# Artificial Neural Networks

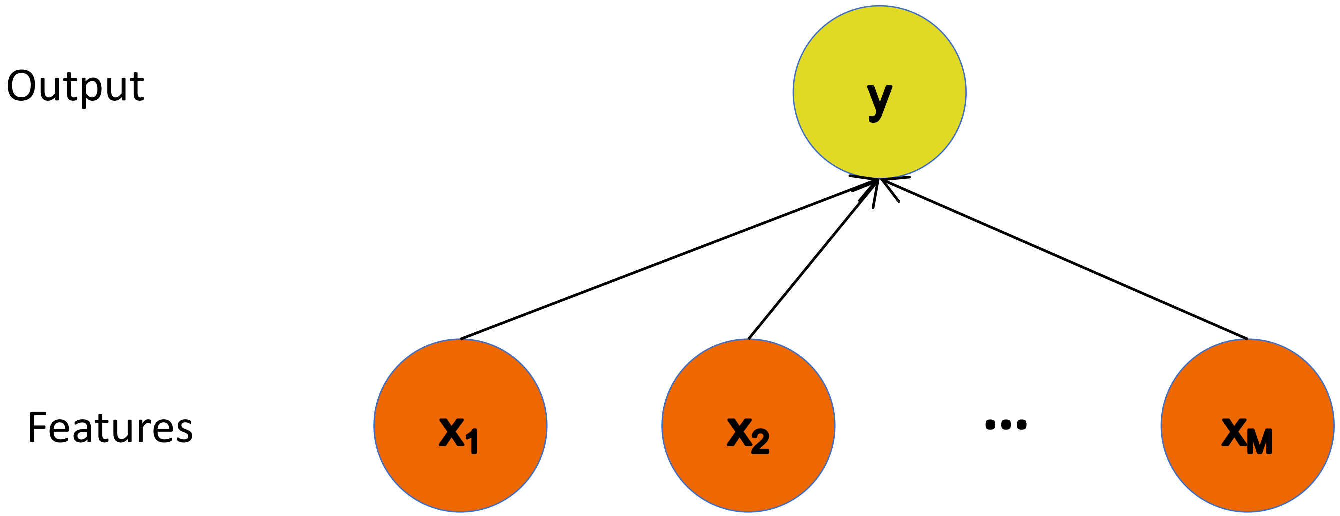
# Handling Non-Linear Data

- Option 1: Add features by hand that make the data separable
  - Requires feature engineering
- Option 2: Learn a small number of additional features that will suffice
  - Today
- Option 3: Kernel trick

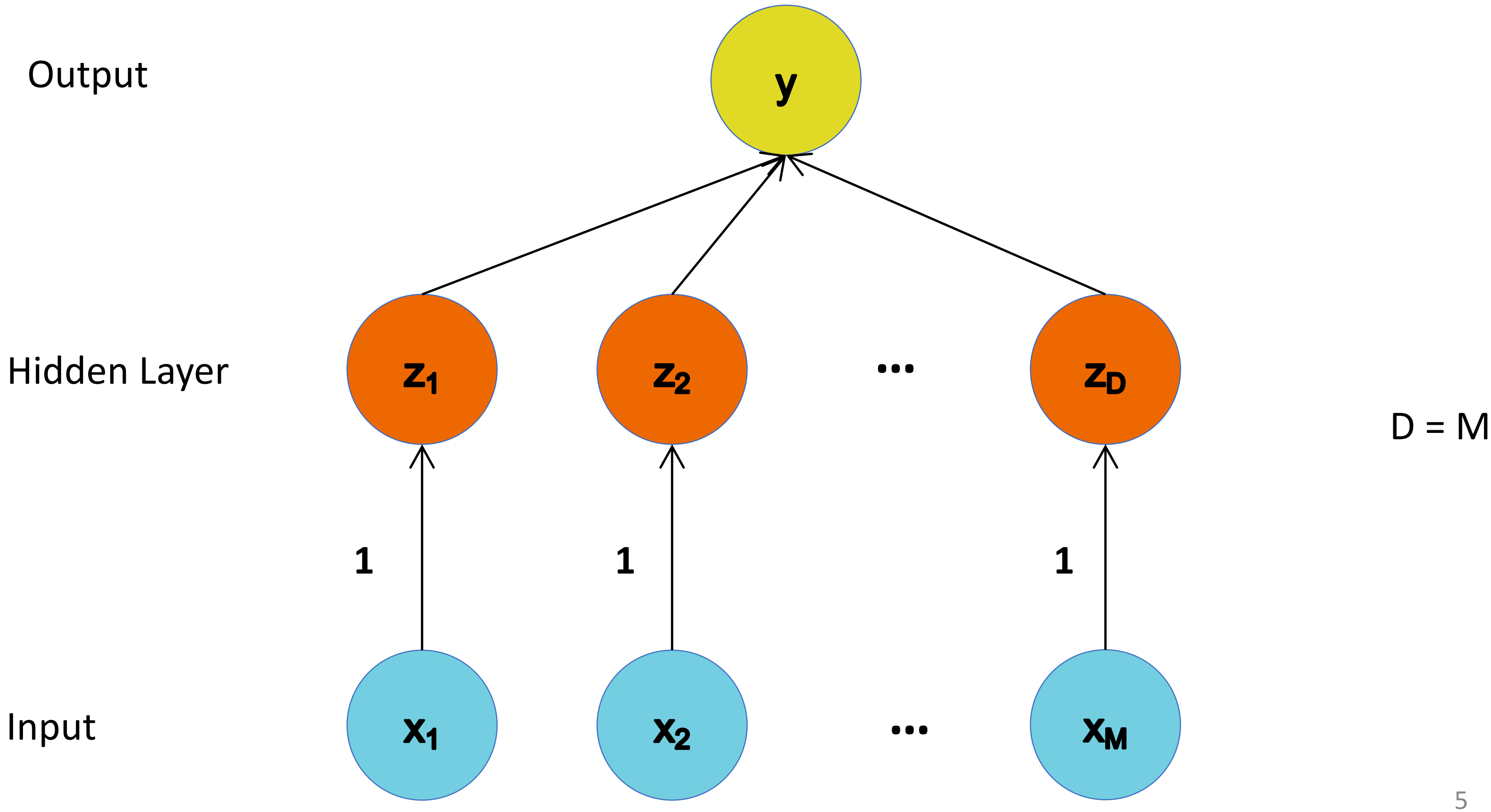
# Motivation

- Where do features come from?
  - We build them by hand
- What if we wanted to learn features?
  - Goal: learn features that give linearly separable data
- After learning features apply usual linear classifier

