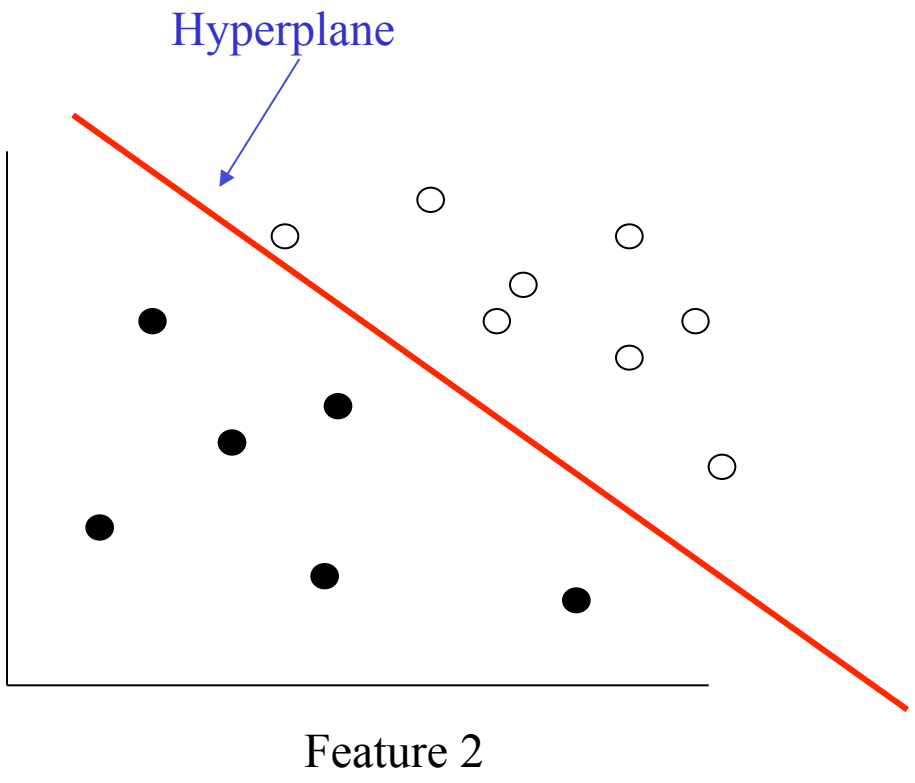
Multilayer Neural Networks

(sometimes called “Multilayer Perceptrons” or MLPs)

Linear separability



In 2D:

$$w_1 x_1 + w_2 x_2 + w_0 = 0$$

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2}$$

A perceptron can separate data that is linearly separable.

