ECE408/CS483/CSE408 Fall 2021

Applied Parallel Programming

Lecture 11:
Feed-Forward Networks and Gradient-Based Training

Course Reminders

- Labs 2 & 3 are graded, check your grades on WebGPU
  - They will be posted in Canvas later today
- We are still grading Lab 4
- Midterm 1 is on Thursday, October 7th
  - On-line, everybody will be taking it at the same time
    - Thursday, Oct. 7th 8:00pm-9:20pm US Central time
    - Friday, Oct. 9th 9:00am-10:20am Beijing time
  - Includes materials from Lecture 1 through Lecture 10
- Project Milestone 1: Rai Installation and baseline CPU implementation is due Friday October 15th
  - Project details to be posted this week on course wiki

Objective

- To learn the basic approach to feedforward neural networks:
  - neural model
  - common functions
  - training through gradient descent

Let's Look at Classification

In a **classification problem**, we model

- a function mapping an input vector to a set of C categories: $\boldsymbol{F} : \mathbb{R}^N \rightarrow \{\mathbf{1}, \ldots, \boldsymbol{C}\}$,
- where the function ***F* is unknown**.

We **approximate *F* using a set of functions *f***

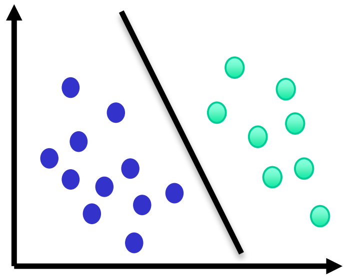
- parametrized by a (large) set of weights, $\boldsymbol{\theta}$
- that map from a vector of N real values* to an integer value representing a category:
- for category i, $\mathbf{prob}(\boldsymbol{i}) = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\theta})$

*floating-point values

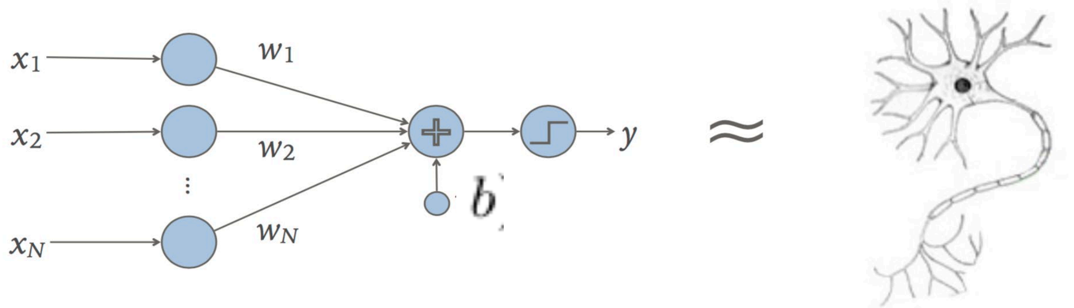
Perceptron is a Simple Example

- Example: a **perceptron**

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



The perceptron *The neuron*



- Dot product: $y = W \bullet \boldsymbol{x}$
- Scalar addition: $+ b$

output
input
weight
bias

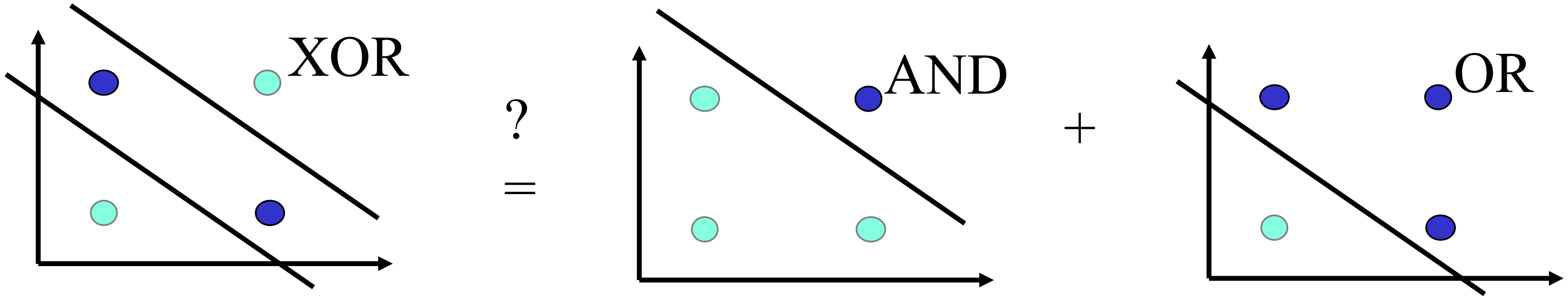
One Perceptron is not Enough

Some functions are non-linear

What can we do?