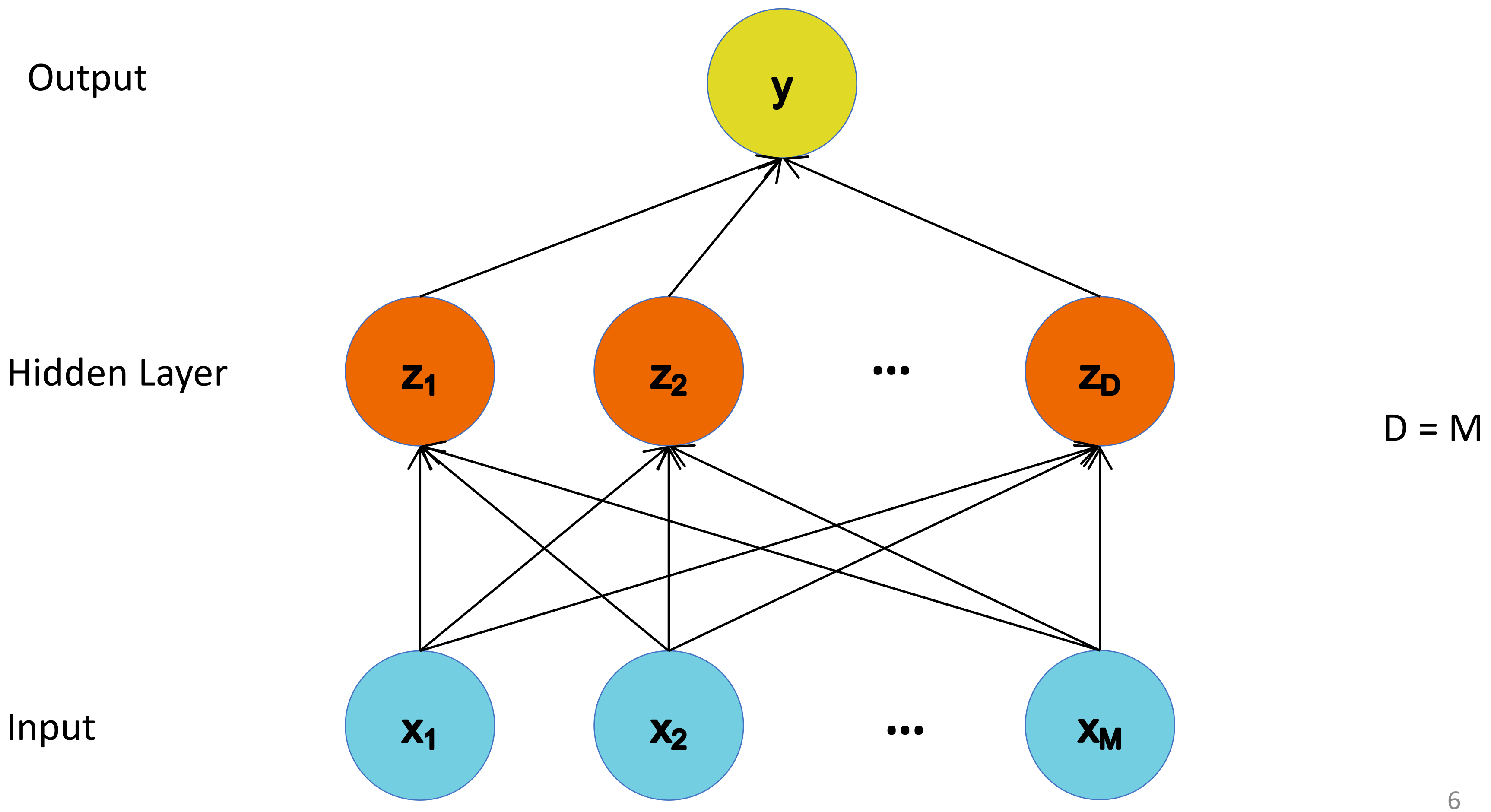
# Perceptron: Graphical Representation



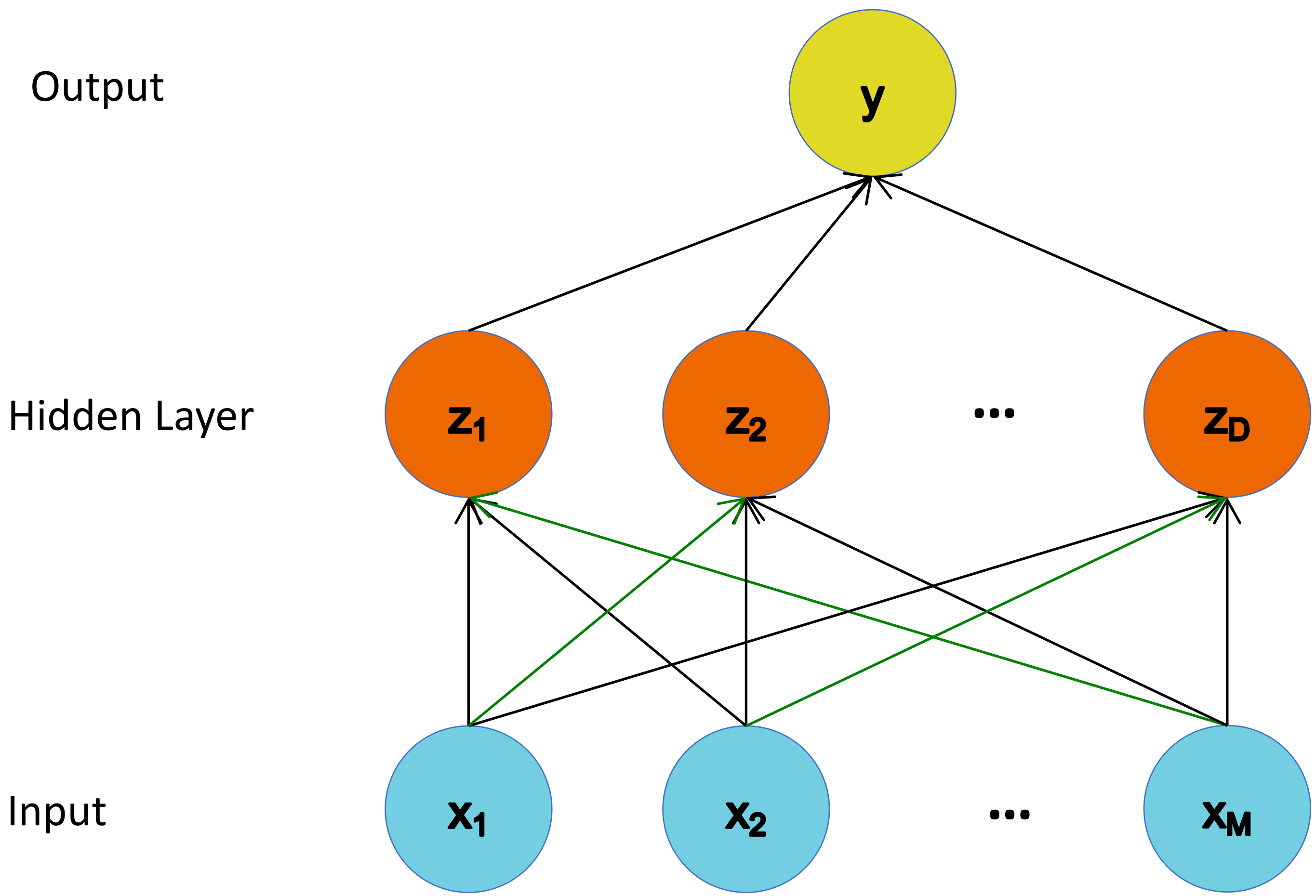
# Perceptron: Graphical Representation



# Perceptron: Graphical Representation

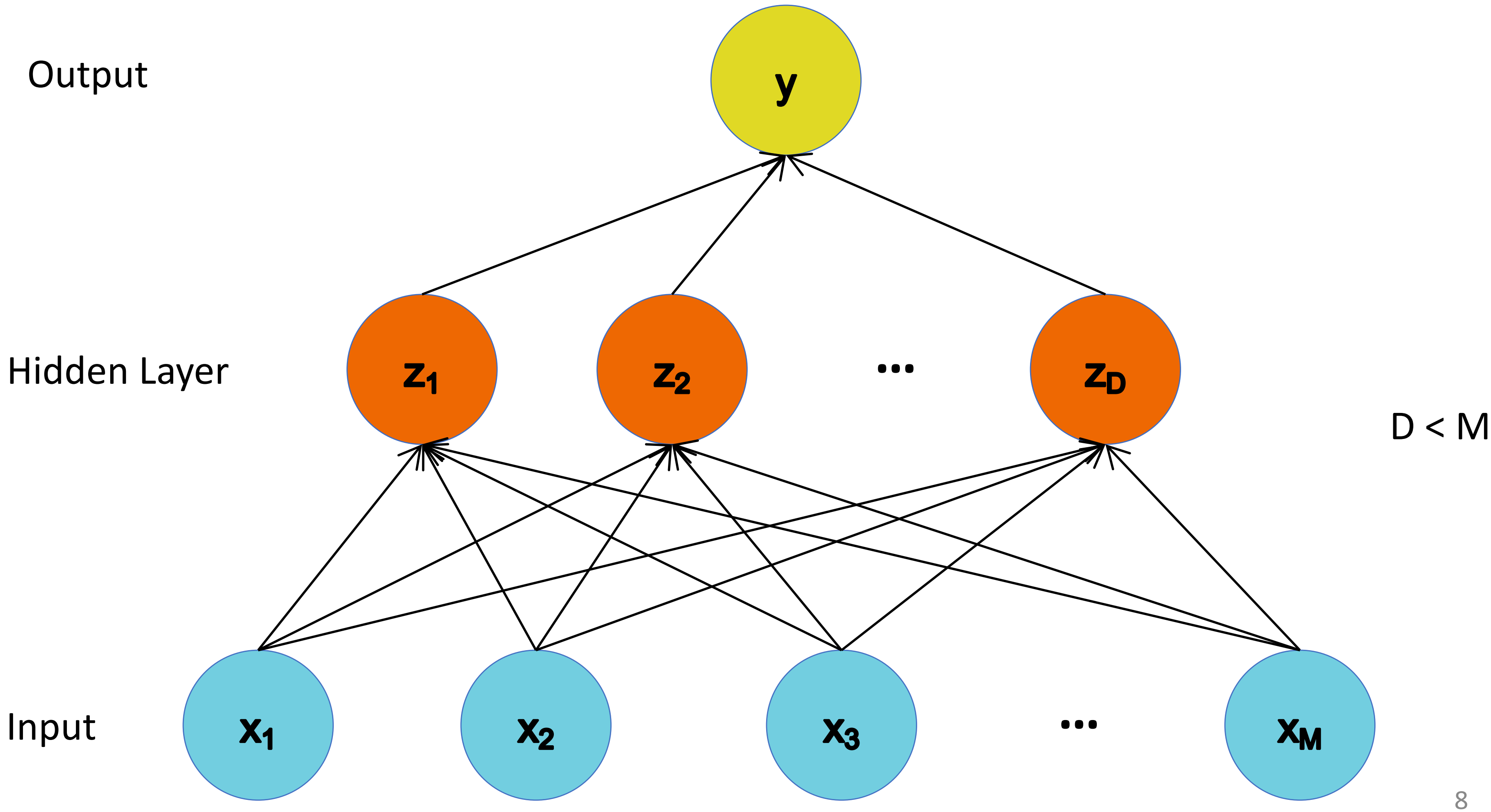


# Perceptron: Graphical Representation




D = M

# Perceptron: Graphical Representation

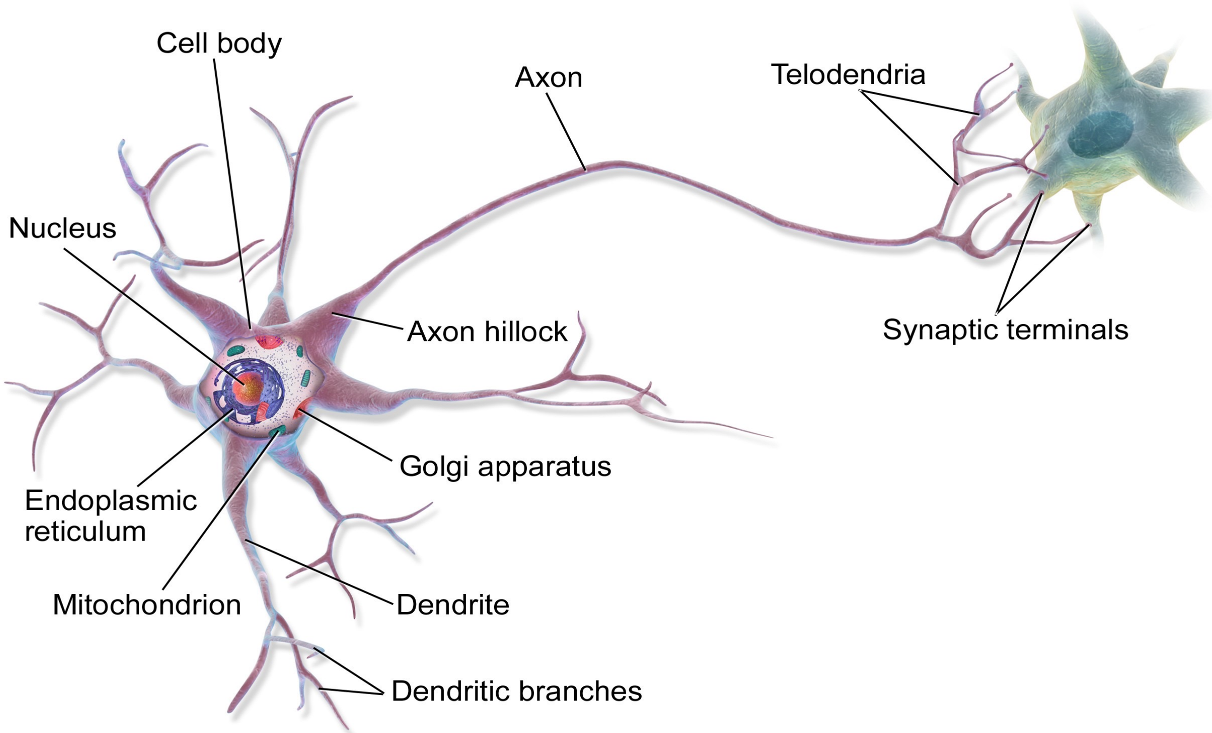


# Why?

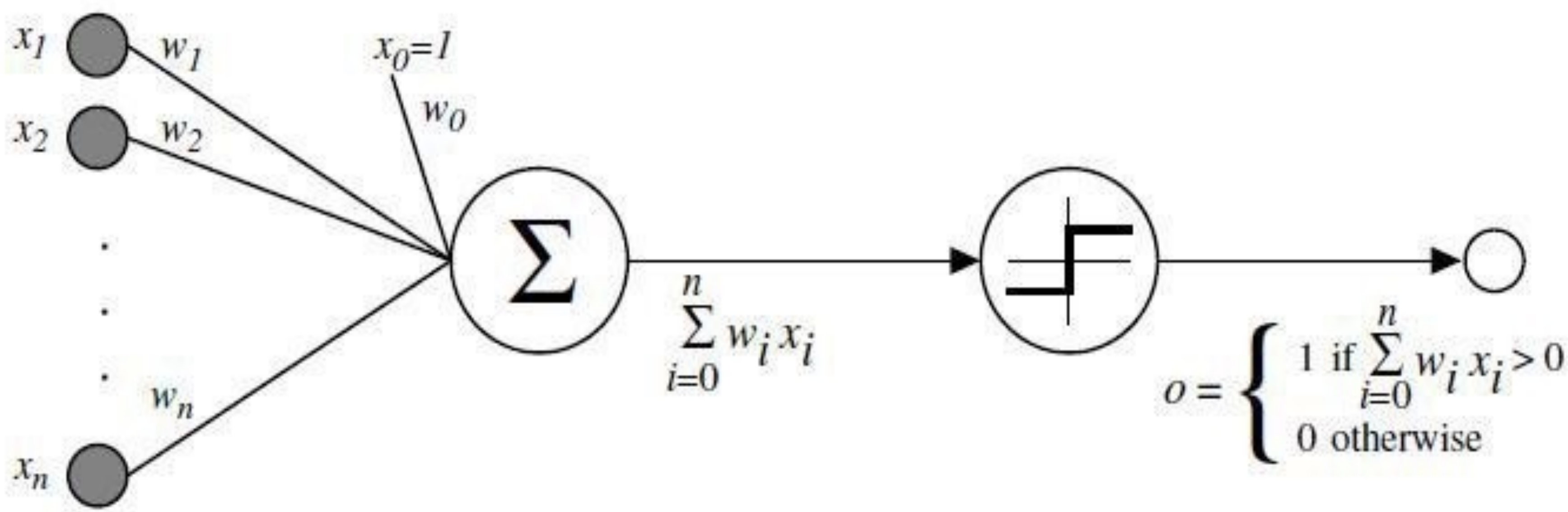
- Constraints from X
  - When D = M, likely to copy the features from X to Z
  - When D < M, cannot make an exact copy of X
    - Must come up with a representation that is more efficient
- Constraints from Y
  - Z should be a representation that helps learn Y
  - Forces the low-dimensional representation to capture properties of X useful in predicting Y

# Why Non-Linear

- Generalized linear classifiers!
  - Start with linear function
    - $w \cdot x$
  - Pass the output through a non-linear function
    - $\hat{y} = h(w \cdot x)$
  - What is $h$?
    - Non-linear function
      - Logistic function
      - Sign function
- Each Z is the output of a non-linear function
  - Combinations of Z are now non-linear in X



Cell body
Axon
Telodendria
Nucleus
Axon hillock
Synaptic terminals
Golgi apparatus
Endoplasmic reticulum
Mitochondrion
Dendrite
Dendritic branches

$x_1$
$w_1$
$x_0=1$
$w_0$
$x_2$
$w_2$
$w_n$
$x_n$
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ 0 \text{ otherwise} \end{cases}$

# Multi-Layer Perceptrons

- **Fitting a function to data**

- **Fitting: what type of optimization algorithm?**
- Function: non-linear: linear combination of generalized linear functions
- Data: Data/model assumptions? How we use data?

