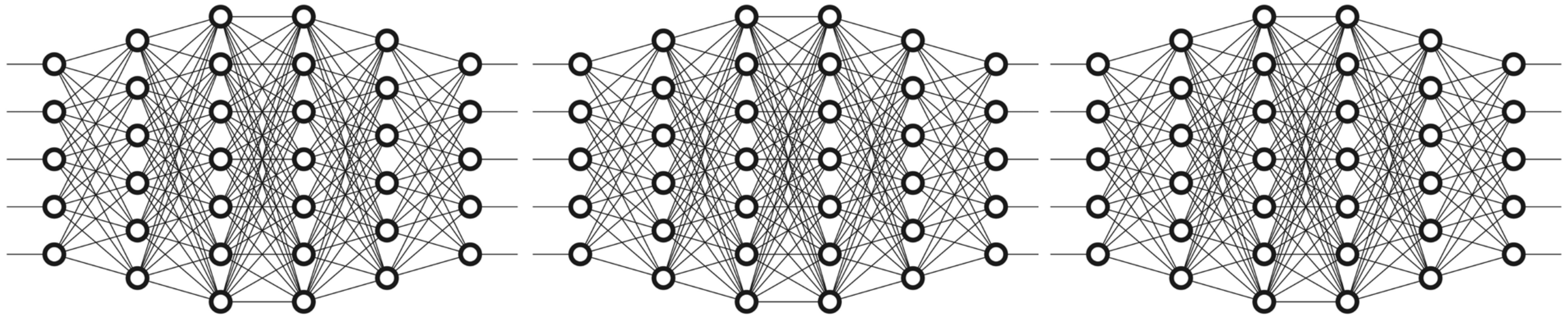
CPSC 425: Computer Vision



Lecture 21: Neural Networks

Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-4 is a great starting point for many models!

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

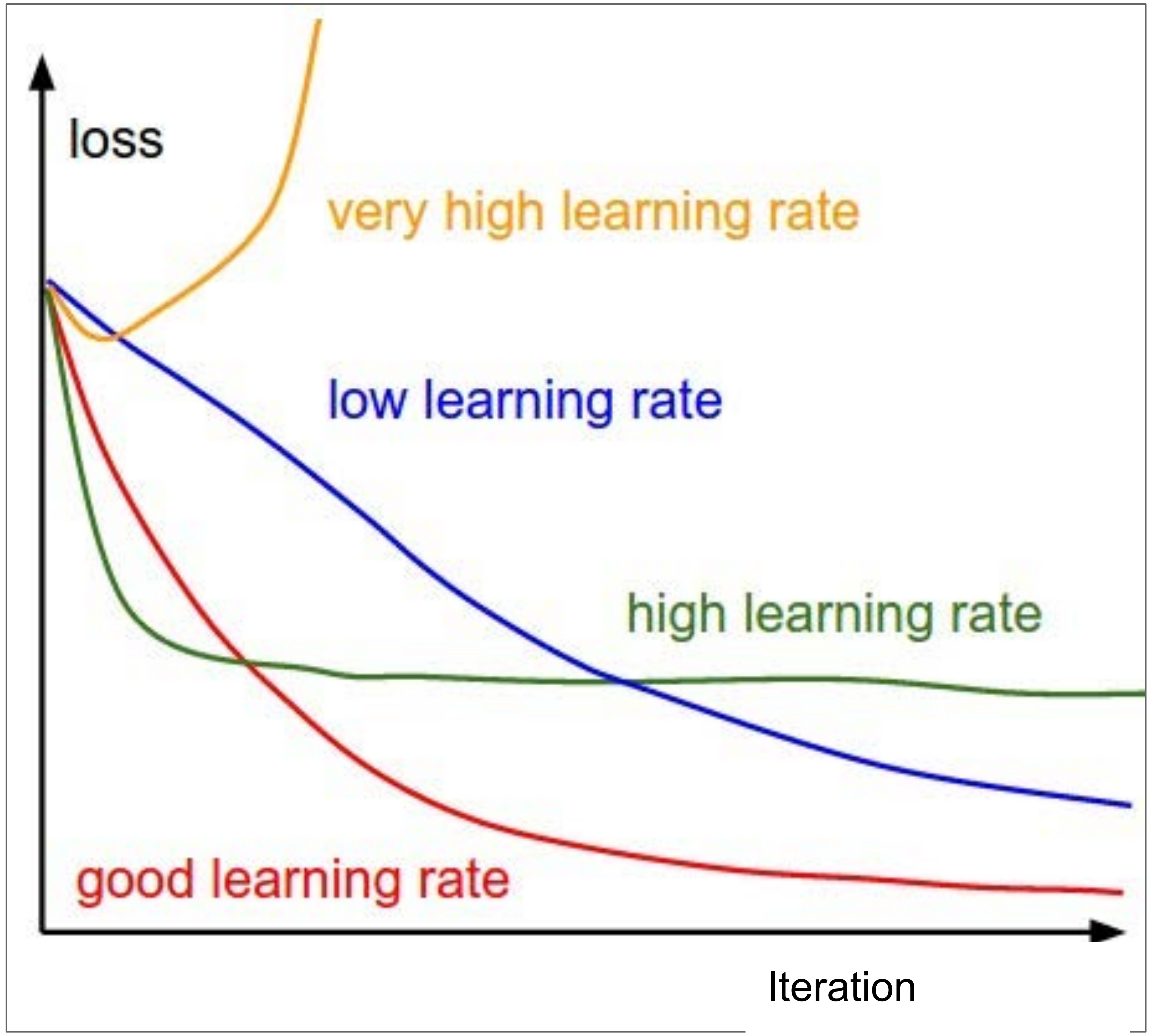
Based on slides for Stanford cs231n by Li, Jonson, and Young. Modified and reused with permission

Adam

- SGD
- SGD+Momentum
- RMSProp
- Adam

Learning rate: hyperparameter

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter



Linear + Softmax Regression

- We found the following gradient descent update rule

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \alpha(\mathbf{h} - \mathbf{t})\mathbf{x}^T$$

prediction ($\mathbf{h}$) targets ($\mathbf{t}$) data ($\mathbf{x}$)

- This applies to:

Linear regression	$\mathbf{h} = \mathbf{W}^T\mathbf{x}$	L2 loss
Softmax regression	$\mathbf{h} = \sigma(\mathbf{W}^T\mathbf{x})$	cross-entropy loss

- The same update rule with a binary prediction function

$$\mathbf{h} = \mathbb{1}_{\max}(\mathbf{W}^T\mathbf{x})$$

implements the multiclass Perceptron learning rule

2-class Perceptron Classifier

- Classification function is

$$\hat{y} = \text{sign}(\mathbf{w}^T \mathbf{x})$$

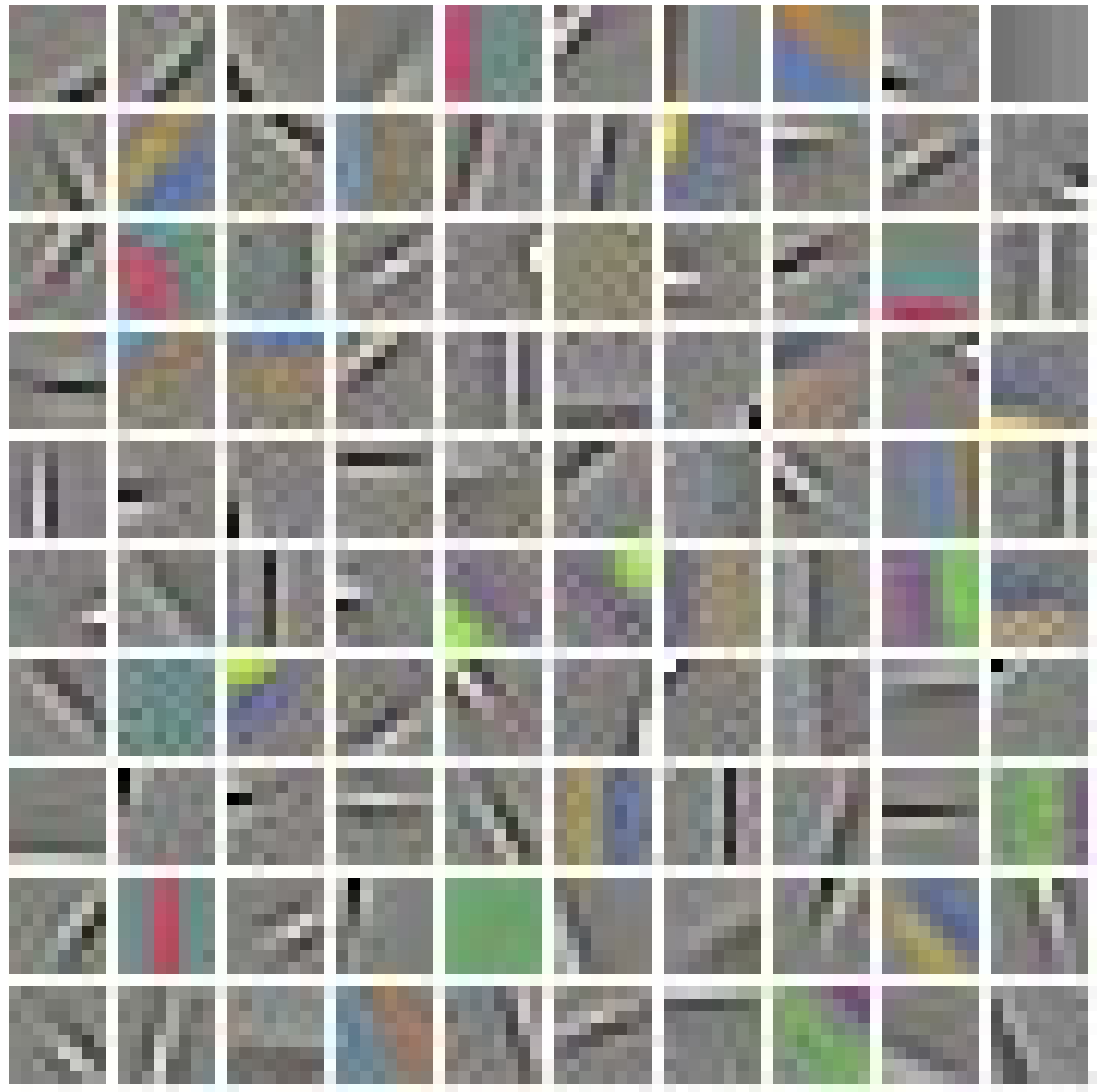
- Linear function of the data (x) followed by 0/1 activation

- Update rule: present data x
  - if correctly classified, do nothing
  - if incorrectly classified, update the weight vector

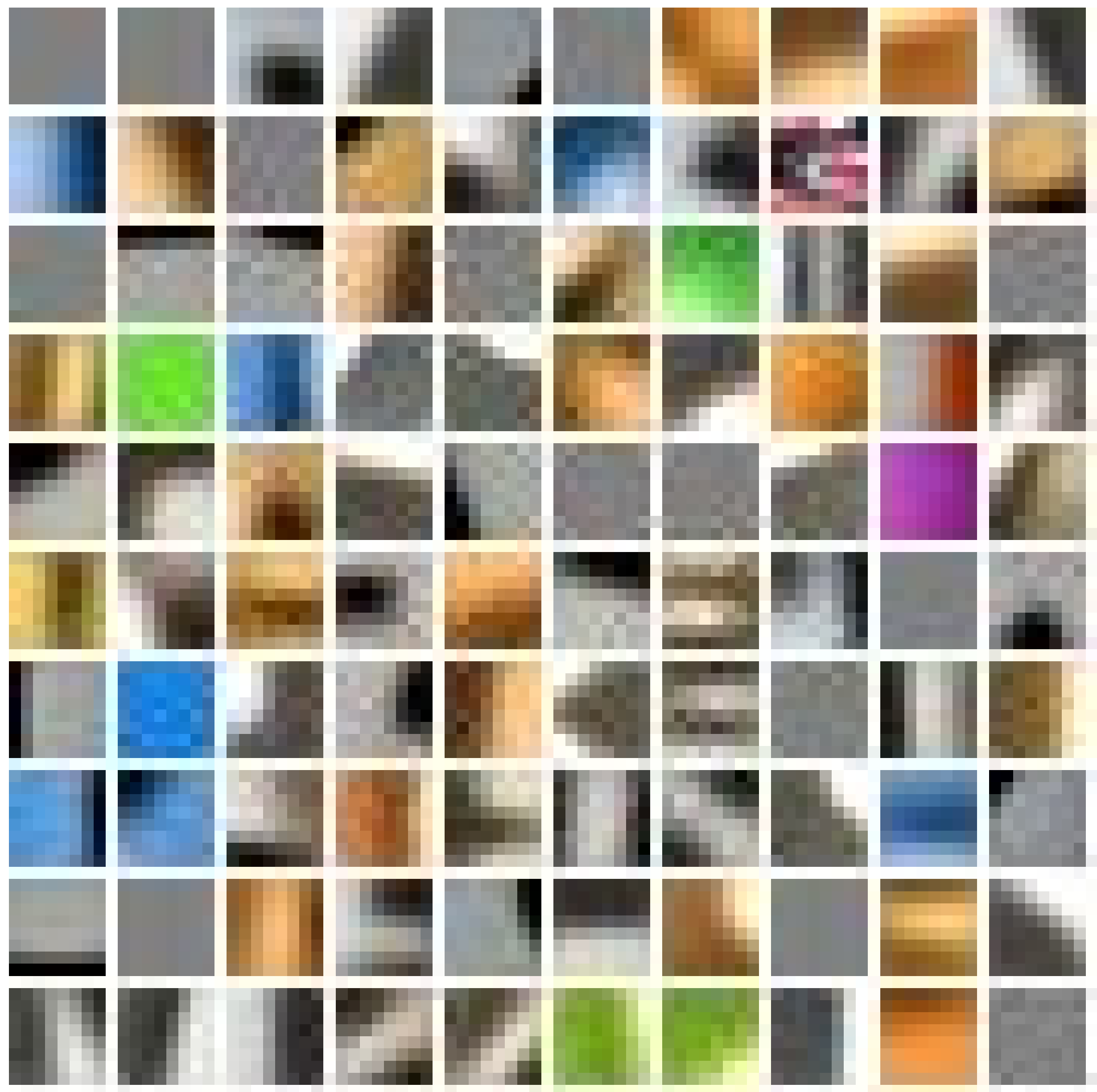
$$\mathbf{w}_{n+1} = \mathbf{w}_n + y_i \mathbf{x}_i$$

CIFAR10 Feature Extraction

- So far, we used RGB pixels as the input to our classifier
- Feature extraction can improve results by a lot
- e.g., Coates et al. achieve 79.6% accuracy on CIFAR10 with a features based on k-means of whitened image patches



k-means, whitened

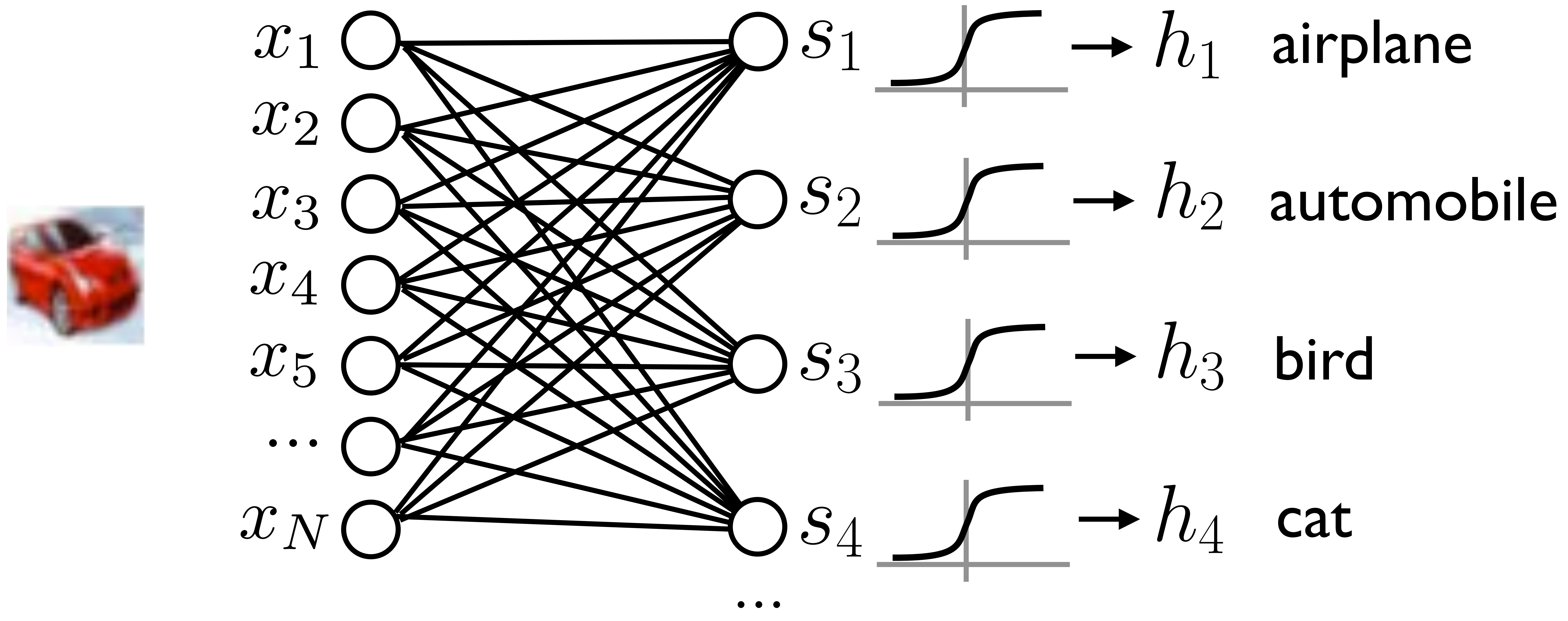


k-means, raw RGB

[Coates et al. 2011]

Linear = Fully Connected Layer

- Note that our linear matrix multiplication classifier is equivalent to a fully connected layer in a neural network

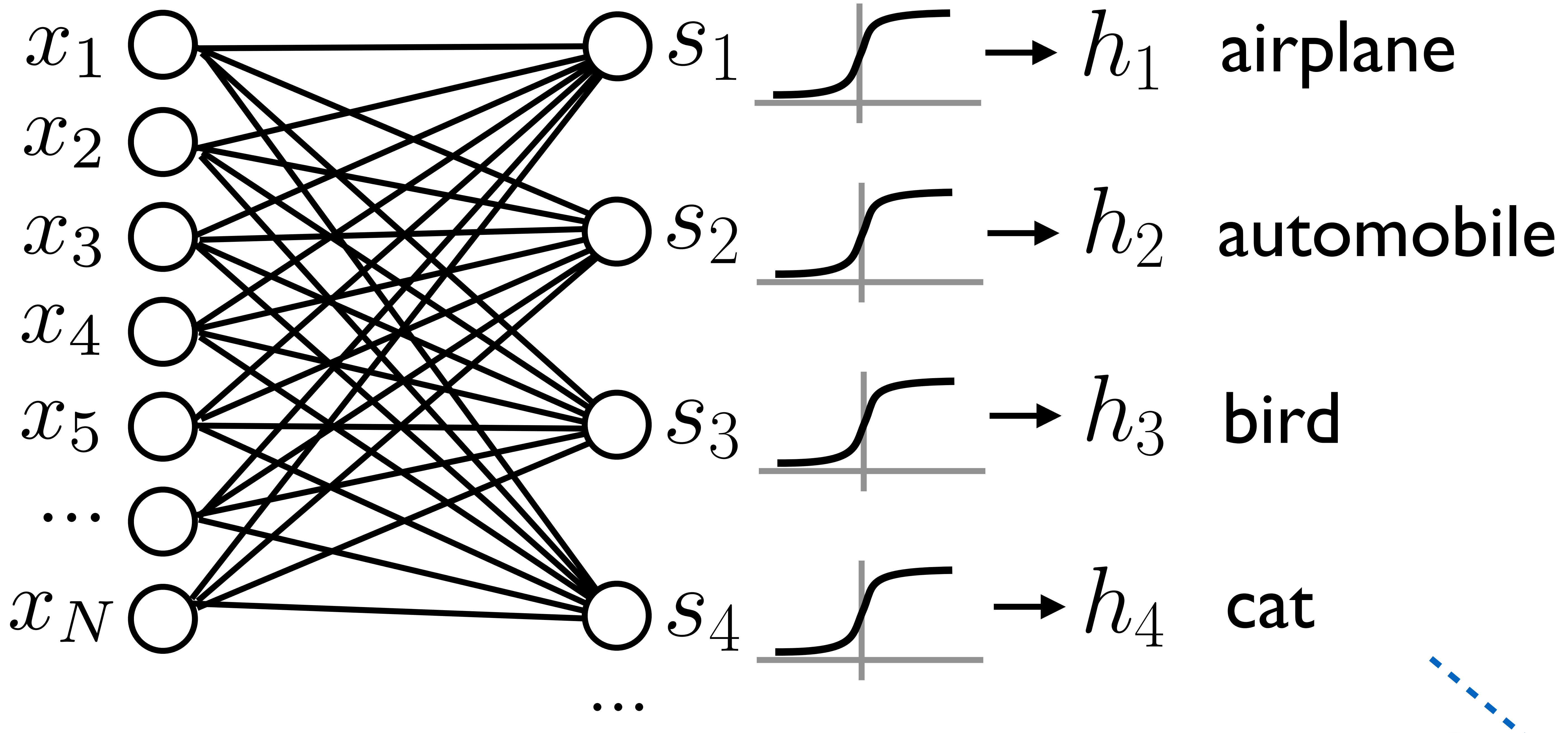


- Typically, we'll also add a bias term b

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

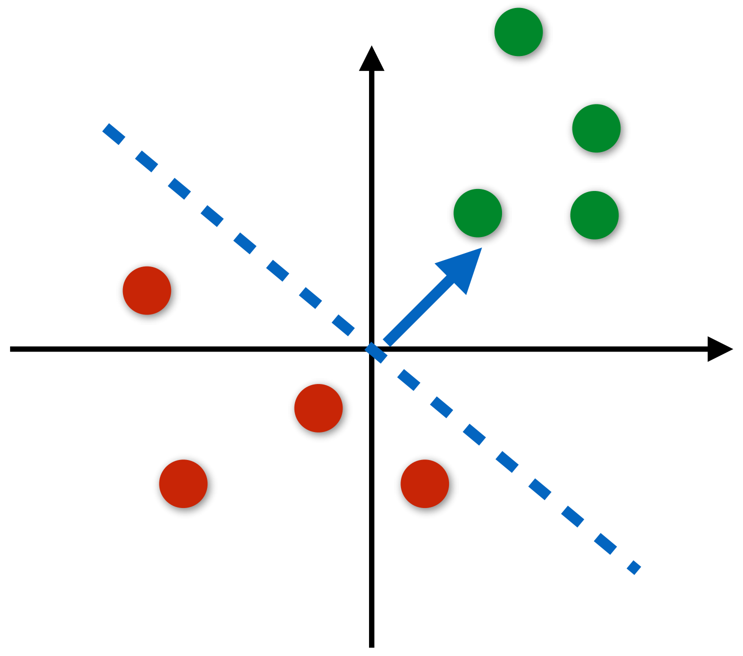
Linear = Fully Connected Layer

- Note that our linear matrix multiplication classifier is equivalent to a fully connected layer in a neural network



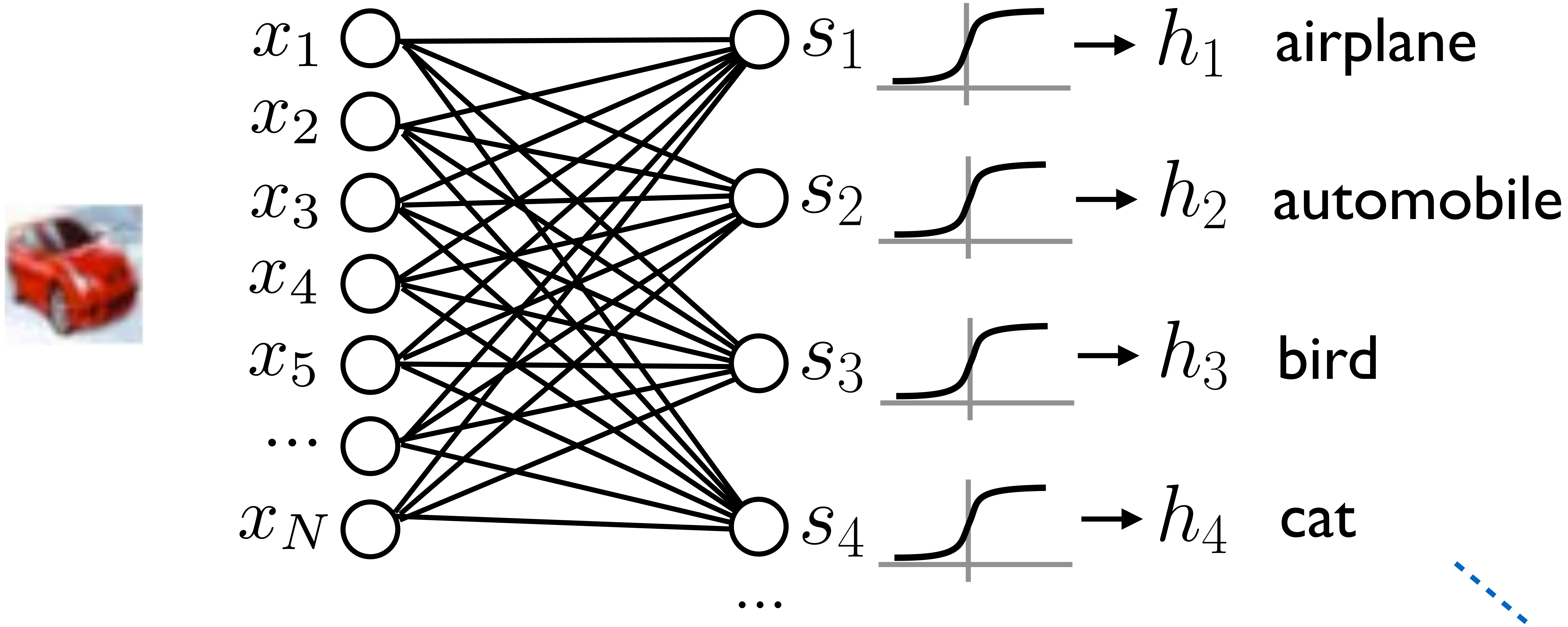
- Typically, we'll also add a bias term b

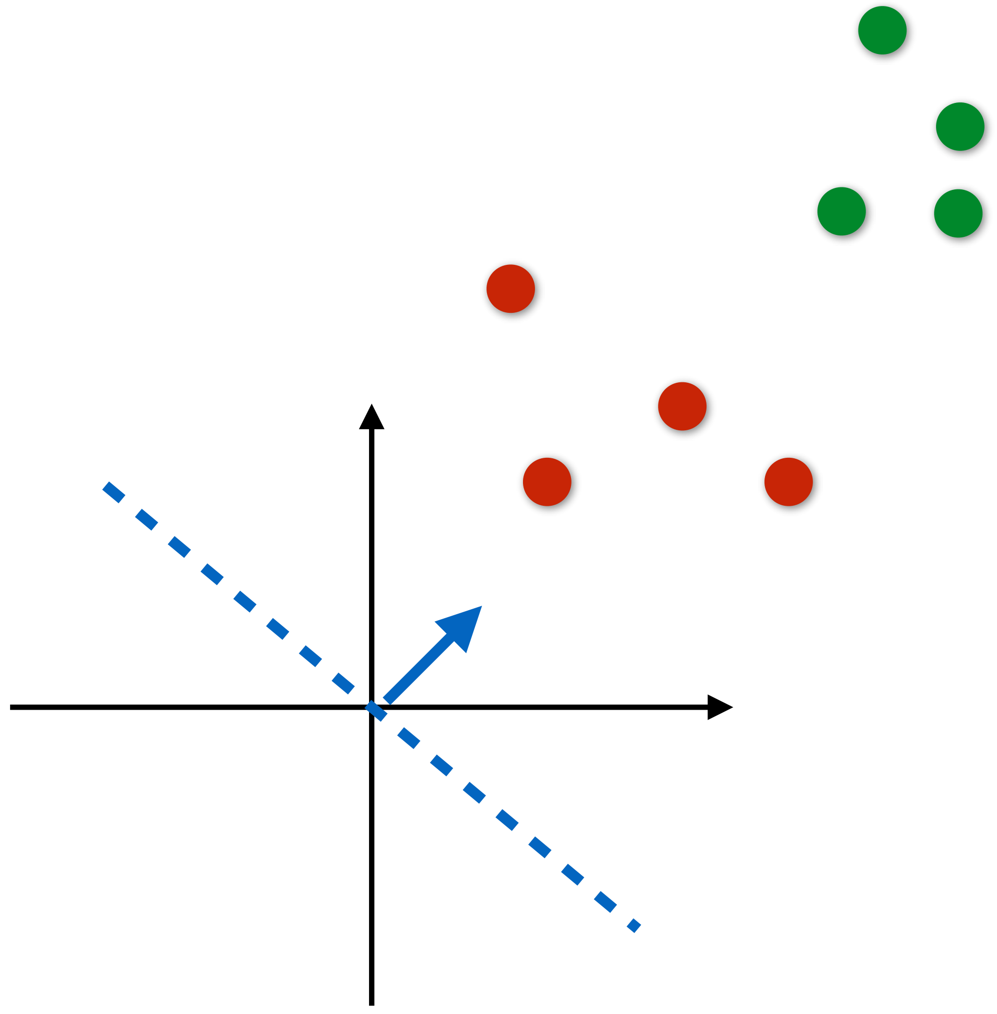
$$\mathbf{h} = \sigma(\mathbf{W}^T\mathbf{x} + \mathbf{b})$$



Linear = Fully Connected Layer

- Note that our linear matrix multiplication classifier is equivalent to a fully connected layer in a neural network



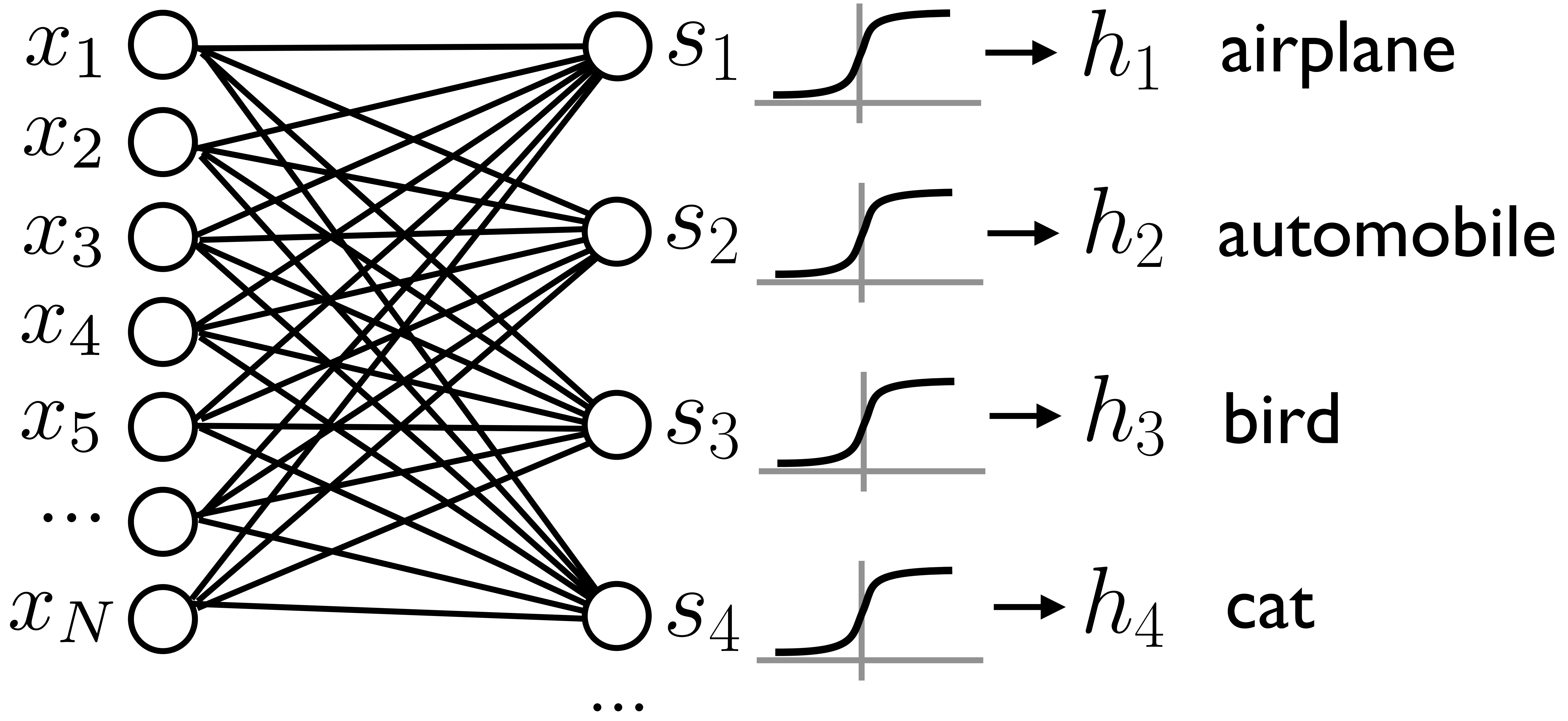


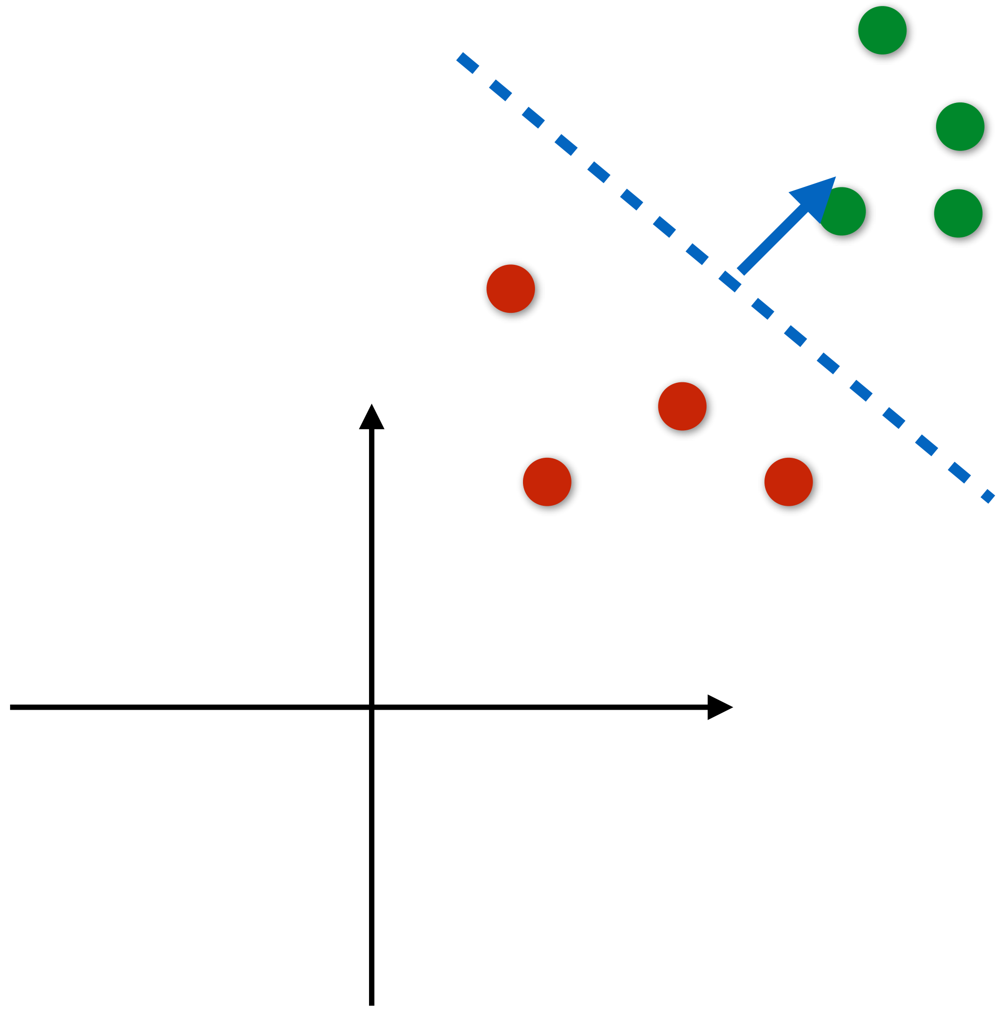
- Typically, we'll also add a bias term b

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

Linear = Fully Connected Layer

- Note that our linear matrix multiplication classifier is equivalent to a fully connected layer in a neural network



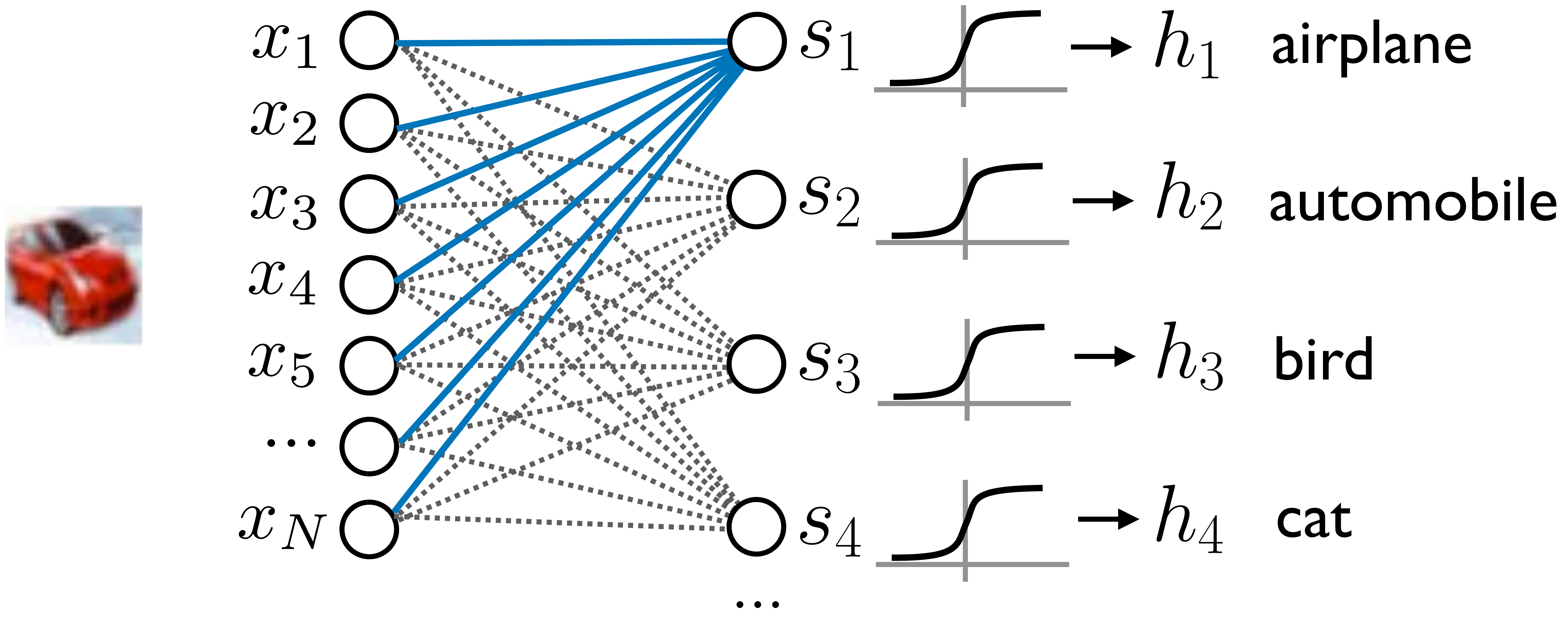


- Typically, we'll also add a bias term b

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

Linear = Fully Connected Layer

- Note that our linear matrix multiplication classifier is equivalent to a fully connected layer in a neural network

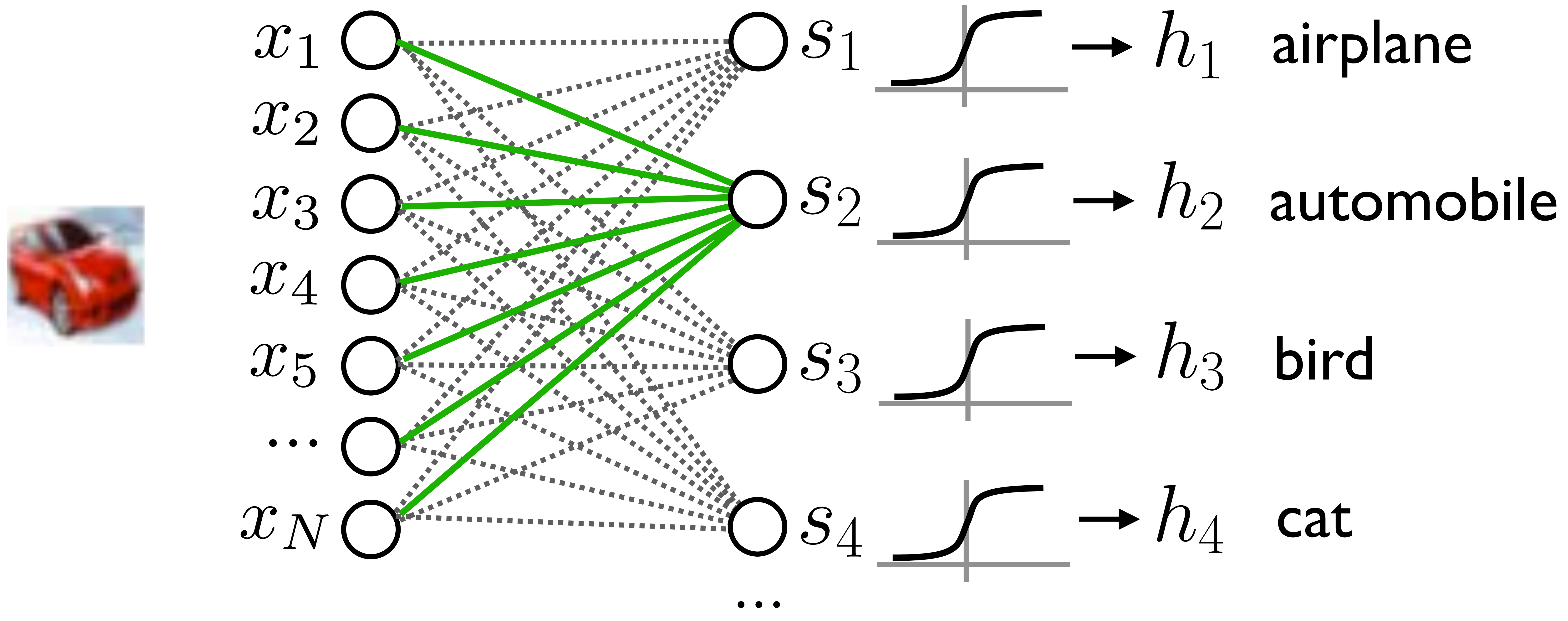


- Typically, we'll also add a bias term b

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

Linear = Fully Connected Layer

- Note that our linear matrix multiplication classifier is equivalent to a fully connected layer in a neural network

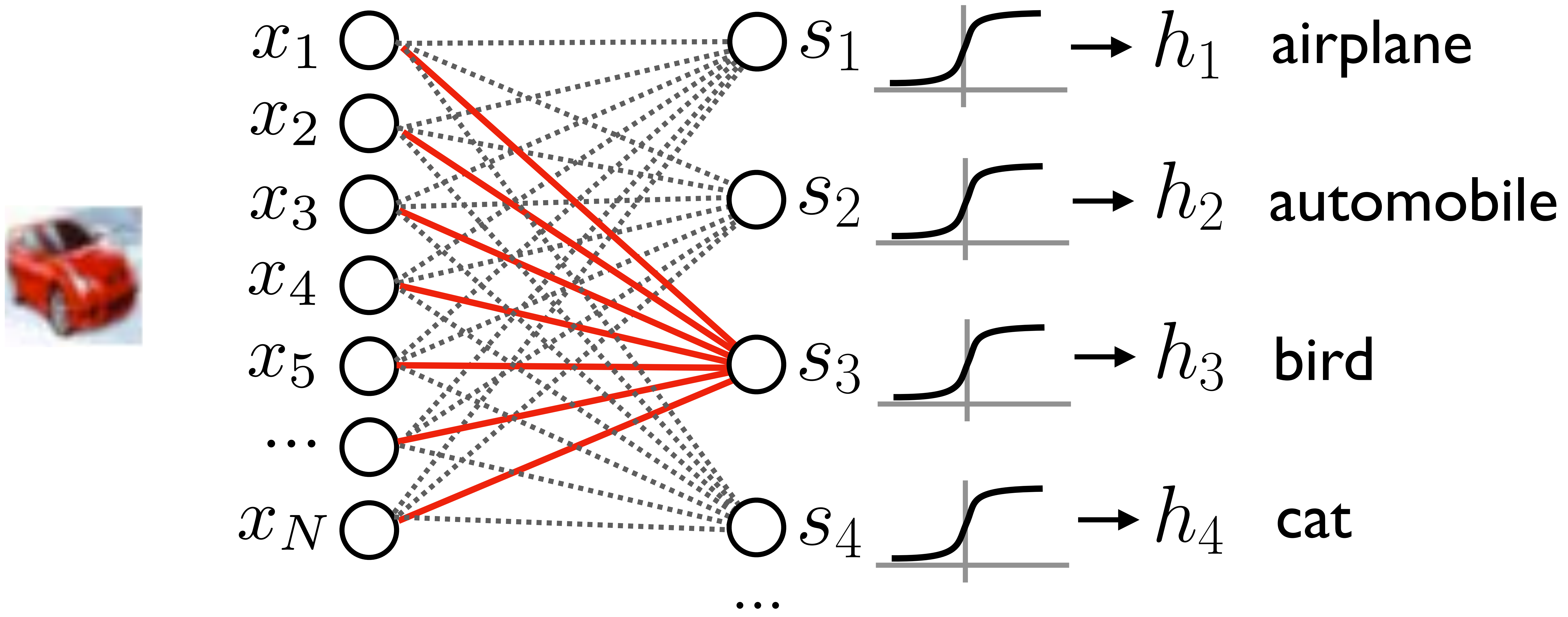


- Typically, we'll also add a bias term b

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

Linear = Fully Connected Layer

- Note that our linear matrix multiplication classifier is equivalent to a fully connected layer in a neural network

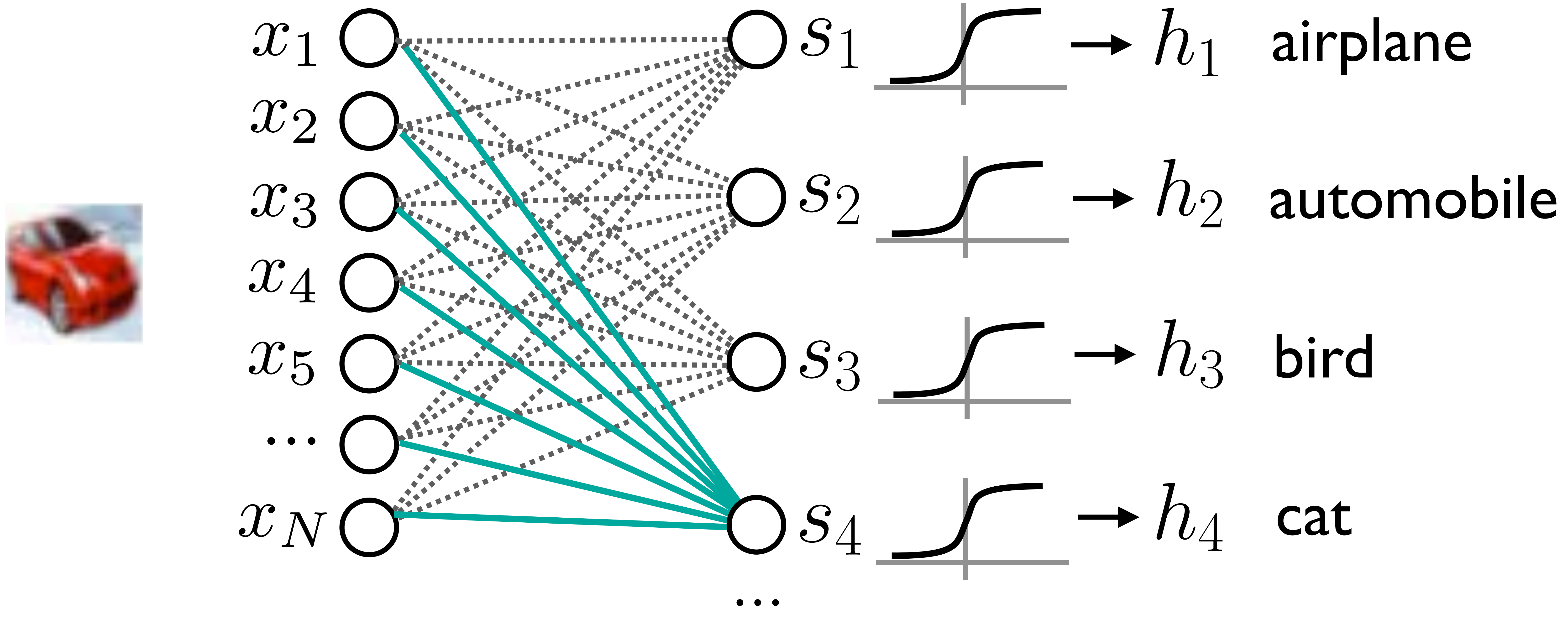


- Typically, we'll also add a bias term b

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

Linear = Fully Connected Layer

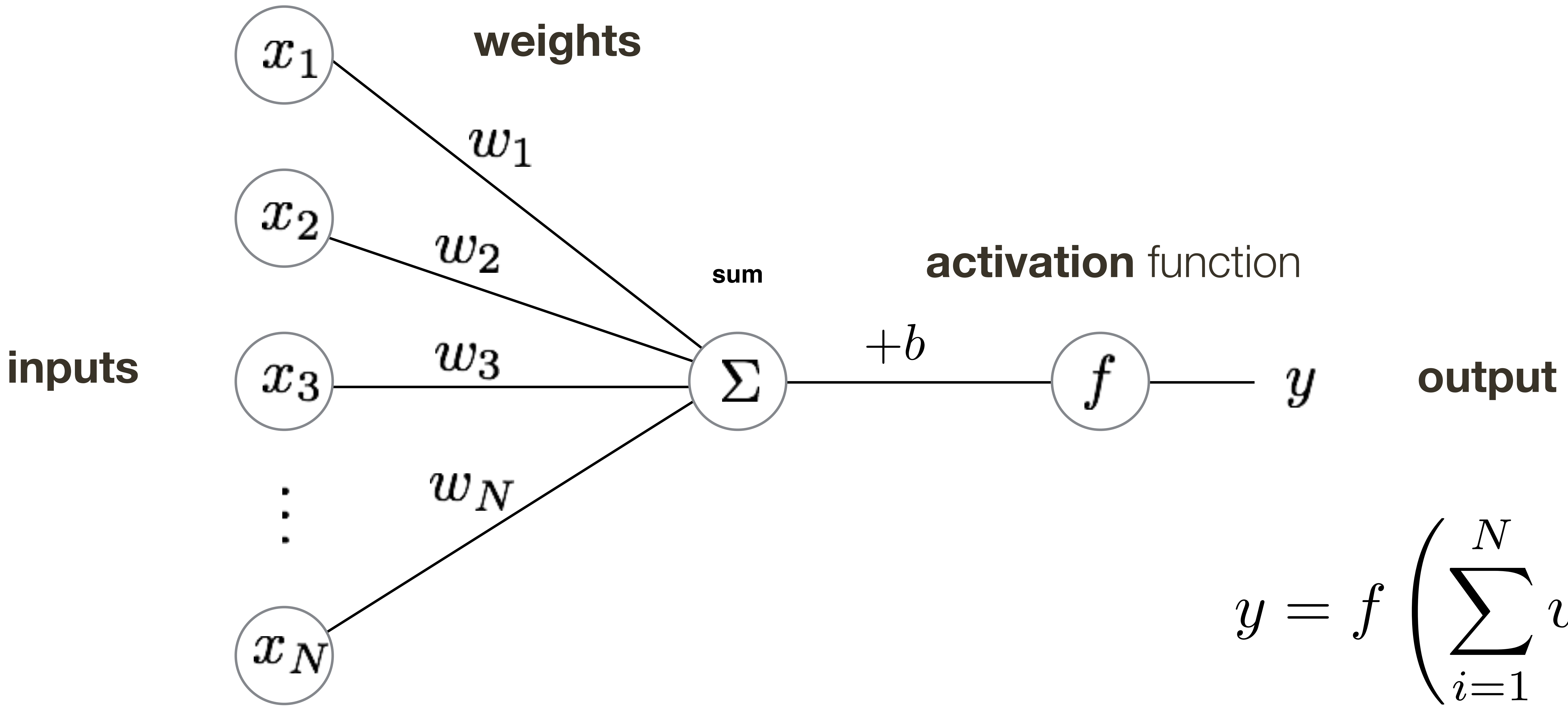
- Note that our linear matrix multiplication classifier is equivalent to a fully connected layer in a neural network



- Typically, we'll also add a bias term b

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

A **Neuron**



$$y = f\left(\sum_{i=1}^{N} w_i x_i + b\right)$$

— The basic unit of computation in a neural network is a neuron.

— A neuron accepts some number of input signals, computes their weighted sum, and applies an **activation function** (or **non-linearity**) to the sum.

— Common activation functions include sigmoid and rectified linear unit (ReLU)

Activation Function: **Sigmoid**

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

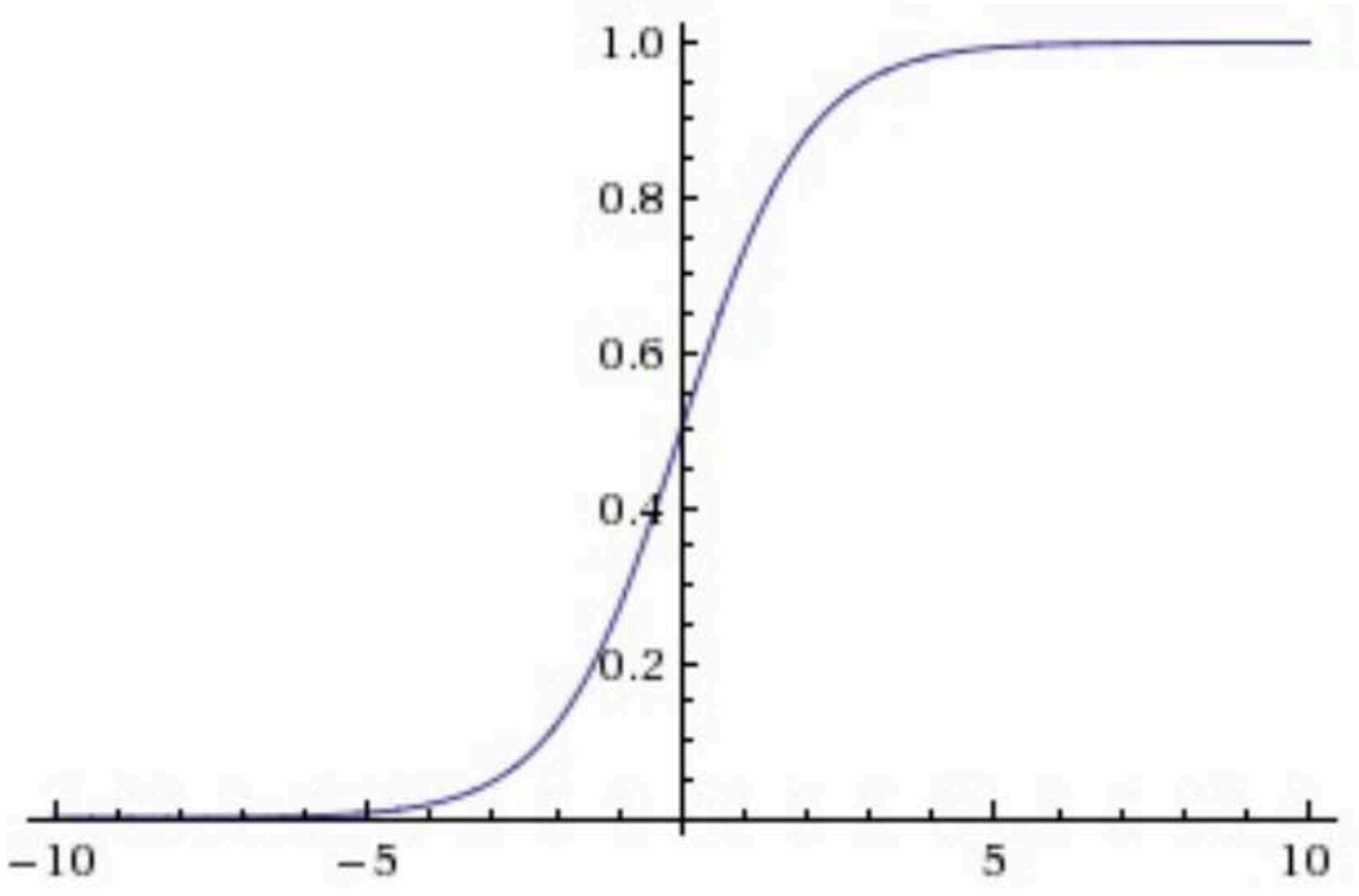


Figure credit: Fei-Fei and Karpathy

Common in many early neural networks

Biological analogy to saturated firing rate of neurons

Maps the input to the range [0,1]

Activation Function: **ReLU** (Rectified Linear Unit)

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

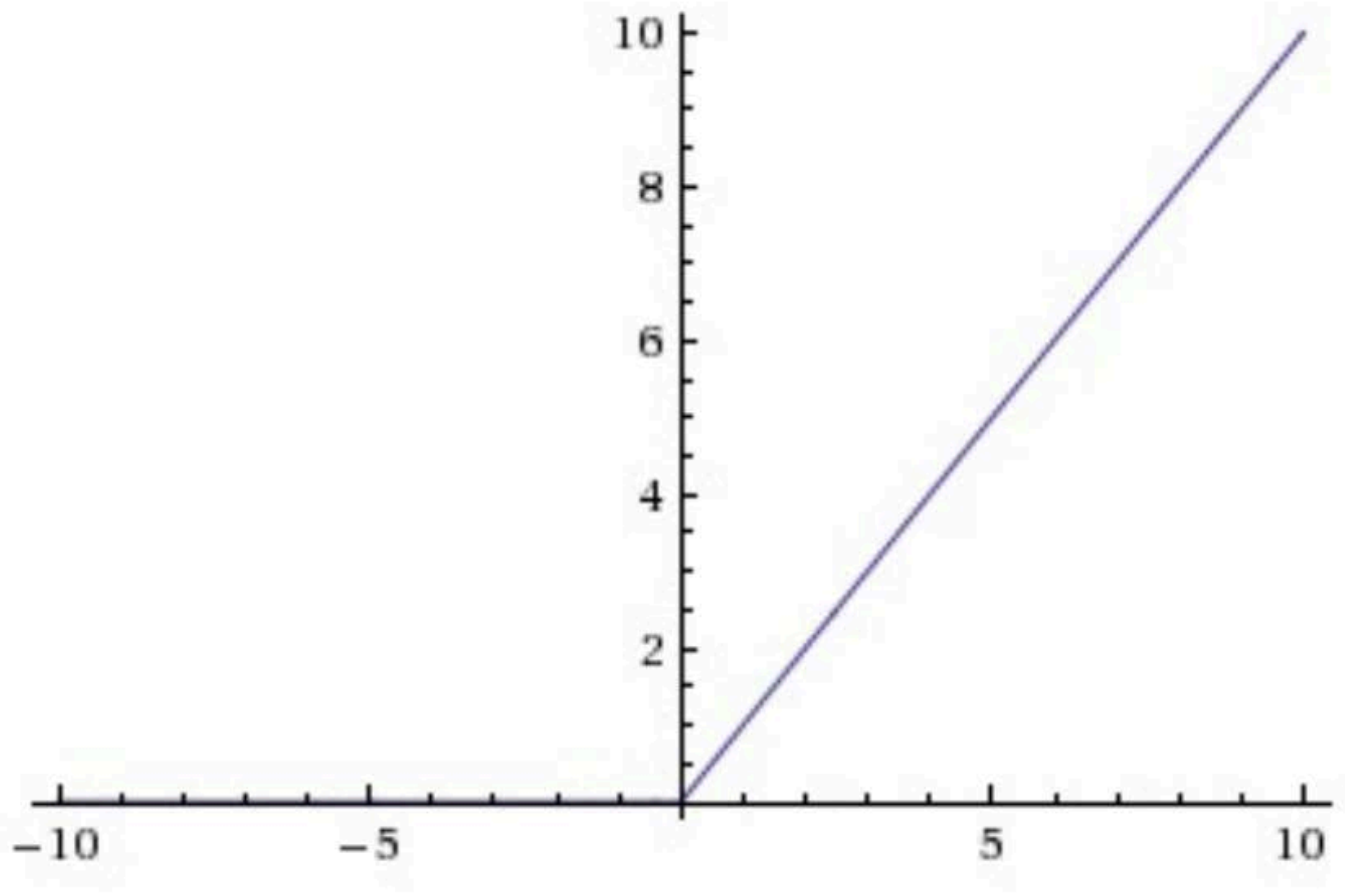


Figure credit: Fei-Fei and Karpathy

Maintains good gradient flow in networks, prevents vanishing gradient problem

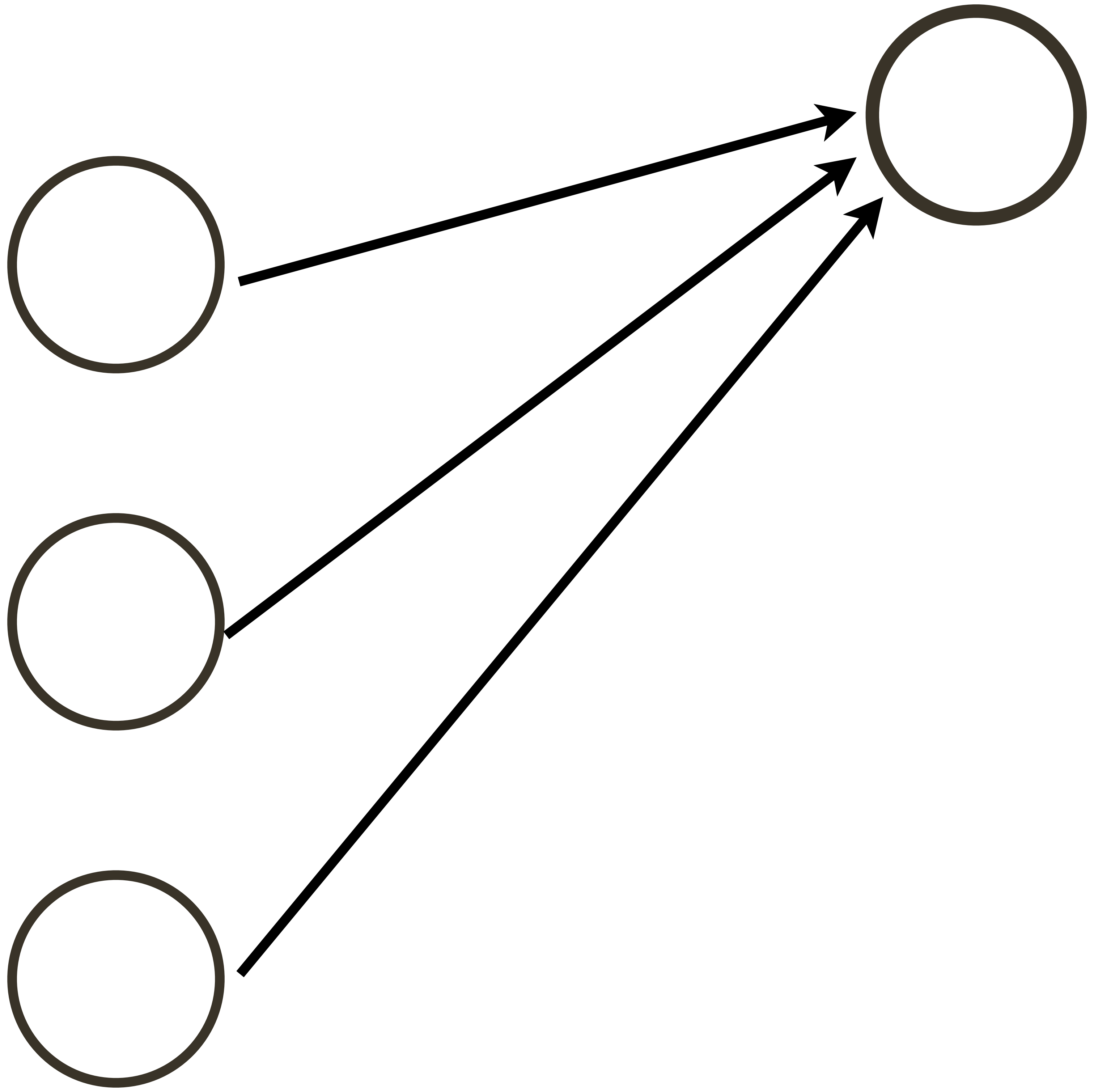
Very commonly used in interior (hidden) layers of neural nets

 19.3

 Why can't we have **linear** activation functions?