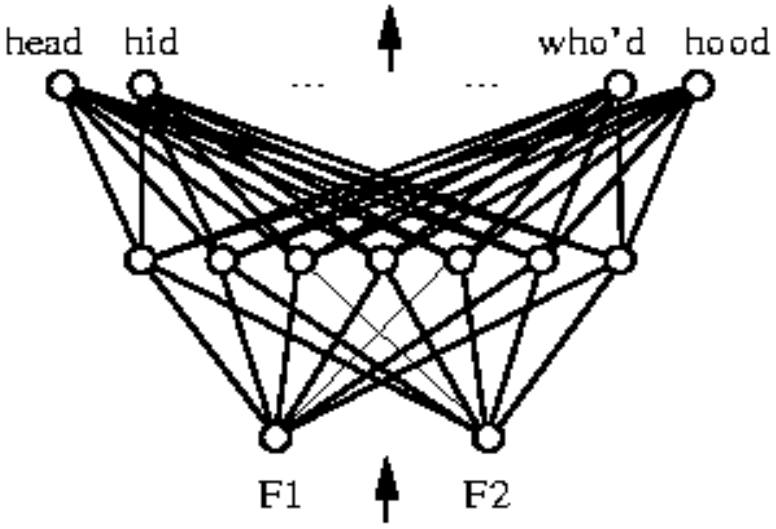
A bit of history

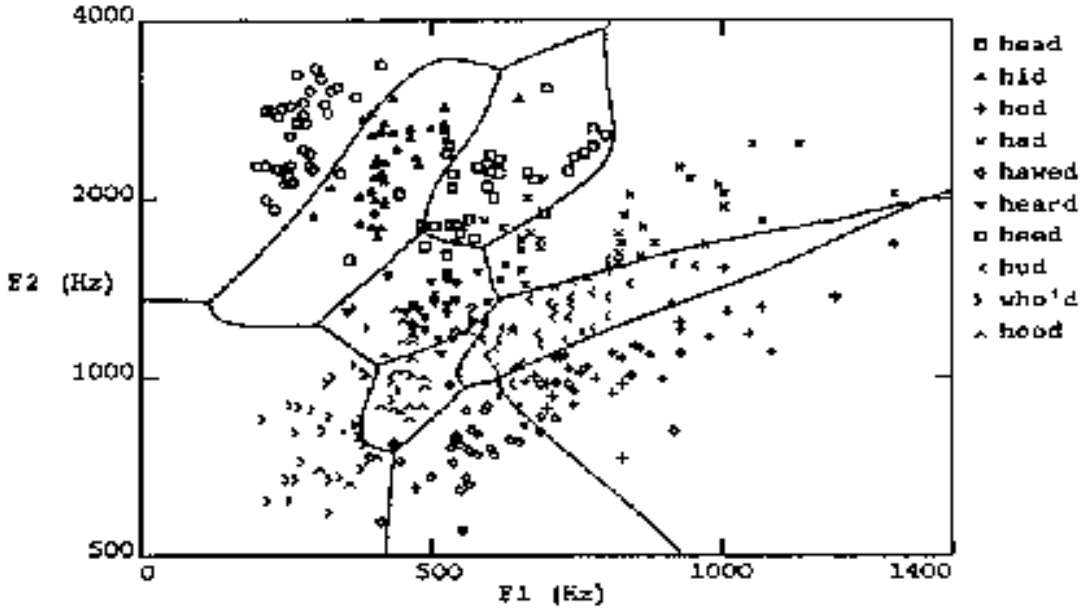
- 1960s: Rosenblatt proved that the perceptron learning rule converges to correct weights in a finite number of steps, provided the training examples are linearly separable.

- 1969: Minsky and Papert proved that perceptrons cannot represent non-linearly separable target functions.

- However, they proved that any transformation can be carried out by adding a fully connected hidden layer.
  - I.e., Multi-layer neural networks can represent non-linear decision surfaces

Multi-layer neural network example





Decision regions of a multilayer feedforward network. (From T. M. Mitchell, *Machine Learning)*

The network was trained to recognize 1 of 10 vowel sounds occurring in the context “h_d” (e.g., “had”, “hid”)

The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound.

The 10 network outputs correspond to the 10 possible vowel sounds.

- **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).

