



# 10-601 Introduction to Machine Learning

Machine Learning Department  
School of Computer Science  
Carnegie Mellon University

# Backpropagation

Matt Gormley  
Lecture 12  
Feb 23, 2018

# Neural Networks Outline

- **Logistic Regression (Recap)**
  - Data, Model, Learning, Prediction
- **Neural Networks**
  - A Recipe for Machine Learning
  - Visual Notation for Neural Networks
  - Example: Logistic Regression Output Surface
  - 2-Layer Neural Network
  - 3-Layer Neural Network
- **Neural Net Architectures**
  - Objective Functions
  - Activation Functions
- **Backpropagation**
  - Basic Chain Rule (of calculus)
  - Chain Rule for Arbitrary Computation Graph
  - Backpropagation Algorithm
  - Module-based Automatic Differentiation (Autodiff)



Last Lecture

This Lecture

# **ARCHITECTURES**

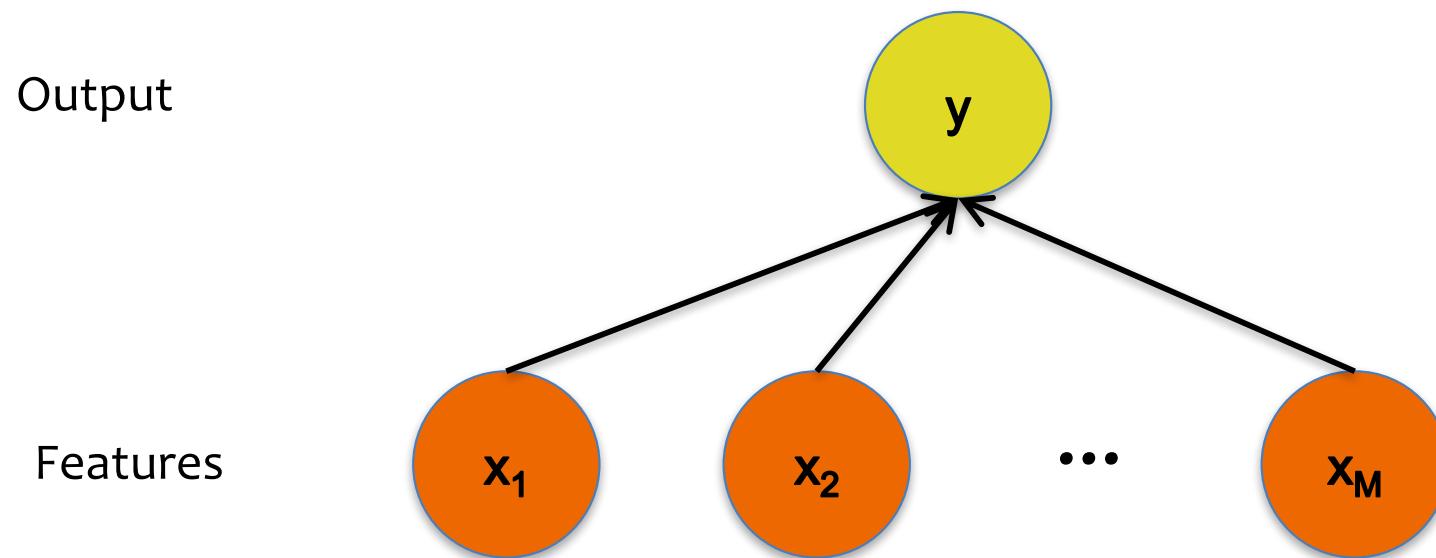
# Neural Network Architectures

Even for a basic Neural Network, there are many design decisions to make:

1. # of hidden layers (depth)
2. # of units per hidden layer (width)
3. Type of activation function (nonlinearity)
4. Form of objective function

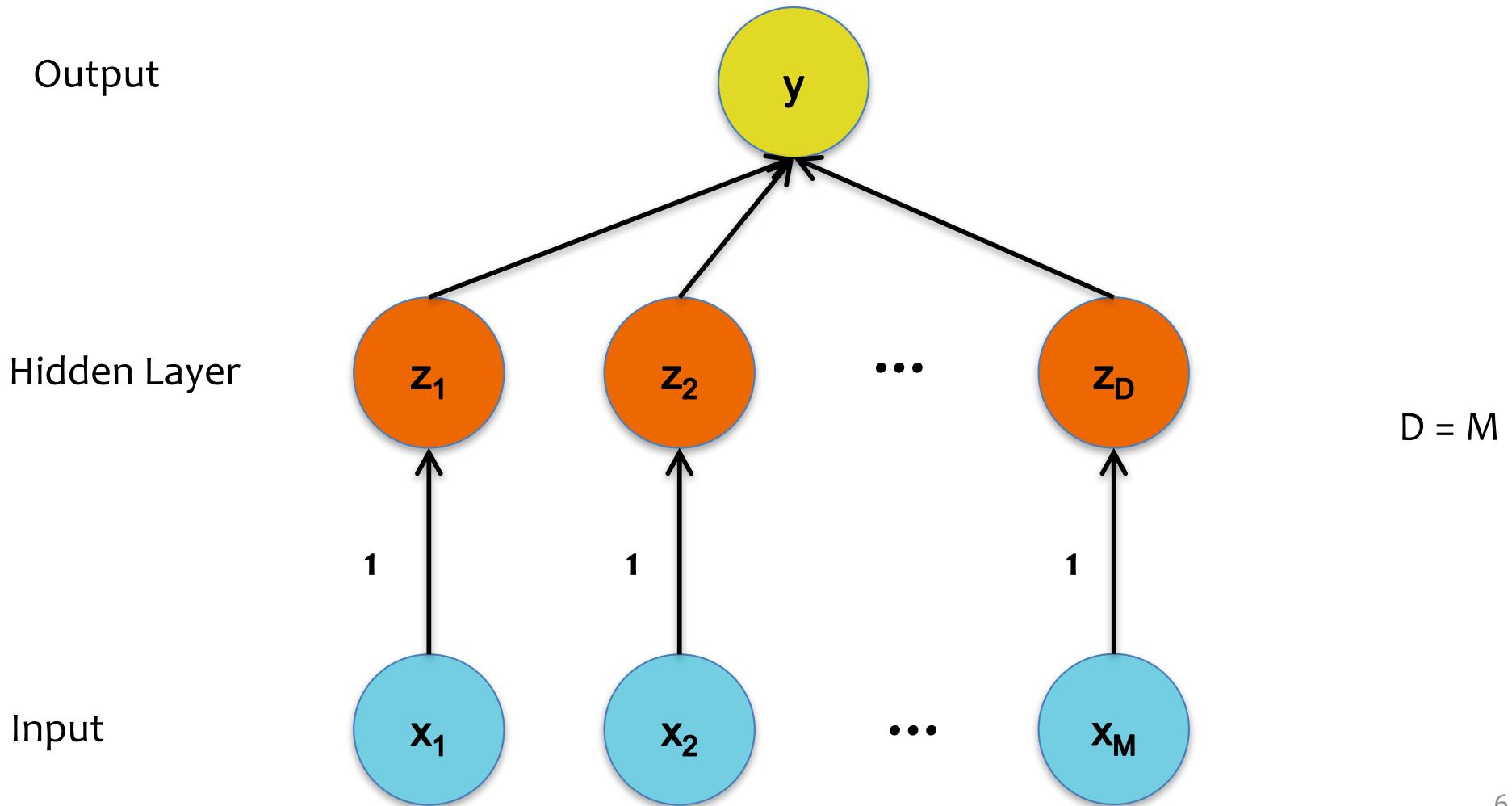
# Building a Neural Net

Q: How many hidden units,  $D$ , should we use?



