Deep Learning

Lecture 3: Autoencoder, CNN and RNN and their Variants

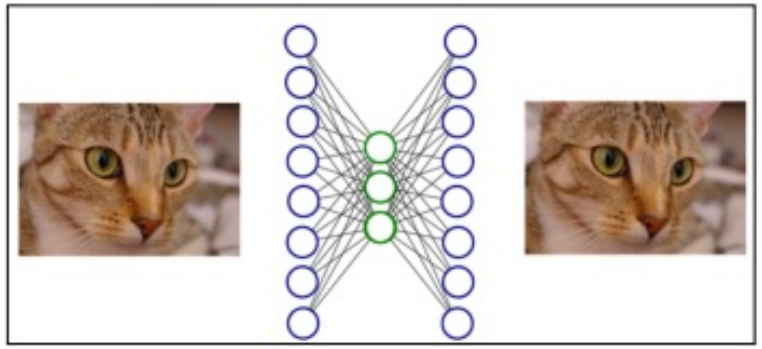
National University of Singapore

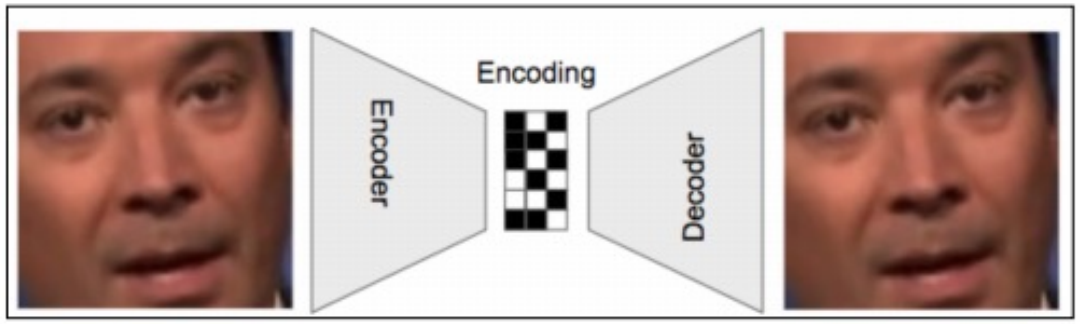
EE5934/6934

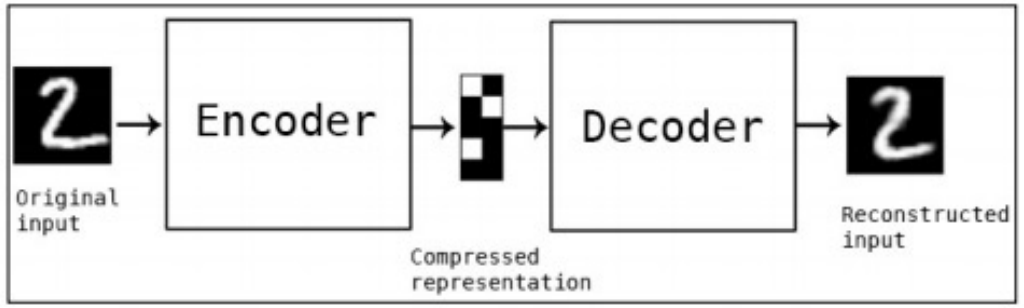
Instructor: Joey Tianyi Zhou

Autoencoder

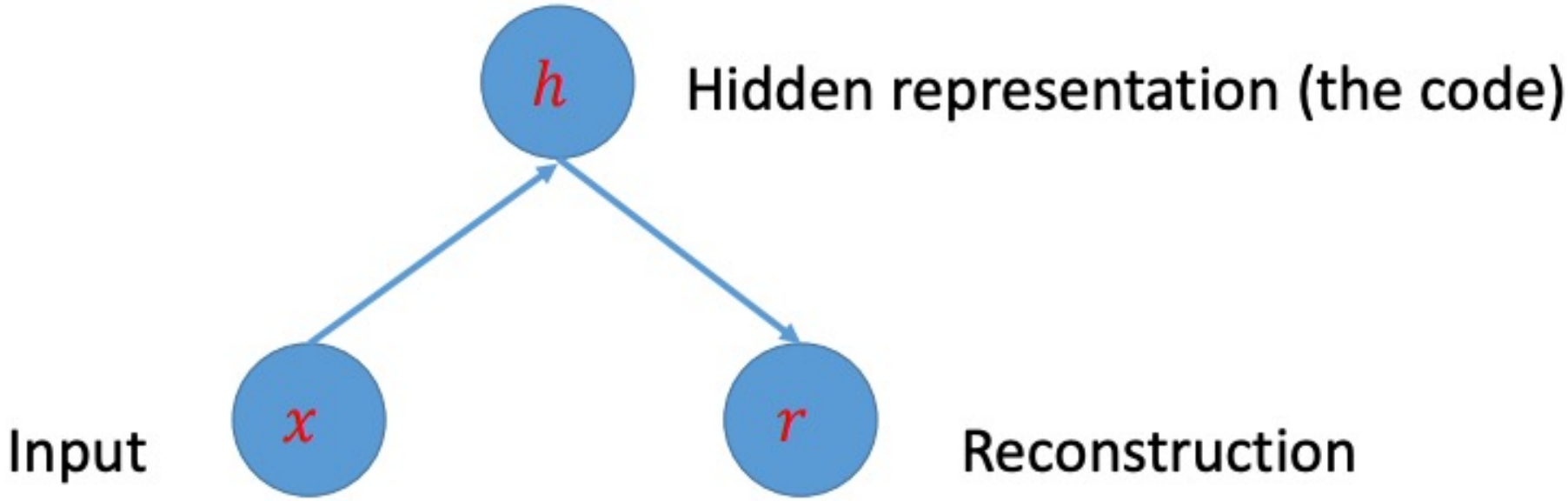
- A neural network trained using unsupervised learning
  - Trained to copy its input to its output
  - Learns an embedding



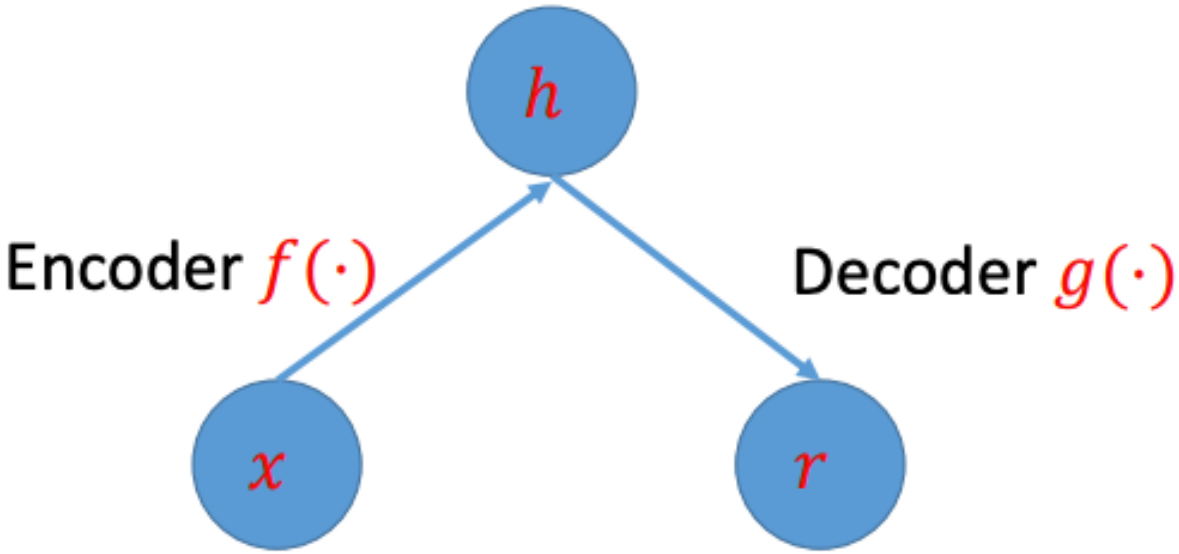




Autoencoder



Autoencoder



$$h = f(x), r = g(h) = g(f(x))$$

Why want to copy input to output

- Not really care about copying

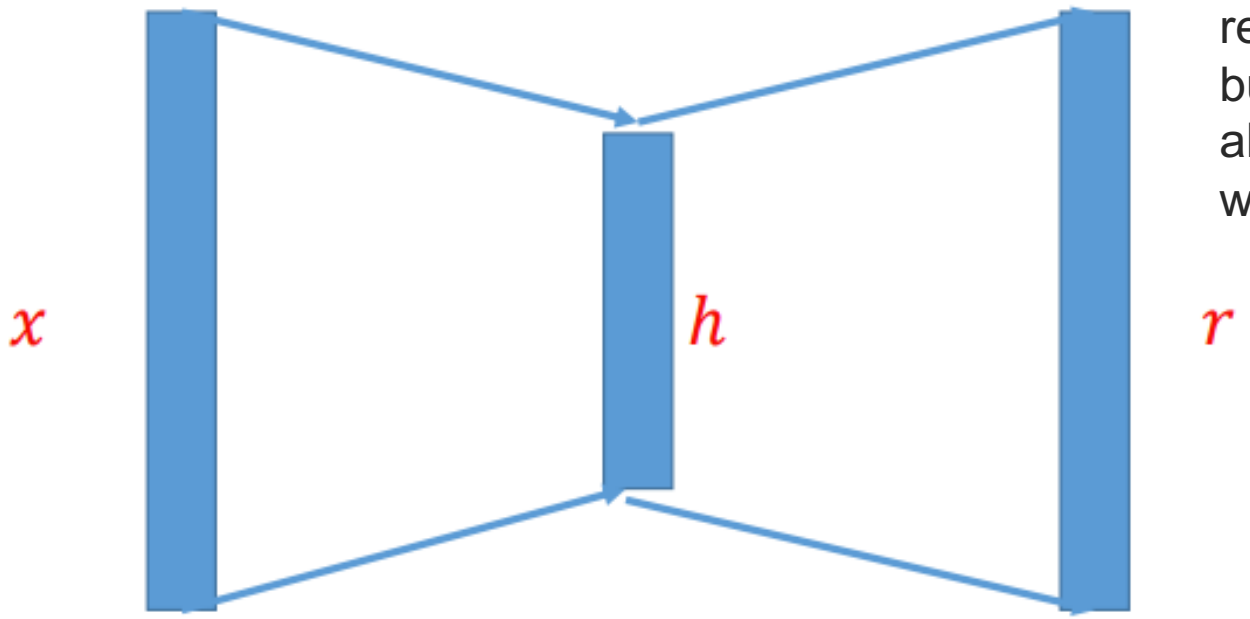
- Interesting case: NOT able to copy exactly but strive to do so
- Autoencoder forced to select which aspects to preserve and thus hopefully can learn useful properties of the data

- Historical note: goes back to (LeCun, 1987; Bourlard and Kamp, 1988; Hinton and Zemel, 1994).

Undercomplete autoencoder

- Constrain the code to have smaller dimension than the input
- Training: minimize a loss function

$$L(x, r) = L(x, g(f(x)))$$



simply learn to copy its inputs to the output, without learning any meaningful representation. It will just mimic the identity function. The autoencoder will reconstruct the training data perfectly, but it will be ***overfitting*** without being able to generalize to new instances, which is not what we want.

Undercomplete autoencoder

- Constrain the code to have smaller dimension than the input
- Training: minimize a loss function

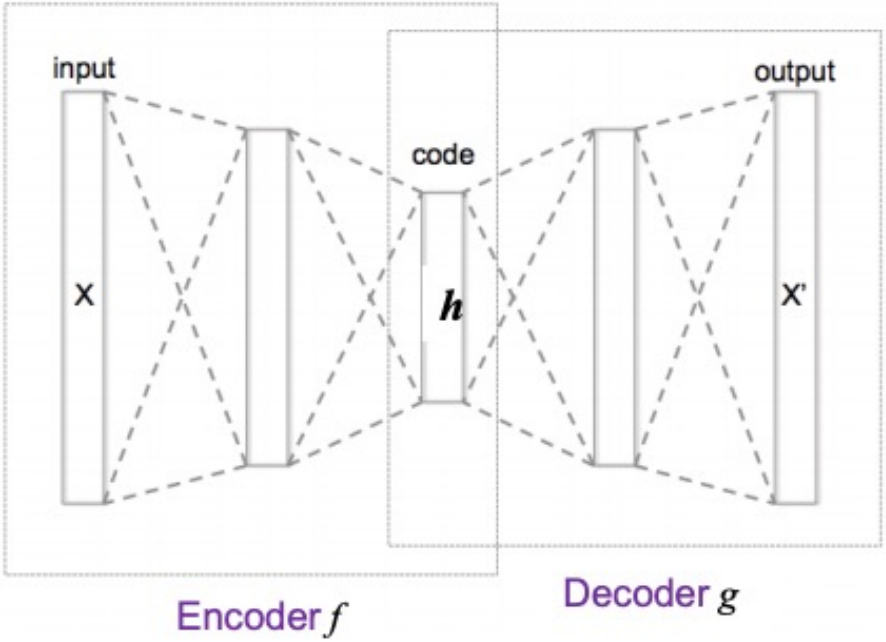
$$L(x, r) = L(x, g(f(x)))$$

- Special case: f, g linear, L mean square error
- Reduces to Principal Component Analysis

- Encoder f and decoder g

$$f : X \rightarrow \boldsymbol{h}$$
$$g : \boldsymbol{h} \rightarrow X$$
$$\arg\min_{f,g} \left\| X - (f \circ g)X \right\|^2$$

Autoencoder with 3 fully connected hidden layers



- One hidden layer
  - Non-linear encoder
  - Takes input $\boldsymbol{x} \in R^d$
  - Maps into output $\boldsymbol{h} \in R^p$

$$\boldsymbol{h} = \sigma_1(W\boldsymbol{x} + \boldsymbol{b})$$
$$\boldsymbol{x}' = \sigma_2(W'\boldsymbol{h} + \boldsymbol{b}')$$

σ is an element-wise activation function such as sigmoid or Relu

Trained to minimize reconstruction error (such as sum of squared errors)

$$L(\boldsymbol{x}, \boldsymbol{x}') = \left\| \boldsymbol{x} - \boldsymbol{x}' \right\|^2 = \left\| \boldsymbol{x} - \sigma_2(W^t(\sigma_1(W\boldsymbol{x} + \boldsymbol{b})) + \boldsymbol{b}') \right\|^2$$

Provides a compressed representation of the input $\boldsymbol{x}$

Encoder/Decoder Capacity

- If encoder f and decoder g are allowed too much capacity
  - autoencoder can learn to perform the copying task without learning any useful information about distribution of data
- Autoencoder with a one-dimensional code and a very powerful nonlinear encoder can learn to map $\boldsymbol{x}^{(i)}$ to code i.
  - The decoder can learn to map these integer indices back to the values of specific training examples
- Autoencoder trained for copying task fails to learn anything useful if f/g capacity is too great

Cases when Autoencoder Learning Fails

- Where autoencoders fail to learn anything useful:
  1. Capacity of encoder/decoder f/g is too high
     - Capacity controlled by depth
  2. Hidden code $\boldsymbol{h}$ has dimension equal to input $\boldsymbol{x}$
  3. *Overcomplete* case: where hidden code $\boldsymbol{h}$ has dimension greater than input $\boldsymbol{x}$
     - Even a linear encoder/decoder can learn to copy input to output without learning anything useful about data distribution

Right Autoencoder Design: Use regularization

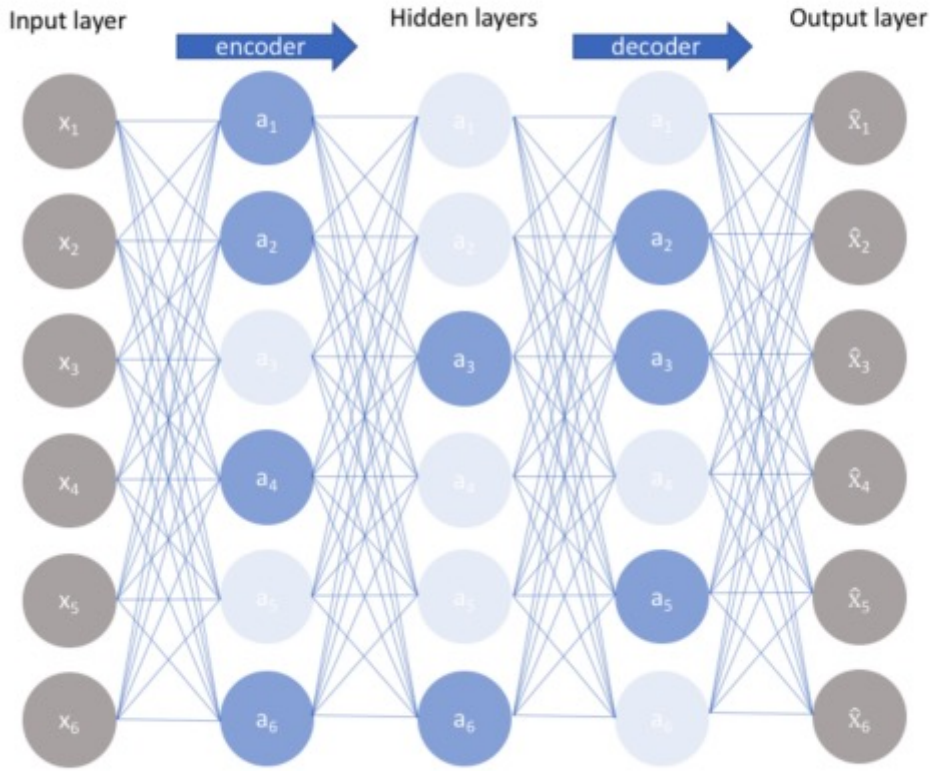
- Ideally, choose code size (dimension of $\boldsymbol{h}$) small and capacity of encoder f and decoder g based on complexity of distribution modeled
- *Regularized autoencoders* provide the ability to do so
  - Rather than limiting model capacity by keeping encoder/decoder shallow and code size small
  - They use a loss function that encourages the model to have properties other than copy its input to output

2. Regularized Autoencoder Properties

- Regularized AEs have properties beyond copying input to output:
  - Sparsity of representation
  - Smallness of the derivative of the representation
  - Robustness to noise
  - Robustness to missing inputs
- Regularized autoencoder can be nonlinear and overcomplete
  - But still learn something useful about the data distribution even if model capacity is great enough to learn trivial identity function

Sparse Autoencoder

Only a few nodes are encouraged to activate when a single sample is fed into the network



Fewer nodes activating while still keeping its performance would guarantee that the autoencoder is actually learning **latent representations** instead of redundant information in our input data

Sparse Autoencoder Loss Function

- A sparse autoencoder is an autoencoder whose
  - Training criterion includes a sparsity penalty $\Omega(\boldsymbol{h})$ on the code layer $\boldsymbol{h}$ in addition to the reconstruction error:

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))) + \Omega(\boldsymbol{h})$$

  - where $g(\boldsymbol{h})$ is the decoder output and typically we have $\boldsymbol{h} = f(\boldsymbol{x})$
- Sparse encoders are typically used to learn features for another task such as classification
- An autoencoder that has been trained to be sparse must respond to unique statistical features of the dataset rather than simply perform the copying task
  - Thus sparsity penalty can yield a model that has learned useful features as a byproduct

2. Regularized Autoencoder Properties

- Regularized AEs have properties beyond copying input to output:
  - Sparsity of representation
  - Smallness of the derivative of the representation
  - Robustness to noise
  - Robustness to missing inputs

How to handle?

- Regularized autoencoder can be nonlinear and overcomplete
  - But still learn something useful about the data distribution even if model capacity is great enough to learn trivial identity function

2. Regularized Autoencoder Properties

- Regularized AEs have properties beyond copying input to output:
  - Sparsity of representation
  - Smallness of the derivative of the representation
  - Robustness to noise
  - Robustness to missing inputs
- Regularized autoencoder can be nonlinear and overcomplete
  - But still learn something useful about the data distribution even if model capacity is great enough to learn trivial identity function

How to handle?

Regularizing by Penalizing Derivatives

- Another strategy for regularizing an autoencoder
- Use penalty as in sparse autoencoders

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))) + \Omega(\boldsymbol{h}, \boldsymbol{x})$$

- But with a different form of Ω

$$\boxed{\Omega(\boldsymbol{h}, \boldsymbol{x}) = \lambda \sum_i \left\| \nabla_{\boldsymbol{x}} h_i \right\|^2}$$

- Forces the model to learn a function that does not change much when $\boldsymbol{x}$ changes slightly
- Called a *Contractive Auto Encoder* (CAE)
- This model has theoretical connections to
  - Denoising autoencoders
  - Manifold learning
  - Probabilistic modeling

2. Regularized Autoencoder Properties

- Regularized AEs have properties beyond copying input to output:
  - Sparsity of representation
  - Smallness of the derivative of the representation
  - Robustness to noise
  - Robustness to missing inputs

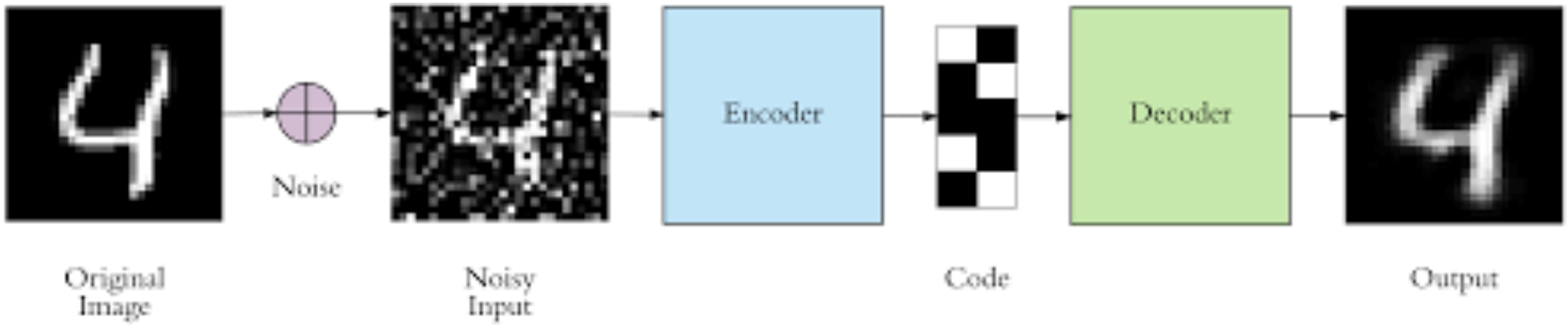
How to handle?

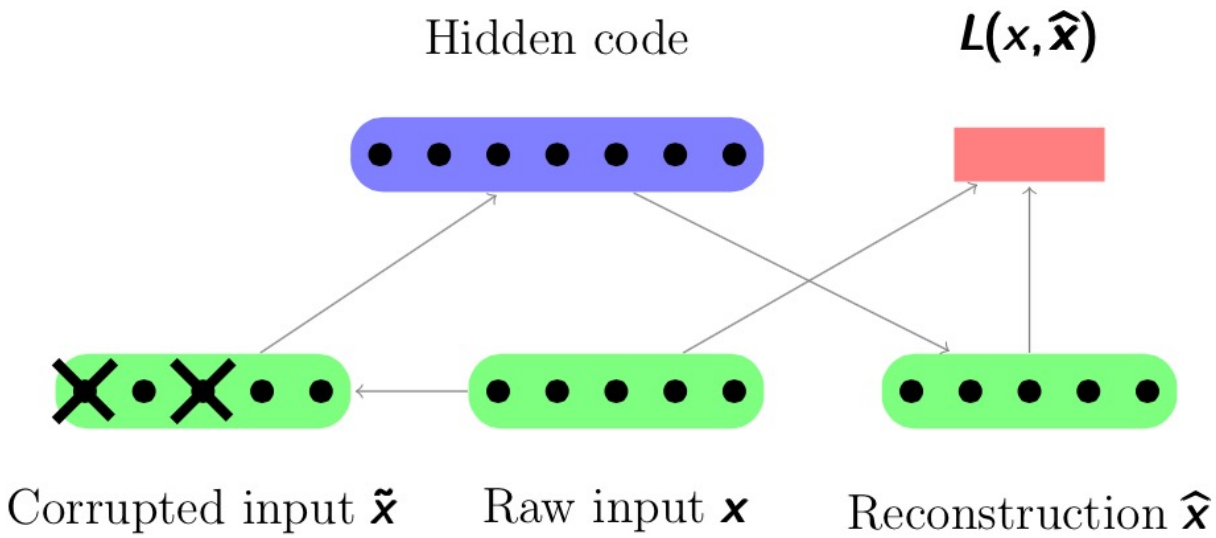
- Regularized autoencoder can be nonlinear and overcomplete
  - But still learn something useful about the data distribution even if model capacity is great enough to learn trivial identity function

Denoising Autoencoders (DAE)

- Rather than adding a penalty Ω to the cost function, we can obtain an autoencoder that learns something useful
  - By changing the reconstruction error term of the cost function
- Traditional autoencoders minimize $L(\boldsymbol{x}, g(f(\boldsymbol{x})))$
  - where L is a loss function penalizing $g(f(\boldsymbol{x}))$ for being dissimilar from $\boldsymbol{x}$, such as L^2 norm of difference: mean squared error
- A DAE minimizes $L(\boldsymbol{x}, g(f(\tilde{\boldsymbol{x}})))$
  - where $\tilde{\boldsymbol{x}}$ is a copy of $\boldsymbol{x}$ that has been corrupted by some form of noise
  - The autoencoder must undo this corruption rather than simply copying their input
- Denoising training forces f and g to implicitly learn the structure of $p_{\text{data}}(\boldsymbol{x})$
- Another example of how useful properties can emerge as a by-product of minimizing reconstruction error

Denoising Autoencoders (DAE)





$$\frac{1}{2n}\sum_{i=1}^{n}\|\mathbf{x}_i - \mathbf{W}\tilde{\mathbf{x}}_i\|^2.$$

$$\frac{1}{2n}\sum_{i=1}^{n}\|\mathbf{x}_i - \mathbf{W}\tilde{\mathbf{x}}_i\|^2. \qquad (1)$$

The solution to (1) depends on which features of each input are randomly corrupted. To lower the variance, we perform multiple passes over the training set, each time with different corruption. We solve for the W that minimizes the overall squared loss

$$\mathcal{L}_{sq}(\mathbf{W}) = \frac{1}{2mn}\sum_{j=1}^{m}\sum_{i=1}^{n}\|\mathbf{x}_i - \mathbf{W}\tilde{\mathbf{x}}_{i,j}\|^2, \qquad (2)$$

where $\tilde{\mathbf{x}}_{i,j}$ represents the jth corrupted version of the original input $\mathbf{x}_i$.

Let us define the design matrix $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_n] \in \mathcal{R}^{d \times n}$ and its m-times repeated version as $\overline{\mathbf{X}} = [\mathbf{X}, \ldots, \mathbf{X}]$. Further, we denote the corrupted version of $\overline{\mathbf{X}}$ as $\widetilde{\mathbf{X}}$. With this notation, the loss in eq. (1) reduces to

$$\mathcal{L}_{sq}(\mathbf{W}) = \frac{1}{2nm} \mathrm{tr} \left[\left(\overline{\mathbf{X}} - \mathbf{W}\widetilde{\mathbf{X}} \right)^\top \left(\overline{\mathbf{X}} - \mathbf{W}\widetilde{\mathbf{X}} \right) \right]. \quad (3)$$

The solution to (3) can be expressed as the well-known closed-form solution for ordinary least squares (Bishop, 2006):

$$\mathbf{W} = \mathbf{P}\mathbf{Q}^{-1} \text{ with } \mathbf{Q} = \widetilde{\mathbf{X}}\widetilde{\mathbf{X}}^\top \text{ and } \mathbf{P} = \overline{\mathbf{X}}\widetilde{\mathbf{X}}^\top. \quad (4)$$

(In practice this can be computed as a system of linear equations, without the costly matrix inversion.)

Marginalized Denoising Autoencoder

The larger m is, the more corruptions we average over. Ideally we would like $m \to \infty$, effectively using infinitely many copies of noisy data to compute the denoising transformation $\mathbf{W}$.

By the weak law of large numbers, the matrices $\mathbf{P}$ and $\mathbf{Q}$, as defined in (3), converge to their expected values as m becomes very large. If we are interested in the limit case, where $m \to \infty$, we can derive the expectations of $\mathbf{Q}$ and $\mathbf{P}$, and express the corresponding mapping $\mathbf{W}$ as

$$\mathbf{W} = E[\mathbf{P}]\boxed{E[\mathbf{Q}]^{-1}}. \tag{5}$$

Marginalized Denoising Autoencoder

$$\mathbf{Q} = \tilde{\mathbf{X}}\tilde{\mathbf{X}}^\top$$

Each feature is set to 0 with probability p.

