Neural Network

Eun Yi Kim

Artificial Inteligence
& Computer Vision
Laboratory

I N D E X

What is a deep learning?

Perceptron

- Activation function
- LMS learning

Multiple Layer Perceptron

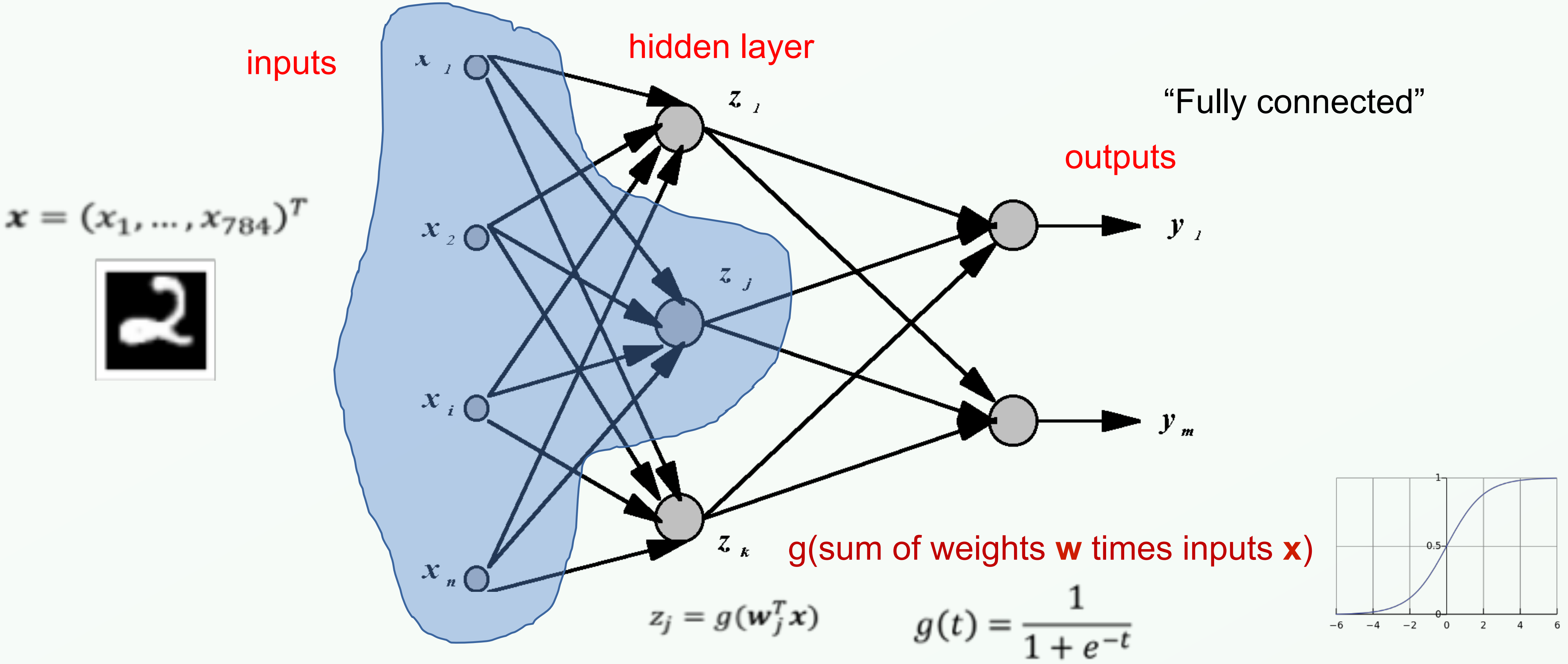
- Hidden units
- Backpropagation

Summary

Artificial Inteligence
& Computer Vision
Laboratory

What is deep learning?

- Deep learning refers to training a neural network
  - A function that fits some data



Neural Networks

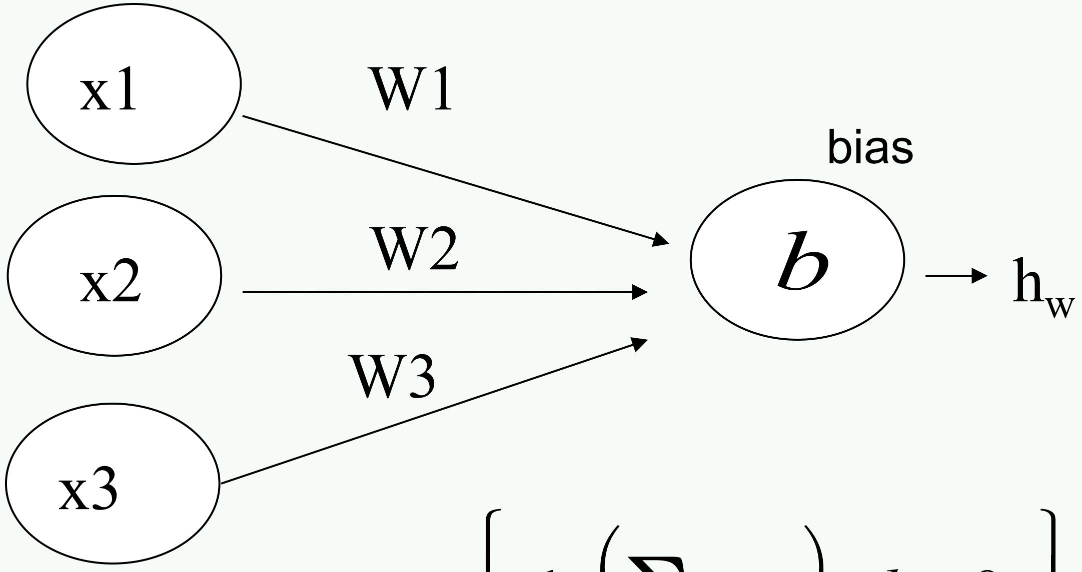
- Perceptron
- Multi-layer perceptron
- Convolutional Neural Network
- Region based CNN (RCNN)
- Recurrent Neural Network
- Long Short-term memory
- Deep Belief Net
- Deep Q-Net
- Auto-encoder
- Memory Network
- Generative Adversarial Network
- Deep Residual Net

Perceptron

Perceptron

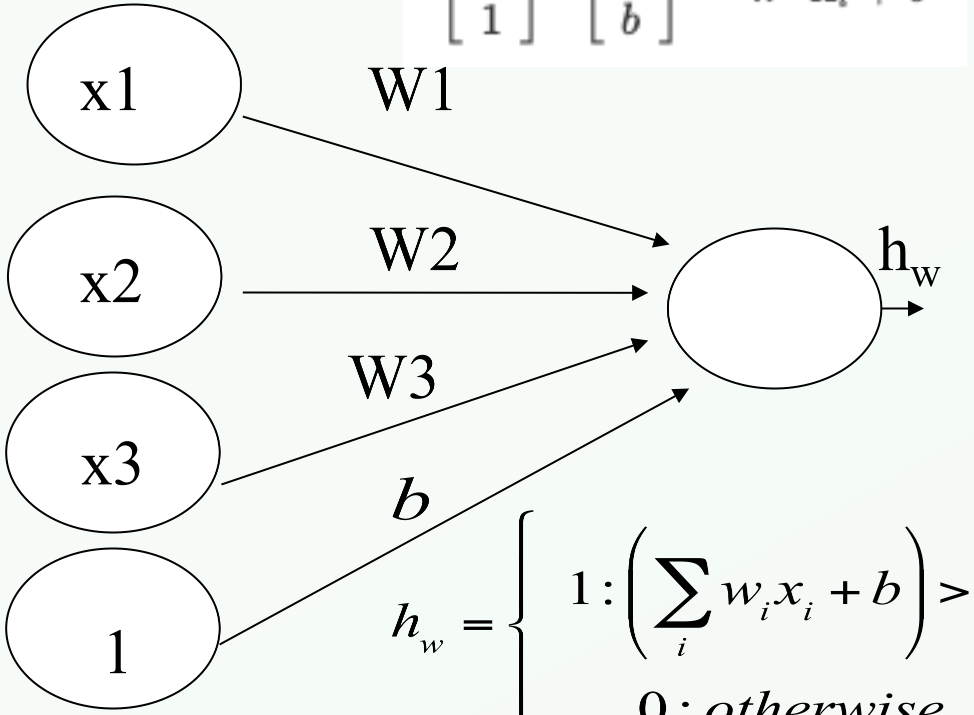
- Initial proposal of connectionist networks
- Rosenblatt, 50's and 60's
- Essentially a linear discriminant composed of nodes, weights

$$\begin{bmatrix} \mathbf{x}_i \\ 1 \end{bmatrix}^\top \begin{bmatrix} \mathbf{w} \\ b \end{bmatrix} = \mathbf{w}^\top \mathbf{x}_i + b$$



Hypothesis $h_w = \begin{cases} 1 : \left(\sum_i w_i x_i\right) + b > 0 \\ 0 : otherwise \end{cases}$

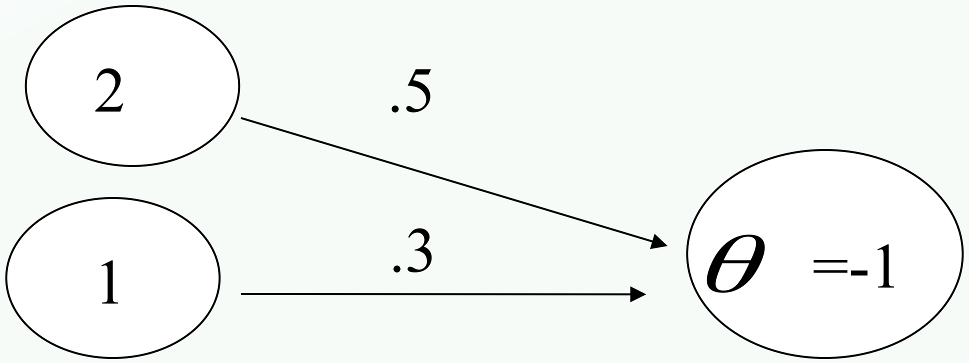
or



$$h_w = \begin{cases} 1 : \left(\sum_i w_i x_i + b\right) > 0 \\ 0 : otherwise \end{cases}$$

