

# 95-865 Unstructured Data Analytics

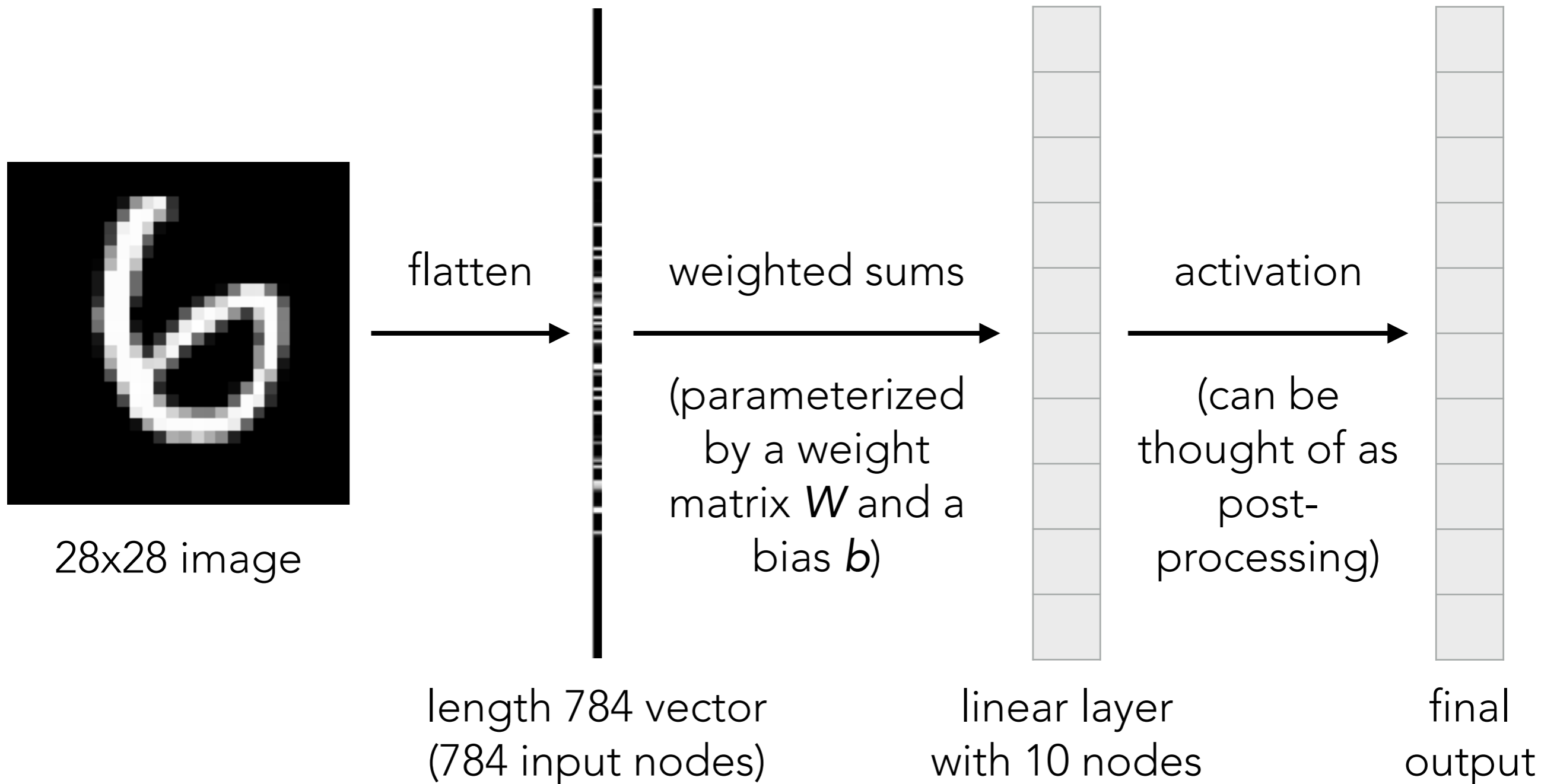
Lecture 12: Wrap up neural net basics;  
image analysis with convolutional neural  
nets (also called CNNs or convnets)

Nearly all slides by George H. Chen  
with a few by Phillip Isola

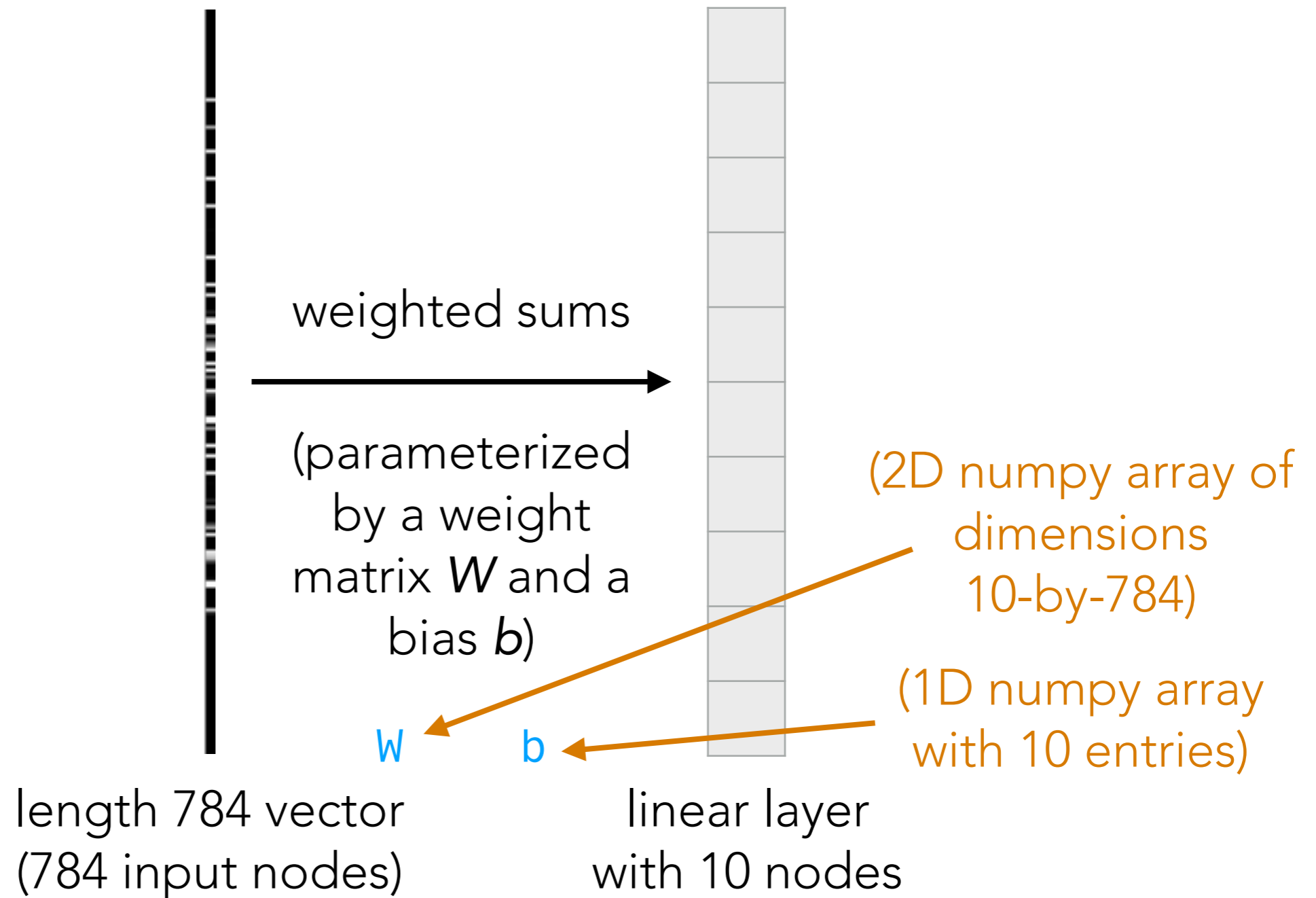
# Administrivia

- Reminder: HW2 due tonight 11:59pm
- Reminder: My office hours just for today have been shifted a little bit earlier and will be from 6:30pm-7:30pm, still over the same Zoom link

# (Flashback) Handwritten Digit Recognition



# (Flashback) Handwritten Digit Recognition



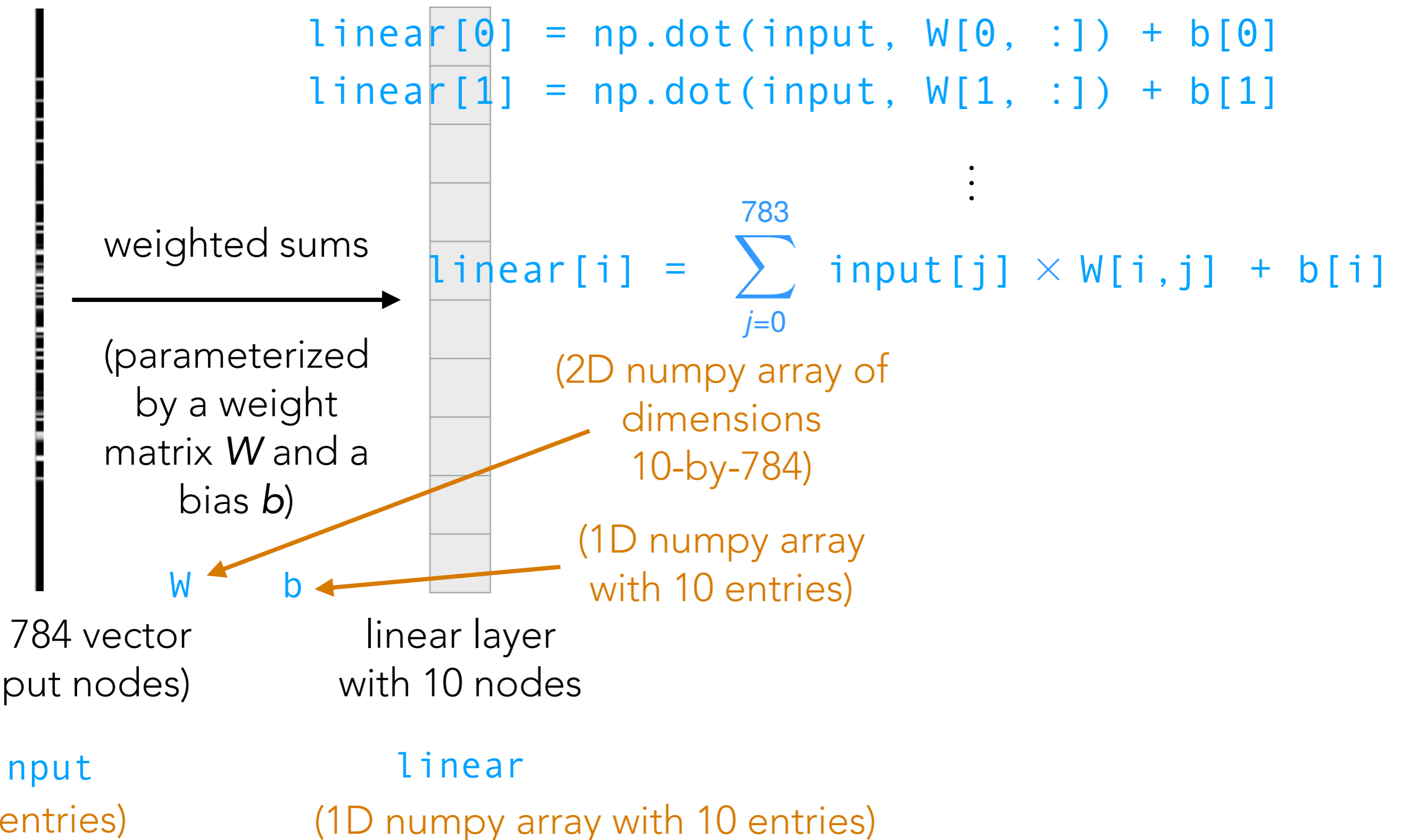
input

(1D numpy array with 784 entries)

