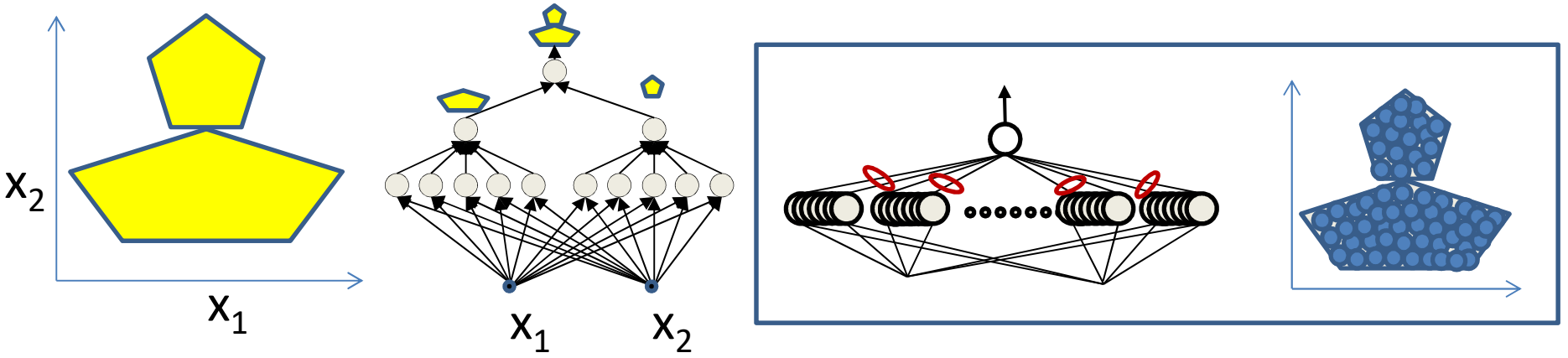
Depth and the universal classifier



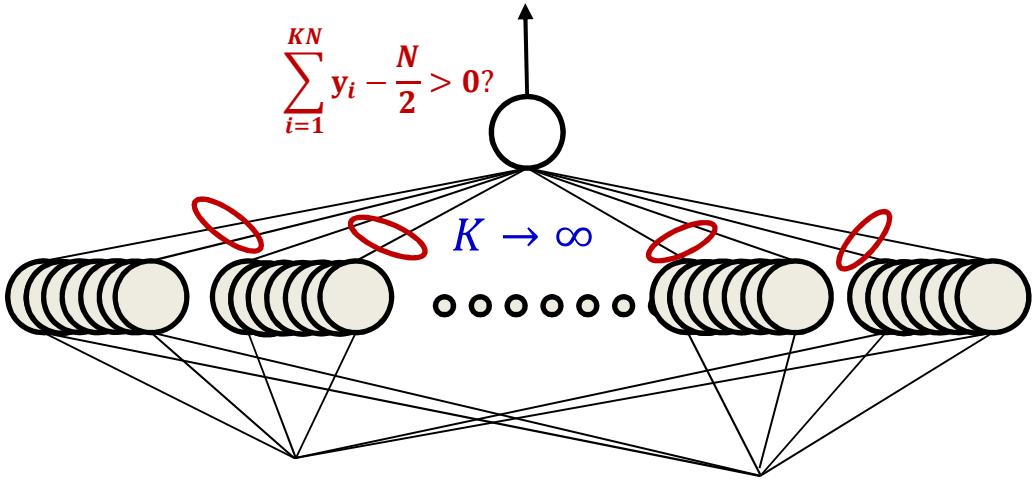
- Deeper networks can require far fewer neurons

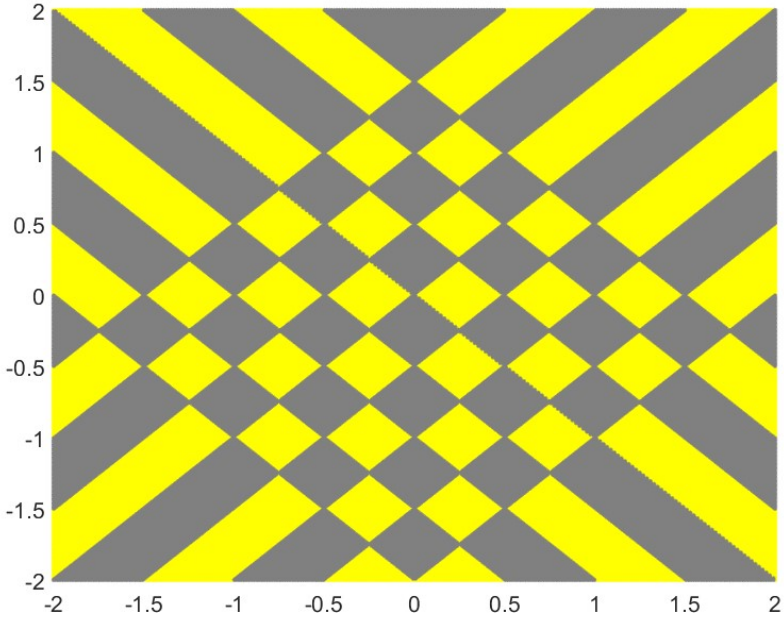
Optimal depth in *generic* nets

- We look at a different pattern:
  - "worst case" decision boundaries

- For *threshold-activation* networks
  - Generalizes to other nets

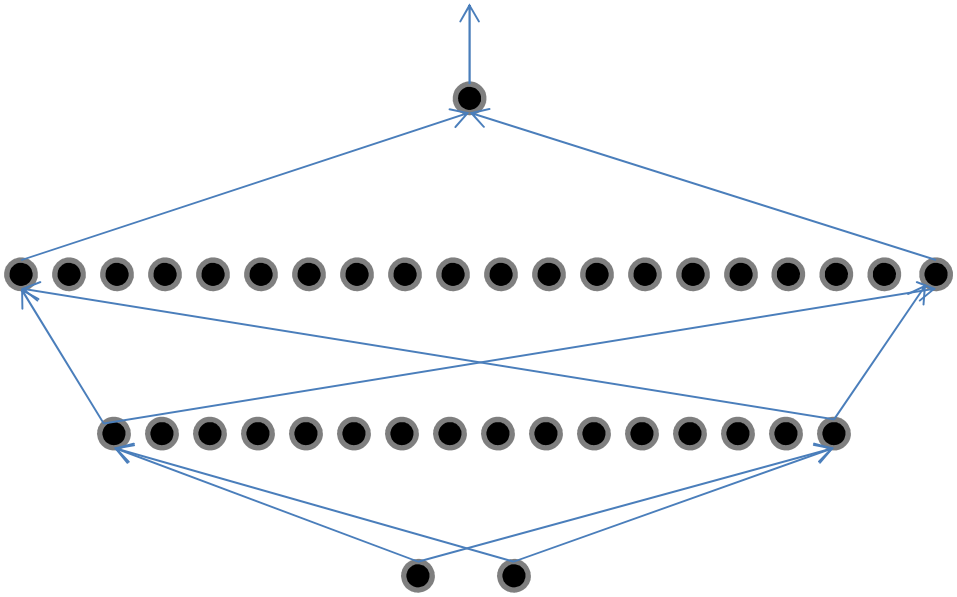
Optimal depth

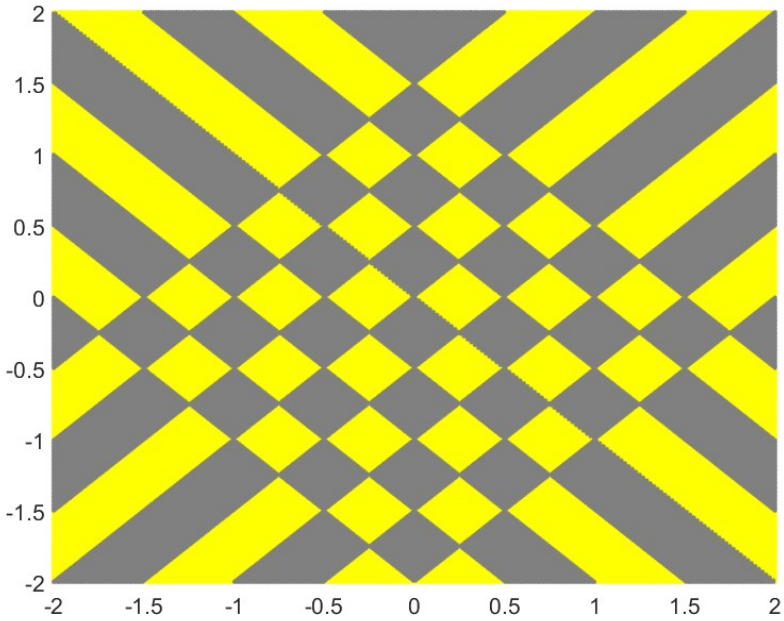




- A naïve one-hidden-layer neural network will required infinite hidden neurons

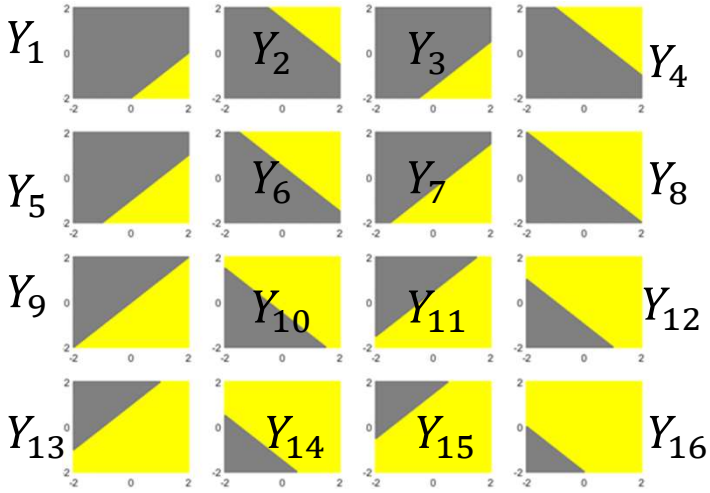
Optimal depth

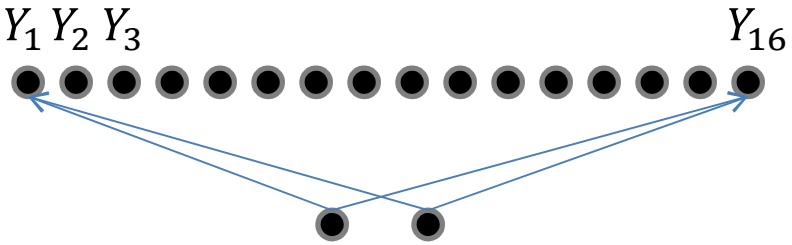


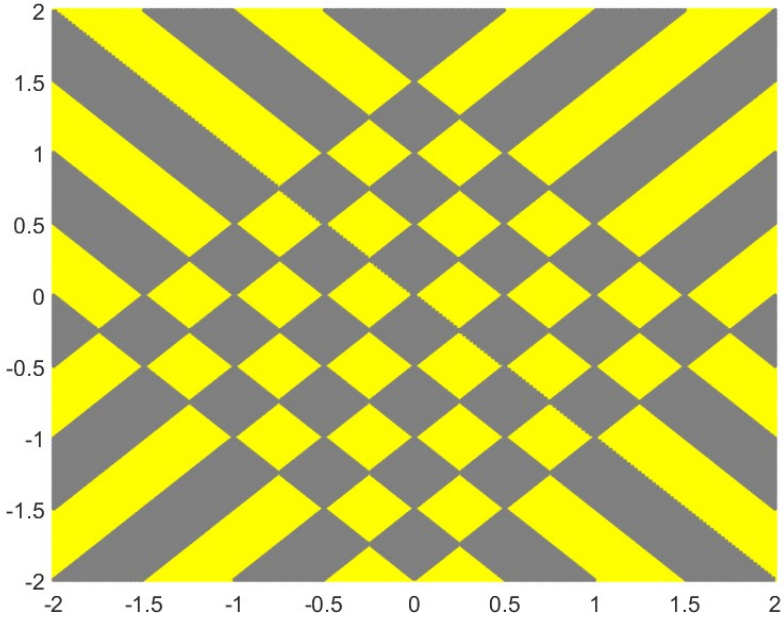


- Two hidden-layer network: 56 hidden neurons

Optimal depth

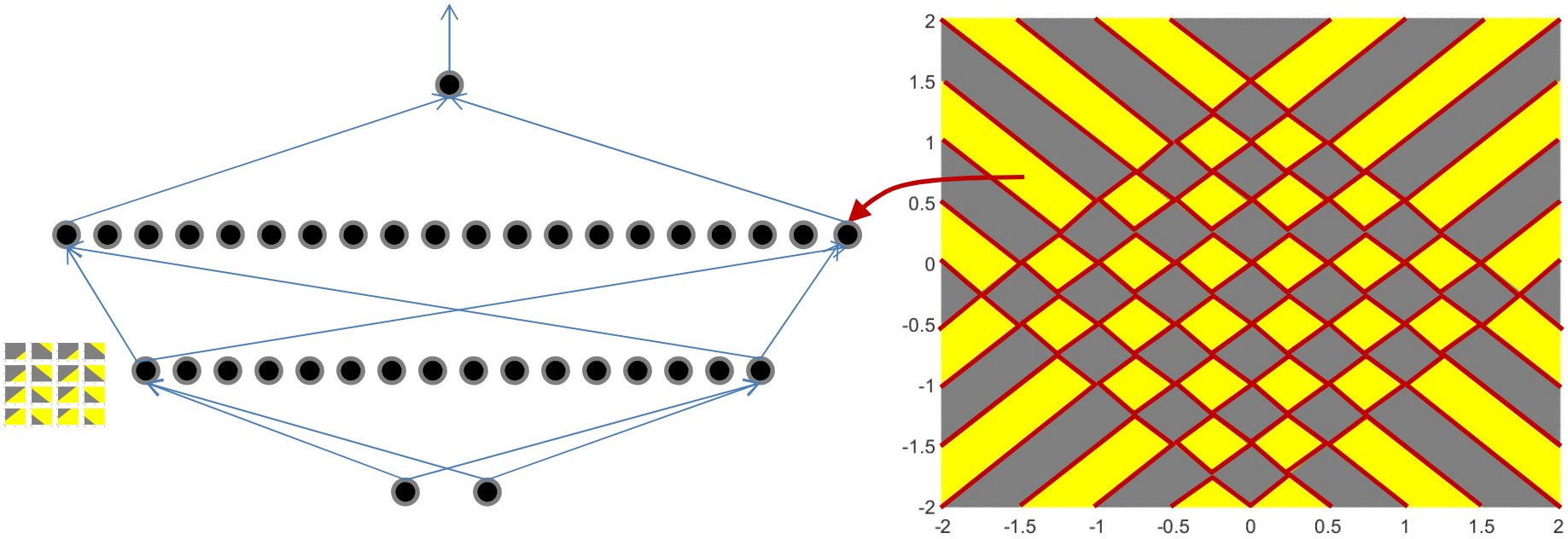






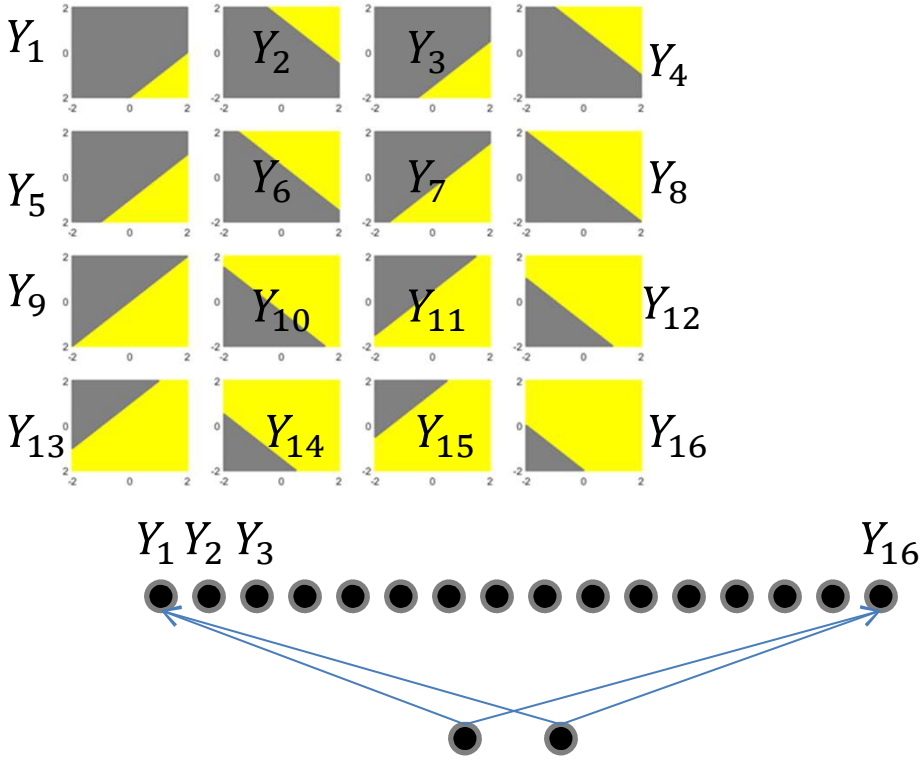
- Two layer network: 56 hidden neurons
  - 16 neurons in hidden layer 1

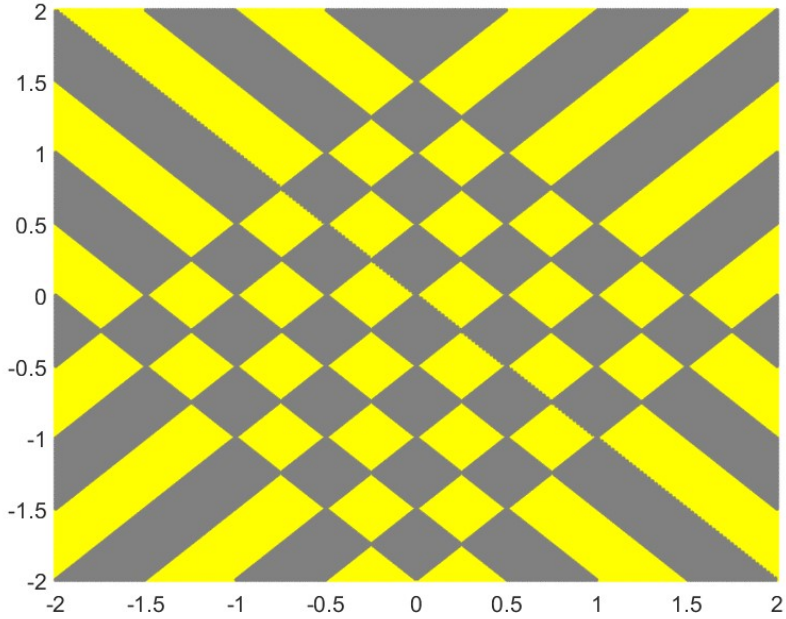
Optimal depth



- Two-layer network: 56 hidden neurons
  - 16 in hidden layer 1
  - 40 in hidden layer 2
  - 57 total neurons, including output neuron

