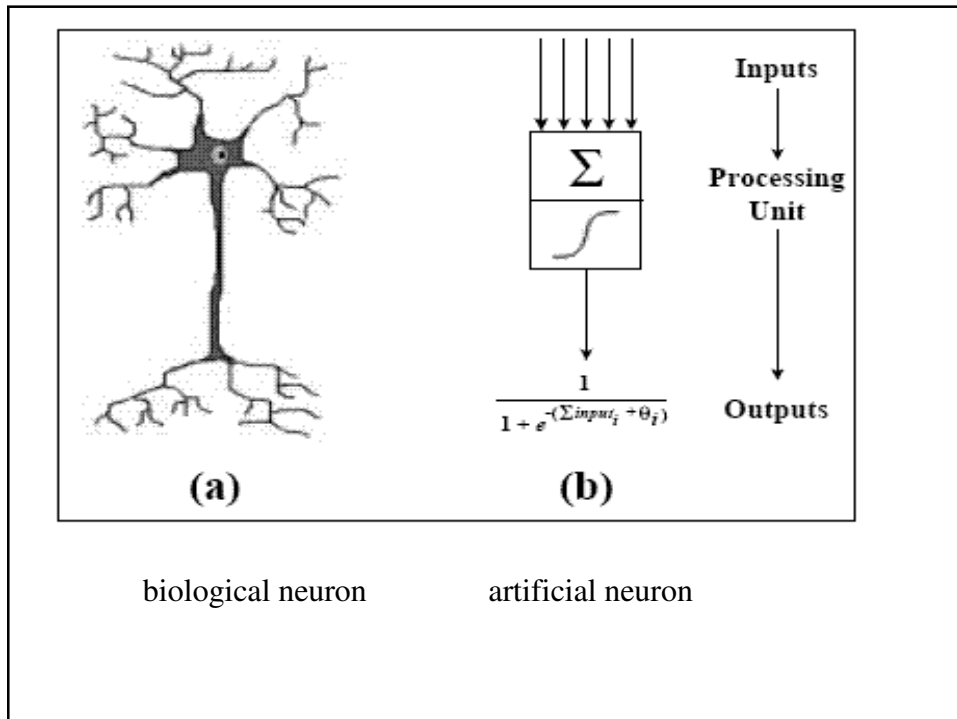
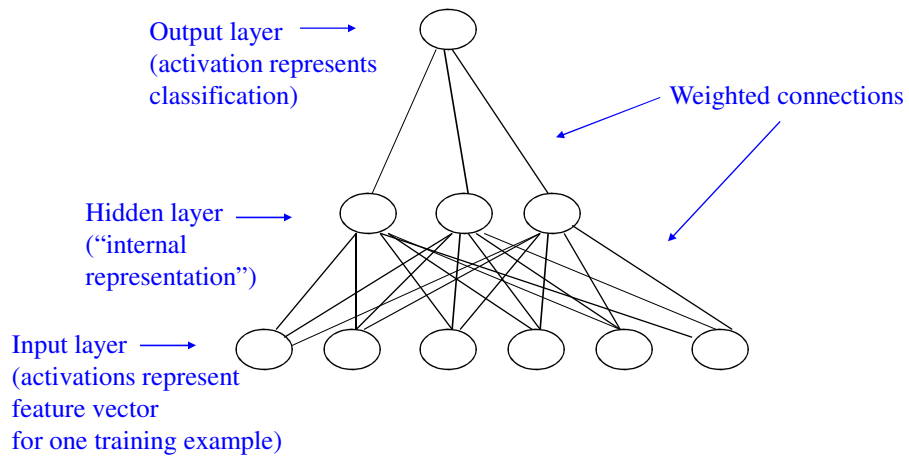