Perceptron Example

Artificial Inteligence & Computer Vision Laboratory



$2(0.5) + 1(0.3) + -1 = 0.3 > 0$, so $h_w=1$

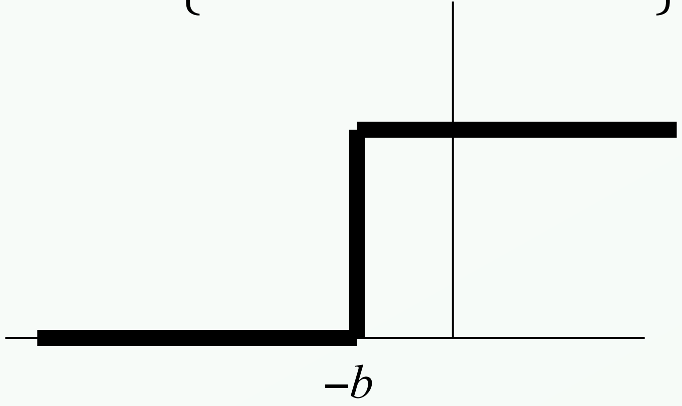
Learning Procedure:

- Randomly assign weights (between 0 and1)
- Present inputs from training data
- Get output hw, nudge weights to gives results toward our desired output T
- Repeat; stop when no errors, or enough epochs completed

Perceptron with Activation Function

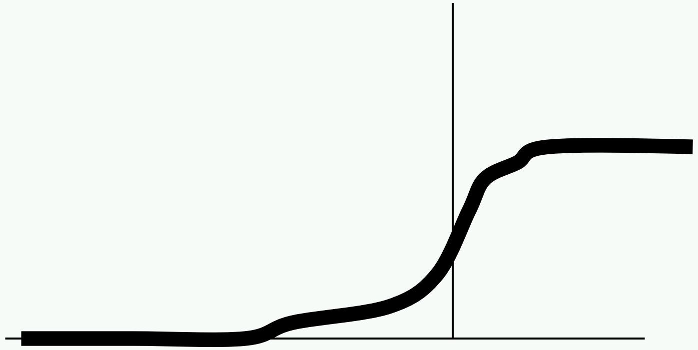
Old:

$$h_w = \begin{cases} 1: \sum_j w_j x_j + b > 0 \\ 0: otherwise \end{cases}$$



New:

$$h_w = \frac{1}{1 + e^{-\sum_j w_j x_l + b}}$$



Activation Function: g

$$h_w = \begin{cases} 1: \left(\sum_i w_i x_i + b\right) > 0 \\ 0: otherwise \end{cases}$$

Perceptron is essentially linear classifier

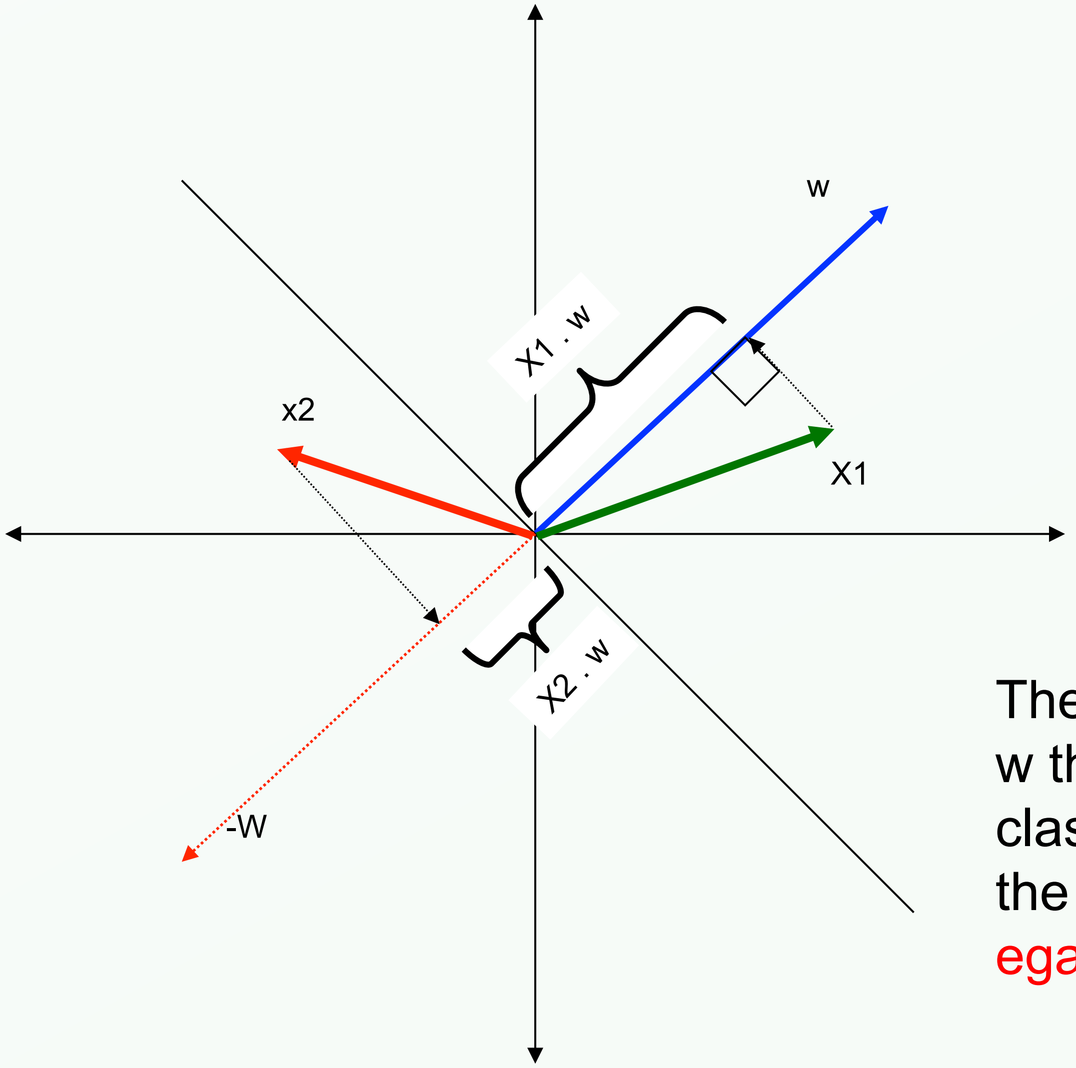
A Linear Classifier

- Let's simplify life by assuming:
  - Every instance is a vector of real numbers, $\mathbf{x}=(x_1,\ldots,x_n)$. (Notation: boldface **x** is a vector.)
  - There are only two classes, $y=(+1)$ and $y=(-1)$
- A linear classifier is vector **w** of the same dimension as **x** that is used to make this prediction:

$$\hat{y} = \mathrm{sign}(w_1 x_1 + w_2 x_2 + \ldots + w_n x_n) = \mathrm{sign}(\mathbf{w} \cdot \mathbf{x})$$

$$\mathrm{sign}(x) = \begin{Bmatrix} +1 & \text{if } x \geq 0 \\ -1 & \text{if} < 0 \end{Bmatrix}$$

A Linear Classifier

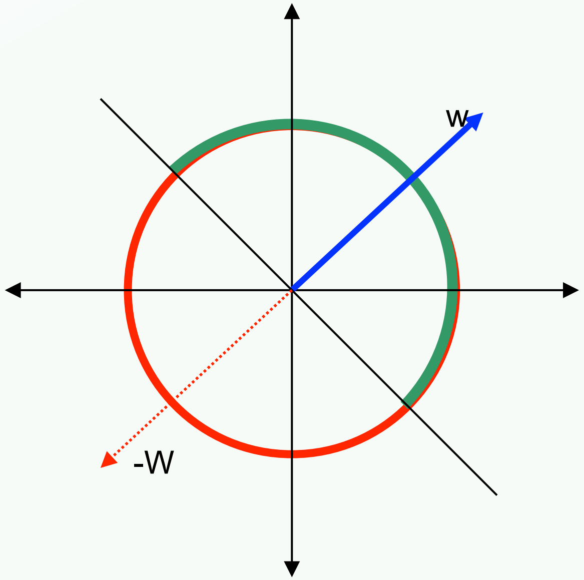


Visually, x · w is th
e distance you get
 if you “project x o
nto w”

In 3d: line→plane
In 4d: plane→hyperplane
…

The line perpendicular to w that divides the vectors classified as positive from the vectors classified as negative.

A Linear Classifier



Notice that the separating hyperplane goes thr ough the origin…if we don't want this we can p reprocess our examples:

$$\cancel{\mathbf{x} = \langle x_1, x_2, \ldots, x_n \rangle}$$

$$\mathbf{x} = \langle 1, x_1, x_2, \ldots, x_n \rangle$$

$$\cancel{\hat{y} = \text{sign}(w_1 x_1 + w_2 x_2 + \ldots + w_n x_n) = \text{sign}(\mathbf{w} \cdot \mathbf{x})}$$

$$\hat{y} = \text{sign}(w_0 1 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n) = \text{sign}(\mathbf{w} \cdot \mathbf{x})$$

Naïve Bayes as a Linear Classifier

$$g_i(\mathbf{x}) = -\frac{1}{2}(\mathbf{x}-\mu_\mathbf{i})^t\Sigma_i^{-1}(\mathbf{x}-\mu_\mathbf{i}) - \frac{d}{2} ln 2\pi - \frac{1}{2}\ln|\Sigma_i| + \ln P(\omega_i)$$

- **$\Sigma_i = \sigma^2 I$** (diagonal matrix)
  - This is true when features are uncorrelated (or statistically independent) with same variance