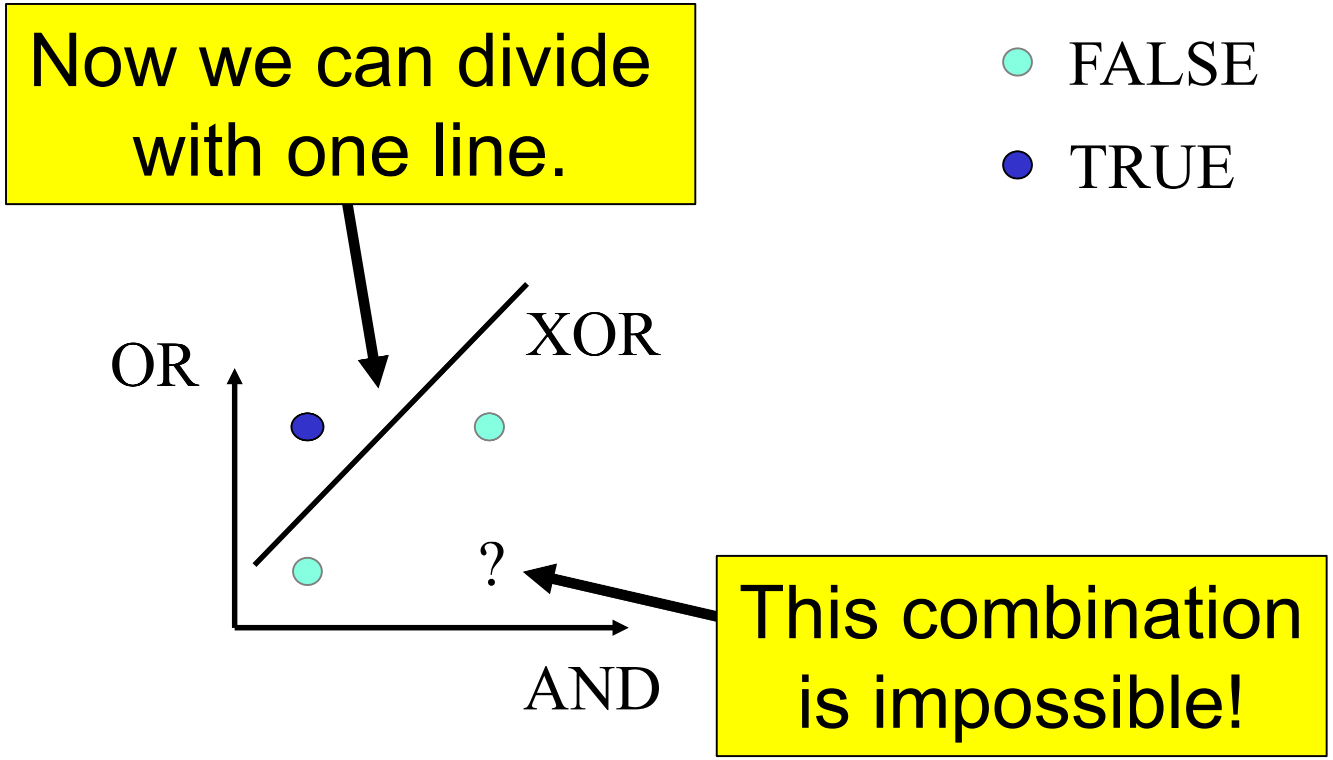
FALSE

TRUE



Multiple Layers Solve More Problems

What if input dimensions are AND and OR?

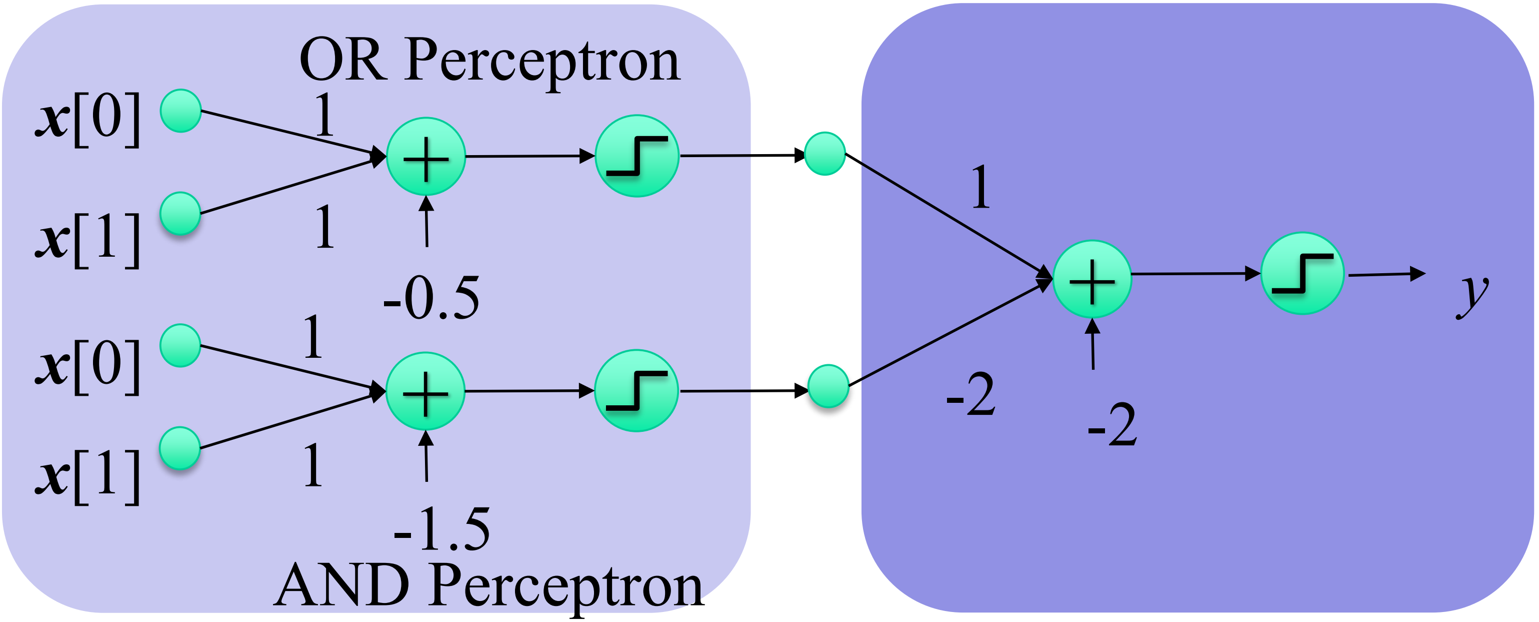


A	B	OR	AND	XOR
0	0	-1	-1	-1
0	1	1	-1	1
1	0	1	-1	1
1	1	1	1	-1

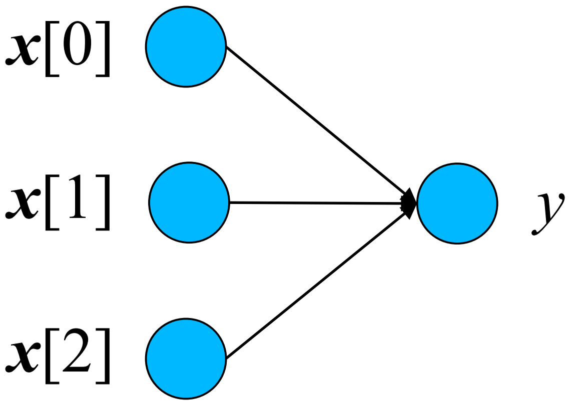
$$\text{AND} = \text{sign}(\boldsymbol{x}[0] + \boldsymbol{x}[1] - 1.5)$$

