ECE408 / CS483 / CSE 408

From Machine Learning to Deep Learning

Course Reminders

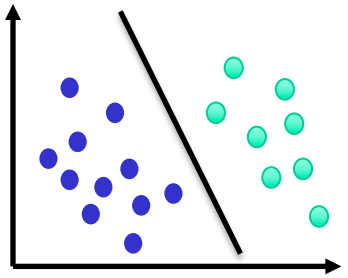
- Lab 4 is due on Friday
- Midterm 1 is on Tuesday, October 10th
- Project Milestone 1: Baseline CPU implementation is due Friday October 13th
  - Project details will be provided by end of this week

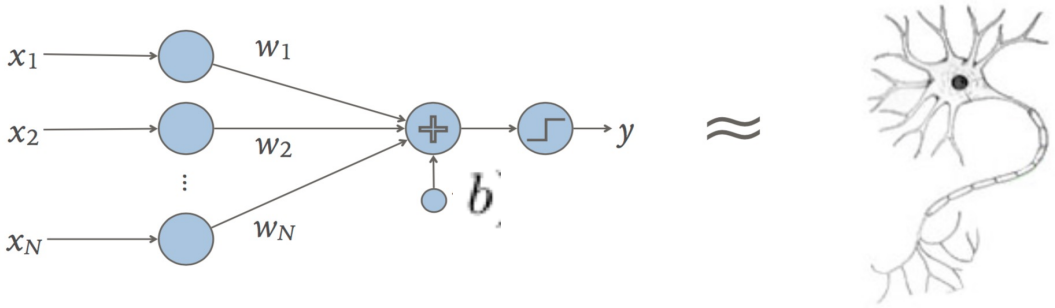
Perceptron is a Simple Example

- Example: a **perceptron**

$$y = \text{sign}(W \bullet x + b) \qquad \Theta = \{W, b\}$$

The perceptron *The neuron*





$$y = W \bullet \boldsymbol{x} + b$$

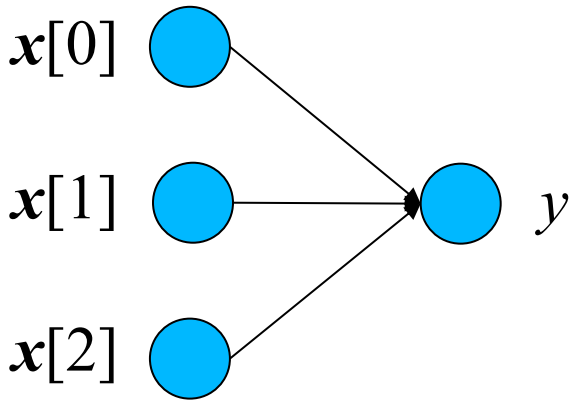
output

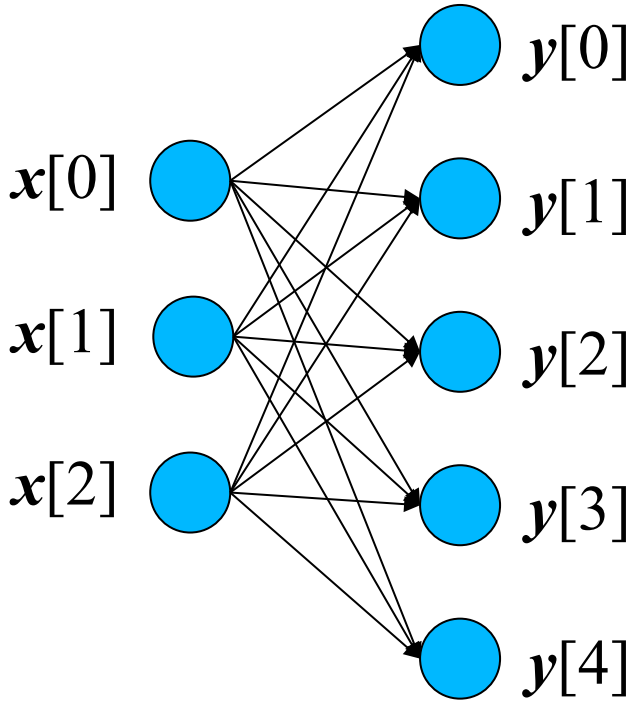
input

bias

weight

Generalize to Fully-Connected Layer

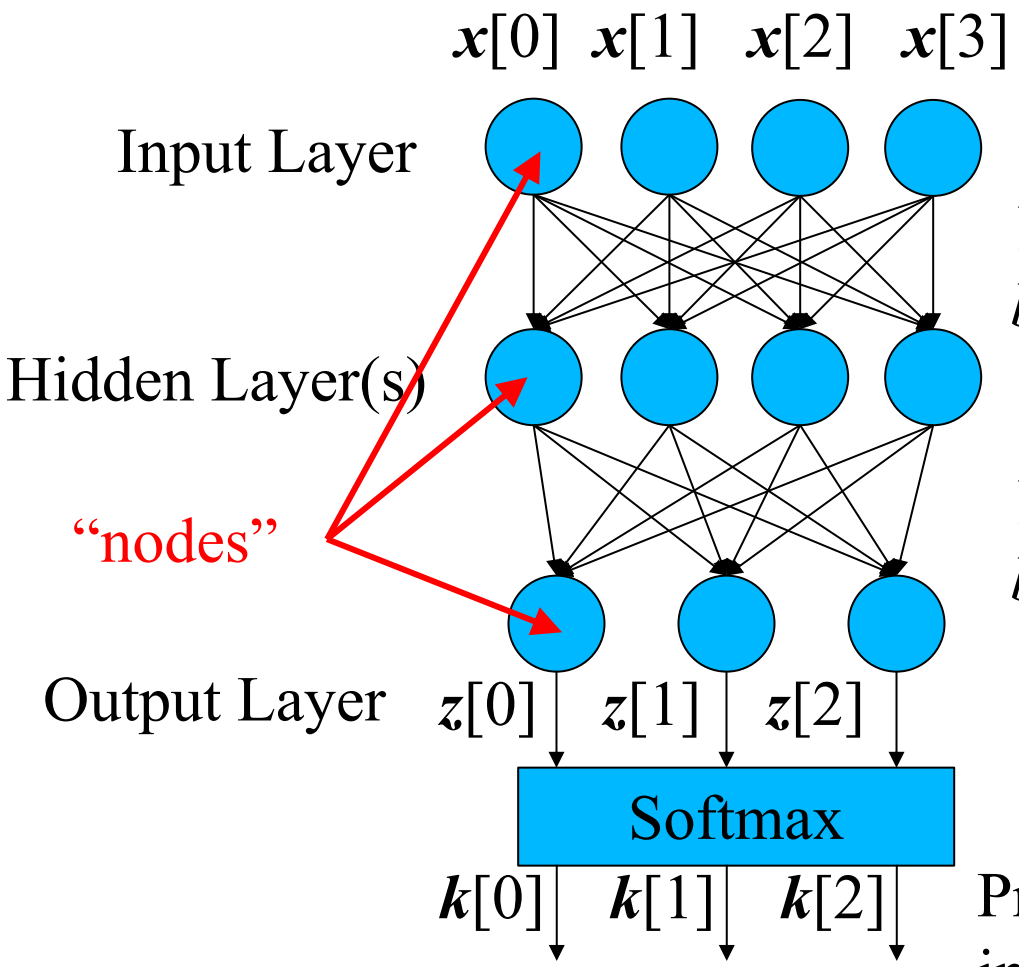




Linear Classifier:
Input vector $\boldsymbol{x}$ × weight vector $\boldsymbol{w}$ plus bias scalar b to produce scalar output y

Fully-connected:
Input vector $\boldsymbol{x}$ × weight matrix $\boldsymbol{w}$ plus bias vector $\boldsymbol{b}$ to produce vector output $\boldsymbol{y}$

Multilayer Terminology



Weight matrices: Entry *i,j* is weight between i^{th} input and j^{th} output