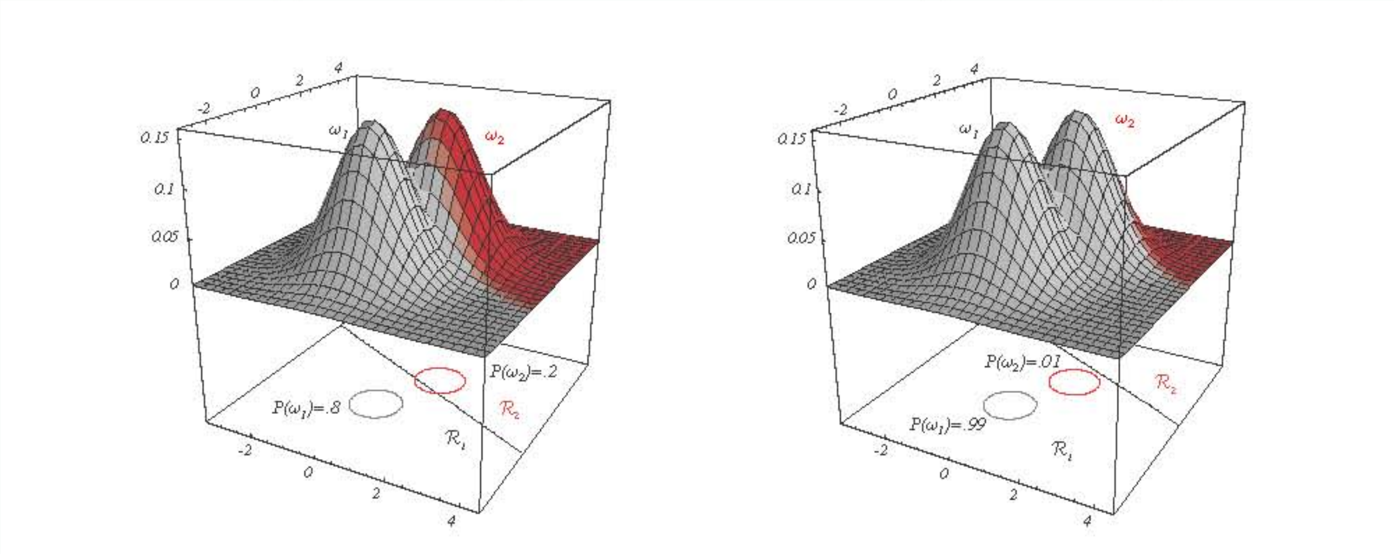
- Decision boundary is determined by hyperplanes; setting $g_i(\mathbf{x}) = g_j(\mathbf{x})$:

$$\boxed{\mathbf{w}^t(\mathbf{x}-\mathbf{x_0}) = 0}$$

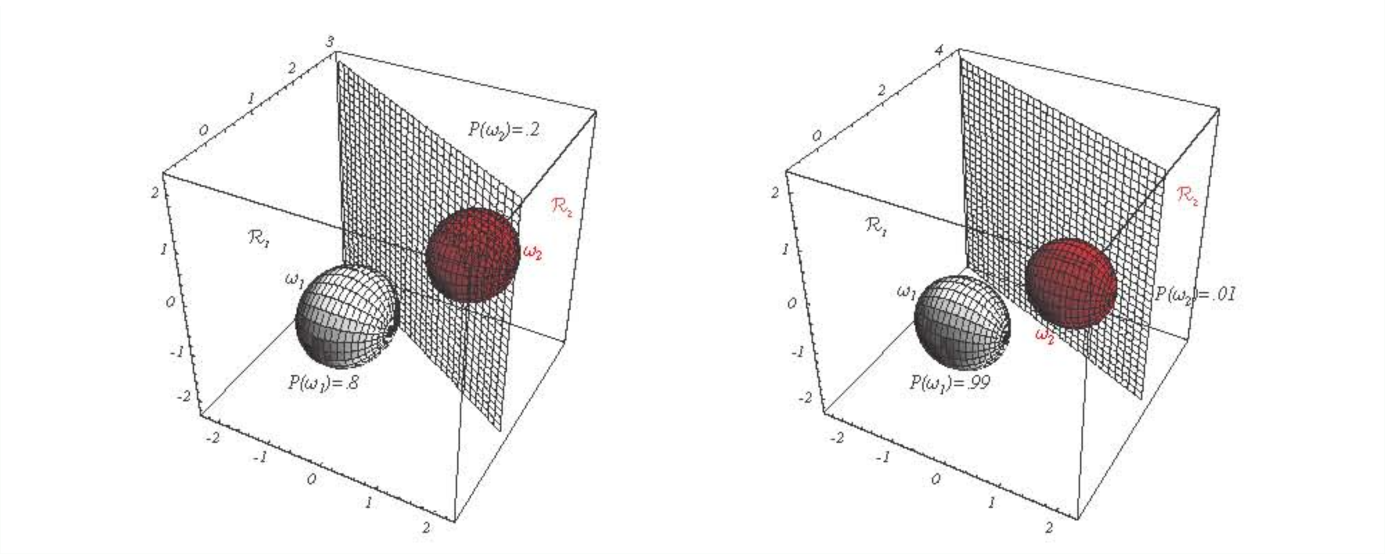
where $\mathbf{w} = \mu_i - \mu_j$, and $\mathbf{x}_0 = \frac{1}{2}(\mu_i + \mu_j) - \frac{\sigma^2}{\|\mu_i - \mu_j\|^2} \ln \frac{P(\omega_i)}{P(\omega_j)}(\mu_i - \mu_j)$

Naïve Bayes as a Linear Classifier

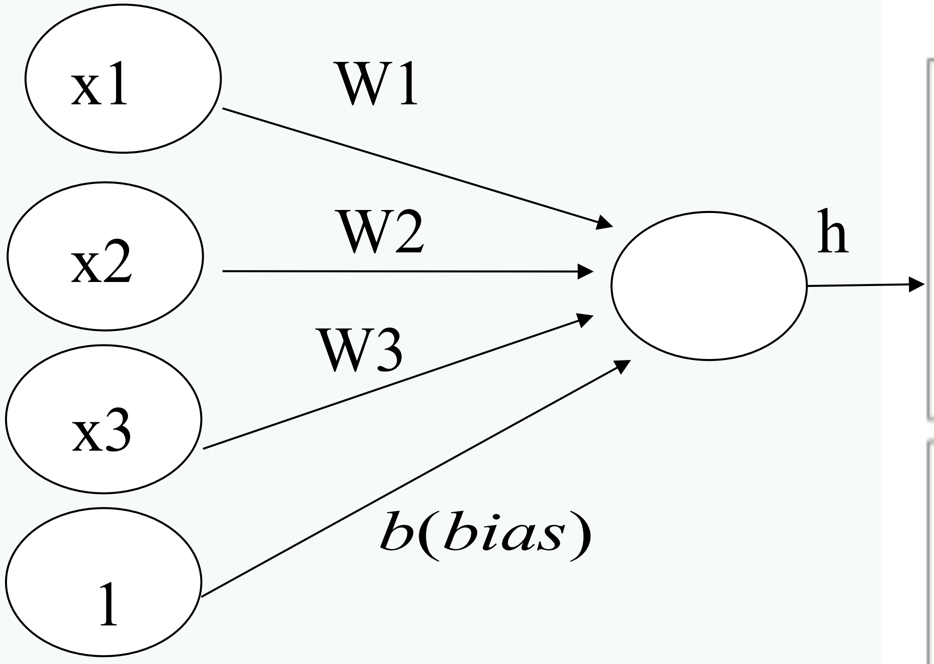
Artificial Inteligence & Computer Vision Laboratory



If $P(\omega_i) \neq P(\omega_j)$, then $\mathbf{x_0}$ shifts away from the most likely category.

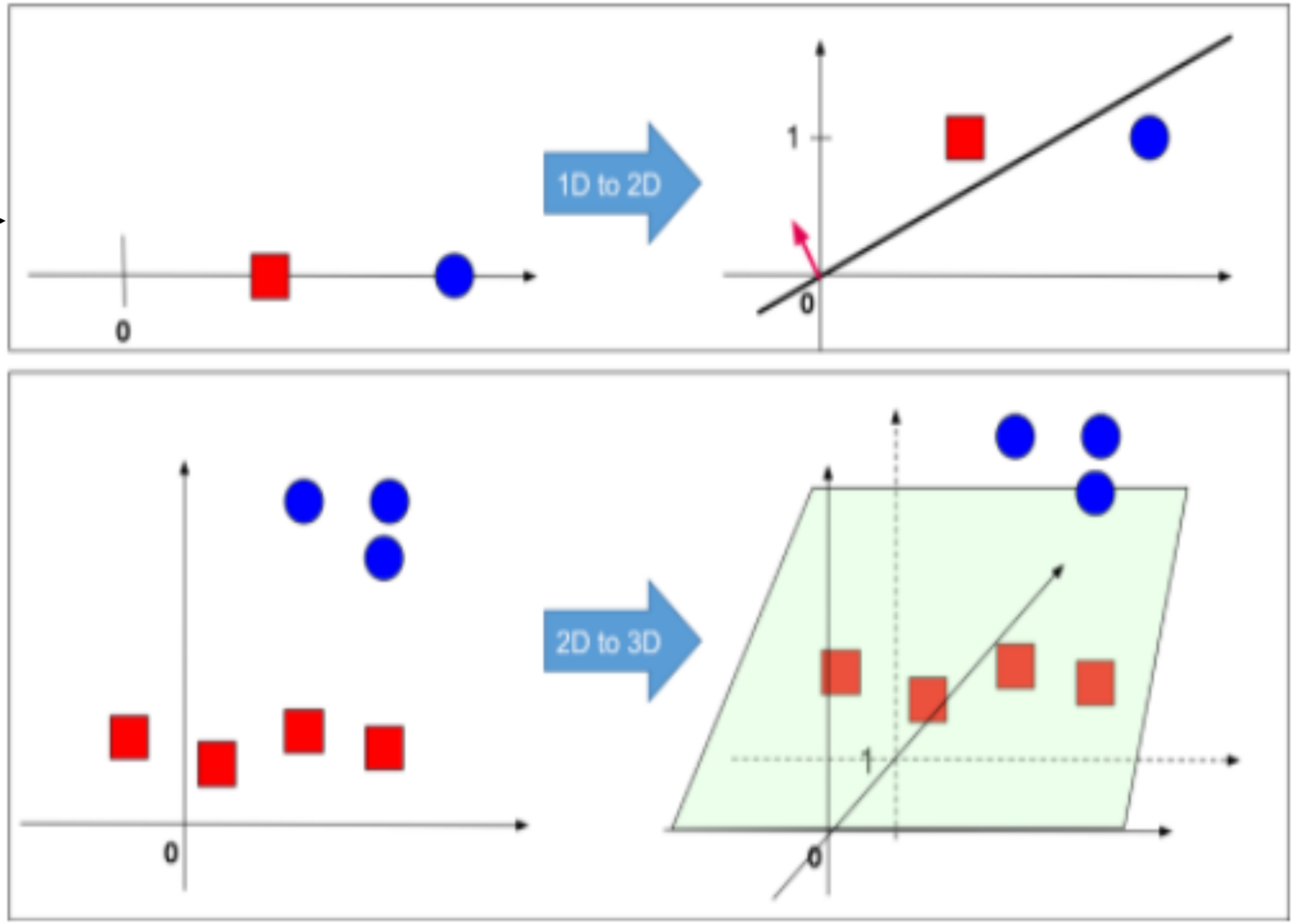


Perceptron as a Linear Classifier

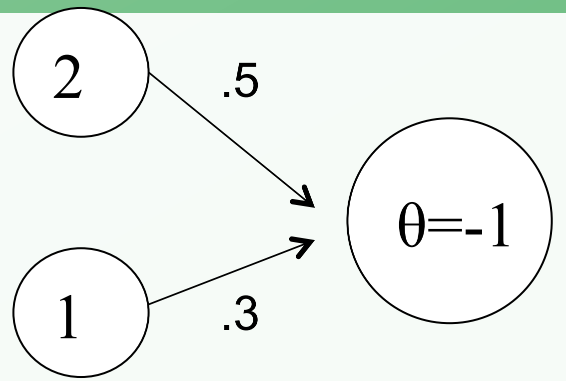


$$h = \left\{ \begin{array}{l} 1 : g\left(\sum_i w_i x_i + b\right) > 0 \\ 0 : otherwise \end{array} \right\}$$

$$h(\mathbf{x}_i) = \mathrm{sign}(\mathbf{w}^\top \mathbf{x})$$



Simple Perception Training



$$w_i(t+1) = w_i(t) + \Delta w_i(t)$$

$$\Delta w_i(t) = (y - h_w) I_i$$

Weights include Threshold. y=Desired, h_w=Actual output.