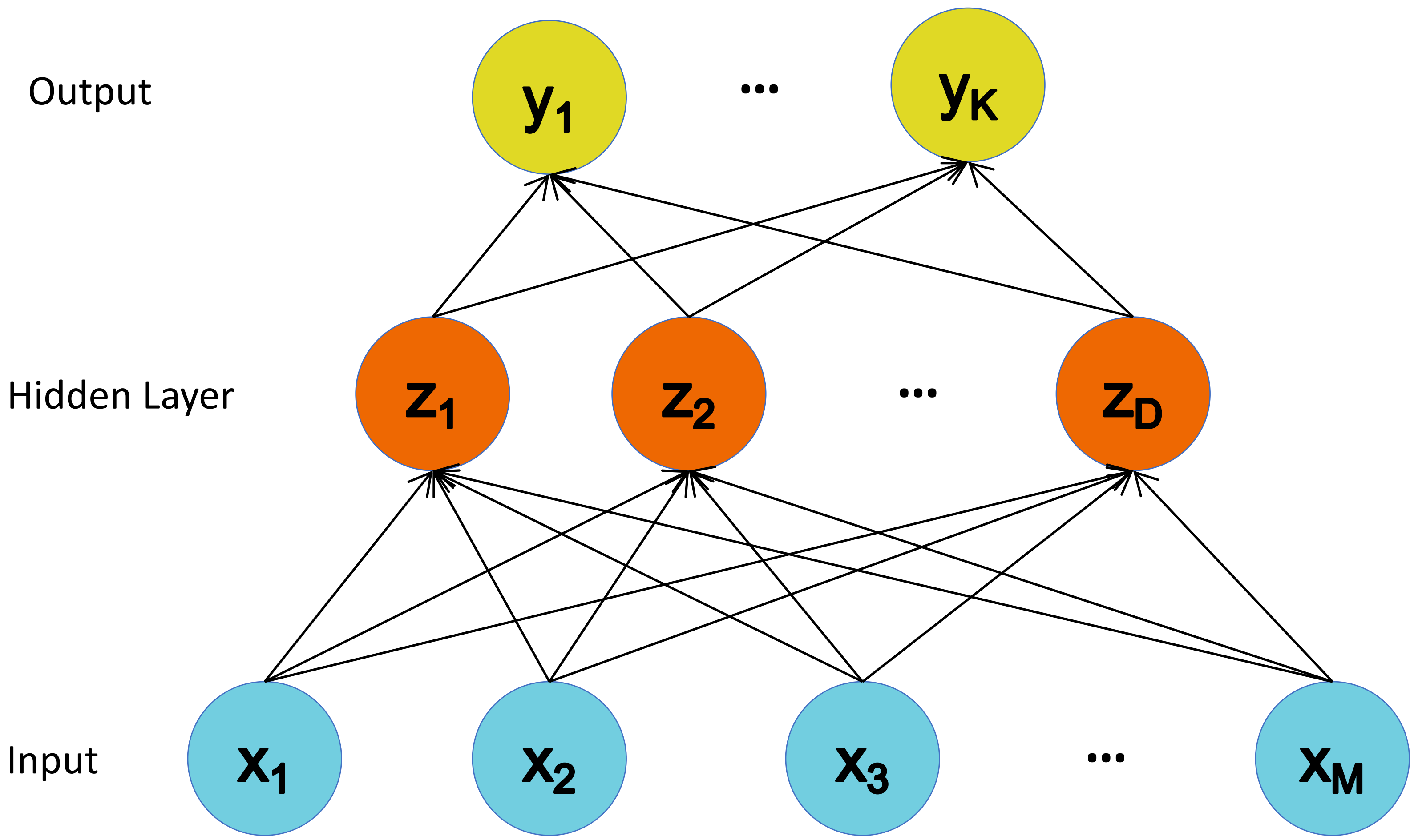
# How Will We Learn?

- Perceptron: a training method for generalized linear classifiers
  - Training method for linear classifiers
  - Minimize the error of the training data
  - Chain multiple Perceptrons together
  - Update rule:

$$w^{i+1} = w^i + \nabla f(x, y)$$

  - The real work will be in computing the gradient

# Multi-Class Output

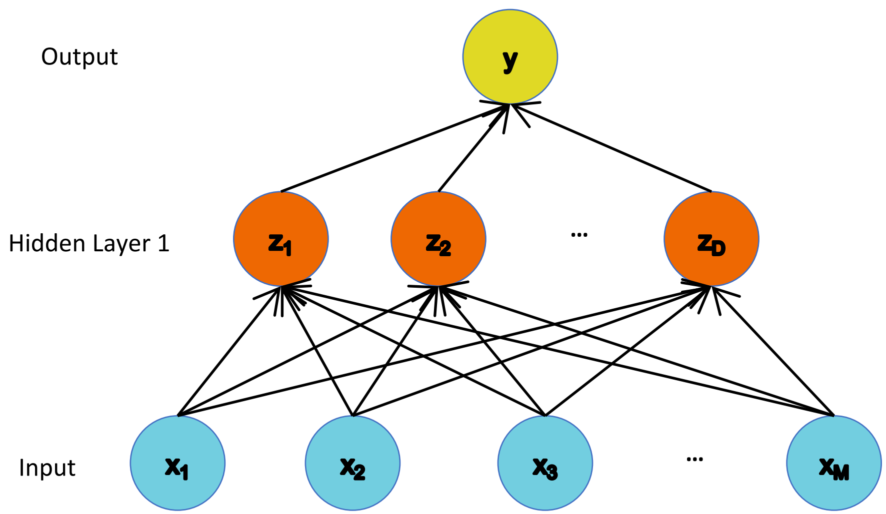


# Network Terminology

- Input nodes: x
- Output node: y
- Hidden nodes: z
  - This network has 1 hidden layer
  - 2 layer network (two layers to learn)
- h for hidden nodes are called activation functions
- h for output depends on task
  - Identity for regression
  - Logistic for classification

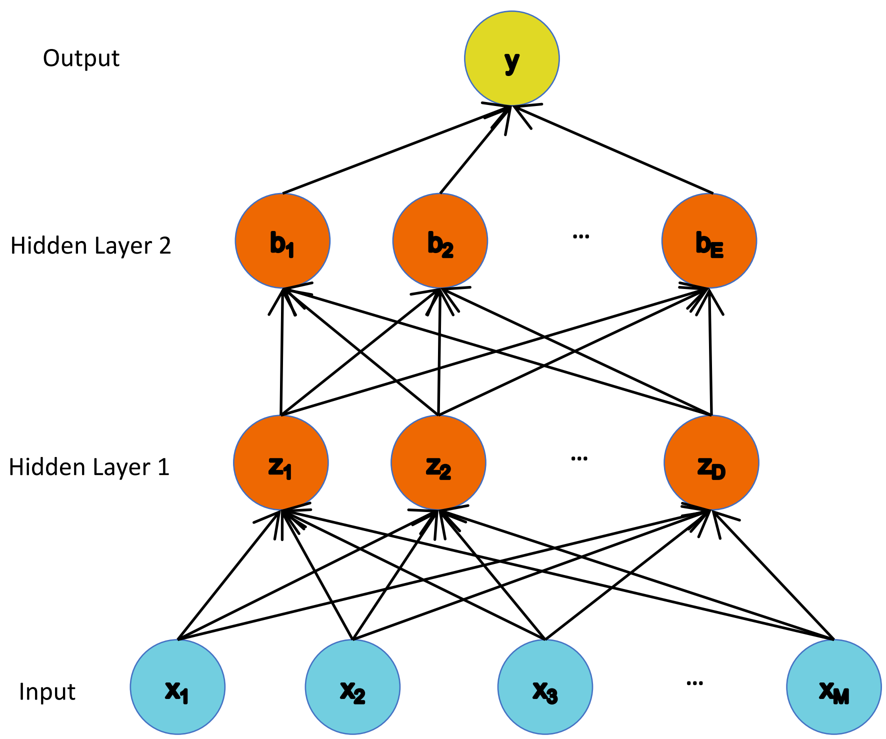
# Deeper Networks

Next lecture:



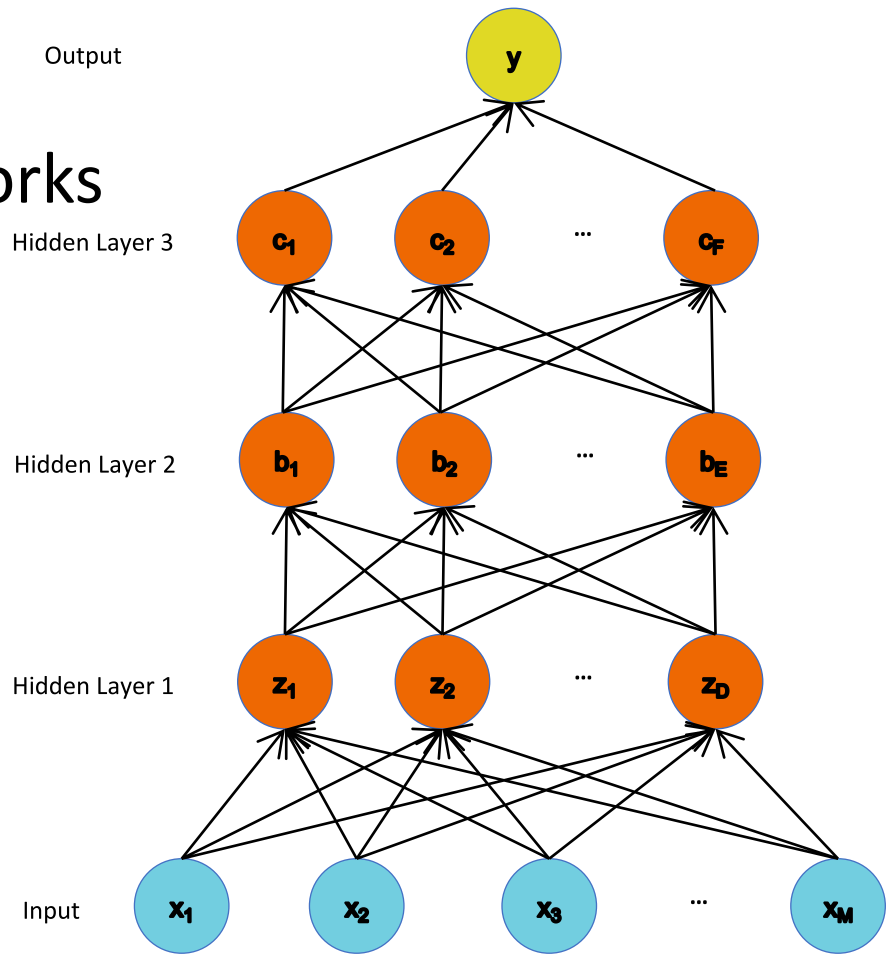
# Deeper Networks

Next lecture:



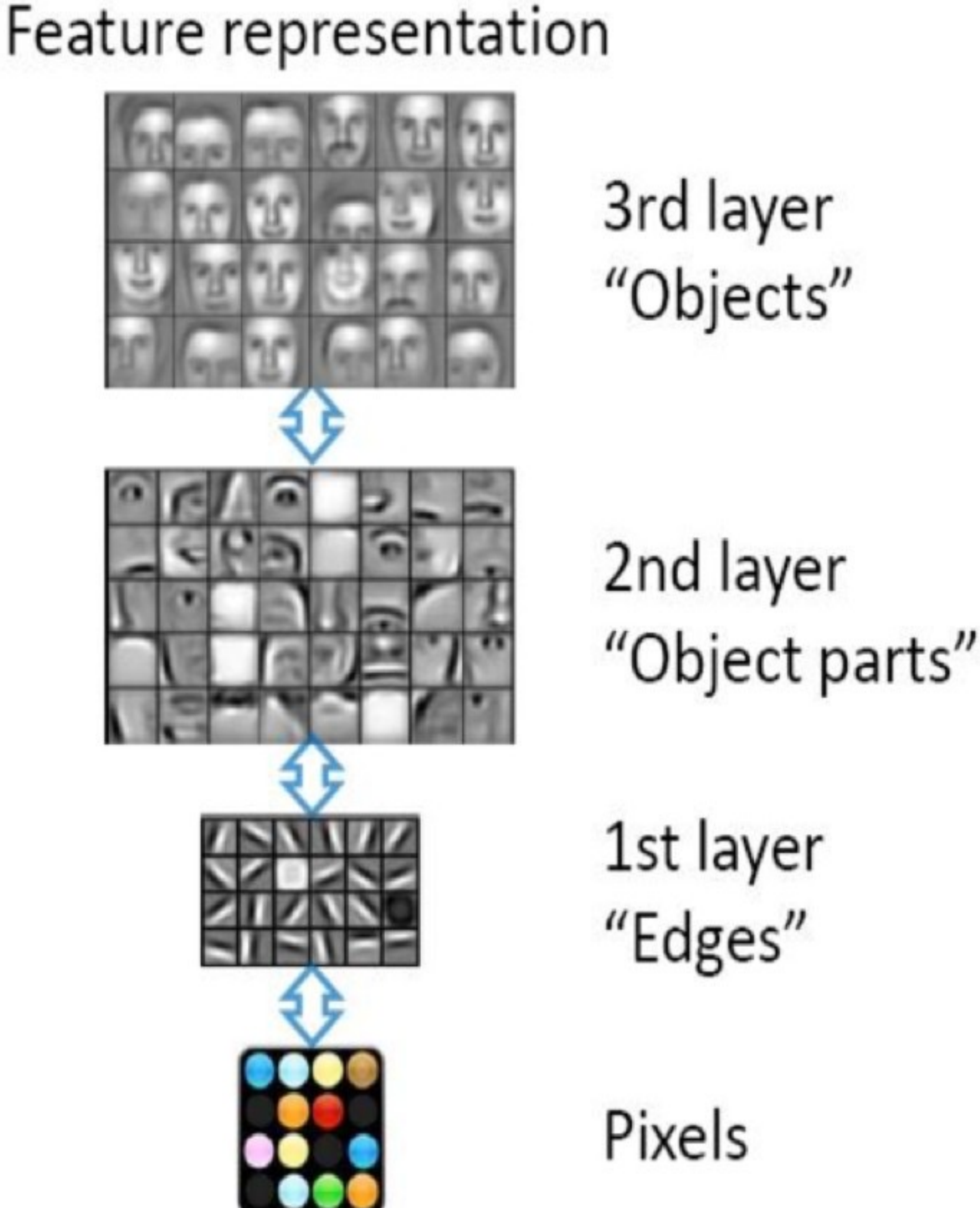
# Deeper Networks

Next lecture: Making the neural networks deeper



# Deep Networks

- Learn multiple levels of features at higher and higher abstractions
- Same learning techniques
  - Just more complex gradients




- Example: Image Processing

# Architectures

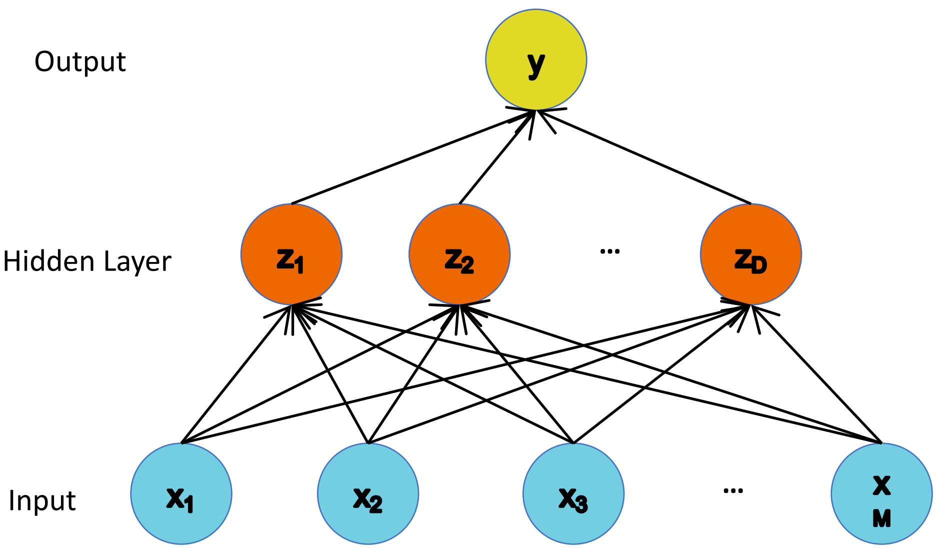
# Neural Network Architectures

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

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

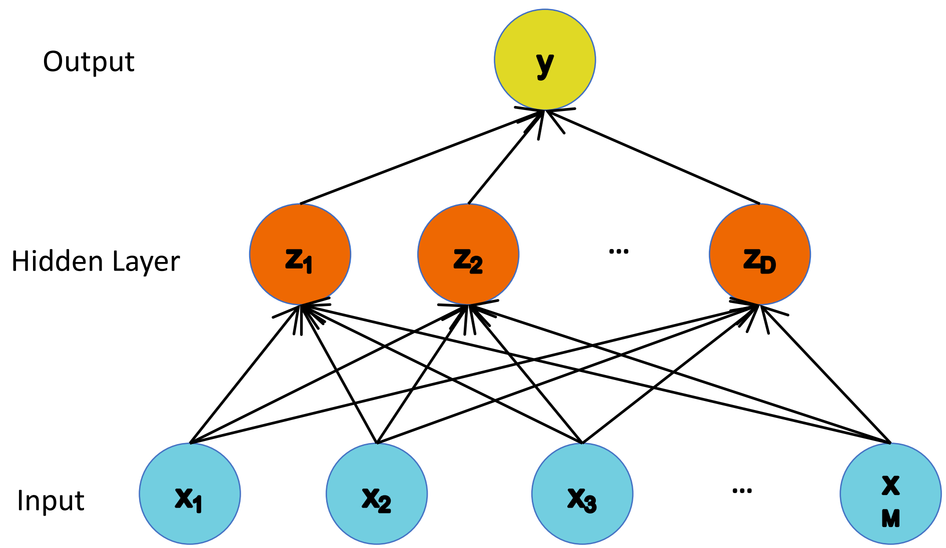
**(C) Hidden (sigmoid)**
$z_j = \frac{1}{1+\exp(-a_j)}, \ \forall j$

**(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




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

**(E) Output (nonlinear)**
$y = \sigma(b)$

**(D) Output (linear)**
$b = \sum_{j=0}^{D} \beta_j z_j$

**(C) Hidden (nonlinear)**
$z_j = \sigma(a_j), \ \forall j$

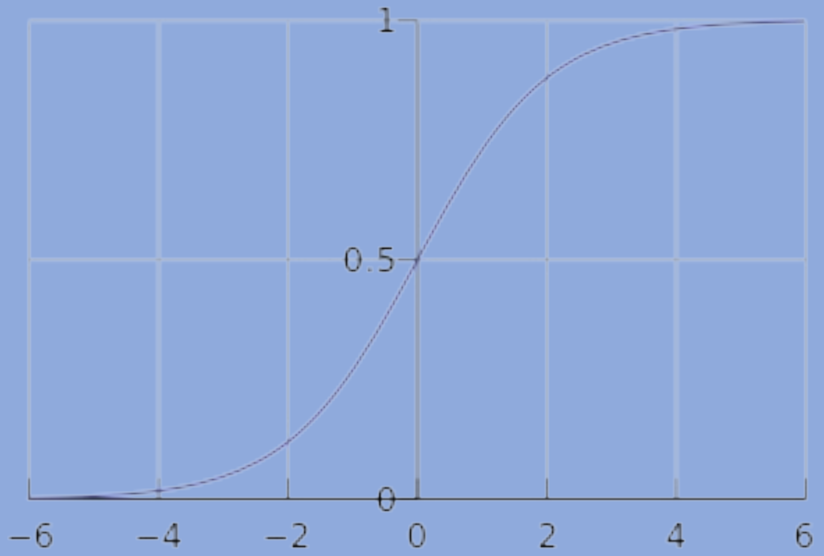
**(B) Hidden (linear)**
$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i, \ \forall j$

**(A) Input**
Given $x_i, \ \forall i$

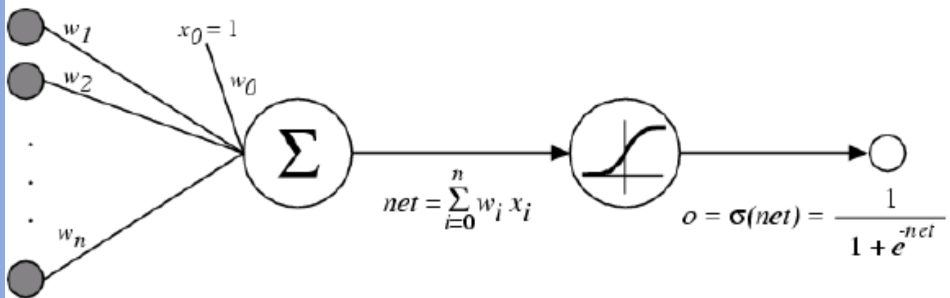
# Activation Functions

## Sigmoid / Logistic Function

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

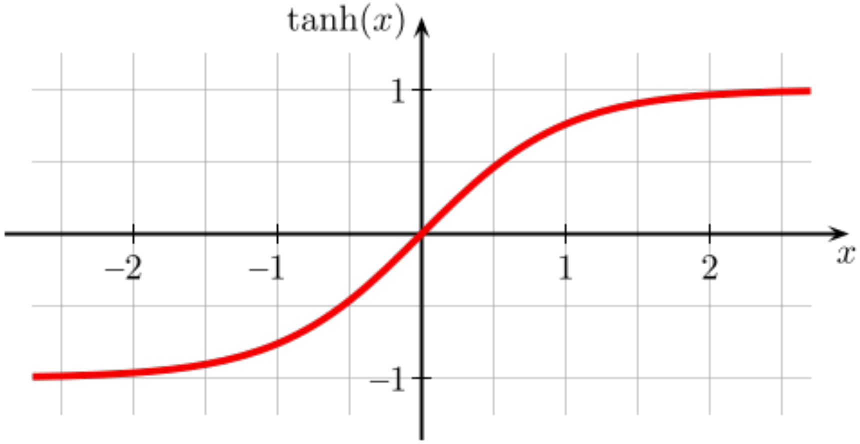



So far, we've assumed that the activation function (nonlinearity) is always the sigmoid function…



