Lecture 19: Neural Networks

- Perceptrons
- Sigmoid neurons
- Adjusting parameters of the sigmoid using LMS
- Feed-forward neural networks
- Backpropagation

The Human Brain

- Contains ~ 10^{11} neurons, each of which may have up to ~ 10^{4-5} input/output connections
- Each neuron is fairly slow, with a switching time of ~ 1 millisecond
- Computers are at least 10^6 times faster in raw switching speed
- Yet the brain is very fast and reliable at computationally intensive tasks (e.g. vision, speech recognition, knowledge retrieval)
- The brain is also fault-tolerant, and exhibits graceful degradation with damage
- Maybe this is due to its architecture, which does massive parallel computation!

Connectionist Models

- Based on the assumption that a computational architecture similar to the brain would duplicate (at least some of) its wonderful abilities.
- *Properties of artificial neural nets (ANNs)*:
  - Many neuron-like threshold switching units
  - Many weighted interconnections among units
  - Highly parallel, distributed process
  - Emphasis on tuning weights automatically
- Many different kinds of architectures, motivated both by biology and mathematics/efficiency of computation

Recall: Linear Hypotheses

- Consider hypotheses of the form:

$$h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} = w_0 + w_1 x_1 + \cdots + w_n x_n$$

  where $\mathbf{w} = \langle w_0, w_1, \ldots w_n \rangle$ is a *weight or parameter vector*
- The goal is to learn weights w_j that *minimize the sum squared error* over the training examples:

$$J(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{m} (y_i - h_{\mathbf{w}}(\mathbf{x_i}))^2$$

  where m is the number of training examples.
- The function $J(\mathbf{w})$ defines an error surface in weight space.
- We use *gradient descent* to search for a good set of weights!

Gradient Descent

- Direction of the steepest descent is given by the *gradient* function:

$$\nabla J = \left[\frac{\partial J}{\partial w_0}, \frac{\partial J}{\partial w_1}, \cdots \frac{\partial J}{\partial w_n}\right]$$

- Training rule:

$$w_j \leftarrow w_j - \alpha \frac{\partial J}{\partial w_j}, \forall j = 0, \ldots n$$

Linear Models for Classification

- The linear hypotheses we discussed are good for predicting real-valued functions
- What if we want to solve a binary classification problem?
  - E.g., predict whether a tumor will recur
  - E.g., predict whether a game will be won or lost from a board position
- Recall: in a binary classification problem the outputs, y_i, take one of two discrete values: $\{0, 1\}$ or $\{-1, +1\}$ as convenient
- Can we develop linear models for classification as we did for regression?

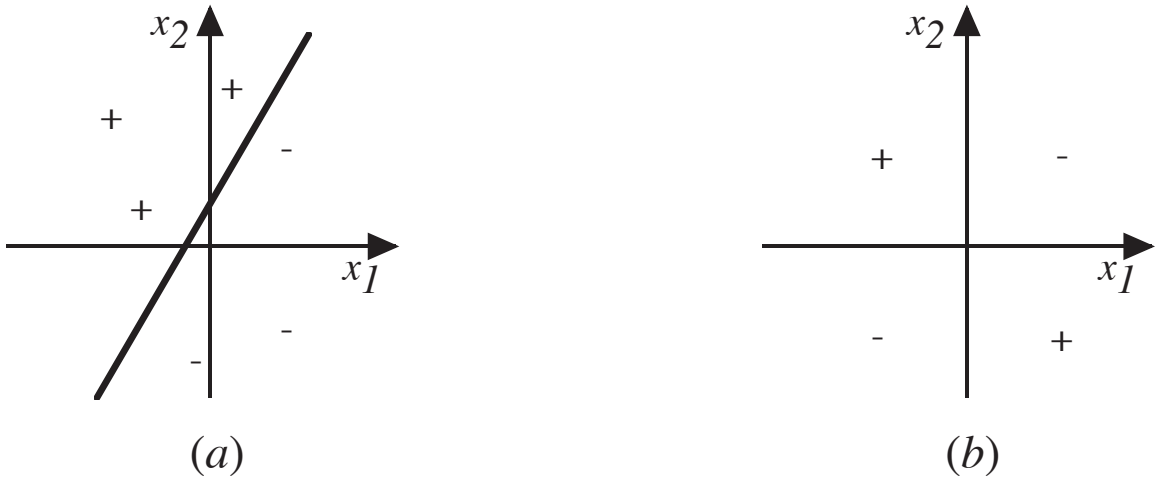
Perceptron

- We can take a linear combination and threshold it:

$$h_{\mathbf{w}}(\mathbf{x}) = \text{sgn}(\mathbf{x} \cdot \mathbf{w}) = \begin{cases} +1 & \text{if } \mathbf{x} \cdot \mathbf{w} > 0 \\ -1 & \text{otherwise} \end{cases}$$

  This is called a *perceptron*.
- The output is taken as the predicted class.

Decision Surface of a Perceptron



- Represents some useful functions.

  E.g., what weights represent $x_1 \wedge x_2$? (Assume boolean x_i)
- But some functions *not linearly separable*!

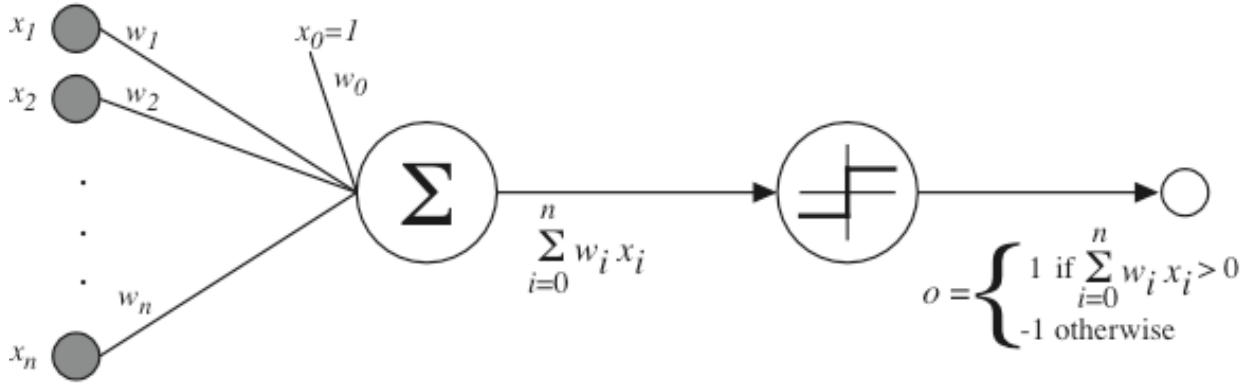
  E.g. XOR.
- Therefore, we need networks of perceptron-like elements.

The Need for Networks

- Perceptrons have very simple decision surfaces (only linearly separable functions)
- If we connect them into networks, the error surface for the network is not differentiable (because of the hard threshold)
- So we cannot apply gradient descent to find a good set of weights.
- We would like a *soft threshold*!

  Nicer math, and closer to biological neurons.

Sigmoid Unit (Neuron)



$\sigma(x)$ is the sigmoid function: $\frac{1}{1+e^{-x}}$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$

We can derive gradient decent rules to train:

- One sigmoid unit
- *Multi-layer networks* of sigmoid units (called Backpropagation)

Logistic (Sigmoid) Hypothesis

$$h_{\mathbf{w}}(\mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}}$$

- We want to determine a "good" weight vector $\mathbf{w}$
- Assume again that we want to minimize the sum-squared error
- Note that in this case:
  - If the output is predicted correctly the error is 0
  - If the output is predicted incorrectly the error is 1

Minimizing Sum-Squared Error

- Error function:

$$J(\mathbf{w}) = 1/2 \sum_i (y_i - h_{\mathbf{w}}(\mathbf{x_i}))^2$$

- Gradient of the error:

$$\nabla J = -\sum_i (y_i - h_{\mathbf{w}}(\mathbf{x_i})) \nabla h_{\mathbf{w}}(\mathbf{x_i})$$