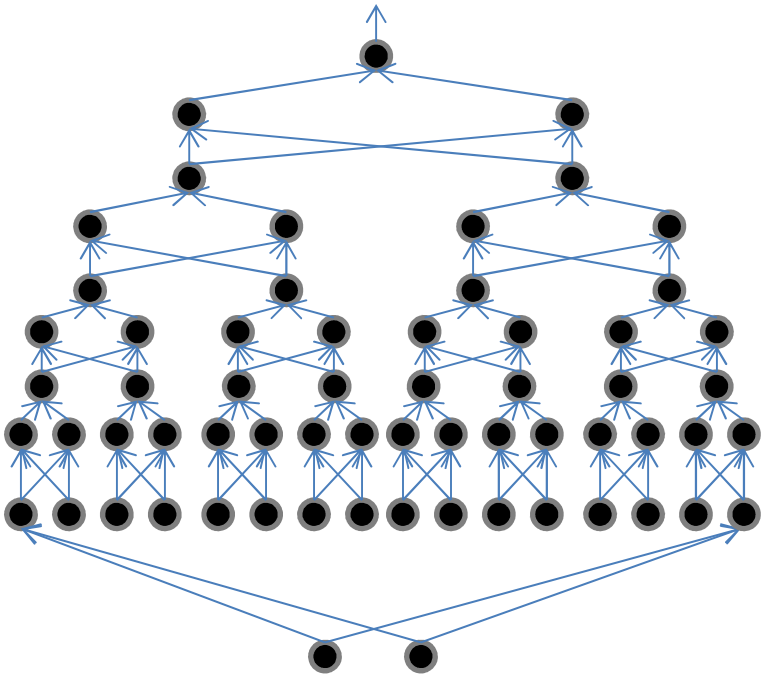
Optimal depth

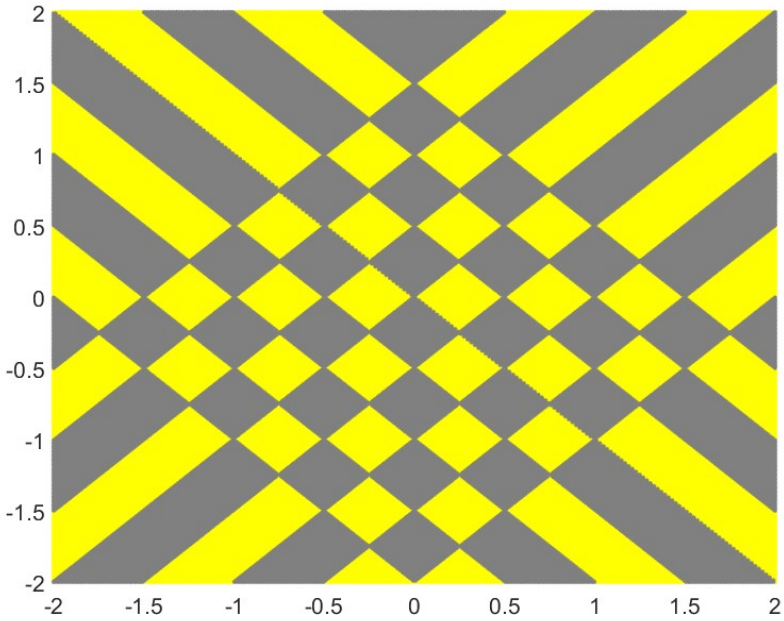




- But this is just $Y_1 \oplus Y_2 \oplus \cdots \oplus Y_{16}$

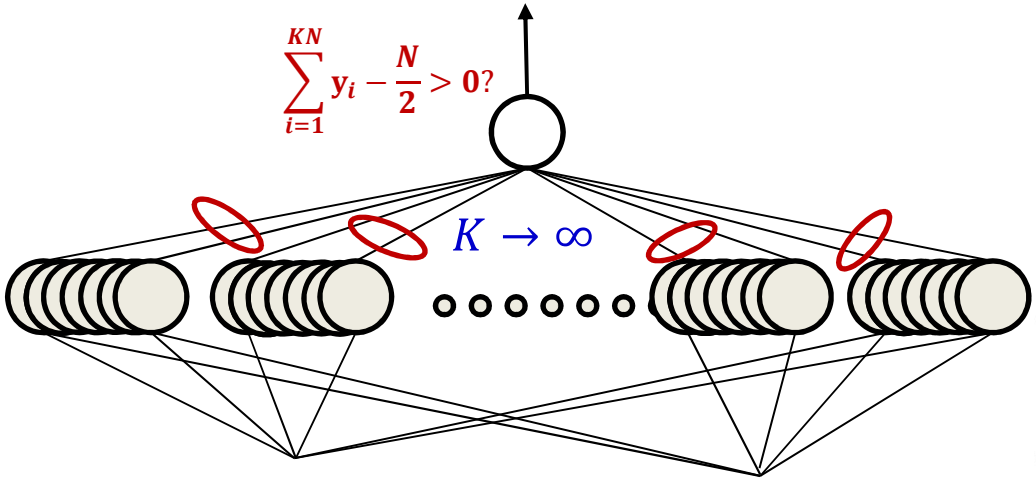
Optimal depth

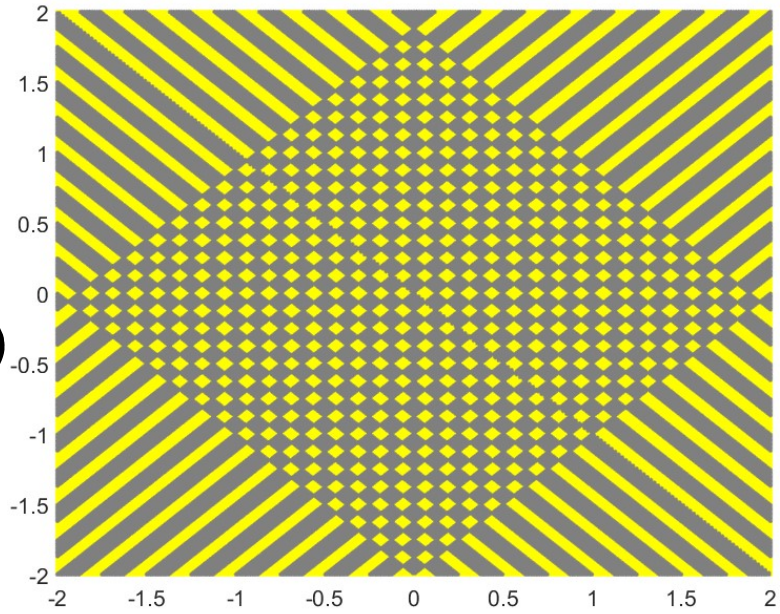




- But this is just $Y_1 \oplus Y_2 \oplus \cdots \oplus Y_{16}$
  - The XOR net will require 16 + 15x3 = 61 neurons
    - 46 neurons if we use a two-gate XOR

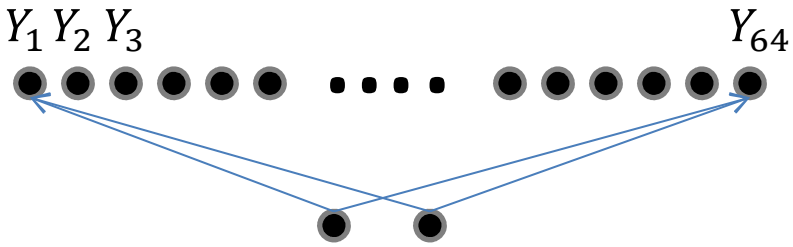
Optimal depth

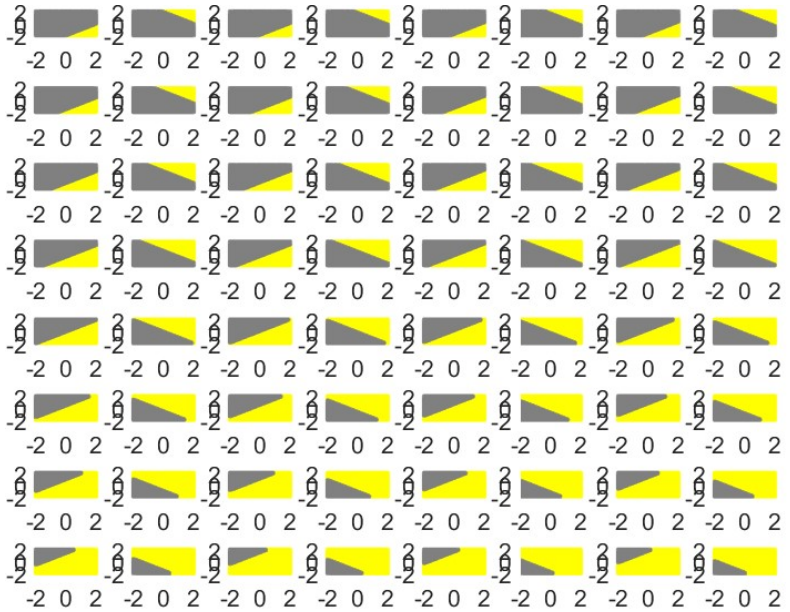




- A naïve one-hidden-layer neural network will required infinite hidden neurons

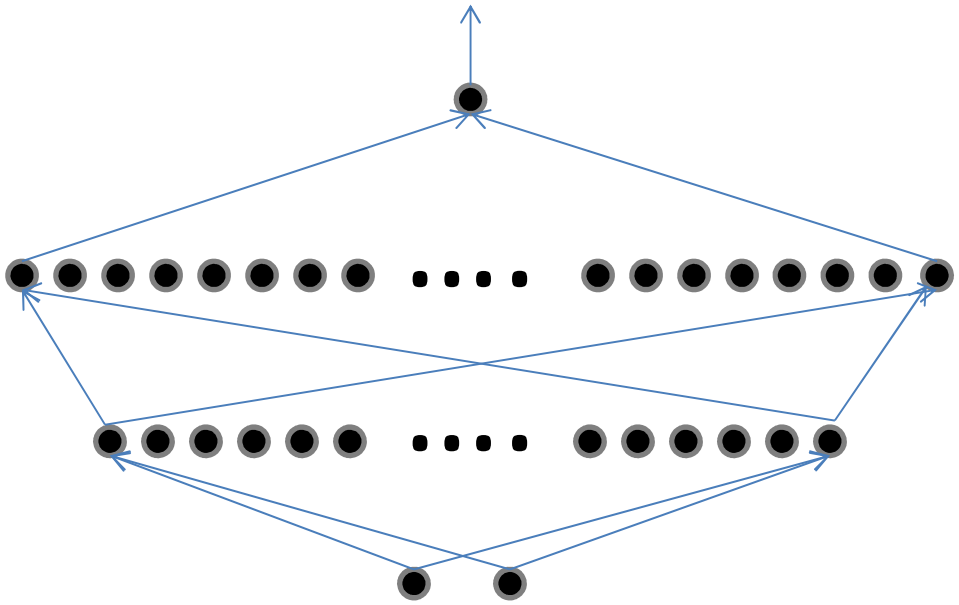
Actual linear units

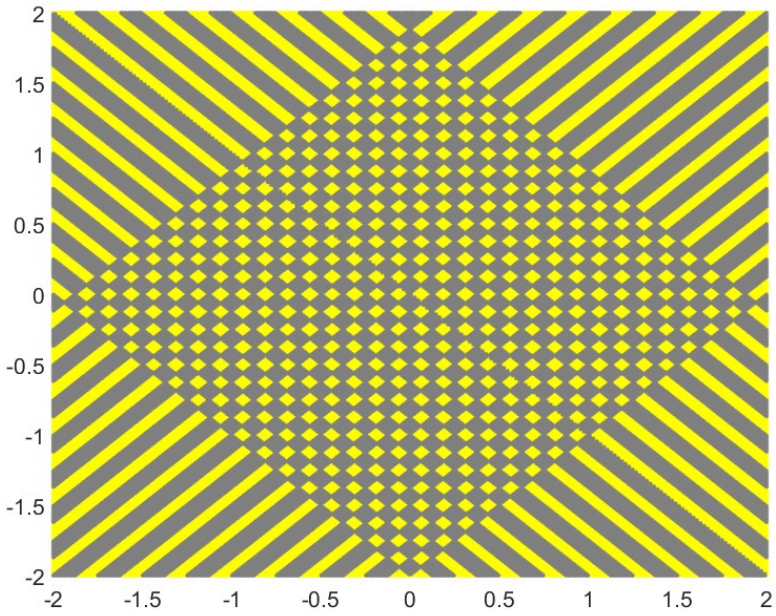




- 64 basic linear feature detectors

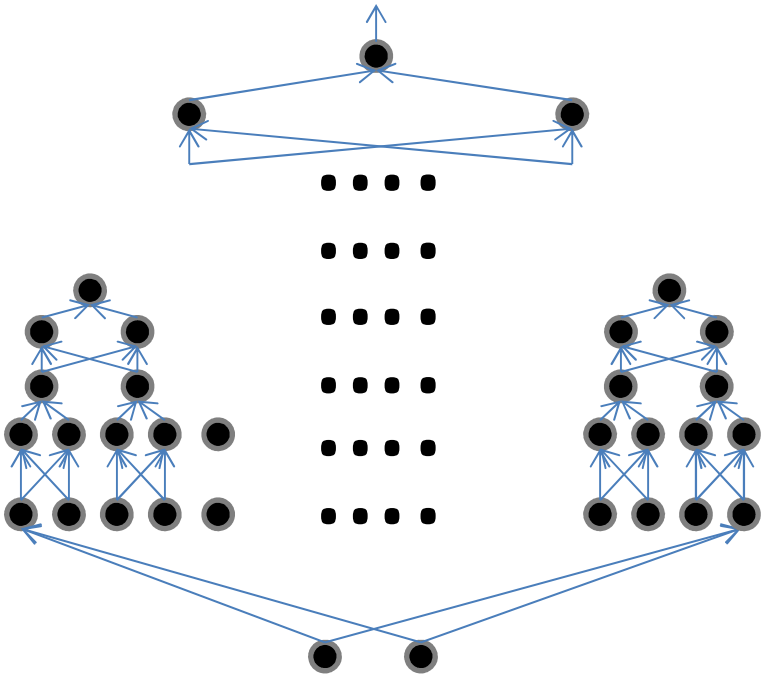
Optimal depth

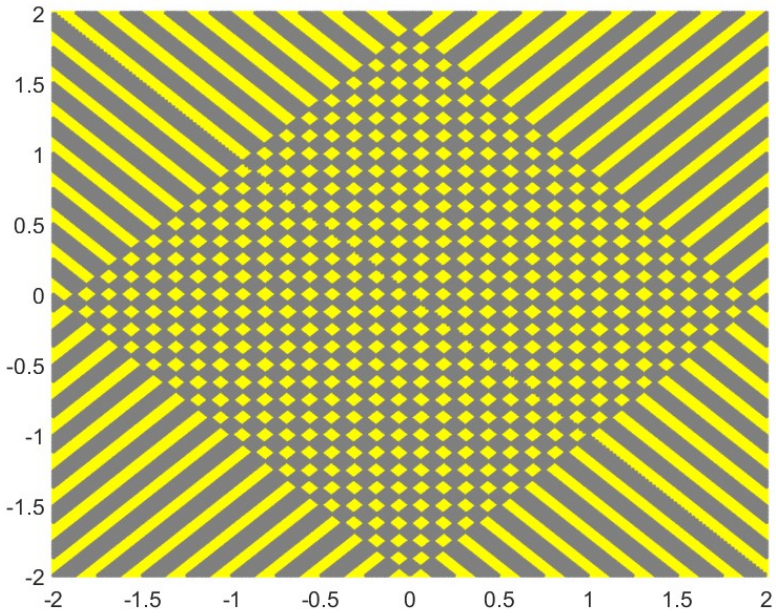




- Two hidden layers: 608 hidden neurons
  - 64 in layer 1
  - 544 in layer 2
- 609 total neurons (including output neuron)

Optimal depth

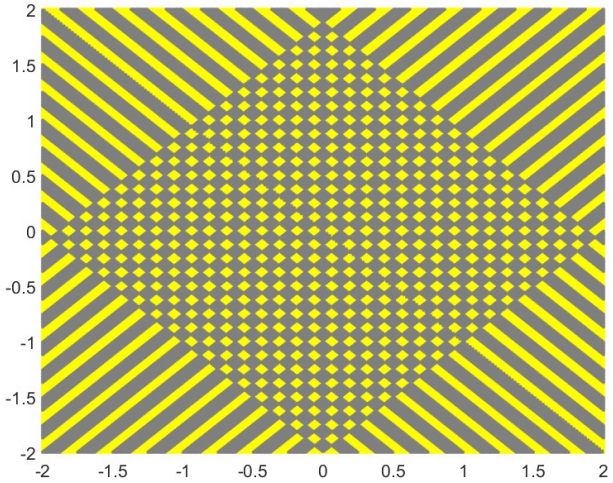




- XOR network (12 hidden layers): 253 neurons
  - 190 neurons with 2-gate XOR
- The difference in size between the deeper optimal (XOR) net and shallower nets increases with increasing pattern complexity and input dimension

Network size?

- In this problem the 2-layer net was *quadratic* in the number of lines
  - $\lfloor (N+2)^2/8 \rfloor$ neurons in 2nd hidden layer
  - Not exponential
  - Even though the pattern is an XOR
  - Why?



- The data are two-dimensional!
  - Only two *fully independent* features
  - The pattern is exponential in the *dimension of the input (two)!*
- For general case of N mutually intersecting hyperplanes in D dimensions, we will need $\mathcal{O}\left(\frac{N^D}{(D-1)!}\right)$ weights (assuming $N \gg D$).
  - Increasing input dimensions can increase the worst-case size of the shallower network exponentially, but not the XOR net
    - The size of the XOR net depends only on the number of first-level linear detectors (N)

Number of regions created by m hyperplanes in n-dimension

Proposition[a]

[a] Chapter6, R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

Let $R(m, n)$ denote the number of different regions defined by m separating hyperplanes of dimension $n-1$, in general position, in an n-dimensional space. We set $R(1, n) = 2$ for $n \geq 1$ and $R(m, 0) = 0, \forall m \geq 1$. For $n \geq 1$ and $m > 1$,

$$R(m, n) = R(m-1, n) + R(m-1, n-1).$$

Proof of the Proposition

The proof is by induction on m[2].

- When $m = 2$ and $n = 1$ the formula is valid. As $R(2, 1) = 3$ and $R(1, 1) = 2$ and $R(1, 0) = 1$.
- If $m = 2$ and $n \geq 2$, we know that $R(2, n) = 4$ and the formula is valid again:

$$R(2, n) = R(1, n) + R(1, n - 1) = 2 + 2 = 4.$$

- Now $m + 1$ hyperplanes of dimension $n - 1$ are given in n-dimensional space and in general position ($n \geq 2$).
  - From the induction hypotheses, it follows that the first m hyperplanes define $R(m, n)$ regions in n-dimensional space.
  - First m hyperplanes divide the $(m + 1)^{th}$ hyperplane into $R(m, n - 1)$ regions. After the cut with the hyperplane $m + 1$, exactly $R(m, n - 1)$ new regions have been created. The new number of regions is therefore $R(m + 1, n) = R(m, n) + R(m, n - 1)$.

[2] Chapter6, R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

What is $R(m, n)$?

