COMP3411/9414/9814: Artificial Intelligence

Week 7: Neural Networks

Russell & Norvig: 18.6, 18.7

Outline

- Linear Separability
- Multi-Layer Networks
- Backpropagation
- Application - ALVINN
- Training Tips

Artificial Neural Networks

(Artificial) Neural Networks are made up of nodes which have

- inputs edges, each with some weight
- outputs edges (with weights)
- an activation level (a function of the inputs)

Weights can be positive or negative and may change over time (learning).

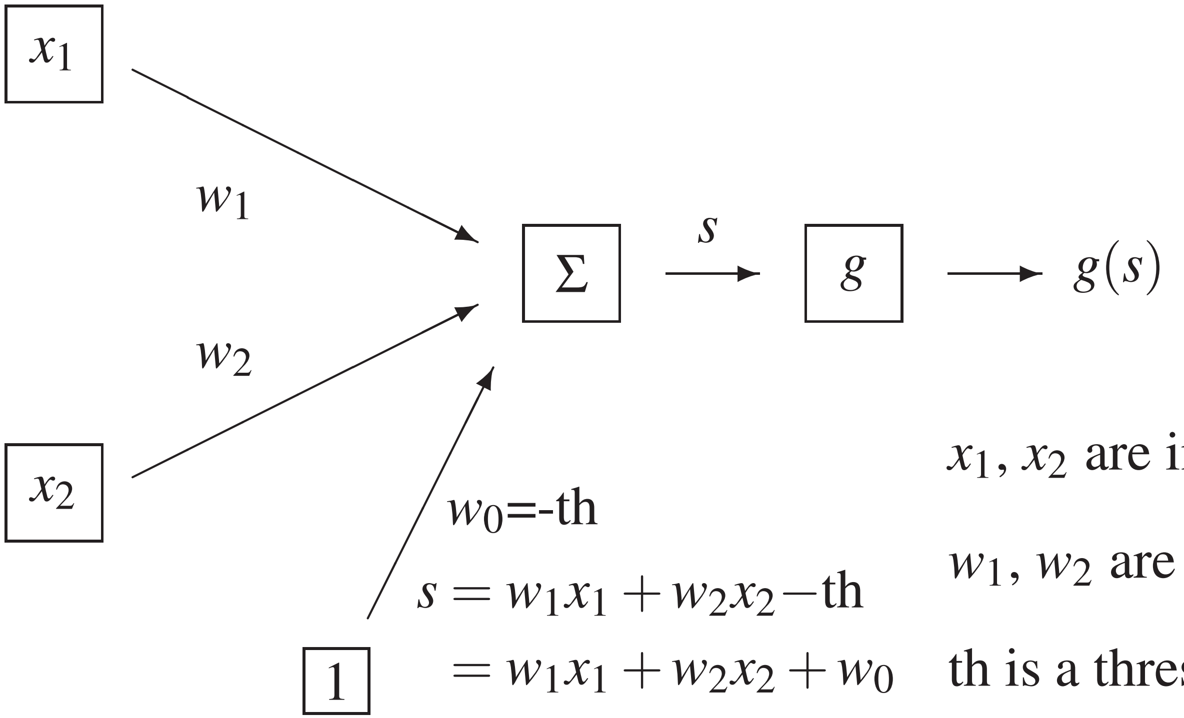
The input function is the weighted sum of the activation levels of inputs.