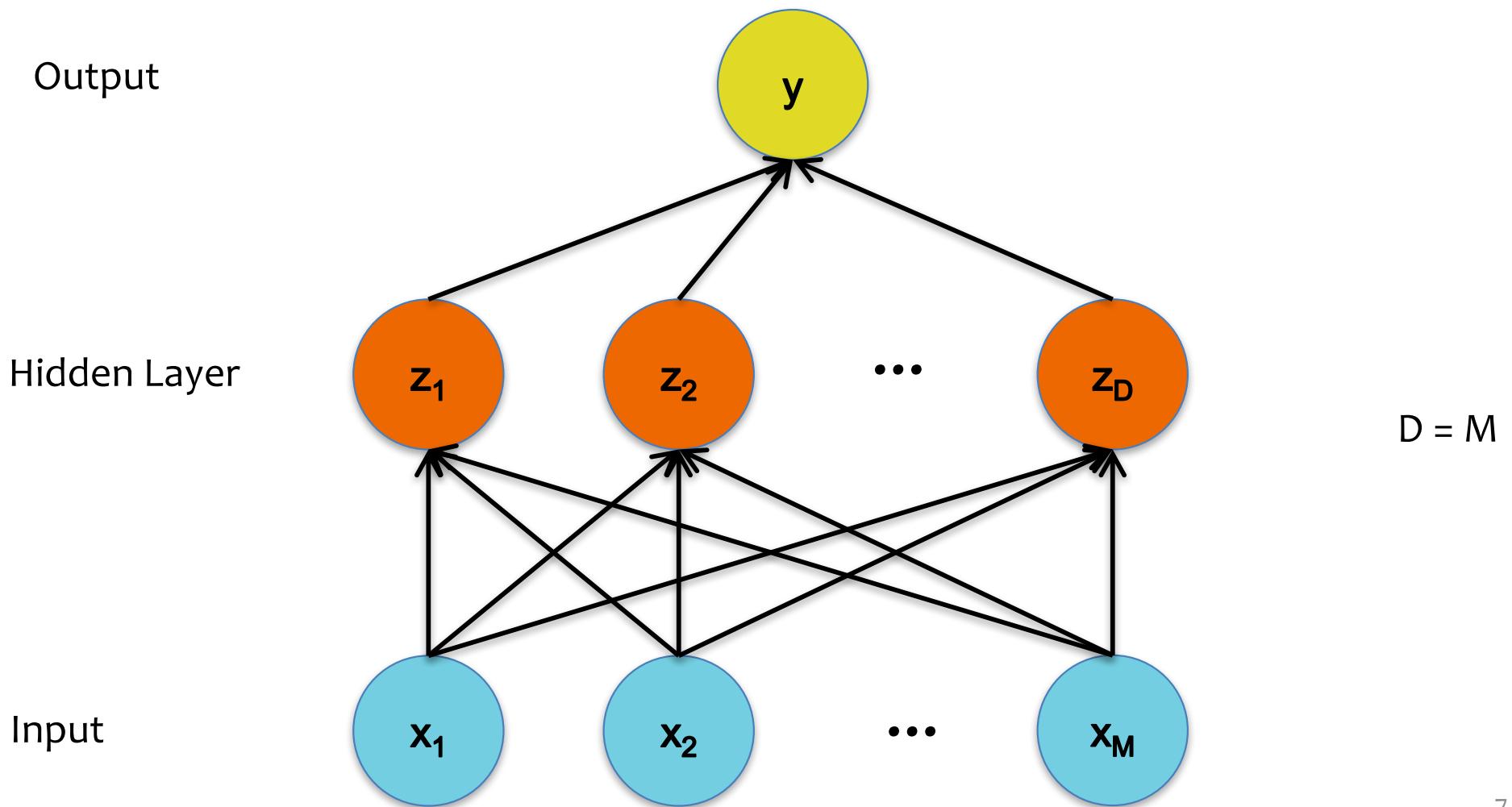
# Building a Neural Net

Q: How many hidden units,  $D$ , should we use?



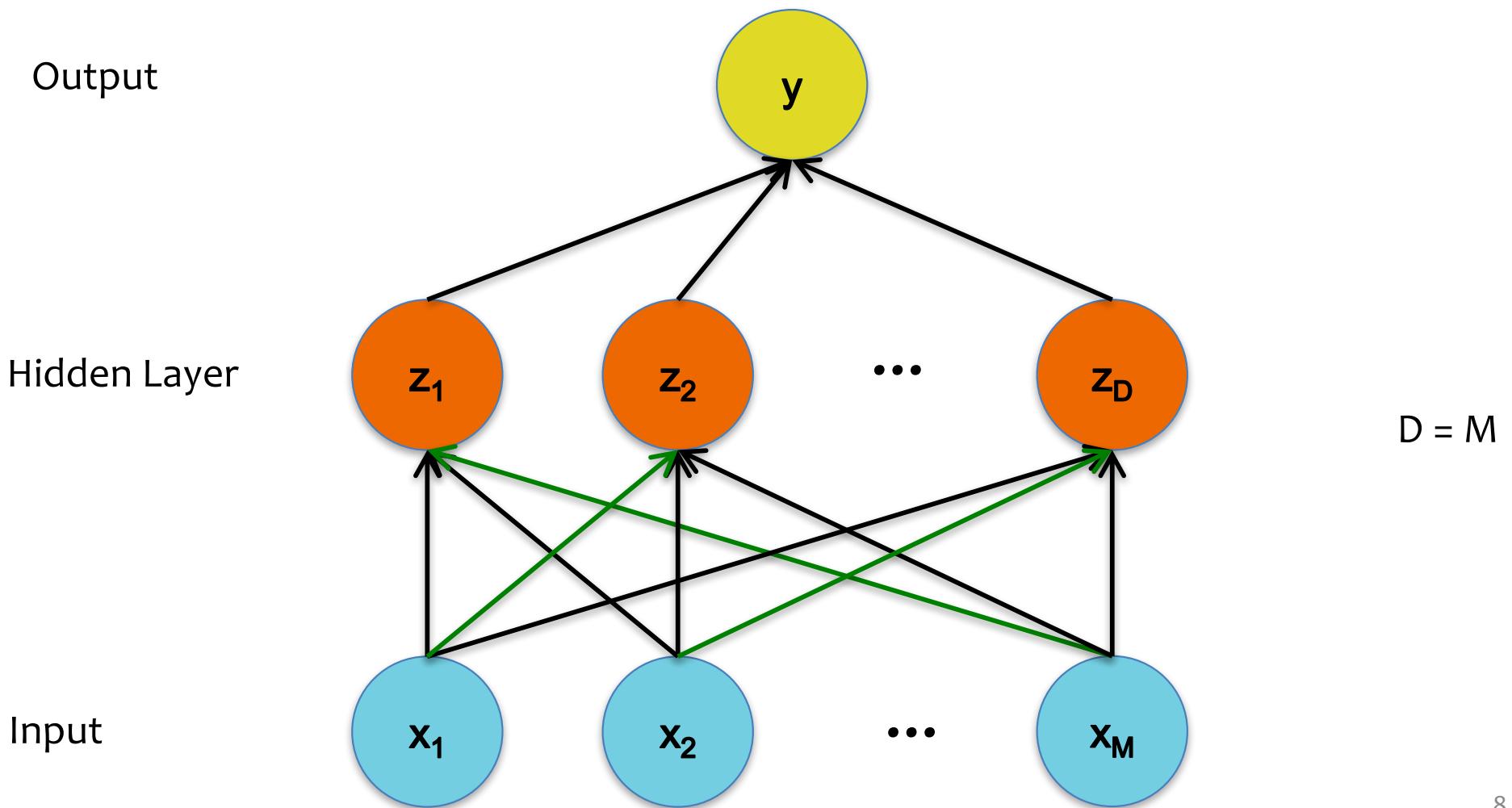
# Building a Neural Net

Q: How many hidden units,  $D$ , should we use?



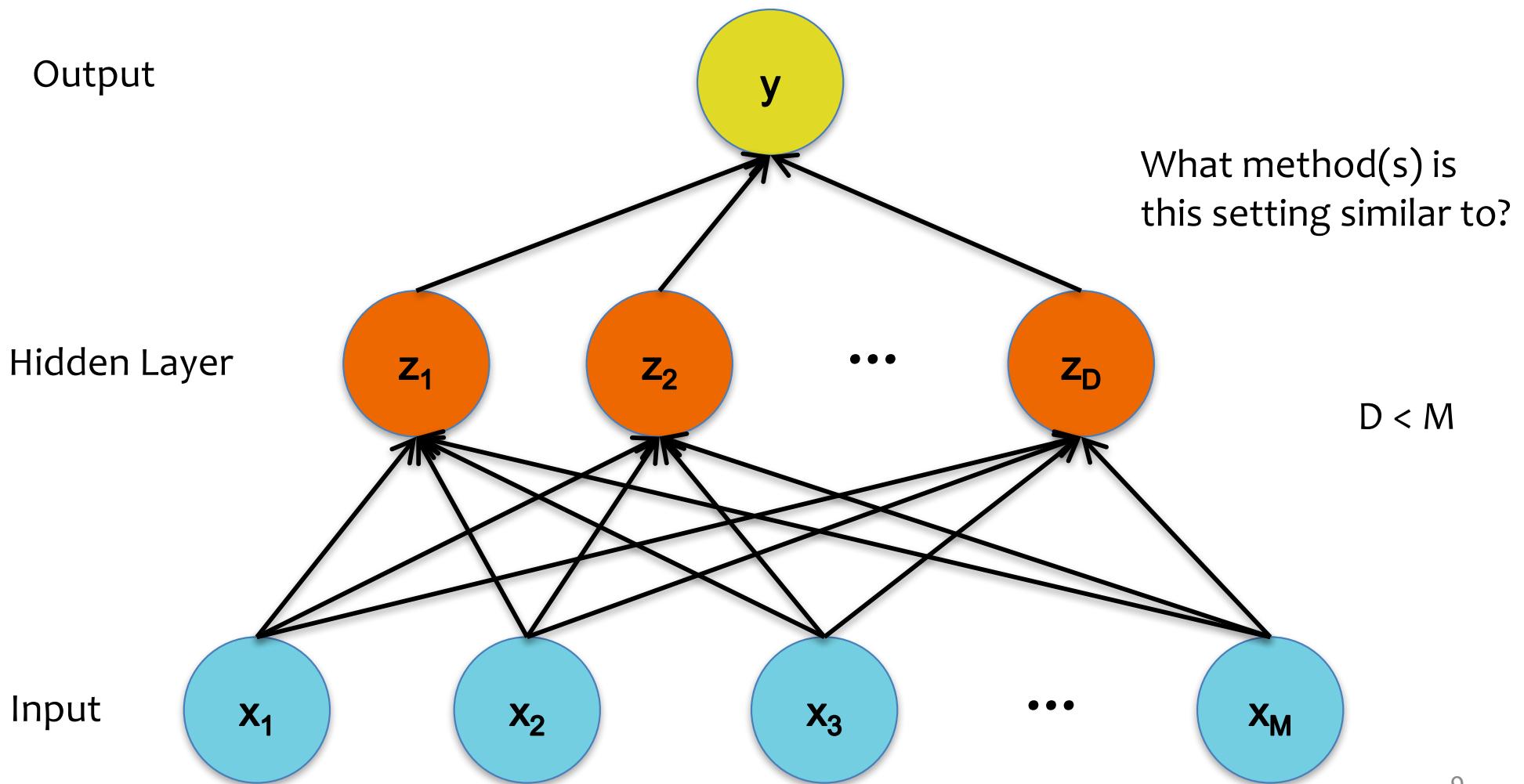
# Building a Neural Net

Q: How many hidden units,  $D$ , should we use?



