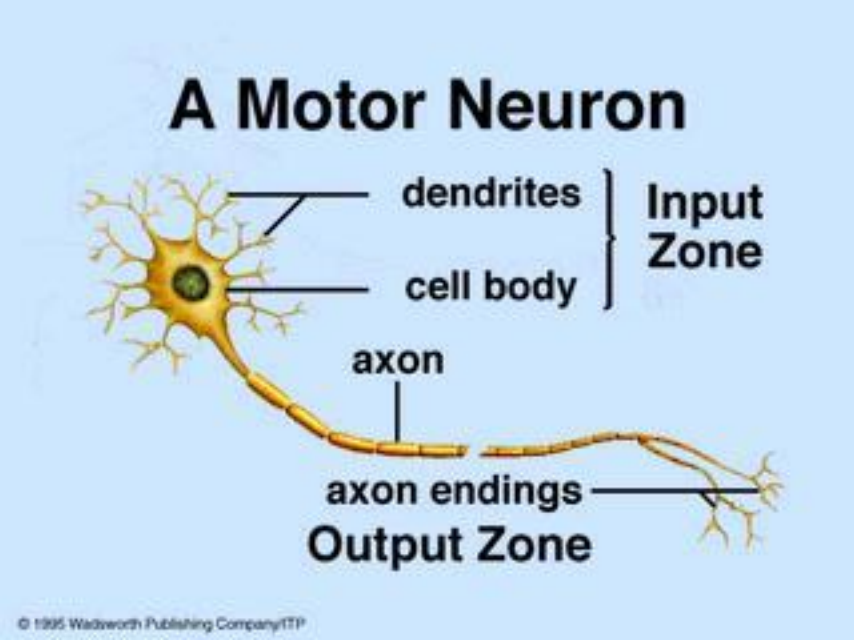
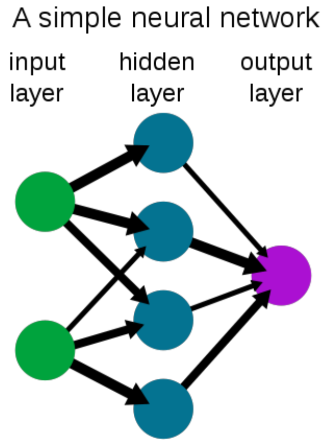
VBM683
Machine Learning

Pinar Duygulu

Slides are adapted from
Dhruv Batra (Virginia Tech),
J. Elder

New Topic: Neural Networks





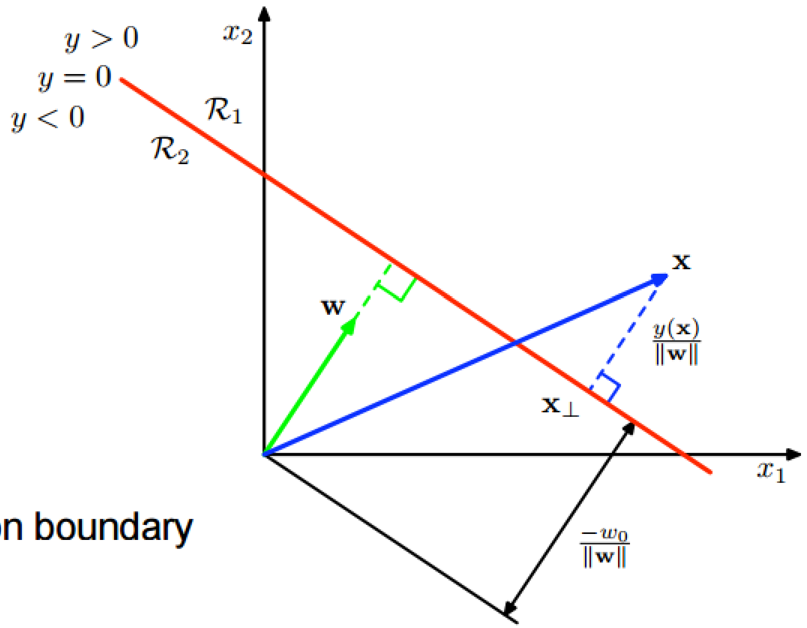
Two class discriminant function

$y(\mathbf{x}) = \mathbf{w}^t\mathbf{x} + w_0$

$y(\mathbf{x}) \geq 0 \rightarrow \mathbf{x}$ assigned to C_1

$y(\mathbf{x}) < 0 \rightarrow \mathbf{x}$ assigned to C_2

Thus $y(\mathbf{x}) = 0$ defines the decision boundary



Two class discriminant function

$y(\mathbf{x}) = \mathbf{w}^t\mathbf{x} + w_0$

$y(\mathbf{x}) \geq 0 \rightarrow \mathbf{x}$ assigned to C_1

$y(\mathbf{x}) < 0 \rightarrow \mathbf{x}$ assigned to C_2

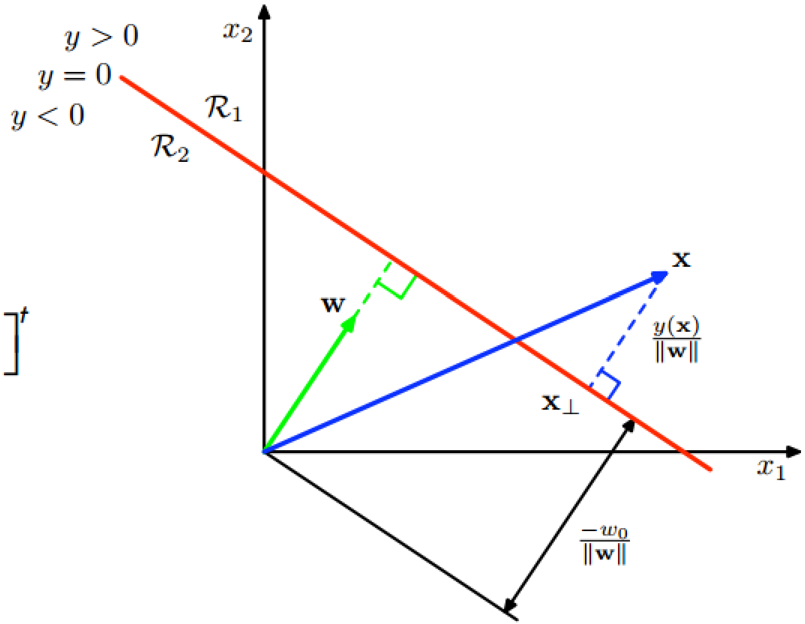
For convenience, let

$\mathbf{w} = \left[w_1 \ldots w_M\right]^t \Rightarrow \left[w_0 \;\; w_1 \ldots w_M\right]^t$

and

$\mathbf{x} = \left[x_1 \ldots x_M\right]^t \Rightarrow \left[1 \;\; x_1 \ldots x_M\right]^t$

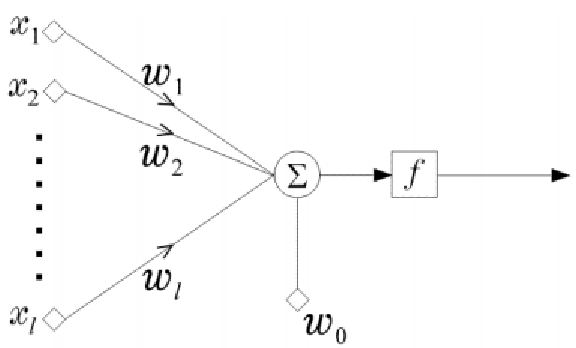
So we can express $y(\mathbf{x}) = \mathbf{w}^t\mathbf{x}$



Perceptron

$$y(\mathbf{x}) = f\left(\mathbf{w}^t\mathbf{x} + w_0\right) \qquad \begin{array}{l} y(\mathbf{x}) \geq 0 \rightarrow \mathbf{x} \text{ assigned to } C_1 \\ y(\mathbf{x}) < 0 \rightarrow \mathbf{x} \text{ assigned to } C_2 \end{array}$$

- A classifier based upon this simple generalized linear model is called a (single layer) **perceptron.**
- It can also be identified with an abstracted model of a neuron called the McCulloch Pitts model.



Synonyms

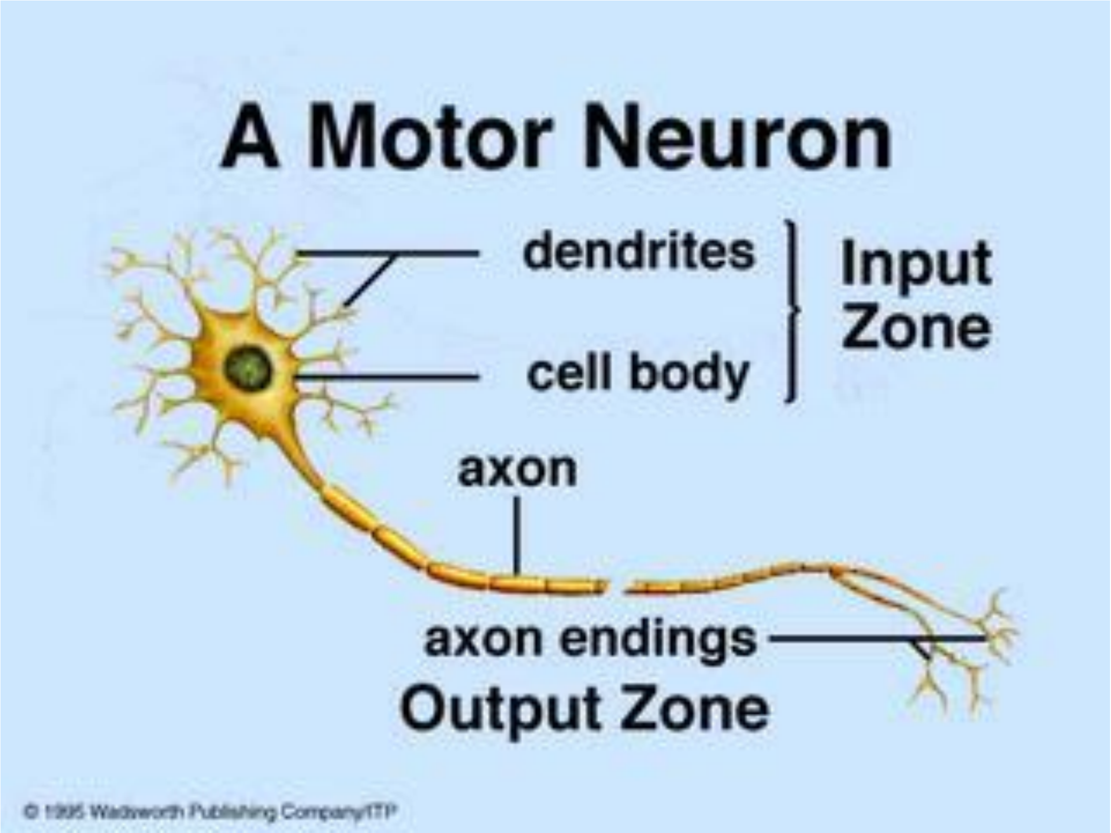
- Neural Networks

- Artificial Neural Network (ANN)
- Feed-forward Networks
- Multilayer Perceptrons (MLP)

- Types of ANN
  - Convolutional Nets
  - Autoencoders
  - Recurrent Neural Nets

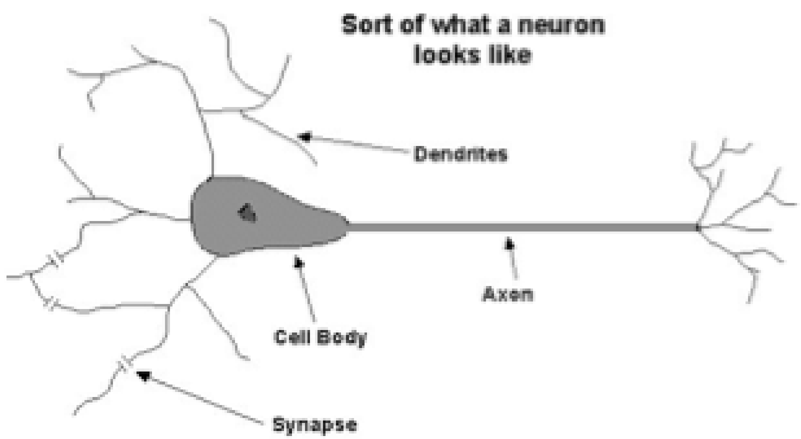
- [Back with a new name]: Deep Nets / Deep Learning

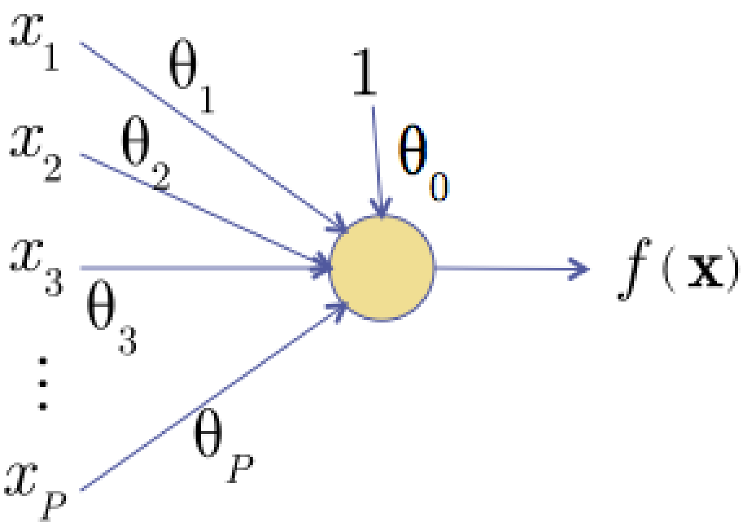
Biological Neuron



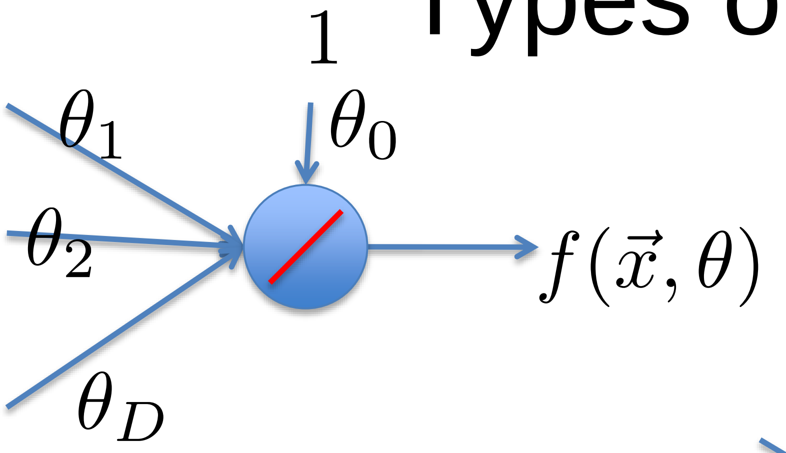
The Neuron Metaphor

- Neurons
  - accept information from multiple inputs,
  - transmit information to other neurons.
- Multiply inputs by weights along edges
- Apply some function to the set of inputs at each node

