Neural Networks
Learning the network: Part 1

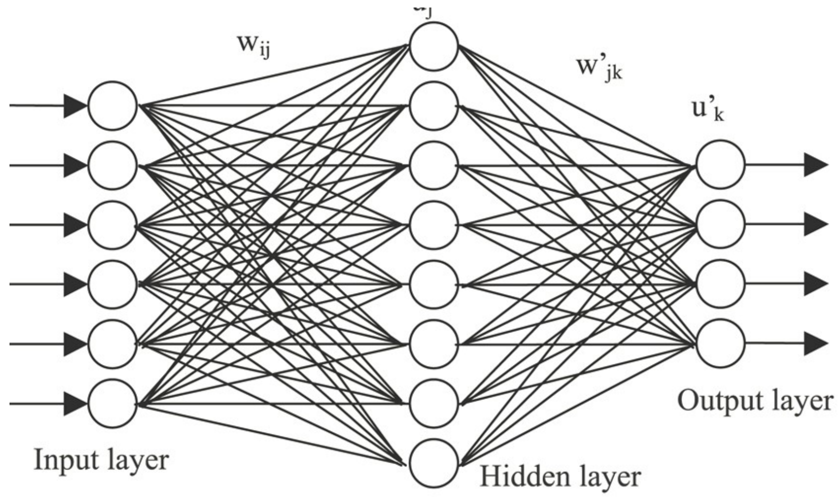
11-785, Spring 2021

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

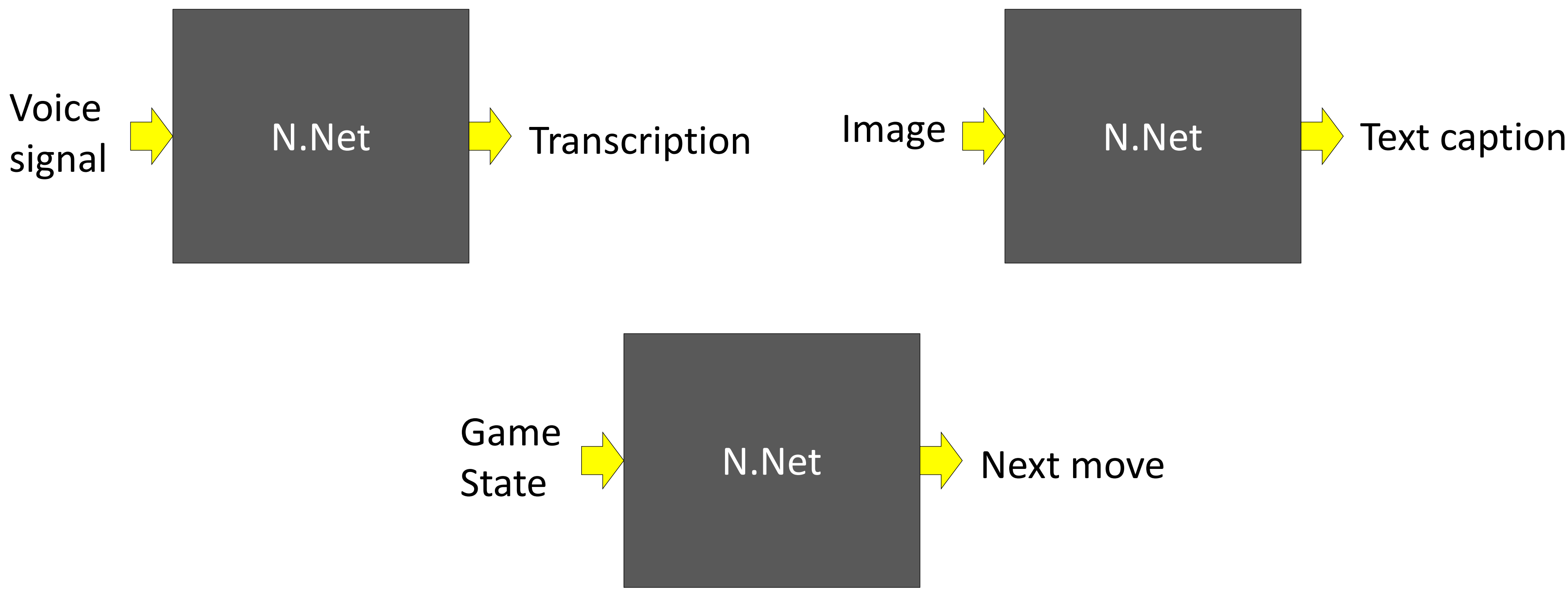
Recap



- **Neural networks are universal function approximators**
  - Can model any Boolean function
  - Can model any classification boundary
  - Can model any continuous valued function

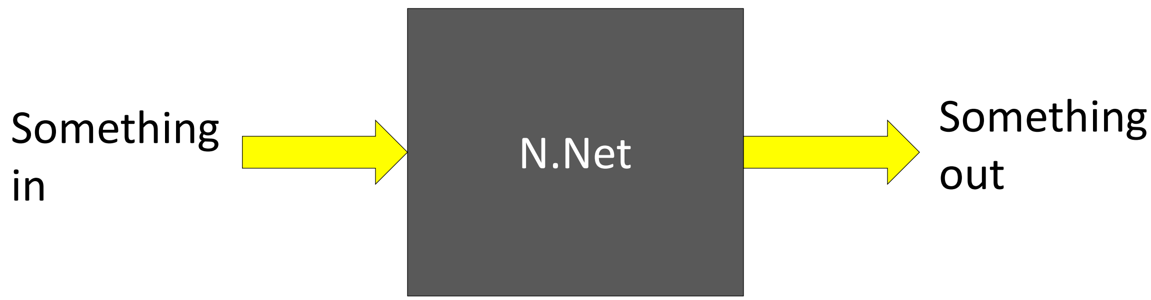
- *Provided the network satisfies minimal architecture constraints*
  - Networks with fewer than the required number of parameters can be very poor approximators

These boxes are functions



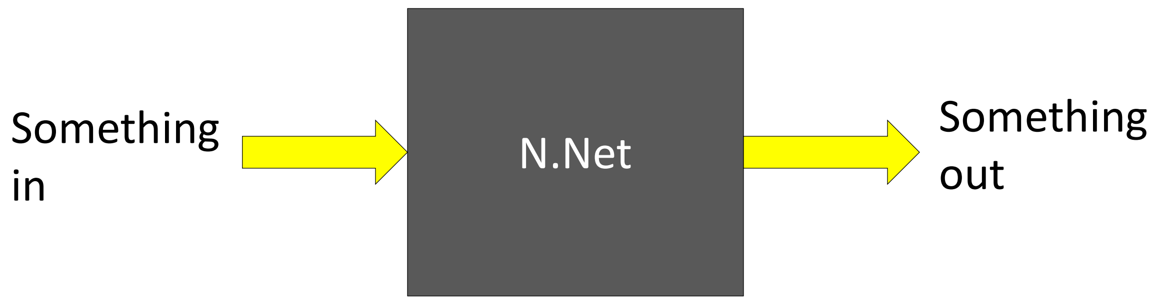
- Take an input
- Produce an output
- Can be modeled by a neural network!

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

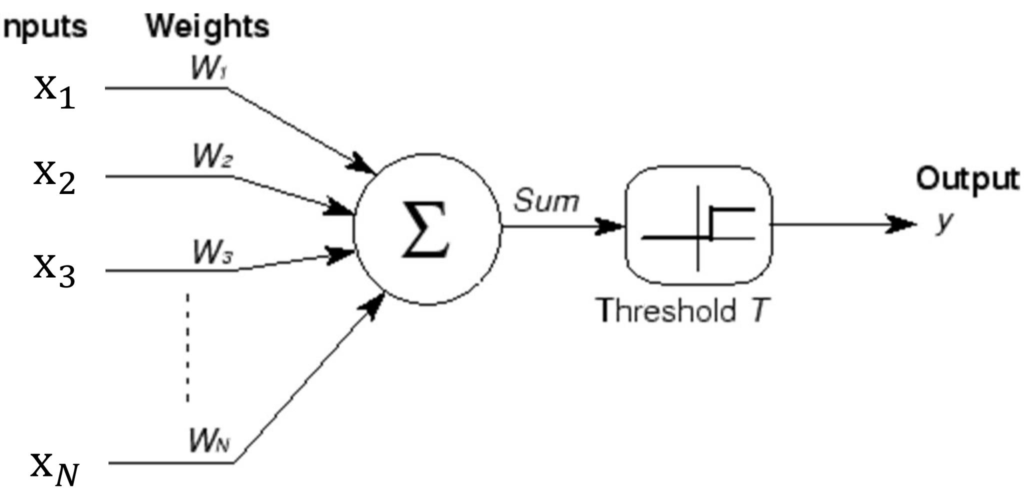


- Preliminaries:
  - How do we represent the input?
  - How do we represent the output?

A bit later in the program

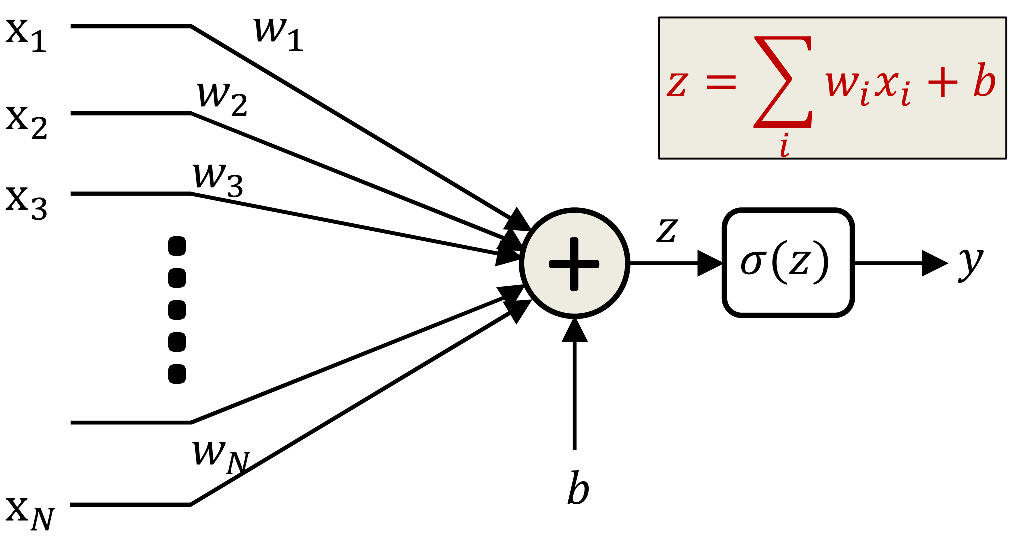
- *How do we compose the network that performs the requisite function?*

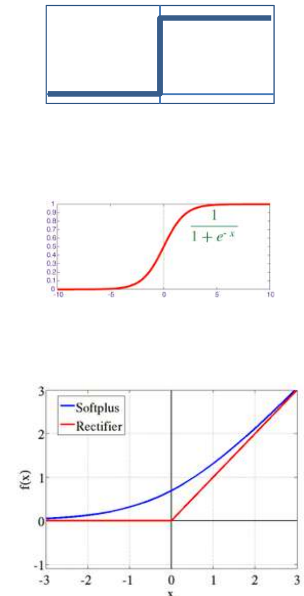
The original perceptron



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

Preliminaries: The units in the network – the perceptron

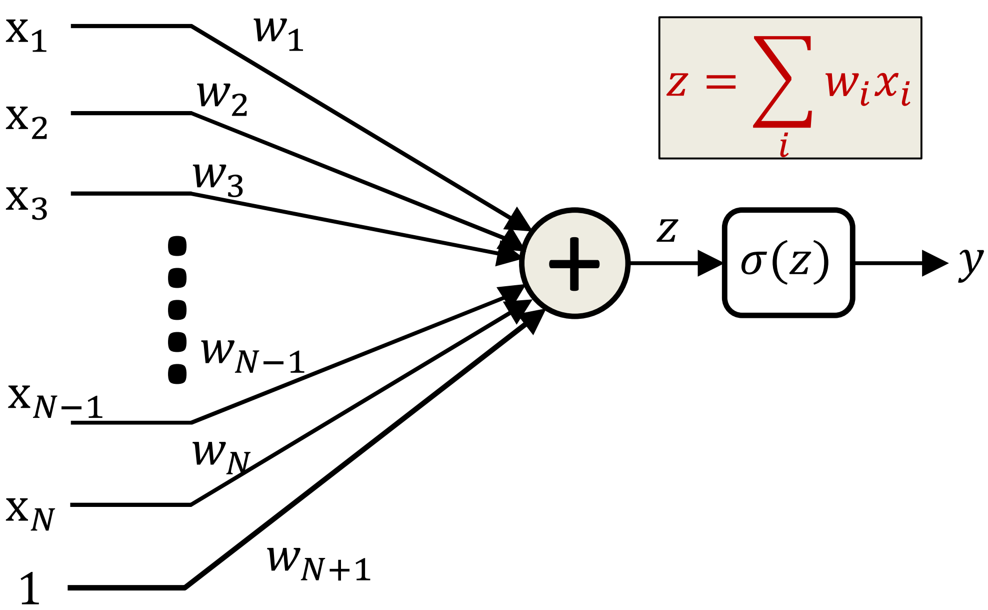


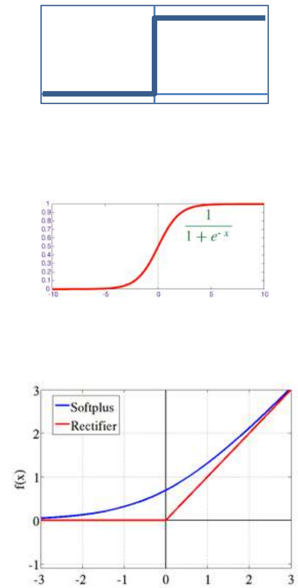


Activation functions $\sigma(z)$

- Perceptron
  - General setting, inputs are real valued
  - A *bias* b representing a threshold to trigger the perceptron
  - Activation functions are not necessarily threshold functions
- The parameters of the perceptron (which determine how it behaves) are its weights and bias

Preliminaries: Redrawing the neuron

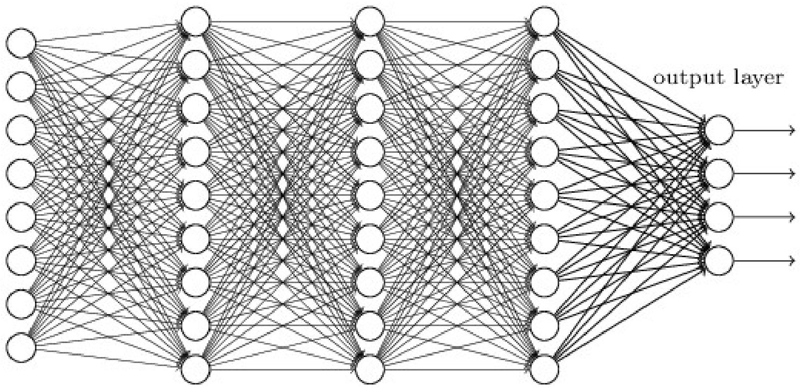


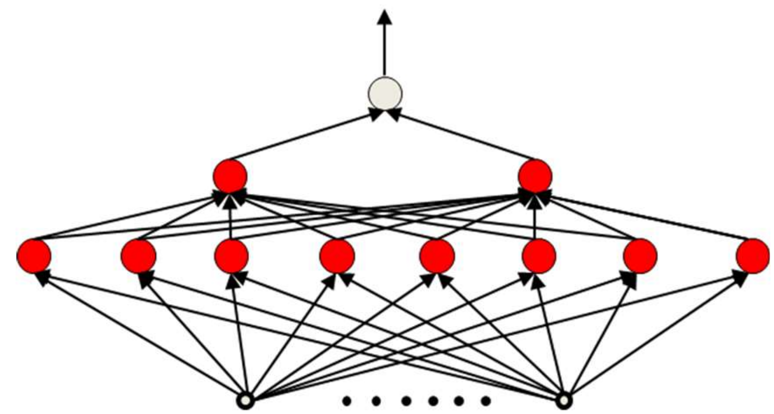


Activation functions $\sigma(z)$

- 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

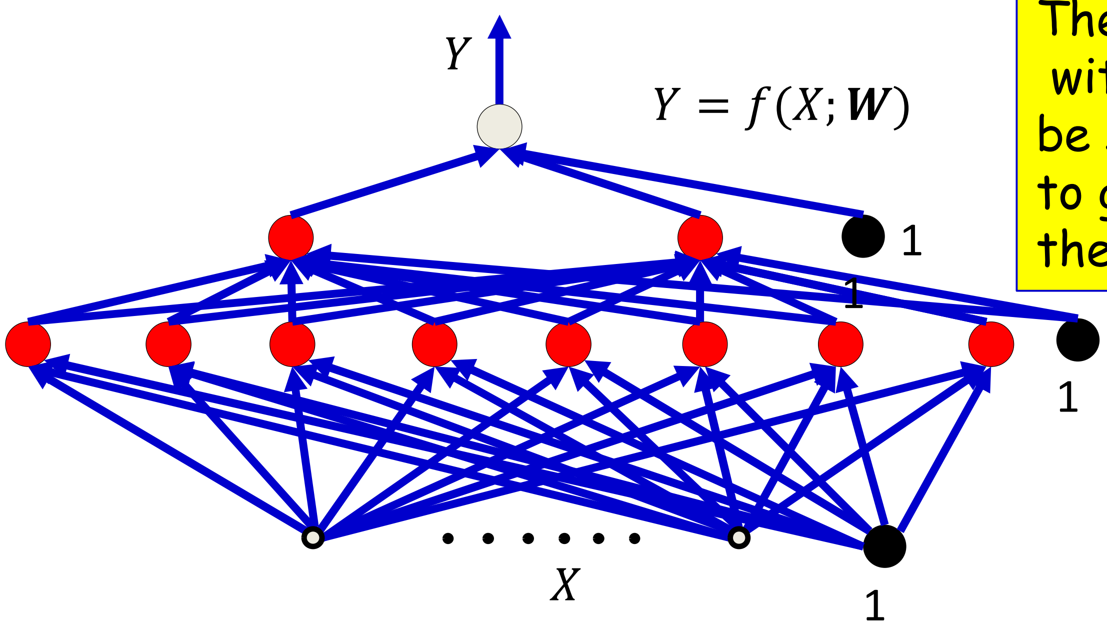
First: the structure of the network





- 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

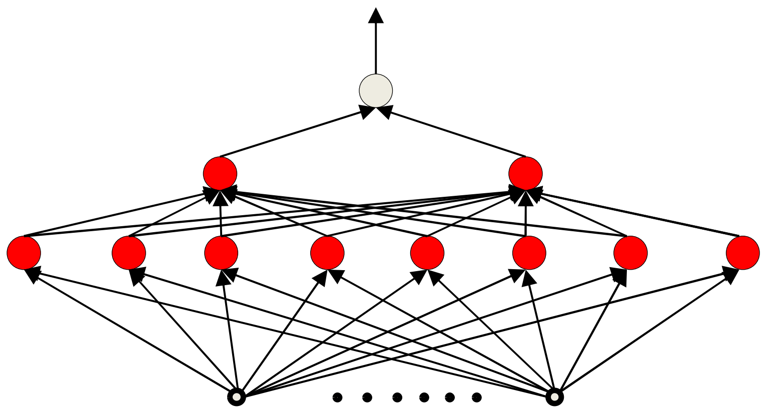


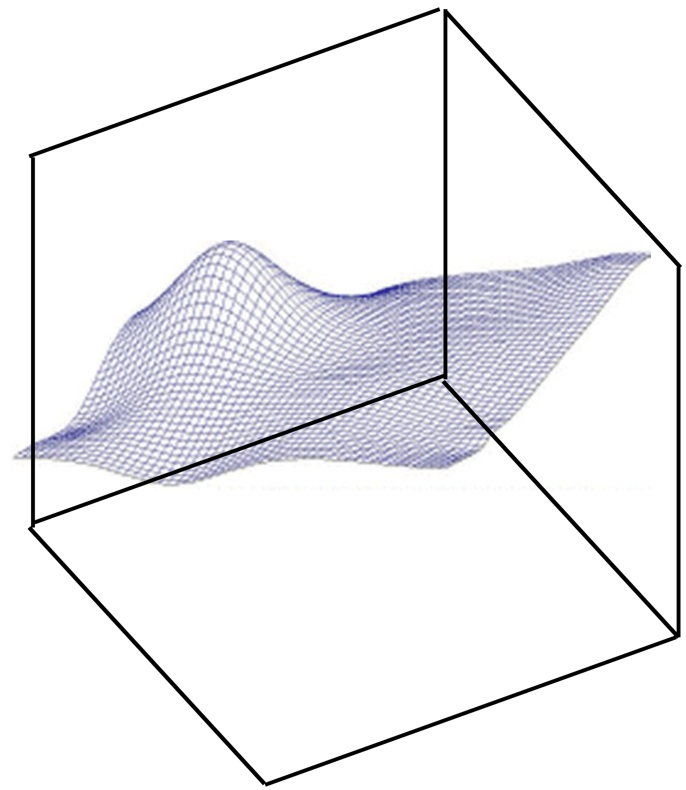
The network is a function f() with parameters W which must be set to the appropriate values to get the desired behavior from the net

- **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..

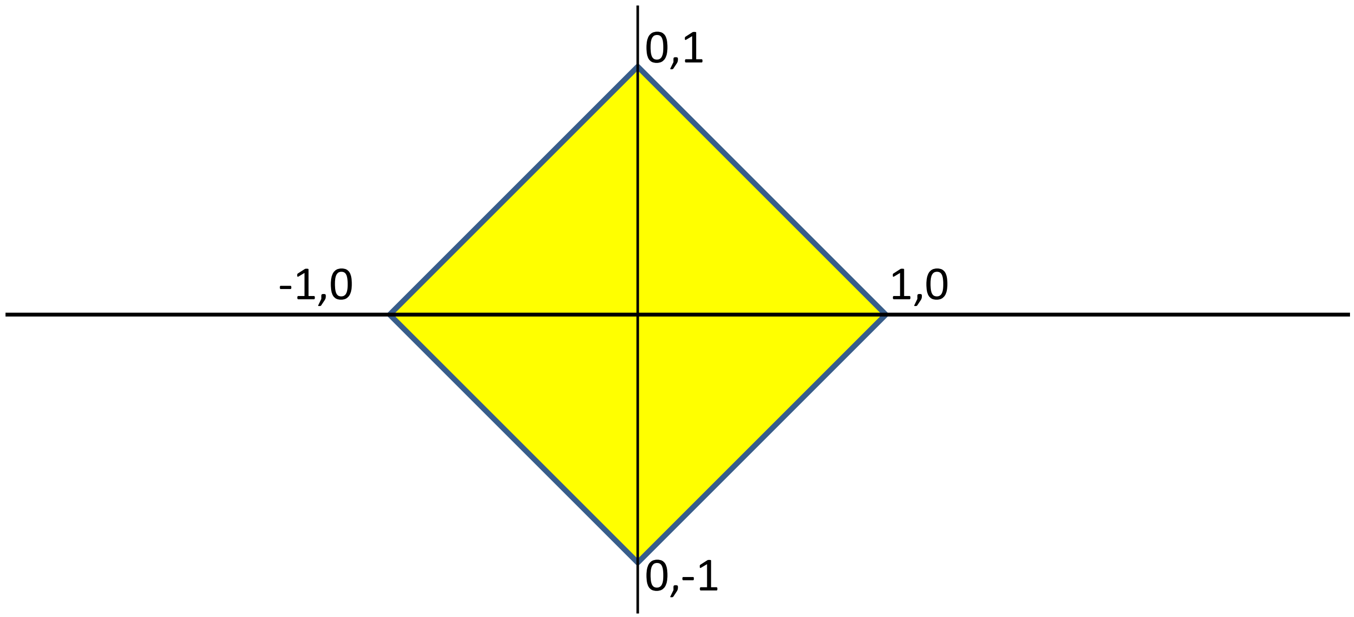
The MLP *can* represent anything





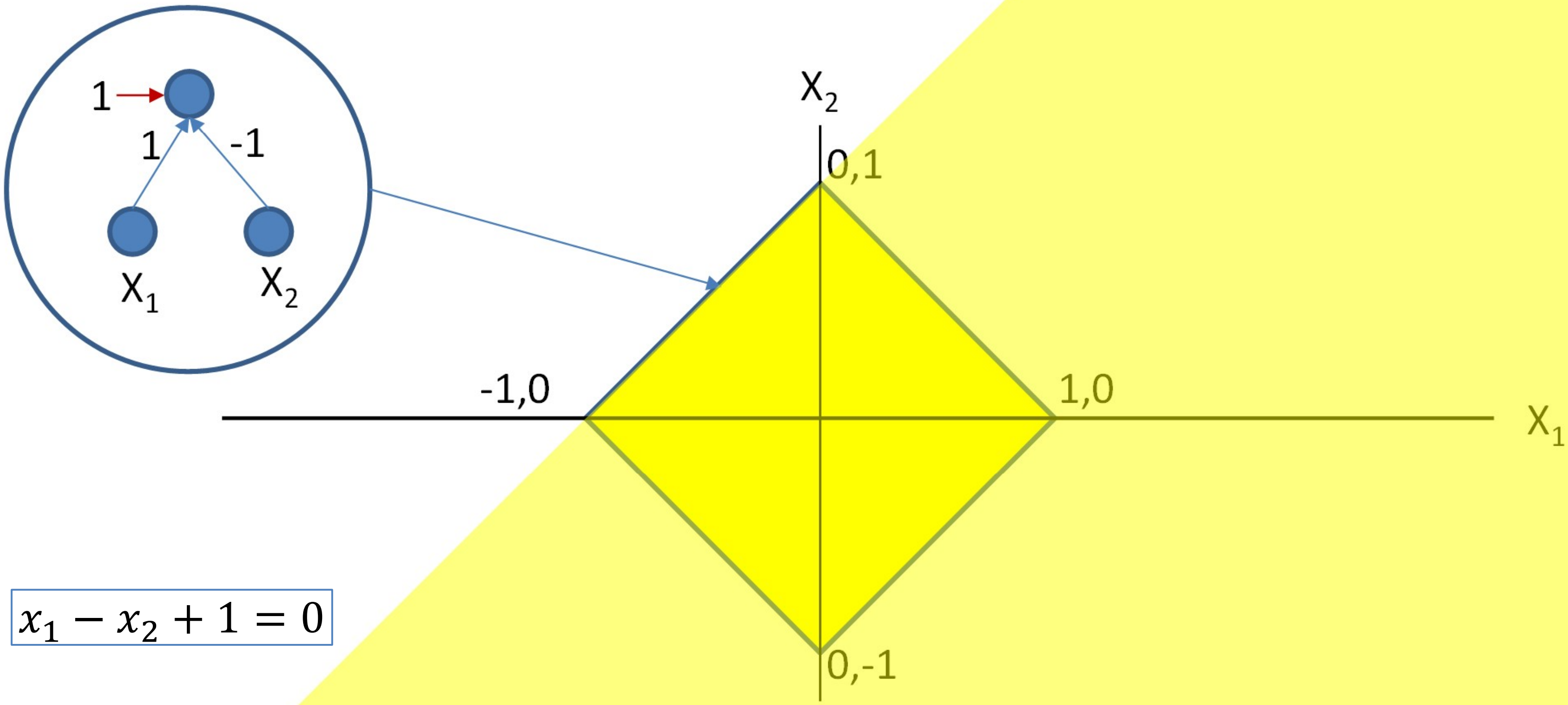
- The MLP *can be constructed* to represent anything
- But *how* do we construct it?

Option 1: Construct by hand



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

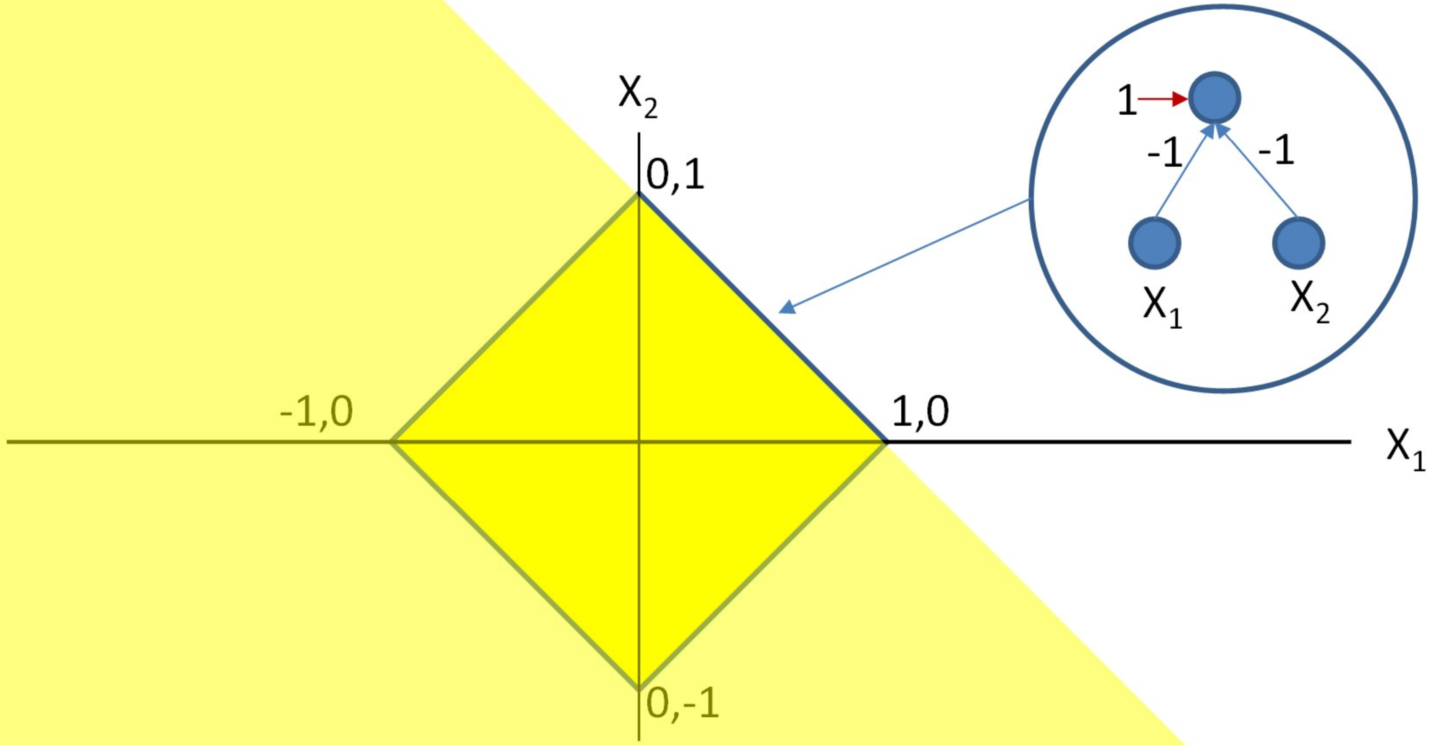
Option 1: Construct by hand



$$\boxed{x_1 - x_2 + 1 = 0}$$

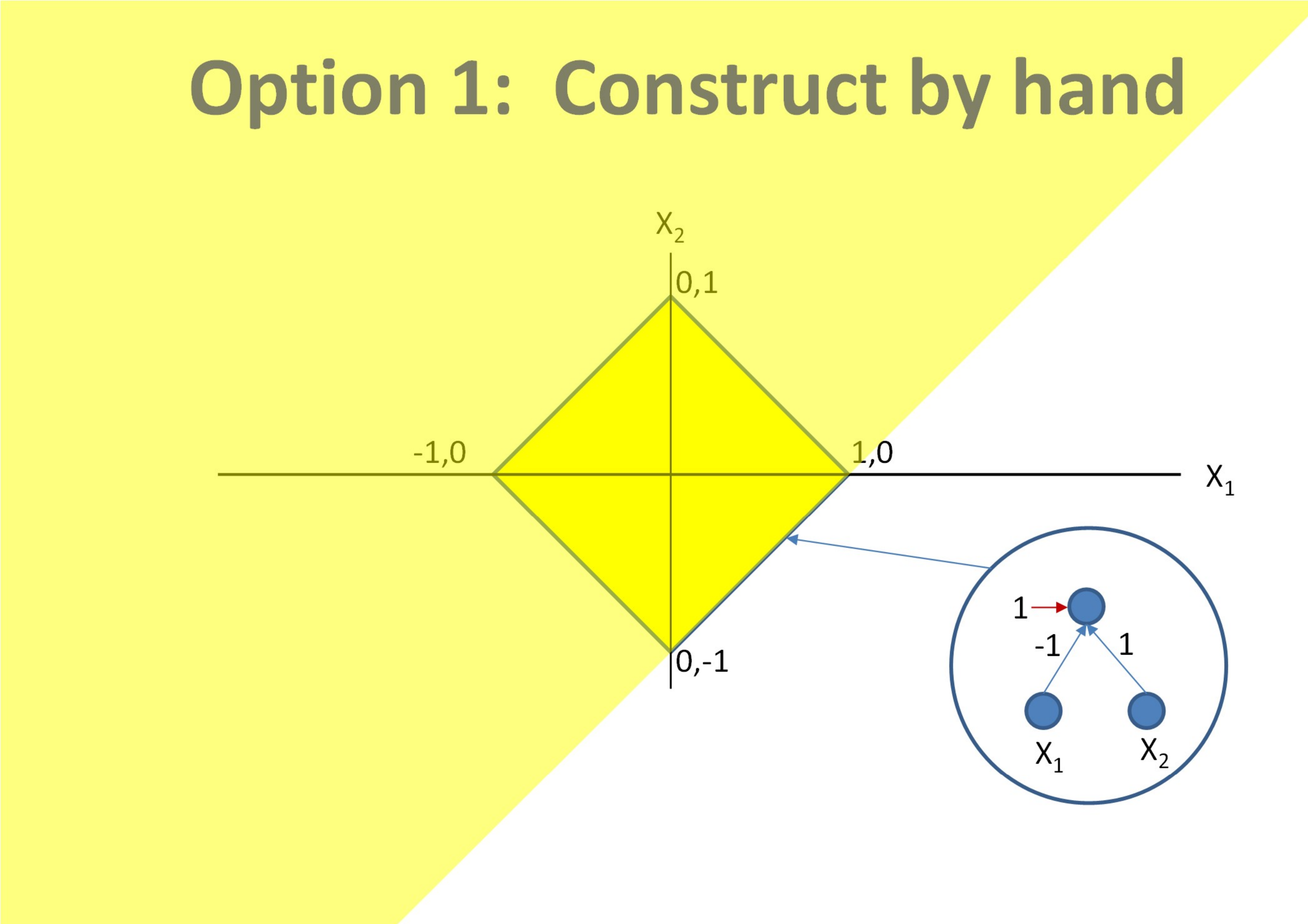
Assuming simple perceptrons:
output = 1 if $\sum_i w_i x_i + b_i \geq 0, else\ 0$

Option 1: Construct by hand



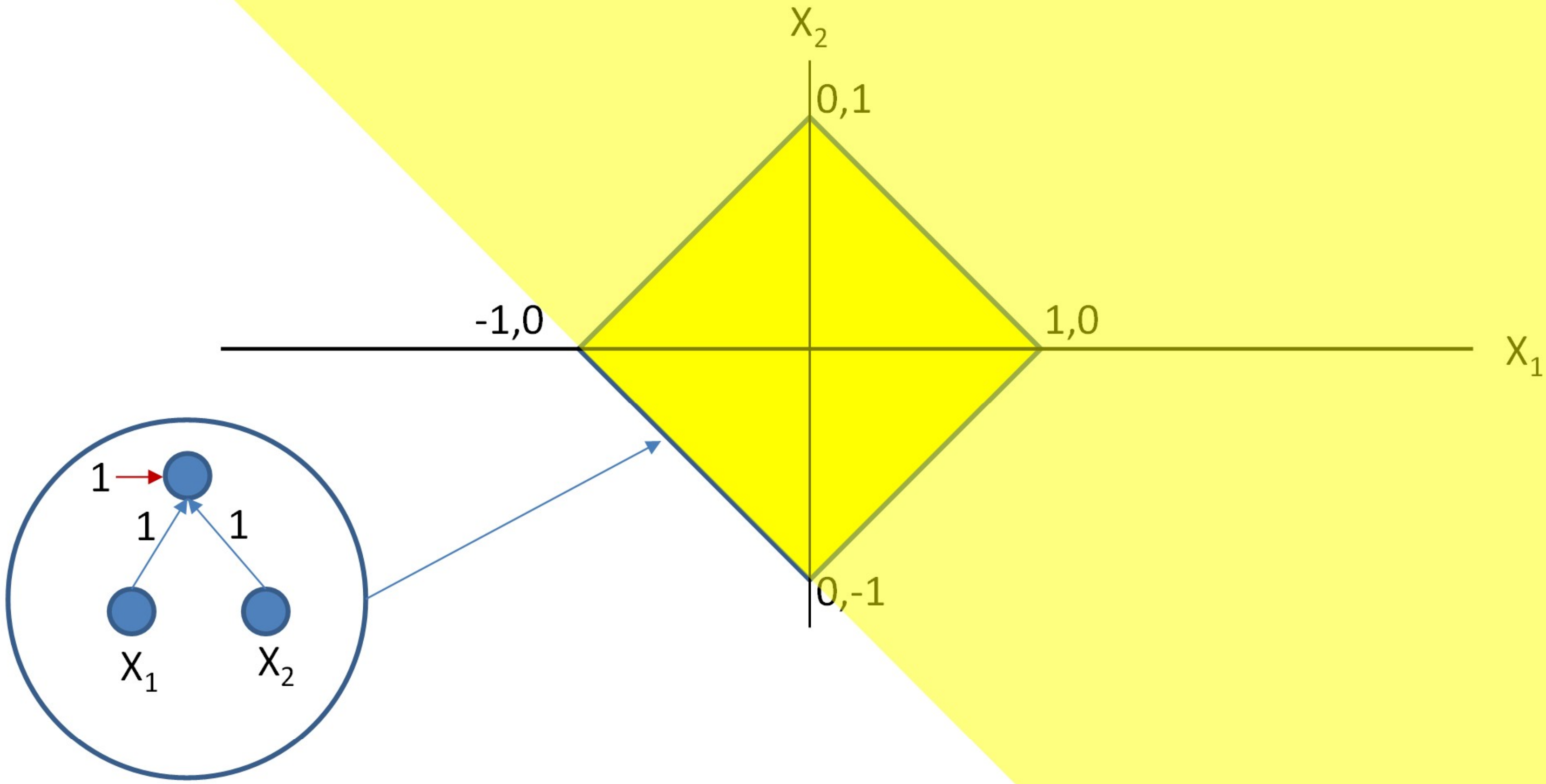
Assuming simple perceptrons:
output = 1 if $\sum_i w_i x_i + b_i \geq 0, else\ 0$

Option 1: Construct by hand



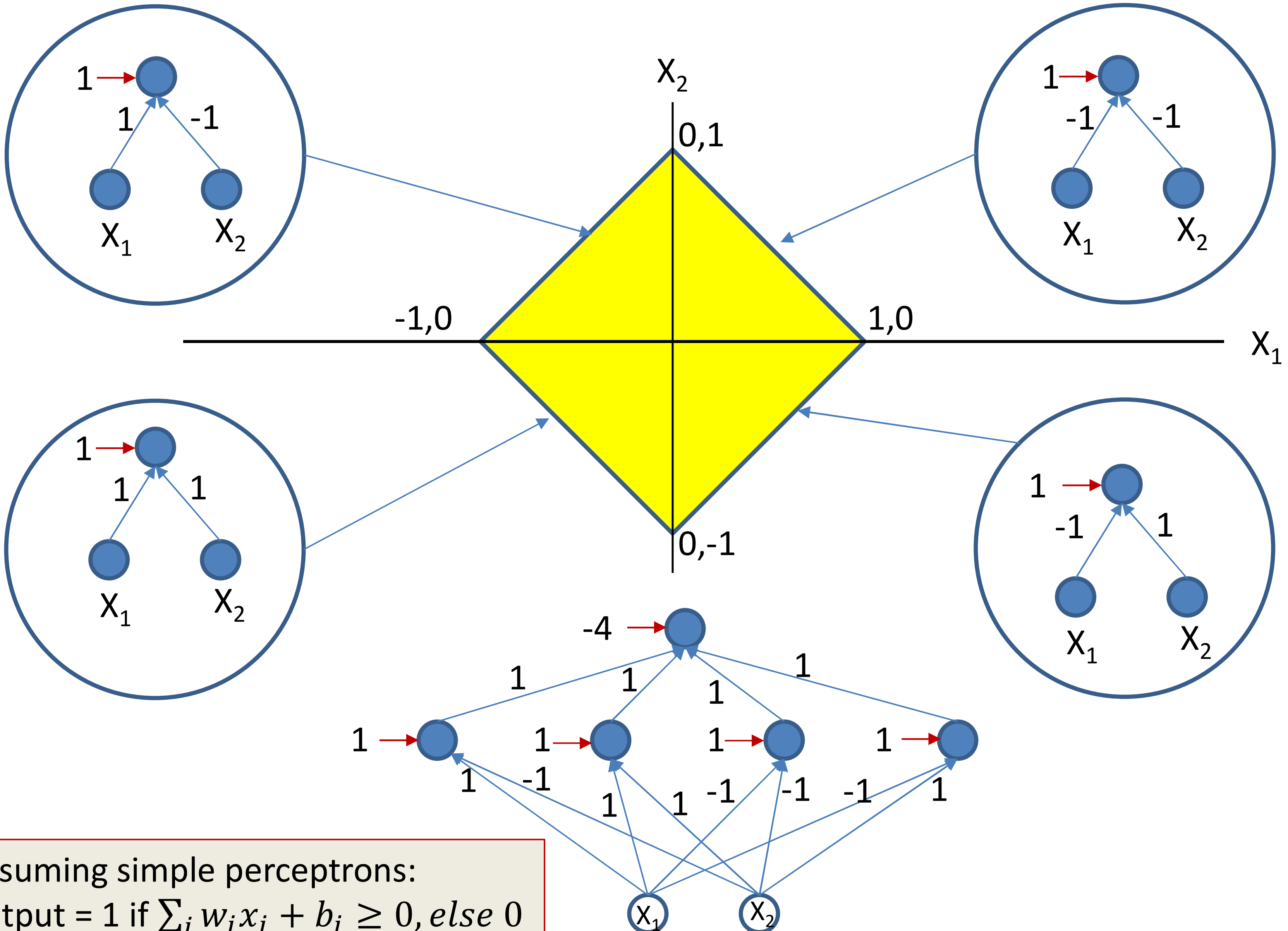
Assuming simple perceptrons:
output = 1 if $\sum_i w_i x_i + b_i \geq 0, else\ 0$

