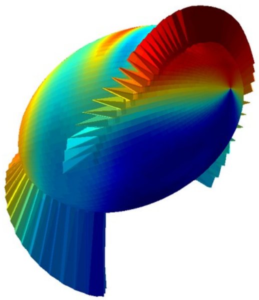
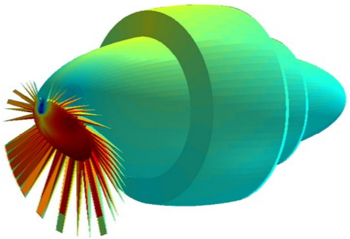
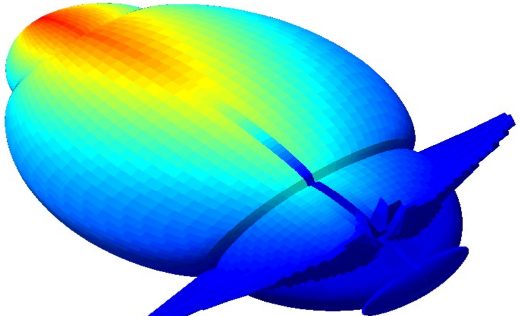
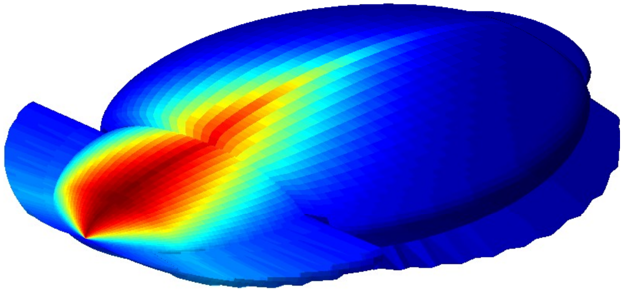




Lecture 5
Gradient Descent Learning





The Supervised Learning Problem

Input patterns $\bar{x}^k \in \mathbf{X} \subset \Re^{n+1} \quad k = 1, \ldots, M$

$$\bar{x}^k = \left[\, x_0^k \;\; x_1^k \;\; x_2^k \;\; \ldots \; x_n^k \,\right]^T$$

Output patterns $\bar{y}^k \in \mathbf{Y} \subset \Re^{l+1} \quad k = 1, \ldots, M$

$$\bar{y}^k = \left[\, y_1^k \;\; y_2^k \;\; \ldots \; y_l^k \,\right]^T$$

Training Set

Fit a model:

$\hat{\bar{y}} = f\left(\bar{x}; \bar{w}\right)$ $\quad \bar{w} = \left[\, w_1 \; w_2 \cdots w_N \,\right]$ are the parameters of the model

to minimize the loss function

$J(\bar{w}) = \sum_k J^k(\bar{w}) = \sum_k e\left(\hat{\bar{y}}^k, \bar{y}^k; \bar{w}\right)$ $\quad e(\hat{\bar{y}}, \bar{y}; w)$ is an error function

by adapting the parameters $\bar{w}$

Loss Functions

The three most commonly used loss functions are:

Absolute Error:

$$J_1(\overline{w}) = \sum_{k=1}^{M} \sum_{i=1}^{l} \left| y_i^k - \hat{y}_i^k \right| \qquad e_i^k \equiv y_i^k - \hat{y}_i^k \qquad J^k \equiv \sum_i \left| e_i^k \right| = \sum_i \left| y_i^k - \hat{y}_i^k \right|$$

Mean-Squared Error (MSE):

$$J_2(\overline{w}) = \sum_{k=1}^{M} \sum_{i=1}^{l} \left(y_i^k - \hat{y}_i^k \right)^2 \qquad e_i^k \equiv y_i^k - \hat{y}_i^k \qquad J^k \equiv \sum_i \left(e_i^k \right)^2 = \sum_i \left(y_i^k - \hat{y}_i^k \right)^2$$

Cross-Entropy Loss:

Used for binary (0/1) desired outputs and sigmoid actual outputs

$$J_{CE}(\overline{w}) = -\sum_{k=1}^{M} \sum_{i=1}^{l} \left(y_i^k \log \hat{y}_i^k + (1 - y_i^k) \log(1 - \hat{y}_i^k) \right)$$