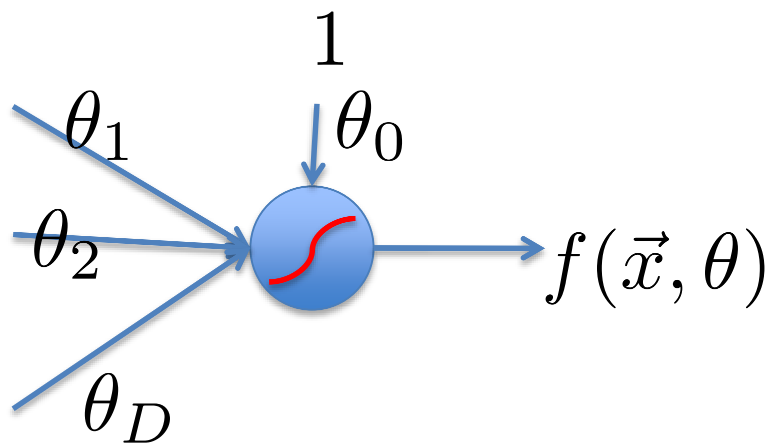




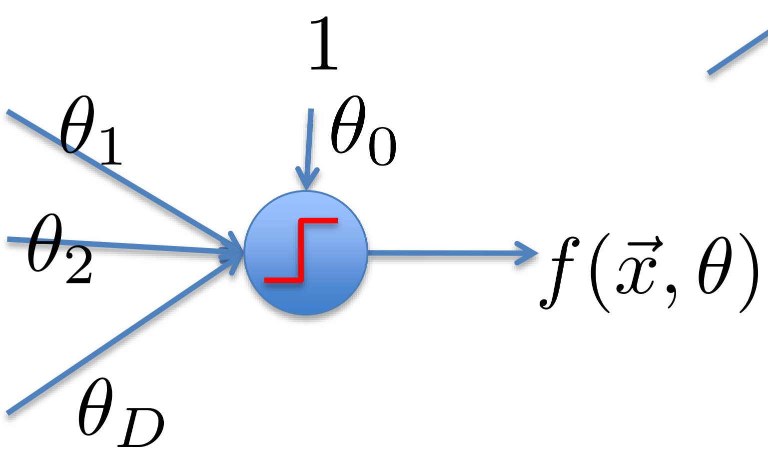
Types of Neurons



Linear Neuron



Logistic Neuron



Perceptron

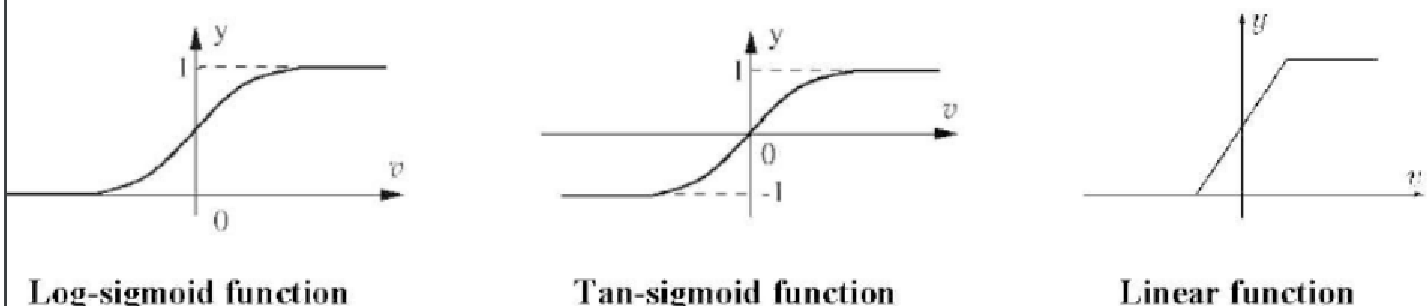
Potentially more. Require a convex loss function for gradient descent training.

Slide Credit: HKUST

Generalized linear models

- For classification problems, we want y to be a predictor of t. In other words, we wish to map the input vector into one of a number of discrete classes, or to posterior probabilities that lie between 0 and 1.
- For this purpose, it is useful to elaborate the linear model by introducing a nonlinear activation function f, which typically will constrain y to lie between -1 and 1 or between 0 and 1.

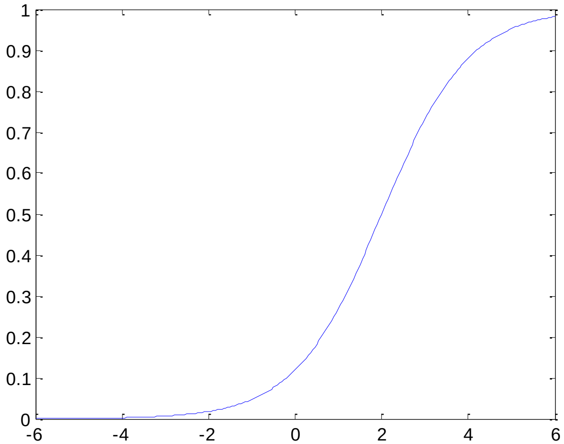
$$y(\mathbf{x}) = f\left(\mathbf{w}^t\mathbf{x} + w_0\right)$$



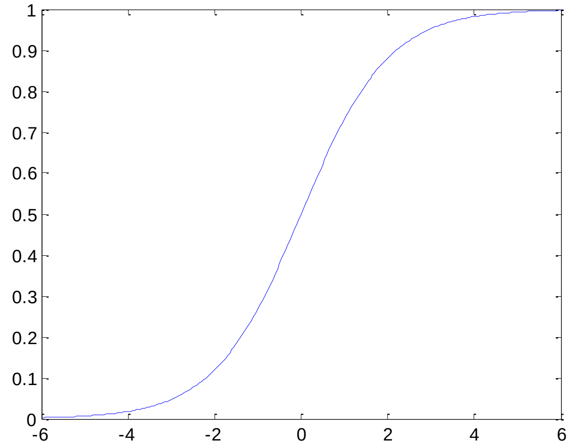
Sigmoid

$$g(w_0 + \sum_i w_i x_i) = \frac{1}{1 + e^{-(w_0 + \sum_i w_i x_i)}}$$

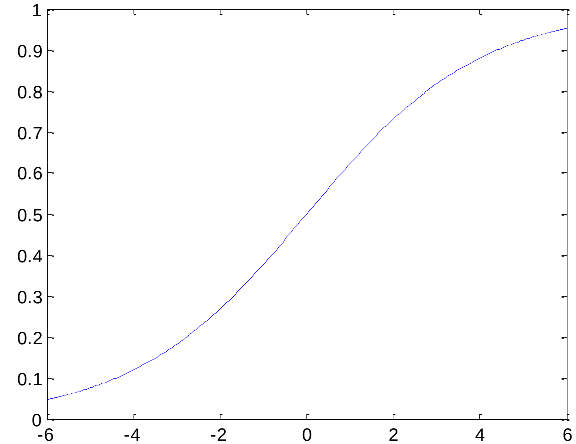
w_0=2, w_1=1



w_0=0, w_1=1

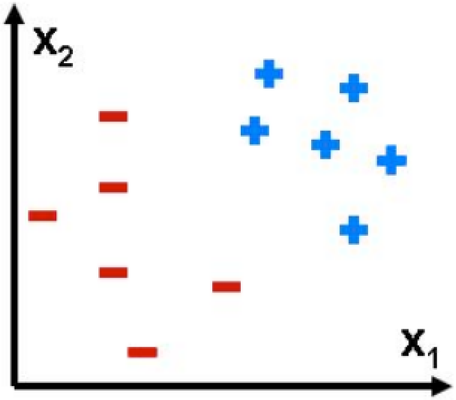


w_0=0, w_1=0.5



Case 1: Linearly separable inputs

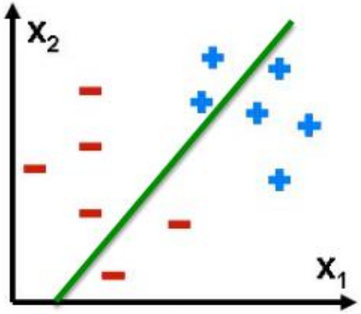
- For starters, let's assume that the training data is in fact perfectly linearly separable.
- In other words, there exists at least one hyperplane (one set of weights) that yields 0 classification error.
- We seek an algorithm that can automatically find such a hyperplane.



The Perceptron algorithm

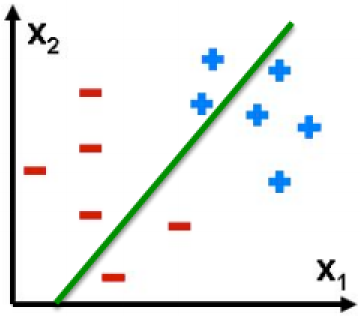
- The perceptron algorithm was invented by Frank Rosenblatt (1962).
- The algorithm is iterative.
- The strategy is to start with a random **guess** at the weights **w**, and to then iteratively change the weights to move the hyperplane in a direction that lowers the classification error.

Frank Rosenblatt (1928 – 1971)



The Perceptron algorithm

- Note that as we change the weights continuously, the classification error changes in discontinuous, piecewise constant fashion.
- Thus we cannot use the classification error per se as our objective function to minimize.
- What would be a better objective function?



The Perceptron criterion

- Note that we seek **w** such that

$$\mathbf{w}^t\mathbf{x} \geq 0 \text{ when } t = +1$$
$$\mathbf{w}^t\mathbf{x} < 0 \text{ when } t = -1$$

- In other words, we would like

$$\mathbf{w}^t\mathbf{x}_n t_n \geq 0 \;\forall n$$

- Thus we seek to minimize

$$E_P(\mathbf{w}) = -\sum_{n \in \mathcal{M}} \mathbf{w}^t\mathbf{x}_n t_n$$

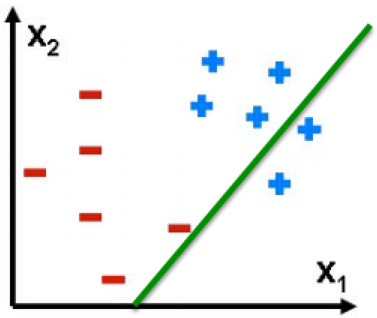
where $\mathcal{M}$ is the set of misclassified inputs.

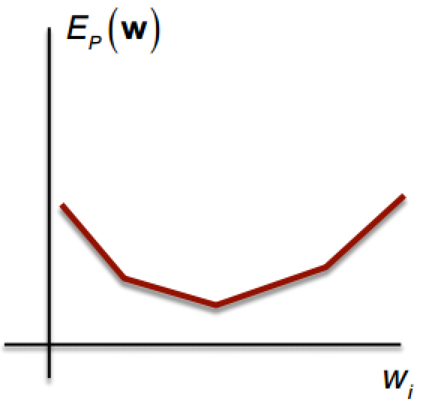
The Perceptron criterion

$$E_P(\mathbf{w}) = -\sum_{n \in \mathcal{M}} \mathbf{w}^t \mathbf{x}_n t_n$$

where $\mathcal{M}$ is the set of misclassified inputs.

- Observations:
  - $E_P(\mathbf{w})$ is always non-negative.
  - $E_P(\mathbf{w})$ is continuous and piecewise linear, and thus easier to minimize.





The Perceptron algorithm

$$E_P(\mathbf{w}) = -\sum_{n \in \mathcal{M}} \mathbf{w}^t \mathbf{x}_n t_n$$

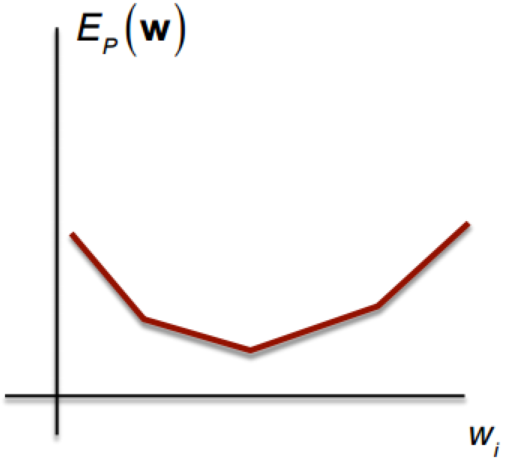
where $\mathcal{M}$ is the set of misclassified inputs.

$$\frac{dE_P(\mathbf{w})}{d\mathbf{w}} = -\sum_{n \in \mathcal{M}} \mathbf{x}_n t_n$$

where the derivative exists.

- Gradient descent:

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \eta \nabla E_P(\mathbf{w}) = \mathbf{w}^{\tau} + \eta \sum_{n \in \mathcal{M}} \mathbf{x}_n t_n$$



The Perceptron algorithm

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \eta \nabla E_P(\mathbf{w}) = \mathbf{w}^{t} + \eta \sum_{n \in \mathcal{M}} \mathbf{x}_n t_n$$

- Why does this make sense?
  - If an input from $C_1 (t = +1)$ is misclassified, we need to make its projection on **w** more positive.
  - If an input from C_2 $(t = -1)$ is misclassified, we need to make its projection on **w** more negative.

The Perceptron algorithm

- The algorithm can be implemented sequentially:
  - Repeat until convergence:
    - For each input ($\mathbf{x}_n$, t_n):
      - If it is correctly classified, do nothing
      - If it is misclassified, update the weight vector to be
        $\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} + \eta \mathbf{x}_n t_n$
      - Note that this will lower the contribution of input n to the objective function:
        $-\left(\mathbf{w}^{(\tau)}\right)^t \mathbf{x}_n t_n \to -\left(\mathbf{w}^{(\tau+1)}\right)^t \mathbf{x}_n t_n = -\left(\mathbf{w}^{(\tau)}\right)^t \mathbf{x}_n t_n - \eta \left(\mathbf{x}_n t_n\right)^t \mathbf{x}_n t_n < -\left(\mathbf{w}^{(\tau)}\right)^t \mathbf{x}_n t_n.$

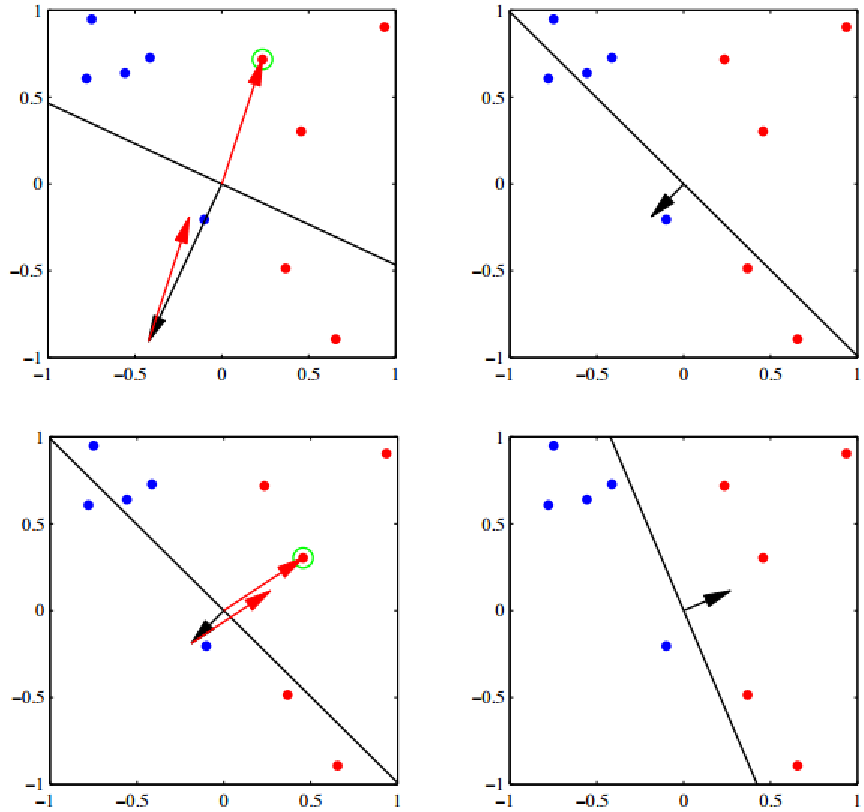
Not monotonic

- While updating with respect to a misclassified input n will lower the error for that input, the error for other misclassified inputs may increase.
- Also, new inputs that had been classified correctly may now be misclassified.
- The result is that the perceptron algorithm is not guaranteed to reduce the total error monotonically at each stage.