Option 1: Construct by hand



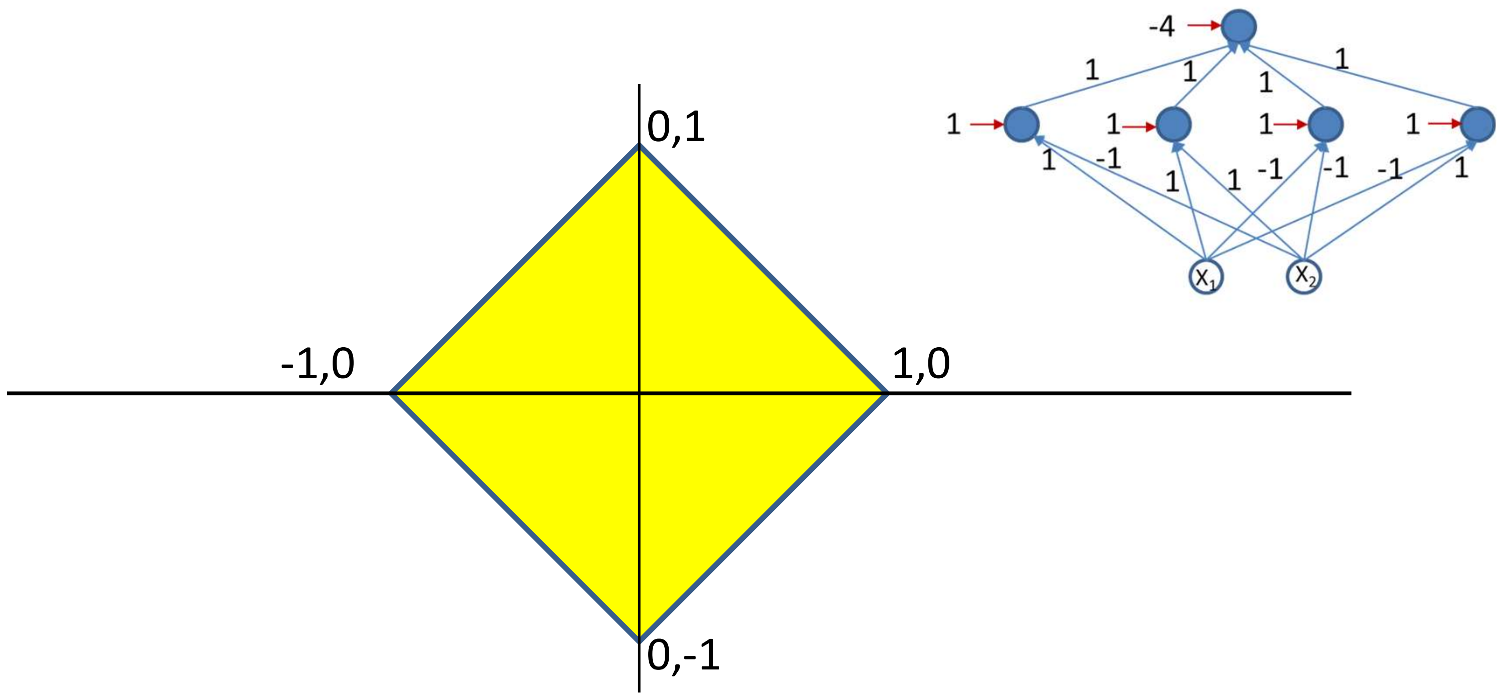
Assuming simple perceptrons:
output = 1 if $\sum_i w_i x_i + b_i \geq 0, else\ 0$

Option 1: Construct by hand



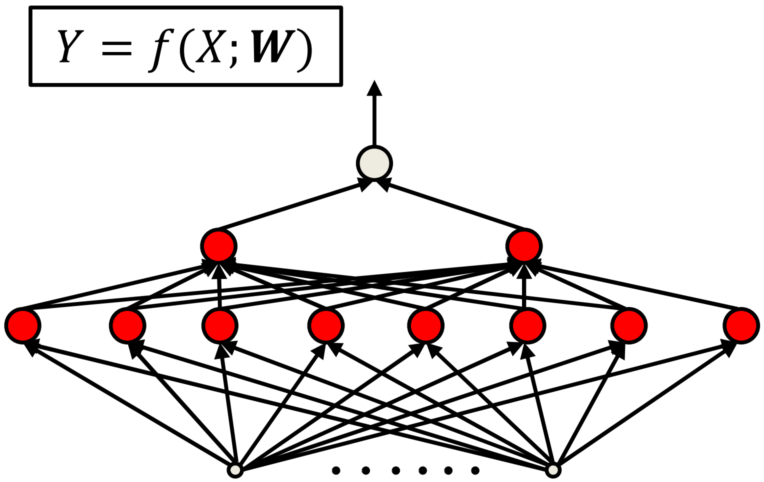
Assuming simple perceptrons:
output = 1 if $\sum_i w_i x_i + b_i \geq 0, else\ 0$

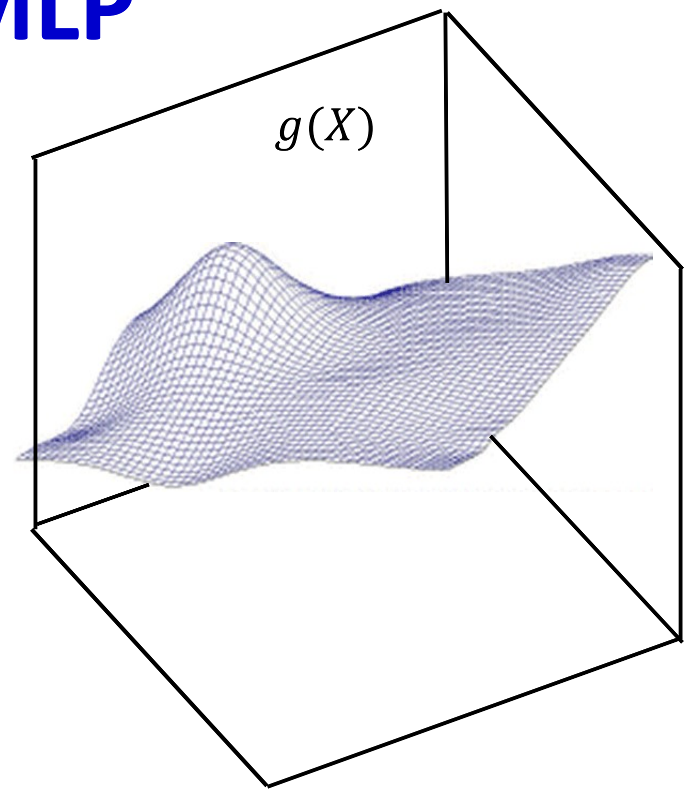
Option 1: Construct by hand



- 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..

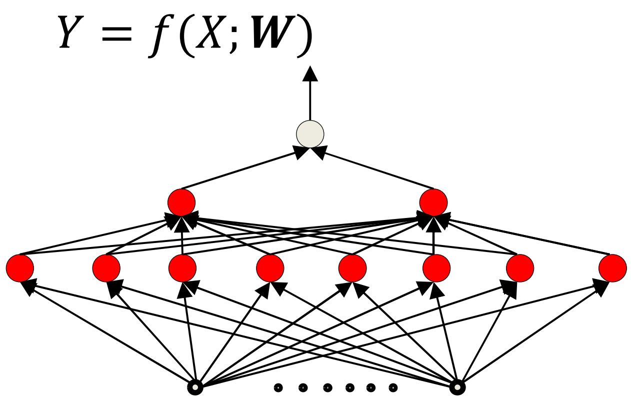
Option 2: Automatic estimation of an MLP

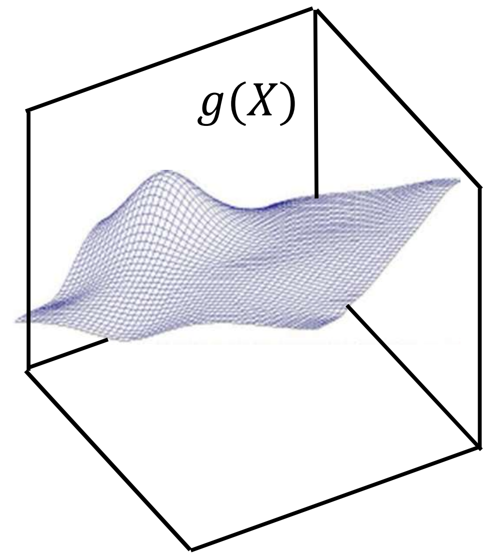




- More generally, *given* the function $g(X)$ to model, we can *derive* the parameters of the network to model it, through computation

How to learn a network?



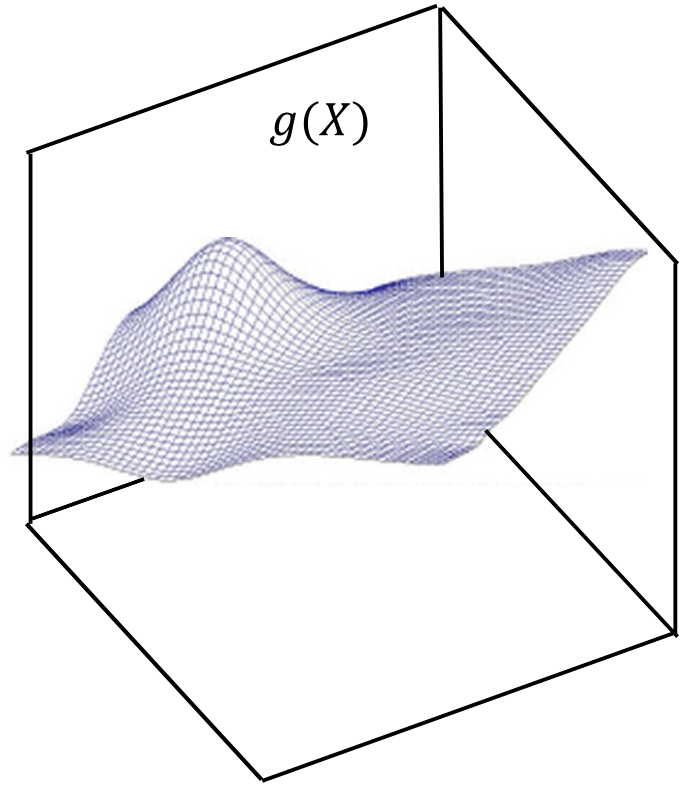


- When $f(X; \boldsymbol{W})$ has the capacity to exactly represent $g(X)$

$$\widehat{\boldsymbol{W}} = \underset{W}{\operatorname{argmin}} \int_X div\big(f(X; W), g(X)\big) dX$$

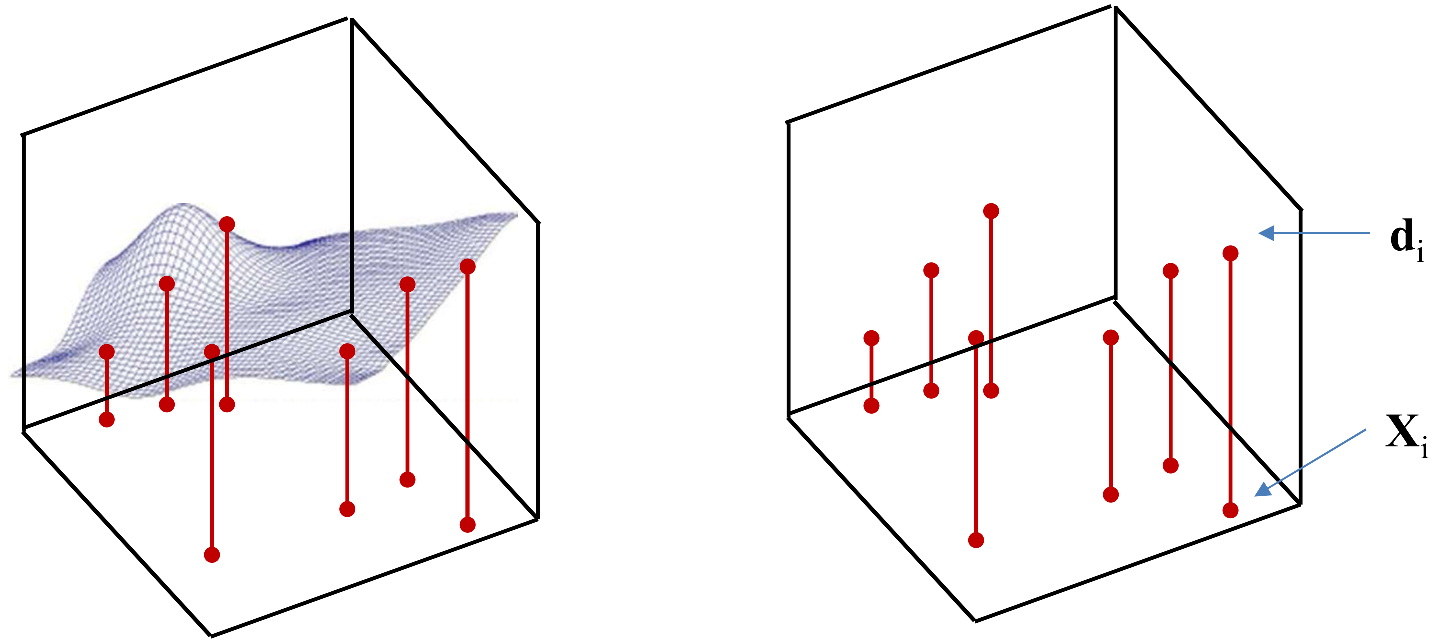
- $div()$ is a *divergence* function that goes to zero when $f(X; W) = g(X)$

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**

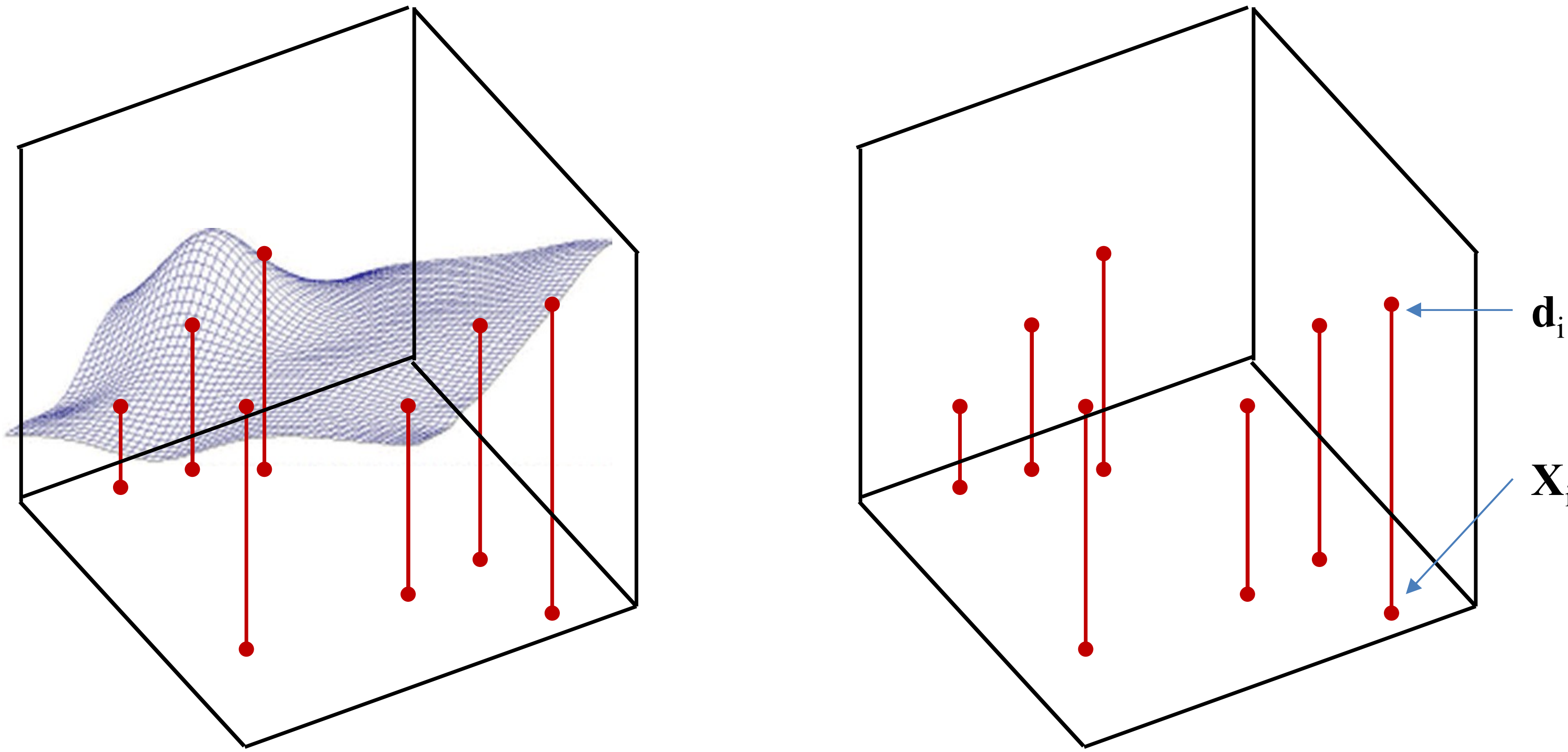
Sampling the function



- *Sample* $g(X)$
  – Basically, get input-output pairs for a number of samples of input X_i
    - Many samples (X_i, d_i), where $d_i = g(X_i) + noise$

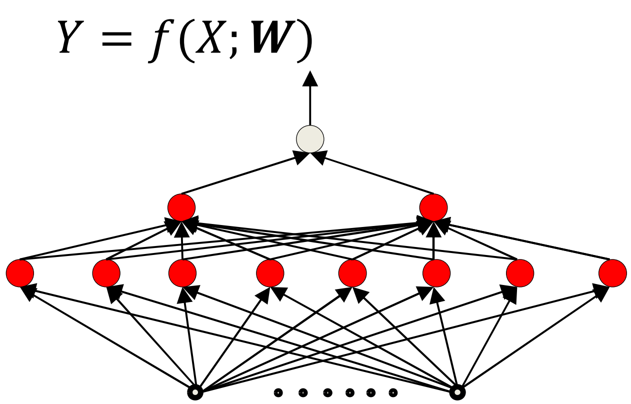
- Very easy to do in most problems: just gather training data
  – E.g. set of images and their class labels
  – E.g. speech recordings and their transcription

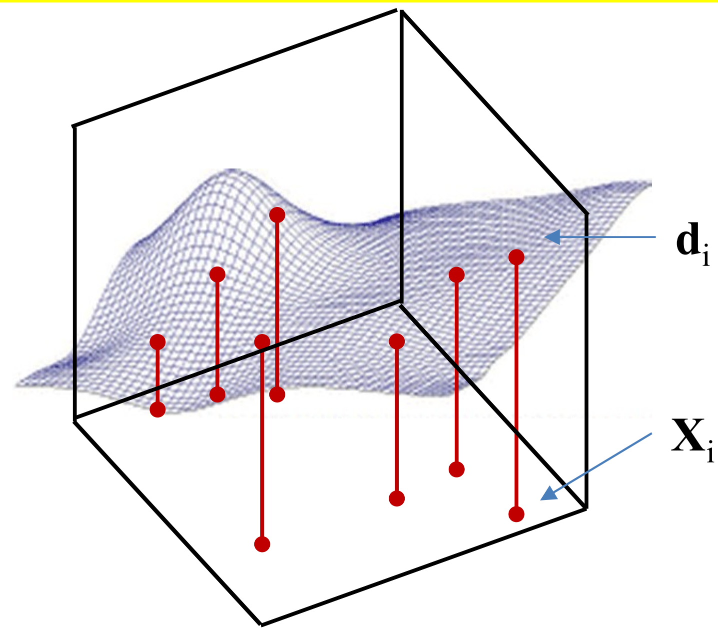
Drawing samples



- 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

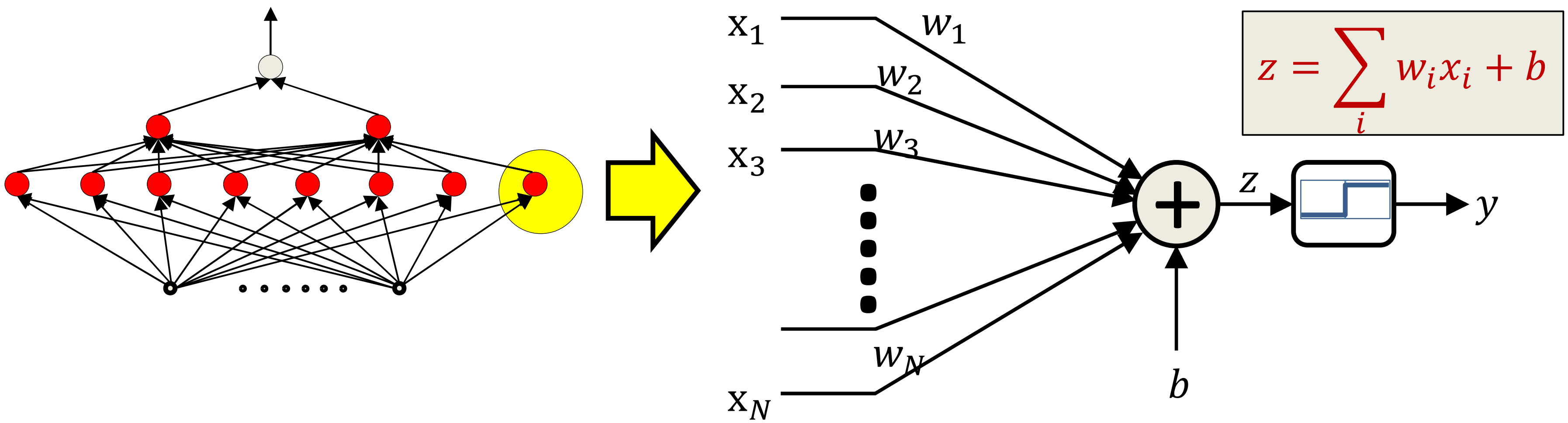
Let’s begin with a simple task

- Learning a *classifier*
  - Simpler than regressions

- This was among the earliest problems addressed using MLPs

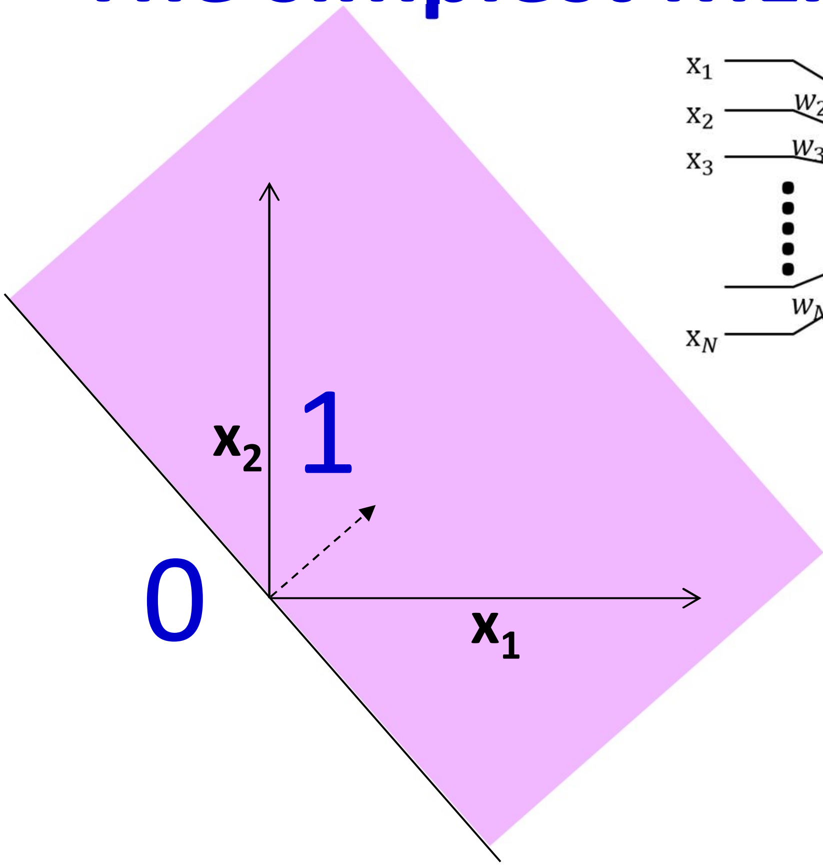
- Specifically, consider *binary* classification
  - Generalizes to multi-class

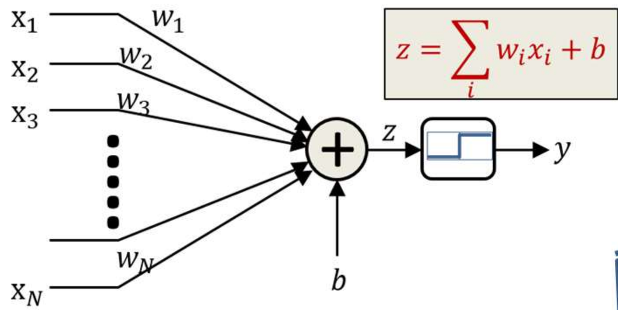
History: The original MLP

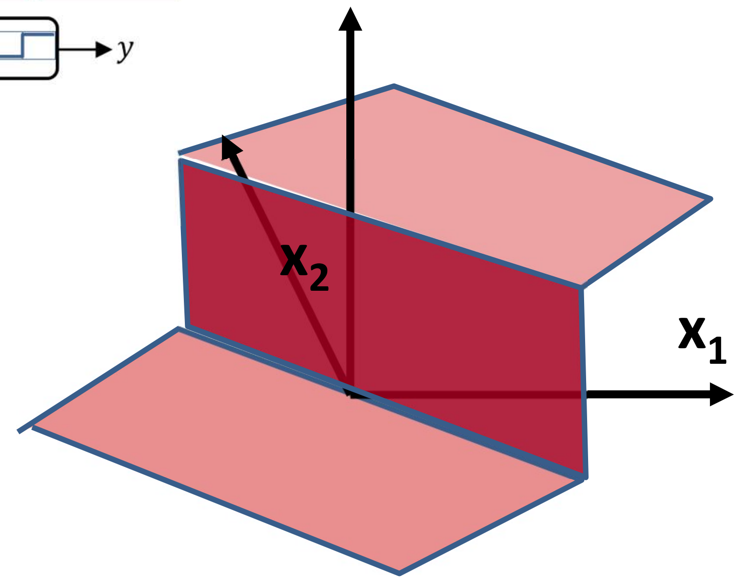


- The original MLP as proposed by Minsky: a network of threshold units
  - But how do you train it?
    - Given only “training” instances of input-output pairs

The simplest MLP: a single perceptron

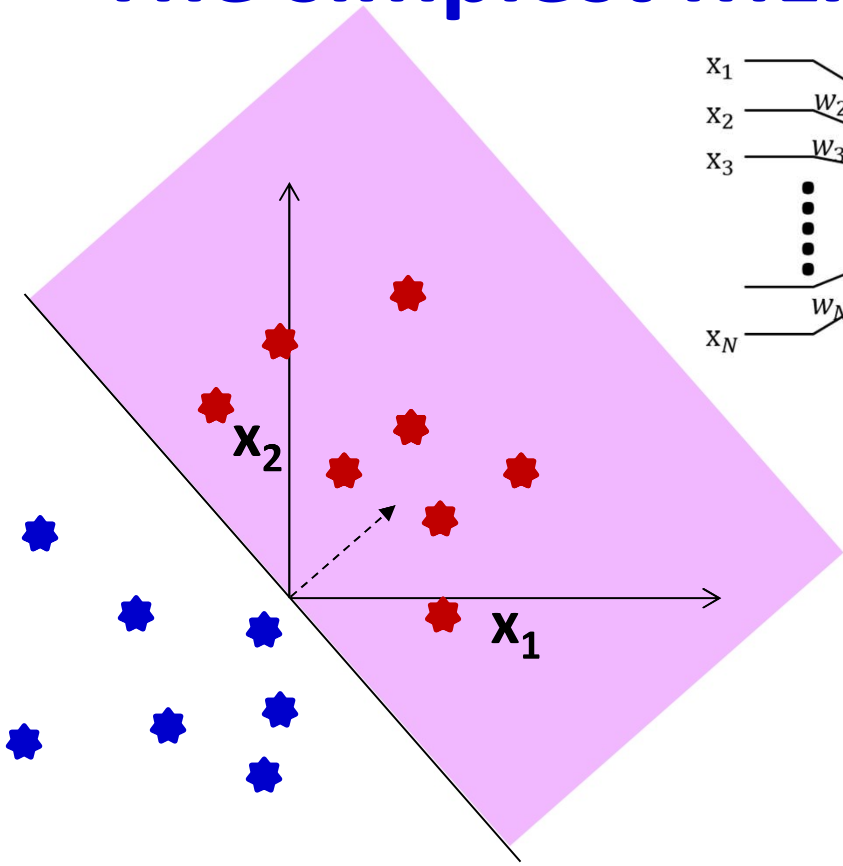


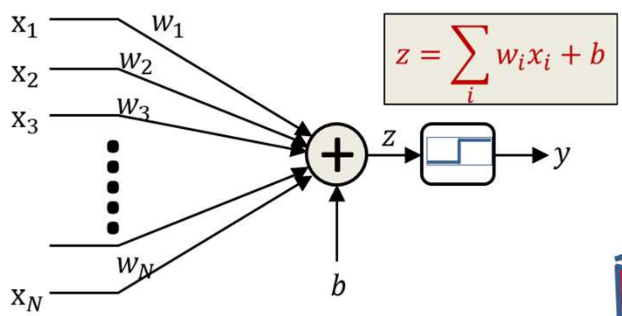


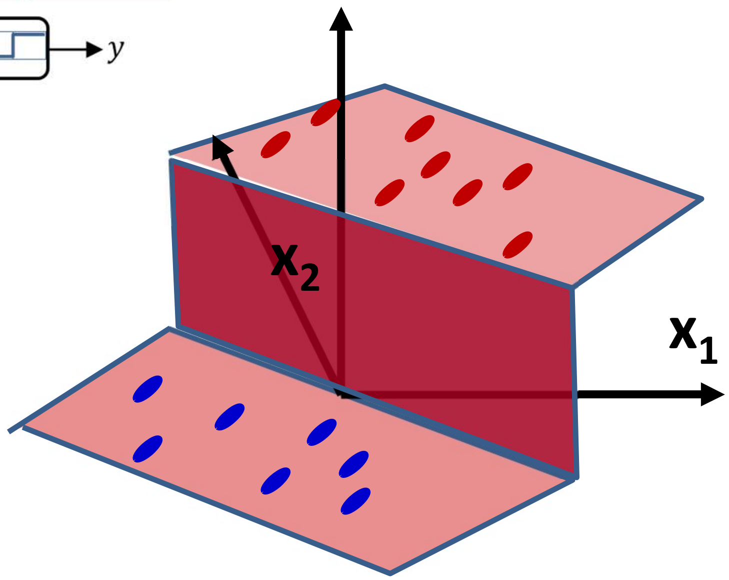


- Learn this function
  - **A step function across a hyperplane**

The simplest MLP: a single perceptron

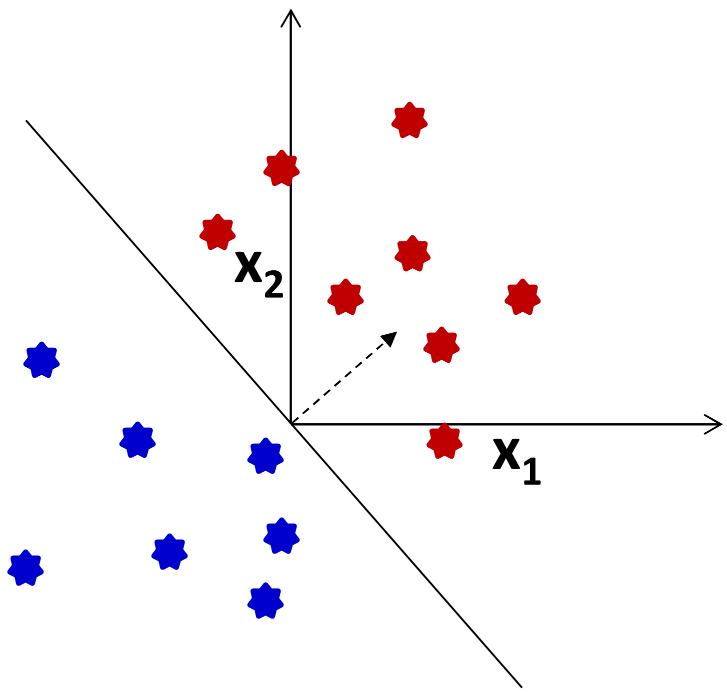


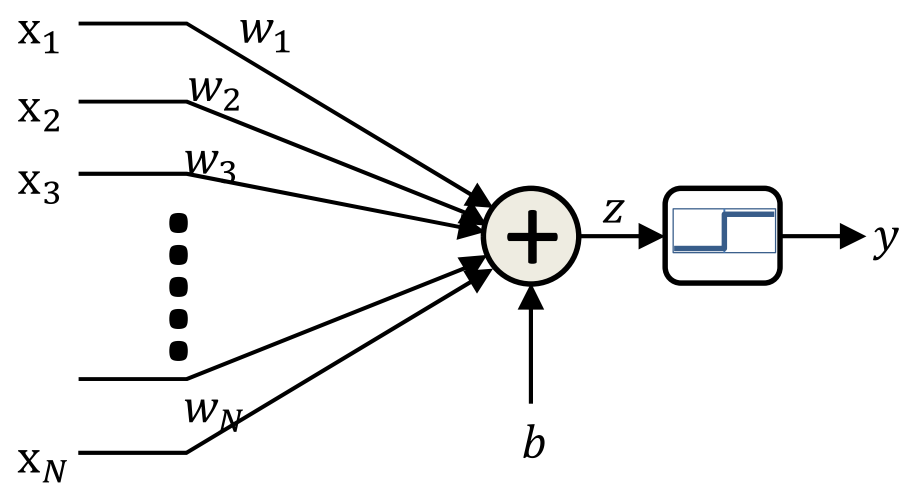




- Learn this function
  - **A step function across a hyperplane**
  - **Given only samples from it**

Learning the perceptron

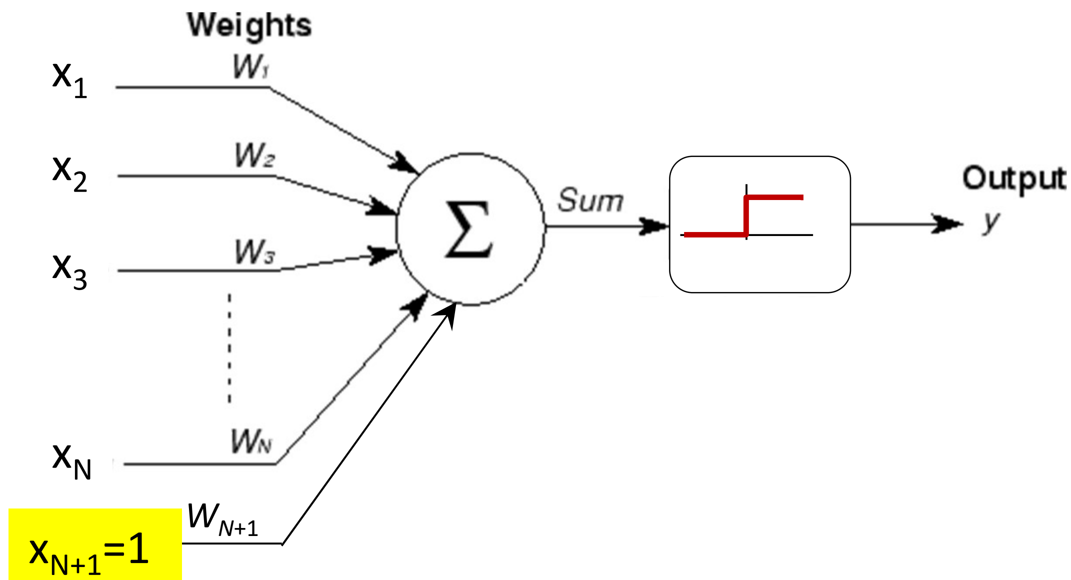




- Given a number of input output pairs, learn the weights and bias
  - $y = \begin{cases} 1 \;\; if \;\; \sum_{i=1}^{N} w_i X_i + b \geq 0 \\ 0 \qquad\qquad\qquad otherwise \end{cases}$
  - Learn $W = [w_1 .. w_N]^T$ and b, given several (X, y) pairs

Boundary: $\sum_{i=1}^{N} w_i X_i + b = 0$

Restating the perceptron



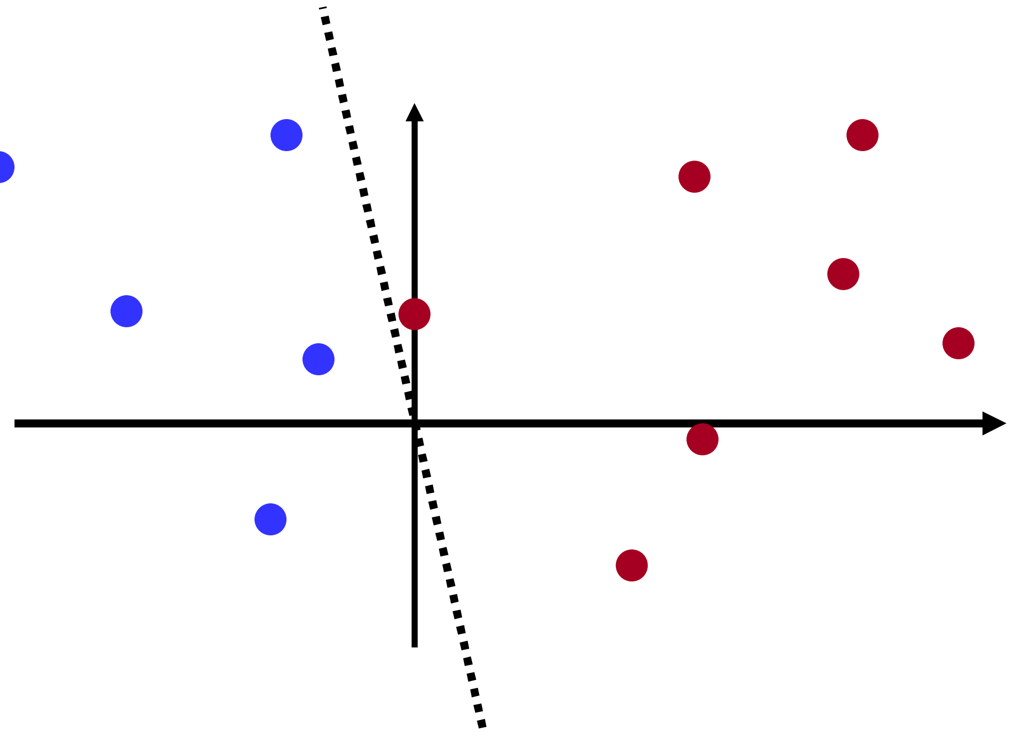
- Restating the perceptron equation by adding another dimension to X

