Intro to Neural Networks
Part 1: What is a neural network

Lisbon Machine Learning School

July 2017

What's in this tutorial

- We will learn about
  - What is a neural network
  - What can neural networks model
  - Issues with learning
  - Some basic models: CNNs and RNNs

Instructor

- Bhiksha Raj

  Professor,
  Language Technologies Institute
  Carnegie Mellon Univ.

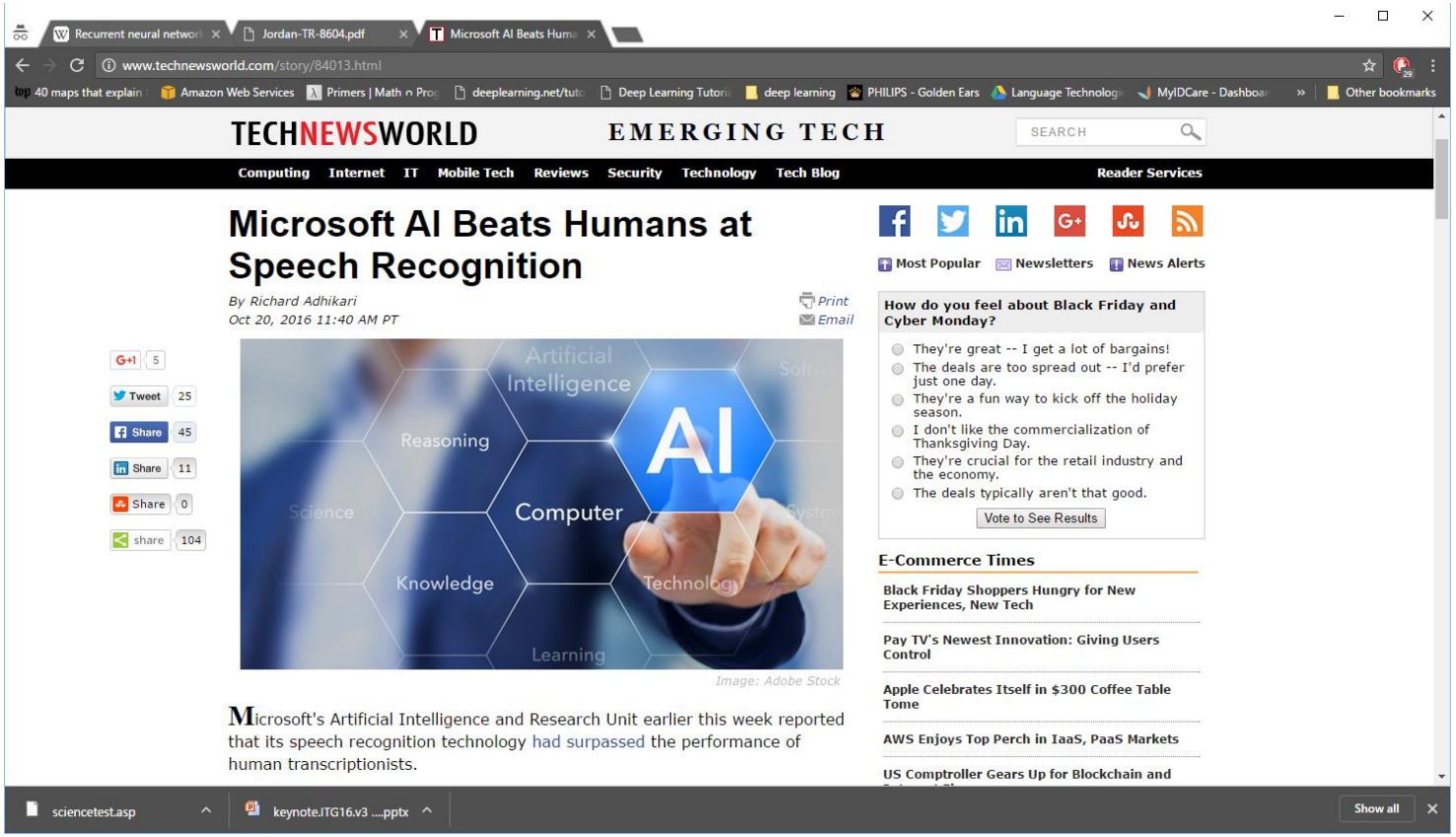
- bhiksha@cs.cmu.edu

Neural Networks are taking over!

- Neural networks have become one of the major thrust areas recently in various pattern recognition, prediction, and analysis problems

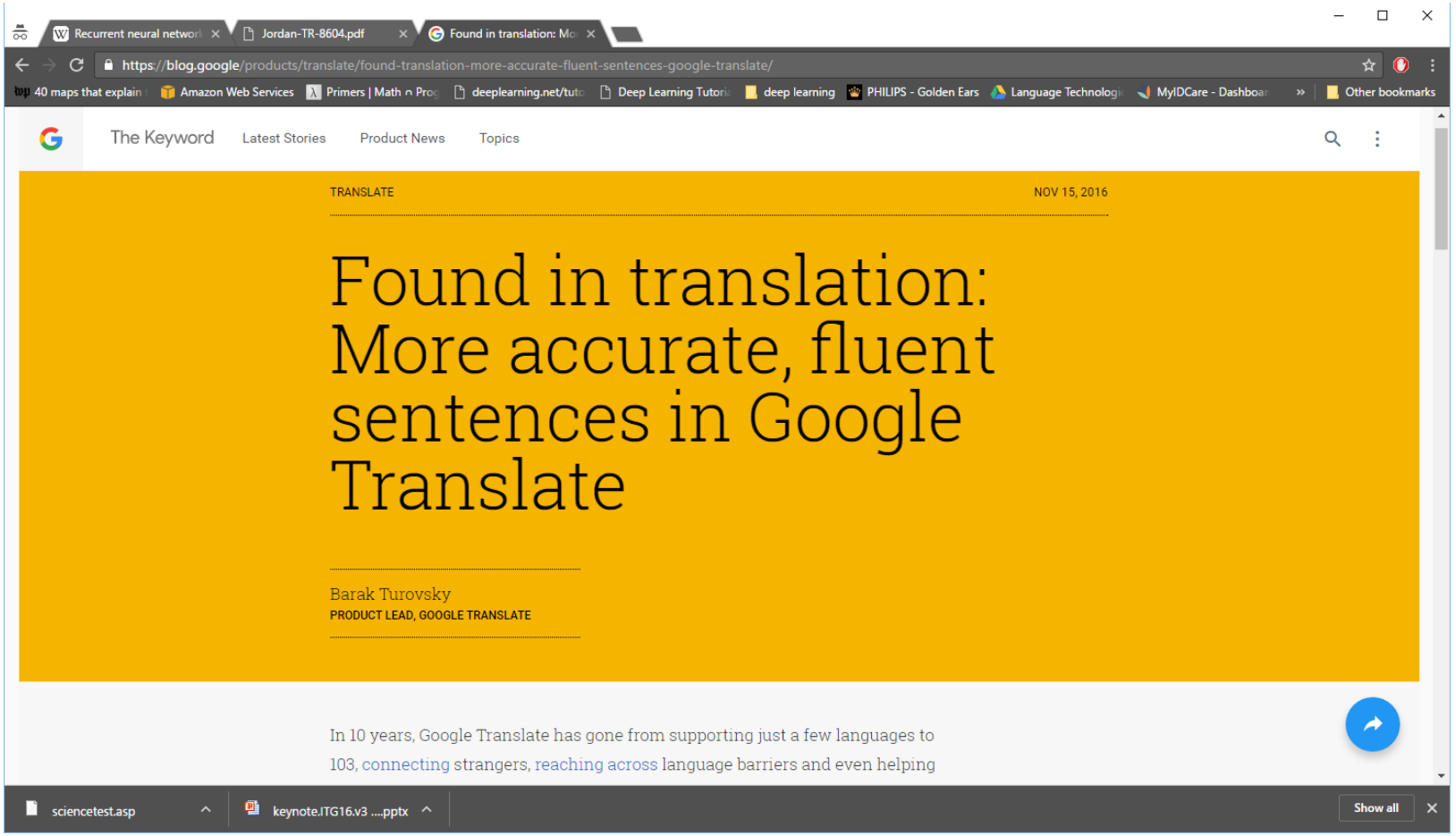
- In many problems they have established the state of the art
  - Often exceeding previous benchmarks by large margins

Recent success with neural networks



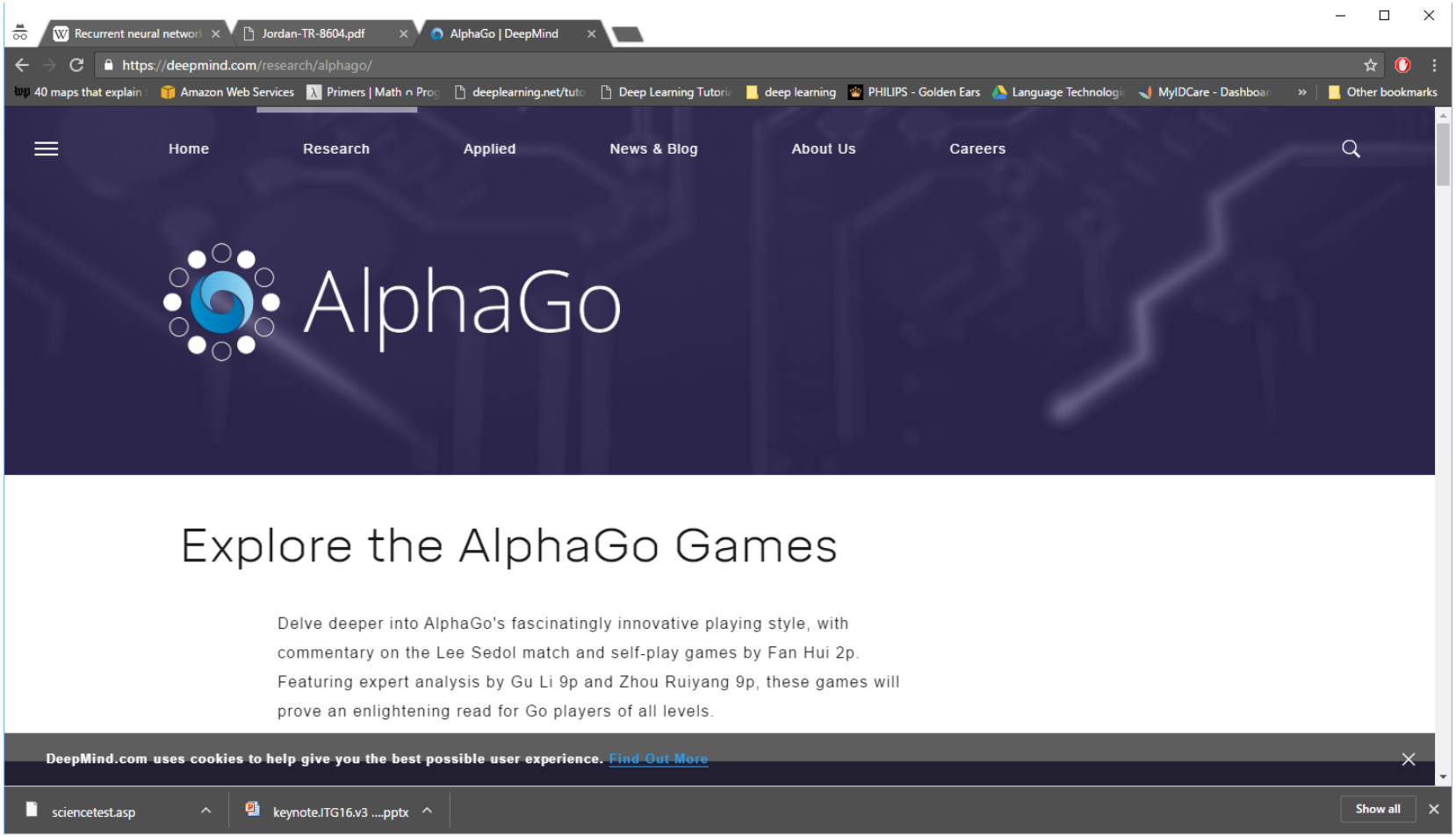
- Some recent successes with neural networks
  - A bit of hyperbole, but still..

Recent success with neural networks



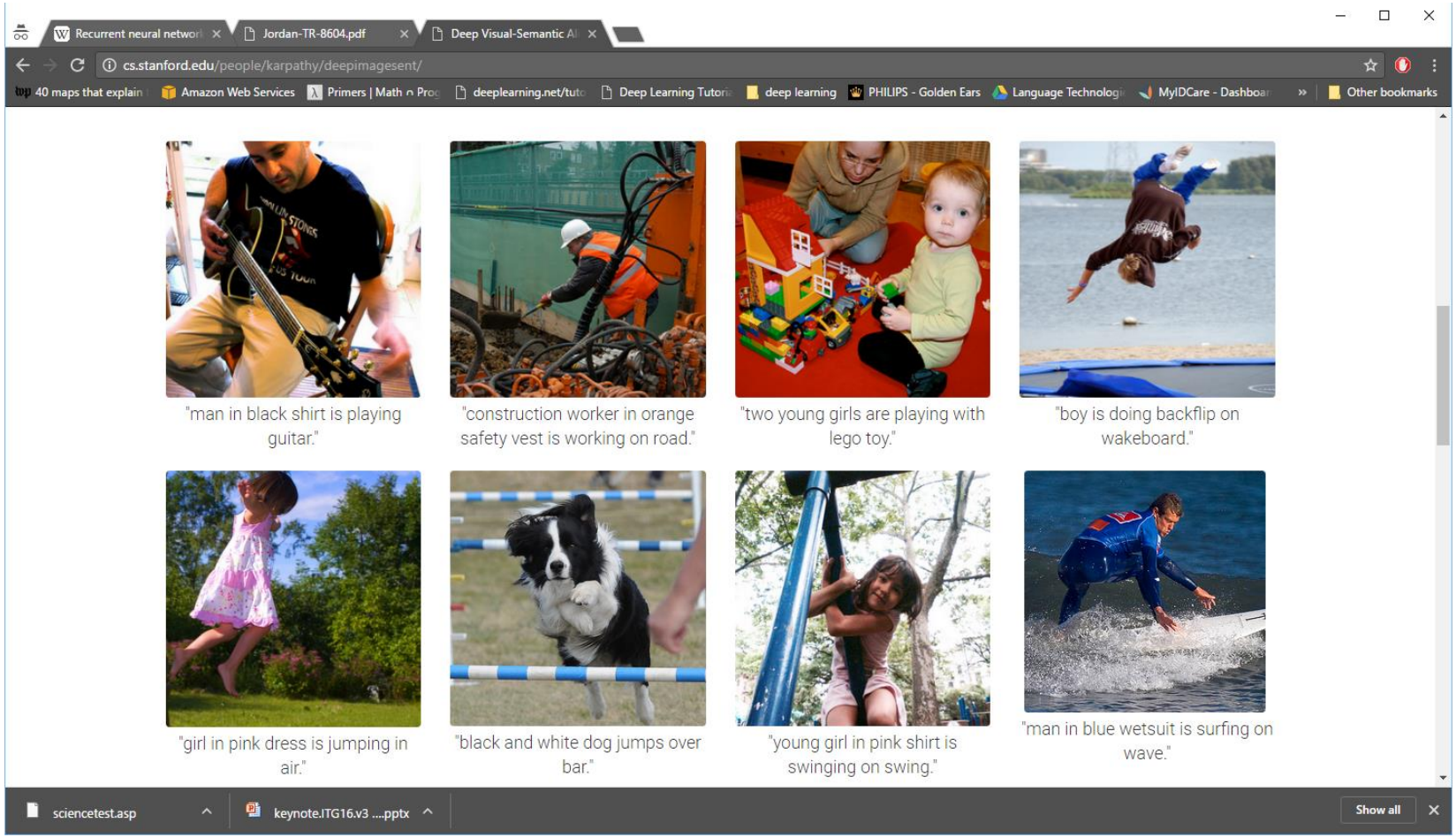
- Some recent successes with neural networks

Recent success with neural networks



- Some recent successes with neural networks

Recent success with neural networks



- Captions generated entirely by a neural network

Successes with neural networks

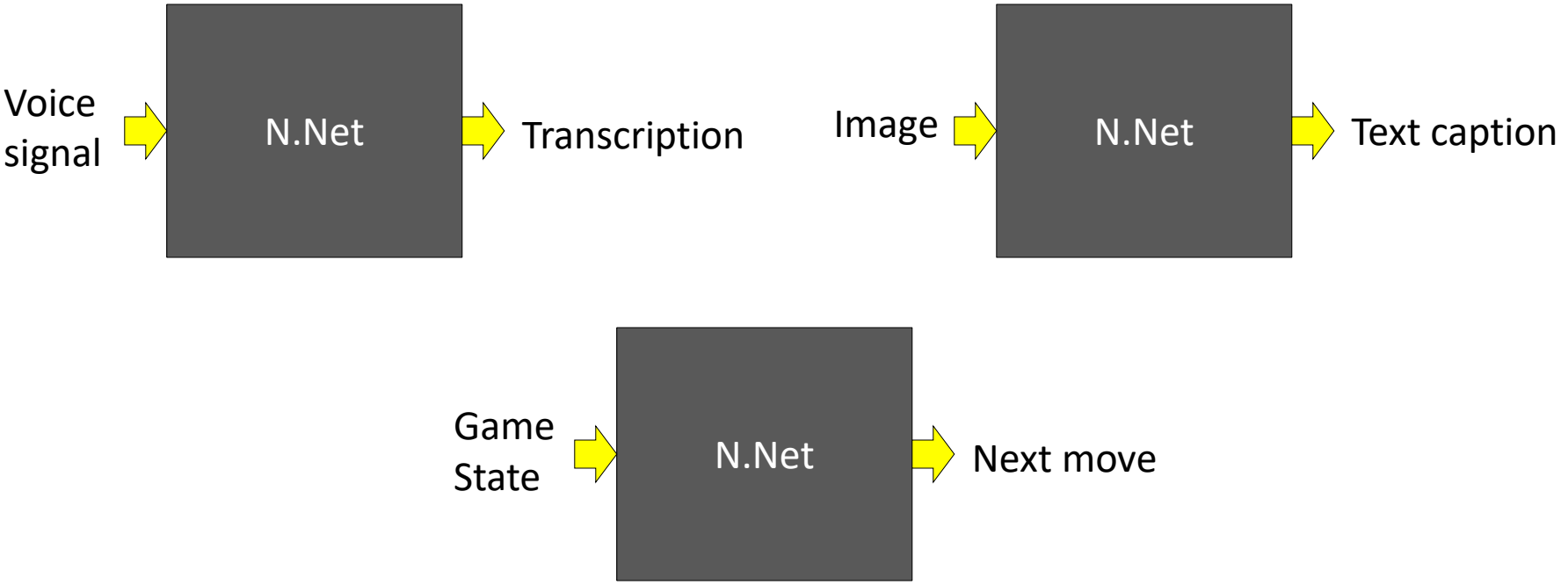
- And a variety of other problems:
  - Image recognition
  - Signal enhancement
  - Even predicting stock markets!

Neural nets and the employment market

This guy didn't know about neural networks (a.k.a deep learning)

This guy learned about neural networks (a.k.a deep learning)

So what are neural networks??

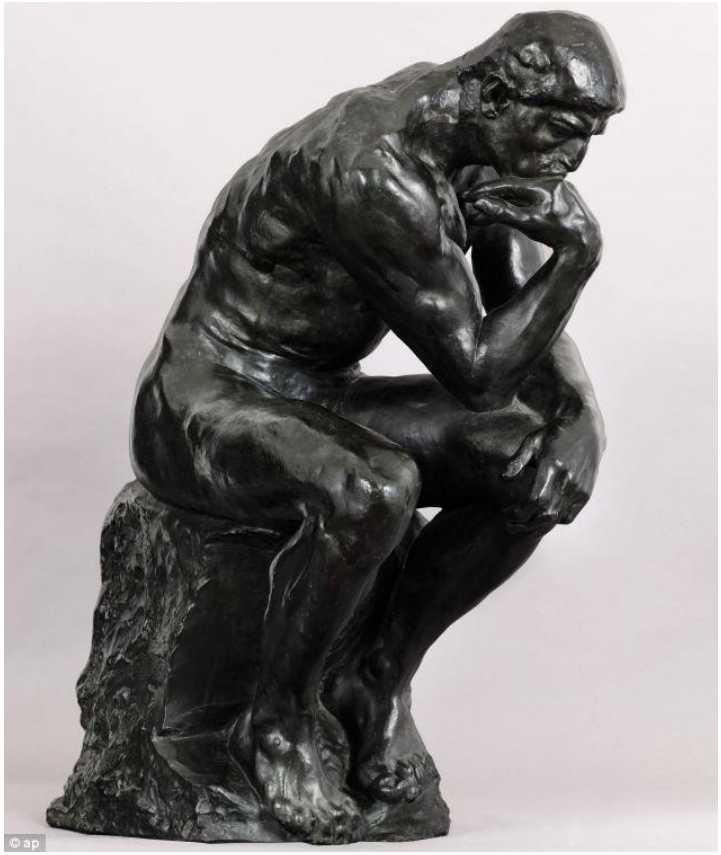


- What are these boxes?

So what are neural networks??

- It begins with this..

So what are neural networks??



"The Thinker!"
by Augustin Rodin

- Or even earlier.. with this..

The magical capacity of humans

- Humans can
  - Learn
  - Solve problems
  - Recognize patterns
  - Create
  - Cogitate
  - …

- Worthy of emulation
- But how do humans “work“?

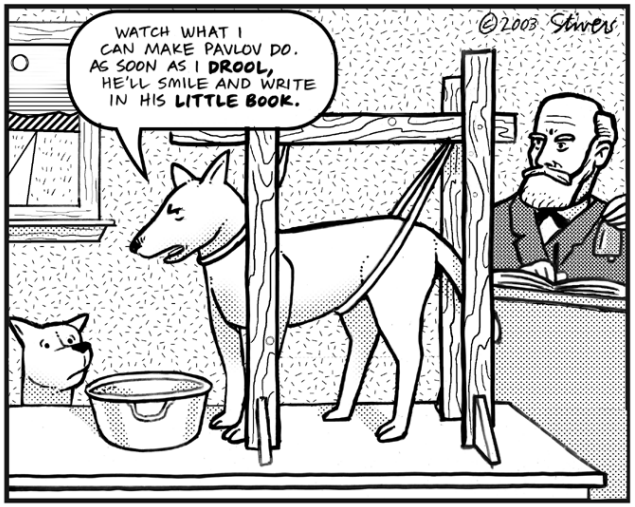
Cognition and the brain..

- “If the brain was simple enough to be understood - we would be too simple to understand it!”
  - Marvin Minsky

Early Models of Human Cognition

- Associationism
  - Humans learn through association
- 400BC-1900AD: Plato, David Hume, Ivan Pavlov..

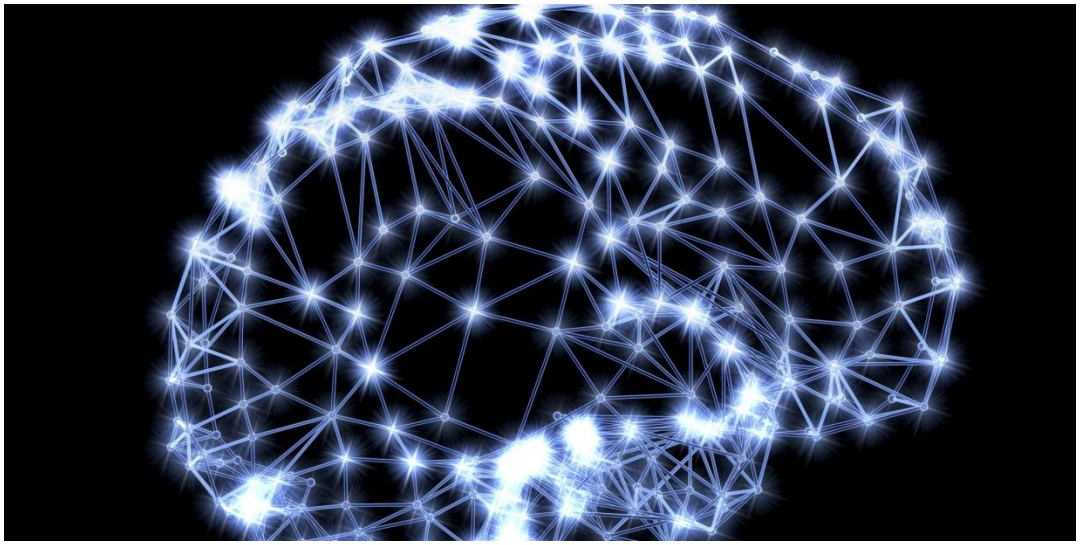
What are "Associations"



- Lightning is generally followed by thunder
  - Ergo – "hey here's a bolt of lightning, we're going to hear thunder"
  - Ergo – "We just heard thunder; did someone get hit by lightning"?

- Association!

Observation: ***The Brain***

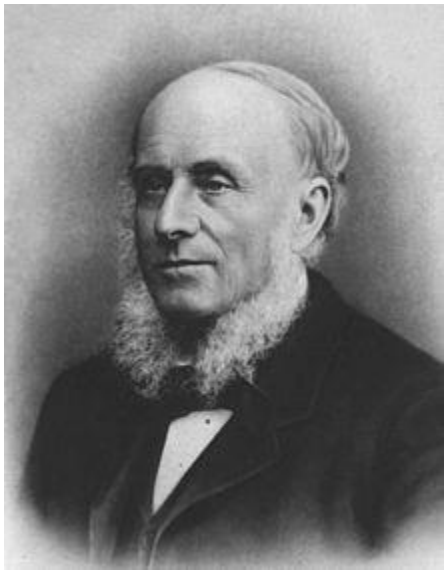


- Mid 1800s: The brain is a mass of interconnected neurons

Brain: Interconnected Neurons

- Many neurons connect *in* to each neuron
- Each neuron connects *out* to many neurons

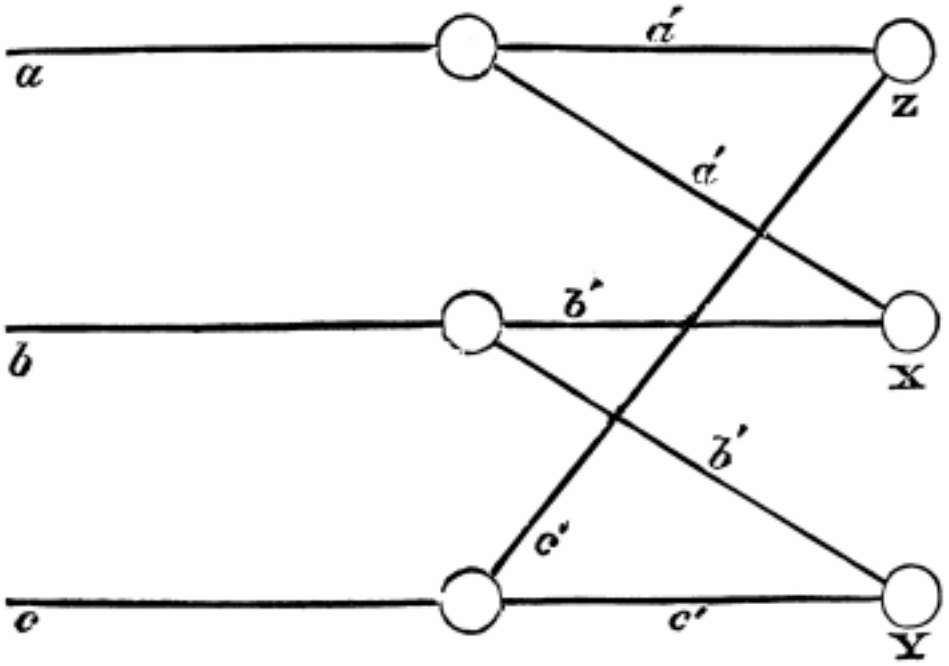
Enter ***Connectionism***



- Alexander Bain, philosopher, mathematician, logician, linguist, professor
- 1873: The information is in the *connections*
  - *The mind and body* (1873)

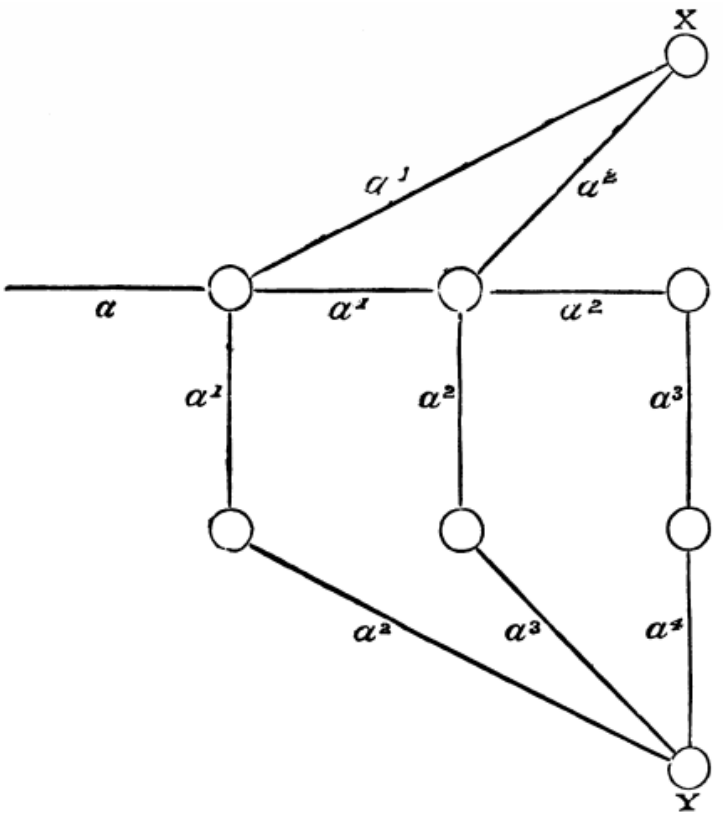
Bain's Idea : **Neural Groupings**

- Neurons excite and stimulate each other
- Different combinations of inputs can result in different outputs



Bain's Idea : **Neural Groupings**

- Different intensities of activation of A lead to the differences in when X and Y are activated



Bain's Idea 2: **Making Memories**

- "when two impressions concur, or closely succeed one another, the nerve currents find some bridge or place of continuity, better or worse, according to the abundance of nerve matter available for the transition."

- Predicts "Hebbian" learning (half a century before Hebb!)

Bain's Doubts

- "***The fundamental cause of the trouble is that in the modern world the stupid are cocksure while the intelligent are full of doubt.***"
  - Bertrand Russell

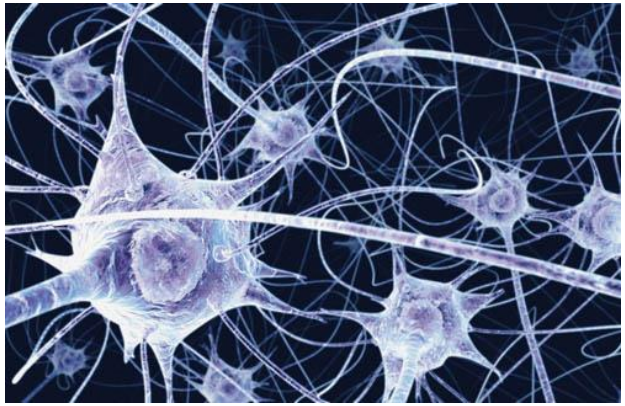
- In 1873, Bain postulated that there must be one million neurons and 5 billion connections relating to 200,000 "acquisitions"

- In 1883, Bain was concerned that he hadn't taken into account the number of "partially formed associations" and the number of neurons responsible for recall/learning

- By the end of his life (1903), recanted all his ideas!
  - Too complex; the brain would need too many neurons and connections

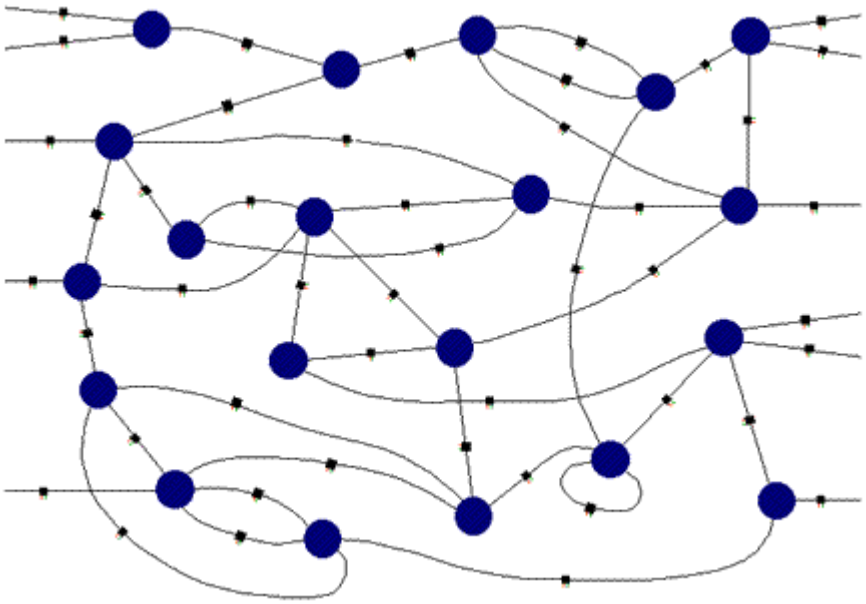
Connectionism lives on..

- The human brain is a connectionist machine
  - Bain, A. (1873). *Mind and body. The theories of their relation.* London: Henry King.
  - Ferrier, D. (1876). *The Functions of the Brain.* London: Smith, Elder and Co



- Neurons connect to other neurons. The processing/capacity of the brain is a function of these connections
- Connectionist machines emulate this structure

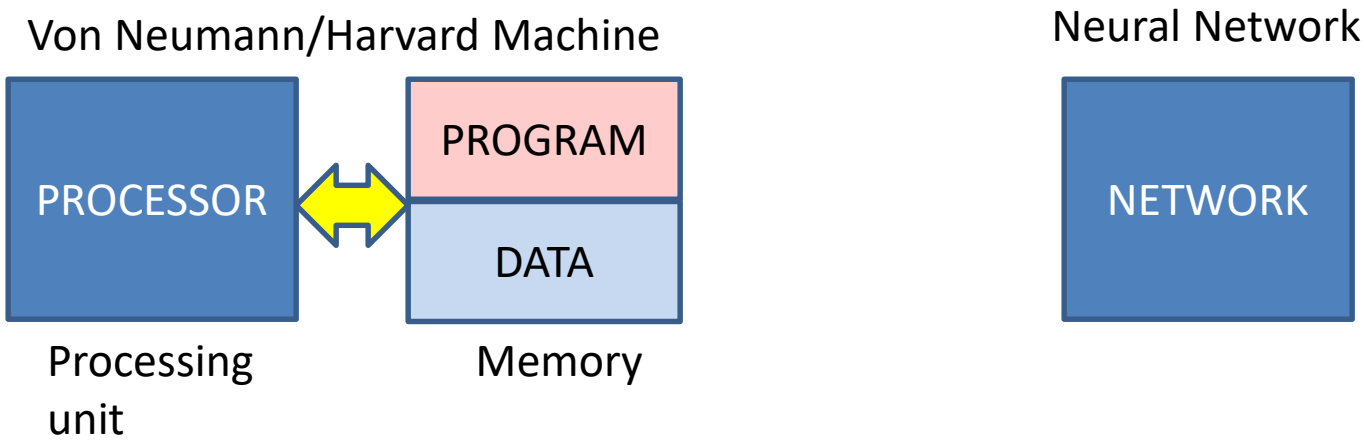
Connectionist Machines



- Network of processing elements
- **All world knowledge is stored in the *connections* between the elements**

Connectionist Machines

- Neural networks are *connectionist* machines
  - As opposed to Von Neumann Machines

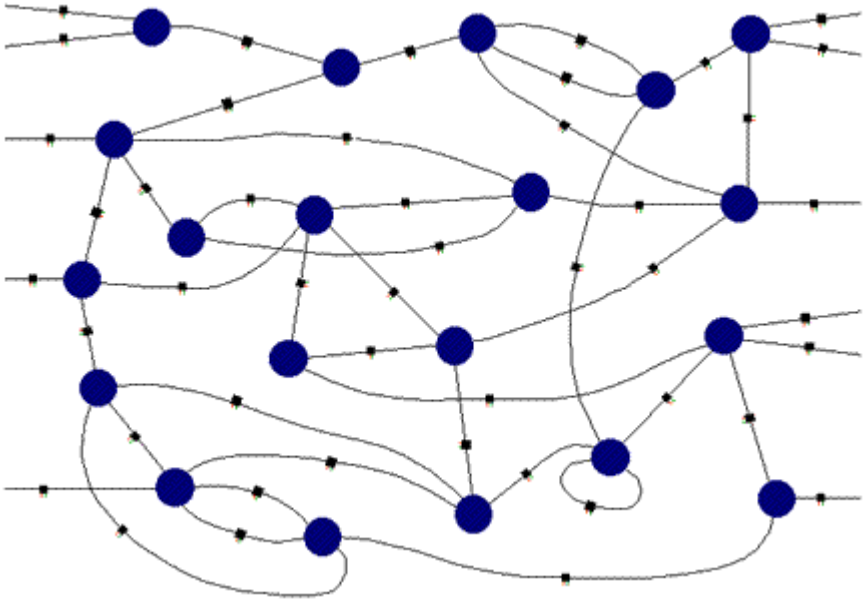


- The machine has many non-linear processing units
  - The program is the connections between these units
    - Connections may also define memory

Recap

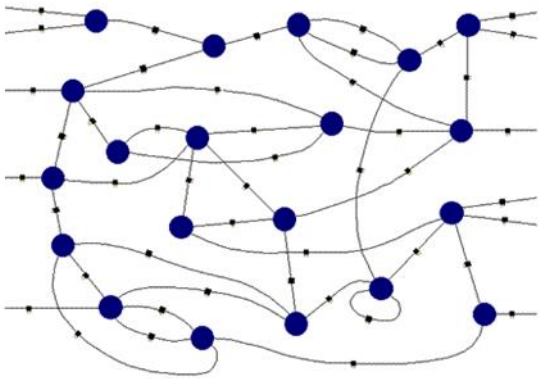
- Neural network based AI has taken over most AI tasks
- Neural networks originally began as computational models of the brain
  - Or more generally, models of cognition
- The earliest model of cognition was *associationism*
- The more recent model of the brain is *connectionist*
  - Neurons connect to neurons
  - The workings of the brain are encoded in these connections
- Current neural network models are *connectionist machines*

Connectionist Machines



- Network of processing elements
- All world knowledge is stored in the *connections* between the elements

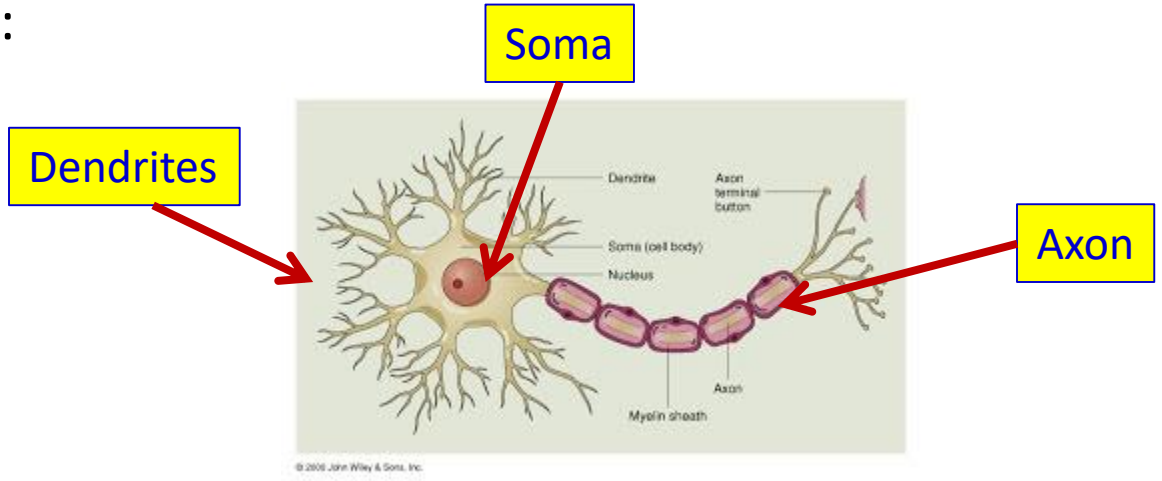
Connectionist Machines



- Connectionist machines are networks of units..
- We need a model for the *units*

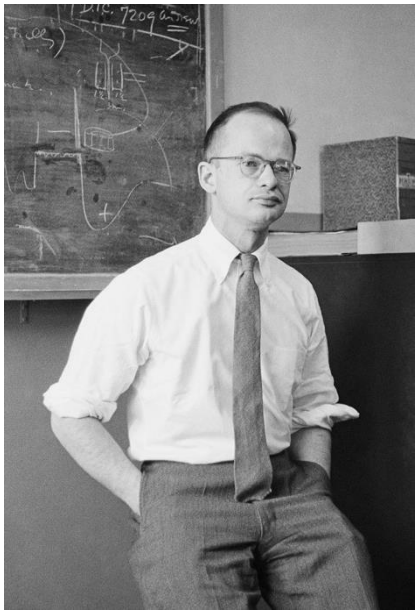
Modelling the brain

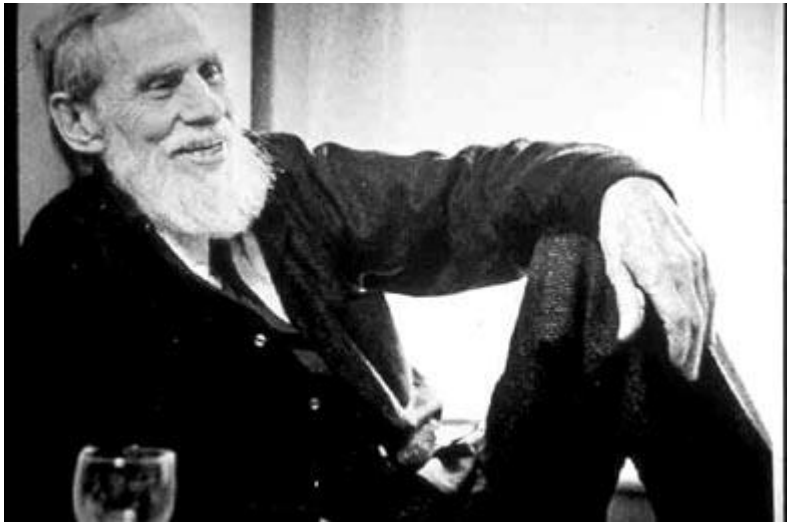
- What are the units?
- A neuron:



- Signals come in through the dendrites into the Soma
- A signal goes out via the axon to other neurons
  - Only one axon per neuron
- Factoid that may only interest me: Neurons do not undergo cell division

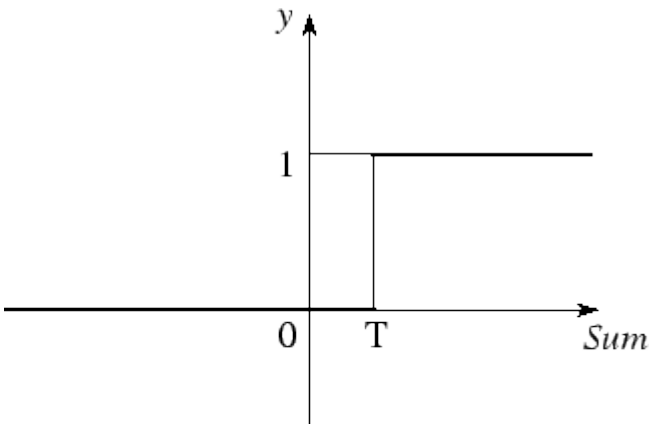
McCullough and Pitts

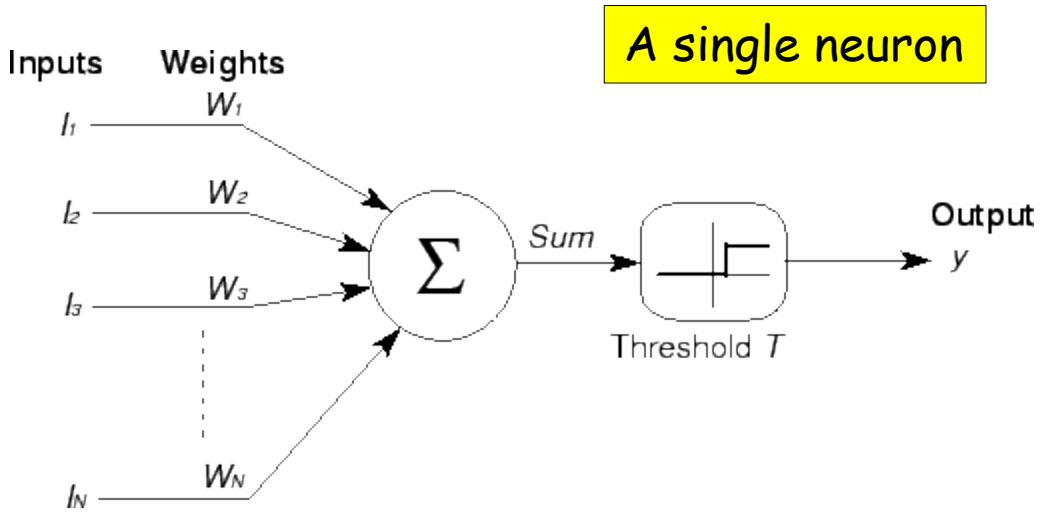




- The Doctor and the Hobo..
  - Warren McCulloch: Neurophysician
  - Walter Pitts: Homeless wannabe logician who arrived at his door

The McCulloch and Pitts model





- A mathematical model of a neuron
  - McCulloch, W.S. & Pitts, W.H. (1943). A Logical Calculus of the Ideas Immanent in Nervous Activity, Bulletin of Mathematical Biophysics, 5:115-137, 1943
    - Pitts was only 20 years old at this time
  - Threshold Logic

Synaptic Model

- *Excitatory synapse:* Transmits weighted input to the neuron

- *Inhibitory synapse:* Any signal from an inhibitory synapse forces output to zero
  - The activity of any inhibitory synapse absolutely prevents excitation of the neuron at that time.
    - Regardless of other inputs

Simple "networks" of neurons can perform Boolean operations

Boolean Gates

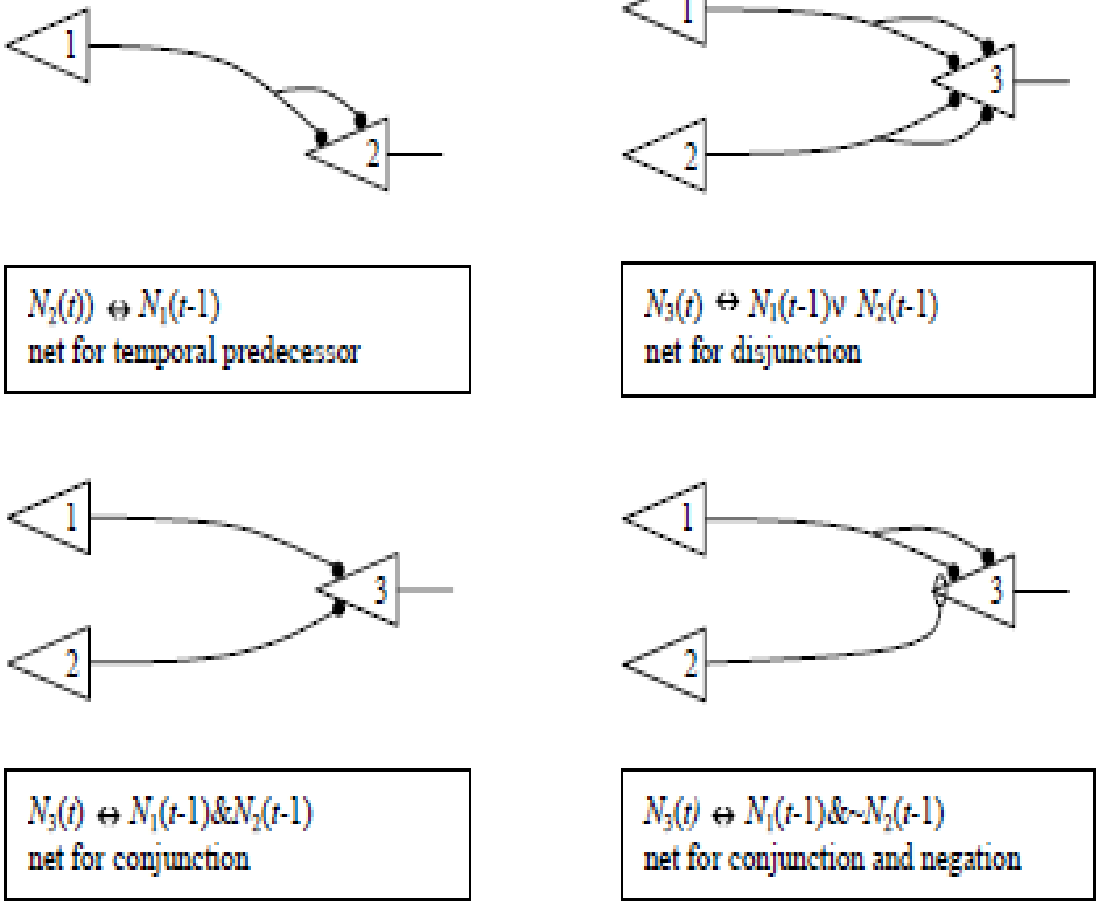


Figure 1. Diagrams of McCulloch and Pitts nets. In order to send an output pulse, each neuron must receive two excitory inputs and no inhibitory inputs. Lines ending in a dot represent excitatory connections; lines ending in a hoop represent inhibitory connections.

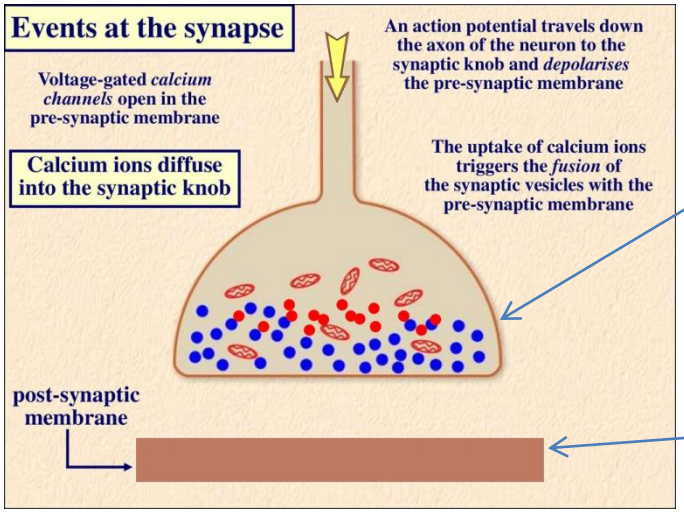
Criticisms

- Several..
  - Claimed their machine could emulate a turing machine
- Didn’t provide a learning mechanism..

Donald Hebb

- “Organization of behavior”, 1949
- A learning mechanism:
  - Neurons that fire together wire together

Hebbian Learning



Axonal connection from neuron X

Dendrite of neuron Y

- If neuron x_i repeatedly triggers neuron y, the synaptic knob connecting x_i to y gets larger
- In a mathematical model:

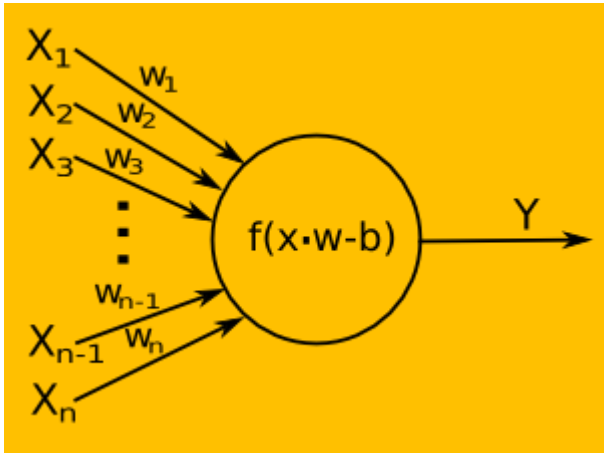
$$w_i = w_i + \eta x_i y$$

  - Weight of i^{th} neuron's input to output neuron y
- This simple formula is actually the basis of many learning algorithms in ML

A better model

- Frank Rosenblatt
  - Psychologist, Logician
  - Inventor of the solution to everything, aka the Perceptron (1958)

Simplified mathematical model

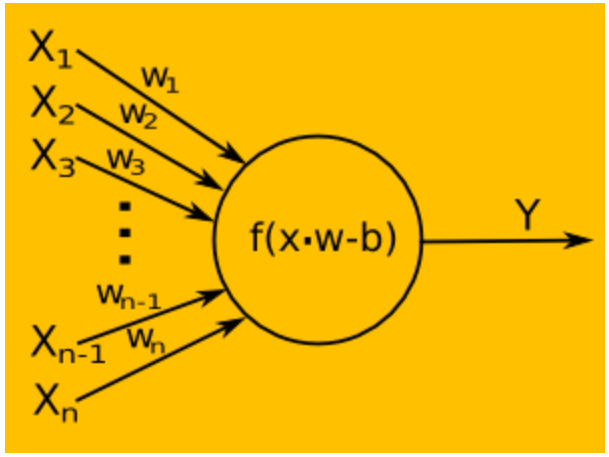


- Number of inputs combine linearly
  - Threshold logic: Fire if combined input exceeds threshold

$$Y = \begin{cases} 1 \;\; if \sum_i w_i x_i + b > 0 \\ 0 \qquad\qquad else \end{cases}$$

His "Simple" Perceptron

- Originally assumed could represent *any* Boolean circuit and perform any logic
  - "*the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence*," New York Times (8 July) 1958
  - "*Frankenstein Monster Designed by Navy That Thinks,*" Tulsa, Oklahoma Times 1958



Also provided a learning algorithm

$$\mathbf{w} = \mathbf{w} + \eta(d(\mathbf{x}) - y(\mathbf{x}))\mathbf{x}$$

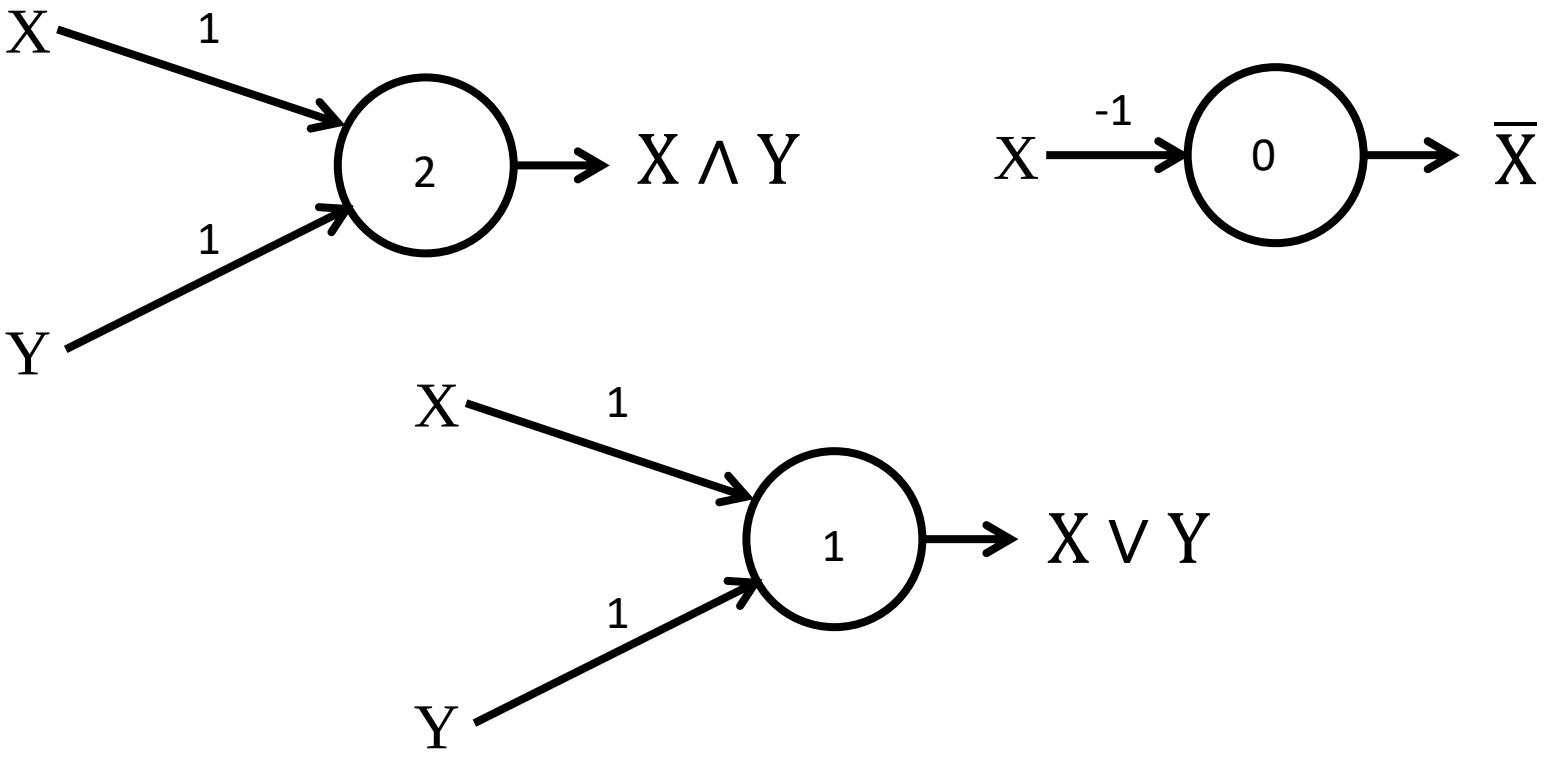
Sequential Learning:

$d(x)$ is the desired output in response to input x

$y(x)$ is the actual output in response to x

- Boolean tasks
- Update the weights whenever the perceptron output is wrong
- Proved convergence

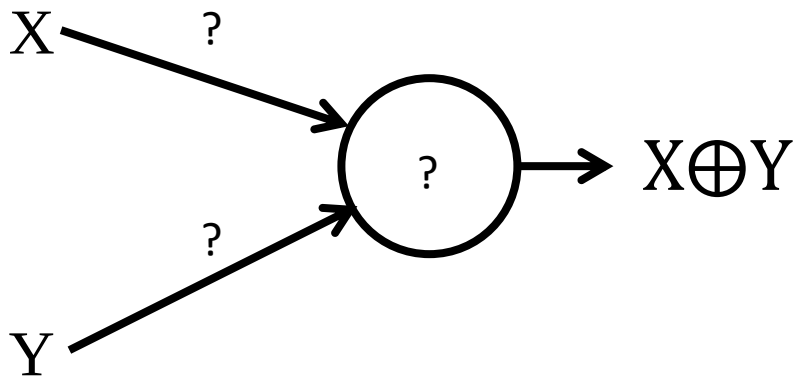
Perceptron



- Easily shown to mimic any Boolean gate
- But…

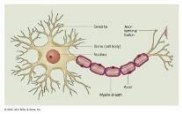
Perceptron

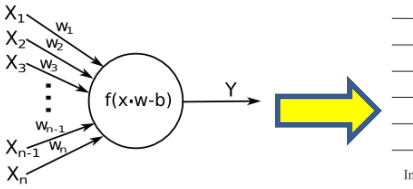
No solution for XOR!
Not universal!

