Neural Networks

(Multi-Layer Perceptrons)

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.

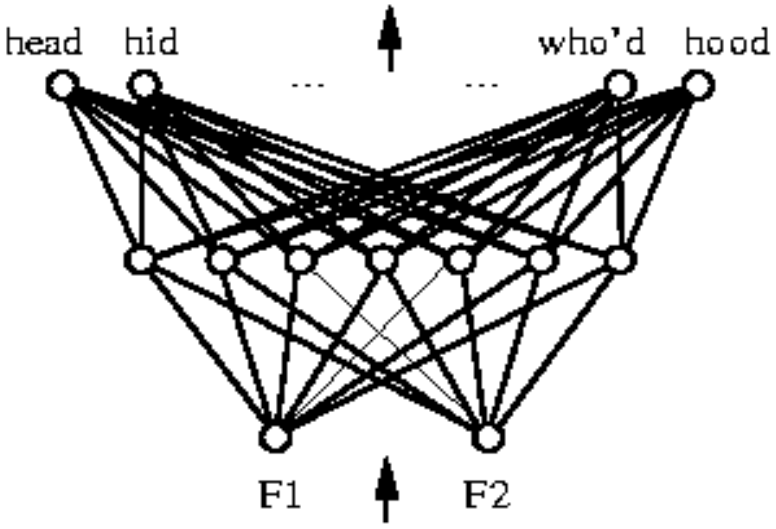
- However, they proved that any transformation can be carried out by adding a fully connected hidden layer.

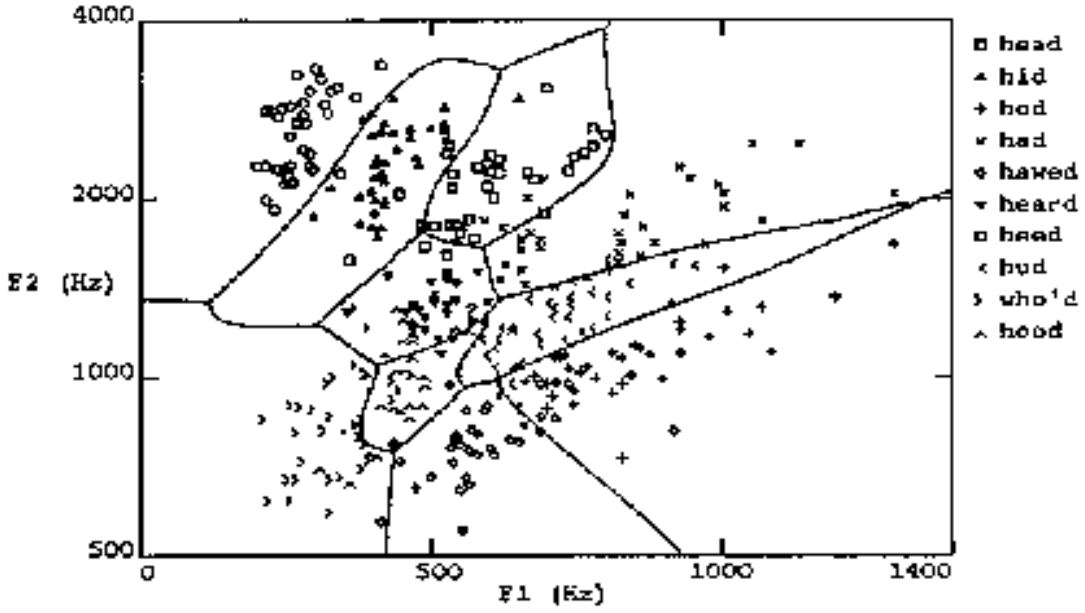
Multi-layer Perceptrons (MLPs)

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

- Multi-layer perceptrons can represent non-linear decision surfaces

Multi-layer perceptron 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?