The activation level is a non-linear transfer function g of this input:

$$\text{activation}_i = g(s_i) = g(\sum_j w_{ij} x_j)$$

Some nodes are inputs (sensing), some are outputs (action)

McCulloch & Pitts Model of a Single Neuron



$$s = w_1 x_1 + w_2 x_2 - \mathrm{th}$$
$$= w_1 x_1 + w_2 x_2 + w_0$$

x_1, x_2 are inputs

w_1, w_2 are synaptic weights

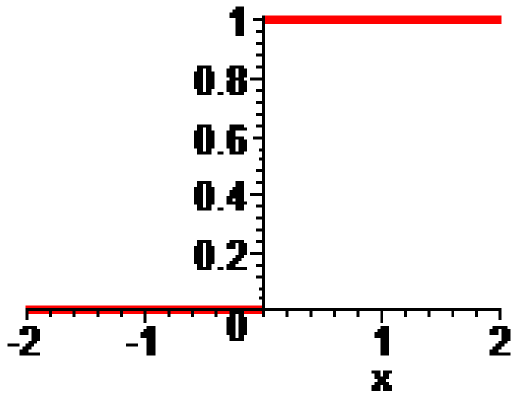
th is a threshold

w_0 is a **bias** weight

g is transfer function

Transfer function

Originally, a (discontinuous) step function was used for the transfer function:



$$g(s) = \begin{cases} 1, & \text{if} \quad s \geq 0 \\ 0, & \text{if} \quad s < 0 \end{cases}$$

Linear Separability

Q: what kind of functions can a perceptron compute?

A: linearly separable functions

Examples include:

AND	$w_1 = w_2 = 1.0,$	$w_0 = -1.5$
OR	$w_1 = w_2 = 1.0,$	$w_0 = -0.5$
NOR	$w_1 = w_2 = -1.0,$	$w_0 = 0.5$

Q: How can we train it to learn a new function?

Perceptron Learning Rule

Adjust the weights as each input is presented.

recall: $s = w_1 x_1 + w_2 x_2 + w_0$

if $g(s) = 0$ but should be 1,

$$\begin{aligned} w_k &\leftarrow w_k + \eta\, x_k \\ w_0 &\leftarrow w_0 + \eta \\ \text{so} \quad s &\leftarrow s + \eta\,(1 + \sum_k x_k^2) \end{aligned}$$

if $g(s) = 1$ but should be 0,

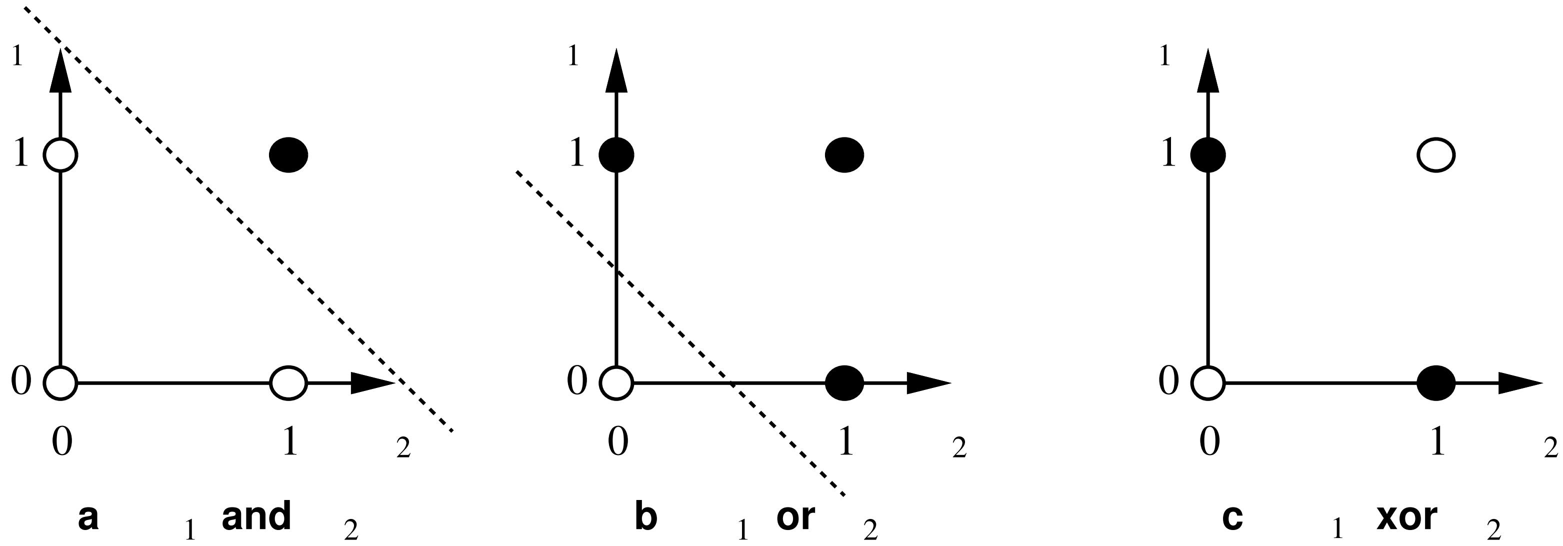
$$\begin{aligned} w_k &\leftarrow w_k - \eta\, x_k \\ w_0 &\leftarrow w_0 - \eta \\ \text{so} \quad s &\leftarrow s - \eta\,(1 + \sum_k x_k^2) \end{aligned}$$

