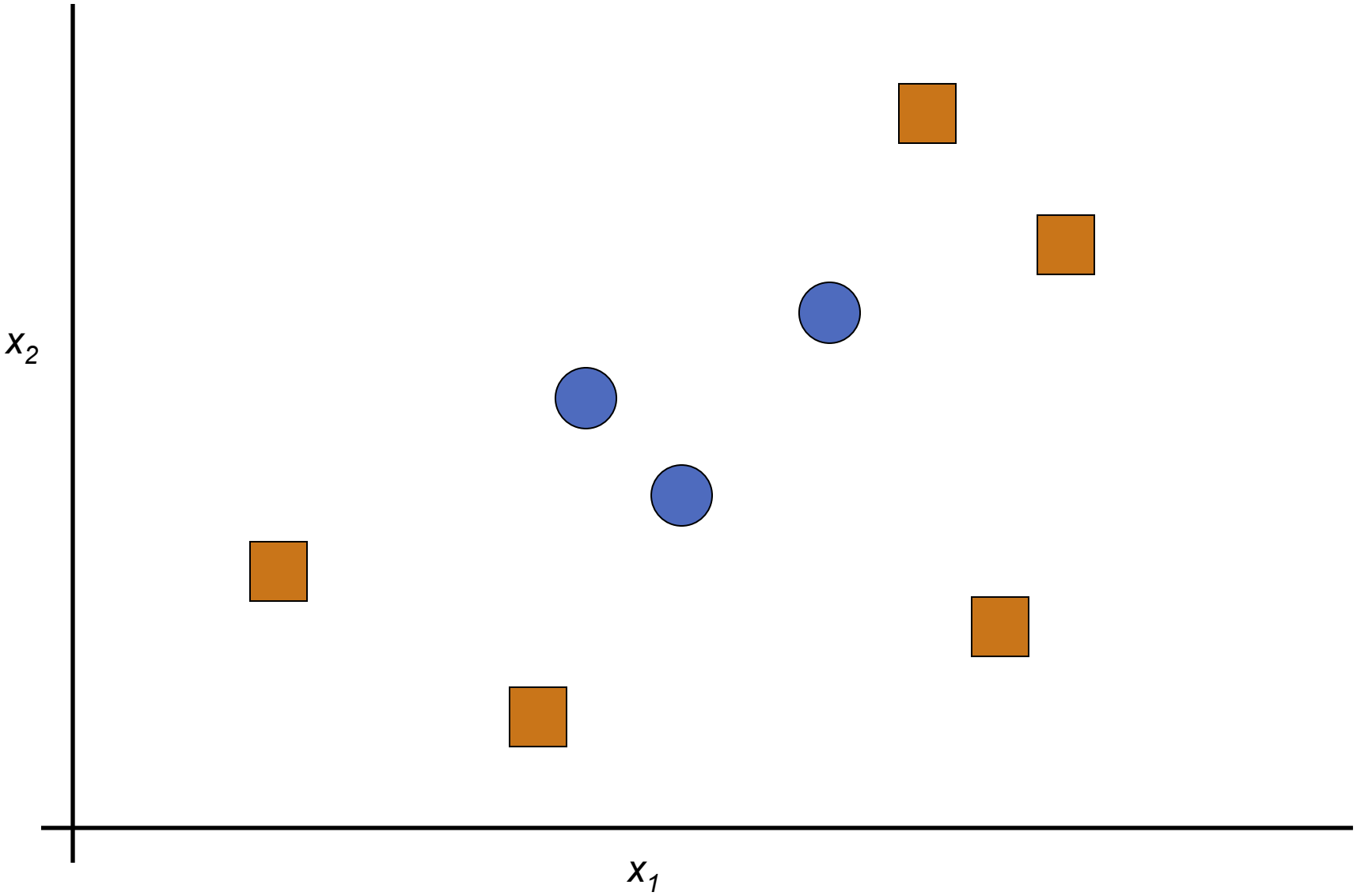
Deep Learning

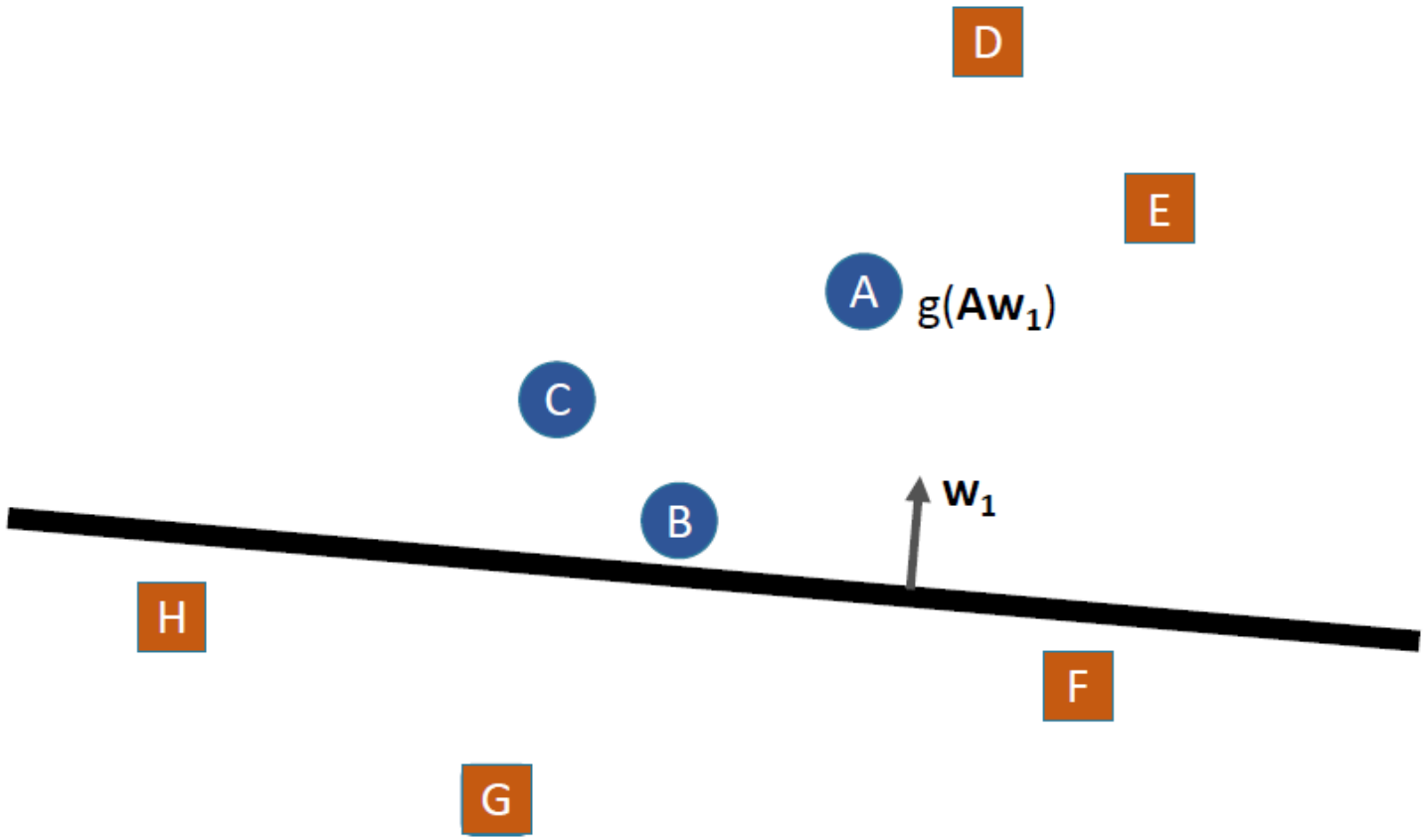
Winter 2026

Multi-layer Perceptrons

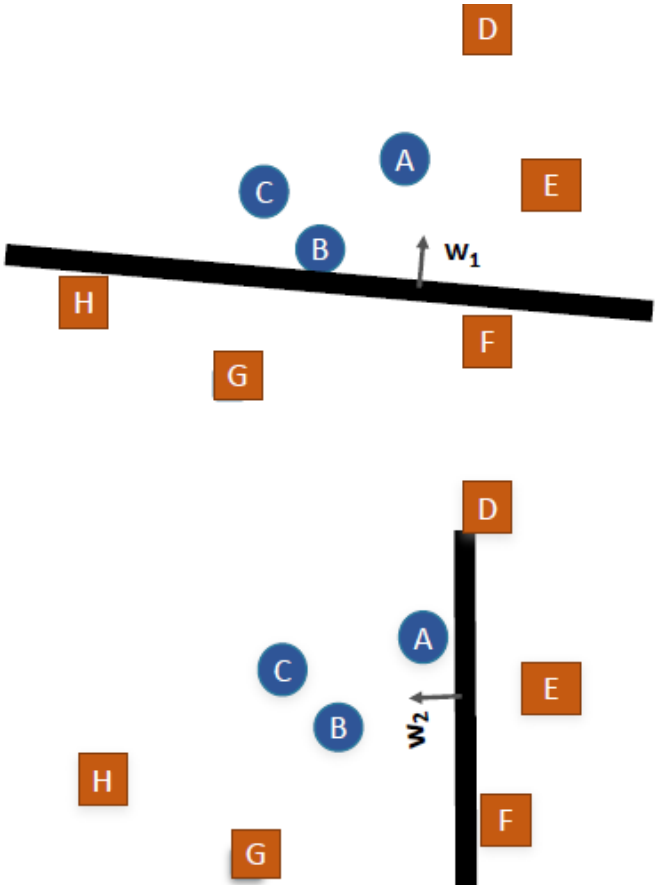
Single Perceptrons

