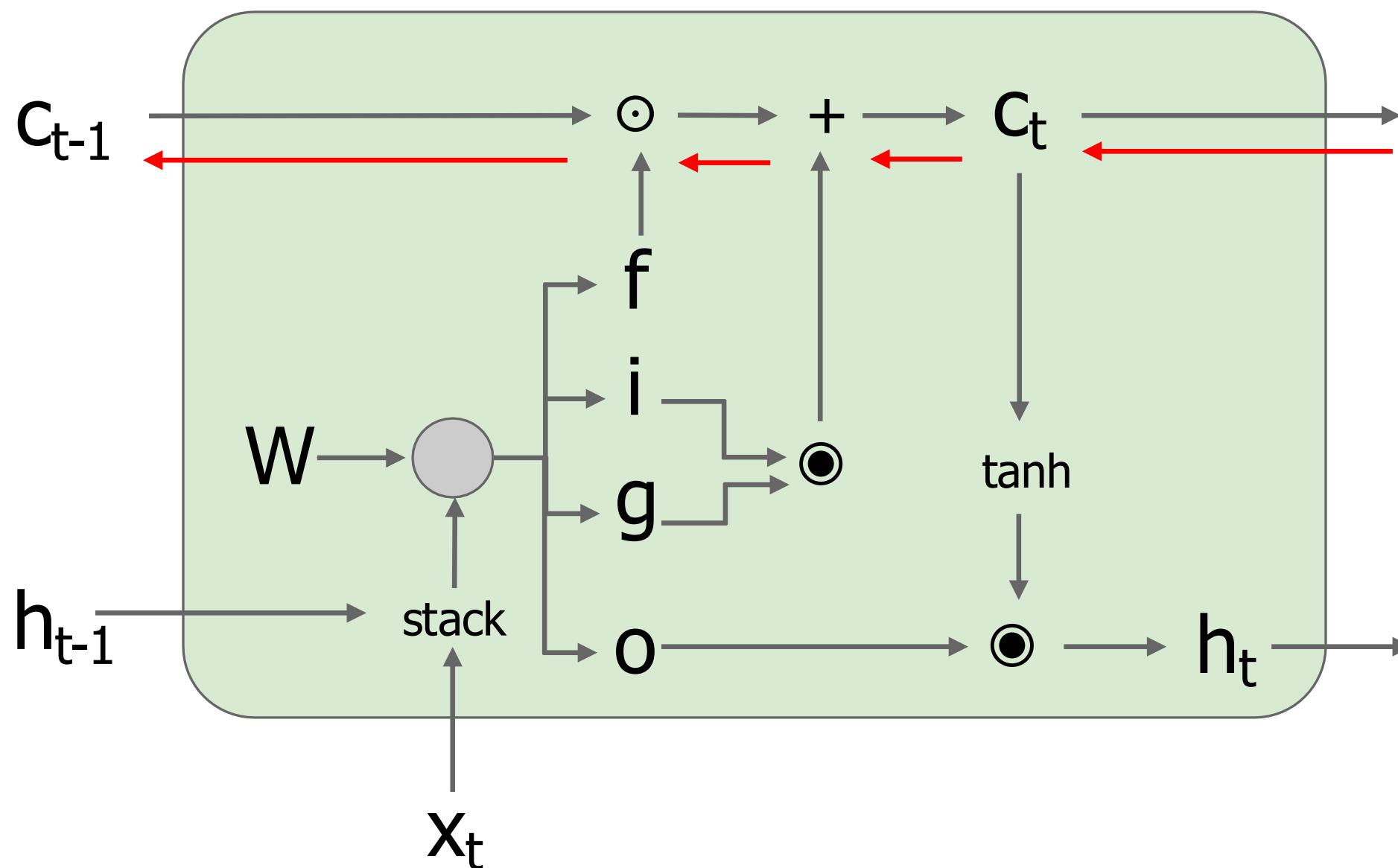
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Long Short Term Memory (LSTM): Gradient Flow

[Hochreiter et al., 1997]



Backpropagation from c_t to c_{t-1} only elementwise multiplication by f , no matrix multiply by W

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

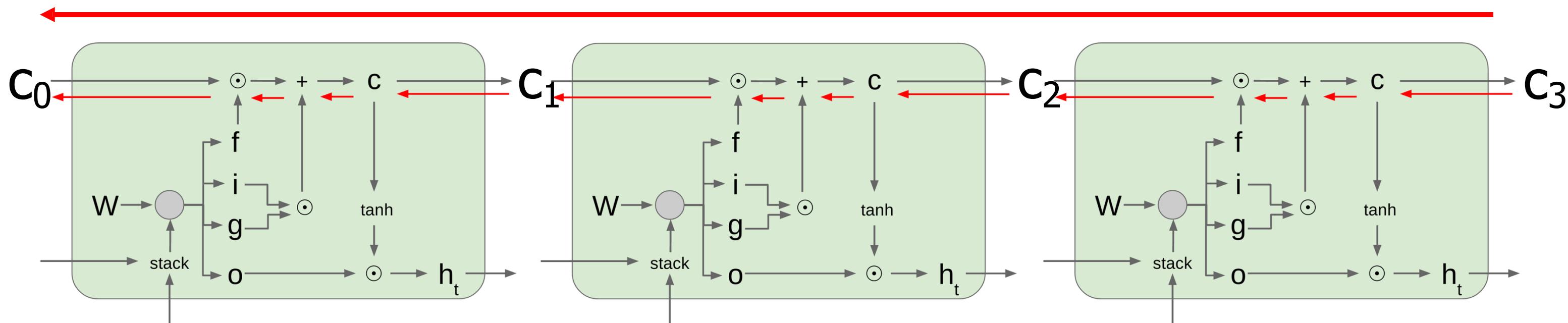
$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Long Short Term Memory (LSTM): Gradient Flow

[Hochreiter et al., 1997]

Uninterrupted gradient flow!



Do LSTMs solve the vanishing gradient problem?

The LSTM architecture makes it easier for the RNN to preserve information over many timesteps

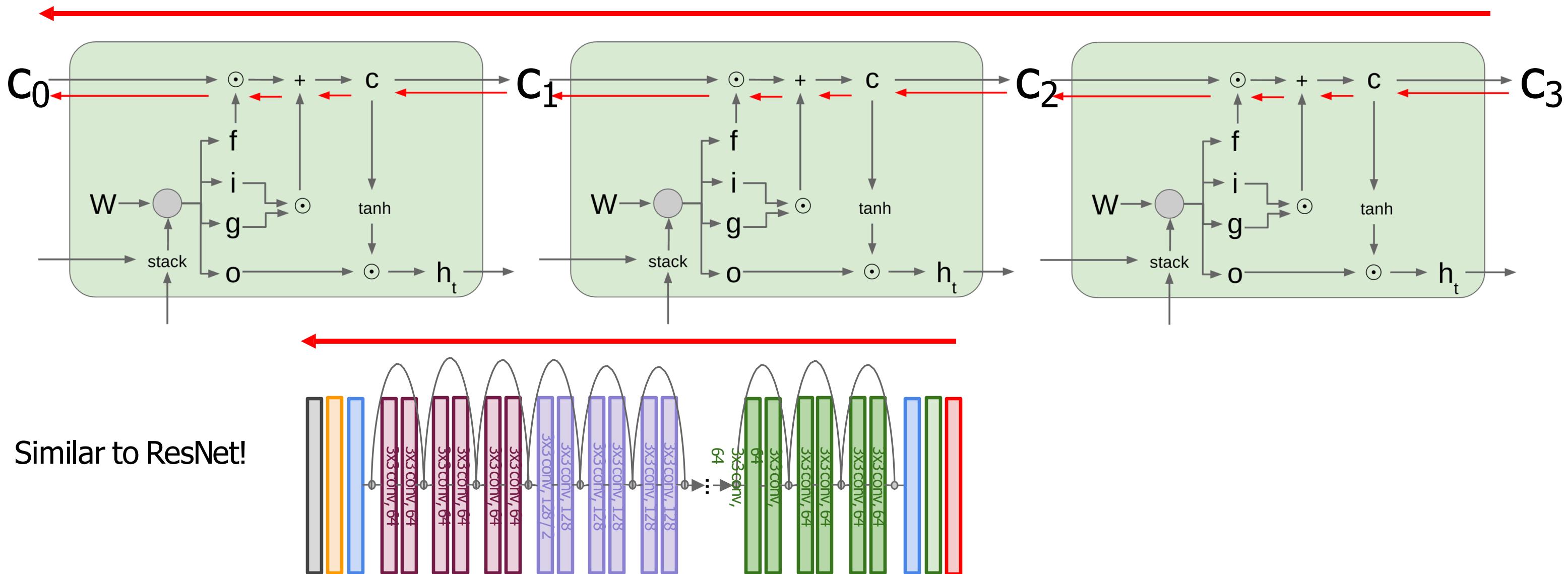
- e.g. if the $f = 1$ and the $i = 0$, then the information of that cell is preserved indefinitely.
- By contrast, it's harder for vanilla RNN to learn a recurrent weight matrix W_h that preserves info in hidden state

LSTM doesn't guarantee that there is no vanishing/exploding gradient, but it does provide an easier way for the model to learn long-distance dependencies

Long Short Term Memory (LSTM): Gradient Flow

[Hochreiter et al., 1997]

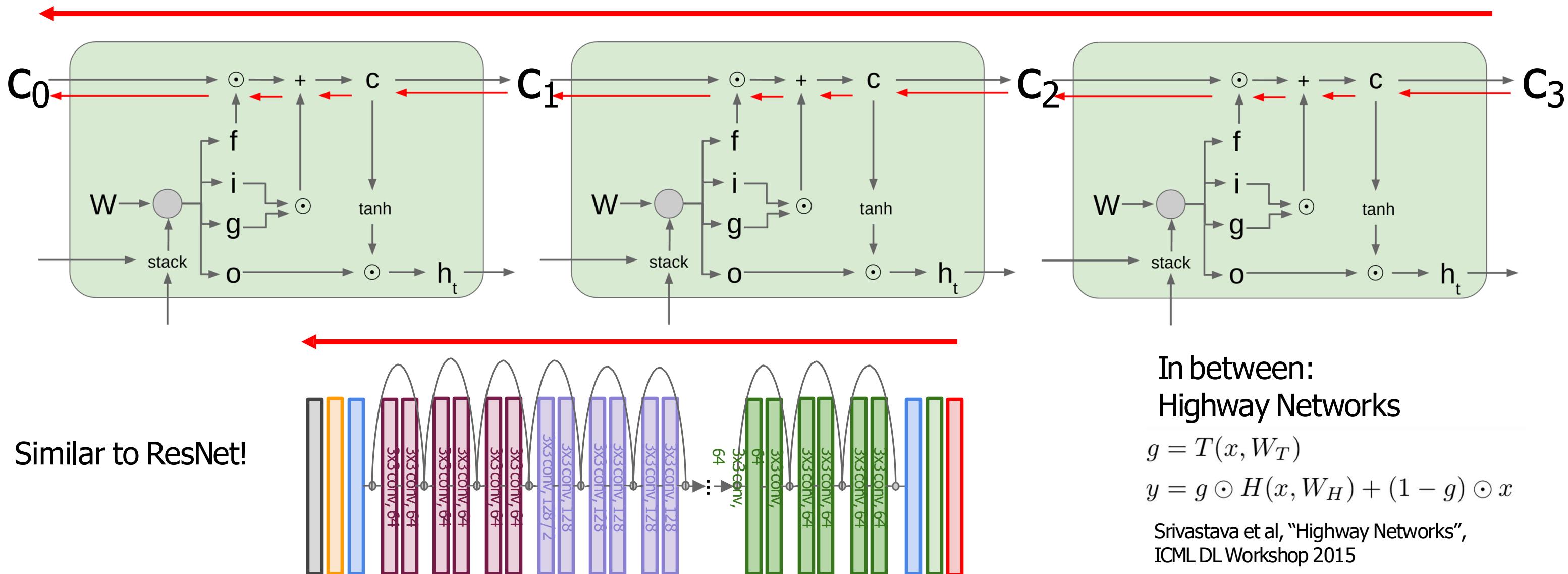
Uninterrupted gradient flow!



Long Short Term Memory (LSTM): Gradient Flow

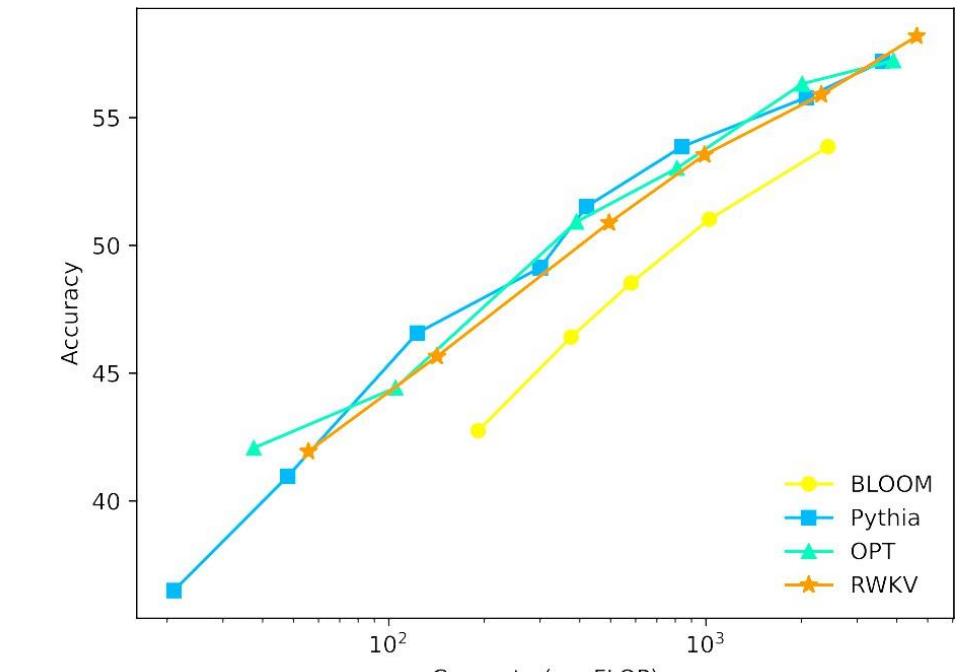
[Hochreiter et al., 1997]

Uninterrupted gradient flow!



Modern RNNs

- Sometimes called “state space models”
 - Hidden state
- Main advantages:
 - Unlimited context length
 - Compute scales linearly with sequence length



RWKV Scaling ([arXiv](#))

SIMPLIFIED STATE SPACE LAYERS FOR SEQUENCE MODELING

Mamba: Linear-Time Sequence Modeling with Selective State Spaces

Summary

- RNNs allow a lot of flexibility in architecture design
- Vanilla RNNs are simple but don't work very well
- More complex variants (e.g. LSTMs, Mamba) can introduce ways to selectively pass information forward
- Backward flow of gradients in RNN can explode or vanish. Exploding is controlled with gradient clipping. Backpropagation through time is often needed.
- Better/simpler architectures are a hot topic of current research, as well as new paradigms for reasoning over sequences

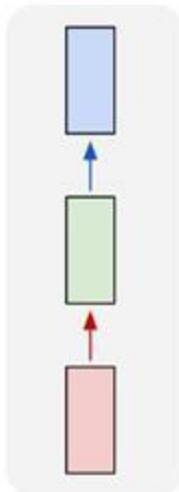
Next time: Attention and Transformers



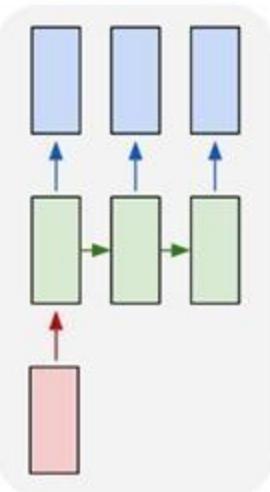
Lecture 8: Attention and Transformers

Last Time: Recurrent Neural Networks

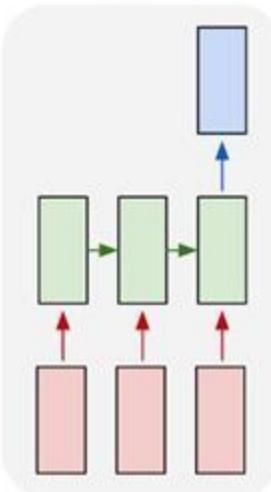
one to one



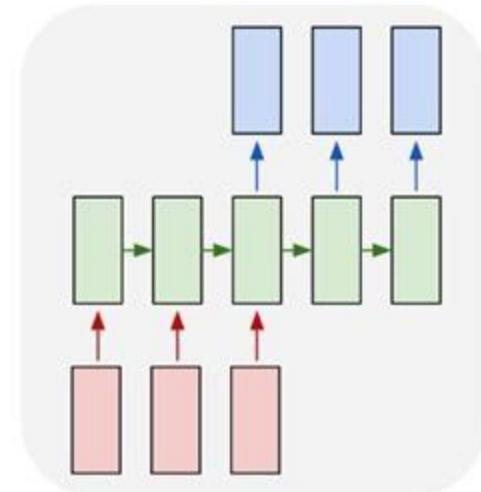
one to many



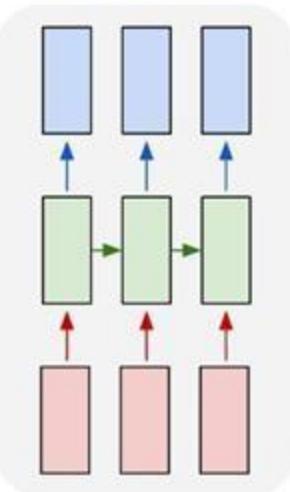
many to one



many to many

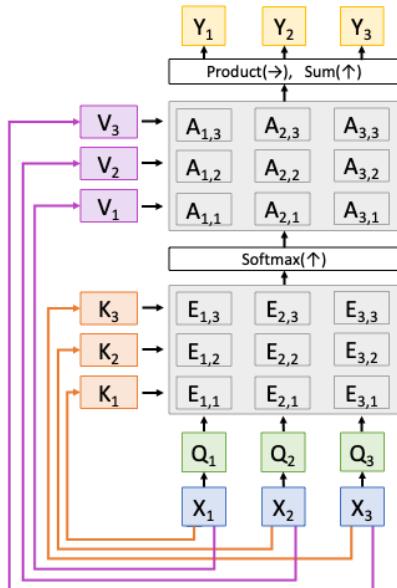


many to many

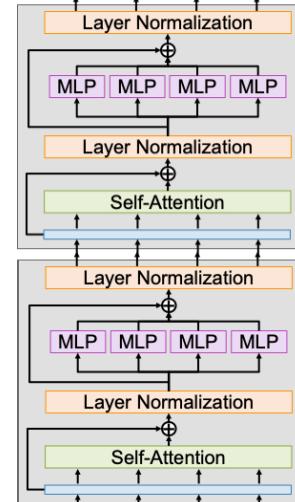


Today: Attention +Transformers

Attention: A new primitive that operates on sets of vectors

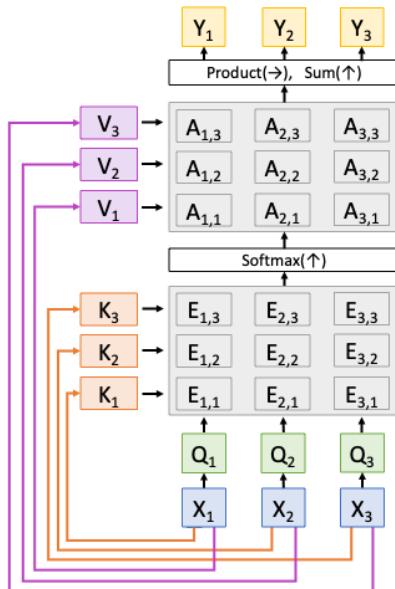


Transformer: A neural network architecture that uses attention everywhere



Today: Attention +Transformers

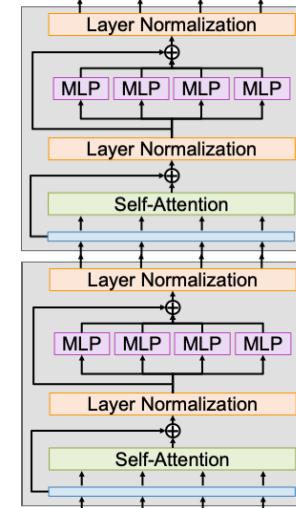
Attention: A new primitive that operates on sets of vectors



Transformers are used everywhere today!

But they developed as an offshoot of RNNs so let's start there

Transformer: A neural network architecture that uses attention everywhere



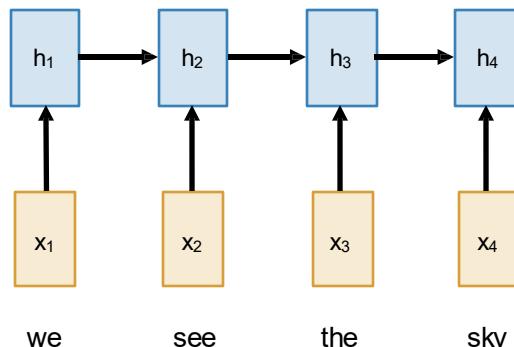
Sequence to Sequence with RNNs: Encoder - Decoder

Input: Sequence x_1, \dots, x_T

Output: Sequence $y_1, \dots, y_{T'}$

A motivating example for today's discussion – machine translation! English → Italian

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

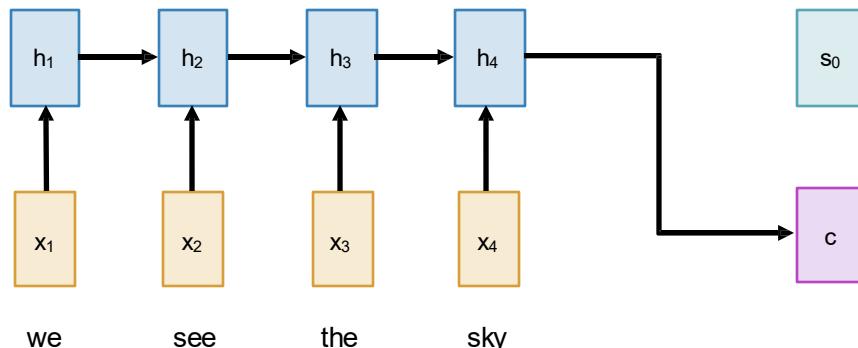


Sequence to Sequence with RNNs

Input: Sequence x_1, \dots, x_T

Output: Sequence $y_1, \dots, y_{T'}$

From final hidden state predict:
Encoder: $h_t = f_W(x_t, h_{t-1})$ **Initial decoder state** s_0
Context vector c (often $c=h_T$)



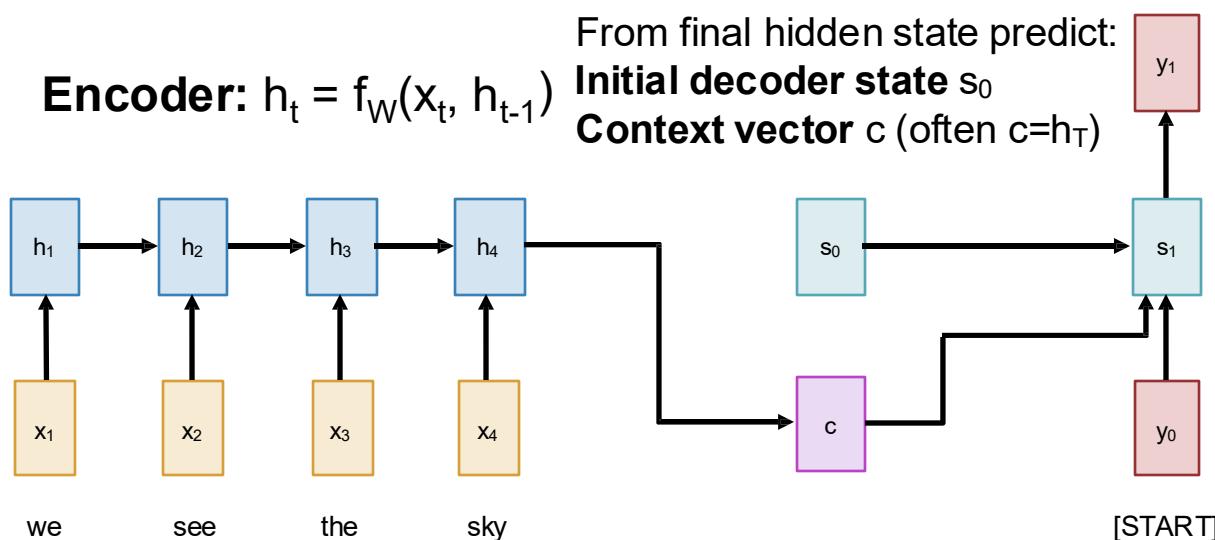
Sequence to Sequence with RNNs

Input: Sequence x_1, \dots, x_T

Output: Sequence $y_1, \dots, y_{T'}$

Decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c)$

vediamo



Sequence to Sequence with RNNs

Input: Sequence x_1, \dots, x_T

Output: Sequence $y_1, \dots, y_{T'}$

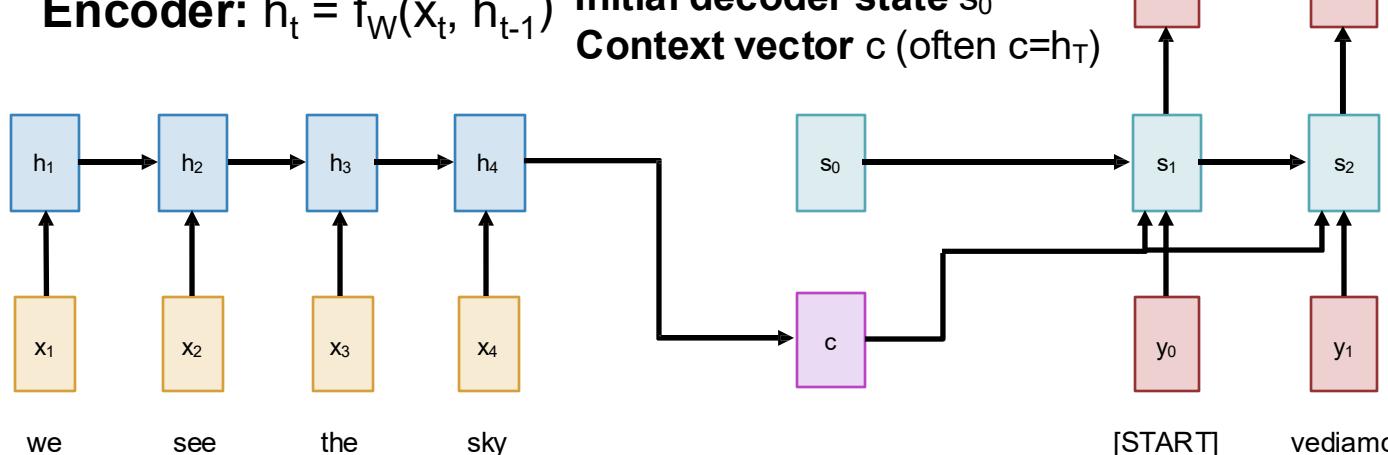
Decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c)$

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

From final hidden state predict:

Initial decoder state s_0

Context vector c (often $c=h_T$)



Sequence to Sequence with RNNs

Input: Sequence x_1, \dots, x_T

Output: Sequence $y_1, \dots, y_{T'}$

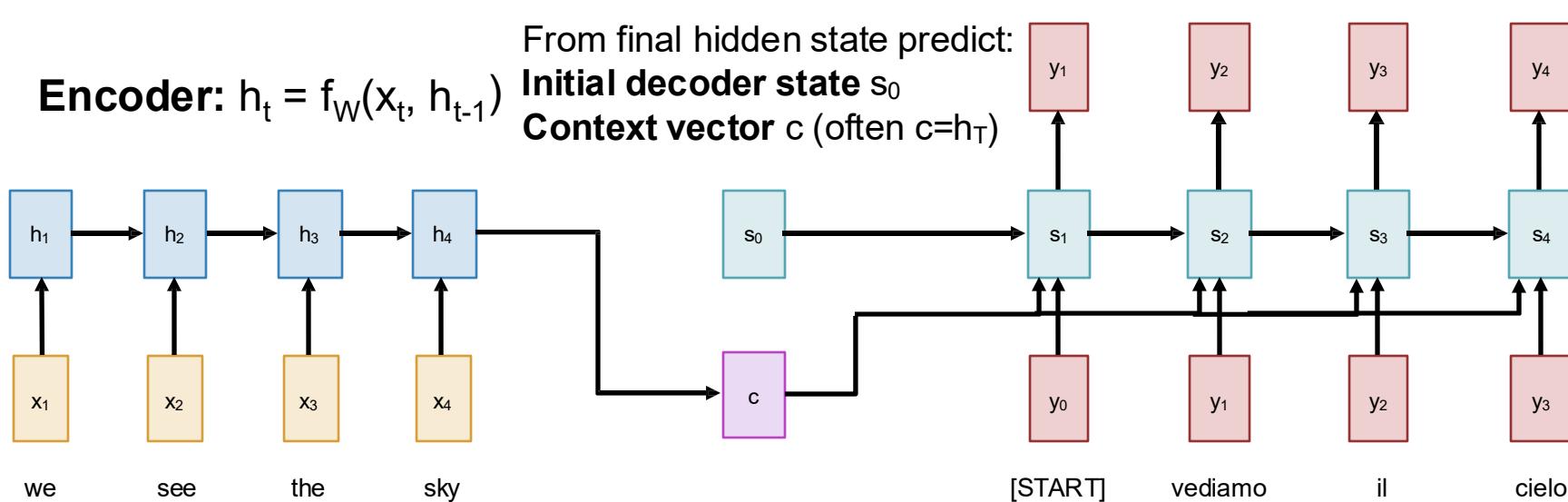
Decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c)$

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

From final hidden state predict:

Initial decoder state s_0

Context vector c (often $c=h_T$)



Sequence to Sequence with RNNs

Input: Sequence x_1, \dots, x_T

Output: Sequence $y_1, \dots, y_{T'}$

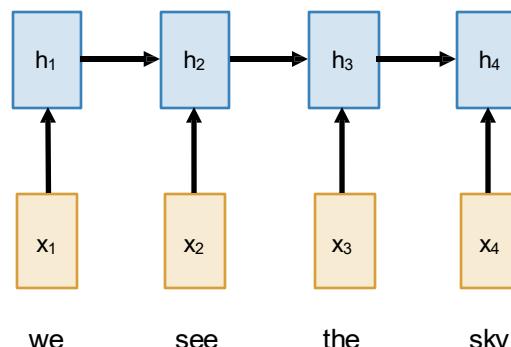
Decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c)$

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

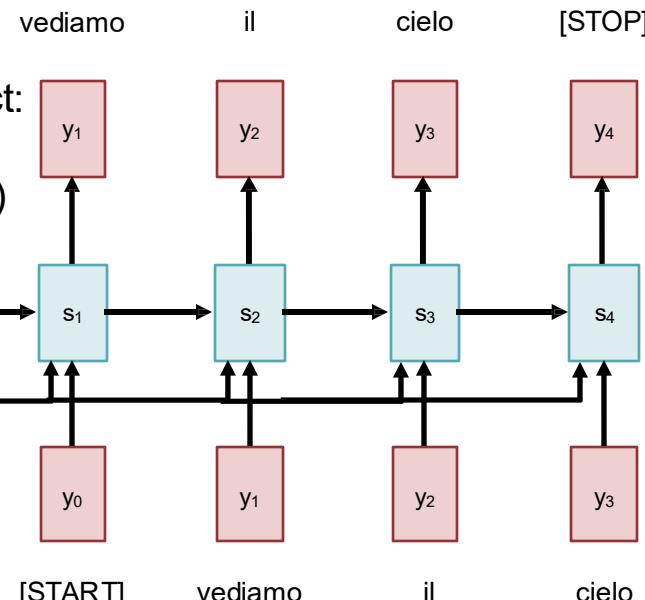
From final hidden state predict:

Initial decoder state s_0

Context vector c (often $c=h_T$)



Problem: Input sequence bottlenecks through fixed sized c . What if $T=1000$?