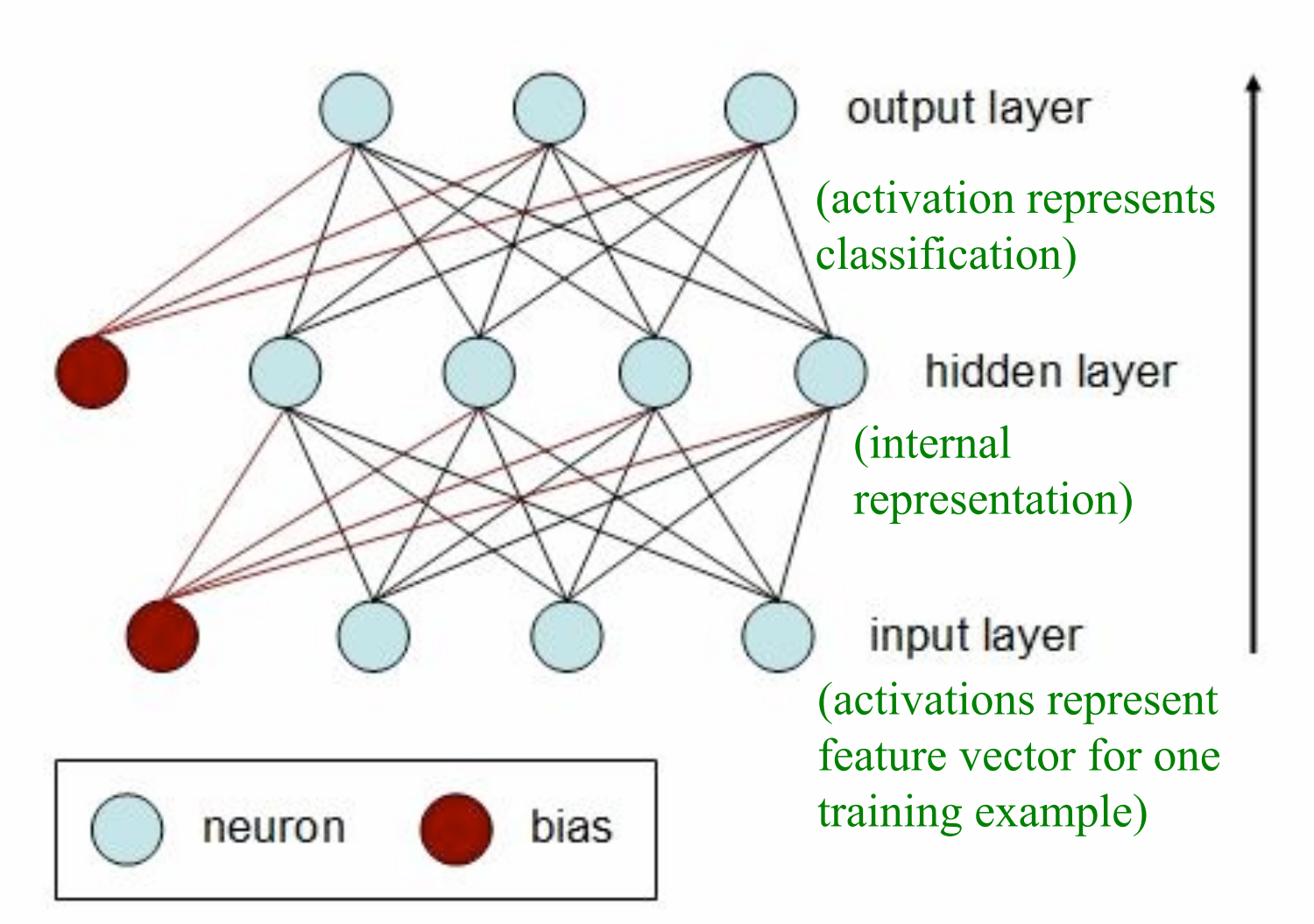
- **Bad news:** However, in multi-layer networks, there is no guarantee of convergence to minimal error weight vector.

But in practice, multi-layer networks often work very well.

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.