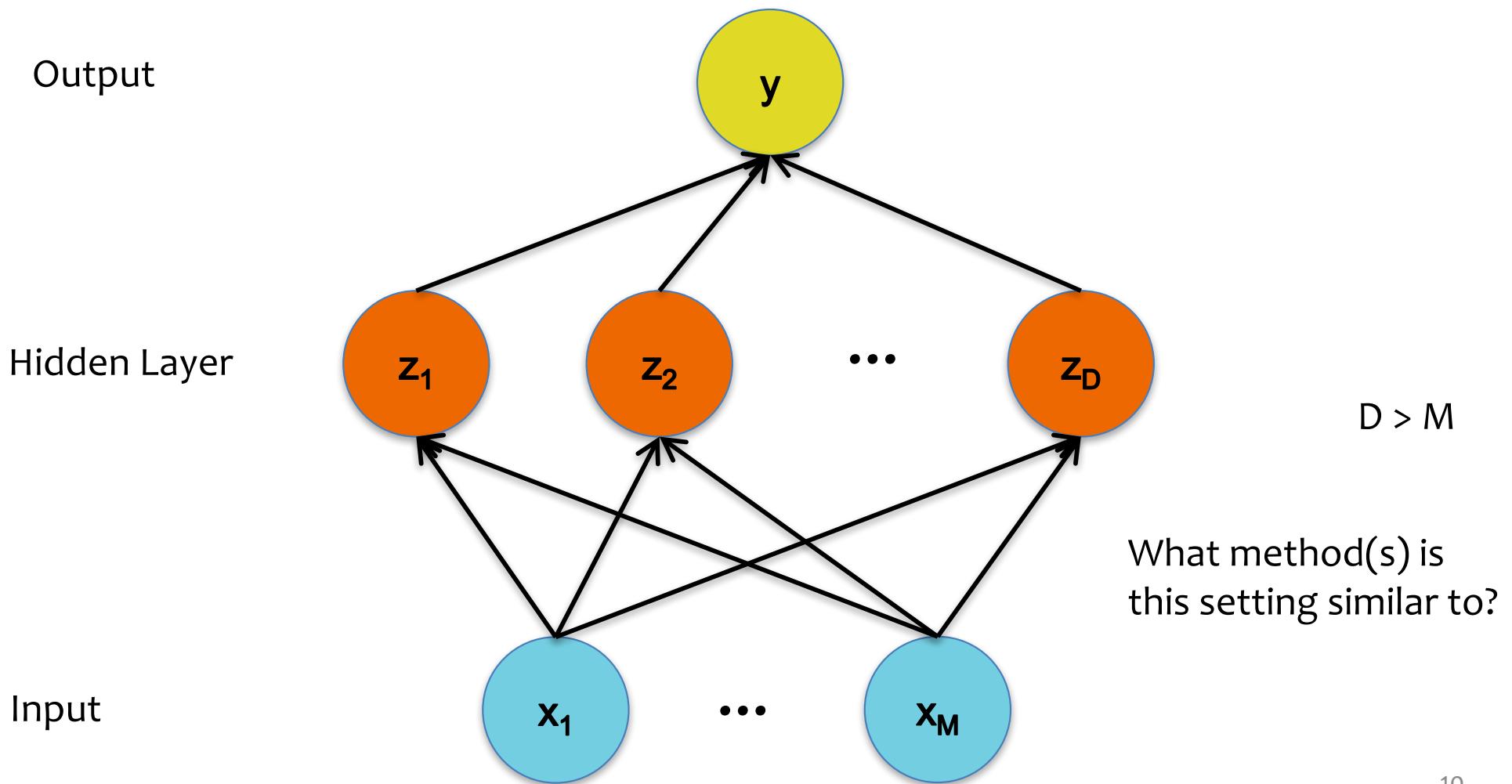
# Building a Neural Net

Q: How many hidden units,  $D$ , should we use?

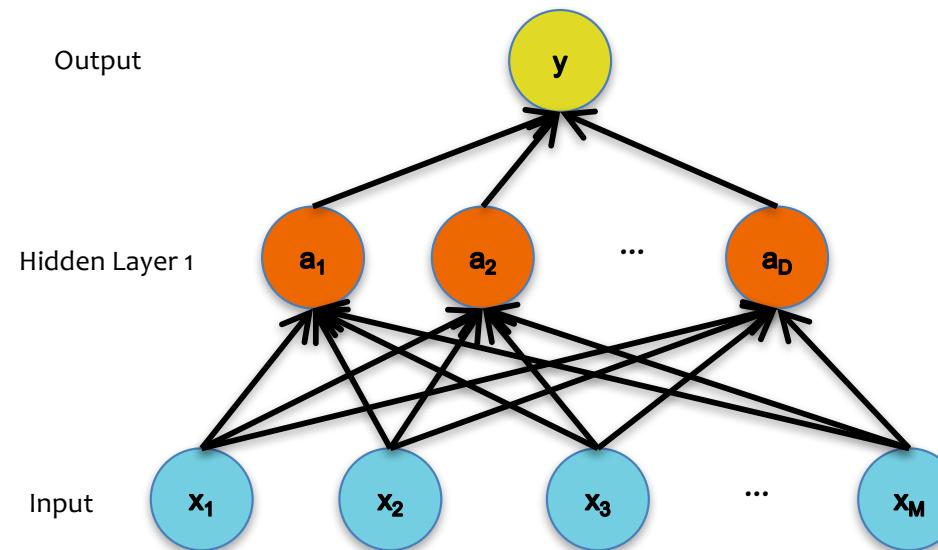


# Building a Neural Net

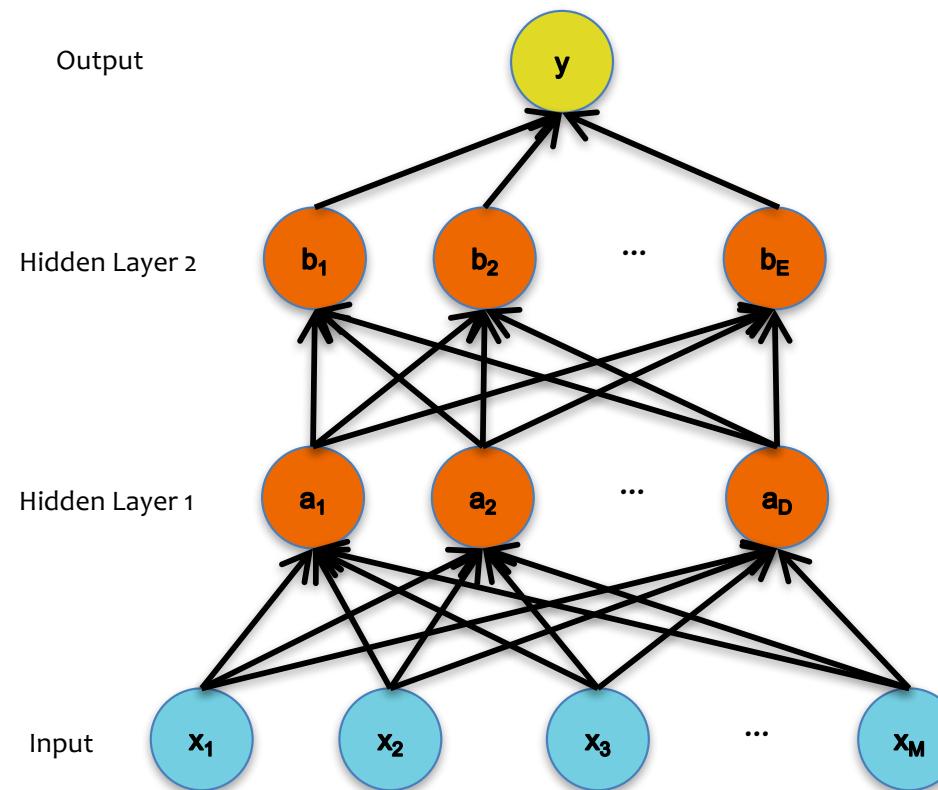
Q: How many hidden units,  $D$ , should we use?



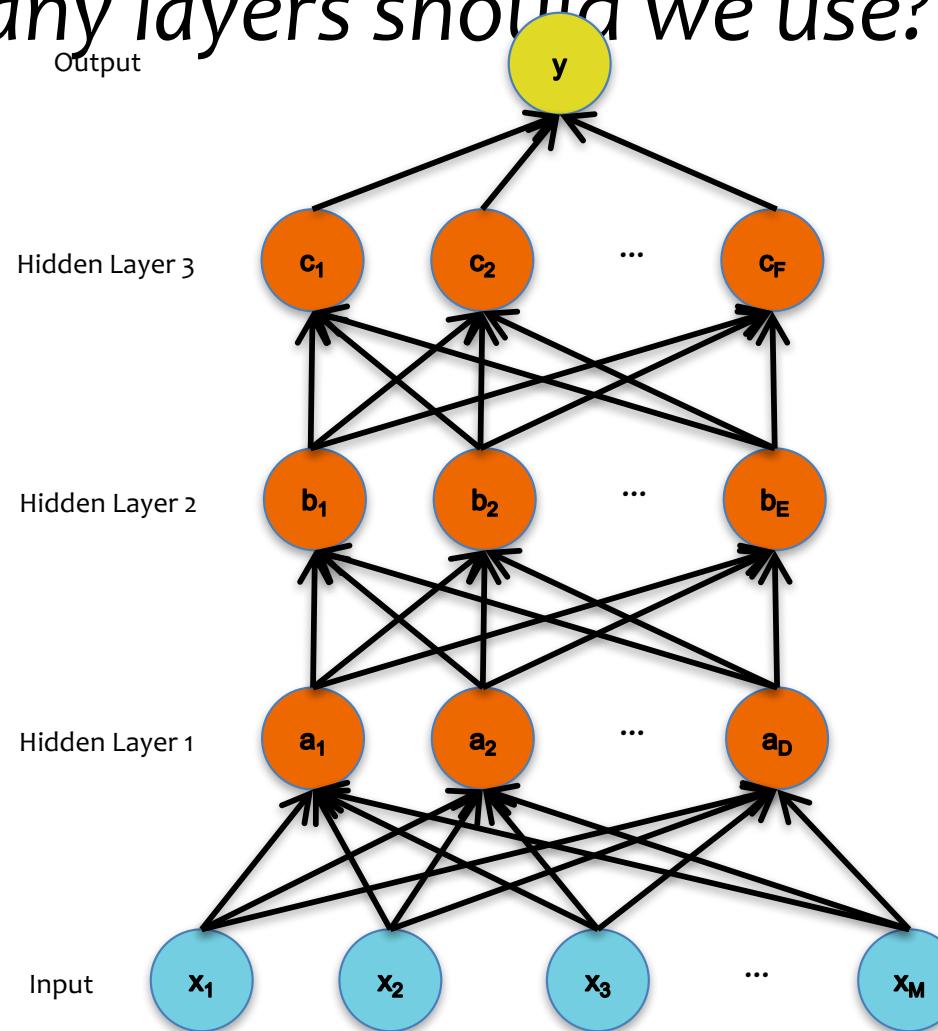
Q: How many layers should we use?