Sequence to Sequence with RNNs

Input: Sequence x_1, \dots, x_T

Output: Sequence $y_1, \dots, y_{T'}$

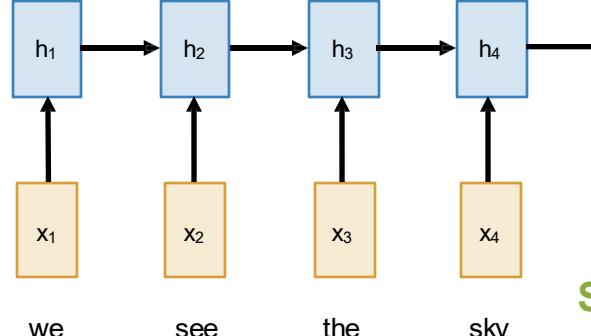
Decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c)$

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

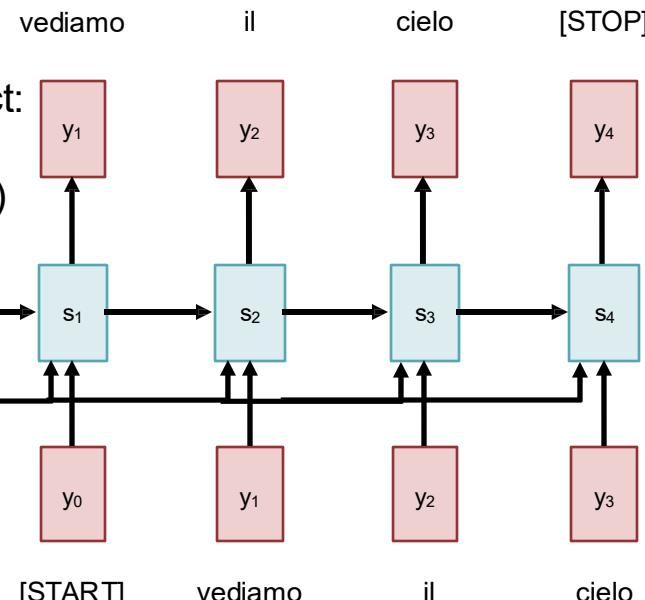
From final hidden state predict:

Initial decoder state s_0

Context vector c (often $c=h_T$)



Solution: Look back at the whole input sequence on each step of the output

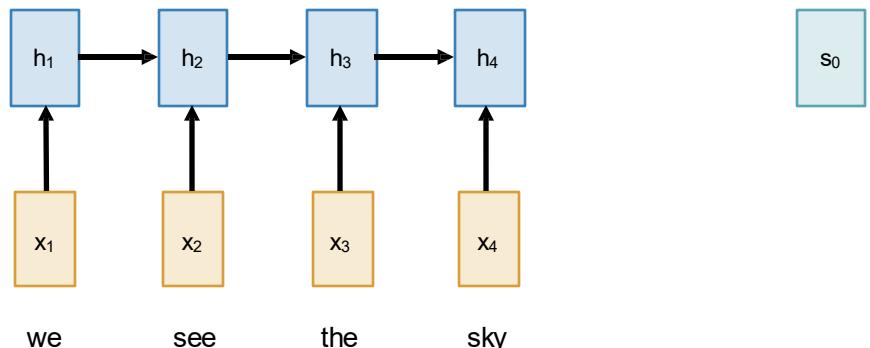


Sequence to Sequence with RNNs and Attention

Input: Sequence x_1, \dots, x_T

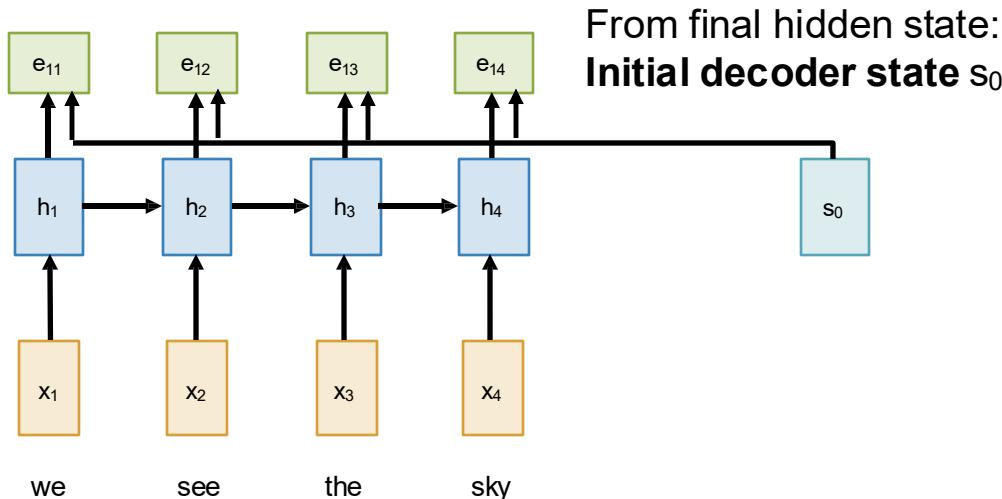
Output: Sequence $y_1, \dots, y_{T'}$

Encoder: $h_t = f_W(x_t, h_{t-1})$ From final hidden state:
Initial decoder state s_0

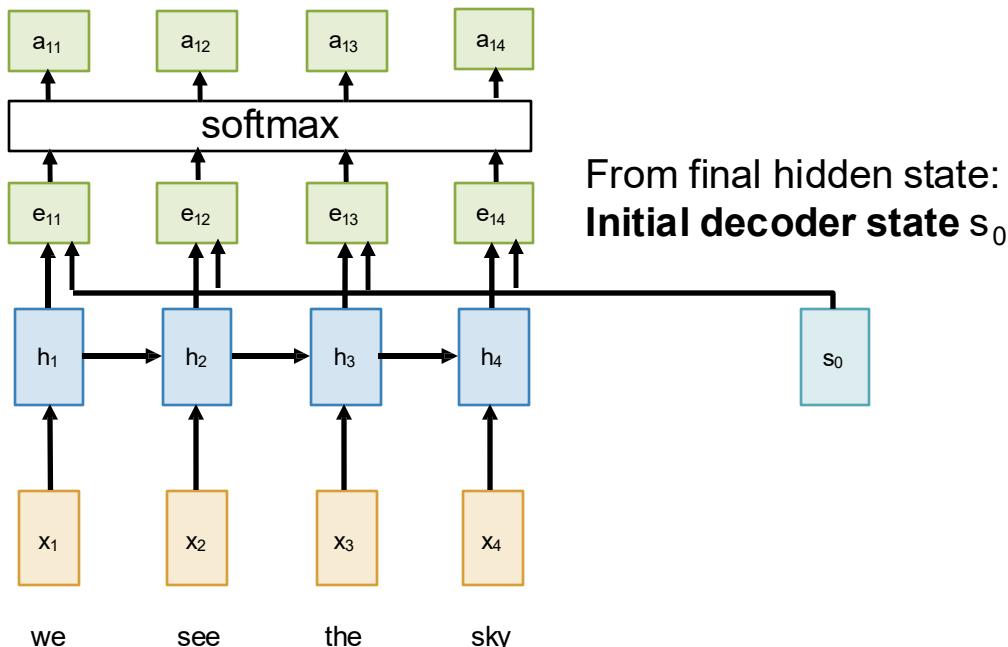


Sequence to Sequence with RNNs and Attention

Compute (scalar) **alignment scores**
 $e_{t,i} = f_{att}(s_{t-1}, h_i)$ (f_{att} is a Linear Layer)



Sequence to Sequence with RNNs and Attention

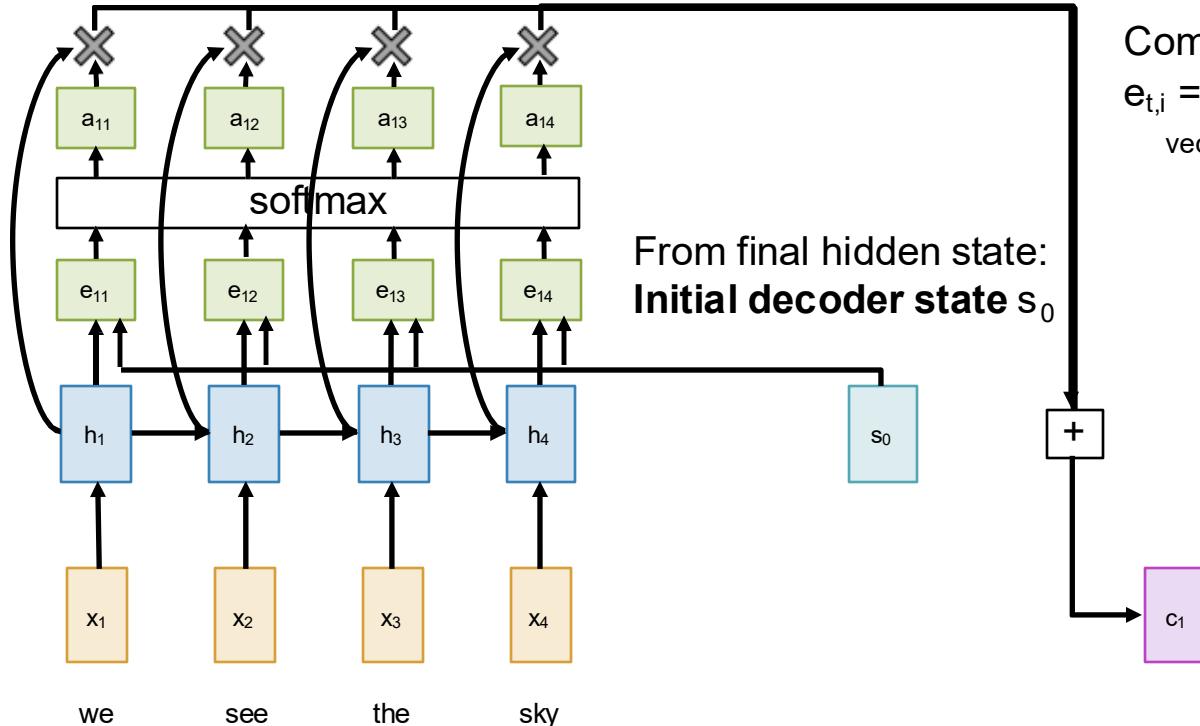


From final hidden state:
Initial decoder state s_0

Compute (scalar) **alignment scores**
 $e_{t,i} = f_{att}(s_{t-1}, h_i)$ (f_{att} is a Linear Layer)

Normalize alignment scores
to get **attention weights**
 $0 < a_{t,i} < 1 \quad \sum_i a_{t,i} = 1$

Sequence to Sequence with RNNs and Attention



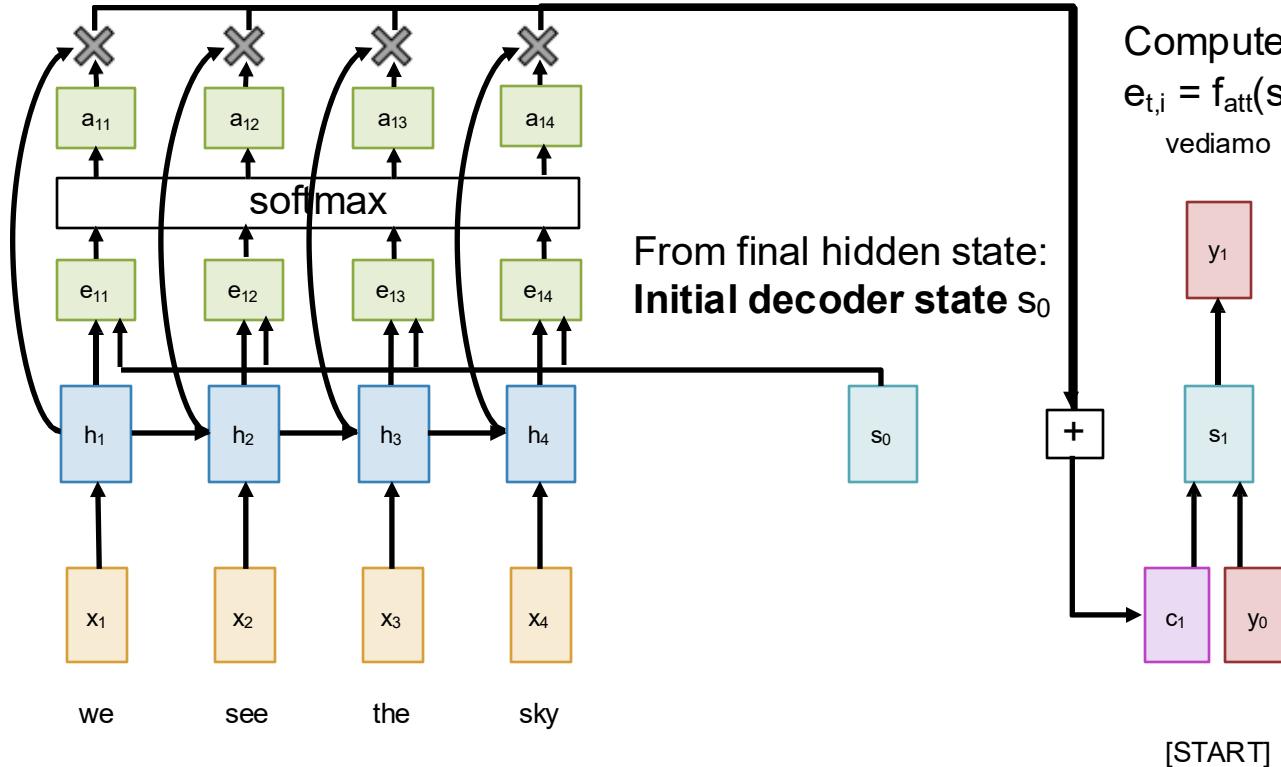
Compute (scalar) **alignment scores**
 $e_{t,i} = f_{att}(s_{t-1}, h_i)$ (f_{att} is a Linear Layer)
vediamo

Normalize alignment scores
to get **attention weights**
 $0 < a_{t,i} < 1 \quad \sum_i a_{t,i} = 1$

Compute context vector as
weighted sum of hidden states

$$c_t = \sum_i a_{t,i} h_i$$

Sequence to Sequence with RNNs and Attention



Compute (scalar) **alignment scores**
 $e_{t,i} = f_{att}(s_{t-1}, h_i)$ (f_{att} is a Linear Layer)

vediamo

Normalize alignment scores
to get **attention weights**
 $0 < a_{t,i} < 1 \quad \sum_i a_{t,i} = 1$

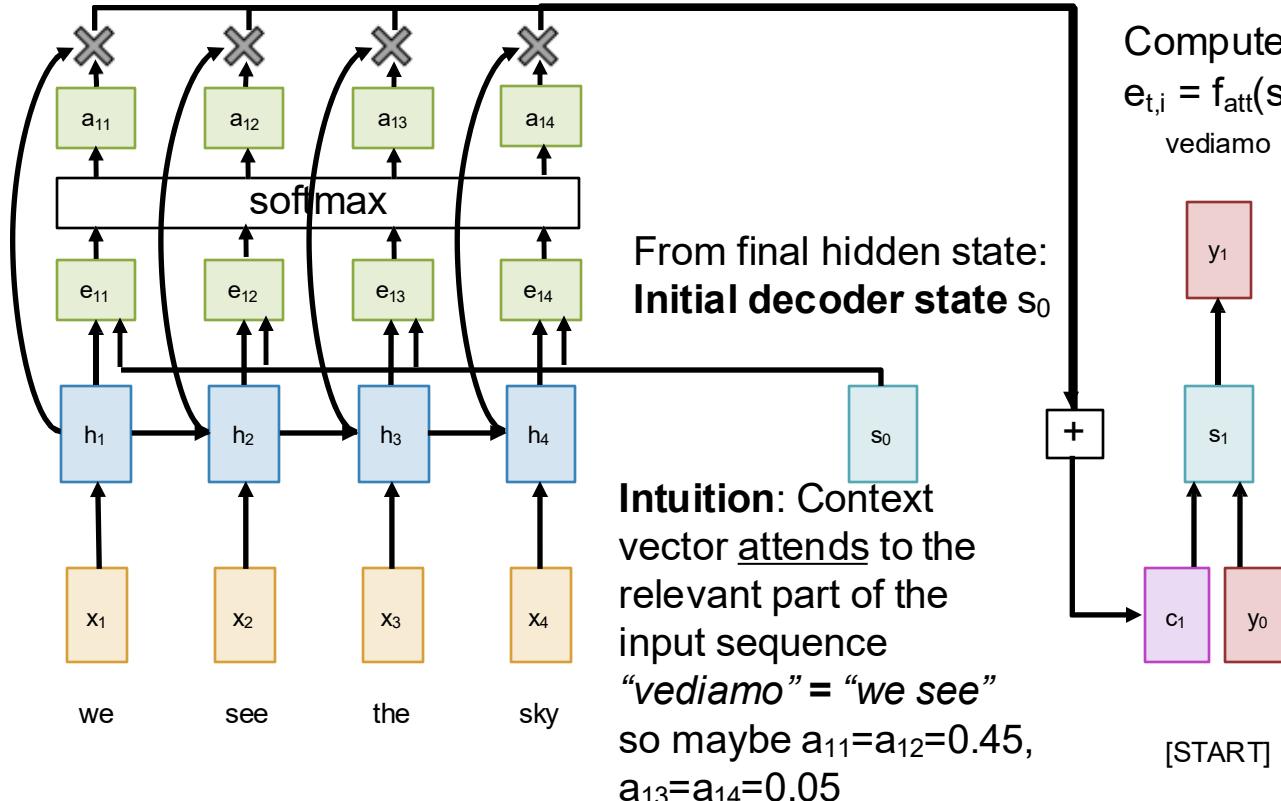
Compute **context vector** as
weighted sum of hidden
states

$$c_t = \sum_i a_{t,i} h_i$$

Use context vector in
decoder: $s_t = g_u(y_{t-1}, s_{t-1}, c_t)$

g_u is an RNN unit
(e.g. LSTM, GRU)

Sequence to Sequence with RNNs and Attention



Compute (scalar) **alignment scores**
 $e_{t,i} = f_{\text{att}}(s_{t-1}, h_i)$ (f_{att} is a Linear Layer)

vediamo

Normalize alignment scores to get **attention weights**
 $0 < a_{t,i} < 1 \quad \sum_j a_{t,i} = 1$

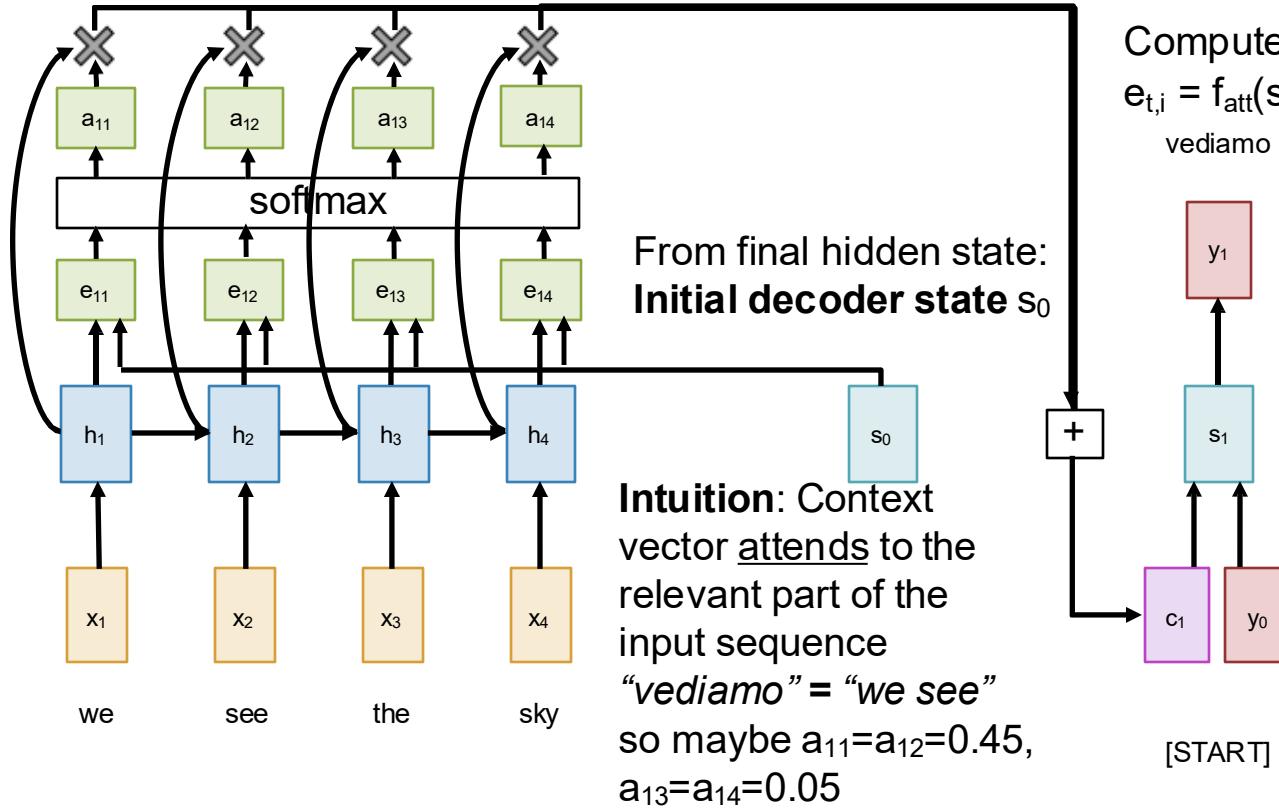
Compute **context vector** as weighted sum of hidden states

$$c_t = \sum_i a_{t,i} h_i$$

Use context vector in decoder: $s_t = g_u(y_{t-1}, s_{t-1}, c_t)$

[START]

Sequence to Sequence with RNNs and Attention



Compute (scalar) **alignment scores**
 $e_{t,i} = f_{\text{att}}(s_{t-1}, h_i)$ (f_{att} is a Linear Layer)

vediamo

Normalize alignment scores to get **attention weights**
 $0 < a_{t,i} < 1 \quad \sum_j a_{t,i} = 1$

Compute **context vector** as weighted sum of hidden states

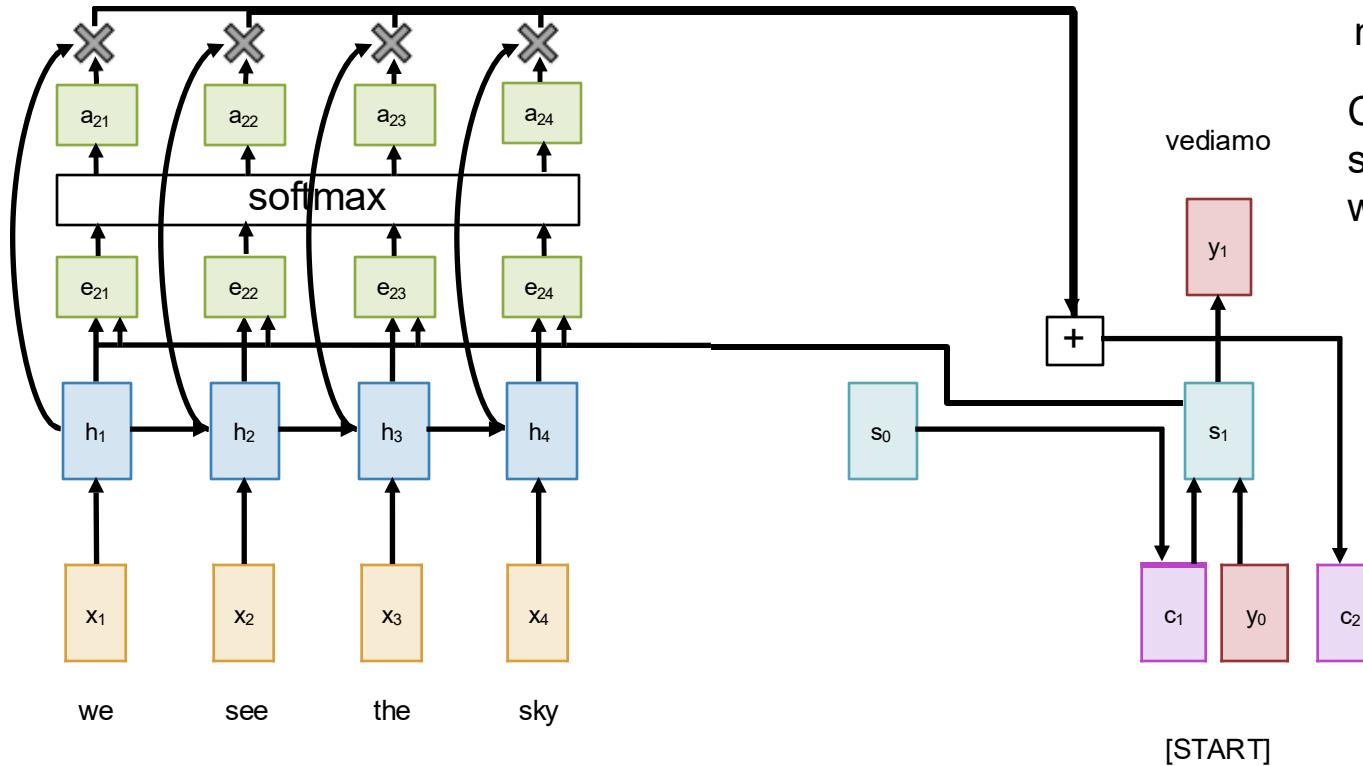
$$c_t = \sum_i a_{t,i} h_i$$

Use context vector in decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c_t)$

[START]

All differentiable! No supervision on attention weights. Backprop through everything

Sequence to Sequence with RNNs and Attention

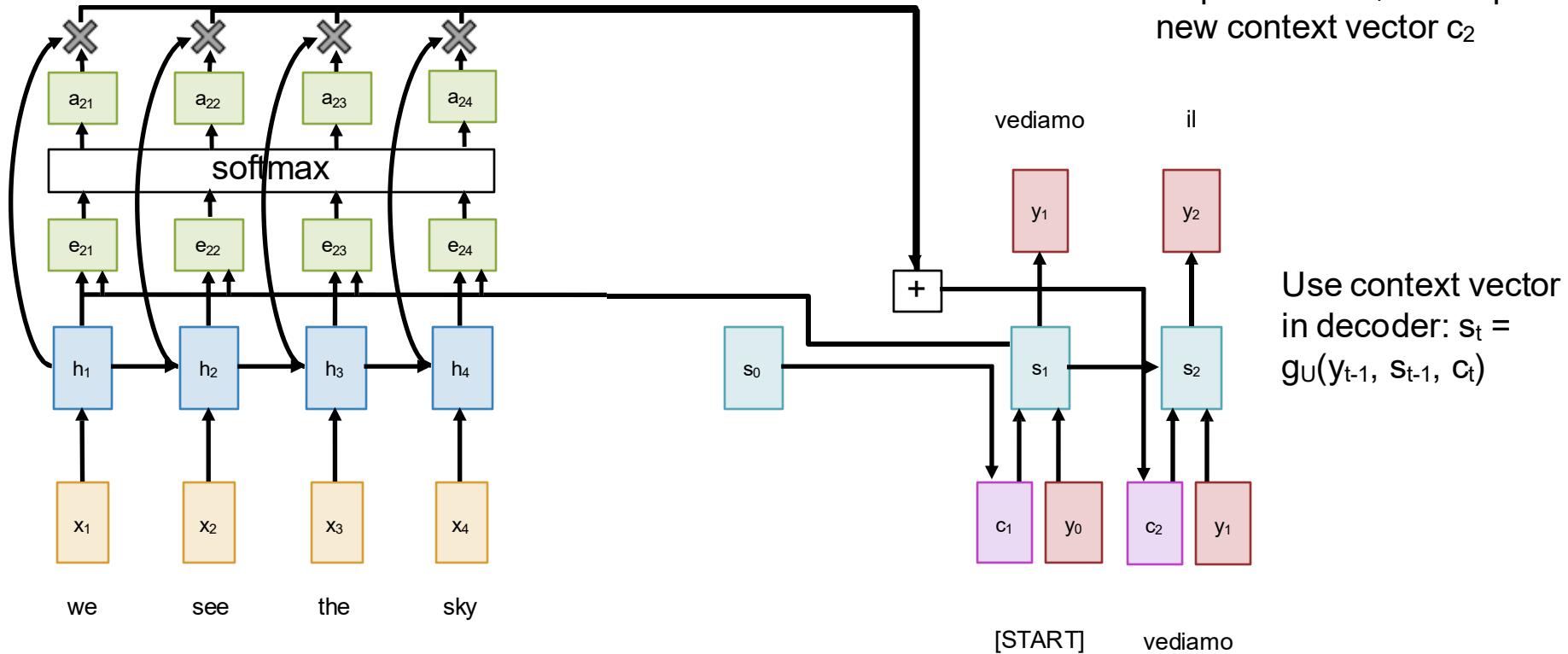


Repeat: Use s_1 to compute new context vector c_2

Compute new alignment scores $e_{2,i}$ and attention weights $a_{2,i}$

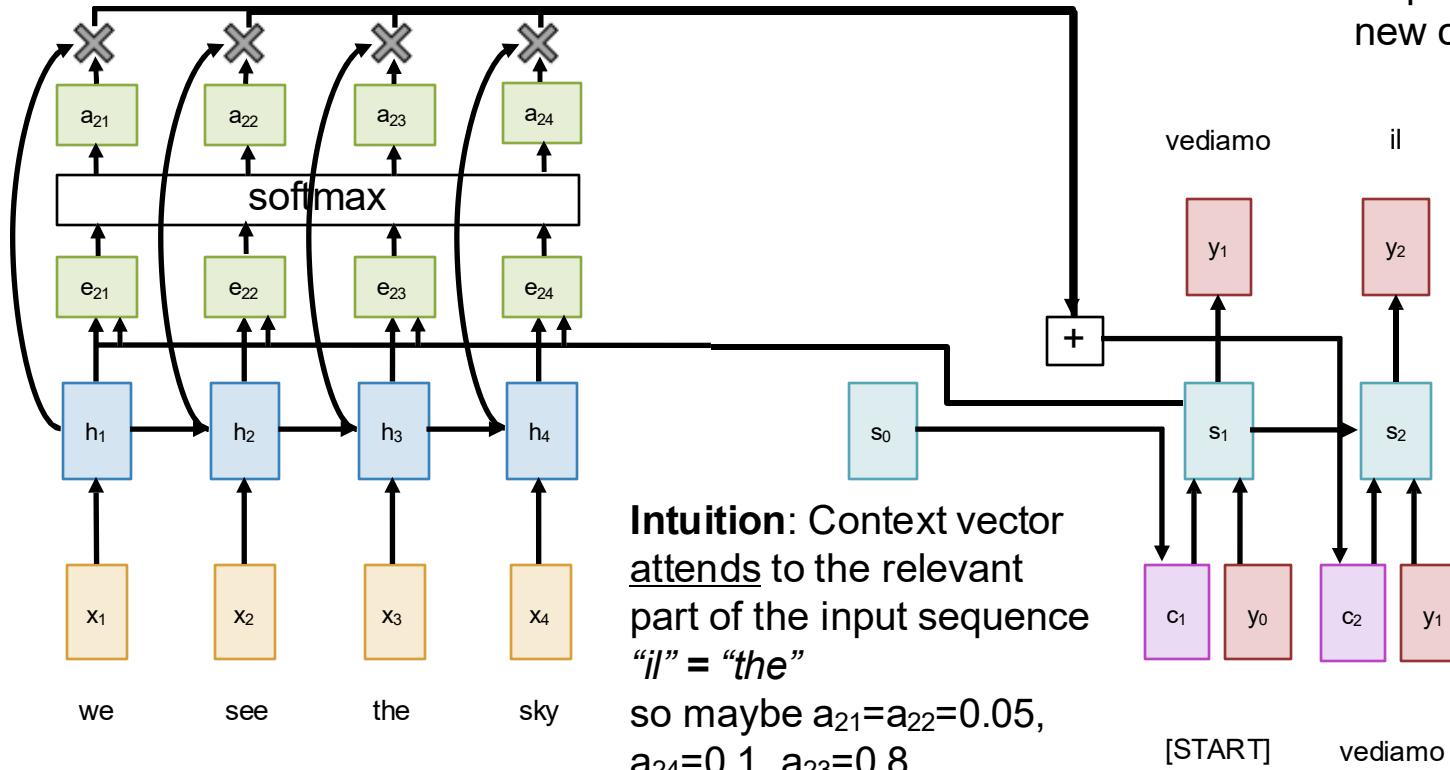
Sequence to Sequence with RNNs and Attention

