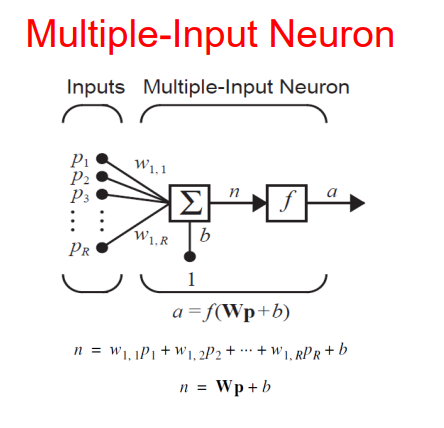
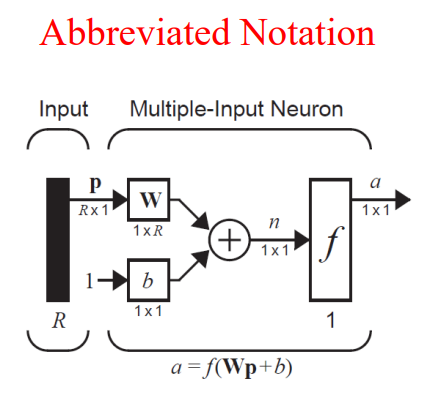
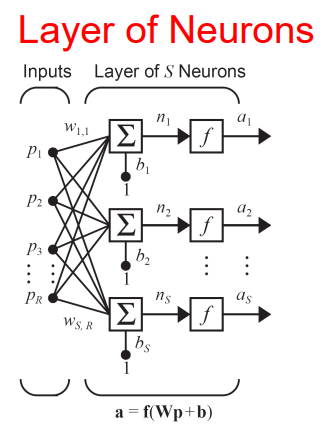
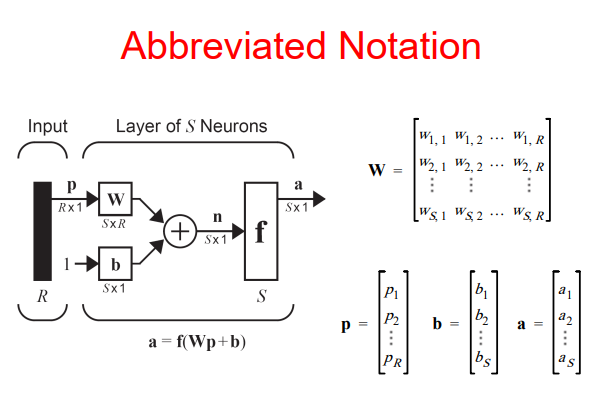
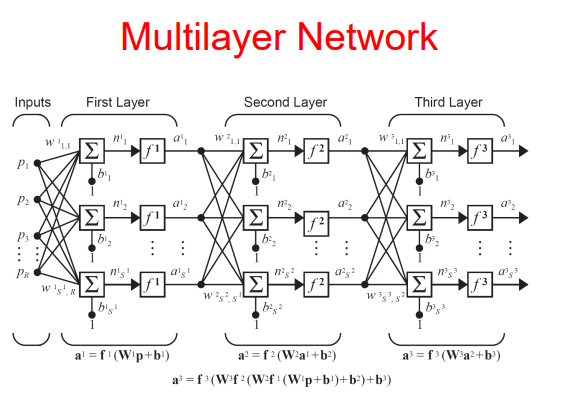
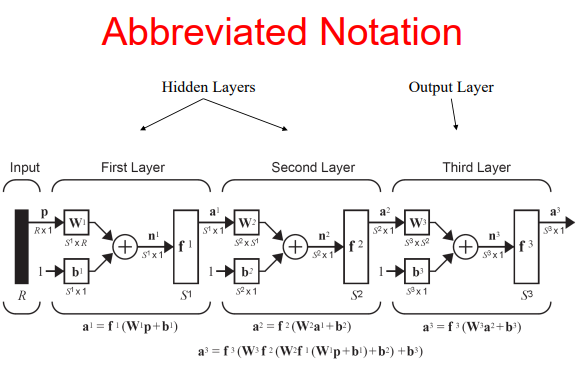
* Brain vs. Computer:

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.

