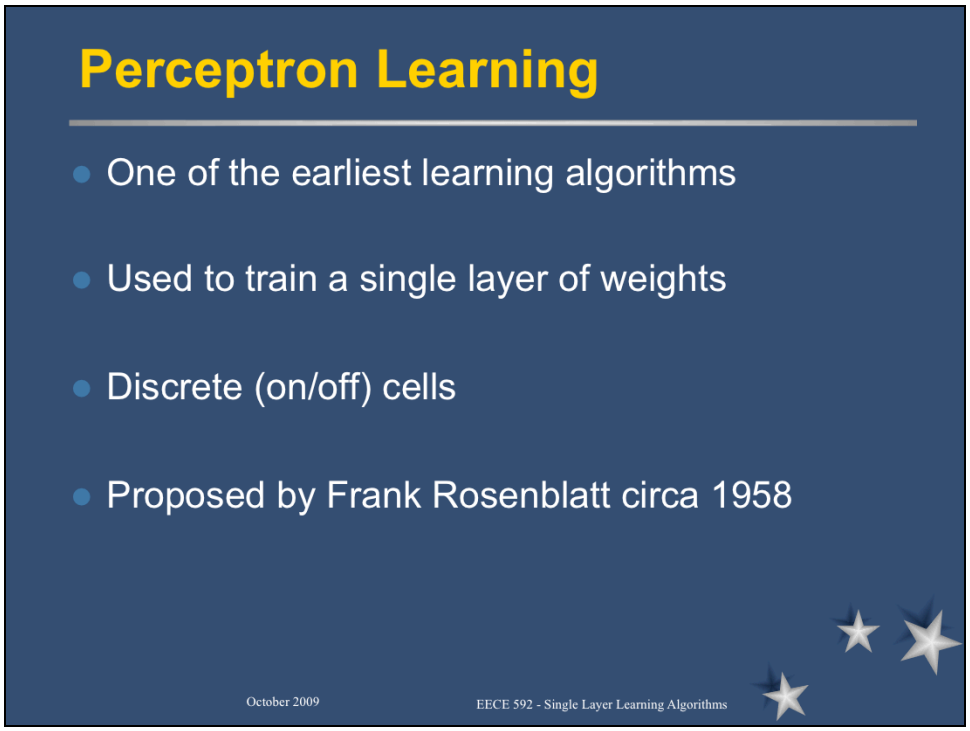

Single Layer Learning
Perceptrons & The Delta Rule

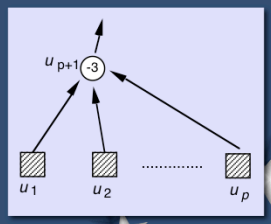



The perceptron learning algorithm is one of the first procedures proposed for learning in neural network models and is mostly credited to Rosenblatt.

The algorithm applies only to single layer models of discrete cells. That is, cells that have a step threshold function.

# Notation

- $u_i$ — $i^{th}$ input (not a neuron)
- $u_{p+1}$ — output neuron
- $p$ — number of inputs (length of input vector)
- $w_{p+1,\,i}$ — weight from input $i$ to output neuron
- $S$ — weighted sum computed by neuron
- $X^k$ — $k^{th}$ $p$ length input vector
- $C^k$ — $k^{th}$ desired output for $X^k$



## Notation cont.

- Weight vector of $u_{p+1}$ :

$$W = [w_{p+1,0}, w_{p+1,1}, w_{p+1,2} \ldots . w_{p+1,p}]$$

- Weighted sum $S = S_{p+1} = \sum_{j=0}^{p} w_j u_j$ where $w_j = w_{p+1,j}$

- Output of $u_{p+1}$ thresholded based upon sign of weighted sum.

October 2009 EECE 592 - Single Layer Learning Algorithms

For a single cell with $p$ inputs let the weights be defined by a vector:
$W = [w_{p+1,0}, w_{p+1,1}, w_{p+1,2} \ldots . w_{p+1,p}]$

