

**ECE408 / CS483 / CSE 408 Fall 2022**

# **From Machine Learning to Deep Learning**

# Course Reminders

- Grading of Labs to be completed by end of week
- Midterm 1 is on Tuesday, October 11<sup>th</sup>
  - On-line, everybody will be taking it at the same time
    - Tuesday, Oct. 11th 7:00pm-8:20pm US Central time
    - Wednesday, Oct. 12th 8:00am-9:20am Beijing time
  - Includes materials from Lecture 1 through Lecture 10
  - If you have a legit conflict, please let us know by Friday Sept 30th
- Project Milestone 1: Baseline CPU implementation is due Friday 8pm US CT October 14<sup>th</sup>
  - Project details to be posted this week on course wiki

# Perceptron is a Simple Example

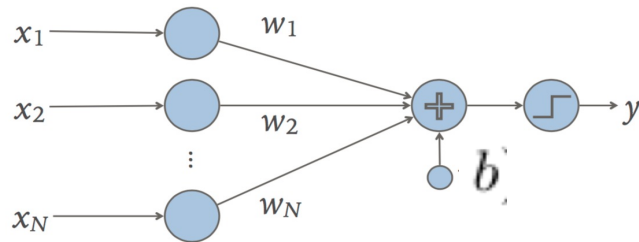
- Example: a **perceptron**

$$y = \text{sign}(W \cdot x + b)$$

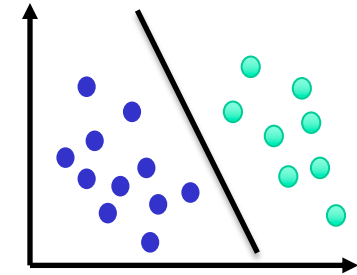
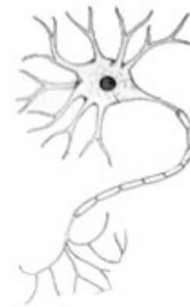
*The perceptron*

$$\Theta = \{W, b\}$$

*The neuron*



$\approx$

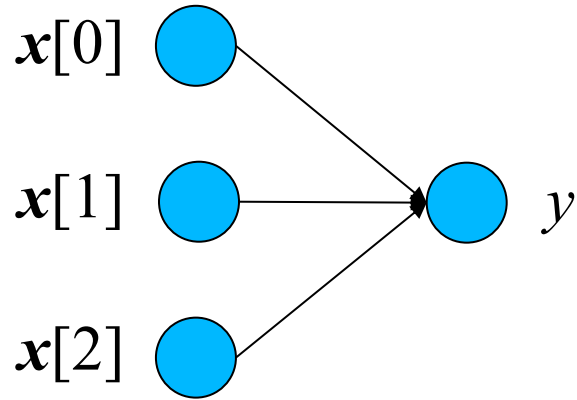


$$y = W \cdot x + b$$

Diagram illustrating the components of the perceptron equation  $y = W \cdot x + b$  with red arrows pointing to the terms:

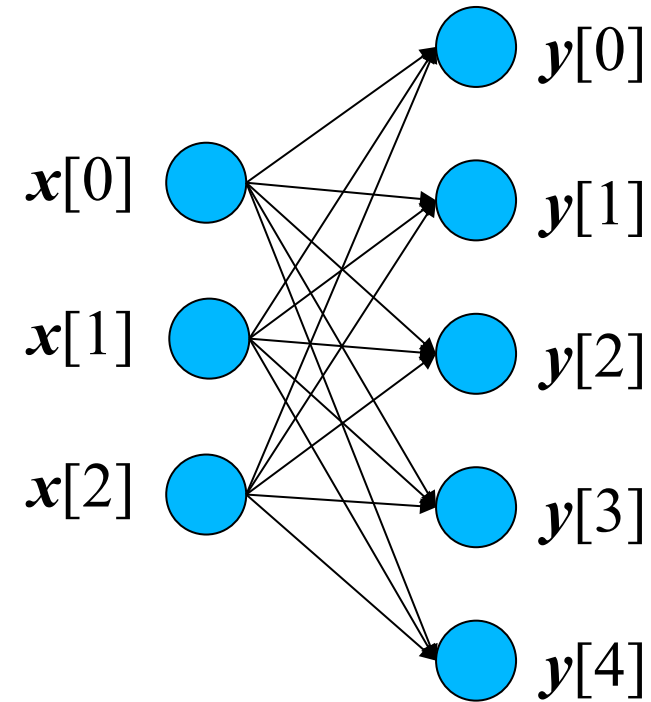
- input**: points to  $x$
- weight**: points to  $W$
- bias**: points to  $b$
- output**: points to  $y$

# Generalize to Fully-Connected Layer



Linear Classifier:

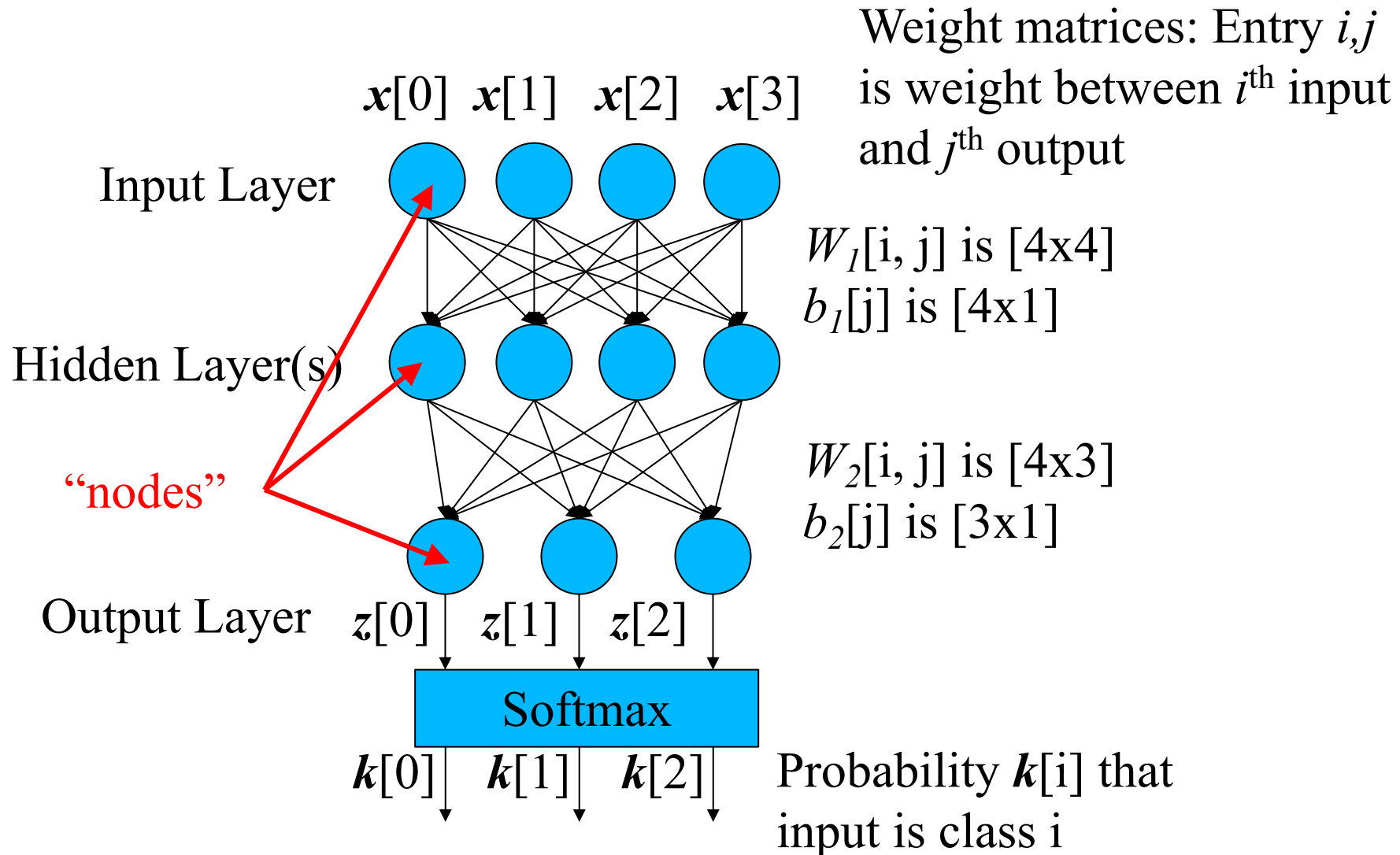
Input vector  $\mathbf{x}$   $\times$  weight vector  $\mathbf{w}$  plus bias scalar  $b$  to produce scalar output  $y$