Neural Networks

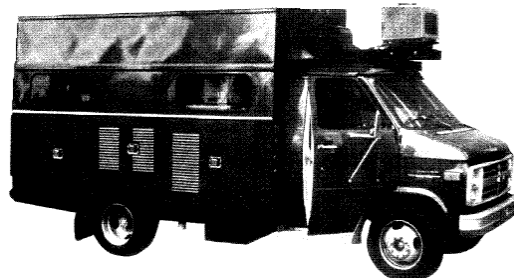


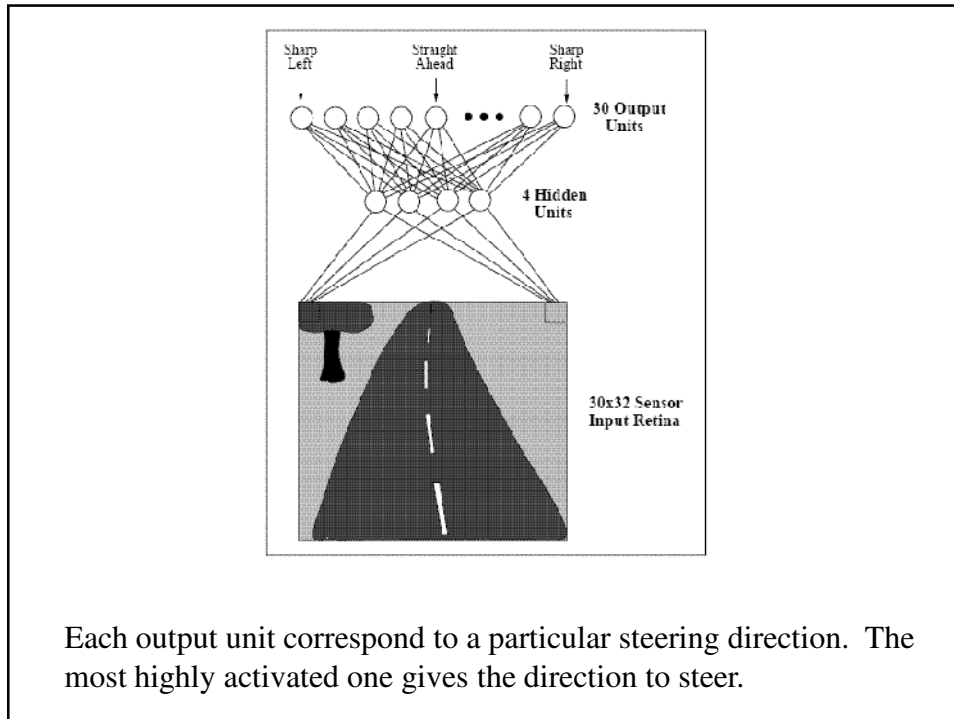
A two-layer neural network



Example (from textbook): ALVINN (Pomerleau, 1993)

- ALVINN learns to drive an autonomous vehicle at normal speeds on public highways (!)
- Input: 30 x 32 grid of pixel intensities from camera





What kinds of problems are suitable for neural networks?

- Have sufficient training data
- Long training times are acceptable
- Not necessary for humans to understand learned target function or hypothesis

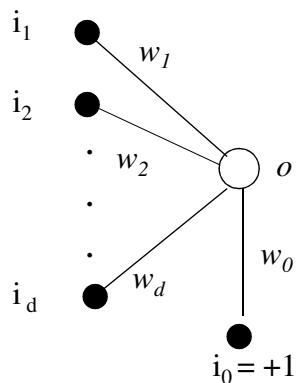
Advantages of neural networks

- Designed to be parallelized
- Robust on noisy training data
- Fast to evaluate new examples
- “Bad at logic, good at frisbee” (Andy Clark)

Perceptrons

(Note: Slides have changed from those posted last week)

- Invented by Rosenblatt, 1950s
- Single-layer feedforward network:

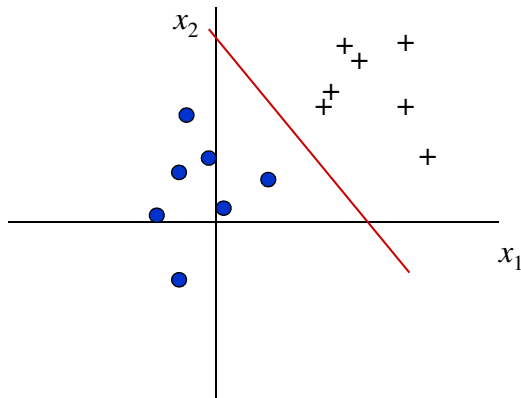


$$\begin{aligned} \text{output } o &= f(\mathbf{i}, \mathbf{w}) \\ &= \text{sgn}\left[\sum_{j=1}^d w_j i_j + w_0\right] \end{aligned}$$

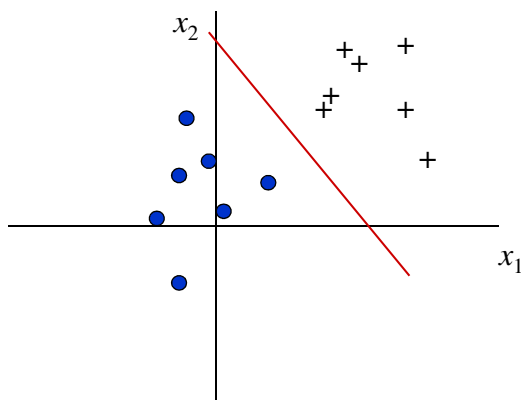
This can be used for classification:

+1 = positive, -1 = negative

- Consider example with two inputs, i_1, i_2 .
- We can view trained perceptron as defining a linear decision surface. What is its equation?



- Consider example with two inputs, i_1, i_2 .
- We can view trained perceptron as defining a linear decision surface. What is its equation?



$$w_1 i_1 + w_2 i_2 + w_0 = 0$$

$$i_2 = -\frac{w_1}{w_2} i_1 - \frac{w_0}{w_2}$$

- Can generalize to d-dimensions:

$$w_1 i_1 + w_2 i_2 + \dots + w_d i_d + w_0 = 0$$

Decision surface is a $(d - 1)$ -dimensional hyperplane.

Notation

- Let $S = \{(\mathbf{i}^k, T^k) : k = 1, 2, \dots, N\}$ be a training set.

Note: \mathbf{i}^k is a vector of inputs, and T^k is $\in \{+1, -1\}$ for binary classification, $T^k \in \mathfrak{R}$ for regression.