Q: How many layers should we use?



Q: How many layers should we use?



Q: How many layers should we use?

- **Theoretical answer:**
  - A neural network with 1 hidden layer is a **universal function approximator**
  - Cybenko (1989): For any continuous function  $g(\mathbf{x})$ , there exists a 1-hidden-layer neural net  $h_\theta(\mathbf{x})$  s.t.  $|h_\theta(\mathbf{x}) - g(\mathbf{x})| < \epsilon$  for all  $\mathbf{x}$ , assuming sigmoid activation functions
- **Empirical answer:**
  - Before 2006: “Deep networks (e.g. 3 or more hidden layers) are too hard to train”
  - After 2006: “Deep networks are easier to train than shallow networks (e.g. 2 or fewer layers) for many problems”

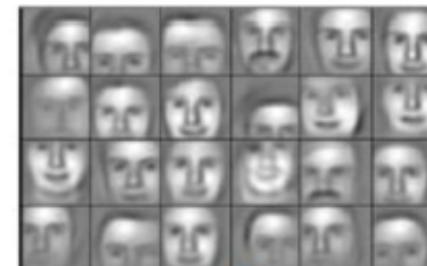
Big caveat: You need to know and use the right tricks.

# Decision Functions

- We don't know the “right” levels of abstraction
- So let the model figure it out!

# Different Levels of Abstraction

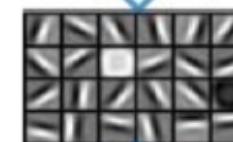
Feature representation



3rd layer  
“Objects”



2nd layer  
“Object parts”



1st layer  
“Edges”



Pixels

## Face Recognition:

- Deep Network can build up increasingly higher levels of abstraction
- Lines, parts, regions

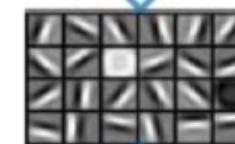
Feature representation



3rd layer  
“Objects”



2nd layer  
“Object parts”



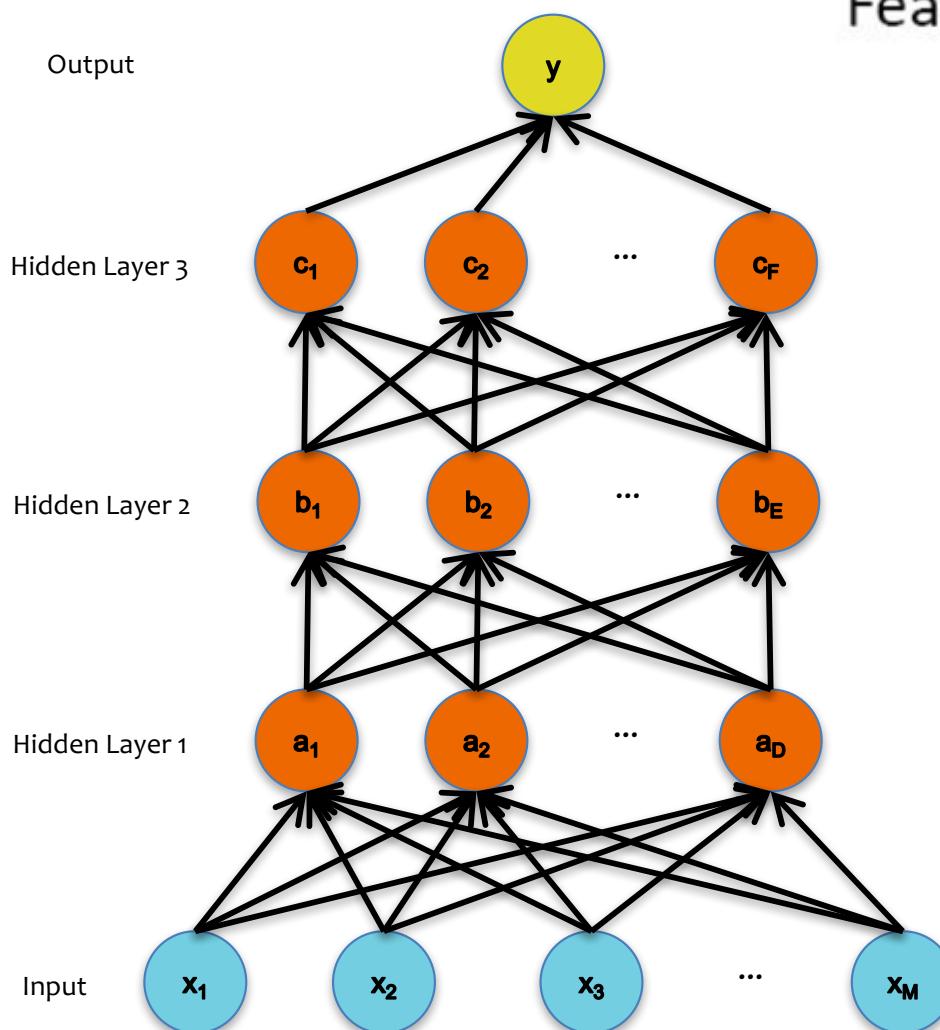
1st layer  
“Edges”



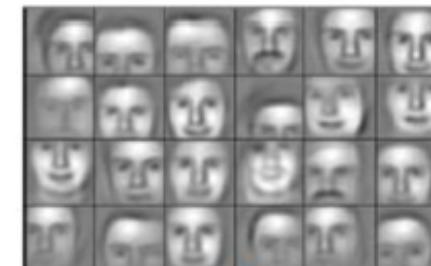
Pixels

# Decision Functions

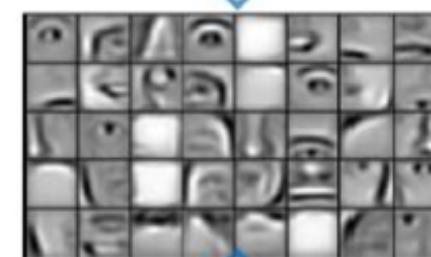
# Different Levels of Abstraction



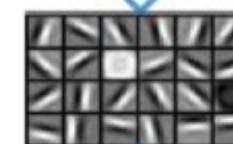
Feature representation



3rd layer  
“Objects”



2nd layer  
“Object parts”



1st layer  
“Edges”

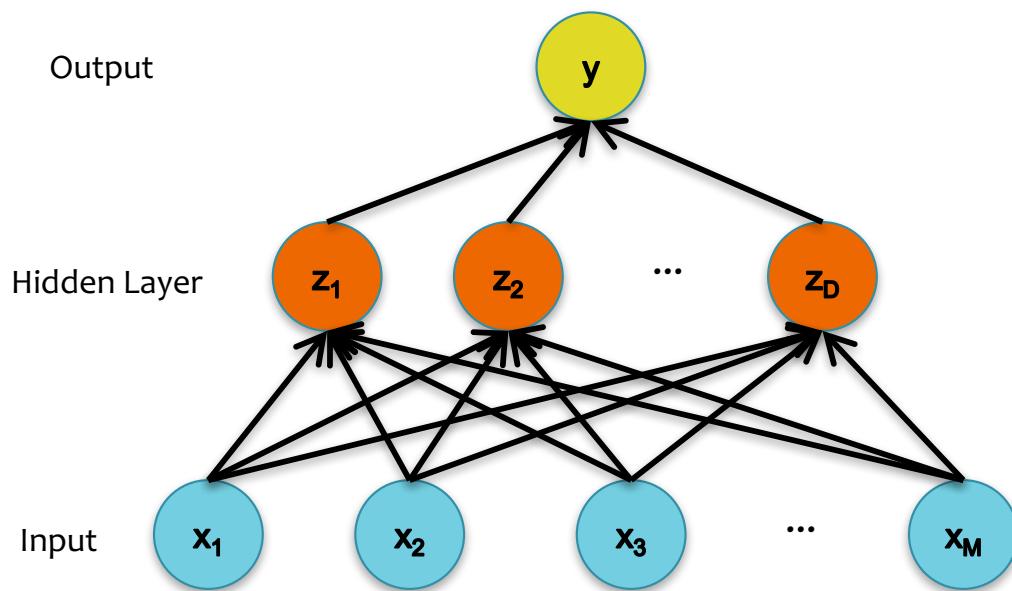


Pixels

Example from Honglak Lee (NIPS 2010)

# Activation Functions

Neural Network with sigmoid activation functions



(F) Loss

$$J = \frac{1}{2}(y - y^*)^2$$

(E) Output (sigmoid)

$$y = \frac{1}{1+\exp(-b)}$$

(D) Output (linear)

$$b = \sum_{j=0}^D \beta_j z_j$$

(C) Hidden (sigmoid)

$$z_j = \frac{1}{1+\exp(-a_j)}, \forall j$$

(B) Hidden (linear)

