Carnegie Mellon University Heinz College

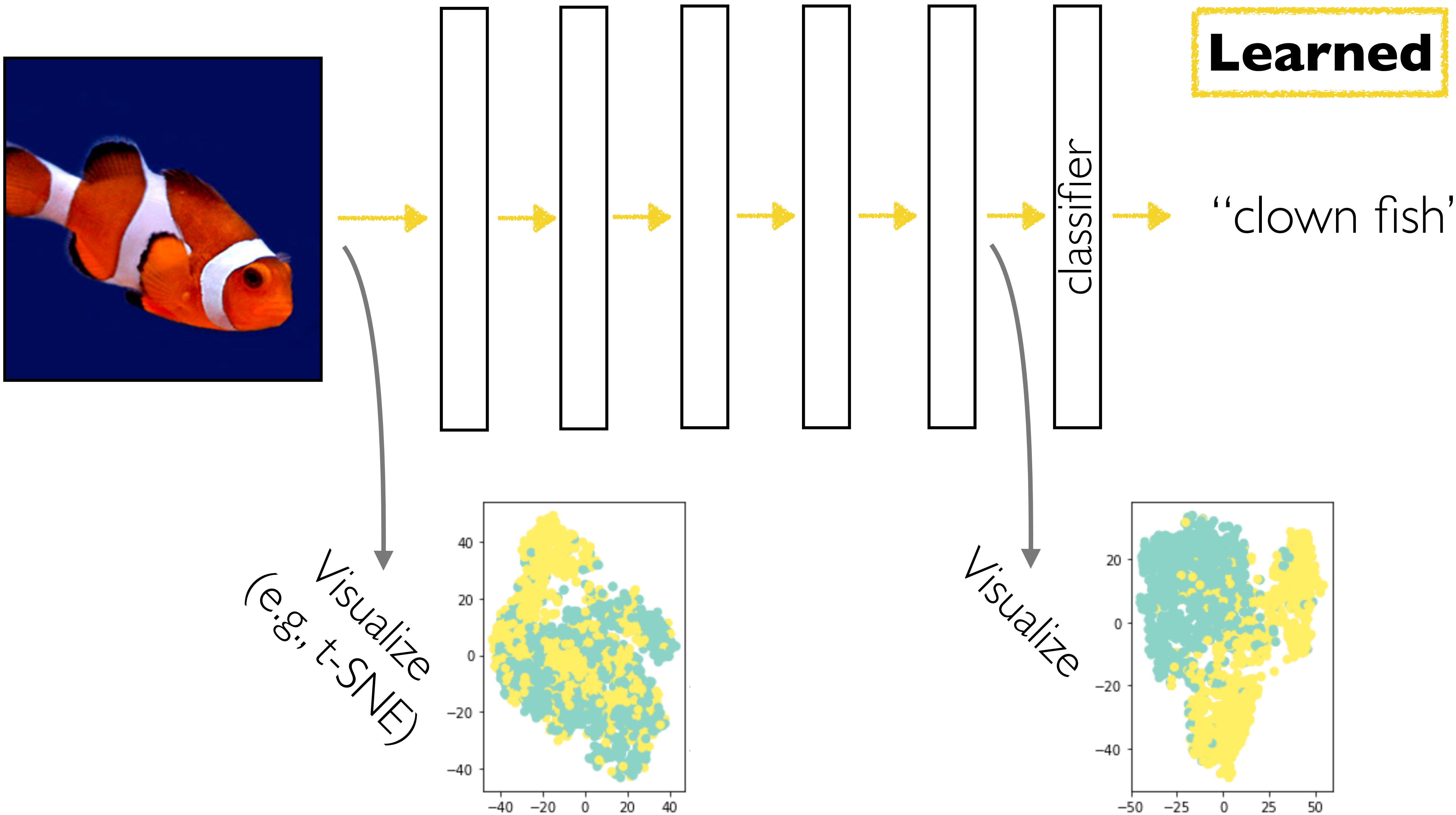
Unstructured Data Analysis for Policy

Lecture 11: Neural nets & deep learning

George Chen

(Last Time) Representation Learning

Each layer's output is *another way we could represent the input data*



Why Does Deep Learning Work?

Actually the ideas behind deep learning are old (~1980's)

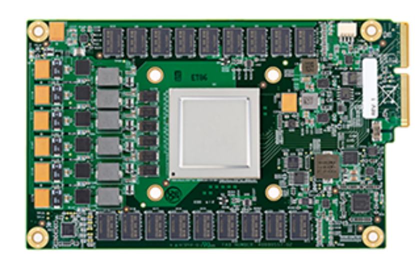
There's even a patent from 1961 that basically amounts to a convolutional neural net for OCR

- Big data

- Better hardware

CPU's
& Moore's law

GPU's



TPU's

- Better algorithms

Many companies now make dedicated hardware for deep nets (e.g., Google, Apple, Tesla)

Structure Present in Data Matters

Neural nets aren't doing black magic

- **Image analysis:** convolutional neural networks (convnets) neatly <u>incorporates basic image processing structure</u>

- **Time series analysis:** recurrent neural networks (RNNs) <u>incorporates ability to remember and forget things over time</u>

