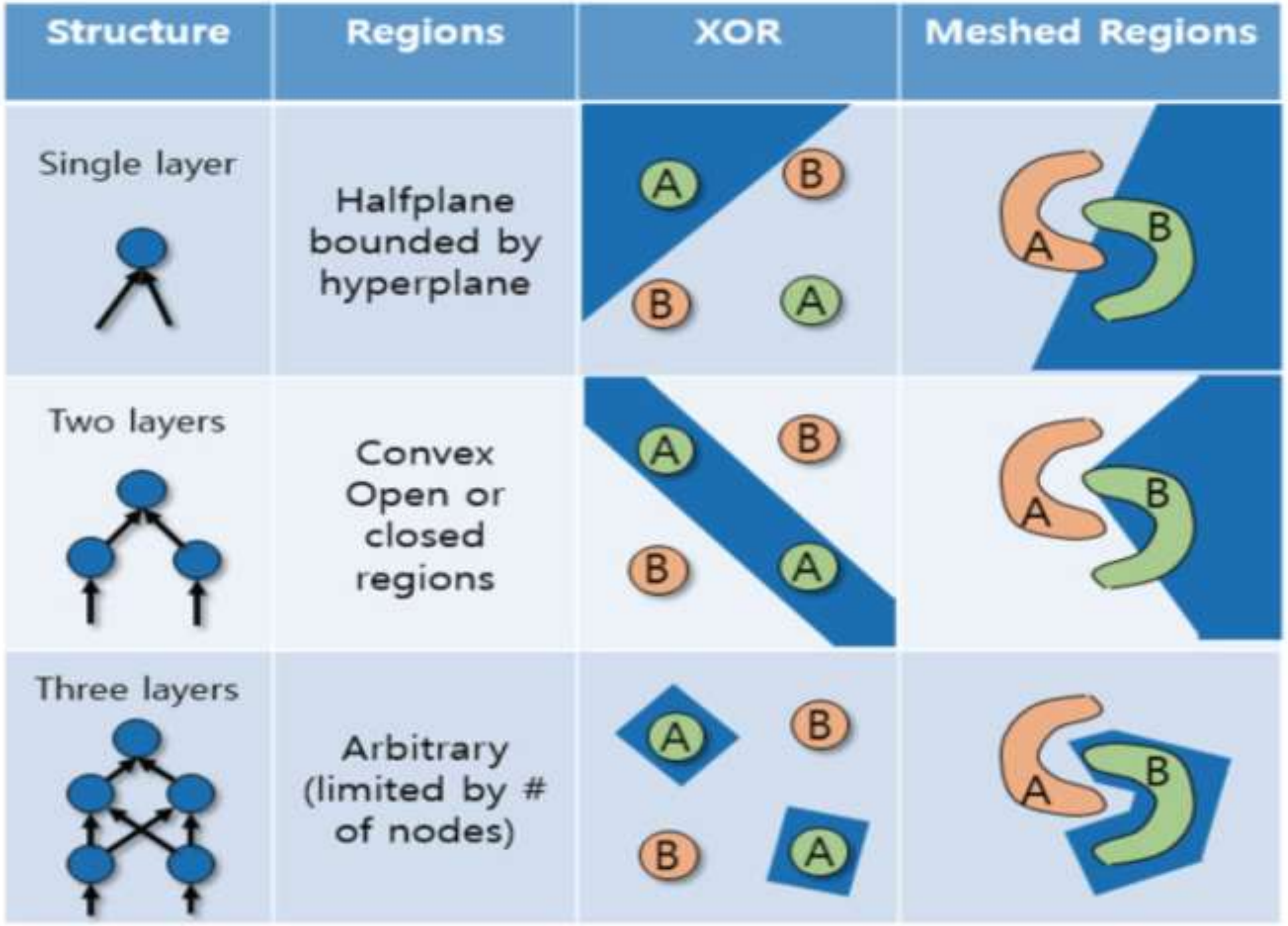
Artificial Inteligence
& Computer Vision
L a b o r a t o r y

Multilayer Perceptron

Multi-layer Perceptron

- Why MLP is needed?