Neural Network

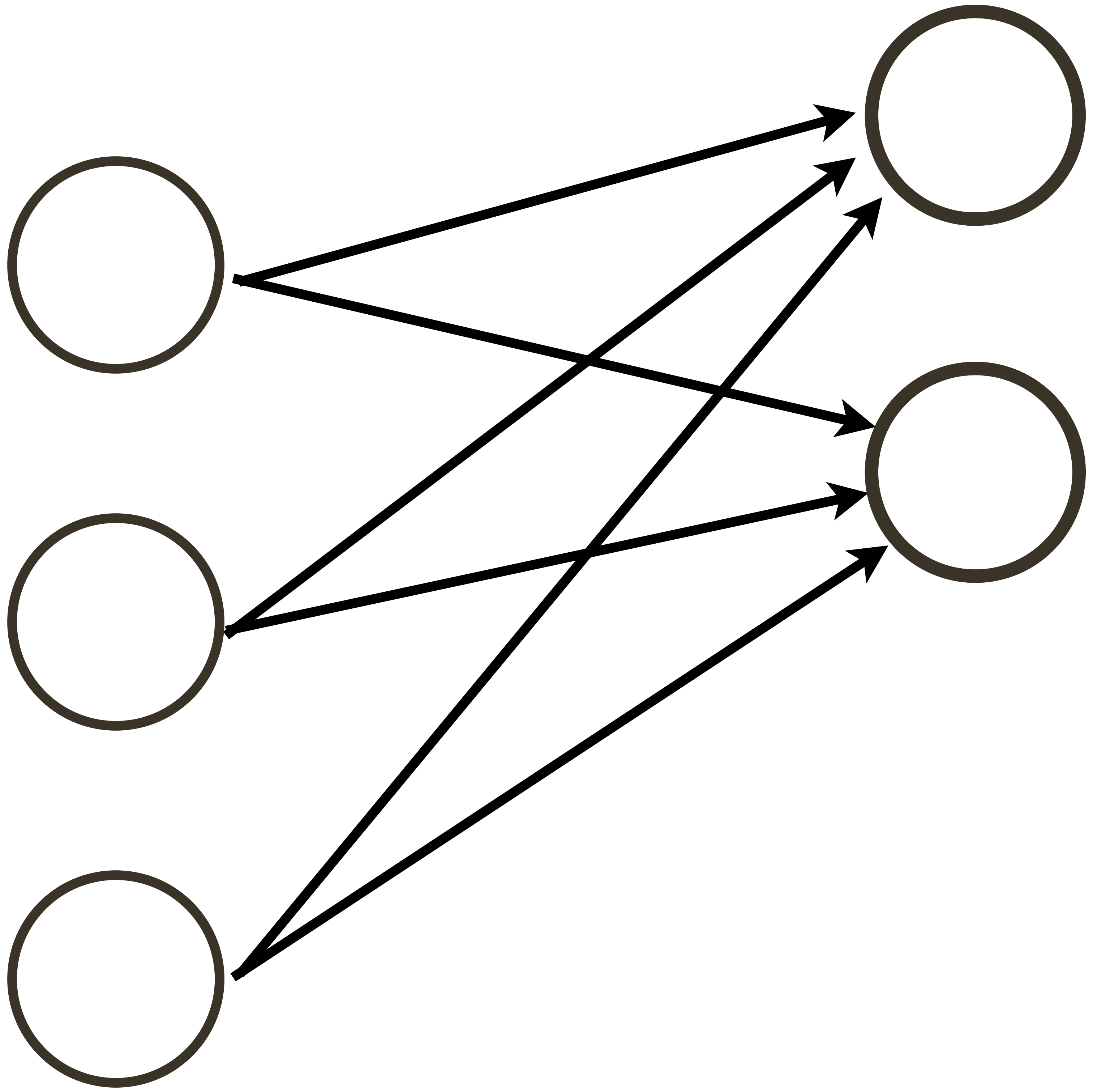
Connect a bunch of neurons together — a collection of connected neurons



‘one neuron’

Neural Network

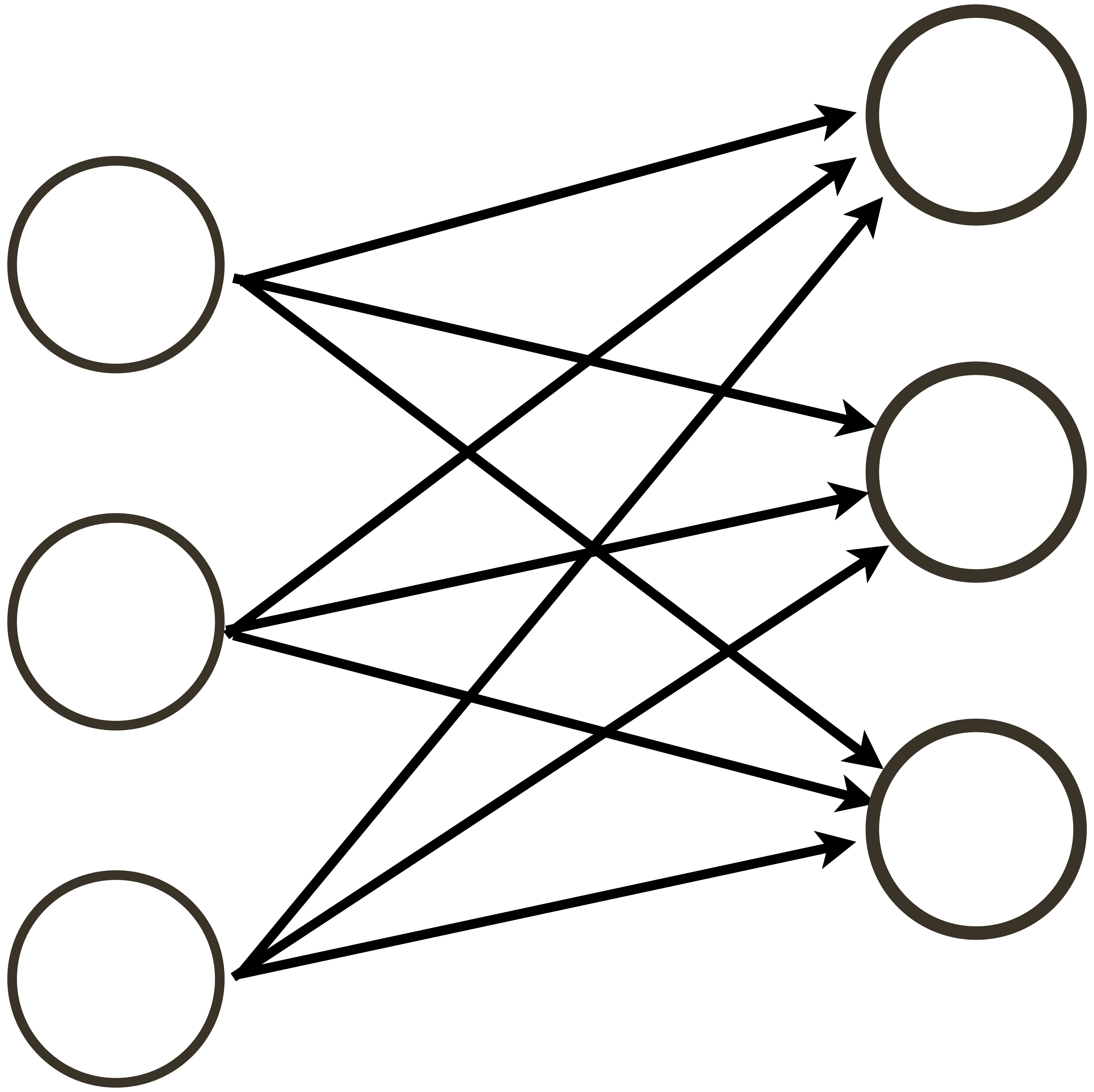
Connect a bunch of neurons together — a collection of connected neurons



‘two neurons’

Neural Network

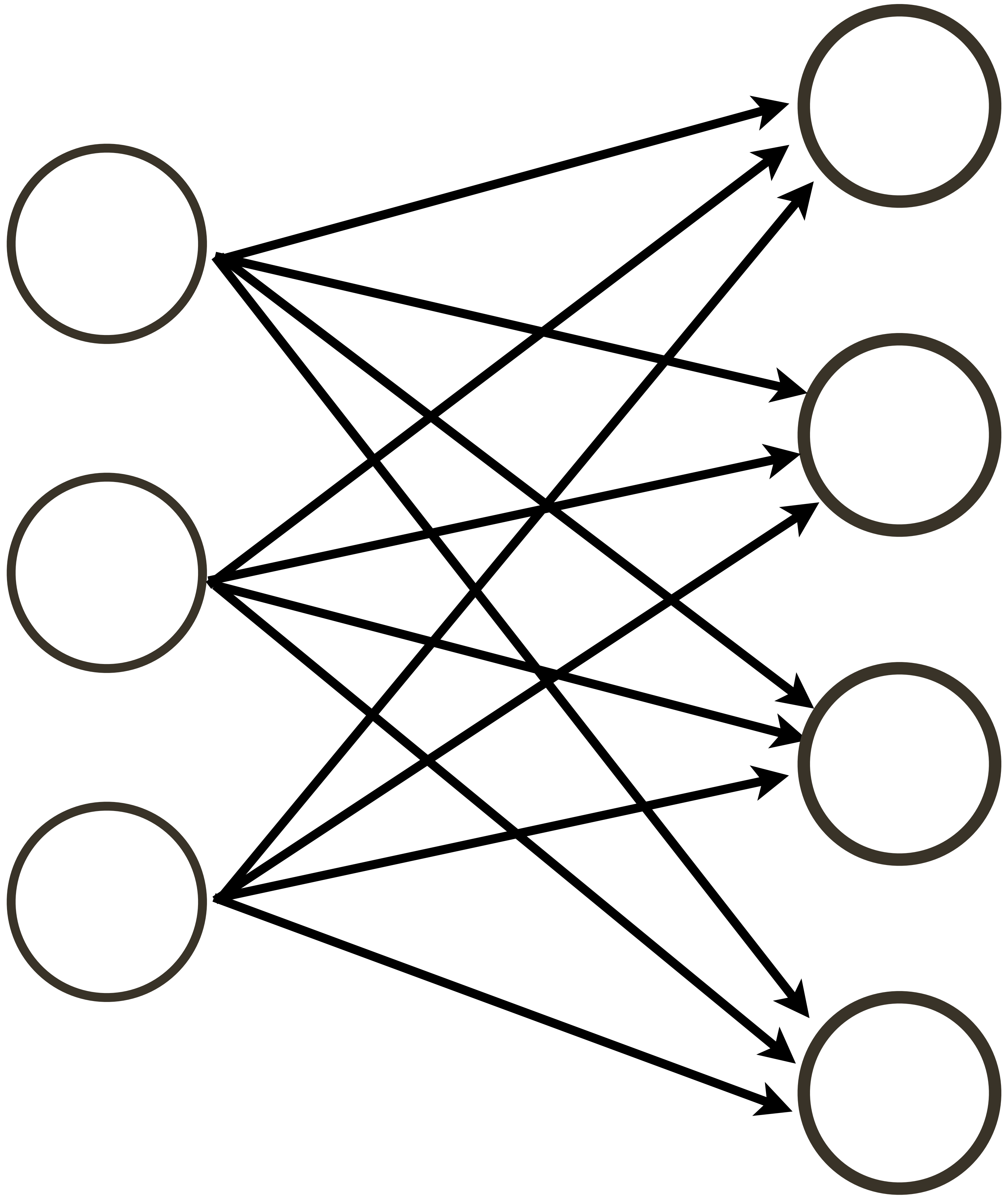
Connect a bunch of neurons together — a collection of connected neurons



‘three neurons’

Neural Network

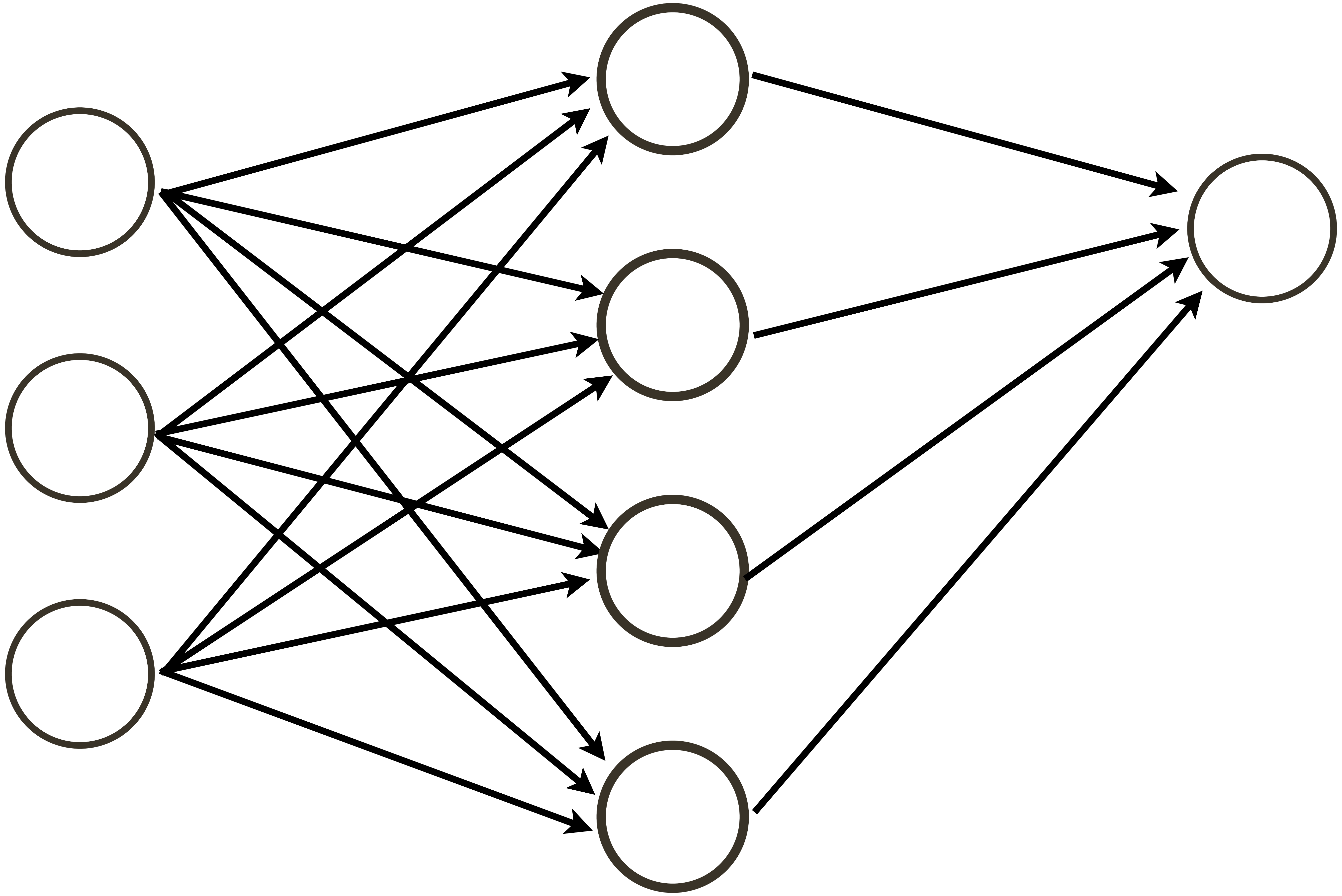
Connect a bunch of neurons together — a collection of connected neurons



'four neurons'

Neural Network

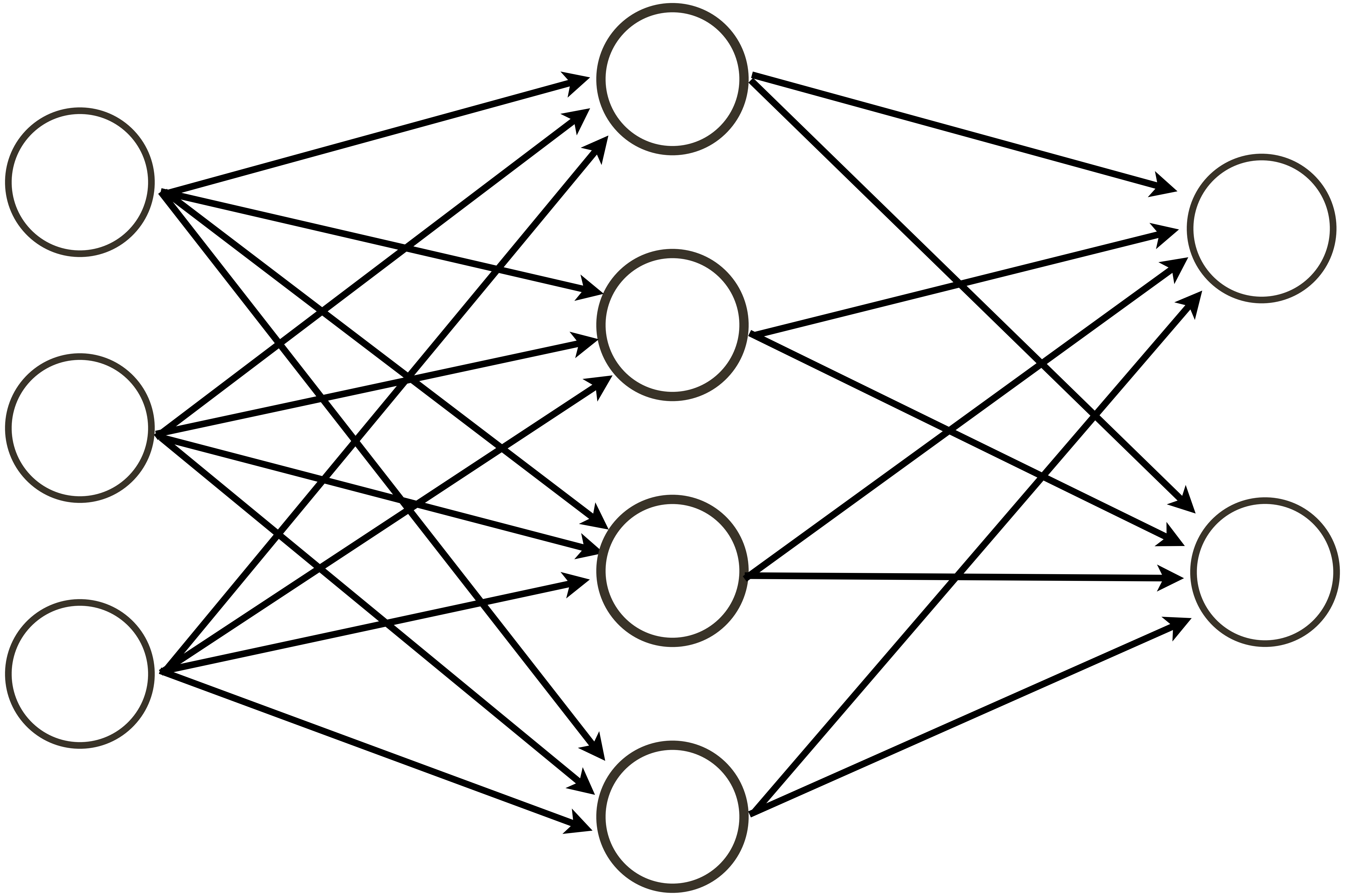
Connect a bunch of neurons together — a collection of connected neurons



‘five neurons’

Neural Network

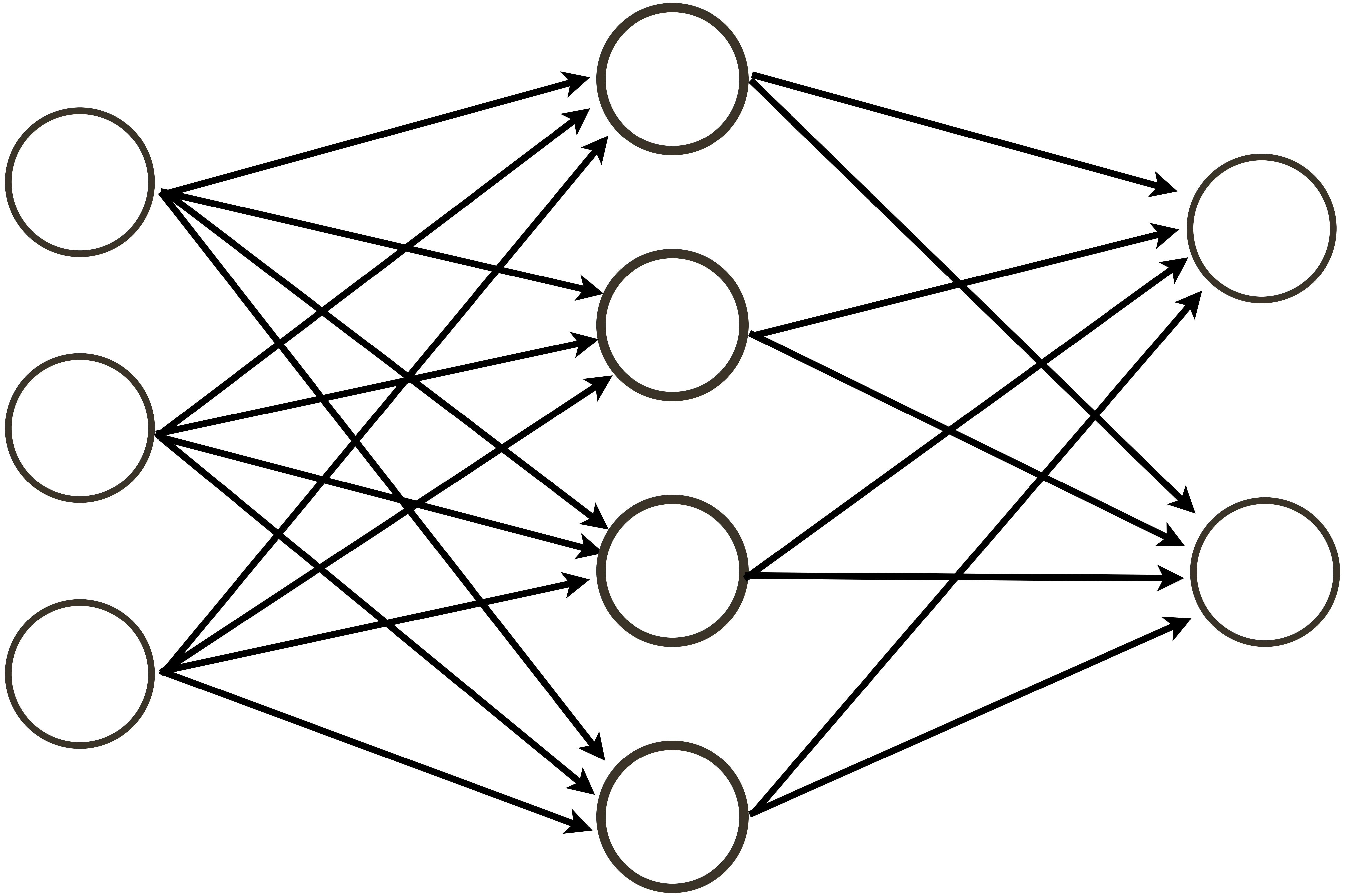
Connect a bunch of neurons together — a collection of connected neurons



‘six neurons’

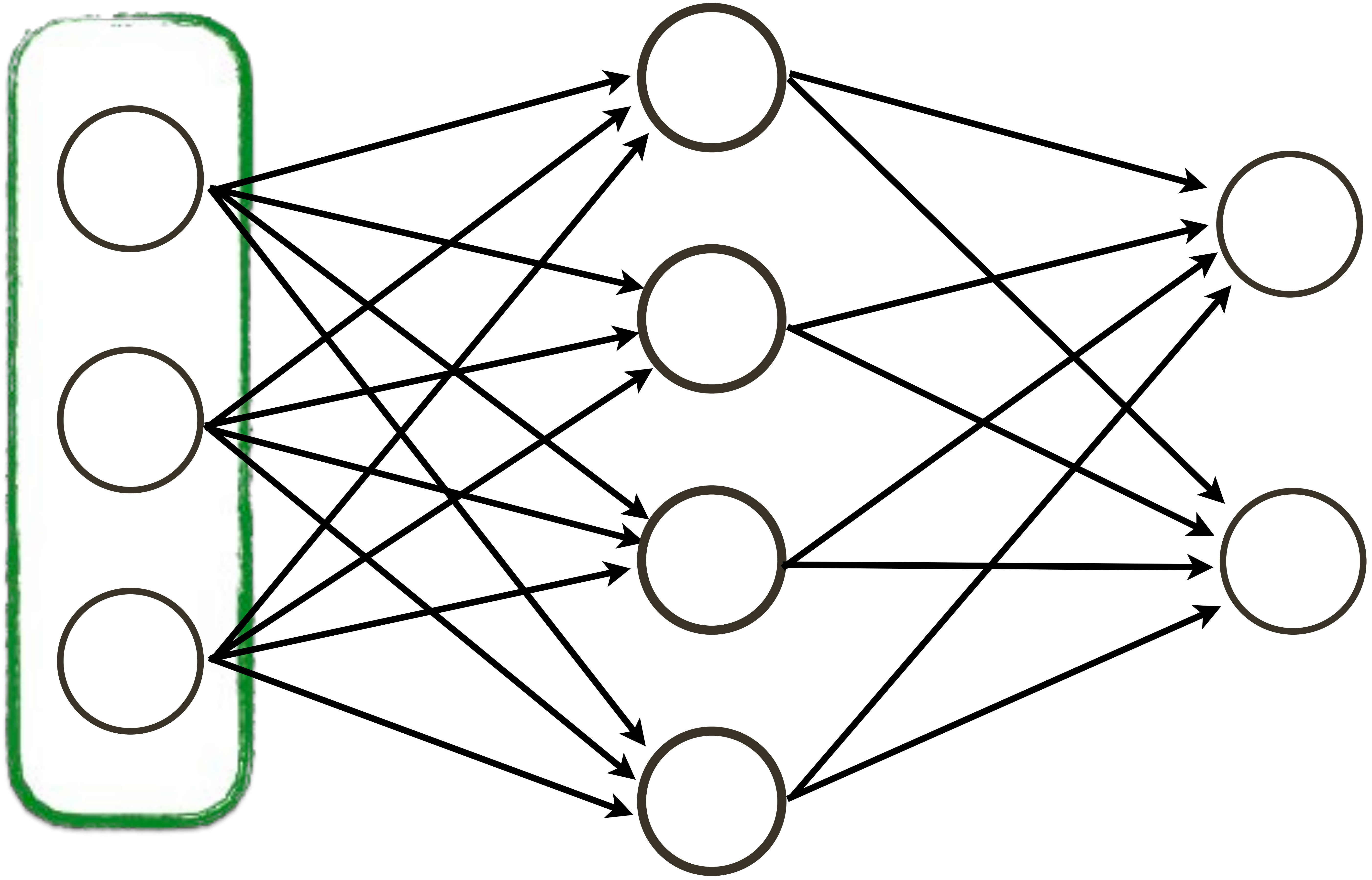
Neural Network

This network is also called a **Multi-layer Perceptron** (MLP)



Neural Network: **Terminology**

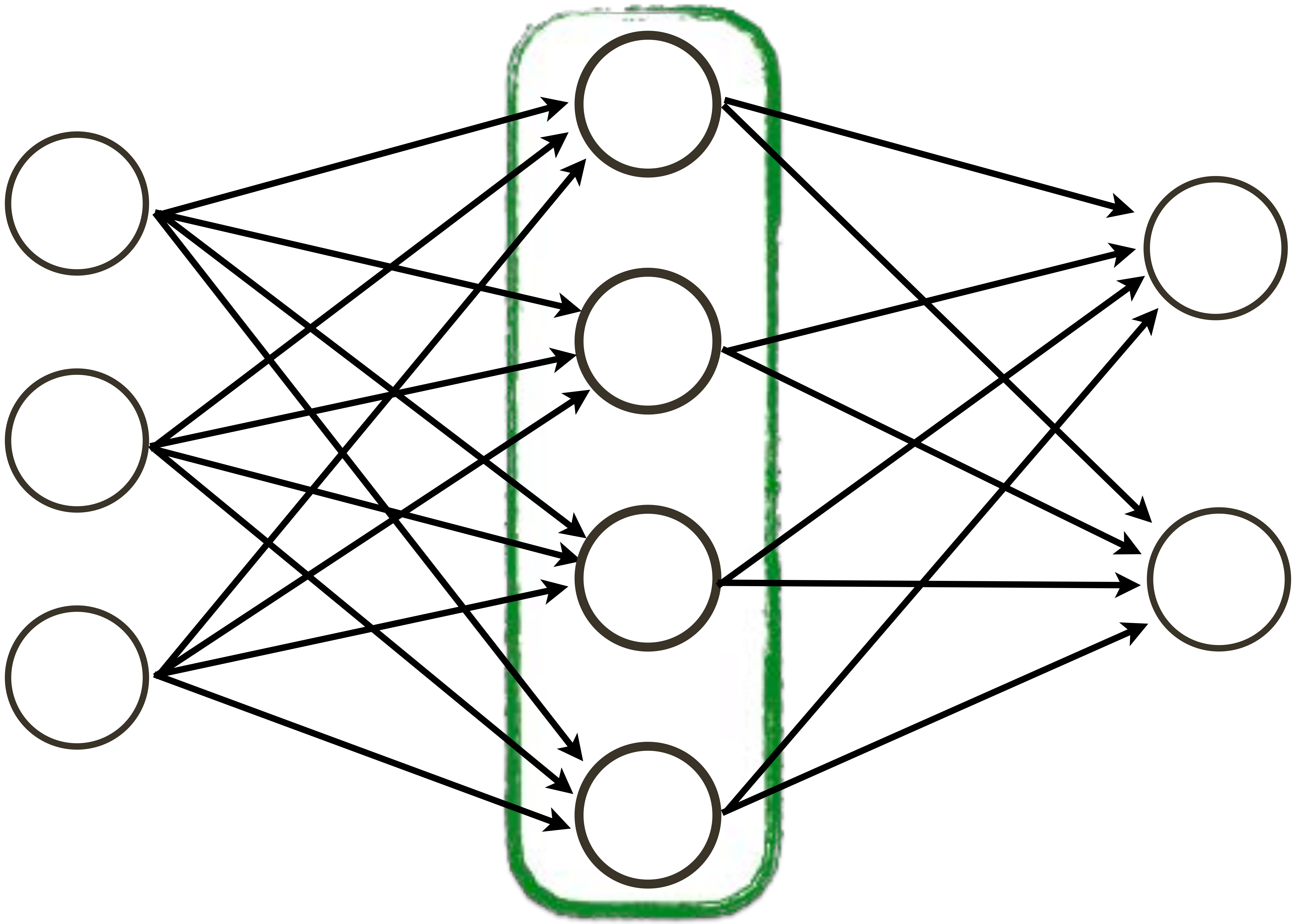
‘**input**’ layer



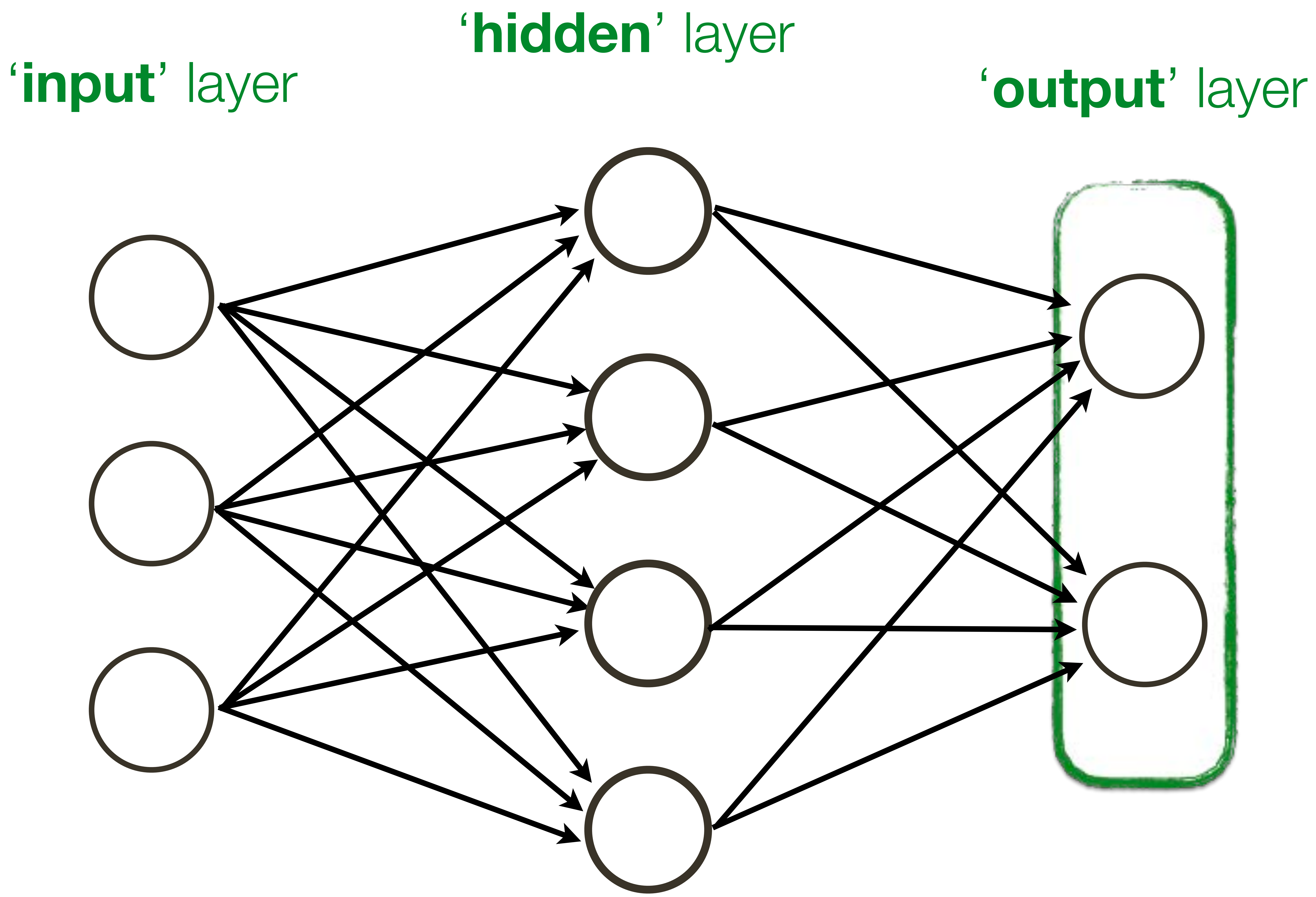
Neural Network: **Terminology**

‘**input**’ layer

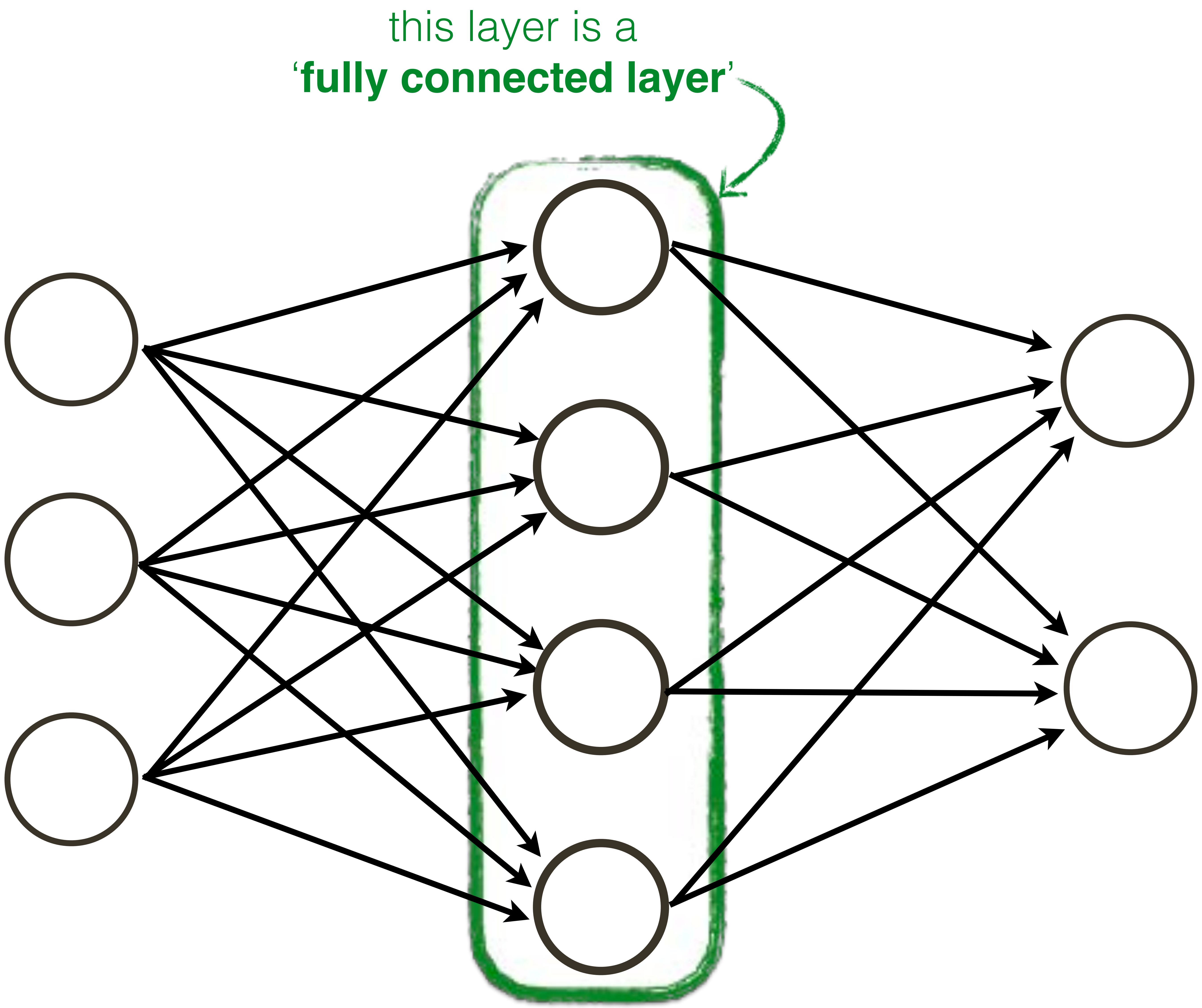
‘**hidden**’ layer



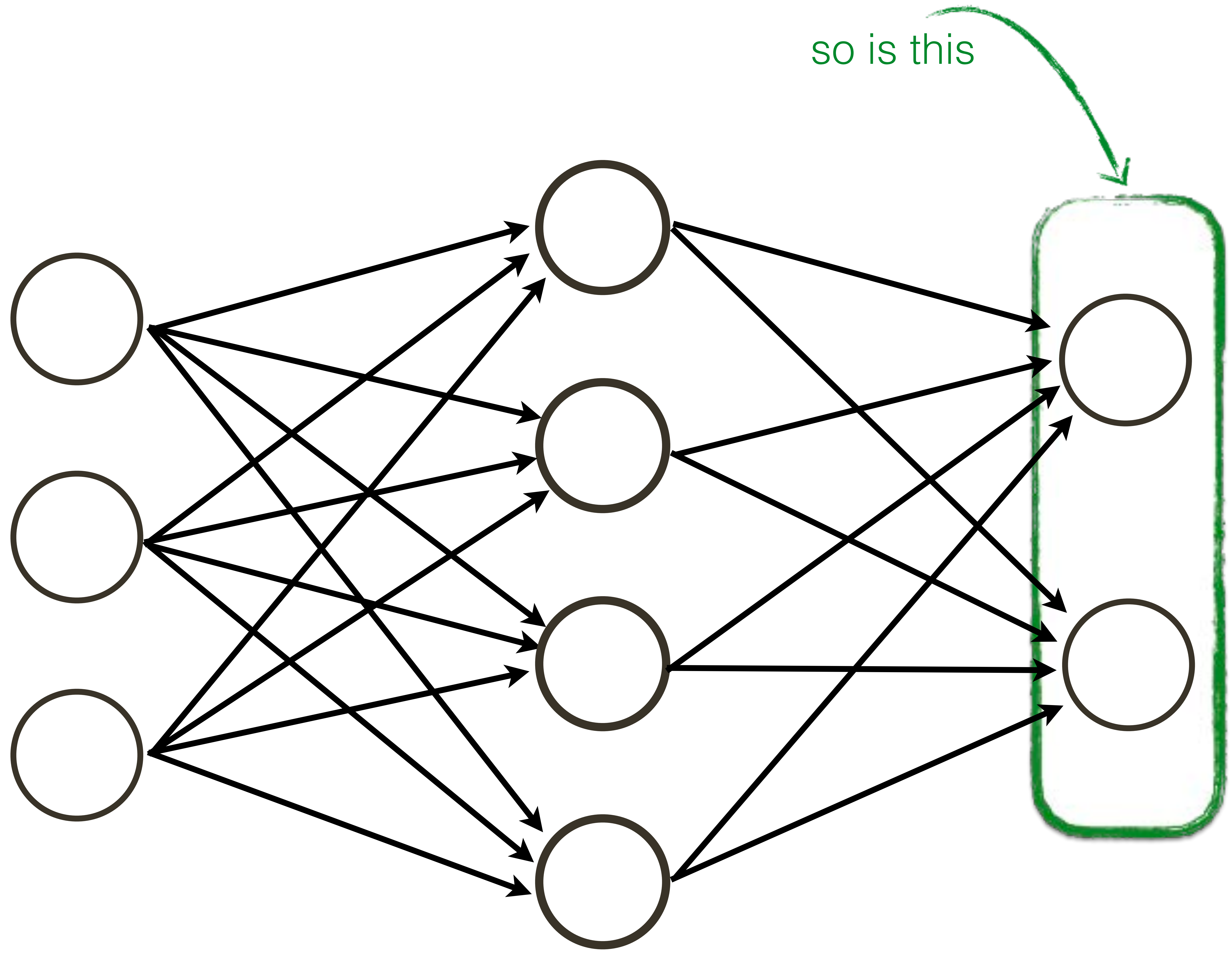
Neural Network: **Terminology**



Neural Network: **Terminology**

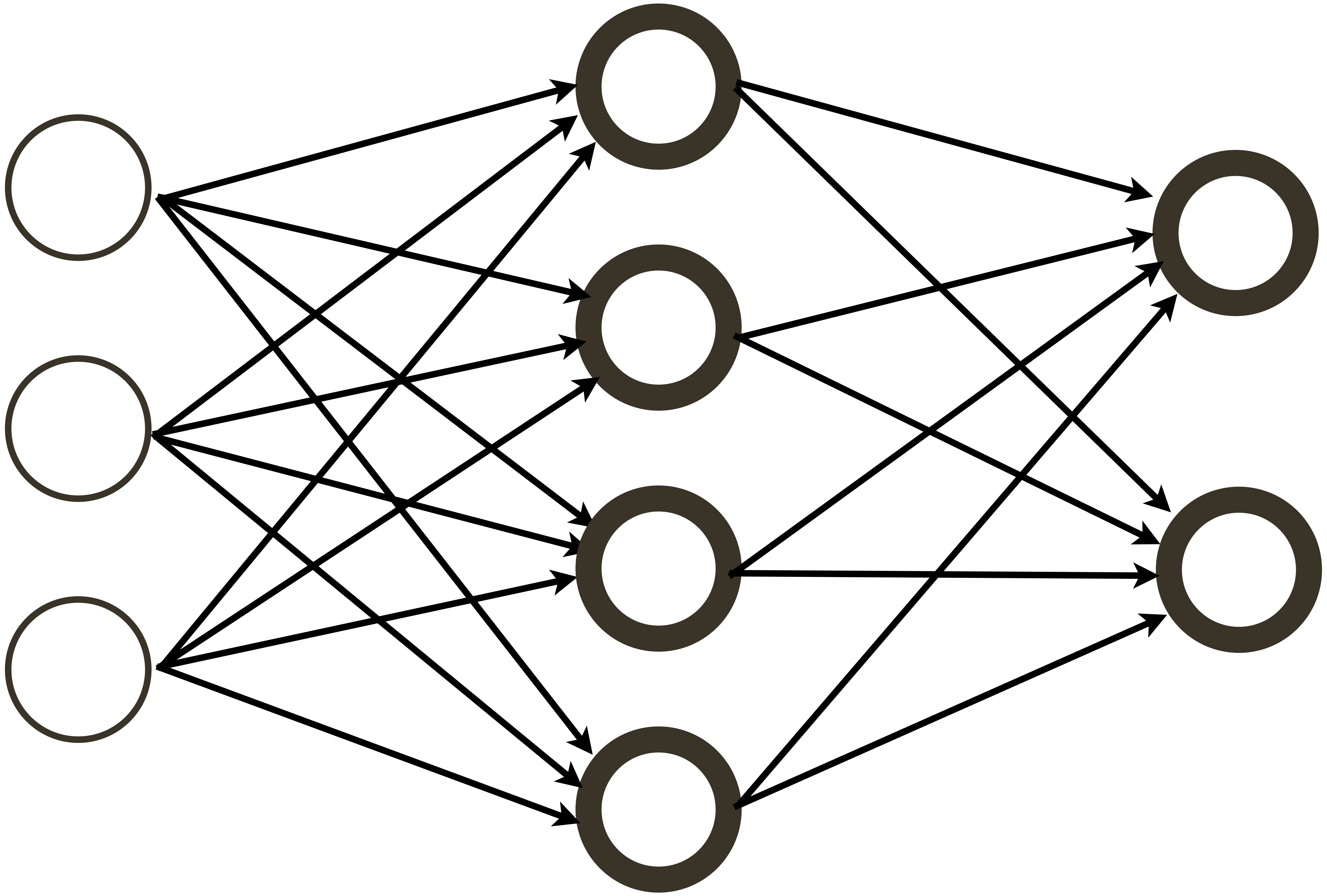


Neural Network: **Terminology**



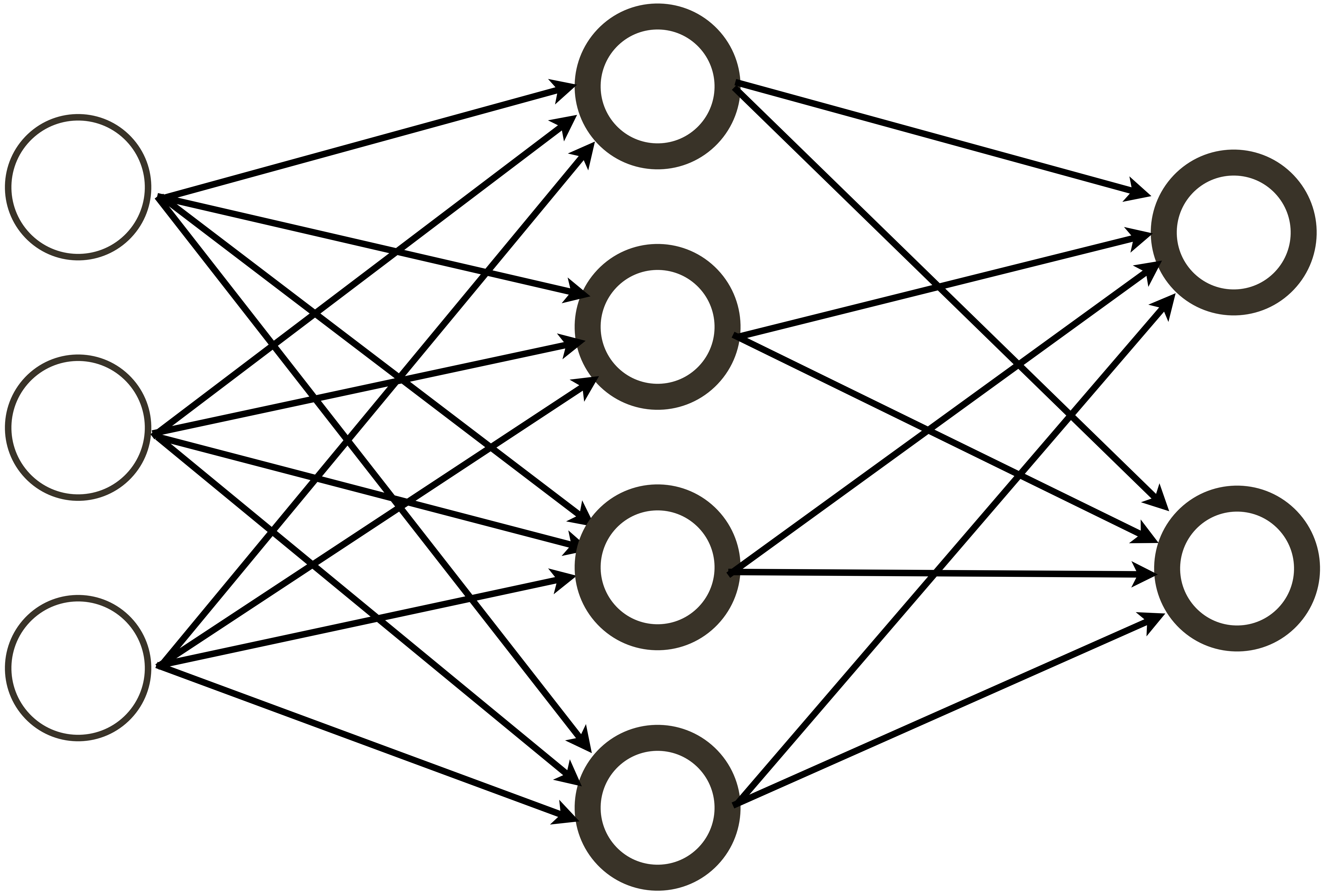
Neural Network

How many neurons?



Neural Network

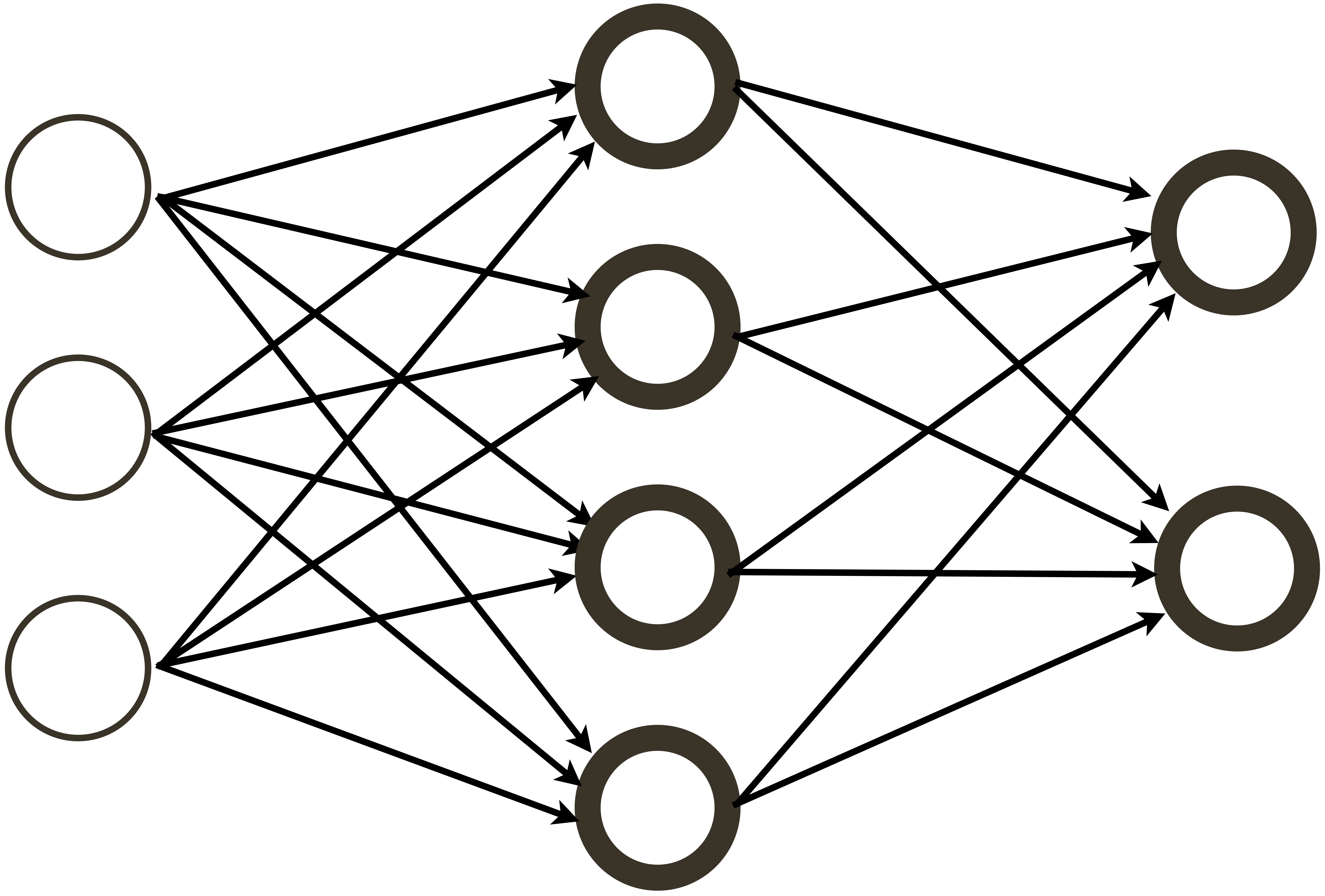
How many neurons?   4+2 = 6



Neural Network

How many neurons? 4+2 = 6

How many weights?

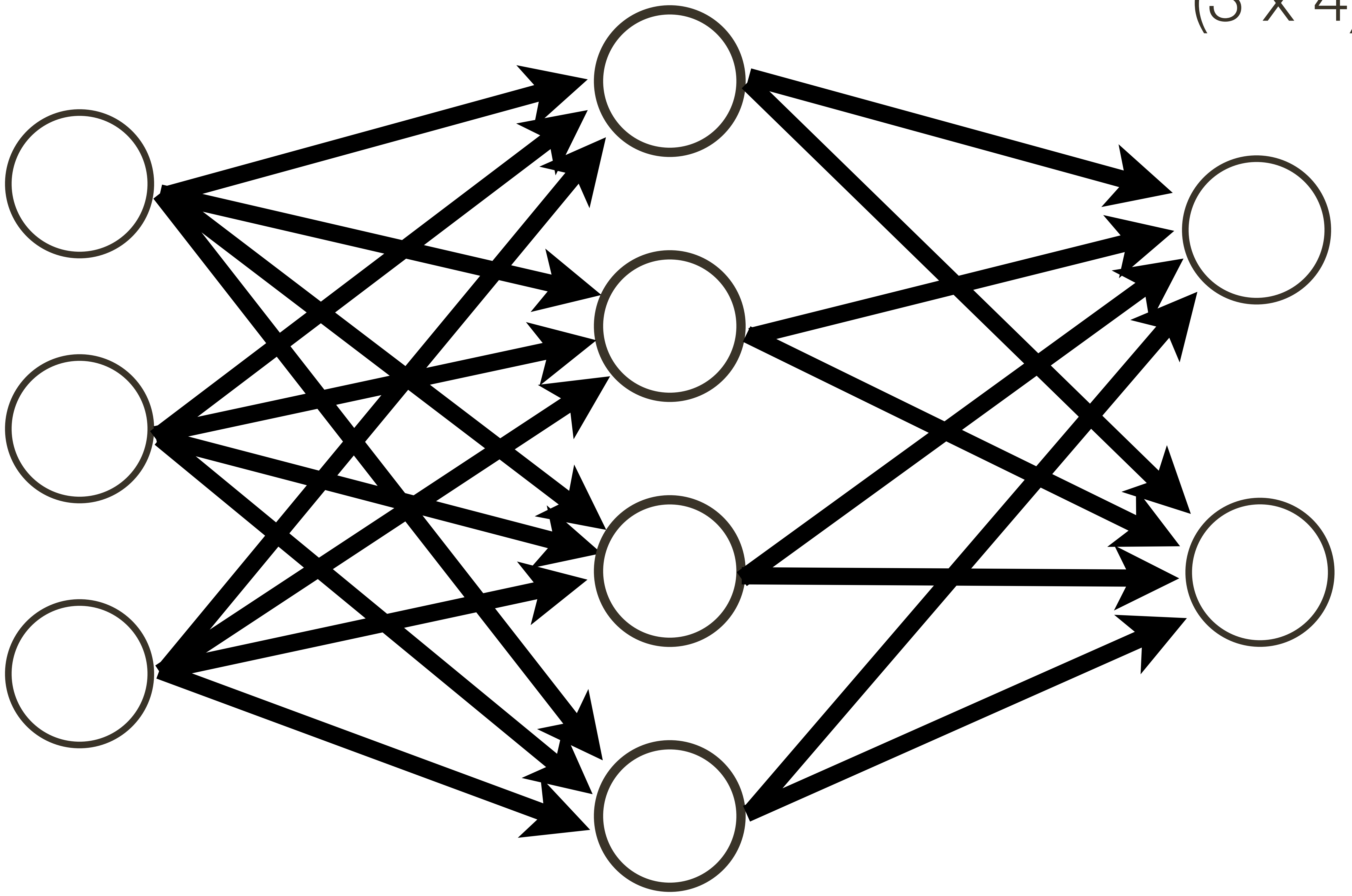


Neural Network

How many neurons? 4+2 = 6

How many weights?

(3 x 4) + (4 x 2) = 20

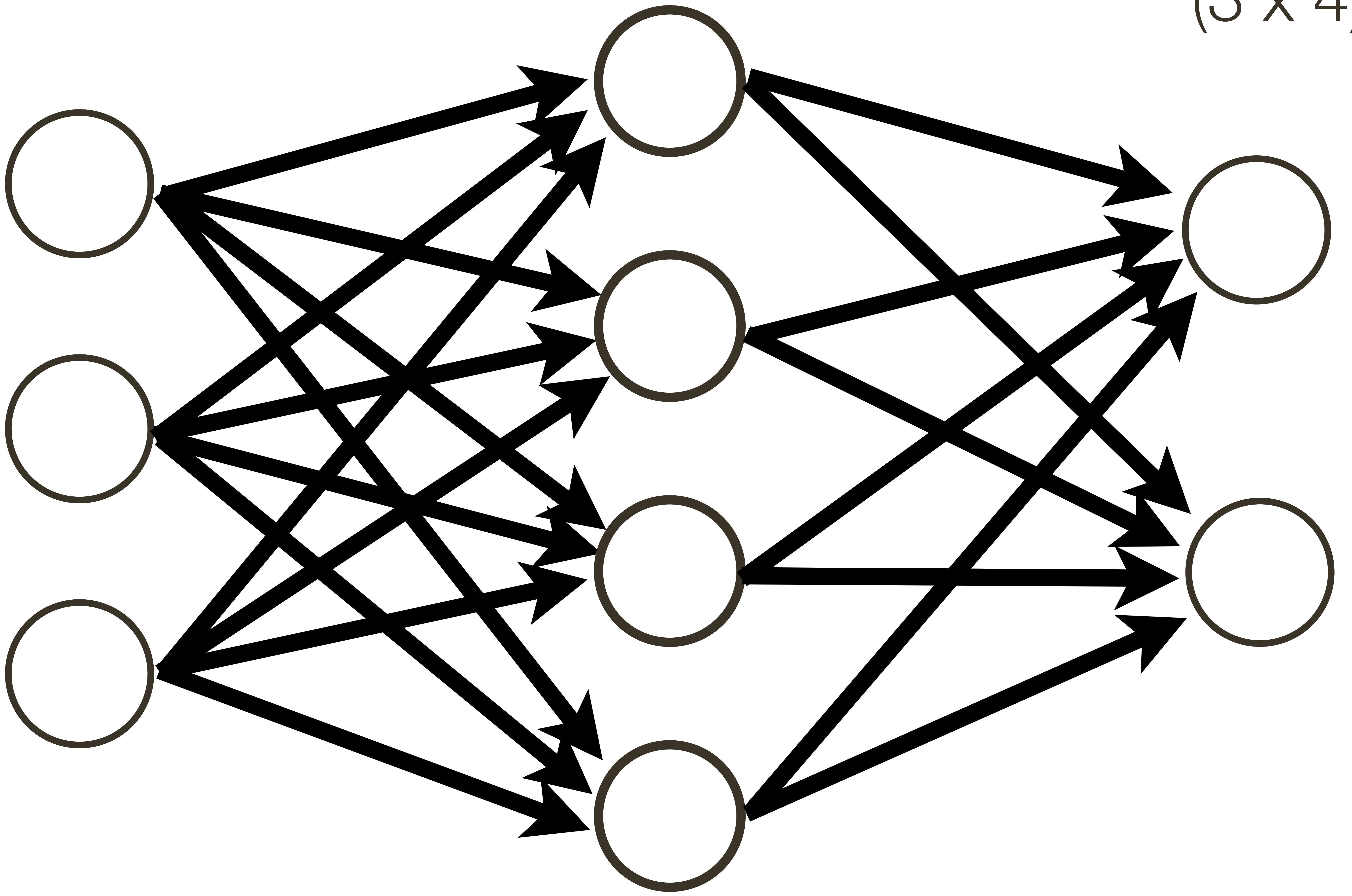


Neural Network

How many neurons? 4+2 = 6

How many weights?