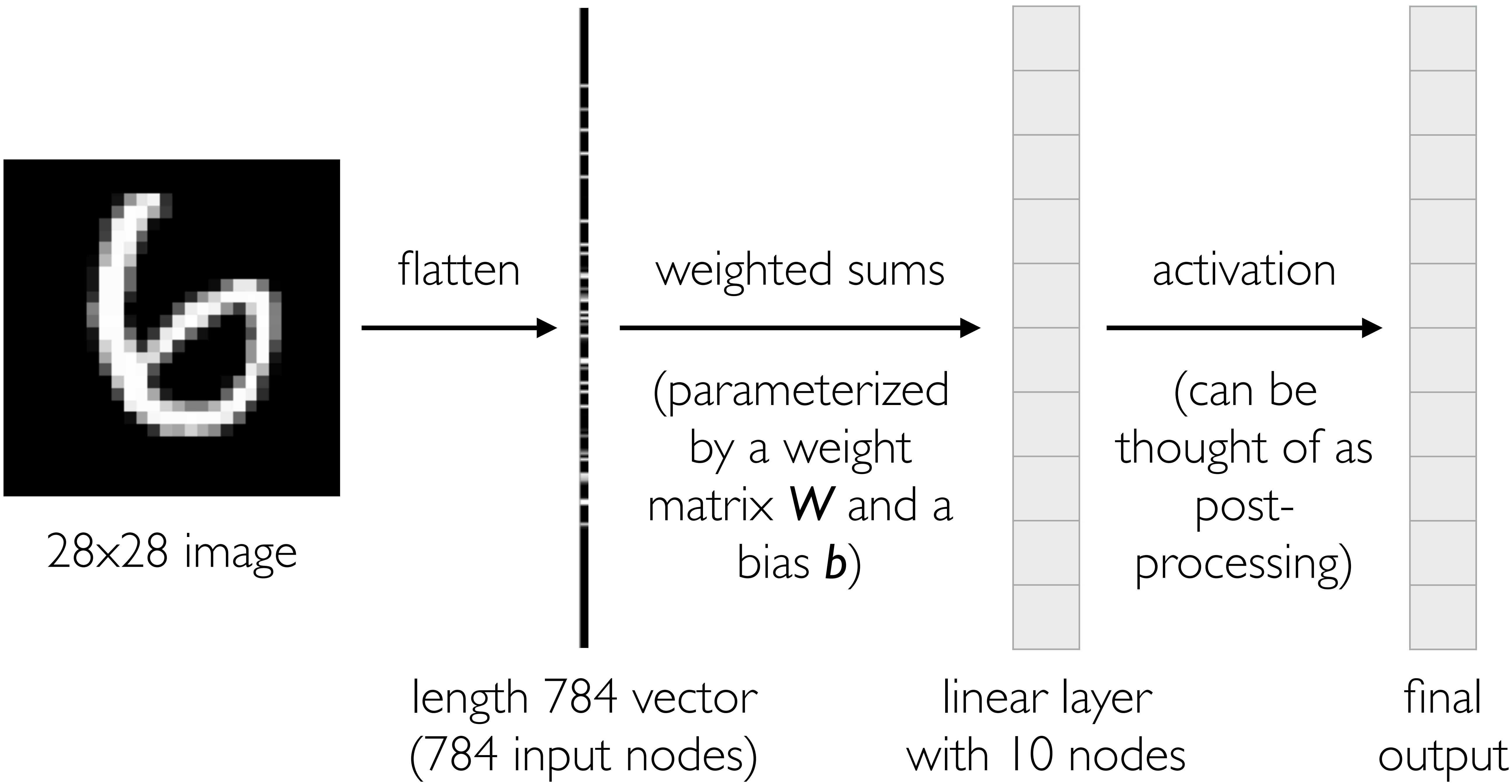
  - Note: text is a time series

  - Note: video is a time series

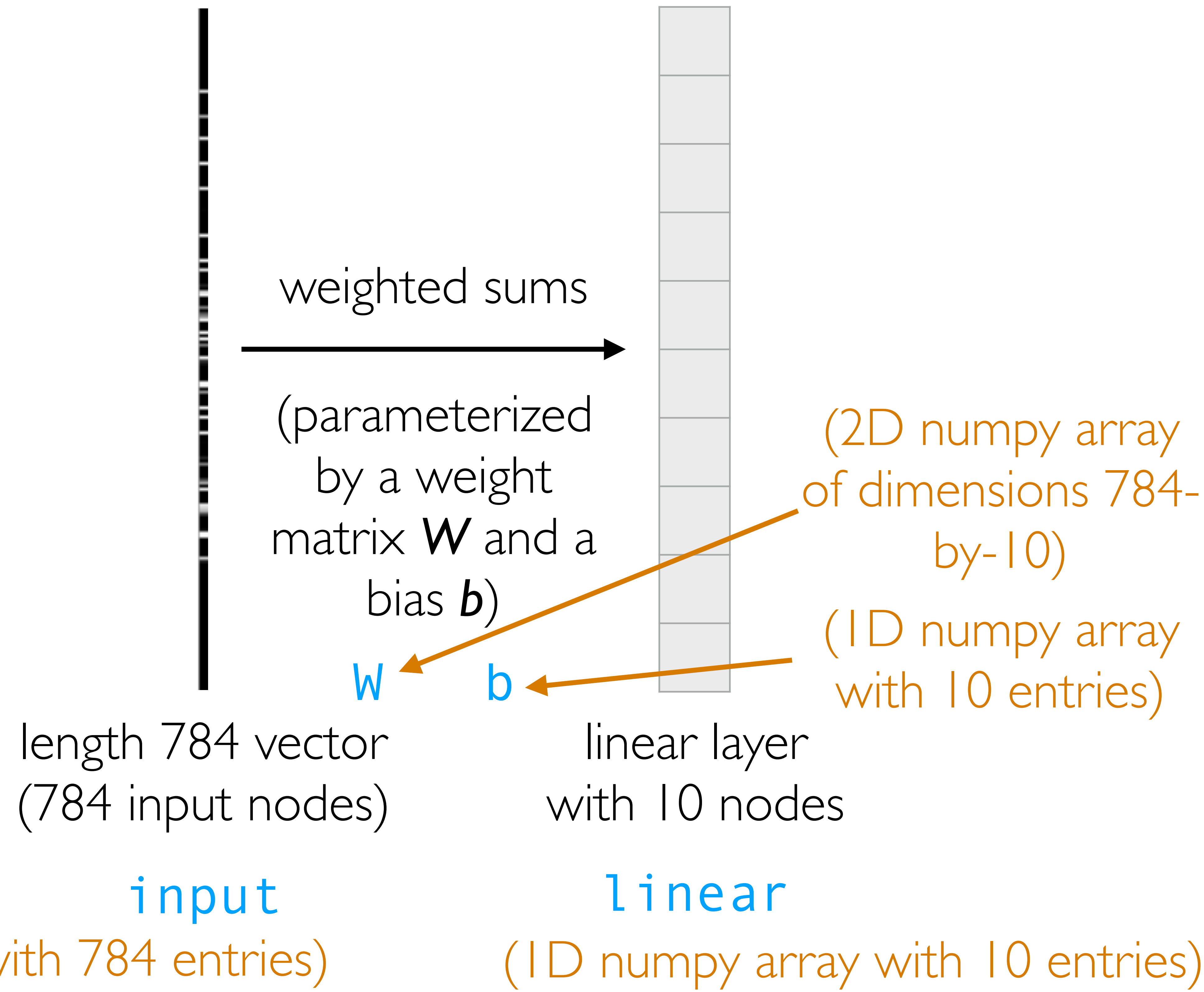
Handwritten Digit Recognition Example

Walkthrough of 2 extremely simple neural nets

Handwritten Digit Recognition



Handwritten Digit Recognition



Handwritten Digit Recognition

```
linear[0] = np.dot(input, W[:, 0]) + b[0]
linear[1] = np.dot(input, W[:, 1]) + b[1]
```

⋮

$$\texttt{linear[j]} = \sum_{i=0}^{783} \texttt{input[i]} \times \texttt{W[i,j]} + \texttt{b[j]}$$

weighted sums

(parameterized by a weight matrix ***W*** and a bias ***b***)

W (2D numpy array of dimensions 784-by-10)

b (1D numpy array with 10 entries)

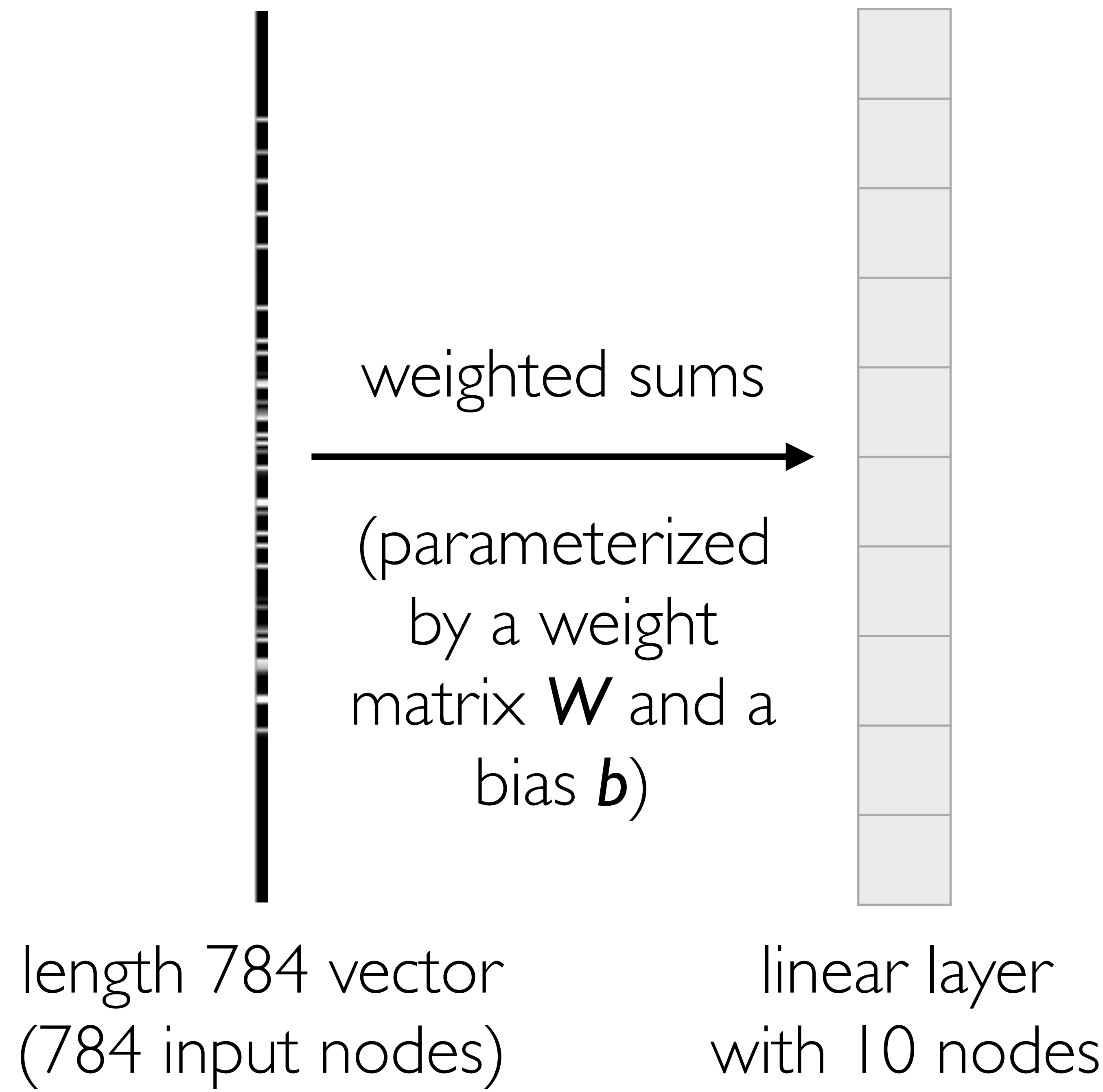
784 vector
nput nodes)

nput
- entries)

