MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.034—Artificial Intelligence
Recitation 10, Friday, November 16, 2001

Agenda

1. Summary sheet for What to Know
2. What's in a Neural Net
3. Gradient Descent
4. The Backpropagation Algorithm
5. Training Neural Nets
6. Classification vs. Regression

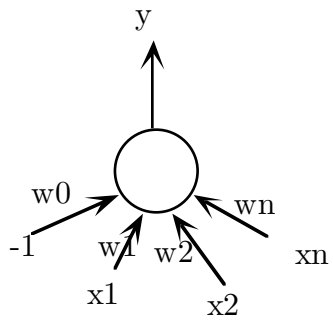
What to Know

```
1       Nearest Neighbors
        a.      Single nearest neighbor boundaries
        b.      Multiple nearest neighbor smoothing
        c.      Scaling and normalizing the data
        d.      Operating with partially labeled data
2       ID Trees
        a.      Calculating entropy for a collection
        b.      Choosing useful features
        c.      Building an ID tree to classify data
        d.      Overfitting and cross-validation
3       Neural Networks
        a.      Forward equation (sum of weighted inputs, through threshold or sigmoid)
        b.      Threshold learning for a simple perceptron
        c.      Sigmoid learning for an output node
        d.      Sigmoid learning for a hidden node
        e.      Learning rate, overshooting
4       Genetic Algorithms
        a.      Steps of the basic GA
        b.      Representation of an individual
        c.      Mutation & Crossover
        d.      Fitness: proportional & rank selection
```

Perceptrons

A perceptron is a single-layer neural net. The output $y = 1$ if the weighted sum on the inputs is greater than 0; it is 0 otherwise. (The idea is to simulate a single neuron.)

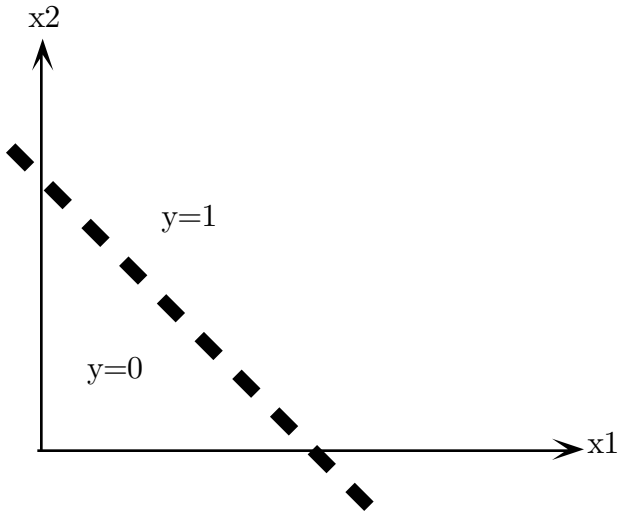
The picture looks like this:



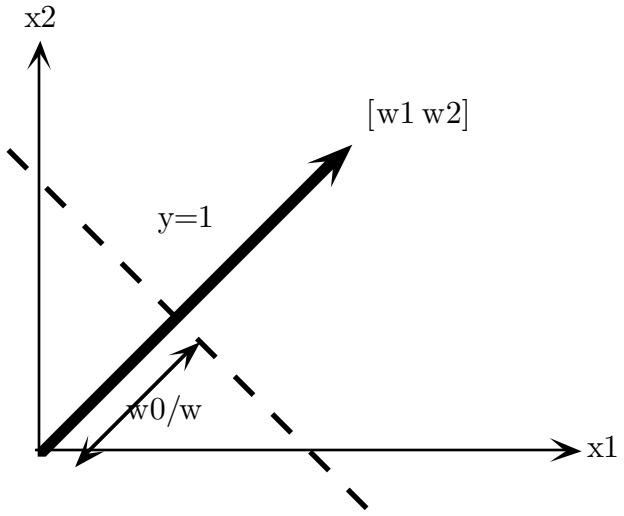
We can understand this mathematically as the equation of a line in a plane (hyperplane). For two variables,