- For sigmoid hypotheses, we have:

$$\nabla h_{\mathbf{w}}(\mathbf{x_i}) = h_{\mathbf{w}}(\mathbf{x_i})(1 - h_{\mathbf{w}}(\mathbf{x_i}))\mathbf{x_i}$$

- We obtain the weight update rule:

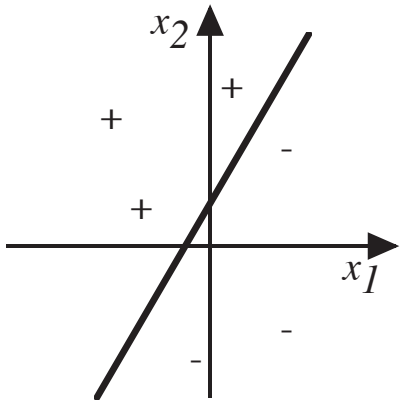
$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \sum_i (y_i - h_{\mathbf{w}}(\mathbf{x_i})) h_{\mathbf{w}}(\mathbf{x_i})(1 - h_{\mathbf{w}}(\mathbf{x_i}))\mathbf{x_i}$$

- We can do batch or on-line updates

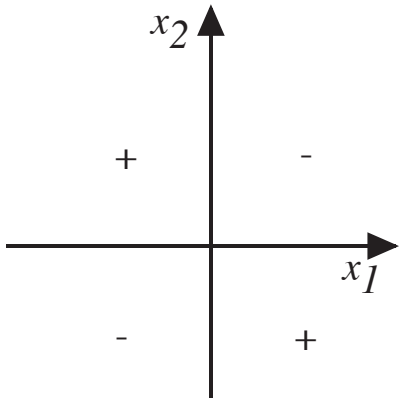
The Need for Networks

Sigmoid units vs. perceptron:

- Sigmoid units provide "soft" threshold, perceptron provides "hard" threshold
- Expressive power is the same: limited to linearly separable instances



(a)

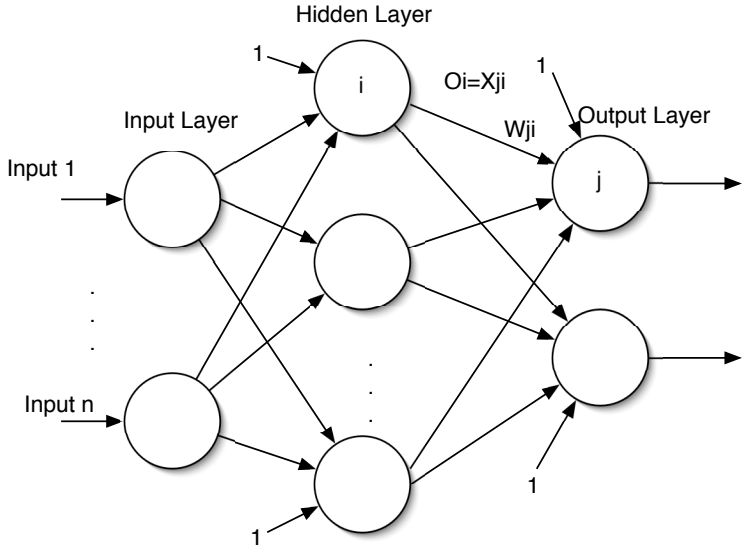


(b)