* 
* 
* 
* 
* 
* 
* Network Architecture Design:

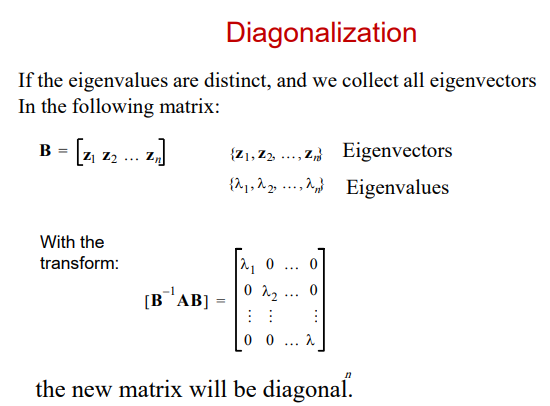
1. Number of network inputs = number of problem inputs
2. Number of neurons in output layer = number of problem outputs
3. Output layer transfer function choice at least partly determined by problem specification of the outputs (for example a probability output would probably use sigmoid for 2 class and softmax for multiclass)

* Linear Algebra Review:
* Dot product of two column vectors:
* Two vectors are orthogonal iff
* P-Norm of a vector: . Specifically, the 2-Norm or Standard Euclidean Norm:
* Eigenvalues and Eigenvectors:
  + An eigenvector and its corresponding eigenvalue, , of a matrix satisfies:
  + Computing the Eigenvalues:

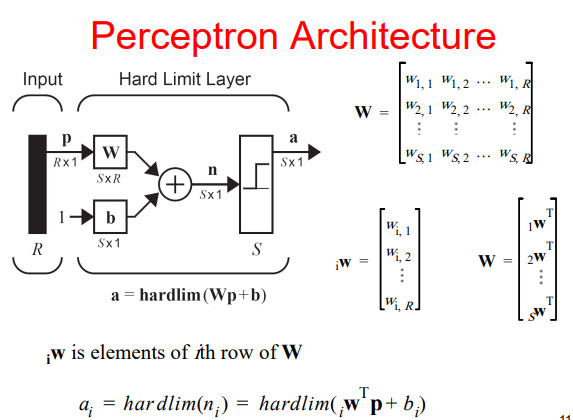
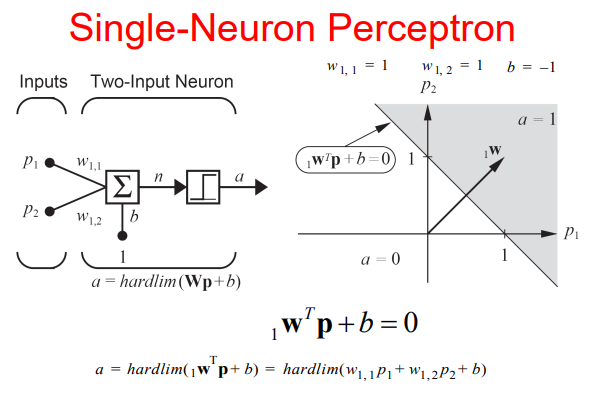
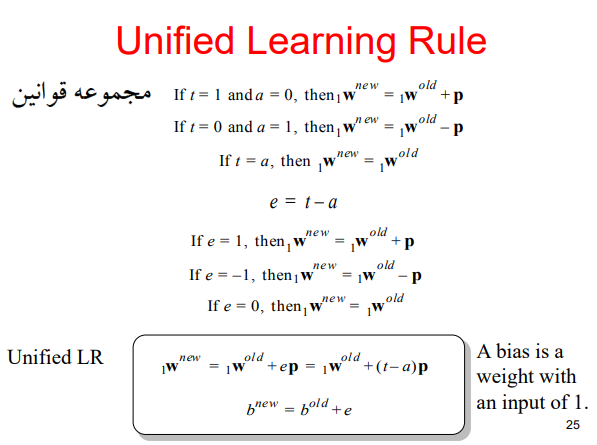
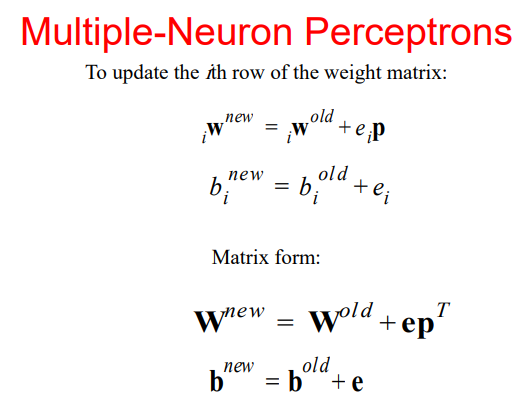
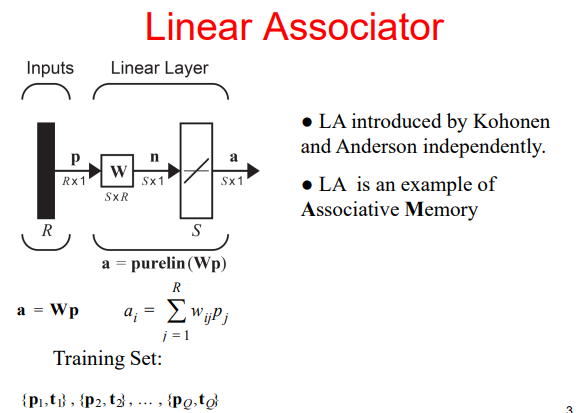
Solutions to this give all eigenvalues .