W_1[i, j] is [4x4]
b_1[j] is [4x1]

W_2[i, j] is [4x3]
b_2[j] is [3x1]

Probability ***k***[i] that input is class i

Example: Digit Recognition

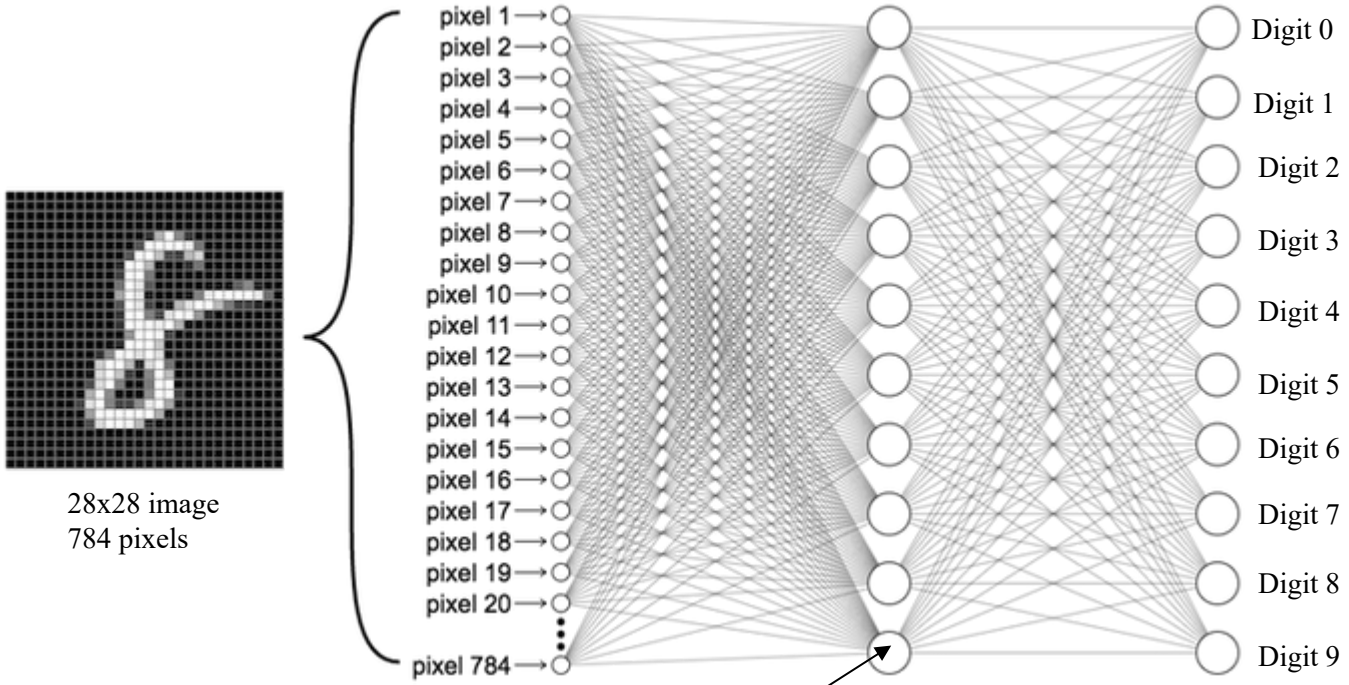
Handwritten digit recognition:

- given a 28×28 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



28x28 image
784 pixels

$$n_k^1 = \text{activation}(\boldsymbol{w}_k \boldsymbol{x} + b_k)$$

This network would has
- 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…

How Do We Determine the Weights?

First layer of perceptrons

- **784** (28^2) inputs, **10** outputs, **fully connected**
- **[784 x 10]** weight matrix ***W***
- **[10 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 ***x*** (for example, an image),
- **use parameters *Θ*** (***W*** and *b* for each layer)
- **to compute probabilities *k[i]*** (ex: for each digit *i*).

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

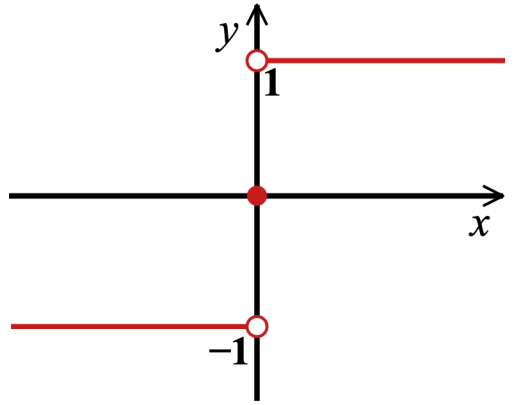
- given input ***x***, parameters ***Θ***, and outputs ***k[i]***,
- **compute error *E*** based on target label ***t***,
- then **adjust *Θ*** proportional to ***E*** to reduce error.