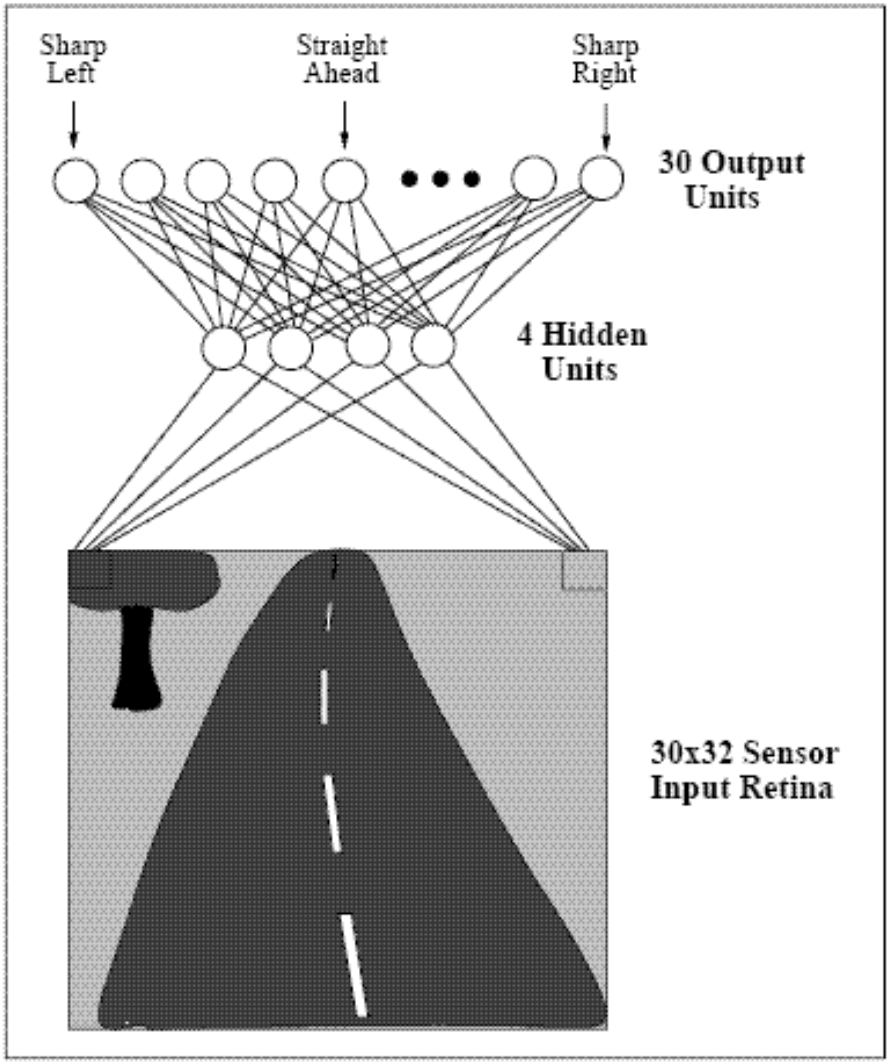
A “two”-layer neural network



Example: 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



(Note: bias unit and weights not shown)

Each output unit correspond to a particular steering direction. The most highly activated one gives the direction to steer.

Activation functions

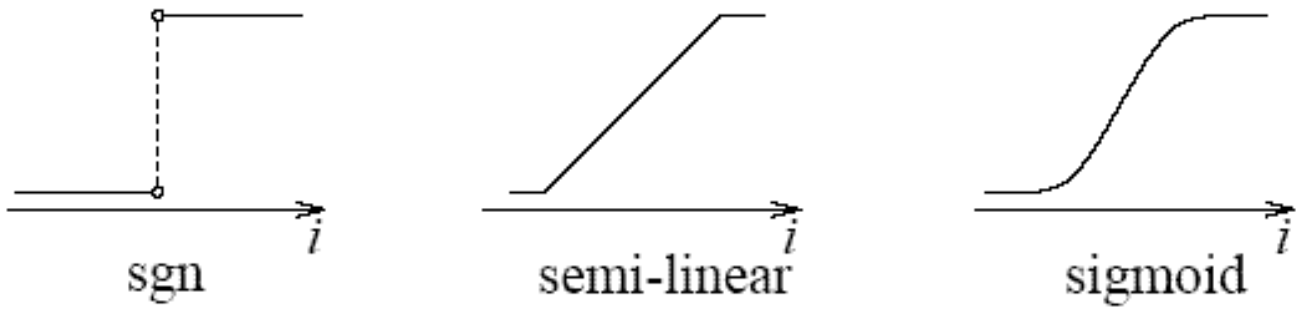
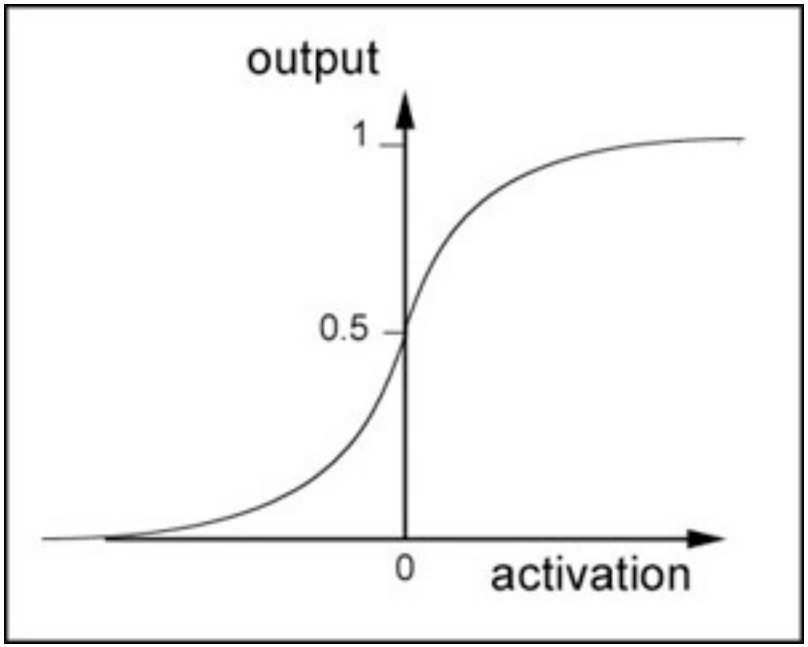


Figure 2.2: Various activation functions for a unit.