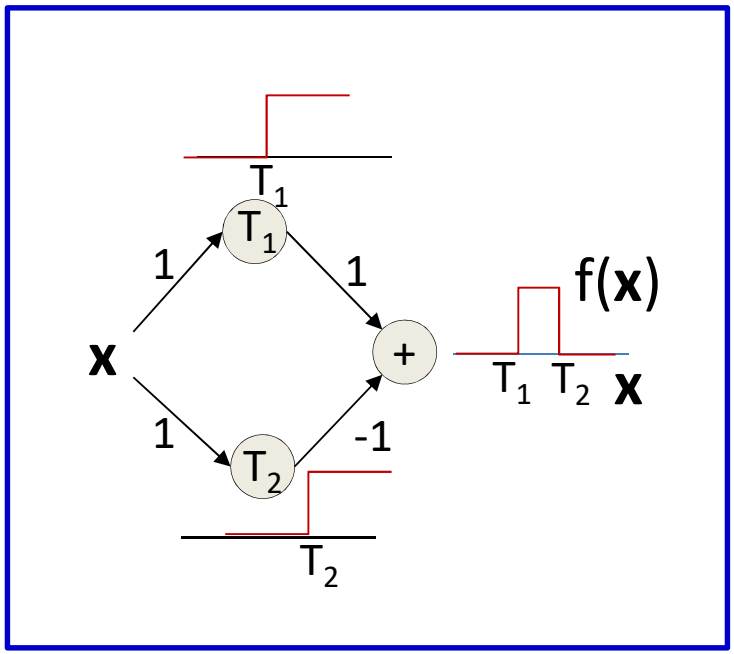
Upper Bound on $R(m, n)$

$$R(m, n) = 2\sum_{i=1}^{n-1} {}^{m-1}C_i < 2\frac{m^n}{n!}$$

Depth: Summary

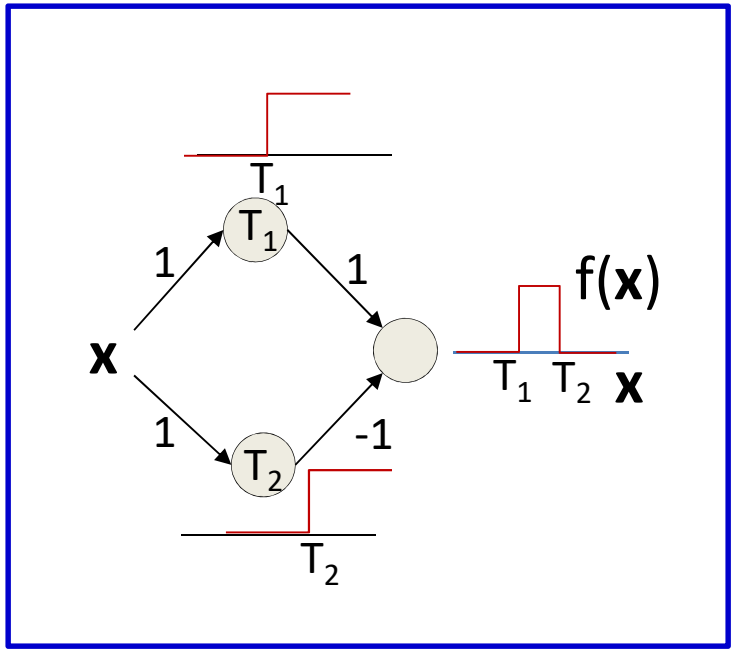
- The number of neurons required in a shallow network is potentially exponential in the dimensionality of the input
  - (this is the worst case)
  - Alternately, exponential in the number of statistically independent features

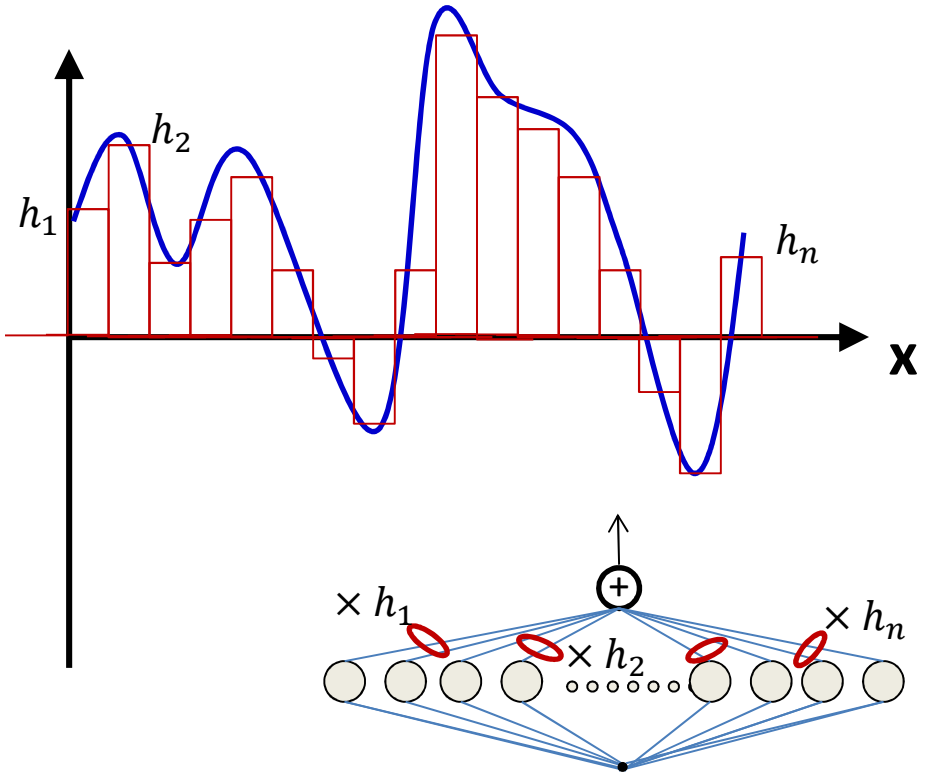
MLP as a continuous-valued regression



- A simple 3-unit MLP with a "summing" output unit can generate a "square pulse" over an input
  - Output is 1 only if the input lies between T_1 and T_2
  - T_1 and T_2 can be arbitrarily specified

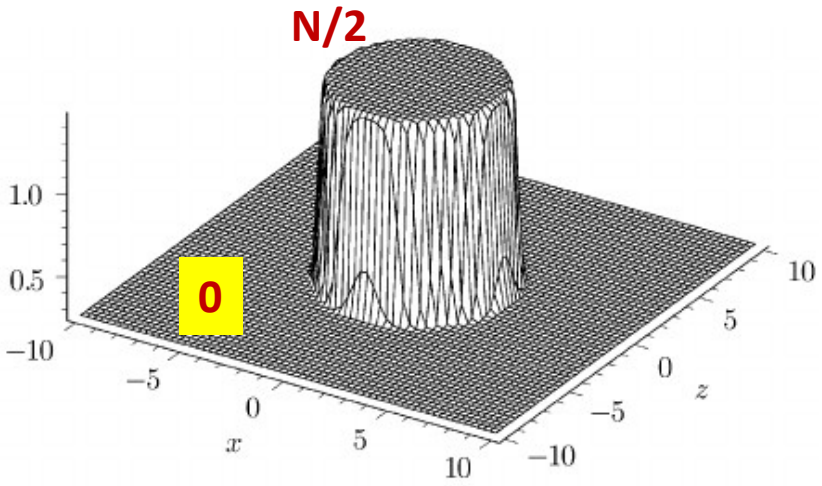
MLP as a continuous-valued regression

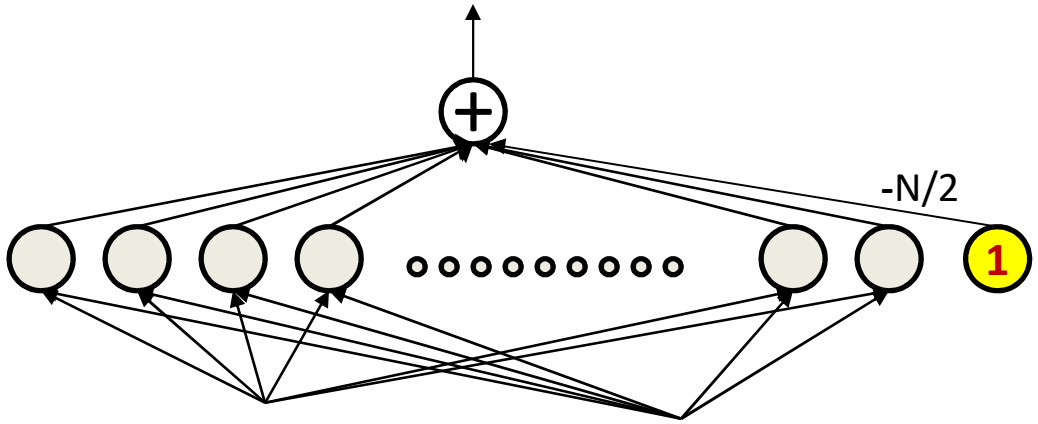




- A simple 3-unit MLP can generate a "square pulse" over an input
- **An MLP with many units can model an arbitrary function over an input**
  - To arbitrary precision
    - Simply make the individual pulses narrower
- ***A one-layer MLP can model an arbitrary function of a single input***

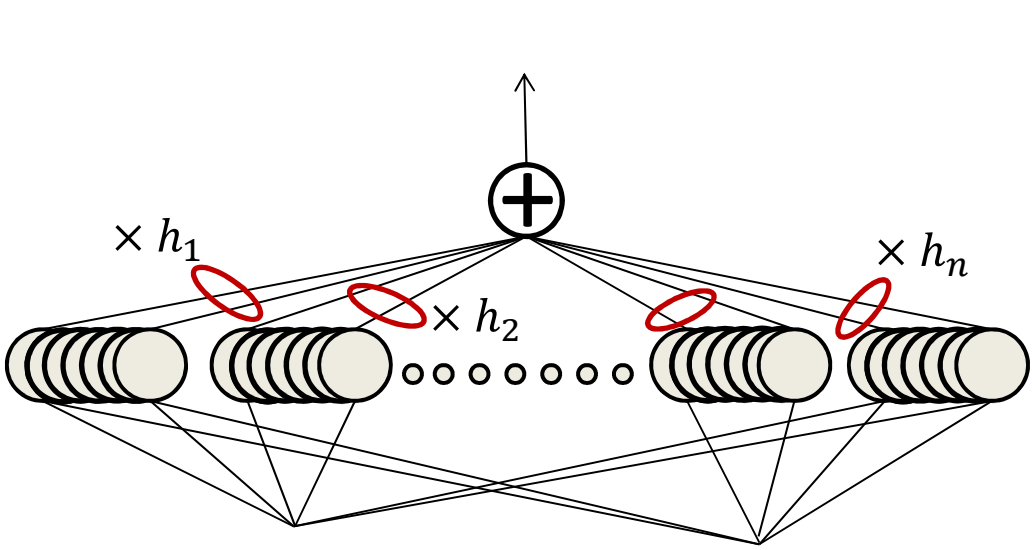
For higher dimensions

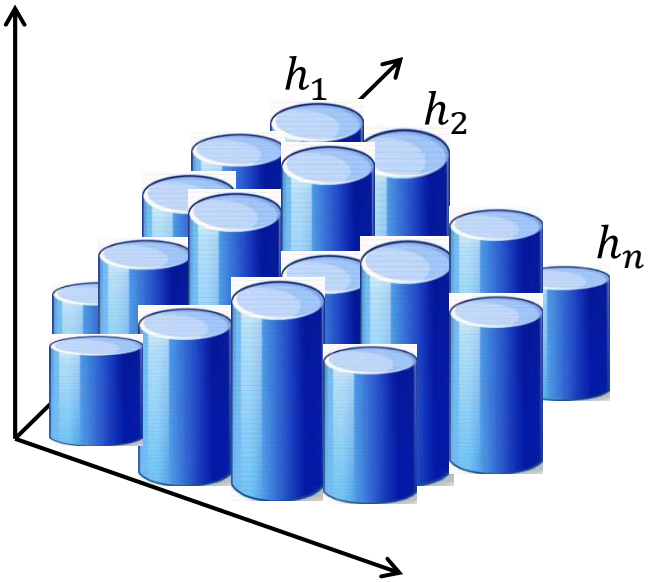




- An MLP can compose a cylinder
  - $N/2$ in the circle, 0 outside

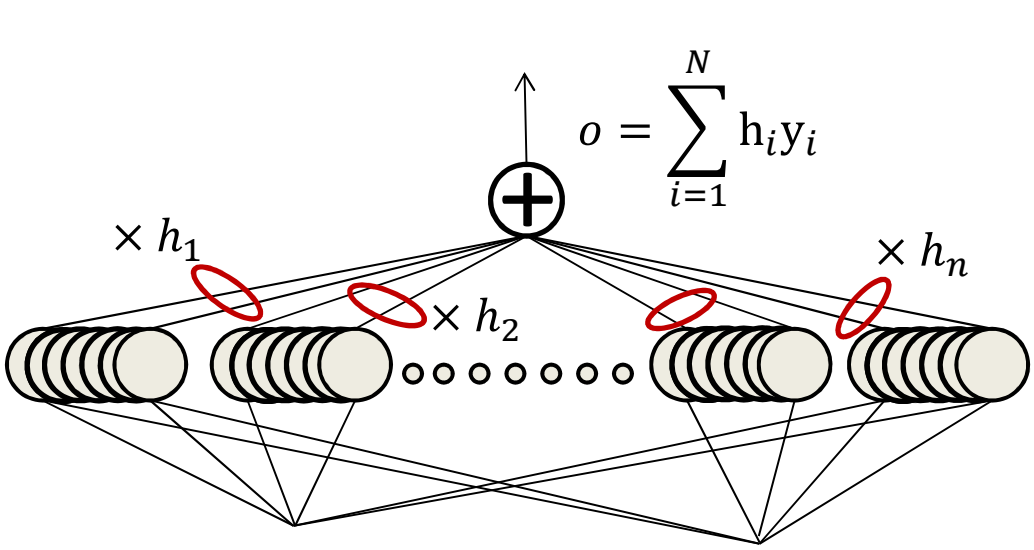
MLP as a continuous-valued function

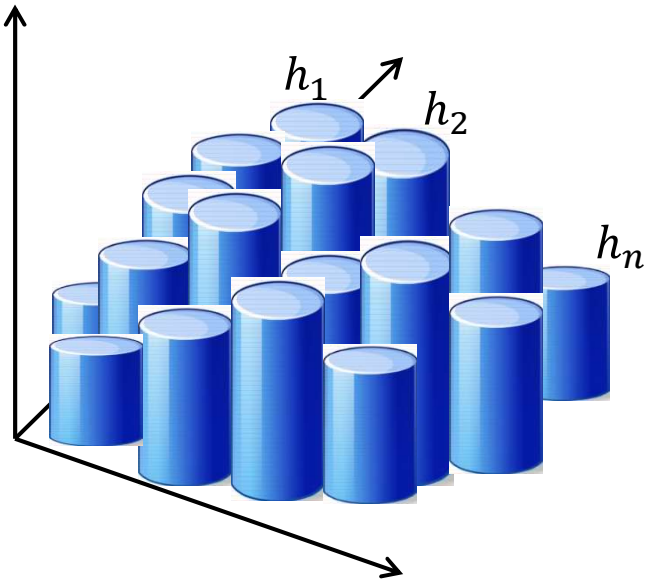




- MLPs can actually compose arbitrary functions in any number of dimensions!
  - Even with only one layer
    - As sums of scaled and shifted cylinders
  - To arbitrary precision
    - By making the cylinders thinner
  - **The MLP is a universal approximator!**

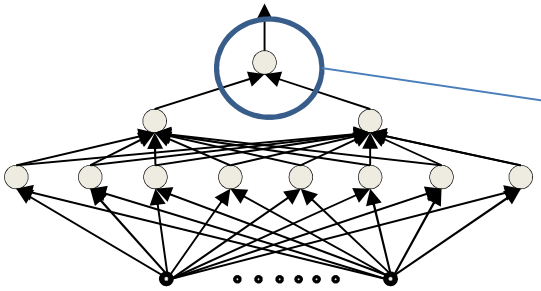
Caution: MLPs with additive output units are universal approximators

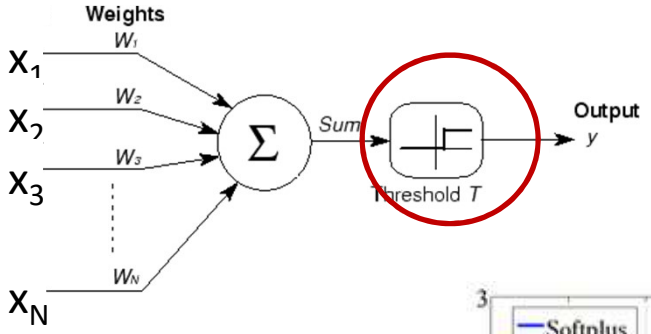




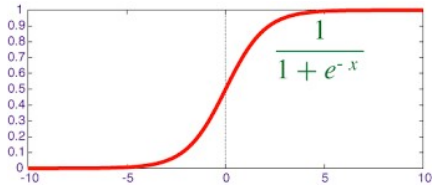
- MLPs can actually compose arbitrary functions
- But explanation so far only holds if the output unit only performs summation
  - i.e. does not have an additional “activation”

"Proper" networks: Outputs with activations

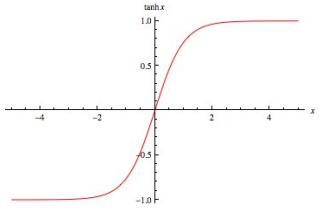


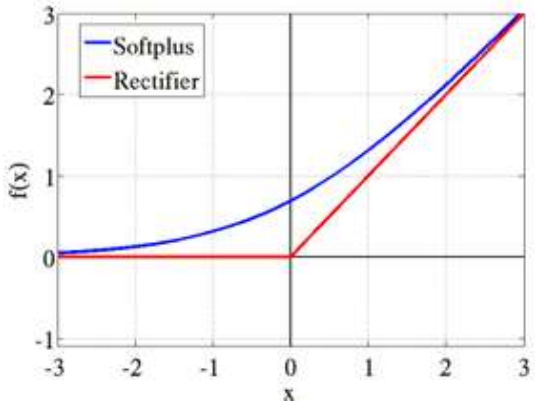


sigmoid



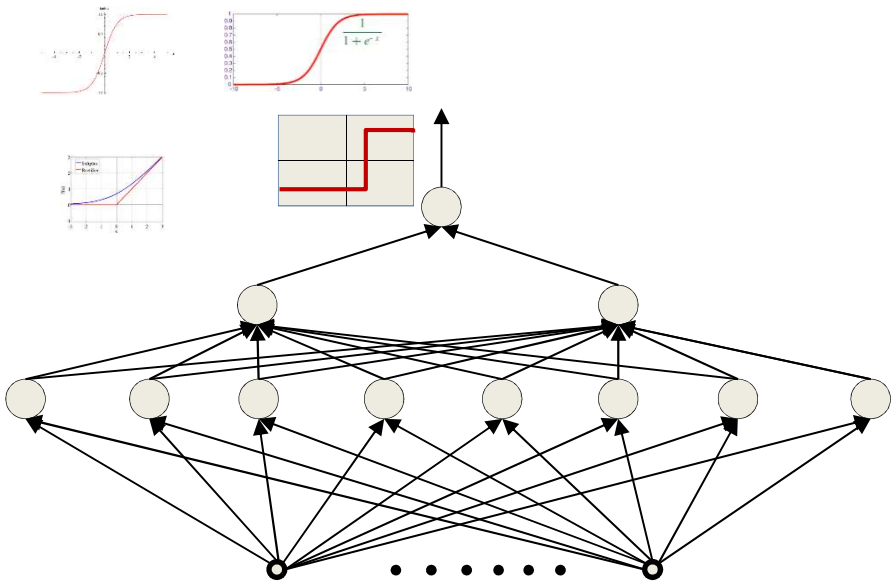
tanh





- Output neuron may have actual "activation"
  - Threshold, sigmoid, tanh, softplus, rectifier, etc.
- What is the property of such networks?