Neural Functions Impact Training

Recall perceptron function: **y = sign (W·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.

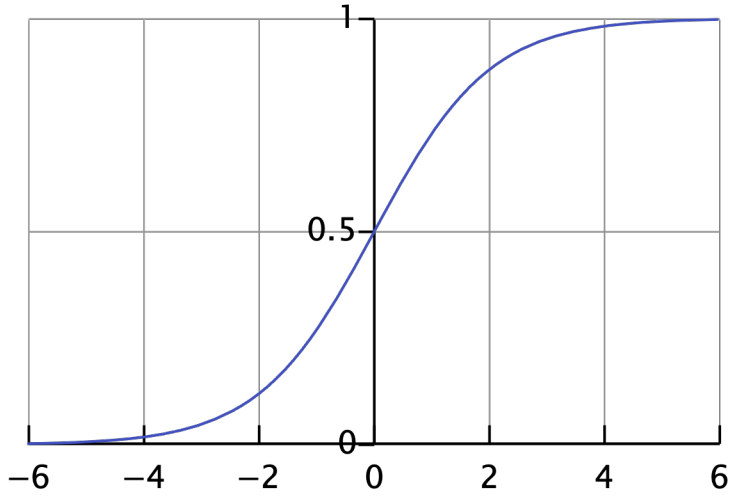
One Choice: Sigmoid/Logistic Function

Until about 2017,

- **sigmoid / logistic function** most popular

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

for replacing sign.



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

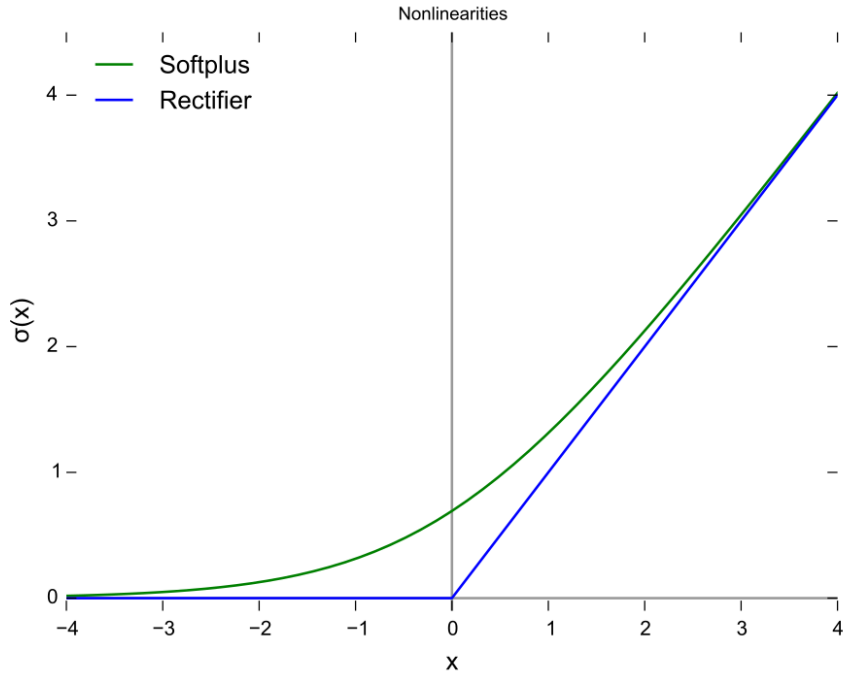
$$\boldsymbol{\frac{df(x)}{dx}} = \frac{e^{-x}}{(1+e^{-x})^2} = f(x)\frac{e^{-x}}{(1+e^{-x})} = \boldsymbol{f(x)(1-f(x))}$$

(Our example used this function.)

Today's Choice: ReLU

In 2017, most common choice became

- **rectified linear unit / ReLU / ramp function**
  $$\boldsymbol{f(x) = \max(0, x)} \quad (\text{f: } \mathbb{R} \rightarrow \mathbb{R}^+)$$
  which is much faster (no exponent required).
- A smooth approximation is **softplus/SmoothReLU**
  $$\boldsymbol{f(x) = \ln(1 + e^x)} \quad (\text{f: } \mathbb{R} \rightarrow \mathbb{R}^+)$$
  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 **Z = (z[0], …, z[C-1])***,
- we produce a second vector **K = (k[0], …, 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***),