otherwise, weights are unchanged. ($\eta > 0$ is called the **learning rate**)

Theorem: This will eventually learn to classify the data correctly, as long as they are **linearly separable**.

Limitations of Perceptrons

Problem: many useful functions are not linearly separable (e.g. XOR)

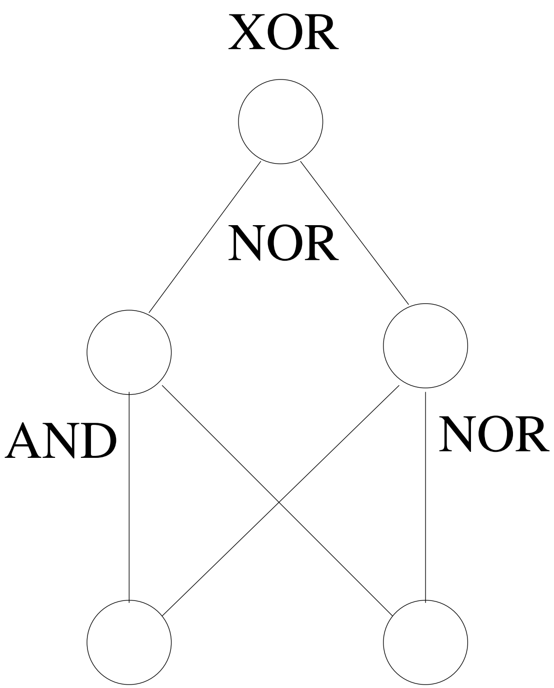


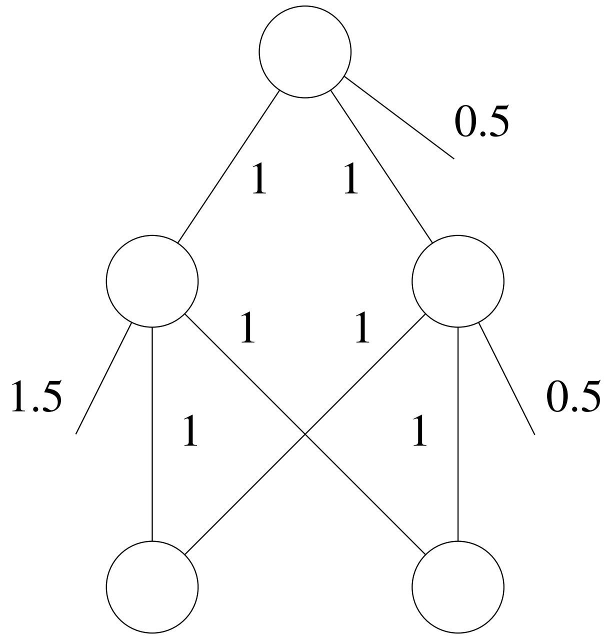
Possible solution:

x_1 XOR x_2 can be written as: (x_1 AND x_2) NOR (x_1 NOR x_2)

Recall that AND, OR and NOR can be implemented by perceptrons.