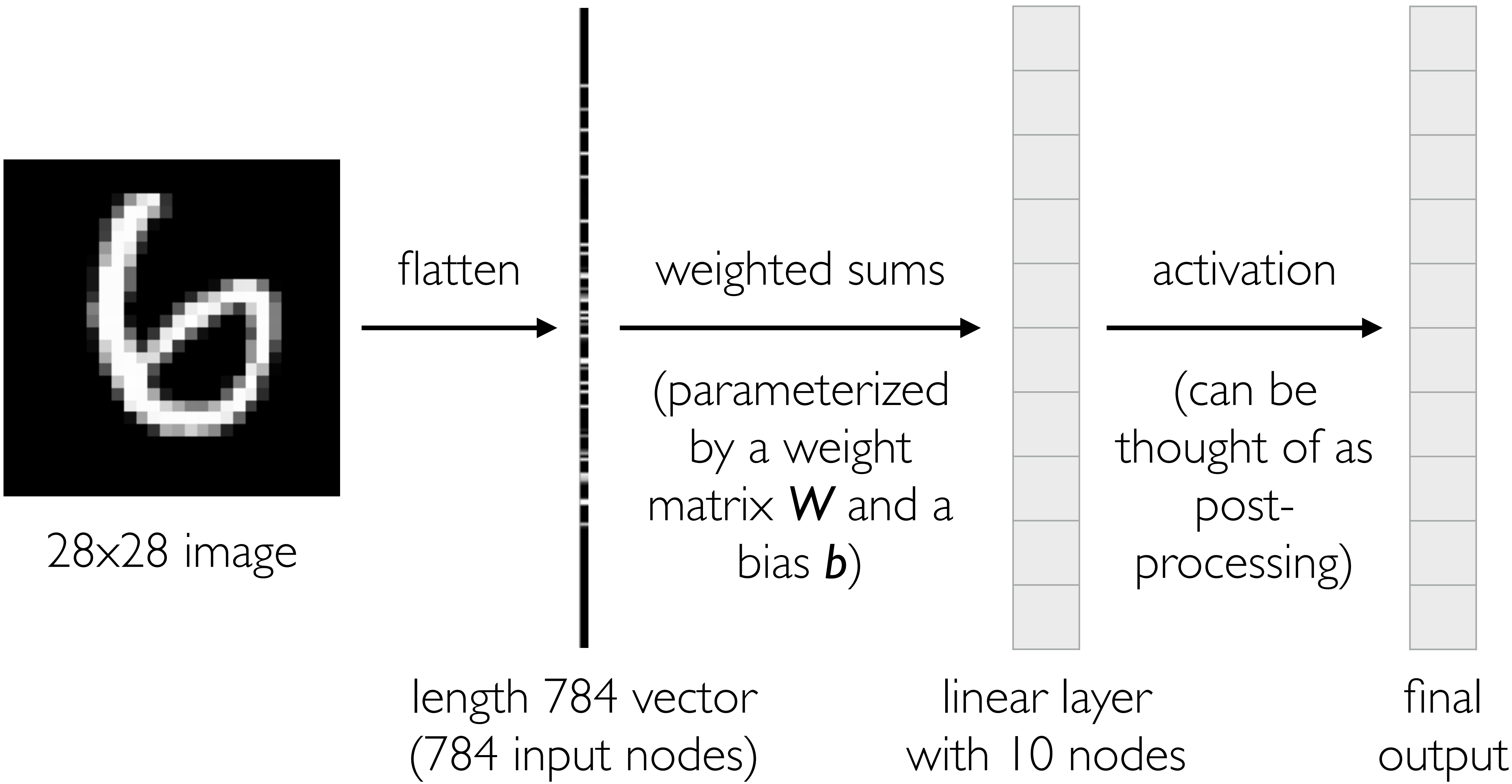
linear layer
with 10 nodes

linear
(1D numpy array with 10 entries)