Fully-connected:

Input vector  $\mathbf{x}$   $\times$  weight matrix  $\mathbf{w}$  plus bias vector  $\mathbf{b}$  to produce vector output  $\mathbf{y}$

# Multilayer Terminology



# Example: Digit Recognition

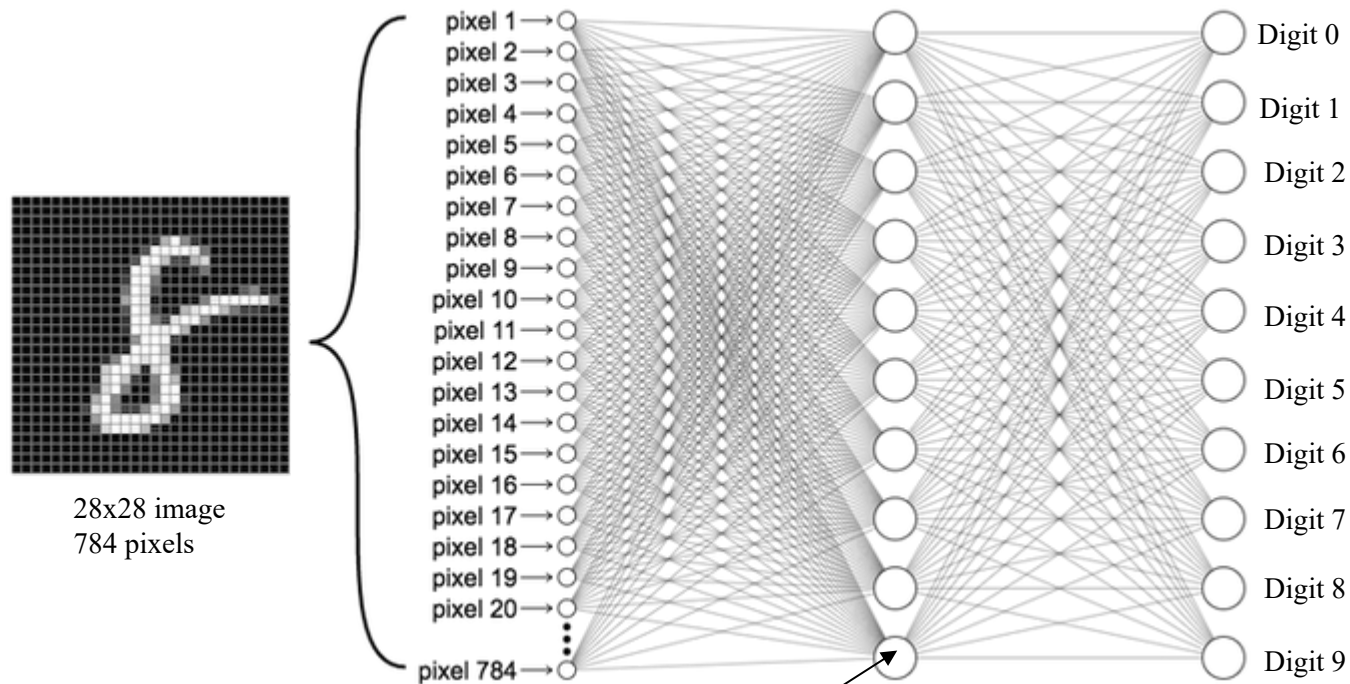
Handwritten digit recognition:

- given a  $28 \times 28$  grayscale image,
- produce a number from 0 to 9.

Input dataset

- 60,000 images
- Each labeled by a human with correct answer.

# MultiLayer Perceptron (MLP) for Digit Recognition



This network would have

- 784 nodes on input layer (L0)
- 10 nodes on hidden layer (L1)
- 10 nodes on output layer (L2)

784\*10 weights + 10 biases for L1

10\*10 weights + 10 biases for L2

A total of 7,960 parameters

Each node represents a function, based on a linear combination of inputs + bias

Activation function “repositions” output value.

Sigmoid, sign, ReLU are common... 7

# How Do We Determine the Weights?

**First layer** of perceptrons

- **784** ( $28^2$ ) inputs, **1024** outputs, **fully connected**
- **[1024 × 784]** weight matrix  **$W$**
- **[1024 x 1]** bias vector  **$b$**

Idea: given enough labeled input data, we can **approximate the input-output function**.



# Forward and Backward Propagation

Forward (**inference**):

- given input  $\mathbf{x}$  (for example, an image),
- **use parameters  $\Theta$**  ( $\mathbf{W}$  and  $\mathbf{b}$  for each layer)
- **to compute probabilities  $k[i]$**  (ex: for each digit  $i$ ).

Backward (**training**) [Rumelhart, Hinton, Williams 1986]:

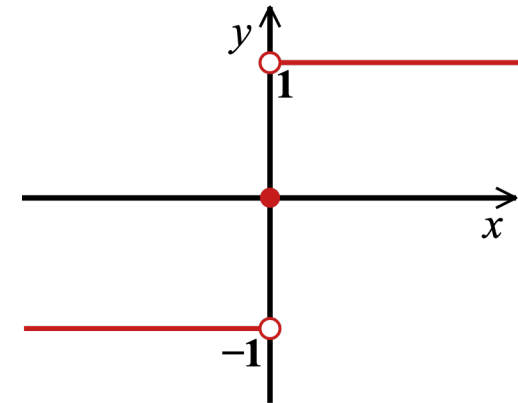
- given input  $\mathbf{x}$ , parameters  $\Theta$ , and outputs  $k[i]$ ,
- **compute error  $E$**  based on target label  $t$ ,
- then **adjust  $\Theta$**  proportional to  $E$  to reduce error.

# Neural Functions Impact Training

Recall perceptron function:  $y = \text{sign}(W \cdot x + b)$

**To propagate error backwards,**

- **use chain rule** from calculus.
- **Smooth functions are useful.**



Sign is not a smooth function, and has regions of 0 slope.

# One Choice: Sigmoid/Logistic Function

Until about 2017,

- **sigmoid / logistic function** most popular

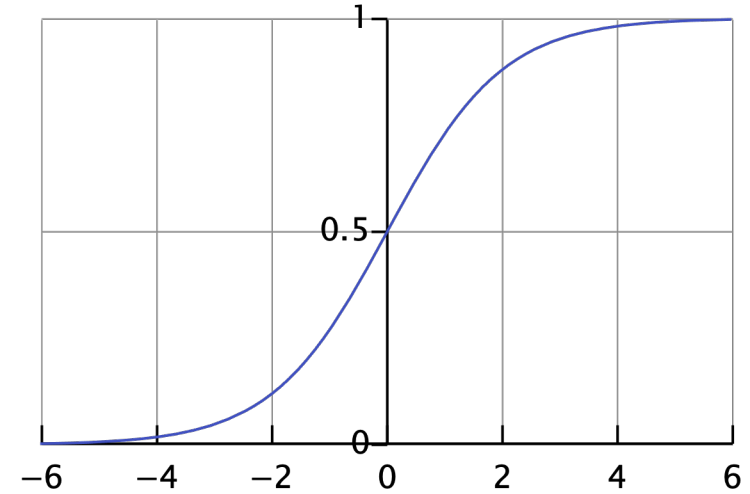
$$f(x) = \frac{1}{1+e^{-x}} \quad (f: \mathbb{R} \rightarrow (0,1) )$$

for replacing sign.