(3 x 4) + (4 x 2) = 20



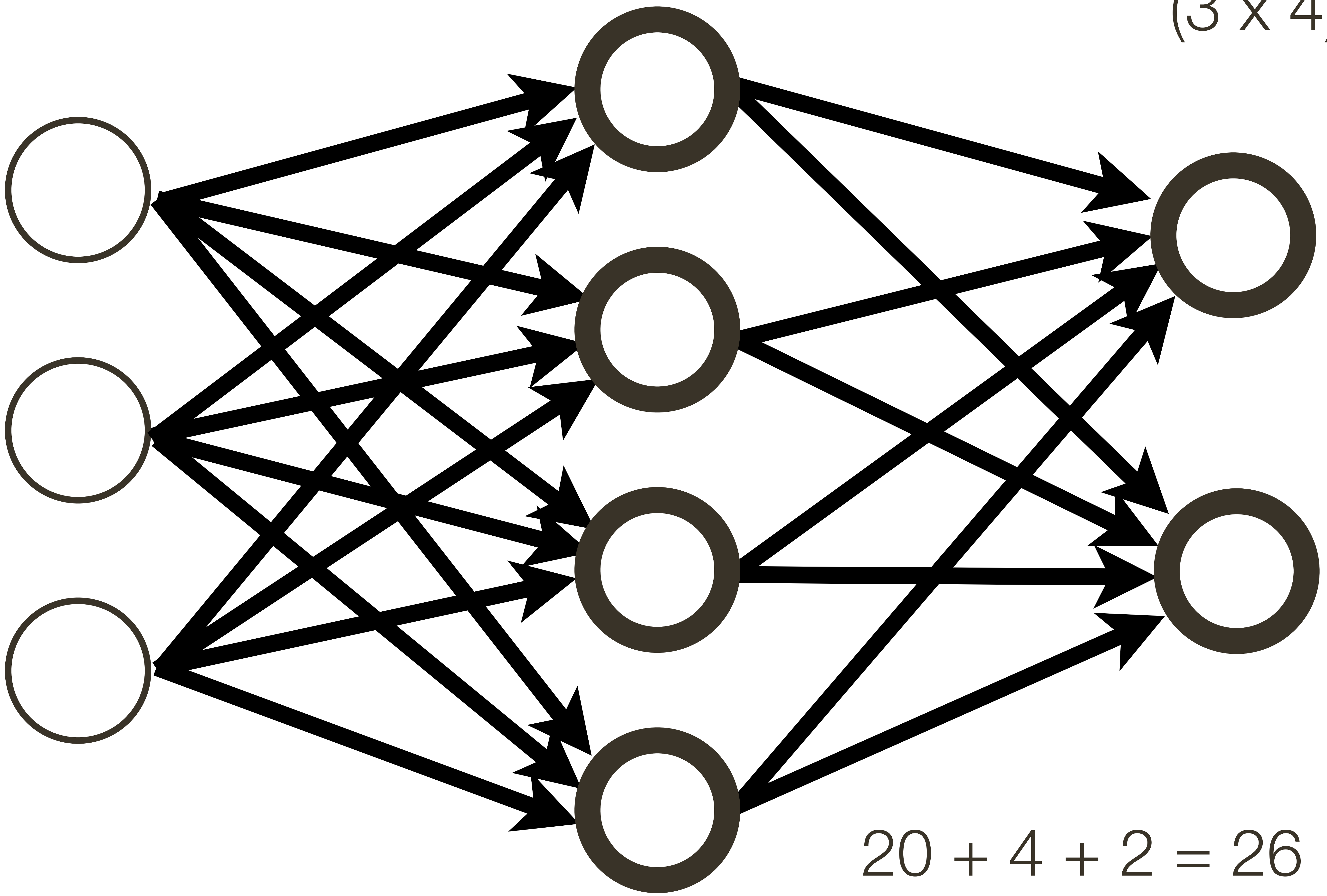
How many learnable parameters?

Neural Network

How many neurons? 4+2 = 6

How many weights?

(3 x 4) + (4 x 2) = 20



20 + 4 + 2 = 26

bias terms

How many learnable parameters?

Neural Network

A neural network comprises neurons connected in an acyclic graph

The outputs of neurons can become inputs to other neurons

Neural networks typically contain multiple layers of neurons

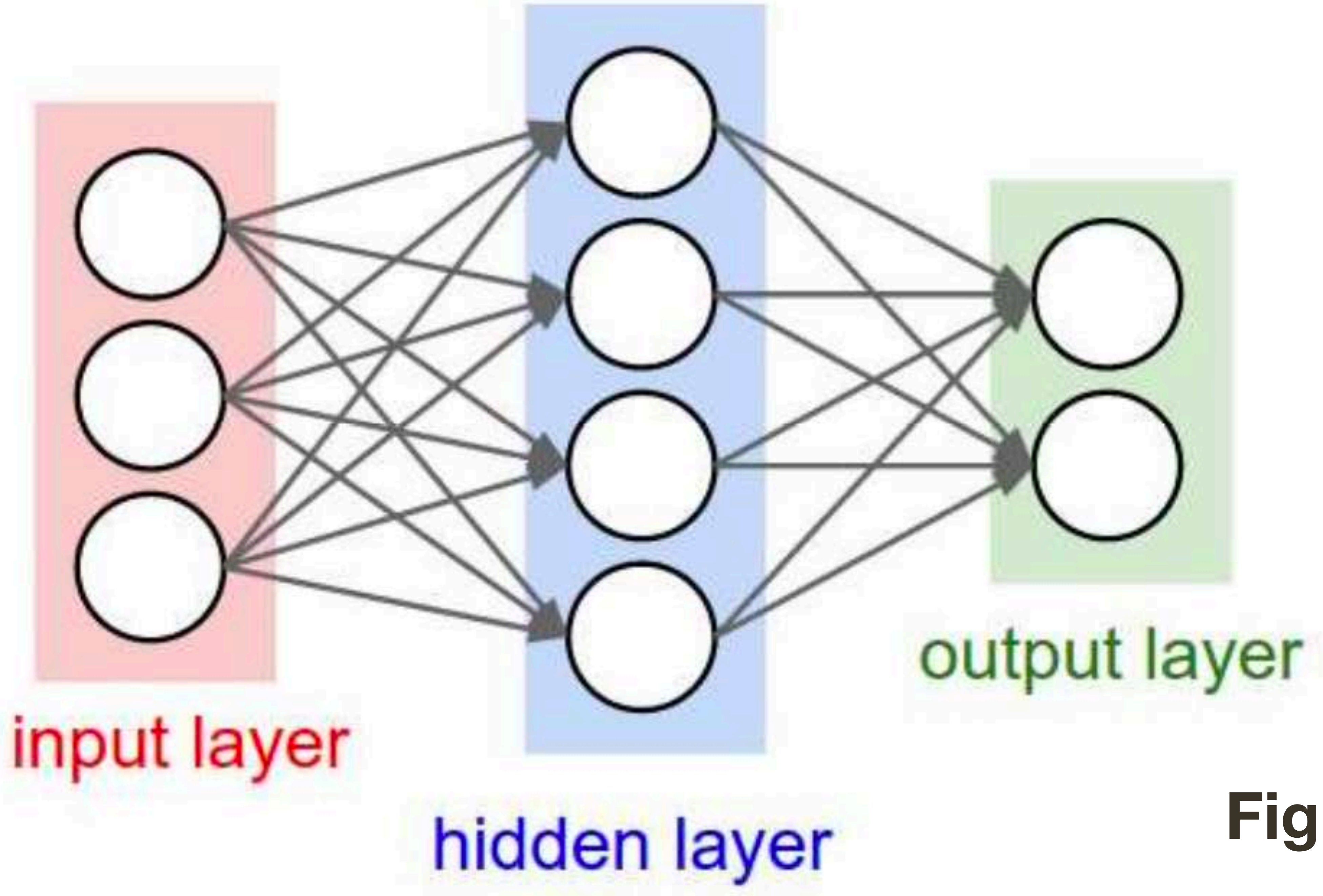


Figure credit: Fei-Fei and Karpathy

Example of a neural network with three inputs, a single hidden layer of four neurons, and an output layer of two neurons

Neural Network **Intuition**

Question: What is a Neural Network?

Answer: Complex mapping from an input (vector) to an output (vector)

* slide from Marc'Aurelio Renzato

Neural Network **Intuition**

Question: What is a Neural Network?

Answer: Complex mapping from an input (vector) to an output (vector)

Question: What class of functions should be considered for this mapping?

Answer: Compositions of simpler functions (a.k.a. layers)? We will talk more about what specific functions next …

* slide from Marc'Aurelio Renzato

Neural Network **Intuition**

Question: What is a Neural Network?

Answer: Complex mapping from an input (vector) to an output (vector)

Question: What class of functions should be considered for this mapping?

Answer: Compositions of simpler functions (a.k.a. layers)? We will talk more about what specific functions next …

Question: What does a hidden unit do?

Answer: It can be thought of as classifier or a feature.

* slide from Marc'Aurelio Renzato

Neural Network **Intuition**

Question: What is a Neural Network?

Answer: Complex mapping from an input (vector) to an output (vector)

Question: What class of functions should be considered for this mapping?

Answer: Compositions of simpler functions (a.k.a. layers)? We will talk more about what specific functions next …

Question: What does a hidden unit do?

Answer: It can be thought of as classifier or a feature.

Question: Why have many layers?

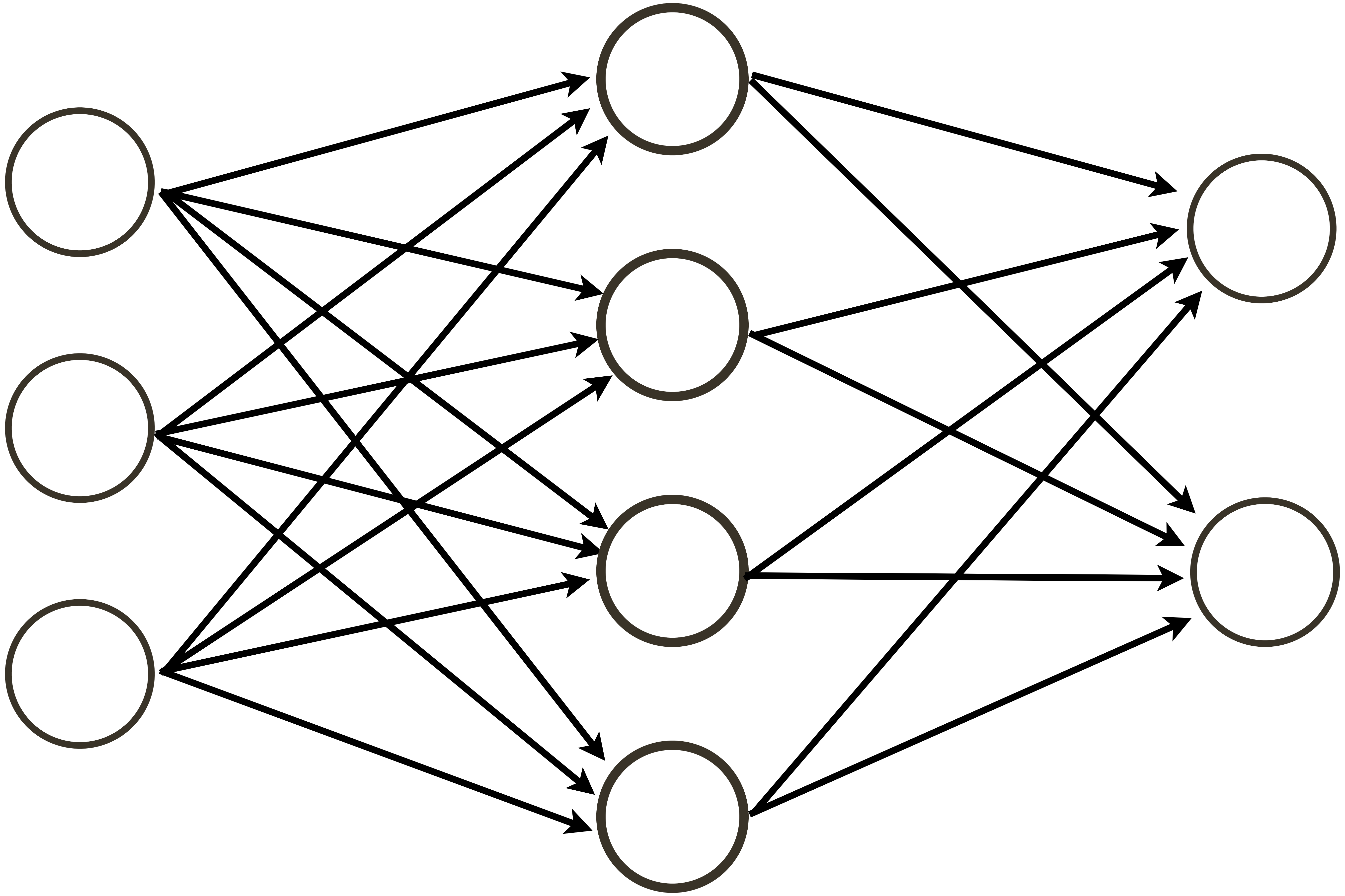
Answer: 1) More layers = more complex functional mapping

2) More efficient due to distributed representation

* slide from Marc'Aurelio Renzato

Activation Function

Why can't we have **linear** activation functions? Why have non-linear activations?



Activation Function

$$\hat{\mathbf{y}} = f(\mathbf{x}, \mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2) = \sigma\left(\mathbf{W}_2^{(2\times4)}\sigma\left(\mathbf{W}_1^{(4\times3)}\mathbf{x} + \mathbf{b}_1^{(4)}\right) + \mathbf{b}_2^{(2)}\right)$$

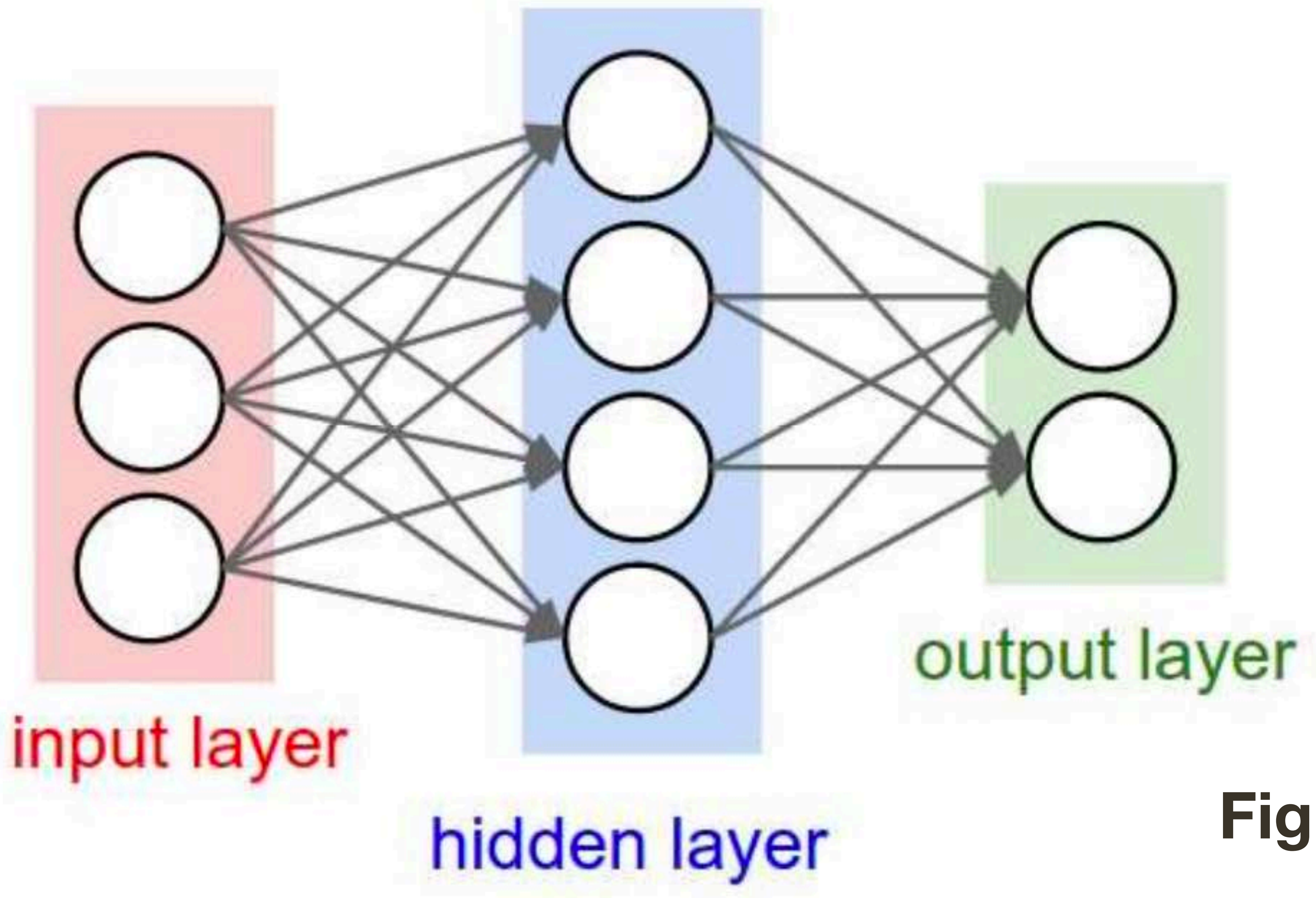


Figure credit: Fei-Fei and Karpathy

Activation Function

$$\hat{\mathbf{y}} = f(\mathbf{x}, \mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2) = \sigma\left(\mathbf{W}_2^{(2\times4)}\sigma\left(\mathbf{W}_1^{(4\times3)}\mathbf{x} + \mathbf{b}_1^{(4)}\right) + \mathbf{b}_2^{(2)}\right)$$

$$= \mathbf{W}_2^{(2\times4)}\left(\mathbf{W}_1^{(4\times3)}\mathbf{x} + \mathbf{b}_1^{(4)}\right) + \mathbf{b}_2^{(2)}$$

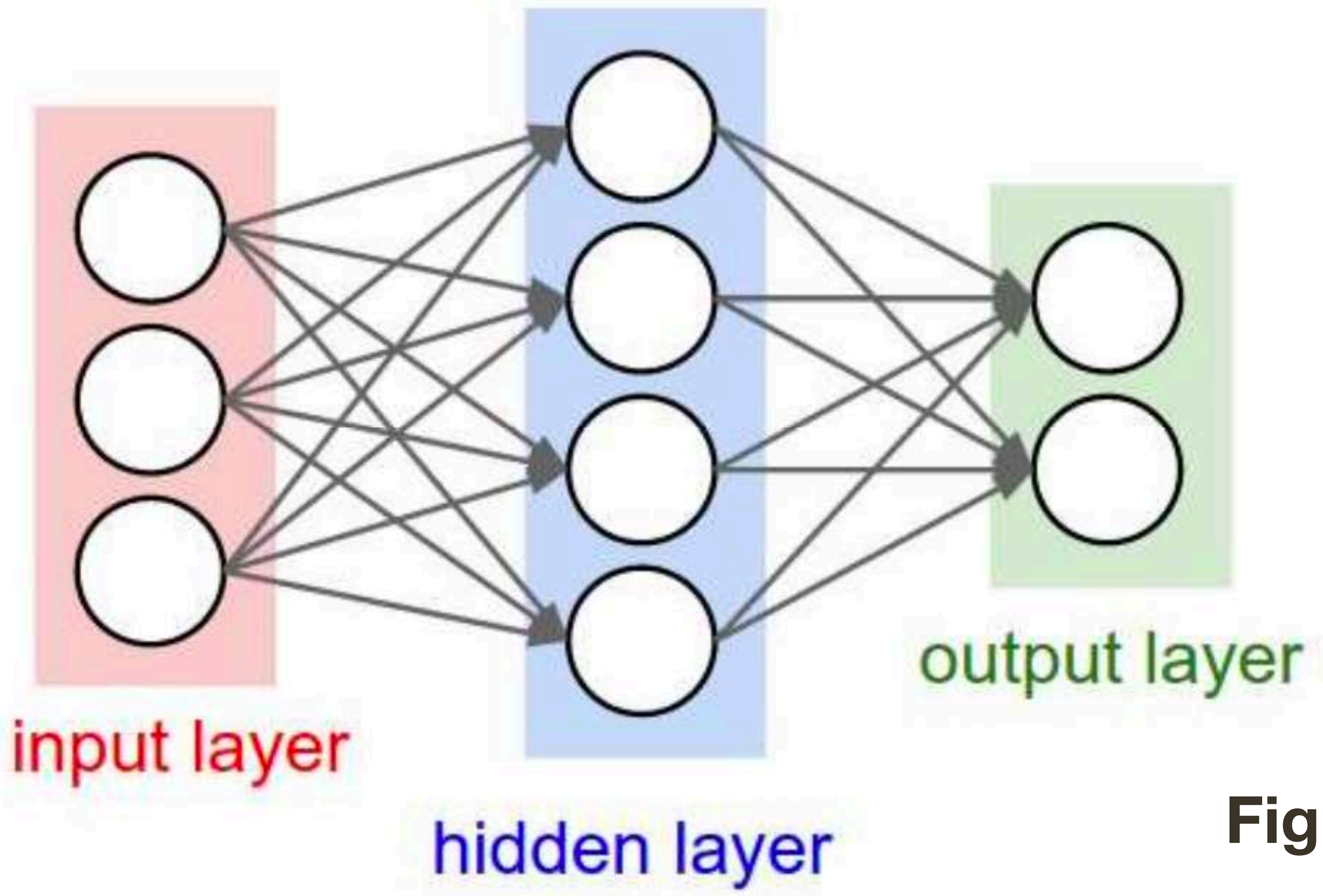


Figure credit: Fei-Fei and Karpathy

Activation Function

$$
\begin{aligned}
\hat{\mathbf{y}} = f(\mathbf{x}, \mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2) &= \sigma\left(\mathbf{W}_2^{(2\times4)}\sigma\left(\mathbf{W}_1^{(4\times3)}\mathbf{x} + \mathbf{b}_1^{(4)}\right) + \mathbf{b}_2^{(2)}\right) \\
&= \mathbf{W}_2^{(2\times4)}\left(\mathbf{W}_1^{(4\times3)}\mathbf{x} + \mathbf{b}_1^{(4)}\right) + \mathbf{b}_2^{(2)} \\
&= \mathbf{W}_2^{(2\times4)}\mathbf{W}_1^{(4\times3)}\mathbf{x} + \mathbf{W}_2^{(2\times4)}\mathbf{b}_1^{(4)} + \mathbf{b}_2^{(2)}
\end{aligned}
$$

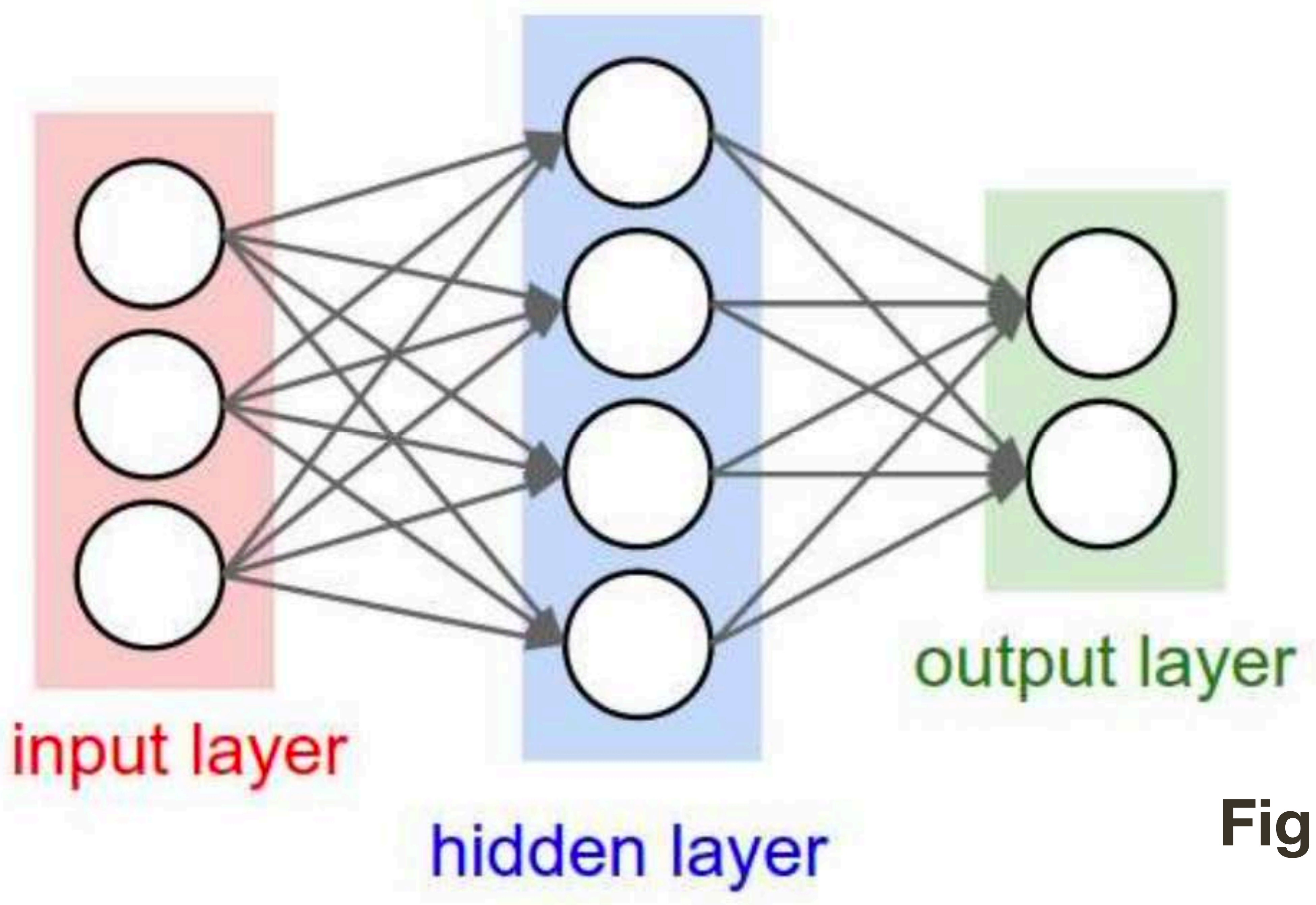


Figure credit: Fei-Fei and Karpathy

Activation Function

$$
\begin{aligned}
\hat{\mathbf{y}} = f(\mathbf{x}, \mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2) &= \sigma\left(\mathbf{W}_2^{(2\times4)}\sigma\left(\mathbf{W}_1^{(4\times3)}\mathbf{x} + \mathbf{b}_1^{(4)}\right) + \mathbf{b}_2^{(2)}\right) \\
&= \mathbf{W}_2^{(2\times4)}\left(\mathbf{W}_1^{(4\times3)}\mathbf{x} + \mathbf{b}_1^{(4)}\right) + \mathbf{b}_2^{(2)} \\
&= \underbrace{\mathbf{W}_2^{(2\times4)}\mathbf{W}_1^{(4\times3)}}_{\mathbf{W}_*^{(2\times3)}}\mathbf{x} + \underbrace{\mathbf{W}_2^{(2\times4)}\mathbf{b}_1^{(4)} + \mathbf{b}_2^{(2)}}_{\mathbf{b}^{(2)}}
\end{aligned}
$$

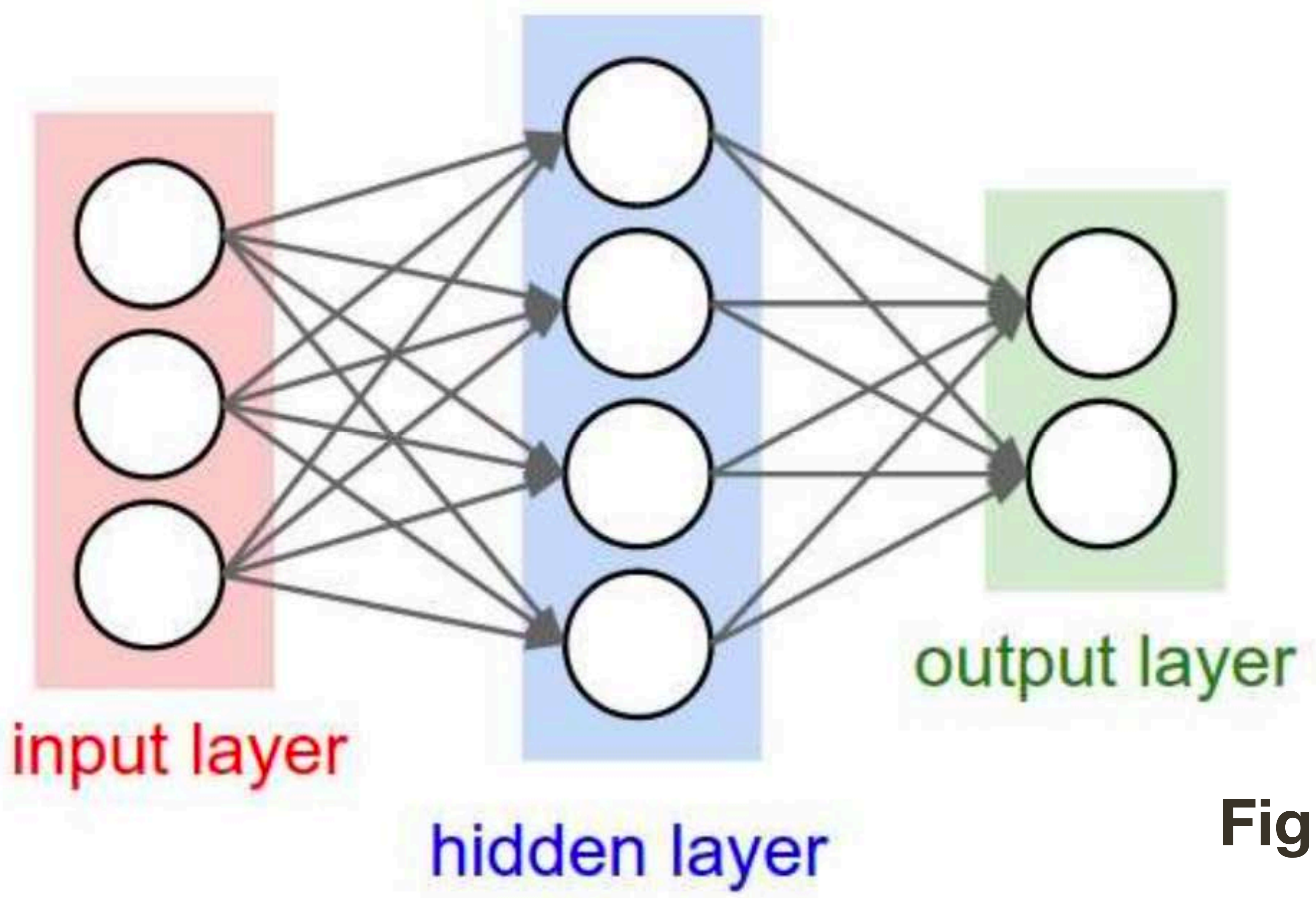
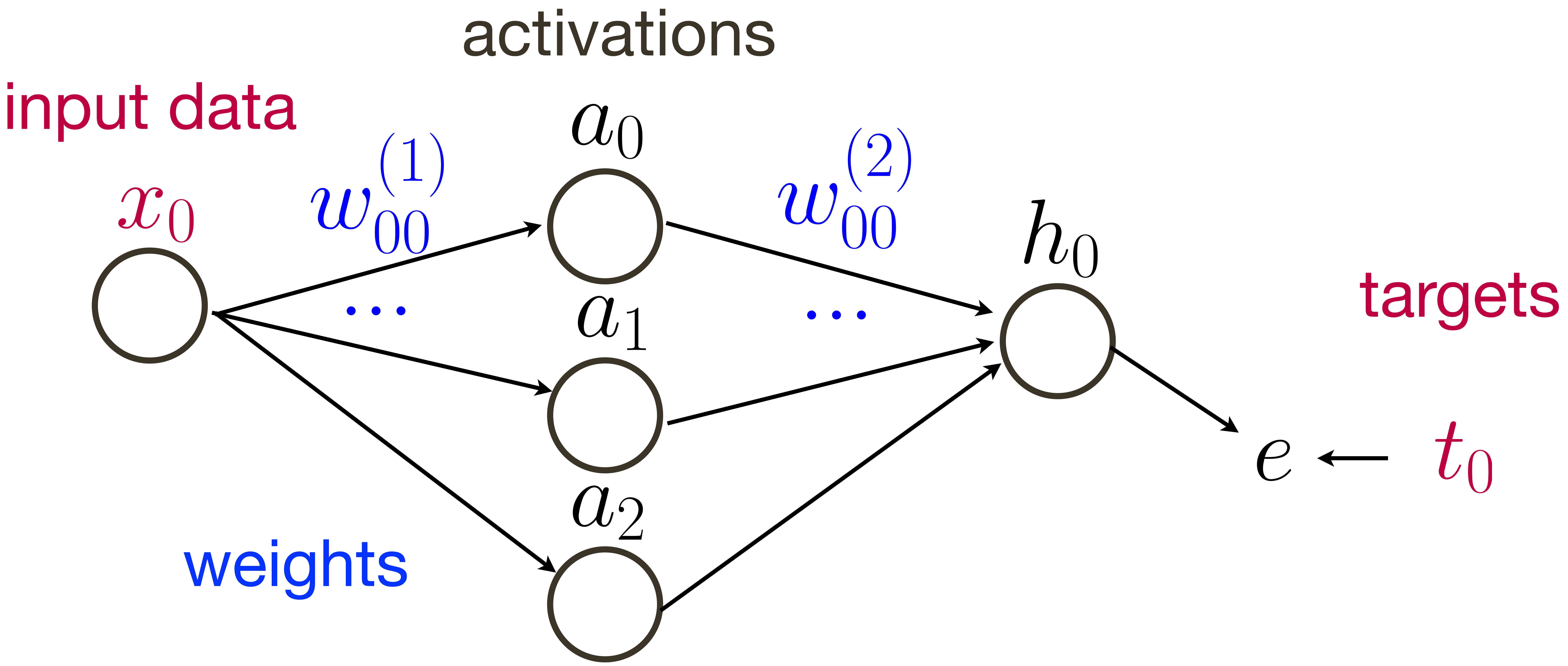


Figure credit: Fei-Fei and Karpathy

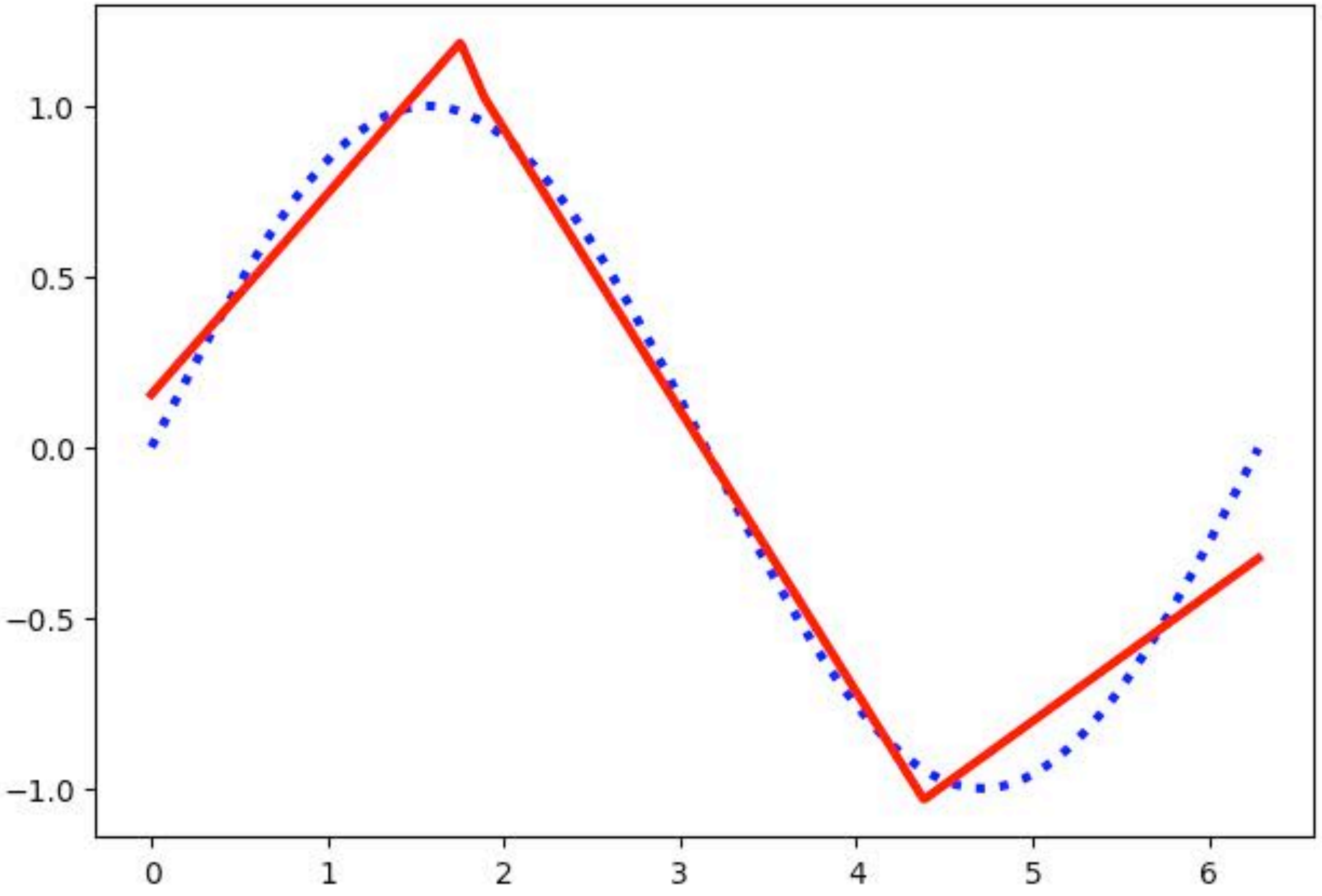
2-Layer **Neural** Network

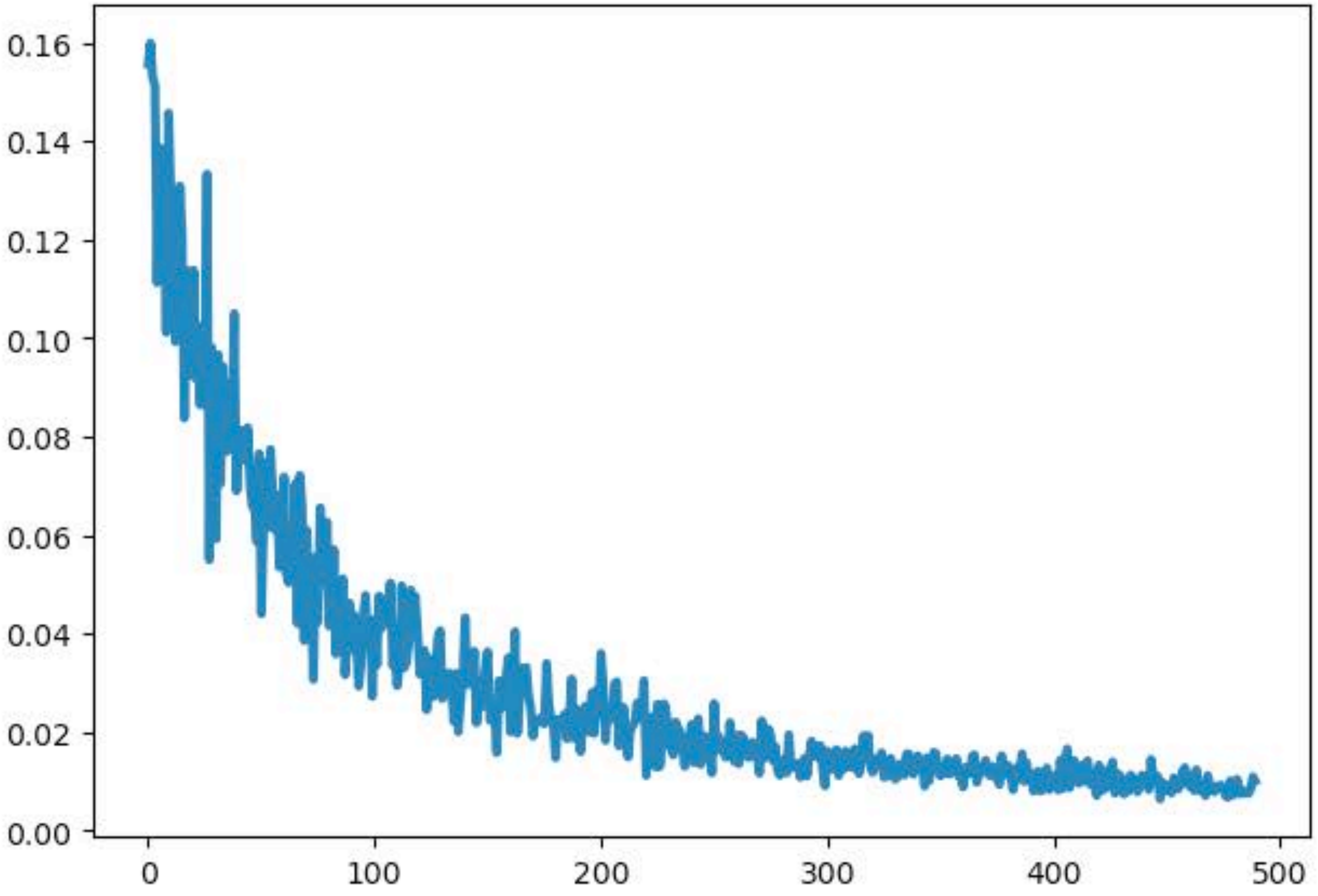
19.4

2-Layer **Neural** Network — n hidden, 1 input/output



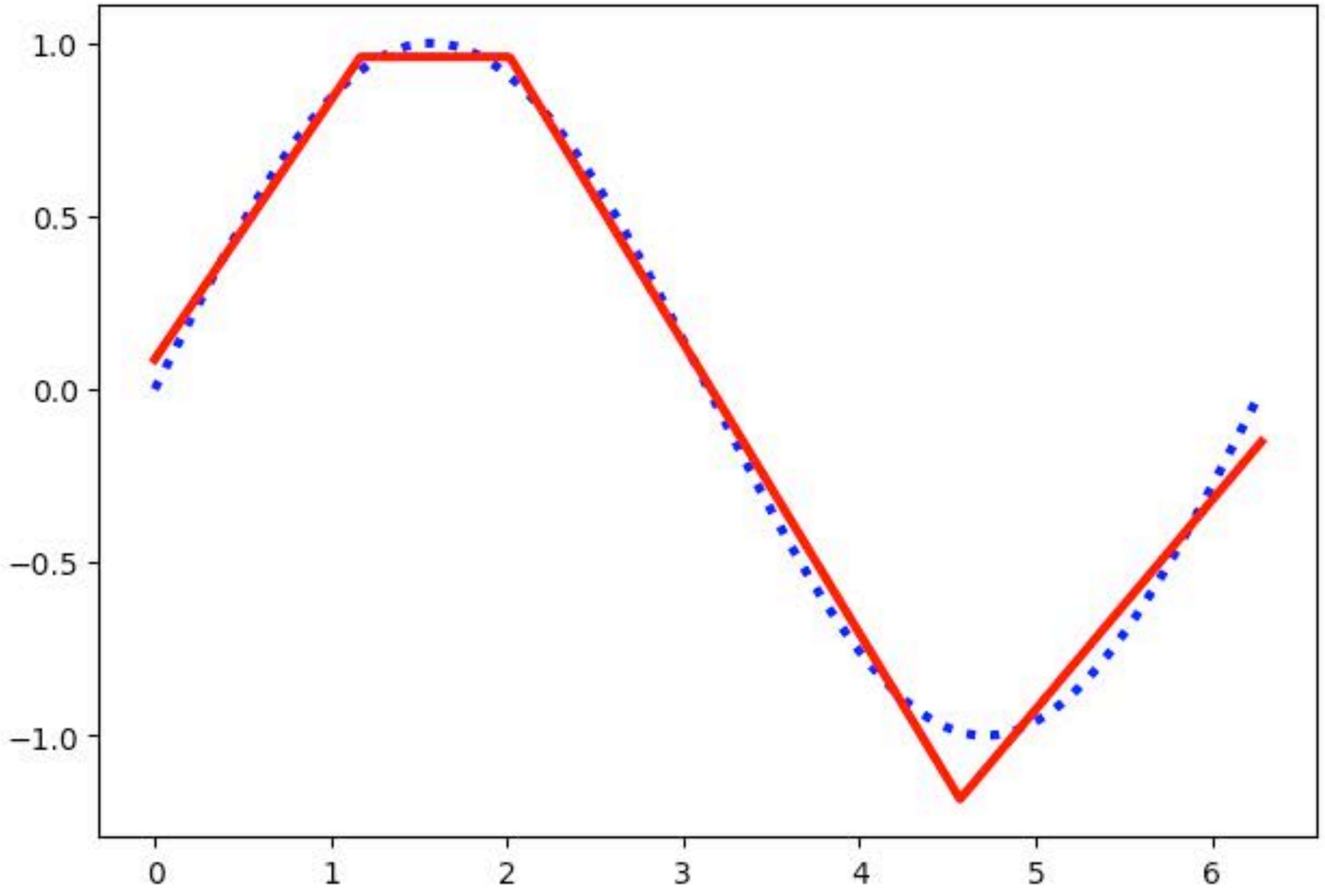
2-Layer **Neural** Network — n hidden, 1 input/output

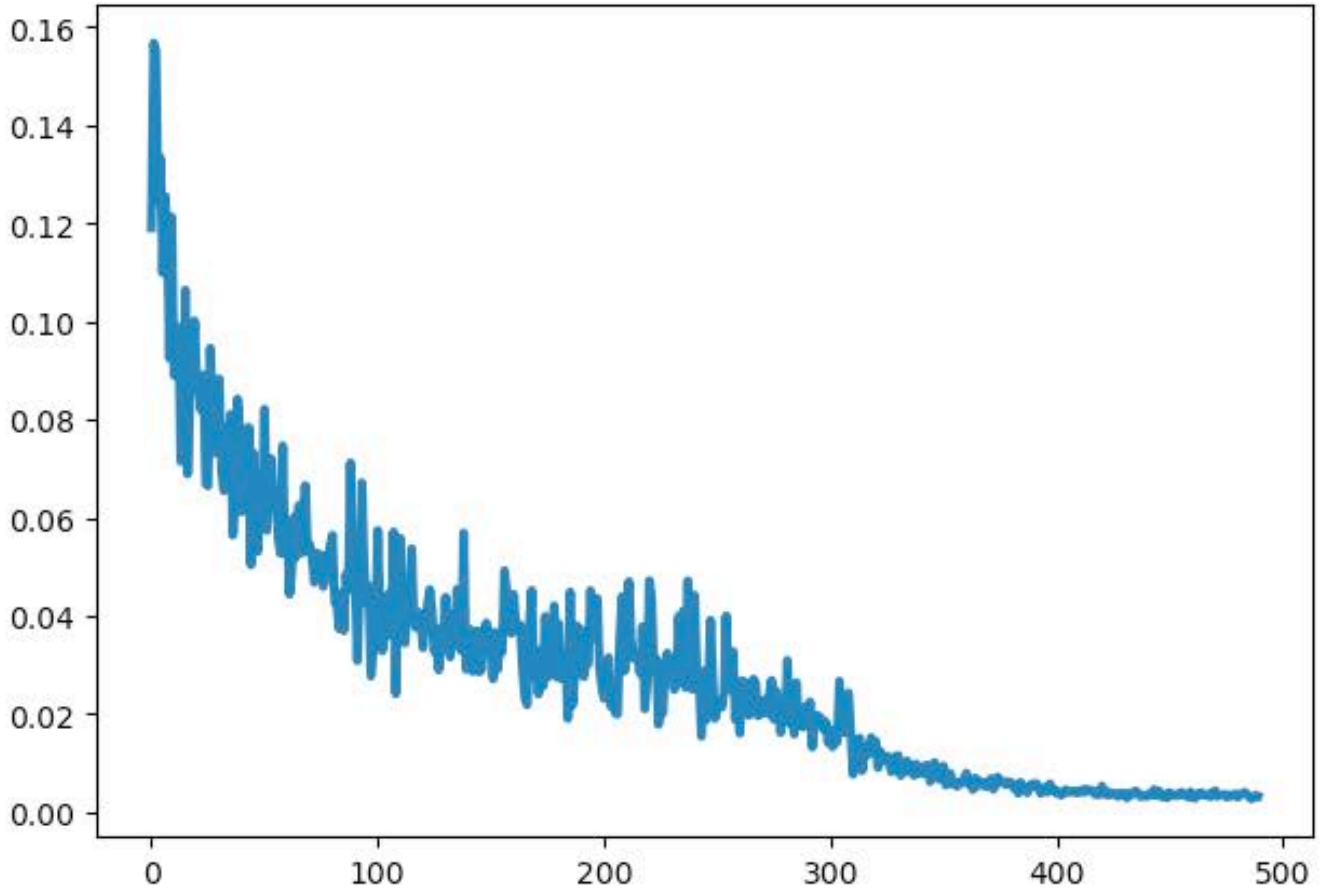




3 hidden units

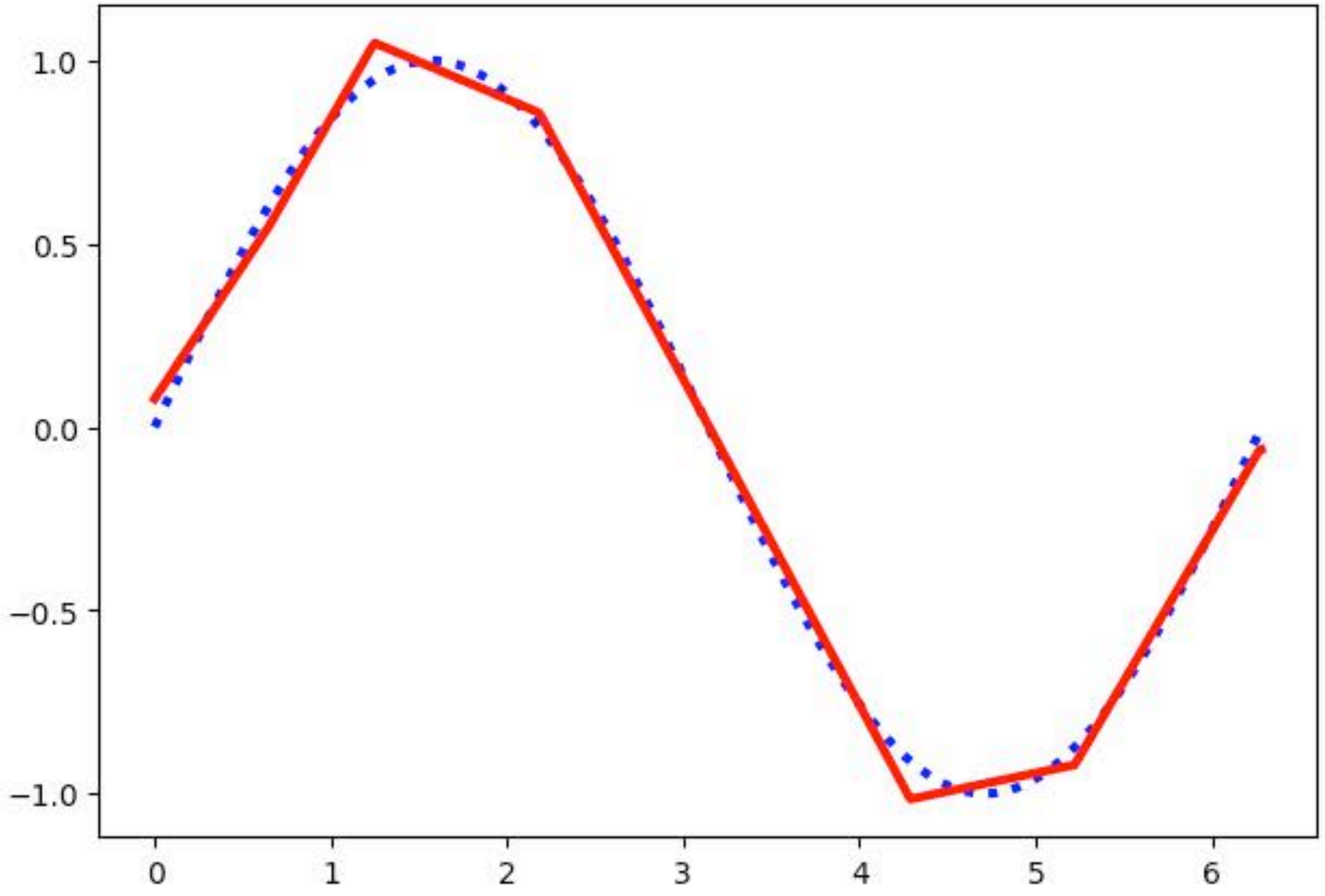
2-Layer **Neural** Network — n hidden, 1 input/output

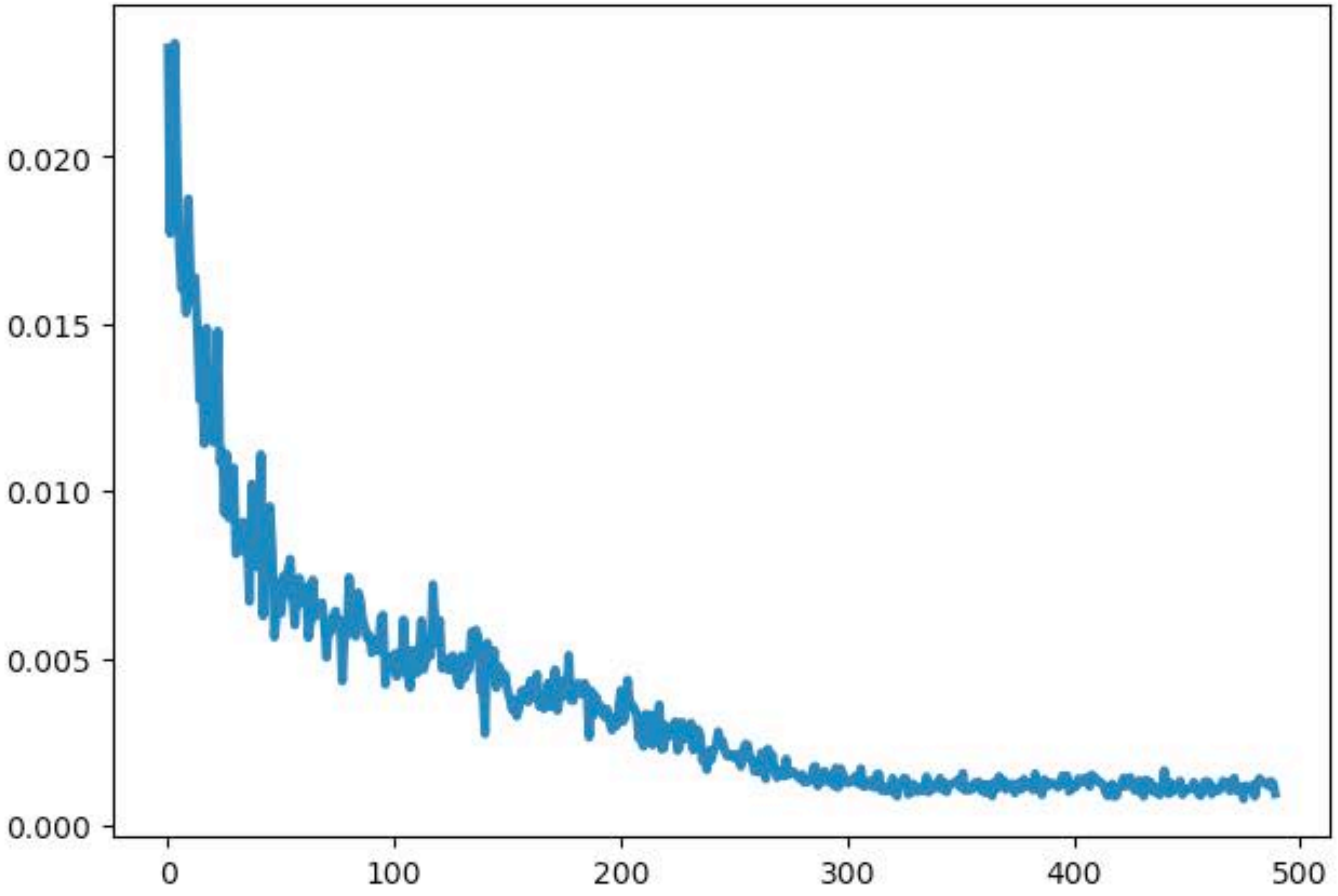




4 hidden units

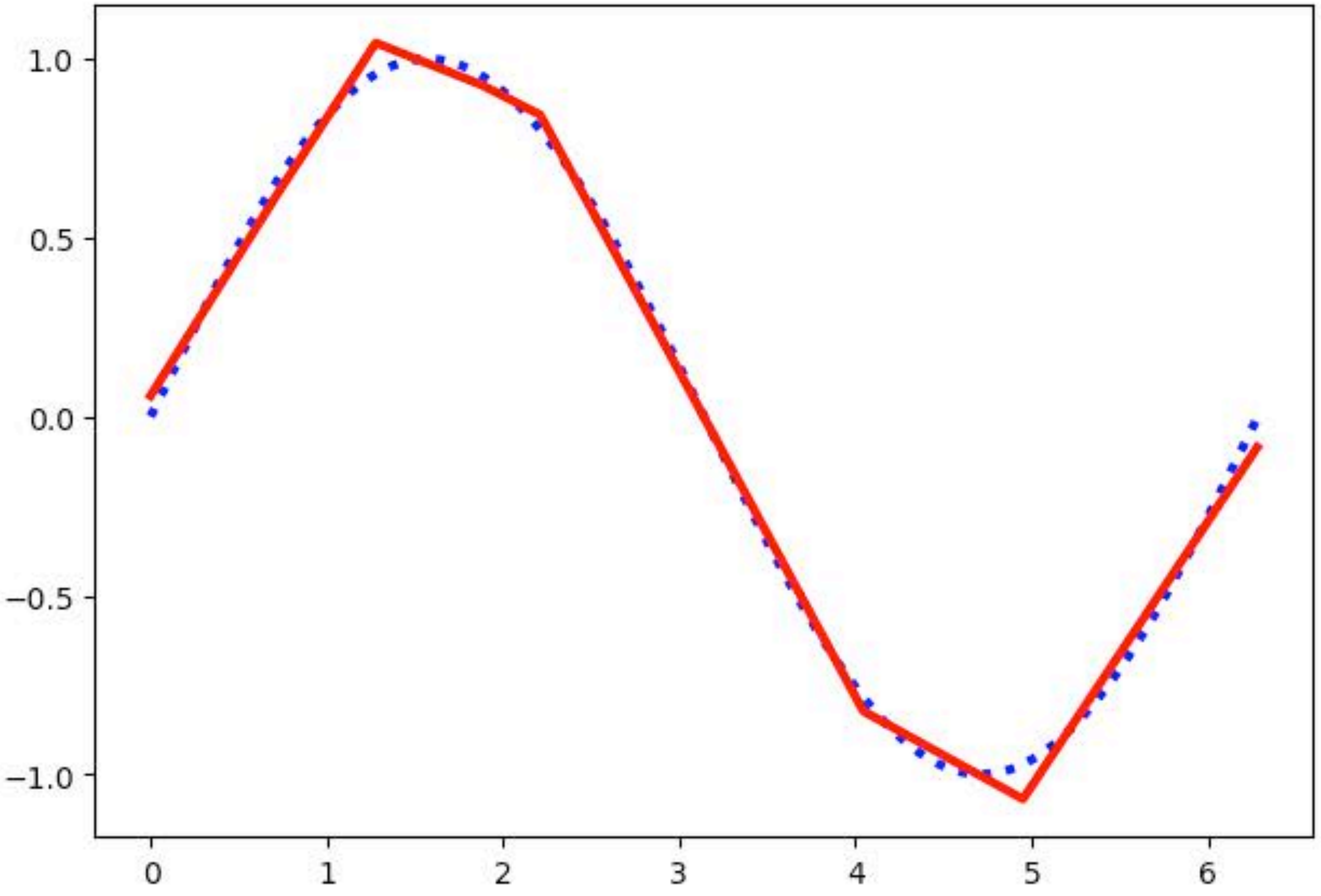
2-Layer **Neural** Network — n hidden, 1 input/output

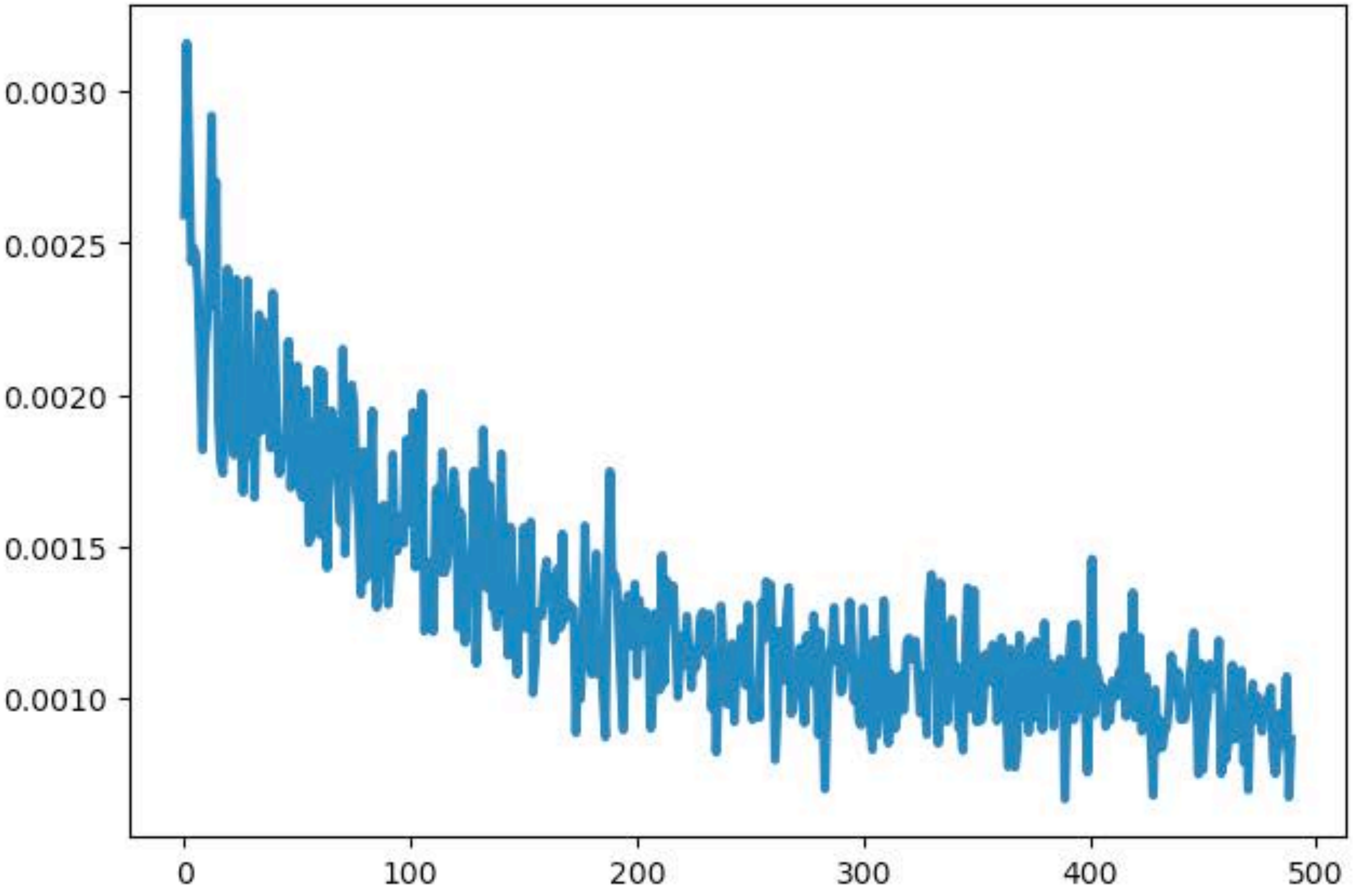




6 hidden units

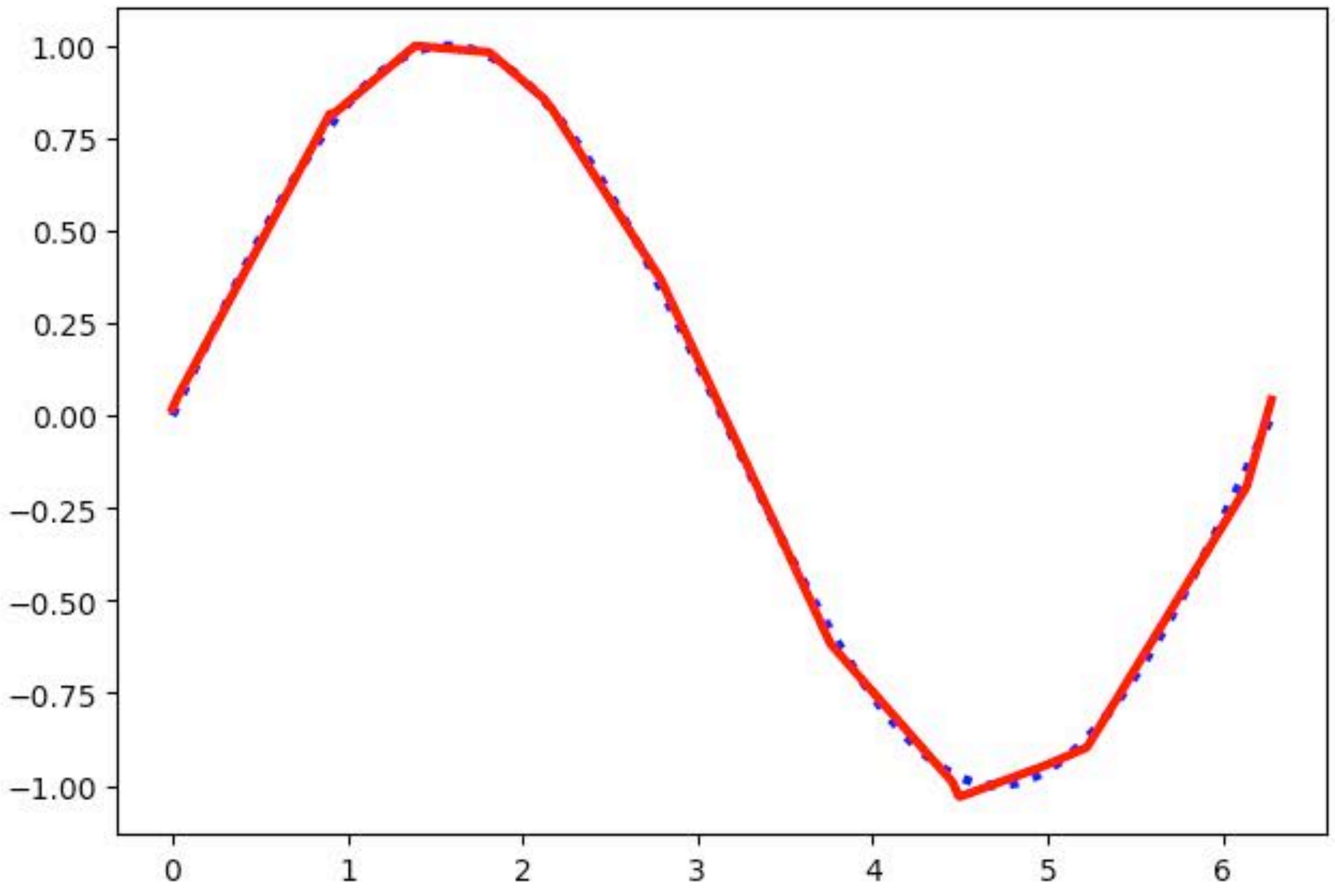
2-Layer **Neural** Network — n hidden, 1 input/output