Handwritten Digit Recognition



Handwritten Digit Recognition

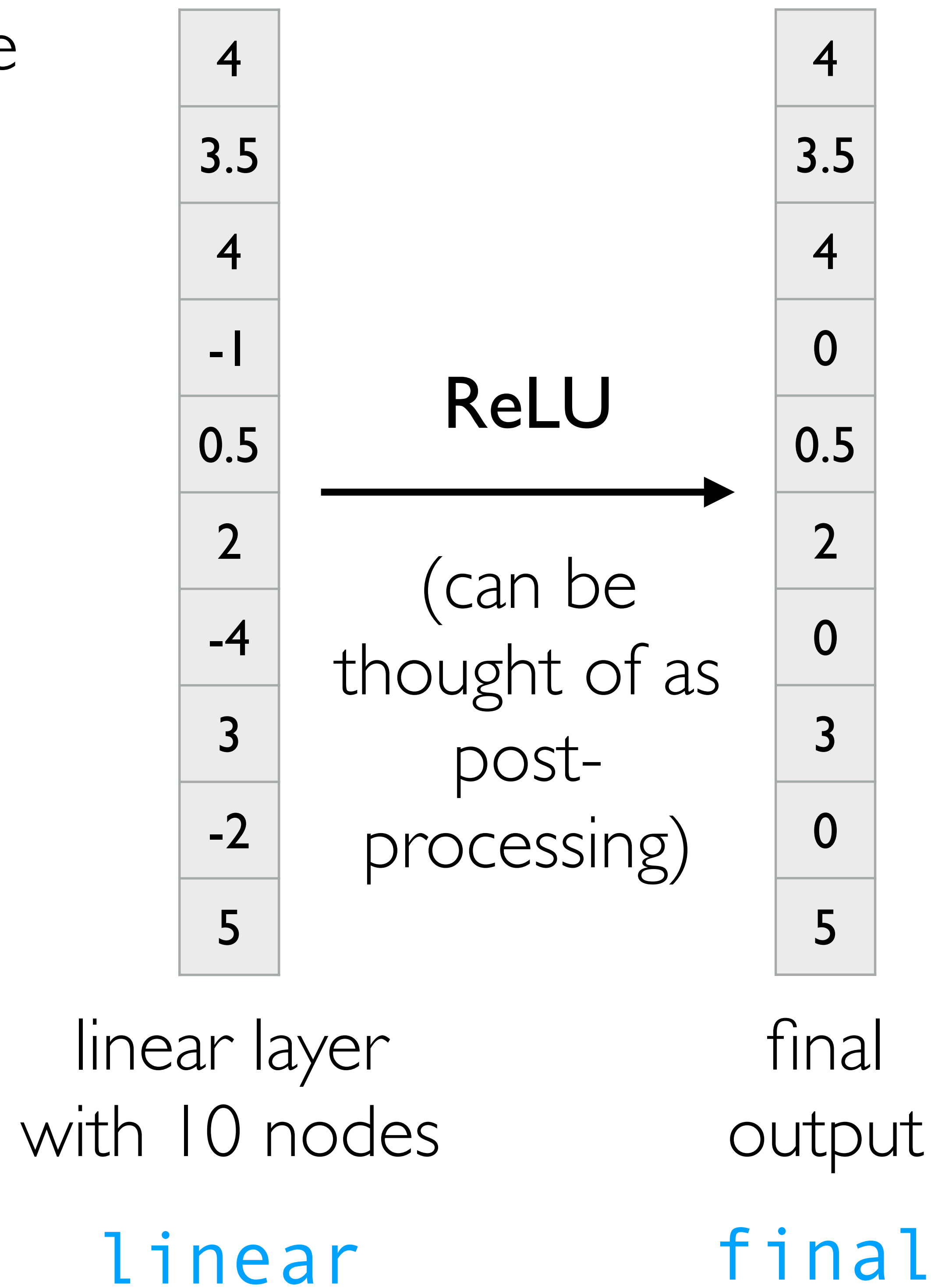


Handwritten Digit Recognition

Many different activation functions possible

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

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

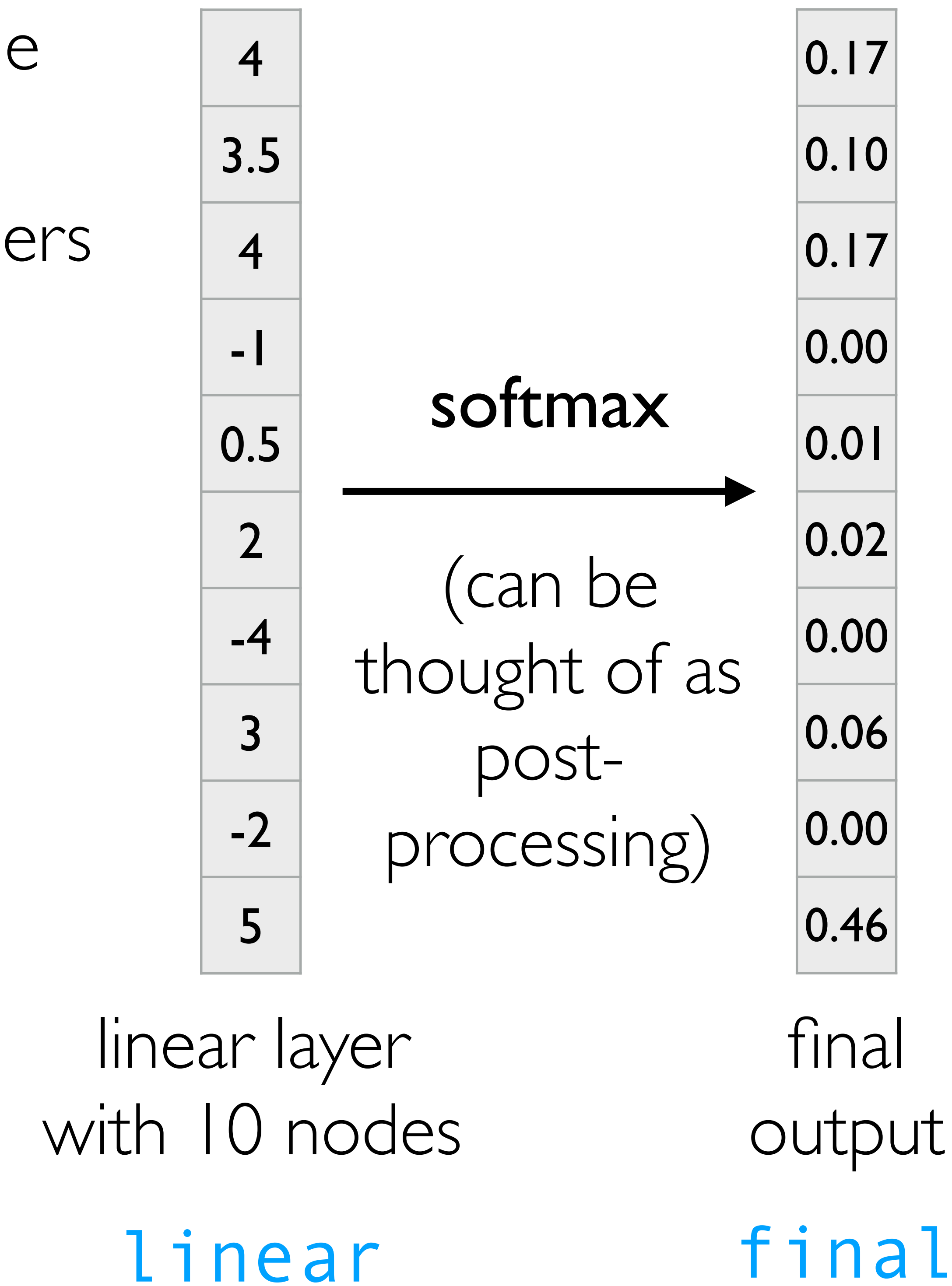


Handwritten Digit Recognition

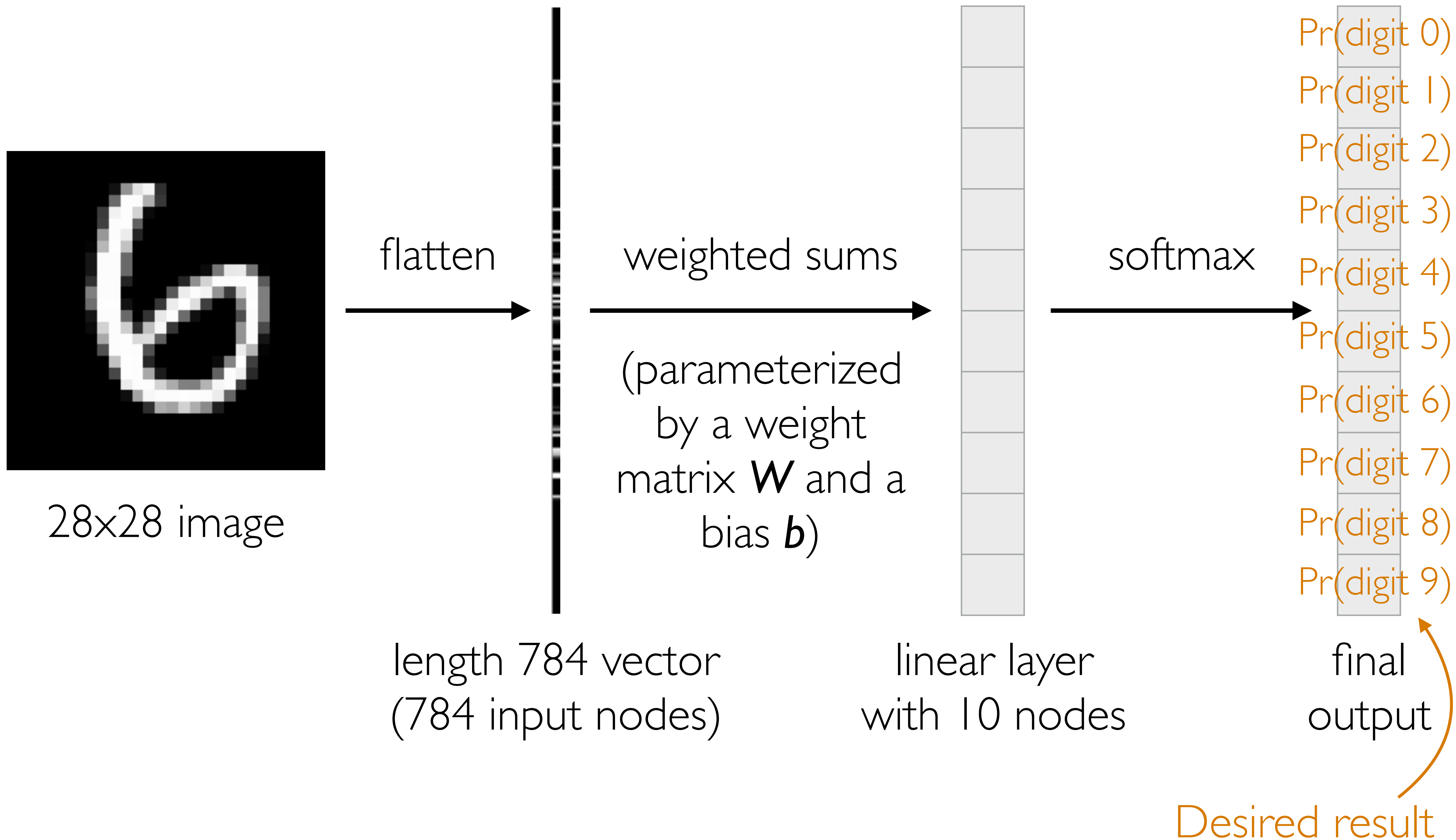
Many different activation functions possible

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

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

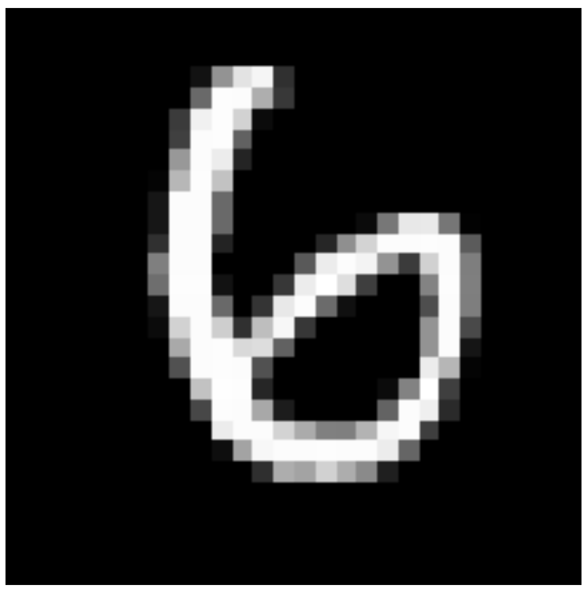


Handwritten Digit Recognition



Handwritten Digit Recognition

Training label: 6



Input

Flatten

Linear (10 nodes)

Softmax

Loss/"error"

error

Error is averaged across training examples

Popular loss function for classification: **categorical cross entropy**

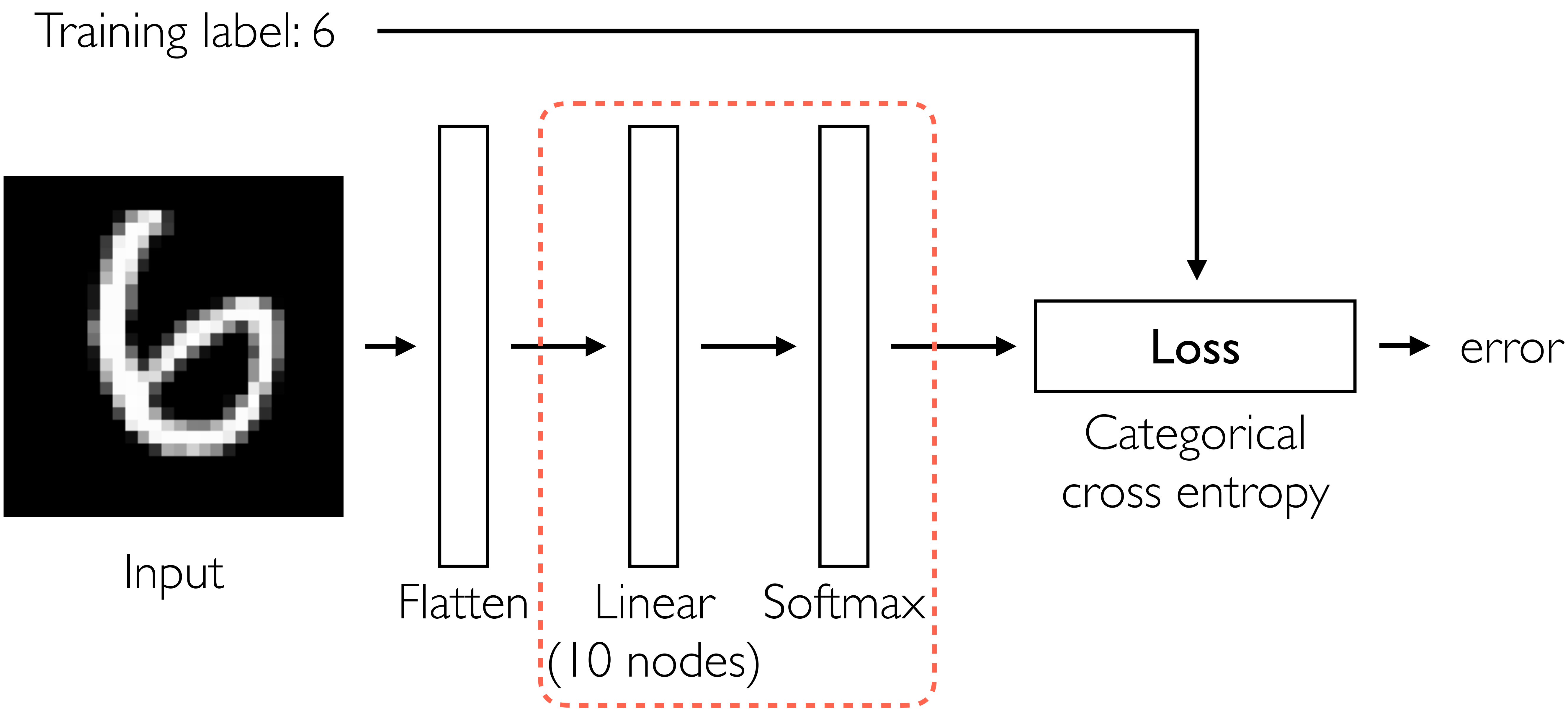
$$\log \frac{1}{\text{estimated Pr(digit 6)}}$$