The Error Surface

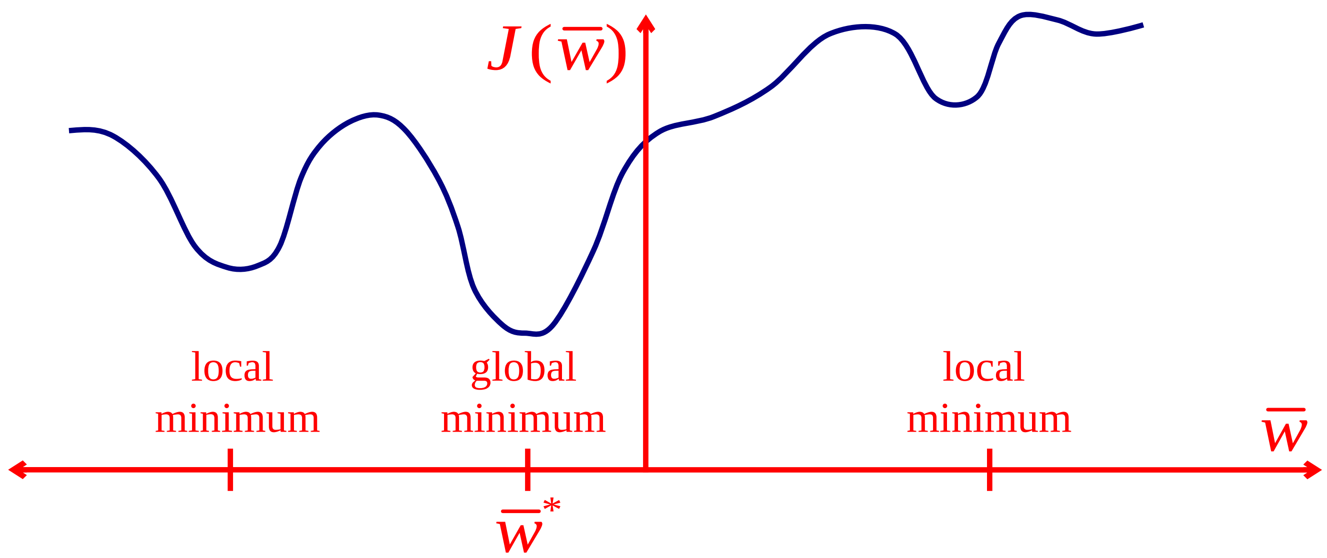
$J(\bar{w})$ defines a surface on the N-dimensional space of parameters $\bar{w}$

$\rightarrow$ **error surface**

Optimization $\leftrightarrow$ Finding the lowest point on this error surface

$\rightarrow$ **global minimum**.

For most non-trivial problems, the error surface is very complicated and has many local minima.



Iterative Optimization

1. Present data points from the training set one at a time to the model.
2. Calculate the loss for each data point.
3. Modify the parameters:

   a) After each step (on-line mode)

   or

   b) after going through all the data points (batch mode)

such that:

$J\left(\overline{w}(T+1)\right) < J\left(\overline{w}(T)\right)$ where T is the learning time step

Training Epoch:

Usually, one pass through all the training data. In some cases, only a subset of the data may be seen in any given epoch.

On-Line Mode:

a. Initialize $\overline{w} = \overline{w}(0)$

b. For $t = 1$ to L_{train} (some large integer)

- Present $\overline{x}(t) = \overline{x}^k \in X$ (thus k is the index of $\overline{x}(t)$ in X)
- Obtain $\hat{\overline{y}}(t) = f\left(\overline{x}(t); \overline{w}\right)$
- Calculate error $J(t) = J^k$
- Calculate parameter change $\Delta\overline{w}(t)$ based on $J(t)$
- Modify $\overline{w}$: $\overline{w}(t+1) = \overline{w}(t) + \Delta\overline{w}(t)$

c. Evaluate error. If ok, end else, repeat from b.

An alternative is to use a repeat…until instead of for so that the # of learning steps depends on the performance rather than a preset value.

Batch Mode:

t =fast dynamics (data points)

T =slow dynamics (epoch)

a. Initialize: $\bar{w} = \bar{w}(0) \qquad T = 0$

b. $\bar{\Delta}_{acc} = \bar{0}$ (same dimension as $\bar{w}$)

c. For $t = 1$ to L_{epoch} (number of data points in the epoch)

- Present $\bar{x}(t) = \bar{x}^k \in X$ (thus k is the index of $\bar{x}(t)$ in X)
- Obtain $\hat{\bar{y}}(t) = f\left(\bar{x}(t); \bar{w}\right)$
- Calculate error $J(t) = J^k$
- Calculate parameter change $\Delta\bar{w}(t)$ based on $J(t)$
- $\bar{\Delta}_{acc} = \bar{\Delta}_{acc} + \Delta\bar{w}(t)$

d. Modify $\bar{w}$: $\bar{w}' = \bar{w}(T) + \bar{\Delta}_{acc}$

e. $T = T + 1 \quad , \quad \bar{w}(T+1) = \bar{w}'$

f. Repeat from b. until done (or exhausted)

Gradient Descent Optimization

The main question in iterative optimization is: How to determine $\Delta\bar{w}$?

