

Neural Networks

Learning the network: Part 1

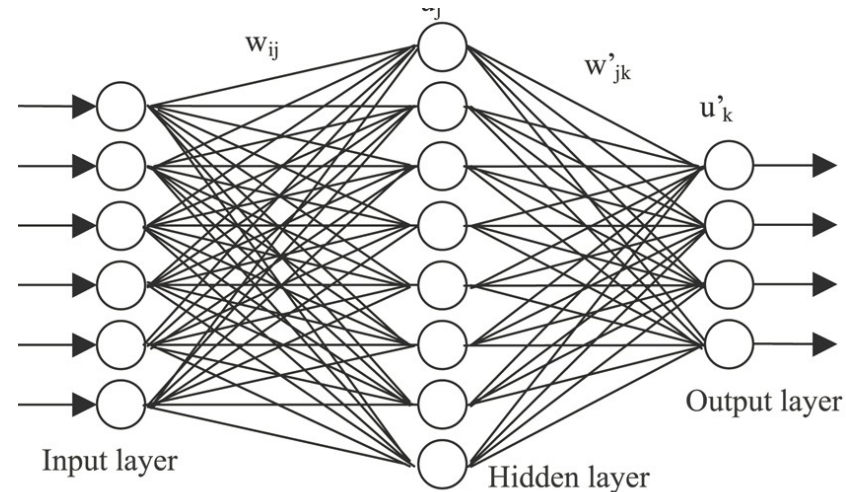
11-785, Spring 2020

Lecture 3

Topics for the day

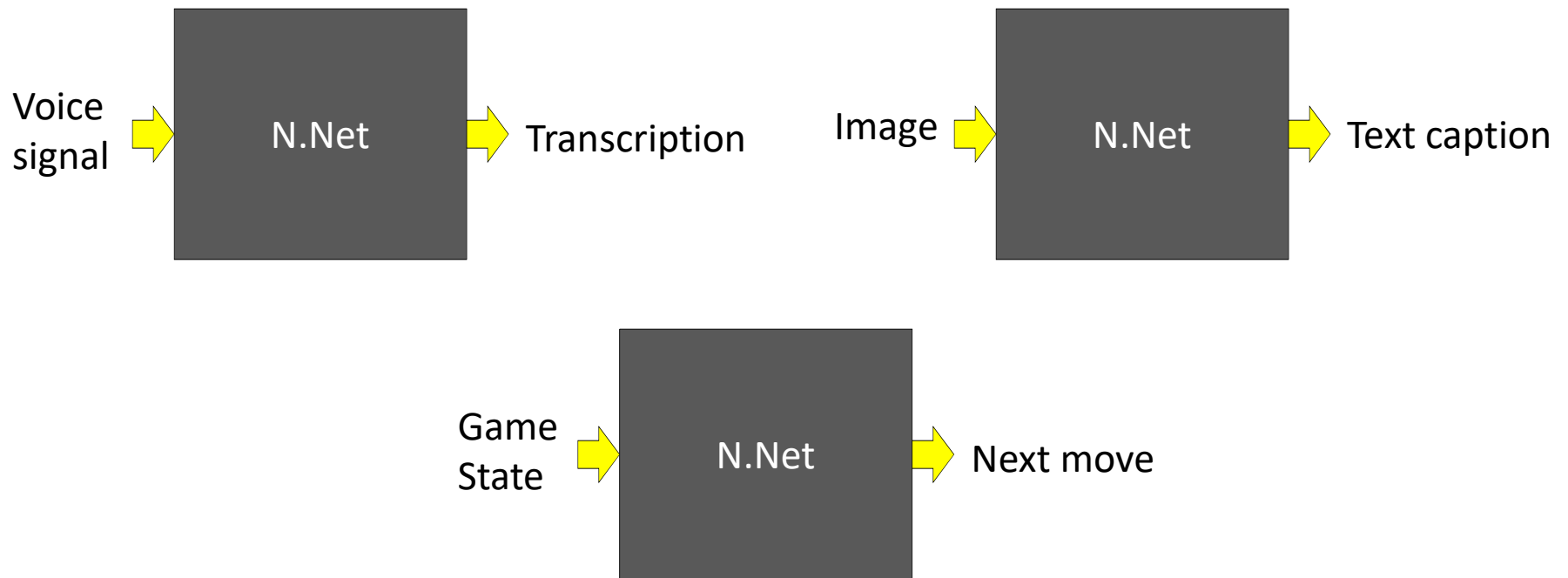
- The problem of learning
- The perceptron rule for perceptrons
 - And its inapplicability to multi-layer perceptrons
- Greedy solutions for classification networks: ADALINE and MADALINE
- Learning through Empirical Risk Minimization
- Intro to function optimization and gradient descent

Recap



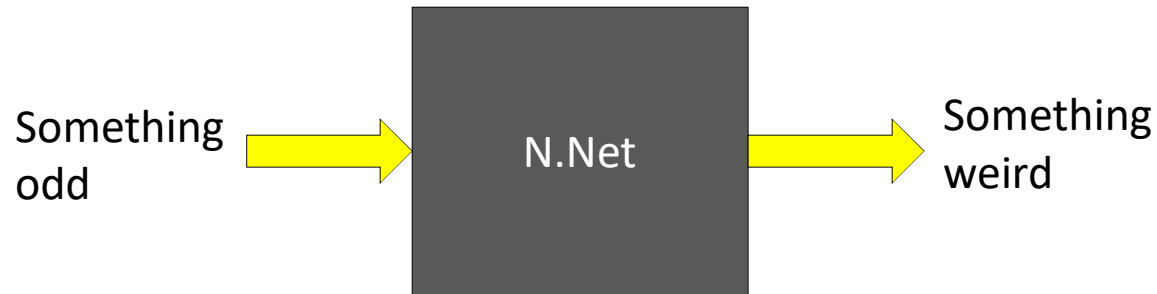
- **Neural networks are universal function approximators**
 - Can model any Boolean function
 - Can model any classification boundary
 - Can model any continuous valued function
- *Provided the network satisfies minimal architecture constraints*
 - Networks with fewer than the required number of parameters can be very poor approximators

These boxes are functions



- Take an input
- Produce an output
- Can be modeled by a neural network!

Questions



- Preliminaries:
 - How do we represent the input?
 - How do we represent the output?
- How do we compose the network that performs the requisite function?

Questions



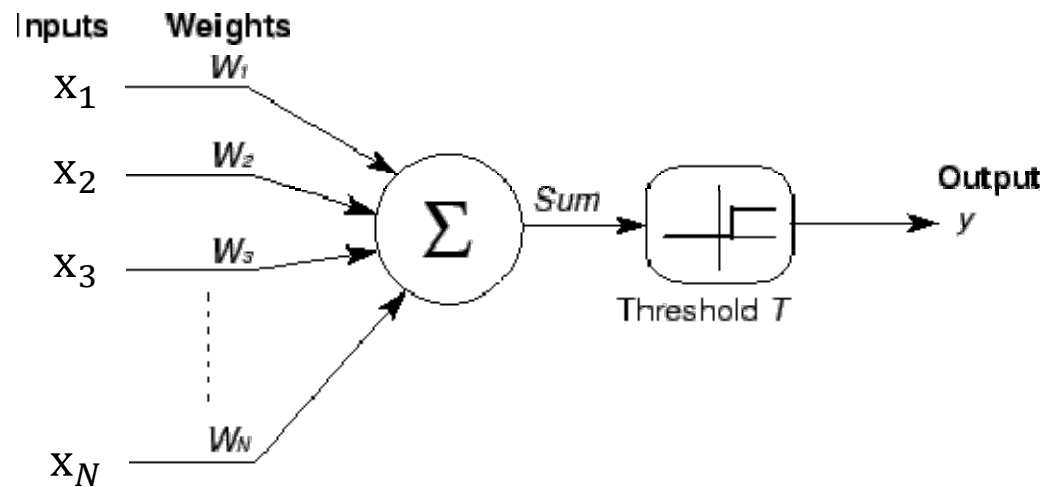
- Preliminaries:

- How do we represent the input?
- How do we represent the output?

A bit later in the program

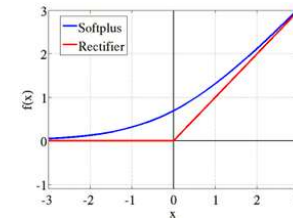
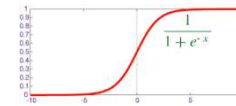
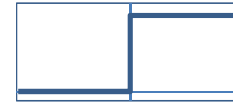
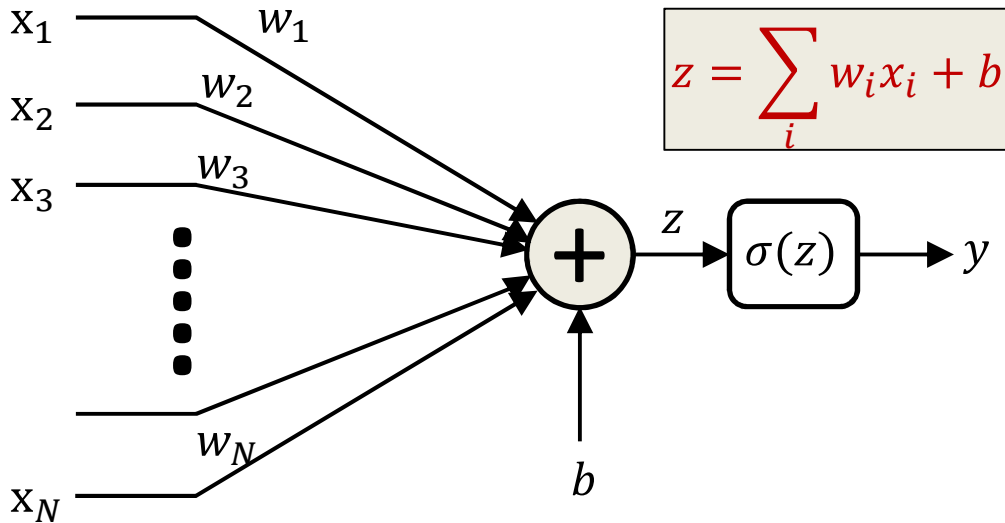
- *How do we compose the network that performs the requisite function?* ←

The original perceptron



- Simple threshold unit
 - Unit comprises a set of weights and a threshold

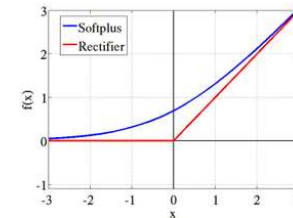
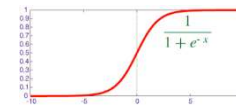
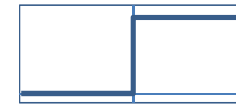
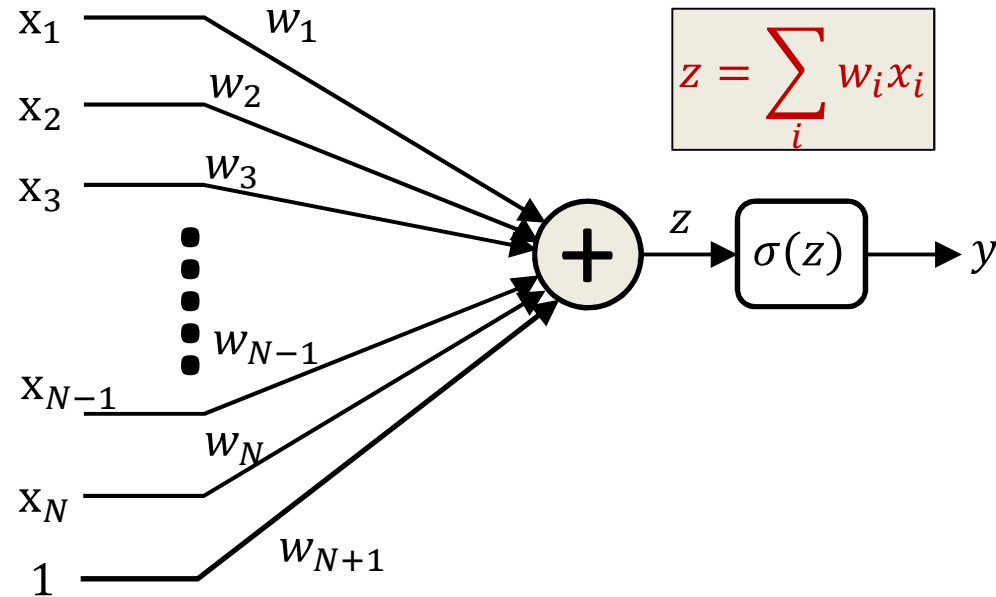
Preliminaries: The units in the network



Activation functions $\sigma(z)$

- Perceptron
 - General setting, inputs are real valued
 - A *bias* b representing a threshold to trigger the perceptron
 - Activation functions are not necessarily threshold functions

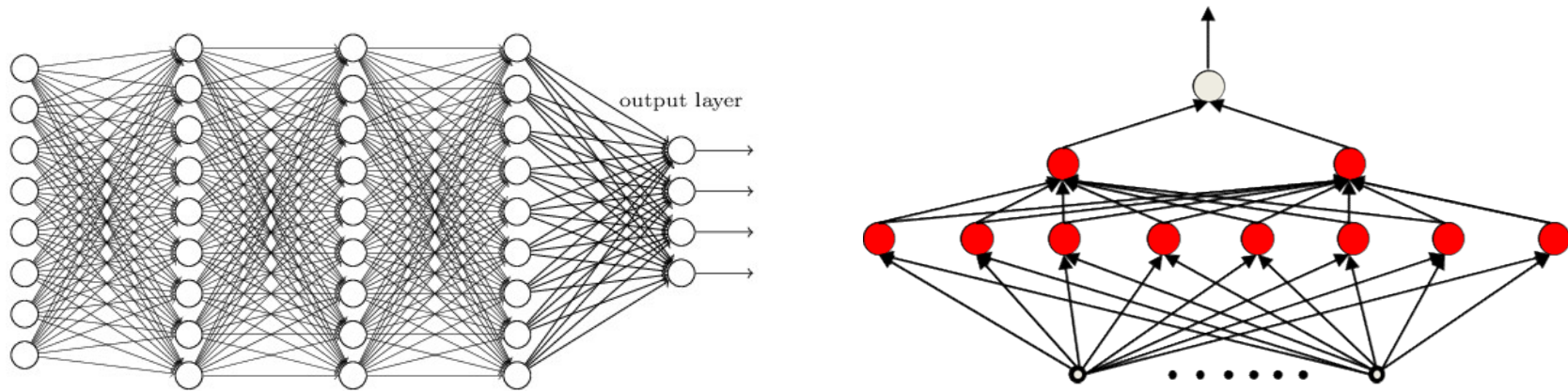
Preliminaries: Redrawing the neuron



Activation functions $\sigma(z)$

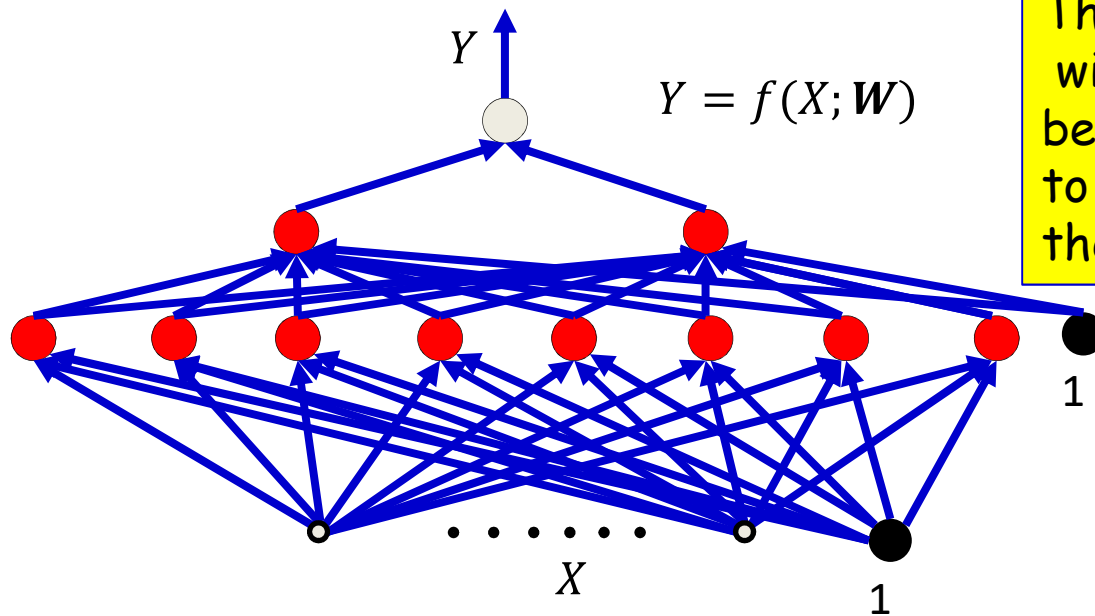
- The bias can also be viewed as the weight of another input component that is always set to 1
 - If the bias is not explicitly mentioned, we will implicitly be assuming that every perceptron has an additional input that is always fixed at 1

First: the structure of the network



- We will assume a *feed-forward* network
 - No loops: Neuron outputs do not feed back to their inputs directly or indirectly
 - Loopy networks are a future topic
- **Part of the design of a network: The architecture**
 - How many layers/neurons, which neuron connects to which and how, etc.
- For now, assume the architecture of the network is capable of representing the needed function

What we learn: The parameters of the network

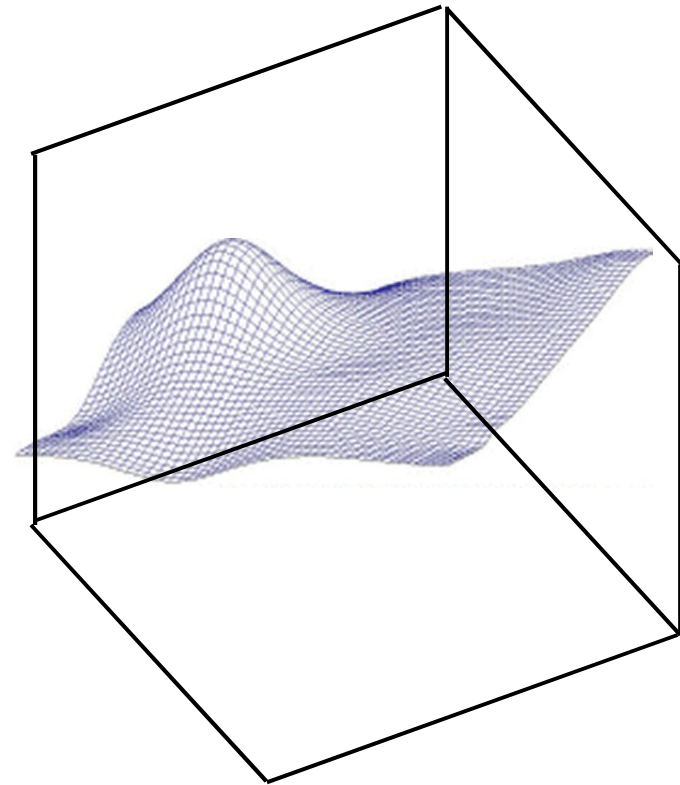
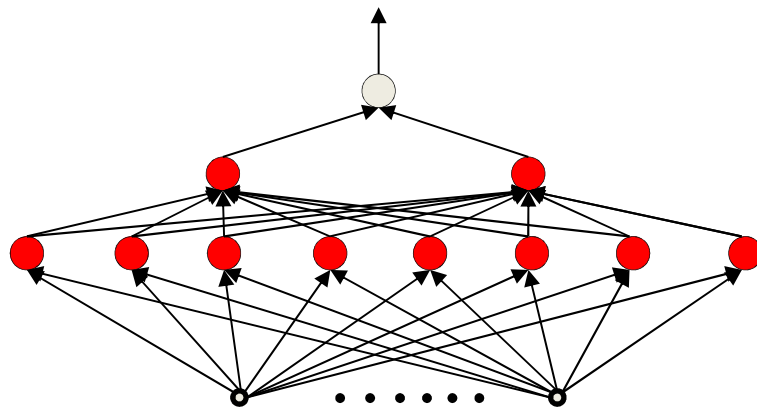


The network is a function $f()$ with parameters W which must be set to the appropriate values to get the desired behavior from the net

- **Given:** the architecture of the network
- **The parameters of the network:** The weights and biases
 - The weights associated with the blue arrows in the picture
- *Learning the network* : Determining the values of these parameters such that the network computes the desired function

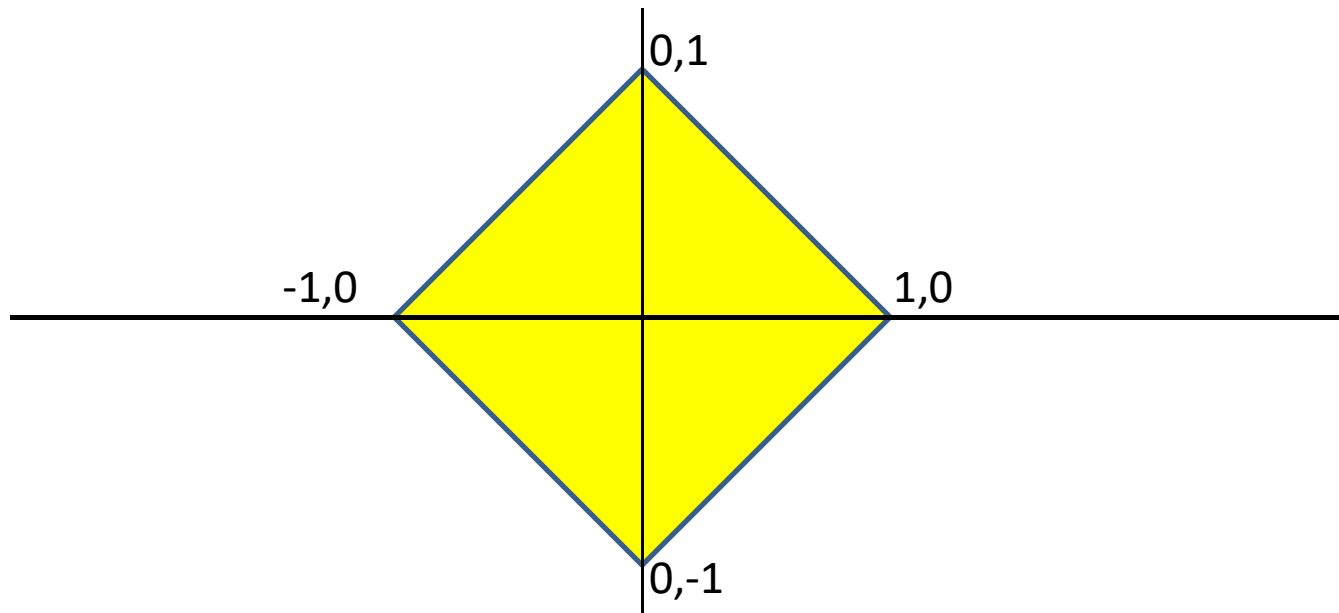
- Moving on..

The MLP *can* represent anything



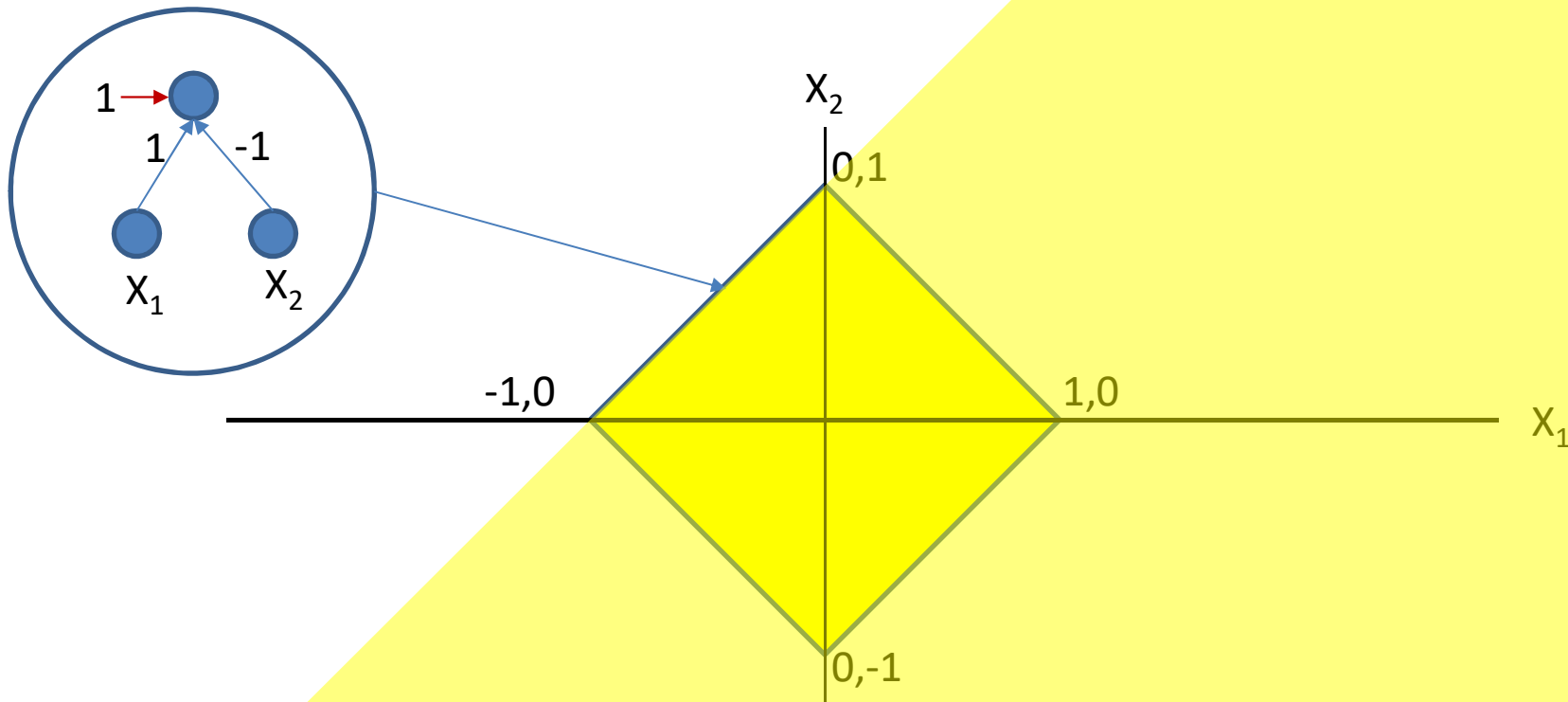
- The MLP *can be constructed* to represent anything
- But *how* do we construct it?

Option 1: Construct by hand



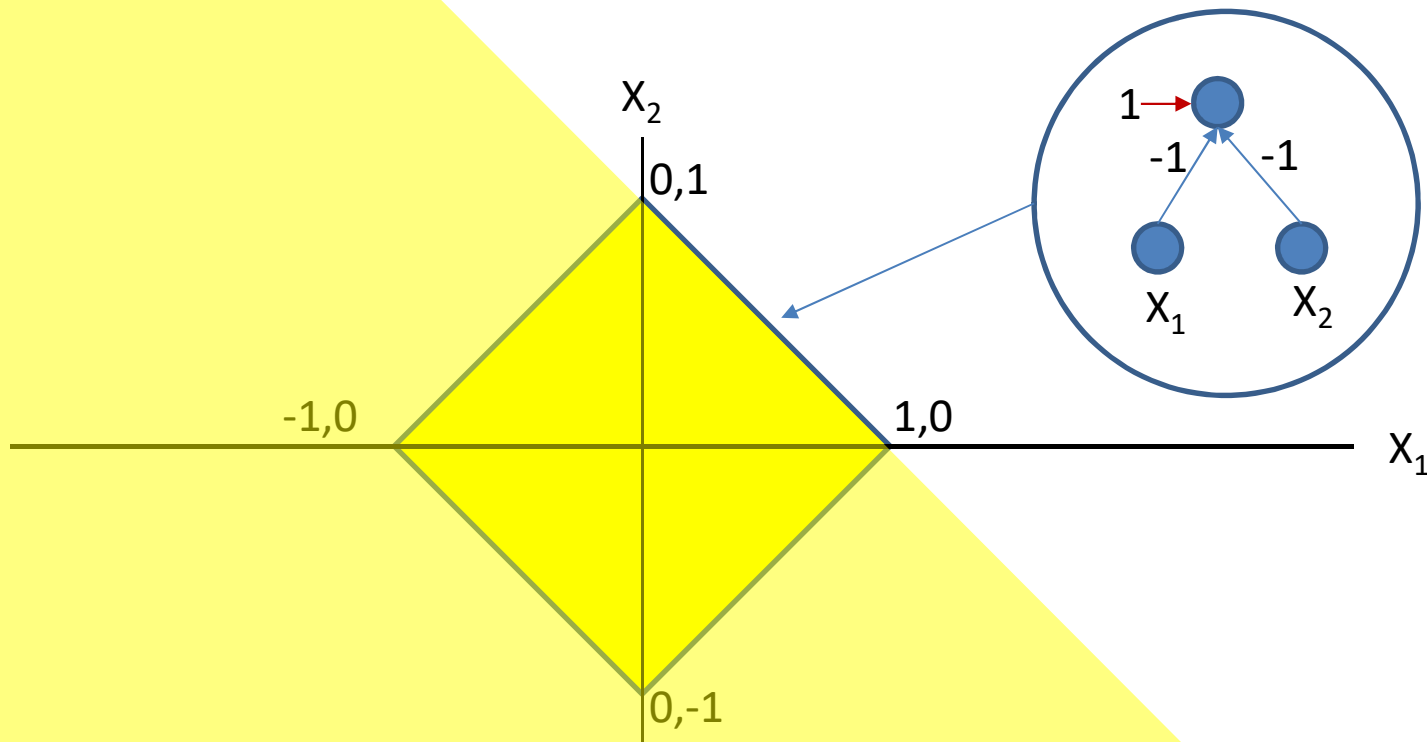
- Given a function, *handcraft* a network to satisfy it
- E.g.: Build an MLP to classify this decision boundary

Option 1: Construct by hand



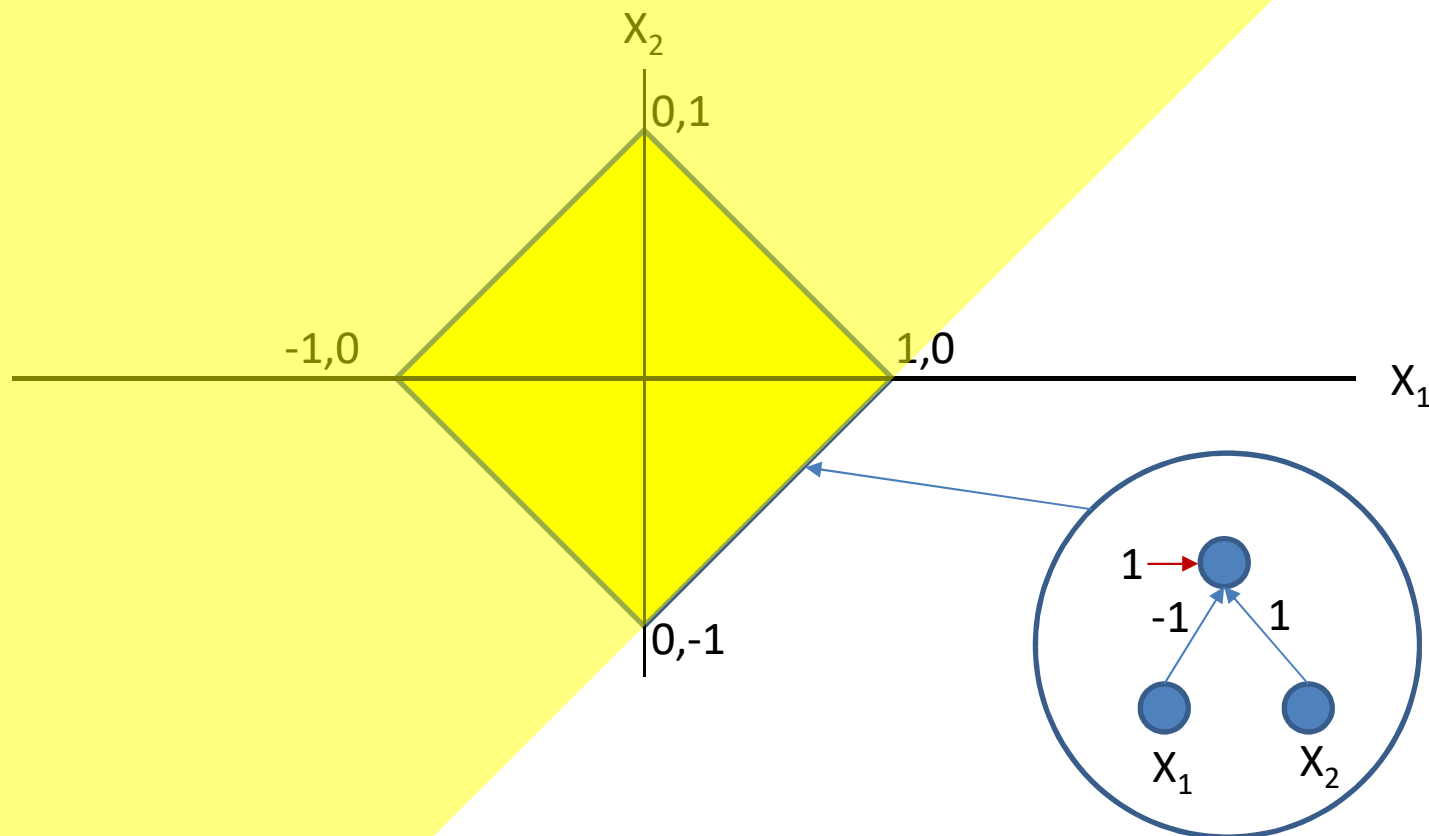
Assuming simple perceptrons:
output = 1 if $\sum_i w_i x_i + b_i \geq 0$, else 0

Option 1: Construct by hand



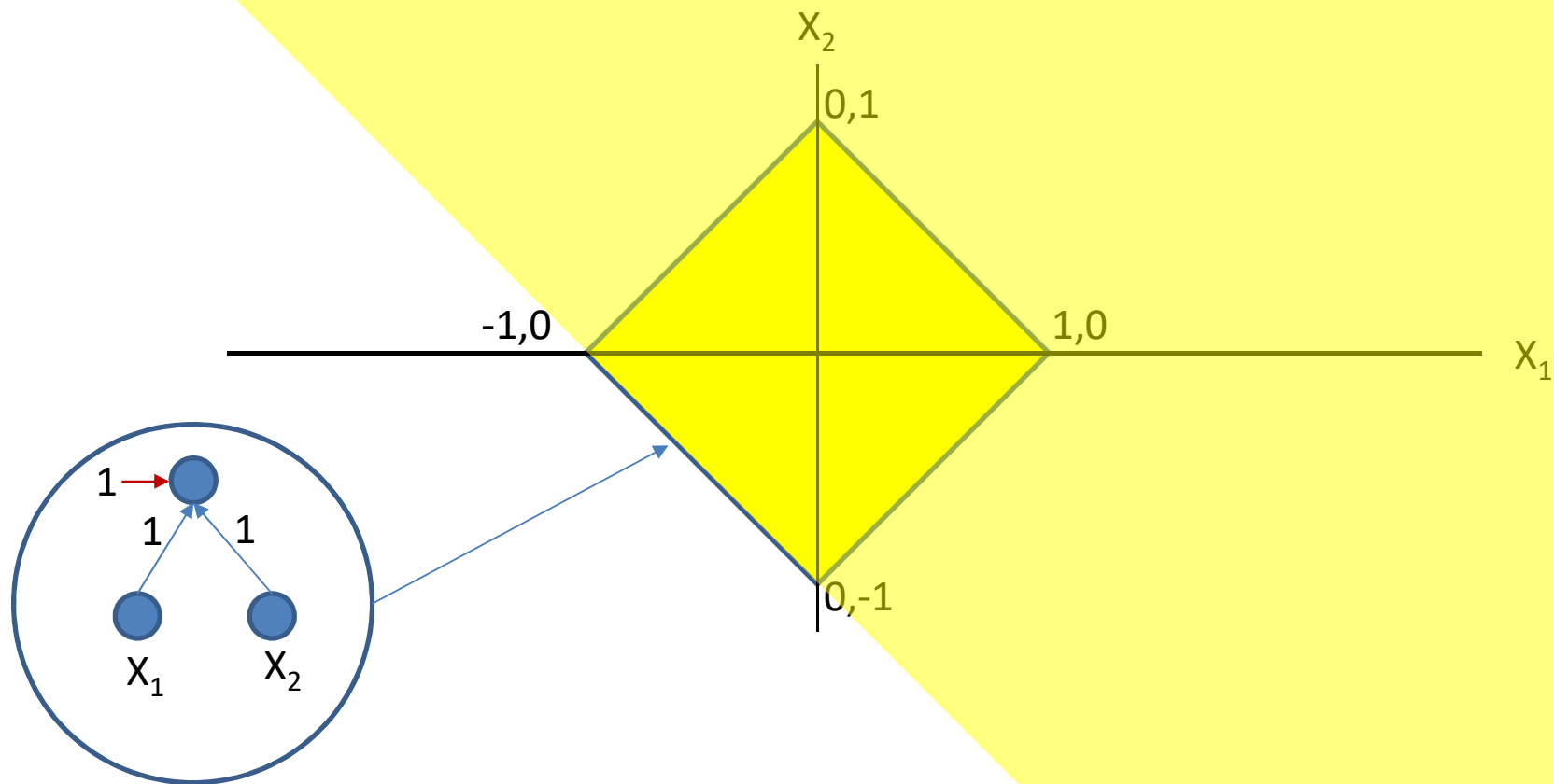
Assuming simple perceptrons:
output = 1 if $\sum_i w_i x_i + b_i \geq 0$, else 0

Option 1: Construct by hand



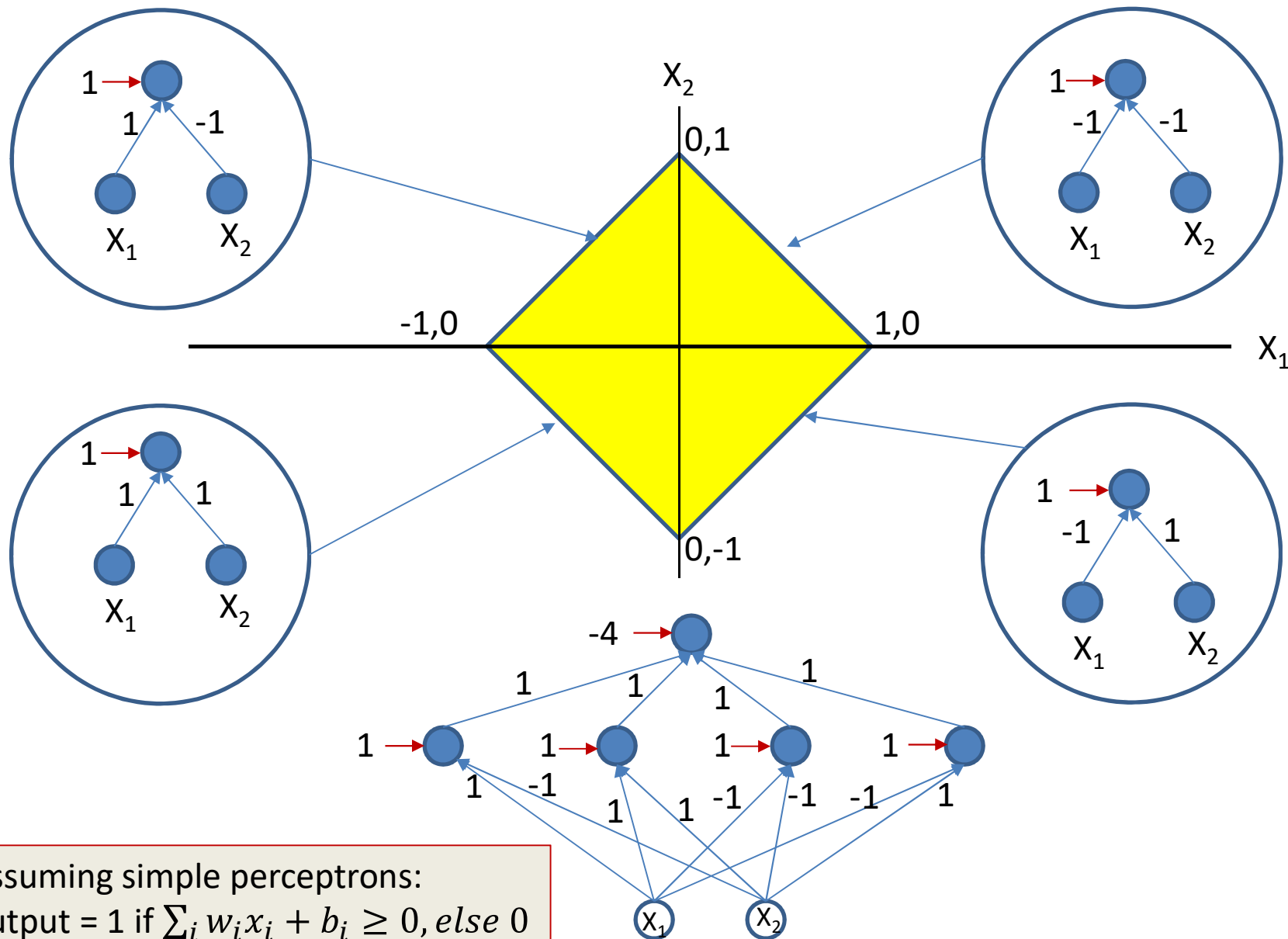
Assuming simple perceptrons:
output = 1 if $\sum_i w_i x_i + b_i \geq 0$, else 0

Option 1: Construct by hand



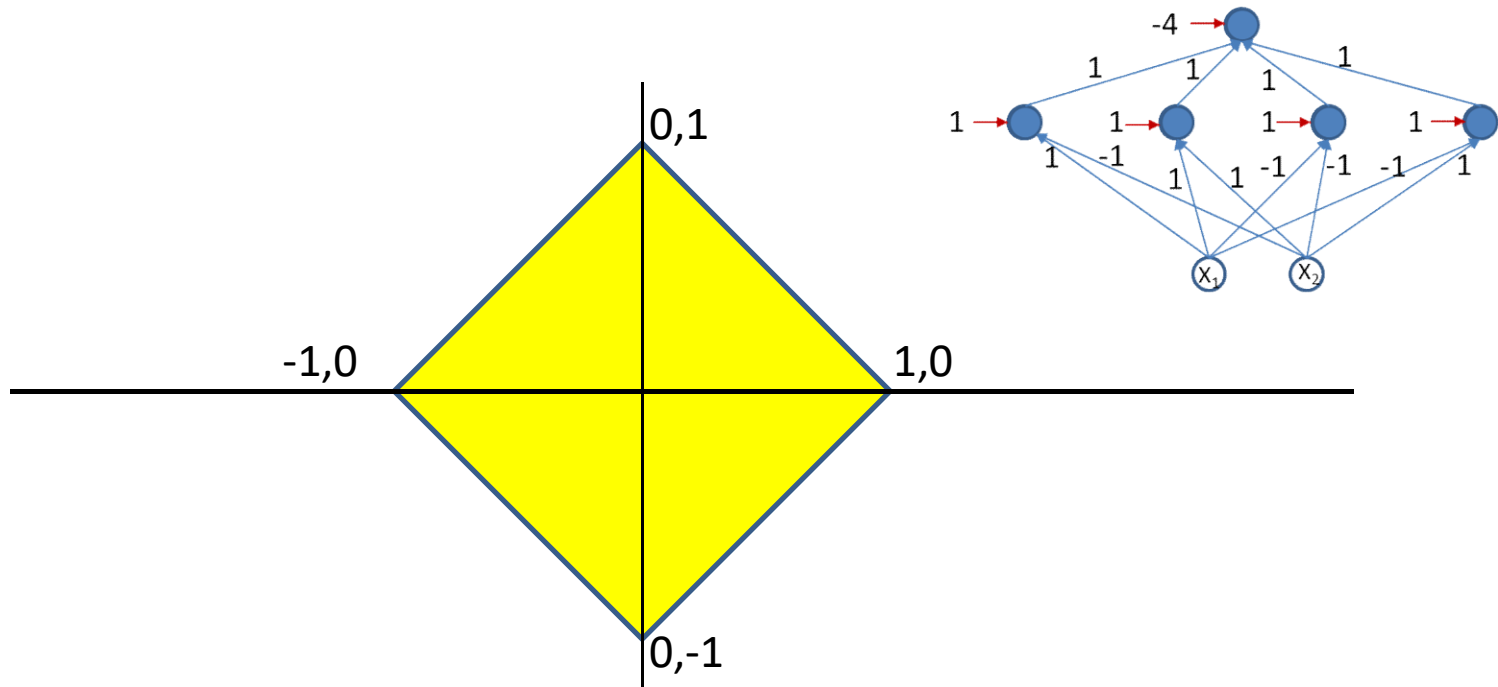
Assuming simple perceptrons:
output = 1 if $\sum_i w_i x_i + b_i \geq 0$, else 0

Option 1: Construct by hand



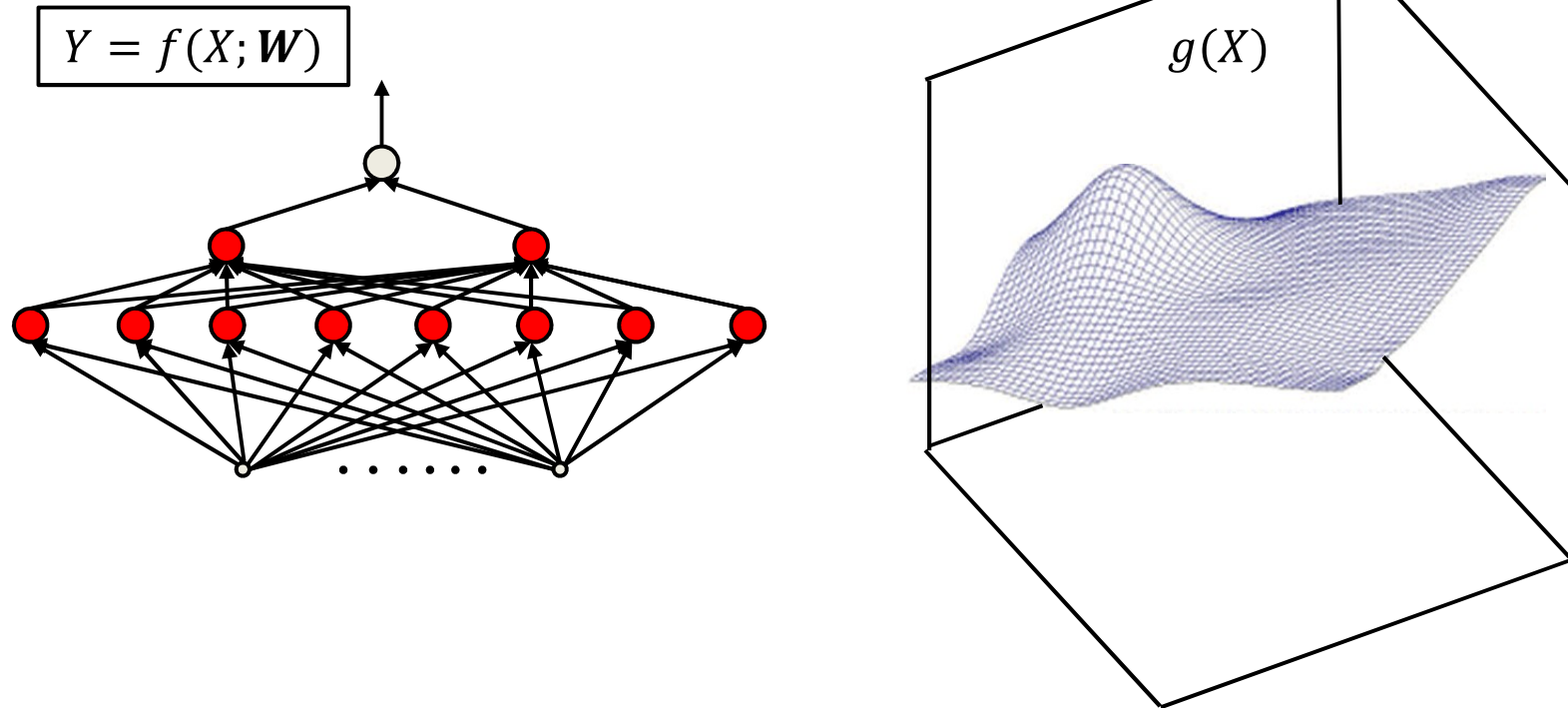
Assuming simple perceptrons:
output = 1 if $\sum_i w_i x_i + b_i \geq 0$, else 0

Option 1: Construct by hand



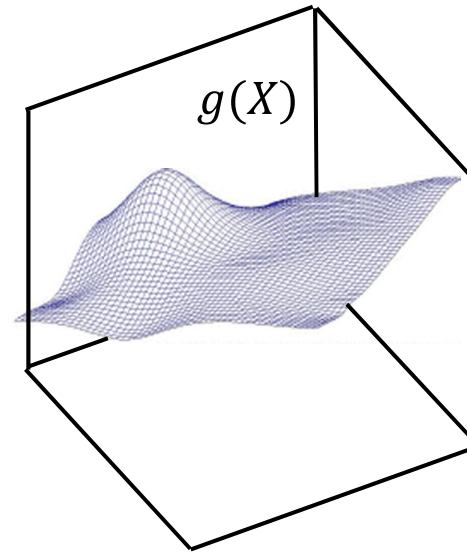
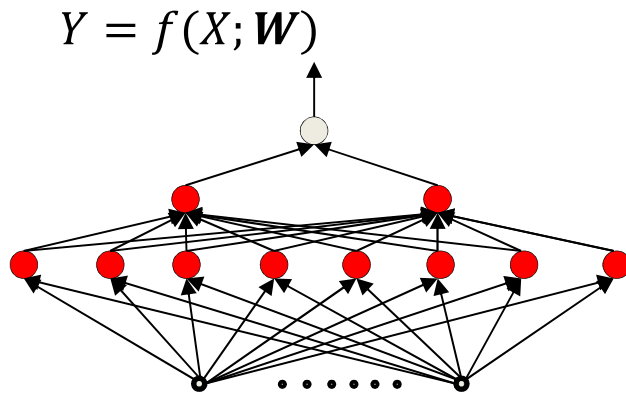
- Given a function, *handcraft* a network to satisfy it
- E.g.: Build an MLP to classify this decision boundary
- Not possible for all but the simplest problems..