for convenience, let
$w_j = w_{p+1,j}$

The weighted sum then becomes
$S = S_{p+1} = \sum_{j=0}^{p} w_j u_j$

The output of the unit $u_j$ is

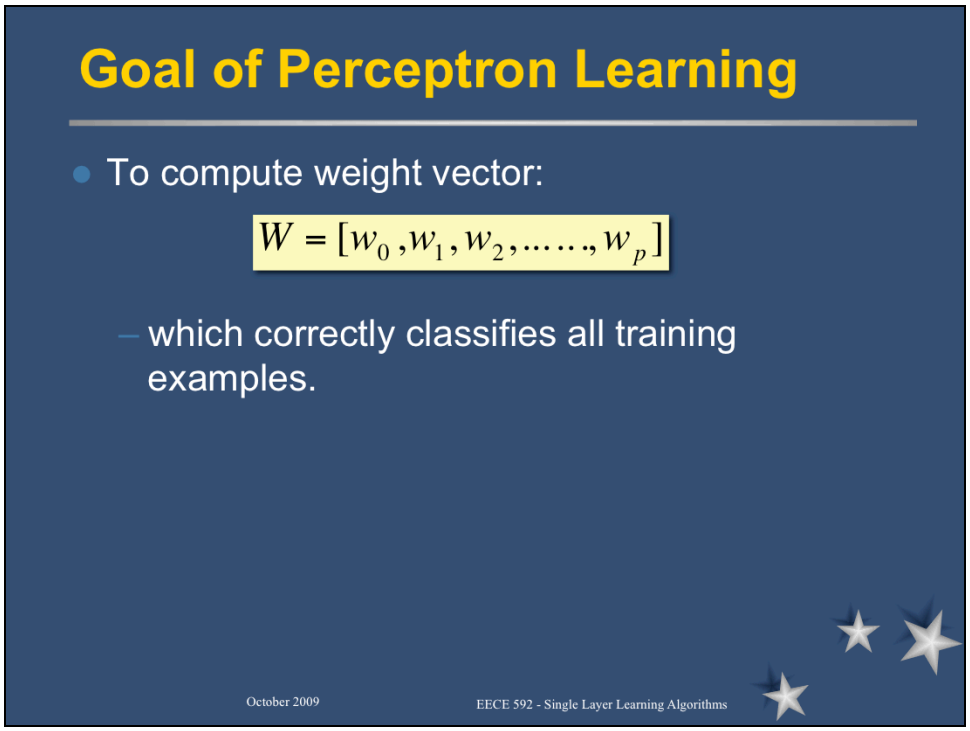
+1 if $S \geq 0$ and

-1 if $S < 0$.

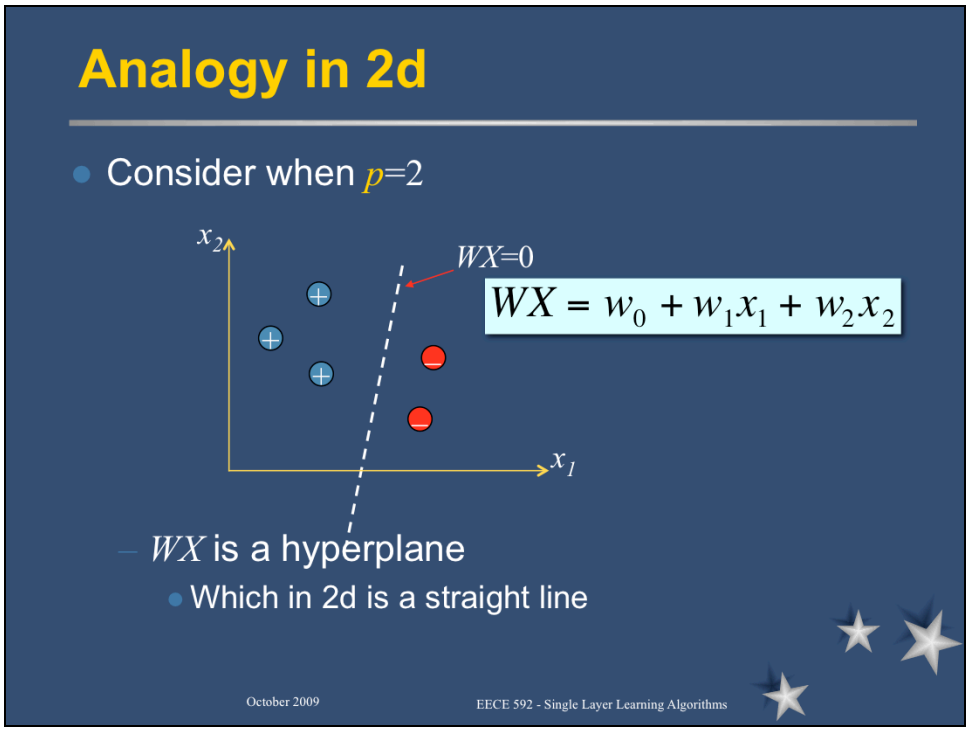
$X^k$ is a $p$ length vector of real numbers corresponding to the $p$ inputs. For perceptron learning, the inputs must belong to the set {+1, -1, 0}

$C^k$ is the correct response or classification for the input pattern and will have a value of either +1 or -1 (Actually, it can be any boolean representation, but we use bipolar).

Note that $C^k$ can be a vector also. In that case there would be a separate neuron for each component of the vector.

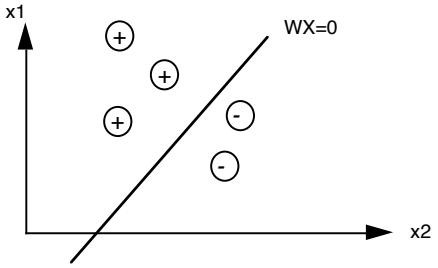



The goal of the perceptron learning algorithm is to find a weight vector that correctly classifies **all** input vectors. That is, for each input pattern $X^k$ used in training, the neuron(s) must generate the required output $C^k$, for the corresponding input.




Consider when $p$=2. This can be shown to be the problem of finding a hyperplane (which in this 2D case is a line) which correctly categorizes the inputs.