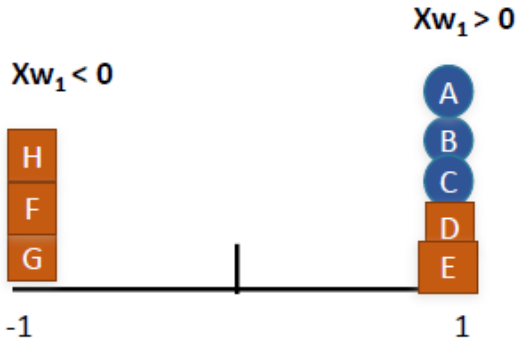


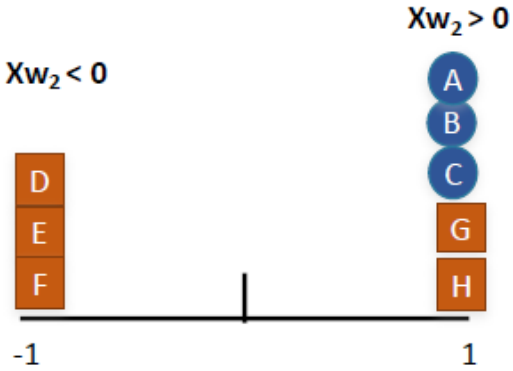
Single Perceptron



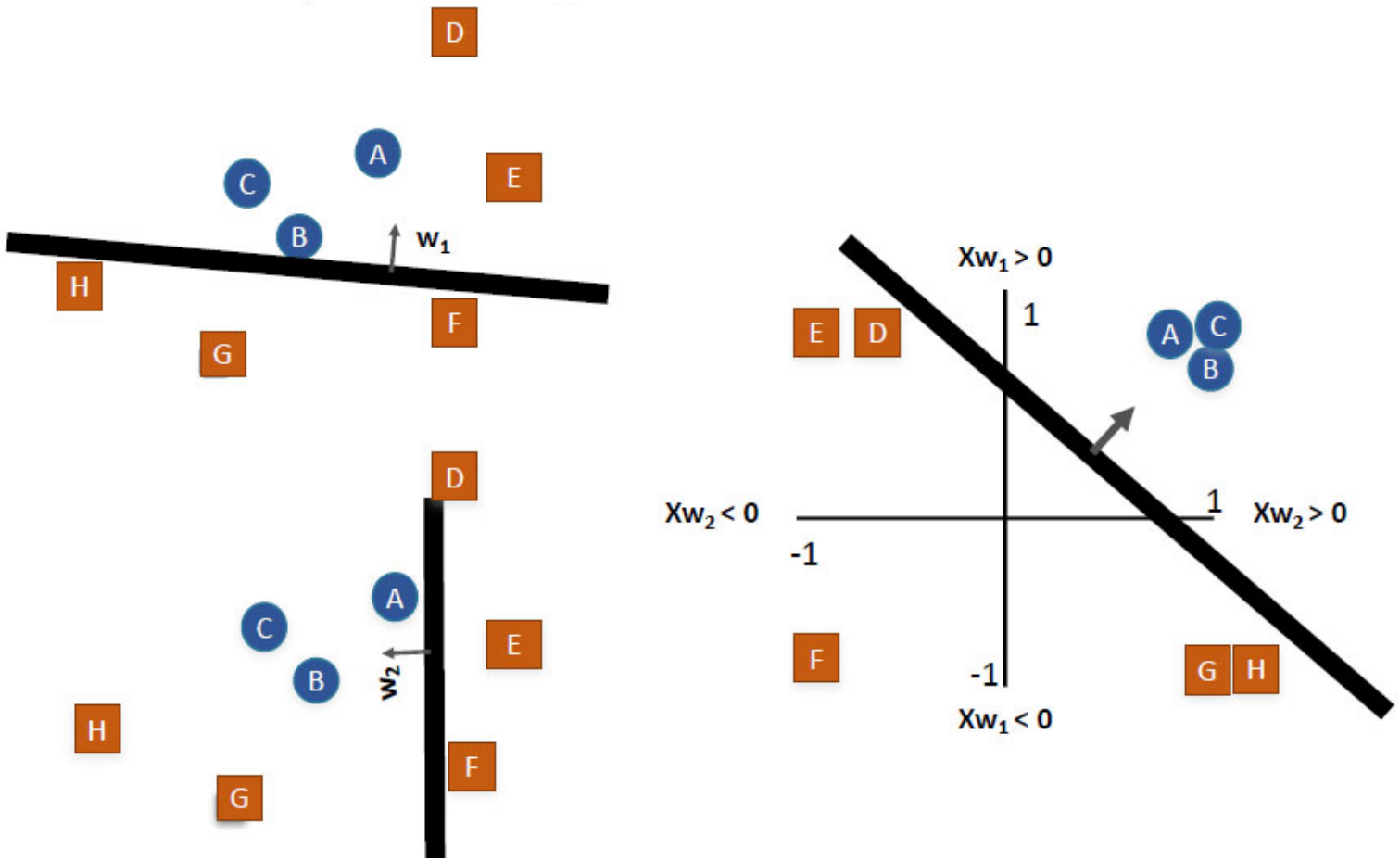
Single Perceptrons -- Another Way to Map Features