- Once we have  $f(x)$ , finding  $df/dx$  is easy:

$$\frac{df(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = f(x) \frac{e^{-x}}{(1+e^{-x})} = f(x)(1-f(x))$$

(Our example used this function.)



# Today's Choice: ReLU

In 2017, most common choice became

- **rectified linear unit / ReLU / ramp function**

$$f(x) = \max(0, x) \quad (f: \mathbb{R} \rightarrow \mathbb{R}^+)$$

which is much faster (no exponent required).

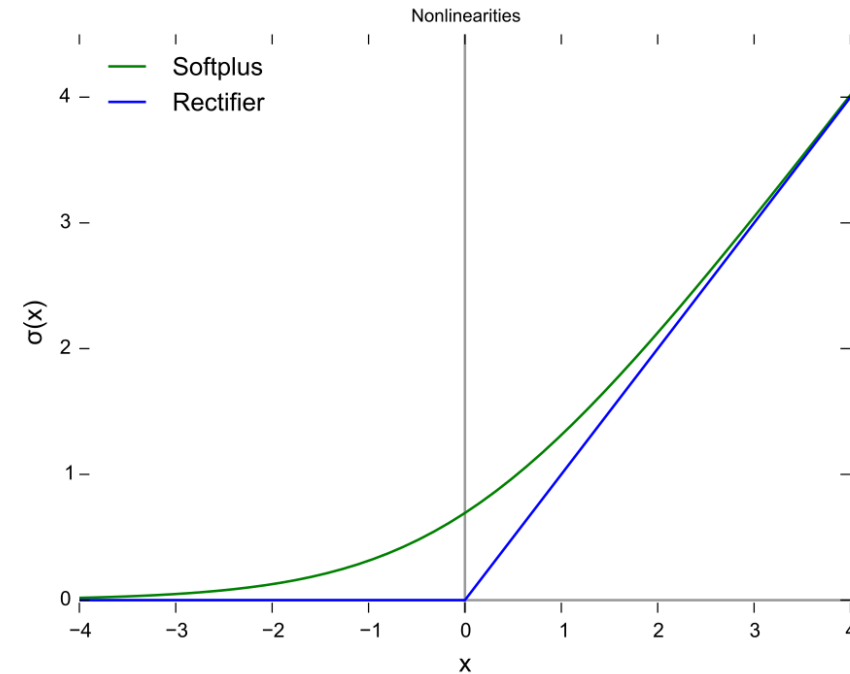
- A smooth approximation is

**softplus/SmoothReLU**

$$f(x) = \ln(1 + e^x) \quad (f: \mathbb{R} \rightarrow \mathbb{R}^+)$$

which is the integral of the logistic function.

- Lots of variations exist. See Wikipedia for an overview and discussion of tradeoffs.



# Use Softmax to Produce Probabilities

**How can sigmoid / ReLU produce probabilities?**

They can't.

- Instead, given output vector  $\mathbf{Z} = (z[0], \dots, z[C-1])^*$ ,
- we produce a second vector  $\mathbf{K} = (k[0], \dots, k[C-1])$
- using the **softmax function**

$$k[i] = \frac{e^{z[i]}}{\sum_{j=0}^{C-1} e^{z[j]}}$$

Notice that **the  $k[i]$  sum to 1.**

\*Remember that we classify into one of  $C$  categories.

# Choosing an Error Function

Many error functions are possible.

For example, **given label  $T$**  (digit  $T$ ),

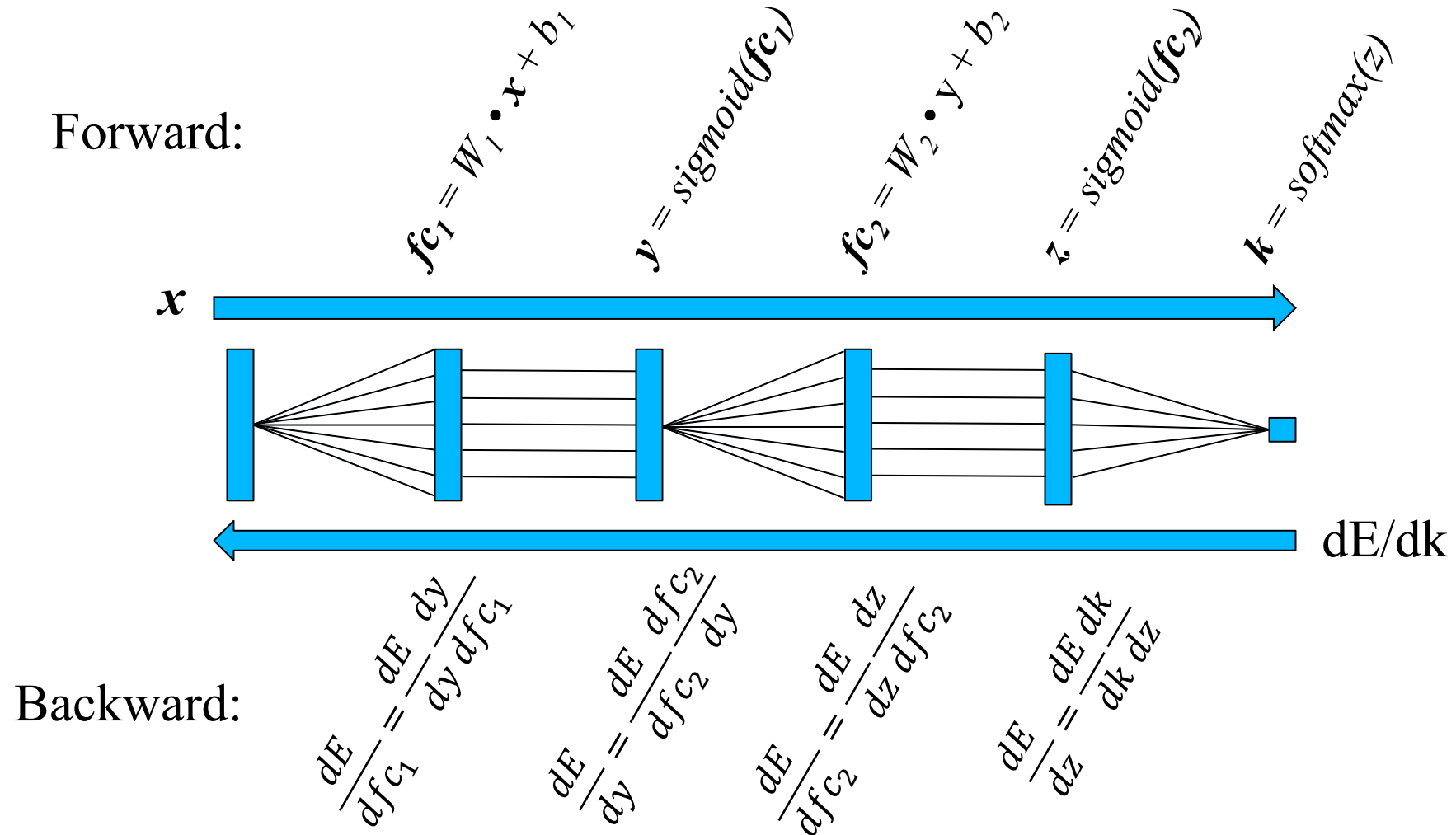
- $E = 1 - k[T]$ , the **probability of not classifying as  $t$** .