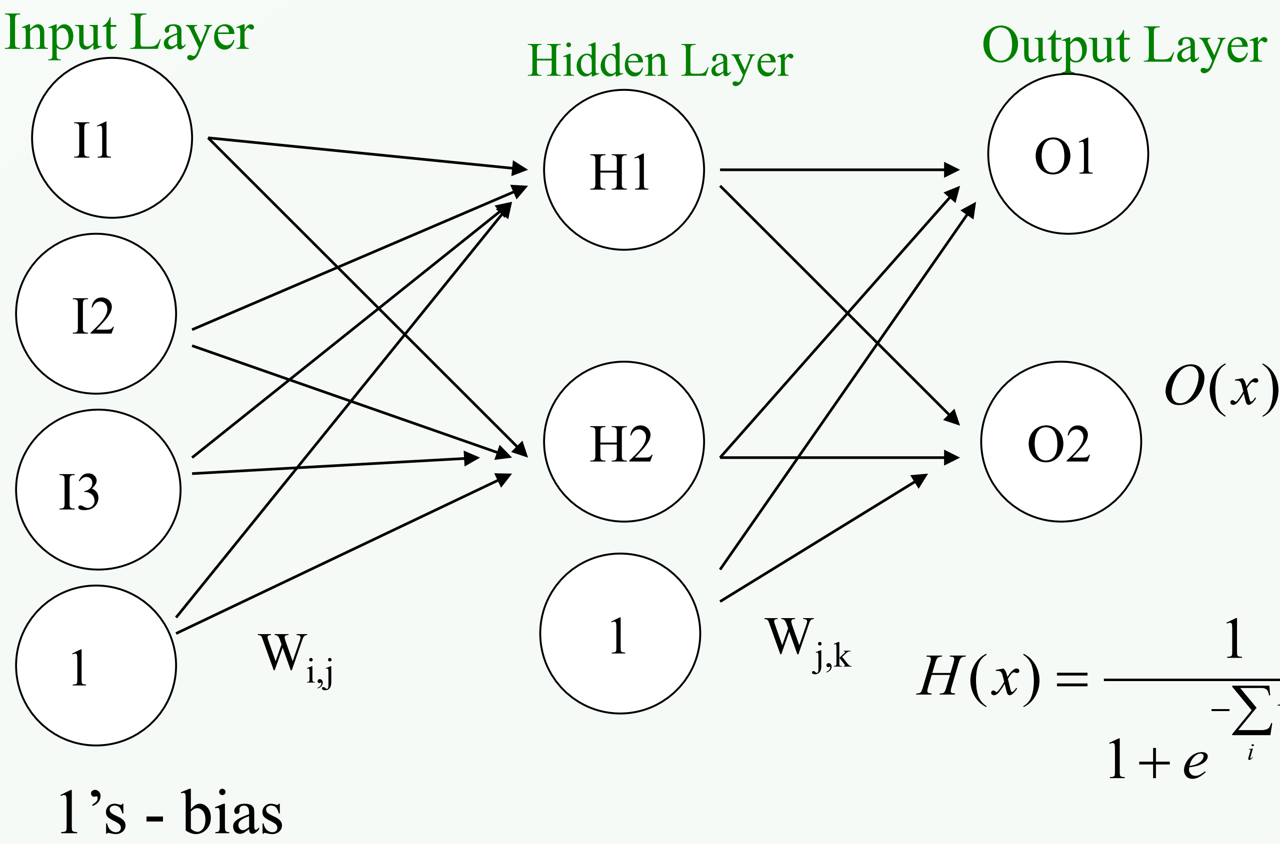
Structure	Regions	XOR	Meshed Regions
Single layer	Halfplane bounded by hyperplane	A B B A	A B
Two layers	Convex Open or closed regions	A B B A	A B
Three layers	Arbitrary (limited by # of nodes)	A B B A	A B

Multiple boundaries needed (e.g. XOR problem) →**Multiple units**

More complex regions needed (e.g. Polygons) → **Multiple layers**

Multilayer Perceptron

- Attributed to Rumelhart and McClelland, late 70's
- To bypass the linear classification problem, we can construct *multilayer* networks.
- Typically we have *fully connected*, *feedforward* networks.



$$O(x) = \frac{1}{1 + e^{-\sum_j w_{j,x} H_j}}$$

$$H(x) = \frac{1}{1 + e^{-\sum_i w_{i,x} I_i}}$$

Hidden Units

- Hidden units are nodes that are situated between the input nodes and the output nodes.
- Hidden units allow a network to learn non-linear functions.
- Hidden units allow the network to represent combinations of the input feat ures.