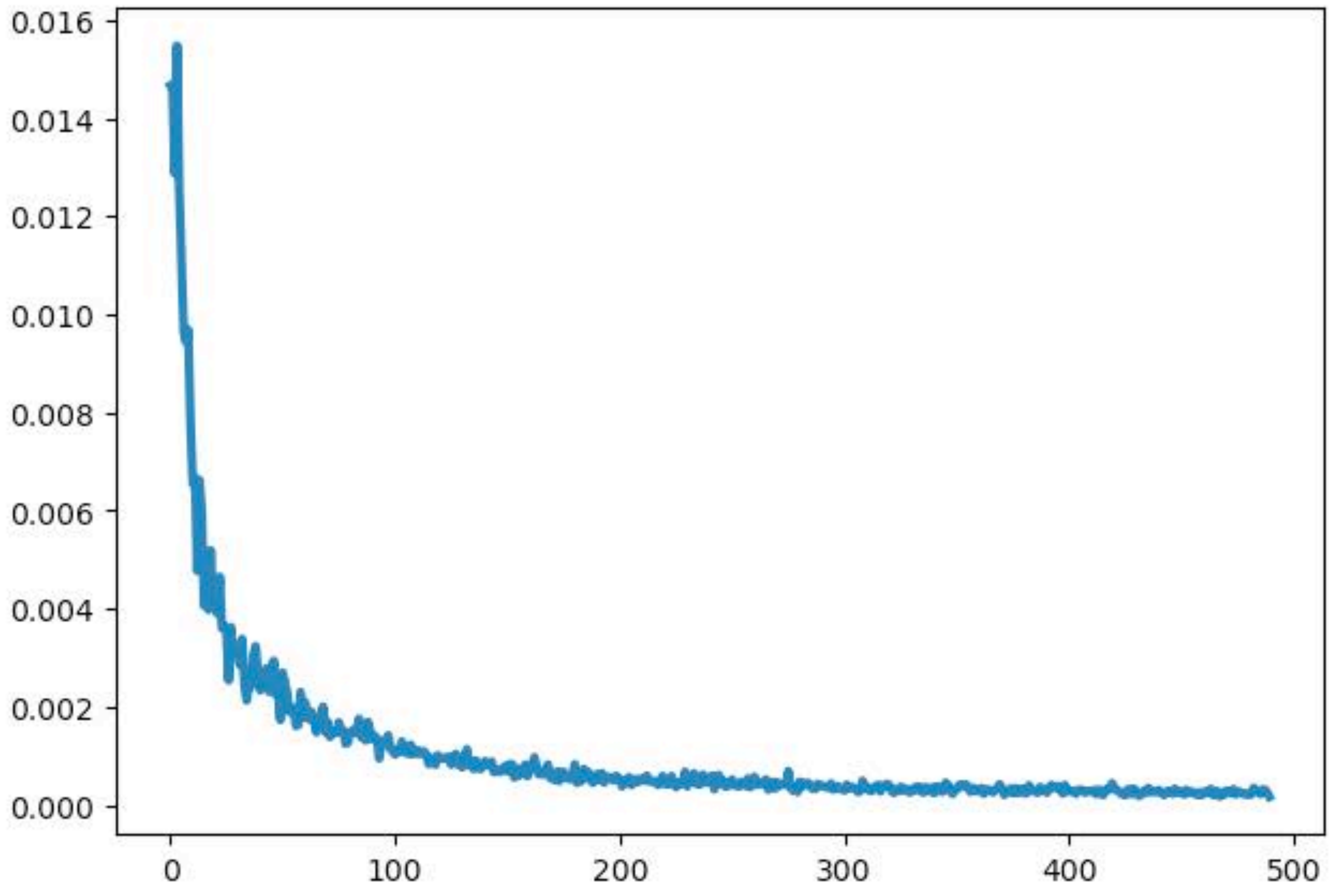




8 hidden units

2-Layer **Neural** Network — n hidden, 1 input/output





20 hidden units

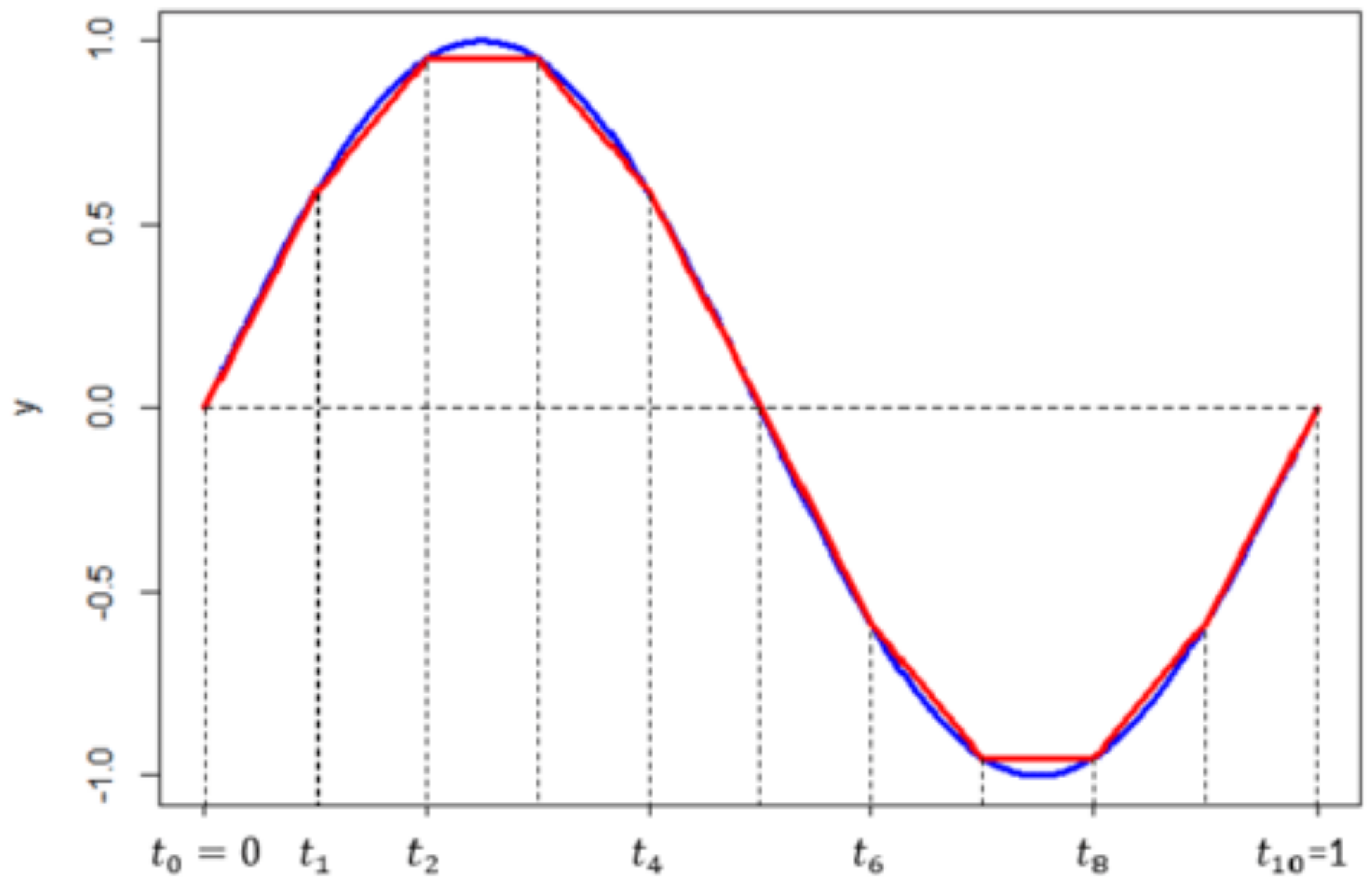
Neural Network as Universal Approximator

Non-linear activation is required to provably make the Neural Net a **universal function approximator**

Intuition: with ReLU activation, we effectively get a linear spline approximation to any function.

Optimization of neural net parameters = finding slops and transitions of linear pieces

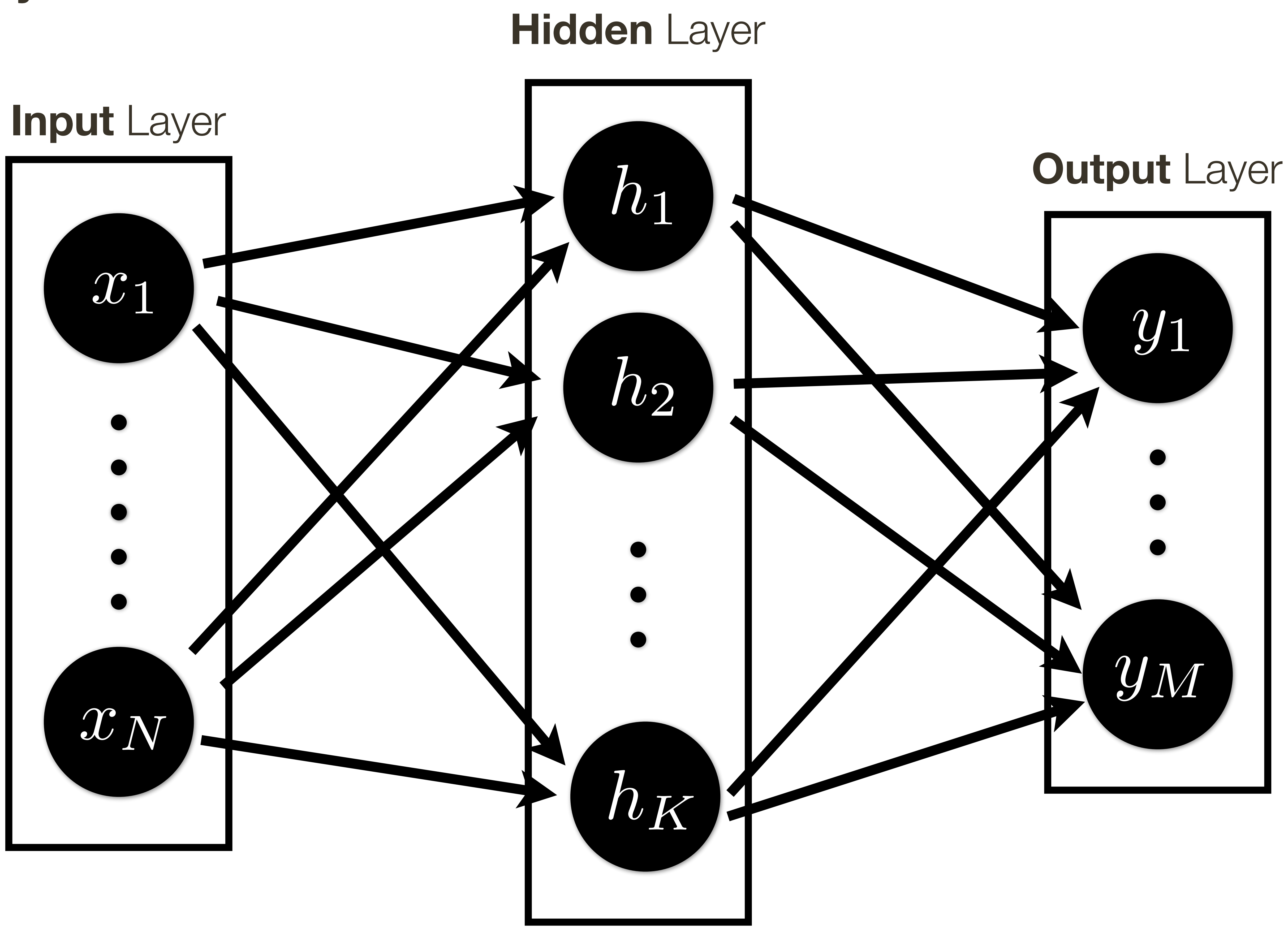
The quality of approximation depends on the number of linear segments



Light Theory: Neural Network as Universal Approximator

Universal Approximation Theorem: Single hidden layer can approximate any continuous function with compact support to arbitrary accuracy, when the width goes to infinity.

[Hornik *et al*., 1989]



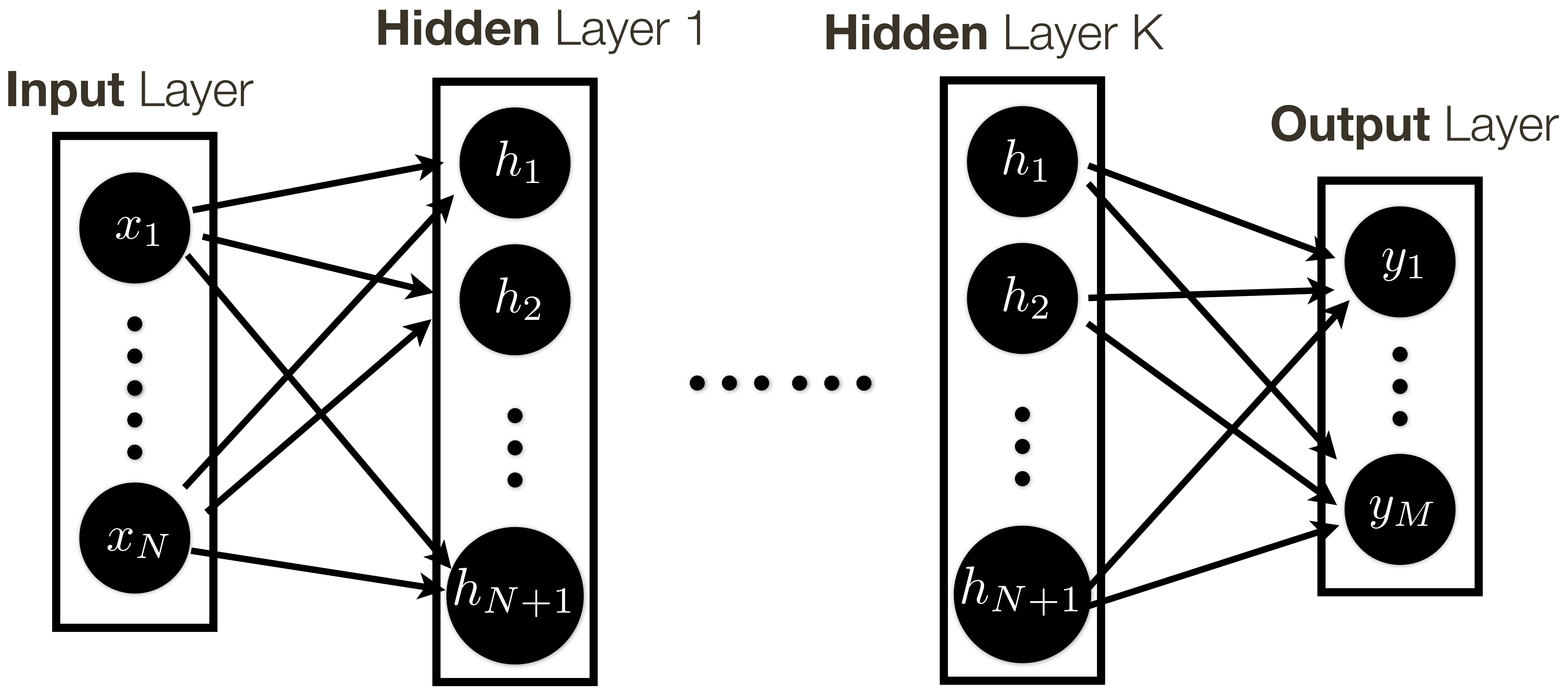
Light Theory: Neural Network as Universal Approximator

Universal Approximation Theorem: Single hidden layer can approximate any continuous function with compact support to arbitrary accuracy, when the width goes to infinity.

[Hornik *et al*., 1989]

Universal Approximation Theorem (revised): A network of infinite depth with a hidden layer of size $d + 1$ neurons, where d is the dimension of the input space, can approximate any continuous function.

[Lu *et al*., NIPS 2017]



Light Theory: Neural Network as Universal Approximator

Universal Approximation Theorem: Single hidden layer can approximate any continuous function with compact support to arbitrary accuracy, when the width goes to infinity.

[Hornik *et al*., 1989]

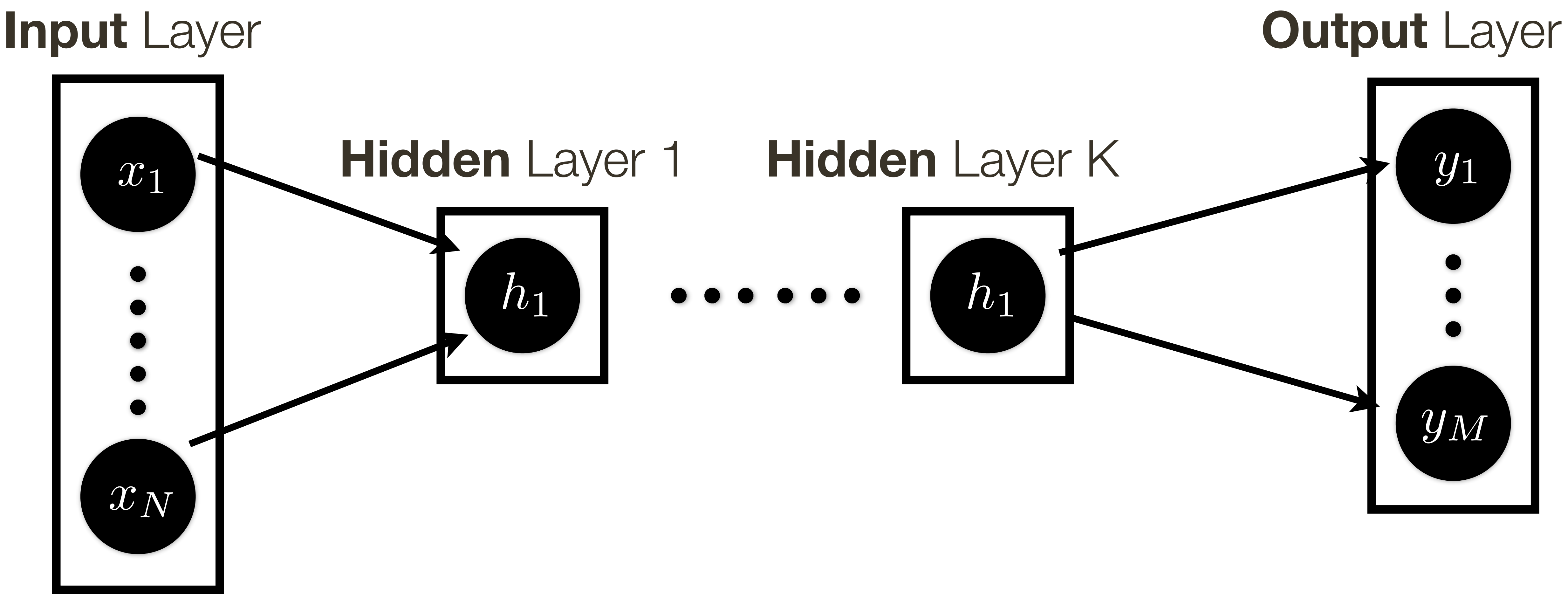
Universal Approximation Theorem (revised): A network of infinite depth with a hidden layer of size $d+1$ neurons, where d is the dimension of the input space, can approximate any continuous function.

[Lu *et al*., NIPS 2017]

Universal Approximation Theorem (further revised): ResNet with a single hidden unit and infinite depth can approximate any continuous function.

[Lin and Jegelka, NIPS 2018]

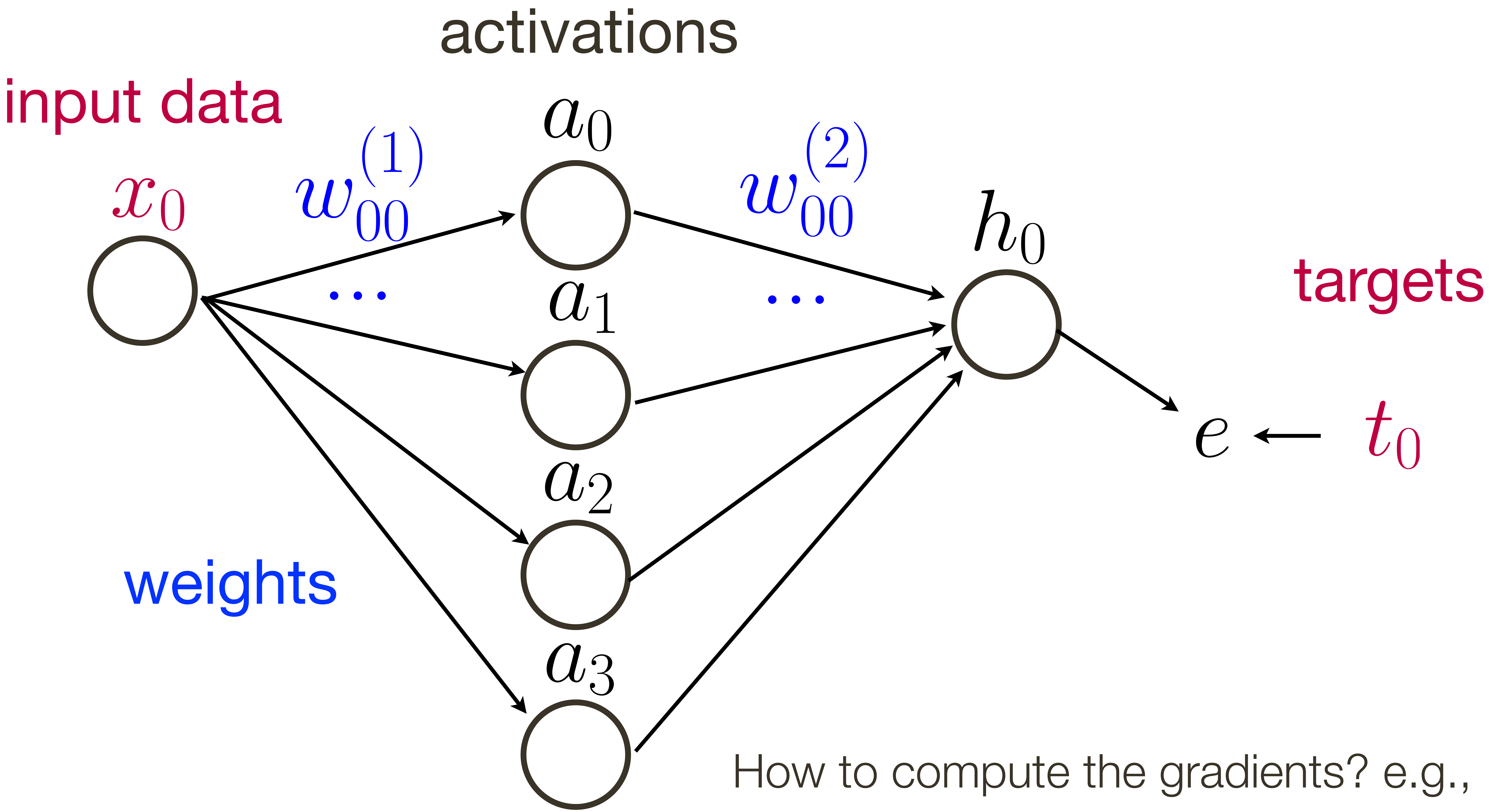
Light Theory: Neural Network as Universal Approximator



Universal Approximation Theorem (further revised): ResNet with a single hidden unit and infinite depth can approximate any continuous function.

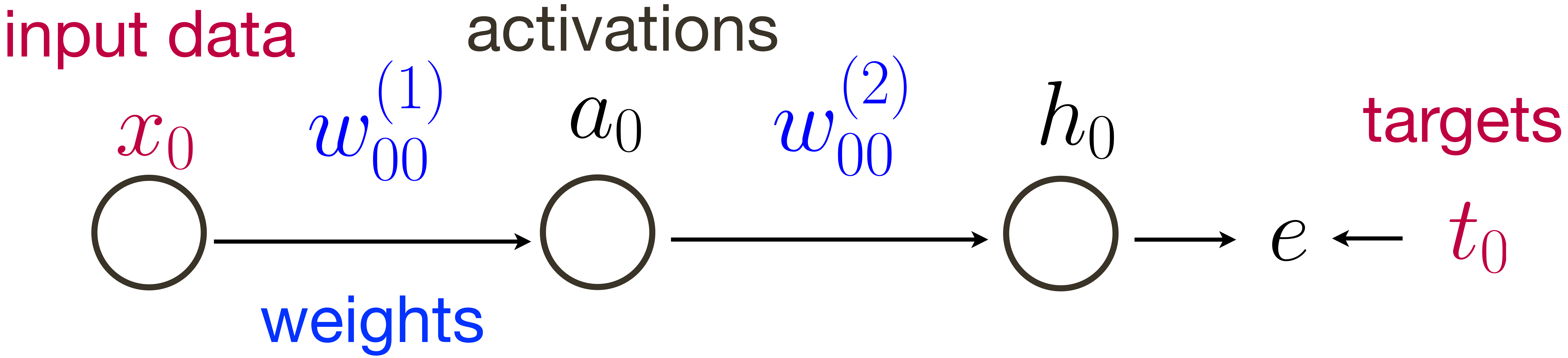
[Lin and Jegelka, NIPS 2018]

2-Layer **Neural** Network — n hidden, 1 input/output



How to compute the gradients? e.g., $\frac{\partial e}{\partial w_{00}^{(1)}}$

2-Layer **Neural** Network — 1 hidden, 1 input/output



2-Layer **Neural** Network — 1 hidden, 1 input/output

$$y = w_2(\max(0, w_1 x + b_1)) + b_2 \qquad L = (y - t)^2$$

Optimise by **gradient descent**

$$\begin{bmatrix} w_1 \\ b_1 \\ w_2 \\ b_2 \end{bmatrix} \to \begin{bmatrix} w_1 \\ b_1 \\ w_2 \\ b_2 \end{bmatrix} - \alpha \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial b_1} \\ \frac{\partial L}{\partial w_2} \\ \frac{\partial L}{\partial b_2} \end{bmatrix}$$

19.5 How to compute the gradients? e.g., $\frac{\partial L}{\partial w_1}$

Neural Networks

(**Before**) Linear score function: $$f = Wx$$

$$x \in \mathbb{R}^D, W \in \mathbb{R}^{C \times D}$$

Neural Networks

(**Before**) Linear score function: $$f = Wx$$

(**Now**) 2-layer Neural Network $$f = W_2 \max(0, W_1 x)$$

$$W_2 \in \mathbb{R}^{C \times H} \quad W_1 \in \mathbb{R}^{H \times D} \quad x \in \mathbb{R}^D$$

(In practice we will usually add a learnable bias at each layer as well)

Neural Networks

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

or 3-layer Neural Network

$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$

$$W_3 \in \mathbb{R}^{C \times H_2} \quad W_2 \in \mathbb{R}^{H_2 \times H_1} \quad W_1 \in \mathbb{R}^{H_1 \times D} \quad x \in \mathbb{R}^D$$

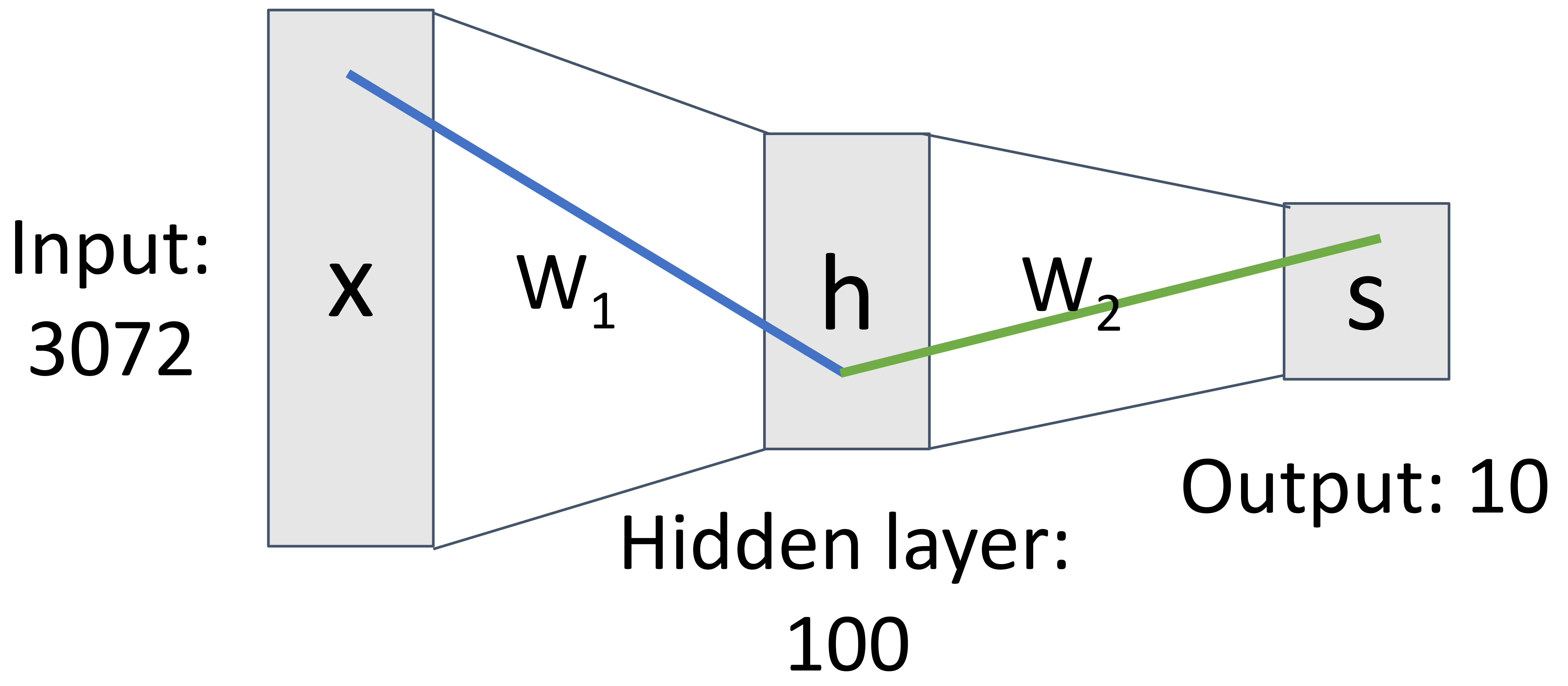
(In practice we will usually add a learnable bias at each layer as well)

Neural Networks

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

Element (i, j) of W_1 gives the effect on h_i from x_j



Element (i, j) of W_2 gives the effect on s_i from h_j

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

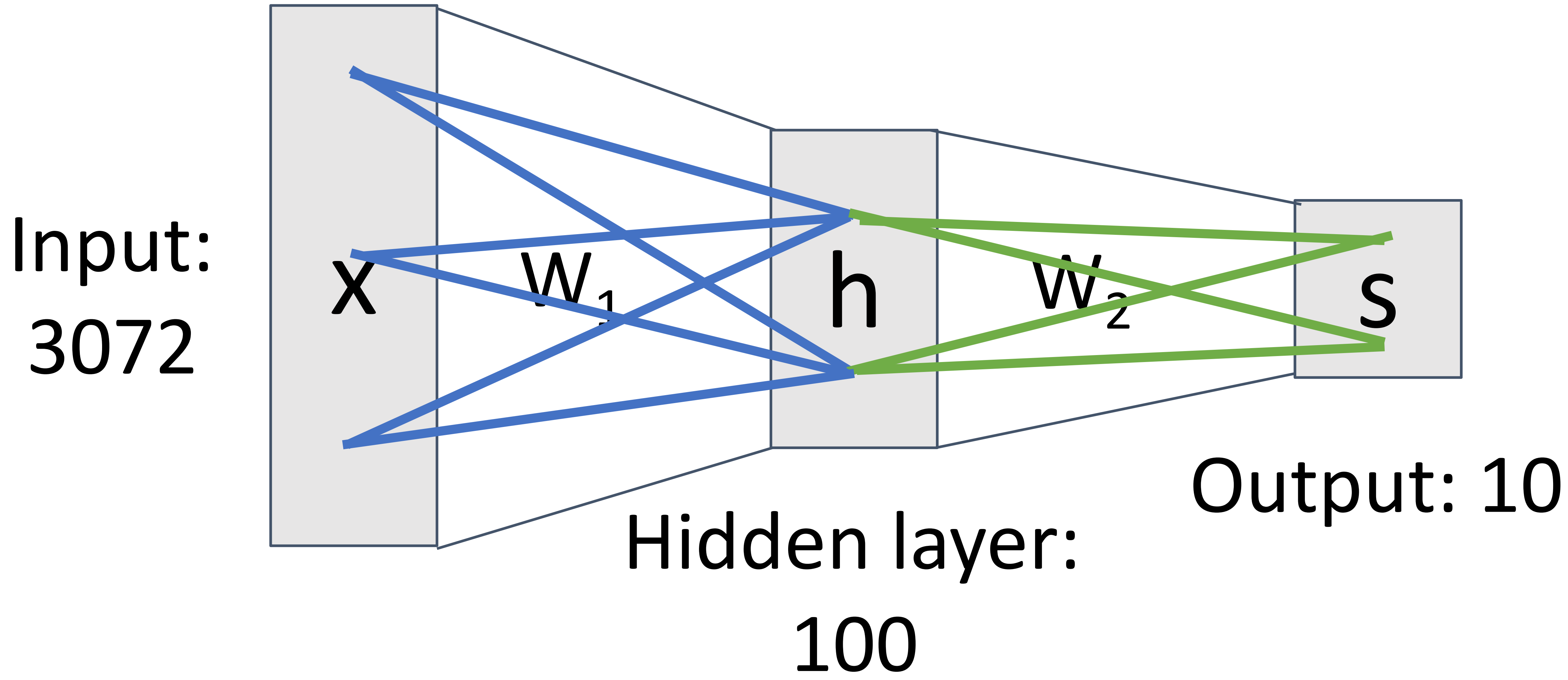
Neural Networks

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

Element (i, j) of W_1 gives the effect on h_i from x_j

All elements of x affect all elements of h



Element (i, j) of W_2 gives the effect on s_i from h_j

All elements of h affect all elements of s

Fully-connected neural network

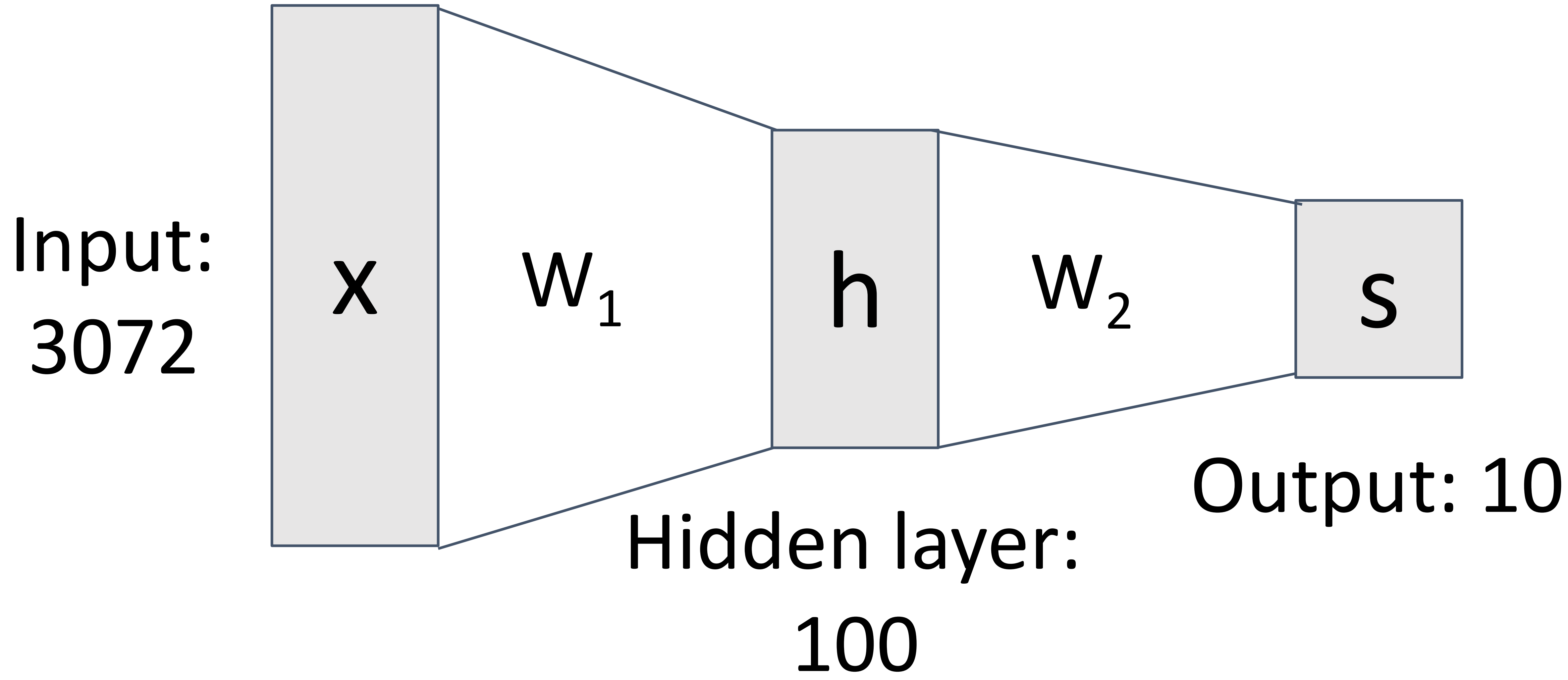
Also "Multi-Layer Perceptron" (MLP)

Neural Networks

(**Before**) Linear score function:

(**Now**) 2-layer Neural Network

Linear classifier: One template per class



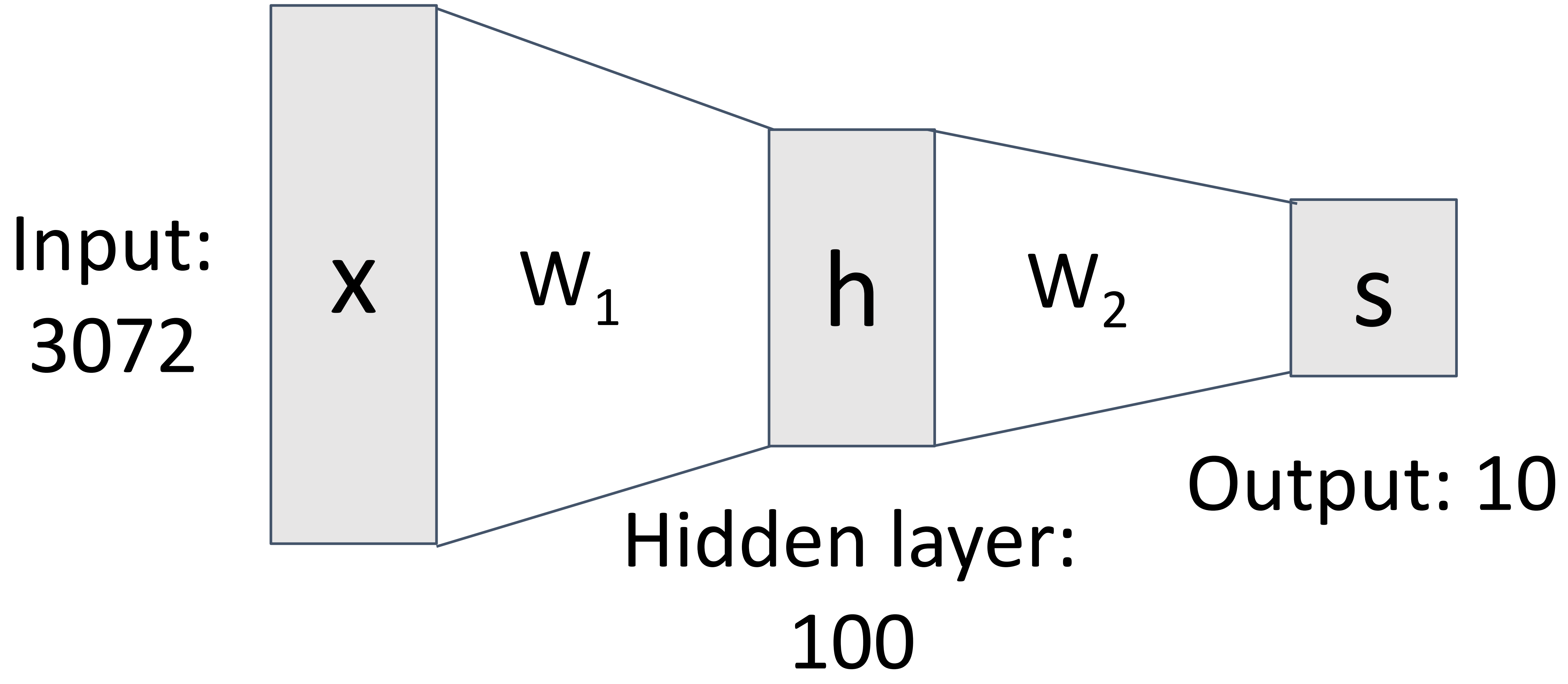
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

Neural net: first layer is bank of templates;
Second layer recombines templates

(**Before**) Linear score function:

(**Now**) 2-layer Neural Network



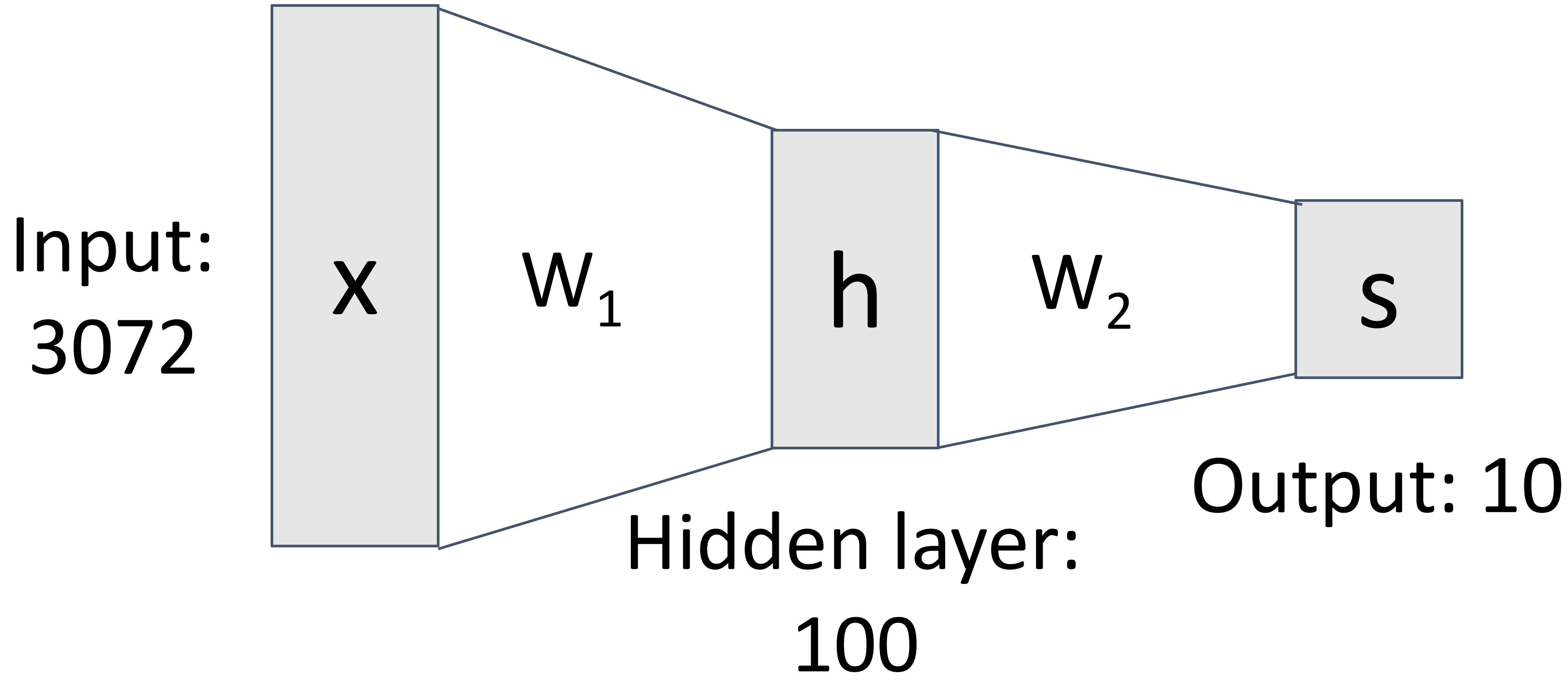
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

Can use different templates to cover multiple modes of a class!

(**Before**) Linear score function:

(**Now**) 2-layer Neural Network



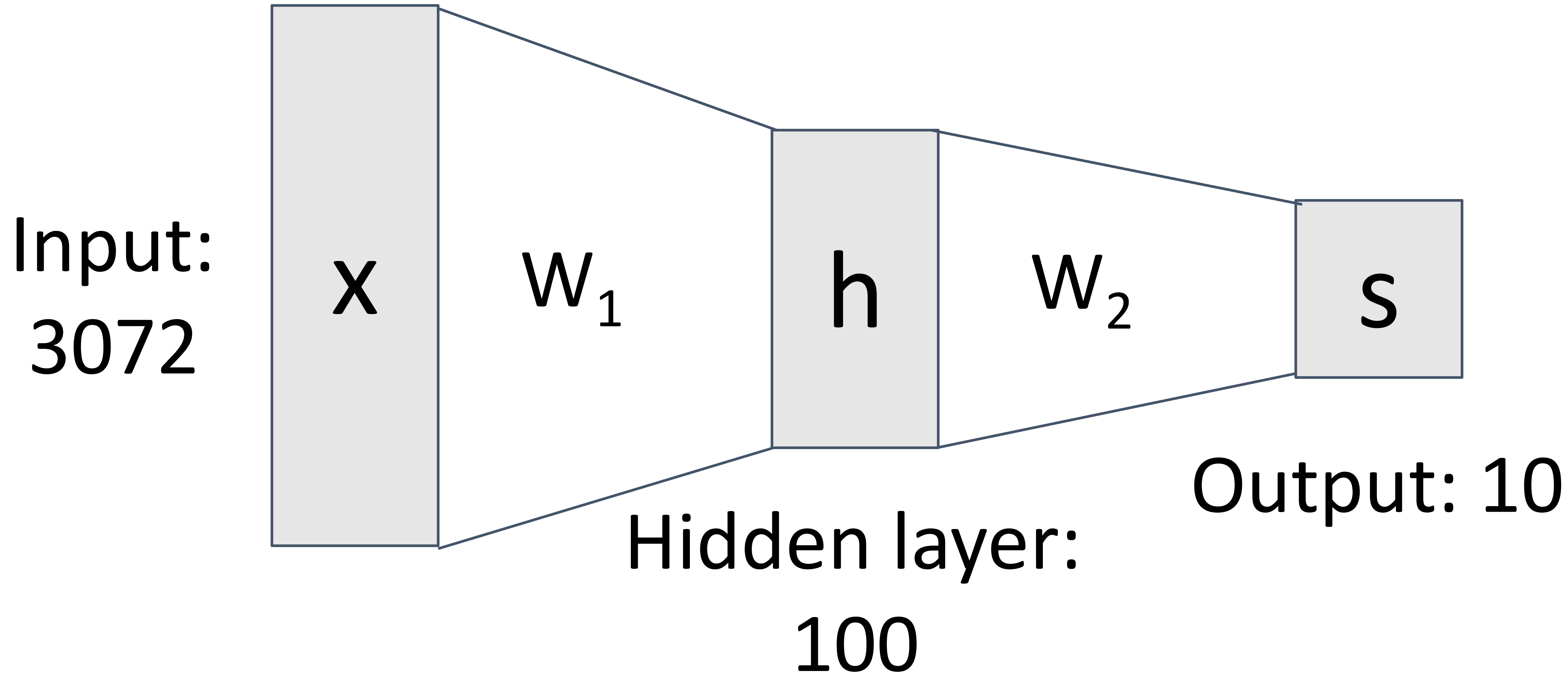
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

"Distributed representation":
Most templates not interpretable!

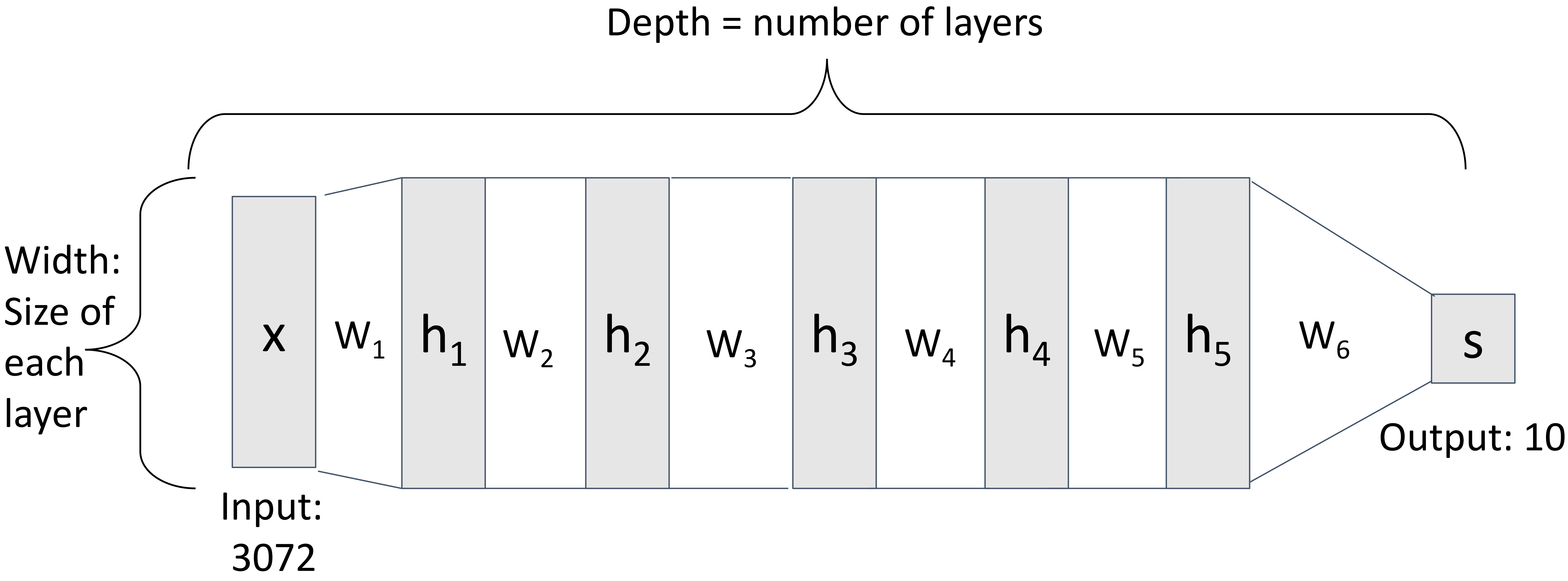
(**Before**) Linear score function:

(**Now**) 2-layer Neural Network



$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Deep Neural Networks



$$s = W_6 \max(0, W_6 \max(0, W_5 \max(0, W_4 \max(0, W_3 \max(0, W_2 \max(0, W_1 x))))))$$

2-Layer **Neural** Network — 1 hidden, 1 input/output

$$y = w_2(\max(0, w_1 x + b_1)) + b_2 \qquad\qquad L = (y - t)^2$$

Optimise by **gradient descent**

$$\begin{bmatrix} w_1 \\ b_1 \\ w_2 \\ b_2 \end{bmatrix} \rightarrow \begin{bmatrix} w_1 \\ b_1 \\ w_2 \\ b_2 \end{bmatrix} - \alpha \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial b_1} \\ \frac{\partial L}{\partial w_2} \\ \frac{\partial L}{\partial b_2} \end{bmatrix}$$

19.5 How to compute the gradients? e.g., $\dfrac{\partial L}{\partial w_1}$

(Bad) Idea: Derive $\nabla_W L$ on paper

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

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$= \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1)$$

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda \sum_k W_k^2$$

$$= \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2$$

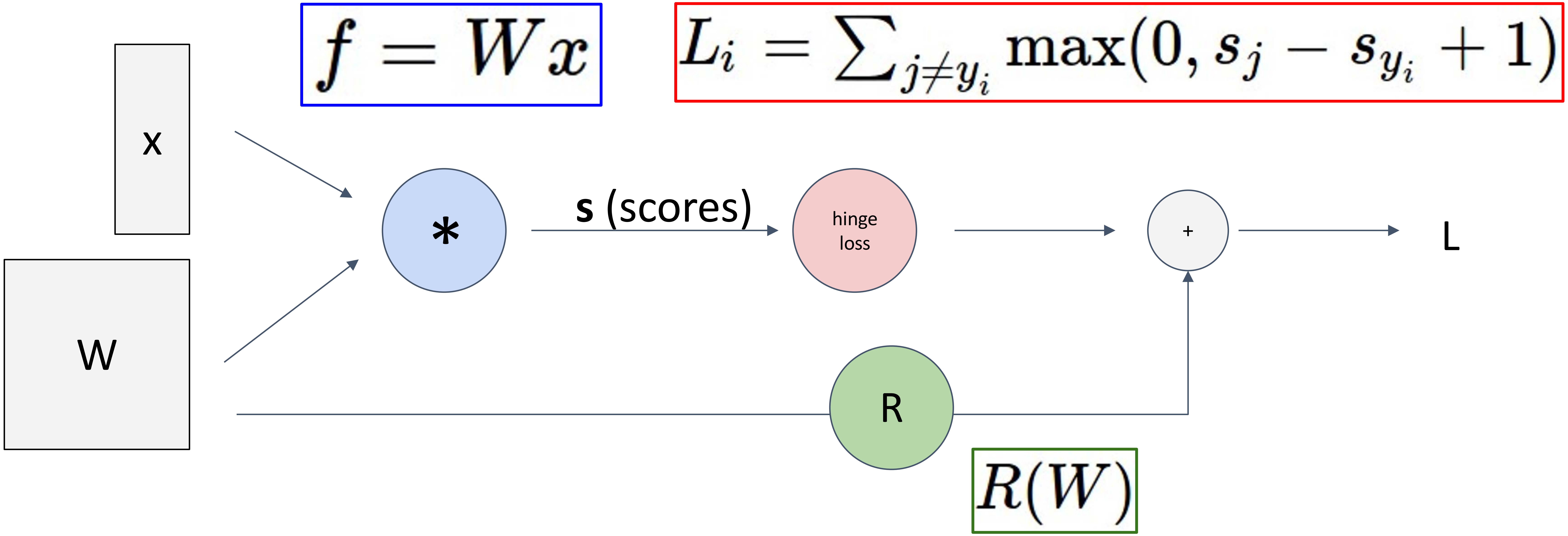
$$\nabla_W L = \nabla_W \left(\frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2 \right)$$

Problem: Very tedious: Lots of matrix calculus, need lots of paper

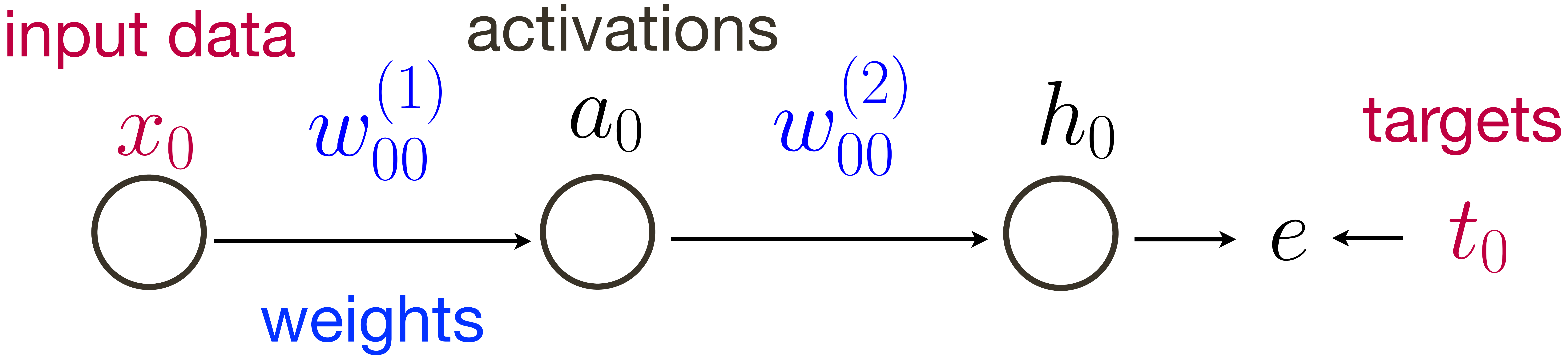
Problem: What if we want to change loss? E.g. use softmax instead of SVM? Need to re-derive from scratch. Not modular!

Problem: Not feasible for very complex models!