Repeat: Use s_1 to compute new context vector c_2



Sequence to Sequence with RNNs and Attention

Repeat: Use s_1 to compute new context vector c_2

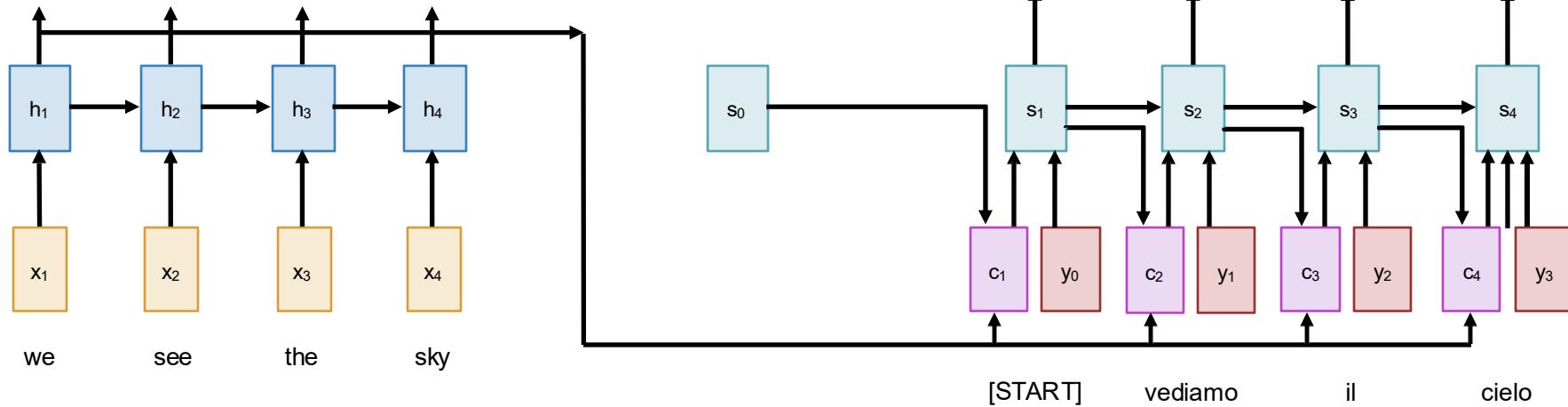


Use context vector in decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c_t)$

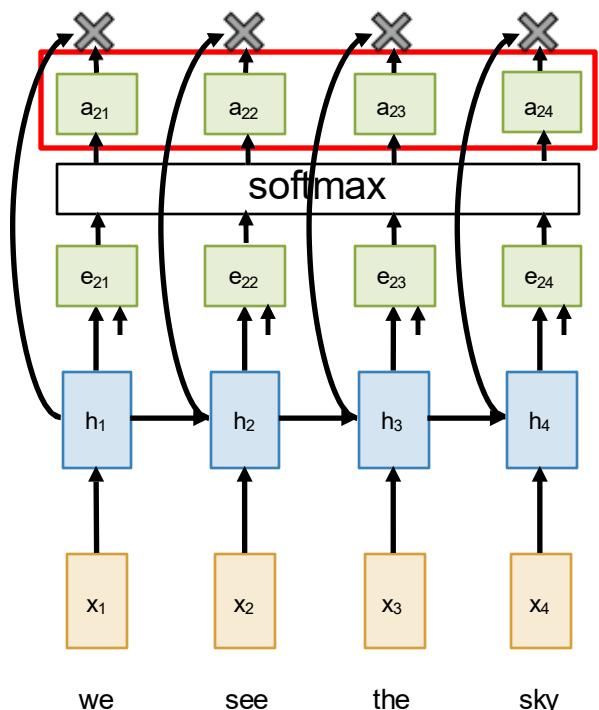
Sequence to Sequence with RNNs and Attention

Use a different context vector in each timestep of decoder

- Input sequence not bottlenecked through single vector
- At each timestep of decoder, context vector “looks at” different parts of the input sequence

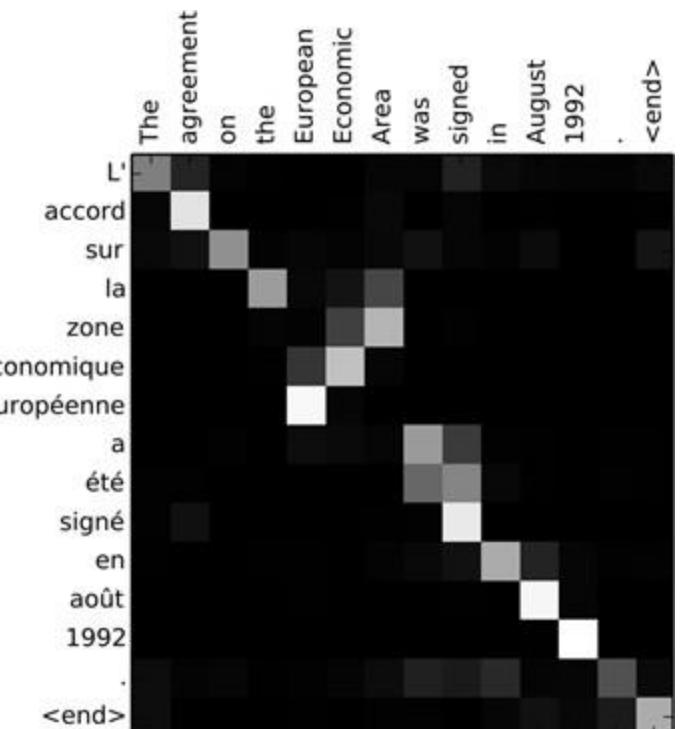


Sequence to Sequence with RNNs and Attention



Example: English to French translation

Visualize attention weights $a_{t,i}$



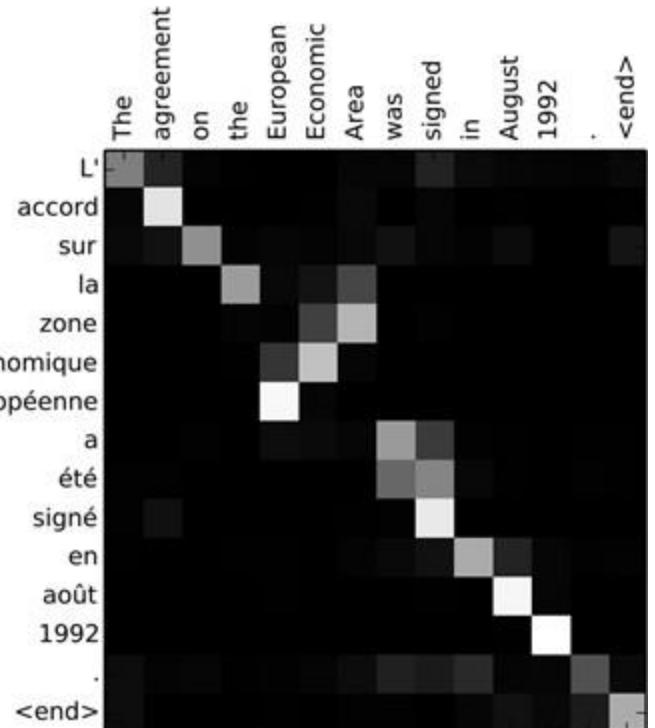
Sequence to Sequence with RNNs and Attention

Example: English to French translation

Input: “The agreement on the European Economic Area was signed in August 1992.”

Output: “L'accord sur la zone économique européenne a été signé en août 1992.”

Visualize attention weights $a_{t,i}$



Sequence to Sequence with RNNs and Attention

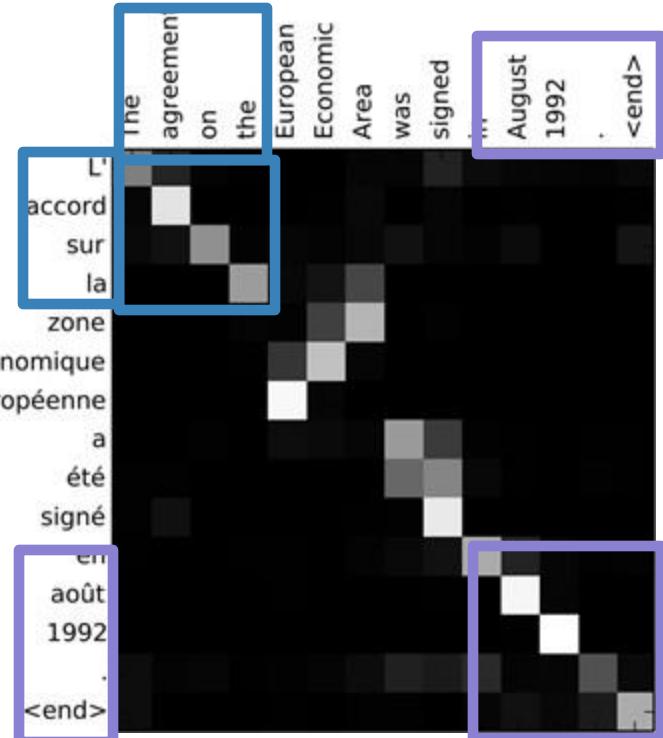
Example: English to French translation

Input: “**The agreement on the European Economic Area was signed in August 1992.**”

Output: “**L'accord sur la zone économique européenne a été signé en août 1992.**”

Diagonal attention means words correspond in order

Visualize attention weights $a_{t,i}$



Diagonal attention means words correspond in order

Sequence to Sequence with RNNs and Attention

Input: “The agreement on the European Economic Area was signed in August 1992.”

Output: “L'accord sur la zone économique européenne a été signé en août 1992.”

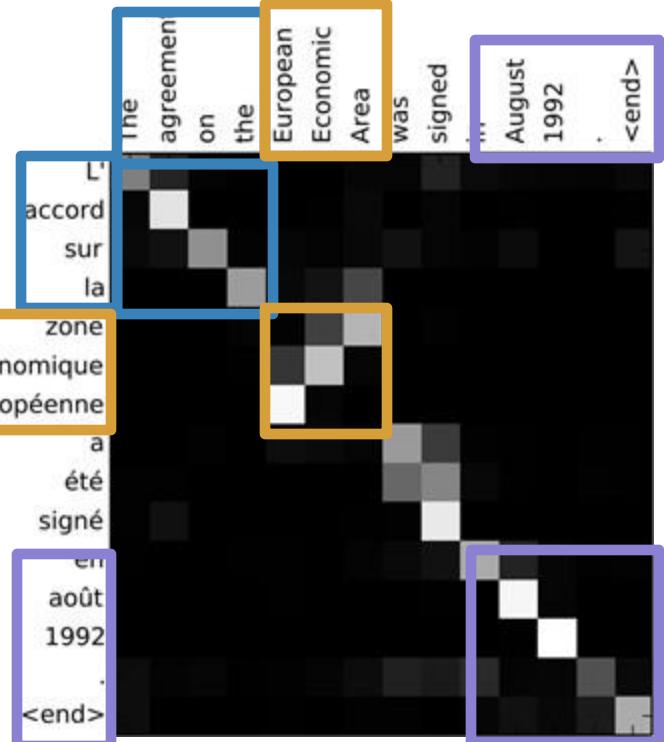
Example: English to French translation

Diagonal attention means words correspond in order

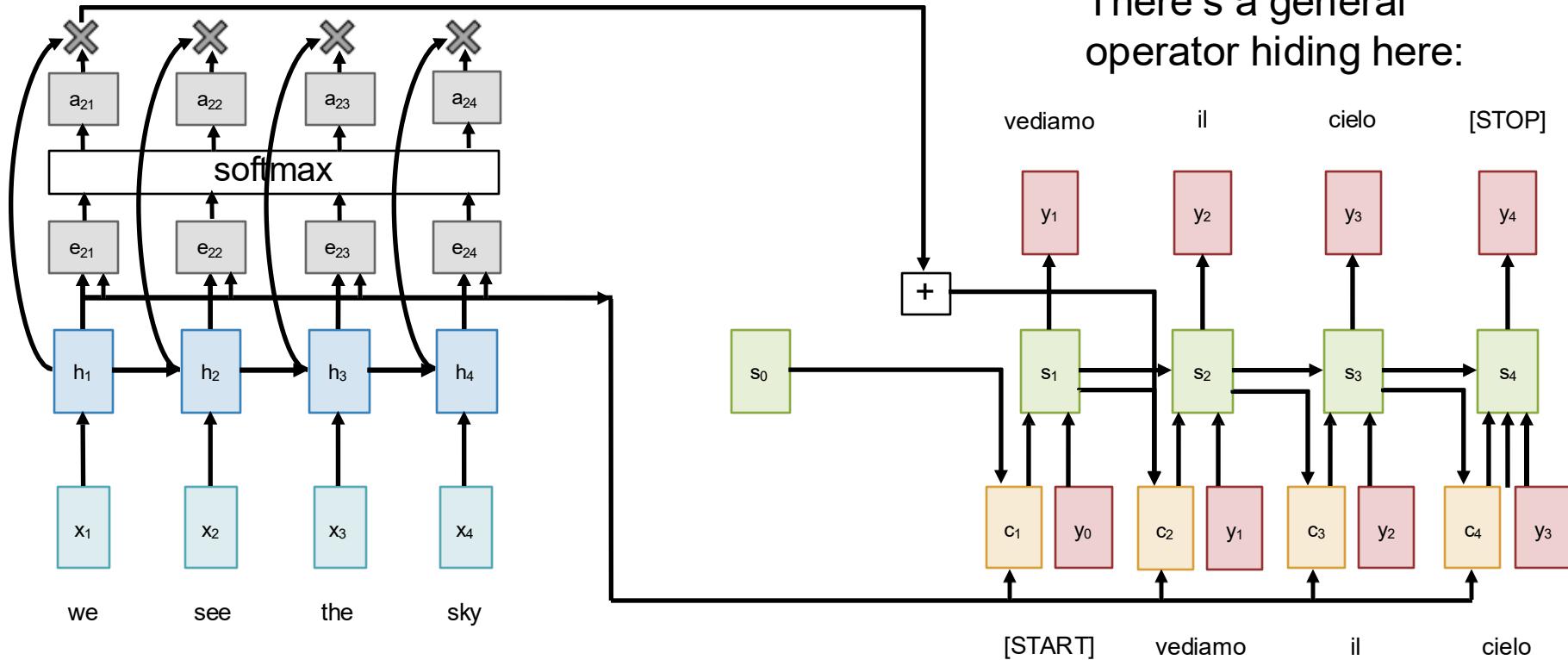
Attention figures out other word orders

Diagonal attention means words correspond in order

Visualize attention weights $a_{t,i}$



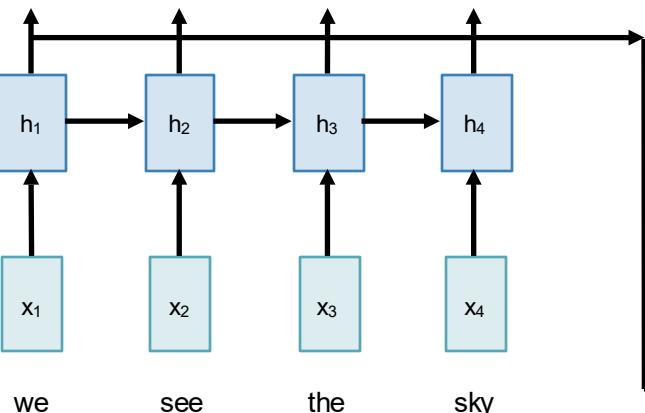
Sequence to Sequence with RNNs and Attention



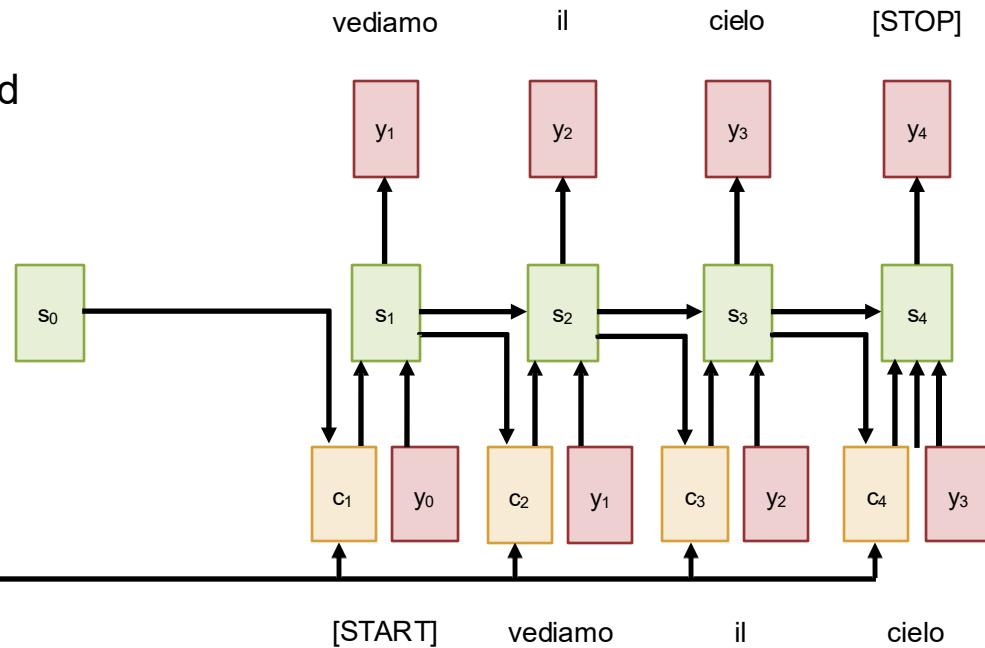
Sequence to Sequence with RNNs and Attention

Query vectors (decoder RNN states) and data vectors (encoder RNN states) get transformed to output vectors (Context states).

Each query attends to all data vectors and gives one output vector



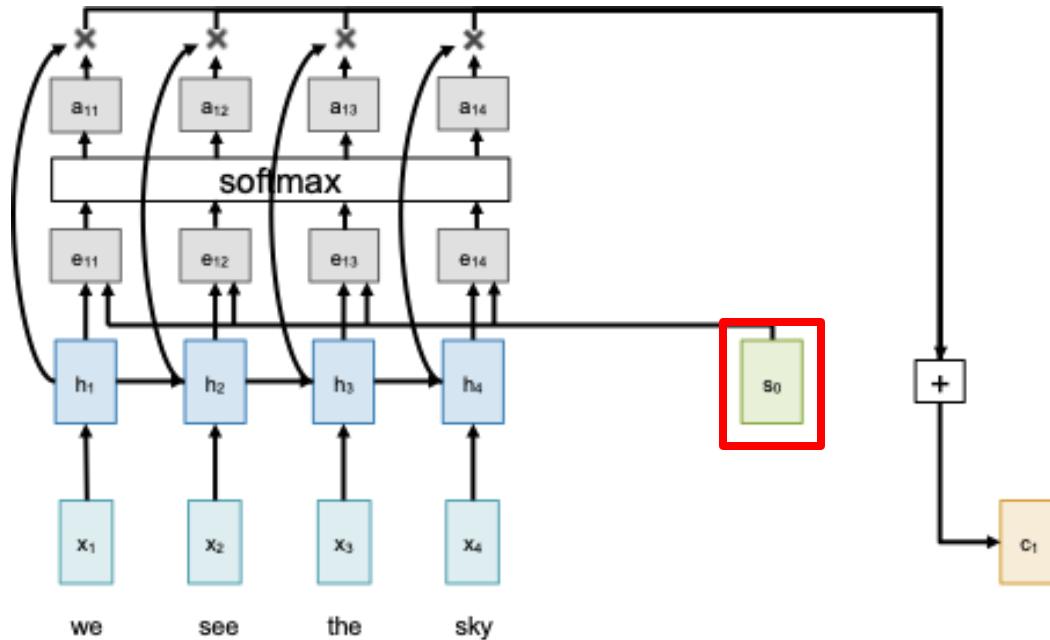
There's a general operator hiding here:



Attention Layer

Inputs:

Query vector: q [D_Q]

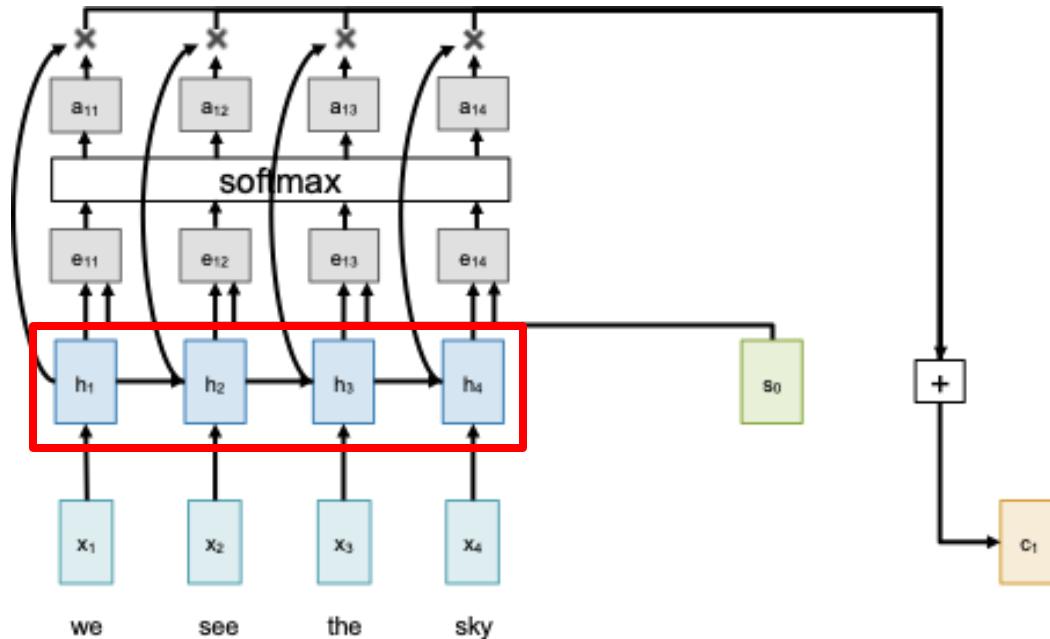


Attention Layer

Inputs:

Query vector: q [D_Q]

Data vectors: X [$N_x \times D_Q$]

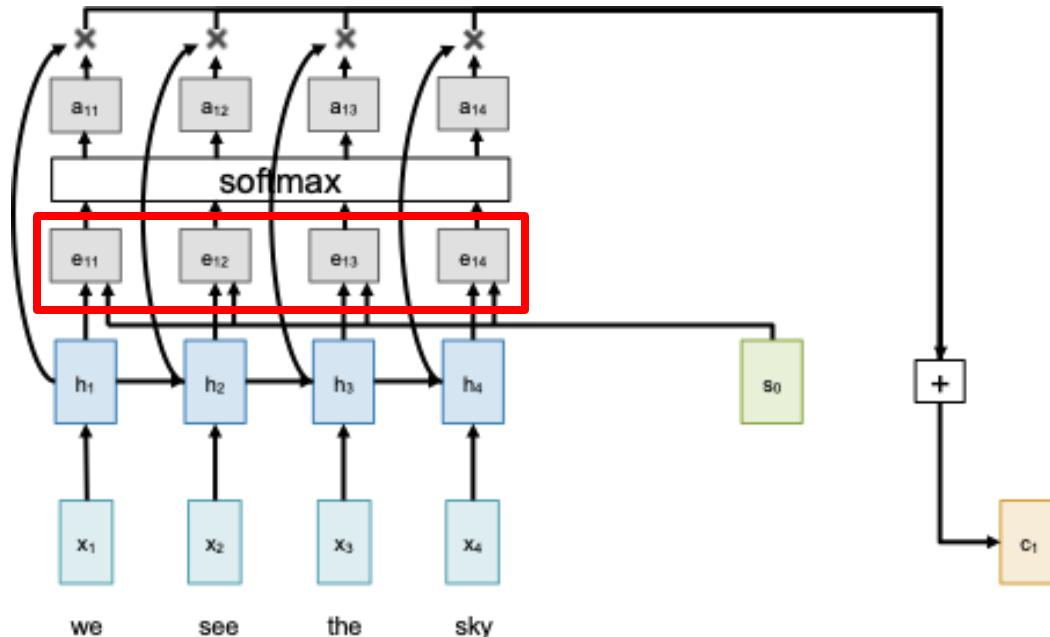


Attention Layer

Inputs:

Query vector: q [D_Q]

Data vectors: X [$N_X \times D_Q$]



Computation:

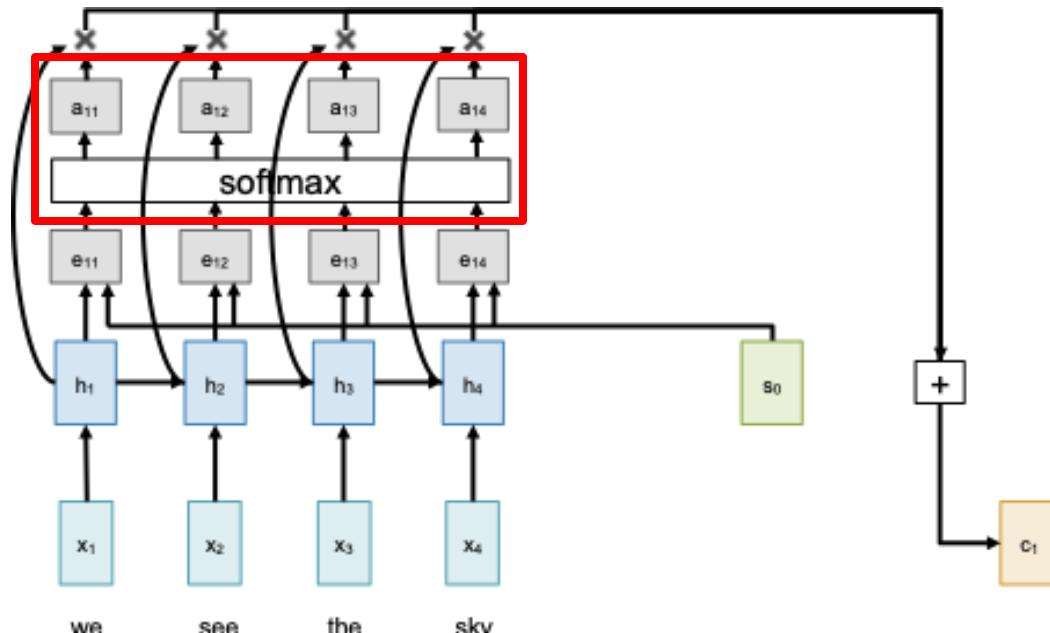
Similarities: e [N_X] $e_i = f_{att}(q, X_i)$

Attention Layer

Inputs:

Query vector: q [D_Q]

Data vectors: X [$N_X \times D_Q$]



Computation:

Similarities: e [N_X] $e_i = f_{att}(q, X_i)$

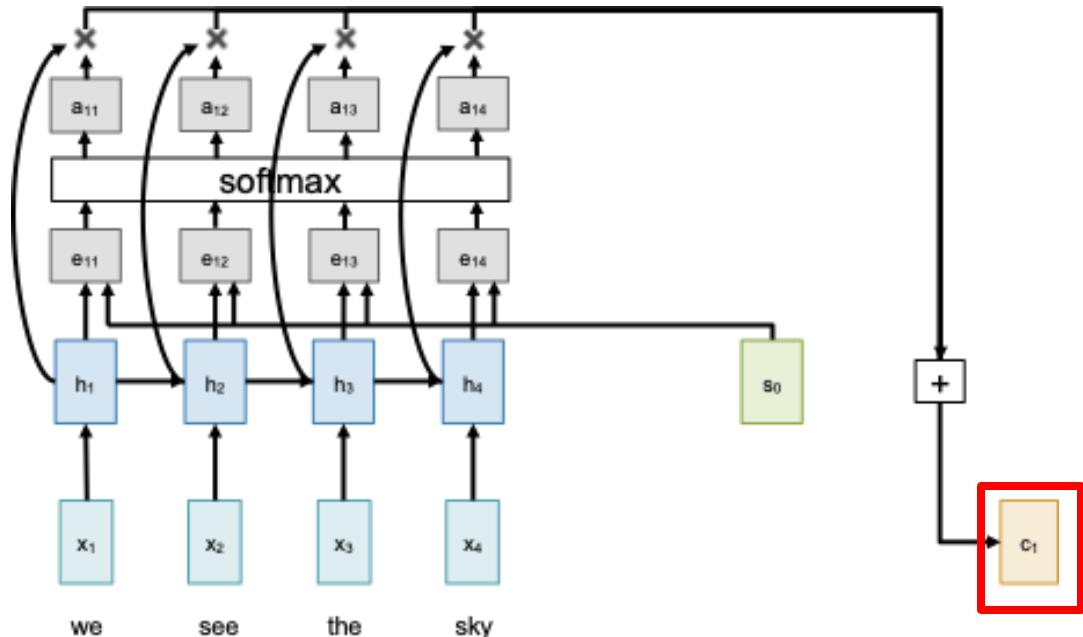
Attention weights: $a = \text{softmax}(e)$ [N_X]

Attention Layer

Inputs:

Query vector: q [D_Q]

Data vectors: X [$N_x \times D_Q$]



Computation:

Similarities: e [N_x] $e_i = f_{att}(q, X_i)$

Attention weights: $a = \text{softmax}(e)$ [N_x]

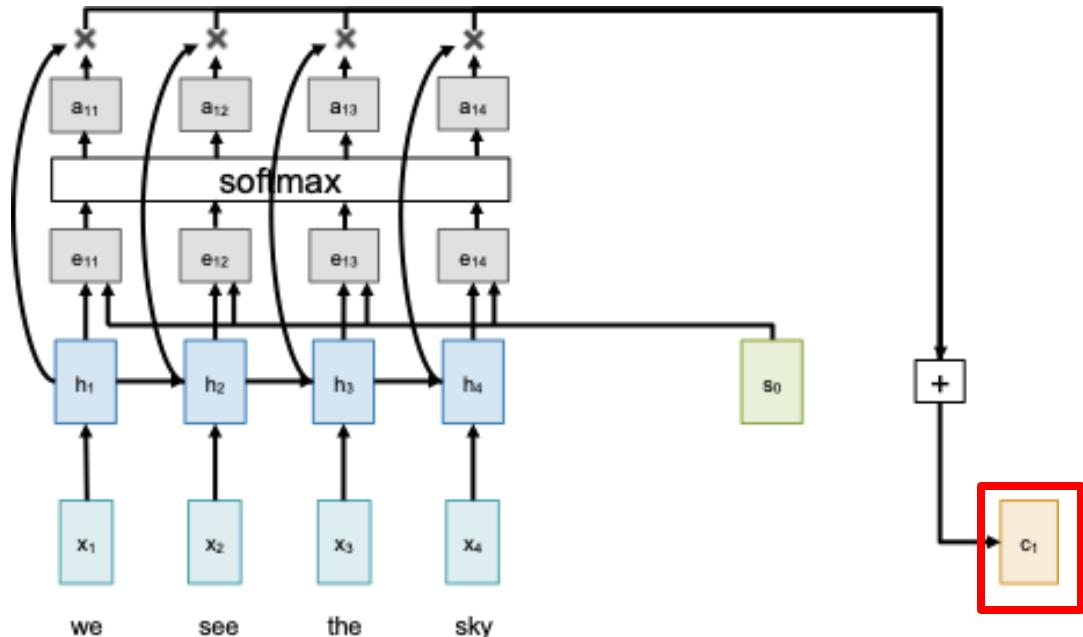
Output vector: $y = \sum_i a_i X_i$ [D_x]

Attention Layer

Inputs:

Query vector: q [D_Q]

Data vectors: X [$N_x \times D_Q$]



Computation:

Similarities: e [N_x] $e_i = f_{att}(q, X_i)$

Attention weights: $a = \text{softmax}(e)$ [N_x]

Output vector: $y = \sum_i a_i X_i$ [D_x]

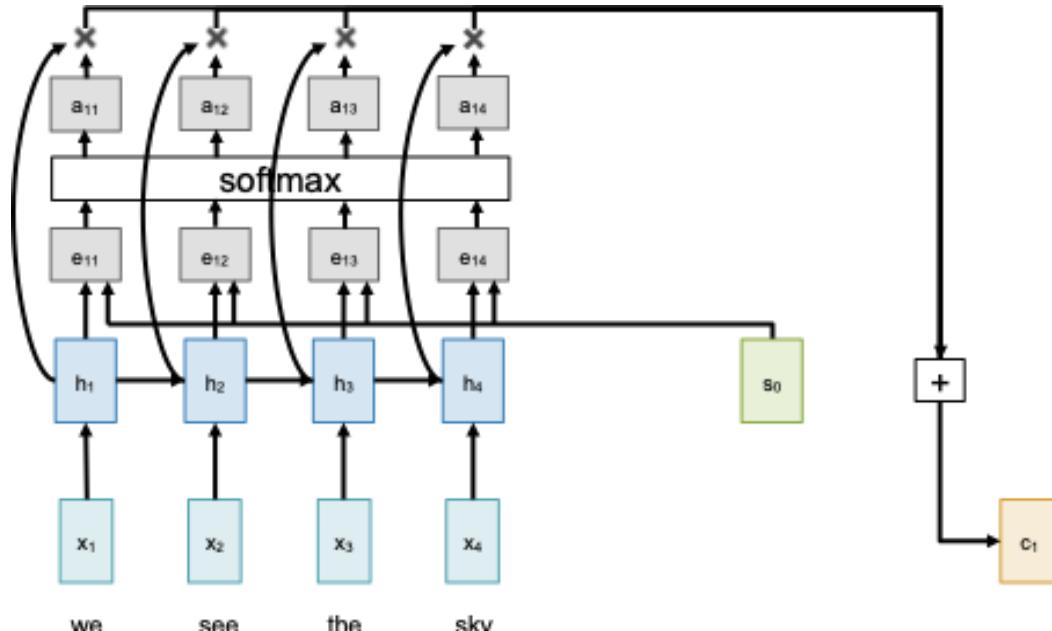
Let's generalize this!