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

where $X_{N+1} = 1$

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

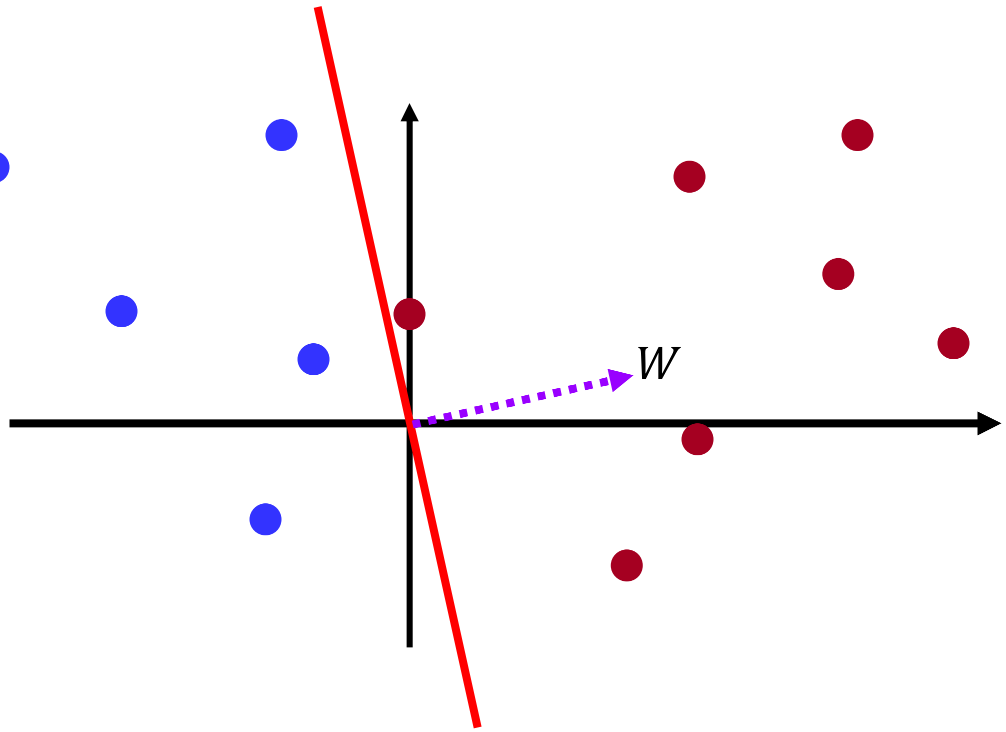
The Perceptron Problem



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

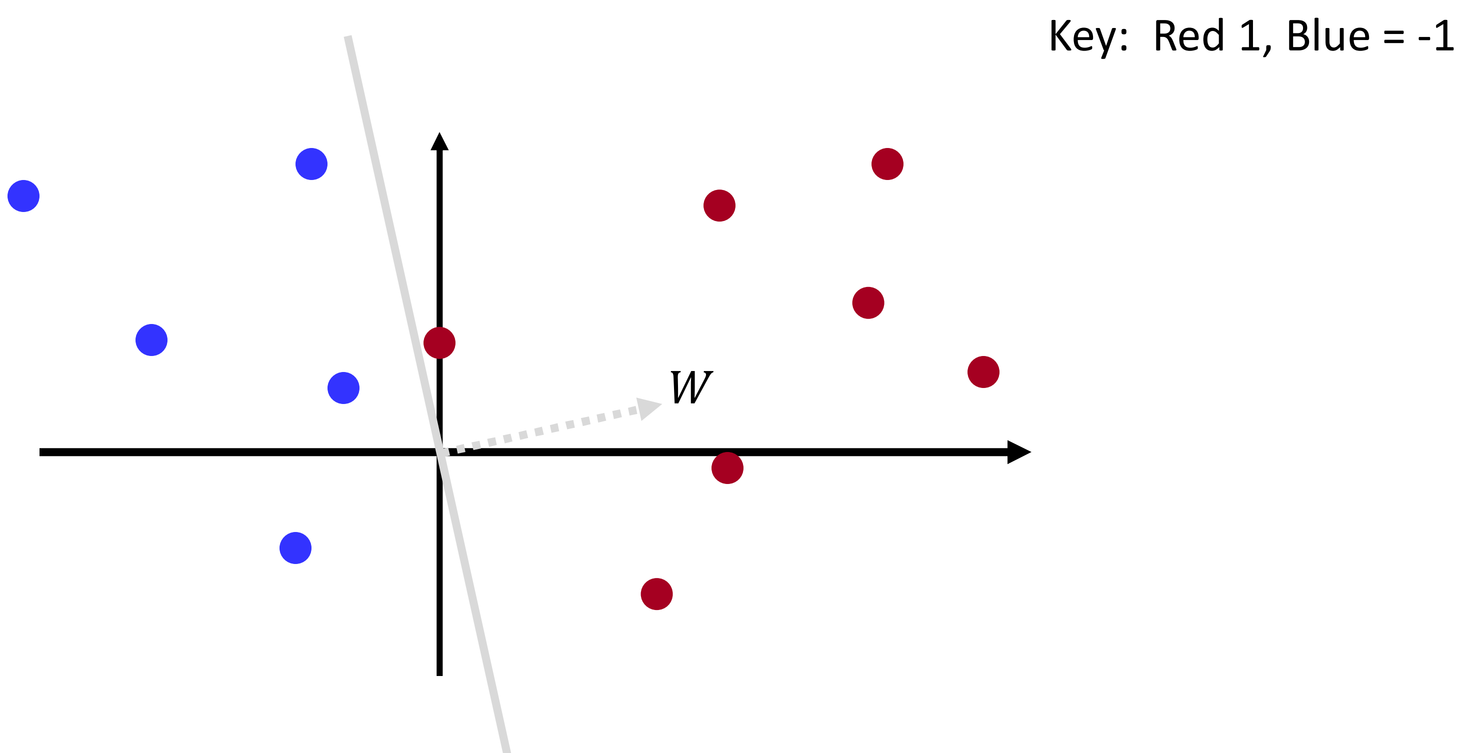
The Perceptron Problem

Key: Red 1, Blue = -1



- Find the hyperplane $\sum_{i=1}^{N+1} w_i X_i = 0$ that perfectly separates the two groups of points
  - Let vector $W = [w_1, w_2, \dots, w_{N+1}]^T$ and vector $X = [x_1, x_2, \dots, x_N, 1]^T$
  - $\sum_{i=1}^{N+1} w_i X_i = W^T X$ is an inner product
  - $W^T X = 0$ is the hyperplane comprising all Xs orthogonal to vector W
    - Learning the perceptron = finding the weight vector W for the separating hyperplane
    - W points in the direction of the positive class

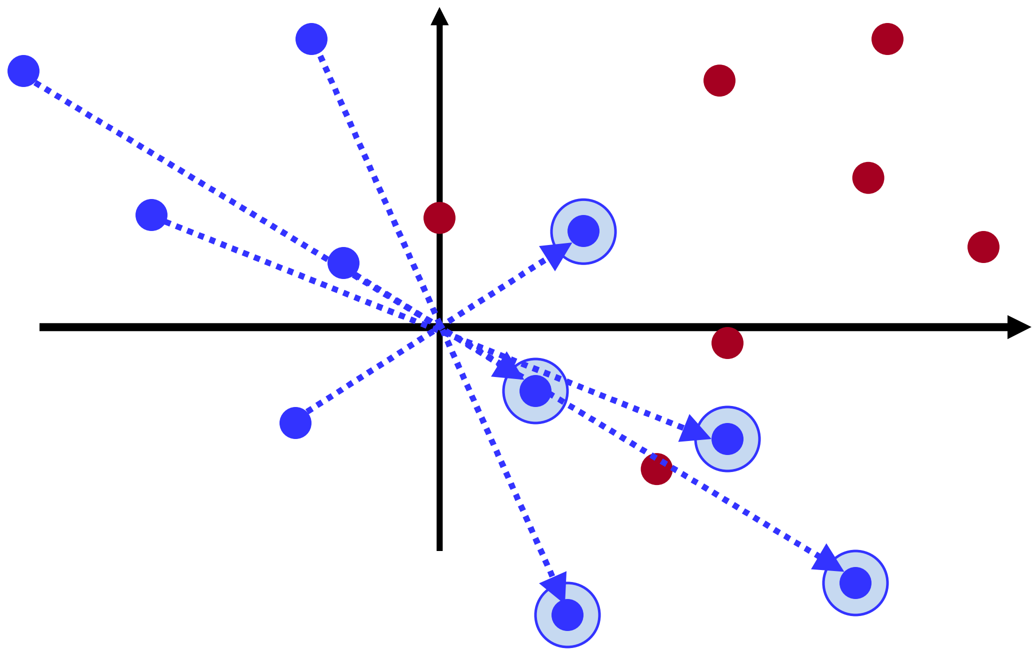
The Perceptron Problem



- Learning the perceptron: Find the weights vector W such that the plane described by $W^T X = 0$ perfectly separates the classes
  - $W^T X$ is positive for all red dots and negative for all blue ones

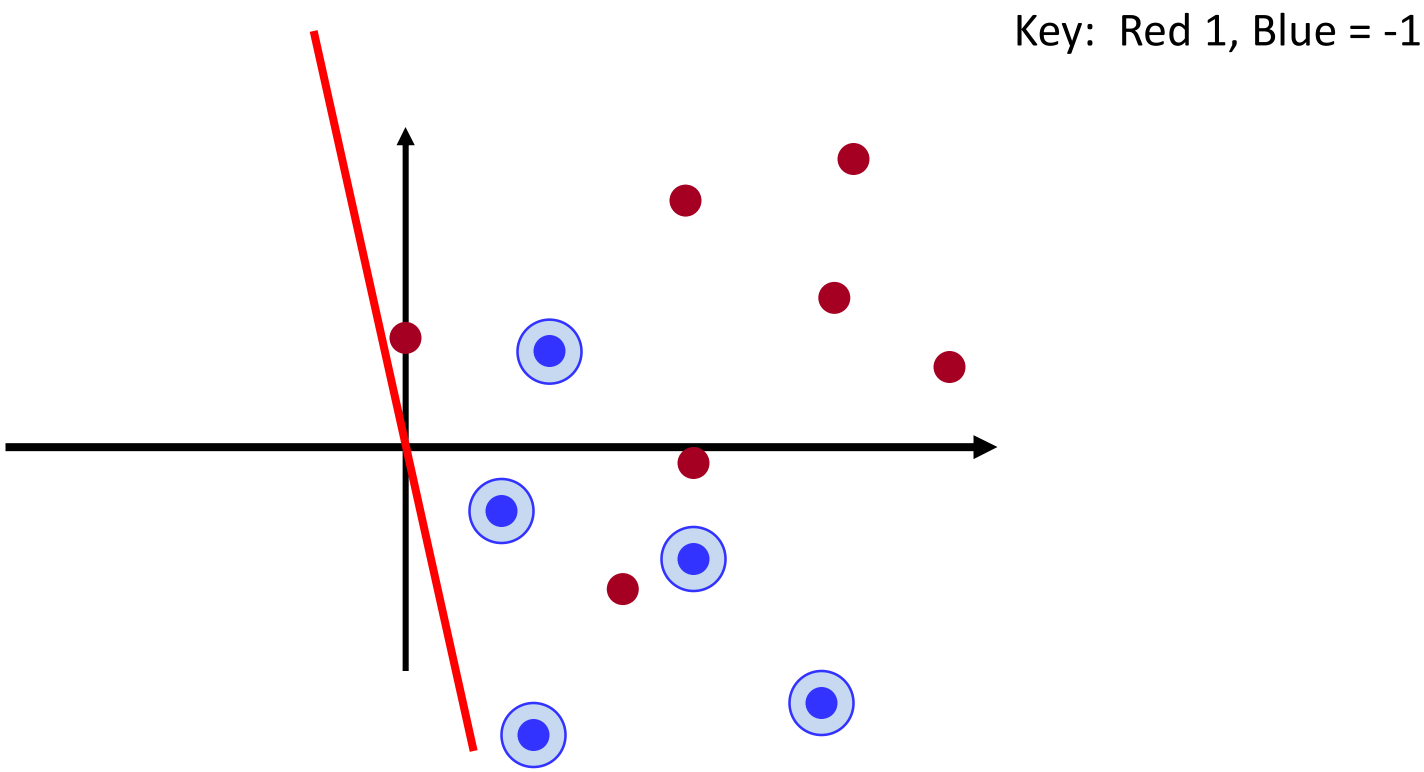
A simple solution

Key: Red 1, Blue = -1



- *Reflect* all the negative instances across the origin
  - Negate every component of vector X
- If we use class $y \in \{+1, -1\}$ notation for the labels (instead of $y \in \{0,1\}$), we can simply write the "reflected" values as $X' = yX$
  - Will retain the features X for the positive class, but reflect/negate them for the negative class

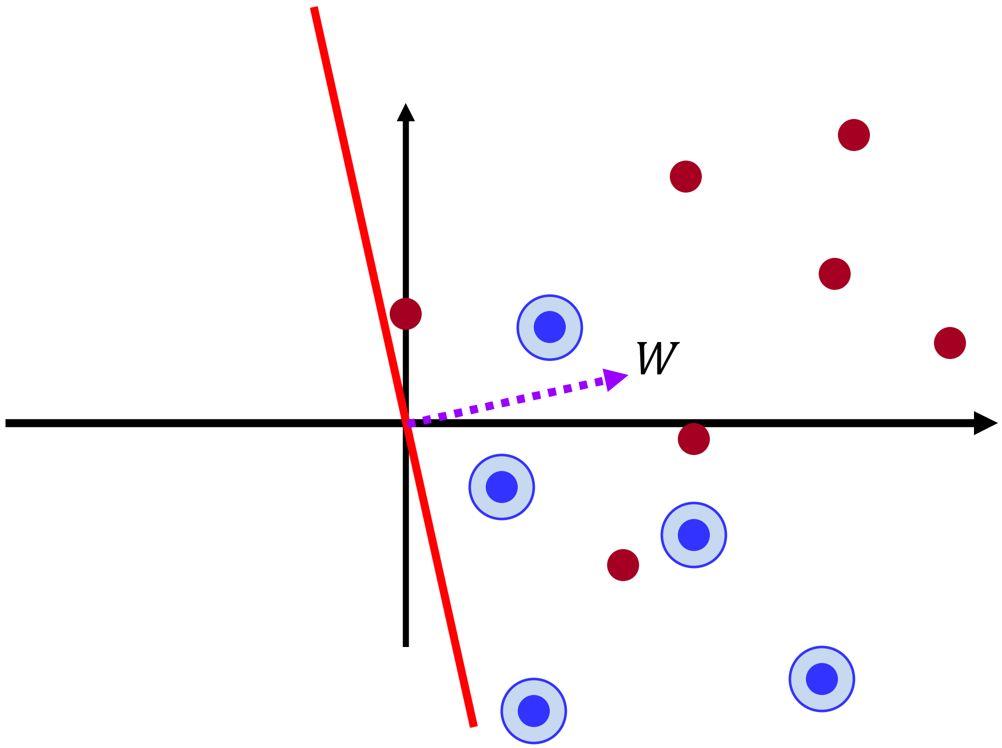
The Perceptron Solution



- Learning the perceptron: Find a plane such that all the modified (X') features lie on one side of the plane
  - Such a plane can always be found if the classes are linearly separable

The Perceptron Solution: Linearly separable case

Key: Red 1, Blue = -1



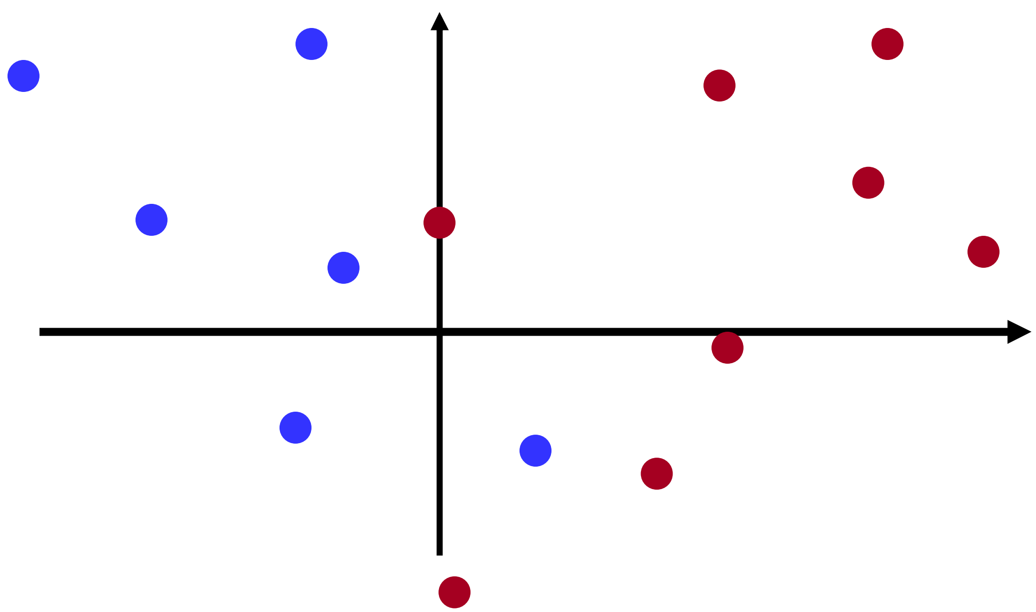
- When classes are linearly separable: a trivial solution

$$W = \frac{1}{N}\sum_i X_i' \qquad = \frac{1}{N}\sum_i y_i X_i$$

- Other solutions are also possible, e.g. max-margin solution

The Perceptron Solution: when classes are not linearly separable

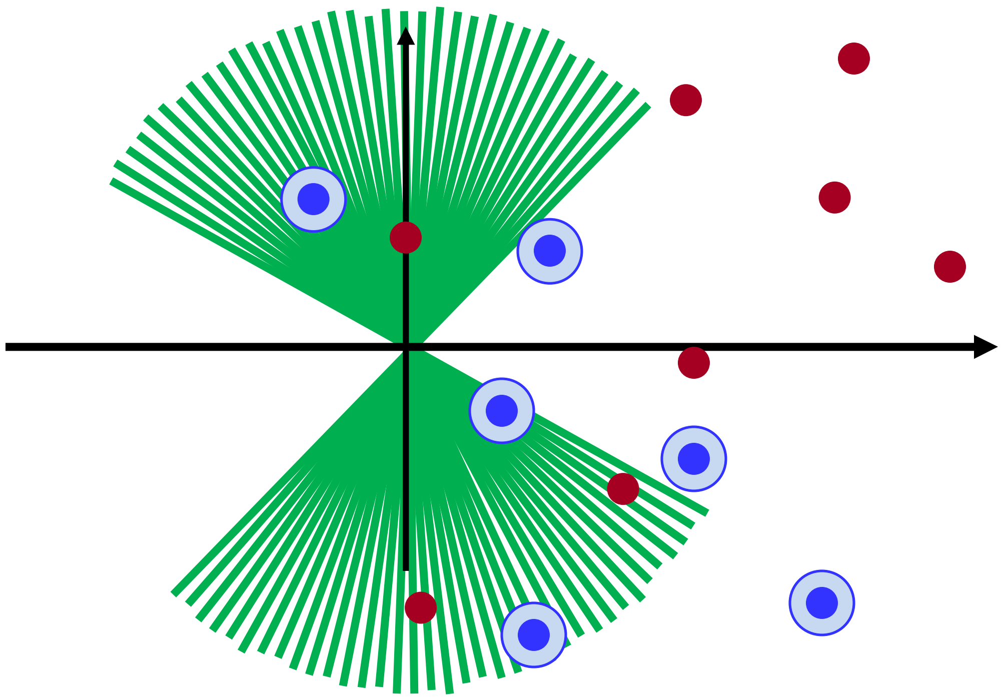
Key: Red 1, Blue = -1



- When classes are not linearly separable, not possible to find a separating hyperplane
  - No "support" plane for reflected data
  - Some points will always lie on the other side
- Model does not support perfect classification of this data

The Perceptron Solution: when classes are not linearly separable

Key: Red 1, Blue = -1



- When classes are not linearly separable, not possible to find a separating hyperplane
  - No “support” plane for reflected data
  - Some points will always lie on the other side
- Model does not support perfect classification of this data

The *online* perceptron solution

- The more popular solution, originally proposed by Rosenblatt is an *online* algorithm
  - The famous “perceptron” algorithm

- Initializes W and incrementally updates it each time we encounter an instance that is incorrectly classified
  - Guaranteed to find the correct solution for linearly separable data
  - On following slides, but will not cover in class

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 $(X_1, y_1), (X_2, y_2), \ldots, (X_N, y_N)$
  - $y_i = +1$ or -1

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

- Initialize W
- Cycle through the training instances:
- do
  - For $i = 1..N_{train}$

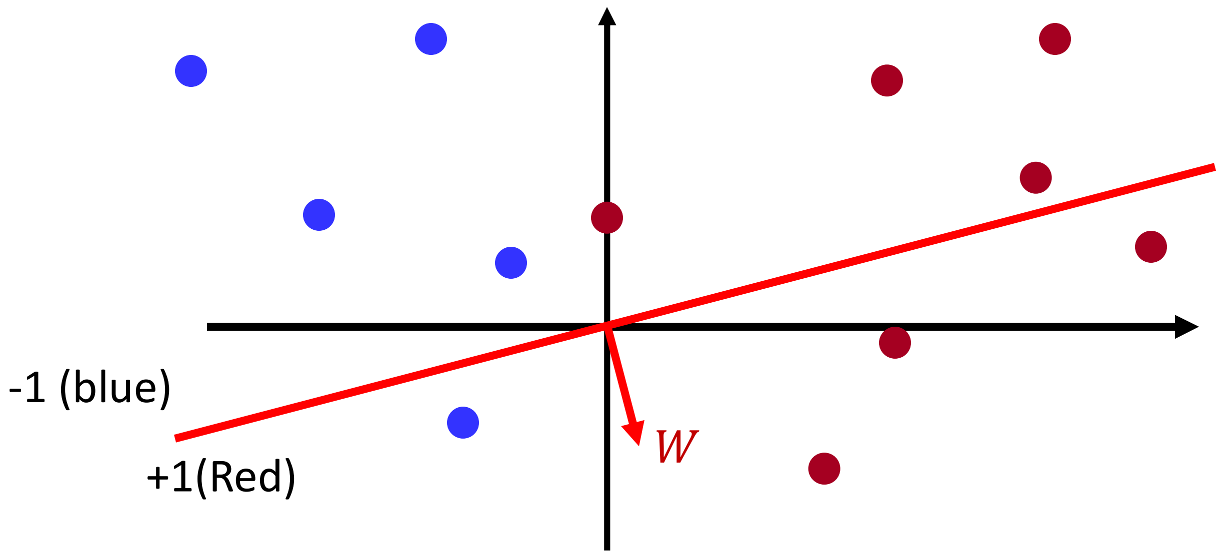
$$O(X_i) = sign(W^T X_i)$$

    - If $O(X_i) \neq y_i$

$$W = W + y_i X_i$$

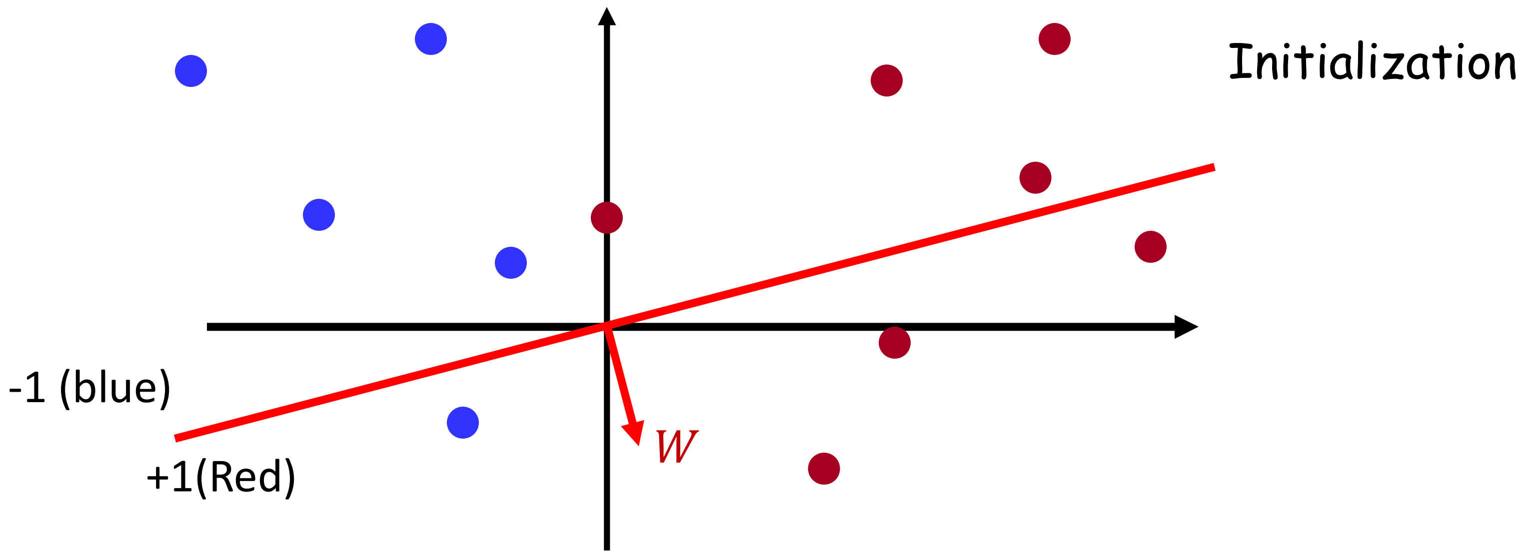
- until no more classification errors

A Simple Method: The Perceptron Algorithm

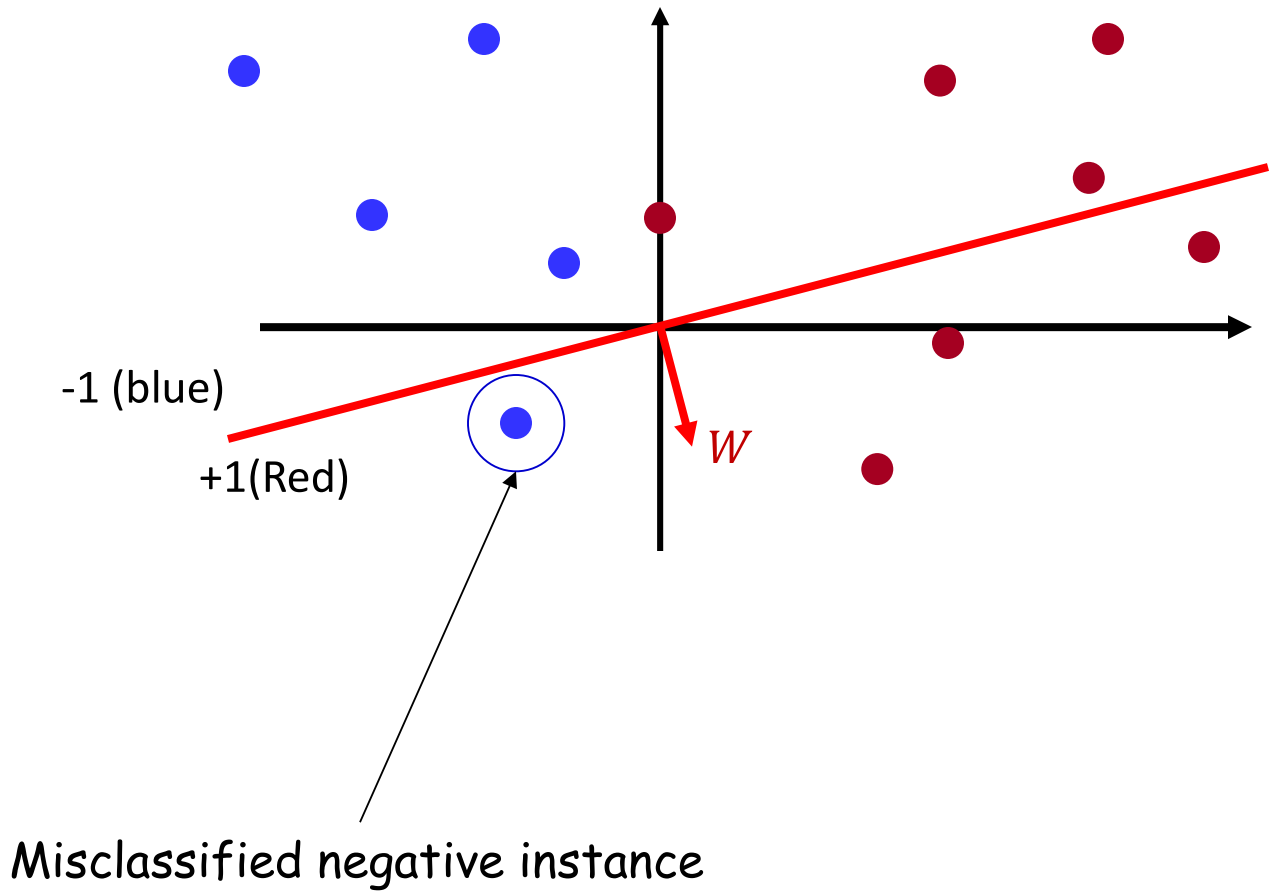


- **Initialize:** Randomly initialize the hyperplane
  - I.e. randomly initialize the normal vector W
- **Classification rule** $sign(W^T X)$
  - Vectors on the same side of the hyperplane as W will be assigned +1 class, and those on the other side will be assigned -1
- The random initial plane will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

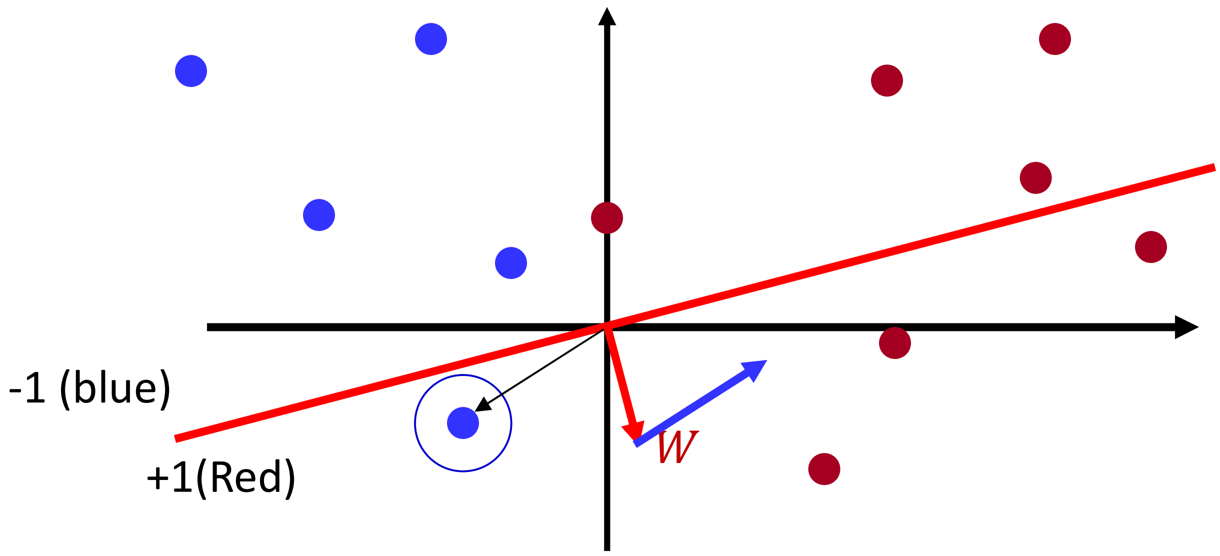
Perceptron Algorithm



Perceptron Algorithm

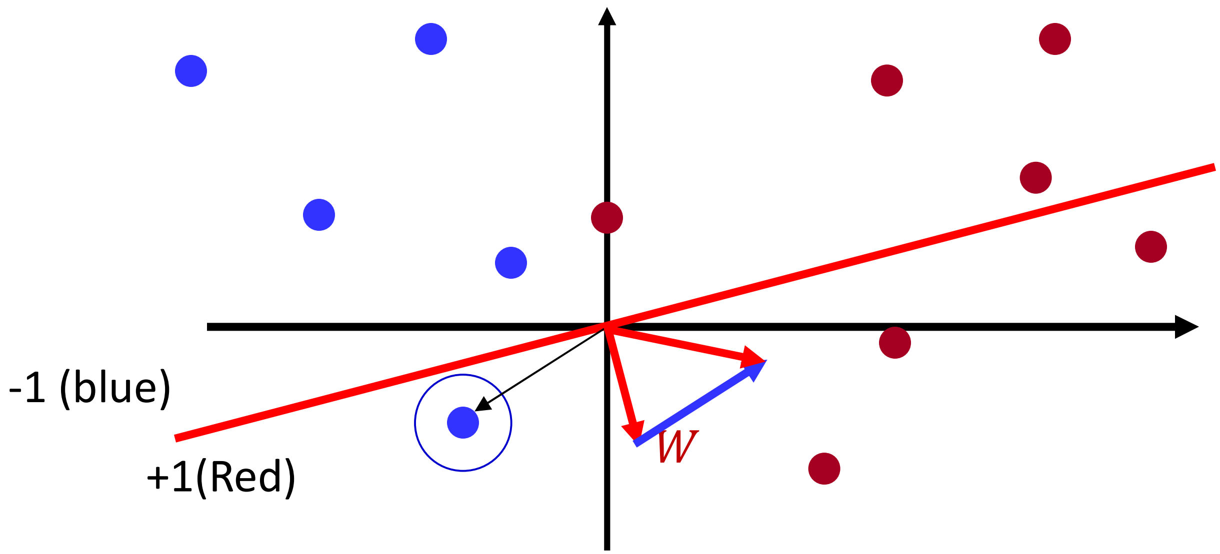


Perceptron Algorithm



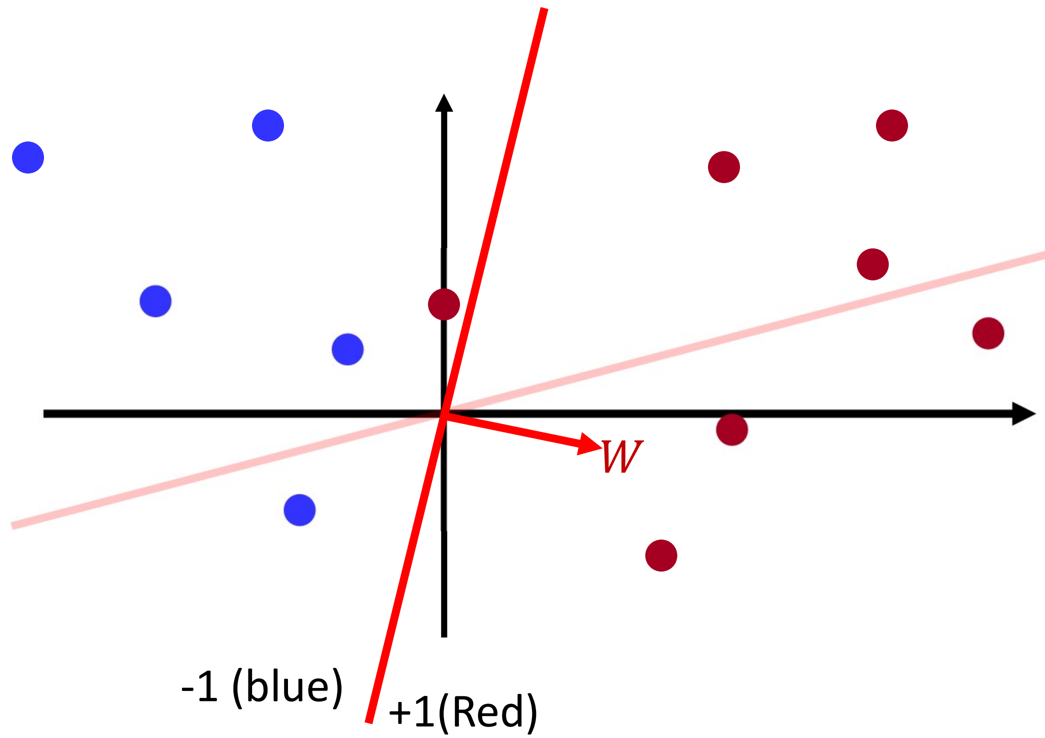
Misclassified *negative* instance, *subtract* it from W

Perceptron Algorithm



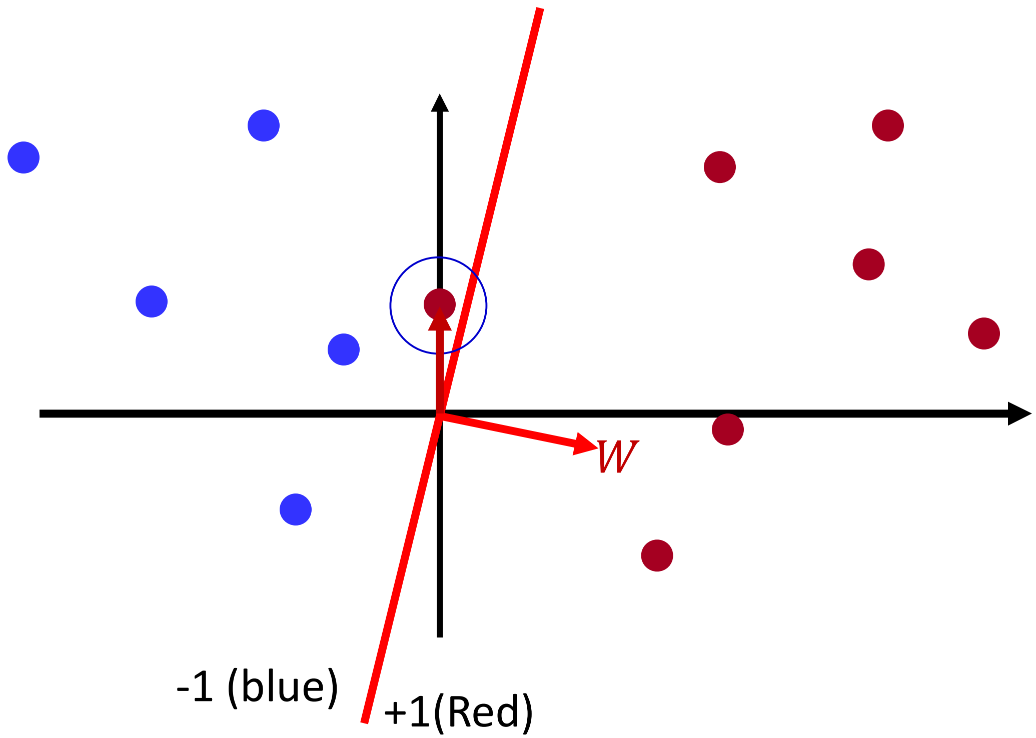
The new weight

Perceptron Algorithm



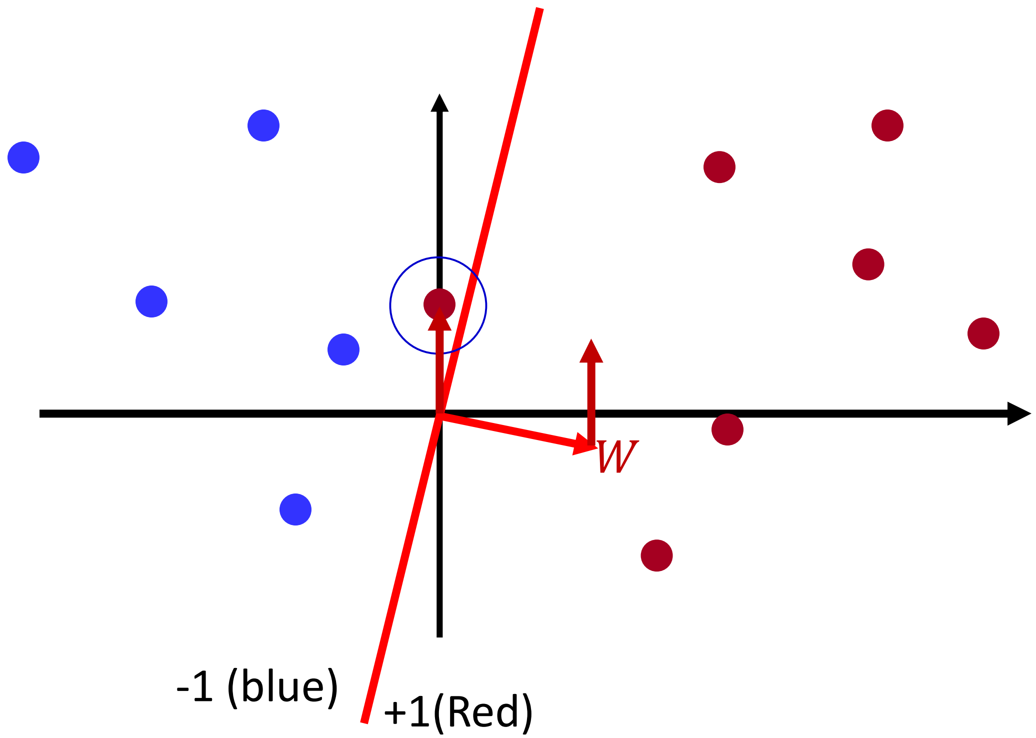
The new weight (and boundary)