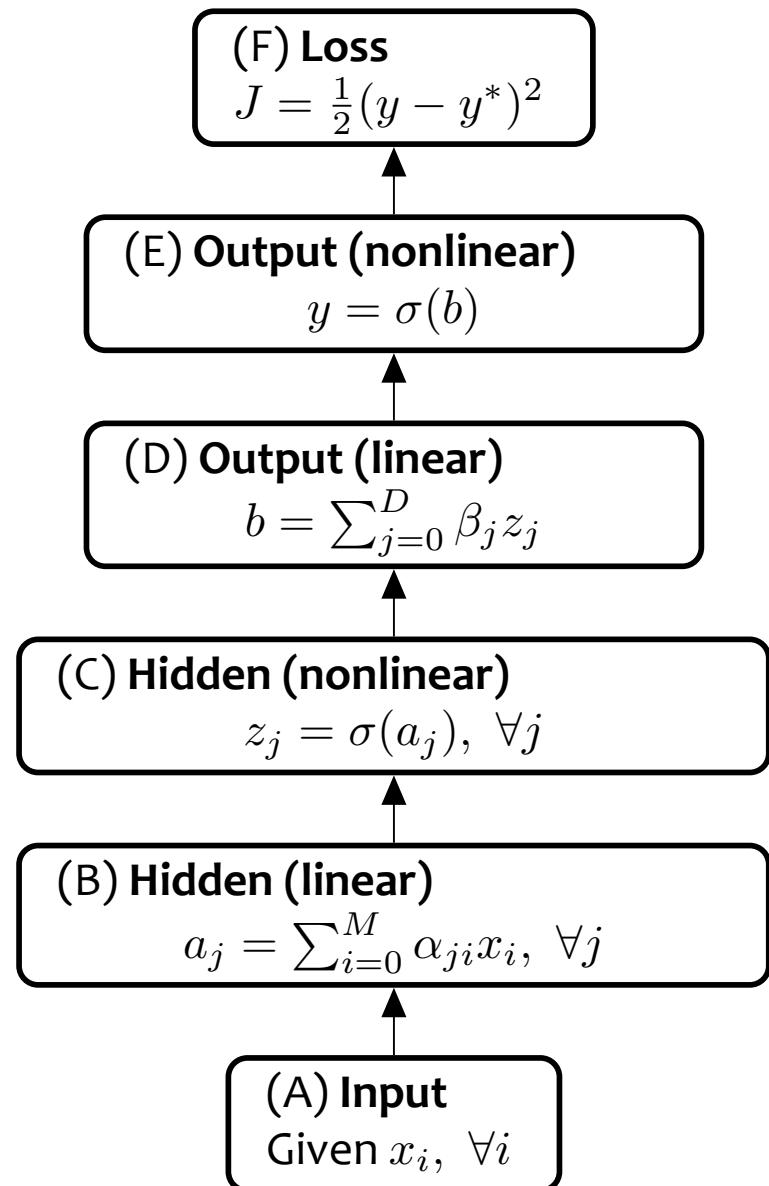
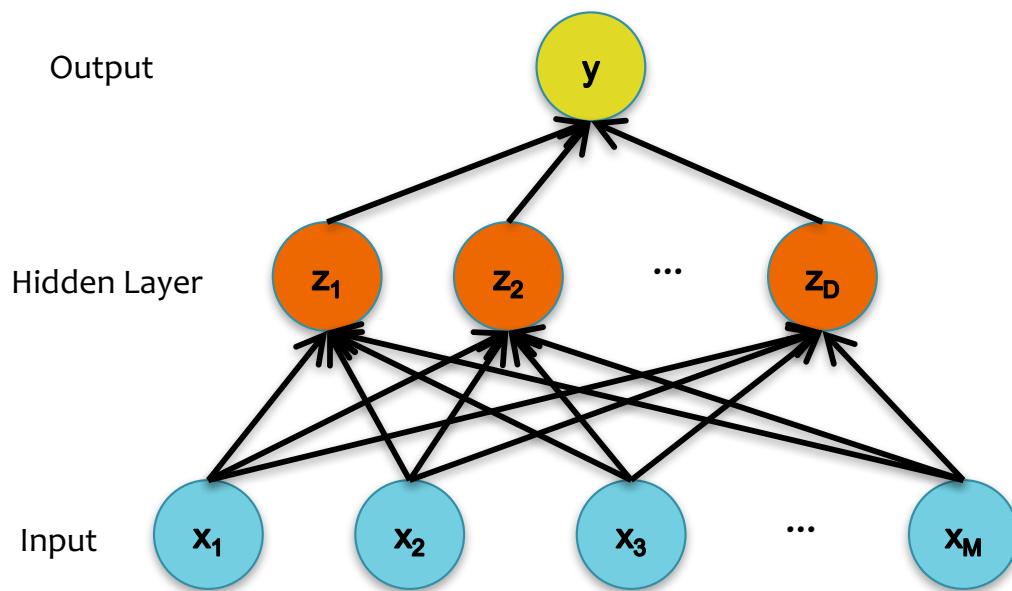
$$a_j = \sum_{i=0}^M \alpha_{ji} x_i, \forall j$$

(A) Input

Given  $x_i, \forall i$

# Activation Functions

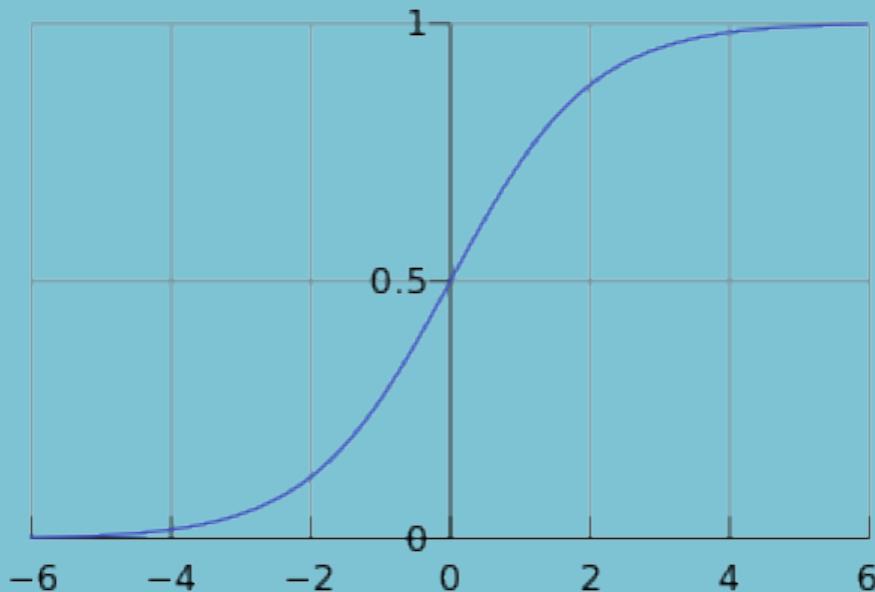
Neural Network with arbitrary nonlinear activation functions



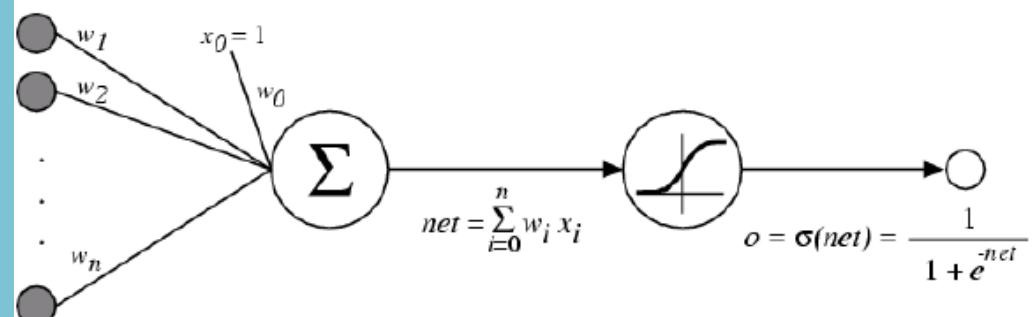
# Activation Functions

Sigmoid / Logistic Function

$$\text{logistic}(u) \equiv \frac{1}{1 + e^{-u}}$$

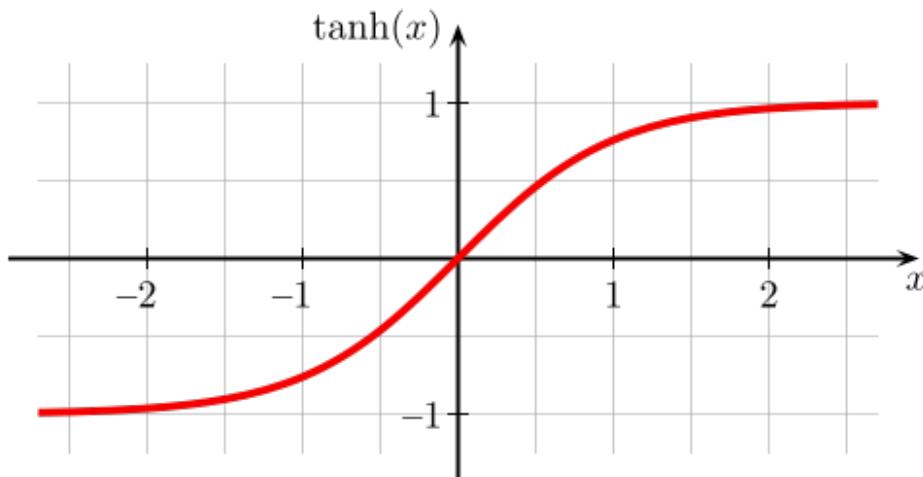


So far, we've assumed that the activation function (nonlinearity) is always the sigmoid function...