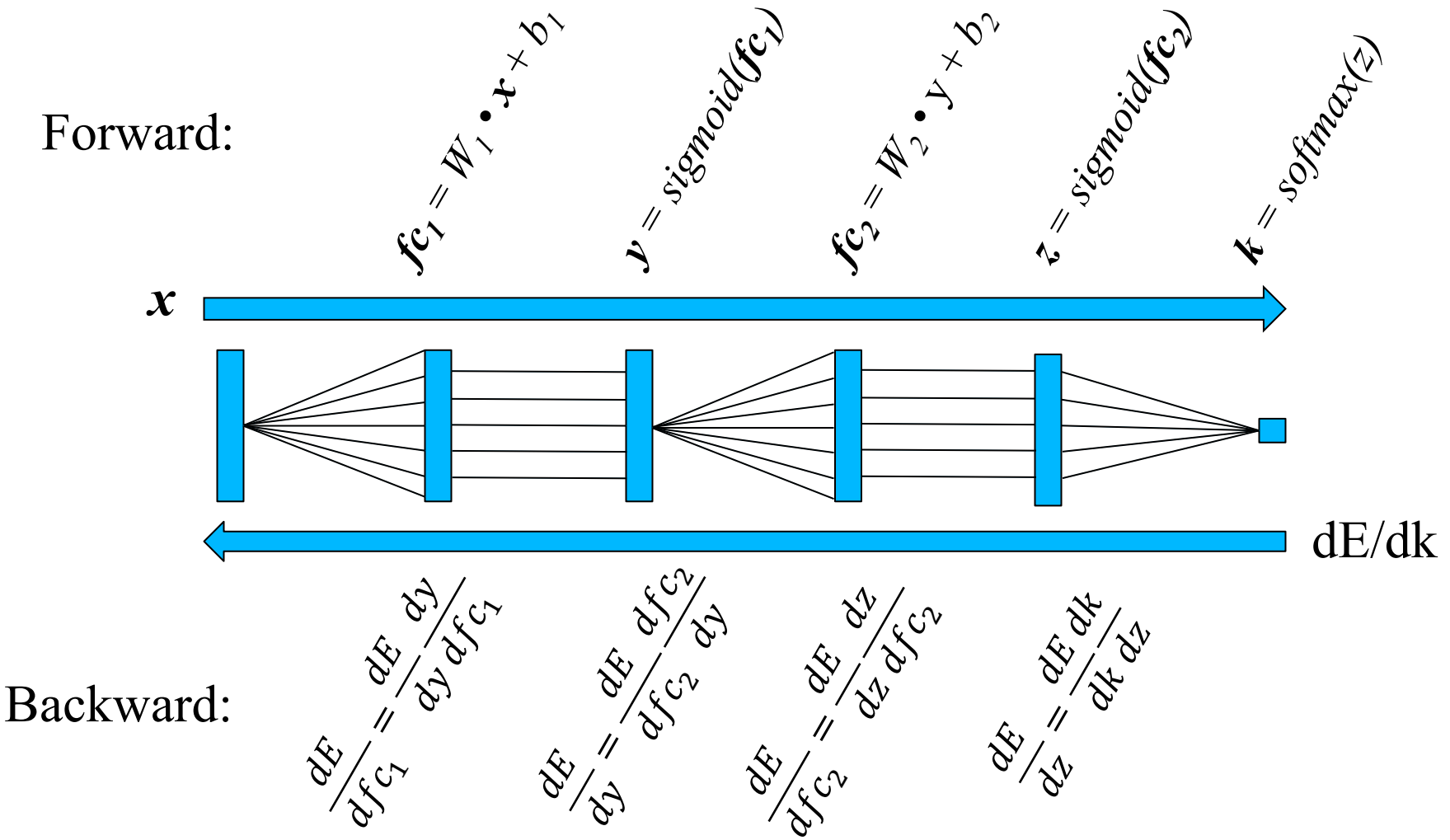
- ***E = 1 – k[T]***, the **probability of not classifying as *t***.

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

$y = W \bullet \boldsymbol{x} + b$ Network function

$\frac{dy}{dW} = x$ Network weight gradient

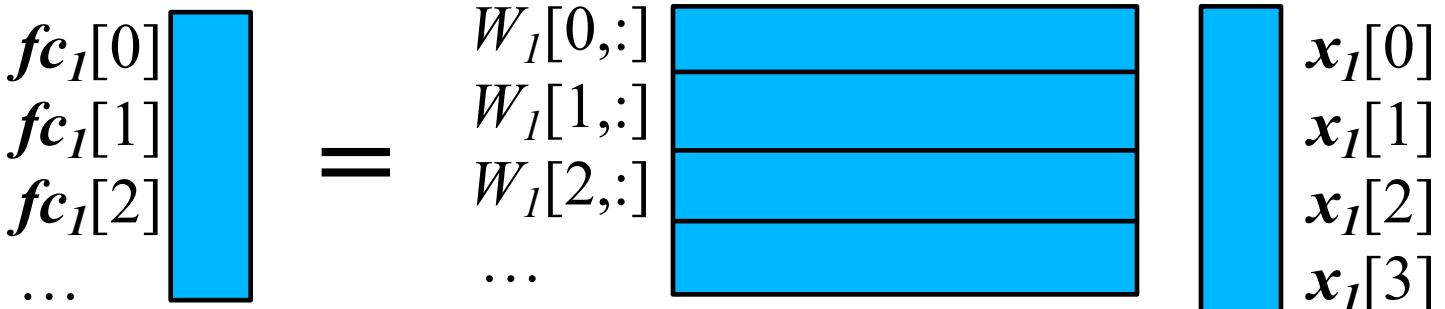
$E = \frac{1}{2} (y - t)^2$ Error function

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

$\Delta W = \frac{dE}{dW} = \frac{dE}{dy}\frac{dy}{dW}$ Full weight update expression

$W_{i+1} = W_i - \varepsilon(W\boldsymbol{x}+b-t)x$ Full weight update term

Fully-Connected Gradient Detail



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

Computed from previous layer

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

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

(The ε 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 Θ 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.