Better Idea: **Computational Graphs**

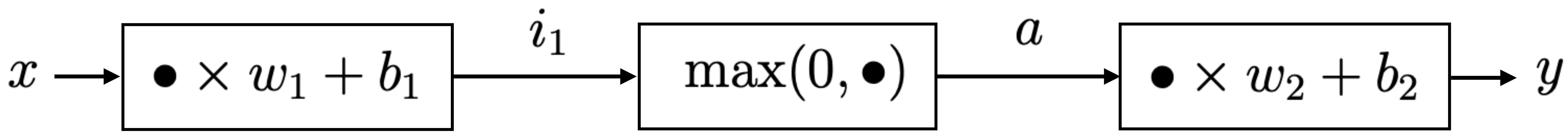


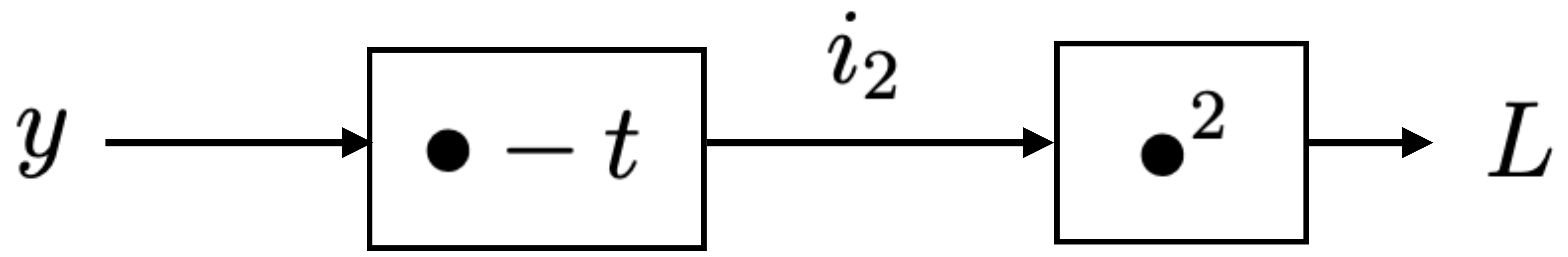
2-Layer **Neural** Network — 1 hidden, 1 input/output



2-Layer **Neural** Network — 1 hidden, 1 input/output

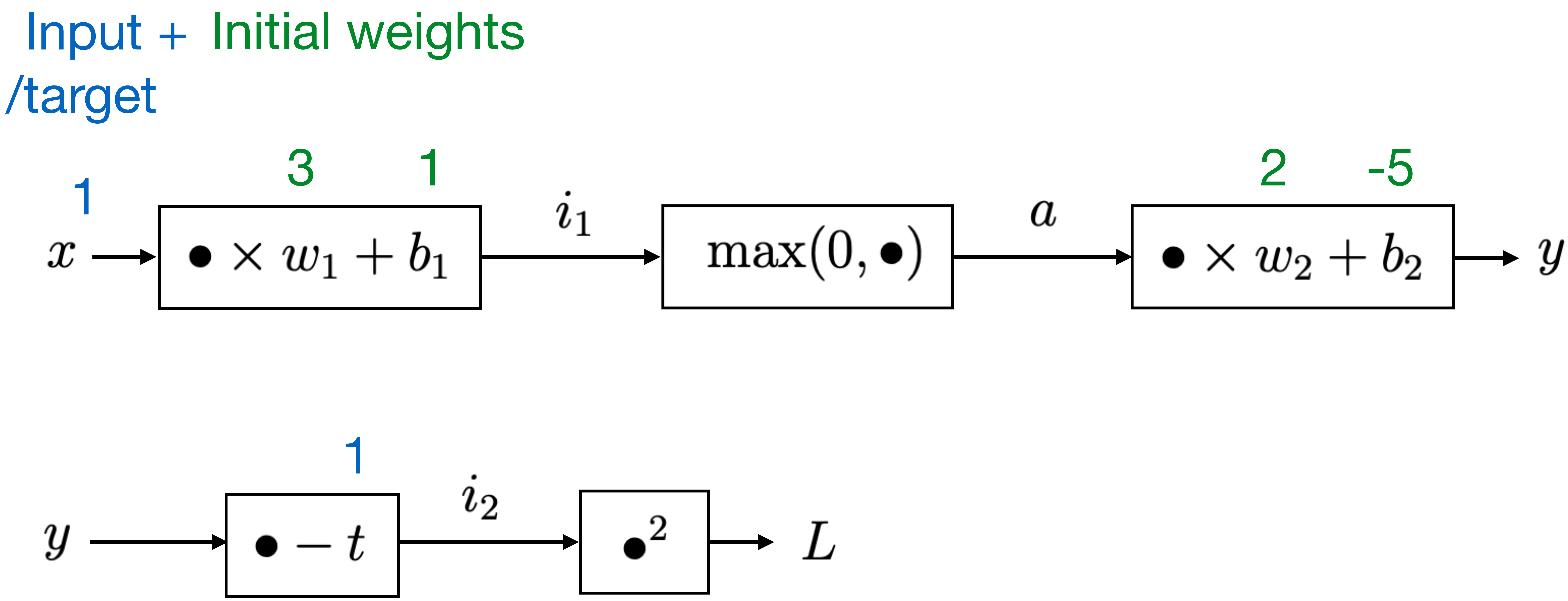
$$y = w_2(\max(0, w_1 x + b_1)) + b_2 \qquad L = (y - t)^2$$





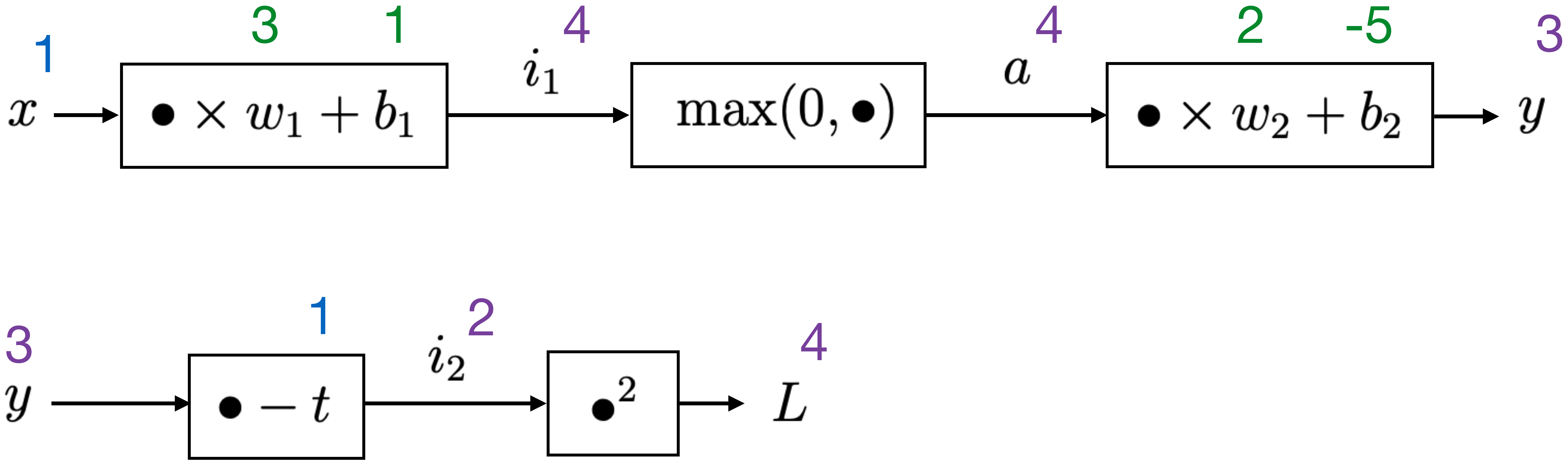
Alternative: build a **computational graph** to apply the **chain rule**

2-Layer **Neural** Network — 1 hidden, 1 input/output



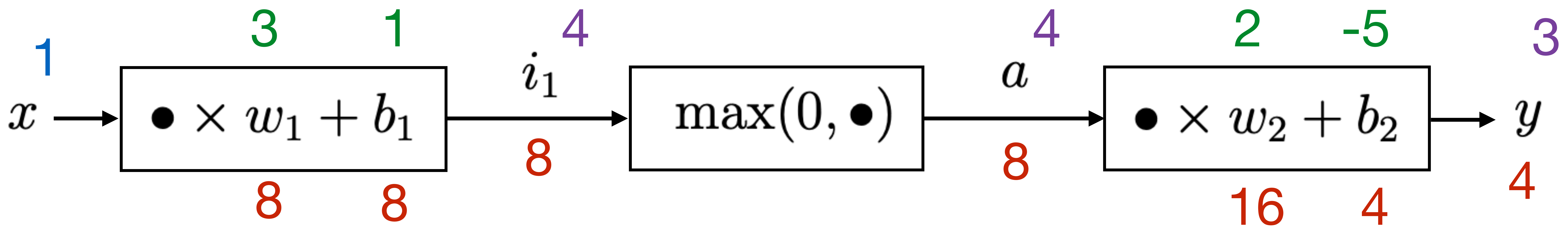
2-Layer **Neural** Network — 1 hidden, 1 input/output

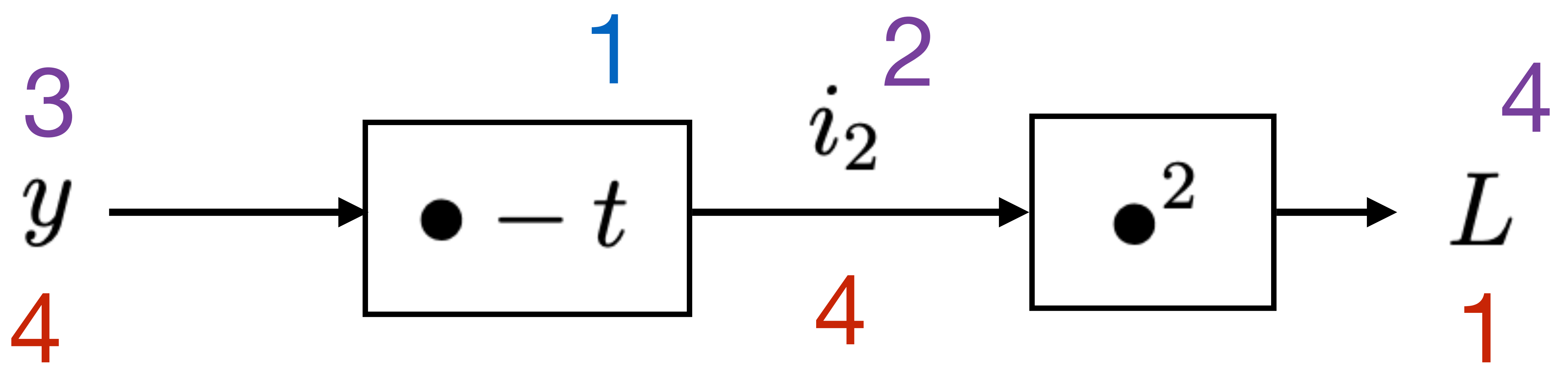
Input + /target Initial weights Forward pass



2-Layer **Neural** Network — 1 hidden, 1 input/output

Input + Initial weights /target   Forward pass   Backward pass $= \frac{\partial L}{\partial \bullet}$





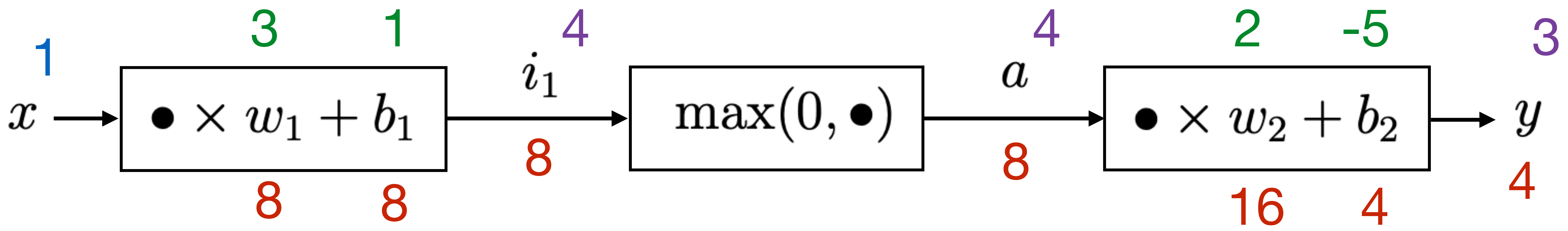
$$\frac{\partial L}{\partial L}$$

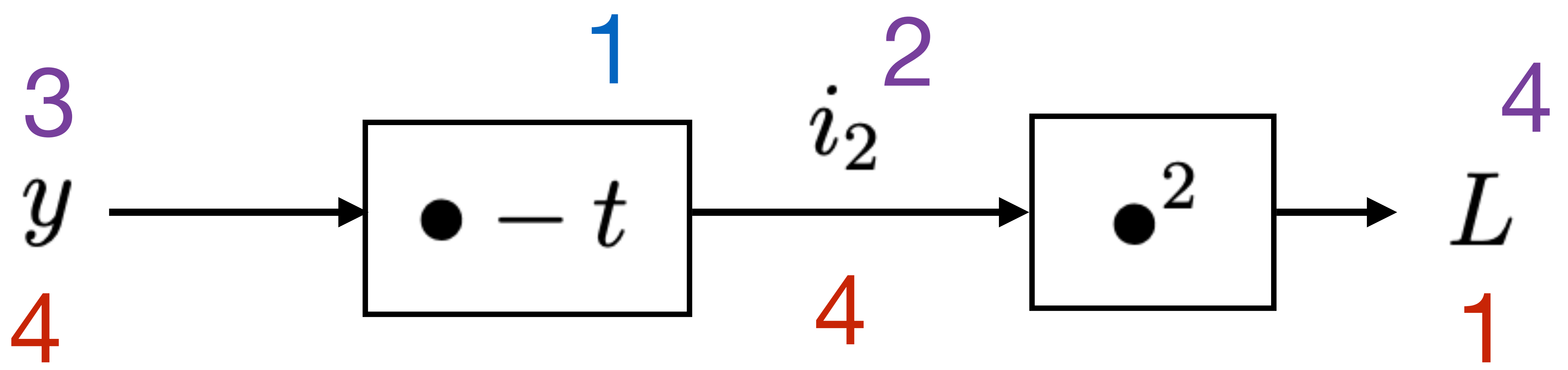
$$\frac{\partial L}{\partial i_2} = \frac{\partial f(i_2)}{\partial i_2} \frac{\partial L}{\partial f(i_2)} = \frac{\partial f(i_2)}{\partial i_2} \frac{\partial L}{\partial L} = \frac{\partial (i_2)^2}{\partial i_2} 1 = 2 i_2$$

$$\frac{\partial L}{\partial y} = \frac{\partial f(y)}{\partial y} \frac{\partial L}{\partial f(y)} = \frac{\partial f(y)}{\partial y} \frac{\partial L}{\partial i_2} = \frac{\partial f(y)}{\partial y} 4 = \frac{\partial y - t}{\partial y} 4 = 1 \times 4$$

2-Layer **Neural** Network — 1 hidden, 1 input/output

Input + Initial weights /target   Forward pass   Backward pass $= \frac{\partial L}{\partial \bullet}$

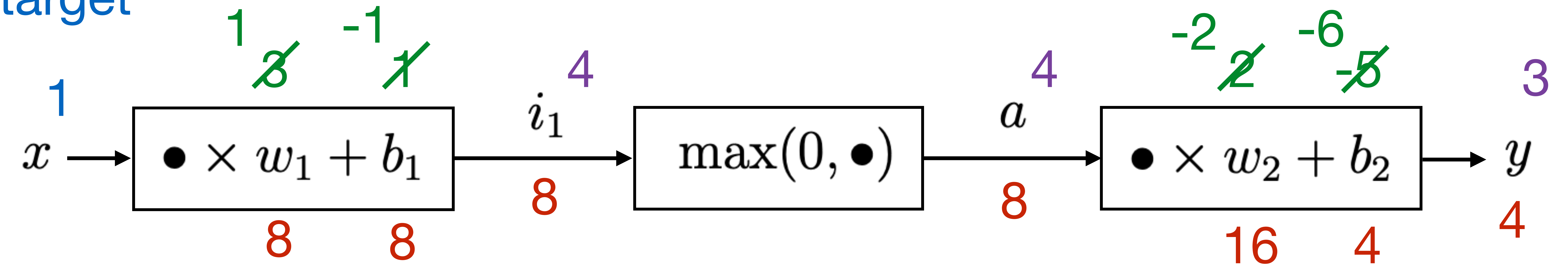


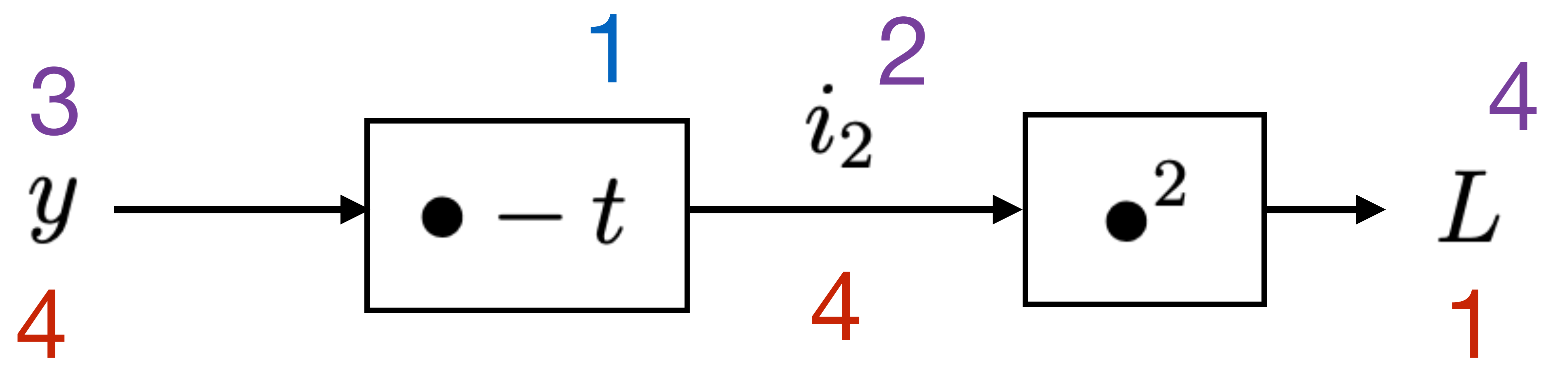


Gradient = $\begin{bmatrix} 8 \\ 8 \\ 16 \\ 4 \end{bmatrix}$

2-Layer **Neural** Network — 1 hidden, 1 input/output

Input + Initial weights /target   Forward pass   Backward pass $= \dfrac{\partial L}{\partial \bullet}$





Repeat: +Input/target, Forward, Backward, Update until convergence!

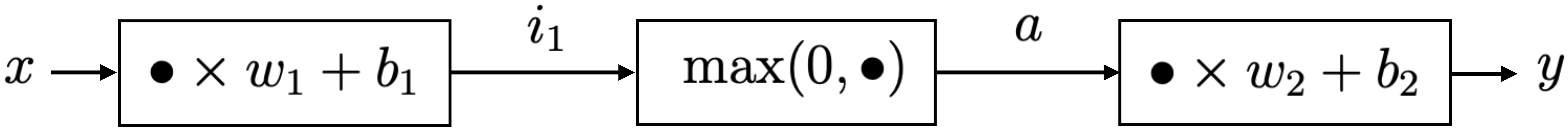
Gradient descent step

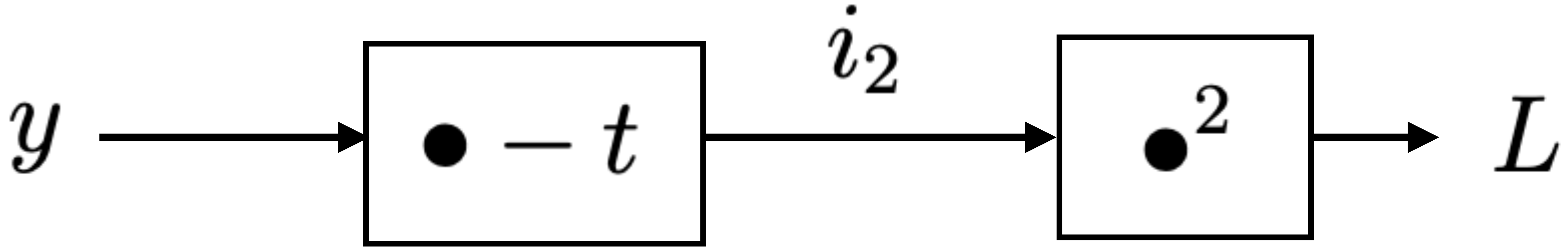
$$\begin{bmatrix} w_1 \\ b_1 \\ w_2 \\ b_2 \end{bmatrix} \rightarrow \begin{bmatrix} 3 \\ 1 \\ 2 \\ -5 \end{bmatrix} - \overset{1/4}{\alpha} \begin{bmatrix} 8 \\ 8 \\ 16 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ -2 \\ -6 \end{bmatrix}$$

+ update weights

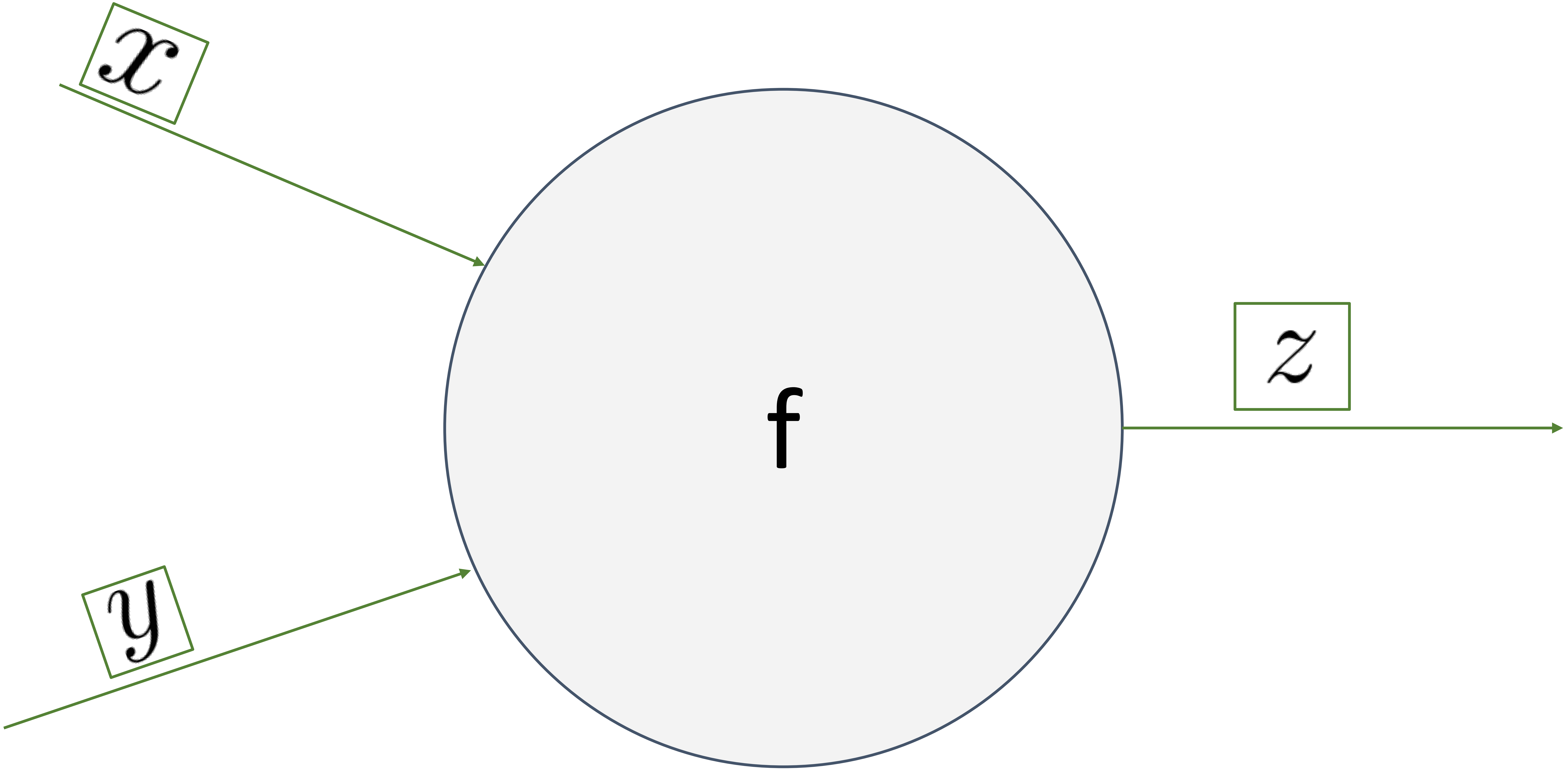
Why **backwards**?

$$y = w_2(\max(0, w_1 x + b_1)) + b_2 \qquad L = (y - t)^2$$

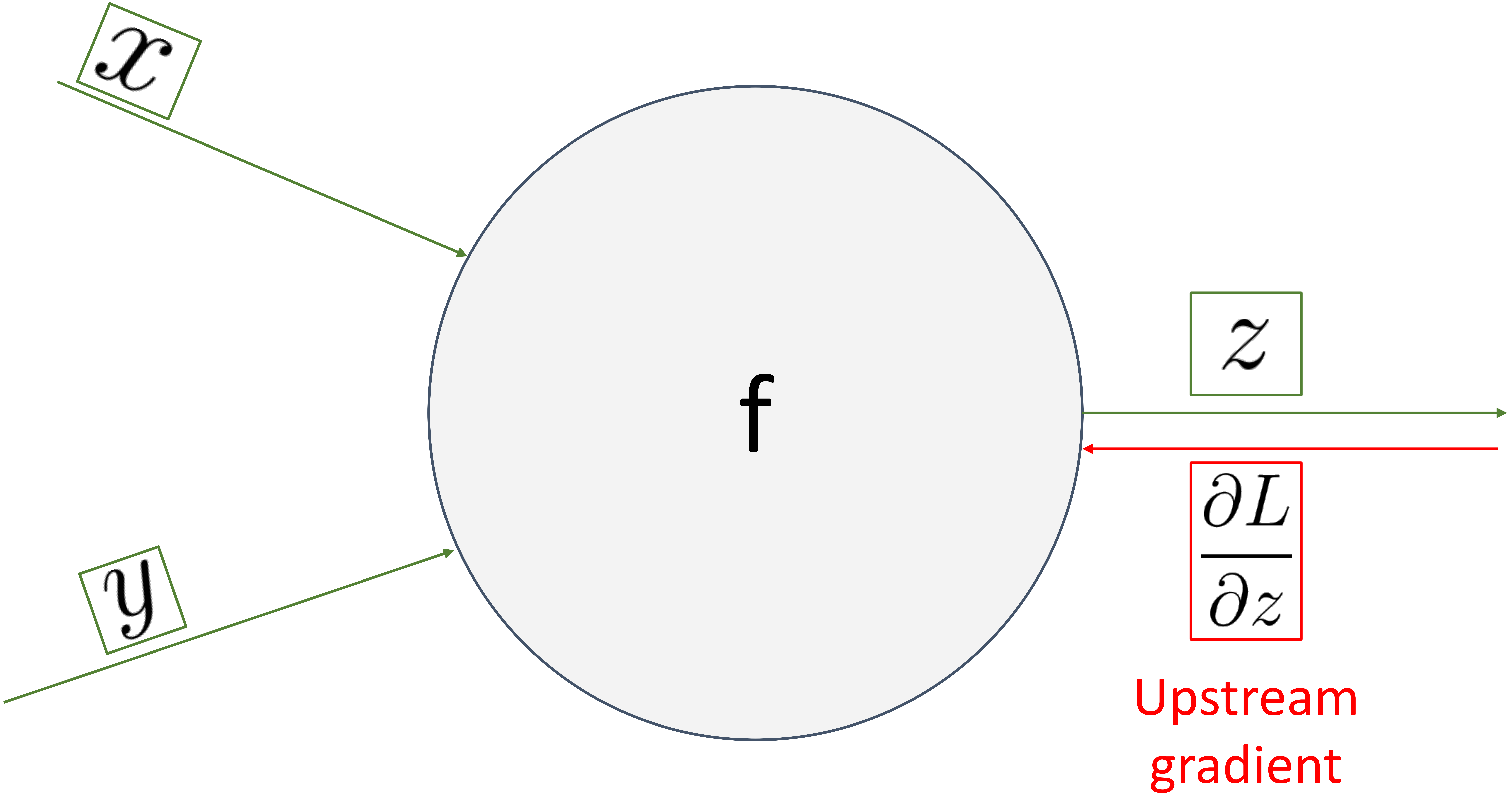




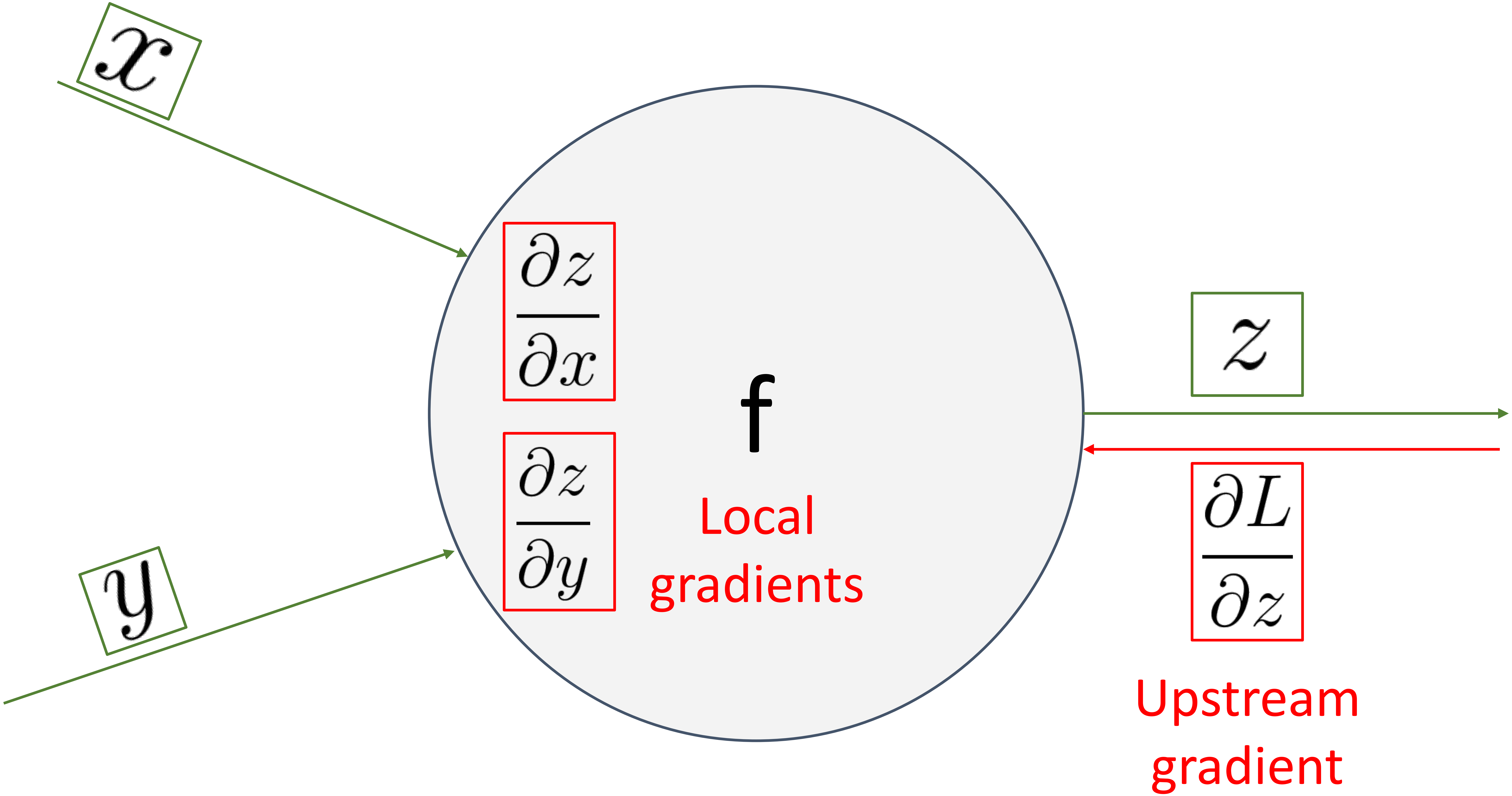
20.1

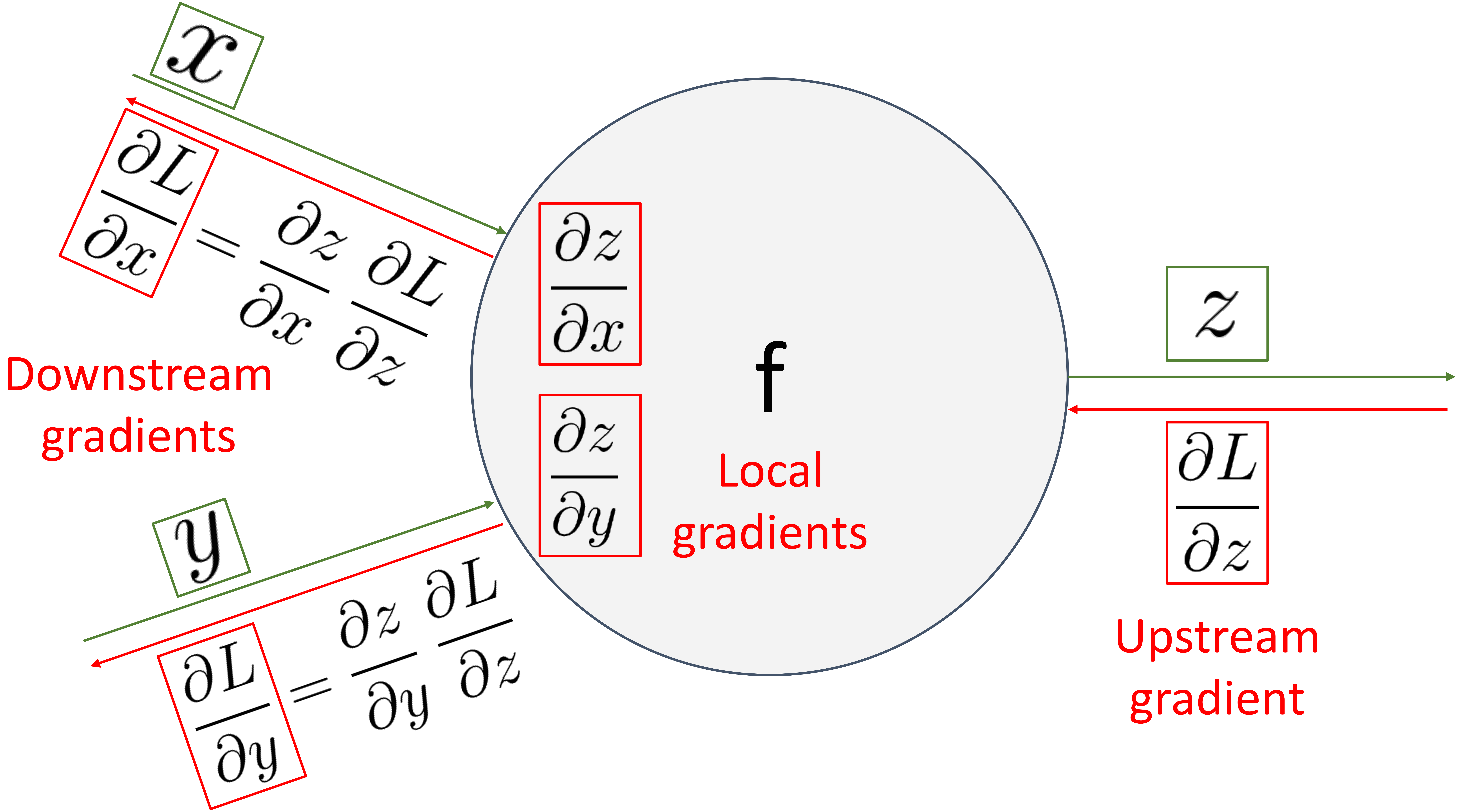


x
y
f
z



x
y
f
z
$\frac{\partial L}{\partial z}$
Upstream gradient


x
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
y
f
Local
gradients
z
$\frac{\partial L}{\partial z}$
Upstream
gradient

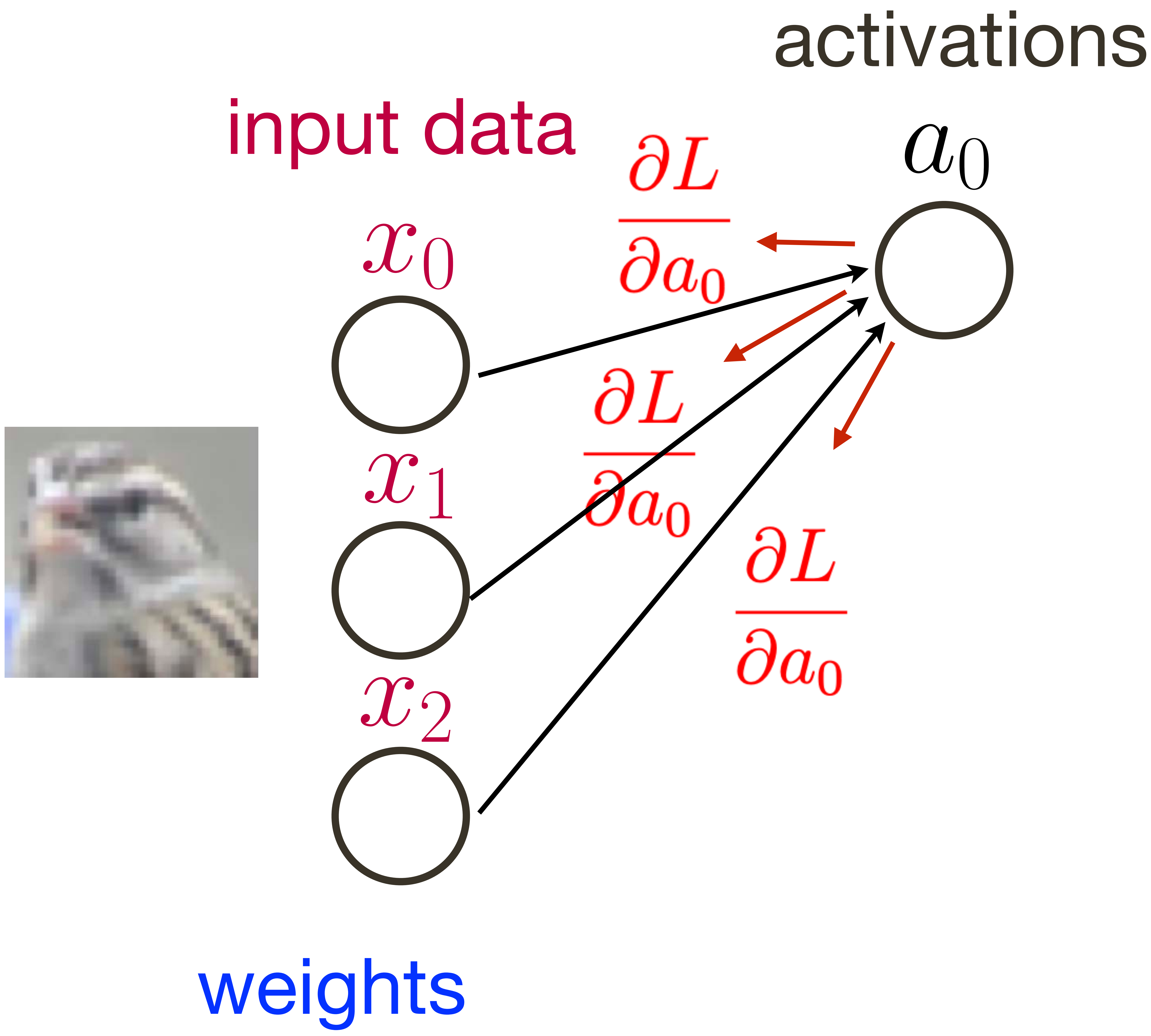


x
$\frac{\partial L}{\partial x} = \frac{\partial z}{\partial x} \frac{\partial L}{\partial z}$
Downstream gradients
y
$\frac{\partial L}{\partial y} = \frac{\partial z}{\partial y} \frac{\partial L}{\partial z}$
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
f
Local gradients
z
$\frac{\partial L}{\partial z}$
Upstream gradient

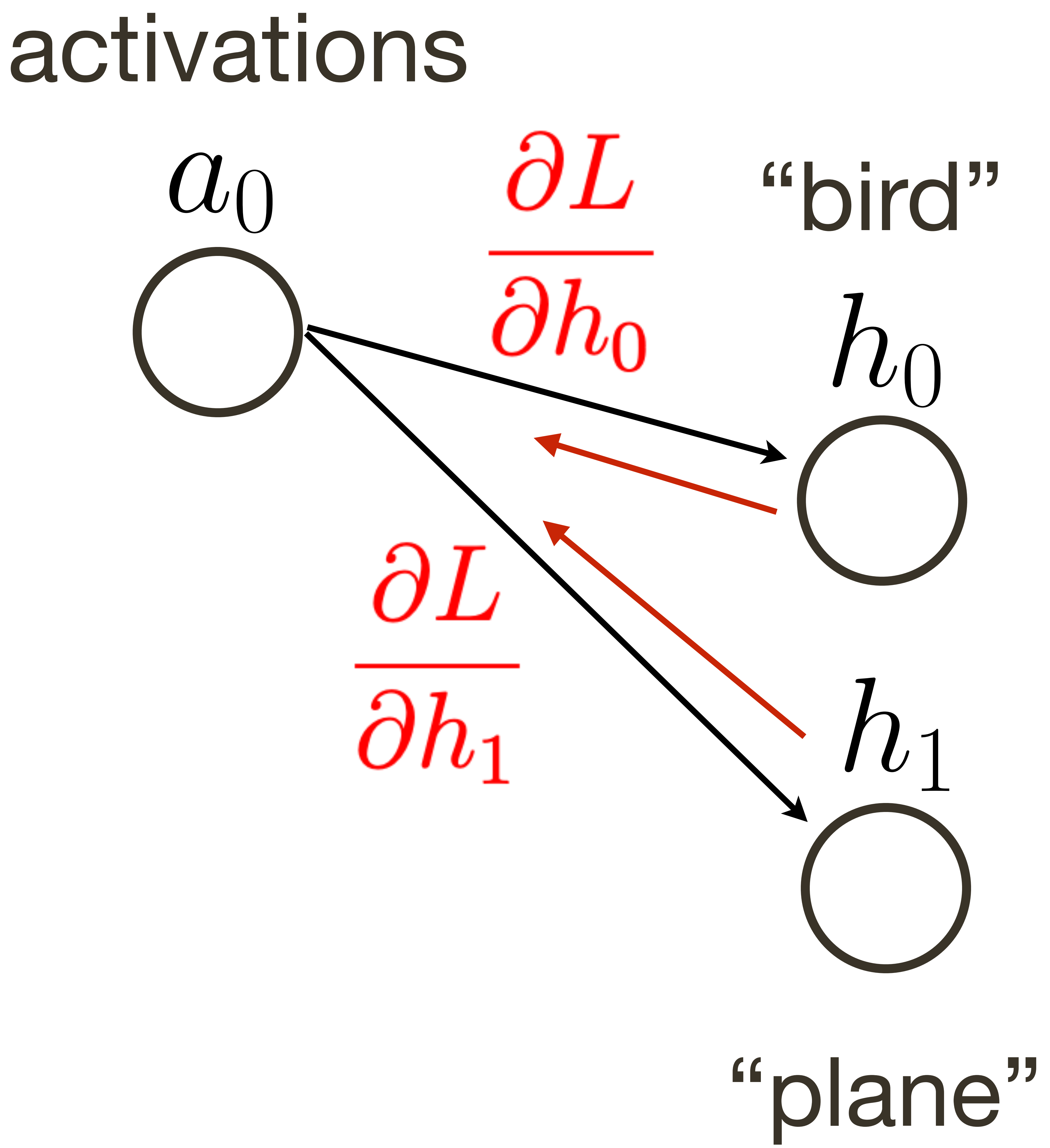

x
$\frac{\partial L}{\partial x} = \frac{\partial z}{\partial x}\frac{\partial L}{\partial z}$
Downstream gradients
y
$\frac{\partial L}{\partial y} = \frac{\partial z}{\partial y}\frac{\partial L}{\partial z}$
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
f
Local gradients
z
$\frac{\partial L}{\partial z}$
Upstream gradient

2-Layer **Neural** Network

2-Layer **Neural** Network — multiple inputs

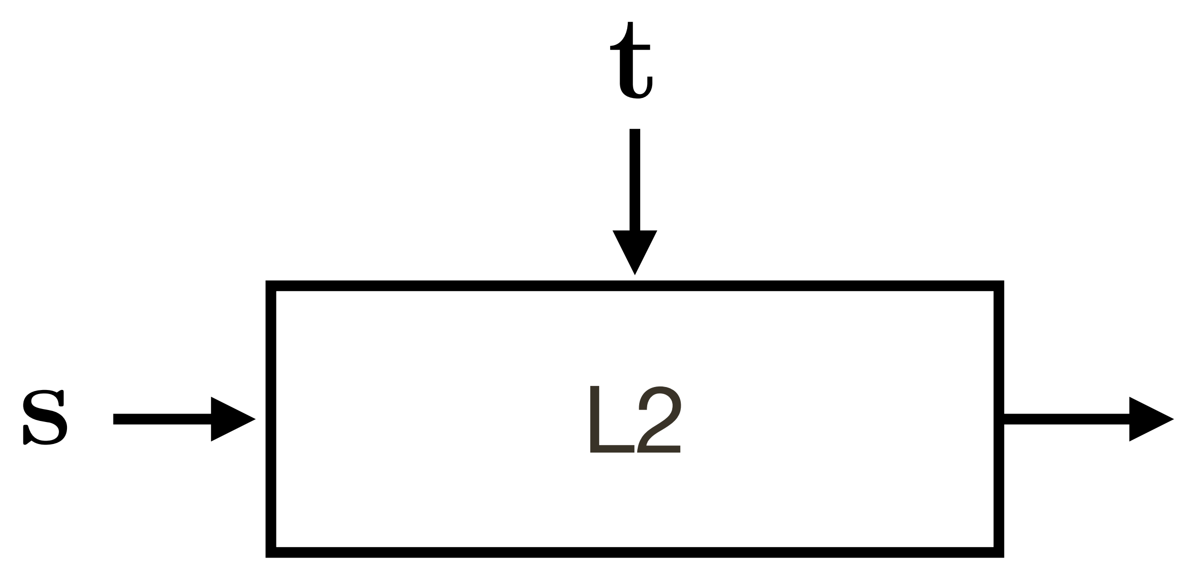


2-Layer **Neural** Network — multiple outputs

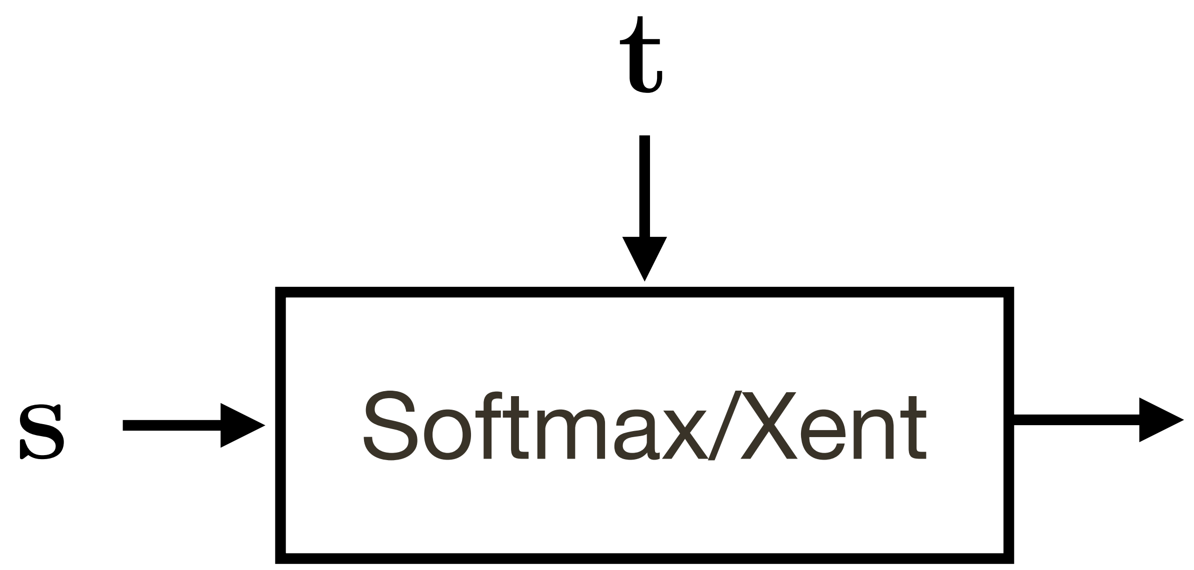


19.7

Backward Pass for Some **Common Layers**



$$e = \frac{1}{2}|\mathbf{s} - \mathbf{t}|^2$$



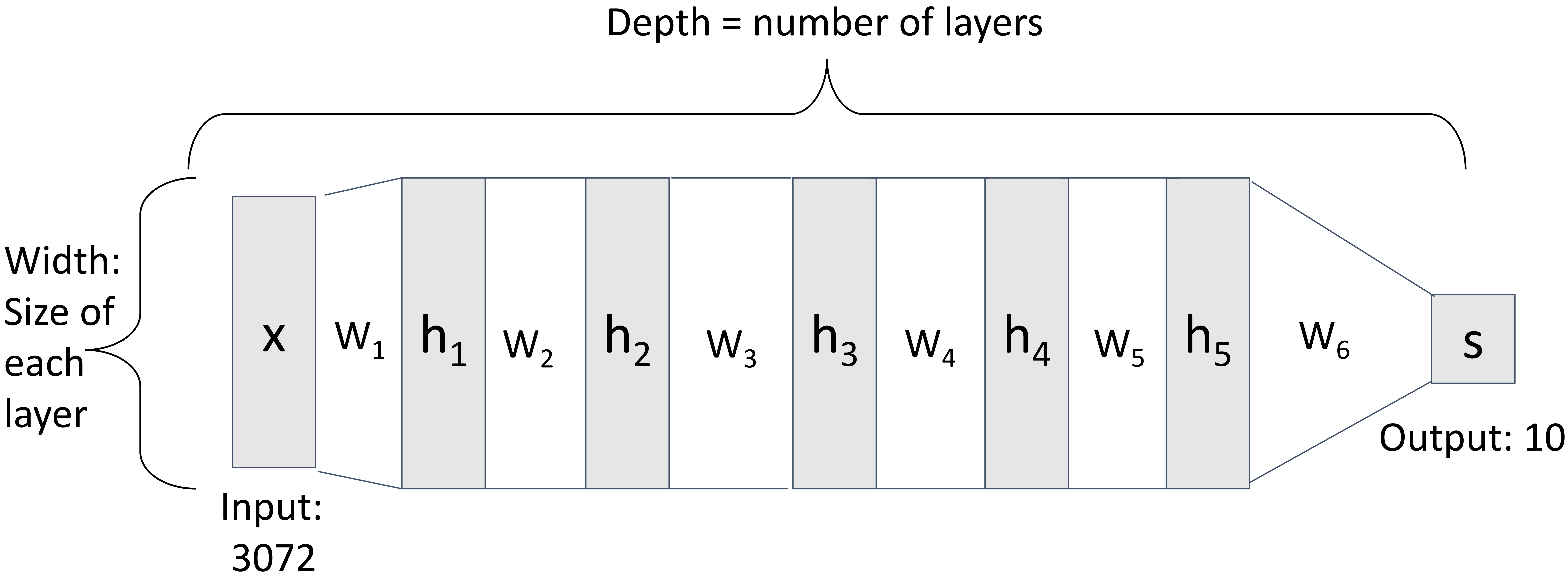
$$e = -\log \frac{e^{s_y}}{\sum_i e^{s_i}}$$

[You will do this for Assignment 6]

$$\frac{\partial e}{\partial \mathbf{s}} = \mathbf{s} - \mathbf{t}$$

$$\frac{\partial e}{\partial \mathbf{s}} = \sigma(\mathbf{s}) - \mathbf{t}$$

Deep Neural Networks



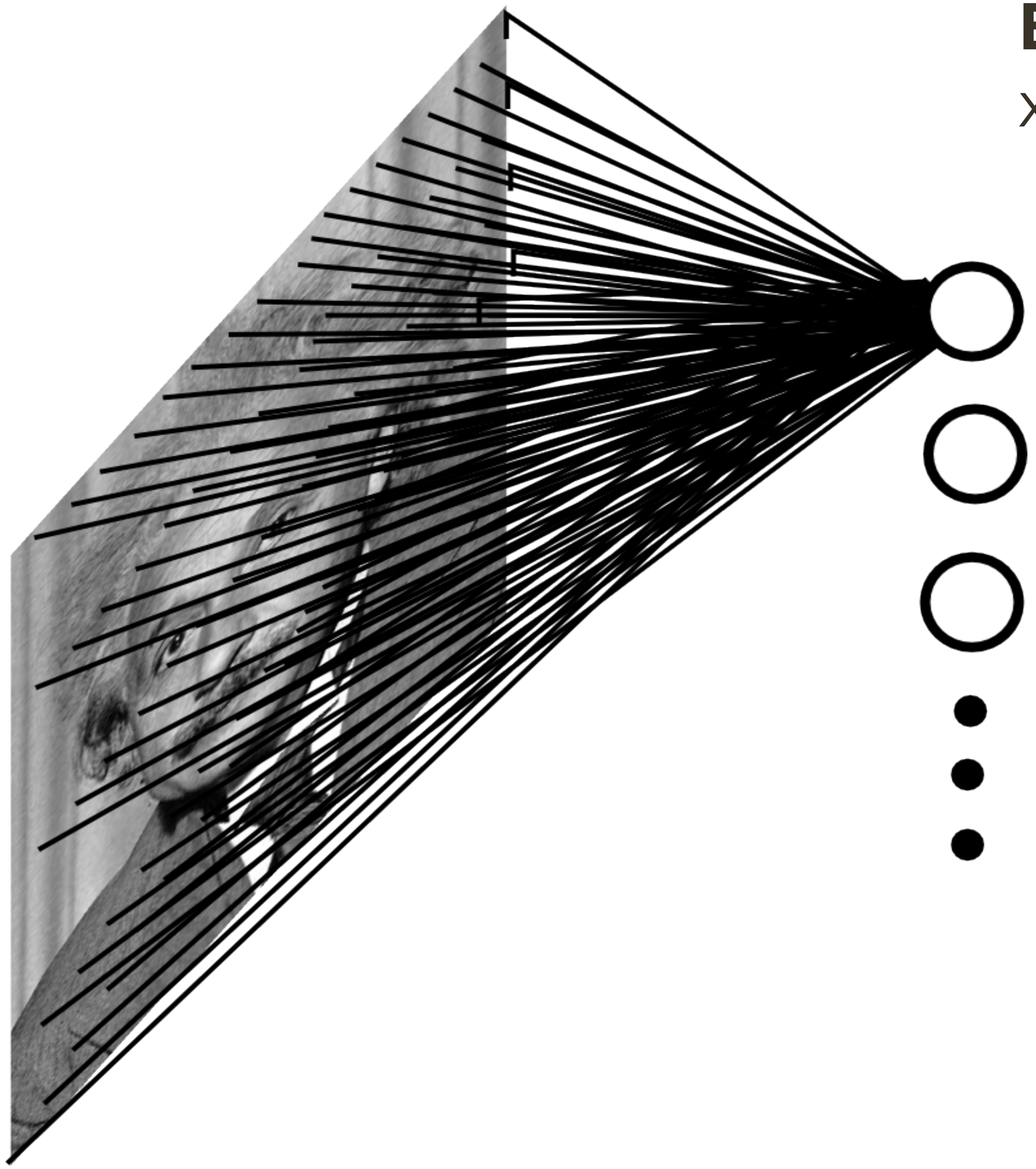
$$s = W_6 \max(0, W_6 \max(0, W_5 \max(0, W_4 \max(0, W_3 \max(0, W_2 \max(0, W_1 x))))))$$

Backward Pass for Some **Common Layers**

Linear layers — fully connected

20.2

Fully Connected Layer



Example: 200 x 200 image (small)
x 40K hidden units (same size)

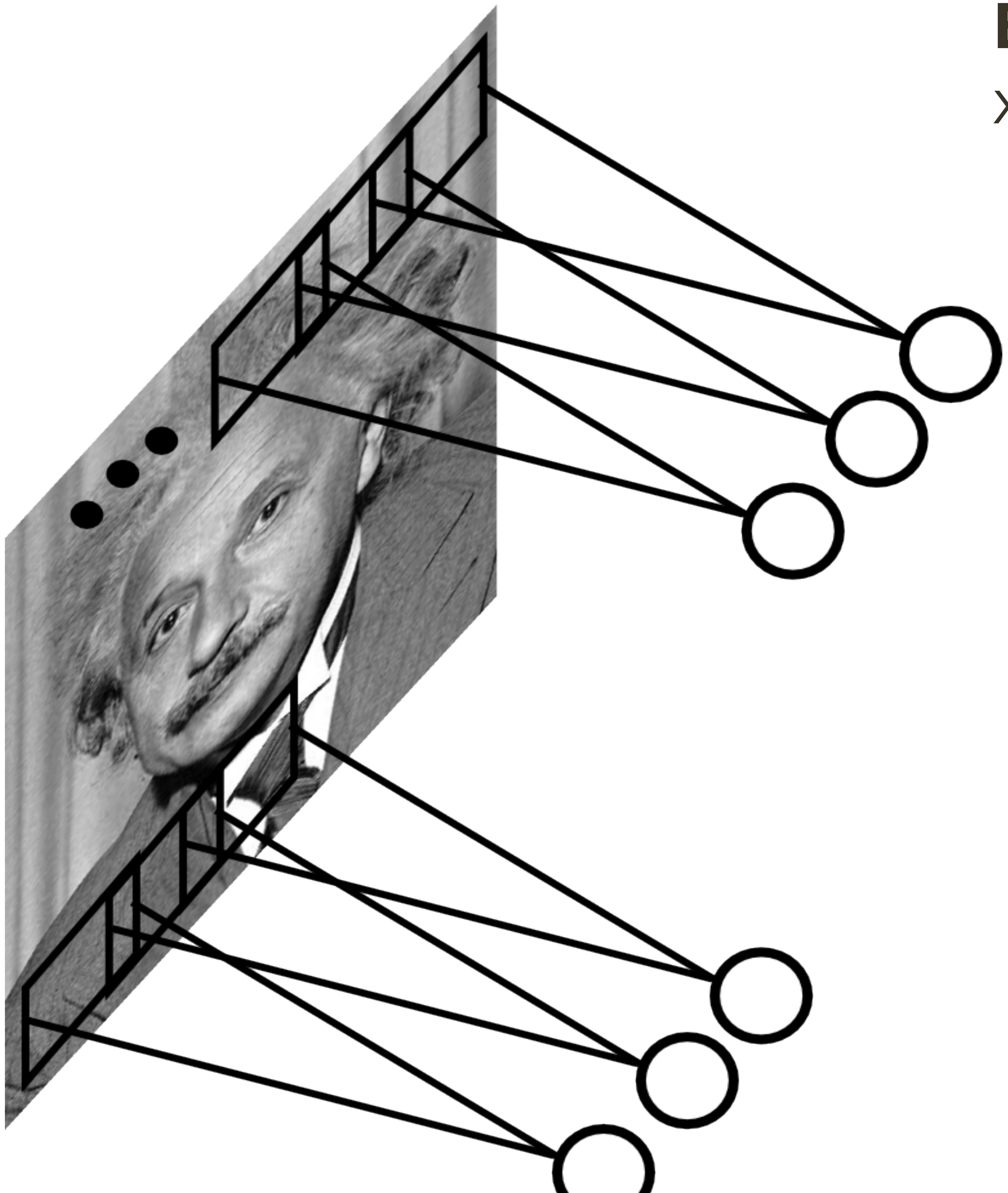
= **1.6 Billion** parameters (for one layer!)