Multi-Layer Neural Networks





Given an explicit logical function, we can design a multi-layer neural network by hand to compute that function. But, if we are just given a set of training data, can we train a multi-layer network to fit these data?

Historical Context

In 1969, Minsky and Papert published a book highlighting the limitations of Perceptrons, and lobbied various funding agencies to redirect funding away from neural network research, preferring instead logic-based methods such as expert systems.

It was known as far back as the 1960's that any given logical function could be implemented in a 2-layer neural network with step function activations. But, the the question of how to learn the weights of a multi-layer neural network based on training examples remained an open problem. The solution, which we describe in the next section, was found in 1976 by Paul Werbos, but did not become widely known until it was rediscovered in 1986 by Rumelhart, Hinton and Williams.

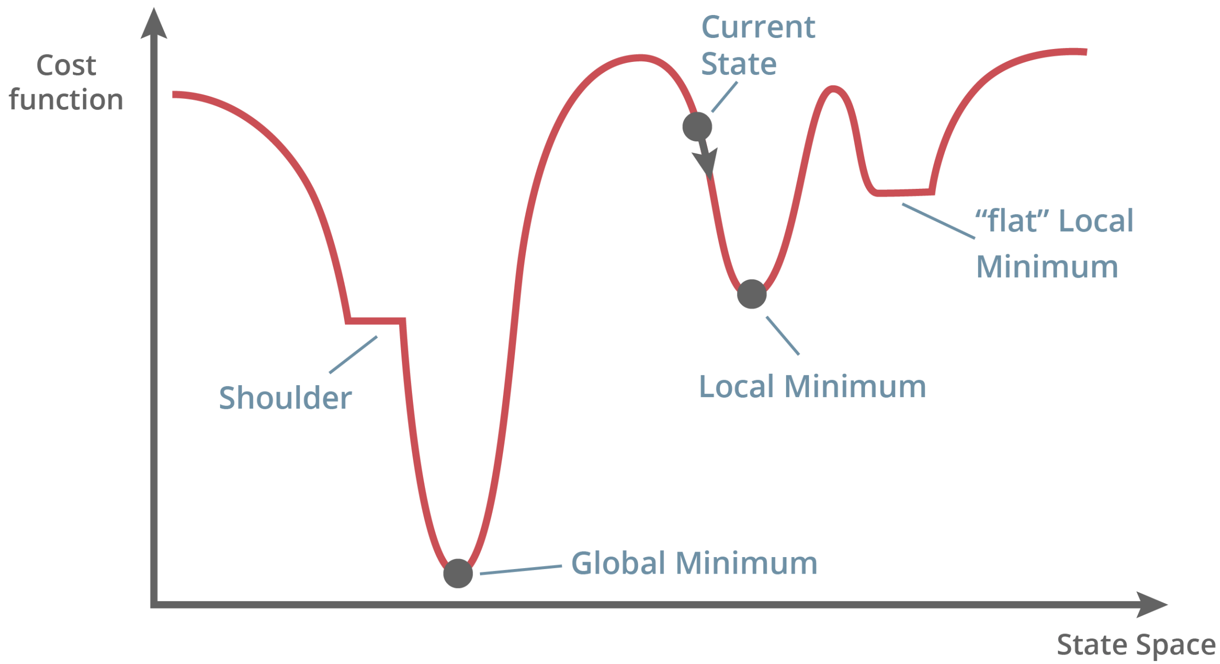
NN Training as Cost Minimization

We define an **error function** E to be (half) the sum over all input patterns of the square of the difference between actual output and desired output