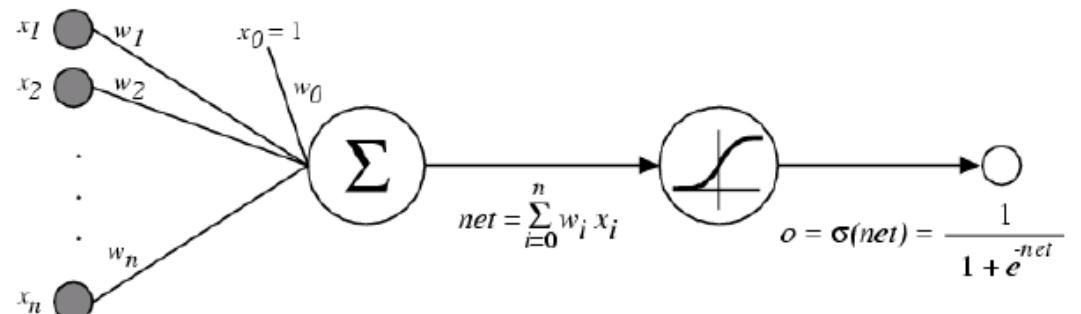
# Activation Functions

- A new change: modifying the nonlinearity
  - The logistic is not widely used in modern ANNs



Alternate 1:  
 $\tanh$

Like logistic function but  
shifted to range  $[-1, +1]$



# Understanding the difficulty of training deep feedforward neural networks

AI Stats 2010

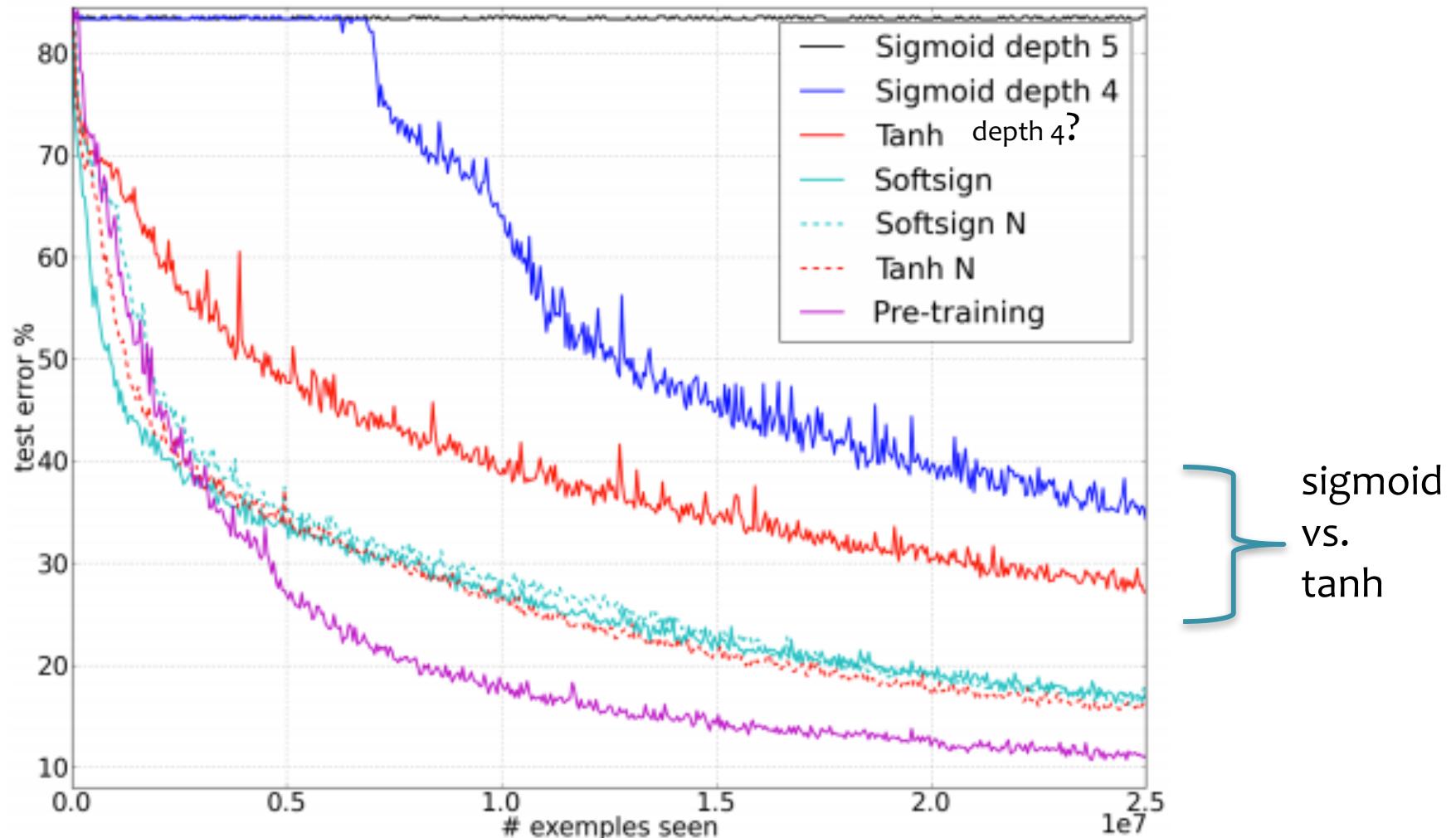
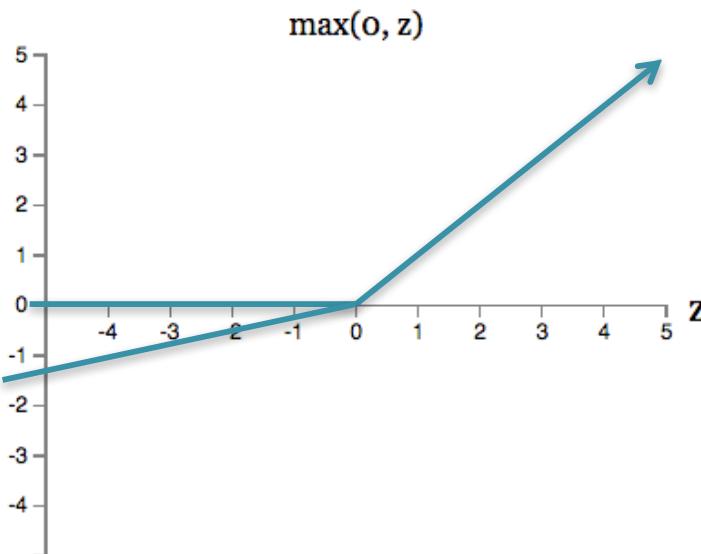


Figure from Glorot & Bentio (2010)

# Activation Functions

- A new change: modifying the nonlinearity
  - reLU often used in vision tasks

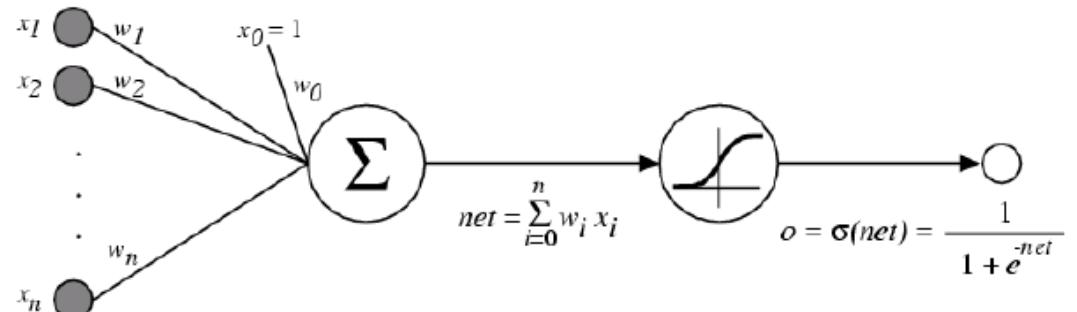


Alternate 2: rectified linear unit

Linear with a cutoff at zero

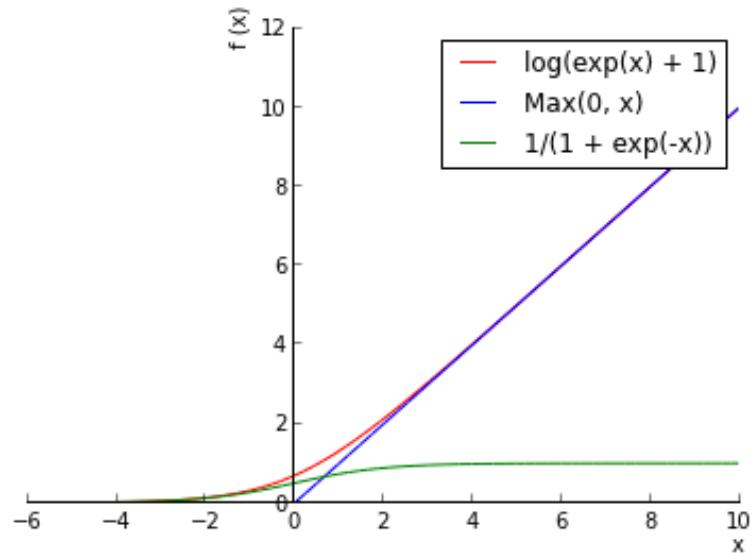
(Implementation: clip the gradient when you pass zero)

$$\max(0, w \cdot x + b).$$



# Activation Functions

- A new change: modifying the nonlinearity
  - reLU often used in vision tasks



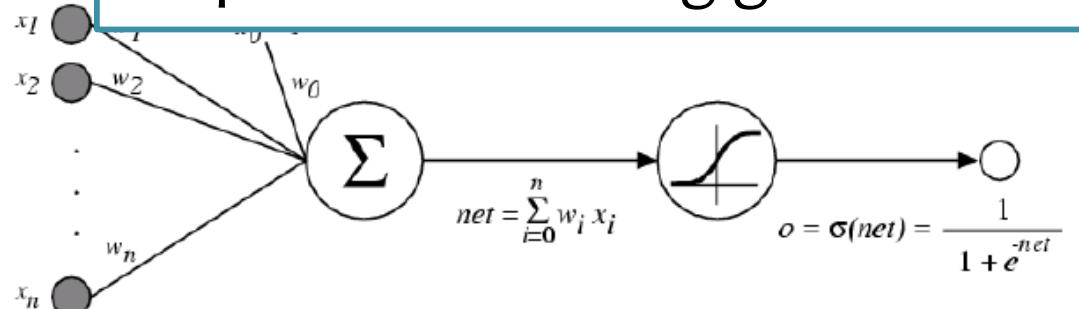
Alternate 2: rectified linear unit

Soft version:  $\log(\exp(x)+1)$

Doesn't saturate (at one end)