Spatial correlations are generally local

Waste of resources + we don't have enough data to train networks this large

* slide from Marc'Aurelio Renzato

Convolutional Layer



Example: 200 x 200 image (small)
x 40K hidden units (same size)

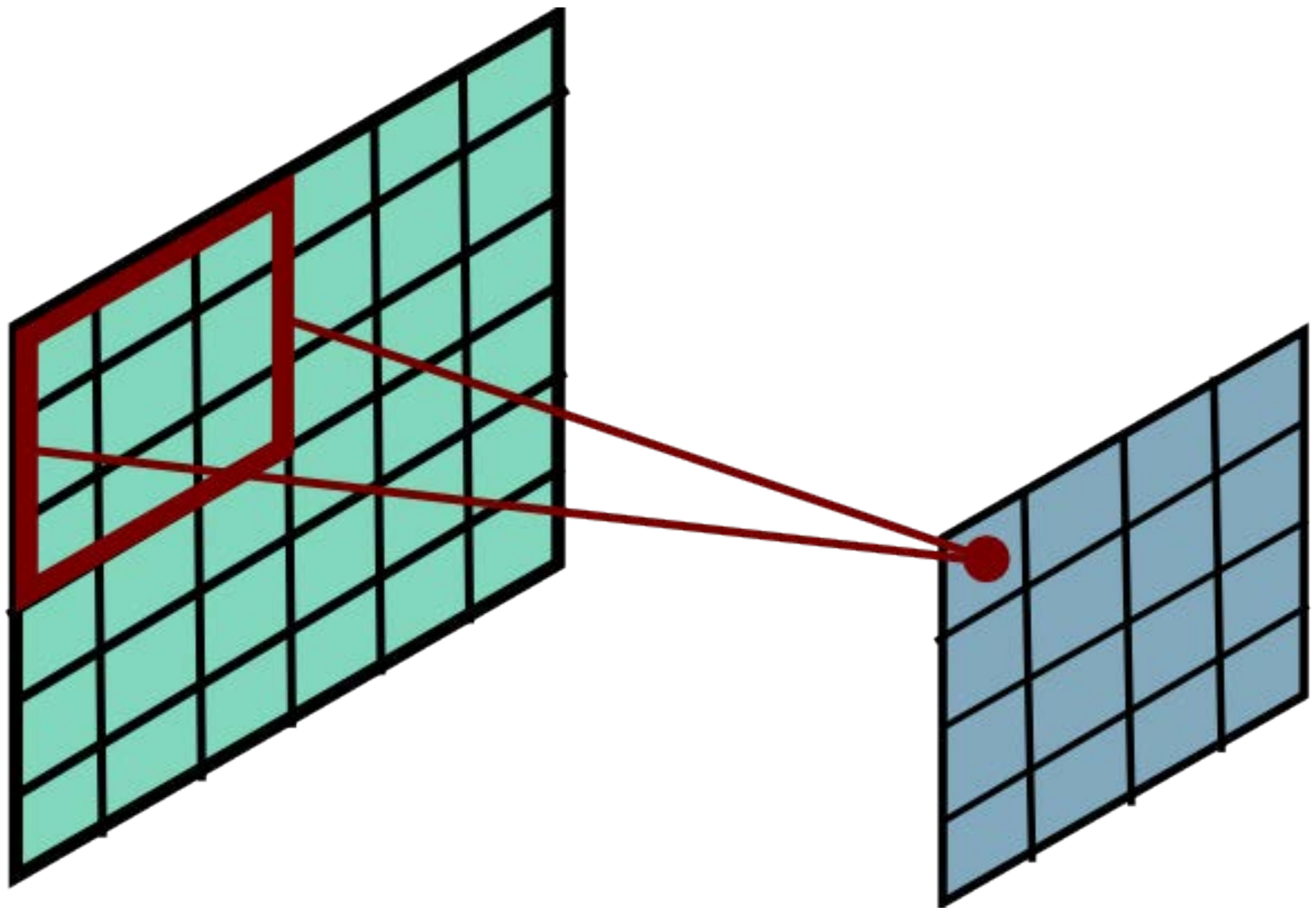
Filter size: 10 x 10

= 100 parameters

Share the same parameters across the locations (assuming input is stationary)

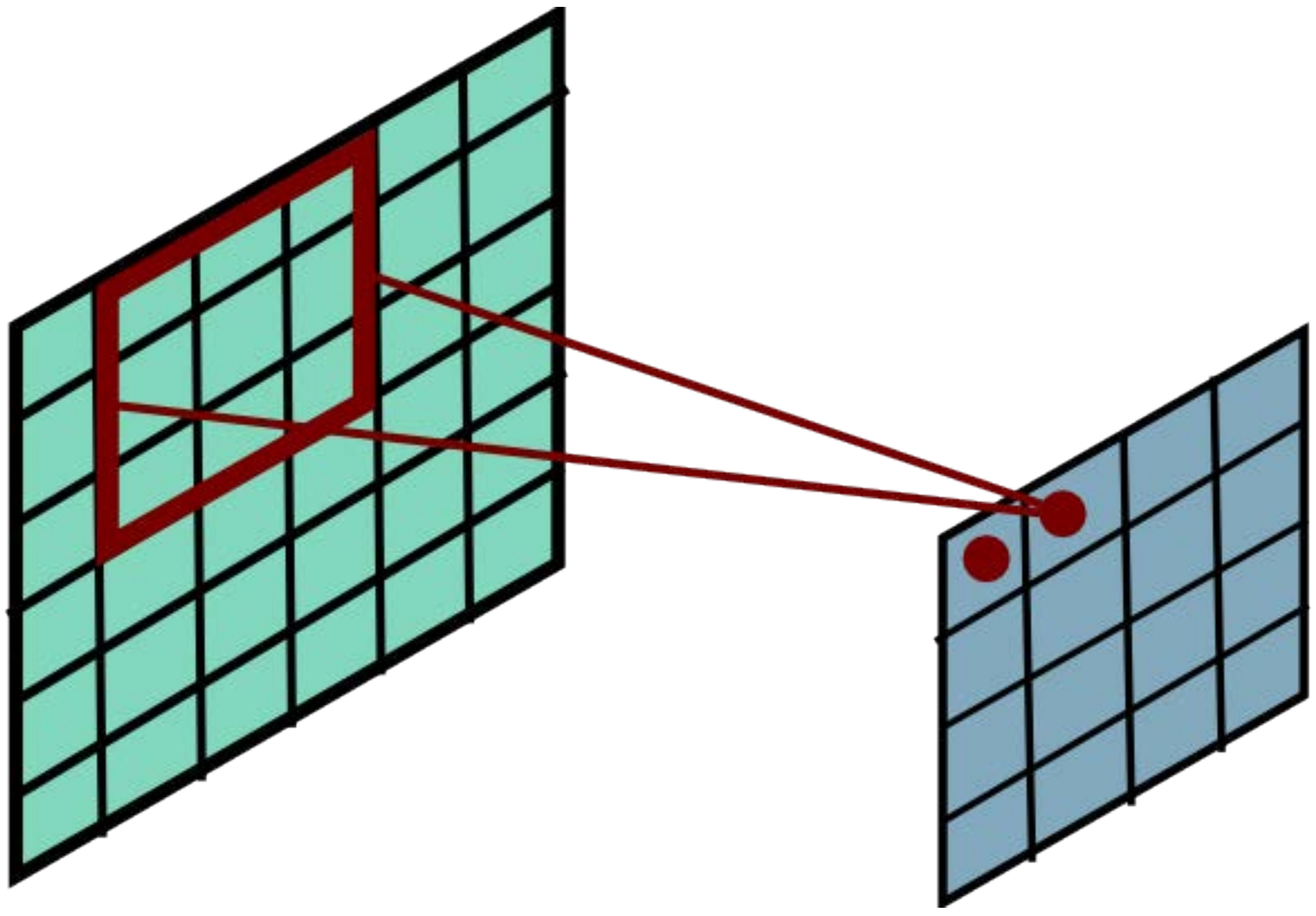
* slide adopted from Marc'Aurelio Renzato

Convolutional Layer



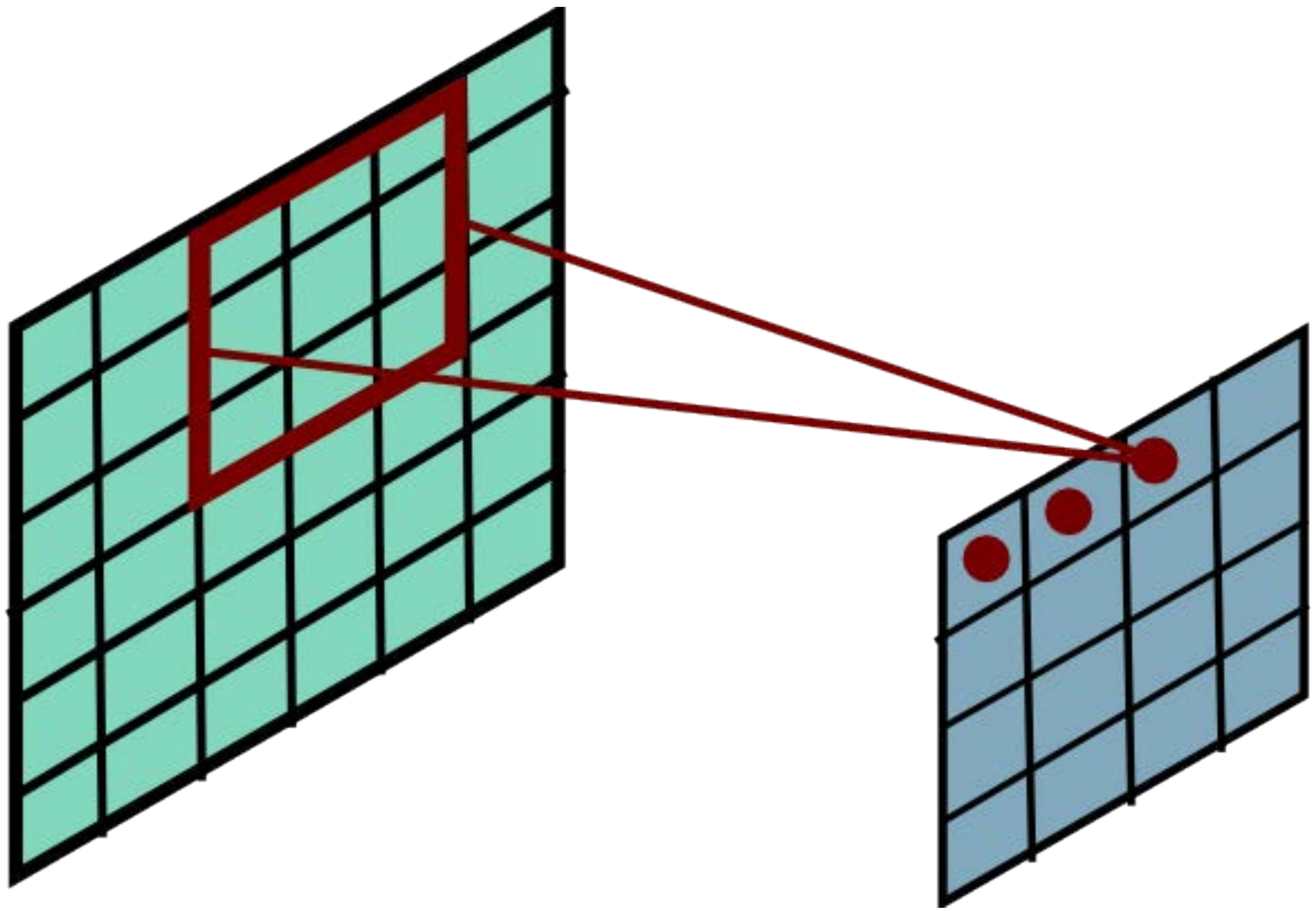
* slide from Marc'Aurelio Renzato

Convolutional Layer



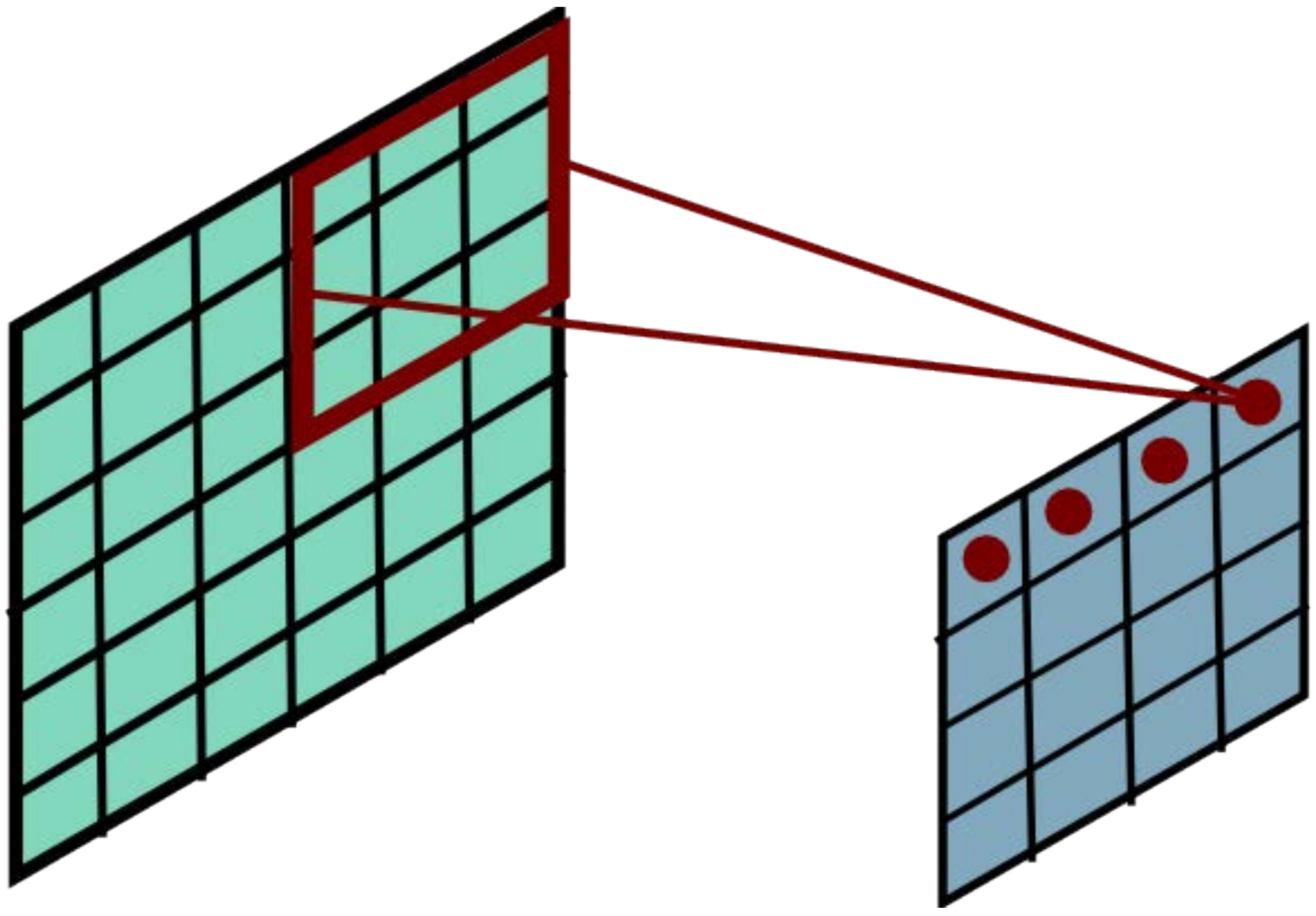
* slide from Marc'Aurelio Renzato

Convolutional Layer



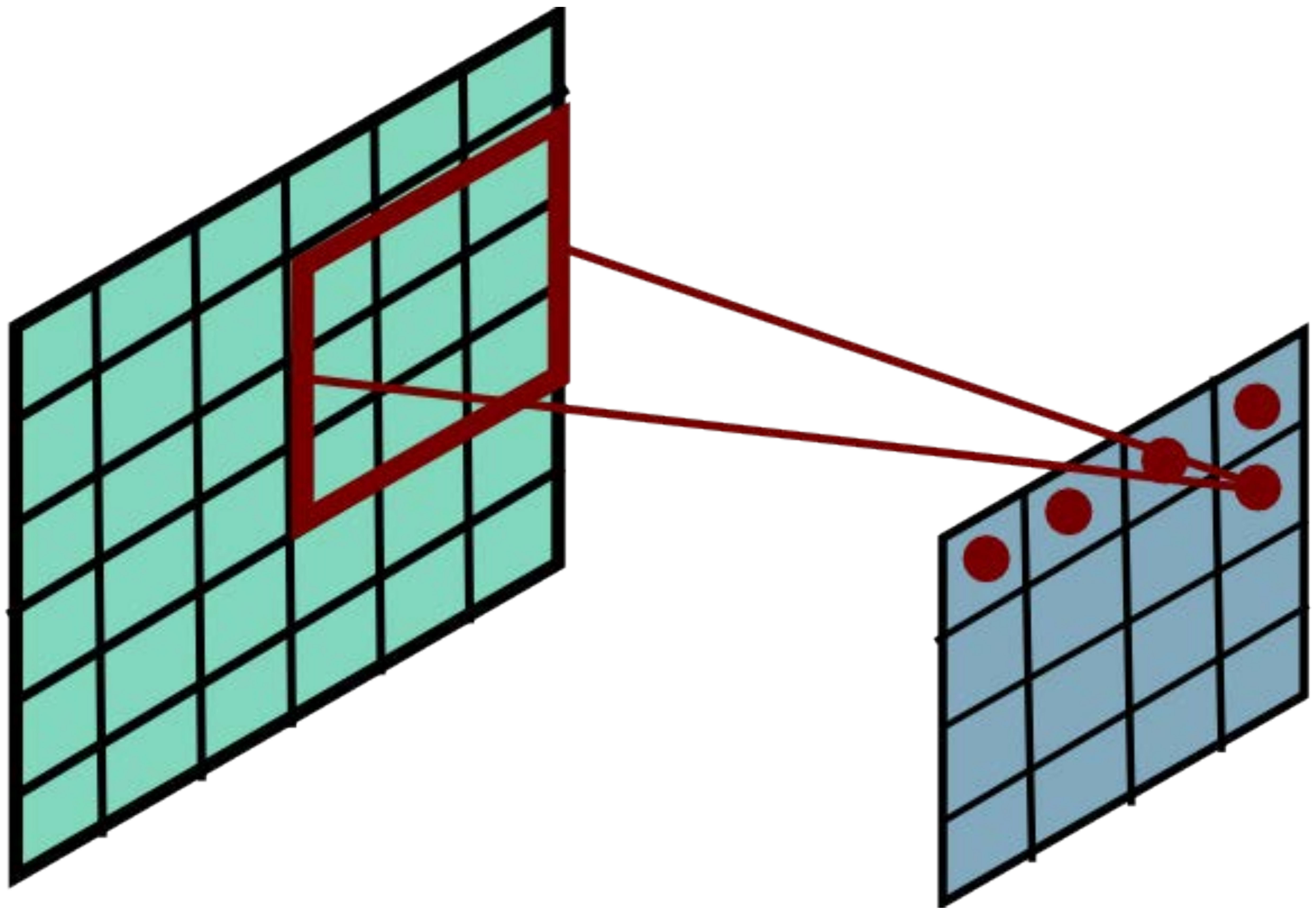
* slide from Marc'Aurelio Renzato

Convolutional Layer



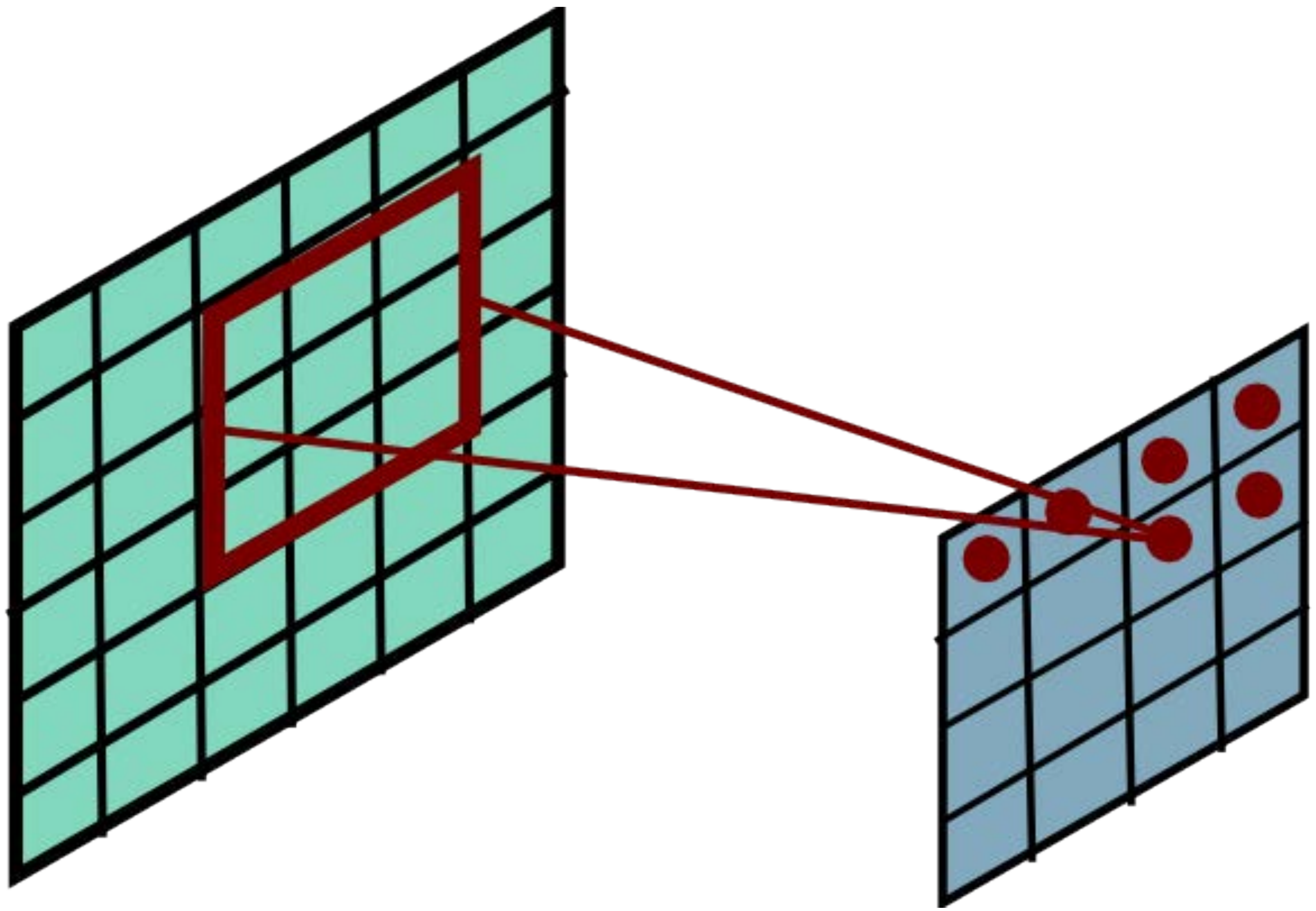
* slide from Marc'Aurelio Renzato

Convolutional Layer



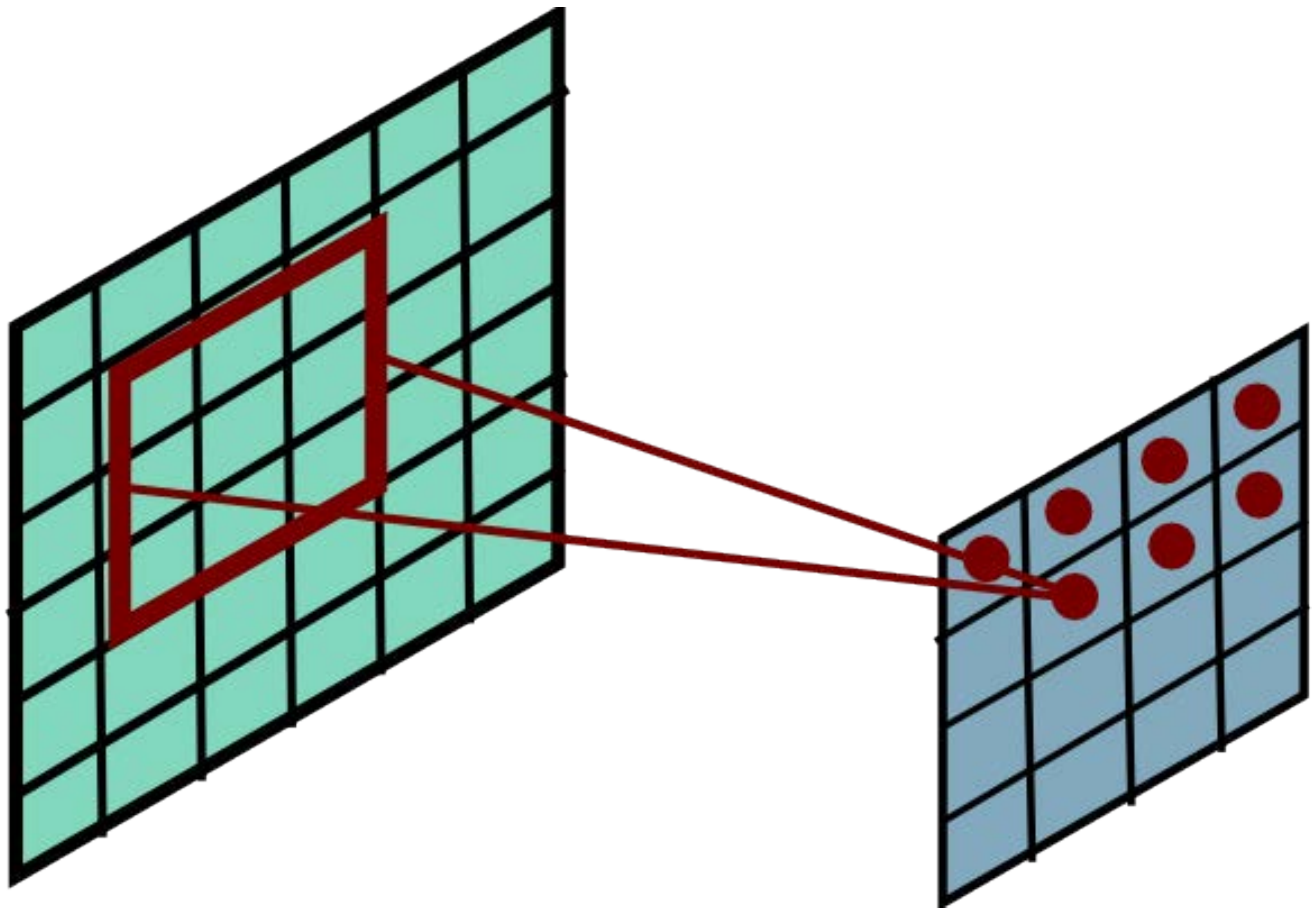
* slide from Marc'Aurelio Renzato

Convolutional Layer



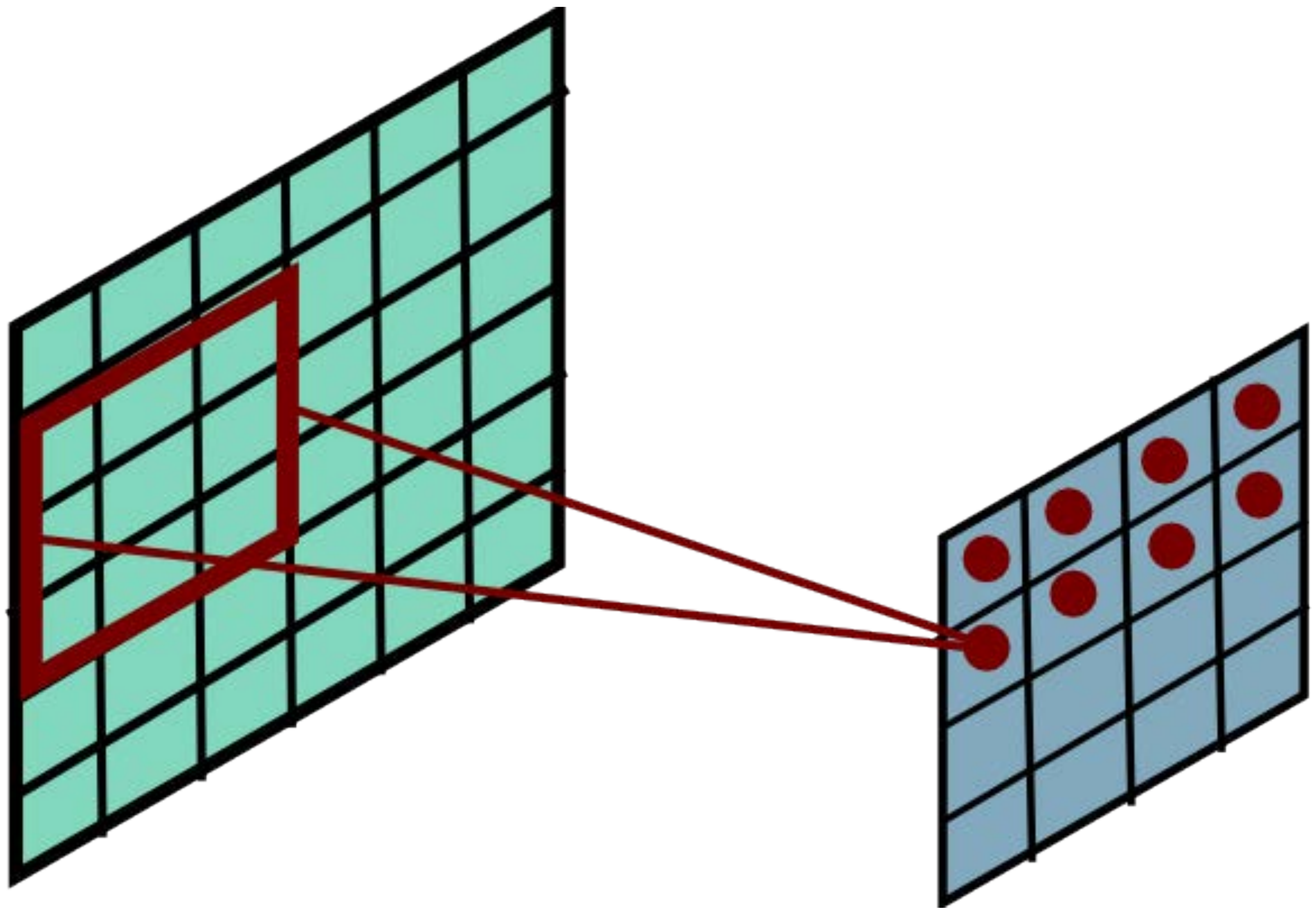
* slide from Marc'Aurelio Renzato

Convolutional Layer



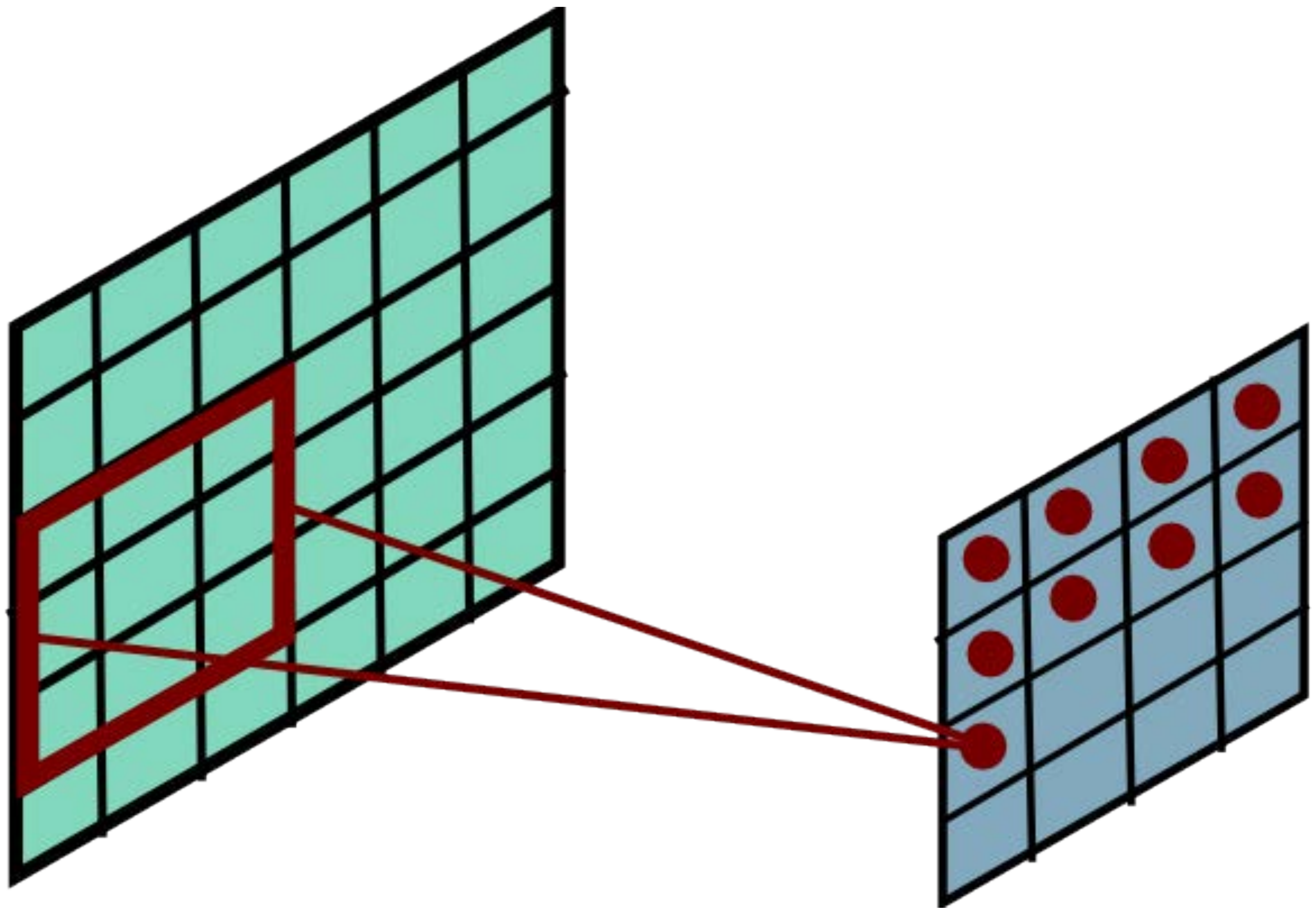
* slide from Marc'Aurelio Renzato

Convolutional Layer



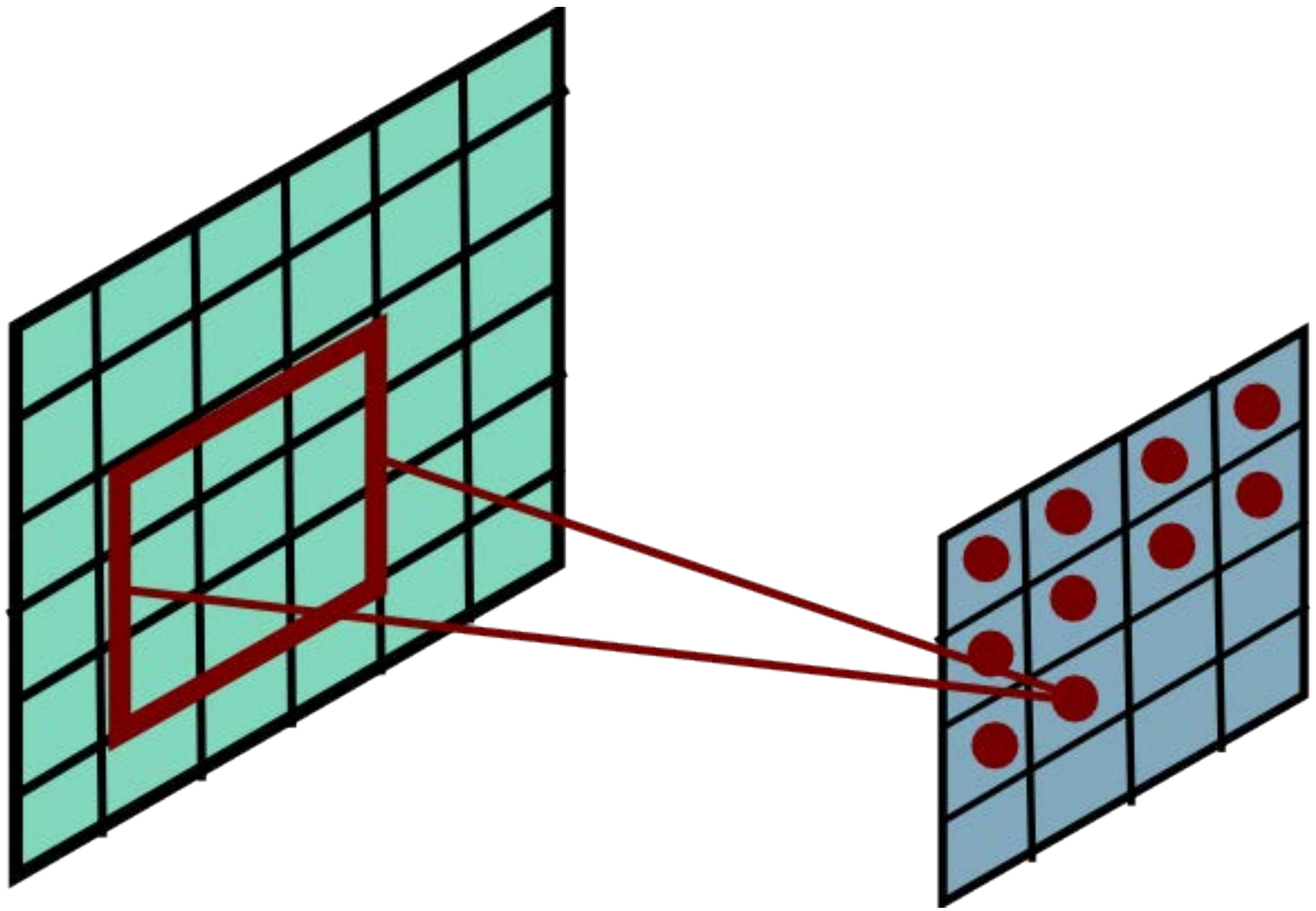
* slide from Marc'Aurelio Renzato

Convolutional Layer



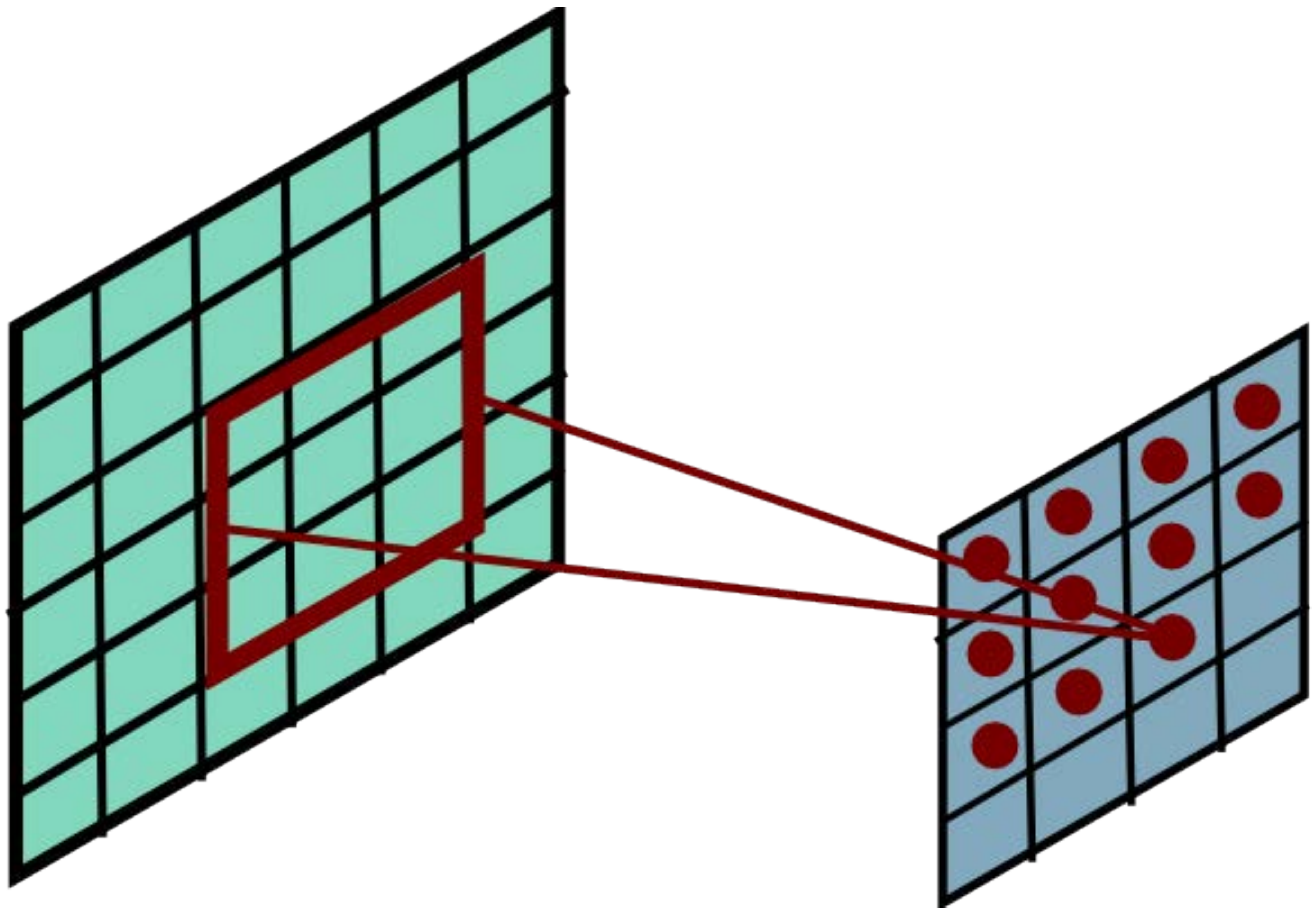
* slide from Marc'Aurelio Renzato

Convolutional Layer



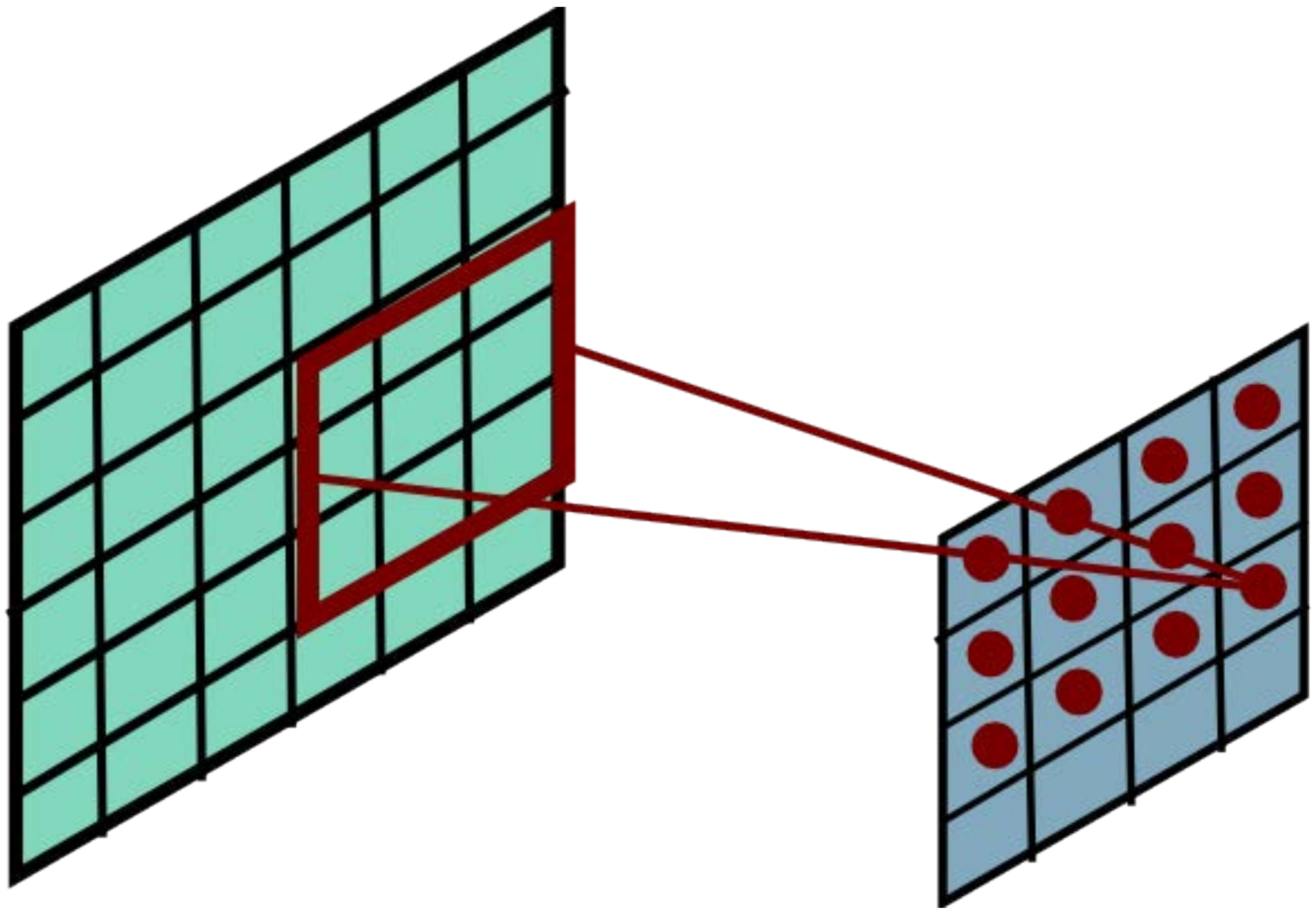
* slide from Marc'Aurelio Renzato

Convolutional Layer



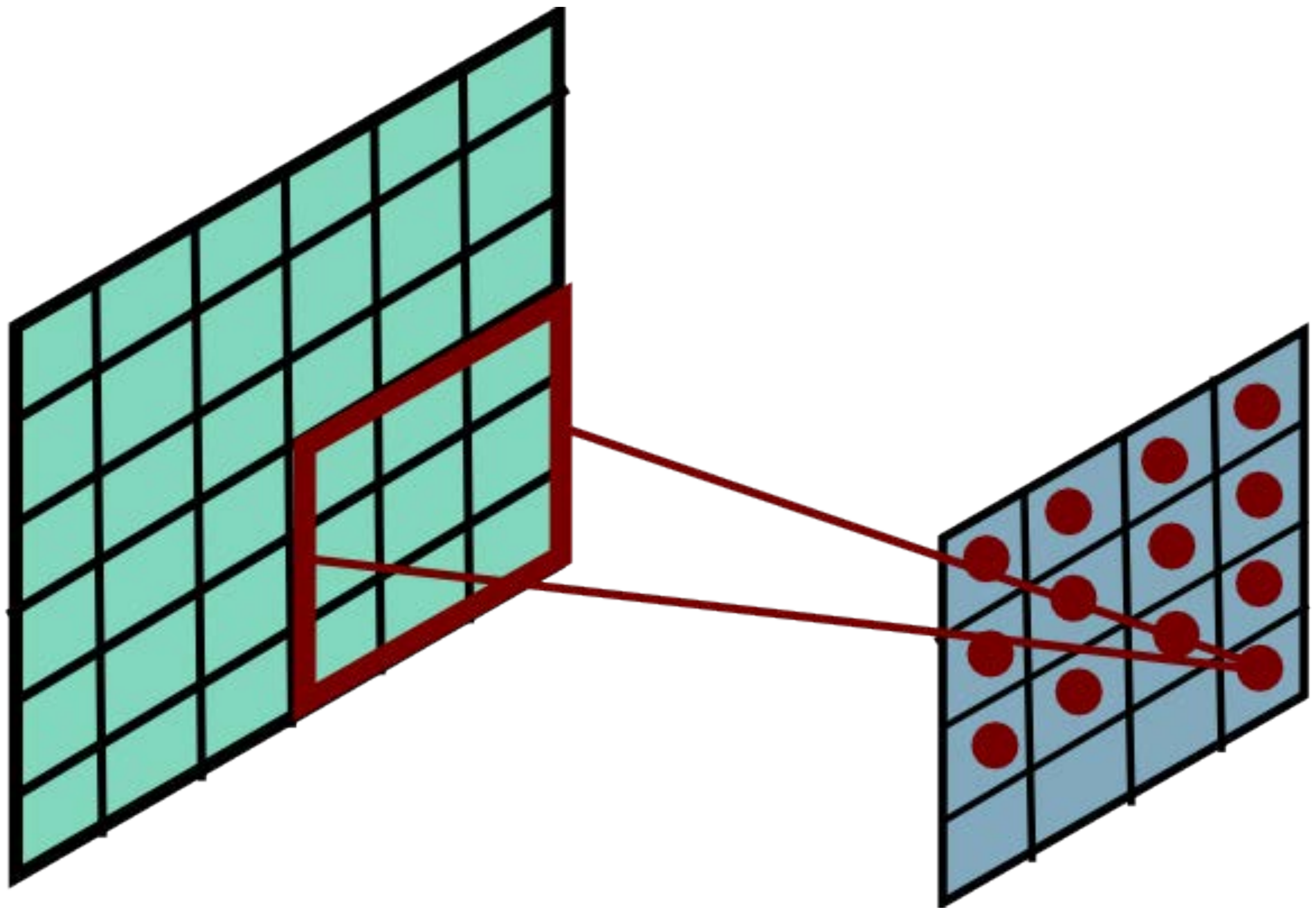
* slide from Marc'Aurelio Renzato

Convolutional Layer



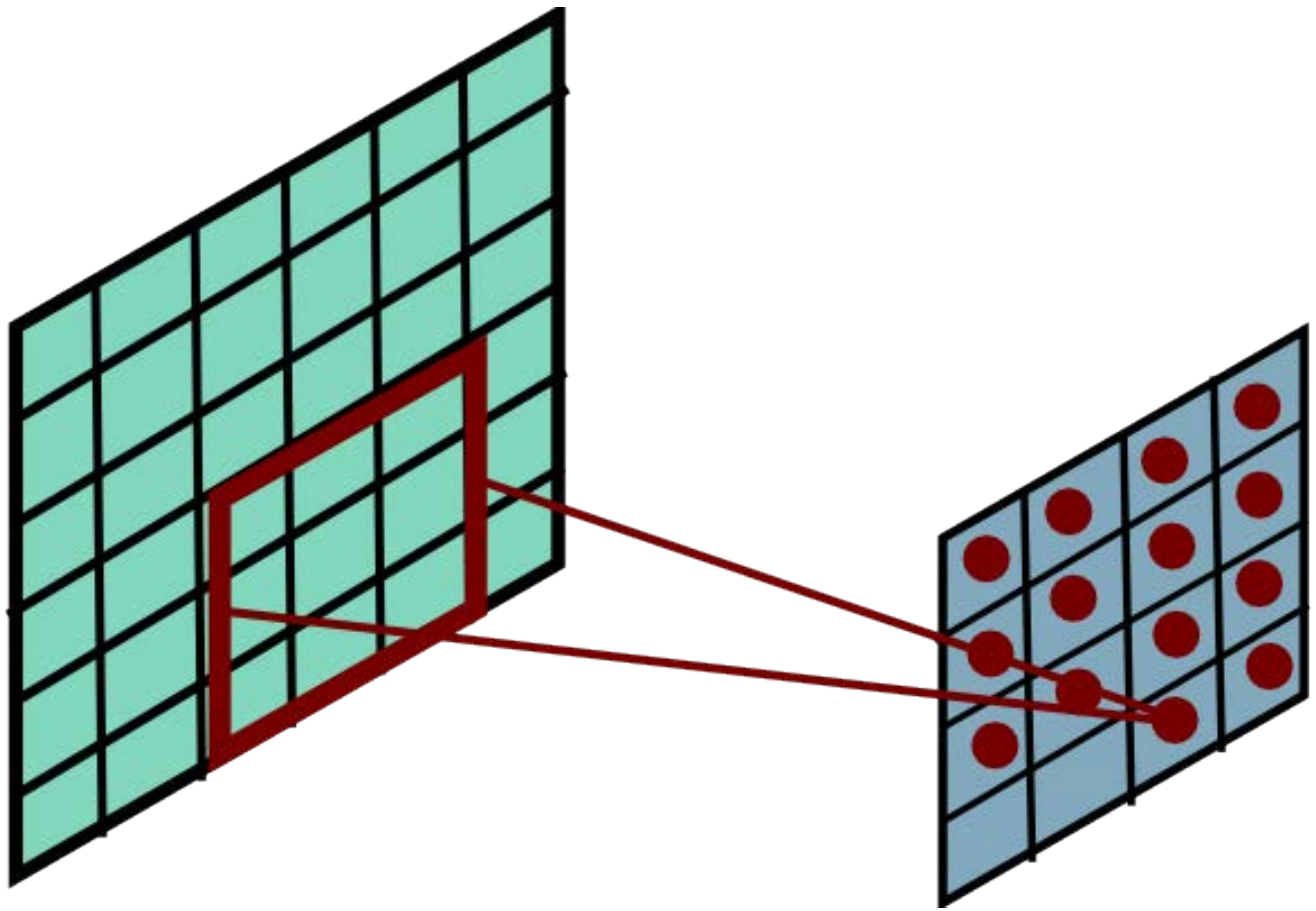
* slide from Marc'Aurelio Renzato

Convolutional Layer



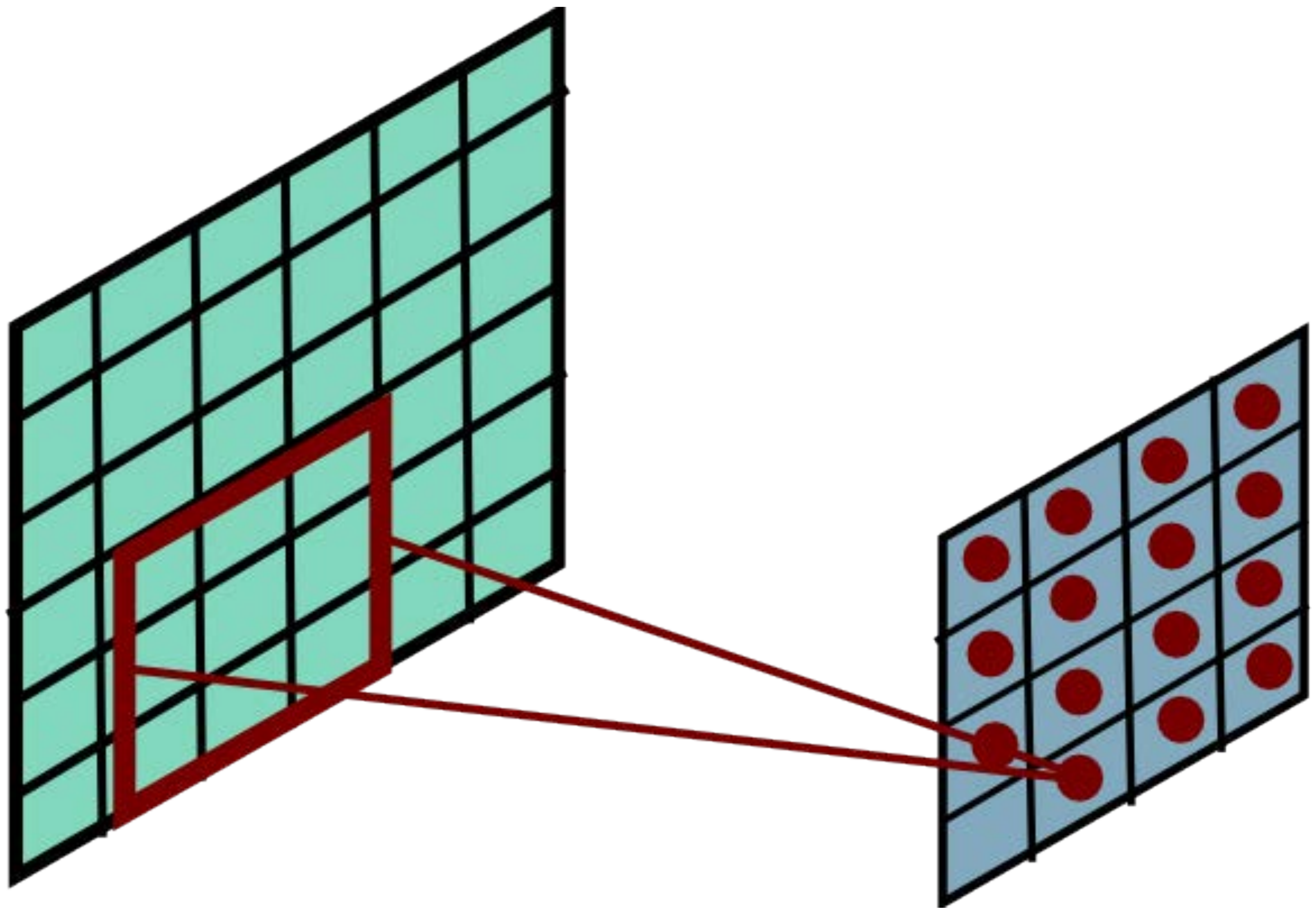
* slide from Marc'Aurelio Renzato

Convolutional Layer



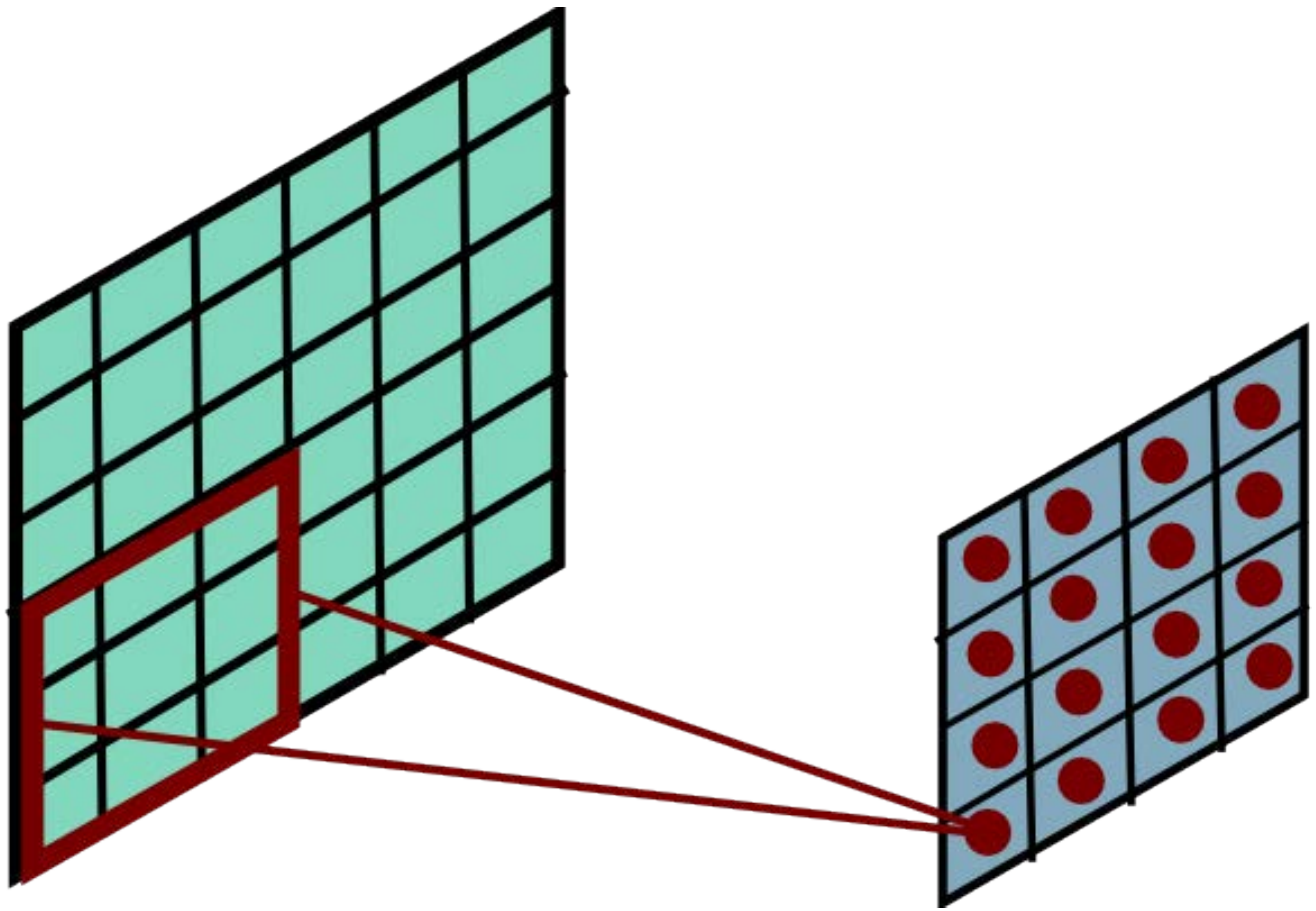
* slide from Marc’Aurelio Renzato

Convolutional Layer



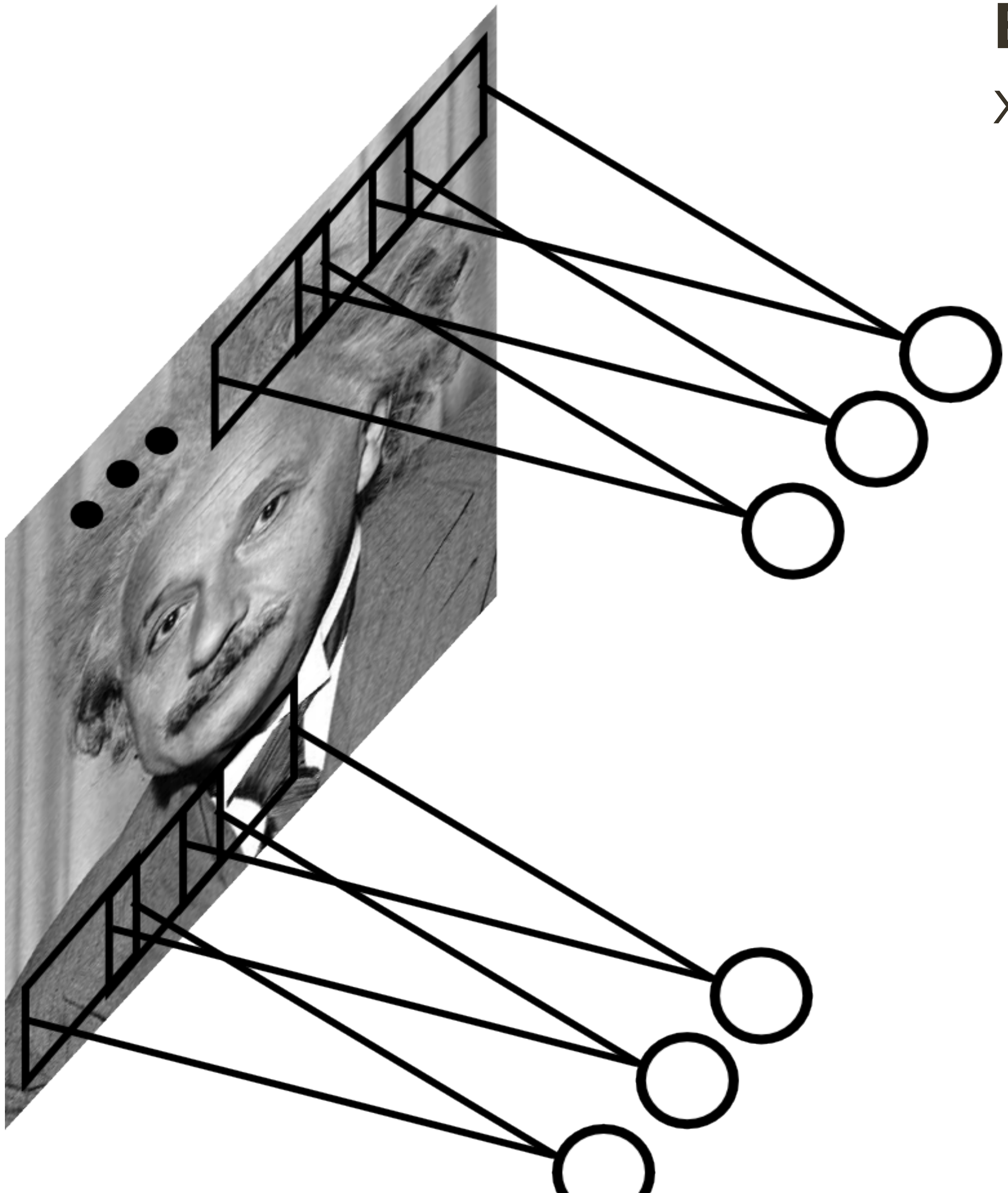
* slide from Marc'Aurelio Renzato

Convolutional Layer



* slide from Marc'Aurelio Renzato

Convolutional Layer



Example: 200 x 200 image (small)
x 40K hidden units (same size)

Filter size: 10 x 10

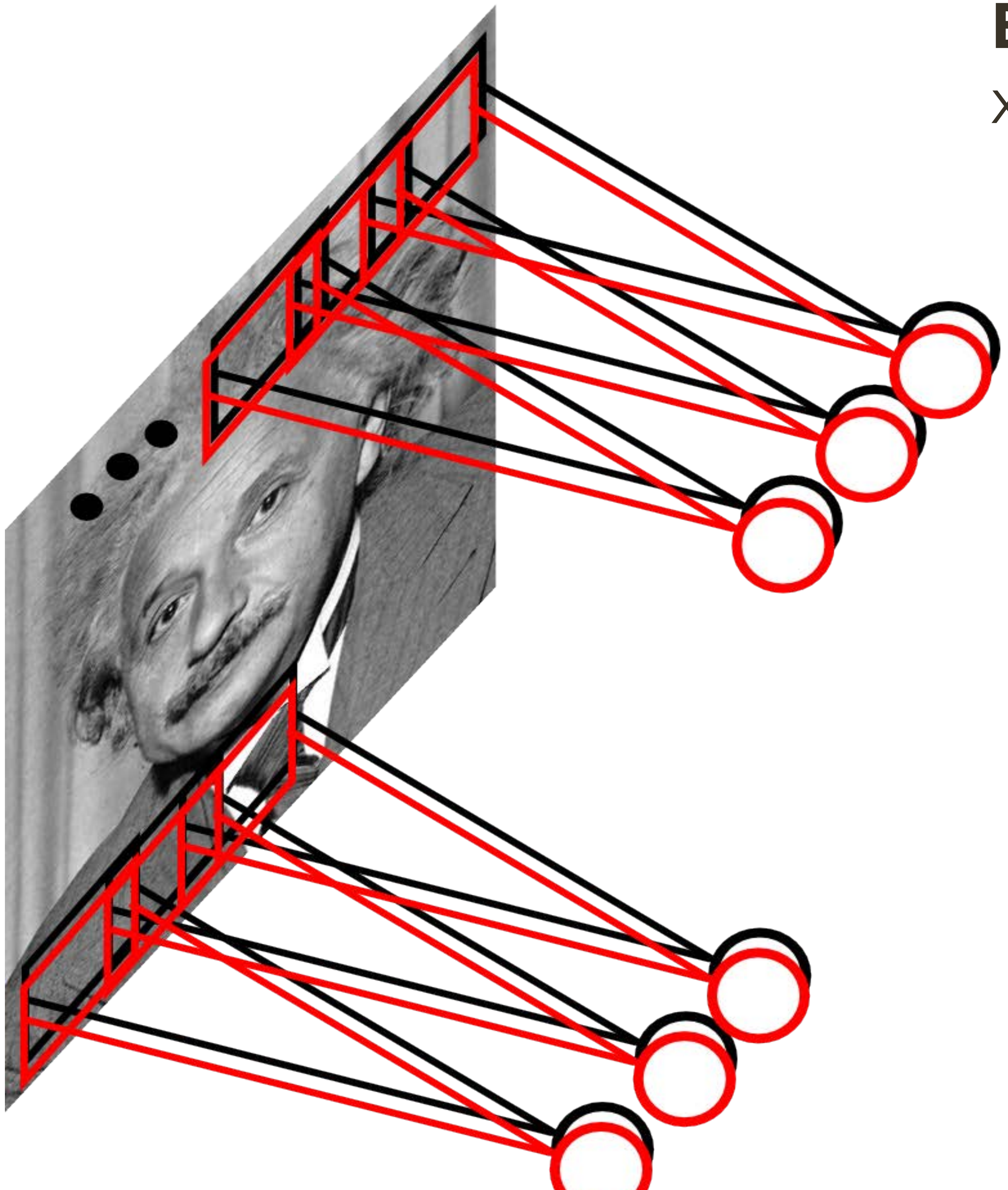
= 100 parameters

Share the same parameters across the locations (assuming input is stationary)

* slide adopted from Marc'Aurelio Renzato

Optional subtitle

Convolutional Layer



Example: 200 x 200 image (small)
x 40K hidden units (same size)

Filter size: 10 x 10

# of filters: 20

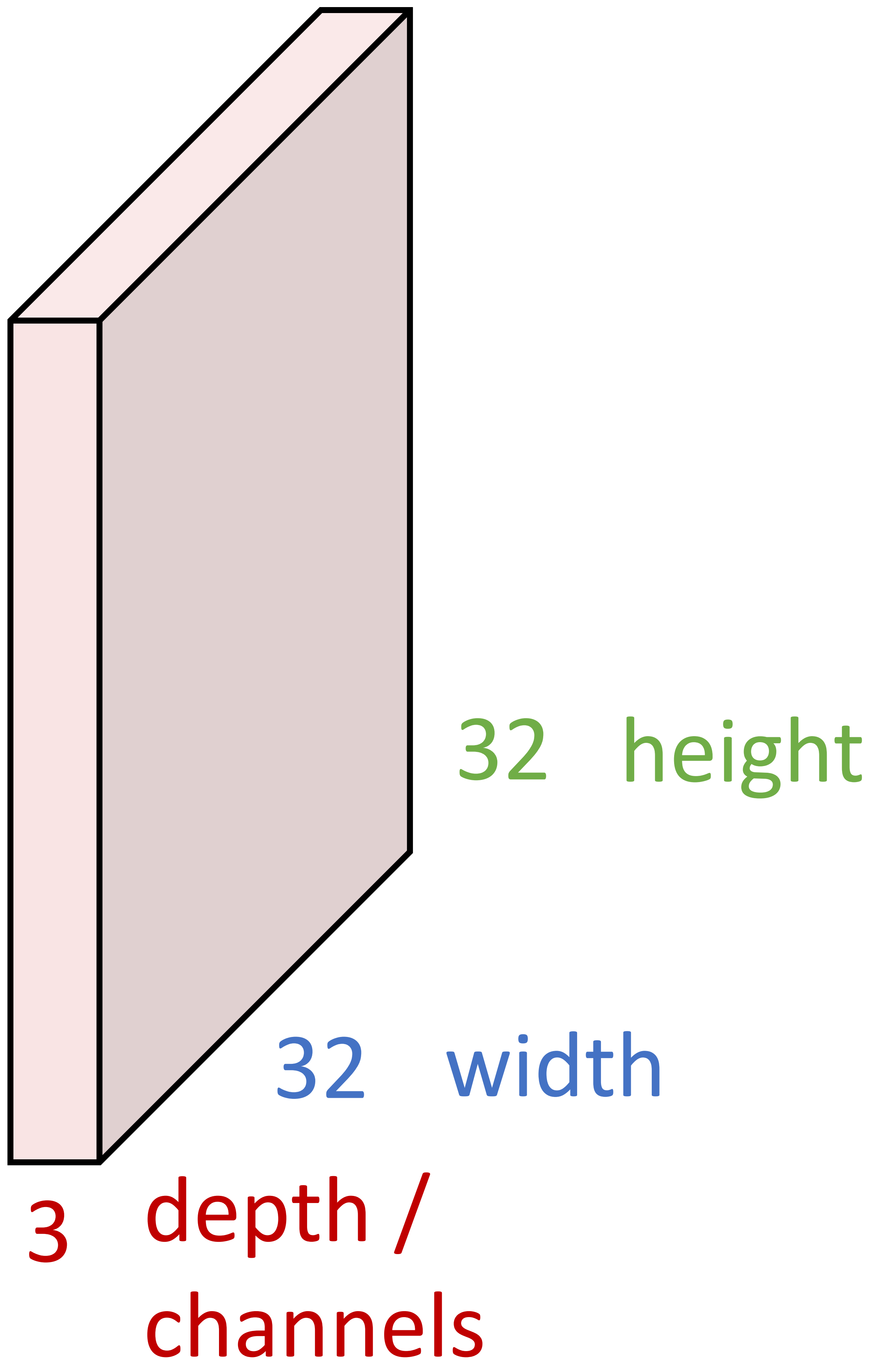
= 2000 parameters

Learn **multiple filters**
→ **multiple output channels**

* slide from Marc'Aurelio Renzato

Convolution Layer

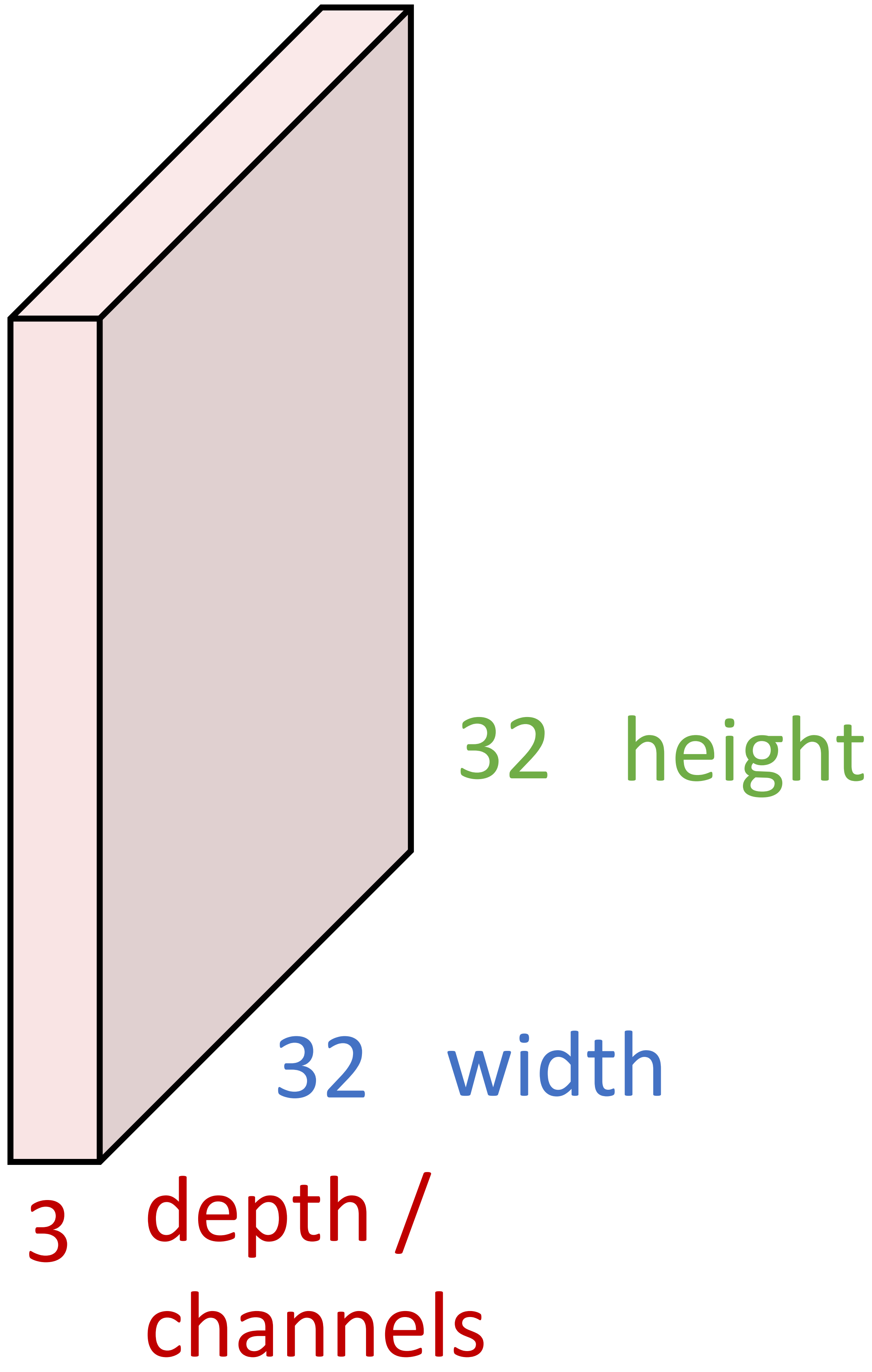
3x32x32 image: preserve spatial structure