$$E = \frac{1}{2}\sum(z-t)^2$$

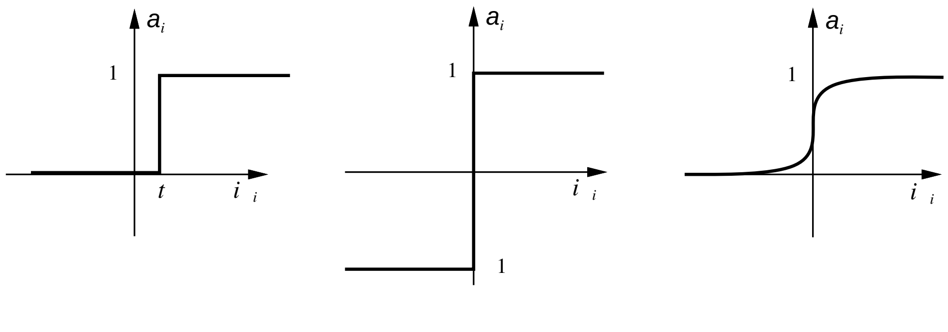
If we think of E as height, it defines an error **landscape** on the weight space. The aim is to find a set of weights for which E is very low.

Local Search in Weight Space

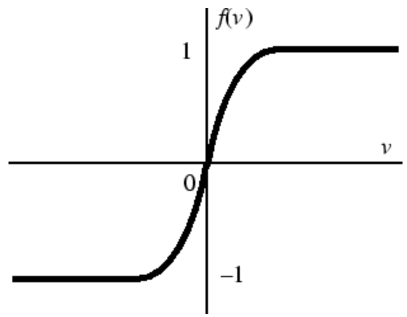


Problem: because of the step function, the landscape will not be smooth but will instead consist almost entirely of flat local regions and "shoulders", with occasional discontinuous jumps.

Key Idea



a Step function b Sign function c Sigmoid function



Replace the (discontinuous) step function with a differentiable function, such as the sigmoid:

$$g(s) = \frac{1}{1+e^{-s}}$$

or hyperbolic tangent

$$g(s) = \tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}} = 2\Big(\frac{1}{1+e^{-2s}}\Big) - 1$$

Gradient Descent

Recall that the **error function** E is (half) the sum over all input patterns of the square of the difference between actual output and desired output

$$E = \frac{1}{2}\sum (z - t)^2$$

The aim is to find a set of weights for which E is very low.

If the functions involved are smooth, we can use multi-variable calculus to adjust the weights in such a way as to take us in the steepest downhill direction.

$$w \leftarrow w - \eta \frac{\partial E}{\partial w}$$

Parameter η is called the learning rate.

Chain Rule

If, say

$$y = y(u)$$

$$u = u(x)$$

Then

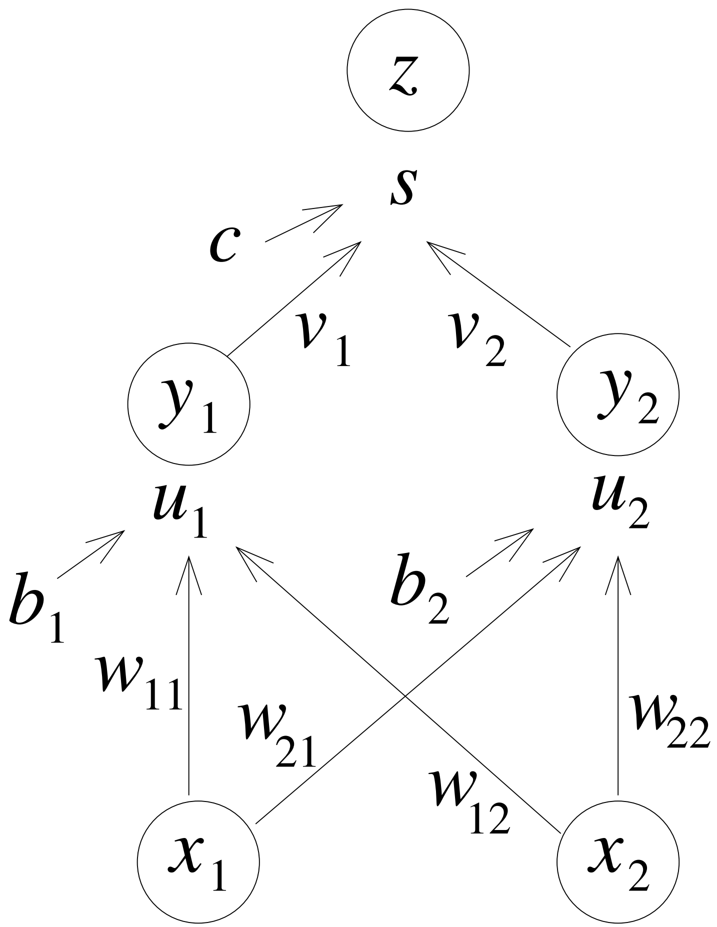
$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\,\frac{\partial u}{\partial x}$$