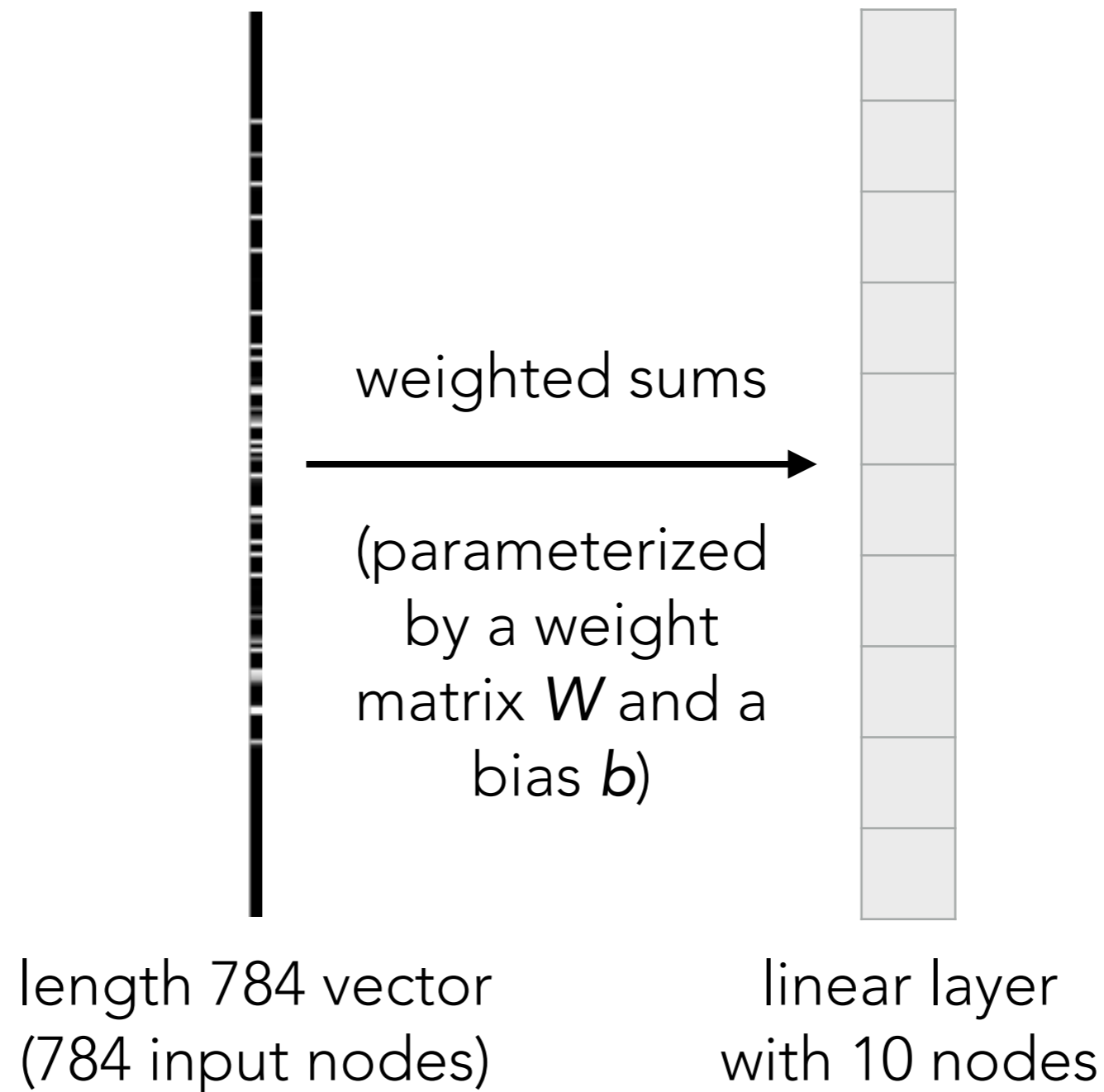
linear

(1D numpy array with 10 entries)

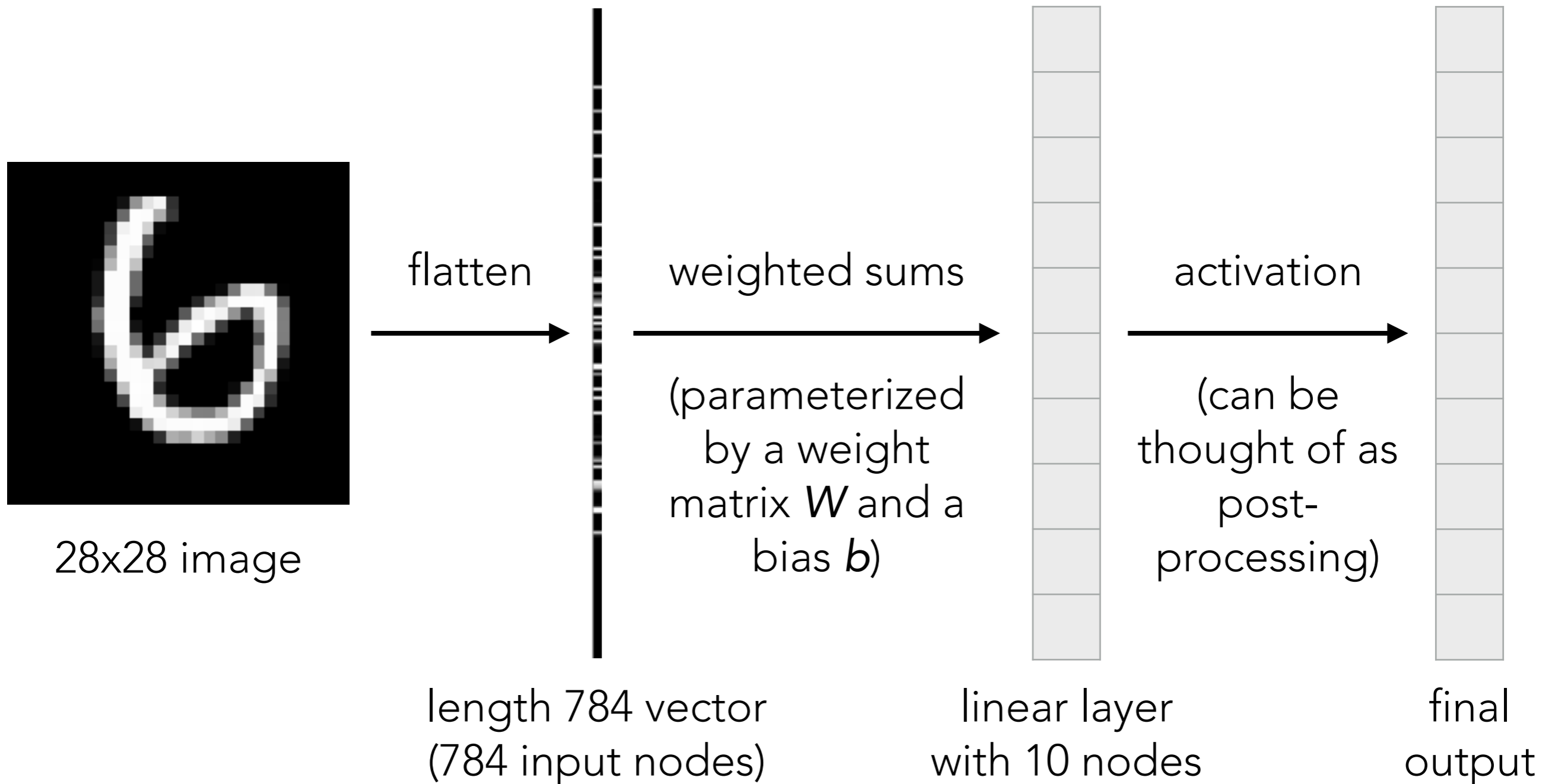
# (Flashback) Handwritten Digit Recognition



# (Flashback) Handwritten Digit Recognition



# (Flashback) Handwritten Digit Recognition

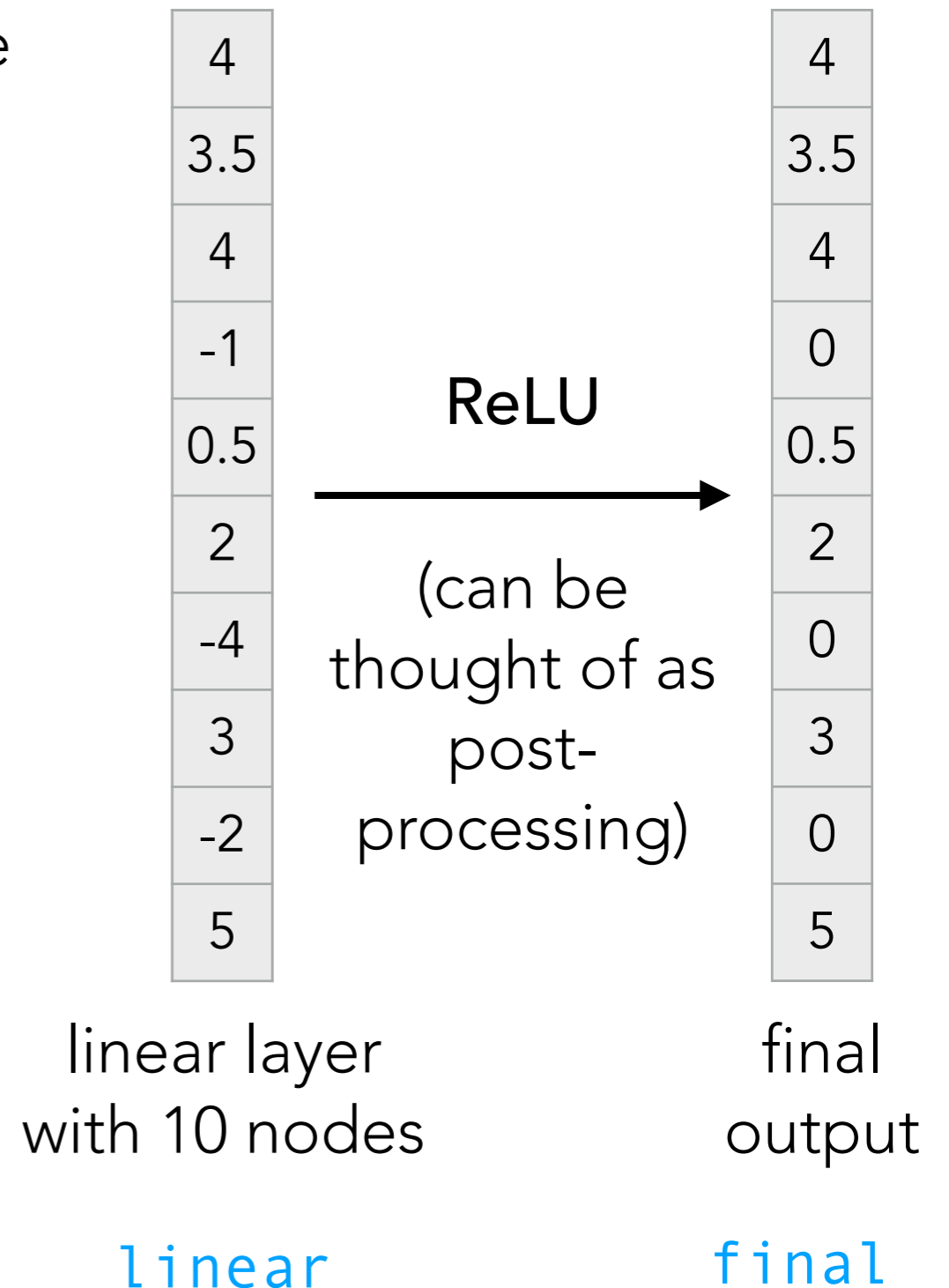


# (Flashback) Handwritten Digit Recognition

Many different activation functions possible

Example: **Rectified linear unit (ReLU)**  
zeros out entries that are negative

```
final = np.maximum(0, linear)
```