$$\text{OR} = \text{sign}(\boldsymbol{x}[0] + \boldsymbol{x}[1] - 0.5)$$

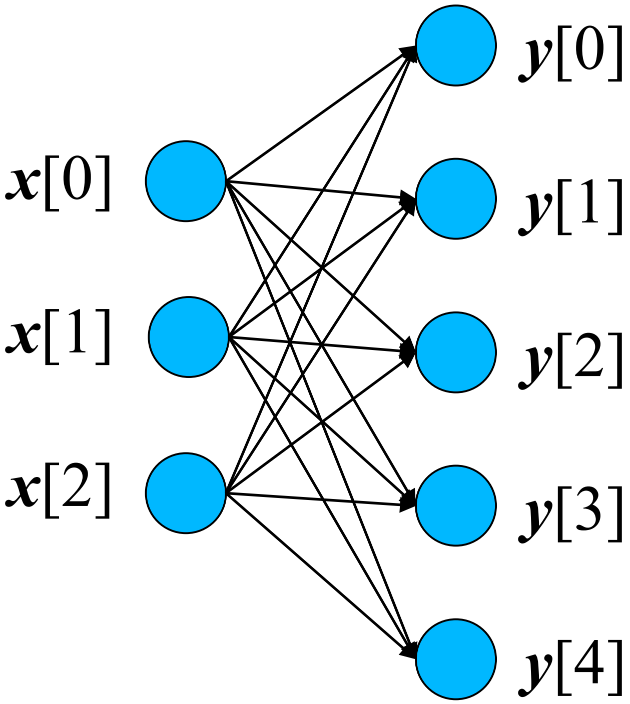
$$\text{XOR} = \text{sign}(2 * \text{OR} - \text{AND} - 2)$$



Generalize to Fully-Connected Layer

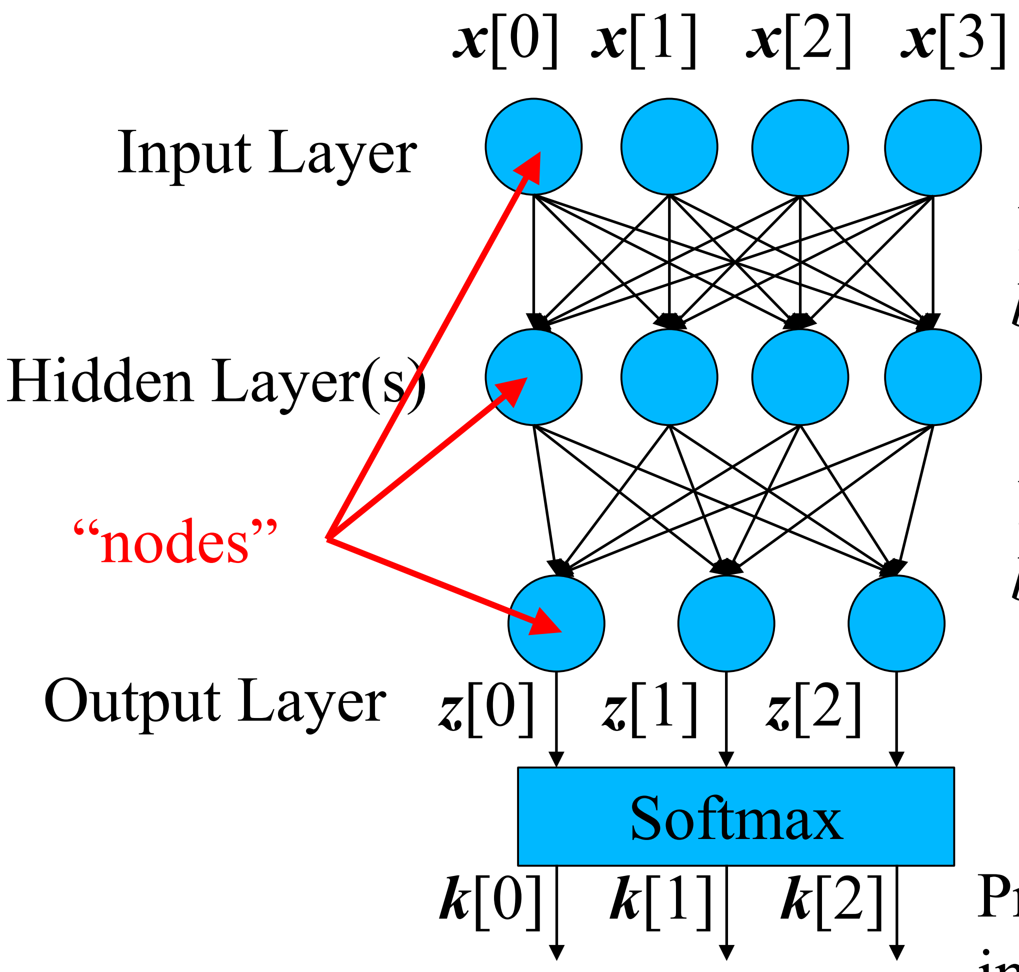


Linear Classifier:
Input vector ***x*** × weight vector ***w*** to produce scalar output *y*



Fully-connected:
Input vector ***x*** × weight <u>matrix</u> ***w*** to produce <u>vector</u> output *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