This principle can be used to compute the partial derivatives in an efficient and localized manner. Note that the transfer function must be differentiable (usually sigmoid, or tanh).

$$\text{Note: if} \quad z(s) = \frac{1}{1+e^{-s}}, \qquad z'(s) = z(1-z).$$

$$\text{if} \quad z(s) = \tanh(s), \qquad z'(s) = 1 - z^2.$$

Forward Pass



$$
\begin{aligned}
u_1 &= b_1 + w_{11}x_1 + w_{12}x_2 \\
y_1 &= g(u_1) \\
s &= c + v_1 y_1 + v_2 y_2 \\
z &= g(s) \\
E &= \frac{1}{2}\sum (z-t)^2
\end{aligned}
$$

Backpropagation

Partial Derivatives

$$
\begin{aligned}
\frac{\partial E}{\partial z} &= z - t \\
\frac{dz}{ds} &= g'(s) = z(1-z) \\
\frac{\partial s}{\partial y_1} &= v_1 \\
\frac{dy_1}{du_1} &= y_1(1-y_1)
\end{aligned}
$$

Useful notation

$$
\delta_{\text{out}} = \frac{\partial E}{\partial s} \quad \delta_1 = \frac{\partial E}{\partial u_1} \quad \delta_2 = \frac{\partial E}{\partial u_2}
$$

Then

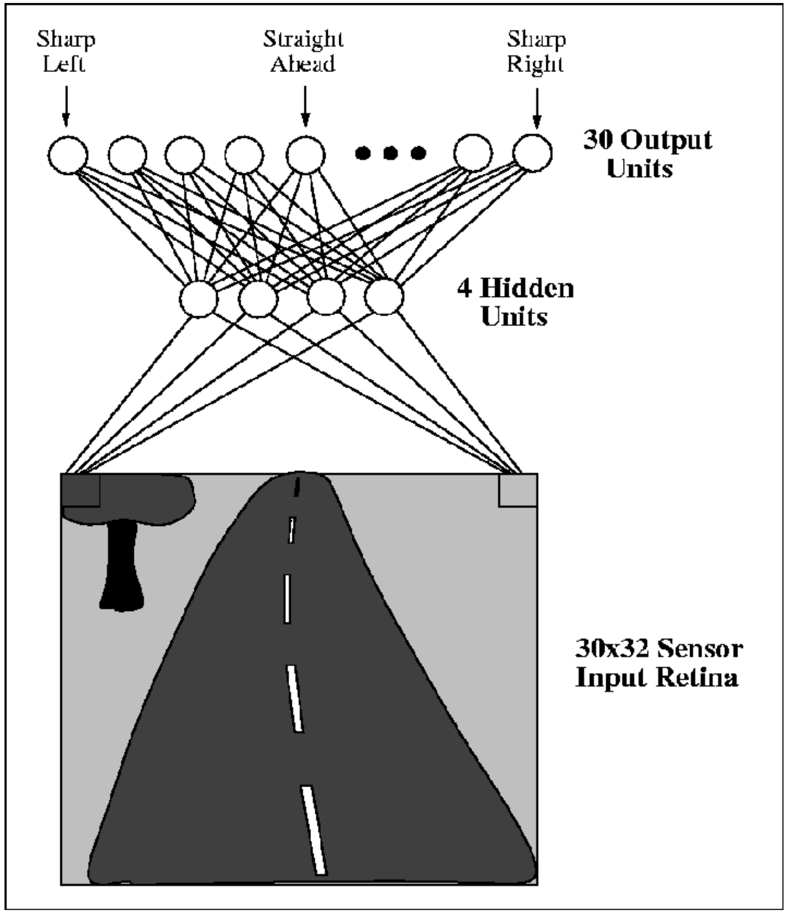
$$
\begin{aligned}
\delta_{\text{out}} &= (z-t)\, z\, (1-z) \\
\frac{\partial E}{\partial v_1} &= \delta_{\text{out}}\, y_1 \\
\delta_1 &= \delta_{\text{out}}\, v_1\, y_1\, (1-y_1) \\
\frac{\partial E}{\partial w_{11}} &= \delta_1\, x_1
\end{aligned}
$$