- Advantages of sigmoid function: nonlinear, differentiable, has real-valued outputs, and approximates the sgn function.

Sigmoid activation function:

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



$\mathbf{w} \cdot \mathbf{x}$	$\sigma(\mathbf{w} \cdot \mathbf{x})$
-2	.12
-1.5	.18
-1	.27
-.5	.38
0	.50
.5	.62
1	.73
1.5	.82
2	.88

- The derivative of the sigmoid activation function is easily expressed in terms of the function itself:

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

This is useful in deriving the back-propagation algorithm.

$$\sigma(z) = \frac{1}{1+e^{-z}} = (1+e^{-z})^{-1}$$

$$\frac{d\sigma}{dz} = -1(1+e^{-z})^{-2}\frac{d}{dz}(1+e^{-z})$$

$$= -\frac{1}{(1+e^{-z})^2}\left(-e^{-z}\right)$$

$$= \frac{e^{-z}}{(1+e^{-z})^2}$$

$$\sigma(z)\cdot(1-\sigma(z))$$

$$= \left(\frac{1}{1+e^{-z}}\right)\left(1-\left(\frac{1}{1+e^{-z}}\right)\right)$$

$$= \left(\frac{1}{1+e^{-z}}\right)-\left(\frac{1}{1+e^{-z}}\right)^2$$

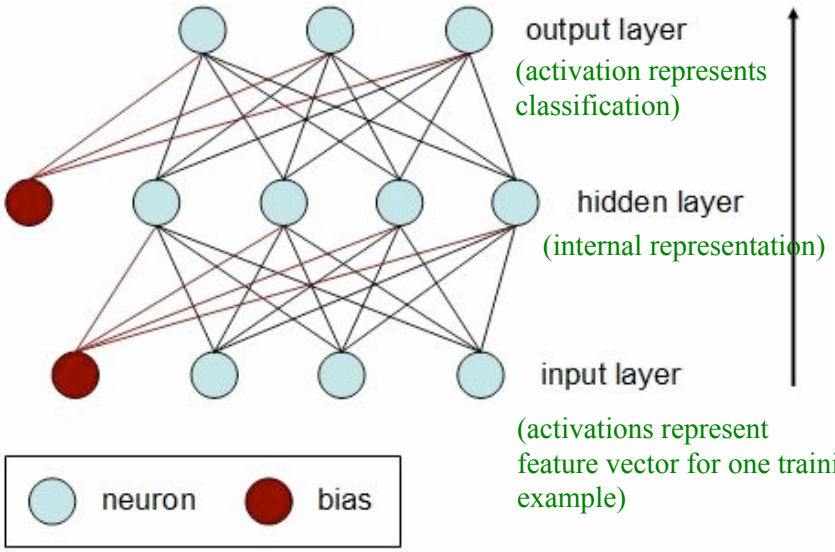
$$= \left(\frac{1}{1+e^{-z}}\right)-\left(\frac{1}{(1+e^{-z})^2}\right)$$

$$= \left(\frac{1+e^{-z}}{(1+e^{-z})^2}\right)-\left(\frac{1}{(1+e^{-z})^2}\right)$$

$$= \frac{e^{-z}}{(1+e^{-z})^2}$$

QED.

Neural network notation



x_i : activation of **input** node i.

h_j : activation of **hidden** node j.

o_k : activation of **output** node k.

w_{ji} : weight from node i to node j.

σ : sigmoid function.

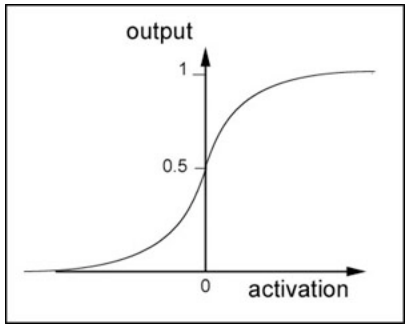
For each node j in hidden layer,

$$h_j = \sigma\left(\sum_{i \in \text{input layer}} w_{ji} x_i + w_{j0}\right)$$

For each node k in output layer,

$$o_k = \sigma\left(\sum_{j \in \text{hidden layer}} w_{kj} h_j + w_{k0}\right)$$

Sigmoid function:



Classification with a two-layer neural network ("Forward propagation")

Assume two-layer networks (i.e., one hidden layer):

I. For each test 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. Interpret the output layer as a classification.

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

Training a multi-layer neural network

Repeat for a given number of epochs or until accuracy on training data is acceptable:

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 *E*.
5. Run the back-propagation algorithm to update all weights in the network.