Attention Layer

Inputs:

Query vector: q [D_Q]

Data vectors: X [$N_x \times D_Q$]



Computation:

Similarities: $e_i = q \cdot X_i$

Attention weights: $a = \text{softmax}(e)$ [N_x]

Output vector: $y = \sum_i a_i X_i$ [D_x]

Changes

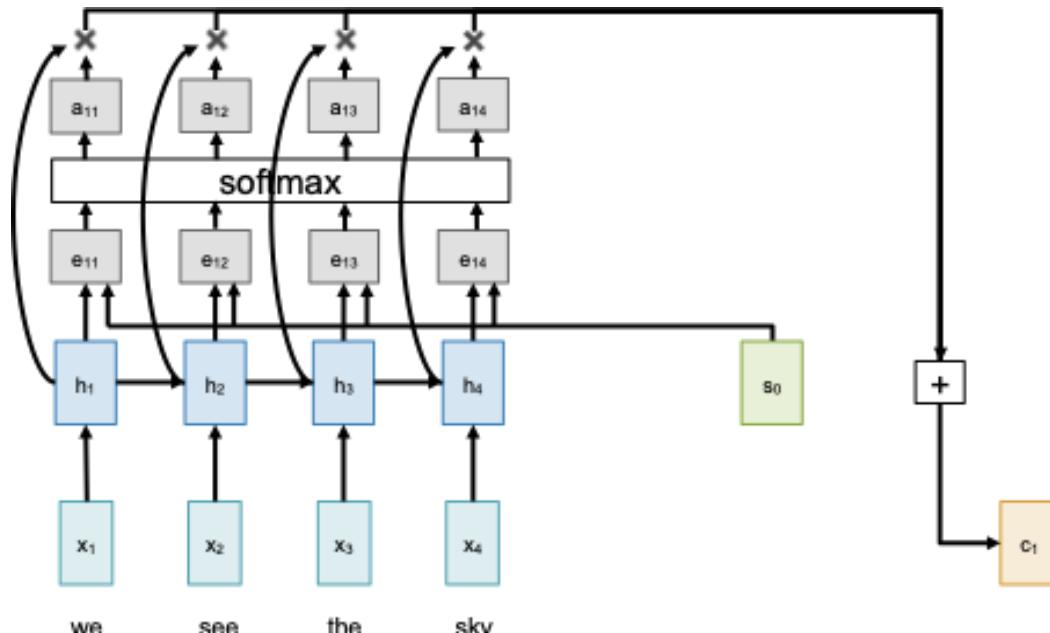
- Use dot product for similarity

Attention Layer

Inputs:

Query vector: q [D_Q]

Data vectors: X [$N_x \times D_Q$]



Computation:

Similarities: $e_i = q \cdot X_i / \sqrt{D_Q}$

Attention weights: $a = \text{softmax}(e)$ [N_x]

Output vector: $y = \sum_i a_i X_i$ [D_x]

Changes

- Use **scaled** dot product for similarity

Attention Layer

Inputs:

Query vector: q [D_Q]

Data vectors: X [$N_x \times D_Q$]

Large similarities will cause softmax to saturate and give vanishing gradients

$$\text{Recall } a \cdot b = |a||b| \cos(\text{angle})$$

Suppose that a and b are constant vectors of dimension D

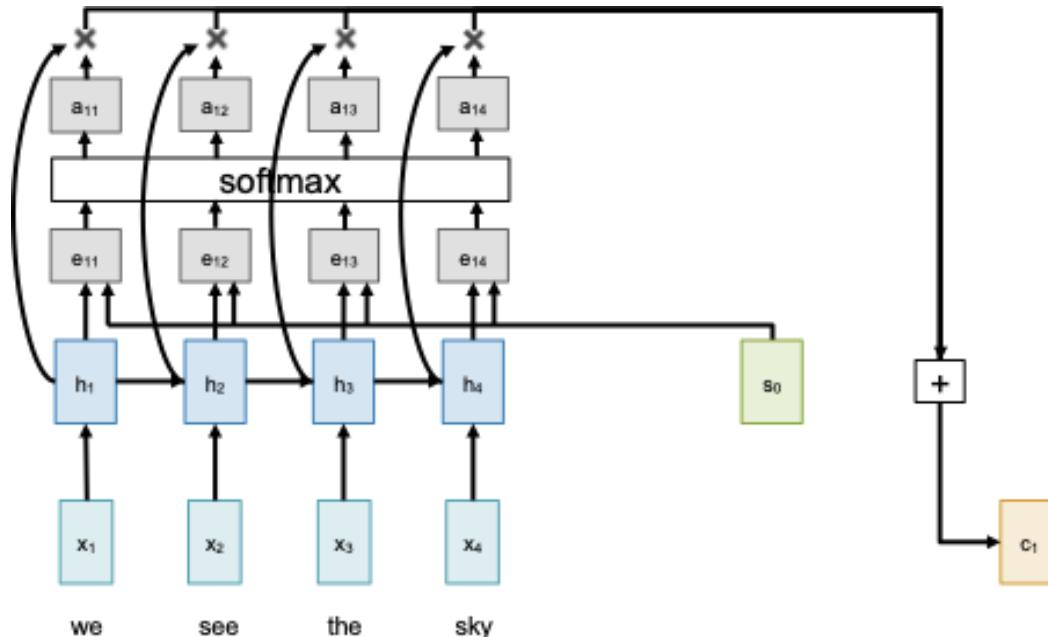
$$\text{Then } |a| = (\sum_i a_i^2)^{1/2} = a \sqrt{D}$$

Computation:

Similarities: e [N_x] $e_i = q \cdot X_i / \sqrt{D_Q}$

Attention weights: $a = \text{softmax}(e)$ [N_x]

Output vector: $y = \sum_i a_i X_i$ [D_x]



Changes

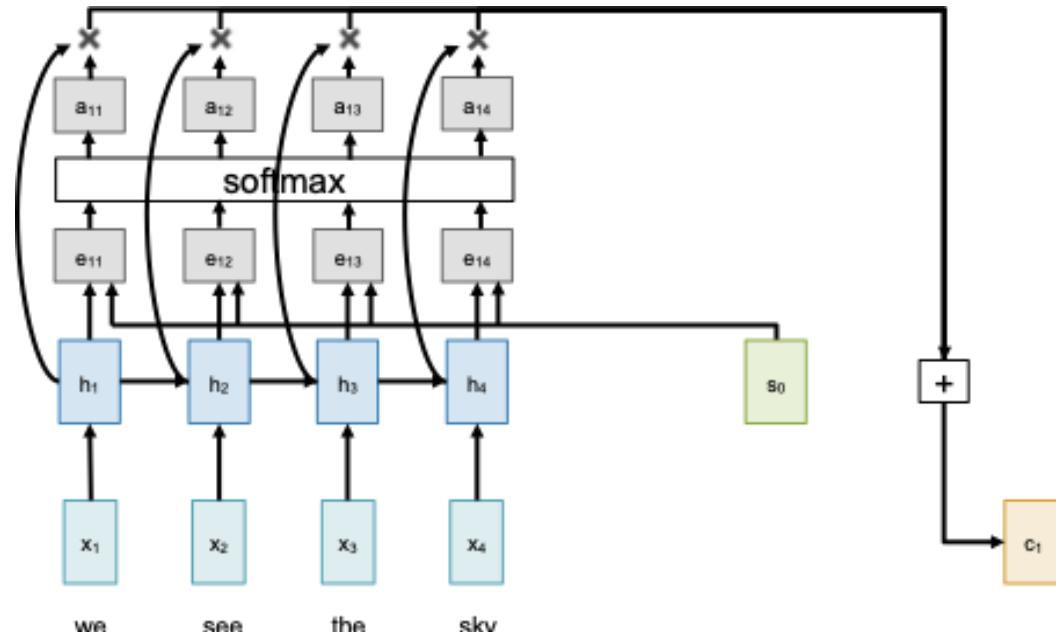
- Use **scaled** dot product for similarity

Attention Layer

Inputs:

Query vector: \mathbf{Q} [$N_Q \times D_Q$]

Data vectors: \mathbf{X} [$N_X \times D_X$]



Computation:

Similarities: $E = \mathbf{Q}\mathbf{X}^T / \sqrt{D_Q}$ [$N_Q \times N_X$]

$$E_{ij} = \mathbf{Q}_i \cdot \mathbf{X}_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [$N_Q \times N_X$]

Output vector: $\mathbf{Y} = A\mathbf{X}$ [$N_Q \times D_X$]

$$\mathbf{Y}_i = \sum_j A_{ij} \mathbf{X}_j$$

Changes

- Use scaled dot product for similarity
- Multiple **query** vectors

Attention Layer

Inputs:

Query vector: Q [$N_Q \times D_Q$]

Data vectors: X [$N_x \times D_Q$]

Key matrix: W_K [$D_x \times D_Q$]

Value matrix: W_V [$D_x \times D_V$]

Computation:

Keys: $K = XW_K$ [$N_x \times D_Q$]

Values: $V = XW_V$ [$N_x \times D_V$]

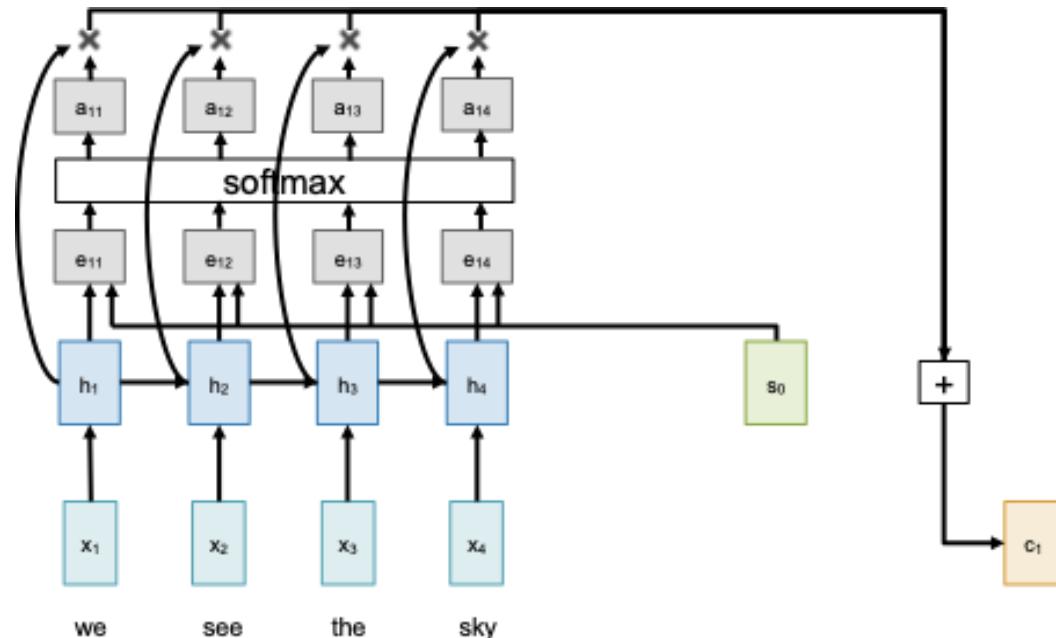
Similarities: $E = QK^T / \sqrt{D_Q}$ [$N_Q \times N_x$]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [$N_Q \times N_x$]

Output vector: $Y = A V$ [$N_Q \times D_x$]

$$Y_i = \sum_j A_{ij} V_j$$



Changes

- Use scaled dot product for similarity
- Multiple **query** vectors
- Separate **key** and **query**

Attention Layer

Inputs:

Query vector: \mathbf{Q} [$N_Q \times D_Q$]

Data vectors: \mathbf{X} [$N_x \times D_Q$]

Key matrix: $\mathbf{W_K}$ [$D_x \times D_Q$]

Value matrix: $\mathbf{W_V}$ [$D_x \times D_V$]

Computation:

Keys: $\mathbf{K} = \mathbf{XW_K}$ [$N_x \times D_Q$]

X_1

X_2

X_3

Values: $\mathbf{V} = \mathbf{XW_V}$ [$D_x \times D_V$]

Similarities: $E = \mathbf{QK^T} / \sqrt{D_Q}$ [$N_Q \times N_x$]

$E_{ij} = \mathbf{Q}_i \cdot \mathbf{K}_j / \sqrt{D_Q}$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [$N_Q \times N_x$]

Output vector: $\mathbf{Y} = A\mathbf{V}$ [$N_Q \times D_V$]

$\mathbf{Y}_i = \sum_j A_{ij} \mathbf{V}_j$

Q_1

Q_2

Q_3

Q_4

Attention Layer

Inputs:

Query vector: \mathbf{Q} [$N_Q \times D_Q$]

Data vectors: \mathbf{X} [$N_x \times D_Q$]

Key matrix: \mathbf{W}_K [$D_x \times D_Q$]

Value matrix: \mathbf{W}_V [$D_x \times D_V$]

Computation:

Keys: $\mathbf{K} = \mathbf{X}\mathbf{W}_K$ [$N_x \times D_Q$]

Values: $\mathbf{V} = \mathbf{X}\mathbf{W}_V$ [$D_x \times D_V$]

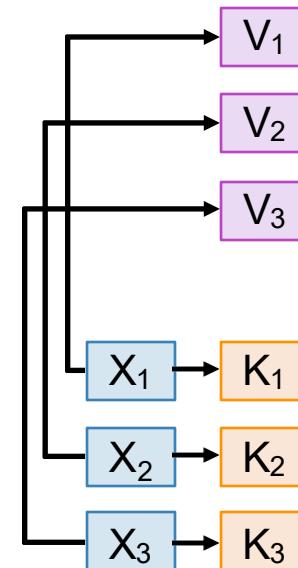
Similarities: $E = \mathbf{Q}\mathbf{K}^T / \sqrt{D_Q}$ [$N_Q \times N_x$]

$$E_{ij} = \mathbf{Q}_i \cdot \mathbf{K}_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [$N_Q \times N_x$]

Output vector: $\mathbf{Y} = A\mathbf{V}$ [$N_Q \times D_V$]

$$\mathbf{Y}_i = \sum_j A_{ij} \mathbf{V}_j$$



Q_1

Q_2

Q_3

Q_4

Attention Layer

Inputs:

Query vector: \mathbf{Q} [$N_Q \times D_Q$]

Data vectors: \mathbf{X} [$N_x \times D_Q$]

Key matrix: \mathbf{W}_K [$D_x \times D_Q$]

Value matrix: \mathbf{W}_V [$D_x \times D_V$]

Computation:

Keys: $\mathbf{K} = \mathbf{X}\mathbf{W}_K$ [$N_x \times D_Q$]

Values: $\mathbf{V} = \mathbf{X}\mathbf{W}_V$ [$D_x \times D_V$]

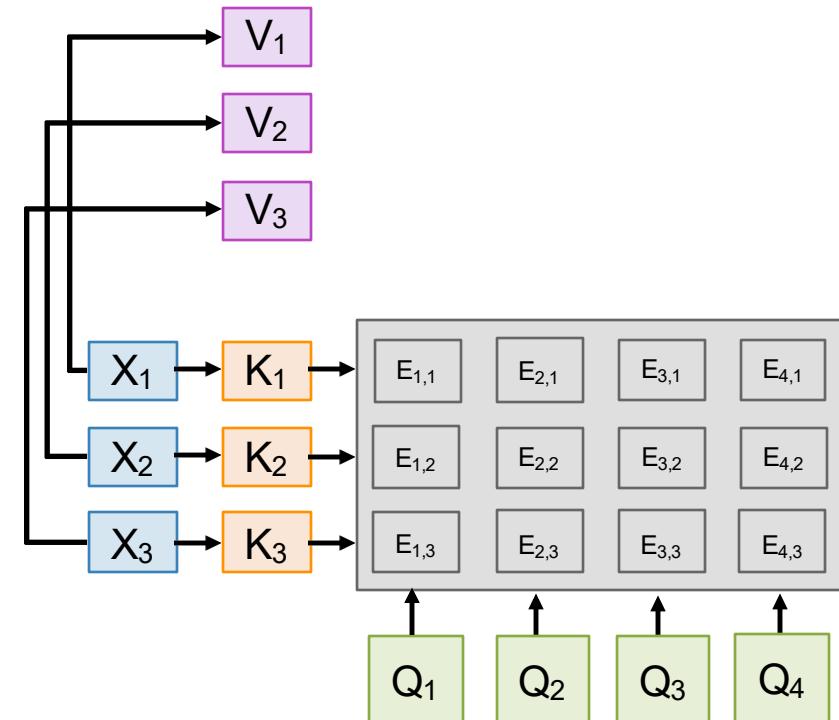
Similarities: $E = \mathbf{Q}\mathbf{K}^T / \sqrt{D_Q}$ [$N_Q \times N_x$]

$$E_{ij} = \mathbf{Q}_i \cdot \mathbf{K}_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [$N_Q \times N_x$]

Output vector: $\mathbf{Y} = A\mathbf{V}$ [$N_Q \times D_V$]

$$\mathbf{Y}_i = \sum_j A_{ij} \mathbf{V}_j$$



Attention Layer

Inputs:

Query vector: Q [$N_Q \times D_Q$]

Data vectors: X [$N_x \times D_Q$]

Key matrix: W_K [$D_X \times D_Q$]

Value matrix: W_V [$D_X \times D_V$]

Softmax normalizes each column: each **query** predicts a distribution over the **keys**

Computation:

Keys: $K = XW_K$ [$N_x \times D_Q$]

Values: $V = XW_V$ [$D_X \times D_V$]

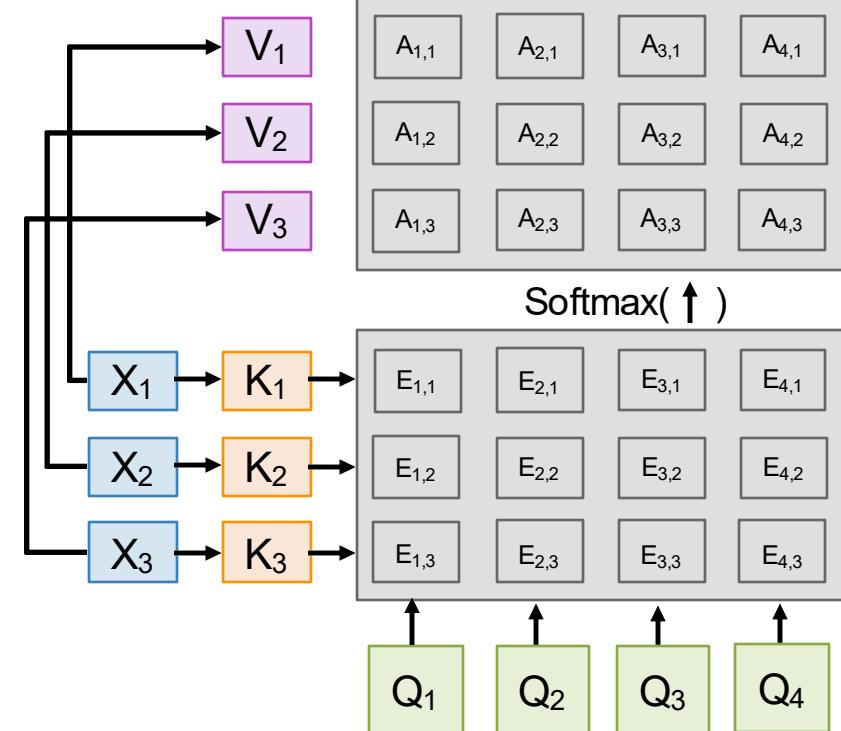
Similarities: $E = QK^T / \sqrt{D_Q}$ [$N_Q \times N_x$]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [$N_Q \times N_x$]

Output vector: $Y = A V$ [$N_Q \times D_V$]

$$Y_i = \sum_j A_{ij} V_j$$



Attention Layer

Inputs:

Query vector: \mathbf{Q} [$N_Q \times D_Q$]

Data vectors: \mathbf{X} [$N_x \times D_Q$]

Key matrix: \mathbf{W}_K [$D_X \times D_Q$]

Value matrix: \mathbf{W}_V [$D_X \times D_V$]

Computation:

Keys: $\mathbf{K} = \mathbf{X}\mathbf{W}_K$ [$N_x \times D_Q$]

Values: $\mathbf{V} = \mathbf{X}\mathbf{W}_V$ [$D_X \times D_V$]

Similarities: $E = \mathbf{Q}\mathbf{K}^T / \sqrt{D_Q}$ [$N_Q \times N_x$]

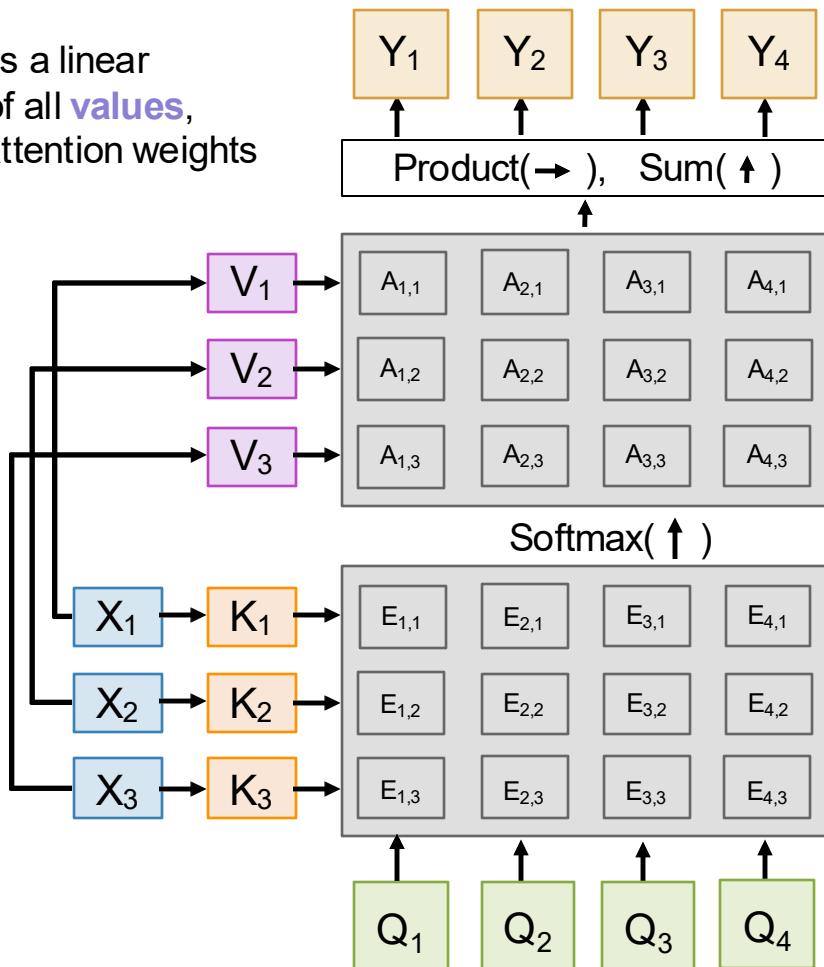
$$E_{ij} = \mathbf{Q}_i \cdot \mathbf{K}_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [$N_Q \times N_x$]

Output vector: $\mathbf{Y} = A\mathbf{V}$ [$N_Q \times D_V$]

$$\mathbf{Y}_i = \sum_j A_{ij} \mathbf{V}_j$$

Each **output** is a linear combination of all **values**, weighted by attention weights



Cross-Attention Layer

Inputs:

Query vector: Q [$N_Q \times D_Q$]

Data vectors: X [$N_x \times D_Q$]

Key matrix: W_K [$D_X \times D_Q$]

Value matrix: W_V [$D_X \times D_V$]

Each **query** produces one **output**, which is a mix of information in the **data** vectors

Computation:

Keys: $K = XW_K$ [$N_x \times D_Q$]

Values: $V = XW_V$ [$D_X \times D_V$]

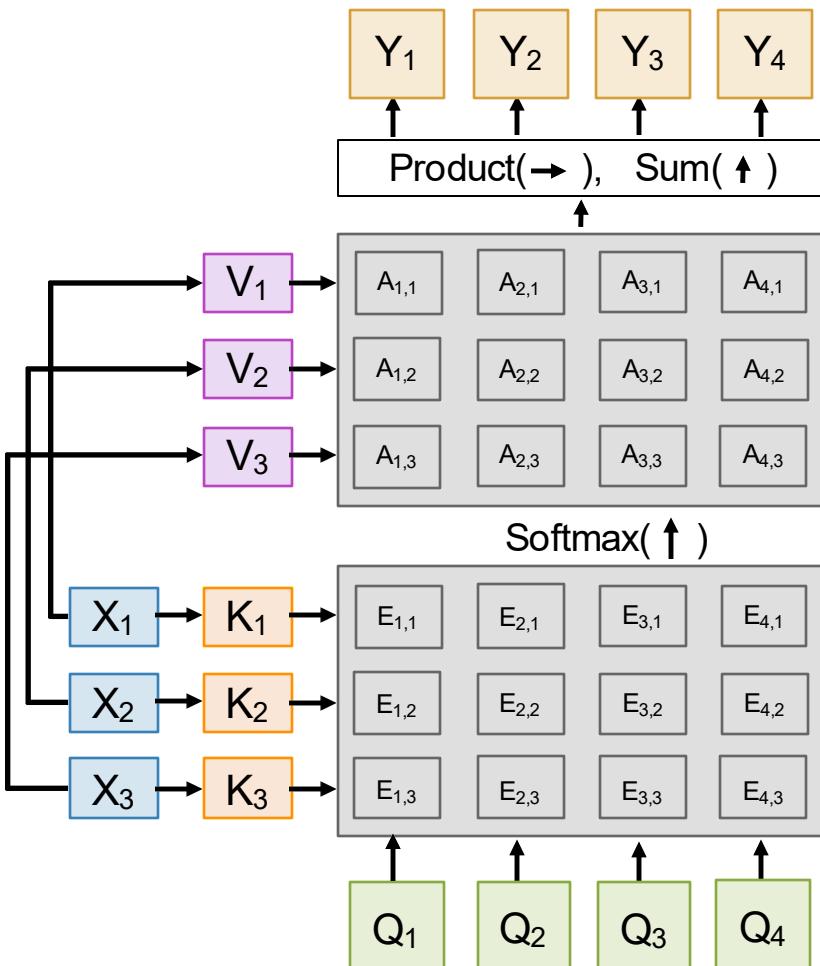
Similarities: $E = QK^T / \sqrt{D_Q}$ [$N_Q \times N_x$]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [$N_Q \times N_x$]

Output vector: $Y = A V$ [$N_Q \times D_V$]

$$Y_i = \sum_j A_{ij} V_j$$



Self-Attention Layer

Inputs:

Input vectors: X [N x D_{in}]

Key matrix: W_K [D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q [D_{in} x D_{out}]

Each **input** produces one **output**, which is a mix of information from all **inputs**

Computation:

Queries: $Q = XW_Q$ [N x D_{out}]

Keys: $K = XW_K$ [N x D_{out}]

Values: $V = XW_V$ [N x D_{out}]

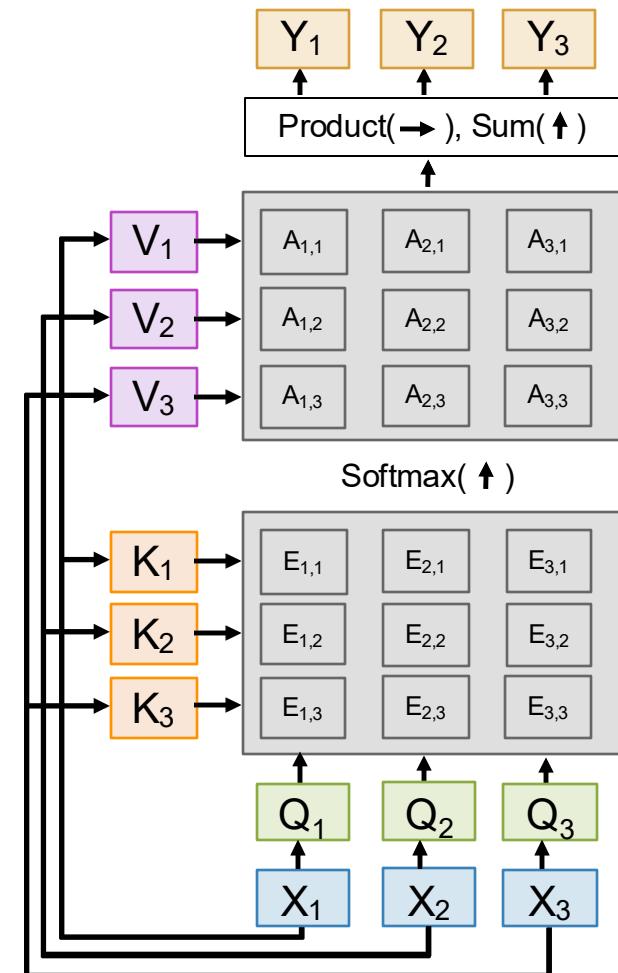
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]

Output vector: $Y = AV$ [N x D_{out}]

$$Y_i = \sum_j A_{ij} V_j$$



Self-Attention Layer

Inputs:

Input vectors: X [N x D_{in}]

Key matrix: W_K [D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q [D_{in} x D_{out}]

Each **input** produces one **output**, which is a mix of information from all **inputs**

Computation:

Queries: $Q = XW_Q$ [N x D_{out}]

Keys: $K = XW_K$ [N x D_{out}]

Values: $V = XW_V$ [N x D_{out}]

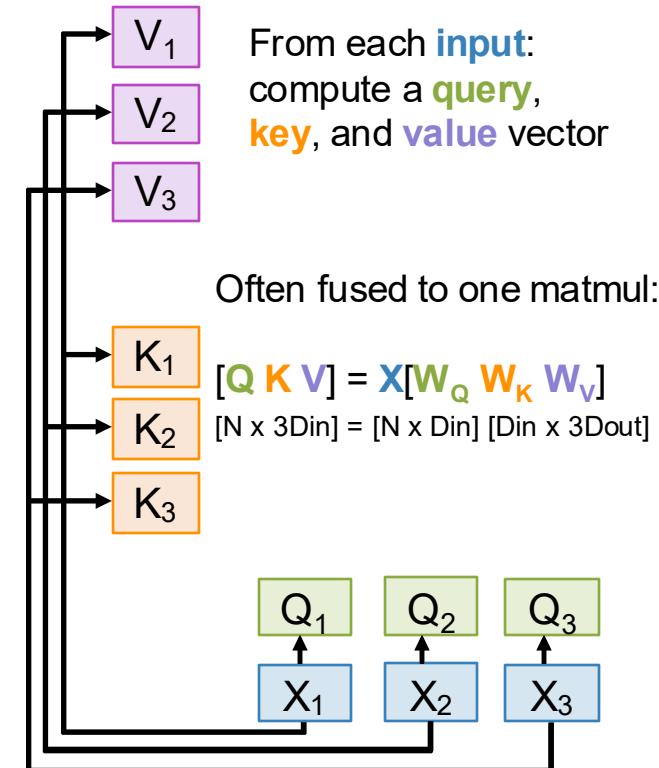
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]