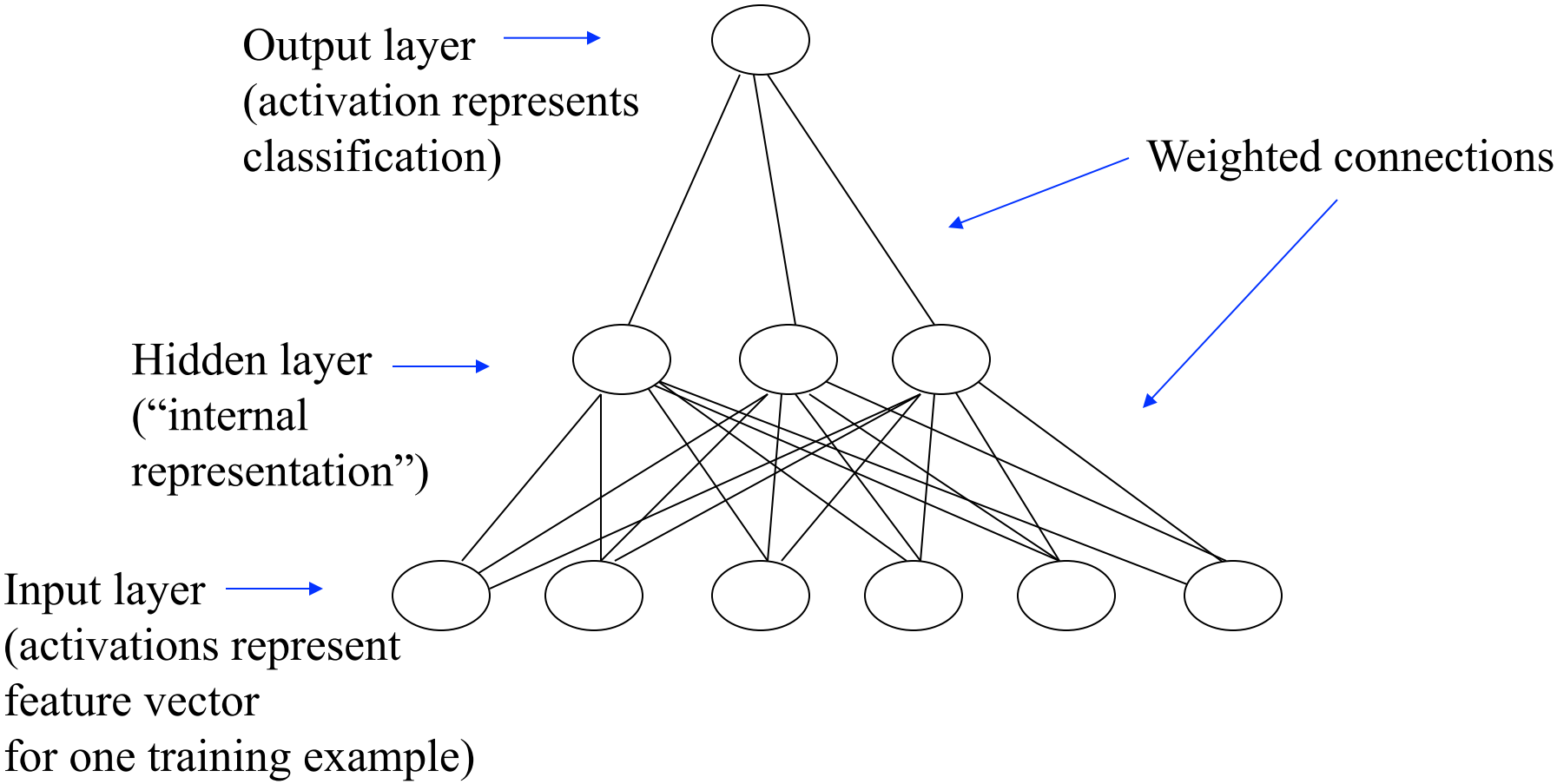
    2. How can such a network learn useful higher-order features?

- **Good news:** Successful credit-apportionment learning algorithms developed soon afterwards (e.g., back-propagation). 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.