Option 2: Automatic estimation of an MLP



- More generally, *given* the function $g(X)$ to model, we can *derive* the parameters of the network to model it, through computation

How to learn a network?

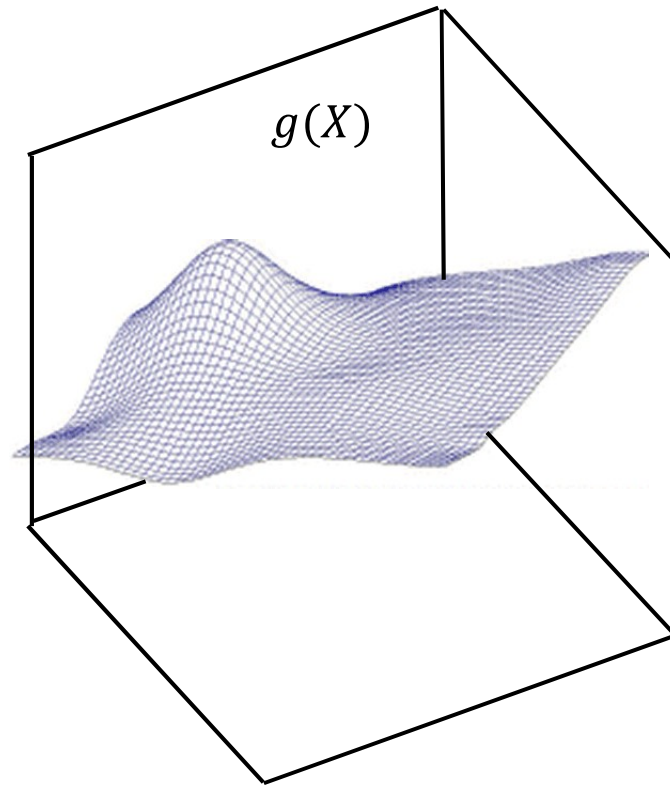


- When $f(X; \mathbf{W})$ has the capacity to exactly represent $g(X)$

$$\widehat{\mathbf{W}} = \operatorname{argmin}_{\mathbf{W}} \int_X \operatorname{div}(f(X; \mathbf{W}), g(X)) dX$$

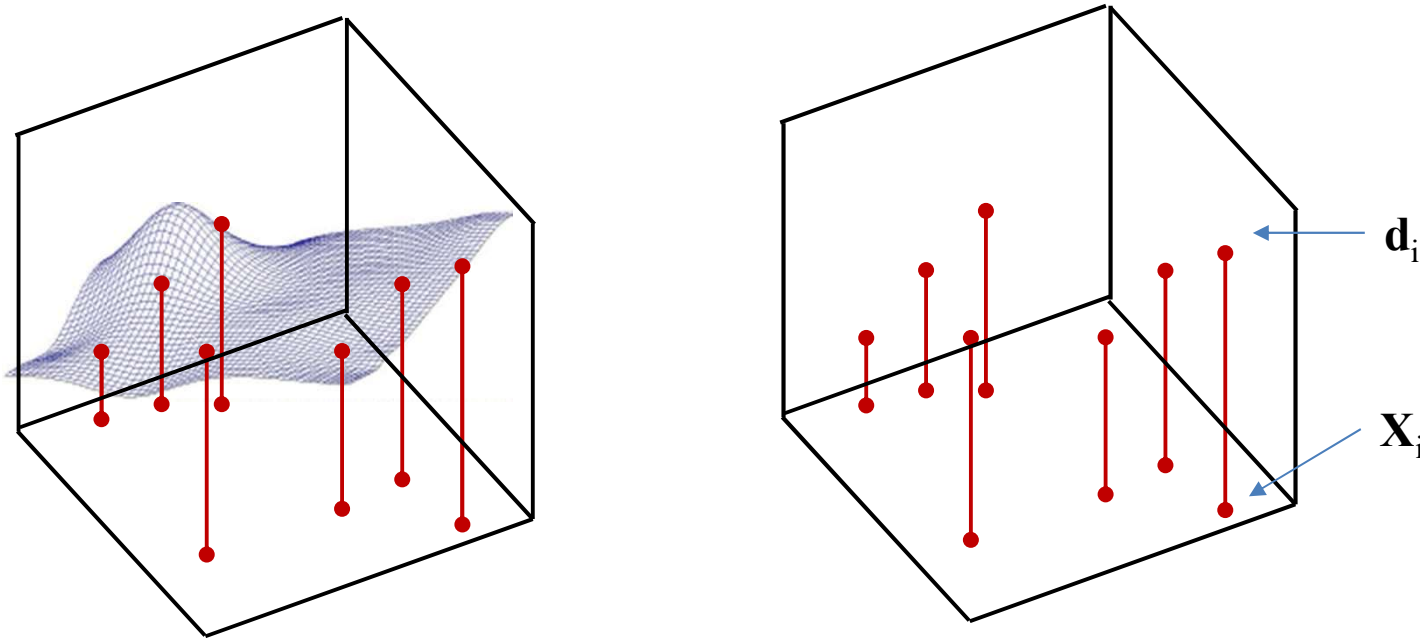
- $\operatorname{div}()$ is a *divergence* function that goes to zero when $f(X; \mathbf{W}) = g(X)$

Problem $g(X)$ is unknown



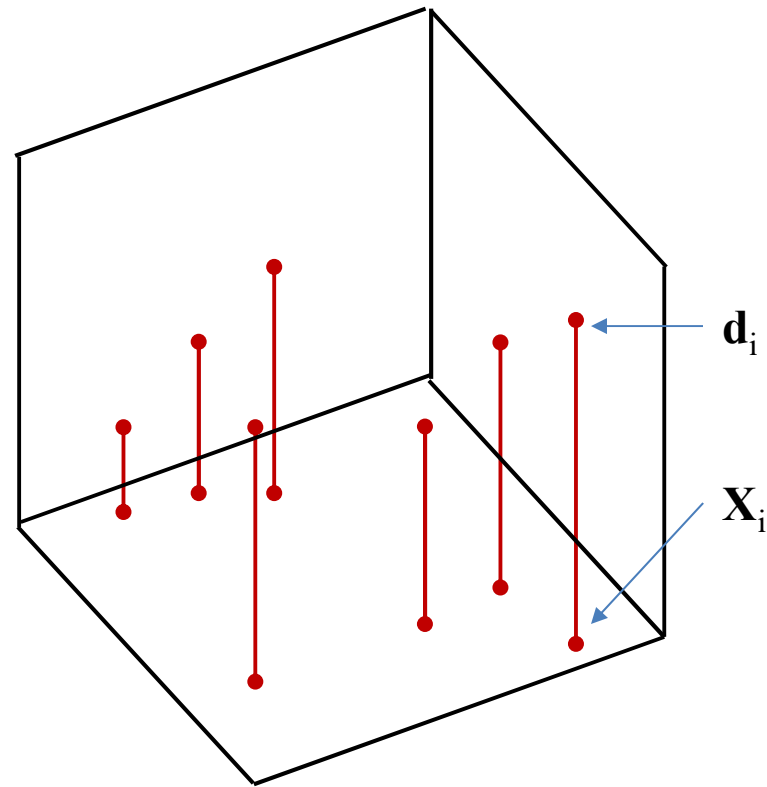
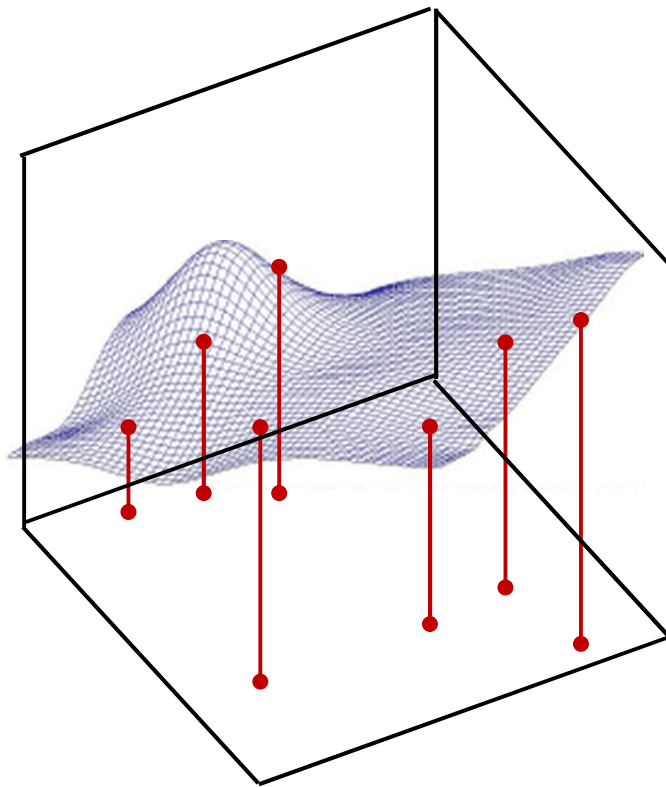
- Function $g(X)$ must be fully specified
 - Known *everywhere*, i.e. for *every* input X
- **In practice we will not have such specification**

Sampling the function



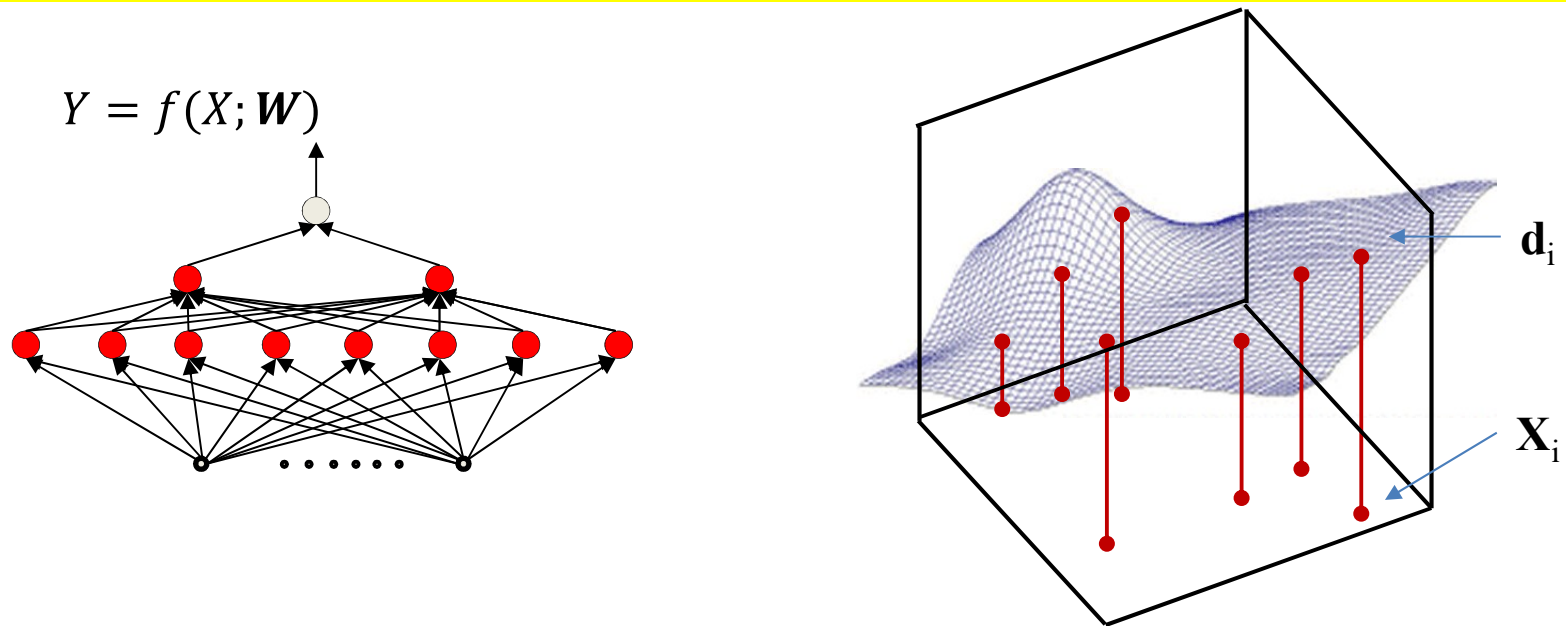
- *Sample $g(X)$*
 - Basically, get input-output pairs for a number of samples of input X_i
 - Many samples (X_i, d_i) , where $d_i = g(X_i) + \text{noise}$
 - Good sampling: the samples of X will be drawn from $P(X)$
- Very easy to do in most problems: just gather training data
 - E.g. set of images and their class labels
 - E.g. speech recordings and their transcription

Drawing samples



- We must *learn* the *entire* function from these few examples
 - The “training” samples

Learning the function



- Estimate the network parameters to “fit” the training points exactly
 - Assuming network architecture is sufficient for such a fit
 - Assuming unique output d at any \mathbf{X}
 - And hopefully the resulting function is also correct where we *don't* have training samples

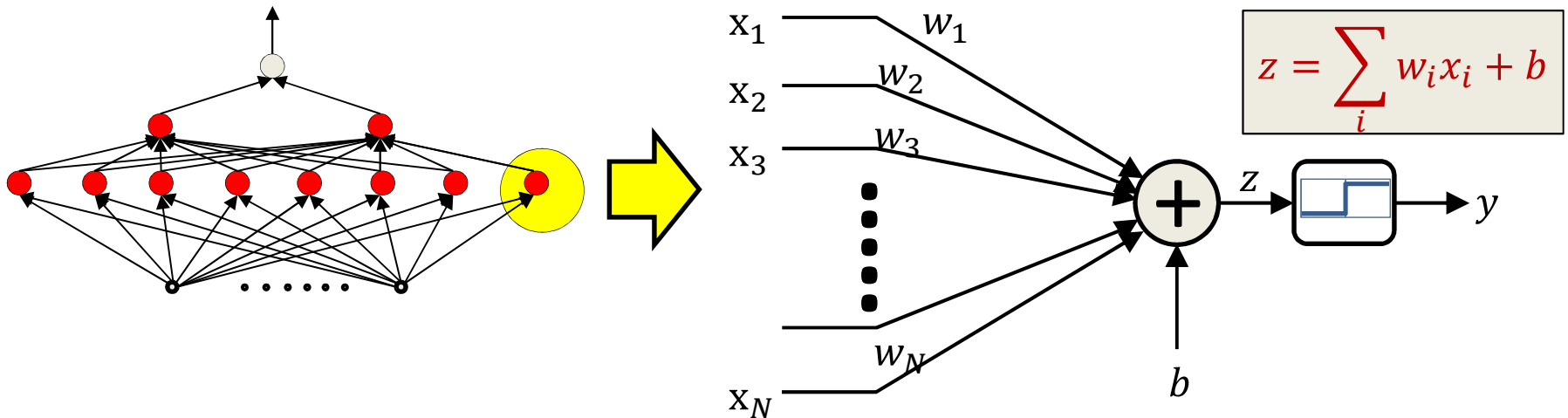
Story so far

- “Learning” a neural network == determining the parameters of the network (weights and biases) required for it to model a desired function
 - The network must have sufficient capacity to model the function
- Ideally, we would like to optimize the network to represent the desired function everywhere
- However this requires knowledge of the function everywhere
- Instead, we draw “input-output” *training* instances from the function and estimate network parameters to “fit” the input-output relation at these instances
 - And hope it fits the function elsewhere as well

Lets begin with a simple task

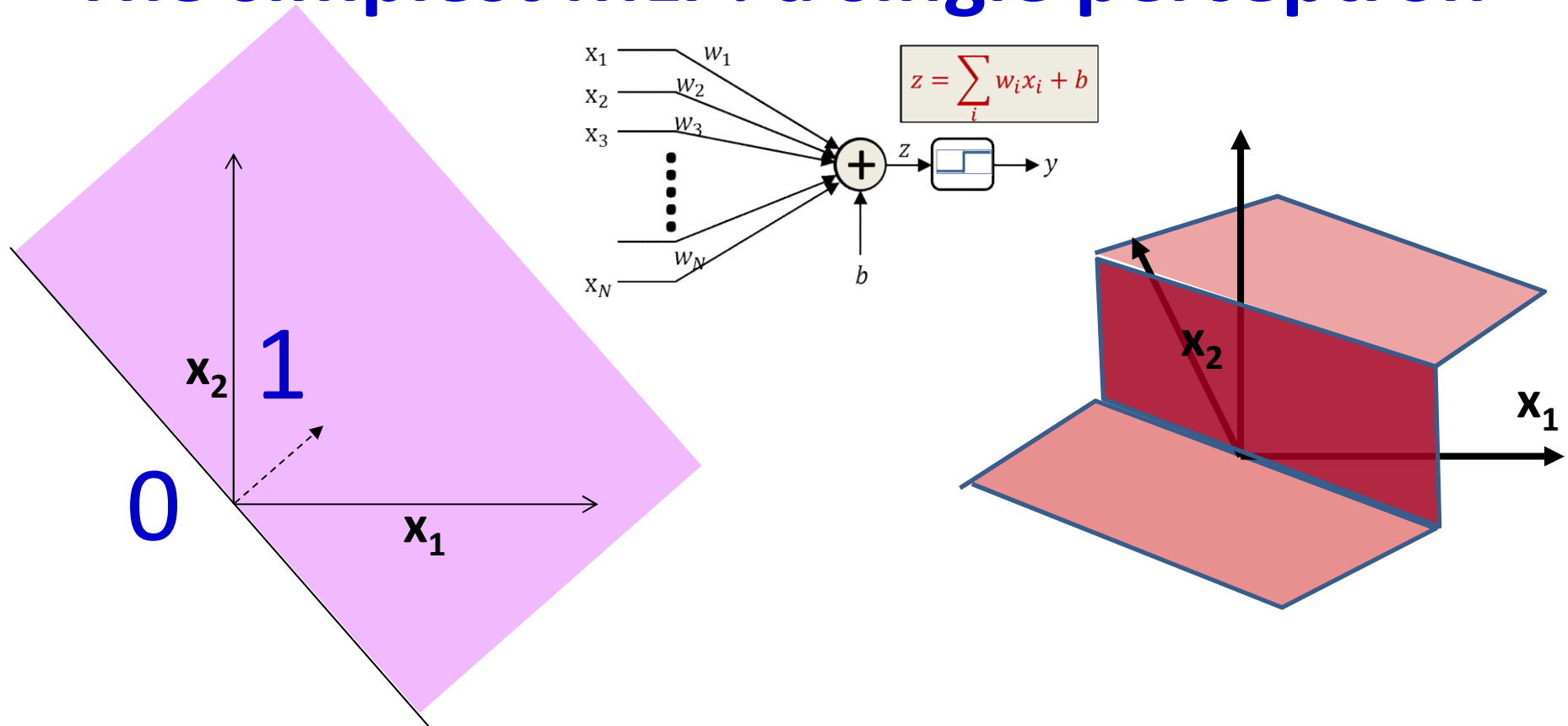
- Learning a *classifier*
 - Simpler than regressions
- This was among the earliest problems addressed using MLPs
- Specifically, consider *binary* classification
 - Generalizes to multi-class

History: The original MLP



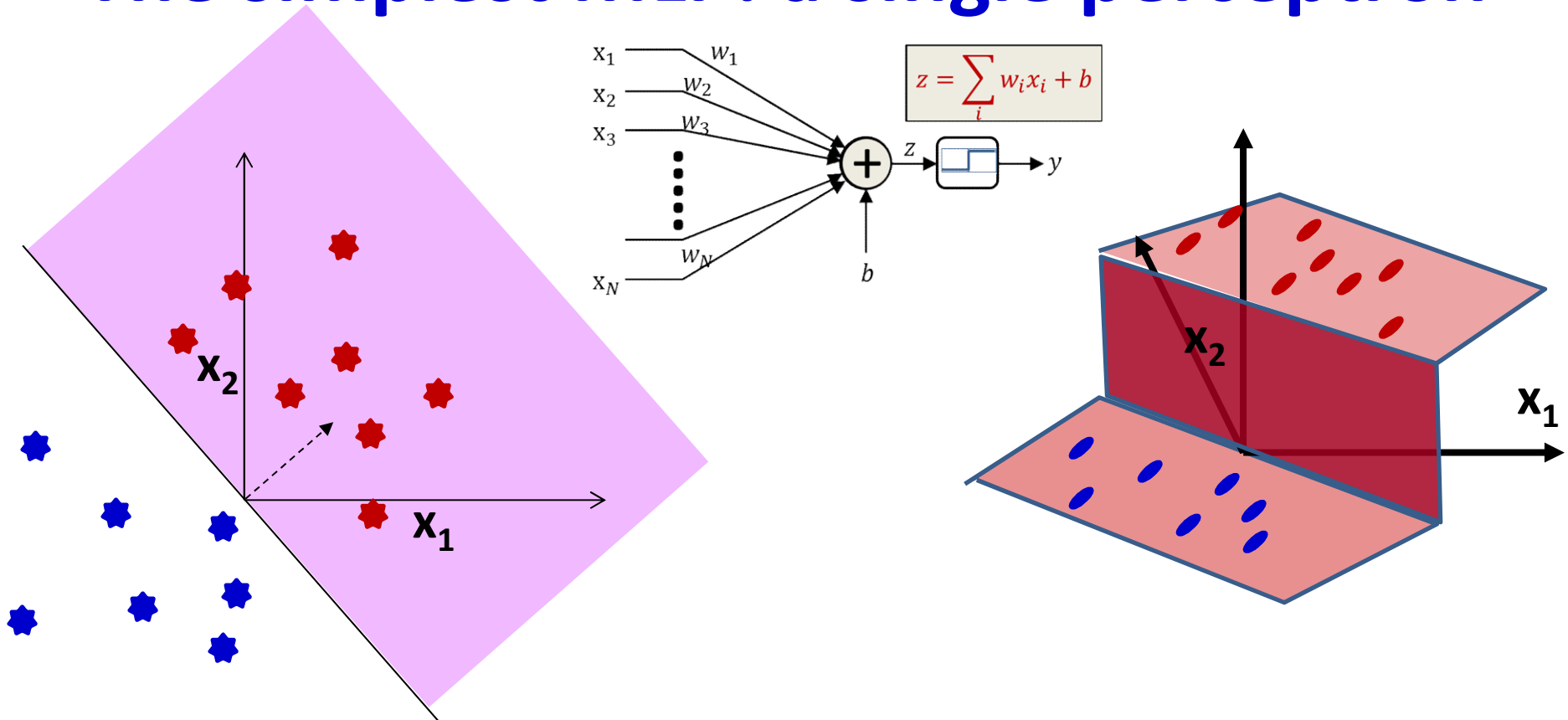
- The original MLP as proposed by Minsky: a network of threshold units
 - But how do you train it?
 - Given only “training” instances of input-output pairs

The simplest MLP: a single perceptron



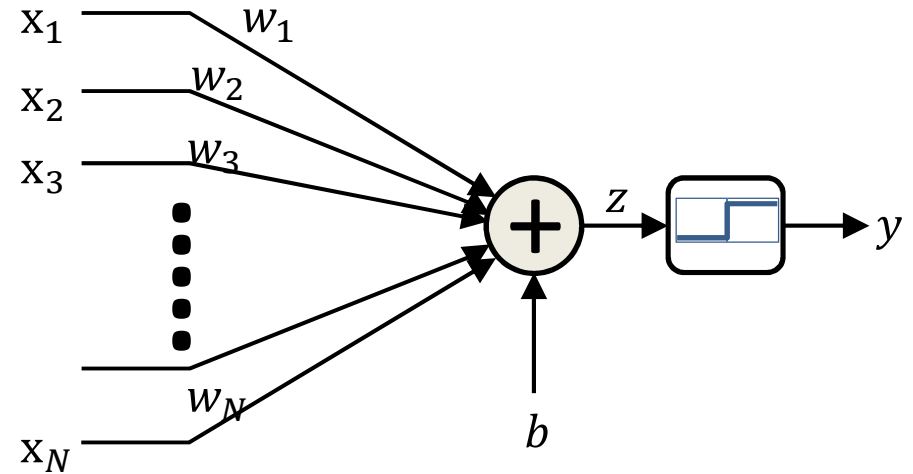
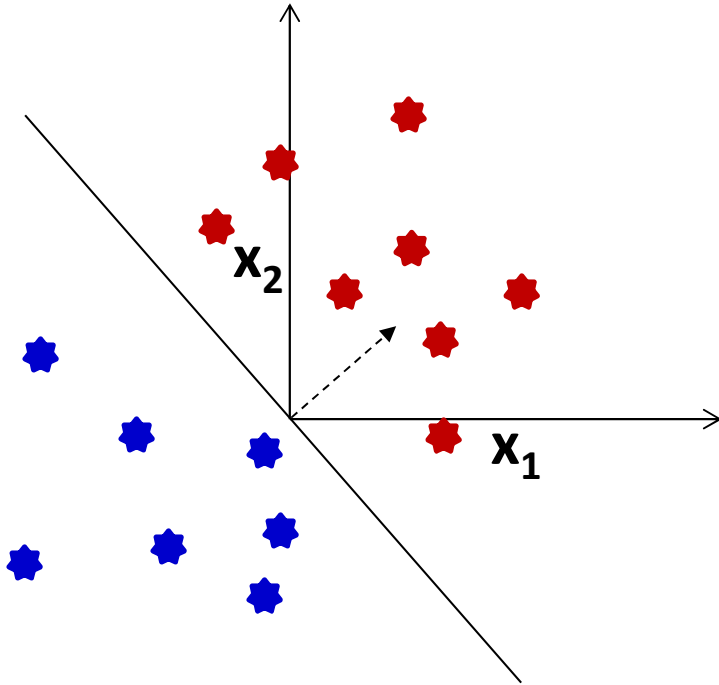
- Learn this function
 - A step function across a hyperplane

The simplest MLP: a single perceptron



- Learn this function
 - A step function across a hyperplane
 - Given only samples from it

Learning the perceptron

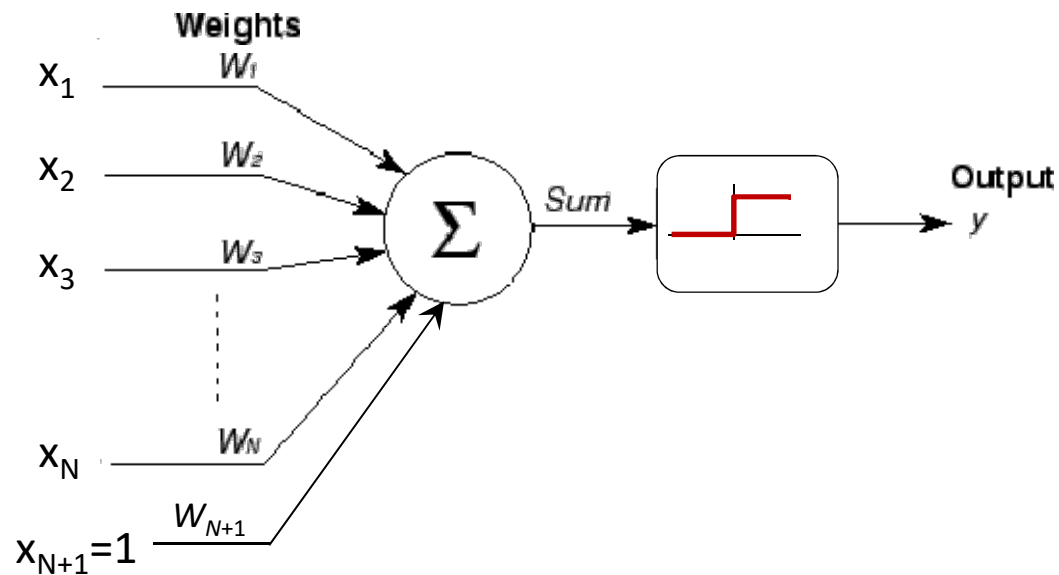


- Given a number of input output pairs, learn the weights and bias

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^N w_i X_i + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

- Learn $W = [w_1 \dots w_N]$ and b , given several (X, y) pairs

Restating the perceptron



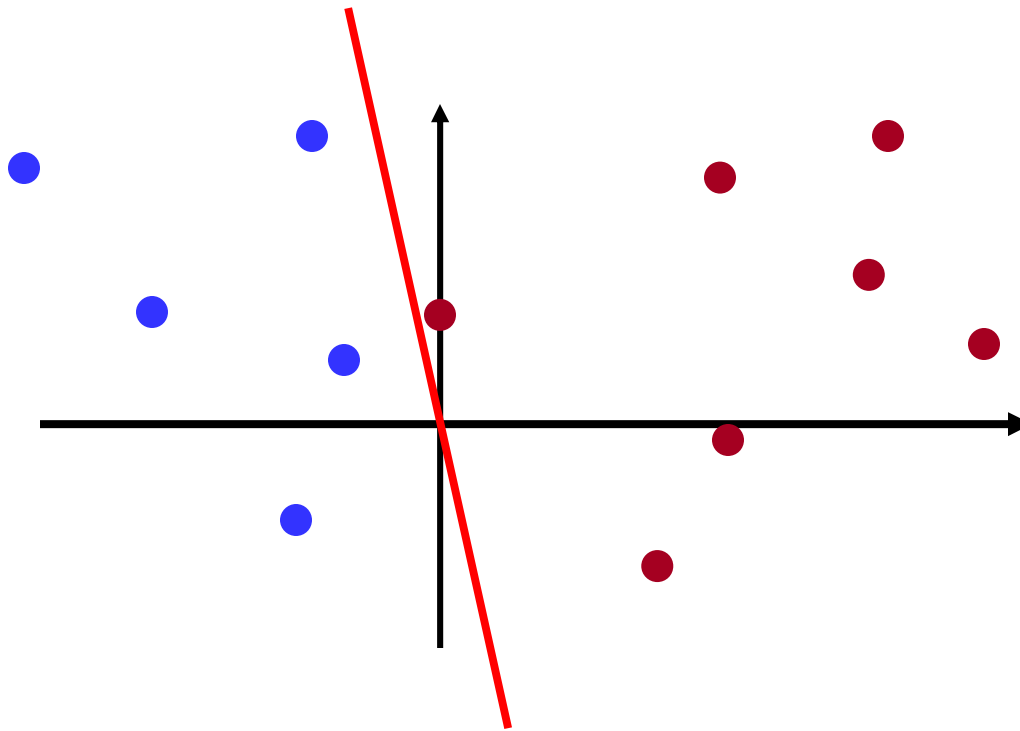
- Restating the perceptron equation by adding another dimension to X

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{N+1} w_i X_i \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where $X_{N+1} = 1$

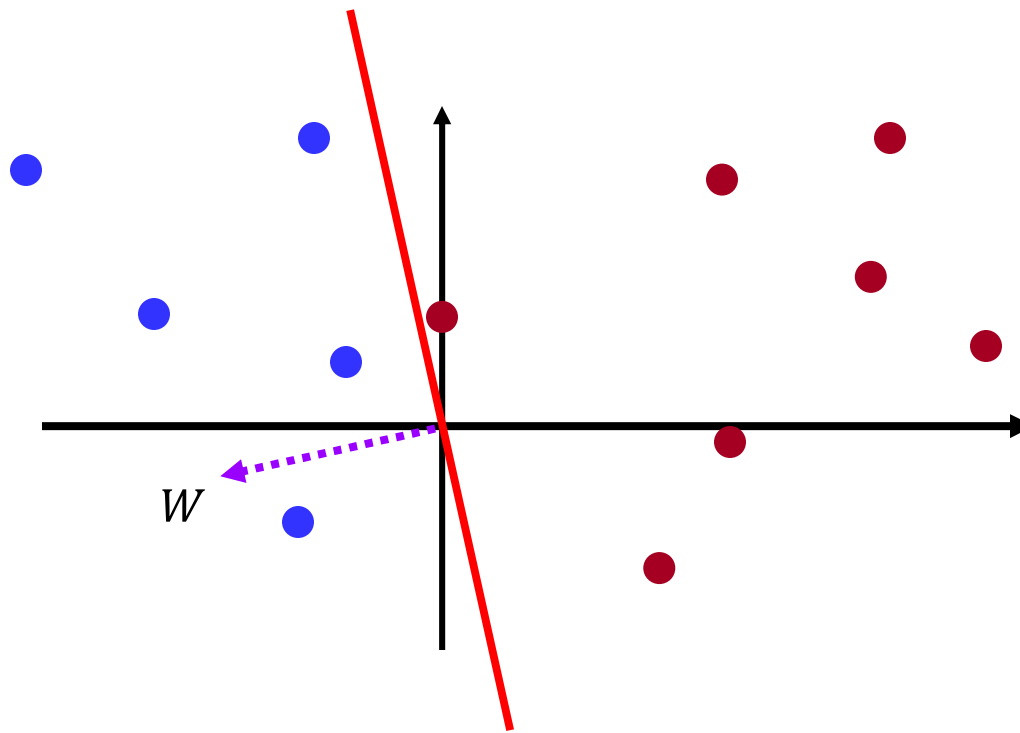
- Note that the boundary $\sum_{i=1}^{N+1} w_i X_i \geq 0$ is now a hyperplane through origin

The Perceptron Problem



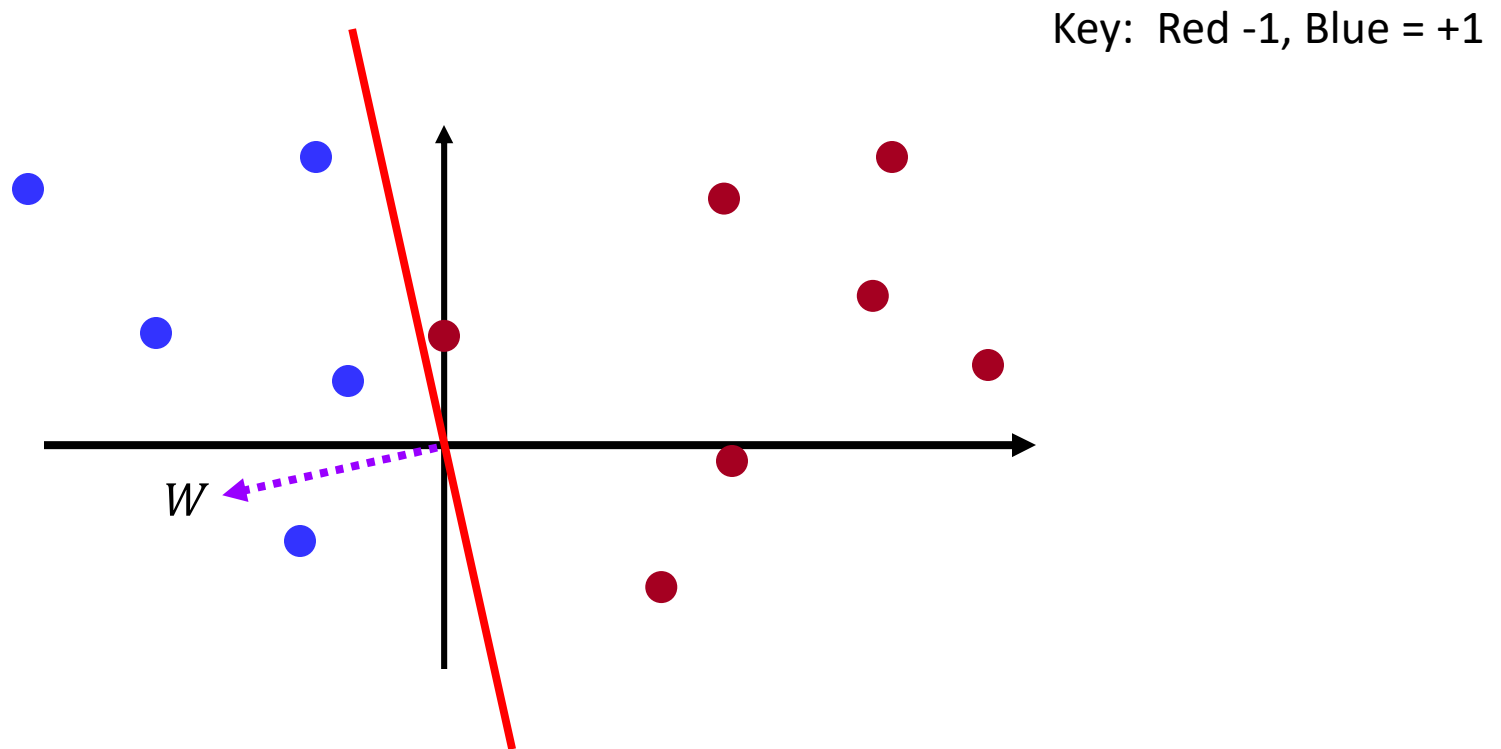
- Find the hyperplane $\sum_{i=1}^{N+1} w_i X_i = 0$ that perfectly separates the two groups of points

The Perceptron Problem



- Find the hyperplane $\sum_{i=1}^{N+1} w_i X_i = 0$ that perfectly separates the two groups of points
 - Note: $W = [w_1, w_2, \dots, w_{N+1}]^T$ is a vector that is orthogonal to the hyperplane
 - In fact the equation for the hyperplane itself means “the set of all X s that are orthogonal to W ” ($\sum_{i=1}^{N+1} w_i X_i = W^T X = 0$)

The Perceptron Problem



- Learning the perceptron: Find the weights vector W such that $W^T X$ is positive for all blue dots and negative for all red ones

Perceptron Algorithm: Summary

- Cycle through the training instances
- Only update W on misclassified instances
- If instance misclassified:
 - If instance is positive class (positive misclassified as negative)

$$W = W + X_i$$

- If instance is negative class (negative misclassified as positive)

$$W = W - X_i$$

Perceptron Learning Algorithm

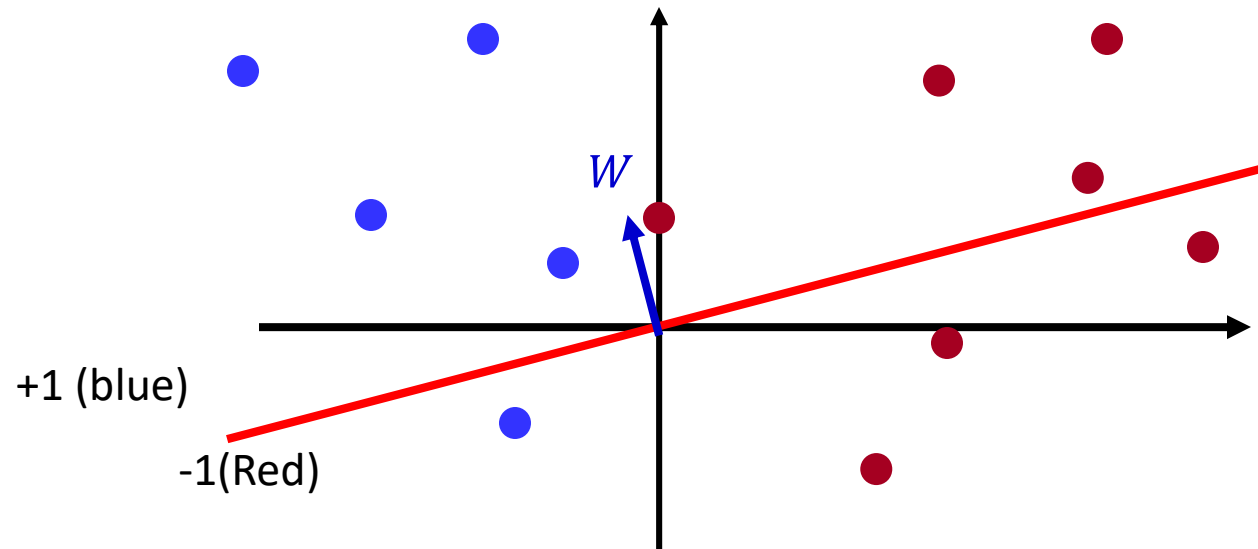
- Given N training instances $(X_1, Y_1), (X_2, Y_2), \dots, (X_N, Y_N)$
 - $Y_i = +1$ or -1

Using a +1/-1 representation
for classes to simplify
notation

- Initialize W
- Cycle through the training instances:

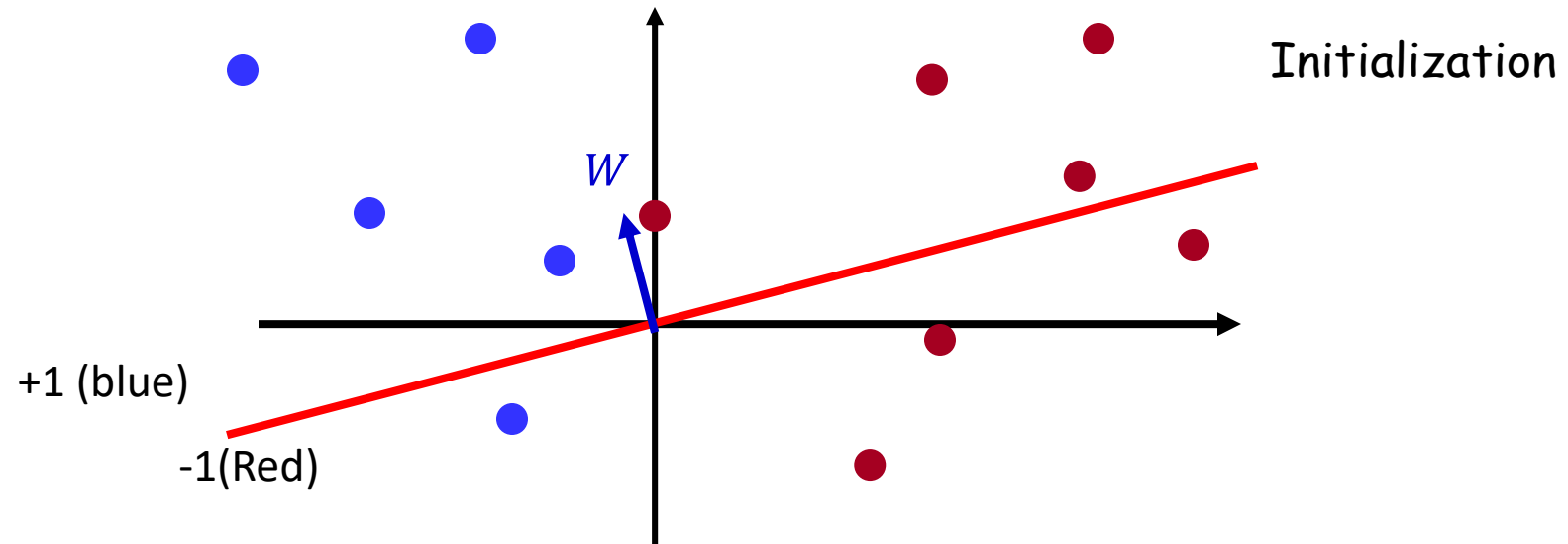
- do
 - For $i = 1 \dots N_{train}$
$$O(X_i) = \text{sign}(W^T X_i)$$
 - If $O(X_i) \neq Y_i$
$$W = W + Y_i X_i$$
- until no more classification errors

A Simple Method: The Perceptron Algorithm

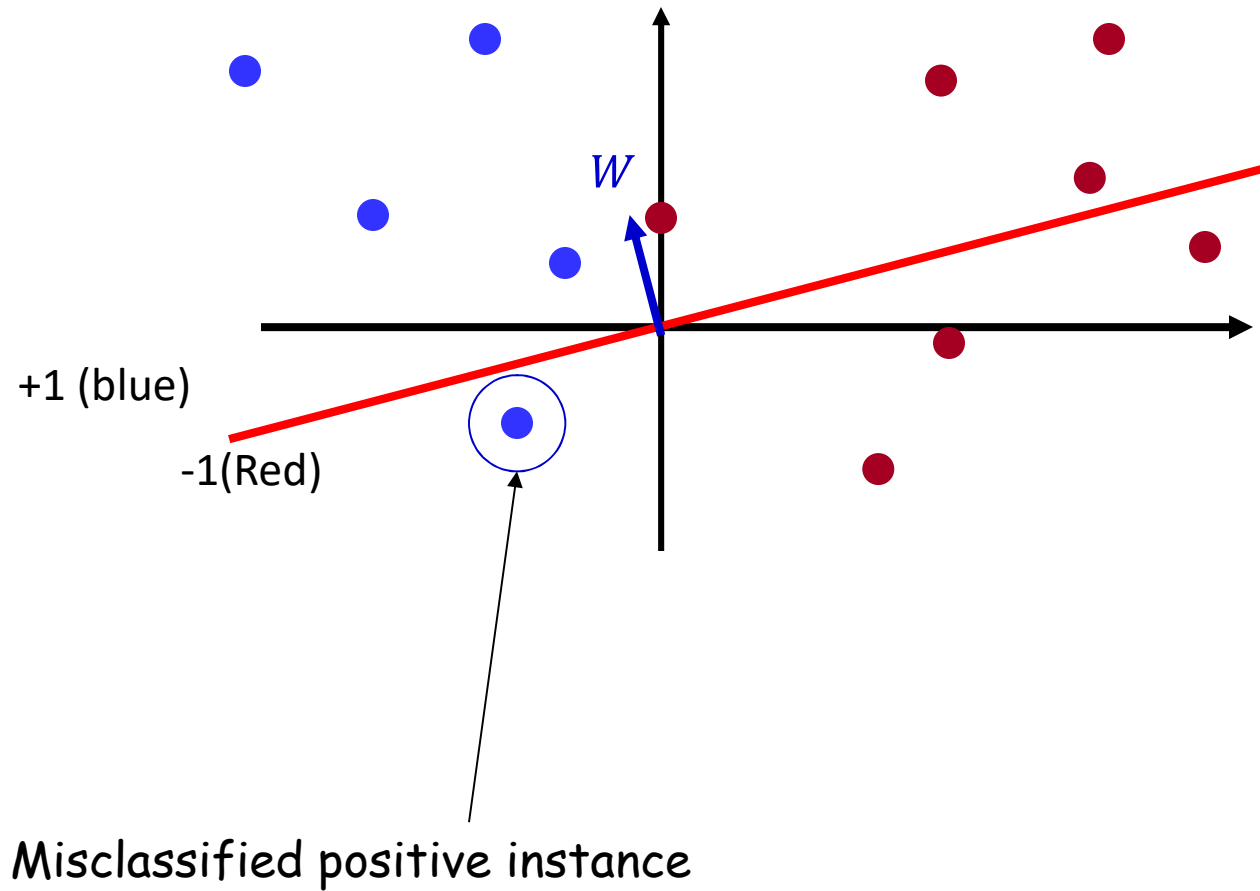


- **Initialize:** Randomly initialize the hyperplane
 - I.e. randomly initialize the normal vector W
- **Classification rule** $\text{sign}(W^T X)$
 - Vectors on the same side of the hyperplane as W will be assigned +1 class, and those on the other side will be assigned -1
- The random initial plane will make mistakes

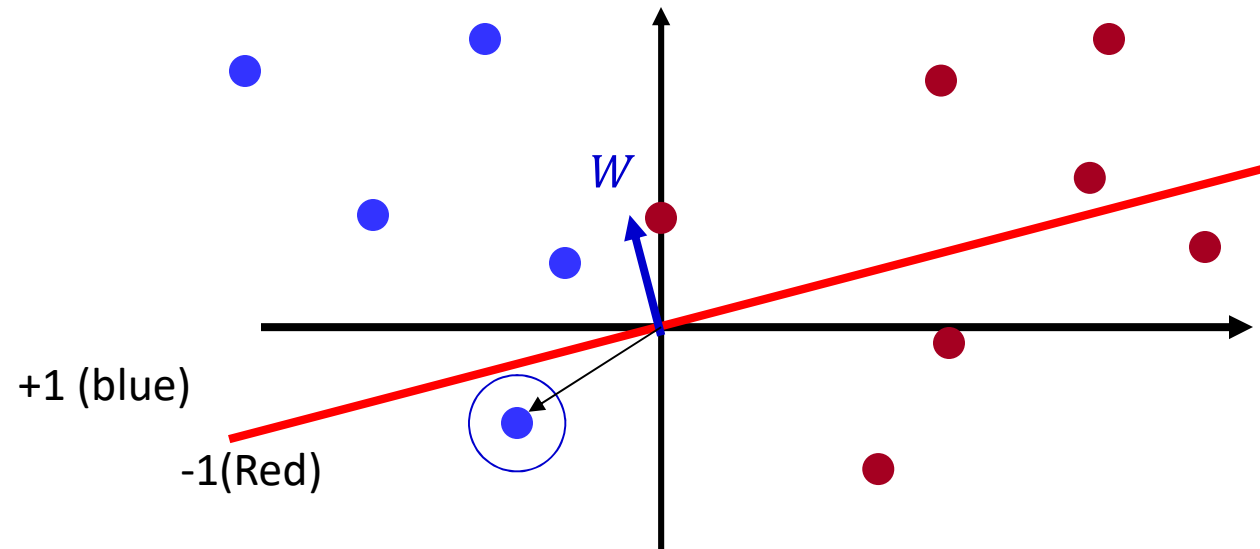
Perceptron Algorithm



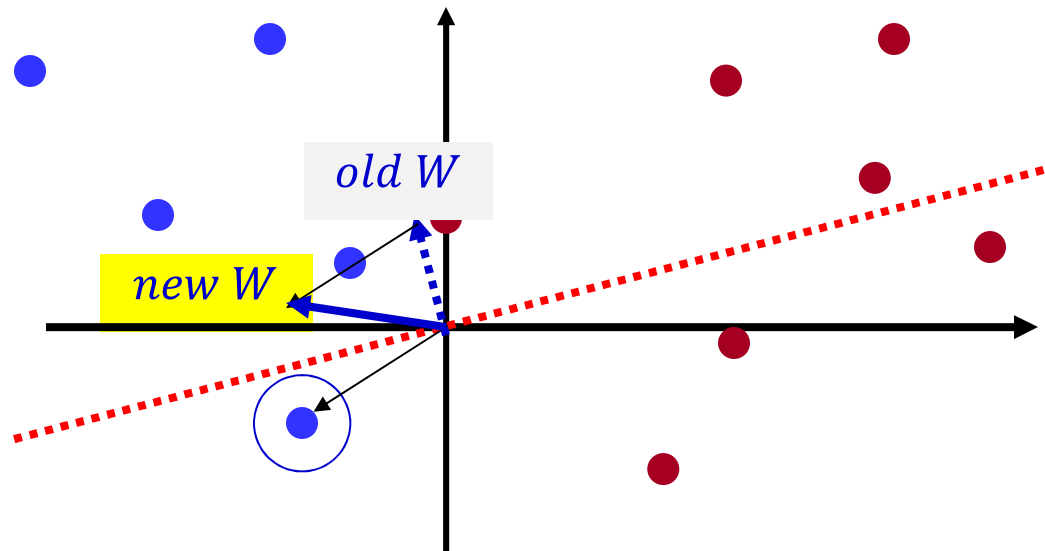
Perceptron Algorithm



Perceptron Algorithm



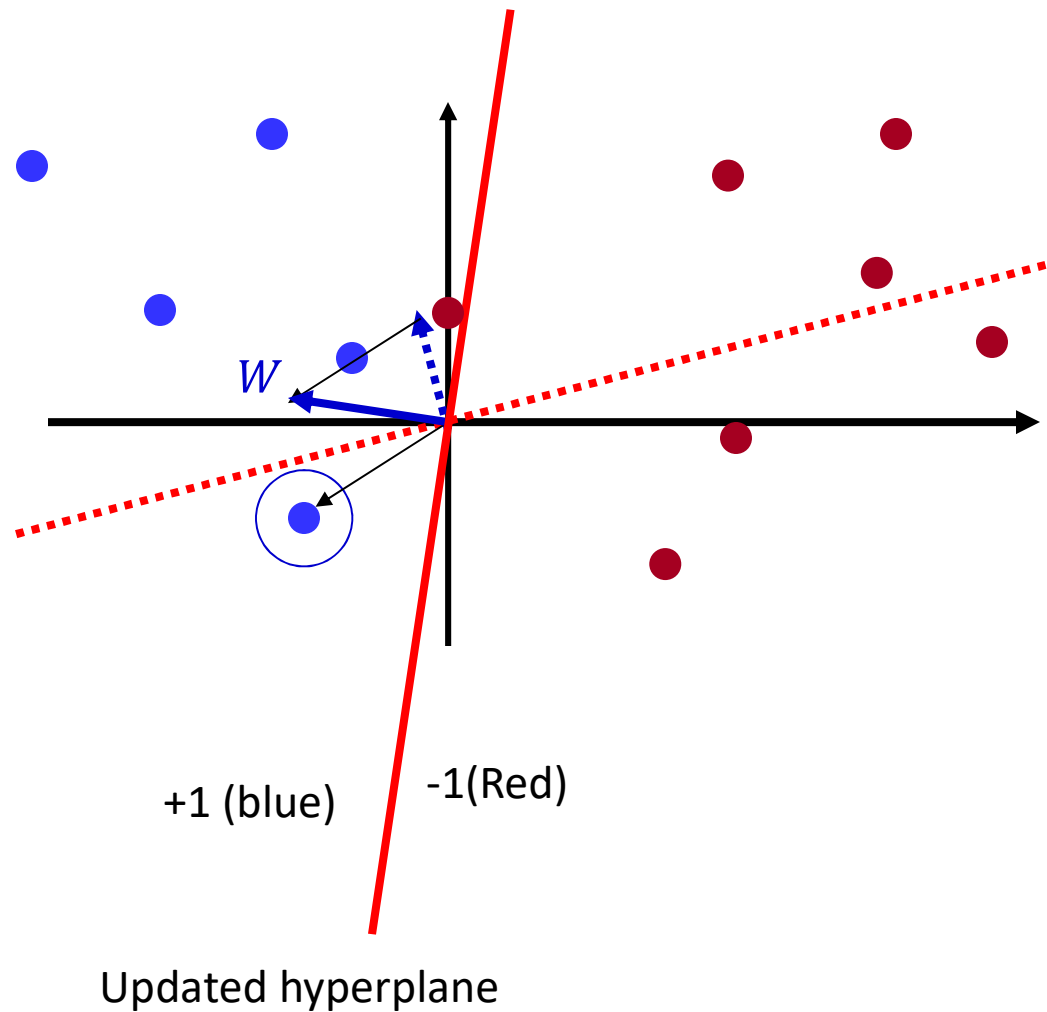
Perceptron Algorithm



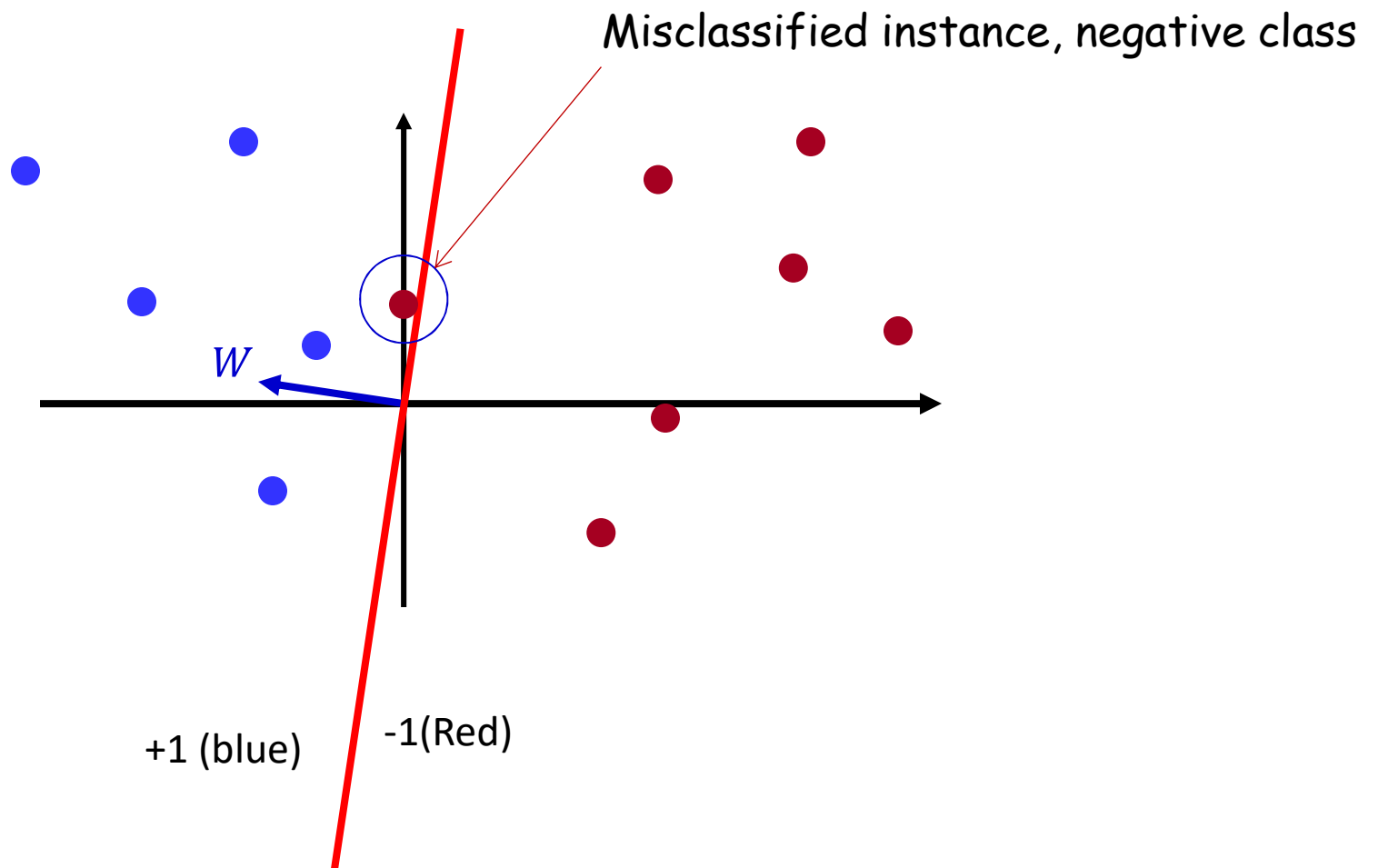
Updated weight vector

Misclassified *positive* instance, *add* it to W

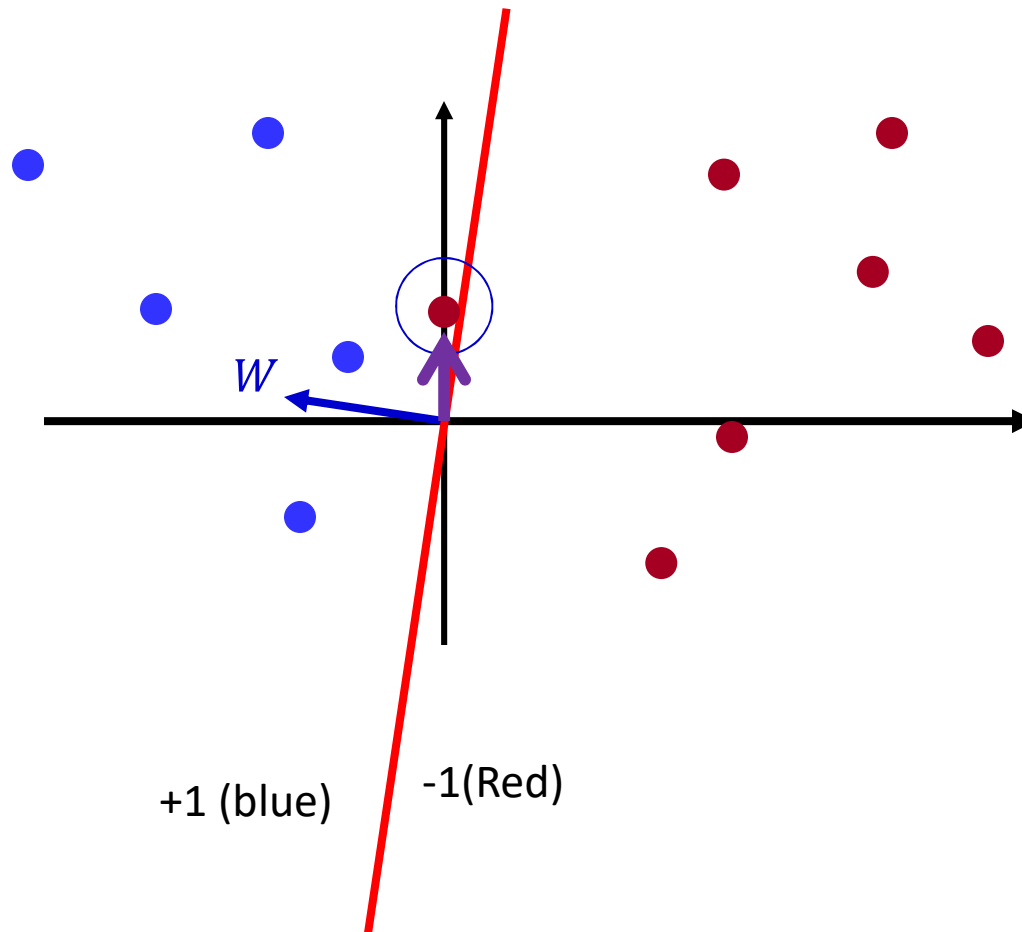
Perceptron Algorithm



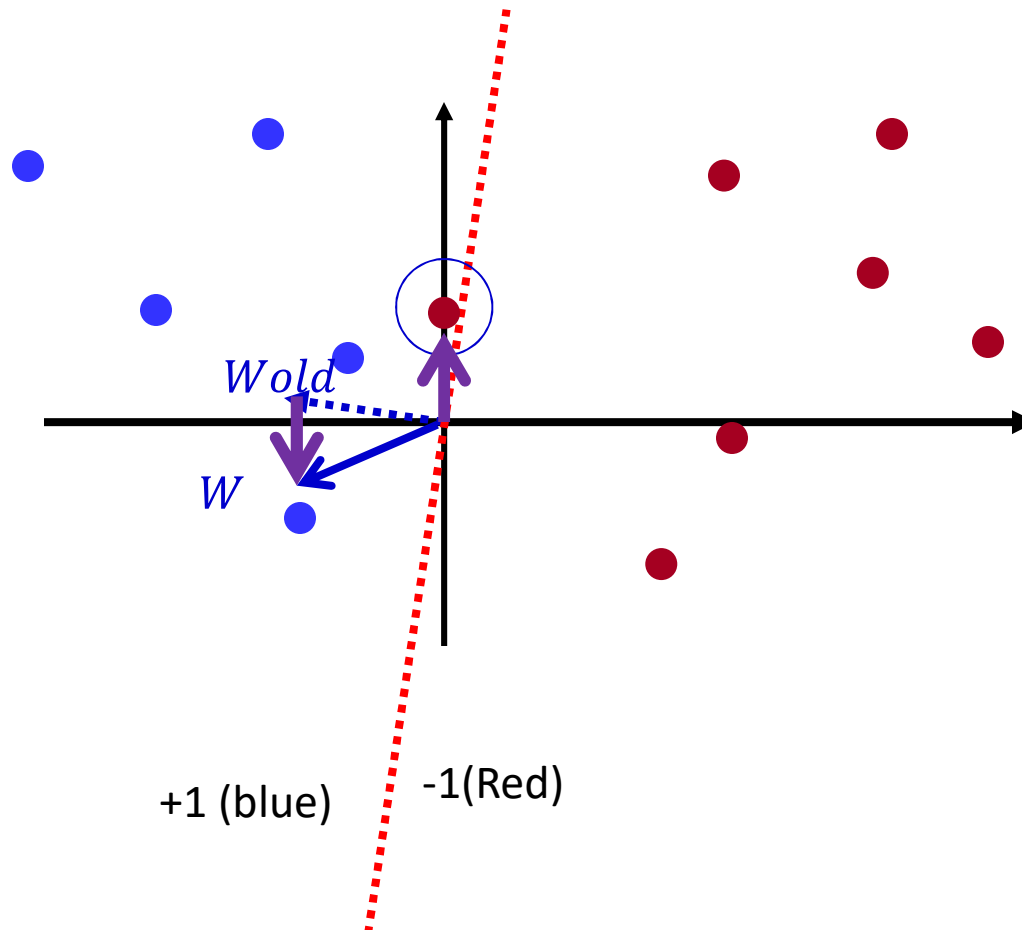
Perceptron Algorithm



Perceptron Algorithm

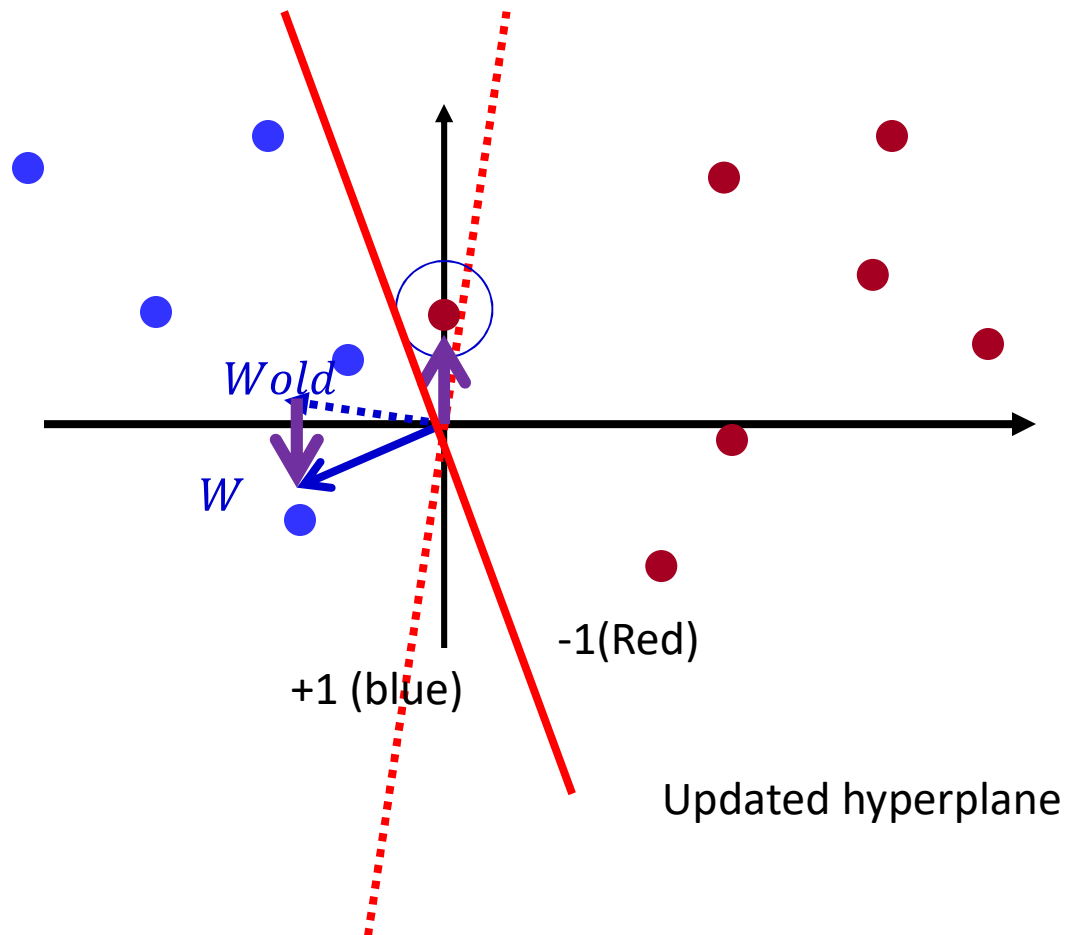


Perceptron Algorithm

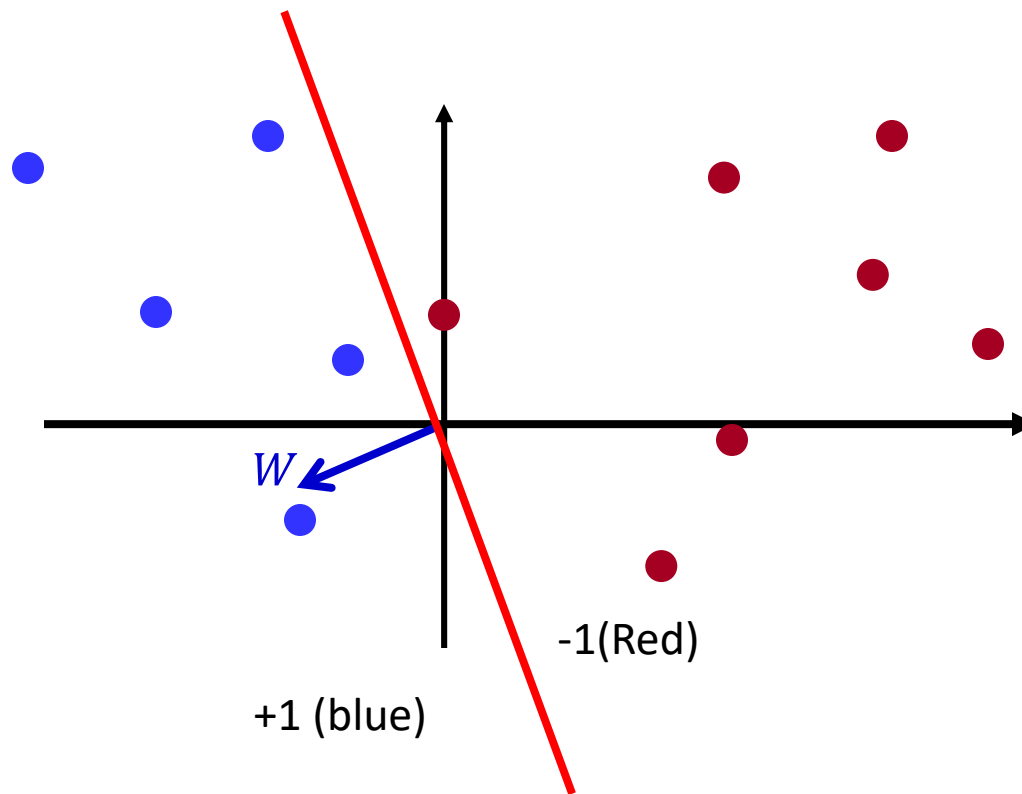


Misclassified *negative* instance, *subtract* it from W

Perceptron Algorithm



Perceptron Algorithm

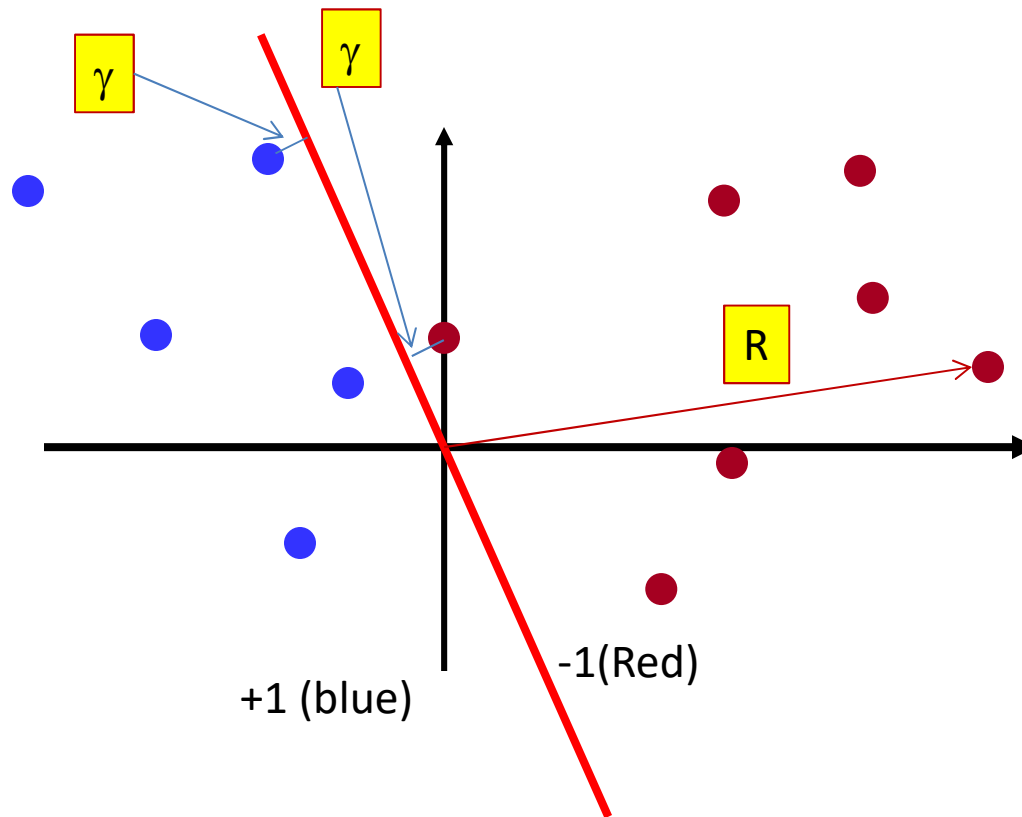


Perfect classification, no more updates

Convergence of Perceptron Algorithm

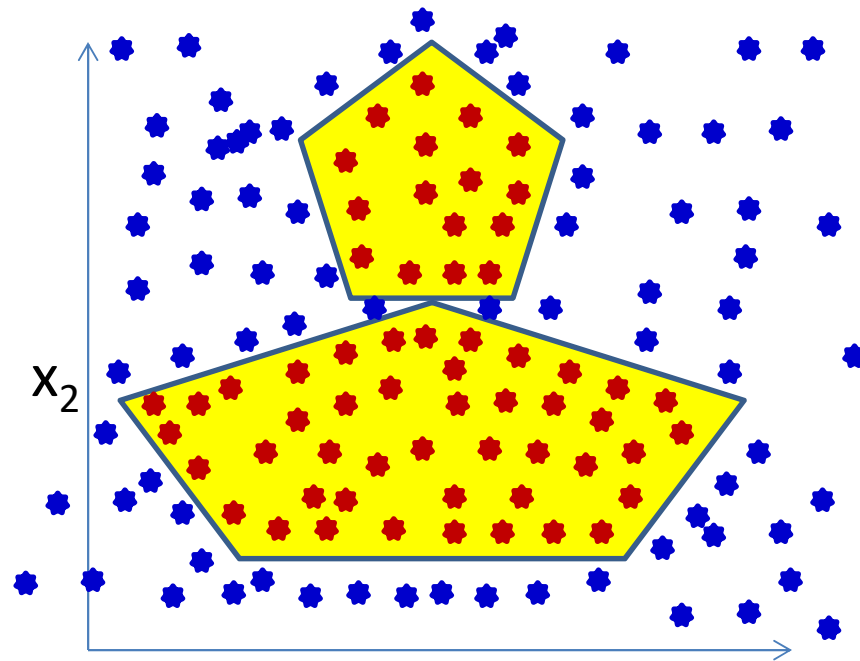
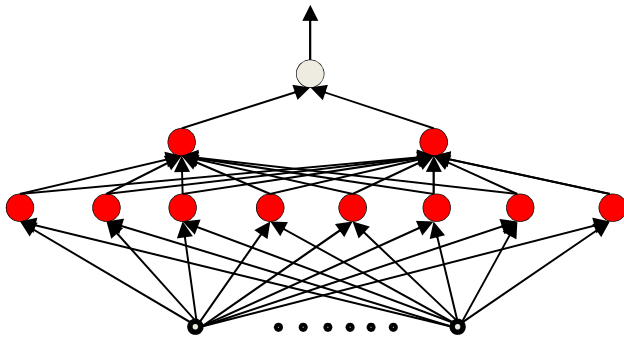
- Guaranteed to converge if classes are linearly separable
 - After no more than $\left(\frac{R}{\gamma}\right)^2$ misclassifications
 - Specifically when W is initialized to 0
 - R is length of longest training point
 - γ is the *best case* closest distance of a training point from the classifier
 - Same as the margin in an SVM
 - Intuitively – takes many increments of size γ to undo an error resulting from a step of size R

Perceptron Algorithm



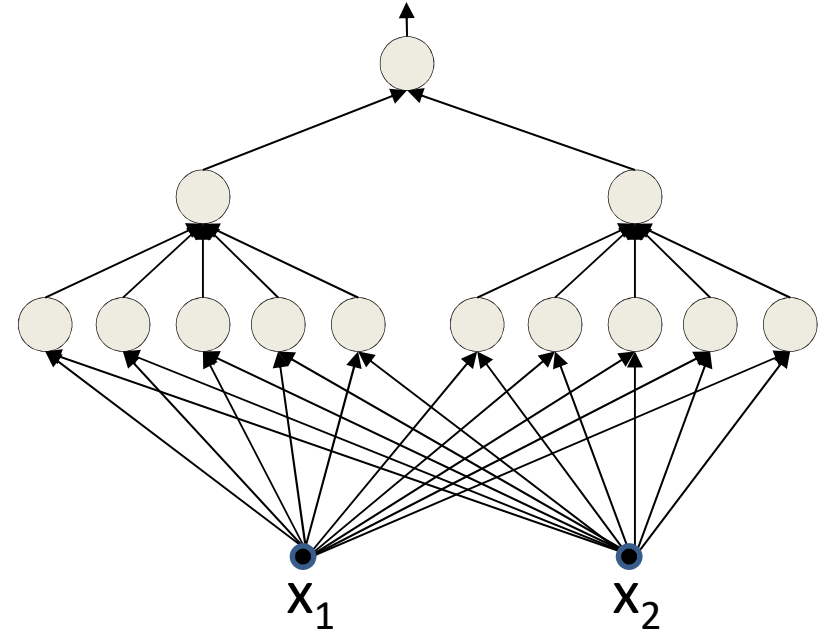
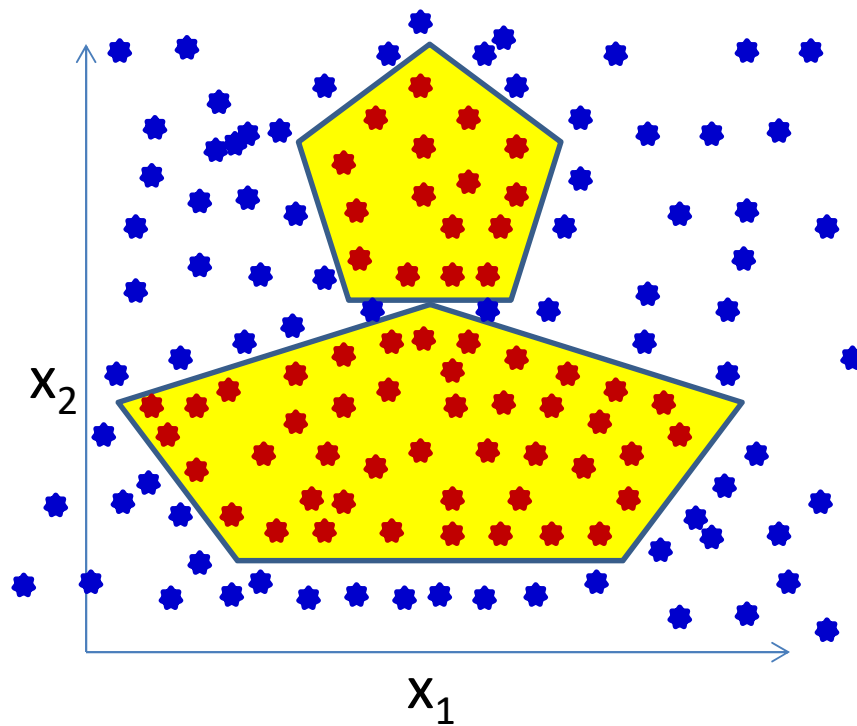
γ is the best-case margin
 R is the length of the longest vector

History: A more complex problem



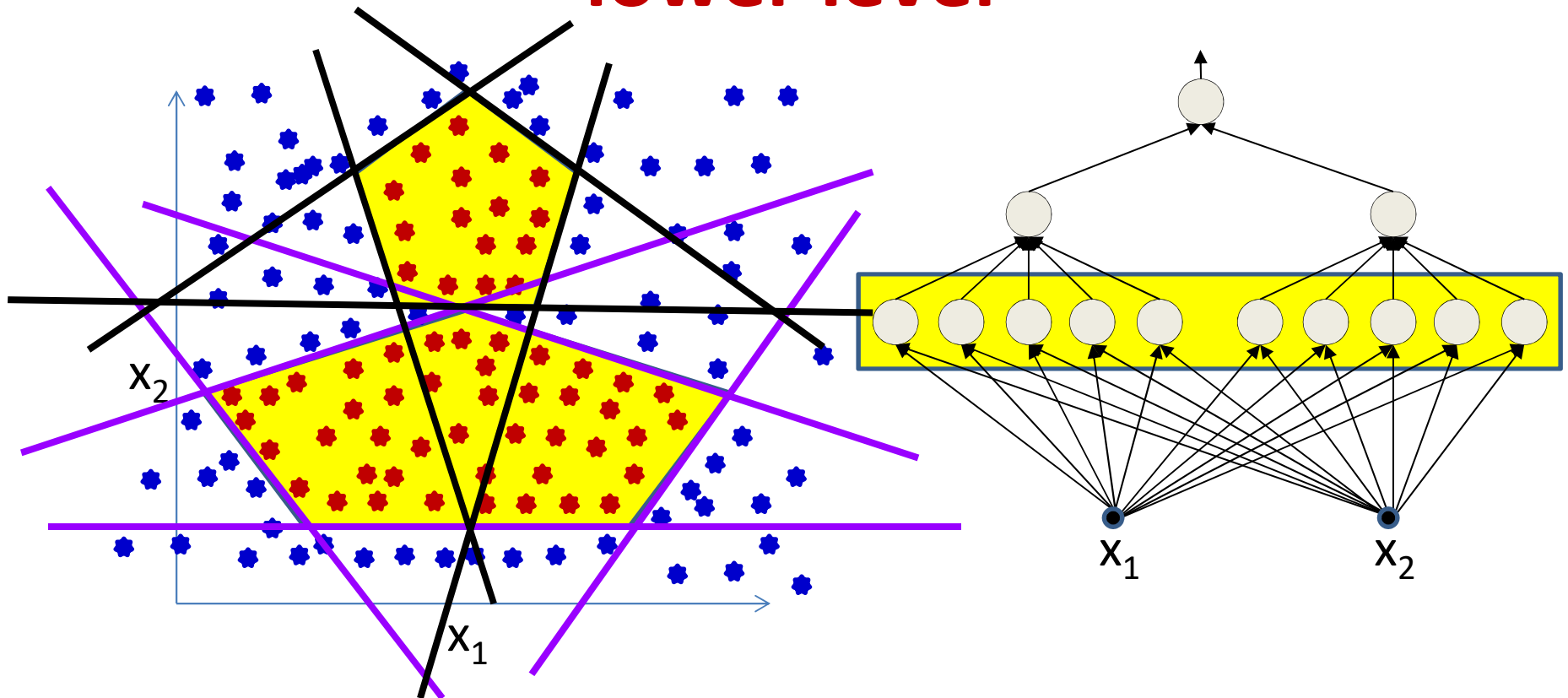
- Learn an *MLP* for this function
 - 1 in the yellow regions, 0 outside
- Using just the samples
- We know this can be perfectly represented using an MLP

More complex decision boundaries



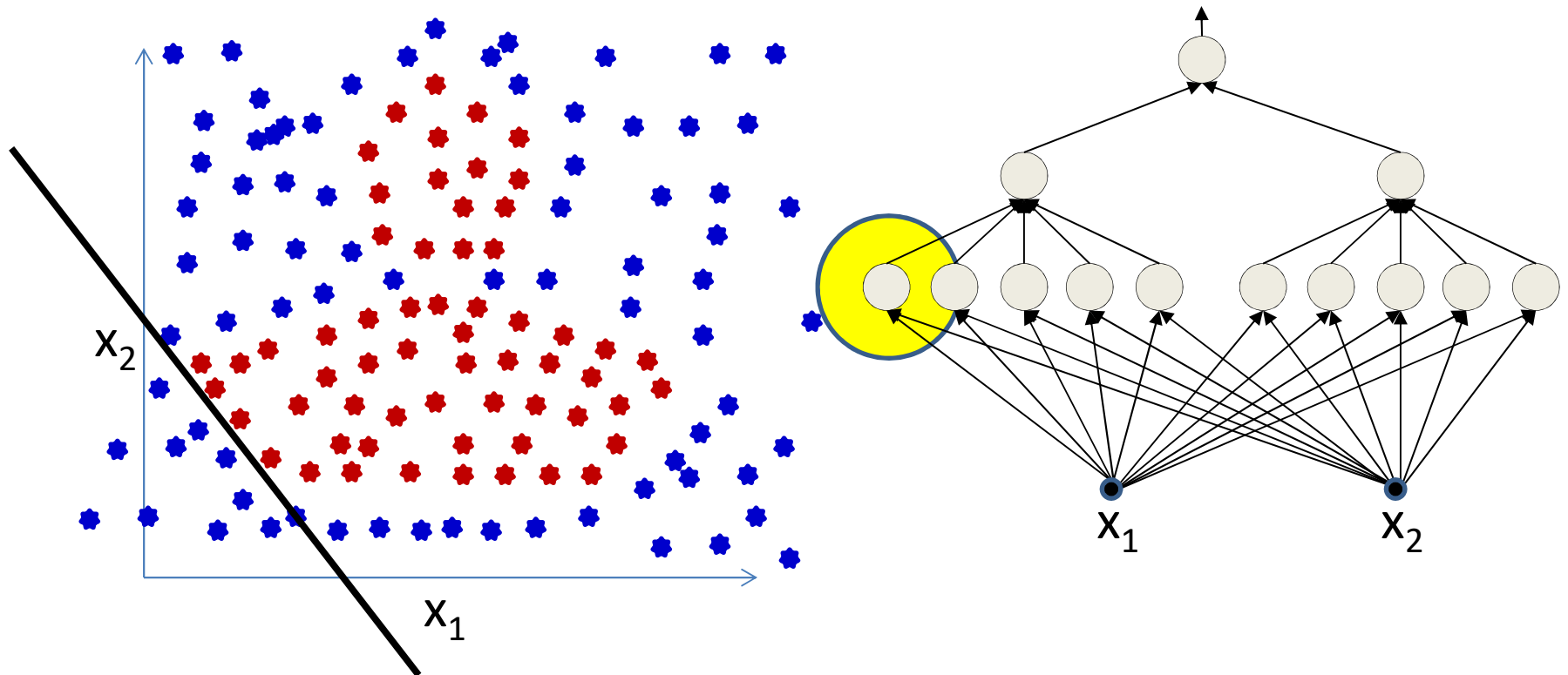
- Even using the perfect architecture
- Can we use the perceptron algorithm?
 - Making incremental corrections every time we encounter an error

The pattern to be learned at the lower level



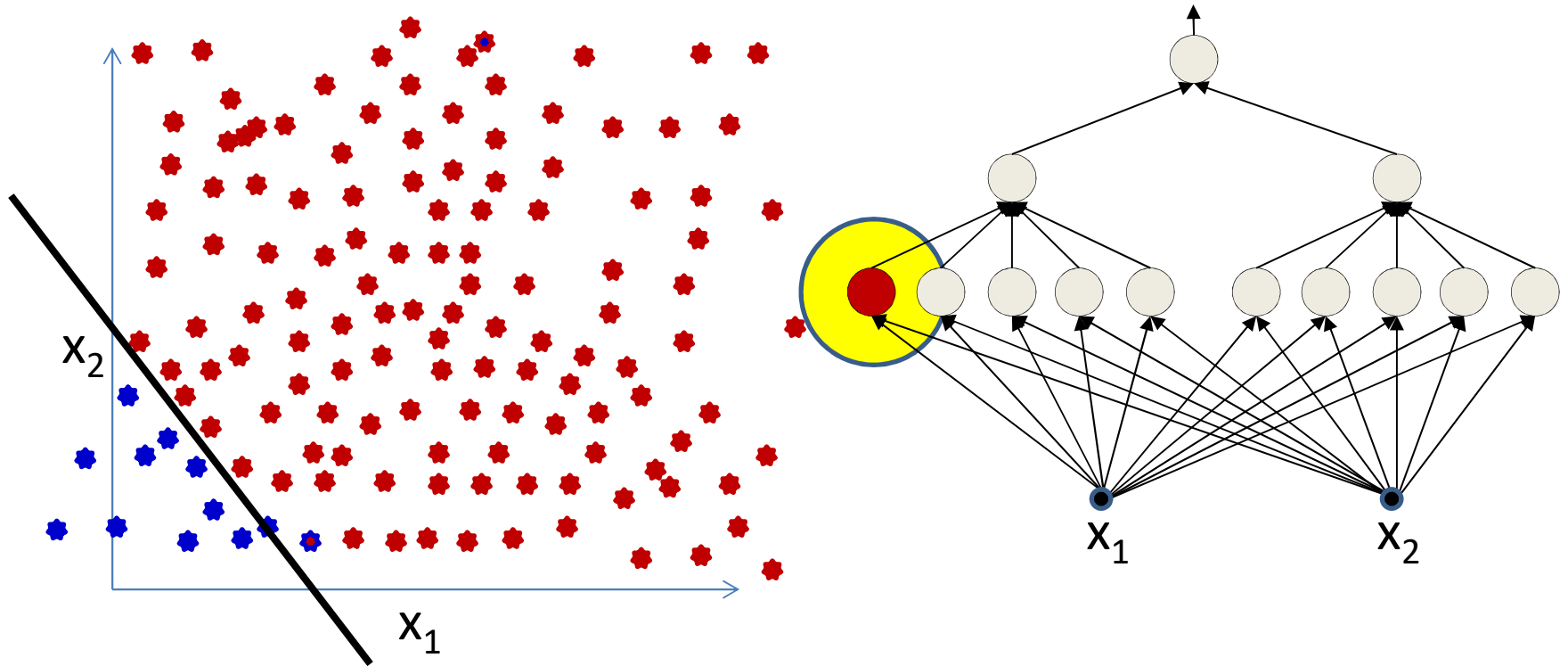
- The lower-level neurons are linear classifiers

The pattern to be learned at the lower level



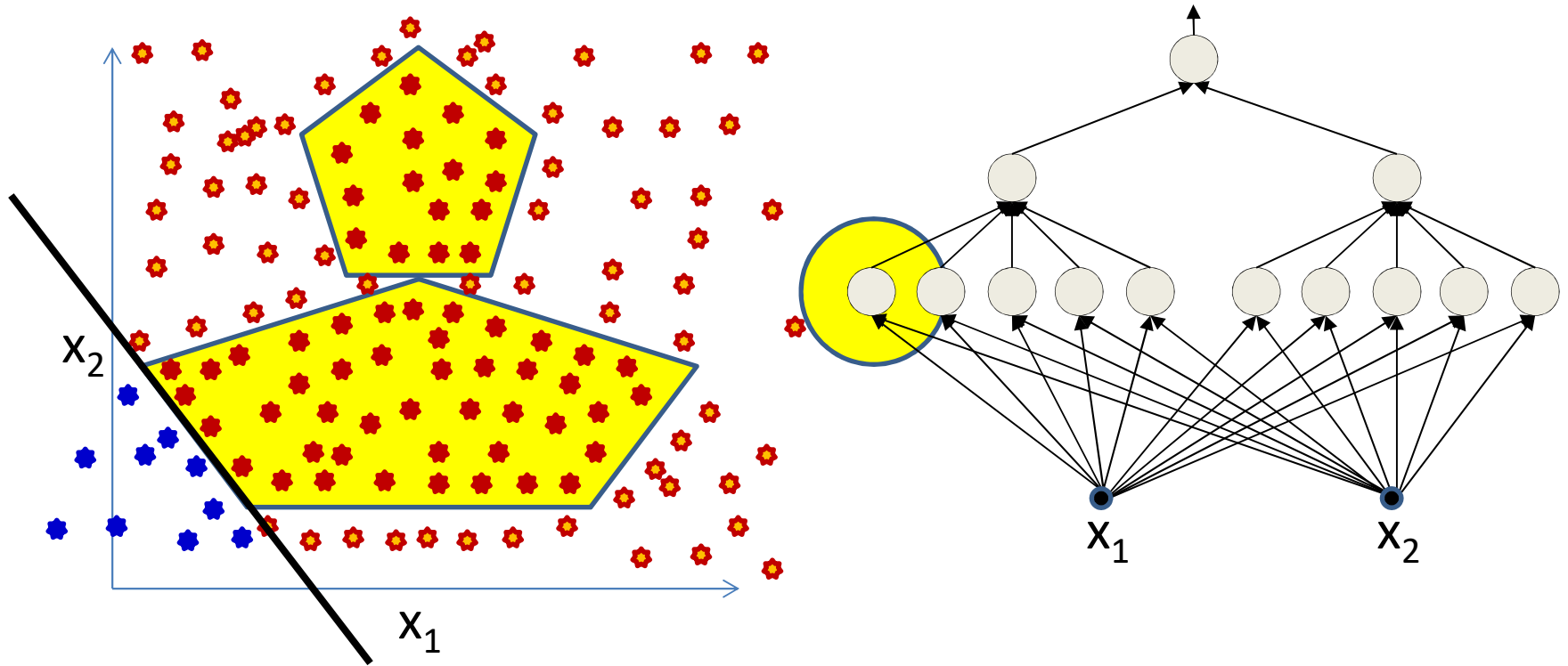
- Consider a single linear classifier that must be learned from the training data
 - Can it be learned from this data?