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

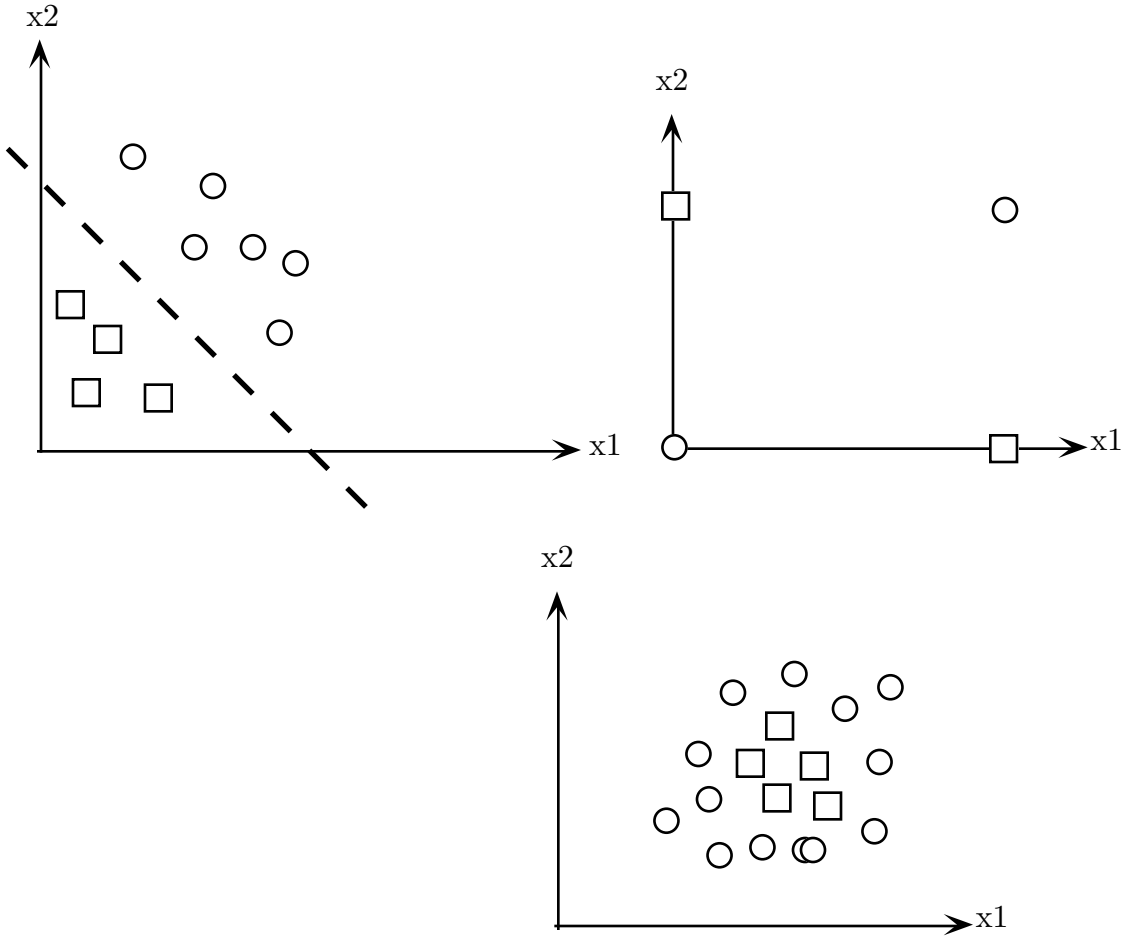
The equation in the plane looks like this:



where x_i are the input variables and w_i are the weights. Alternatively, as a matrix equation, we have that the dot product of $[x_1 s_2]$ and $[w_1 w_2]$ is equal to the constant 0. This is true of the points on the line perpendicular to the vector $[w_1 w_2]$. If we let w be the length of the vector $[w1w2]$, i.e., $\sqrt{w_1^2 + w_2^2}$, then we can write this as:



We can use a single layer net (perceptron) to make decision boundaries that are *linearly separable*:



Perceptron Learning rule

Because outputs are directly tied to inputs, the perceptron learning rule is easy: change the weights of inputs that are non-zero in the direction of the error, where the *error* is the difference between the true output and the output the perceptron with the current weight gives. (If the error is negative, i.e., the weight is too large for that example, then lower the weight; if the error is positive, then raise the weight. Ignore weights that are zero — they are contributing nothing.)