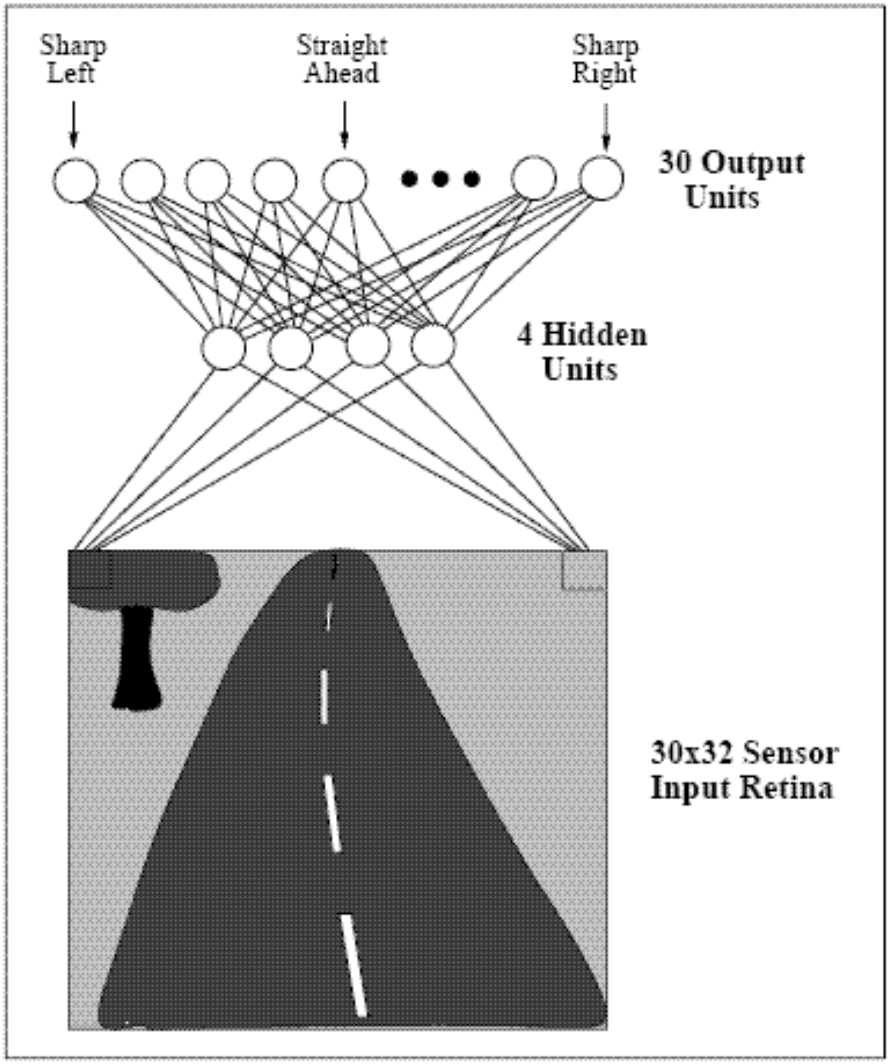
A two-layer neural network



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



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

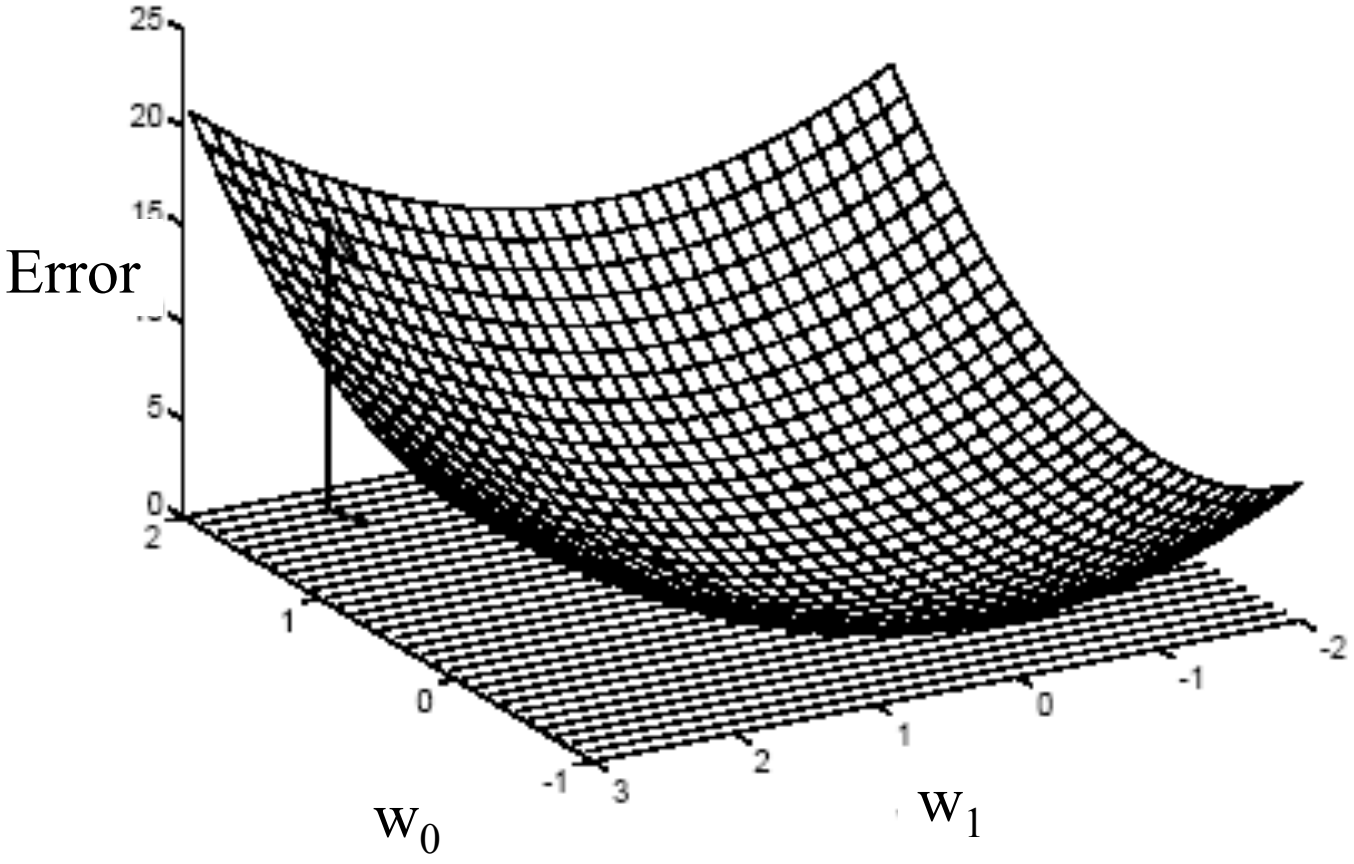
What kinds of problems are suitable for neural networks?

- 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

Learning in Multilayer Neural Networks: Gradient descent in weight space



From T. M. Mitchell, *Machine Learning*

Differentiable Threshold Units

- In general, in order to represent non-linear functions, need non-linear output function at each unit.

- In order to do gradient descent on weights, need differentiable error function (and thus differentiable output function).