Hidden Units

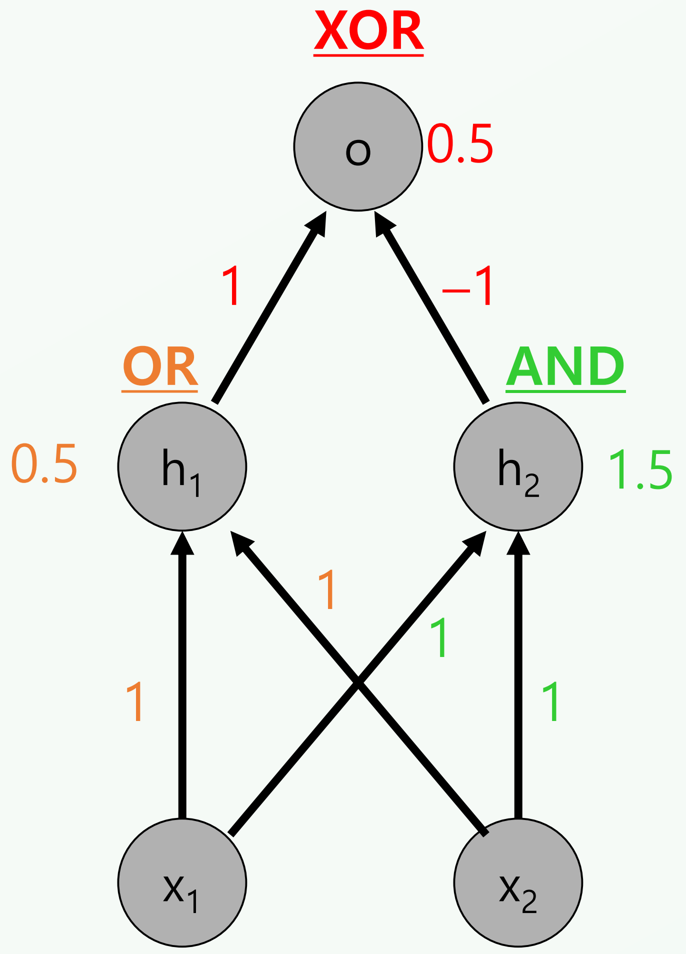
- Boolean XOR

$$X1 \oplus X2 \Leftrightarrow (X1 \vee X2) \wedge \neg(X1 \wedge X2)$$

Not Linear separable → Cannot be represented by a single-layer perceptron

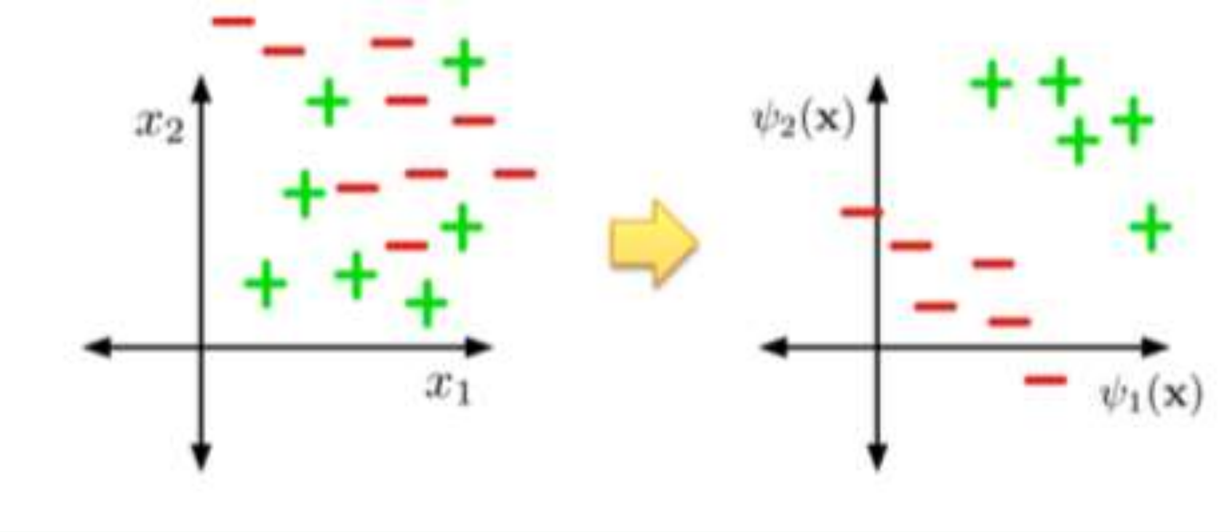
Let's consider a single hidden layer network, using as building blocks threshold units.