The pattern to be learned at the lower level



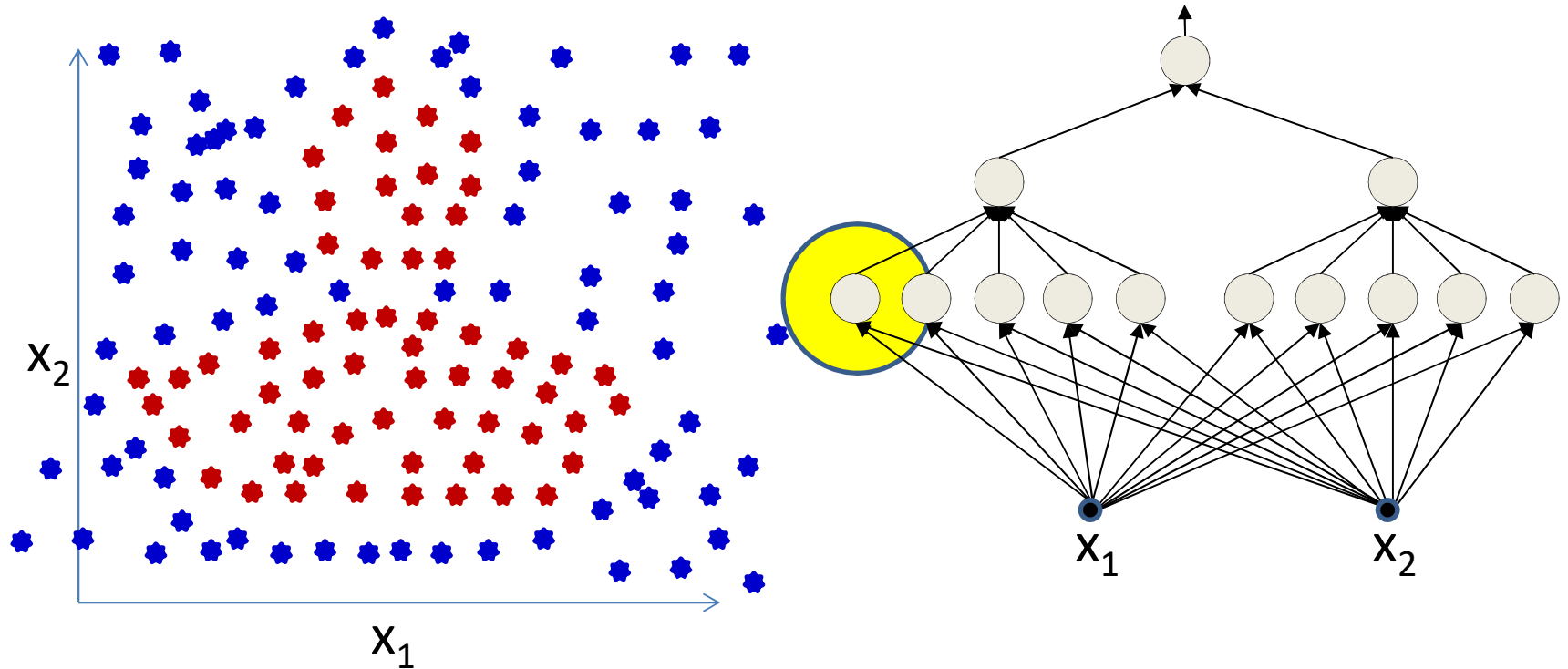
- Consider a single linear classifier that must be learned from the training data
 - Can it be learned from this data?
 - The individual classifier actually requires the kind of labelling shown here
 - Which is *not* given!!

The pattern to be learned at the lower level



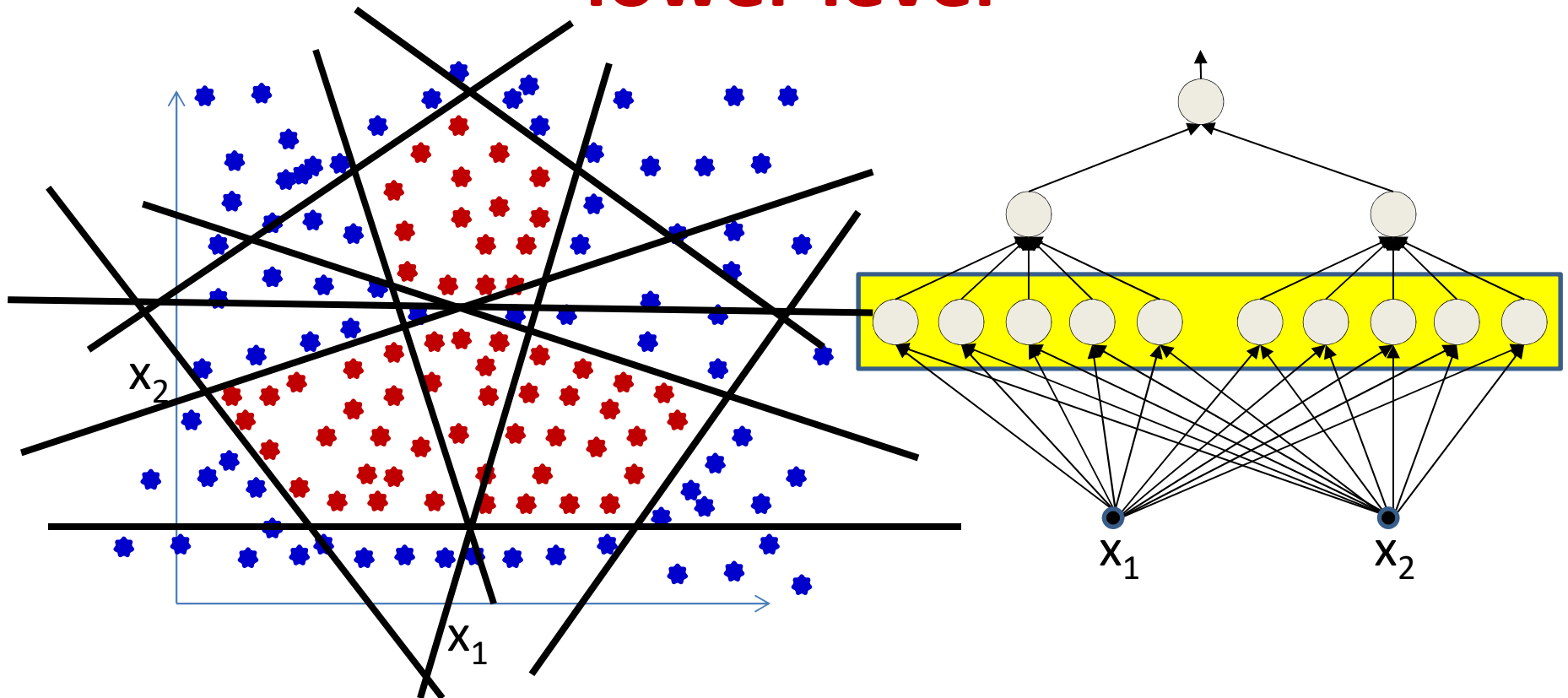
- The lower-level neurons are linear classifiers
 - They require linearly separated labels to be learned
 - The actually provided labels are not linearly separated
 - *Challenge: Must also learn the labels for the lowest units!* ⁵⁷

The pattern to be learned at the lower level



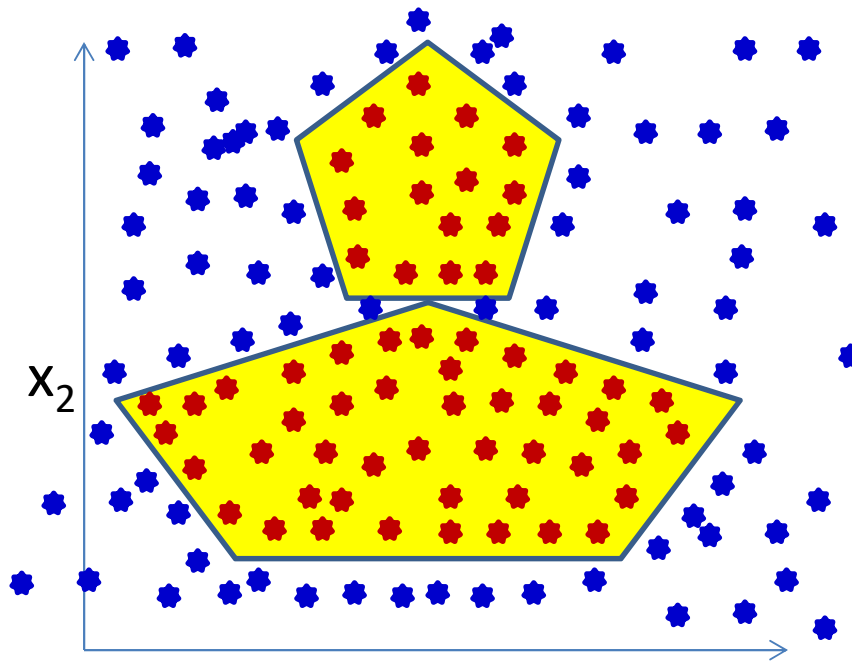
- For a single line:
 - Try out *every possible way of relabeling the blue dots such that we can learn a line that keeps all the red dots on one side!*

The pattern to be learned at the lower level



- This must be done for *each* of the lines (perceptrons)
- Such that, when all of them are combined by the higher-level perceptrons we get the desired pattern
 - Basically an exponential search over inputs

Individual neurons represent one of the lines that compose the figure (linear classifiers)



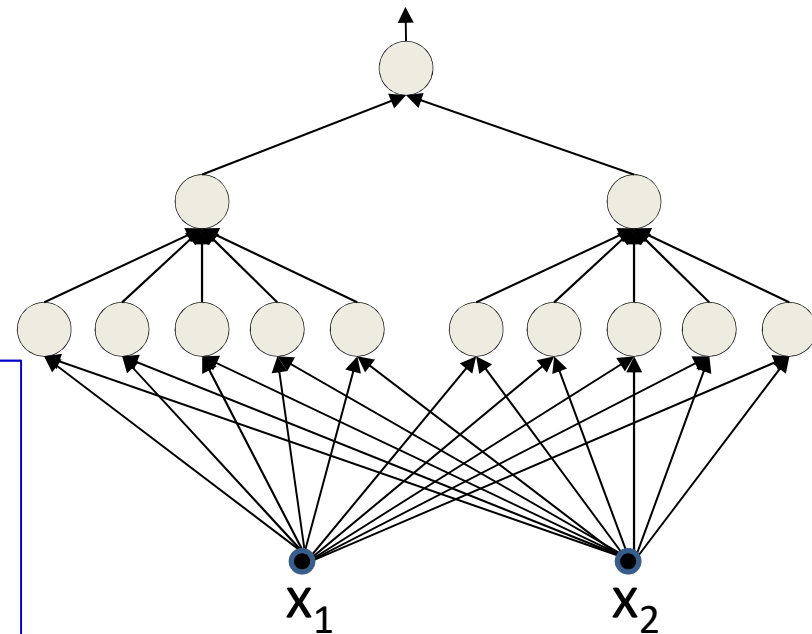
Must know the output of every neuron for *every* training instance, in order to learn this neuron

The outputs should be such that the neuron individually has a linearly separable task

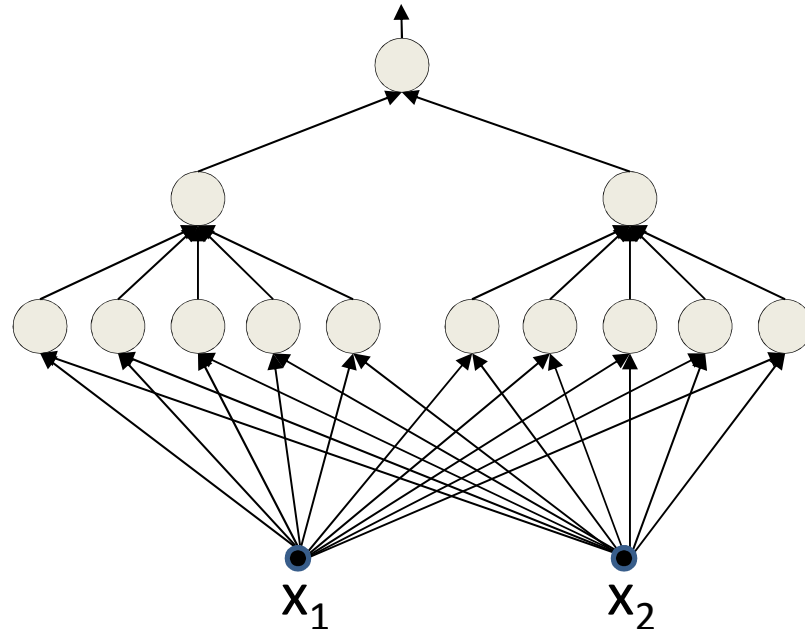
The linear separators must combine to form the desired boundary

This must be done for *every* neuron

Getting any of them wrong will result in incorrect output!



Learning a *multilayer* perceptron



Training data only specifies
input and output of network

Intermediate outputs (outputs
of individual neurons) are not specified

- Training this network using the perceptron rule is a combinatorial optimization problems
- We don't know the outputs of the individual intermediate neurons in the network for any training input
- **Must also determine the correct output for *each* neuron for *every* training instance**
- **NP! Exponential time complexity**

Greedy algorithms: Adaline and Madaline

- The perceptron learning algorithm cannot directly be used to learn an MLP
 - Exponential complexity of assigning intermediate labels
 - Even worse when classes are not actually separable
- Can we use a *greedy* algorithm instead?
 - Adaline / Madaline
 - On slides, will skip in class (check the quiz)

A little bit of History: Widrow



Bernie Widrow

- Scientist, Professor, Entrepreneur
 - Inventor of most useful things in signal processing and machine learning!
-
- First known attempt at an analytical solution to training the perceptron and the MLP
 - Now famous as the LMS algorithm
 - Used everywhere
 - Also known as the “delta rule”

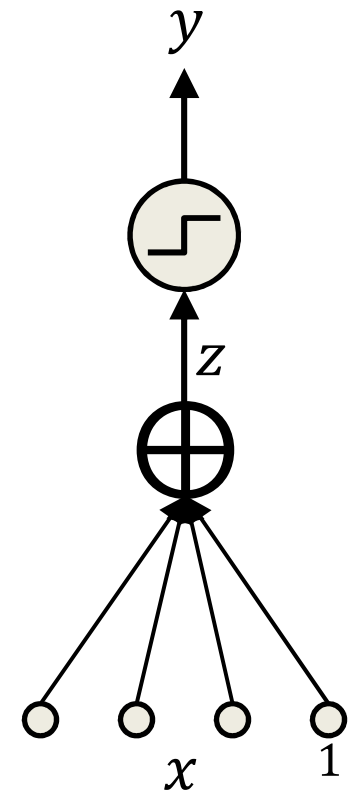
History: ADALINE

$$z = \sum_t w_i x_i$$

Using 1-extended vector notation to account for bias

$$y = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases}$$

- Adaptive *linear* element (Hopf and Widrow, 1960)
- Actually just a regular perceptron
 - Weighted sum on inputs and bias passed through a thresholding function
- ADALINE differs in the *learning rule*



History: Learning in ADALINE

$$z = \sum_t w_i x_i$$

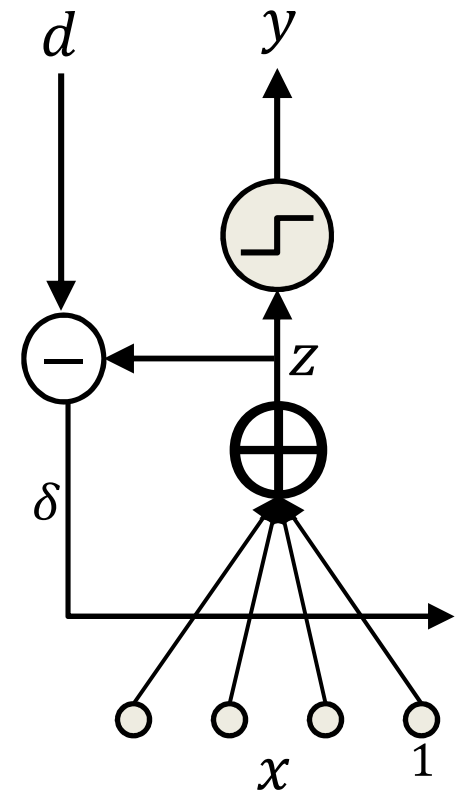
$$out = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases}$$

- During learning, minimize the squared error assuming z to be real output
- The desired output is still binary!

$$Err(x) = \frac{1}{2} (d - z)^2$$

Error for a single input

$$\frac{dErr(x)}{dw_i} = -(d - z)x_i$$



History: Learning in ADALINE

$$z = \sum_t w_i x_i$$

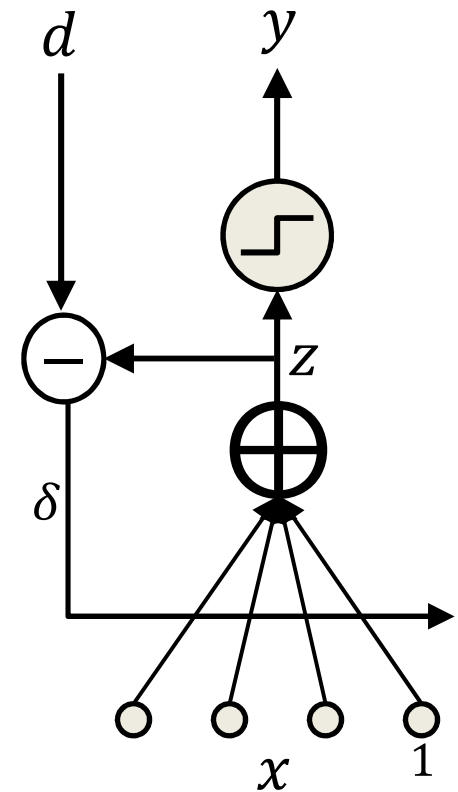
$$Err(x) = \frac{1}{2} (d - z)^2$$

Error for a single input

$$\frac{dErr(x)}{dw_i} = -(d - z)x_i$$

- If we just have a single training input, the *gradient descent* update rule is

$$w_i = w_i + \eta(d - z)x_i$$



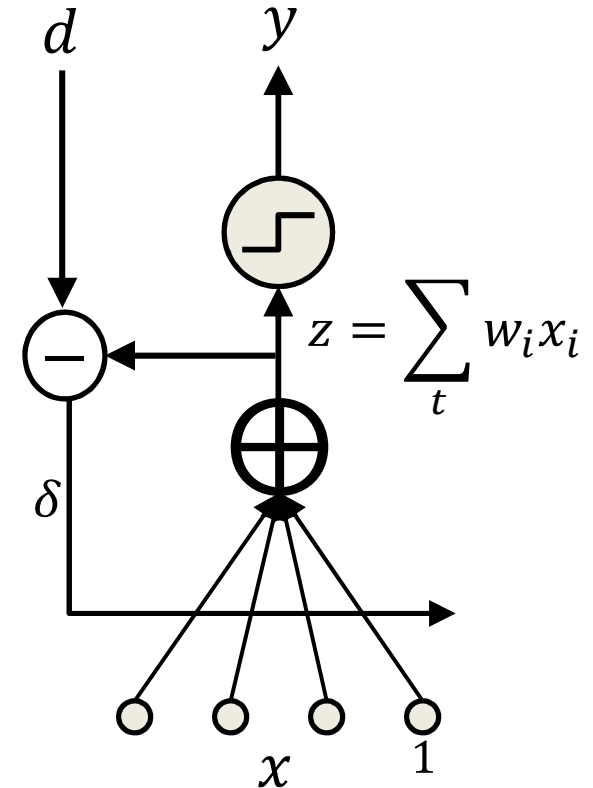
The ADALINE learning rule

- Online learning rule
- **After each input \mathbf{x}** , that has target (binary) output d , compute and update:

$$\delta = d - z$$

$$w_i = w_i + \eta \delta x_i$$

- This is the famous *delta rule*
 - Also called the LMS update rule



The Delta Rule

- In fact both the Perceptron and ADALINE use variants of the delta rule!

- Perceptron: Output used in delta rule is y

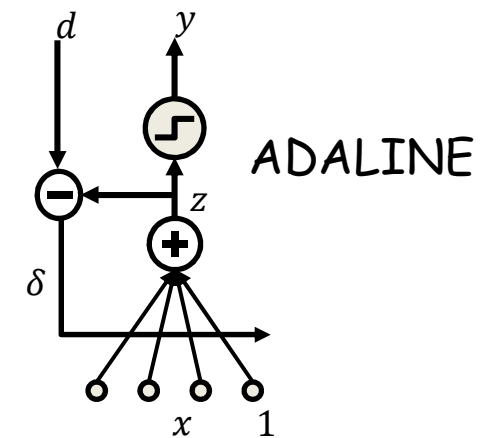
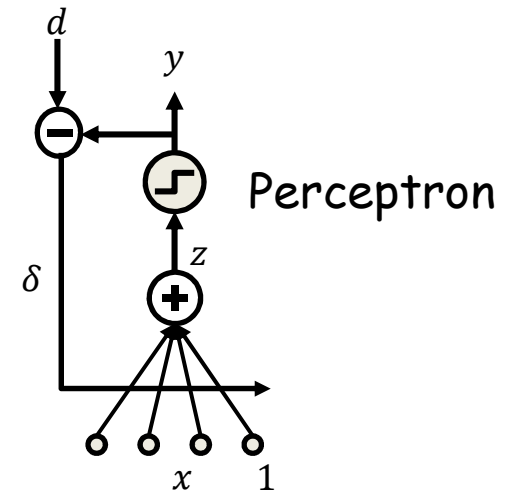
$$\delta = d - y$$

- ADALINE: Output used to estimate weights is z

$$\delta = d - z$$

- For both

$$w_i = w_i + \eta \delta x_i$$



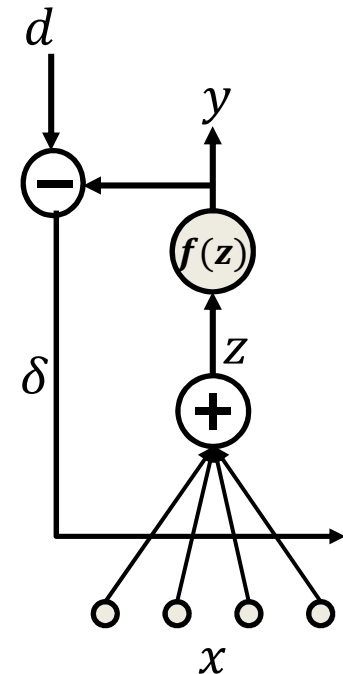
Aside: Generalized delta rule

- For any differentiable activation function the following update rule is used

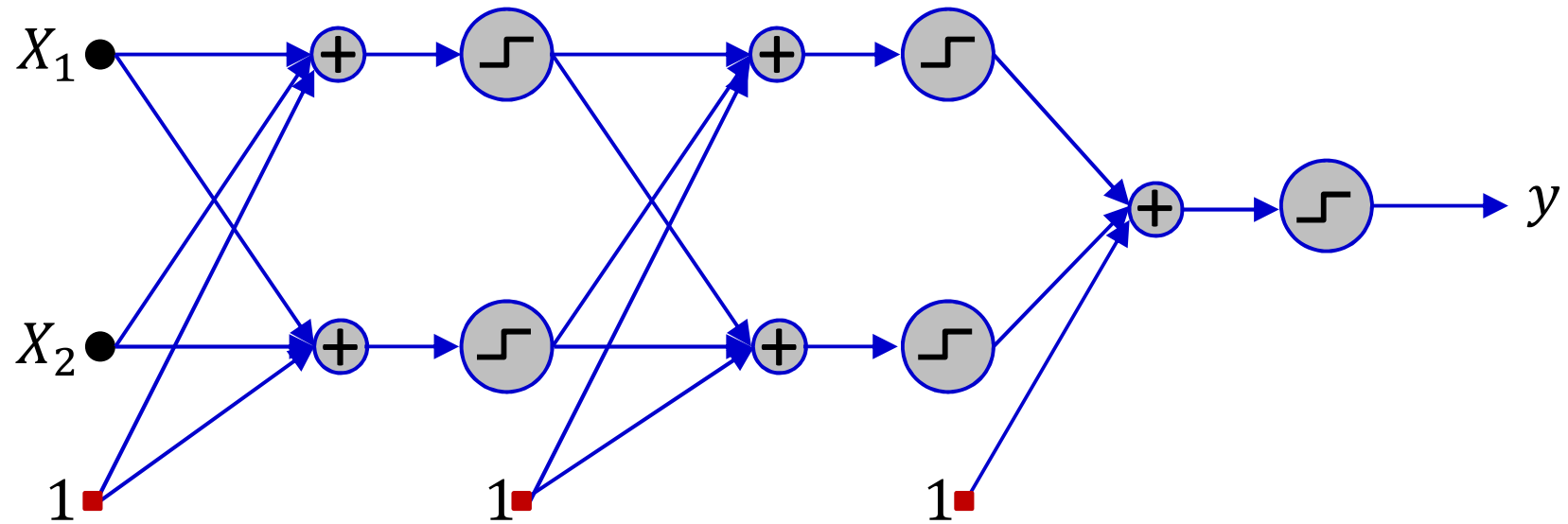
$$\delta = d - y$$

$$w_i = w_i + \eta \delta f'(z) x_i$$

- This is the famous Widrow-Hoff update rule
 - Lookahead: Note that this is *exactly* backpropagation in multilayer nets if we let $f(z)$ represent the entire network between z and y
- It is possibly the most-used update rule in machine learning and signal processing
 - Variants of it appear in almost every problem

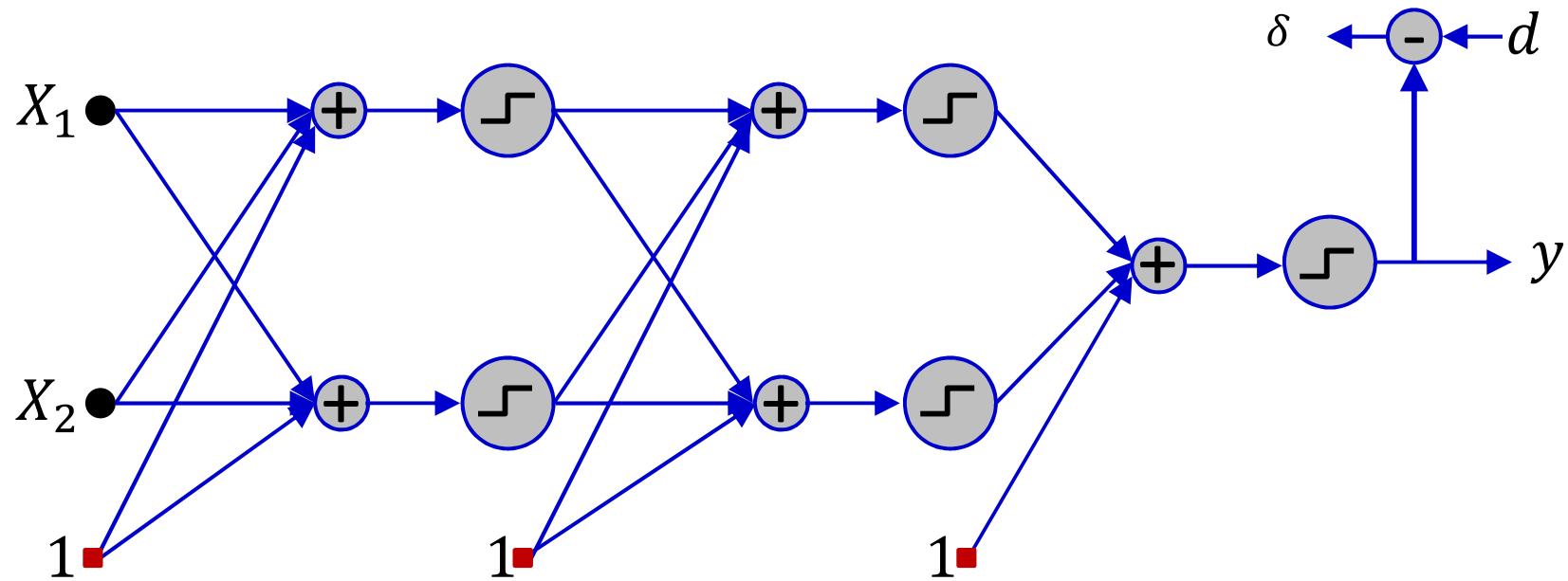


Multilayer perceptron: MADALINE



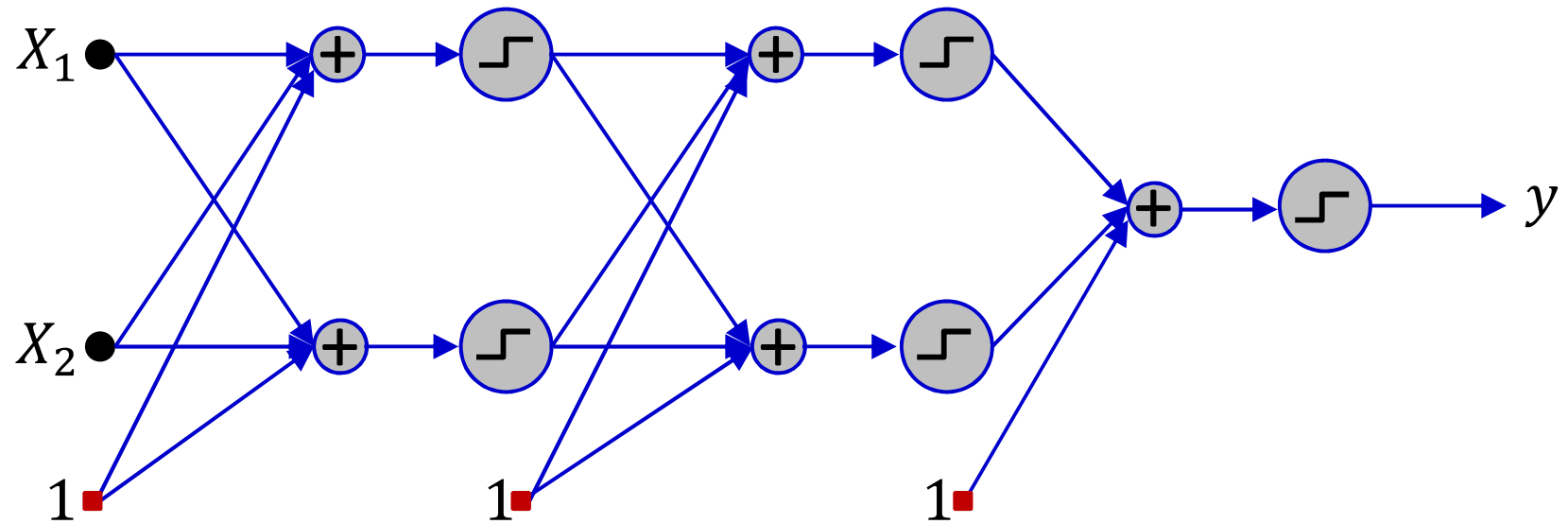
- *Multiple Adaline*
 - A multilayer perceptron with threshold activations
 - The MADALINE

MADALINE Training



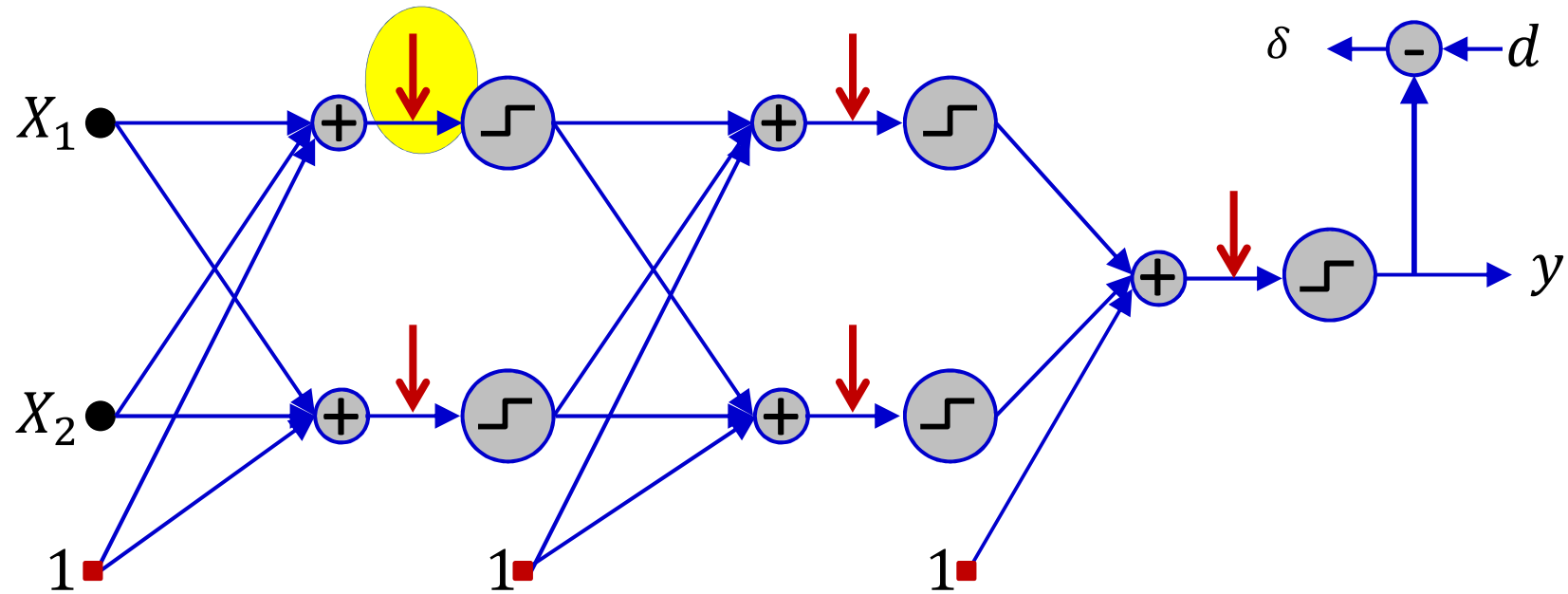
- *Update only on error*
 - $\delta \neq 0$
 - On inputs for which output and target values differ

MADALINE Training



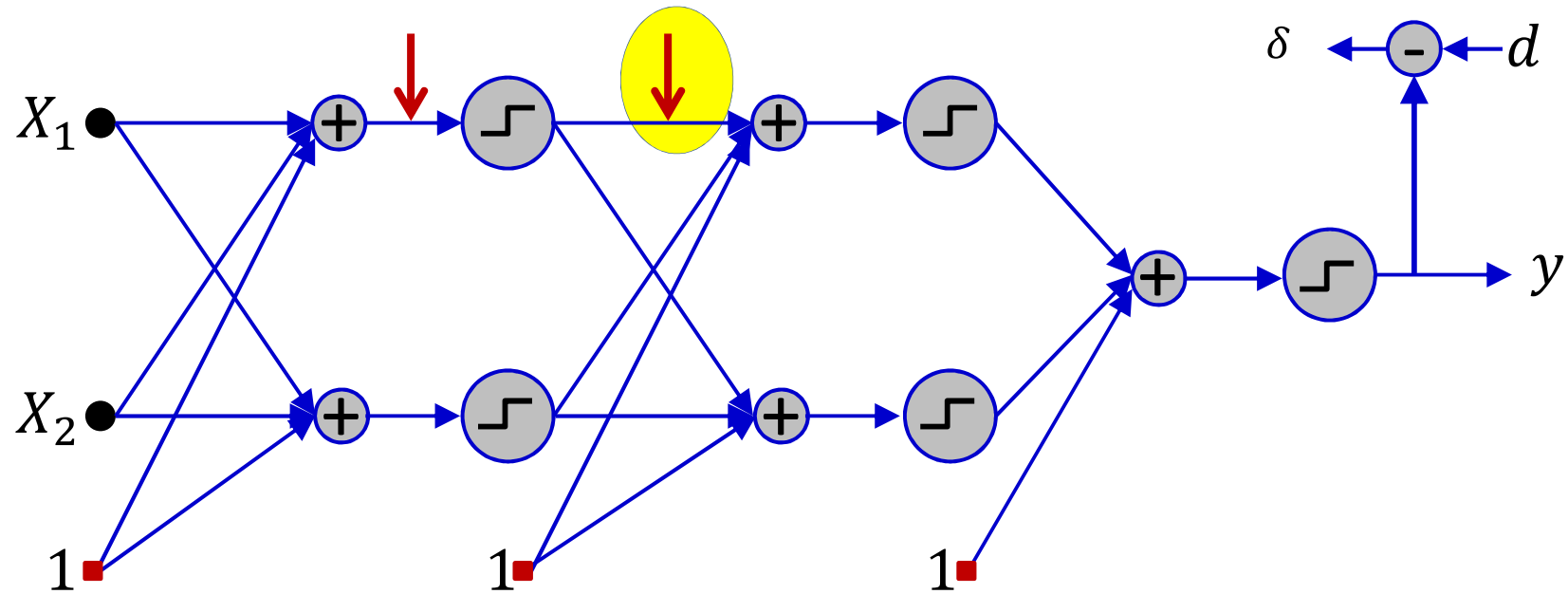
- While stopping criterion not met do:
 - Classify an input

MADALINE Training



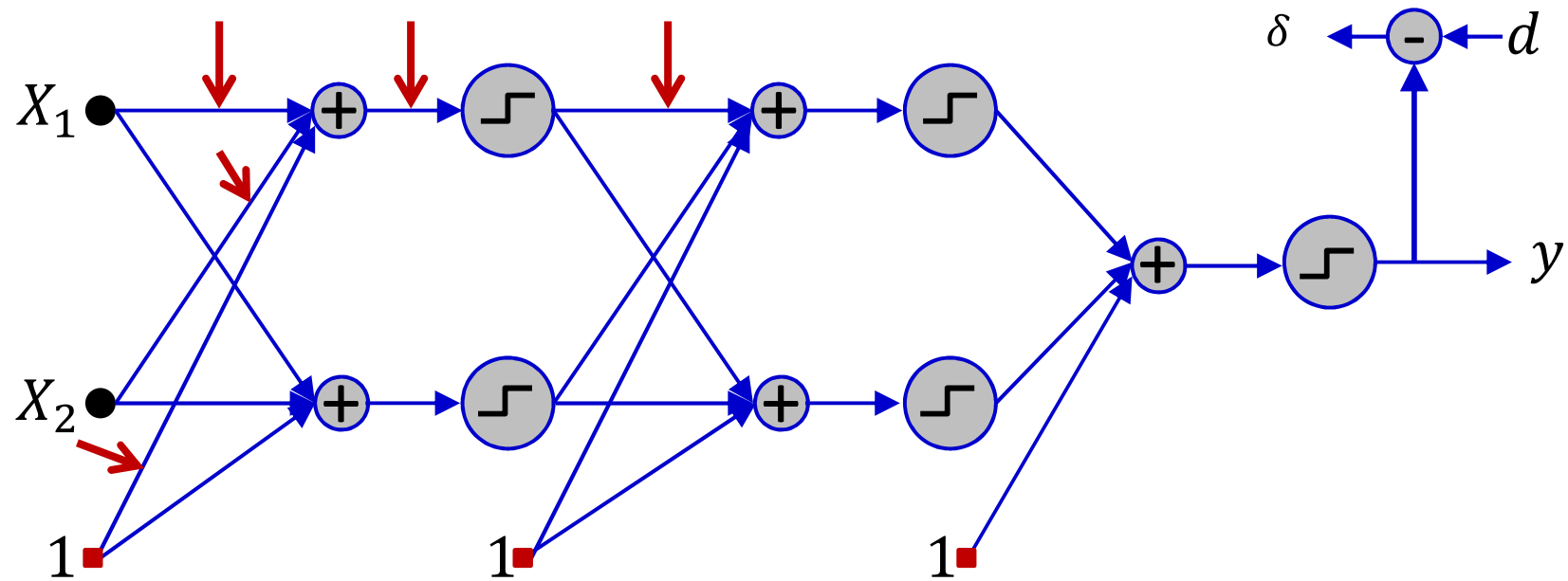
- While stopping criterion not met do:
 - Classify an input
 - If error, find the z that is closest to 0

MADALINE Training



- While stopping criterion not met do:
 - Classify an input
 - If error, find the z that is closest to 0
 - Flip the output of corresponding unit and compute new output

MADALINE Training



- While stopping criterion not met do:
 - Classify an input
 - If error, find the z that is closest to 0
 - Flip the output of corresponding unit and compute new output
 - If error reduces:
 - Set the desired output of the unit to the flipped value
 - Apply ADALINE rule to update weights of the unit

MADALINE

- Greedy algorithm, effective for small networks
- Not very useful for large nets
 - Too expensive
 - Too greedy

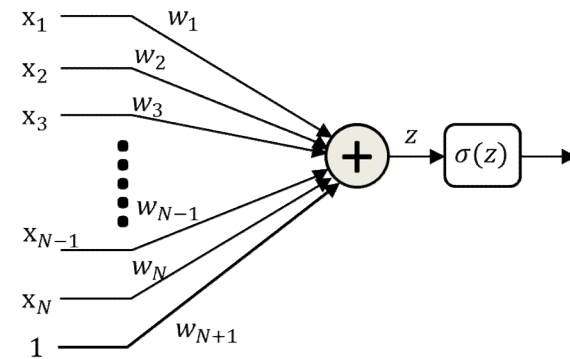
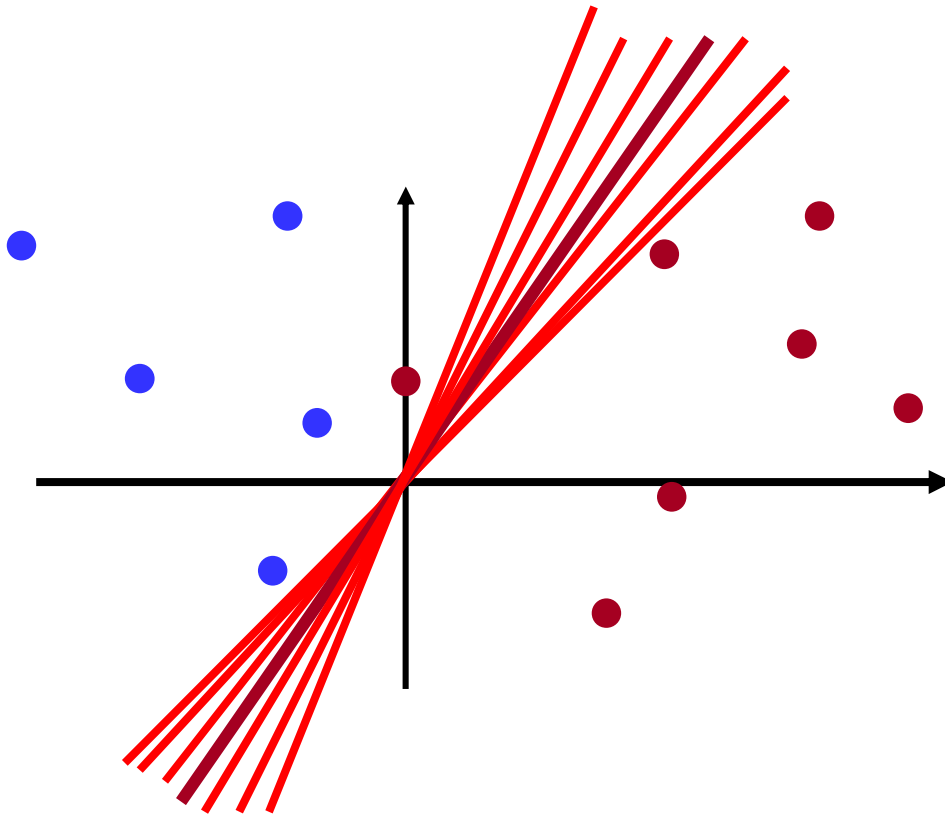
Story so far

- “Learning” a network = learning the weights and biases to compute a target function
 - Will require a network with sufficient “capacity”
- In practice, we learn networks by “fitting” them to match the input-output relation of “training” instances drawn from the target function
- A linear decision boundary can be learned by a single perceptron (with a threshold-function activation) in linear time if classes are linearly separable
- Non-linear decision boundaries require networks of perceptrons
- Training an MLP with threshold-function activation perceptrons will require knowledge of the input-output relation for every training instance, for *every* perceptron in the network
 - These must be determined as part of training
 - For threshold activations, this is an NP-complete combinatorial optimization problem

History..