The learning equation to change the weights is:

$$\Delta w = \alpha(y^* - y)x$$

where y^* is the *correct* output, y is the current output with the current weights, and α is the learning rate (amount of allowed change in weights, usually small so we don't overshoot). Note that the difference between the correct and computed values can only be $-1, 0, +1$ in this case.

This method is guaranteed to converge for linearly separable problems, and is chaotic otherwise.

Representing line equations: we use $ax + by - c = 0$ since that lets us represent vertical lines.

What's In a Neural Net

We stack the layers of a perceptron. The lower layers transform the input problem into more tractable problems for later layers — now can learn non-half-space regions (as with XOR). Note that if we can draw TWO lines, then that lets us easily classify XOR.

The first inputs are as before, the next units, since they are neither input nor output units, are called *hidden* units.

For the XOR example, one solution with the net shown is to have,

$$\begin{array}{lll} w_{10} = \frac{3}{2} & w_{11} = 1 & w_{12} = 1 \\ w_{20} = \frac{1}{2} & w_{21} = 1 & w_{22} = 1 \\ w_{30} = \frac{1}{2} & w_{31} = -1 & w_{32} = 1 \end{array}$$

x_1	x_2	O_1	O_2	y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Computational Power

If we let the weights on the units be arbitrary real numbers, then a 2-layer neural net can compute *any computable function*, i.e., equivalent to any of your favorite programming languages (except those developed by Microsoft).

In the simpler case of a neural net where all the hidden units are perceptrons, i.e., the outputs are 0 or 1 rather than 'smoothed' by the sigmoid function, there is an easier way to see how to

Learning: Backpropagation

However, there is no good learning rule for a multi-layer perceptron, as there is for a single-layer perceptron.

Instead of using a discontinuous (step) function to relate input to output, we can use a smooth threshold function that approximates it, and then use *gradient descent* to improve the weights.

Sigmoid Threshold Units

A single threshold unit is like a perceptron with a "soft" threshold. On the perceptron, the decision boundary output is 1/2; far from this line, the output is 0 or 1. We let g denote the sigmoid threshold,

$$g(v) = \frac{1}{1-e^{-v}}$$
$$v = \Sigma_{i=0}^{n} w_i x_i$$

Training

Consider a multi-layer (actually, just the 2 layer) neural net. Let y be the output. Then

$$y = g(w_{31} \cdot (g(w_{11}x_1 + w_{12}x_2 - w_{10}) + w_{21} \cdot (g(w_{21}x_1 + w22x_2 - w_{20}) - w_{30})$$

Or,

$$y = F(w, x)$$

where x is a vector of inputs, and w is a vector of weights.

For a FIXED x_i vector, we have $y_i = F_i(w)$, and the DESIRED output is denoted y_i^*

For any weight vector w_i, denote the ERROR E over all the training set as,

$$E = \Sigma_i \frac{1}{2}(y_i^* - y_i)^2 = \Sigma_i \frac{1}{2}(y_i^* - F_i w))^2$$

Our GOAL is to find the weight vector that MINIMIZES the error E.

Minimization of Error by Gradient Descent

Basic Problem: Given an initial set of parameters w_o with error $E_o = G(w_o)$, find a (new) value of w that is a local minimum of the error of G. At a local minimum, no incremental change in w produces a change in $G(w_o)$, i.e., all the first derivatives are zero. (I.e., if we "wiggle" w a tiny bit, how much does the G change?

