COMP 652: Machine Learning

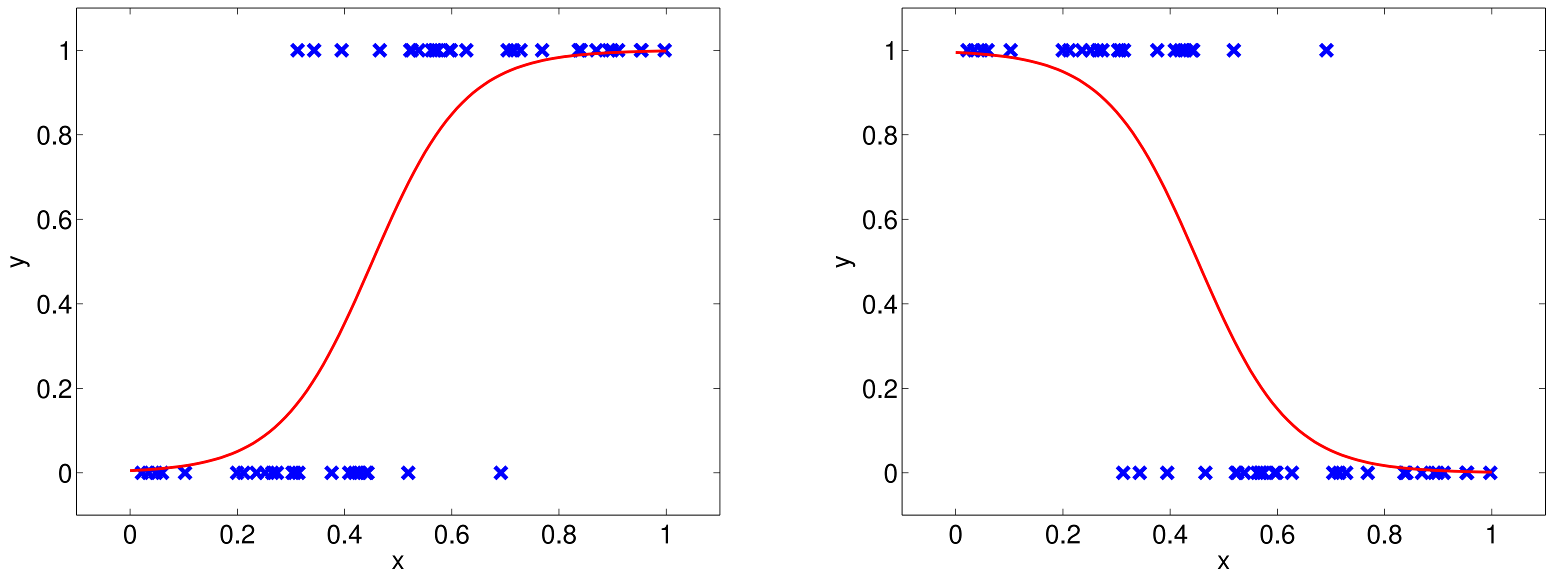
Lecture 5

Today

- Network architecture
- Backpropagation algorithm
- Tweaks:
  - Avoiding local minima
  - Choosing learning rates
  - Encoding the inputs and outputs

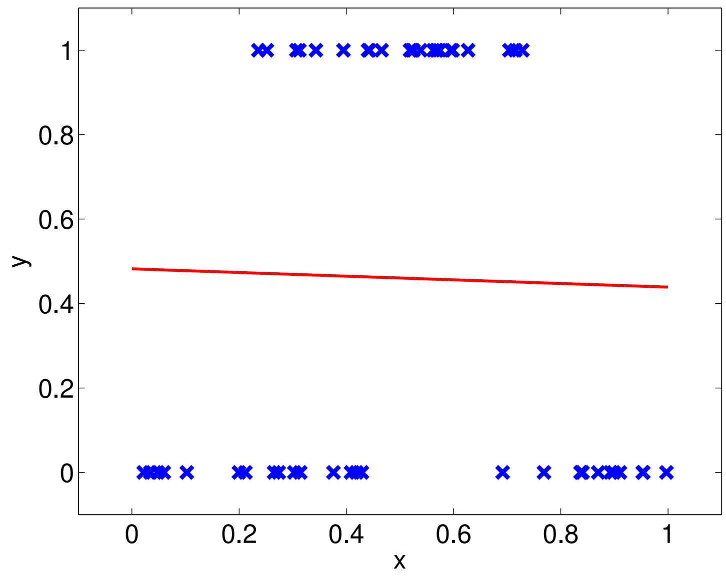
Monotonicity of logistic output

- The output of a logistic regressor or classifier is monotone in each input variable.
- This limits the range of functions that can be captured.