The perceptron convergence theorem

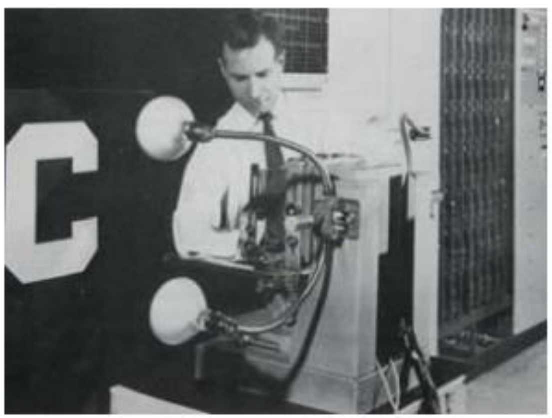
- Despite this non-monotonicity, **if in fact the data are linearly separable, then the algorithm is guaranteed to find an exact solution in a finite number of steps** (Rosenblatt, 1962).

Example



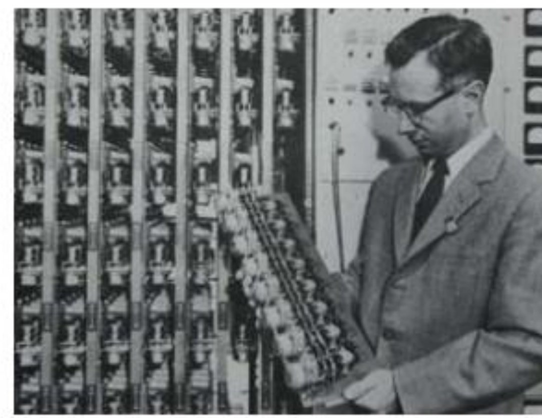
The first learning machine

- Mark 1 Perceptron Hardware (c. 1960)



Visual Inputs

Patch board allowing configuration of inputs ϕ



Rack of adaptive weights **w** (motor-driven potentiometers)

Practical limitations

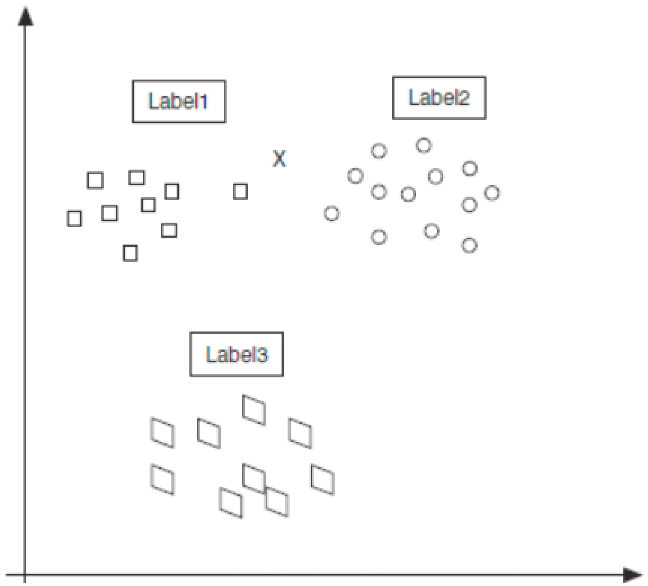
- The Perceptron Convergence Theorem is an important result. However, there are practical limitations:
  - Convergence may be slow
  - If the data are not separable, the algorithm will not converge.
  - We will only know that the data are separable once the algorithm converges.
  - The solution is in general not unique, and will depend upon initialization, scheduling of input vectors, and the learning rate η.

Generalization to not linearly separable inputs

- The single-layer perceptron can be generalized to yield good linear solutions to problems that are not linearly separable.
- Example: The Pocket Algorithm (Gal 1990)
  - Idea:
    - Run the perceptron algorithm
    - Keep track of the weight vector **w*** that has produced the best classification error achieved so far.
    - It can be shown that **w*** will converge to an optimal solution with probability 1.

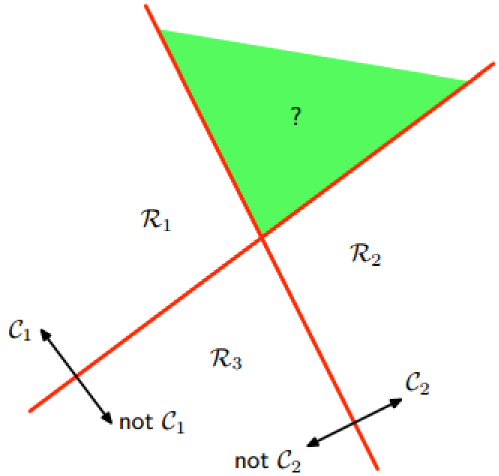
Generalization to multiclass problems

- How can we use perceptrons, or linear classifiers in general, to classify inputs when there are $K > 2$ classes?



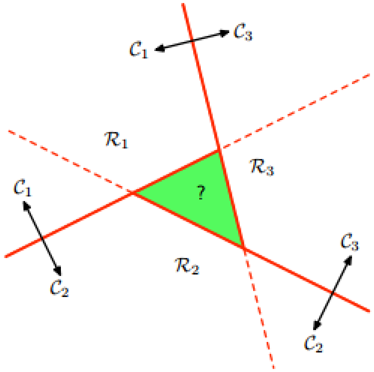
K>2 classes

- Idea #1: Just use *K-1* discriminant functions, each of which separates one class C_k from the rest. (One-versus-the-rest classifier.)
- Problem: Ambiguous regions



K>2 classes

- Idea #2: Use $K(K-1)/2$ discriminant functions, each of which separates two classes C_j, C_k from each other. (One-versus-one classifier.)
- Each point classified by majority vote.
- Problem: Ambiguous regions



K>2 classes

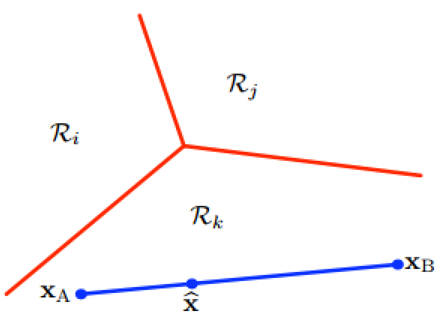
- Idea #3: Use K discriminant functions $y_k(x)$
- Use the **magnitude** of $y_k(x)$, not just the sign.

$$y_k(\mathbf{x}) = \mathbf{w}_k^t \mathbf{x}$$

$\mathbf{x}$ assigned to C_k if $y_k(\mathbf{x}) > y_j(\mathbf{x}) \forall j \neq k$

Decision boundary $y_k(\mathbf{x}) = y_j(\mathbf{x}) \rightarrow \left(w_k - w_j\right)^t x + \left(w_{k0} - w_{j0}\right) = 0$

Results in decision regions that are simply-connected and convex.



1-of-K coding scheme

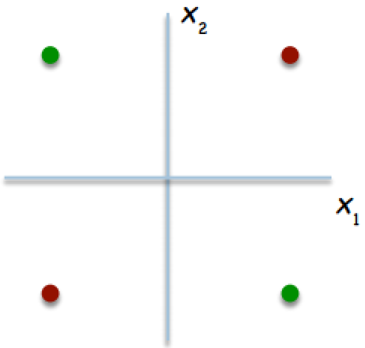
- When there are K>2 classes, target variables can be coded using the 1-of-K coding scheme:

$$\text{Input from Class } C_i \Leftrightarrow t = [0\ 0\ \ldots 1 \ldots 0\ 0]^t$$

Element i

Computational limitations of perceptrons

- Initially, the perceptron was thought to be a potentially powerful learning machine that could model human neural processing.
- However, Minsky & Papert (1969) showed that the single-layer perceptron could not learn a simple XOR function.
- This is just one example of a non-linearly separable pattern that cannot be learned by a single-layer perceptron.



Marvin Minsky (1927 -)

Limitation

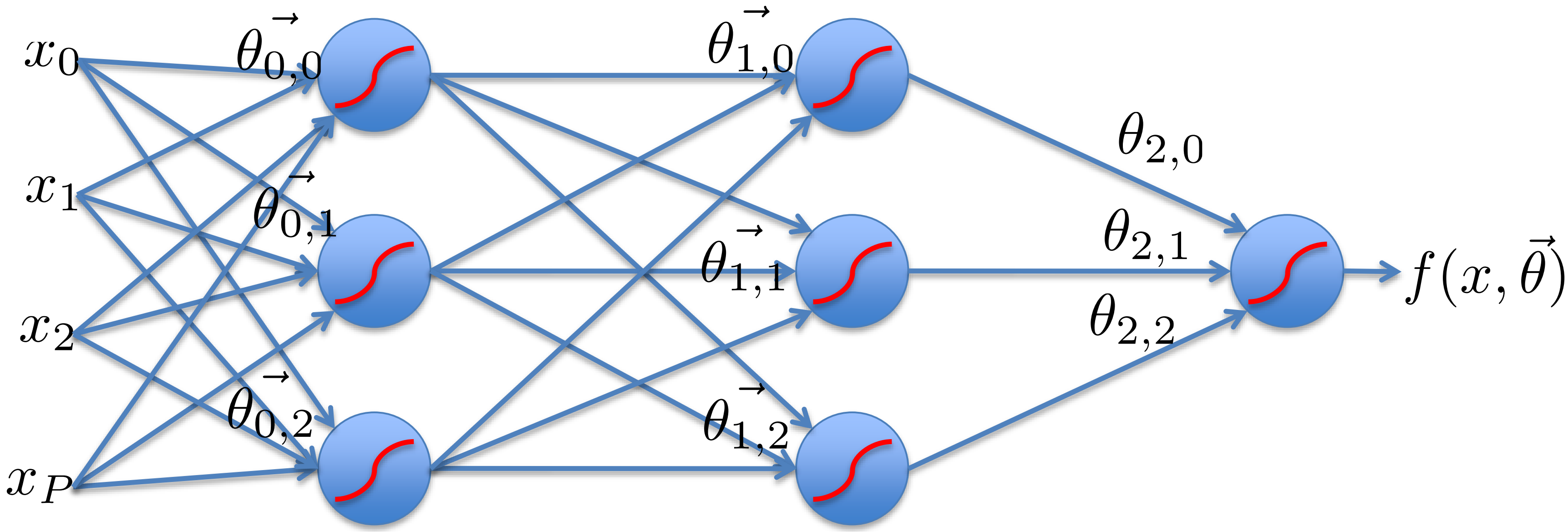
- A single “neuron” is still a linear decision boundary

- What to do?

- Idea: Stack a bunch of them together!

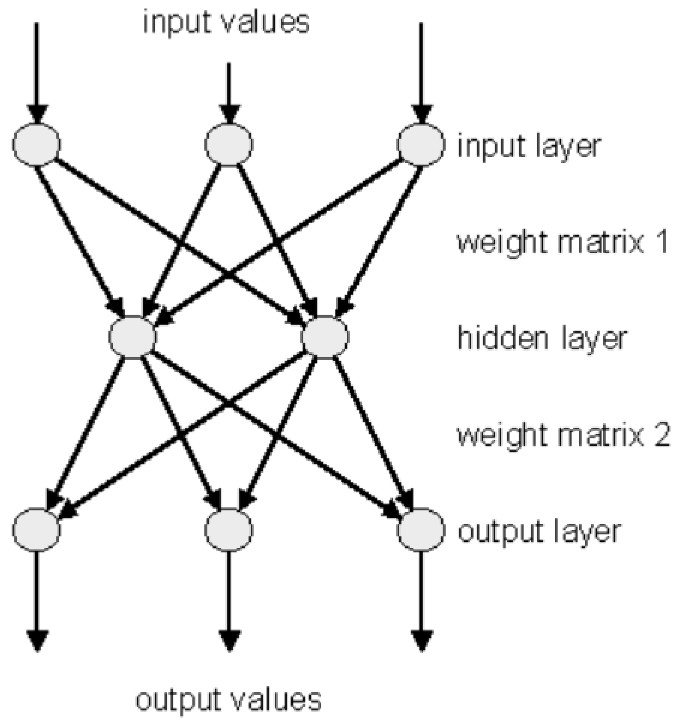
Multilayer Networks

- Cascade Neurons together
- The output from one layer is the input to the next
- Each Layer has its own sets of weights



Multi-layer perceptrons

- Minsky & Papert's book was widely misinterpreted as showing that artificial neural networks were inherently limited.
- This contributed to a decline in the reputation of neural network research through the 70s and 80s.
- However, their findings apply only to single-layer perceptrons. Multi-layer perceptrons are capable of learning highly nonlinear functions, and are used in many practical applications.

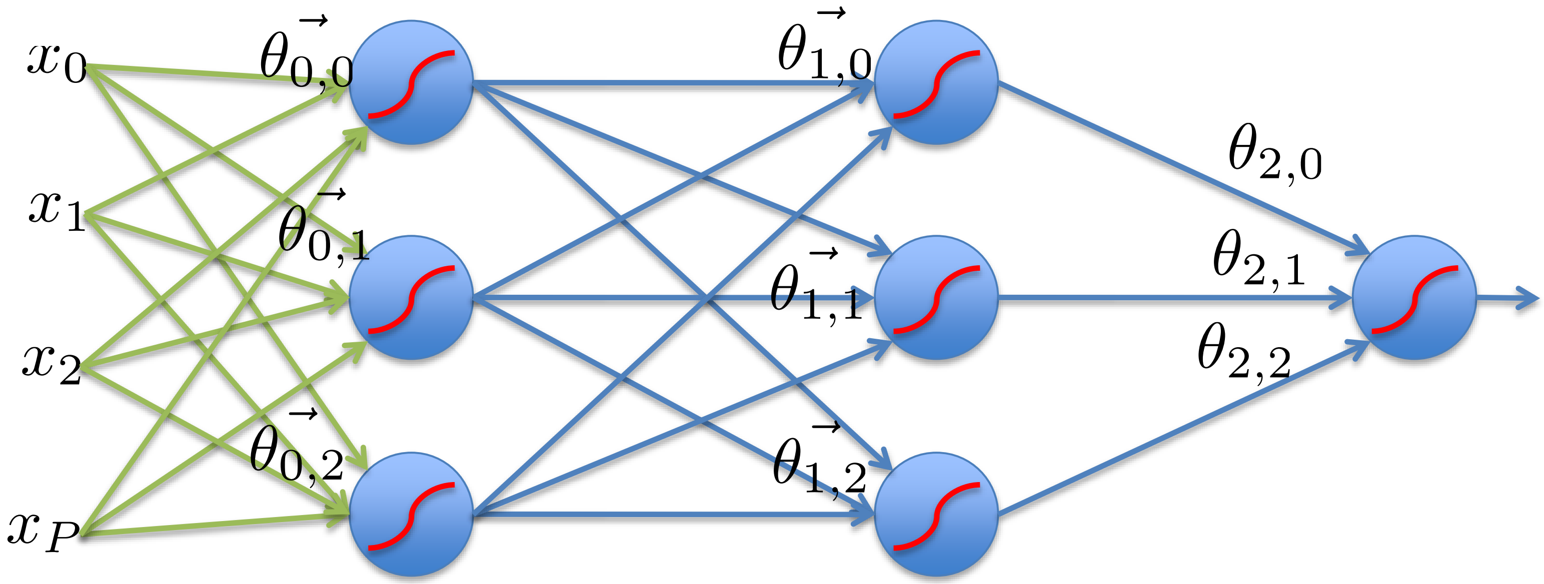


Universal Function Approximators

- Theorem
  - 3-layer network with linear outputs can uniformly approximate any continuous function to arbitrary accuracy, given enough hidden units [Funahashi '89]

