Adalines

Primitive Artificial Neurons
Building blocks for Multi-Layer Networks
(called “multi-layer perceptrons” (MLP) strangely enough)
Adaline

- The Adaline ("Adaptive Linear Neuron" or "Adaptive Linear Element") is a model similar to the Perceptron.

- There are several variations:
  - One has the threshold function same as a perceptron.
  - Another uses a pure linear function with no threshold.
  - Others can be used, such as a sigmoid function.
Adaline Inventors
Bernard Widrow and Marcian (Ted) Hoff

Bernard Widrow,
Professor Emeritus of E.E.,
Stanford University

Marcian Hoff
Co-inventor of Patent 3,821,715
Microprocessor Concept and Architecture
With or without the threshold, the Adaline is trained based on the output of the linear function rather than the final output.
The catch here is that we have to state the desired value in terms of the output of the linear part, rather than the output after the limiter.

What is this for a classifier?
A reasonable approach is to use separable nominal target values such as -1 as desired for a “no” classification and 1 for a “yes” classification. The hardlim will convert it to \{1, 0\} (unipolar, or \{1, -1\} bipolar).
The formula for Adaline weight updating will be seen as very similar to the Perceptron: Add to the weights $\Delta w$ where

$$\Delta w = \varepsilon \eta [1, x_1, x_2, \ldots, x_n]$$

only now the error is not just 1 or -1 as before.

Instead, the error can have a fractional value, as it is based on the output of the linear part of the device.
Adaline Training (4)

- One major difference from this vs. the Perceptron is that the learning rate $\eta$ can’t be as arbitrary. It will generally need to be less than 1.

- There is theory that tells us how large we can make the learning rate and still converge.
The Adaline admits a refined stopping criterion:

The **Mean-Squared Error (MSE)** is the average of the squares of the error taken over all samples.

Squaring makes the measure insensitive to the sign of the error. It also provides certain analytic properties.

This quantity ideally converges toward a specific minimum (which might never be exactly attained). The algorithm can be set to stop when the MSE reaches a desired value.
Adaline Example

- We’ll use the same example as for perceptron. But now we’ll train on the output of the linear portion and target for +1 for a “yes” answer and -1 for a “no” answer (bipolar domain).
  - (4, 5)  +1
  - (6, 1)  +1
  - (4, 1)  -1
  - (1, 2)  -1

- Try a learning rate of 0.01.

- Also, here I used a phantom input of -1 rather than +1, so the corresponding weight will adapt with the opposite sign.
Adaline Training Example

- Start with initial weights all 0.
- (In general, can use random weights.)
- Progress:
  - trying sample desired: 1, inputs: -1 4 5 output: -1
  - diff is 2
  - error is 1
  - new weights: -0.01 0.04 0.05
weights: -0.01 0.04 0.05

trying sample desired: 1, inputs: -1 6 1 output: 1
diff is 0
error is 0.7
new weights: -0.017 0.082 0.057

trying sample desired: -1, inputs: -1 4 1 output: 1
diff is -2
error is -1.402
new weights: -0.00298 0.02592 0.04298
Adaline Training Example

- trying sample desired: -1, inputs: -1 4 1 output: 1
- diff is -2
- error is -1.402
- new weights: -0.00298 0.02592 0.04298

- trying sample desired: -1, inputs: -1 1 2 output: 1
- diff is -2
- error is -1.11486
- new weights: 0.0081686 0.0147714 0.0206828

- **epoch 1**: wrong = 3, mse = 1.17463
Adaline Training Example

- epoch 1: wrong = 3, mse = 1.17463
- epoch 2: wrong = 2, mse = 1.11176
- epoch 3: wrong = 2, mse = 1.08817
- epoch 4: wrong = 2, mse = 1.07521
- epoch 5: wrong = 2, mse = 1.06563
  ... 
- epoch 30: wrong = 2, mse = 0.888344
- epoch 31: wrong = 1, mse = 0.881999
  ... 
- epoch 197: wrong = 1, mse = 0.363726
- epoch 198: wrong = 0, mse = 0.362493
- Final weights: 1.54436 0.273645 0.252003
Adaline Training Example

- **Final weights:** 1.54436 0.273645 0.252003

<table>
<thead>
<tr>
<th>input</th>
<th>desired</th>
<th>weighted sum</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4, 5)</td>
<td>+1</td>
<td>0.810235</td>
<td>1</td>
</tr>
<tr>
<td>(6, 1)</td>
<td>+1</td>
<td>0.359513</td>
<td>1</td>
</tr>
<tr>
<td>(4, 1)</td>
<td>-1</td>
<td>-0.197777</td>
<td>-1</td>
</tr>
<tr>
<td>(1, 2)</td>
<td>-1</td>
<td>-0.766709</td>
<td>-1</td>
</tr>
</tbody>
</table>
Alternate Rule Names

• Because the Adaline rule minimizes MSE, it is sometimes called the "LMS rule" [LMS = "least mean square"].