The network as a function



$f: \{0,1\}^N \rightarrow \{0,1\}$ $Boolean$

$f: R^N \rightarrow \{0,1\}$ $Threshold$

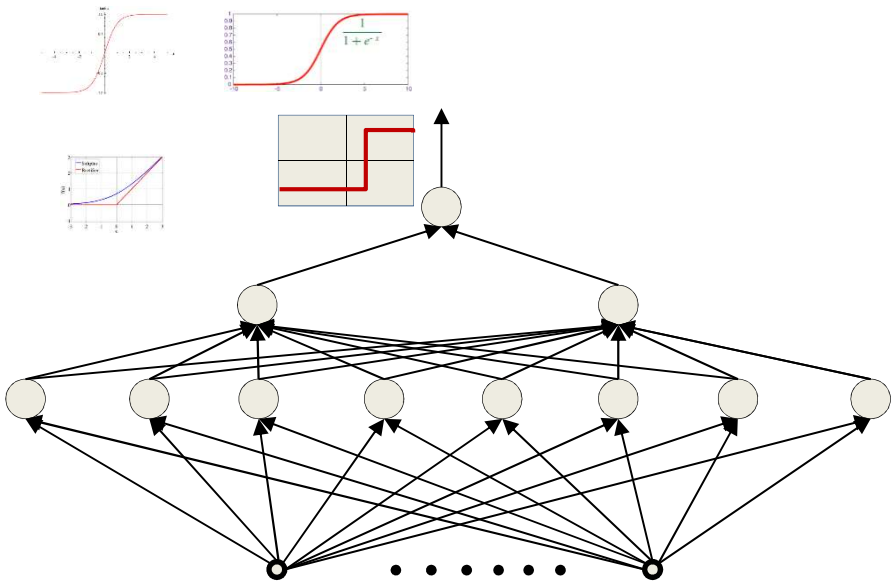
$f: R^N \rightarrow (0,1)$ $Sigmoid$

$f: R^N \rightarrow (-1,1)$ $Tanh$

$f: R^N \rightarrow (0,\infty)$ $Softrectifier, Rectifier$

- Output unit with *activation function*
  - Threshold or Sigmoid, or any other
- The network is actually a universal map from the entire domain of input values to the entire range of the output activation
  - All values the activation function of the output neuron

The network as a function



$f: \{0,1\}^N \rightarrow \{0,1\}$ *Boolean*

$f: R^N \rightarrow \{0,1\}$ *Threshold*

$f: R^N \rightarrow (0,1)$ *Sigmoid*

$f: R^N \rightarrow (-1,1)$ *Tanh*

$f: R^N \rightarrow (0, \infty)$ *Softrectifier, Rectifier*

The MLP is a *Universal Approximator* for the entire *class* of functions (maps) it represents!

~~Output unit with activation function~~

- Threshold or Sigmoid, or any other

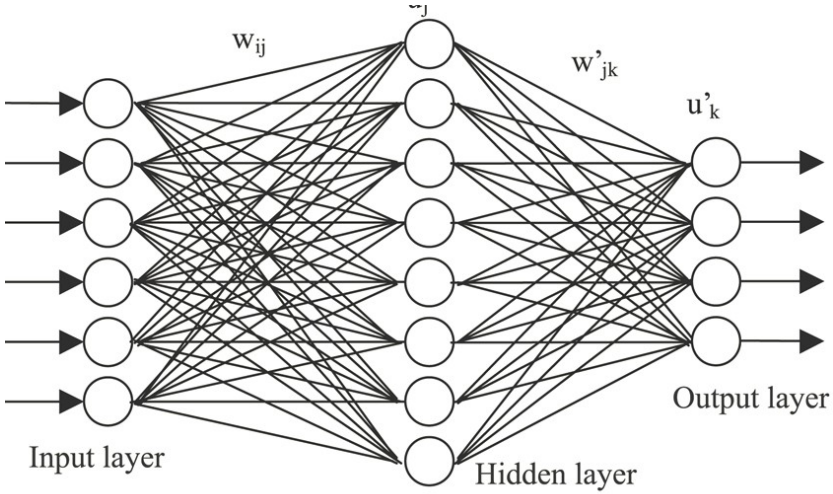
- The network is actually a universal map from the entire domain of input values to the entire range of the output activation
  - All values the activation function of the output neuron

The issue of depth

- Previous discussion showed that a *single-layer* MLP is a universal function approximator
  - Can approximate any function to arbitrary precision
  - But may require infinite neurons in the layer

- More generally, deeper networks will require far fewer neurons for the same approximation error
  - The network is a generic map
    - The same principles that apply for Boolean networks apply here
  - Can be exponentially fewer than the 1-layer network

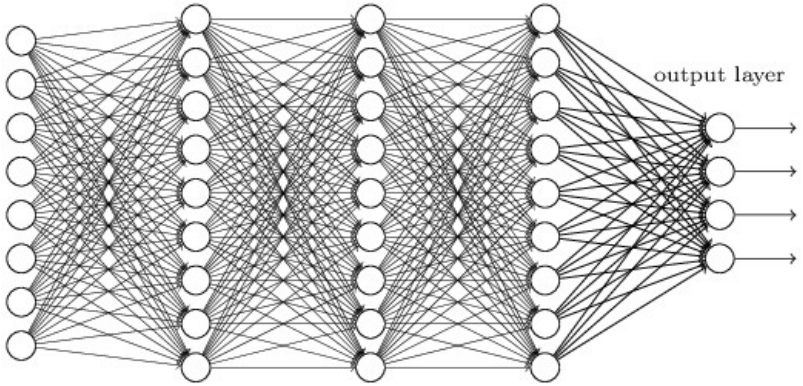
Recap

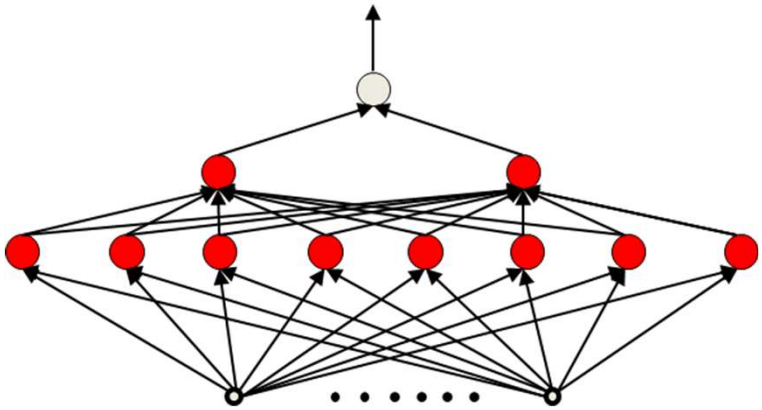


- **Neural networks are universal function approximators**
  - Can model any Boolean function
  - Can model any classification boundary
  - Can model any continuous valued function

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

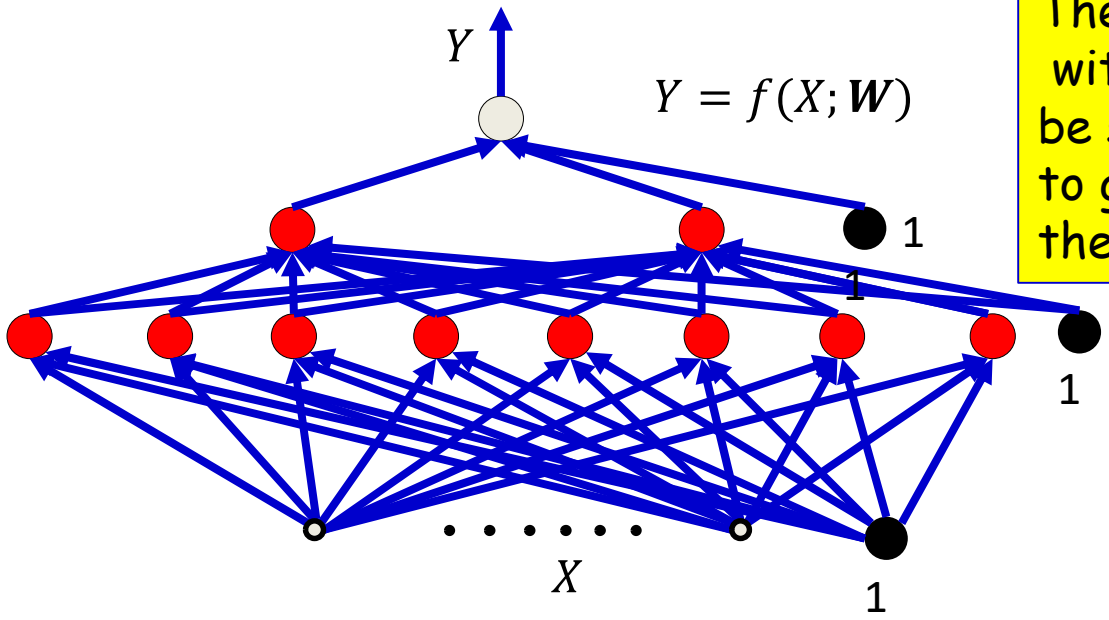
First: the structure of the network





- 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

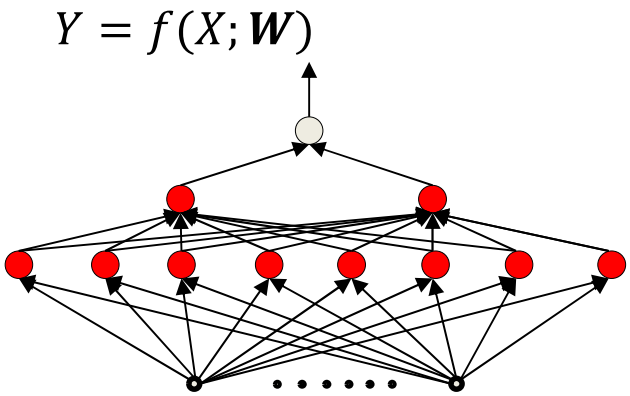
What we learn: The parameters of the network

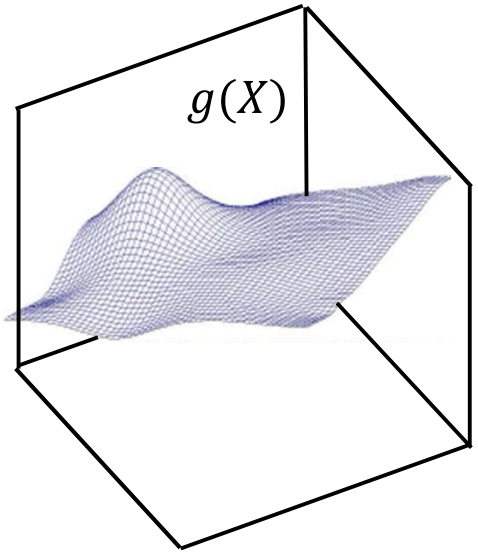


The network is a function f() with parameters W which must be set to the appropriate values to get the desired behavior from the net

- **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**

How to learn a network?



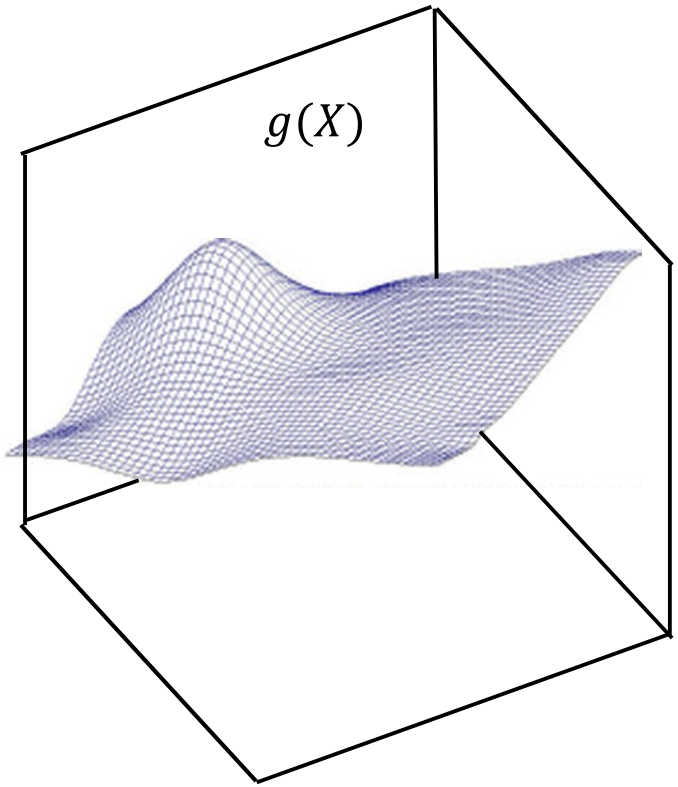


- When $f(X; \boldsymbol{W})$ has the capacity to exactly represent $g(X)$

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

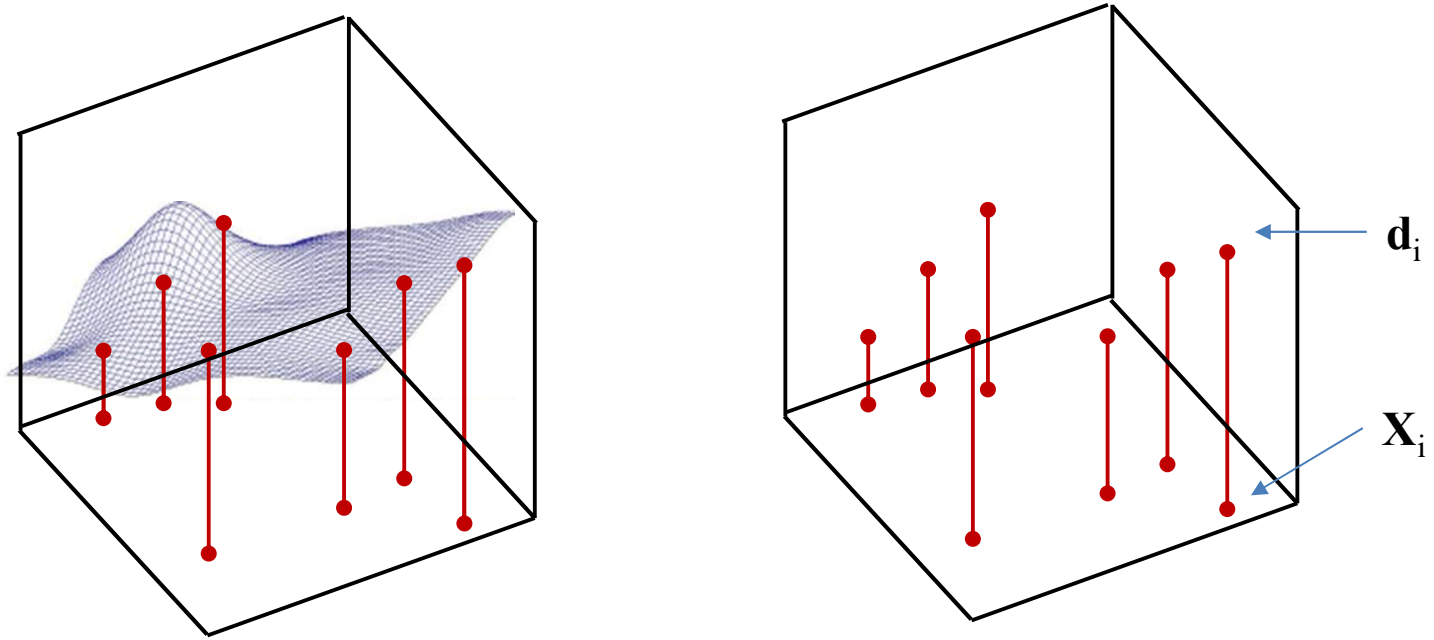
- $div()$ is a *divergence* function that goes to zero when $f(X; W) = g(X)$

Problem $g(X)$ is unknown



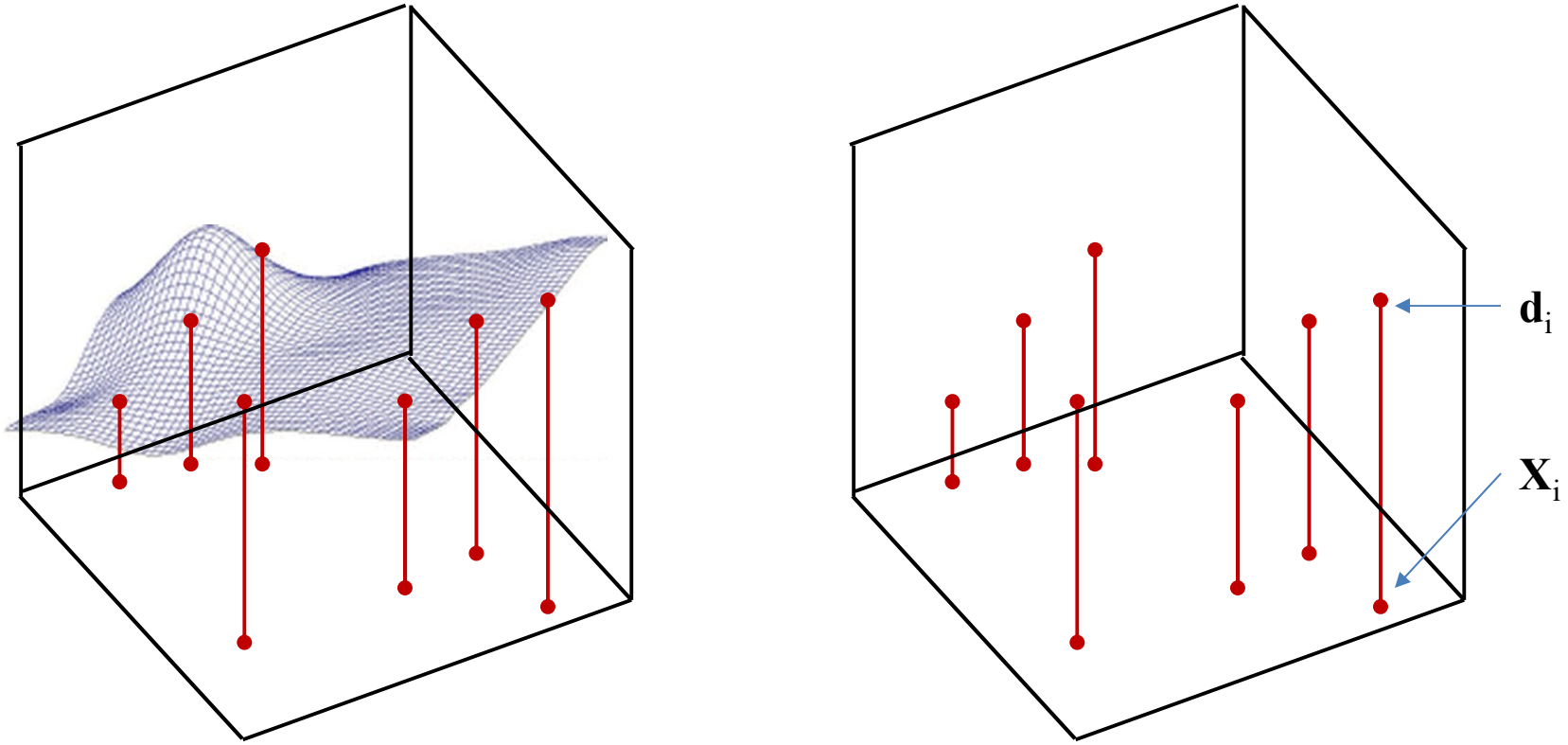
- Function $g(X)$ must be fully specified
  - Known *everywhere,* i.e. for *every* input X
- **In practice we will not have such specification**

