**The Perceptron**

* The perceptron learning rule is a method for finding the weights in a network.
* We consider the problem of**supervised learning** for **classification** although other types of problems can also be solved.
* A nice feature of the perceptron learning rule is that*if there exist* a set of weights that solve the problem, then the perceptron will find these weights. This is true for either binary or bipolar representations.

**Assumptions:**

* We have single layer network whose output is, as before,

output = f(net) = f(W X)

where f is a binary step function f whose values are (+-1).

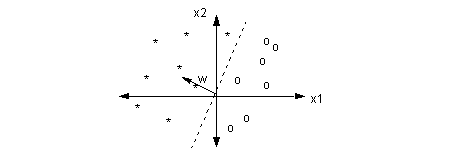
* We assume that the bias treated as just an extra input whose value is 1
* p = number of training examples (x,t) where t = +1 or -1

**Geometric Interpretation:**

With this binary function f, the problem reduces to finding weights such that

sign( W X) = t

That is, the weight must be chosen so that the projection of pattern X onto W has the same sign as the target t. But the boundary between positive and negative projections is just the plane W X = 0 , i.e. the same decision boundary we saw before.



**The Perceptron Algorithm**

1. initialize the weights (either to zero or to a small random value)
2. pick a learning rate m ( this is a number between 0 and 1)
3. Until stopping condition is satisfied (e.g. weights don't change):

For each training pattern (x, t):

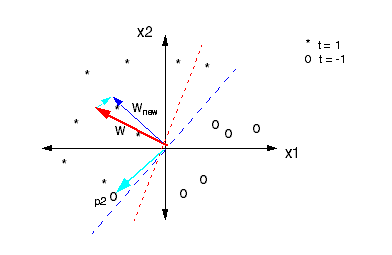
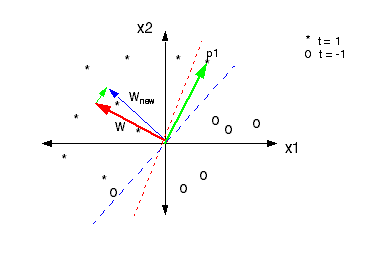
* compute output activation y = f(w x)
* If y = t, don't change weights
* If y != t, update the weights:

w(new) = w(old) + 2 m t x

or

w(new) = w(old) + m (t - y ) x, for all t

Consider wht happens below when the training pattern p1 or p2 is chosen. Before updating the weight W, we note that both p1 and p2 are incorrectly classified (red dashed line is decision boundary). Suppose we choose p1 to update the weights as in picture below on the left. P1 has target value t=1, so that the weight is moved a small amount in the direction of p1. Suppose we choose p2 to update the weights. P2 has target value t=-1 so the weight is moved a small amount in the direction of -p2. In either case, the new boundary (blue dashed line) is better than before.



**Comments on Perceptron**

* The choice of learning rate m does not matter because it just changes the scaling of w.
* The decision surface (for 2 inputs and one bias) has equation:

x2 = - (w1/w2) x1 - w3 / w2

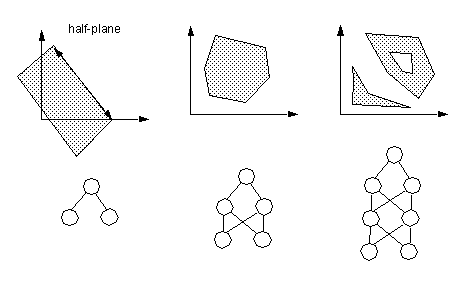
where we have defined w3 to be the bias: W = (w1,w2,b) = (w1,w2,w3)

* From this we see that the equation remains the same if W is scaled by a constant.

*The perceptron is guaranteed to converge in a finite number of steps if the problem is separable. May be unstable if the problem is not separable.*

Outline: Find a lower bound L(k) for |w|2 as a function of iteration k. Then find an upper bound U(k) for |w|2. Then show that the lower bound grows at a faster rate than the upper bound. Since the lower bound can't be larger than the upper bound, there must be a finite k such that the weight is no longer updated. However, this can only happen if all patterns are correctly classified.

**Perceptron Decision Boundaries**

Two Layer Net: The above is not the most general region. Here, we have assumed the top layer is an **AND** function.

**Problem**: In the general for the 2- and 3- layers cases, there is no simple way to determine the weights.

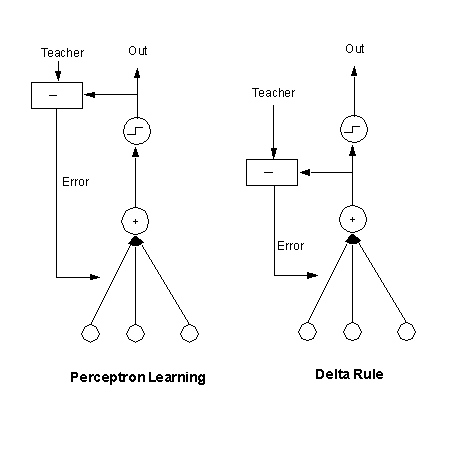
**Delta Rule**

**Also known by the names:**

* Adaline Rule
* Widrow-Hoff Rule
* Least Mean Squares (LMS) Rule

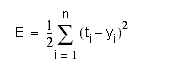
**Change from Perceptron:**

* Replace the step function in the with a continuous (differentiable) activation function, e.g linear
* For classification problems, use the step function only to determine the class and not to update the weights.
* Note: this is the same algorithm we saw for regression. All that really differs is how the classes are determined.



**Delta Rule: Training by Gradient Descent Revisited**

Construct a cost function E that measures how well the network has learned. For example

 (one output node)

where

n = number of examples

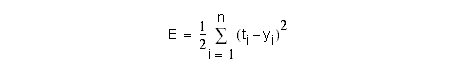
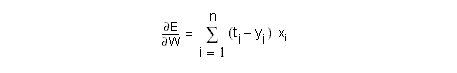
ti = desired target value associated with the i-th example

yi = output of network when the i-th input pattern is presented to network

* To train the network, we adjust the weights in the network so as to decrease the cost (this is where we require differentiability). This is called gradient descent.

**Algorithm**

* Initialize the weights with some small random value
* Until E is within desired tolerance, update the weights according to where E is evaluated at W(old), m is the learning rate.: and the gradient is

https://www.willamette.edu/~gorr/classes/cs449/Classification/delta-3.gif  
  


 **More than Two Classes.**

If there are mor ethan 2 classes we could still use the same network but instead of having a binary target, we can let the target take on discrete values. For example of there ar 5 classes, we could have t=1,2,3,4,5 or t= -2,-1,0,1,2. It turns out, however, that the network has a much easier time if we have one output for class. We can think of each output node as trying to solve a binary problem (it is either in the given class or it isn't).