**Alternatively**, since our categories are numeric,  
we can **penalize quadratically**:

$$E = \sum_{j=0}^{C-1} k[j](j - T)^2$$

Let's **go with the latter**.

# Forward and Backward Propagation



# Example: Gradient Update with One Layer

$$\Theta_{i+1} = \Theta_i - \varepsilon \Delta \Theta \quad W_{i+1} = W_i - \varepsilon \Delta W$$

Parameter Update

$$y = W \cdot \mathbf{x} + b$$

Network function

$$\frac{dy}{dW} = x$$

Network weight gradient

$$E = \frac{1}{2} (y - t)^2$$

Error function

$$\frac{dE}{dy} = y - t = Wx + b - t$$

Error function gradient

$$\Delta W = \frac{dE}{dW} = \frac{dE}{dy} \frac{dy}{dW}$$

Full weight update expression

$$W_{i+1} = W_i - \varepsilon (W\mathbf{x} + b - t)x$$

Full weight update term



# Fully-Connected Gradient Detail

$$\begin{array}{c}
 fc_1[0] \\
 fc_1[1] \\
 fc_1[2] \\
 \dots
 \end{array}
 \begin{array}{|c|}
 \hline
 \\
 \hline
 \end{array}
 =
 \begin{array}{c}
 W_1[0,:] \\
 W_1[1,:] \\
 W_1[2,:] \\
 \dots
 \end{array}
 \begin{array}{|c|}
 \hline
 \\
 \hline
 \end{array}
 \begin{array}{c}
 x_1[0] \\
 x_1[1] \\
 x_1[2] \\
 x_1[3] \\
 \dots
 \end{array}$$

$$fc_1 = \sum_j W_1[i,j] x_1[j]$$

---


$$\frac{dE}{dW_1[i,j]} = \frac{dE}{dfc_1[i]} \frac{dfc_1[i]}{dW_1[i,j]} = \frac{dE}{dfc_1[i]} x_1[j]$$

Computed from  
previous layer

Need input to this layer

# Batch Gradient Descent

## How do we calculate the weights?

For ALL inputs in Batch of  $X$ ,

1. evaluate network to compute  $k[i]$  (forward),
2. then use  $k[i]$  and label  $T$  (target digit) to compute error  $E$ .
3. Backpropagate error derivative to find derivatives for each parameter.

Adjust  $\Theta$  to reduce total  $E$ :  $\Theta_{i+1} = \Theta_i - \varepsilon \Delta \Theta$   
(The  $\varepsilon$  is called the learning rate)

$X$  is the entire training set of data. We divide  $X$  into randomly sampled partitions called Batches.

# Stochastic Gradient Descent

For a random input in Batch of  $X$ ,

1. evaluate network to compute  $k[i]$  (forward),
2. then use  $k[i]$  and label  $T$  (target digit) to compute error  $E$ .
3. Backpropagate error derivative to find derivatives for each parameter.

Adjust  $\Theta$  to reduce total  $E$ :  $\Theta_{i+1} = \Theta_i - \varepsilon \Delta \Theta$

Far less computation than BGD

# Mini-Batch Gradient Descent

For a set of random input in Batch of  $X$ ,

1. evaluate network to compute  $k[i]$  (forward),
2. then use  $k[i]$  and label  $T$  (target digit) to compute error  $E$ .
3. Backpropagate error derivative to find derivatives for each parameter.

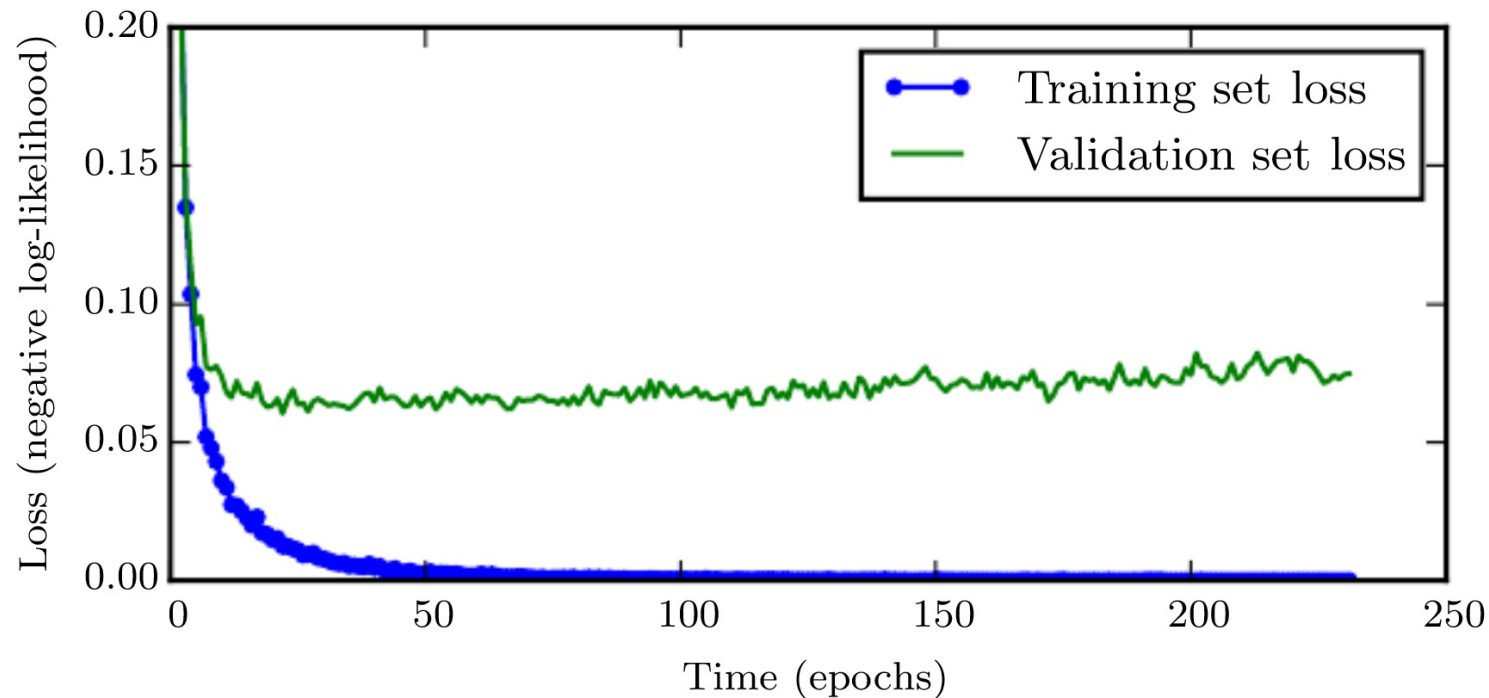
Adjust  $\Theta$  to reduce total  $E$ :  $\Theta_{i+1} = \Theta_i - \varepsilon \Delta \Theta$

Balance between accuracy of gradient estimation (BGD) and speed (SGD)