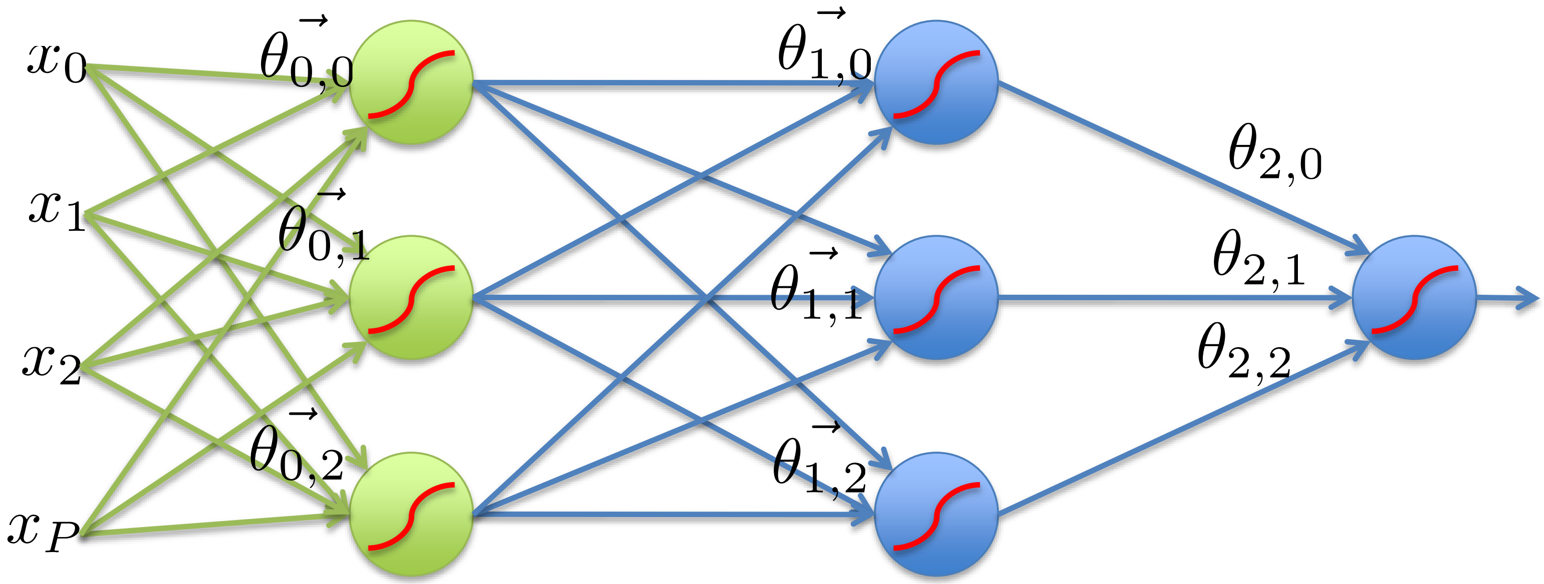
Feed-Forward Networks

- Predictions are fed forward through the network to classify



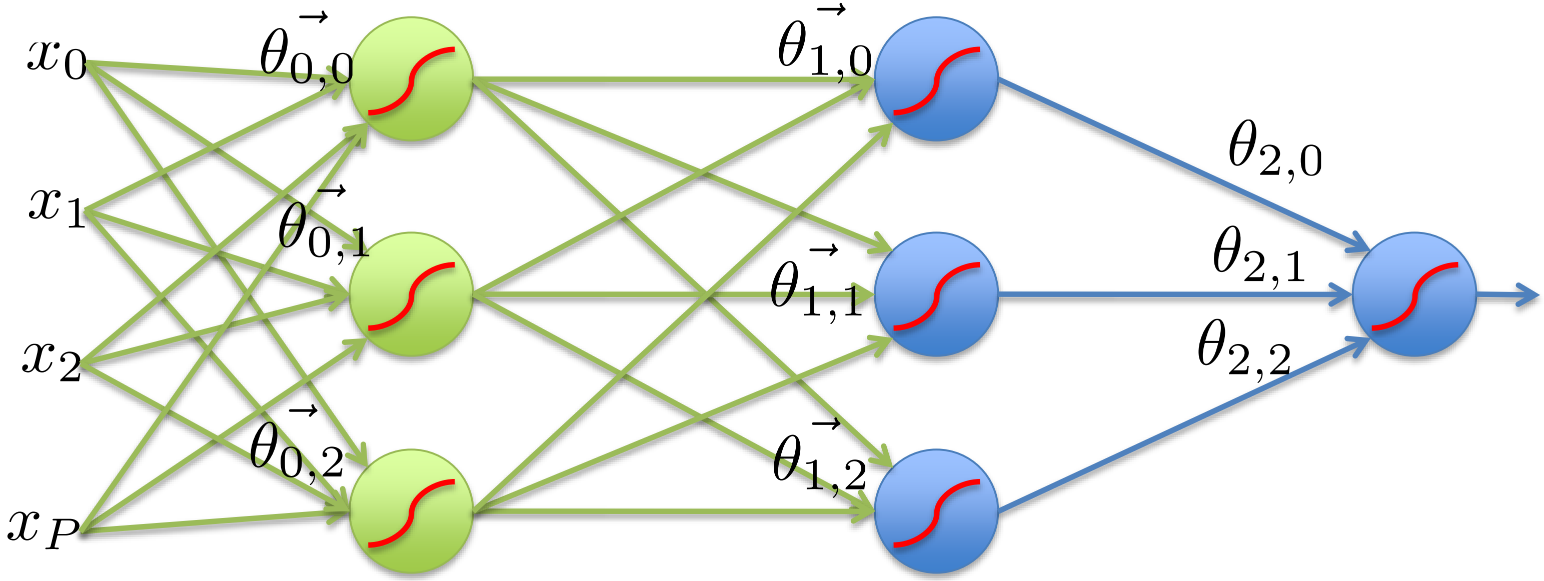
Feed-Forward Networks

- Predictions are fed forward through the network to classify



Feed-Forward Networks

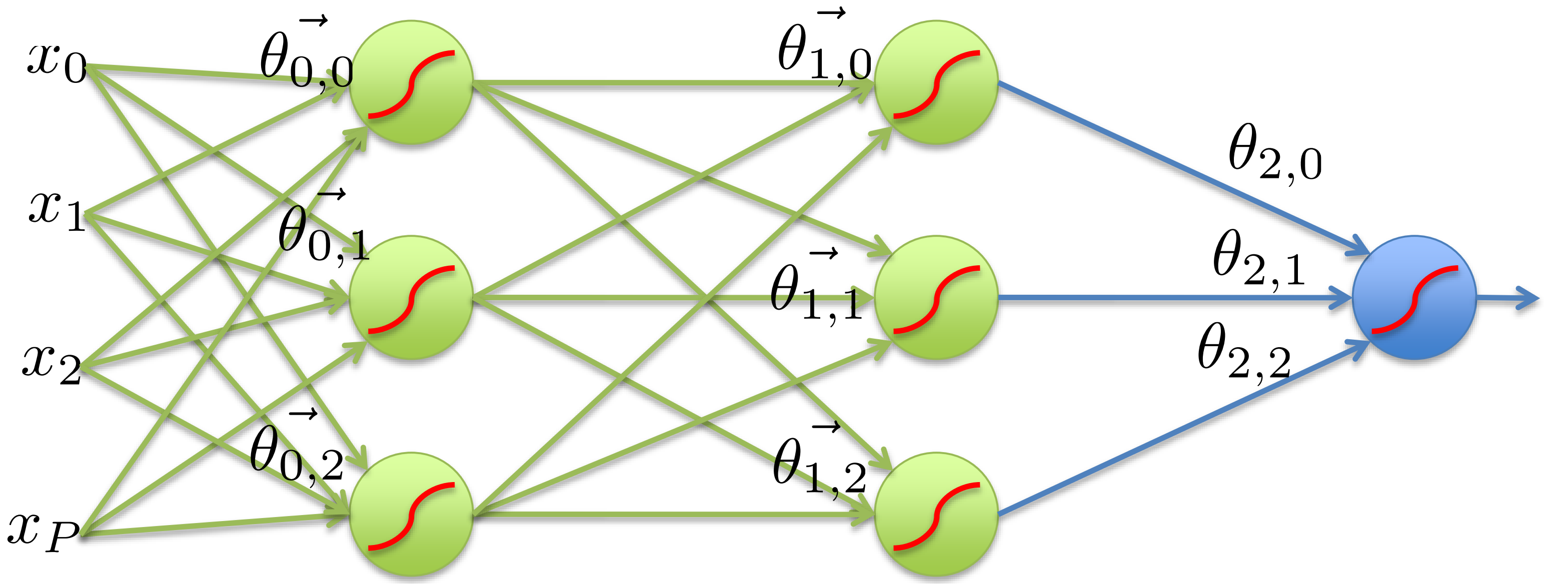
- Predictions are fed forward through the network to classify



Slide Credit: HKUST

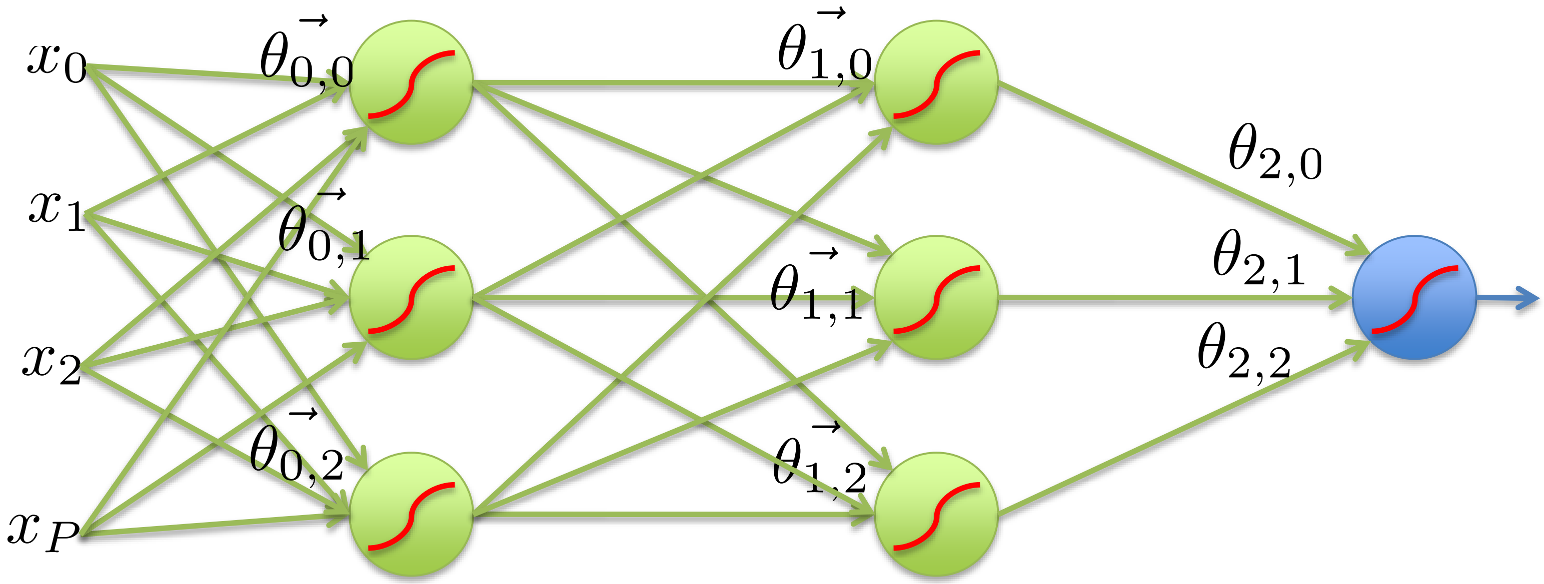
Feed-Forward Networks

- Predictions are fed forward through the network to classify



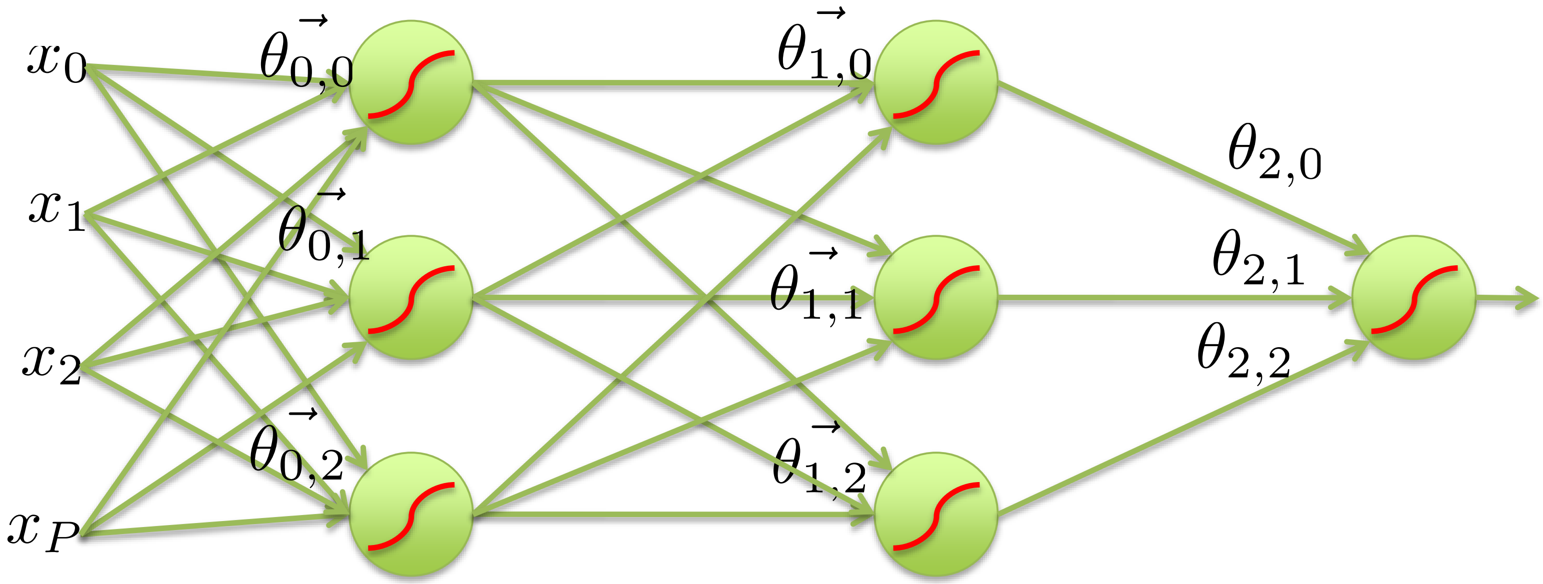
Feed-Forward Networks

- Predictions are fed forward through the network to classify



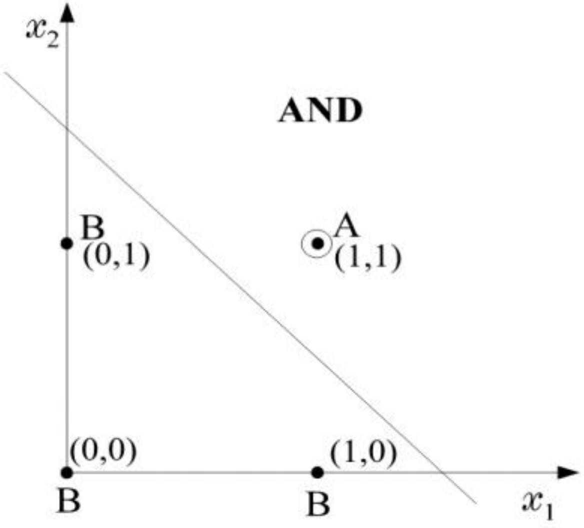
Feed-Forward Networks

- Predictions are fed forward through the network to classify



Implementing logical relations

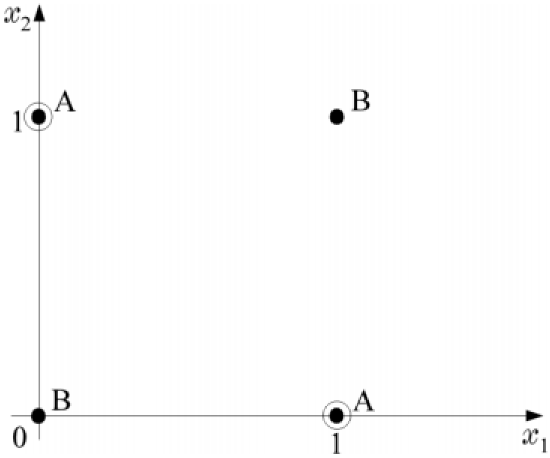
- AND and OR operations are linearly separable problems



