Adalines

Primitive Artificial Neurons
Building blocks for Multi-Layer Networks
(called “multi-layer perceptrons” (MLP) strangely enough)
Sweet 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 devised.
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 any nominal target value such as -1 as desired for a “no” classification and a +1 for a “yes” classification.

\[ w_0 \sum w_1 w_n \]

adjust

desired = 1 or -1, say
Adaline Training (3)

- The formula for Adaline weight updating will be seen as very similar to the Perceptron:
  Add to the weights $\Delta w$ where

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

Adaline learning rule

only now the error is not limited to 1, -1, 0 as before; it can have a fractional value, since 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 can’t be so arbitrary. It will generally need to be a lot less than 1.

- There is a theory that tells us how large we can make the learning rate.
Adaline Convergence (2)

- The Adaline admits a more 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 before. 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.
  - (4, 5) +1
  - (6, 1) +1
  - (4, 1) -1
  - (1, 2) -1
- Try a learning rate of 0.01.
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
Adaline Training Example

- 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**" \([\text{LMS} = \text{“least mean square”}]\).

- 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.
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 is a 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:

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

Apply this operator to the error function.
Gradient Descent

MSE

Negative gradient direction

weight
The previous diagram is mainly to enhance our intuition.

A single dimension for weights (including threshold or bias) 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 vs. 2-D Weight Space
(one weight is threshold or bias)
2-D Gradient Descent
2-D Gradient Descent Projection
Computing Gradients for the Adaline

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

- ‘desired’ is a fixed value for each sample.

- actual $= \sum w_jx_j$ (\sum over input lines, including phantom input for threshold or bias)

- So $J(w) = \sum (\text{desired}- \sum w_jx_j)^2/n$
On-Line Approximation to Gradient

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

- $J \approx (d - \sum w_j x_j)^2 \quad (d = \text{desired})$

- $i^{th}$ gradient component = $\frac{\partial J}{\partial w_i}$
  $= \frac{\partial}{\partial w_i} (d - \sum w_j x_j)^2$
  $= 2 (d - \sum w_j x_j) \frac{\partial}{\partial w_i} (d - \sum w_j x_j) = -2 \varepsilon x_i$

  
  $= \text{error}, \varepsilon \quad \text{and} \quad -x_i$
Computing Gradients

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

  Amount to add to weight is
  \[ \Delta w_i = 2 \varepsilon \eta x_i \]
  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

- \( \text{MSE} = J(w) = E[(d - w^T x)^2] \)  
  \( E = \text{expectation or mean averaged over all samples} \)

\[
\begin{align*}
  &\text{MSE} = J(w) = E[(d - w^T x)^2] \\
  &= E[d^2 - 2dw^T x + w^T x x^T w] \\
  &= E[d^2] - E[2dw^T x] + E[w^T x x^T w] \\
  &= E[d^2] - 2w^T E[d x] + w^T E[x x^T] w \\
  &= c - 2w^T h + w^T R w, \text{ for appropriate const. } c, h, R
\end{align*}
\]
Vector Analysis of Gradient Descent for Adaline

- $J(w) = c - 2w^T h + w^T R w$

  where $c = E[d^2]$, $h = E[d x]$, $R = E[x x^T]$

- This is a **quadratic form in** $w$ with coefficients derived from the data vectors $x$.

- $R$ is called the (auto-)**correlation matrix**.
Standard Quadratic Form in $w$

- $J(w) = c + w^T b + (1/2) w^T A w$

  where $c = E[d^2]$
  $b = -2E[d \times]$
  $A = 2E[x \times^T]$

- $A$ is called the **Hessian** matrix. It is the matrix of 2nd partial derivatives of the surface.
Analytic Minimization

- \( \nabla_w J(w) = \nabla_w (c + w^Tb + (1/2)w^TAw) = b + Aw \)

- It can be shown that if \( J \) has a minimum, it will be at a point \( w^* \) where \( \nabla J(w^*) = 0 \), i.e. \( b + Aw^* = 0 \)
  - i.e. \( w^* = A^{-1}b \)

recalling \( A = 2E[x x^T] \), \( b = -2E[dx] \)
Stable Points

- In general, $w^* = A^{-1}b$ is just a **stable** point.
- It may correspond to a minimum, maximum, or saddle.
Outline of Convergence Analysis of Gradient Descent for Adaline

- $\Delta w = 2 \varepsilon \eta x$ (weight change at $k^{th}$ step)
  $w(k+1) = w(k) + 2 \eta \varepsilon(k)x(k)$

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

(I is the identity matrix, and $h$ a constant vector)

- 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 one.