Output vector: $Y = AV$ [N x D_{out}]

$$Y_i = \sum_j A_{ij} V_j$$



Self-Attention Layer

Inputs:

Input vectors: X [N x D_{in}]

Key matrix: W_K [D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q [D_{in} x D_{out}]

Each **input** produces one **output**, which is a mix of information from all **inputs**

Computation:

Queries: $Q = XW_Q$ [N x D_{out}]

Keys: $K = XW_K$ [N x D_{out}]

Values: $V = XW_V$ [N x D_{out}]

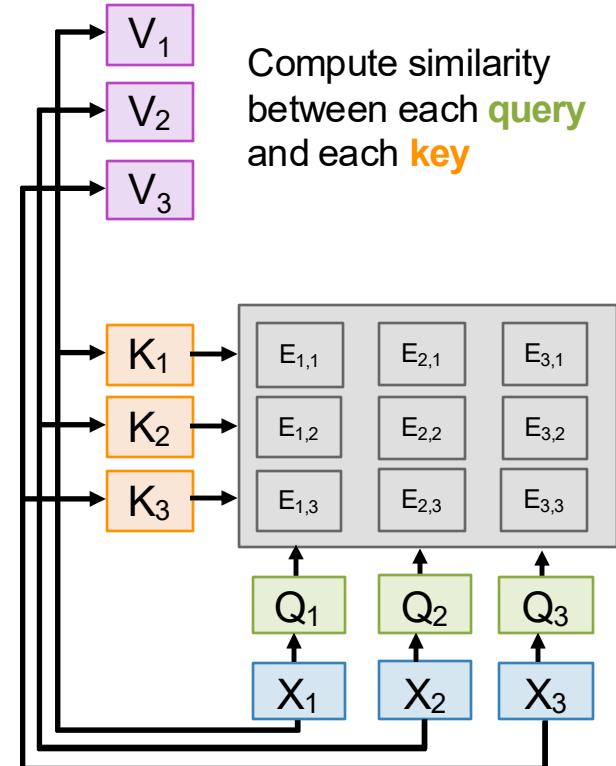
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]

Output vector: $Y = AV$ [N x D_{out}]

$$Y_i = \sum_j A_{ij} V_j$$



Self-Attention Layer

Inputs:

Input vectors: X [N x D_{in}]

Key matrix: W_K [D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q [D_{in} x D_{out}]

Each **input** produces one **output**, which is a mix of information from all **inputs**

Computation:

Queries: $Q = XW_Q$ [N x D_{out}]

Keys: $K = XW_K$ [N x D_{out}]

Values: $V = XW_V$ [N x D_{out}]

Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]

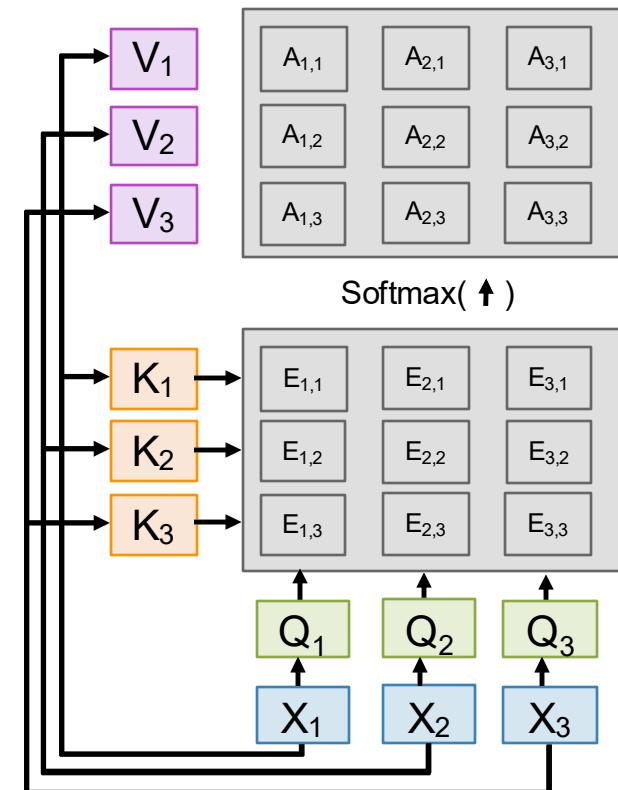
$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]

Output vector: $Y = AV$ [N x D_{out}]

$$Y_i = \sum_j A_{ij} V_j$$

Normalize over each column:
each **query** computes a distribution over **keys**



Self-Attention Layer

Inputs:

Input vectors: X [N x D_{in}]

Key matrix: W_K [D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q [D_{in} x D_{out}]

Each **input** produces one **output**, which is a mix of information from all **inputs**

Computation:

Queries: $Q = XW_Q$ [N x D_{out}]

Keys: $K = XW_K$ [N x D_{out}]

Values: $V = XW_V$ [N x D_{out}]

Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]

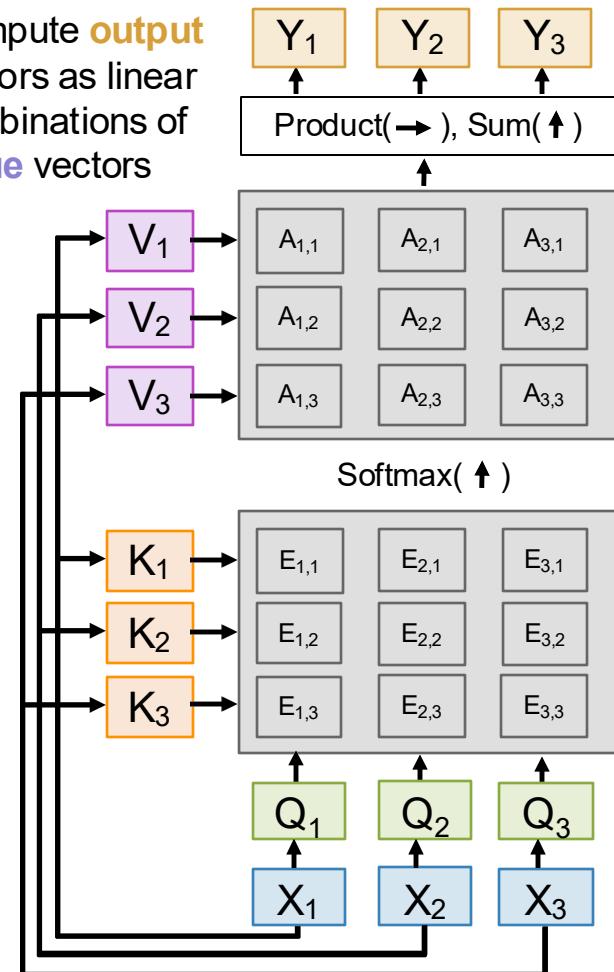
$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]

Output vector: $Y = AV$ [N x D_{out}]

$$Y_i = \sum_j A_{ij} V_j$$

Compute **output** vectors as linear combinations of **value** vectors



Self-Attention Layer

Consider permuting **inputs**:

Inputs:

Input vectors: X [N x D_{in}]

Key matrix: W_K [D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q [D_{in} x D_{out}]

Computation:

Queries: $Q = XW_Q$ [N x D_{out}]

Keys: $K = XW_K$ [N x D_{out}]

Values: $V = XW_V$ [N x D_{out}]

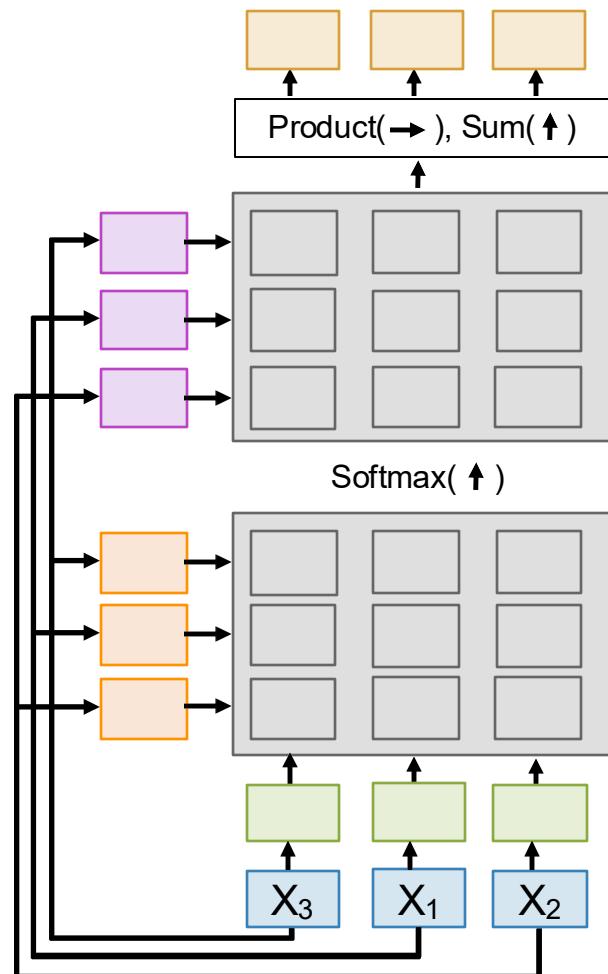
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]

Output vector: $Y = AV$ [N x D_{out}]

$$Y_i = \sum_j A_{ij} V_j$$



Self-Attention Layer

Inputs:

Input vectors: \mathbf{X} [$N \times D_{in}$]

Key matrix: \mathbf{W}_K [$D_{in} \times D_{out}$]

Value matrix: \mathbf{W}_V [$D_{in} \times D_{out}$]

Query matrix: \mathbf{W}_Q [$D_{in} \times D_{out}$]

Computation:

Queries: $\mathbf{Q} = \mathbf{X}\mathbf{W}_Q$ [$N \times D_{out}$]

Keys: $\mathbf{K} = \mathbf{X}\mathbf{W}_K$ [$N \times D_{out}$]

Values: $\mathbf{V} = \mathbf{X}\mathbf{W}_V$ [$N \times D_{out}$]

Similarities: $E = \mathbf{Q}\mathbf{K}^T / \sqrt{D_Q}$ [$N \times N$]

$$E_{ij} = \mathbf{Q}_i \cdot \mathbf{K}_j / \sqrt{D_Q}$$

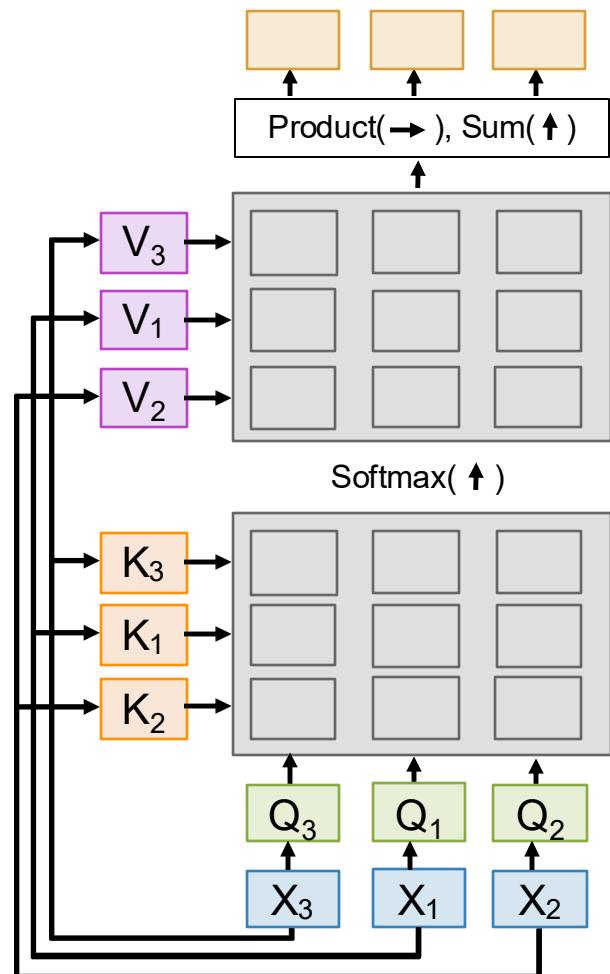
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [$N \times N$]

Output vector: $\mathbf{Y} = A\mathbf{V}$ [$N \times D_{out}$]

$$\mathbf{Y}_i = \sum_j A_{ij} \mathbf{V}_j$$

Consider permuting **inputs**:

Queries, keys, and values
will be the same but permuted



Self-Attention Layer

Inputs:

Input vectors: X [N x D_{in}]

Key matrix: W_K [D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q [D_{in} x D_{out}]

Computation:

Queries: $Q = XW_Q$ [N x D_{out}]

Keys: $K = XW_K$ [N x D_{out}]

Values: $V = XW_V$ [N x D_{out}]

Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]

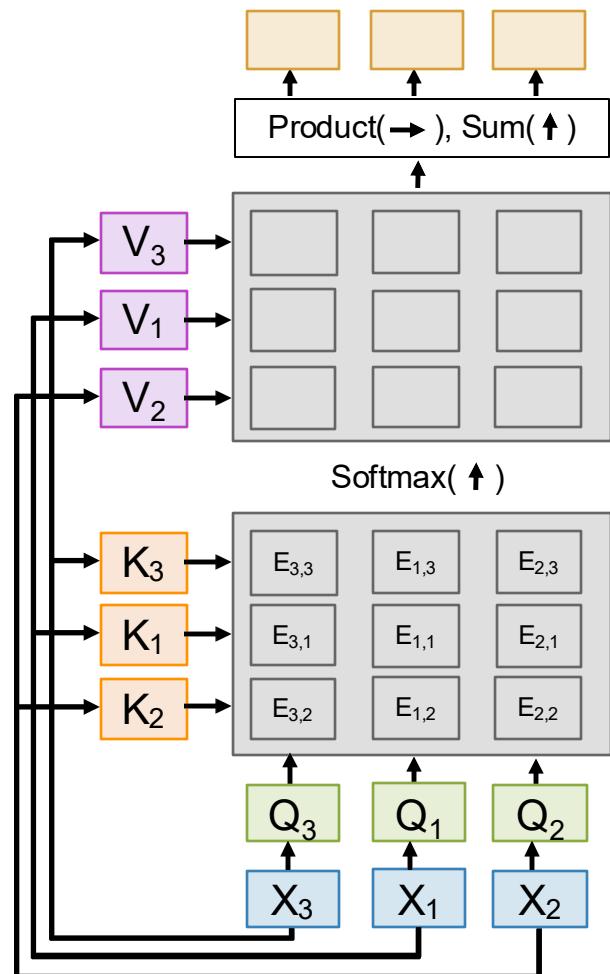
Output vector: $Y = AV$ [N x D_{out}]

$$Y_i = \sum_j A_{ij} V_j$$

Consider permuting **inputs**:

Queries, keys, and values
will be the same but permuted

Similarities are the same but
permuted



Self-Attention Layer

Inputs:

Input vectors: X [N x D_{in}]

Key matrix: W_K [D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q [D_{in} x D_{out}]

Computation:

Queries: $Q = XW_Q$ [N x D_{out}]

Keys: $K = XW_K$ [N x D_{out}]

Values: $V = XW_V$ [N x D_{out}]

Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]

Output vector: $Y = AV$ [N x D_{out}]

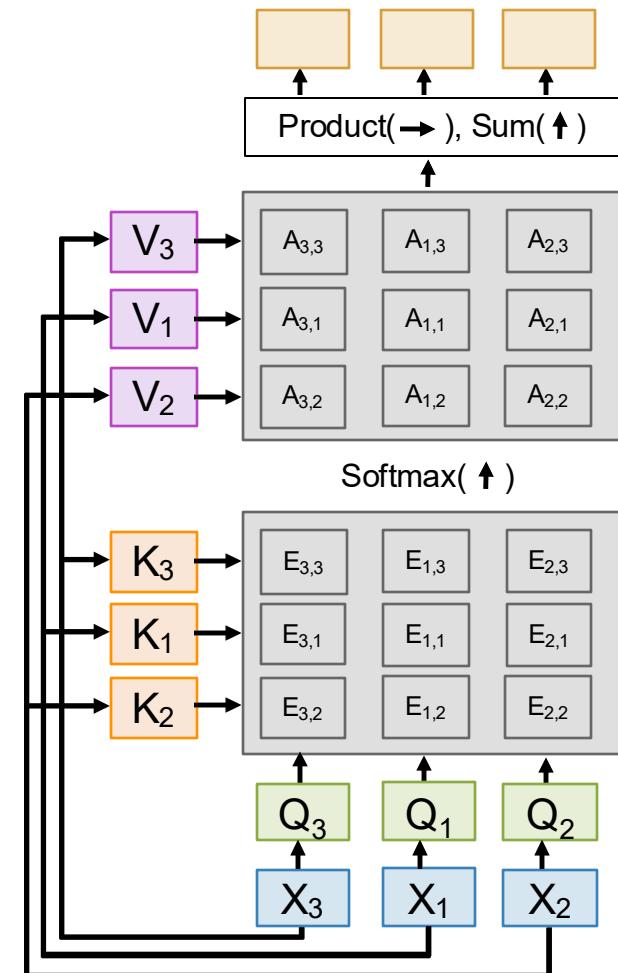
$$Y_i = \sum_j A_{ij} V_j$$

Consider permuting **inputs**:

Queries, keys, and values
will be the same but permuted

Similarities are the same but
permuted

Attention weights are the
same but permuted



Self-Attention Layer

Inputs:

Input vectors: X [N x D_{in}]

Key matrix: W_K [D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q [D_{in} x D_{out}]

Computation:

Queries: $Q = XW_Q$ [N x D_{out}]

Keys: $K = XW_K$ [N x D_{out}]

Values: $V = XW_V$ [N x D_{out}]

Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]

Output vector: $Y = AV$ [N x D_{out}]

$$Y_i = \sum_j A_{ij} V_j$$

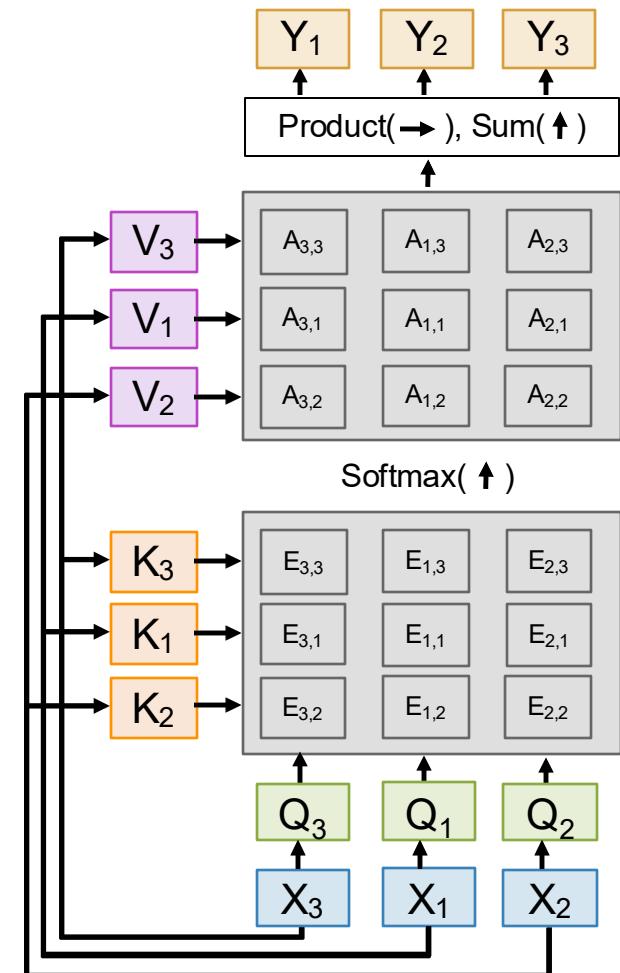
Consider permuting **inputs**:

Queries, keys, and values will be the same but permuted

Similarities are the same but permuted

Attention weights are the same but permuted

Outputs are the same but permuted



Self-Attention Layer

Inputs:

Input vectors: X [N x D_{in}]

Key matrix: W_K [D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q [D_{in} x D_{out}]

Self-Attention is
permutation equivariant:
 $F(\sigma(X)) = \sigma(F(X))$

Computation:

Queries: $Q = XW_Q$ [N x D_{out}]

Keys: $K = XW_K$ [N x D_{out}]

Values: $V = XW_V$ [N x D_{out}]

Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]

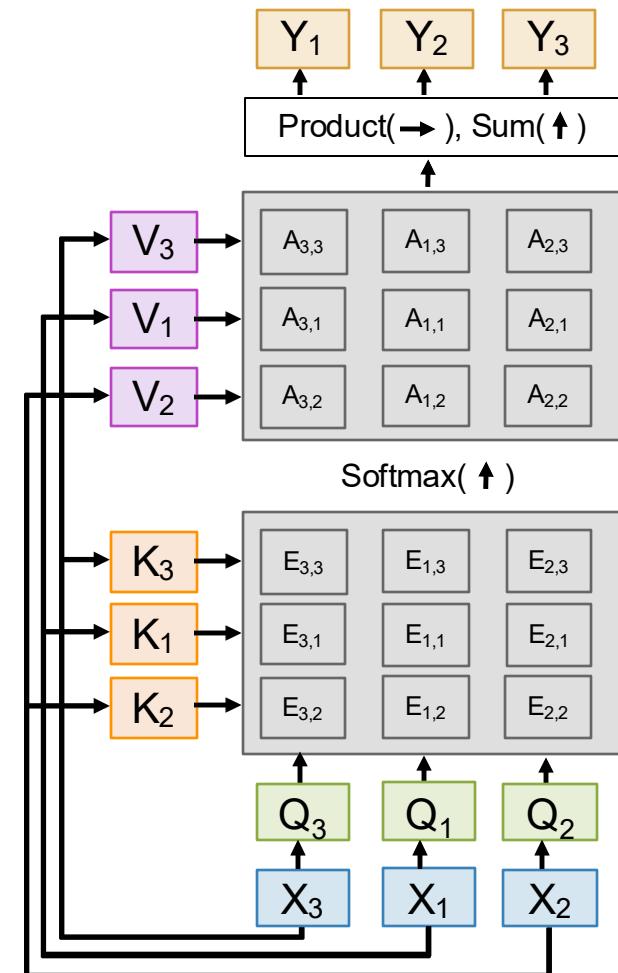
$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]

Output vector: $Y = AV$ [N x D_{out}]

$$Y_i = \sum_j A_{ij} V_j$$

This means that Self-Attention
works on **sets of vectors**



Self-Attention Layer

Inputs:

Input vectors: X [N x D_{in}]

Key matrix: W_K [D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q [D_{in} x D_{out}]

Problem: Self-Attention does not know the order of the sequence

Computation:

Queries: $Q = XW_Q$ [N x D_{out}]

Keys: $K = XW_K$ [N x D_{out}]

Values: $V = XW_V$ [N x D_{out}]

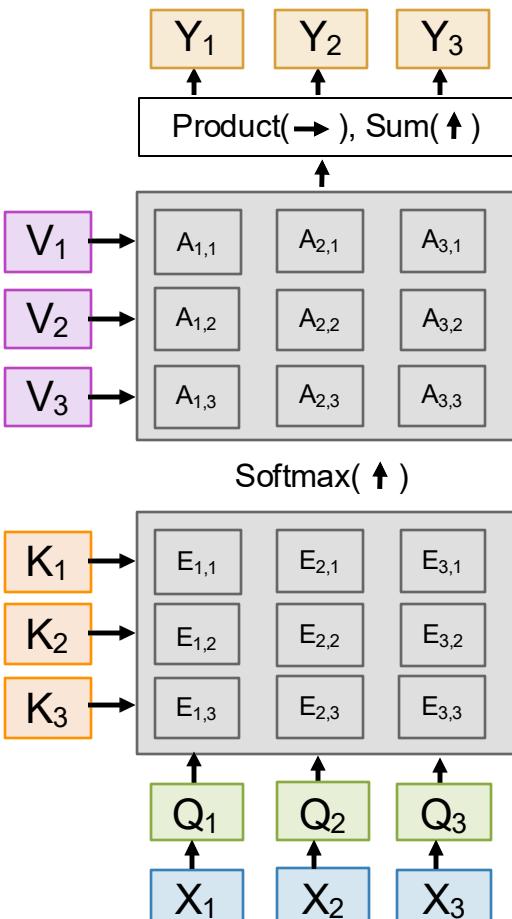
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]

Output vector: $Y = AV$ [N x D_{out}]

$$Y_i = \sum_j A_{ij} V_j$$



Self-Attention Layer

Inputs:

Input vectors: X [N x D_{in}]

Key matrix: W_K [D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q [D_{in} x D_{out}]

Computation:

Queries: $Q = XW_Q$ [N x D_{out}]

Keys: $K = XW_K$ [N x D_{out}]

Values: $V = XW_V$ [N x D_{out}]

Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

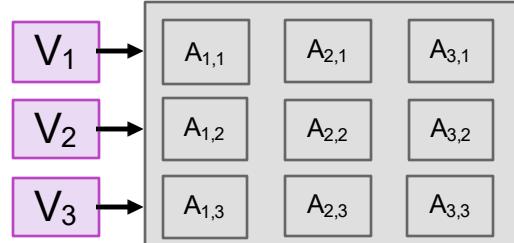
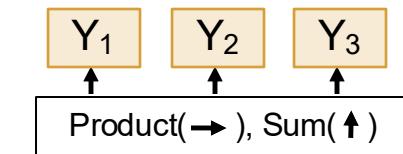
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]

Output vector: $Y = AV$ [N x D_{out}]

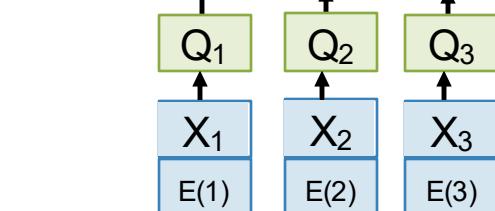
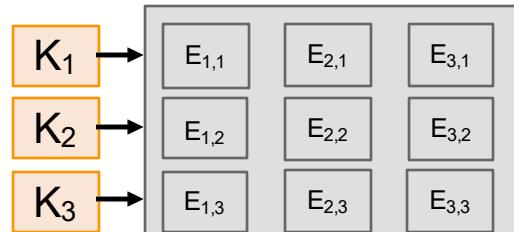
$$Y_i = \sum_j A_{ij} V_j$$

Problem: Self-Attention does not know the order of the sequence

Solution: Add positional encoding to each input; this is a vector that is a fixed function of the index



Softmax(↑)



Masked Self-Attention Layer

Don't let vectors "look ahead" in the sequence

Inputs:

Input vectors: X [N x D_{in}]

Key matrix: W_K [D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q [D_{in} x D_{out}]

Override similarities with -inf;
this controls which inputs each
vector is allowed to look at.

Computation:

Queries: $Q = XW_Q$ [N x D_{out}]

Keys: $K = XW_K$ [N x D_{out}]

Values: $V = XW_V$ [N x D_{out}]

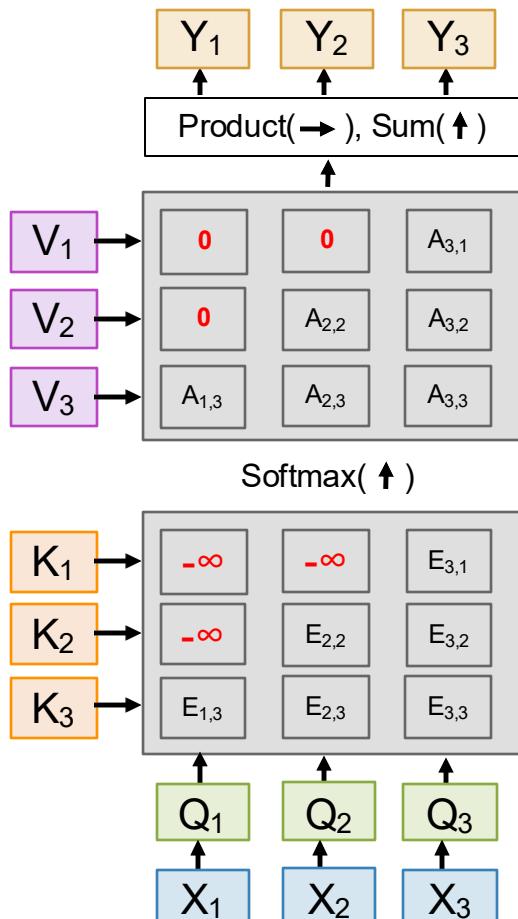
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]

Output vector: $Y = AV$ [N x D_{out}]

$$Y_i = \sum_j A_{ij} V_j$$



Masked Self-Attention Layer

Don't let vectors "look ahead" in the sequence

Inputs:

Input vectors: X [N x D_{in}]

Key matrix: W_K [D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q [D_{in} x D_{out}]

Override similarities with -inf;
this controls which inputs each
vector is allowed to look at.