- Output o :

Given input \mathbf{i} and weights \mathbf{w}

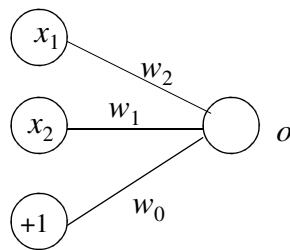
$$o = \text{sgn} \left[\sum_{j=1}^d w_j i_j + w_0 \right] = \text{sgn} [\mathbf{w}^T \cdot \mathbf{i}]$$

Example

- $S = \{((0,0), -1), ((0,1), 1)\}$
- Let $\mathbf{w} = \{w_0, w_1, w_2\} = \{0.1, 0.1, -0.3\}$

What is o^1 (first training example)?

What is o^2 (second training example)?



Examples

- Give a perceptron that represents the OR function.
- Give a perceptron that represents the AND function.

Perceptron learning rule

Define the error with respect to a training example \mathbf{i} and a weight vector \mathbf{w} :

$$Err(\mathbf{i}, \mathbf{w}) = o - T$$

After processing each training example \mathbf{i} , modify each weight as follows:

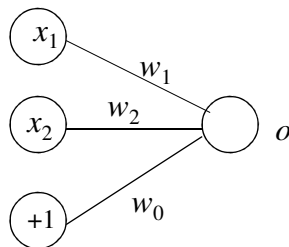
$$w_j = w_j + \alpha(o - T)i_j$$

where α is the user-set *learning rate*.

Example

- $S = \{((0,0), -1), ((0,1), 1)\}$
- Let $\mathbf{w} = \{w_0, w_1, w_2\} = \{0.1, 0.1, -0.3\}$
- Let $\eta = .2$.

Perceptron weight updates:

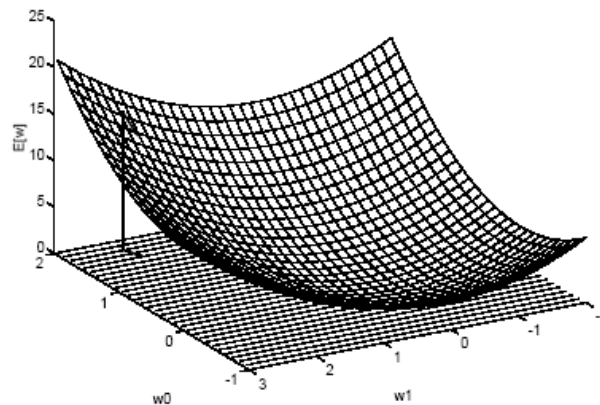


New perceptron weights:

Training a perceptron

1. Start with random weights, $\mathbf{w} = (w_1, w_2, \dots, w_d)$.
2. Select training example $(\mathbf{i}^k, T^k) \in S$.
3. Run the perceptron with input \mathbf{x}^k and weights \mathbf{w} to obtain o .
4. Let α be the training rate. Now,
$$w_j \leftarrow w_j + \Delta w_j$$
where
$$\Delta w_j = \alpha (o^k - T^k) i_j^k$$
5. Go to 2.

- Note: The perceptron learning rule performs gradient descent in "weight space" :



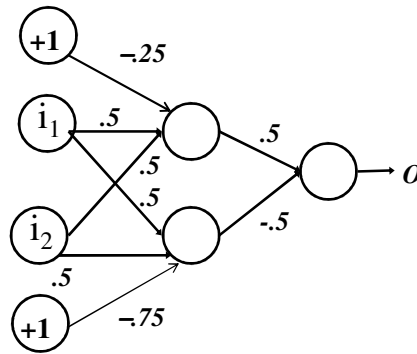
From T. M. Mitchell, *Machine Learning*

Perceptron Convergence Theorem

- The perceptron learning rule has been proven to converge to correct weights in a finite number of steps, independent of the initial weights, provided the training examples are linearly separable.
- What is an example of a function that is not linearly separable (and thus not learnable by a perceptron)?

- 1969: Minsky and Papert proved that perceptrons cannot represent non-linearly separable target functions.
- However, they proved that for binary inputs, any transformation can be carried out by adding fully connected hidden layer.

Two layer network to perform **xor** function



- **Good news:** Adding hidden layer allows more target functions to be represented.
- **Bad news:** No algorithm for learning in multi-layered networks, and no convergence theorem!
- Quote from Minsky and Papert's book, *Perceptrons* (1969):

"[The perceptron] has many features to attract attention: its linearity; its intriguing learning theorem; its clear paradigmatic simplicity as a kind of parallel computation. There is no reason to suppose that any of these virtues carry over to the many-layered version. Nevertheless, we consider it to be an important research problem to elucidate (or reject) our intuitive judgment that the extension is sterile."