Assume that $J(\bar{w})$ is differentiable

Gradient of $J(\bar{w})$ is

$$\nabla J(\bar{w}) = \left[\frac{\partial J}{\partial w_1} \quad \frac{\partial J}{\partial w_2} \quad . \quad . \quad . \quad . \quad \frac{\partial J}{\partial w_N} \right]^T$$

where N = total # of parameters = dimension of parameter space

Optimum: $J(\bar{w}^*) \le J(\bar{w}) \quad \forall \; \bar{w} \quad \Rightarrow \quad \nabla J(\bar{w}^*) = 0$

Gradient Descent Rule: Change $\bar{w}$ in the direction of the negative gradient.

$$\bar{w}(T+1) = \bar{w}(T) + \Delta\bar{w}(T) = \bar{w}(T) - \eta \nabla J\left(\bar{w}(T)\right) \qquad \eta = \text{learning rate}$$

$$\Delta w_i(t) = -\eta \frac{\partial J}{\partial w_i}$$

To see why this works, expand $J\left(\overline{w}(T+1)\right)$ in a Taylor series.

$$J\left(\overline{w}(T+1)\right) \approx J\left(\overline{w}(T)\right) + \left[\nabla J\left(\overline{w}(T)\right)\right]^T \Delta\overline{w}(T) + \text{higher order terms.}$$

To lower LHS. $\left(\nabla J\right)^T \Delta\overline{w} \;<\; 0$

Since $(\nabla J)^T \Delta\overline{w} = \|\nabla J\|\|\Delta\overline{w}\|\cos\theta$ $\quad\theta$ is the angle between ∇J and $\Delta\overline{w}$

Loss is reduced most when cos = -1 $\Rightarrow \;\Delta\overline{w} \;\propto\; -\nabla J$

Intuitive Explanation:

- If you are trying to get to the bottom of a valley, the fastest way down is to go down the steepest slope at every step.
- The gradient vector points up the steepest slope at every point.
- Thus, going in the opposite direction of the gradient is the steepest way down.

Stochastic Gradient Descent (Online)

Note that, ideally, gradient descent training of a network requires:

$$\overline{w}(T+1) = \overline{w}(T) - \eta \nabla J\left(\overline{w}(T)\right)$$

where $\nabla J(\overline{w}(T)) = \nabla \sum_{k=1}^{m} J^k(\overline{w}(T))$ i.e., the gradient must be calculated over ***all*** data summed together, using weights from epoch T.

This can be problematic for very large data-sets

Solution: Train in on-line mode, selecting data points ***randomly*** from the data-set at each step, and updating immediately.

a. Initialize $\overline{w} = \overline{w}(0)$

b. For $t = 1$ to M (some large integer)

- Choose $\overline{x}(t) = x^k \in X$ randomly using some policy
- Obtain $\hat{y}(t) = f\left(\overline{w}^T \overline{x}(t)\right)$
- Calculate error $e(t)$ and local gradient $\nabla J^k(\overline{w}(t))$
- Modify $\overline{w}$: $\overline{w}(t+1) = \overline{w}(t) - \eta \nabla J^k(\overline{w}(t))$

c. Evaluate error. If ok, end else, repeat from b.

Stochastic gradient descent is used widely in applications of supervised learning to ***big data***, including ***deep learning***.

Stochastic Gradient Descent (Minibatch)

- Divide up each epoch into several minibatches of data (random or regular).
- Average gradient over each minibatch and adjust weights.

a. Initialize $\overline{w} = \overline{w}(0)$

b. For $T = 1$ to N_{epochs}

Set minibatch gradient $\nabla J_{mb} = 0$

For $t = 1$ to q

- Choose $\overline{x}(t) = x^k \in X$ randomly using some policy
- Obtain $\hat{\overline{y}}(t) = f\left(\overline{w}^T \overline{x}(t)\right)$
- Calculate error $e(t)$ and local gradient $\nabla J^k(\overline{w}(t))$
- Accumulate minibatch gradient $\nabla J_{mb} = \nabla J_{mb} + \nabla J^k(\overline{w}(t))$

Modify $\overline{w}$: $\overline{w}(t+1) = \overline{w}(t) - \eta \frac{1}{q} \nabla J_{mb}$

Obviously, this can also be run in *repeat …. until* mode instead of a *for* loop with a pre-specified number of epochs.

Active Management of Training Data

- Use examples on which error is greater.

- Use examples that are very different from those already used (to cover new regions of sample space)

- Randomize the order of presentation.

- Weight probability of presentation for different classes.

- Watch out for outliers that can really skew the weights catastrophically.

- Use multiple, overlapping training and testing sets.