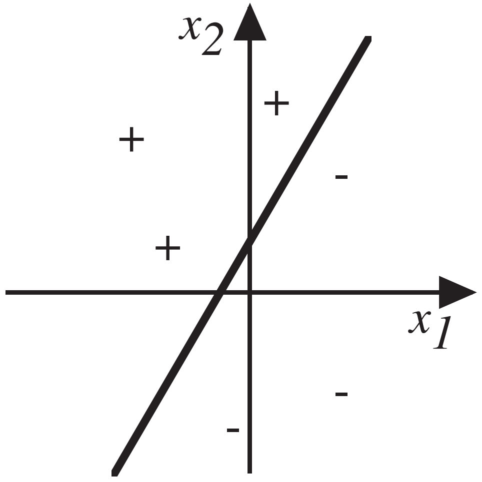
Monotonicity of logistic output

- The output of a logistic regressor or classifier is monotone in each input variable.
- This limits the range of functions that can be captured.

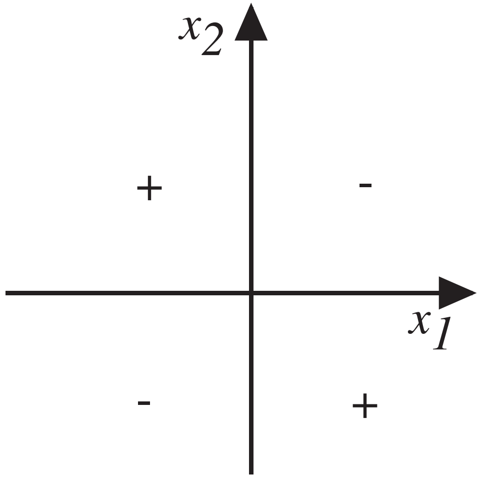


Monotonicity of logistic output

- The output of a logistic regressor or classifier is monotone in each input variable.
- This limits the range of functions that can be captured.