Perceptron Algorithm



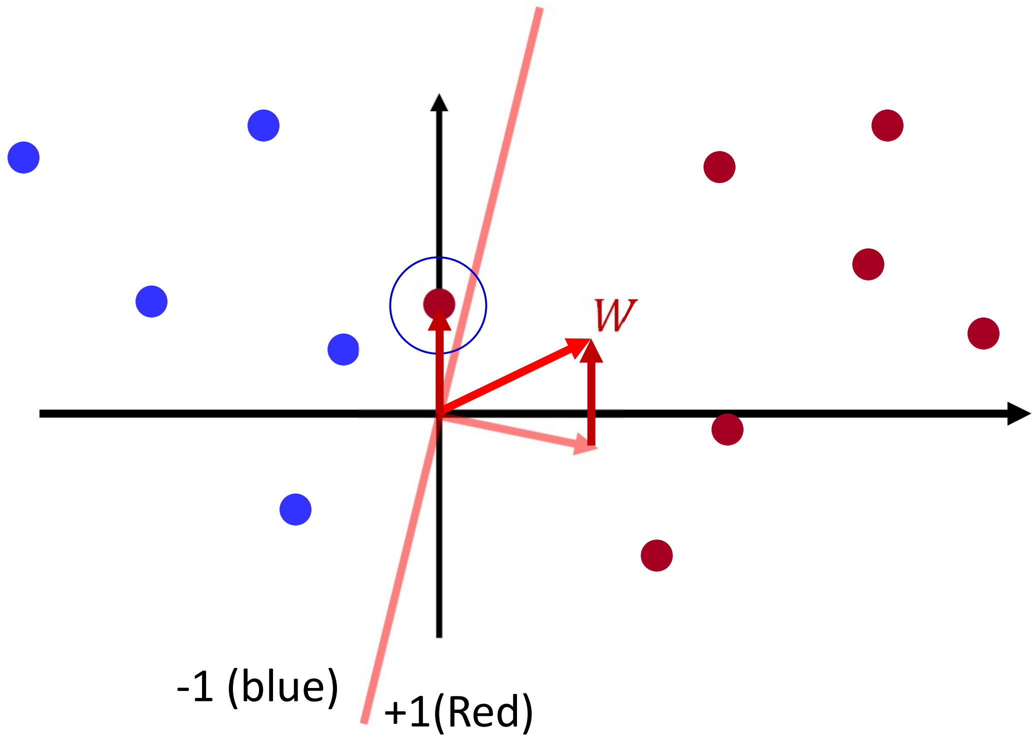
Misclassified *positive* instance

Perceptron Algorithm



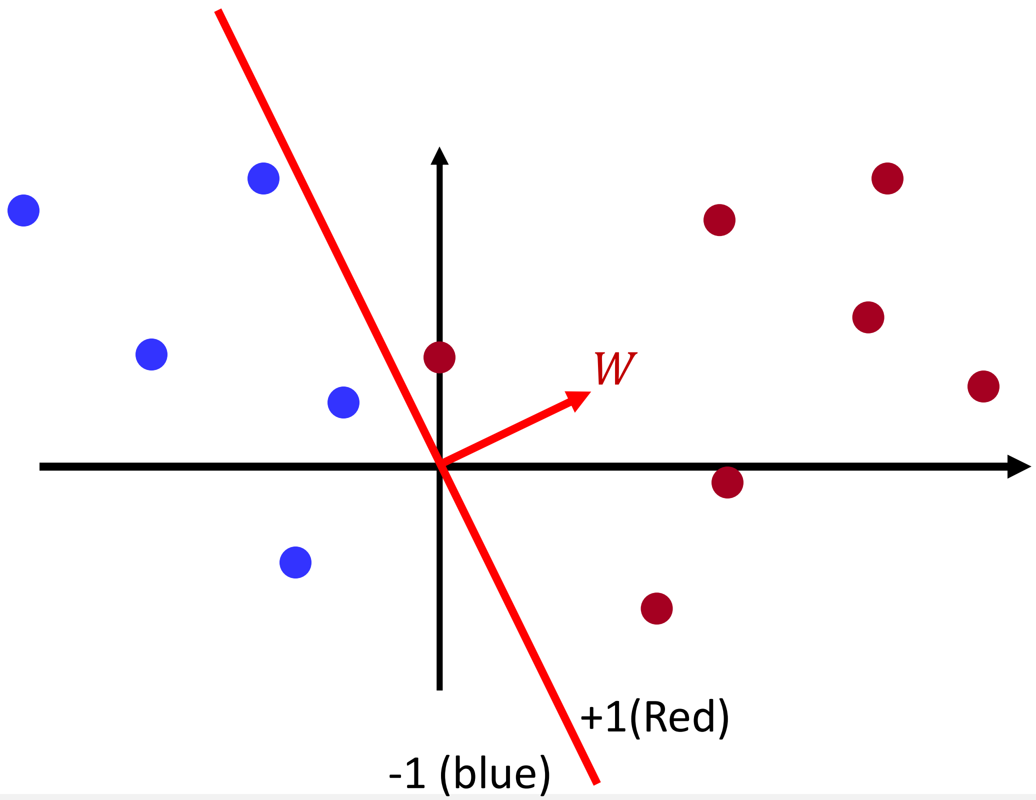
Misclassified *positive* instance, *add* it to W

Perceptron Algorithm



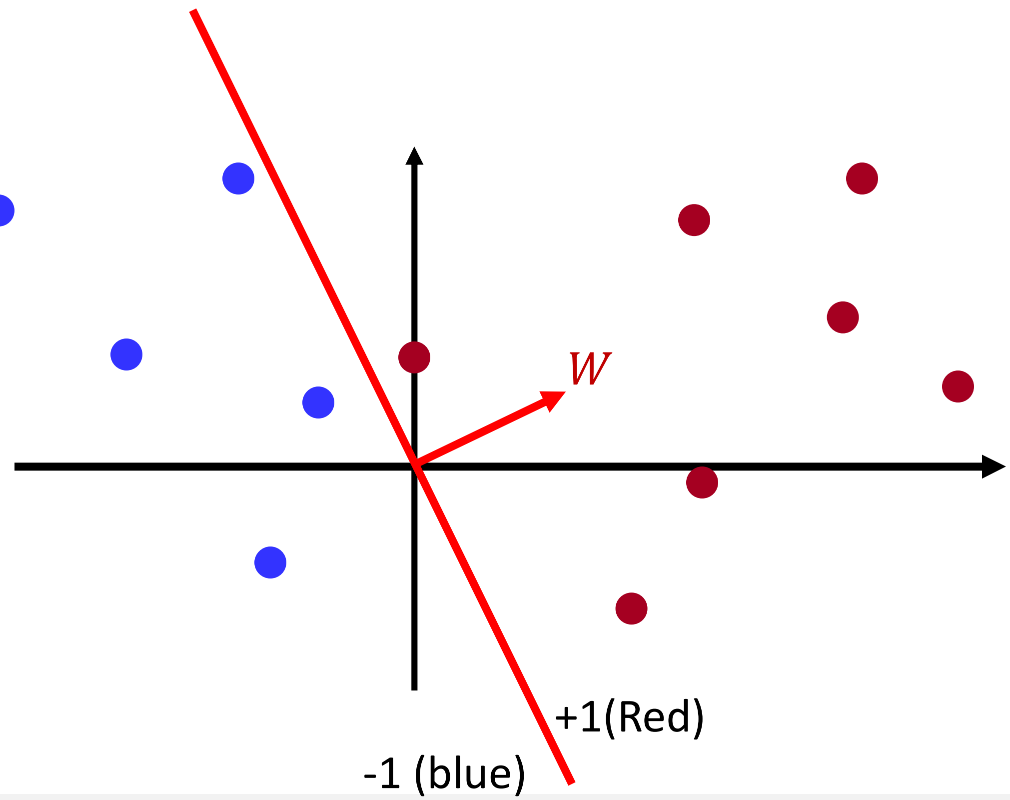
The new weight vector

Perceptron Algorithm



The new decision boundary
Perfect classification, no more updates, we are done

Perceptron Algorithm



The new decision boundary

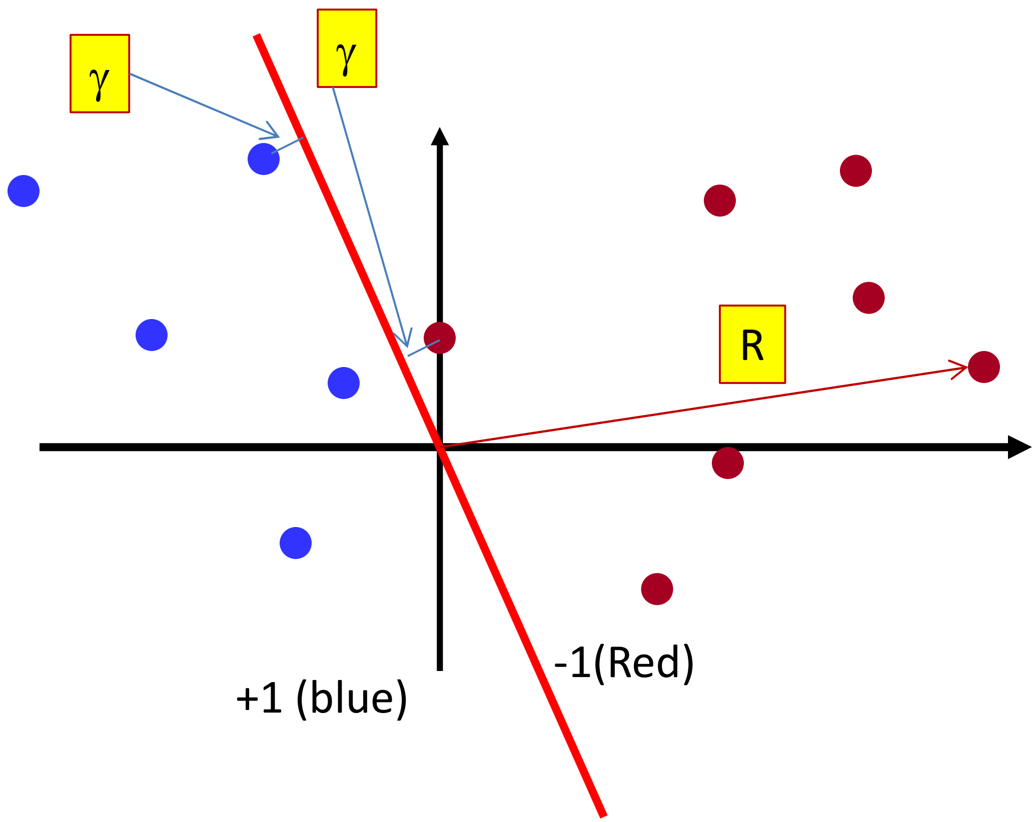
Perfect classification, no more updates, we are done

If the classes are linearly separable, guaranteed to converge in a finite number of steps

Convergence of Perceptron Algorithm

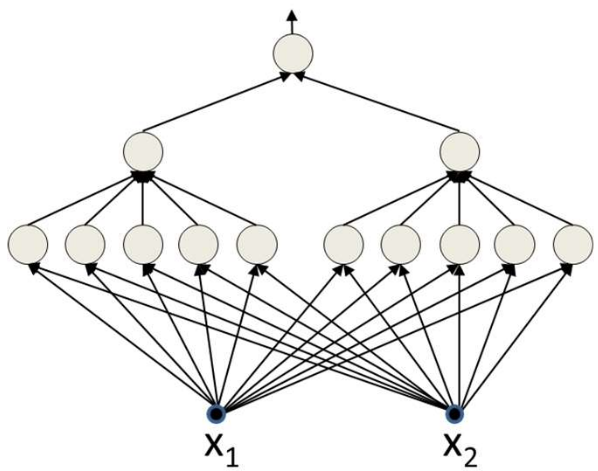
- Guaranteed to converge if classes are linearly separable
  - After no more than $\left(\frac{R}{\gamma}\right)^2$ misclassifications
    - Specifically when W is initialized to 0
  - R is length of longest training point
  - γ is the *best case* closest distance of a training point from the classifier
    - 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

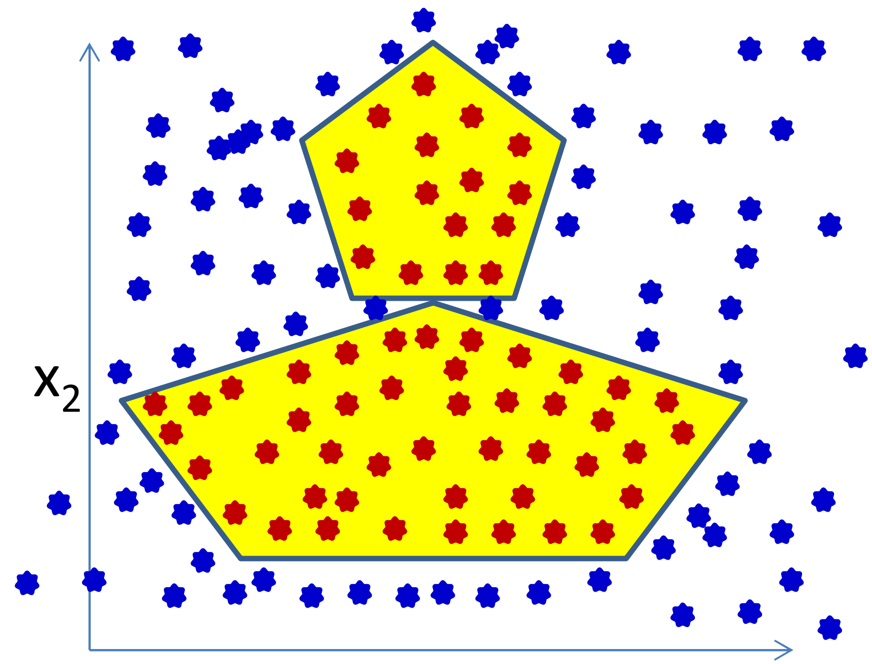
Perceptron Algorithm



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

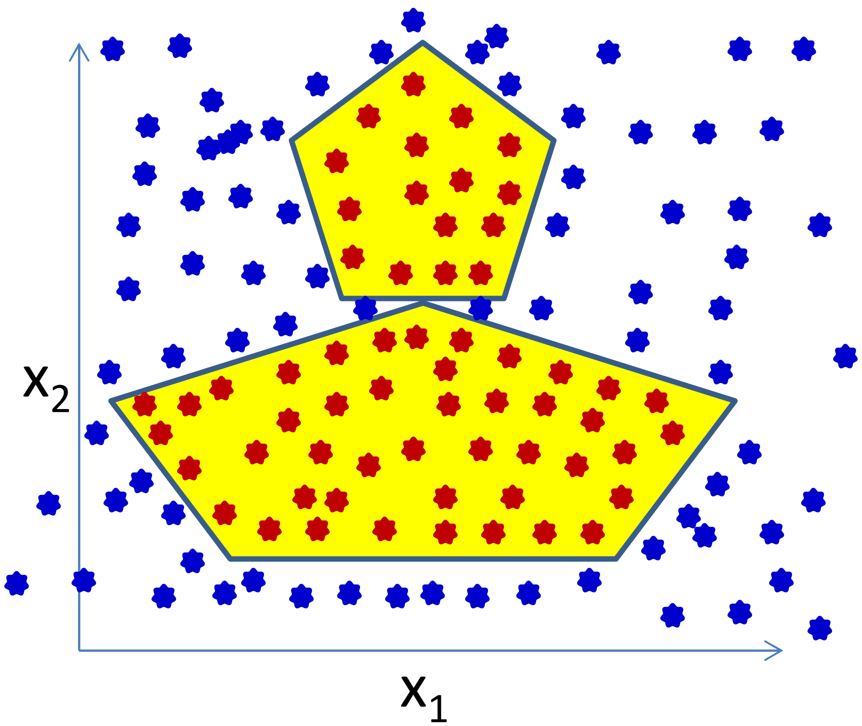
History: A more complex problem

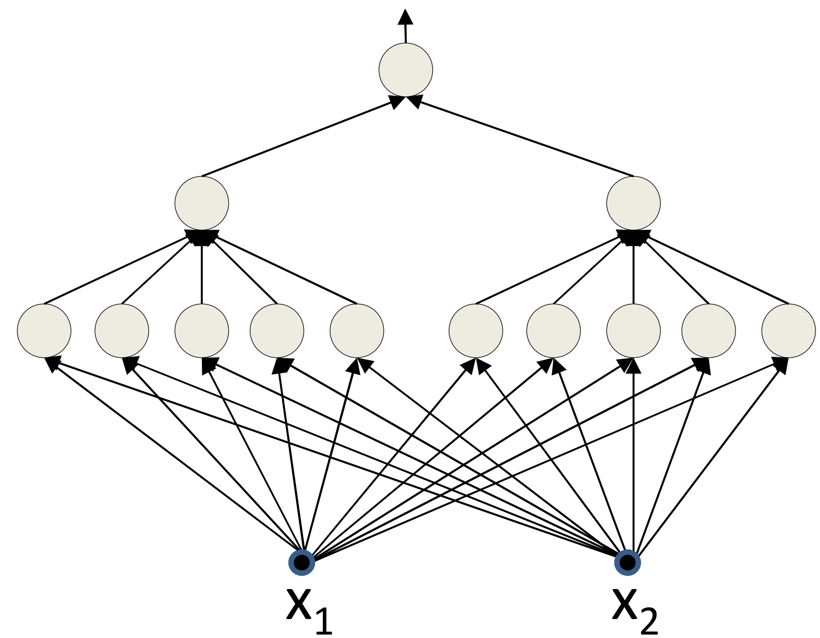




- Learn an *MLP* for this function
  - 1 in the yellow regions, 0 outside
- Using just the samples
- We know this can be perfectly represented using an MLP

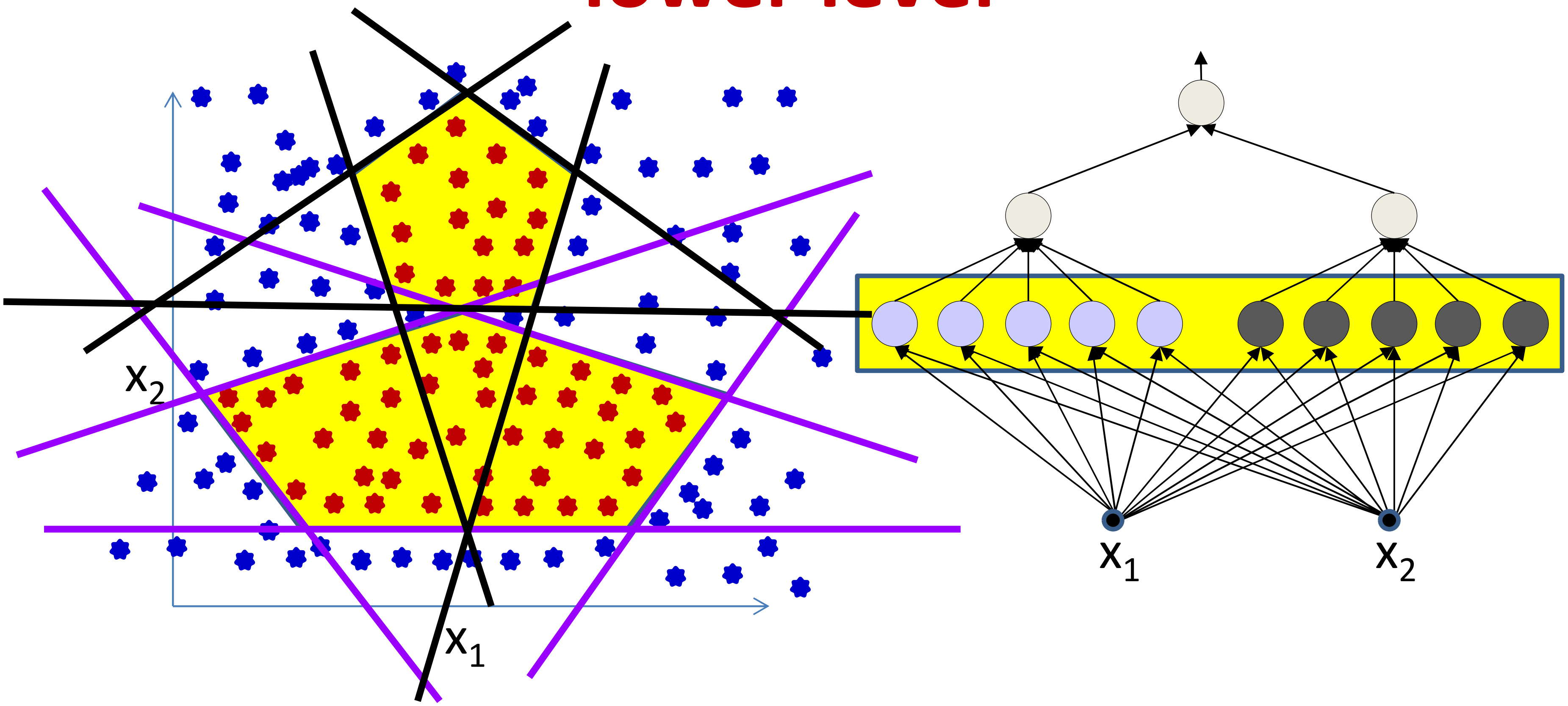
More complex decision boundaries





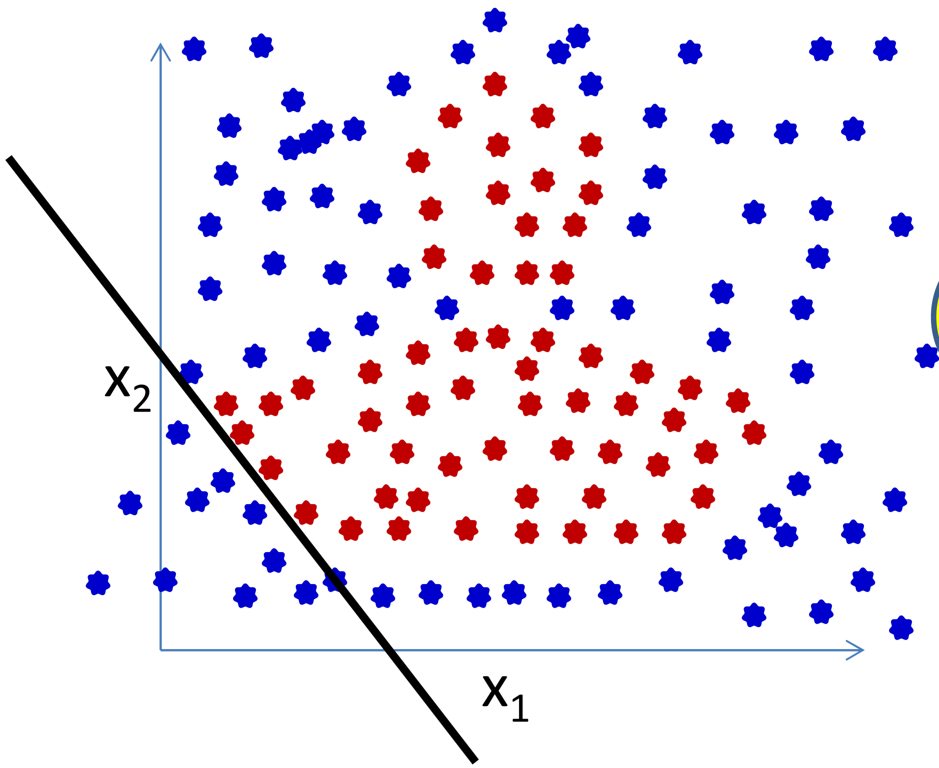
- Even using the perfect architecture…
- … can we use perceptron learning rules to learn this classification function?

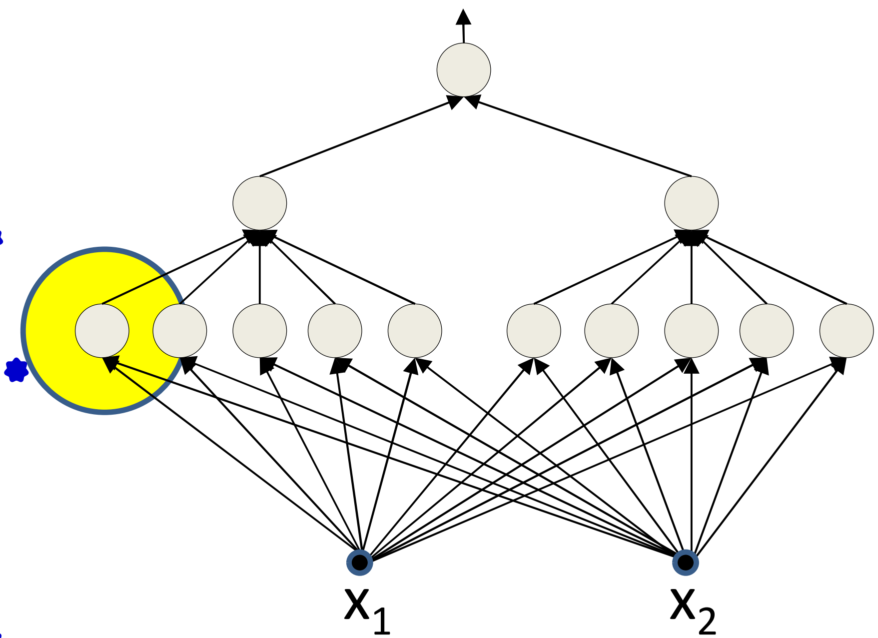
The pattern to be learned at the lower level



- 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!*

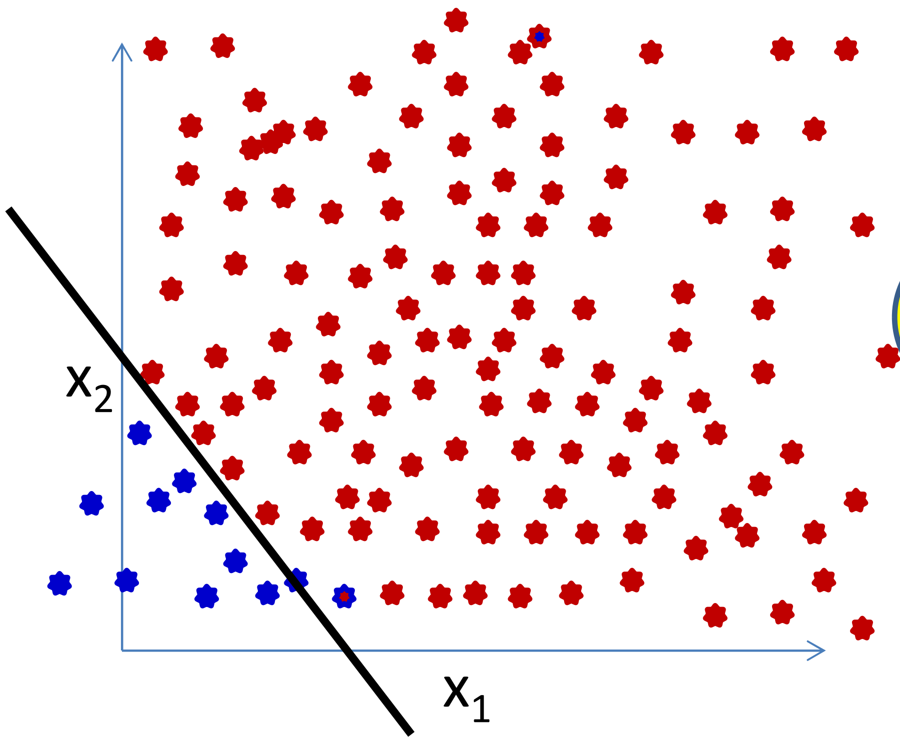
The pattern to be learned at the lower level

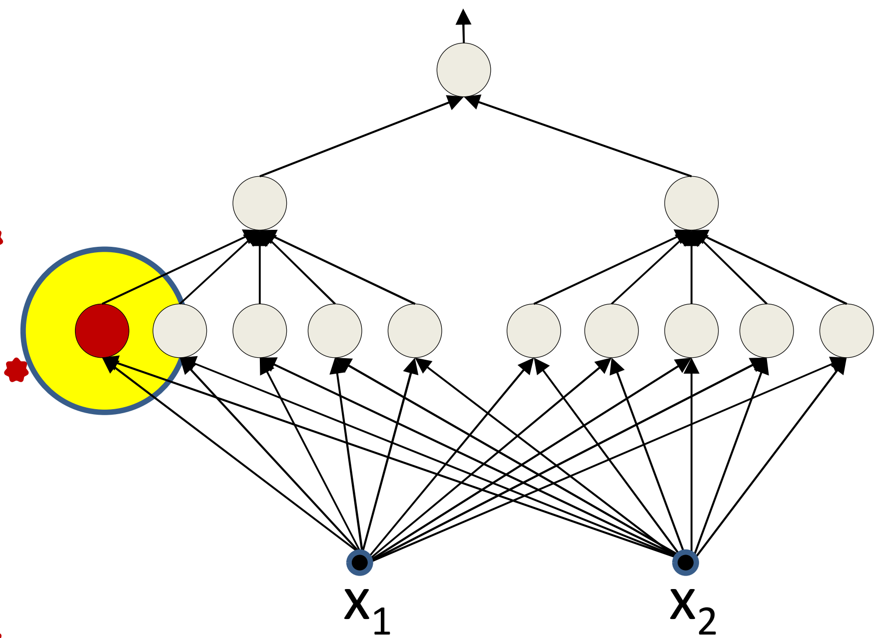




- Consider a single linear classifier that must be learned from the training data
  - Can it be learned from this data?

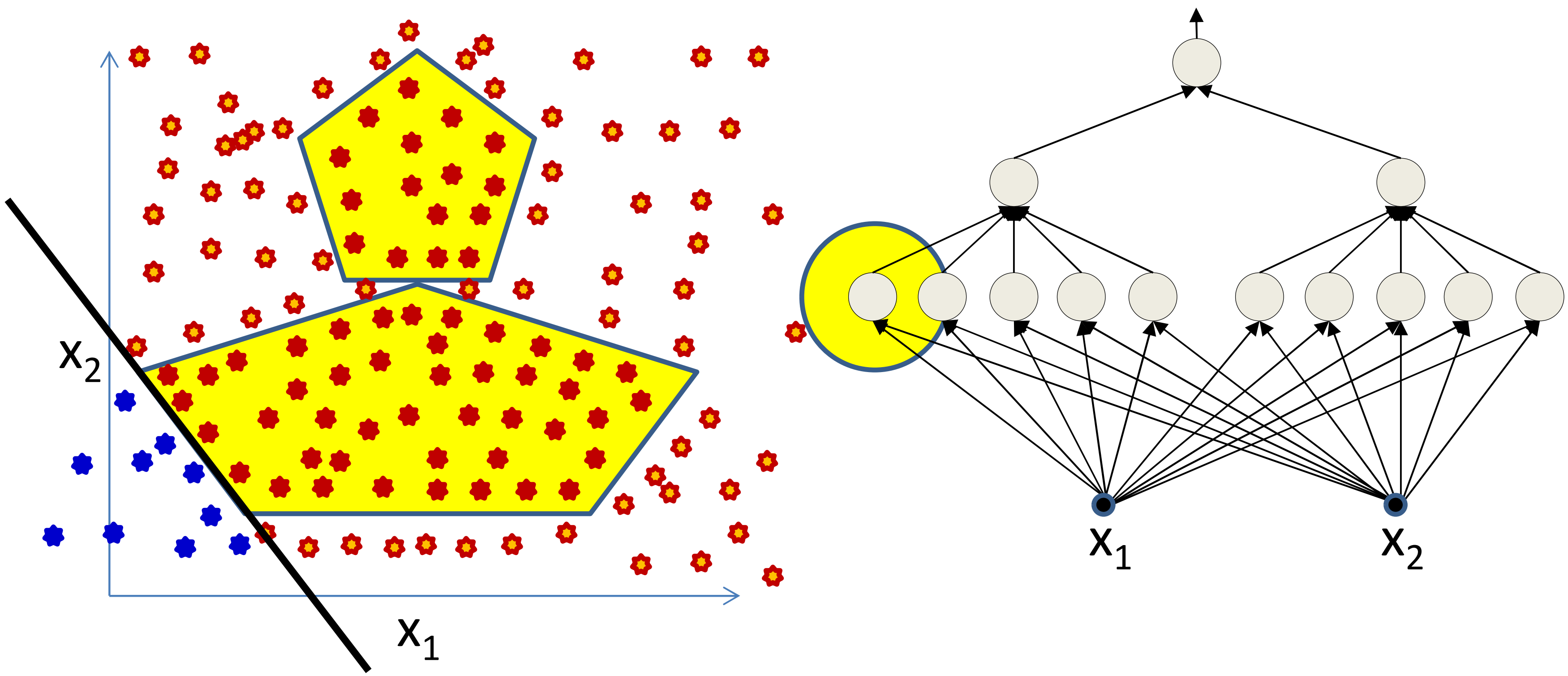
The pattern to be learned at the lower level





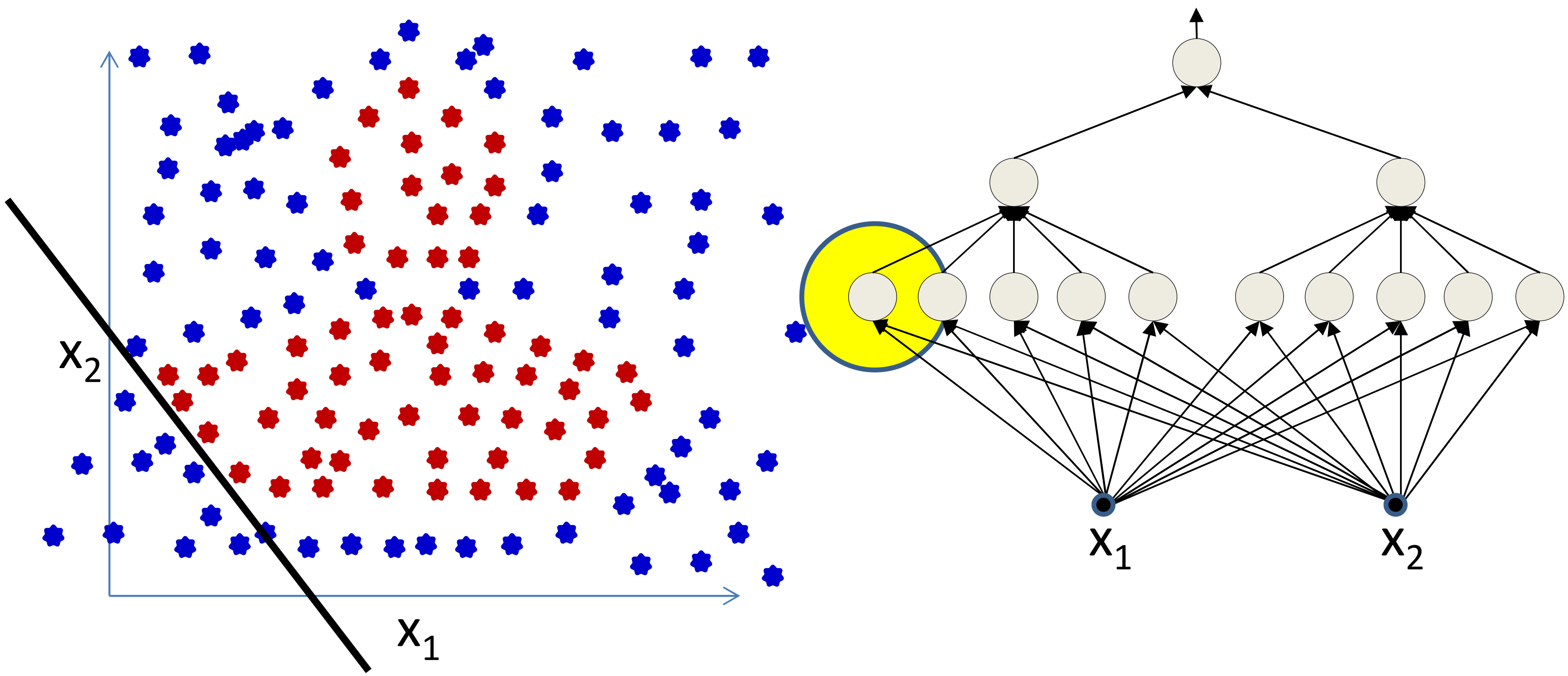
- Consider a single linear classifier that must be learned from the training data
  - Can it be learned from this data?
  - The individual classifier actually requires the kind of labelling shown here
    - Which is *not* given!!

The pattern to be learned at the lower level



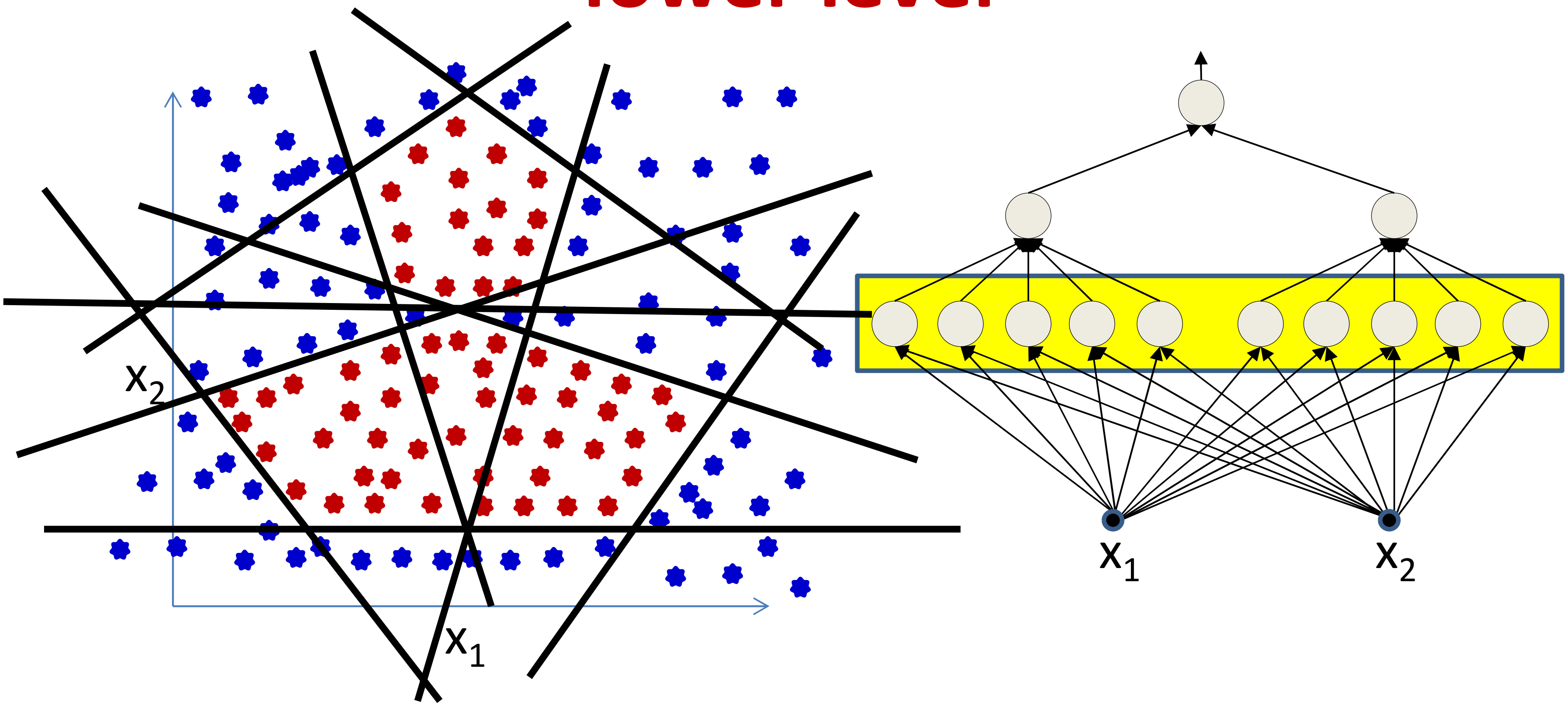
- The lower-level neurons are linear classifiers
  - They require linearly separated labels to be learned
  - The actually provided labels are not linearly separated
  - Challenge: *Must also learn the labels for the lowest units!*

The pattern to be learned at the lower level



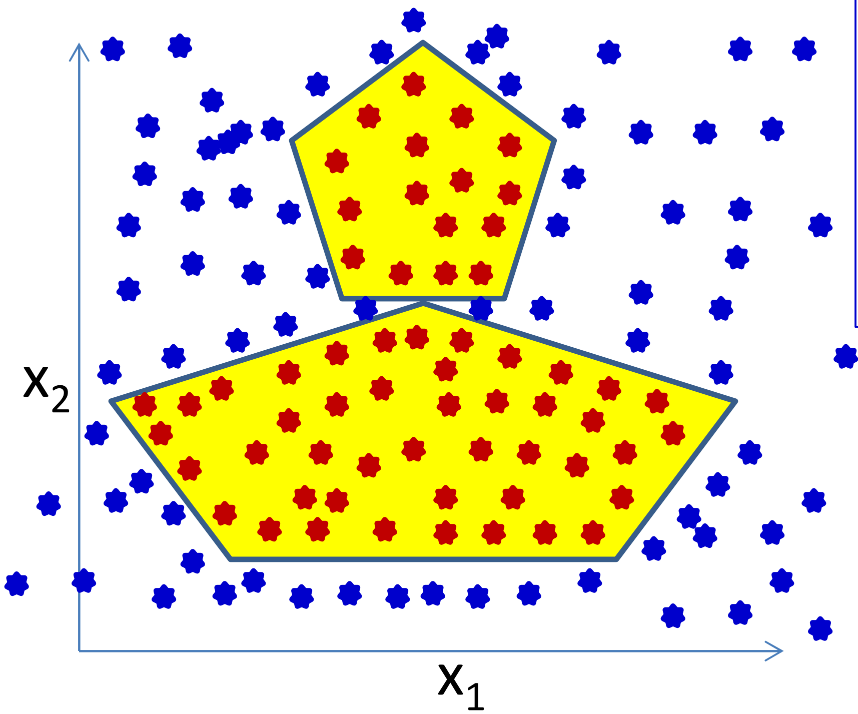
- For a single line:
  - Try out *every possible way of relabeling the blue dots such that we can learn a line that keeps all the red dots on one side!*

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

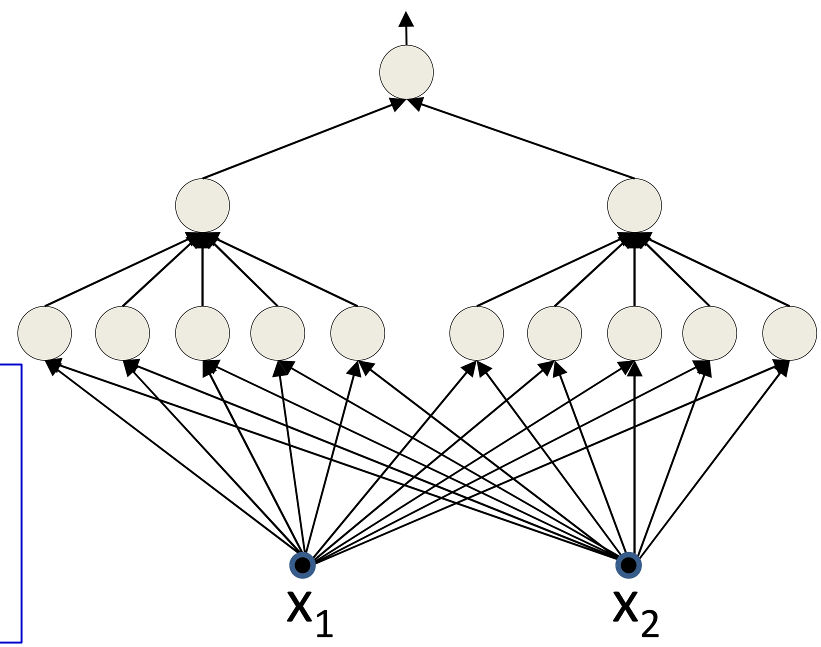
Individual neurons represent one of the lines that compose the figure (linear classifiers)



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

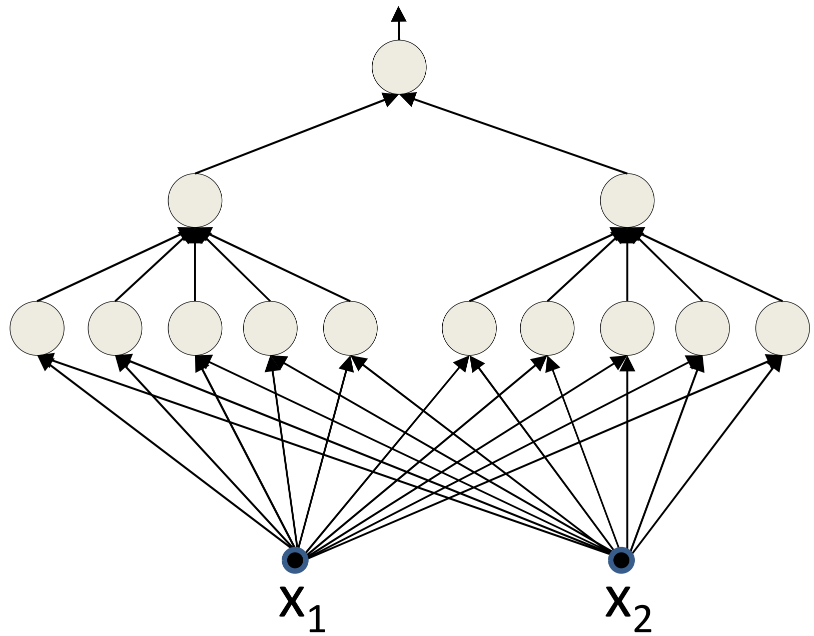
The linear separators must combine to form the desired boundary



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

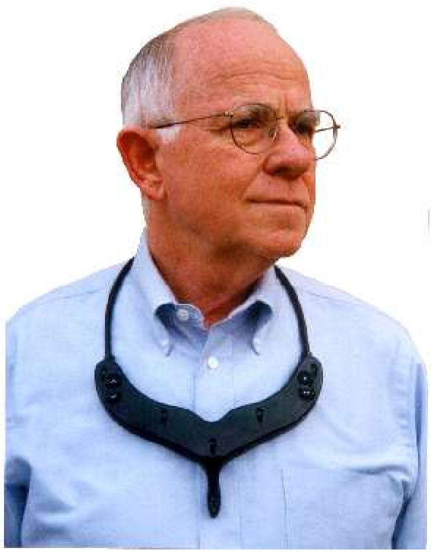
- Training this network using the perceptron rule is a combinatorial optimization problem
- We don't know the outputs of the individual intermediate neurons in the network for any training input
- **Must also determine the correct output for *each* neuron for *every* training instance**
- **At least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

Greedy algorithms: Adaline and Madaline

- Perceptron learning rules 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!