$WX = w_0 + w_1 x_1 + w_2 x_2$ equation of a straight line.




The weight vector must correctly satisfy all inputs $(x_1, x_2)$

## The Bias Term

- Consider a three input problem:

| | $u_1$ | $u_2$ | $u_3$ | | $u_4$ |
|---|---|---|---|---|---|
| $X^1$= | −1 | −1 | − 1 | $C^1$ = | + 1 |
| $X^2$= | +1 | −1 | − 1 | $C^2$ = | − 1 |
| $X^3$= | +1 | +1 | + 1 | $C^3$ = | + 1 |

- $w_0$ represents the bias term
  - Without bias, hyperplane forced to intersect origin

October 2013 EECE 592 - Single Layer Learning Algorithms

Bias Term : Analogy with a straight line

A popular form for the equation of a straight line is $y = mx + c$

Here $m$ is the gradient of the line and $c$ is the point at which the line intersects the $y$ axis. When $c$ is zero, the line intersects the origin.

Without a bias term, the hyperplane arrived at by the neural net is restricted to intersect the origin. This can be problematic and in many cases may prevent a solution from being found, even though the problem is linearly separable.

Thus it is useful to add a bias term to each neuron. The bias is analogous to "$c$" in $y = mx + c$

# The Bias Term cont.

- Adding a bias term:

| | $u_0$ | $u_1$ | $u_2$ | $u_3$ | | $u_4$ |
|---|---|---|---|---|---|---|
| $X^1$= | 1 | −1 | −1 | − 1 | $C^1$ = | + 1 |
| $X^2$= | 1 | +1 | −1 | − 1 | $C^2$ = | − 1 |
| $X^3$= | 1 | +1 | +1 | + 1 | $C^3$ = | + 1 |

  - Note the addition of the $u_o$ term.
    - It is always 1

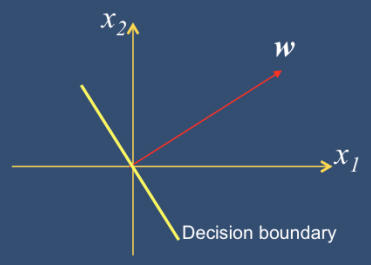


A bias term will in effect turn a three input problem into a four input problem, where the additional input is the bias term regarded as a weight from an input which is always +1.

When implementing a neural net, it is a good idea to include in the weighted sum for each neuron, a weight from an input that is always 1.

## Weight Vector & Decision Boundary

- Let $\boldsymbol{w}$ represent the weight vector




- For any neuron $S = W.X^k$

October 2009 EECE 592 - Single Layer Learning Algorithms

A neuron computes the weighted sum of its inputs $S=W.X^k$

S will be zero for any data point represented by vector $\boldsymbol{X^k}$ that is normal to $\boldsymbol{w}$. I.e. any point lying on the decision boundary will generate a zero weighted sum. Thus the weight vector is normal to the decision boundary. $S$ is the projection of $\boldsymbol{X^k}$ onto $\boldsymbol{w}$.

A perceptron is a special kind of neuron that thresholds the weighted sum using a step threshold function. Data points falling one side of the decision boundary have a negative projection and those falling the other side have a positive projection. So our perceptron is computing what is known in mathematics, as the scalar product or "dot" product of two vectors.

## The Perceptron Learning Algorithm

- 1 Set $w$ to the 0 vector
- 2 Choose $X^k$ and $C^k$ from training set
- 3 If correctly classified, do nothing else
  - $\boldsymbol{w}^* = \boldsymbol{w} + X^k.C^k$ (vector addition)
- 4 If all $X^k$ are correctly classified, stop else
  - goto step 2



The perceptron learning algorithm is very simple indeed.

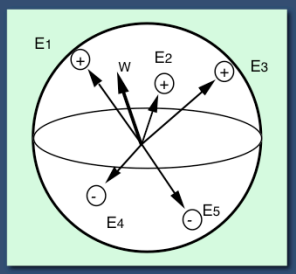
The goal is to arrive at a weight vector $\boldsymbol{w}$ which correctly classifies all samples in the training set. This is the stopping criteria.

A weight update step involves selecting a training sample that is not correctly classified and changing the weights according to the term in step 3 above. Since for perceptron leaning $C^k$ is actually a scalar that is either +1 or -1, the weight update step effectively is either adding or subtracting the example vector $\boldsymbol{X}^k$ from the current weight vector. That's it!

Note of course, the assumption is that the problem is linearly separable. That is that a linear hyperplane can be found that separates all positively classified samples from negatively samples.

## Theory of Perceptron Learning

- $W.X^k$ is the projection of the example vector onto the weight vector.
- The projection may be positive or negative





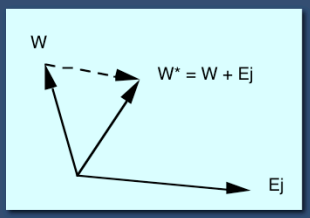

Each training example corresponds to a vector on a sphere as does the weight vector. (Note with the biploar representation, the Cartesian length of all input vectors is the same. This is not true for the binary representation).