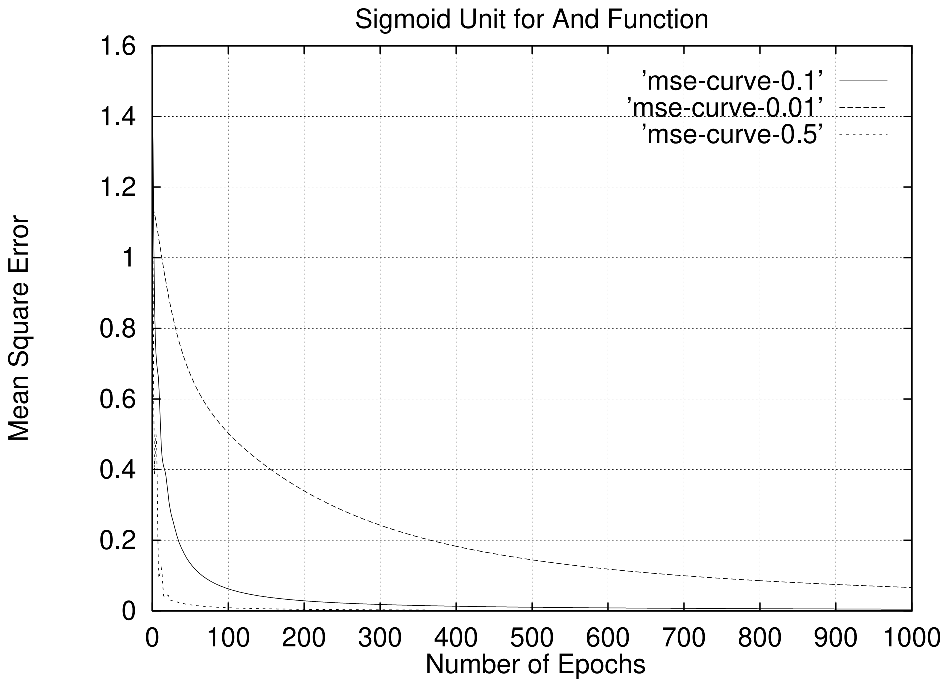


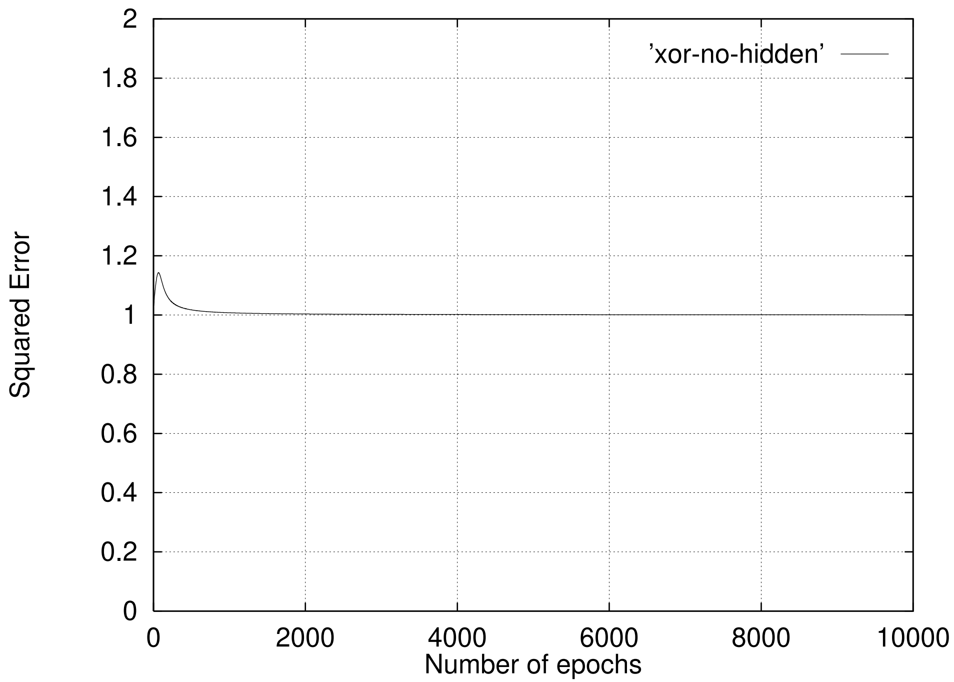
(a)



(b)

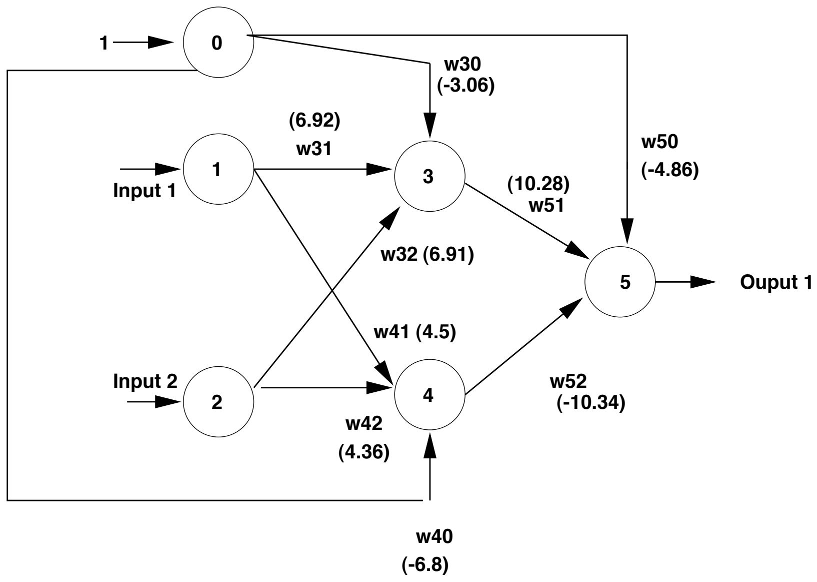
Example: Logical functions of two variables