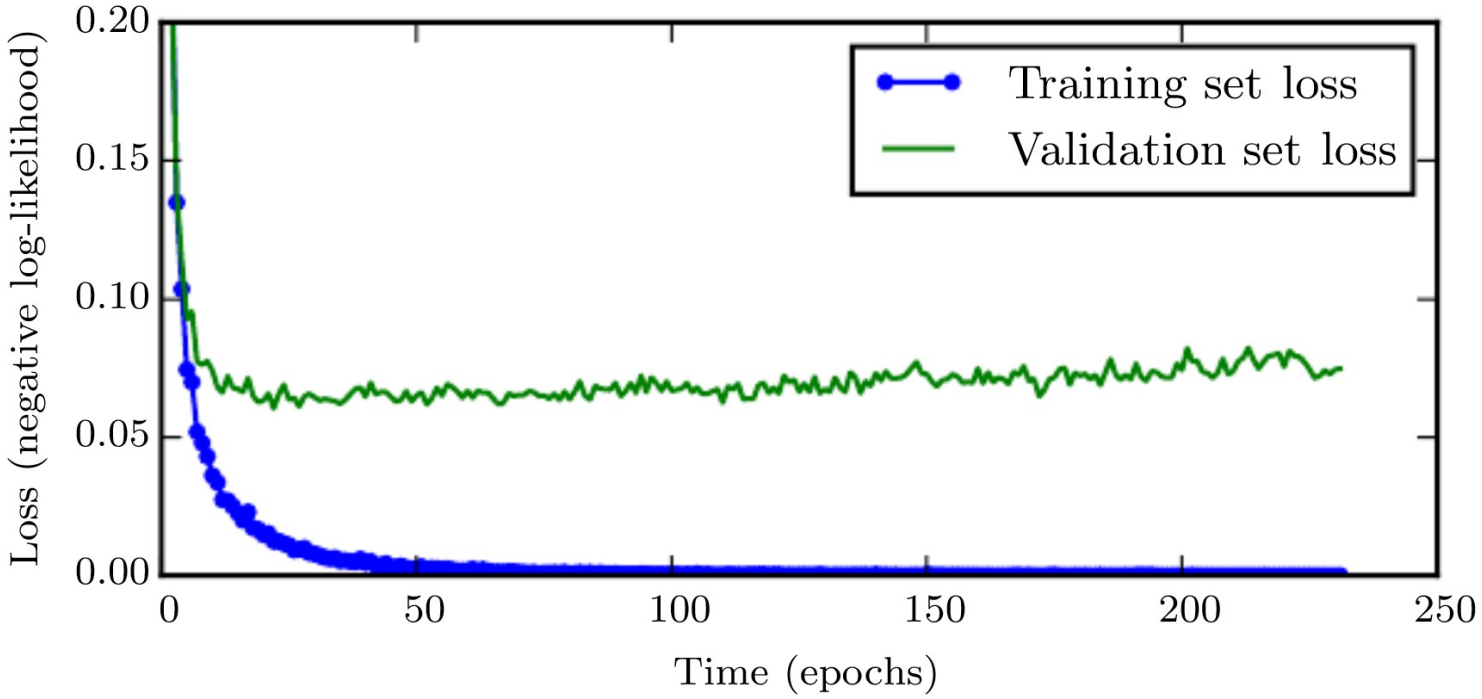
Adjust Θ to reduce total E: $\Theta_{i+1} = \Theta_i - \varepsilon \Delta\Theta$

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

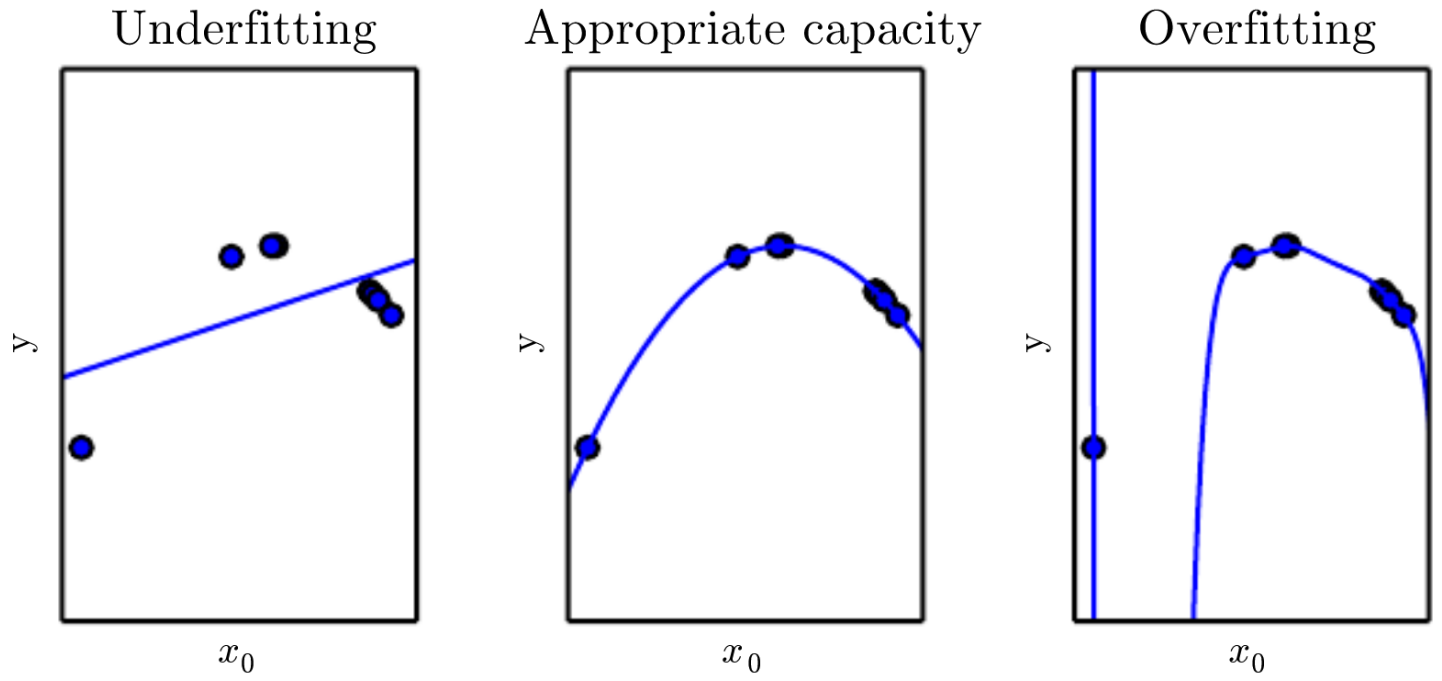
When is Training Done?

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!