- The realization that training an entire MLP was a combinatorial optimization problem stalled development of neural networks for well over a decade!

Why this problem?

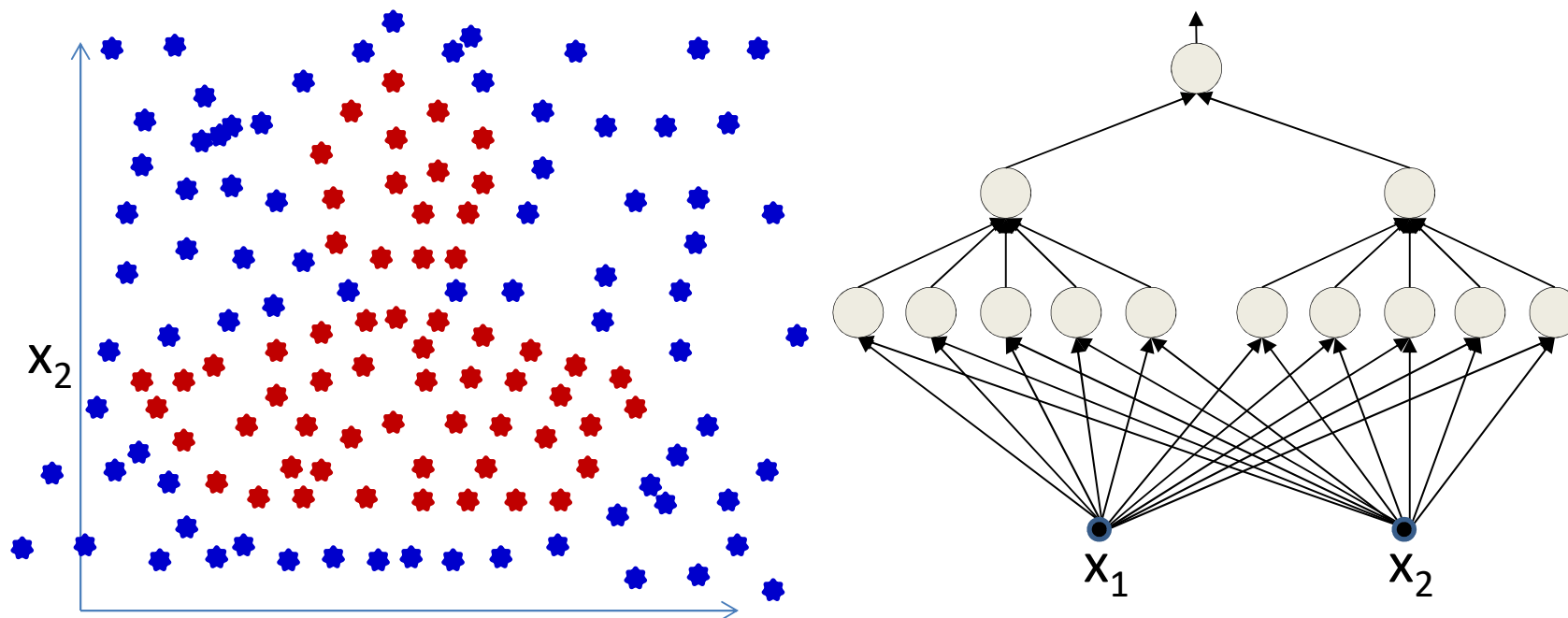


$$\sigma(z) =$$



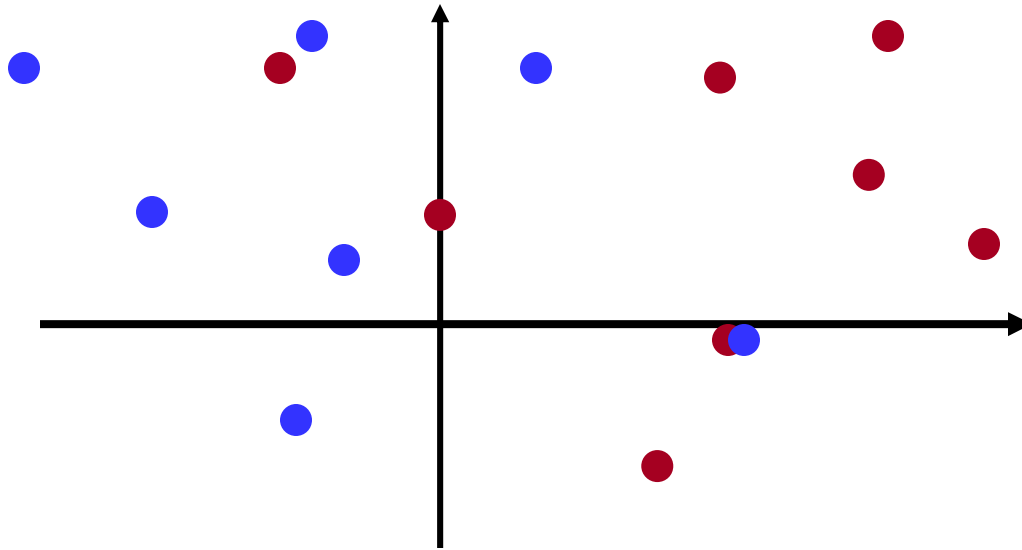
- The perceptron is a flat function with zero derivative everywhere, except at 0 where it is non-differentiable
 - You can vary the weights a *lot* without changing the error
 - There is no indication of which direction to change the weights to reduce error

This only compounds on larger problems



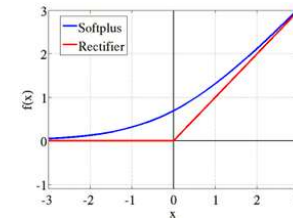
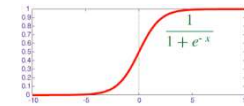
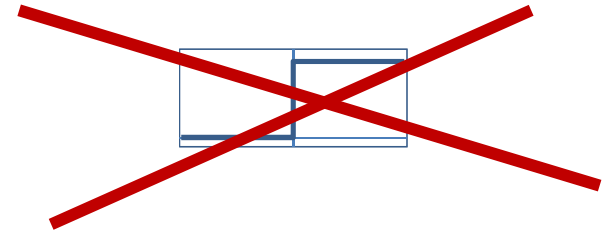
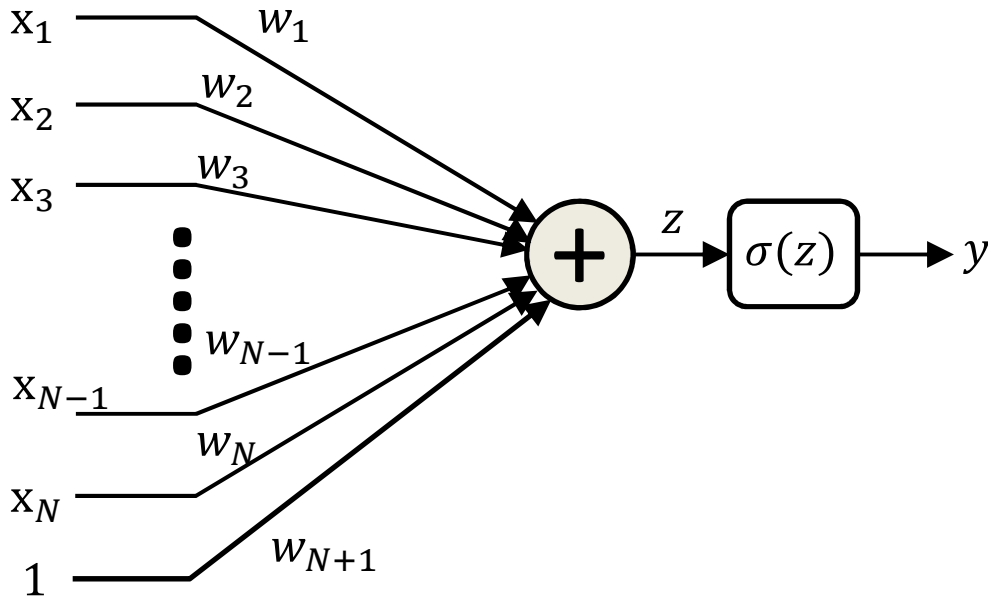
- Individual neurons' weights can change significantly without changing overall error
- The simple MLP is a flat, non-differentiable function
 - Actually a function with 0 derivative nearly everywhere, and no derivatives at the boundaries

A second problem: What we *actually* model



- Real-life data are rarely clean
 - Not linearly separable
 - Rosenblatt's perceptron wouldn't work in the first place

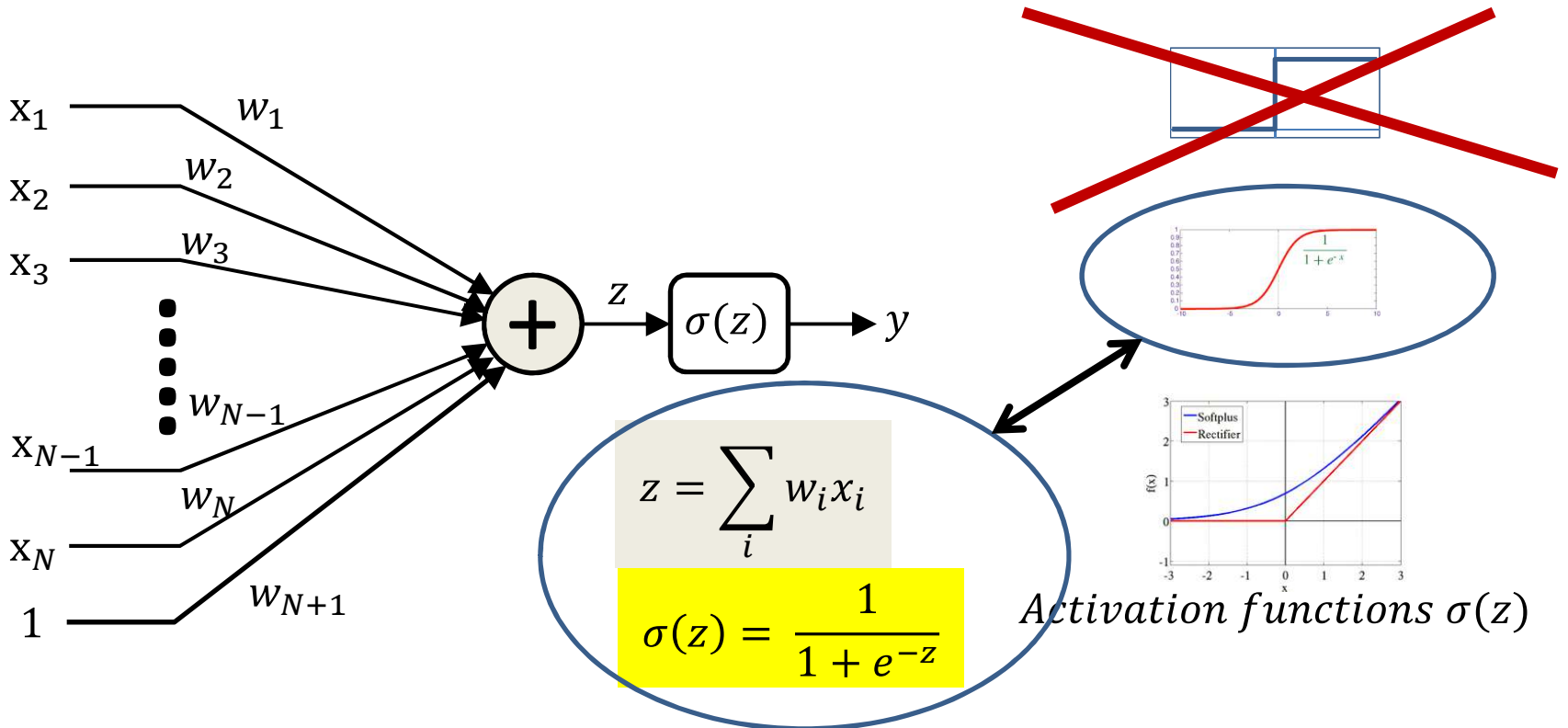
Solution



Activation functions $\sigma(z)$

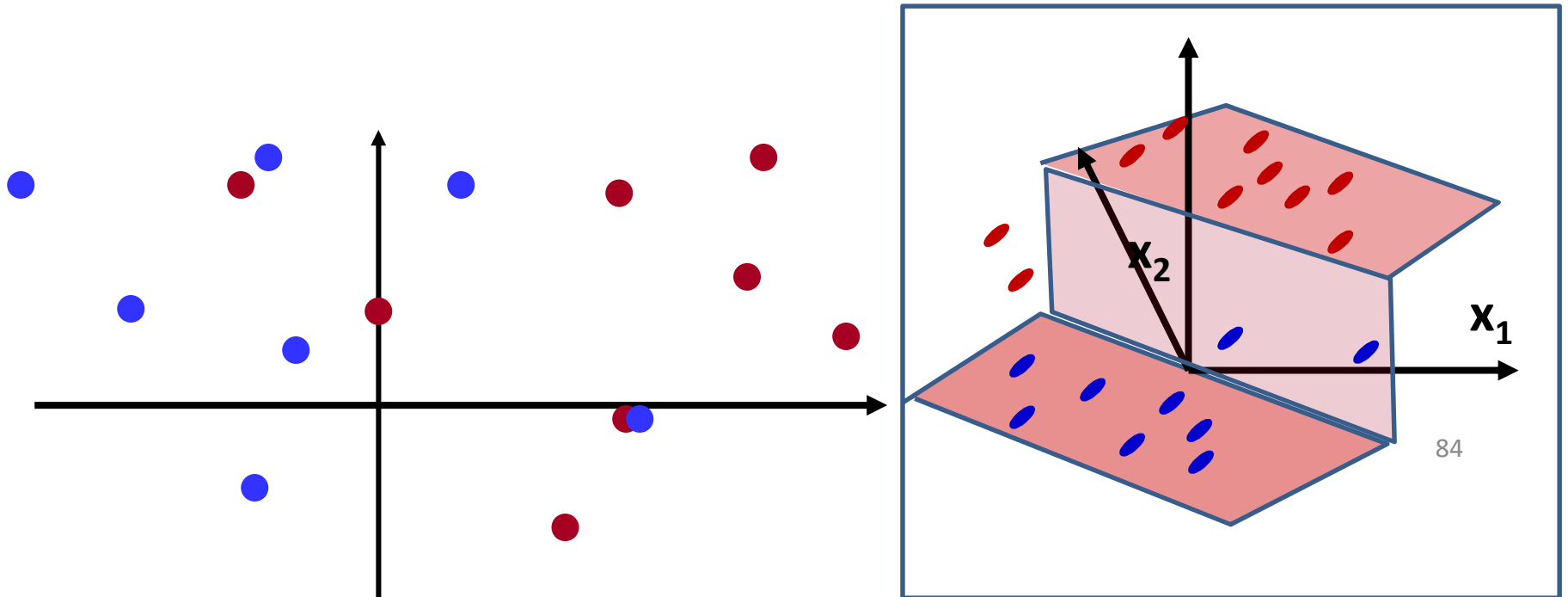
- Lets make the neuron differentiable, *with non-zero derivatives over much of the input space*
 - Small changes in weight can result in non-negligible changes in output
 - This enables us to estimate the parameters using gradient descent techniques..

Differentiable Activations: An aside



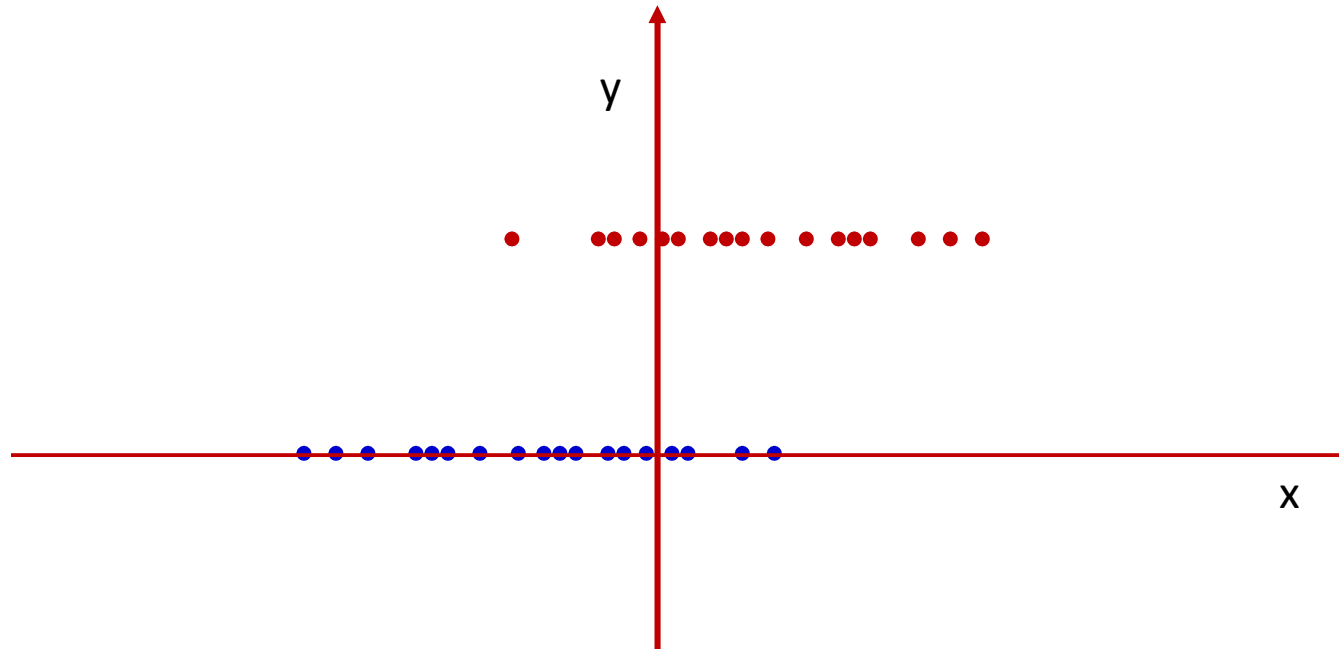
- This particular one has a nice interpretation

Non-linearly separable data



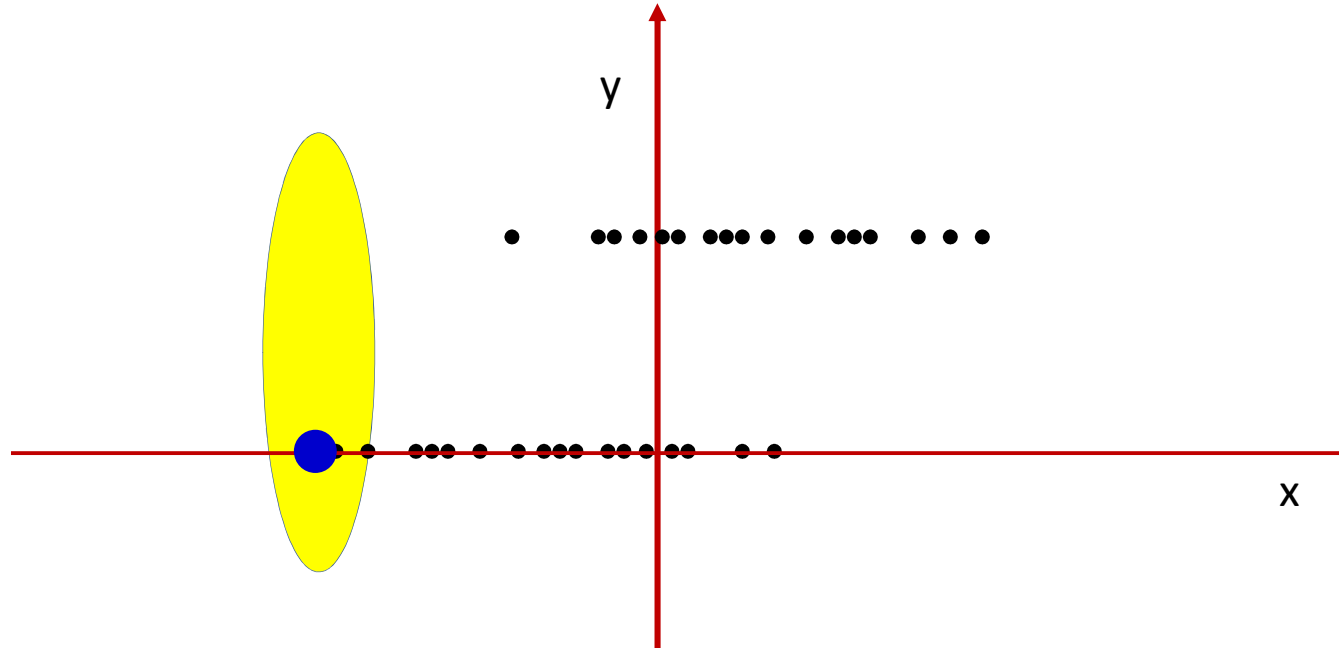
- Two-dimensional example
 - Blue dots (on the floor) on the “red” side
 - Red dots (suspended at $Y=1$) on the “blue” side
 - No line will cleanly separate the two colors

Non-linearly separable data: 1-D example



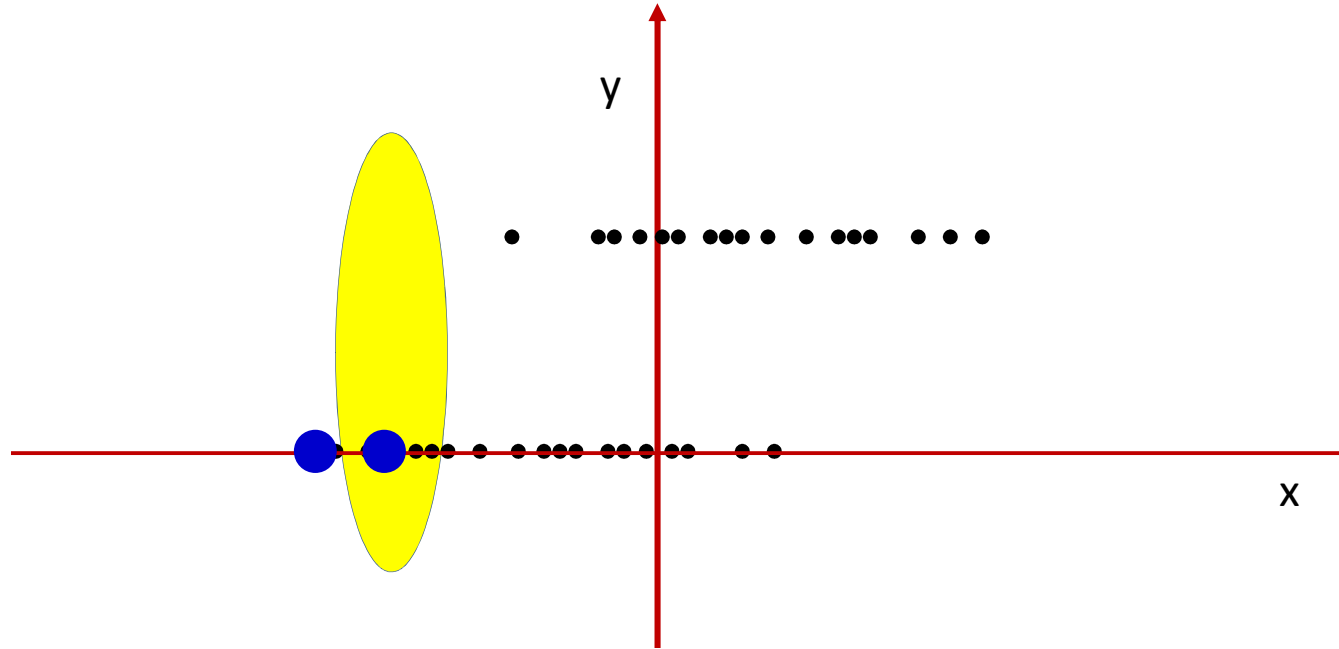
- One-dimensional example for visualization
 - All (red) dots at $Y=1$ represent instances of class $Y=1$
 - All (blue) dots at $Y=0$ are from class $Y=0$
 - The data are not linearly separable
 - In this 1-D example, a linear separator is a threshold
 - No threshold will cleanly separate red and blue dots

The *probability* of $y=1$



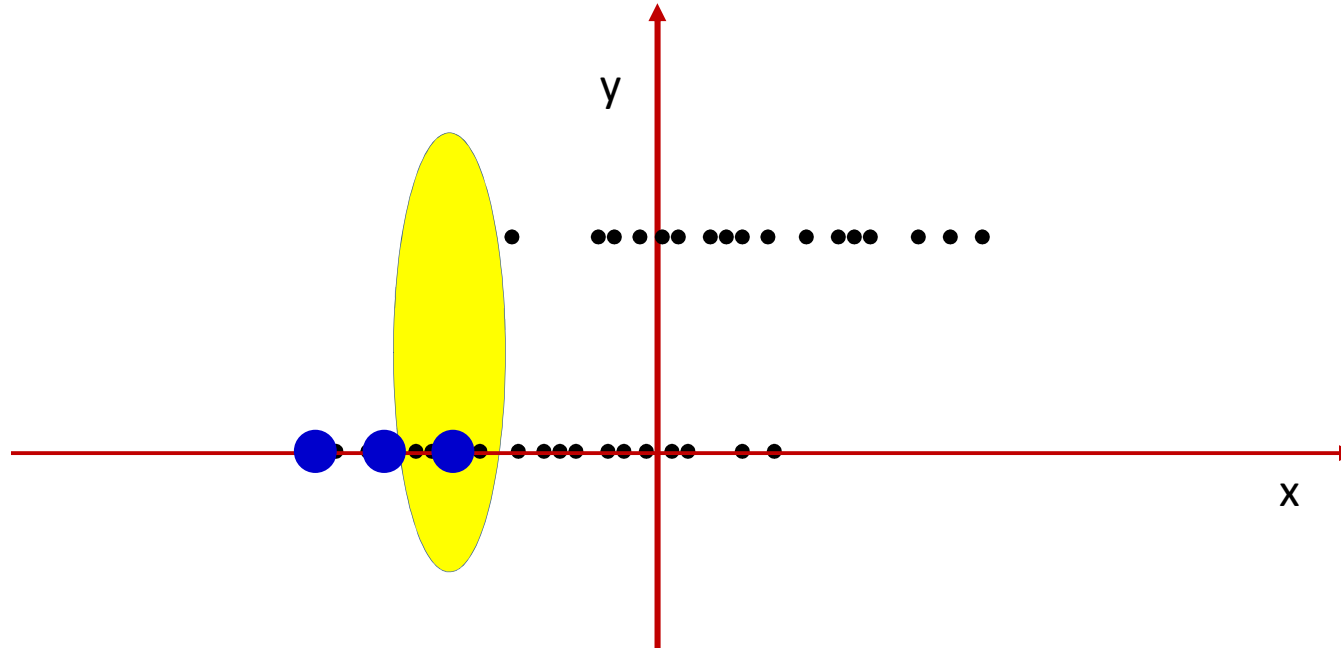
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of $Y=1$ at that point

The *probability* of $y=1$



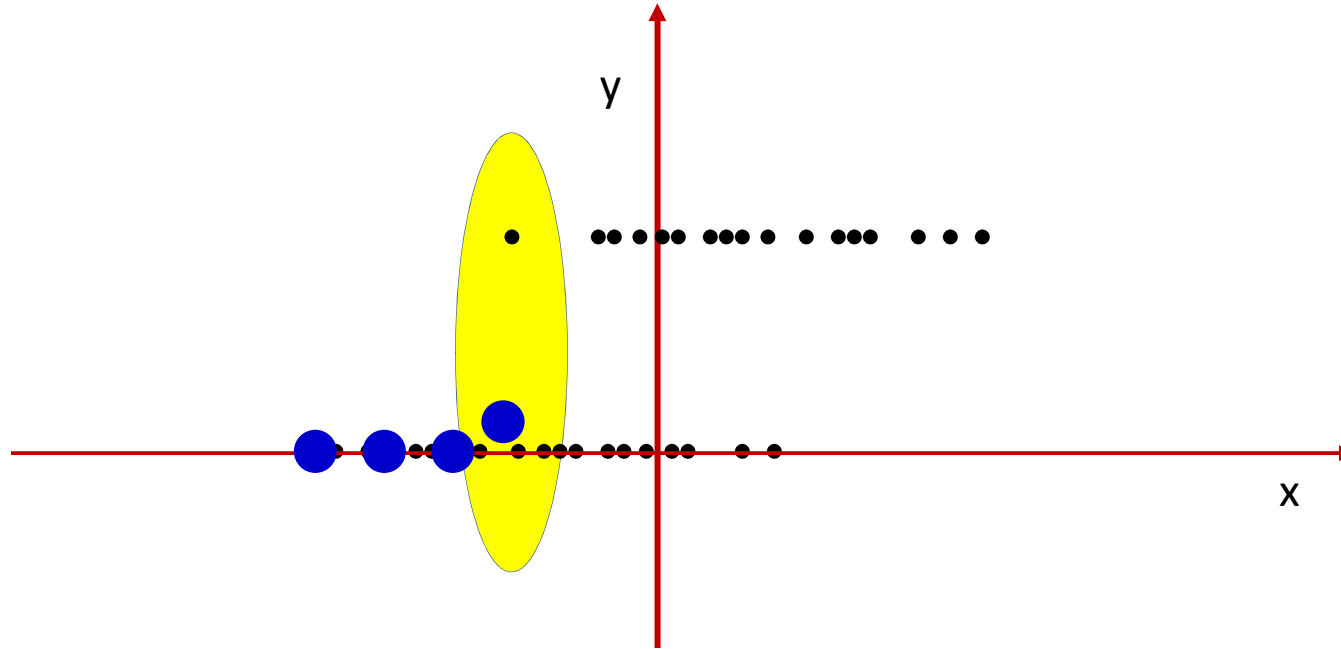
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



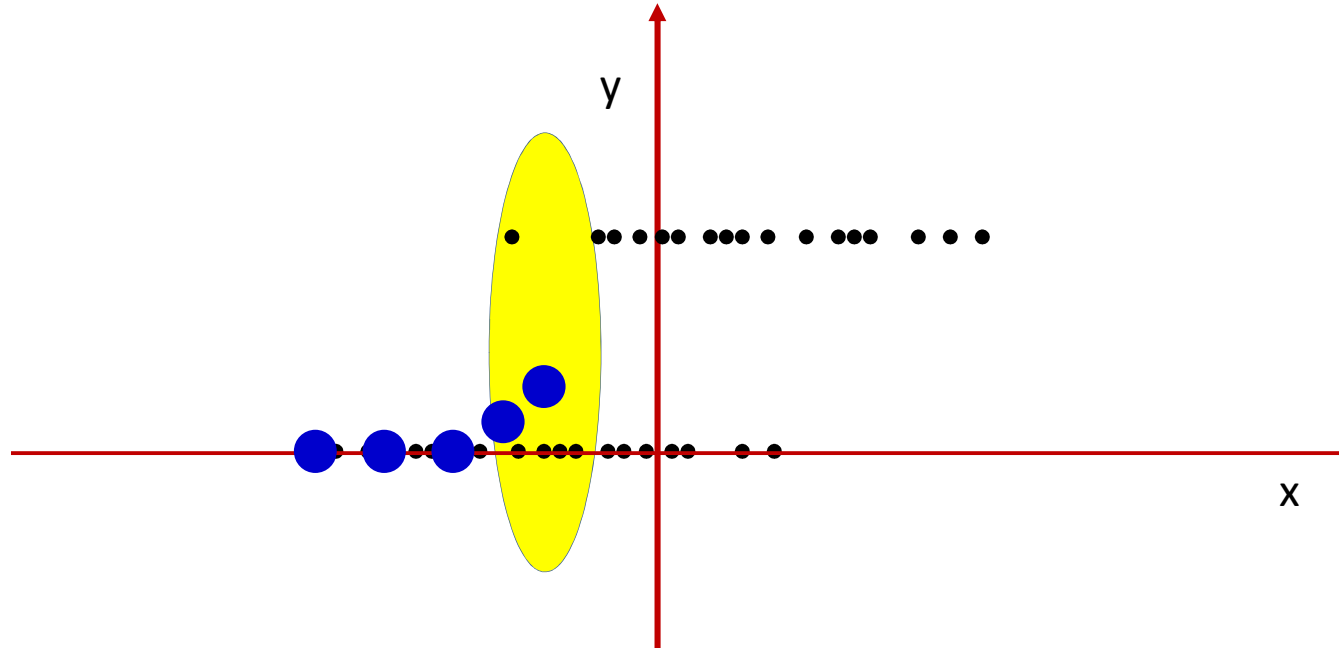
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



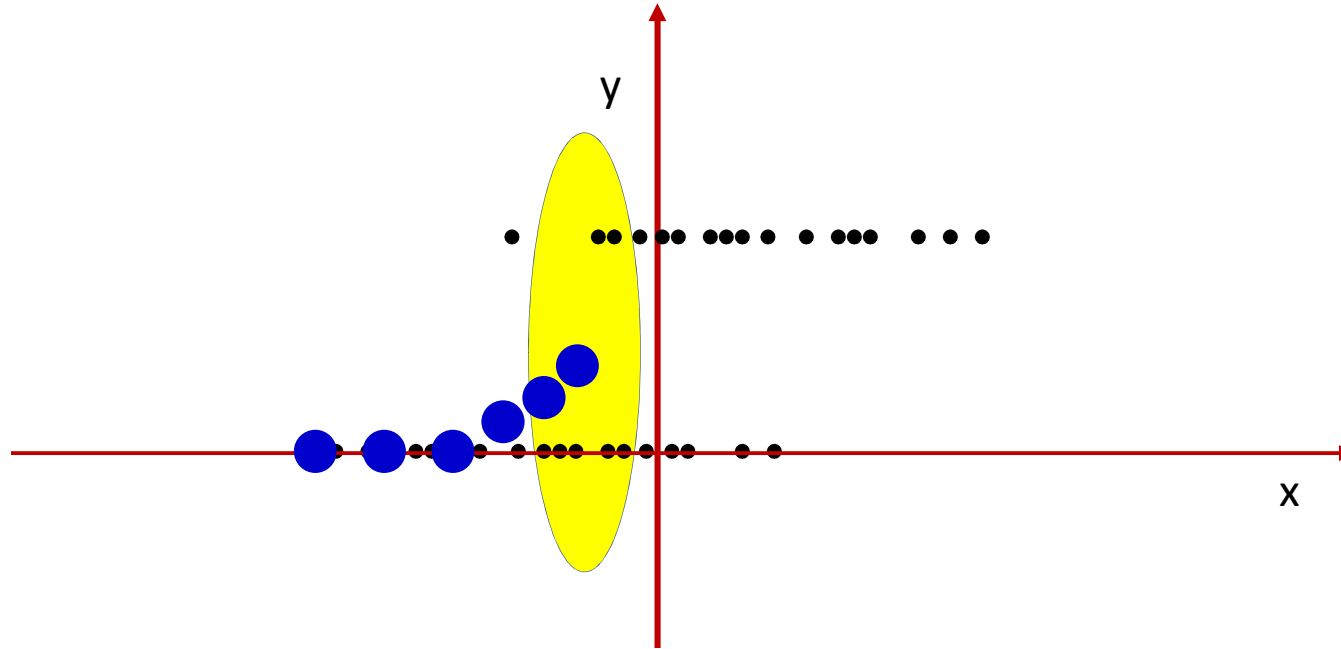
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



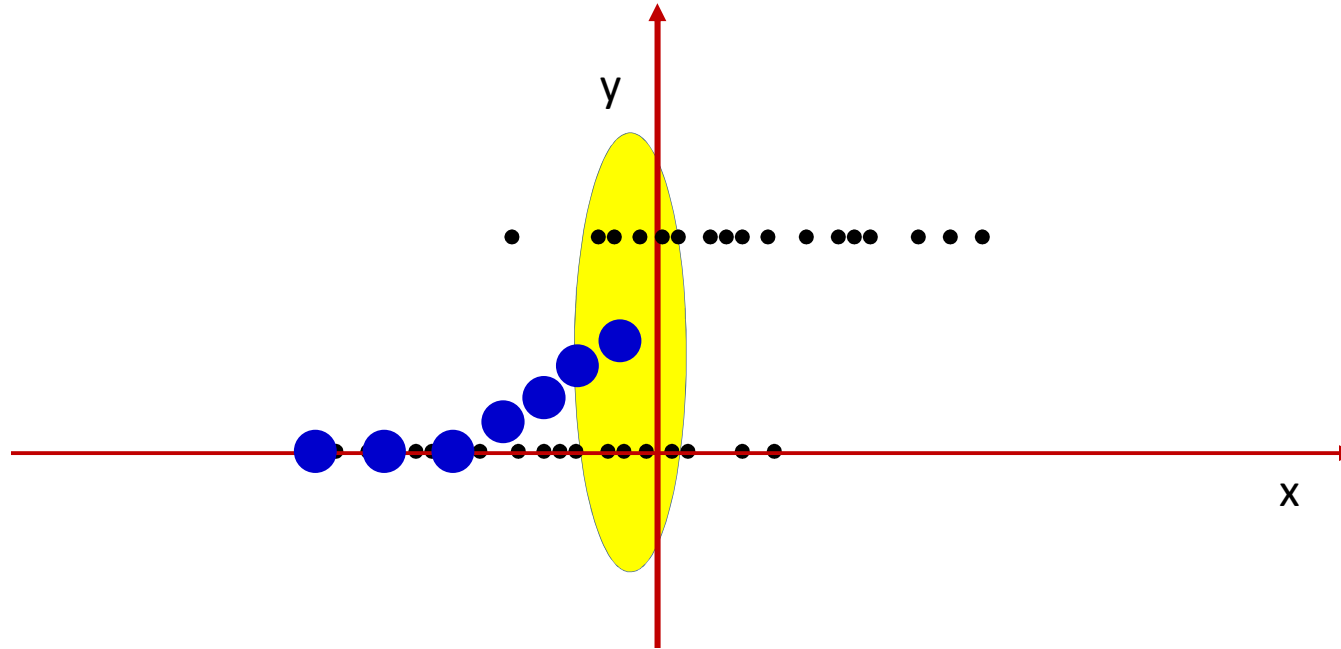
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



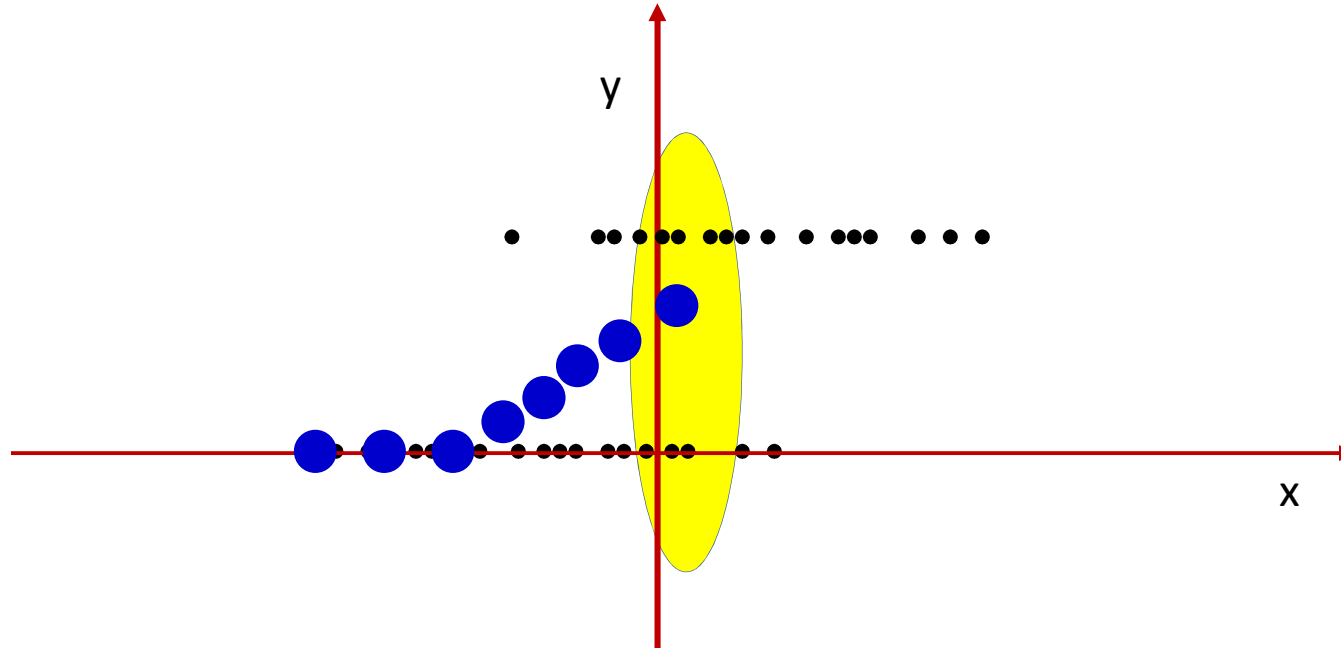
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



