Last updated: Nov 26, 2012

MULTILAYER PERCEPTRONS

J. Elder

CSE 4404/5327 Introduction to Machine Learning and Pattern Recognition

Outline

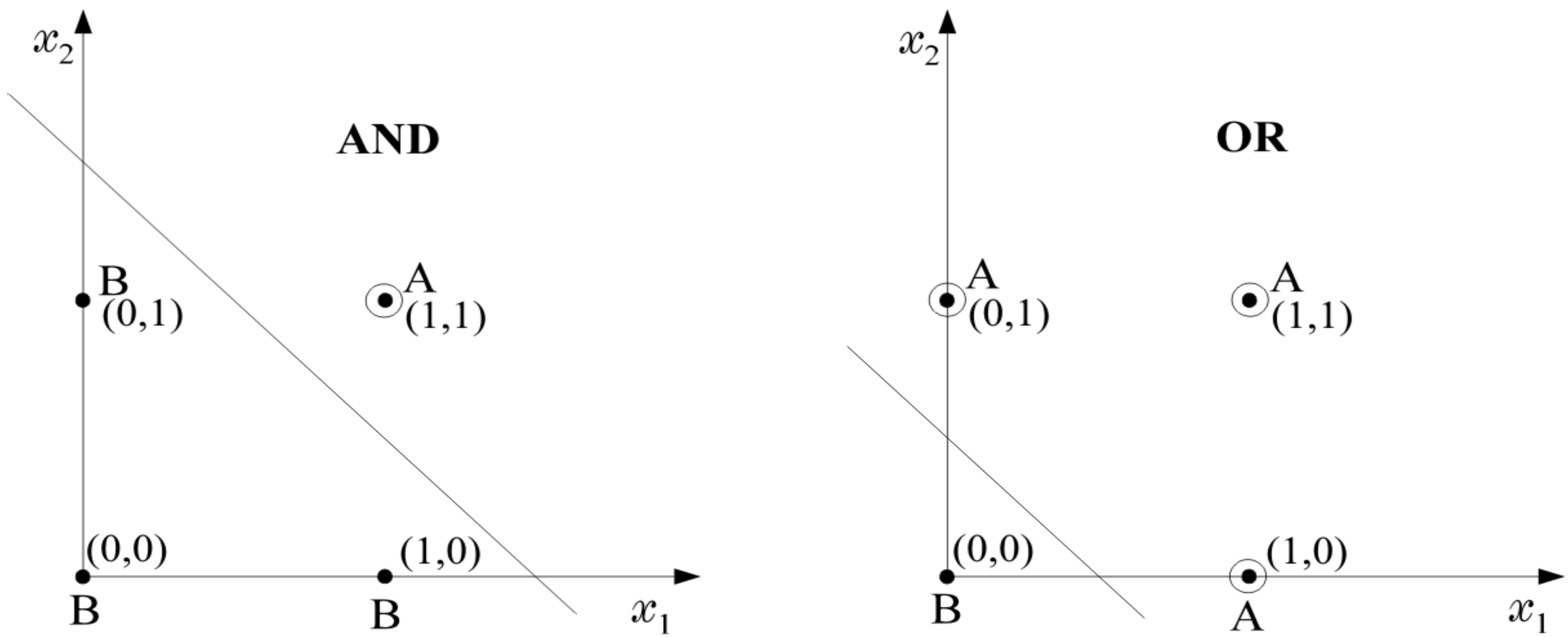
- Combining Linear Classifiers
- Learning Parameters

Outline

- **Combining Linear Classifiers**
- Learning Parameters

Implementing Logical Relations

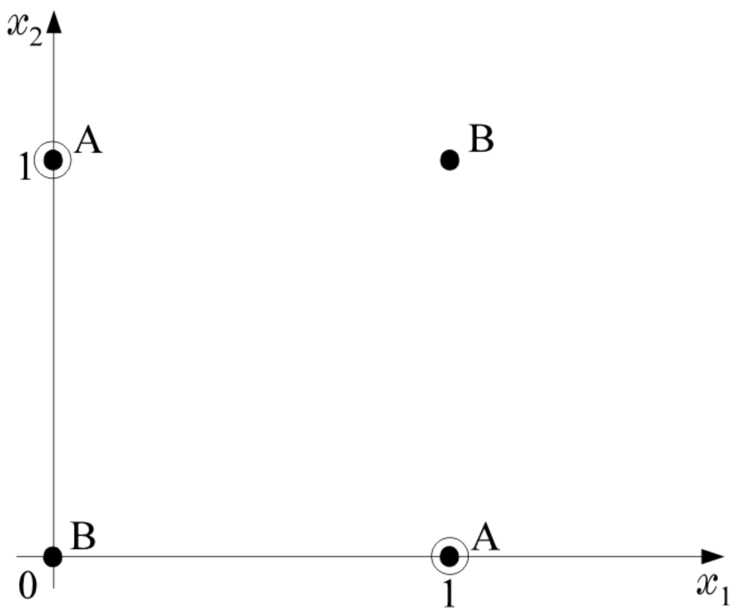
- AND and OR operations are linearly separable problems



The XOR Problem

- XOR is not linearly separable.

x_1	x_2	XOR	Class
0	0	0	B
0	1	1	A
1	0	1	A
1	1	0	B



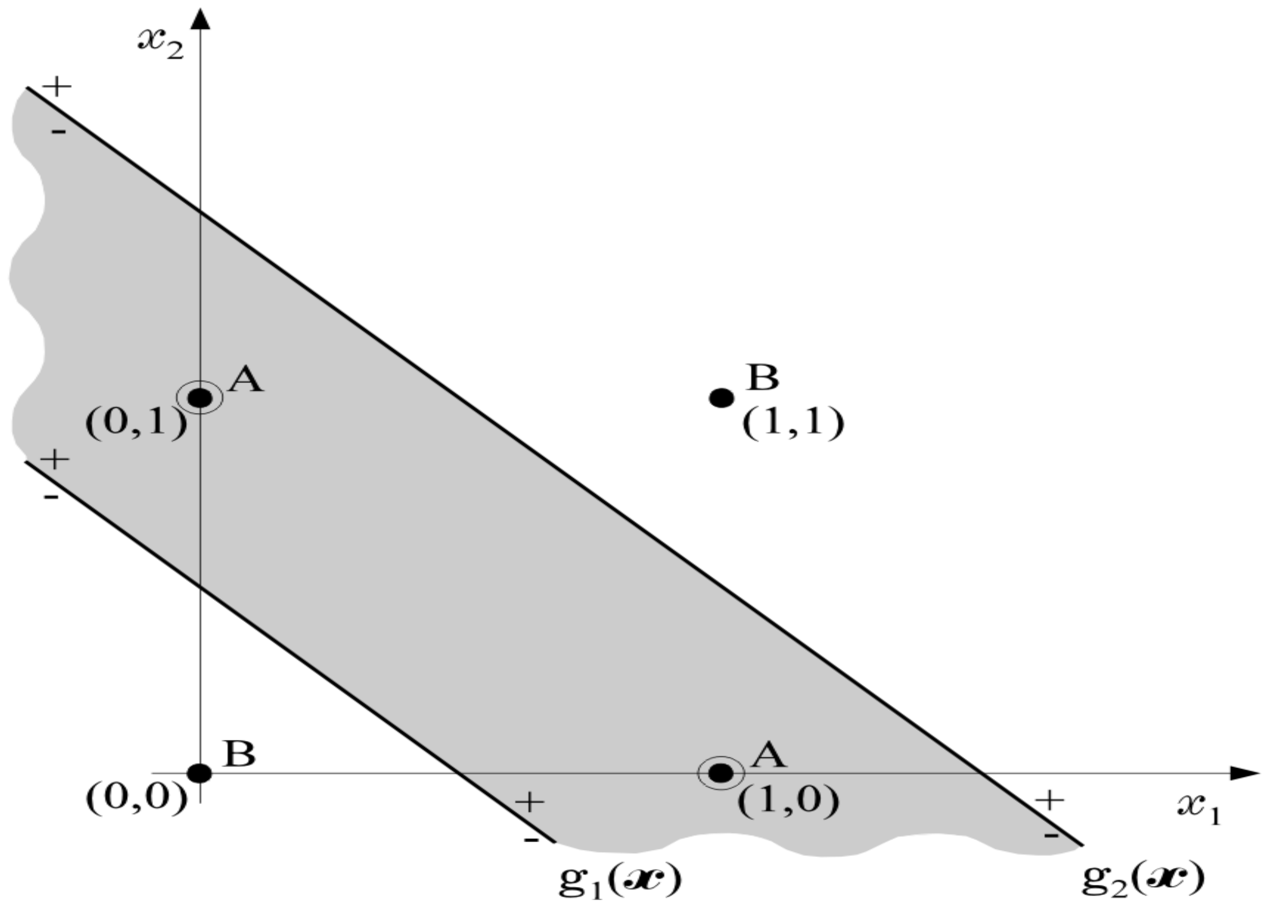
- How can we use linear classifiers to solve this problem?

Combining two linear classifiers

- Idea: use a logical combination of two linear classifiers.

$$g_2(x) = x_1 + x_2 - \frac{3}{2}$$

$$g_1(x) = x_1 + x_2 - \frac{1}{2}$$



Combining two linear classifiers

Let $f(x)$ be the unit step activation function:

$f(x) = 0, \;\; x < 0$

$f(x) = 1, \;\; x \geq 0$

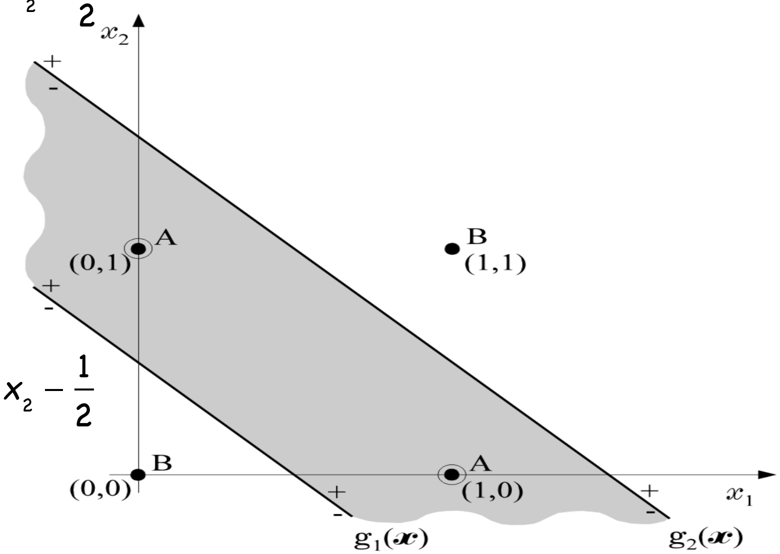
Observe that the classification problem is then solved by

$$f\left(y_1 - y_2 - \frac{1}{2}\right)$$

where

$y_1 = f\left(g_1(x)\right)$ and $y_2 = f\left(g_2(x)\right)$

$g_2(x) = x_1 + x_2 - \frac{3}{2}$



$g_1(x) = x_1 + x_2 - \frac{1}{2}$

Combining two linear classifiers

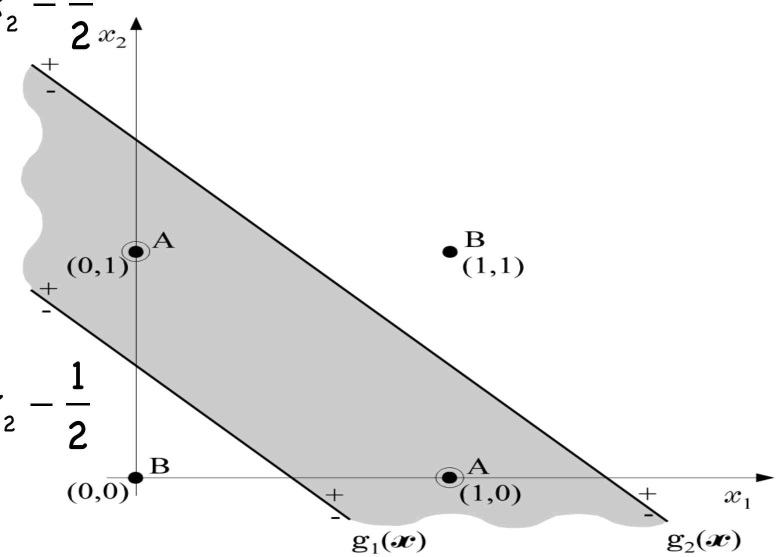
- This calculation can be implemented sequentially:
    1. Compute y_1 and y_2 from x_1 and x_2.
    2. Compute the decision from y_1 and y_2.
- Each layer in the sequence consists of one or more linear classifications.
- This is therefore a two-layer perceptron.