Example: y=0, h_w=1, W1=0.5, W2=0.3, I1=2, I2=1, θ=-1

$$w_1(t+1) = 0.5 + (0-1)(2) = -1.5$$

$$w_2(t+1) = 0.3 + (0-1)(1) = -0.7$$

$$w_\theta(t+1) = -1 + (0-1)(1) = -2$$

If we present this input again, we'd output 0 instead

A learning rule for Perceptron

- Threshold perceptrons have some advantages , in particular
  - Simple learning algorithm that fits a threshold perceptron to any linearly separable training set.

- Key idea: Learn by adjusting weights to reduce error on training set.
  - update weights repeatedly (epochs) for each example.

- We'll use Sum of squared errors
  - Learning is an optimization search problem in weight space.

A learning rule for Perceptron

- Let S = {(x_i, y_i): i = 1, 2, ..., N} be a training set. (Note, x is a vector of inputs, and y is the vector of the true outputs.)
- Let h_w be the perceptron classifier represented by the weight vector w.
- Definition:

$$E(\mathbf{x}) = Squared\ Error(\mathbf{x}) = \frac{1}{2}(y - h_{\mathbf{w}}(\mathbf{x}))^2$$

A learning rule for Perceptron

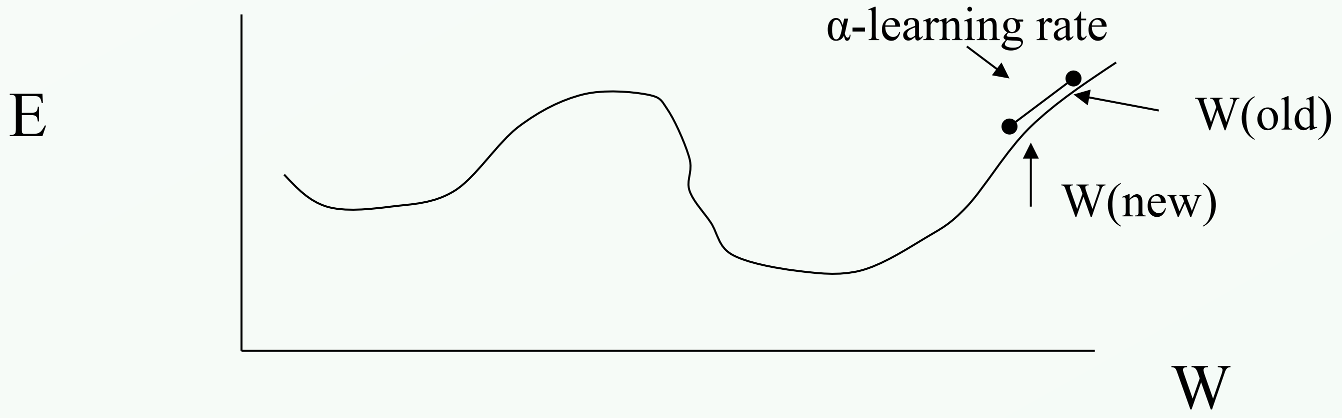
- LMS = Least Mean Square Learning Systems, general perceptron learning rule. The concept is to minimize the total error, as measured over all training examples, N.

- h is the raw output, as calculated by $\sum_j w_j I_j + \theta$

$$Distance(LMS) = \frac{1}{2}\sum_p \left(y_P - h_p\right)^2$$

E.g. if we have two patterns and
y1=1, h1=0.8, y2=0, h2=0.5 then $E=(0.5)[(1-0.8)^2+(0-0.5)^2]=.145$

We want to minimize the LMS:



A learning rule for Perceptron

- The squared error for a single training example with input x and true output y is:

$$E = \frac{1}{2}Err^2 \equiv \frac{1}{2}(y - h_{\mathbf{W}}(\mathbf{x}))^2 ,$$

$$h_w = g(\sum_j w_j I_j + \Theta) = \frac{1}{1 + e^{-\sum_j w_j I_j + \Theta}}$$

- Where $h_w(x)$ is the output of the perceptron on the example and y is the true output value.
- We can use the gradient descent to reduce the squared error by calculating the partial derivatives of E with respect to each weight.

$$\frac{\partial E}{\partial W_j} = Err \times \frac{\partial Err}{\partial W_j} = Err \times \frac{\partial}{\partial W_j}\left(y - g(\Sigma_{j=0}^{n} W_j x_j)\right)$$
$$= -Err \times g'(in) \times x_j$$

  - Note: g’(in) derivative of the activation function. For sigmoid g’=g(1-g).
  - For threshold perceptrons, g’(n) is undefined, the original perceptron rule simply omitted it.

A learning rule for Perceptron

Artificial Inteligence
& Computer Vision
Laboratory

$$\frac{\partial E}{\partial W_j} = -Err \times g'(in) \times x_j$$

- Gradient descent algorithm → we want to reduce , E, for each weight w_i , change weight in direction of steepest descent:

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

α - learning rate

$$W_j \leftarrow W_j + \alpha \times I_j \times Err$$

- Intuitively:
  - $Err = y - h_W(x)$ positive
    → weights are increased for positive inputs and decreased for negative inputs.
  - $Err = y - h_W(x)$ negative
    → opposite

Perceptron Learning: Intuition

Artificial Inteligence & Computer Vision Laboratory

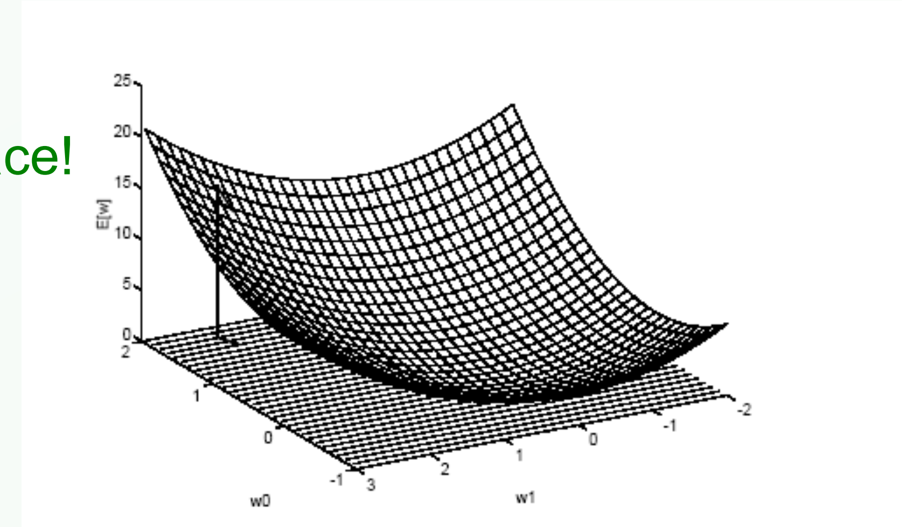
Rule is intuitively correct!

Greedy Search:

Gradient descent through weight space!

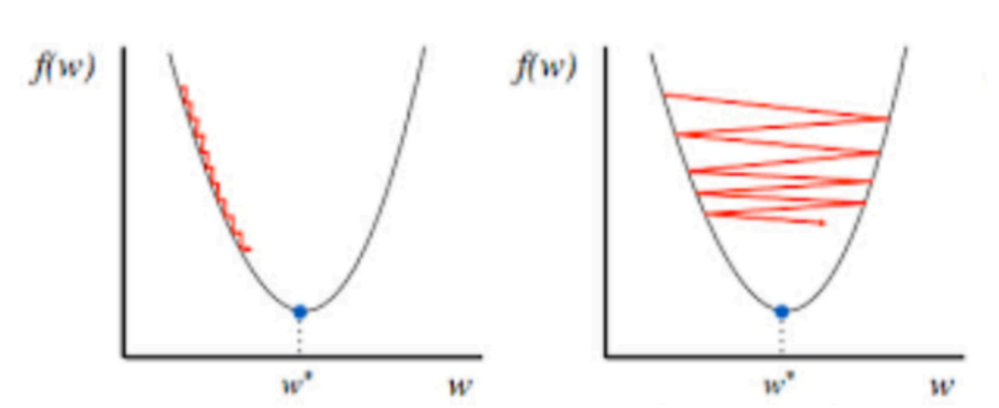
Surprising proof of convergence:

Weight space has no local minima!



With enough examples, it will find the target function!

(provide α not too large)



Pot of the cost as a function of the weights. Left: small learning rate. Right: large learning rate.

$W_j \leftarrow W_j + \alpha \times I_j \times Err$

Perceptron learning rule:

1. Start with random weights, $\mathbf{w} = (w_1, w_2, \ldots, w_n)$.
2. Select a training example $(\mathbf{x}, y) \in S$.
3. Run the perceptron with input $\mathbf{x}$ and weights $\mathbf{w}$ to obtain g
4. Let α be the training rate (a user-set parameter).

$$\forall w_i, \; w_i \leftarrow w_i + \Delta w_i,$$

where

$$\Delta w_i = \alpha (y - g(in)) g'(in) x_i$$

5. Go to 2.

Epoch → cycle through the examples

Epochs are repeated until some stopping criterion is reached—typically, that the weight changes have become very small.