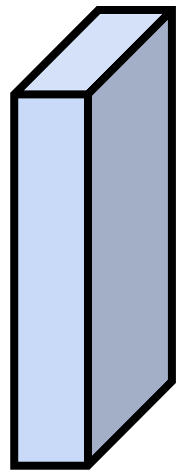
Convolution Layer

Filters always extend the full depth of the input volume

3x32x32 image

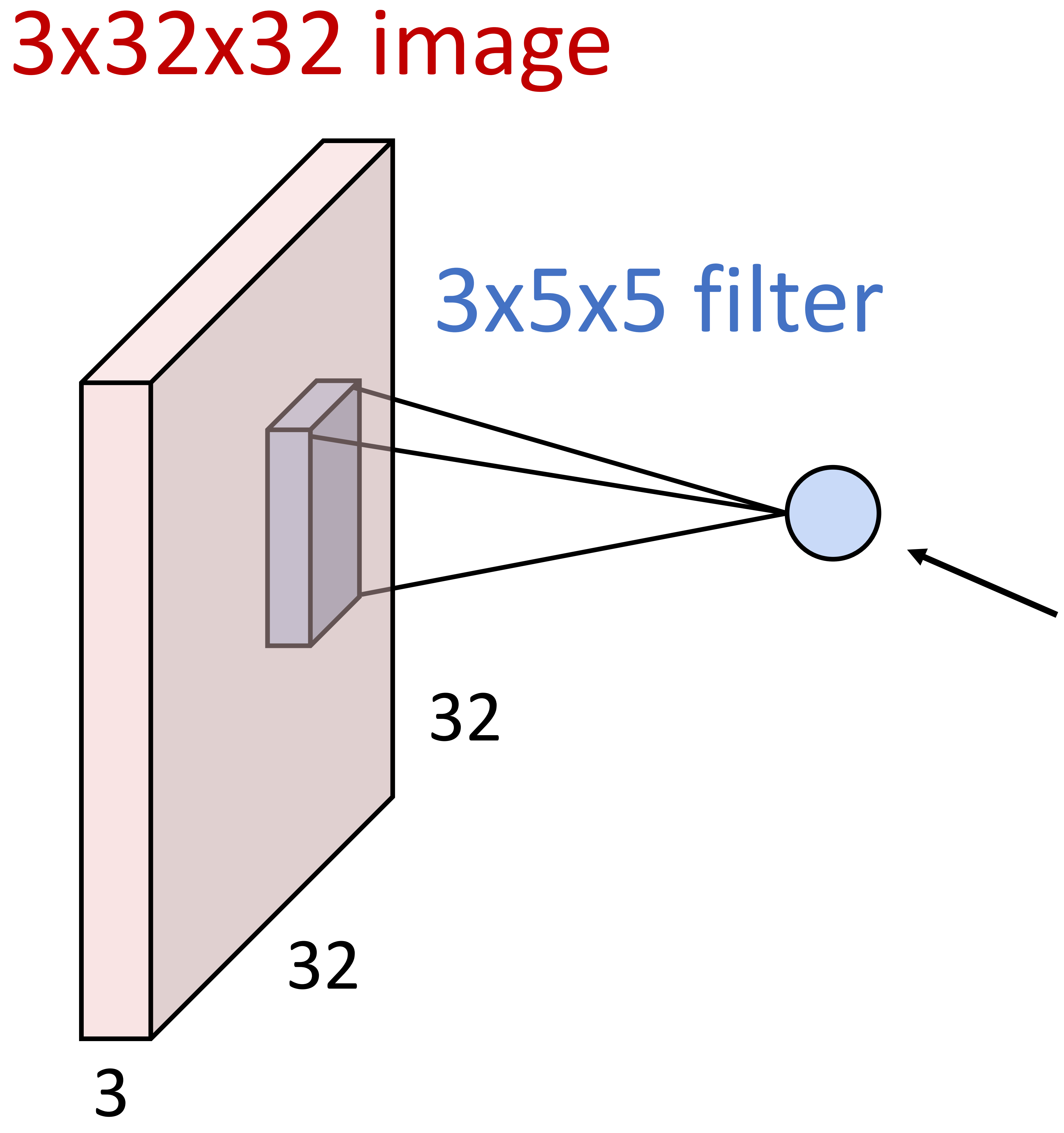


3x5x5 filter



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

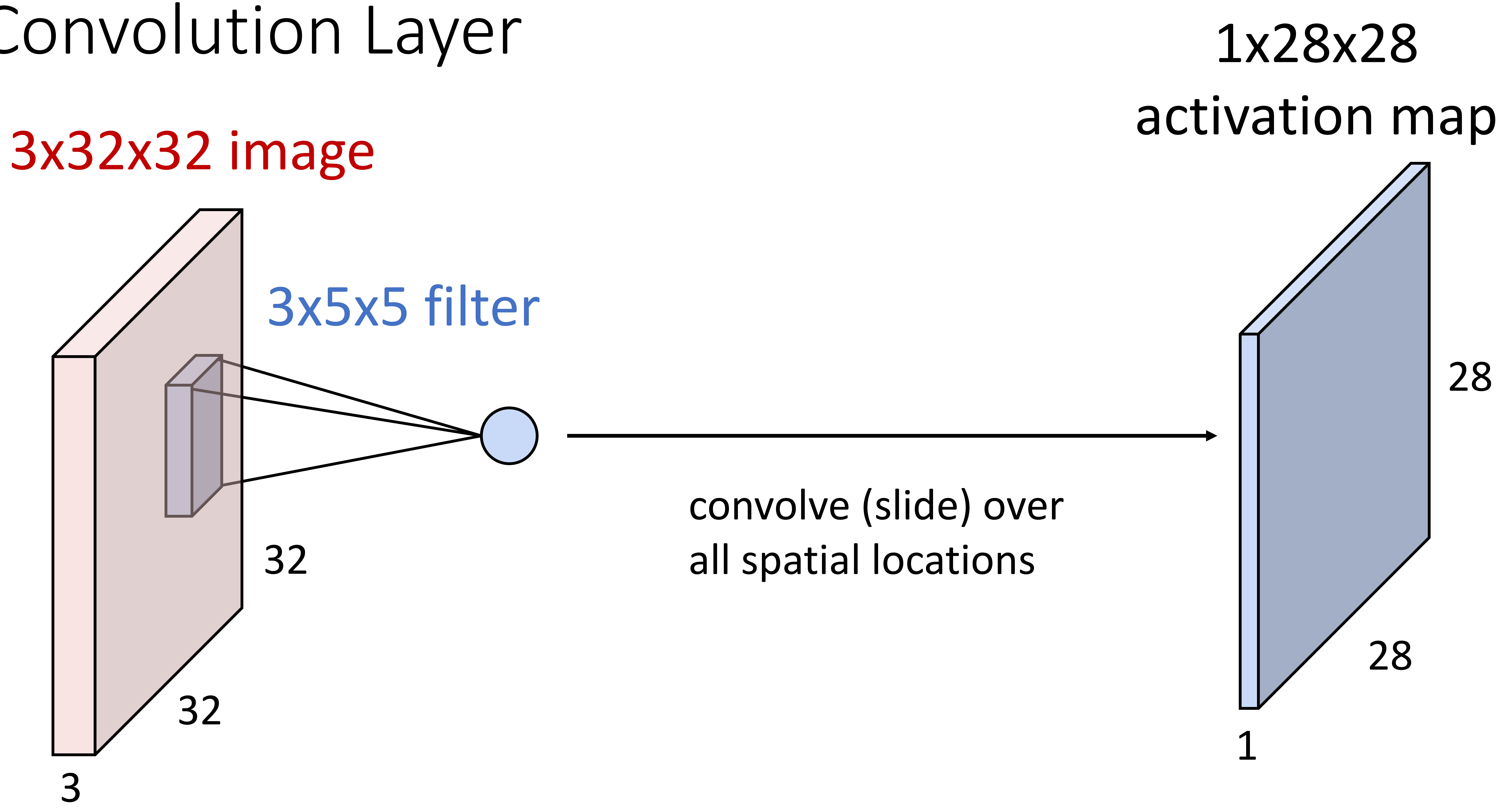
Convolution Layer



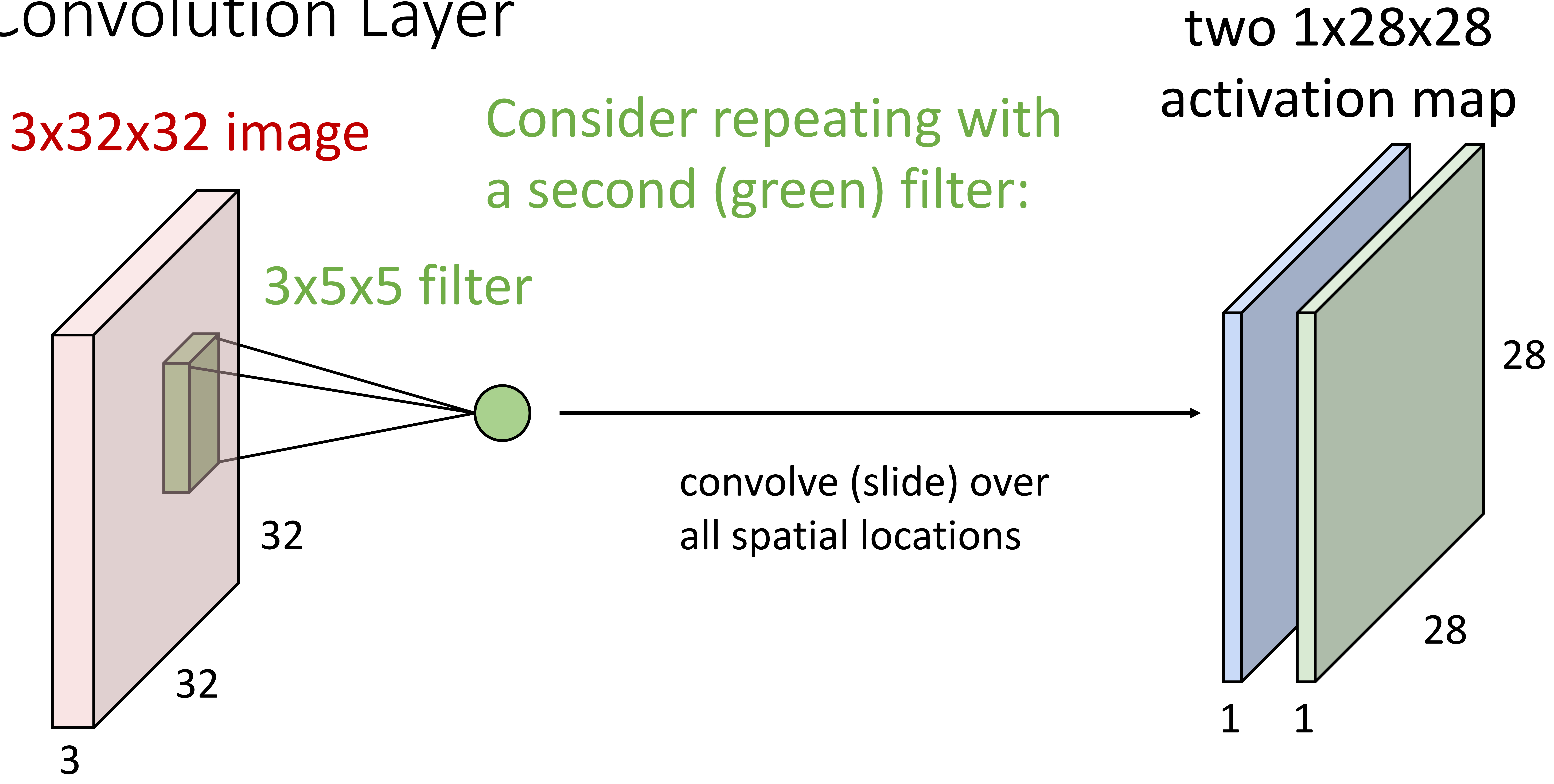
1 number:
the result of taking a dot product between the filter and a small 3x5x5 chunk of the image
(i.e. 3*5*5 = 75-dimensional dot product + bias)

$$w^T x + b$$

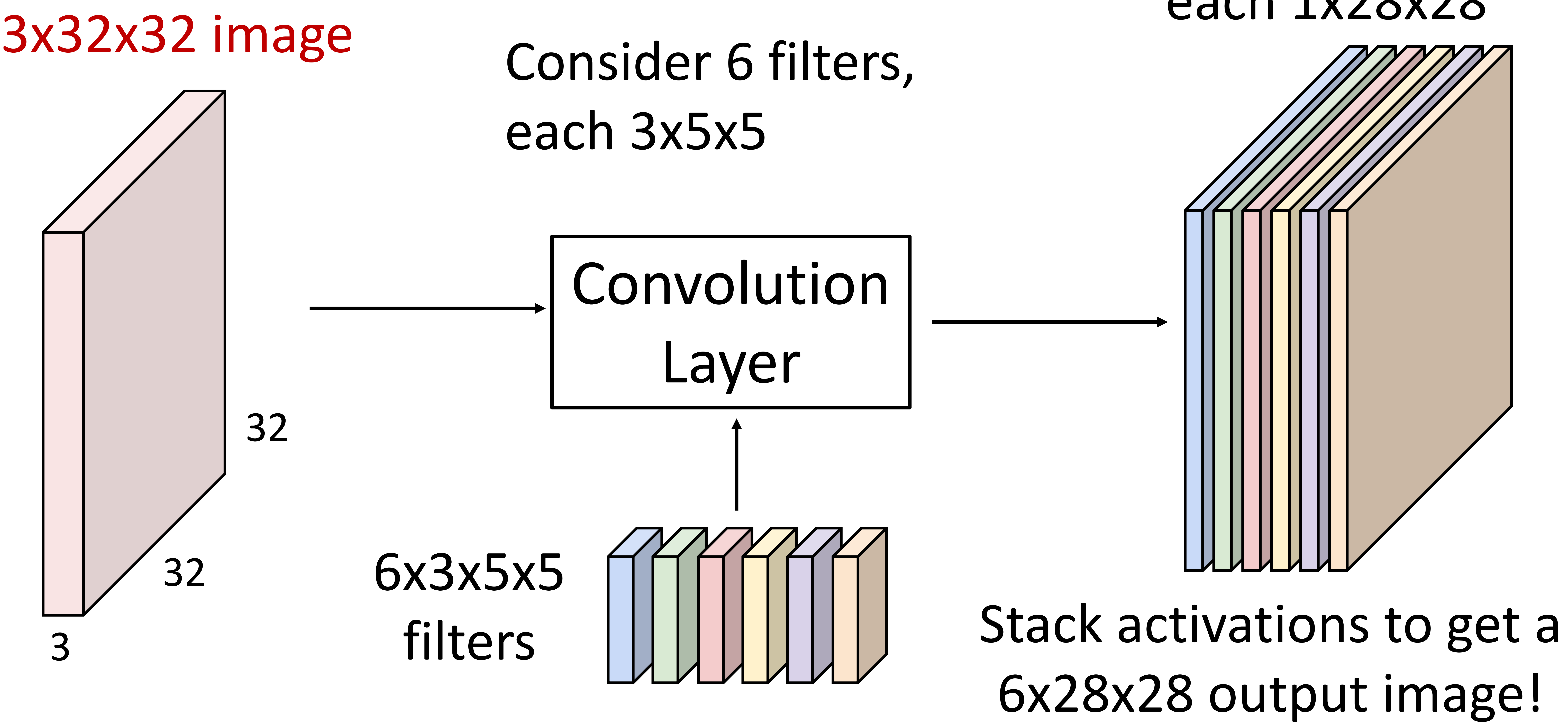
Convolution Layer



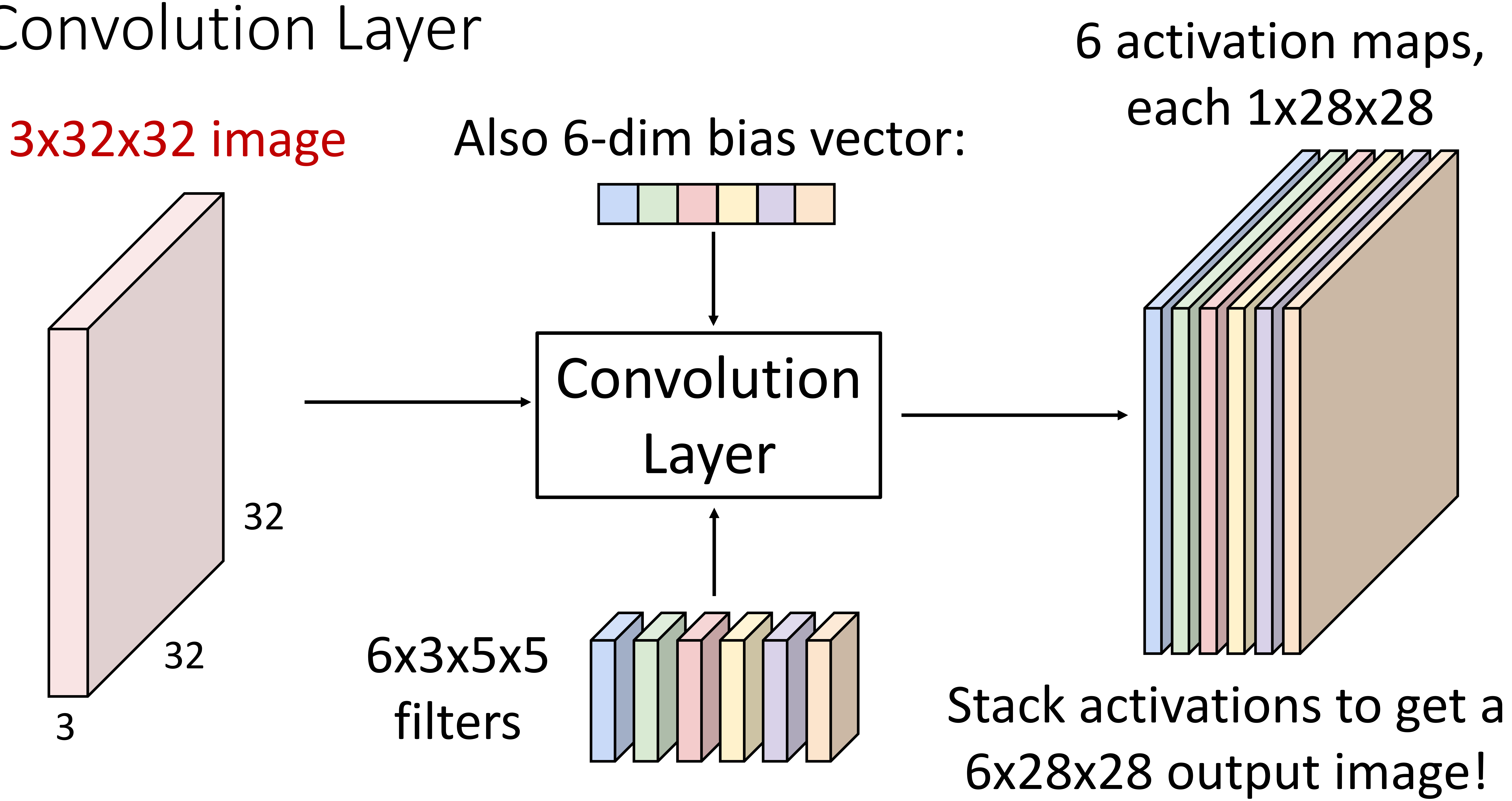
Convolution Layer



Convolution Layer



Convolution Layer

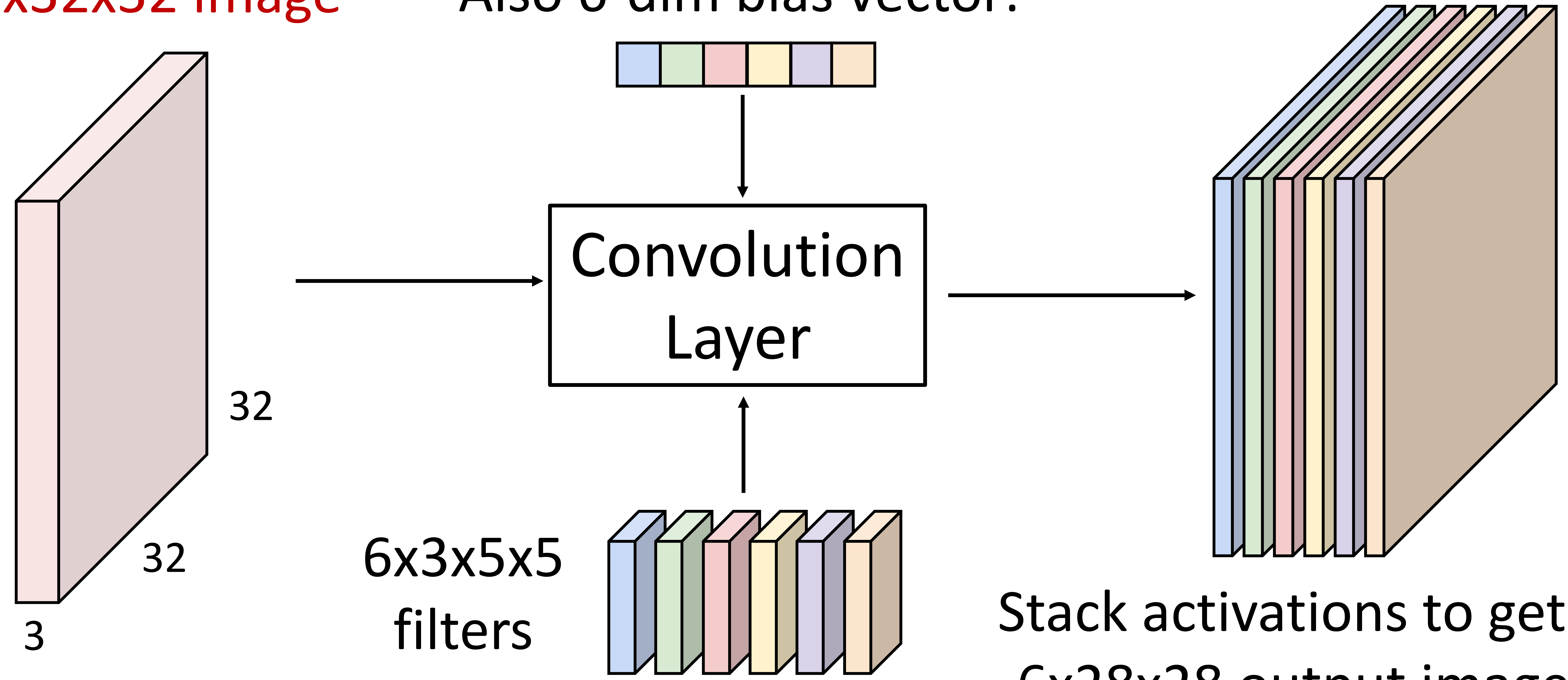


Convolution Layer

3x32x32 image

Also 6-dim bias vector:

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

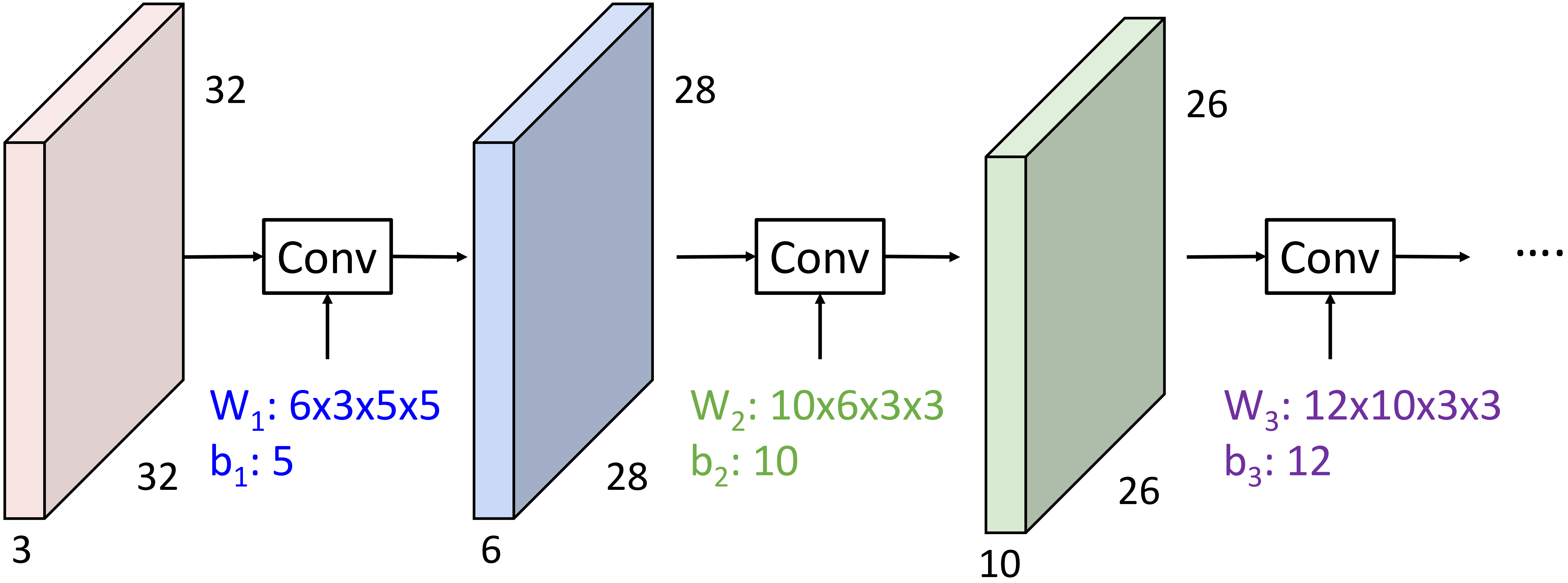


Stack activations to get a 6x28x28 output image!

Convolution Layer

Convolution Layer

Stacking Convolutions



Input:
N x 3 x 32 x 32

First hidden layer:
N x 6 x 28 x 28

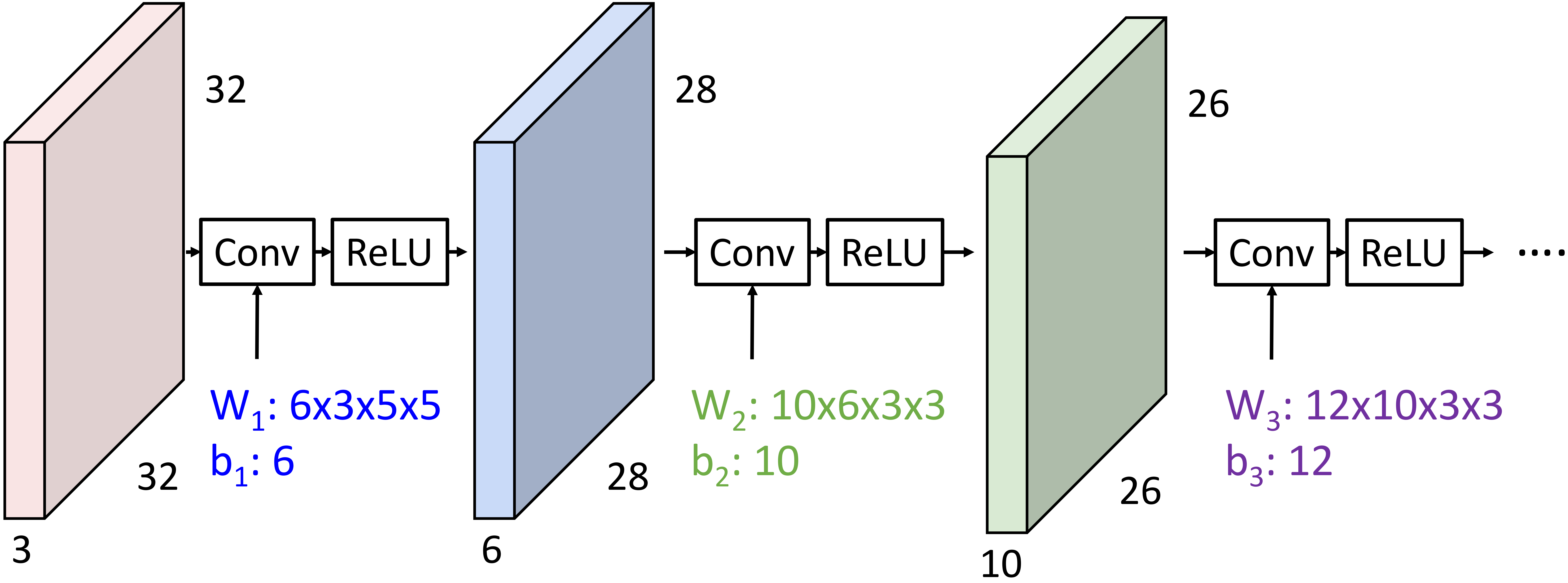
Second hidden layer:
N x 10 x 26 x 26

Stacking Convolutions

Q: What happens if we stack two convolution layers?

(Recall $y=W_2W_1x$ is a linear classifier)

A: We get another convolution!



Input:
N x 3 x 32 x 32

First hidden layer:
N x 6 x 28 x 28

Second hidden layer:
N x 10 x 26 x 26

Convolutional Neural Networks

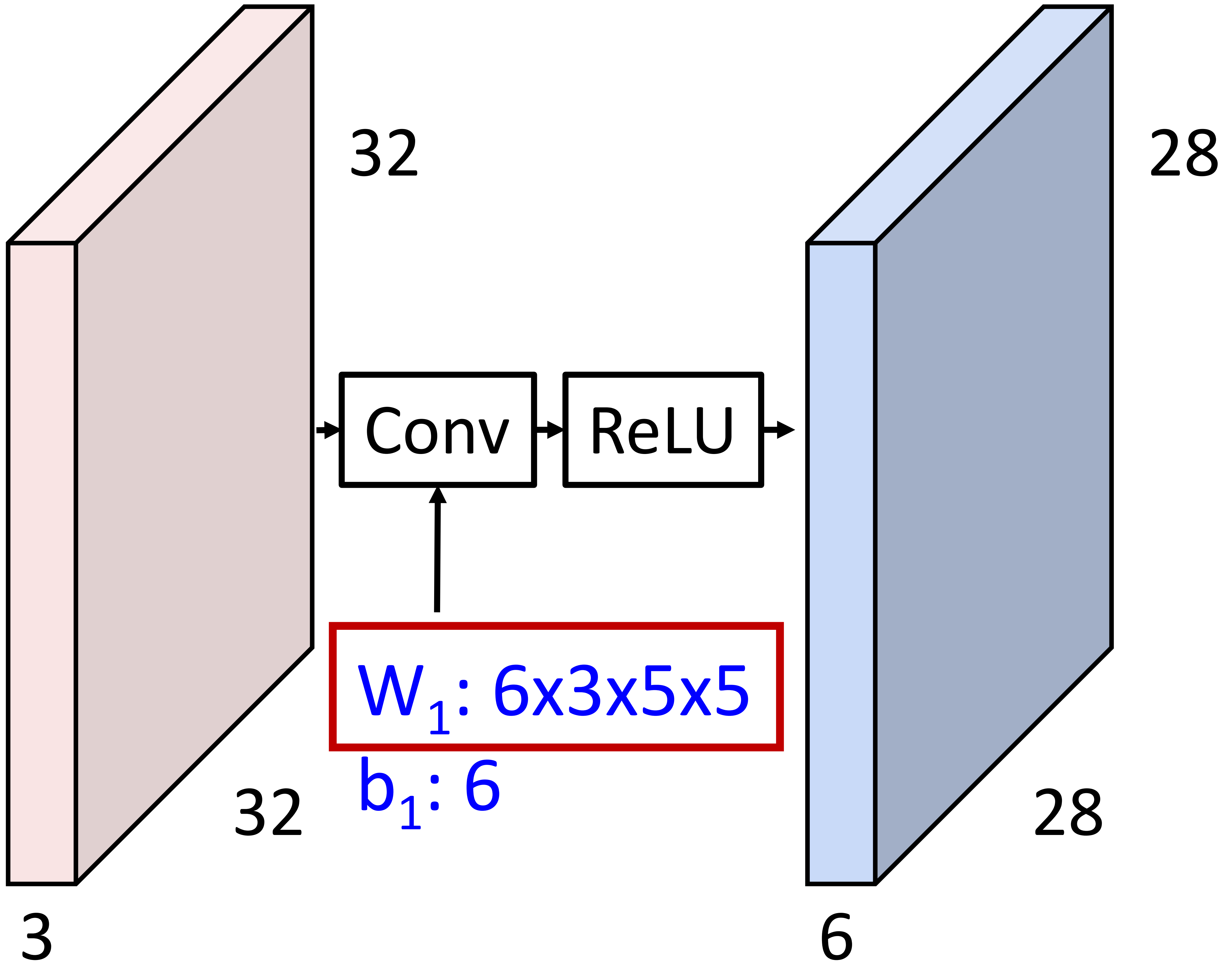
VGG-16 Network

Backward Pass for Some **Common Layers**

Convolutional layer

20.3

What do convolutional filters learn?

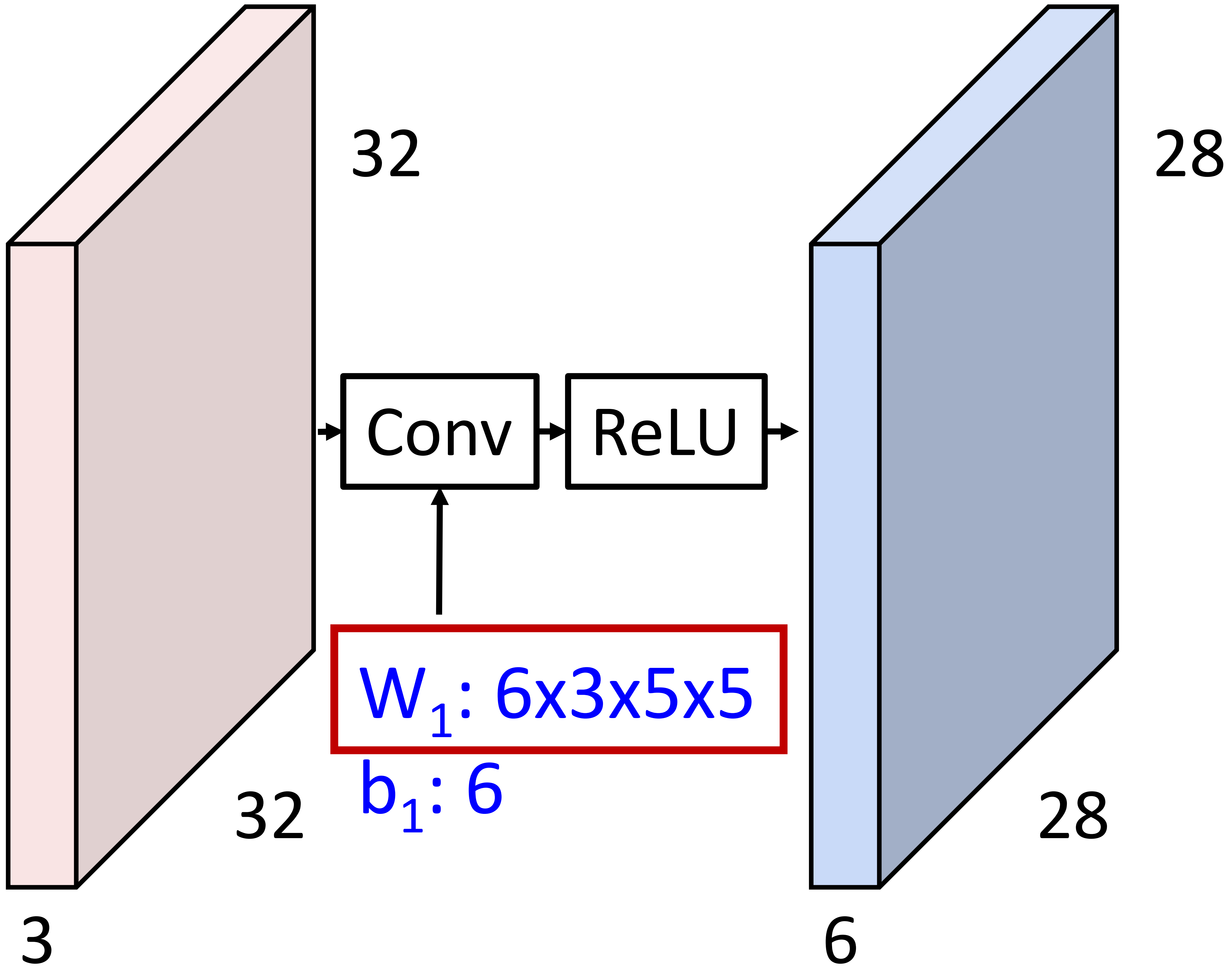


Input:
N x 3 x 32 x 32

First hidden layer:
N x 6 x 28 x 28

Linear classifier: One template per class

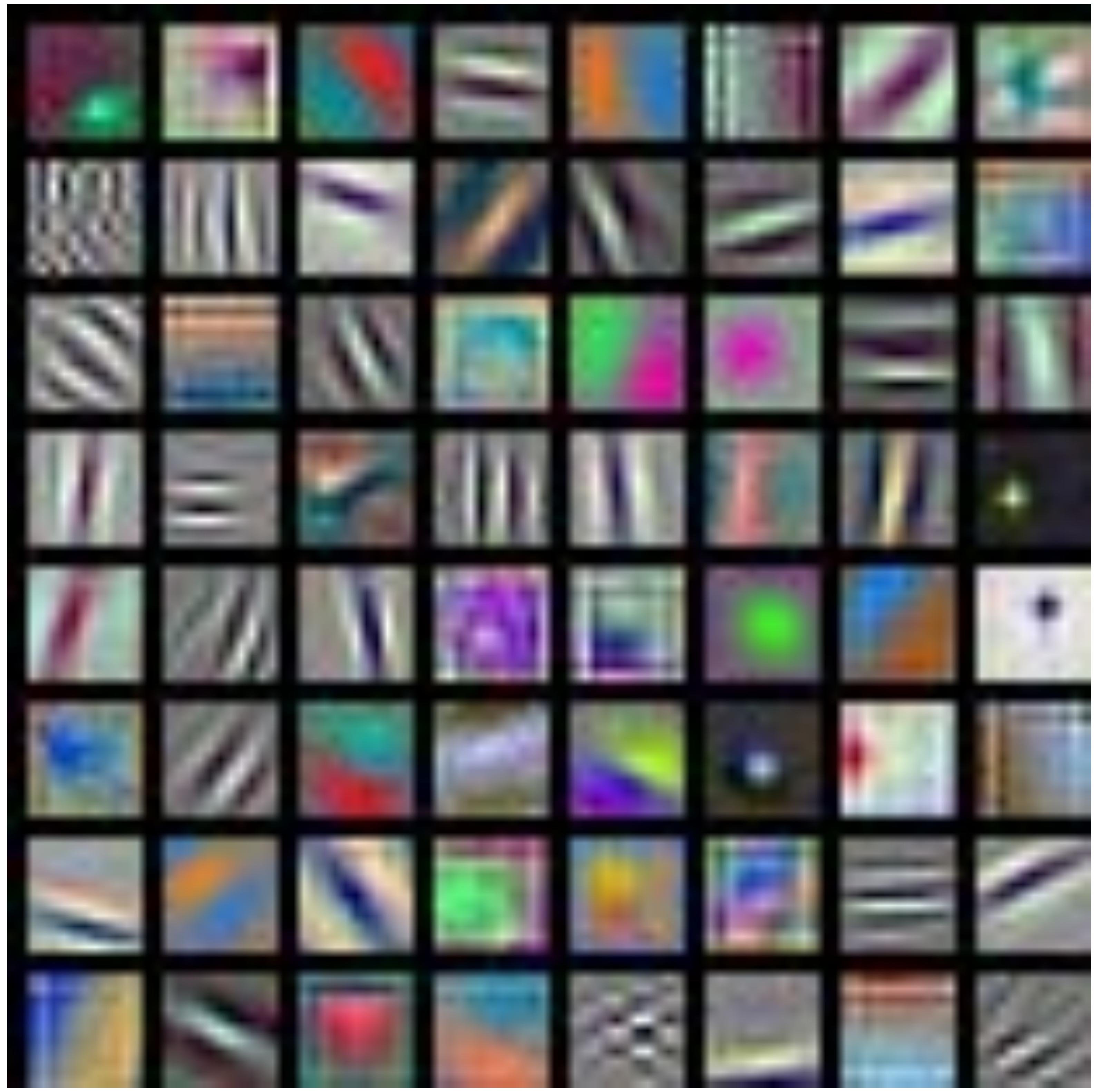
What do convolutional filters learn?



Input:
N x 3 x 32 x 32

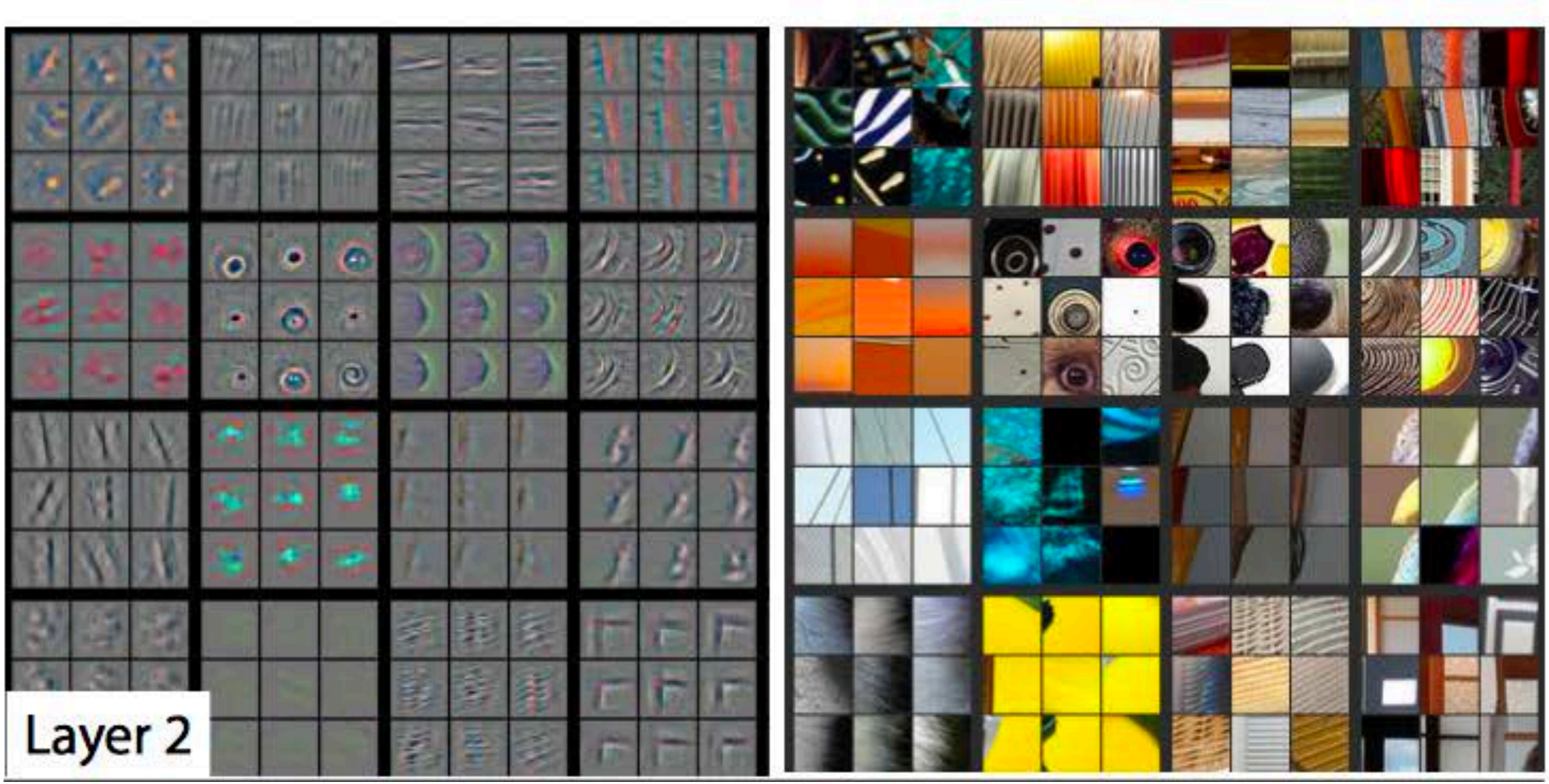
First hidden layer:
N x 6 x 28 x 28

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

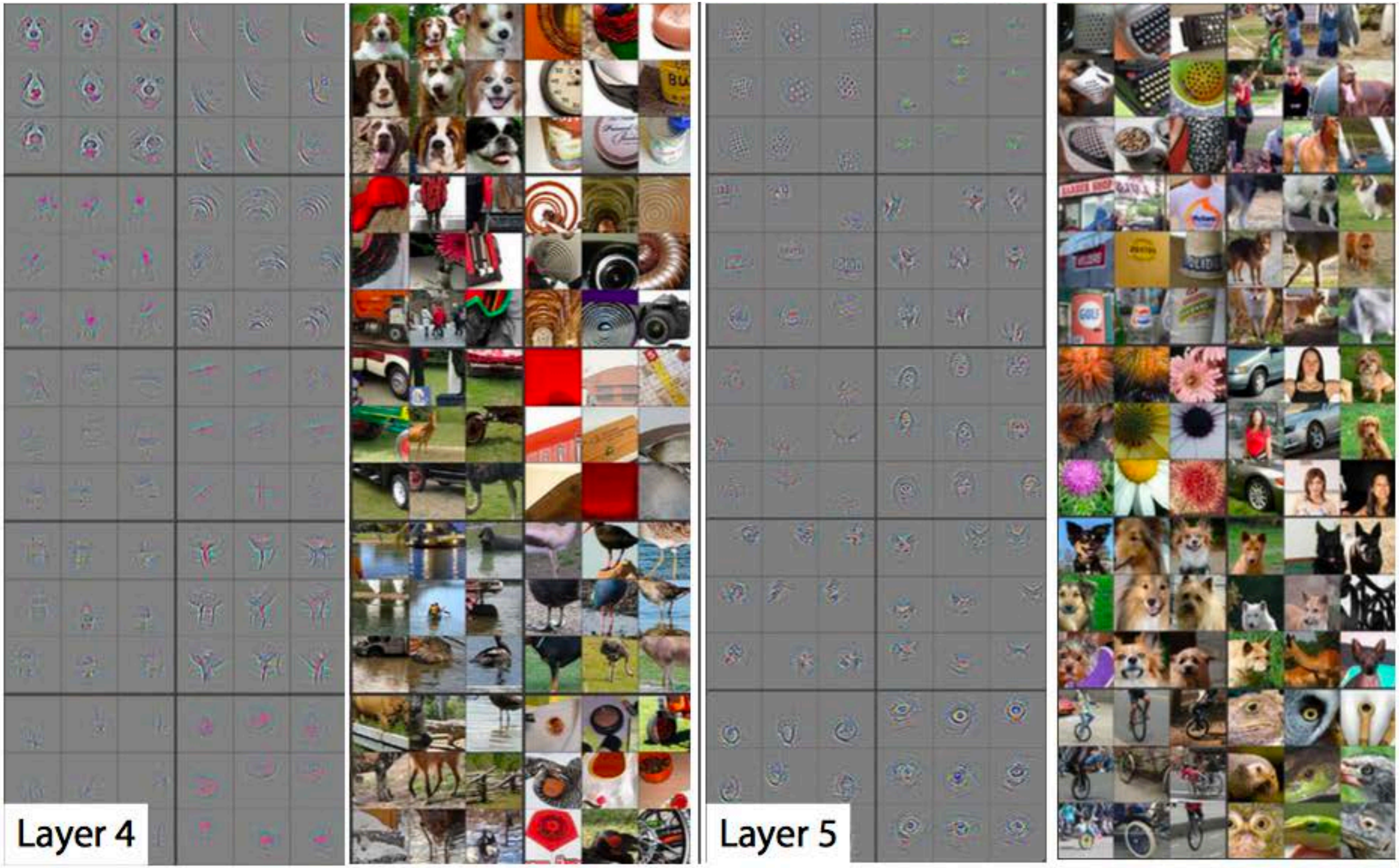


AlexNet: 64 filters, each 3x11x11

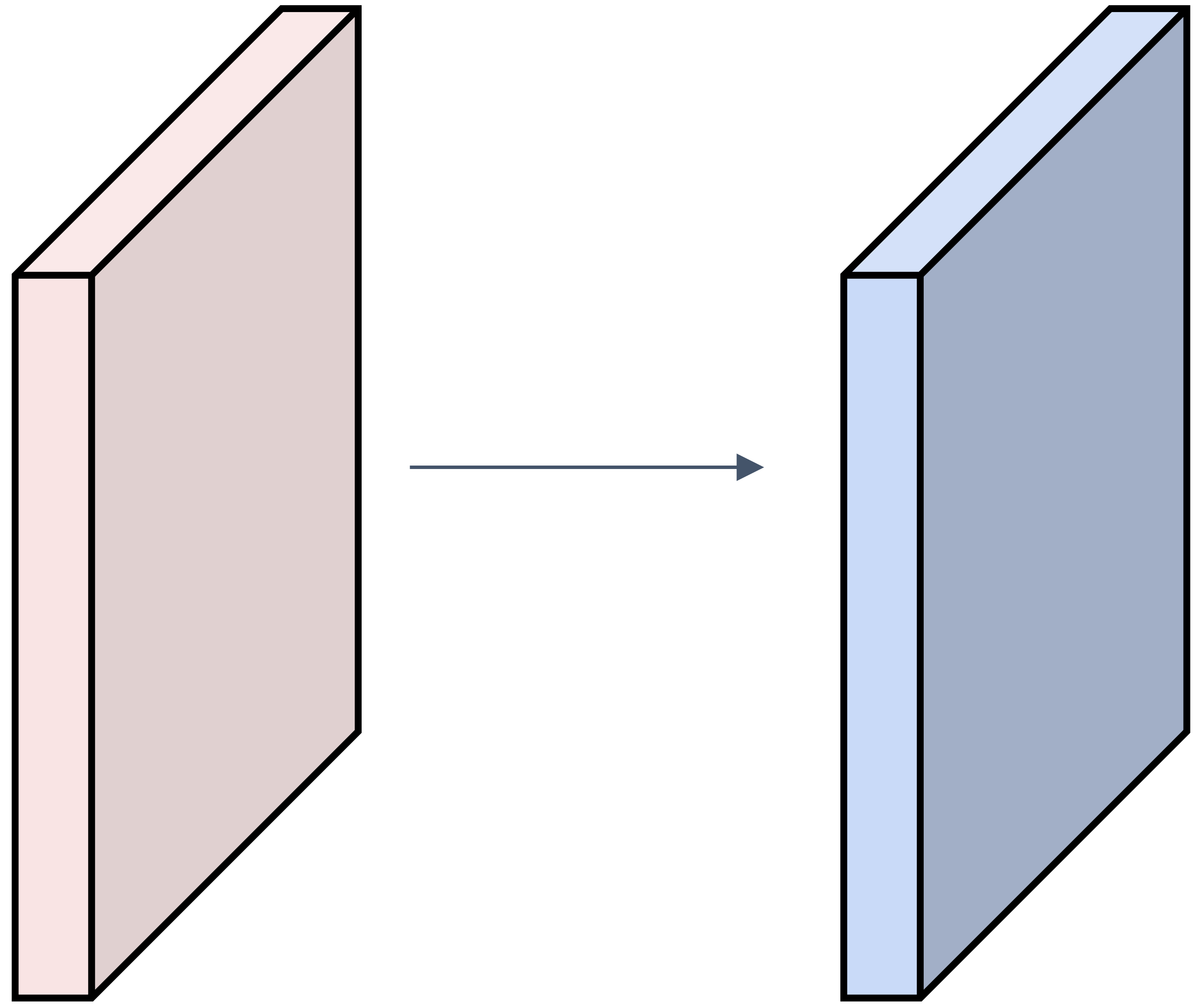
What **filters** do networks learn?



[Zeiler and Fergus, 2013]


Layer 4
Layer 5

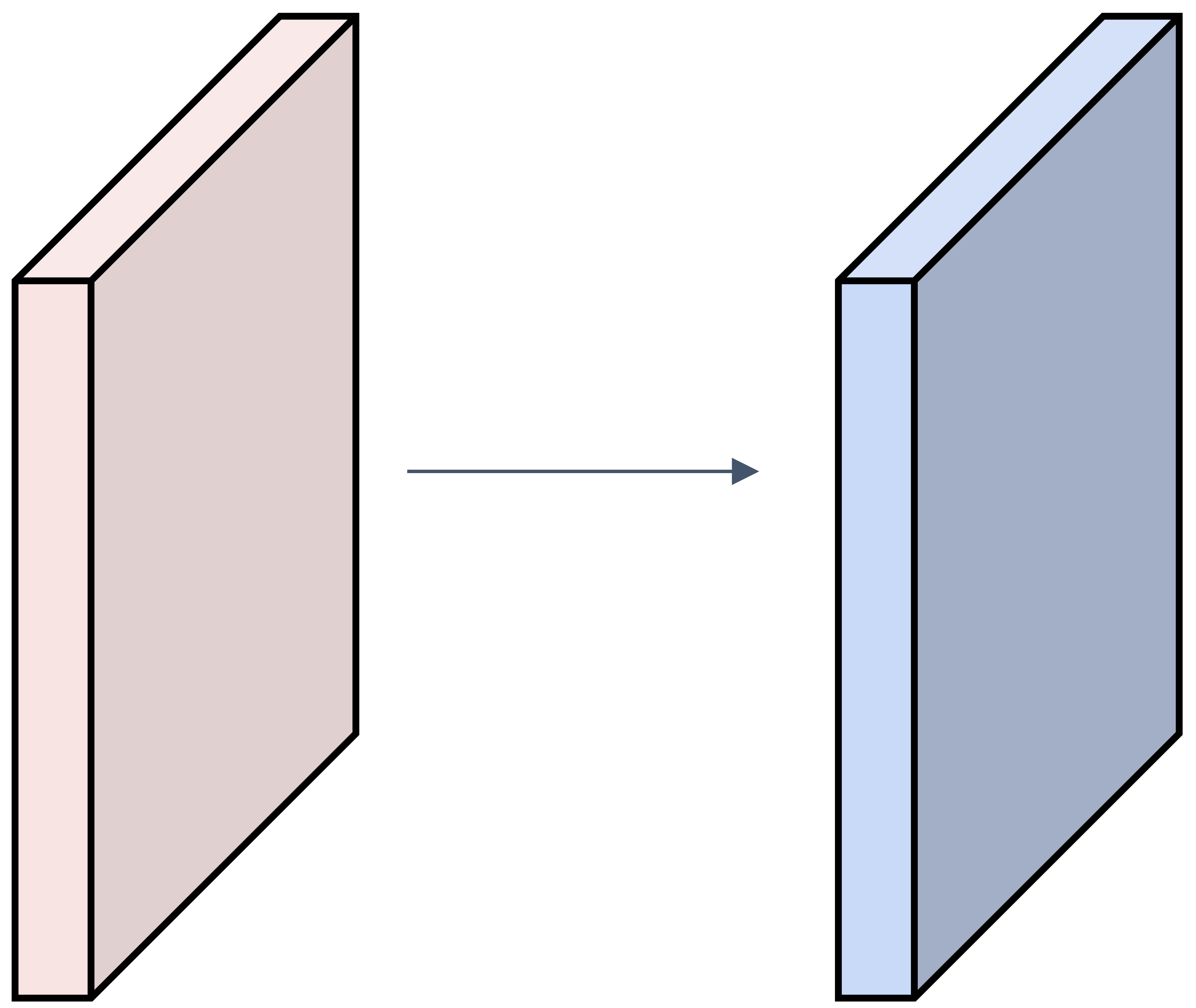
Convolution Example



Input volume: 3 x 32 x 32
10 5x5 filters with stride 1, pad 2

Output volume size: ?

Convolution Example



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

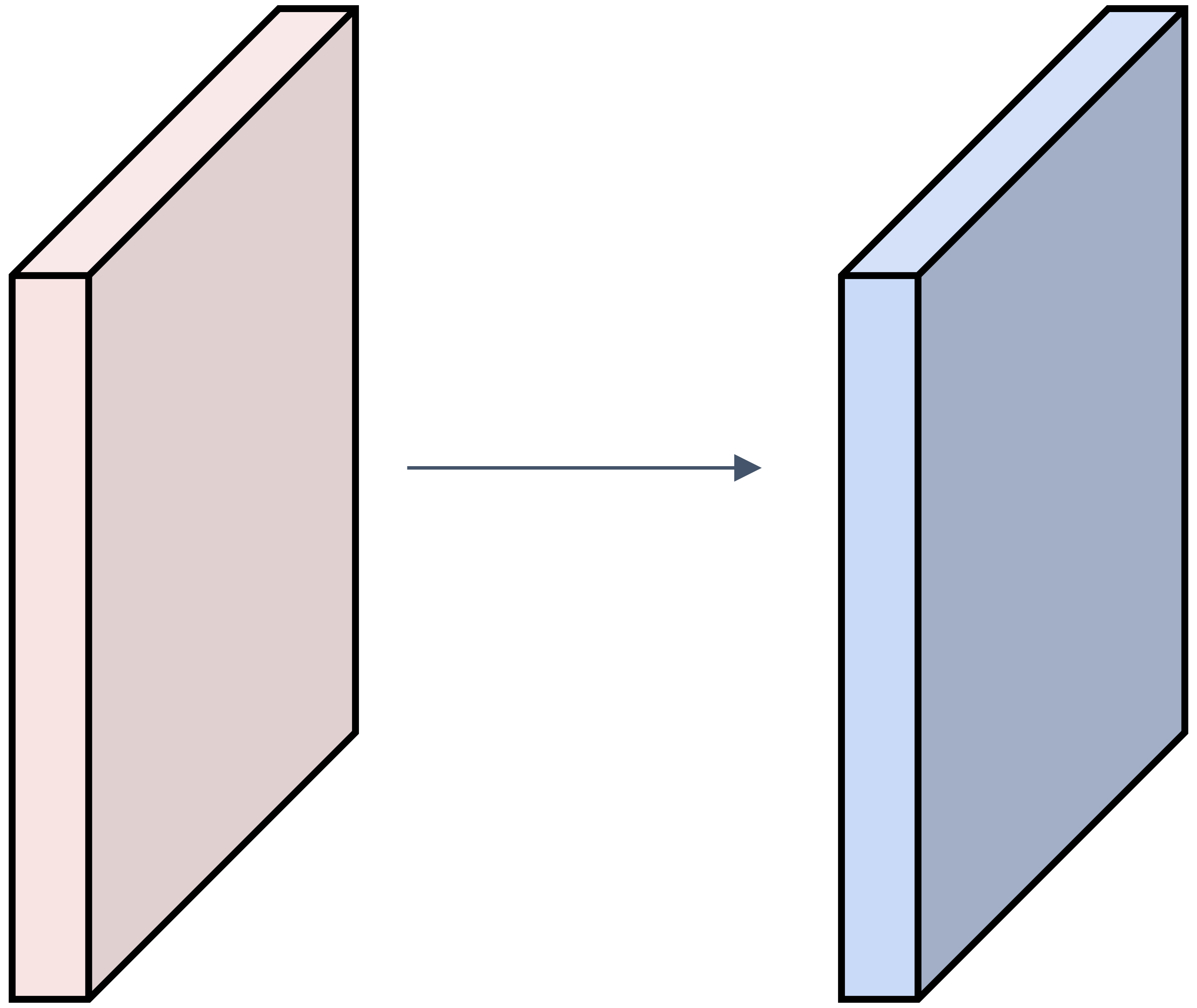
10 5x5 filters with stride **1**, pad **2**

Output volume size:

(**32**+2***2**-**5**)/**1**+1 = 32 spatially, so

10 x 32 x 32

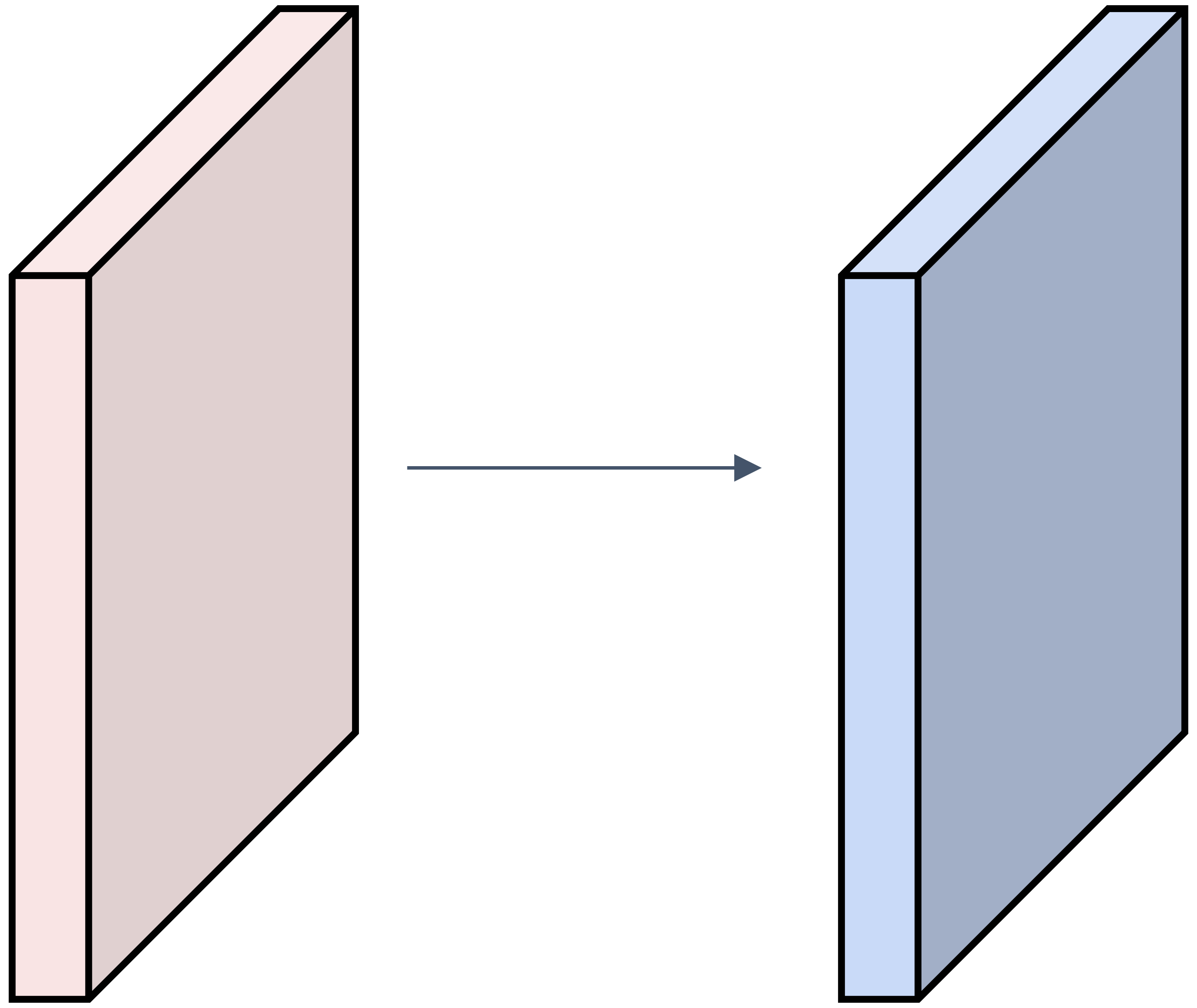
Convolution Example



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: ?