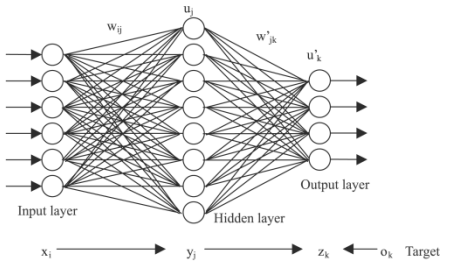


- Minsky and Papert, 1968

A single neuron is not enough

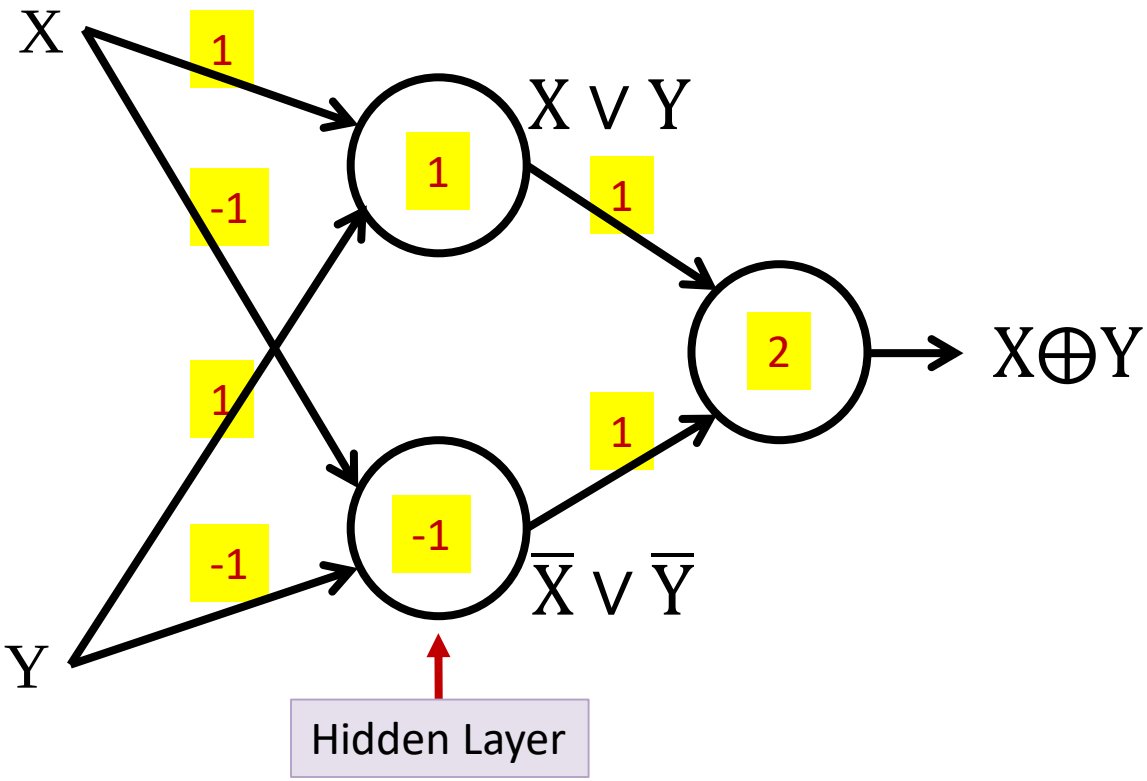






- Individual elements are weak computational elements
  - Marvin Minsky and Seymour Papert, 1969, *Perceptrons: An Introduction to Computational Geometry*
- *Networked* elements are required

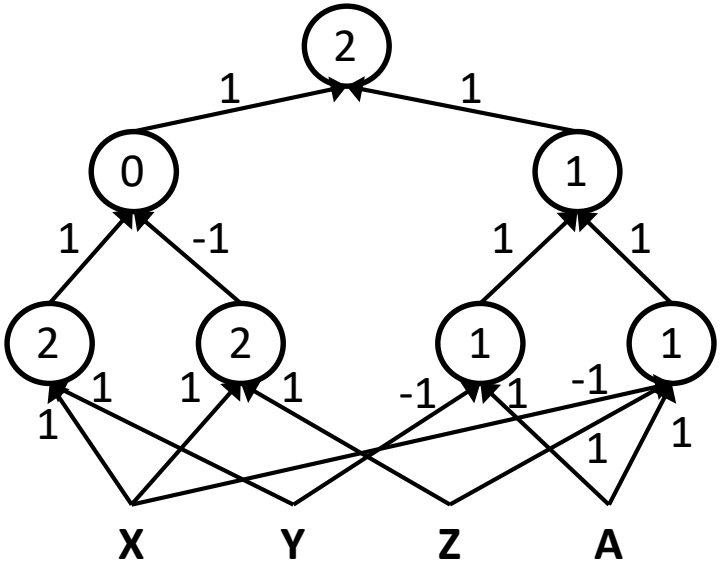
Multi-layer Perceptron!



- XOR
  - The first layer is a "hidden" layer
  - Also originally suggested by Minsky and Paper 1968

A more generic model

$$((A\&\bar{X}\&Z)|(A\&\bar{Y}))\&((X\ \&\ Y)|\overline{(X\&Z)})$$

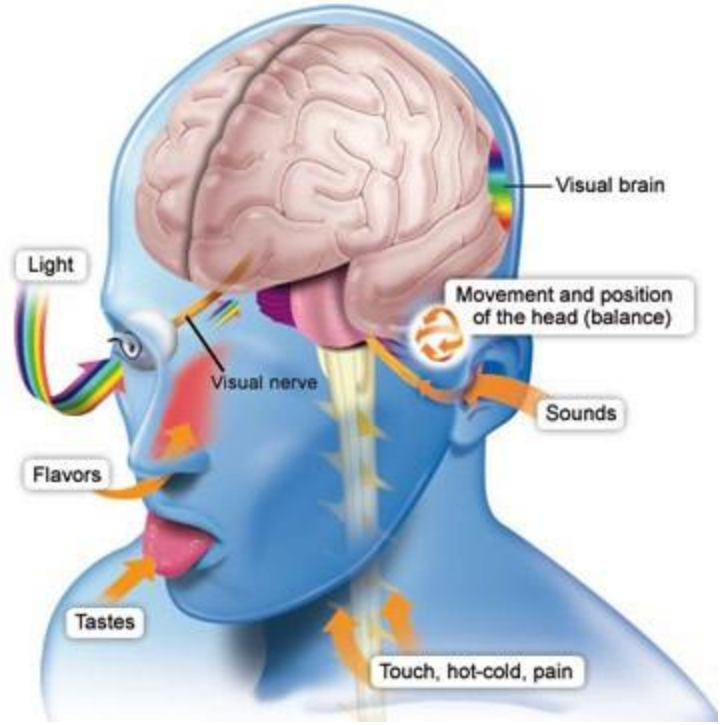


- A “multi-layer” perceptron
- Can compose arbitrarily complicated Boolean functions!
  - More on this in the next part

Story so far

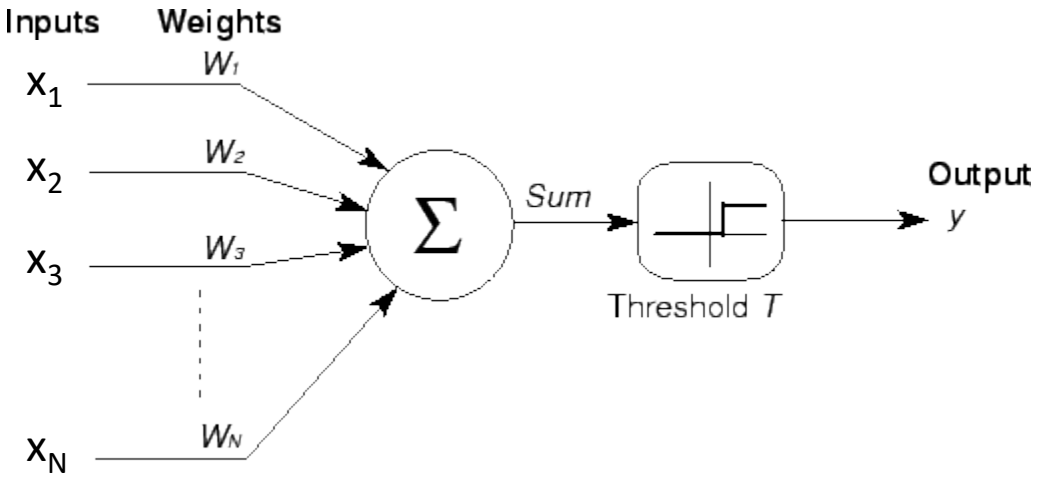
- Neural networks began as computational models of the brain
- Neural network models are *connectionist machines*
  - The comprise networks of neural units
- McCullough and Pitt model: Neurons as Boolean threshold units
  - Models the brain as performing propositional logic
  - But no learning rule
- Hebb's learning rule: Neurons that fire together wire together
  - Unstable
- Rosenblatt's perceptron : A variant of the McCulloch and Pitt neuron with a provably convergent learning rule
  - But individual perceptrons are limited in their capacity (Minsky and Papert)
- Multi-layer perceptrons can model arbitrarily complex Boolean functions

But our brain is not Boolean



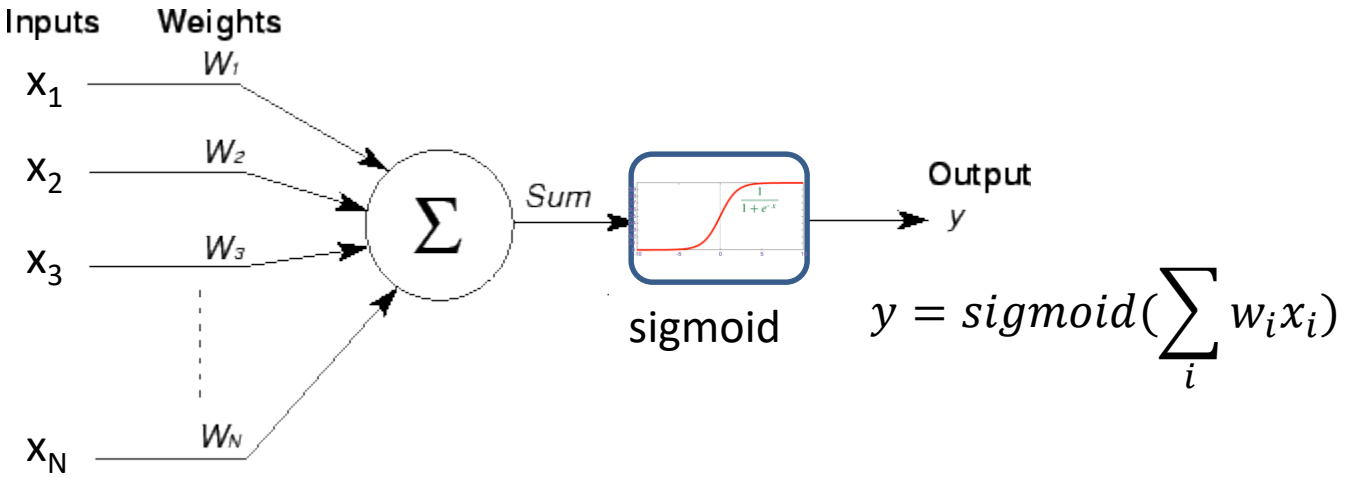
- We have real inputs
- We make non-Boolean inferences/predictions

The perceptron with *real* inputs



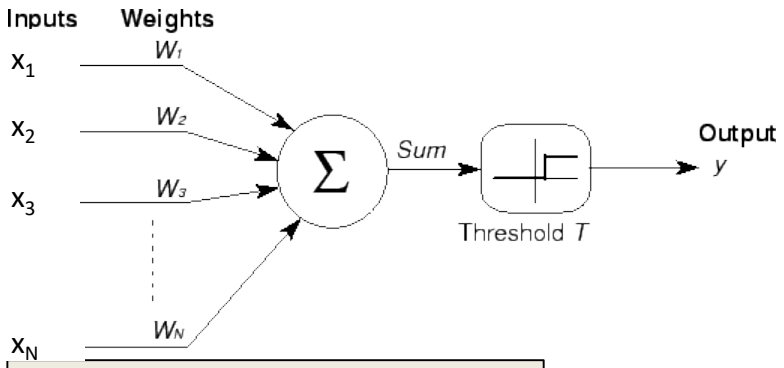
- $x_1 \ldots x_N$ are real valued
- $W_1 \ldots W_N$ are real valued
- Unit "fires" if weighted input exceeds a threshold

The perceptron with *real* inputs and a real *output*

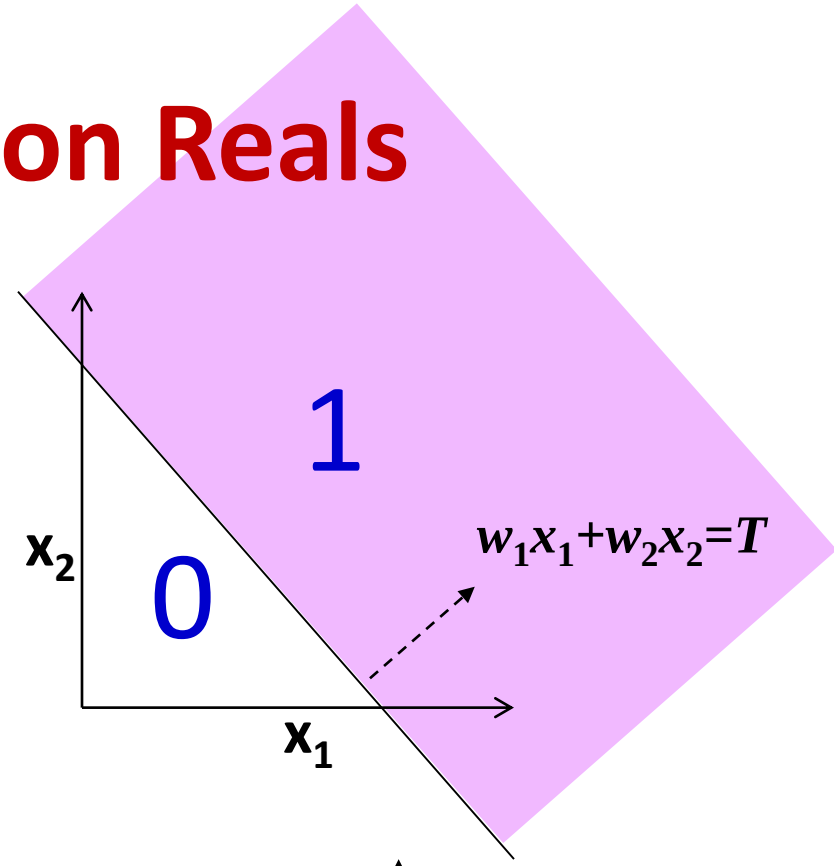


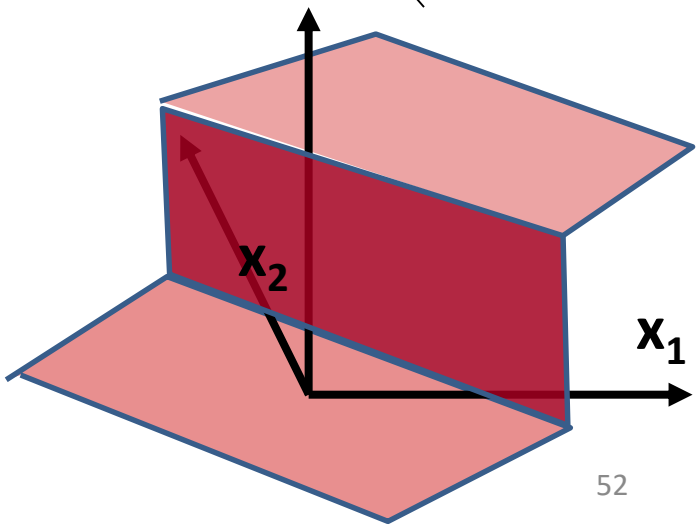
- $x_1 \ldots x_N$ are real valued
- $W_1 \ldots W_N$ are real valued
- The output y can also be real valued
  - Sometimes viewed as the "probability" of firing
  - *Is useful to continue assuming Boolean outputs though*

A Perceptron on Reals



$$y = \begin{cases} 1 \text{ if } \sum_i w_i x_i \geq T \\ 0 \text{ else} \end{cases}$$

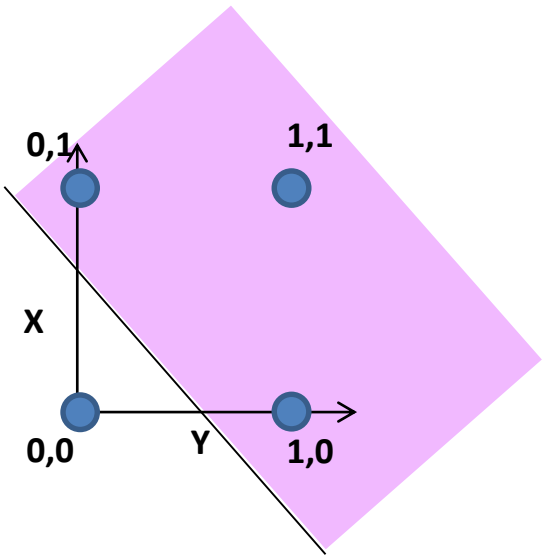


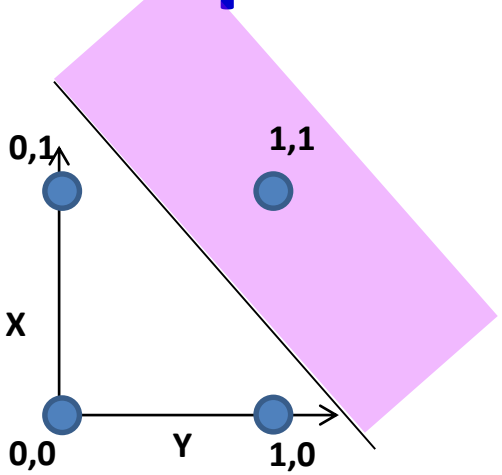


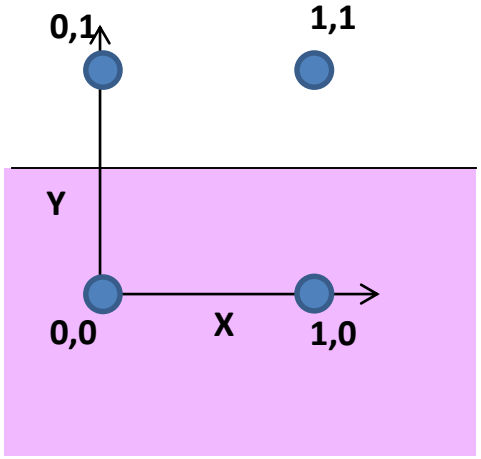
- A perceptron operates on *real*-valued vectors

– This is a *linear classifier*

Boolean functions with a real perceptron

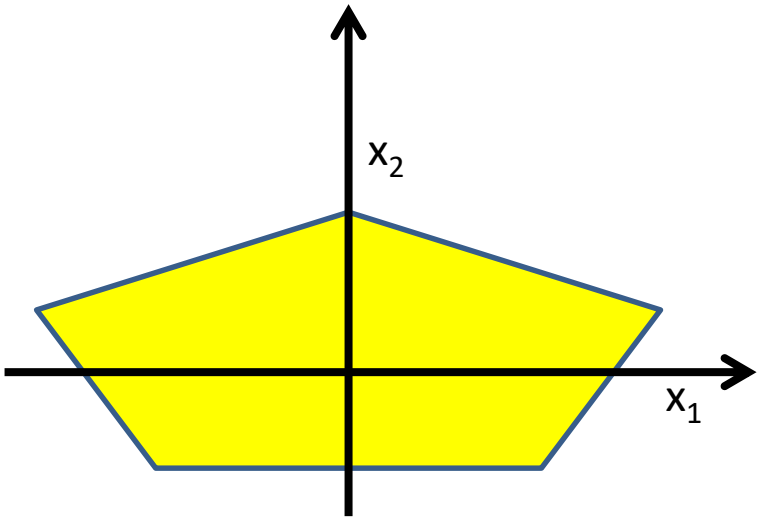






- Boolean perceptrons are also linear classifiers
  - Purple regions have output 1 in the figures
  - What are these functions
  - Why can we not compose an XOR?

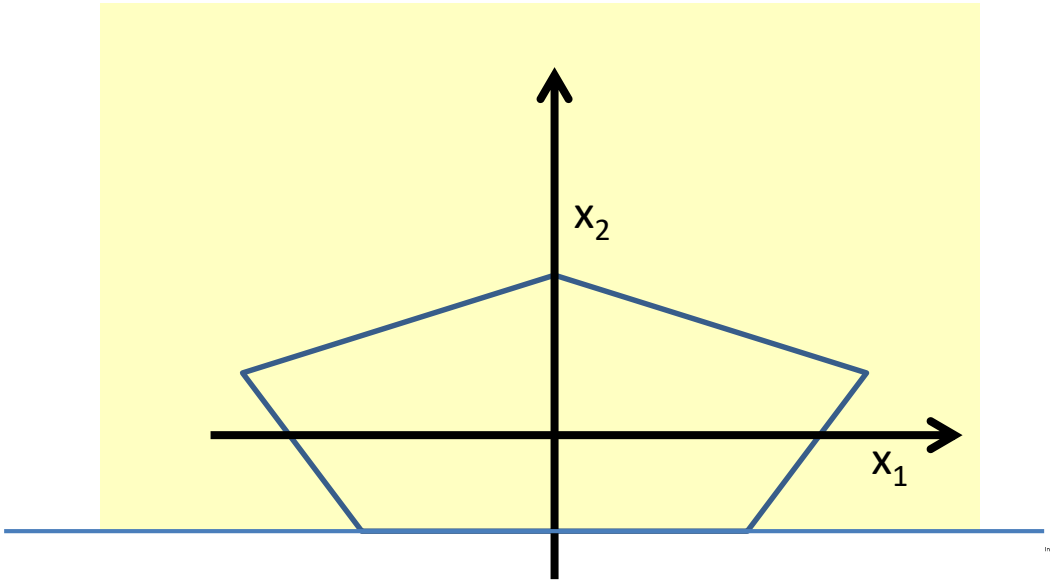
Composing complicated "decision" boundaries

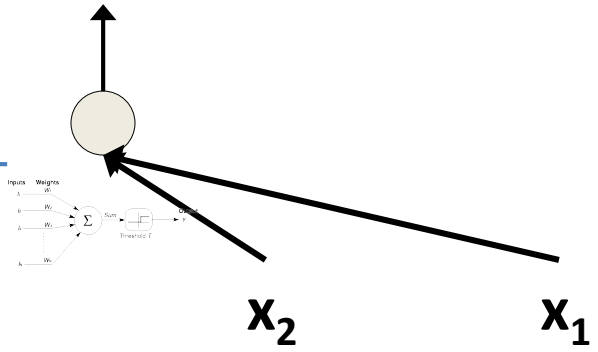


Can now be composed into "networks" to compute arbitrary classification "boundaries"

- Build a network of units with a single output that fires if the input is in the coloured area

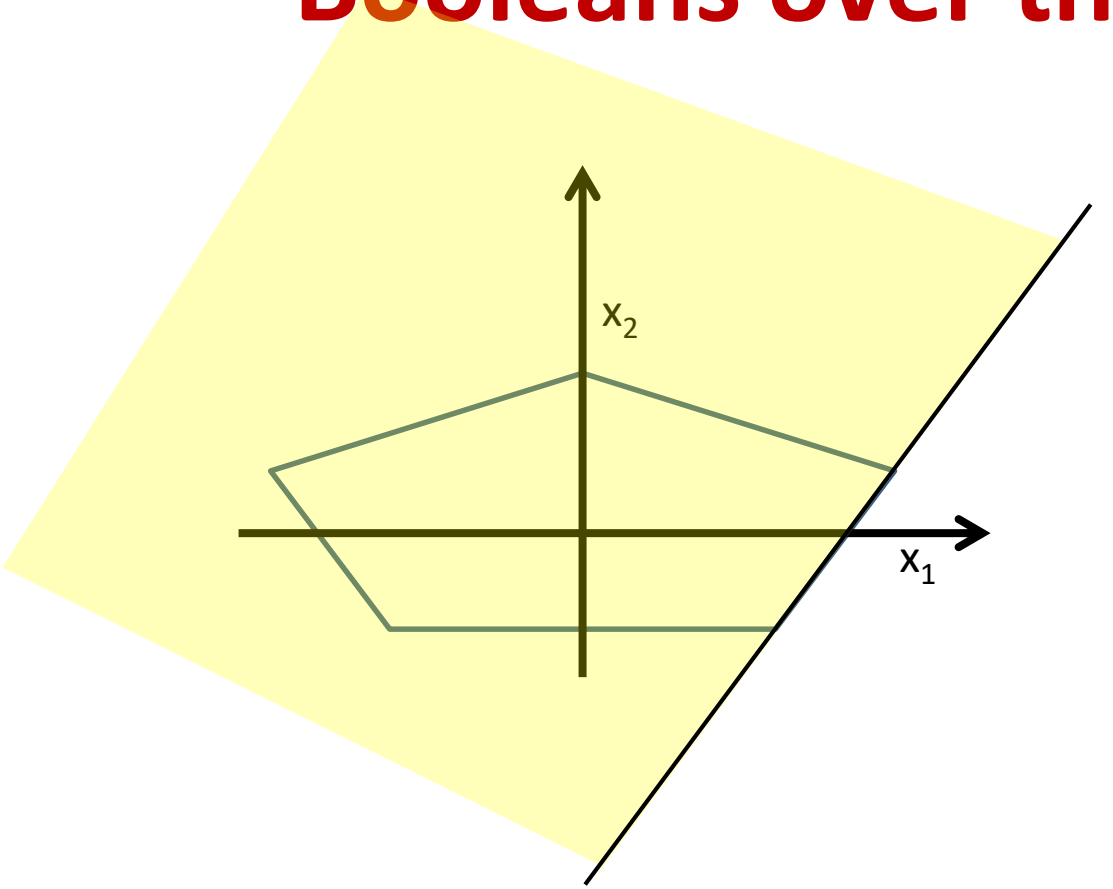
Booleans over the reals

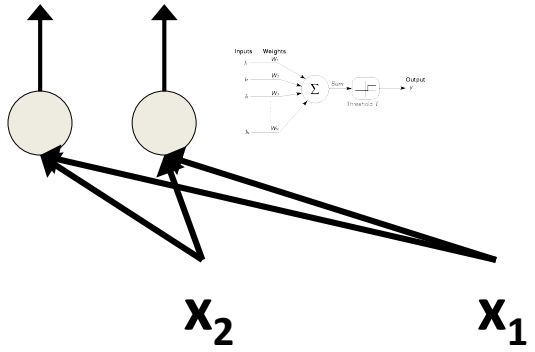




- The network must fire if the input is in the coloured area

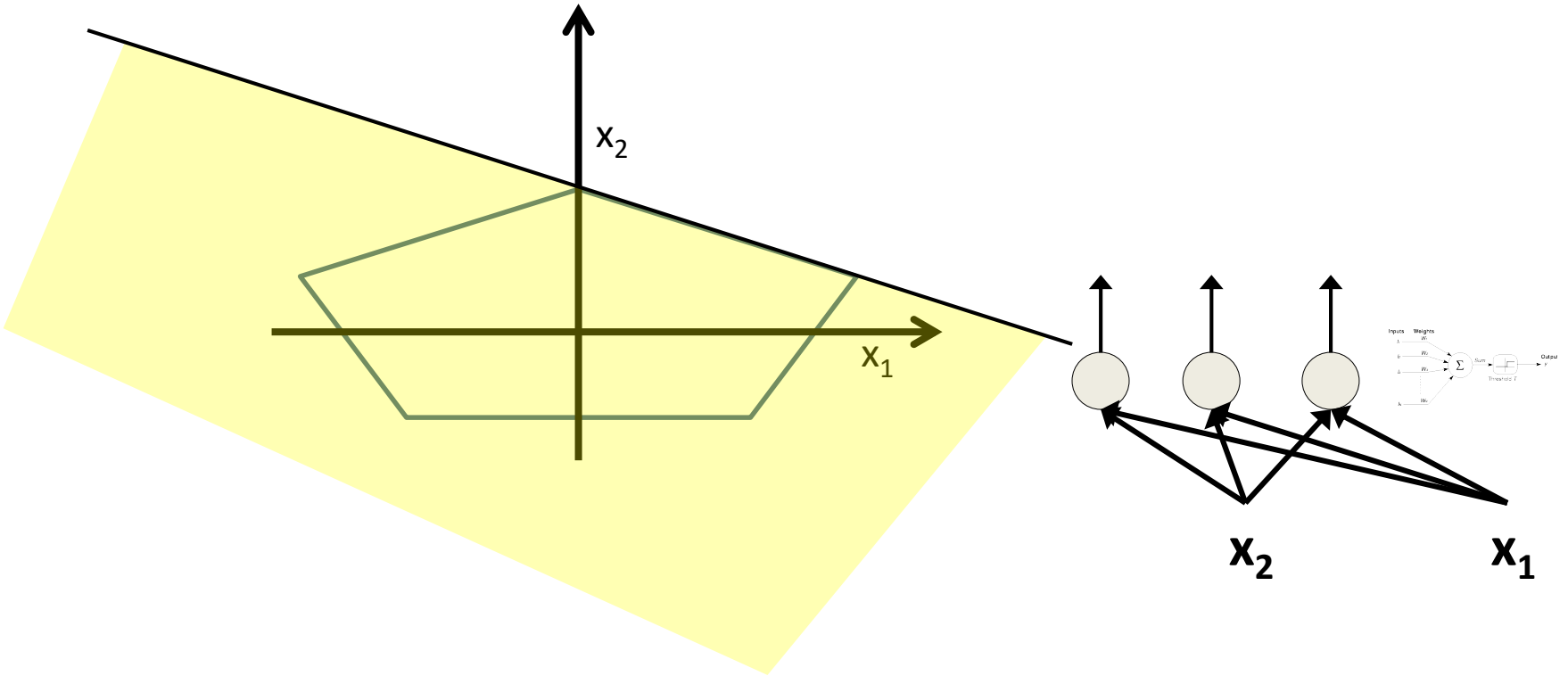
Booleans over the reals





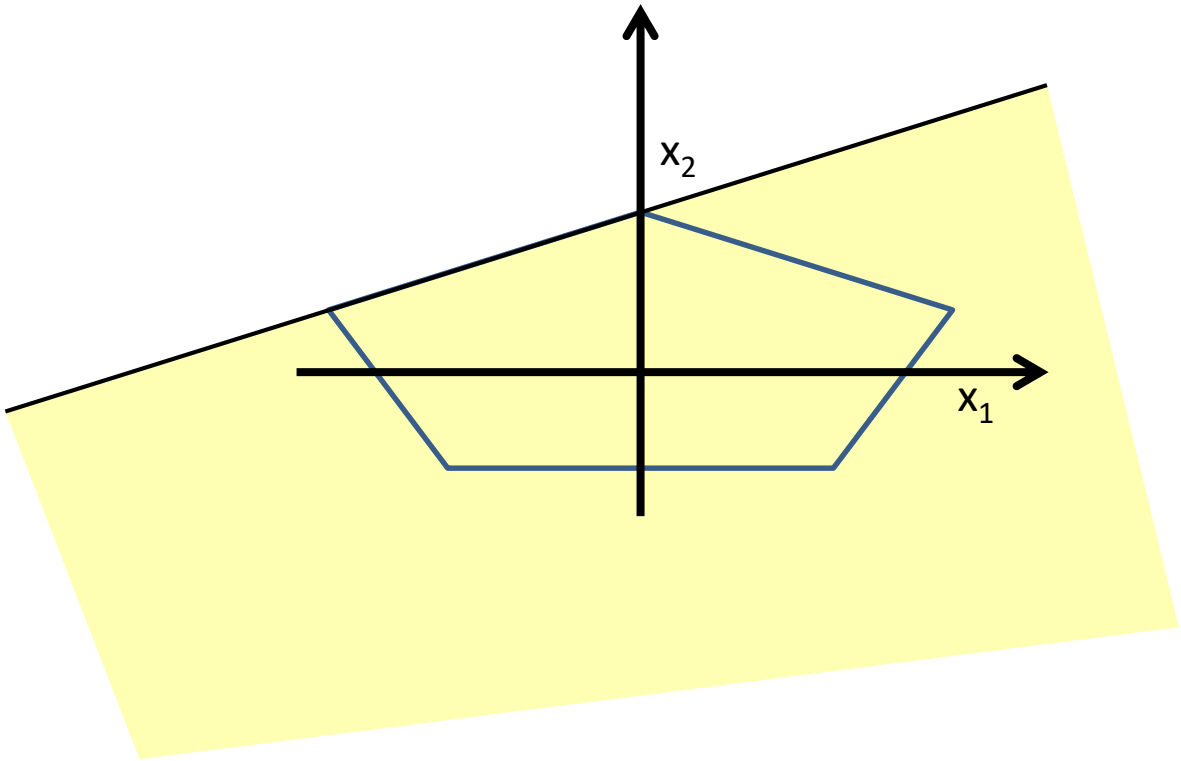
- The network must fire if the input is in the coloured area

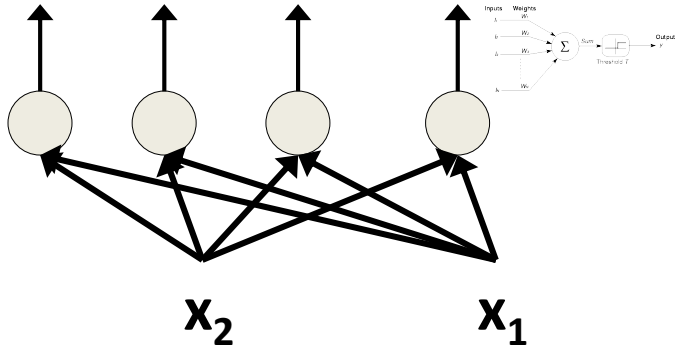
Booleans over the reals



- The network must fire if the input is in the coloured area

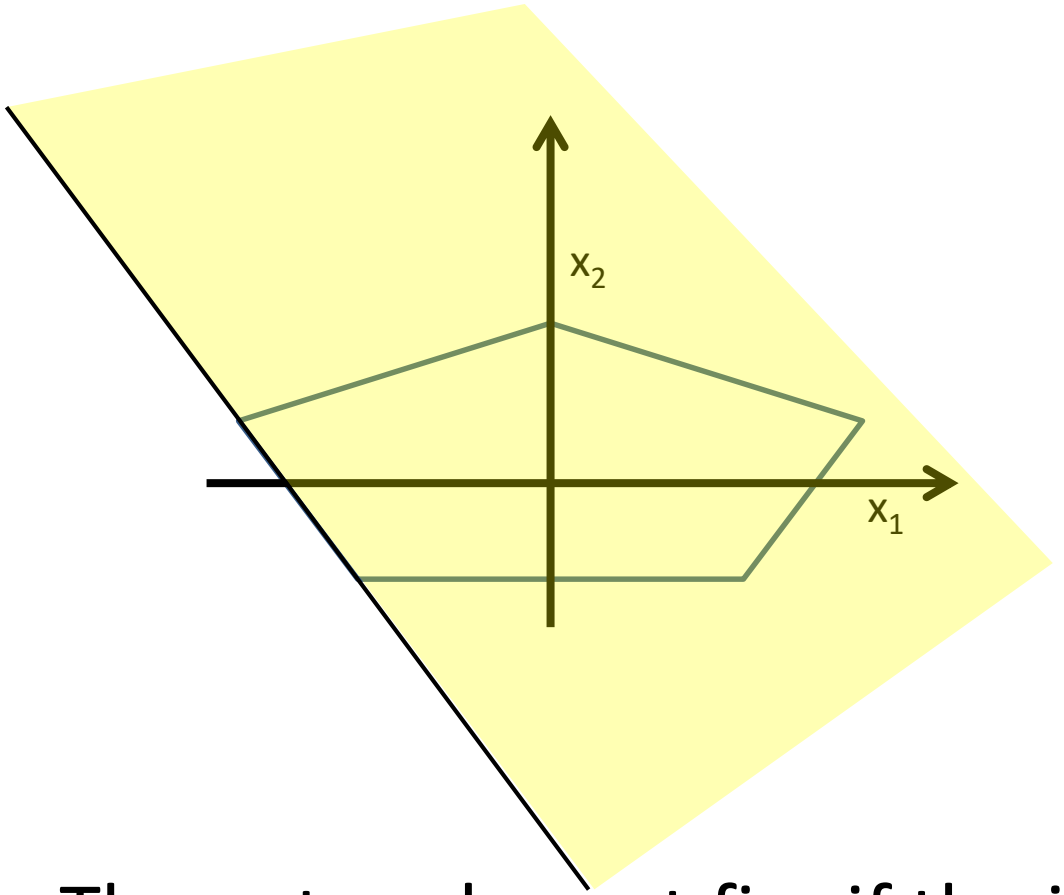
Booleans over the reals

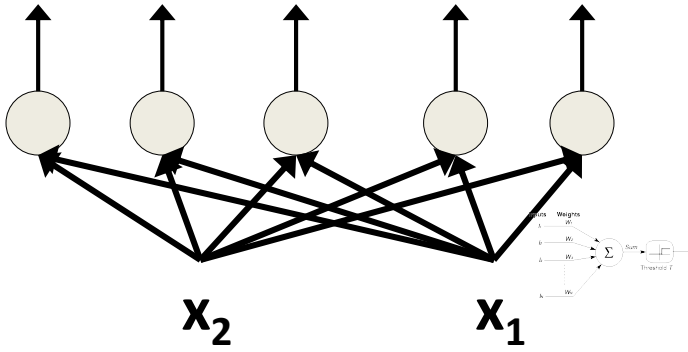




- The network must fire if the input is in the coloured area

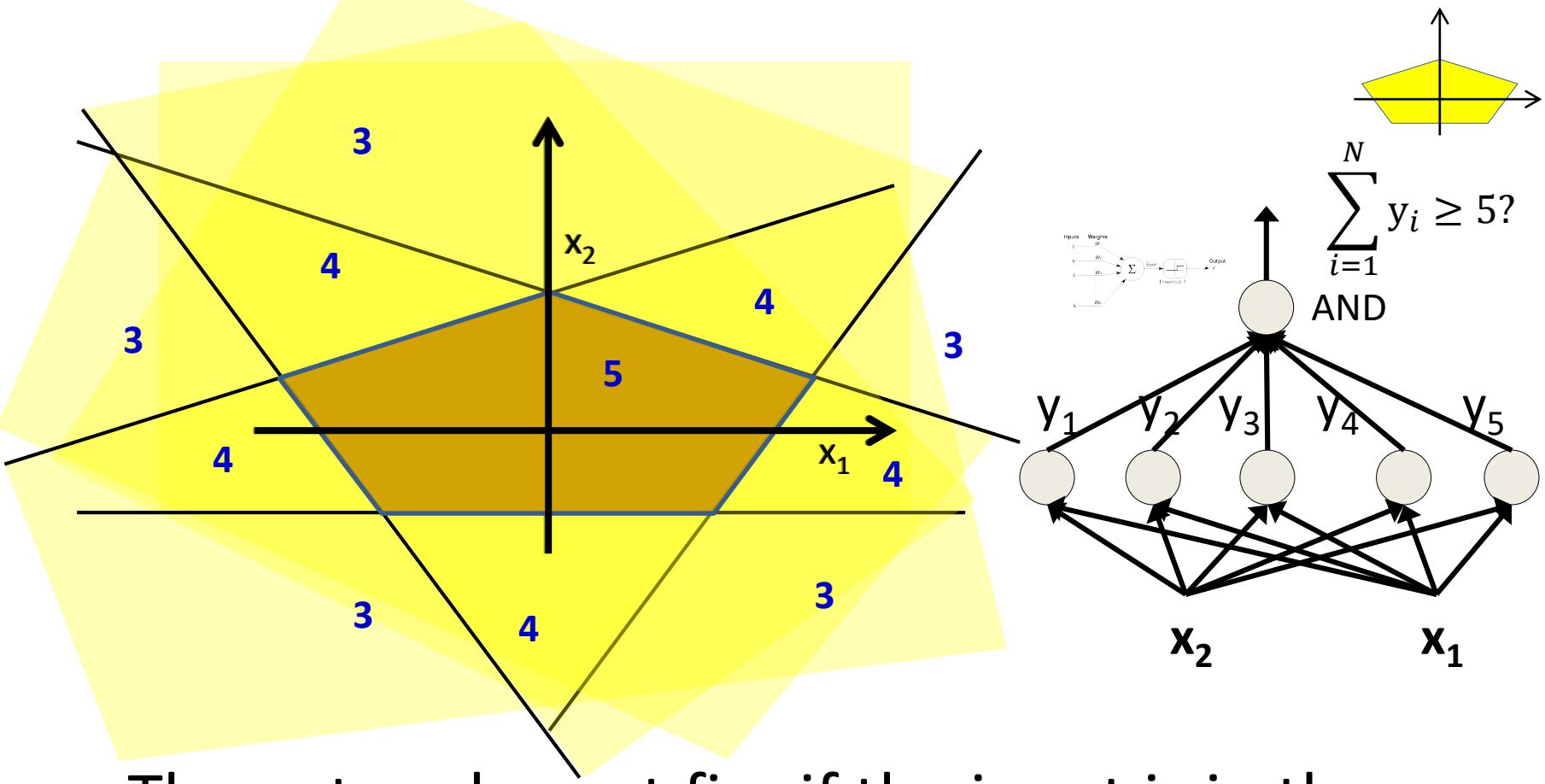
Booleans over the reals





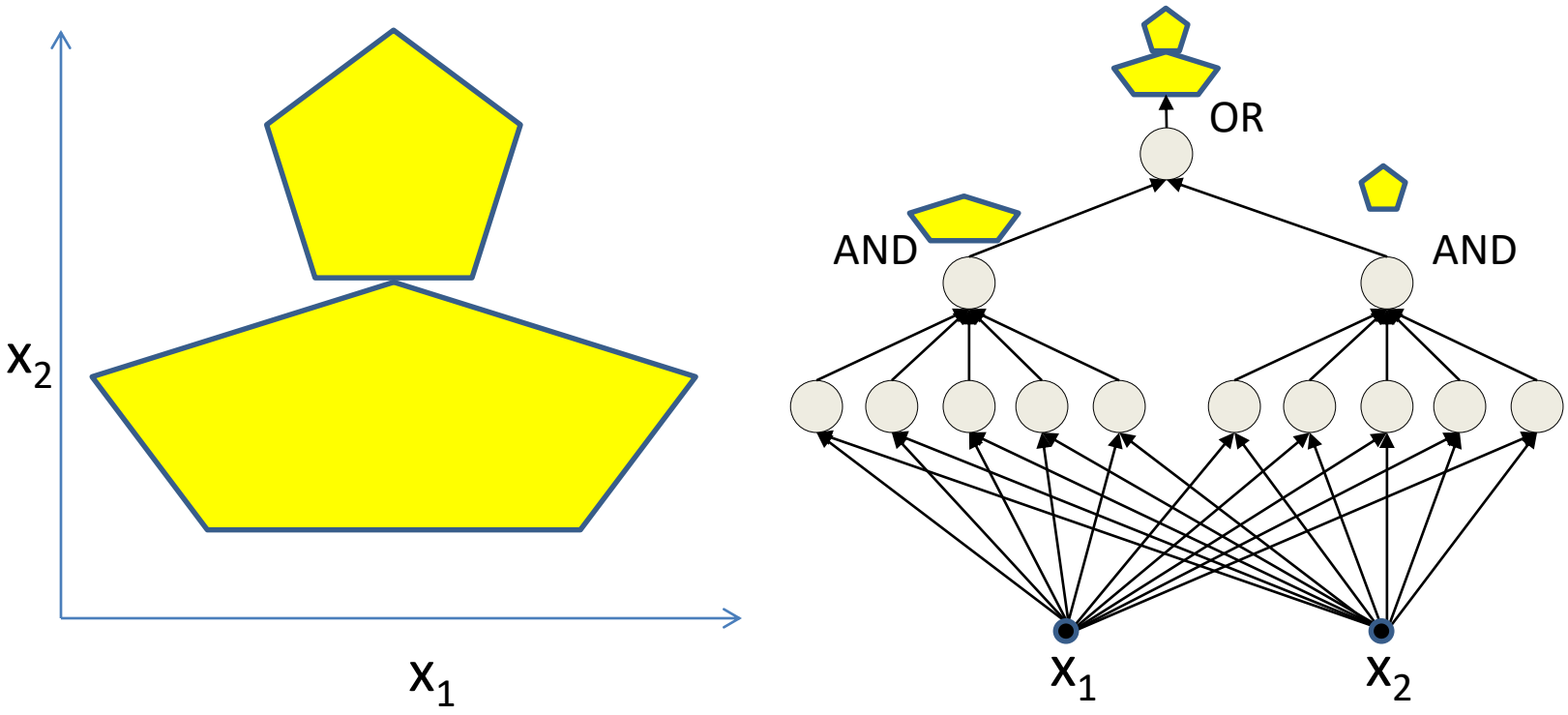
- The network must fire if the input is in the coloured area

Booleans over the reals



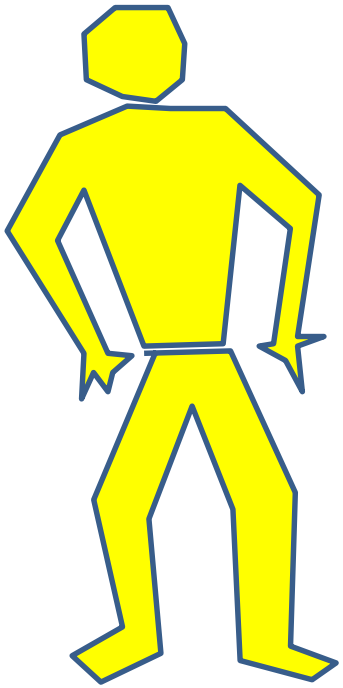
- The network must fire if the input is in the coloured area

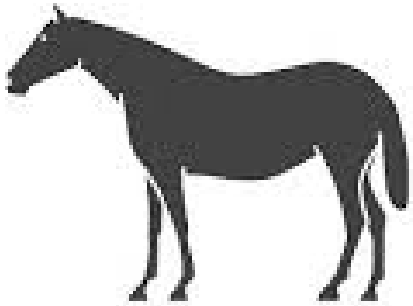
More complex decision boundaries



- Network to fire if the input is in the yellow area
  - "OR" two polygons
  - A third layer is required

Complex decision boundaries

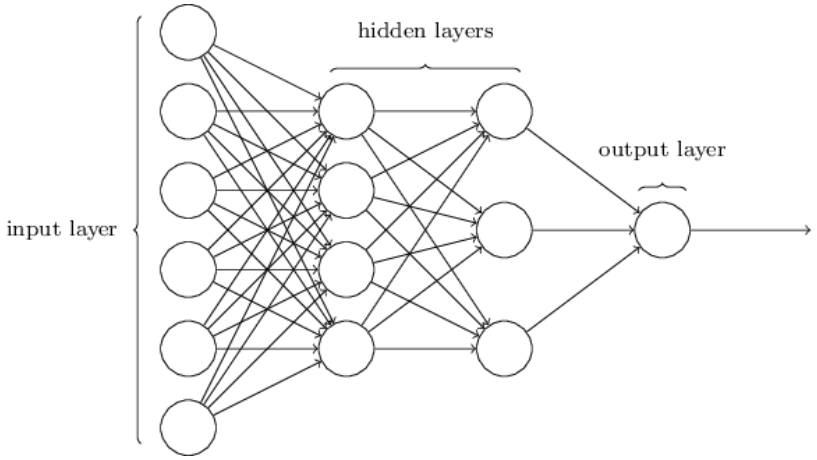


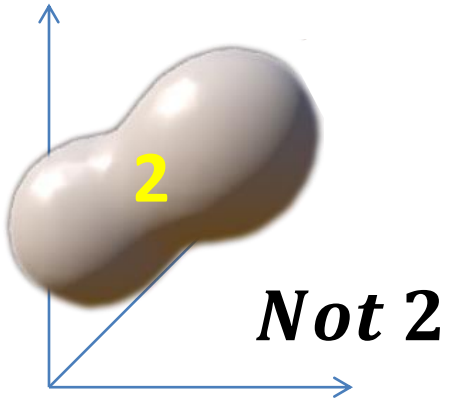


- Can compose very complex decision boundaries
  - How complex exactly? More on this in the next class

Complex decision boundaries

784 dimensions
(MNIST)





784 dimensions

- Classification problems: finding decision boundaries in high-dimensional space

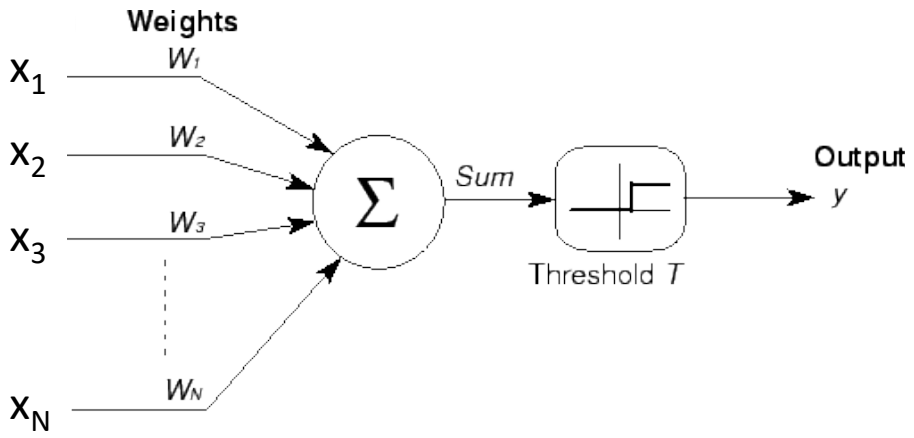
Story so far

- **MLPs are connectionist computational models**
  - Individual perceptrons are computational equivalent of neurons
  - The MLP is a layered composition of many perceptrons
- **MLPs can model Boolean functions**
  - Individual perceptrons can act as Boolean gates
  - Networks of perceptrons are Boolean functions
- **MLPs are Boolean *machines***
  - They represent Boolean functions over linear boundaries
  - They can represent arbitrary decision boundaries
  - They can be used to *classify* data

So what does the perceptron really model?

- Is there a “semantic” interpretation?