$$w_1\ x_1 + w_2\ x_2 - w_0 > 0$$



Hidden Units

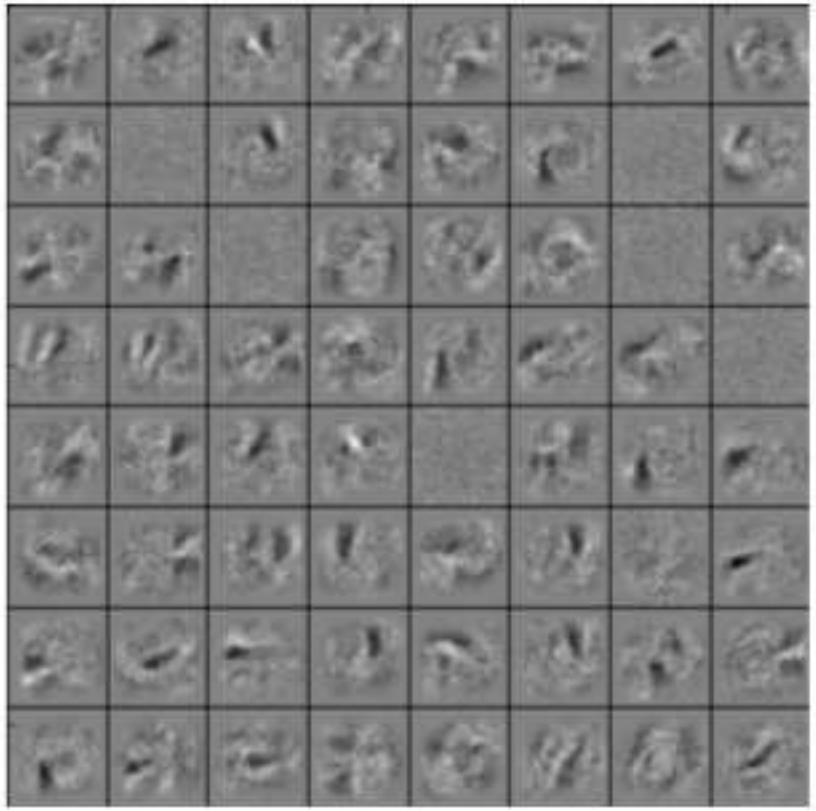
- Neural nets can be thought of as a way of learning nonlinear feature mapping
- The last hidden layer can be thought of as a feature map
- The last layer weights can be thought of as a linear model using those features