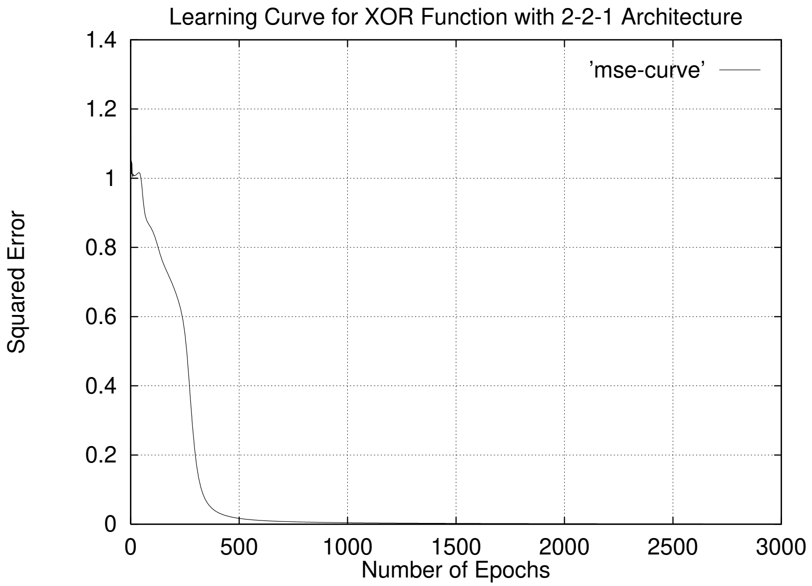




- One sigmoid neuron (a.k.a. logistic) can learn the AND function (left) but not the XOR function (right)
- In order to learn discrimination in data sets that are not linearly separable, we need *networks of sigmoid units*

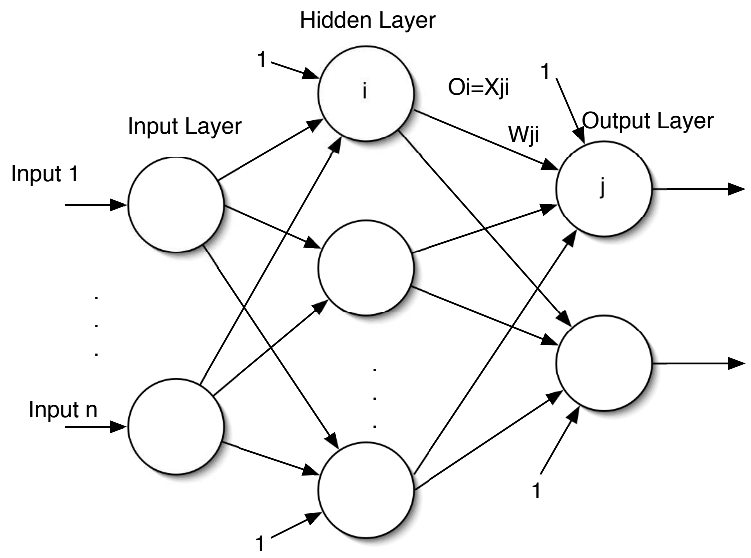
Example: A network representing the XOR function





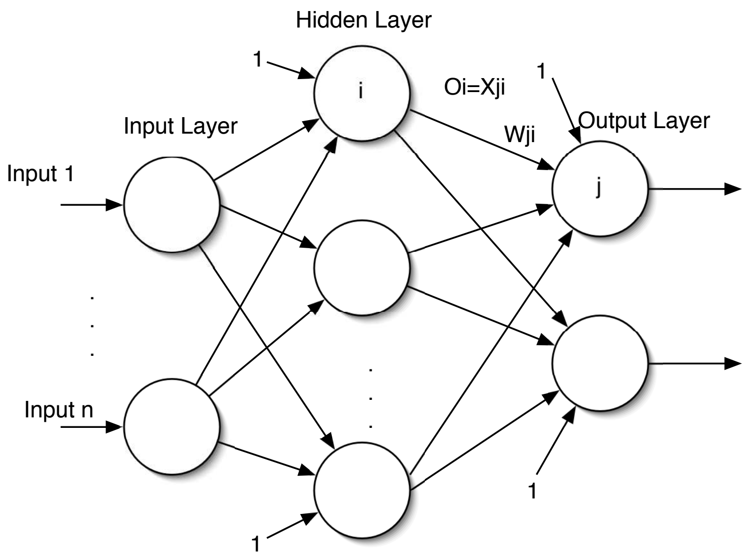
O1 (Input 1)	O2 (Input 2)	O3	O4	O5 (Output)
0	0	0.0448	0.0011	0.0120
0	1	0.9792	0.0802	0.9876
1	0	0.9794	0.0911	0.9862
1	1	1.0000	0.8870	0.0229