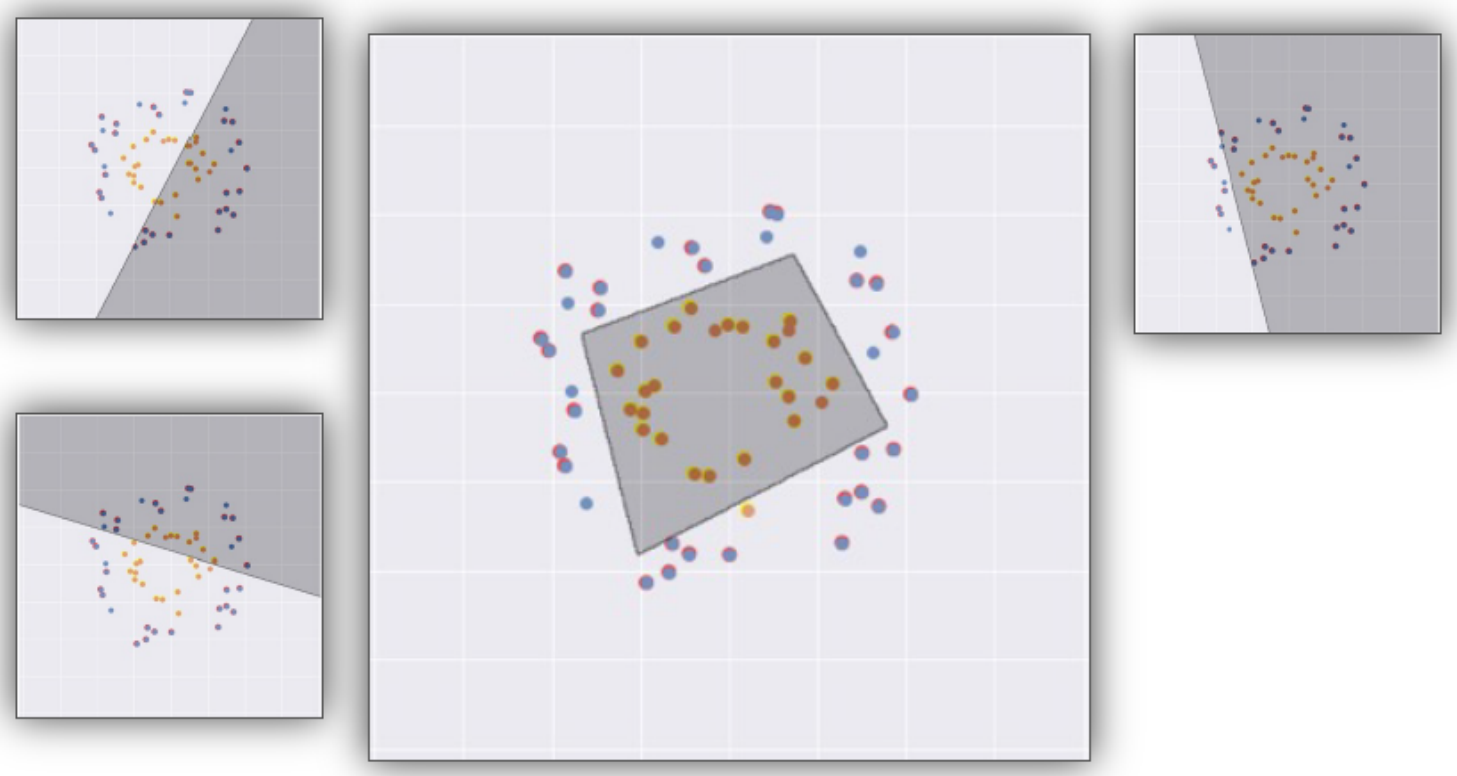




Single Perceptrons -- Another Way to Map Features

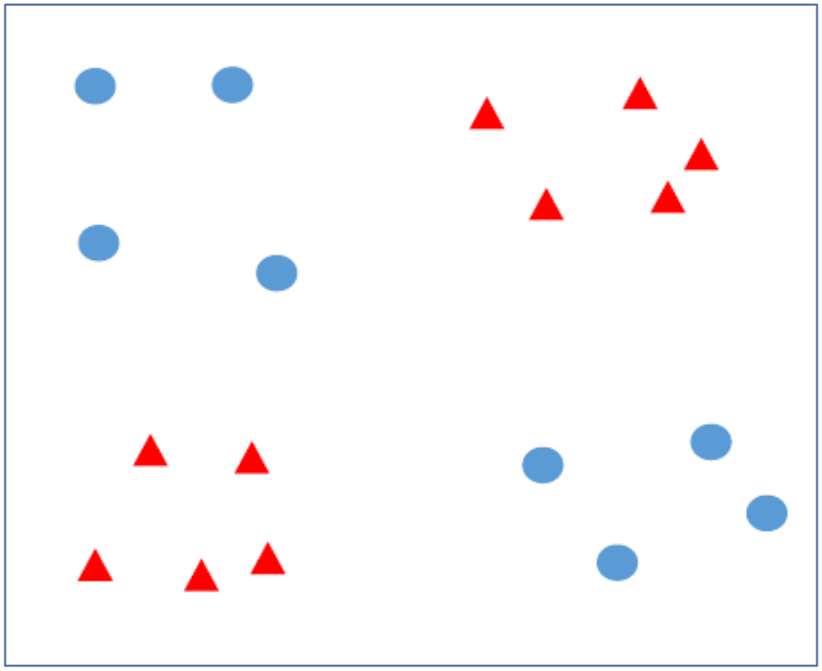


Single Perceptrons -- Another Way to Map Features



Different from XOR?