The scalar product $W.X^k$ is the projection of the weight vector onto the example vector. The projection may be *positive* or *negative*.

The goal is to find a $W$ which has a positive projection for $X^k$ classified as $C^k = +1$ and a negative projection for all $C^k = -1$.

## Theory cont.

- Goal
- To find a w for which
  - $W.X^k$ = positive when $C^k$ = +1, and
  - $W.X^k$ = negative when $C^k$ = -1
- Turns out, step 3, will always improve misclassification

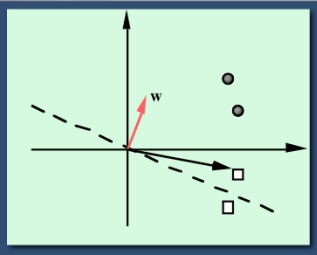




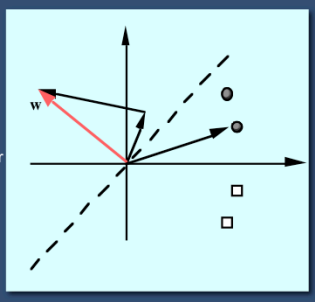

The modification in step 3 will always cause the weight vector to move closer to examples that should be positive and further from examples that should be negatively classified.

The proof of this theory is beyond the scope of this lecture and it is not intuitively obvious. For a formal proof to this theory, the reader is referred to the textbook by Fausett, recommended for this course.

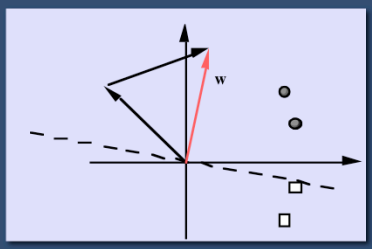
# Example




One of the squares is initially misclassified

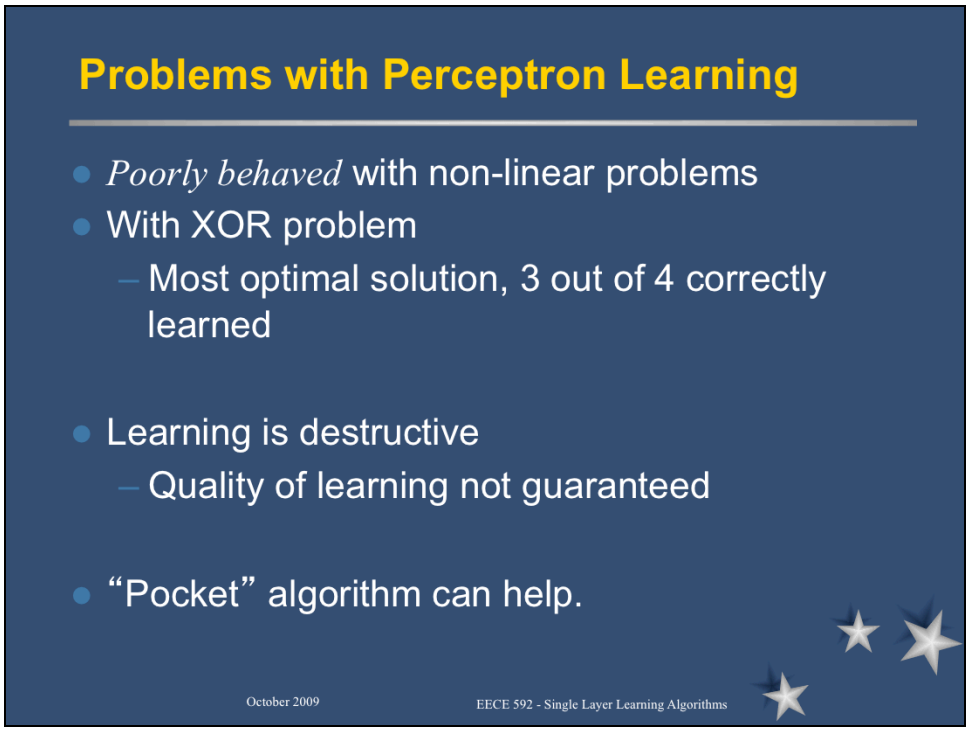



The misclassified vector is subtracted from the weight vector




This time a positive example is misclassified and the final update results in a perfect decision boundary.

from C.M.Bishop, "Neural Networks for Pattern Recognition", 1995




Problems

The perceptron learning algorithm is *poorly behaved* when presented with a non-separable set of training examples.

For example, when learning the XOR relationship, the best set of weights that could be hoped for is:

$$w_0\,(\text{bias}) = +1, w_1 = +1, \qquad w_2 = +1$$

which correctly classifies 3 out of the four training examples. This is not so bad.