* + Computing the Eigenvectors:

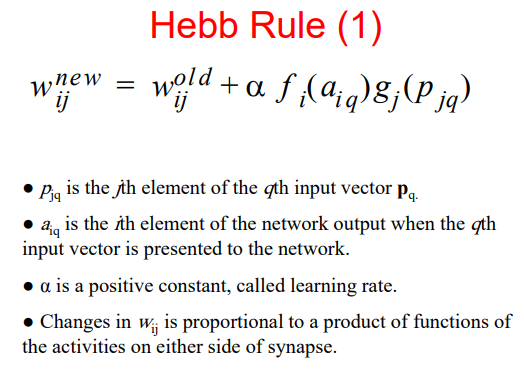
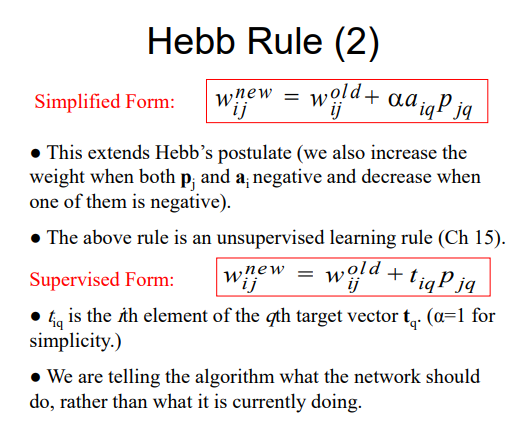
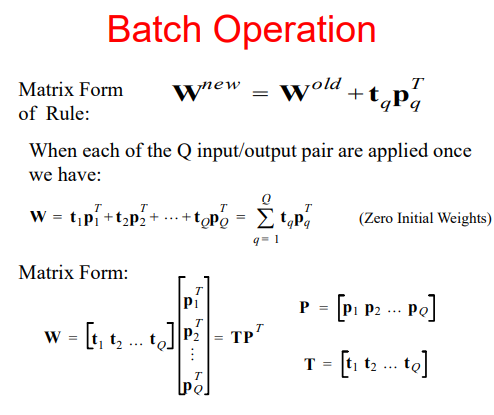
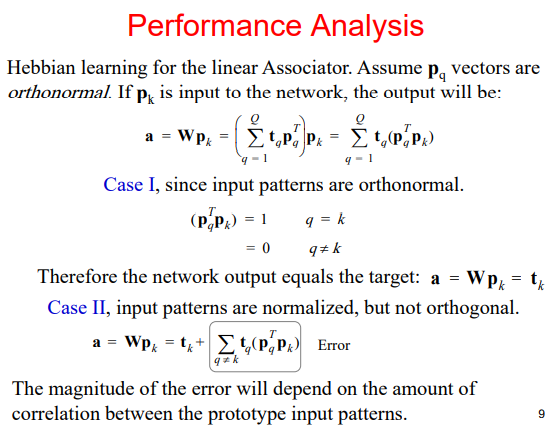
Plug the s from the previous step into and solve for .