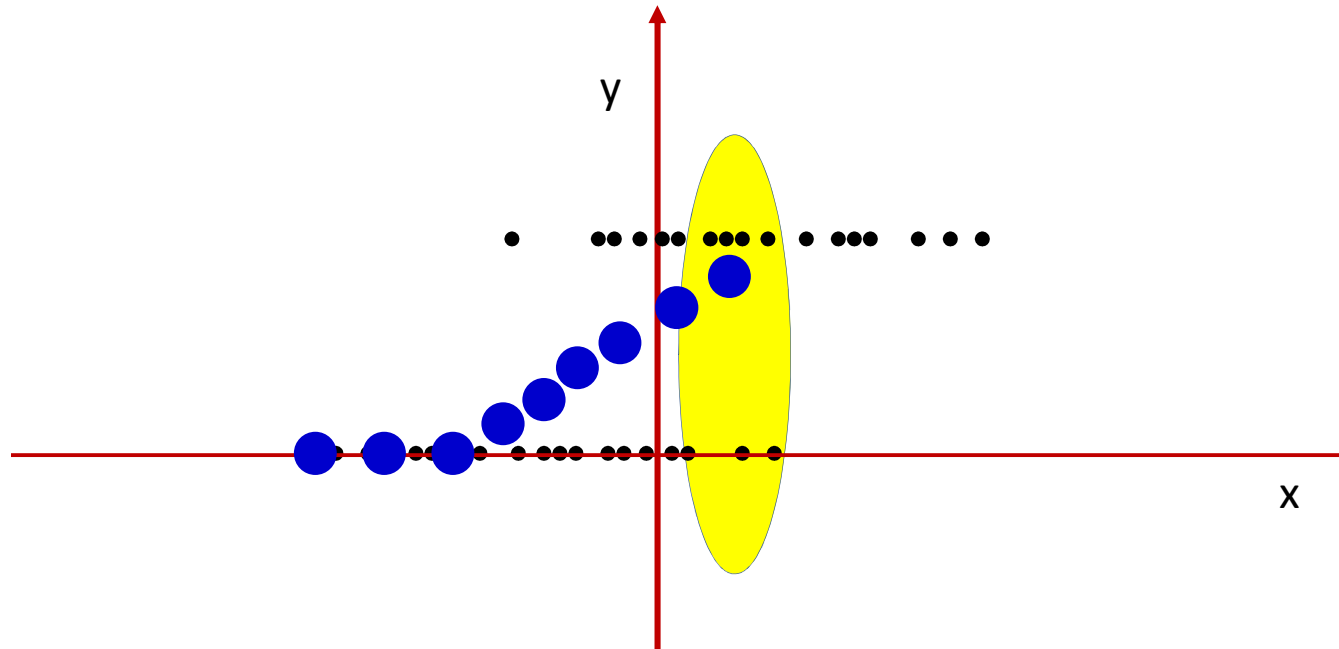
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



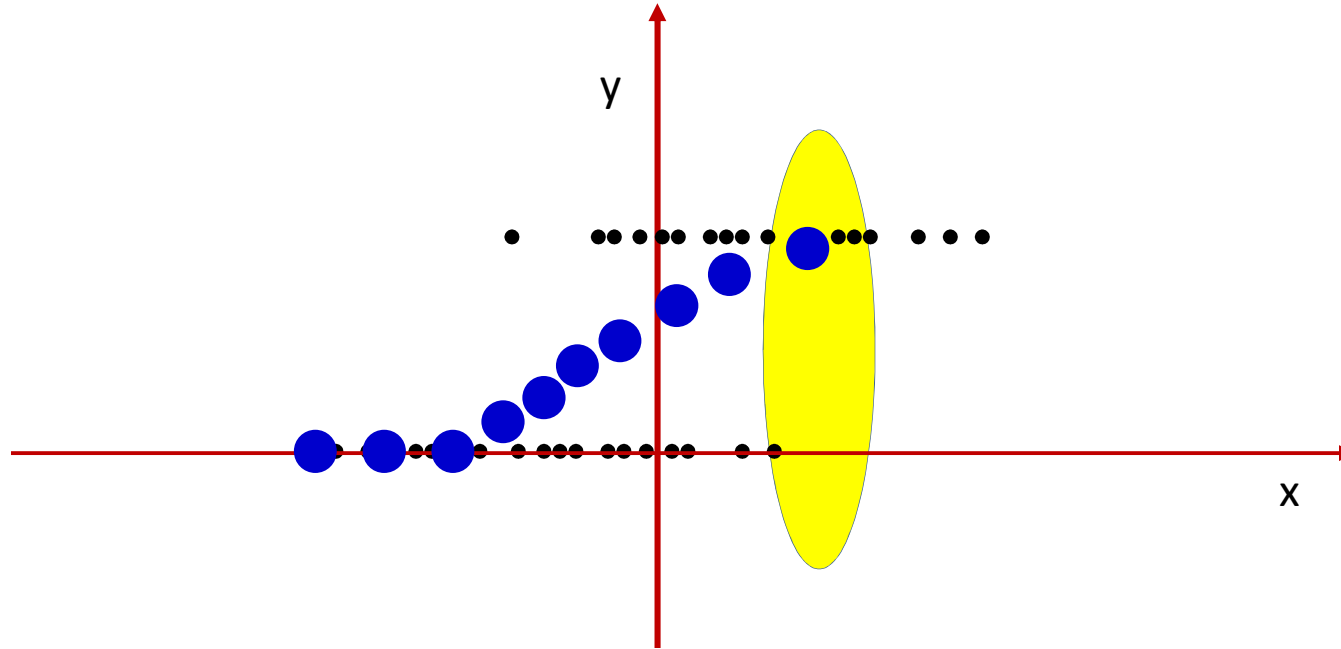
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



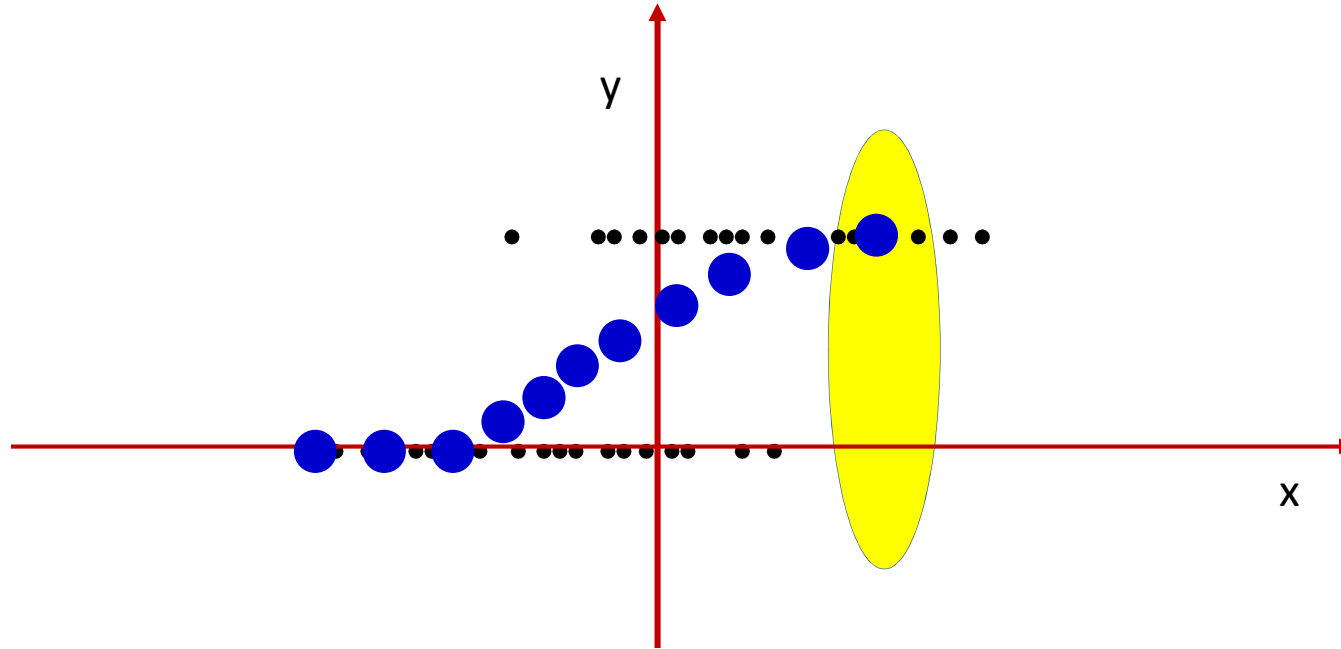
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



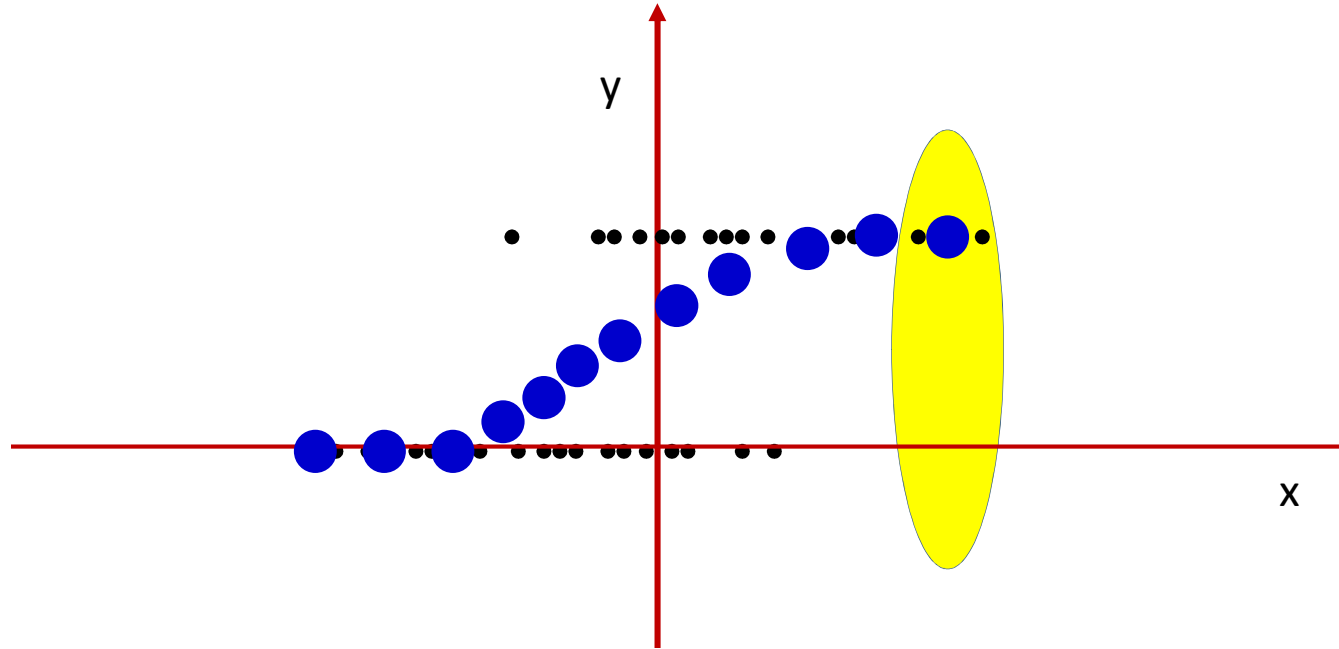
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



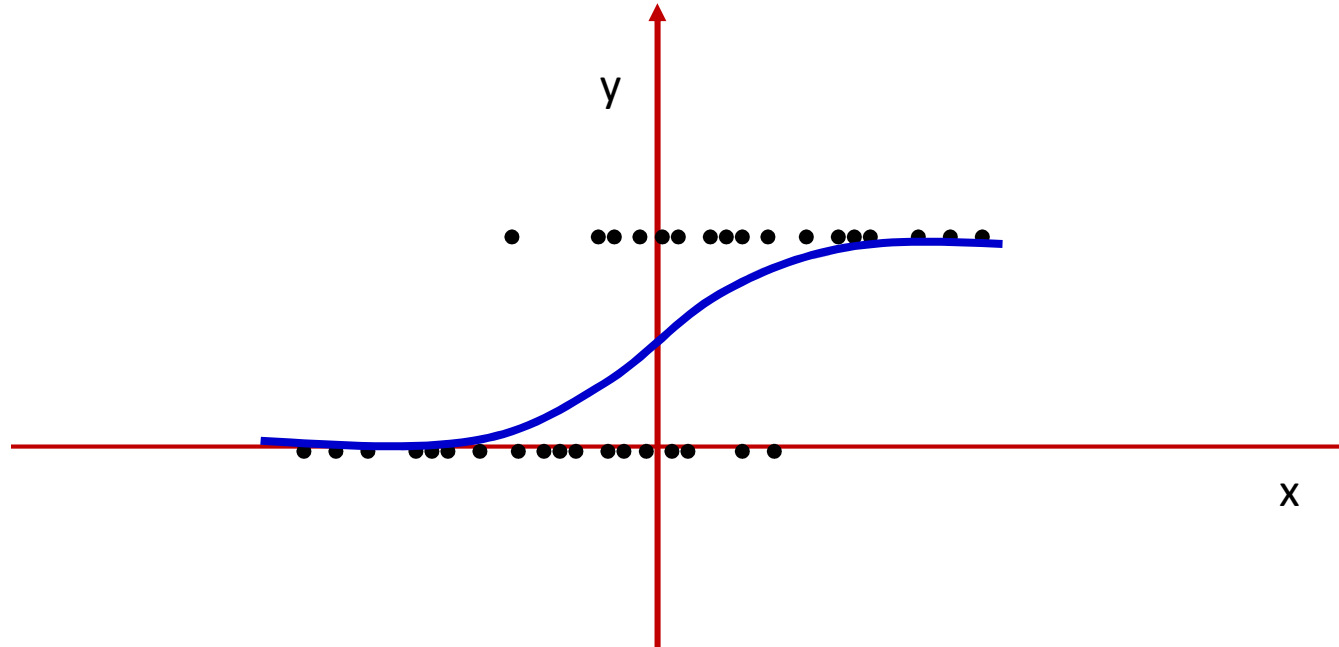
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



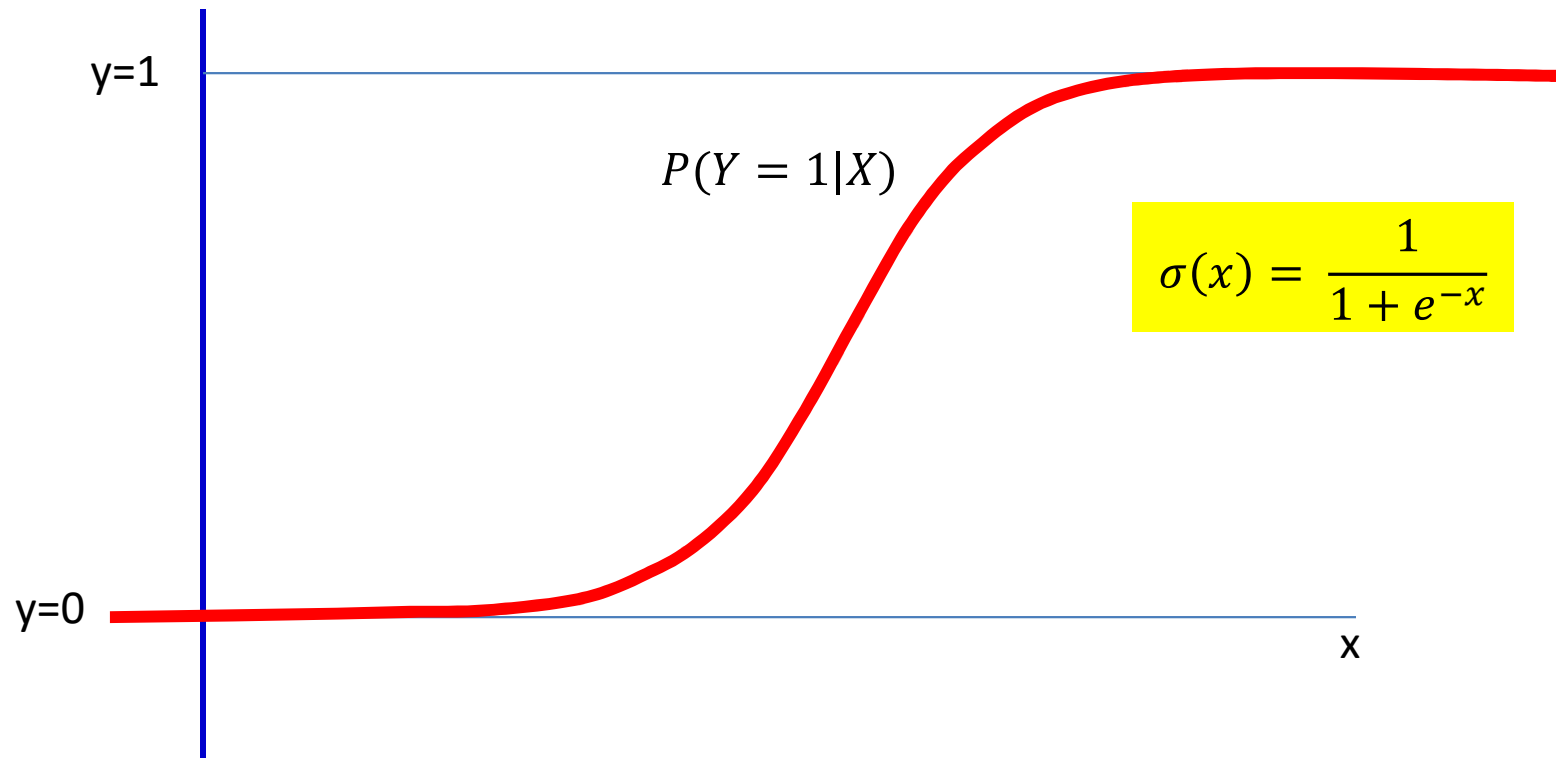
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



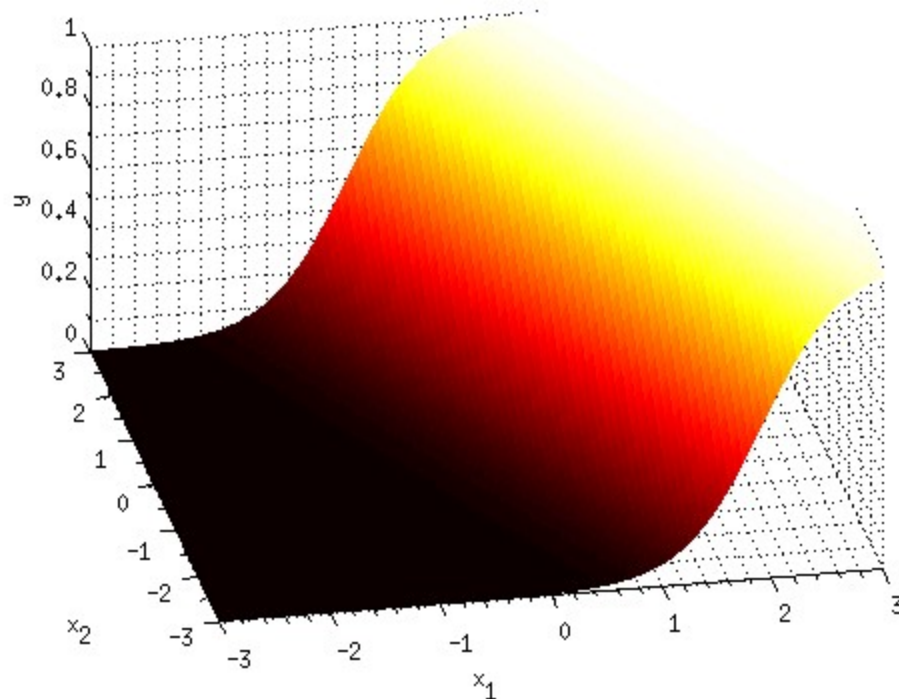
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The logistic regression model

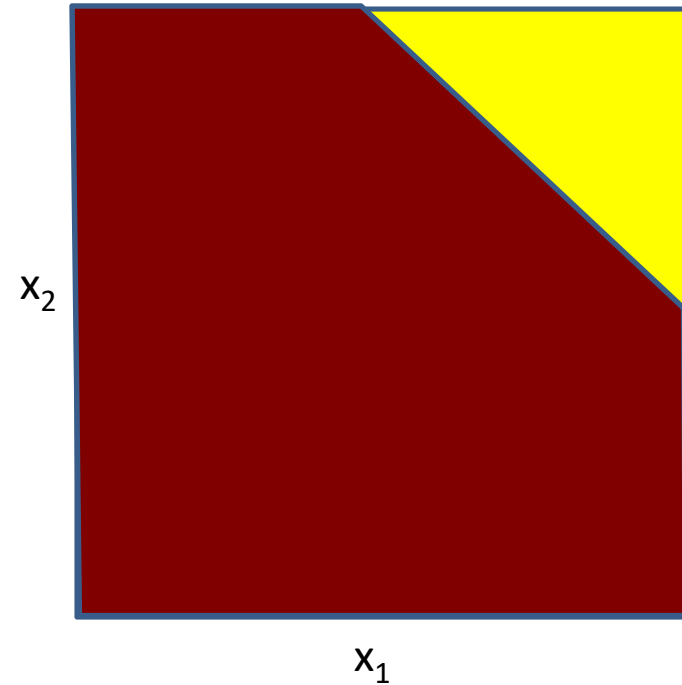


- Class 1 becomes increasingly probable going left to right
 - Very typical in many problems

Logistic regression



Decision: $y > 0.5$?



When X is a 2-D variable

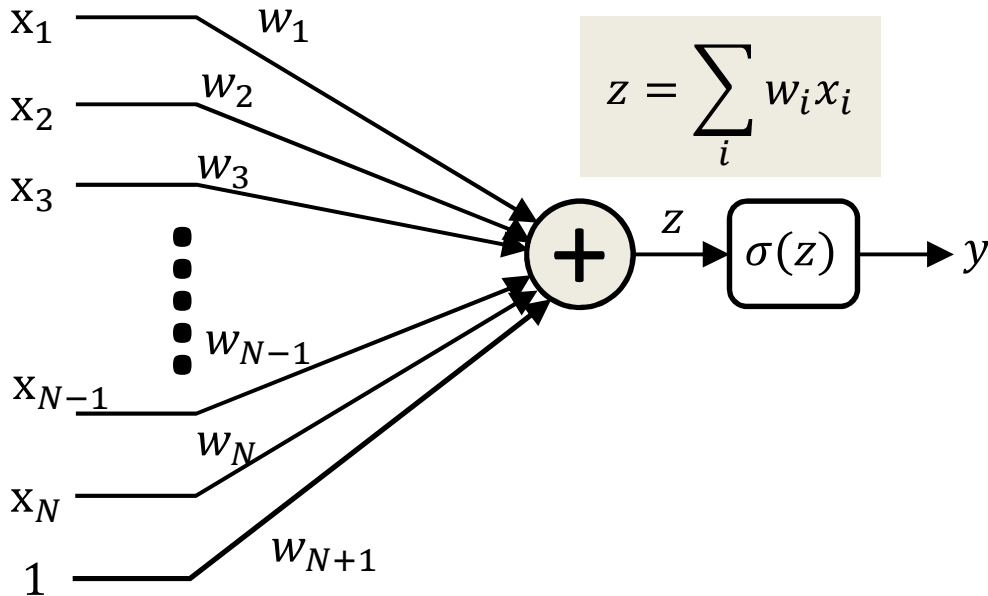
$$P(Y = 1|X) = \frac{1}{1 + \exp(-\sum_i w_i x_i - b)}$$

- This is the perceptron with a sigmoid activation
 - It actually computes the *probability* that the input belongs to class 1

Perceptrons and probabilities

- We will return to the fact that perceptrons with sigmoidal activations actually model class probabilities in a later lecture
- But for now moving on..

Perceptrons with differentiable activation functions



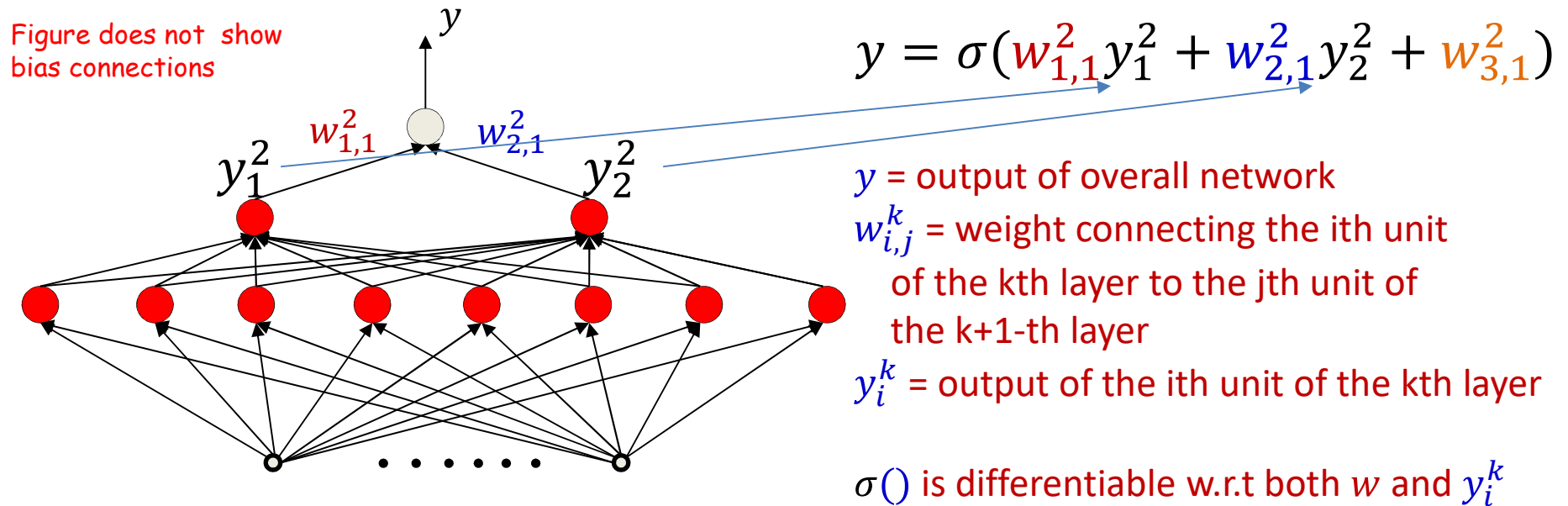
$$\frac{dy}{dz} = \sigma'(z)$$

$$\frac{dy}{dw_i} = \frac{dy}{dz} \frac{dz}{dw_i} = \sigma'(z) x_i$$

$$\frac{dy}{dx_i} = \frac{dy}{dz} \frac{dz}{dx_i} = \sigma'(z) w_i$$

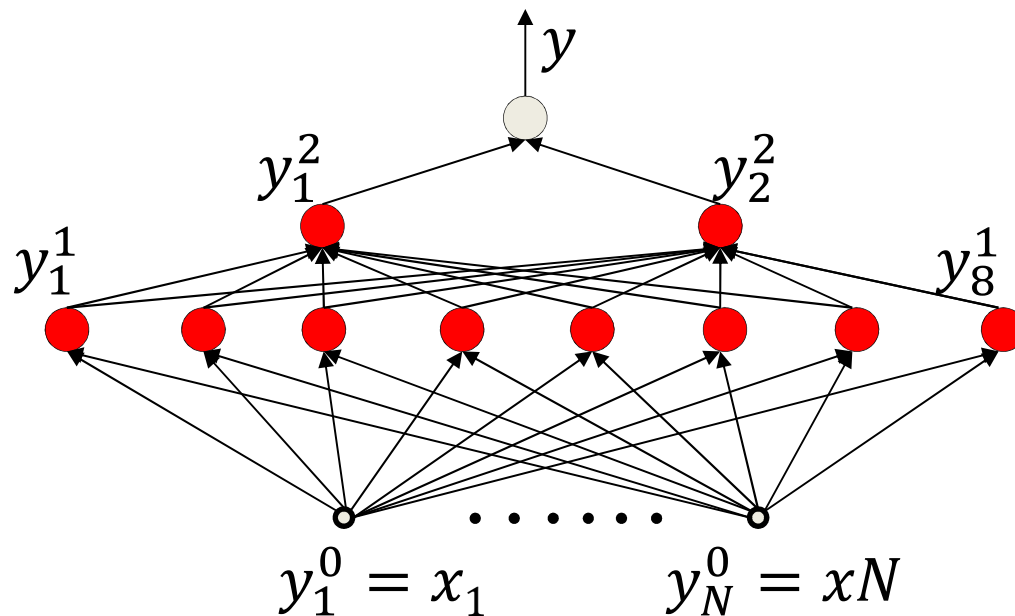
- $\sigma(z)$ is a differentiable function of z
 - $\frac{d\sigma(z)}{dz}$ is well-defined and finite for all z
- Using the chain rule, y is a differentiable function of both inputs x_i and weights w_i
- This means that we can compute the change in the output for *small* changes in either the input or the weights

Overall network is differentiable



- Every individual perceptron is differentiable w.r.t its inputs and its weights (including “bias” weight)
- By the chain rule, the overall function is differentiable w.r.t every parameter (weight or bias)
 - Small changes in the parameters result in measurable changes in output

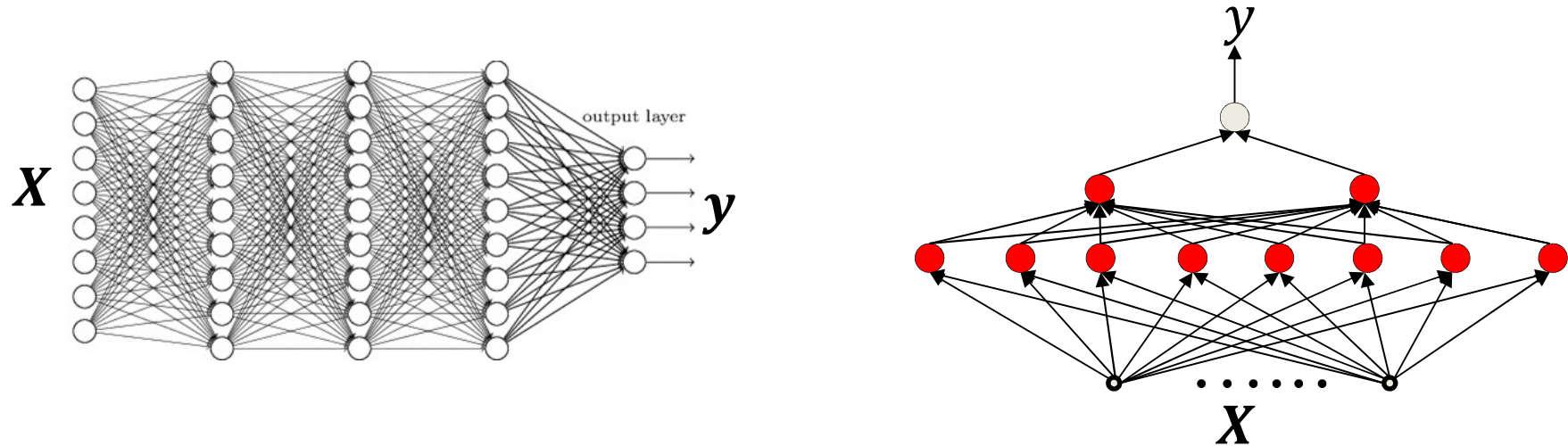
Overall function is differentiable



$$y_j^k = \sigma \left(\sum_i w_{i,j}^{k-1} y_i^{k-1} \right)$$

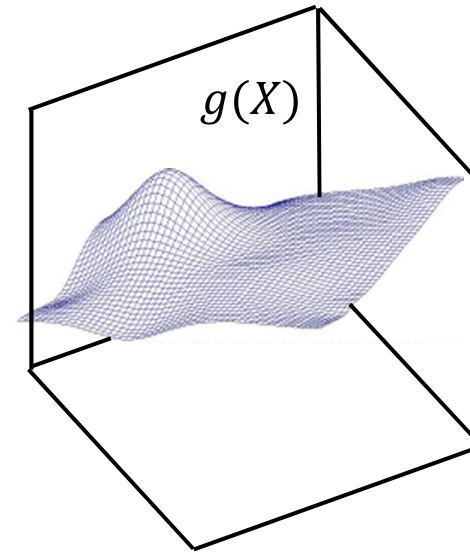
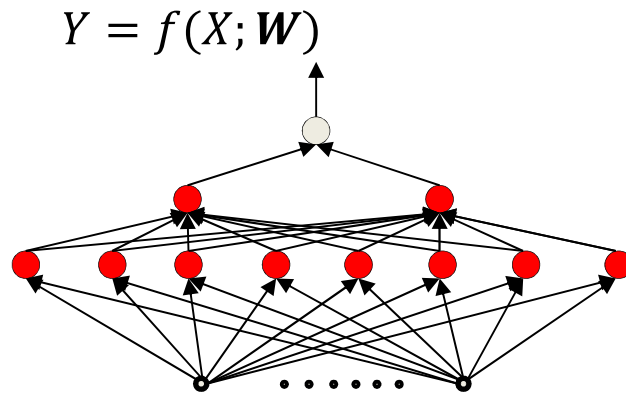
- The overall function is differentiable w.r.t every parameter
 - Small changes in the parameters result in measurable changes in the output
 - We will derive the actual derivatives using the chain rule later

Overall setting for “Learning” the MLP



- Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \dots, (X_N, d_N) \dots$
 - d is the *desired output* of the network in response to X
 - X and d may both be vectors
- ...we must find the network parameters such that the network produces the desired output for each training input
 - Or a close approximation of it
 - **The *architecture* of the network must be specified by us**

Recap: Learning the function

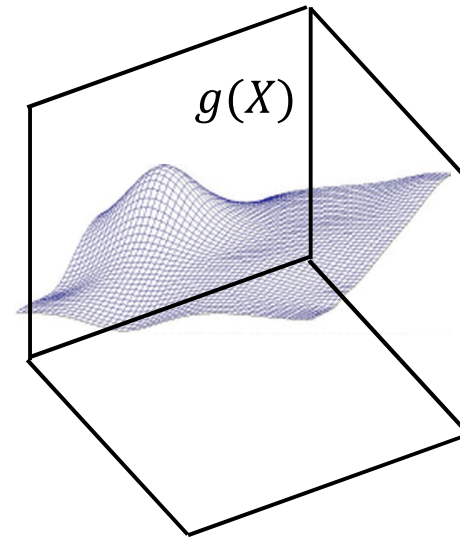
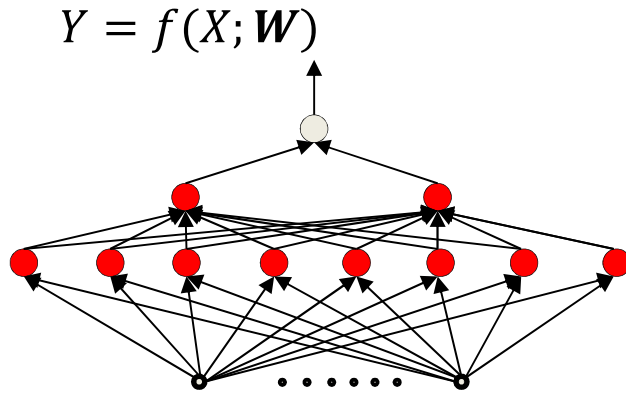


- When $f(X; \mathbf{W})$ has the capacity to exactly represent $g(X)$

$$\widehat{\mathbf{W}} = \operatorname{argmin}_{\mathbf{W}} \int_X \operatorname{div}(f(X; \mathbf{W}), g(X)) dX$$

- $\operatorname{div}()$ is a divergence function that goes to zero when $f(X; \mathbf{W}) = g(X)$

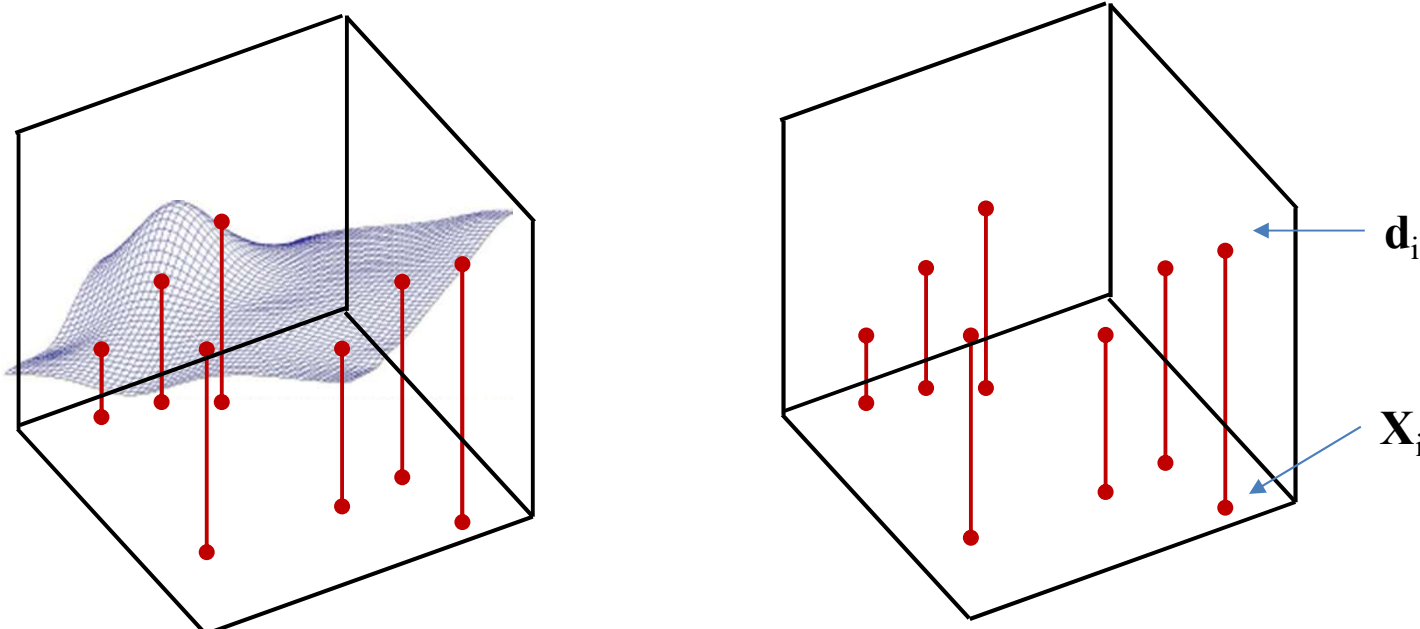
Minimizing *expected* error



- More generally, assuming X is a random variable

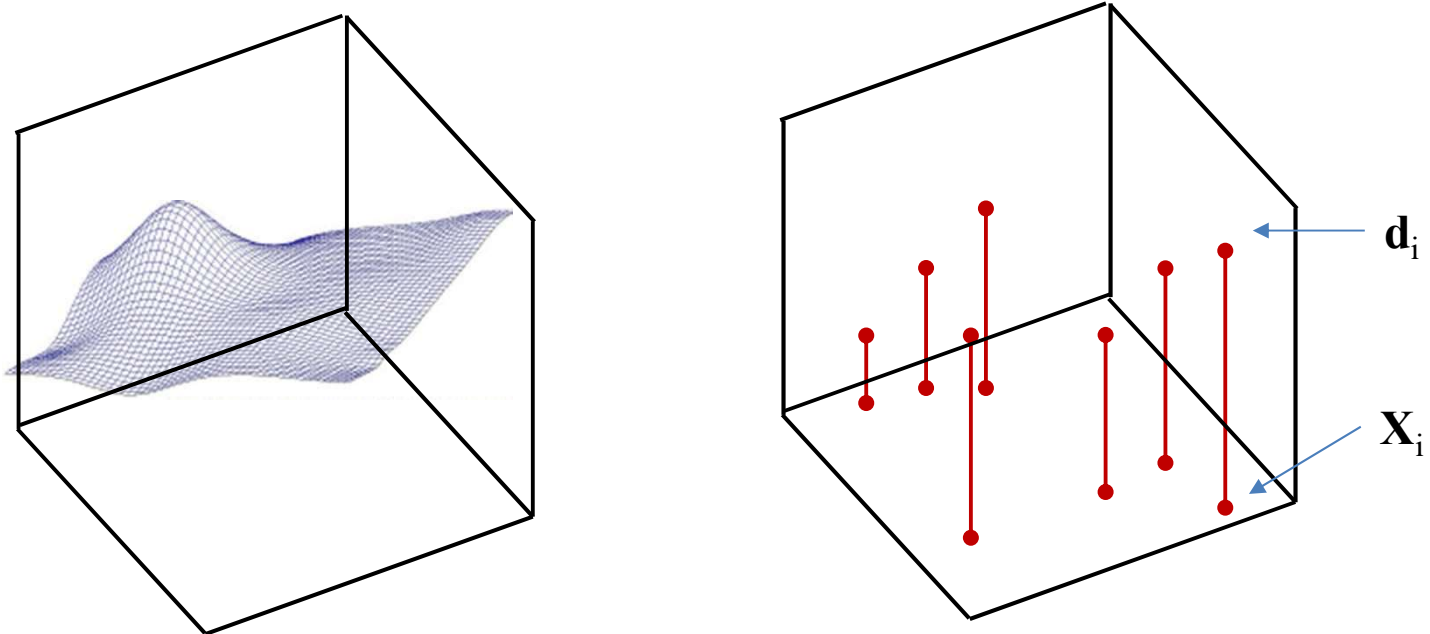
$$\begin{aligned}\widehat{\mathbf{W}} &= \operatorname{argmin}_W \int_X \operatorname{div}(f(X; W), g(X)) P(X) dX \\ &= \operatorname{argmin}_W E[\operatorname{div}(f(X; W), g(X))]\end{aligned}$$

Recap: Sampling the function



- *Sample $g(X)$*
 - Basically, get input-output pairs for a number of samples of input X_i
 - Many samples (X_i, d_i) , where $d_i = g(X_i) + \text{noise}$
 - Good sampling: the samples of X will be drawn from $P(X)$
- Estimate function from the samples

The *Empirical* risk



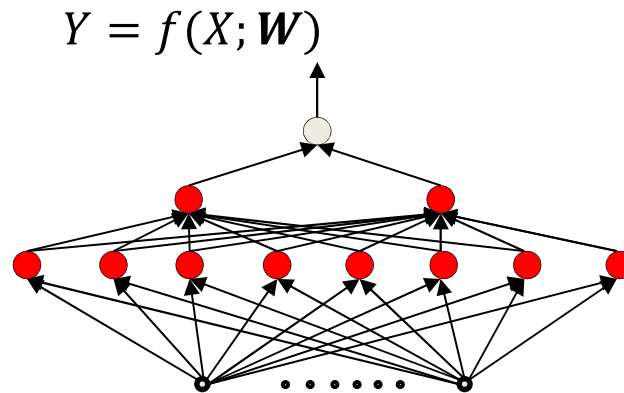
- The *expected* error (or risk) is the average error over the entire input space

$$E[\text{div}(f(X; W), g(X))] = \int_X \text{div}(f(X; W), g(X)) P(X) dX$$

- The *empirical estimate* of the expected error (or risk) is the *average* error over the samples

$$E[\text{div}(f(X; W), g(X))] \approx \frac{1}{N} \sum_{i=1}^N \text{div}(f(X_i; W), d_i)$$

Empirical Risk Minimization



- Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \dots, (X_N, d_N)$
 - Error on the i th instance: $\text{div}(f(X_i; W), d_i)$
 - Empirical average error (Empirical Risk) on all training data:

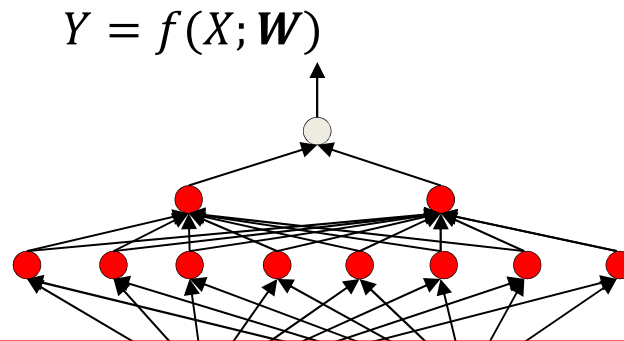
$$\text{Loss}(W) = \frac{1}{N} \sum_i \text{div}(f(X_i; W), d_i)$$

- Estimate the parameters to minimize the empirical estimate of expected error (empirical risk)

$$\widehat{W} = \underset{W}{\operatorname{argmin}} \text{Loss}(W)$$

- I.e. minimize the *empirical risk* over the drawn samples

Empirical Risk Minimization



Note : Its really a measure of error, but using standard terminology, we will call it a "Loss"

- Note 2: The empirical risk $Loss(W)$ is only an empirical approximation to the true risk $E[div(f(X; W), g(X))]$ which is our *actual* minimization objective

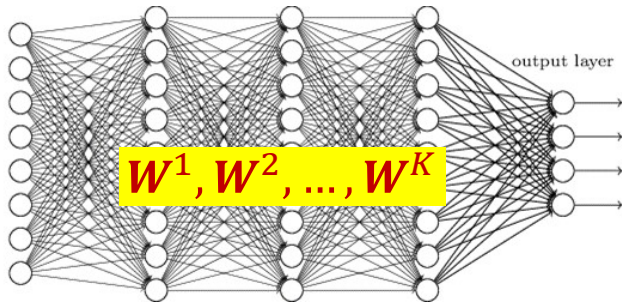
$$Loss(W) = \frac{1}{N} \sum_i div(f(X_i; W), d_i)$$

- Estimate the parameters to minimize the empirical estimate of expected error

$$\widehat{W} = \underset{W}{\operatorname{argmin}} Loss(W)$$

- I.e. minimize the *empirical error* over the drawn samples

ERM for neural networks



Actual output of network:

$$Y_i = \text{net}(X_i; \{w_{i,j}^k \forall i, j, k\}) \\ = \text{net}(X_i; W^1, W^2, \dots, W^K)$$

Desired output of network: d_i

Error on i-th training input: $\text{Div}(Y_i, d_i; W^1, W^2, \dots, W^K)$

Average training error(loss):

$$\text{Loss}(W^1, W^2, \dots, W^K) = \frac{1}{N} \sum_{i=1}^N \text{Div}(Y_i, d_i; W^1, W^2, \dots, W^K)$$

- What is the exact form of $\text{Div}()$? More on this later
- Optimize network parameters to minimize the total error over all training inputs

Problem Statement

- Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \dots, (X_N, d_N)$

- Minimize the following function

$$Loss(W) = \frac{1}{N} \sum_i div(f(X_i; W), d_i)$$

w.r.t W

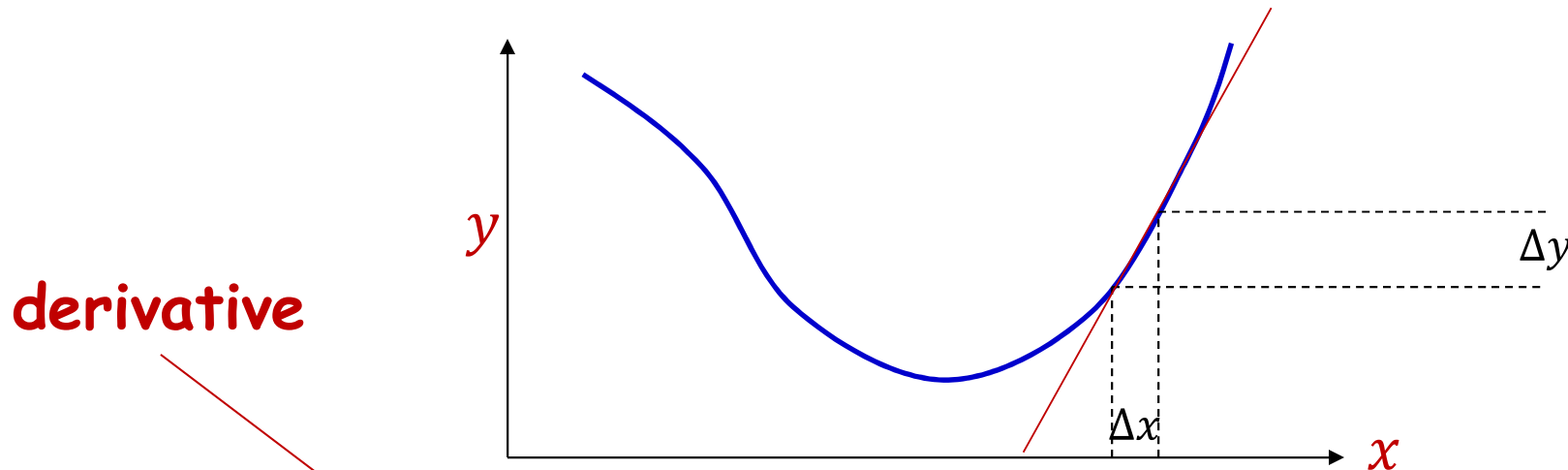
- This is problem of function minimization
 - An instance of optimization

Story so far

- We learn networks by “fitting” them to training instances drawn from a target function
- Learning networks of threshold-activation perceptrons requires solving a hard combinatorial-optimization problem
 - Because we cannot compute the influence of small changes to the parameters on the overall error
- Instead we use continuous activation functions with non-zero derivatives to enables us to estimate network parameters
 - This makes the output of the network differentiable w.r.t every parameter in the network
 - The *logistic* activation perceptron actually computes the *a posteriori* probability of the output given the input
- We define differentiable *divergence* between the output of the network and the desired output for the training instances
 - And a total error, which is the average divergence over all training instances
- We optimize network parameters to minimize this error
 - Empirical risk minimization
- This is an instance of function minimization

- **A CRASH COURSE ON FUNCTION OPTIMIZATION**

A brief note on derivatives..