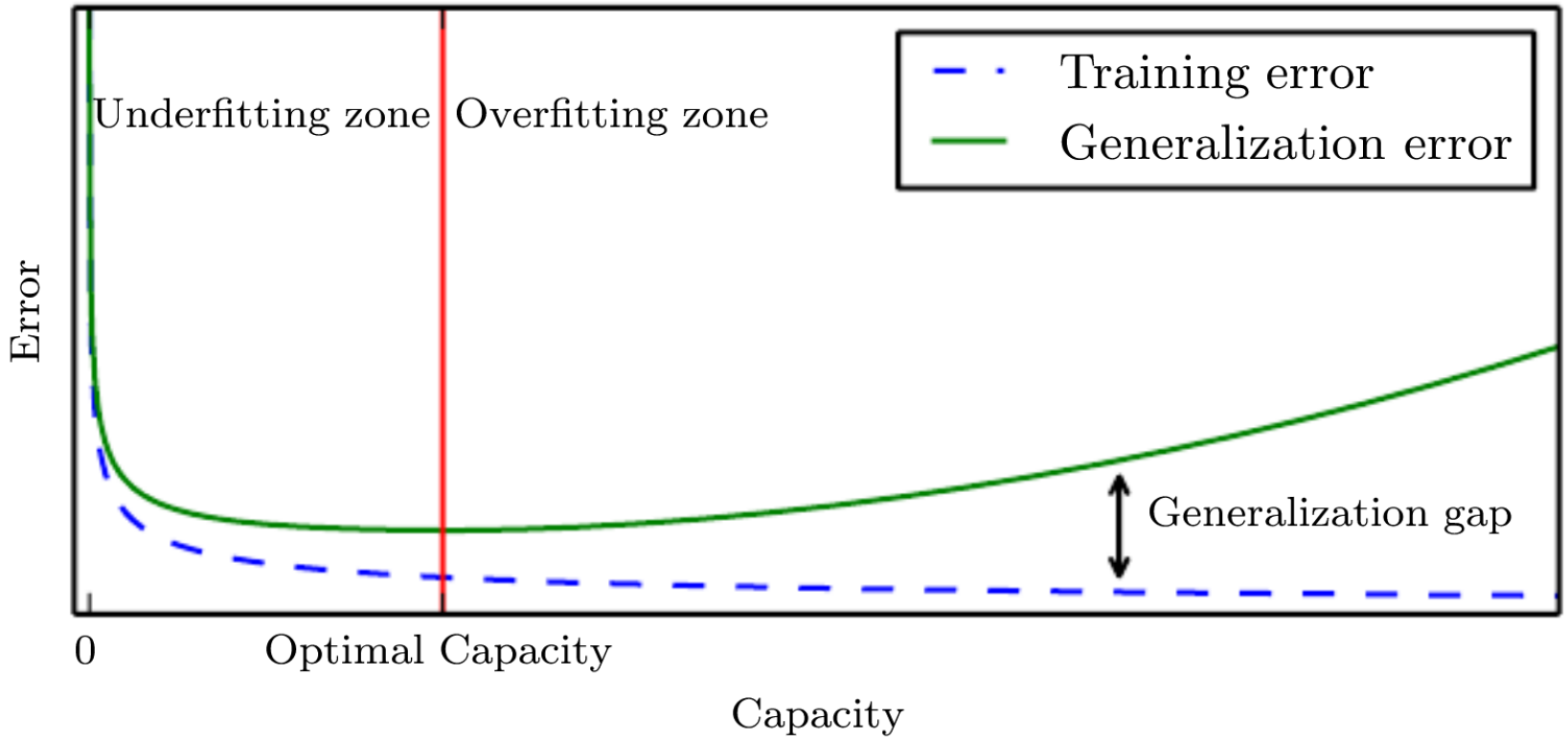


How Complicated Should a Network Be?



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

Overtraining Decreases Accuracy



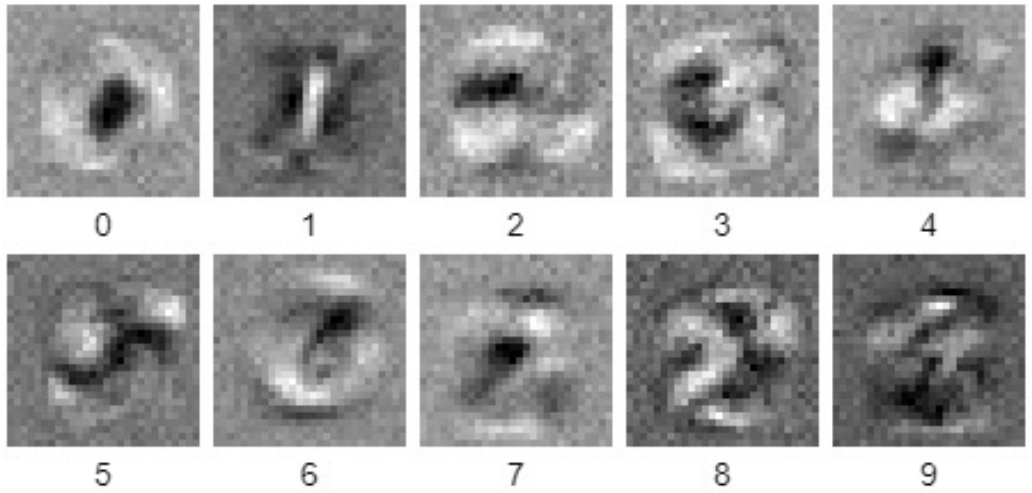
If network works too well for training data,
new inputs cause big unpredictable output changes.

No Free Lunch Theorem

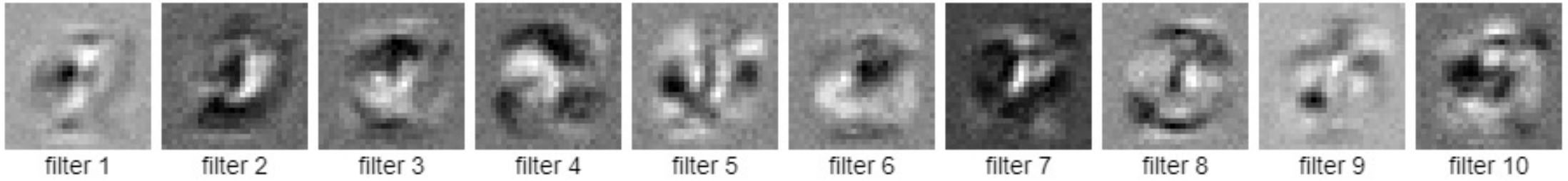
- Every classification algorithm has the same error rate when classifying previously unobserved inputs when averaged over all possible input-generating distributions.