Hidden Units

- The last hidden layer can be thought of as a feature map

MNIST handwritten digit dataset



A subset of learned first layer features: many of them pick up oriented image

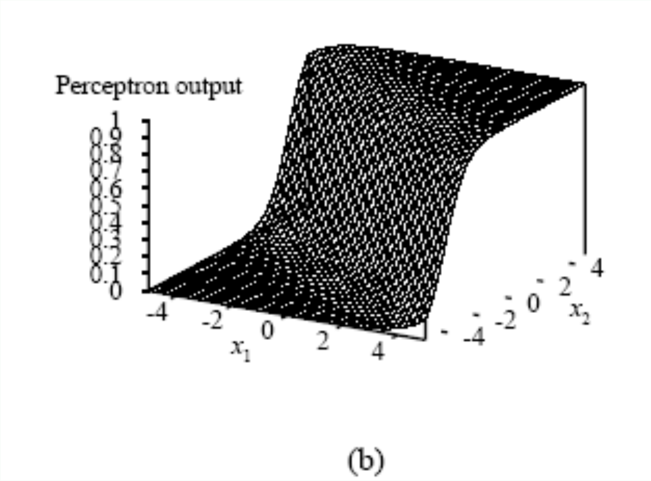
Expressiveness of MLP: Soft Threshold

Artificial Inteligence & Computer Vision Laboratory

- Advantage of adding hidden layers

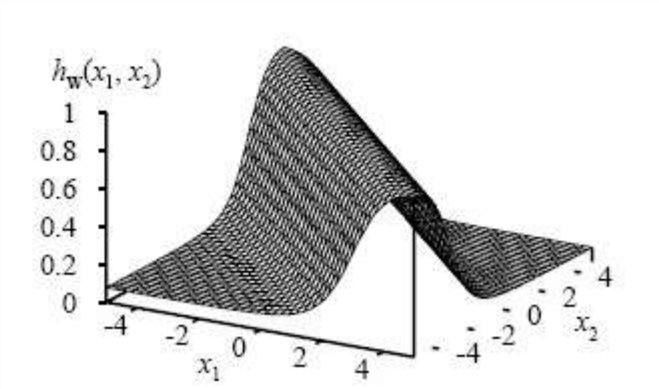
→ It enlarge the space of hypotheses that the network can represent

Example: we can think of each hidden unit as a perceptron that represents a soft threshold function in the input space, and an output unit as as a soft-thresholded linear combination of several such functions.

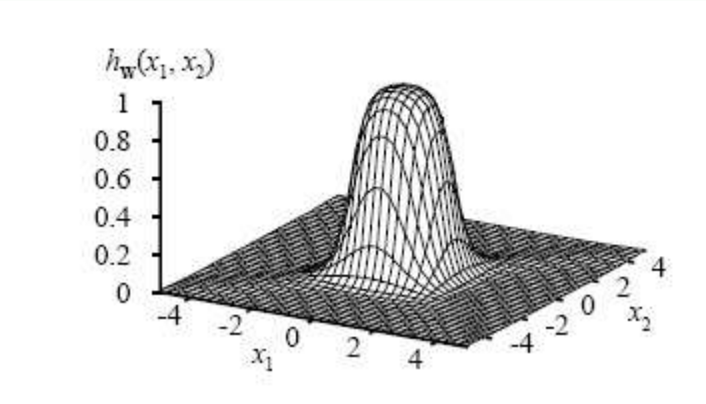


Soft threshold function

Expressiveness of MLP: Soft Threshold



a)



b)

(a) The result of combining two opposite-facing soft threshold functions to produce a ridg e.

(b) The result of combining two ridges to produce a bump.

Add bumps of various sizes and locations to any surface

All continuous functions w/ 2 layers, all functions w/ 3 layers

Expressiveness of MLP

- With a single, sufficiently large hidden layer, it is possible to represent a ny continuous function of the inputs with arbitrary accuracy;

- With two layers, even discontinuous functions can be represented.

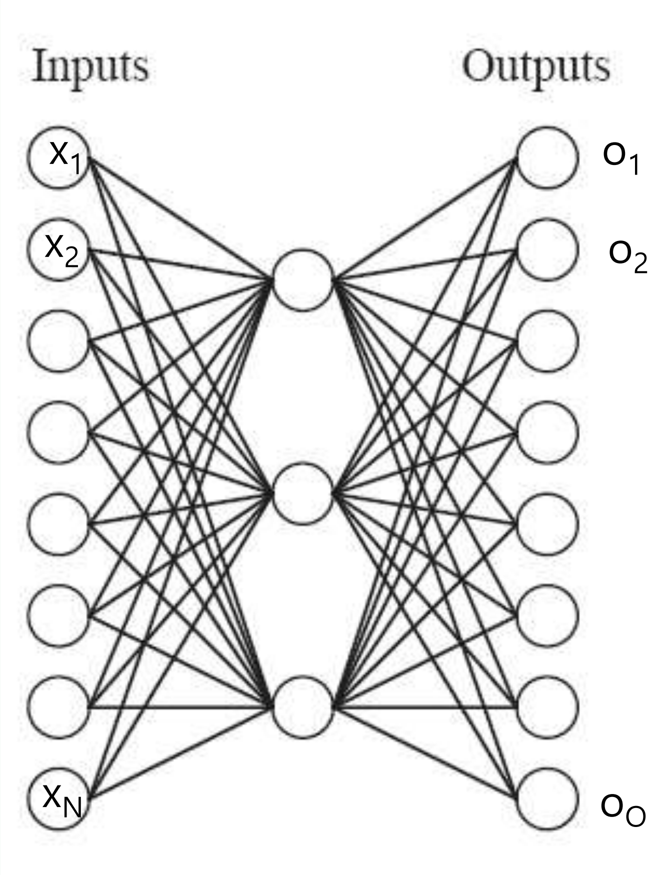
  - The proof is complex → main point, required number of hidden u nits grows exponentially with the number of inputs.
  - For example, $2^n/n$ hidden units are needed to encode all Boolean f unctions of n inputs.