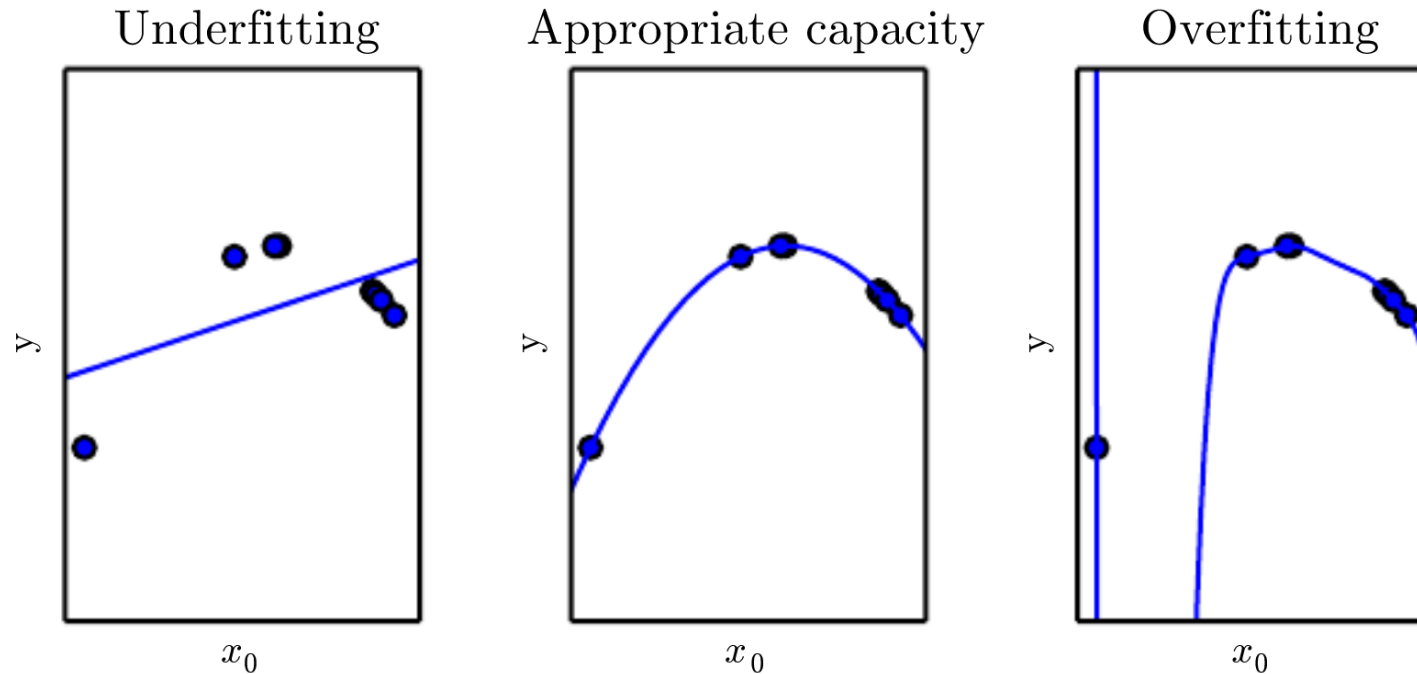
# When is Training Done?

Split labeled data into *training* and *test* sets.

- Training set is used to compute parameter updates.
- Testing set checks how well model generalizes to new inputs
- The network can become *too good* at classifying training inputs!

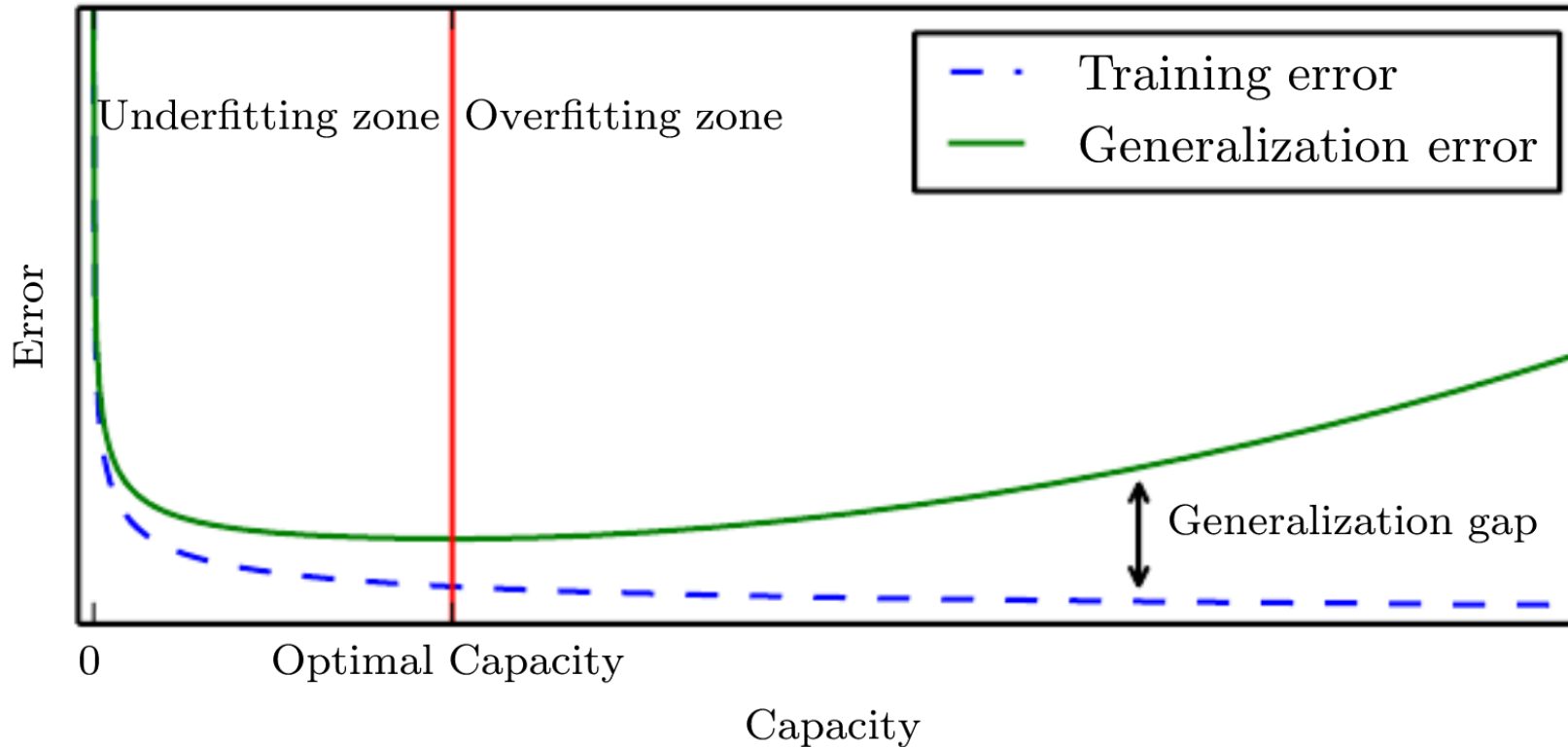


# How Complicated Should a Network Be?



Intuition: like a polynomial fit. High-order terms improve fit, but add unpredictable swings for inputs outside the training set.