The stochastic gradient method selects examples randomly from the training set rather than cycling through them.

Perceptron Learning: Gradient Descent Learning Algorithm

function PERCEPTRON-LEARNING(*examples*, *network*) **returns** a perceptron hypothesis
  inputs: *examples*, a set of examples, each with input $\mathbf{x} = x_1, \ldots, x_n$ and output y
    network, a perceptron with weights $W_j,\ j = 0 \ldots n$, and activation function g

  repeat
    for each e **in** *examples* **do**
      $in \leftarrow \sum_{j=0}^{n} W_j\, x_j[e]$
      $Err \leftarrow y[e] - g(in)$
      $W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j[e]$
  until some stopping criterion is satisfied
  return NEURAL-NET-HYPOTHESIS(*network*)

Figure 20.21 The gradient descent learning algorithm for perceptrons, assuming a differentiable activation function g. For threshold perceptrons, the factor $g'(in)$ is omitted from the weight update. NEURAL-NET-HYPOTHESIS returns a hypothesis that computes the network output for any given example.

Perceptron Learning (details)

Update the weights by minimizing the errors

$$E_d = \frac{1}{2}\left(y^{(d)} - f^{(d)}\right)^2$$

$$f^{(d)} = f\left(x^{(d)}; w\right) = \sum_i w_i x_i$$

$$w_i \leftarrow w_i + \Delta w_i$$

$$\Delta w_i = -\eta \frac{\partial E_d}{\partial w_i}$$

$\eta \in (0,1)$ 은 학습률 (learning rate)

$$\frac{\partial E_d}{\partial w_i} = \frac{\partial E_d}{\partial f^{(d)}} \frac{\partial f^{(d)}}{\partial w_i} = \frac{\partial}{\partial f^{(d)}} \frac{1}{2}\left(y^{(d)} - f^{(d)}\right)^2 \frac{\partial f^{(d)}}{\partial w_i}$$

$$= \frac{1}{2}(-2)\left(y^{(d)} - f^{(d)}\right)x_i^{(d)} = -\left(y^{(d)} - f^{(d)}\right)x_i^{(d)}$$

$$\boxed{w_i \leftarrow w_i + \eta\left(y^{(d)} - f^{(d)}\right)x_i^{(d)}}$$

Learning of Sigmoid Neuron (details)

- Output of sigmoid unit

$$s^{(d)} = \sum_i w_i x_i^{(d)} \qquad f^{(d)} = f(\mathbf{x}^{(d)}; \mathbf{w}) = \frac{1}{1 + \exp(-s^{(d)})}$$

- Weights

$$\frac{\partial E_d}{\partial w_i} = \frac{\partial E_d}{\partial f^{(d)}} \frac{\partial f^{(d)}}{\partial w_i} = \frac{\partial}{\partial f^{(d)}} \frac{1}{2} \left(y^{(d)} - f^{(d)}\right)^2 \frac{\partial f^{(d)}}{\partial s^{(d)}} \frac{\partial s^{(d)}}{\partial w_i}$$

$$= \frac{1}{2}(-2)\left(y^{(d)} - f^{(d)}\right) f^{(d)} \left(1 - f^{(d)}\right) x_i^{(d)}$$

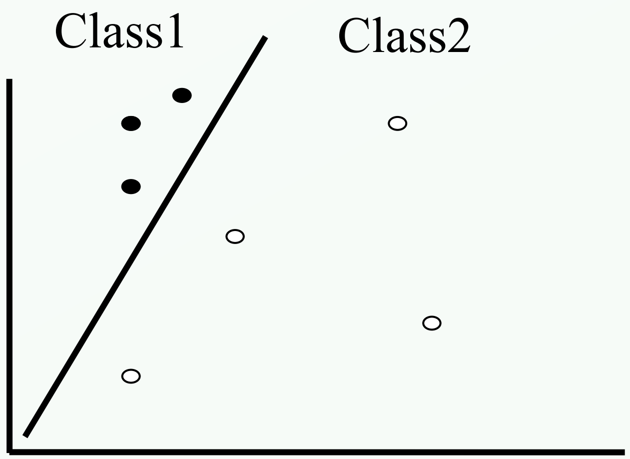
$$= -\left(y^{(d)} - f^{(d)}\right) f^{(d)} \left(1 - f^{(d)}\right) x_i^{(d)}$$

$$\boxed{w_i \leftarrow w_i + \eta\left(y^{(d)} - f^{(d)}\right) \boxed{f^{(d)}\left(1 - f^{(d)}\right)} x_i^{(d)}}$$

$$W_j \leftarrow W_j + \alpha \times I_j \times Err$$

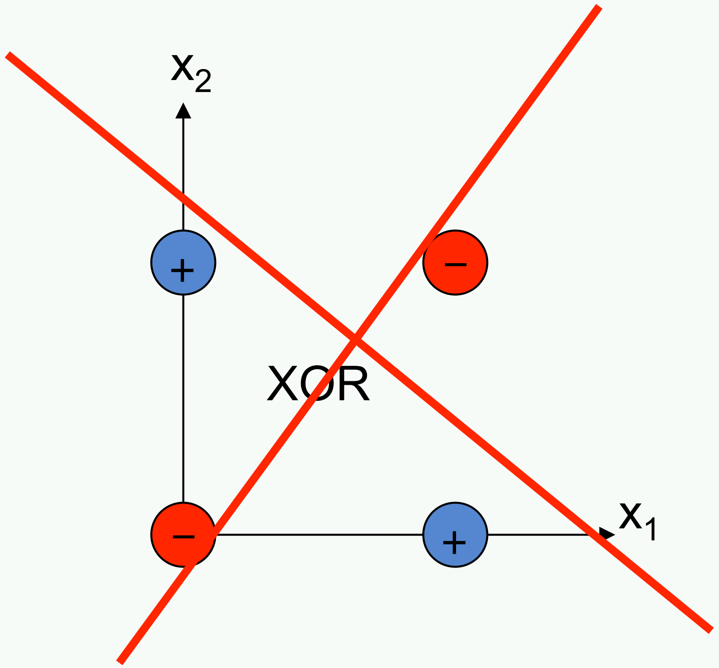
Perceptrons are not powerful

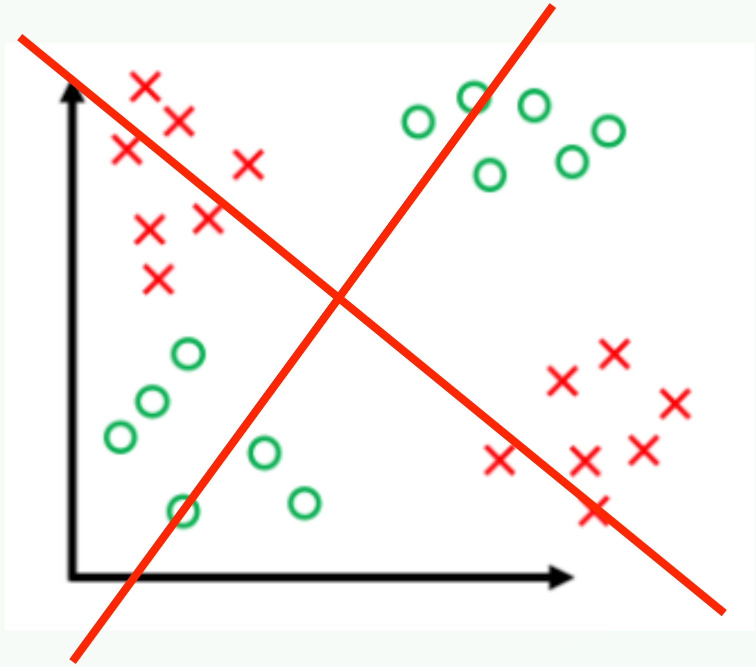
- Essentially a linear discriminant
- Perceptron theorem: If a linear discriminant exists that can separate the cl asses without error, the training procedure is guaranteed to find that line or plane.



Linear Separable Functions

- Minsky & Papert (1969): Perceptrons can only represent linearly separable functi ons