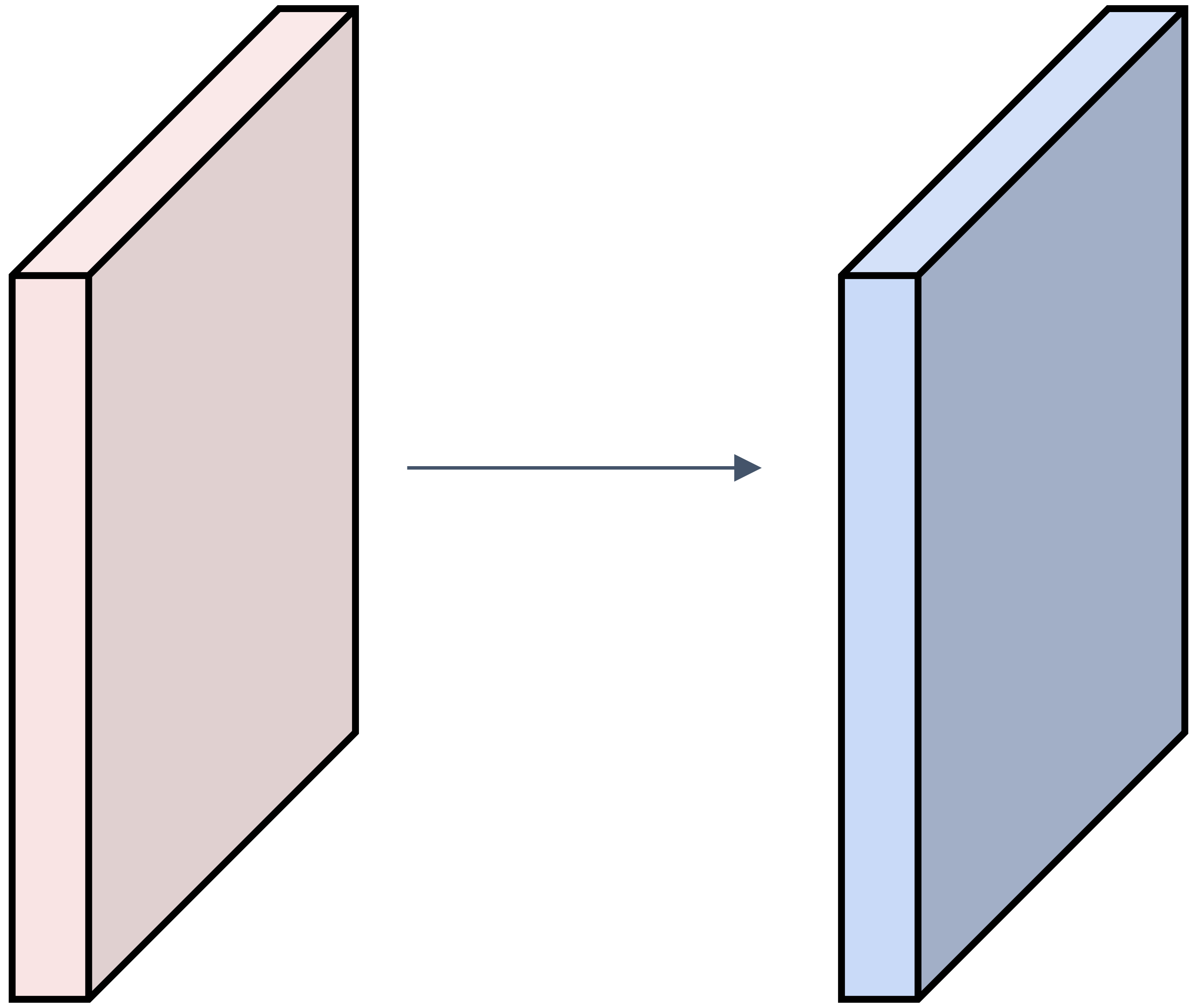
Convolution Example



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**

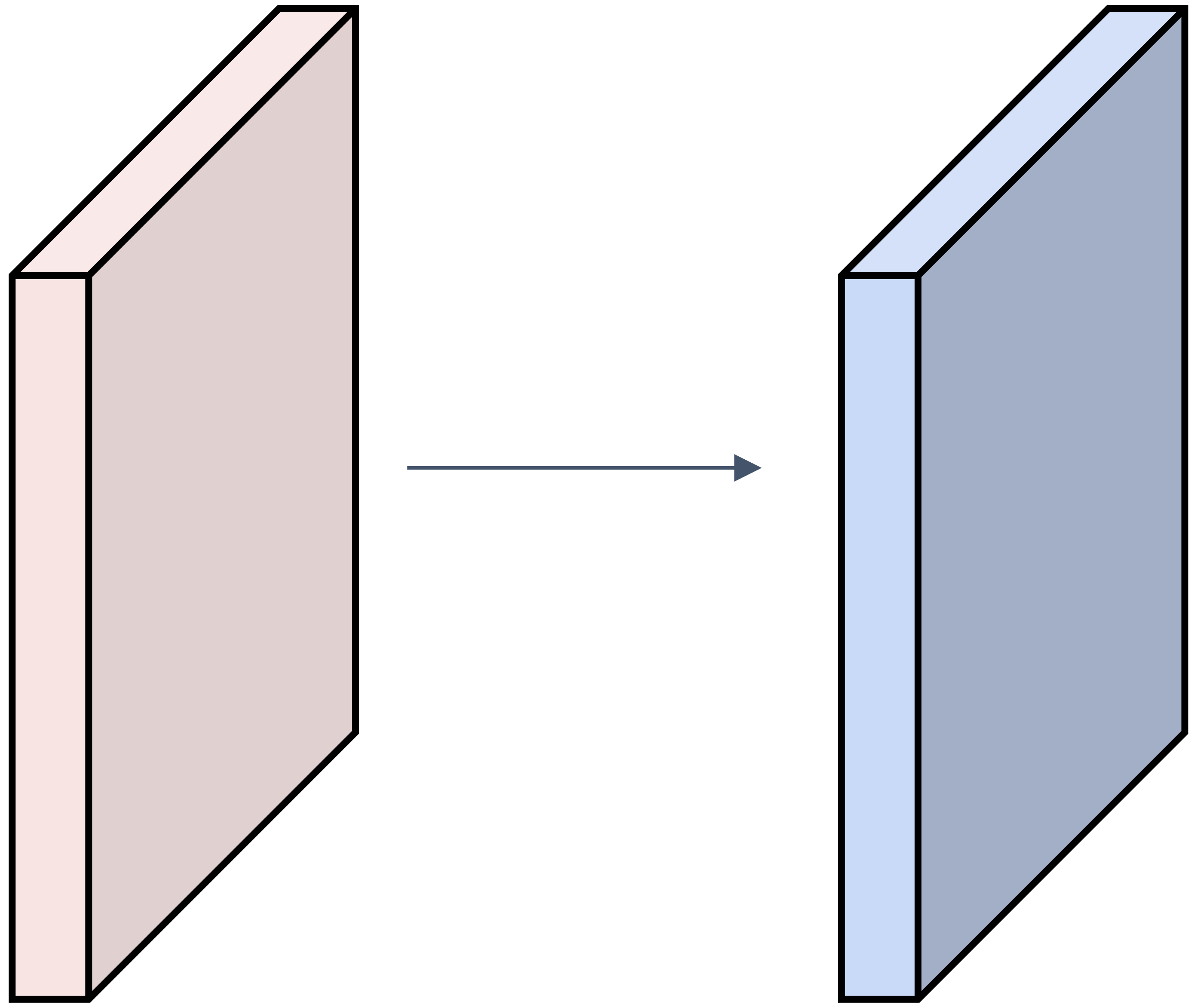
Convolution Example



Input volume: 3 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
Number of multiply-add operations: ?

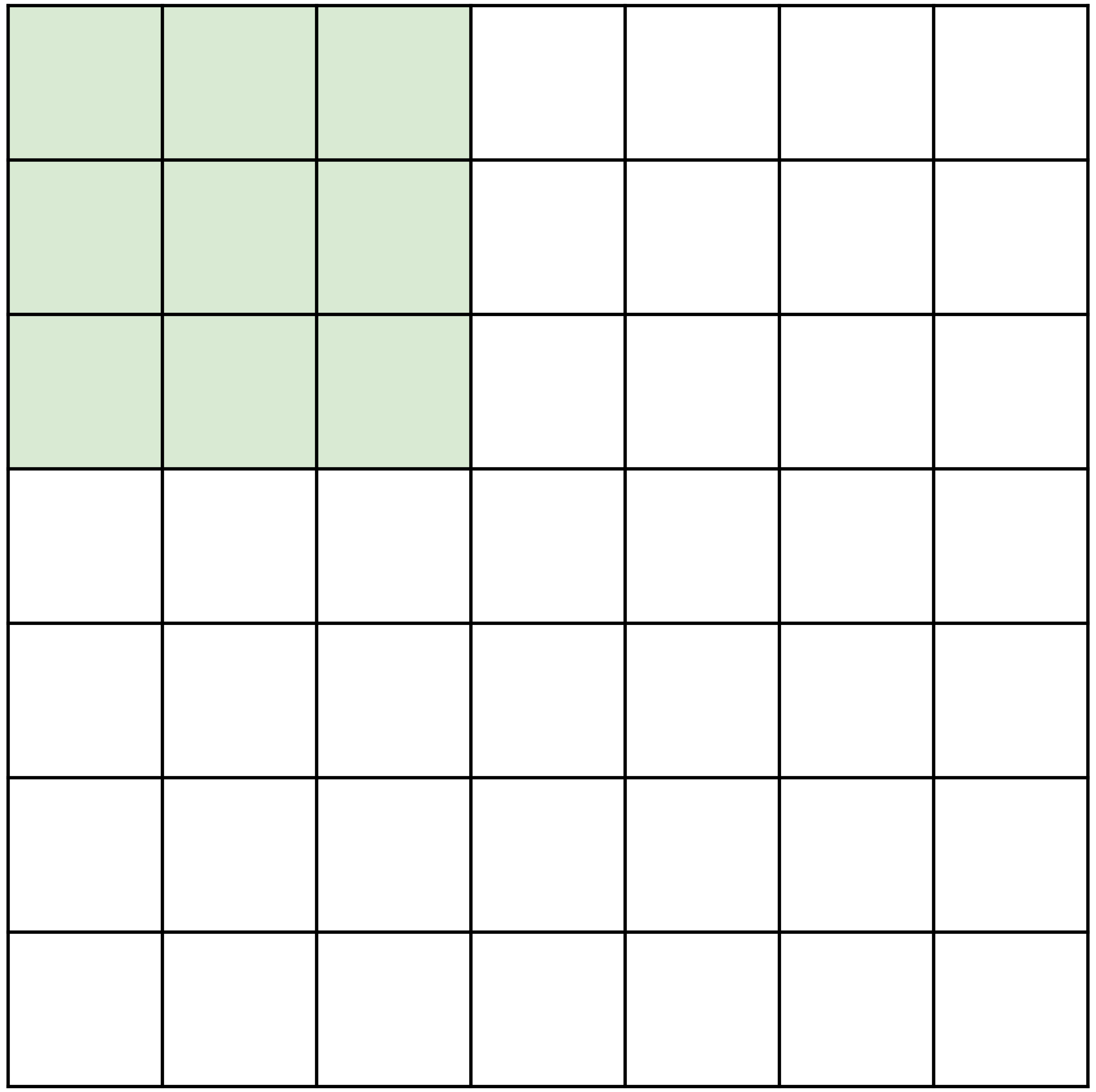
Convolution Example



Input volume: **3** x 32 x 32
10 **5x5** filters with stride 1, pad 2

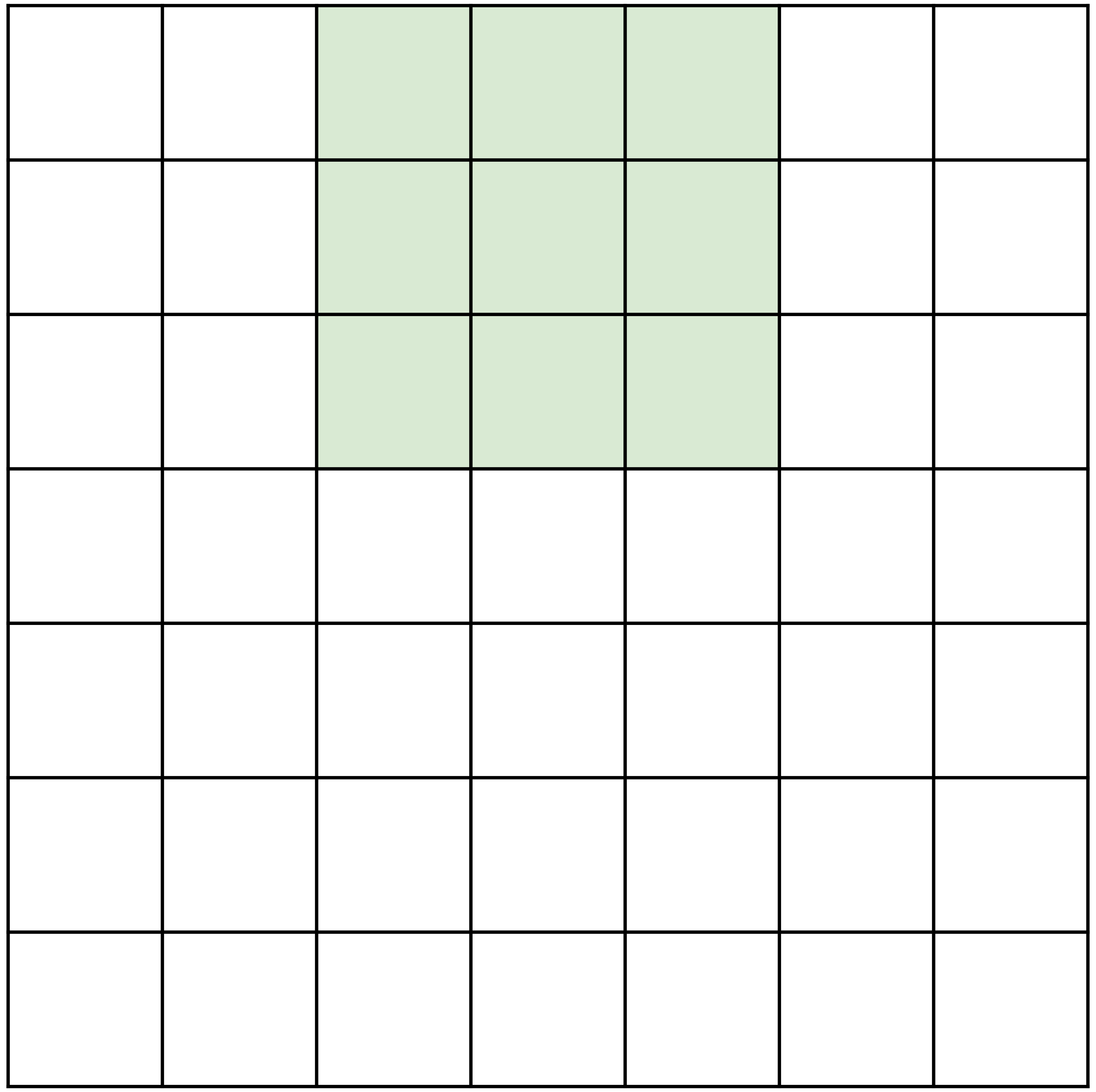
Output volume size: **10 x 32 x 32**
Number of learnable parameters: 760
Number of multiply-add operations: **768,000**
10*32*32 = 10,240 outputs; each output is the inner product of two **3**x**5x5** tensors (75 elems); total = 75*10240 = **768K**

Strided Convolution



Input: 7x7
Filter: 3x3
Stride: 2

Strided Convolution

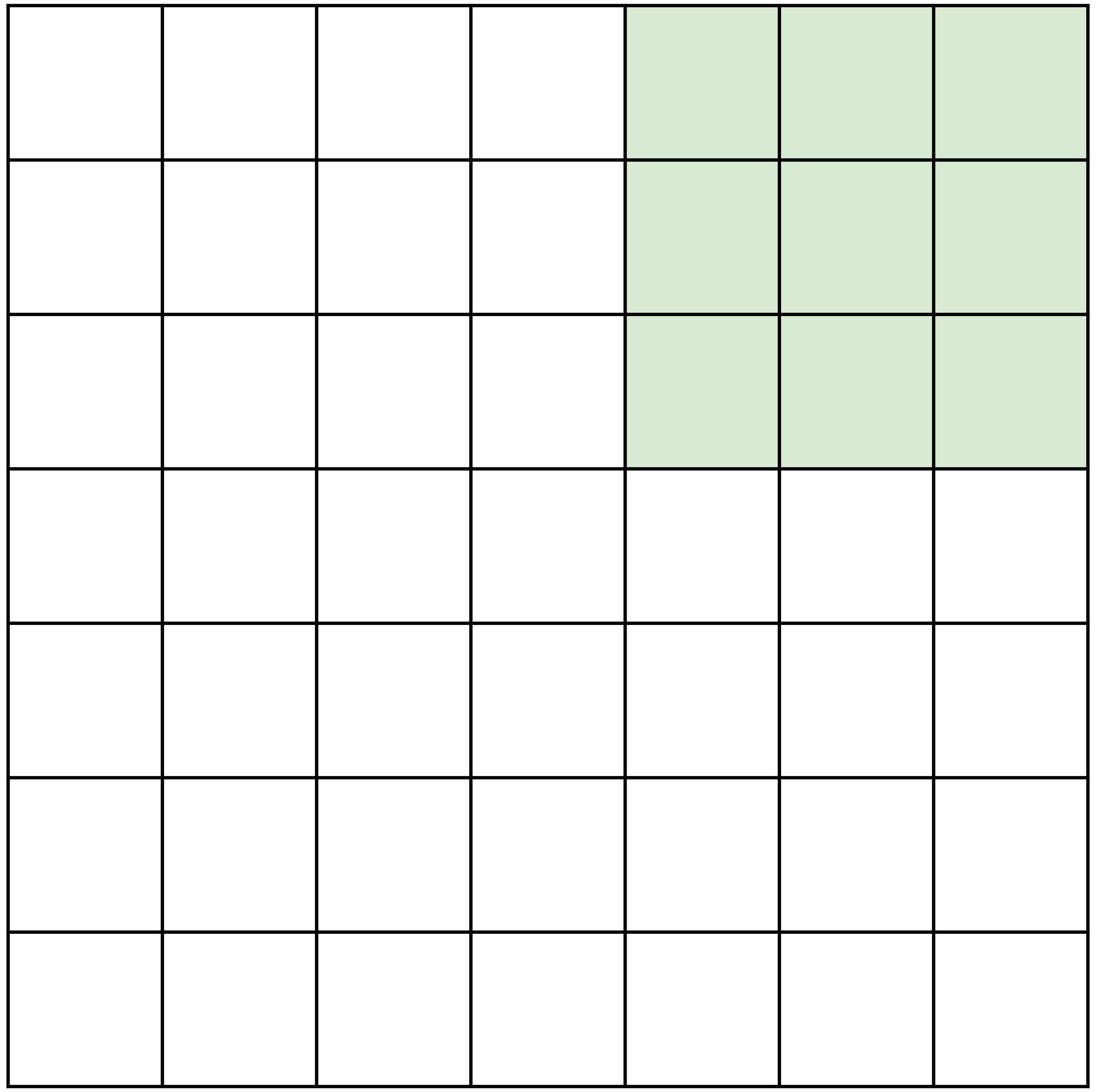


Input: 7x7

Filter: 3x3

Stride: 2

Strided Convolution



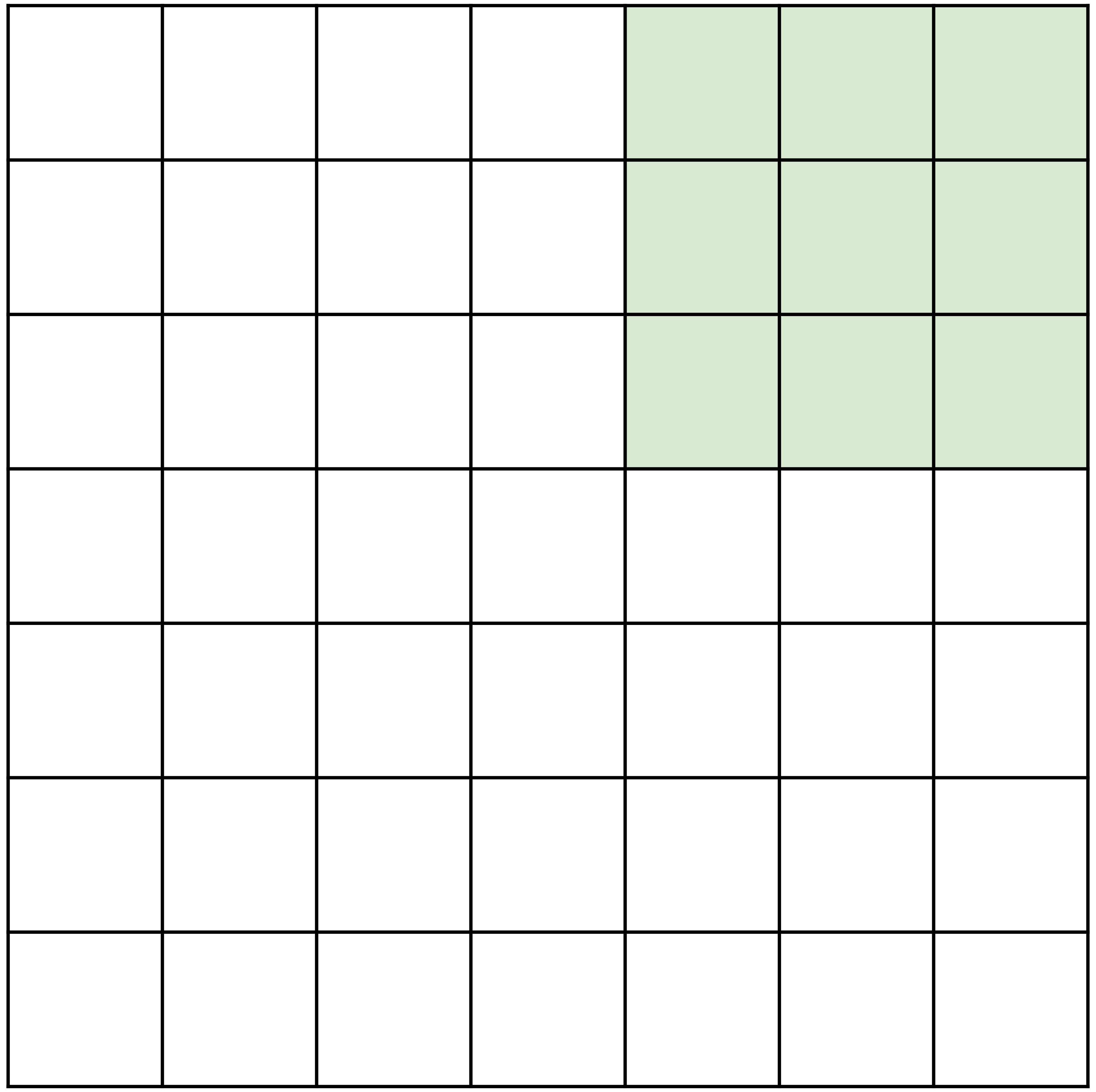
Input: 7x7

Filter: 3x3

Stride: 2

Output: 3x3

Strided Convolution

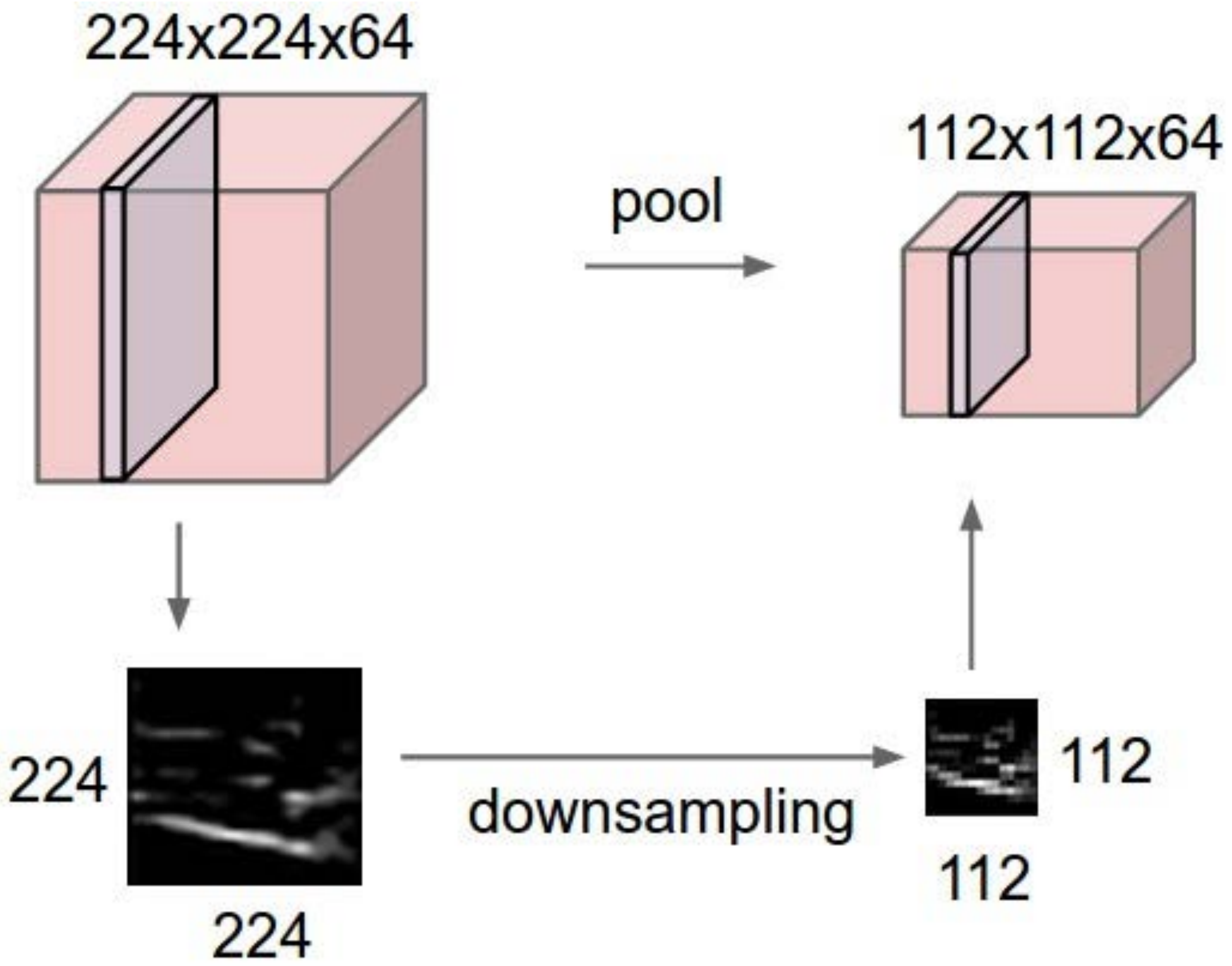


Input: 7x7
Filter: 3x3
Stride: 2

Output: 3x3

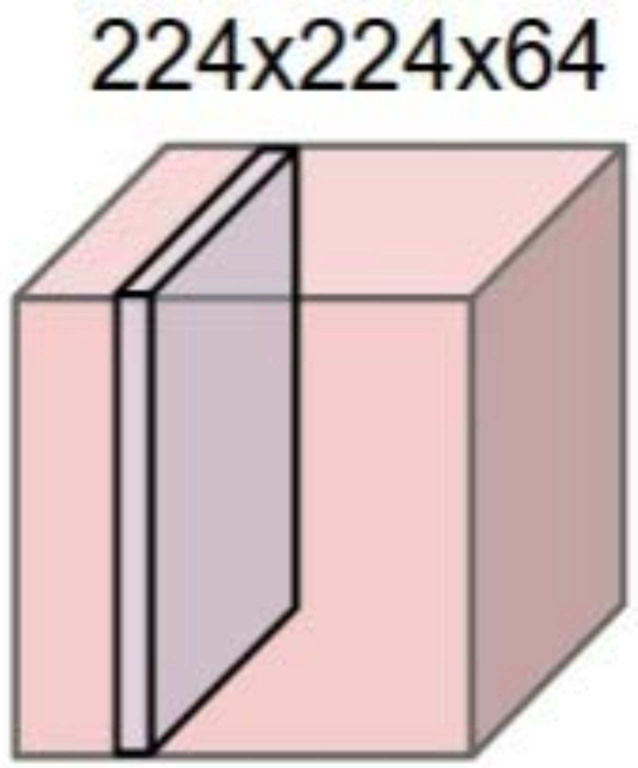
In general:
Input: W
Filter: K
Padding: P
Stride: S
Output: (W – K + 2P) / S + 1

Pooling Layers: Another way to downsample



Hyperparameters:
Kernel Size
Stride
Pooling function

Max Pooling



Single depth slice

x ↑

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

→ y

Max pooling with 2x2 kernel size and stride 2

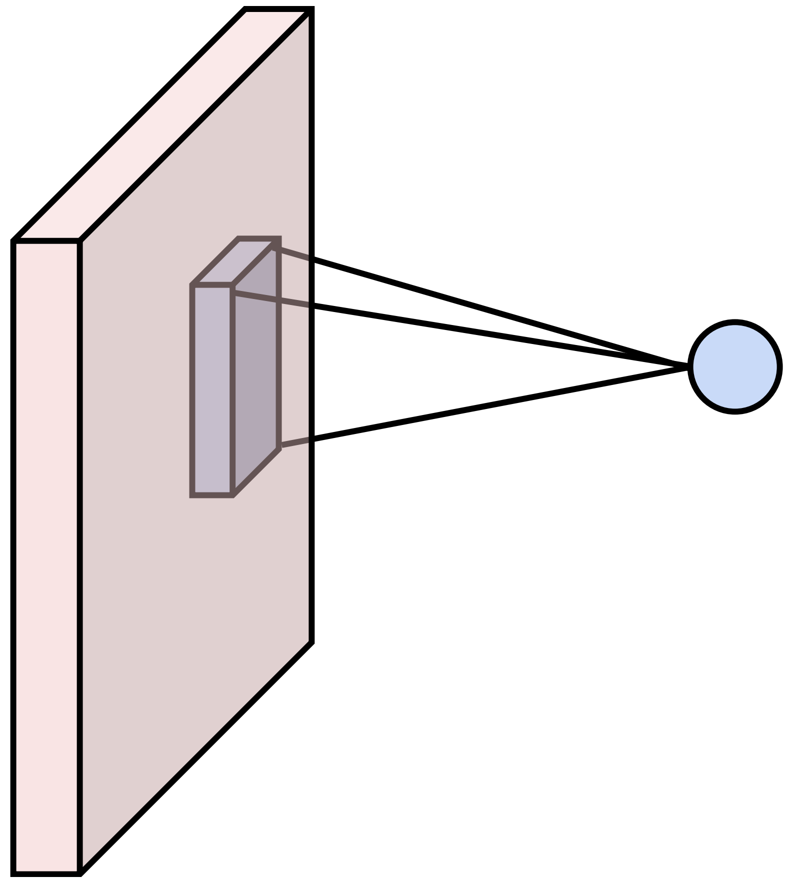
6	8
3	4

Introduces **invariance** to small spatial shifts

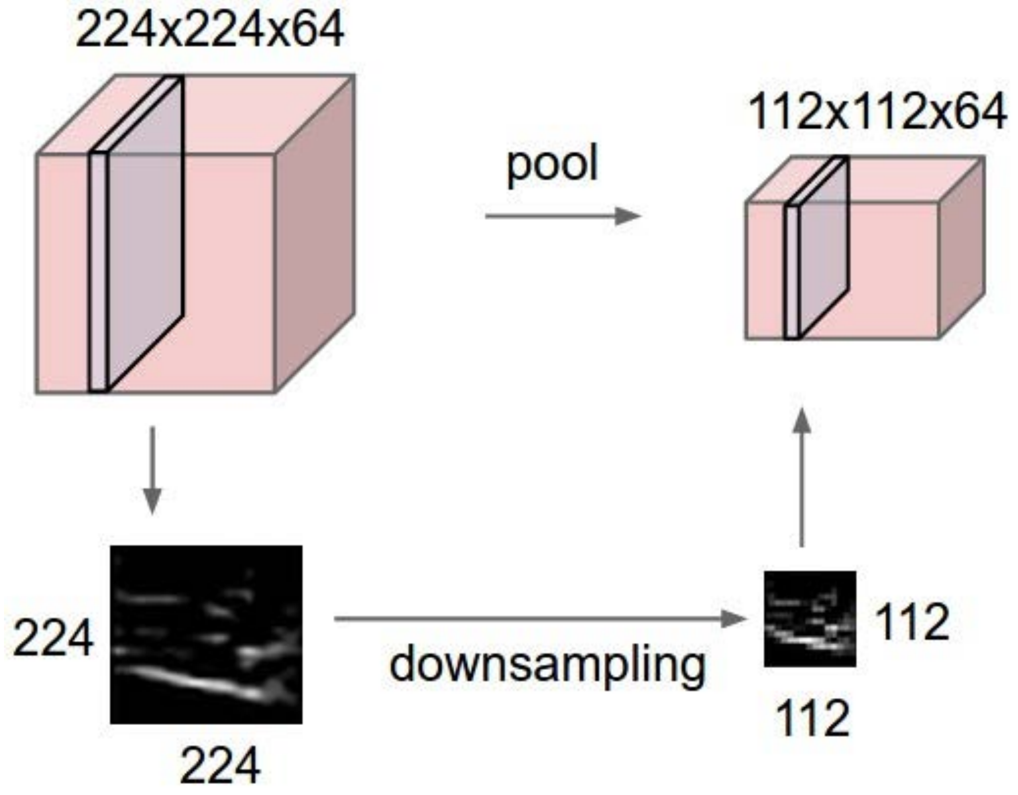
No learnable parameters!

Components of a Convolutional Network

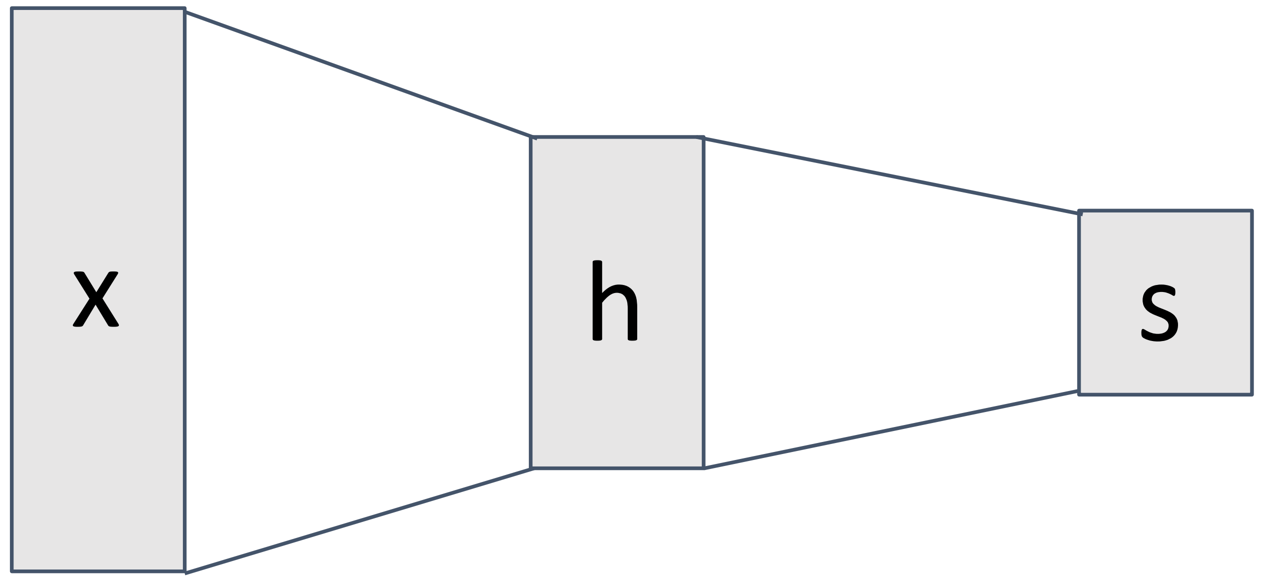
Convolution Layers



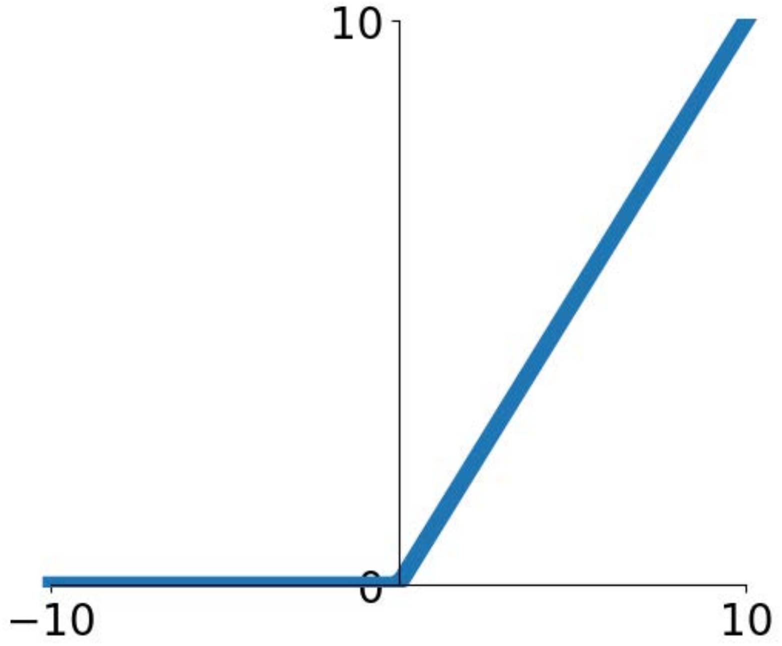
Pooling Layers



Fully-Connected Layers



Activation Function



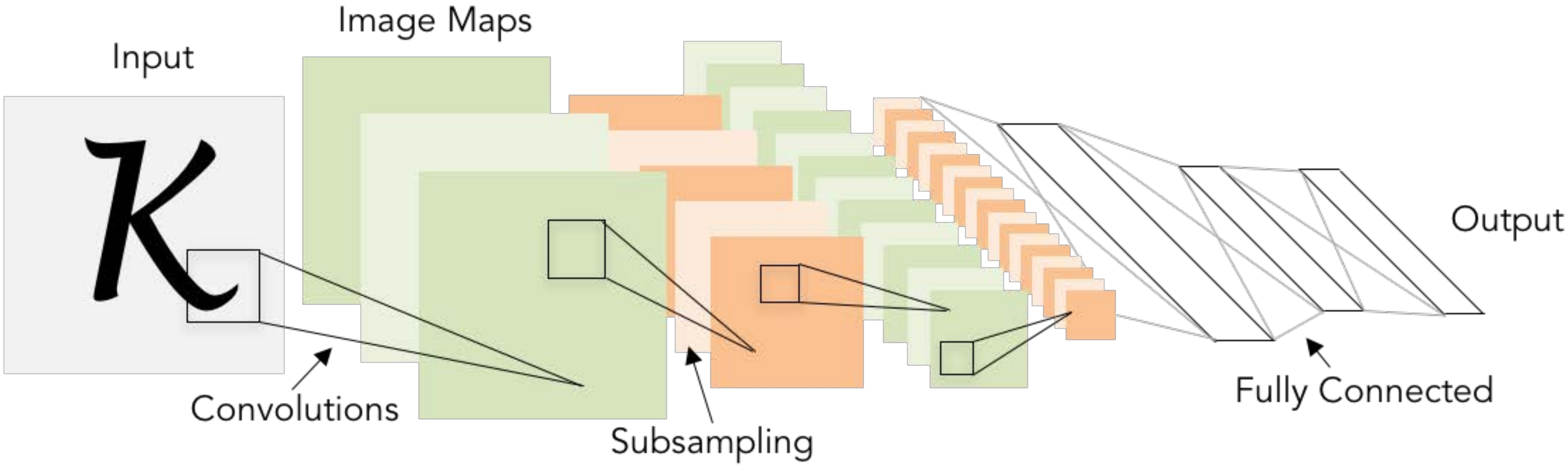
Normalization

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Convolutional Networks

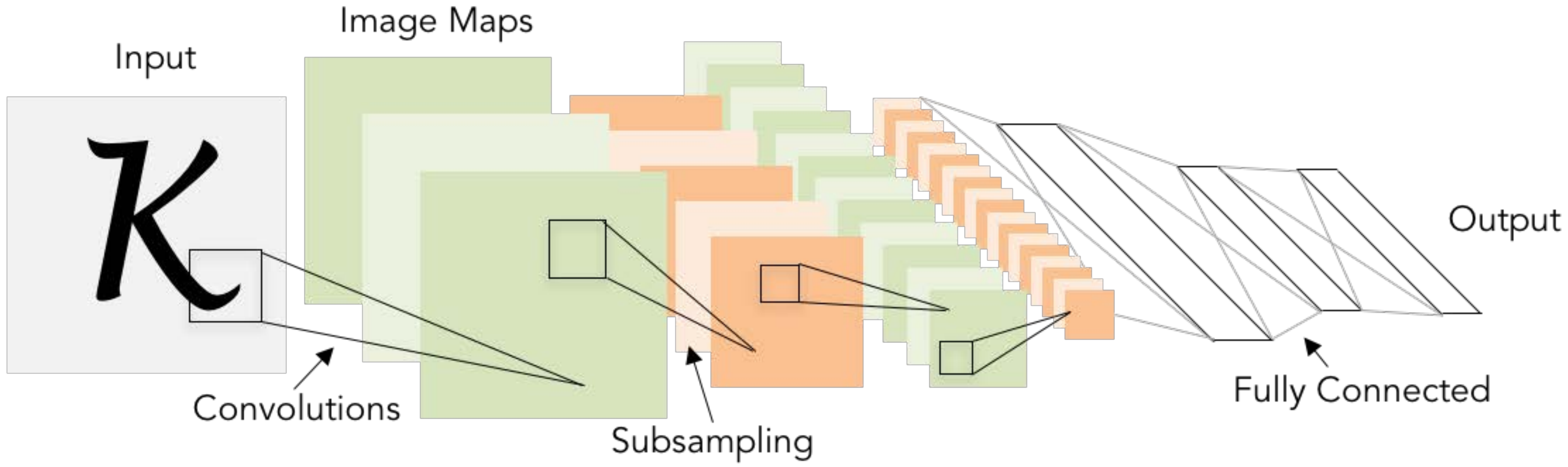
Classic architecture: [Conv, ReLU, Pool] x N, flatten, [FC, ReLU] x N, FC

Example: LeNet-5



Lecun et al, "Gradient-based learning applied to document recognition", 1998

Example: LeNet-5



Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv (C_{out}=20, K=5, P=2, S=1)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	
MaxPool(K=2, S=2)	20 x 14 x 14	
Conv (C_{out}=50, K=5, P=2, S=1)	50 x 14 x 14	50 x 20 x 5 x 5
ReLU	50 x 14 x 14	
MaxPool(K=2, S=2)	50 x 7 x 7	
Flatten	2450	
Linear (2450 -> 500)	500	2450 x 500
ReLU	500	
Linear (500 -> 10)	10	500 x 10

As we go through the network:

Spatial size **decreases**
(using pooling or strided conv)

Number of channels **increases**
(total "volume" is preserved!)

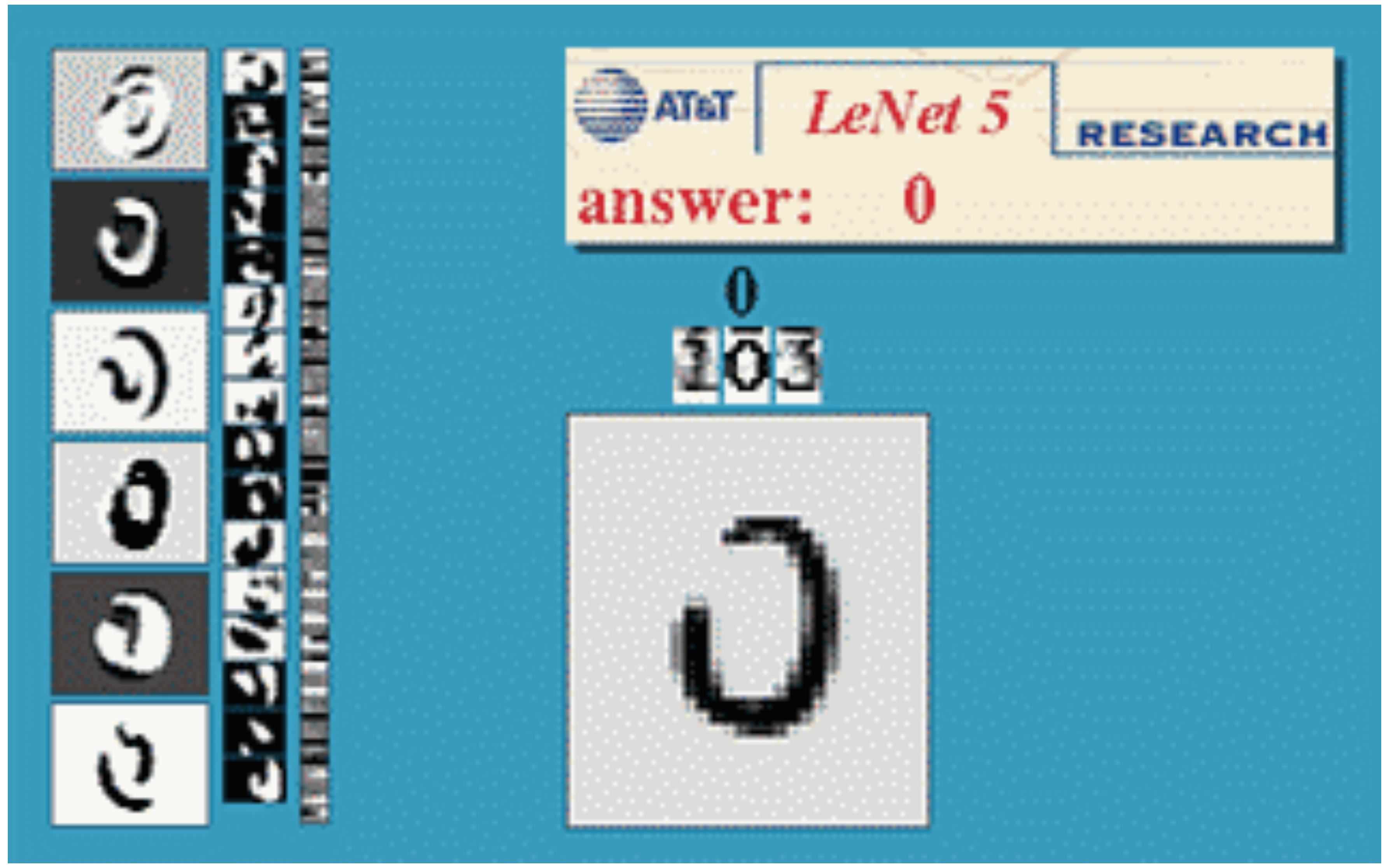
Lecun et al, "Gradient-based learning applied to document recognition", 1998

Optical Character Recognition (**OCR**)

Technology to convert **scanned documents to text**

(comes with any scanner now days)

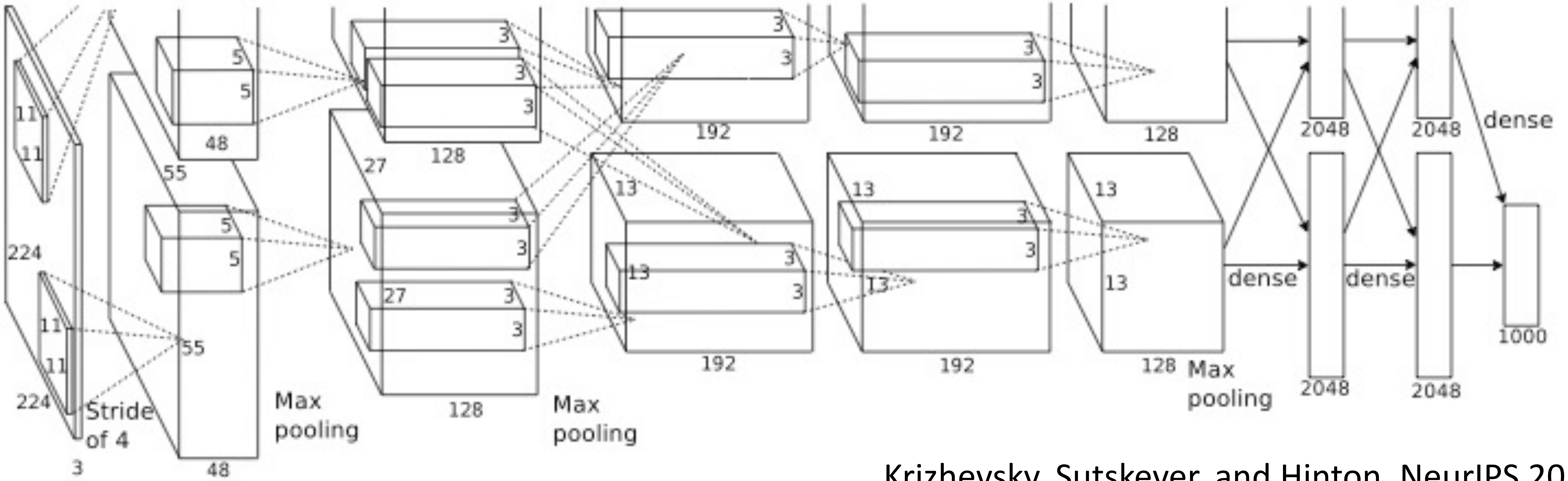
Yann LeCun



Digit recognition, AT&T labs
http://www.research.att.com/~yann/

License plate readers
http://en.wikipedia.org/wiki/Automatic_number_plate_recognition

AlexNet: Deep Learning Goes Mainstream



Krizhevsky, Sutskever, and Hinton, NeurIPS 2012

1959 Hubel & Wiesel

1963 Roberts

1970s David Marr

1979 Gen. Cylinders

1986 Canny

AI Winter

1997 Norm. Cuts

1999 SIFT

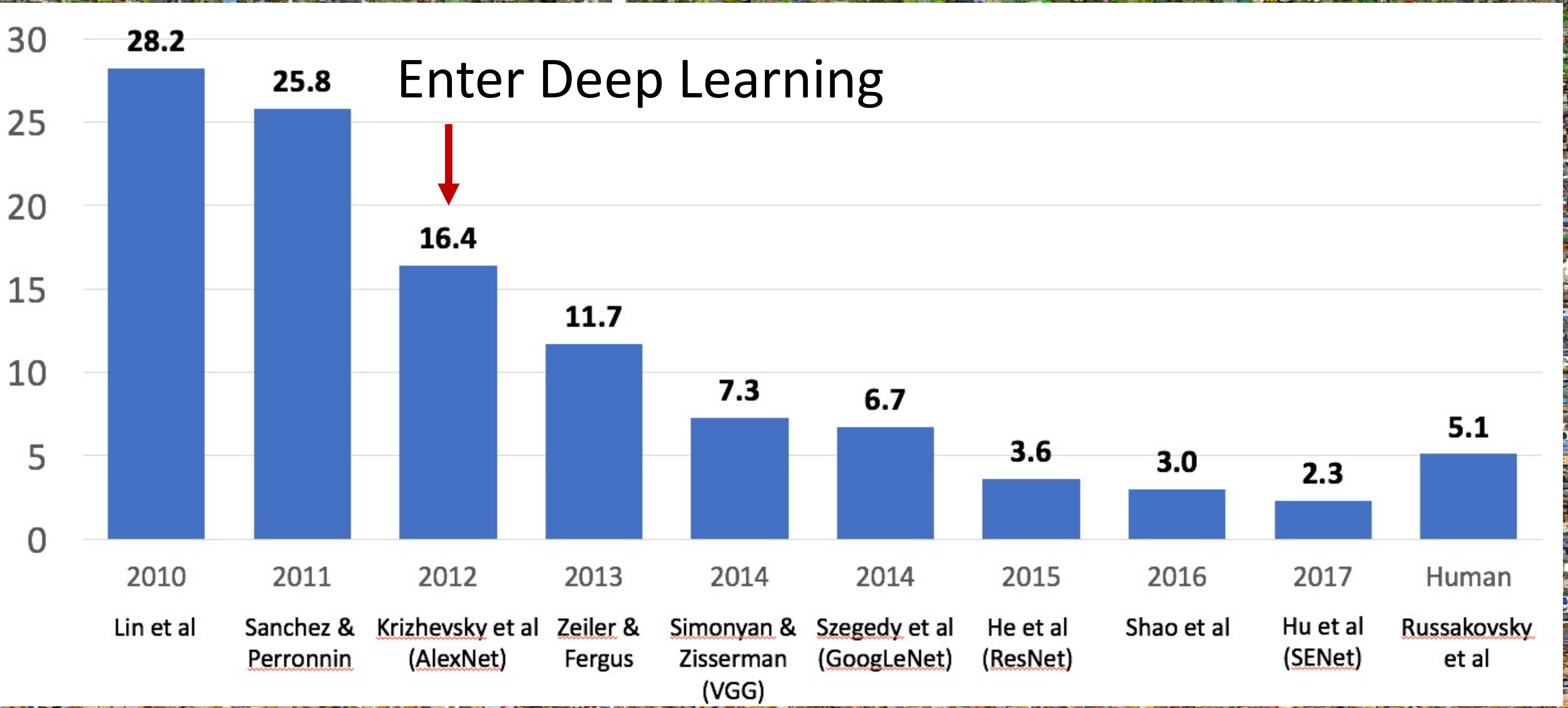
2001 V&J

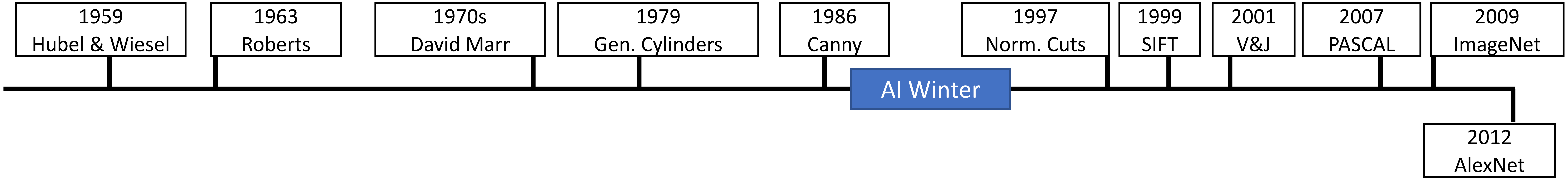
2007 PASCAL

2009 ImageNet

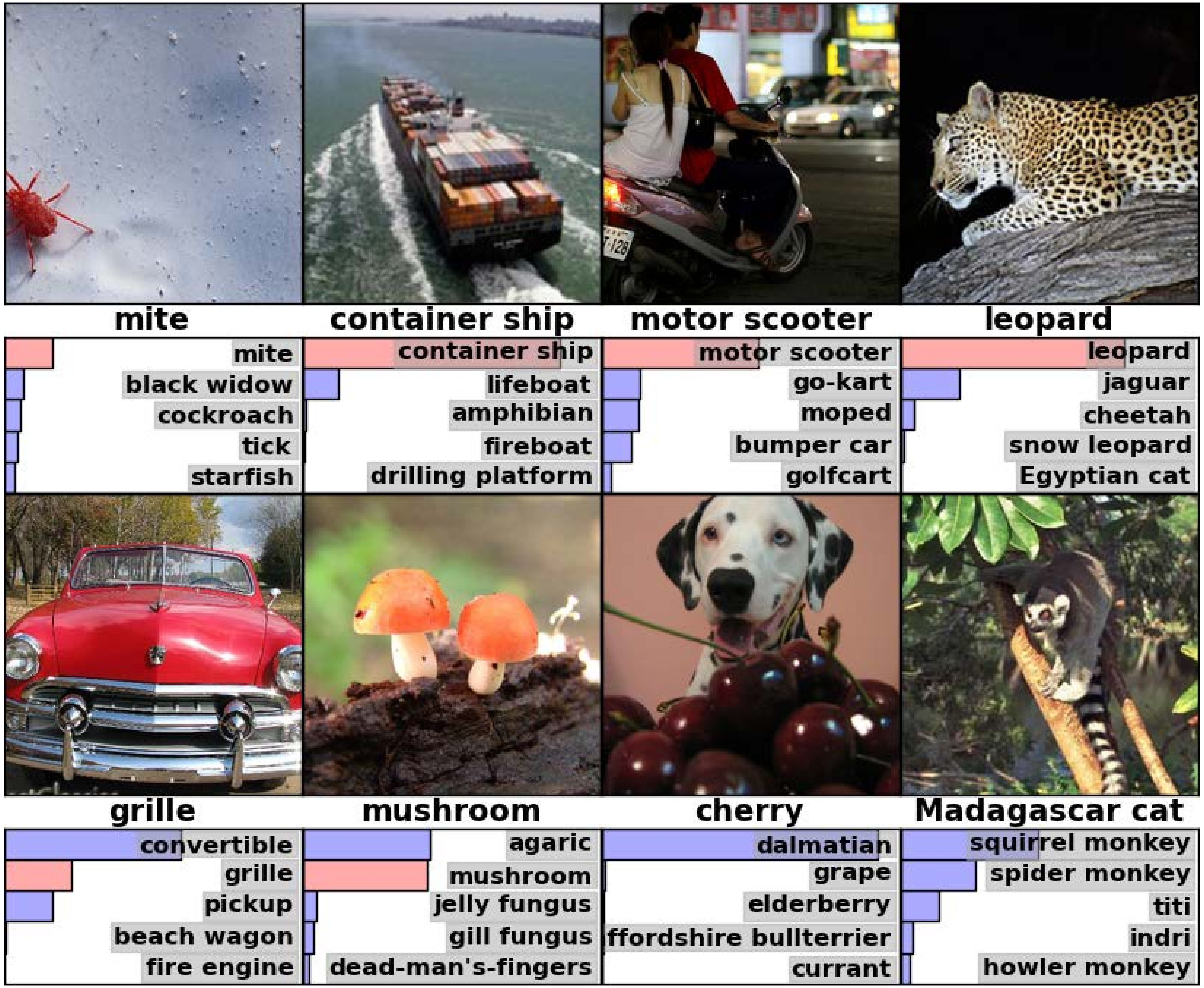
2012 AlexNet

IMAGENET Large Scale Visual Recognition Challenge

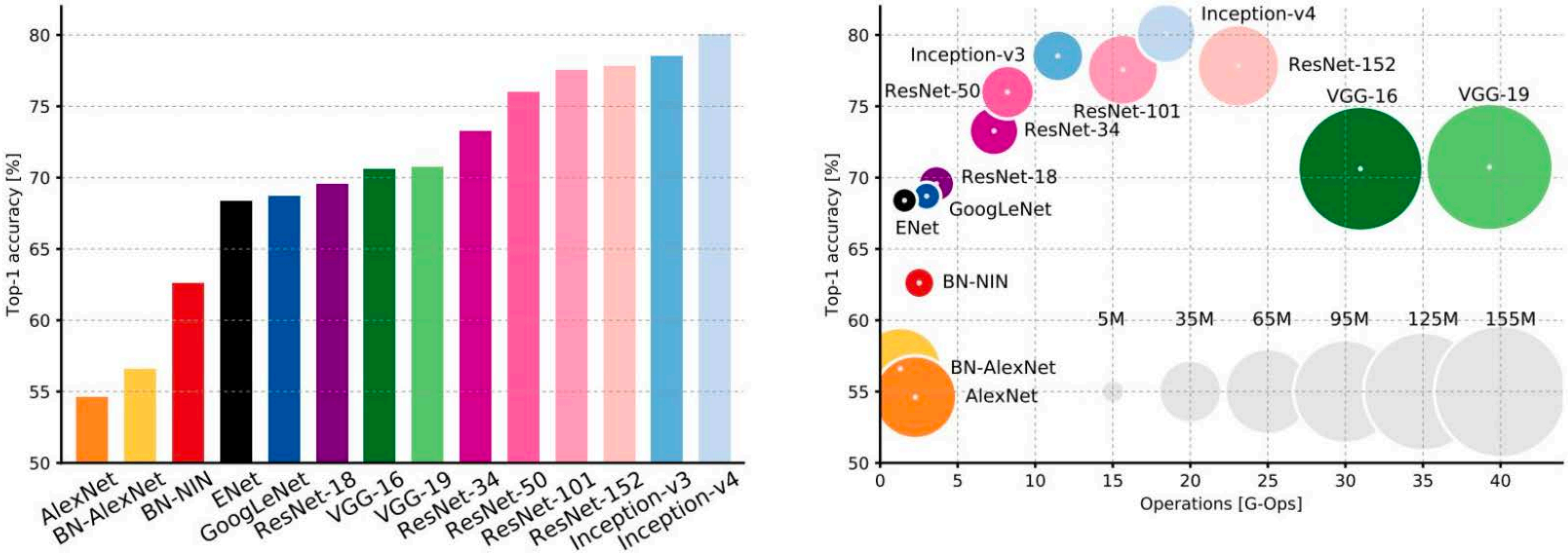




AlexNet on ImageNet



Comparing **Complexity**



An Analysis of Deep Neural Network Models for Practical Applications, 2017.

* adopted from Fei-Dei Li, Justin Johnson, Serena Yeung, **cs231n Stanford**

Summary

The parameters of a neural network are learned using **backpropagation**, which computes gradients via recursive application of the chain rule

A **convolutional neural network** assumes inputs are images, and constrains the network architecture to reduce the number of parameters

A **convolutional layer** applies a set of learnable filters

A **pooling layer** performs spatial downsampling

A **fully-connected** layer is the same as in a regular neural network

Convolutional neural networks can be seen as learning a hierarchy of filters