Lets look at the weights

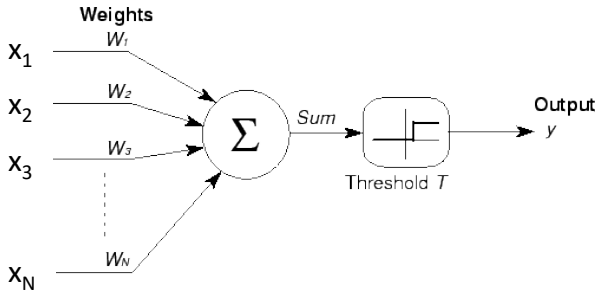


$$y = \begin{cases} 1 \; if \; \sum_i w_i x_i \geq T \\ 0 \; else \end{cases}$$

$$y = \begin{cases} 1 \; if \; \mathbf{x}^T \mathbf{w} \geq T \\ 0 \; else \end{cases}$$

- What do the *weights* tell us?
  - The neuron fires if the inner product between the weights and the inputs exceeds a threshold

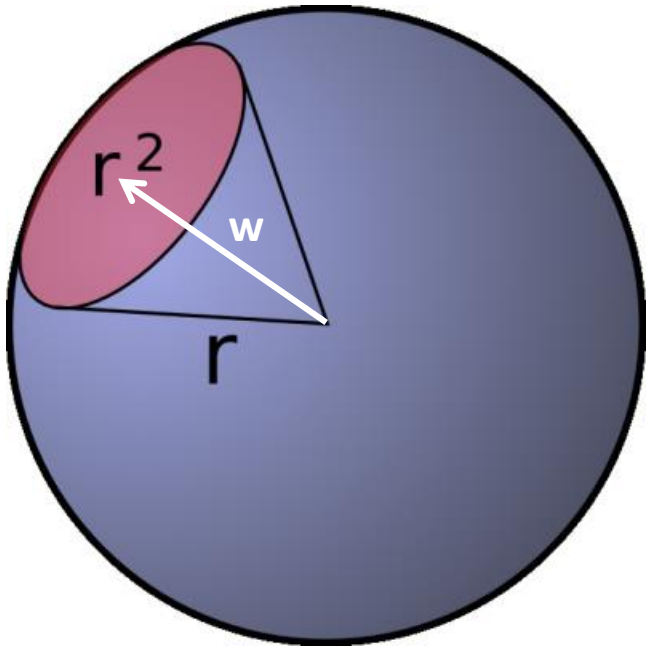
The weight as a "template"



$$X^T W > T$$

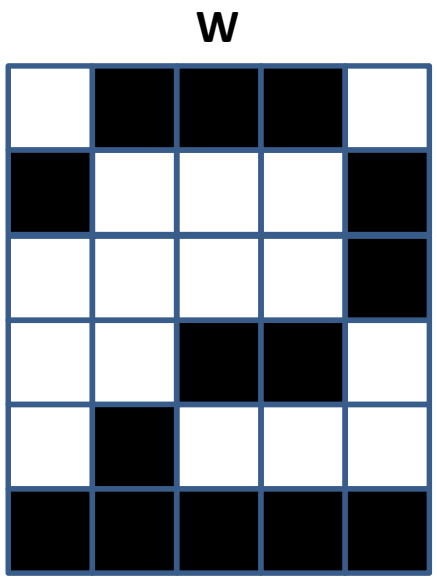
$$\cos\theta > \frac{T}{|X|}$$

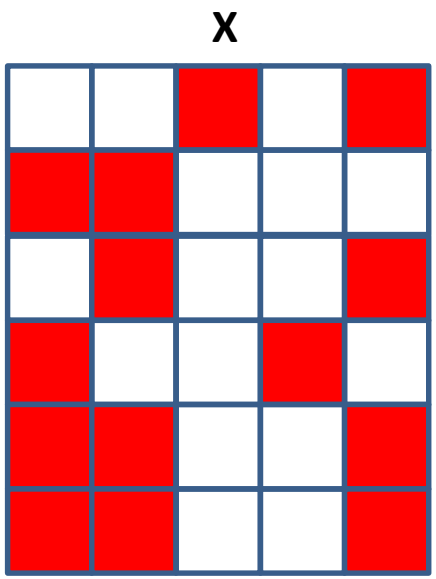
$$\theta < cos^{-1}\left(\frac{T}{|X|}\right)$$



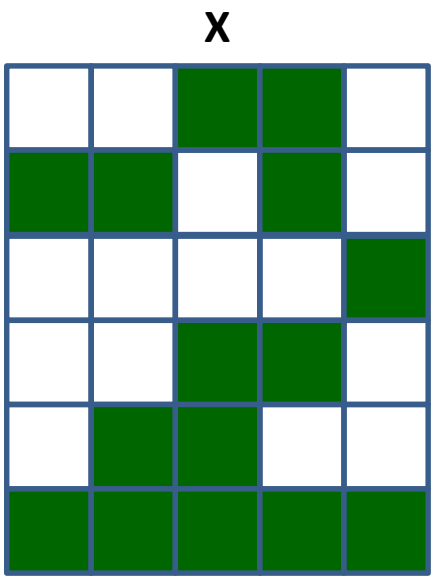
- The perceptron fires if the input is within a specified angle of the weight
- Neuron fires if the input vector is close enough to the weight vector.
  - If the input pattern matches the weight pattern closely enough

The weight as a template





Correlation = 0.57

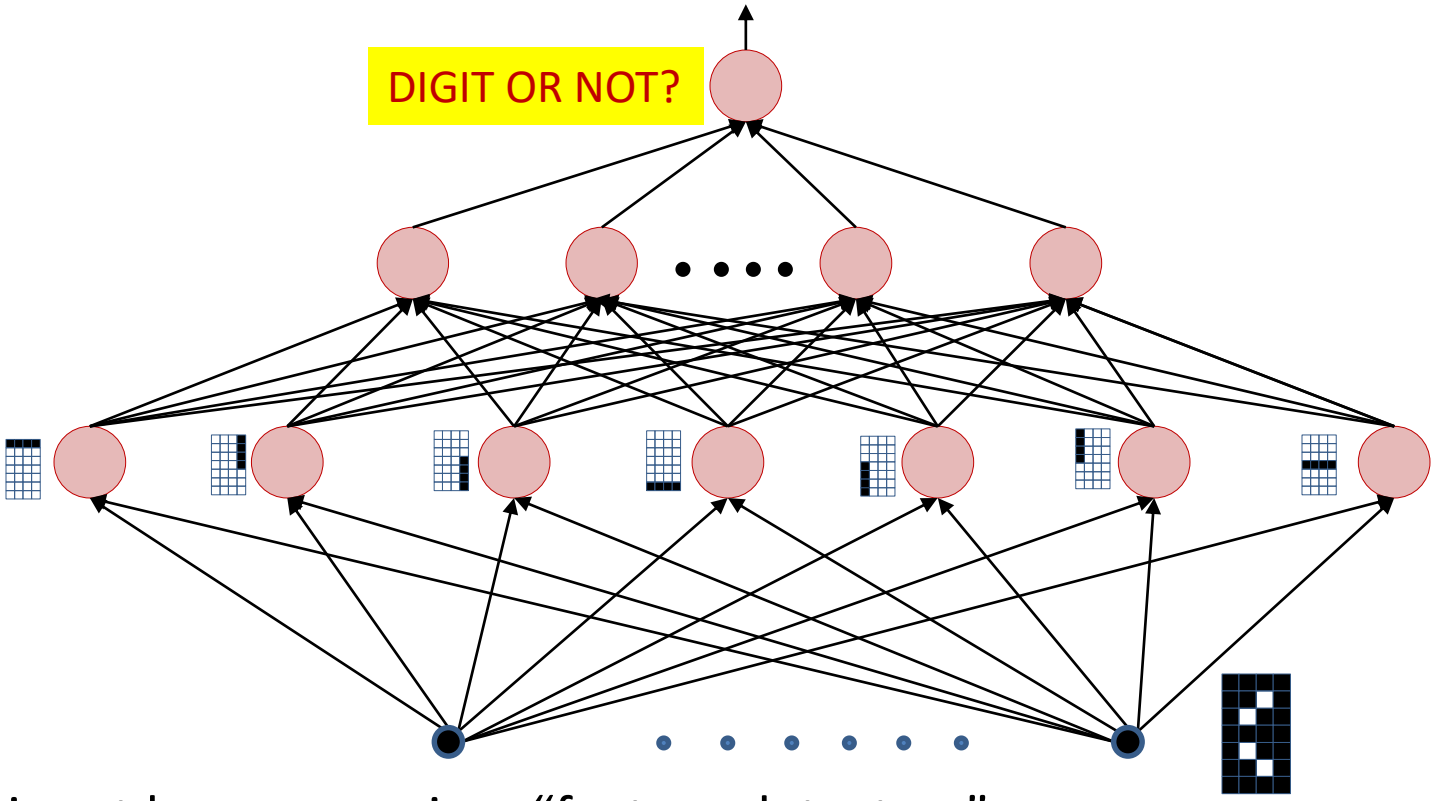


Correlation = 0.82

$$y = \begin{cases} 1 \; if \sum_i w_i x_i \geq T \\ 0 \; else \end{cases}$$

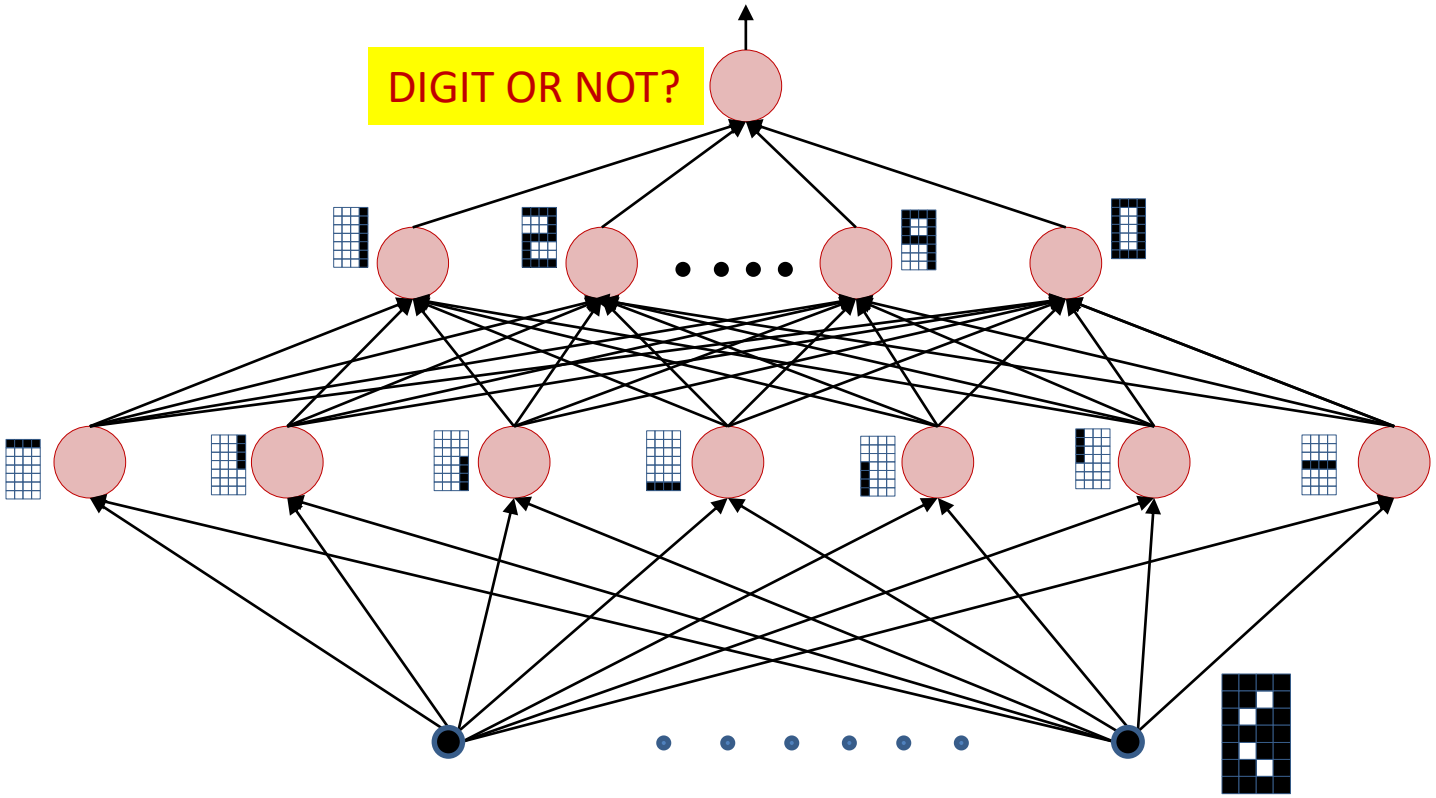
- If the *correlation* between the weight pattern and the inputs exceeds a threshold, fire
- The perceptron is a *correlation filter!*

The MLP as a Boolean function over feature detectors



- The input layer comprises “feature detectors”
  - Detect if certain patterns have occurred in the input
- The network is a Boolean function over the feature detectors
- I.e. it is important for the *first* layer to capture relevant patterns

The MLP as a cascade of feature detectors

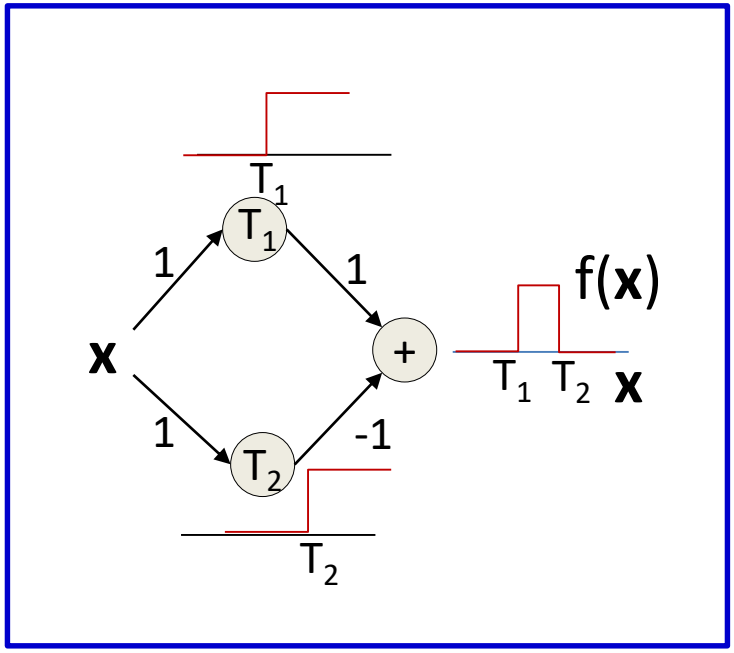


- The network is a cascade of feature detectors
  - Higher level neurons compose complex templates from features represented by lower-level neurons

Story so far

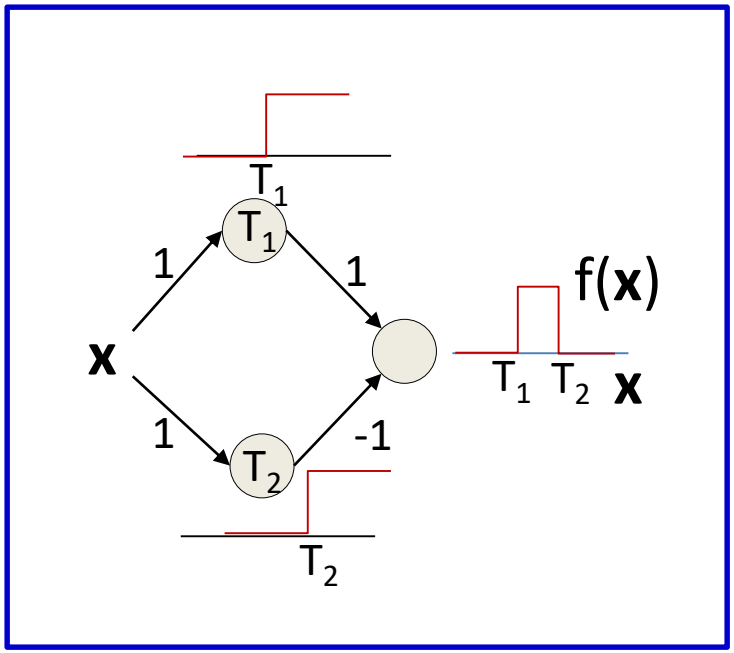
- **Multi-layer perceptrons are connectionist computational models**
- **MLPs are Boolean *machines***
  - They can model Boolean functions
  - They can represent arbitrary decision boundaries over real inputs
- **Perceptrons are *correlation filters***
  - They detect patterns in the input
- **MLPs are Boolean formulae over patterns detected by perceptrons**
  - Higher-level perceptrons may also be viewed as feature detectors
- **Extra: MLP in classification**
  - The network will fire if the combination of the detected basic features matches an "acceptable" pattern for a desired class of signal
    - E.g. Appropriate combinations of (Nose, Eyes, Eyebrows, Cheek, Chin) → Face

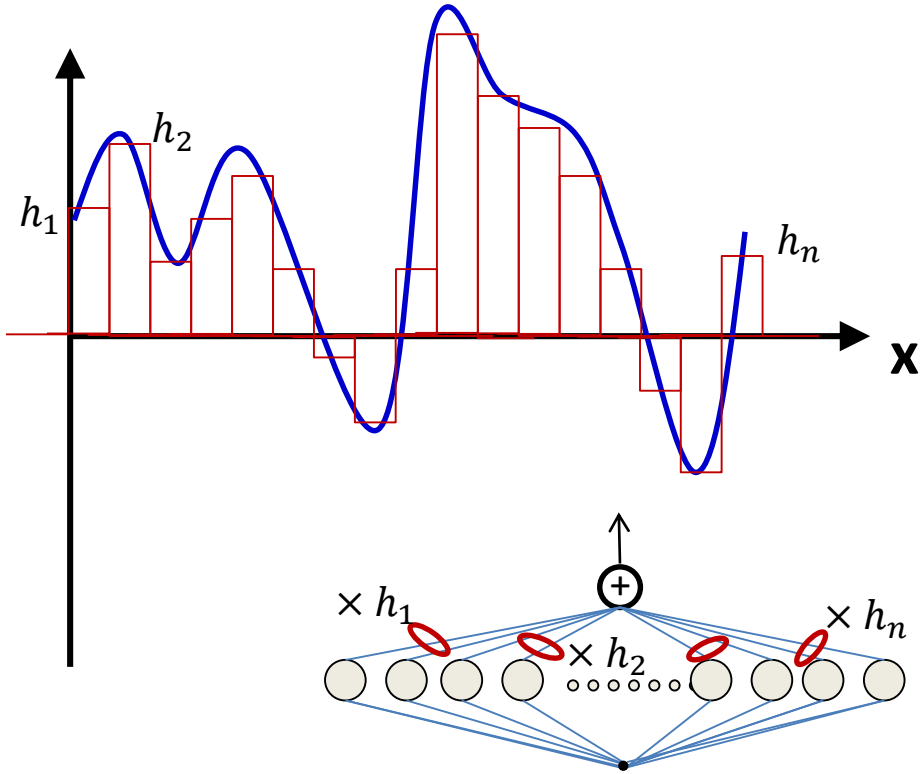
MLP as a continuous-valued regression



- 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





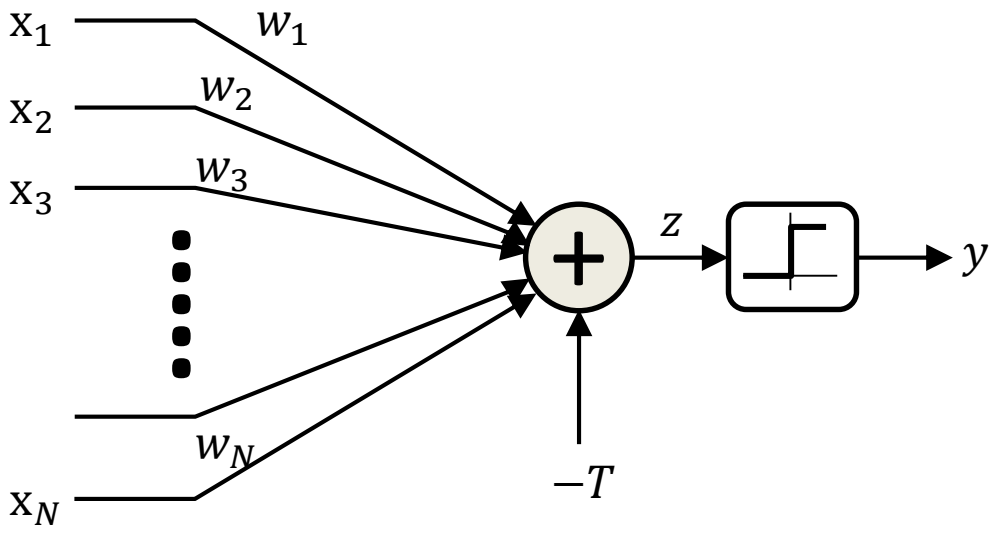
- A simple 3-unit MLP can generate a "square pulse" over an input
- **An MLP with many units can model an arbitrary function over an input**
  - To arbitrary precision
    - Simply make the individual pulses narrower
- This generalizes to functions of any number of inputs (next part)

Story so far

- **Multi-layer perceptrons are connectionist computational models**
- **MLPs are *classification engines***
  - They can identify classes in the data
  - Individual perceptrons are feature detectors
  - The network will fire if the combination of the detected basic features matches an "acceptable" pattern for a desired class of signal
- **MLP can also model continuous valued functions**

Neural Networks:
Part 2: What can a network represent

Recap: The perceptron

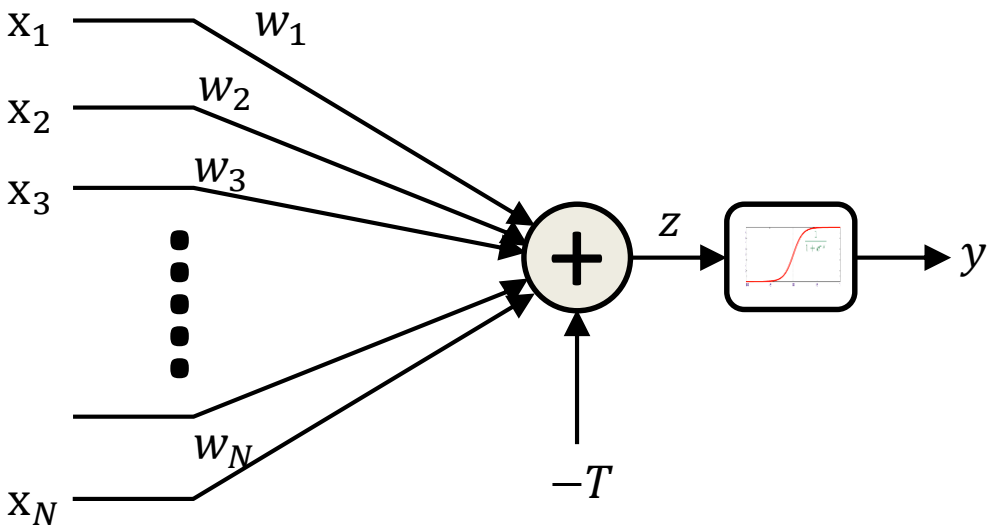


$$z = \sum_i w_i x_i - T$$

$$y = \begin{cases} 1 \; if \; z \geq 0 \\ 0 \; else \end{cases}$$

- A threshold unit
  - "Fires" if the weighted sum of inputs and the "bias" T is positive

The "soft" perceptron

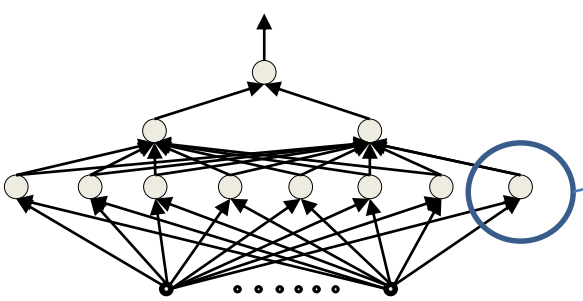


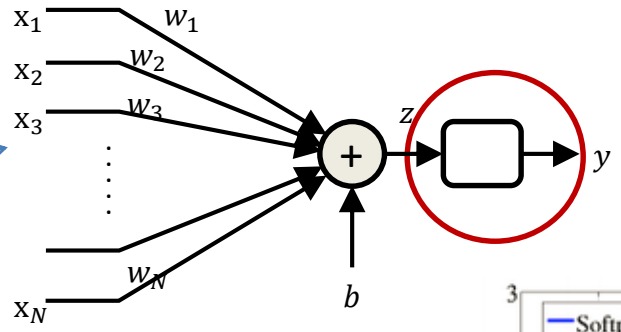
$$z = \sum_i w_i x_i - T$$

$$y = \frac{1}{1 + exp(-z)}$$

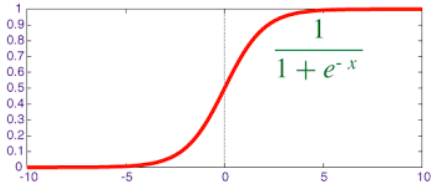
- A "squashing" function instead of a threshold at the output
  - The sigmoid "activation" replaces the threshold
    - **Activation:** The function that acts on the weighted combination of inputs (and threshold)

Other “activations”

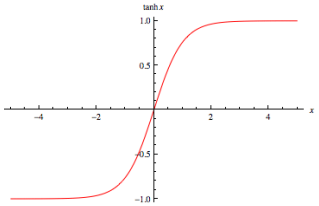


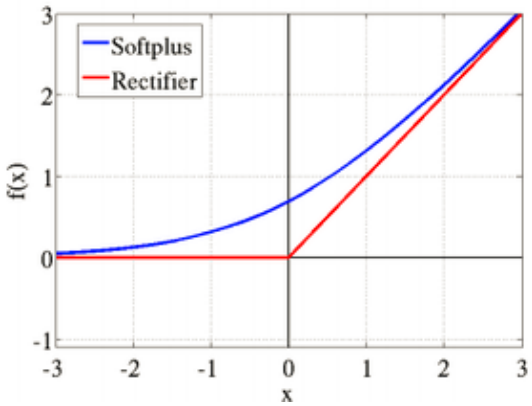


sigmoid



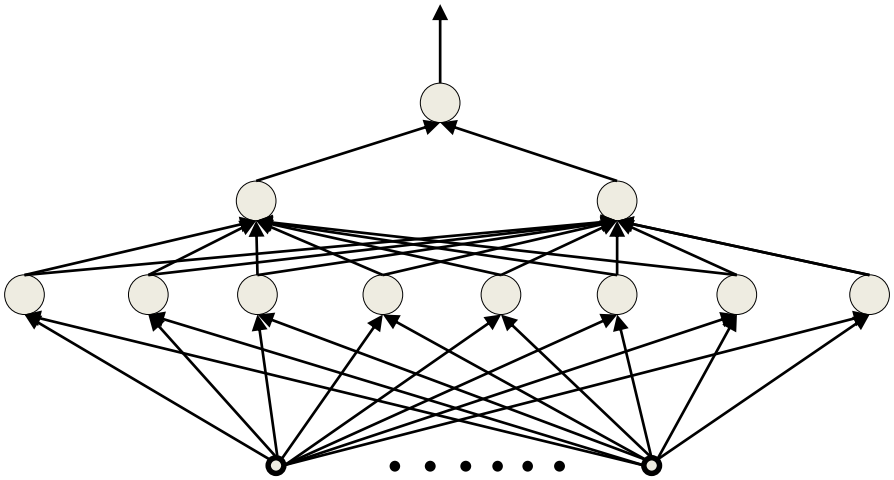
tanh

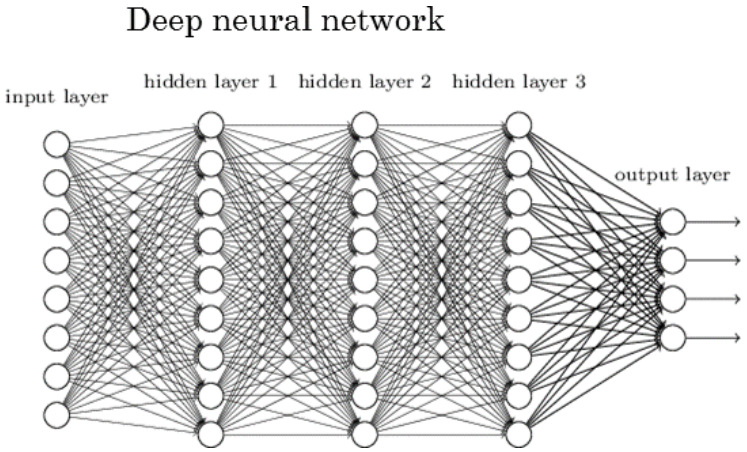




- Does not always have to be a squashing function
- We will continue to assume a “threshold” activation in this lecture

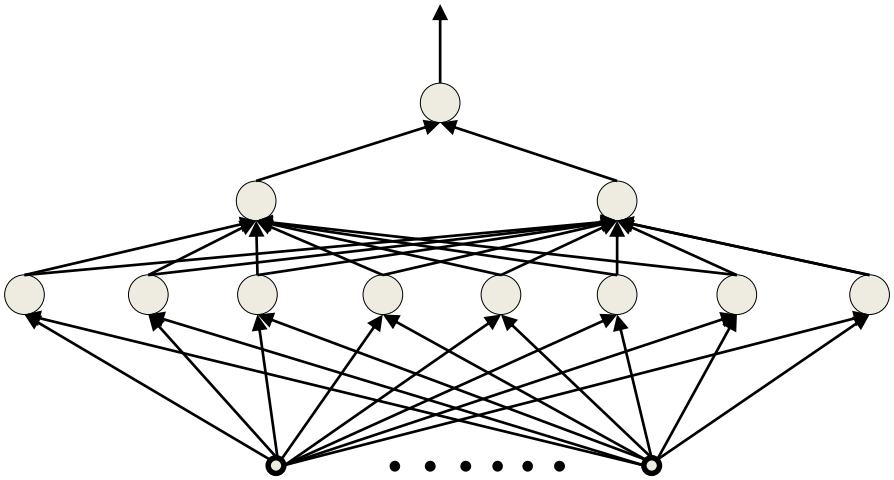
Recap: the multi-layer perceptron

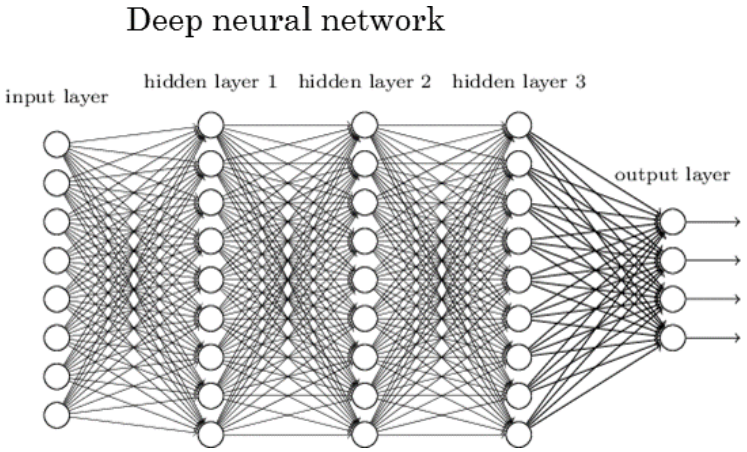




- A network of perceptrons
  - Generally “layered”

Aside: Note on “depth”

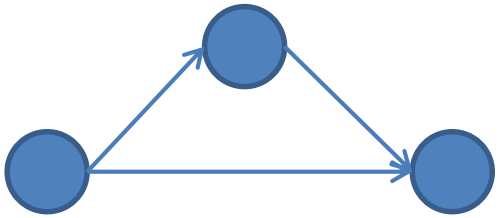


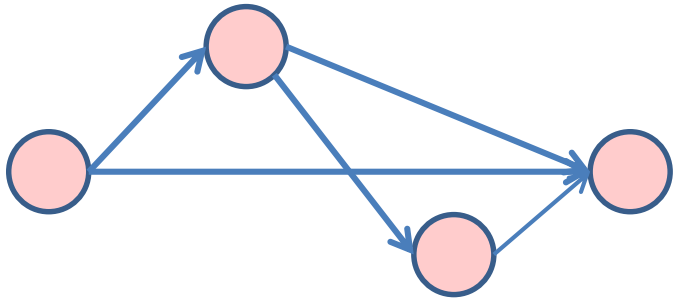


- What is a “deep” network

Deep Structures

- In any directed network of computational elements with input source nodes and output sink nodes, “depth” is the length of the longest path from a source to a sink

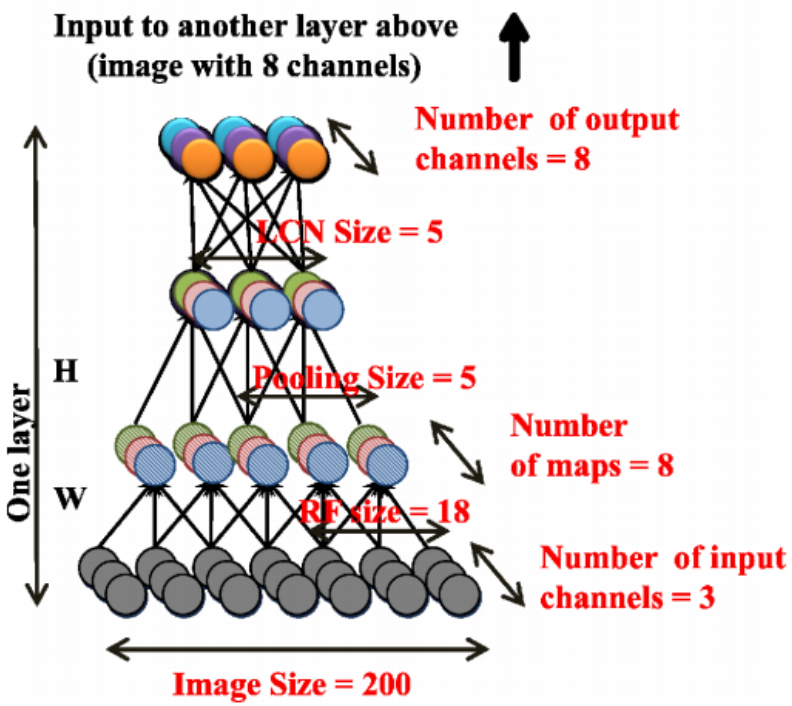




- Left: Depth = 2. Right: Depth = 3

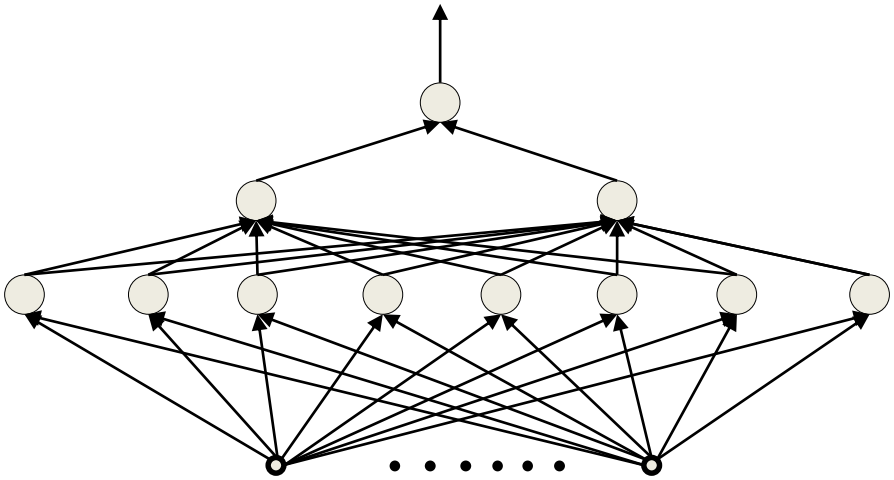
Deep Structures

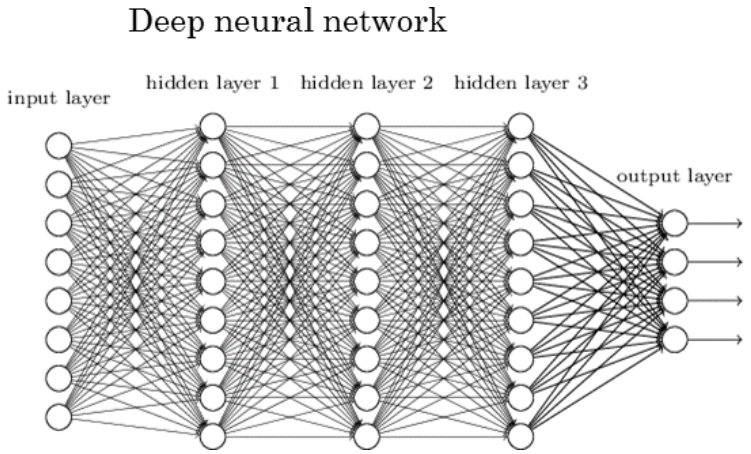
- *Layered* deep structure



- "Deep" → Depth > 2

The multi-layer perceptron

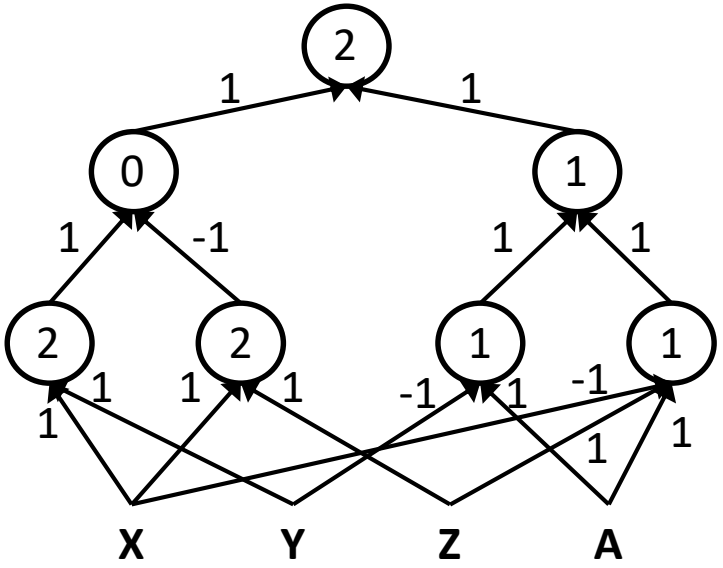


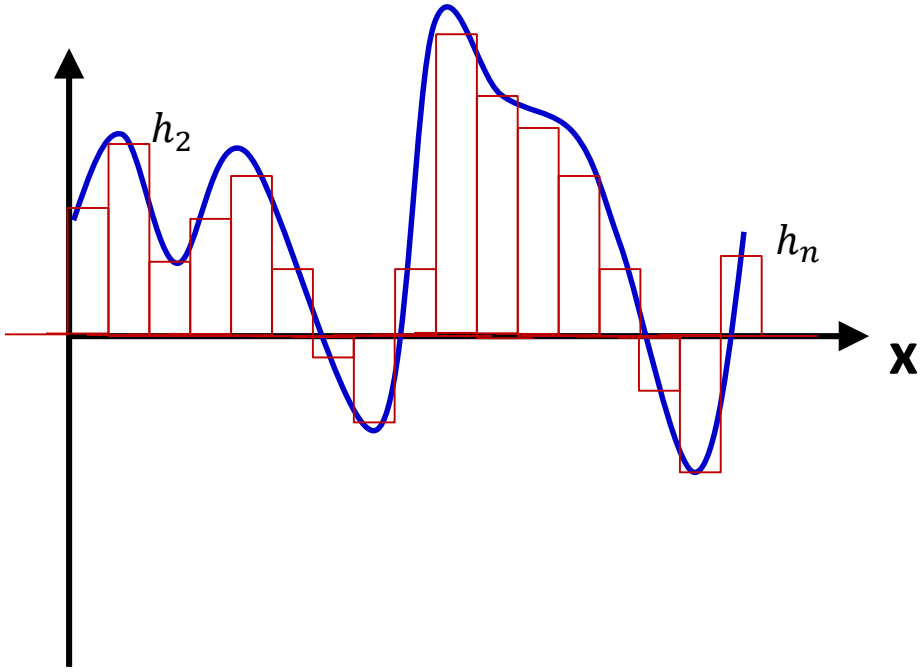


- Inputs are real or Boolean stimuli
- Outputs are real or Boolean values
  - Can have multiple outputs for a single input
- **What can this network compute?**
  - **What kinds of input/output relationships can it model?**

MLPs approximate functions

$((A\&\bar{X}\&Z)|(A\&\bar{Y}))\&((X\ \&\ Y)|\overline{(X\&Z)})$



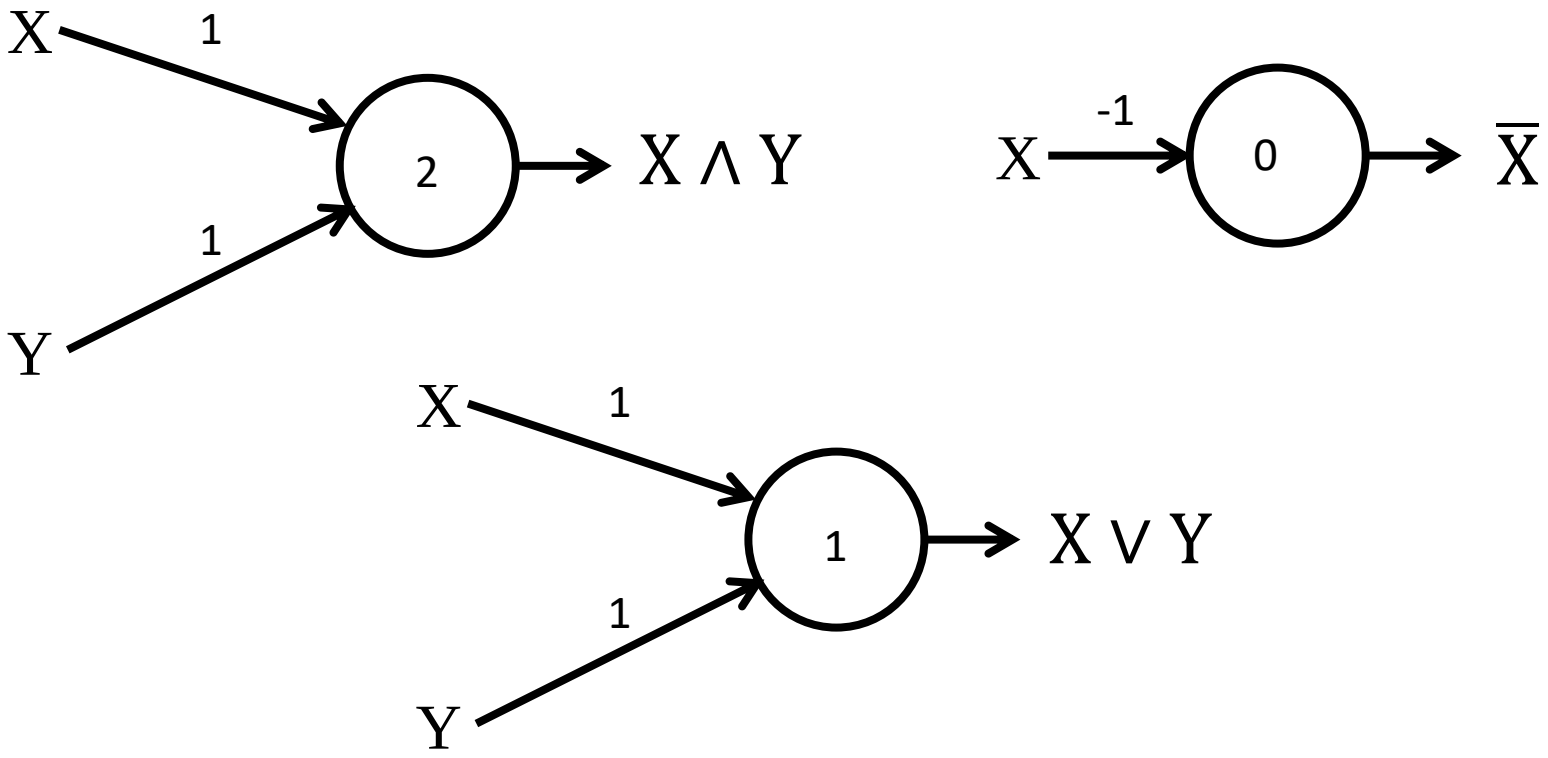


- MLPs can compose Boolean functions
- MLPs can compose real-valued functions
- What are the limitations?

The MLP as a Boolean function

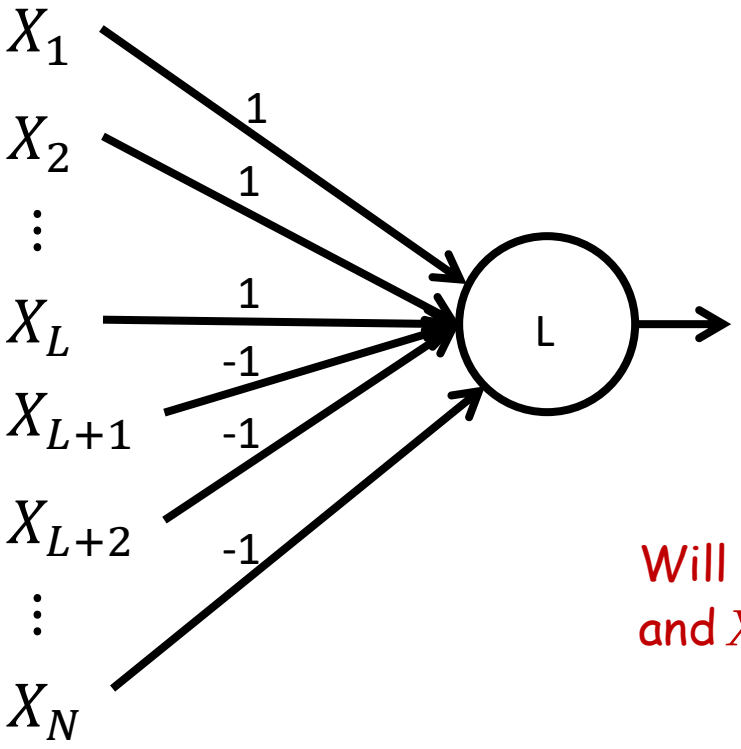
- How well do MLPs model Boolean functions?

The perceptron as a Boolean gate



- A perceptron can model any simple binary Boolean gate

Perceptron as a Boolean gate

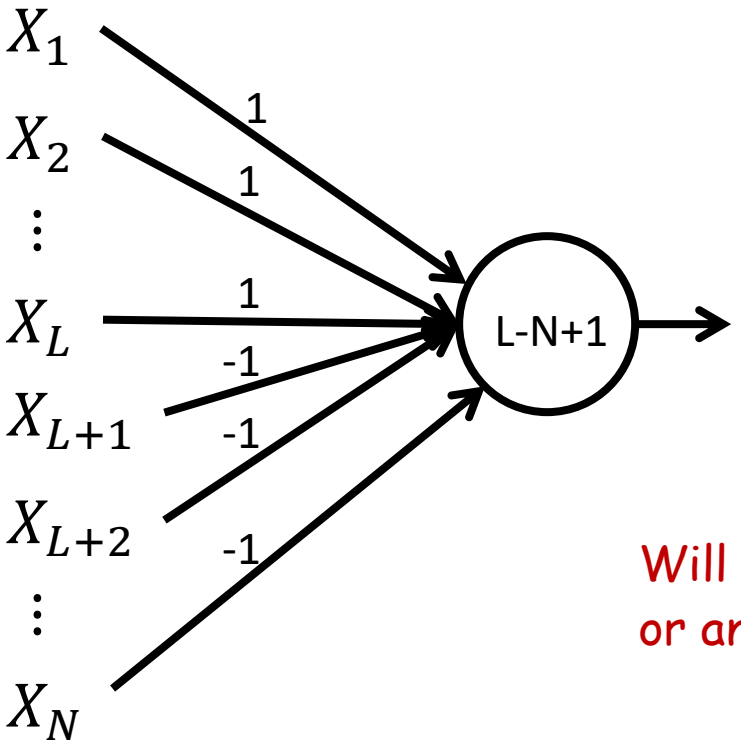


$$\left(\bigwedge_{i=1}^{L} X_i\right) \wedge \left(\bigwedge_{i=L+1}^{N} \bar{X}_i\right)$$

Will fire only if X_1 .. X_L are all 1
and X_{L+1} .. X_N are all 0

- The universal AND gate
  - AND any number of inputs
    - Any subset of who may be negated

Perceptron as a Boolean gate

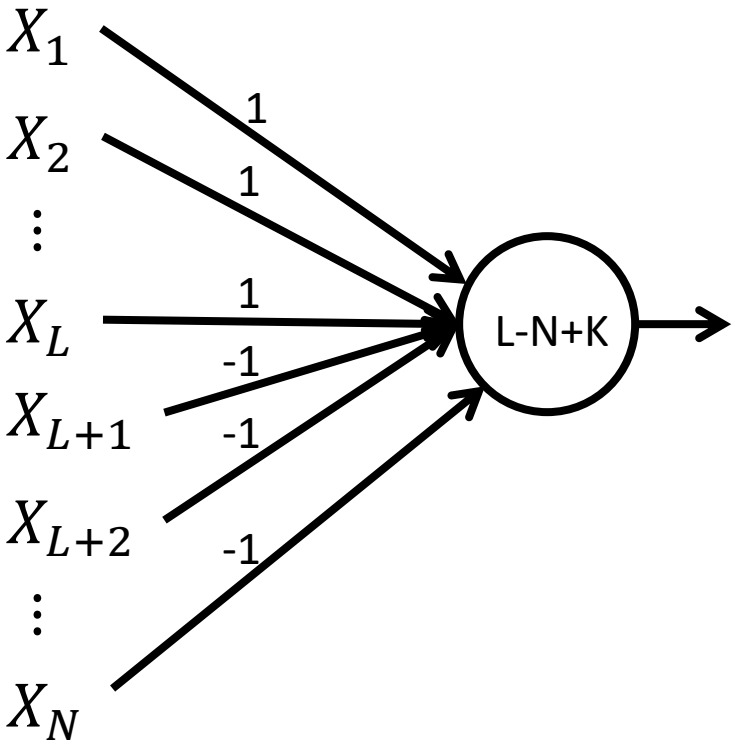


$$\left(\bigvee_{i=1}^{L} X_i\right) \vee \left(\bigvee_{i=L+1}^{N} \bar{X}_i\right)$$

Will fire only if any of X_1 .. X_L are 1
or any of X_{L+1} .. X_N are 0

- The universal OR gate
  - OR any number of inputs
    - Any subset of who may be negated

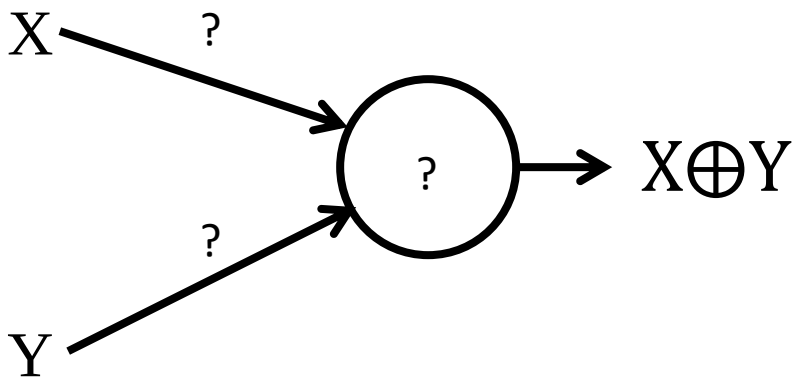
Perceptron as a Boolean Gate



Will fire only if the total number of of X_1 .. X_L that are 1 or X_{L+1} .. X_N that are 0 is at least K

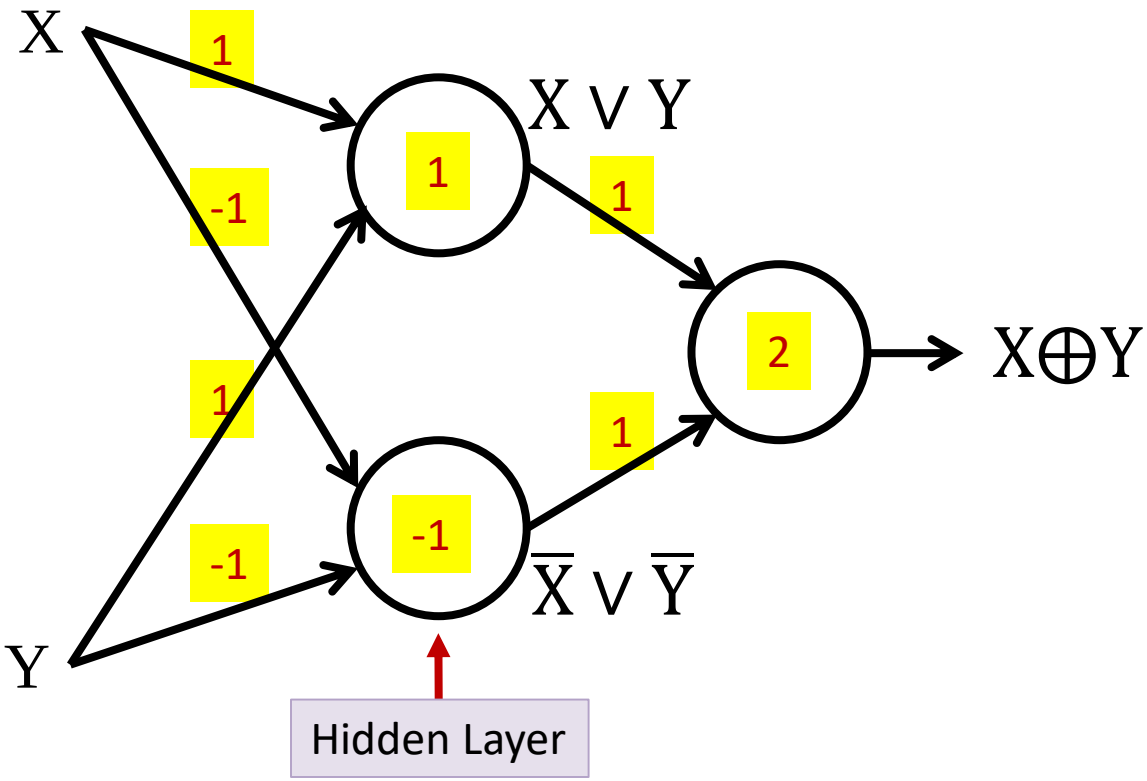
- Universal OR:
  - Fire if any K-*subset* of inputs is "ON"

The perceptron is not enough



- Cannot compute an XOR

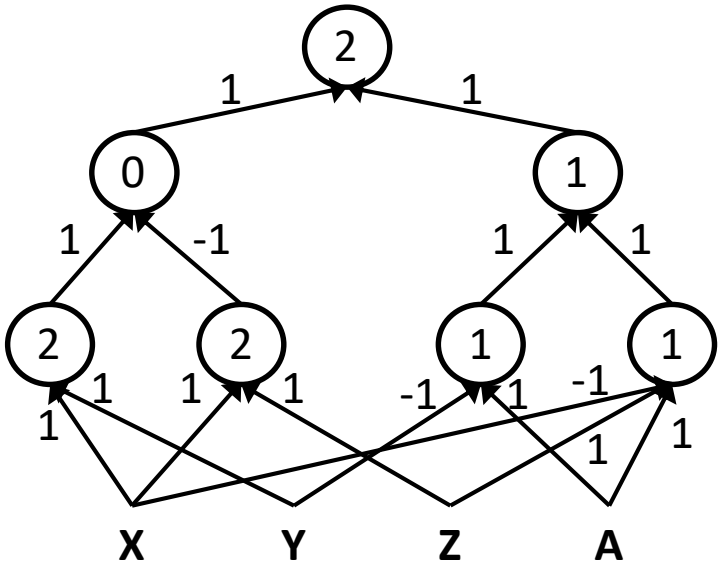
Multi-layer perceptron



- MLPs can compute the XOR

Multi-layer perceptron

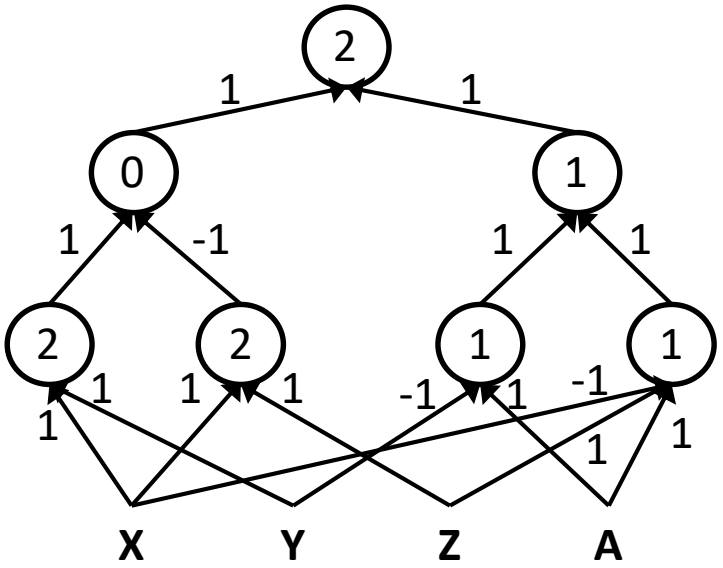
$$((A\&\bar{X}\&Z)|(A\&\bar{Y}))\&((X\ \&\ Y)|\overline{(X\&Z)})$$

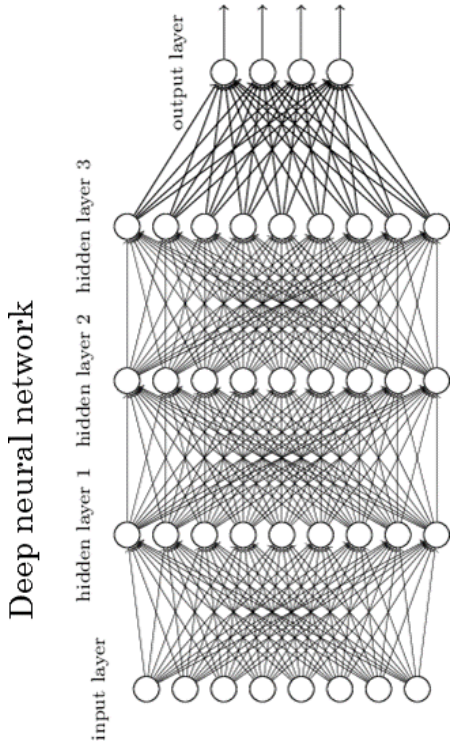


- MLPs can compute more complex Boolean functions
- MLPs can compute *any* Boolean function
  - Since they can emulate individual gates
- **MLPs are *universal Boolean functions***

MLP as Boolean Functions

$((A\&\bar{X}\&Z)|(A\&\bar{Y}))\&((X\ \&\ Y)|\overline{(X\&Z)})$





- MLPs are universal Boolean functions
  - Any function over any number of inputs and any number of outputs
- But how many “layers” will they need?