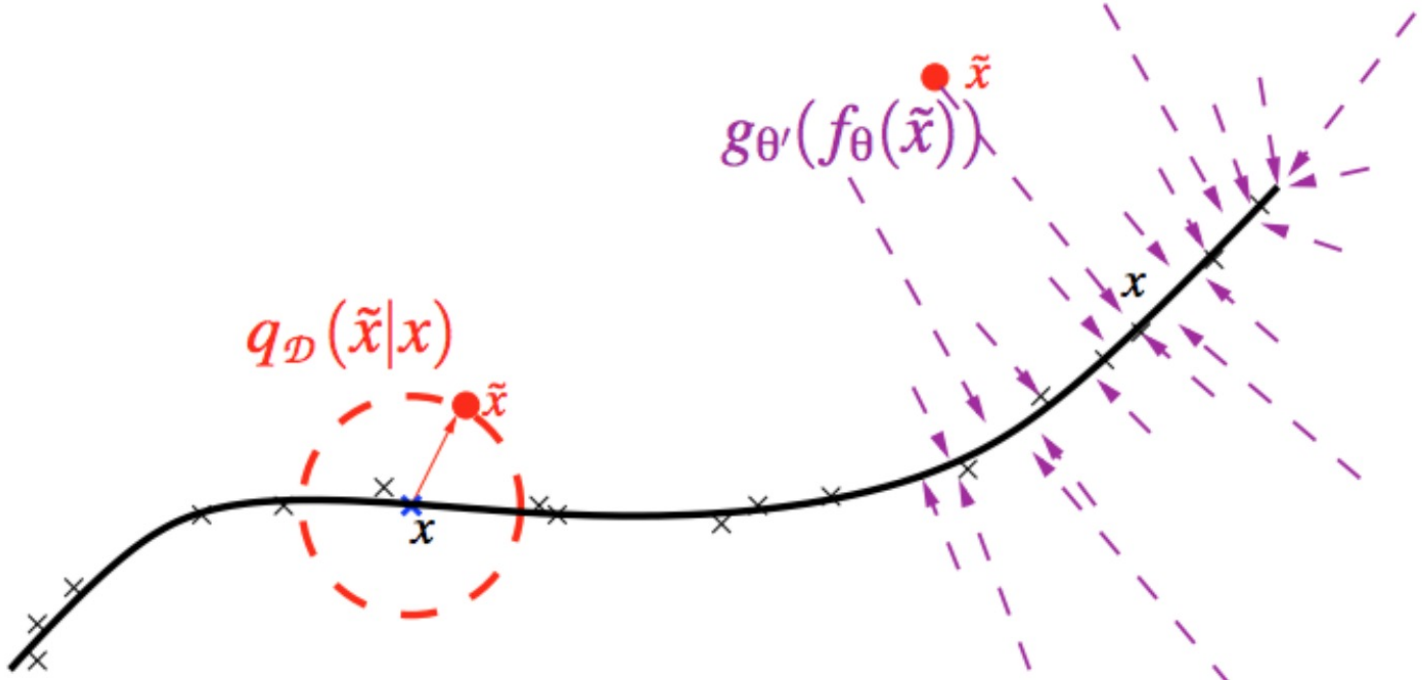
An off-diagonal entry in the matrix $\tilde{\mathbf{x}}_i\tilde{\mathbf{x}}_i^\top$ is uncorrupted if the two features α and β both "survived" the corruption, which happens with probability $(1-p)^2$. For the diagonal entries, this holds with probability $1-p$. Let us define a vector $\mathbf{q} = [1-p, \ldots, 1-p, 1]^\top \in \mathcal{R}^{d+1}$, where $\mathbf{q}_\alpha$ represents the probability of a feature α "surviving" the corruption. As the constant feature is never corrupted, we have $\mathbf{q}_{d+1} = 1$. If we further define the scatter matrix of the original uncorrupted input as $\mathbf{S} = \mathbf{X}\mathbf{X}^\top$, we can express the expectation of the matrix Q as

$$E[\mathbf{Q}]_{\alpha,\beta} = \begin{cases} \mathbf{S}_{\alpha\beta}\mathbf{q}_\alpha\mathbf{q}_\beta & \text{if} \quad \alpha \neq \beta \\ \mathbf{S}_{\alpha\beta}\mathbf{q}_\alpha & \text{if} \quad \alpha = \beta \end{cases} . \tag{7}$$

Similarly, we obtain the expectation of $\mathbf{P}$ in closed-form as $E[\mathbf{P}]_{\alpha\beta} = \mathbf{S}_{\alpha\beta}\mathbf{q}_\beta$.

Denoising Autoencoder Why works well?

- How does it work?
- Even though, e.g., MNIST data are in a 784-dimensional space, they lie on a low-dimensional **manifold** that captures their most important features
- Corruption process moves instance $\boldsymbol{x}$ off of manifold
- Encoder f_θ and decoder $g_{\theta'}$ are trained to project $\tilde{\boldsymbol{x}}$ back onto manifold



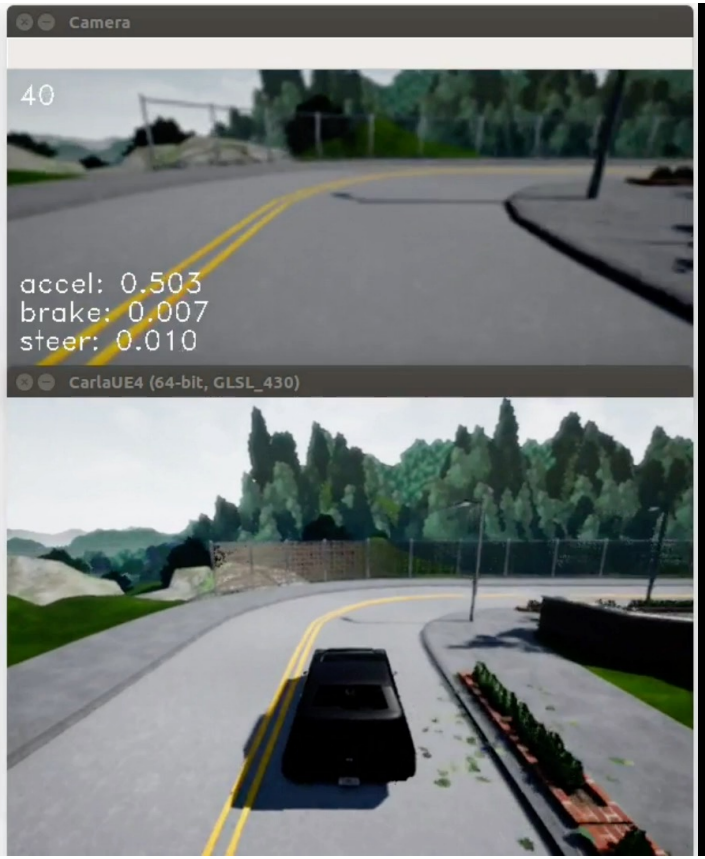
Denoising Autoencoder vs Adversarial Examples

Classified as panda | Small noise | Classified as gibbon

Noise can be from adversarial attack or natural conditions

The stop sign attack

Attacking End-to-End Autonomous Car



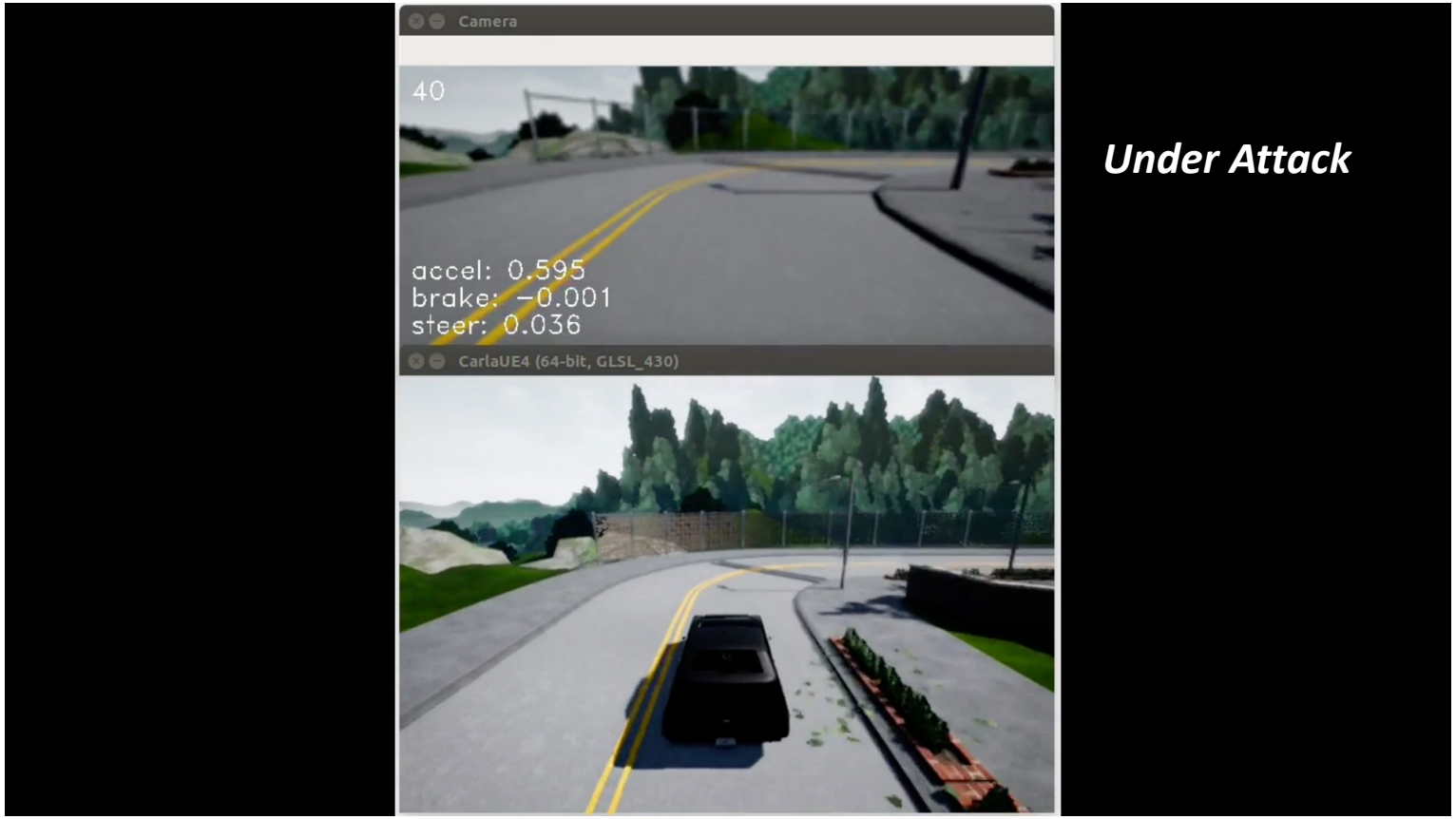
No Attack

CNN for end-to-end autonomous driving

Attacking End-to-End Autonomous Car

Attack: add (paint) this object to the road (and rotate)

Attacking End-to-End Autonomous Car



Adversarial Training in Vision

Classified as panda | Small noise | Classified as gibbon

Adversarial Training: An adversarial sample is built by adding the original sample with a <u>perturbation</u> bounded by a small norm ϵ to **maximize** the loss function as

$$\eta_{\mathbf{x}} = \arg \max_{\eta : \|\eta\|_2 \leq \epsilon} \ell(\Theta; \mathbf{x} + \eta)$$

where Θ is the current model parameters set. η is estimated by

$$\eta_{\mathbf{x}} = \epsilon \frac{\mathbf{g}}{\|\mathbf{g}\|_2}, \quad \text{where } \mathbf{g} = \nabla \ell(\Theta; \mathbf{x})$$

Adversarial Training in NLP

The high-resource dataset does not always have the correct annotation!

- We found some **mislabeled** data in CoNLL and Ontonotes.

Example:

Rare O
Hendrix B-PER
song O
draft O
(Rare Hendrix is a singer)

Incomplete & mistyped word

Example:

I am soooooo happy today. Happy “Niu” Year

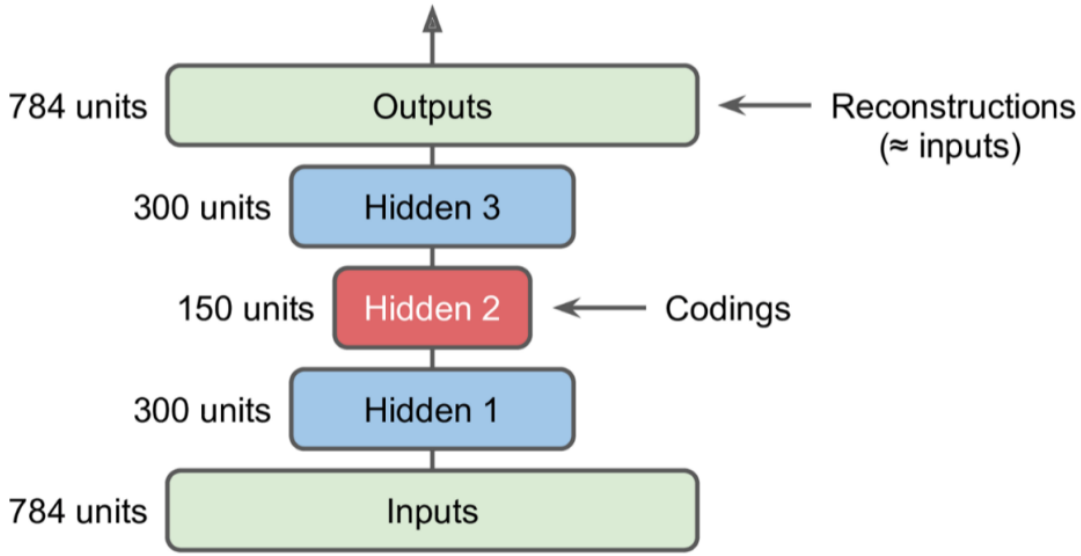
Adversarial Training: An adversarial sample is built by adding the original sample with a perturbation bounded by a small norm ϵ to **maximize** the loss function as

$$\eta_{\mathbf{x}} = \arg \max_{\eta : \|\eta\|_2 \leq \epsilon} \ell(\Theta; \mathbf{x} + \eta)$$

where Θ is the current model parameters set. η is estimated by

$$\eta_{\mathbf{x}} = \epsilon \frac{\mathbf{g}}{\|\mathbf{g}\|_2}, \quad \text{where } \mathbf{g} = \nabla \ell(\Theta; \mathbf{x})$$

Stacked Autoencoders

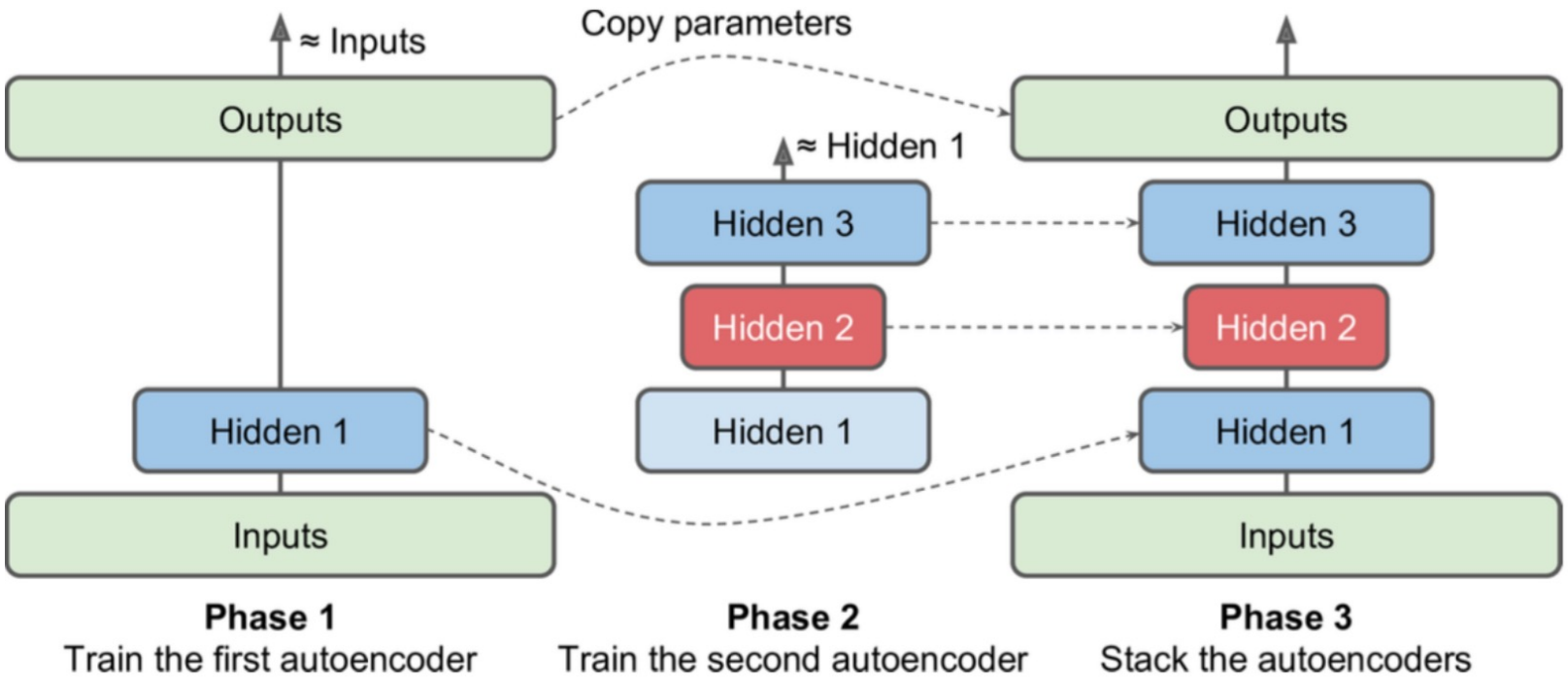


- A **stacked autoencoder** has multiple hidden layers

- Can share parameters to reduce their number by exploiting symmetry: $W_4 = W_1^\top$ and $W_3 = W_2^\top$

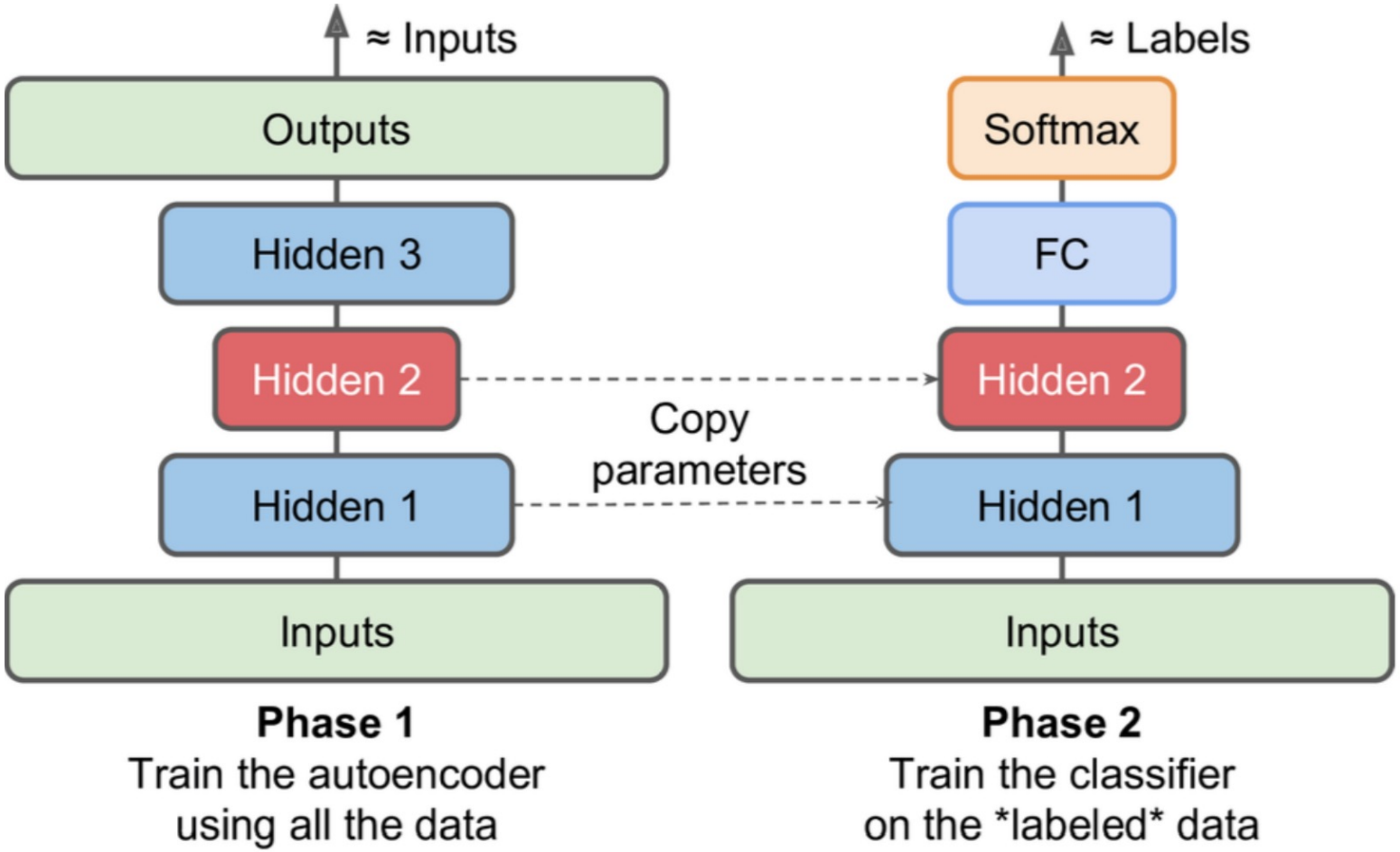
```
weights1 = tf.Variable(weights1_init, dtype=tf.float32, name="weights1")
weights2 = tf.Variable(weights2_init, dtype=tf.float32, name="weights2")
weights3 = tf.transpose(weights2, name="weights3")      # shared weights
weights4 = tf.transpose(weights1, name="weights4")      # shared weights
```

Stacked Autoencoders



- Can simplify training by starting with single hidden layer H_1
- Then, train a second AE to mimic the output of H_1
- Insert this into first network
- Can build by using H_1's output as training set for Phase 2

Stacked Autoencoders for Supervised Learning



- Can **pre-train** network with unlabeled data

⇒ learn useful features and then train "logic" of dense layer with labeled data

Time Break- 15 Mins

CONVOLUTIONAL NEURAL NETWORKS

- CNN -

CNN Architecture

CNN: a Neural network with specialized connectivity structure .. **multiple layers of feature extractors** are stacked: **low-level layers** extract local features; **high-level layers** extract learn global patterns.

Biologically Inspired…

Distinct types of layers of the **CNN** :

- **Convolutional layer** .. several usually
- **Non-linear layer** .. most popular is "ReLU"
- **Pooling layer** .. most often Max pooling

After some rounds of cascading, last convolutional layer is connected to a **fully connected layer** then proceeds as in a usual neural network.

CNN transform input data into an **output class / prediction**.

Convolutional neural networks

- Strong empirical application performance

- Convolutional networks: neural networks that use convolution in place of general matrix multiplication in at least one of their layers

$$h = \sigma(W^T x + b)$$

for a specific kind of weight matrix W

Convolution: math formula

- Given functions $u(t)$ and $w(t)$, their convolution is a function $s(t)$

$$s(t) = \int u(a)w(t-a)da$$

- Written as

$$s = (u * w) \quad \text{or} \quad s(t) = (u * w)(t)$$

Convolution: discrete version

- Given array u_t and w_t, their convolution is a function s_t

$$s_t = \sum_{a=-\infty}^{+\infty} u_a w_{t-a}$$

U: left to right
W: right to left

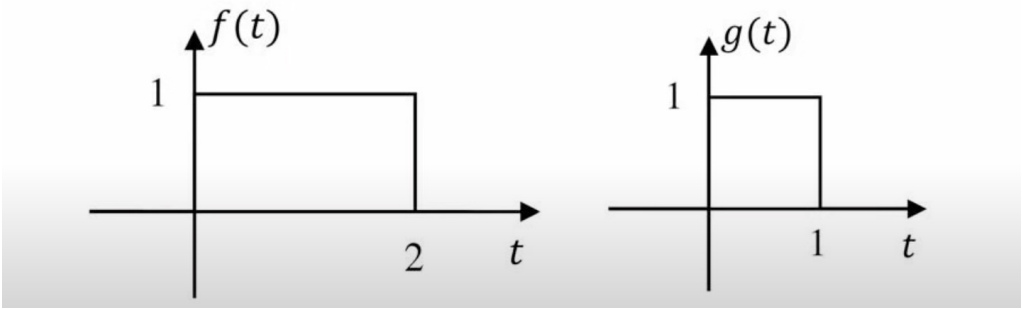
- Written as

$$s = (u * w) \quad \text{or} \quad s_t = (u * w)_t$$

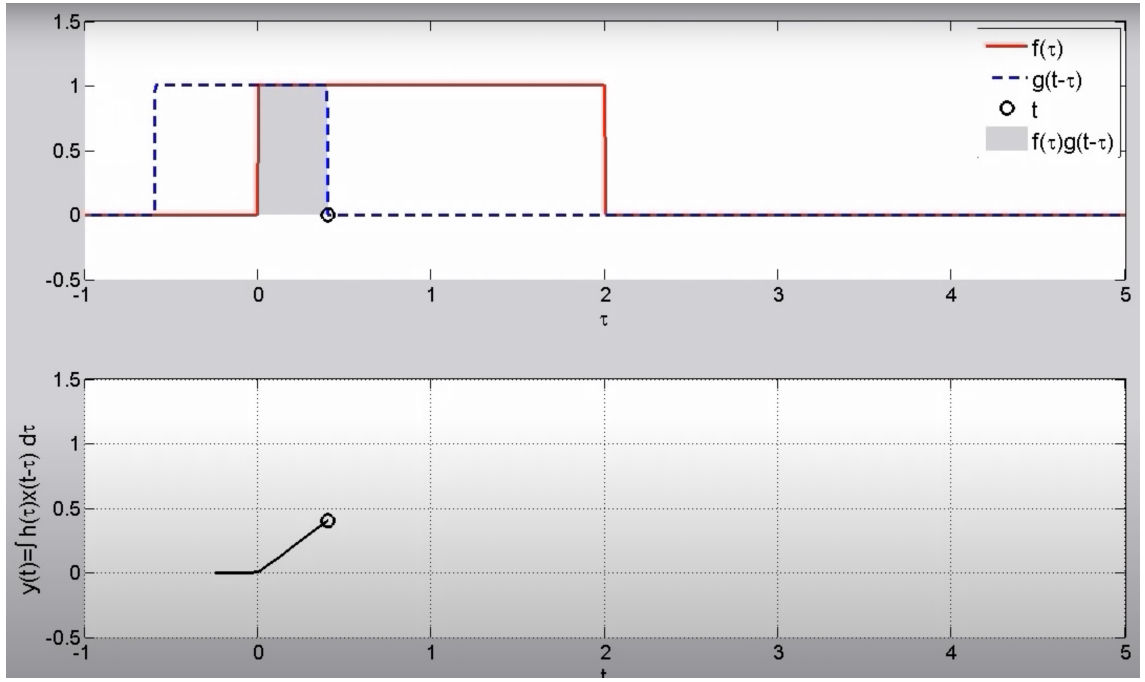
- When u_t or w_t is not defined, assumed to be 0