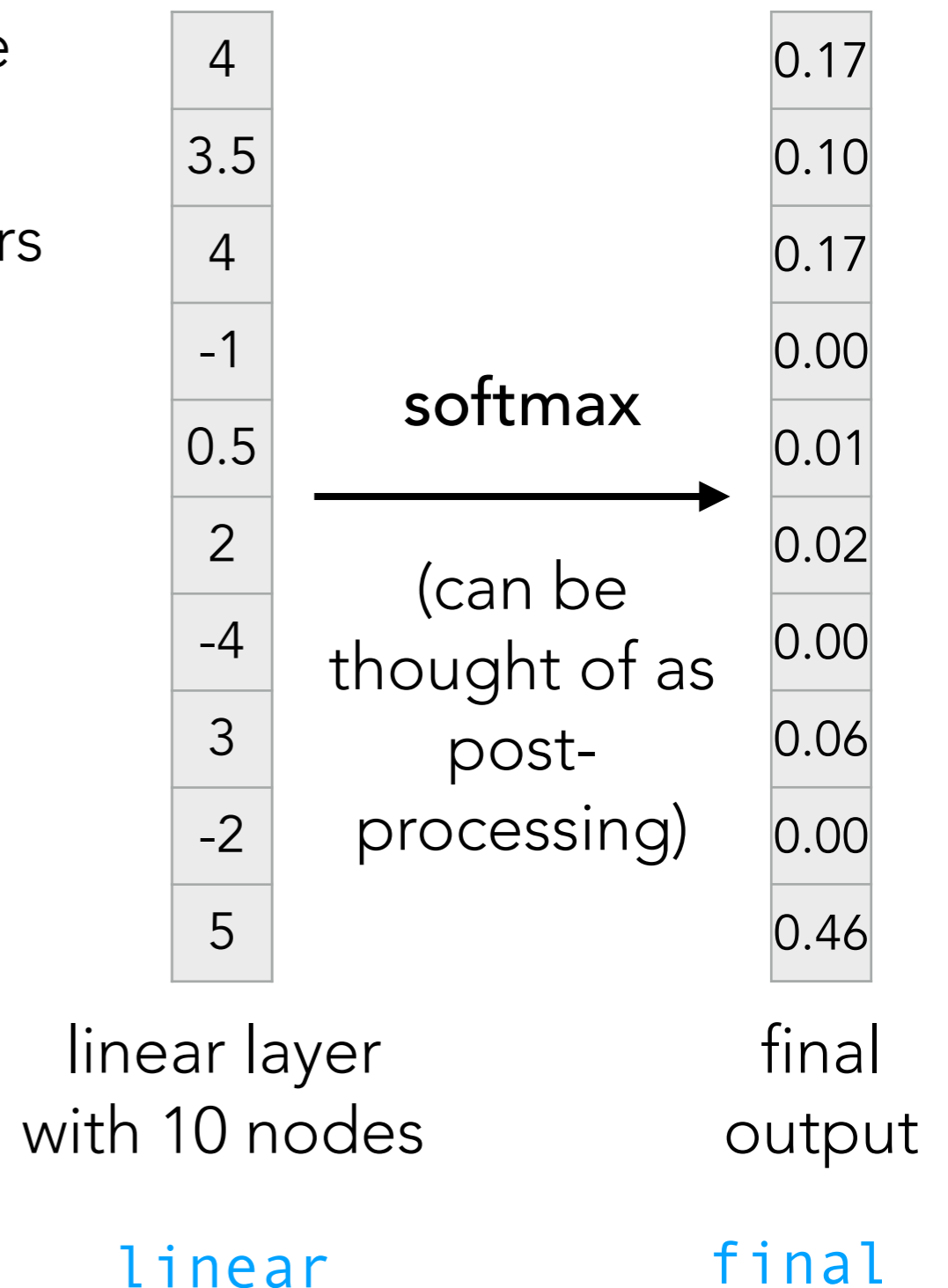


# (Flashback) Handwritten Digit Recognition

Many different activation functions possible

Example: **softmax** converts a table of numbers into a probability distribution

```
exp = np.exp(linear)
final = exp / exp.sum()
```



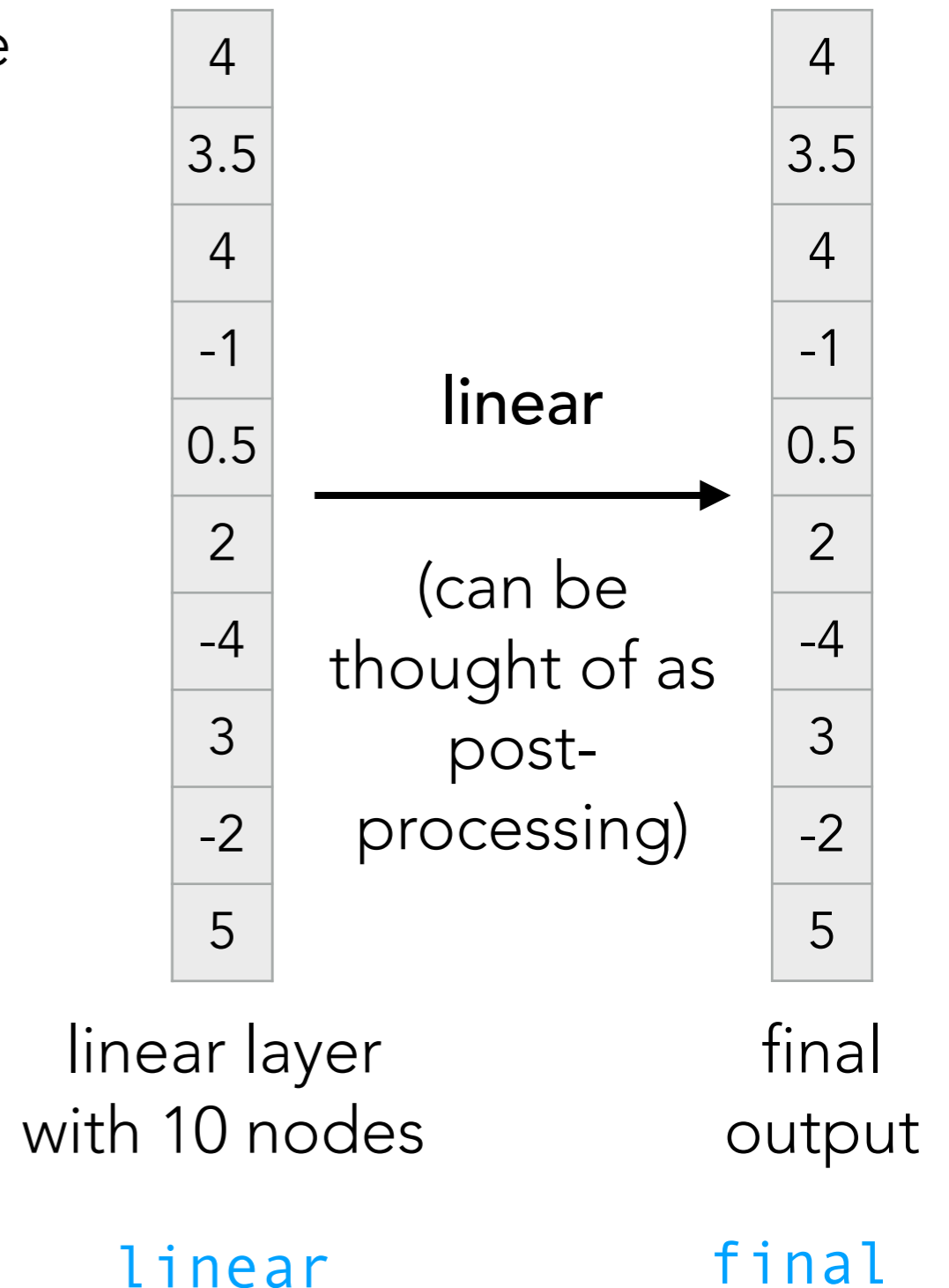
# (Flashback) Handwritten Digit Recognition

Many different activation functions possible

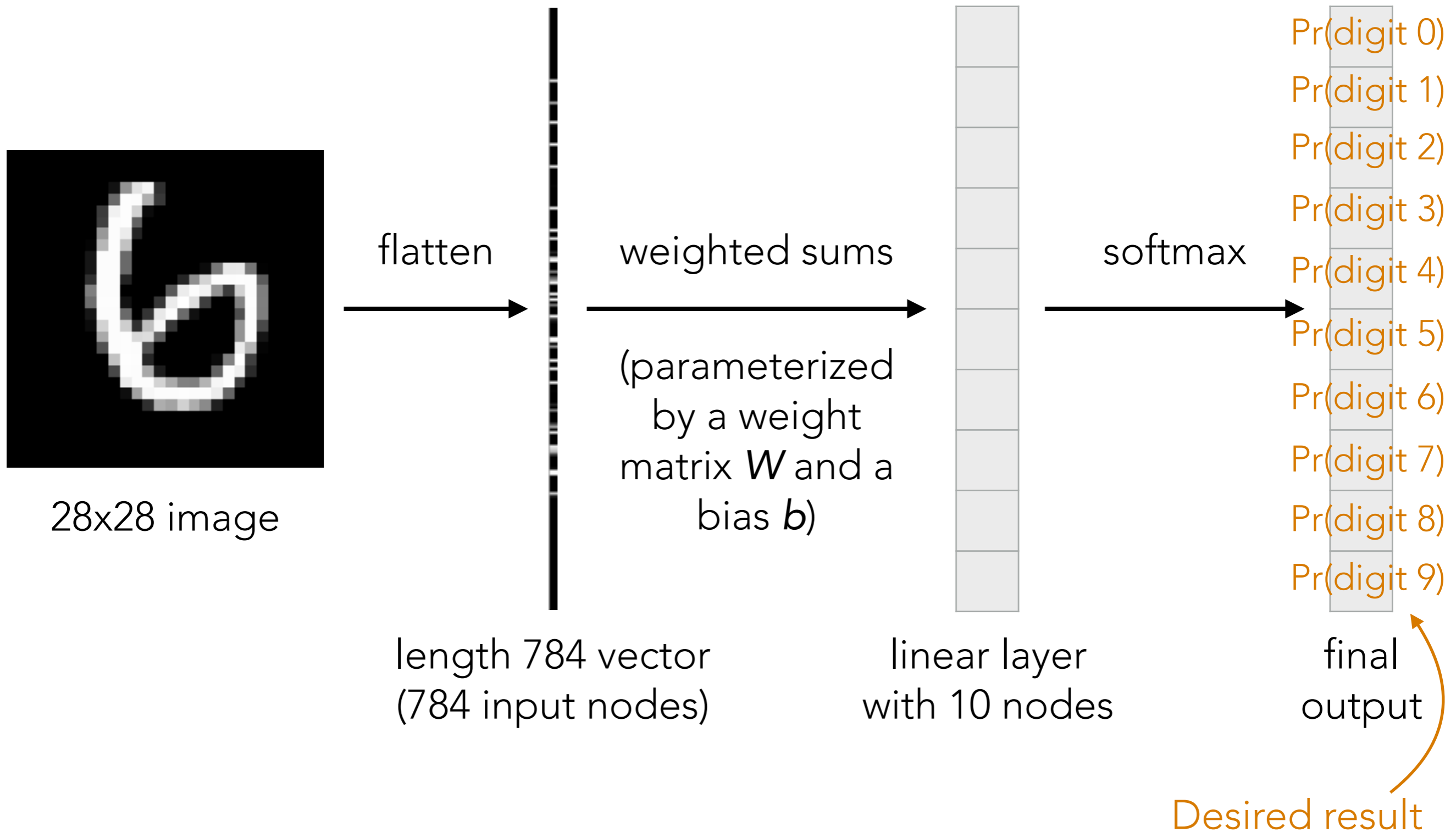
Example: **linear** activation does nothing