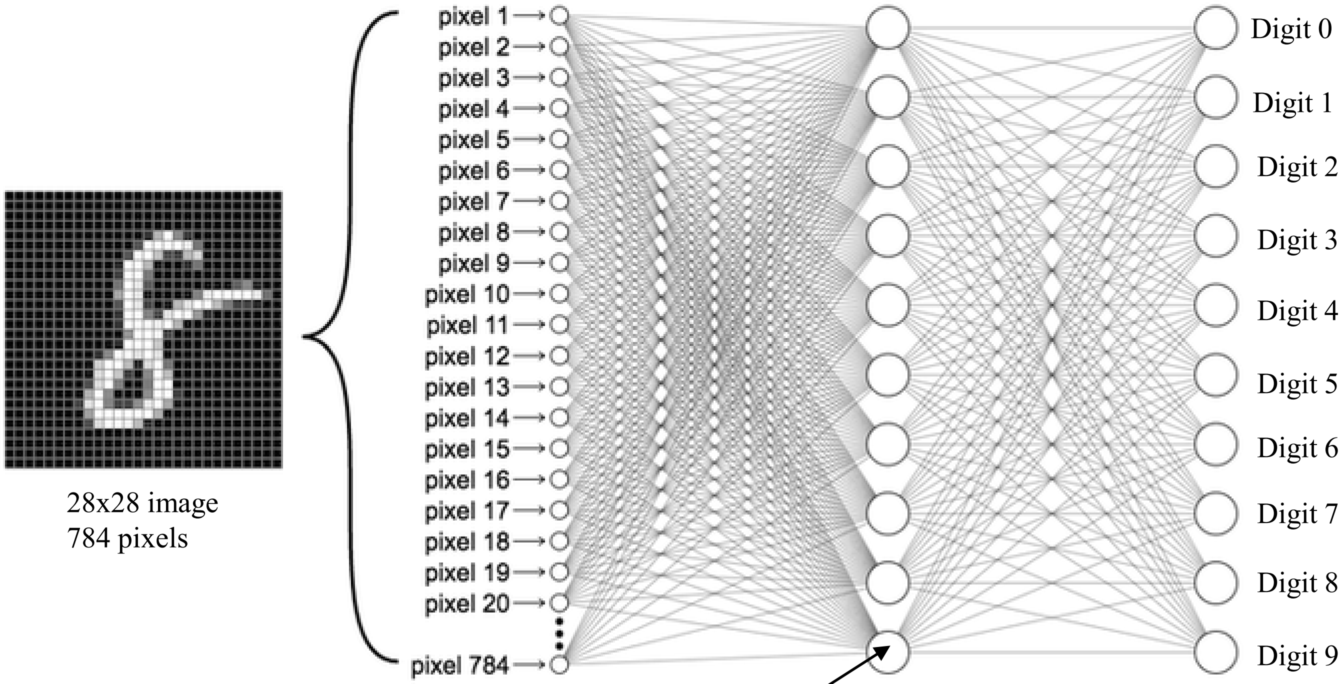
Let's consider an example.

- **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



n_k^1 = 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, **1024** outputs, **fully connected**
- **[1024 × 784]** weight matrix ***W***
- **[1024 x 1]** bias vector ***b***

Use labeled training data to pick weights.

Idea:

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

Forward and Backward Propagation

Forward (**inference**):

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

Backward (**training**):

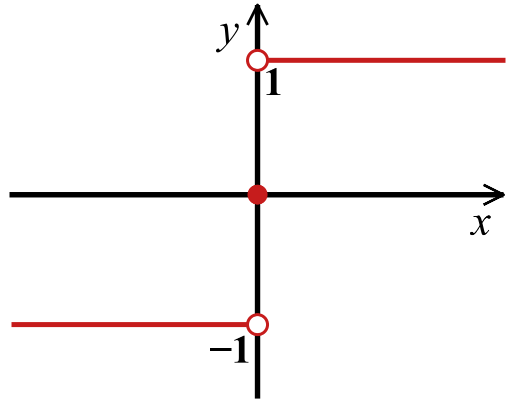
- given input $\boldsymbol{x}$, parameters $\boldsymbol{\theta}$, and outputs ***k[i]***,
- **compute error $\boldsymbol{E}$** based on target label $\boldsymbol{t}$,
- then **adjust $\boldsymbol{\theta}$** proportional to $\boldsymbol{E}$ to reduce error.

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.