Convolution of two box functions

- $f(t) * g(t)$

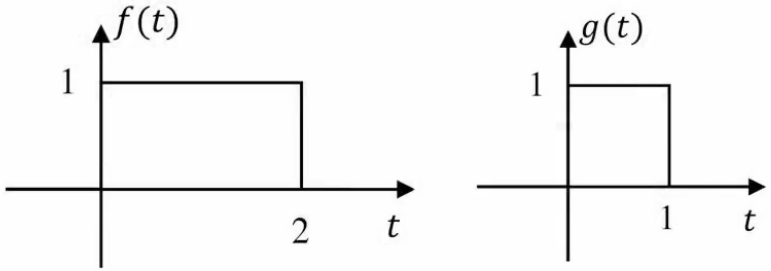


https://www.youtube.com/watch?v=C1N55M1VD2o



Convolution of two box functions

- $f(t) * g(t)$



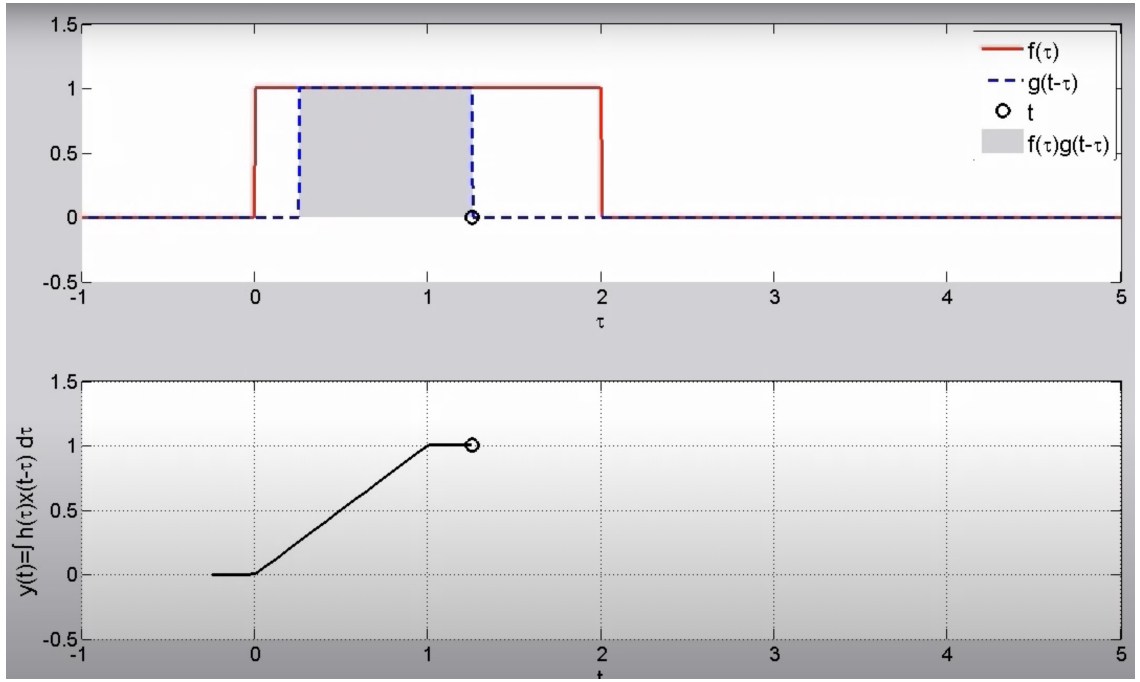


Illustration 1

w = [z, y, x]

u = [a, b, c, d, e, f]

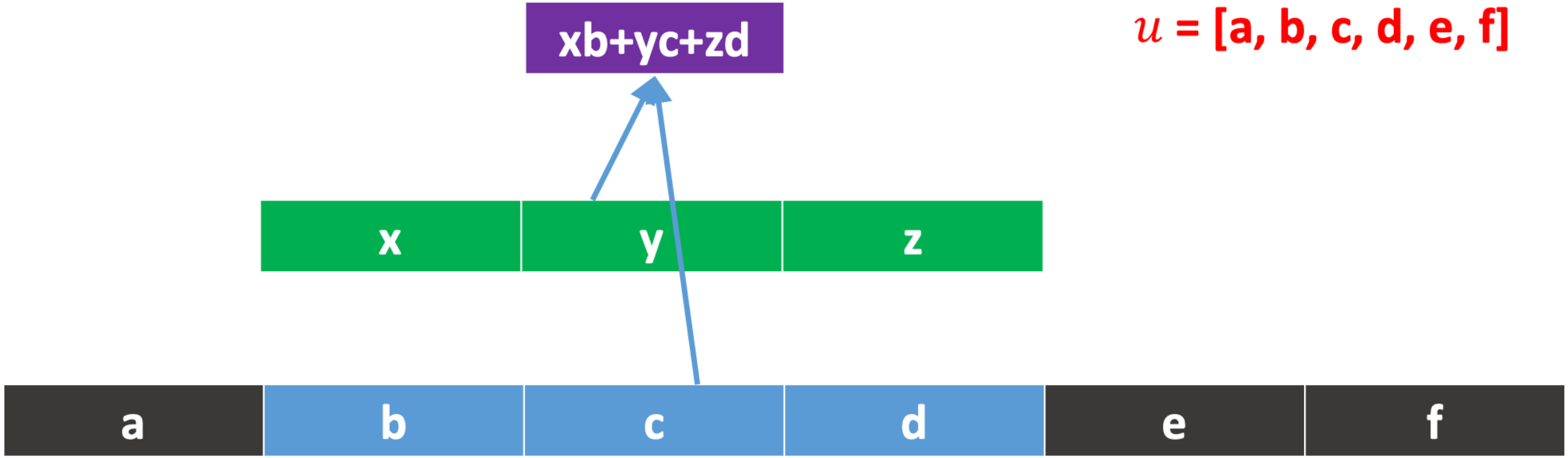


Illustration 1

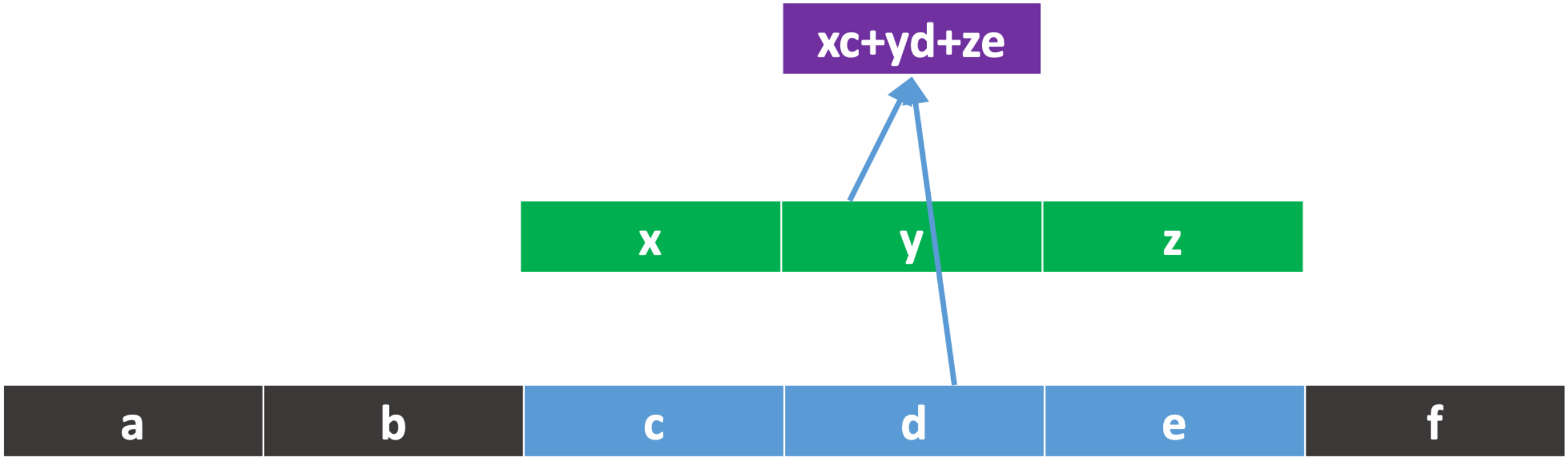


Illustration 1

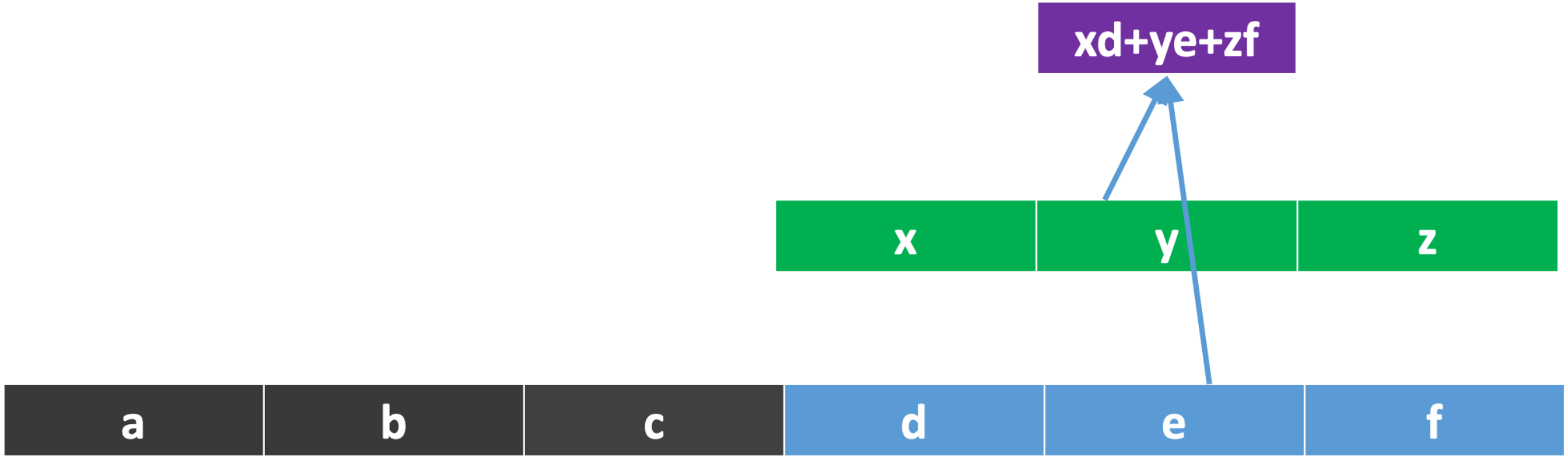


Illustration 1: boundary case

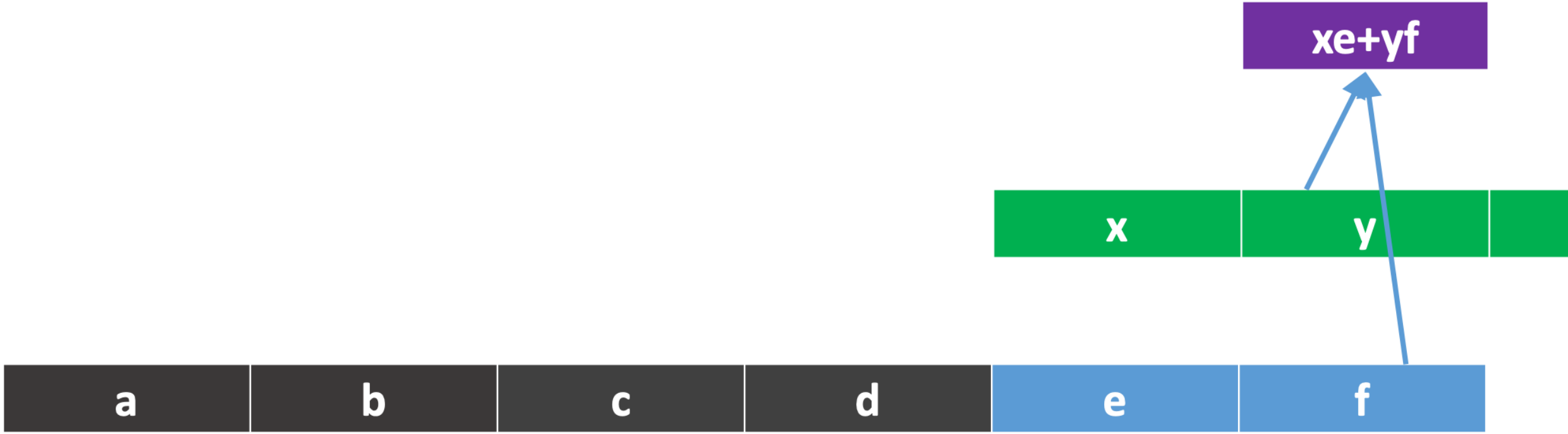


Illustration 1 as matrix multiplication

y	z				
x	y	z			
	x	y	z		
		x	y	z	
			x	y	z
				x	y

a
b
c
d
e
f

Illustration 2: two dimensional case

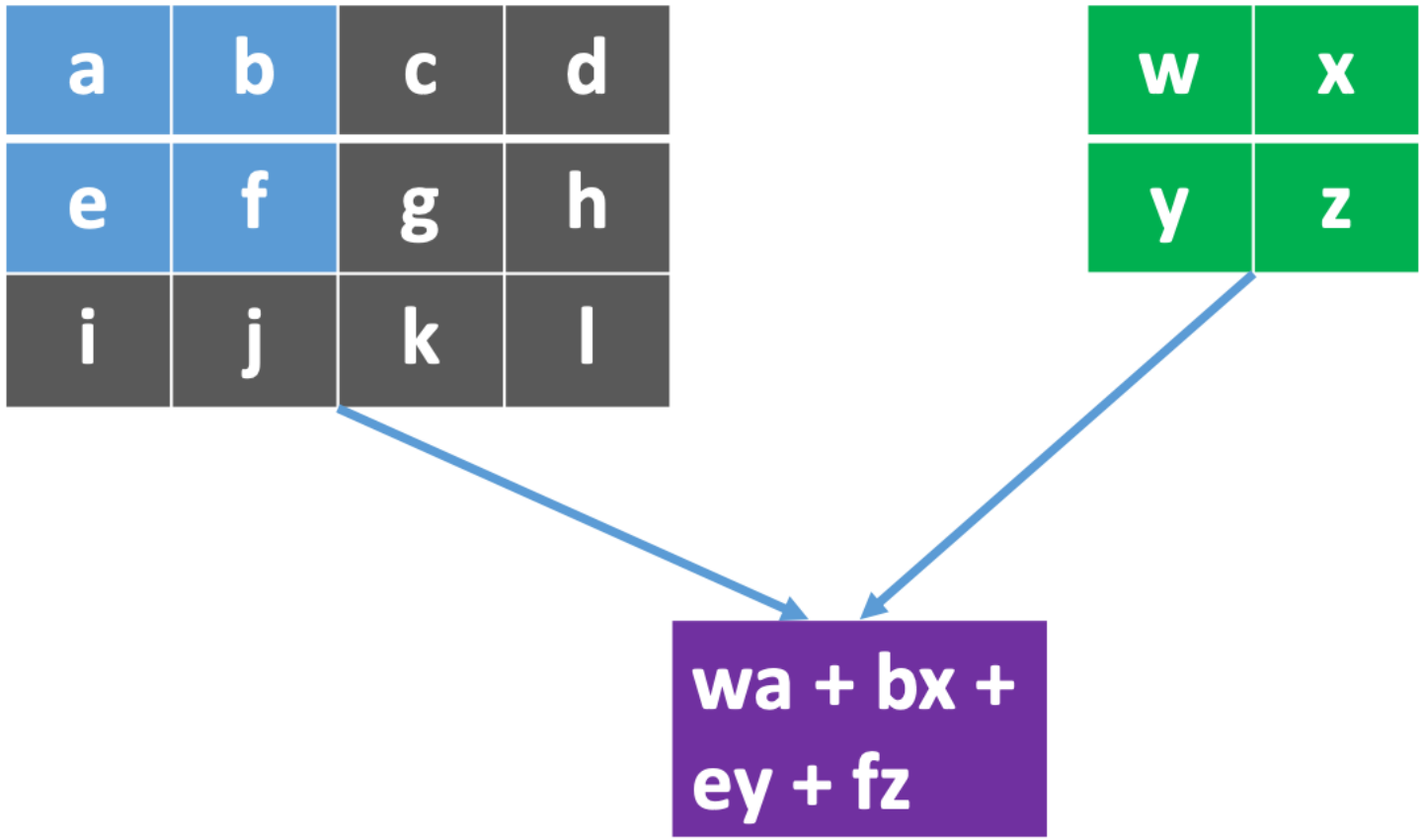


Illustration 2

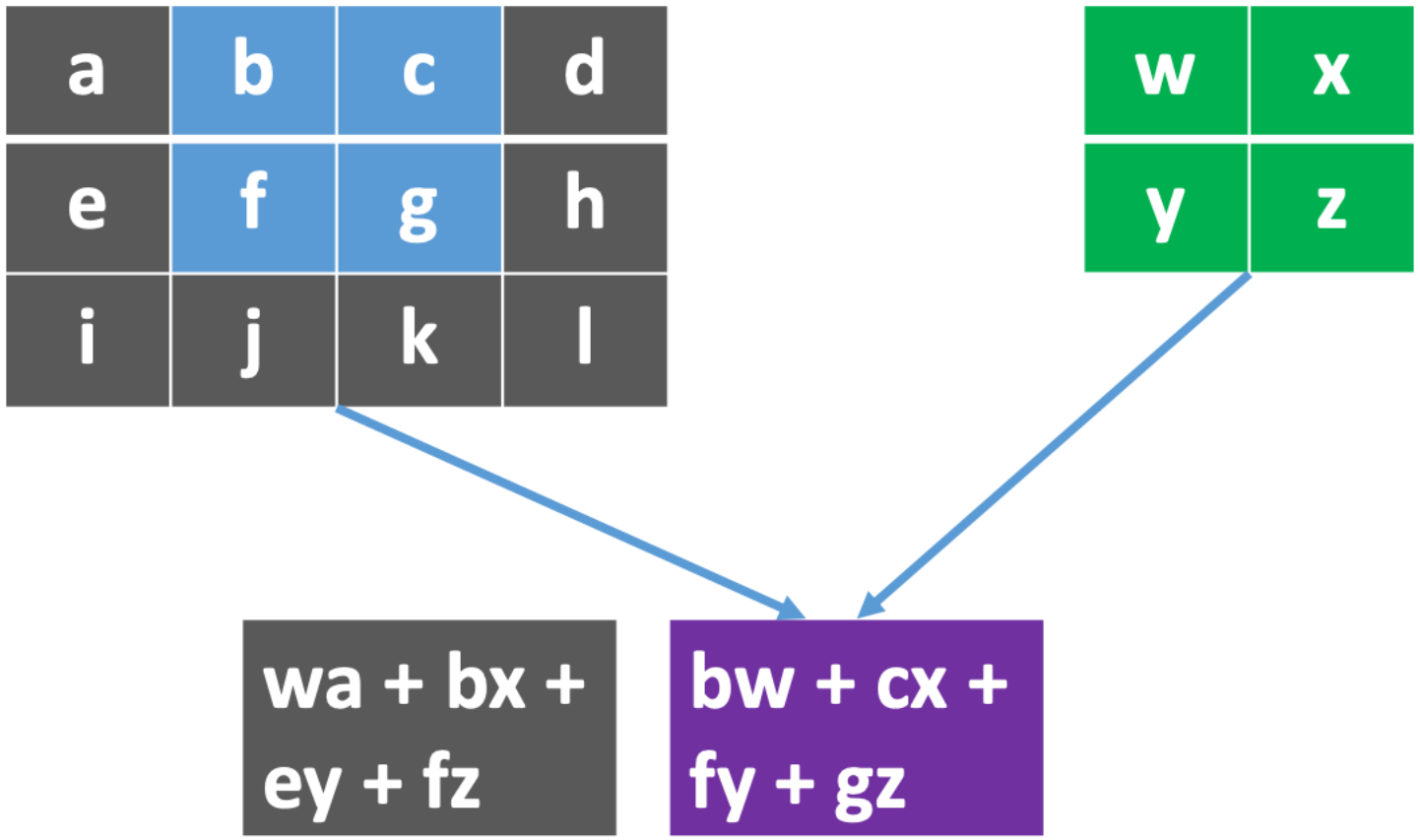
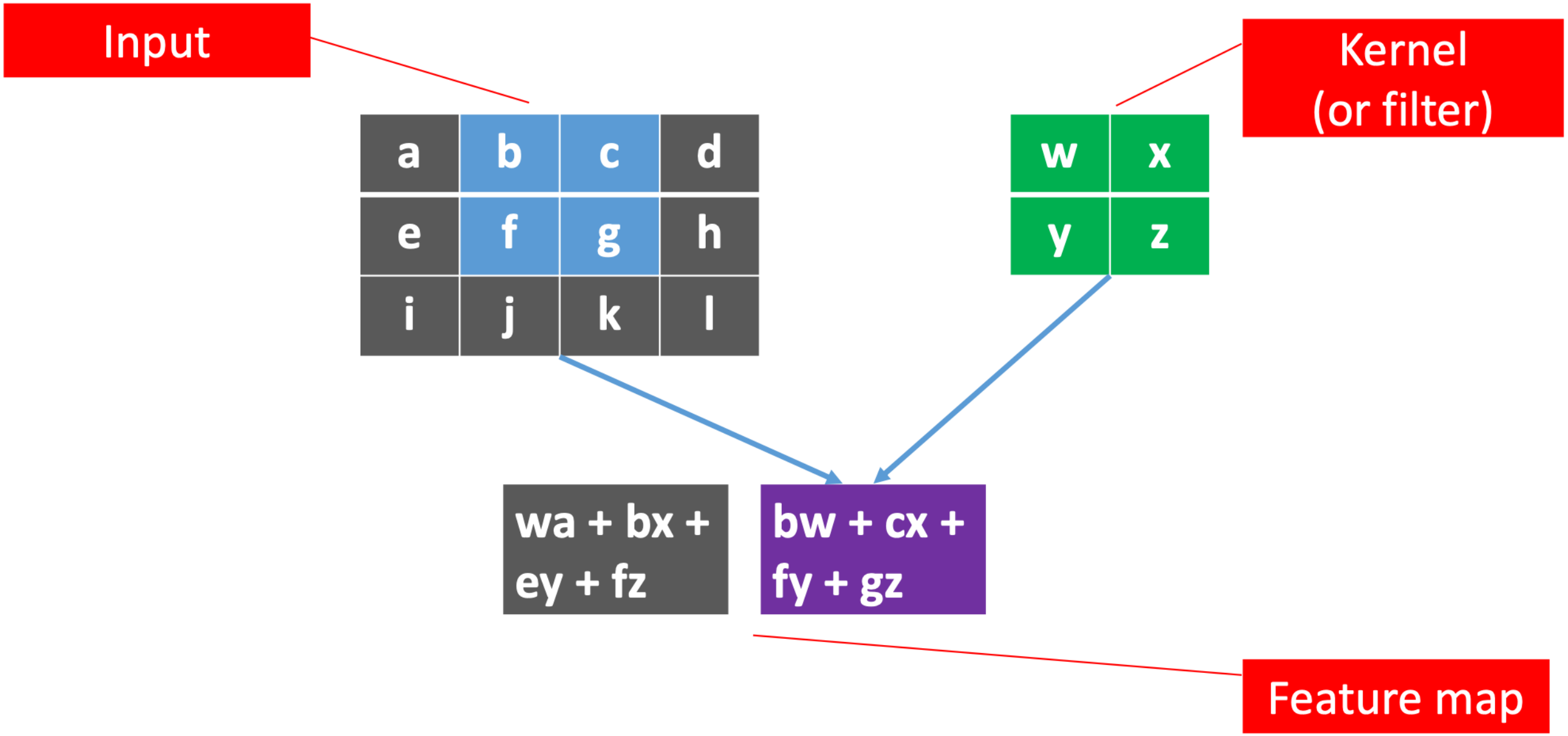


Illustration 2



CNN Architecture: Convolutional Layer

Convolutional Layer: core **CNN** layer which consists of a set of **filters**:

- each filter covers a spatially small portion of the input data

Each filter is **convolved** across the dimensions of input data, producing a **multidimensional feature map**:

- involves computing **DOT product** between parameters of the filter & input.

Network will "**learn**" filters that activate when they "see" some specific type of feature at some spatial position.

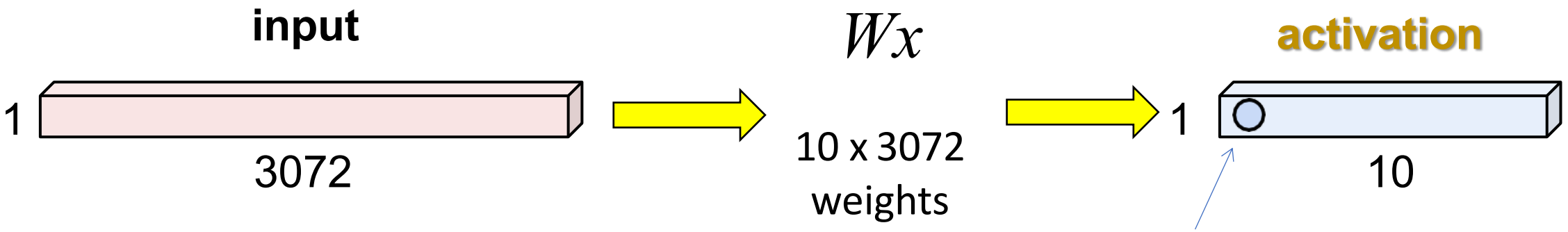
Key architectural characteristics: *local connectivity* & *shared weights*.

Architecture ensures that learnt feature extractors produce strongest response to a spatially local input pattern.

Fully Connected Layer

32x32x3 image ⇨ *stretch to 3072 x 1*

each neuron looks at the full input volume



result of taking **DOT** product between a row of *W* & input ⇨ *a 3072-dimensional **DOT** pdt*

Convolution …

3_0	3_1	2_2	1	0
0_2	0_2	1_0	3	1
3_0	1_1	2_2	2	3
2	0	0	2	2
2	0	0	0	1

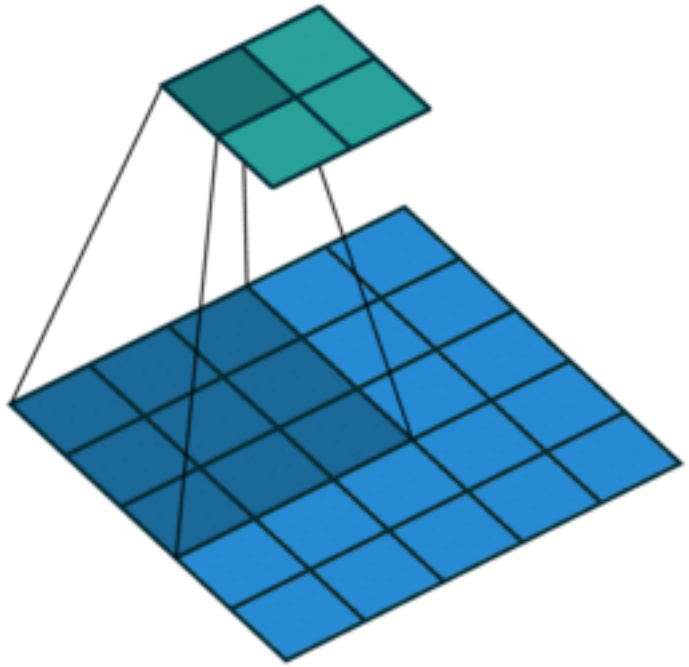
12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

5x5 input (spatially)
assume 3x3 filter

⇨ **3x3 output**

A closer look at spatial dimensions:

Convolution …

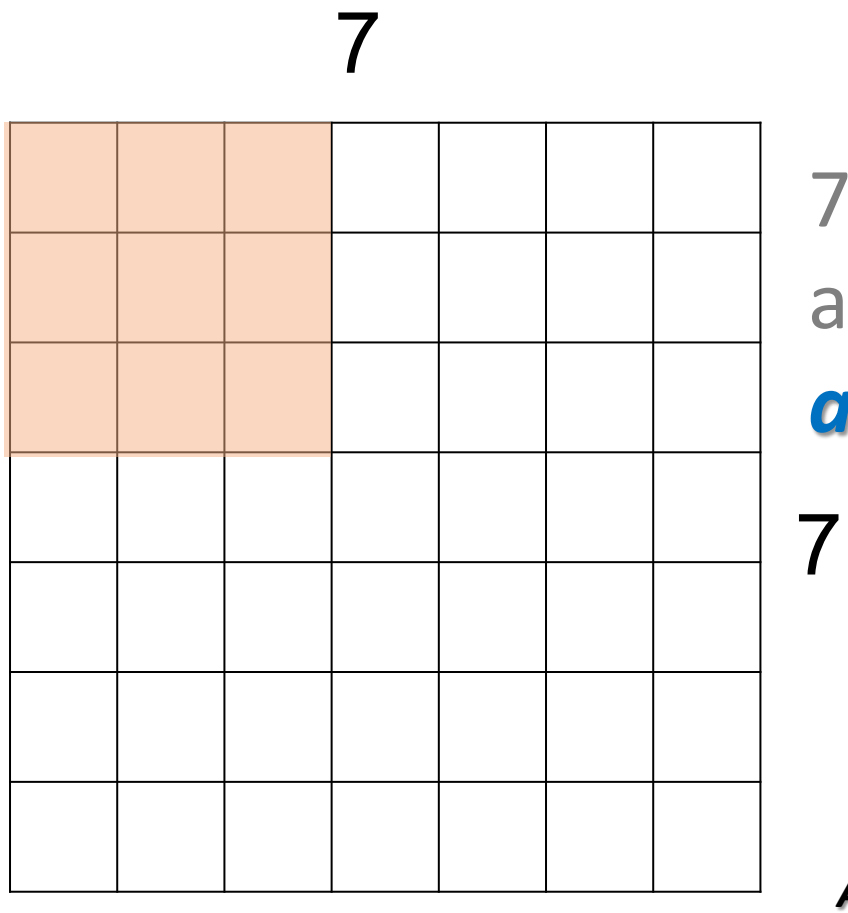


5x5 input (spatially)
assume 3x3 filter
applied with stride 2

⇨ **2x2 output**

A closer look at spatial dimensions:

Convolution …

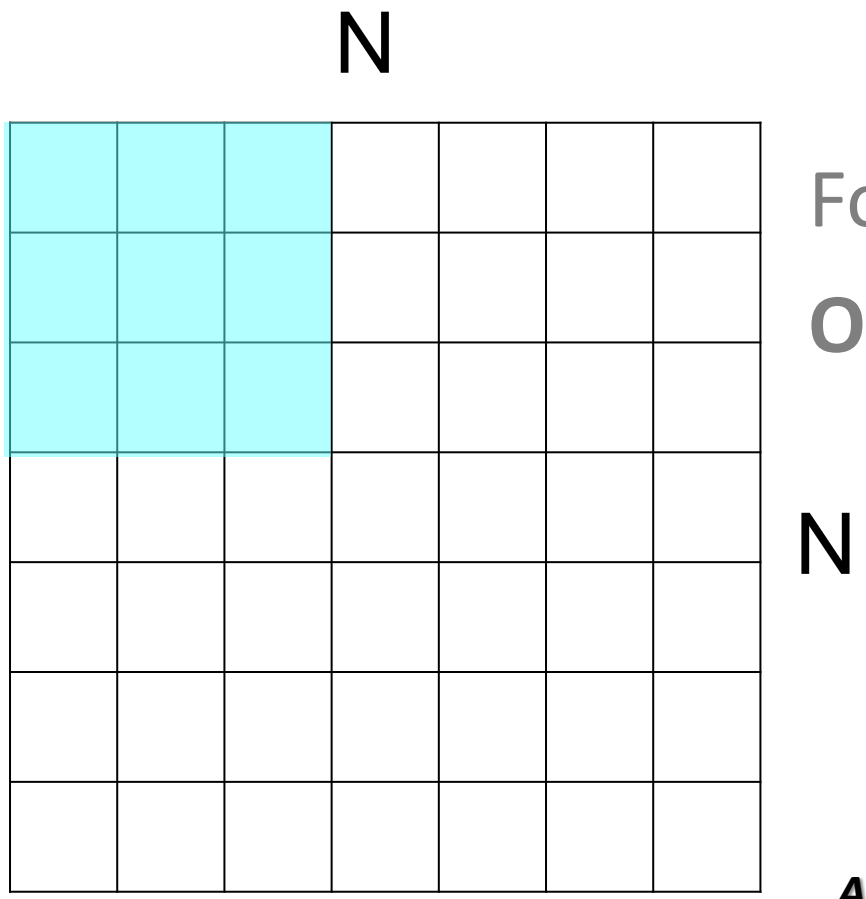


7x7 input (spatially)
assume 3x3 filter
applied with stride 3??