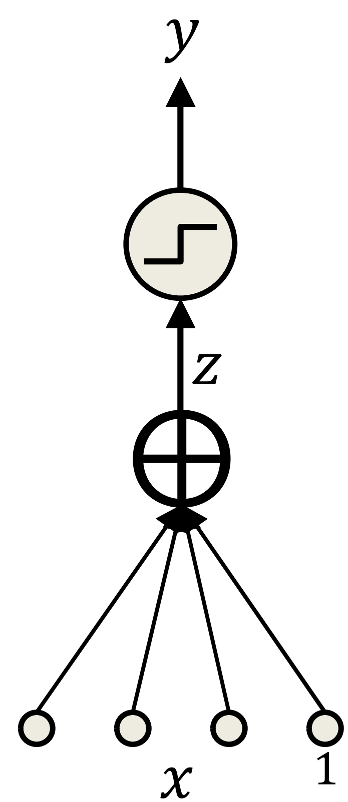
- First known attempt at an analytical solution to training the perceptron and the MLP
- Now famous as the LMS algorithm
  – Used everywhere
  – Also known as the "delta rule"

History: ADALINE

$$z = \sum_t w_i x_i$$

Using 1-extended vector notation to account for bias

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



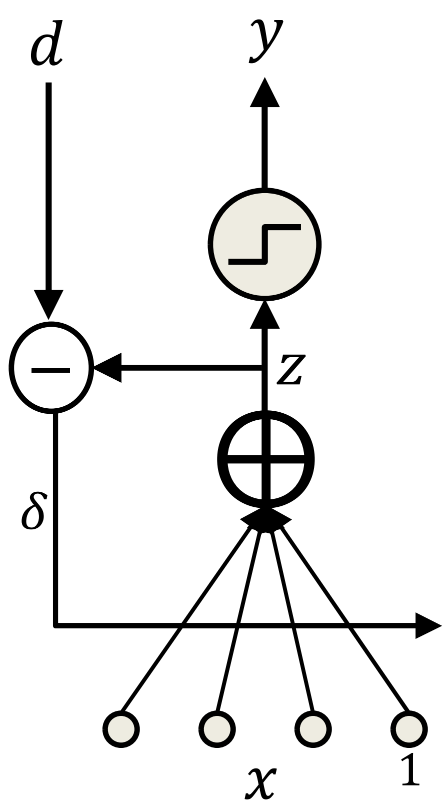
- Adaptive *linear* element (Hopf and Widrow, 1960)

- 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

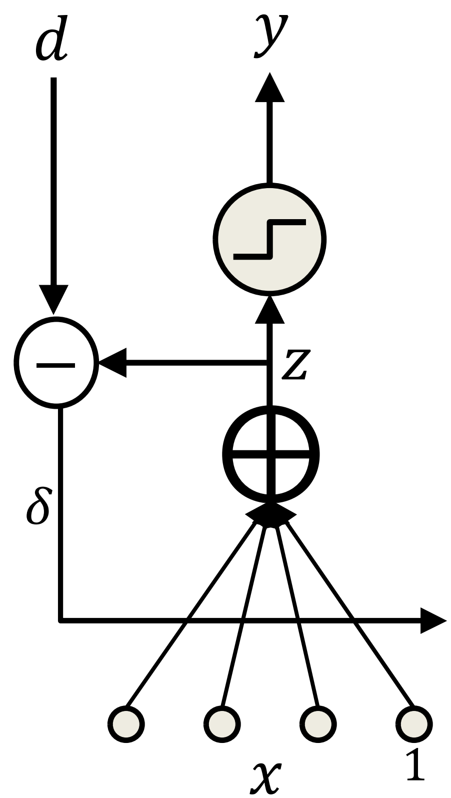
$$z = \sum_t w_i x_i$$

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

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



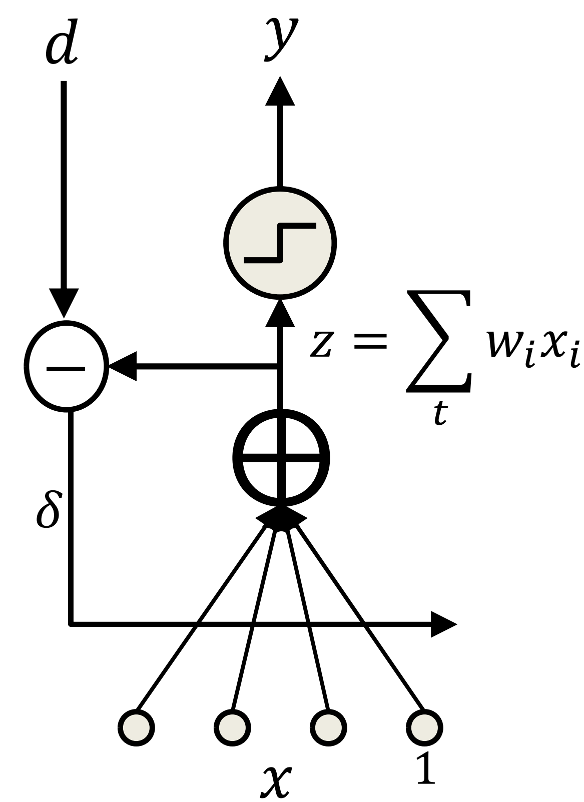
The ADALINE learning rule

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

$$\delta = d - z$$

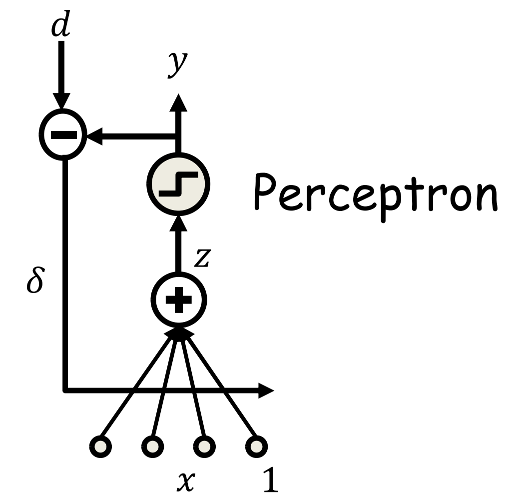
$$w_i = w_i + \eta \delta x_i$$

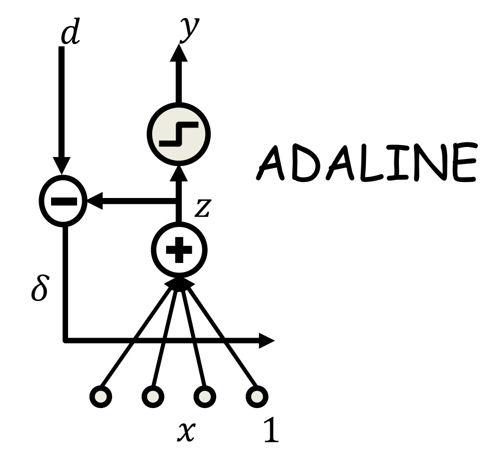
- This is the famous *delta rule*
  - Also called the LMS update rule



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



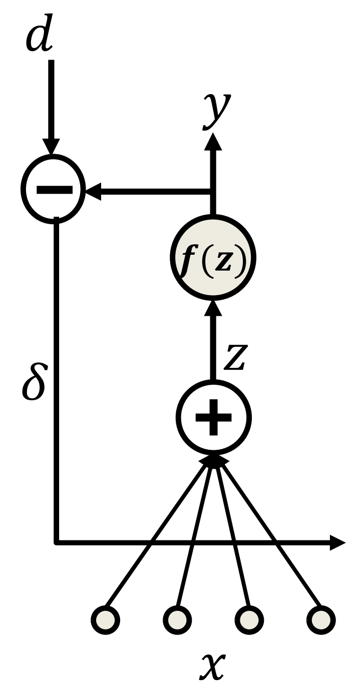


Aside: Generalized delta rule

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

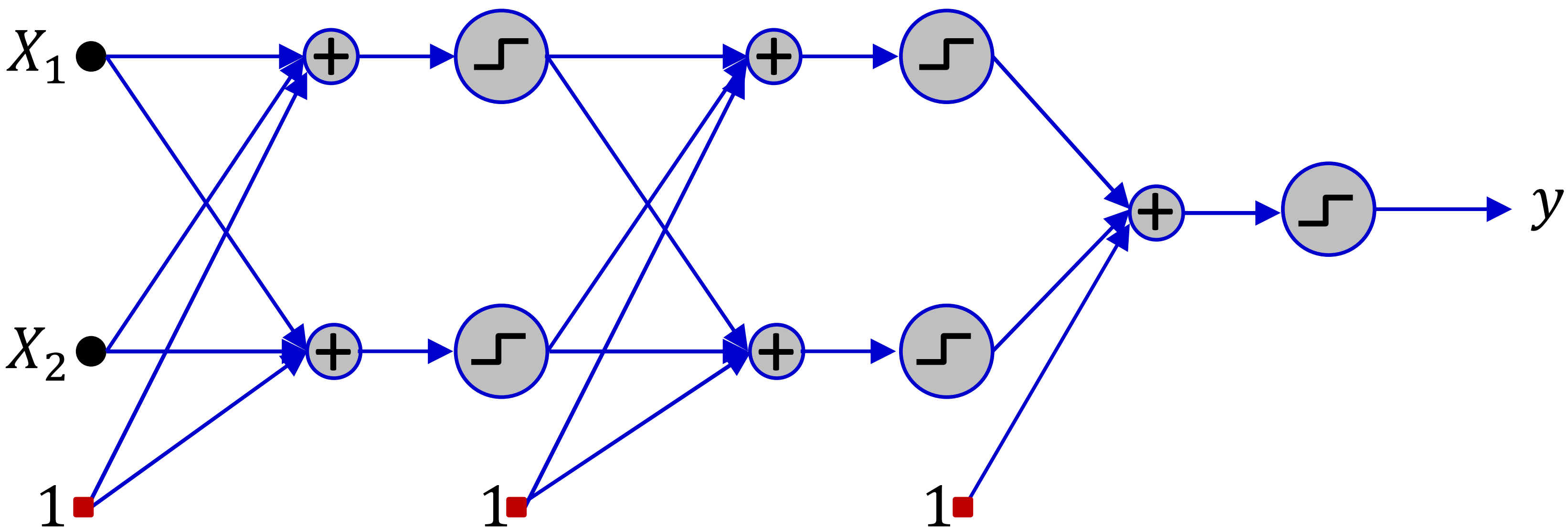
$$\delta = d - y$$

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



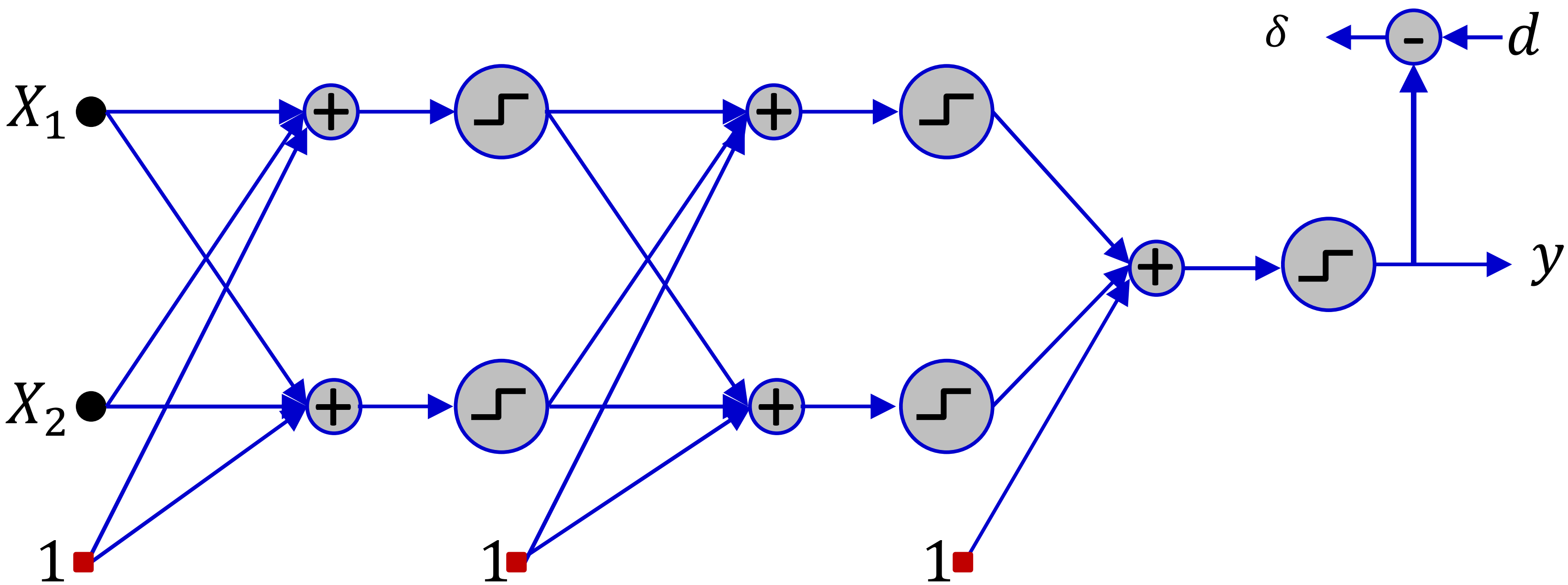
- This is the famous Widrow-Hoff update rule
  - Lookahead: Note that this is *exactly* backpropagation in multilayer nets if we let $f(z)$ represent the entire network between z and y
- It is possibly the most-used update rule in machine learning and signal processing
  - Variants of it appear in almost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

***Multilayer perceptron*: MADALINE**



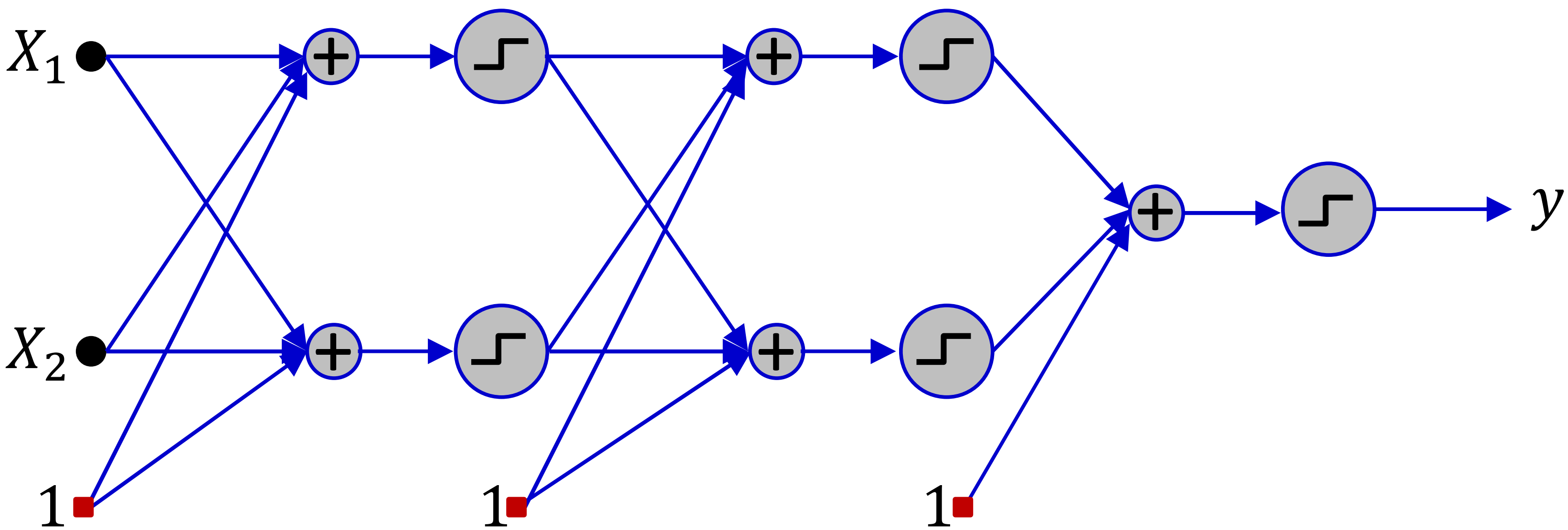
- *Multiple* Adaline
  - A multilayer perceptron with threshold activations
  - The MADALINE

MADALINE Training



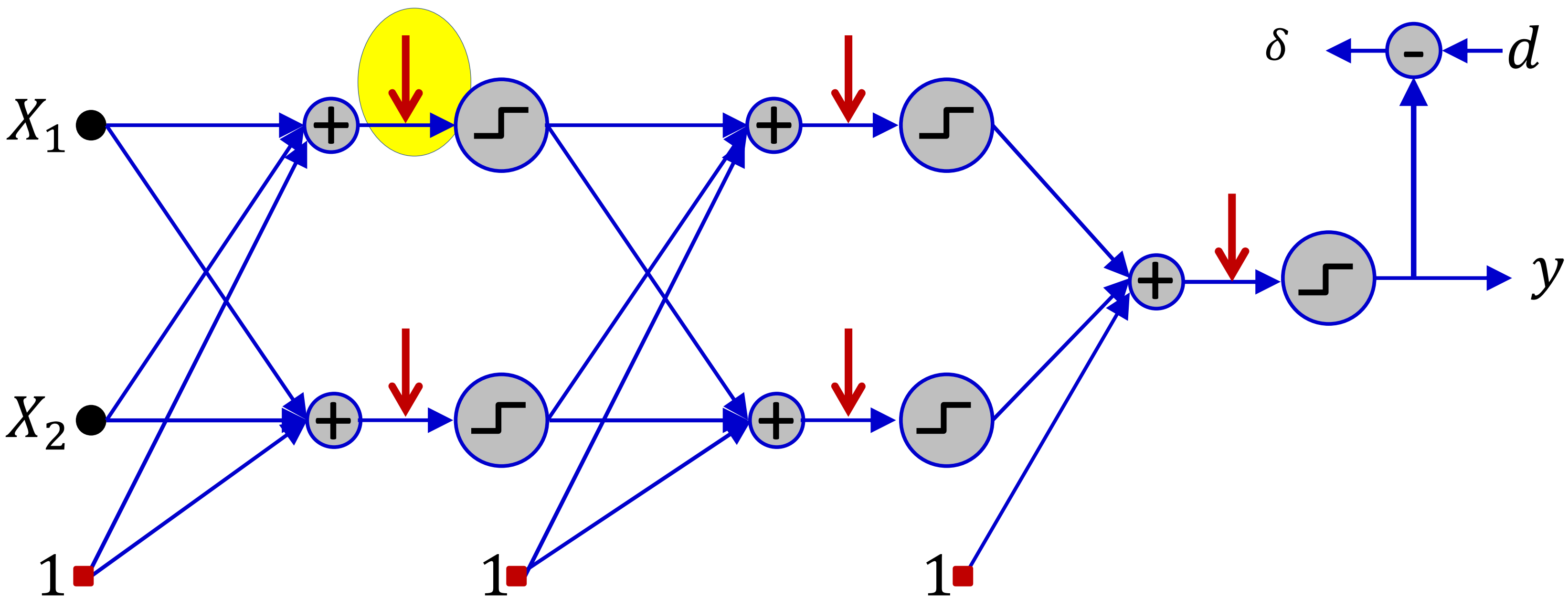
- *Update only on error*
  - $\delta \neq 0$
  - On inputs for which output and target values differ

MADALINE Training



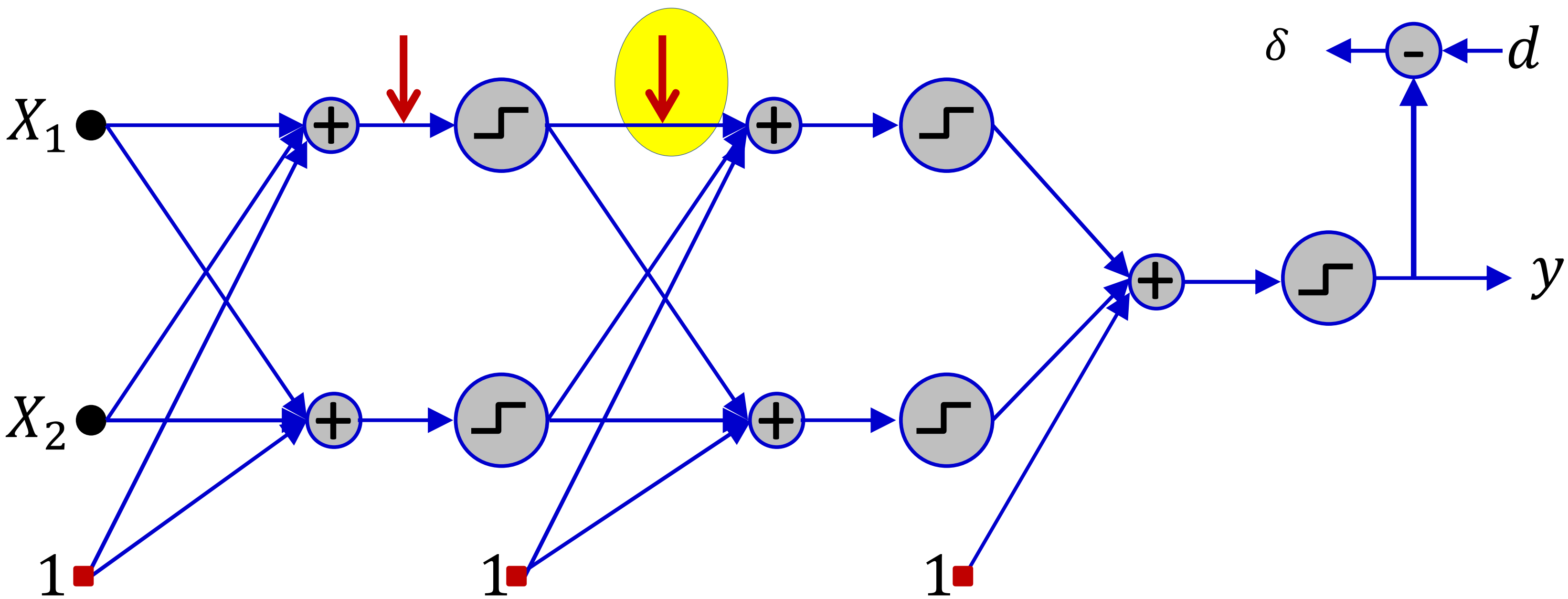
- While stopping criterion not met do:
  - Classify an input

MADALINE Training



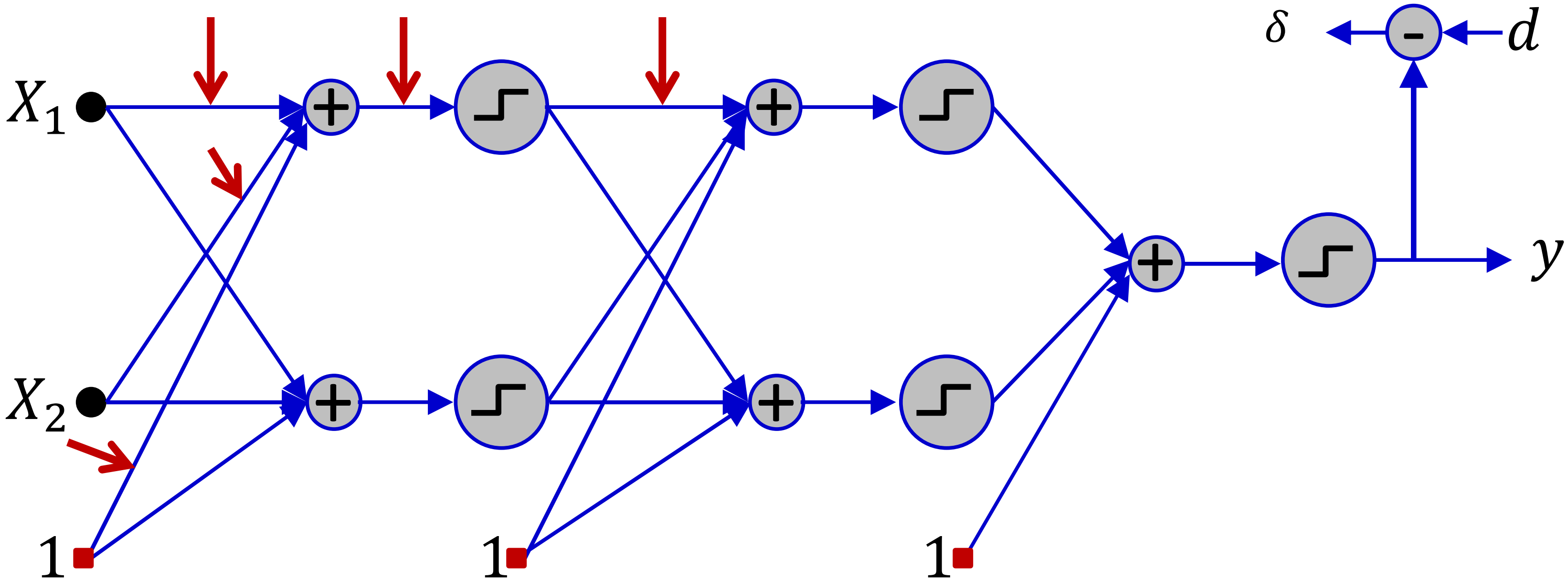
- While stopping criterion not met do:
  - Classify an input
  - If error, find the z that is closest to 0

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

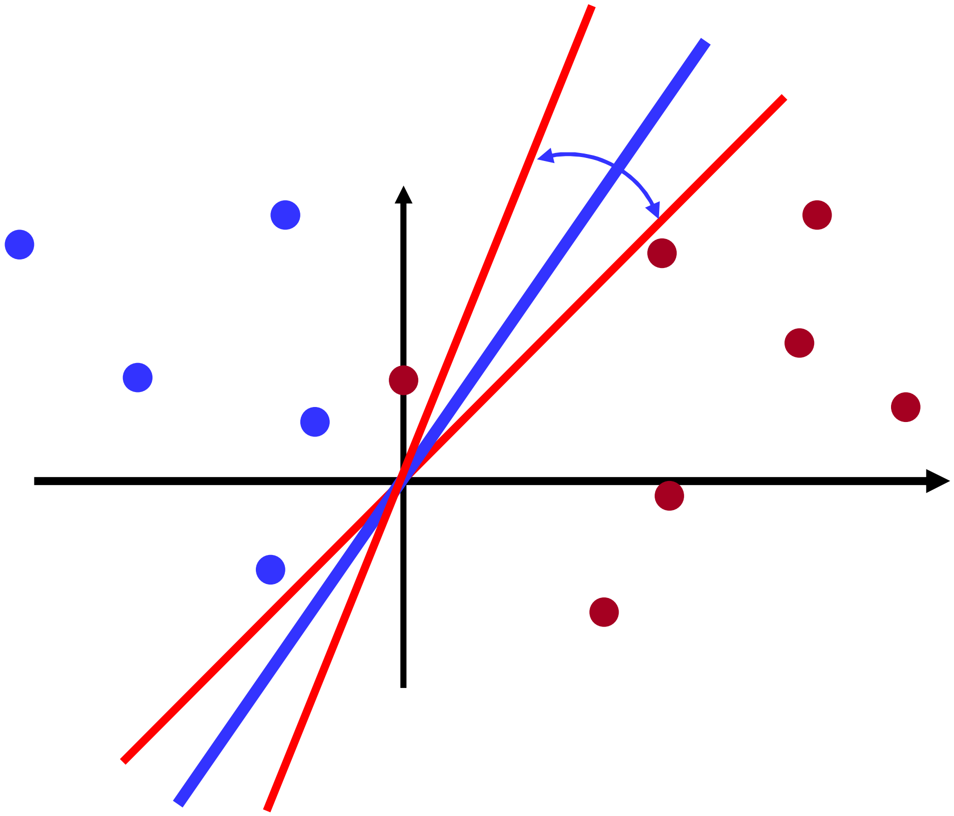
Story so far

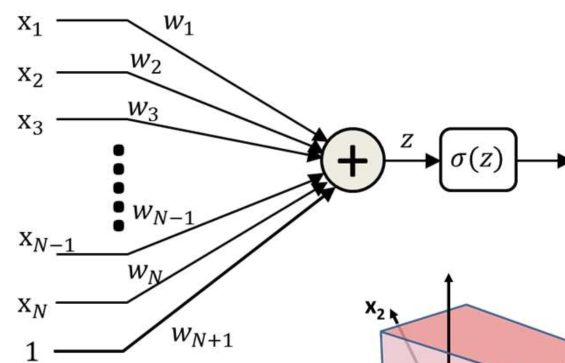
- "Learning" a network = learning the weights and biases to compute a target function
  - Will require a network with sufficient "capacity"
- In practice, we learn networks by "fitting" them to match the input-output relation of "training" instances drawn from the target function
- A linear decision 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

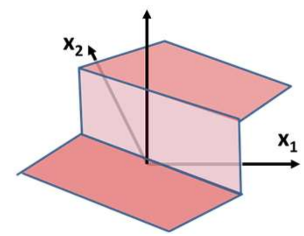
History..

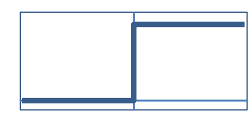
- The realization that training an entire MLP was a combinatorial optimization problem stalled development of neural networks for well over a decade!

Why this problem?



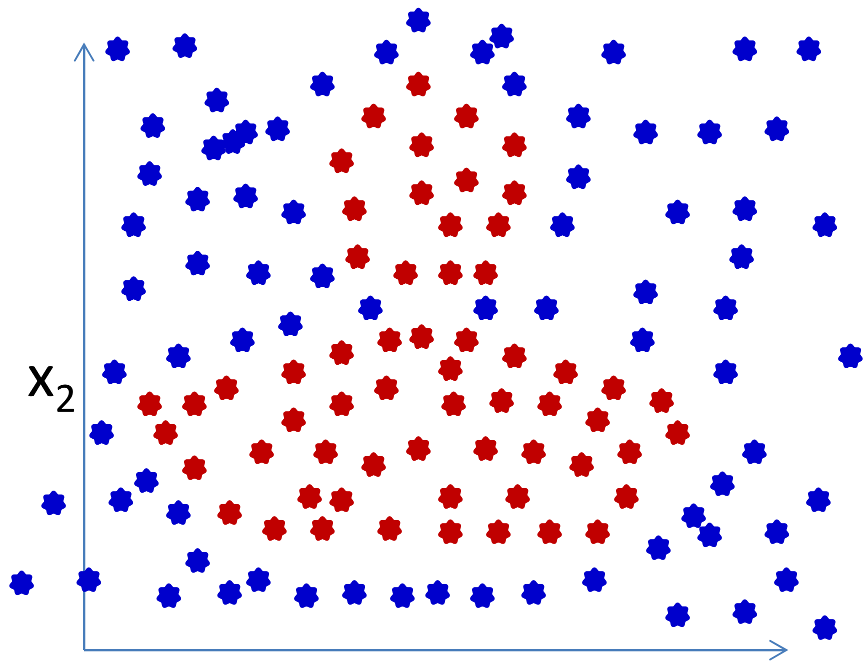


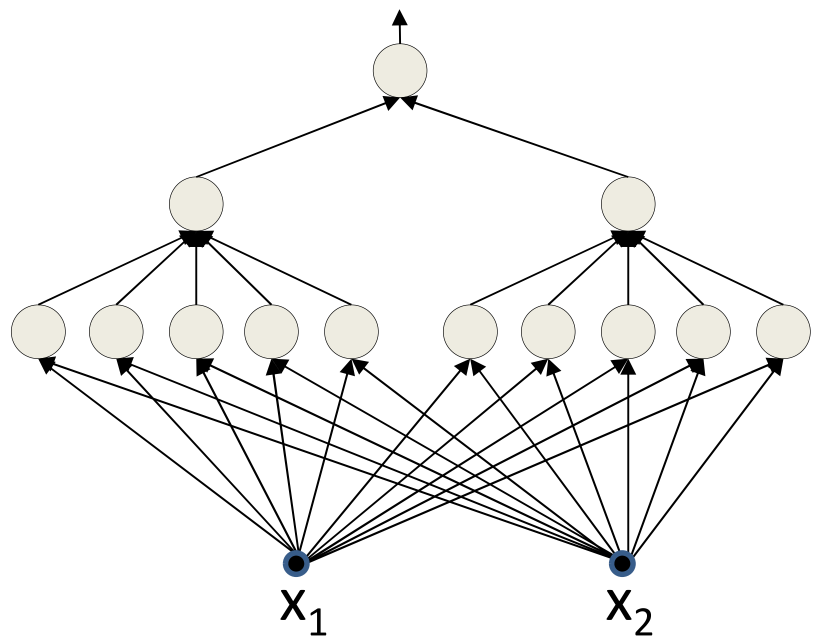


$\sigma(z) =$ 

- 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

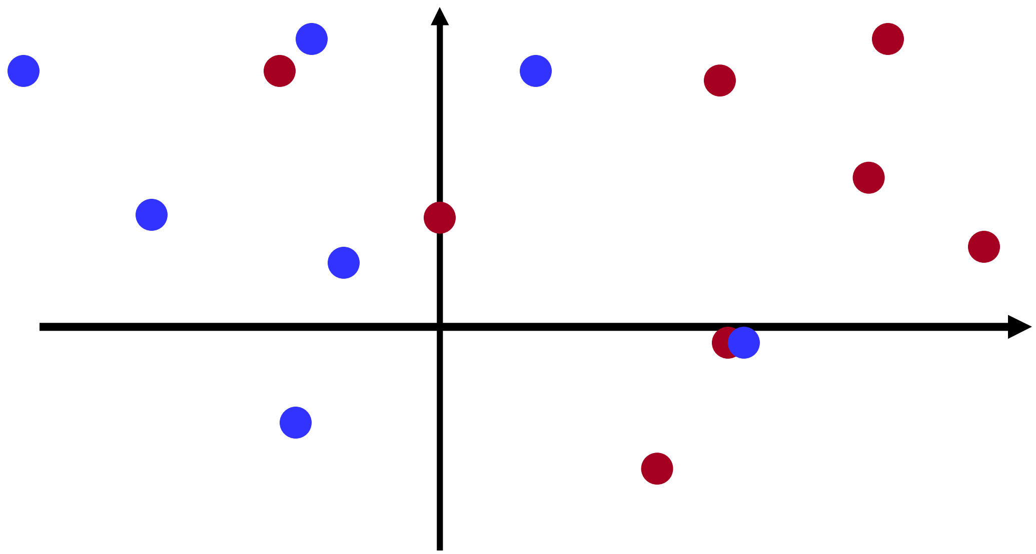
This only compounds on larger problems





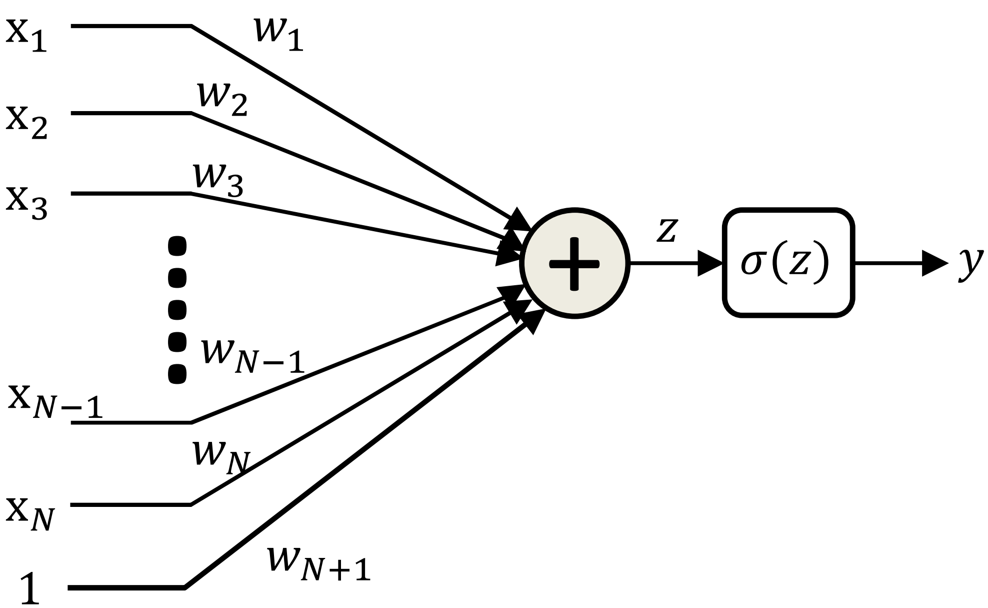
- Individual neurons' weights can change significantly without changing overall error
- The simple MLP is a flat, non-differentiable function
  - Actually a function with 0 derivative nearly everywhere, and no derivatives at the boundaries