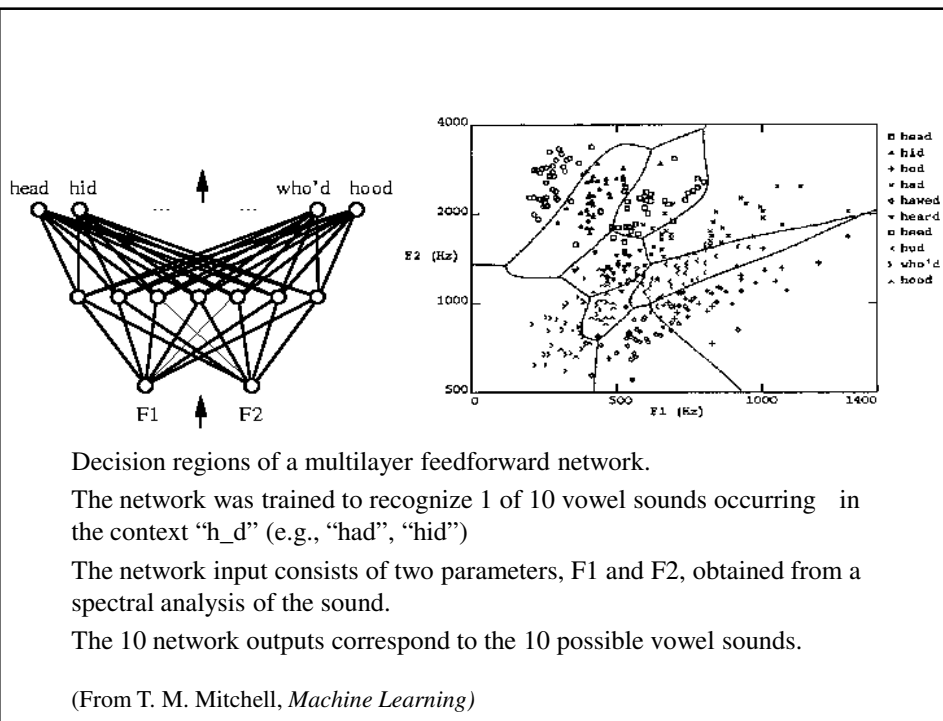
- Two major problems they saw were:
 1. How can the learning algorithm apportion credit (or blame) to individual weights for incorrect classifications depending on a (sometimes) large number of weights?
 2. How can such a network learn useful higher-order features?
- **Good news:** Successful credit-apportionment learning algorithms developed soon afterwards (e.g., back-propagation). Still successful, in spite of lack of convergence theorem.

Limitations of perceptrons

- Perceptrons only be 100% accurate only on linearly separable problems.
- Multi-layer networks (often called *multi-layer perceptrons*, or *MLPs*) can represent any target function.
- However, in multi-layer networks, there is no guarantee of convergence to minimal error weight vector.

Multi-layer Perceptrons (MLPs)

- Single-layer perceptrons can only represent linear decision surfaces.
- Multi-layer perceptrons can represent non-linear decision surfaces



Differentiable Threshold Units

- In general, in order to represent non-linear functions, need non-linear activation function at each unit.
- In order to do gradient descent on weights, need differentiable activation function.

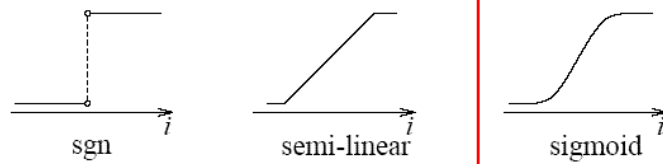


Figure 2.2: Various activation functions for a unit.

Sigmoid activation function:

$$o = \sigma(\mathbf{w} \cdot \mathbf{x}), \text{ where } \sigma(y) = \frac{1}{1 + e^{-y}}$$

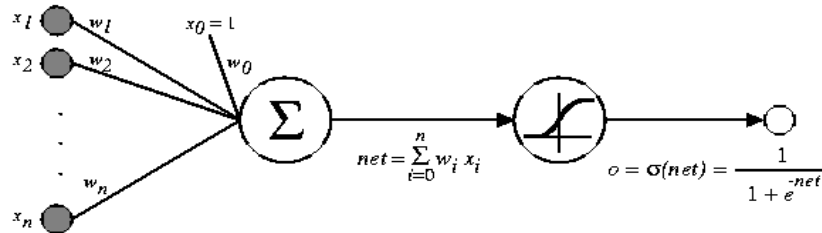
To do gradient descent in multi-layer networks, need to generalize perceptron learning rule for non-linear activation functions.

Why use sigmoid activation function?

- Note that sigmoid activation function is non-linear, differentiable, and approximates a sgn function.
- The derivative of the sigmoid activation function is easily expressed in terms of its output:

$$\frac{d\sigma(y)}{dz} = \sigma(y) \cdot (1 - \sigma(y))$$

This is useful in deriving the back-propagation algorithm.



$\sigma(x)$ is the sigmoid function

$$\frac{1}{1 + e^{-x}}$$

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

From T. M. Mitchell, *Machine Learning*

Training multi-layer perceptrons

Assume two-layer networks:

I. For each training example:

1. Present input to the input layer.
2. Forward propagate the activations times the weights to each node in the hidden layer.

3. Forward propagate the activations times weights from the hidden layer to the output layer.
4. At each output unit, determine the error Err .
5. Run the back-propagation algorithm to update all weights in the network.