$$f\left(y_1 - y_2 - \frac{1}{2}\right)$$

where

$$y_1 = f\left(g_1(x)\right) \text{ and } y_2 = f\left(g_2(x)\right)$$

$$g_2(x) = x_1 + x_2 - \frac{3}{2}$$



$$g_1(x) = x_1 + x_2 - \frac{1}{2}$$

The Two-Layer Perceptron

Layer 1				Layer 2
x_1	x_2	y_1	y_2	
0	0	0(-)	0(-)	B(0)
0	1	1(+)	0(-)	A(1)
1	0	1(+)	0(-)	A(1)
1	1	1(+)	1(+)	B(0)

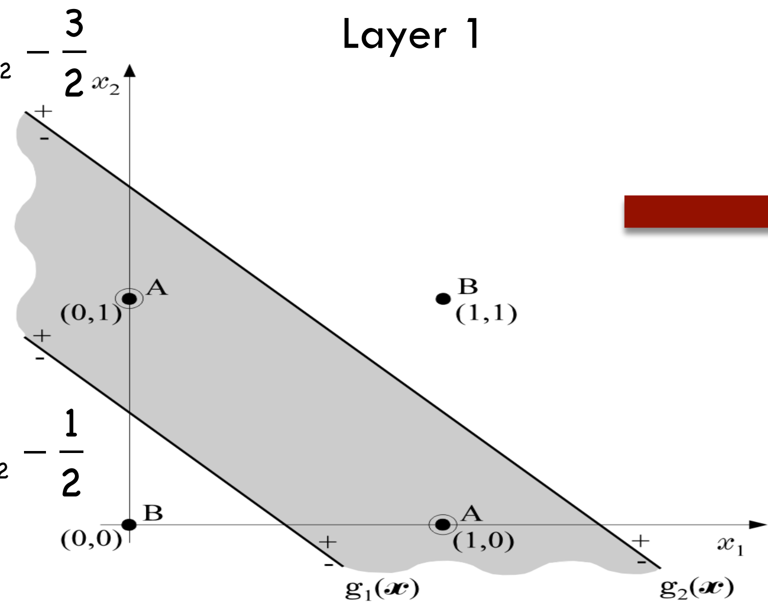
$$f\left(y_1 - y_2 - \frac{1}{2}\right)$$

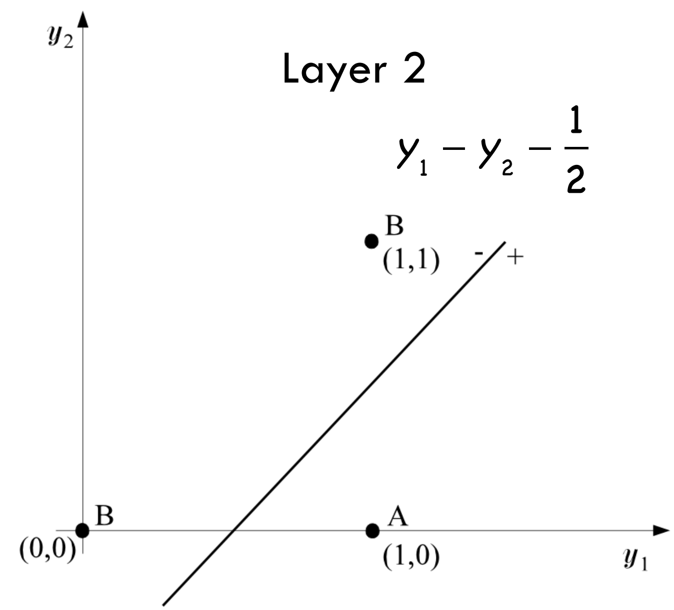
where

$$y_1 = f\left(g_1(x)\right) \text{ and } y_2 = f\left(g_2(x)\right)$$

$$g_2(x) = x_1 + x_2 - \frac{3}{2}$$

$$g_1(x) = x_1 + x_2 - \frac{1}{2}$$





The Two-Layer Perceptron

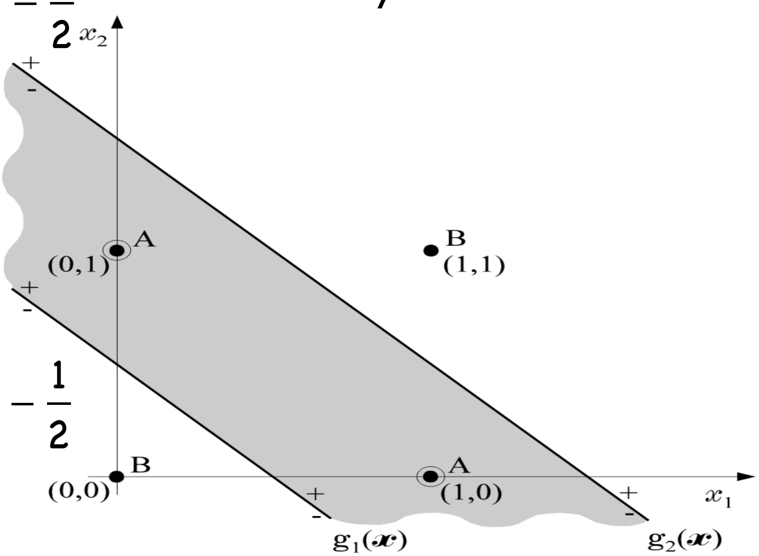
- The first layer performs a nonlinear mapping that makes the data linearly separable.