⇨ **doesn't fit! … CANNOT apply 3x3 filter on 7x7 input with stride 3**

A closer look at spatial dimensions:

Convolution …



For *N*x*N* input with a *F*x*F* filter, the

Output size: $\frac{N-F}{STRIDE}+1$

⇨ say N = 7, F = 3:

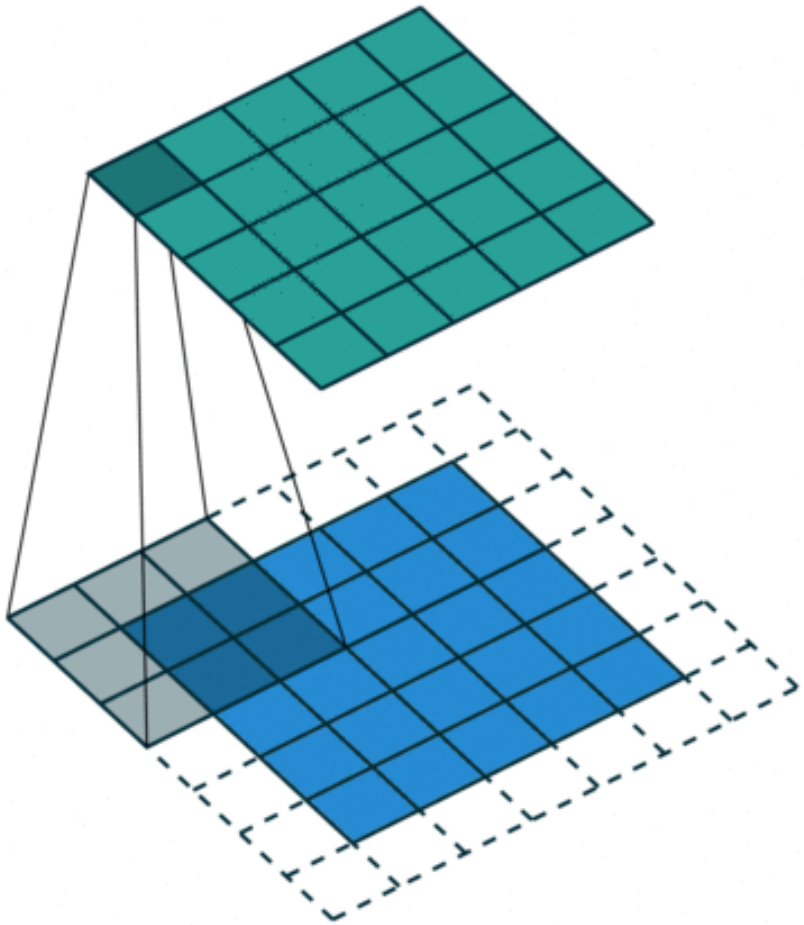
stride 1 ⇨ $\frac{7-3}{1}+1 = 5$

stride 2 ⇨ $\frac{7-3}{2}+1 = 3$

stride 3 ⇨ $\frac{7-3}{3}+1 = 2.33..?$

A closer look at spatial dimensions:

Zero pad the border… common



5x5 input :
3x3 filter applied with stride 1 on input padded with **1 pixel border**

⇨ output **??** : $\frac{N-F}{STRIDE}+1$ ⇨ **5x5**

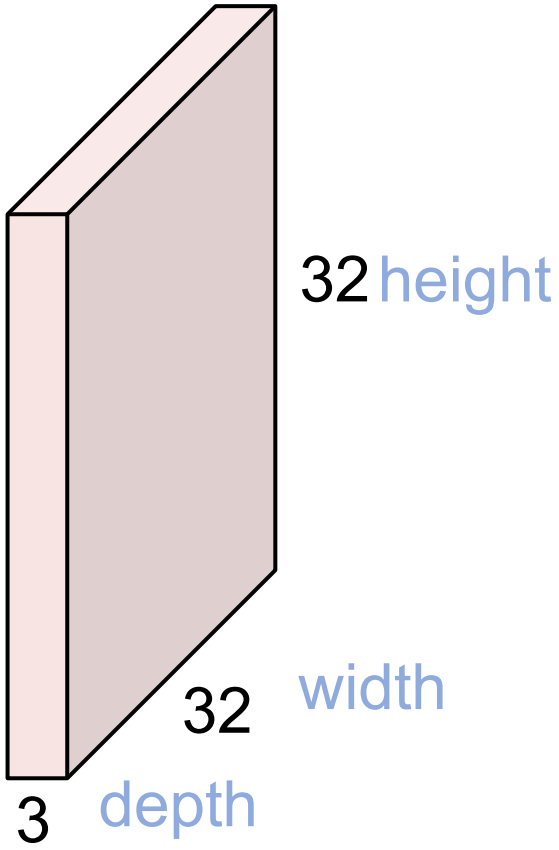
Common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2 ; preserves size spatially..

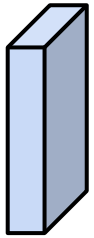
e.g. F = 3 ⇨ zero pad with 1
F = 5 ⇨ zero pad with 2
F = 7 ⇨ zero pad with 3

A closer look at spatial dimensions:

Convolution Layer

32x32x3 image → preserve spatial structure

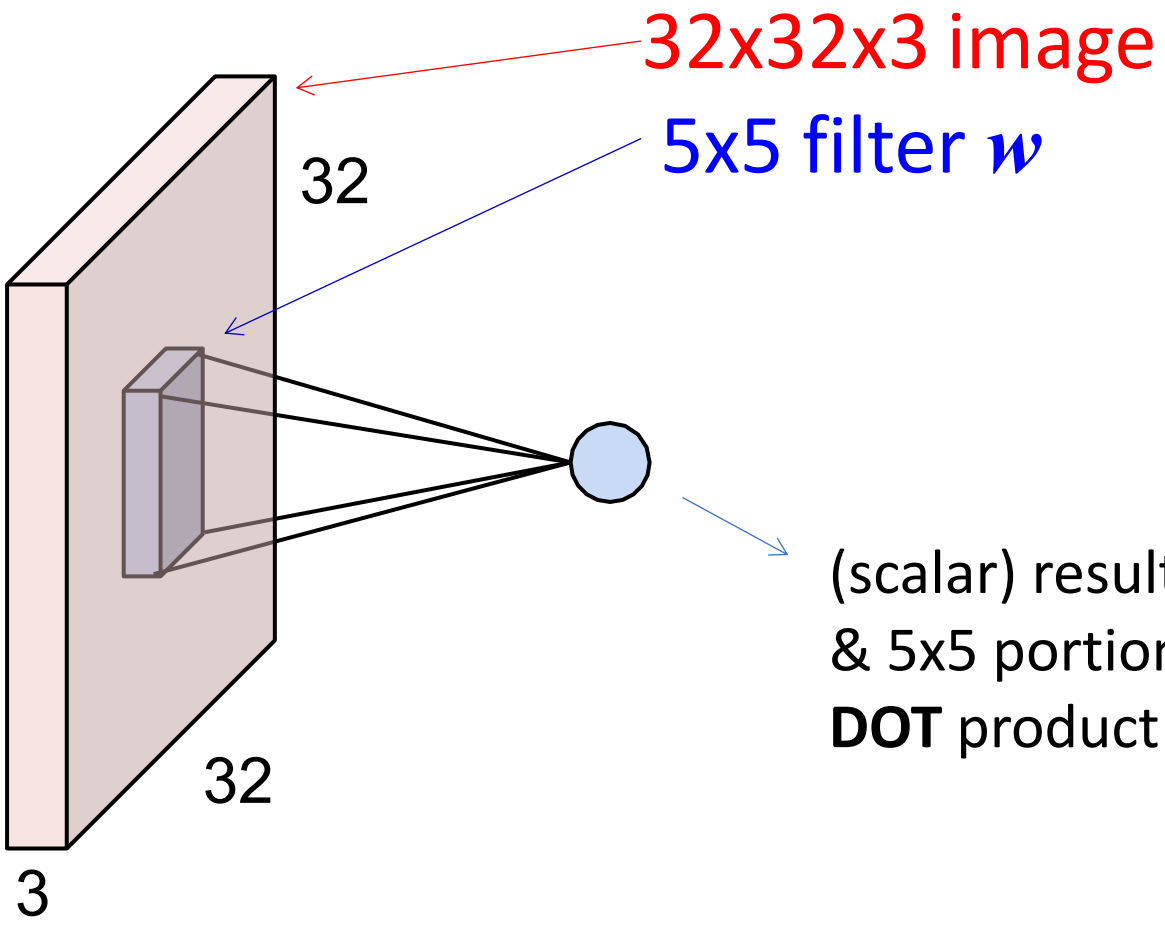




5x5 filter

Convolve Filter with the image .. "slide" over image spatially; computing **DOT products.**

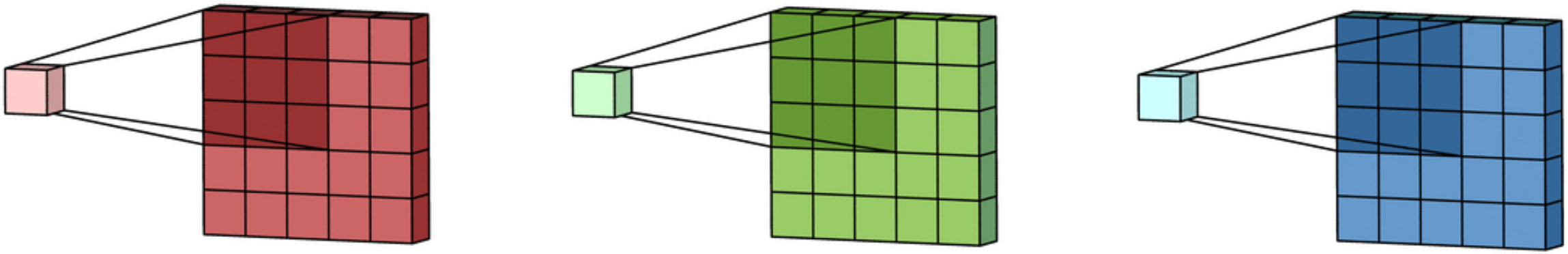
Convolution Layer



(scalar) result of taking **DOT product** between filter & 5x5 portion of image i.e., 5x5x3 = 75-dimensional **DOT** product + bias:

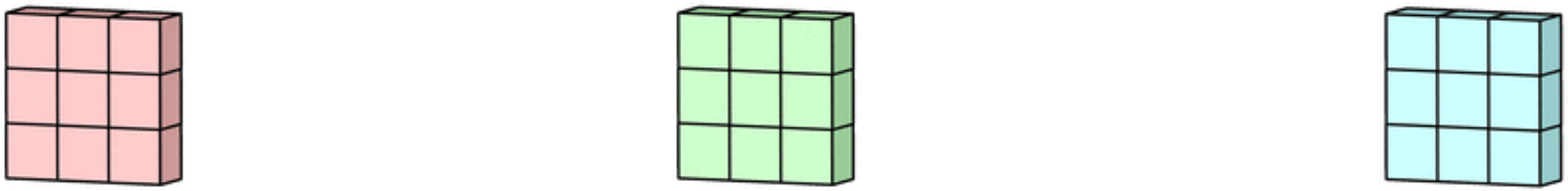
$$w^T x + b$$

Convolution Layer



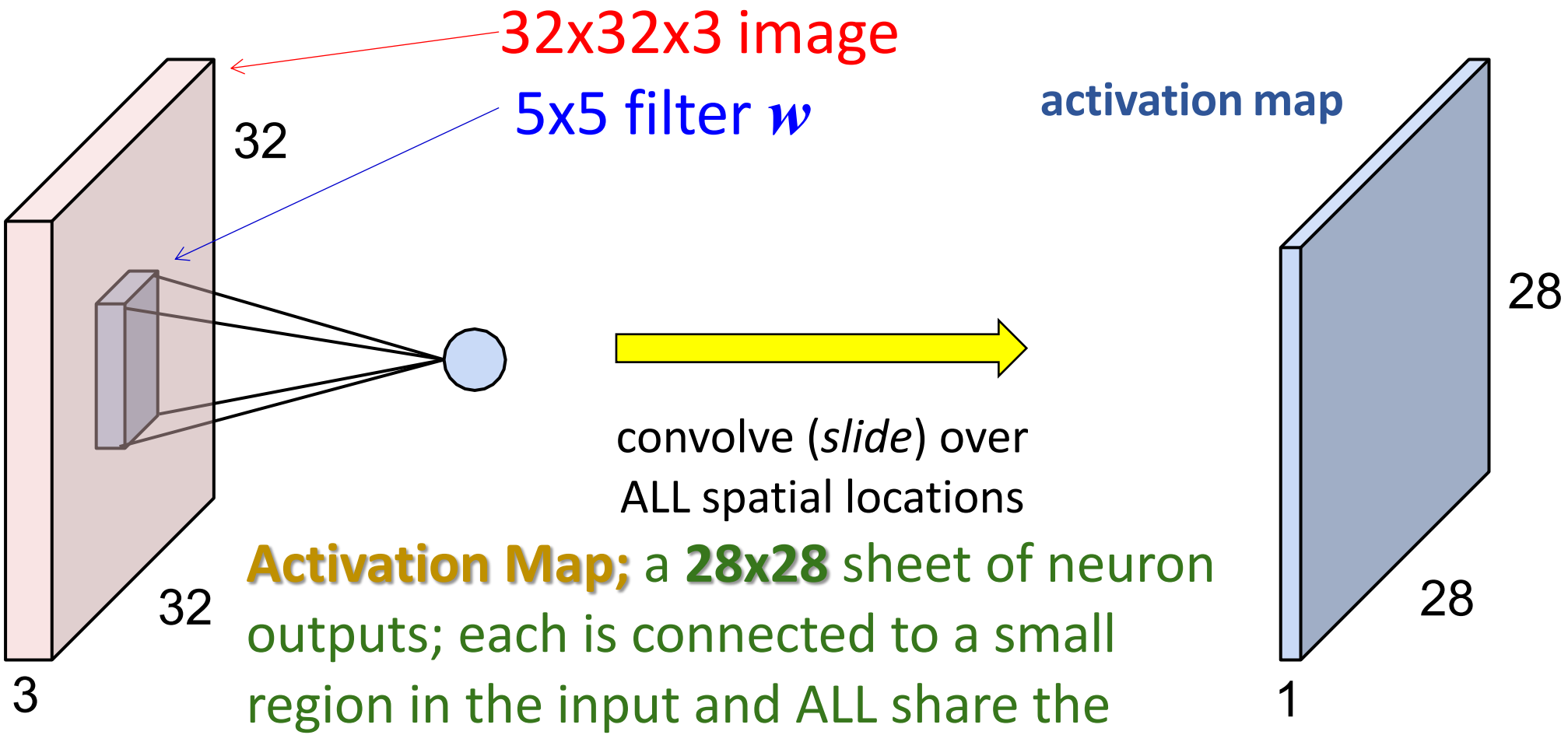
Each of the kernels of the filter “slides” over their respective input channels, producing a processed version of each.

Convolution Layer



Each of the per-channel processed versions are then summed together to form *one* channel.

Convolution Layer

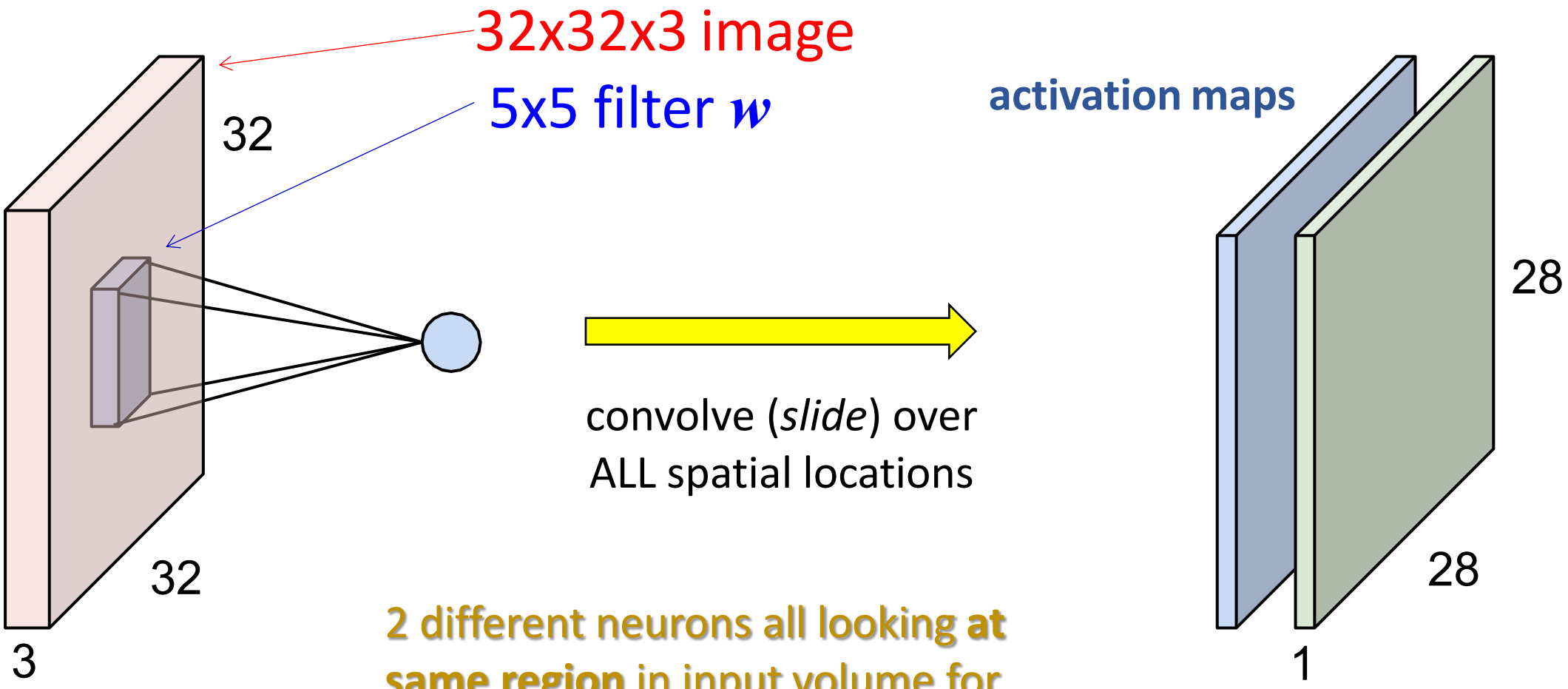


Activation Map; a **28x28** sheet of neuron outputs; each is connected to a small region in the input and ALL share the parameters…

"**5x5 filter**" ⇨5x5 receptive field for each neuron..

consider a second filter

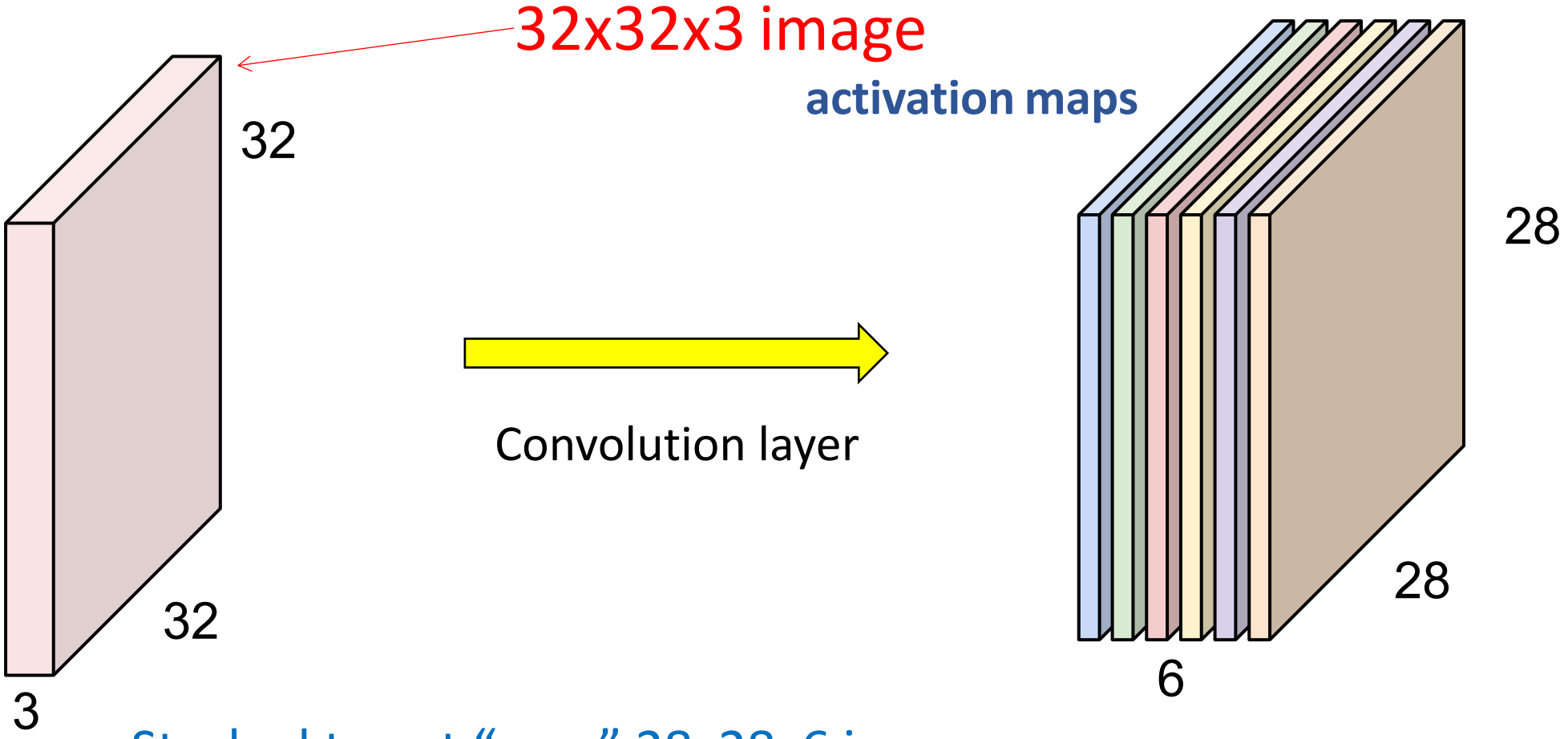
Convolution Layer



2 different neurons all looking **at same region** in input volume for different details..

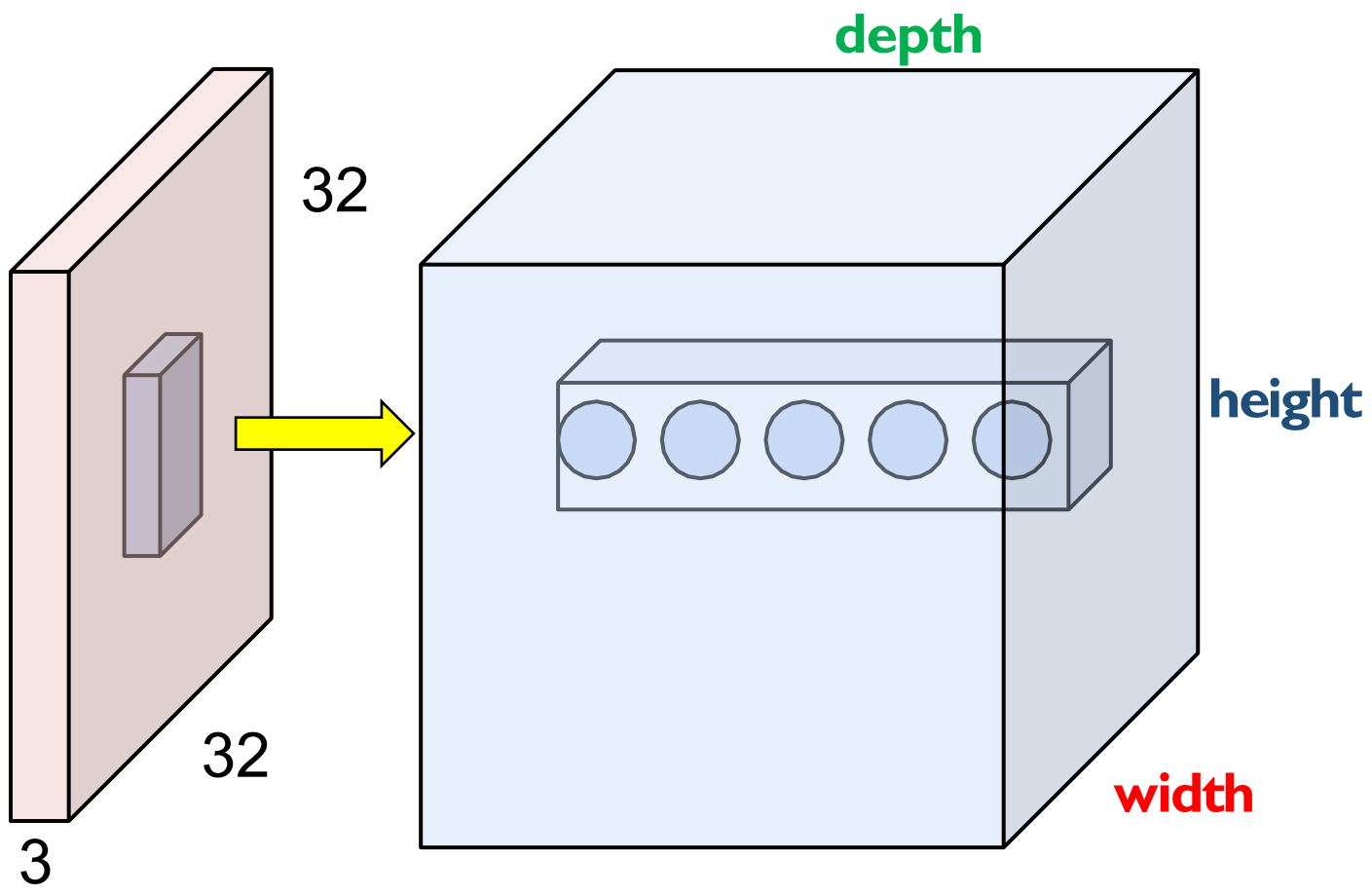
Convolution Layer

SIX 5x5 filters, ⇨ SIX separate activation maps



Stacked to get “new” 28x28x6 image..