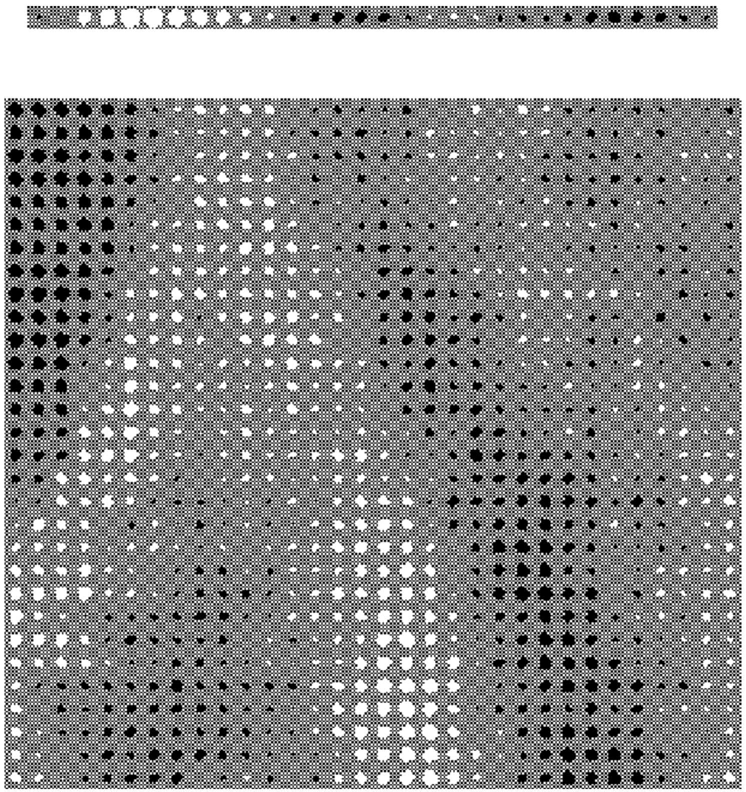
Partial derivatives can be calculated efficiently by packpropagating deltas through the network.

Neural Network – Applications

- Autonomous Driving
- Game Playing
- Credit Card Fraud Detection
- Handwriting Recognition
- Financial Prediction

ALVINN


Sharp Left
Straight Ahead
Sharp Right
30 Output Units
4 Hidden Units
30x32 Sensor Input Retina



ALVINN

- Autonomous Land Vehicle In a Neural Network
- later version included a sonar range finder
  - 8×32 range finder input retina
  - 29 hidden units
  - 45 output units
- Supervised Learning, from human actions (Behavioral Cloning)
  - additional "transformed" training items to cover emergency situations
- drove autonomously from coast to coast

Training Tips

- re-scale inputs and outputs to be in the range 0 to 1 or -1 to 1
- initialize weights to very small random values
- on-line or batch learning
- three different ways to prevent overfitting:
  - limit the number of hidden nodes or connections
  - limit the training time, using a validation set
  - weight decay
- adjust learning rate (and momentum) to suit the particular task