Training a multilayer neural network with back-propagation (stochastic gradient descent)

- Suppose training example has form (**x**, **t**)

  (i.e., both input and target are vectors).

- Error (or "loss") E is sum-squared error over all output units:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{k \in output\ layer} (t_k - o_k)^2$$

- Goal of learning is to minimize the mean sum-squared error over the training set.

I’m not going to derive the back-propagation equations here, but you can find derivations in the optional reading (or in many other places online).

Here, I’ll just give the actual algorithm.

Training algorithm

- Initialize the network weights **w** to small random numbers (e.g., between -0.05 and +0.05).

- Until the termination condition is met, Do:
  - For each (**x**,**t**) $\in$ training set, Do:
    1. *Propagate the input forward:*
       - Input **x** to the network and compute the activation h_j of each hidden unit j.
       - Compute the activation o_k of each output unit k.

2. *Calculate error terms*

- 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 j, calculate error term δ_j :

$$\delta_j \leftarrow h_j(1-h_j)\left(\sum_{k \in \text{output units}} w_{kj}\ \delta_k\right)$$

3. Update weights

- For each weight w_{kj} from the hidden to output layer :

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

where

$$\Delta w_{kj} = \eta \delta_k h_j$$

- For each weight w_{ji} from the input to hidden layer :

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

where

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

Step-by-step back-propagation example (and other resources)

http://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

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:

$$\Delta w_{ji}^{t} = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}^{t-1}$$

for all network weights w_{ji}, where t is the iteration through the main loop of back-propagation. α is a parameter between 0 and 1.

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

Example: Face recognition

(From T. M. Mitchell, *Machine Learning*, Chapter 4)

Code (C) and data at http://www.cs.cmu.edu/~tom/faces.html

- **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.

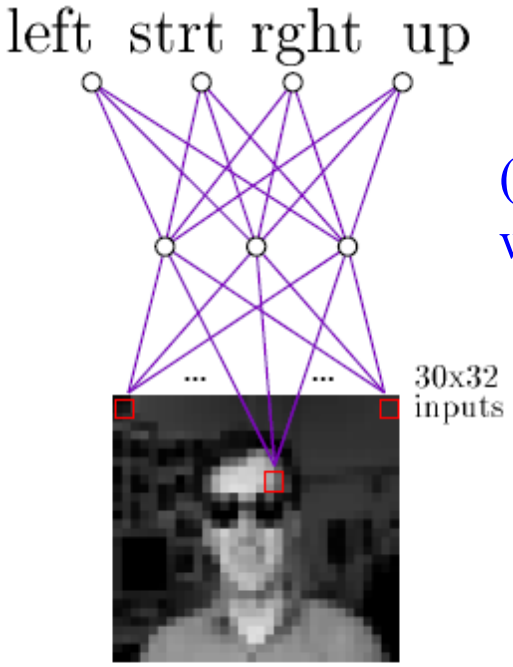
an2i_left_angry_open_4

an2i_right_sad_sunglasses_4

glickman_left_angry_open_4

Design Choices

- Input encoding
- Output encoding
- Network topology
- Learning rate
- Momentum



(Note: bias unit and weights not shown)

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 30x32 = 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".