Approach:

1. Find the "slope" of G at vector w_o.
2. Move to w_1 which is "downslope" (make the stepsize proportional to the magnitude of the slope). (Note: gradient is n-dimensional slope, the direction where G changes the fastest.)

More formally:

$$w_{i+1} = w_i - \alpha \times \text{grad}G$$

where α (also called r) is some "small" step-size. If r is too big, we can jump over minima.

(2) Stop when either the (a) gradient is very small; or (b) the change in G is minimal.

Important: this is NOT guaranteed to find a *global* minimum of G!

Recall that the gradient is the n-dimensional "slope", the direction where G changes fastest.

An example:

$$E = G(w) = \frac{1}{2}(w_1^2 + w_2^2)$$

then

NB: for a nonlinear function, the gradient changes with "position"; the gradient vector is always perpendicular to contour lines.

The Backpropagation Algorithm

The Backpropagation Algorithm is an efficient method of doing on-line gradient descent for neural networks.

We need two rules:

- A **gradient descent rule**: what direction and how far to *move* at each step (what weights to change and how much)

  The change in a weight $w_{i,j}$ that goes FROM unit i TO unit j is simply the step size or learning rate r times the local gradient of the activation function at j (here, the sigmoid function) times the value of the current input y_i on the "line" from i to j, i.e., from unit $w_{i,j}$ to $w_{j,k}$.

- A **backpropagation rule**: a method that tells us how to change the *interior* unit weights, given a gradient descent step.

The algorithm runs as follows:

1. [Initialize] Initialize the weights to *small* random values (if big, causes saturation).
2. [Random start] Choose a sample input vector x_i
3. [Forward propagation] Compute the output that this input vector produces (the *activation* or value v_j at each unit, and the output for each unit, y_j (activation after being passed through a sigmoid).
4. [Start back propagation] Compute the gradient change or *delta* δ_n for each **output** layer unit n (O= the actual, true, observed output, and d is the computed output by forward propagation). ϕ' is the derivative of the sigmoid or activation function.

$$\begin{aligned}\delta_n &= \phi'(v_n) \times (O_n - d_n) \\ &= \frac{\partial g}{\partial v} \times (y_n^{\text{true}} - Y_n^{\text{computed}})\end{aligned}$$

5. Compute the gradient change δ for each *preceding* layer in the network, by backpropagation (going backwards, layer by layer):

$$\delta_j = \phi'(v_j)\Sigma_k f_k w_{j \Rightarrow k}$$

6. Compute the weight change by the gradient descent rule:

$$\Delta w_{i \Rightarrow j} = r\delta_j y_i$$

   Note: in the Winston book, we have $\delta_j = o_j(1 - o_j) \times \beta_j$; $o(1 - o)$ is the derivative of the sigmoid function. This product is $o_j(1 - o_j)$ is $\phi'(v_j)$.

7. Use this value to iterate back to the next layer

In practice, the backprop algorithm is very sensitive to the choice of the step size, r, otherwise known as the *learning rate*.

Training neural nets

Given a data set and desired outputs, a neural net with m weights, we want to find a setting of the weights that will give good predictive performance on new data (we also want to estimate the performance of the trained net on new data).

This is done by using a training set and a validation set, and a test set. The training set is used to pick the weights; the validation set is used to stop training; the test set is used to evaluate performance.

We first pick an initial set of weights, random and small.

Second, we perform interative minimization of error over the training set by either

(i) on-line training: present a sample x_i and y_i chosen randomly and change weights to reduce error on this sample; repeat

(ii) off-line training: change weights to reduce error using entire training set

Third, we stop when error on the *validation set* reaches a minimum — this is to avoid overfitting the training set.

Fourth, can repeat the above steps and get the best weight vector performance

Fifth — use this best weight vector to compute error on the test set — we do not repeat training to improve this value.

We can use cross-validation when we don't have enough data to split it into three groups.

Example

See attached.