II. Repeat (I) for a given number of “epochs” or until accuracy on training or test data is acceptable.

Face recognition via Neural Networks

(From T. M. Mitchell, *Machine Learning*, Chapter 4, on electronic reserve at library)

- **Task:** classify camera images of various people in various poses.
- **Data:** Photos, varying:
 - Facial expression: *happy, sad, angry, neutral*
 - Direction person is facing: *left, right, straight ahead, up*
 - Wearing sunglasses?: *yes, no*

Within these, variation in background, clothes, position of face for a given person.



an2i_left_angry_open_4



an2i_right_sad_sunglasses_4



glickman_left_angry_open_4

Training feed-forward neural networks

Assume two-layer network with random initial weights (small positive and negative values):

I. For each training example:

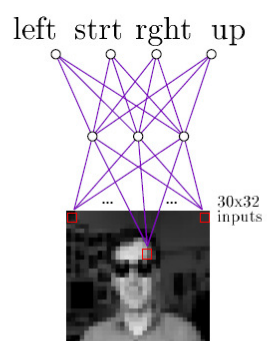
1. Present input to the input layer.
2. Forward propagate the activations times the weights to each node in the hidden layer.

3. Forward propagate the activations times weights from the hidden layer to the output layer.
4. At each output unit, determine the error *Err*.
5. Run the back-propagation algorithm to update all weights in the network.
 - **Learning rate:** Determines magnitude of weight updates
 - **Momentum:** Determines magnitude of influence of previous weight change (to keep changes going in same direction)

II. Repeat (I) for a given number of “epochs” or until accuracy on training or test data is acceptable.

Design Choices

- Input encoding
- Output encoding
- Number of hidden units
- Learning rate
- Momentum

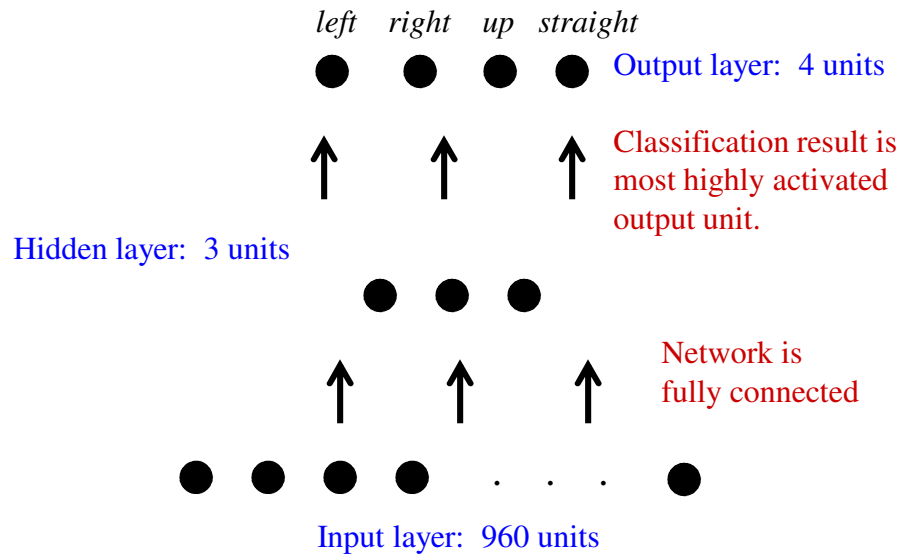


Typical input images

- Preprocessing of photo:
 - Create 30x32 coarse resolution version of 120x128 image
 - This makes size of neural network more manageable
- Input to neural network:
 - Photo is encoded as $30 \times 32 = 960$ pixel intensity values, scaled to be in $[0,1]$
 - One input unit per pixel
- Output units:
 - Encode classification of input photo

- Possible target functions for neural network:
 - Direction person is facing
 - Identity of person
 - Gender of person
 - Facial expression
 - etc.
- As an example, consider target of “direction person is facing”.

Network architecture



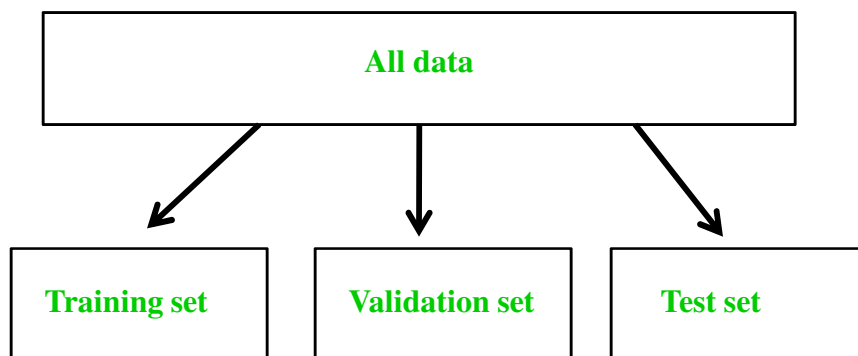
Target function

- Target function is:
 - Output unit should have activation 0.9 if it corresponds to correct classification
 - Otherwise output unit should have activation 0.1
- Use these values instead of 1 and 0, since sigmoid units can't produce 1 and 0 activation.