Example: Logical Functions of Two Variables

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

Feed-Forward Neural Networks

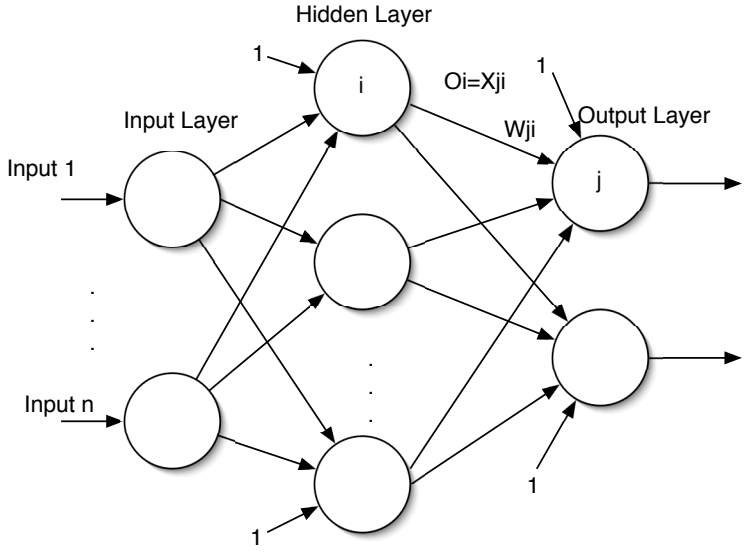


- A collection of units (neurons) with sigmoid activation, arranged in *layers*
- Layers 0 is the *input layer*, its units just copy the inputs
- Last layer, K, is called *output layer*, its units provide the output
- Layers $1, \ldots K-1$ are *hidden layers*, cannot be detected outside of the network

Why This Name?

- In *feed-forward networks* the output of units in layer k becomes an input for units in layers $k+1, k+2 \ldots K$.
- 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 to units in layer $k+1$ *only*
- In *fully connected networks*, all units in layer k are connected to all units in layer $k+1$

Notation



- $w_{j,i}$ is the weight on the connection from unit i to unit j
- By convention, $x_{j,0} = 1, \forall j$
- The output of unit j, denoted o_j, is computed using a sigmoid: $o_j = \sigma(\mathbf{w}_j \cdot \mathbf{x}_j)$ where $\mathbf{w}_j$ is vector of weights entering unit j and $\mathbf{x}_j$ is vector of inputs to unit j
- By definition of the connections, $x_{j,i} = o_i$

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 a single *forward pass*:

For layer $k = 1$ to K

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

$$o_j = \sigma(\mathbf{w}_j \cdot \mathbf{x}_j), \forall j \in \text{Layer } k$$

2. Copy this output as inputs to the next layer:

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

Learning in Feed-Forward Neural Networks

- Assume the network structure (units and connections) is given
- The learning problem is finding a *good set of weights*
- The answer: *gradient descent*, because 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

Gradient Descent Update for Neural Networks

- Assume we have a fully connected network:
  - N input units (indexed $1, \ldots N$)
  - One hidden layer with H hidden units (indexed $N+1, \ldots N+H$)
  - One output unit (indexed $N+H+1$)
- Suppose we want to compute the weight update after seeing 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 input $\mathbf{x}$
- The sum-squared error function is:

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

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 as 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:

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

Gradient Descent Update for Networks (3)

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

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

- 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:

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

Backpropagation Algorithm

- Just do 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:

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

Backpropagation Algorithm in Detail

- Initialize all weights to small random numbers.
- Repeat until satisfied:
  - Pick a training example
  - Input example to the network and compute output o_{N+H+1}
  - For the output unit, compute the correction: $\delta_{N+H+1} \leftarrow o_{N+H+1}(1 - o_{N+H+1})(y - o_{N+H+1})$
  - For each hidden unit h, compute its share of the correction:

$$\delta_h \leftarrow o_h(1 - o_h)w_{N+H+1,h}\delta_{N+H+1}$$

  - Update each network weight: For $h = 1, \ldots H$,

$$w_{h,i} \leftarrow w_{h,i} + \alpha_{h,i}\delta_h x_{h,i}, i = 1, \ldots N$$

$$w_{N+H+1,h} \leftarrow w_{N+H+1,h} + \alpha_{N+H+1,h}\delta_{N+H+1}o_h$$

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 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
- *Any function* can be approximated to arbitrary accuracy by a network with two hidden layers