CNNs Spatial Arrangement of Output Volume



Layer Dimensions:

h x w x d

where h and w are spatial dimensions

d (depth) = number of filters

Stride:

Filter step size

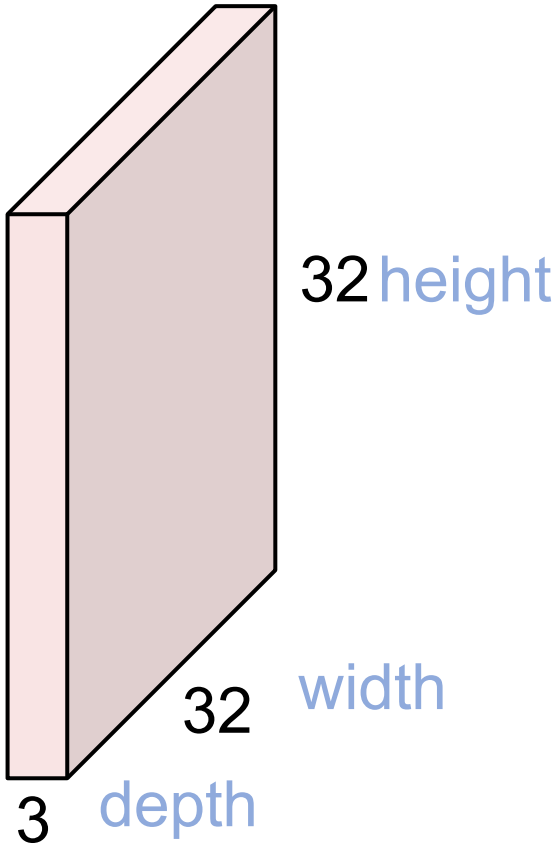
Receptive Field:

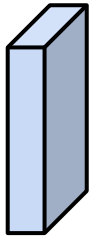
Locations in input image that a node is path connected to

Convolution Layer: Another Type

32x32x3 image → preserve spatial structure

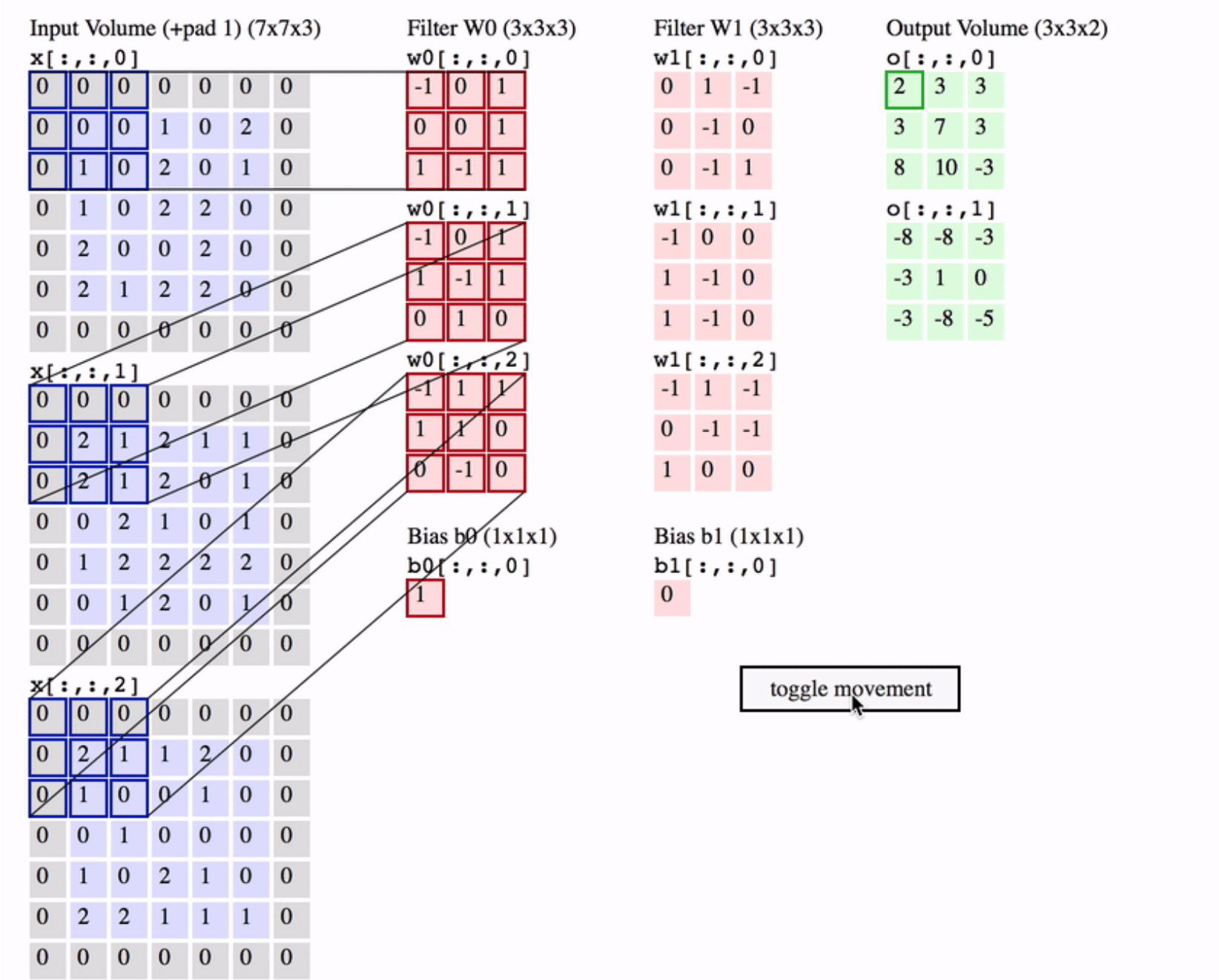
filters extend to full
depth of input volume



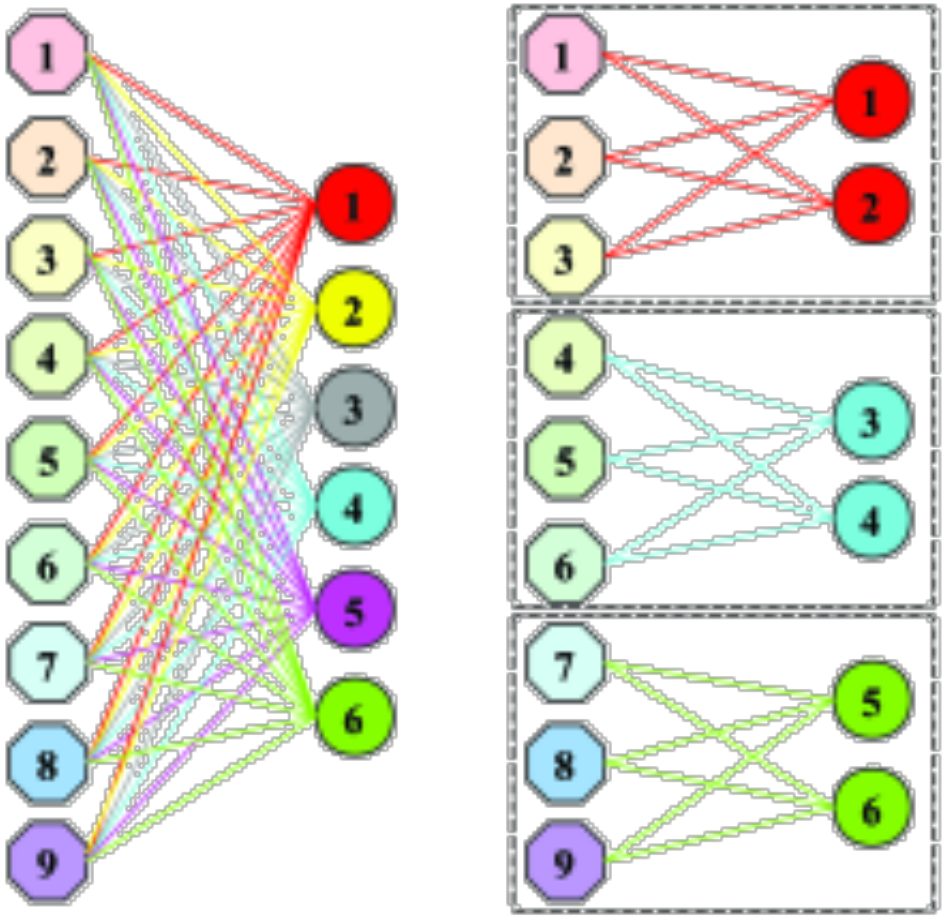


5x5x3 filter

Convolution Layer: Another Type



Pros vs Cons

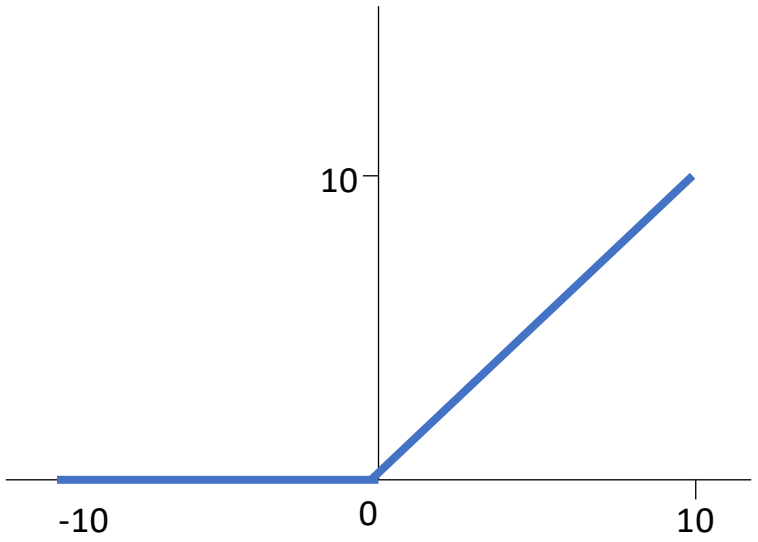


(a) Normal conv (b) Standard group conv

Convolution Layer.. non-linearity

- Apply after every convolution operation (i.e., after convolutional layers)
- **ReLU**: pixel-by-pixel operation that replaces all negative values by zero. **Non-linear operation**

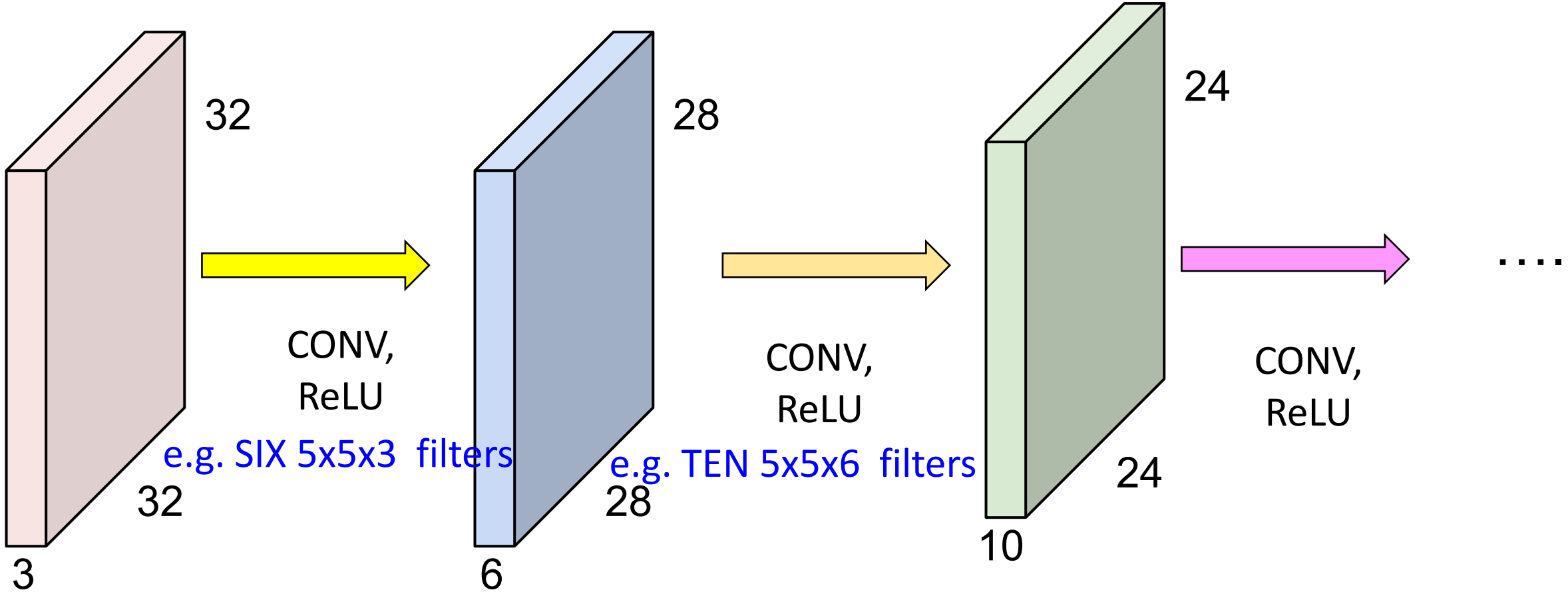
Rectified Linear Unit (ReLU)



$$g(z) = \max(0, z)$$

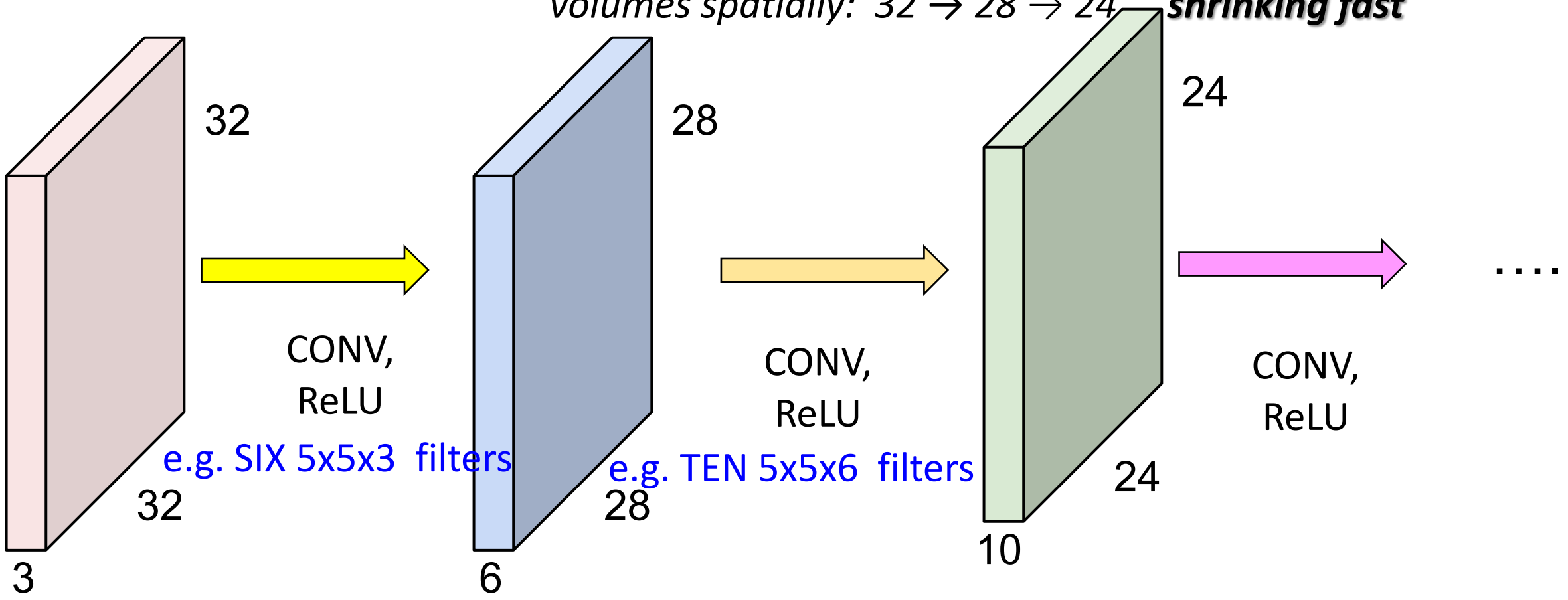
Convolution Layers.. ConvNet

ConvNet: *a sequence of* ***Convolution Layers****, interspersed with* *Activation functions* *(eg ReLU) .. each producing an output to the next* ***Convolution Layer****..*



Convolution Layers.. ConvNet

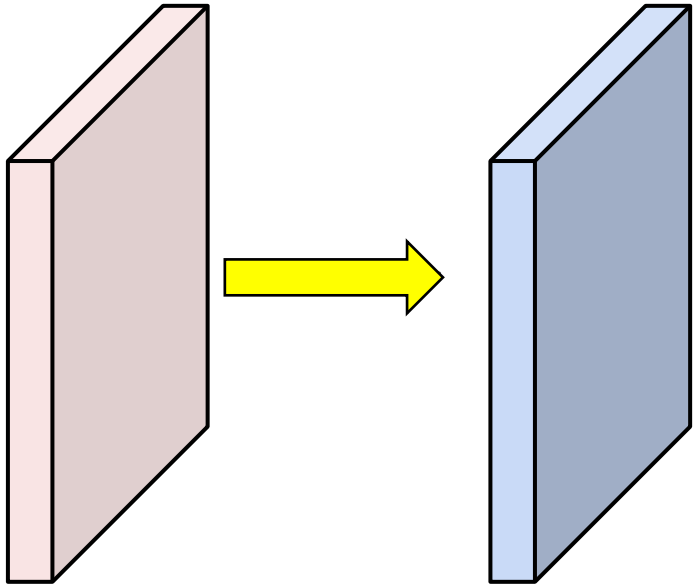
32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially: 32 → 28 → 24 ***shrinking fast***



Convolution.. example

Input volume: **32x32x3**

10 **5x5x3** filters with stride 1, pad 2



Output volume ⇨

$$\frac{(N=32)+(PAD=2\times2)-(F=5)}{STRIDE=1}+1=32$$

⇨ **32x32x10**

Number of **parameters** in this layer?

⇨ each filter has **5x5x3+1=76** **parameters**.. (1 for bias)

⇨ **76x10=760**

CONV Layer… summary

ACCEPTS a volume of size: W_1 x H_1 x D_1

Requires FOUR hyper-parameters:

- number of filters K
- spatial extent F
- stride S
- amount of zero padding P

Common settings:

K = (powers of 2 : 32, 64, 128, 512)

- F = 3, S = 1, P = 1
- F = 5, S = 1, P = 2
- F = 1, S = 1, P = 0

Produces a volume of size: W_2 x H_2 x D_2 where:

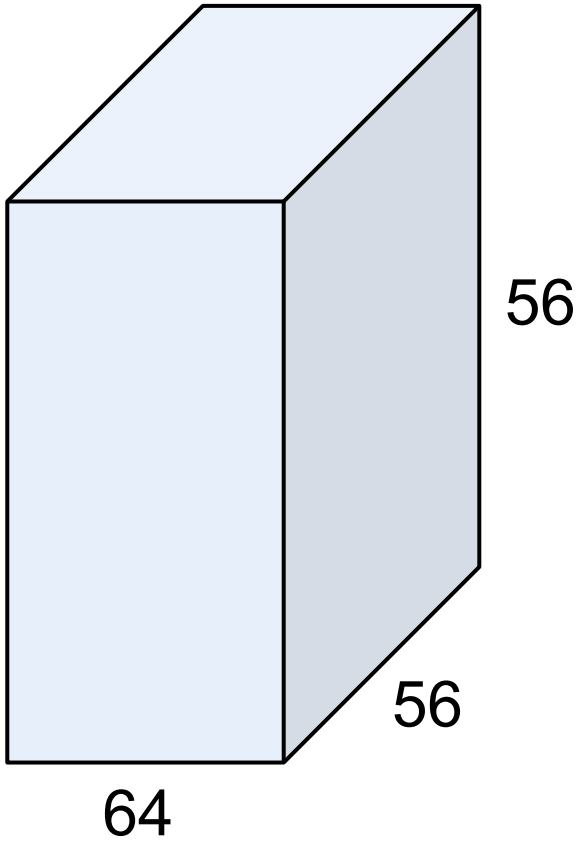
- $W_2 = (W_1 - F + 2P) / S + 1$
- $H_2 = (H_1 - F + 2P) / S + 1$
- $D_2 = K$

CONV Layer… summary

With parameter sharing, it produces $F.\ F.\ D_1$ weights per filter for a total of $(F.\ F.\ D_1\ ).\ K$ weights and K biases

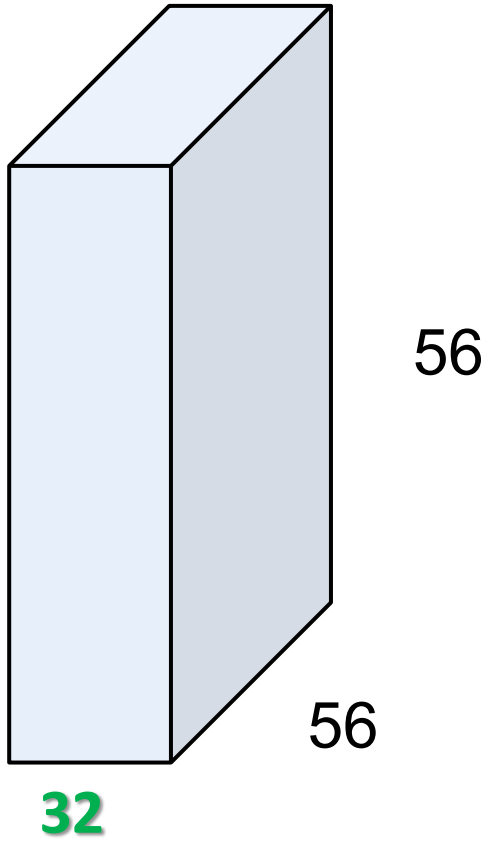
In the output volume, the d th depth slice of size W_2 x H_2 is the result of performing a valid convolution of d th filter over the input volume with a stride of S and then offset by d th bias.

Convolution.. 1x1 CONV



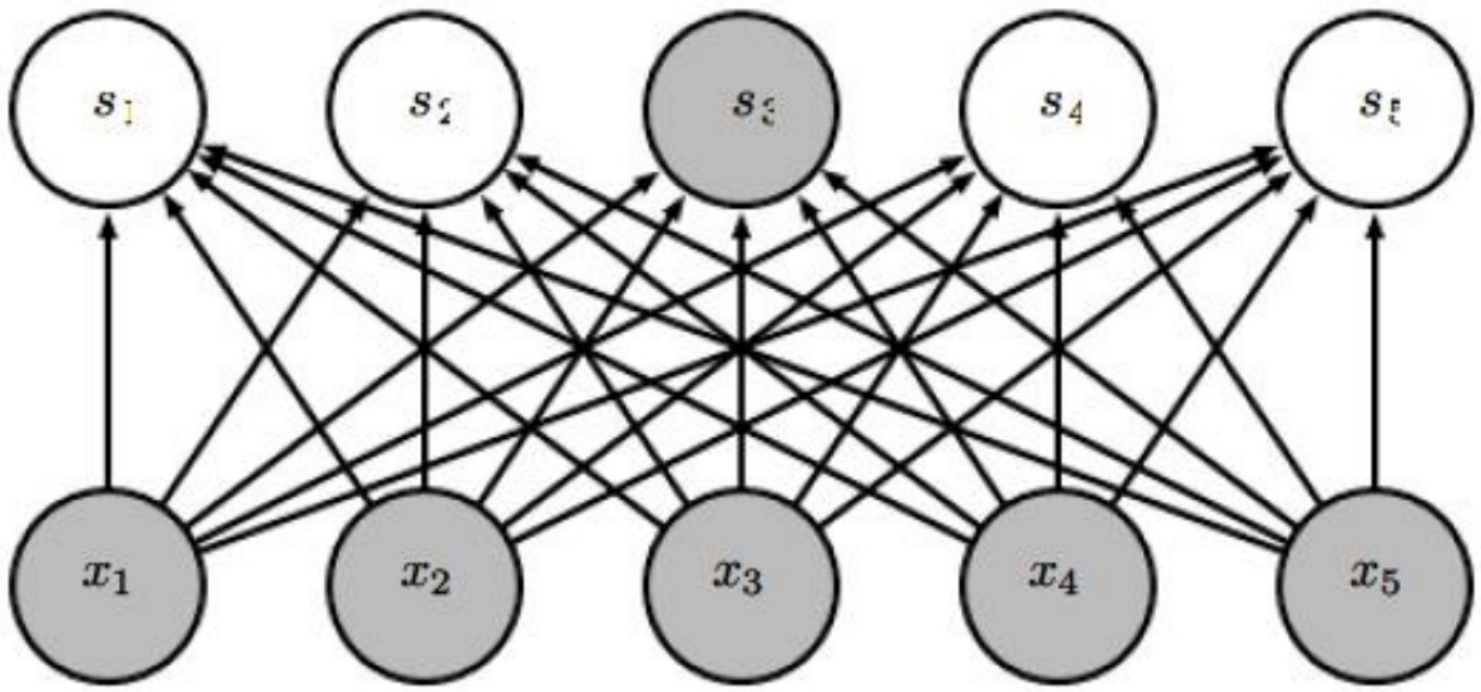
1x1 CONV
with **32** filters

Each filter size: 1x1x64
Performs a 64-dimensional
DOT product..