Gradient Descent with Momentum

Original gradient descent update:

$$\Delta w_{ij}(t) = -\eta \frac{\partial J}{\partial w_{ij}}$$

Gradient descent with momentum:

$$\Delta w_{ij}(t) = -\eta \frac{\partial J}{\partial w_{ij}} + \alpha \, \Delta w_{ij}(t-1)$$

$$0 \le |\alpha| < 1$$

If $|\alpha| \geq 1$, the equation can become unstable.

What does momentum do?

if Δw_i has the same sign for several consecutive iterations

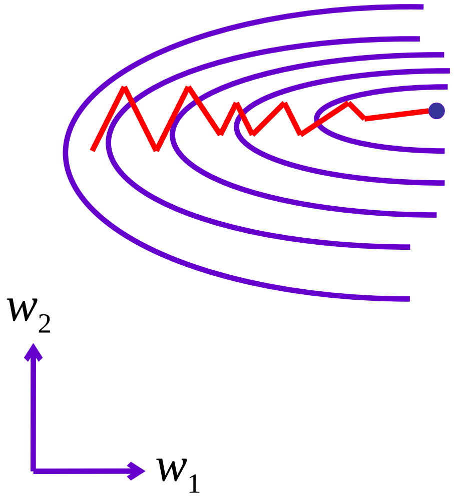
$\Rightarrow$ stable downhill direction for w_i

$\rightarrow$ momentum increases descent rate

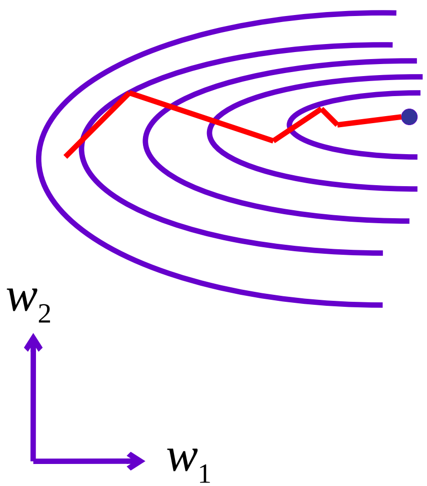
if Δw_i has alternating signs over consecutive iterations

$\Rightarrow$ w_i is not contributing much to descent

$\rightarrow$ momentum damps movement in w_i



Without momentum, Δw_2 is not contributing to progress. This is indicated by its oscillating sign.



With momentum, Δw_2 is damped and Δw_1 increased to find minimum faster

Momentum magnifies progress in directions of sustained descent and diminishes progress in other directions

The *Adam* (Adaptive Moment) Algorithm

Kingma & Ba (2015) ADAM: A Method for Stochastic Optimization, Proc. ICLR 2015

Parameters: Learning rate η , Biases β_1, β_2
Variables: 1st moment $\bar{m}(t), \bar{M}(t)$, 2nd moment $\bar{v}(t), \bar{V}(t)$
Weights: $\bar{w}(t) = \left[w_{ij}(t) \right]$

Initialize: $\bar{w}(0)$, $\bar{m}(0) = 0$, $\bar{v}(0) = 0$, $t = 0$

Elementwise product

while $\bar{w}$ not converged **do**

$t = t + 1$

$\bar{g}(t) = \nabla J(\bar{w}(t-1))$ current loss gradient

$\bar{m}(t) = \beta_1 \bar{m}(t-1) + (1-\beta_1)\bar{g}(t)$ update biased 1st moment

$\bar{v}(t) = \beta_2 \bar{v}(t-1) + (1-\beta_2)\left(\bar{g}(t) \odot \bar{g}(t)\right)$ update biased 2nd moment