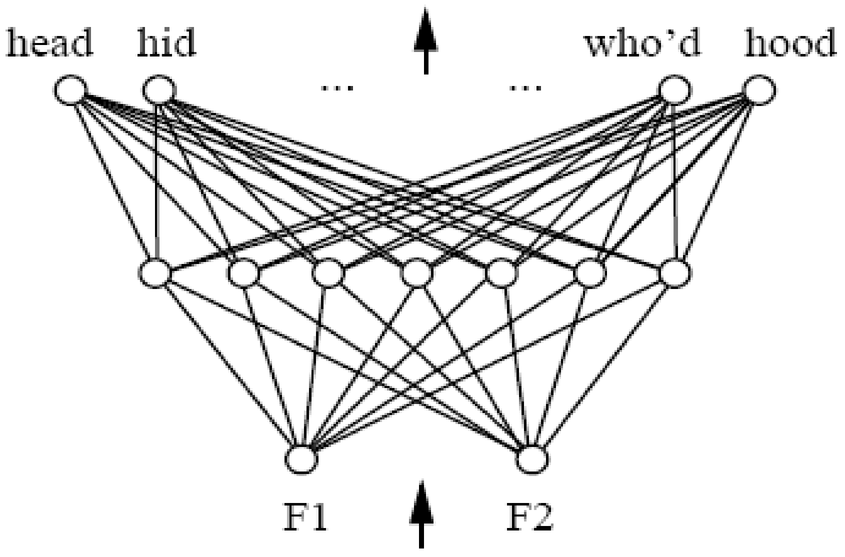


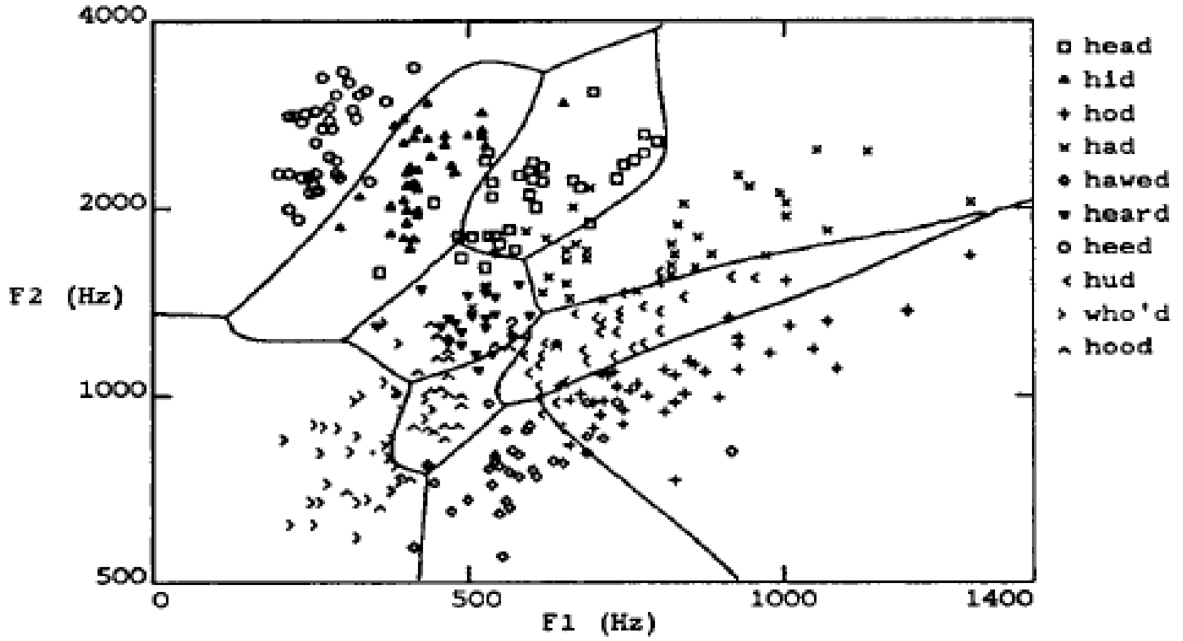
XOR problem

Not linearly separable

Linear Separable Functions

- Minsky & Papert (1969)
  - Perceptrons can only represent linearly separable functions
  - But, adding hidden layer allows more target functions to be represented

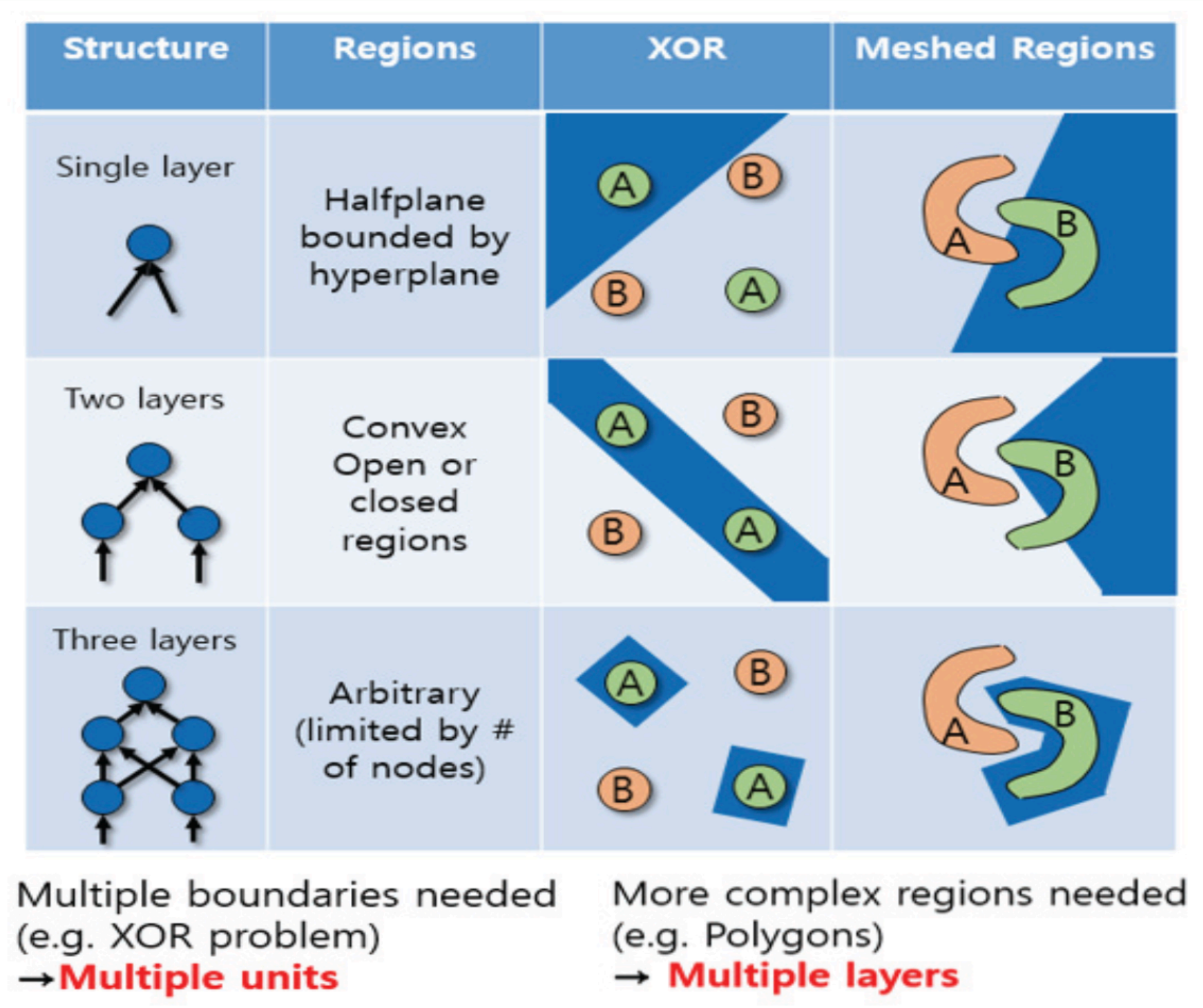




Multilayer Perceptron

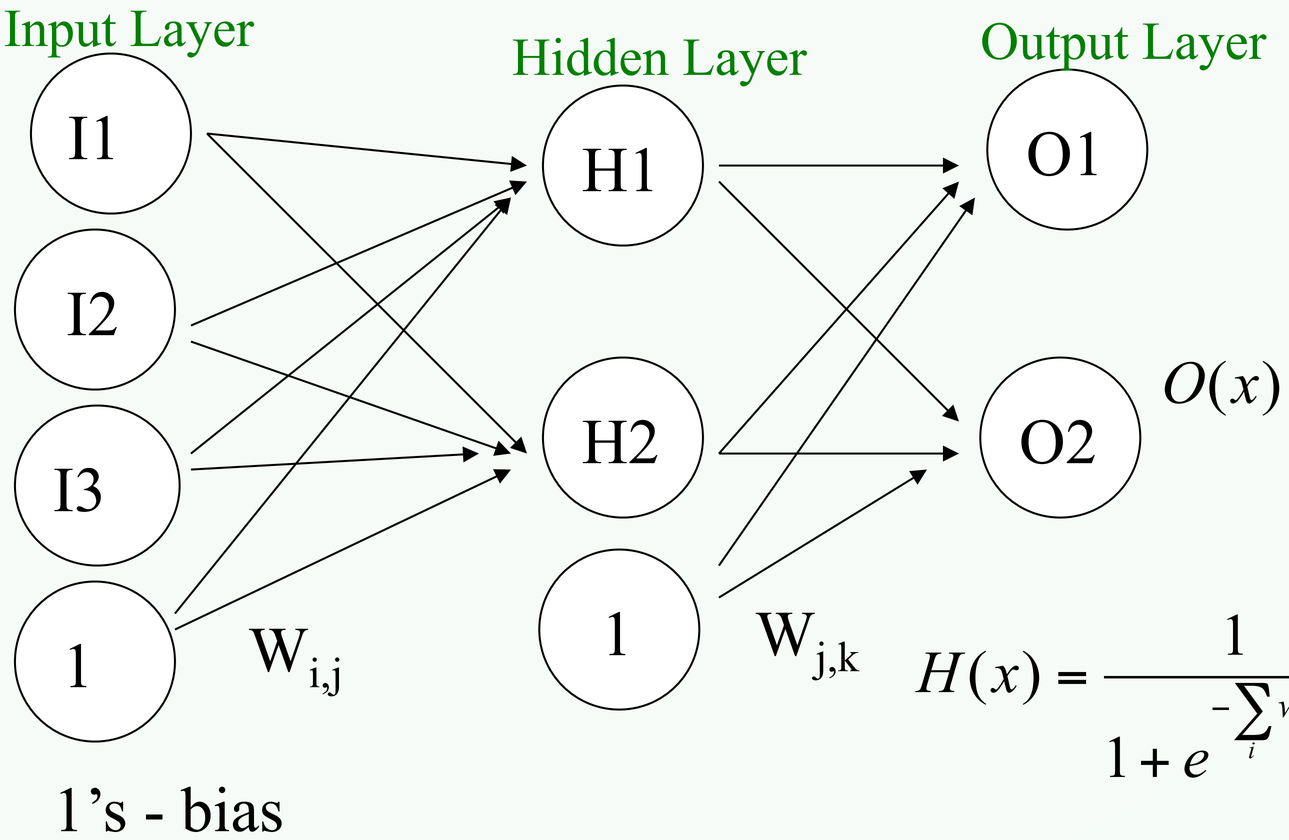
Multi-layer Perceptron

- Why MLP is needed?



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

Artificial Inteligence & Computer Vision Laboratory

- Hidden units are nodes that are situated between the input nodes and the outp ut nodes.
- Hidden units allow a network to learn non-linear functions.
- Hidden units allow the network to represent combinations of the input features.