Backpropagation Variations

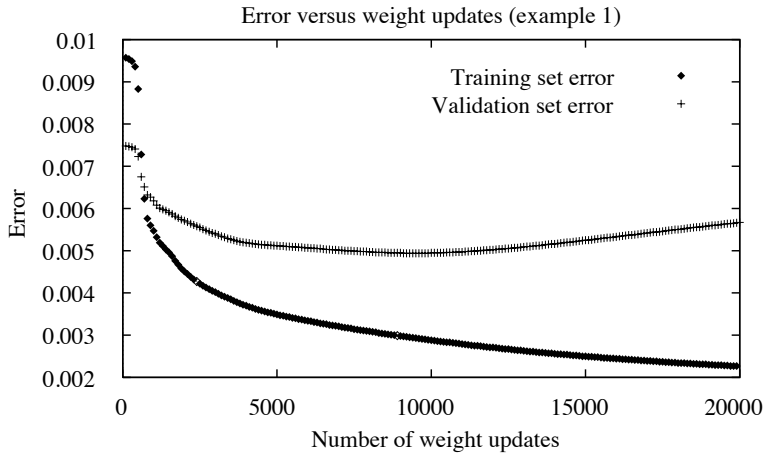
- Previous version corresponds to incremental (stochastic) gradient descent
- An analogous *batch version* can be used as well:
  - Loop through the training data, accumulating weight changes
  - Update weights
- One pass through the data set is called an *epoch*
- Algorithm can be easily generalized to predict probabilities, instead of minimizing sum-squared error
- It can also be generalized to arbitrary directed graphs

Convergence of Backpropagation

- Backpropagation performs gradient descent over *all* the parameters in the network
- Hence, if the learning rate is appropriate, the algorithm is *guaranteed to converge to a local minimum* of the cost function
  - NOT the global minimum
  - Can be much WORSE than global minimum
  - There can be MANY local minima (Auer et al, 1997)
- Solution: *random restarts* = train multiple nets with different initial weights.
- In practice, the solution found is often very good
- Training can take thousands of iterations $\rightarrow$ VERY SLOW!

  But using network after training is very fast.

Overfitting in Feed-Forward Networks



Overfitting in neural nets comes from three sources:

- Too many weights
- Training for too long
- Weights that have become too extreme

Use a validation set to decide when to stop training!

Practical Issues

- The choice of initial weights has great impact on convergence!
  - If the input size is N, and N is large, a good heuristic is to choose initial weights between $-1/N$ and $1/N$.
- Backpropagation is very sensitive to learning rate
  - If it is too large, the weights diverge.
  - If it is too small, convergence is very slow
- Sometimes it is appropriate to use different learning rates for different layers and units
- There are algorithms that try to change the learning rate automatically

More Practical Issues

- It is bad to have inputs of very different magnitude
- To avoid this, sometimes we re-encode the input variables. E.g.
  - *1-of-n encoding:* discretize into a given number of intervals n

    E.g. If $x \in [0, 1000], n = 10$:

    $x \in [0, 100] \rightarrow 1\ 0\ 0\ 0\ 0\ 0\ 0$

    $x \in [100, 200] \rightarrow 0\ 1\ 0\ 0\ 0\ 0\ 0$

    etc.
  - *Thermometer encoding:* like 1-of-n, but if the variable falls in the i-th interval, all bits $1 \ldots i$ are set to 1

    E.g. If $x \in [0, 1000], n = 10$:

    $x \in [0, 100] \rightarrow 1\ 0\ 0\ 0\ 0\ 0\ 0$

    $x \in [100, 200] \rightarrow 1\ 1\ 0\ 0\ 0\ 0\ 0$
- A thermometer encoding is usually better than 1-of-n