How many layers for a Boolean MLP?

Truth table shows *all* input combinations for which output is 1

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

- Expressed in disjunctive normal form

How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows *all* input combinations for which output is 1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$

- Expressed in disjunctive normal form

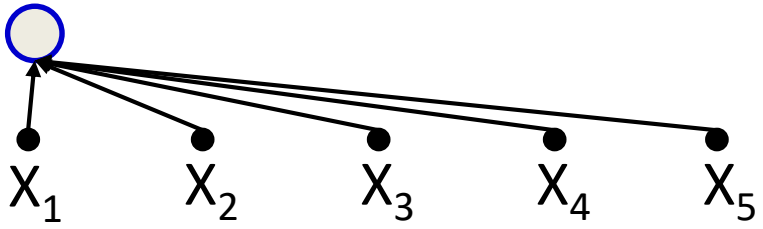
How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows *all* input combinations for which output is 1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3X_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

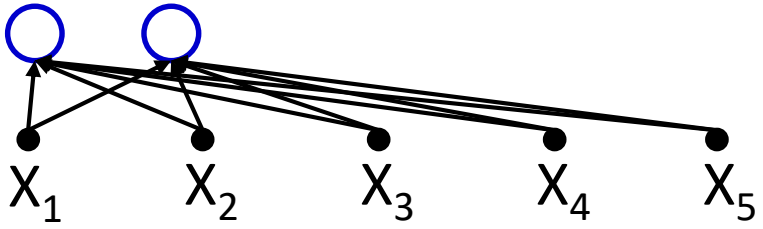
How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows *all* input combinations for which output is 1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

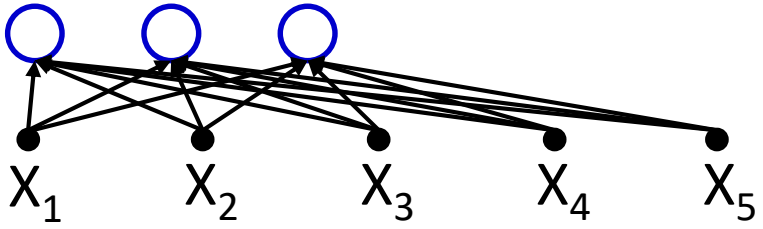
How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows *all* input combinations for which output is 1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2X_3X_4X_5$$



- Expressed in disjunctive normal form

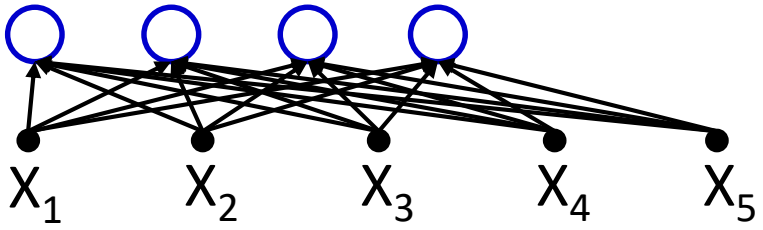
How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows *all* input combinations for which output is 1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

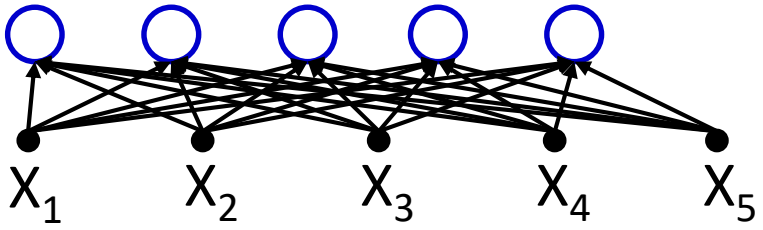
How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows *all* input combinations for which output is 1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

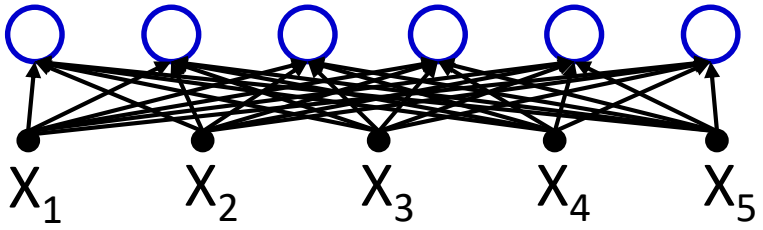
How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows *all* input combinations for which output is 1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

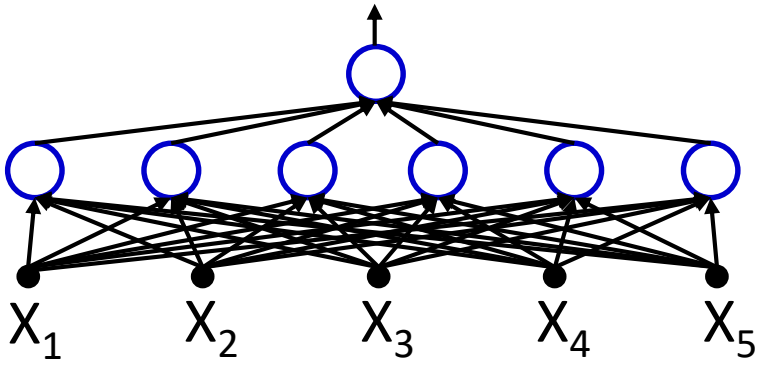
How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows *all* input combinations for which output is 1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

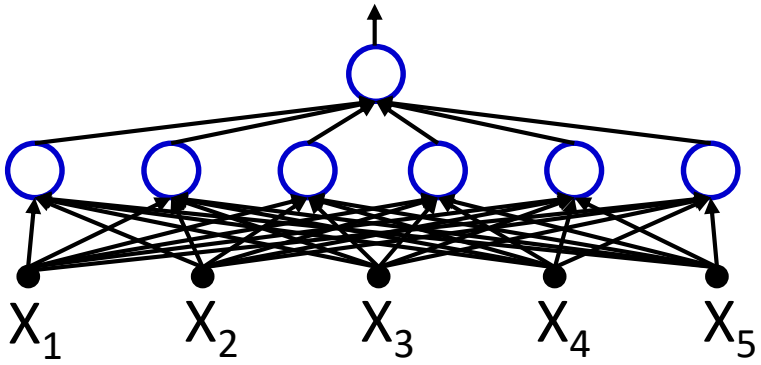
How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows *all* input combinations for which output is 1

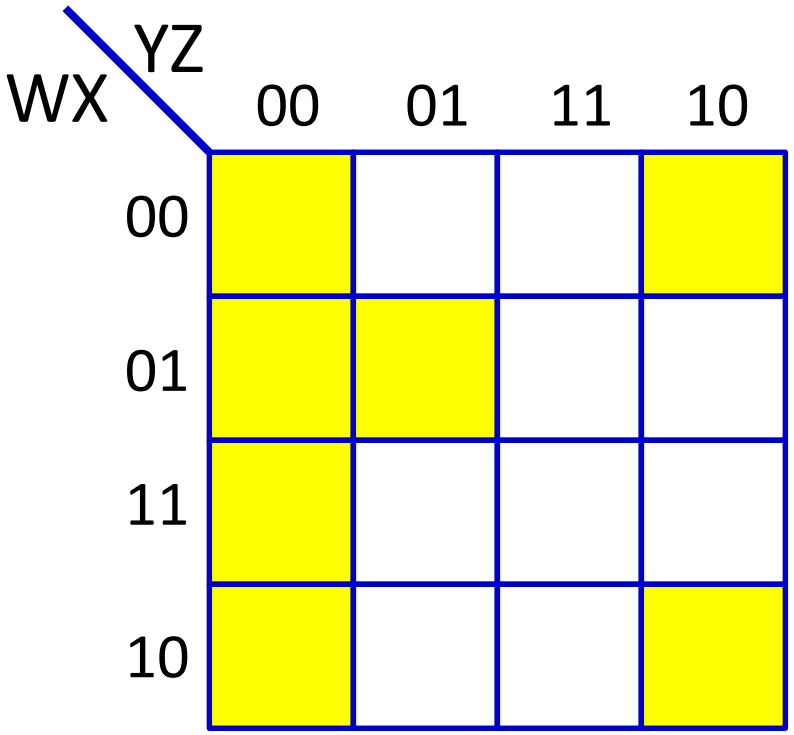
$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- *Any truth table can be expressed in this manner!*
- **A one-hidden-layer MLP is a Universal Boolean Function**

But what is the largest number of perceptrons required in the single hidden layer for an N-input-variable function?

Reducing a Boolean Function



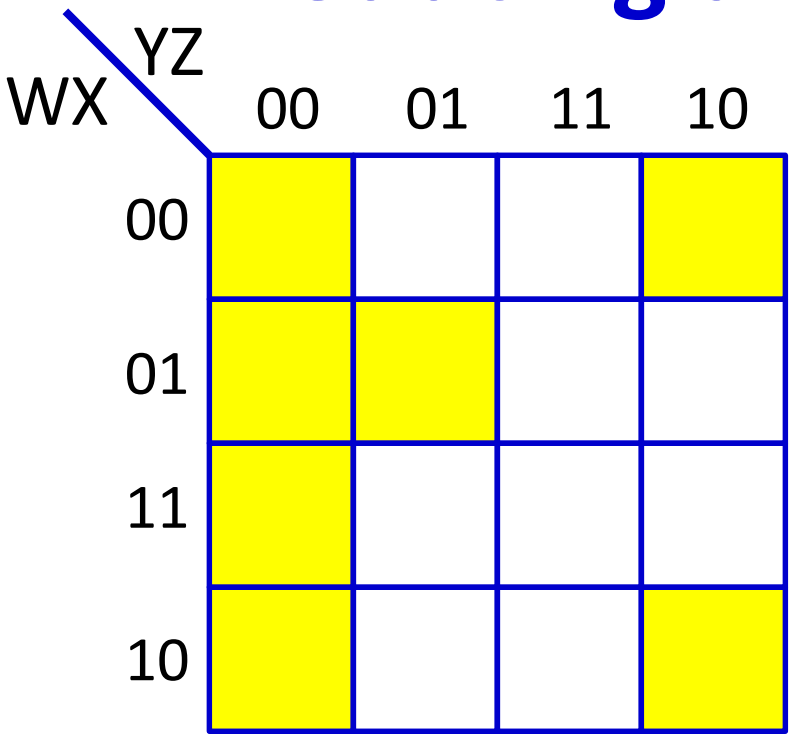
This is a "Karnaugh Map"

It represents a truth table as a grid
Filled boxes represent input combinations for which output is 1; blank boxes have output 0

Adjacent boxes can be "grouped" to reduce the complexity of the DNF formula for the table

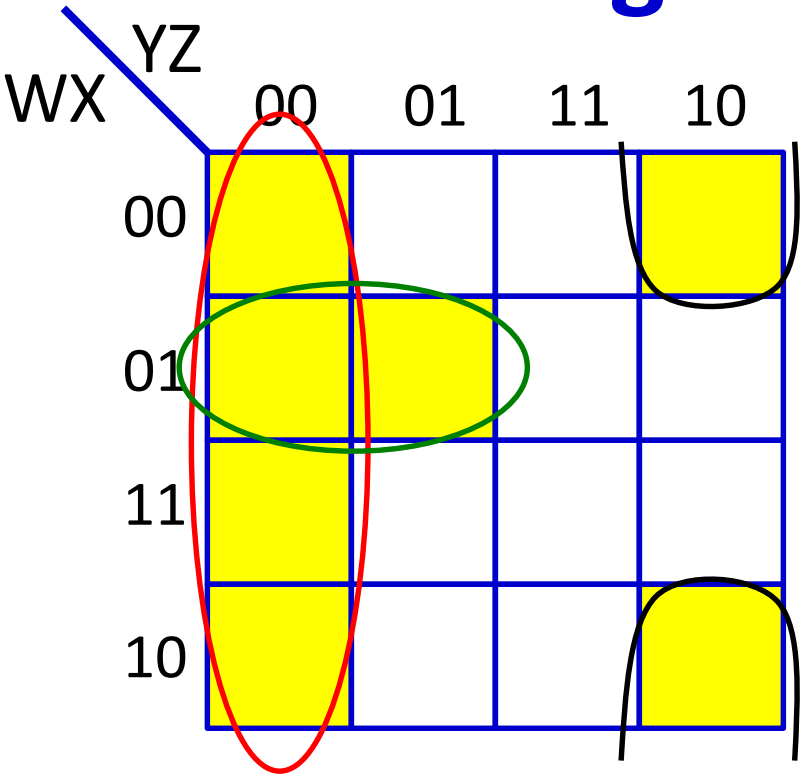
- DNF form:
  - Find groups
  - Express as reduced DNF

Reducing a Boolean Function



Basic DNF formula will require 7 terms

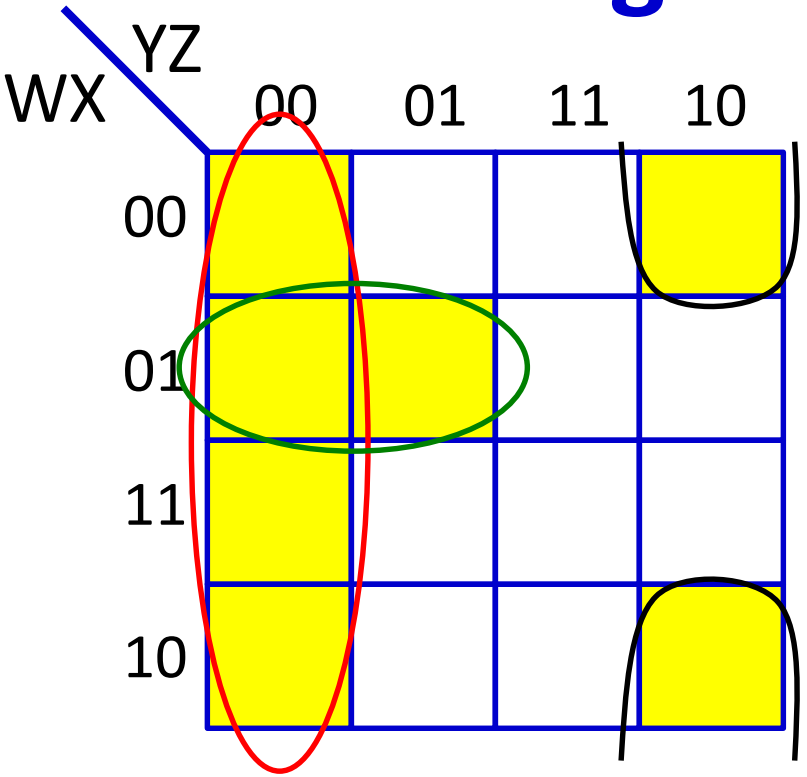
Reducing a Boolean Function



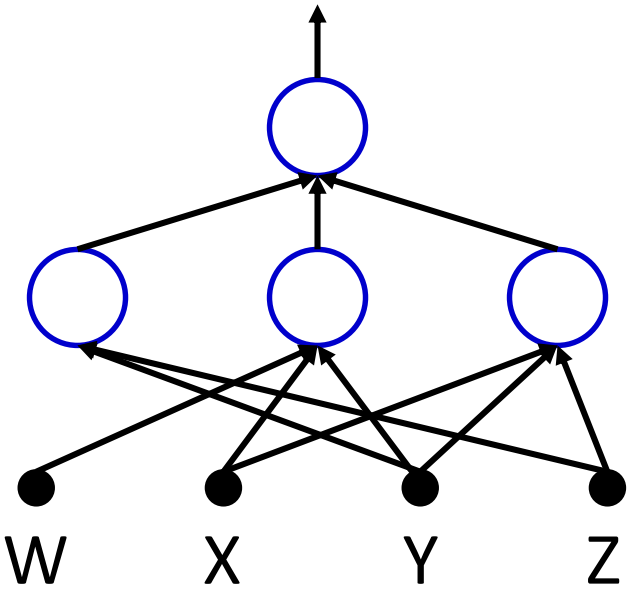
$$O = \bar{Y}\bar{Z} + \bar{W}X\bar{Y} + \bar{X}Y\bar{Z}$$

- *Reduced* DNF form:
  - Find groups
  - Express as reduced DNF

Reducing a Boolean Function

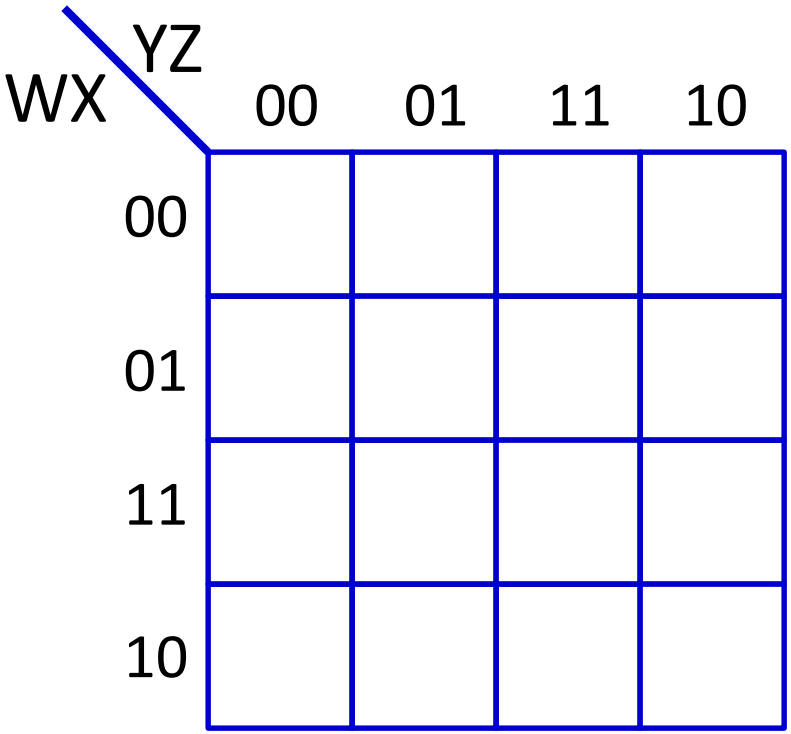


$$O = \bar{Y}\bar{Z} + \bar{W}X\bar{Y} + \bar{X}Y\bar{Z}$$



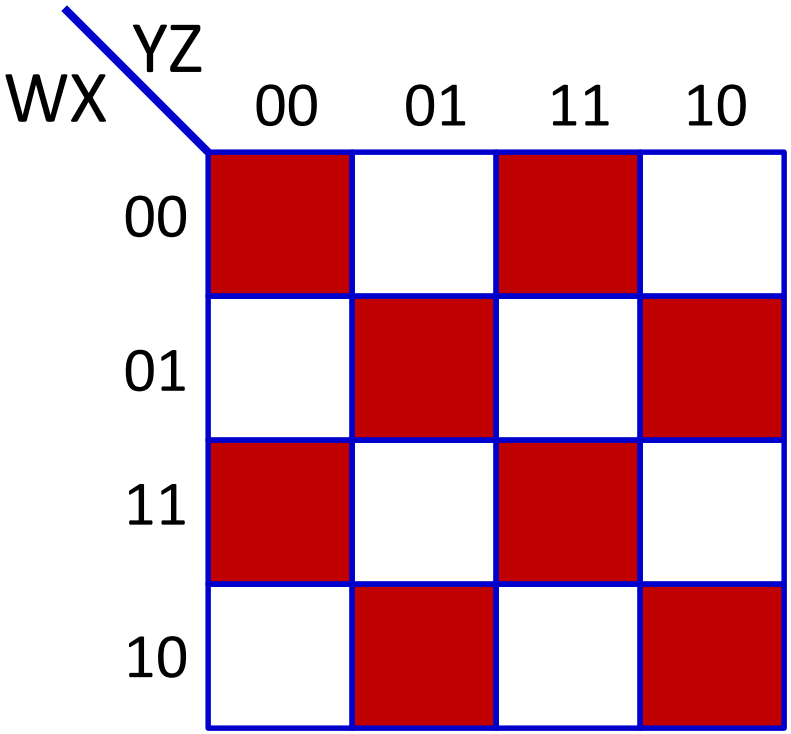
- *Reduced* DNF form:
  - Find groups
  - Express as reduced DNF

Largest irreducible DNF?



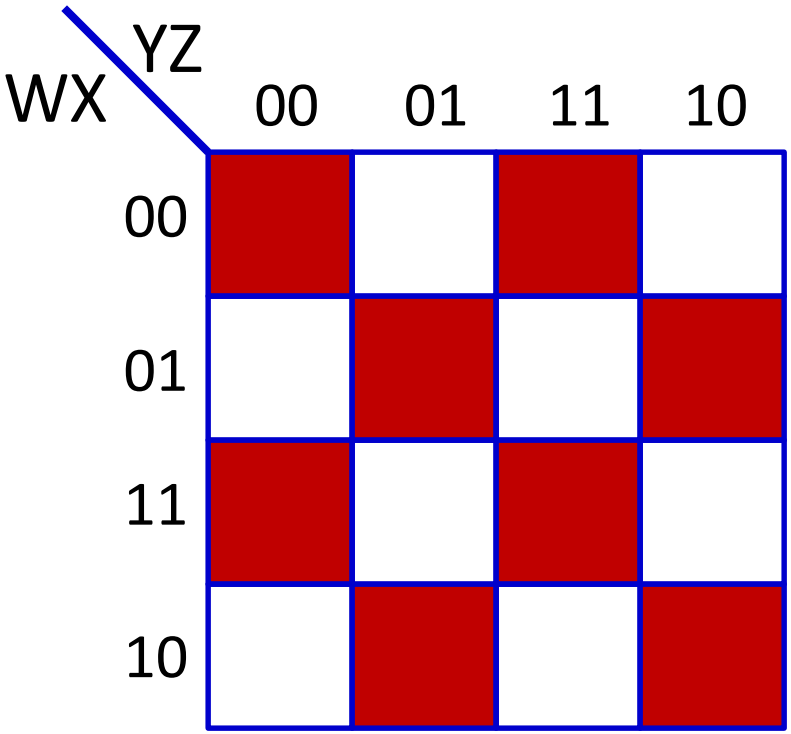
- What arrangement of ones and zeros simply cannot be reduced further?

Largest irreducible DNF?



- What arrangement of ones and zeros simply cannot be reduced further?

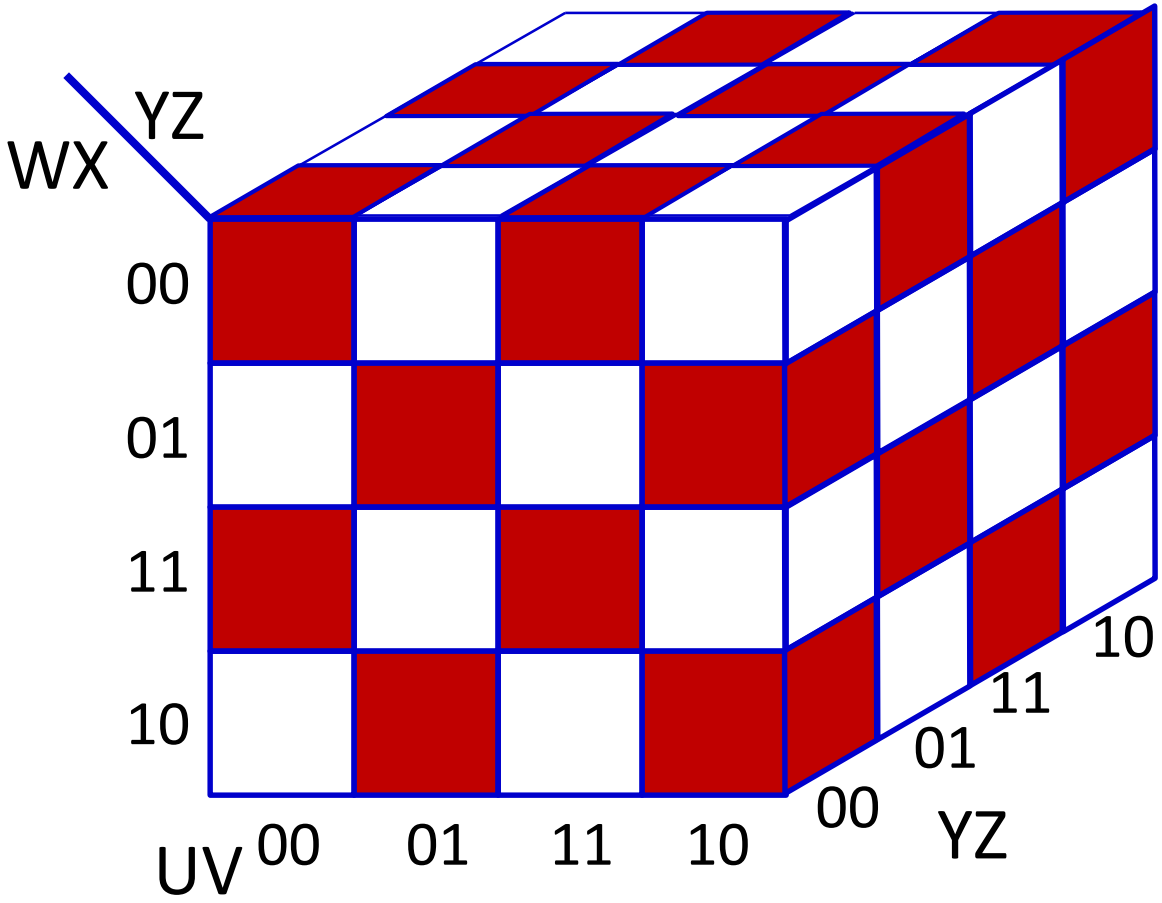
Largest irreducible DNF?



How many neurons in a DNF (one-hidden-layer) MLP for this Boolean function?

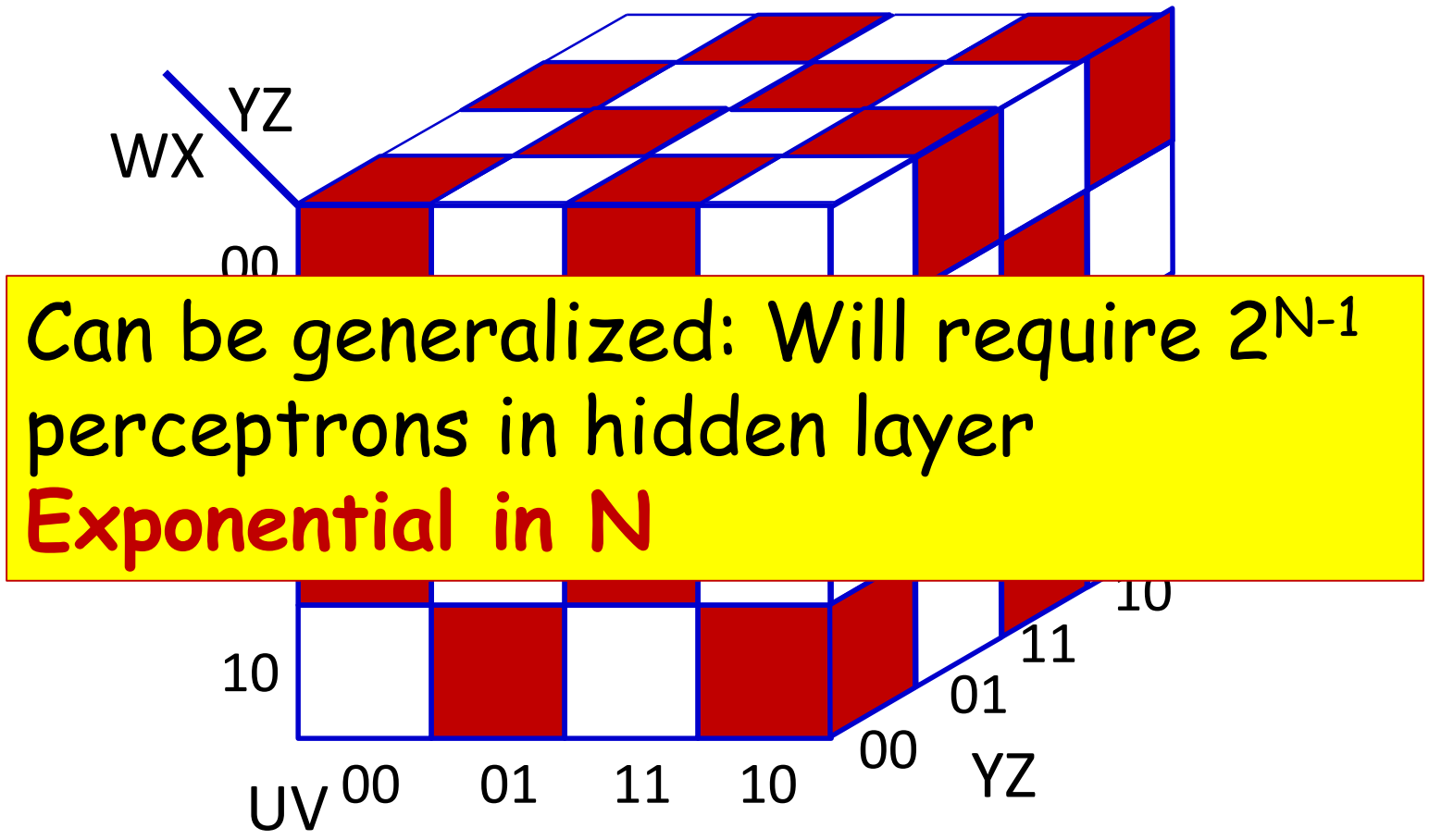
- What arrangement of ones and zeros simply cannot be reduced further?

Width of a single-layer Boolean MLP



- How many neurons in a DNF (one-hidden-layer) MLP for this Boolean function of 6 variables?

Width of a single-layer Boolean MLP

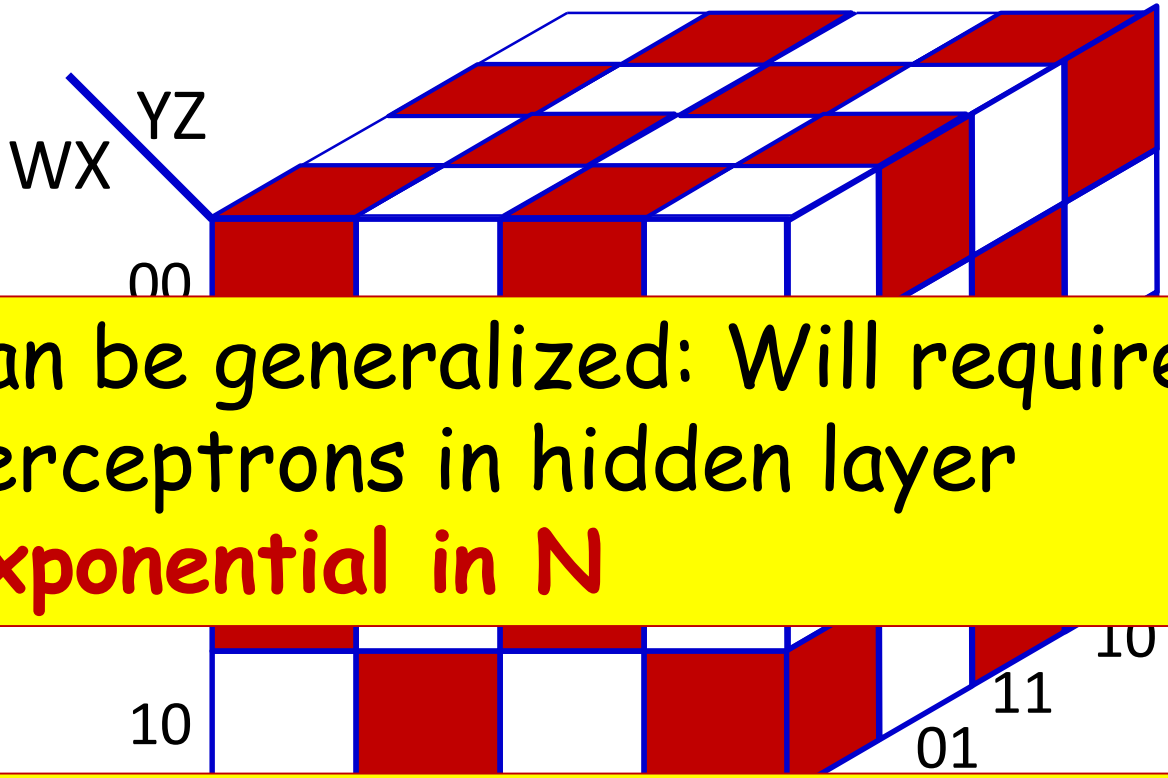


Can be generalized: Will require 2^{N-1} perceptrons in hidden layer

Exponential in N

- How many neurons in a DNF (one-hidden-layer) MLP for this Boolean function

Width of a single-layer Boolean MLP



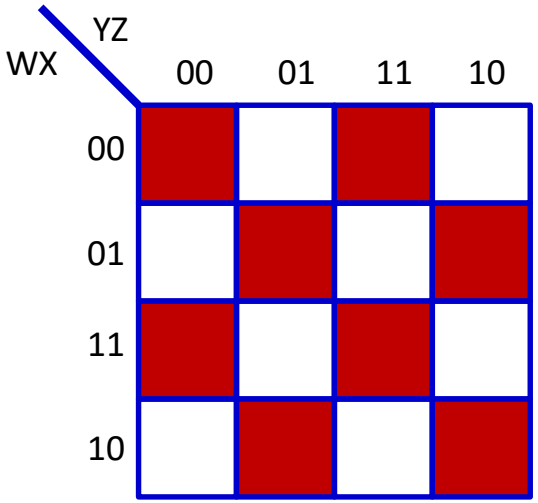
Can be generalized: Will require 2^{N-1} perceptrons in hidden layer

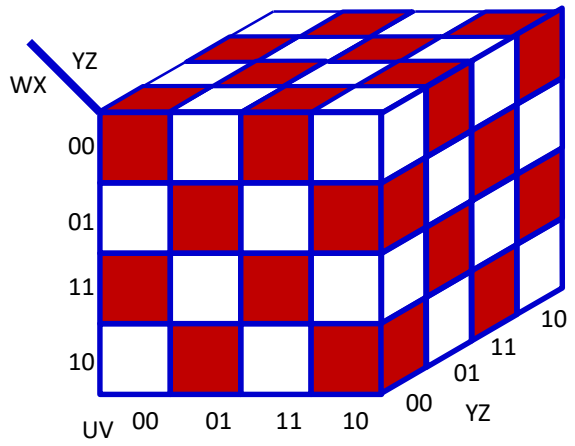
Exponential in N

How many units if we use *multiple layers*?

- How many neurons in a DNF (one-hidden-layer) MLP for this Boolean function

Width of a deep MLP

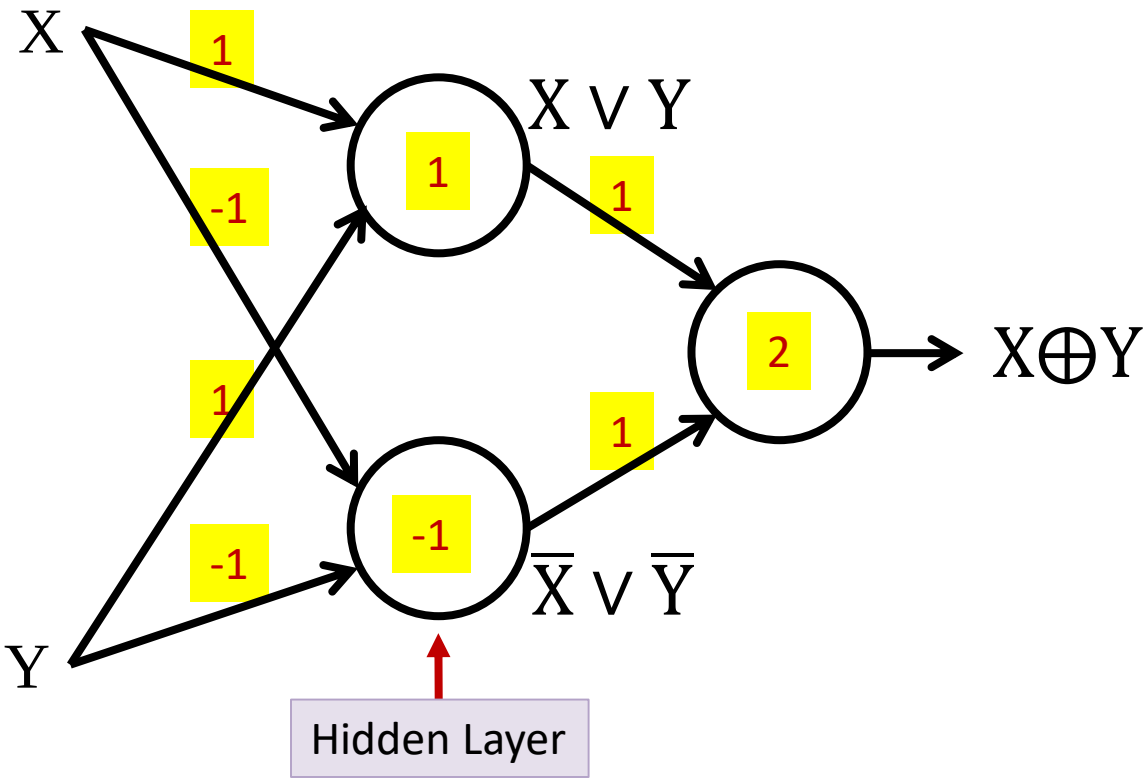




$$O = W \oplus X \oplus Y \oplus Z$$

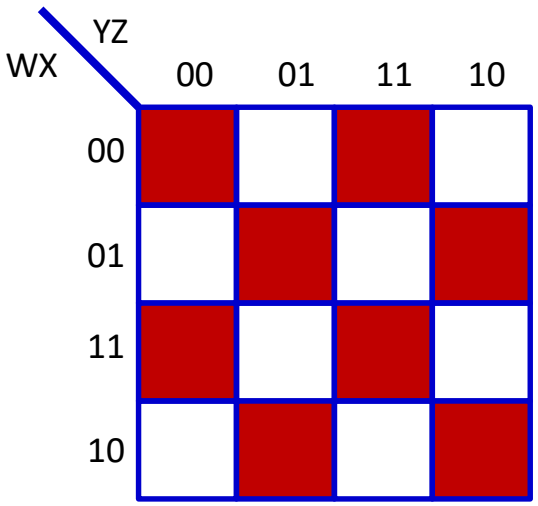
$$O = U \oplus V \oplus W \oplus X \oplus Y \oplus Z$$

Multi-layer perceptron XOR

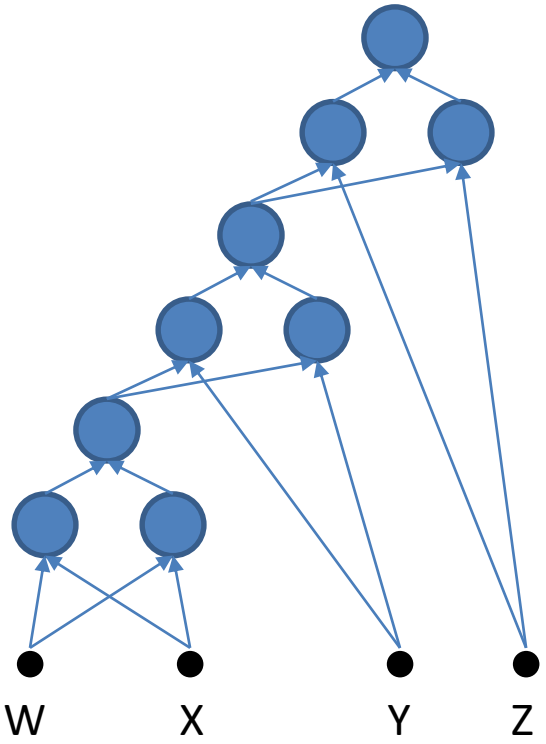


- An XOR takes three perceptrons

Width of a deep MLP



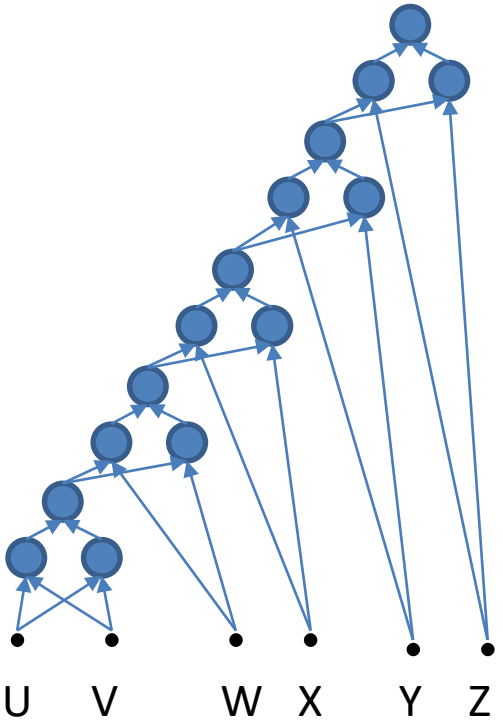
$$O = W \oplus X \oplus Y \oplus Z$$

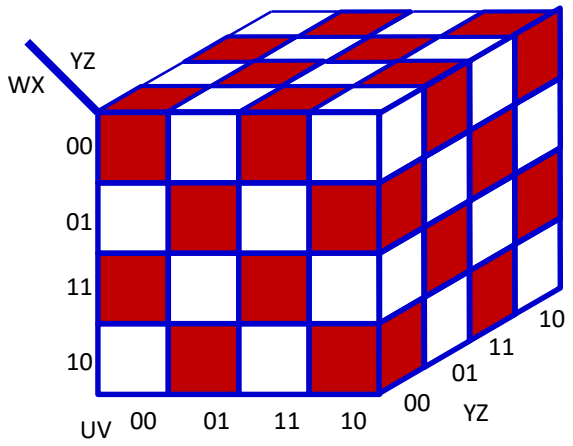


9 perceptrons

- An XOR needs 3 perceptrons
- This network will require 3x3 = 9 perceptrons

Width of a deep MLP



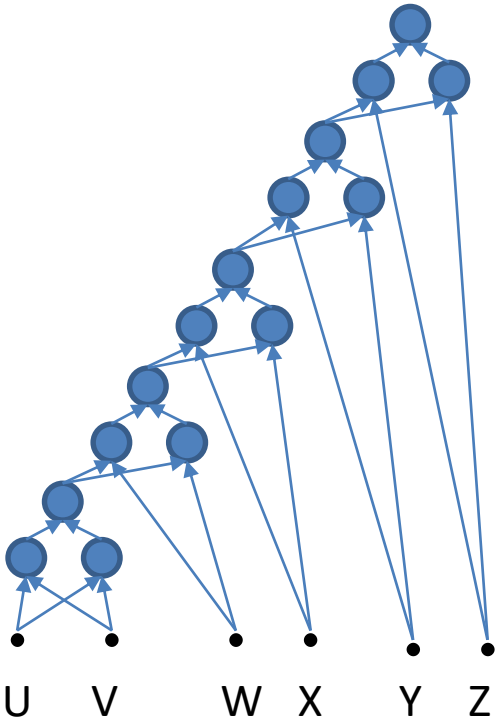


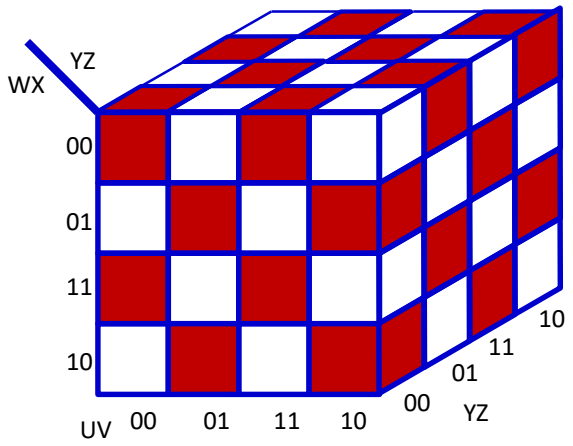
$$O = U \oplus V \oplus W \oplus X \oplus Y \oplus Z$$

15 perceptrons

- An XOR needs 3 perceptrons
- This network will require 3x5 = 15 perceptrons

Width of a deep MLP



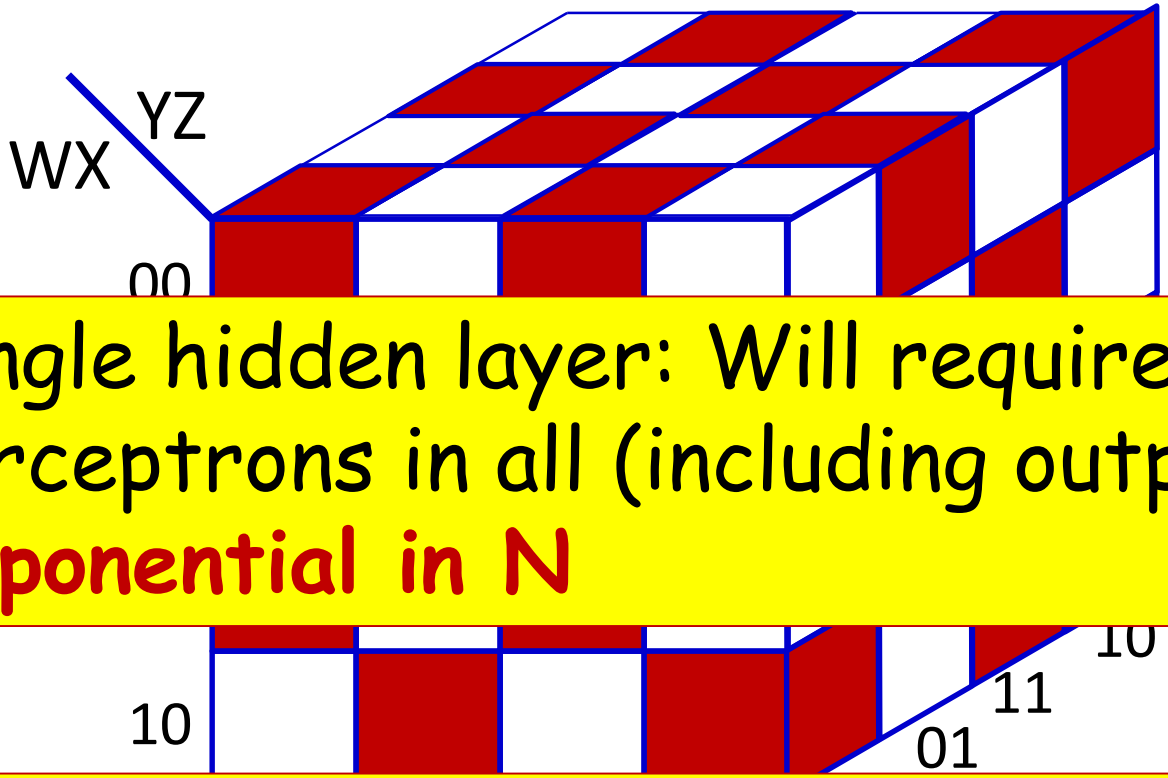


$$O = U \oplus V \oplus W \oplus X \oplus Y \oplus Z$$

More generally, the XOR of N variables will require 3(N-1) perceptrons!!

- An XOR needs 3 perceptrons
- This network will require 3x5 = 15 perceptrons

Width of a single-layer Boolean MLP



Single hidden layer: Will require $2^{N-1}+1$ perceptrons in all (including output unit)

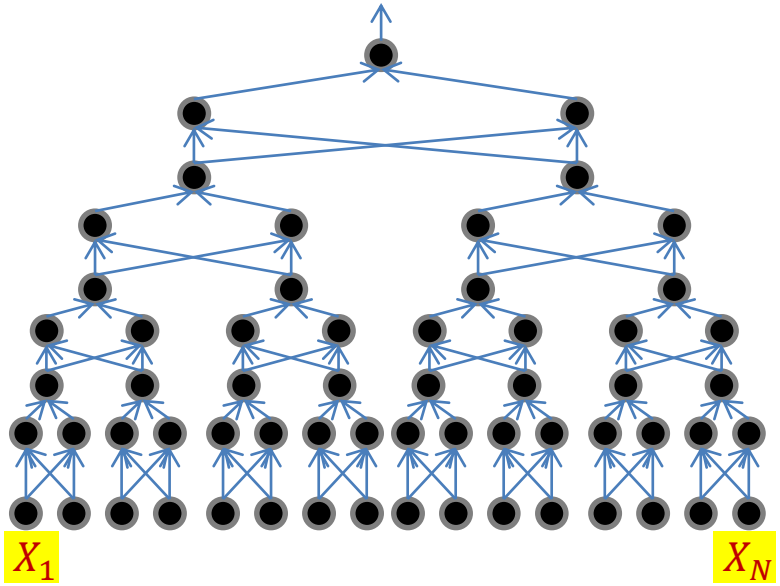
Exponential in N

Will require 3(N-1) perceptrons in a deep network

Linear in N!!!

Can be arranged in only $2\log_2(N)$ layers

A better representation

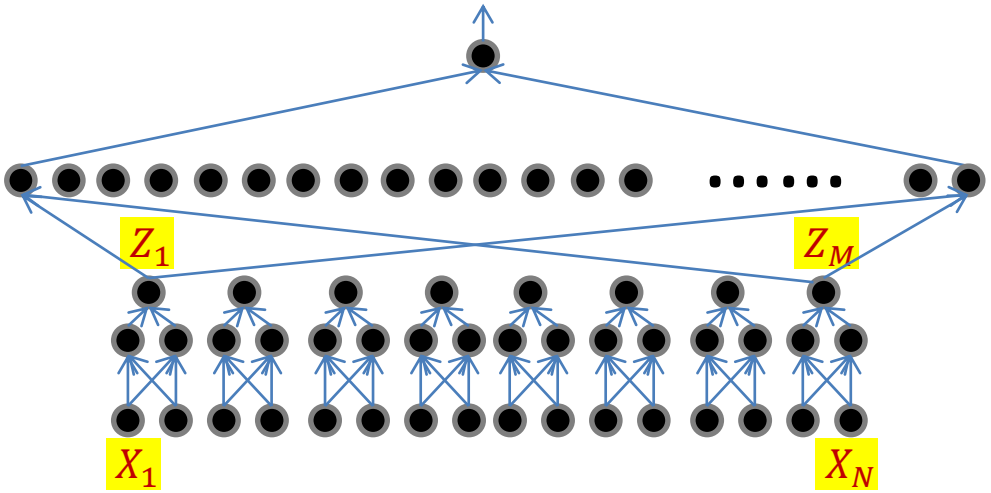


$$O = X_1 \oplus X_2 \oplus \cdots \oplus X_N$$

- Only $2 \log_2 N$ layers
  - By pairing terms
  - 2 layers per XOR

$$O = ((((((X_1 \oplus X_2) \oplus (X_1 \oplus X_2)) \oplus ((X_5 \oplus X_6) \oplus (X_7 \oplus X_8))) \oplus (((...$$

The challenge of depth



$$O = X_1 \oplus X_2 \oplus \cdots \oplus X_N$$

$$= Z_1 \oplus Z_2 \oplus \cdots \oplus Z_M$$

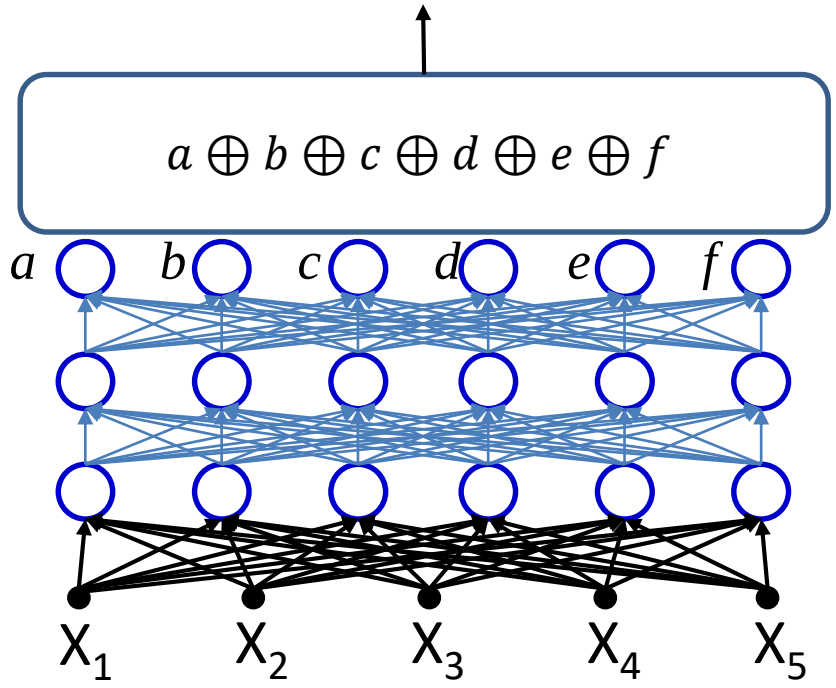
- Using only K hidden layers will require $O(2^{(N-K/2)})$ neurons in the Kth layer
  - Because the output can be shown to be the XOR of all the outputs of the K-1th hidden layer
  - **I.e. reducing the number of layers below the minimum will result in an exponentially sized network to express the function fully**
  - **A network with fewer than the required number of neurons *cannot* model the function**

Recap: The need for depth

- *Deep* Boolean MLPs that scale *linearly* with the number of inputs …
- … can become exponentially large if recast using only one layer

- It gets worse..