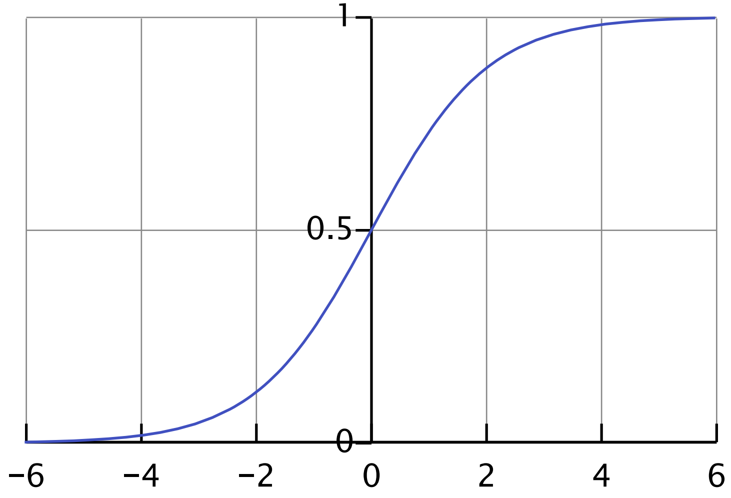
One Choice: Sigmoid/Logistic Function

Until about 2017,

- **sigmoid / logistic function** most popular

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

  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**

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

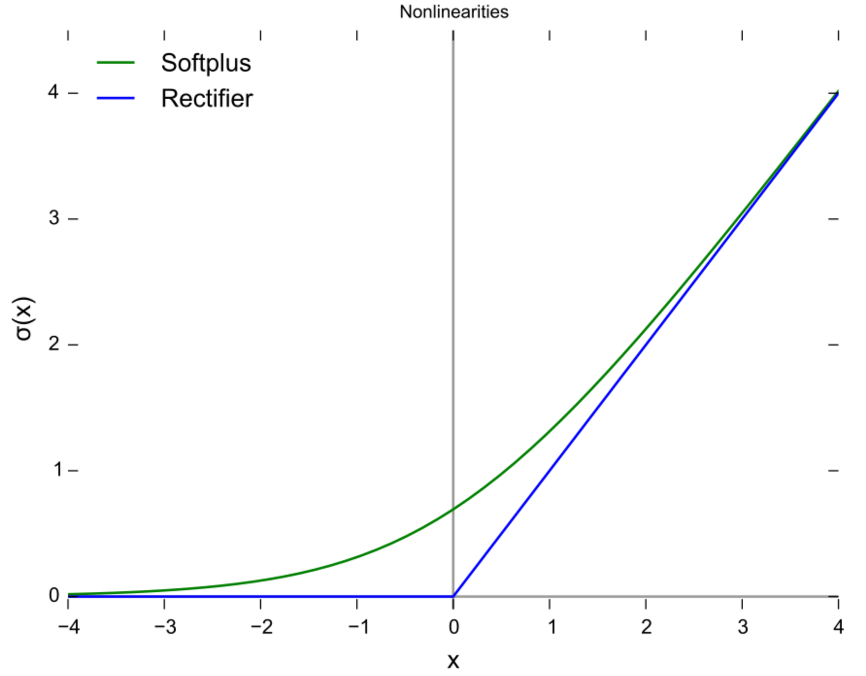
  which is much faster (no exponent required).

- 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**

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

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

*Remember that we classify into one of C categories.

Softmax Derivatives Needed to Train

We also need the **derivatives of softmax**,

$$\frac{dk[i]}{dz[m]} = k[i](\delta_{i,m} - k[m]),$$

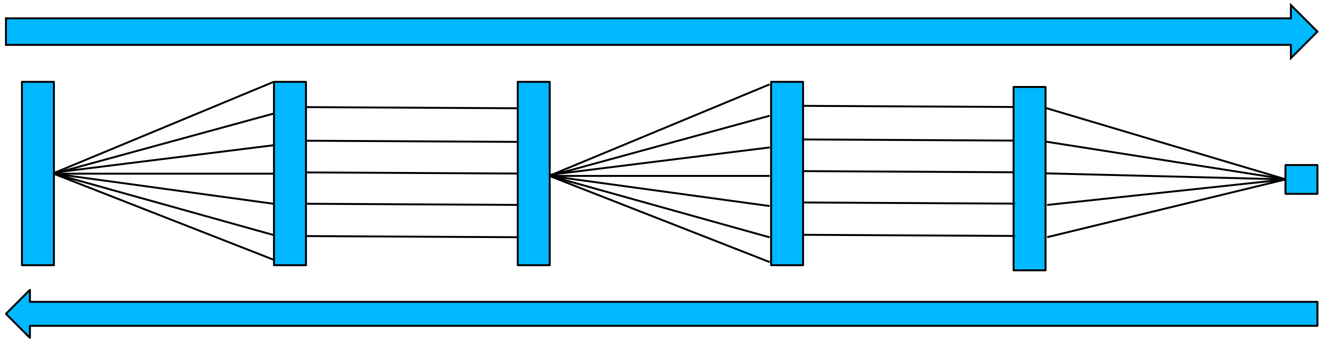
where $\delta_{i,m}$ is the Kronecker delta
(1 if i = m, and 0 otherwise).

Forward and Backward Propagation

Forward:

$fc_1 = W_1 \bullet x + b_1$ $\quad y = sigmoid(fc_1)$ $\quad fc_2 = W_2 \bullet y + b_2$ $\quad z = sigmoid(fc_2)$ $\quad k = softmax(z)$

$\boldsymbol{x}$



dE/dk

Backward:

$\frac{dE}{dfc_1} = \frac{dE}{dy}\frac{dy}{dfc_1}$ $\quad \frac{dE}{dy} = \frac{dE}{dfc_2}\frac{dfc_2}{dy}$ $\quad \frac{dE}{dfc_2} = \frac{dE}{dz}\frac{dz}{dfc_2}$ $\quad \frac{dE}{dz} = \frac{dE}{dk}\frac{dk}{dz}$

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**:

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

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

Stochastic Gradient Descent

How do we calculate the weights?

One common answer: **stochastic gradient descent**.

1. **Calculate**
   - **derivative** of sum **of error *E***
   - **over all** training **inputs**
   - **for** all network parameters $\boldsymbol{\theta}$.
2. **Change $\boldsymbol{\theta}$ slightly** in the opposite direction (to decrease error).
3. **Repeat**.