* + Diagonalization through Eigenvalues and Eigenvectors:
  + 

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

* 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).
* ****
* ****
* **Weight vector always orthogonal to the decision boundary (why?)**
  + Consider two points on the decision boundary and . The vector is parallel to the decision boundary.
  + However, we know that for both and we have:
  + .
  + And is parallel to the decision boundary line therefore is also normal to the decision boundary.
  + Another way to think about this: without loss of generality subtract from every -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 which directly proves that is normal to the decision boundary. Because is normal in this transformed space, it is also normal in every other space through a linear transform (for example adding to -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 categories where is the number of neurons.
* ****
* ****
* 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 .
* 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).
* ****
* Pattern Association: an associative memory learns by association:

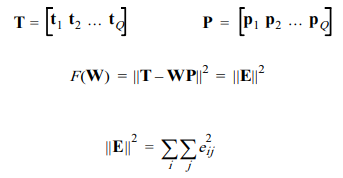
In simple words, an associative memory can associate a specific input with a specific output. If that input is slightly changed, we expect the target to still be the same as before.

* **Auto-Association:** If , i.e., the network is expected to “memorize” a set of patterns to be able to later reconstruct them.
* **Hetero-Association:** If , i.e., each input is paired with an arbitrary target that is not the input itself.
* 
* 
* 
* 
* In order to reduce this error, we can employ the Pseudoinverse Rule:
  + Task of the LA is to produce target given input . Or:

If 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:

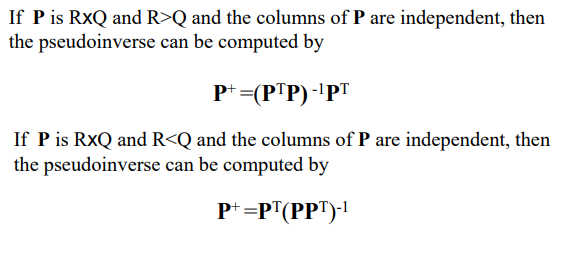
If the input vectors are not orthonormal, will not be zero and Hebb’s Rule does not guarantee that the resulting weights minimize . In order to obtain that minimizes we use the Pseudoinverse Rule.

Vectorized notation:



Note that if has a solution for , can be made zero. In other words, can be made zero if is invertible so that:

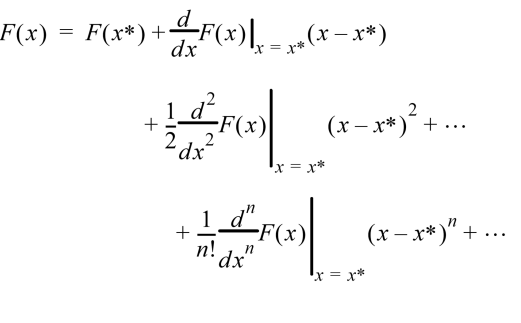
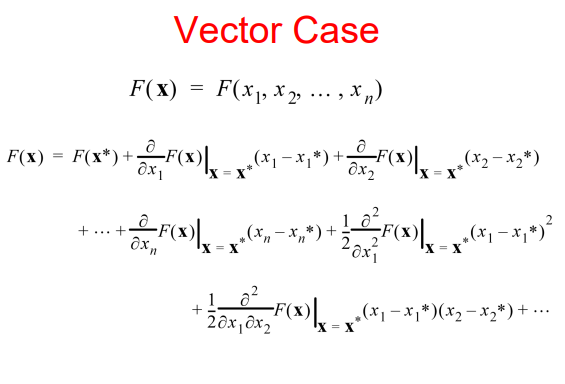
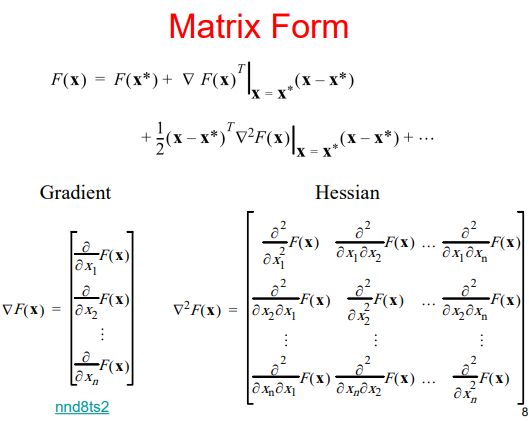
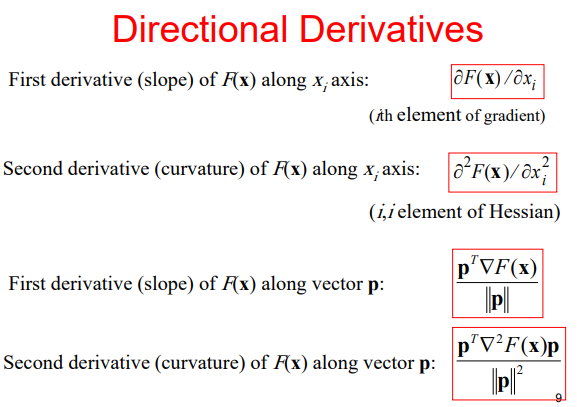
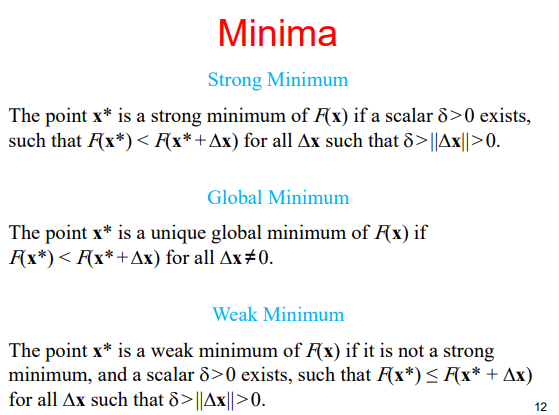
But this doesn’t happen often because is very rarely a square matrix. However, when an inverse does not exist it is proven that the pseudoinverse of minimized or:

* **IMPORTANT**
* 
* If the input patterns are orthonormal and will both be (depending on if the #cols > #rows or not). And hence . Therefore, Pseudoinverse Rule and Hebb’s Rule are equivalent for orthonormal inputs.
* Variations of Hebbian Learning:

As approaches one, this rule quickly forgets old inputs and remembers only the most recent patterns. **This keeps the weight matrix from growing unboundedly.**

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.