Feed-forward neural networks



- A collection of units (neurons) with sigmoid or sigmoid-like activations, arranged in **layers**
- Layers 0 is the **input layer**, and its units just copy the inputs (by convention)

Feed-forward neural networks



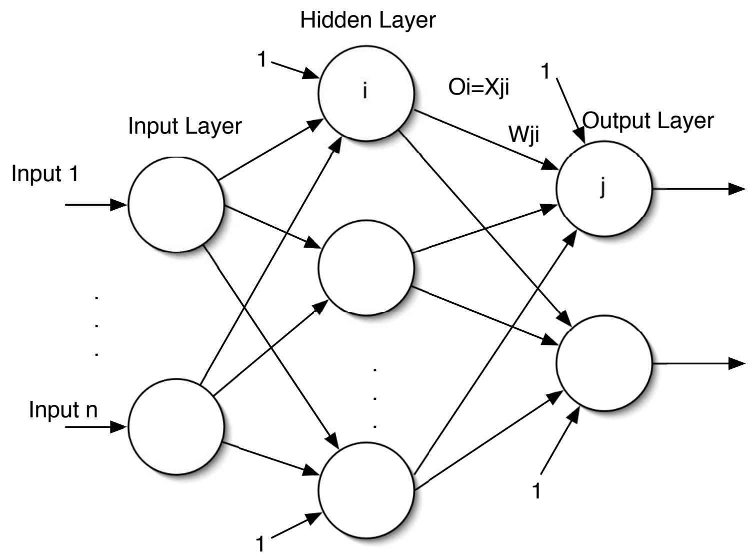
- The last layer, K, is called **output layer**, since its units provide the result of the network
- Layers $1, \dots K-1$ are usually called **hidden layers** (their presence cannot be detected from outside the network)

Why this name?

In **feed-forward networks** the outputs of units in layer k become inputs for units in layers with index $> k$

- There are no cross-connections between units in the same layer
- There are no backward ("recurrent") connections from layers downstream
- Typically, units in layer k provide input *only* to units in layer $k + 1$
- In **fully connected networks**, all units in layer k are connected to all units in layer $k + 1$

Notation



- o_j is the output of unit j
- $w_{j,i}$ is the weight on the connection from unit i to unit j

Output values

- Input unit i just repeats the i^{th} input to the network. That is, on input vector $\mathbf{x}$, $o_i = x_i$.
- For all other units j, o_j is computed using a sigmoid:

$$o_j = \sigma(\mathbf{w}_j^T \mathbf{x}_j)$$

  where $\mathbf{w}_j$ is the vector of weights on the connections *entering* unit j, and and $\mathbf{x}_j$ is the vector of inputs to unit j.

  – Note! $\mathbf{x}_j$ is not the input vector to the whole newtork, $\mathbf{x}$! Rather, $x_{j,i} = o_i$, where o_i is the i^{th} input to unit j.)
  – By convention, $x_{j,0} = 1, \forall j$

Computing the output of the network

- Suppose that we want the network to make a prediction for instance $\langle \mathbf{x}, y \rangle$
- In a feed-forward network, this can be done in one **forward pass**:
  For layer $k = 1$ to K

  1. Compute the output of all neurons in layer k:

  $$o_j \leftarrow \sigma(\mathbf{w}_j^T \mathbf{x}_j), \forall j \in \text{Layer } k$$

  2. Copy these outputs as inputs to the next layer:

  $$x_{j,i} \leftarrow o_i, \forall i \in \text{Layer } k, \forall j \in \text{Layer } k+1$$

Expressiveness of feed-forward neural networks

- A single sigmoid neuron has the same representational power as a perceptron: Boolean AND, OR, NOT, but not XOR
- Every Boolean function can be represented by a network with a single hidden layer, but might require a number of hidden units that is exponential in the number of inputs
- Every bounded continuous function can be approximated with arbitrary precision by a network with one, sufficiently large hidden layer [Cybenko 1989; Hornik et al. 1989]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