This is equivalent to there being  
no activation function

`final = linear`

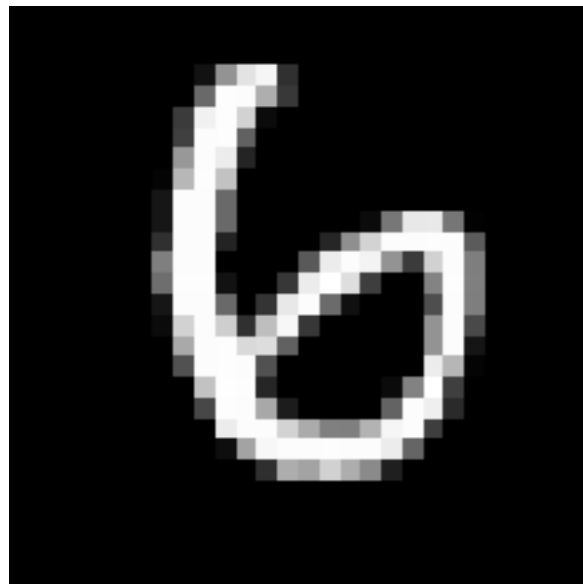


# Handwritten Digit Recognition

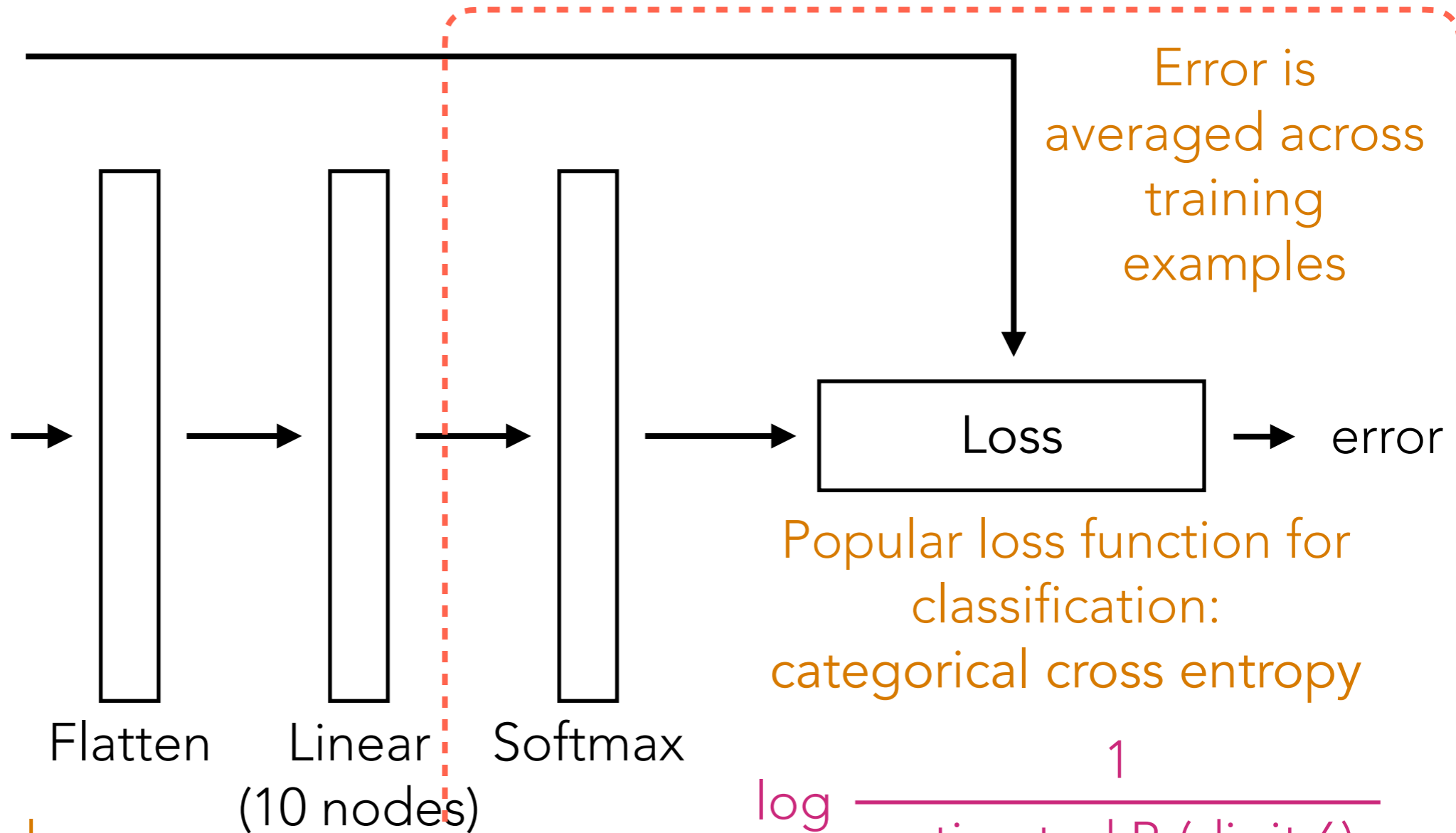


# Handwritten Digit Recognition

Training label: 6



Input



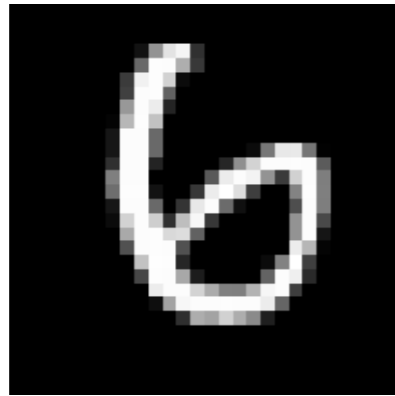
Learning this neural net means finding  $W$  and  $b$  that minimize categorical cross entropy loss

Also called *fully-connected* or *dense* layer

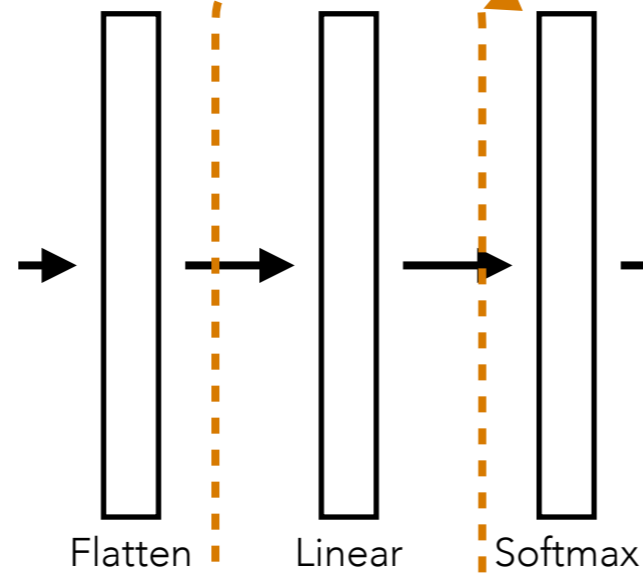
⚠ In PyTorch, softmax is included as part of the cross entropy loss

# Handwritten Digit Recognition

Training label: 6



Input



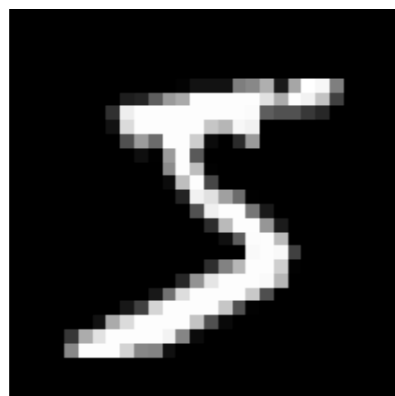
Popular loss function for classification: categorical cross entropy

$$\log \frac{1}{\text{estimated Pr}(\text{digit } 6)}$$

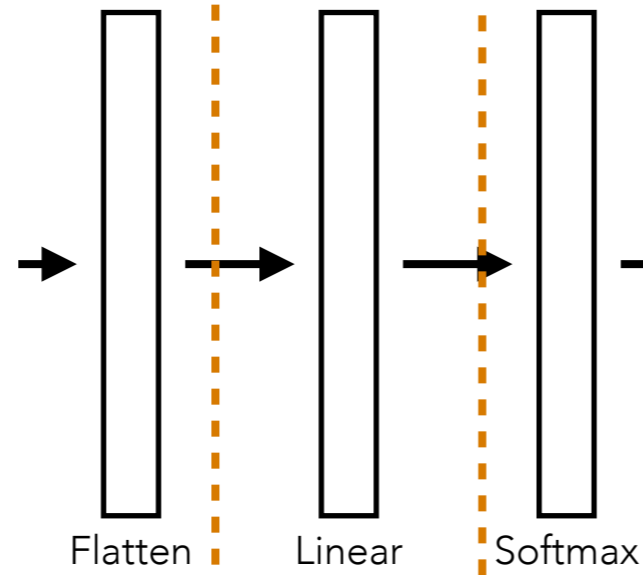
error

Important: across different training data, we are using the same linear layer (same  $W$  and  $b$  parameters)

Training label: 5



Input



Popular loss function for classification: categorical cross entropy

$$\log \frac{1}{\text{estimated Pr}(\text{digit } 5)}$$

error

average loss/error

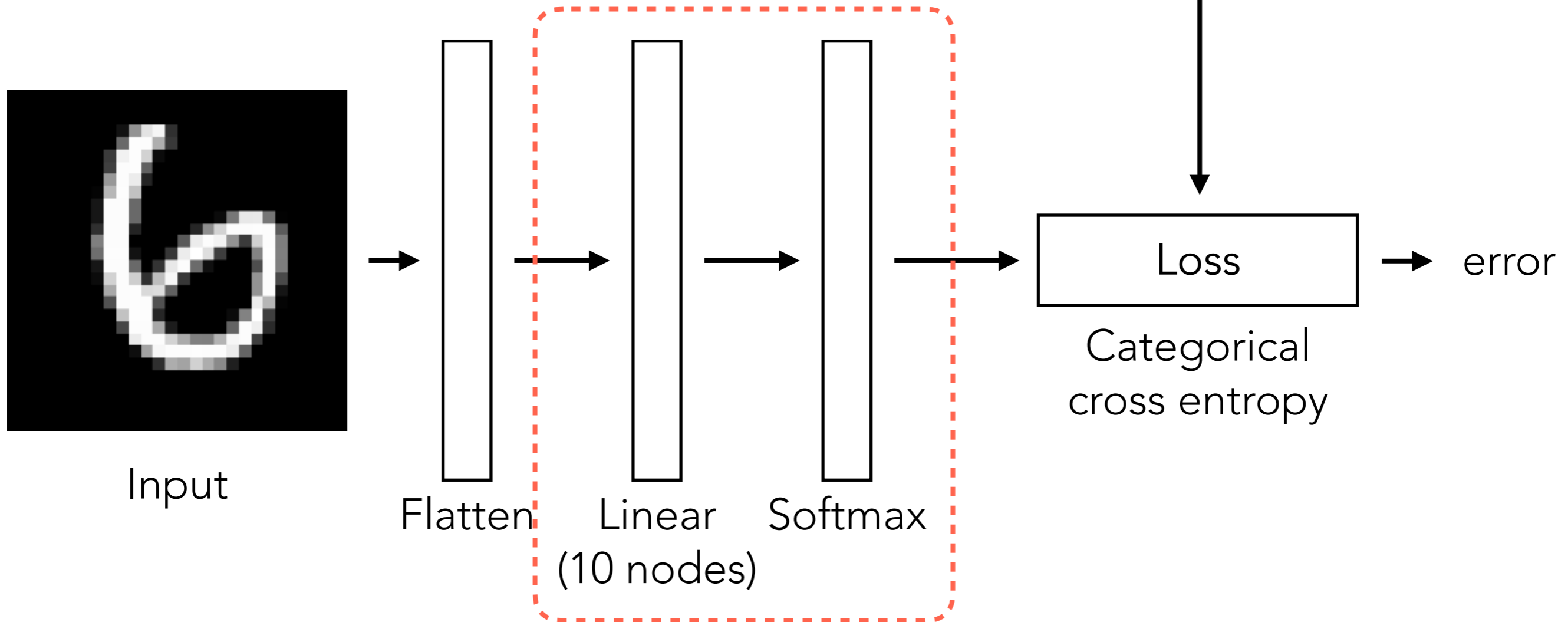
Example: 2 training points

Learning this neural net means finding  $W$  and  $b$  that minimize categorical cross entropy loss

(averaged across training examples)