- Issue: For any particular network structure, it is harder to characterize ex actly which functions can be represented and which ones cannot.

Multi-Layer Feedforward Networks

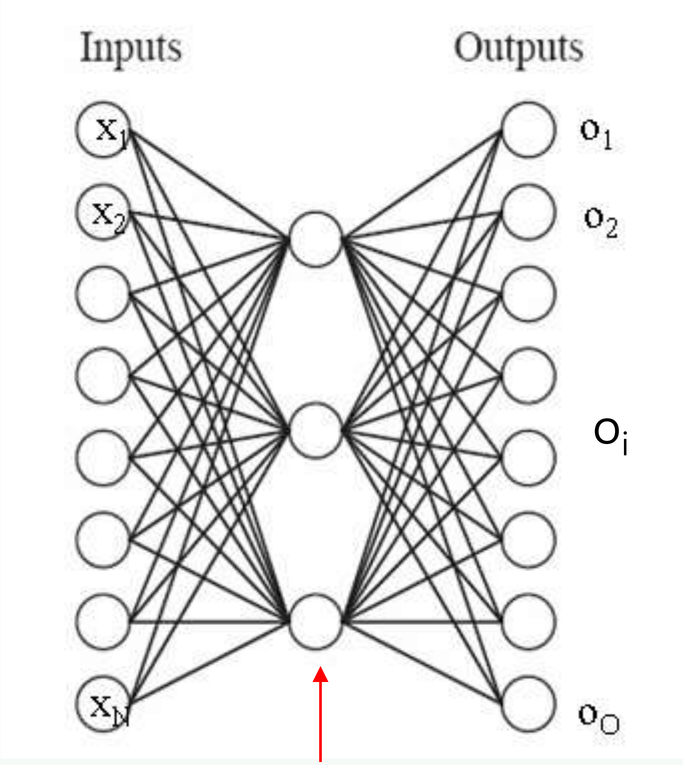
Artificial Inteligence & Computer Vision Laboratory

Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].



$$o_i = g\left(\sum_h w_{h,i}\, g\left(\sum_j w_{j,h}\, x_j\right)\right)$$

Learning Algorithms for MLP



o_i

$Err_1 = y_1 - o_1$

$Err_2 = y_2 - o_2$

$Err_i = y_i - o_i$

$Err_o = y_o - o_o$

How to compute the errors for the hidden units?