$$y_1 = f\left(g_1(x)\right) \text{ and } y_2 = f\left(g_2(x)\right)$$

Layer 1

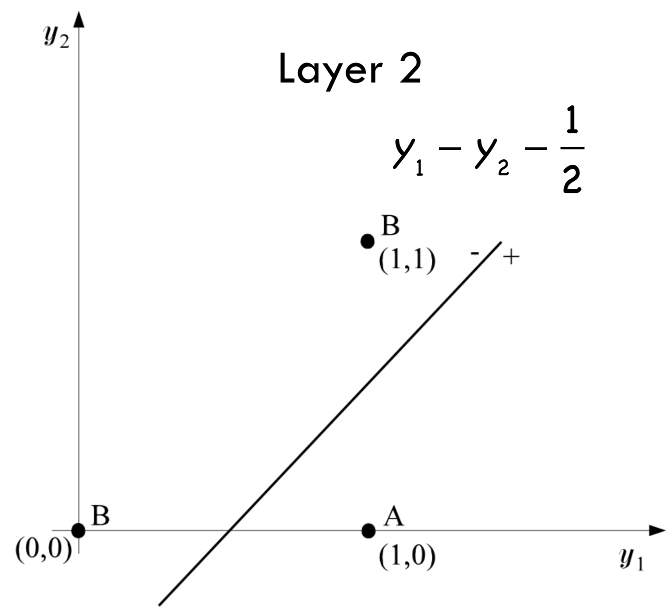
$$g_2(x) = x_1 + x_2 - \frac{3}{2}$$

$$g_1(x) = x_1 + x_2 - \frac{1}{2}$$

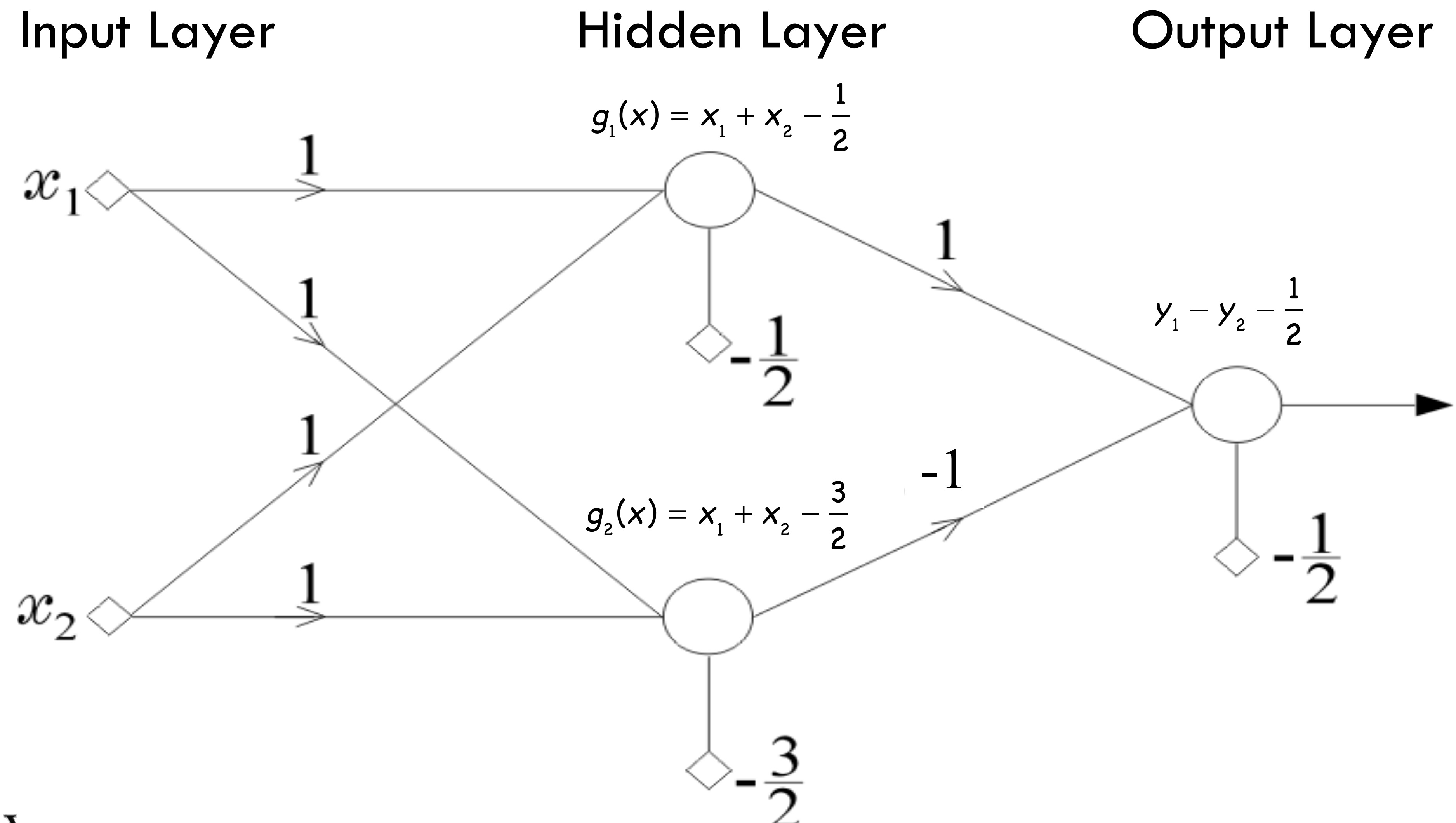


Layer 2

$$y_1 - y_2 - \frac{1}{2}$$



The Two-Layer Perceptron Architecture



The Two-Layer Perceptron

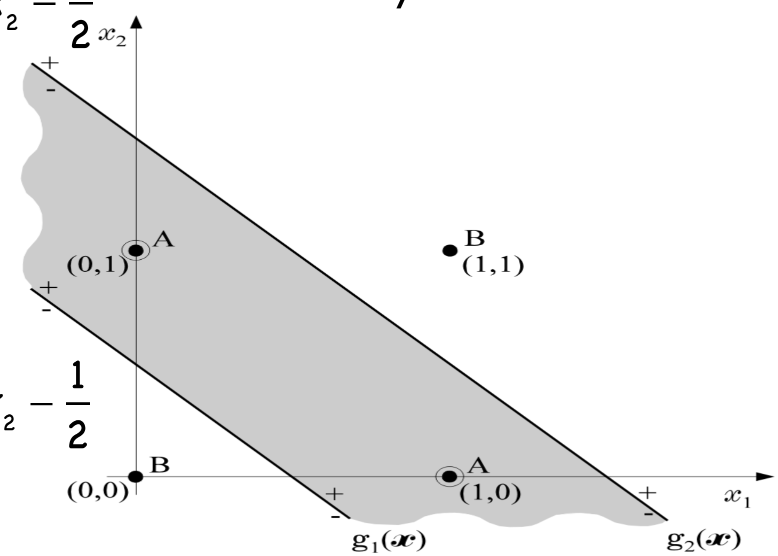
- Note that the hidden layer maps the plane onto the vertices of a unit square.

$$y_1 = f\left(g_1(x)\right) \text{ and } y_2 = f\left(g_2(x)\right)$$

Layer 1

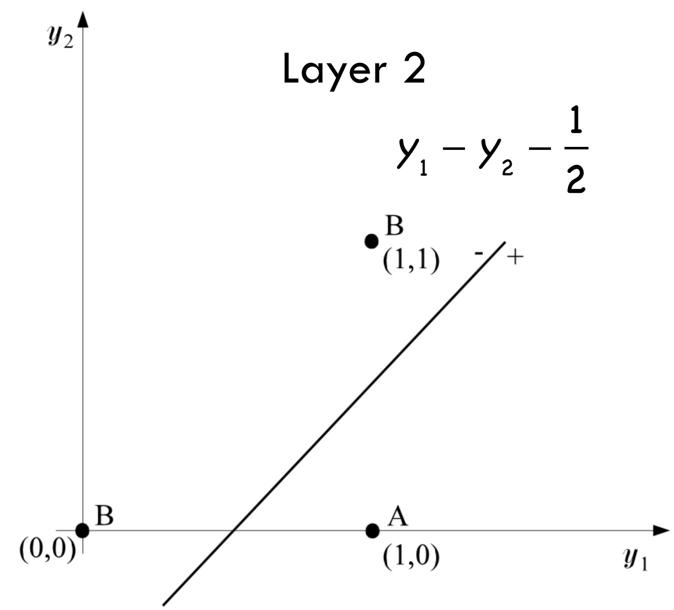
$$g_2(x) = x_1 + x_2 - \frac{3}{2}$$

$$g_1(x) = x_1 + x_2 - \frac{1}{2}$$



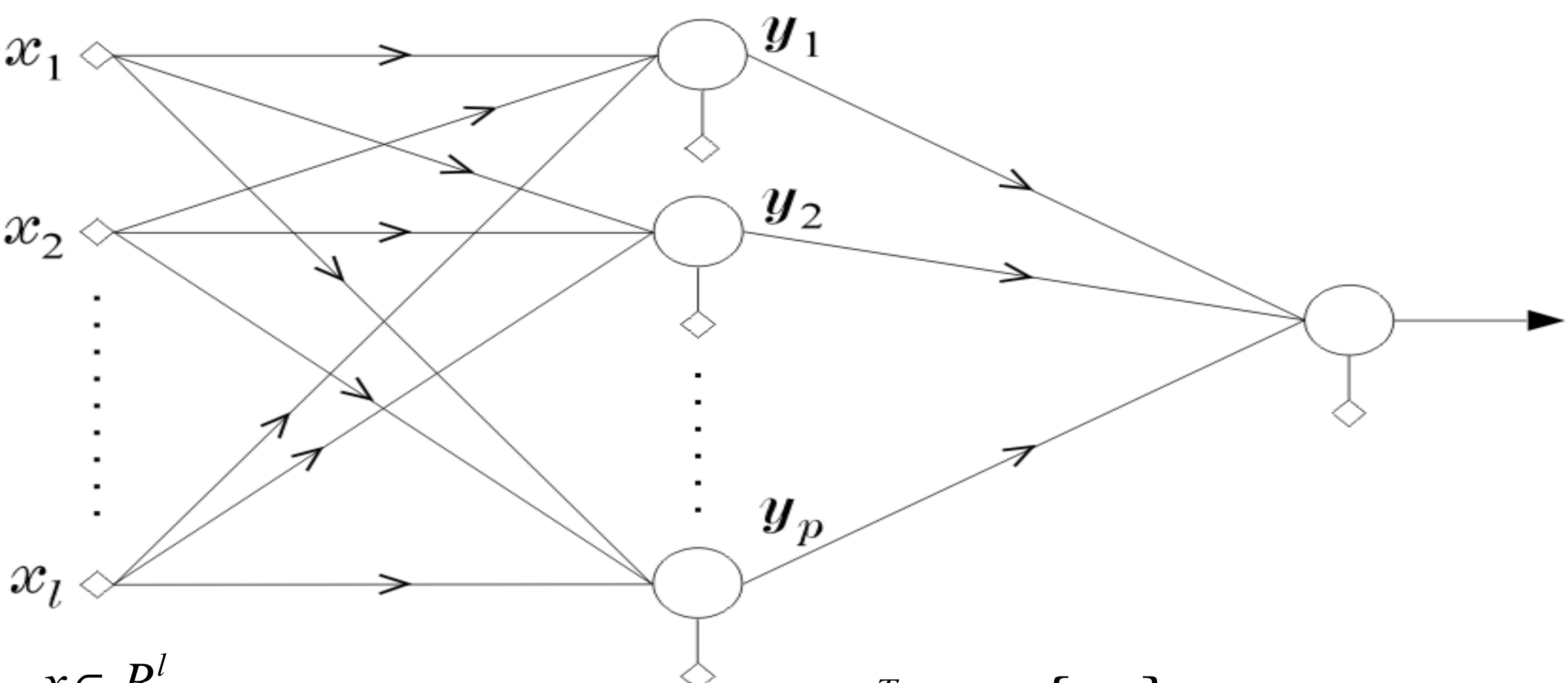
Layer 2

$$y_1 - y_2 - \frac{1}{2}$$



Higher Dimensions

- Each hidden unit realizes a hyperplane discriminant function.
- The output of each hidden unit is 0 or 1 depending upon the location of the input vector relative to the hyperplane.

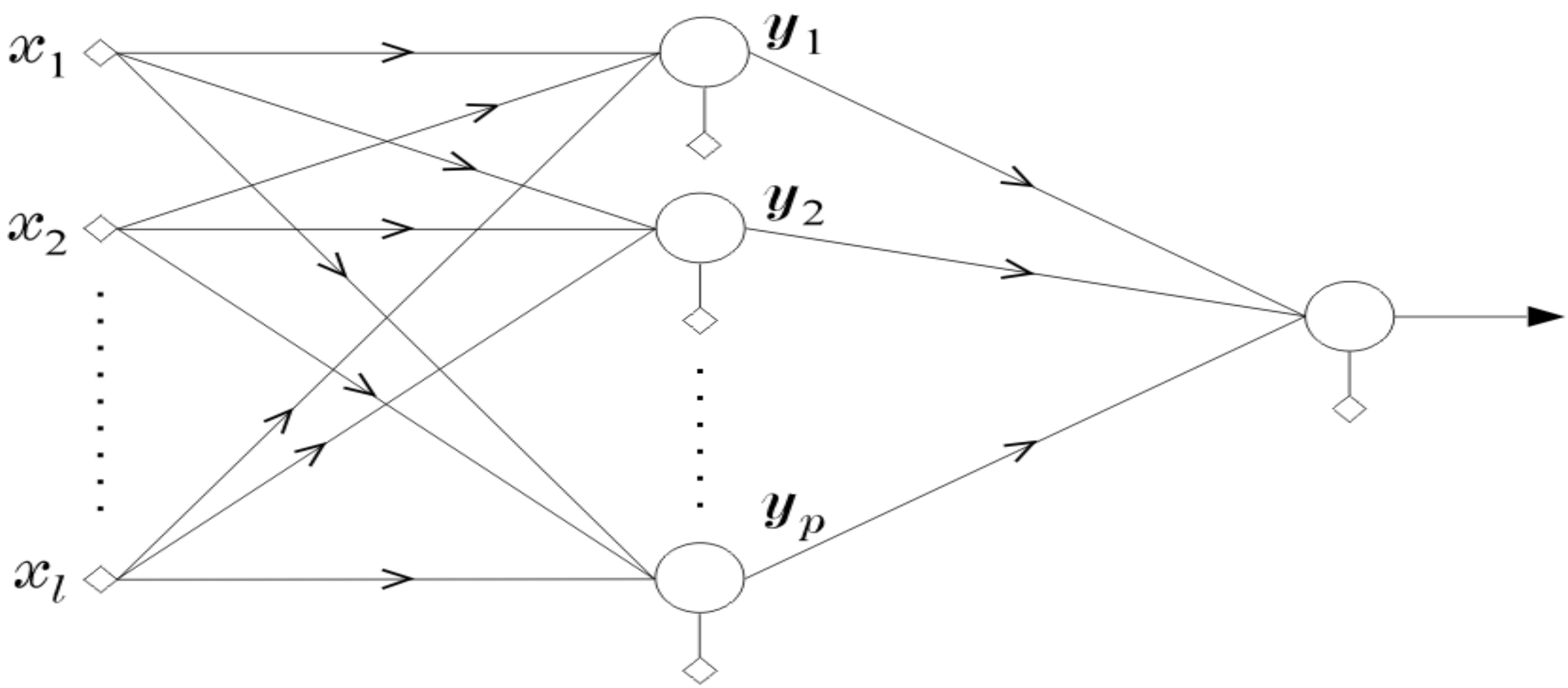


$\underline{x} \in R^l$

$$\underline{x} \rightarrow \underline{y} = [y_1, \ldots y_p]^T,\ y_i \in \{0,1\}\ i = 1, 2, \ldots p$$

Higher Dimensions

- Together, the hidden units map the input onto the vertices of a *p*-dimensional unit hypercube.