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 $d_i = g(X_i) + noise$
  - 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

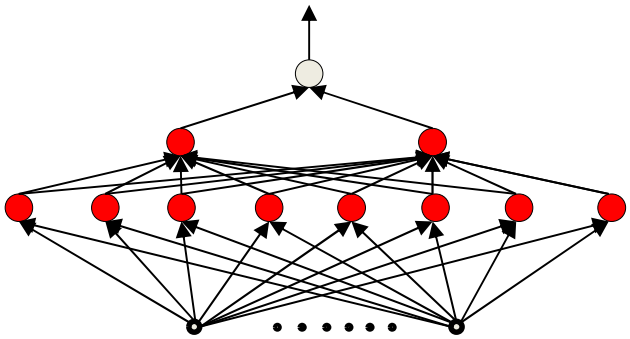
Drawing samples

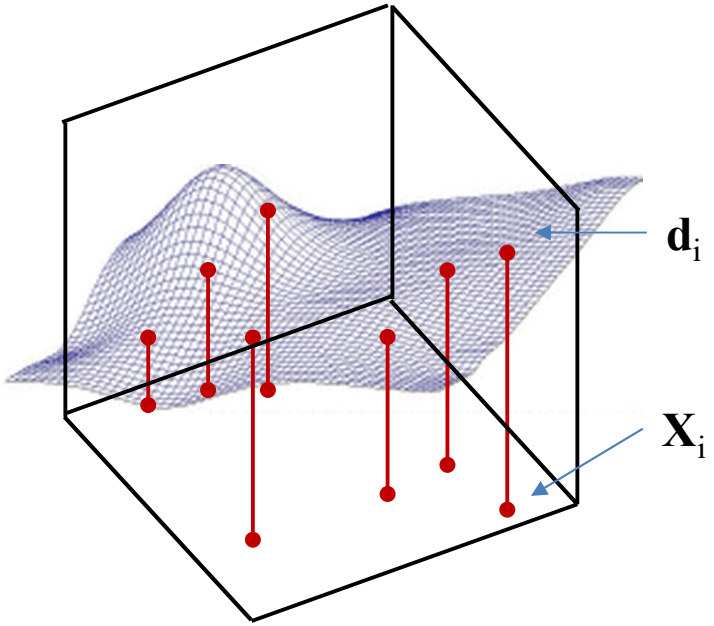


- We must ***learn*** the *entire* function from these few examples
  - The "training" samples

Learning the function

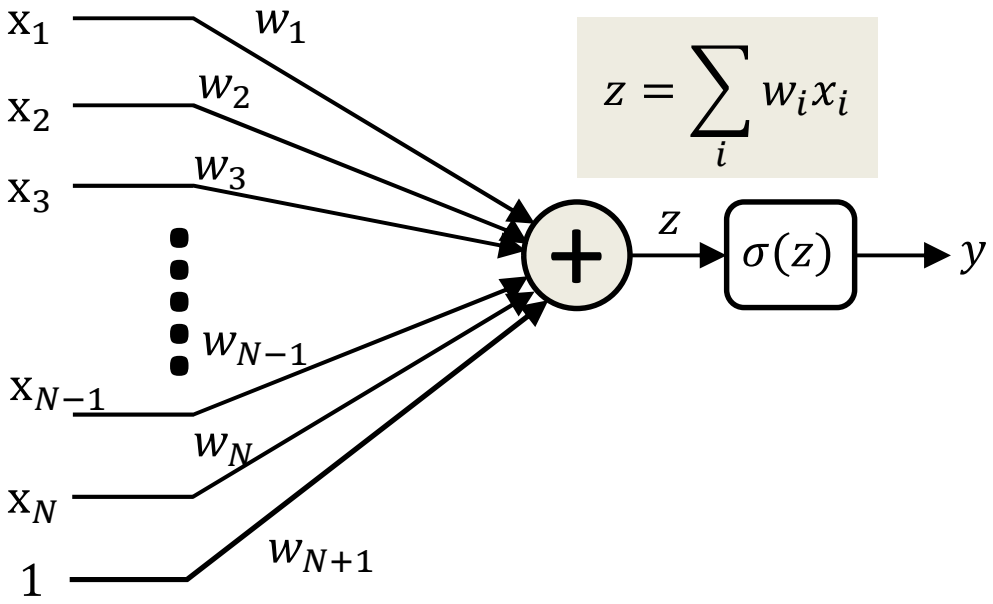
$Y = f(X; \boldsymbol{W})$





- 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

Perceptrons with differentiable activation functions



$$z = \sum_i w_i x_i$$

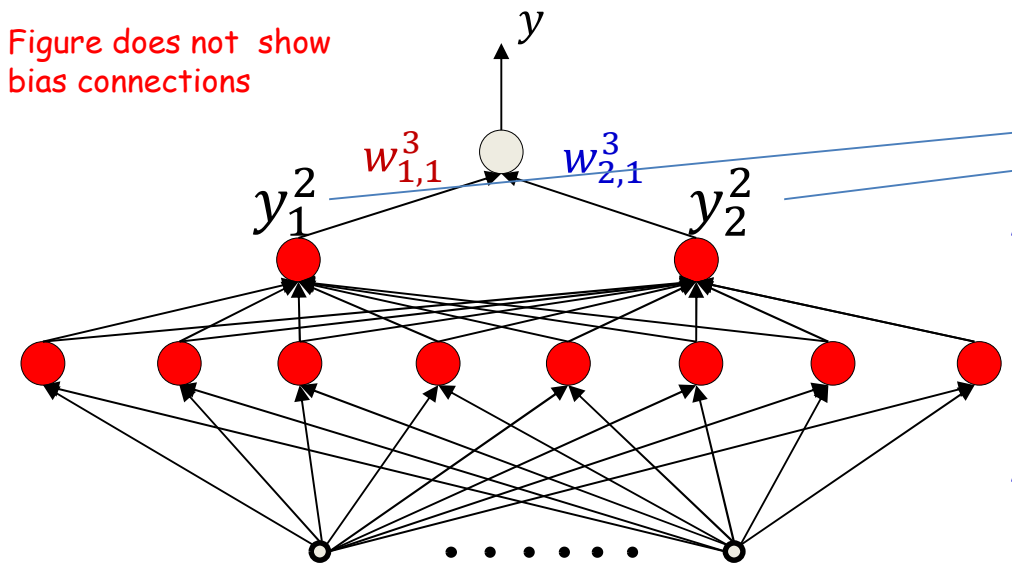
$$\frac{dy}{dz} = \sigma'(z)$$

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

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

- **$\sigma(z)$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



$$y = \sigma(w^3_{1,1}y^2_1 + w^3_{2,1}y^2_2 + w^3_{3,1})$$

y = output of overall network

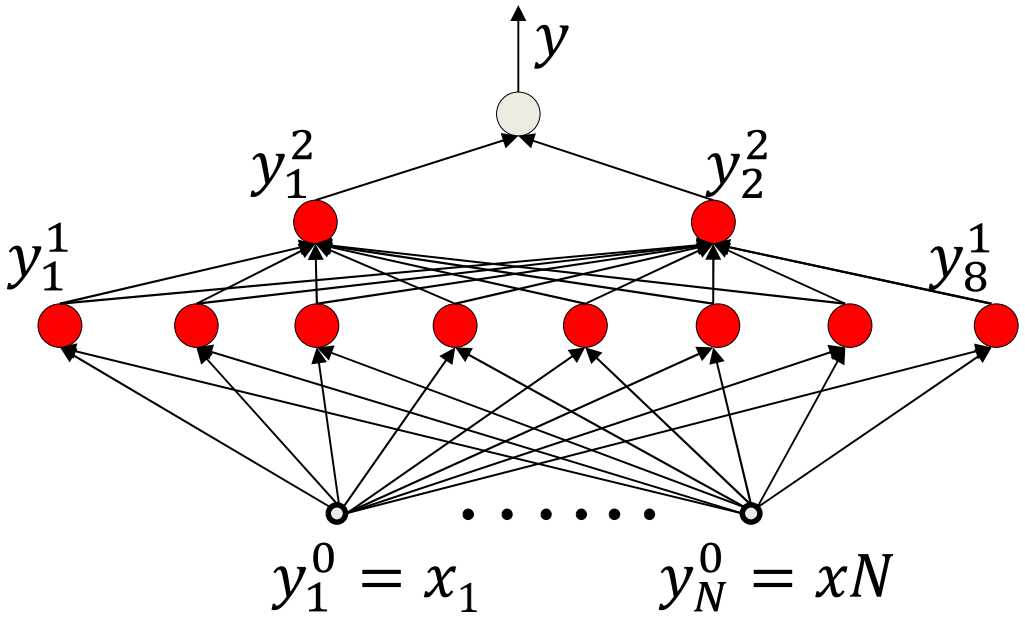
$w^k_{i,j}$ = weight connecting the ith unit of the (k-1)th layer to the jth unit of the k-th layer

y^k_i = output of the ith unit of the kth layer

$\sigma()$ is differentiable w.r.t both w and y^k_i

- 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

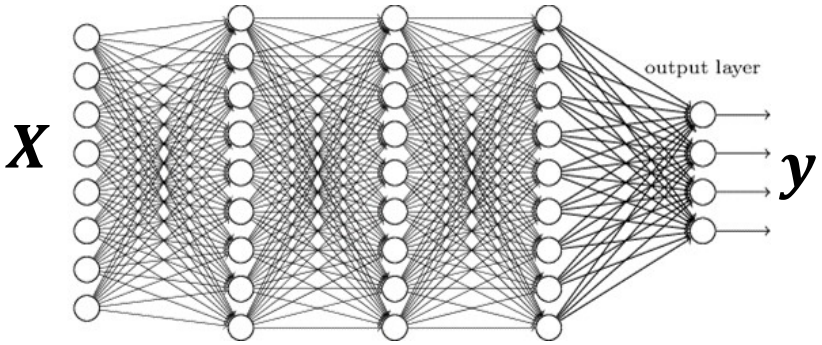
Overall function is differentiable

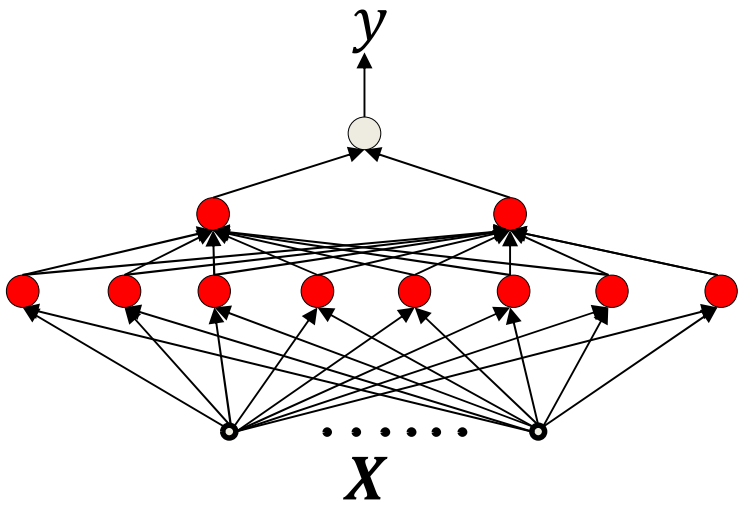


$$y_j^k = \sigma\left(\sum_i w_{i,j}^{k-1} y_i^{k-1}\right)$$

- The overall function is differentiable w.r.t every parameter
  - 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

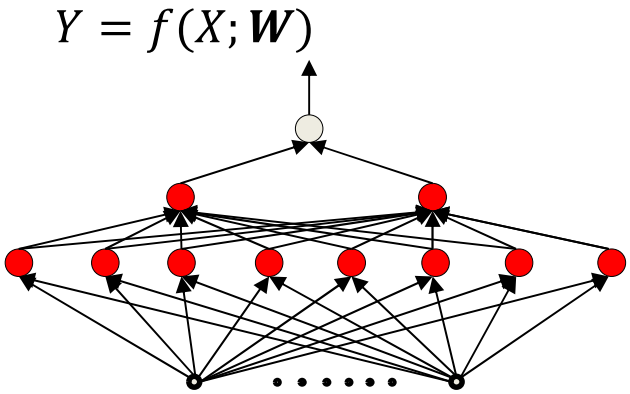
Overall setting for "Learning" the MLP

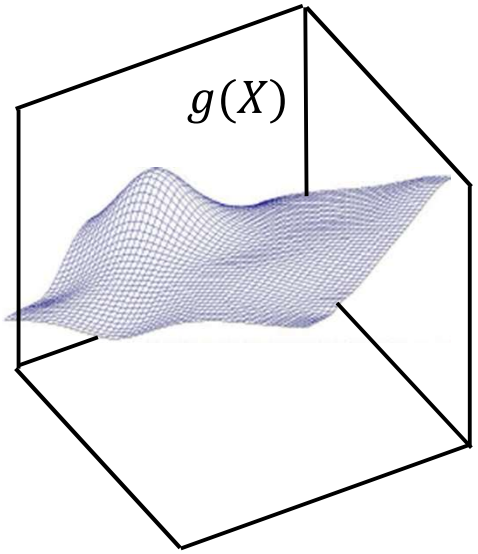




- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_N, \boldsymbol{d}_N)$ …
  - $\boldsymbol{d}$ is the *desired output* of the network in response to $\boldsymbol{X}$
  - $\boldsymbol{X}$ and $\boldsymbol{d}$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



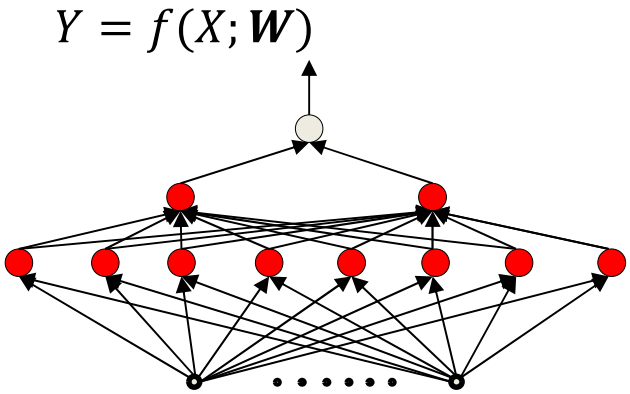


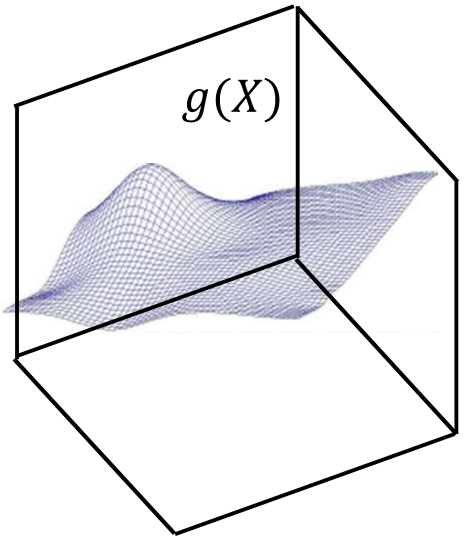
- When $f(X; \boldsymbol{W})$ has the capacity to exactly represent $g(X)$

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

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

Minimizing *expected* error



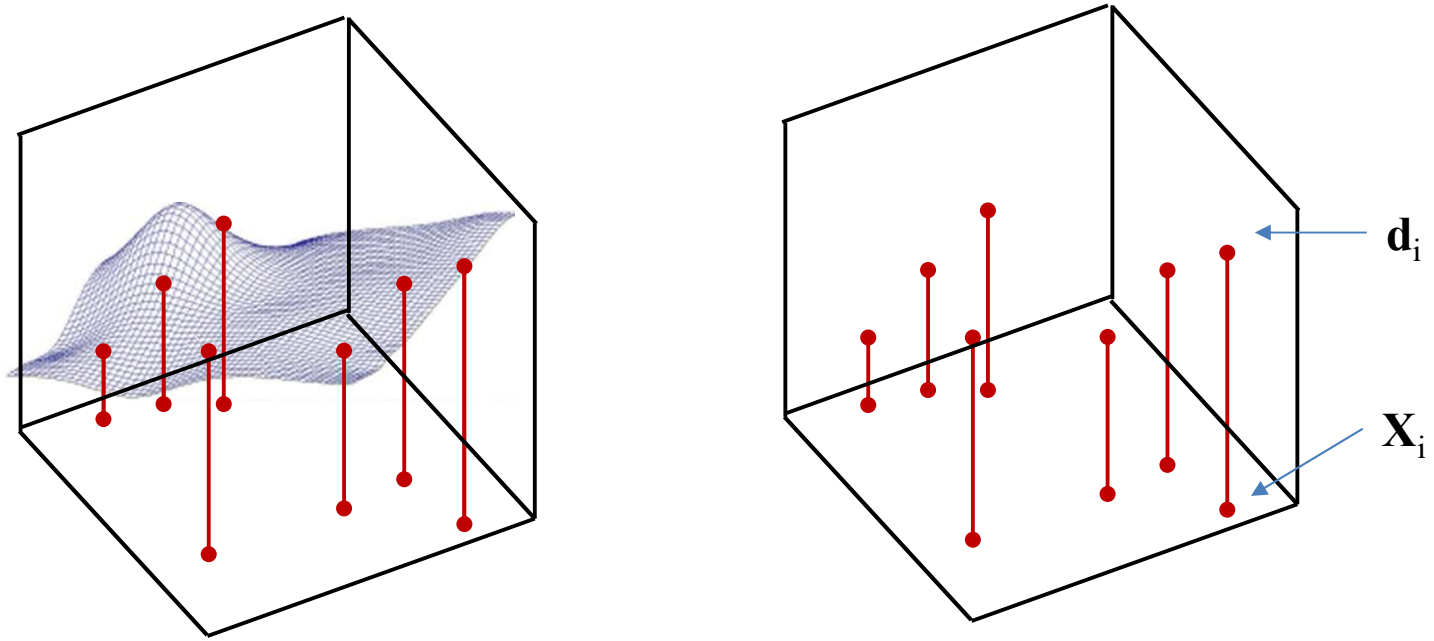


- More generally, assuming X is a random variable

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

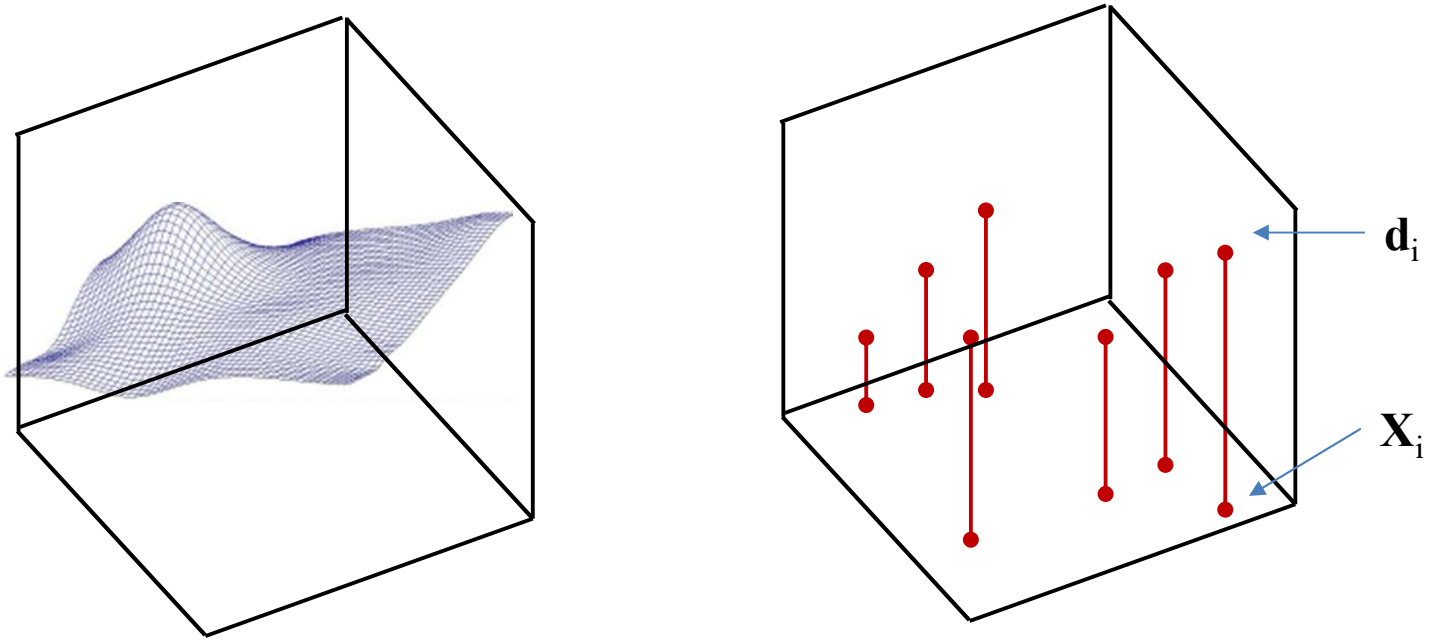
$$= \underset{W}{\operatorname{argmin}}\, E\big[div\big(f(X;W), g(X)\big)\big]$$