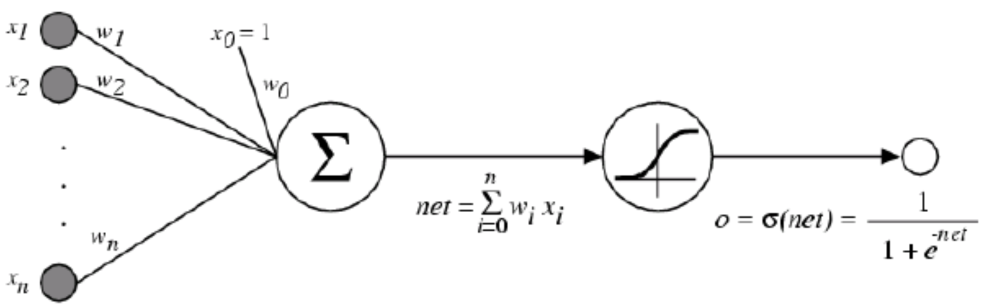
# Activation Functions

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




Alternate 1: tanh

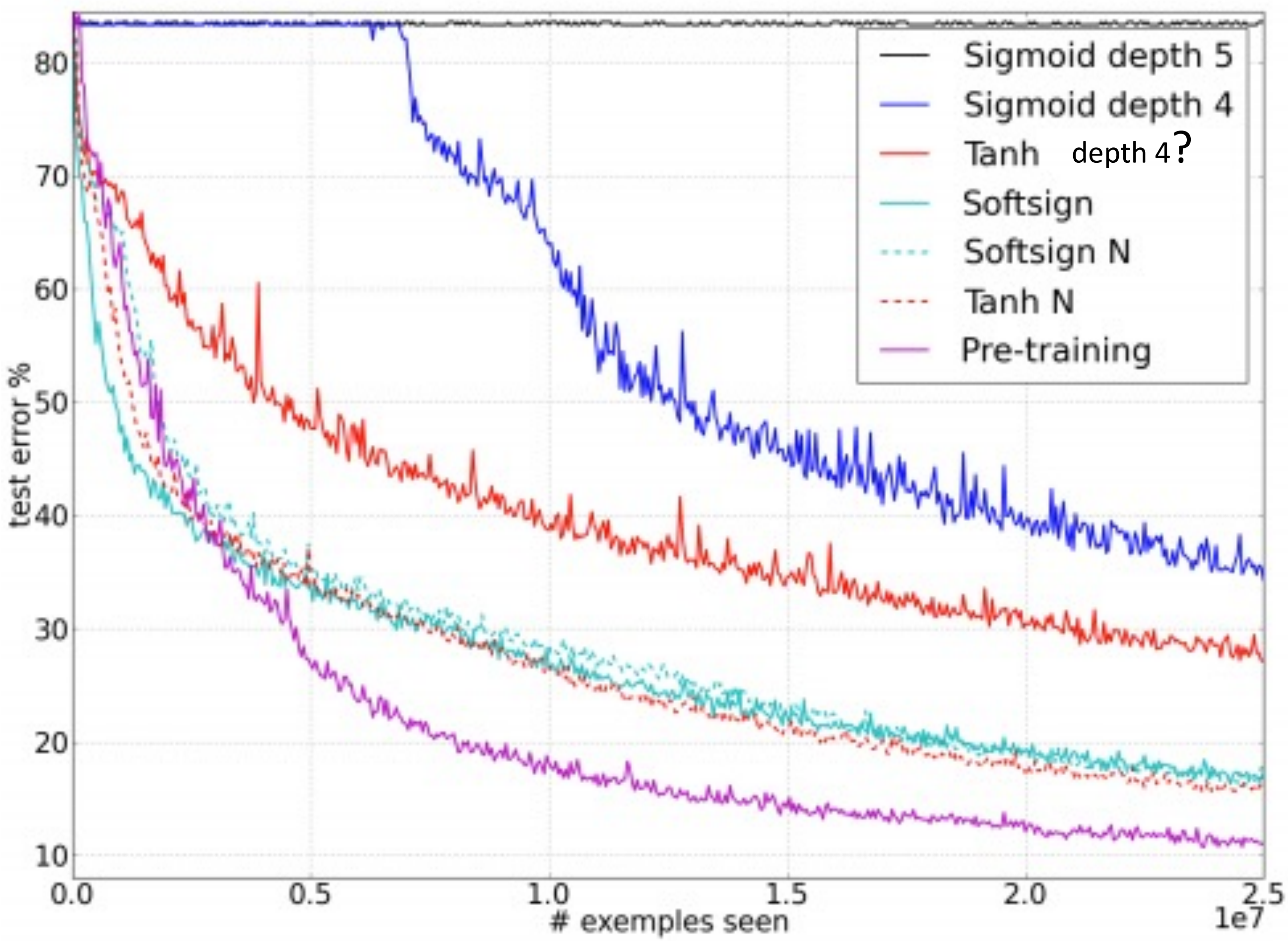
Like logistic function but shifted to range [-1, +1]




Slide from William Cohen

# Understanding the difficulty of training deep feedforward neural networks

AI Stats 2010

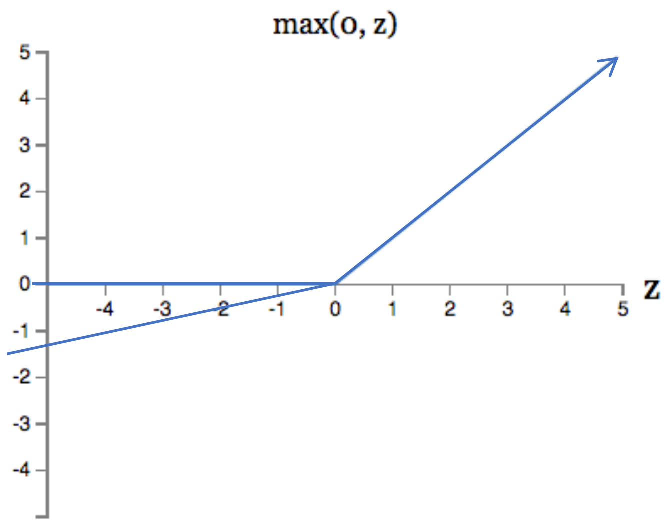



sigmoid
vs.
tanh

Figure from Glorot & Bentio (2010)

# Activation Functions

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

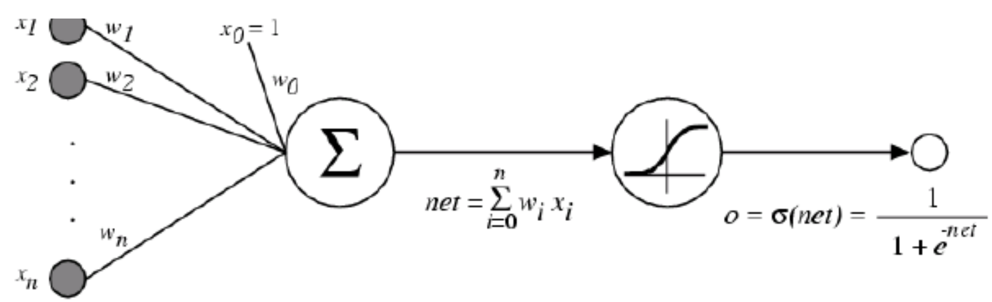



Alternate 2: rectified linear unit

Linear with a cutoff at zero

(Implementation: clip the gradient when you pass zero)

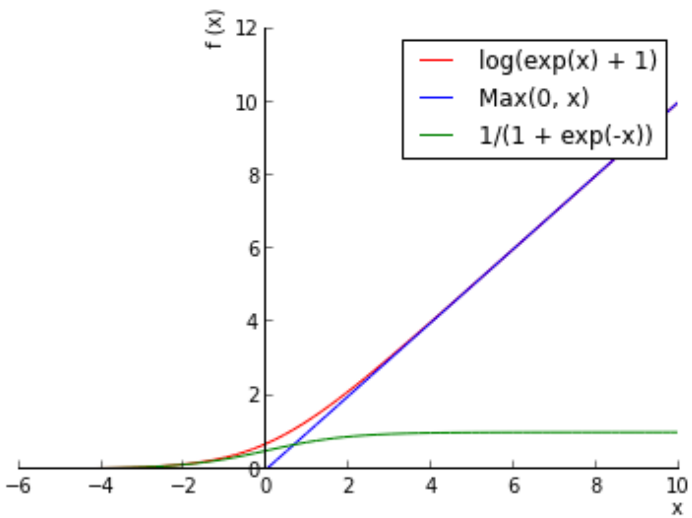
$$\max(0, w \cdot x + b).$$




Slide from William Cohen

# Activation Functions

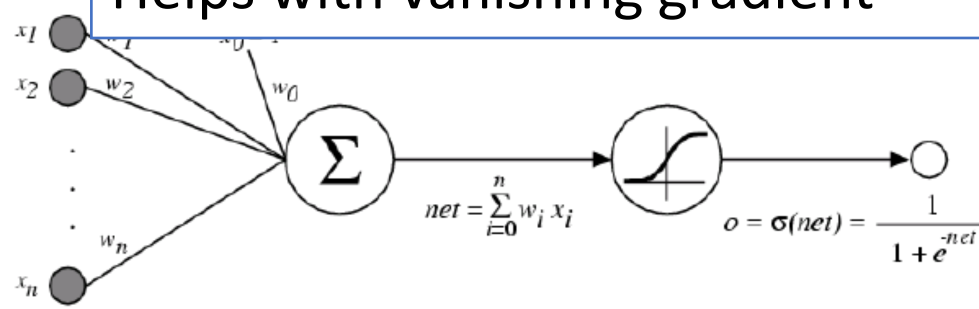
- A new change: modifying the nonlinearity
  - reLU often used in vision tasks




Alternate 2: rectified linear unit

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

Doesn't saturate (at one end)
Sparsifies outputs
Helps with vanishing gradient



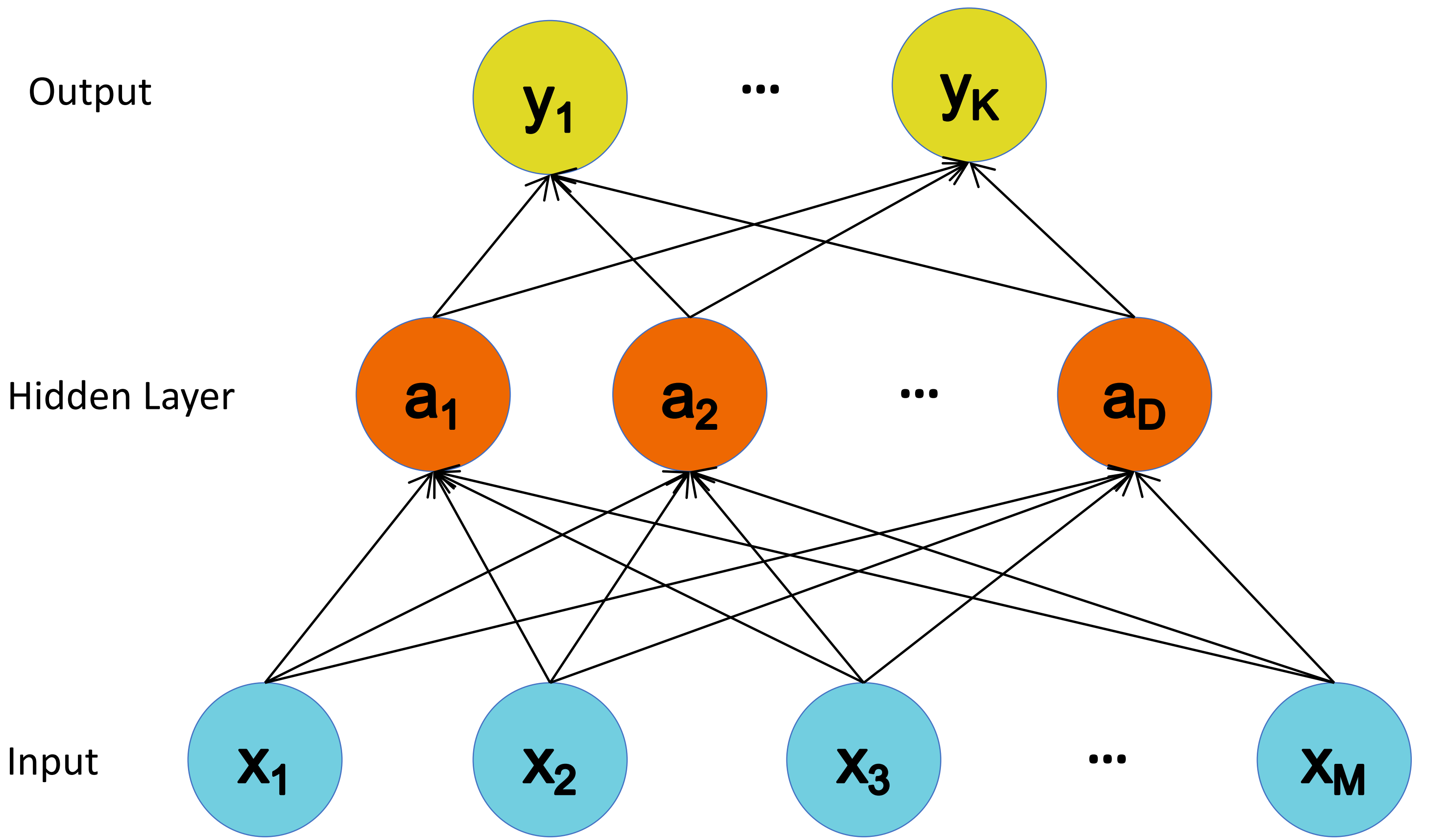

Slide from William Cohen

# Objective Functions for NNs

- Regression:
  - Use the same objective as Linear Regression
  - Quadratic loss (i.e. mean squared error)
- Classification:
  - Use the same objective as Logistic Regression
  - Cross-entropy (i.e. negative log likelihood)
  - This requires probabilities, so we add an additional "softmax" layer at the end of our network

| | Forward | Backward |
|---|---|---|
| Quadratic | $J = \frac{1}{2}(y - y^*)^2$ | $\frac{dJ}{dy} = y - y^*$ |
| Cross Entropy | $J = y^* \log(y) + (1 - y^*) \log(1 - y)$ | $\frac{dJ}{dy} = y^* \frac{1}{y} + (1 - y^*) \frac{1}{y - 1}$ |