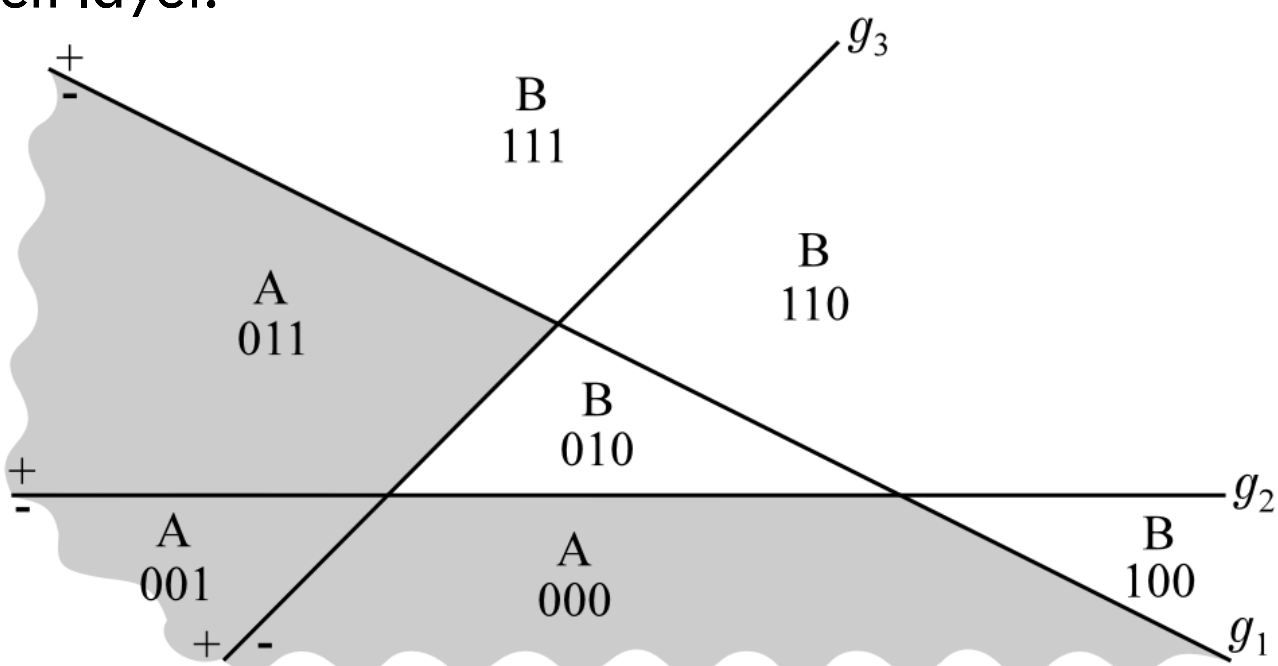


$\underline{x} \in R^l$

$$\underline{x} \rightarrow \underline{y} = [y_1, \ldots y_p]^T, \; y_i \in \{0,1\} \; i = 1, 2, \ldots p$$

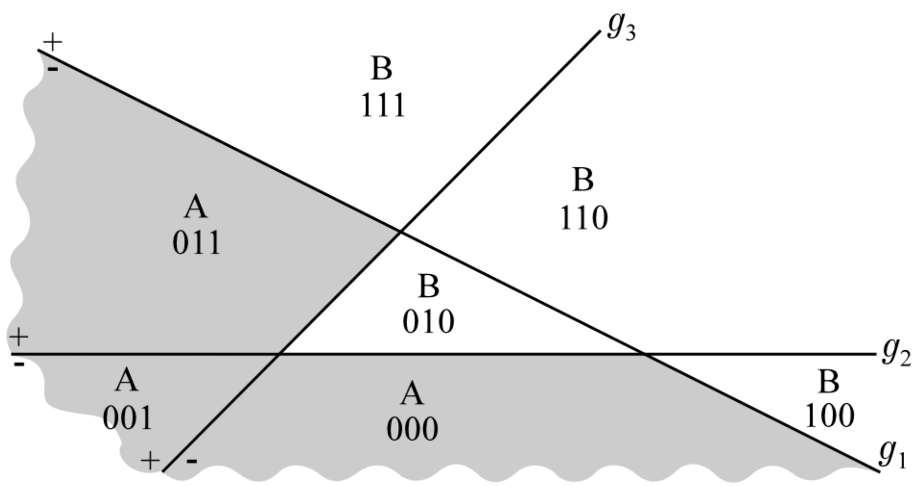
Two-Layer Perceptron

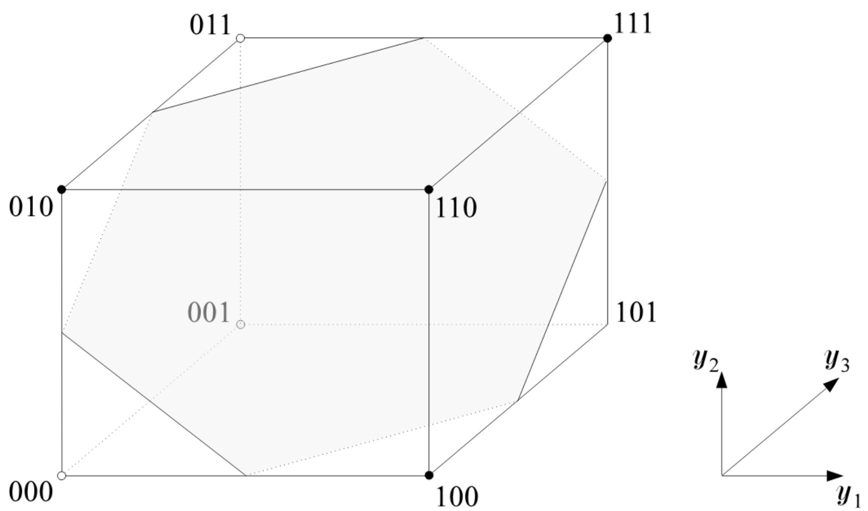
- These *p* hyperplanes partition the *l*-dimensional input space into polyhedral regions
- Each region corresponds to a different vertex of the *p*-dimensional hypercube represented by the outputs of the hidden layer.



Two-Layer Perceptron

- In this example, the vertex (0, 0, 1) corresponds to the region of the input space where:
  - $g_1(x) < 0$
  - $g_2(x) < 0$
  - $g_3(x) > 0$



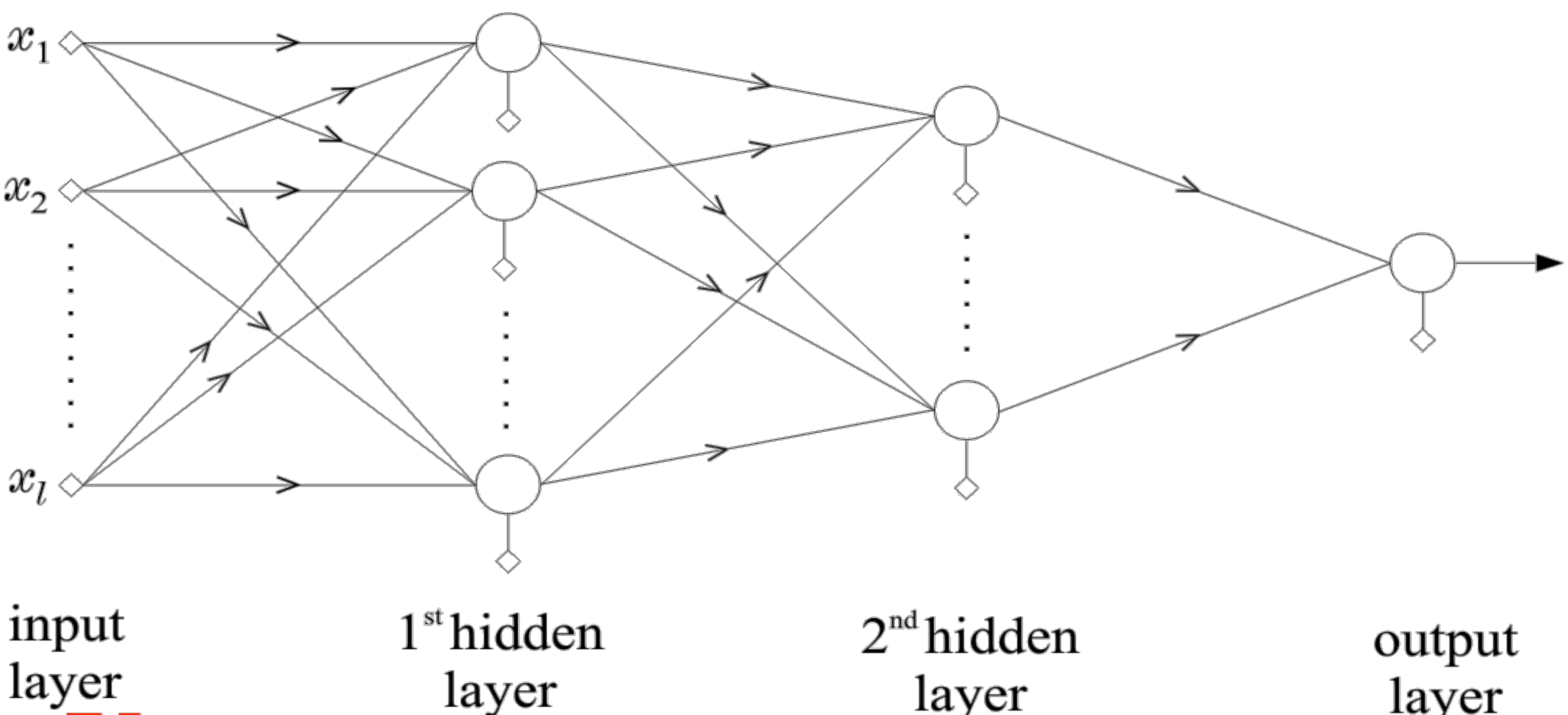


Limitations of a Two-Layer Perceptron

- The output neuron realizes a hyperplane in the transformed space that partitions the p vertices into two sets.
- Thus, the two layer perceptron has the capability to classify vectors into **classes that consist of unions of polyhedral regions**.
- But **NOT ANY** union. It depends on the relative position of the corresponding vertices.
- How can we solve this problem?