$\eta < 1/\lambda_{\text{max}}$

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

Sample 1 \[ \begin{cases} x_1 = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}, & d_1 = \begin{bmatrix} -1 \end{bmatrix} \end{cases} \]

Sample 2 \[ \begin{cases} x_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, & d_2 = \begin{bmatrix} 1 \end{bmatrix} \end{cases} \]

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

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

\[ \lambda_1 = 1.0, \quad \lambda_2 = 0.0, \quad \lambda_3 = 2.0 \]

\[ \eta_{\text{max}} = \frac{1}{\lambda_{\text{max}}} = \frac{1}{2.0} = 0.5 \]
Training, 1st Epoch, with $\eta = 0.2 < \eta_{\text{max}}$

Sample 1

\[
\begin{align*}
  a(0) &= \mathbf{W}(0) \mathbf{p}(0) = \mathbf{W}(0) \mathbf{x}_1 = 
  \begin{bmatrix} 0 & 0 & 0 \\ 1 & -1 \\ -1 \end{bmatrix} = 0 \\
  \varepsilon(0) &= d(0) - a(0) = d_1 - a(0) = -1 - 0 = -1 \\
  \mathbf{W}(1) &= \mathbf{W}(0) + 2 \eta \varepsilon(0) \mathbf{x}^T(0) \\
  \mathbf{W}(1) &= 
  \begin{bmatrix} 0 & 0 & 0 \\ 1 & -1 \\ -1 \end{bmatrix} + 2(0.2)(-1) 
  \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}^T = 
  \begin{bmatrix} 0.4 & -0.4 & 0.4 \end{bmatrix}
\end{align*}
\]
2nd Epoch

Sample 2  \( a(1) = W(1)p(1) = W(1)p_2 = \begin{bmatrix} 0.4 & -0.4 & 0.4 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} = -0.4 \)

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

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

\[ a^{(2)} = W^{(2)}p^{(2)} = W^{(2)}p_1 = \begin{bmatrix} 0.96 & 0.16 & -0.16 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} = -0.64 \]

\[ \epsilon^{(2)} = d^{(2)} - a^{(2)} = d_1 - a^{(2)} = -1 - (-0.64) = -0.36 \]

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

\[ W^{(\infty)} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \]
“Learning Curve”

The diagram illustrates the concept of a learning curve with different scenarios based on the learning rate $\eta$.

- **Small $\eta_1$:** The weight tracks are close to the optimal weight $w^*$, indicating effective learning.
- **$\eta_2 > \eta_1$:** The weight tracks fluctuate around $w^*$, showing a balance between learning and stability.
- **Too large $\eta_3$:** The weight tracks diverge significantly from $w^*$, suggesting overfitting.

The number of iterations vs. the weight $w(k)$ shows how the weight changes with iterations for different learning rates.
Generalizing the Adaline

- We have already discussed the problem of a threshold output in this picture
Suppose that we replace the threshold stage with a general analytic function $f$ and revert to expressing desired in terms of its output:
Generalizing the Adaline

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

- \[ J \approx (d - f(\sum w_j x_j))^2 \] (on-line approximation)
Generalizing the Adaline

- \( J \approx (d - f(\Sigma w_j x_j))^2 \) (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}
Generalized LMS Rule
(or Delta Rule)

\[ \Delta w = 2 \varepsilon \eta f'(\sum w_j x_j) x_i \]
assuming that \( f \) has a derivative \( f' \).

\[ \sum w_j x_j \]
is often called the “net” value or “activation” value, and \( f \) the activation function.

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 Step 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 are a type of curve that is often used in various fields such as mathematics, statistics, and machine learning. They are characterized by their S-shaped curve, which is why they are called sigmoid curves.

The S-shaped curve is used to model various phenomena, such as the growth of populations, the spread of diseases, and the spread of information in social networks.

Another important use of sigmoid curves is in the field of machine learning, where they are used in the activation functions of neural networks. Sigmoid functions are particularly useful in binary classification problems, where the output is either 0 or 1.
Logisitic Sigmoid

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

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

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

- \[ f'(x) = 1 - f^2(x) \text{ shortcut to derivative} \]
Squashing Functions

• Sigmoid functions, 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.