Advantage: sparse interaction

Fully connected layer, $m \times n$ edges



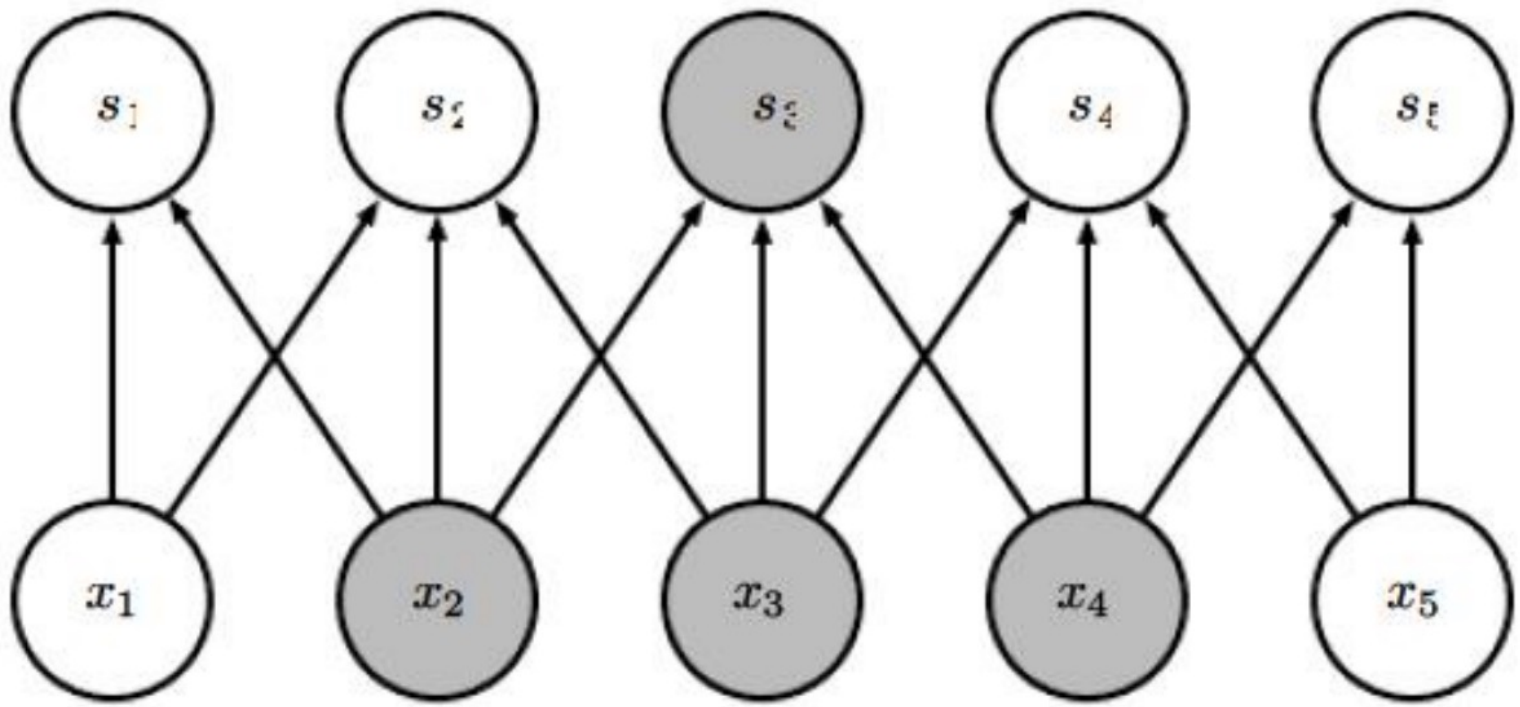
m output nodes

n input nodes

Figure from *Deep Learning,* by Goodfellow, Bengio, and Courville

Advantage: sparse interaction

Convolutional layer, $\leq m \times k$ edges



m output nodes

k kernel size

n input nodes

Figure from *Deep Learning,* by Goodfellow, Bengio, and Courville

Advantage: sparse interaction

Multiple convolutional layers: larger receptive field

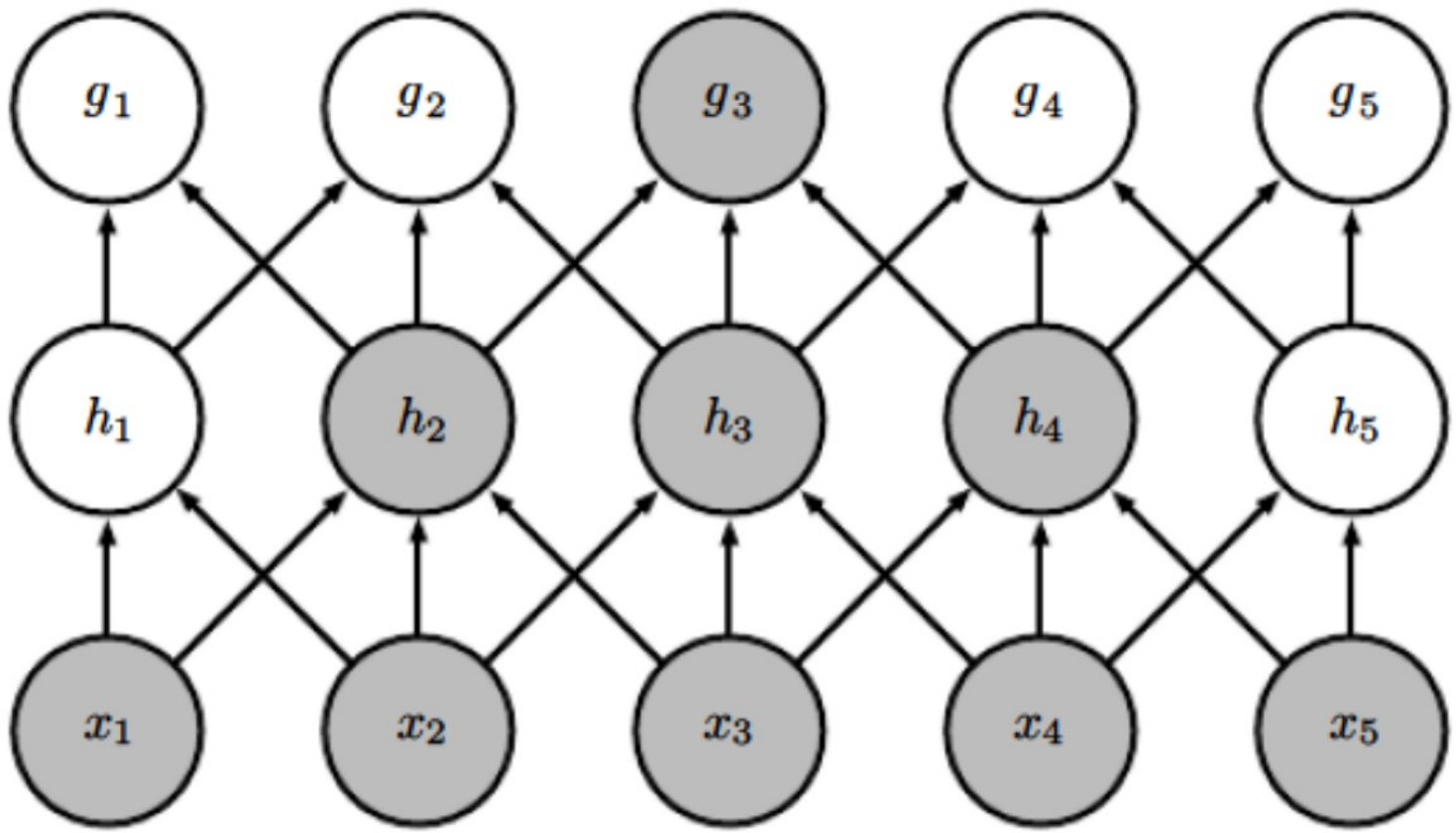
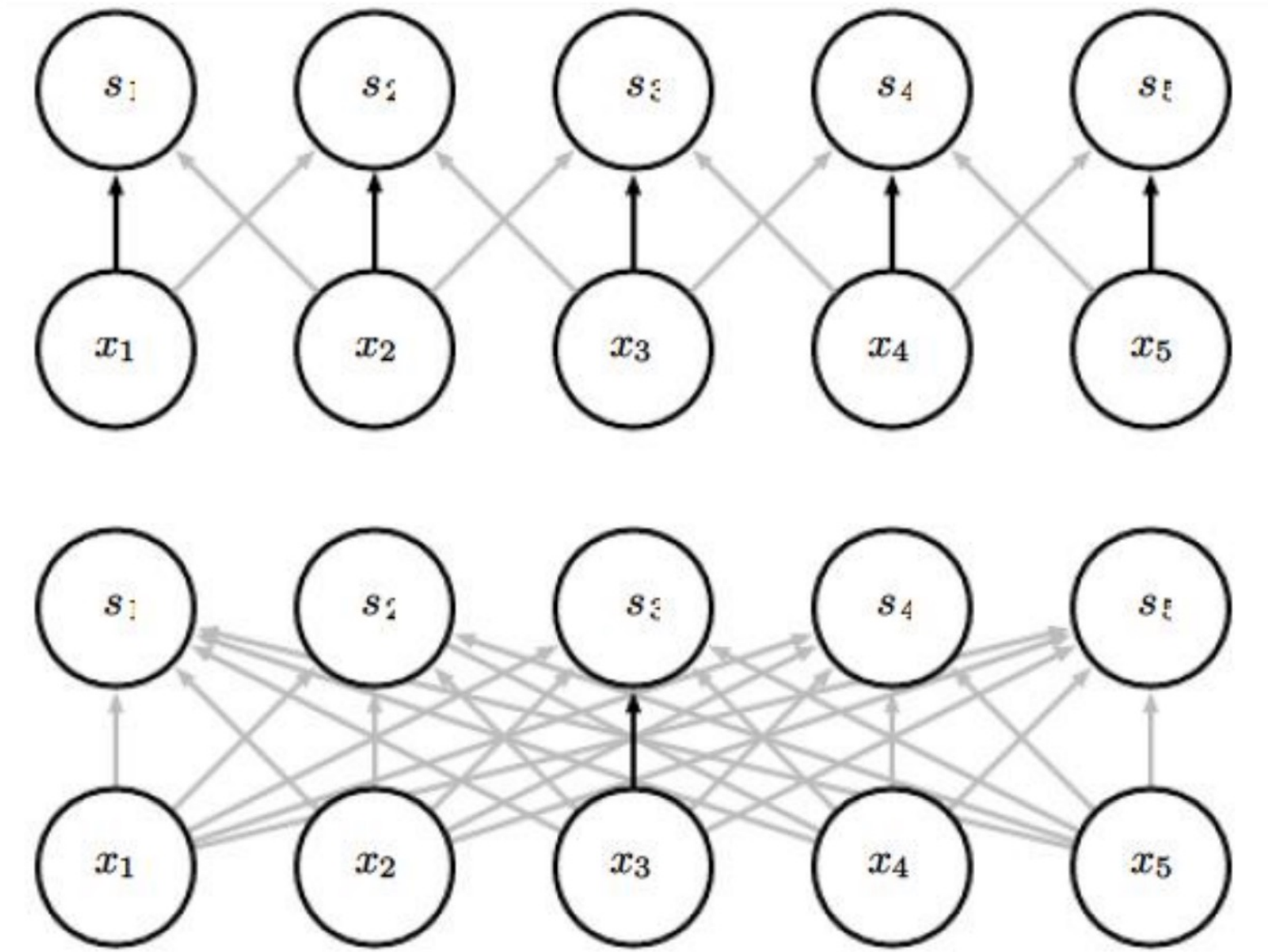


Figure from *Deep Learning,* by Goodfellow, Bengio, and Courville

Advantage: parameter sharing



The same kernel are used repeatedly. E.g., the black edge is the same weight in the kernel.

Figure from *Deep Learning,* by Goodfellow, Bengio, and Courville

Advantage: equivariant representations

- Equivariant: transforming the input = transforming the output

- Example: input is an image, transformation is shifting
- Convolution(shift(input)) = shift(Convolution(input))

- Useful when care only about the existence of a pattern, rather than the location

Translation Invariance

Rotation/Viewpoint Invariance

Size Invariance

Illumination Invariance

Matt Krause
mattkrau.se

CNN Architecture: Non-Linear Layer

Increases **nonlinearity** of entire architecture without affecting receptive fields of the convolution layer.

A layer of neurons that applies the **non-linear activation function**:

- $f(x) = \max(0, x)$
- $f(x) = \tanh x$
- $f(x) = |\tanh x|$
- $f(x) = (1 + e^{-x})^{-1}$

CNN Architecture: Pooling Layer

To progressively **reduce the spatial size** of the representation to reduce amount of parameters & computations in the network, and hence to also control overfitting.

Pooling partitions the input image into a **set of non-overlapping rectangles** and, for each such sub-region, outputs the **maximum** value of the features in that region (*for max pooling*). Keeps most important information for next layer.

POOLING LAYER

Makes representations smaller & more manageable ..
operates over each activation map ***independently***:

MAX POOLING

Single depth slice

x

1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

y

max pool with 2x2 filters
& stride 2

6	8
3	4

1) Reduced dimensionality
2) Spatial invariance

How to downsample & preserve spatial invariance?

POOLING LAYER

ACCEPTS a volume of size: W_1 x H_1 x D_1

Requires THREE hyper-parameters:

- spatial extent F
- stride S

(not common to use zero padding P)

Common settings:

F = 2, S = 2

F = 3, S = 2

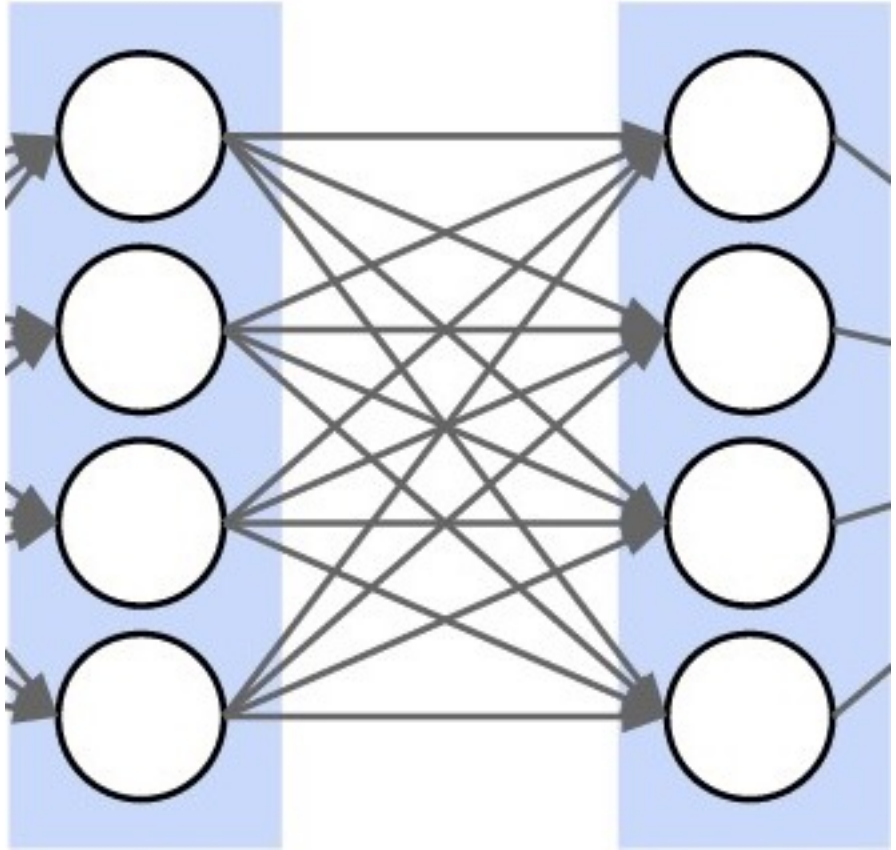
Produces a volume of size: W_2 x H_2 x D_2 where:

- $W_2 = (W_1 - F\ ) / S + 1$
- $H_2 = (H_1 - F\ ) / S + 1$
- $D_2 = D_1$

Introduces ZERO parameters

Fully Connected (FC) Layer

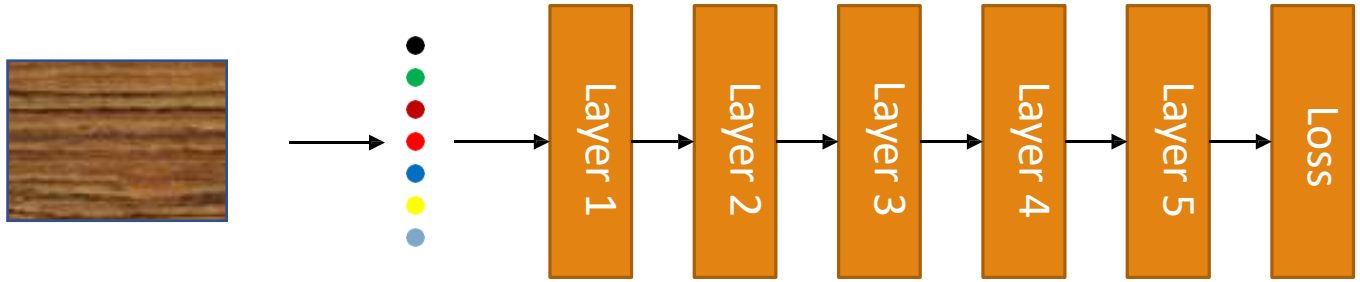
Contains neurons that connect to the entire input volume from final convolutional layer and computes the "**final score**" function.



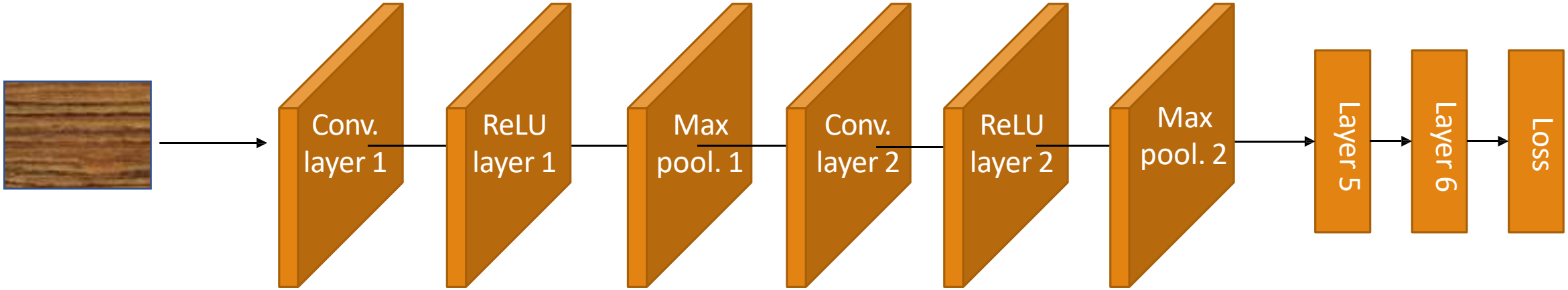
A Fully connected layer is the actual component that does the **discriminative** learning in a Deep Neural Network.

CNN vs Standard Neural Networks

Neural Network



Convolutional Neural Network



Recurrent Neural Networks

Recurrent neural networks

- Dates back to (Rumelhart *et al.*, 1986)
- A family of neural networks for handling sequential data, which involves variable length inputs or outputs

- Especially, for natural language processing (NLP)

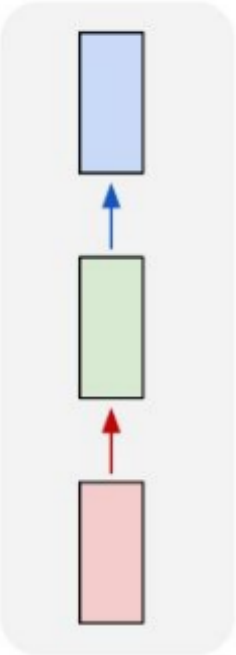
Sequential data

- Each data point: A sequence of vectors $x^{(t)}$, for $1 \leq t \leq \tau$
- Batch data: many sequences with different lengths τ
- Label: can be a scalar, a vector, or even a sequence

- Example
  - Sentiment analysis
  - Machine translation

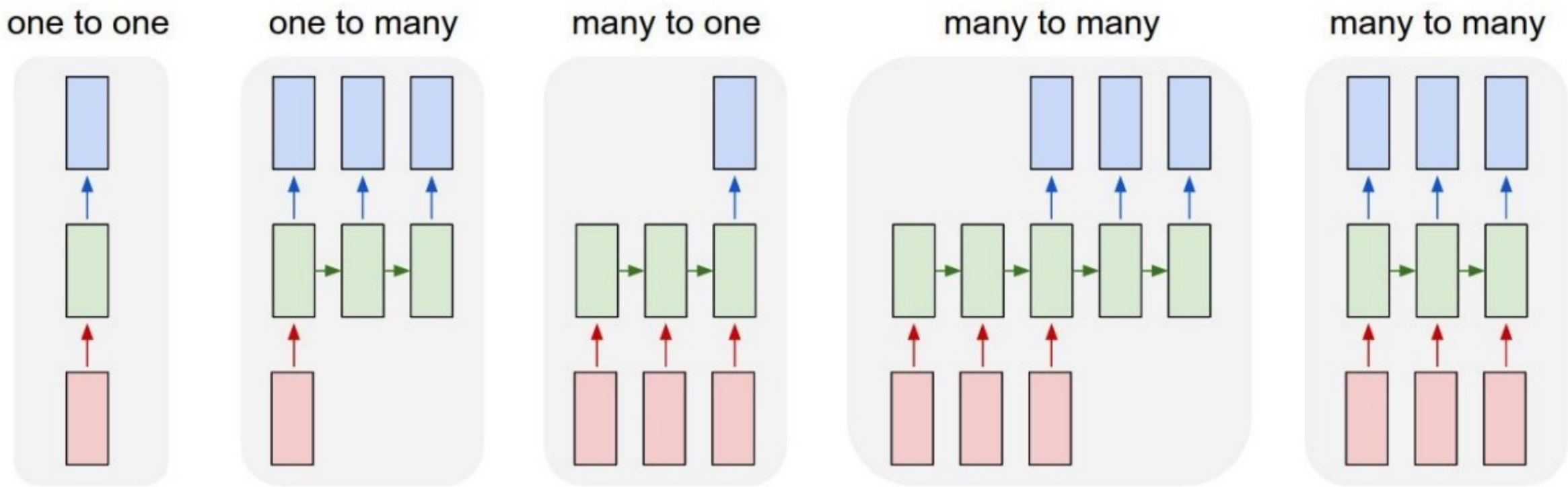
“Vanilla” Neural Network

one to one



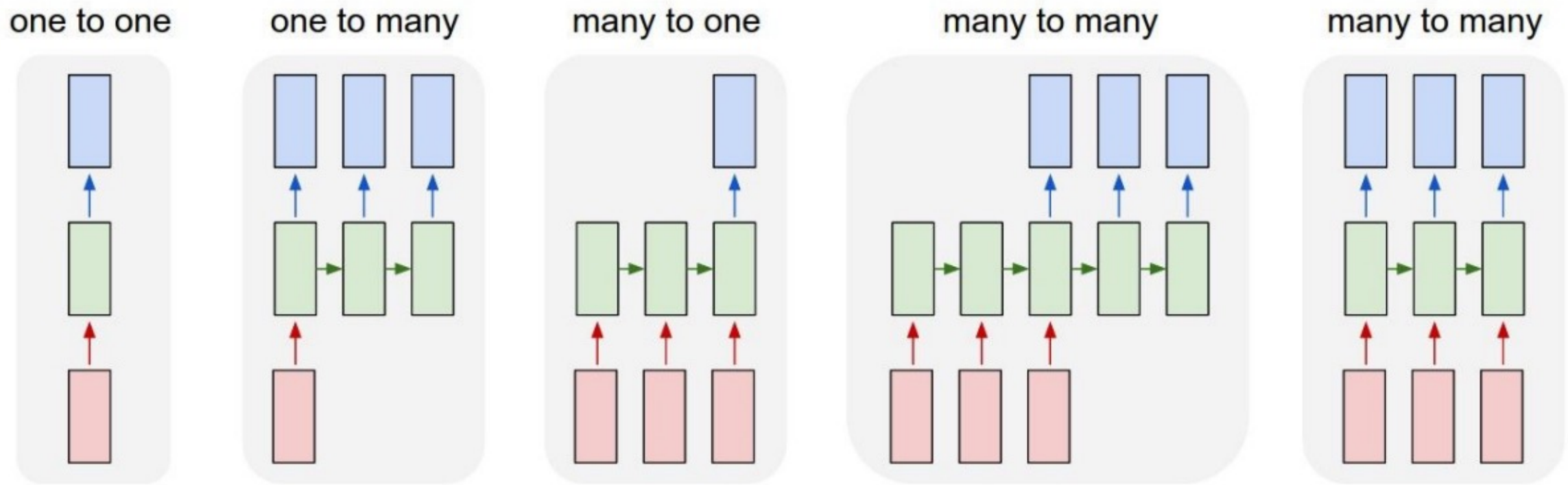
Vanilla Neural Networks

Recurrent Neural Networks: Process Sequences



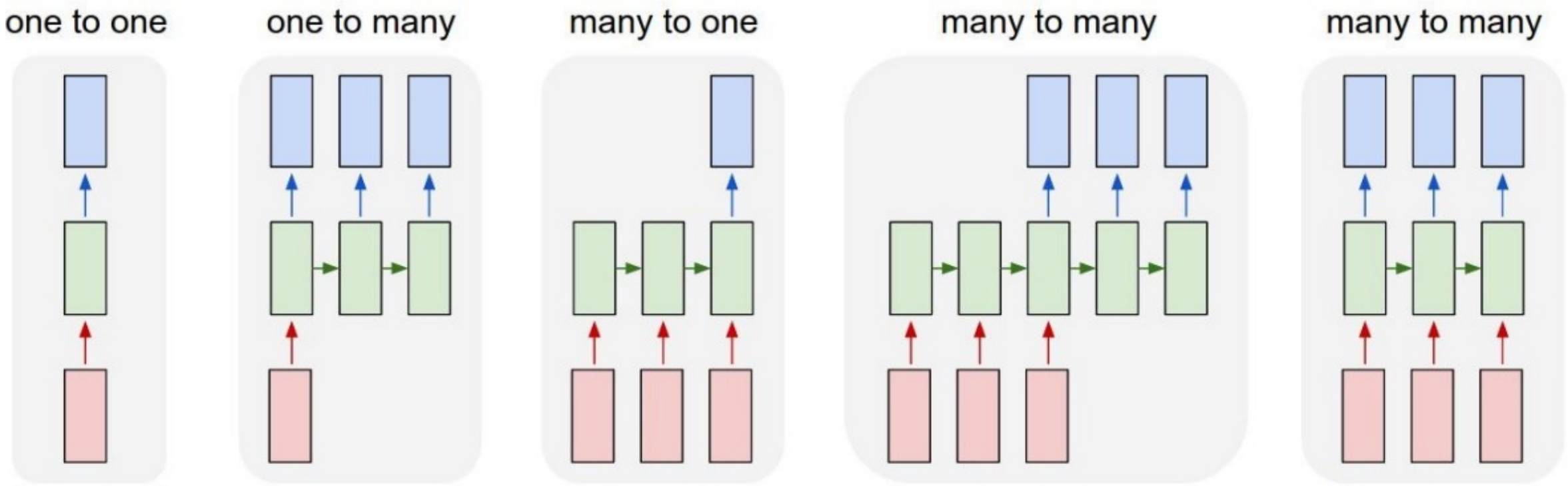
e.g. **Image Captioning**
image -> sequence of words

Recurrent Neural Networks: Process Sequences



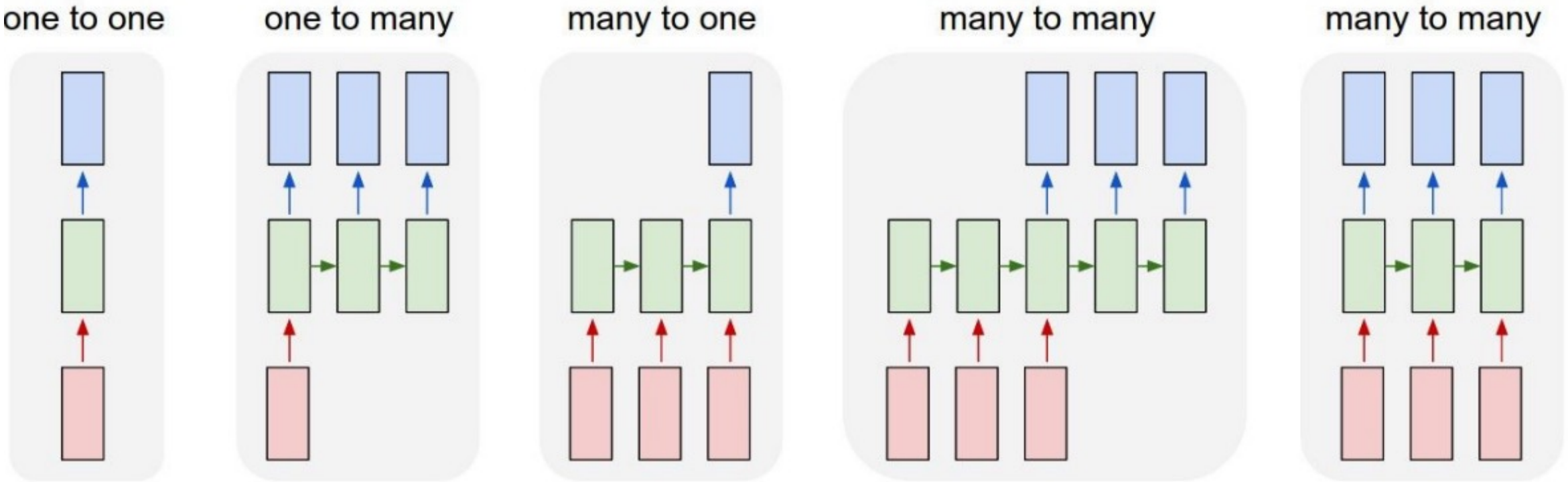
e.g. **Sentiment Classification**
sequence of words -> sentiment

Recurrent Neural Networks: Process Sequences



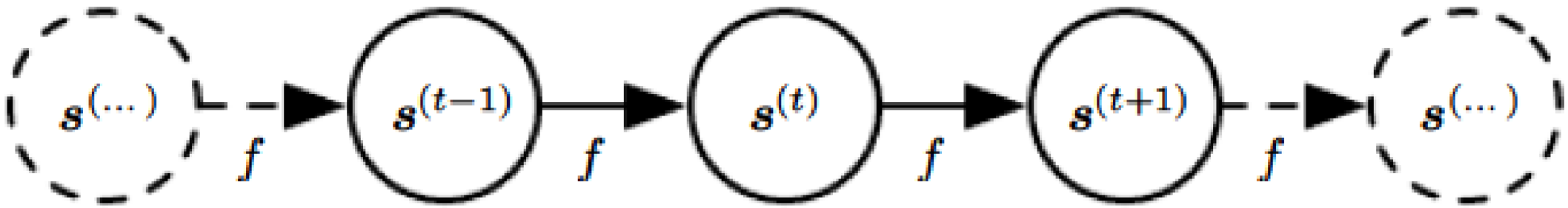
e.g. **Machine Translation**
seq of words -> seq of words