Learning this neural net means learning ***W*** and ***b***

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

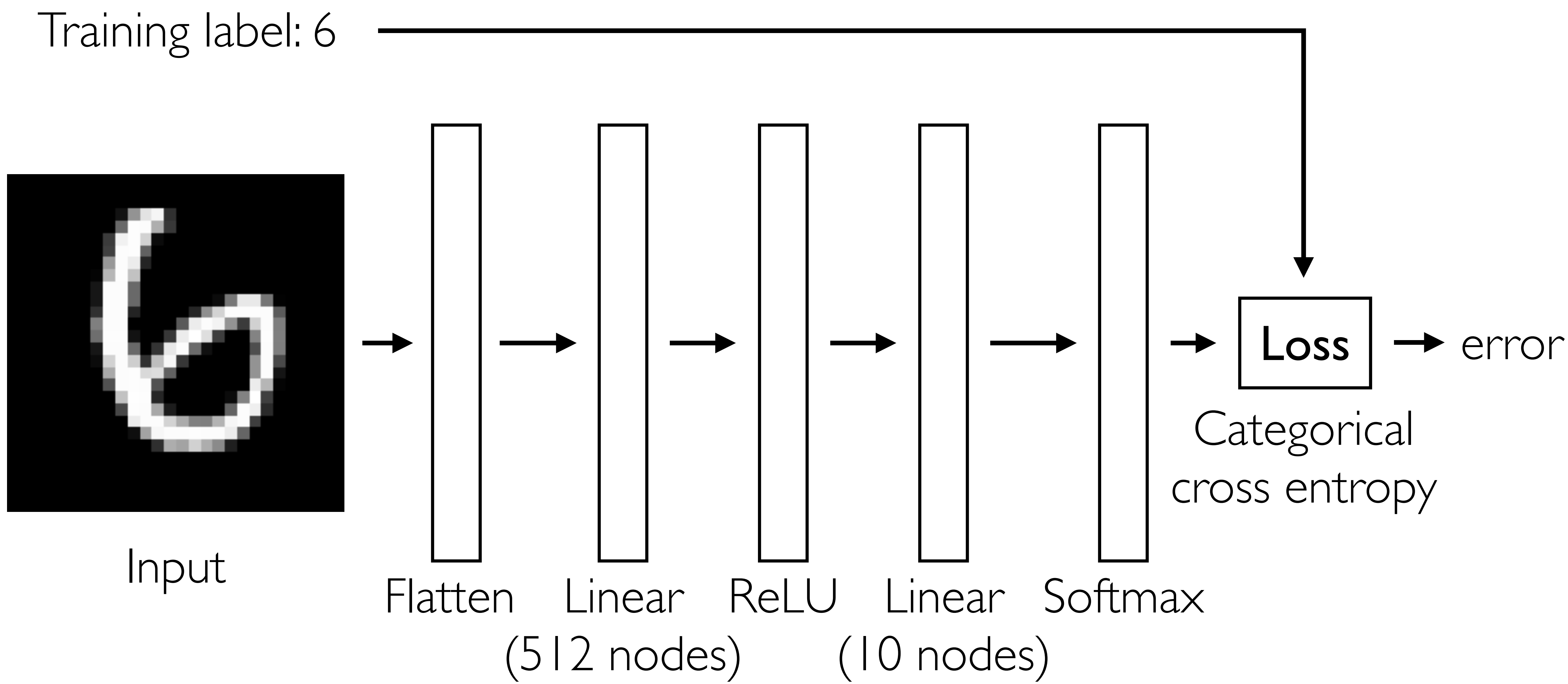
⚠️ In PyTorch, softmax is included as part of the cross entropy loss

Handwritten Digit Recognition



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

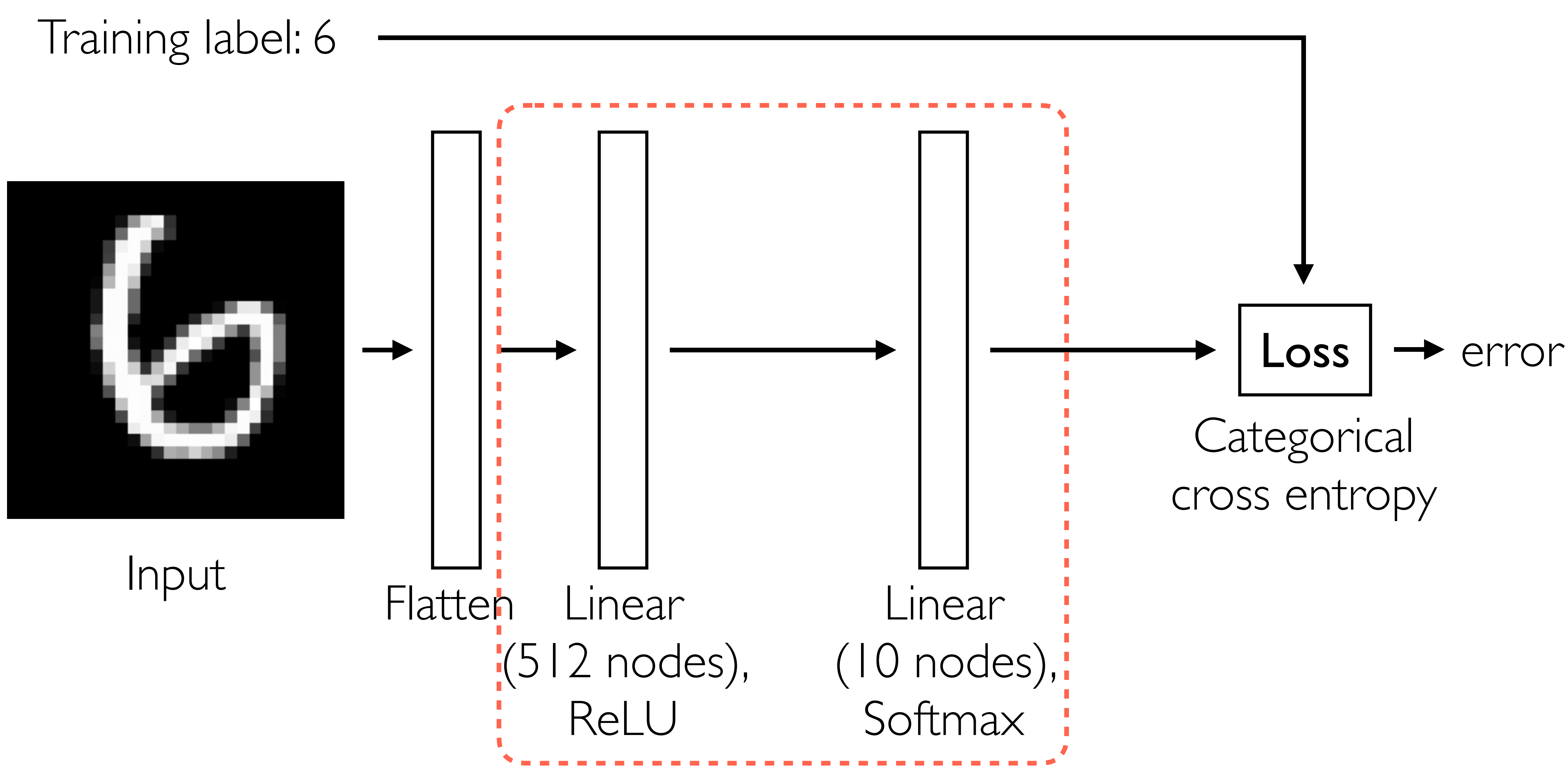
Handwritten Digit Recognition



Learning this neural net means learning parameters of both linear layers!

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

Handwritten Digit Recognition



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 will some times use this notation instead

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)
- What's the big difference then? Why not just use NumPy?
    - PyTorch does not use NumPy arrays and instead uses tensors (so instead of `np.array`, you use `torch.tensor`)
        - 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 keep track of "gradient" information (we'll discuss more about what this means in a few lectures)

PyTorch code is often harder to debug than NumPy code

There's a PyTorch tutorial posted in supplemental reading

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!
  - Can choose between different options using hyperparameter selection strategy from earlier lectures
    - 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 for image classification/object recognition)