- Neural networks must be tuned for specific tasks

Visualizing Neural Network Weights



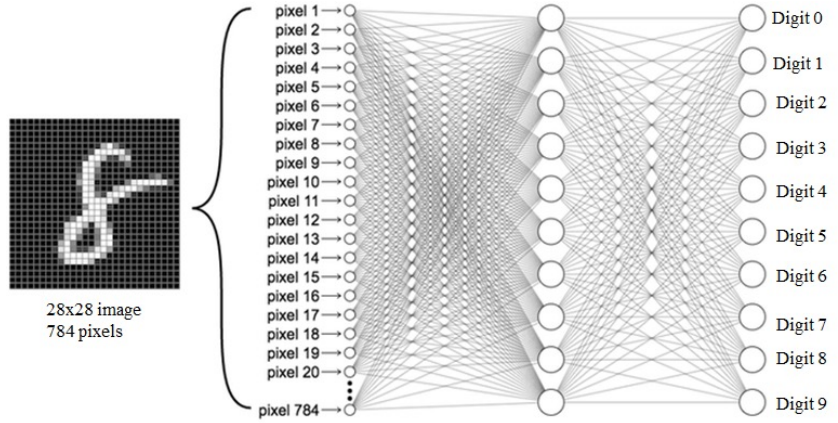
MNIST 1st layer



MNIST 2nd layer

From https://ml4a.github.io/ml4a/looking_inside_neural_nets/

Multi-Layer Perceptron (MLP) for an Image



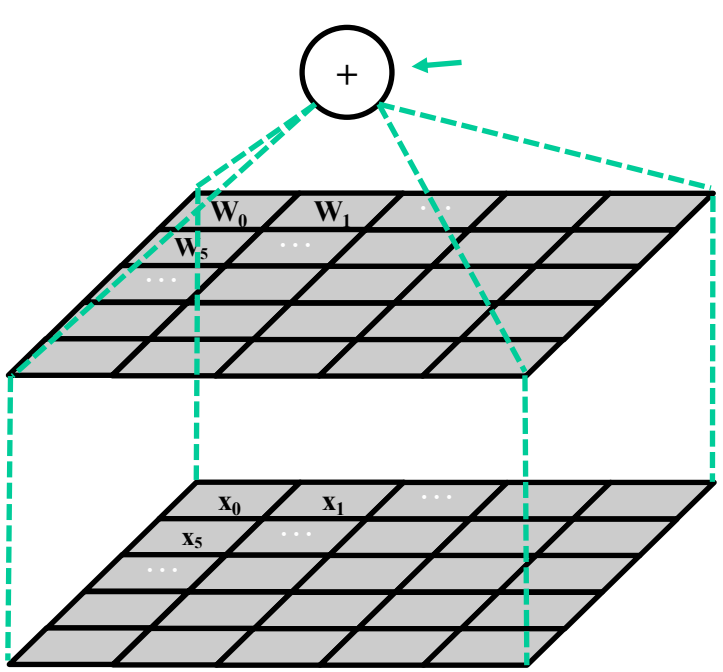
Consider a 250 x 250 image…

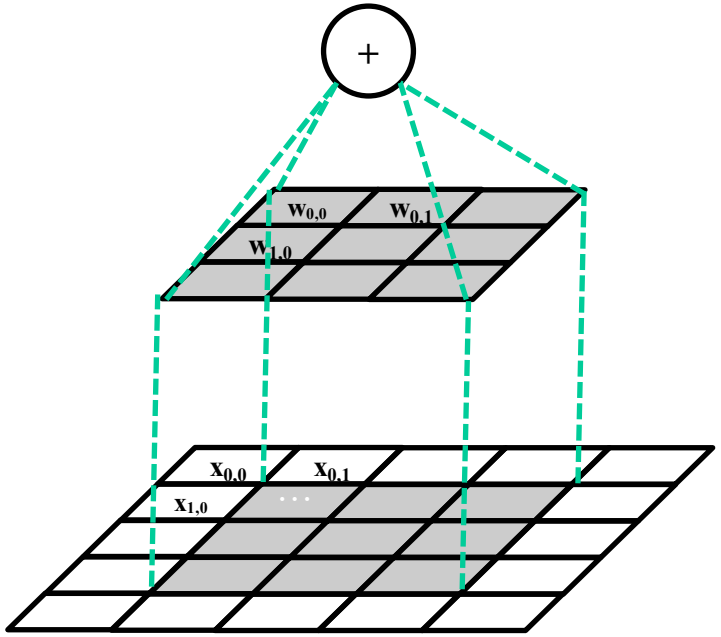
- input: 2D image treated as 1D vector
- Fully connected layer is huge:
  - 62,500 (250^2) weights per node!
  - Comparable number of nodes gives ~4B weights total!
- Need >1 hidden layer? Bigger images?
- Too much computation, and too much memory.

Traditional feature detection in image processing uses

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

Convolution vs Fully-Connected (Weight Sharing)

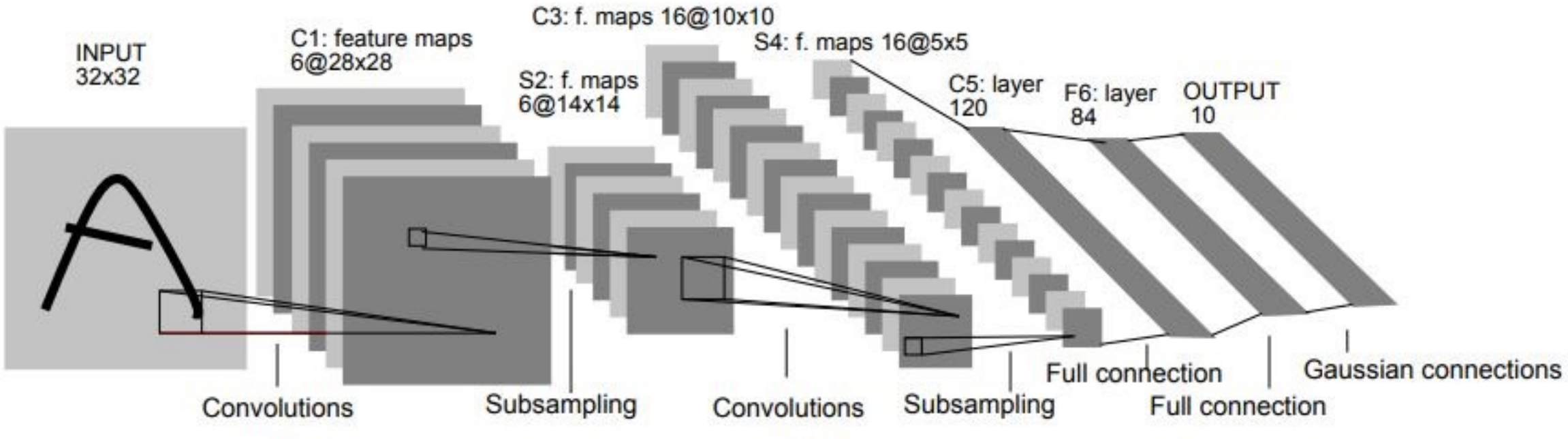




Why Convolution?