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



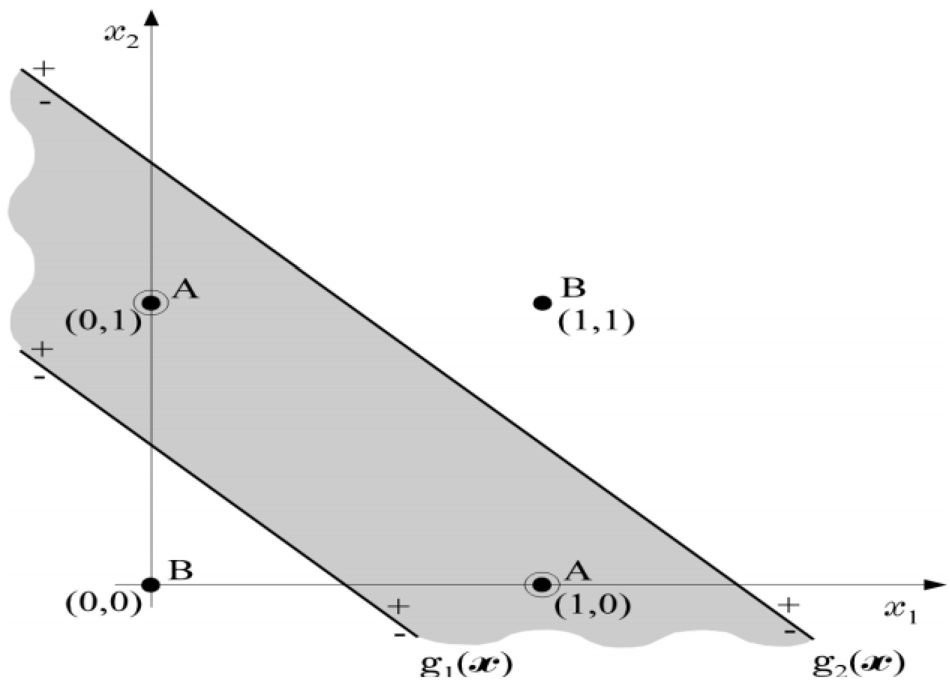
- How can we use linear classifiers to solve this problem?

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:

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

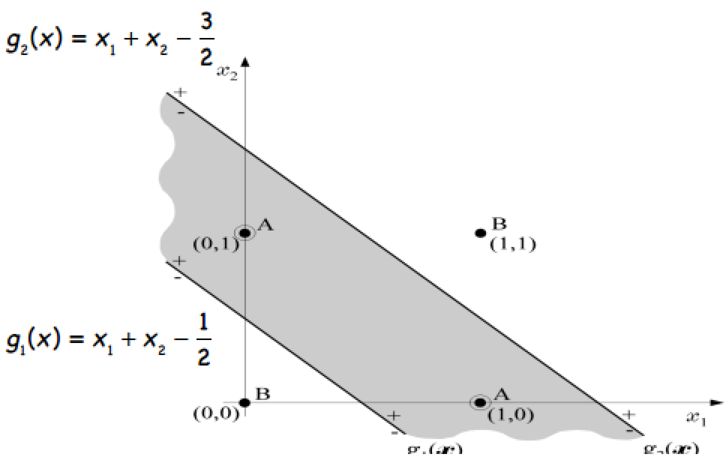
$f(x) = 1, \ \ x \geq 0$

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 $y_2 = f\left(g_2(x)\right)$



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.

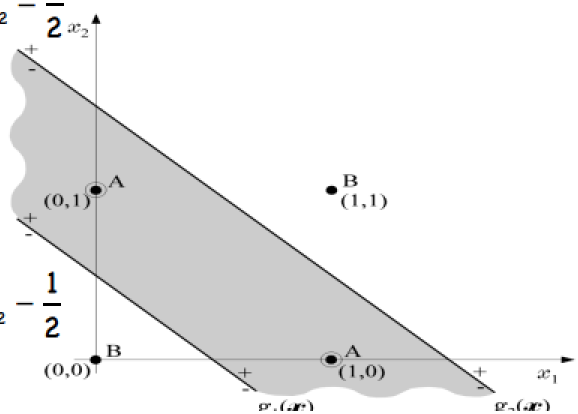
$$f\left(y_1 - y_2 - \frac{1}{2}\right)$$

where

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

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

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



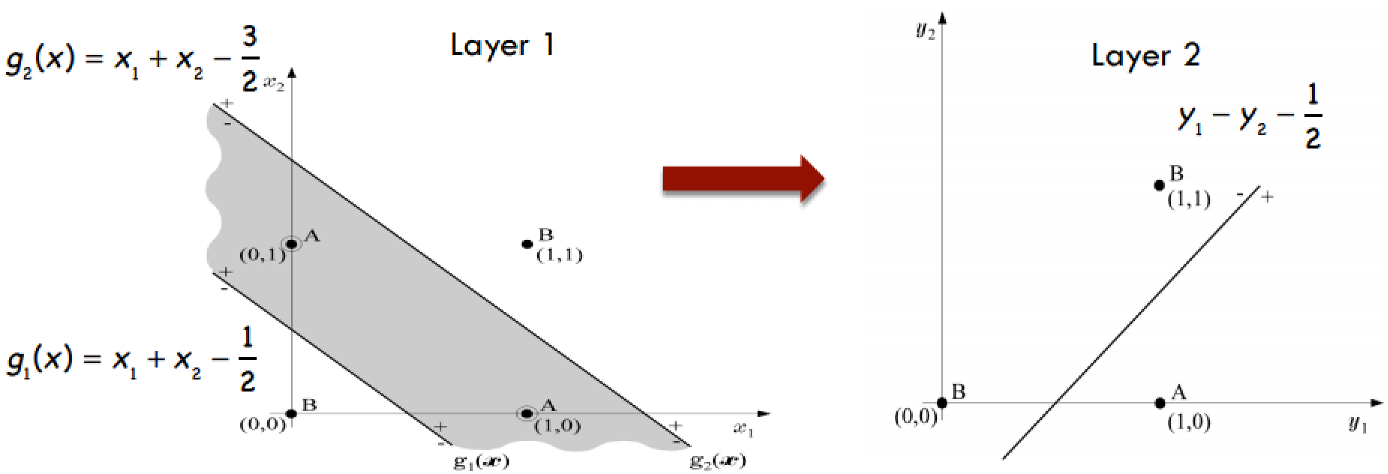
Two-layer perceptron

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

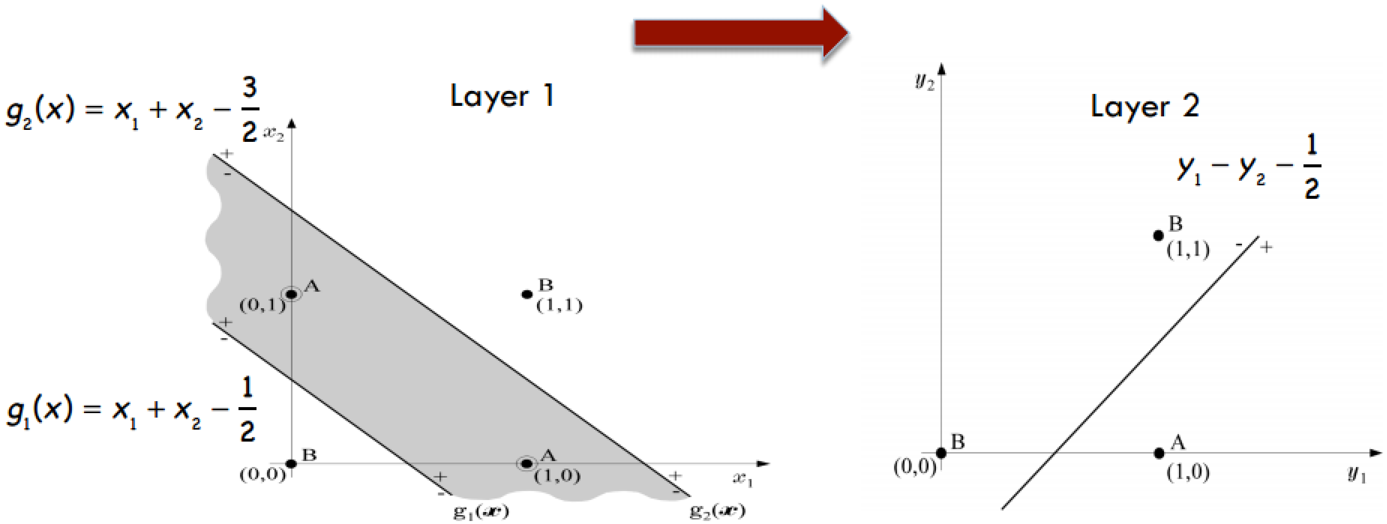
$$y_1 = f\left(g_1(x)\right) \text{ and } y_2 = f\left(g_2(x)\right)$$



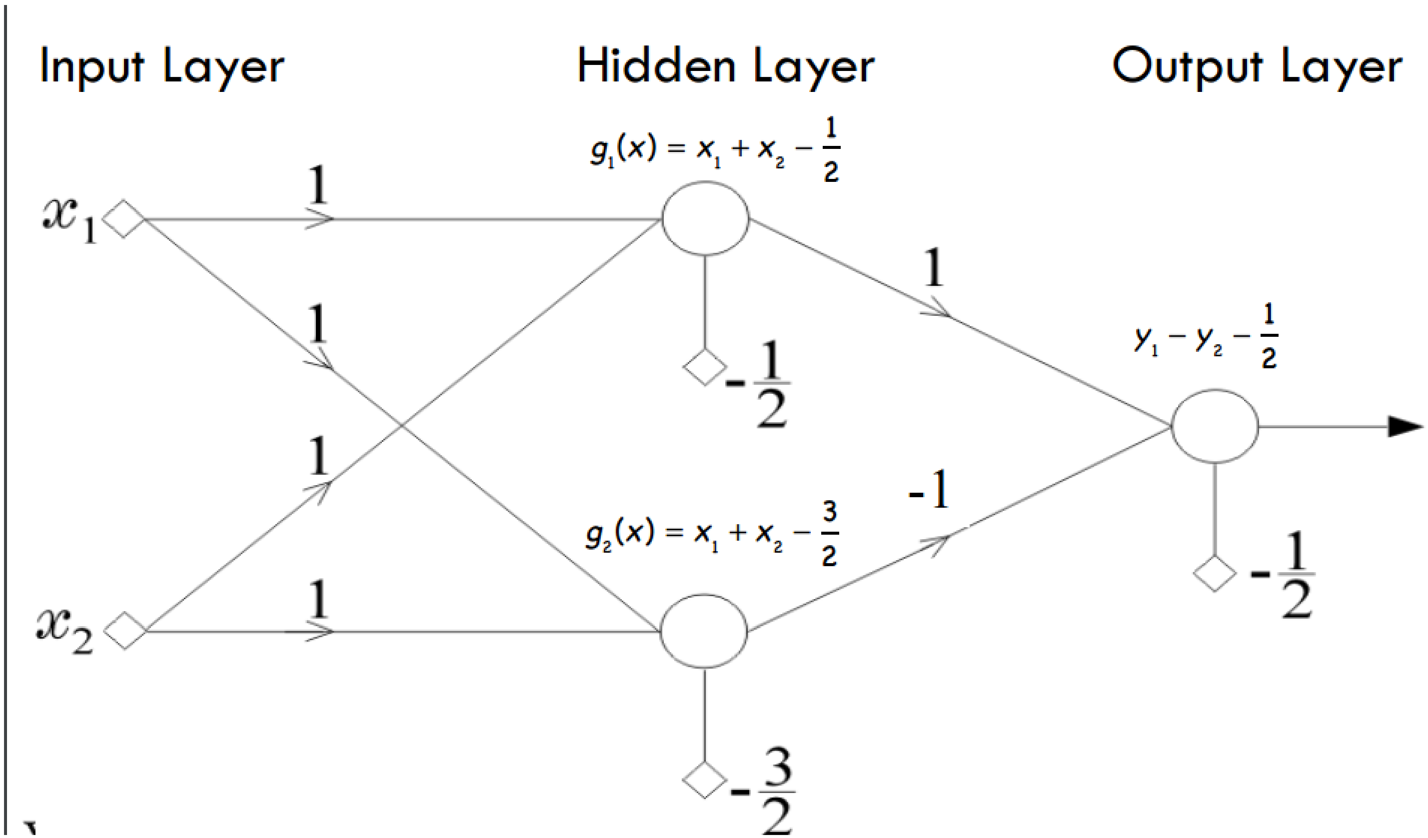
Two-layer perceptron

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

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



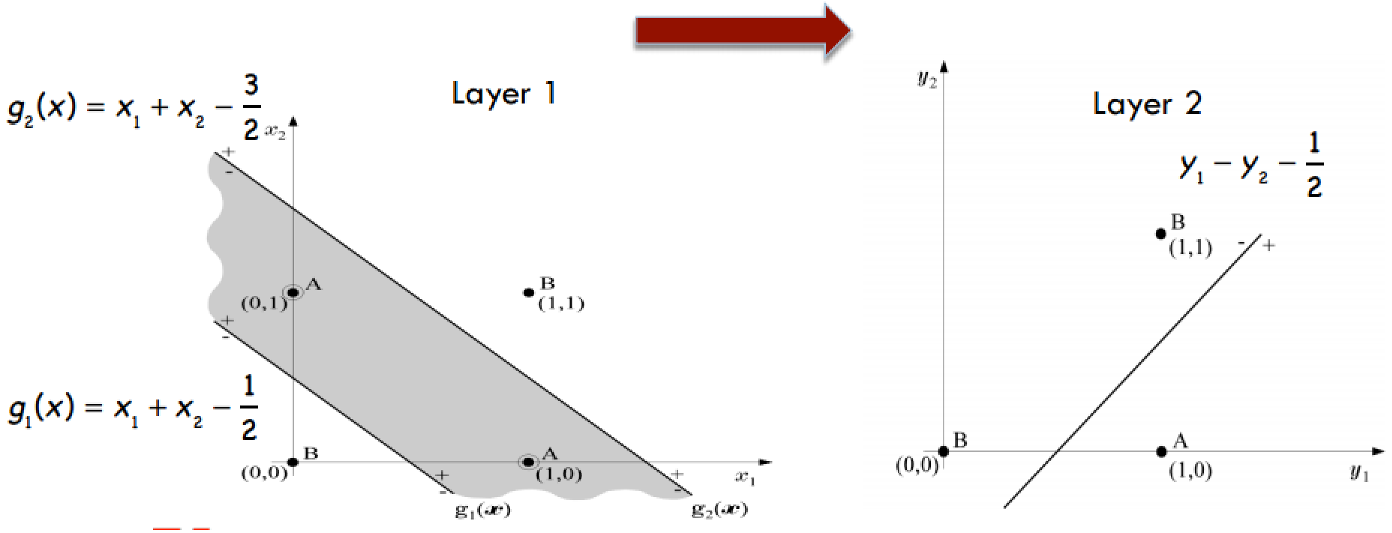
Two-layer perceptron



Two-layer perceptron

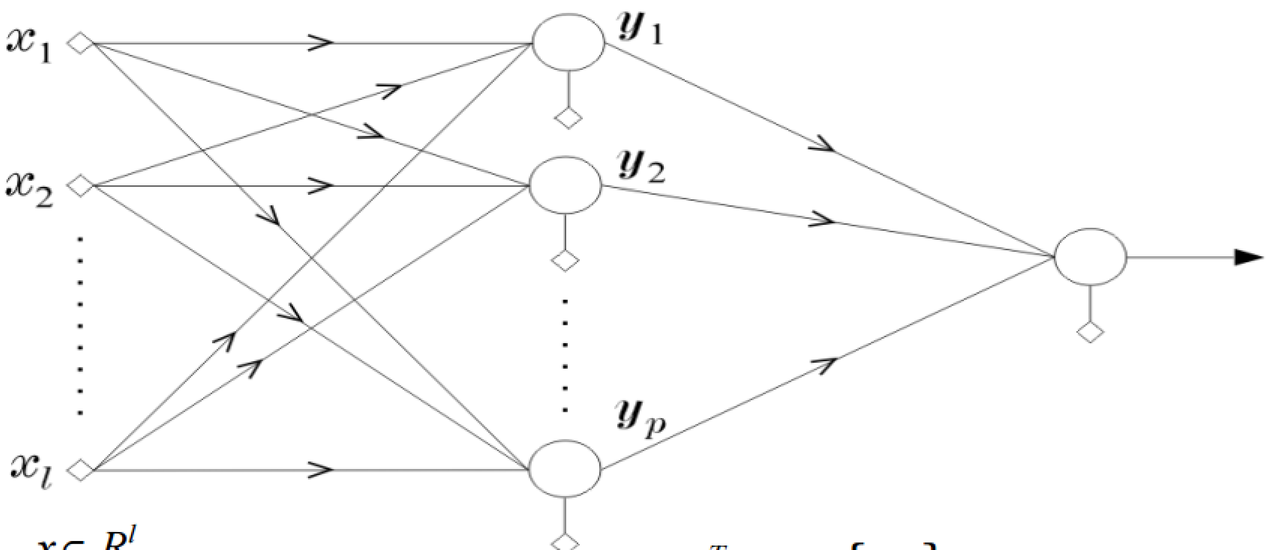
- Note that the hidden layer maps the plane onto the vertices of a unit square.