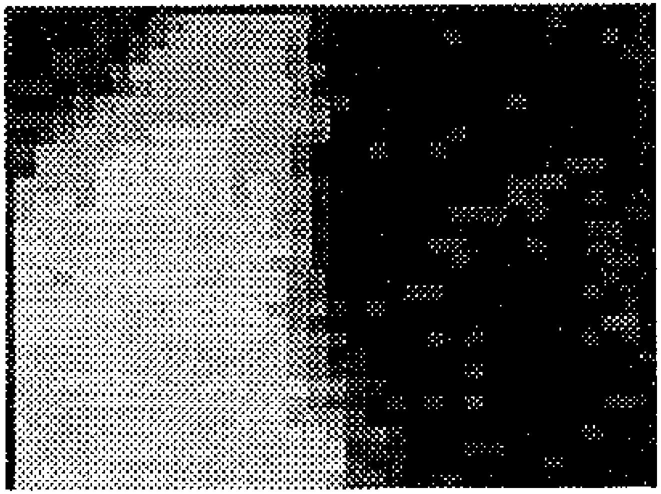
- Historical aside: Minsky & Papert in *Perceptrons* 1969, pointed out the XOR and other weaknesses of logistic-like models, practically killing off ANN research for a decade.

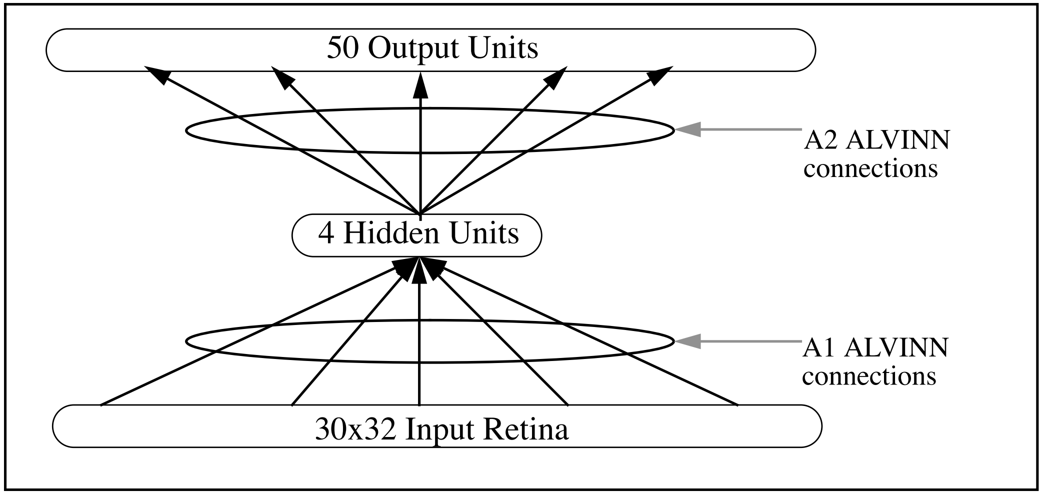
Tons of successful applications of neural networks

Financial forecasting, Fraud detection, Market segmentation, Credit ratings, Real estate evaluation, Human resources, Plant modeling and control, Electrical load balancing, Robotics, Automotive device control, Medical diagnosis, Bioinformatics, Drug interaction prediction, Survival probability estimation, Handwriting recognition, Lipreading, Control of Game AIs, ...