# Overtraining Decreases Accuracy



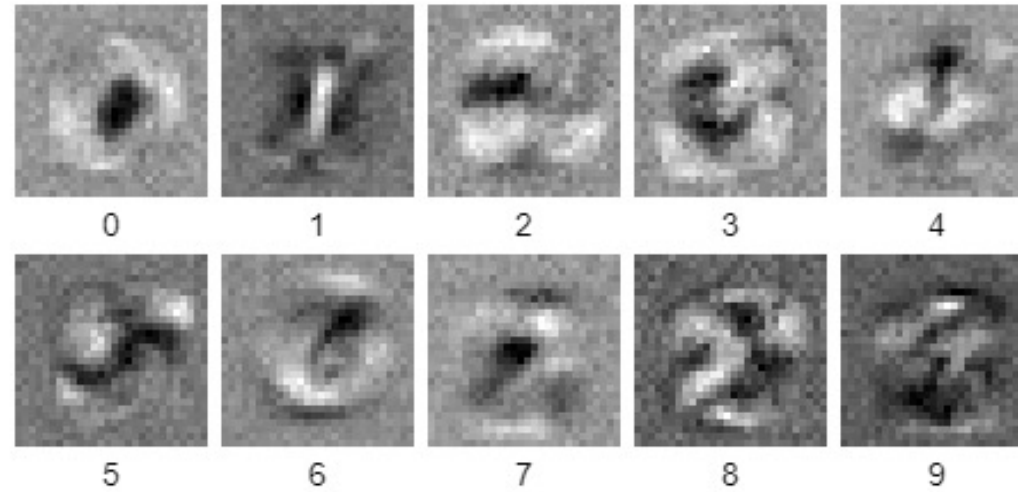
If network works too well for training data,  
new inputs cause big unpredictable output changes.

# No Free Lunch Theorem

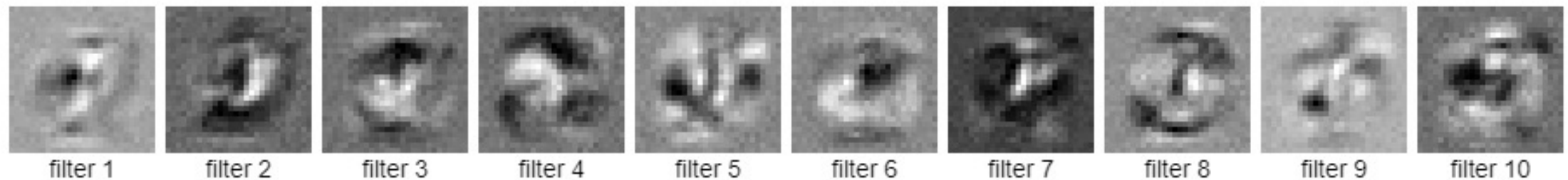
- Every classification algorithm has the same error rate when classifying previously unobserved inputs when averaged over all possible input-generating distributions.
- Neural networks must be tuned for specific tasks



# Visualizing Neural Network Weights



MNIST 1<sup>st</sup> layer



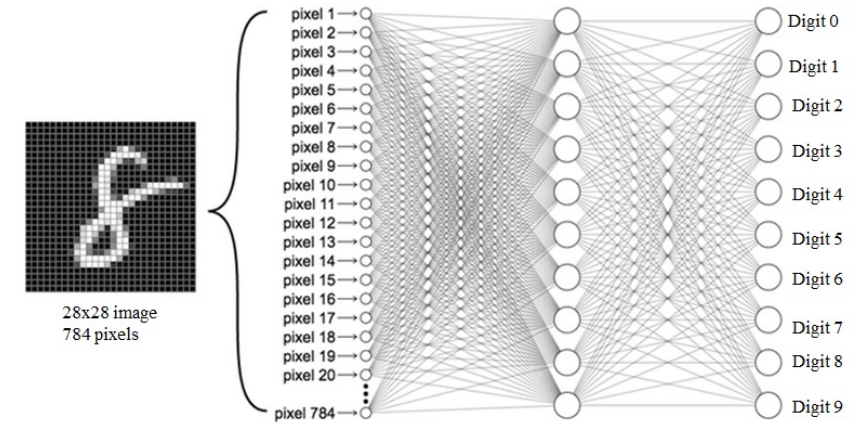
MNIST 2<sup>nd</sup> layer

From [https://ml4a.github.io/ml4a/looking\\_inside\\_neural\\_nets/](https://ml4a.github.io/ml4a/looking_inside_neural_nets/)

# Multi-Layer Perceptron (MLP) for an Image

Consider a 250 x 250 image...

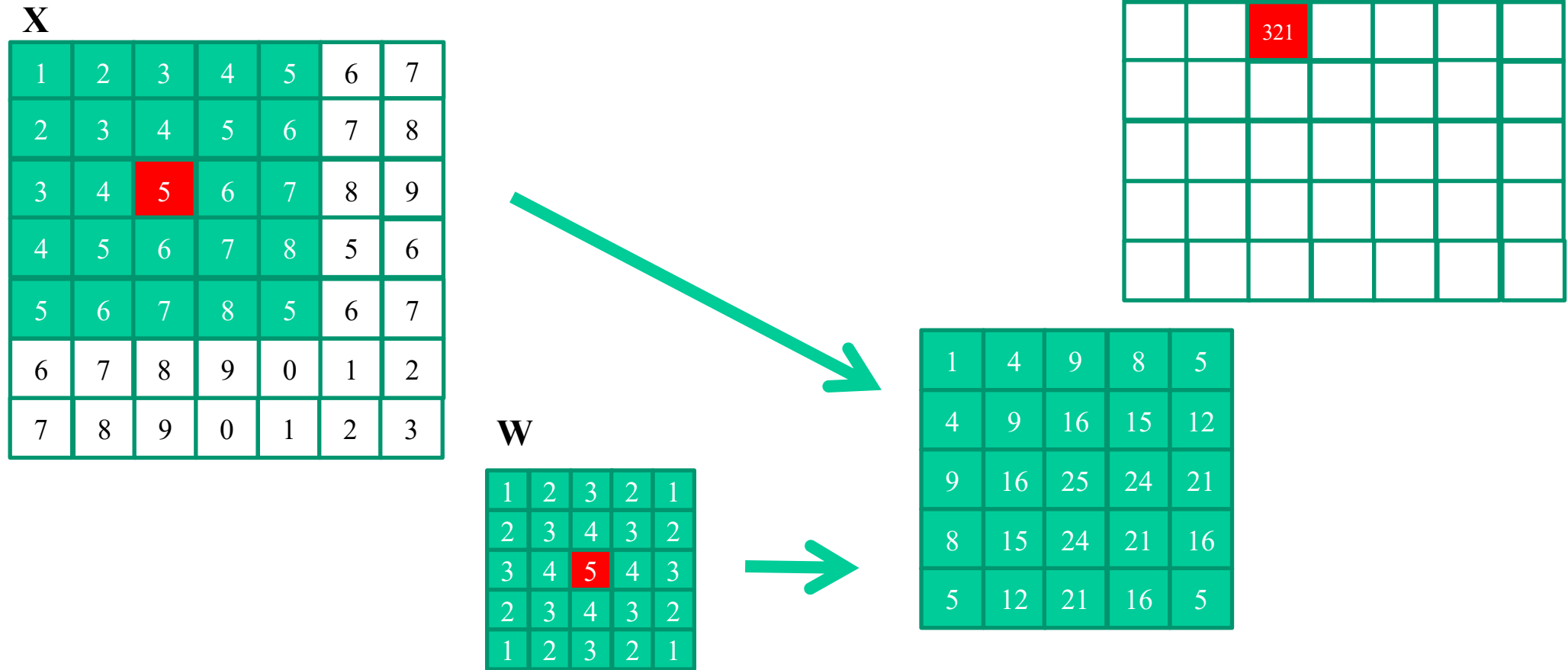
- input: 2D image treated as 1D vector
- Fully connected layer is huge:
  - 62,500 ( $250^2$ ) weights per node!
  - Comparable number of nodes gives ~4B weights total!
- Need >1 hidden layer? Bigger images?
- Too much computation, and too much memory.