Recurrent Neural Networks: Process Sequences



e.g. **Video classification on frame level**

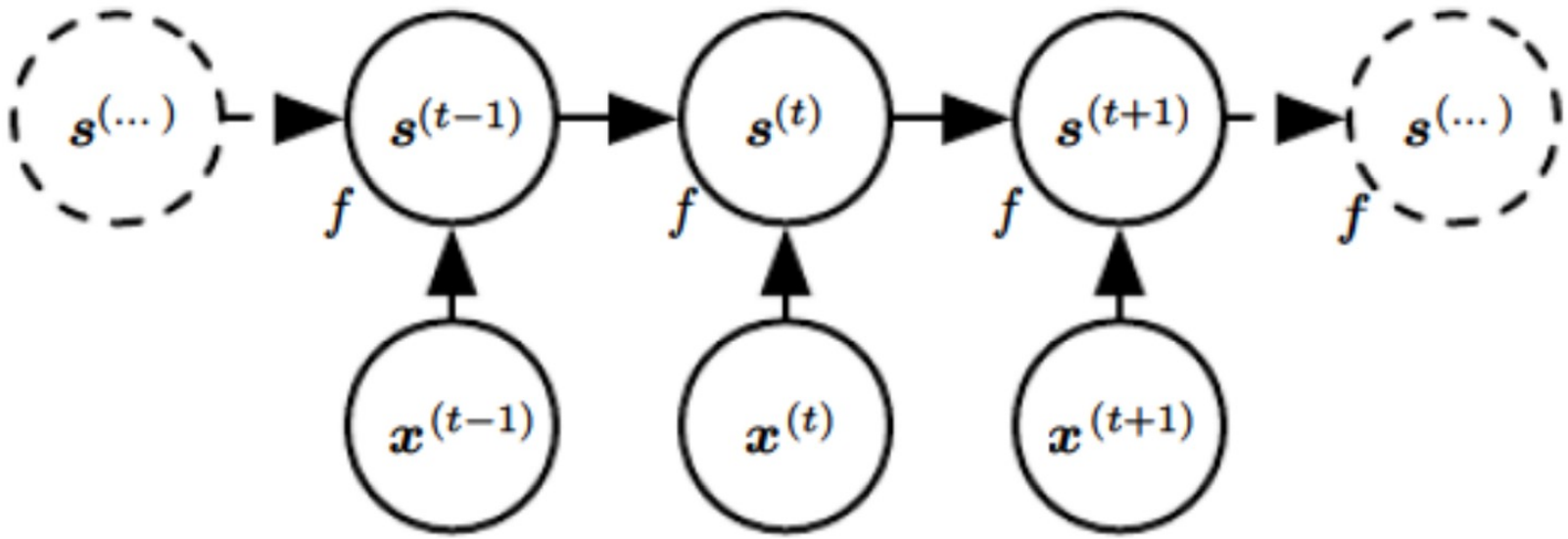
A typical dynamic system



$$s^{(t+1)} = f(s^{(t)}; \theta)$$

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

A system driven by external data

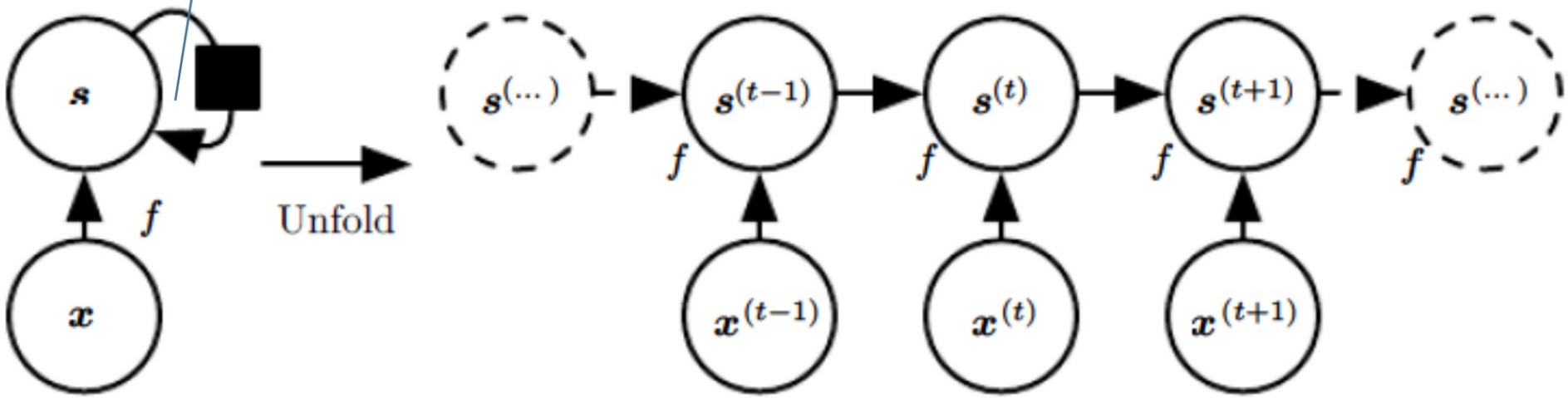


$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Compact view

square: one step time delay



$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

Key: the same f and θ for all time steps

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Recurrent neural networks

- Use the same computational function and parameters across different time steps of the sequence
- Each time step: takes the input entry and the previous hidden state to compute the output entry
- Loss: typically computed every time step

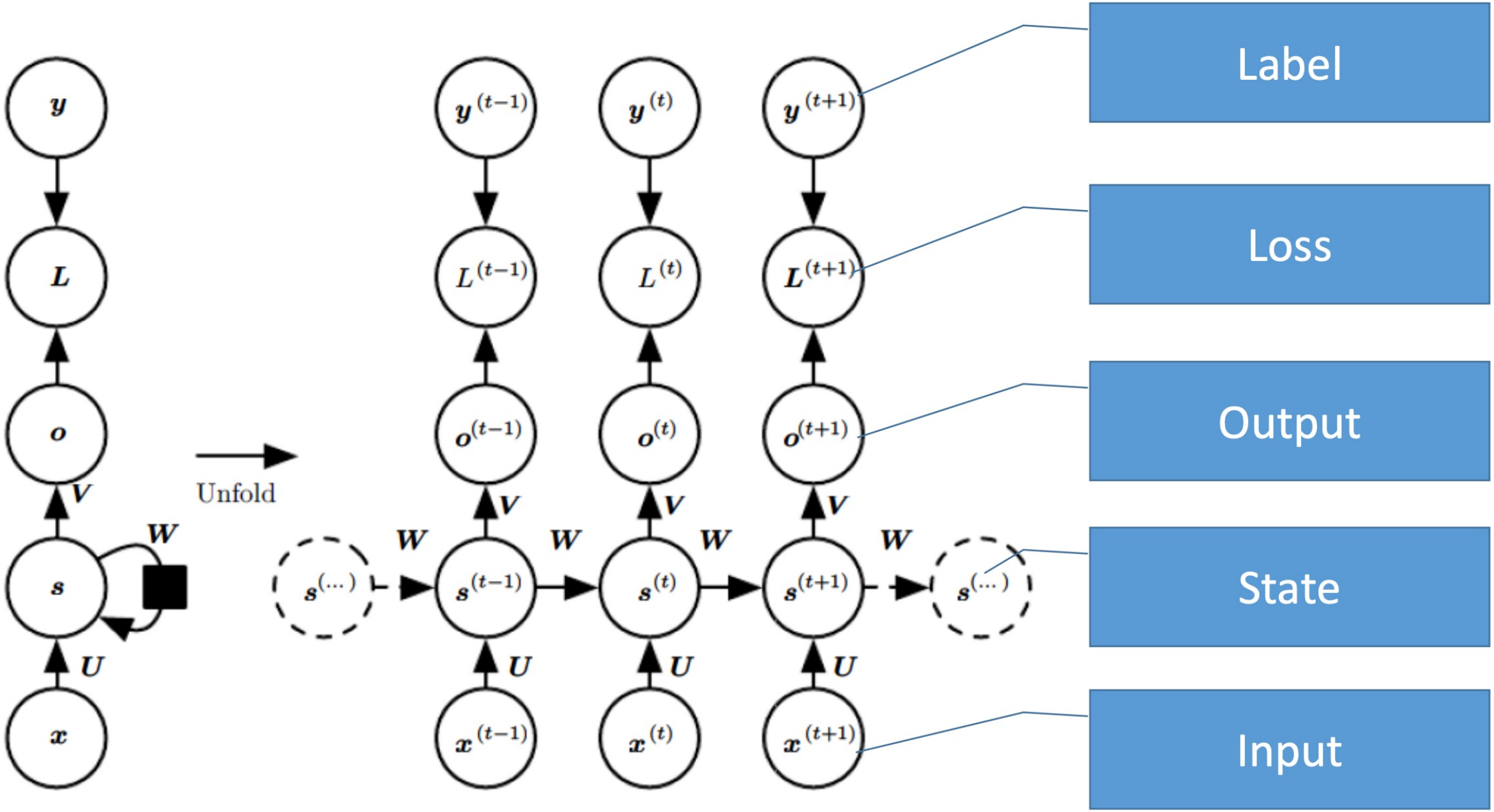
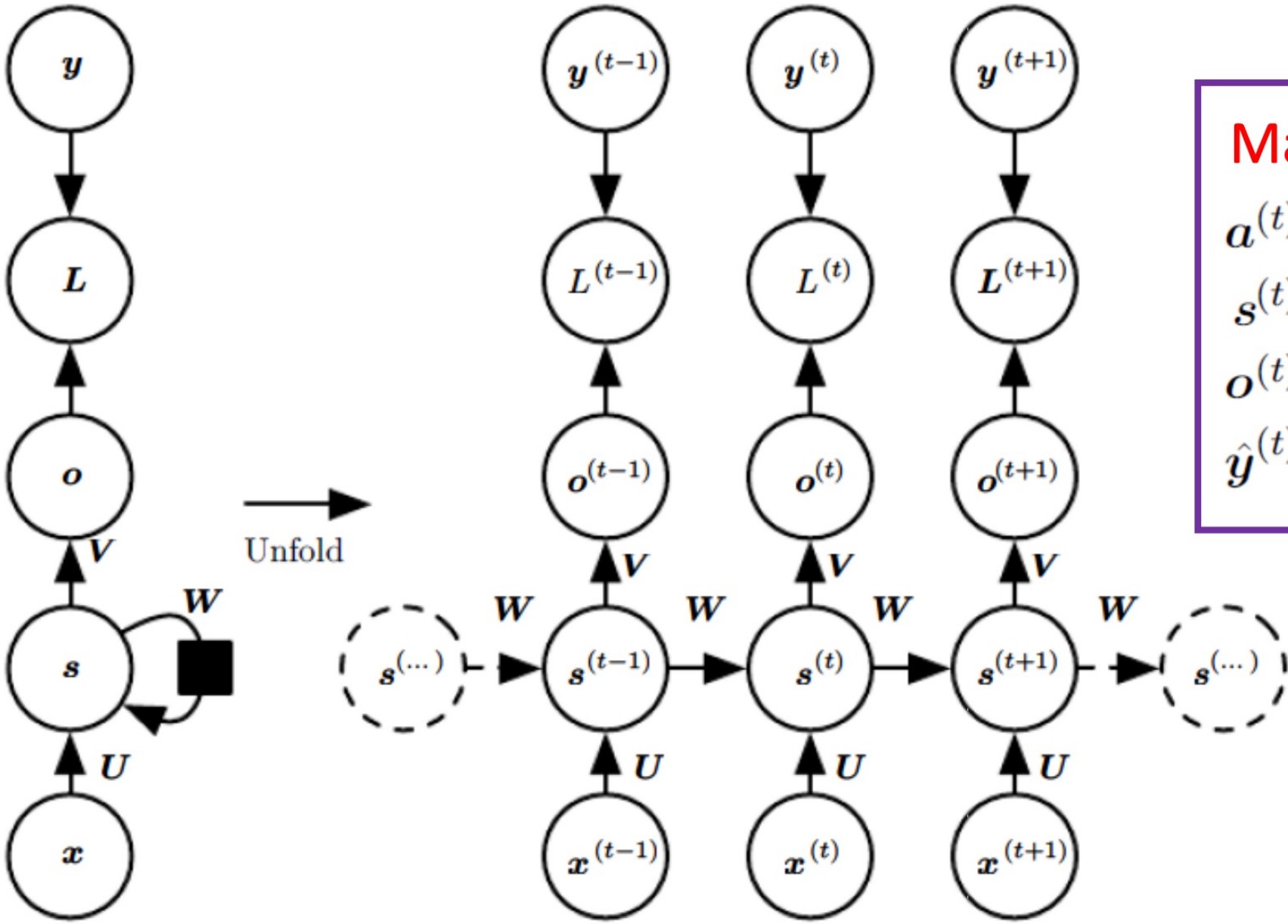


Figure from *Deep Learning*, by Goodfellow, Bengio and Courville



Math formula:

$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{s}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{s}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{s}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \mathrm{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}
$$

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

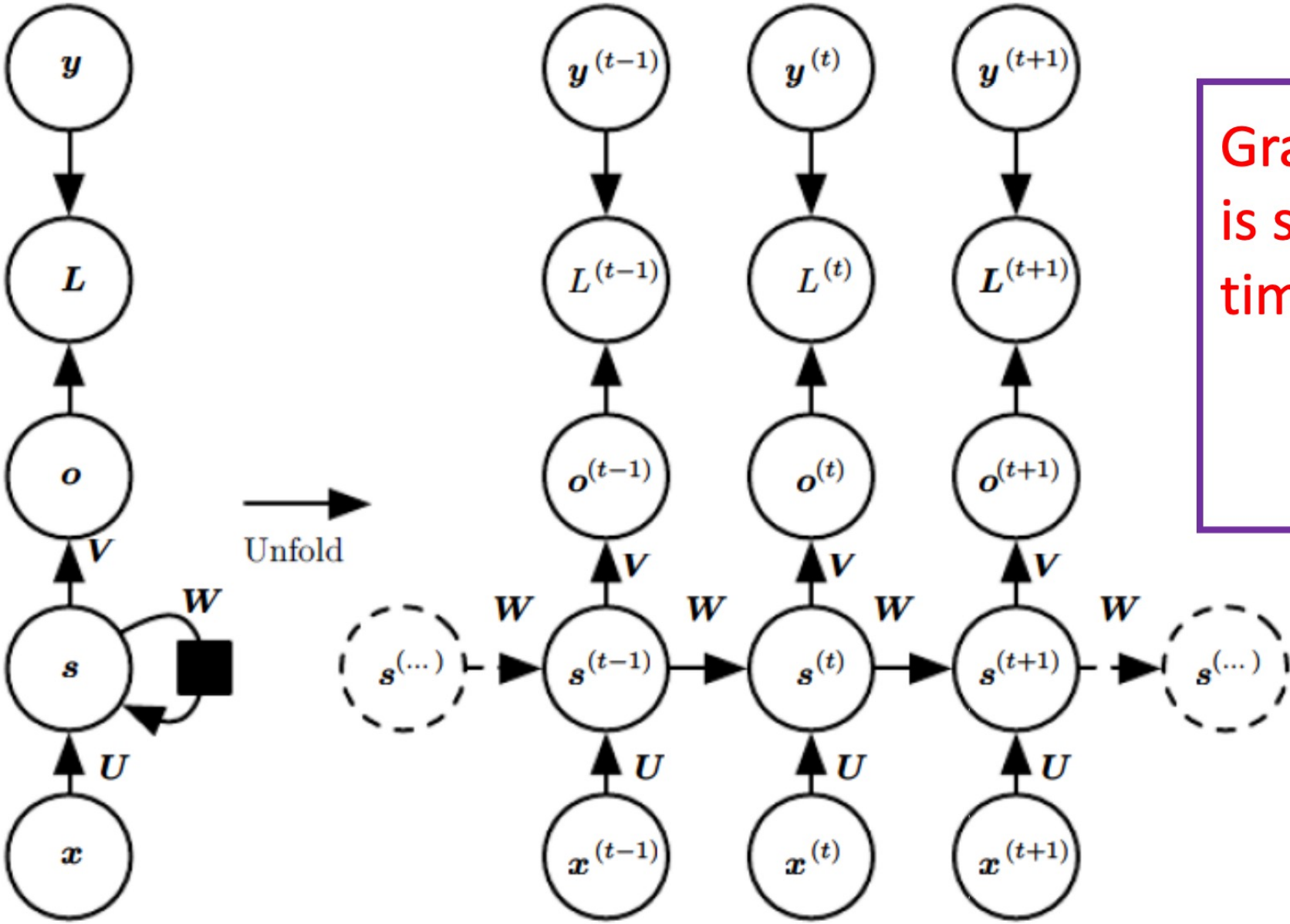
Advantage

- Hidden state: a lossy summary of the past
- Shared functions and parameters: greatly reduce the capacity and good for generalization in learning
- Explicitly use the prior knowledge that the sequential data can be processed by in the same way at different time step (e.g., NLP)

Training RNN

- Principle: unfold the computational graph, and use backpropagation
- Called back-propagation through time (BPTT) algorithm
- Can then apply any general-purpose gradient-based techniques

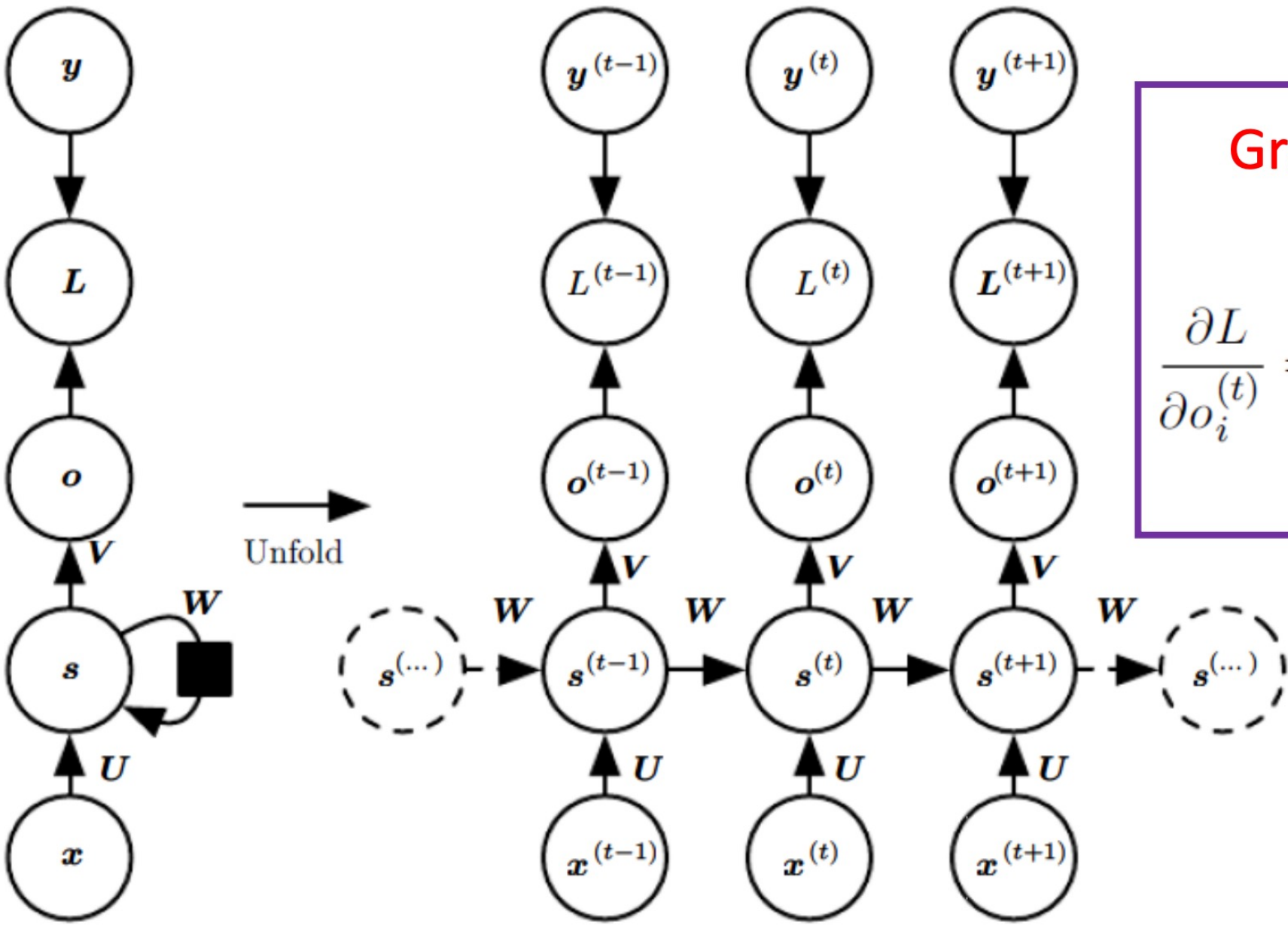
- Conceptually: first compute the gradients of the internal nodes, then compute the gradients of the parameters



Gradient at $L^{(t)}$: (total loss is sum of those at different time steps)

$$\frac{\partial L}{\partial L^{(t)}} = 1.$$

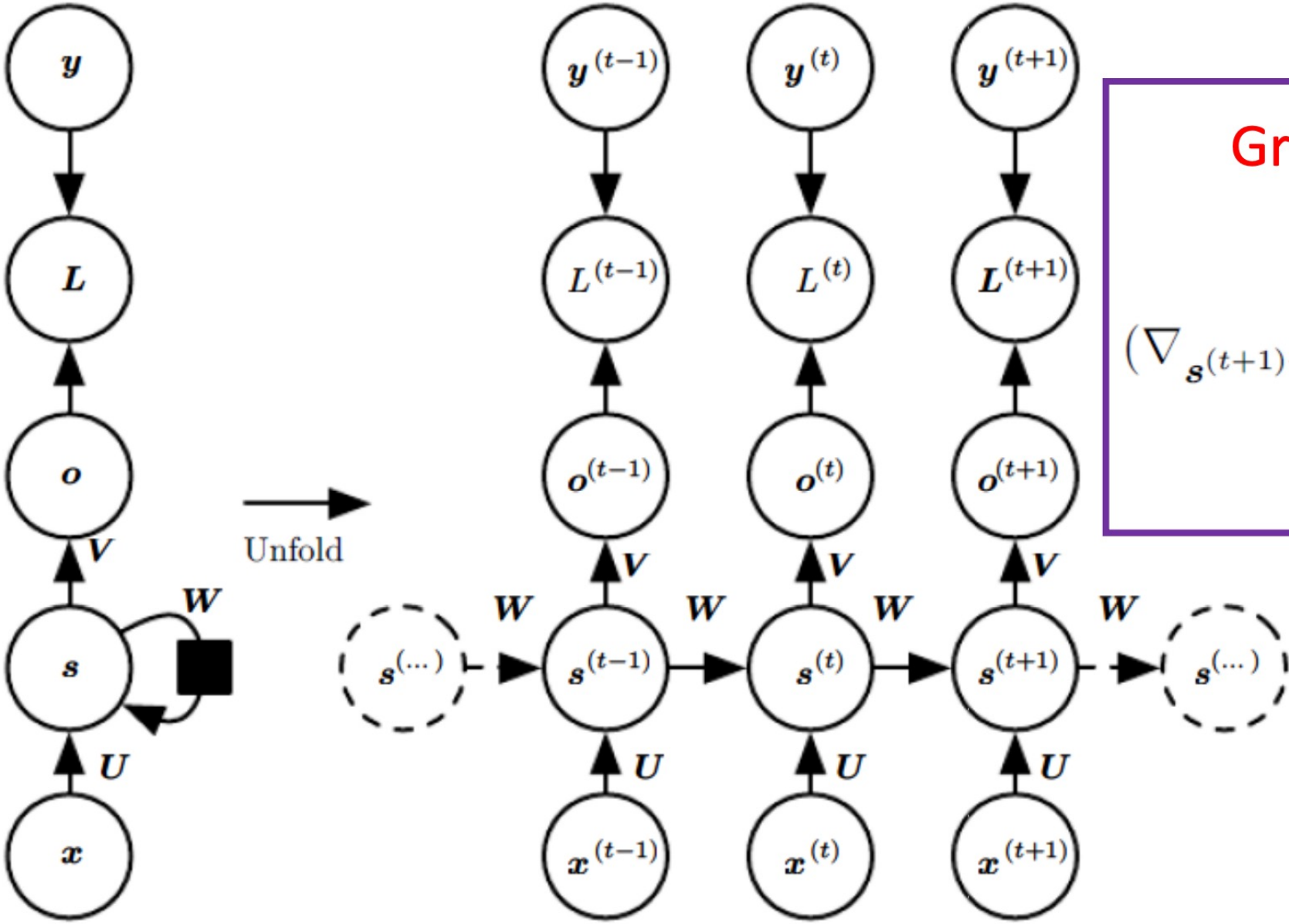
Figure from *Deep Learning*, Goodfellow, Bengio and Courville



Gradient at $o^{(t)}$:

$$\frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - \mathbf{1}_{i,y^{(t)}}$$

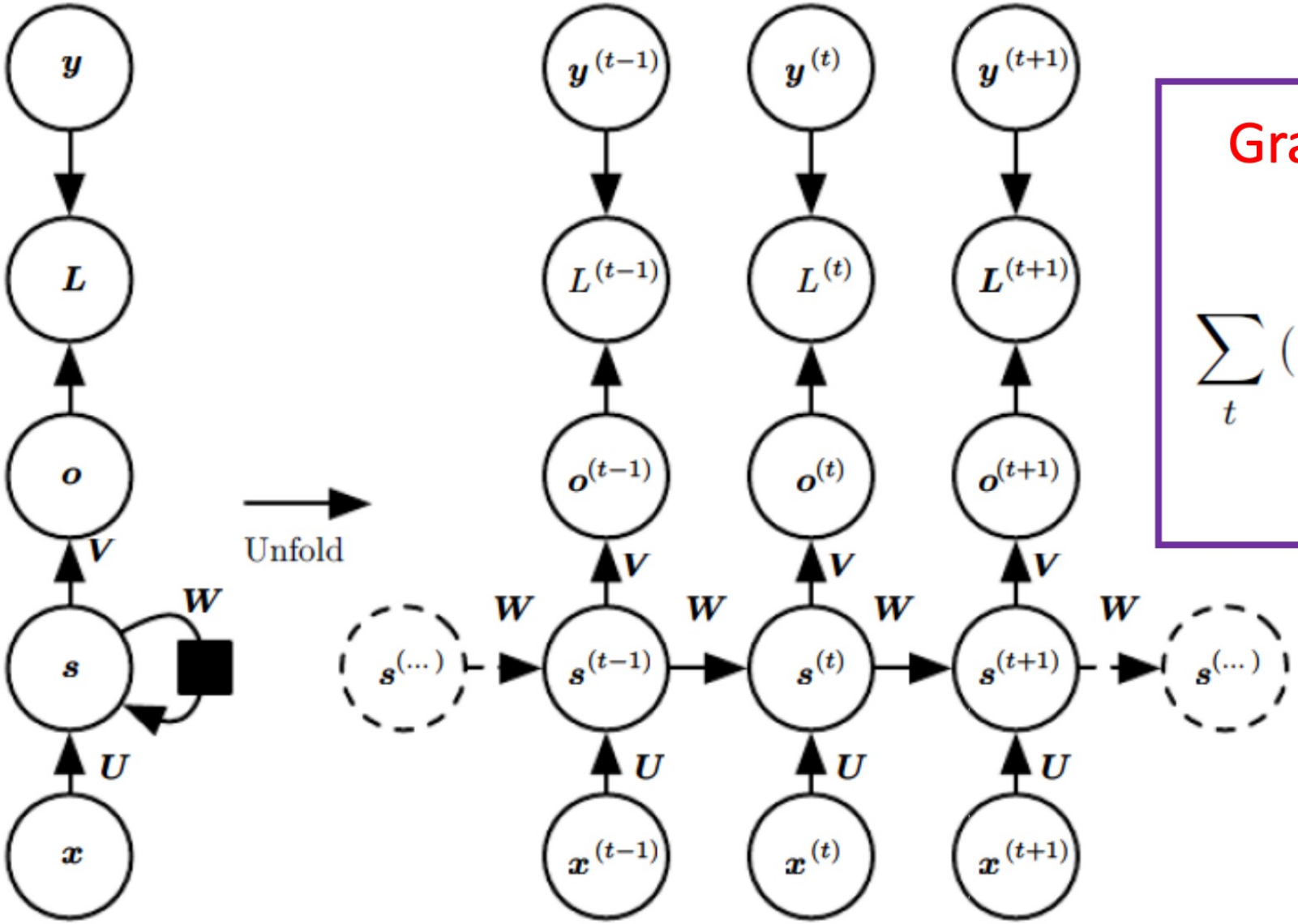
Figure from *Deep Learning*, Goodfellow, Bengio and Courville



Gradient at $s^{(t)}$:

$$(\nabla_{\boldsymbol{s}^{(t+1)}} L)\,\frac{\partial \boldsymbol{s}^{(t+1)}}{\partial \boldsymbol{s}^{(t)}} + (\nabla_{\boldsymbol{o}^{(t)}} L)\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{s}^{(t)}}$$