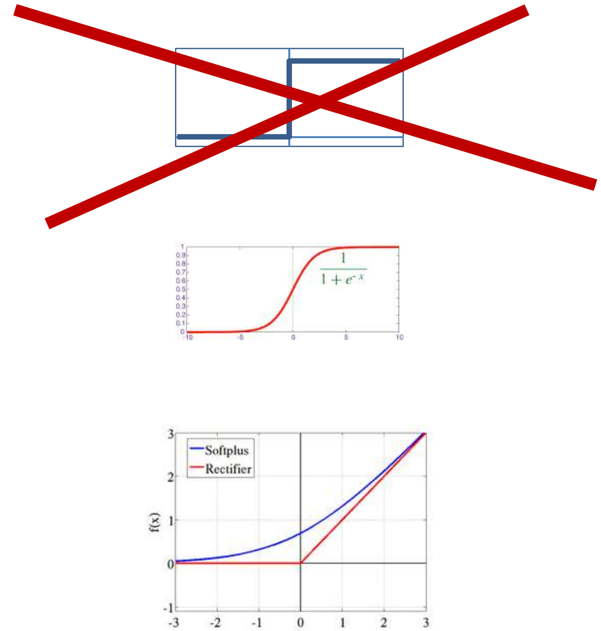
A second problem: What we *actually* model



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

Solution

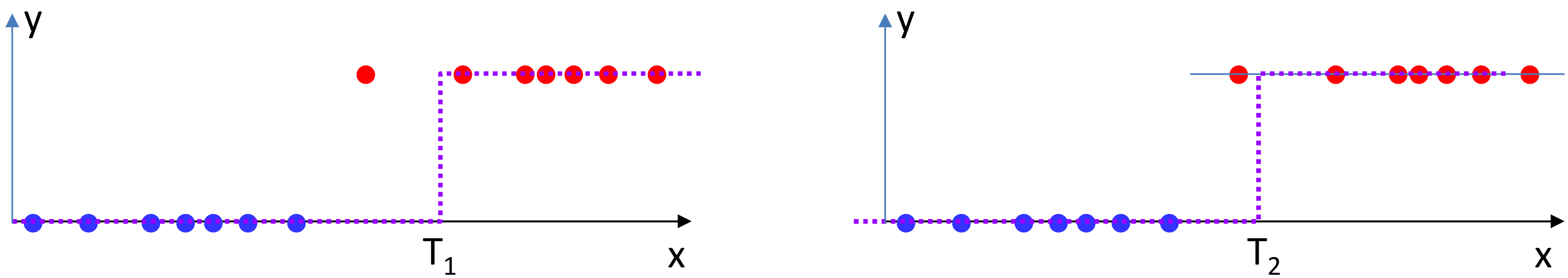




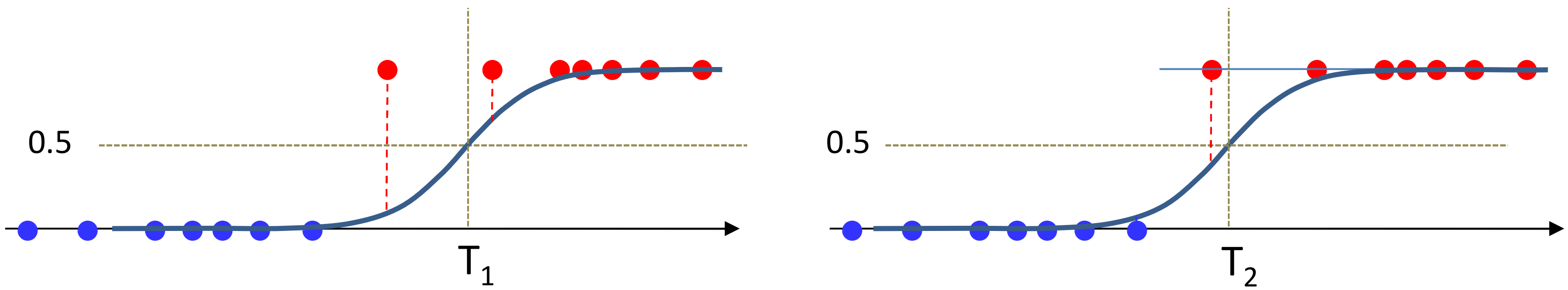
Activation functions $\sigma(z)$

- 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 activation function

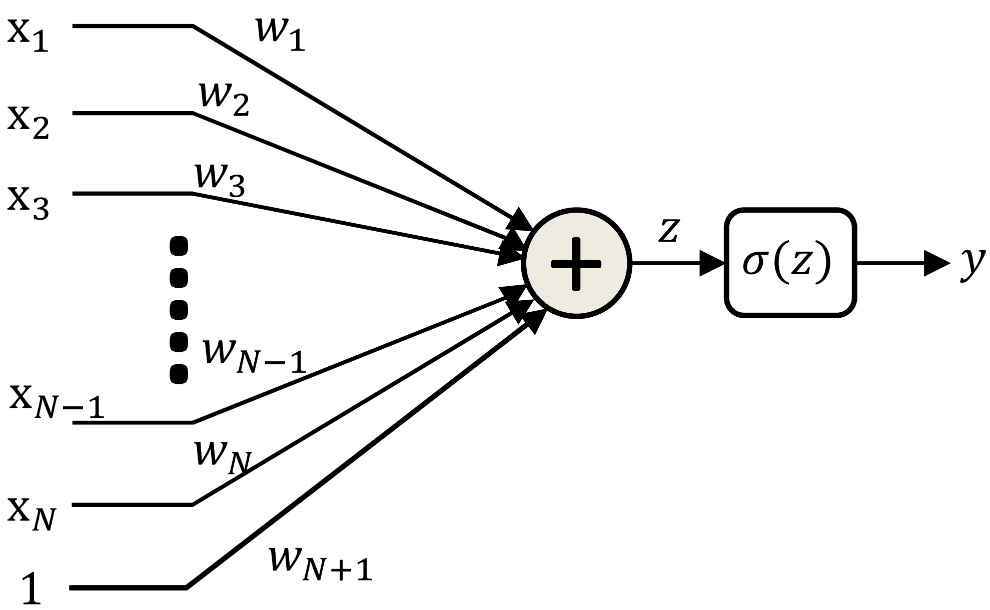


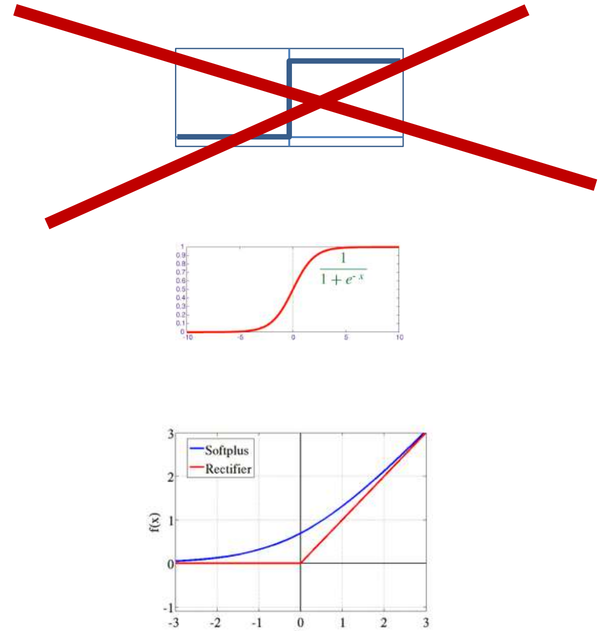
- Threshold activation: shifting the threshold from T_1 to T_2 does not change classification error
  - Does not indicate if moving the threshold left was good or not



- Smooth, continuously varying activation: Classification based on whether the output is greater than 0.5 or less
  - Can now quantify *how much* the output differs from the desired target value (0 or 1)
  - Moving the function left or right changes this quantity, even if the classification error itself doesn't change

Continuous Activations



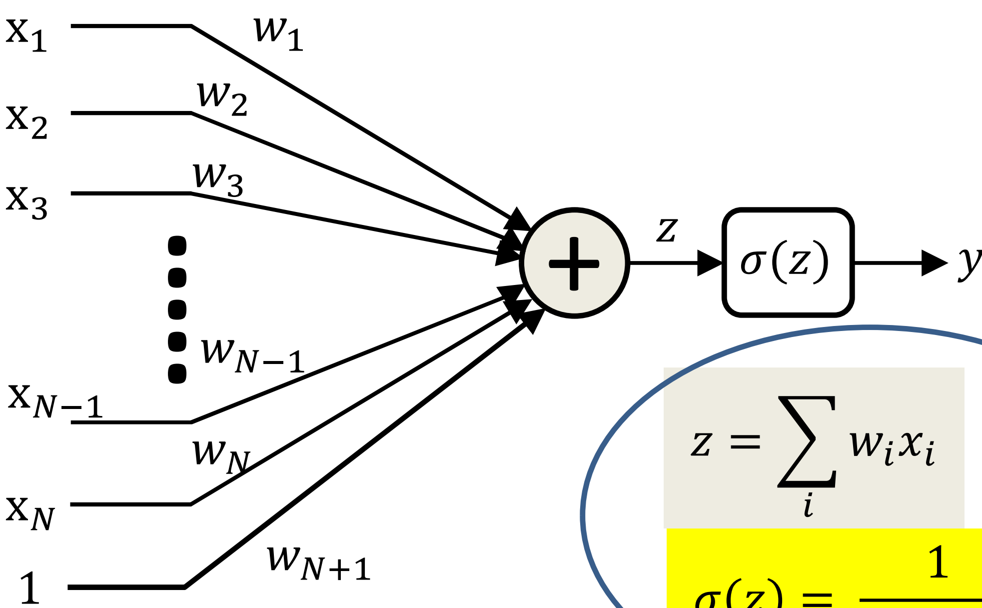


Activation functions $\sigma(z)$

- Replace the threshold activation with continuous graded activations
  - E.g. RELU, softplus, sigmoid etc.

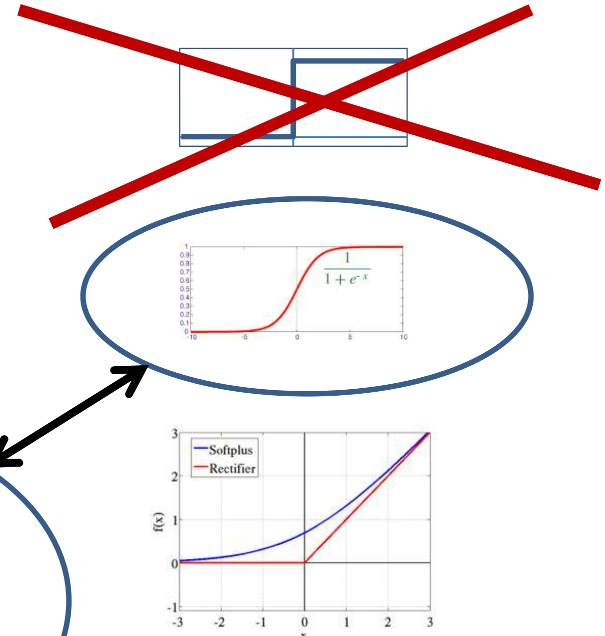
- The activations are *differentiable* almost everywhere
  - Have derivatives almost everywhere
  - And have “subderivatives” at non-differentiable corners
    - Bounds on the derivative that can substitute for derivatives in our setting
    - More on these later

The sigmoid activation is special



$$z = \sum_i w_i x_i$$

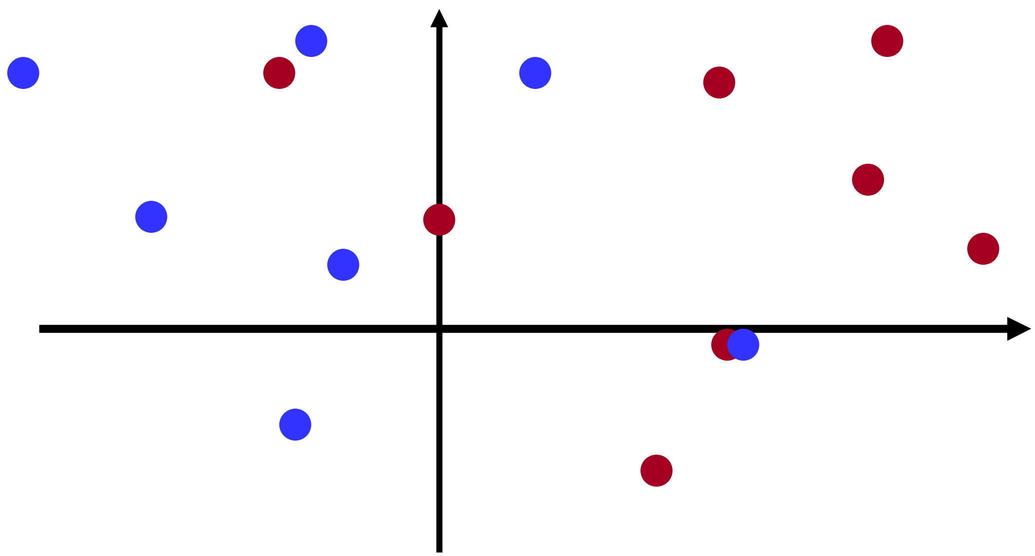
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

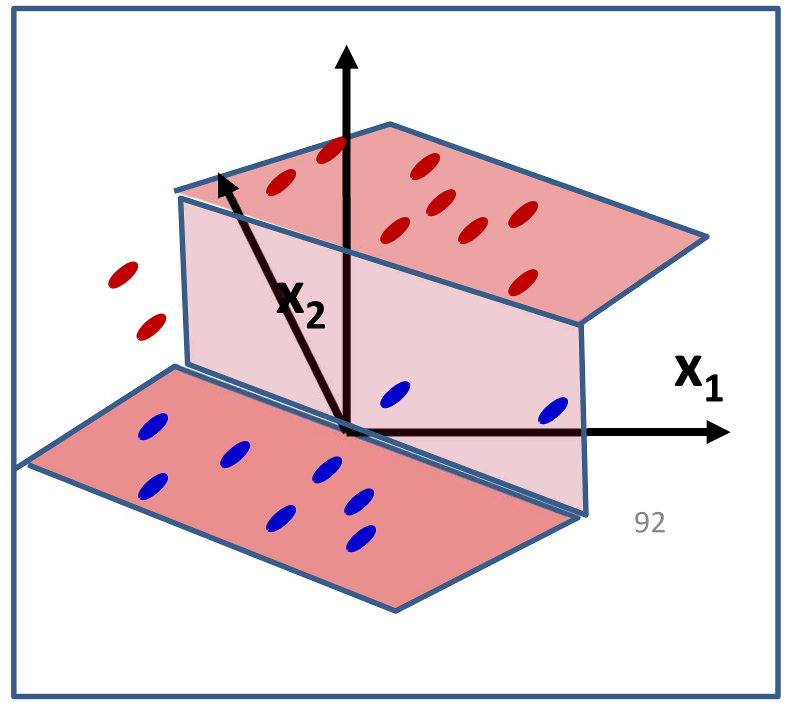


Activation functions $\sigma(z)$

- This particular one has a nice interpretation
- It can be interpreted as $P(y = 1|x)$

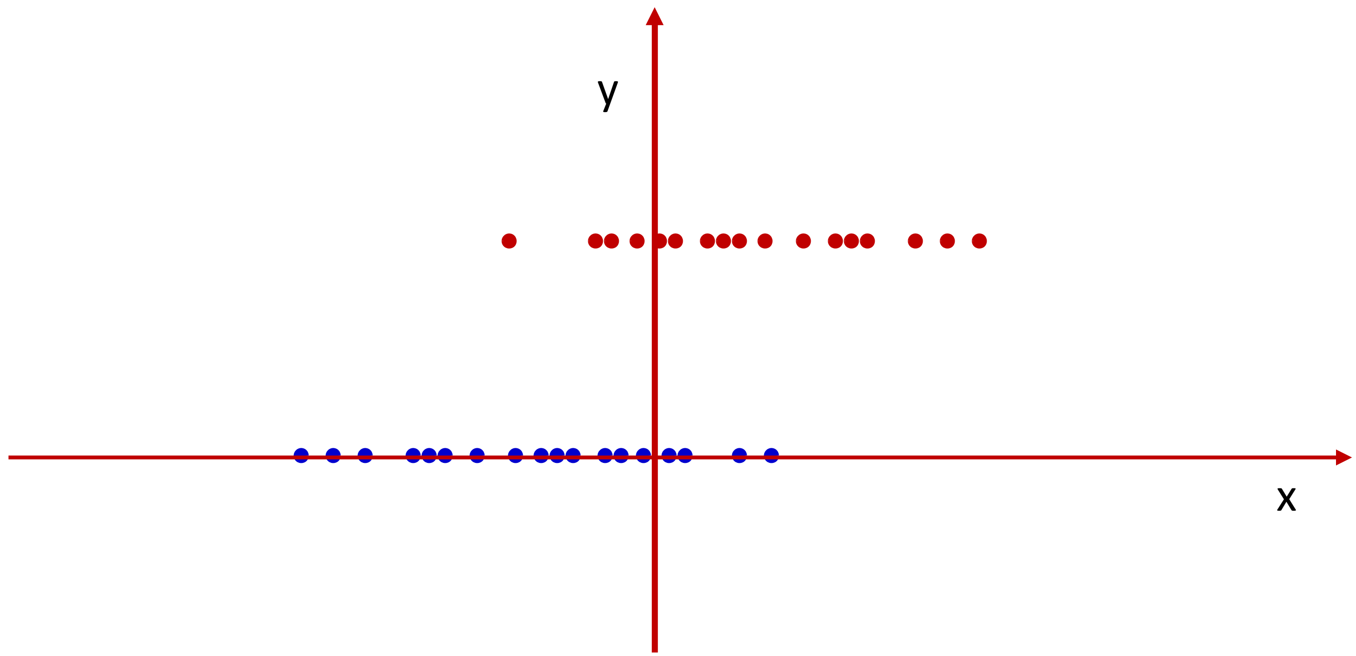
Non-linearly separable data





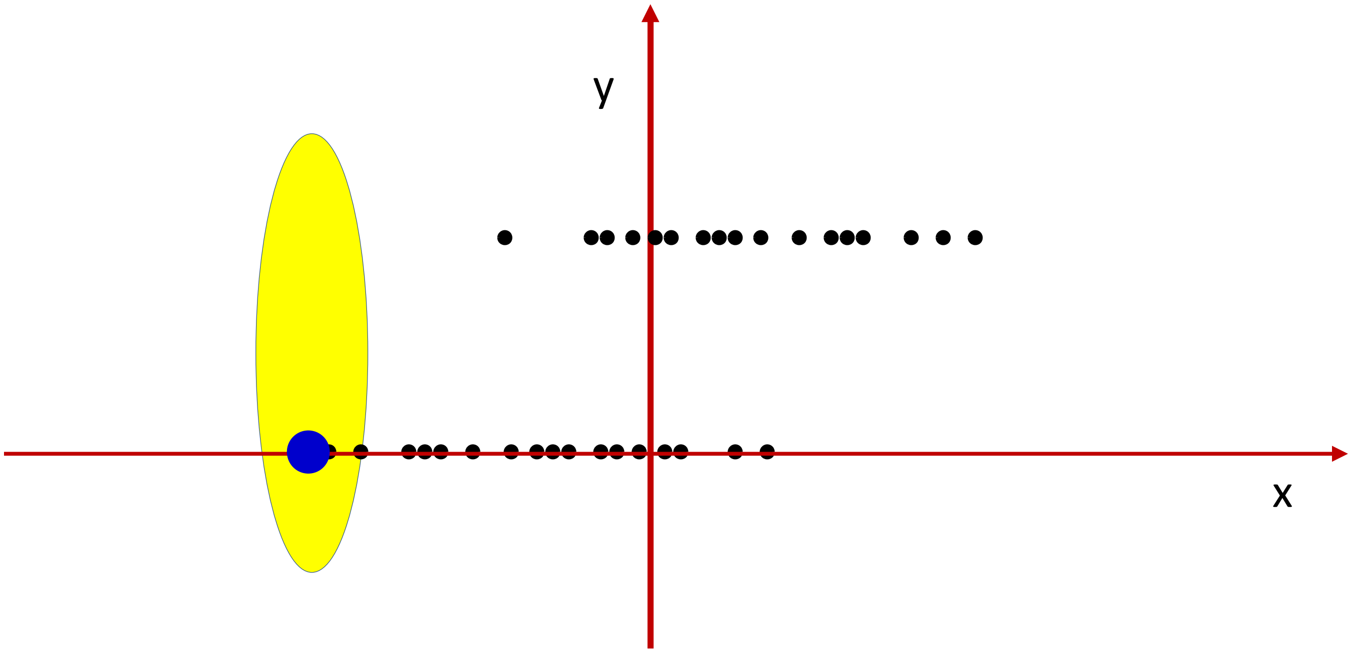
- Two-dimensional example
  - Blue dots (on the floor) on the "red" side
  - Red dots (suspended at Y=1) on the "blue" side
  - No line will cleanly separate the two colors

Non-linearly separable data: 1-D example



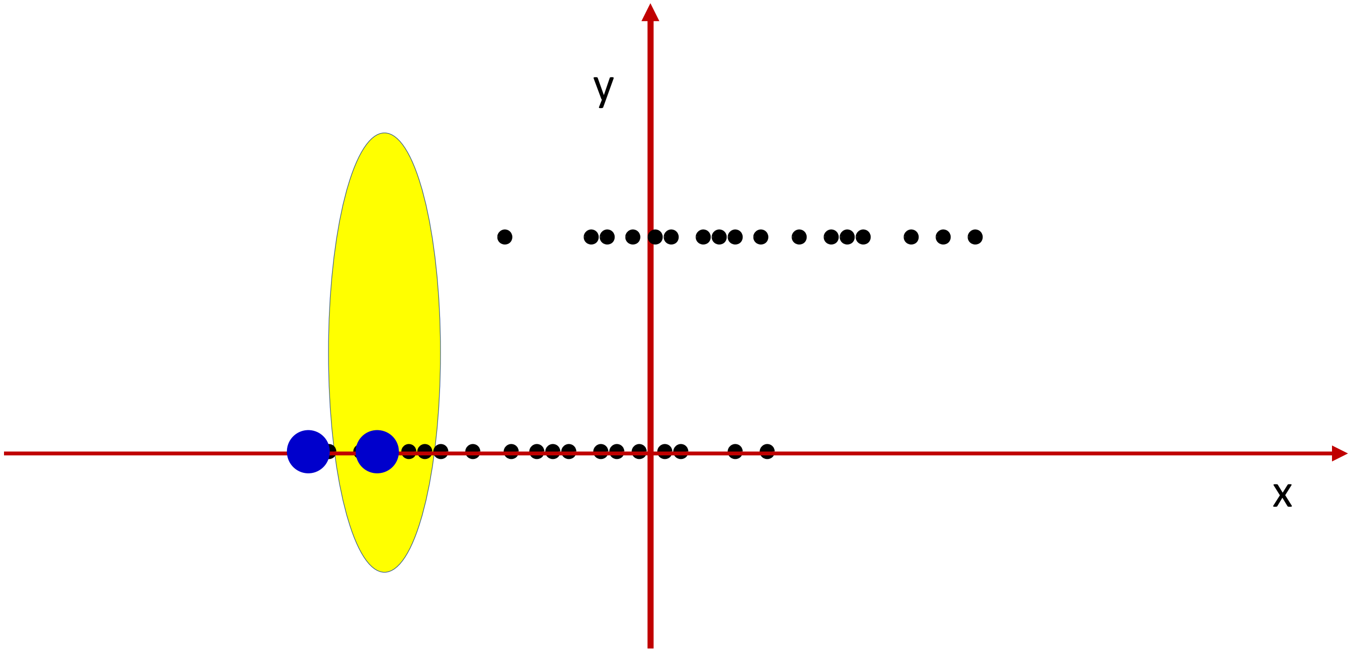
- One-dimensional example for visualization
  - All (red) dots at Y=1 represent instances of class Y=1
  - All (blue) dots at Y=0 are from class Y=0
  - The data are not linearly separable
    - In this 1-D example, a linear separator is a threshold
    - No threshold will cleanly separate red and blue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

The *probability* of y=1



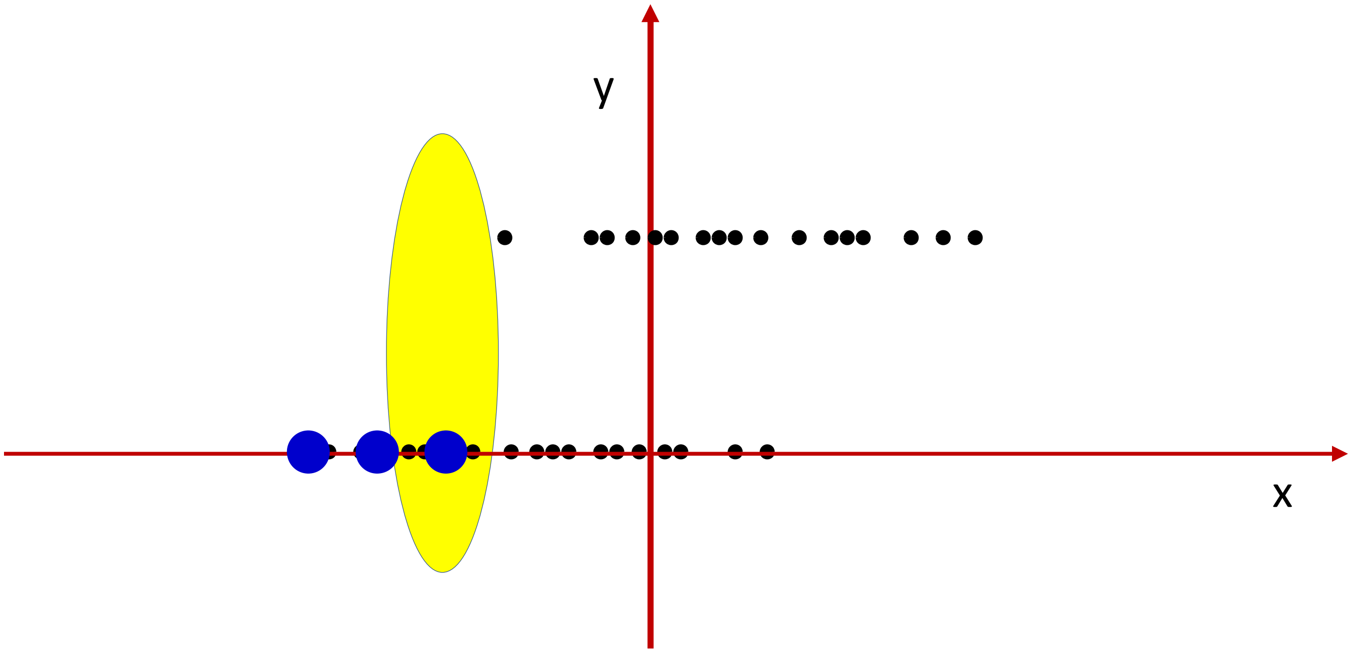
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
  - This is an approximation of the *probability* of Y=1 at that point

The *probability* of y=1



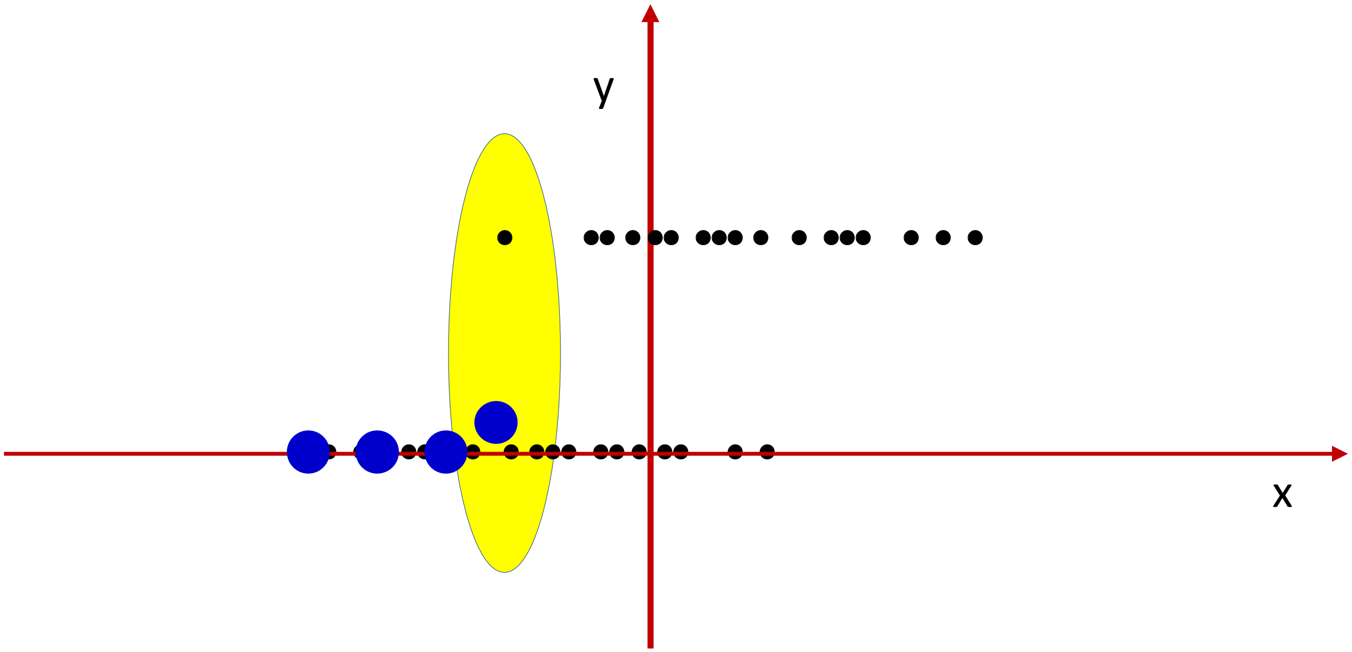
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
  – This is an approximation of the *probability* of 1 at that point

The *probability* of y=1



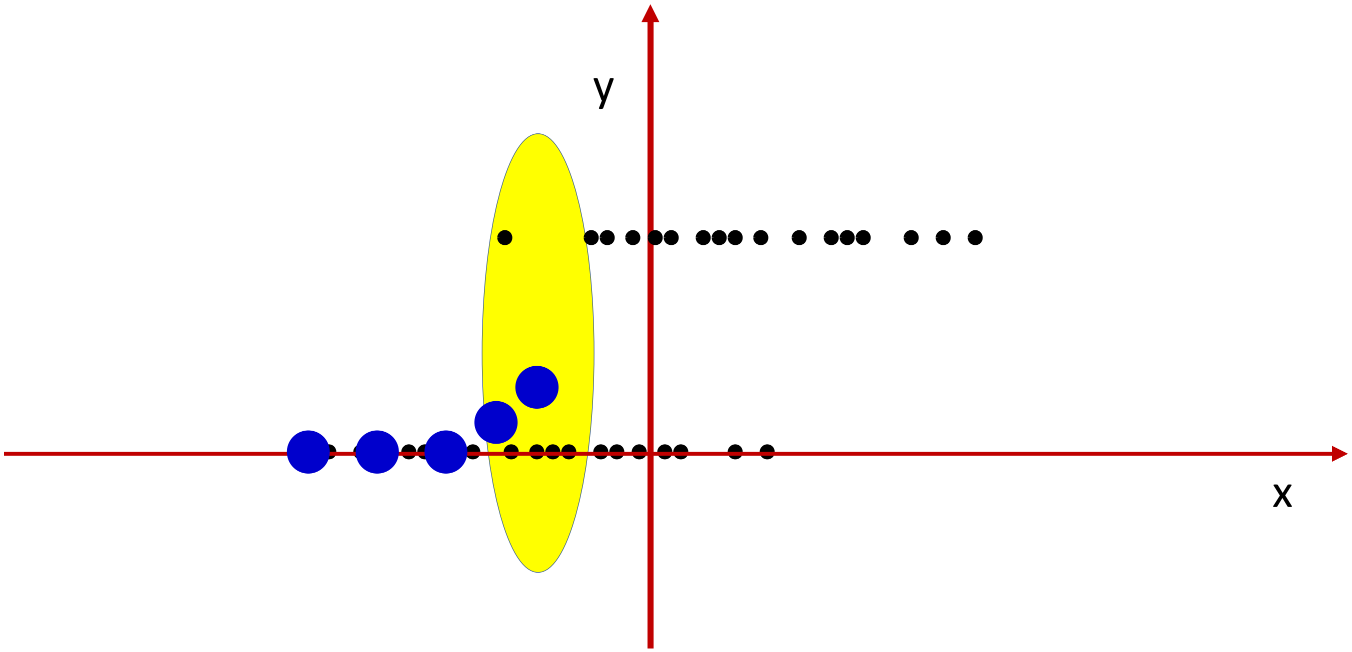
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
  - This is an approximation of the *probability* of 1 at that point

The *probability* of y=1



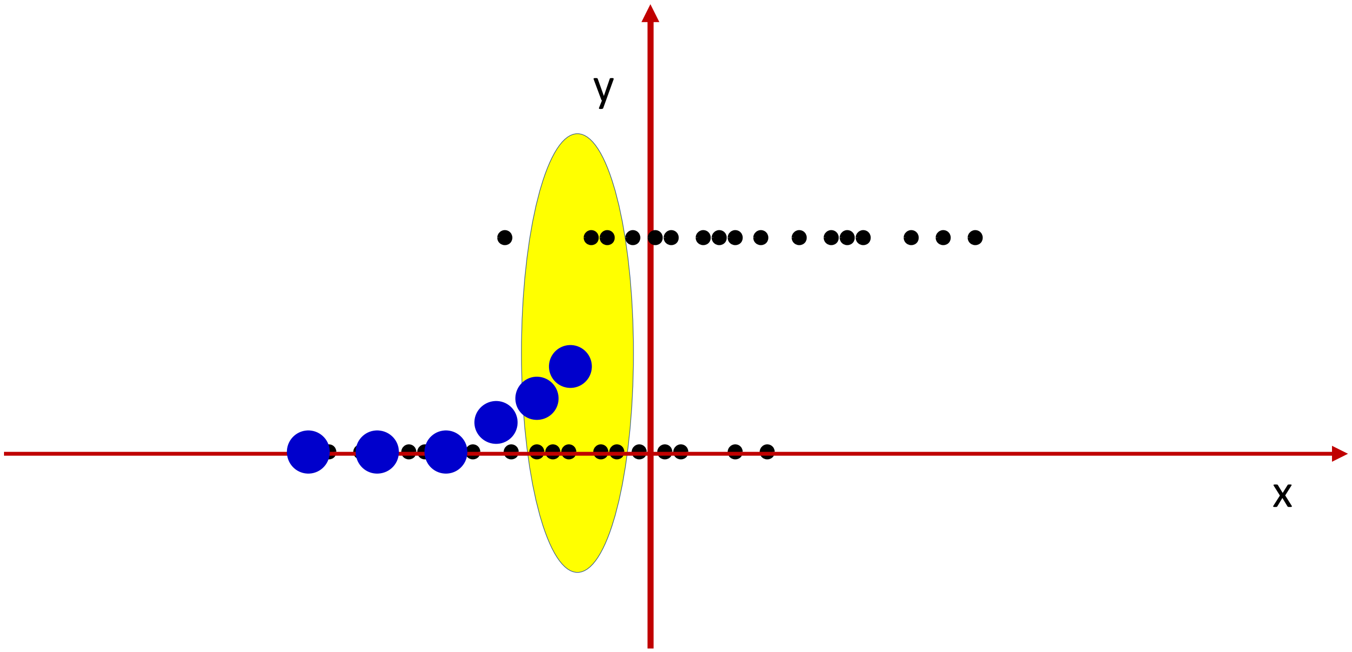
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
  - This is an approximation of the *probability* of 1 at that point

The *probability* of y=1



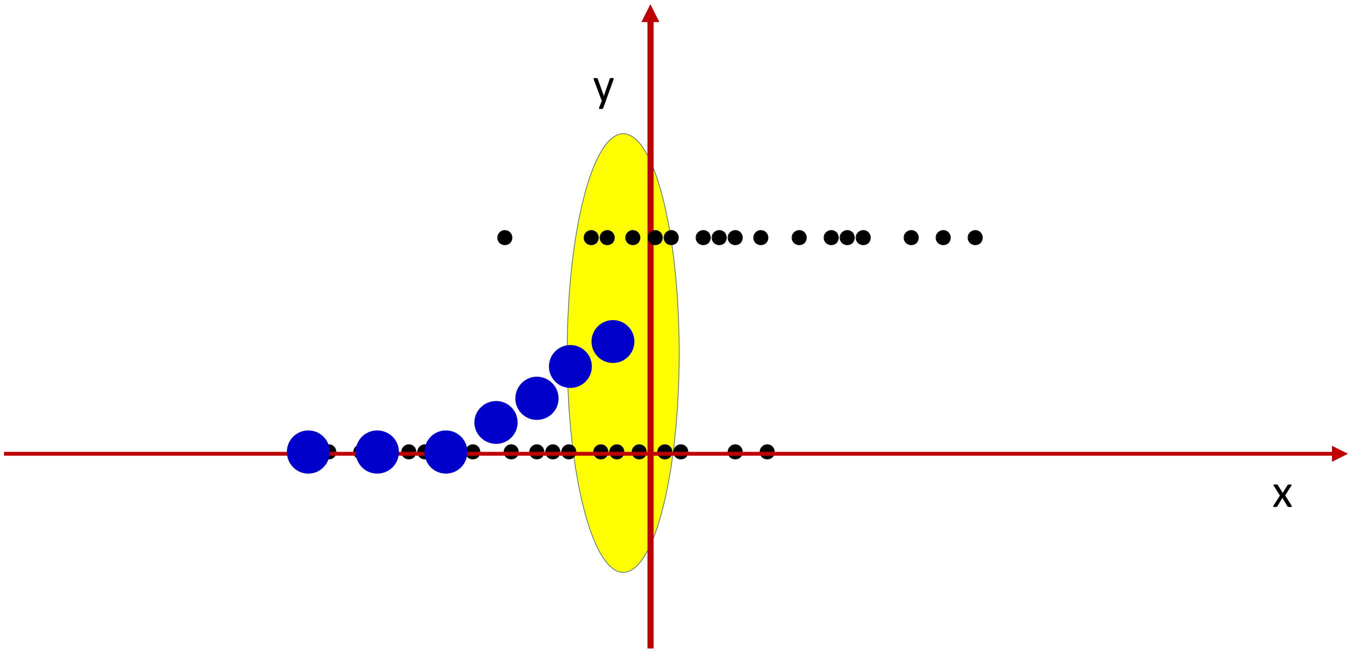
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
  - This is an approximation of the *probability* of 1 at that point

The *probability* of y=1



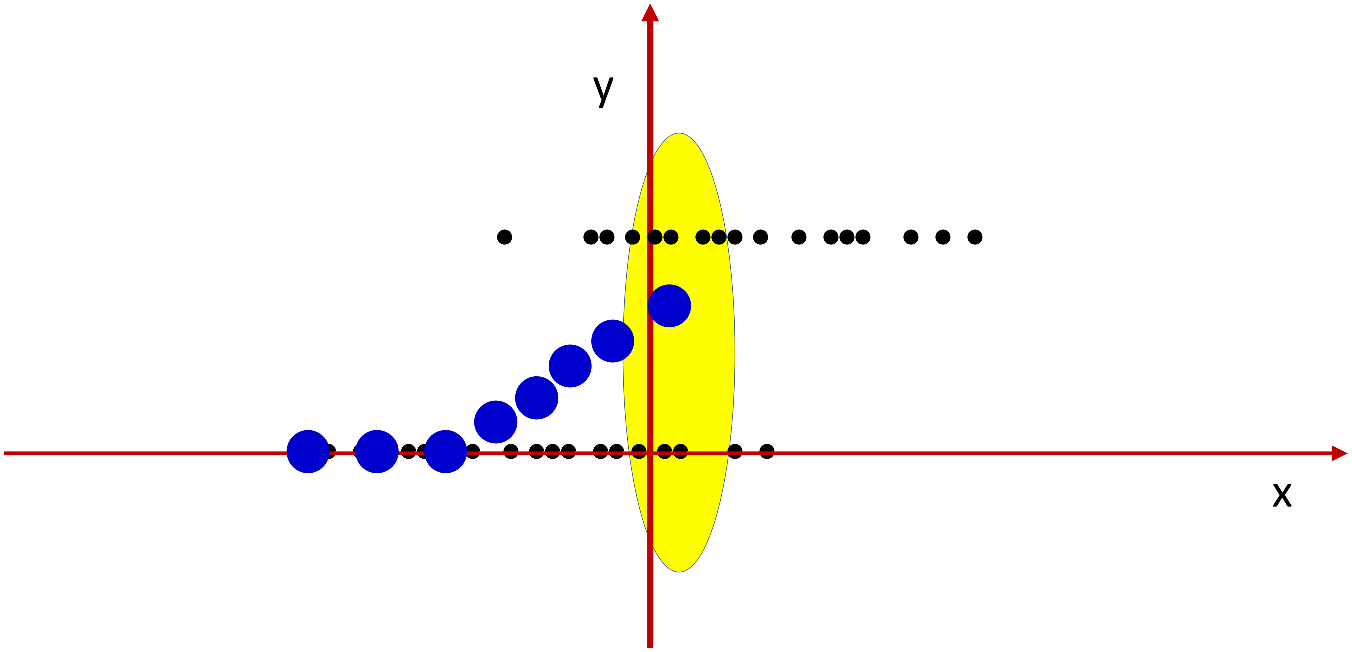
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
  - This is an approximation of the *probability* of 1 at that point