• The term "Delta rule" is also sometimes used, although this will be seen to be a rule for a more general class of networks.

• The term "Widrow-Hoff" rule is also used.
Pattern Recognition Demo of Widrow-Hoff Model

Matlab demo nnd10lc (lc = linear classification)
16 inputs
1 neuron
What function is minimized?

- Think of the inputs as being **constants**.

- The **weights** are the variables.

- We want to find the weights that minimize the **MSE** as a function of the weights.

- The fact that the MSE is defined **analytically** is a big help.
Gradient Descent

- Gradient descent, aka steepest descent, is a general method for finding the minimum of the error function (a function of the weights).

- It consists of computing the gradient of the function, then taking a small step in the direction of negative gradient, which hopefully corresponds to decreased function value, then repeating for the new value of the dependent variable.
Gradient?

- The gradient is simply a generalization of the ordinary derivative $d/dw$ to $n$ dimensions.

- Specifically, it is the **vector** of partial derivatives, one component for each dimension:

$$\nabla_w = [\partial / \partial w_1, \partial / \partial w_2, \ldots, \partial / \partial w_n]$$

Applying this **operator** to the error function $J(w)$, a function of just the weights (given fixed input and desired values) gives a vector, one element for each weight parameter.

$$\nabla_w J = [\partial J / \partial w_1, \partial J / \partial w_2, \ldots, \partial J / \partial w_n]$$
Gradient Descent

Negative-of-gradient directions shown by arrows

slope, i.e. gradient, is positive

$\Delta w \propto -\text{gradient}$

MSE

weight points at which gradient is evaluated

w weight
Gradient Descent

- The previous diagram is mainly to guide intuition.

- A single dimension for weights is not typical. We need a two dimensional domain for just a single weight plus a bias.

- For the general case, the gradient is a vector of gradient components, one for each weight (including bias).
Error Function over 2-D Weight Space
(bias counts as a weight)

\[ J = \text{error function} \]
Gradient Descent for 2-D
2-D Gradient Descent Projection
Computing Gradients for the Adaline

- \( \text{MSE} = J(w) = \Sigma (\text{desired}-\text{actual})^2/n \) where \( \Sigma \) is over \( n \) samples.

- ‘\( \text{desired}_j \)’ is a fixed value for each sample \( j \).

- \( \text{actual} = \Sigma w_j x_j \) (\( \Sigma \) over input lines, including phantom input for threshold or bias)

- So \( J(w) = \Sigma (\text{desired}_j - \Sigma w_j x_j)^2/n \)
**On-Line Approximation to Gradient**

- “On-line” means based on a single sample, vs. “batch”, which means using all samples.

- $J \approx (d_j - \sum w_j x_j)^2$

- $i^{th}$ gradient component = $\frac{\partial J}{\partial w_i}$

\[
= \frac{\partial}{\partial w_i} (d - \sum w_j x_j)^2 \\
= 2 (d - \sum w_j x_j) \frac{\partial}{\partial w_i} (d - \sum w_j x_j) = -2 \varepsilon x_i
\]

= error, $\varepsilon$

= $-x_i$
Computing Gradients

- \( i^{th} \) gradient component = \(-2 \varepsilon x_i\)
- However we want to move in the direction of negative gradient, and also temper by the learning rate \( \eta \), so:

\[
\Delta w_i = 2 \varepsilon \eta x_i
\]

which we recognize as the LMS (Adaline) rule (2 could be folded into \( \eta \)).

- This is the rule used in our earlier demonstration.
Vector Analysis of Gradient Descent for Adaline

- For simplicity, assume $w$ is a row vector, and $x$ is a matrix of columns, each column being an input sample, with phantom input as before, $d$ is vector of desired.
- Error vector $e = d - wx$ is a function of $w$
- $MSE = J(w) = ee'/n \quad (n = \#\text{samples}, ' \text{ is transpose})$
  \[= (d - wx)(d-wx)'/n \]
  \[= [dd' - 2(wx)d' + (wx)*(wx)']/n \]
  \[= c - 2wh + wRw' \]

where $c = dd'$, $h = xd'$, $R = xx'$

This is quadratic form in variables $w$. 
Convergence Analysis of Gradient Descent for Adaline

- $J(w) = c - 2wh + wRw'$

  Quadratic form in $w$ with coefficients derived from the sample inputs $x$ and desired values $d$.