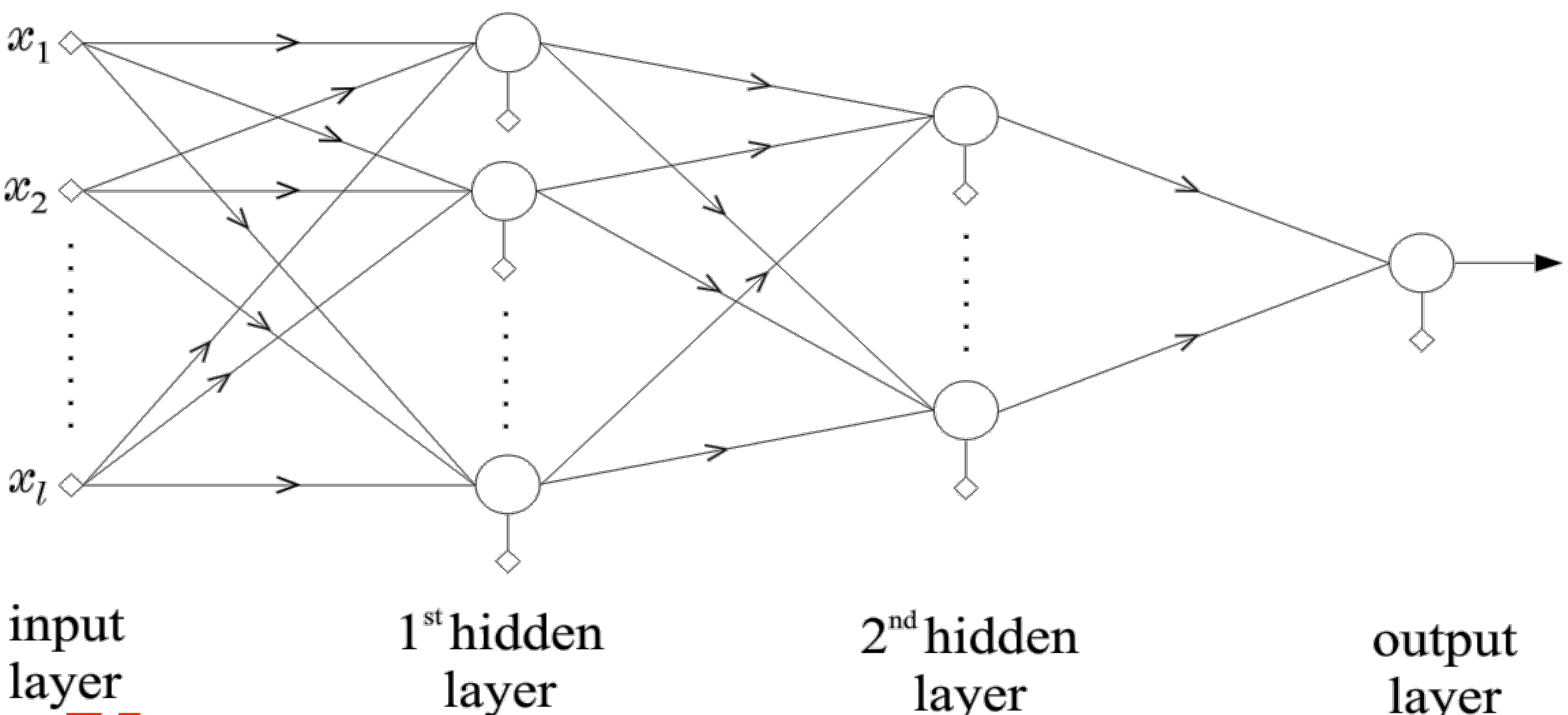
The Three-Layer Perceptron

- Suppose that Class A consists of the union of K polyhedra in the input space.
- Use K neurons in the 2nd hidden layer.
- Train each to classify one Class A vertex as positive, the rest negative.
- Now use an output neuron that implements the OR function.



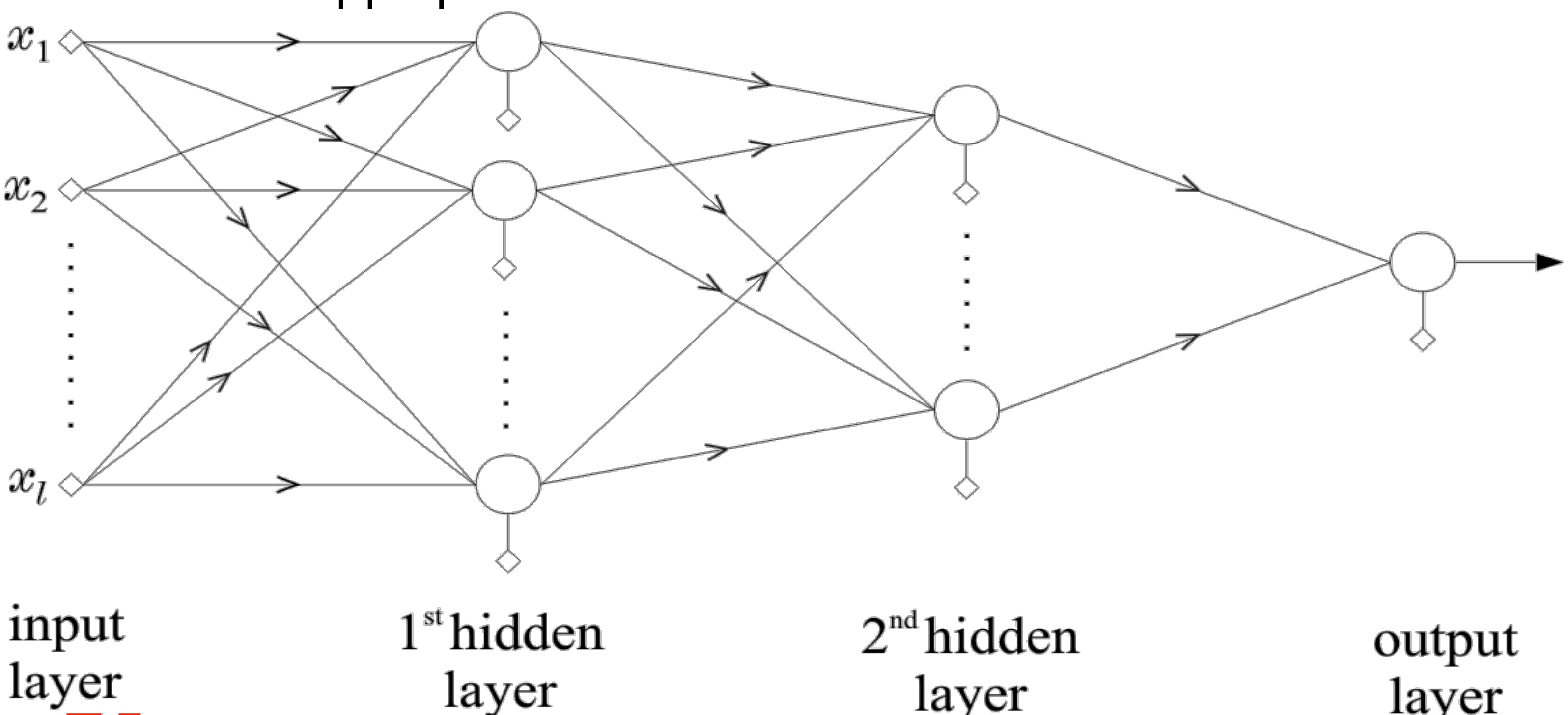
The Three-Layer Perceptron

- Thus the three-layer perceptron can separate classes resulting from any union of polyhedral regions in the input space.



The Three-Layer Perceptron

- The first layer of the network forms the **hyperplanes** in the input space.
- The second layer of the network forms the **polyhedral regions** of the input space
- The third layer forms the appropriate **unions of these regions** and maps each to the appropriate class.



Outline

- Combining Linear Classifiers
- **Learning Parameters**

Training Data

- The training data consist of N input-output pairs:

$$\left(\mathbf{y}(i), \mathbf{x}(i)\right), \qquad i \in 1, \ldots N$$

where

$$\mathbf{y}(i) = \left[y_1(i), \ldots, y_{k_L}(i) \right]^t$$

and

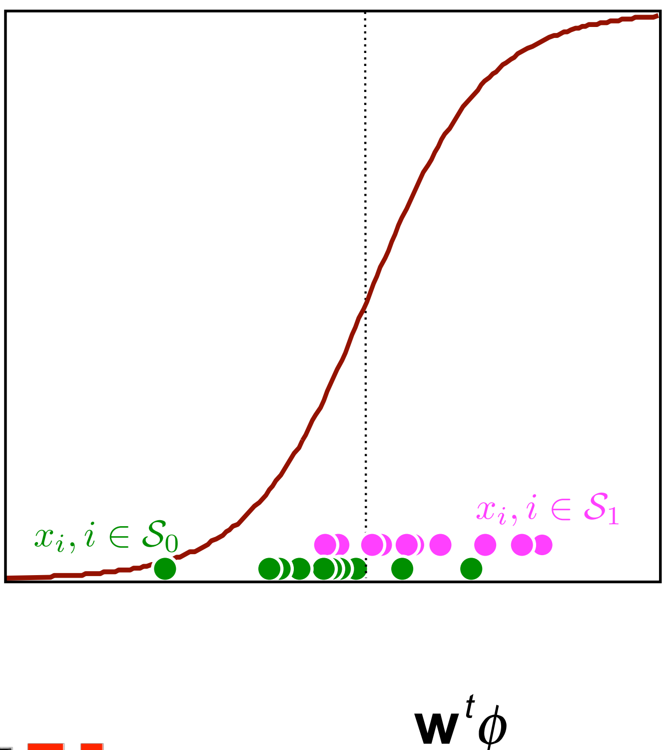
$$x(i) = \left[x_1(i), \ldots, x_{k_0}(i) \right]^t$$

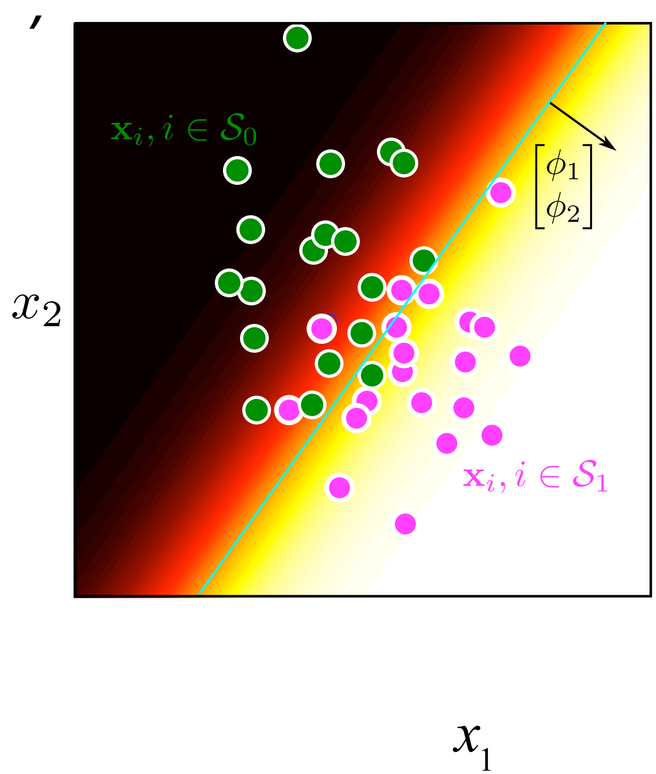
Choosing an Activation Function

- The unit step activation function means that the error rate of the network is a discontinuous function of the weights.
- This makes it difficult to learn optimal weights by minimizing the error.
- To fix this problem, we need to use a smooth activation function.
- A popular choice is the sigmoid function we used for logistic regression:

Smooth Activation Function

$$f(a) = \frac{1}{1+\exp(-a)}$$





Output: Two Classes

- For a binary classification problem, there is a single output node with activation function given by

$$f(a) = \frac{1}{1+\exp(-a)}$$

- Since the output is constrained to lie between 0 and 1, it can be interpreted as the probability of the input vector belonging to Class 1.