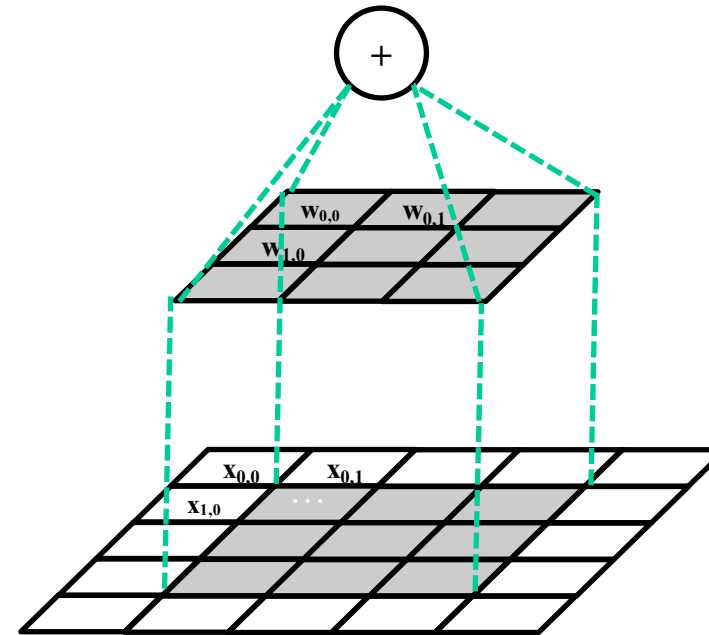
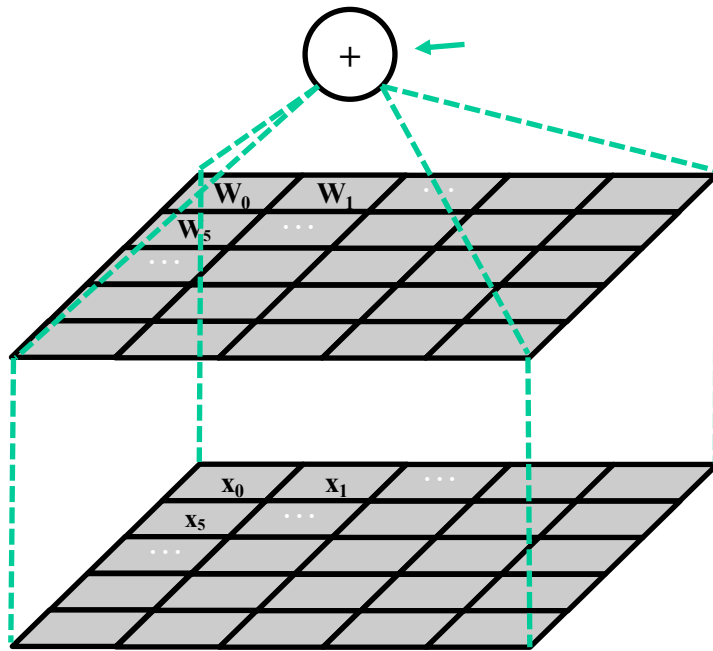
Traditional feature detection in image processing uses

- Filters → Convolution kernels
- Can we use them in neural networks?

# 2-D Convolution



# Convolution vs Fully-Connected (Weight Sharing)



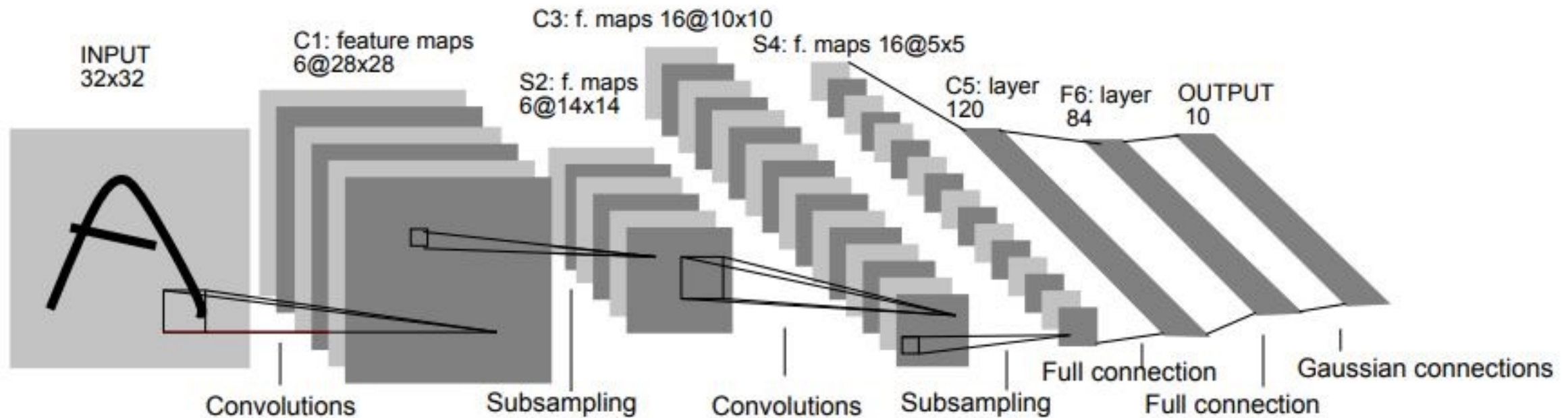
# Convolution Naturally Supports Varying Input Sizes

- As discussed so far,
  - perceptron layers have fixed structure, so
  - number of inputs / outputs is fixed.
- Convolution enables variably-sized inputs (observations of the same kind of thing)
  - Audio recording of different lengths
  - Image with more/fewer pixels

# Example Convolution Inputs

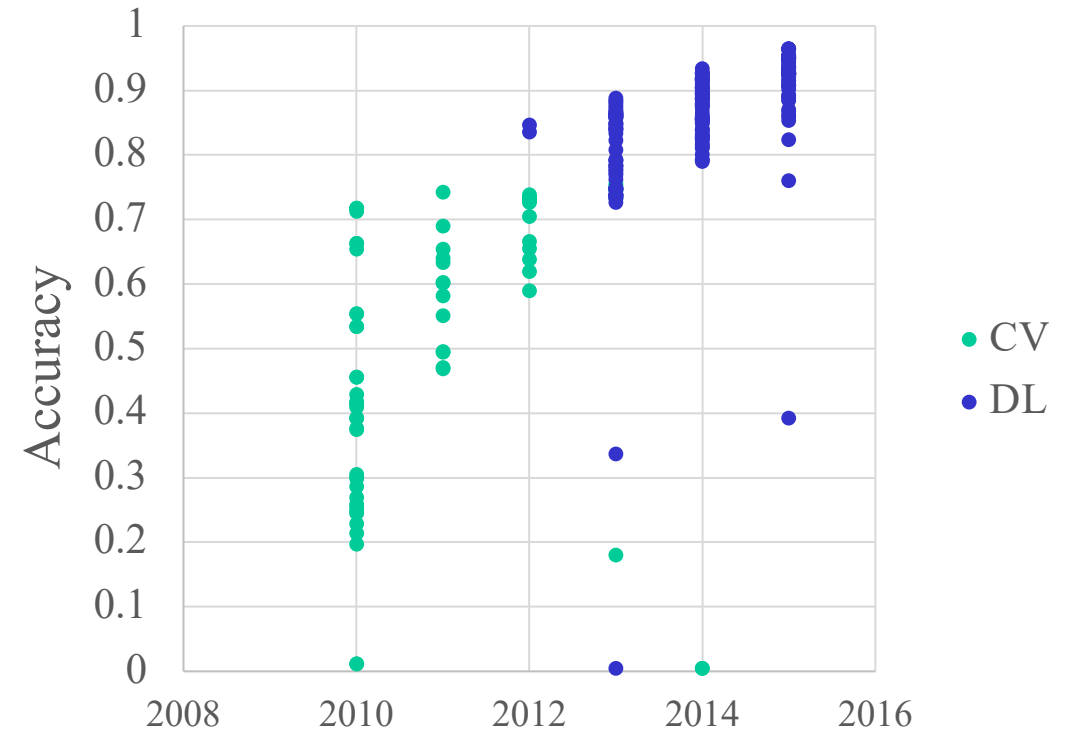
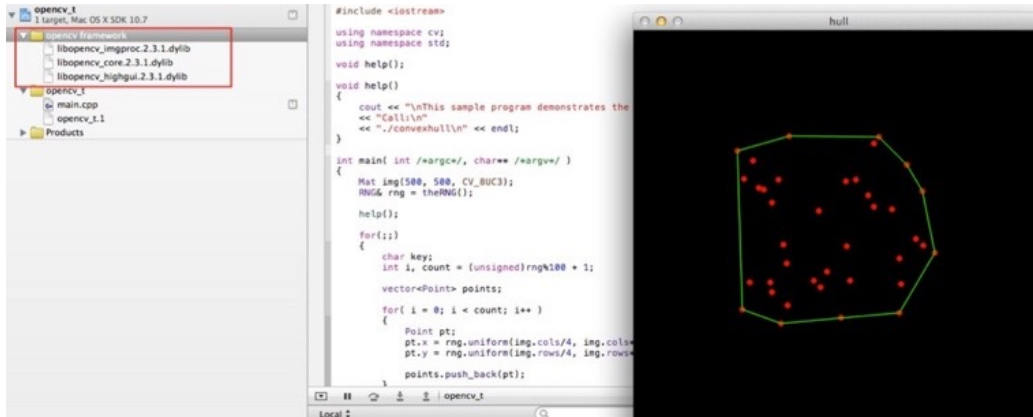
	Single-channel	Multi-channel
1D	audio waveform	Skeleton animation data: 1-D joint angles for each joint
2D	Fourier-transformed audio data Convolve over frequency axis: invariant to frequency shifts Convolve over time axis: invariant to shifts in time	Color image data: 2D data for R,G,B channels
3D	Volumetric data (example: medical imaging)	Color video: 2D data across 1D time for R,G,B channels