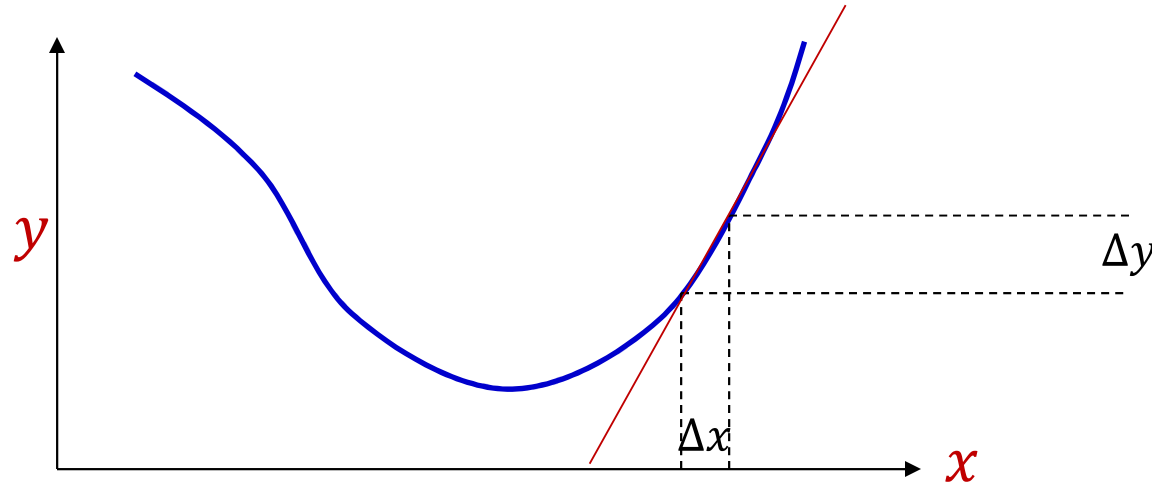


- A derivative of a function at any point tells us how much a minute increment to the *argument* of the function will increment the *value* of the function
 - For any $y = f(x)$, expressed as a multiplier α to a tiny increment Δx to obtain the increments Δy to the output

$$\Delta y = \alpha \Delta x$$

- Based on the fact that at a fine enough resolution, any smooth, continuous function is locally linear at any point

Scalar function of scalar argument



- When x and y are scalar

$$y = f(x)$$

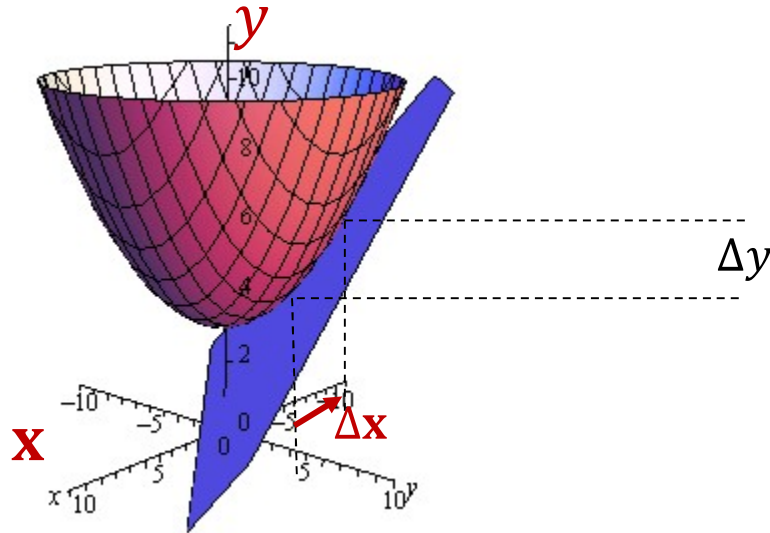
- Derivative:

$$\Delta y = \alpha \Delta x$$

- Often represented (using somewhat inaccurate notation) as $\frac{dy}{dx}$
- Or alternately (and more reasonably) as $f'(x)$

Multivariate scalar function:

Scalar function of *vector* argument



Note: $\Delta \mathbf{x}$ is now a vector

$$\Delta \mathbf{x} = \begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_D \end{bmatrix}$$

$$\Delta y = \alpha \Delta \mathbf{x}$$

- Giving us that α is a row vector: $\alpha = [\alpha_1 \quad \cdots \quad \alpha_D]$

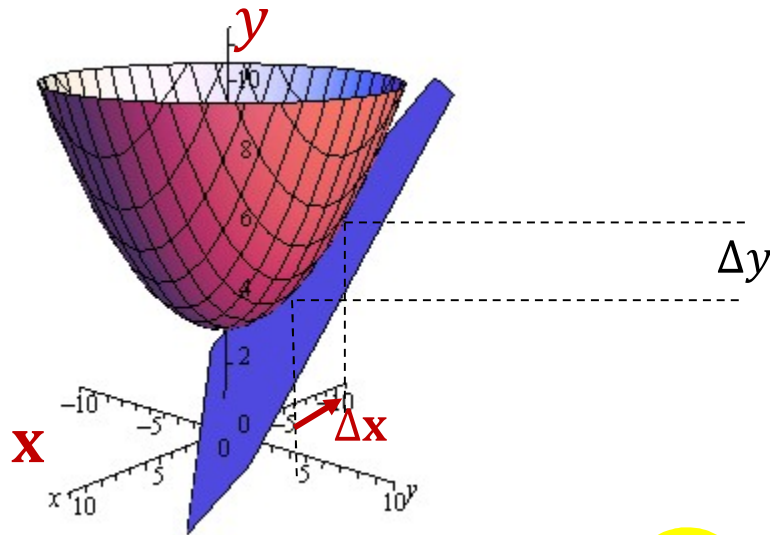
$$\Delta y = \alpha_1 \Delta x_1 + \alpha_2 \Delta x_2 + \cdots + \alpha_D \Delta x_D$$

- The *partial* derivative α_i gives us how y increments when *only* x_i is incremented
- Often represented as $\frac{\partial y}{\partial x_i}$

$$\Delta y = \frac{\partial y}{\partial x_1} \Delta x_1 + \frac{\partial y}{\partial x_2} \Delta x_2 + \cdots + \frac{\partial y}{\partial x_D} \Delta x_D$$

Multivariate scalar function:

Scalar function of *vector* argument



Note: $\Delta \mathbf{x}$ is now a vector

$$\Delta \mathbf{x} = \begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_D \end{bmatrix}$$

$$\Delta y = \nabla_{\mathbf{x}} y \Delta \mathbf{x}$$

- Where

$$\nabla_{\mathbf{x}} y = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \dots & \frac{\partial y}{\partial x_D} \end{bmatrix}$$

We will be using this symbol for vector and matrix derivatives

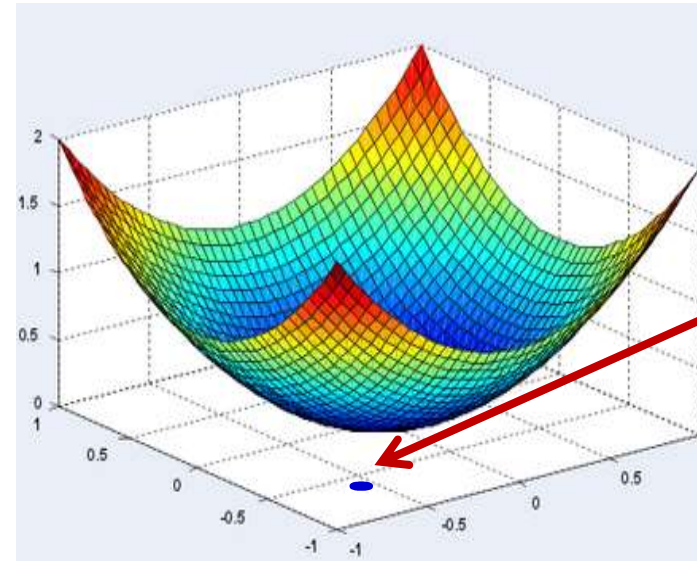
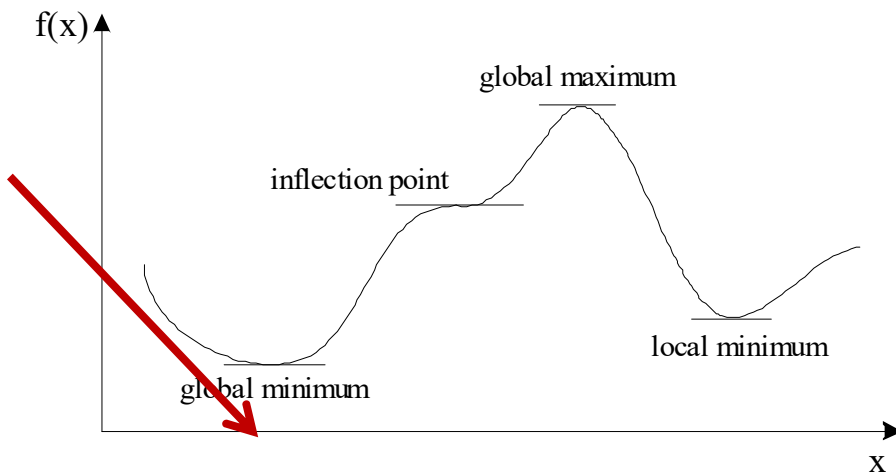
- You may be more familiar with the term “gradient” which is actually defined as the transpose of the derivative

Caveat about following slides

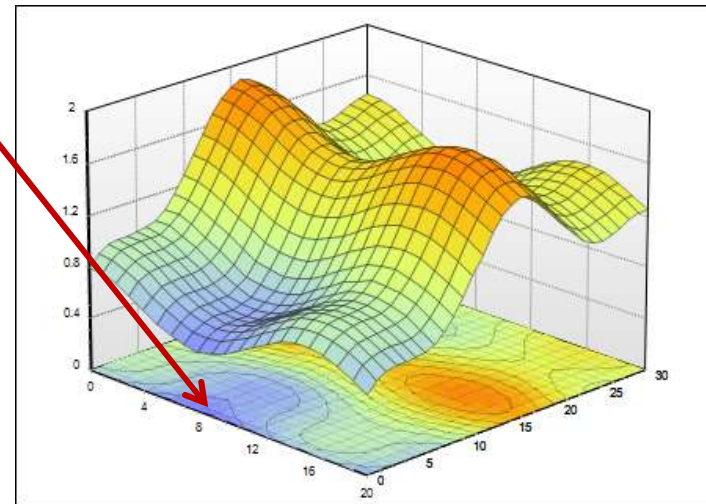
- The following slides speak of optimizing a function w.r.t a variable “ x ”
- This is only mathematical notation. In our actual network optimization problem we would be optimizing w.r.t. network weights “ w ”
- To reiterate – “ x ” in the slides represents the variable that we’re optimizing a function over and not the input to a neural network
- **Do not get confused!**



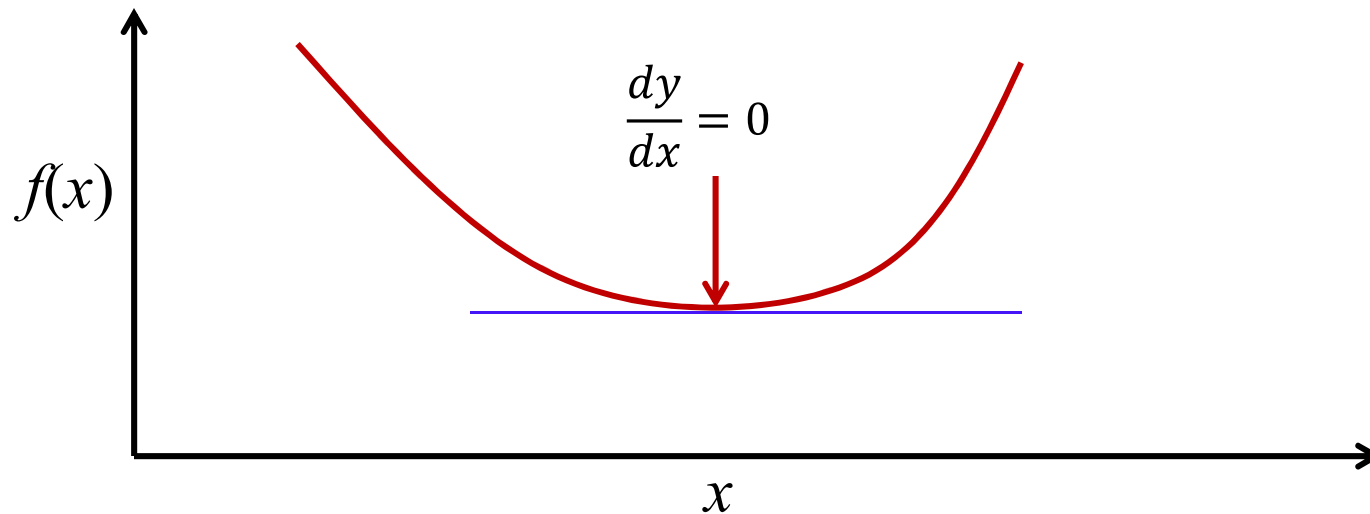
The problem of optimization



- General problem of optimization: find the value of x where $f(x)$ is minimum



Finding the minimum of a function

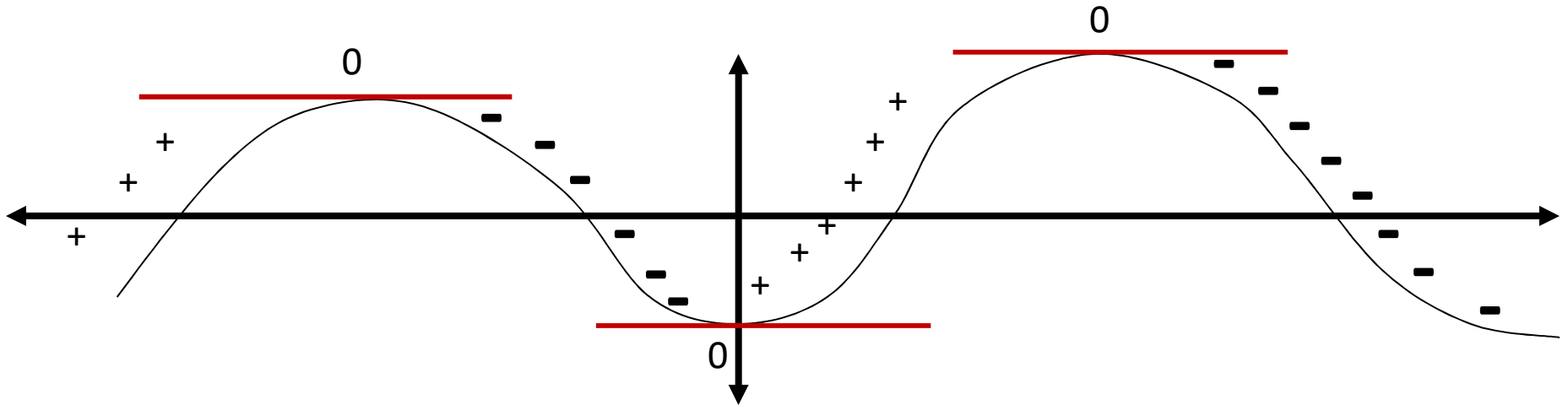


- Find the value x at which $f'(x) = 0$
 - Solve

$$\frac{df(x)}{dx} = 0$$

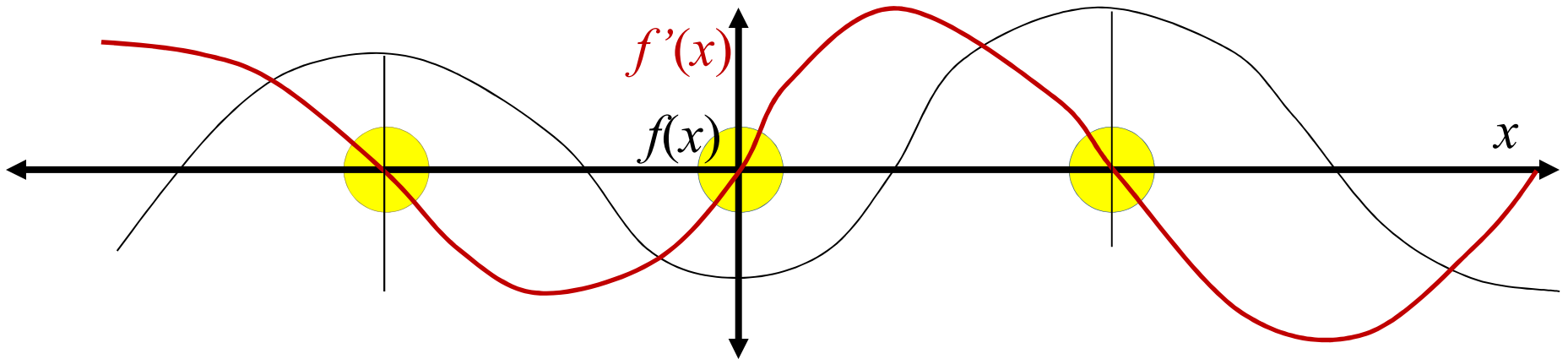
- The solution is a “turning point”
 - Derivatives go from positive to negative or vice versa at this point
- But is it a minimum?

Turning Points



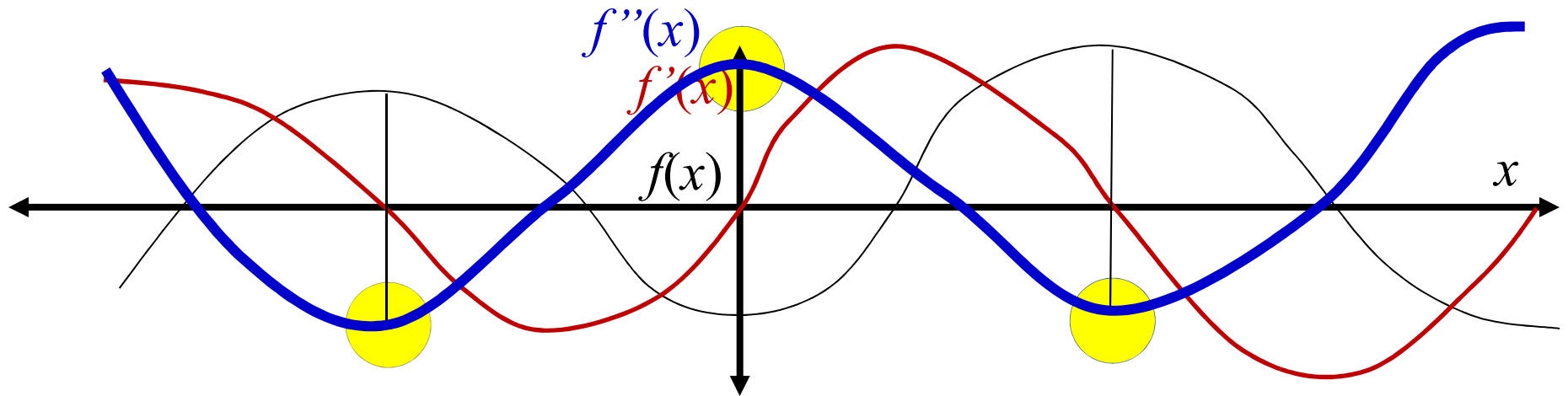
- Both *maxima* and *minima* have zero derivative
- Both are turning points

Derivatives of a curve



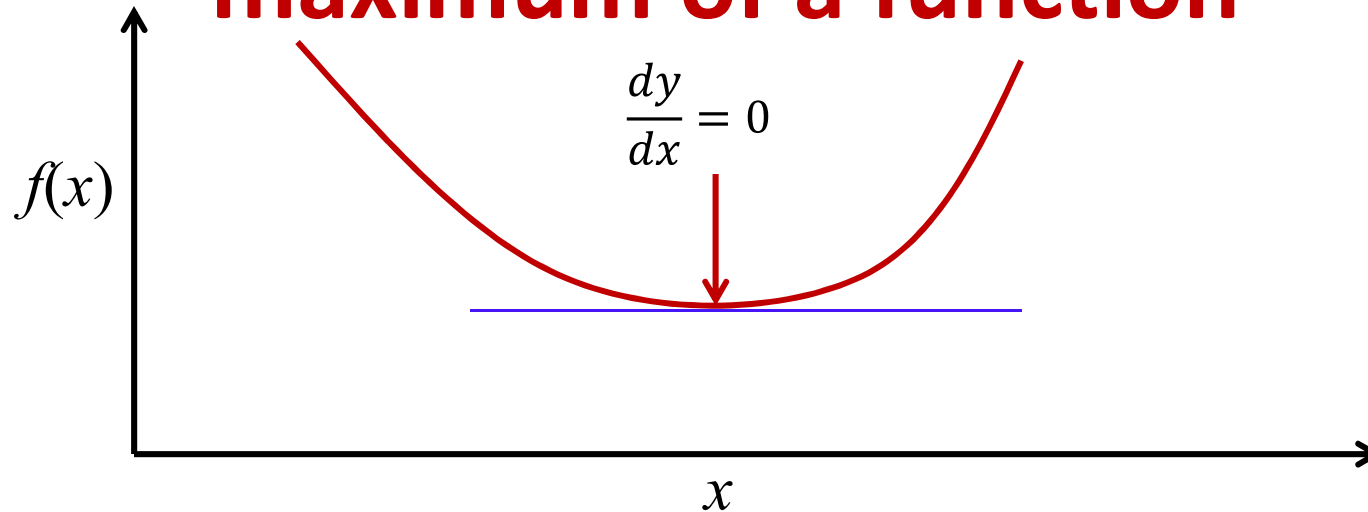
- Both *maxima* and *minima* are turning points
- Both *maxima* and *minima* have **zero derivative**

Derivative of the derivative of the curve



- Both *maxima* and *minima* are turning points
- Both *maxima* and *minima* have zero derivative
- The *second derivative* $f''(x)$ is –ve at maxima and +ve at minima!

Soln: Finding the minimum or maximum of a function



- Find the value x at which $f'(x) = 0$: Solve

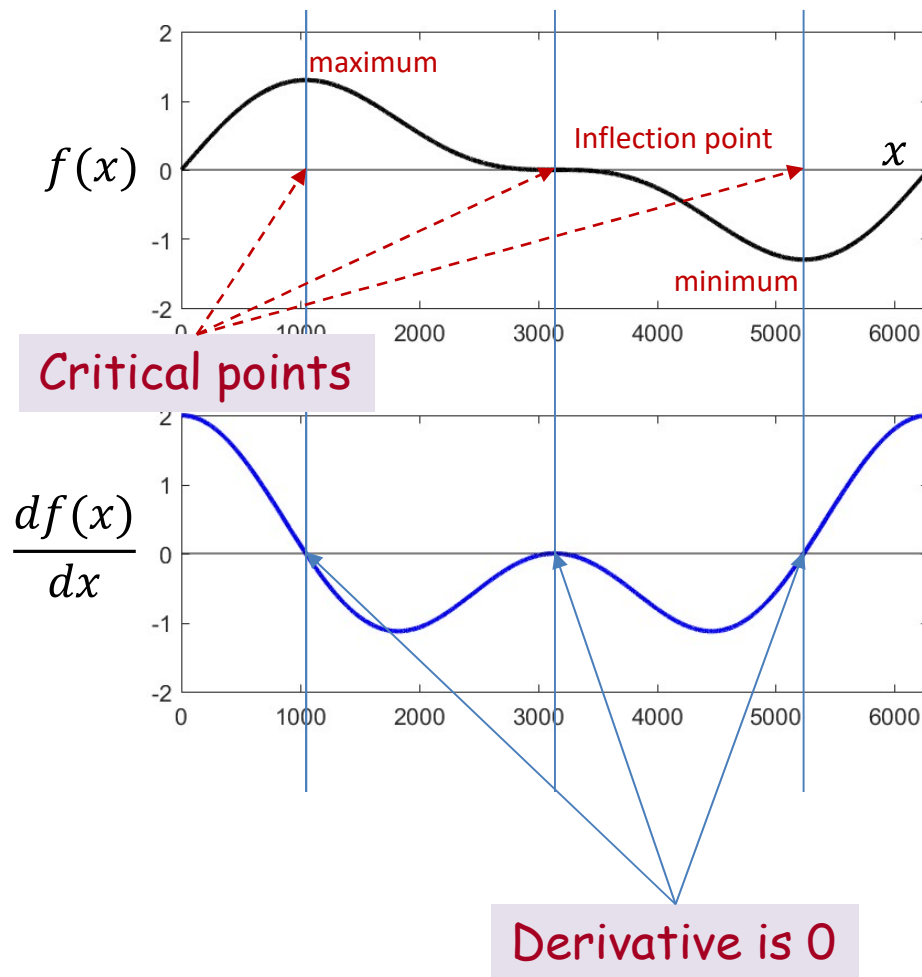
$$\frac{df(x)}{dx} = 0$$

- The solution x_{soln} is a turning point
- Check the double derivative at x_{soln} : compute

$$f''(x_{soln}) = \frac{df'(x_{soln})}{dx}$$

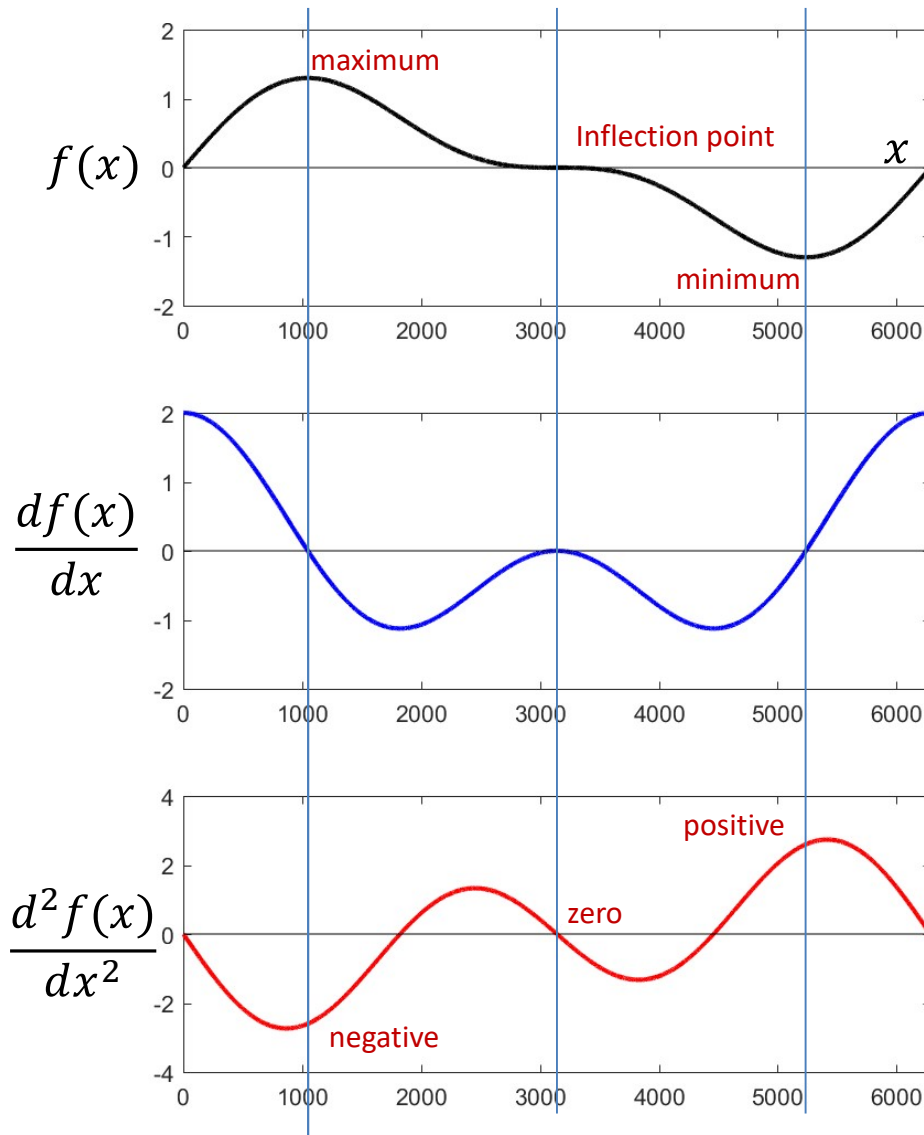
- If $f''(x_{soln})$ is positive x_{soln} is a minimum, otherwise it is a maximum

A note on derivatives of functions of single variable



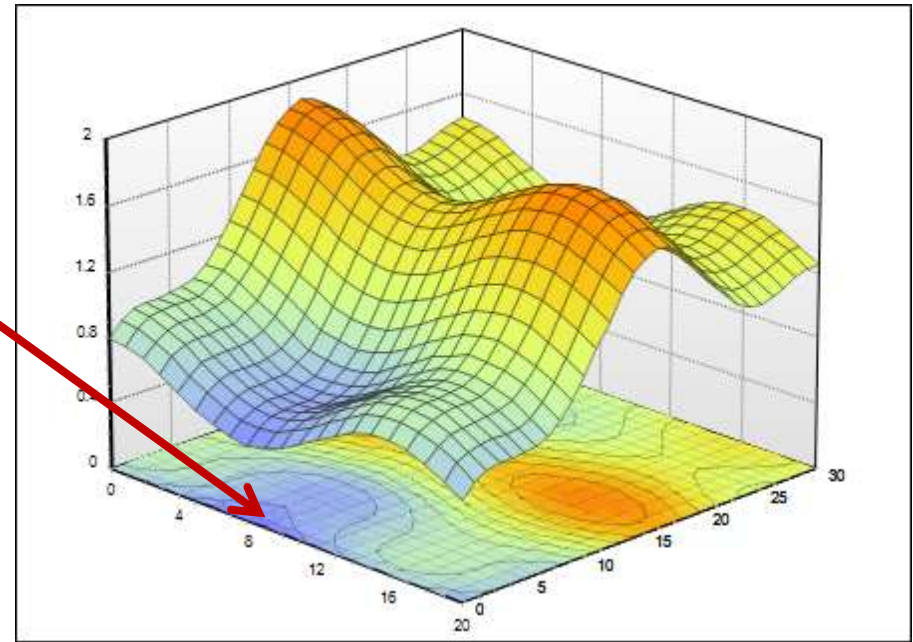
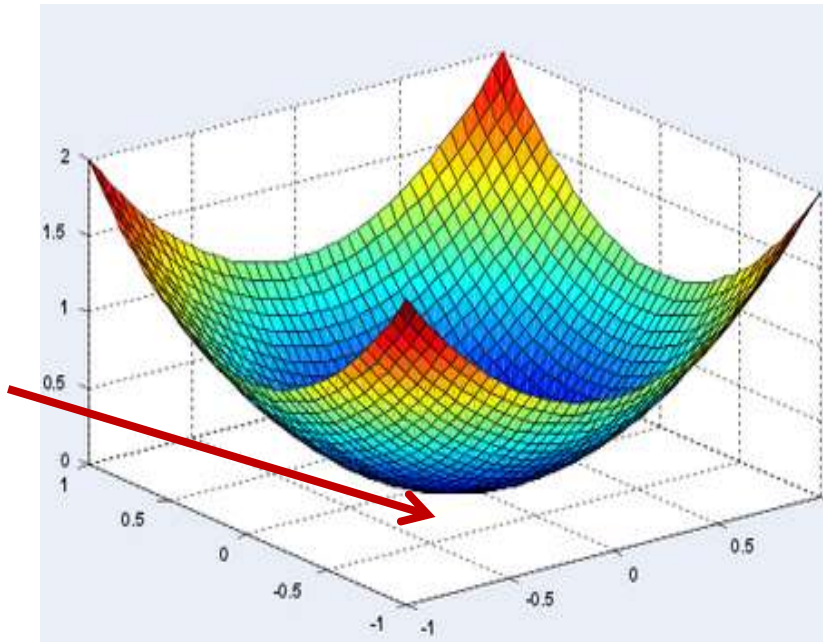
- All locations with zero derivative are *critical* points
 - These can be local maxima, local minima, or inflection points

A note on derivatives of functions of single variable



- All locations with zero derivative are *critical* points
 - These can be local maxima, local minima, or inflection points
- The *second* derivative is
 - ≥ 0 at minima
 - ≤ 0 at maxima
 - Zero at inflection points
- It's a little more complicated for functions of multiple variables..

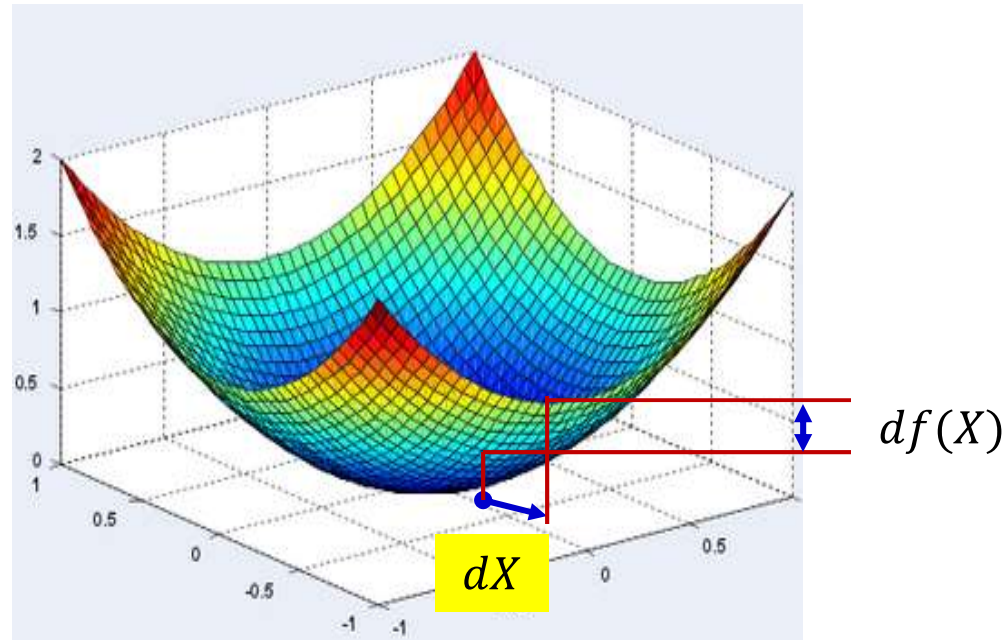
What about functions of multiple variables?



- The optimum point is still “turning” point
 - Shifting in any direction will increase the value
 - For smooth functions, miniscule shifts will not result in any change at all
- We must find a point where shifting in any direction by a microscopic amount will not change the value of the function

A brief note on derivatives of multivariate functions

The *Gradient* of a scalar function



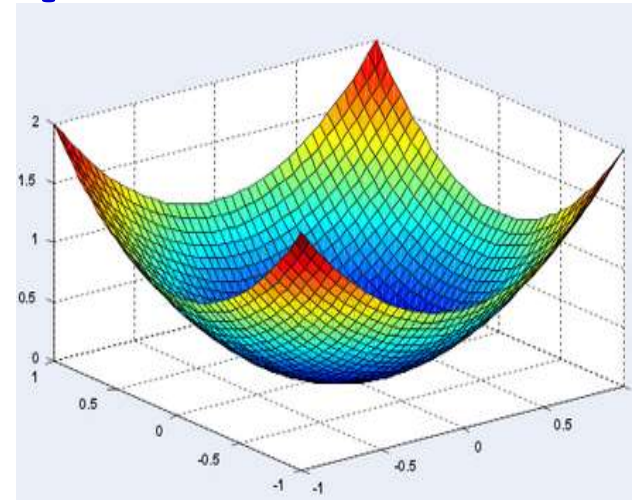
- The *derivative* $\nabla_X f(X)$ of a scalar function $f(X)$ of a multi-variate input X is a multiplicative factor that gives us the change in $f(X)$ for tiny variations in X

$$df(X) = \nabla_X f(X) dX$$

- The **gradient** is the transpose of the derivative $\nabla_X f(X)^T$

Gradients of scalar functions with multi-variate inputs

- Consider $f(X) = f(x_1, x_2, \dots, x_n)$



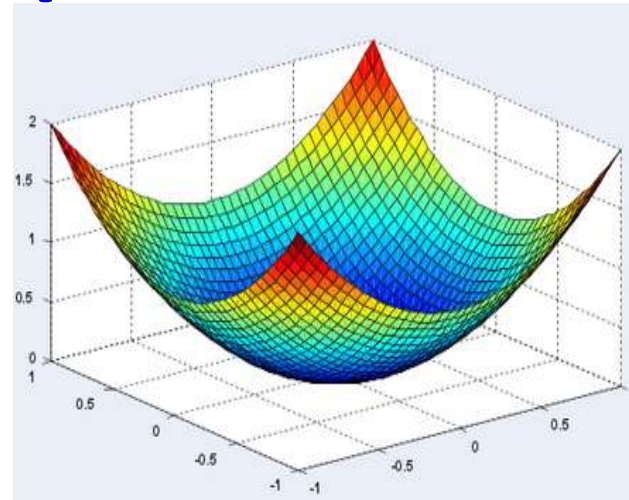
$$\nabla_X f(X) = \left[\frac{\partial f(X)}{\partial x_1} \quad \frac{\partial f(X)}{\partial x_2} \quad \dots \quad \frac{\partial f(X)}{\partial x_n} \right]$$

- Relation:

$$\begin{aligned} df(X) &= \nabla_X f(X) dX \\ &= \frac{\partial f(X)}{\partial x_1} dx_1 + \frac{\partial f(X)}{\partial x_2} dx_2 + \dots + \frac{\partial f(X)}{\partial x_n} dx_n \end{aligned}$$

Gradients of scalar functions with multi-variate inputs

- Consider $f(X) = f(x_1, x_2, \dots, x_n)$



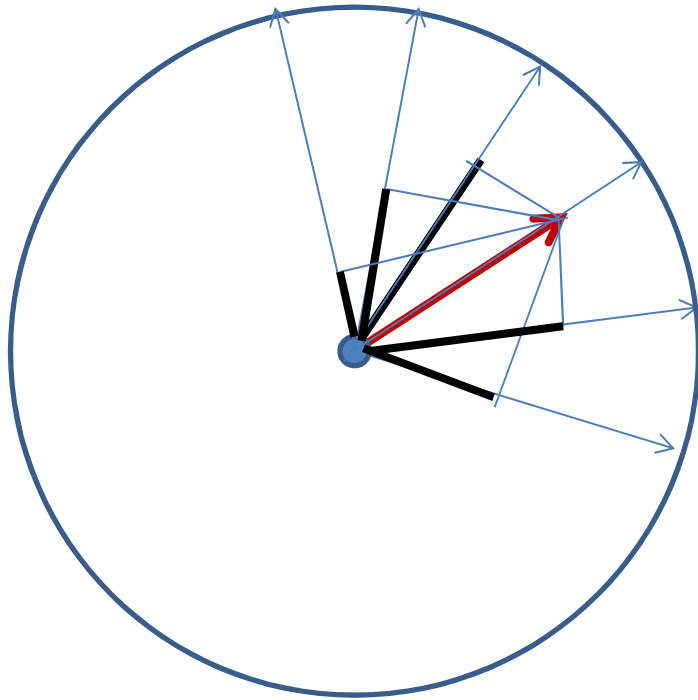
$$\nabla_X f(X) = \left[\frac{\partial f(X)}{\partial x_1} \quad \frac{\partial f(X)}{\partial x_2} \quad \dots \quad \frac{\partial f(X)}{\partial x_n} \right]$$

- Relation:

$$df(X) = \nabla_X f(X) dX$$

This is a vector inner product. To understand its behavior let's consider a well-known property of inner products

A well-known vector property



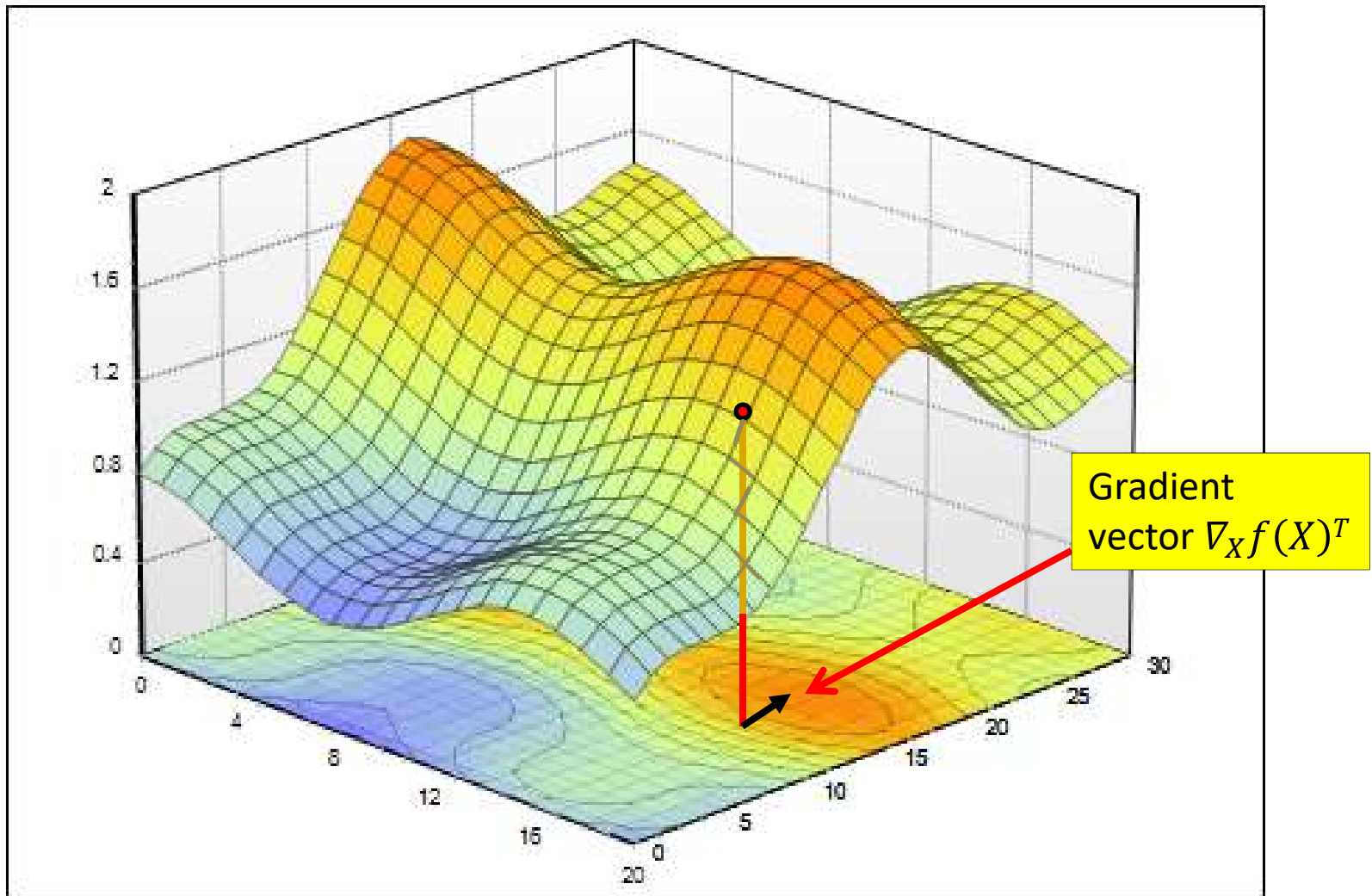
$$\mathbf{u}^T \mathbf{v} = |\mathbf{u}| |\mathbf{v}| \cos \theta$$