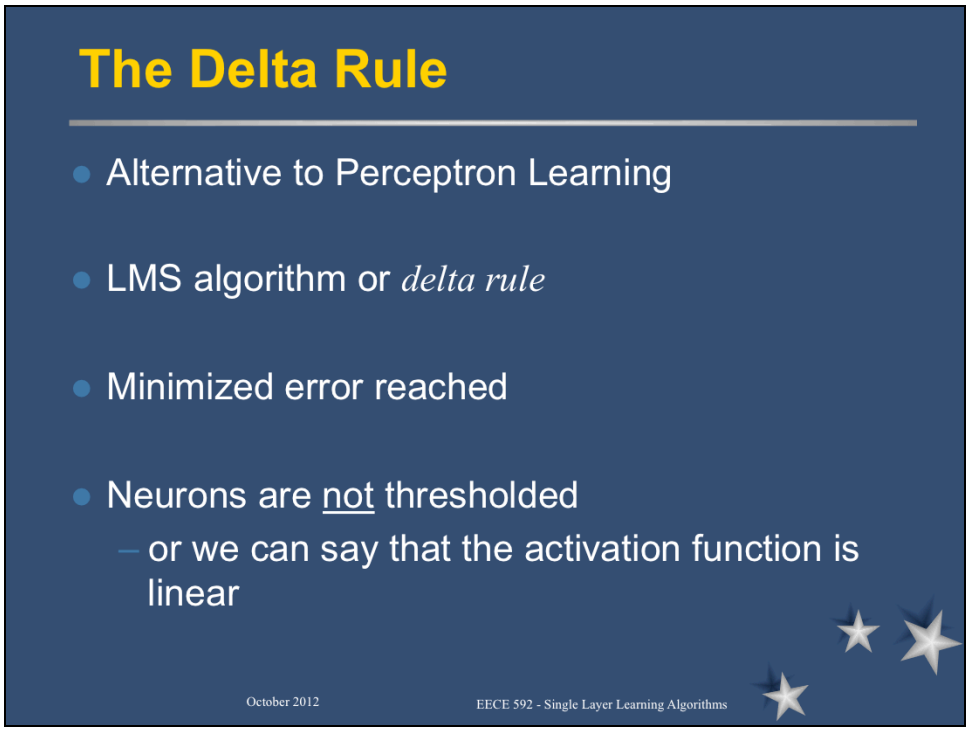
The problem is when the algorithm attempts to learn a misclassified example, it is not unusual for the algorithm to change the weights to a new configuration, eg.

$$w_0\,(\text{bias}) = 0, \qquad w_1 = +0, \qquad w_2 = +0$$

which classify none of the training set correctly! Thus after a given number of iterations, there is no guarantee of the quality of the weights.

The "Pocket" algorithm

Gallant has proposed a solution to this which called the Pocket algorithm. This works by *pocketing* the best set of weights encountered so far. The best or most optimal are those weights which classify as many of the training set as possible.




Widrow and Hoff (circa 1970s) presented a famous learning algorithm based upon least mean square errors, now known as the LMS algorithm or the *delta rule*.

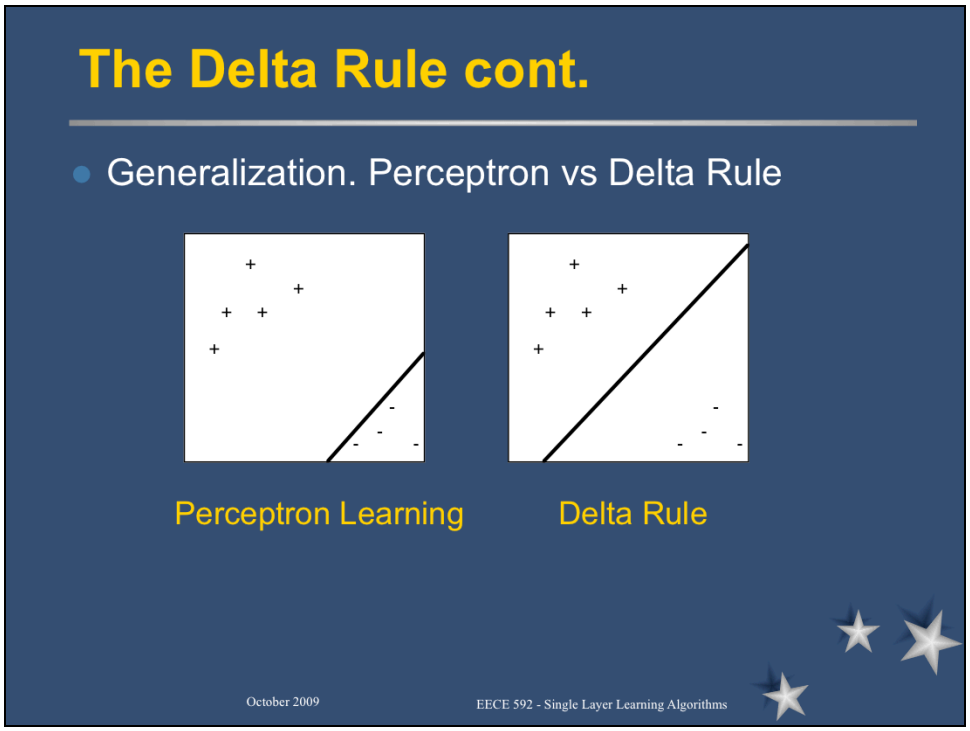
As the name suggests, the approach is based upon minimizing the squared error as a means of learning and provides an alternative to the perceptron learning algorithm for single layer neural net models.