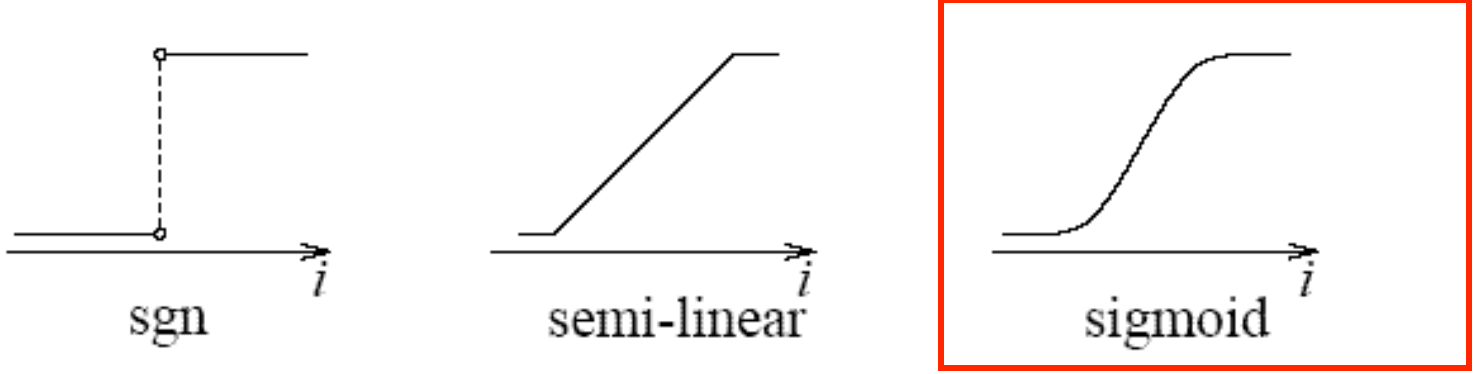


Figure 2.2: Various activation functions for a unit.

Sigmoid activation function:

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

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:

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

This is useful in deriving the back-propagation algorithm.

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.

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.

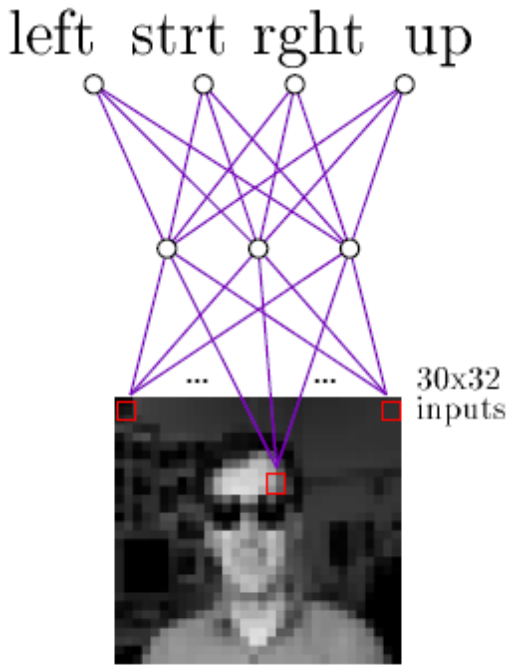
an2i_left_angry_open_4

an2i_right_sad_sunglasses_4

glickman_left_angry_open_4

Design Choices

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



Typical input images

- Preprocessing of photo:
  - Create 30x32 coarse resolution version of 120x128 image
  - This makes size of neural network more manageable