Stochastic Gradient Descent

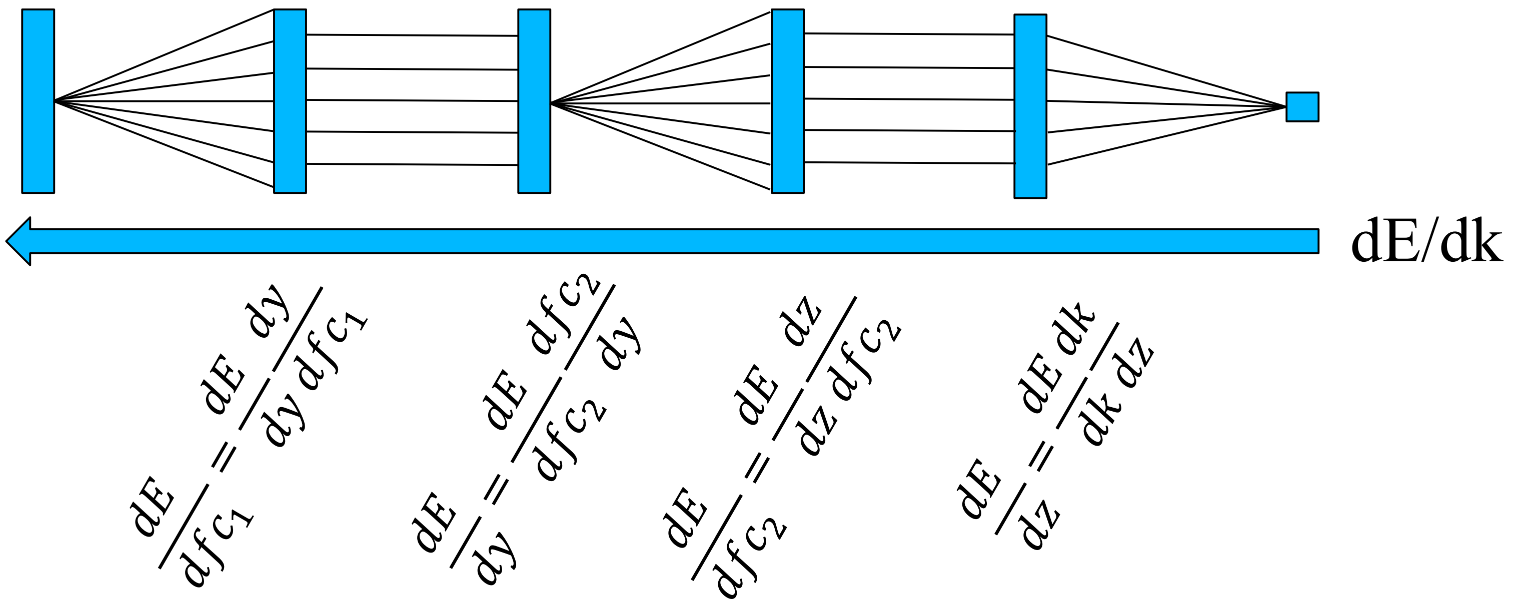
More precisely,

1. **For every input *X***,
2. evaluate network to **compute *k[i]*** (forward),
3. then **use *k[i]* and label *T*** (target digit) **to compute error *E***.
4. Backpropagate error derivative to **find derivatives for each parameter**.
5. **Adjust θ to reduce total *E*: $\theta_{i+1} = \theta_i - \varepsilon \Delta\theta$**

(Update ε uses most accurate minima estimation.)

Parameter Updates and Propagation

$$\boldsymbol{fc_1} = W_1 \bullet \boldsymbol{x} + b_1$$



Need propagated error gradient (from backward pass)

Weight update

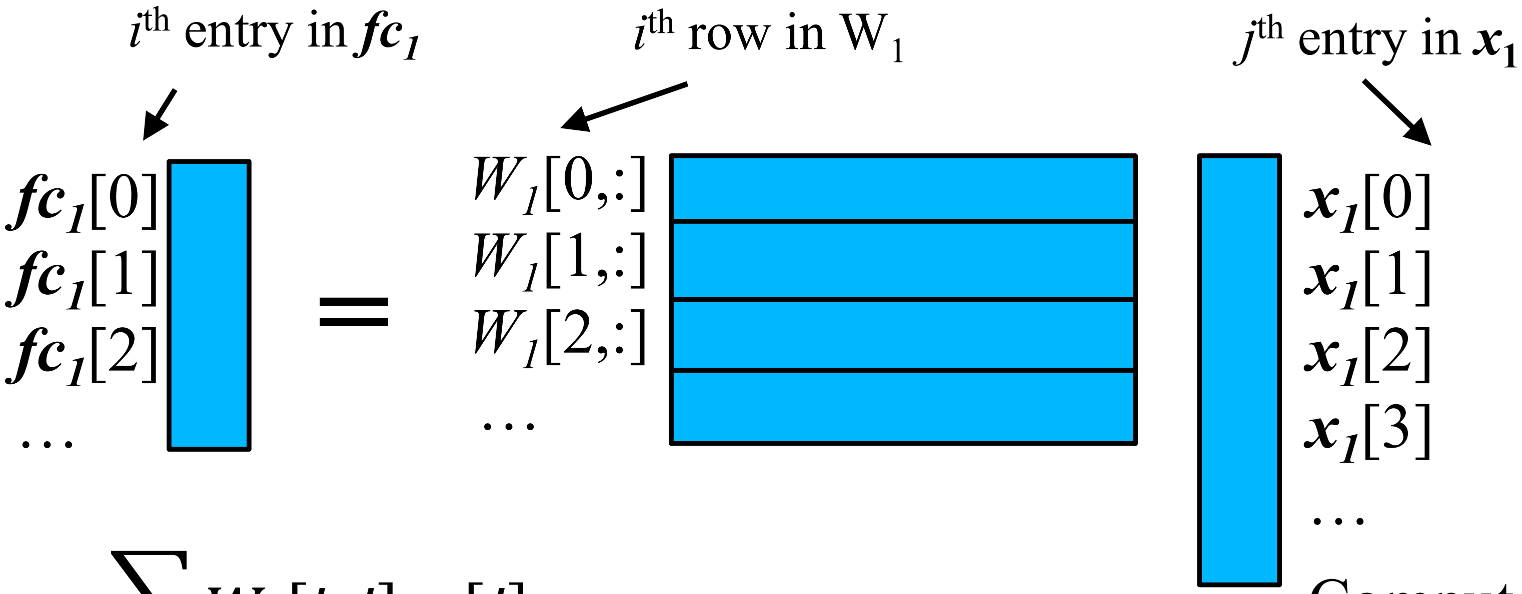
$$\frac{dE}{dW_1} = \frac{dE}{dfc_1}\frac{dfc_1}{dW_1} = \frac{dE}{dfc_1}x$$

Need input (from forward pass)

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
$E = ½\,(y - t)^2$	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

Batched Stochastic Gradient Descent

- A training *epoch* (a pass through whole training set)
  - *Set* $\Delta\Theta = 0$
  - For each labeled image:
    - Read data to initialize input layer
    - Evaluate network to get *y* (forward)
    - Compare with target label *t* to get error *E*
    - Backpropagate error derivative to get parameter updates
    - Accumulate parameter updates into $\Delta\Theta$
  - $\Theta_{i+1} = \Theta_i - \varepsilon\Delta\Theta$