Whereas perceptron learning approaches aim to find an optimal weight vector capable of correctly classifying the training set, the delta rule can be used with continuous valued cells and gradient descent to reach a *minimized error* for a particular problem.

Given a training set of N examples, the total error in the net can be represented as :

$$E = \sum_N E^k = \frac{1}{2}\sum_N (C^k - S^k)^2$$

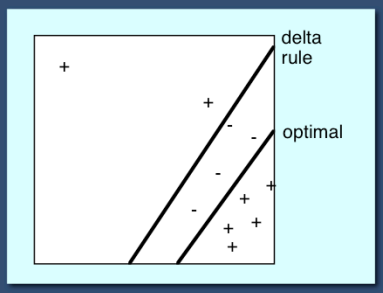
Where $S^k$ is the weighted sum for pattern $k$ and $C^k$ is the correct output for that pattern. Note that cells for the *delta rule* typically use a linear threshold function (such as the identity function).




As the diagram shows, the delta rule is capable of reaching a better solution than perceptron learning. In this case, *better* means better generalization.

## The Delta Rule cont.

- Optimality of solutions

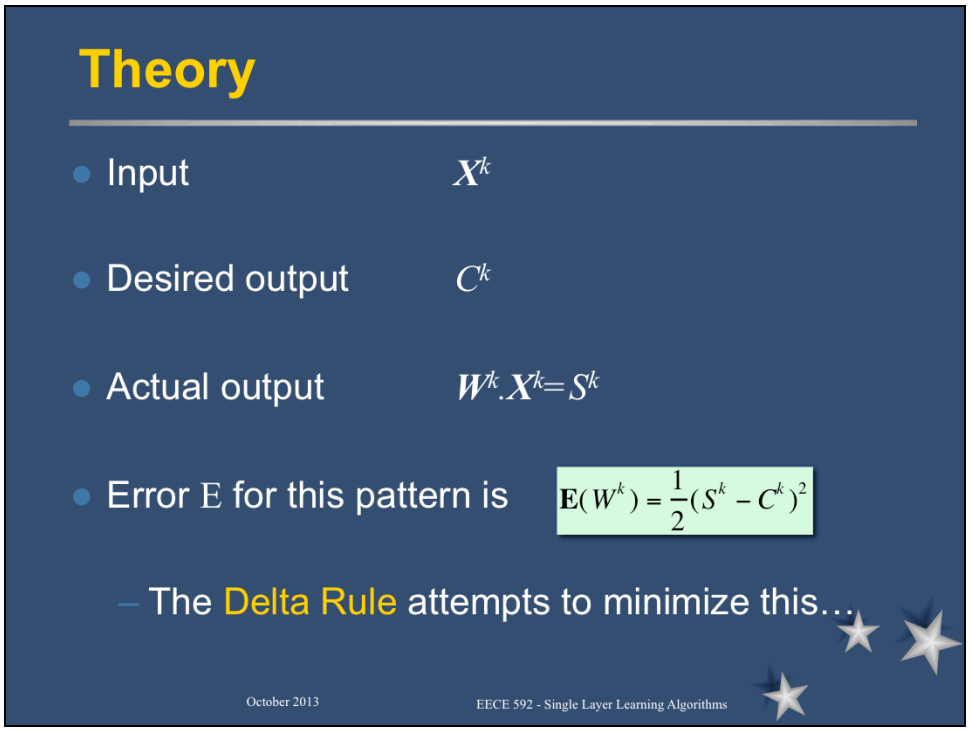



- In this case, Perceptron learning fares better



In this example, the delta rule over emphasizes a far-out point resulting in a solution that misclassifies many points. This point may even be spurious.

So the Perceptron learning algorithm (Gallant's "pocketing" variant) fares better than the Delta rule in this case.




Suppose for example $X^k$ the correct output is $C^k$ but the actual output is the weighted sum $W^k.X^k = (S^k)$

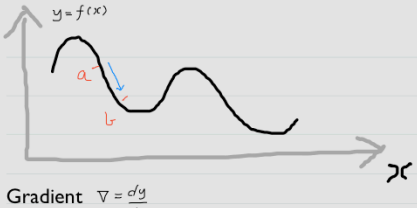
then the error **E** for this pattern is $\mathbf{E}(W^k) = \frac{1}{2}(S^k - C^k)^2$

The trick is determining how to update the weight so that it results in a reduction in the overall computed error **E** for all patterns *k*. For this we apply gradient descent.

# Gradient Descent

A quick refresher

Consider a simple function




Gradient $\nabla = \frac{dy}{dx}$

Move in the direction of the negative gradient

and $G = a - \rho \frac{dy}{dx}(a)$

the step size

For a multivariate function $y = f(x_1, x_2, x_3 \cdots x_n)$

Gradient is a vector of partial derivatives $\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial x_3} \cdots \frac{\partial f}{\partial x_n}\right)$

## Theory cont.

- Apply gradient descent to:

$$E = \frac{1}{2}\sum_{k}(S^k - C^k)^2$$

- Where:
  - $S^k$ is a function of the weights



Gradient descent is defined as minimization of **E** over all patterns *k*.