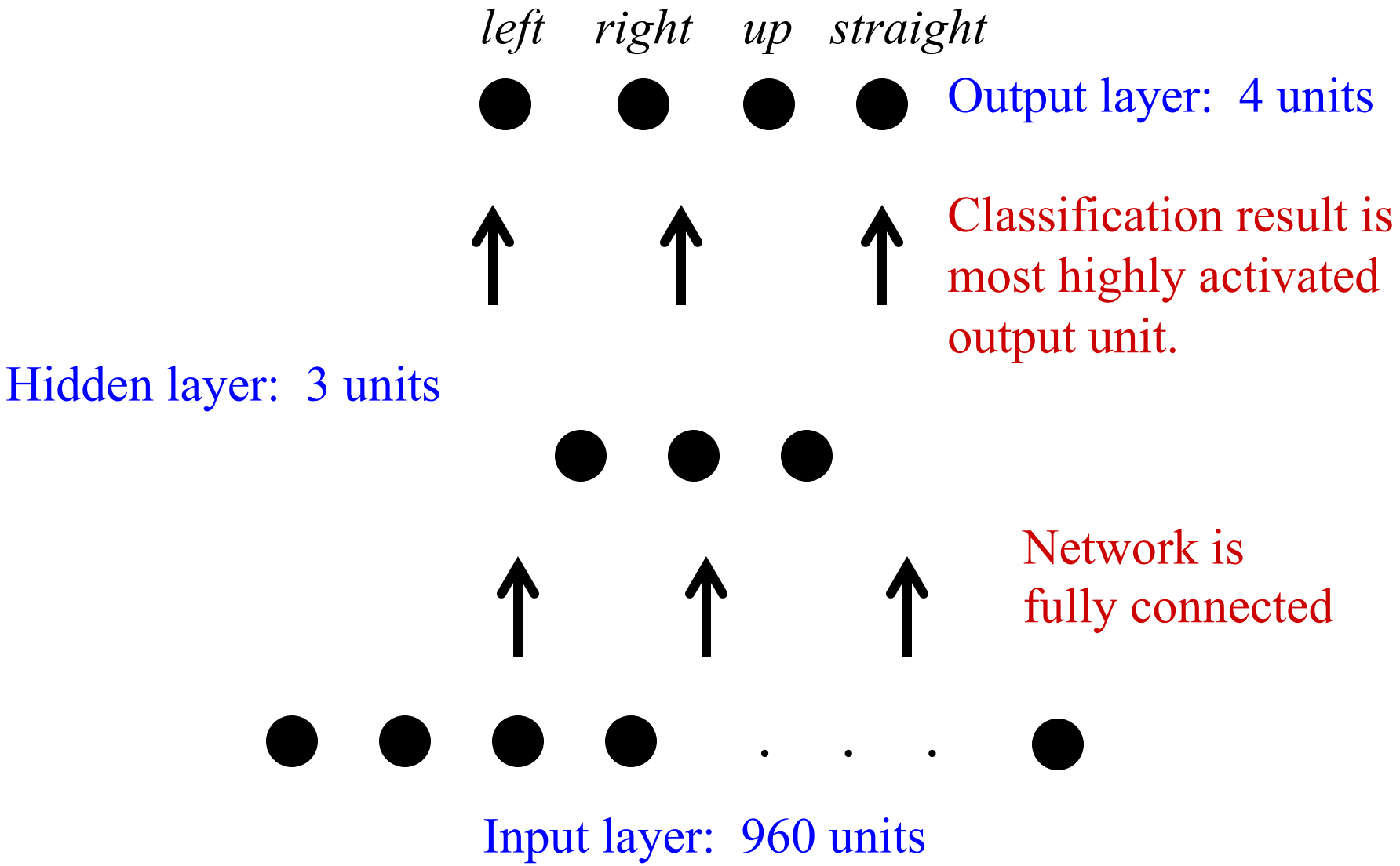
- Input to neural network:
  - Photo is encoded as 30x32 = 960 pixel intensity values, scaled to be in [0,1]
  - One input unit per pixel

- 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".

Network architecture



Target function

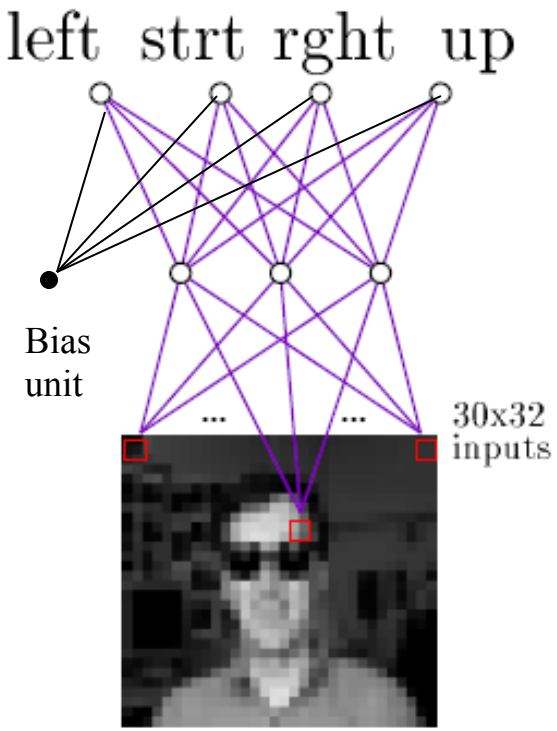
- Target function is:
  - Output unit should have activation 0.9 if it corresponds to correct classification
  - Otherwise output unit should have activation 0.1
- Use these values instead of 1 and 0, since sigmoid units can't produce 1 and 0 activation.