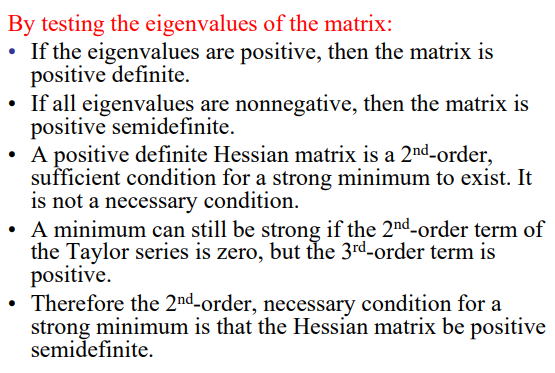
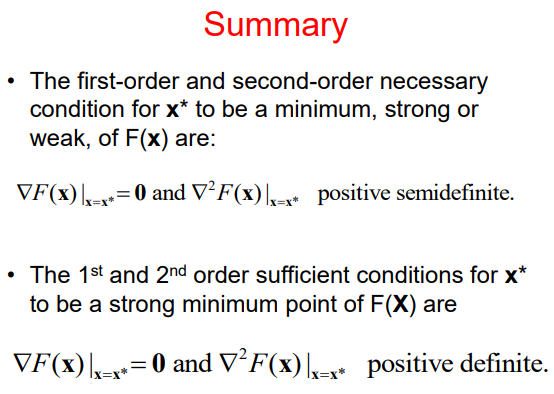
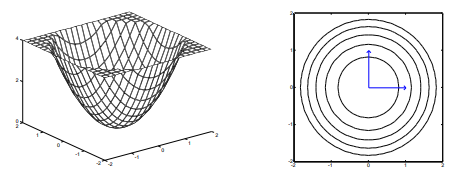
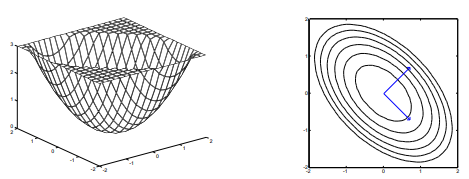
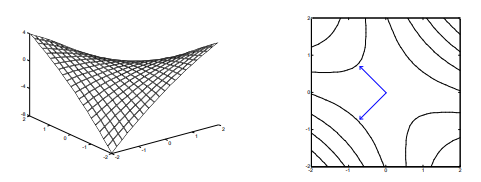
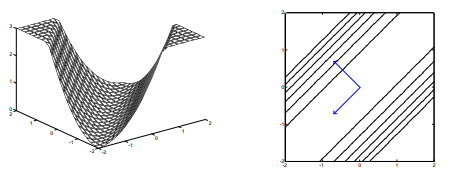
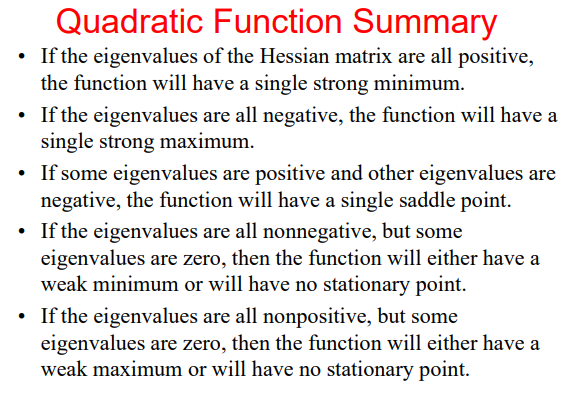
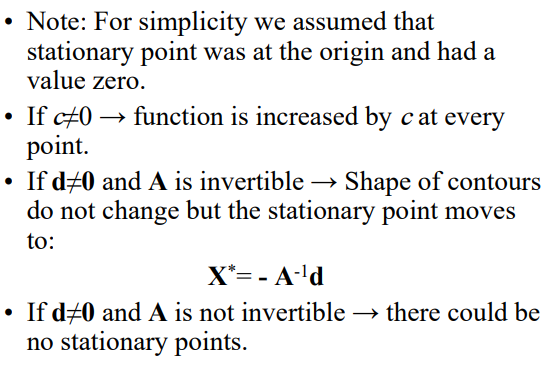
* Any arbitrary function can be approximated around a certain point with a finite number of its **Taylor Expansion** terms.
* 
* 
* 
* 
* 
* **First-Order Optimality Condition:** (because if the gradient is anything other than zero then cannot be a minimum. If it’s greater than zero, moving in the opposite direction of the gradient, and if it’s less than zero moving in the direction of the gradient both produce a secondary where )
* **This is a necessary but not sufficient condition for to be a local minimum.**
* **Any points that satisfy this condition are called stationary points.**
* **Second-order Optimality Condition:**