PyTorch GitHub Has Lots of Examples

github.com/pytorch/examples

PyTorch Examples

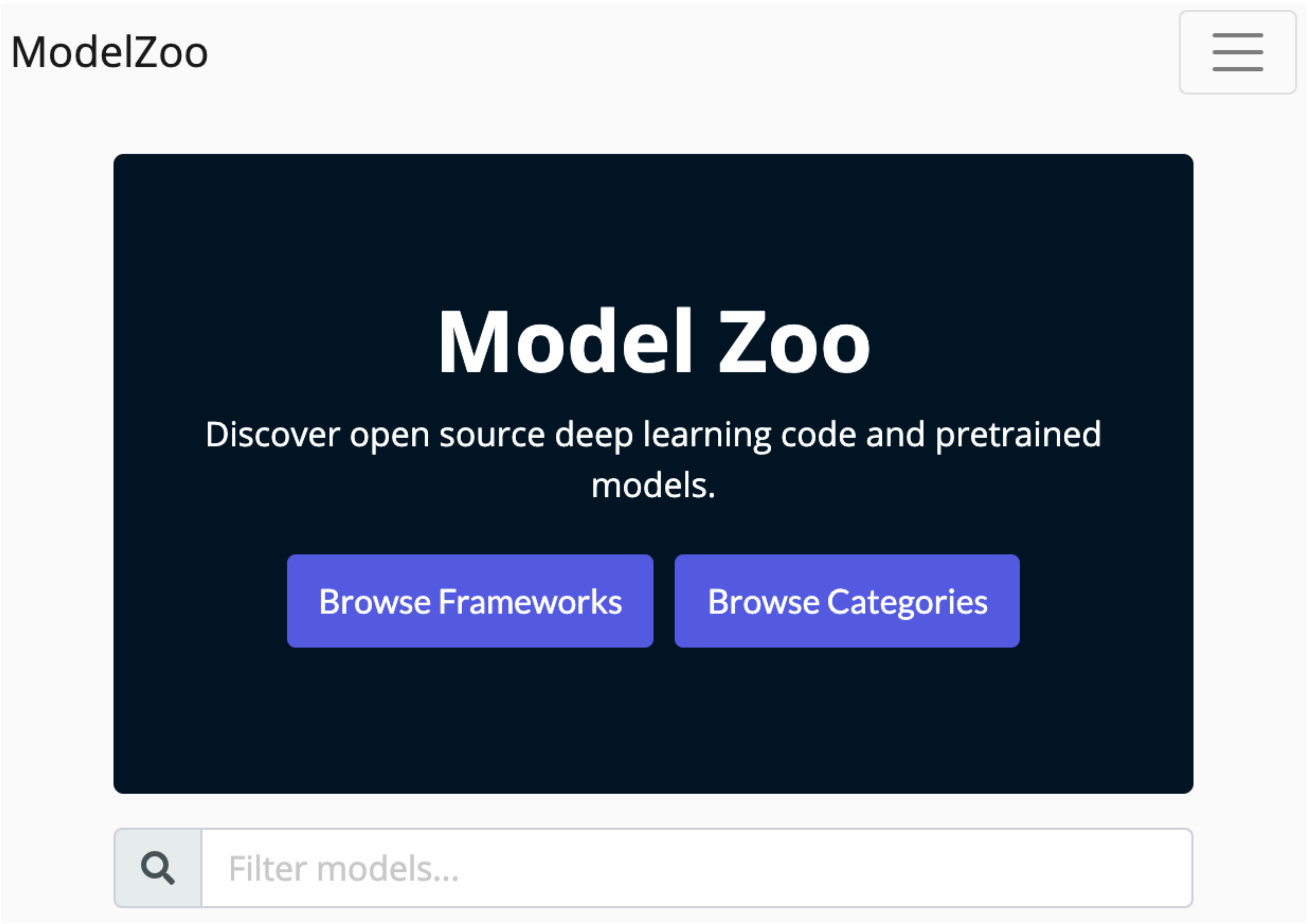
A repository showcasing examples of using PyTorch

- Image classification (MNIST) using Convnets
- Word level Language Modeling using LSTM RNNs
- Training Imagenet Classifiers with Residual Networks
- Generative Adversarial Networks (DCGAN)
- Variational Auto-Encoders
- Superresolution using an efficient sub-pixel convolutional neural network
- Hogwild training of shared ConvNets across multiple processes on MNIST
- Training a CartPole to balance in OpenAI Gym with actor-critic
- Natural Language Inference (SNLI) with GloVe vectors, LSTMs, and torchtext
- Time sequence prediction - use an LSTM to learn Sine waves
- Implement the Neural Style Transfer algorithm on images
- Several examples illustrating the C++ Frontend

Additionally, a list of good examples hosted in their own repositories:

- Neural Machine Translation using sequence-to-sequence RNN with attention (OpenNMT)