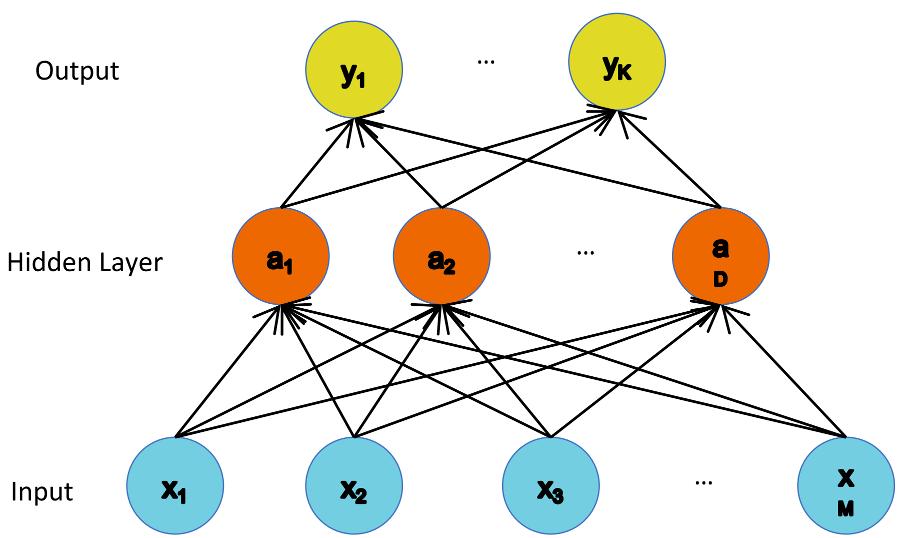
# Multi-Class Output

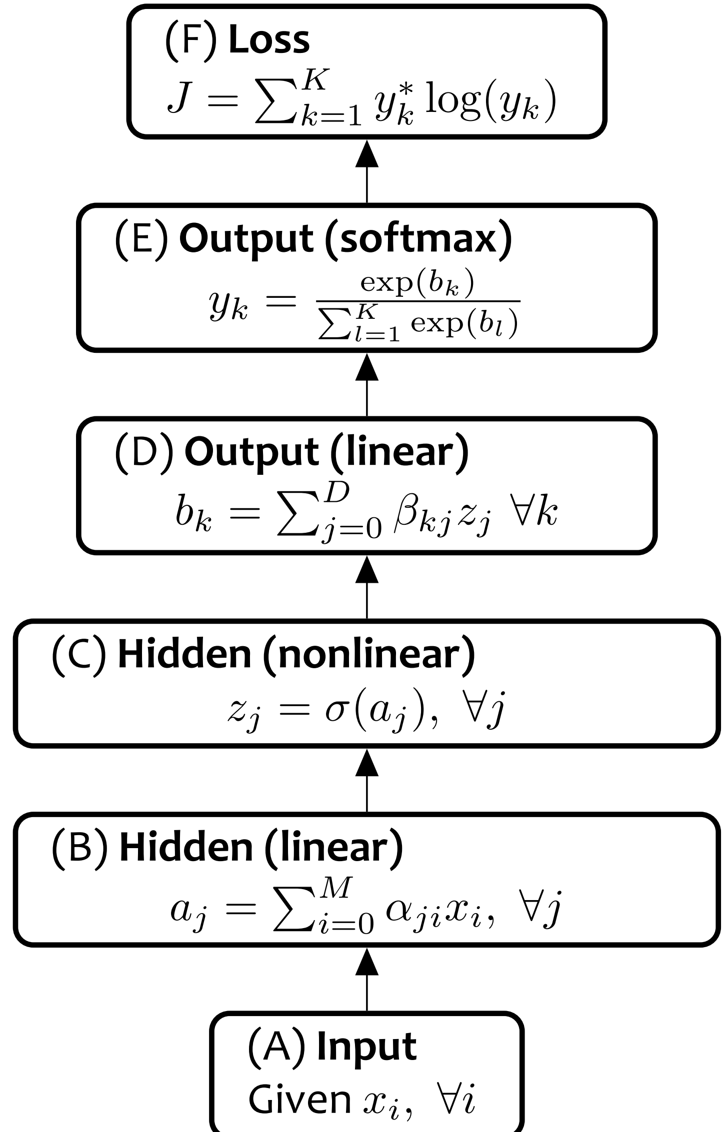


# Multi-Class Output

## Softmax:

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



# Cross-entropy vs. Quadratic loss

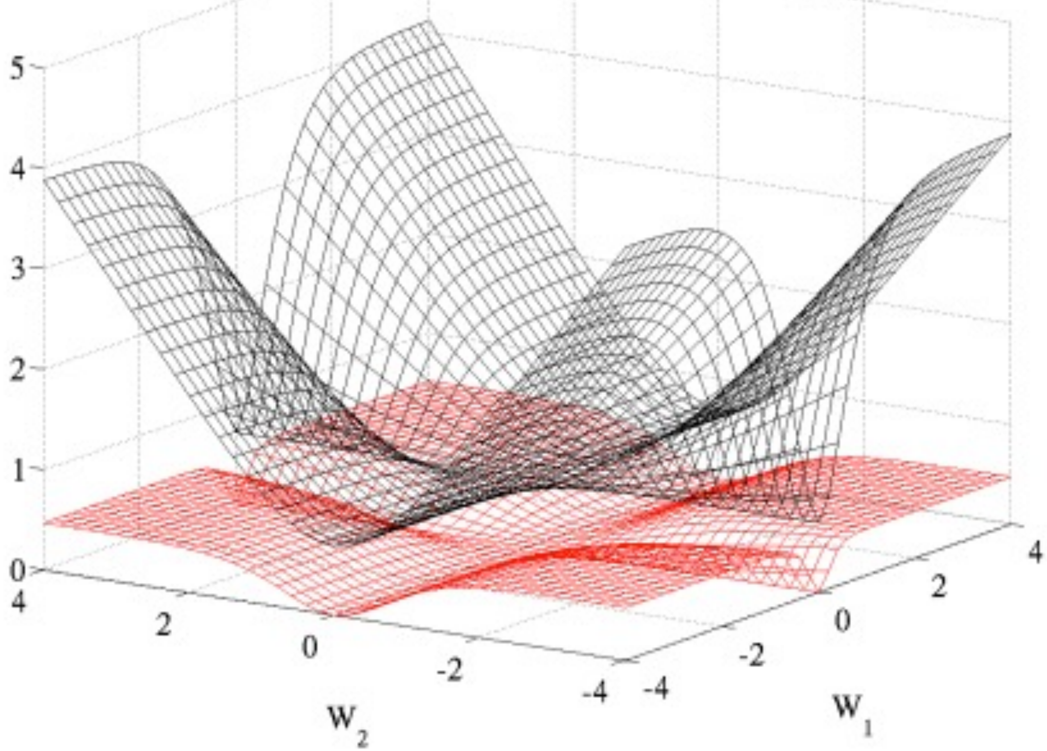



Figure 5: *Cross entropy (black, surface on top) and quadratic (red, bottom surface) cost as a function of two weights (one at each layer) of a network with two layers, $W_1$ respectively on the first layer and $W_2$ on the second, output layer.*

Figure from Glorot & Bentio (2010)

# A Recipe for Machine Learning

1. Given training data:

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

2. Choose each of these:

– Decision function

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

– Loss function

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

3. Define goal:

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

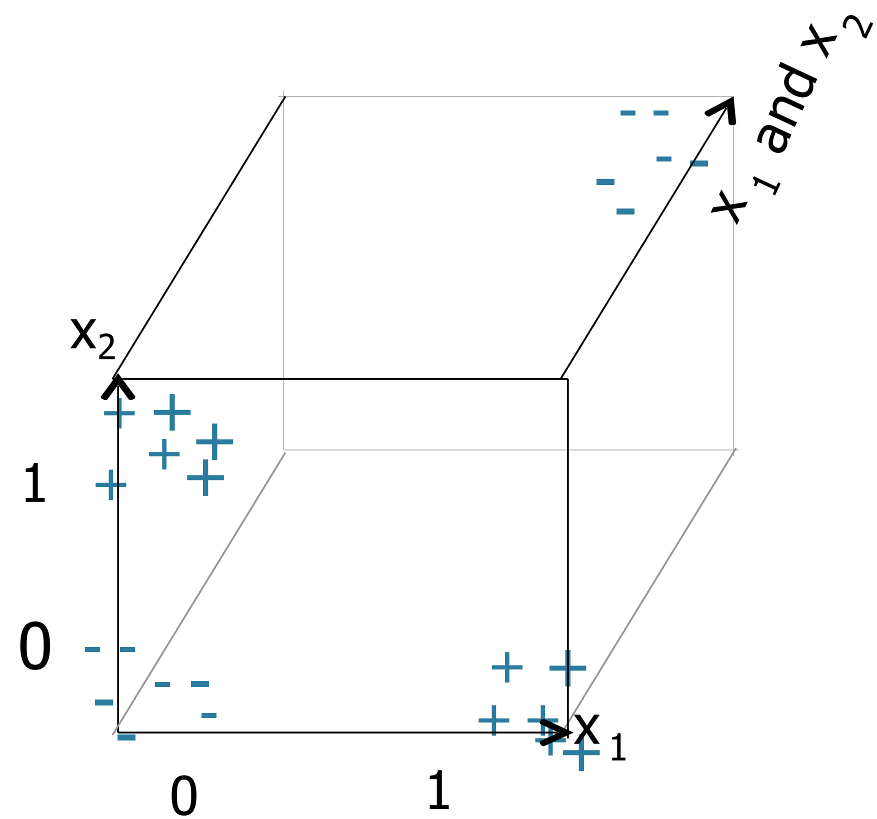
4. Train with SGD:

(take small steps opposite the gradient)

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

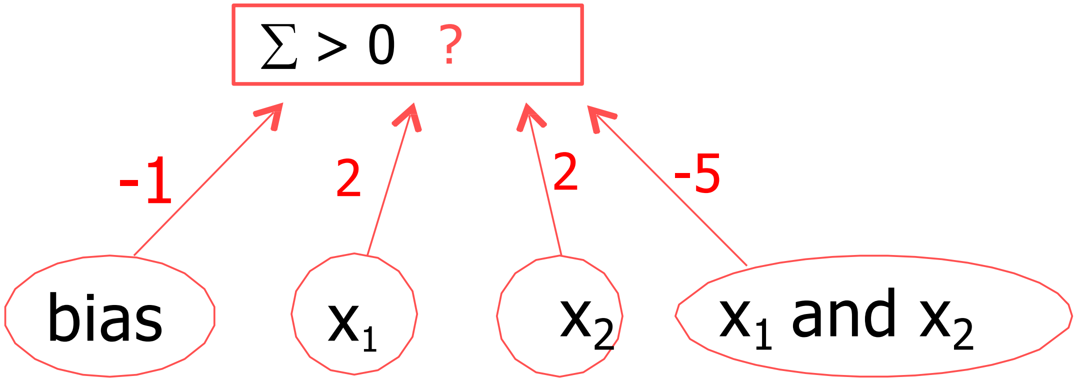
# An Non-linear Example

- Consider the xor function
  - $y(x) = 1$ iff $x_1$ xor $x_2$
- Clearly non-linear
  - No values for w will produce desired output
- We could solve this by adding a new feature
  - $x_3 = x_1$ xor $x_2$