- $R$ is called the **correlation matrix**.
The standard quadratic form is

\[ J(w) = c + wb + (1/2)wAw' \]

where \( b = -2h \)
\[ A = 2R \]

Here \( A \) is also called the Hessian matrix. It is the matrix of 2nd partial derivatives of the quadratic form.
Demo `nnd8gf`
(quadratic function (2D))

Adjust $A$, $d$, $c$ to observe shape change.

$$F(x) = \frac{1}{2}x'Ax + d'x + c$$

$A = \begin{bmatrix} 1.5 & -0.7 \\ -0.7 & 1 \end{bmatrix}$
$d = \begin{bmatrix} 0.25 \\ 0.25 \end{bmatrix}$
$c = \begin{bmatrix} 1 \end{bmatrix}$
Analytic Minimization

- $\nabla_w$ is the gradient (vector of derivs) operator
- $\nabla_w J(w) = \nabla_w (c + wb + (1/2)wAw')$
  $= b + A w'$ (a vector)

- It can be shown that if $J$ has a stable point, it will be at a point $w^*$ where $\nabla J(w^*) = 0$, i.e. $b + A w^* = 0$

so $w^* = -A^{-1} b$, provided $A$ is invertible
Worked Example using Matlab

- Done in class
Stable Points

- In general, \( w^* = -A^{-1}b \) is just a stable point.

- It may correspond to a minimum, maximum, or saddle.

- Ideally we have a minimum.
Eigenvalues of the Hessian, $A$

- The curvature along the principal axes of the surface is proportional to the *eigenvalues* of $A$.

- The limiting factor on convergence rate is the largest eigenvalue, i.e. the greatest curvature.
Characteristic of Stationary Point
Dependent on Eigenvalues of Hessian

<table>
<thead>
<tr>
<th>Eigenvalues</th>
<th>Stationary Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>All positive</td>
<td>Strong minimum</td>
</tr>
<tr>
<td>All negative</td>
<td>Strong maximum</td>
</tr>
<tr>
<td>Mixture of negative and positive</td>
<td>Saddle point</td>
</tr>
<tr>
<td>All non-negative, but some 0</td>
<td>Weak minimum</td>
</tr>
<tr>
<td>All non-positive, but some 0</td>
<td>Weak maximum</td>
</tr>
</tbody>
</table>
Outline of Convergence Analysis of On-Line Gradient Descent for Adaline

\[ \Delta w = 2 \epsilon \eta x \]  
\[ w(k+1) = w(k) + 2 \eta \epsilon(k) x(k) \]  

Thus \[ E[w(k+1)] = E[w(k)] + 2\eta E[\epsilon(k) x(k)] \]  
... math ...  
\[ = (I-2\eta R) E[w(k)] + 2\eta h \]  

(E is expectation, I is the identity matrix, and h a constant vector. R is the correlation matrix).

For convergence, the eigenvalues of the matrix I-2\eta R must be within the unit circle.
Convergence of
Gradient Descent for Adaline

- If $\lambda_i$ is an eigenvalue of $R$, convergence requires $|1 - 2 \eta \lambda_i| < 1$.

- This simplifies to $\eta < 1/\lambda_i$ for all eigenvalues $\lambda_i$, in particular for the maximum eigenvalue.

  \[
  \eta < 1/\lambda_{\text{max}}
  \]

Bound on learning rate for convergence of Adaline training by gradient descent
Worked On-Line Learning Example
(from Neural Network Design)

Sample1 \( \{ x_1 = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}, \quad d_1 = \begin{bmatrix} -1 \end{bmatrix} \} \)

Sample 2 \( \{ x_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, \quad d_2 = \begin{bmatrix} 1 \end{bmatrix} \} \)

Correlation Matrix: \( R = E[xx^T] = \frac{1}{2} x_1 x_1^T + \frac{1}{2} x_2 x_2^T \)

\[
R = \frac{1}{2} \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} \begin{bmatrix} -1 & 1 & -1 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} \begin{bmatrix} 1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & -1 & 1 \end{bmatrix}
\]

\( \lambda_1 = 1.0, \quad \lambda_2 = 0.0, \quad \lambda_3 = 2.0 \)

\( \eta_{\text{max}} = \frac{1}{\lambda_{\text{max}}} = \frac{1}{2.0} = 0.5 \)
Training, 1st Step, with $\eta = 0.2 < \eta_{\text{max}}$