Find a Massive Collection of Models at the Model Zoo



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) + 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) + b)
```

Multilayer perceptron:

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

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

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

Complexity of a Neural Net?

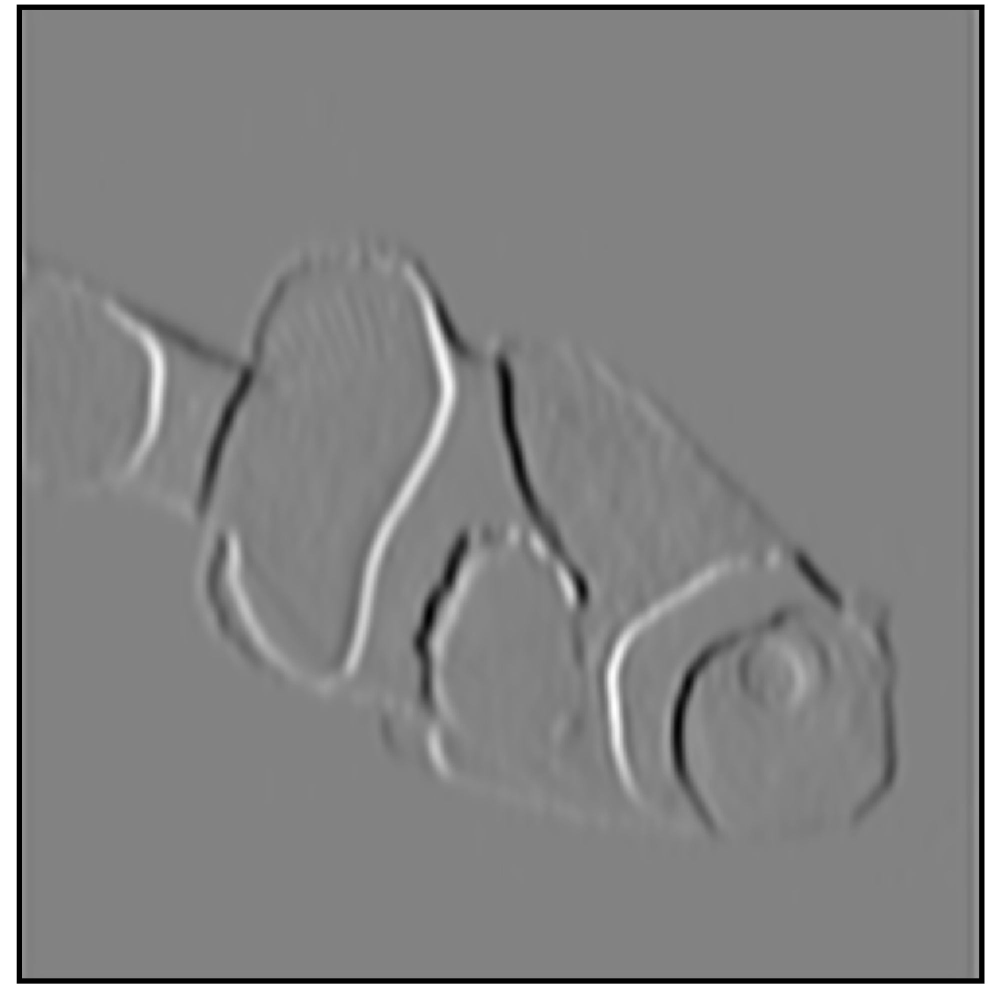
- Increasing number of layers (depth) makes neural net more "complex"
  - Learn computer program that has more lines of code
  - Sometimes, more parameters may be needed
    - If so, more training data may be needed

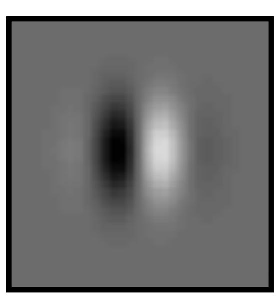
Earlier: multinomial logistic regression had fewer parameters than multilayer perceptron example

Upcoming: we'll see examples of deep nets with *fewer* parameters than "shallower" nets

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

Convolution



filter 

Slide by Phillip Isola

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

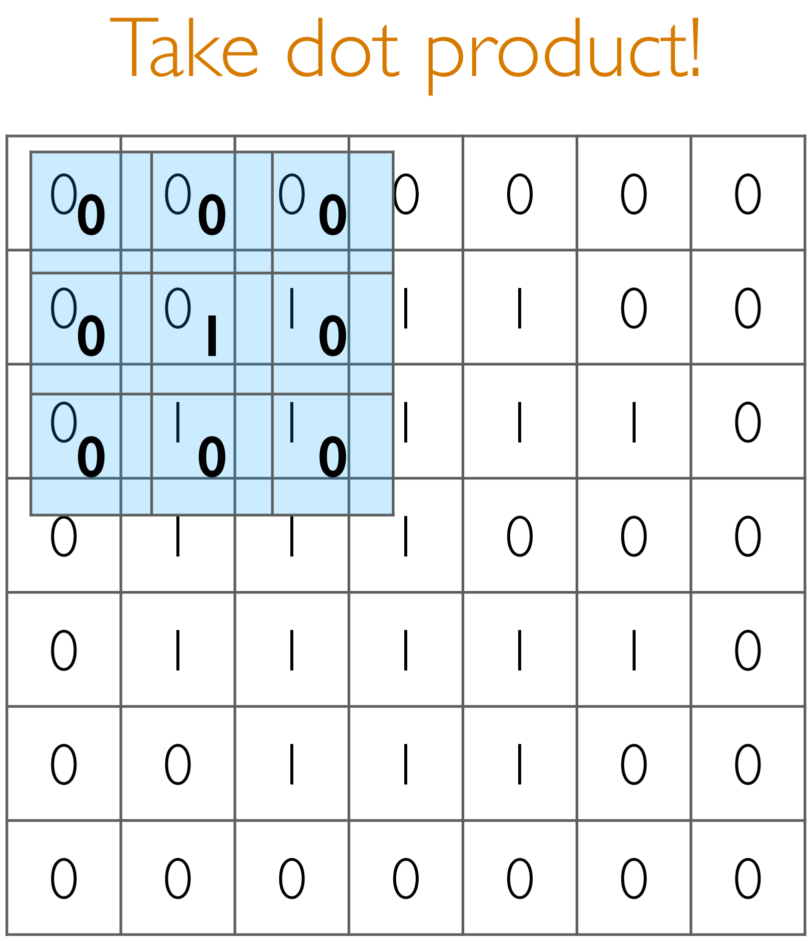
0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	1	1	1	1	1	0
0	1	1	1	0	0	0
0	1	1	1	1	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

Input image

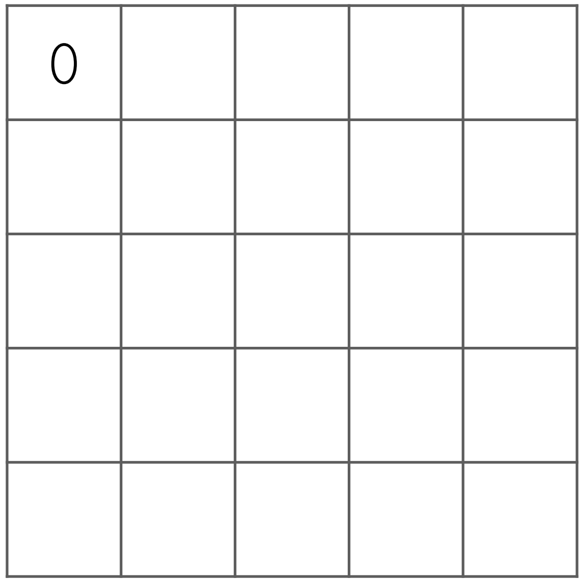
0	**0**	**0**
0	**1**	**0**
0	**0**	**0**

Filter
(also called “kernel”)

Convolution



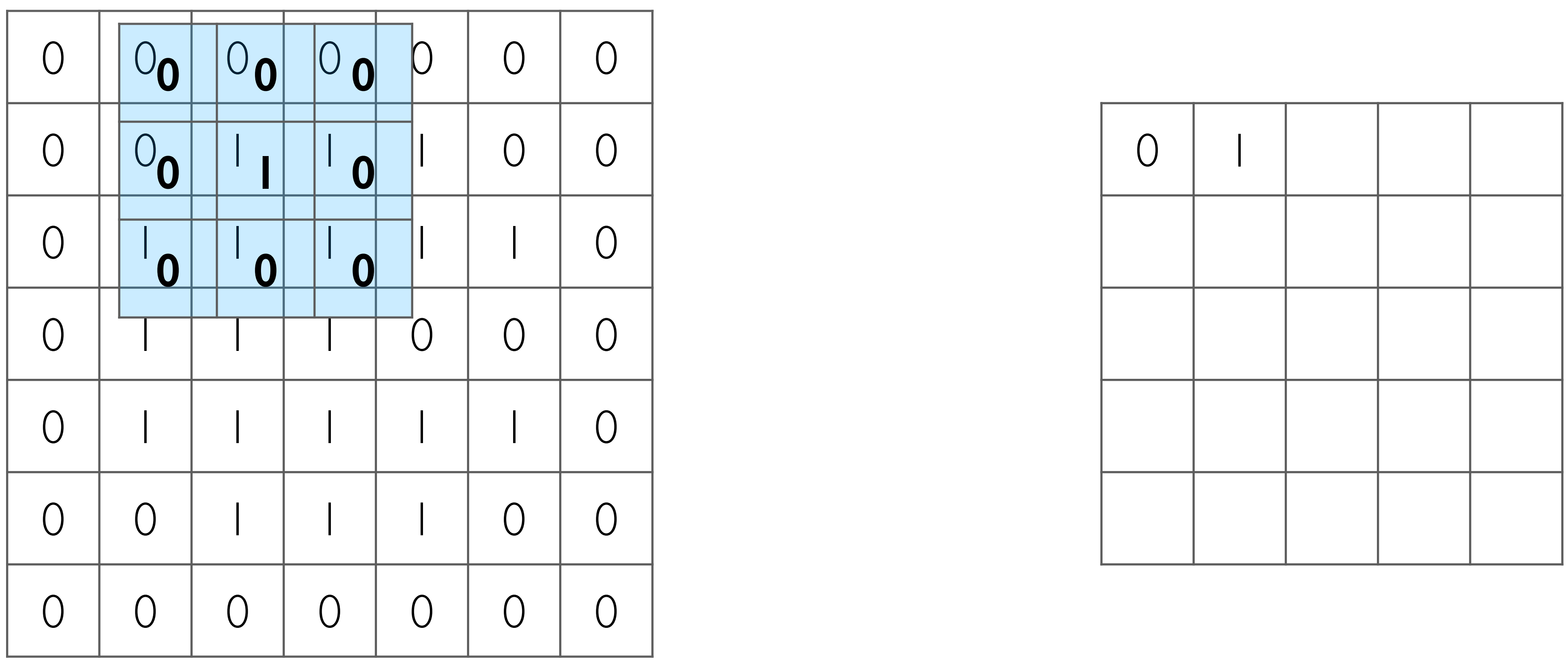
Input image



Output image

Convolution

Take dot product!

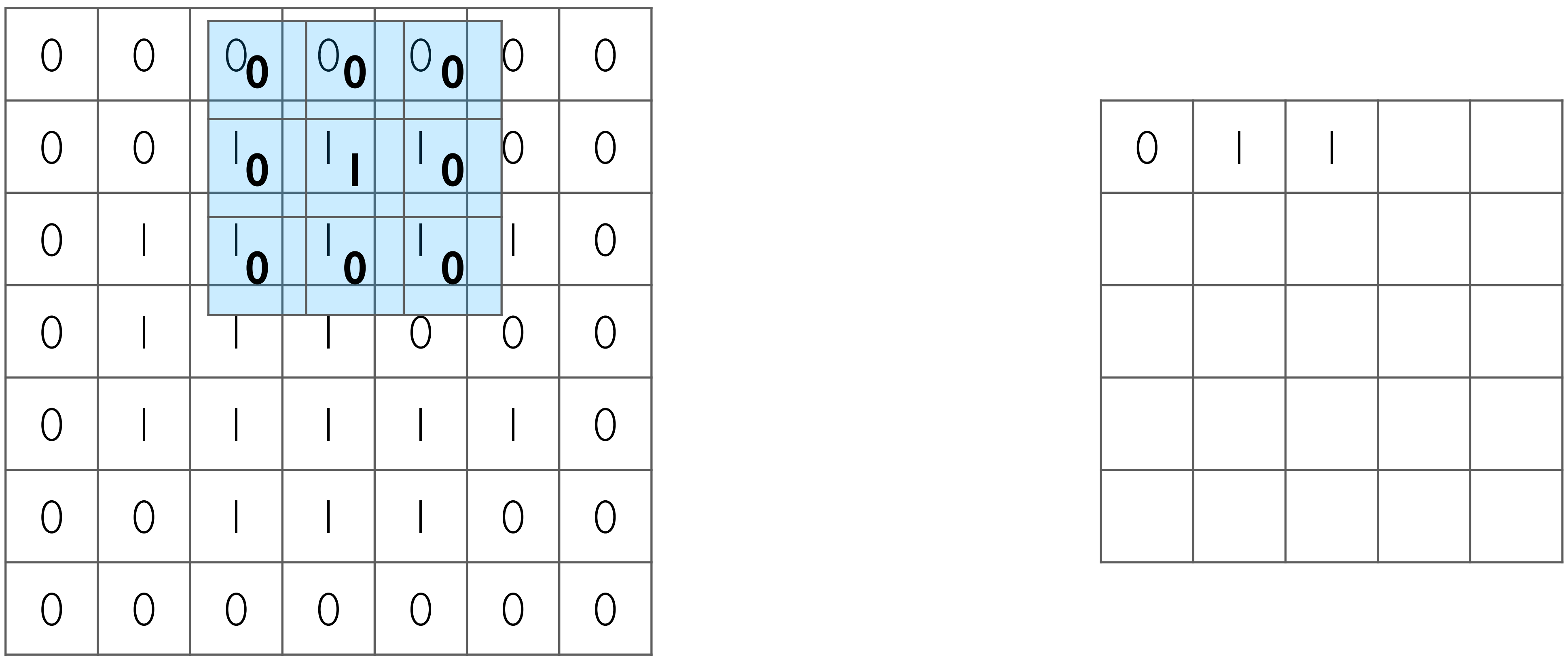


Input image

Output image

Convolution

Take dot product!

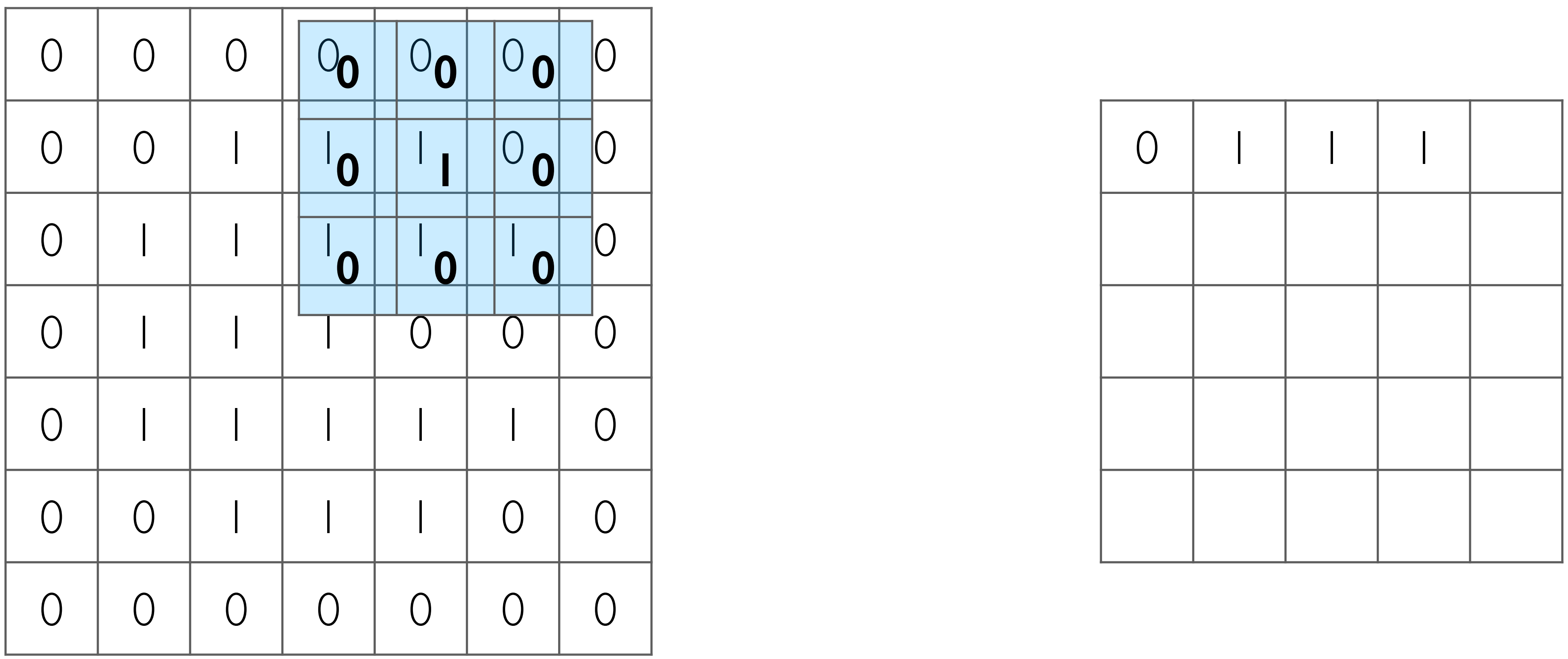


Input image

Output image

Convolution

Take dot product!