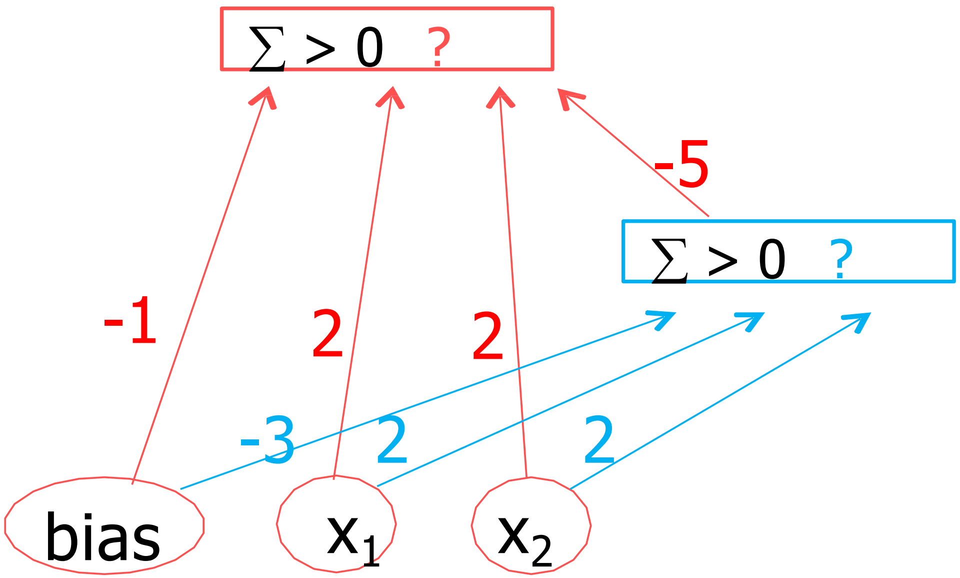
# The Neural Network Solution

- Learn new features that are linearly separable




- We now have a linear classifier for XOR
- How do we learn these features?

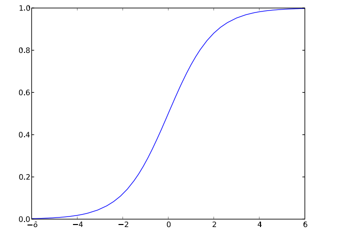
# The Neural Network Solution




- The new features are learned by linear classifiers
  - All other hidden nodes (not shown) just replicate input
- The activation function makes the feature 1 or 0

# Non-linear Activation Functions

- What non-linear function should we use for activation function h?
  - Typically use sigmoid functions
    - Logistic function




- Each hidden node has a threshold for activation
  - Will be 0 and then quickly transition to 1
- This is what we use when we stack Perceptron

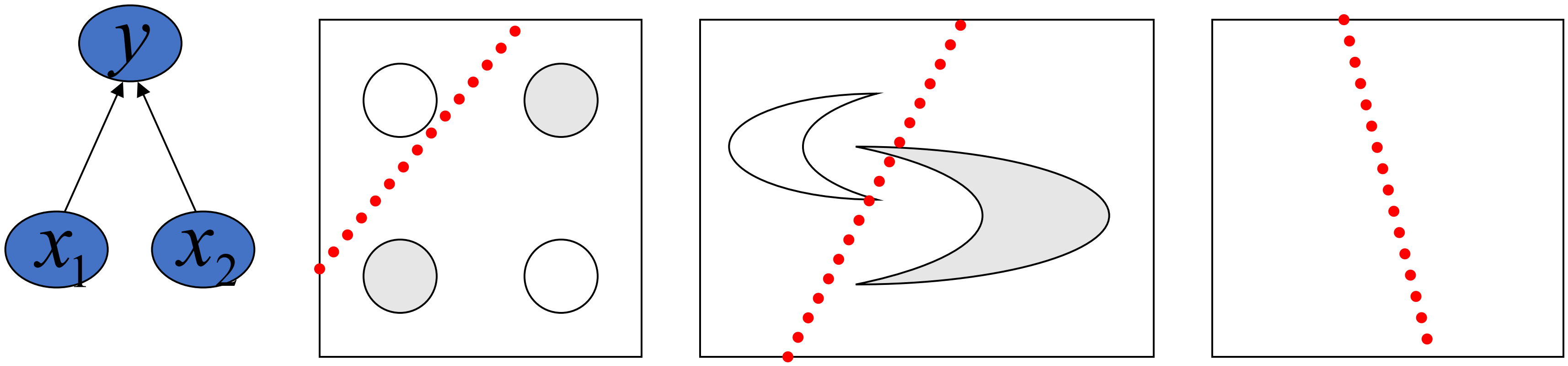
- This is why we think of hidden nodes as features
  - They are off and then when enough input they turn on
  - Learning input weights turns on the feature!

# Hypothesis Class

- What can a neural network learn?
  - Obviously highly non-linear outputs
- Universal approximators
  - With enough hidden layers and hidden nodes a neural network can model any continuous function on compact input domain (some number of inputs)
  - The power of the networks depends on its structure
    - General result independent of activation functions

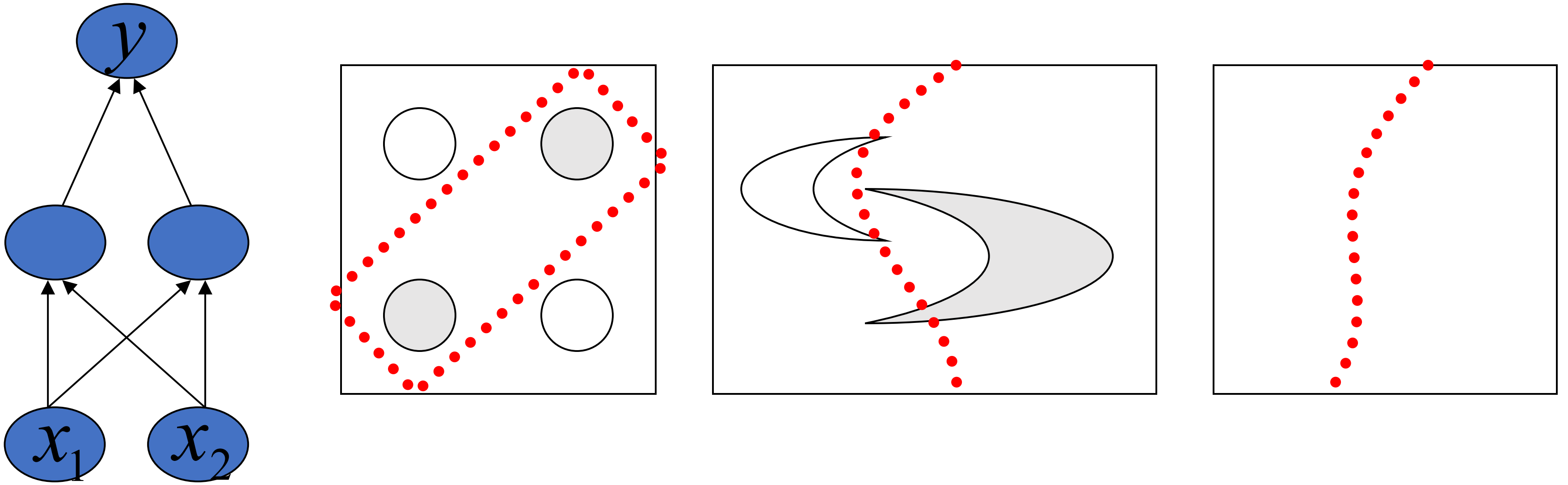
# Decision Boundary

- 0 hidden layers: linear classifier
  - Hyperplanes

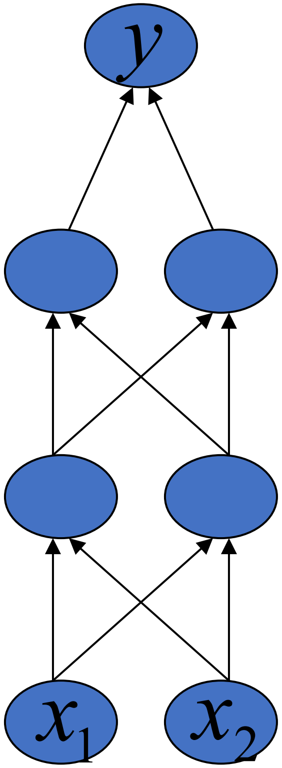


# Decision Boundary

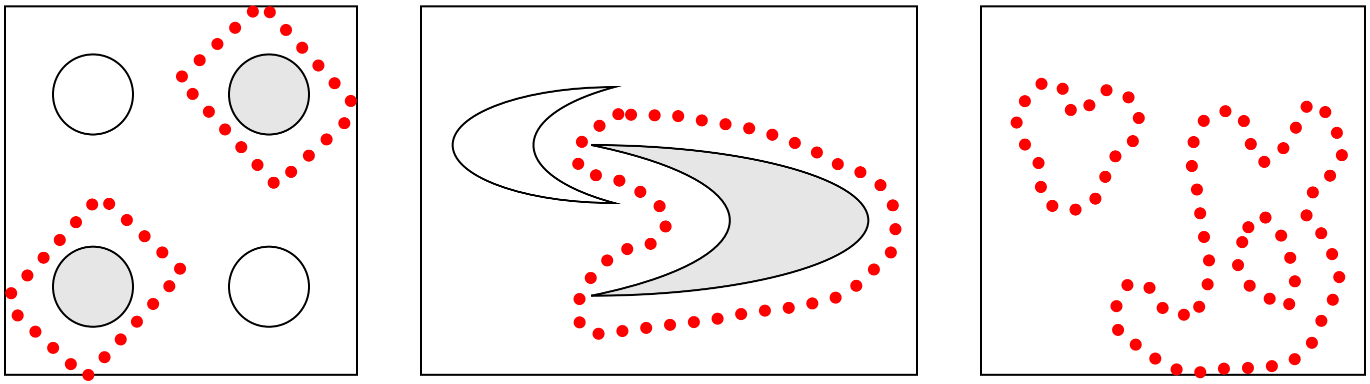
- 1 hidden layer
  - Boundary of convex region (open or closed)



# Decision Boundary




- 2 hidden layers
  - Combinations of convex regions

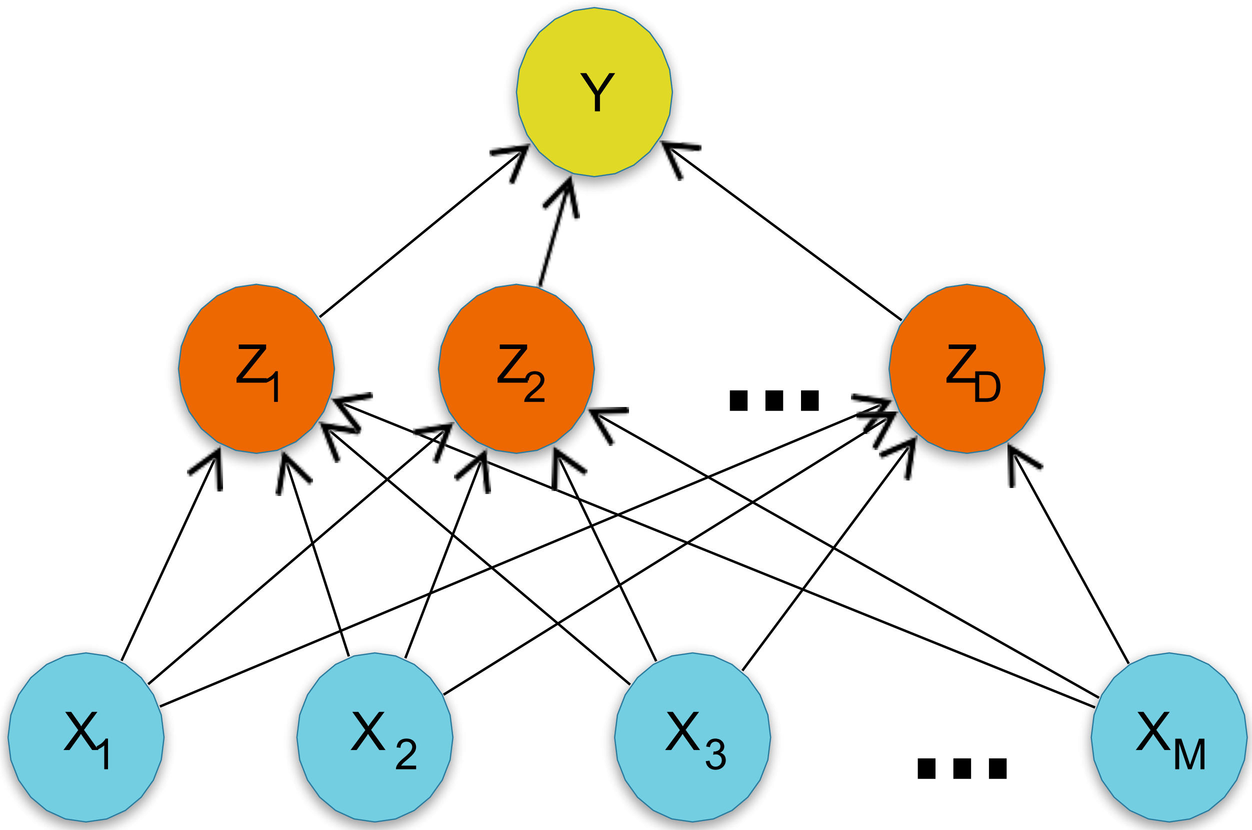


# Prediction

Compute linear combination of hidden nodes and pass through logistic function

Hidden nodes are now new features

Compute each hidden node as linear combination of x and pass to logistic




Forward propagation through the network

# Classification Objective

- Define an error function and minimize
- Cross entropy error function

$$E(w) = -\sum_{i=1}^{N}\{y_i ln\hat{y_i} + (1 - y_i)\ln(1 - y_i)\}$$

- This arises naturally when we consider a logistic probability model and take the negative log likelihood