Hidden Units

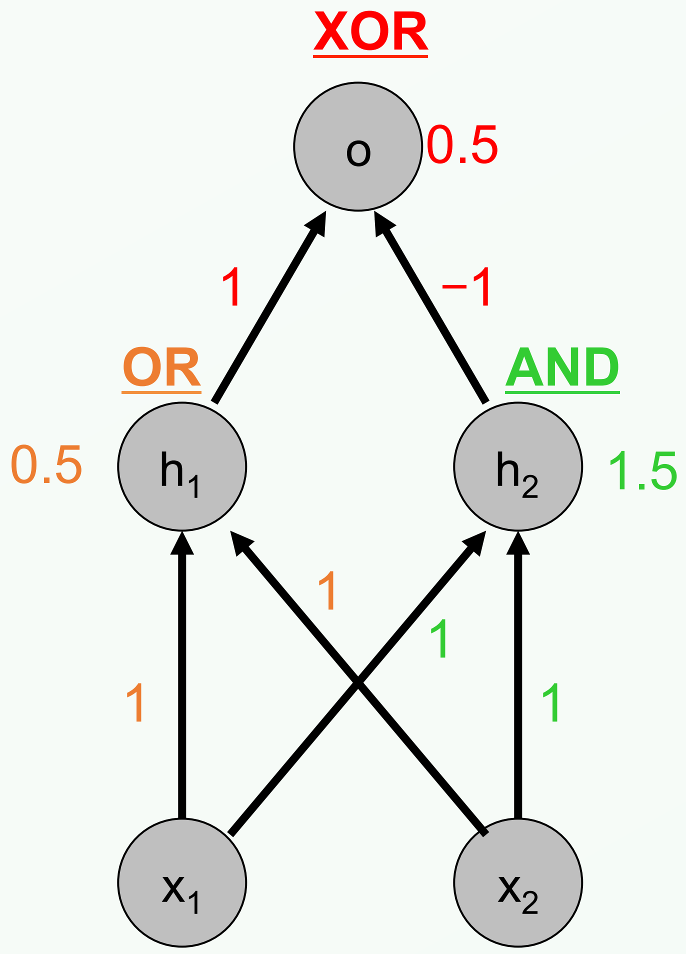
- Boolean XOR

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

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

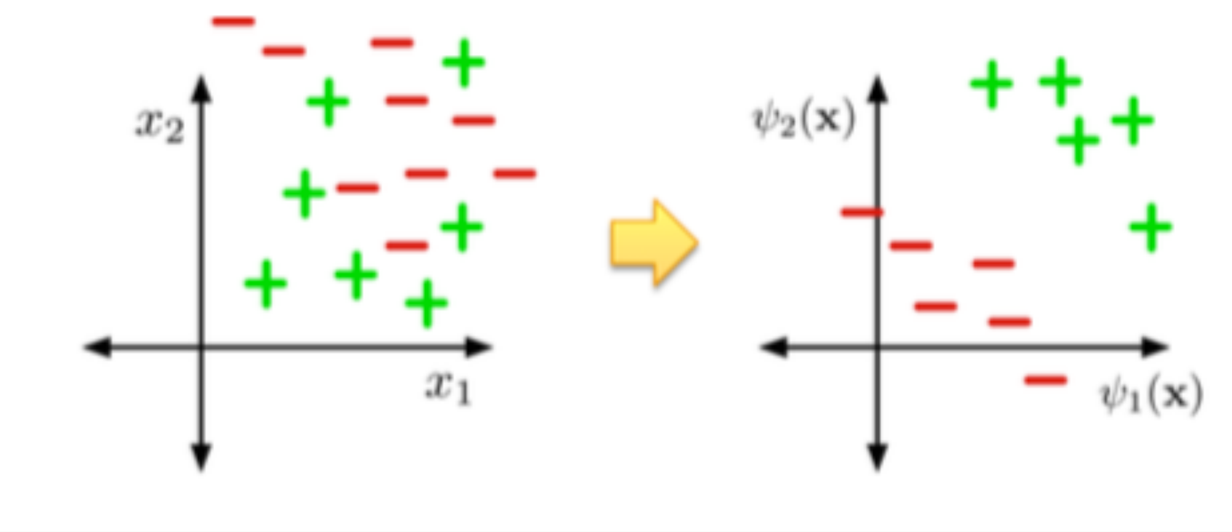
Let's consider a single hidden layer network, using as building blocks threshold units.

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



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

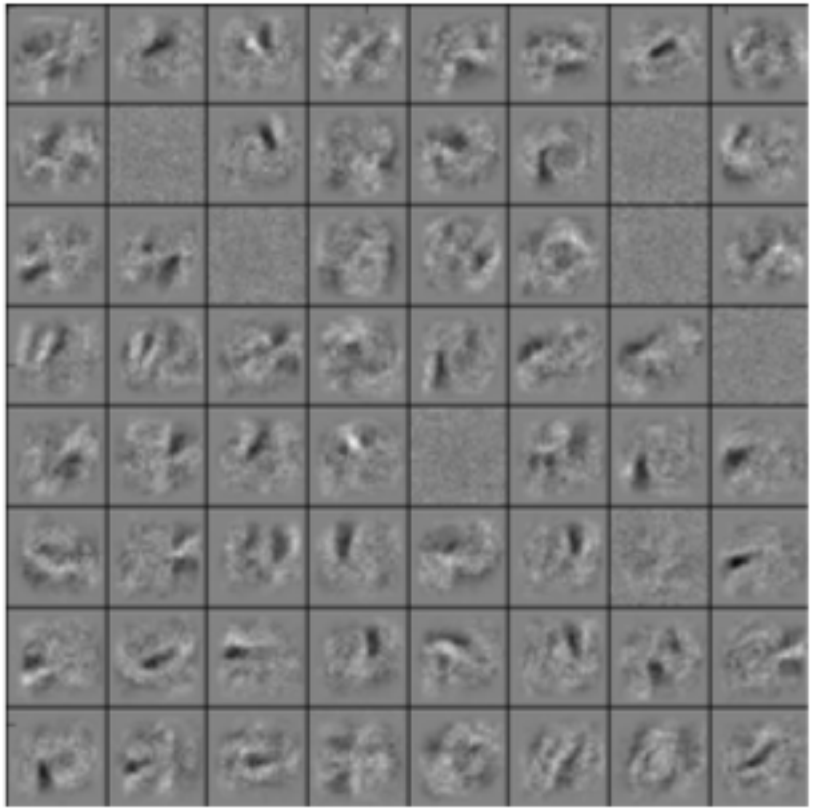


Hidden Units

Artificial Inteligence & Computer Vision Laboratory

- 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

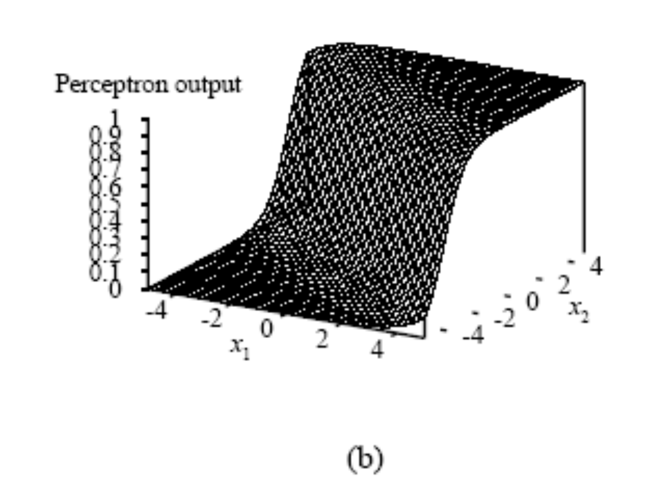
Expressiveness of MLP: Soft Threshold

Artificial Inteligence & Computer Vision Laboratory

- Advantage of adding hidden layers

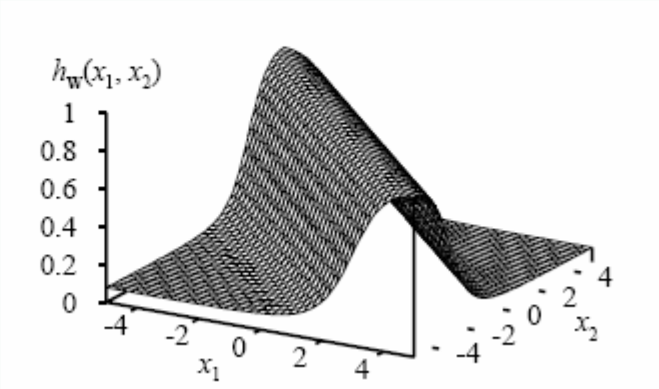
→ It enlarge the space of hypotheses that the network can represent

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.

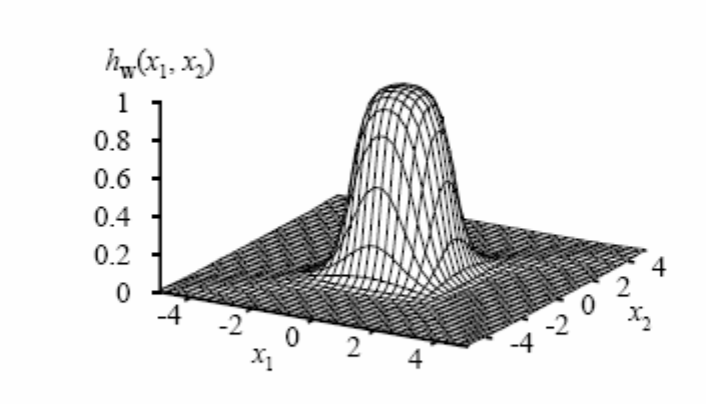


Soft threshold function

Expressiveness of MLP: Soft Threshold



a)



b)

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

(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 an y continuous function of the inputs with arbitrary accuracy;

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

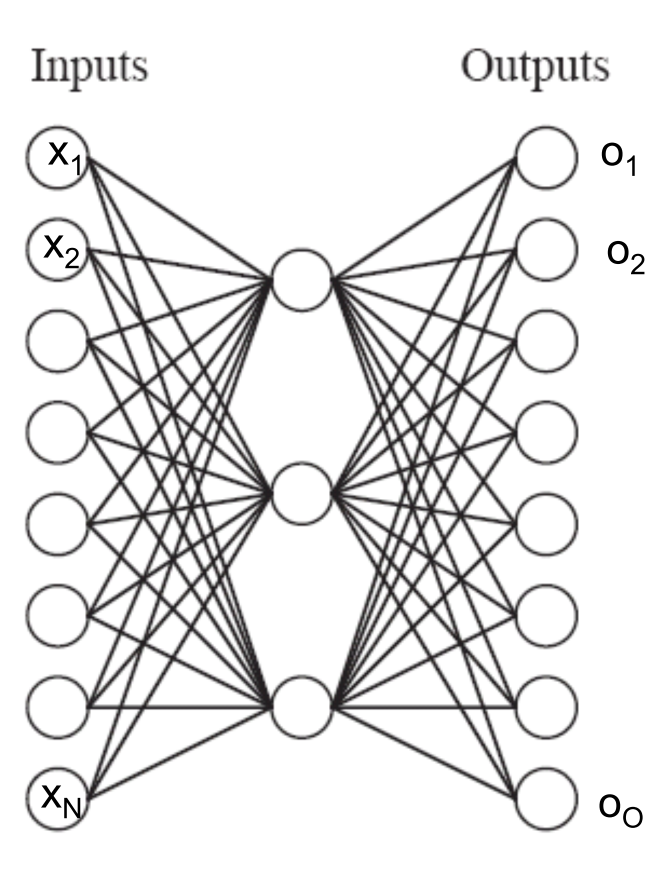
  - The proof is complex → main point, required number of hidden uni ts grows exponentially with the number of inputs.
  - For example, $2^n/n$ hidden units are needed to encode all Boolean f unctions of n inputs.