Computation:

Queries: $Q = XW_Q$ [N x D_{out}]

Keys: $K = XW_K$ [N x D_{out}]

Values: $V = XW_V$ [N x D_{out}]

Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]

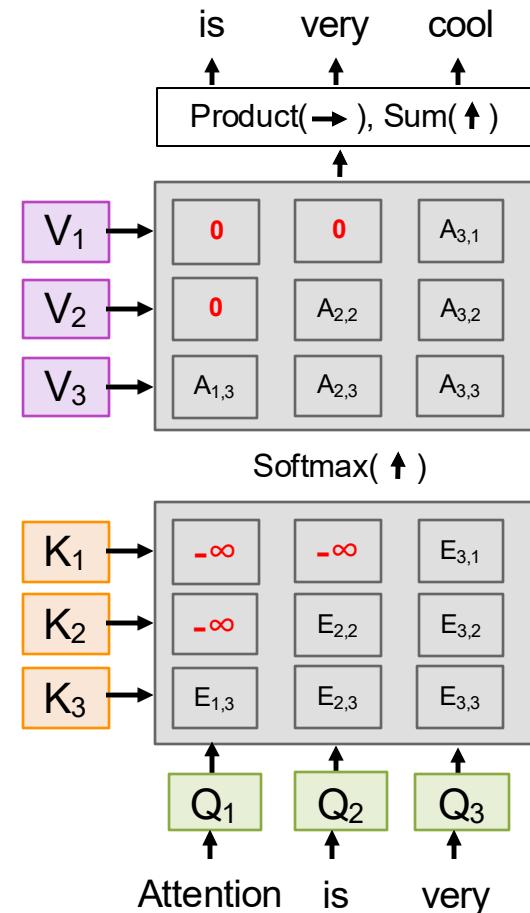
$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]

Output vector: $Y = AV$ [N x D_{out}]

$$Y_i = \sum_j A_{ij} V_j$$

Used for language modeling
where you want to predict the
next word



Multiheaded Self-Attention Layer

Run H copies of Self-Attention in parallel

Inputs:

Input vectors: \mathbf{X} [N x D_{in}]

Key matrix: \mathbf{W}_K [D_{in} x D_{out}]

Value matrix: \mathbf{W}_V [D_{in} x D_{out}]

Query matrix: \mathbf{W}_Q [D_{in} x D_{out}]

Computation:

Queries: $\mathbf{Q} = \mathbf{X}\mathbf{W}_Q$ [N x D_{out}]

Keys: $\mathbf{K} = \mathbf{X}\mathbf{W}_K$ [N x D_{out}]

Values: $\mathbf{V} = \mathbf{X}\mathbf{W}_V$ [N x D_{out}]

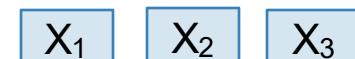
Similarities: $E = \mathbf{Q}\mathbf{K}^T / \sqrt{D_Q}$ [N x N]

$$E_{ij} = \mathbf{Q}_i \cdot \mathbf{K}_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]

Output vector: $\mathbf{Y} = \mathbf{A}\mathbf{X}$ [N x D_{out}]

$$\mathbf{Y}_i = \sum_j A_{ij} \mathbf{V}_j$$



Multiheaded Self-Attention Layer

Run H copies of Self-Attention in parallel

Inputs:

Input vectors: X [N x D_{in}]

Key matrix: W_K [D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q [D_{in} x D_{out}]

Computation:

Queries: $Q = XW_Q$ [N x D_{out}]

Keys: $K = XW_K$ [N x D_{out}]

Values: $V = XW_V$ [N x D_{out}]

Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]

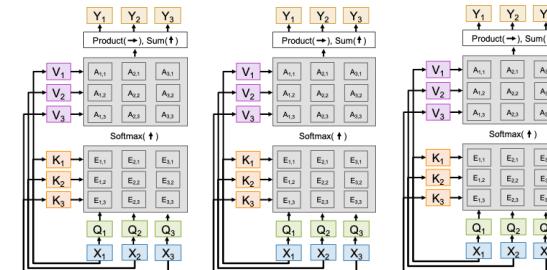
$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]

Output vector: $Y = AX$ [N x D_{out}]

$$Y_i = \sum_j A_{ij} V_j$$

H = 3 independent self-attention layers (called heads), each with their own weights



X_1

X_2

X_3

Multiheaded Self-Attention Layer

Run H copies of Self-Attention in parallel

Inputs:

Input vectors: X [N x D_{in}]

Key matrix: W_K [D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q [D_{in} x D_{out}]

Computation:

Queries: $Q = XW_Q$ [N x D_{out}]

Keys: $K = XW_K$ [N x D_{out}]

Values: $V = XW_V$ [N x D_{out}]

Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]

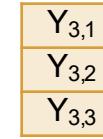
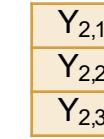
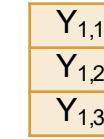
$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]

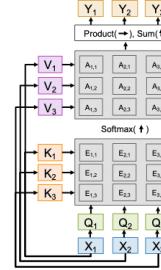
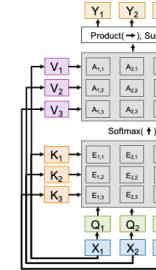
Output vector: $Y = AX$ [N x D_{out}]

$$Y_i = \sum_j A_{ij} V_j$$

Stack up the H independent outputs for each input X



H = 3 independent self-attention layers (called heads), each with their own weights



X_1

X_2

X_3

Multiheaded Self-Attention Layer

Run H copies of Self-Attention in parallel

Inputs:

Input vectors: X [N x D_{in}]

Key matrix: W_K [D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q [D_{in} x D_{out}]

Computation:

Queries: $Q = XW_Q$ [N x D_{out}]

Keys: $K = XW_K$ [N x D_{out}]

Values: $V = XW_V$ [N x D_{out}]

Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

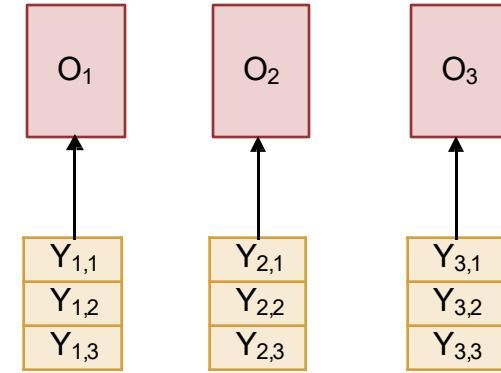
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]

Output vector: $Y = AX$ [N x D_{out}]

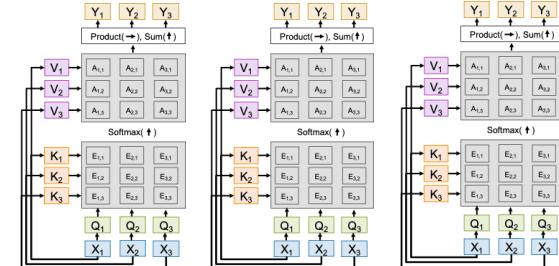
$$Y_i = \sum_j A_{ij} V_j$$

Output projection fuses
data from each head

Stack up the H
independent outputs
for each input X



H = 3 independent
self-attention layers
(called heads), each
with their own weights



X_1 X_2 X_3

Multiheaded Self-Attention Layer

Run H copies of Self-Attention in parallel

Inputs:

Input vectors: X [N x D]

Key matrix: W_K [D x HD_H]

Value matrix: W_V [D x HD_H]

Query matrix: W_Q [D x HD_H]

Output matrix: W_O [HD_H x D]

Each of the H parallel layers use a qkv dim of D_H = “head dim”

Usually $D_H = D / H$, so inputs and outputs have the same dimension

Computation:

Queries: $Q = XW_Q$ [H x N x D_H]

Keys: $K = XW_K$ [H x N x D_H]

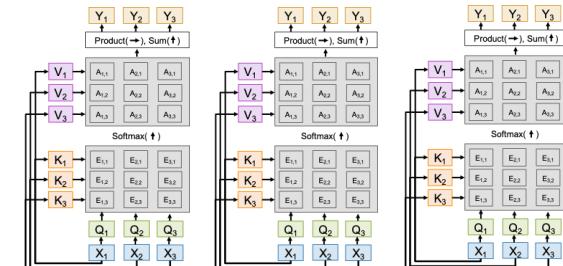
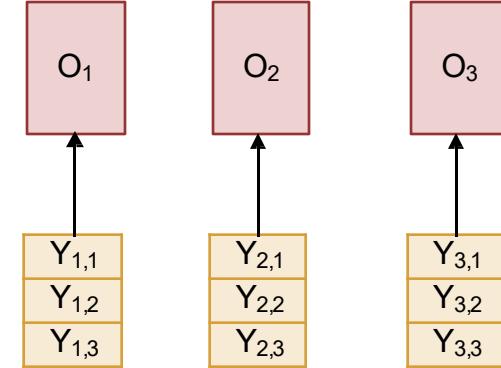
Values: $V = XW_V$ [H x N x D_H]

Similarities: $E = QK^T / \sqrt{D_Q}$ [H x N x N]

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [H x N x N]

Head outputs: $Y = AV$ [H x N x D_H] \Rightarrow [N x HD_H]

Outputs: $O = YW_O$ [N x D]



X_1 X_2 X_3

Multiheaded Self-Attention Layer

Run H copies of Self-Attention in parallel

Inputs:

Input vectors: X [N x D]

Key matrix: W_K [D x HD_H]

Value matrix: W_V [D x HD_H]

Query matrix: W_Q [D x HD_H]

Output matrix: W_O [HD_H x D]

In practice, compute all H heads in parallel using batched matrix multiply operations.

Computation:

Queries: $Q = XW_Q$ [H x N x D_H]

Keys: $K = XW_K$ [H x N x D_H]

Values: $V = XW_V$ [H x N x D_H]

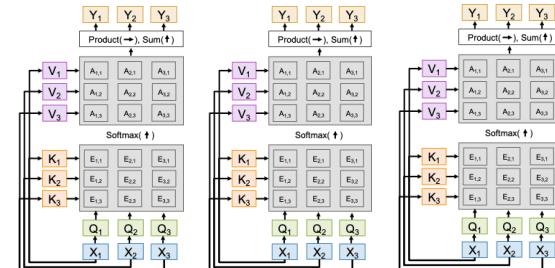
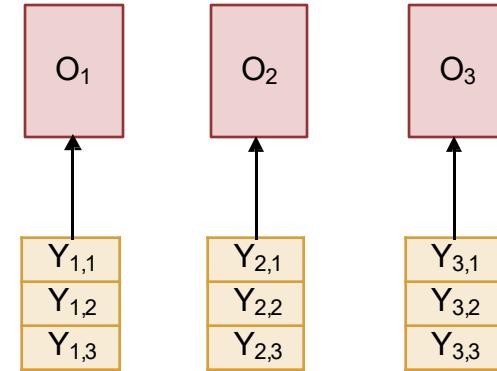
Similarities: $E = QK^T / \sqrt{D_Q}$ [H x N x N]

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [H x N x N]

Head outputs: $Y = AV$ [H x N x D_H] \Rightarrow [N x HD_H]

Outputs: $O = YW_O$ [N x D]

Used everywhere in practice.



X_1 X_2 X_3

Self-Attention is Four Matrix Multiplies!

Inputs:

Input vectors: \mathbf{X} [N x D]

Key matrix: \mathbf{W}_K [D x HD_H]

Value matrix: \mathbf{W}_V [D x HD_H]

Query matrix: \mathbf{W}_Q [D x HD_H]

Output matrix: \mathbf{W}_O [HD_H x D]

Computation:

Queries: $\mathbf{Q} = \mathbf{X}\mathbf{W}_Q$ [H x N x D_H]

Keys: $\mathbf{K} = \mathbf{X}\mathbf{W}_K$ [H x N x D_H]

Values: $\mathbf{V} = \mathbf{X}\mathbf{W}_V$ [H x N x D_H]

Similarities: $\mathbf{E} = \mathbf{Q}\mathbf{K}^T / \sqrt{D_Q}$ [H x N x N]

Attention weights: $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$ [H x N x N]

Head outputs: $\mathbf{Y} = \mathbf{A}\mathbf{V}$ [H x N x D_H] \Rightarrow [N x HD_H]

Outputs: $\mathbf{O} = \mathbf{Y}\mathbf{W}_O$ [N x D]

Self-Attention is Four Matrix Multiplies!

Inputs:

Input vectors: \mathbf{X} [N x D]

Key matrix: \mathbf{W}_K [D x HD_H]

Value matrix: \mathbf{W}_V [D x HD_H]

Query matrix: \mathbf{W}_Q [D x HD_H]

Output matrix: \mathbf{W}_O [HD_H x D]

1. QKV Projection

[N x D] [D x 3HD_H] => [N x 3HD_H]

Split and reshape to get \mathbf{Q} , \mathbf{K} , \mathbf{V} each of shape [H x N x D_H]

Computation:

Queries: $\mathbf{Q} = \mathbf{XW}_Q$ [H x N x D_H]

Keys: $\mathbf{K} = \mathbf{XW}_K$ [H x N x D_H]

Values: $\mathbf{V} = \mathbf{XW}_V$ [H x N x D_H]

Similarities: $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$ [H x N x N]

Attention weights: $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$ [H x N x N]

Head outputs: $\mathbf{Y} = \mathbf{AV}$ [H x N x D_H] => [N x HD_H]

Outputs: $\mathbf{O} = \mathbf{YW}_O$ [N x D]

Self-Attention is Four Matrix Multiplies!

Inputs:

Input vectors: \mathbf{X} [N x D]

Key matrix: \mathbf{W}_K [D x HD_H]

Value matrix: \mathbf{W}_V [D x HD_H]

Query matrix: \mathbf{W}_Q [D x HD_H]

Output matrix: \mathbf{W}_O [HD_H x D]

1. QKV Projection

[N x D] [D x 3HD_H] => [N x 3HD_H]

Split and reshape to get \mathbf{Q} , \mathbf{K} , \mathbf{V} each of shape [H x N x D_H]

2. QK Similarity

[H x N x D_H] [H x N x D_H] => [H x N x N]

Computation:

Queries: $\mathbf{Q} = \mathbf{XW}_Q$ [H x N x D_H]

Keys: $\mathbf{K} = \mathbf{XW}_K$ [H x N x D_H]

Values: $\mathbf{V} = \mathbf{XW}_V$ [H x N x D_H]

Similarities: $E = \mathbf{QK}^T / \sqrt{D_Q}$ [H x N x N]

Attention weights: $\mathbf{A} = \text{softmax}(E, \text{dim}=1)$ [H x N x N]

Head outputs: $\mathbf{Y} = \mathbf{AV}$ [H x N x D_H] => [N x HD_H]

Outputs: $\mathbf{O} = \mathbf{YW}_O$ [N x D]

Self-Attention is Four Matrix Multiplies!

Inputs:

Input vectors: \mathbf{X} [N x D]

Key matrix: \mathbf{W}_K [D x HD_H]

Value matrix: \mathbf{W}_V [D x HD_H]

Query matrix: \mathbf{W}_Q [D x HD_H]

Output matrix: \mathbf{W}_O [HD_H x D]

Computation:

Queries: $\mathbf{Q} = \mathbf{X}\mathbf{W}_Q$ [H x N x D_H]

Keys: $\mathbf{K} = \mathbf{X}\mathbf{W}_K$ [H x N x D_H]

Values: $\mathbf{V} = \mathbf{X}\mathbf{W}_V$ [H x N x D_H]

Similarities: $\mathbf{E} = \mathbf{Q}\mathbf{K}^T / \sqrt{D_Q}$ [H x N x N]

Attention weights: $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$ [H x N x N]

Head outputs: $\mathbf{Y} = \mathbf{A}\mathbf{V}$ [H x N x D_H] \Rightarrow [N x HD_H]

Outputs: $\mathbf{O} = \mathbf{Y}\mathbf{W}_O$ [N x D]

1. QKV Projection

[N x D] [D x 3HD_H] \Rightarrow [N x 3HD_H]

Split and reshape to get \mathbf{Q} , \mathbf{K} , \mathbf{V} each of shape [H x N x D_H]

2. QK Similarity

[H x N x D_H] [H x N x D_H] \Rightarrow [H x N x N]

3. V-Weighting

[H x N x N] [H x D x D_H] \Rightarrow [H x N x D_H]

Reshape to [N x HD_H]

Self-Attention is Four Matrix Multiplies!

Inputs:

Input vectors: \mathbf{X} [N x D]

Key matrix: \mathbf{W}_K [D x HD_H]

Value matrix: \mathbf{W}_V [D x HD_H]

Query matrix: \mathbf{W}_Q [D x HD_H]

Output matrix: \mathbf{W}_O [HD_H x D]

Computation:

Queries: $\mathbf{Q} = \mathbf{X}\mathbf{W}_Q$ [H x N x D_H]

Keys: $\mathbf{K} = \mathbf{X}\mathbf{W}_K$ [H x N x D_H]

Values: $\mathbf{V} = \mathbf{X}\mathbf{W}_V$ [H x N x D_H]

Similarities: $E = \mathbf{Q}\mathbf{K}^T / \sqrt{D_Q}$ [H x N x N]

Attention weights: $\mathbf{A} = \text{softmax}(E, \text{dim}=1)$ [H x N x N]

Head outputs: $\mathbf{Y} = \mathbf{AV}$ [H x N x D_H] \Rightarrow [N x HD_H]

Outputs: $\mathbf{O} = \mathbf{YW}_O$ [N x D]

1. QKV Projection

[N x D] [D x 3HD_H] \Rightarrow [N x 3HD_H]

Split and reshape to get \mathbf{Q} , \mathbf{K} , \mathbf{V} each of shape [H x N x D_H]

2. QK Similarity

[H x N x D_H] [H x N x D_H] \Rightarrow [H x N x N]

3. V-Weighting

[H x N x N] [H x D x D_H] \Rightarrow [H x N x D_H]

Reshape to [N x HD_H]

4. Output Projection

[N x HD_H] [HD_H x D] \Rightarrow [N x D]

Self-Attention is Four Matrix Multiplies!

Inputs:

Input vectors: \mathbf{X} [N x D]

Key matrix: \mathbf{W}_K [D x HD_H]

Value matrix: \mathbf{W}_V [D x HD_H]

Query matrix: \mathbf{W}_Q [D x HD_H]

Output matrix: \mathbf{W}_O [HD_H x D]

Computation:

Queries: $\mathbf{Q} = \mathbf{X}\mathbf{W}_Q$ [H x N x D_H]

Keys: $\mathbf{K} = \mathbf{X}\mathbf{W}_K$ [H x N x D_H]

Values: $\mathbf{V} = \mathbf{X}\mathbf{W}_V$ [H x N x D_H]

Similarities: $E = \mathbf{Q}\mathbf{K}^T / \sqrt{D_Q}$ [H x N x N]

Attention weights: $\mathbf{A} = \text{softmax}(E, \text{dim}=1)$ [H x N x N]

Head outputs: $\mathbf{Y} = \mathbf{AV}$ [H x N x D_H] => [N x HD_H]

Outputs: $\mathbf{O} = \mathbf{YW}_O$ [N x D]

1. QKV Projection

[N x D] [D x 3HD_H] => [N x 3HD_H]

Split and reshape to get \mathbf{Q} , \mathbf{K} , \mathbf{V} each of shape [H x N x D_H]

2. QK Similarity

[H x N x D_H] [H x N x D_H] => [H x N x N]

3. V-Weighting

[H x N x N] [H x D x D_H] => [H x N x D_H]

Reshape to [N x HD_H]

4. Output Projection

[N x HD_H] [HD_H x D] => [N x D]

Q: How much compute does this take as the number of vectors N increases?

Self-Attention is Four Matrix Multiplies!

Inputs:

Input vectors: \mathbf{X} [N x D]

Key matrix: \mathbf{W}_K [D x HD_H]

Value matrix: \mathbf{W}_V [D x HD_H]

Query matrix: \mathbf{W}_Q [D x HD_H]

Output matrix: \mathbf{W}_O [HD_H x D]

Computation:

Queries: $\mathbf{Q} = \mathbf{X}\mathbf{W}_Q$ [H x N x D_H]

Keys: $\mathbf{K} = \mathbf{X}\mathbf{W}_K$ [H x N x D_H]

Values: $\mathbf{V} = \mathbf{X}\mathbf{W}_V$ [H x N x D_H]

Similarities: $E = \mathbf{Q}\mathbf{K}^T / \sqrt{D_Q}$ [H x N x N]

Attention weights: $\mathbf{A} = \text{softmax}(E, \text{dim}=1)$ [H x N x N]

Head outputs: $\mathbf{Y} = \mathbf{AV}$ [H x N x D_H] => [N x HD_H]

Outputs: $\mathbf{O} = \mathbf{YW}_O$ [N x D]

1. QKV Projection

[N x D] [D x 3HD_H] => [N x 3HD_H]

Split and reshape to get \mathbf{Q} , \mathbf{K} , \mathbf{V} each of shape [H x N x D_H]

2. QK Similarity

[H x N x D_H] [H x N x D_H] => [H x N x N]

3. V-Weighting

[H x N x N] [H x D x D_H] => [H x N x D_H]

Reshape to [N x HD_H]

4. Output Projection

[N x HD_H] [HD_H x D] => [N x D]

Q: How much compute does this take as the number of vectors N increases?

A: O(N²)

Self-Attention is Four Matrix Multiplies!

Inputs:

Input vectors: \mathbf{X} [N x D]

Key matrix: \mathbf{W}_K [D x HD_H]

Value matrix: \mathbf{W}_V [D x HD_H]

Query matrix: \mathbf{W}_Q [D x HD_H]

Output matrix: \mathbf{W}_O [HD_H x D]

Computation:

Queries: $\mathbf{Q} = \mathbf{X}\mathbf{W}_Q$ [H x N x D_H]

Keys: $\mathbf{K} = \mathbf{X}\mathbf{W}_K$ [H x N x D_H]

Values: $\mathbf{V} = \mathbf{X}\mathbf{W}_V$ [H x N x D_H]

Similarities: $E = \mathbf{Q}\mathbf{K}^T / \sqrt{D_Q}$ [H x N x N]

Attention weights: $\mathbf{A} = \text{softmax}(E, \text{dim}=1)$ [H x N x N]

Head outputs: $\mathbf{Y} = \mathbf{AV}$ [H x N x D_H] => [N x HD_H]

Outputs: $\mathbf{O} = \mathbf{YW}_O$ [N x D]

1. QKV Projection

[N x D] [D x 3HD_H] => [N x 3HD_H]

Split and reshape to get \mathbf{Q} , \mathbf{K} , \mathbf{V} each of shape [H x N x D_H]

2. QK Similarity

[H x N x D_H] [H x N x D_H] => [H x N x N]

3. V-Weighting

[H x N x N] [H x D x D_H] => [H x N x D_H]

Reshape to [N x HD_H]

4. Output Projection

[N x HD_H] [HD_H x D] => [N x D]

Q: How much memory does this take as the number of vectors N increases?

Self-Attention is Four Matrix Multiplies!

Inputs:

Input vectors: \mathbf{X} [N x D]

Key matrix: \mathbf{W}_K [D x HD_H]

Value matrix: \mathbf{W}_V [D x HD_H]

Query matrix: \mathbf{W}_Q [D x HD_H]

Output matrix: \mathbf{W}_O [HD_H x D]

Computation:

Queries: $\mathbf{Q} = \mathbf{X}\mathbf{W}_Q$ [H x N x D_H]

Keys: $\mathbf{K} = \mathbf{X}\mathbf{W}_K$ [H x N x D_H]

Values: $\mathbf{V} = \mathbf{X}\mathbf{W}_V$ [H x N x D_H]

Similarities: $E = \mathbf{Q}\mathbf{K}^T / \sqrt{D_Q}$ [H x N x N]

Attention weights: $\mathbf{A} = \text{softmax}(E, \text{dim}=1)$ [H x N x N]

Head outputs: $\mathbf{Y} = \mathbf{AV}$ [H x N x D_H] => [N x HD_H]

Outputs: $\mathbf{O} = \mathbf{YW}_O$ [N x D]

1. QKV Projection

[N x D] [D x 3HD_H] => [N x 3HD_H]

Split and reshape to get \mathbf{Q} , \mathbf{K} , \mathbf{V} each of shape [H x N x D_H]

2. QK Similarity

[H x N x D_H] [H x N x D_H] => [H x N x N]

3. V-Weighting

[H x N x N] [H x D x D_H] => [H x N x D_H]

Reshape to [N x HD_H]

4. Output Projection

[N x HD_H] [HD_H x D] => [N x D]

Q: How much memory does this take as the number of vectors N increases?

A: O(N²)

Self-Attention is Four Matrix Multiplies!

If $N=100K$, $H=64$ then
 $H \times N \times N$ attention weights
take 1.192 TB! GPUs don't
have that much memory...

Inputs:

Input vectors: \mathbf{X} [$N \times D$]

Key matrix: \mathbf{W}_K [$D \times HD_H$]

Value matrix: \mathbf{W}_V [$D \times HD_H$]

Query matrix: \mathbf{W}_Q [$D \times HD_H$]

Output matrix: \mathbf{W}_O [$HD_H \times D$]

Computation:

Queries: $\mathbf{Q} = \mathbf{XW}_Q$ [$H \times N \times D_H$]

Keys: $\mathbf{K} = \mathbf{XW}_K$ [$H \times N \times D_H$]

Values: $\mathbf{V} = \mathbf{XW}_V$ [$H \times N \times D_H$]

Similarities: $E = \mathbf{QK}^T / \sqrt{D_Q}$ [$H \times N \times N$]

Attention weights: $\mathbf{A} = \text{softmax}(E, \text{dim}=1)$ [$H \times N \times N$]

Head outputs: $\mathbf{Y} = \mathbf{AV}$ [$H \times N \times D_H$] \Rightarrow [$N \times HD_H$]

Outputs: $\mathbf{O} = \mathbf{YW}_O$ [$N \times D$]

1. QKV Projection

$[N \times D]$ [$D \times 3HD_H$] \Rightarrow [$N \times 3HD_H$]

Split and reshape to get \mathbf{Q} , \mathbf{K} , \mathbf{V} each of
shape [$H \times N \times D_H$]

2. QK Similarity

$[H \times N \times D_H]$ $[H \times N \times D_H]$ \Rightarrow $[H \times N \times N]$

3. V-Weighting

$[H \times N \times N]$ $[H \times D \times D_H]$ \Rightarrow $[H \times N \times D_H]$

Reshape to $[N \times HD_H]$

4. Output Projection

$[N \times HD_H]$ $[HD_H \times D]$ \Rightarrow [$N \times D$]

Q: How much memory does this take
as the number of vectors N increases?

A: $O(N^2)$

Self-Attention is Four Matrix Multiplies!

Inputs:

Input vectors: \mathbf{X} [N x D]

Key matrix: \mathbf{W}_K [D x HD_H]

Value matrix: \mathbf{W}_V [D x HD_H]

Query matrix: \mathbf{W}_Q [D x HD_H]

Output matrix: \mathbf{W}_O [HD_H x D]

Flash Attention

algorithm computes
2+3 at the same time
without storing the
full attention matrix!

Computation:

Queries: $\mathbf{Q} = \mathbf{XW}_Q$ [H x N x D_H]

Keys: $\mathbf{K} = \mathbf{XW}_K$ [H x N x D_H]

Values: $\mathbf{V} = \mathbf{XW}_V$ [H x N x D_H]

Similarities: $E = \mathbf{QK}^T / \sqrt{D_Q}$ [H x N x N]

Attention weights: $\mathbf{A} = \text{softmax}(E, \text{dim}=1)$ [H x N x N]

Head outputs: $\mathbf{Y} = \mathbf{AV}$ [H x N x D_H] \Rightarrow [N x HD_H]

Outputs: $\mathbf{O} = \mathbf{YW}_O$ [N x D]

If N=100K, H=64 then
HxNxN attention weights
take 1.192 TB! GPUs don't
have that much memory...

QKV Projection

[N x D] [D x 3HD_H] \Rightarrow [N x 3HD_H]

Split and reshape to get \mathbf{Q} , \mathbf{K} , \mathbf{V} each of
shape [H x N x D_H]

QK Similarity

[H x N x D_H] [H x N x D_H] \Rightarrow [H x N x N]

V-Weighting

[H x N x N] [H x D x D_H] \Rightarrow [H x N x D_H]

Reshape to [N x HD_H]

Output Projection

[N x HD_H] [HD_H x D] \Rightarrow [N x D]

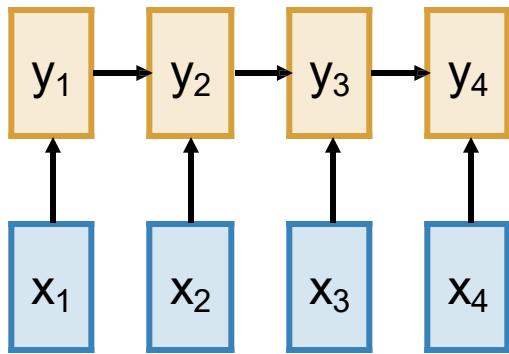
Q: How much memory does this take
as the number of vectors N increases?

A: O(N) with Flash Attention

Three Ways of Processing Sequences

Three Ways of Processing Sequences

Recurrent Neural Network

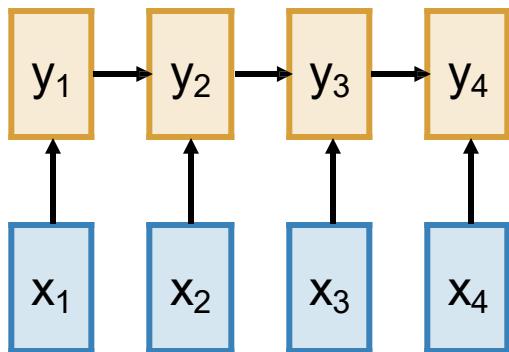


Works on **1D ordered sequences**

- (+) Theoretically good at long sequences: $O(N)$ compute and memory for a sequence of length N
- (-) Not parallelizable. Need to compute hidden states sequentially

Three Ways of Processing Sequences

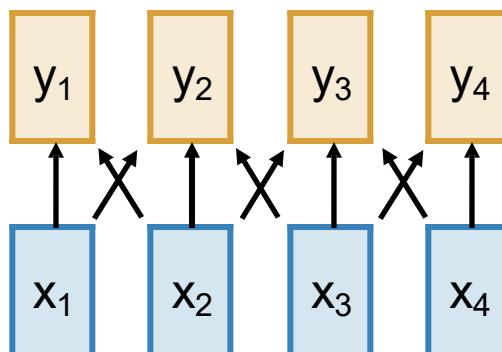
Recurrent Neural Network



Works on **1D ordered sequences**

- (+) Theoretically good at long sequences: $O(N)$ compute and memory for a sequence of length N
- (-) Not parallelizable. Need to compute hidden states sequentially

Convolution

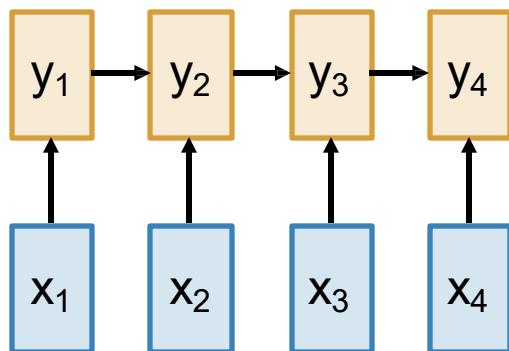


Works on **N-dimensional grids**

- (-) Bad for long sequences: need to stack many layers to build up large receptive fields
- (+) Parallelizable, outputs can be computed in parallel

Three Ways of Processing Sequences

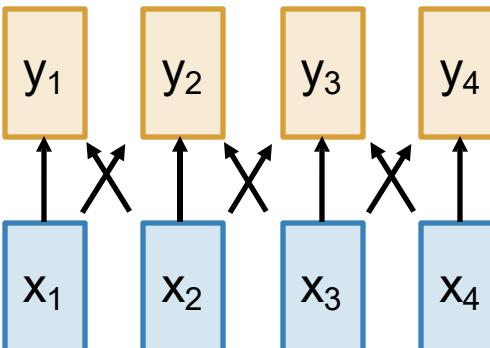
Recurrent Neural Network



Works on **1D ordered sequences**

- (+) Theoretically good at long sequences: $O(N)$ compute and memory for a sequence of length N
- (-) Not parallelizable. Need to compute hidden states sequentially

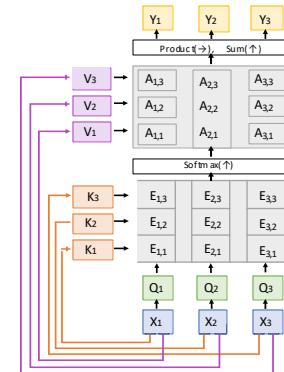
Convolution



Works on **N-dimensional grids**

- (-) Bad for long sequences: need to stack many layers to build up large receptive fields
- (+) Parallelizable, outputs can be computed in parallel

Self-Attention



Works on **sets of vectors**

- (+) Great for long sequences; each output depends directly on all inputs
- (+) Highly parallel, it's just 4 matmuls
- (-) Expensive: $O(N^2)$ compute, $O(N)$ memory for sequence of length N

Three Ways of Processing Sequences

Recurrent Neural Network

Convolution

Self-Attention



Attention is All You Need

Vaswani et al, NeurIPS 2017

Sequences. $O(N)$ compute and memory for a sequence of length N
(-) Not parallelizable. Need to compute hidden states sequentially

Stack many layers to build up large receptive fields
(+) Parallelizable, outputs can be computed in parallel

Output depends directly on all inputs
(+) Highly parallel, it's just 4 matmuls
(-) Expensive: $O(N^2)$ compute, $O(N)$ memory for sequence of length N

The Transformer

Transformer Block

Input: Set of vectors x

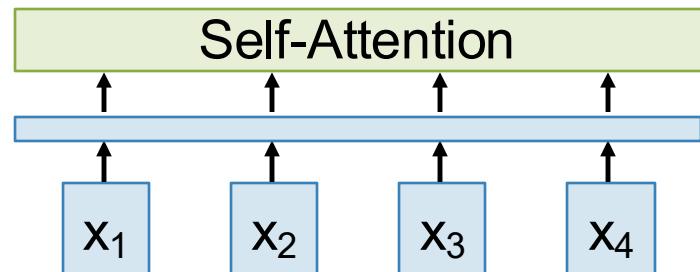


The Transformer

Transformer Block

Input: Set of vectors x

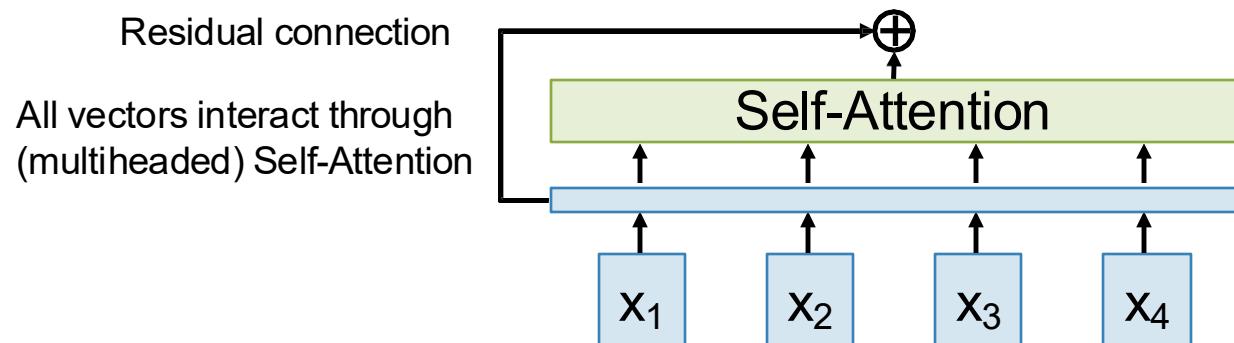
All vectors interact through
(multiheaded) Self-Attention