Other parameters

- Learning rate $\eta = 0.3$
- Momentum $\alpha = 0.3$
- If these are set too high, training fails to converge on network with acceptable error over training set.
- If these are set too low, training takes much longer.

Training



Training

- For maximum of M epochs:
 - For each training example
 - Input photo to network
 - Propagate activations to output units
 - Determine error in output units
 - Adjust weights using back-propagation algorithm
 - Test accuracy of network on validation set. If accuracy is acceptable, stop algorithm, and run resulting network on test set.
- Demo of code

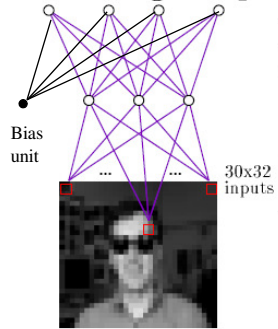
Epoch	delta	trainperf	trainerr	t1perf	t1err	t2perf	t2err
0	0.0	50.9025	0.0972029	46.7626	0.103391	0.0	0.0
1	29.4377	50.9025	0.07842	47.482	0.0805035	0.0	0.0
2	30.6895	59.2058	0.0737351	53.2374	0.0763508	0.0	0.0
3	31.2238	67.87	0.0665212	63.3094	0.0704276	0.0	0.0
4	30.8616	76.8953	0.0578792	66.1871	0.0638382	0.0	0.0
5	29.4834	80.5054	0.0506615	71.223	0.0588367	0.0	0.0
6	27.4804	83.7545	0.0452837	75.5396	0.0551372	0.0	0.0
7	25.418	85.1986	0.0412303	74.8201	0.0522873	0.0	0.0
8	23.5952	85.5596	0.0380707	76.9784	0.0501809	0.0	0.0
9	21.9776	85.9206	0.035415	76.9784	0.0485298	0.0	0.0
10	20.3628	86.2816	0.0330951	78.4173	0.0469859	0.0	0.0

After 100 epochs of training.

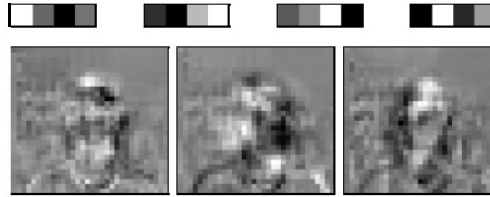
Understanding weight values

(From T. M. Mitchell, Machine Learning)

left strt right up



Weights from each hidden unit to four output units



Weights from each pixel to hidden units 1, 2, 3 (white = high, black = low)

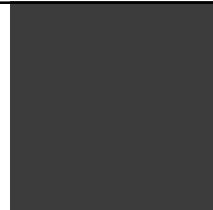
Hidden unit 2 has high positive weights from right side of face. If person is looking to the right, this will cause unit 2 to have high activation. Output unit *right* has high positive weight from hidden unit 2.