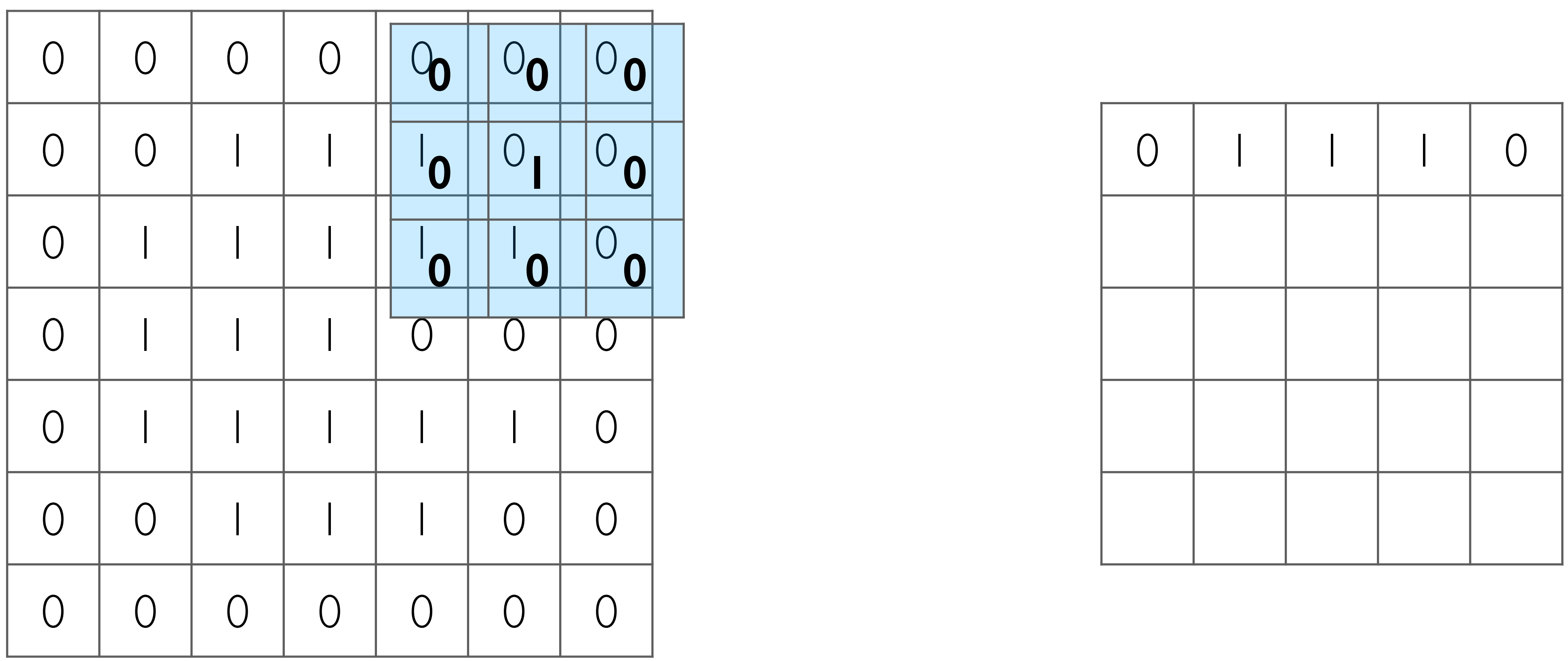


Input image

Output image

Convolution

Take dot product!



Input image

Output image

Convolution

Take dot product!

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

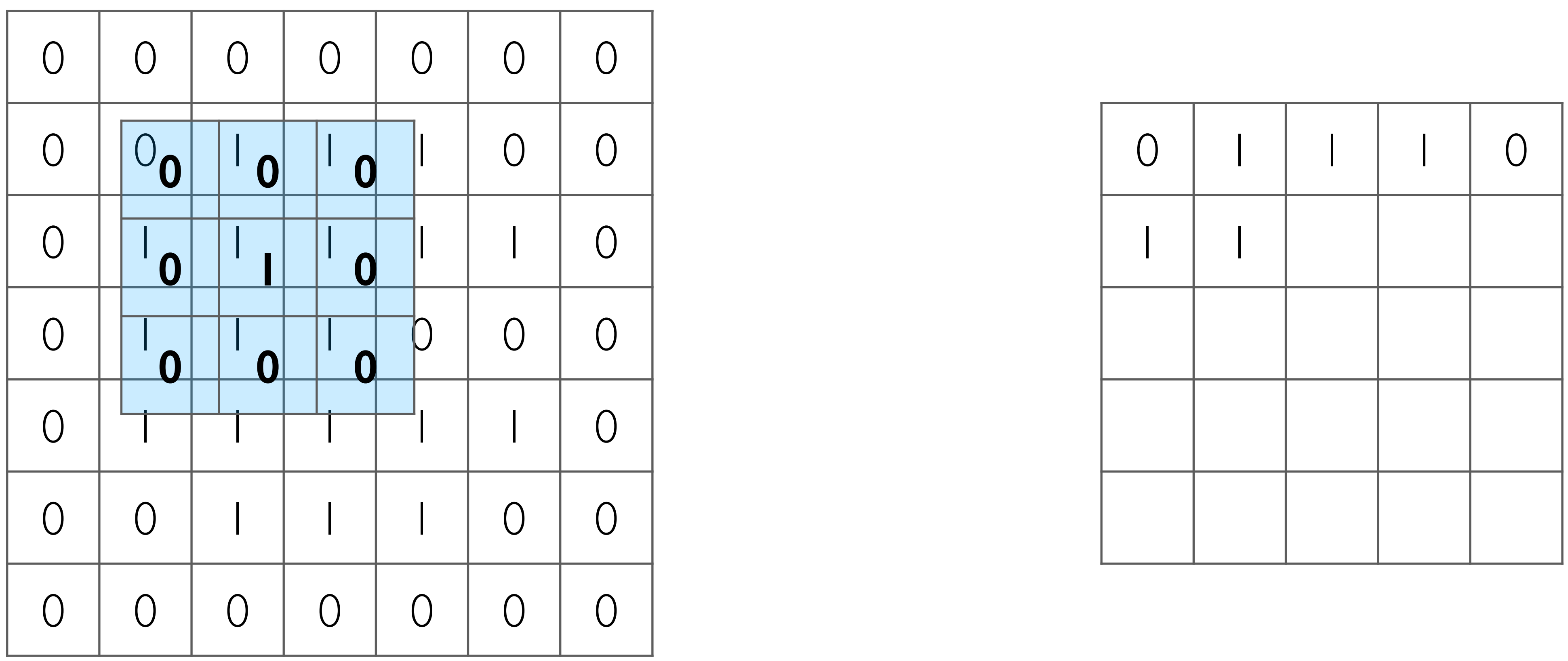
Input image

0	1	1	1	0
1				

Output image

Convolution

Take dot product!



Input image

Output image

Convolution

$$
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
*
\begin{bmatrix}
\mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{1} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0}
\end{bmatrix}
=
\begin{bmatrix}
0 & 1 & 1 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 0
\end{bmatrix}
$$

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

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	0	0	0	0	0	0	0	**0**
0	0	0	1	1	1	0	0	**0**
0	0	1	1	1	1	1	0	**0**
0	0	1	1	1	0	0	0	**0**
0	0	1	1	1	1	1	0	**0**
0	0	0	1	1	1	0	0	**0**
0	0	0	0	0	0	0	0	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

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

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \frac{1}{9}\begin{bmatrix} 3 & 5 & 6 & 5 & 3 \\ 5 & 8 & 8 & 6 & 3 \\ 6 & 9 & 8 & 7 & 4 \\ 5 & 8 & 8 & 6 & 3 \\ 3 & 5 & 6 & 5 & 3 \end{bmatrix}$$

Input image

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

*

-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

Input image

Output image

Convolution

Very commonly used for:

- Blurring an image

*

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

=

- Finding edges

*

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

=

(this example finds horizontal edges)

Images from: http://aishack.in/tutorials/image-convolution-examples/