Clear error at the output layer

Goal: minimize sum squared errors

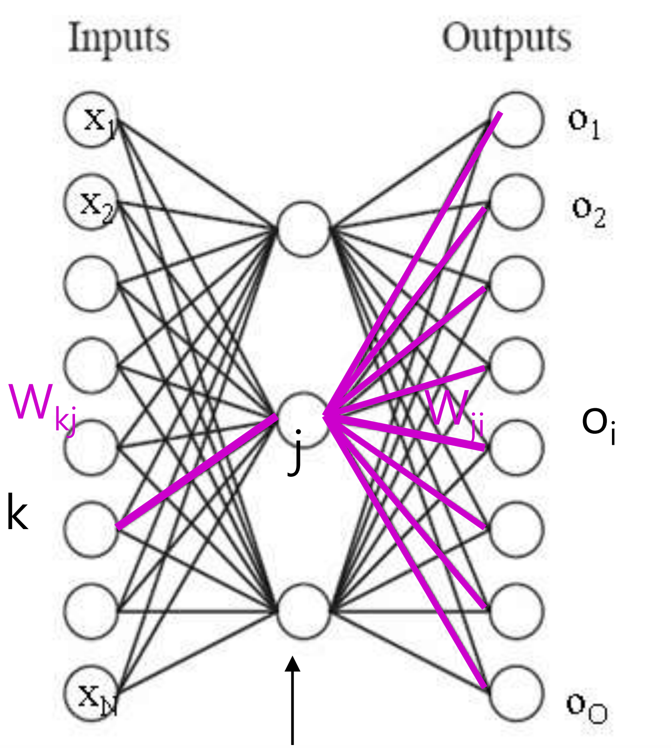
$$E = \frac{1}{2}\sum_i (y_i - o_i)^2$$

$$o_i = g\left(\sum_h w_{h,i}\, g\left(\sum_j w_{j,h}\, x_j\right)\right)$$

parameterized function of inputs: weights are the parameters of the function.

We can **back-propagate** the error from the output layer to the hidden layers.
The back-propagation process emerges directly from a
derivation of the overall error gradient.

Backpropagation Learning Algorithms for MLP



Perceptron update:

$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j$$

$Err_i = y_i - o_i$

Output layer weight update (similar to perceptron)

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

$$\Delta_i = Err_i \times g'(in_i)$$

Hidden layer: **back-propagate** the error from the output layer:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j \ .$$

$$\Delta_j = g'(in_j) \underbrace{\sum_i W_{j,i}\Delta_i}_{} \ .$$

Err_j → "Error" for hidden node j

Hidden node j is "responsible" for some fraction of the error i in each of the output nodes to which it connects → depending on the strength of the connection between the hidden node and the output node i.

Backpropagation Training (Overview)

Artificial Inteligence & Computer Vision Laboratory

Optimization Problem

- Obj.: minimize E

$$E = \frac{1}{2}\sum_i (y_i - a_i)^2 \,,$$

Choice of learning rate α

How many restarts (local optima) of search to find good optimum of objective function?

Variables: network weights w_{ij}

Algorithm: local search via gradient descent.

Randomly initialize weights.

Until performance is satisfactory, cycle through examples (epochs):

- Update each weight:

Output node:

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

$$\Delta_i = Err_i \times g'(in_i)$$

Hidden node:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j$$

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i \,.$$

See derivation details in the next slides

Learning Algorithms for MLP

- Similar to the perceptron learning algorithm:
  - One minor difference is that we may have several outputs, so we have an output vector $h_W(x)$ rather than a single value, and each example has an output vector y.
  - The major difference is that, whereas the error $y - h_W$ at the perceptron output layer is clear, the error at the hidden layers seems mysterious because the training data does not say what value the hidden nodes should have

We can **back-propagate** the error from the output layer to the hidden layers. The back-propagation process emerges directly from a derivation of the overall error gradient.

Back Propagation Learning

Artificial Inteligence
& Computer Vision
Laboratory

Output layer: same as for single-layer perceptron,

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

where $\Delta_i = Err_i \times g'(in_i)$

Perceptron update:

$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j$$

Err_i → i^{th} component of vector y - h_W

Hidden layer: back-propagate the error from the output layer:

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i \ .$$

Update rule for weights in hidden layer:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j \ .$$

Hidden node j is "responsible"
for some fraction of the error i
in each of the output nodes to which it connec
ts → depending on . the strength of the conne
ction between the hidden node and the output
node i.

Back Propagation Learning

- Derivation

The squared error on a single example is defined as

$$E = \frac{1}{2}\sum_i (y_i - a_i)^2 ,$$

where the sum is over the nodes in the output layer.

$$\begin{aligned}
\frac{\partial E}{\partial W_{j,i}} &= -(y_i - a_i)\frac{\partial a_i}{\partial W_{j,i}} = -(y_i - a_i)\frac{\partial g(in_i)}{\partial W_{j,i}} \\
&= -(y_i - a_i)g'(in_i)\frac{\partial in_i}{\partial W_{j,i}} = -(y_i - a_i)g'(in_i)\frac{\partial}{\partial W_{j,i}}\left(\sum_j W_{j,i}a_j\right) \\
&= -(y_i - a_i)g'(in_i)a_j = -a_j\Delta_i
\end{aligned}$$