The need for depth



- The wide function can happen at any layer
- Having a few extra layers can greatly reduce network size

Network size: summary

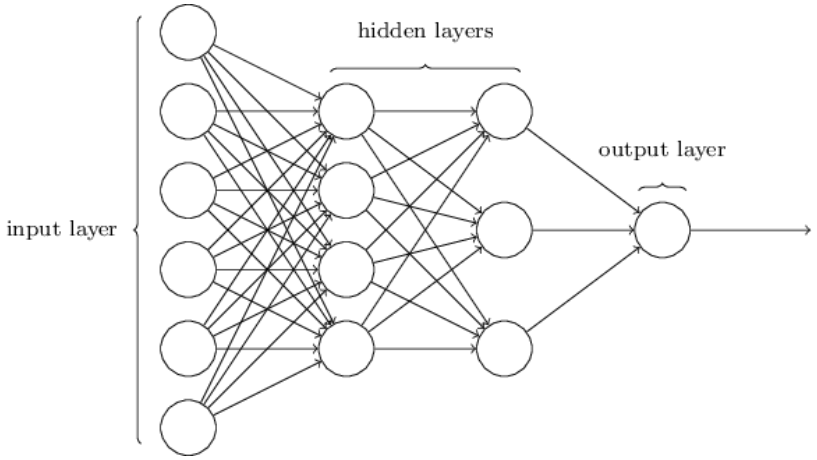
- An MLP is a universal Boolean function
- But can represent a given function only if
  - It is sufficiently wide
  - It is sufficiently deep
  - Depth can be traded off for (sometimes) exponential growth of the width of the network
- Optimal width and depth depend on the number of variables and the complexity of the Boolean function
  - Complexity: minimal number of terms in DNF formula to represent it

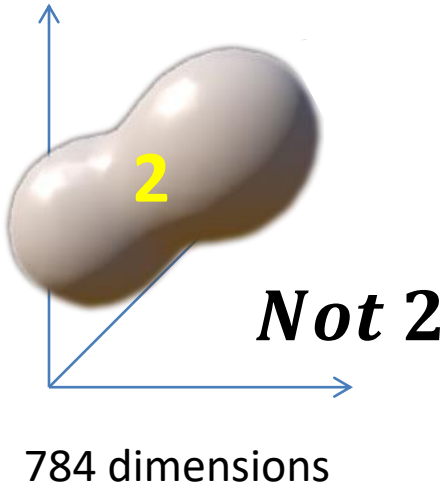
Story so far

- Multi-layer perceptrons are *Universal Boolean Machines*
- Even a network with a *single* hidden layer is a universal Boolean machine
  - But a single-layer network may require an exponentially large number of perceptrons
- Deeper networks may require far fewer neurons than shallower networks to express the same function
  - Could be *exponentially* smaller

The MLP as a classifier

784 dimensions
(MNIST)

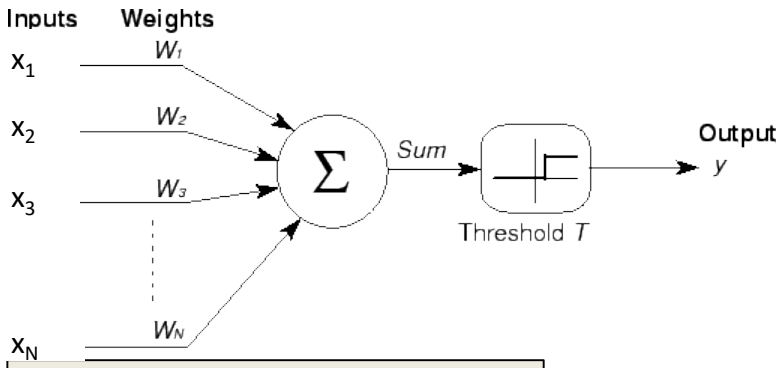




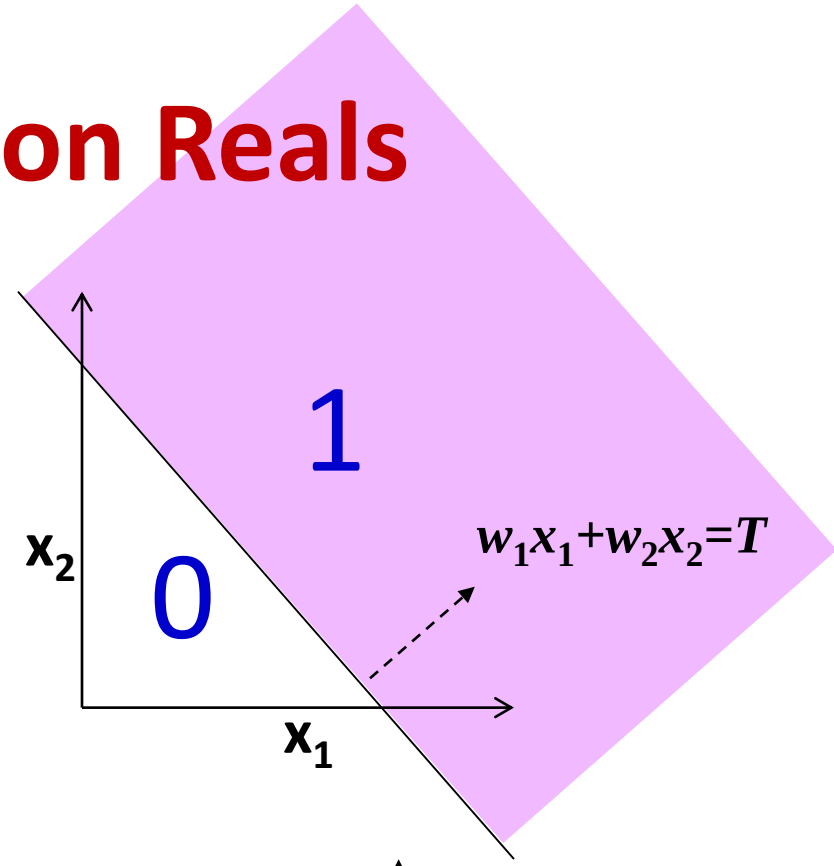
784 dimensions

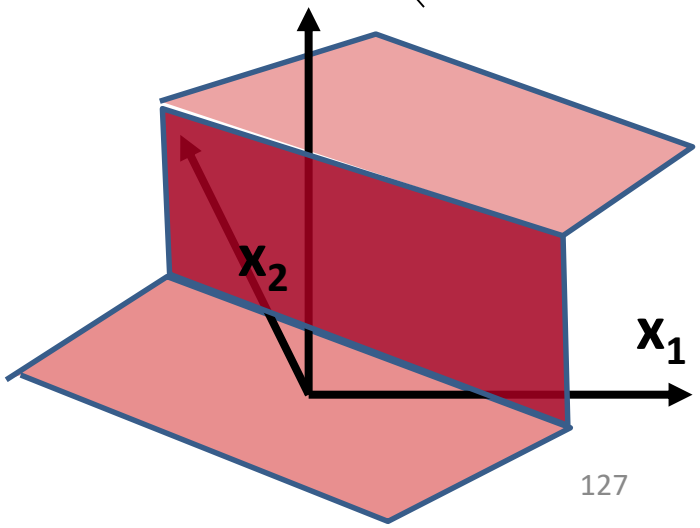
- MLP as a function over real inputs
- MLP as a function that finds a complex "decision boundary" over a space of *reals*

A Perceptron on Reals



$$y = \begin{cases} 1 \text{ if } \sum_i w_i x_i \geq T \\ 0 \text{ else} \end{cases}$$

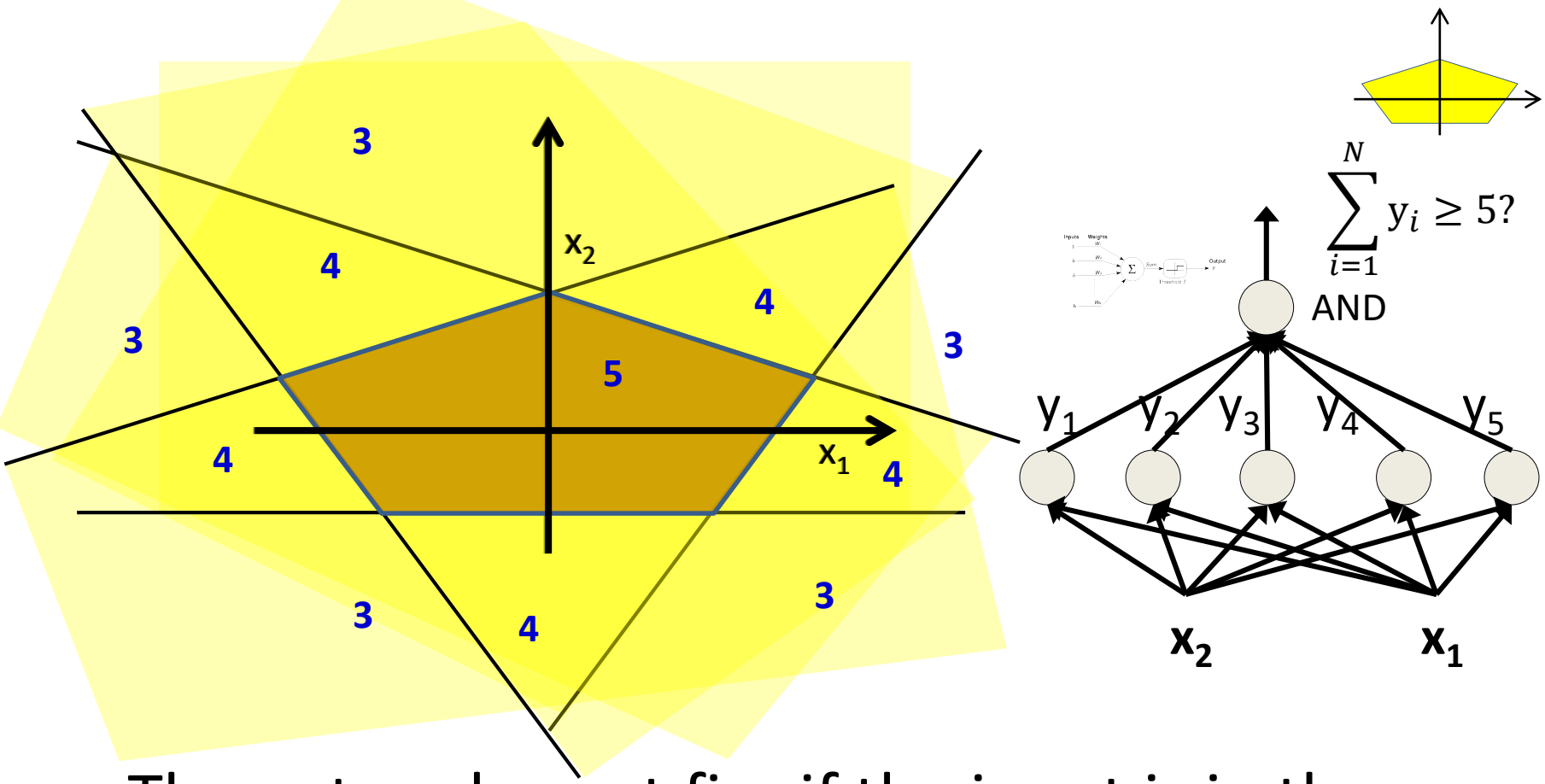




- A perceptron operates on *real*-valued vectors

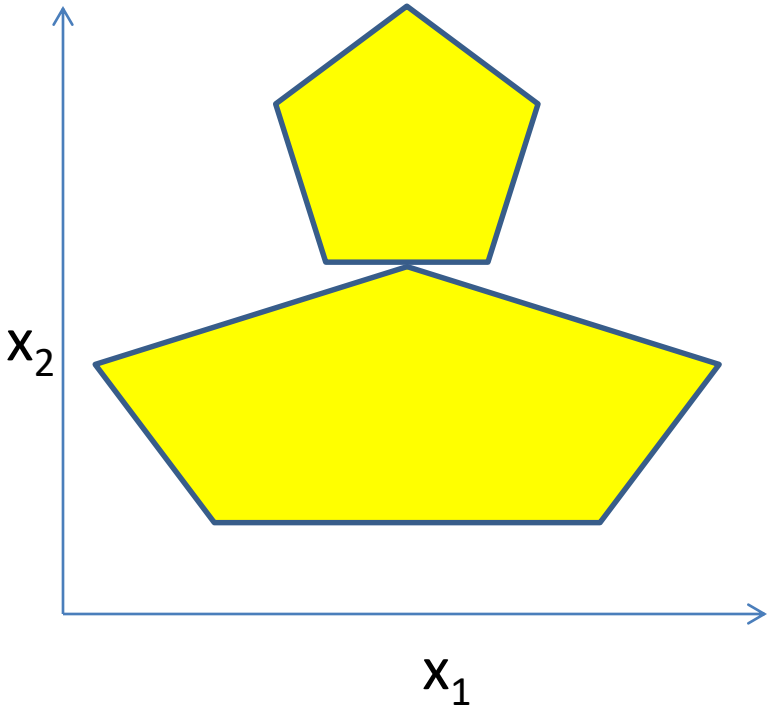
– This is a *linear classifier*

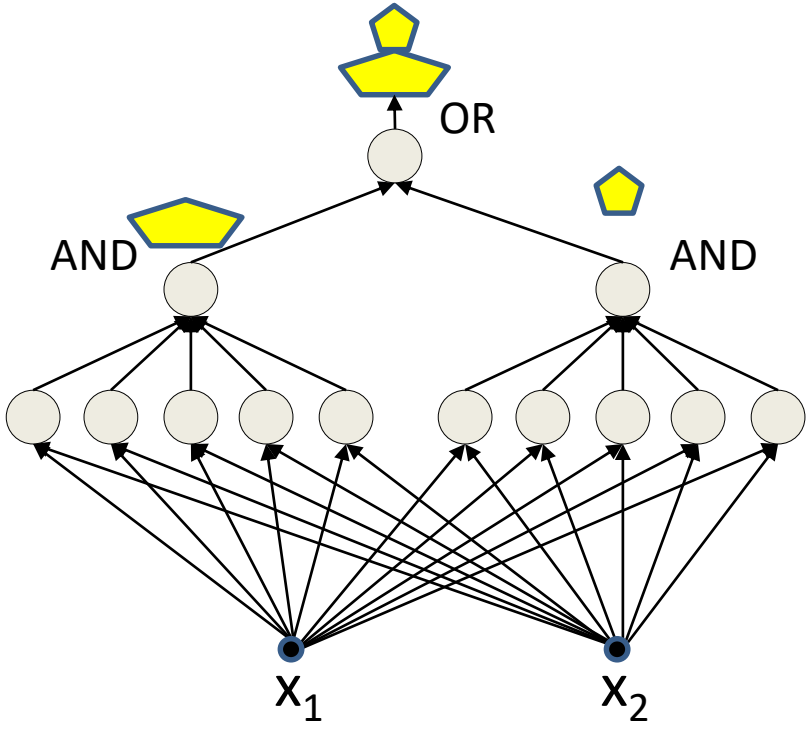
Booleans over the reals



- The network must fire if the input is in the coloured area

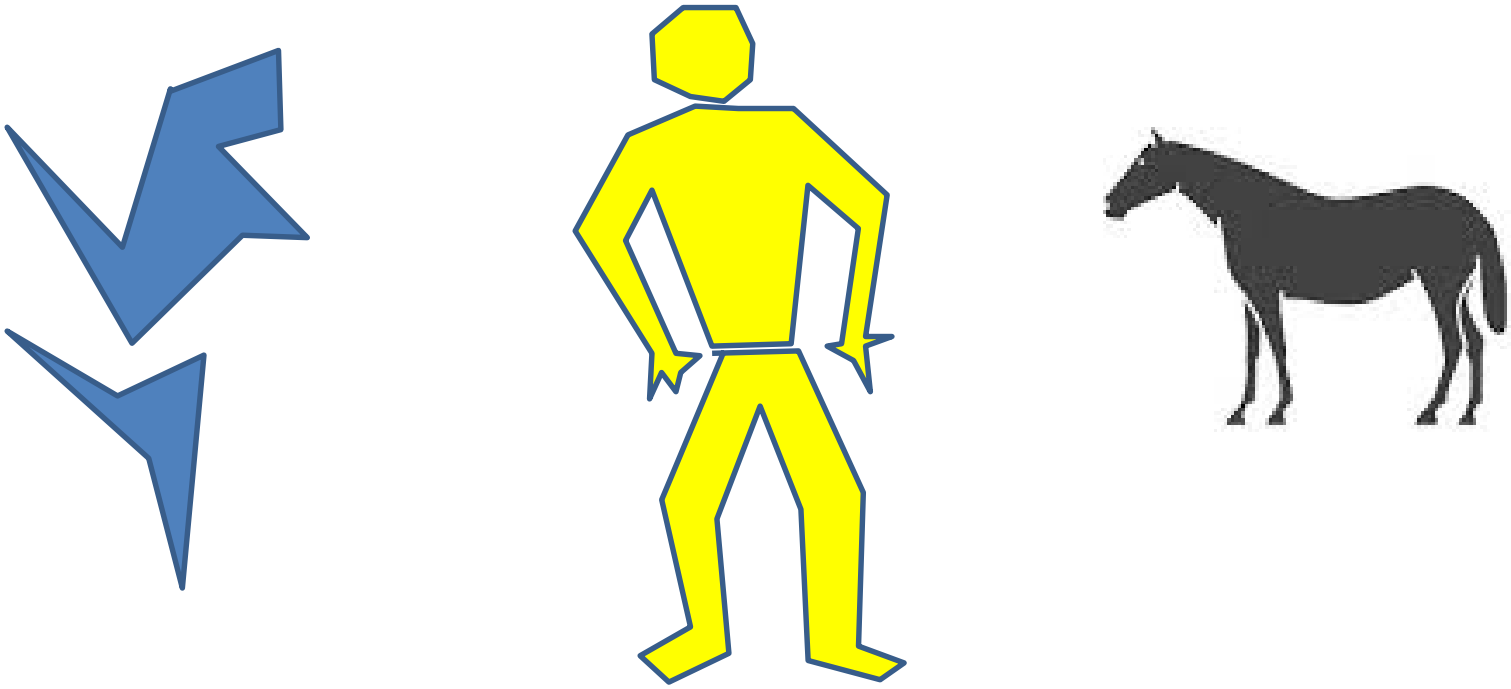
More complex decision boundaries





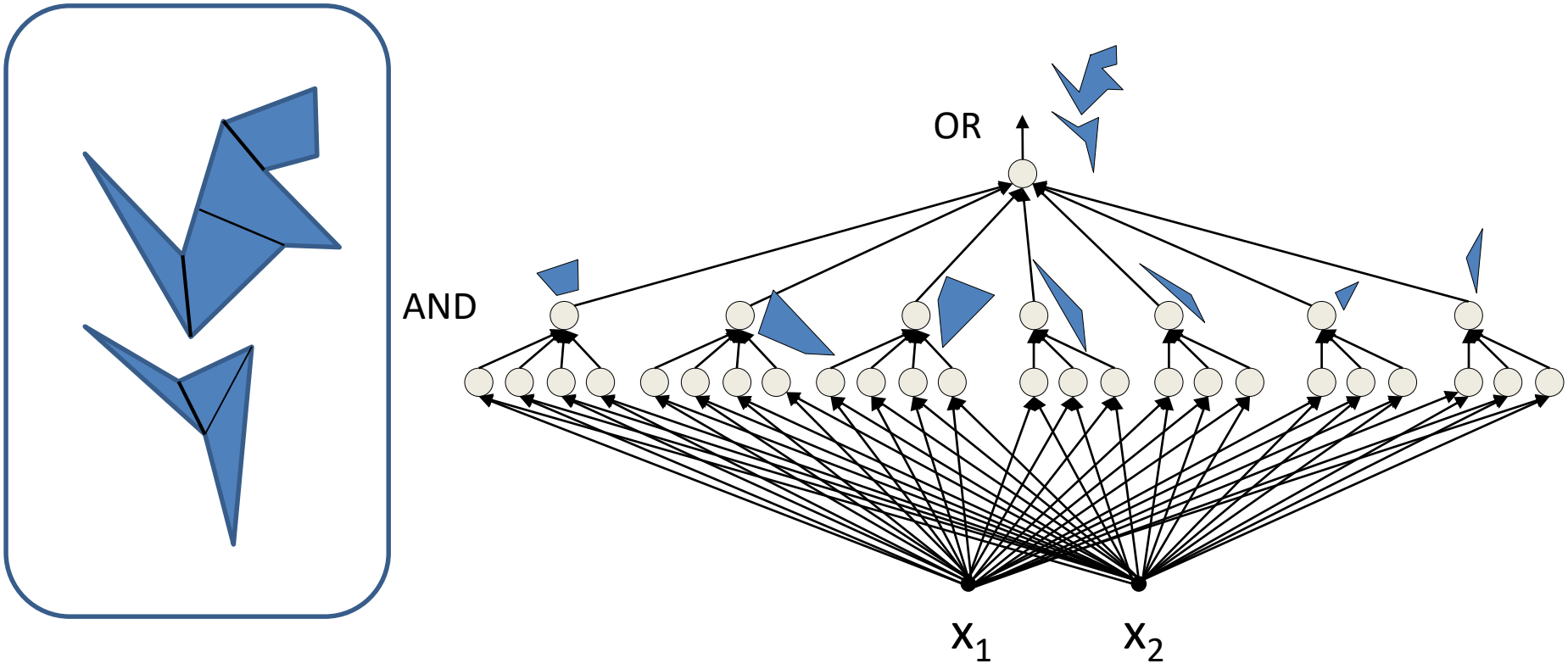
- Network to fire if the input is in the yellow area
  - "OR" two polygons
  - A third layer is required

Complex decision boundaries



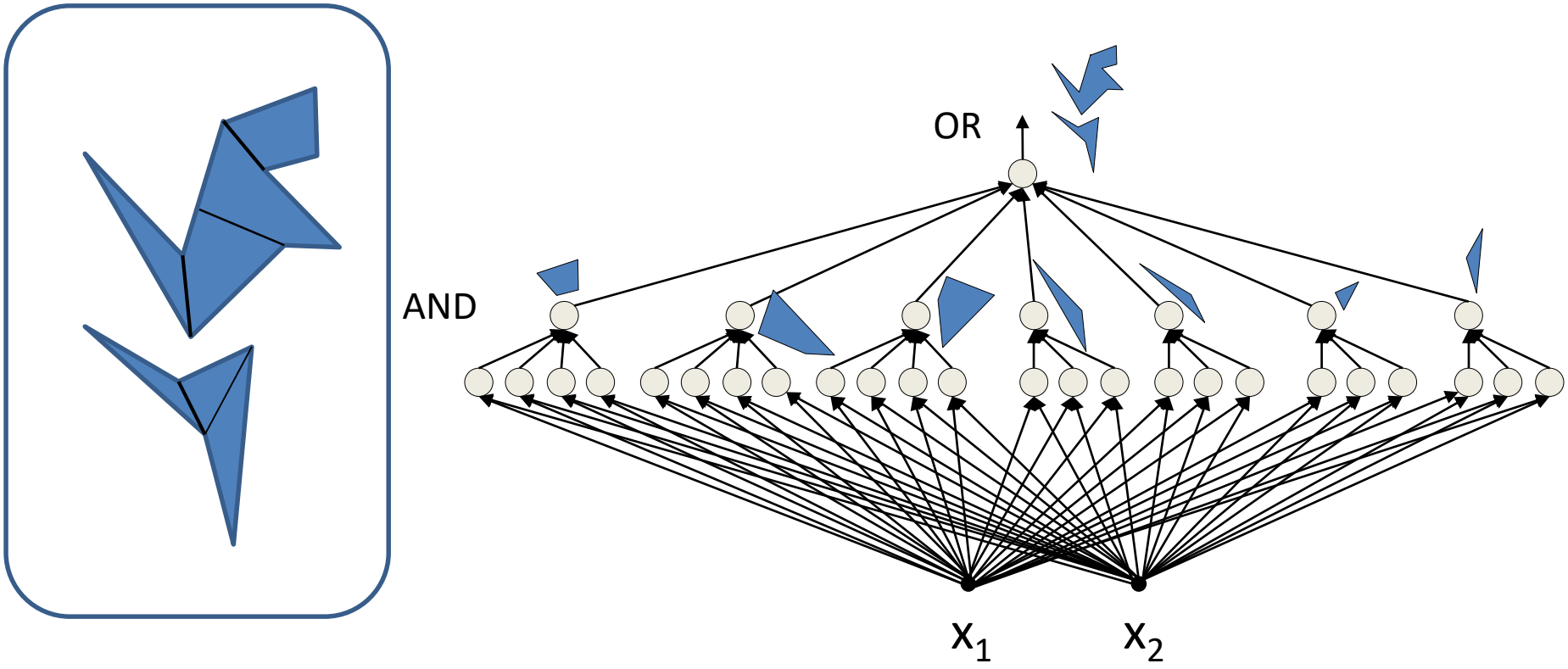
- Can compose *arbitrarily* complex decision boundaries

Complex decision boundaries



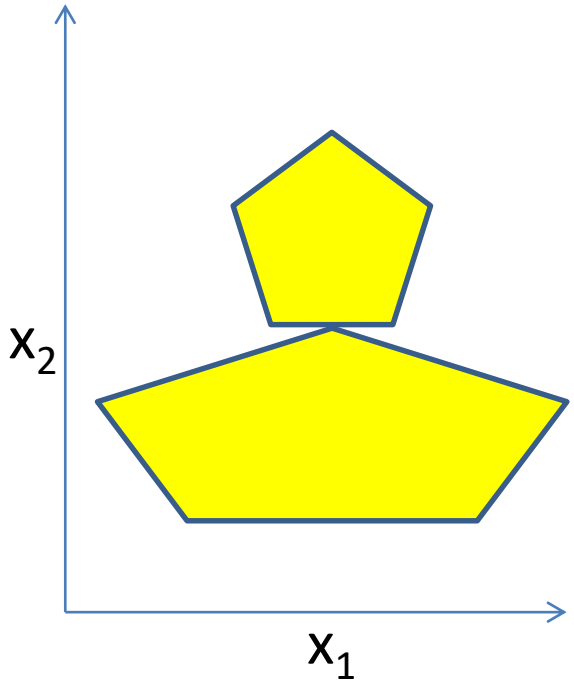
- Can compose *arbitrarily* complex decision boundaries

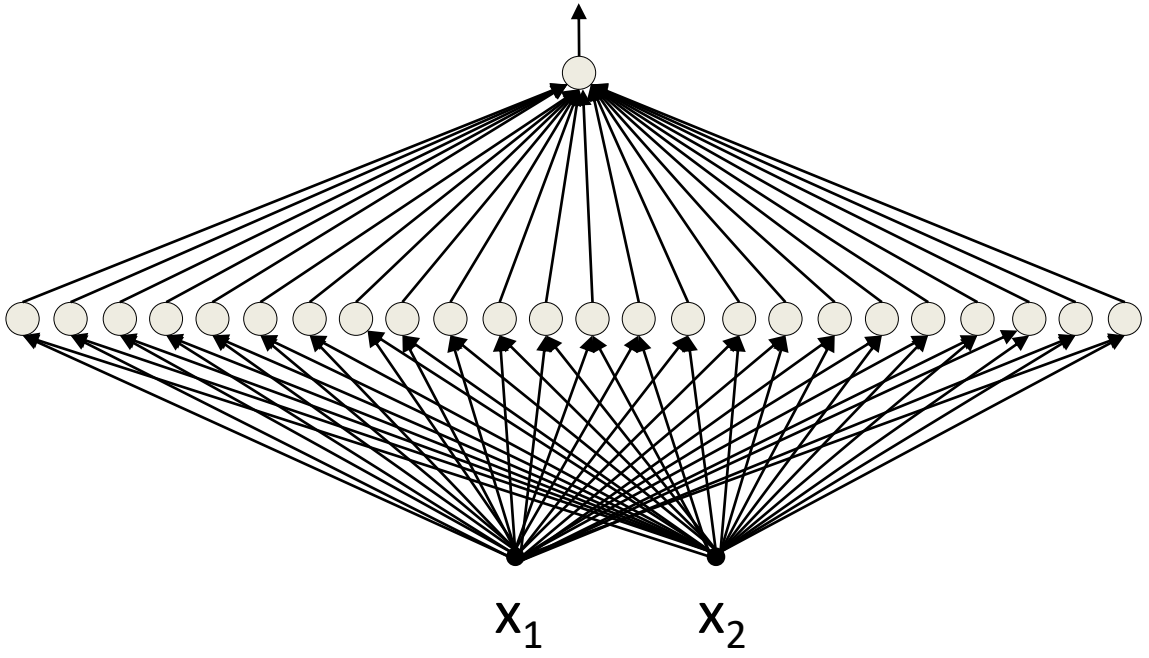
Complex decision boundaries



- Can compose *arbitrarily* complex decision boundaries
  - With *only one hidden layer!*
  - **How?**

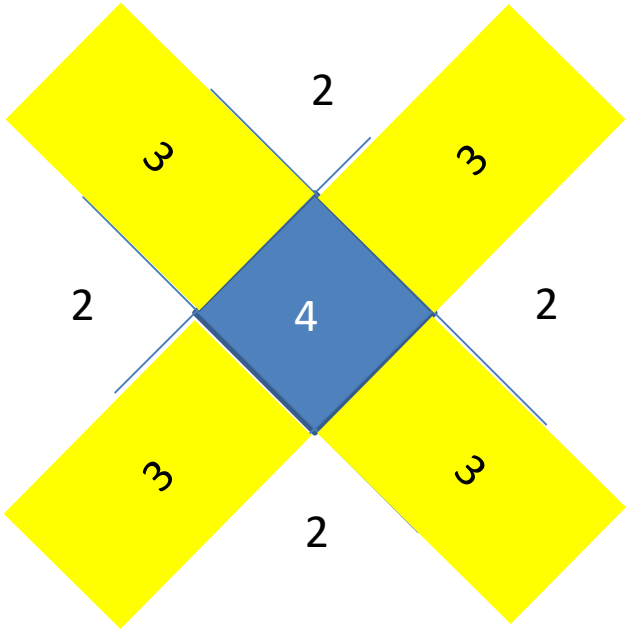
Exercise: compose this with one hidden layer

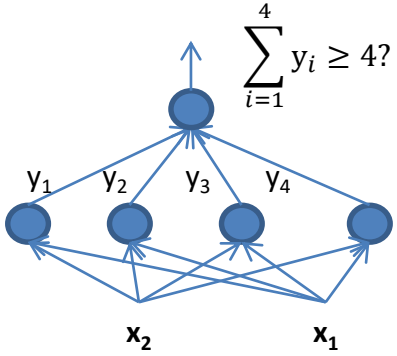




- How would you compose the decision boundary to the left with only *one* hidden layer?

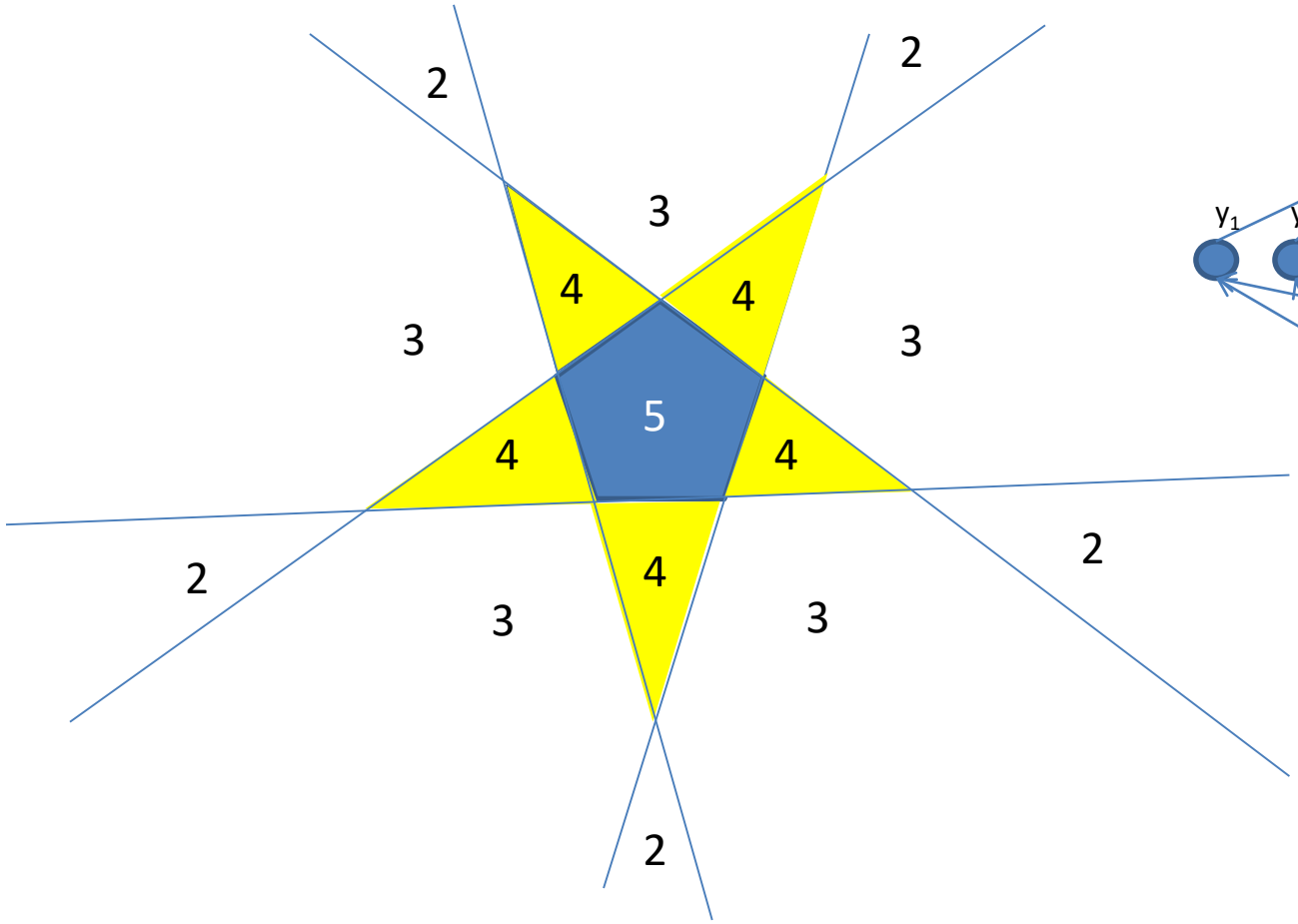
Composing a Square decision boundary

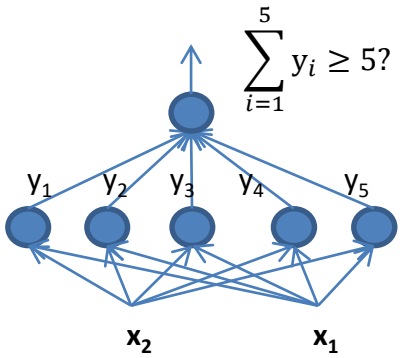




- The polygon net

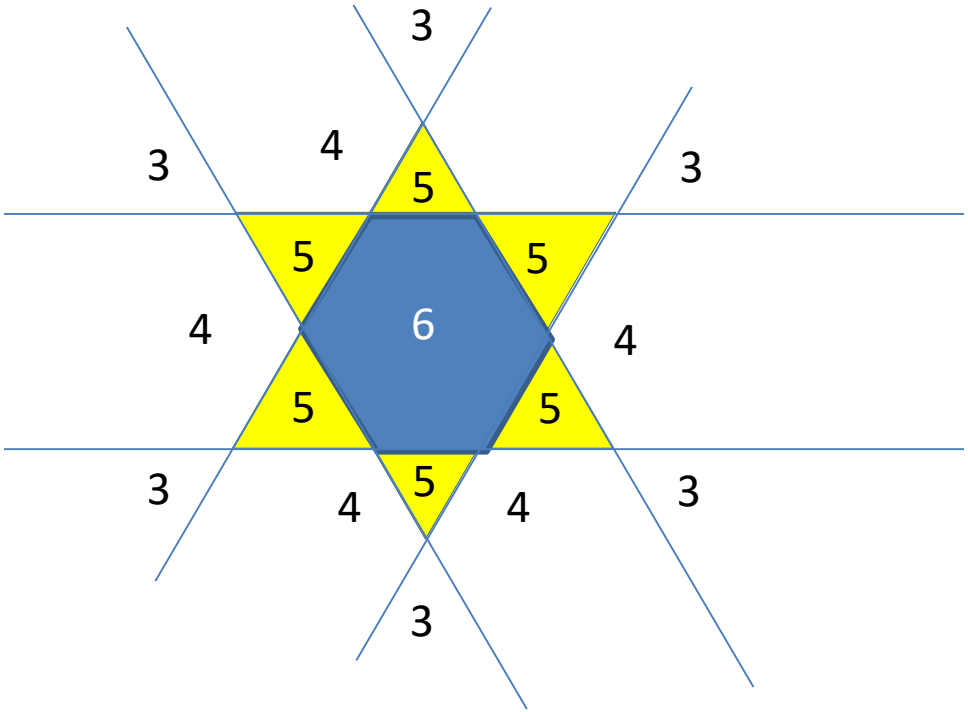
Composing a pentagon

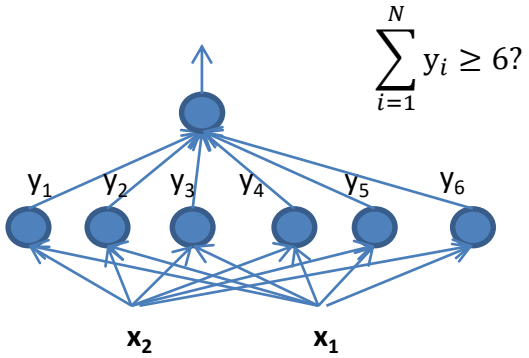




- The polygon net

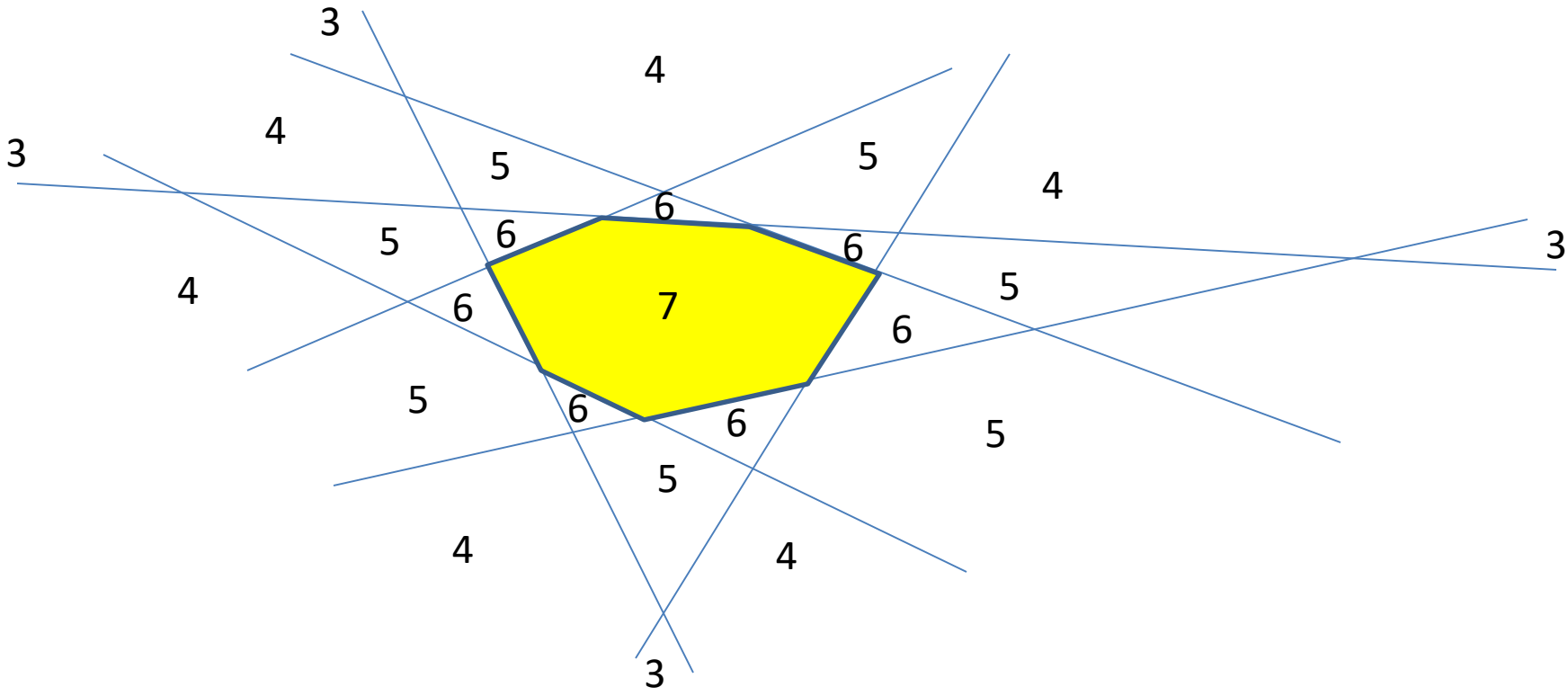
Composing a hexagon





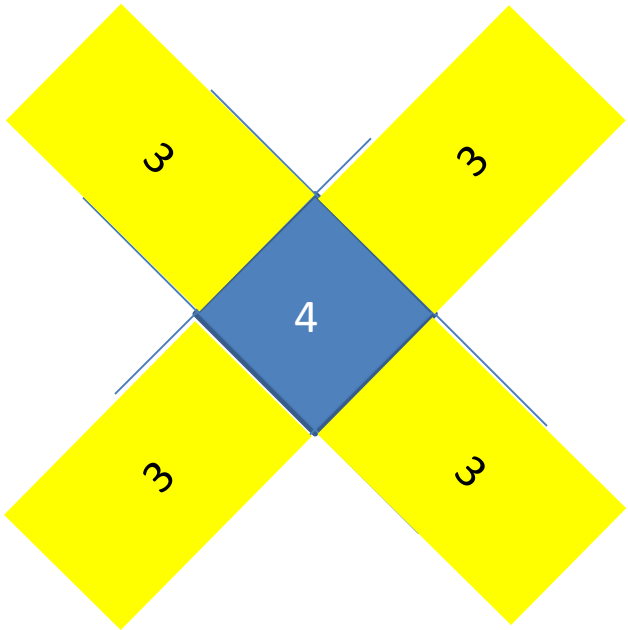
- The polygon net

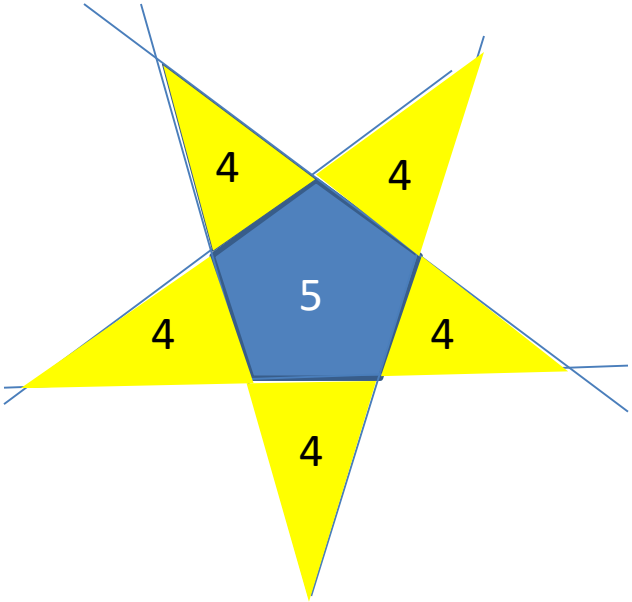
How about a heptagon

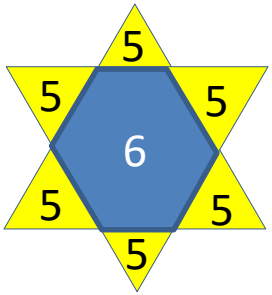


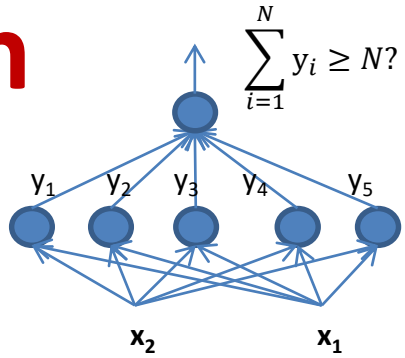
- What are the sums in the different regions?
  - A pattern emerges as we consider N > 6..

Composing a polygon



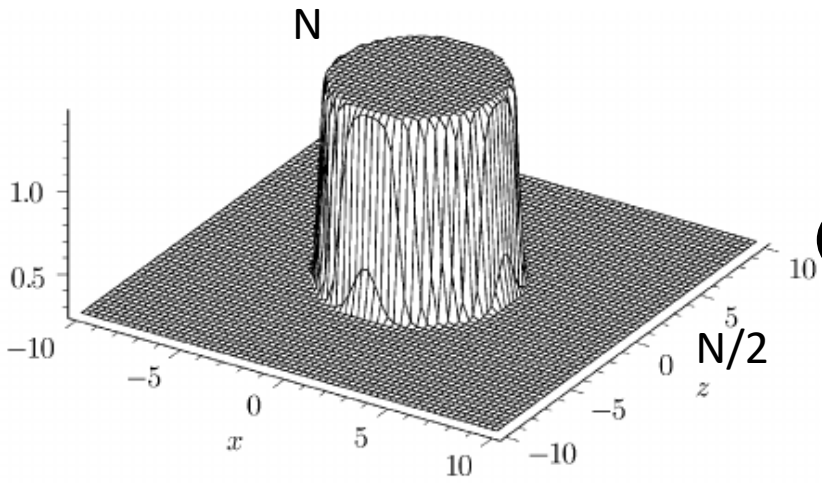


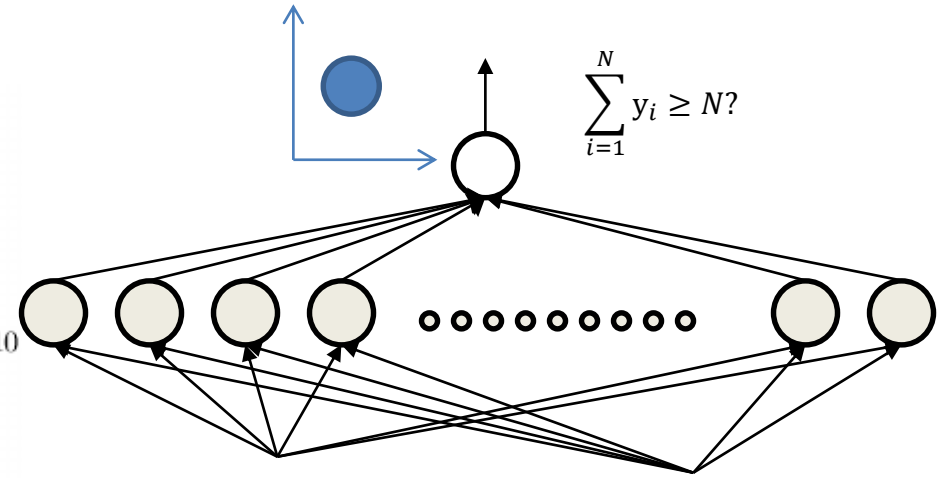




- The polygon net
- Increasing the number of sides reduces the area outside the polygon that have N/2 < Sum < N

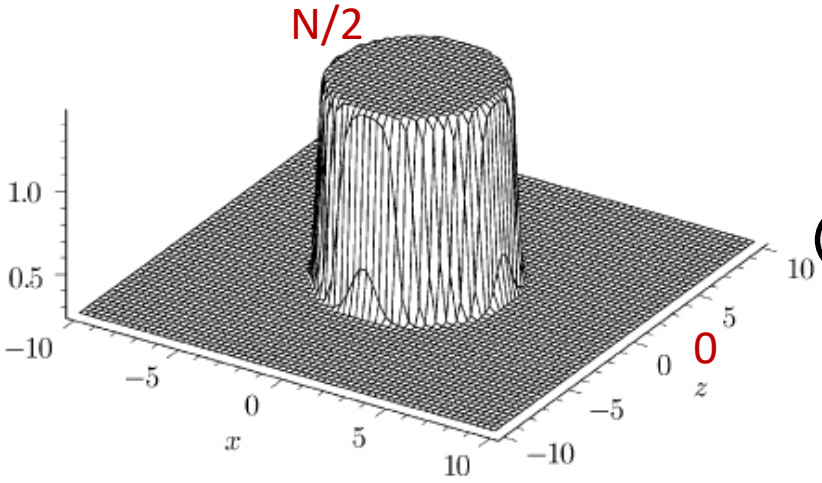
Composing a circle

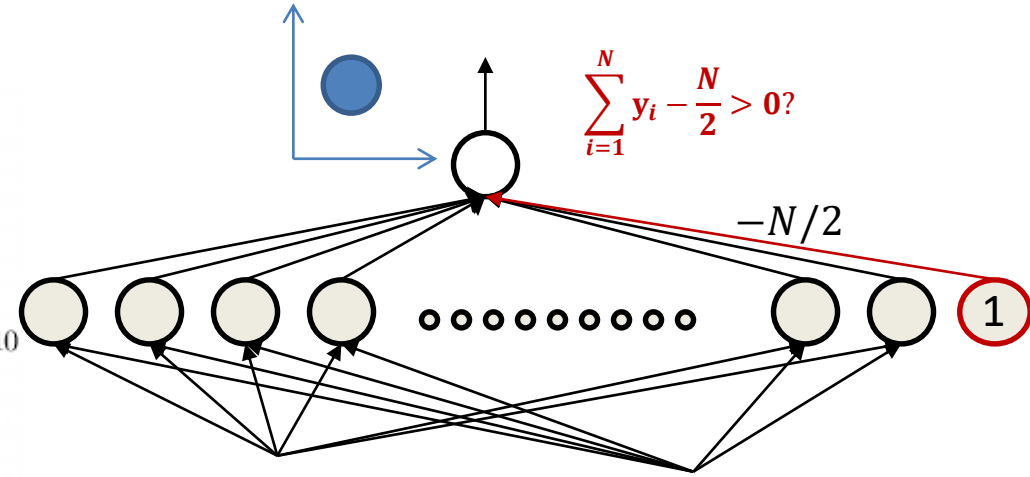




- The circle net
  - Very large number of neurons
  - *Sum is N inside the circle, N/2 outside everywhere*
  - Circle can be of arbitrary diameter, at any location

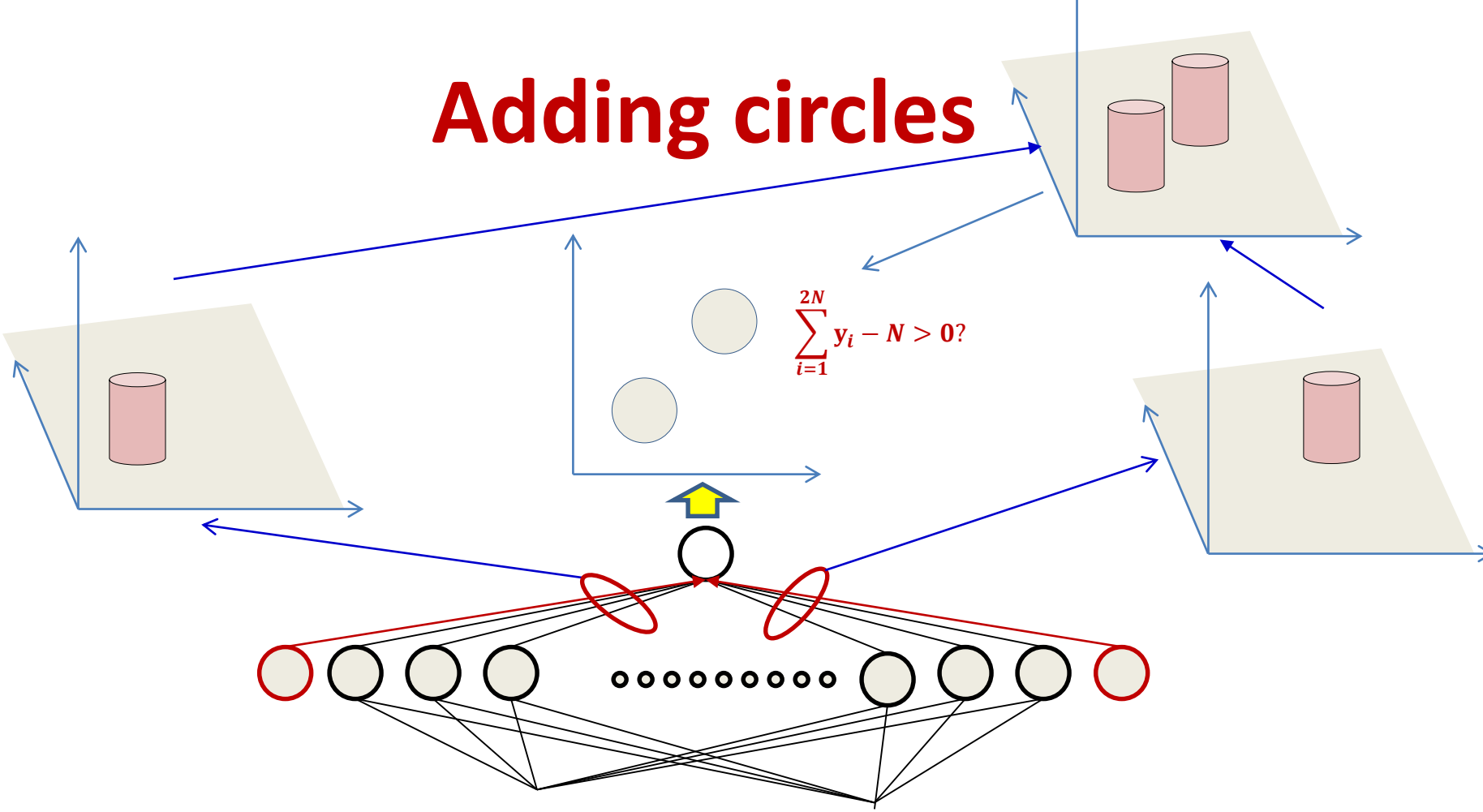
Composing a circle





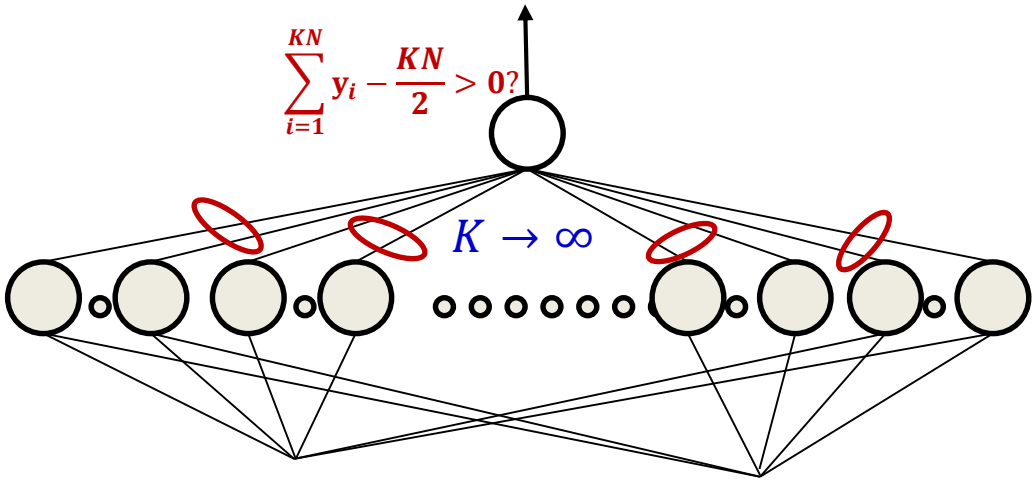
- The circle net
  - Very large number of neurons
  - *Sum is N/2 inside the circle, 0 outside everywhere*
  - Circle can be of arbitrary diameter, at any location