Sparsifies outputs

Helps with vanishing gradient



# Objective Functions for NNs

## 1. Quadratic Loss:

- the same objective as Linear Regression
- i.e. mean squared error

## 2. Cross-Entropy:

- the same objective as Logistic Regression
- i.e. negative log likelihood
- This requires probabilities, so we add an additional “softmax” layer at the end of our network

Forward

Quadratic     $J = \frac{1}{2}(y - y^*)^2$

Cross Entropy     $J = y^* \log(y) + (1 - y^*) \log(1 - y)$

Backward

$$\frac{dJ}{dy} = y - y^*$$

$$\frac{dJ}{dy} = y^* \frac{1}{y} + (1 - y^*) \frac{1}{1 - y}$$

# Objective Functions for NNs

Cross-entropy vs. Quadratic loss

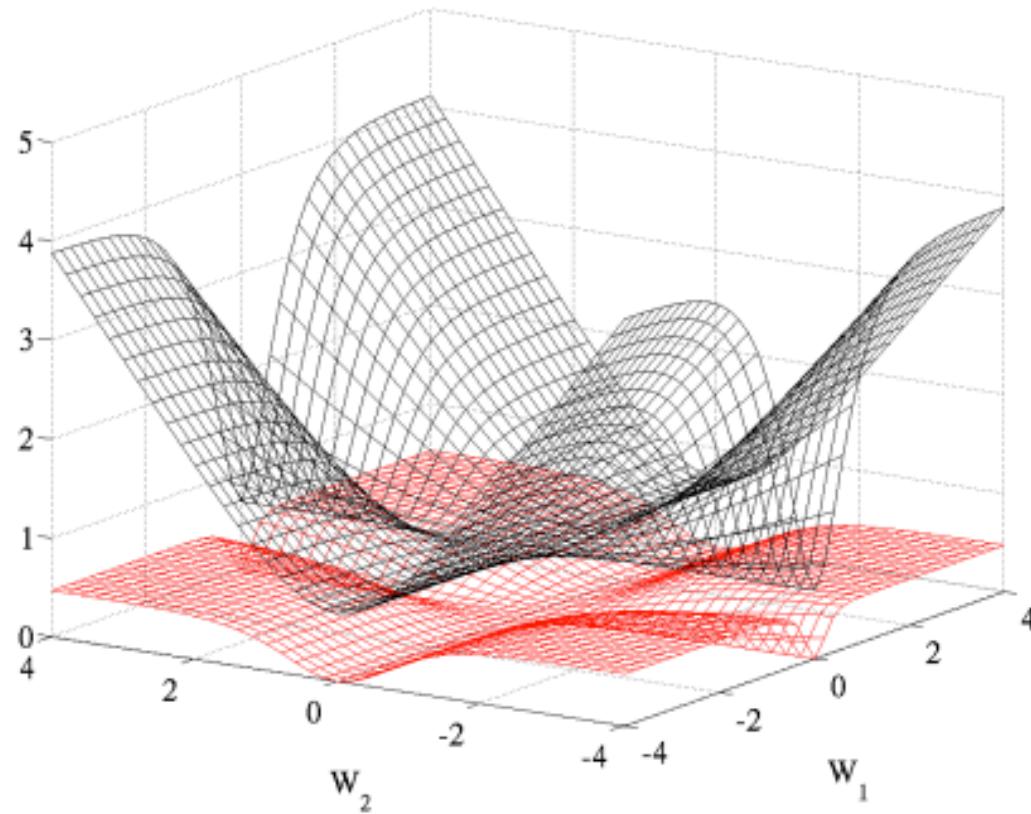
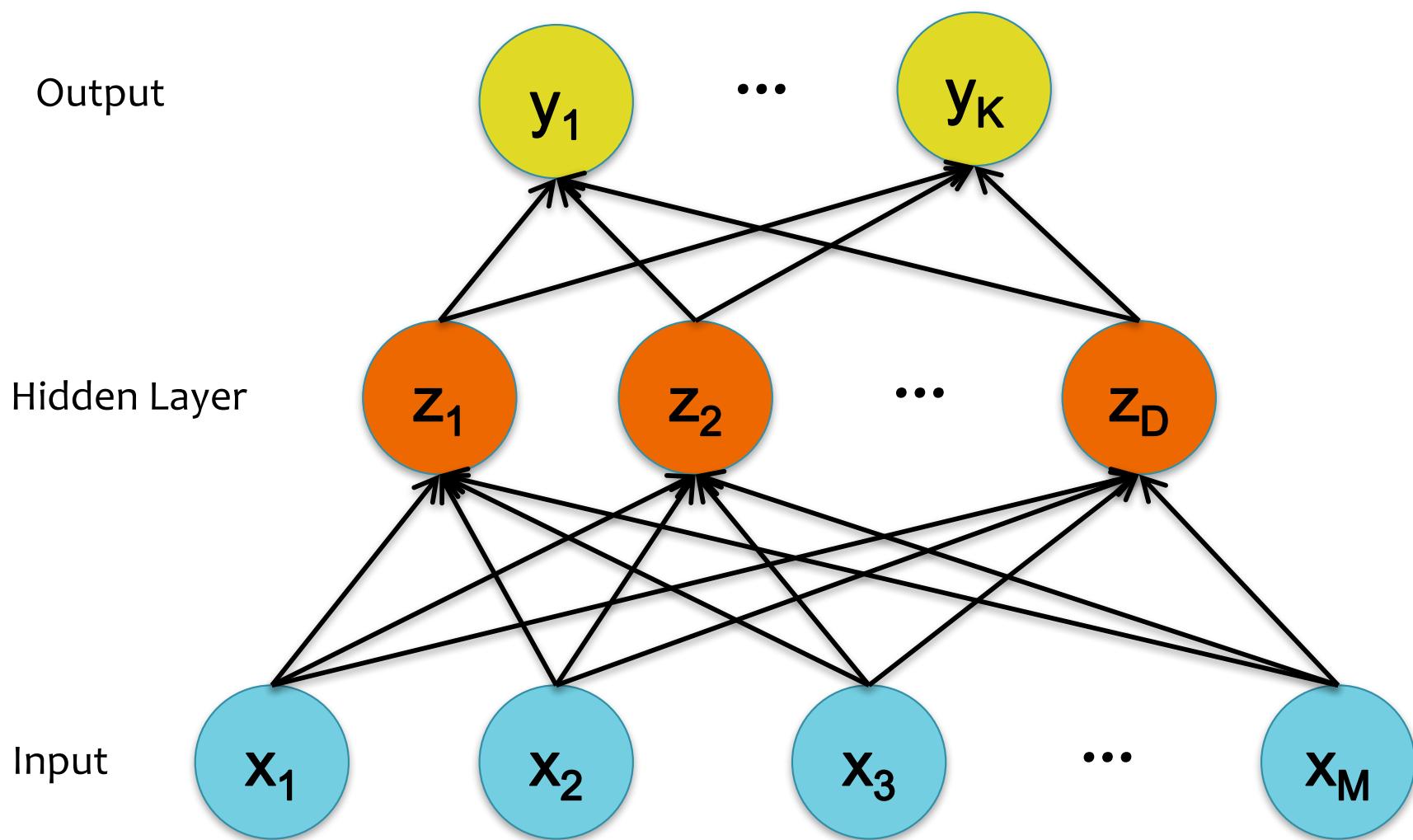


Figure 5: Cross entropy (black, surface on top) and quadratic (red, bottom surface) cost as a function of two weights (one at each layer) of a network with two layers,  $W_1$  respectively on the first layer and  $W_2$  on the second, output layer.