Aggregate gradient update most accurately reflects true gradient

Mini-batch Stochastic Gradient

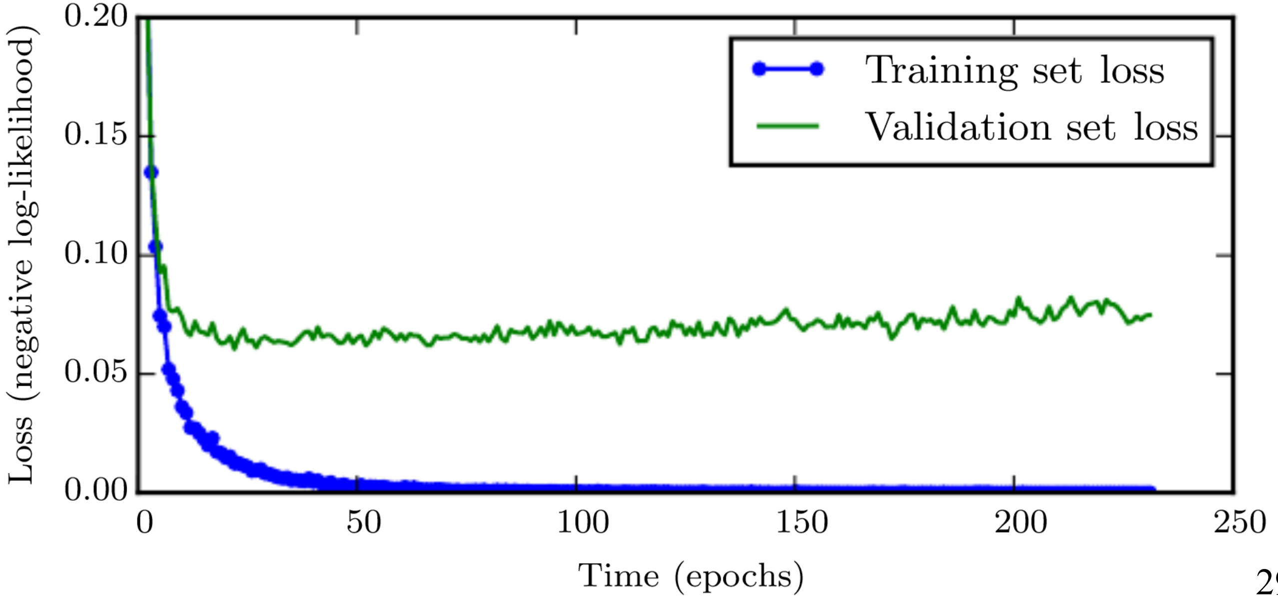
- For each batch in training set
  - For each labeled image in batch:
    - Read data to initialize input layer
    - Evaluate network to get *y* (forward)
    - Compare with target label *t* to get error *E*
    - Backpropagate error derivative to get parameter updates
    - Accumulate parameter updates into $\Delta\Theta$
  - $\Theta_{i+1} = \Theta_i - \varepsilon\Delta\Theta$

Balance between accuracy of gradient estimation and parallelism

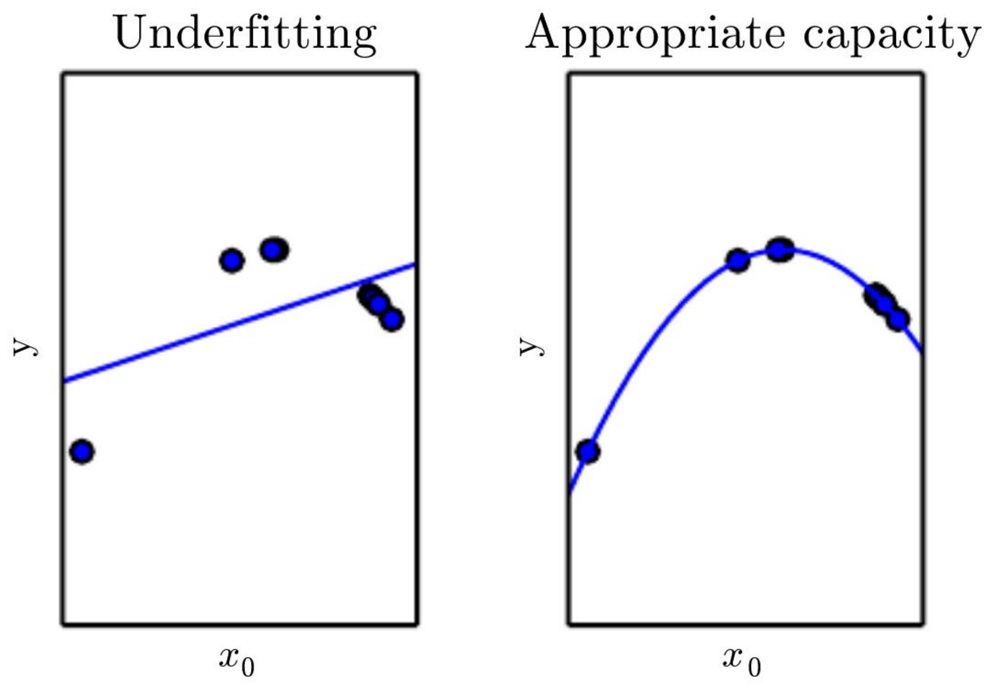
When is Training Done?

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

- Training data to compute parameter updates.
- Test data to check how model generalizes to new inputs (the ultimate goal!)
- The network can become *too good* at classifying training inputs!