Recap: Sampling the function



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

The *Empirical* risk



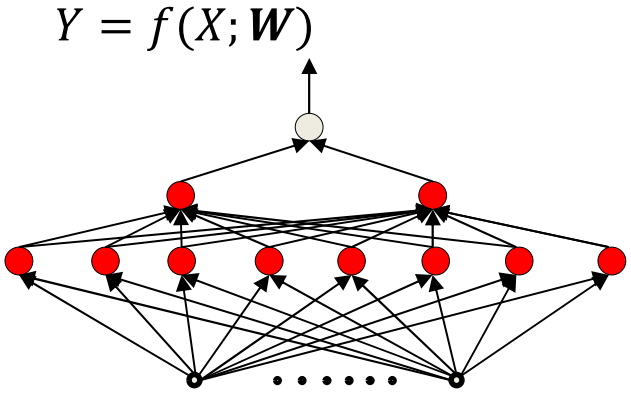
- The *expected* divergence (or risk) is the average divergence over the entire input space

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

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

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

Empirical Risk Minimization



- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_N, \boldsymbol{d}_N)$
  - Quantification of error on the i[th] 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

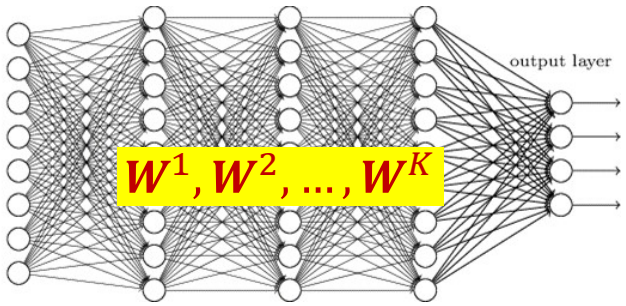
$$Loss(W) = \frac{1}{N}\sum_{i} div(f(X_i; W), d_i)$$

- 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

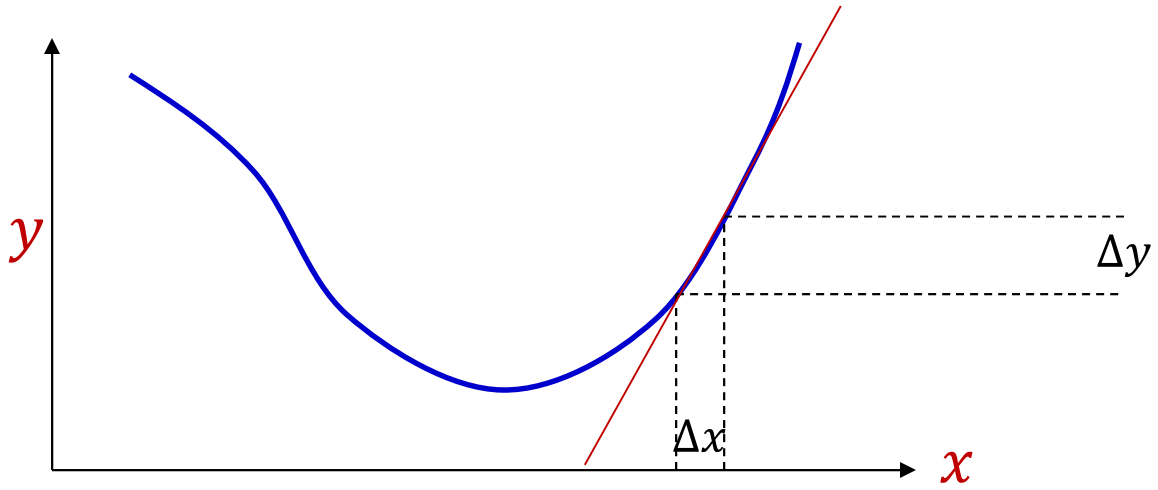
- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_N, \boldsymbol{d}_N)$
- Minimize the following function

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

w.r.t W

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

A brief note on derivatives..



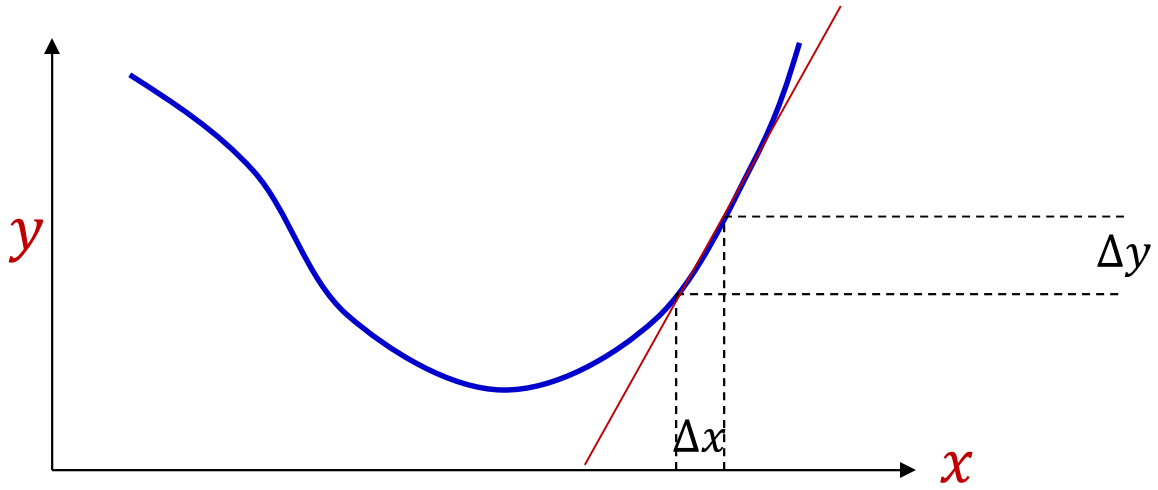
derivative

- 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

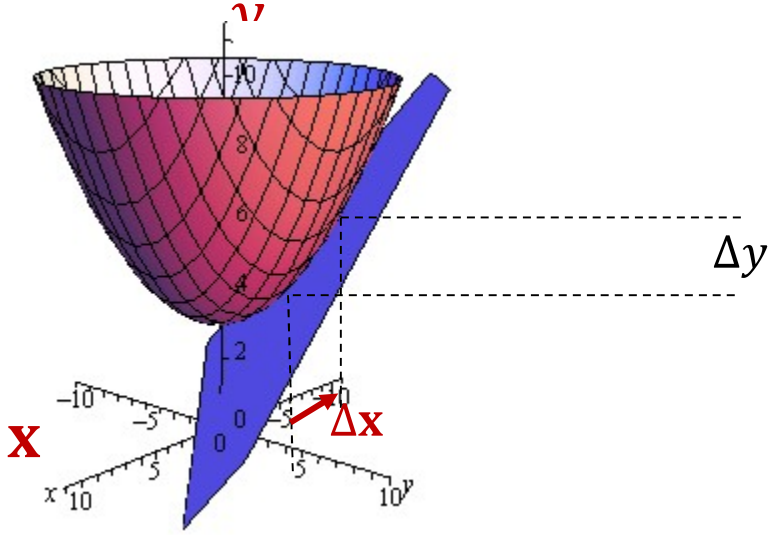
$$y = f(x)$$

  - Derivative:

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

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

Multivariate scalar function:
Scalar function of *vector* argument



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

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

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

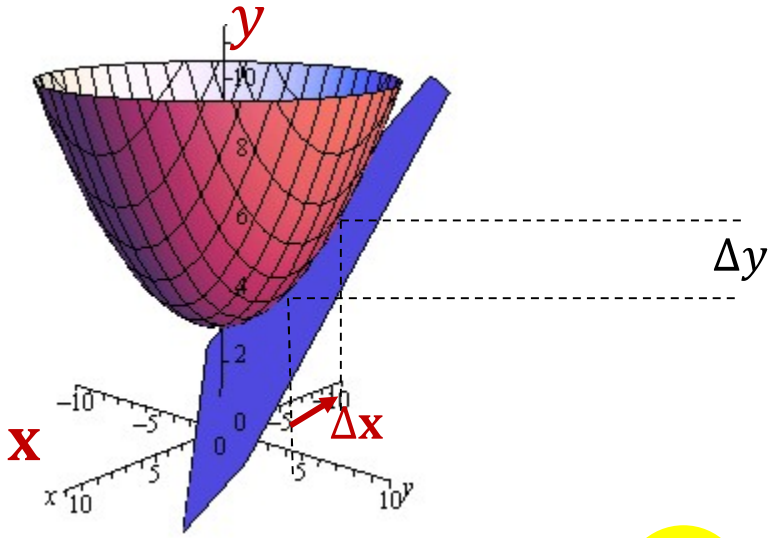
- Giving us that α is a row vector: $\alpha = [\alpha_1 \quad \cdots \quad \alpha_D]$

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

- 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

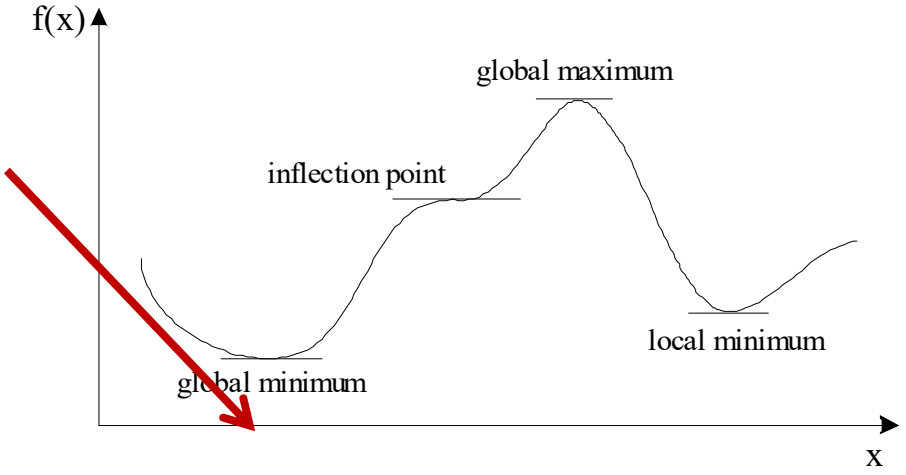
$$\nabla_{\mathbf{x}} y = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \cdots & \frac{\partial y}{\partial x_D} \end{bmatrix}$$

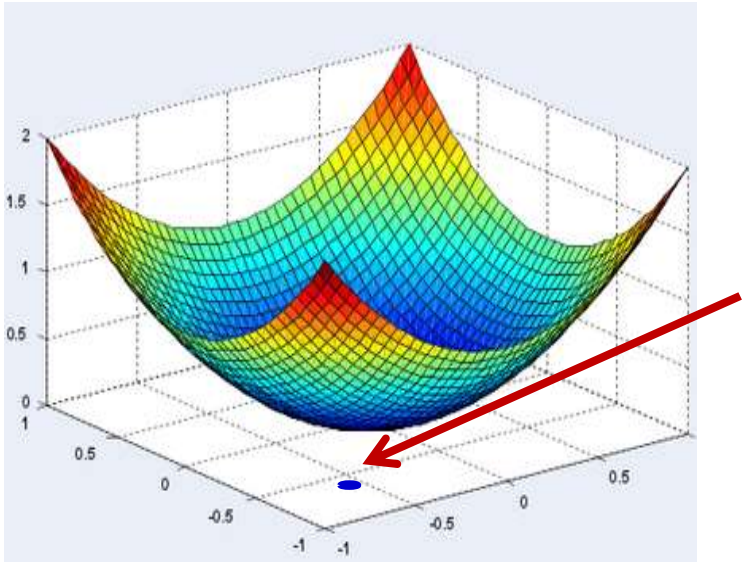
  - You may be more familiar with the term "gradient" which is actually defined as the transpose of the derivative

Caveat about following slides

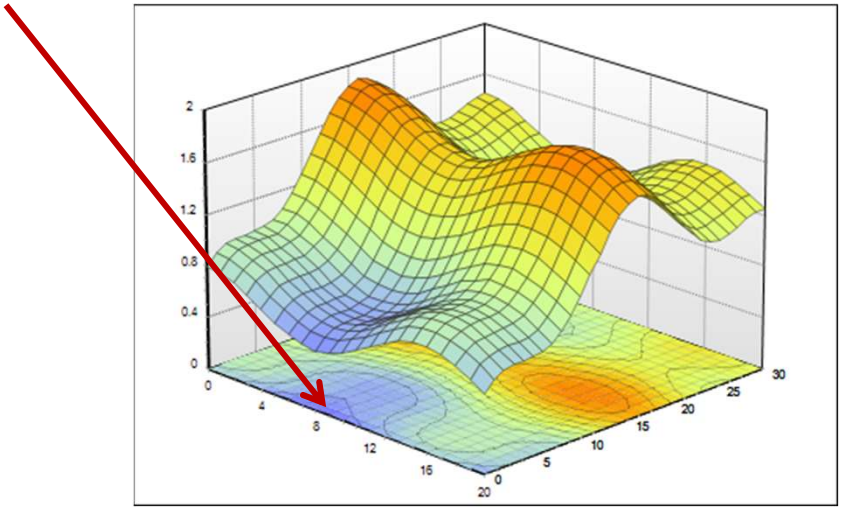
- The following slides speak of optimizing a function w.r.t a variable “x”
- This is only mathematical notation. In our actual network optimization problem we would be optimizing w.r.t. network weights “w”
- To reiterate – “x” in the slides represents the variable that we’re optimizing a function over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

The problem of optimization

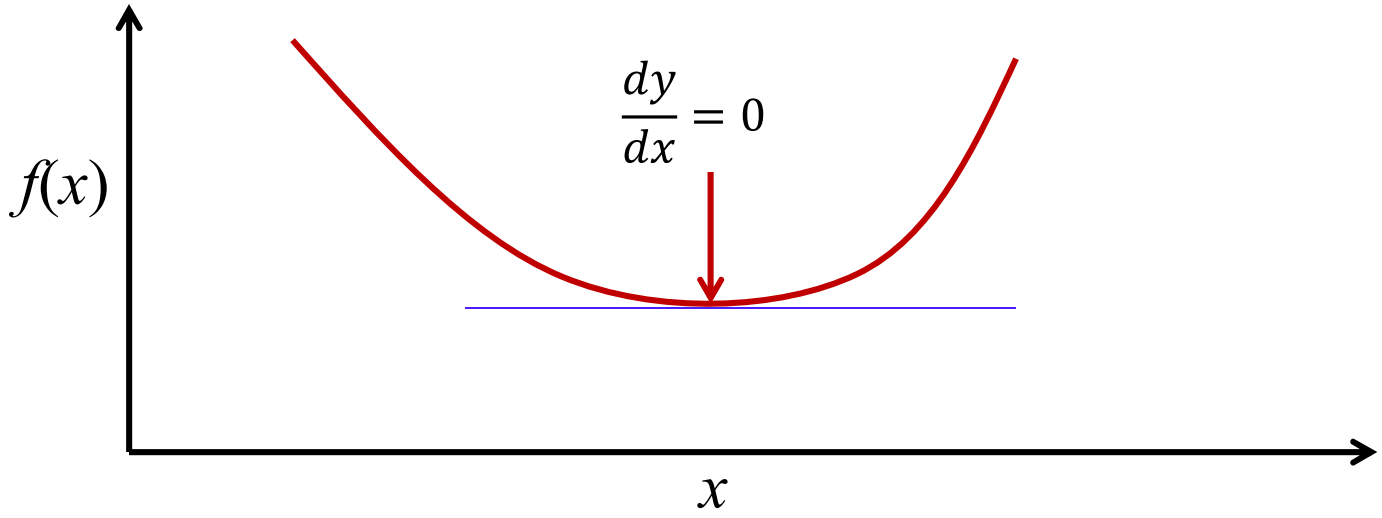




- General problem of optimization: find the value of x where $\mathbf{f}(x)$ is minimum