Figure from Glorot & Bentio (2010)

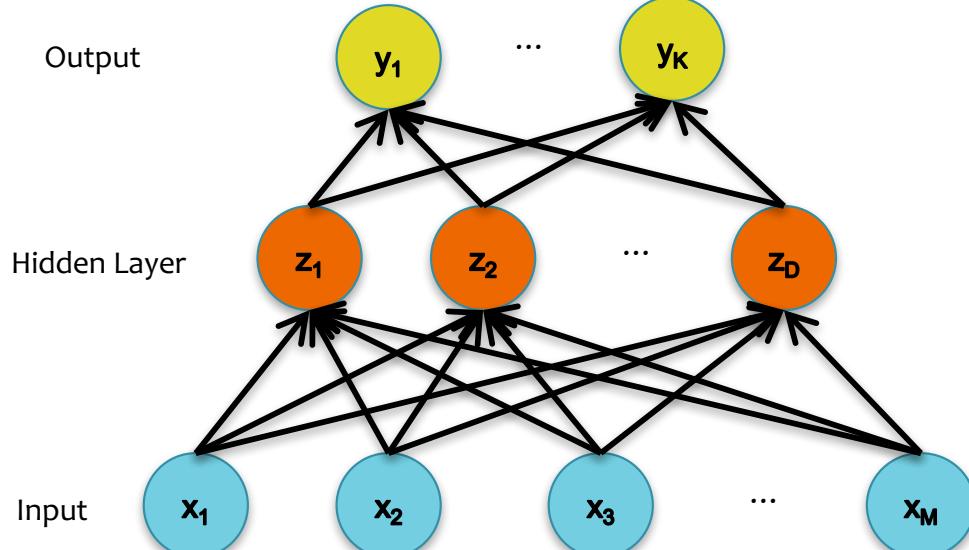
# Multi-Class Output



# Multi-Class Output

Softmax:

$$y_k = \frac{\exp(b_k)}{\sum_{l=1}^K \exp(b_l)}$$



(F) Loss

$$J = \sum_{k=1}^K y_k^* \log(y_k)$$

(E) Output (softmax)

$$y_k = \frac{\exp(b_k)}{\sum_{l=1}^K \exp(b_l)}$$

(D) Output (linear)

$$b_k = \sum_{j=0}^D \beta_{kj} z_j \quad \forall k$$

(C) Hidden (nonlinear)

$$z_j = \sigma(a_j), \quad \forall j$$

(B) Hidden (linear)

$$a_j = \sum_{i=0}^M \alpha_{ji} x_i, \quad \forall j$$

(A) Input

Given  $x_i, \quad \forall i$

# **BACKPROPAGATION**

## Background

# A Recipe for Machine Learning

1. Given training data:

$$\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^N$$

2. Choose each of these:

- Decision function

$$\hat{\mathbf{y}} = f_{\boldsymbol{\theta}}(\mathbf{x}_i)$$

- Loss function

$$\ell(\hat{\mathbf{y}}, \mathbf{y}_i) \in \mathbb{R}$$

3. Define goal:

$$\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} \sum_{i=1}^N \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), \mathbf{y}_i)$$

4. Train with SGD:

(take small steps  
opposite the gradient)

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta_t \nabla \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), \mathbf{y}_i)$$

- **Question 1:**  
When can we compute the gradients of the parameters of an arbitrary neural network?
- **Question 2:**  
When can we make the gradient computation efficient?

# Approaches to Differentiation

1. Finite Difference Method
  - Pro: Great for testing implementations of backpropagation
  - Con: Slow for high dimensional inputs / outputs
  - Required: Ability to call the function  $f(\mathbf{x})$  on any input  $\mathbf{x}$
2. Symbolic Differentiation
  - Note: The method you learned in high-school
  - Note: Used by Mathematica / Wolfram Alpha / Maple
  - Pro: Yields easily interpretable derivatives
  - Con: Leads to exponential computation time if not carefully implemented
  - Required: Mathematical expression that defines  $f(\mathbf{x})$
3. Automatic Differentiation - Reverse Mode
  - Note: Called Backpropagation when applied to Neural Nets
  - Pro: Computes partial derivatives of one output  $f(\mathbf{x})_i$  with respect to all inputs  $x_j$  in time proportional to computation of  $f(\mathbf{x})$
  - Con: Slow for high dimensional outputs (e.g. vector-valued functions)
  - Required: Algorithm for computing  $f(\mathbf{x})$
4. Automatic Differentiation - Forward Mode
  - Note: Easy to implement. Uses dual numbers.
  - Pro: Computes partial derivatives of all outputs  $f(\mathbf{x})_i$  with respect to one input  $x_j$  in time proportional to computation of  $f(\mathbf{x})$
  - Con: Slow for high dimensional inputs (e.g. vector-valued  $\mathbf{x}$ )
  - Required: Algorithm for computing  $f(\mathbf{x})$

Given  $f : \mathbb{R}^A \rightarrow \mathbb{R}^B, f(\mathbf{x})$   
Compute  $\frac{\partial f(\mathbf{x})_i}{\partial x_j} \forall i, j$

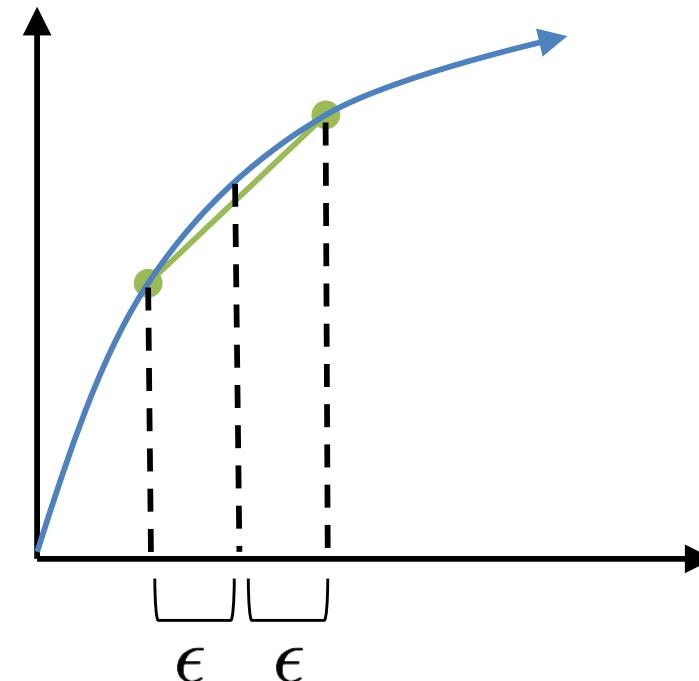
The centered finite difference approximation is:

$$\frac{\partial}{\partial \theta_i} J(\boldsymbol{\theta}) \approx \frac{(J(\boldsymbol{\theta} + \epsilon \cdot \mathbf{d}_i) - J(\boldsymbol{\theta} - \epsilon \cdot \mathbf{d}_i))}{2\epsilon} \quad (1)$$

where  $\mathbf{d}_i$  is a 1-hot vector consisting of all zeros except for the  $i$ th entry of  $\mathbf{d}_i$ , which has value 1.

#### Notes:

- Suffers from issues of floating point precision, in practice
- Typically only appropriate to use on small examples with an appropriately chosen epsilon



**Chain Rule Quiz #1:**

Suppose  $x = 2$  and  $z = 3$ , what are  $dy/dx$  and  $dy/dz$  for the function below?

$$y = \exp(xz) + \frac{xz}{\log(x)} + \frac{\sin(\log(x))}{\exp(xz)}$$

## Calculus Quiz #2:

A neural network with 2 hidden layers can be written as:

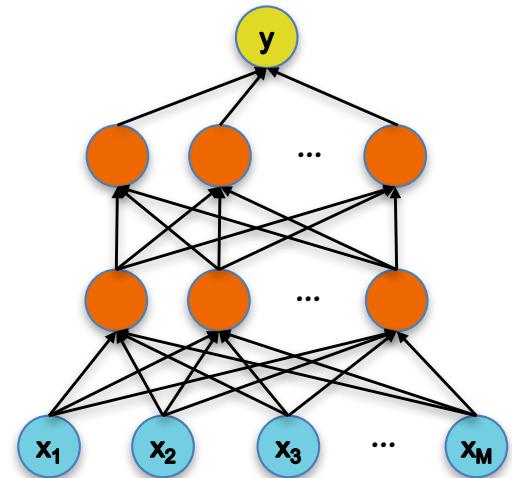
$$y = \sigma(\beta^T \sigma((\alpha^{(2)})^T \sigma((\alpha^{(1)})^T \mathbf{x})))$$

where  $y \in \mathbb{R}$ ,  $\mathbf{x} \in \mathbb{R}^{D^{(0)}}$ ,  $\beta \in \mathbb{R}^{D^{(2)}}$  and  $\alpha^{(i)}$  is a  $D^{(i)} \times D^{(i-1)}$  matrix. Nonlinear functions are applied elementwise:

$$\sigma(\mathbf{a}) = [\sigma(a_1), \dots, \sigma(a_K)]^T$$

Let  $\sigma$  be sigmoid:  $\sigma(a) = \frac{1}{1+exp-a}$

What is  $\frac{\partial y}{\partial \beta_j}$  and  $\frac{\partial y}{\partial \alpha_j^{(i)}}$  for all  $i, j$ .



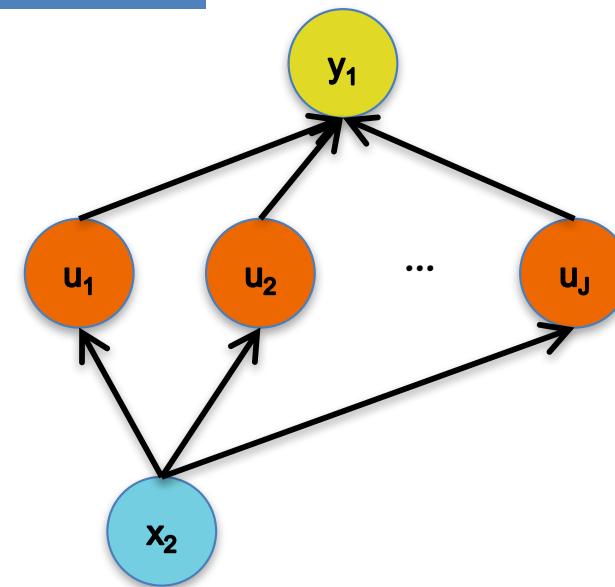
## *Whiteboard*

– Chain Rule of Calculus

**Given:**  $y = g(u)$  and  $u = h(x)$ .

**Chain Rule:**

$$\frac{dy_i}{dx_k} = \sum_{j=1}^J \frac{dy_i}{du_j} \frac{du_j}{dx_k}, \quad \forall i, k$$



# Training

# Chain Rule

**Given:**  $y = g(u)$  and  $u = h(x)$ .

**Chain Rule:**

$$\frac{dy_i}{dx_k} = \sum_{j=1}^J \frac{dy_i}{du_j} \frac{du_j}{dx_k}, \quad \forall i, k$$

**Backpropagation**  
is just repeated  
application of the  
**chain rule** from  
Calculus 101.