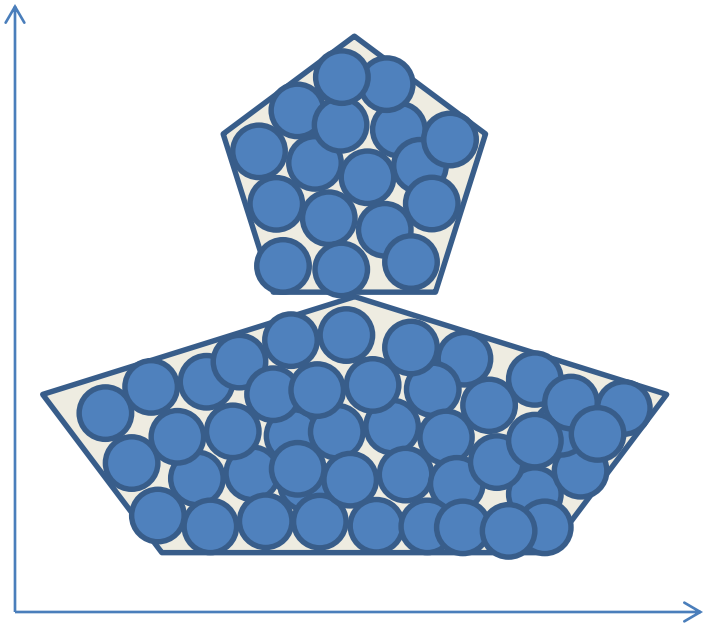
Adding circles



- The “sum” of two circles sub nets is exactly N/2 inside either circle, and 0 outside

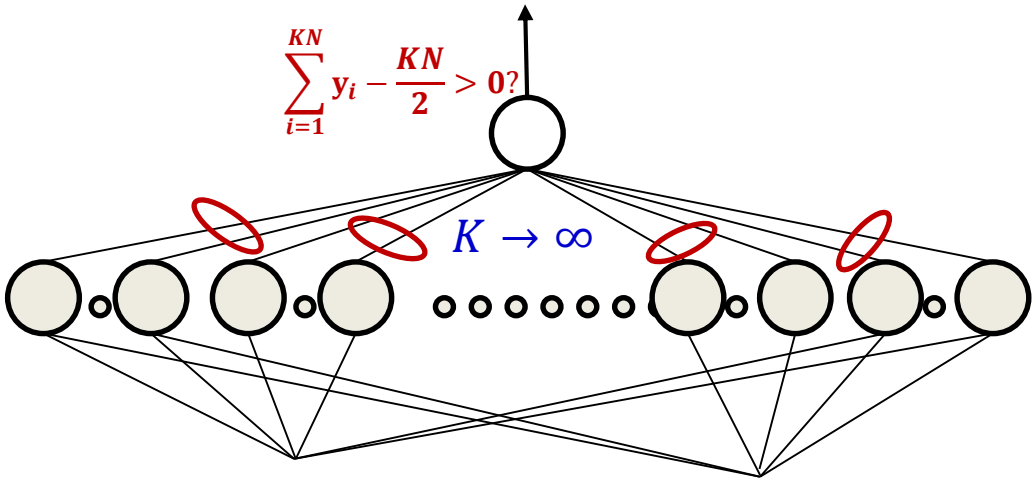
Composing an arbitrary figure

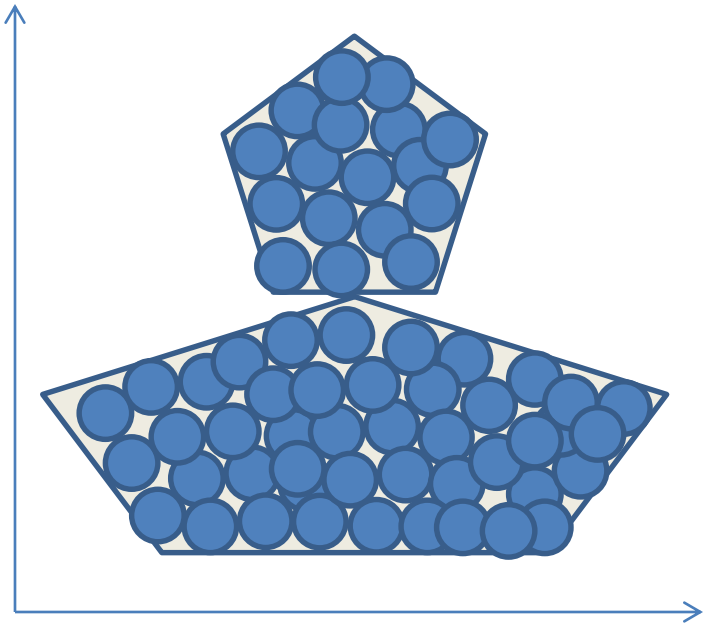




- Just fit in an arbitrary number of circles
  - More accurate approximation with greater number of smaller circles
  - Can achieve arbitrary precision

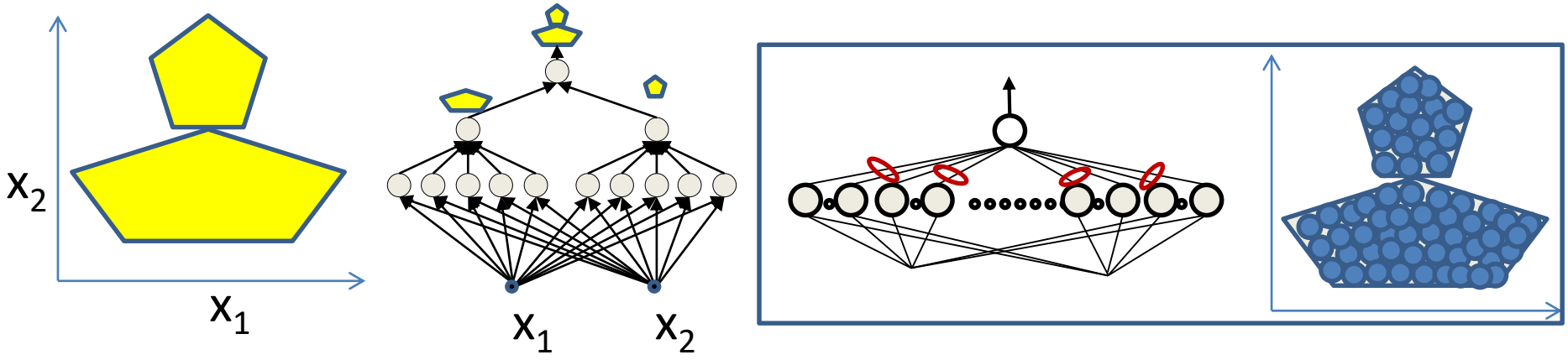
MLP: Universal classifier





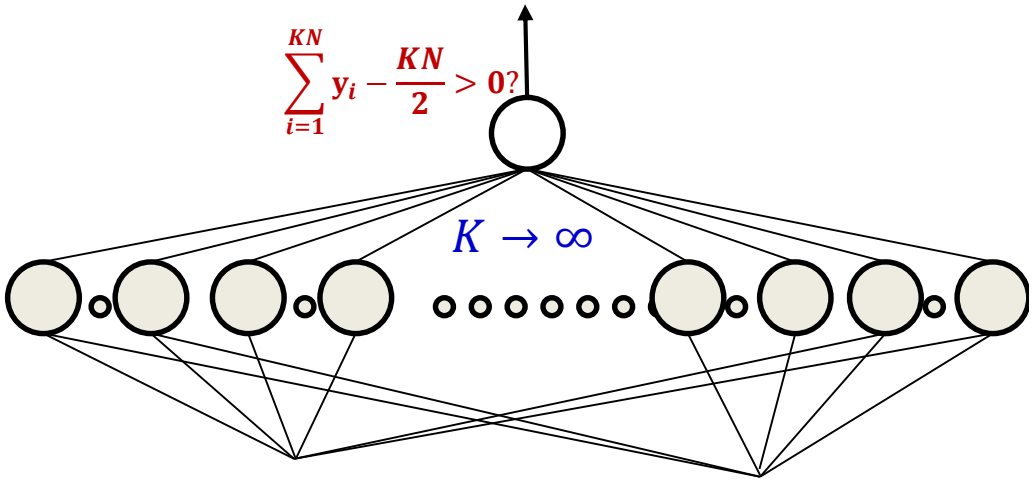
- MLPs can capture *any* classification boundary
- A *one-layer MLP* can model any classification boundary
- *MLPs are universal classifiers*

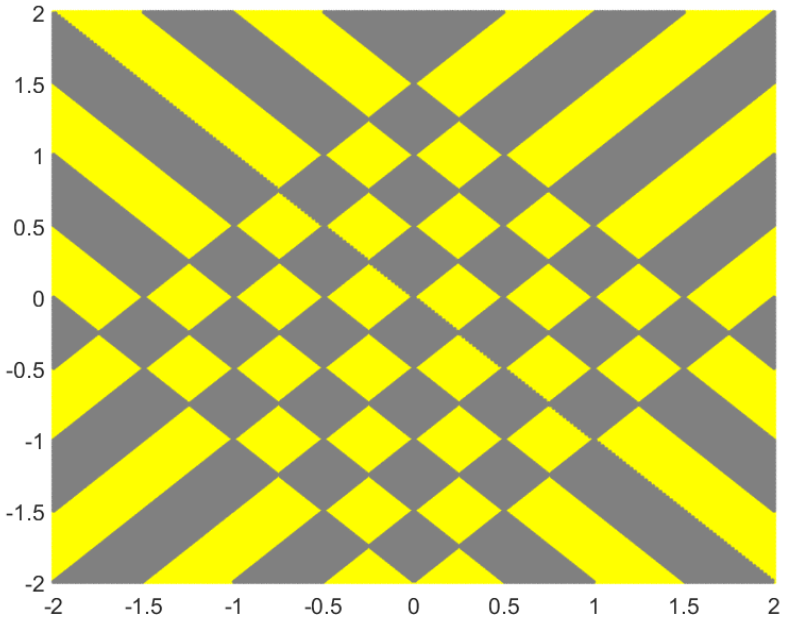
Depth and the universal classifier



- Deeper networks can require far fewer neurons

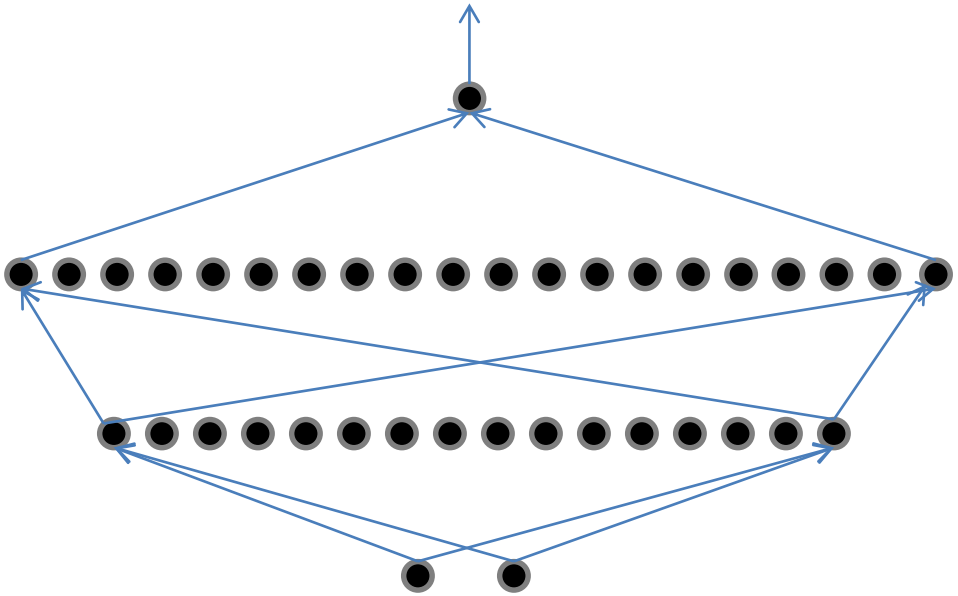
Optimal depth

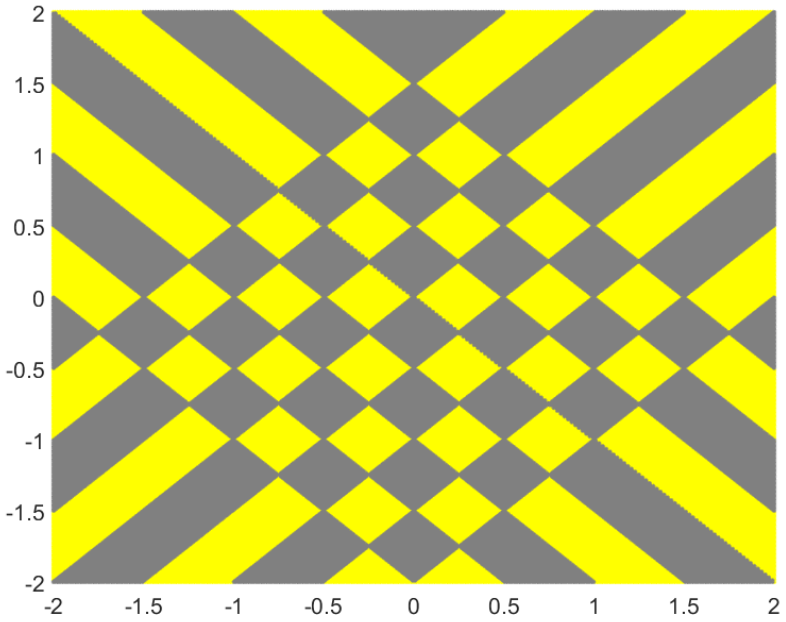




- A one-hidden-layer neural network will required infinite hidden neurons

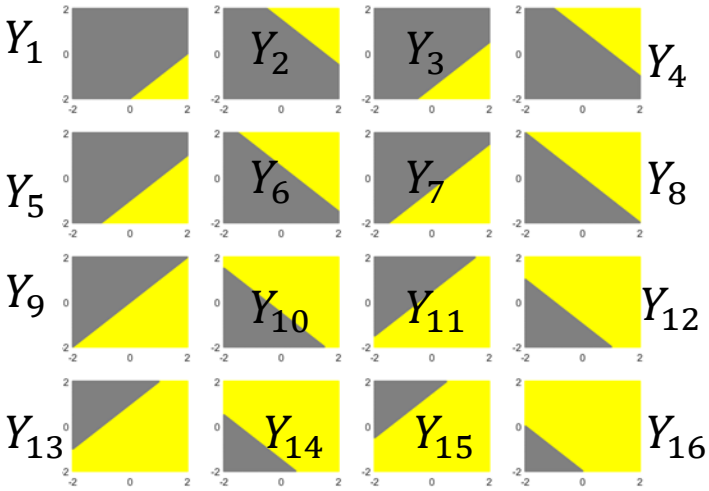
Optimal depth

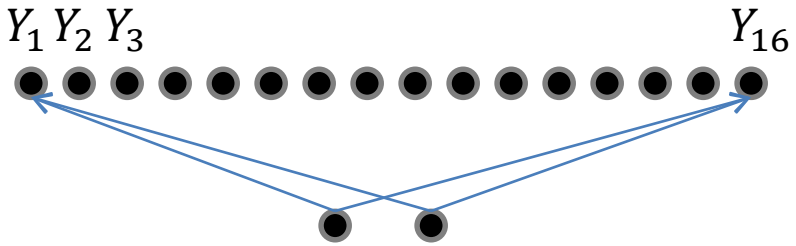


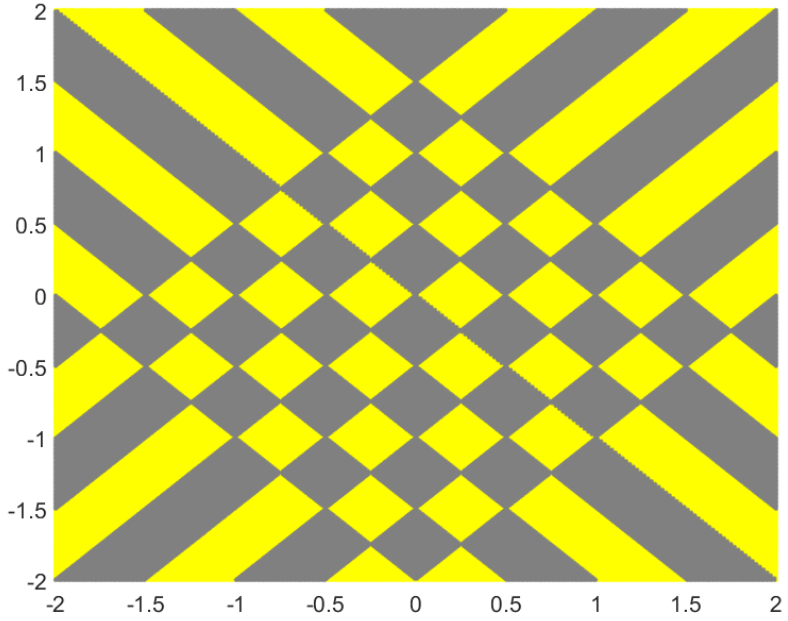


- Two layer network: 56 hidden neurons

Optimal depth

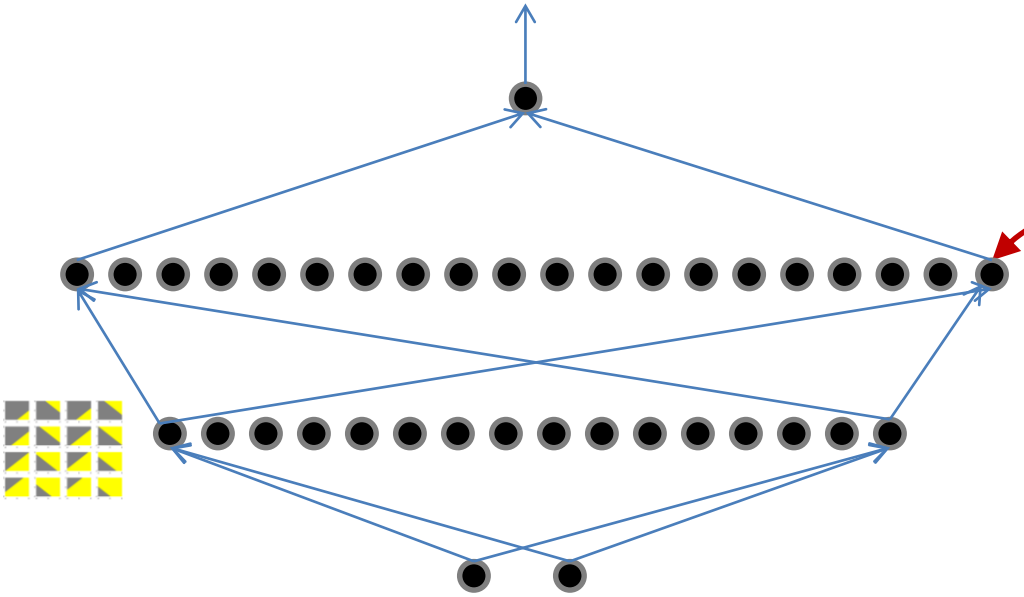


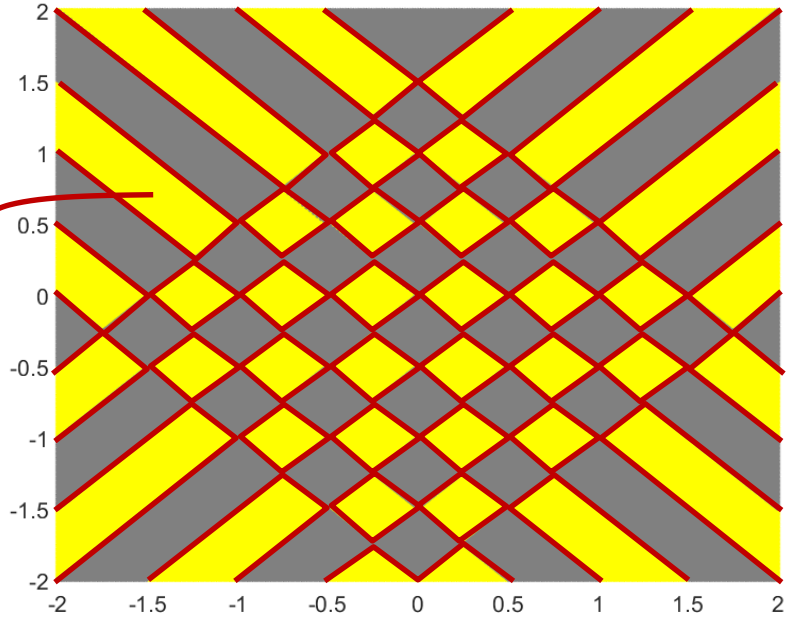




- Two layer network: 56 hidden neurons
  - 16 neurons in hidden layer 1

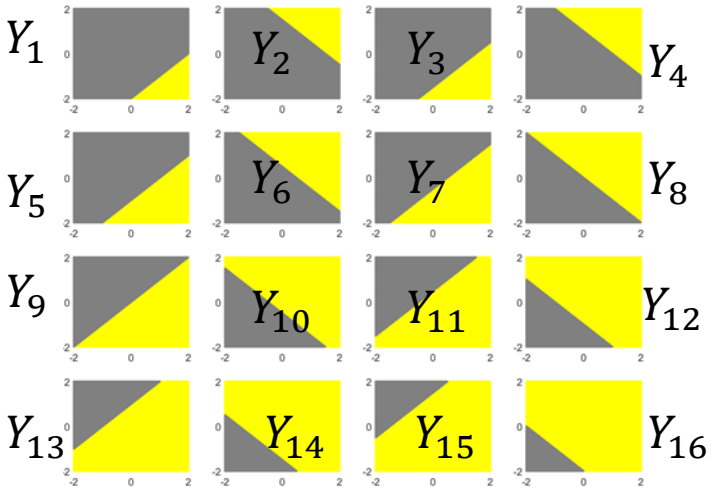
Optimal depth

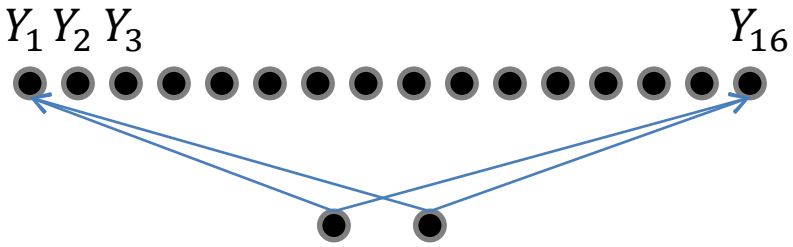


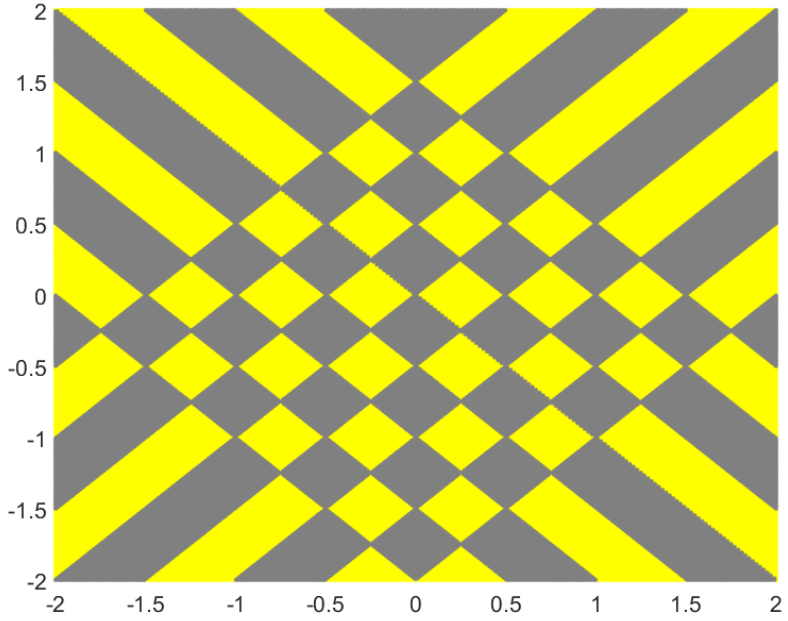


- Two-layer network: 56 hidden neurons
  - 16 in hidden layer 1
  - 40 in hidden layer 2
  - 57 total neurons, including output neuron

Optimal depth

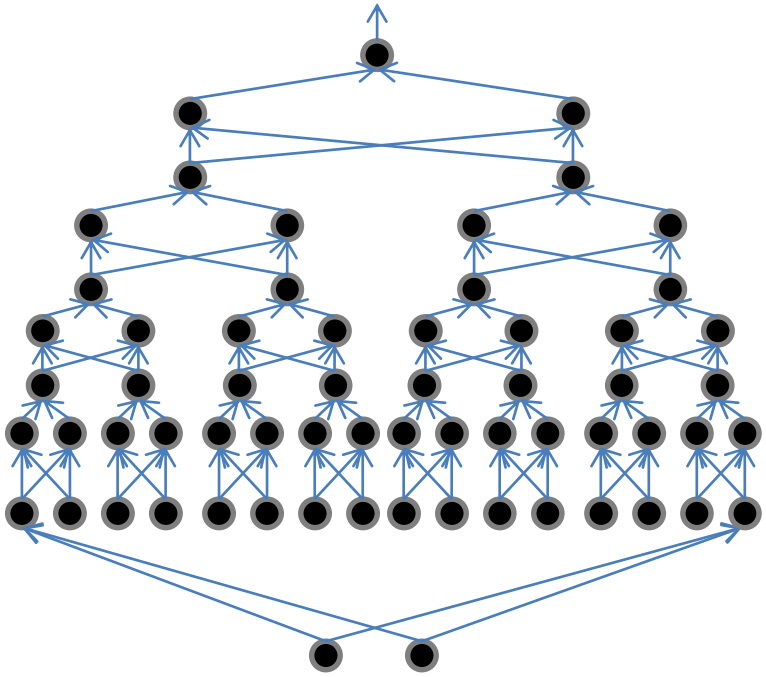


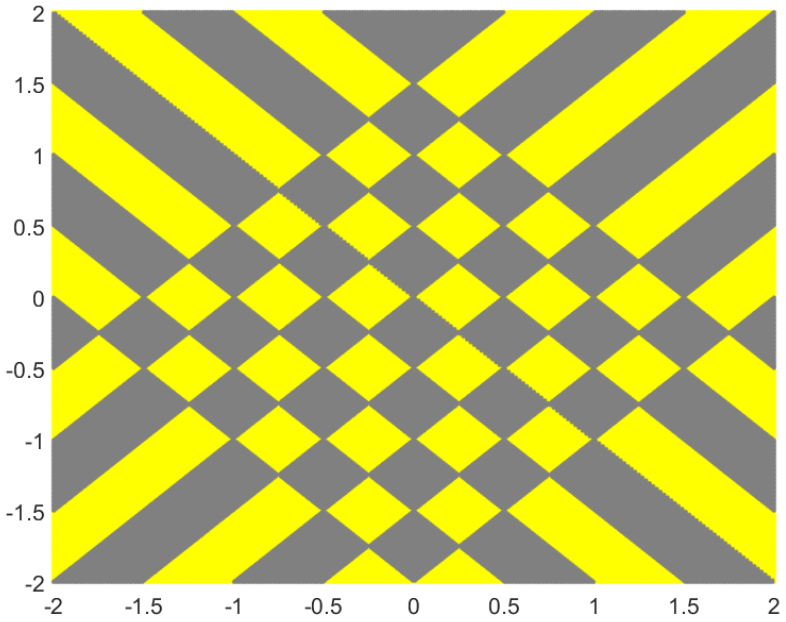




- But this is just $Y_1 \oplus Y_2 \oplus \cdots \oplus Y_{16}$

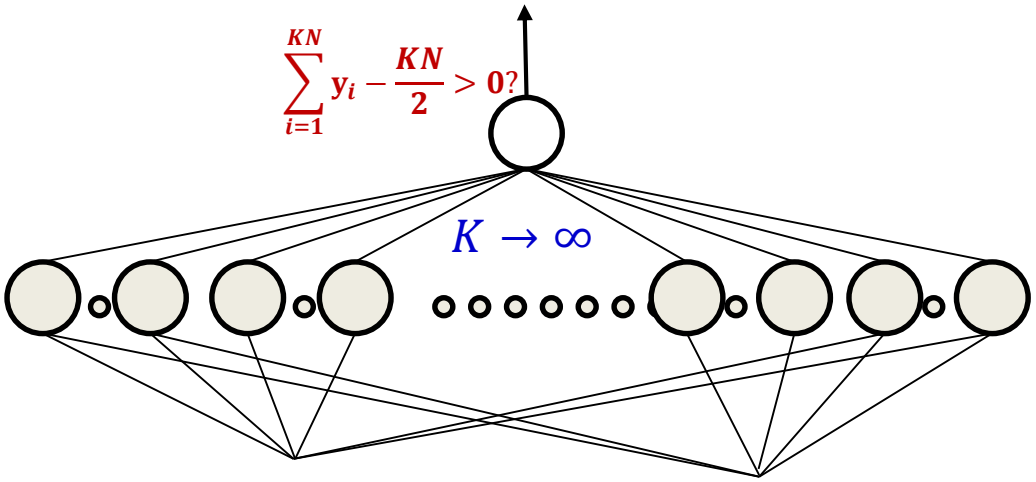
Optimal depth

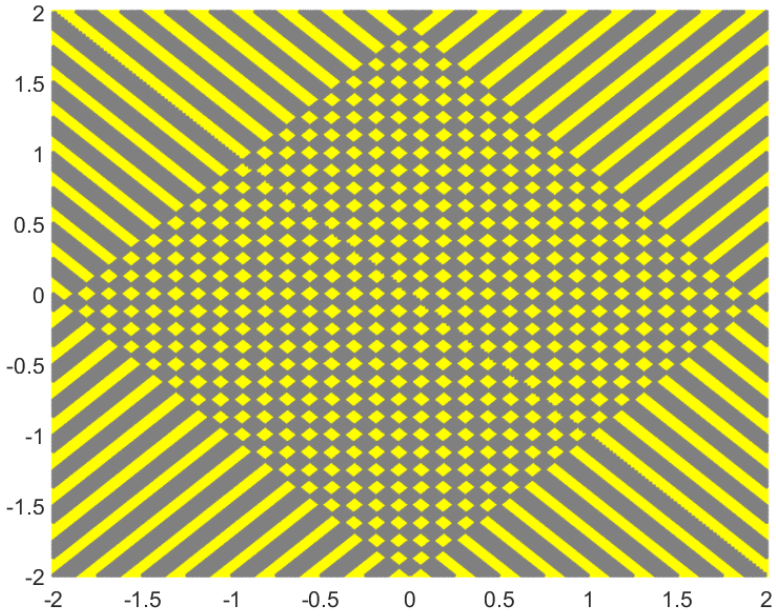




- But this is just $Y_1 \oplus Y_2 \oplus \cdots \oplus Y_{16}$
  - The XOR net will require 16 + 15x3 = 61 neurons
    - Greater than the 2-layer network with only 52 neurons

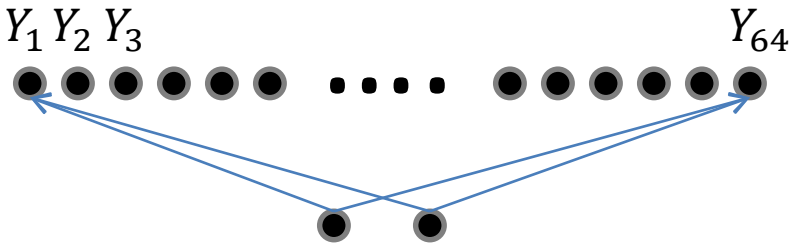
Optimal depth

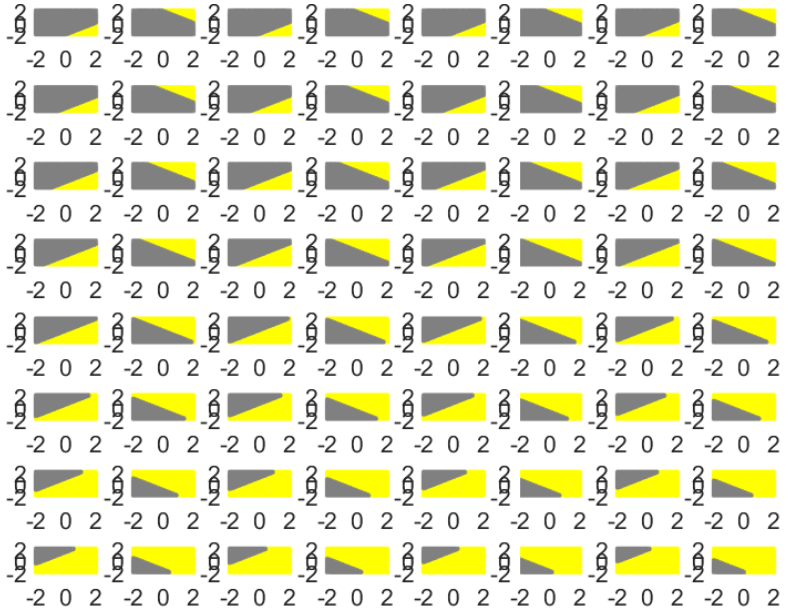




- A one-hidden-layer neural network will required infinite hidden neurons

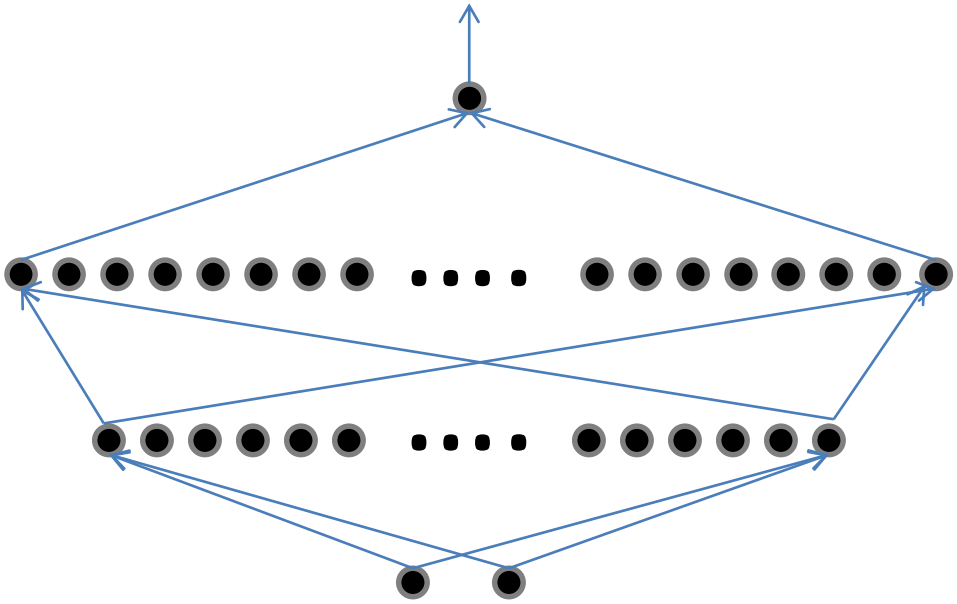
Actual linear units

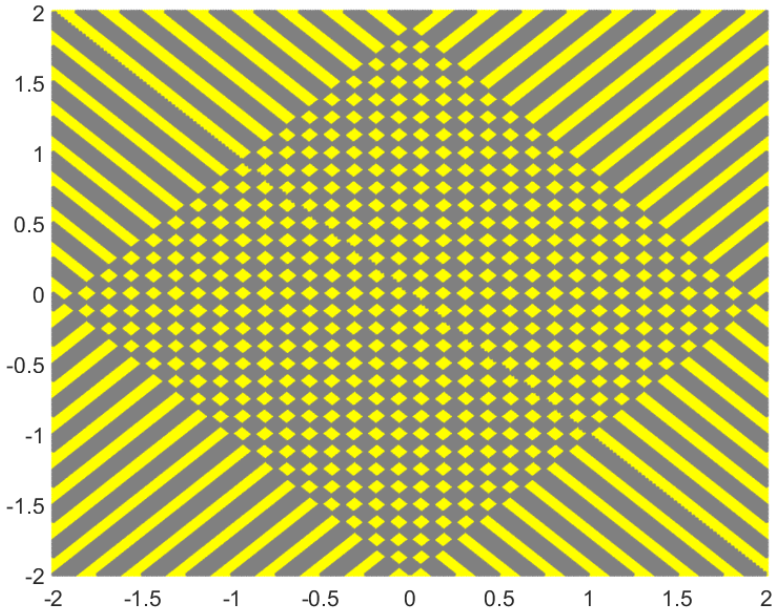




- 64 basic linear feature detectors

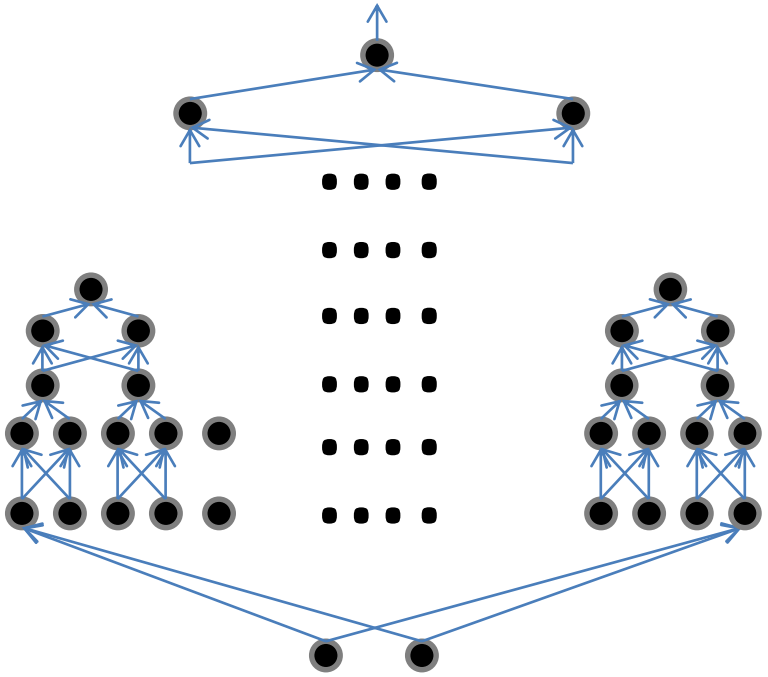
Optimal depth

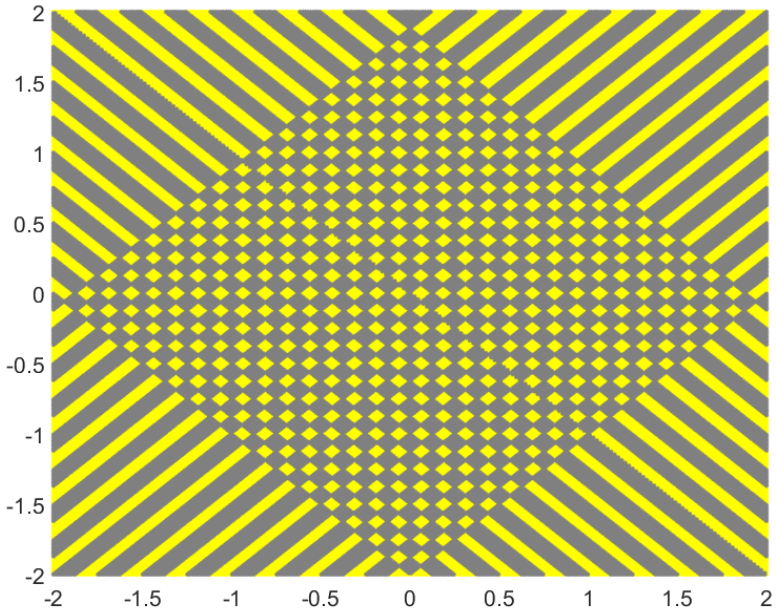




- Two hidden layers: 608 hidden neurons
  - 64 in layer 1
  - 544 in layer 2
- 609 total neurons (including output neuron)

Optimal depth

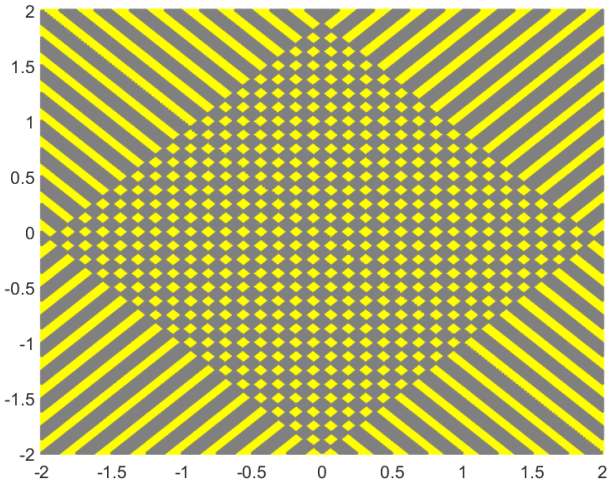




- XOR network (12 hidden layers): 253 neurons
- The difference in size between the deeper optimal (XOR) net and shallower nets increases with increasing pattern complexity

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!
  - The pattern is exponential in the *dimension of the input (two)!*
- For general case of N lines distributed over D dimensions, we will need up to $\frac{1}{2}\left(\frac{N}{D}+1\right)^D$
  - 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)

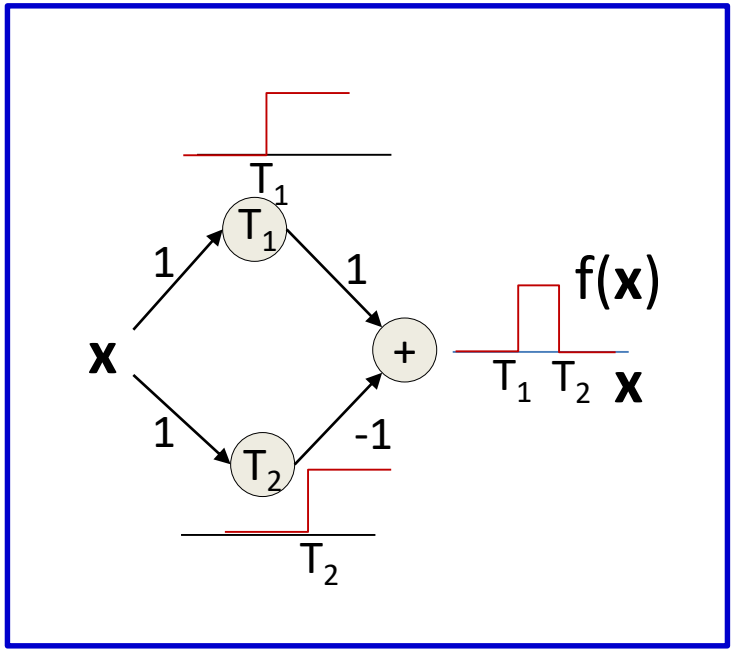
Depth: Summary

- The number of neurons required in a shallow network is
  - Polynomial in the number of basic patterns
  - Exponential in the dimensionality of the input
  - (this is the worst case)

Story so far

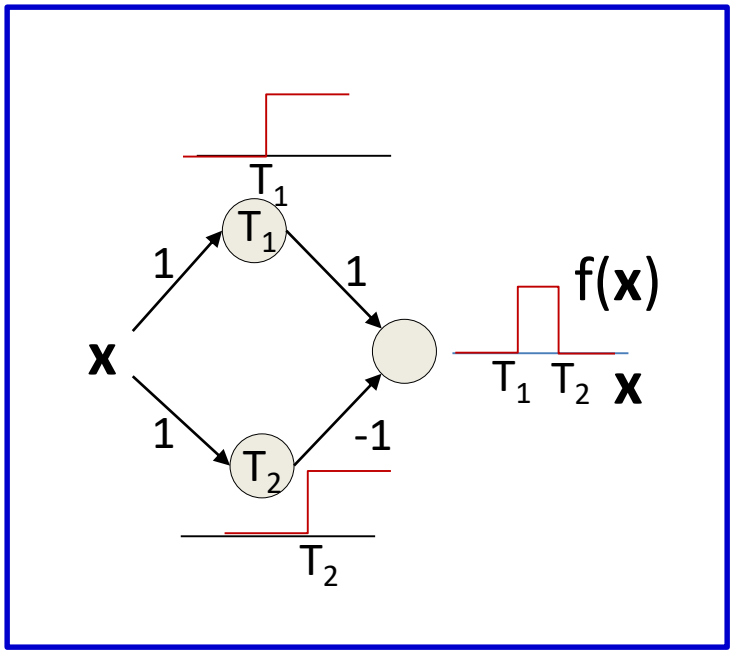
- Multi-layer perceptrons are *Universal Boolean Machines*
  - Even a network with a *single* hidden layer is a universal Boolean machine
- Multi-layer perceptrons are *Universal Classification Functions*
  - Even a network with a single hidden layer is a universal classifier
- But a single-layer network may require an exponentially large number of perceptrons than a deep one
- Deeper networks may require exponentially fewer neurons than shallower networks to express the same function
  - Could be *exponentially* smaller
  - Deeper networks are more *expressive*

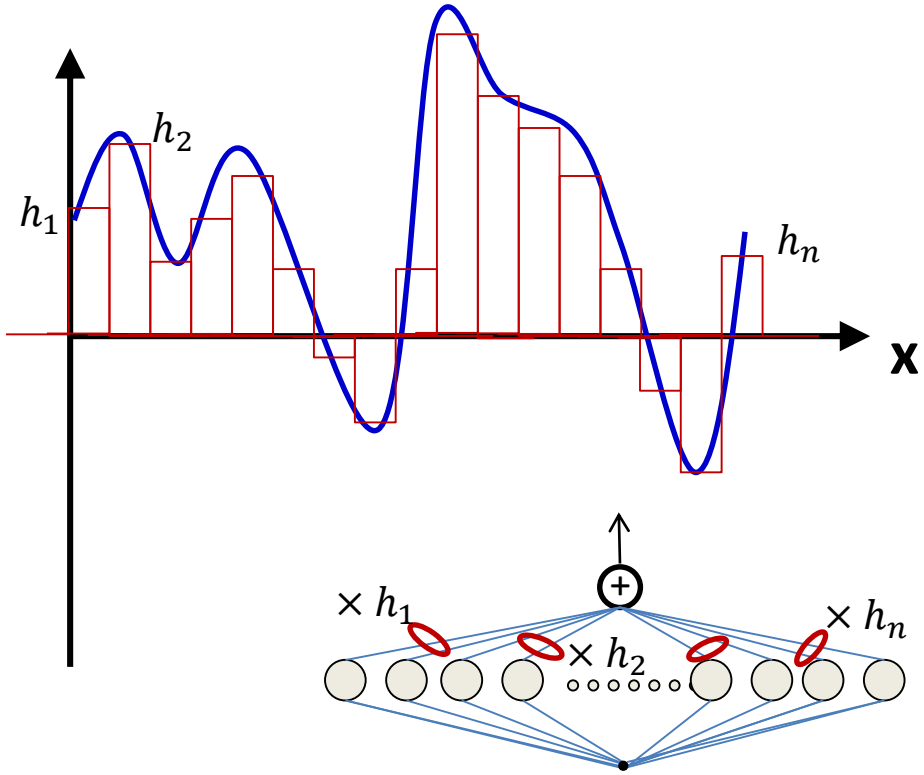
MLP as a continuous-valued regression



- 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

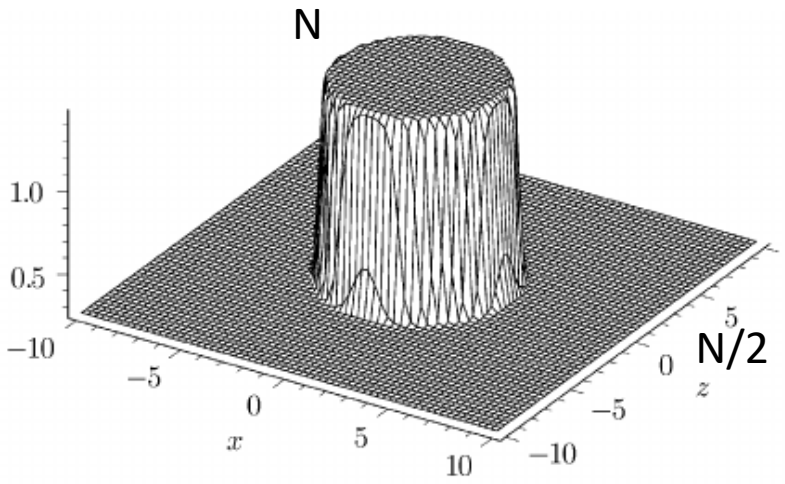
MLP as a continuous-valued regression

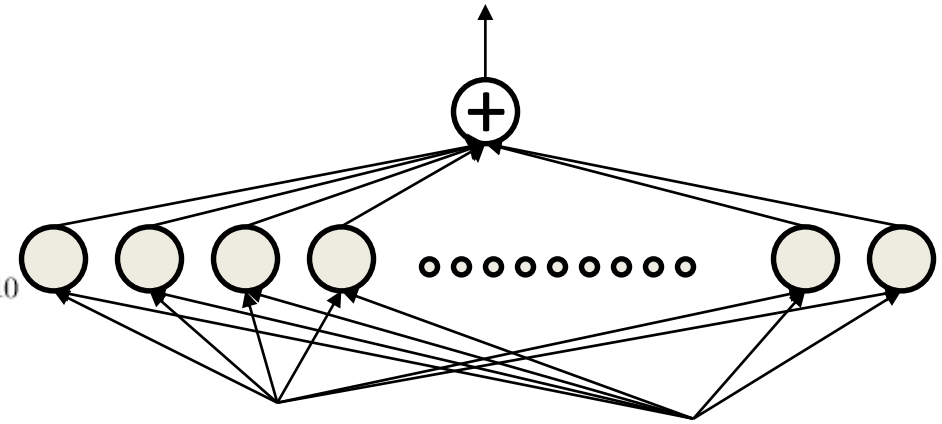




- A simple 3-unit MLP can generate a "square pulse" over an input
- **An MLP with many units can model an arbitrary function over an input**
  - To arbitrary precision
    - Simply make the individual pulses narrower
- ***A one-layer MLP can model an arbitrary function of a single input***

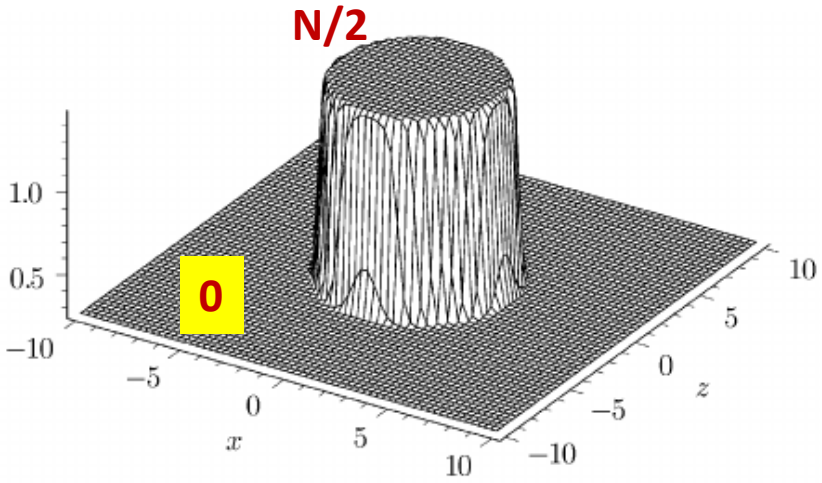
For higher-dimensional functions

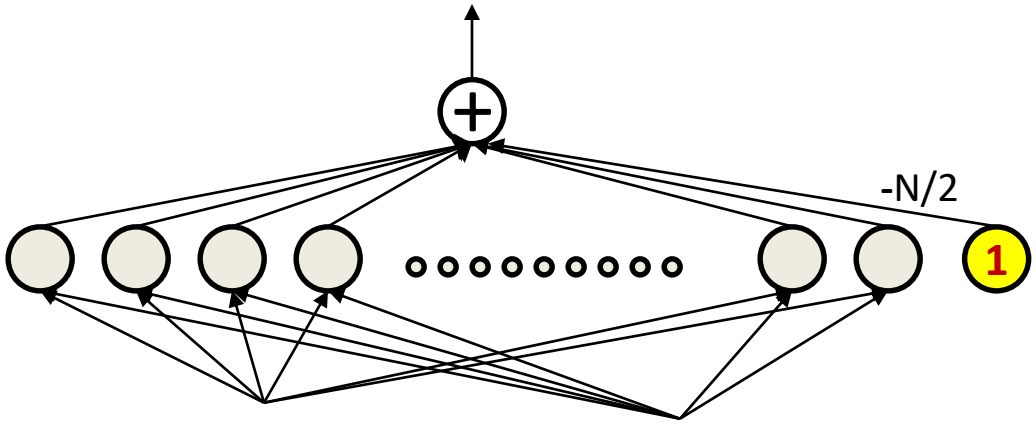




- An MLP can compose a cylinder
  - N in the circle, N/2 outside

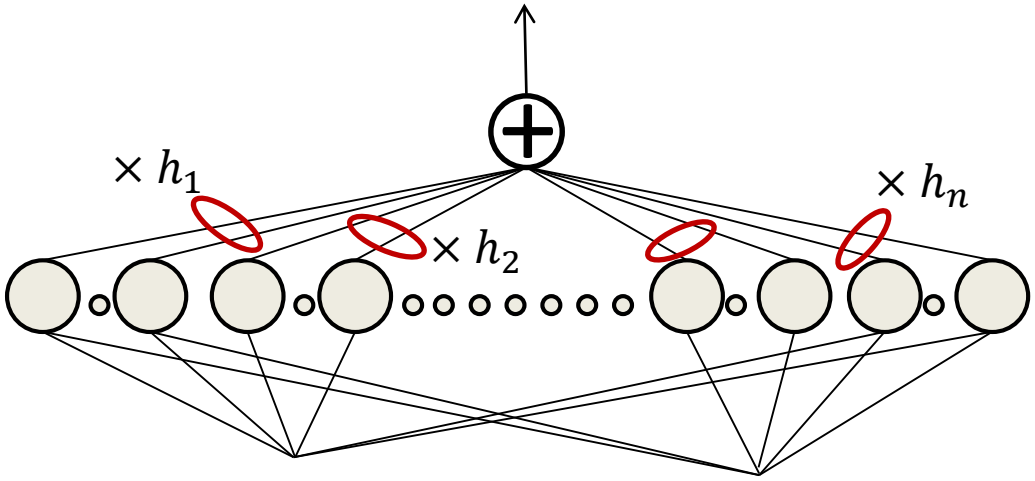
A “true” cylinder

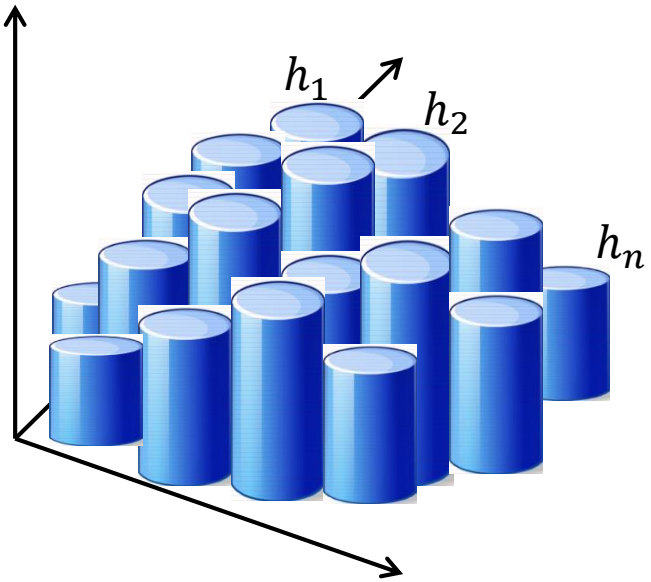




- An MLP can compose a *true* cylinder
  - N/2 in the circle, 0 outside
  - By adding a “bias”
  - We will encounter bias terms again
    - They are standard components of perceptrons

MLP as a continuous-valued function





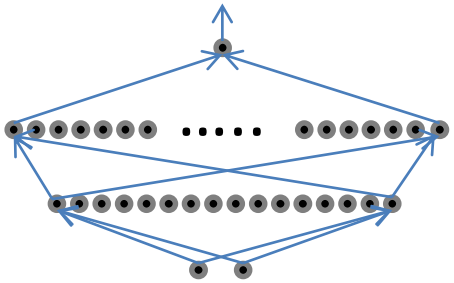
- MLPs can actually compose arbitrary functions
  - Even with only one layer
    - As sums of scaled and shifted cylinders
  - To arbitrary precision
    - By making the cylinders thinner
  - **The MLP is a universal approximator!**

The issue of depth

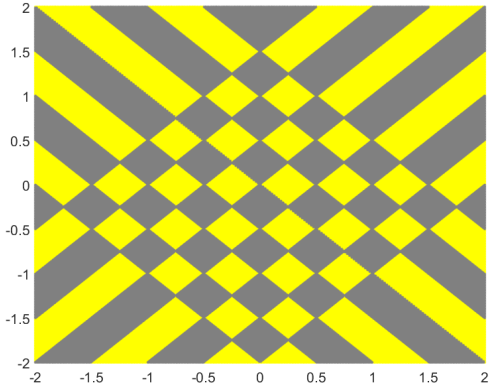
- Previous discussion showed that a *single-layer* MLP is a universal function approximator
  - Can approximate any function to arbitrary precision
  - But may require infinite neurons in the layer

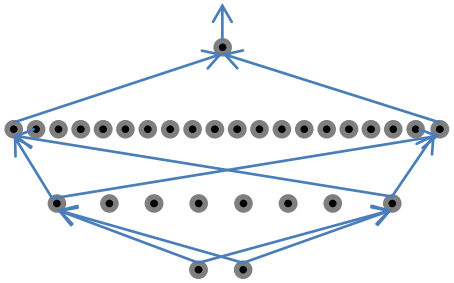
- More generally, deeper networks will require far fewer neurons for the same approximation error
  - The network is a generic map
    - The same principles that apply for Boolean networks apply here
  - Can be exponentially fewer than the 1-layer network

Sufficiency of architecture



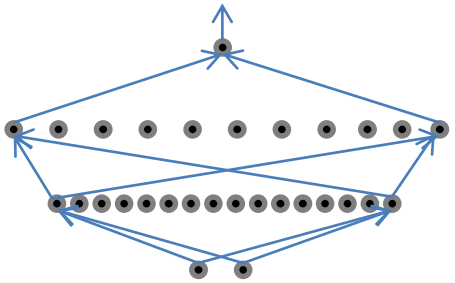
A network with 16 or more neurons in the first layer *is capable of* representing the figure to the right perfectly





A network with less than 16 neurons in the first layer cannot represent this pattern exactly

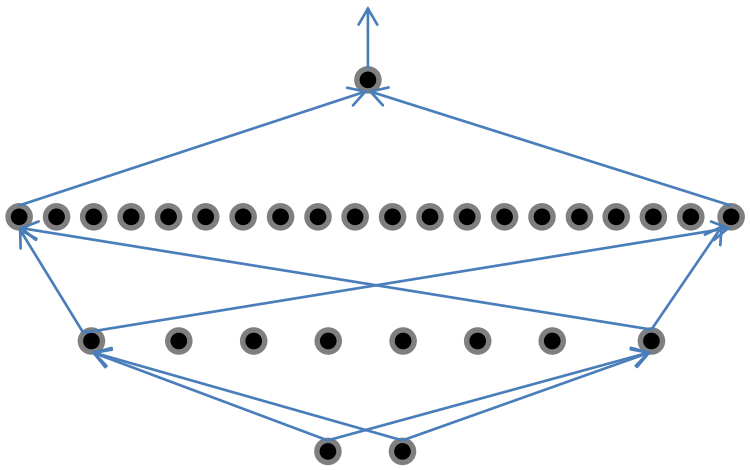
- With caveats..