The *probability* of y=1



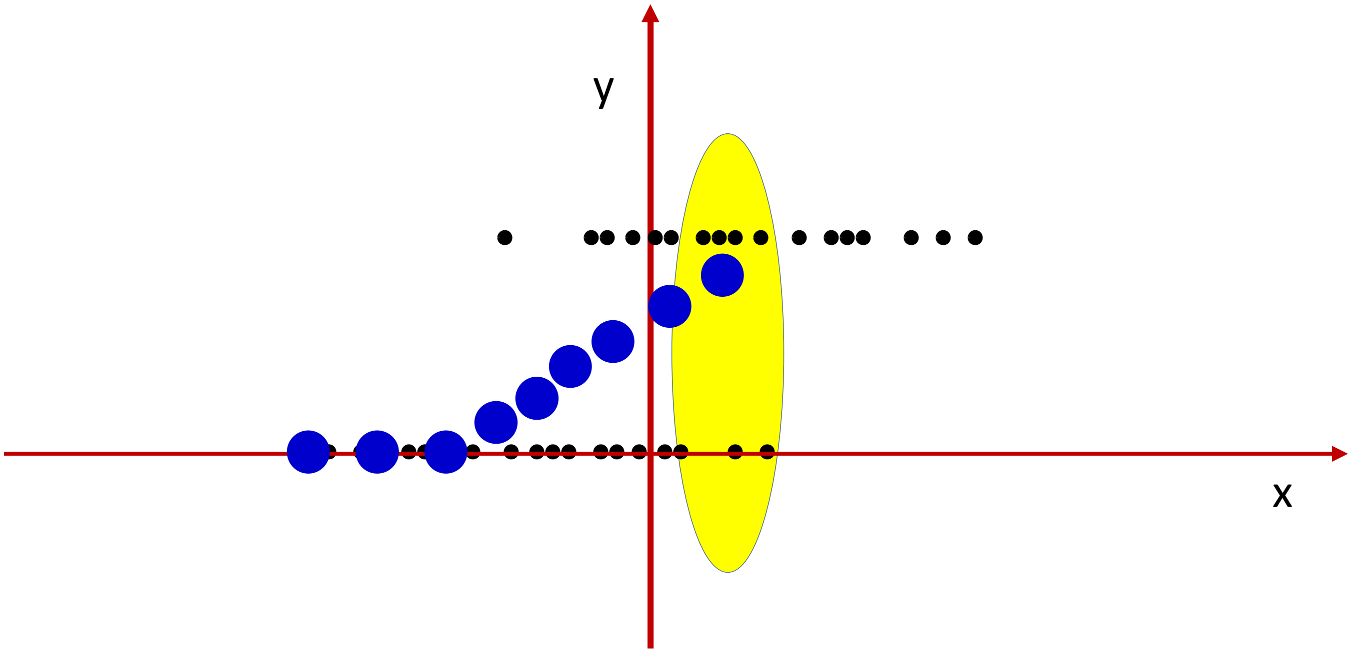
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
  - This is an approximation of the *probability* of 1 at that point

The *probability* of y=1



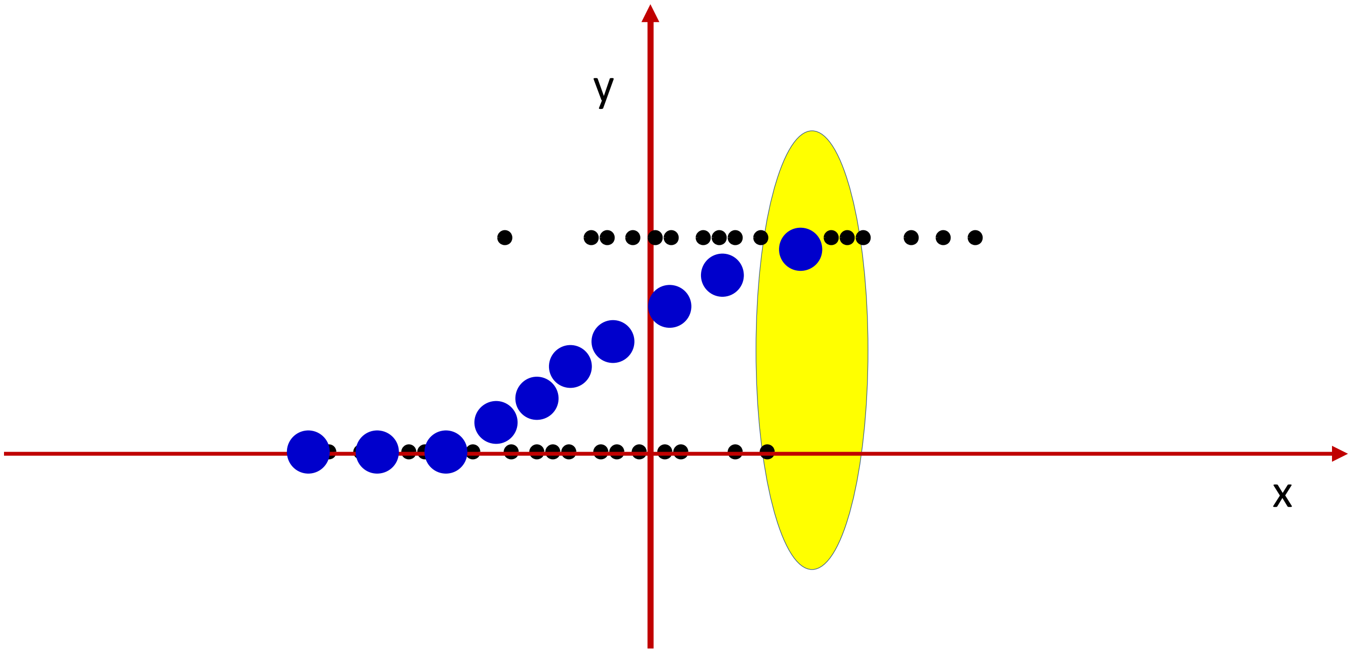
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
  - This is an approximation of the *probability* of 1 at that point

The *probability* of y=1



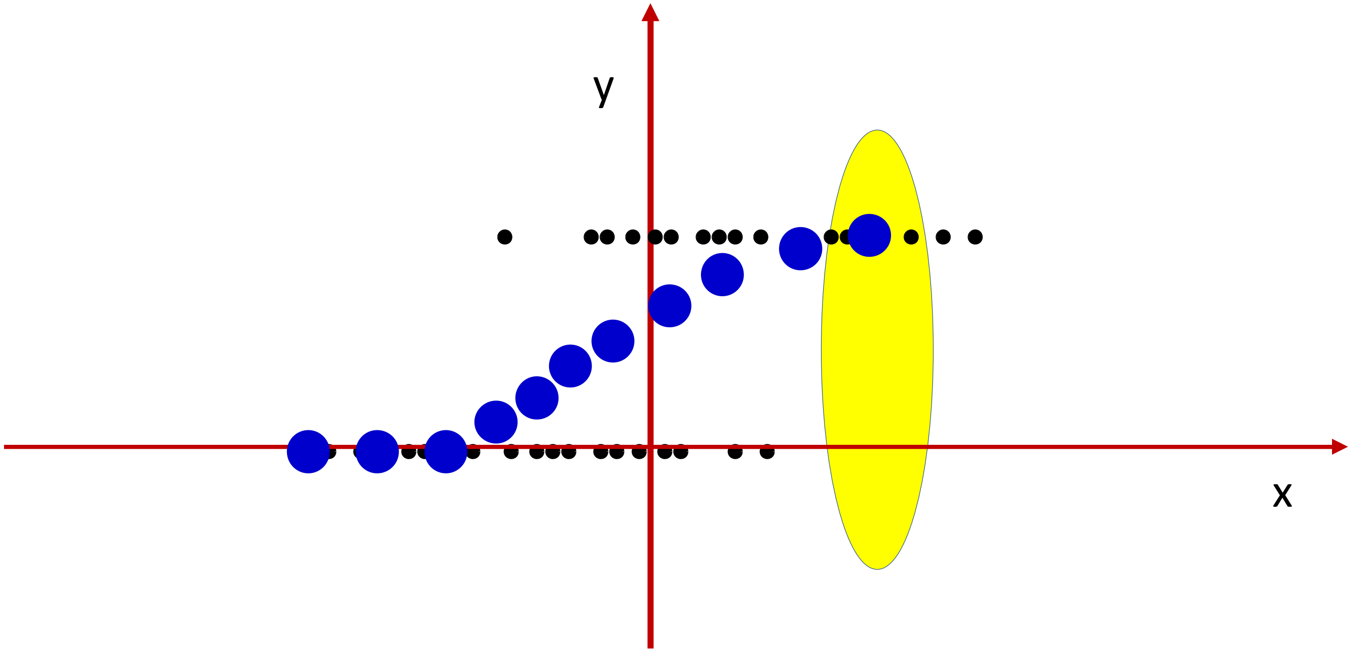
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
  - This is an approximation of the *probability* of 1 at that point

The *probability* of y=1



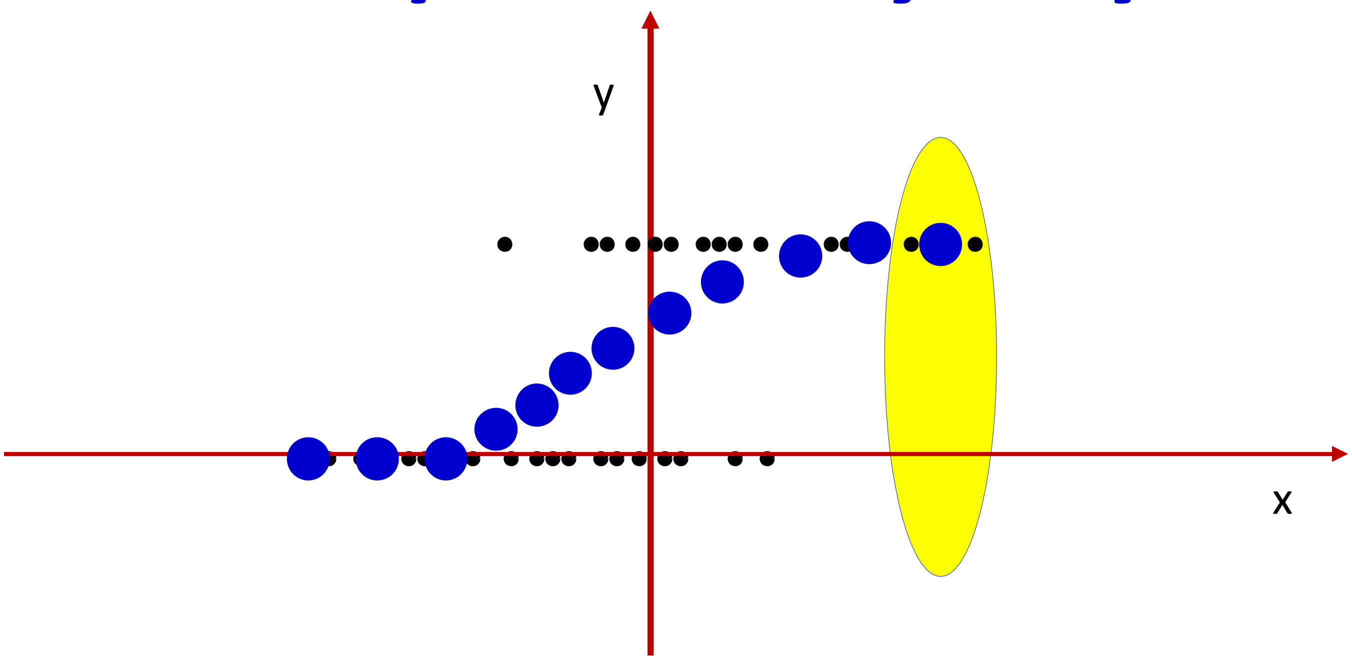
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
  - This is an approximation of the *probability* of 1 at that point

The *probability* of y=1



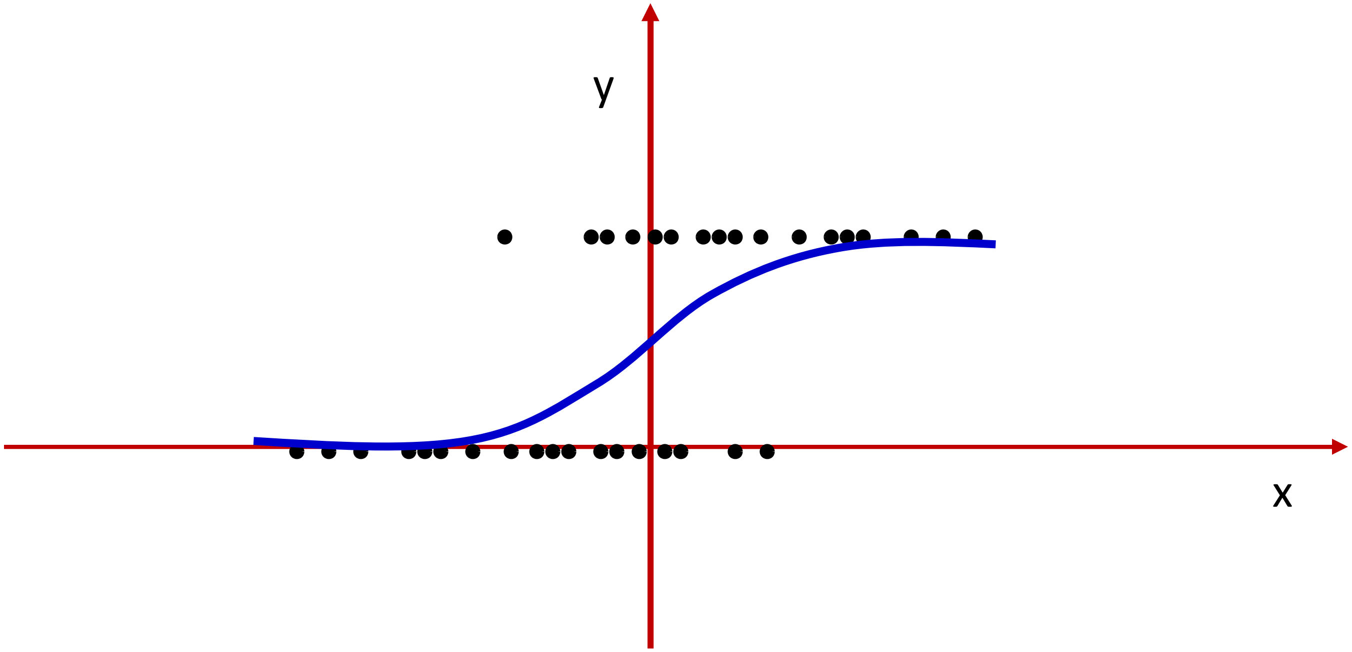
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
  - This is an approximation of the *probability* of 1 at that point

The *probability* of y=1



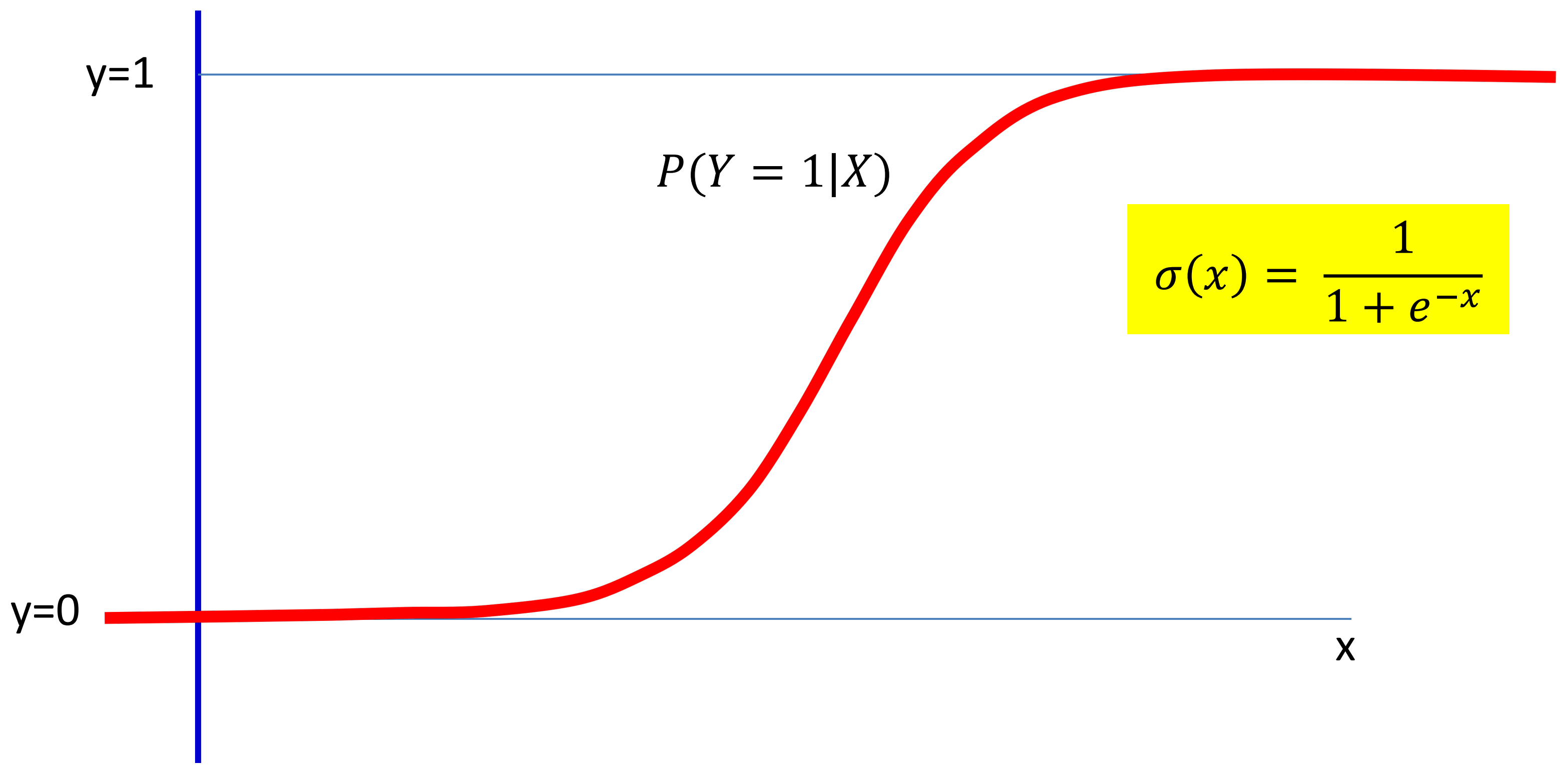
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
  - This is an approximation of the *probability* of 1 at that point

The *probability* of y=1



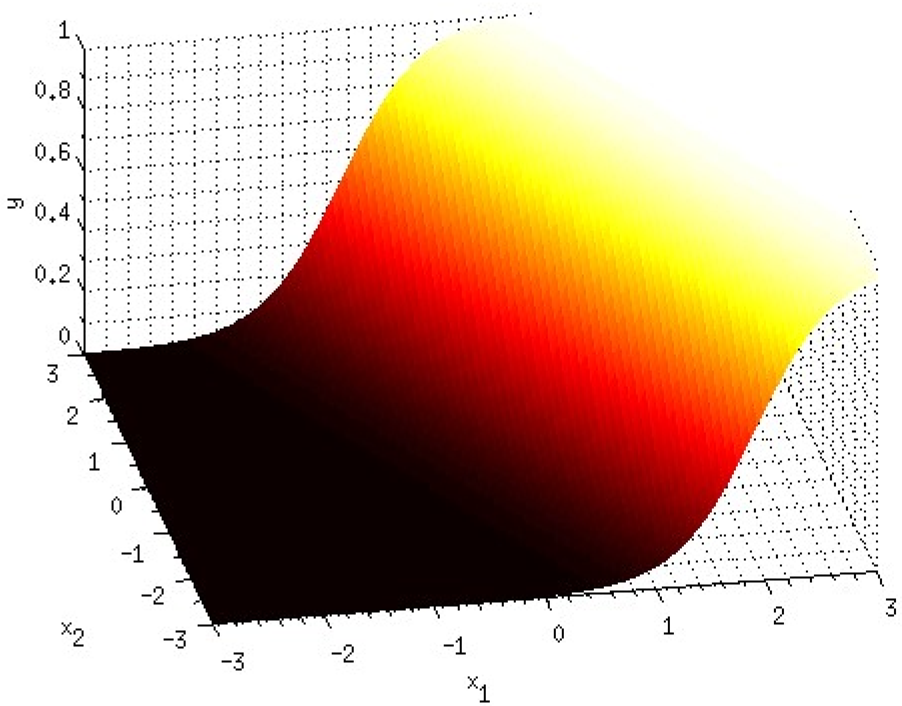
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
  - This is an approximation of the *probability* of 1 at that point

The logistic regression model

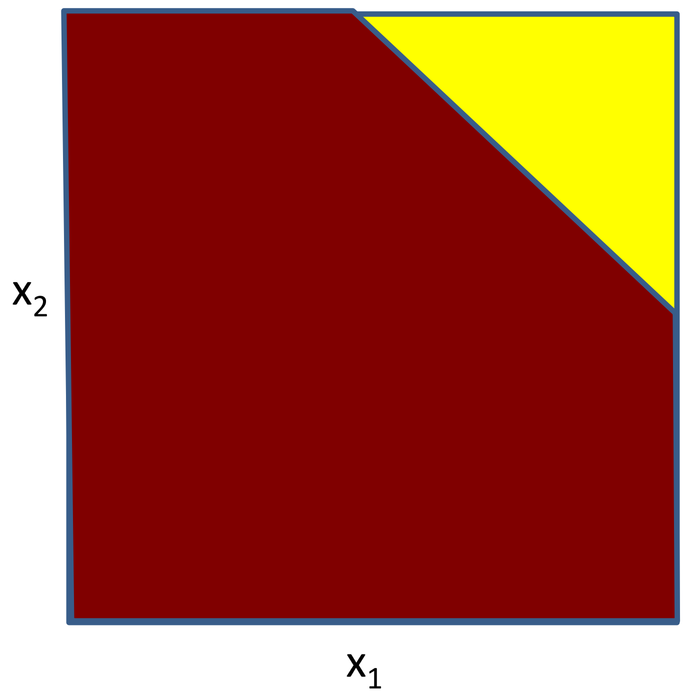


- Class 1 becomes increasingly probable going left to right
  - Very typical in many problems

Logistic regression



Decision: y > 0.5?



When X is a 2-D variable

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

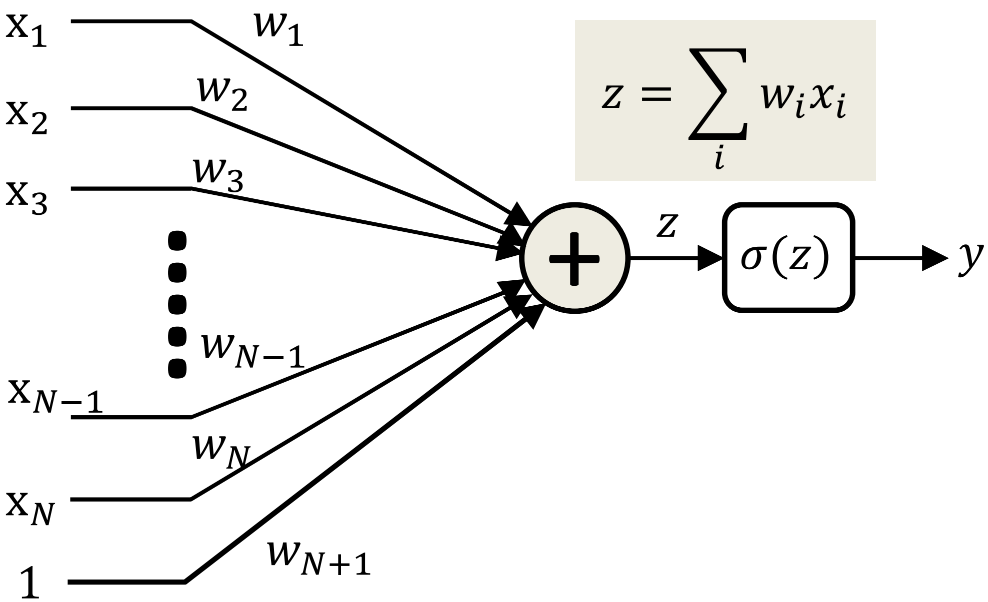
- This the perceptron with a sigmoid activation
  - It actually computes the *probability* that the input belongs to class 1

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



$$z = \sum_i w_i x_i$$

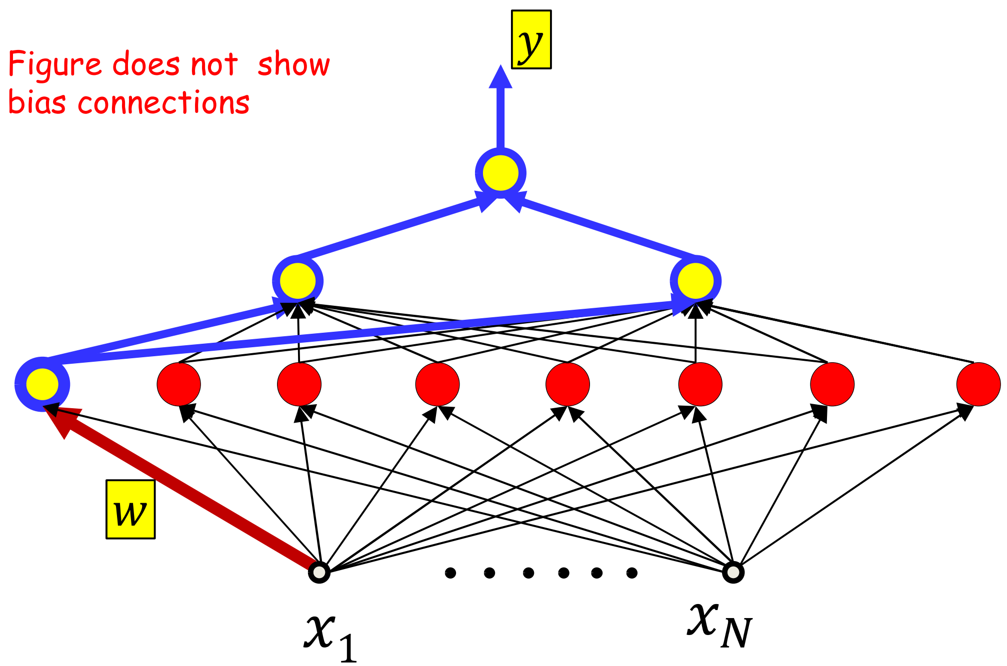
$$\frac{dy}{dz} = \sigma'(z)$$

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

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

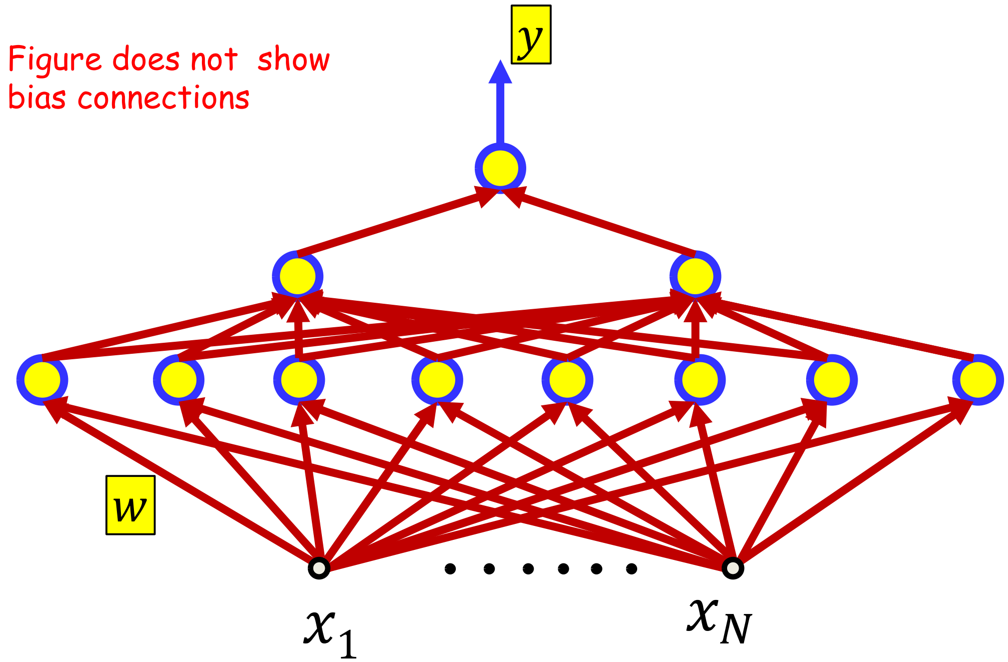
- **$\sigma(z)$is a differentiable function of z**
  - $\frac{d\sigma(z)}{dz}$ is well-defined and finite for all z
- **Using the chain rule, y is a differentiable function of both inputs x_i and weights w_i**
- **This means that we can compute the change in the output for *small* changes in either the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

Overall network is differentiable



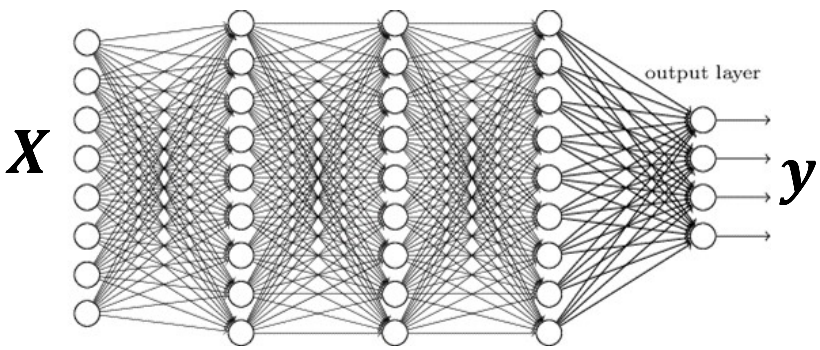
- Every individual perceptron is differentiable w.r.t its inputs and its weights (including "bias" weight)
  - Small changes in the parameters result in measurable changes in output
- Using the chain rule can compute how small perturbations of a parameter change the output of the network
  - The network output is differentiable with respect to the parameter

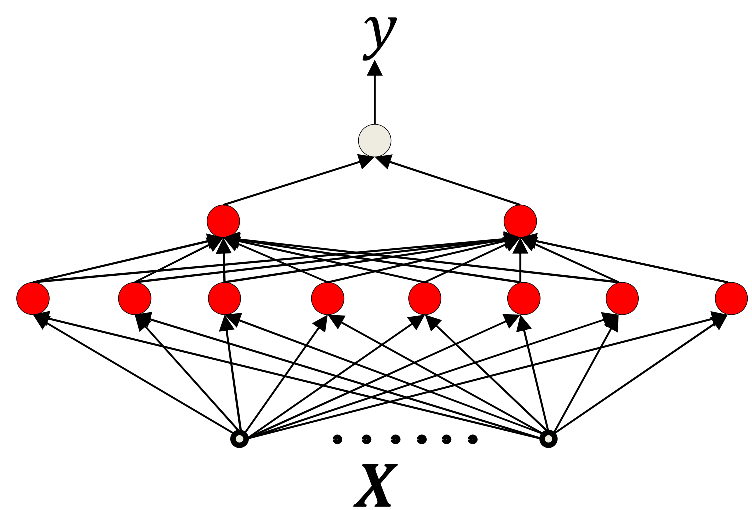
Overall function is differentiable



- By extension, the overall function is differentiable w.r.t every parameter in the network
  - We can compute how small changes in the parameters change the output
    - For non-threshold activations the derivative are finite and generally non-zero
- We will derive the actual derivatives using the chain rule later

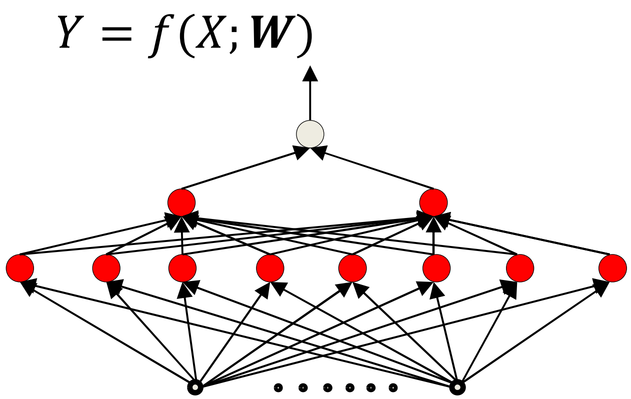
Overall setting for “Learning” the MLP

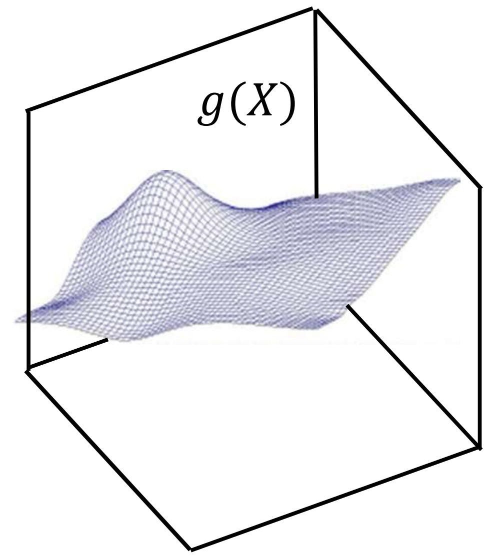




- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \dots, (\boldsymbol{X}_N, \boldsymbol{d}_N)$ …
  - $\boldsymbol{d}$ is the *desired output* of the network in response to $\boldsymbol{X}$
  - $\boldsymbol{X}$ and $\boldsymbol{d}$ 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



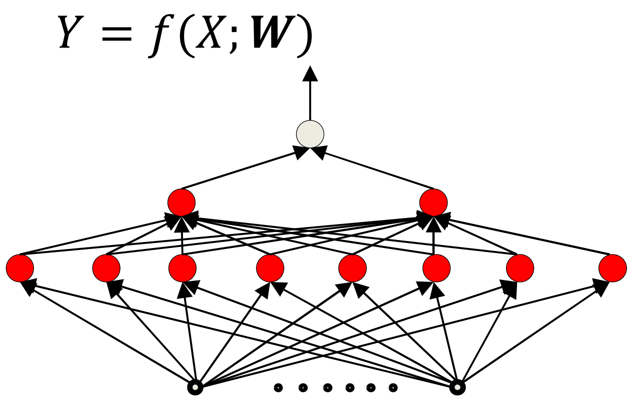


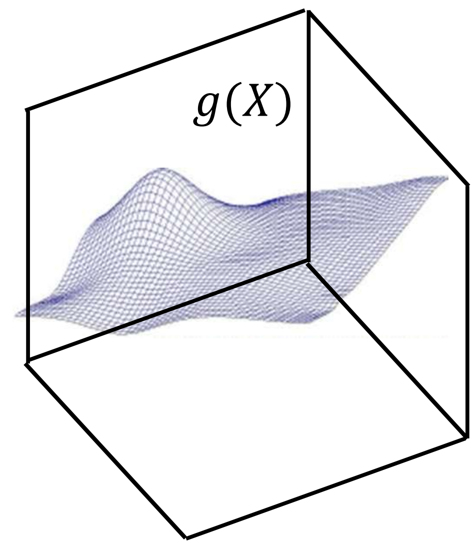
- When $f(X; \boldsymbol{W})$ has the capacity to exactly represent $g(X)$

$$\widehat{\boldsymbol{W}} = \underset{W}{\operatorname{argmin}} \int_X div\big(f(X; W), g(X)\big) dX$$

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

Minimizing *expected* divergence



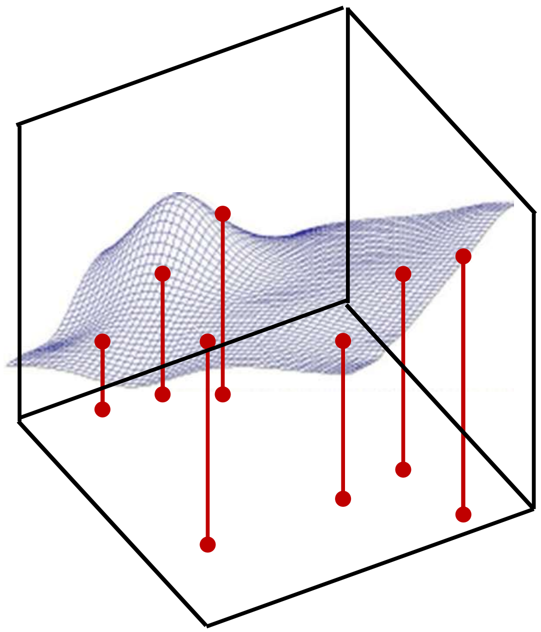


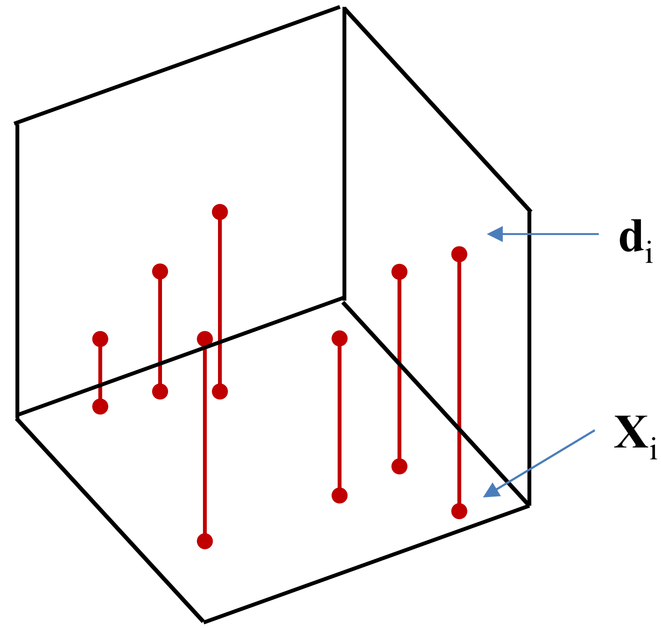
- More generally, assuming X is a random variable

$$\widehat{\boldsymbol{W}} = \underset{W}{\operatorname{argmin}} \int_X div\big(f(X;W), g(X)\big)P(X)dX$$

$$= \underset{W}{\operatorname{argmin}}\, E\big[div\big(f(X;W), g(X)\big)\big]$$

Recap: Sampling the function

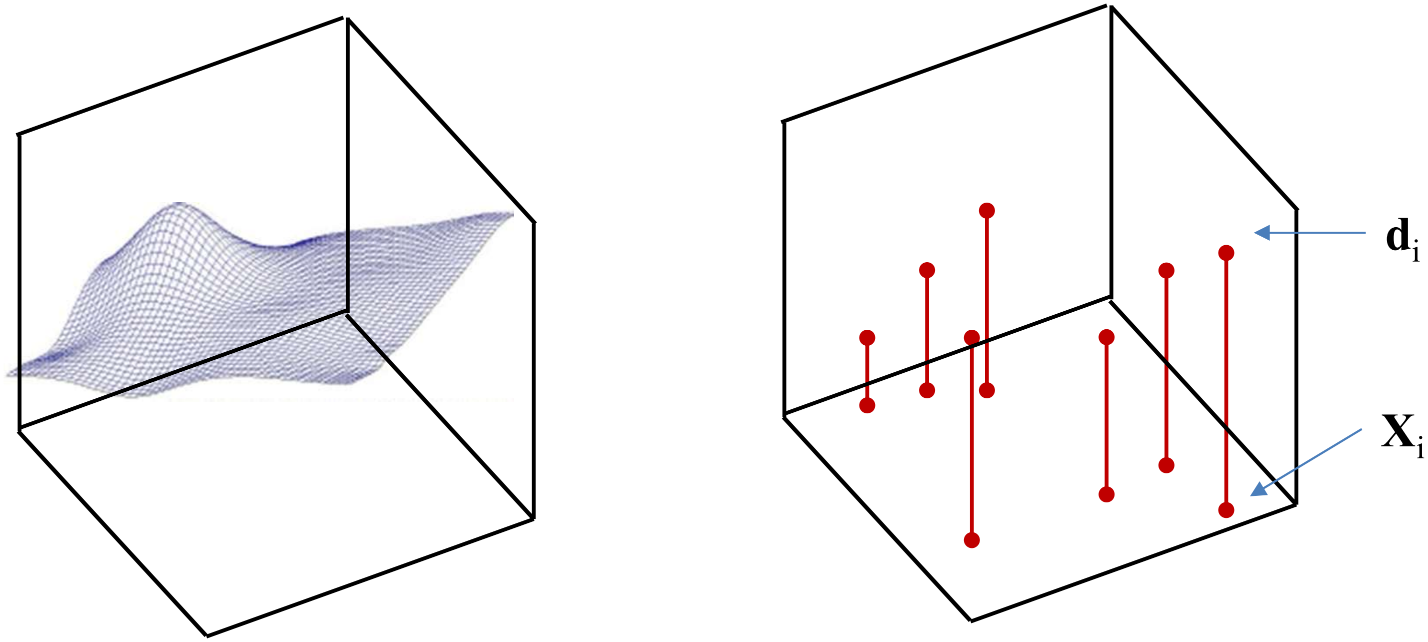




- *We don't have g(X) so sample* $g(X)$
  - Obtain input-output pairs for a number of samples of input X_i
  - Good sampling: the samples of X will be drawn from $P(X)$

- Estimate function from the samples

The *Empirical* risk



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

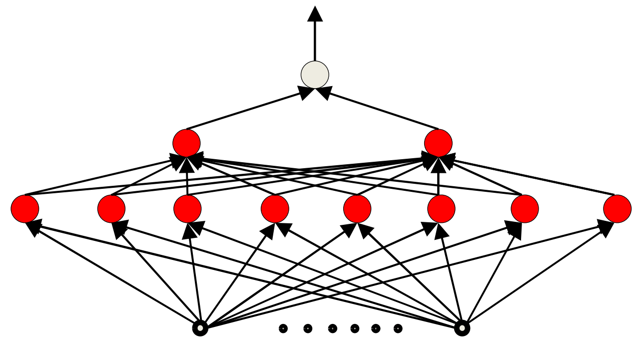
$$E\left[div\left(f(X;W),g(X)\right)\right] = \int_X div\left(f(X;W),g(X)\right)P(X)dX$$

- The *empirical estimate* of the expected risk is the *average* divergence over the samples

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

Empirical Risk Minimization

$Y = f(X; \boldsymbol{W})$



- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \dots, (\boldsymbol{X}_N, \boldsymbol{d}_N)$
  - Quantification of error on the ith instance: $div(f(X_i; W), d_i)$
  - Empirical average divergence (Empirical Risk) on all training data:

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

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

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

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

Empirical Risk Minimization

$$Y = f(X; \boldsymbol{W})$$

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 $E\left[div\left(f(X;W), g(X)\right)\right]$ which is our *actual* minimization objective

Note 3: For a *given training set* the loss is only a function of W

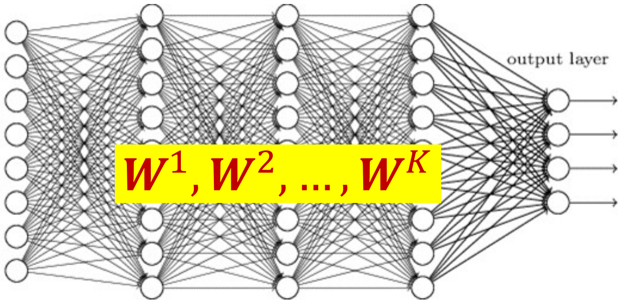
$$Loss(W) = \frac{1}{N}\sum_i div(f(X_i;W), d_i)$$

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

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

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

ERM for neural networks



Actual output of network:

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

Desired output of network: d_i

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

Average training error(loss):

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

– What is the exact form of 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 $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_N, \boldsymbol{d}_N)$
- Minimize the following function

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

w.r.t W

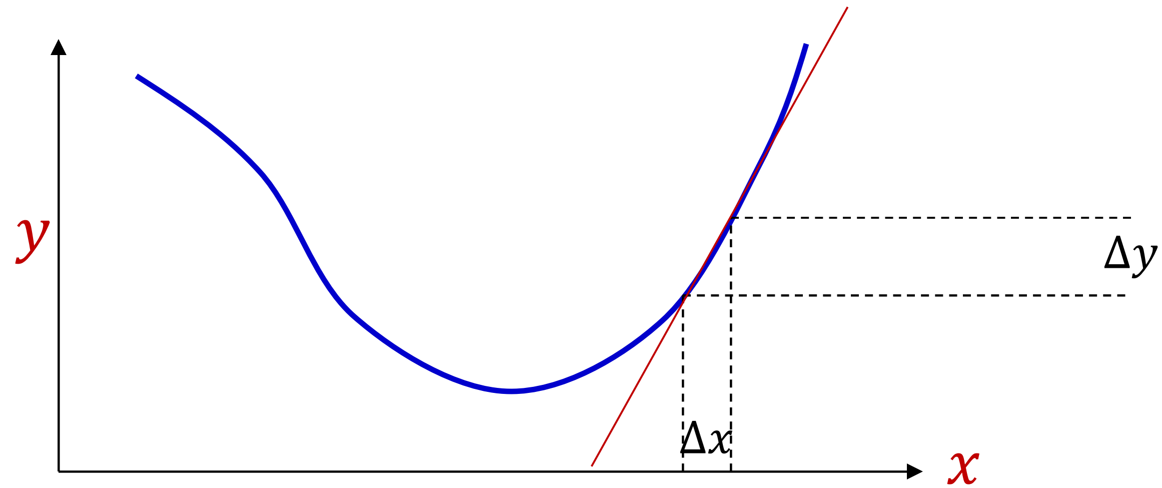
- This is problem of function minimization
  - An instance of optimization

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**
  - **With an initial discussion of derivatives**

A brief note on derivatives..