raised to power *t*

$\bar{M}(t) = \bar{m}(t) / \left(1-\beta_1^t\right)$ compute unbiased 1st moment

$\bar{V}(t) = \bar{v}(t) / \left(1-\beta_2^t\right)$ compute unbiased 2nd moment

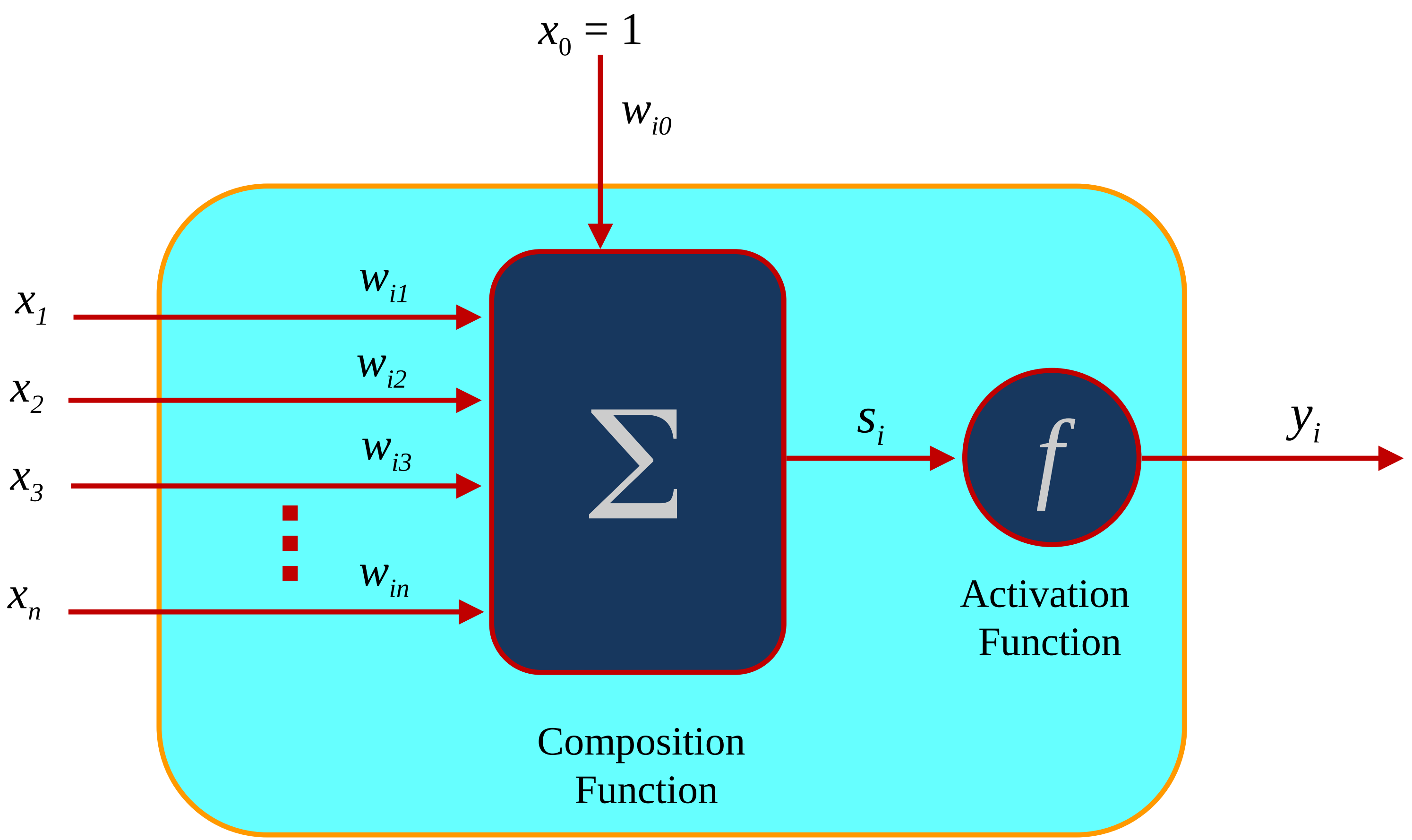
Elementwise ratio

$\Delta\bar{w}(t) = -\eta \bar{M}(t) / \left(\sqrt{\bar{V}(t)} + \varepsilon\right)$ update weights

end **while**
return $\bar{w}(t)$

Recommended choices: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$

Simple Neuron Model



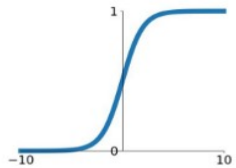
$$s_i = \sum_j w_{ij} x_j$$

$$y_i = f(s_i)$$

Activation Functions

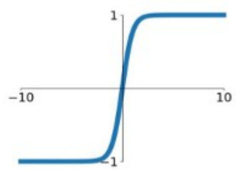
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



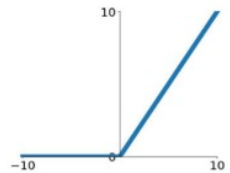
tanh

$\tanh(x)$



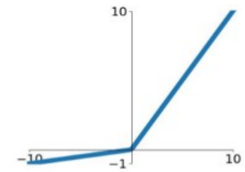
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

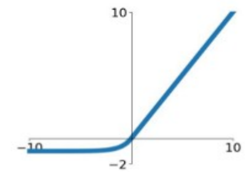


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$



Source: https://www.quora.com/

Why use ReLU?

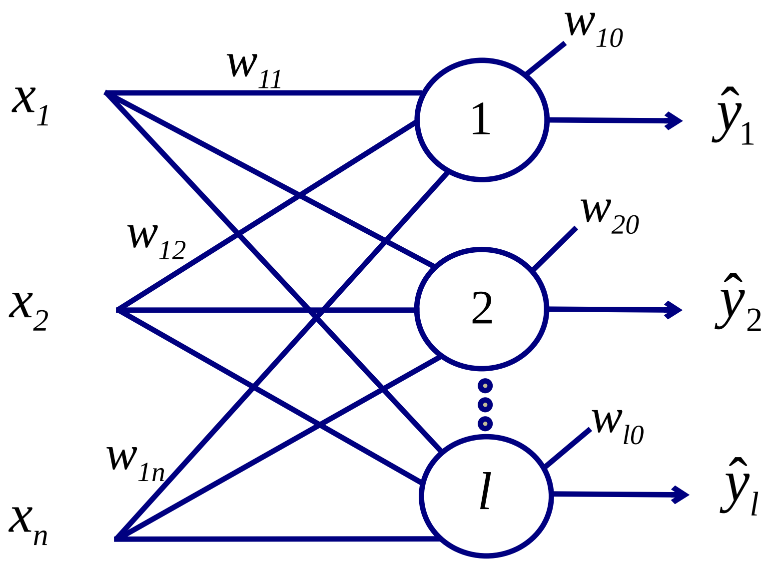
- It is nonlinear and monotonic (both useful/necessary)
- Its gradient does not vanish on the positive side (unlike sigmoids).
- It gives infinite range of output values (0 to +∞)

But

- ReLU gradient vanishes on negative side → Leaky ReLU
- ReLU is non-differentiable at 0 → ELU

The Least Mean Square (LMS) Algorithm

Consider a single layer of neurons



For data point $\left(\bar{x}^q, \bar{y}^q\right)$

Define:

$$J^q = \frac{1}{2}\sum_{i=1}^{l}\left(e_i^q\right)^2 = \frac{1}{2}\sum_{i=1}^{l}\left(y_i^q - \hat{y}_i^q\right)^2$$

$$J = \sum_{q=1}^{M} J^q$$

$$J = \frac{1}{2}\sum_{q=1}^{M}\sum_{i=1}^{l}\left[y_i^q - f\left(\sum_{j=0}^{n} w_{ij} x_j^q\right)\right]^2$$

Input patterns $\bar{x}^q \in X \subset \Re^{n+1} \qquad q = 1, \ldots, M$

$$\bar{x}^q = \begin{bmatrix} 1 & x_1^q & x_2^q & \ldots & x_n^q \end{bmatrix}^T$$

bias

Output patterns $\bar{y}^q \in Y \subset \Re^l$

Note that y_i are not necessarily 0,1

Weight matrix $\bar{w} = \begin{bmatrix} \bar{w}_1 & \bar{w}_2 & \ldots & \bar{w}_l \end{bmatrix}^T$

$$\bar{w}_i = \begin{bmatrix} w_{i0} & w_{i1} & w_{i2} \ldots & w_{in} \end{bmatrix}^T$$

bias weight

$$\bar{w}_i \in \Re^{n+1}$$

$$\bar{w} = \begin{bmatrix} w_{10} & w_{11} & w_{12} & . & . & . & . & . & . & w_{1n} \\ w_{20} & w_{21} & w_{22} & . & . & . & . & . & . & w_{2n} \\ . \\ . \\ . \\ . \\ w_{l0} & w_{l1} & w_{l2} & & & & & & & w_{ln} \end{bmatrix} = l \times (n+1) \equiv N$$

$\overline{x}(t) \equiv$ pattern presented at time step t

$$x(t) \in X$$

$\overline{y}(t) \equiv$ desired response at step t

$$\overline{y}(t) \in Y$$

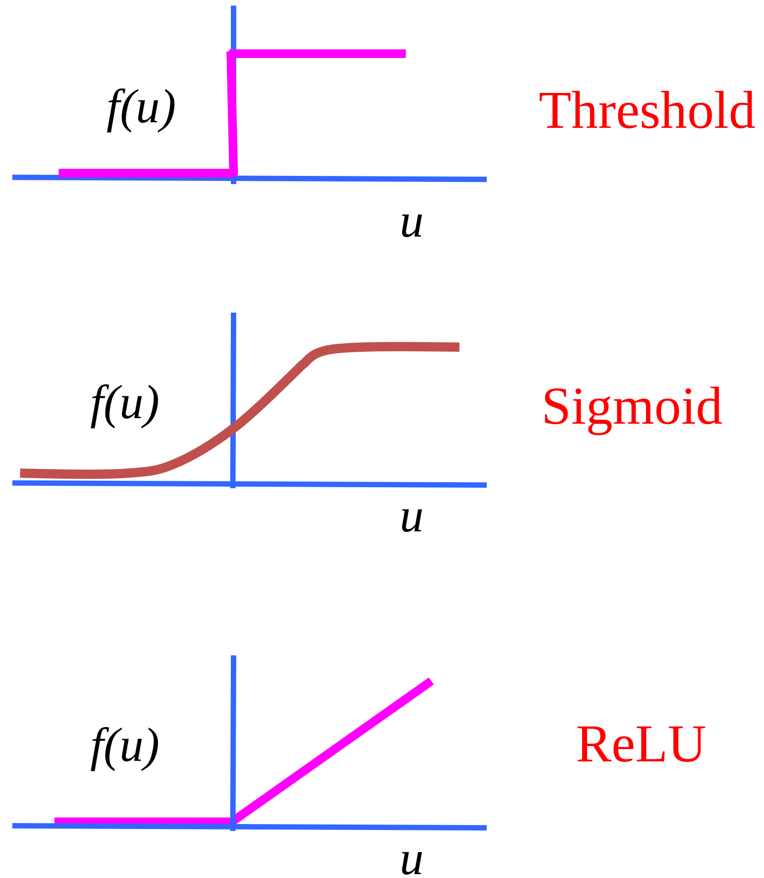
$\overline{w}(t) \equiv$ weight matrix at step t

$\hat{\overline{y}}(t) \equiv$ actual output at step t

$$\hat{\overline{y}}(t) = \left[\hat{y}_1(t) \quad \hat{y}_2(t) \quad \dots. \quad \hat{y}_l(t)\right]^T$$

$$\hat{y}_i(t) = f\left(\overline{w}_i^T(t) \ \overline{x}(t)\right)$$

Typical $f(u)$



if $\bar{x}^q$ is presented to the network

$$\Delta \bar{w} = -\eta \nabla J^q$$

$$\nabla J^q = \begin{bmatrix} \dfrac{\partial J^q}{\partial w_{10}} \\ . \\ . \\ . \\ \dfrac{\partial J^q}{\partial w_{l\,n}} \end{bmatrix}$$

so

$$\Delta w_{ij} = -\eta \frac{\partial J^q}{\partial w_{ij}}$$

$$\frac{\partial J^q}{\partial w_{ij}} = \frac{\partial \frac{1}{2} \sum_p \left(e_p^q\right)^2}{\partial w_{ij}}$$

p = index of units.
$p = 1, \ldots, l$

$$= \frac{1}{2} \sum_p \frac{\partial \left(e_p^q\right)^2}{\partial w_{ij}}$$

superposition of derivatives

$$= \frac{1}{2} \frac{\partial \left(e_i^q\right)^2}{\partial w_{ij}}$$

since w_{ij} affects only e_i^k

$$\frac{\partial J^k}{\partial w_{ij}} = \frac{1}{2}\frac{\partial\left(e_i^q\right)^2}{\partial w_{ij}}$$

$$= \frac{1}{2}\frac{\partial\left(e_i^q\right)^2}{\partial e_i^q}\frac{\partial e_i^q}{\partial w_{ij}}$$

$$= e_i^q \frac{\partial e_i^q}{\partial \hat{y}_i^q}\frac{\partial \hat{y}_i^q}{\partial w_{ij}}$$

$$= e_i^q \cdot \frac{\partial e_i^q}{\partial \hat{y}_i^q} \cdot \frac{\partial \hat{y}_i^q}{\partial s_i^q} \cdot \frac{\partial s_i^q}{\partial w_{ij}}$$

$$= e_i^q \cdot (-1) \cdot f'\left(s_i^q\right) \cdot x_j^q$$

$$\boxed{\begin{aligned}\frac{\partial J^q}{\partial w_{ij}} &= -e_i^q f'\left(s_i^q\right) x_j^q \\ &= -\left(y_i^q - \hat{y}_i^q\right) f'\left(s_i^q\right) x_j^q\end{aligned}}$$

$$\boxed{\Delta w_{ij} = \eta\left(y_i^q - \hat{y}_i^q\right) f'\left(s_i^q\right) x_j^q}$$

$$\frac{\partial J^k}{\partial w_{ij}} = \frac{1}{2}\frac{\partial\left(e_i^q\right)^2}{\partial w_{ij}}$$

$$= \frac{1}{2}\frac{\partial\left(e_i^q\right)^2}{\partial e_i^q}\frac{\partial e_i^q}{\partial w_{ij}}$$

$$= e_i^q \frac{\partial e_i^q}{\partial \hat{y}_i^q}\frac{\partial \hat{y}_i^q}{\partial w_{ij}}$$

$$= \boxed{e_i^q \cdot \frac{\partial e_i^q}{\partial \hat{y}_i^q}} \cdot \frac{\partial \hat{y}_i^q}{\partial s_i^q} \cdot \frac{\partial s_i^q}{\partial w_{ij}}$$

$$= e_i^q \cdot (-1) \cdot f'\left(s_i^q\right) \cdot x_j^q$$

Note that the error terms are dissociated from others, so we can use any differentiable error metric without changing the algorithm.

$$\frac{\partial J^q}{\partial w_{ij}} = -e_i^q f'\left(s_i^q\right) x_j^q$$

$$= -\left(y_i^q - \hat{y}_i^q\right) f'\left(s_i^q\right) x_j^q$$

$$\Delta w_{ij} = \eta\left(y_i^q - \hat{y}_i^q\right) f'\left(s_i^q\right) x_j^q$$

$$\frac{\partial J^k}{\partial w_{ij}} = \frac{1}{2}\frac{\partial\left(e_i^q\right)^2}{\partial w_{ij}}$$

$$= \frac{1}{2}\frac{\partial\left(e_i^q\right)^2}{\partial e_i^q}\frac{\partial e_i^q}{\partial w_{ij}}$$

$$= e_i^q \frac{\partial e_i^q}{\partial \hat{y}_i^q}\frac{\partial \hat{y}_i^q}{\partial w_{ij}}$$

$$= e_i^q \cdot \frac{\partial e_i^q}{\partial \hat{y}_i^q} \cdot \frac{\partial \hat{y}_i^q}{\partial s_i^q} \cdot \frac{\partial s_i^q}{\partial w_{ij}}$$

$$= e_i^q \cdot (-1) \cdot \boxed{f'\left(s_i^q\right)} \cdot x_j^q$$

And any activation function – as long as it is differentiable

$$\boxed{\begin{aligned}\frac{\partial J^q}{\partial w_{ij}} &= -e_i^q f'\left(s_i^q\right) x_j^q \\ &= -\left(y_i^q - \hat{y}_i^q\right) f'\left(s_i^q\right) x_j^q\end{aligned}}$$

$$\boxed{\Delta w_{ij} = \eta\left(y_i^q - \hat{y}_i^q\right) f'\left(s_i^q\right) x_j^q}$$

$$\frac{\partial J^k}{\partial w_{ij}} = \frac{1}{2}\frac{\partial\left(e_i^q\right)^2}{\partial w_{ij}}$$

$$= \frac{1}{2}\frac{\partial\left(e_i^q\right)^2}{\partial e_i^q}\frac{\partial e_i^q}{\partial w_{ij}}$$

$$= e_i^q \frac{\partial e_i^q}{\partial \hat{y}_i^q}\frac{\partial \hat{y}_i^q}{\partial w_{ij}}$$

$$= e_i^q \cdot \frac{\partial e_i^q}{\partial \hat{y}_i^q} \cdot \frac{\partial \hat{y}_i^q}{\partial s_i^q} \cdot \frac{\partial s_i^q}{\partial w_{ij}}$$

$$= e_i^q \cdot (-1) \cdot f'\left(s_i^q\right) \cdot x_j^q$$

$$\frac{\partial J^q}{\partial w_{ij}} = -e_i^q f'\left(s_i^q\right) x_j^q$$

$$= -\left(y_i^q - \hat{y}_i^q\right) f'\left(s_i^q\right) x_j^q$$

$$\Delta w_{ij} = \eta \underbrace{\boxed{\left(y_i^q - \hat{y}_i^q\right) f'\left(s_i^q\right) x_j^q}}_{-\nabla J(\overline{w})}$$

Sigmoid *f(u)*

$$f(u) = \frac{1}{1+e^{-u}}$$

$$\frac{df}{du} = \frac{e^{-u}}{\left(1+e^{-u}\right)^2} = \frac{e^{-u}}{1+e^{-u}} \cdot \frac{1}{1+e^{-u}}$$

$$= \left(1 - f(u)\right) f(u)$$

$$\boxed{\frac{\partial J^q}{\partial w_{ij}} = -\left(y_i^q - \hat{y}_i^q\right) \hat{y}_i^q \left(1 - \hat{y}_i^q\right) x_j^q}$$

defining $\delta_i^q = \hat{y}_i^q \left(1 - \hat{y}_i^q\right) \left(y_i^q - \hat{y}_i^q\right)$

$$\boxed{\Delta w_{ij} = \eta \delta_i^q x_j^q}$$

Linear *f(u)*

$$\hat{y}_i^q = s_i^q = \sum_{j=0}^{n} w_{ij} x_j^q$$

$$f'\left(s_i^q\right) = 1$$

$$\boxed{\frac{\partial J^q}{\partial w_{ij}} = -\left(y_i^q - \hat{y}_i^q\right) x_j^q}$$

$\Rightarrow$ LMS for linear neurons = perceptron learning

$$\boxed{\Delta w_{ij} = \eta\left(y_i^q - \hat{y}_i^q\right) x_j^q}$$

which is the perceptron learning rule.