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



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.

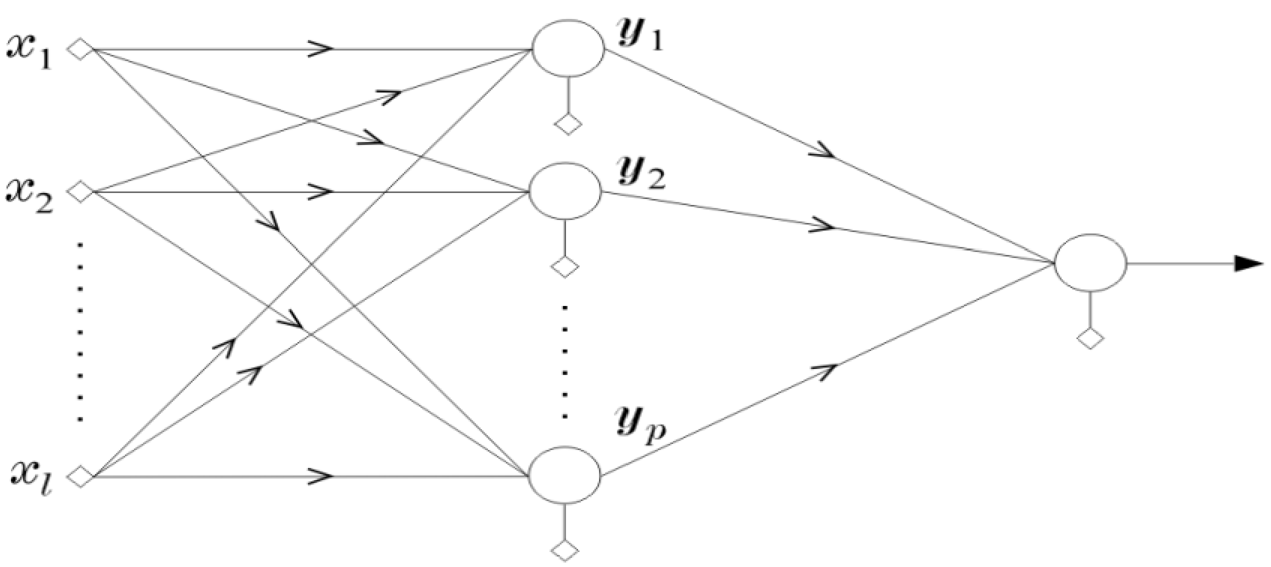


$\underline{x} \in R^l$

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

Higher dimensions

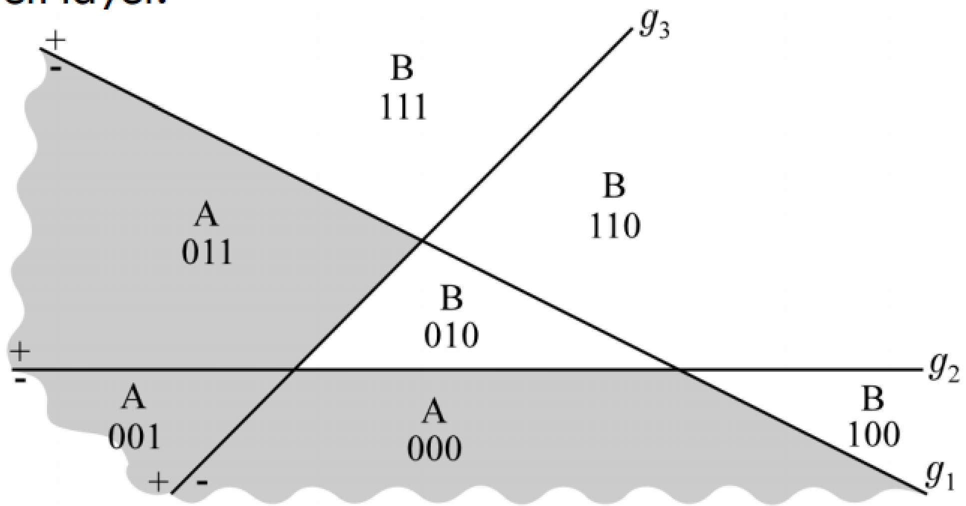
- Together, the hidden units map the input onto the vertices of a p-dimensional unit hypercube.



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

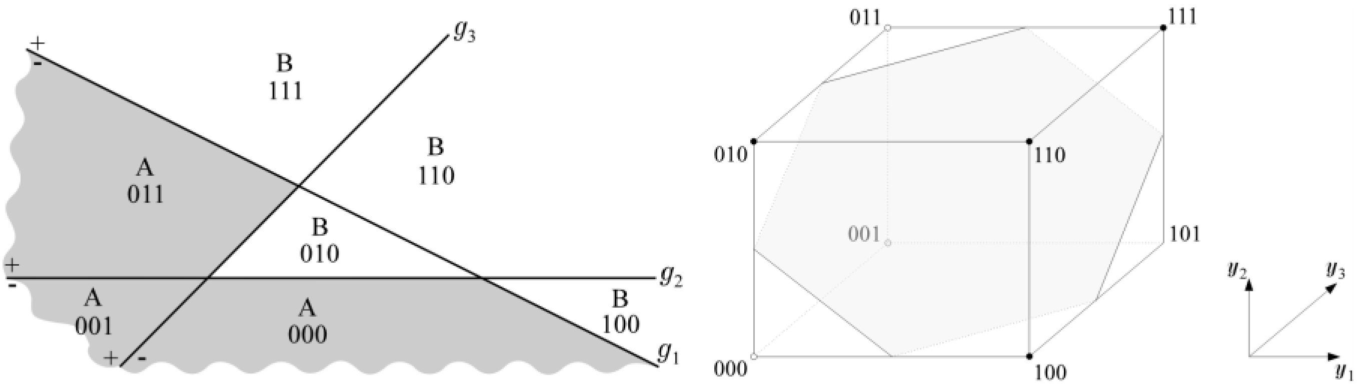
Two-layer perceptron

- 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:
  - $g_1(x) < 0$
  - $g_2(x) < 0$
  - $g_3(x) > 0$

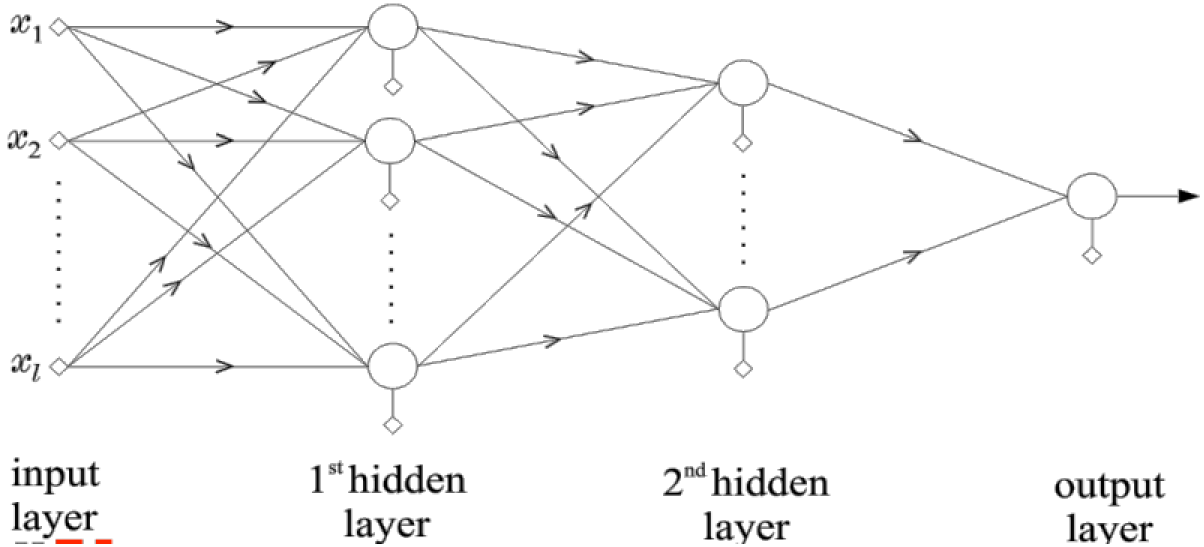


Limitations of a two-layer perceptron

- 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?

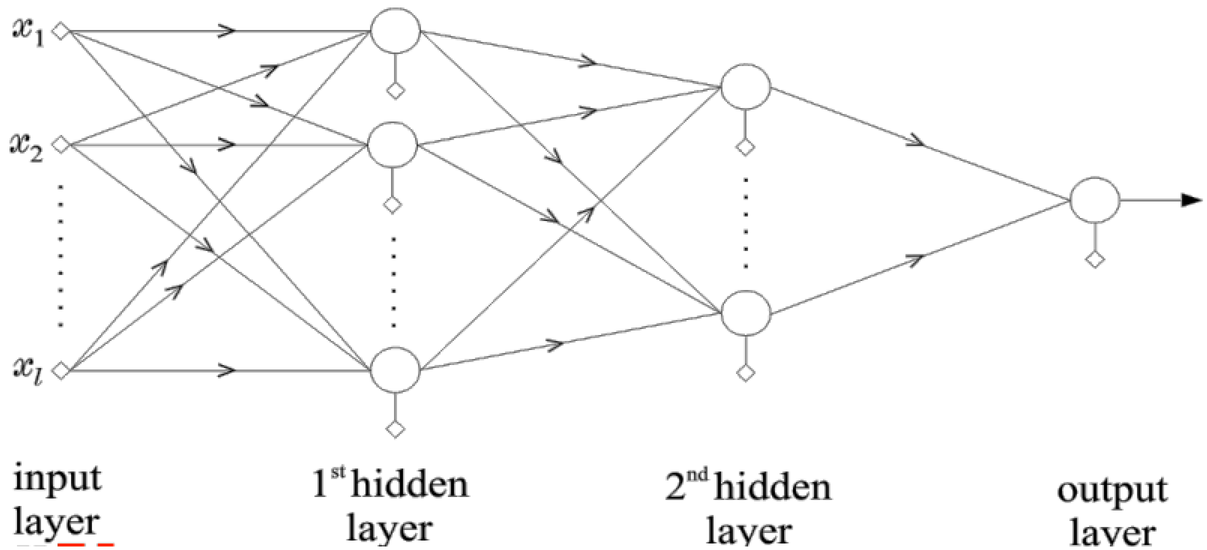
Three layer perceptron

- 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.



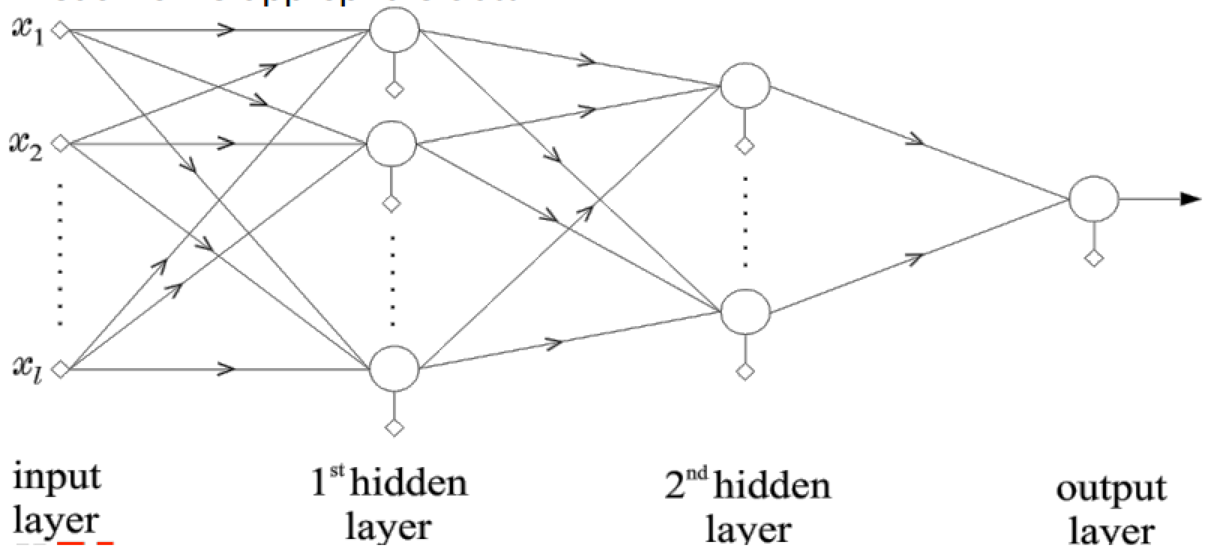
Three layer perceptron

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



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.