Assuming the first-order condition is satisfied, the second order Taylor Expansion of will then become:

The second term must be greater than (or at least equal to) zero because if it isn’t that would mean that .

Therefore . We also know that a matrix is positive-definite if for any the following holds . Comparing this the equation above tells us that must be positive-definite.

Positive-definiteness of is only a **sufficient** condition and it is not necessary. However, in order for to be a local minimum must **at least (necessary** condition) be positive-semidefinite ().

* ****
* ****
* **Quadratic Functions**:
* Therefore: and .
* **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 and , the smallest and the largest eigenvalues of that function’s Hessian matrix ().
* 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**:
* ****
* ****
* ****
* ****
* ****
* ****
* **Iterative Optimization:**

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

There are different approaches for choosing but it should be noted that we can derive a stability condition for the maximum possible that does not collapse the training process.

It can be proven that if then is the stability condition for the learning rate where is the greatest eigenvalue of (Hessian), from .

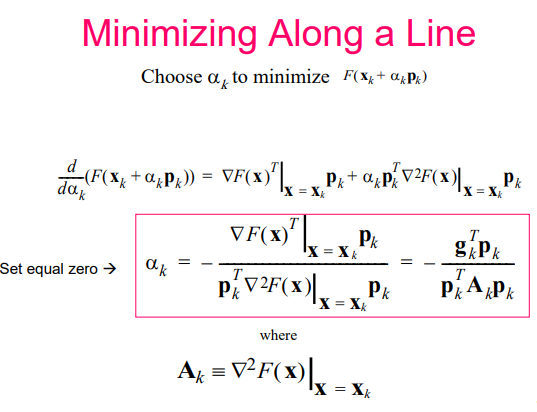
3 approaches are discussed from obtaining as discussed below.

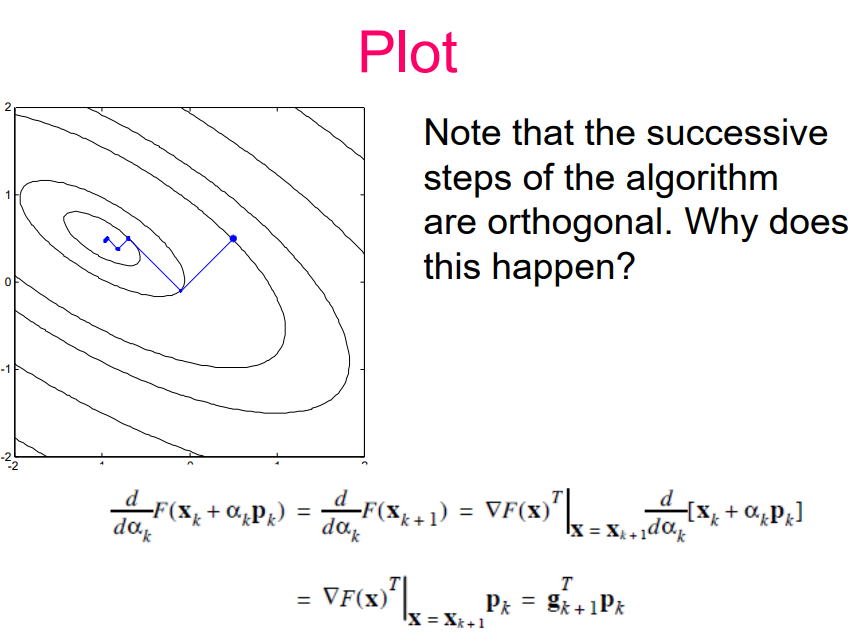
* **Steepest Descent (Gradient Descent):** (Based on the first order approximation of *(*