Finding the minimum of a function

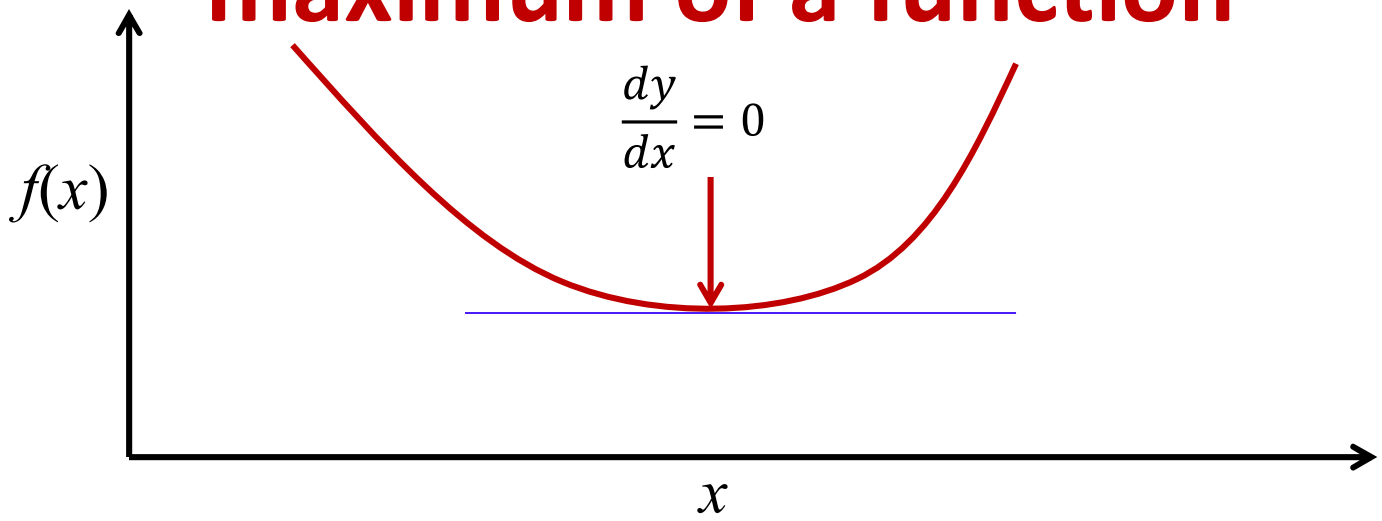


- Find the value x at which $f'(x) = 0$
  - Solve

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

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

Solution: Finding the minimum or maximum of a function



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

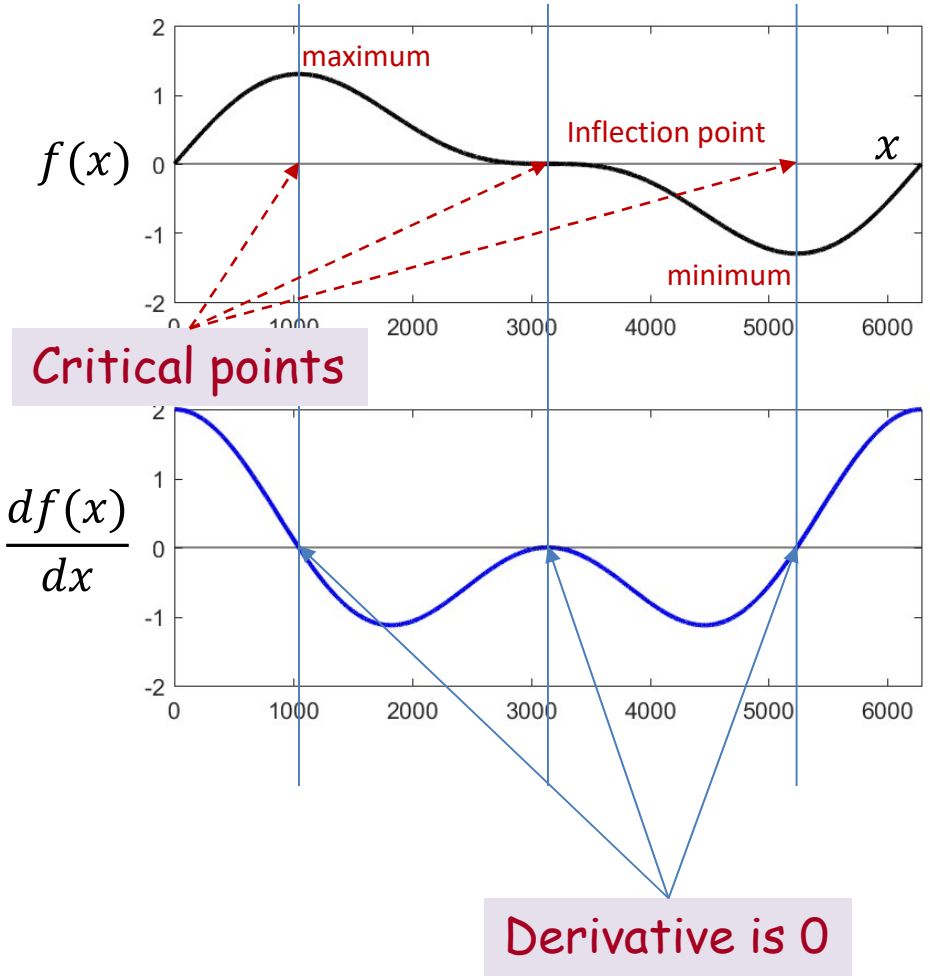
$$\frac{df(x)}{dx} = 0$$

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

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

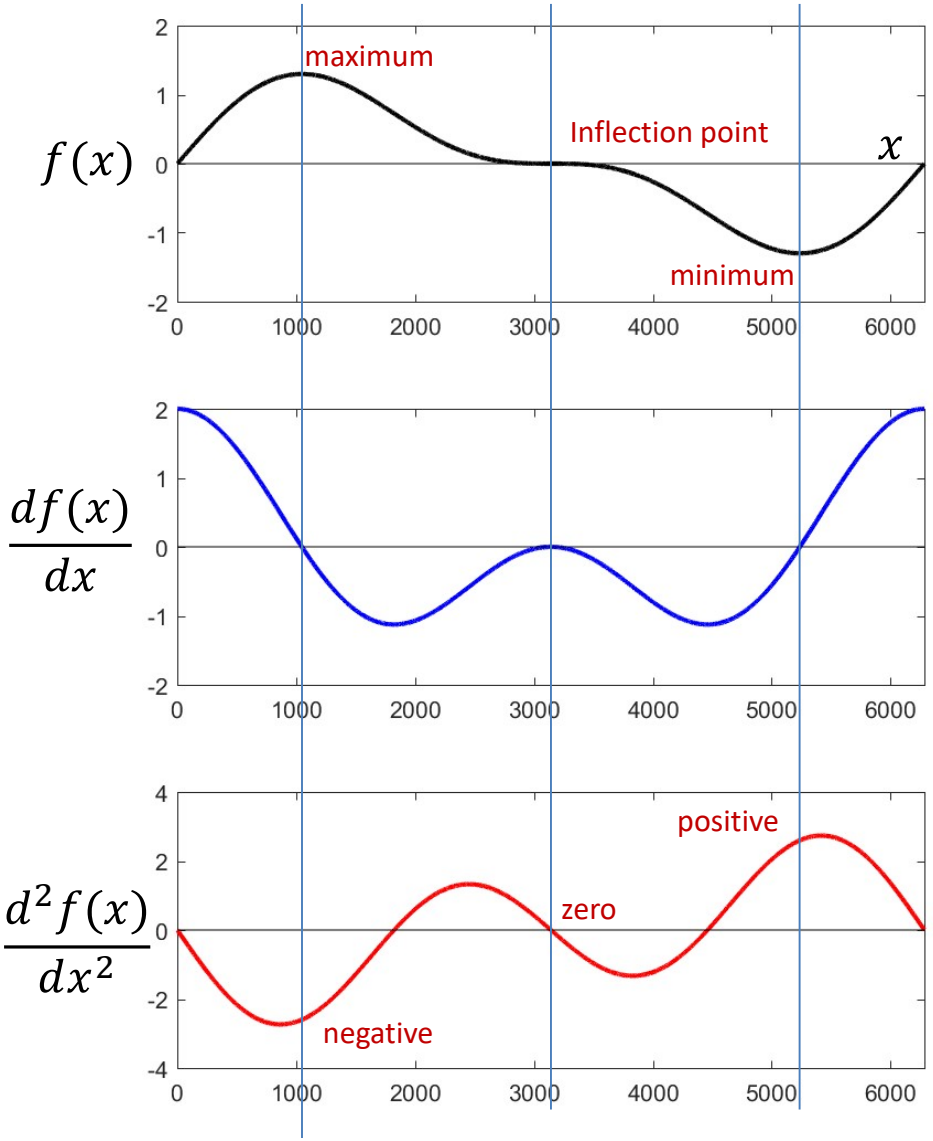
- If $f''(x_{soln})$ is positive x_{soln} is a minimum, otherwise it is a maximum

A note on derivatives of functions of single variable



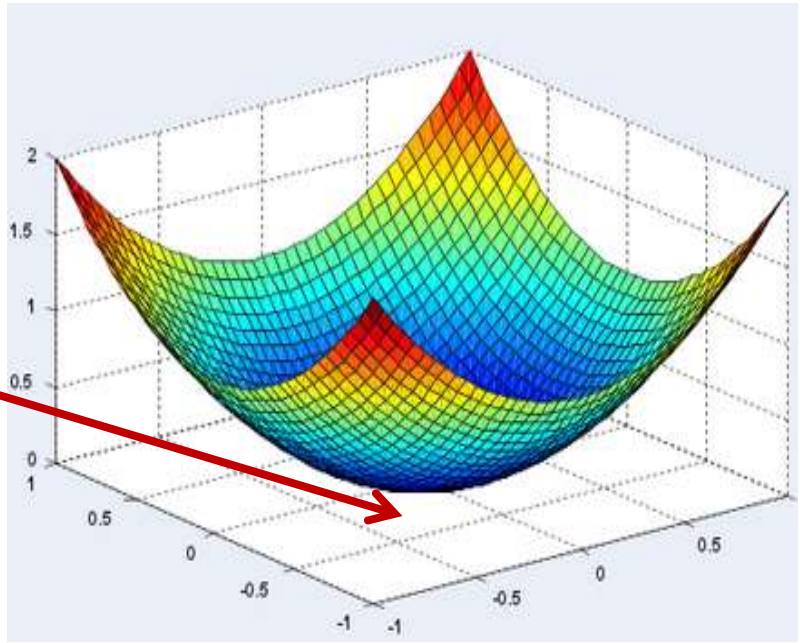
- All locations with zero derivative are *critical* points
  - These can be local maxima, local minima, or inflection points
- The *second* derivative is
  - Positive (or 0) at minima
  - Negative (or 0) at maxima
  - Zero at inflection points

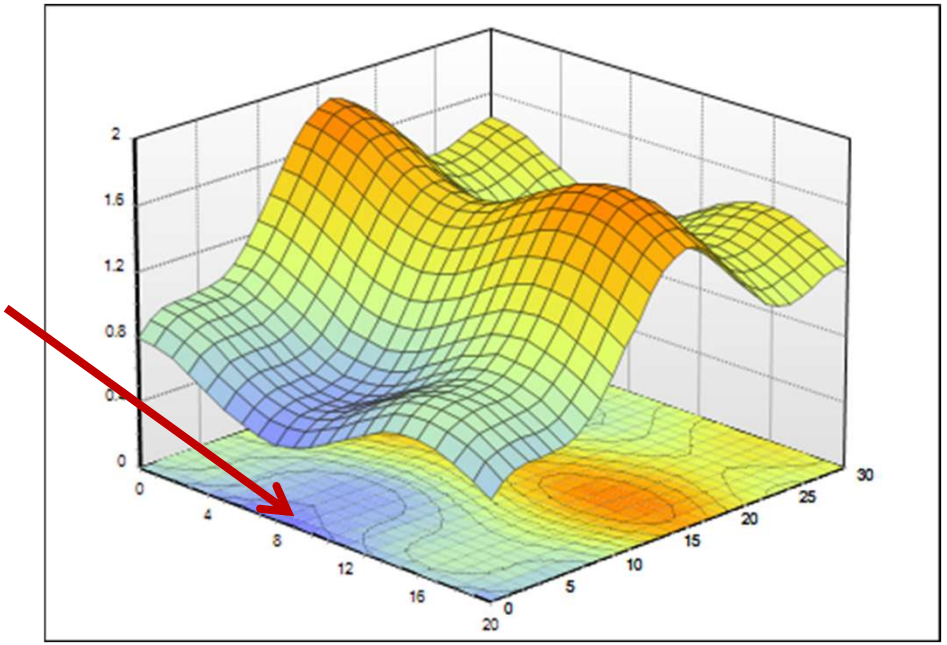
A note on derivatives of functions of single variable



- 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?

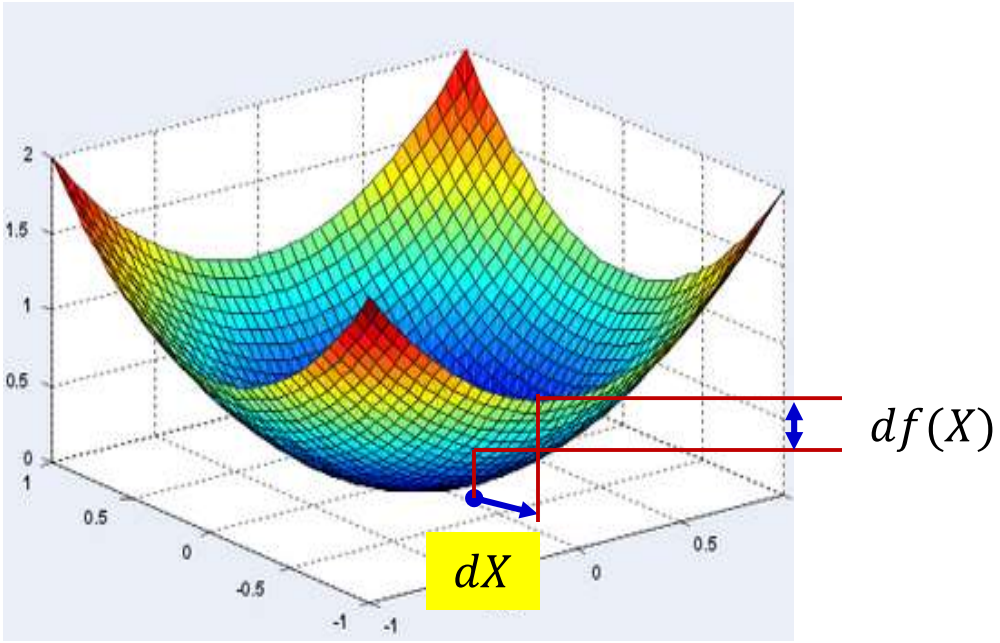




- The optimum point is still "turning" point
  - Shifting in any direction will increase the value
  - For smooth functions, miniscule shifts will not result in any change at all
- We must find a point where shifting in any direction by a microscopic amount will not change the value of the function

A brief note on derivatives of multivariate functions

The *Gradient* of a scalar function



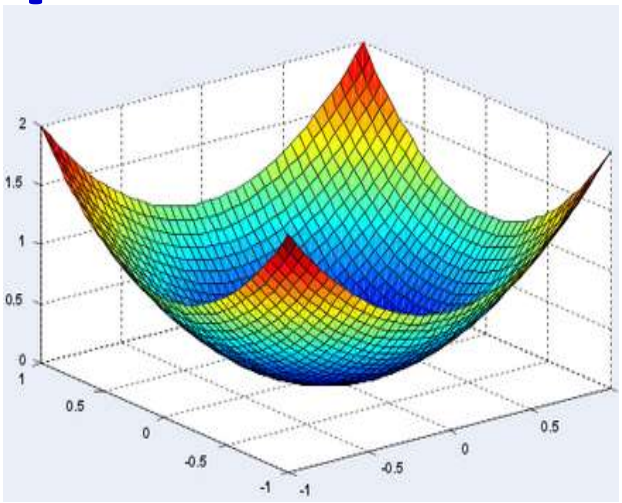
- The *derivative* $\nabla_X f(X)$ of a scalar function $f(X)$ of a multi-variate input X is a multiplicative factor that gives us the change in $f(X)$ for tiny variations in X

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

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

Gradients of scalar functions with multi-variate inputs

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



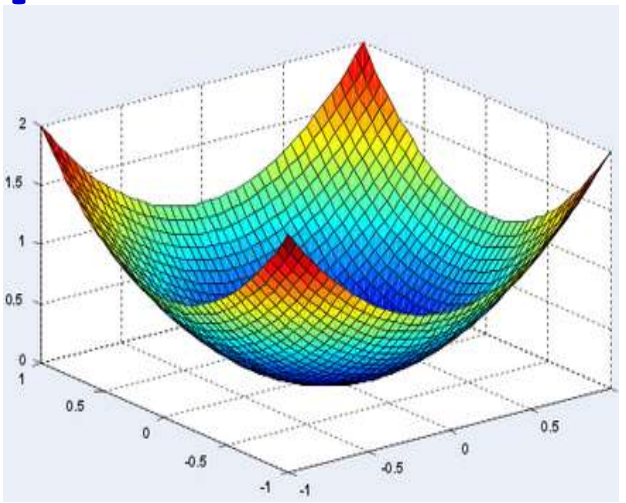
$$\nabla_X f(X) = \left[\frac{\partial f(X)}{\partial x_1} \quad \frac{\partial f(X)}{\partial x_2} \quad \cdots \quad \frac{\partial f(X)}{\partial x_n}\right]$$

- Relation:

$$df(X) = \nabla_X f(X) dX$$
$$= \frac{\partial f(X)}{\partial x_1} dx_1 + \frac{\partial f(X)}{\partial x_2} dx_2 + \cdots + \frac{\partial f(X)}{\partial x_n} dx_n$$

Gradients of scalar functions with multivariate inputs

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



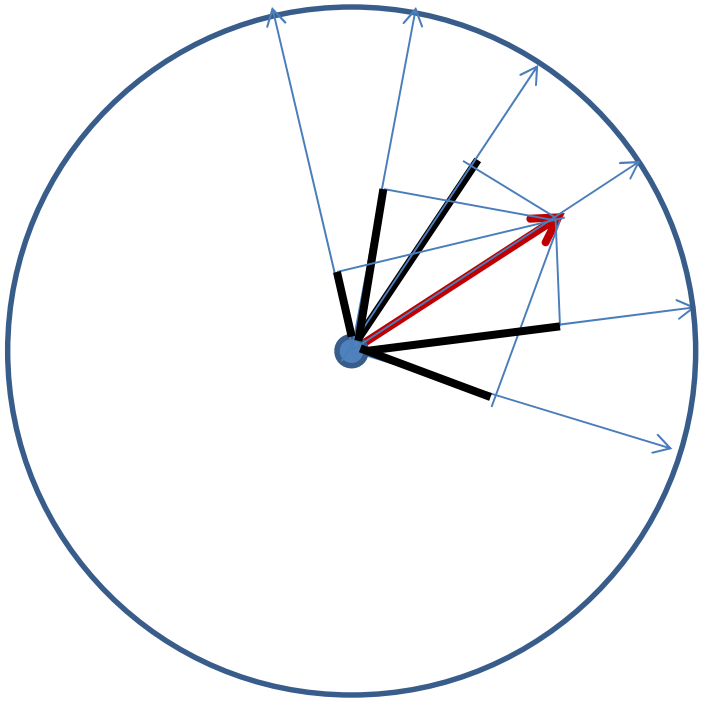
$$\nabla_X f(X) = \left[\frac{\partial f(X)}{\partial x_1} \quad \frac{\partial f(X)}{\partial x_2} \quad \cdots \quad \frac{\partial f(X)}{\partial x_n}\right]$$

- Relation:

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

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

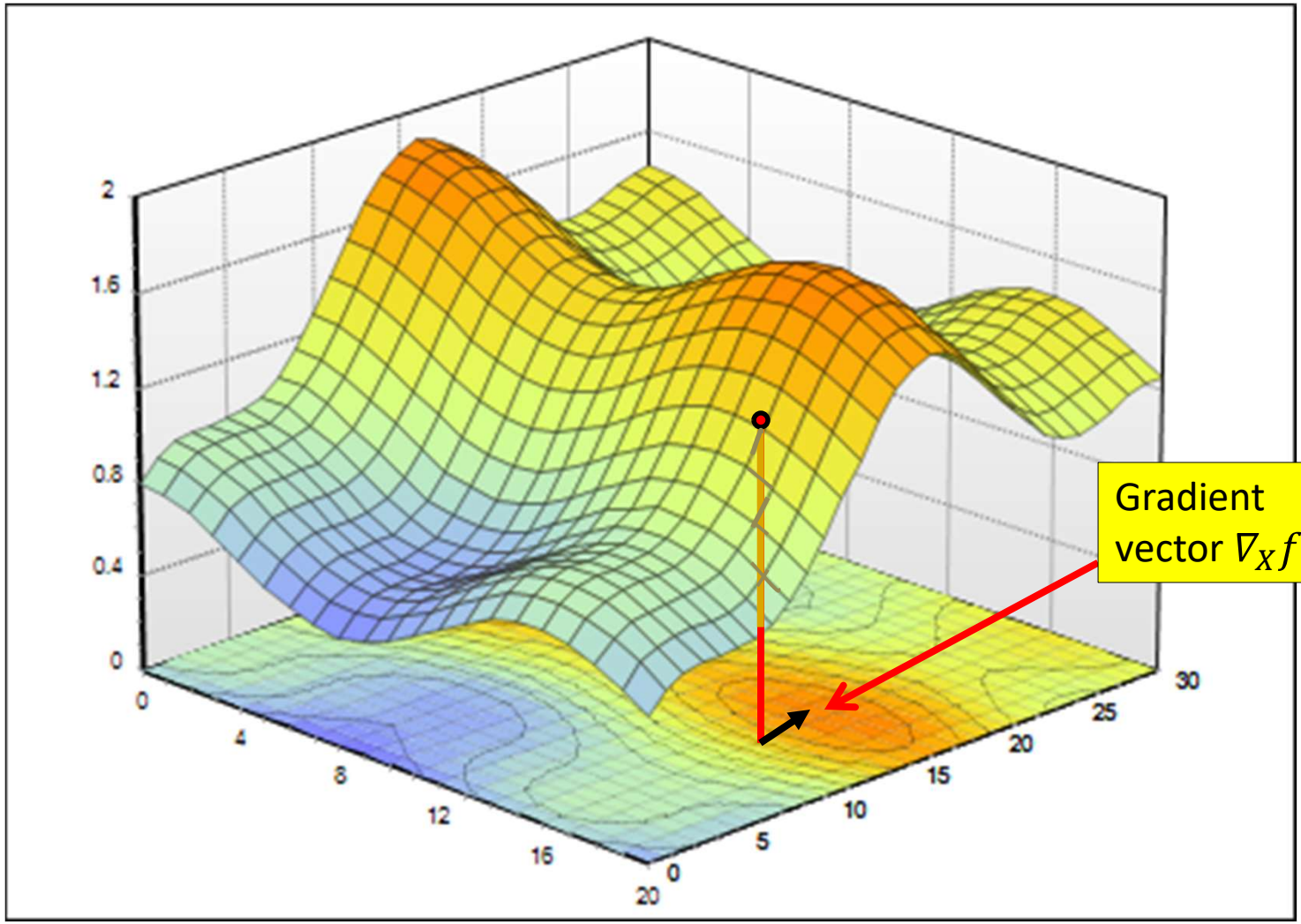
- $df(X) = \nabla_X f(X) dX$
  - The inner product between $\nabla_X f(X)^{\mathrm{T}}$ and dX

- Fixing the length of dX
  - E.g. $|dX| = 1$

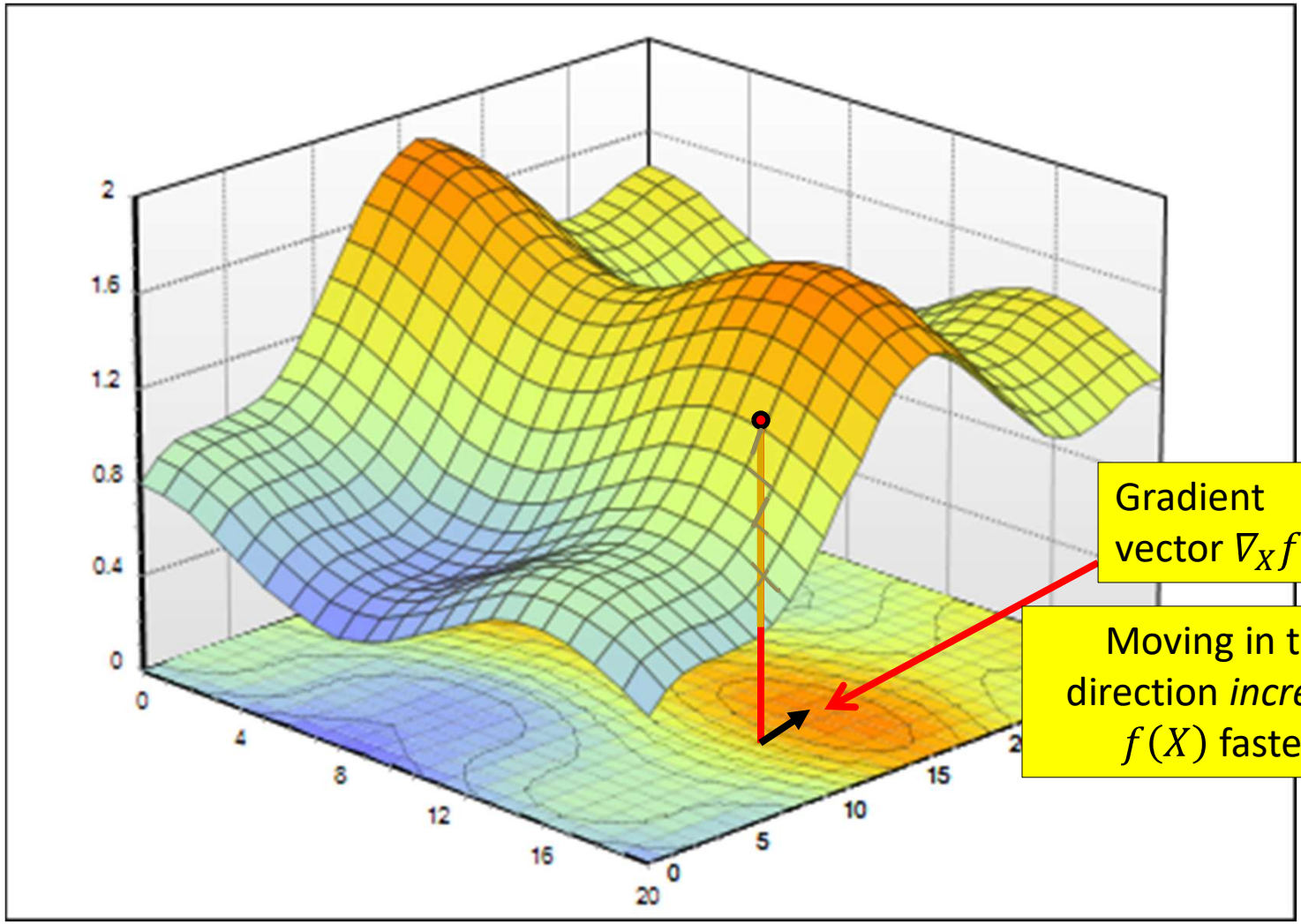
- $df(X)$ is max if dX is aligned with $\nabla_X f(X)^{\mathrm{T}}$
  - $\angle(\nabla_X f(X)^{\mathrm{T}}, dX) = 0$
  - The function $f(X)$ increases most rapidly if the input increment dX is perfectly aligned to $\nabla_X f(X)^{\mathrm{T}}$

- The gradient is the direction of fastest increase in $f(X)$

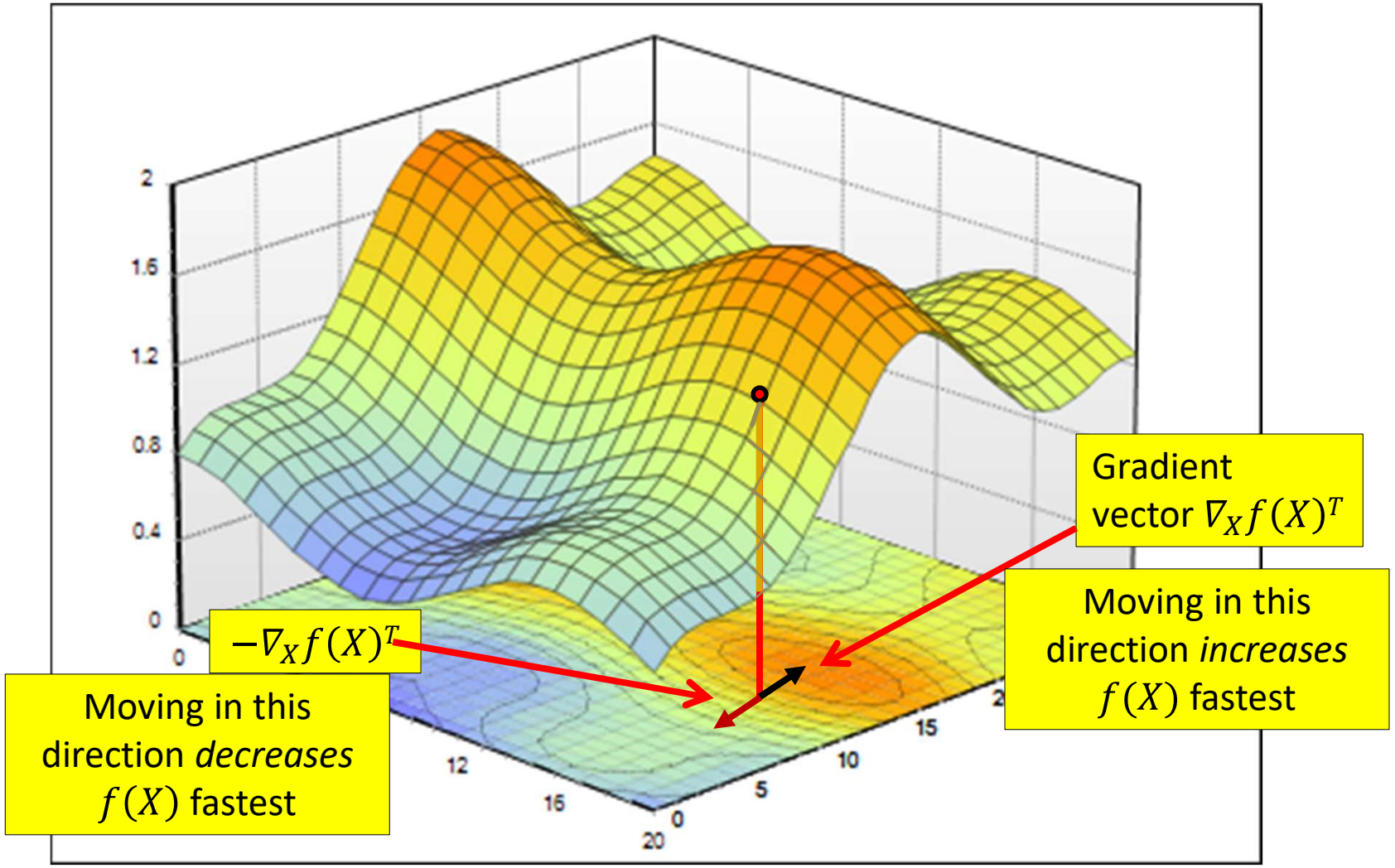
Gradient


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 $\nabla_X f(X)^T$

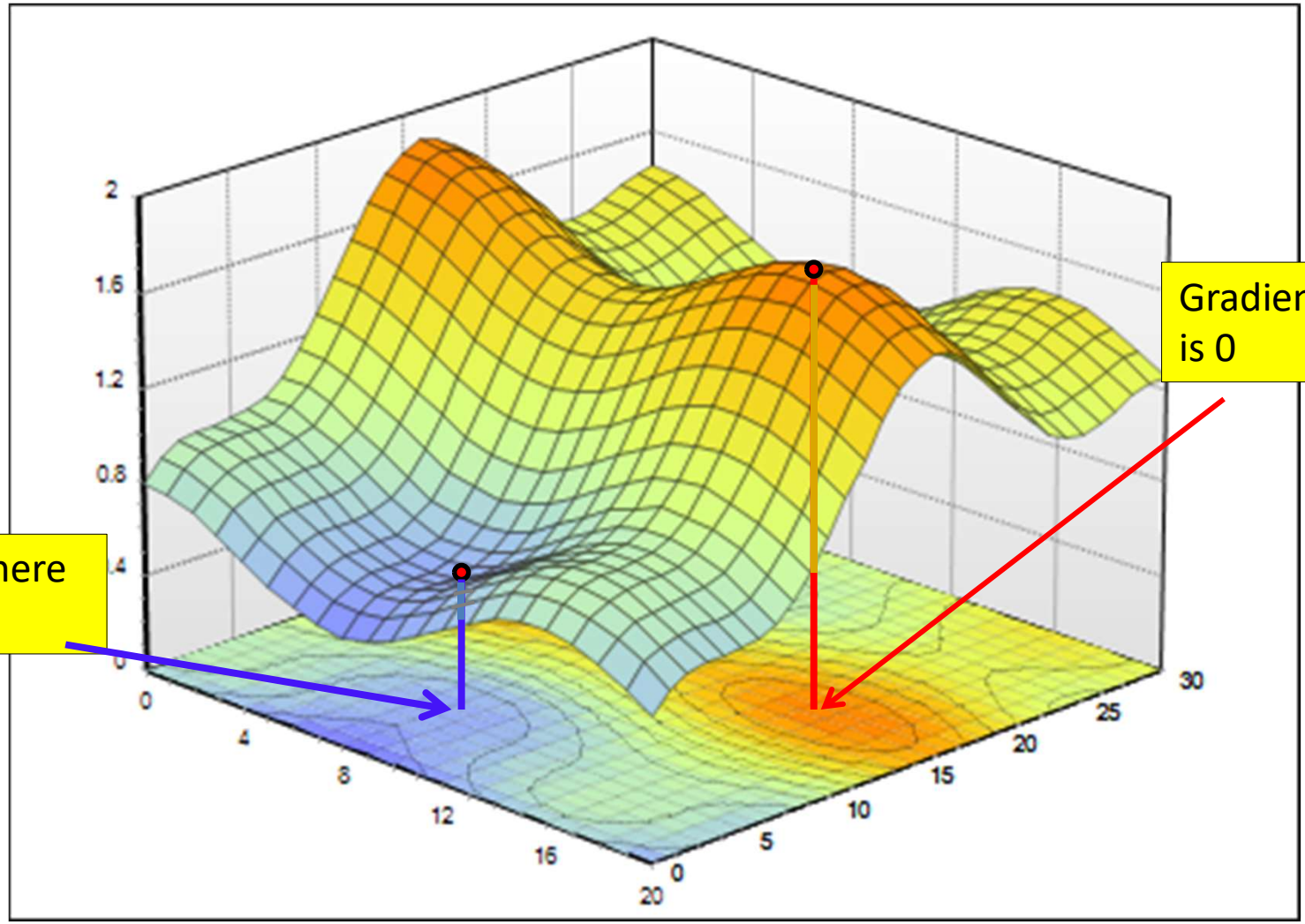
Gradient


2
1.6
1.2
0.8
0.4
0
0
4
8
12
16
20
0
5
10
15
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

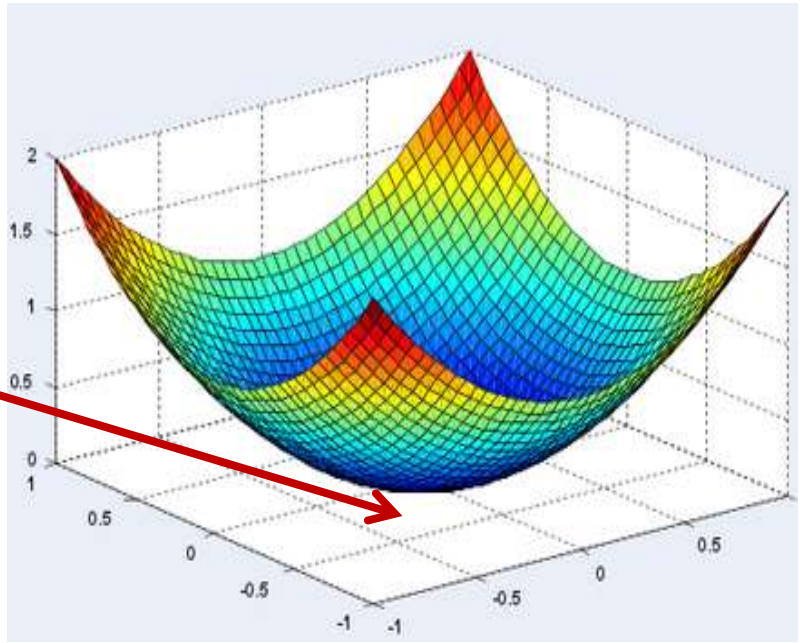

2
1.6
1.2
0.8
0
4
8
12
16
20
0
5
10
15
20
25
30
Gradient here is 0
Gradient here is 0

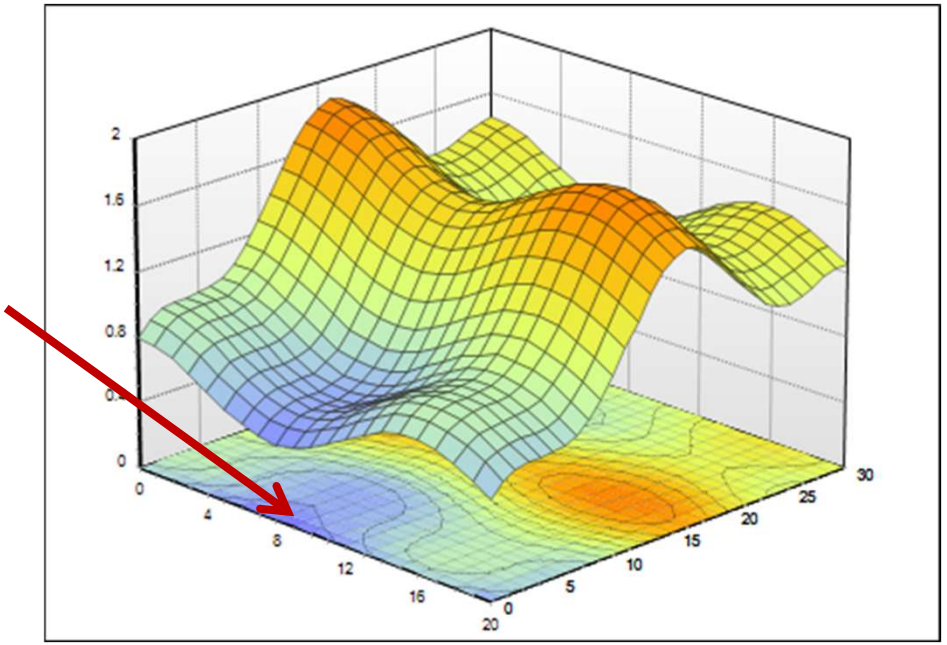
The Hessian

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

$$\nabla^2_X f(x_1,\ldots,x_n) := \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^{\,2}} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^{\,2}} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^{\,2}} \end{bmatrix}$$

Finding the minimum of a scalar function of a multivariate 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