# Regression Objective

- For regression we use the sum of squares error

$$E(w) = \frac{1}{2}\sum_{i=1}^{N}(\widehat{y_i} - y_i)^2$$

- If we assume a Gaussian model for y, the error function arises from maximizing the likelihood function
  - We saw the same thing for linear regression

# Combined Model

- Writing the generalized linear classifier with generalized non-linear basis functions

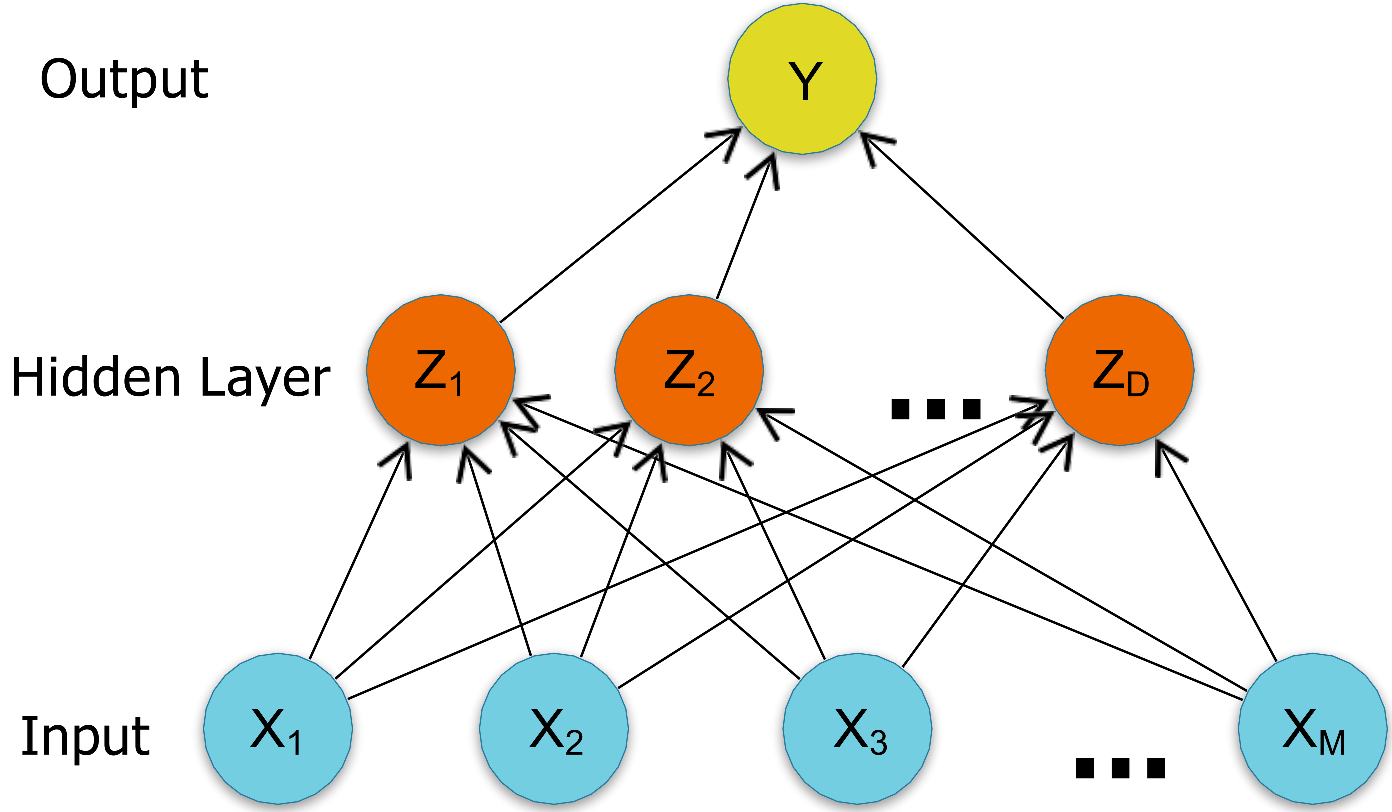
$$y(x; w) = h^{(2)}\left(\sum_{j=1}^{D} w_j^{(2)} h^{(1)}\left(\sum_{i=1}^{M} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_0^{(2)}\right)$$

- $h^{(1)}$ is the non-linear function for the basis function
- $h^{(2)}$ is the non-linear function for the output
- $w^{(1)}$ are the parameters for the basis function
- $w^{(2)}$ are the parameters for the linear model
- $w_0$ are the bias parameters (shown here for clarity)

# Training

- Prediction is relatively easy
- Learning is where the magic happens
- Strategy: compute the gradient of the objective function
    - Similar to perceptron
    - Gradient based update

# Graphical Representation

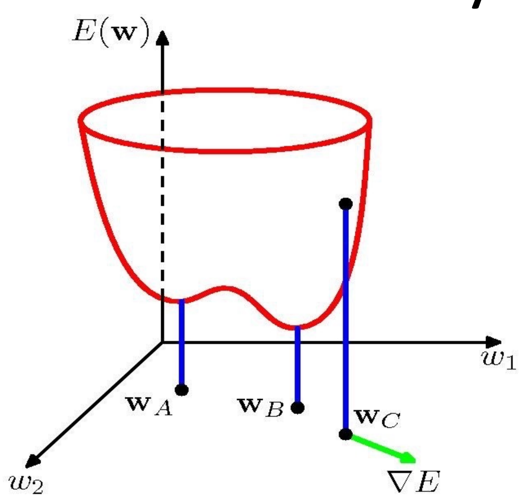


# Training

- Prediction is relatively easy
- Learning is where the magic happens
- Strategy: compute the gradient of the objective function
    - Similar to perceptron
    - Gradient based update
- For the moment: assume black box computes gradient

# Gradient Based Optimization

- The objective function is now non-convex
- Gradient based optimization NOT guaranteed to find  global optimum
- For now: use gradient stochastic  gradient and hope for the best
- Next time: tricks for non-convexity  key to learning good networks



# Computing the Gradient

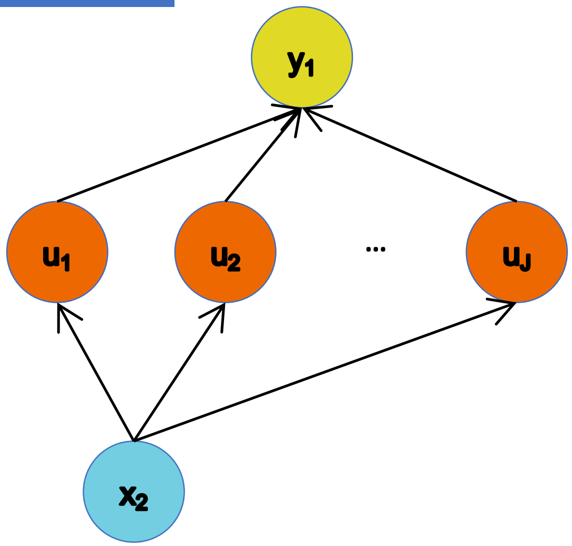
- For arbitrary Neural Network architectures we can use Backpropagation!

# Chain Rule

**Given:** $\boldsymbol{y} = g(\boldsymbol{u})$ and $\boldsymbol{u} = h(\boldsymbol{x})$.

**Chain Rule:**

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



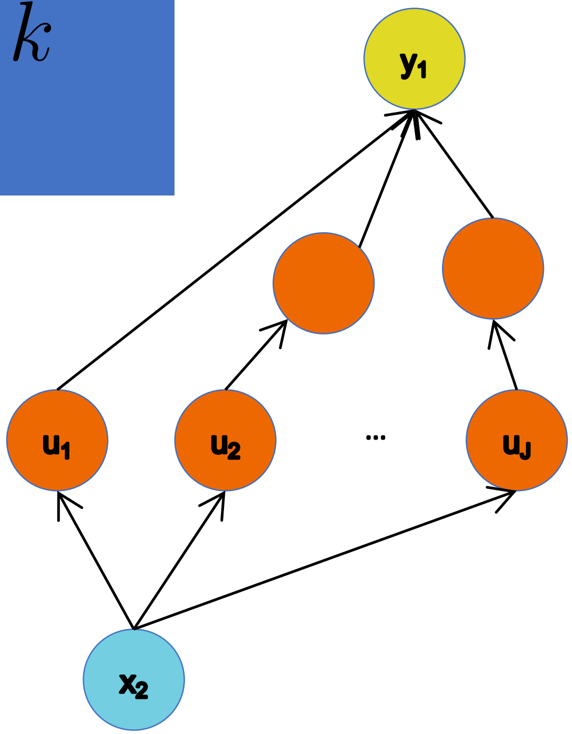
# Chain Rule

**Given:** $\boldsymbol{y} = g(\boldsymbol{u})$ and $\boldsymbol{u} = h(\boldsymbol{x})$.

**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.

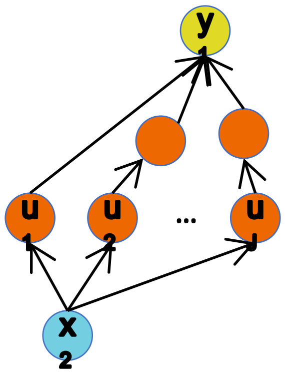


# Chain Rule

**Given:** $\boldsymbol{y} = g(\boldsymbol{u})$ and $\boldsymbol{u} = h(\boldsymbol{x})$.

**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:**

1. **Instantiate the computation as a directed acyclic graph**, where each intermediate quantity is a node
2. At each node, store (a) the quantity computed in the forward pass and (b) the **partial derivative** of the goal with respect to that node's intermediate quantity.
3. **Initialize** all partial derivatives to 0.
4. Visit each node in **reverse topological order**. At each node, add its contribution to the partial derivatives of its parents

This algorithm is also called **automatic differentiation in the reverse-mode**

# Algorithm: Neural Network

- Train: Given examples X and Y
  - Y can be multiple outputs
  - Define a network structure
  - ex. 2 layer feed forward, D nodes in hidden layer
  - Learn parameters w
- Predict: given example x
  - For 2 layer feed forward, compute output as

$$y(x; w) = h^{(2)}\left(\sum_{j=1}^{D} w_j^{(2)} h^{(1)}\left(\sum_{i=1}^{M} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_0^{(2)}\right)$$

# Multi-Layer Perceptrons

- **Fitting a function to data**

- Fitting: gradient based optimization with back-propogation
- Function: non-linear: linear combination of generalized linear functions
  - Universal approximations
  - can model any continuous function on compact input domain (some number of inputs)
- Data: Batch training using stochastic methods