Output: K > 2 Classes

- For a *K*-class problem, we use *K* outputs, and the softmax function given by

$$y_k = \frac{\exp(a_k)}{\sum_j \exp(a_j)}$$

- Since the outputs are constrained to lie between 0 and 1, and sum to 1, y_k can be interpreted as the probability that the input vector belongs to Class *K*.

Non-Convex

- Now each layer of our multi-layer perceptron is a logistic regressor.
- Recall that optimizing the weights in logistic regression results in a convex optimization problem.
- Unfortunately the cascading of logistic regressors in the multi-layer perceptron makes the problem non-convex.
- This makes it difficult to determine an exact solution.
- Instead, we typically use **gradient descent** to find a locally optimal solution to the weights.
- The specific learning algorithm is called the **backpropagation** algorithm.

Nonlinear Classification and Regression: Outline

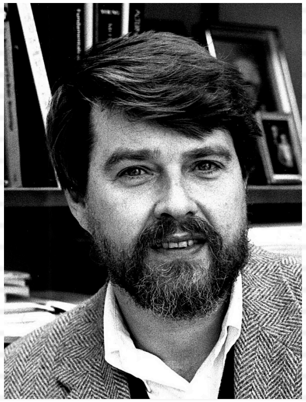
- Multi-Layer Perceptrons
  - **The Back-Propagation Learning Algorithm**
- Generalized Linear Models
  - Radial Basis Function Networks
  - Sparse Kernel Machines
    - Nonlinear SVMs and the Kernel Trick
    - Relevance Vector Machines

The Backpropagation Algorithm

Paul J. Werbos. Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. PhD thesis, Harvard University, 1974

Rumelhart, David E.; Hinton, Geoffrey E., Williams, Ronald J. (8 October 1986). "Learning representations by back-propagating errors". *Nature* **323** (6088): 533–536.

Werbos

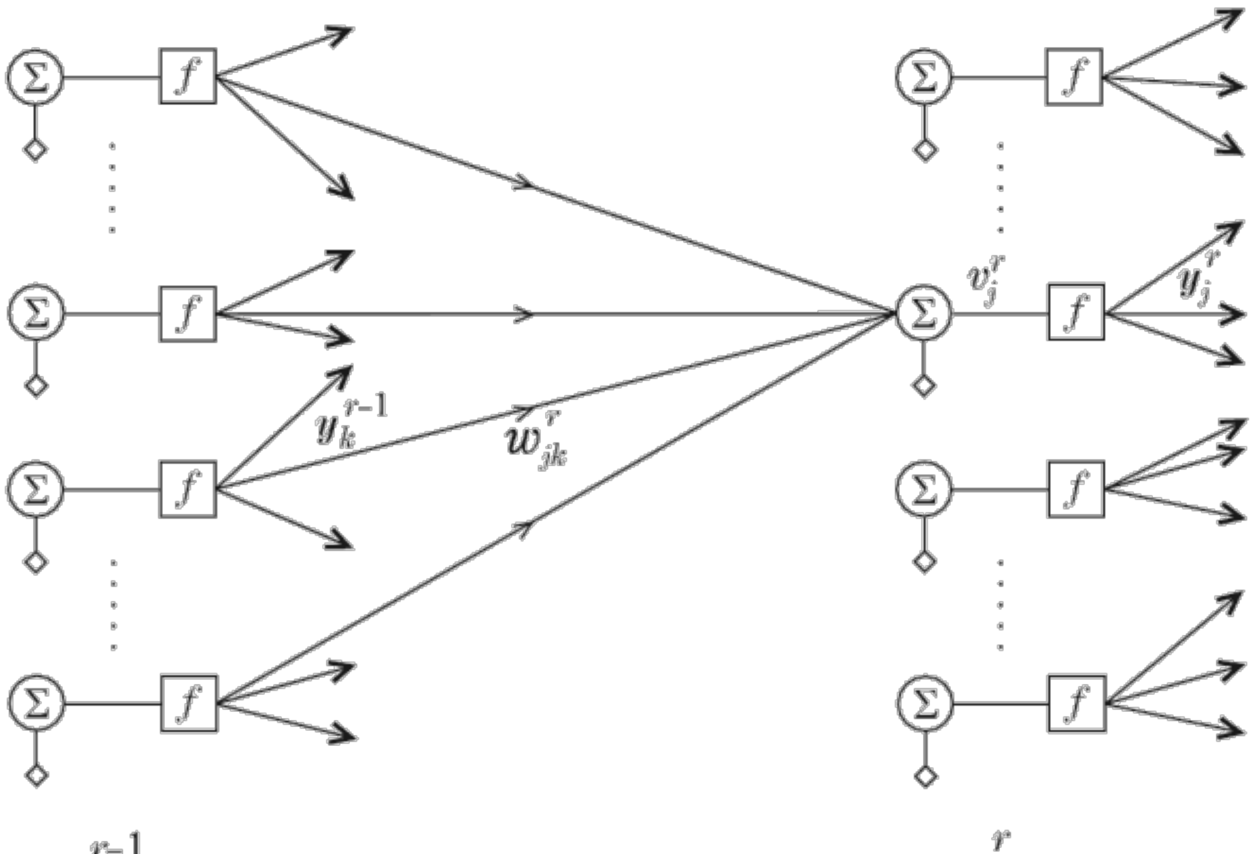


Rumelhart

Hinton

Notation

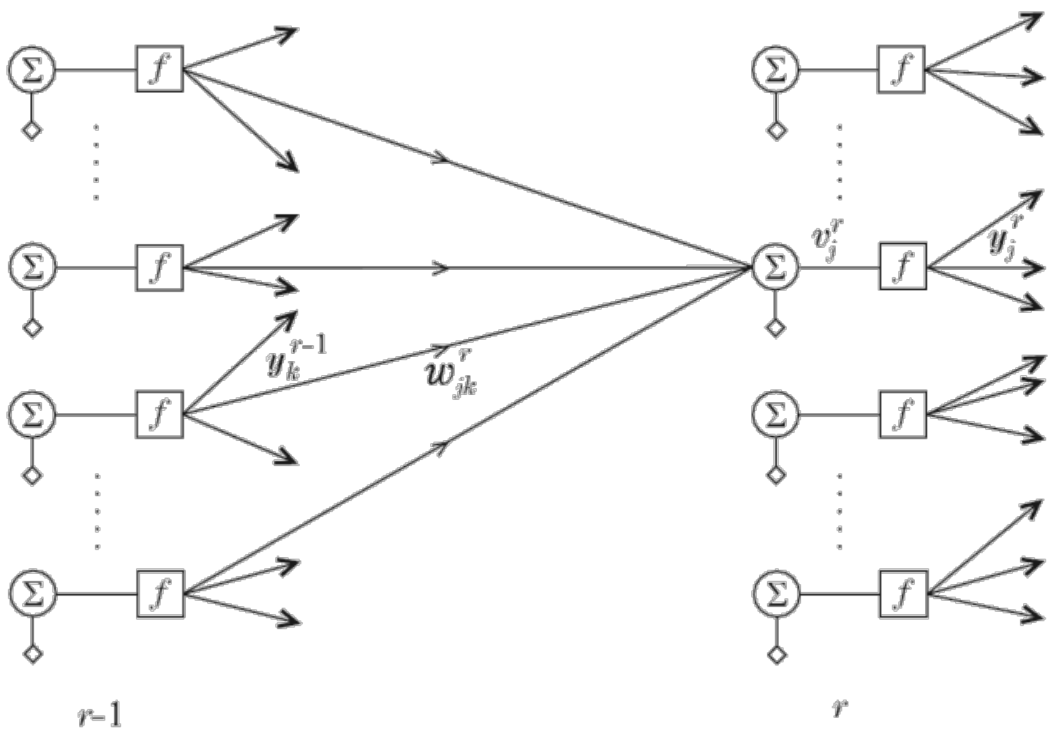
- Assume a network with *L* layers
  - k_0 nodes in the input layer.
  - k_r nodes in the r'th layer.



Notation

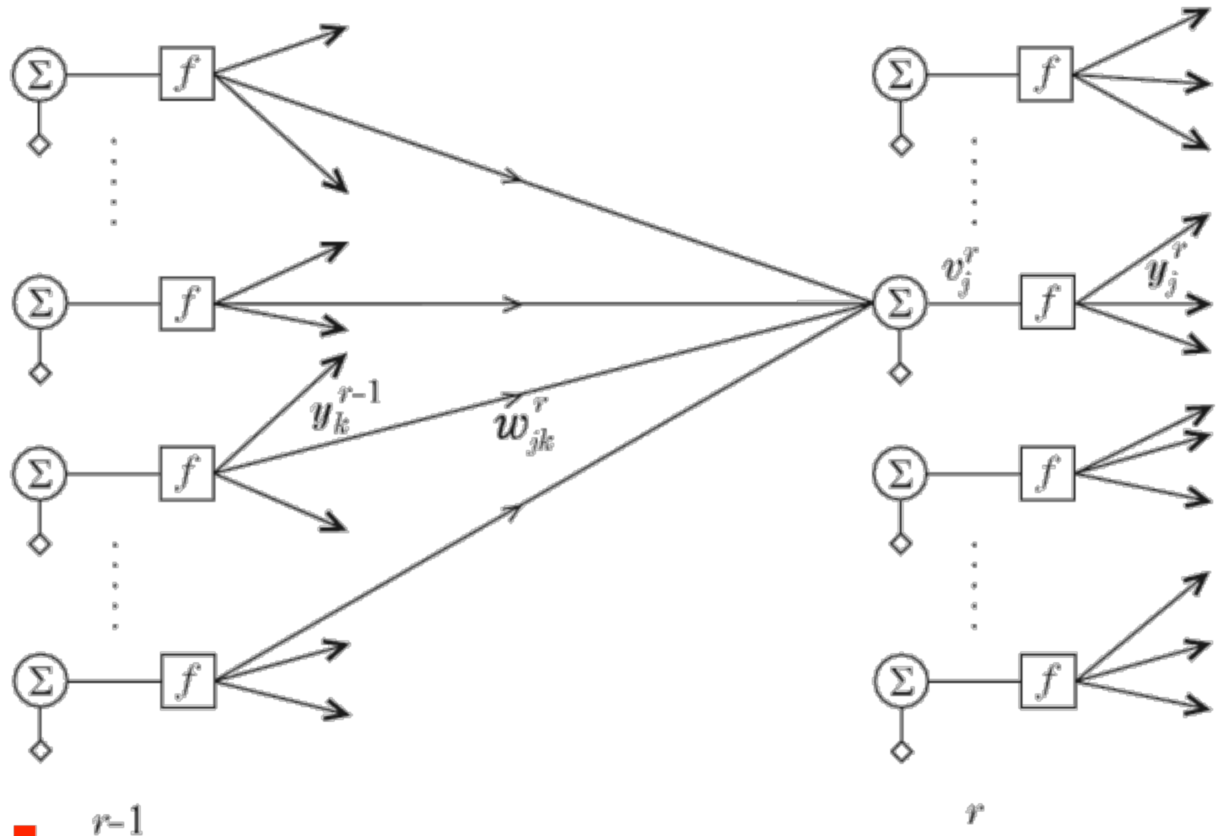
Let y_k^{r-1} be the output of the kth neuron of layer $r-1$.