The Transformer

Transformer Block

Input: Set of vectors x



The Transformer

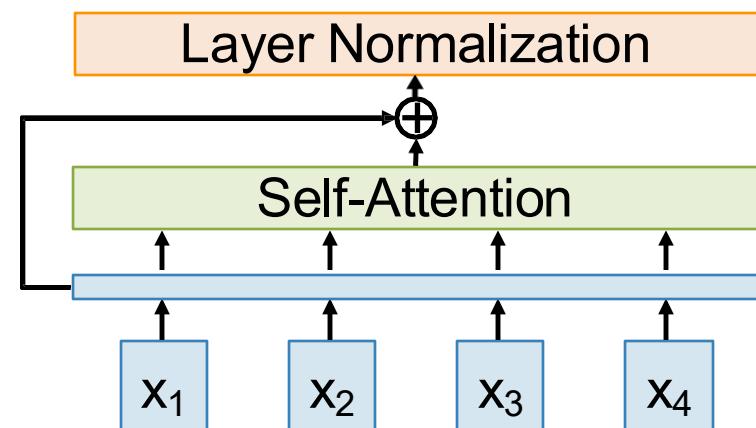
Transformer Block

Input: Set of vectors x

Layer normalization
normalizes all vectors
Residual connection
All vectors interact through
(multiheaded) Self-Attention

Recall **Layer Normalization**:
Given h_1, \dots, h_N (Shape: D)
scale: γ (Shape: D)
shift: β (Shape: D)
 $\mu_i = (\sum_j h_{i,j})/D$ (scalar)
 $\sigma_i = (\sum_j (h_{i,j} - \mu_i)^2/D)^{1/2}$ (scalar)
 $z_i = (h_i - \mu_i) / \sigma_i$
 $y_i = \gamma * z_i + \beta$

Ba et al, 2016



The Transformer

Transformer Block

Input: Set of vectors x

MLP independently
on each vector

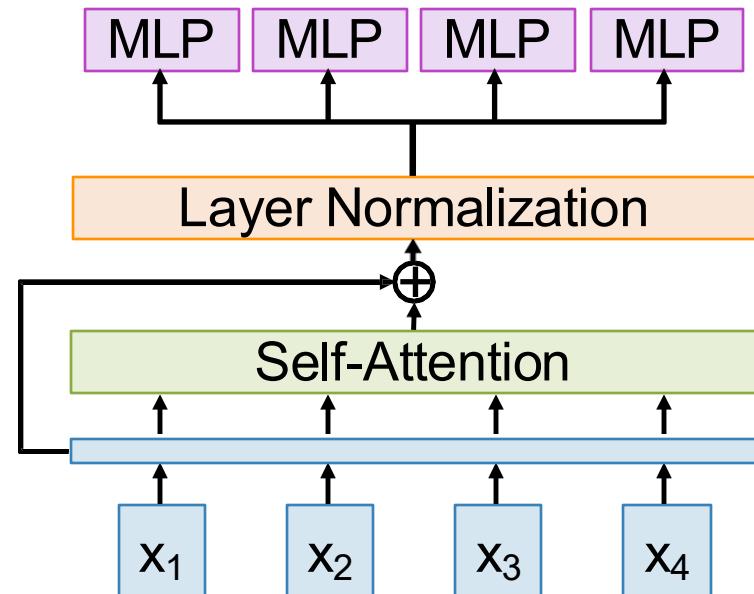
Layer normalization
normalizes all vectors

Residual connection

All vectors interact through
(multiheaded) Self-Attention

Usually a two-layer MLP;
classic setup is
 $D \Rightarrow 4D \Rightarrow D$

Also sometimes called FFN
(Feed-Forward Network)



The Transformer

Transformer Block

Input: Set of vectors x

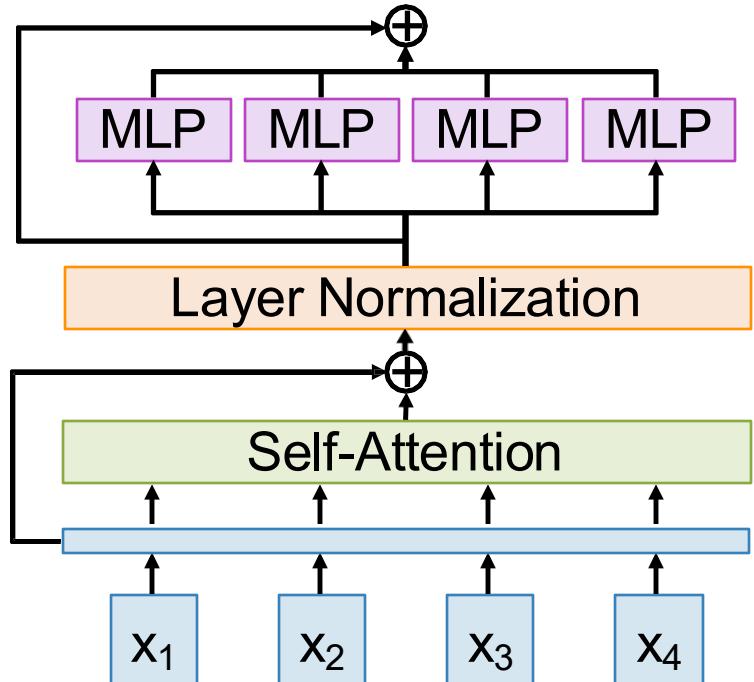
Residual connection

MLP independently
on each vector

Layer normalization
normalizes all vectors

Residual connection

All vectors interact through
(multiheaded) Self-Attention



The Transformer

Transformer Block

Input: Set of vectors x

Another Layer Norm

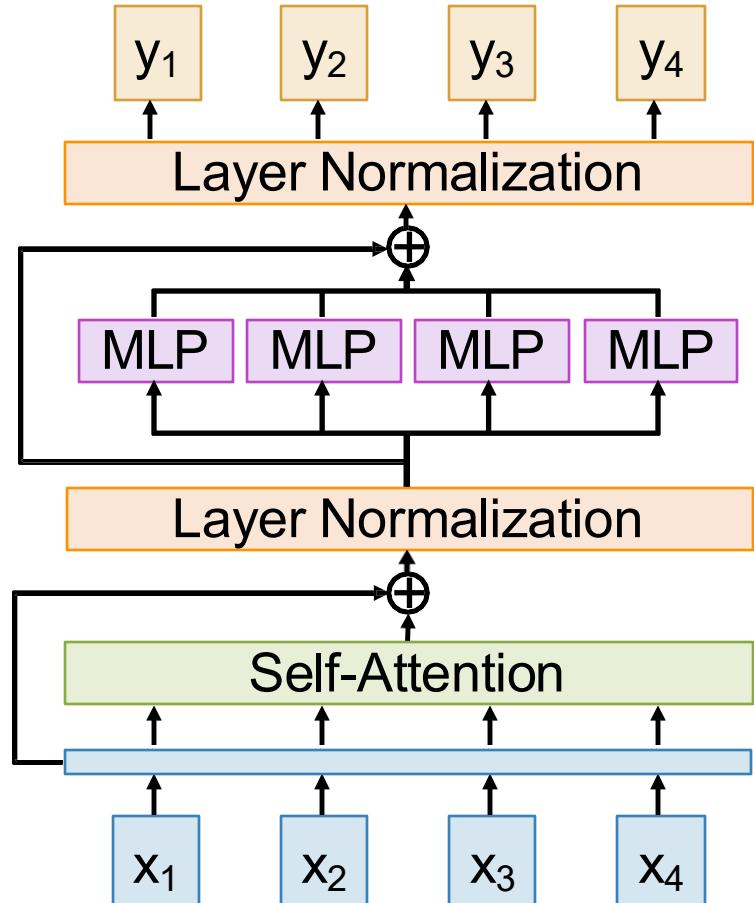
Residual connection

MLP independently
on each vector

Layer normalization
normalizes all vectors

Residual connection

All vectors interact through
(multiheaded) Self-Attention



The Transformer

Transformer Block

Input: Set of vectors x

Output: Set of vectors y

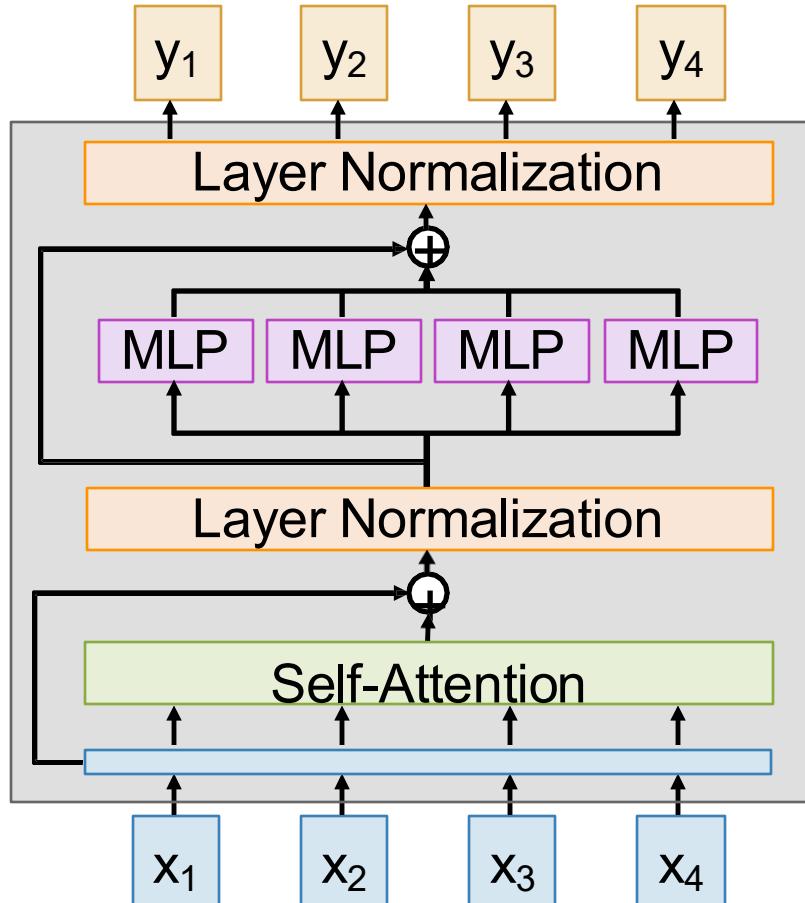
Self-Attention is the only interaction between vectors

LayerNorm and MLP work on each vector independently

Highly scalable and parallelizable, most of the compute is just 6 matmuls:

4 from Self-Attention

2 from MLP



The Transformer

Transformer Block

Input: Set of vectors x

Output: Set of vectors y

Self-Attention is the only interaction between vectors

LayerNorm and MLP work on each vector independently

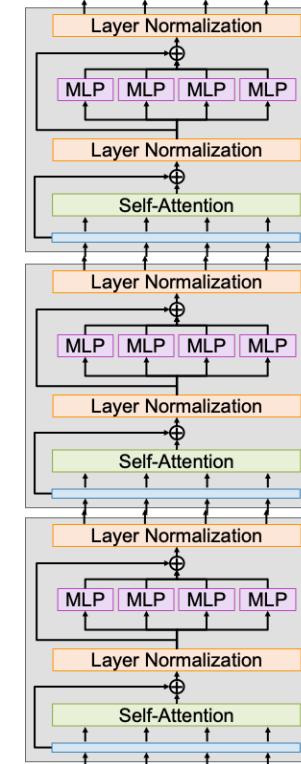
Highly scalable and parallelizable, most of the compute is just 6 matmuls:

4 from Self-Attention

2 from MLP

A **Transformer** is just a stack of identical Transformer blocks!

They have not changed much since 2017... but have gotten a lot bigger



The Transformer

Transformer Block

Input: Set of vectors x

Output: Set of vectors y

Self-Attention is the only interaction between vectors

LayerNorm and MLP work on each vector independently

Highly scalable and parallelizable, most of the compute is just 6 matmuls:

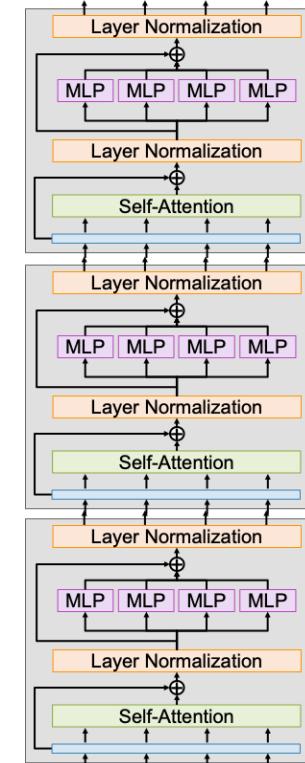
4 from Self-Attention

2 from MLP

A **Transformer** is just a stack of identical Transformer blocks!

They have not changed much since 2017... but have gotten a lot bigger

Original: [Vaswani et al, 2017]
12 blocks, D=1024, H=16, N=512
213M params



The Transformer

Transformer Block

Input: Set of vectors x

Output: Set of vectors y

Self-Attention is the only interaction between vectors

LayerNorm and MLP work on each vector independently

Highly scalable and parallelizable, most of the compute is just 6 matmuls:

4 from Self-Attention

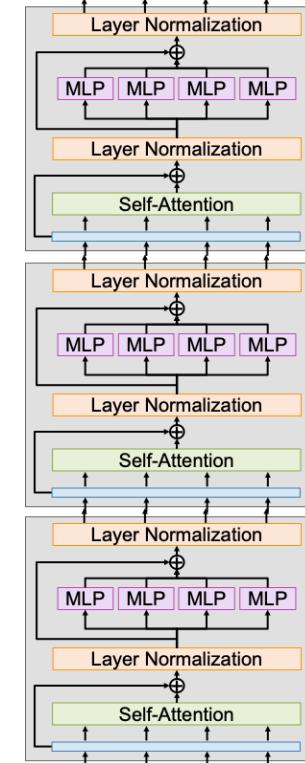
2 from MLP

A **Transformer** is just a stack of identical Transformer blocks!

They have not changed much since 2017... but have gotten a lot bigger

Original: [Vaswani et al, 2017]
12 blocks, D=1024, H=16, N=512
213M params

GPT-2: [Radford et al, 2019]
48 blocks, D=1600, H=25, N=1024
1.5B params



The Transformer

Transformer Block

Input: Set of vectors x

Output: Set of vectors y

Self-Attention is the only interaction between vectors

LayerNorm and MLP work on each vector independently

Highly scalable and parallelizable, most of the compute is just 6 matmuls:

4 from Self-Attention

2 from MLP

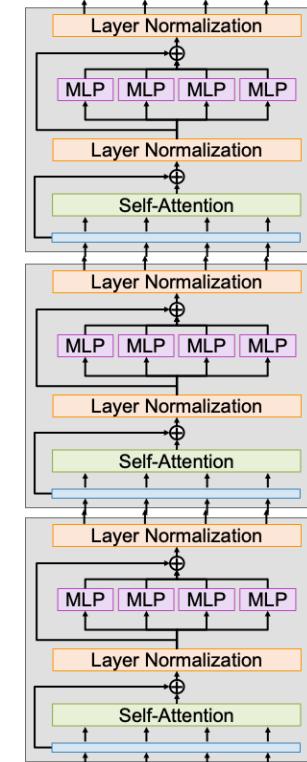
A **Transformer** is just a stack of identical Transformer blocks!

They have not changed much since 2017... but have gotten a lot bigger

Original: [Vaswani et al, 2017]
12 blocks, D=1024, H=16, N=512
213M params

GPT-2: [Radford et al, 2019]
48 blocks, D=1600, H=25, N=1024
1.5B params

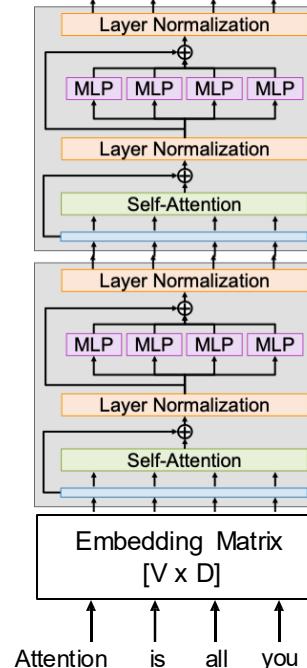
GPT-3: [Brown et al, 2020]
96 blocks, D=12288, H=96, N=2048
175B params



Transformers for Language Modeling (LLM)

Learn an embedding matrix at the start of the model to convert words into vectors.

Given vocab size V and model dimension D , it's a lookup table of shape $[V \times D]$

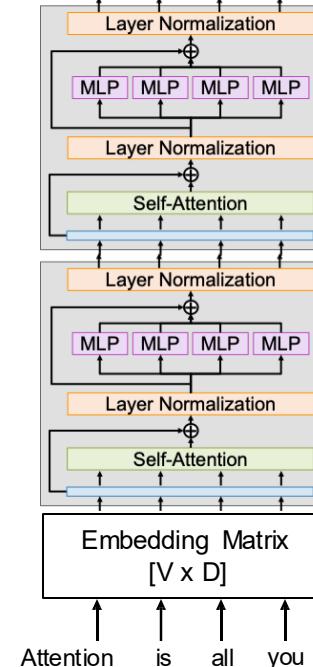


Transformers for Language Modeling (LLM)

Learn an embedding matrix at the start of the model to convert words into vectors.

Given vocab size V and model dimension D , it's a lookup table of shape $[V \times D]$

Use masked attention inside each transformer block so each token can only see the ones before it



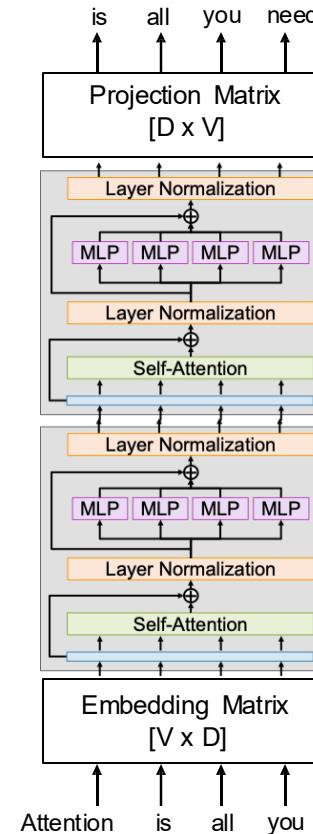
Transformers for Language Modeling (LLM)

Learn an embedding matrix at the start of the model to convert words into vectors.

Given vocab size V and model dimension D , it's a lookup table of shape $[V \times D]$

Use masked attention inside each transformer block so each token can only see the ones before it

At the end, learn a projection matrix of shape $[D \times V]$ to project each D -dim vector to a V -dim vector of scores for each element of the vocabulary.



Transformers for Language Modeling (LLM)

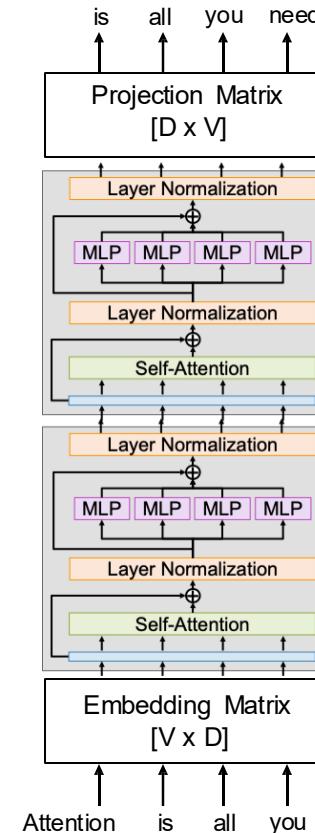
Learn an embedding matrix at the start of the model to convert words into vectors.

Given vocab size V and model dimension D , it's a lookup table of shape $[V \times D]$

Use masked attention inside each transformer block so each token can only see the ones before it

At the end, learn a projection matrix of shape $[D \times V]$ to project each D -dim vector to a V -dim vector of scores for each element of the vocabulary.

Train to predict next token using softmax + cross-entropy loss



Vision Transformers (ViT)



Input image:
e.g. 224x224x3

Vision Transformers (ViT)



Input image:
e.g. 224x224x3

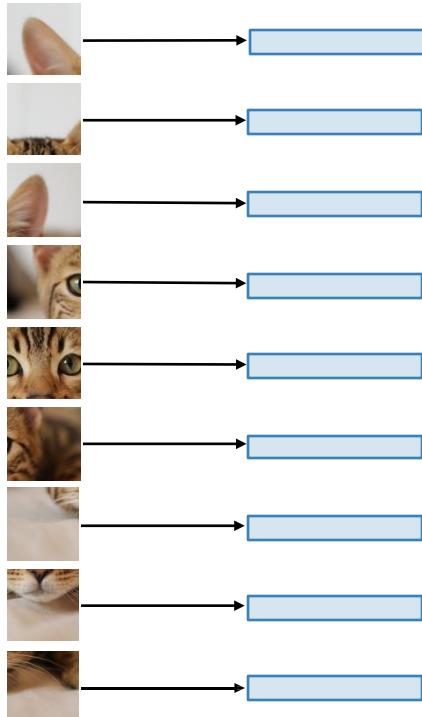


Break into patches
e.g. 16x16x3

Vision Transformers (ViT)



Input image:
e.g. 224x224x3



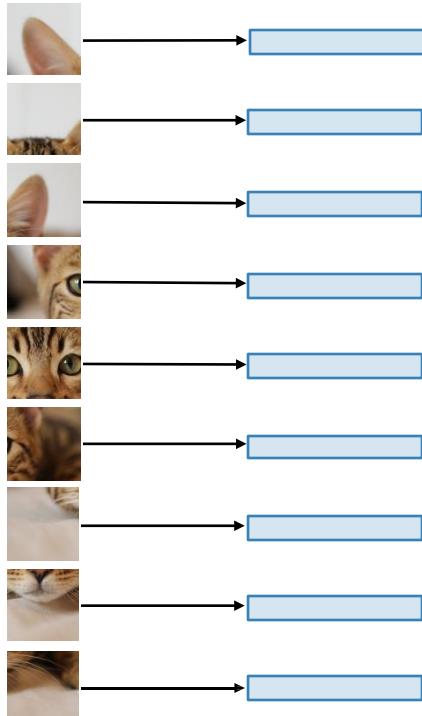
Break into patches
e.g. 16x16x3

Flatten and apply a linear
transform 768 => D

Vision Transformers (ViT)



Input image:
e.g. 224x224x3



Break into patches
e.g. 16x16x3

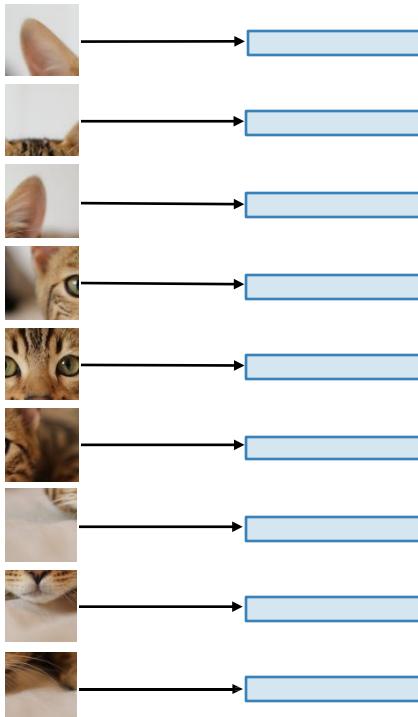
Flatten and apply a linear
transform $768 \Rightarrow D$

Q: Any other way to
describe this operation?

Vision Transformers (ViT)



Input image:
e.g. 224x224x3



Break into patches
e.g. 16x16x3

Flatten and apply a linear
transform 768 => D

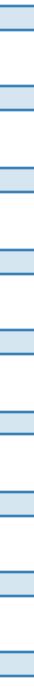
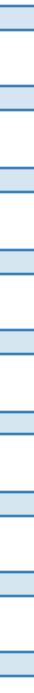
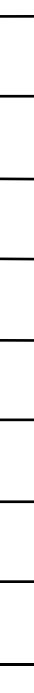
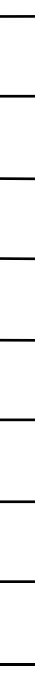
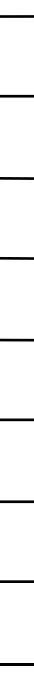
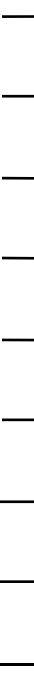
Q: Any other way to
describe this operation?

A: 16x16 conv with stride
16, 3 input channels, D
output channels

Vision Transformers (ViT)

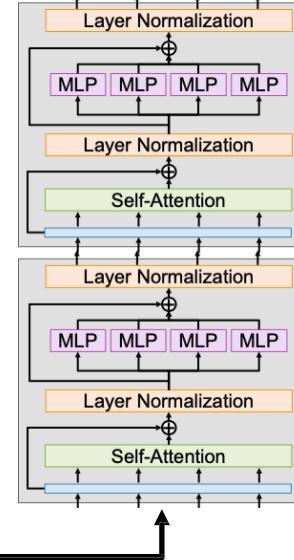


Input image:
e.g. 224x224x3



Break into patches
e.g. 16x16x3

Flatten and apply a linear
transform 768 => D

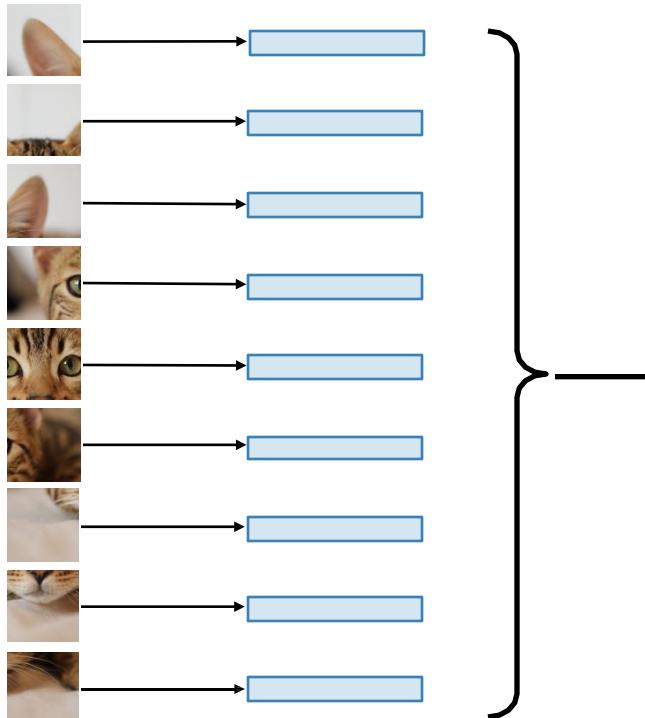


D-dim vector per patch
are the input vectors to
the Transformer

Vision Transformers (ViT)

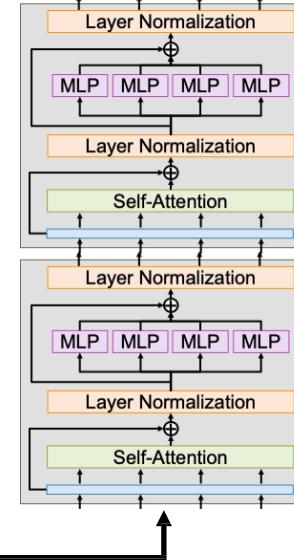


Input image:
e.g. 224x224x3



Break into patches
e.g. 16x16x3

Flatten and apply a linear
transform $768 \Rightarrow D$



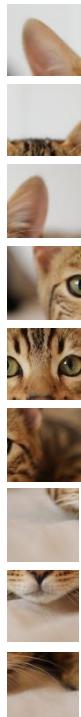
D-dim vector per patch
are the input vectors to
the Transformer

Use positional
encoding to tell
the transformer
the 2D position
of each patch

Vision Transformers (ViT)



Input image:
e.g. 224x224x3



→



→



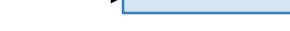
→



→



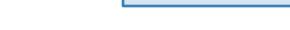
→



→



→



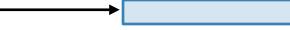
→



→

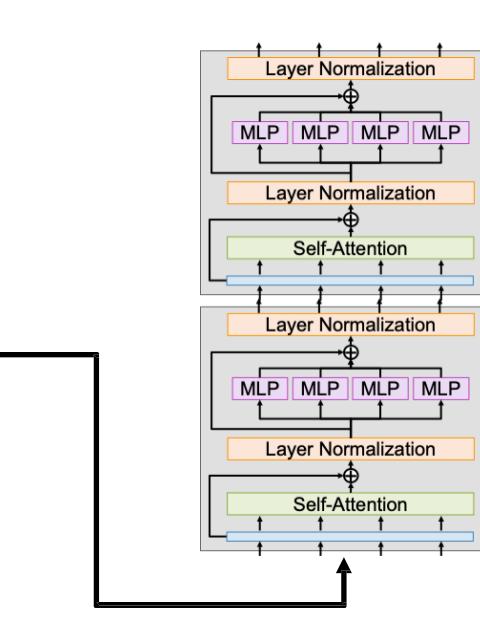


→



Break into patches
e.g. 16x16x3

Flatten and apply a linear
transform $768 \Rightarrow D$



D-dim vector per patch
are the input vectors to
the Transformer

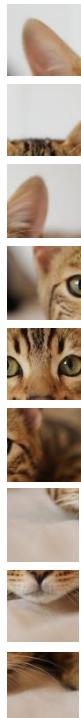
Don't use any
masking; each
image patch can
look at all other
image patches

Use positional
encoding to tell
the transformer
the 2D position
of each patch

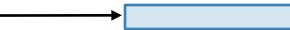
Vision Transformers (ViT)



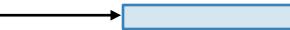
Input image:
e.g. 224x224x3



→



→



→



→



→



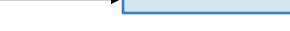
→



→



→



→

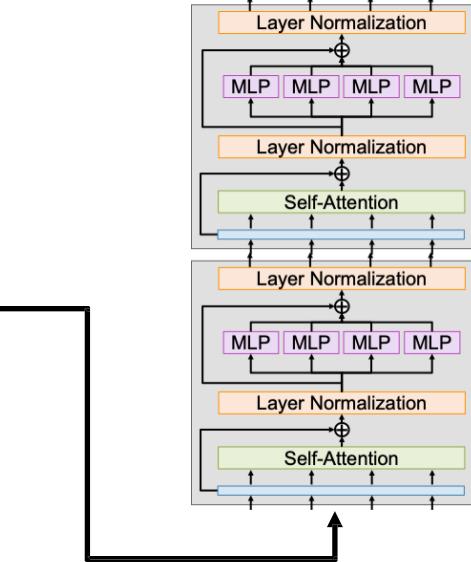


→



Break into patches
e.g. 16x16x3

Flatten and apply a linear
transform $768 \Rightarrow D$



D-dim vector per patch
are the input vectors to
the Transformer

Transformer
gives an output
vector per patch

Don't use any
masking; each
image patch can
look at all other
image patches

Use positional
encoding to tell
the transformer
the 2D position
of each patch

Vision Transformers (ViT)



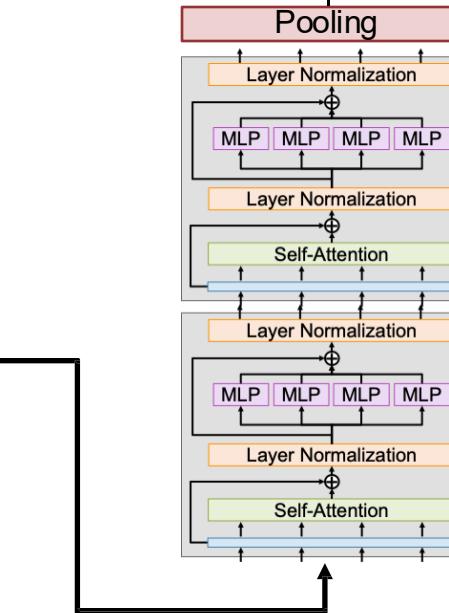
Input image:
e.g. 224x224x3



Break into patches
e.g. 16x16x3

Flatten and apply a linear
transform $768 \Rightarrow D$

Average pool NxD vectors to
1xD, apply a linear layer
 $D \Rightarrow C$ to predict class scores



D-dim vector per patch
are the input vectors to
the Transformer

Transformer
gives an output
vector per patch

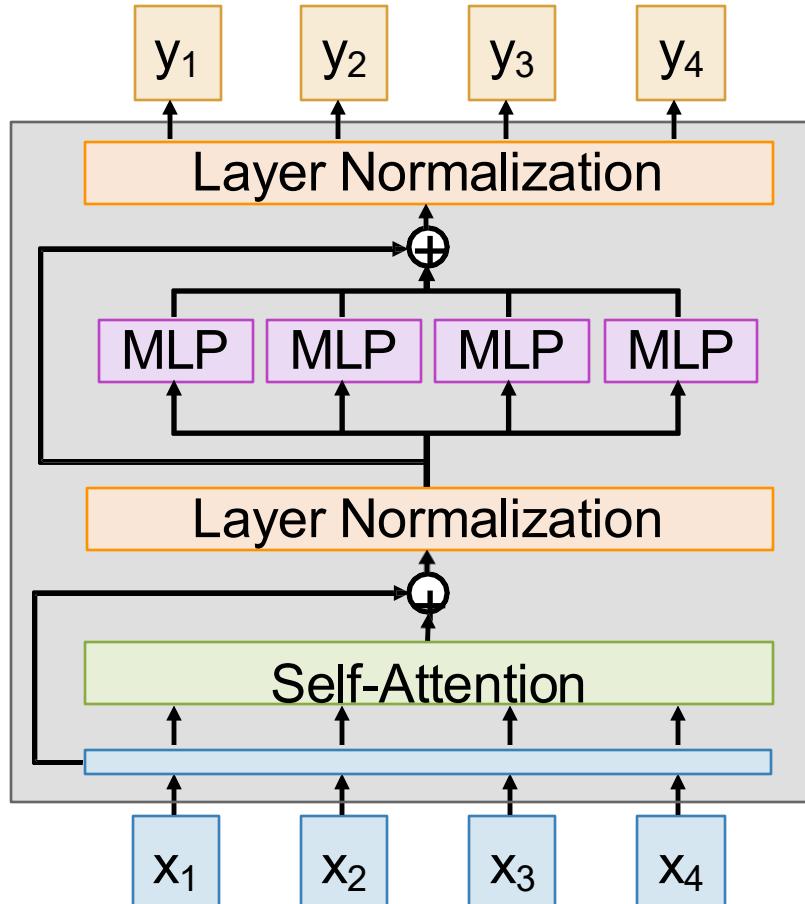
Don't use any
masking; each
image patch can
look at all other
image patches

Use positional
encoding to tell
the transformer
the 2D position
of each patch

Tweaking Transformers

The Transformer architecture has not changed much since 2017.