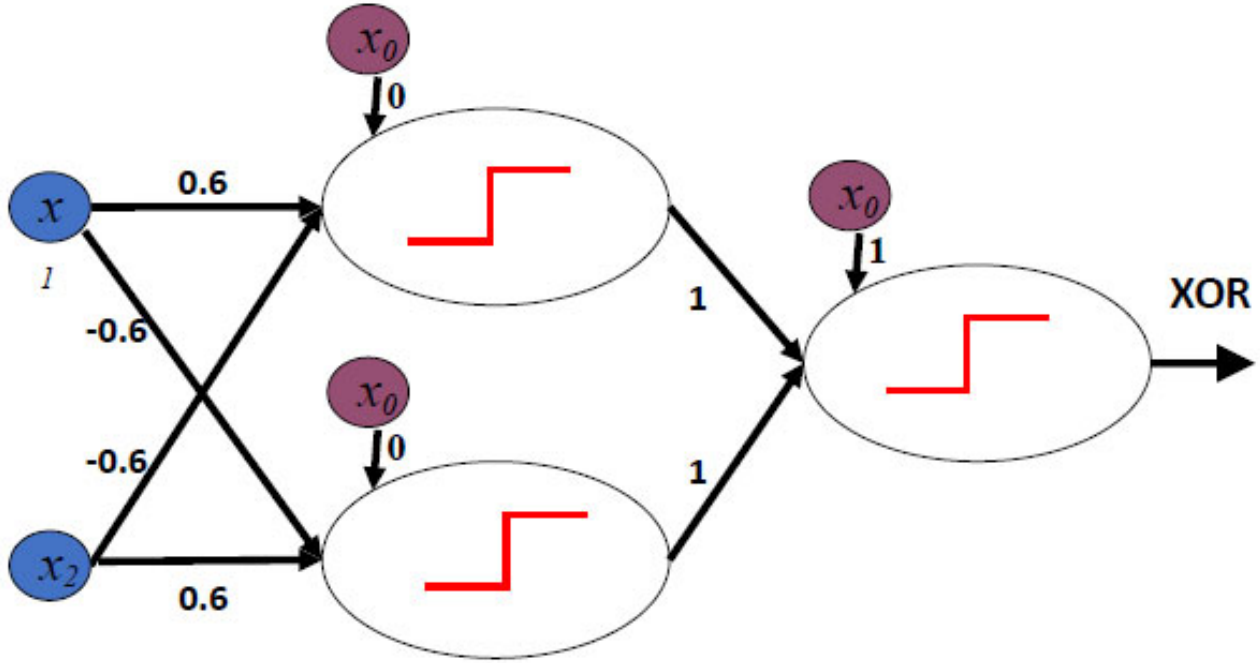
One perceptron: Only linear decisions



This is XOR.

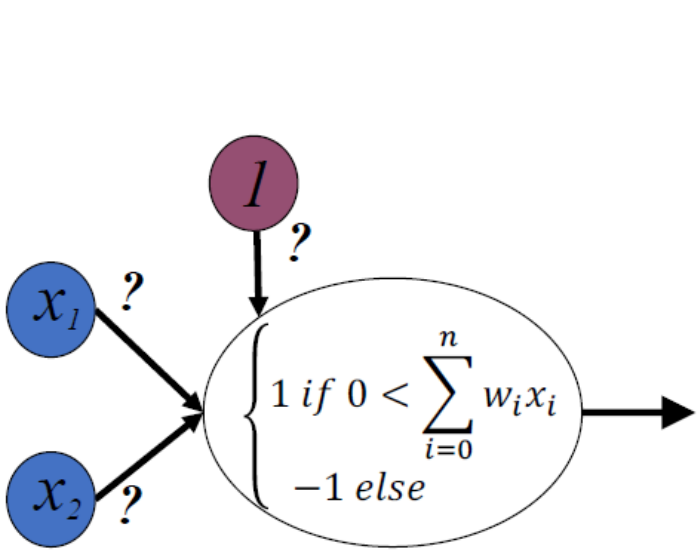
It can't learn XOR.

Combining Perceptrons Can Make Any Boolean Function

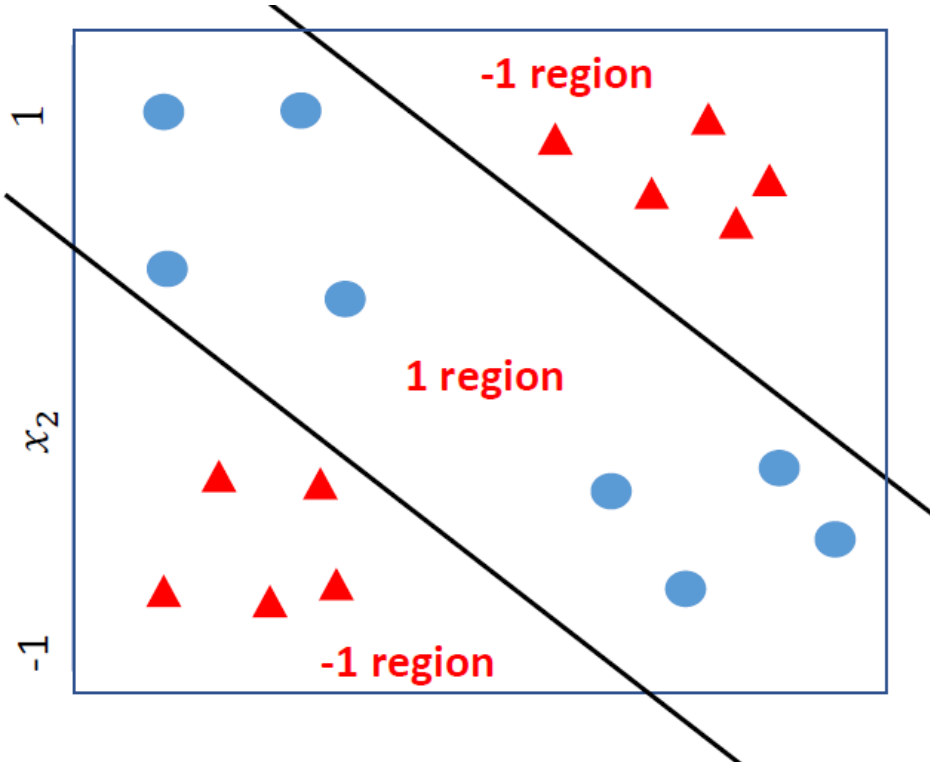


...if you can set the weights & connections right

What About XOR?