## Theory cont.

- Weight change is proportional to -E
  - This can be defined in terms of gradient descent:

$$\Delta_k w_j \propto -\frac{\partial E^k}{\partial w_j}$$

- This leads to the following weight update step

$$W^* = W + \rho\delta X$$



The idea is to make a change in the weight that will cause the total error **E** to decrease. I.e. we want the weight change to be in the direction of the negative gradient of **E** with to the negative of the error as measured on the current pattern with respect to each weight. Lets start with the definition of gradient descent:

$$\Delta_k w_j \propto -\frac{\partial E^k}{\partial w_j} \quad \text{leading to} \quad \frac{\partial E^k}{\partial w_j} = \frac{\partial E^k}{\partial S_k}\frac{\partial S^k}{\partial w_j}$$

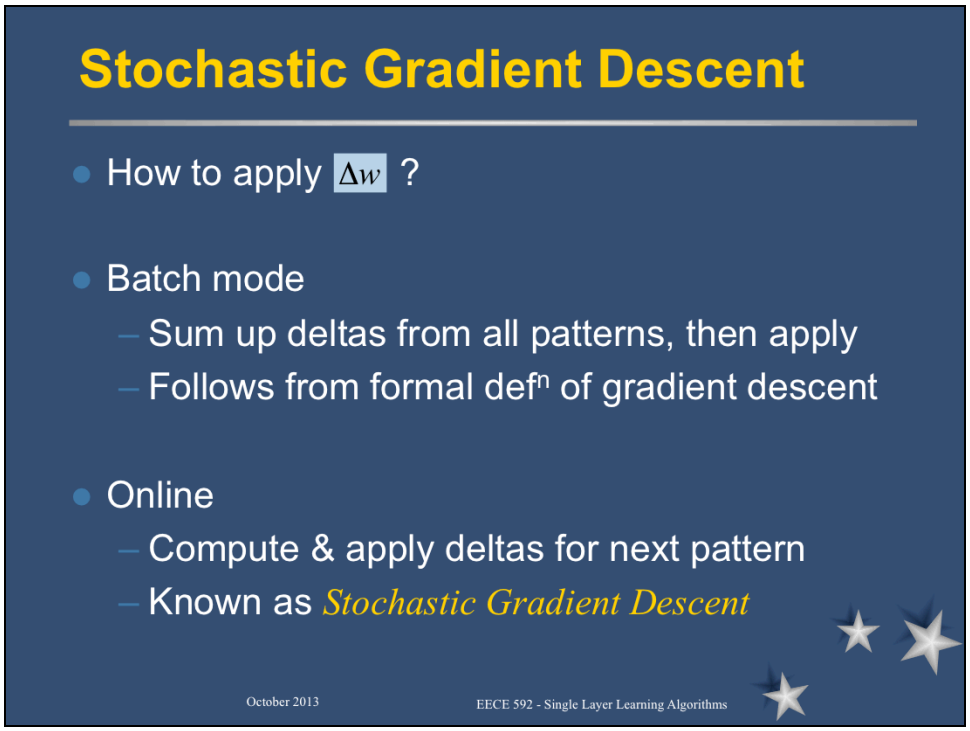
where $S^k$ is the output for the k$^{th}$ pattern. Since the delta rule uses linear elements, $S^k = W^k.X^k$

Then $\frac{\partial S^k}{\partial w_j} = X^k$ and also $\frac{\partial E^k}{\partial S^k} = -(C^k - S^k)$

and then $\Delta_k w_j = \rho\delta^k X^k$

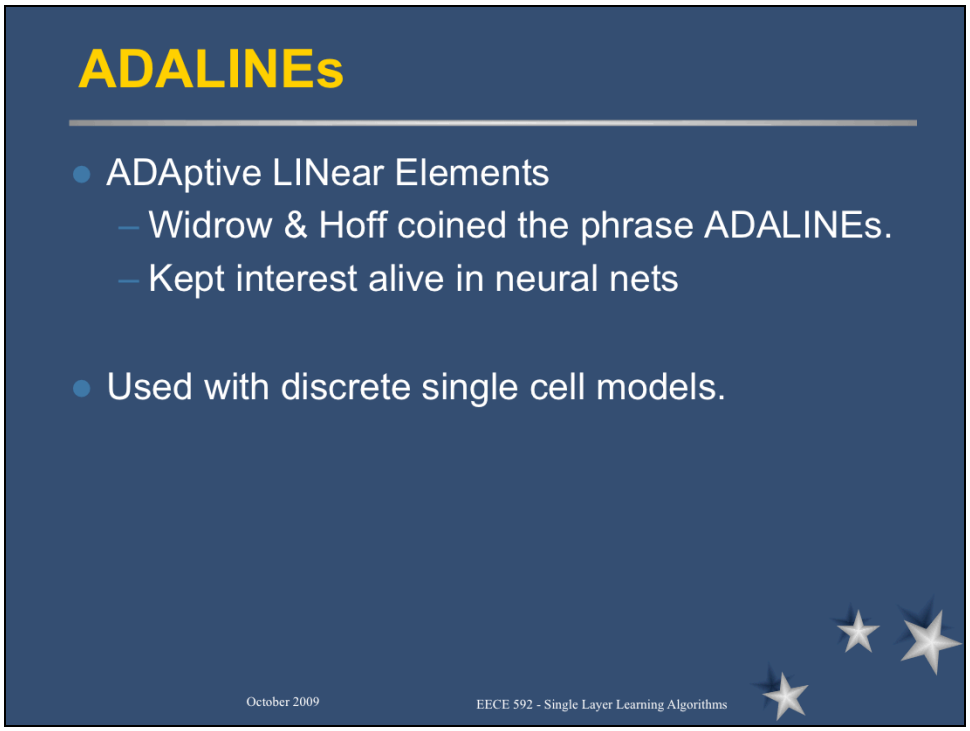
Where $\delta^k = (C^k - S^k)$ and $\rho$ is the gradient step size which we use to remove the constant of proportionality.