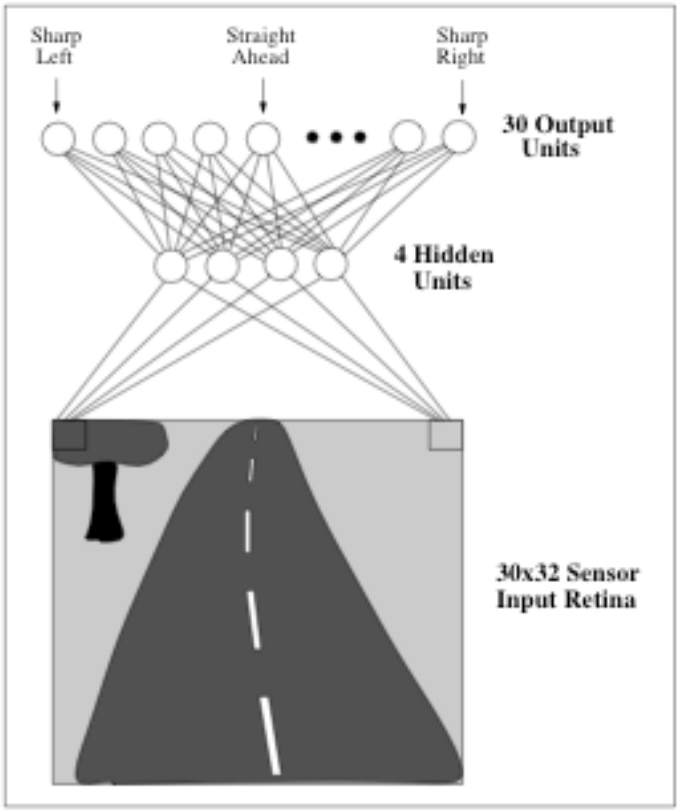
And Yet More Practical Issues...

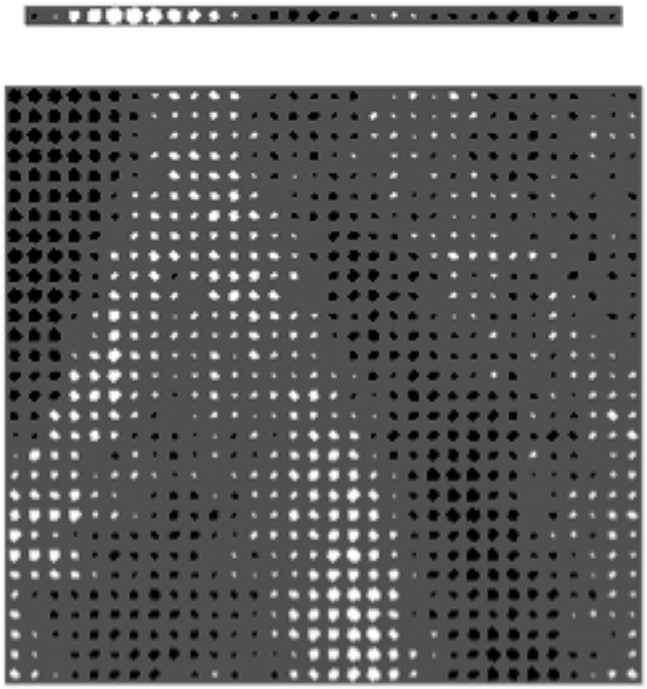
- Too many hidden units hurt! Why?
  - Good heuristic: $\log(N)$, where N is the number of inputs.
- Too many hidden layers also usually hurt!
- Remember: Two layers are always enough

Example: ALVINN (Pomerleau, 1993)

- *Task:* learn how to steer a car automatically
- *Inputs:* grey-level pixels from images captured by a camera on top of the car
- *Output:* 30 units, corresponding to different steering angles
- The action is picked according to which unit has the highest activation
- Training data gathered during roughly 2 hours of driving by a person
- Training algorithm: backpropagation
- Was able to drive across the U.S (with a person braking, and on highways only).

Example: ALVINN (Pomerleau, 1993)





The right shows the weights of one of the hidden units to the output (top row) and the weights coming into the same hidden unit from the inputs (square)

When to Consider Using Neural Networks

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete or real valued, or a vector of values
- Possibly noisy data
- Training time is unimportant
- Form of target function is unknown
- Human readability of results is unimportant
- Computation of the output based on the input has to be fast

Examples:

- Speech phoneme recognition [Waibel] and synthesis [Nettalk]
- Image classification [Kanade, Baluja, Rowley]
- Financial prediction