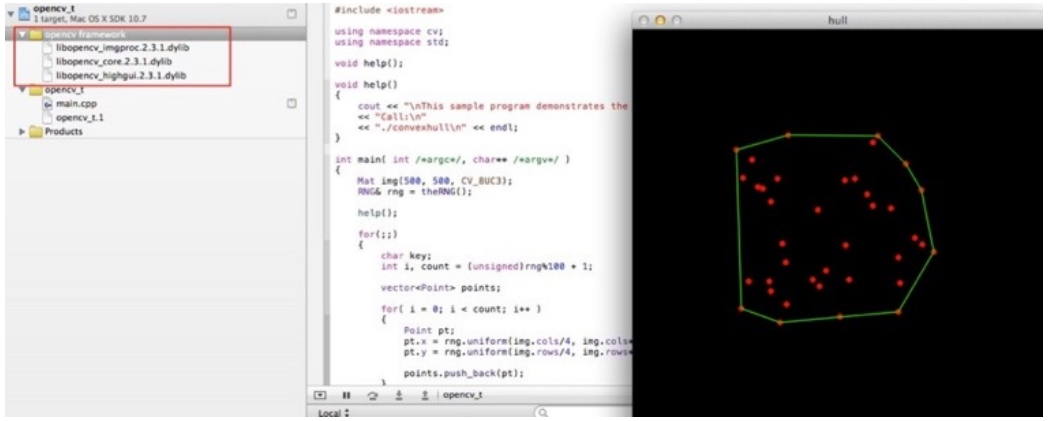
- Convolution enables variably-sized inputs
  - Audio recording of different lengths
  - Image with more/fewer pixels
- Weight-sharing reduces trainable parameters
- Convolutions are a well-understood primitive in computer vision

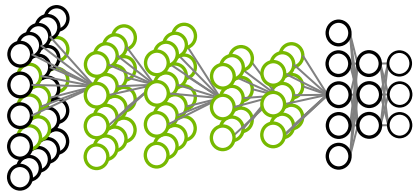
LeNet-5:CNN for hand-written digit recognition

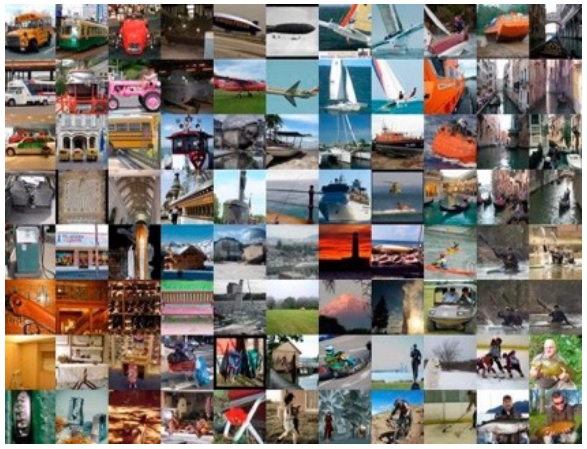


Lecun, Bottou, Bengio, Haffner, “Gradient-based Learning Applied to Document Recognition”, 1998

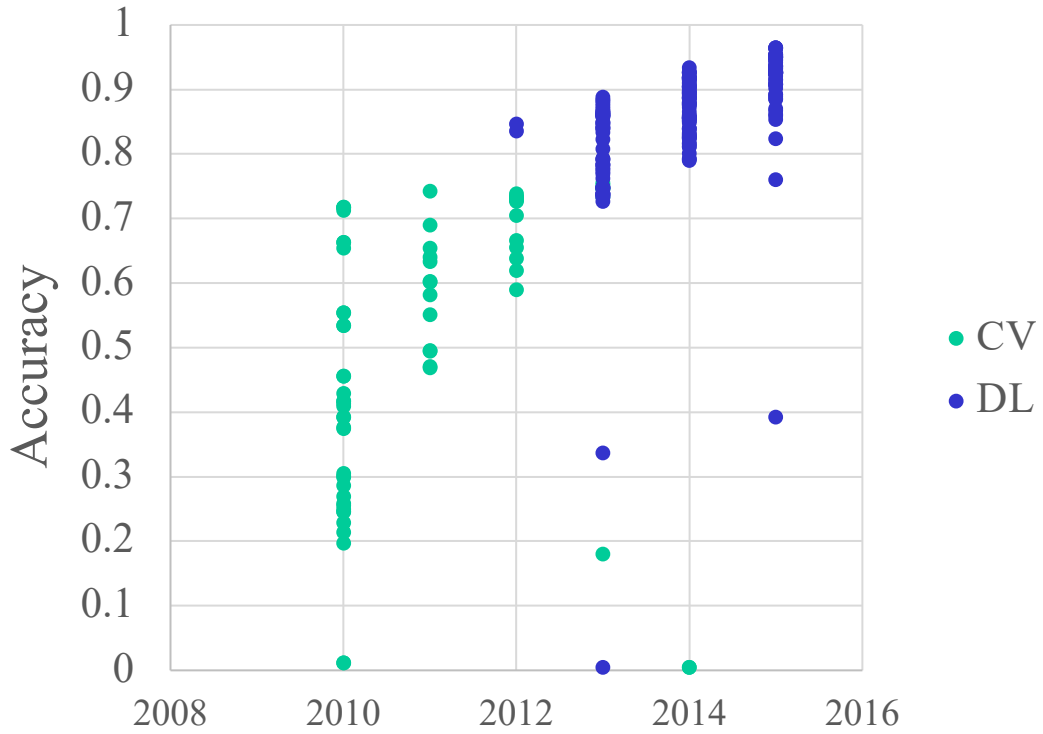
Deep Learning Impact in Computer Vision







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



ANY MORE QUESTIONS?