Other parameters

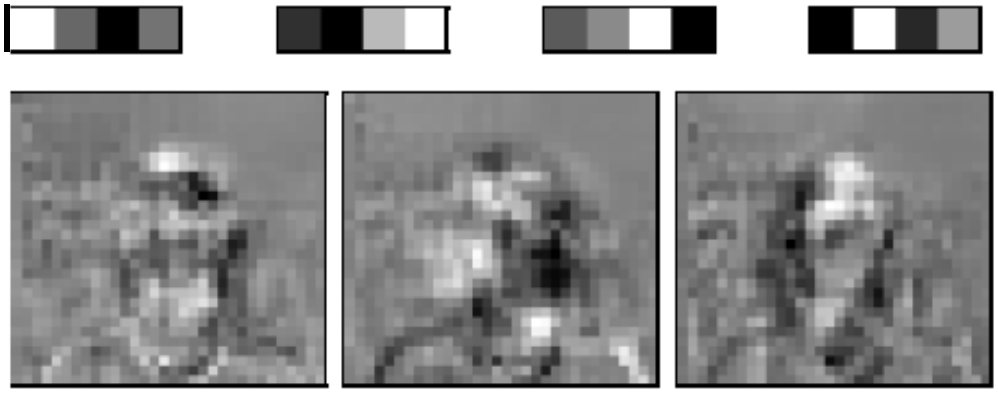
- Learning rate $\eta = 0.3$
- Momentum $\alpha = 0.3$
- If these are set too high, training fails to converge on network with acceptable error over training set.
- If these are set too low, training takes much longer.

Training

- 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.
- Demo of code



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.

- In most applications, two-layer network with sigmoid activation function used.

Gradient descent

- We want to find **w** so as to minimize *sum-squared error*:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{k=0}^{m}(t^k - o^k)^2$$

(recall that k is an index over training examples, not a power)

$$\nabla E(\mathbf{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n}\right]$$

- To minimize, take the derivative of $E(\mathbf{w})$ with respect to **w**.

- A vector derivative is called a "gradient": $\nabla E(\mathbf{w})$

- Here is how we change each weight:

$$w_j \leftarrow w_j + \Delta w_j$$

where

$$\Delta w_j = -\eta \frac{\partial E}{\partial w_j}$$

and η is the *learning rate*.

- Error function

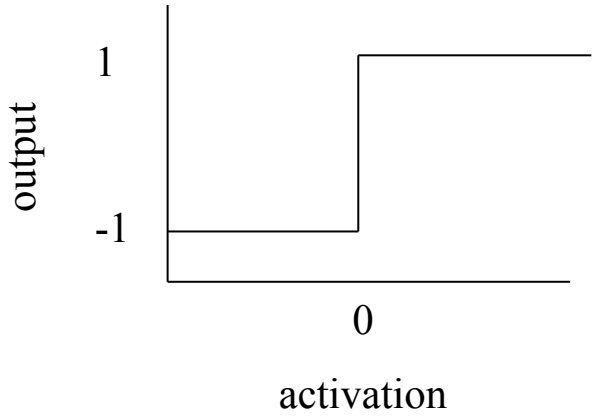
$$E(\mathbf{w}) = \frac{1}{2}\sum_{k=0}^{m}(t^k - o^k)^2$$