- 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

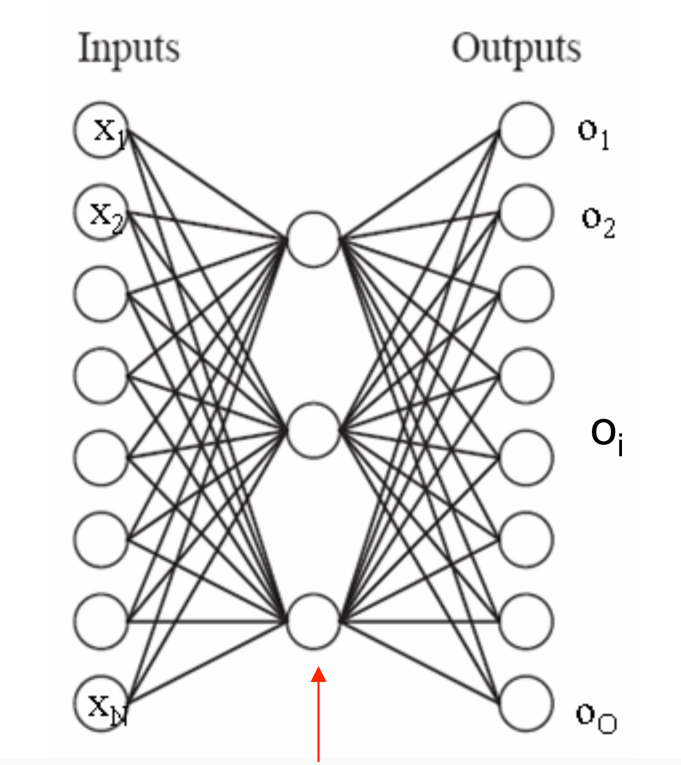
Artificial Inteligence
& Computer Vision
Laboratory

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



How to compute the errors for the hidden units?

$Err_1=y_1-o_1$

$Err_2=y_2-o_2$

$Err_i=y_i-o_i$

$Err_o=y_o-o_o$

Clear error at the output layer

Goal: minimize sum squared errors

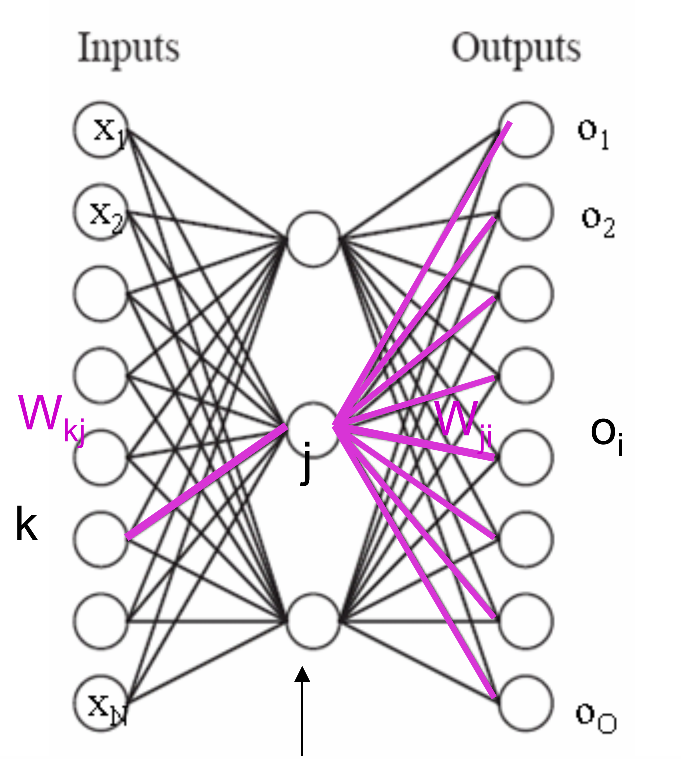
$$E = \frac{1}{2}\sum_i (y_i - o_i)^2$$

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

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)

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

Artificial Inteligence & Computer Vision Laboratory

- 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

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 connects
→ depending on . the strength of the connection
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 Algorithm

function BACK-PROP-LEARNING(*examples*, *network*) **returns** a neural network
  inputs: *examples*, a set of examples, each with input vector $\mathbf{x}$ and output vector $\mathbf{y}$
    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

- **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?)

Learning NN Structures

- Fully connected networks
  - How many layers? How many hidden units?
  - Cross-validation to choose the one with the highest prediction accuracy on the validation sets.
- Not fully connected networks – search for right topology (large space)
  - Optimal - Brain damage : start with a fully connected network; Try removing connections from it.
  - Tiling : algorithm for growing a network starting with a single unit

How long should you train the net?

Artificial Inteligence & Computer Vision Laboratory

- The goal is to achieve a balance between correct responses for the training patterns and correct responses for new patterns. (That is, a balance between memorization and generalization).
- If you train the net for too long, then you run the risk of overfitting.
- Select number of training iterations via cross-validation on a holdout set.

Multi Layer Networks: expressiveness vs. computational complexity

Multi- Layer networks → very expressive!

They can represent general non-linear functions!!!!

But…

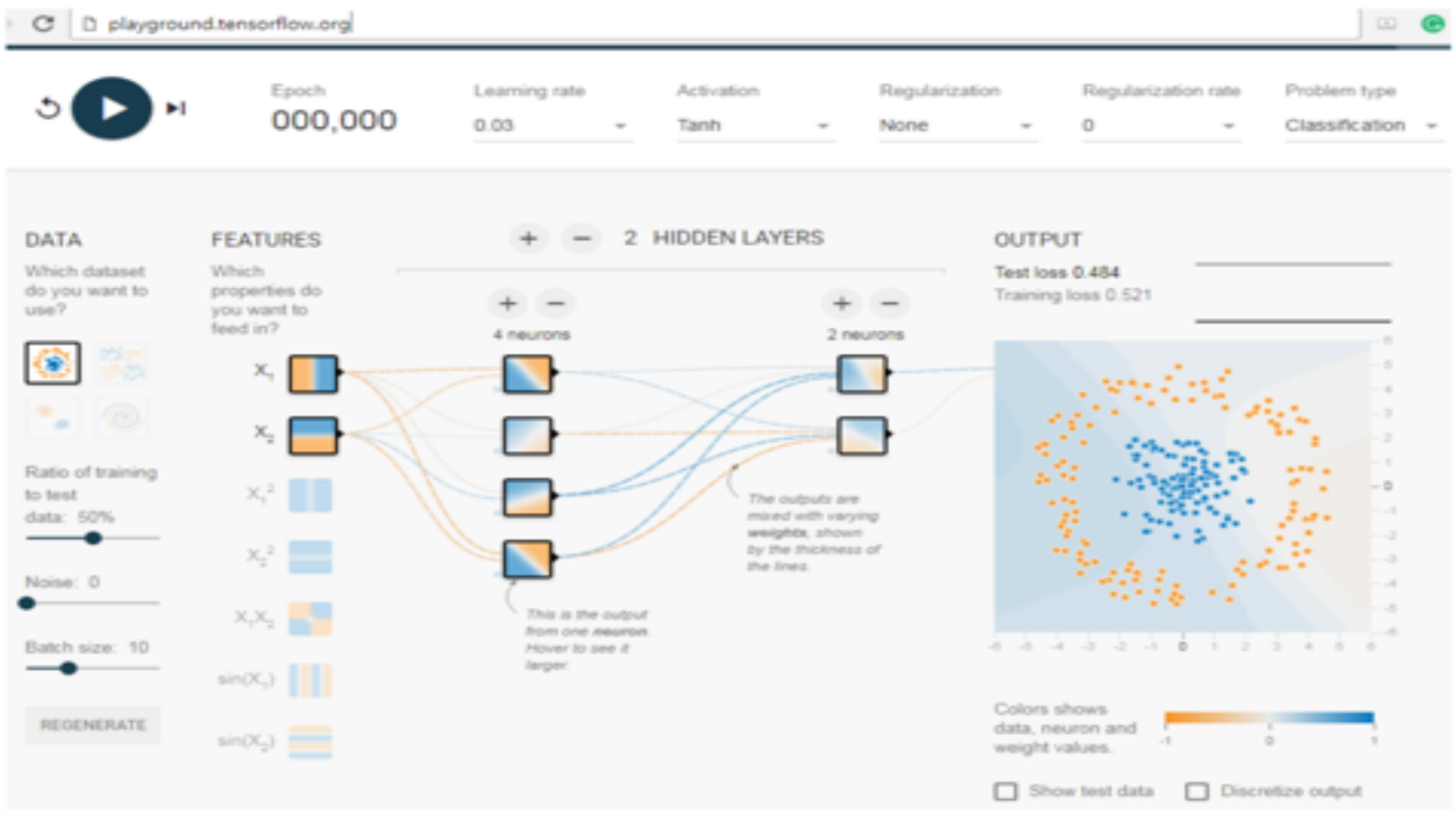
In general they are hard to train due to the abundance of local minima and high dimensionality of search space

Also resulting hypotheses cannot be understood easily

Let’s look at how these work

Artificial Inteligence
& Computer Vision
Laboratory

Playground (playground.tensorflow.org)



Summary

Artificial Inteligence & Computer Vision Laboratory

- Perceptrons (one-layer networks) limited expressive power—they can lea rn 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 hi gh dimensionality of the weight space
- Many applications: speech, driving, handwriting, fraud detection, etc.

From NNs to Convolutional NNs

- Local connectivity
- Shared ("tied") weights
- Multiple feature maps
- Pooling