Bias to output weights



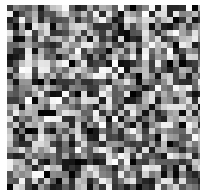
Hidden1 to output weights



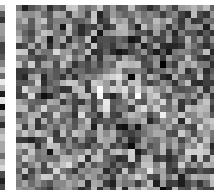
Hidden2 to output weights



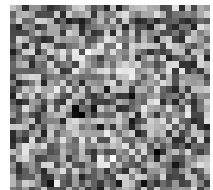
Hidden3 to output weights



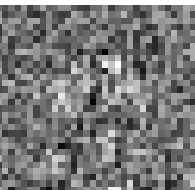
Input to bias weights



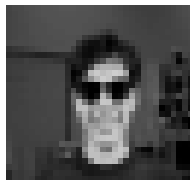
Input to hidden1 weights



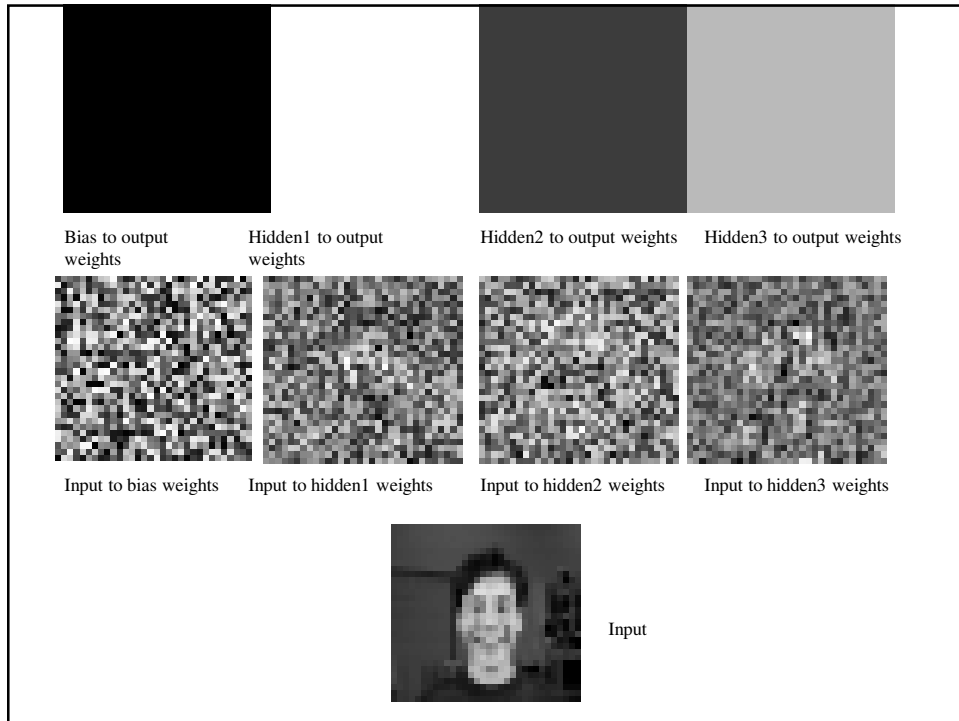
Input to hidden2 weights



Input to hidden3 weights



Input



Multi-layer networks can do everything

- **Universal approximation theorem:** One layer of hidden units suffices to approximate any function with finitely many discontinuities to arbitrary precision, if the activation functions of the hidden units are non-linear.
- In most applications, two-layer network with sigmoid activation function used.

Back-propagation

- Extends gradient descent algorithm to multi-layer network with (possibly) multiple output units.
- Error E is now sum of errors over all output units:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{d \in S} \sum_{k \in \text{outputs}} (t_{kd} - o_{kd})^2,$$

where d indexes the training examples in training set S , and k indexes the output units in the network.

Back-propagation algorithm (stochastic version)

- Assume you are given a set S of training examples (\mathbf{x}, \mathbf{t}) , where \mathbf{x} is a vector of n_{in} network input values and \mathbf{t} is a vector of n_{out} target network output values.
- Denote the input from unit i to unit j by x_{ji} and the weight from unit i to unit j by w_{ji} .
- Create a feed-forward network with n_{in} input units, n_{out} output units, and n_{hidden} hidden units.

- Initialize the network weights \mathbf{w} to small random numbers (e.g., between -0.05 and $+0.05$).
- Until the termination condition is met, Do:
 - For each $(\mathbf{x}, \mathbf{t}) \in \mathbf{S}$, Do:

1. *Propagate the input forward:*

- Input \mathbf{x} to the network and compute the output o_u of every unit u in the network.

2. *Propagate the errors backward:*

- For each output unit k , calculate error term δ_k :

$$\delta_k \leftarrow o_k(1-o_k)(t_k - o_k)$$

- For each hidden unit h , calculate error term δ_h :

$$\delta_h \leftarrow o_h(1-o_h) \sum_{k \in \text{output units}} w_{kh} \delta_k$$

3. *Update the weights :*

- For each network weight w_{ji} :

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

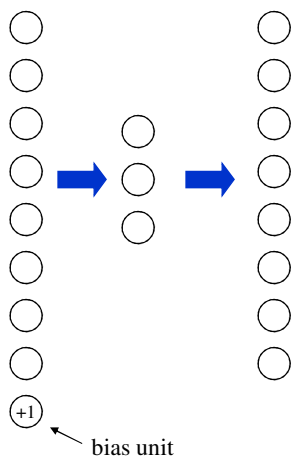
$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

Will this algorithm converge on optimal weights?

- Not always, because of local minima in error surface.
- “In many practical applications the problem of local minima has not been found to be as severe as one might fear.”
- Especially true with large networks, since many dimensions give many “escape routes”.
- But “no methods are known to predict with certainty when local minima will cause difficulties.”

Hidden-layer representation in multi-layer perceptrons

“Auto-associator network”



Input	Hidden activations	Output
10000000	→ .89 .04 .08	→ 10000000
01000000	→ .15 .99 .99	→ 01000000
00100000	→ .01 .97 .27	→ 00100000
00010000	→ .99 .97 .71	→ 00010000
00001000	→ .03 .05 .02	→ 00001000
00000100	→ .01 .11 .88	→ 00000100
00000010	→ .80 .01 .98	→ 00000010
00000001	→ .60 .94 .01	→ 00000001

Results after network was trained for 5000 epochs. What is the encoding?

- One advantage of neural networks is that “high-level feature extraction” can be done automatically!

(Similar idea in face-recognition task. Note that you didn't have to extract high-level features ahead of time.)

How the auto-associator network was trained

- Initial weights set to random values in $(-0.1, 0.1)$.
- Learning rate $\eta = 0.3$
- Momentum $\alpha = 0$
- 5000 epochs of training with back-propagation

Example:
Handwritten zip code recognition with neural
networks
(Le Cun et al, 1990)

- Data: 9298 segmented numerals digitized from handwritten zipcodes

40004 75246
14189-2087 23505
96203 14310
44151 05753

(Note: segmentation already done)

1410119134857268032264141
8663597202992997225100467
0130841115910106154061036
3110641110304752620099799
6689120867285571314279554
6020187301871129910899709
8401097075973319720155190
6510735182551828143580909
4317875416554603546035460
5518255108503047520439401

Figure 2: Examples of normalized digits from the testing set.