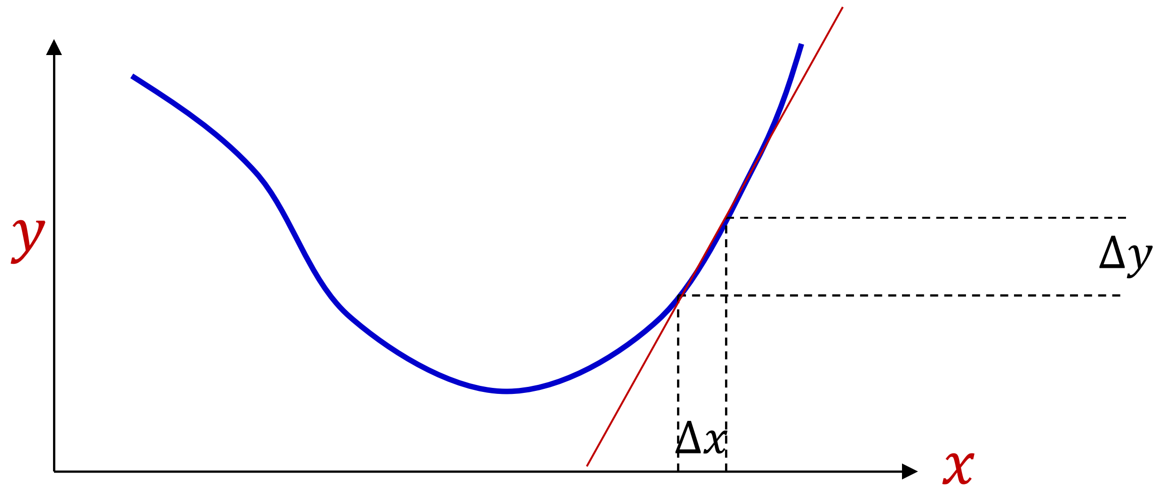
derivative

- 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

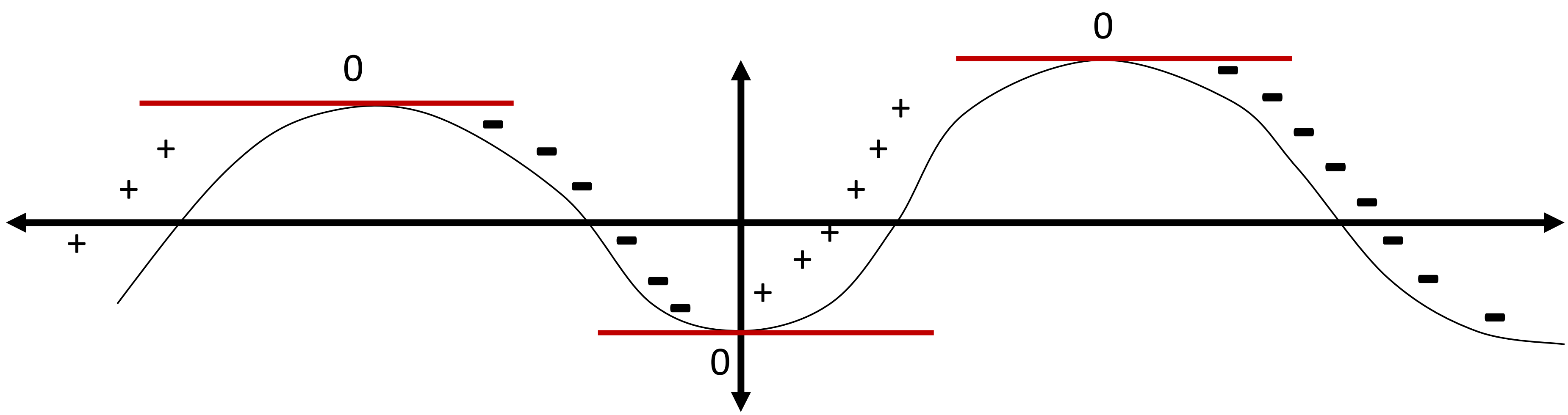
$$y = f(x)$$

  - Derivative:

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

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

Scalar function of scalar argument

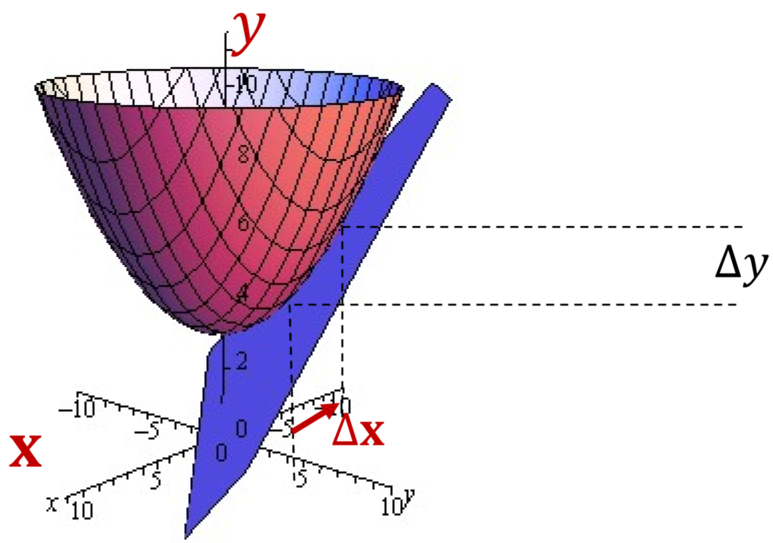


- Derivative $f'(x)$ is the *rate of change* of the function at x
  - How fast it increases with increasing x
  - The magnitude of f'(x) gives you the steepness of the curve at x
    - Larger |f'(x)| → the function is increasing or decreasing more rapidly

- It will be positive where a small increase in x results in an *increase* of f(x)
  - Regions of positive slope
- It will be negative where a small increase in x results in a *decrease* of f(x)
  - Regions of negative slope

- It will be 0 where the function is locally flat (neither increasing nor decreasing)

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: $\alpha = [\alpha_1 \quad \cdots \quad \alpha_D]$

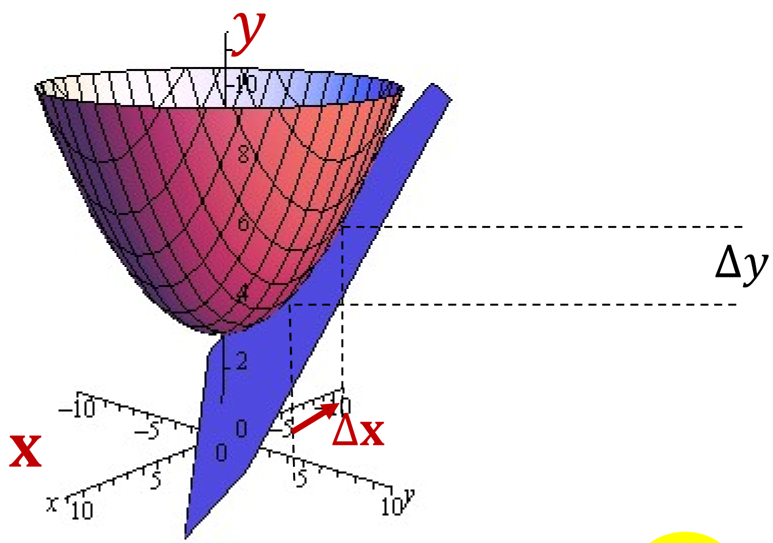
$$\Delta y = \alpha_1 \Delta x_1 + \alpha_2 \Delta x_2 + \cdots + \alpha_D \Delta x_D$$

- 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

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

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

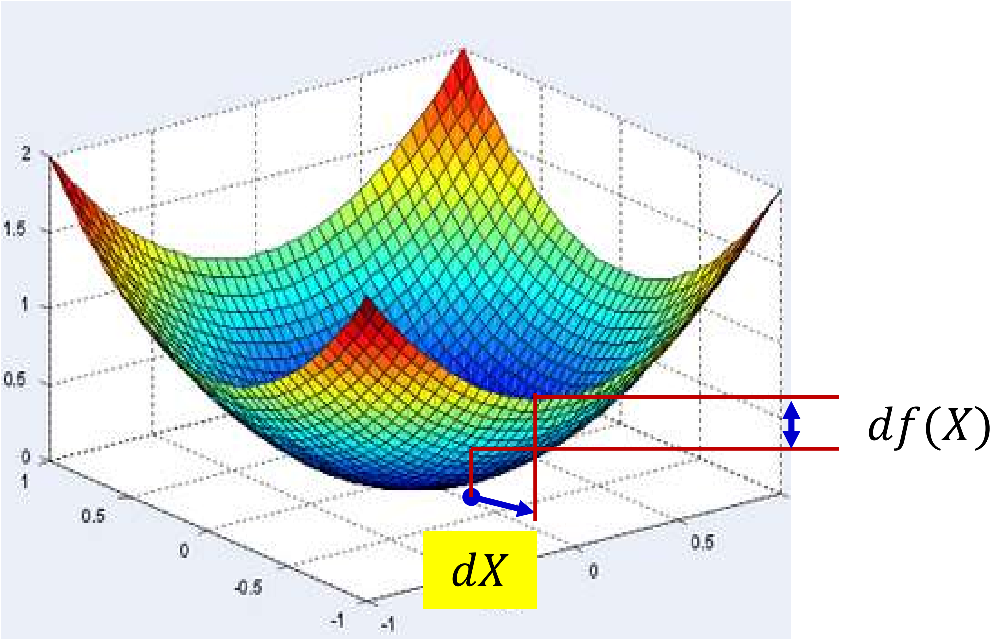
We will be using this symbol for vector and matrix derivatives

- Where

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

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

Gradient of a scalar function of a vector



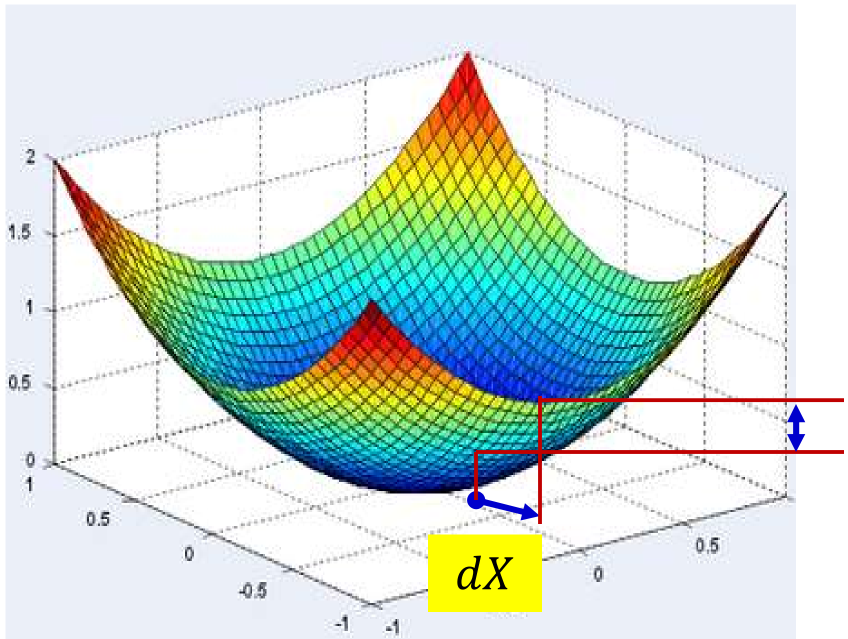
- The *derivative* $\nabla_X f(X)$ of a scalar function $f(X)$ of a multi-variate input X is a multiplicative factor that gives us the change in $f(X)$ for tiny variations in X

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

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

- The ***gradient*** is the transpose of the derivative $\nabla_X f(X)^T$
  - A column vector of the same dimensionality as X

Gradient of a scalar function of a vector



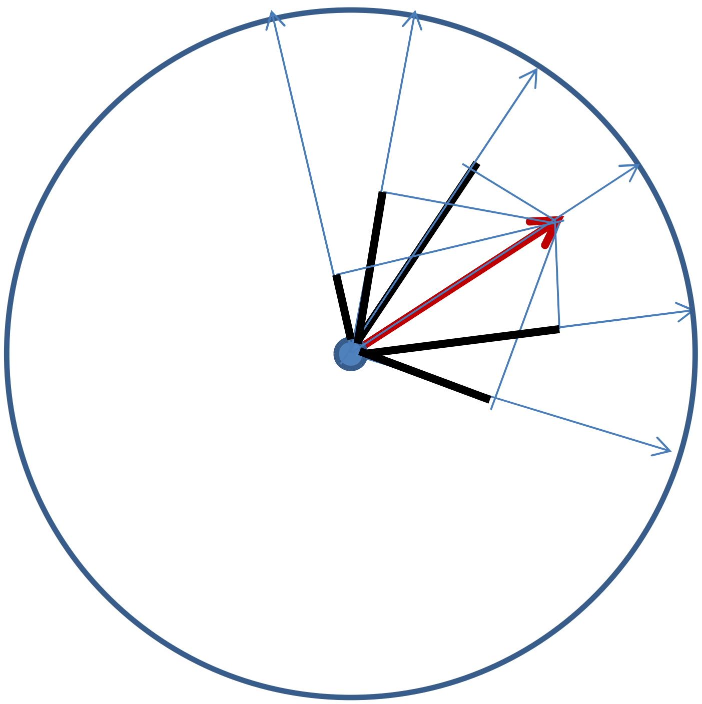
- The *derivative* $\nabla_X f(X)$ of a scalar function $f(X)$ of a multi-variate input X is a multiplicative factor that gives us the change in $f(X)$ for tiny variations in X

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

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

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

A well-known vector property

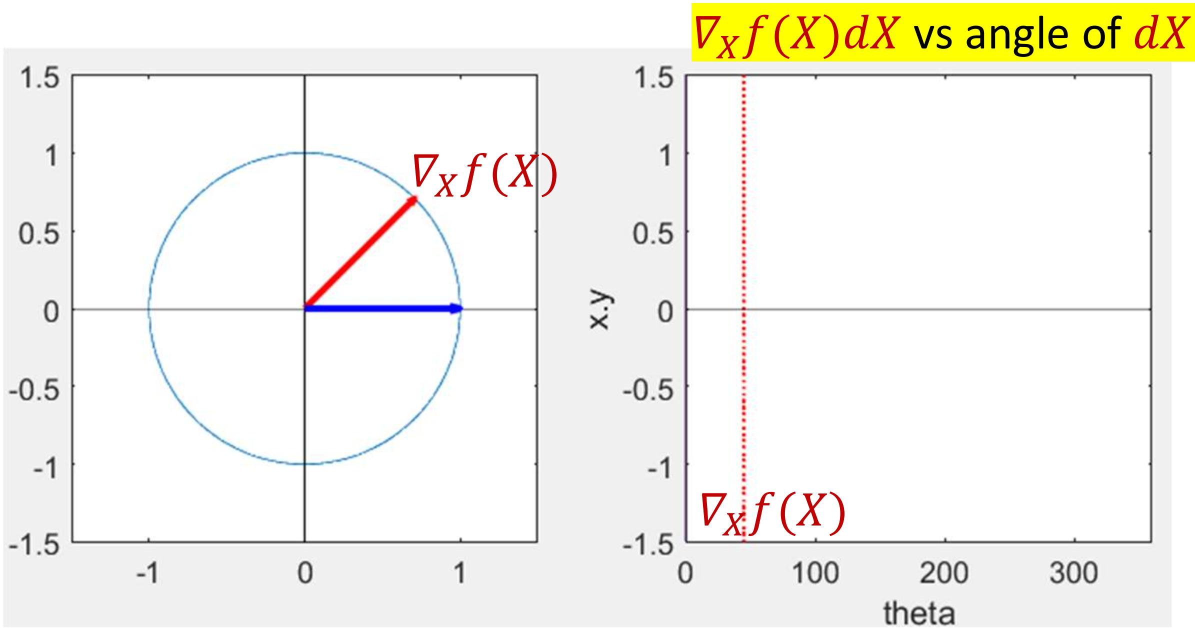


$$\mathbf{u}^{\mathrm{T}}\mathbf{v} = |\mathbf{u}||\mathbf{v}|cos\theta$$

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

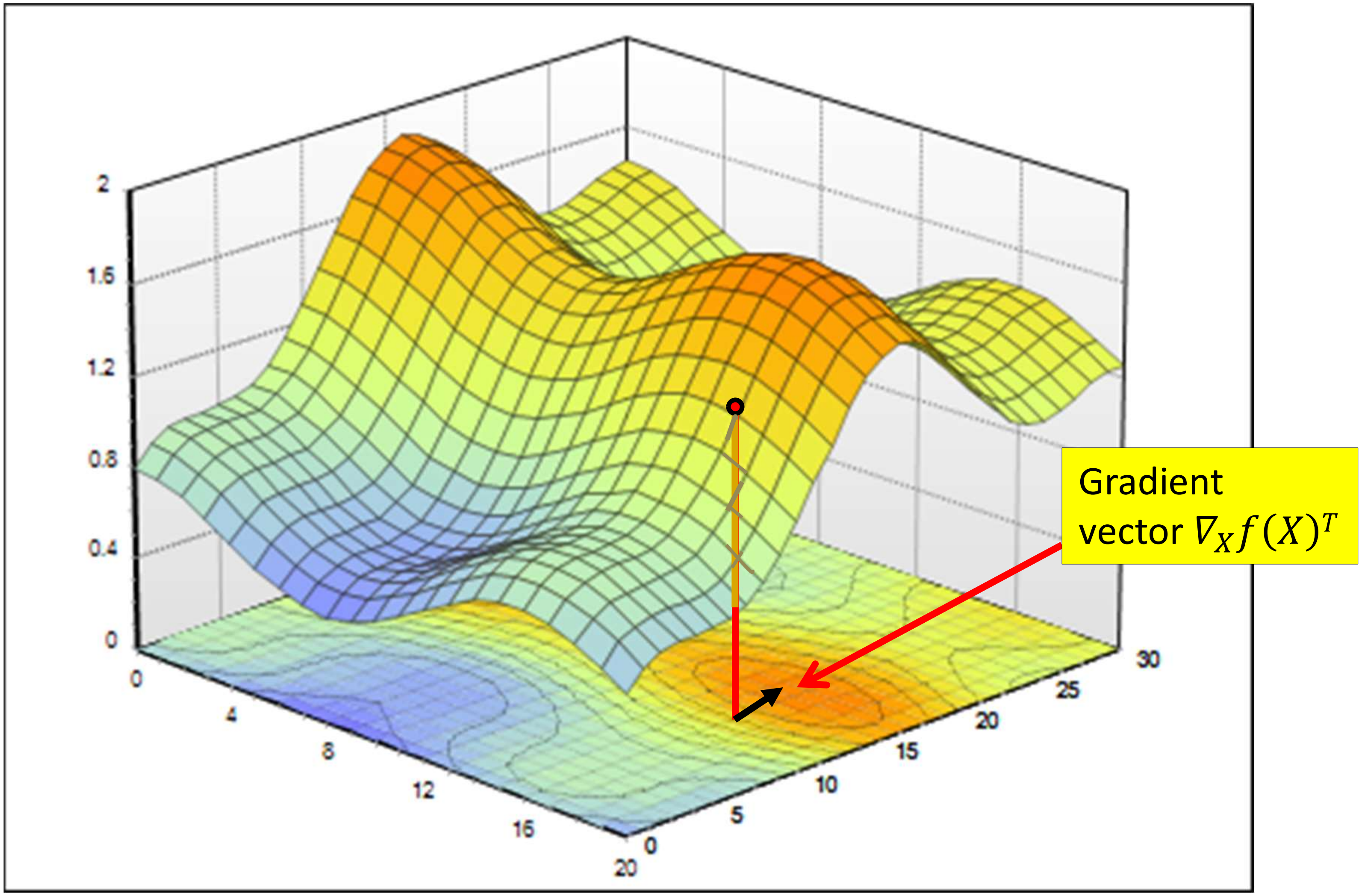
Properties of Gradient

Blue arrow is dX



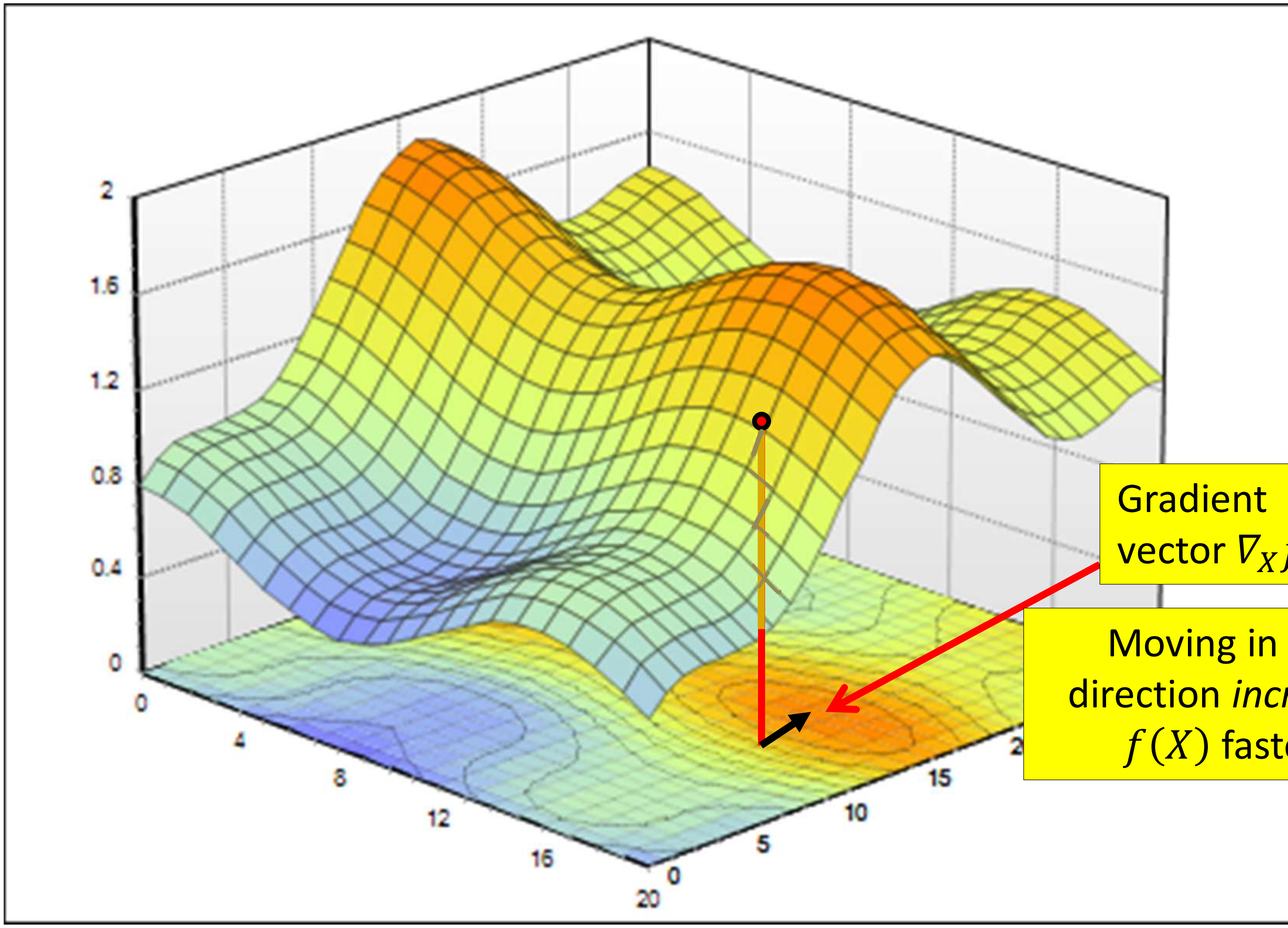
- $df(X) = \nabla_X f(X) dX$
- For an increment dX of any given length $df(X)$ is max if dX is aligned with $\nabla_X f(X)^{\mathrm{T}}$
  – The function $f(X)$ increases most rapidly if the input increment dX is exactly in the direction of $\nabla_X f(X)^{\mathrm{T}}$
- The gradient is the direction of fastest increase in $f(X)$

Gradient



2
1.6
1.2
0.8
0.4
0
0
4
8
12
16
20
0
5
10
15
20
25
30
Gradient vector $\nabla_X f(X)^T$

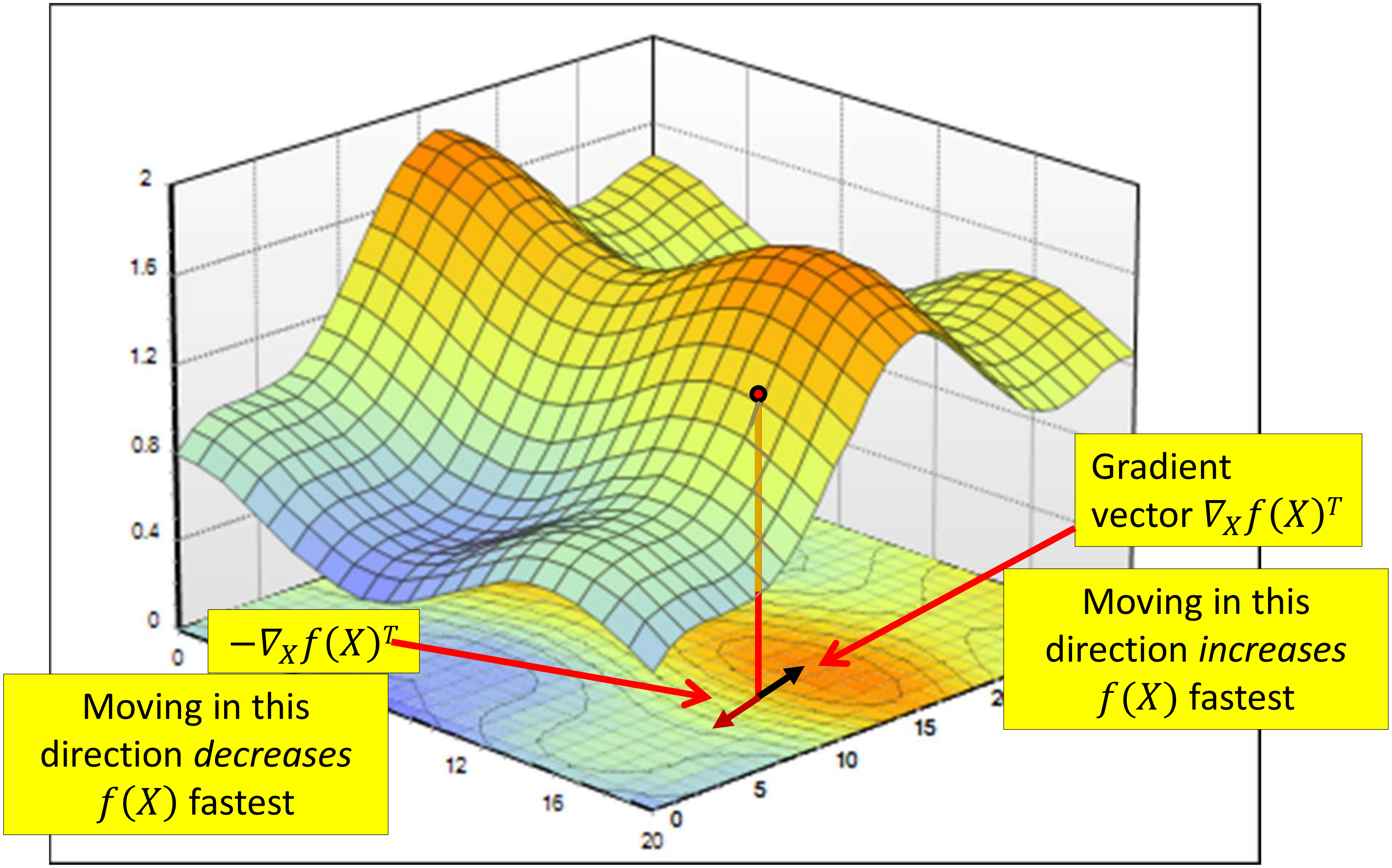
Gradient



Gradient vector $\nabla_X f(X)^T$

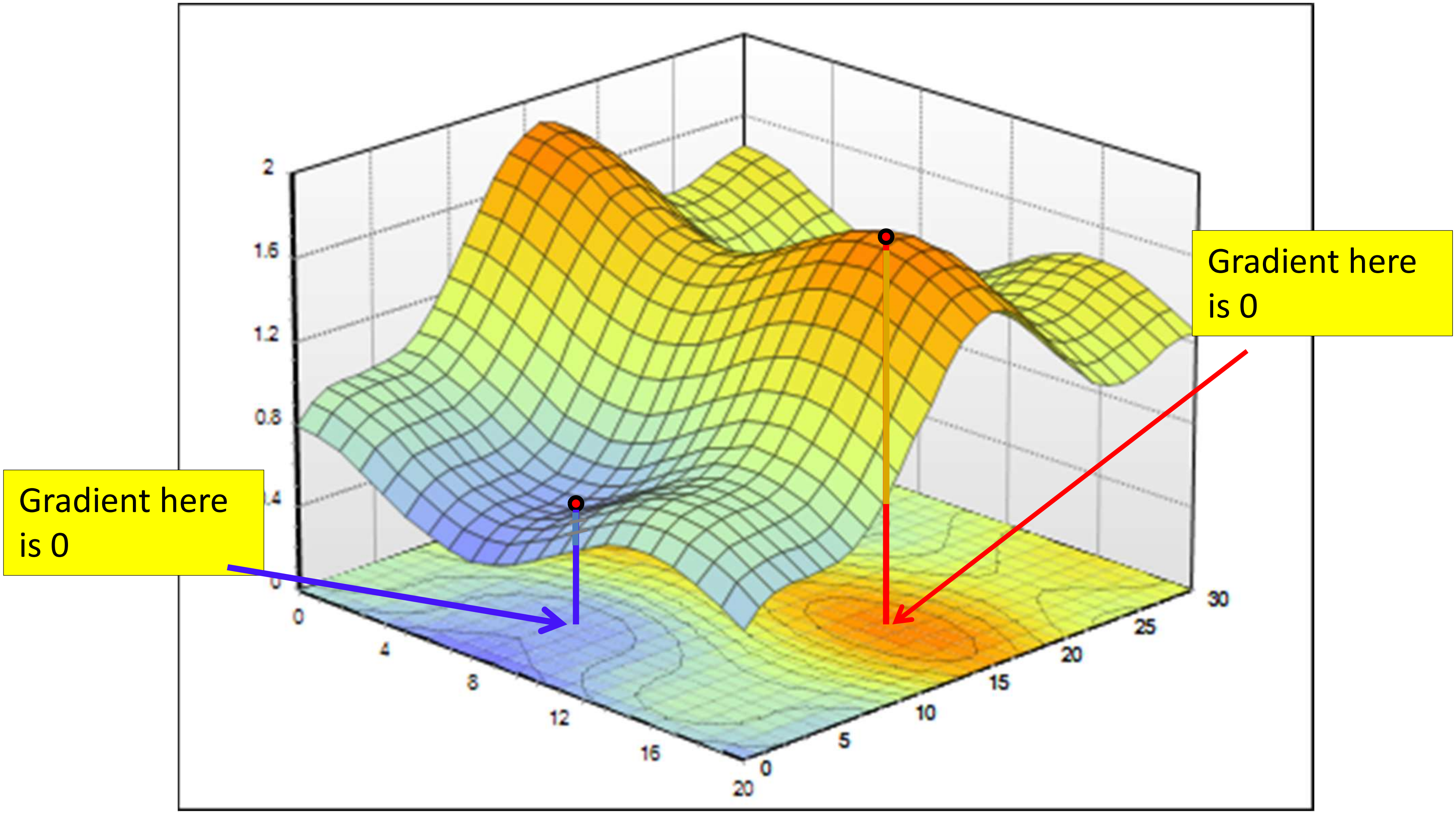
Moving in this direction *increases* $f(X)$ fastest

Gradient

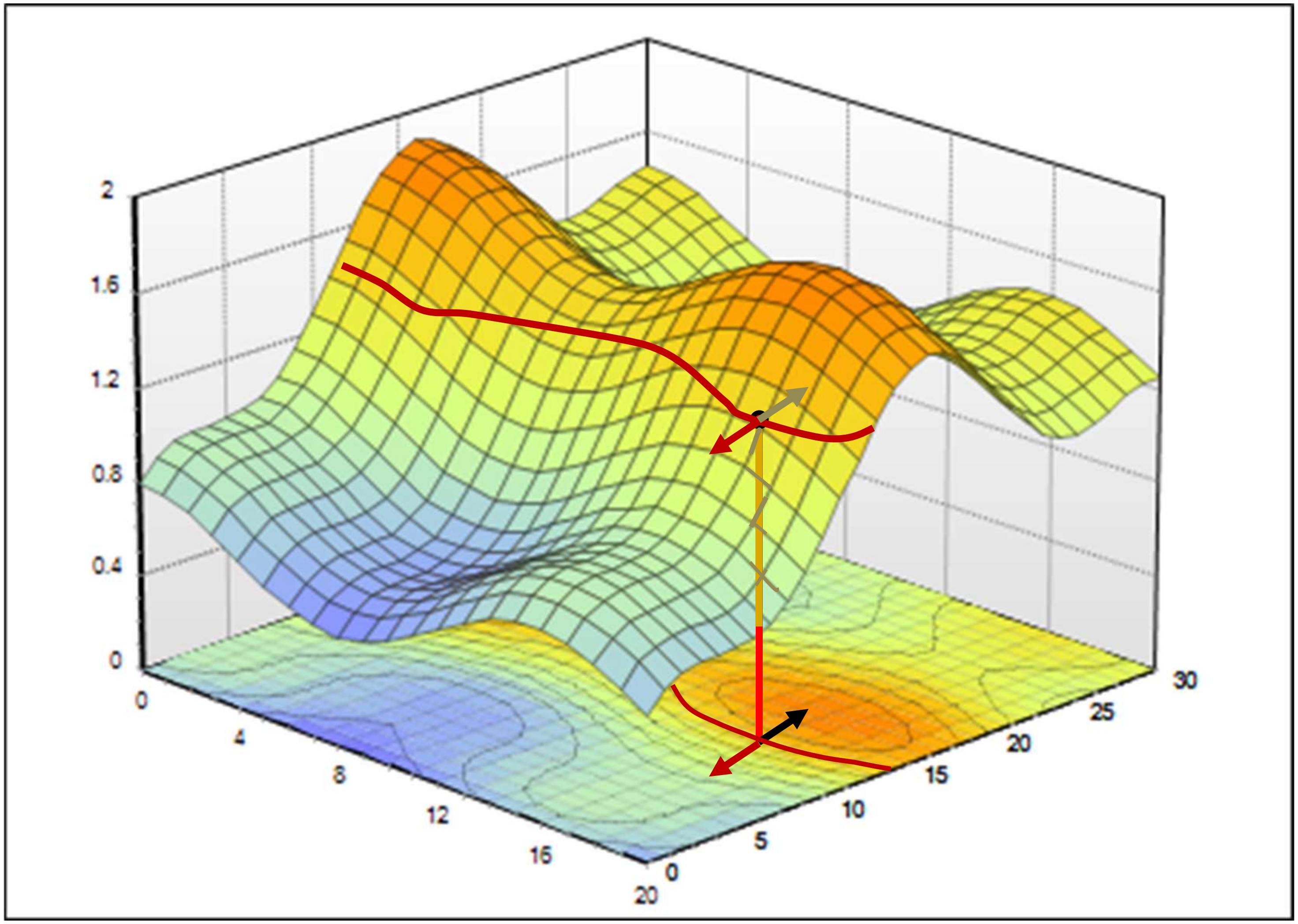


2
1.6
1.2
0.8
0.4
0
0
12
16
20
0
5
10
15
Gradient vector $\nabla_X f(X)^T$
Moving in this direction *increases* $f(X)$ fastest
$-\nabla_X f(X)^T$
Moving in this direction *decreases* $f(X)$ fastest

Gradient


Gradient here is 0
Gradient here is 0
2
1.6
1.2
0.8
0
4
8
12
16
20
0
5
10
15
20
25
30

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 $f(x_1, x_2, \ldots, x_n)$ is given by the second derivative

$$\nabla_X^2 f(x_1,\ldots,x_n) := \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1{}^2} & \dfrac{\partial^2 f}{\partial x_1 \partial x_2} & . & . & \dfrac{\partial^2 f}{\partial x_1 \partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2 \partial x_1} & \dfrac{\partial^2 f}{\partial x_2{}^2} & . & . & \dfrac{\partial^2 f}{\partial x_2 \partial x_n} \\ . & . & . & . & . \\ . & . & . & . & . \\ \dfrac{\partial^2 f}{\partial x_n \partial x_1} & \dfrac{\partial^2 f}{\partial x_n \partial x_2} & . & . & \dfrac{\partial^2 f}{\partial x_n{}^2} \end{bmatrix}$$

Next up

- Continuing on function optimization
- Gradient descent to train neural networks
- A.K.A. Back propagation