Starting from any initial guess we simply set the direction to be the opposite of the gradient of the which we call or . *(Why? Imagine (the input) is a multivariate vector of dimension . I.e., . It can be proven that minimizing with respect to each of the s is equivalent to minimizing with respect to as a whole. Therefore, let’s think about the image of on each of the planes that constitute the space in which the -dimensional input vector resides. In each of these planes, the image of , represented by , is a univariate function. If , this means that increasing values of , result in increasing values of 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 . If , this means that increasing values of , result in decreasing values of which is desirable. However, since the gradient is negative, we still move in the opposite direction to* ***increase*** *the values of .)*

The update rule therefore becomes:

There are two ways to choose :

1. Setting to a fixed or variable value (like ).
2. Minimizing w.r.t. along the line .

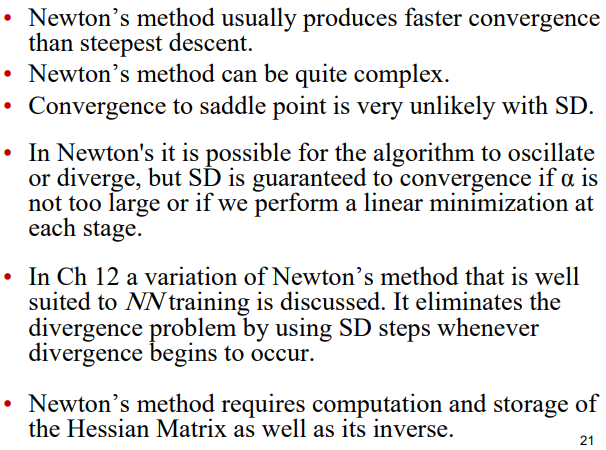


* **Newton’s Method:** (Based on the second order approximation of )

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

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 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

There are an infinite number of mutually conjugate vectors that span a given n-dimensional space. If is the identity matrix, conjugacy is equivalent to the notion of orthogonality.

One set of conjugate vectors are the eigenvectors of : .

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.

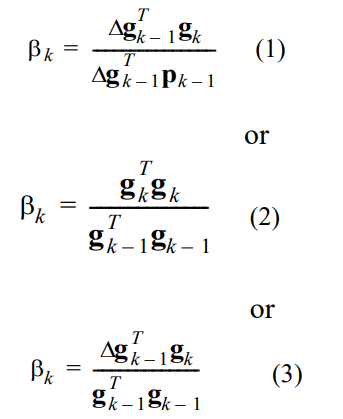
It can be shown that if we search along **any set of conjugate directions**  then **the exact minimum of any quadratic function with**  **parameters will be reached in at most**  **searches.** (So we don’t have to use the eigenvectors of , any set of size of conjugate directions will do).

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:
2. Take a step according to: , where minimizes along the line. For quadratic functions we set:
3. Select the next search direction using:

Where is obtained from:

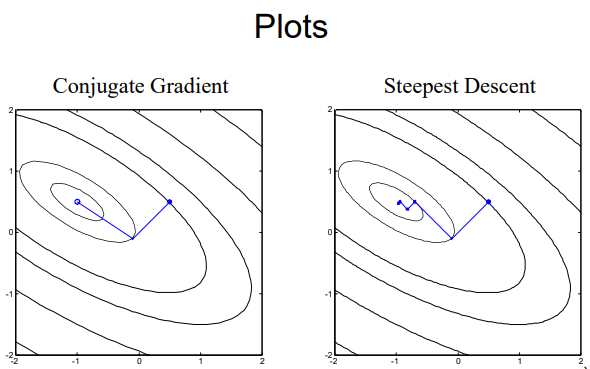


(**Note: (2) is the most convenient since it doesn’t rely on** )

1. If not converged, return to step 2.

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

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

****