# LeNet-5: CNN for hand-written digit recognition



# Deep Learning Impact in Computer Vision

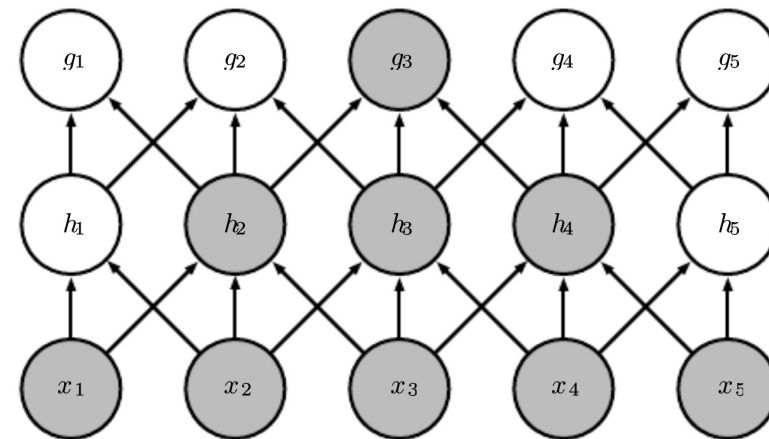
The Toronto team used GPUs and trained on 1.2M images in their 2012 winning entry at the Large Scale Visual Recognition Challenge





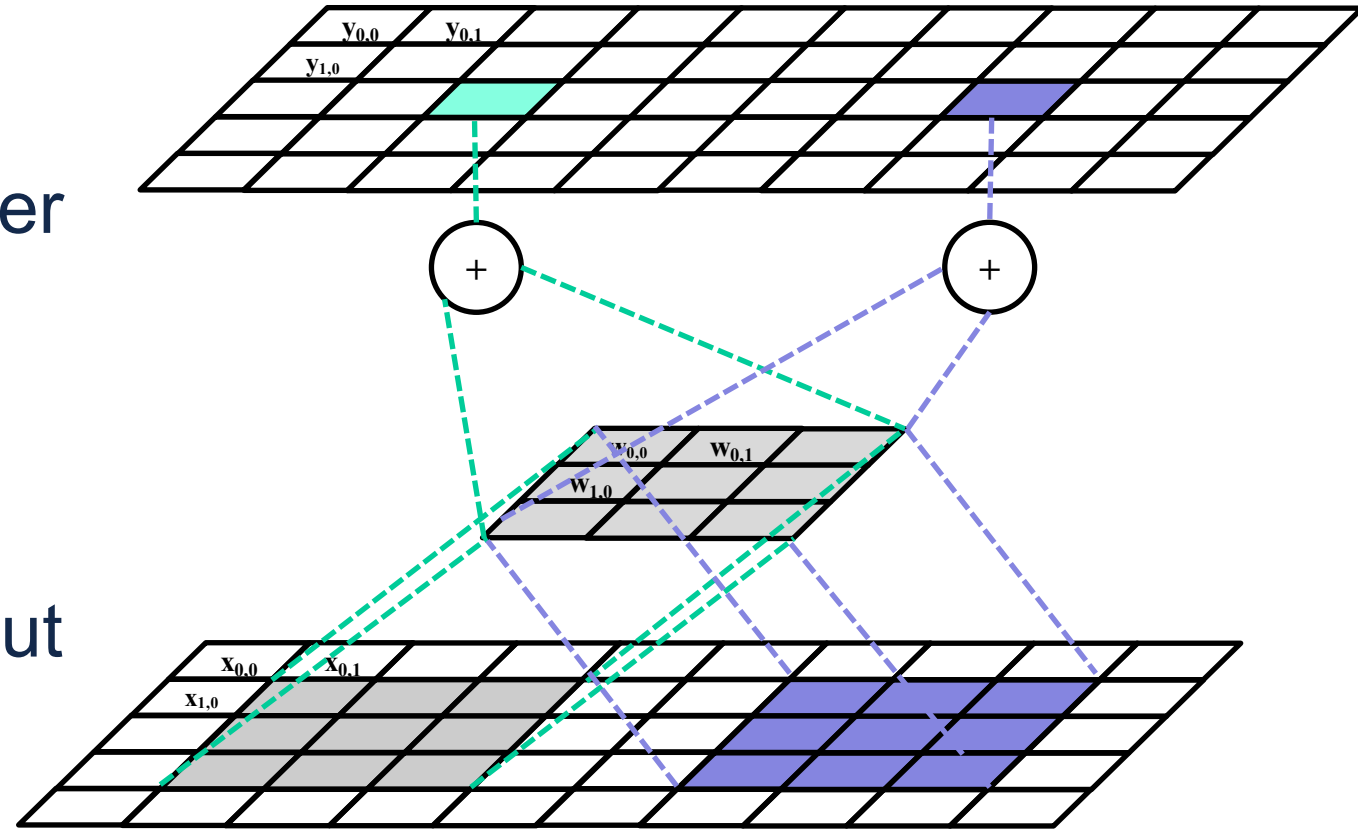
# Why Convolution

- Sparse interactions
  - Meaningful features in small spatial regions
  - Need fewer parameters (less storage, better statistical characteristics, faster training)
  - Need multiple layers for wide receptive field



# Why Convolution

- Parameter sharing
  - Kernel mask is applied repeatedly computing layer output
- Equivariant Representations
  - If input is translated, output is similarly translated
  - Output is a map of where features appear in input



# Convolution

- 2-D Matrix
- $Y = W \otimes X$
- Kernel smaller than input:  
smaller receptive field
- Fewer Weights

# MLP

- Vector
- $Y = w x + b$
- Maximum receptive  
field
- More weights

**ANY MORE QUESTIONS?**