Sample 1

$$a(0) = W(0)p(0) = W(0)x_1 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ -1 & -1 \end{bmatrix} = 0$$

$$\epsilon(0) = d(0) - a(0) = d_1 - a(0) = -1 - 0 = -1$$

$$W(1) = W(0) + 2\eta\epsilon(0)x^T(0)$$

$$W(1) = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} + 2(0.2)(-1) \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}^T = \begin{bmatrix} 0.4 & -0.4 & 0.4 \end{bmatrix}$$
2nd Step

\[
a(1) = W(1)p(1) = W(1)x_2 = \begin{bmatrix} 0.4 & -0.4 & 0.4 \\
1 & 1 & -1 \end{bmatrix} = -0.4
\]

\[
\varepsilon(1) = d(1) - a(1) = d_2 - a(1) = 1 - (-0.4) = 1.4
\]

\[
W(2) = \begin{bmatrix} 0.4 & -0.4 & 0.4 \\
1 & 1 & -1 \end{bmatrix}^T + 2(0.2)(1.4) = \begin{bmatrix} 0.96 & 0.16 & -0.16 \end{bmatrix}
\]
3rd Step

Sample 1 again

\[ a(2) = W(2)p(2) = W(2)x_1 = \begin{bmatrix} 0.96 & 0.16 & -0.16 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} = -0.64 \]

\[ \varepsilon(2) = d(2) - a(2) = d_1 - a(2) = -1 - (-0.64) = -0.36 \]

\[ W(3) = W(2) + 2\eta\varepsilon(2)x^T(2) = \begin{bmatrix} 1.1040 & 0.0160 & -0.0160 \end{bmatrix} \]

In the limit …

\[ W(\infty) = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \]
“Learning Curve” Comparisons for Convergence
Demo nnd9sdq
(steepes descent for quadratic)

Select starting point in weight space and learning rate to observe convergence (or lack thereof).
Generalizing the Adaline
(leading to multi-layer network training)

Suppose that we replace the threshold stage with a **general analytic function** $f$ and revert to expressing desired in terms of its output:

![Diagram](attachment:image.png)

-1

$w_0$

$w_1$

$\sum$

$w_n$

**analytic function** $f$, the *activation function*

adjust

desired
Generalizing the Adaline with analytic activation function

- Consider again the derivation of the LMS rule:
  \[ J(w) = \frac{\sum (d - f(\sum w_j x_j))^2}{n} \]

- \[ J \approx (d - f(\sum w_j x_j))^2 \quad \text{(on-line approximation)} \]
Deriving Learning Rule for Generalized Adaline

\[ J \approx (d - f(\Sigma w_j x_j))^2 \text{ (on-line approximation)} \]

\( i^{th} \) gradient component = \( \frac{\partial J}{\partial w_i} \)

\[
= \frac{\partial}{\partial w_i}(d - f(\Sigma w_j x_j))^2 \\
= 2(d - f(\Sigma w_j x_j)) \frac{\partial}{\partial w_i} (d - f(\Sigma w_j x_j)) \\
= -2 \varepsilon \frac{\partial}{\partial w_i} f(\Sigma w_j x_j) \\
= -2 \varepsilon f'(\Sigma w_j x_j) \frac{\partial}{\partial w_i} \Sigma w_j x_j \\
= -2 \varepsilon x_i f'(\Sigma w_j x_j) \text{ where } f' \text{ is the ordinary derivative of the activation function} \]
Generalized LMS Rule
(or Delta Rule)

- In vector form, $\Delta w = 2 \varepsilon \eta f'(\sum w_j x_j) \times$ assuming that activation function $f$ has derivative $f'$.

- $\sum w_j x_j$ is often called the “net” value or “activation” value.

- For the special case of $f$ being the identity function, this reduces to the LMS rule we had before.

- Next we see uses of the more general case.
Approximating Limiter Function Analytically

- In the Adaline with threshold, we can’t very well treat the model analytically, due to the fact that we have a non-continuous function at the output.

- But we can approximate the non-continuous function with a continuous one:
Sigmoid Curves

- The “S” shape on the right of the previous slide is called a sigmoid curve.

- This is a generic term and there are several different analytic functions that behave this way.
Logistic Sigmoid

- Logistic function ("logsig"-Matlab):
  \[ f(x) = \frac{1}{1+\exp(-ax)} \]

- \( f'(x) = f(x)(1-f(x)) \) is shortcut to derivative
Hyperbolic Sigmoid

- Hyperbolic tangent function ("tansig"): 
  \[ f(x) = \tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \]

- \[ f'(x) = 1 - f^2(x) \] is shortcut to derivative
“Squashing” Functions

- Sigmoids, step functions, and other functions that force their results to be in a limited range are called “squashing functions”.

- It is generally accepted that biological neural system is based on such functions, as there are physical limits to the response level.