- The inner product between two vectors of fixed lengths is maximum when the two vectors are aligned
 - i.e. when $\theta = 0$

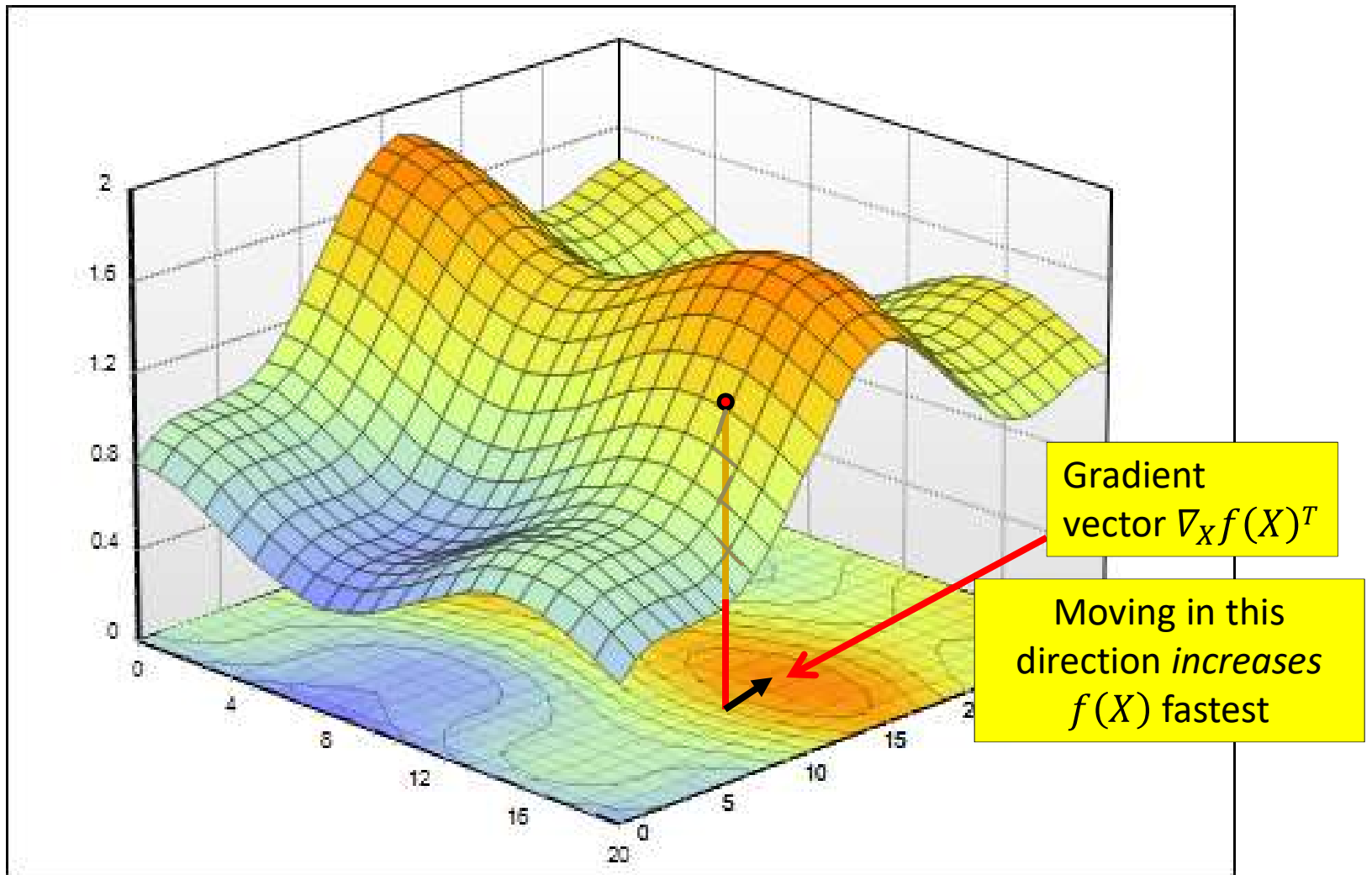
Properties of Gradient

- $df(X) = \nabla_X f(X) dX$
 - The inner product between $\nabla_X f(X)^T$ and dX
- Fixing the length of dX
 - E.g. $|dX| = 1$
- $df(X)$ is max if dX is aligned with $\nabla_X f(X)^T$
 - $\angle(\nabla_X f(X)^T, dX) = 0$
 - The function $f(X)$ increases most rapidly if the input increment dX is perfectly aligned to $\nabla_X f(X)^T$
- The gradient is the direction of fastest increase in $f(X)$

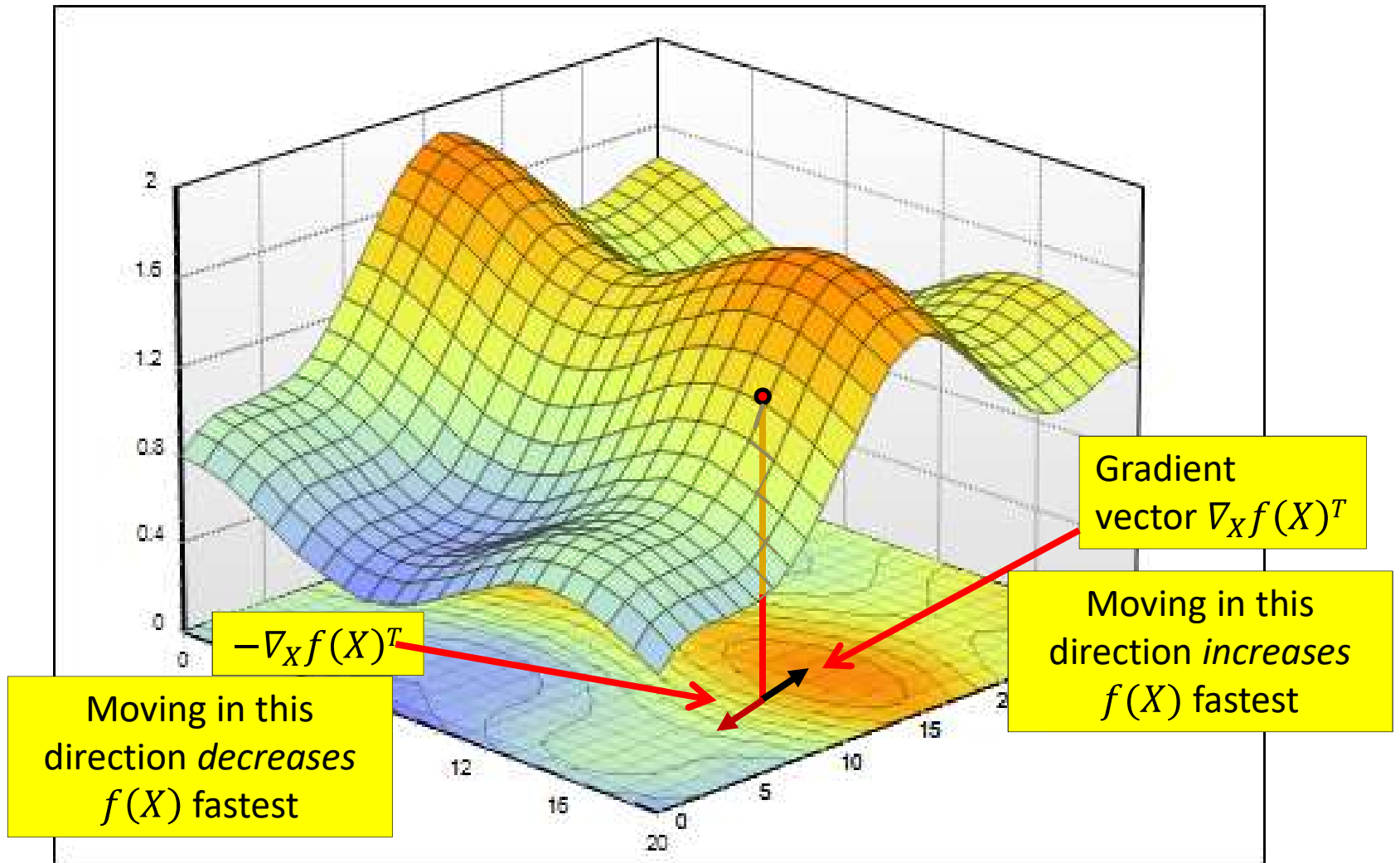
Gradient



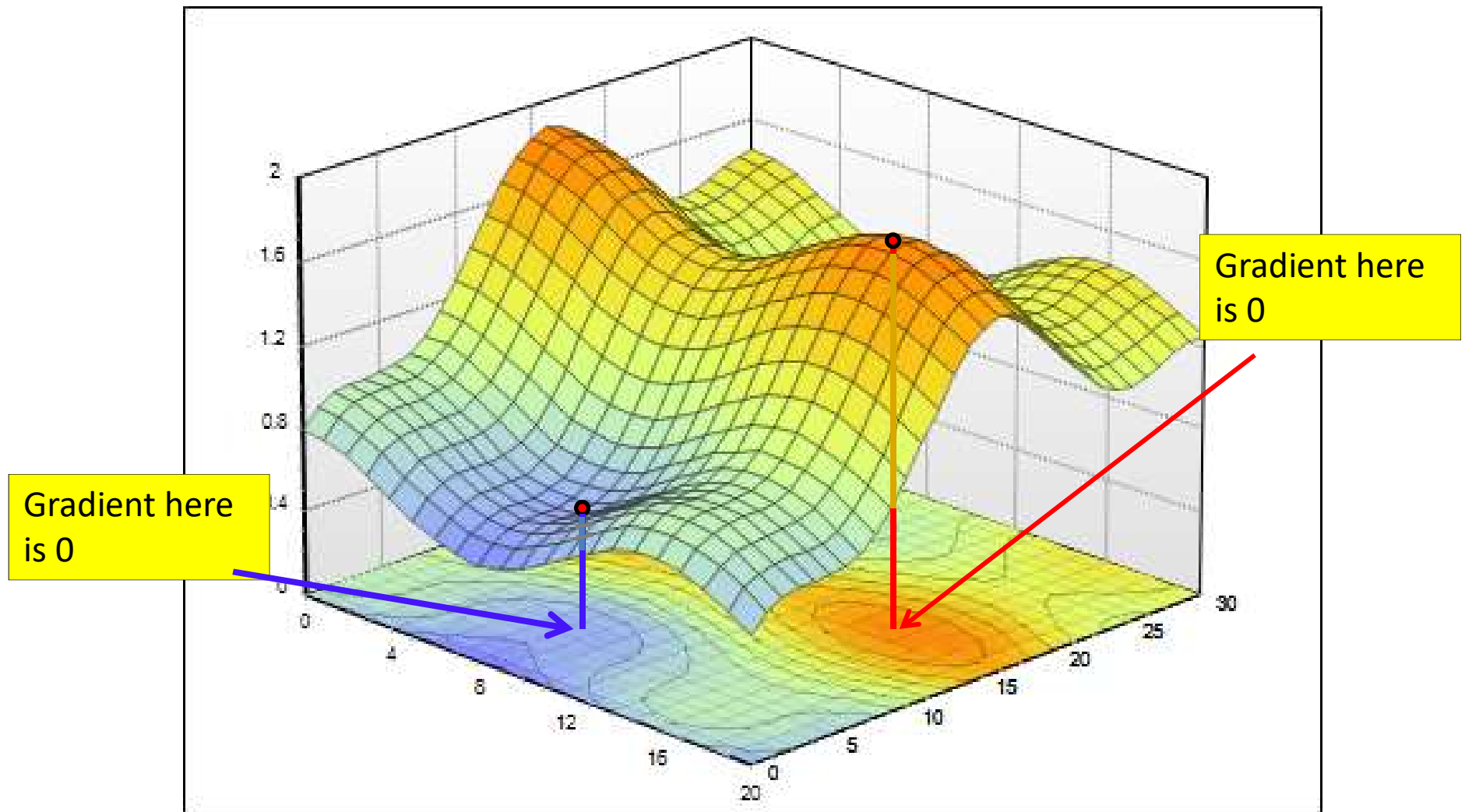
Gradient



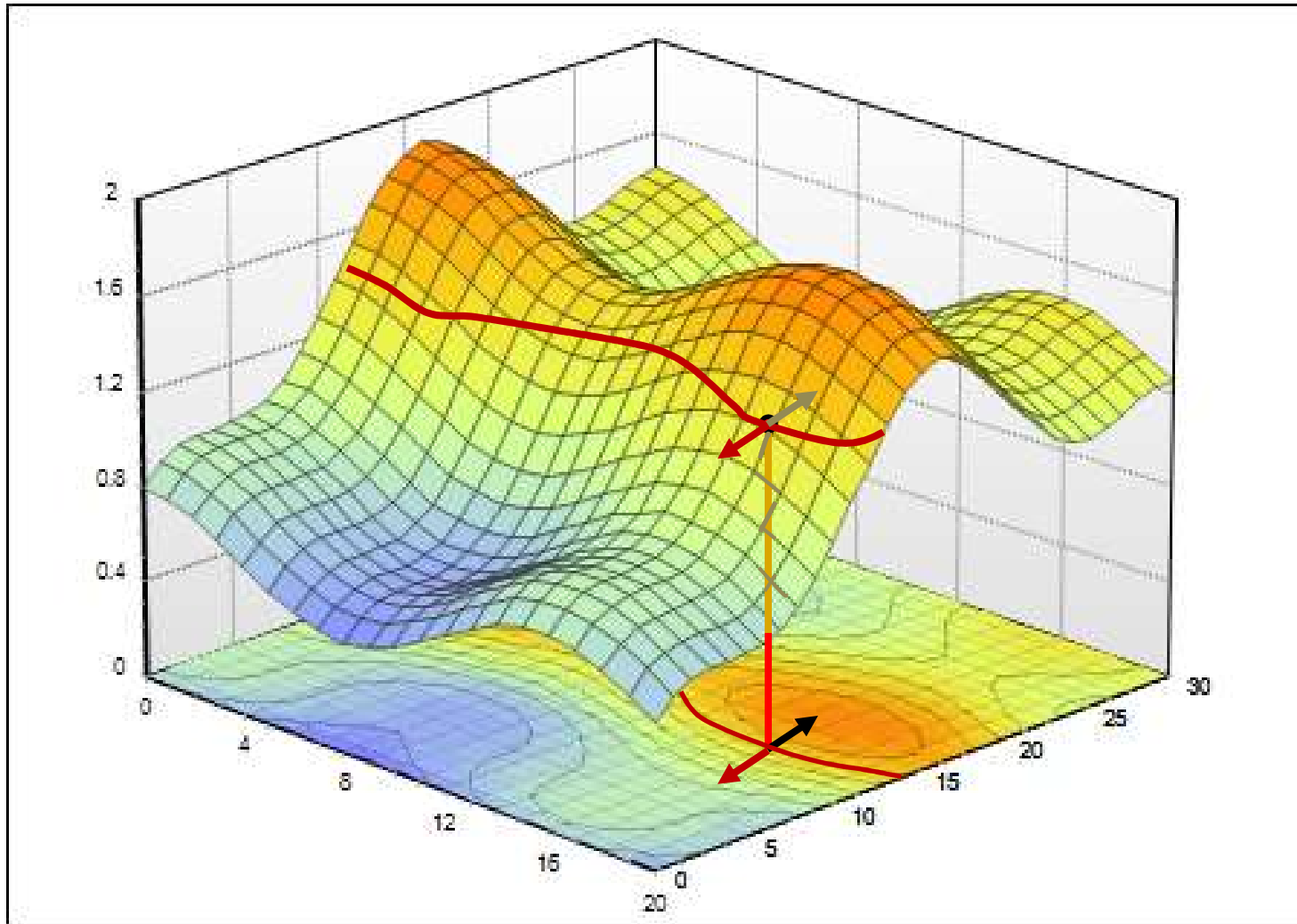
Gradient



Gradient



Properties of Gradient: 2



- The gradient vector $\nabla_x f(X)^T$ is perpendicular to the level curve

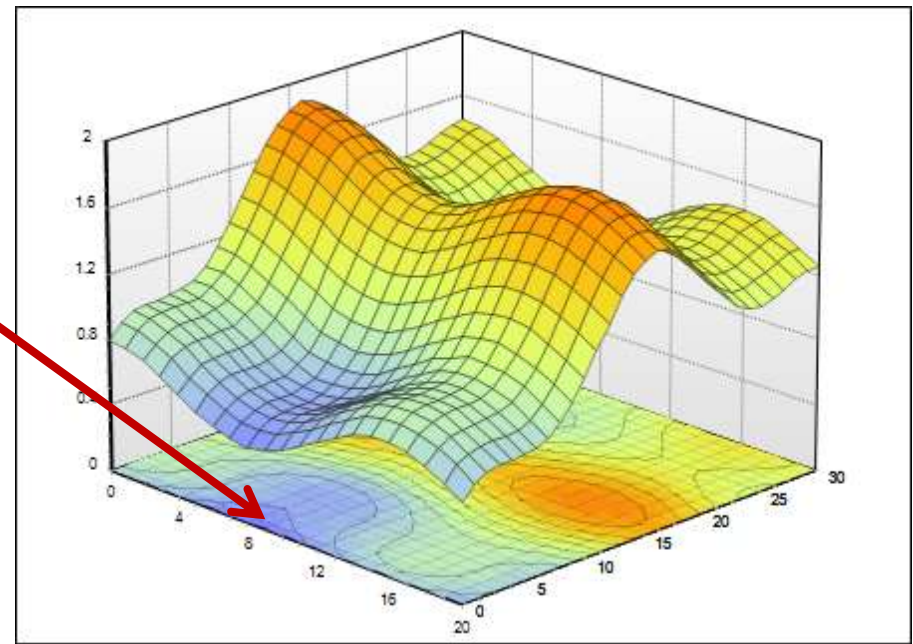
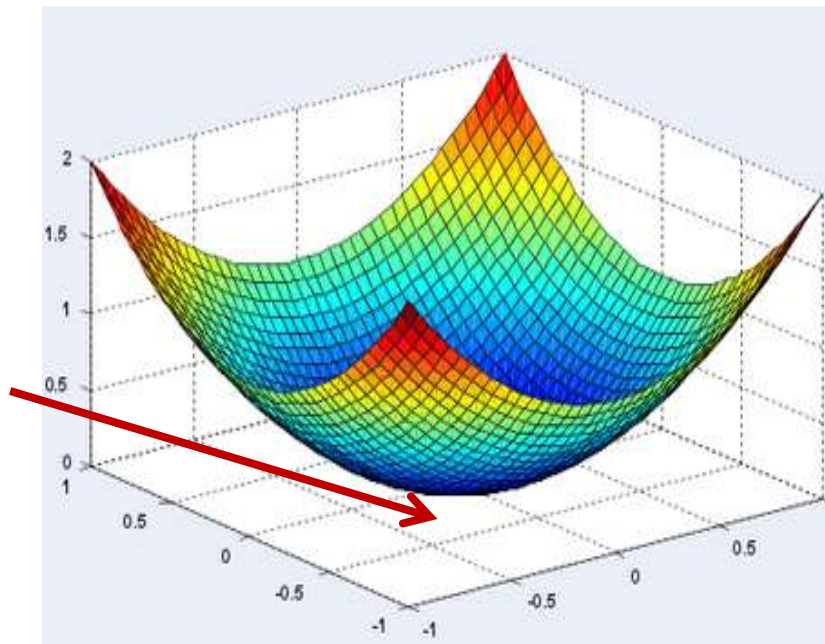
The Hessian

- The Hessian of a function $f(x_1, x_2, \dots, x_n)$ is given by the second derivative

$$\nabla_x^2 f(x_1, \dots, x_n) := \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdot & \cdot & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdot & \cdot & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdot & \cdot & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Returning to direct optimization...

Finding the minimum of a scalar function of a multi-variate input



- The optimum point is a turning point – the gradient will be 0

Unconstrained Minimization of function (Multivariate)

1. Solve for the X where the derivative (or gradient) equals to zero

$$\nabla_X f(X) = 0$$

2. Compute the Hessian Matrix $\nabla_X^2 f(X)$ at the candidate solution and verify that
 - Hessian is positive definite (eigenvalues positive) -> to identify local minima
 - Hessian is negative definite (eigenvalues negative) -> to identify local maxima

Unconstrained Minimization of function (Example)

- Minimize

$$f(x_1, x_2, x_3) = (x_1)^2 + x_1(1 - x_2) + (x_2)^2 - x_2x_3 + (x_3)^2 + x_3$$

- Gradient

$$\nabla_X f^T = \begin{bmatrix} 2x_1 + 1 - x_2 \\ -x_1 + 2x_2 - x_3 \\ -x_2 + 2x_3 + 1 \end{bmatrix}$$

Unconstrained Minimization of function (Example)

- Set the gradient to null

$$\nabla_x f = 0 \Rightarrow \begin{bmatrix} 2x_1 + 1 - x_2 \\ -x_1 + 2x_2 - x_3 \\ -x_2 + 2x_3 + 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

- Solving the 3 equations system with 3 unknowns

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}$$

Unconstrained Minimization of function (Example)

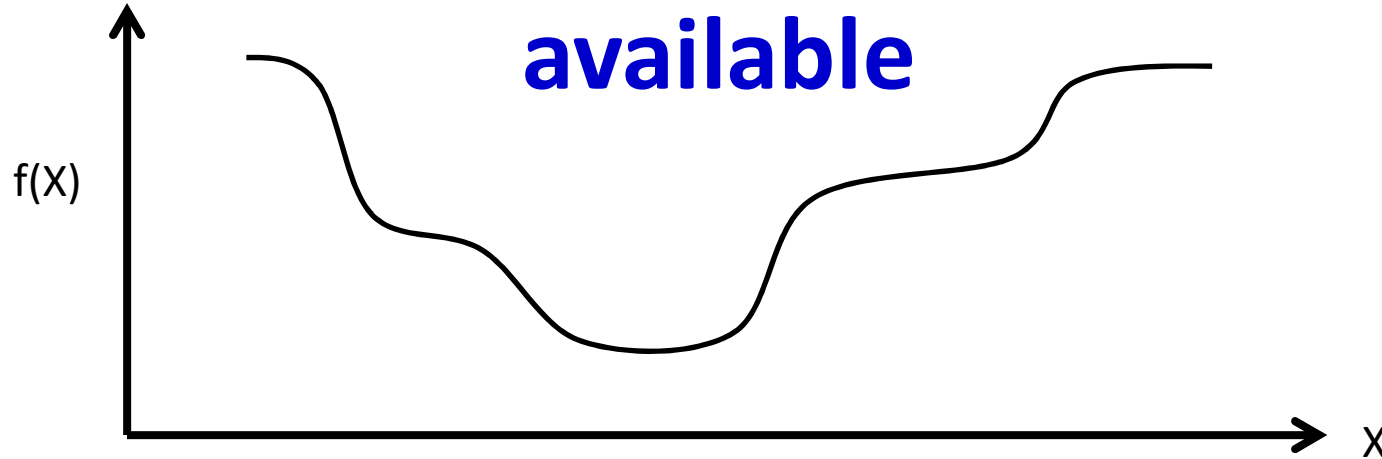
- Compute the Hessian matrix $\nabla_x^2 f = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$
- Evaluate the eigenvalues of the Hessian matrix

$$\lambda_1 = 3.414, \quad \lambda_2 = 0.586, \quad \lambda_3 = 2$$

- All the eigenvalues are positives \Rightarrow the Hessian matrix is positive definite

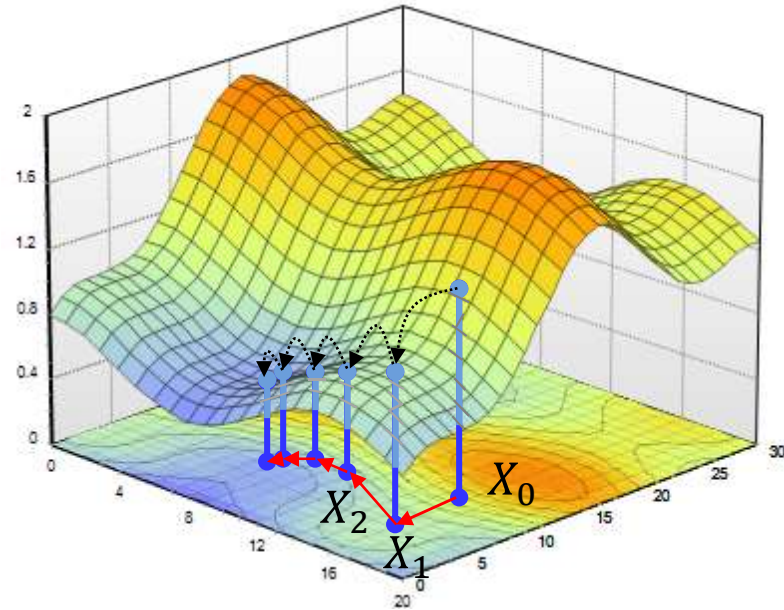
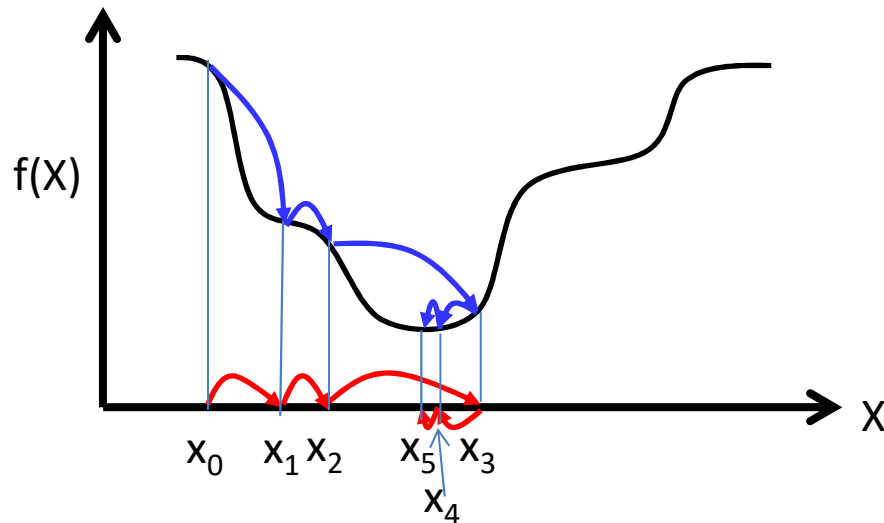
- The point $x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}$ is a minimum

Closed Form Solutions are not always available



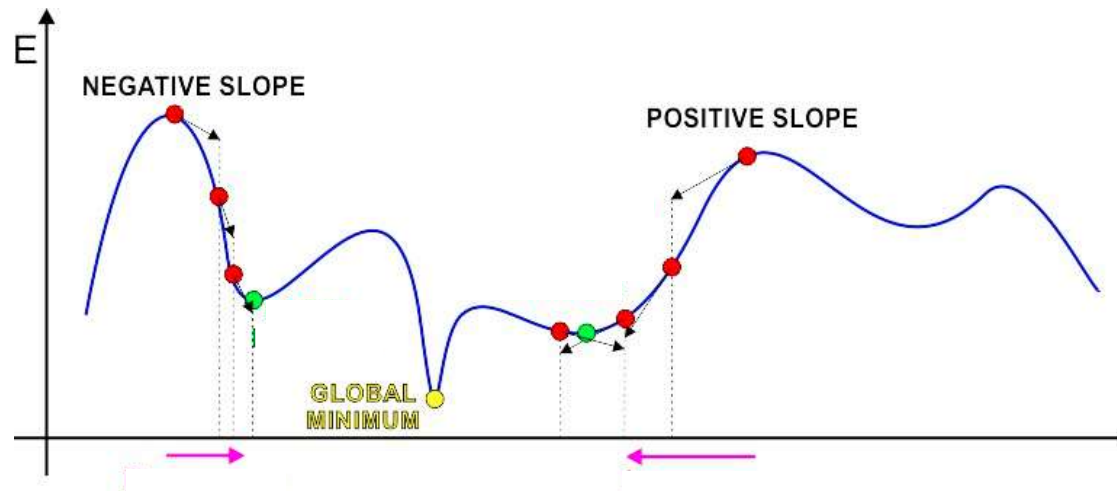
- Often it is not possible to simply solve $\nabla_x f(X) = 0$
 - The function to minimize/maximize may have an intractable form
- In these situations, iterative solutions are used
 - Begin with a “guess” for the optimal X and refine it iteratively until the correct value is obtained

Iterative solutions



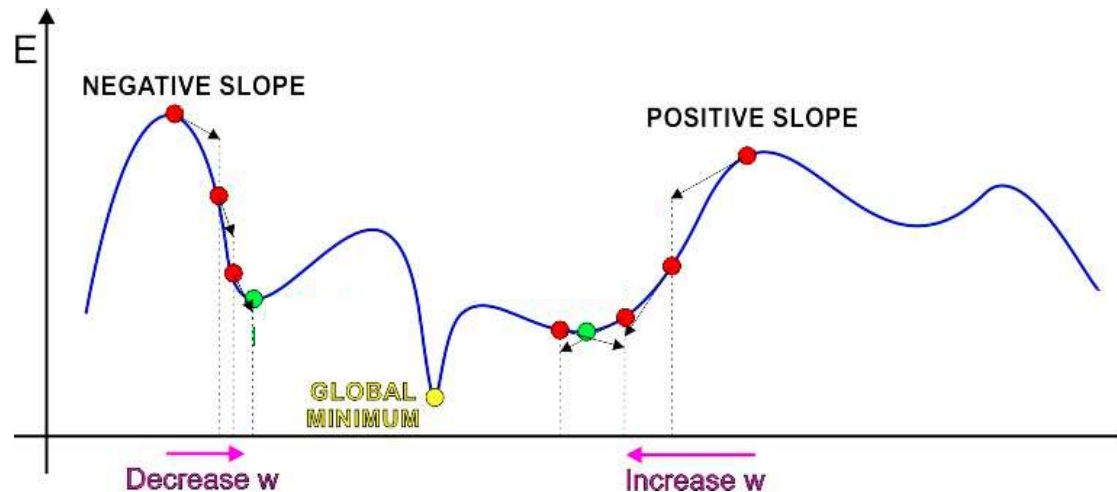
- Iterative solutions
 - Start from an initial guess X_0 for the optimal X
 - Update the guess towards a (hopefully) “better” value of $f(X)$
 - Stop when $f(X)$ no longer decreases
- Problems:
 - Which direction to step in
 - How big must the steps be

The Approach of Gradient Descent



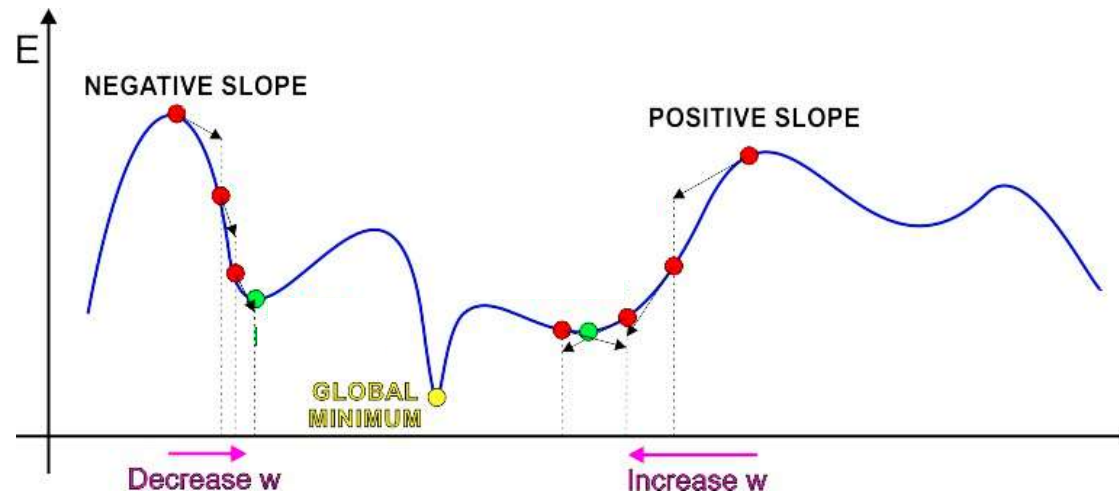
- Iterative solution:
 - Start at some point
 - Find direction in which to shift this point to decrease error
 - This can be found from the derivative of the function
 - A positive derivative \rightarrow moving left decreases error
 - A negative derivative \rightarrow moving right decreases error
 - Shift point in this direction

The Approach of Gradient Descent



- Iterative solution: Trivial algorithm
 - Initialize x^0
 - While $f'(x^k) \neq 0$
 - If $\text{sign}(f'(x^k))$ is positive:
$$x^{k+1} = x^k - \text{step}$$
 - Else
$$x^{k+1} = x^k + \text{step}$$
- What must step be to ensure we actually get to the optimum?

The Approach of Gradient Descent



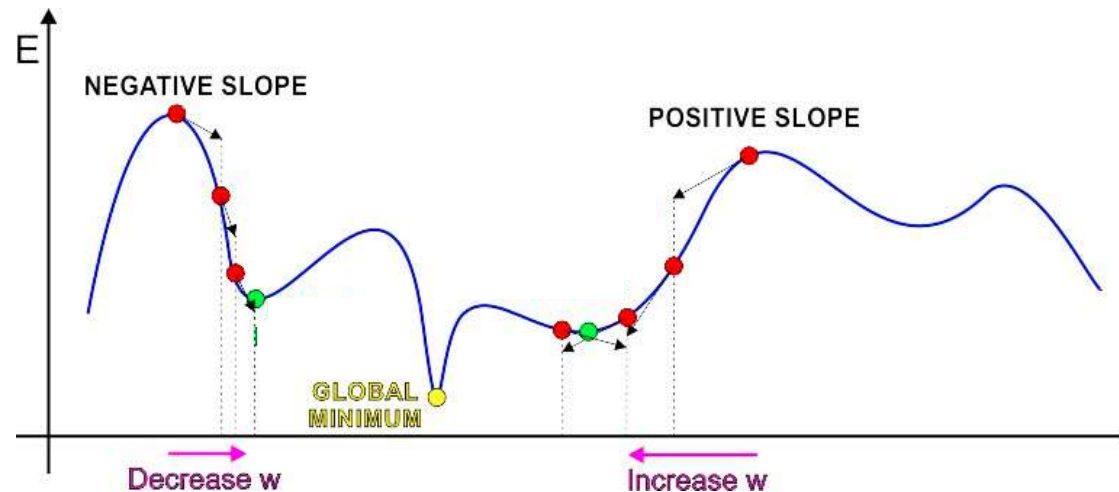
- Iterative solution: Trivial algorithm

- Initialize x^0
- While $f'(x^k) \neq 0$

$$x^{k+1} = x^k - \text{sign}(f'(x^k)) \cdot \text{step}$$

- Identical to previous algorithm

The Approach of Gradient Descent



- Iterative solution: Trivial algorithm
 - Initialize x^0
 - While $f'(x^k) \neq 0$
$$x^{k+1} = x^k - \eta^k f'(x^k)$$
- η^k is the “step size”

Gradient descent/ascent (multivariate)

- The gradient descent/ascent method to find the minimum or maximum of a function f iteratively
 - To find a *maximum* move *in the direction of the gradient*

$$x^{k+1} = x^k + \eta^k \nabla_x f(x^k)^T$$

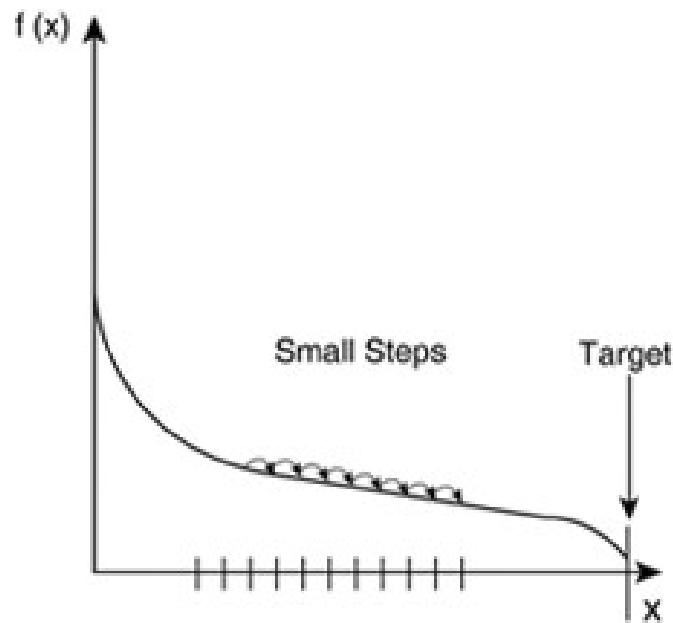
- To find a *minimum* move *exactly opposite the direction of the gradient*

$$x^{k+1} = x^k - \eta^k \nabla_x f(x^k)^T$$

- Many solutions to choosing step size η^k

1. Fixed step size

- Fixed step size
 - Use fixed value for η^k

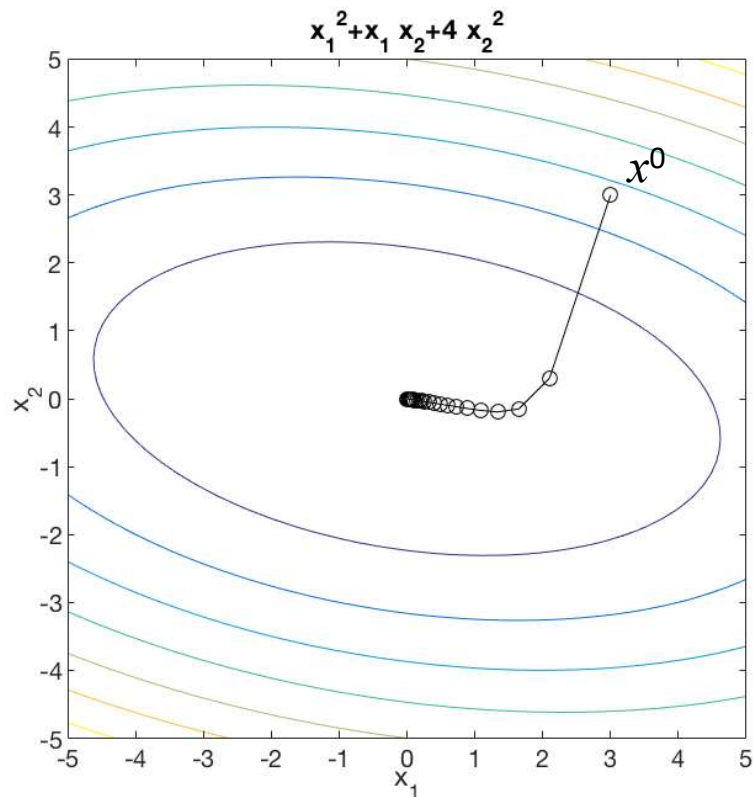


Influence of step size example (constant step size)

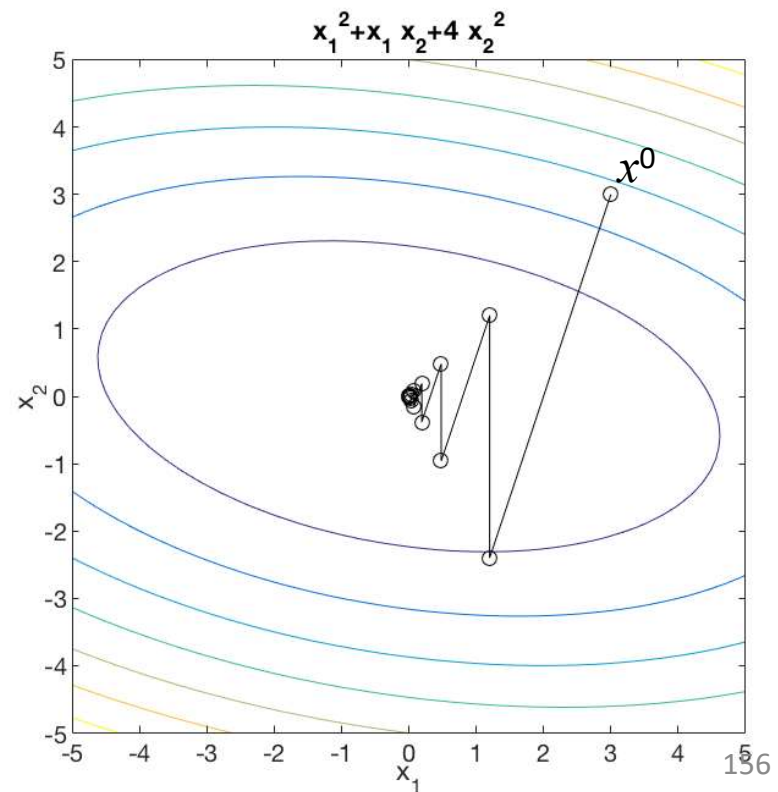
$$f(x_1, x_2) = (x_1)^2 + x_1 x_2 + 4(x_2)^2$$

$$x^{initial} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

$$\eta = 0.1$$



$$\eta = 0.2$$



What is the optimal step size?

- Step size is critical for fast optimization
- Will revisit this topic later
- For now, simply assume a potentially-iteration-dependent step size

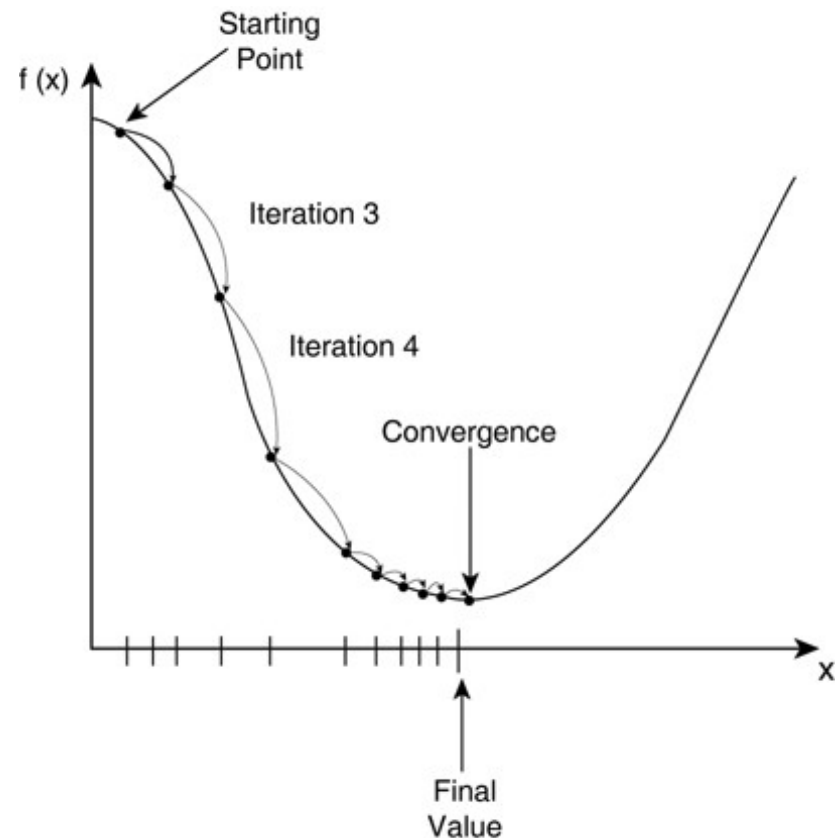
Gradient descent convergence criteria

- The gradient descent algorithm converges when one of the following criteria is satisfied

$$|f(x^{k+1}) - f(x^k)| < \varepsilon_1$$

- Or

$$\|\nabla_x f(x^k)\| < \varepsilon_2$$



Overall Gradient Descent Algorithm

- Initialize:
 - x^0
 - $k = 0$
- do
 - $x^{k+1} = x^k - \eta^k \nabla_x f(x^k)^T$
 - $k = k + 1$
- while $|f(x^{k+1}) - f(x^k)| > \varepsilon$

Next up

- Gradient descent to train neural networks
- A.K.A. Back propagation