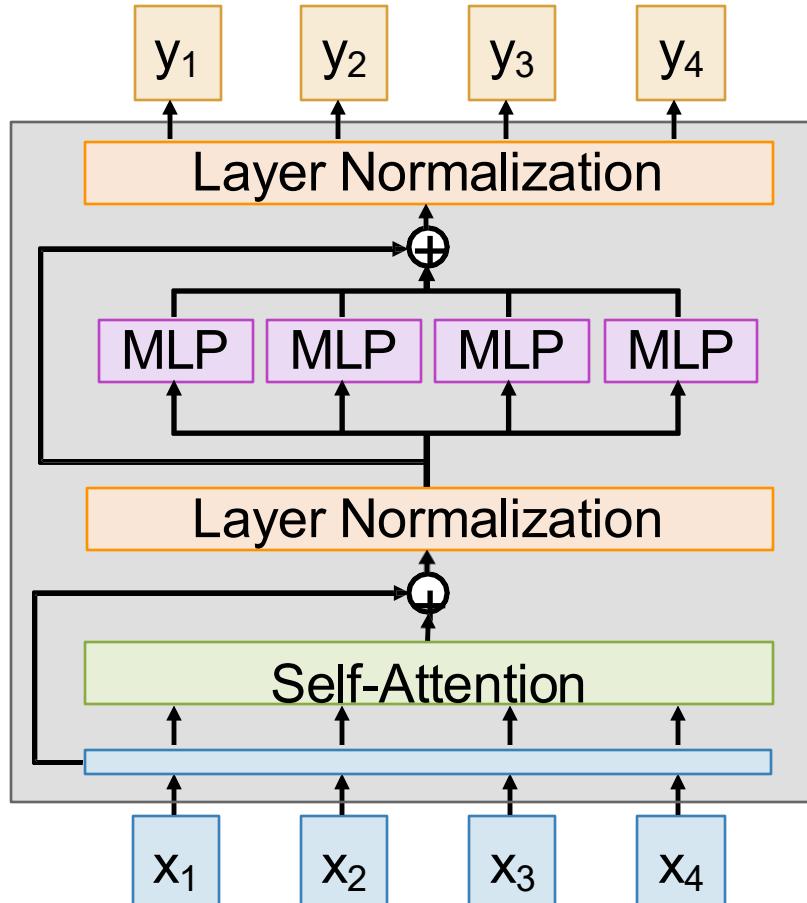
But a few changes have become common:



Pre-Norm Transformer

Layer normalization is outside the residual connections

Kind of weird, the model can't actually learn the identity function

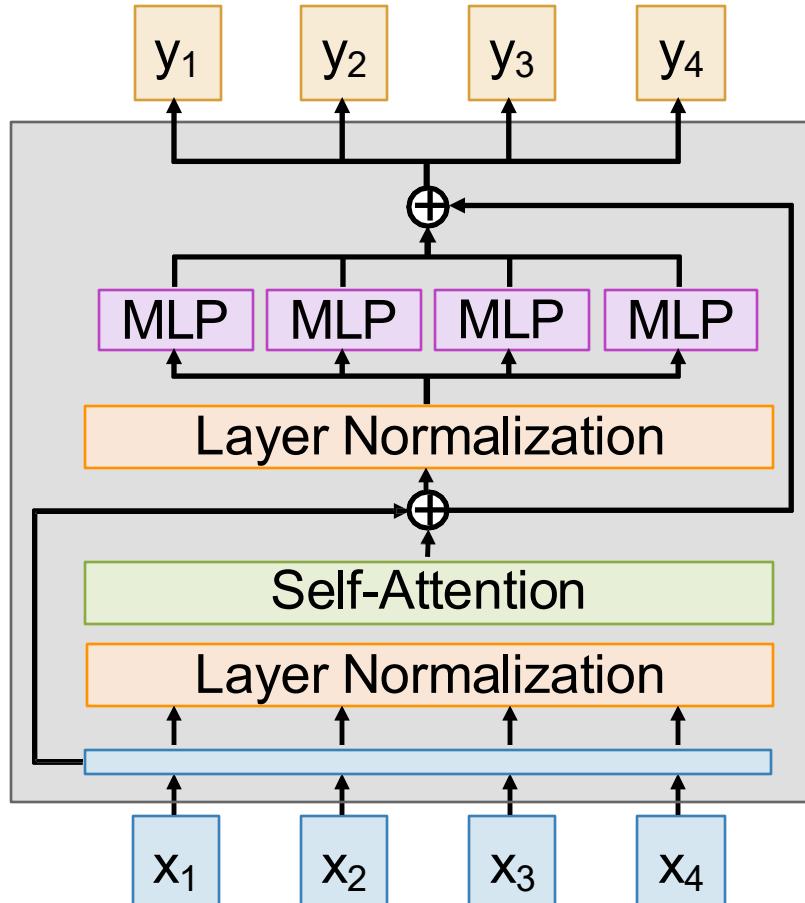


Pre-Norm Transformer

Layer normalization is outside the residual connections

Kind of weird, the model can't actually learn the identity function

Solution: Move layer normalization before the Self-Attention and MLP, inside the residual connections. Training is more stable.



RMSNorm

Replace Layer Normalization
with Root-Mean-Square
Normalization (RMSNorm)

Input: x [shape D]

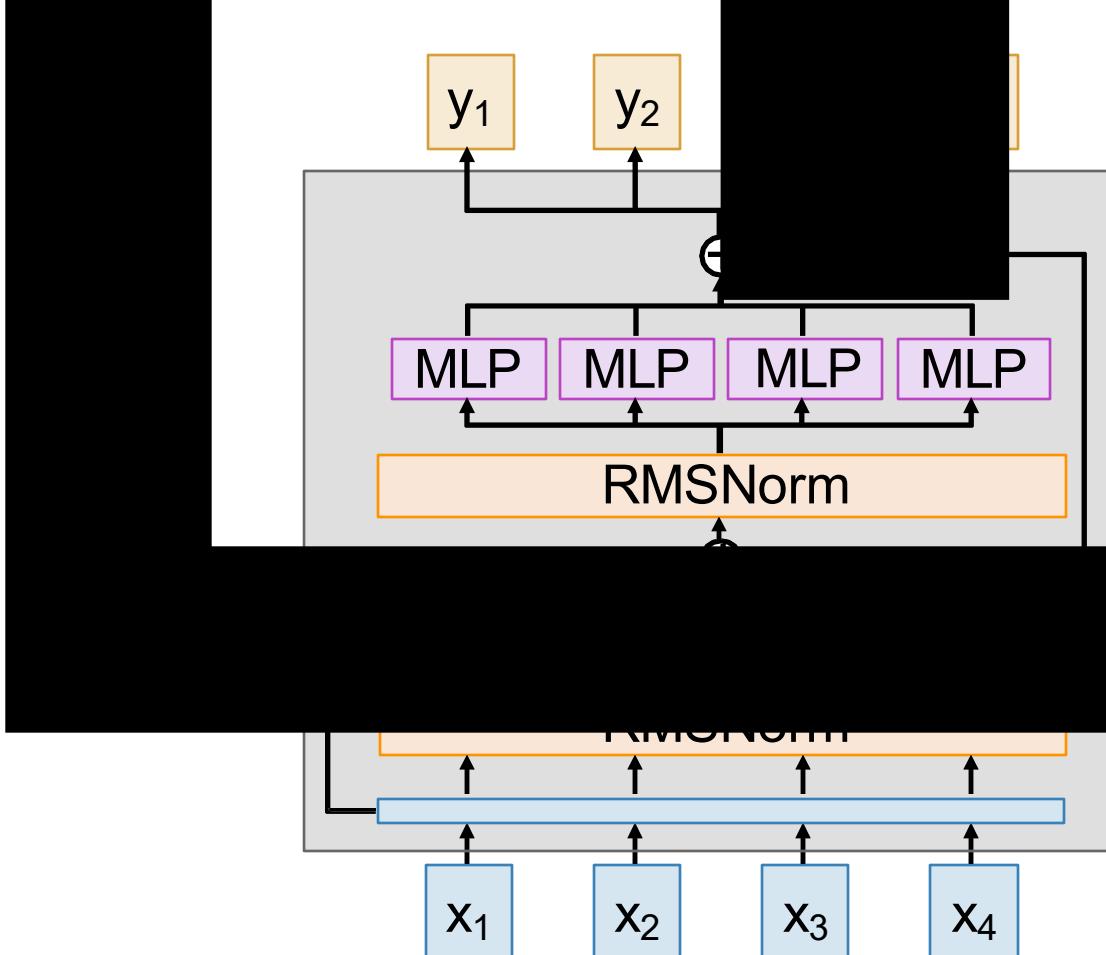
Output: y [shape D]

Weight: γ [shape D]

$$y_i = \frac{x_i}{RMS(x)} * \gamma_i$$

$$RMS(x) = \sqrt{\varepsilon + \frac{1}{N} \sum_{i=1}^N x_i^2}$$

Training is a bit more stable



SwiGLU MLP

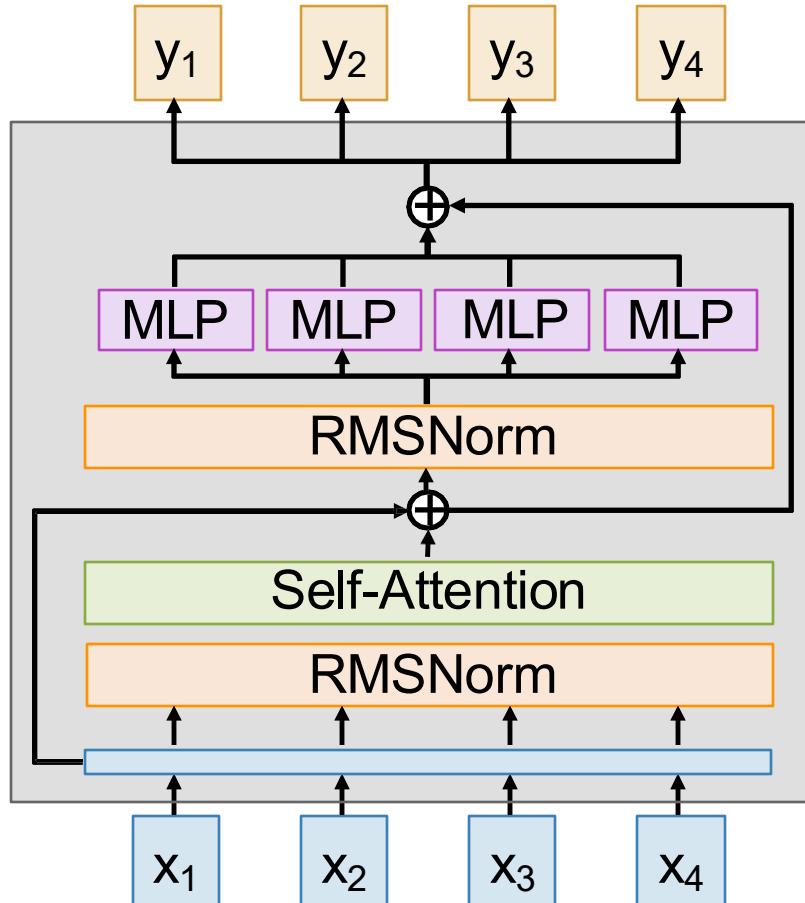
Classic MLP:

Input: X [N x D]

Weights: W_1 [D x 4D]

W_2 [4D x D]

Output: $Y = \sigma(XW_1)W_2$ [N x D]



SwiGLU MLP

Classic MLP:

Input: $X [N \times D]$

Weights: $W_1 [D \times 4D]$

$W_2 [4D \times D]$

Output: $Y = \sigma(XW_1)W_2 [N \times D]$

SwiGLU MLP:

Input: $X [N \times D]$

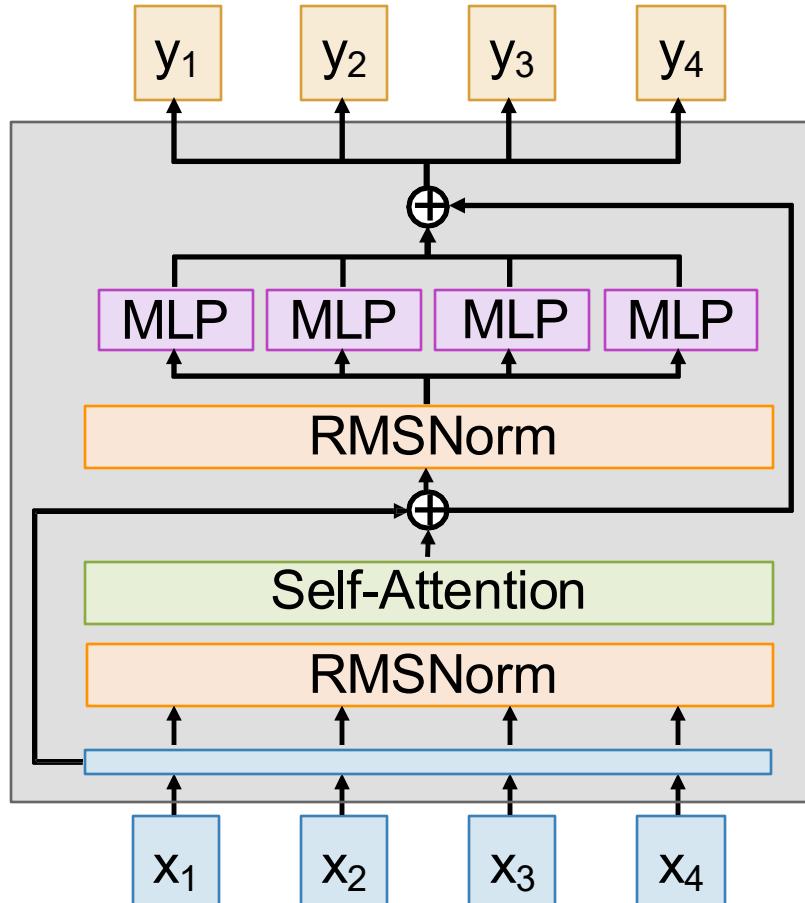
Weights: $W_1, W_2 [D \times H]$

$W_3 [H \times D]$

Output:

$$Y = (\sigma(XW_1) \odot XW_2)W_3$$

Setting $H = 8D/3$ keeps
same total params



SwiGLU MLP

Classic MLP:

Input: $X [N \times D]$

Weights: $W_1 [D \times 4D]$

$W_2 [4D \times D]$

Output: $Y = \sigma(XW_1)W_2 [N \times D]$

SwiGLU MLP:

Input: $X [N \times D]$

Weights: $W_1, W_2 [D \times H]$

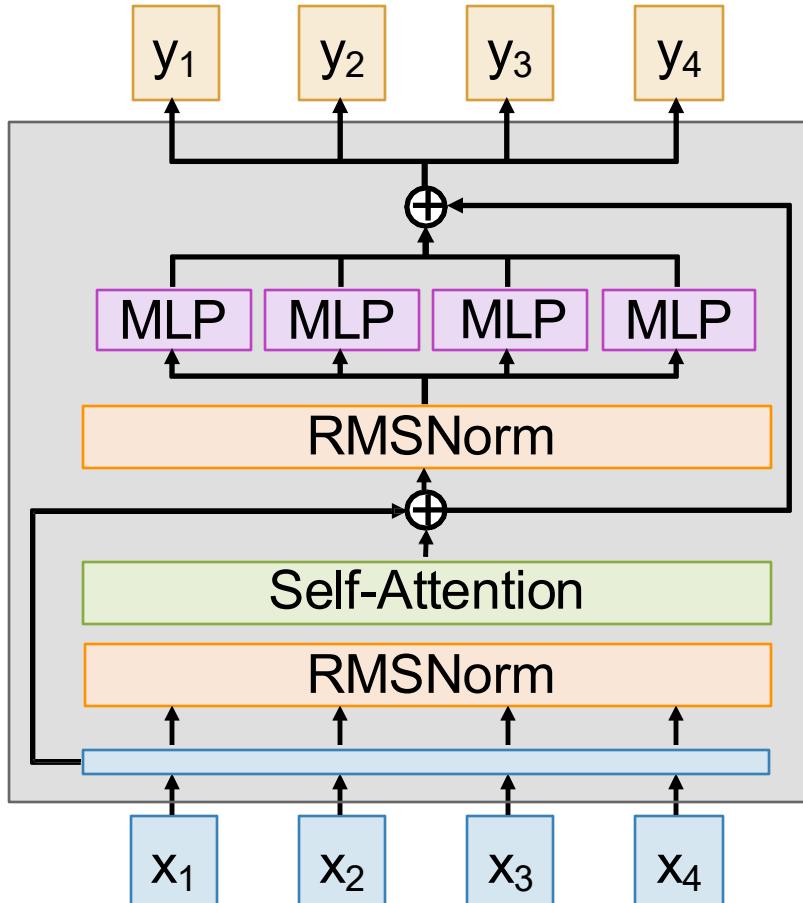
$W_3 [H \times D]$

Output:

$$Y = (\sigma(XW_1) \odot XW_2)W_3$$

Setting $H = 8D/3$ keeps
same total params

*We offer no explanation as
to why these architectures
seem to work; we attribute
their success, as all else,
to divine benevolence.*

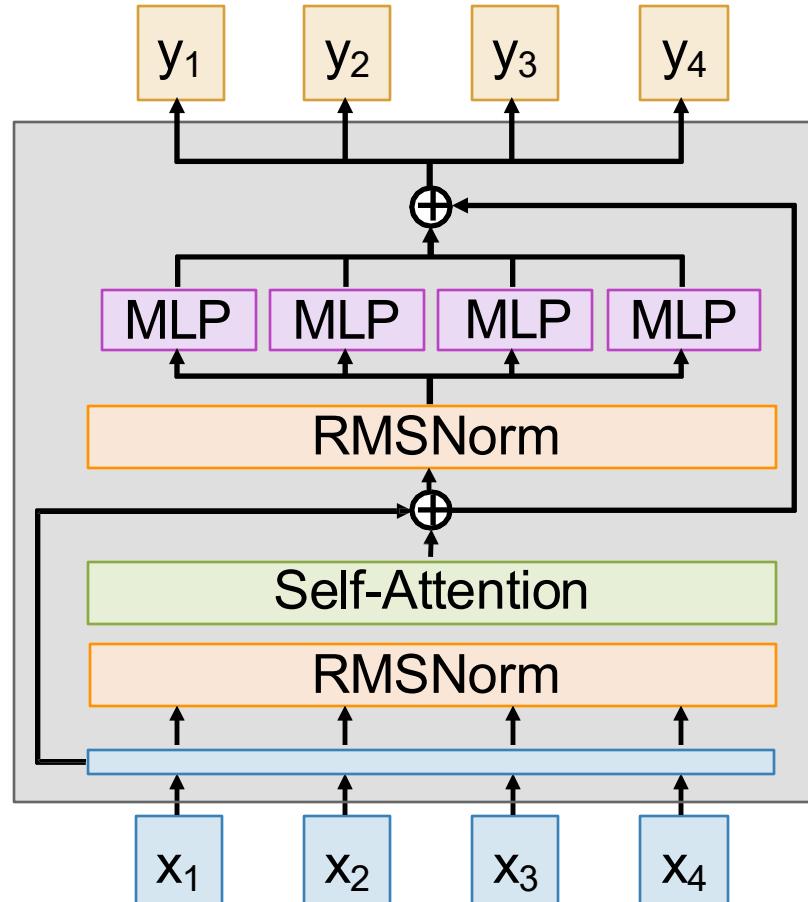


Mixture of Experts (MoE)

Learn E separate sets of MLP weights in each block; each MLP is an *expert*

$$W_1: [D \times 4D] \Rightarrow [E \times D \times 4D]$$

$$W_2: [4D \times D] \Rightarrow [E \times 4D \times D]$$



Mixture of Experts (MoE)

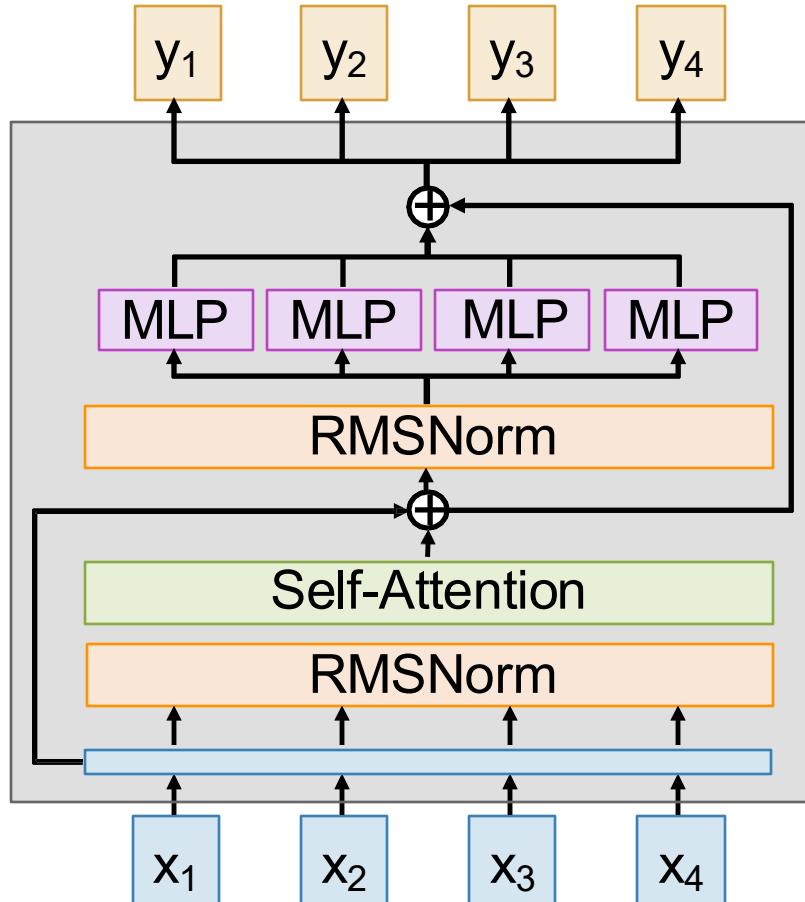
Learn E separate sets of MLP weights in each block; each MLP is an *expert*

$$W_1: [D \times 4D] \Rightarrow [E \times D \times 4D]$$

$$W_2: [4D \times D] \Rightarrow [E \times 4D \times D]$$

Each token gets *routed* to A < E of the experts. These are the *active experts*.

Increases params by E,
But only increases compute by A



Mixture of Experts (MoE)

Learn E separate sets of MLP weights in each block; each MLP is an *expert*

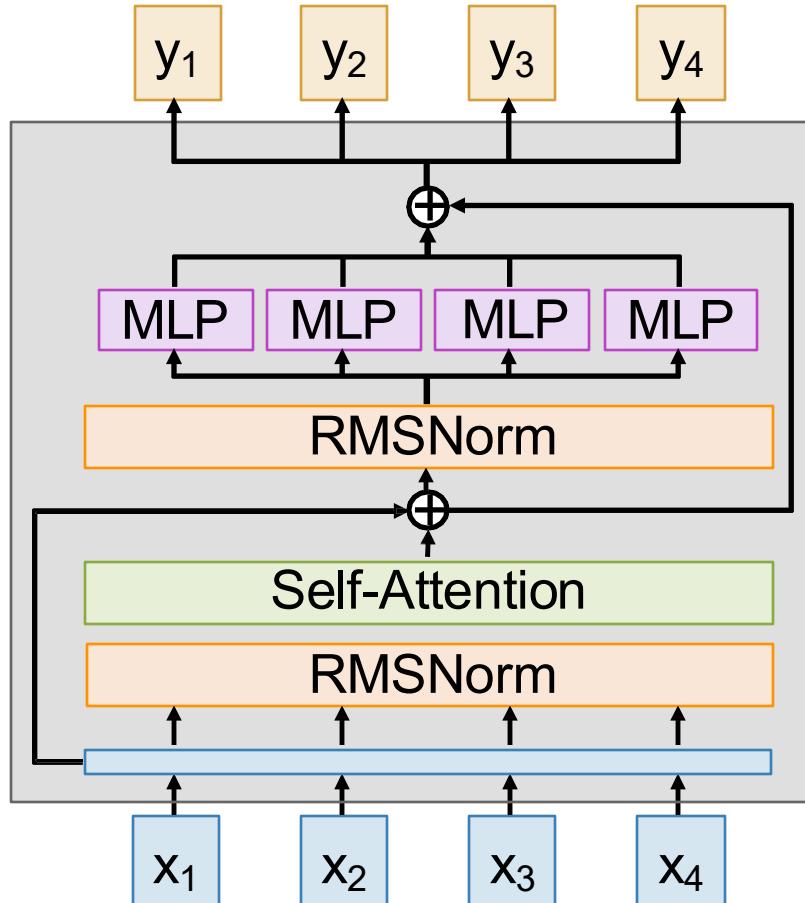
$$W_1: [D \times 4D] \Rightarrow [E \times D \times 4D]$$

$$W_2: [4D \times D] \Rightarrow [E \times 4D \times D]$$

Each token gets *routed* to A < E of the experts. These are the *active experts*.

Increases params by E,
But only increases compute by A

All of the biggest LLMs today (e.g. GPT4o, GPT4.5, Claude 3.7, Gemini 2.5 Pro, etc) almost certainly use MoE and have > 1T params; but they don't publish details anymore

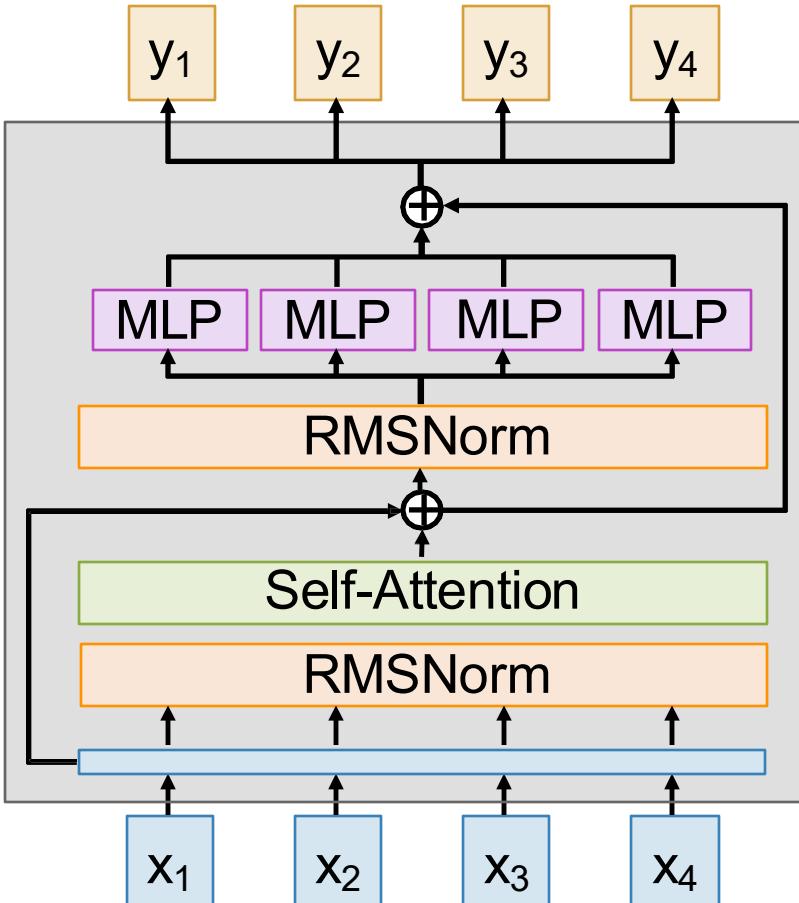


Tweaking Transformers

The Transformer architecture has not changed much since 2017.

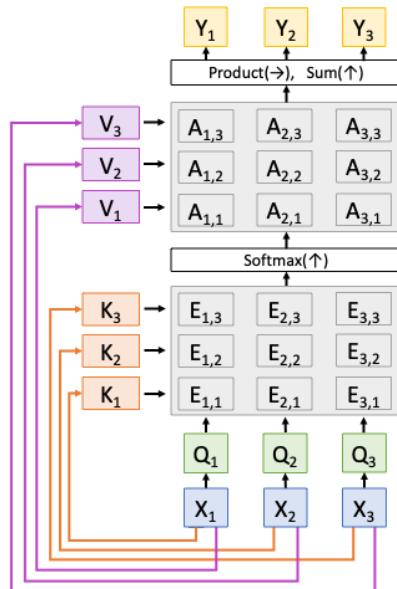
But a few changes have become common:

- **Pre-Norm:** Move normalization inside residual
- **RMSNorm:** Different normalization layer
- **SwiGLU:** Different MLP architecture
- **Mixture of Experts (MoE):** Learn E different MLPs, use A < E of them per token. Massively increase params, modest increase to compute cost.



Summary: Attention +Transformers

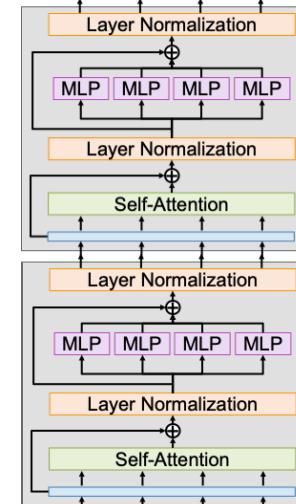
Attention: A new primitive that operates on sets of vectors



Transformers are the backbone of all large AI models today!

Used for language, vision, speech, ...

Transformer: A neural network architecture that uses attention everywhere



Next Time:
Detection, Segmentation,
Visualization