Let w_{jk}^{r} be the weight of the synapse on the jth neuron of layer r from the kth neuron of layer $r-1$.



Input

$$y_k^0(i) = x_k(i),\ k = 1,\ldots,k_0$$



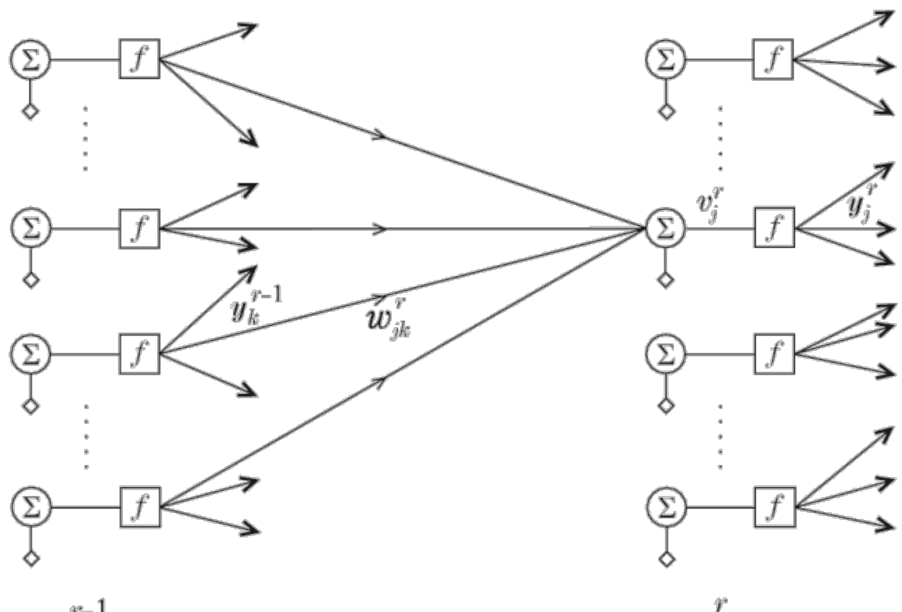
Notation

Let v_j^r be the total input to the jth neuron of layer r :

$$v_j^r(i) = \left(\mathbf{w}_j^r\right)^t \mathbf{y}^{r-1}(i) = \sum_{k=0}^{k_{r-1}} w_{jk}^r y_k^{r-1}(i)$$

where we define $y_0^r(i) = +1$ to incorporate the bias term.

Then $y_j^r(i) = f\left(v_j^r(i)\right) = f\left(\sum_{k=0}^{k_{r-1}} w_{jk}^r y_k^{r-1}(i)\right)$



Cost Function

- A common cost function is the squared error:

$$J = \sum_{i=1}^{N} \varepsilon(i)$$

where $\varepsilon(i) \triangleq \frac{1}{2}\sum_{m=1}^{k_L}\left(e_m(i)\right)^2 = \frac{1}{2}\sum_{m=1}^{k_L}\left(y_m(i) - \hat{y}_m(i)\right)^2$

and

$\hat{y}_m(i) = y_k^r(i)$ is the output of the network.

Cost Function

- To summarize, the error for input *i* is given by

$$\varepsilon(i) = \frac{1}{2}\sum_{m=1}^{k_L}\left(e_m(i)\right)^2 = \frac{1}{2}\sum_{m=1}^{k_L}\left(\hat{y}_m(i) - y_m(i)\right)^2$$

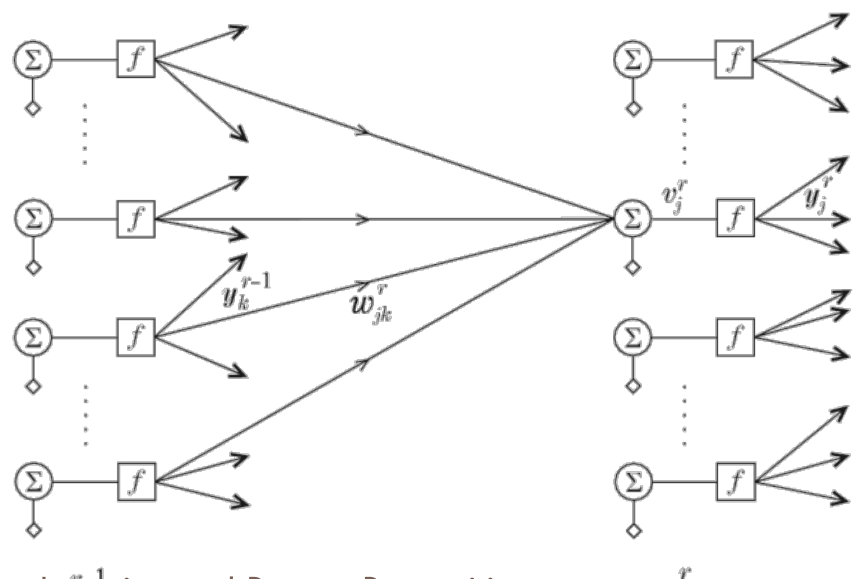
where $\hat{y}_m(i) = y_k^r(i)$ is the output of the output layer

and each layer is related to the previous layer through

$$y_j^r(i) = f\left(v_j^r(i)\right)$$

and

$$v_j^r(i) = \left(\mathbf{w}_j^r\right)^t \mathbf{y}^{r-1}(i)$$

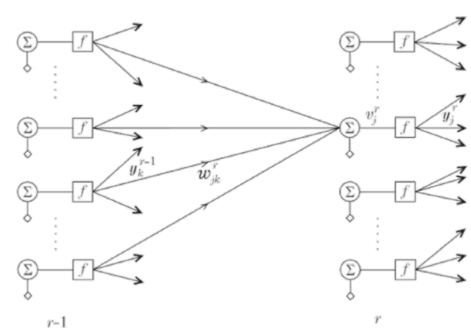


Gradient Descent

$$\varepsilon(i) = \frac{1}{2}\sum_{m=1}^{k_L}\left(e_m(i)\right)^2 = \frac{1}{2}\sum_{m=1}^{k_L}\left(\hat{y}_m(i) - y_m(i)\right)^2$$

- Gradient descent starts with an initial guess at the weights over all layers of the network.
- We then use these weights to compute the network output $\hat{y}(i)$ for each input vector **x**(i) in the training data.
- This allows us to calculate the error ε (i) for each of these inputs.
- Then, in order to minimize this error, we incrementally update the weights in the negative gradient direction:

$$\mathbf{w}_j^r(\text{new}) = w_j^r(\text{old}) - \mu\frac{\partial J}{\partial \mathbf{w}_j^r} = w_j^r(\text{old}) - \mu\sum_{i=1}^{N}\frac{\partial \varepsilon(i)}{\partial \mathbf{w}_j^r}$$



Gradient Descent

- Since $v_j^r(i) = \left(\mathbf{w}_j^r\right)^t \mathbf{y}^{r-1}(i)$,

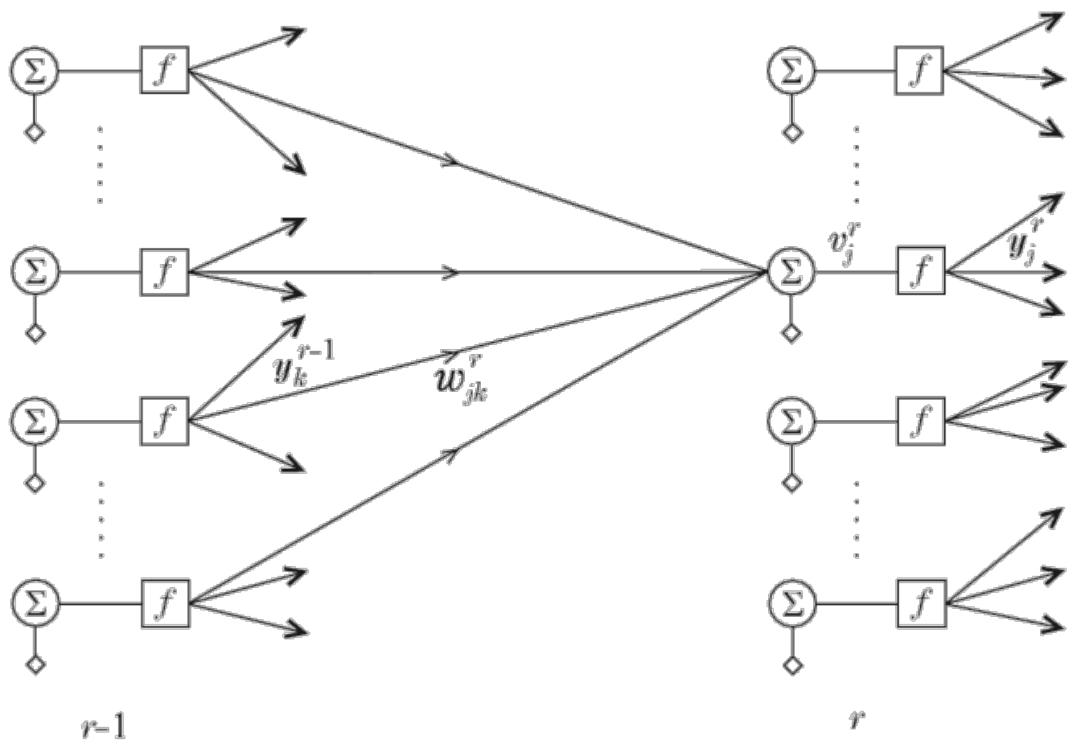
  the influence of the *j*th weight of the *r*th layer on the error can be expressed as:

$$\frac{\partial \varepsilon(i)}{\partial \mathbf{w}_j^r} = \frac{\partial \varepsilon(i)}{\partial v_j^r(i)} \frac{\partial v_j^r(i)}{\partial \mathbf{w}_j^r}$$