# Handwritten Digit Recognition

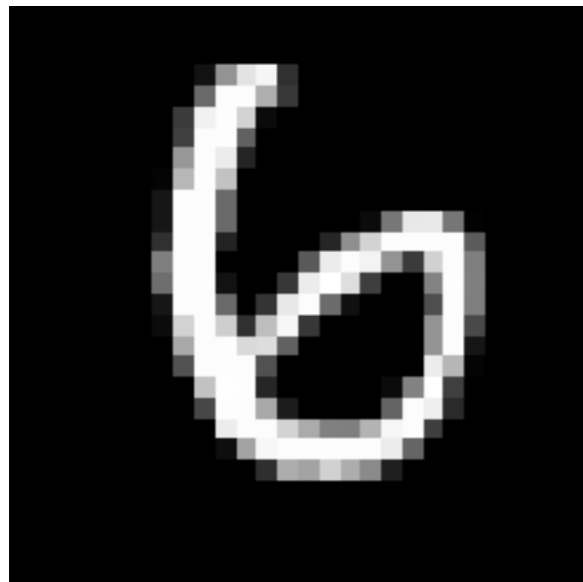
Training label: 6



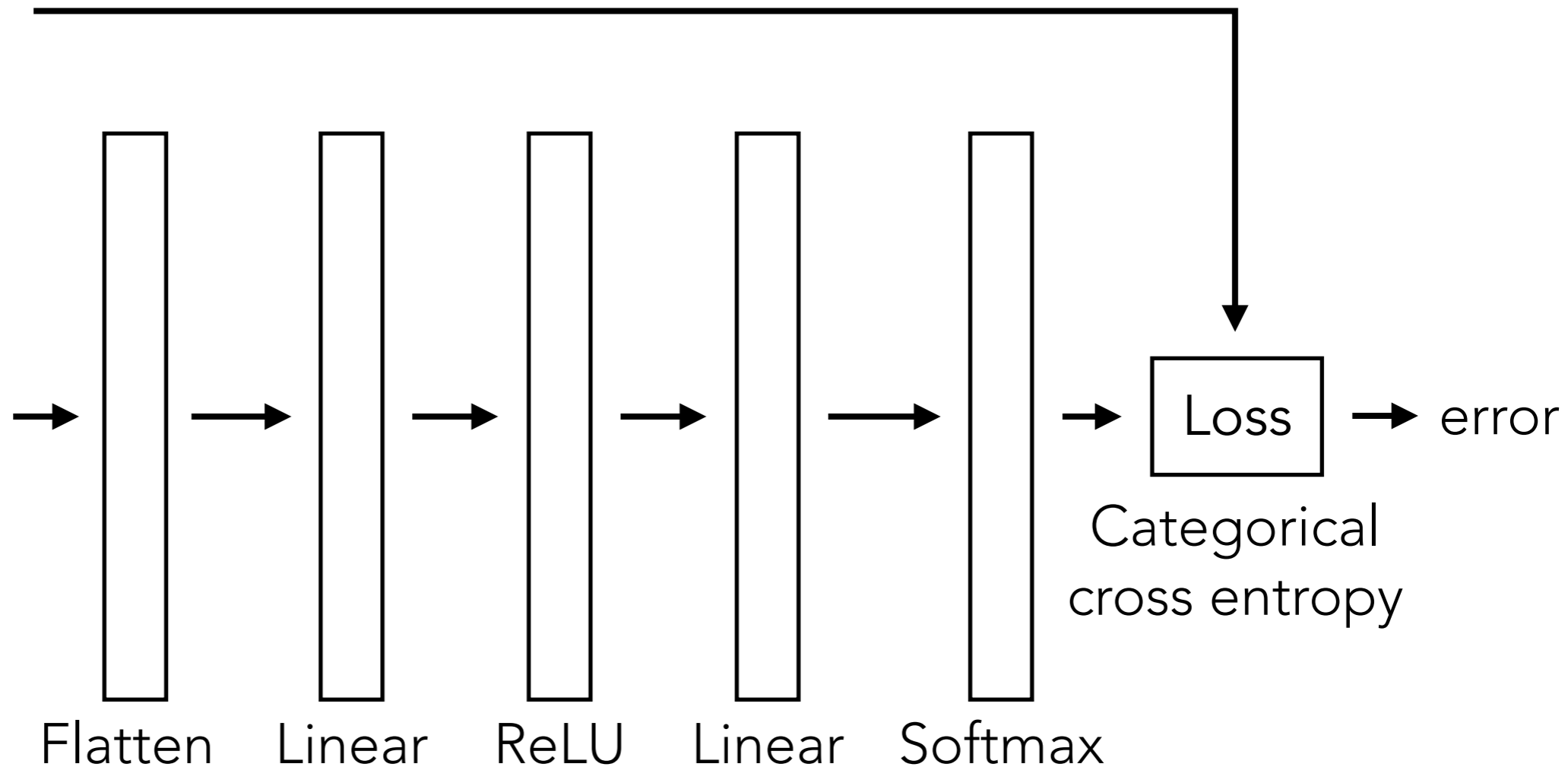
This neural net has a name: **multinomial logistic regression** (when there are only 2 classes, it's called **logistic regression**)

# Handwritten Digit Recognition

Training label: 6



Input



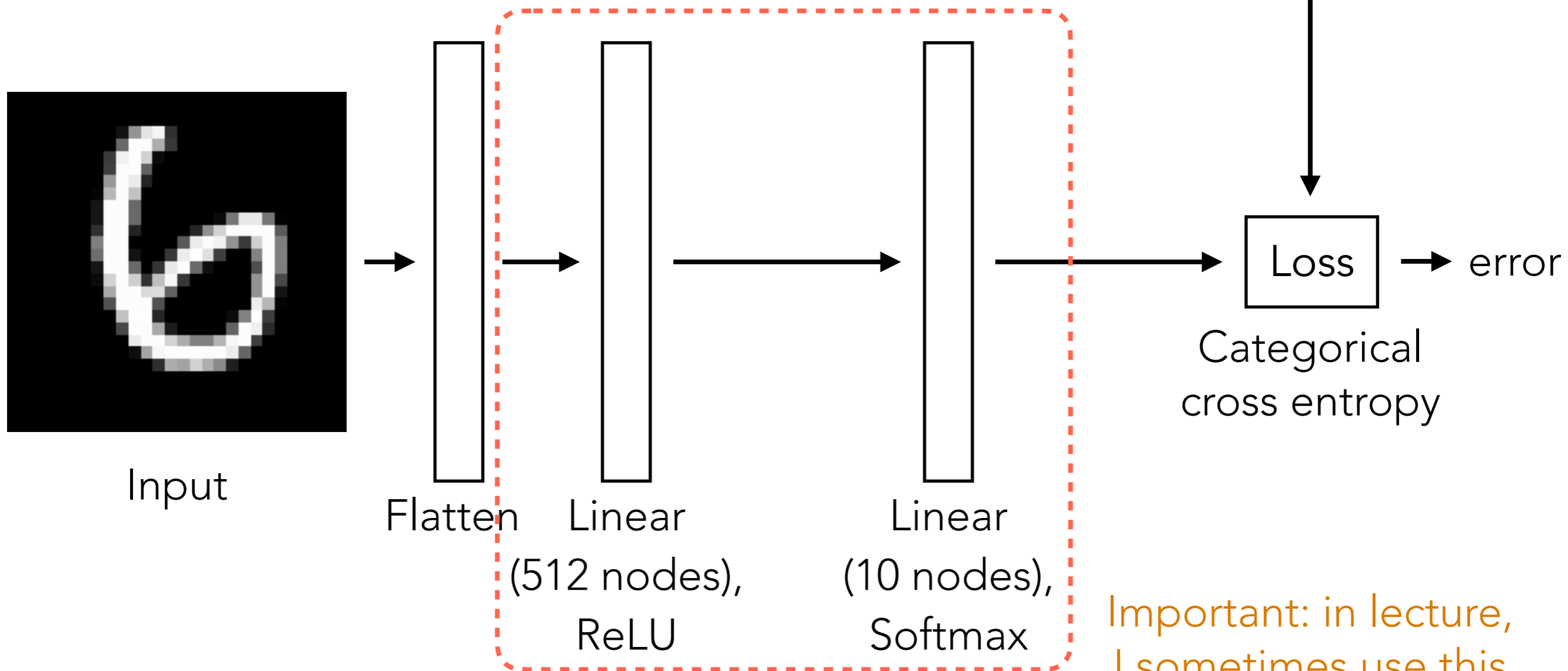
Different linear layers; each has its own weight matrix and bias vector

Basic building block of neural nets: *linear layer with nonlinear activation*

Learning this neural net  $\Rightarrow$  learn parameters of both linear layers

# Handwritten Digit Recognition

Training label: 6





This neural net is called a multilayer perceptron  
(# nodes need not be 512 & 10;  
activations need not be ReLU and softmax)

Important: in lecture,  
I sometimes use this  
shorthand notation  
(specifying activation to  
go with each linear layer)



# PyTorch

- Designed to be like NumPy
  - A lot of (but not all) function names are the same as numpy (e.g., instead of calling `np.sum`, you would call `torch.sum`, etc)
  -  PyTorch does not use NumPy arrays and instead uses tensors (so instead of `np.array`, you use `torch.tensor`)
- What's the big difference then? Why not just use NumPy?
  - **PyTorch tensors keep track of what device they reside on**
    -  For example, trying to add a tensor stored on the CPU and a tensor stored on a GPU will result in an error!
  - **PyTorch tensors can automatically store "gradient" information** (important for learning model parameters; details in later lecture)

PyTorch code is often harder to debug than NumPy code

There's a PyTorch tutorial posted in supplemental materials

# Handwritten Digit Recognition

Demo

# Architecting Neural Nets

- Basic building block that is often repeated:  
*linear* layer followed by *nonlinear* activation
  - Without nonlinear activation, two consecutive linear layers is mathematically equivalent to having a single linear layer!
- How to select # of nodes in a layer, or # of layers?
  - These are hyperparameters! *Infinite* possibilities!
  - Choose between different hyperparameter settings by using the strategy from last lecture (choose based on validation accuracy)
    - Very expensive in practice!  
(Active area of research: neural architecture search)
  - Much more common in practice: modify existing architectures that are known to work well  
(e.g., ResNet or CLIP for image classification/object recognition)

# PyTorch Has Lots of Examples

 Search Docs

PyTorch Examples

Docs > PyTorch Examples



## PYTORCH EXAMPLES

This pages lists various PyTorch examples that you can use to learn and experiment with PyTorch.

### Image Classification using Vision Transformer

This example shows how to train a **Vision Transformer** from scratch on the **CIFAR10** database.

[GO TO EXAMPLE](#) 

### Image Classification Using ConvNets

This example demonstrates how to run image classification with **Convolutional Neural Networks ConvNets** on the **MNIST** database.