Learning parameters – Training data

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

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

where

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

and

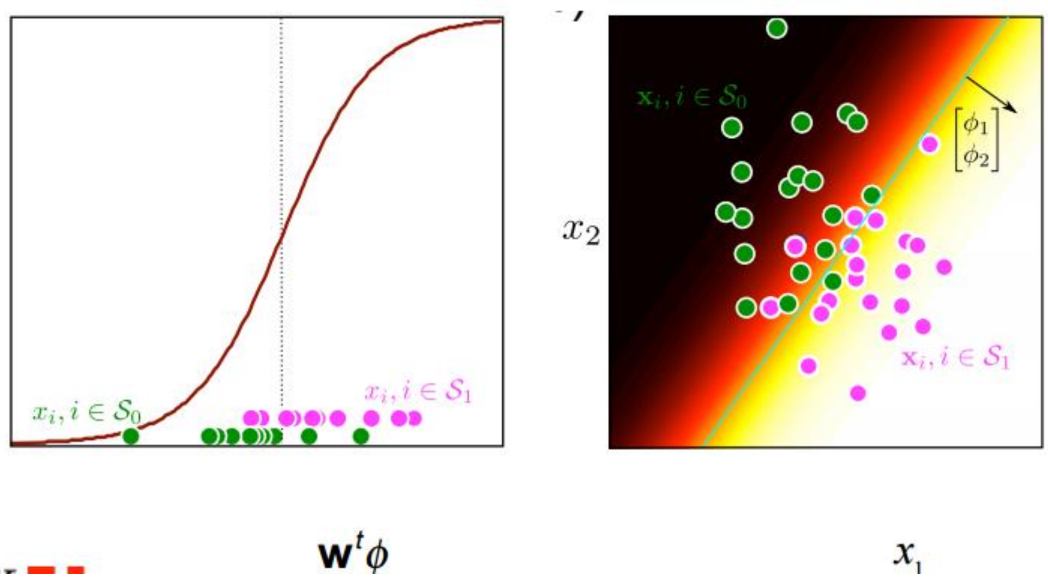
$$x(i) = \left[x_1(i), \ldots, x_{k_0}(i)\right]^t$$

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.

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.

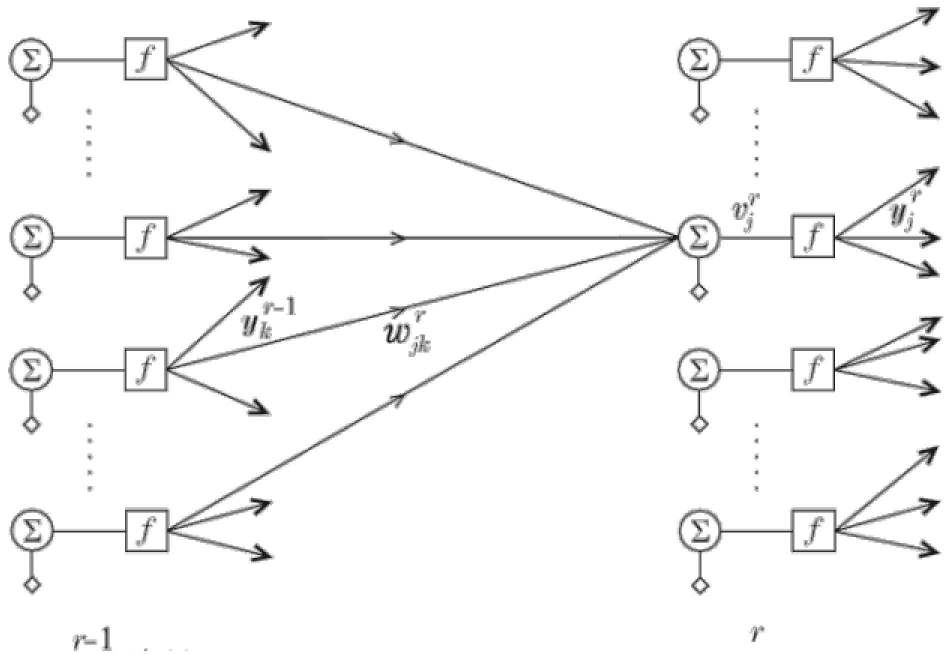
Werbos

Rumelhart

Hinton

Notation

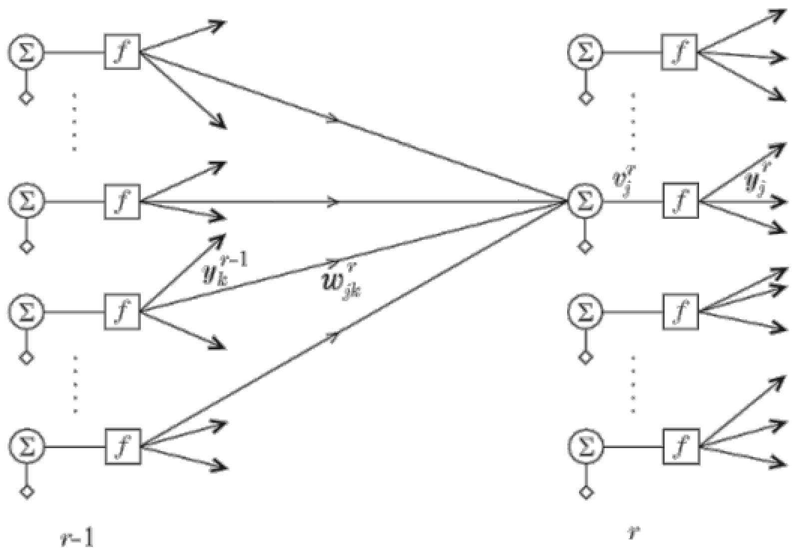
- Assume a network with L layers
  - k_0 nodes in the input layer.
  - k_r nodes in the r'th layer.



Notation

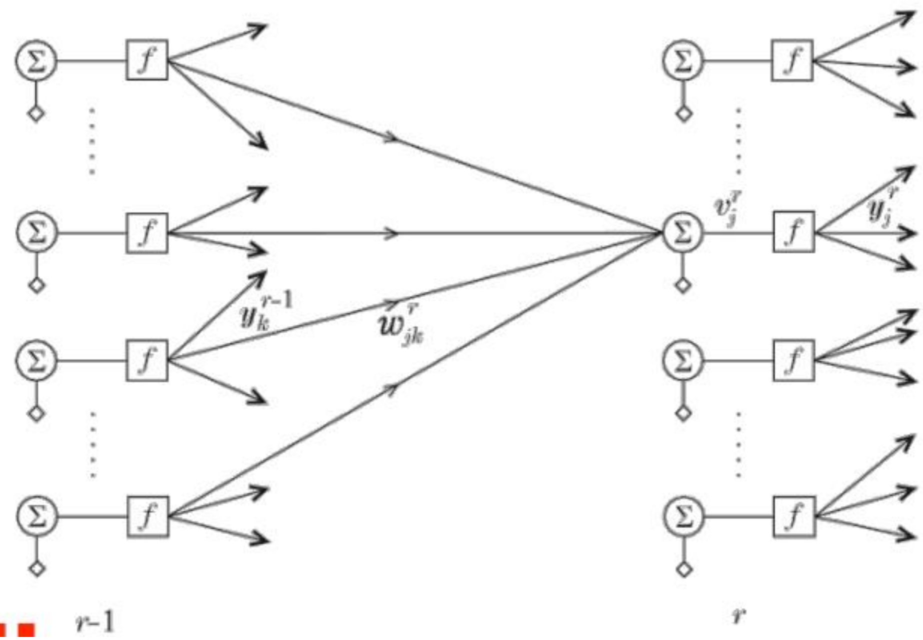
Let y_k^{r-1} be the output of the kth neuron of layer $r-1$.

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

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



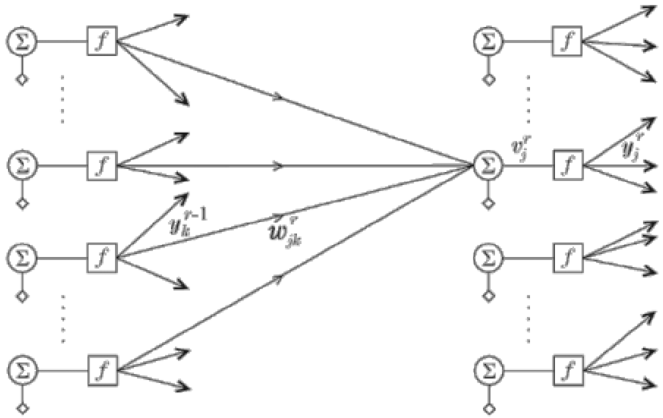
Notation

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

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

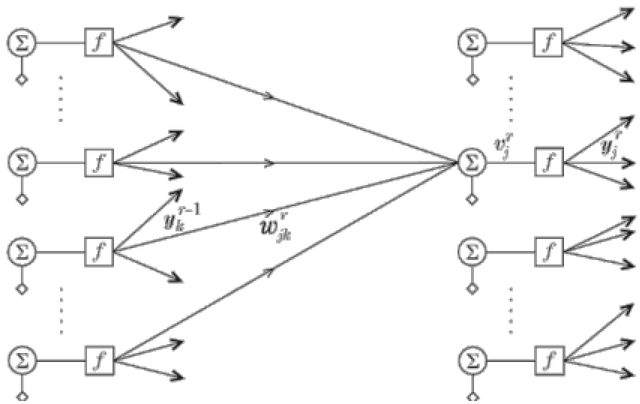
where $\hat{y}_m(i) = y_k^r(i)$ is the output of the output layer

and each layer is related to the previous layer through

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

and

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

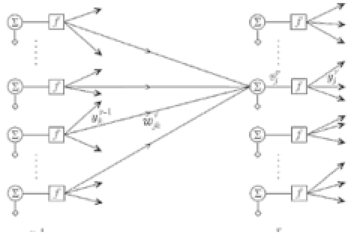


Gradient descent

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

- 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 $\hat{\mathbf{y}}(i)$ for each input vector $\mathbf{x}(i)$ 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 $v_j^r(i) = \left(\mathbf{w}_j^r\right)^t \mathbf{y}^{r-1}(i)$,

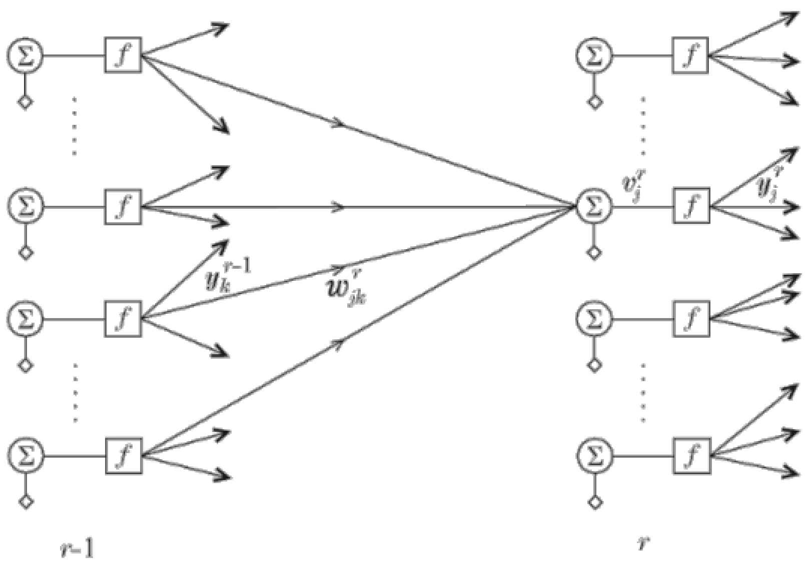
  the influence of the jth weight of the rth layer on the error can be expressed as:

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

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

where

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

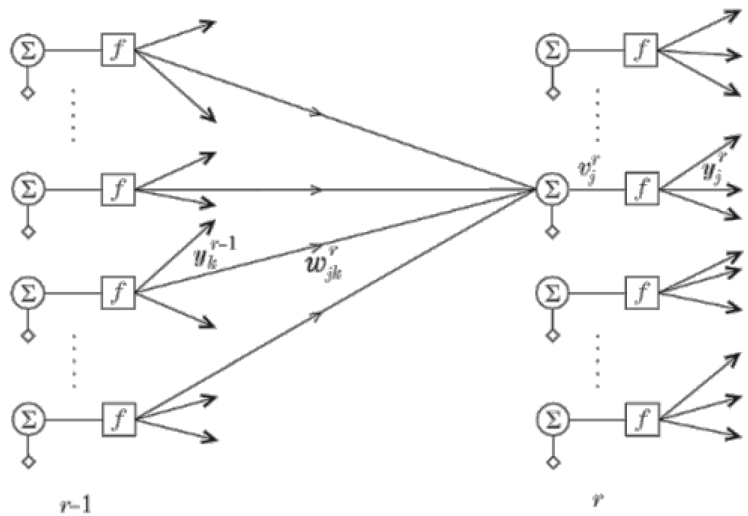


Gradient descent

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

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

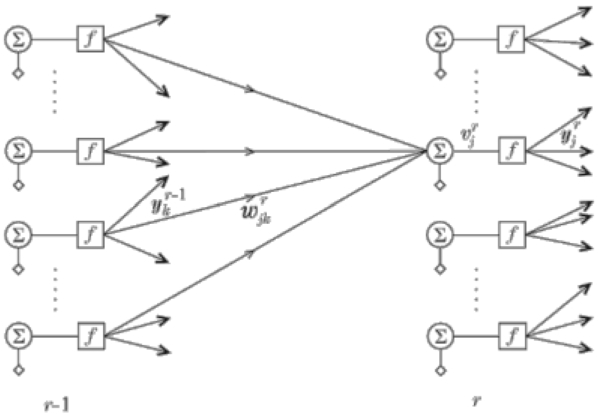
Recall that $\hat{y}_m(i) = y_j^L(i) = f\left(v_j^L(i)\right)$

Thus at the output layer we have

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

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

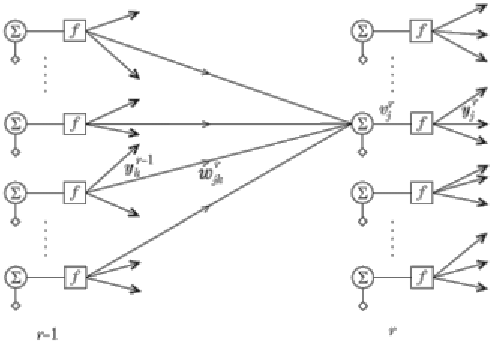
$$\boxed{\delta_j^L(i) = e_j^L(i) f\left(v_j^L(i)\right)\left(1 - f\left(v_j^L(i)\right)\right)}$$



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 the 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:

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

     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 versus 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 versus 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.