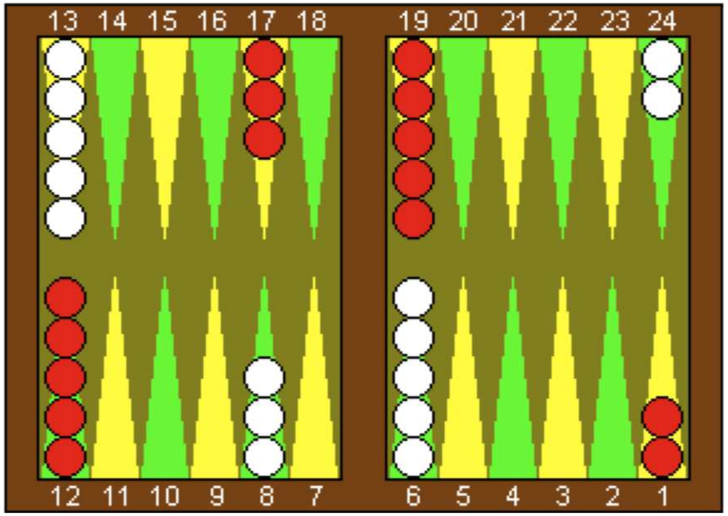
e.g., ALVINN (starting with Pomerleau (1991))

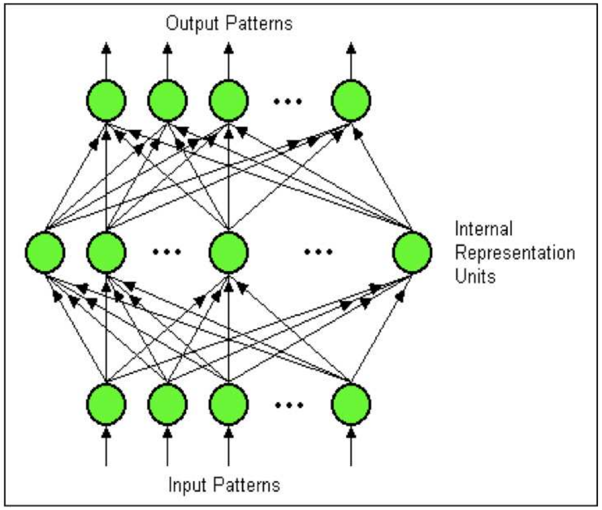
- The ALVINN system was trained to follow roads, and drove a Humvee (at first, on paths around CMU, and in later versions, cross-country with little assistance).
- Inputs were coarse grayscale images; outputs were desired driving direction.





- TD-gammon was trained to play backgammon, by playing simulated games against itself.
- Inputs were raw board position (# black and # white checkers at each position), plus features defined by human experts (e.g. "blockade strength" or "probability of being hit").
- Four outputs: the probabilities of each player winning or gammoning.
- Achieved world-class play, and changed human strategies in certain positions.





Learning in feed-forward neural networks

- Usually, the network structure (units and interconnections) is specified by the designer
- The learning problem is finding a *good set of weights*
- The answer: *gradient descent*, because the form of the hypothesis formed by the network, $h_{\mathbf{w}}$, is
  - Differentiable! Because of the choice of sigmoid units
  - Very complex! Hence, direct computation of the optimal weights is not possible

Deriving a gradient descent update for networks

- Suppose for simplicity that we have a fully connected network with N input units (of indices $1, \ldots N$), H hidden units (of indices $N+1, \ldots N+H$) and one output unit (index $N+H+1$)
- Suppose we want to compute the weight update after seeing one instance $\langle \mathbf{x}, y \rangle$
- Let $o_i, i = 1, \ldots N+H+1$ be the outputs of all units in the network for the given inputs $\mathbf{x}$ (we omit the argument for simplicity of notation)
- The sum-squared error function is:

$$J(\mathbf{w}) = \frac{1}{2}(y - h_{\mathbf{w}}(\mathbf{x}))^2 = \frac{1}{2}(y - o_{N+H+1})^2$$

Deriving a gradient descent update for networks (2)

- The derivative with respect to the weights $w_{N+H+1,j}$ entering o_{N+H+1} is computed like usual:

$$\frac{\partial J}{\partial w_{N+H+1,j}} = -(y - o_{N+H+1})o_{N+H+1}(1 - o_{N+H+1})x_{N+H+1,j}$$

- For convenience, let

$$\delta_{N+H+1} = (y - o_{N+H+1})o_{N+H+1}(1 - o_{N+H+1})$$

- Hence, we can write:

$$\frac{\partial J}{\partial w_{N+H+1,j}} = -\delta_{N+H+1}x_{N+H+1,j}$$

Deriving a gradient descent update for networks (3)