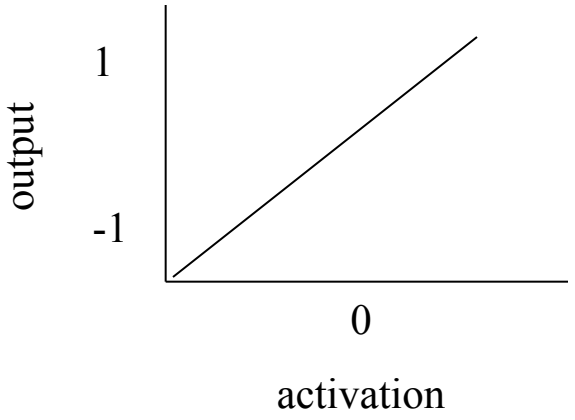
has to be differentiable, so *output function o* also has to be differentiable.

- Simplest case: A linear unit:

$$o = w_0 + w_1 x_1 + \ldots + w_n x_n = \sum_{j=1}^{n} w_j x_j + w_0$$

Activation functions





$$o = \text{sgn}\left(\sum_j w_j x_j + w_0\right)$$

Not differentiable

$$o = \sum_j w_j x_j + w_0$$

Differentiable

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_k (t^k - o^k)^2 \qquad (1)$$

$$= \frac{1}{2}\sum_k \frac{\partial}{\partial w_i}(t^k - o^k)^2 \qquad (2)$$

$$= \frac{1}{2}\sum_k 2(t^k - o^k)\frac{\partial}{\partial w_i}(t^k - o^k) \qquad (3)$$

$$= \sum_k (t^k - o^k)\frac{\partial}{\partial w_i}(t^k - \mathbf{w}\cdot \mathbf{x}^k) \qquad (4)$$

$$= \sum_k (t^k - o^k)(-x_i^k) \qquad (5)$$

So,

$$\Delta w_i = \eta \sum_k (t^k - o^k) x_i^k \qquad (6)$$

This is called the *perceptron learning rule,* with "***true gradient descent***".

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:

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

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 (**x**, **t**), where **x** is a vector of n_{in} network input values and **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 **w** to small random numbers (e.g., between -0.05 and +0.05).

- Until the termination condition is met, Do:
  - For each (**x**,**t**) ∈ S, Do:

    1. *Propagate the input forward:*
       - Input **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

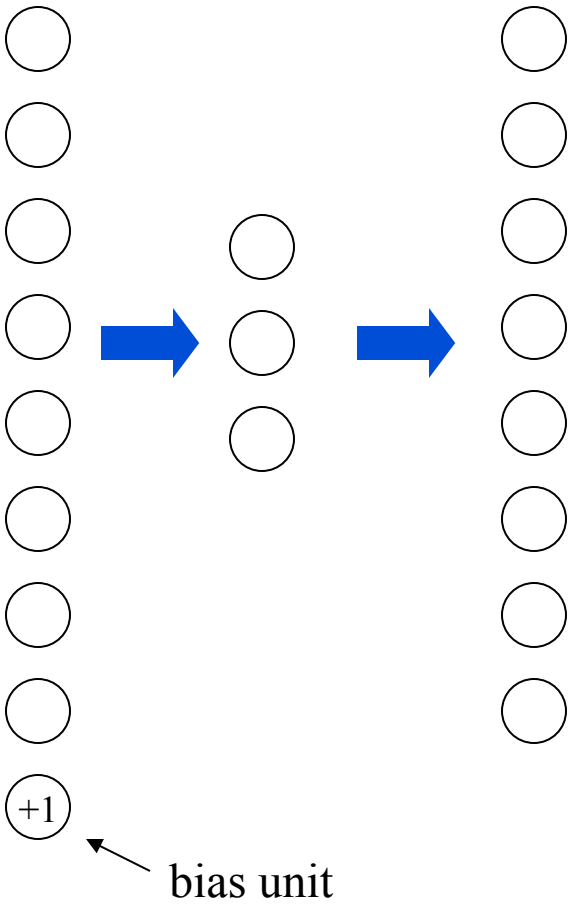
$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

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?

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

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

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.

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):

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

Many other topics I’m not covering

E.g.,

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