[GO TO EXAMPLE](#) 

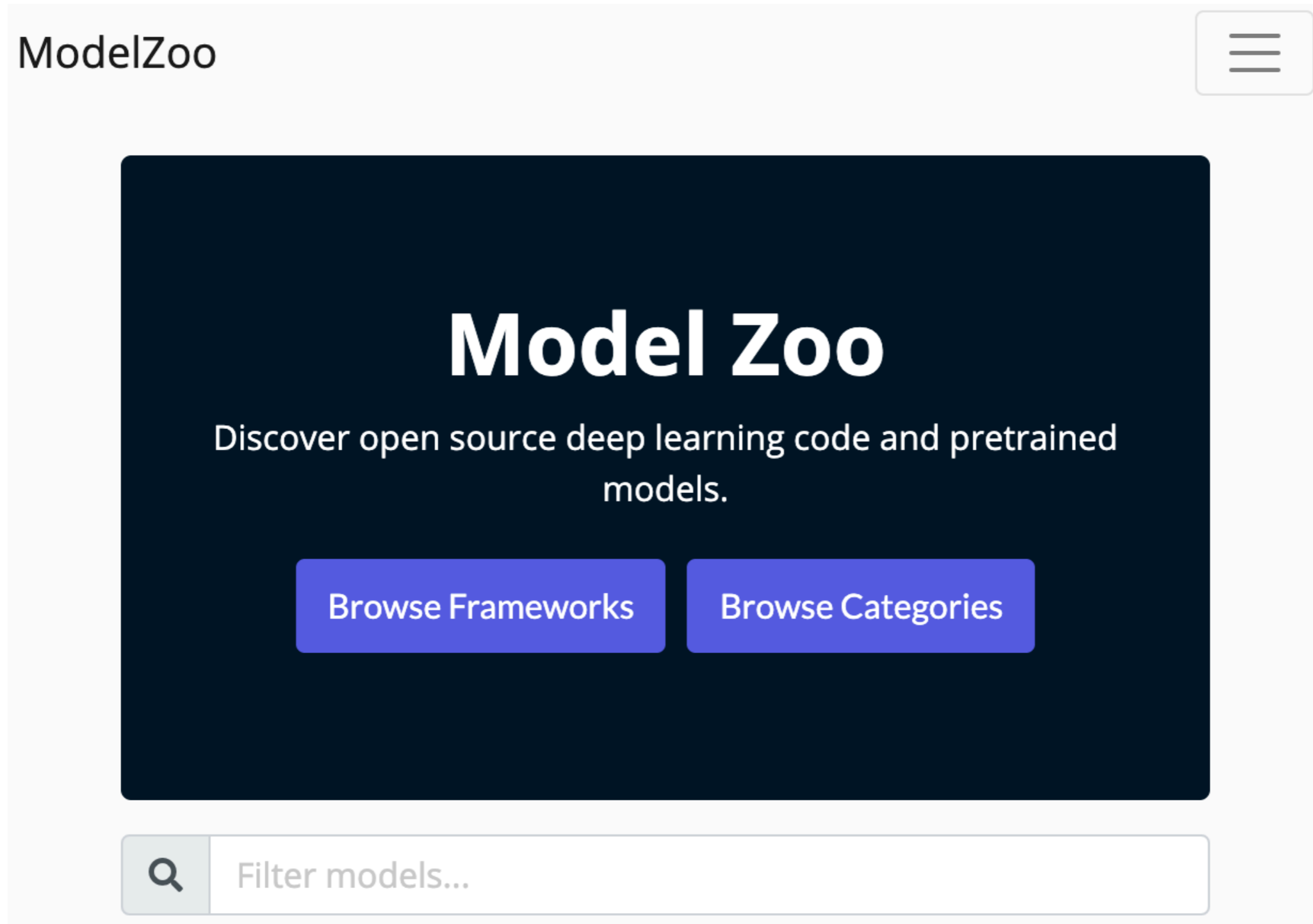
### Measuring Similarity using Siamese Network

This example demonstrates how to measure similarity between two images using **Siamese**

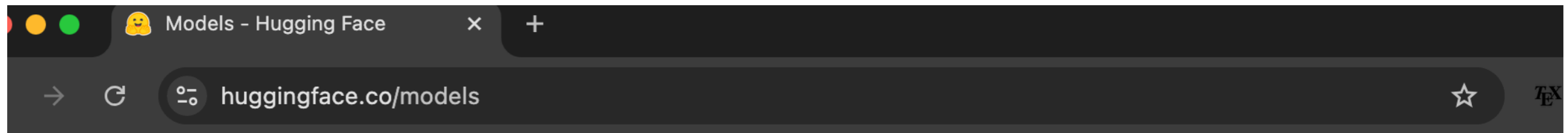
### Word-level Language Modeling using RNN and Transformer

This example demonstrates how to train a multi-

# Find a Massive Collection of Models at the Model Zoo



# More Recently: Lots of Models are on Hugging Face 🤗



Search models, datasets, users...

Models Datasets Spaces

Hugging Face is way more fun with friends and colleagues! 🤗 [Join an organization](#)

Tasks Libraries Datasets Languages Licenses Other

Filter Tasks by name

Multimodal

Audio-Text-to-Text Image-Text-to-Text

Visual Question Answering

Document Question Answering

Video-Text-to-Text Any-to-Any

Computer Vision

Depth Estimation Image Classification

Object Detection Image Segmentation

Text-to-Image Image-to-Text

Models 1,146,122 Filter by name

Qwen/Qwen2.5-Coder-32B-Instruct  
Text Generation • Updated 5 days ago • 58.4k • 922

mistralai/Pixtral-Large-Instruct-2411  
Updated 3 days ago • 382 • 282

NexaAIDev/omnivision-968M  
Updated about 5 hours ago • 7.33k • 365

black-forest-labs/FLUX.1-dev  
Text-to-Image • Updated Aug 16 • 1.29M • 6.59k

briaai/RMBG-2.0

Learning a neural net amounts to  
"curve fitting"

We're just estimating a function

# Neural Net as Function Approximation

Given `input`, learn a computer program that computes `output`

this is a function

Multinomial logistic regression:

```
def f(input):
```

```
    output = softmax(np.dot(input, W.T) + b)
```

```
    return output
```

the only things that we are learning  
(we fix their dimensions in advance)

We are fixing what the function `f` looks like in code and are only adjusting `W` and `b`!!!



# Neural Net as Function Approximation

Given `input`, learn a computer program that computes `output`

Multinomial logistic regression:

```
output = softmax(np.dot(input, W.T) + b)
```

Multilayer perceptron:

```
intermediate = relu(np.dot(input, W1.T) + b1)
```

```
output = softmax(np.dot(intermediate, W2.T) + b2)
```

Learning a neural net: learning a simple computer program that maps inputs (raw feature vectors) to outputs (predictions)

# Complexity of a Neural Net?

Increasing number of layers (depth) makes neural net more “complex”

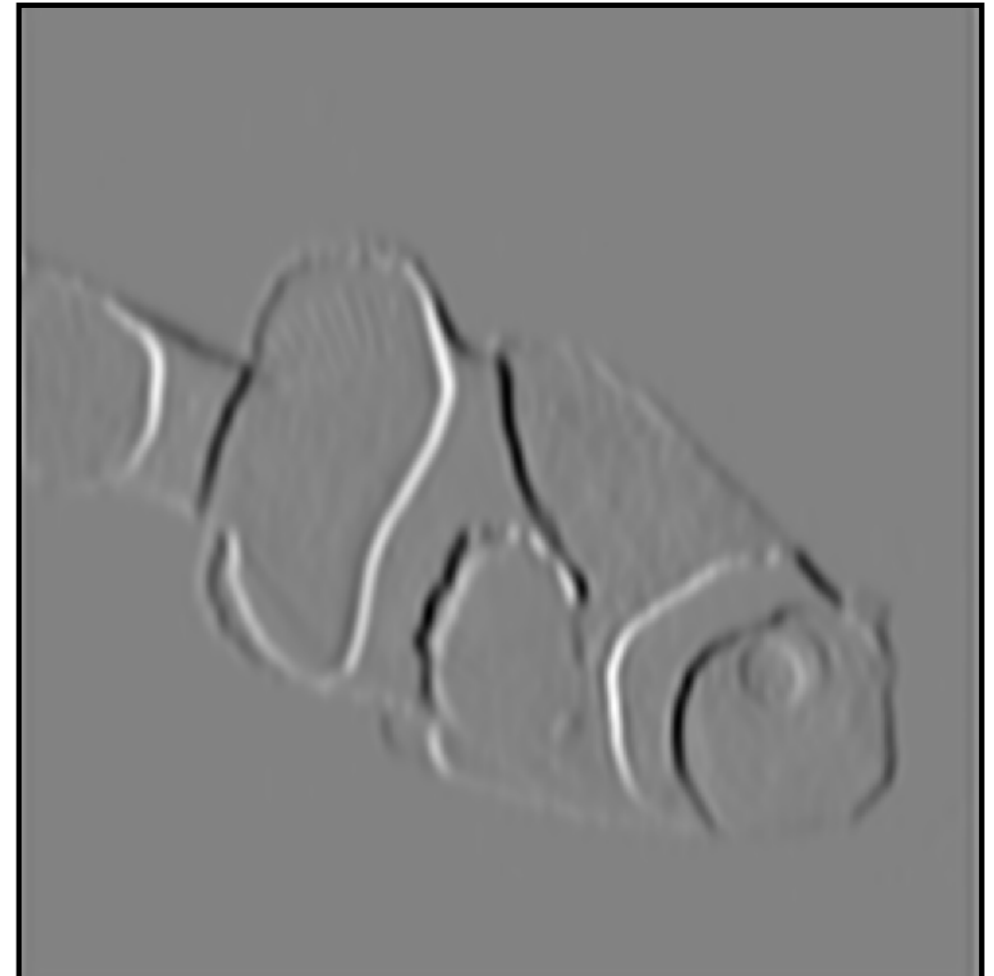
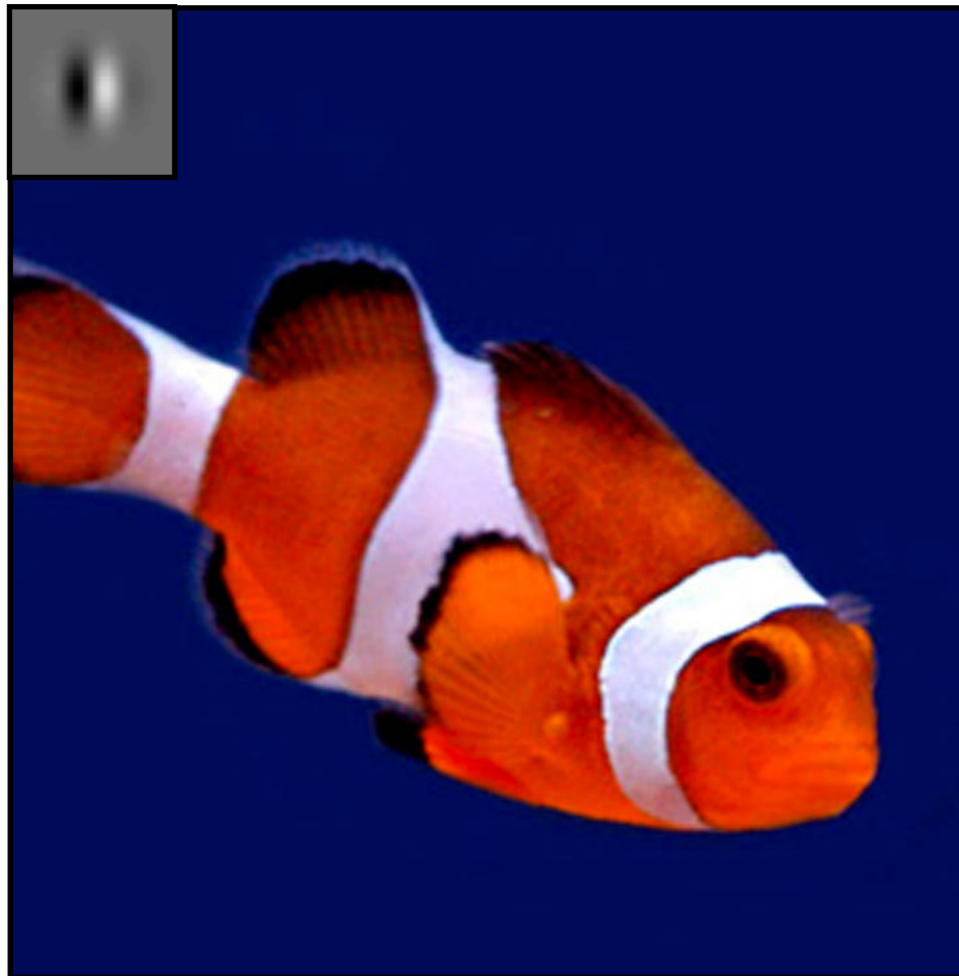
⇒ Learn computer program that has more lines of code

Earlier: MLP had more parameters than logistic regression

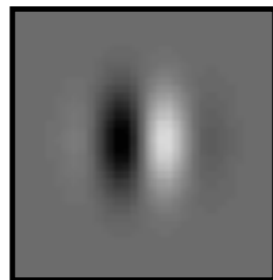
Upcoming: we'll see an example where a deeper network has *fewer* parameters than a shallower one

Accounting for image structure:  
convolutional neural nets  
(CNNs or convnets)

# Convolution



filter



# Convolution

0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	1	1	1	1	1	0
0	1	1	1	0	0	0
0	1	1	1	1	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

Input image

0	0	0
0	1	0
0	0	0

Filter  
(also called "kernel")

# Convolution

0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	1	1	1	1	1	0
0	1	1	1	0	0	0
0	1	1	1	1	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

Input image

0	0	0
0	1	0
0	0	0

Filter  
(also called "kernel")

# Convolution

Take dot product!