Training and test sets

- Training set: 7291 handwritten digits plus 2549 typeset digits from 35 different fonts
- Test set 2087 handwritten digits and 700 typeset digits (different fonts than in training set)
- Both sets contain some noise (misclassified digits)

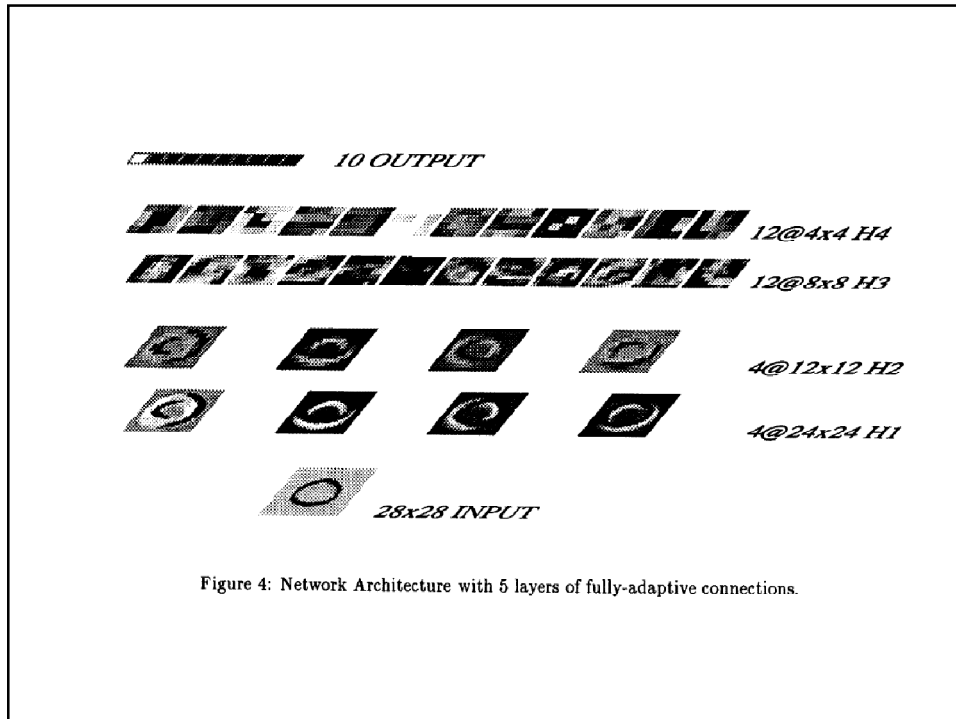
Input and output

- Input to network: 28x28 normalized image with binary-valued pixels. 784 input units.
- Output units: 10 units, one per class
- Target output: $\approx +1$ at correct unit, ≈ -1 at all other units

Network structure

- Problem with fully connected (“unstructured”) network:
 - If too few hidden units, function can’t be learned
 - If too many hidden units, overfitting will occur (**why?**)
- Their solution: Use prior knowledge about shape recognition to structure network
 - Shape recognition often works by detecting and combining local higher-level features (e.g., vertical bar + 3 horizontal bars)
 - Same high-level features can appear at different locations in different characters
 - Distorted characters might have higher-level features slightly displaced

- Each layer is “locally connected” to next layer, except for output layer. Notion of “receptive field” of a group of units
- Network uses “feature maps”.



Results

- After 30 epochs of training, error on
 - training set: 1.1%
 - test set: 3.4%
- All errors were on handwritten characters
- Performance was improved using higher-resolution input images.
- “A complete training session (30 passes through the training set plus performance measure on the test set) tasks about 3 days on a Sun SPARCstation 1.... After training, a recognition is performed in about 1.5 seconds on a Sun 3.”
- They implemented final network on a DSP board and reduced recognition time to 90 ms. Most of this time was spent normalizing the image size.

Heuristics for avoiding local minima in weight space

- Use stochastic gradient descent rather than true gradient descent. There is a different “error surface” for each training example. This adds “noise” to search process which can prevent it from getting stuck in local optima.
- Train multiple networks using the same data, but start each one with different random weights. If different weight vectors are found after training, select best one using separate validation set, or use ensemble of networks to vote on correct classification.
- Add “momentum” term to weight update rule.
- Simulated annealing
- Boosting

Learning rate and momentum

- Recall that stochastic gradient descent approaches “true” gradient descent as learning rate $\eta \rightarrow 0$.
- For practical purposes: choose η large enough so that learning is efficient, but small enough that a good approximation to true gradient descent is maintained, and oscillations are not reached.
- To avoid oscillations at large η , introduce *momentum*, in which change in weight is dependent on past weight change:

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1)$$

where n is the iteration through the main loop of backpropagation

The idea is to keep weight changes moving in the same direction.

Hypothesis Space and reachability

- What is the hypothesis space of a given neural network?
- Are all target functions reachable by back-propagation?

Regularization (“Alternative Error Functions”)

- “Regularization” means to modify the error function to constrain the network in various ways.
- Examples:
 - Add a penalty term for the magnitude of the weight vector (to decrease overfitting):

$$E(\mathbf{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

Other topics

- Other methods for training the weights
- Recurrent networks
- Dynamically modifying network structure