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

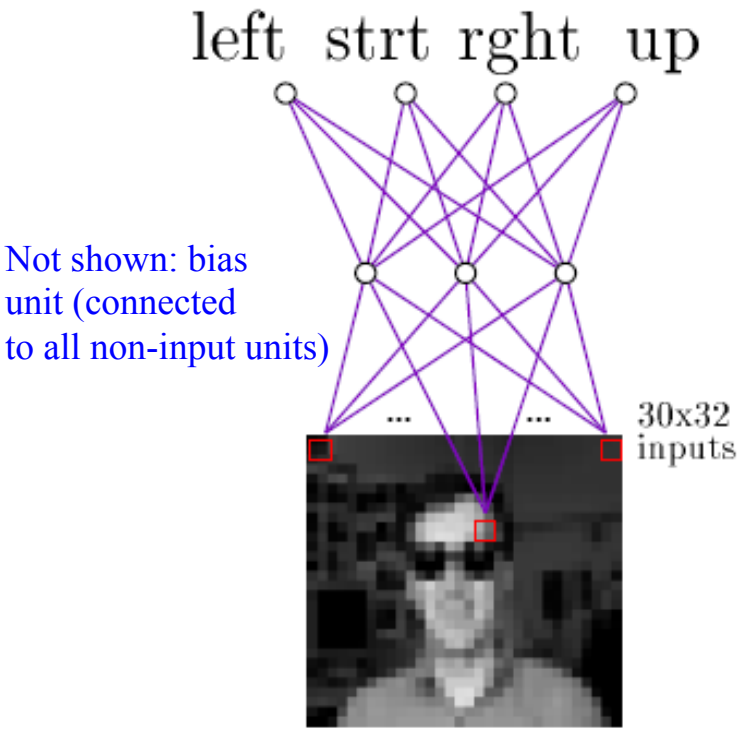
- Learning rate $\eta = 0.3$
- Momentum $\alpha = 0.3$
- If these are set too high, training fails to converge on network with acceptable error over training set.
- If these are set too low, training takes much longer.

Training

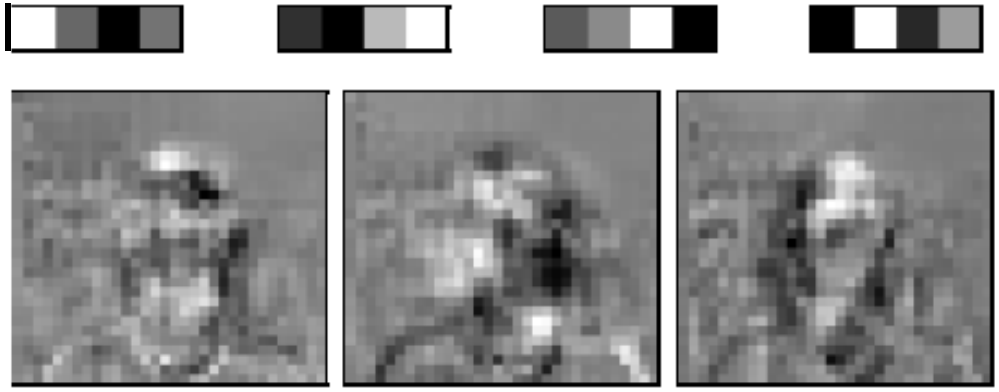
- For maximum number of epochs:
  - For each training example
    - Input photo to network
    - Propagate activations to output units
    - Determine error in output units
    - Adjust weights using back-propagation algorithm

- Demo of code

(from http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/faces.html)



Weights from each hidden unit to four output units



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

After 100 epochs of training.
(From T. M. Mitchell, Machine Learning)

- 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.

Understanding weight values

- After training:
  - Weights from input to hidden layer: high positive in certain facial regions
  - "Right" output unit has strong positive from second hidden unit, strong negative from third hidden unit.
    - Second hidden unit has positive weights on right side of face (aligns with bright skin of person turned to right) and negative weights on top of head (aligns with dark hair of person turned to right)
    - Third hidden unit has negative weights on right side of face, so will output value close to zero for person turned to right.

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.

Multi-layer networks are prone to overfitting

Don’t run too many epochs.

Use validation set instead of training set for testing after each epoch. Keep weights for best performing network on validation set.

(Can also use cross-validation.)

Then train on full training set using this many epochs.

Hidden Units

- Two few – can’t represent target function
- Too many – leads to overfitting

Use cross-validation to decide number of hidden units

Weight decay

- Modify error function to add a penalty for magnitude of weight vector, to decrease overfitting.

- This modifies weight update formula (with momemtum) to:

$$\Delta w_{ji}^{t} = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}^{t-1} - \lambda w_{ji}^{t-1}$$

where λ is a parameter between 0 and 1.

Many other topics I’m not covering

E.g.,

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