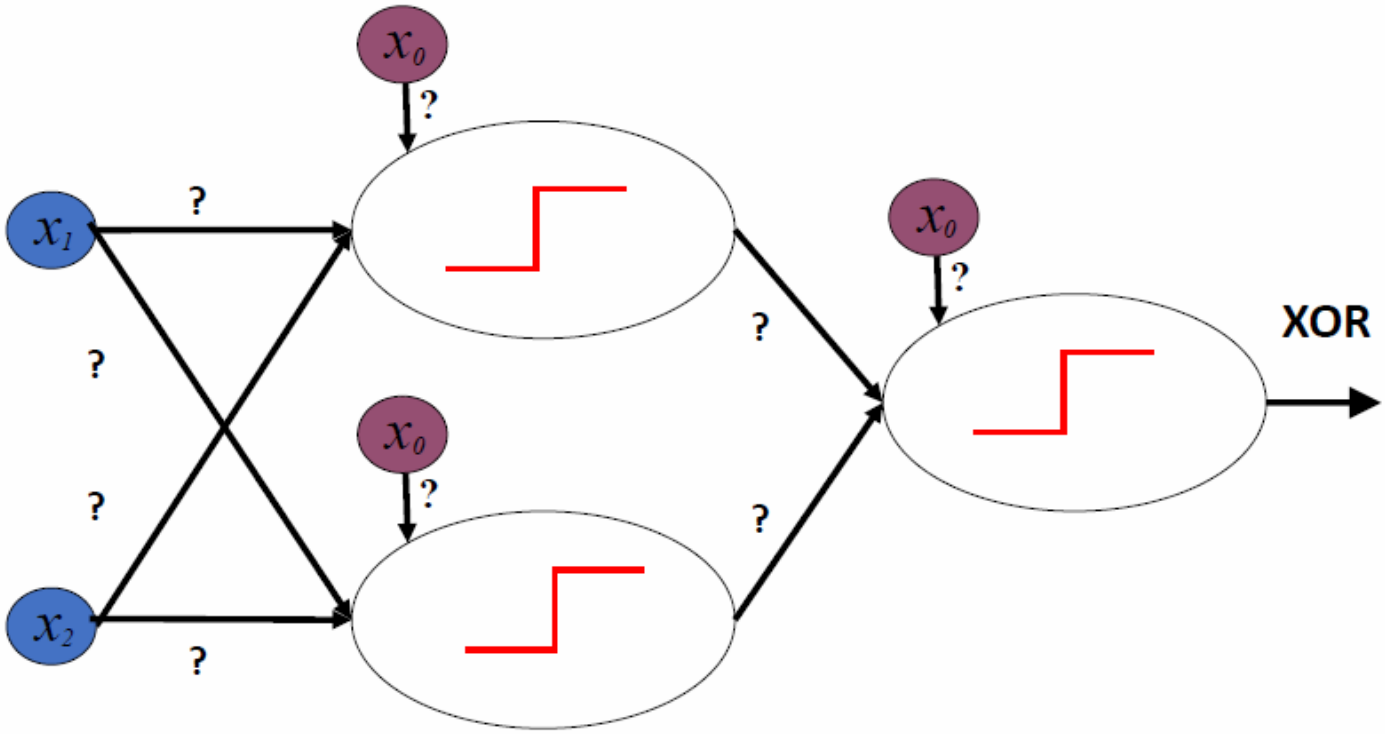


XOR



Perceptron is a *linear* classifier; it only learns linear boundaries.

Multi-layer Perceptrons



$$\hat{y} = g(v_0 + v_1 \cdot g(\mathbf{x}\mathbf{w_1}) + v_2 \cdot g(\mathbf{x}\mathbf{w_2}))$$

Perceptron Algorithm

The decision boundary

$$0 = g(x) = \mathbf{w}^{\mathbf{T}}\mathbf{x}$$

The classification function

$$h(\mathbf{x}) = \begin{cases} 1 & \text{if } g(\mathbf{x}) > 0 \\ -1 & \text{otherwise} \end{cases}$$

$$m = |D| = size\ of\ data\ set$$

The weight update algorithm

$\mathbf{w} =$ *some random setting*

Do

$k = (k+1) \bmod (m)$

if $h(\mathbf{x}_k) != y_k$

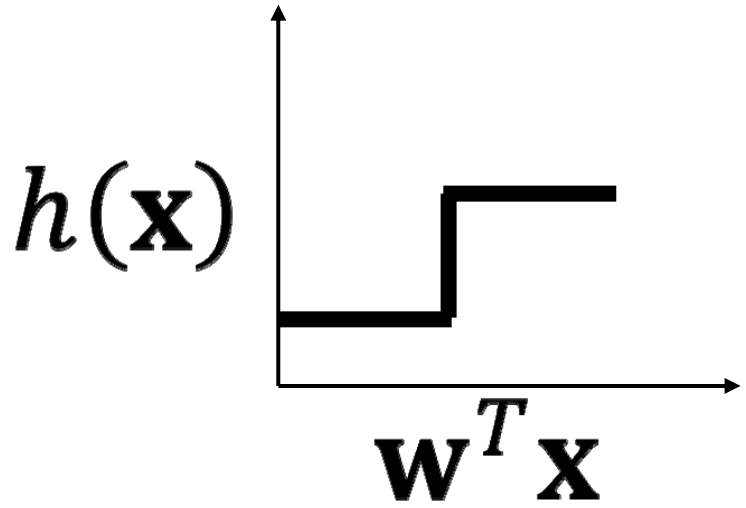
$\mathbf{w} = \mathbf{w} + \mathbf{x}y$

Until $\forall k,\ h(\mathbf{x}_k) = y_k$

Warning: Only guaranteed to terminate if classes are linearly separable!

This means you have to add another exit condition for when you've gone through the data too many times and suspect you'll never terminate.