$$= \delta_j^r(i)\mathbf{y}^{r-1}(i)$$

where

$$\delta_j^r(i) \triangleq \frac{\partial \varepsilon(i)}{\partial v_j^r(i)}$$

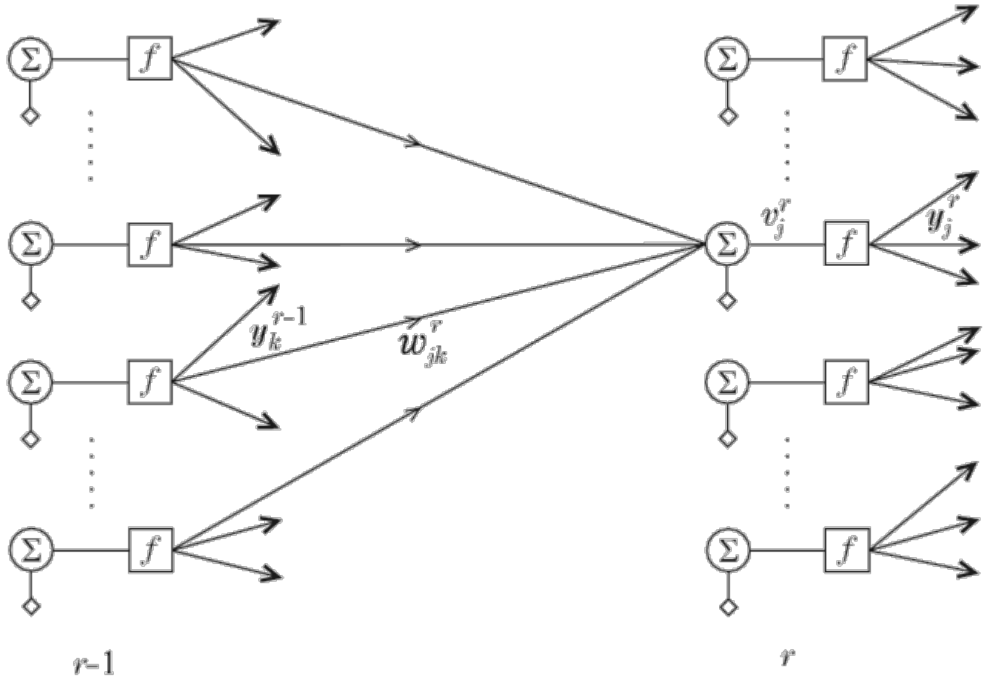


Gradient Descent

$$\frac{\partial \varepsilon(i)}{\partial \mathbf{w}_j^r} = \delta_j^r(i)\mathbf{y}^{r-1}(i),$$

where

$$\delta_j^r(i) \triangleq \frac{\partial \varepsilon(i)}{\partial v_j^r(i)}$$



For an intermediate layer r,

we cannot compute $\delta_j^r(i)$ directly.

However, $\delta_j^r(i)$ can be computed inductively,

starting from the output layer.

Backpropagation: The Output Layer

$$\frac{\partial \varepsilon(i)}{\partial \mathbf{w}_j^r} = \delta_j^r(i)\mathbf{y}^{r-1}(i), \quad \text{where} \quad \delta_j^r(i) \triangleq \frac{\partial \varepsilon(i)}{\partial v_j^r(i)}$$

and $\varepsilon(i) = \frac{1}{2}\sum_{m=1}^{k_L}\left(e_m(i)\right)^2 = \frac{1}{2}\sum_{m=1}^{k_L}\left(\hat{y}_m(i) - y_m(i)\right)^2$

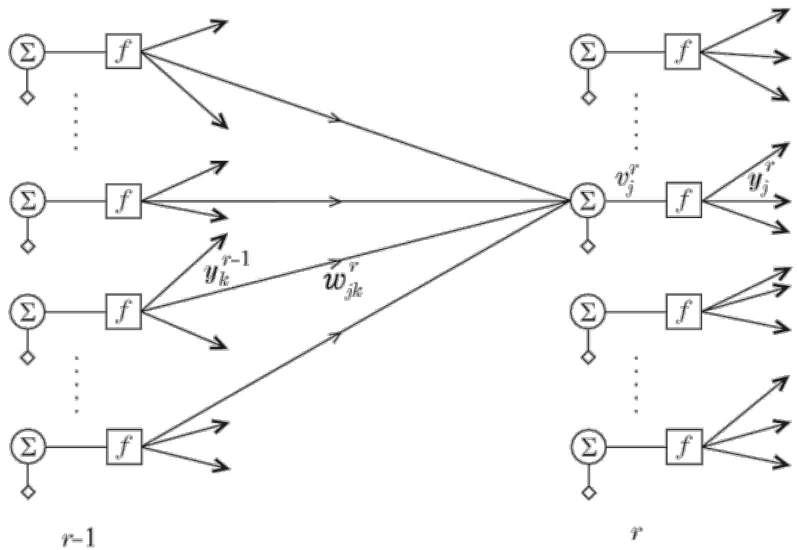
Recall that $\hat{y}_m(i) = y_j^L(i) = f\left(v_j^L(i)\right)$

Thus at the output layer we have

$$\delta_j^L(i) = \frac{\partial \varepsilon(i)}{\partial v_j^L(i)} = \frac{\partial \varepsilon(i)}{\partial e_j^L(i)}\frac{\partial e_j^L(i)}{\partial v_j^L(i)} = e_j^L(i)f'\left(v_j^L(i)\right)$$

$$f(a) = \frac{1}{1+\exp(-a)} \rightarrow f'(a) = f(a)\left(1 - f(a)\right)$$

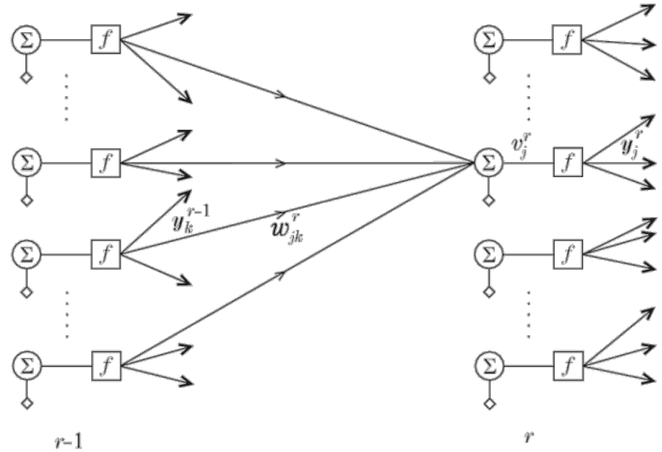
$$\boxed{\delta_j^L(i) = e_j^L(i)f\left(v_j^L(i)\right)\left(1 - f\left(v_j^L(i)\right)\right)}$$



Backpropagation: Hidden Layers

- Observe that the dependence of the error on the total input to a neuron in a previous layer can be expressed in terms of the dependence on the total input of neurons in the following layer:

$$\delta_j^{r-1}(i) = \frac{\partial \varepsilon(i)}{\partial v_j^{r-1}(i)} = \sum_{k=1}^{k_r} \frac{\partial \varepsilon(i)}{\partial v_k^r(i)} \frac{\partial v_k^r(i)}{\partial v_j^{r-1}(i)} = \sum_{k=1}^{k_r} \delta_k^r(i) \frac{\partial v_k^r(i)}{\partial v_j^{r-1}(i)}$$



where $v_k^r(i) = \sum_{m=0}^{k_{r-1}} w_{km}^r y_m^{r-1}(i) = \sum_{m=0}^{k_{r-1}} w_{km}^r f\left(v_m^{r-1}(i)\right)$

Thus we have $\dfrac{\partial v_k^r(i)}{\partial v_j^{r-1}(i)} = w_{kj}^r f'\left(v_j^{r-1}(i)\right)$

and so $\delta_j^{r-1}(i) = \dfrac{\partial \varepsilon(i)}{\partial v_j^{r-1}(i)} = f'\left(v_j^{r-1}(i)\right) \sum_{k=1}^{k_r} \delta_k^r(i) w_{kj}^r = f\left(v_j^L(i)\right)\left(1 - f\left(v_j^L(i)\right)\right) \sum_{k=1}^{k_r} \delta_k^r(i) w_{kj}^r$

Thus once the $\delta_k^r(i)$ are determined they can be propagated backward to calculate $\delta_j^{r-1}(i)$ using this inductive formula.

Backpropagation: Summary of Algorithm

Repeat until convergence

1. Initialization
   - Initialize all weights with small random values
2. Forward Pass
   - For each input vector, run the network in the forward direction, calculating:

     $$v_j^r(i) = \left(\mathbf{w}_j^r\right)^t \mathbf{y}^{r-1}(i); \qquad y_j^r(i) = f\left(v_j^r(i)\right)$$

     and finally $\varepsilon(i) = \frac{1}{2}\sum_{m=1}^{k_L}\left(e_m(i)\right)^2 = \frac{1}{2}\sum_{m=1}^{k_L}\left(\hat{y}_m(i) - y_m(i)\right)^2$
3. Backward Pass
   - Starting with the output layer, use our inductive formula to compute the $\delta_j^{r-1}(i)$:
     - Output Layer (Base Case): $\delta_j^L(i) = e_j^L(i)f'\left(v_j^L(i)\right)$
     - Hidden Layers (Inductive Case): $\delta_j^{r-1}(i) = f'\left(v_j^{r-1}(i)\right)\sum_{k=1}^{k_r}\delta_k^r(i)w_{kj}^r$
4. Update Weights

$$\mathbf{w}_j^r(\text{new}) = \mathbf{w}_j^r(\text{old}) - \mu\sum_{i=1}^{N}\frac{\partial\varepsilon(i)}{\partial\mathbf{w}_j^r} \quad \text{where} \quad \frac{\partial\varepsilon(i)}{\partial\mathbf{w}_j^r} = \delta_j^r(i)\mathbf{y}^{r-1}(i)$$

Batch vs Online Learning

- As described, on each iteration backprop updates the weights based upon all of the training data. This is called **batch learning**.

$$\mathbf{w}_j^r(\text{new}) = \mathbf{w}_j^r(\text{old}) - \mu \sum_{i=1}^{N} \frac{\partial \varepsilon(i)}{\partial \mathbf{w}_j^r} \quad \text{where} \quad \frac{\partial \varepsilon(i)}{\partial \mathbf{w}_j^r} = \delta_j^r(i)\mathbf{y}^{r-1}(i)$$

- An alternative is to update the weights after each training input has been processed by the network, based only upon the error for that input. This is called **online learning**.

$$\mathbf{w}_j^r(\text{new}) = \mathbf{w}_j^r(\text{old}) - \mu \frac{\partial \varepsilon(i)}{\partial \mathbf{w}_j^r} \quad \text{where} \quad \frac{\partial \varepsilon(i)}{\partial \mathbf{w}_j^r} = \delta_j^r(i)\mathbf{y}^{r-1}(i)$$

Batch vs Online Learning

- One advantage of batch learning is that averaging over all inputs when updating the weights should lead to smoother convergence.
- On the other hand, the randomness associated with online learning might help to prevent convergence toward a local minimum.
- Changing the order of presentation of the inputs from epoch to epoch may also improve results.

Remarks

- Local Minima
  - The objective function is in general non-convex, and so the solution may not be globally optimal.
- Stopping Criterion
  - Typically stop when the change in weights or the change in the error function falls below a threshold.
- Learning Rate
  - The speed and reliability of convergence depends on the learning rate μ.