0	0	0	0	0	0	0
0	0	1	1	1	1	0
0	1	0	1	0	1	0
0	1	1	1	1	0	0
0	1	1	1	1	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

Input image

0				

Output image

# Convolution

Take dot product!

0	0	0	0	0	0	0
0	0	1	1	0	1	0
0	1	1	1	0	1	0
0	1	1	1	0	0	0
0	1	1	1	1	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

Input image

0	1			

Output image



# Convolution

Take dot product!

0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	1	1	0	1	0	0
0	1	1	1	0	0	0
0	1	1	1	1	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

Input image

0	1	1		

Output image

# Convolution

Take dot product!

0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	1	1	1	0	0	0
0	1	1	1	1	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

Input image

0	1	1	1	

Output image

# Convolution

Take dot product!

0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	1	1	1	1	0	0
0	1	1	1	0	0	0
0	1	1	1	1	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

Input image

0	1	1	1	0

Output image

# Convolution

Take dot product!

0	0	0	0	0	0	0		
0	0	0	1	0	1	1	0	0
0	0	1	1	1	0	1	1	0
0	0	1	0	1	0	1	0	0
0	1	1	1	1	1	1	1	0
0	0	1	1	1	0	0	0	
0	0	0	0	0	0	0	0	

Input image

0	1	1	1	0
1				

Output image

# Convolution

Take dot product!

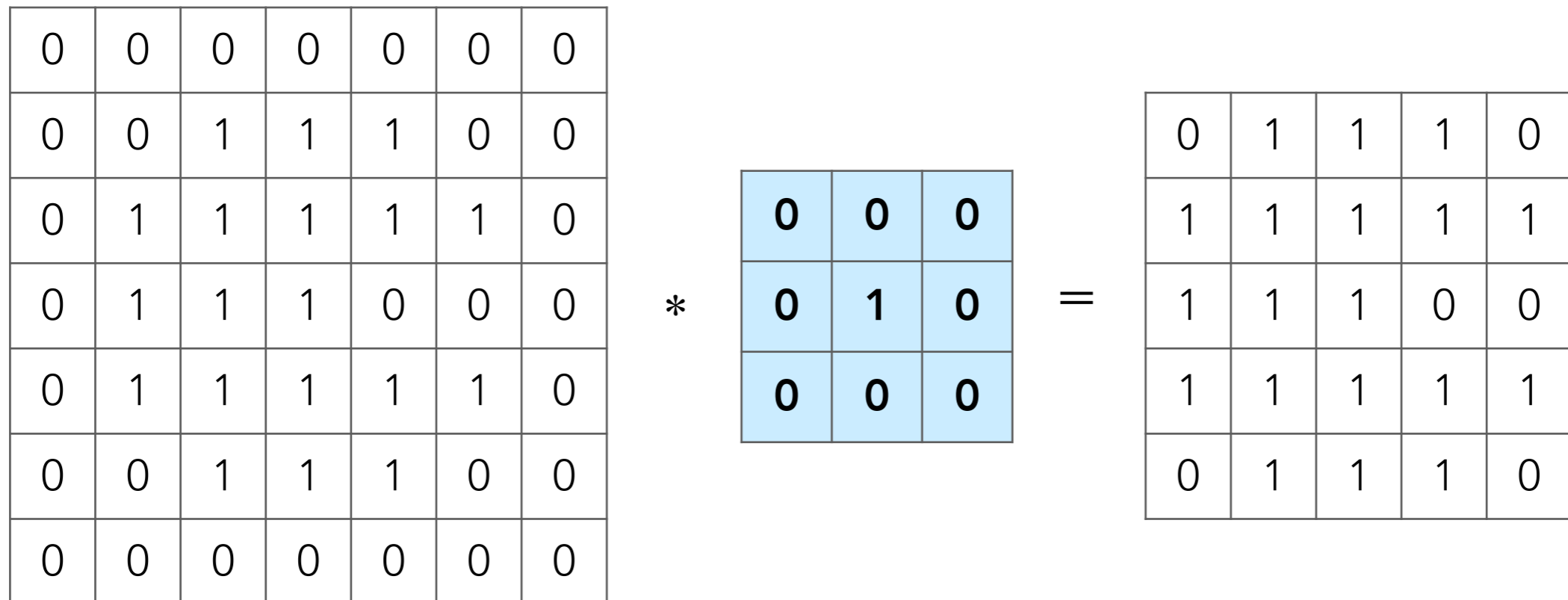
0	0	0	0	0	0	0
0	0	1	1	0	1	0
0	1	1	1	0	1	0
0	1	0	1	0	0	0
0	1	1	1	1	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

Input image

0	1	1	1	0
1	1			

Output image

# Convolution



Input image

Output image

Note: output image is smaller than input image

If you want output size to be same as input, pad 0's to input

# Convolution

<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>
<b>0</b>	0	0	0	0	0	0	0	<b>0</b>
<b>0</b>	0	0	1	1	1	0	0	<b>0</b>
<b>0</b>	0	1	1	1	1	1	0	<b>0</b>
<b>0</b>	0	1	1	1	0	0	0	<b>0</b>
<b>0</b>	0	1	1	1	1	1	0	<b>0</b>
<b>0</b>	0	0	1	1	1	0	0	<b>0</b>
<b>0</b>	0	0	0	0	0	0	0	<b>0</b>
<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>

Input image

\*

0	0	0
0	1	0
0	0	0

=

0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	1	1	1	1	1	0
0	1	1	1	0	0	0
0	1	1	1	1	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

Output image

Note: output image is smaller than input image

If you want output size to be same as input, pad 0's to input

# Convolution

0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	1	1	1	1	1	0
0	1	1	1	0	0	0
0	1	1	1	1	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

Input image

\*

0	0	0
0	1	0
0	0	0

=

0	1	1	1	0
1	1	1	1	1
1	1	1	0	0
1	1	1	1	1
0	1	1	1	0

Output image



# Convolution

0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	1	1	1	1	1	0
0	1	1	1	0	0	0
0	1	1	1	1	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

Input image

*	$\frac{1}{9}$	1	1	1
		1	1	1
		1	1	1

=	$\frac{1}{9}$	3	5	6	5	3
		5	8	8	6	3
		6	9	8	7	4
		5	8	8	6	3
		3	5	6	5	3

Output image

# Convolution

0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	1	1	1	1	1	0
0	1	1	1	0	0	0
0	1	1	1	1	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

Input image

\*

-1	-1	-1
2	2	2
-1	-1	-1

=

0	1	3	1	0
1	1	1	3	3
0	0	-2	-4	-4
1	1	1	3	3
0	1	3	1	0

Output image

# Convolution

Very commonly used for:

- Blurring an image



$$\begin{matrix} * & \begin{matrix} \begin{matrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{matrix} \end{matrix} & = \end{matrix}$$



- Finding edges



$$\begin{matrix} * & \begin{matrix} \begin{matrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{matrix} \end{matrix} & = \end{matrix}$$



(this example finds horizontal edges)

# Convolution Layer

Activation layer  
(such as ReLU)

Conv2d  
layer



1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9



add bias

apply  
activation

-1	-1	-1
2	2	2
-1	-1	-1



add bias

apply  
activation

convolve with  
each filter

0	-1	0
-1	4	-1
0	-1	0

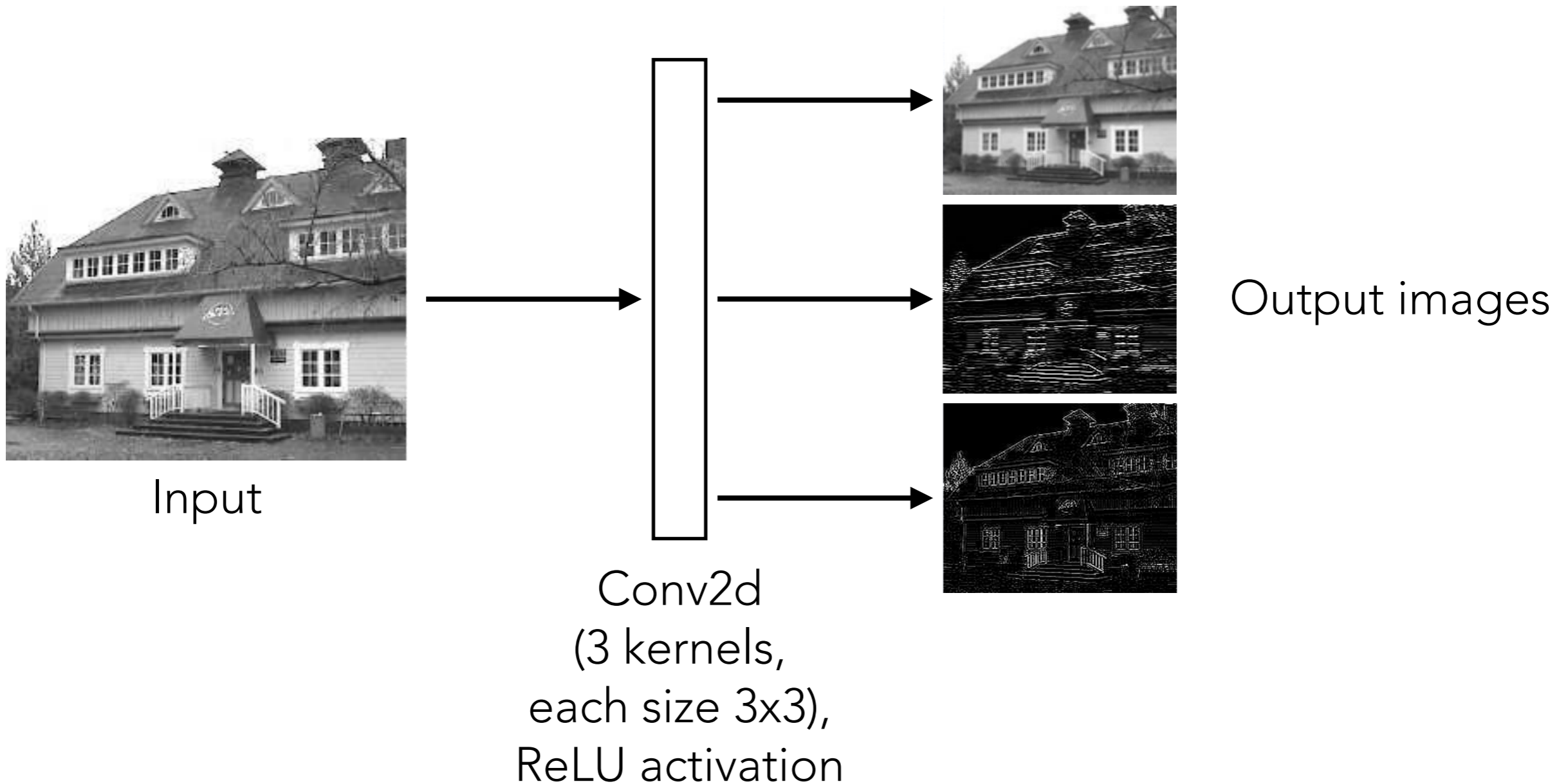


add bias

apply  
activation

filters & biases (1 bias number per filter)  
are unknown and are learned!