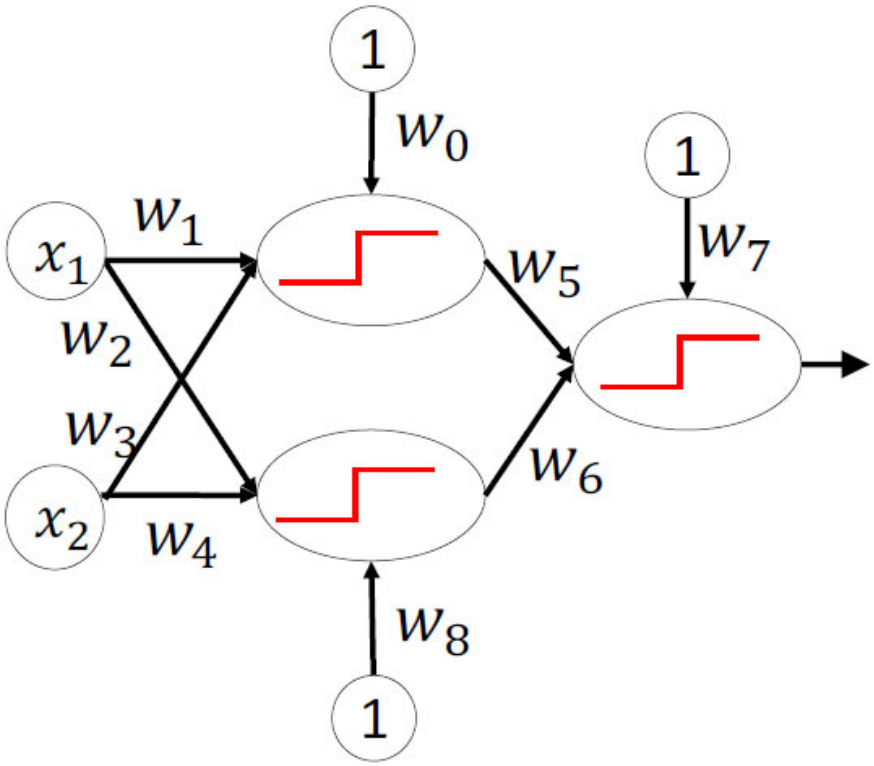
Perceptron Problem: The Step Function



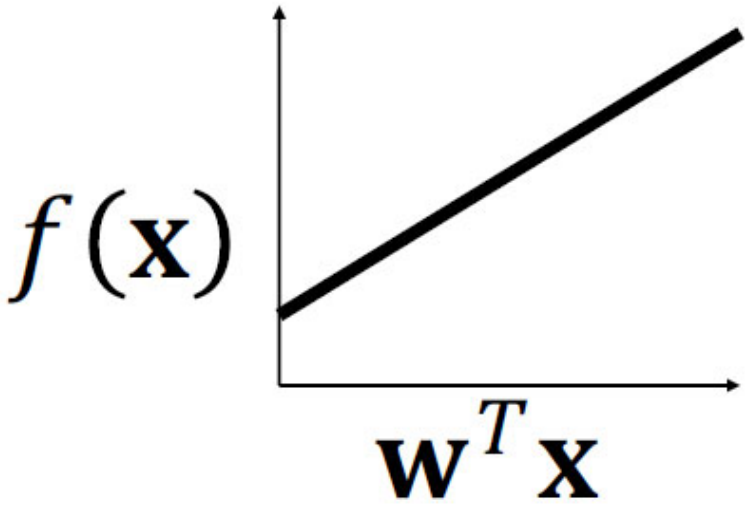
$$h(x) = \begin{cases} 1 & if\ 0 < \mathbf{w}^T\mathbf{x} \\ 0 & else \end{cases}$$

The Problems with Step Functions

- Stymies multi-layer weight learning
- Limits us to a single layer of units
- Thus, only linear functions
- You can hand-wire XOR perceptrons, but the sytem can't learn XOR with perceptrons



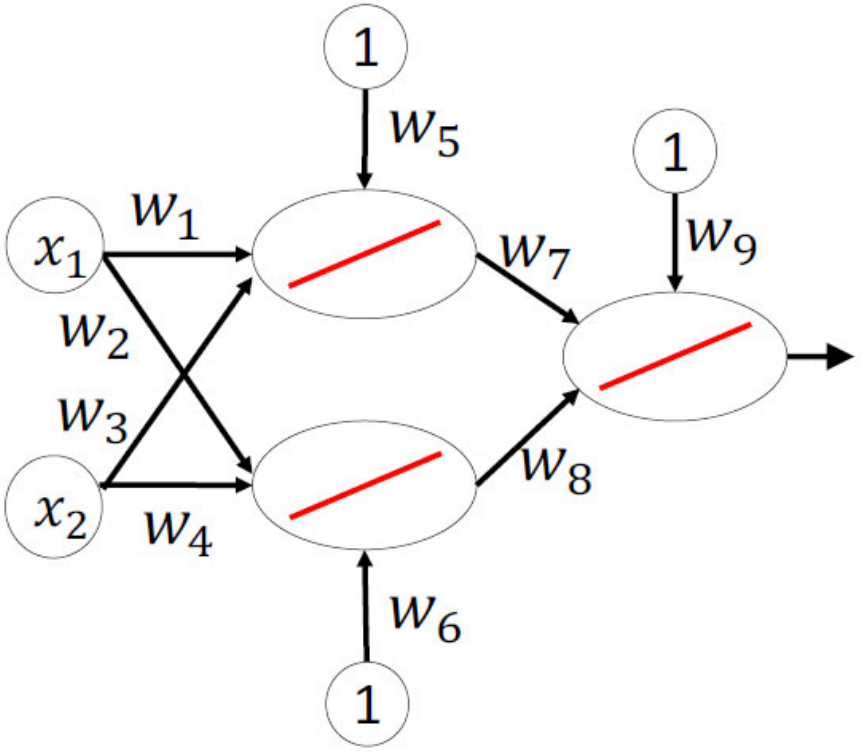
Solution: Remove The Step Function



$$f(\mathbf{x}) = \sum_{i=0}^{n} w_i x_i = \mathbf{w}^T\mathbf{x}$$

What If We Use Linear Functions?

- All changes in input result in changed output
- This gives us a gradient everywhere
- We can learn multiple layers of weights.
- **Combining linear functions only gives you linear functions**
- you can't represent XOR



Why Do We Need Non-Linearities?

- Allows a MLP to learn nonlinear relationships

- What functions can we use?
  - Logistic, Tanh, ReLU, Sign, etc.