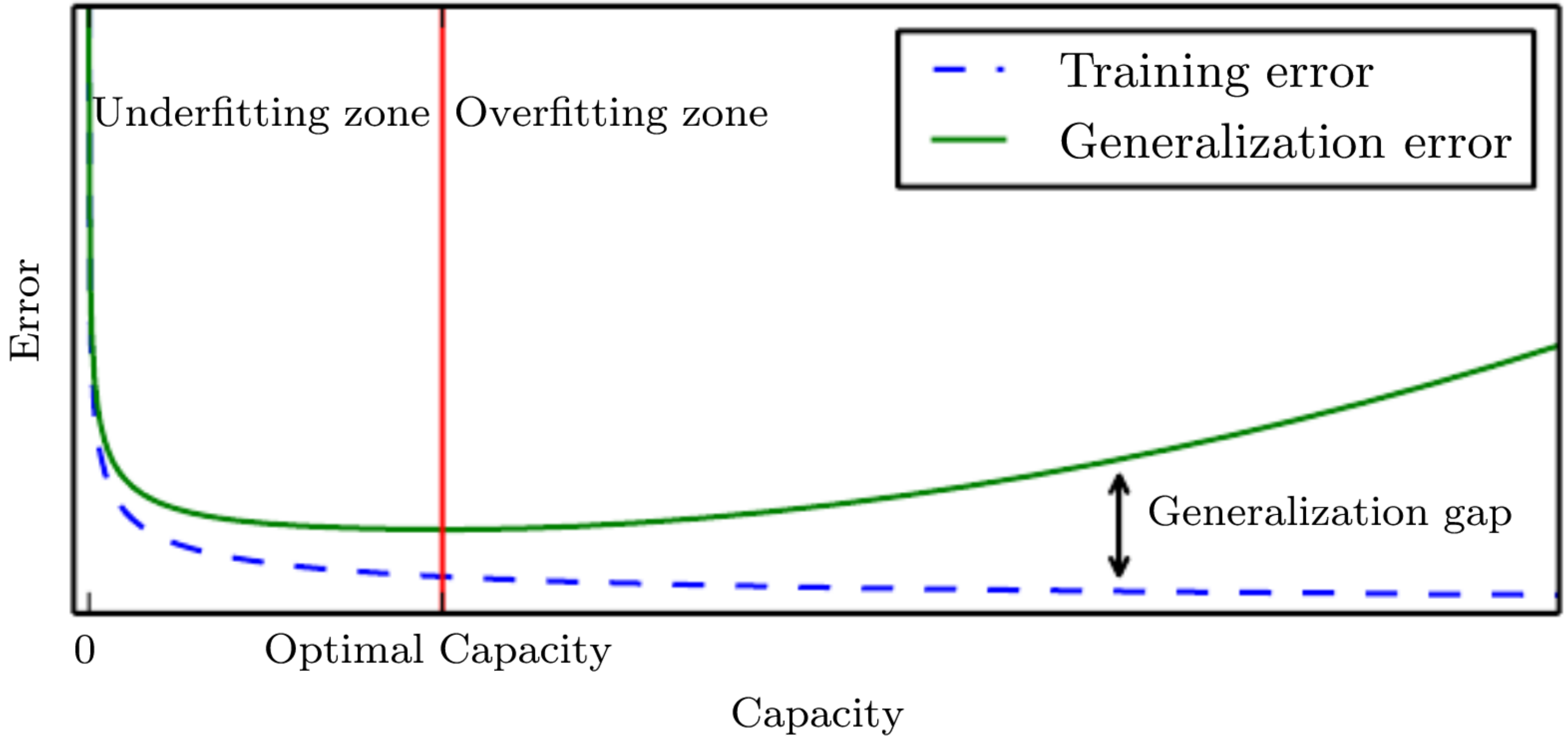


How Complicated Should a Network Be?



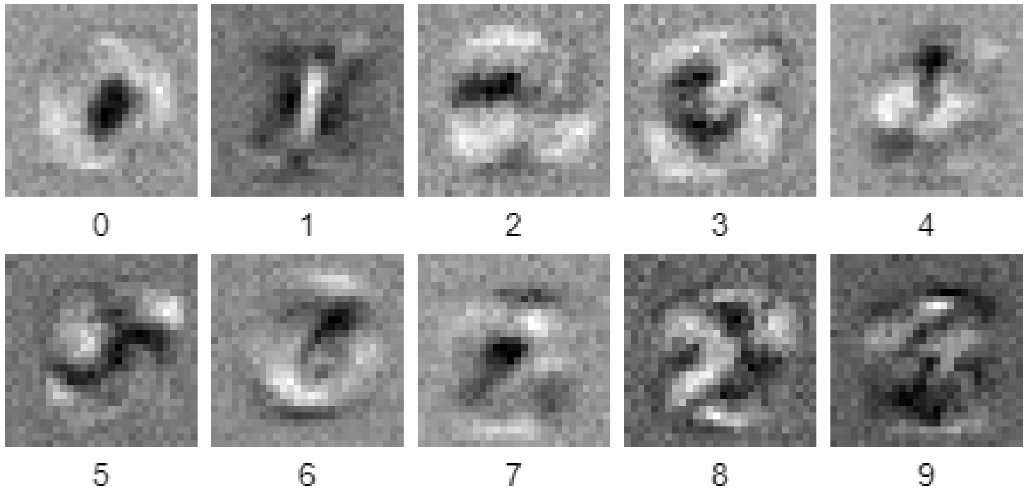
Intuition: like a polynomial fit. High-order terms improve fit, but add unpredictable swings for inputs outside the training set.

Overtraining Decreases Accuracy

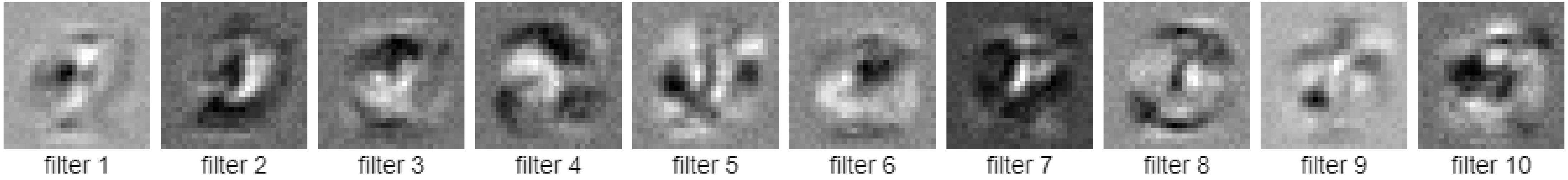


If network works too well for training data,
new inputs cause big unpredictable output changes.

Visualizing Neural Network Weights



MNIST 1st layer



MNIST 2nd layer

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

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

Summary (1)

- Classification:
  - $f : \mathbb{R}^N \rightarrow \{1, \ldots, C\}$
  - $k[i] = f(x, \theta)$
- Current ML work driven by cheap compute, lots of available data
- Perceptron as a trivial deep network
  - $y = sign(W \bullet \boldsymbol{x} + b)$
- Forward for inference, backward for training

Summary (2)

- Chain rule to compute parameter updates
- Stochastic gradient descent for training

ANY MORE QUESTIONS?