- The derivative wrt the other weights, $w_{l,j}$ where $j = 1, \dots, N$ and $l = N+1, \dots, N+H$, can be computed using chain rule:

$$\begin{aligned}\frac{\partial J}{\partial w_{l,j}} &= -(y - o_{N+H+1})o_{N+H+1}(1 - o_{N+H+1}) \cdot \\ &\quad \cdot \frac{\partial}{\partial w_{l,j}} \mathbf{w}_{N+H+1}^T \mathbf{x}_{N+H+1} \\ &= -\delta_{N+H+1} w_{N+H+1,l} \frac{\partial}{\partial w_{l,j}} x_{N+H+1,l}\end{aligned}$$

- Recall that $x_{N+H+1,l} = o_l$. Hence, we have:

$$\frac{\partial}{\partial w_{l,j}} x_{N+H+1,l} = o_l(1 - o_l)x_{l,j}$$

- Putting these together, and using similar notation as before, we have:

$$\frac{\partial J}{\partial w_{l,j}} = -o_l(1 - o_l)\delta_{N+H+1} w_{N+H+1,l} x_{l,j} = -\delta_l x_{l,j}$$

Backpropagation algorithm

- Just gradient descent over all weights in the network!
- We put together the two phases described above:

  1. **Forward pass:** Compute the outputs of all units in the network, $o_k, k = N+1, \ldots N+H+1$, going in increasing order of the layers
  2. **Backward pass:** Compute the δ_k updates described before, going from $k = N+H+1$ down to $k = N+1$ (in decreasing order of the layers)
  3. Update to all the weights in the network. For weight $w_{i,j}$, $\frac{\partial}{\partial w_{i,j}} J = \delta_i x_{i,j} = \delta_i o_j$. Thus:

$$w_{i,j} \leftarrow w_{i,j} + \alpha_{i,j} \delta_i x_{i,j}$$

     where $\alpha_{i,j}$ is a step size.

Backpropagation algorithm (more details)

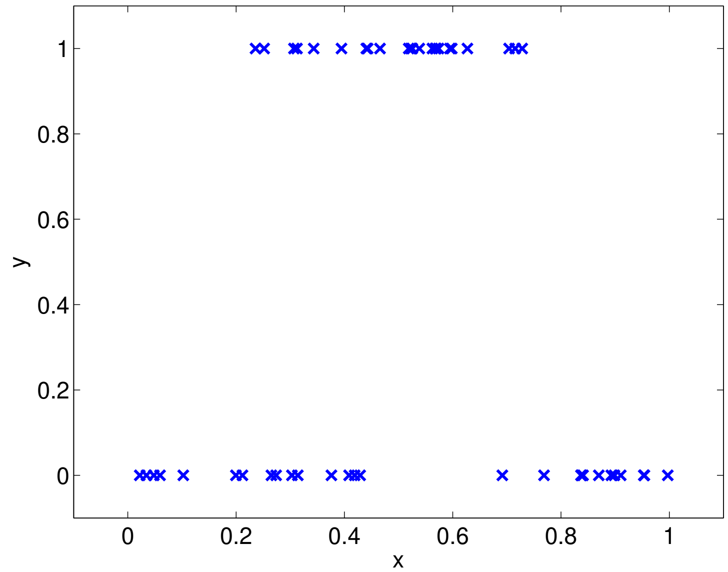
1. Initialize all weights to small random numbers.
2. Repeat until satisfied:
   (a) Pick a training example
   (b) Input example to the network and compute the outputs o_k
   (c) For each output unit k, $\delta_k \leftarrow o_k(1-o_k)(y_k-o_k)$
   (d) For each hidden unit l

$$\delta_l \leftarrow o_l(1-o_l)\sum_{k \in outputs} w_{k,l}\delta_k$$

   (e) Update each network weight: $w_{i,j} \leftarrow w_{i,j} + \alpha_{i,j}\delta_i x_{i,j}$
      - $x_{i,j}$ is the input to unit i from unit j
      - $\alpha_{i,j}$ is the learning rate or step size

Visual example of backprop in action

- Training on univariate, unimodal data set:



- 1-2-1 architecture (two inputs, two hidden, one output)
- Small random initial weights (normally distributed, with $\sigma = 0.1$)
- Fixed learning rate 0.1
- 100000 epochs of batch backprop (i.e., true gradient descent)

Show series "ANN.ModeFit.eps"!