  - Any function g such

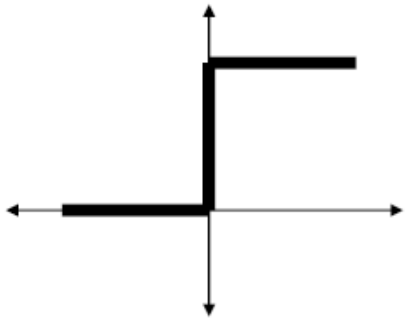
$$\hat{y} = g(v_1 \cdot g(\mathbf{w_1}^T \mathbf{x}) + v_2 \cdot g(\mathbf{w_2}^T \mathbf{x}))$$

$$\neq g(\mathbf{u_1}^T \mathbf{x})$$

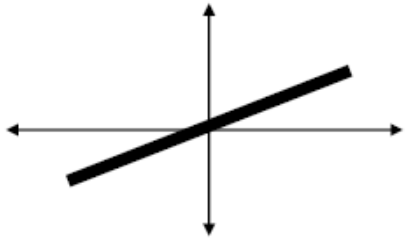
for any $\mathbf{u_1}$ (that's a function of $\mathbf{w}, \mathbf{v}$)

Sigmoid is the Best of Both

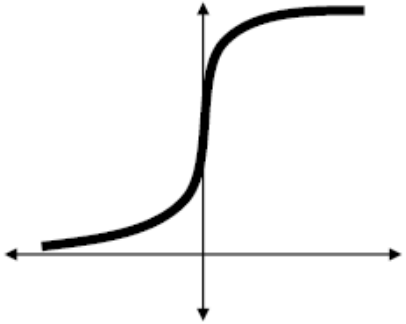
- Perceptron $$f(x) = \begin{cases} 1 \; if \; 0 < \sum_{i=0}^{n} w_i x_i \\ -1 \; else \end{cases}$$



- Linear $$f(x) = \mathbf{w}^T \mathbf{x} = \sum_{i=0}^{n} w_i x_i$$

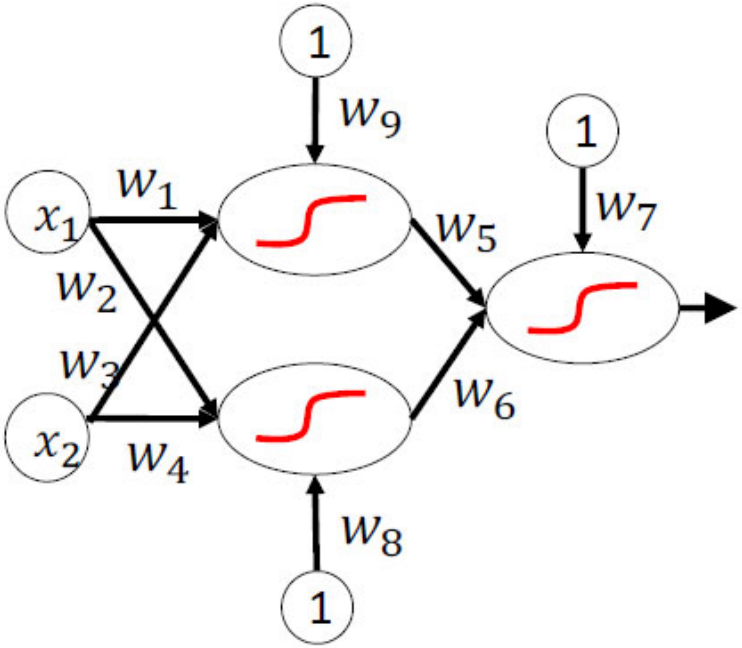


- Sigmoid $$f(x) = \sigma(x) = \frac{1}{1 + e^{-(\mathbf{w}^T \mathbf{x})}}$$



A Network of Sigmoid Units

- Gradients defined everywhere, so we can use gradient descent



Universal Approximators

- Neural networks can learn nonlinear decision boundaries
  - Is there anything they can't learn?
- Universal approximators
  - A large enough neural network can represent any continuous function
  - The power of the networks depends on its structure: is it wide enough? Is it deep enough?
  - These results are **theoretical**, not constructive. We haven't yet discussed how to train them!

How We Learn

- Perceptron: simple approach for a linear classifier
- Multilayer Perceptron
  - Combine several simple classifiers with nonlinearities
  - Can learn (almost) any nonlinear function
  - Gradient descent is still our friend:

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

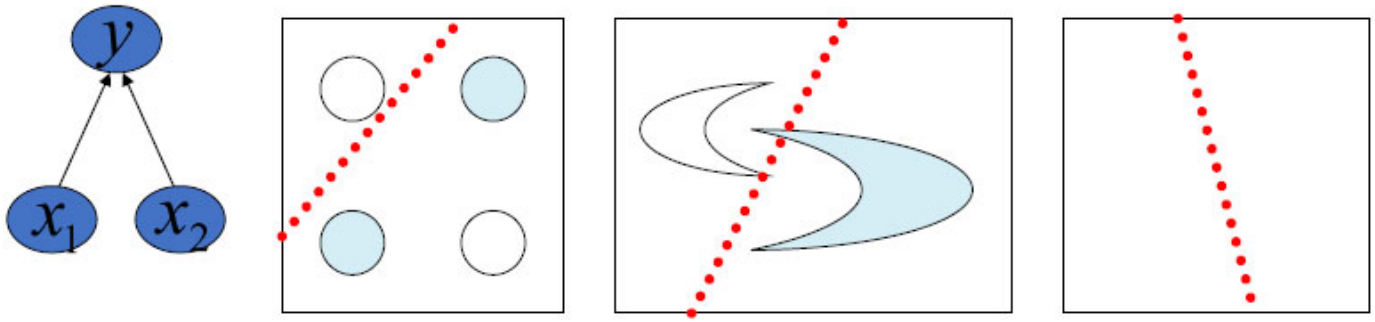
  - The real work will be in computing the gradient

The Promise of Many Layers

- Each layer learns an abstraction of its input representation (we hope)
- As we go up the layers, representations become increasingly abstract
- The hope is that the intermediate abstractions facilitate learning functions that require non-local connections in the input space (recognizing rotated & translated digits in images, for example)
- Modern neural networks can often be 100 layers deep