# Convolution Layer

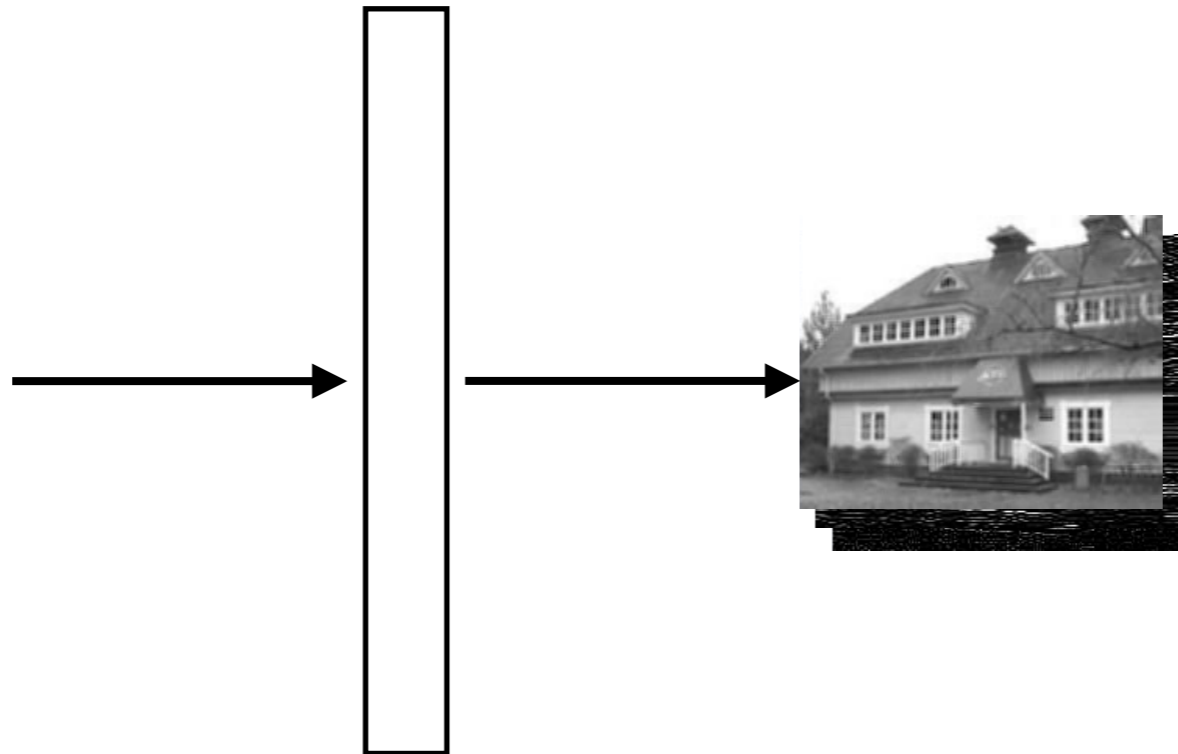


# Convolution Layer



Input

shape:  
1 (# channels),  
height,  
width



Conv2d  
(3 kernels,  
each size 3x3),  
ReLU activation



Stack output  
images into a  
single "output  
feature map"

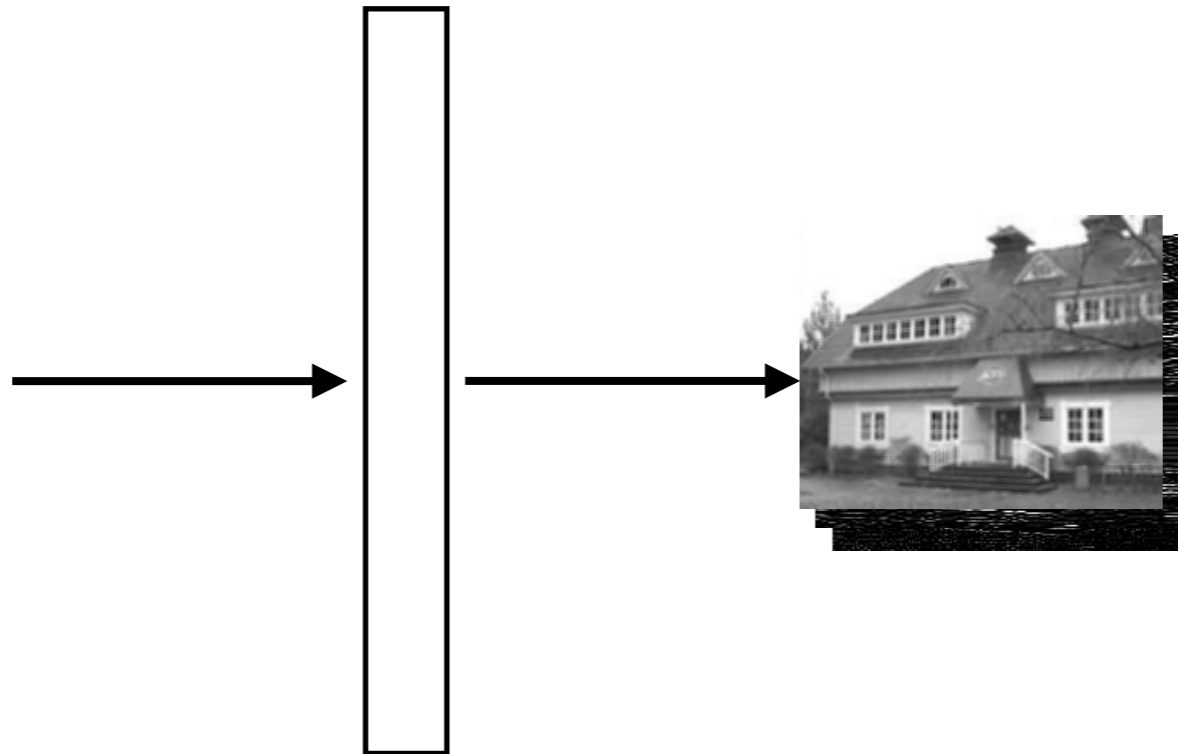
shape:  
3,  
height-2,  
width-2

# Convolution Layer



Input

shape:  
1 (# channels),  
height,  
width



Conv2d  
( $k$  kernels  
each size  $3 \times 3$ ),  
ReLU activation



Stack output  
images into a  
single "output  
feature map"

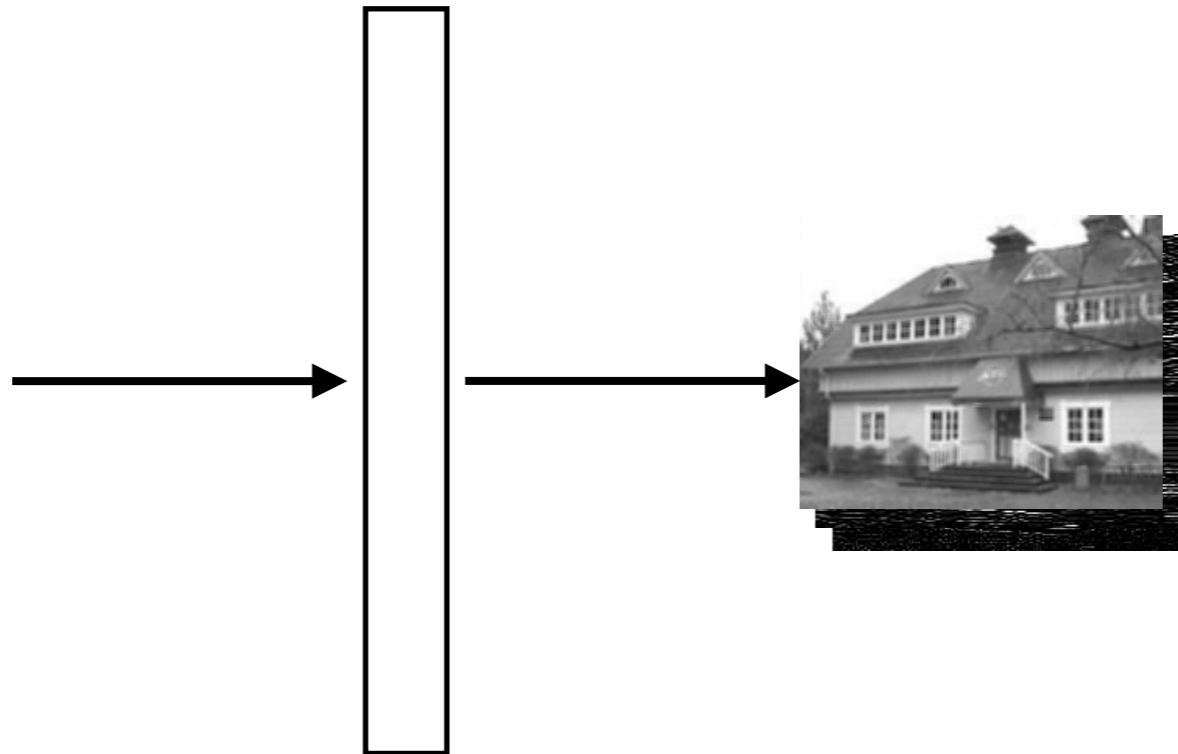
shape:  
 $k$ ,  
height-2,  
width-2

# Convolution Layer



Input

shape:  
 $d$  (# channels)  
height,  
width



Conv2d  
( $k$  kernels  
each size  $d \times 3 \times 3$ ),  
ReLU activation



Stack output  
images into a  
single "output  
feature map"

shape:  
 $k$ ,  
height-2,  
width-2

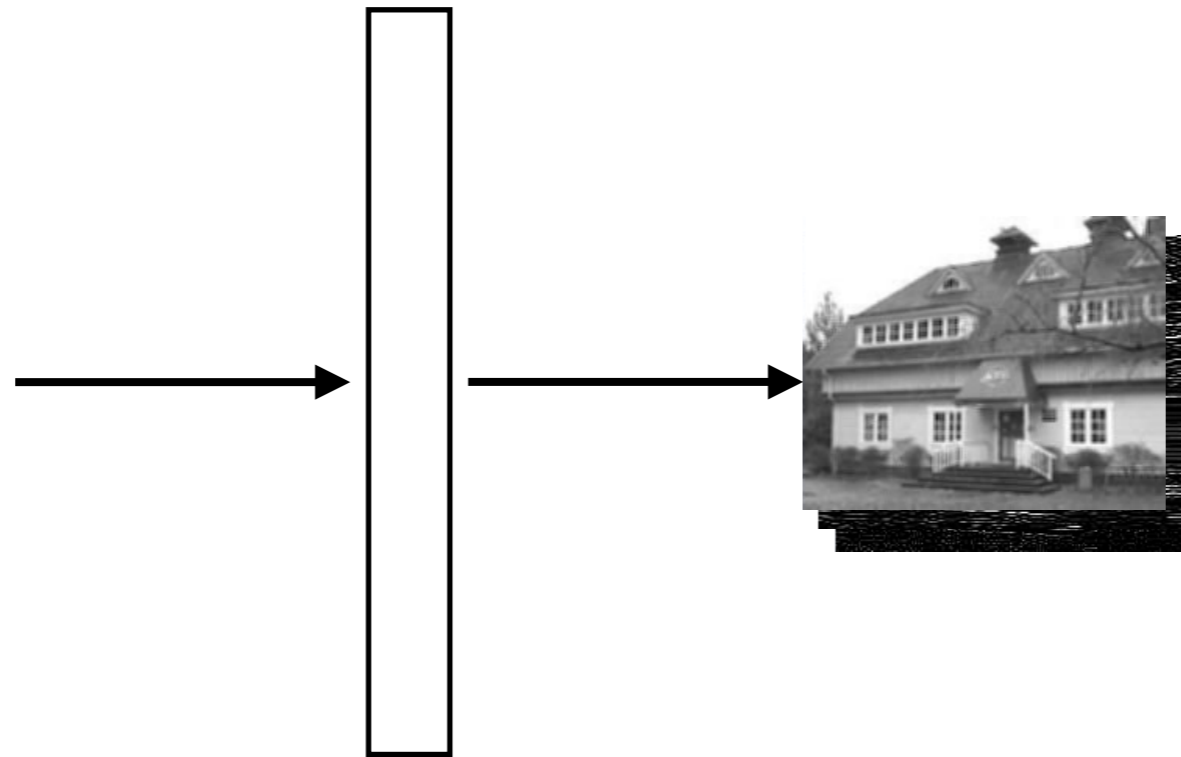


# Convolution Layer



Input

shape:  
 $d$  (# channels)  
height,  
width



Conv2d  
( $k$  kernels  
each size  $d \times 3 \times 3$ ),  
ReLU activation

Stack output  
images into a  
single "output  
feature map"

shape:  
 $k$ ,  
height-2,  
width-2

Each filter:

