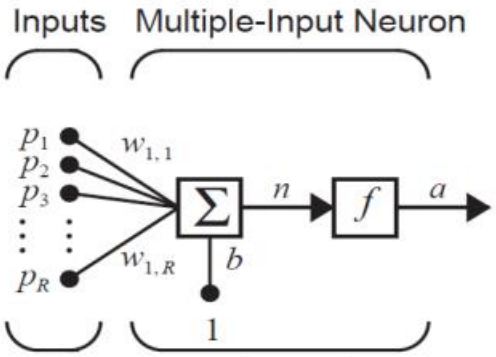
- Brain vs. Computer:
  1. Neurons respond more slowly compared to Artificial Neurons.
  2. Neurons perform massively parallelized computation compared to Artificial Neurons

  Brain wins because although it is slower in terms of single Neuron performance, it is much more parallelized.

- Why use ANNs?
  1. Some specific problems require massively parallel and adaptive processing.
  2. They can be used to simulate components of the human (or animal) brains, thereby giving us insight into natural information processing.

## Multiple-Input Neuron

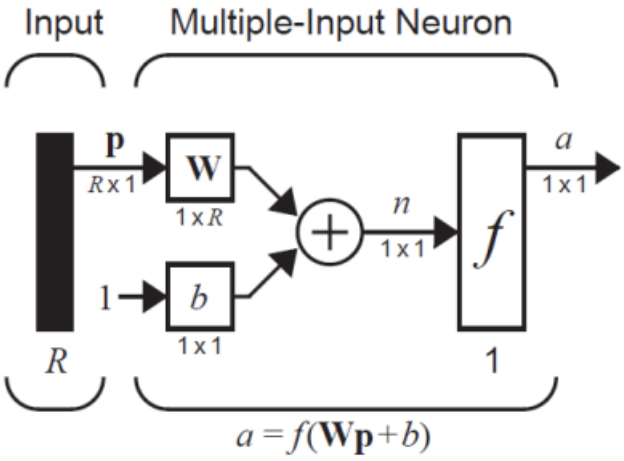



$$a = f(\mathbf{W}\mathbf{p} + b)$$

$$n = w_{1,1}p_1 + w_{1,2}p_2 + \cdots + w_{1,R}p_R + b$$

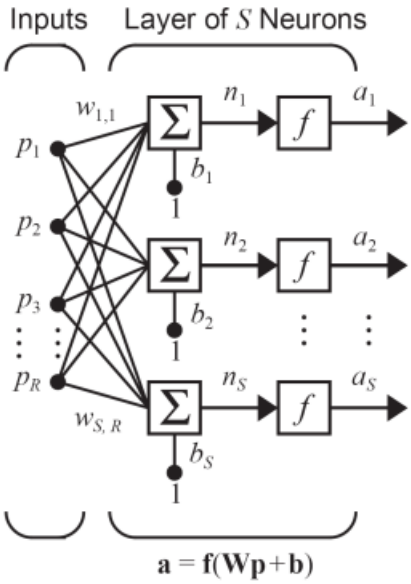
$$n = \mathbf{W}\mathbf{p} + b$$

## Abbreviated Notation

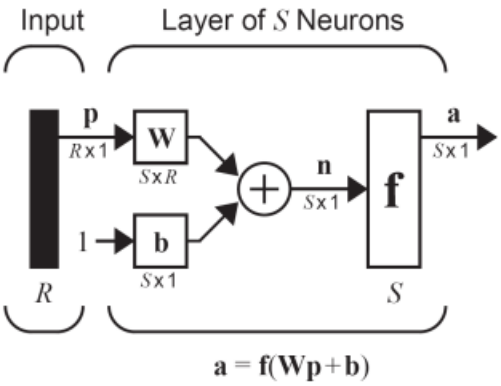



$$a = f(\mathbf{W}\mathbf{p} + b)$$

# Layer of Neurons



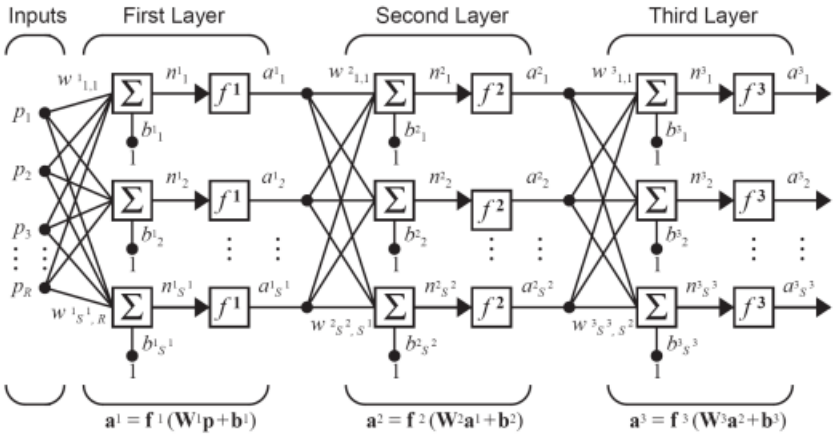

# Abbreviated Notation




$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$

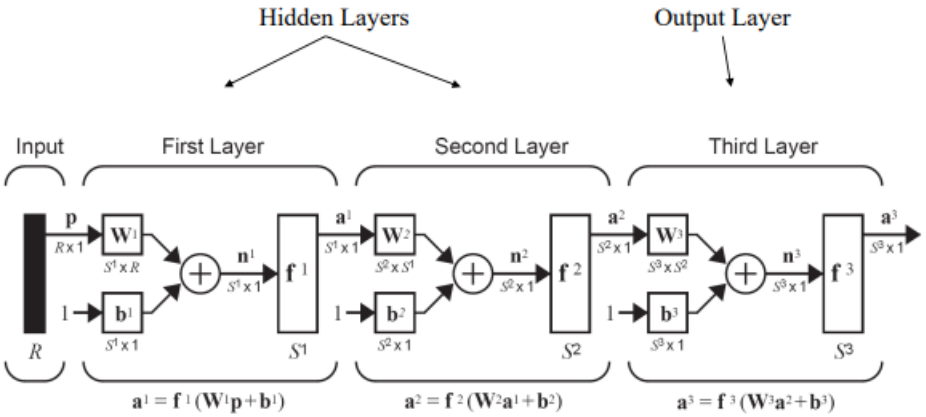
$$\mathbf{p} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_R \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_S \end{bmatrix} \quad \mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_S \end{bmatrix}$$

# Multilayer Network




$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{f}^2(\mathbf{W}^2\mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1)+\mathbf{b}^2)+\mathbf{b}^3)$$

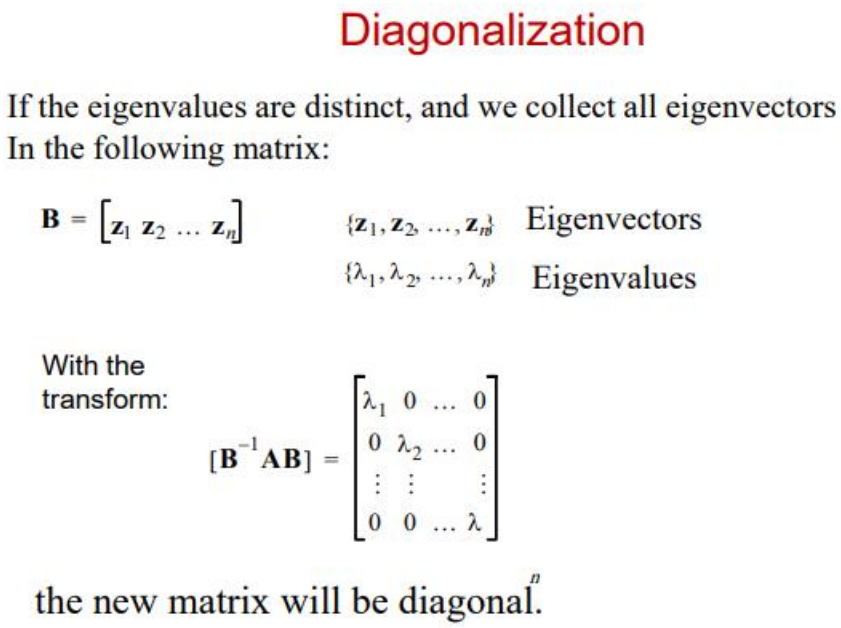
# Abbreviated Notation




$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{f}^2(\mathbf{W}^2\mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1)+\mathbf{b}^2)+\mathbf{b}^3)$$

- Network Architecture Design:
  1. Number of network inputs = number of problem inputs
  2. Number of neurons in output layer = number of problem outputs
  3. Output layer transfer function choice at least partly determined by problem specification of the outputs (for example a probability output would probably use sigmoid for 2 class and softmax for multiclass)
- Linear Algebra Review:
  - Dot product of two column vectors: $x.y = \sum_{i=1}^{n} x_i y_i = x^T y$
  - Two vectors are orthogonal iff $x^T y = x.y = 0$
  - P-Norm of a vector: $\|x\|_p = (\sum_{i=1}^{n} |x_i|^p)^{\frac{1}{p}}$. Specifically, the 2-Norm or Standard Euclidean Norm: $\|x\| = (x^T x)^{\frac{1}{2}} = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$

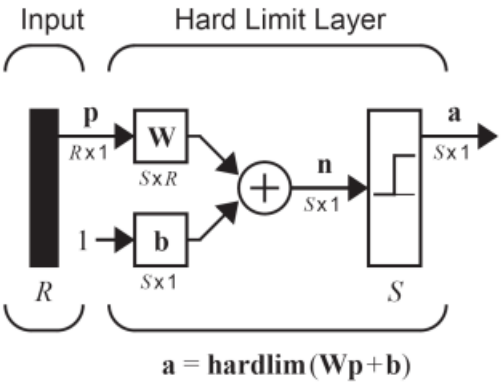
- $\cos(\theta) = \frac{x.y}{\|x\|\|y\|}, where\ \theta\ is\ the\ is\ the\ angle\ between\ two\ vectors$
- Eigenvalues and Eigenvectors:
    - An eigenvector $u_i$ and its corresponding eigenvalue, $\lambda_i$, of a matrix $A \in R^{n \times n}$ satisfies: $Au_i = \lambda_i u_i$
    - Computing the Eigenvalues:
      $Az = \lambda z \rightarrow [A - \lambda I]z = 0 \rightarrow z = 0\ is\ a\ trivial\ solution\ therefore\ |[A - \lambda I]| = 0$
      Solutions to this give all eigenvalues $\lambda$.
    - Computing the Eigenvectors:
      Plug the $\lambda$s from the previous step into $[A - \lambda I]z = 0$ and solve for $z$.
    - Diagonalization through Eigenvalues and Eigenvectors:




**Note that the diagonals of the matrix are equal to the Eigenvalues of $A$.**

- Types of Learning:
    - Supervised Learning (for each input there's a label, we would like to predict the label given a never-seen-before input.)
    - Unsupervised Learning (there are only inputs and no labels. The model is to perform a task on the inputs without any prior knowledge of possible labels.)
    - Reinforcement Learning (A reward-based algorithm where there are no specific labels but each action the intelligent agent takes affects the environment and the environment rewards or punishes the agent depending on how closer or further it is now from its target).

# Perceptron Architecture



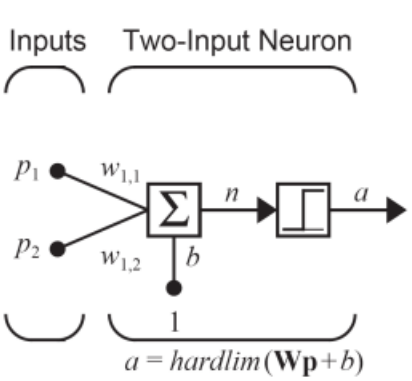

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$

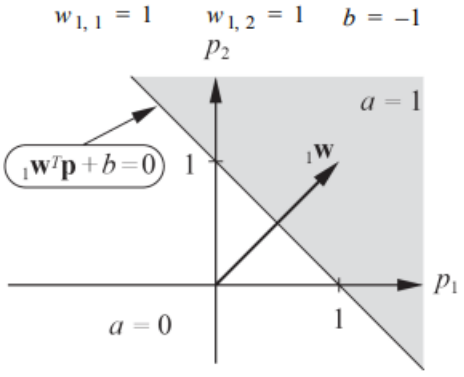
$$_i\mathbf{w} = \begin{bmatrix} w_{i,1} \\ w_{i,2} \\ \vdots \\ w_{i,R} \end{bmatrix} \qquad \mathbf{W} = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \\ \vdots \\ {}_S\mathbf{w}^T \end{bmatrix}$$

$_i\mathbf{w}$ is elements of $i$th row of $\mathbf{W}$

- $$a_i = hardlim(n_i) = hardlim({}_i\mathbf{w}^T\mathbf{p} + b_i)$$

# Single-Neuron Perceptron




$$_1\mathbf{w}^T\mathbf{p} + b = 0$$

$$a = hardlim({}_1\mathbf{w}^T\mathbf{p} + b) = hardlim(w_{1,1}p_1 + w_{1,2}p_2 + b)$$

- **Weight vector always orthogonal to the decision boundary (why?)**
  - Consider two points on the decision boundary $p_1$ and $p_2$. The vector $p_1 - p_2$ is parallel to the decision boundary.
  - However, we know that for both $p_1$ and $p_2$ we have: $w^T p_i + b = 0$.
  - $\begin{cases} w^T p_1 + b = 0 \\ w^T p_2 + b = 0 \end{cases} \rightarrow subtract: w^T(p_1 - p_2) = 0 \rightarrow w\ is\ normal\ to\ p_1 - p_2.$
  - And $p_1 - p_2$ is parallel to the decision boundary line therefore $w$ is also normal to the decision boundary.

- Another way to think about this: without loss of generality subtract $b$ from every $y$-axis value on the plane so that the decision boundary now passes from the point of origin in the transformed cartesian plane (note that this transformation is easily revertible).
- In the new transformed space, we have $w^T p = 0$ which directly proves that $w$ is normal to the decision boundary. Because $w$ is normal in this transformed space, it is also normal in every other space through a linear transform (for example adding $b$ to $y$-axis values). Therefore, the proof also holds for the original space.

- **And it always points to the side that represents 1 in the hardlim transfer function.**
- **Study a couple of perceptron design problems (from the textbook or slides)**
- In Multi-Neuron Perceptrons each neuron will have its own decision boundary. So it can classify input vectors into two categories. Meaning that a multi-neuron perceptron can classify input vectors into $2^s$ categories where $s$ is the number of neurons.

## Unified Learning Rule

مجموعه قوانین

If $t = 1$ and $a = 0$, then ${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}$

If $t = 0$ and $a = 1$, then ${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}$

If $t = a$, then ${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old}$

$$e = t - a$$

If $e = 1$, then ${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}$

If $e = -1$, then ${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}$

If $e = 0$, then ${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old}$

Unified LR

$${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + e\mathbf{p} = {}_1\mathbf{w}^{old} + (t-a)\mathbf{p}$$

$$b^{new} = b^{old} + e$$

A bias is a weight with an input of 1.

25

## Multiple-Neuron Perceptrons

To update the $i$th row of the weight matrix:

$${}_i\mathbf{w}^{new} = {}_i\mathbf{w}^{old} + e_i\mathbf{p}$$

$$b_i^{new} = b_i^{old} + e_i$$

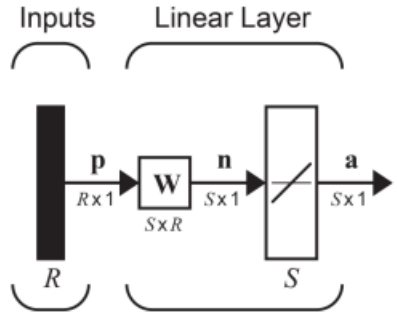
Matrix form:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^T$$

$$\mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e}$$

- Novikoff proves that **given samples from two linearly separable classes**, the perceptron algorithm terminates in finitely many steps and does a **perfect classification** with **no regards** to the initial random **non-zero** weight vector $w_0$.
- However, a perceptron can **ONLY** classify input vectors that are linearly separable since its decision boundary is always a single line (or an ensemble of lines in the case of a multi-neuron perceptron).

# Linear Associator




- LA introduced by Kohonen and Anderson independently.
- LA is an example of Associative Memory

$$\mathbf{a} = \mathbf{purelin}(\mathbf{Wp})$$

$$\mathbf{a} = \mathbf{Wp} \qquad a_i = \sum_{j=1}^{R} w_{ij} p_j$$

Training Set:

- $\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$



- Pattern Association: an associative memory learns by association:

$$if\ p = p_q \rightarrow a = t_q\ for\ q = 1, 2, \cdots, Q\ and\ if\ p = p_q + \delta \rightarrow a = t_q$$

In simple words, an associative memory can associate a specific input with a specific output. If that input is slightly changed, we expect the target to still be the same as before.

- **Auto-Association:** If $t_q = p_q$, i.e., the network is expected to "memorize" a set of patterns to be able to later reconstruct them.
- **Hetero-Association:** If $t_q \neq p_q$, i.e., each input is paired with an arbitrary target that is not the input itself.

# Hebb Rule (1)

$$w_{ij}^{new} = w_{ij}^{old} + \alpha\, f_i(a_{iq}) g_j(p_{jq})$$

- $p_{jq}$ is the $j$th element of the $q$th input vector $\mathbf{p}_q$.
- $a_{iq}$ is the $i$th element of the network output when the $q$th input vector is presented to the network.
- $\alpha$ is a positive constant, called learning rate.
- Changes in $w_{ij}$ is proportional to a product of functions of the activities on either side of synapse.

# Hebb Rule (2)

Simplified Form: $w_{ij}^{new} = w_{ij}^{old} + \alpha a_{iq} p_{jq}$

- This extends Hebb's postulate (we also increase the weight when both $\mathbf{p}_j$ and $\mathbf{a}_i$ negative and decrease when one of them is negative).
- The above rule is an unsupervised learning rule (Ch 15).

Supervised Form: $w_{ij}^{new} = w_{ij}^{old} + t_{iq} p_{jq}$

- $t_{iq}$ is the $i$th element of the $q$th target vector $\mathbf{t}_q$. ($\alpha$=1 for simplicity.)
- We are telling the algorithm what the network should do, rather than what it is currently doing.

# Batch Operation

Matrix Form of Rule: $\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{t}_q \mathbf{p}_q^T$

When each of the Q input/output pair are applied once we have:

$$\mathbf{W} = \mathbf{t}_1\mathbf{p}_1^T + \mathbf{t}_2\mathbf{p}_2^T + \cdots + \mathbf{t}_Q\mathbf{p}_Q^T = \sum_{q=1}^{Q} \mathbf{t}_q\mathbf{p}_q^T \qquad \text{(Zero Initial Weights)}$$

Matrix Form:

$$\mathbf{W} = \begin{bmatrix} \mathbf{t}_1 & \mathbf{t}_2 & \cdots & \mathbf{t}_Q \end{bmatrix} \begin{bmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \\ \vdots \\ \mathbf{p}_Q^T \end{bmatrix} = \mathbf{T}\mathbf{P}^T$$

$$\mathbf{P} = \begin{bmatrix} \mathbf{p}_1 & \mathbf{p}_2 & \cdots & \mathbf{p}_Q \end{bmatrix}$$

$$\mathbf{T} = \begin{bmatrix} \mathbf{t}_1 & \mathbf{t}_2 & \cdots & \mathbf{t}_Q \end{bmatrix}$$

# Performance Analysis

Hebbian learning for the linear Associator. Assume $\mathbf{p}_q$ vectors are *orthonormal*. If $\mathbf{p}_k$ is input to the network, the output will be:

$$\mathbf{a} = \mathbf{W}\mathbf{p}_k = \left(\sum_{q=1}^{Q} \mathbf{t}_q \mathbf{p}_q^T\right)\mathbf{p}_k = \sum_{q=1}^{Q} \mathbf{t}_q(\mathbf{p}_q^T\mathbf{p}_k)$$

Case I, since input patterns are orthonormal.

$$(\mathbf{p}_q^T\mathbf{p}_k) = 1 \quad q = k$$
$$= 0 \quad q \neq k$$

Therefore the network output equals the target: $\mathbf{a} = \mathbf{W}\mathbf{p}_k = \mathbf{t}_k$

Case II, input patterns are normalized, but not orthogonal.

$$\mathbf{a} = \mathbf{W}\mathbf{p}_k = \mathbf{t}_k + \boxed{\sum_{q \neq k} \mathbf{t}_q(\mathbf{p}_q^T\mathbf{p}_k)} \quad \text{Error}$$

The magnitude of the error will depend on the amount of correlation between the prototype input patterns.

9

- In order to reduce this error, we can employ the Pseudoinverse Rule:
  - Task of the LA is to produce target $t_q$ given input $p_q$. Or:
    $Wp_q = t_q, q = 1, 2, \ldots, Q$
    If $W$ doesn't exist such that these equations are exactly and simultaneously satisfied, then we can hope approximately satisfy them. In order to do so we define a performance index to be minimized:

$$F(W) = \sum_{q=1}^{Q} \|t_q - Wp_q\|^2$$

If the input vectors are not orthonormal, $F(W)$ will not be zero and Hebb's Rule does not guarantee that the resulting weights minimize $F(W)$. In order to obtain $W$ that minimizes $F(W)$ we use the Pseudoinverse Rule.
Vectorized notation:

$$\mathbf{T} = [\mathbf{t}_1\ \mathbf{t}_2\ \ldots\ \mathbf{t}_Q] \qquad \mathbf{P} = [\mathbf{p}_1\ \mathbf{p}_2\ \ldots\ \mathbf{p}_Q]$$

$$F(\mathbf{W}) = \|\mathbf{T} - \mathbf{W}\mathbf{P}\|^2 = \|\mathbf{E}\|^2$$

$$\|\mathbf{E}\|^2 = \sum_i \sum_j e_{ij}^2$$

Note that if $WP = T$ has a solution for $W$, $F(W)$ can be made zero. In other words, $F(W)$ can be made zero if $P$ is invertible so that:

$$W = TP^{-1}$$

But this doesn't happen often because $P$ is very rarely a square matrix. However, when an inverse does not exist it is proven that the pseudoinverse of $P$ minimized $F(W)$ or:

$$W = TP^{+}$$

- **IMPORTANT**

If **P** is RxQ and R>Q and the columns of **P** are independent, then the pseudoinverse can be computed by

$$\mathbf{P}^{+}=(\mathbf{P}^{T}\mathbf{P})^{-1}\mathbf{P}^{T}$$

If **P** is RxQ and R<Q and the columns of **P** are independent, then the pseudoinverse can be computed by

$$\mathbf{P}^{+}=\mathbf{P}^{T}(\mathbf{P}\mathbf{P}^{T})^{-1}$$

- If the input patterns are orthonormal $PP^T$ and $P^TP$ will both be $I$ (depending on if the #cols > #rows or not). And hence $P^+ = P^T$. Therefore, Pseudoinverse Rule and Hebb's Rule are equivalent for orthonormal inputs.
- Variations of Hebbian Learning:
  - $Basic\ Rule: W^{new} = W^{old} + t_q p_q^T$
  - $Learning\ Rate: W^{new} = W^{old} + \alpha t_q p_q^T$
  - $Forgetting\ Factor: W^{new} = W^{old} + \alpha t_q p_q^T - \gamma W^{old} = (1-\gamma)W^{old} + \alpha t_q p_q^T$
    As $\gamma$ approaches one, this rule quickly forgets old inputs and remembers only the most recent patterns. **This keeps the weight matrix from growing unboundedly.**
  - $Delta\ Rule: W^{new} = W^{old} + \alpha(t_q - a_q)p_q^T$
    This is also known as the **Widrow-Hoff algorithm**. The delta rule adjusts the weights so as to minimize the **mean square error**. **For this reason, it produces the same results as the pseudoinverse rule, which also minimizes the sum of squares of errors.**
    However, the advantage of the delta rule is that it can perform updates after each new input pattern, whereas the pseudo-inverse rule computes the weights only after ALL the input/target pairs are known. This sequential updating makes the delta rule more adaptable to a changing environment.

$$Unsupervised\ Delta\ Rule: W^{new} = W^{old} + \alpha a_q p_q^T$$

- Any arbitrary function can be approximated around a certain point $x = x^*$ with a finite number of its **Taylor Expansion** terms.

$$F(x) = F(x^*) + \frac{d}{dx}F(x)\Big|_{x=x^*}(x-x^*) + \frac{1}{2}\frac{d^2}{dx^2}F(x)\Big|_{x=x^*}(x-x^*)^2 + \cdots + \frac{1}{n!}\frac{d^n}{dx^n}F(x)\Big|_{x=x^*}(x-x^*)^n + \cdots$$

# Vector Case

$$F(\mathbf{x}) = F(x_1, x_2, \ldots, x_n)$$

$$F(\mathbf{x}) = F(\mathbf{x}^*) + \frac{\partial}{\partial x_1} F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}(x_1 - x_1^*) + \frac{\partial}{\partial x_2} F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}(x_2 - x_2^*)$$

$$+ \cdots + \frac{\partial}{\partial x_n} F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}(x_n - x_n^*) + \frac{1}{2}\frac{\partial^2}{\partial x_1^2} F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}(x_1 - x_1^*)^2$$

$$+ \frac{1}{2}\frac{\partial^2}{\partial x_1 \partial x_2} F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}(x_1 - x_1^*)(x_2 - x_2^*) + \cdots$$

# Matrix Form

$$F(\mathbf{x}) = F(\mathbf{x}^*) + \nabla F(\mathbf{x})^T\Big|_{\mathbf{x}=\mathbf{x}^*}(\mathbf{x} - \mathbf{x}^*)$$

$$+ \frac{1}{2}(\mathbf{x} - \mathbf{x}^*)^T \nabla^2 F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}(\mathbf{x} - \mathbf{x}^*) + \cdots$$

Gradient

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial x_n} F(\mathbf{x}) \end{bmatrix}$$

Hessian

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2}{\partial x_1^2} F(\mathbf{x}) & \frac{\partial^2}{\partial x_1 \partial x_2} F(\mathbf{x}) & \cdots & \frac{\partial^2}{\partial x_1 \partial x_n} F(\mathbf{x}) \\ \frac{\partial^2}{\partial x_2 \partial x_1} F(\mathbf{x}) & \frac{\partial^2}{\partial x_2^2} F(\mathbf{x}) & \cdots & \frac{\partial^2}{\partial x_2 \partial x_n} F(\mathbf{x}) \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2}{\partial x_n \partial x_1} F(\mathbf{x}) & \frac{\partial^2}{\partial x_n \partial x_2} F(\mathbf{x}) & \cdots & \frac{\partial^2}{\partial x_n^2} F(\mathbf{x}) \end{bmatrix}$$

nnd8ts2



# Directional Derivatives

First derivative (slope) of $F(\mathbf{x})$ along $x_i$ axis: $\partial F(\mathbf{x})/\partial x_i$

($i$th element of gradient)

Second derivative (curvature) of $F(\mathbf{x})$ along $x_i$ axis: $\partial^2 F(\mathbf{x})/\partial x_i^2$

($i,i$ element of Hessian)

First derivative (slope) of $F(\mathbf{x})$ along vector $\mathbf{p}$: $\dfrac{\mathbf{p}^T \nabla F(\mathbf{x})}{\|\mathbf{p}\|}$

Second derivative (curvature) of $F(\mathbf{x})$ along vector $\mathbf{p}$: $\dfrac{\mathbf{p}^T \nabla^2 F(\mathbf{x}) \mathbf{p}}{\|\mathbf{p}\|^2}$

# Minima

## Strong Minimum

The point $\mathbf{x}^*$ is a strong minimum of $F(\mathbf{x})$ if a scalar $\delta > 0$ exists, such that $F(\mathbf{x}^*) < F(\mathbf{x}^* + \Delta\mathbf{x})$ for all $\Delta\mathbf{x}$ such that $\delta > \|\Delta\mathbf{x}\| > 0$.

## Global Minimum

The point $\mathbf{x}^*$ is a unique global minimum of $F(\mathbf{x})$ if $F(\mathbf{x}^*) < F(\mathbf{x}^* + \Delta\mathbf{x})$ for all $\Delta\mathbf{x} \neq 0$.

## Weak Minimum

The point $\mathbf{x}^*$ is a weak minimum of $F(\mathbf{x})$ if it is not a strong minimum, and a scalar $\delta > 0$ exists, such that $F(\mathbf{x}^*) \leq F(\mathbf{x}^* + \Delta\mathbf{x})$ for all $\Delta\mathbf{x}$ such that $\delta > \|\Delta\mathbf{x}\| > 0$.

12

- **First-Order Optimality Condition:** $\nabla F(x = x^*) = 0$ (because if the gradient is anything other than zero then $x^*$ cannot be a minimum. If it's greater than zero, moving in the opposite direction of the gradient, and if it's less than zero moving in the direction of the gradient both produce a secondary $x'$ where $F(x') < F(x^*)$ )
- **This is a necessary but not sufficient condition for $x^*$ to be a local minimum.**
- **Any points that satisfy this condition are called stationary points.**
- **Second-order Optimality Condition:** $\nabla^2 F(x = x^*) > 0$
  Assuming the first-order condition is satisfied, the second order Taylor Expansion of $F(x)$ will then become:

$$F(x^* + \Delta x) = F(x^*) + \frac{1}{2}\Delta x^T \nabla^2 F(x = x^*)\Delta x$$

The second term must be greater than (or at least equal to) zero because if it isn't that would mean that $F(x^*) < F(x^* + \Delta x)$.
Therefore $\Delta x^T \nabla^2 F(x = x^*)\Delta x > 0$. We also know that a matrix $A$ is positive-definite if for any $z \neq 0$ the following holds $z^T Az > 0$. Comparing this the equation above tells us that $\nabla^2 F(x = x^*)$ must be positive-definite.
Positive-definiteness of $\nabla^2 F(x = x^*)$ is only a **sufficient** condition and it is not necessary. However, in order for $x^*$ to be a local minimum $\nabla^2 F(x = x^*)$ must **at least (necessary** condition) be positive-semidefinite ( $\nabla^2 F(x = x^*) \geq 0$).

By testing the eigenvalues of the matrix:

- If the eigenvalues are positive, then the matrix is positive definite.
- If all eigenvalues are nonnegative, then the matrix is positive semidefinite.
- A positive definite Hessian matrix is a 2nd-order, sufficient condition for a strong minimum to exist. It is not a necessary condition.
- A minimum can still be strong if the 2nd-order term of the Taylor series is zero, but the 3rd-order term is positive.
- Therefore the 2nd-order, necessary condition for a strong minimum is that the Hessian matrix be positive semidefinite.

## Summary

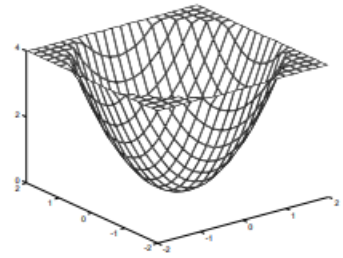
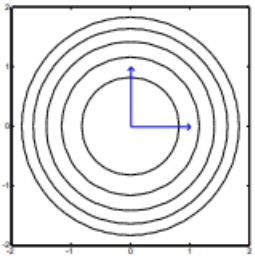
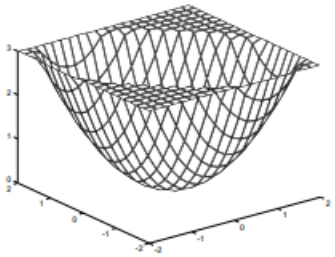
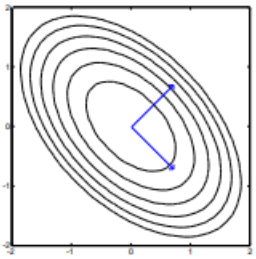
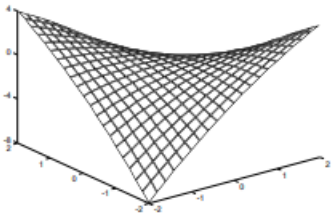
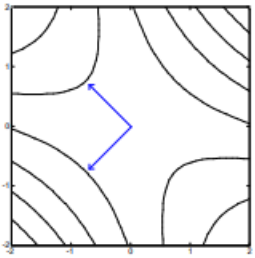
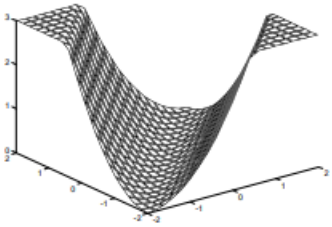
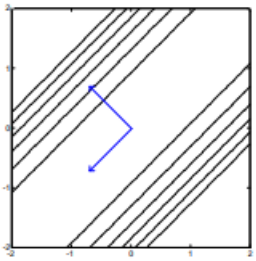
- The first-order and second-order necessary condition for **x*** to be a minimum, strong or weak, of F(**x**) are:

$$\nabla F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}^*} = \mathbf{0} \text{ and } \nabla^2 F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}^*} \quad \text{positive semidefinite.}$$

- The 1st and 2nd order sufficient conditions for **x*** to be a strong minimum point of F(**X**) are

$$\nabla F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}^*} = \mathbf{0} \text{ and } \nabla^2 F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}^*} \quad \text{positive definite.}$$

- **Quadratic Functions**: $F(x) = \frac{1}{2}x^T Ax + d^T x + c$ $(can\ always\ assume\ A\ is\ symmetric)$
- $\nabla(h^T x) = \nabla(x^T h) = h$ $(for\ constant\ vector\ h)$
- $\nabla x^T Qx = Qx + Q^T x = 2Qx$ $(for\ symmetric\ Q)$
- Therefore: $\nabla F(x) = Ax + d$ and $\nabla^2 F(x) = A$.
- **In quadratic functions derivatives of order 3 and beyond all are zero therefore a second order Taylor Expansion of a quadratic function gives an exact representation of it.**
- **All analytic functions behave like quadratics over a small neighborhood.**
- It can be shown that the second derivative of a quadratic function in the direction of an arbitrary vector is always between $\lambda_{min}$ and $\lambda_{max}$, the smallest and the largest eigenvalues of that function's Hessian matrix ($A$).

- This means that since eigenvalues represent curvature along the eigenvectors, the surface area of the function rapidly changes in the direction of the **biggest eigenvector** and has the least amount of changes in the direction of the **smallest eigenvector.**
- **Contours**:
-   $\lambda_1 = \lambda_2, \lambda_1 \; and \; \lambda_2 > 0$
-   $\lambda_1 \neq \lambda_2, \lambda_1 \; and \; \lambda_2 > 0$
-   $\lambda_1 \neq \lambda_2, \lambda_1 . \lambda_2 < 0$
-   $\lambda_1 > 0, \lambda_2 = 0$

## Quadratic Function Summary

- If the eigenvalues of the Hessian matrix are all positive, the function will have a single strong minimum.
- If the eigenvalues are all negative, the function will have a single strong maximum.
- If some eigenvalues are positive and other eigenvalues are negative, the function will have a single saddle point.
- If the eigenvalues are all nonnegative, but some eigenvalues are zero, then the function will either have a weak minimum or will have no stationary point.
- If the eigenvalues are all nonpositive, but some eigenvalues are zero, then the function will either have a weak maximum or will have no stationary point.

- Note: For simplicity we assumed that stationary point was at the origin and had a value zero.
- If $c \neq 0 \rightarrow$ function is increased by $c$ at every point.
- If $\mathbf{d} \neq \mathbf{0}$ and $\mathbf{A}$ is invertible $\rightarrow$ Shape of contours do not change but the stationary point moves to:

$$\mathbf{X}^* = -\mathbf{A}^{-1}\mathbf{d}$$

- If $\mathbf{d} \neq \mathbf{0}$ and $\mathbf{A}$ is not invertible $\rightarrow$ there could be no stationary points.

- **Iterative Optimization:**

In order to optimize a performance index $F(x)$ (usually we mean to minimize this w.r.t. $x$), we start from some initial guess $x_0$and then update our guess in iterations as:

$$x_{k+1} = x_k + \alpha_k p_k, p_k \rightarrow search\ direction, \alpha_k \rightarrow learning\ rate$$

There are different approaches for choosing $\alpha_k$ but it should be noted that we can derive a stability condition for the maximum possible $\alpha_k$ that does not collapse the training process. It can be proven that if $F(x) = \frac{1}{2}x^T Ax + d^T x + c$ then $\alpha < \frac{2}{\lambda_{max}}$ is the stability condition for the learning rate where $\lambda_{max}$ is the greatest eigenvalue of $A$ (Hessian), from $F(x)$.

3 approaches are discussed from obtaining $p_k$ as discussed below.

- **Steepest Descent (Gradient Descent):** (Based on the first order approximation of $F(x)$) Starting from any initial guess $x_0$ we simply set the direction $p_k$ to be the opposite of the gradient of the $F(x)$ which we call $\nabla F(x = x_k)$ or $g_k$. *(Why? Imagine $x$ (the input) is a*

*multivariate vector of dimension S. I.e., $x = [x_1, x_2, \dots, x_S]^T$. It can be proven that minimizing $F(x)$ with respect to each of the $x_i$s is equivalent to minimizing $F(x)$ with respect to $x$ as a whole. Therefore, let's think about the image of $F(x)$ on each of the S planes that constitute the space in which the S-dimensional input vector $x$ resides. In each of these planes, the image of $F(x)$, represented by $\Gamma_i(F(x)), i = 1,2, \dots, S$, is a univariate function. If $\frac{\partial \Gamma_i(F(k))}{\partial x_i} > 0$, this means that increasing values of $x_i$, result in increasing values of $\Gamma_i(F(k))$ which is not desirable since we would like to arrive a minimum. Therefore, in this case it's best to move in the opposite direction of the gradient and **decrease** the values of $x_i$. If $\frac{\partial \Gamma_i(F(k))}{\partial x_i} < 0$, this means that increasing values of $x_i$, result in decreasing values of $\Gamma_i(F(k))$ which is desirable. However, since the gradient is negative, we still move in the opposite direction to **increase** the values of $x_i$.)*

The update rule therefore becomes:

$$x_{k+1} = x_k - \alpha_k g_k$$

There are two ways to choose $\alpha_k$:

1. Setting $\alpha_k$ to a fixed or variable value (like $\frac{1}{k}$).
2. Minimizing $F(x)$ w.r.t. $\alpha_k$along the line $x_k - \alpha_k g_k$.

## Minimizing Along a Line

Choose $\alpha_k$ to minimize $F(\mathbf{x}_k + \alpha_k \mathbf{p}_k)$

$$\frac{d}{d\alpha_k}(F(\mathbf{x}_k + \alpha_k \mathbf{p}_k)) = \nabla F(\mathbf{x})^T\Big|_{\mathbf{x} = \mathbf{x}_k} \mathbf{p}_k + \alpha_k \mathbf{p}_k^T \nabla^2 F(\mathbf{x})\Big|_{\mathbf{x} = \mathbf{x}_k} \mathbf{p}_k$$

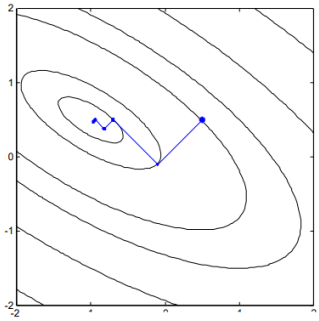
Set equal zero →

$$\alpha_k = -\frac{\nabla F(\mathbf{x})^T\Big|_{\mathbf{x} = \mathbf{x}_k} \mathbf{p}_k}{\mathbf{p}_k^T \nabla^2 F(\mathbf{x})\Big|_{\mathbf{x} = \mathbf{x}_k} \mathbf{p}_k} = -\frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{p}_k^T \mathbf{A}_k \mathbf{p}_k}$$

where

$$\mathbf{A}_k \equiv \nabla^2 F(\mathbf{x})\Big|_{\mathbf{x} = \mathbf{x}_k}$$

## Plot




Note that the successive steps of the algorithm are orthogonal. Why does this happen?

$$\frac{d}{d\alpha_k} F(\mathbf{x}_k + \alpha_k \mathbf{p}_k) = \frac{d}{d\alpha_k} F(\mathbf{x}_{k+1}) = \nabla F(\mathbf{x})^T\Big|_{\mathbf{x} = \mathbf{x}_{k+1}} \frac{d}{d\alpha_k}[\mathbf{x}_k + \alpha_k \mathbf{p}_k]$$

$$= \nabla F(\mathbf{x})^T\Big|_{\mathbf{x} = \mathbf{x}_{k+1}} \mathbf{p}_k = \mathbf{g}_{k+1}^T \mathbf{p}_k$$

- **Newton's Method:** (Based on the second order approximation of $F(x)$)

$$F(x_{k+1}) = F(x_k + \Delta x_k) \approx F(x_k) + g_k^T \Delta x_k + \frac{1}{2}\Delta x_k^T A_k \Delta x_k$$

We approach the stationary point of this second-order approximation by setting its gradient to zero. (*Note that because of this, $x_{k+1}$ is **NOT** necessarily a minimum. It can also be a stationary point. Therefore, the success of this method largely depends on the initial guess*)

$$\frac{\partial F(x = x_{k+1})}{\partial \Delta x_k} = g_k + A_k \Delta x_k = 0 \rightarrow \Delta x_k = -A_k^{-1} g_k \rightarrow x_{k+1} = x_k - A_k^{-1} g_k$$

This method always arrives at the stationary point of the quadratic function **one step.**
This is because Newton's Method approximates a function as a quadratic using its second-order Taylor Expansion and then locates the stationary point of the that quadratic function. Meaning that if the function is already quadratic Newton's Method will arrive at its stationary point in one step but if it's not, it will not converge in one step.

- Newton's method usually produces faster convergence than steepest descent.
- Newton's method can be quite complex.
- Convergence to saddle point is very unlikely with SD.
- In Newton's it is possible for the algorithm to oscillate or diverge, but SD is guaranteed to convergence if α is not too large or if we perform a linear minimization at each stage.
- In Ch 12 a variation of Newton's method that is well suited to *NN* training is discussed. It eliminates the divergence problem by using SD steps whenever divergence begins to occur.
- Newton's method requires computation and storage of the Hessian Matrix as well as its inverse.

21

Newton's method has quadratic termination. **It means that it an minimize a quadratic function precisely in a finite number of iterations.**
However, when the number of parameters is large, it is difficult to compute and store the Hessian matrix and its inverse.

- **Conjugate Gradient:**

Is a method that requires only the 1st derivative but still has the quadratic termination property like **Newton's Method**.
A set of vectors is mutually conjugate w.r.t. a positive-definite Hessian matrix A if

$$p_k^T A p_j = 0 \quad k \neq j$$

There are an infinite number of mutually conjugate vectors that span a given n-dimensional space. If $A$ is the identity matrix, conjugacy is equivalent to the notion of orthogonality.
One set of conjugate vectors are the eigenvectors of $A$: $z_k^T A z_j = \lambda_j z_k^T z_j = 0, k \neq j$.
Also **the eigenvectors of symmetric matrices are orthogonal.**

We can minimize a quadratic function by searching along the eigenvectors of the Hessian matrix, because they form the principal axes of the function contours. But this requires the computation of Hessian matrix.

It can be shown that if we search along **any set of conjugate directions** $\{p_1, p_2, \dots, p_n\}$ then **the exact minimum of any quadratic function with $n$ parameters will be reached in at most $n$ searches.** (So we don't have to use the eigenvectors of $A$, any set of size $n$ of conjugate directions will do).

Search directions will be conjugate if they are orthogonal to the changes in the gradients.

Steps of the Conjugate Gradient algorithm:

1. Choose the initial search direction as the negative of the gradient: $p_0 = -g_0$
2. Take a step according to: $\Delta x_k = \alpha_k p_k$, where $\alpha_k$ minimizes along the line. For quadratic functions we set: $\alpha_k = -\frac{g_k^T p_k}{p_k^T A_k p_k}$
3. Select the next search direction using: $p_k = -g_k + \beta_k p_{k-1}$
   Where $\beta_k$ is obtained from:

$$\beta_k = \frac{\Delta \mathbf{g}_{k-1}^T \mathbf{g}_k}{\Delta \mathbf{g}_{k-1}^T \mathbf{p}_{k-1}} \quad (1)$$

or

$$\beta_k = \frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}} \quad (2)$$

or

$$\beta_k = \frac{\Delta \mathbf{g}_{k-1}^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}} \quad (3)$$

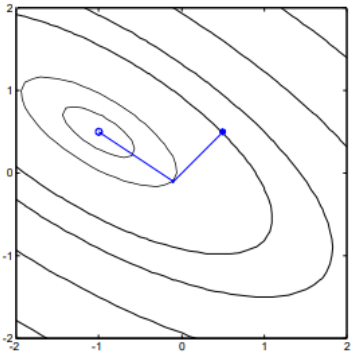
   (**Note: (2) is the most convenient since it doesn't rely on $\Delta g_k$**)
4. If not converged, return to step 2.

A quadratic $n$-variate function is guaranteed to be minimized in at most $n$ steps.

**Note: The first step of CG is equivalent to SD with minimization along the line.**

# Plots

Conjugate Gradient



Steepest Descent