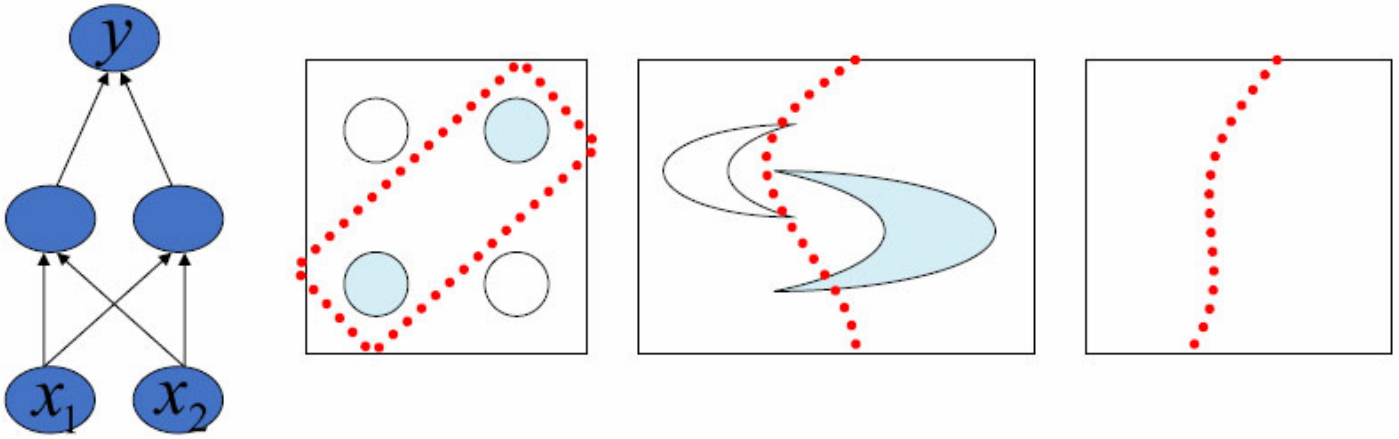
Decision Boundaries

- 0 hidden layers: linear classifier
  - Hyperplanes

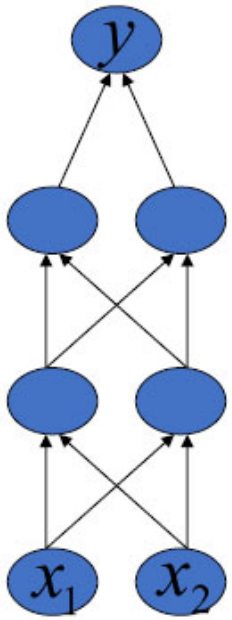


Decision Boundaries

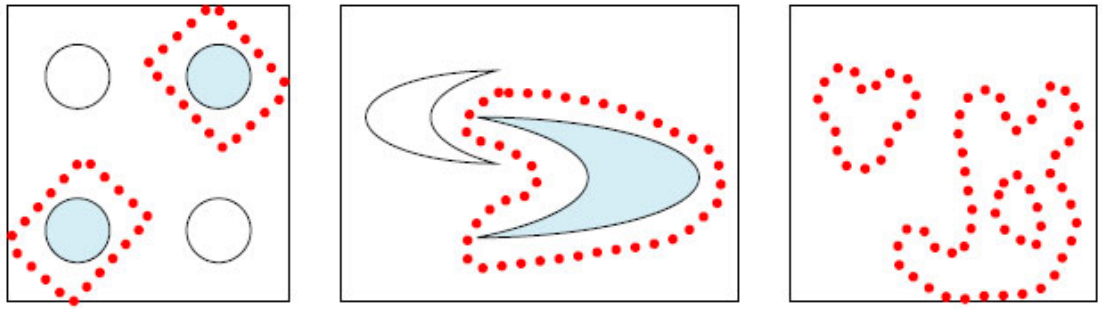
- 1 hidden layer
  - Boundary of convex region (open or closed)



Decision Boundaries

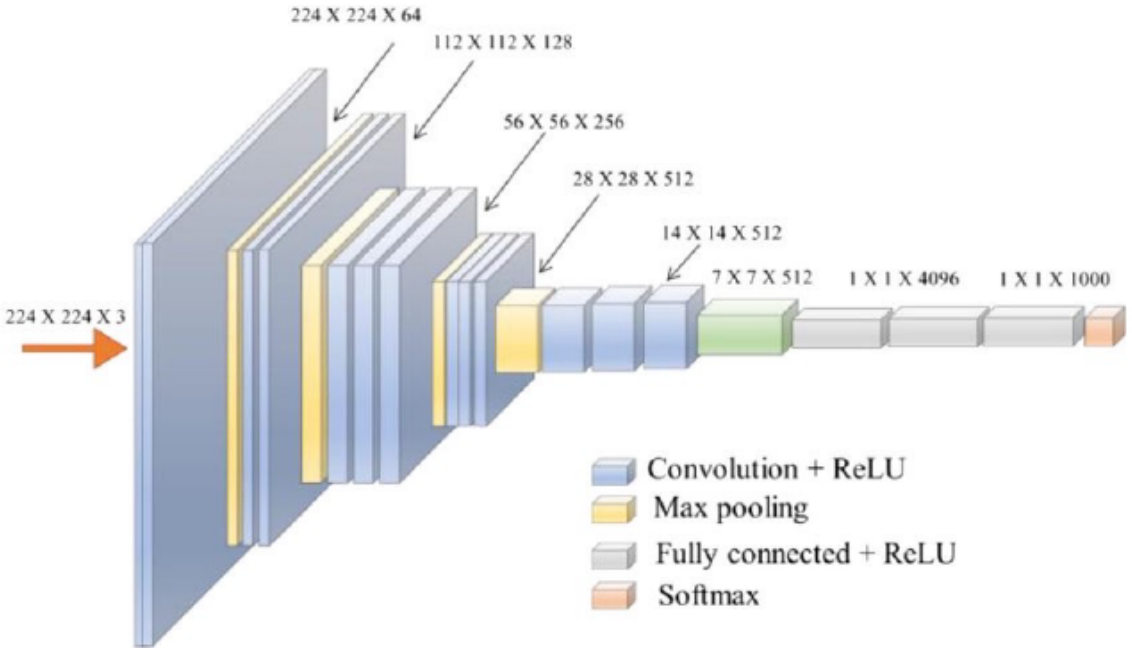


- 2 hidden layers
  - Combinations of convex regions

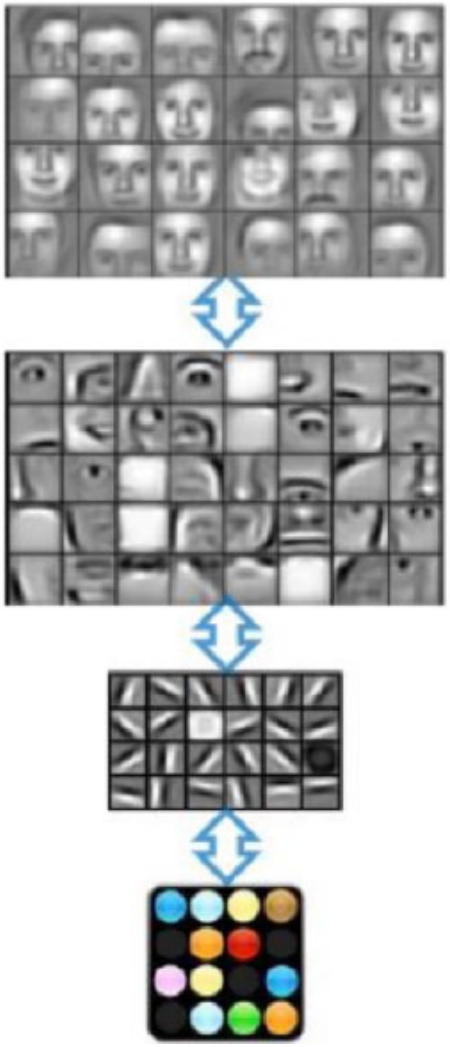


Deep Networks

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



Feature Representations



Example: Image Processing

3rd layer
"Objects"

2nd layer
"Object parts"

1st layer
"Edges"

Pixels