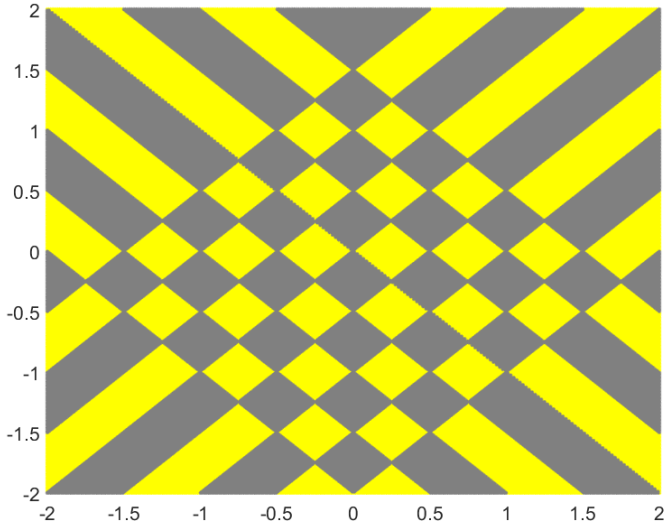


A 2-layer network with 16 neurons in the first layer cannot represent the pattern with less than 41 neurons in the second layer

- A neural network *can* represent any function provided it has sufficient *capacity*
  - I.e. sufficiently broad and deep to represent the function
- Not all architectures can represent any function

Sufficiency of architecture





- The *capacity* of a network has various definitions
  - *Information* or *Storage* capacity: how many patterns can it remember
  - VC dimension
    - bounded by the square of the number of weights in the network
  - From our perspective: largest number of disconnected convex regions it can represent

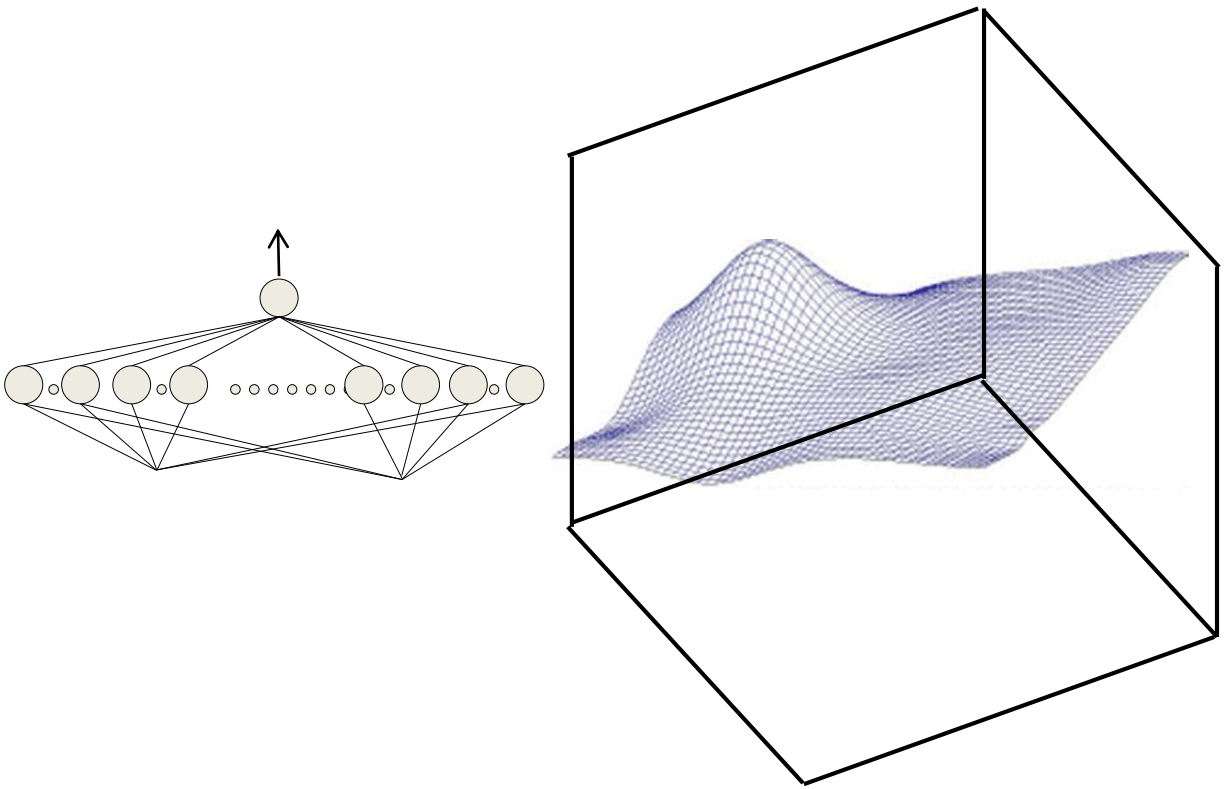
- A network with insufficient capacity *cannot* exactly model a function that requires a greater minimal number of convex hulls than the capacity of the network
  - But can approximate it with error

Lessons

- MLPs are universal Boolean function
- MLPs are universal classifiers
- MLPs are universal function approximators

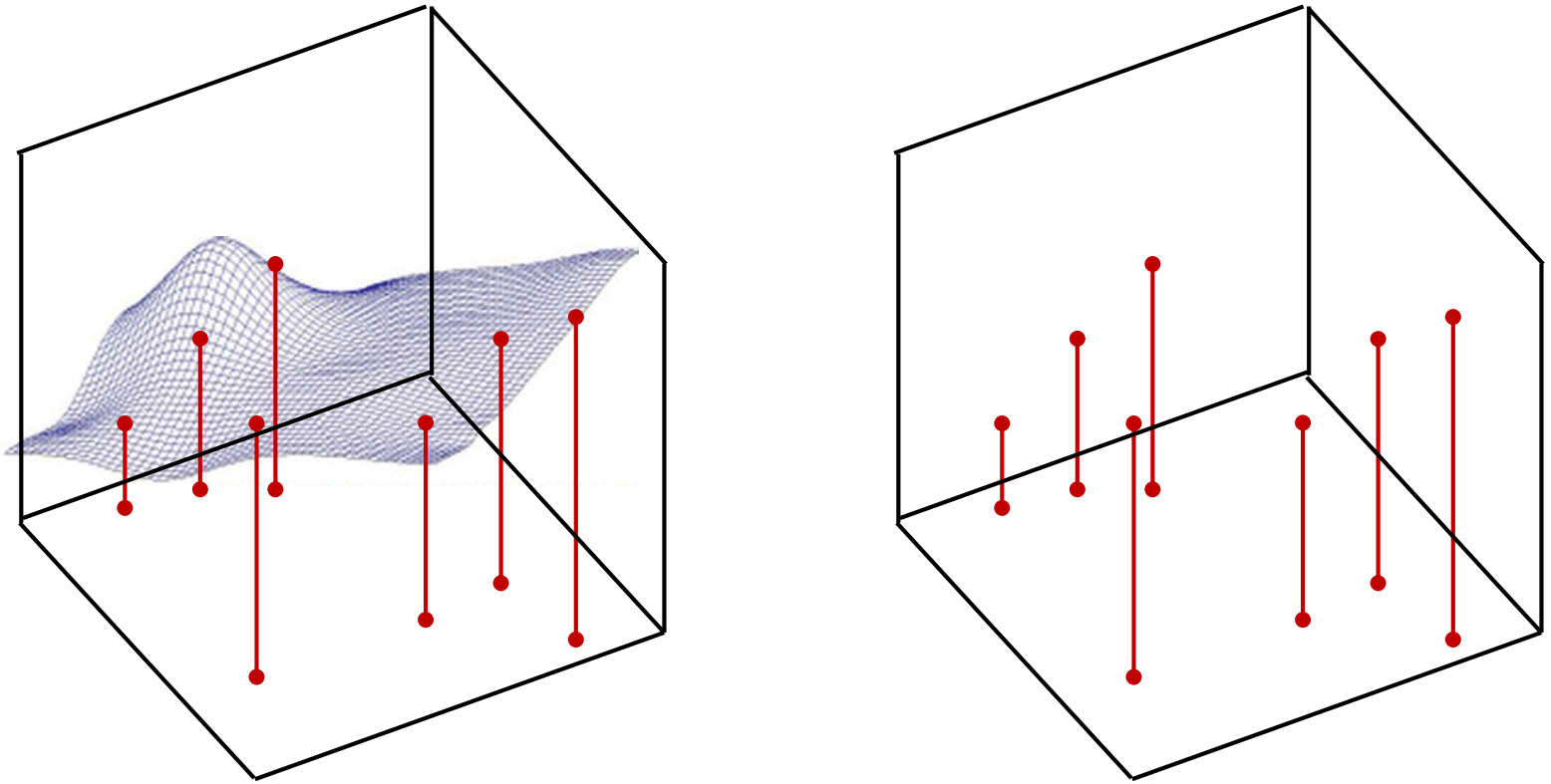
- A *single-layer* MLP can approximate anything to arbitrary precision
  - But could be exponentially or even infinitely wide in its inputs size
- Deeper MLPs can achieve the same precision with far fewer neurons
  - Deeper networks are more expressive

Learning the network



- The neural network can approximate *any* function
- But only if the function is known *a priori*

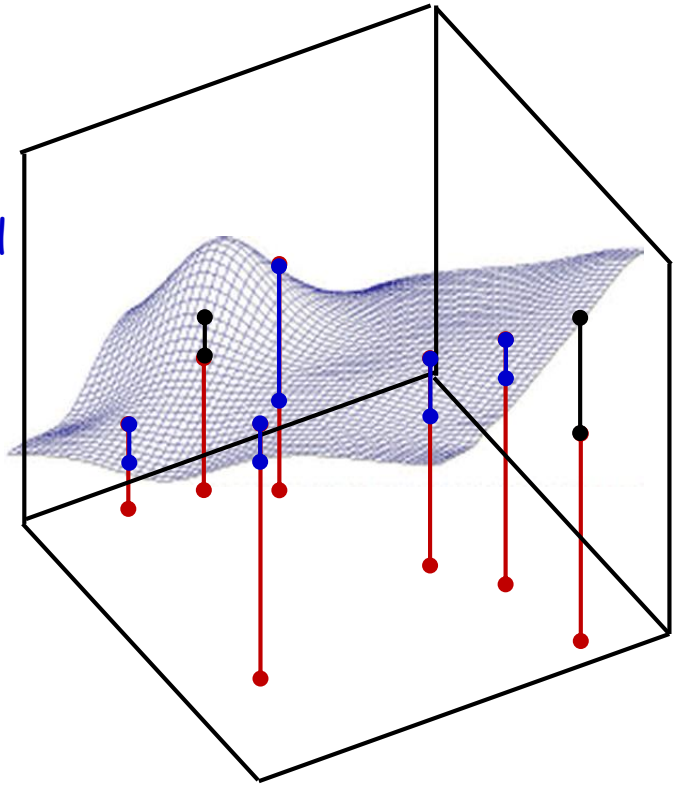
Learning the network



- In reality, we will only get a few *snapshots* of the function to learn it from
- We must learn the entire function from these “training” snapshots

General approach to training

Blue lines: error when function is *below* desired output

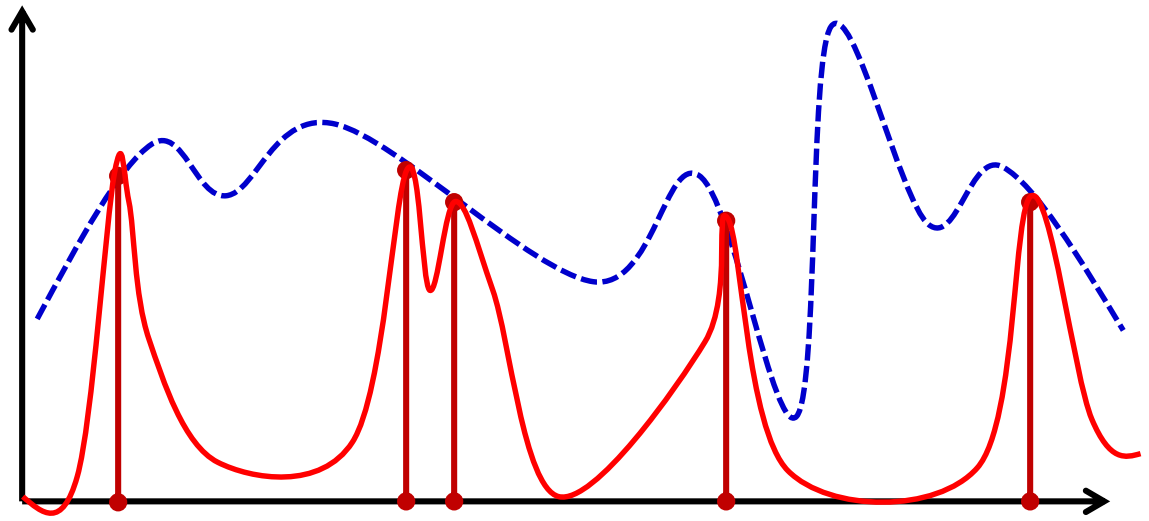


Black lines: error when function is *above* desired output

$$E = \sum_i (y_i - f(\mathbf{x}_i, \mathbf{W}))^2$$

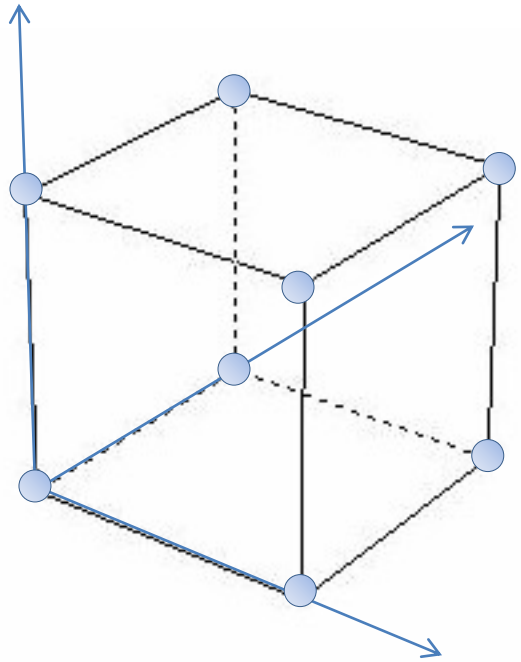
- Define an *error* between the ***actual*** network output for any parameter value and the *desired* output
  - Error typically defined as the *sum* of the squared error over individual training instances

General approach to training



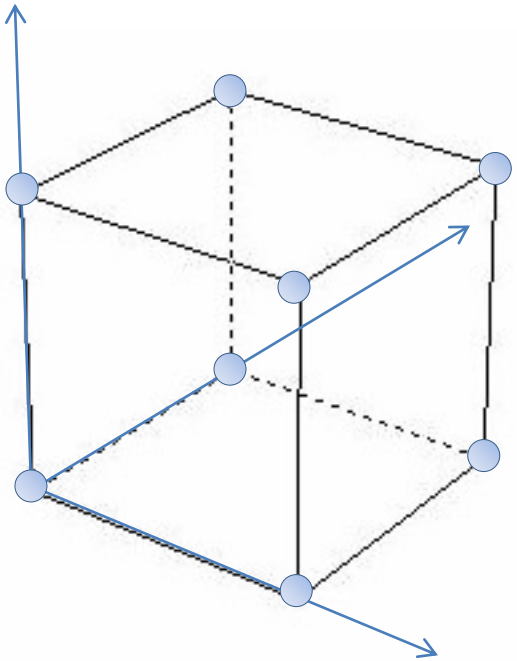
- Problem: Network may just learn the values at the inputs
  - Learn the red curve instead of the dotted blue one
    - Given only the red vertical bars as inputs
  - Need “smoothness” constraints

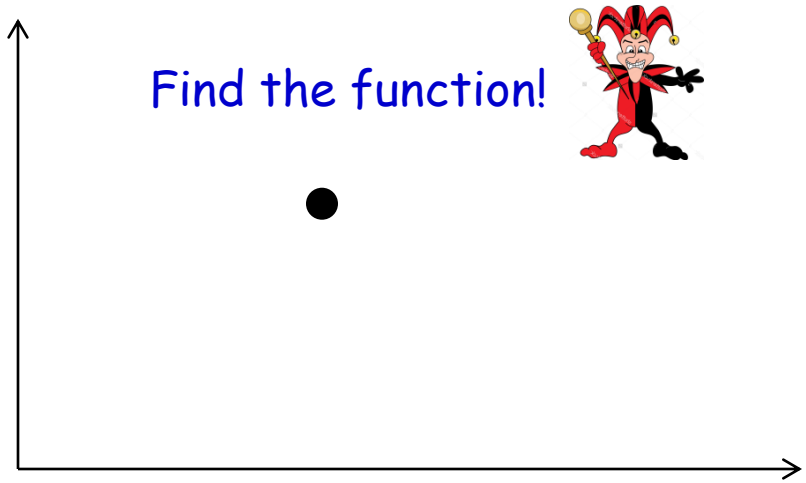
Data under-specification in learning



- Consider a binary 100-dimensional input
- There are $2^{100}=10^{30}$ possible inputs
- Complete specification of the function will require specification of 10^{30} output values
- A training set with only 10^{15} training instances will be off by a factor of 10^{15}

Data under-specification in learning

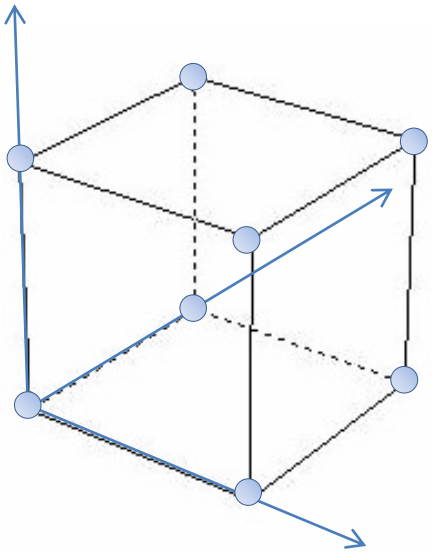




- Consider a binary 100-dimensional input
- There are $2^{100}=10^{30}$ possible inputs
- Complete specification of the function will require specification of 10^{30} output values
- A training set with only 10^{15} training instances will be off by a factor of 10^{15}

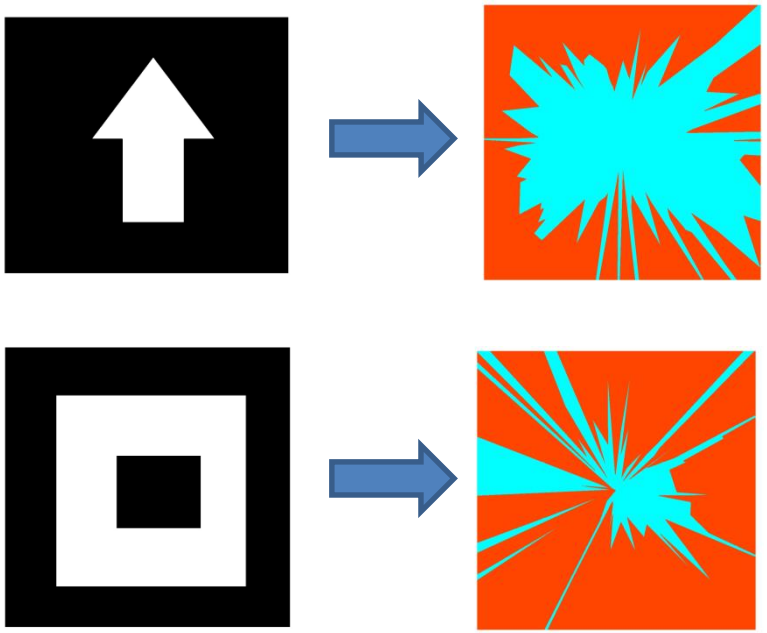
Data under-specification in learning

- MLPs naturally impose constraints
- MLPs are universal approximators
  - Arbitrarily increasing size can give you arbitrarily wiggly functions
  - The function will remain ill-defined on the majority of the space



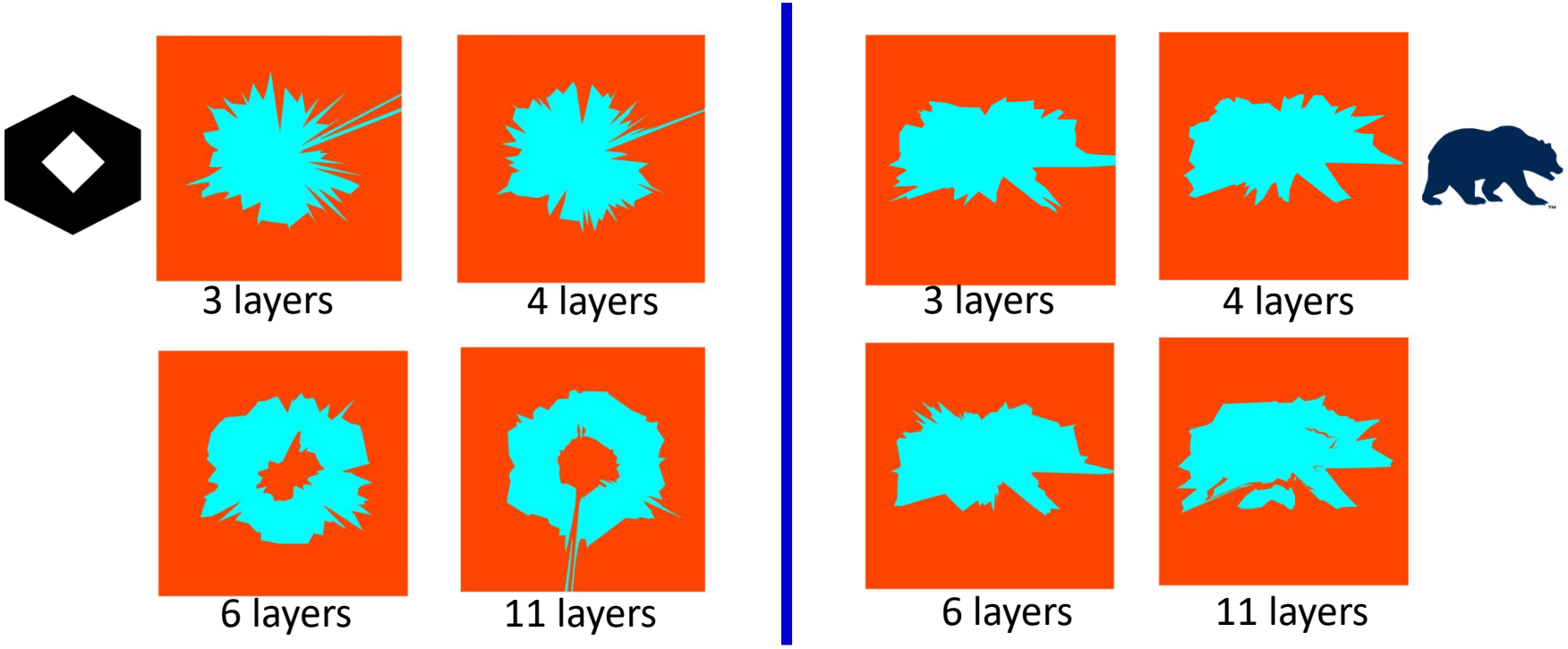
- *For a given number of parameters deeper networks impose more smoothness than shallow ones*
  - Each layer works on the already smooth surface output by the previous layer

Even when we get it all right



- Typical results (varies with initialization)
- 1000 training points Many orders of magnitude more than you usually get
- All the training tricks known to mankind

But depth and training data help



- Deeper networks seem to learn better, for the same number of total neurons
  - ***Implicit smoothness constraints***
    - ***As opposed to explicit constraints from more conventional classification models***
- **Similar functions not learnable using more usual pattern-recognition models!!**

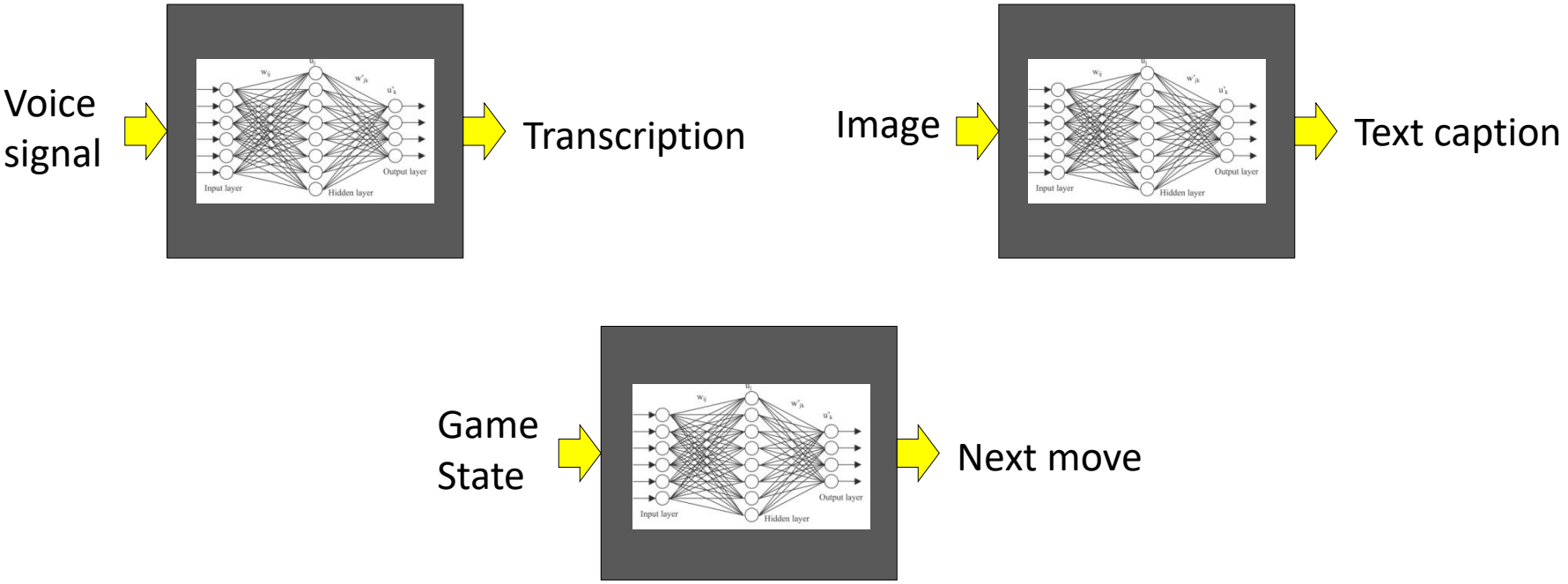
10000 training instances

Part 3: Learning the network

NNets in AI

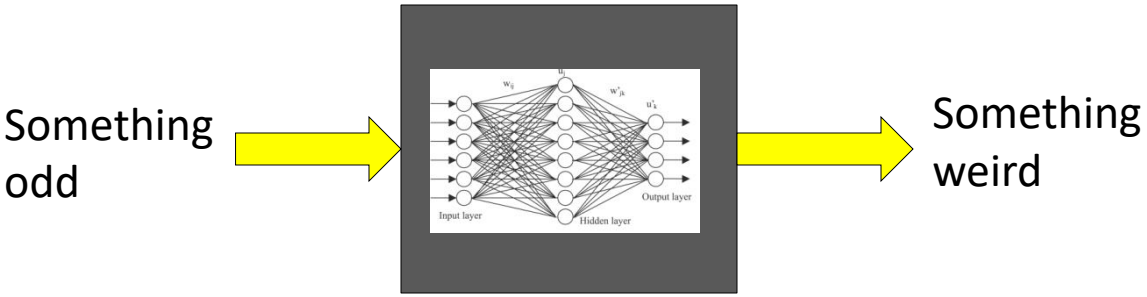
- The network is a function
  - Given an input, it computes the function layer wise to predict an output

These tasks are *functions*



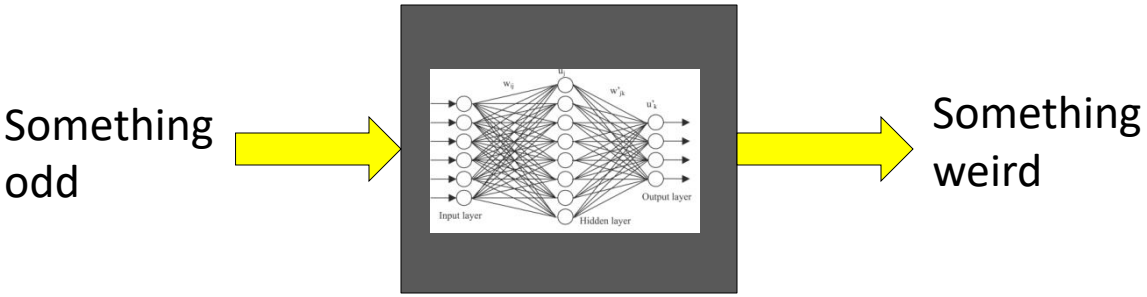
- Each of these boxes is actually a function
  - E.g f: Image → Caption

Questions



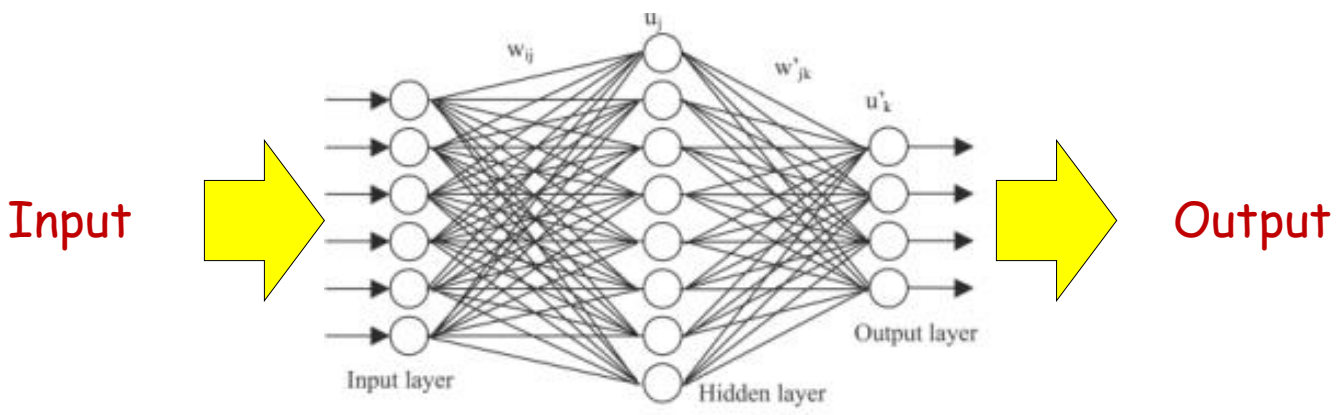
- Preliminaries:
  - How do we represent the input?
  - How do we represent the output?
- How do we compose the network that performs the requisite function?

Questions



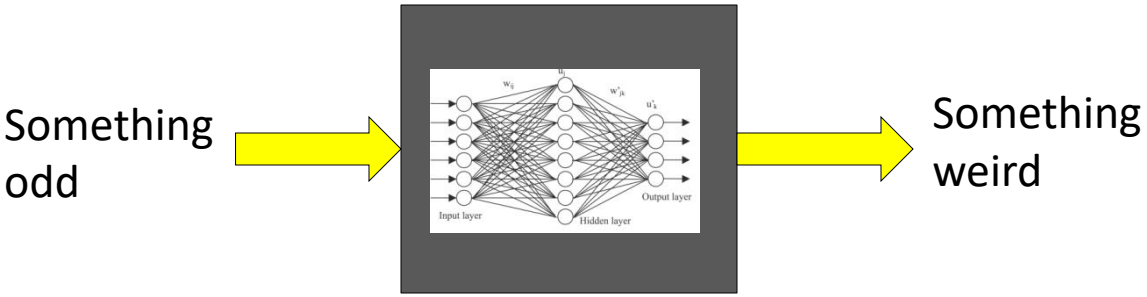
- Preliminaries:
  - How do we represent the input?
  - How do we represent the output?
- How do we compose the network that performs the requisite function?

The network as a function



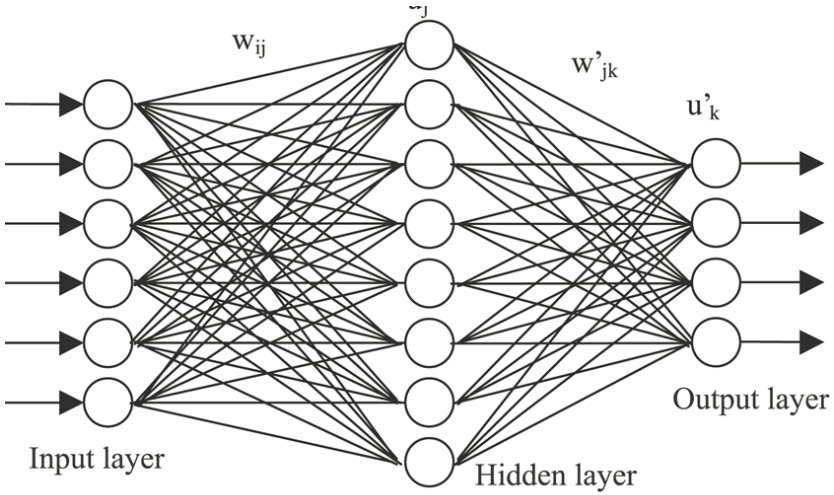
- Inputs are numeric vectors
  - Numeric representation of input, e.g. audio, image, game state, etc.
- Outputs are numeric scalars or vectors
  - Numeric “encoding” of output from which actual output can be derived
  - E.g. a score, which can be compared to a threshold to decide if the input is a face or not
  - Output may be multi-dimensional, if task requires it
- **More on this later**

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?* ⟵

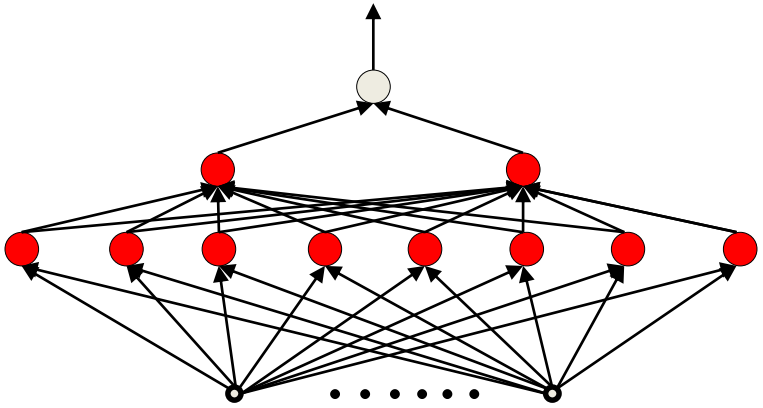
Recap

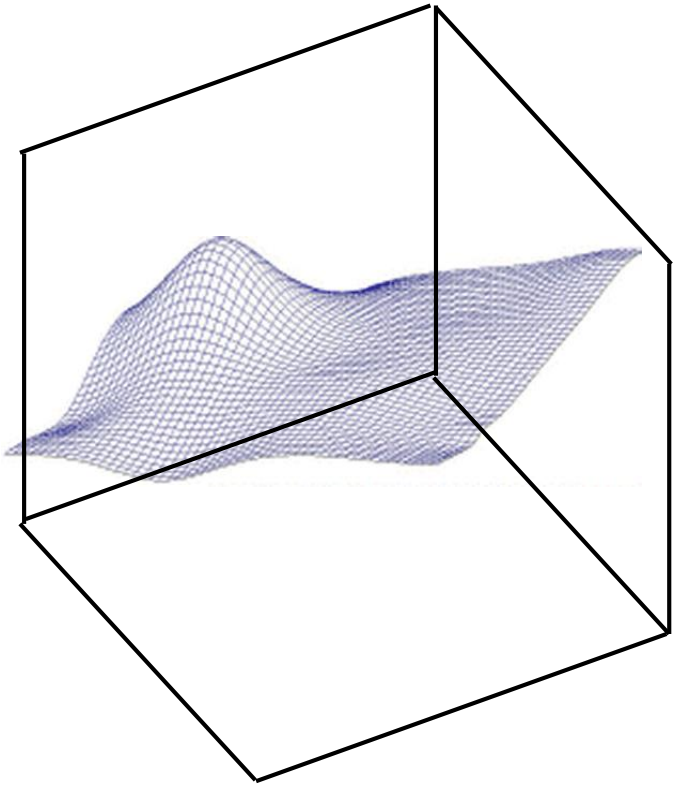


- **Neural networks are universal function approximators**
  - Can model any Boolean function
  - Can model any classification boundary
  - Can model any continuous valued function

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

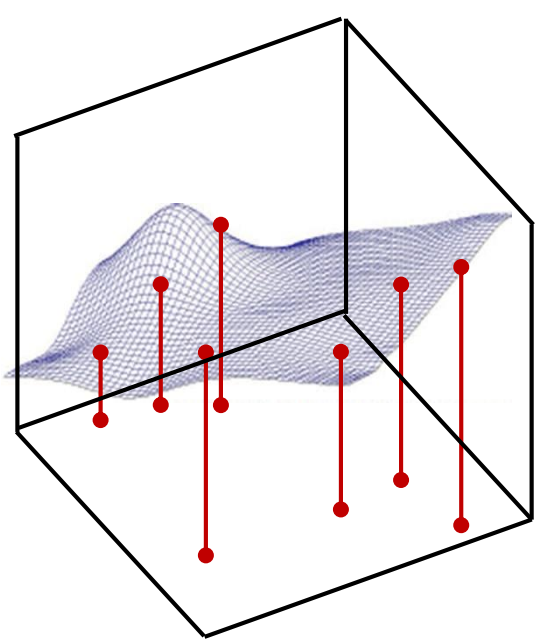
The MLP *can* represent anything

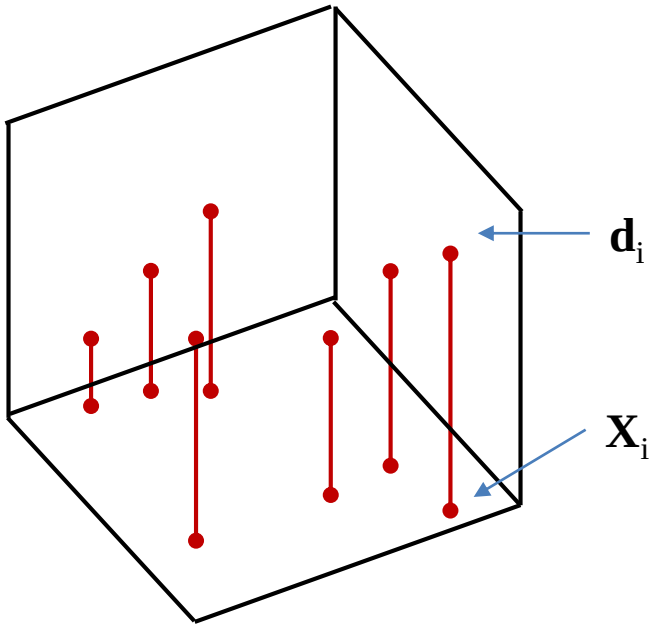




- The MLP *can be constructed* to represent anything
- But *how* do we construct it?

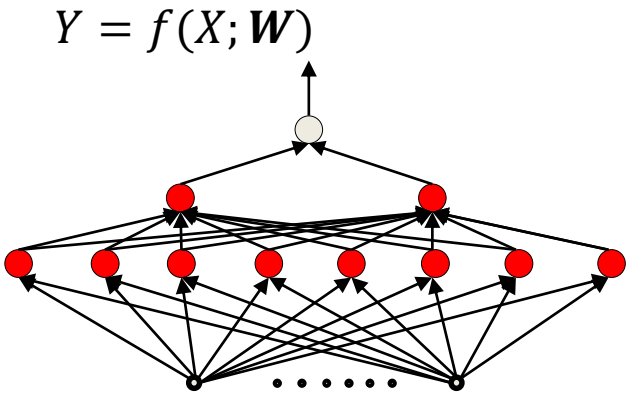
Empirical Risk Minimization

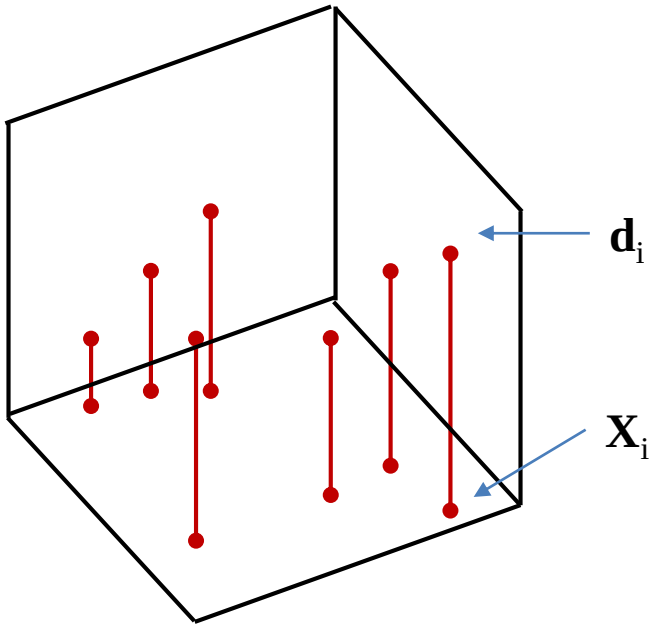




- *Get samples from* $g(X)$
  - Basically, get input-output pairs for a number of samples of input X_i
    - Many samples (X_i, d_i), where $d_i = g(X_i)$
- Very easy to do in most problems: just gather training data
  - E.g. images and labels
  - Speech and transcriptions

Empirical Risk Minimization



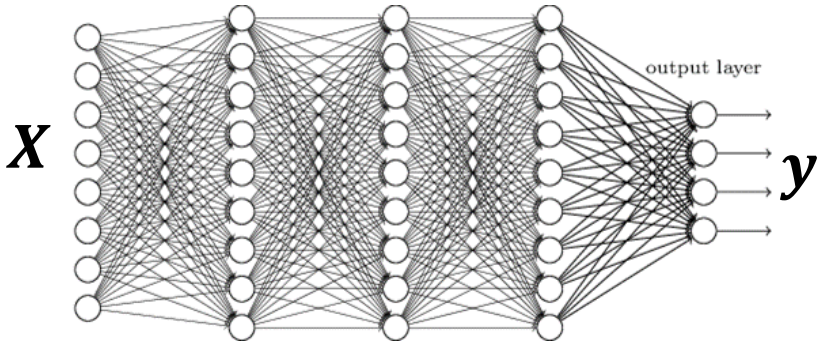


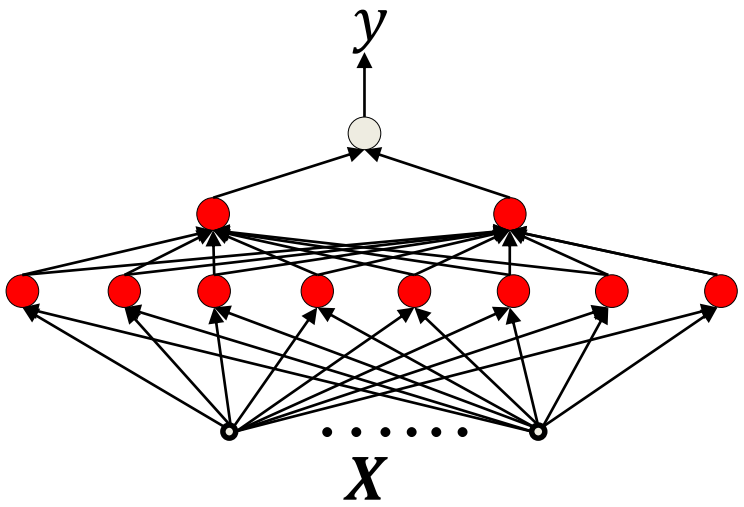
- Estimate

$$\widehat{\boldsymbol{W}} = \underset{W}{\operatorname{argmin}} \sum_i div(f(X_i; W), d_i)$$

  - WHAT IS DIV()
- Reduce the *empirical error* over the drawn samples
- Note that this is the *empirical estimate* of the true loss function $E[div(f(X; W), g(X))]$

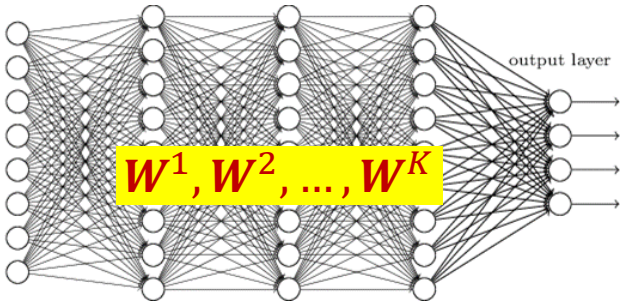
Overall setting for "Learning" the MLP





- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}), \ldots, (\boldsymbol{X}_L, \boldsymbol{y}_T)$ …
  - $\boldsymbol{d}$ 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 (weights and biases) such that the network produces the desired output for each training input
  - **The *architecture* of the network must be specified by us**

Procedural outline



Actual output of network:

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

Desired output of network: d_i

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

Total training error:

$$Err(W^1, W^2, \ldots, W^K) = \sum_i Div(Y_i, d_i; W^1, W^2, \ldots, W^K)$$

- Optimize network parameters to minimize the total error over all training inputs

Problem Statement

- Minimize the following function

$$Err(\boldsymbol{W}^1, \boldsymbol{W}^2, \ldots, \boldsymbol{W}^K)$$

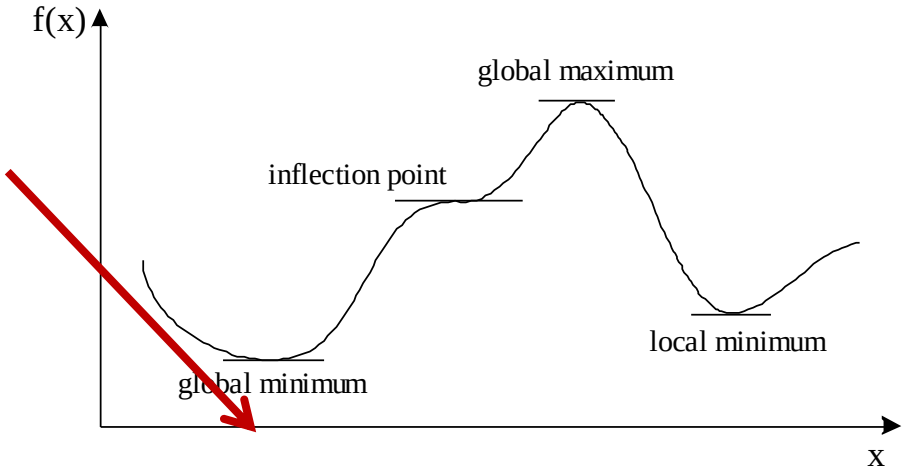
$$= \sum_i \boldsymbol{Div}(\boldsymbol{Y}_i, \boldsymbol{d}_i; \boldsymbol{W}^1, \boldsymbol{W}^2, \ldots, \boldsymbol{W}^K)$$

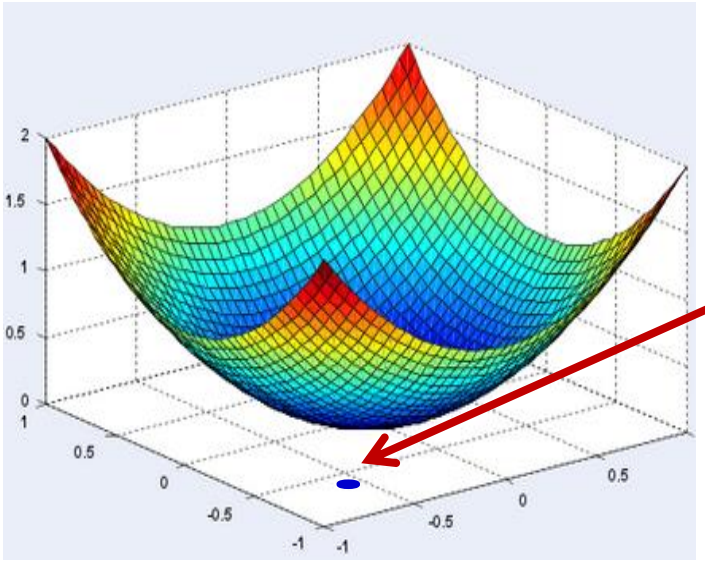
w.r.t $\boldsymbol{W}^1, \boldsymbol{W}^2, \ldots, \boldsymbol{W}^K$

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

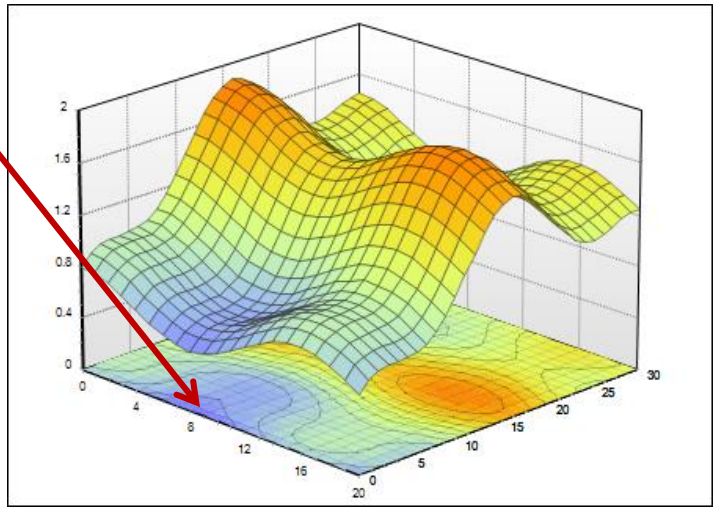
- **A CRASH COURSE ON FUNCTION OPTIMIZATION**

The problem of optimization

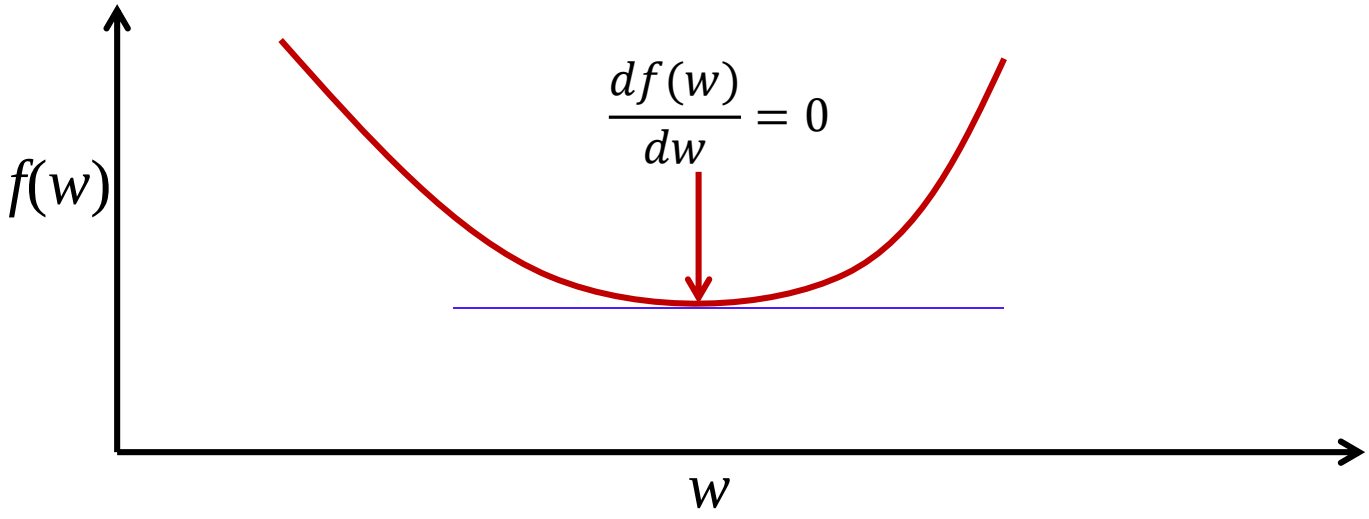




- General problem of optimization: find the value of $\boldsymbol{w}$ where $\mathbf{f}(\boldsymbol{w})$ is minimum



Finding the minimum of a function

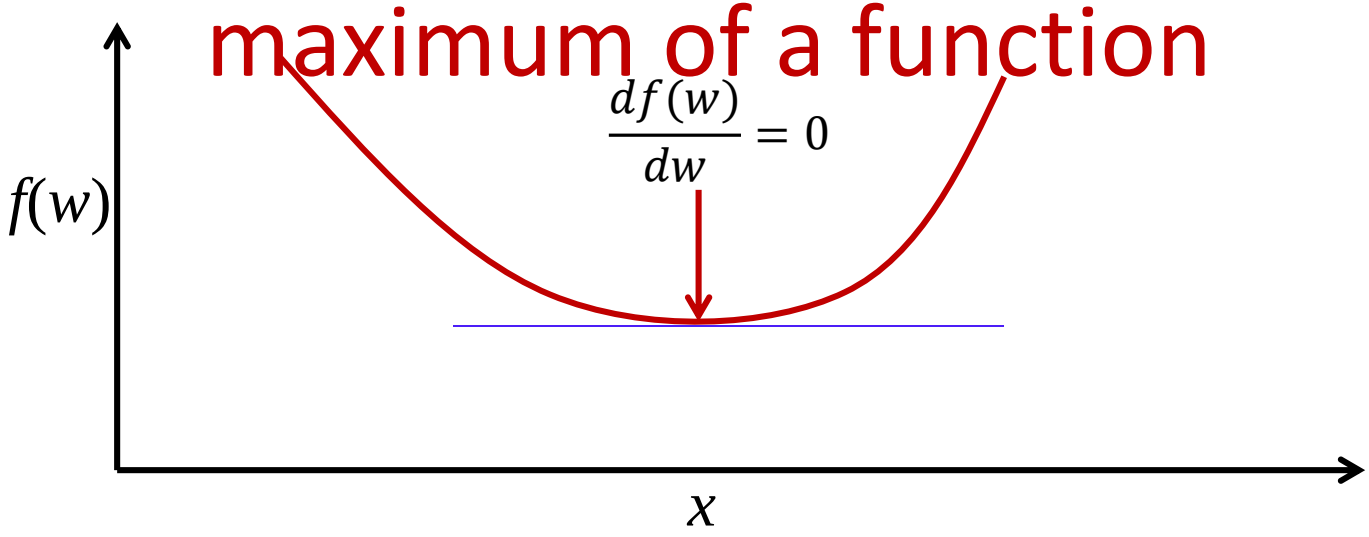


- Find the value w at which $f'(w) = 0$
  - Solve

$$\frac{df(w)}{dw} = 0$$

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

Soln: Finding the minimum or maximum of a function



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

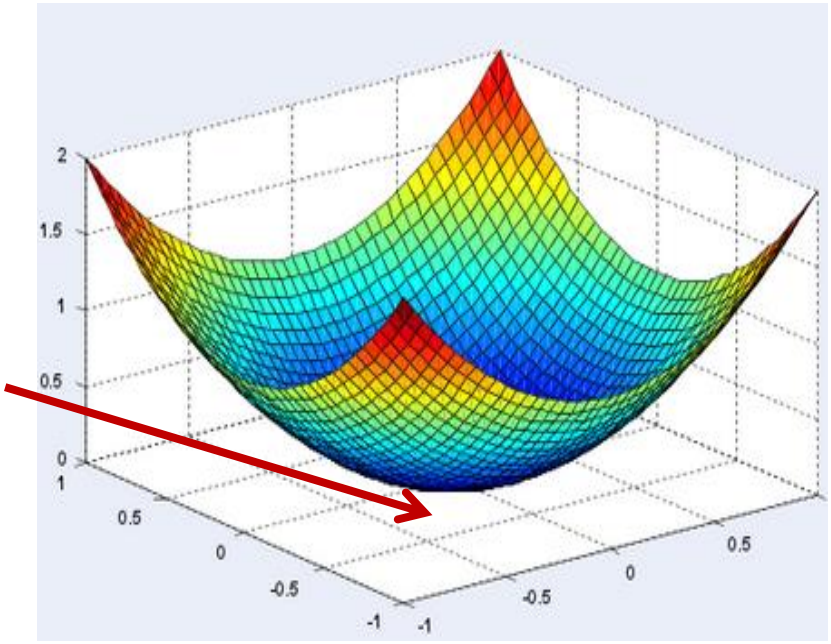
$$\frac{df(w)}{dw} = 0$$

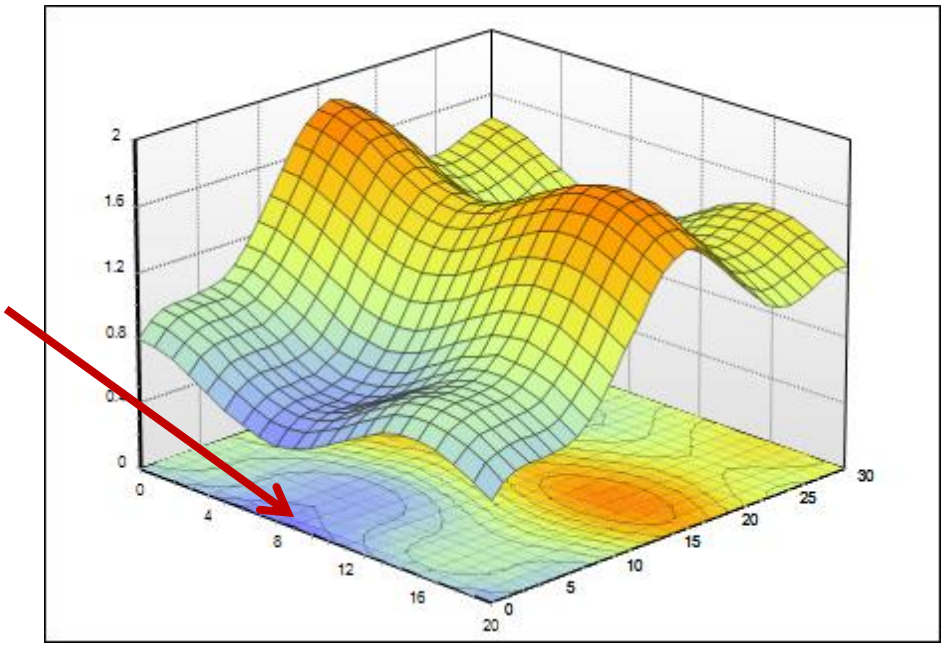
- The solution w_{soln} is a turning point
- Check the double derivative at x_{soln} : compute

$$f''(w_{soln}) = \frac{df'(w_{soln})}{dw}$$

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

What about functions of multiple variables?