$$W^* = W + \rho\delta X$$



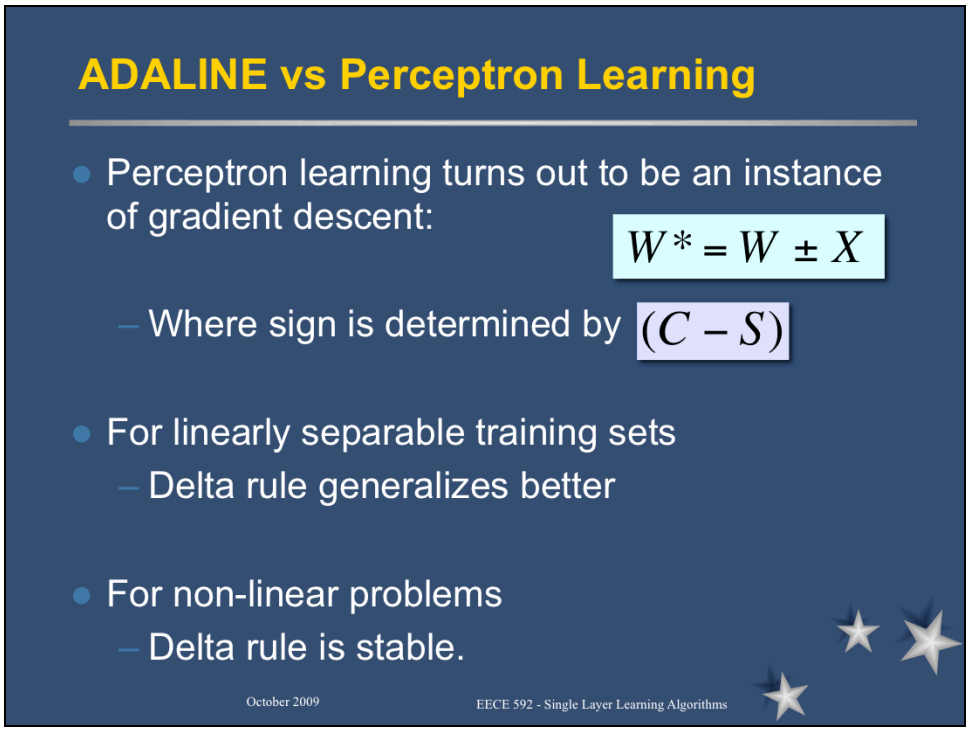

The definition of gradient descent suggests that the $\Delta w$ is computed from all training patterns prior to actually changing the weights. This is referred to as batch mode training since the weight updates from each training pattern are batched up and a single update made once the entire training set has been presented.

In practice however, an alternative form is used. In this case, following presentation of each and every training pattern, a weight update is computed and applied immediately. This approach performs the updates in an 'online' manner and is a form of *stochastic gradient descent*. Stochastic because the next weight change may not necessarily be in the direction of the negative gradient and maybe in any random direction. The approach works because on average, the update will be in the right direction.




Widrow and Hoff used the delta rule to train weights for a discrete single cell model which they called the ADALINE - ADAptive LINear Element.

Note that the term ADALINE was originally an abbreviation for *adaptive linear neuron*, however Widrow & Hoff acknowledged that activity in neural network was not in "fashion" and opted to not use it feeling that their work may not be received favourably as a result!




Comparing the ADALINE to Perceptron learning, it turns out that Perceptron learning is a less general form of gradient descent that uses a constant step size:

$$W^* = W \pm X$$

where the sign is the same as

$$(C - S)$$

For separable training sets, Perceptron learning guarantees finding an optimal solution, whereas the Widrow and Hoff approach will tend to asymptotically converge.

The Widrow and Hoff algorithm is however likely to find a solution that generalizes better.

Also, for non-separable training sets, the Widrow and Hoff rule is stable unlike Perceptron learning.