Figure from *Deep Learning*, Goodfellow, Bengio and Courville



Gradient at parameter V:

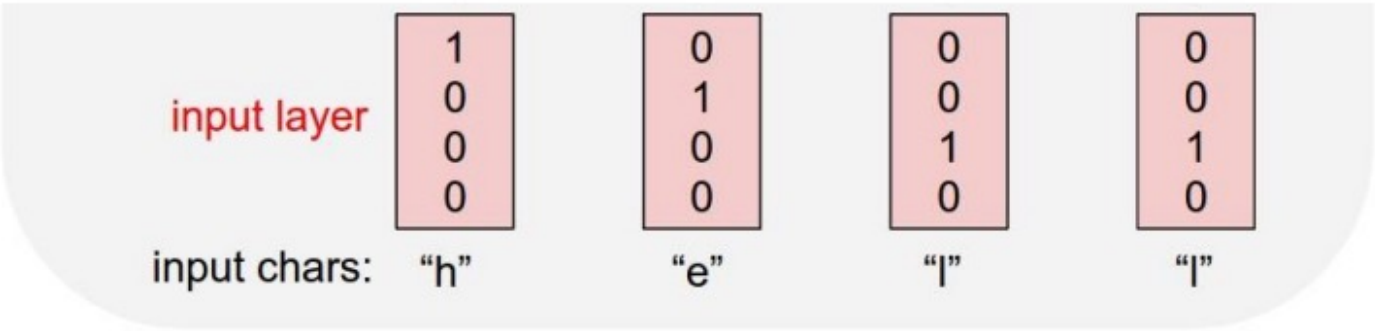
$$\sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{V}} = \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \boldsymbol{s}^{(t)\top}$$

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Example:
Character-level
Language Model

Vocabulary:
[h,e,l,o]

Example training sequence:
“hello”

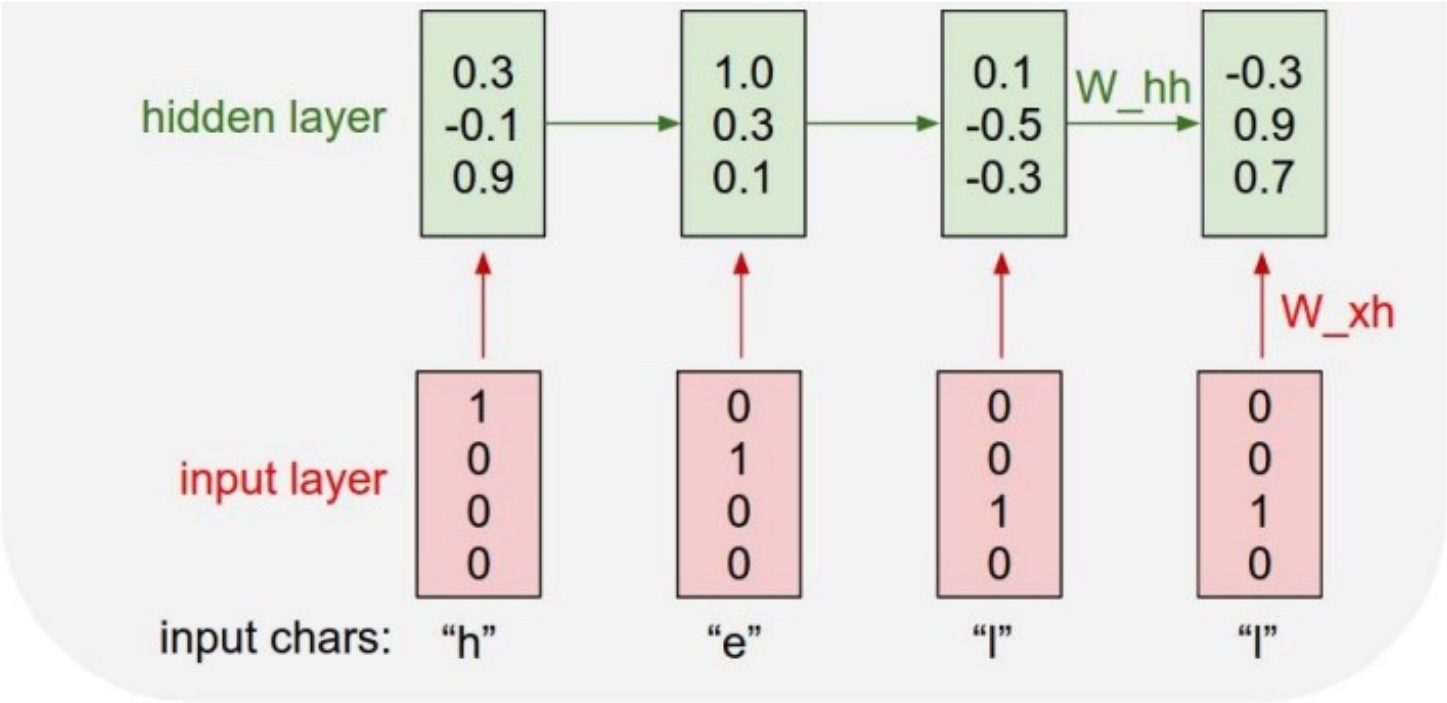


Example: Character-level Language Model

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

Vocabulary:
[h,e,l,o]

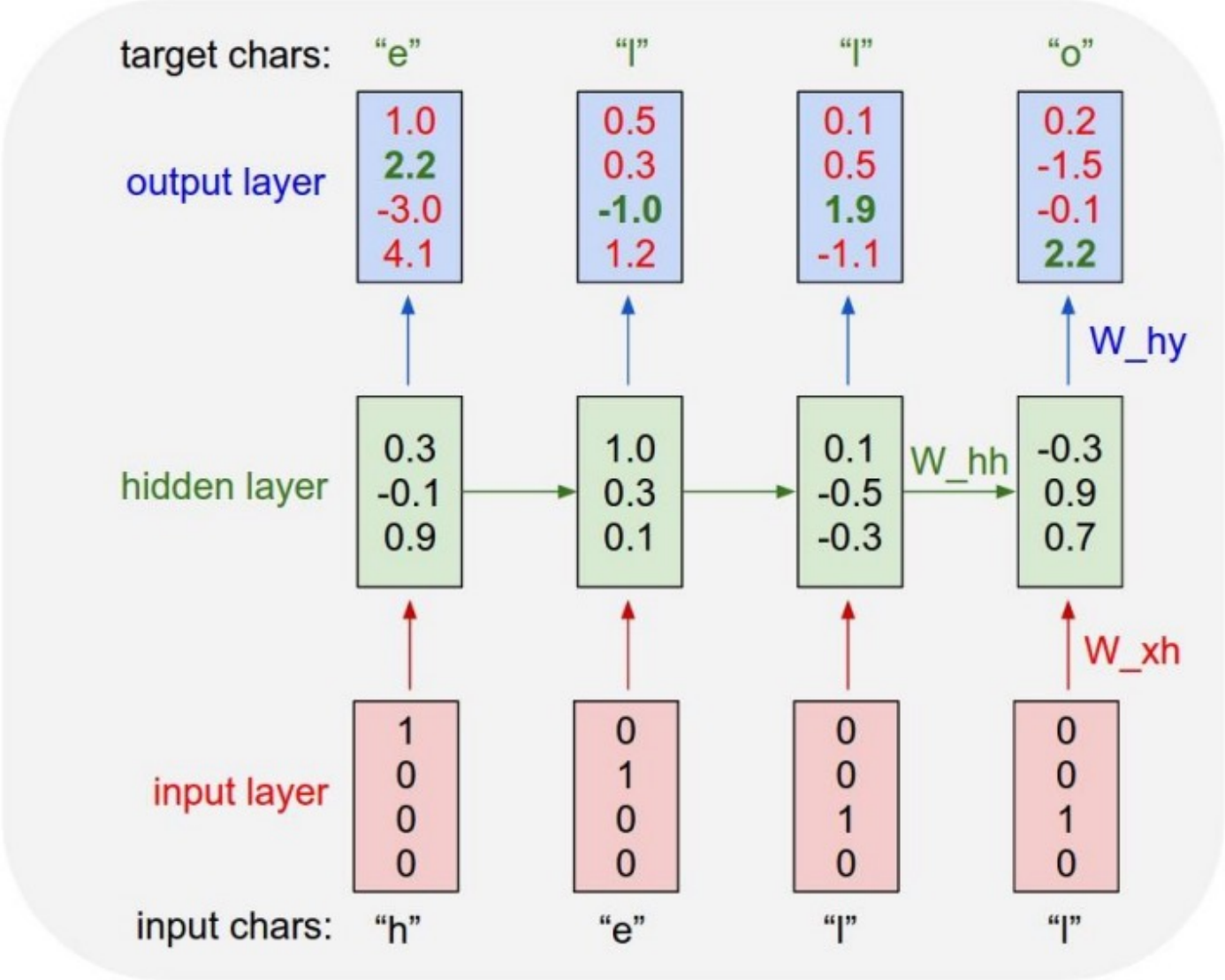
Example training sequence:
"hello"



Example: Character-level Language Model

Vocabulary:
[h,e,l,o]

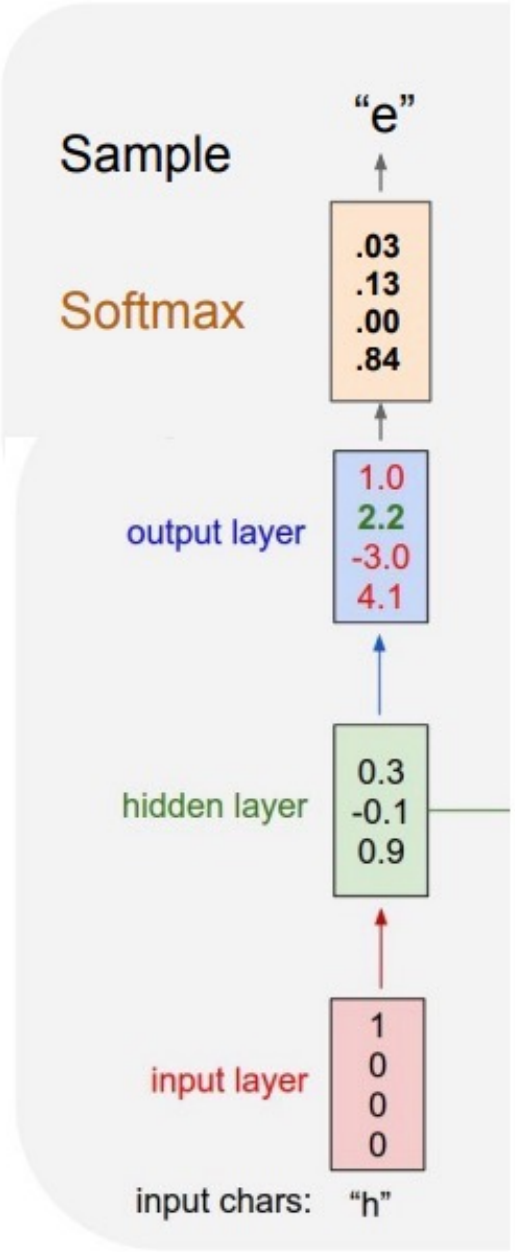
Example training sequence:
“hello”



Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

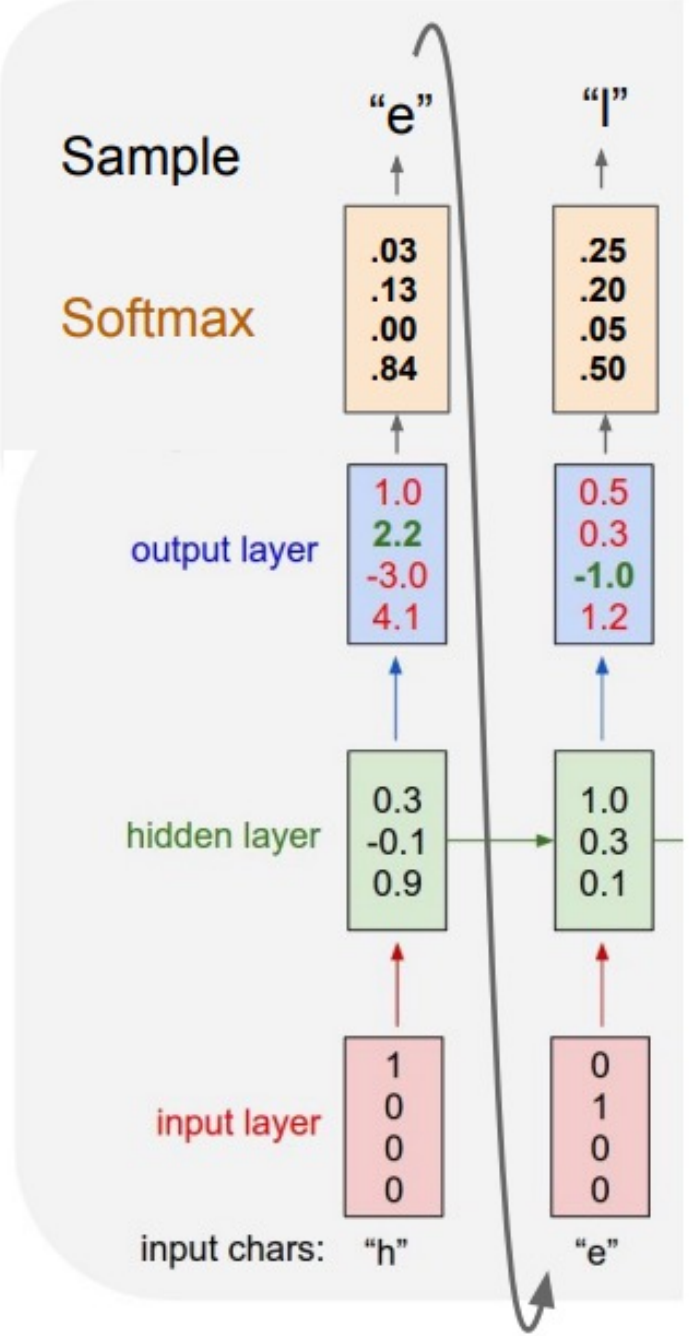
At test-time sample characters one at a time, feed back to model



Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

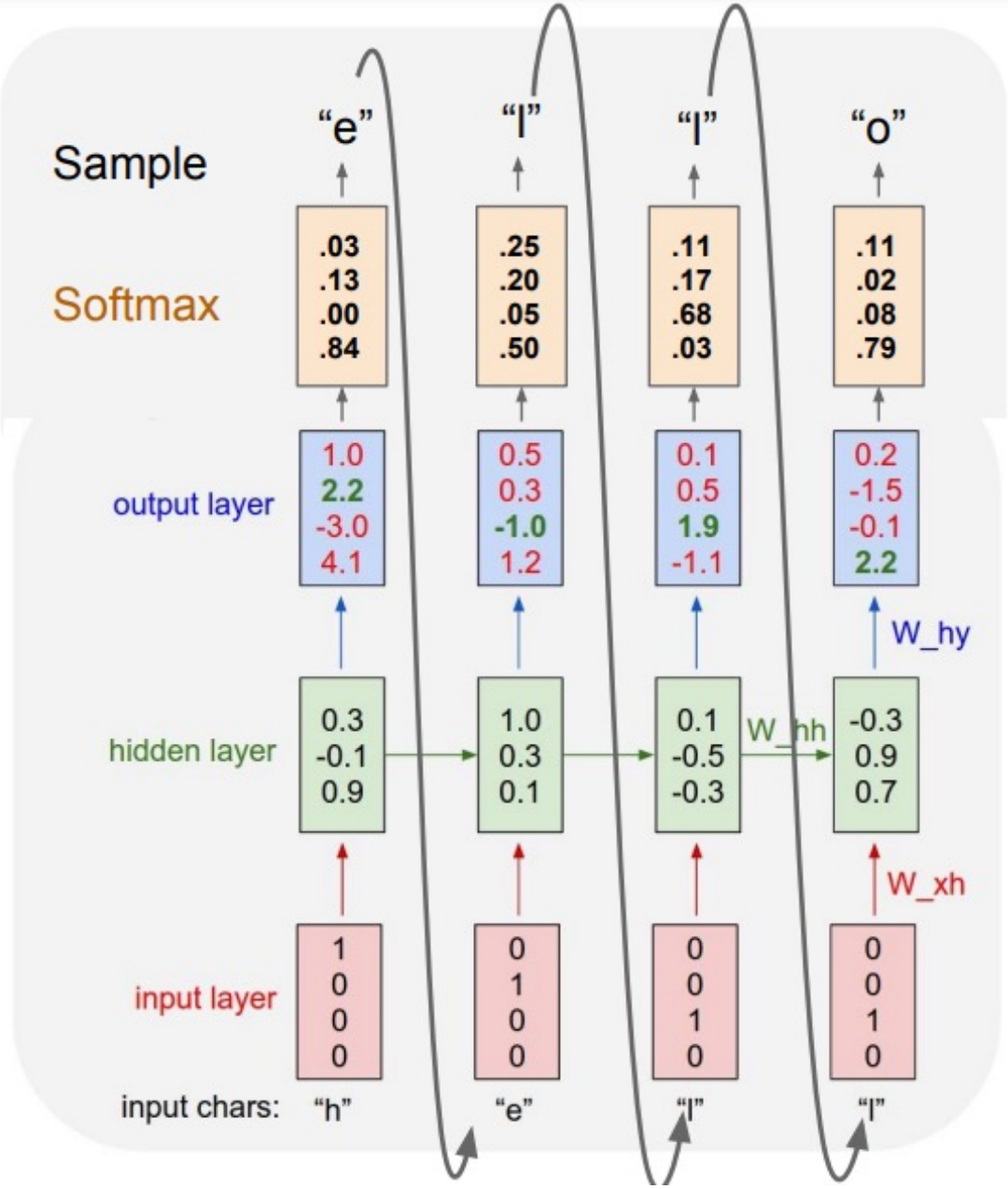
At test-time sample characters one at a time, feed back to model



Example: Character-level Language Model Sampling

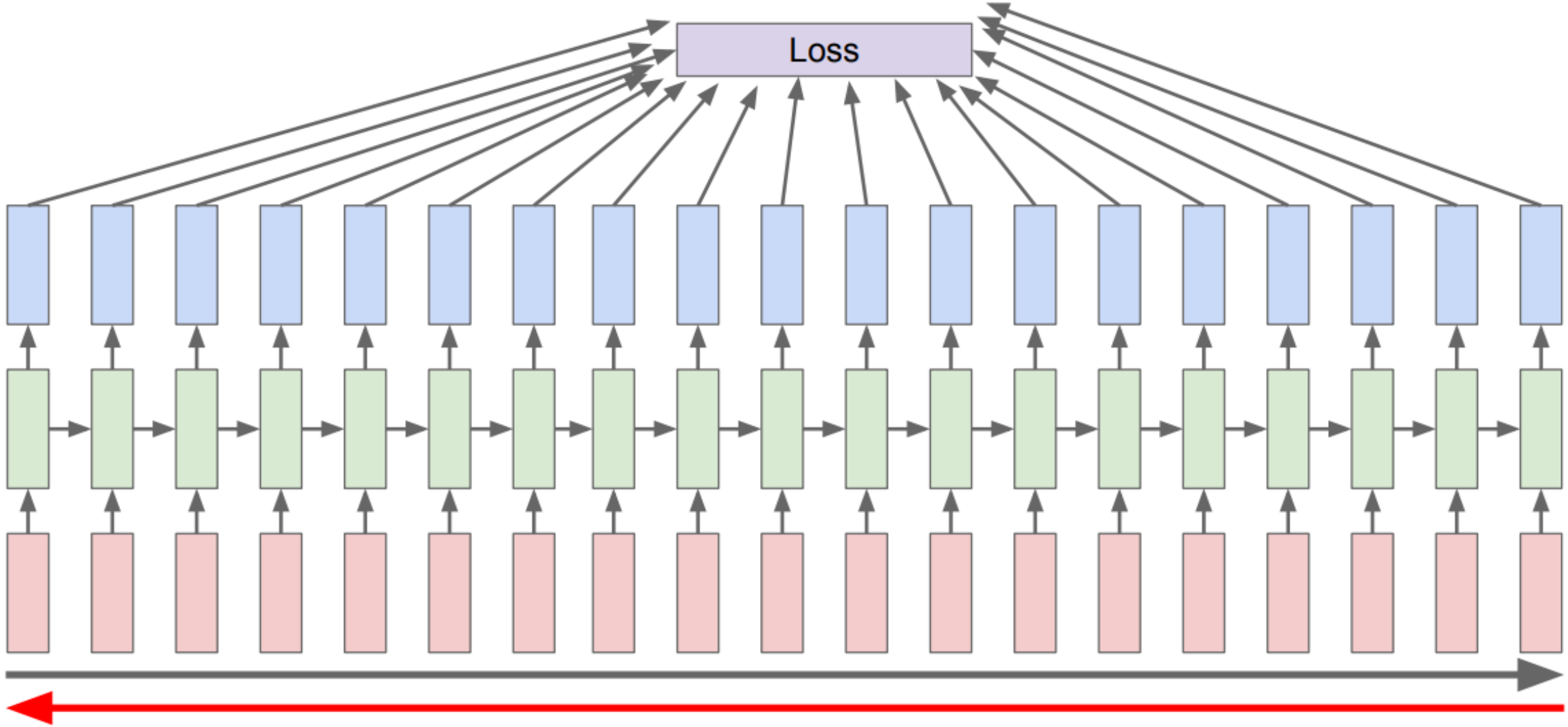
Vocabulary:
[h,e,l,o]

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

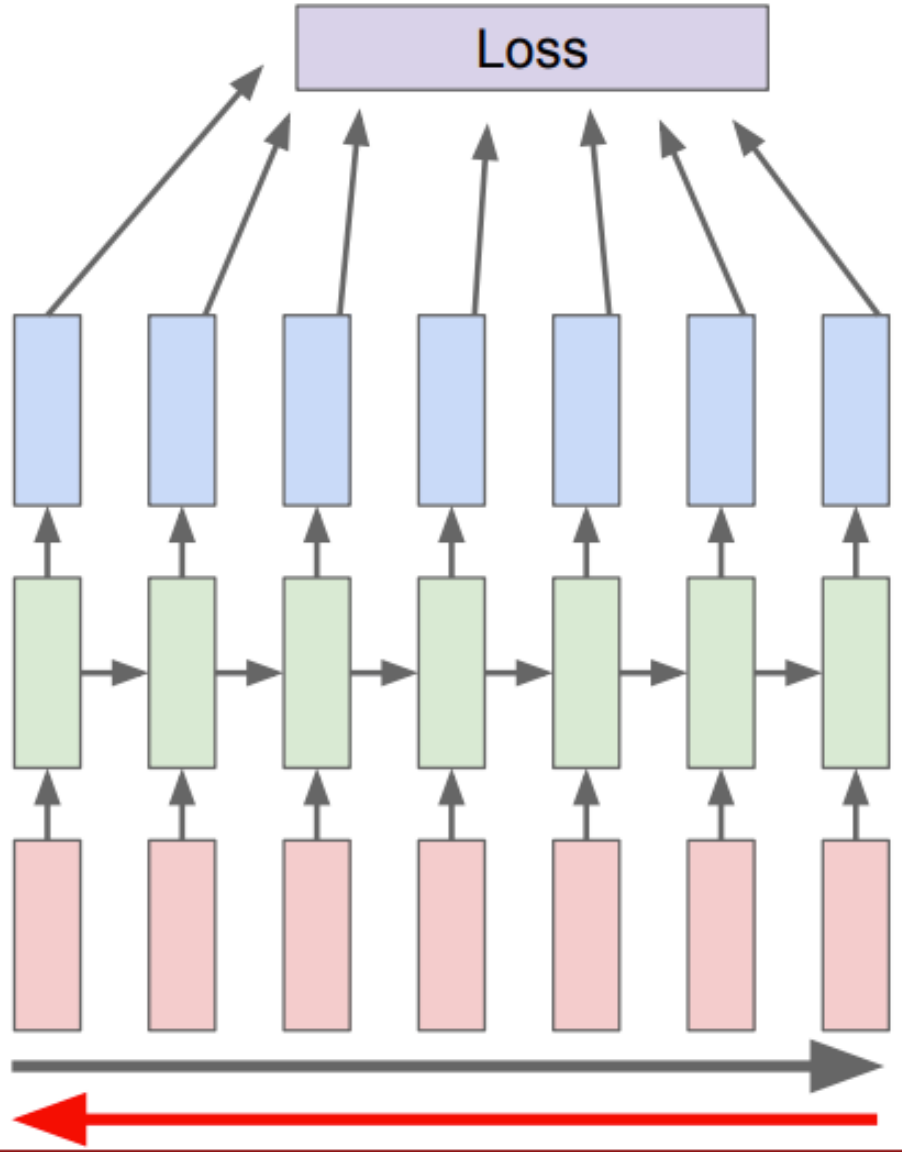


Backpropagation through time

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

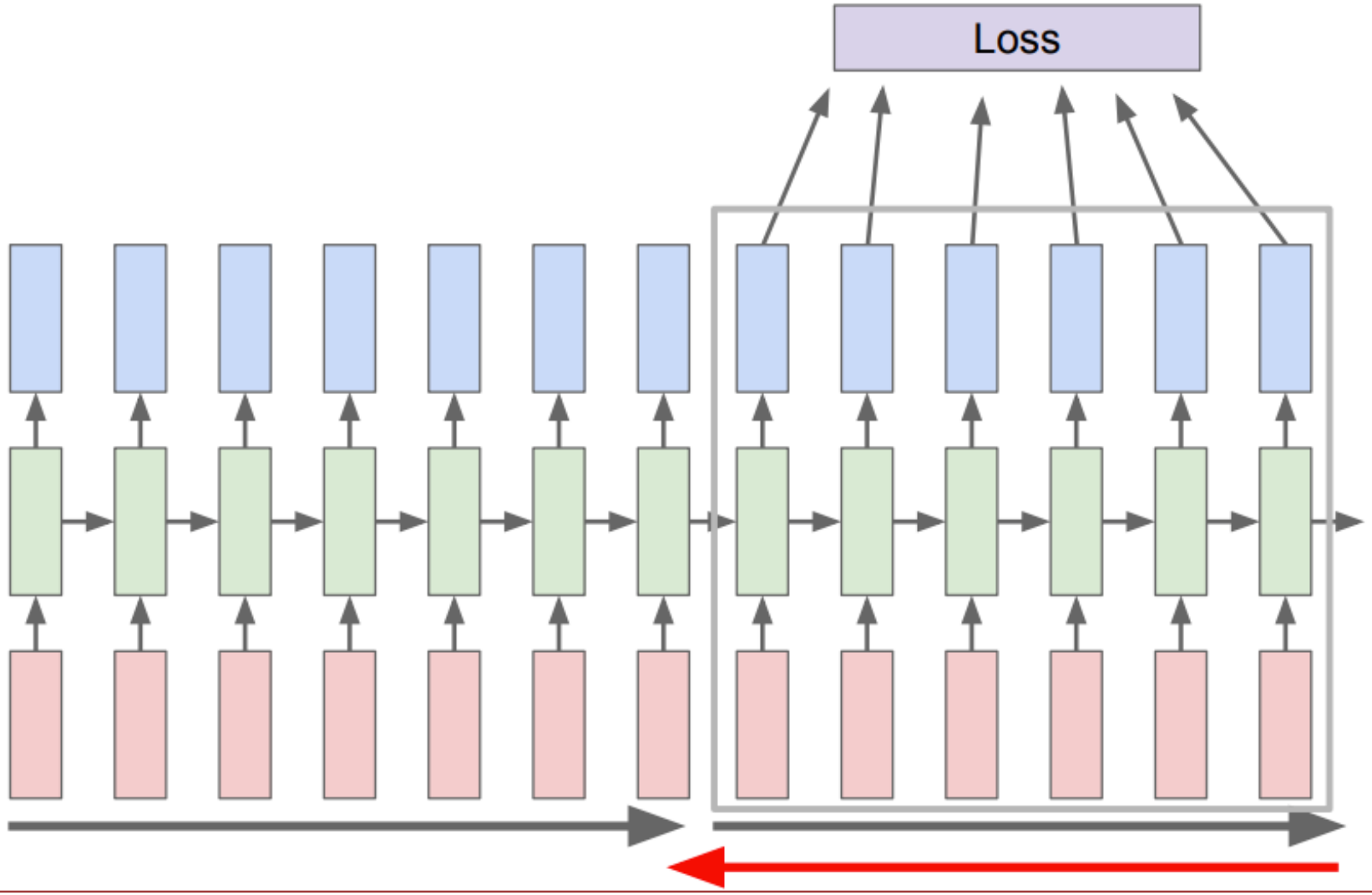


Truncated Backpropagation through time



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