Back Propagation Learning

- Derivation

$$
\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= -\sum_i (y_i - a_i)\frac{\partial a_i}{\partial W_{k,j}} = -\sum_i (y_i - a_i)\frac{\partial g(in_i)}{\partial W_{k,j}} \\
&= -\sum_i (y_i - a_i) g'(in_i)\frac{\partial in_i}{\partial W_{k,j}} = -\sum_i \Delta_i \frac{\partial}{\partial W_{k,j}}\left(\sum_j W_{j,i}a_j\right) \\
&= -\sum_i \Delta_i W_{j,i}\frac{\partial a_j}{\partial W_{k,j}} = -\sum_i \Delta_i W_{j,i}\frac{\partial g(in_j)}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial in_j}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial}{\partial W_{k,j}}\left(\sum_k W_{k,j}a_k\right) \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j) a_k = -a_k \Delta_j
\end{aligned}
$$

Back Propagation Learning

- Derivation

$$\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= -\sum_i (y_i - a_i)\frac{\partial a_i}{\partial W_{k,j}} = -\sum_i (y_i - a_i)\frac{\partial g(in_i)}{\partial W_{k,j}} \\
&= -\sum_i (y_i - a_i) g'(in_i)\frac{\partial in_i}{\partial W_{k,j}} = -\sum_i \Delta_i \frac{\partial}{\partial W_{k,j}}\left(\sum_j W_{j,i} a_j\right) \\
&= -\sum_i \Delta_i W_{j,i} \frac{\partial a_j}{\partial W_{k,j}} = -\sum_i \Delta_i W_{j,i} \frac{\partial g(in_j)}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j) \frac{\partial in_j}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j) \frac{\partial}{\partial W_{k,j}}\left(\sum_k W_{k,j} a_k\right) \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j) a_k = -a_k \Delta_j
\end{aligned}$$

Back Propagation Learning Algorithm

Artificial Inteligence
& Computer Vision
Laboratory

function BACK-PROP-LEARNING(*examples*, *network*) **returns** a neural network
inputs: *examples*, a set of examples, each with input vector **x** and output vector **y**
network, a multilayer network with L layers, weights $W_{j,i}$, activation function g

repeat
for each e **in** *examples* **do**
for each node j in the input layer **do** $a_j \leftarrow x_j[e]$
for $\ell = 2$ **to** M **do**
$in_i \leftarrow \sum_j W_{j,i}\, a_j$
$a_i \leftarrow g(in_i)$
for each node i in the output layer **do**
$\Delta_i \leftarrow g'(in_i) \times (y_i[e] - a_i)$
for $\ell = M - 1$ **to** 1 **do**
for each node j in layer ℓ **do**
$\Delta_j \leftarrow g'(in_j) \sum_i W_{j,i}\, \Delta_i$
for each node i in layer $\ell + 1$ **do**
$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$
until some stopping criterion is satisfied
return NEURAL-NET-HYPOTHESIS(*network*)

Design Decisions

Artificial Inteligence
& Computer Vision
Laboratory

- **Network architecture**
  - How many hidden layers? How many hidden units per layer?
    - Given too many hidden units, a neural net will simply memorize the input patterns (overfitting).
    - Given too few hidden units, the network may not be able to represent all of the necessary generalizations (underfitting).
  - How should the units be connected? (Fully? Partial? Use domain knowledge?)

Summary

- Perceptrons (one-layer networks) limited expressive power—they can l earn only linear decision boundaries in the input space.
- Single-layer networks have a simple and efficient learning algorithm;
- Multi-layer networks are sufficiently expressive
  - they can represent general nonlinear function
  - they can be trained by gradient descent, i.e., error back-propagation.
- Problems of Generalization vs. Memorization.
  - With too many units, we will tend to memorize the input and not generalize well.
  - Someschemes exist to “prune” the neural network.
- MLP harder to train because of the abundance of local minima and the high dimensionality of the weight space
- Many applications: speech, driving, handwriting, fraud detection, etc.