Neural Nets

- Best performers on OCR
  - http://yann.lecun.com/exdb/lenet/index.html
- NetTalk
  - Text to Speech system from 1987
  - http://youtu.be/tXMaFhO6dIY?t=45m15s
- Rick Rashid speaks Mandarin
  - http://youtu.be/Nu-nlQqFCKg?t=7m30s

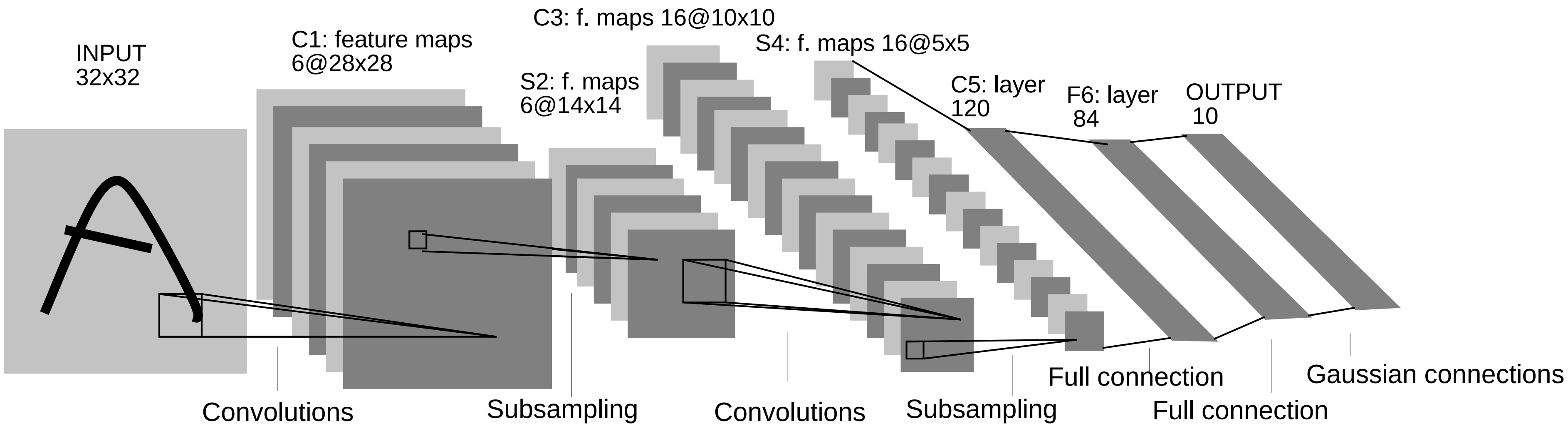
Convergence of backprop

- Perceptron leads to convex optimization
  - Gradient descent reaches **global minima**

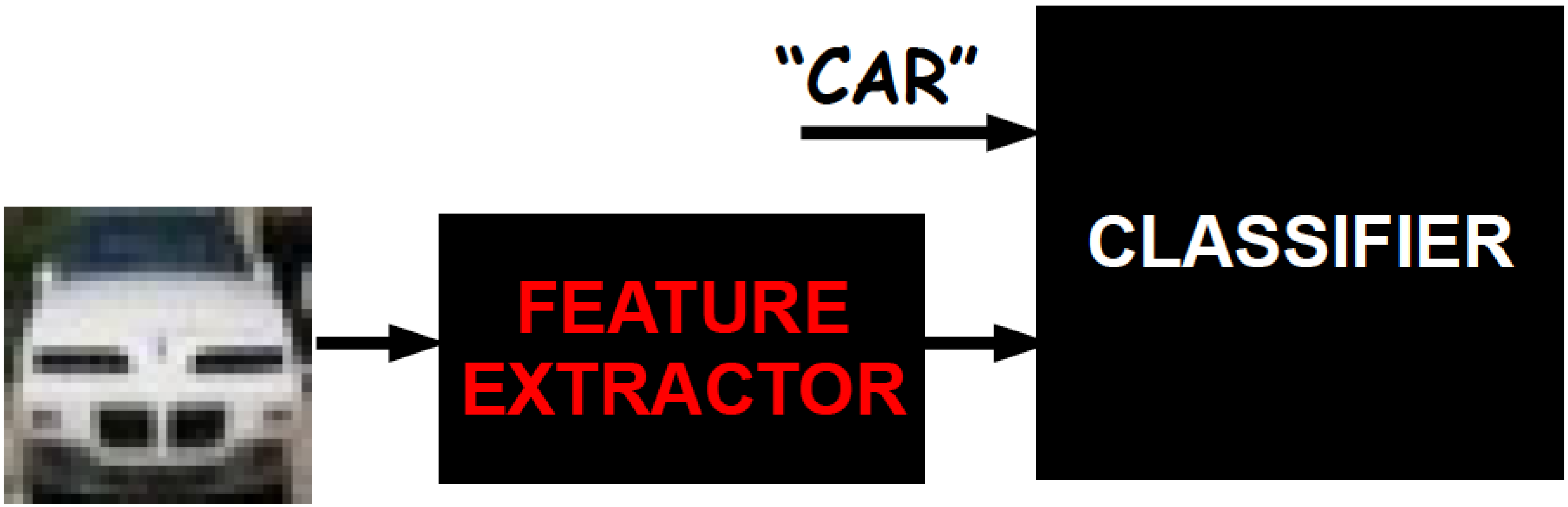
- Multilayer neural nets **not convex**
  - Gradient descent gets stuck in local minima
  - Hard to set learning rate
  - Selecting number of hidden units and layers = fuzzy process
  - NNs had fallen out of fashion in 90s, early 2000s
  - Back with a new name and significantly improved performance!!!!
    - Deep networks
      - Dropout and trained on much larger corpus

Convolutional Nets

- Example:
  - http://yann.lecun.com/exdb/lenet/index.html



Building an Object Recognition System

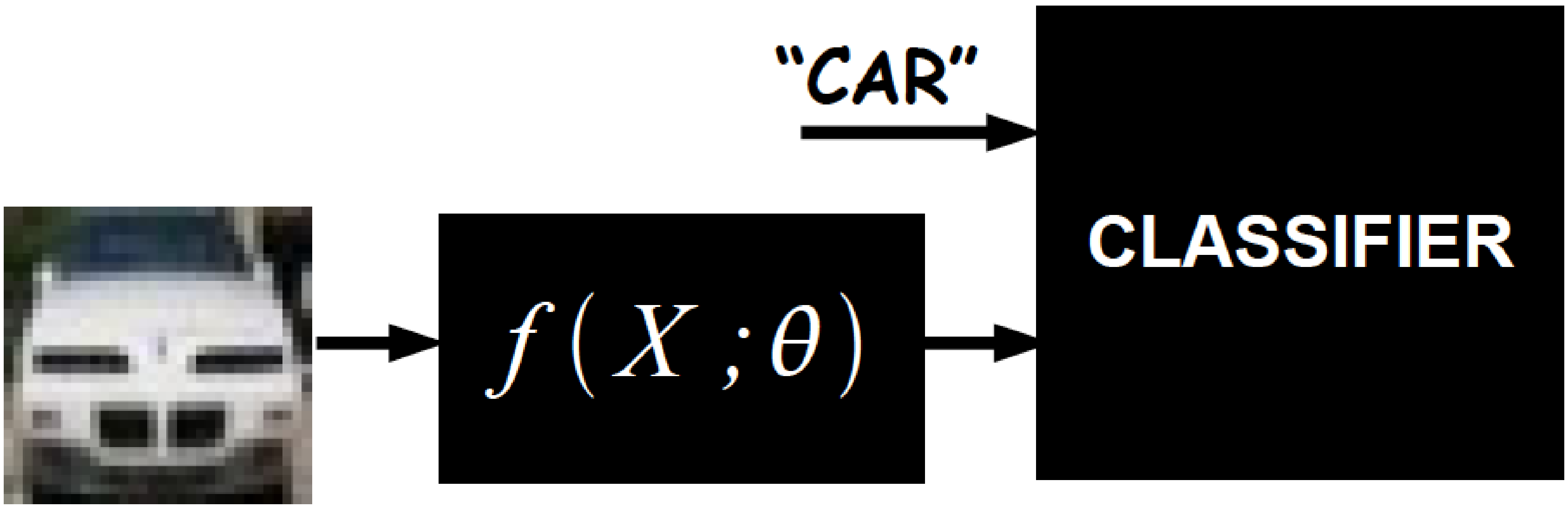


IDEA: Use data to optimize features for the given task.

Ranzato

Slide Credit: Marc'Aurelio Ranzato

Building an Object Recognition System



What we want: Use parameterized function such that

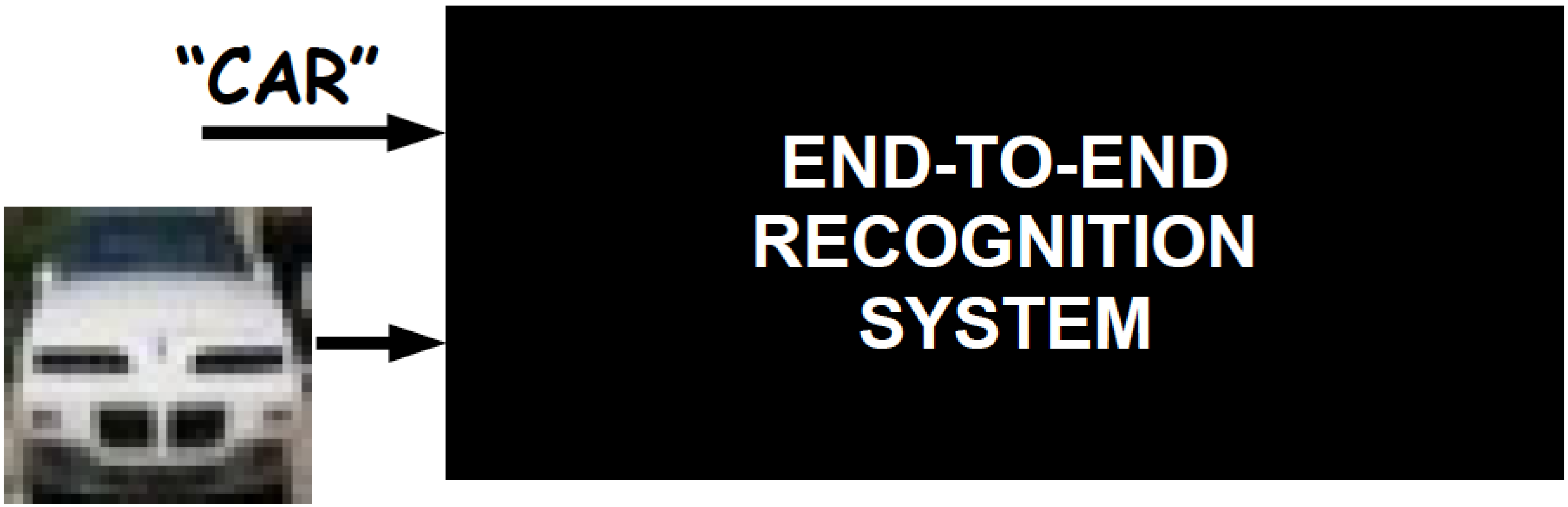
a) features are computed efficiently

b) features can be trained efficiently

Ranzato

Slide Credit: Marc'Aurelio Ranzato

Building an Object Recognition System



- Everything becomes adaptive.
- No distiction between feature extractor and classifier.
- Big non-linear system trained from raw pixels to labels.

Slide Credit: Marc'Aurelio Ranzato

Visualizing Learned Filters

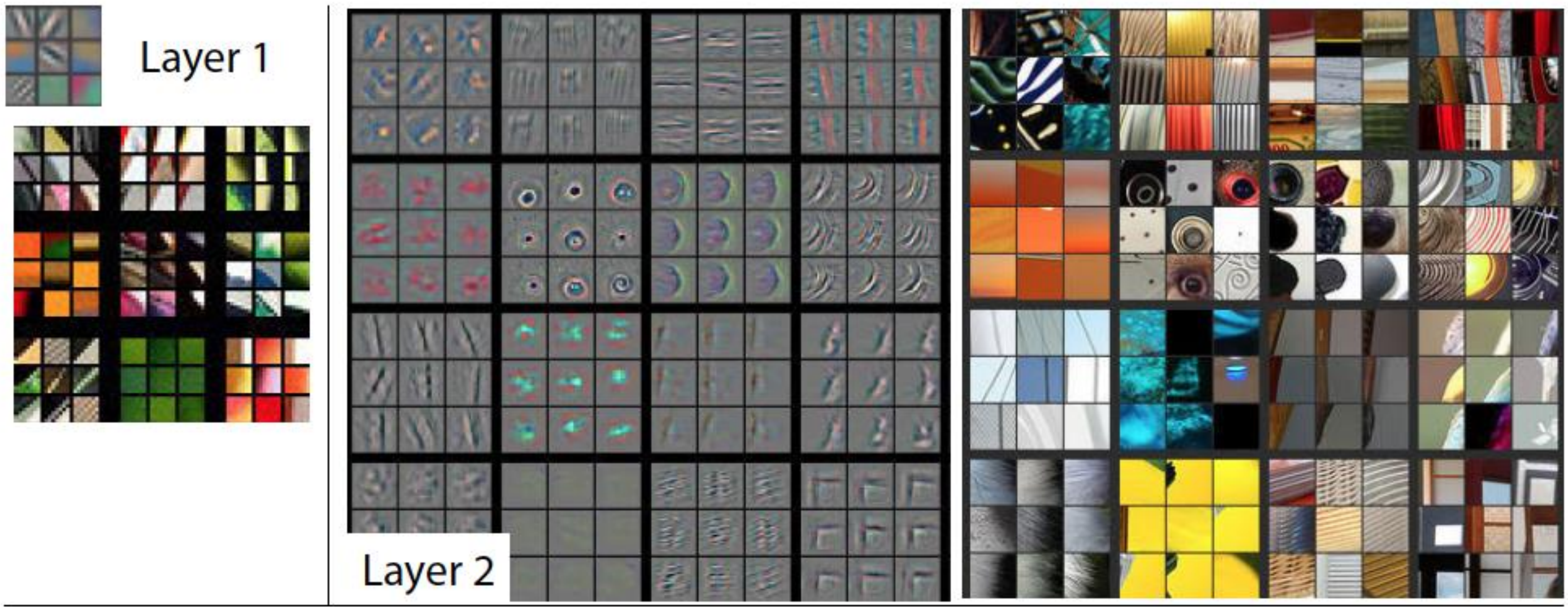


Figure Credit: [Zeiler & Fergus ECCV14]

Visualizing Learned Filters

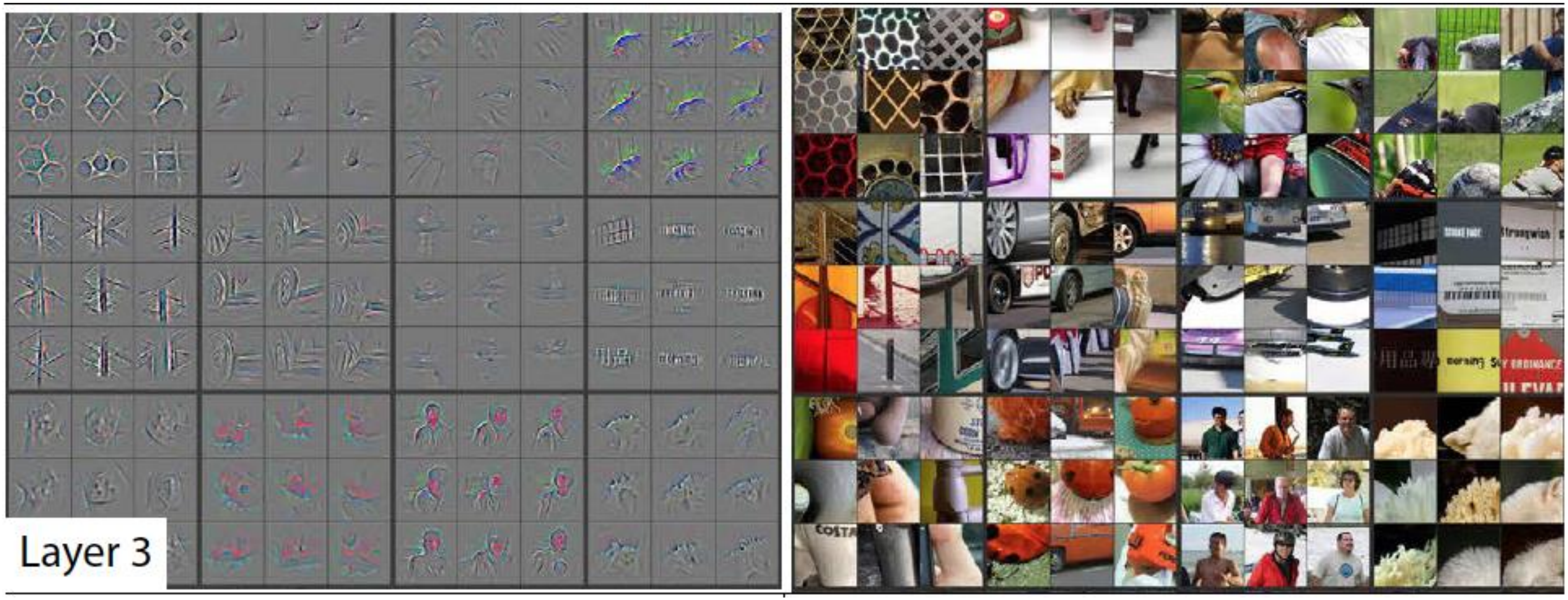


Figure Credit: [Zeiler & Fergus ECCV14]

Visualizing Learned Filters