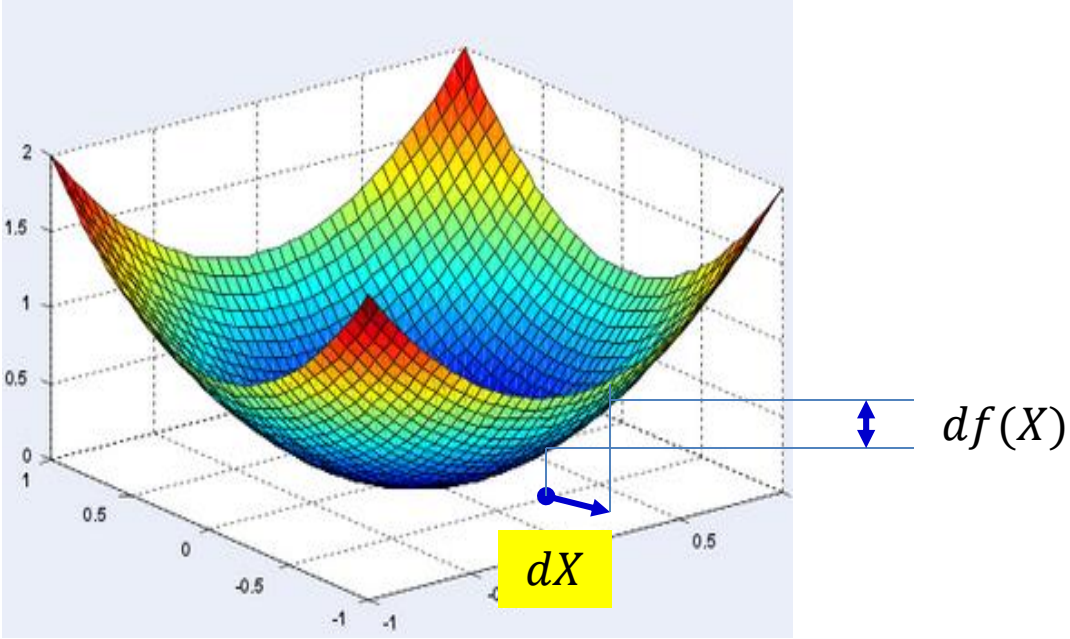




- The optimum point is still "turning" point
  - Shifting in any direction will increase the value
  - For smooth functions, miniscule shifts will not result in any change at all
- We must find a point where shifting in any direction by a microscopic amount will not change the value of the function

A brief note on derivatives of multivariate functions

The *Gradient* of a scalar function



- The *Gradient* $\nabla f(X)$ 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

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

Gradients of scalar functions with multi-variate inputs

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

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

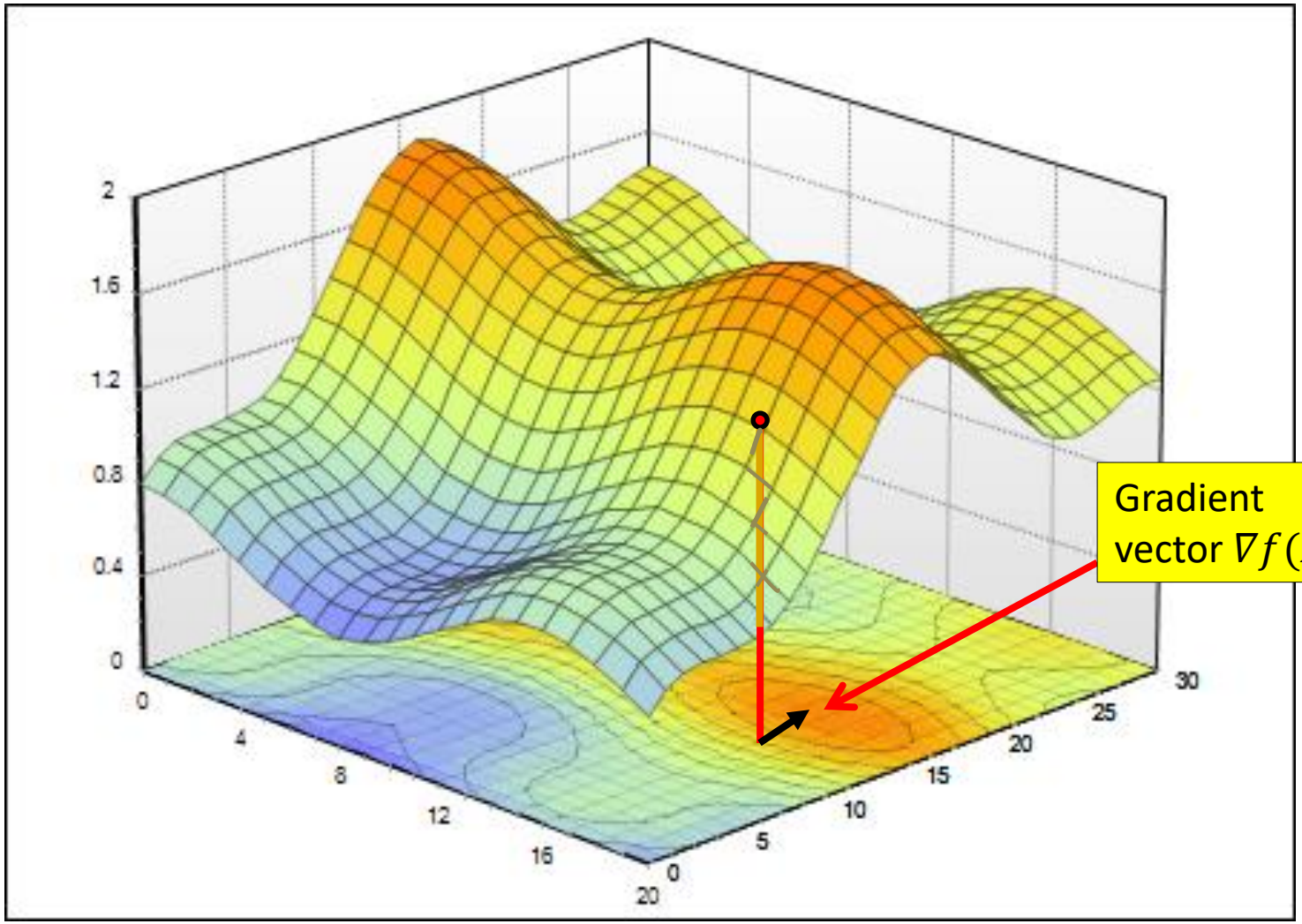
- Check:

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

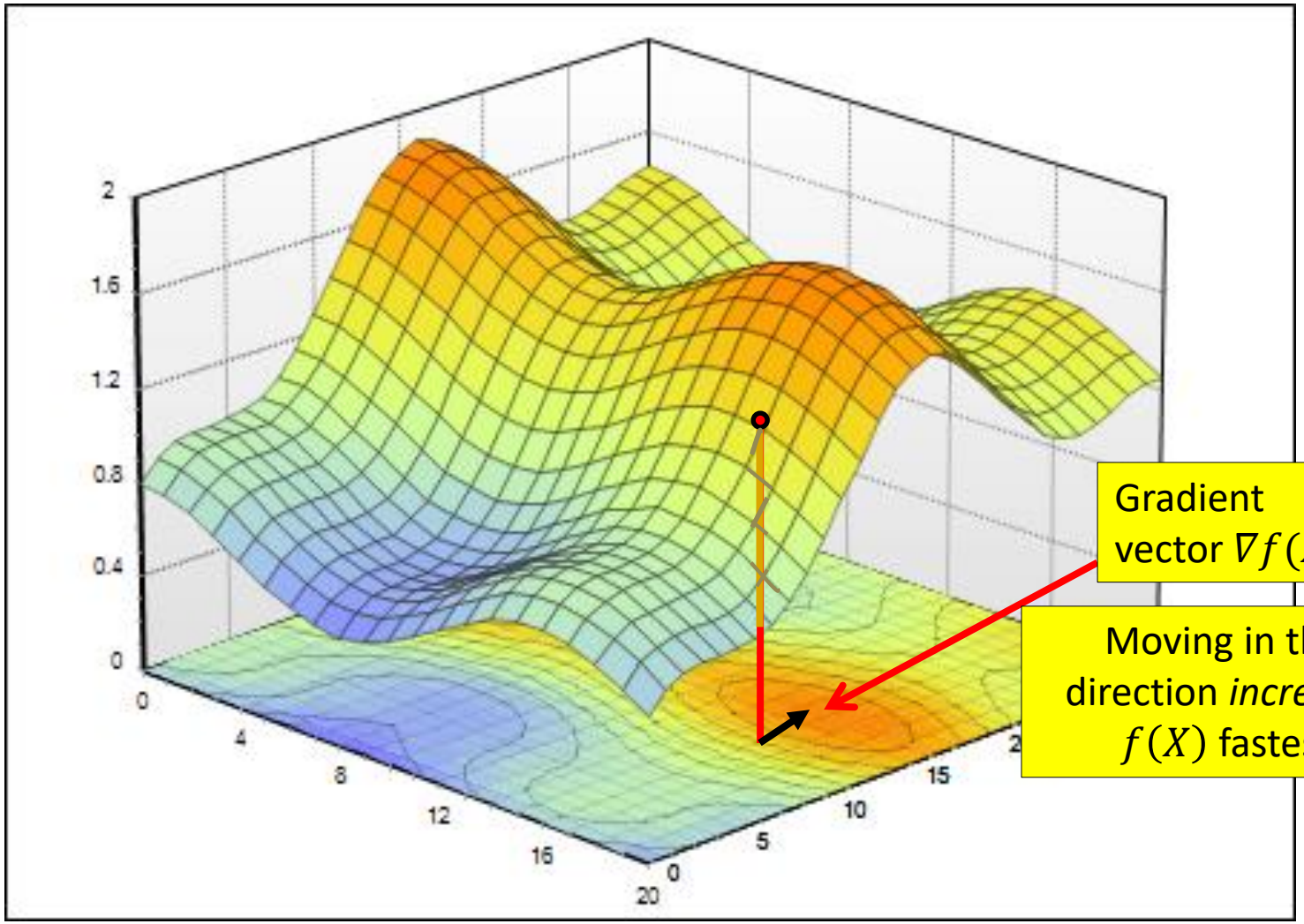
$$= \frac{\partial f(X)}{\partial x} dx_1 + \frac{\partial f(X)}{\partial x} dx_2 + \cdots + \frac{\partial f(X)}{\partial x} dx_n$$

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

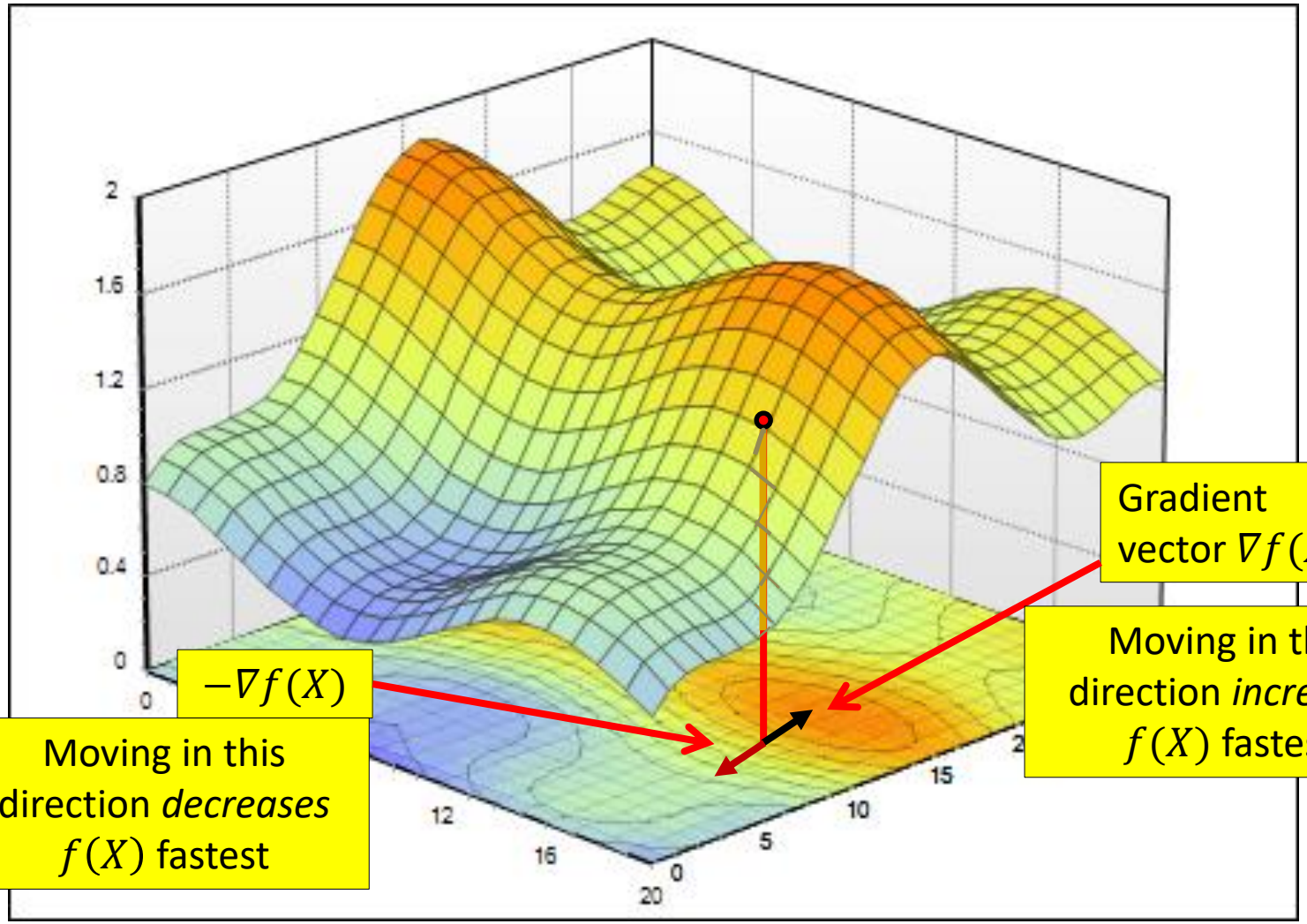
Gradient


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 $\nabla f(X)$

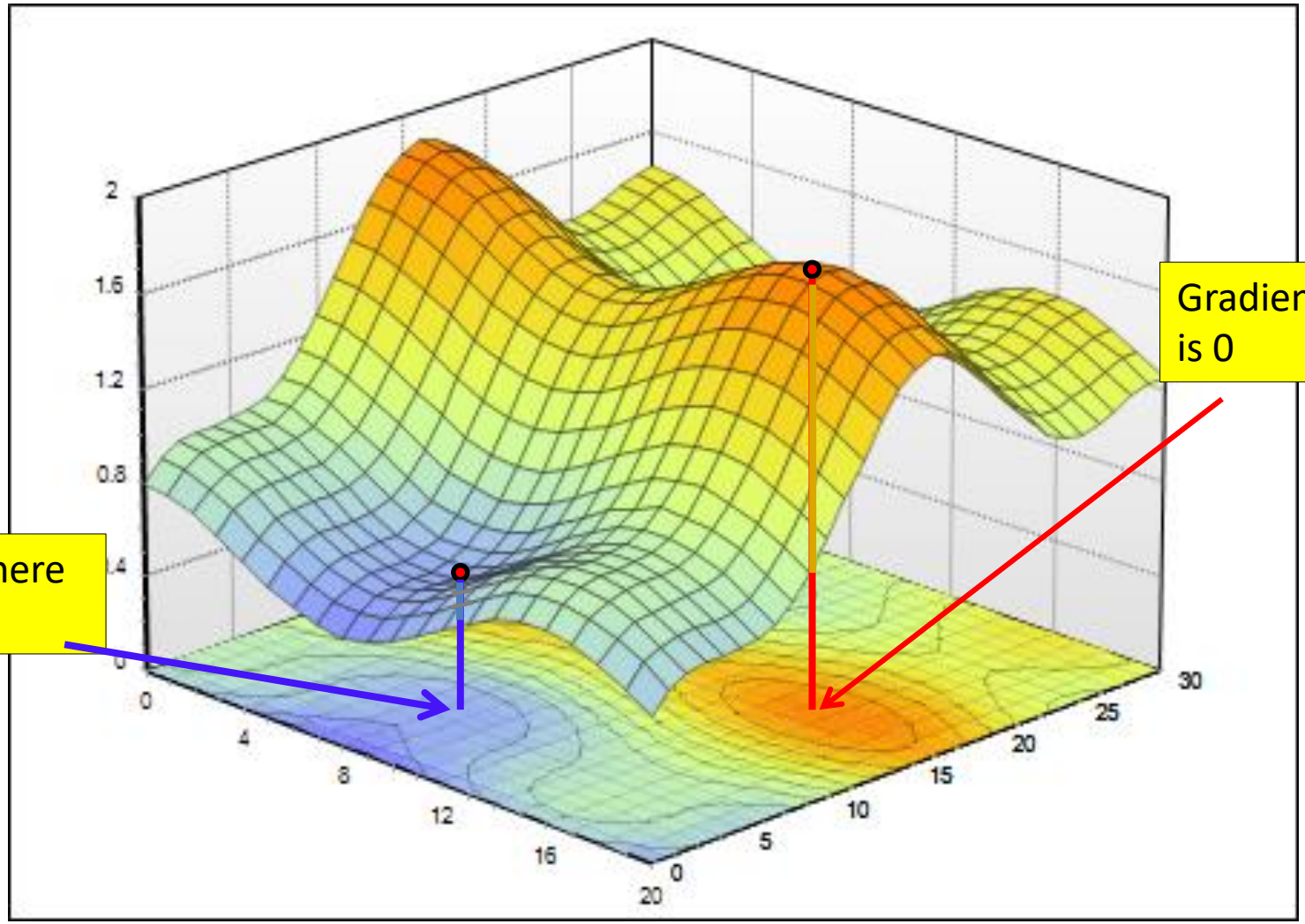
Gradient


2
1.6
1.2
0.8
0.4
0
0
4
8
12
16
20
0
5
10
15
Gradient vector $\nabla f(X)$
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 $\nabla f(X)$
Moving in this direction *increases* $f(X)$ fastest
$-\nabla f(X)$
Moving in this direction *decreases* $f(X)$ fastest

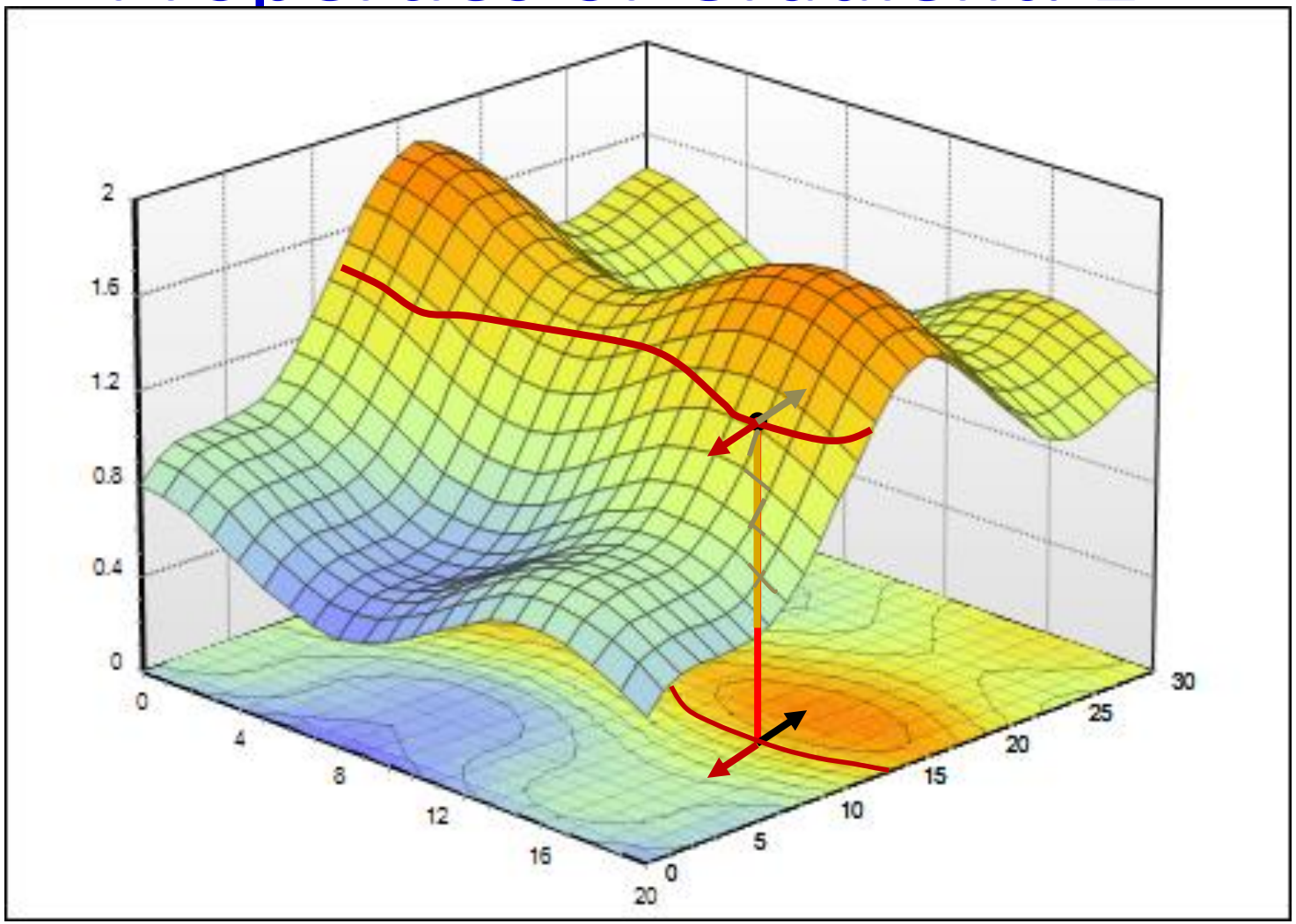
Gradient



Gradient here is 0

Gradient here is 0

Properties of Gradient: 2



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

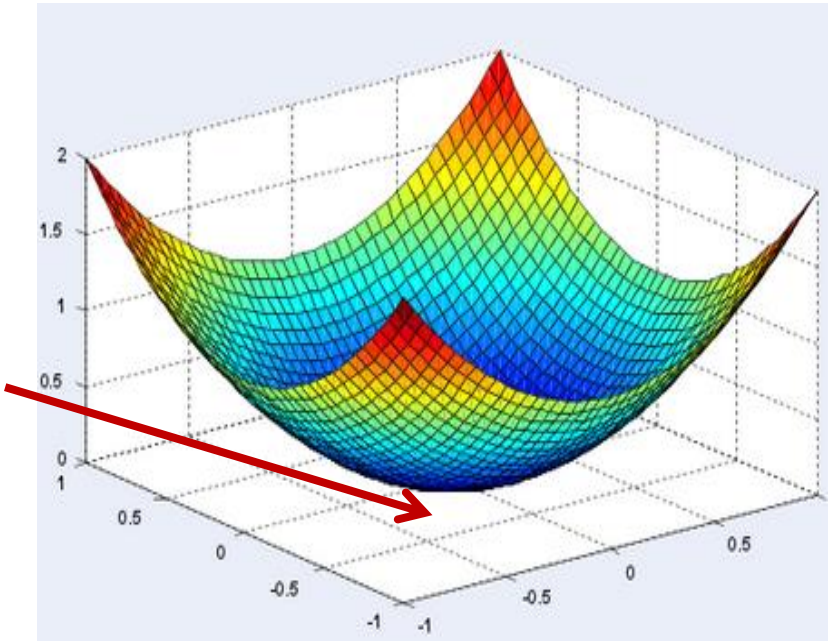
The Hessian

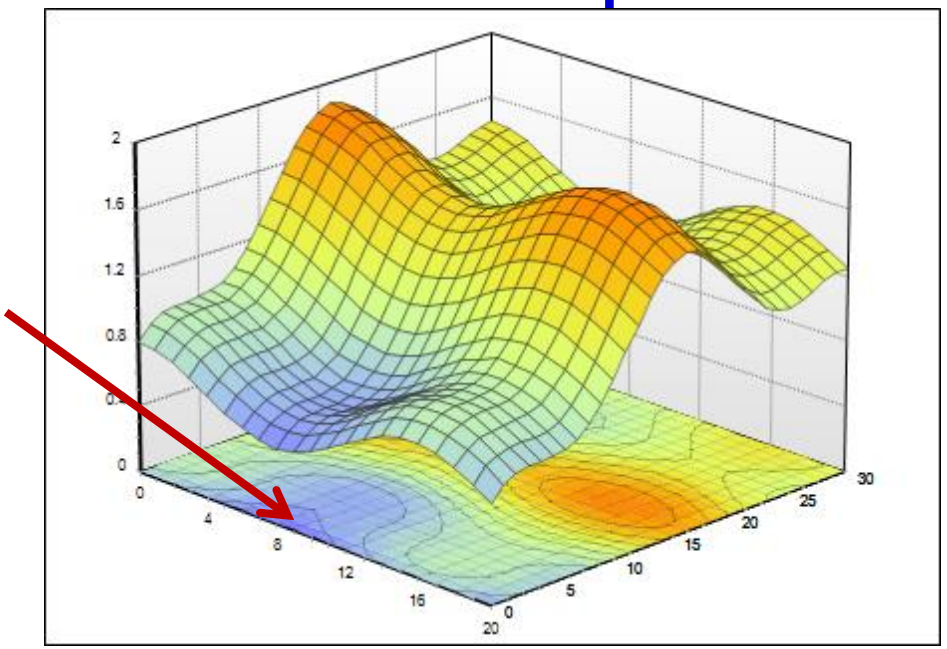
- The Hessian of a function $f(x_1, x_2, \ldots, x_n)$ is given by the second derivative

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

Returning to direct optimization…

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





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

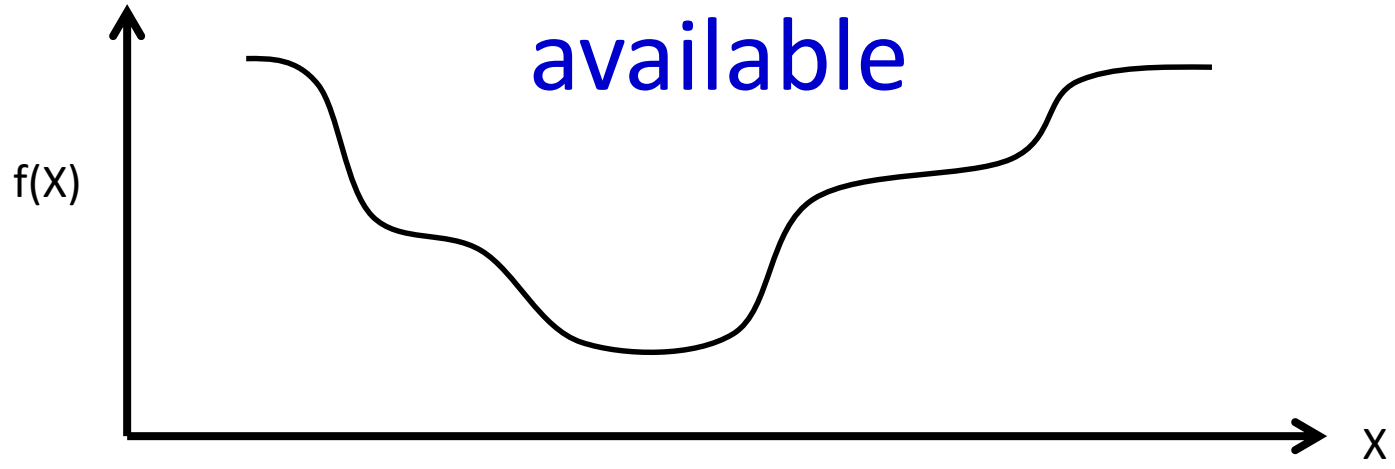
Unconstrained Minimization of function (Multivariate)

1. Solve for the X where the gradient equation equals to zero

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

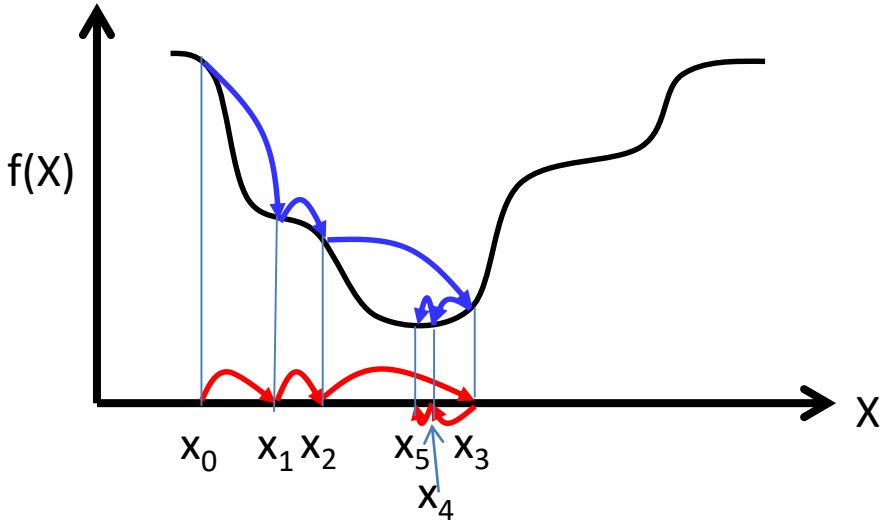
2. Compute the Hessian Matrix $\nabla^2 f(X)$ at the candidate solution and verify that
   - Hessian is positive definite (eigenvalues positive) -> to identify local minima
   - Hessian is negative definite (eigenvalues negative) -> to identify local maxima

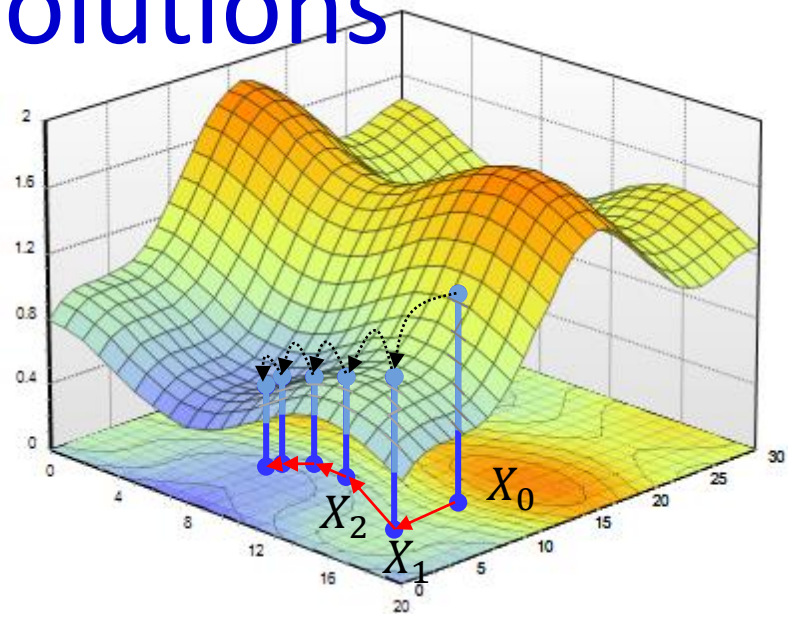
Closed Form Solutions are not always available



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

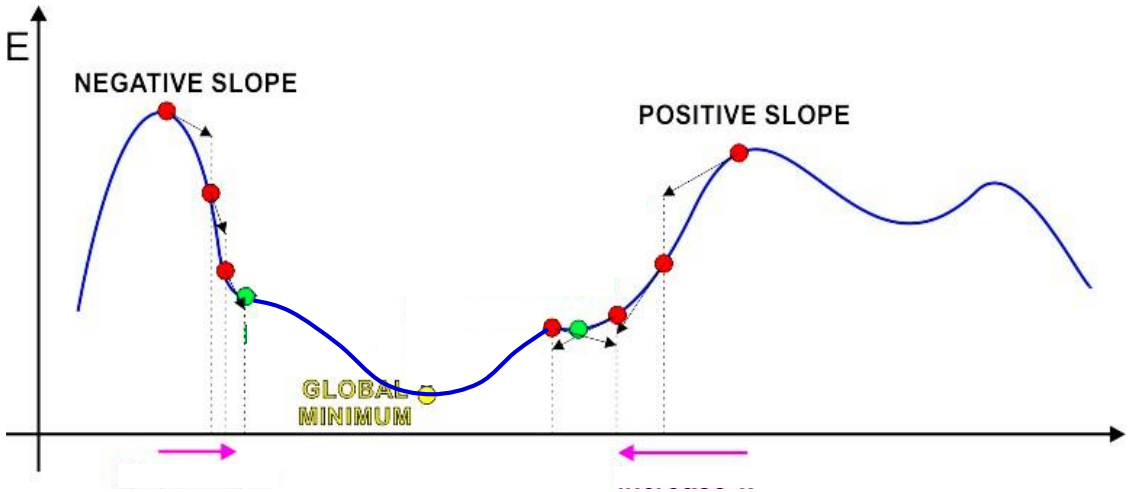
Iterative solutions





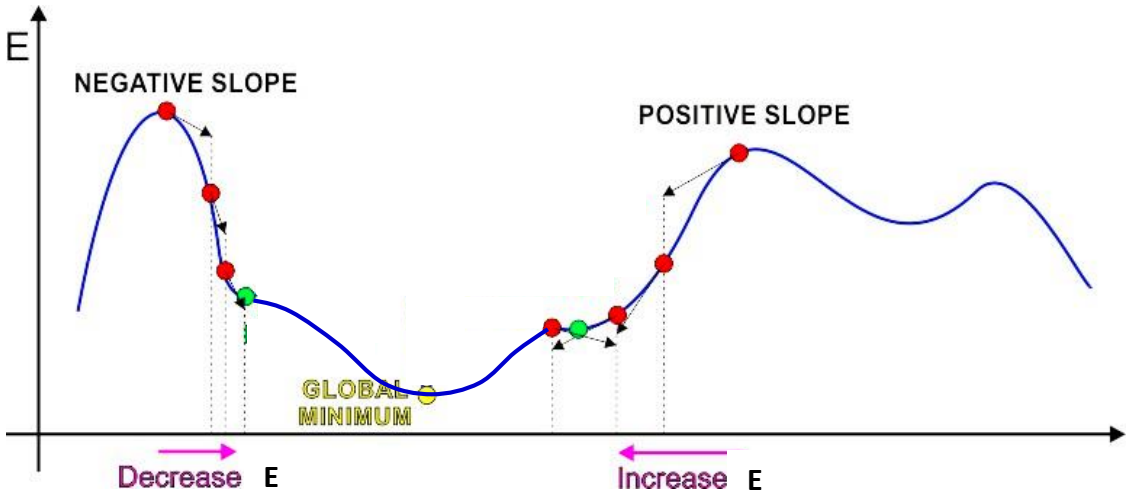
- Iterative solutions
  - Start from an initial guess X_0 for the optimal X
  - Update the guess towards a (hopefully) "better" value of $f(X)$
  - Stop when $f(X)$ no longer decreases
- Problems:
  - Which direction to step in
  - How big must the steps be

The Approach of Gradient Descent



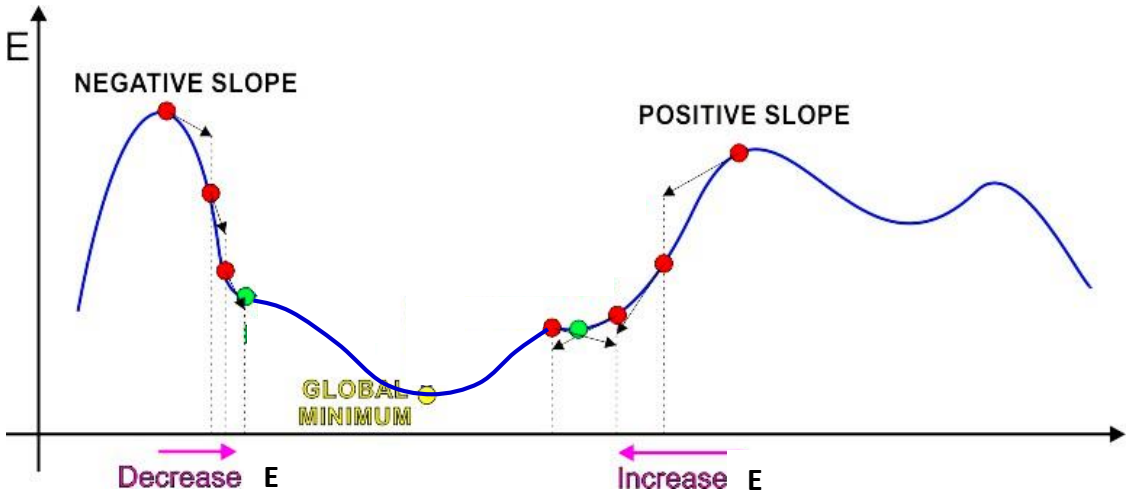
- Iterative solution:
  - Start at some point
  - Find direction in which to shift this point to decrease error
    - This can be found from the derivative of the function
      - A positive derivative → moving left decreases error
      - A negative derivative → moving right decreases error
  - Shift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

The Approach of Gradient Descent



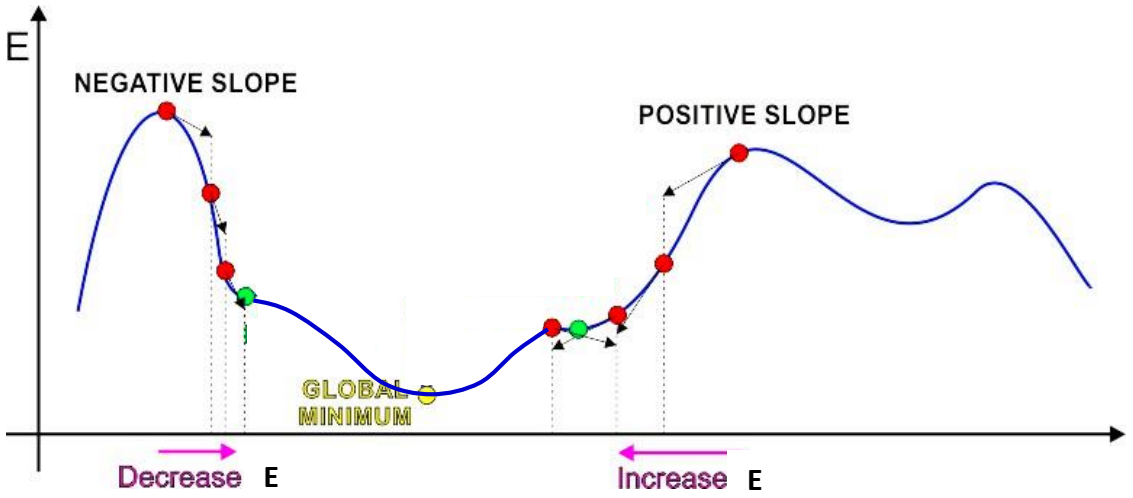
- Iterative solution: Trivial algorithm
  - Initialize x^0
  - While $f'(x^k) \neq 0$
    - If $sign\left(f'(x^k)\right)$ is positive:
      - $x^{k+1} = x^k - step$
    - Else
      - $x^{k+1} = x^k + step$
  - What must step be to ensure we actually get to the optimum?

The Approach of Gradient Descent



- Iterative solution: Trivial algorithm
  - Initialize x^0
  - While $f'(x^k) \neq 0$
    - $x^{k+1} = x^k - sign\left(f'(x^k)\right).step$
  - Identical to previous algorithm

The Approach of Gradient Descent



- Iterative solution: Trivial algorithm
  - Initialize x_0
  - While $f'(x^k) \neq 0$
    - $x^{k+1} = x^k - \eta^k f'(x^k)$
  - η^k is the "step size"

Gradient descent/ascent (multivariate)

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

$$x^{k+1} = x^k + \eta^k \nabla f\left(x^k\right)^T$$

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

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

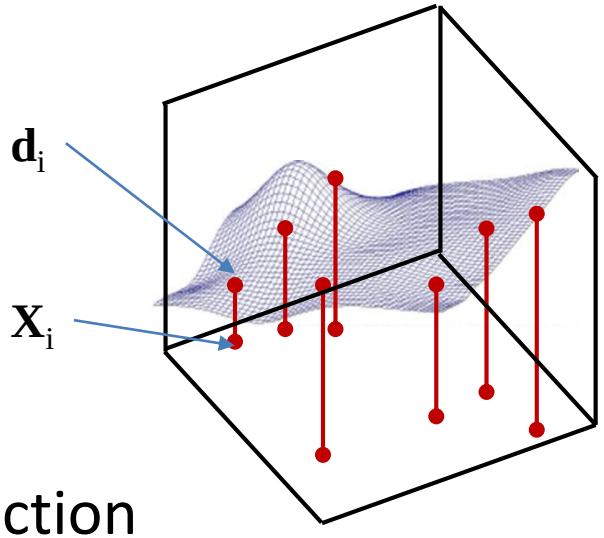
- Many solutions to choosing step size η^k

Gradient Descent Algorithm

- In order to minimize any function $f(x)$ w.r.t. x
- Initialize:
  - x^0
  - $k = 0$
- While $\left|f\left(x^{k+1}\right) - f\left(x^k\right)\right| > \varepsilon$
  - $x^{k+1} = x^k - \eta^k \nabla f\left(x^k\right)^T$
  - $k = k + 1$

- Back to Neural Networks

Recap



- Neural networks can model any function
- They must be *trained* to represent the function
- In the usual setting, they must be trained from input-output pairs
- We will use empirical risk minimization to learn network parameters
- We will use (variants of) gradient descent to learn them

Typical Problem Statement

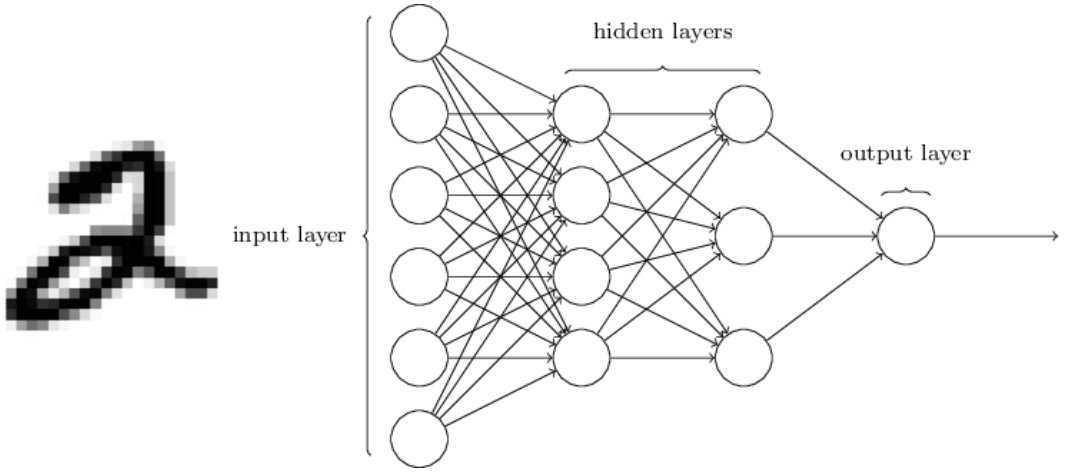
- We are given a number of “training” data instances
- E.g. images of digits, along with information about which digit the image represents
- Tasks:
  - Binary recognition: Is this a “2” or not
  - Multi-class recognition: Which digit is this? Is this a digit in the first place?

Representing the output

- If the desired output is real-valued, no special tricks are necessary
  - Scalar Output : single output neuron
    - d = scalar (real value)
  - Vector Output : as many output neurons as the dimension of the desired output
    - d = [d_1 d_2 .. d_N] (vector of real values)

- If the desired output is binary (is this a cat or not), use a simple 1/0 output
  - 1 = Yes it's a cat
  - 0 = No it's not a cat.
- For multi-class outputs: one-hot representations
  - For N classes, an N-dimensional binary vector of the form [0 0 0 1 0 0..]
  - The single "1" in the k-th position represents an instance of the kth class

Problem Setting

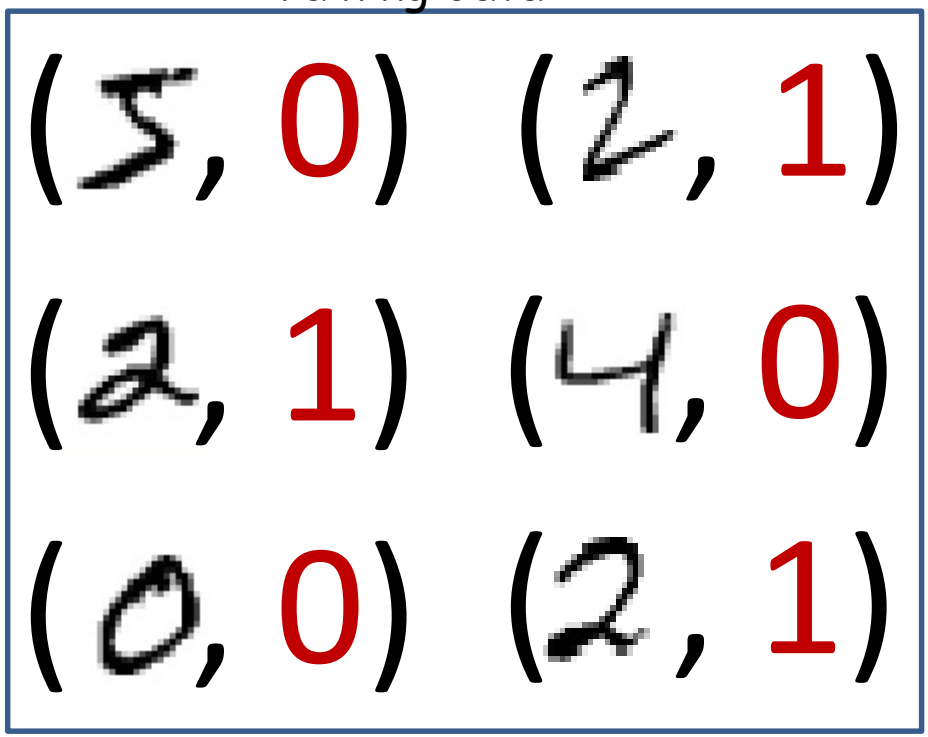


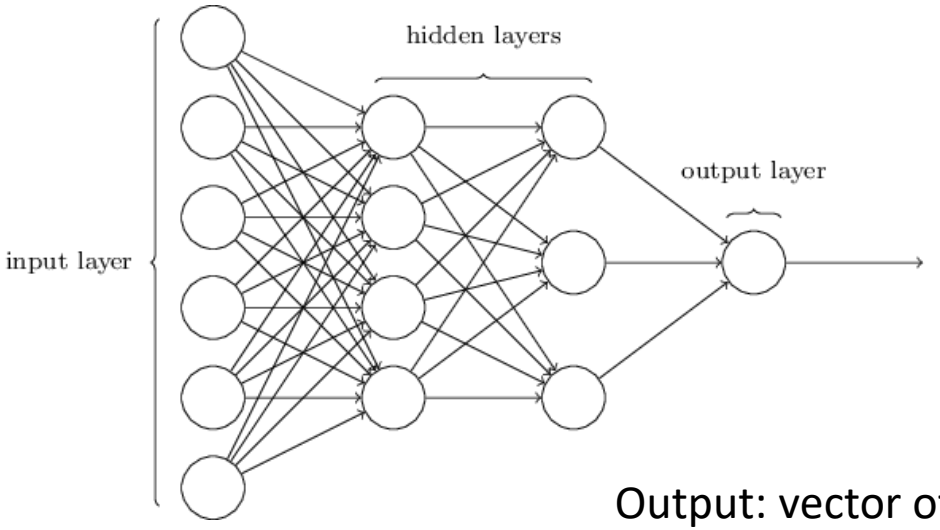
(0,1)

- We design a network with the necessary number of inputs and outputs
  - In our example we will consider a *binary* classification task
    - Is this “2” or not
    - The network has only one output
  - We will assume the structure of the network (no. of layers, no. of neurons in each layer) is given
- Challenge: how to make this network recognize “2” (e.g.)

Problem Setting

Training data



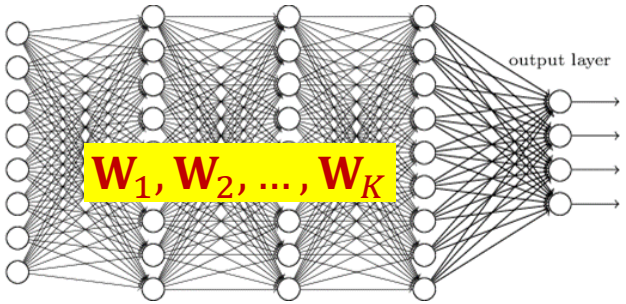


Input: vector of pixel values

Output: vector of pixel values

- Generic "training" setting:
  - Given, many positive and negative examples (training data), …
  - … learn all weights such that the network does the desired job

Recap: Procedural outline



Actual output of network:

$$\boldsymbol{Y} = \boldsymbol{g}\left(\boldsymbol{X}; \left\{w_{i,j}^{(k)} \forall i, j, k\right\}\right)$$
$$= \boldsymbol{g}(\boldsymbol{X}; \mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K)$$

Desired output of network: $\boldsymbol{d}$

Error on t-th training input:

$$\boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t}; \mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K)$$

- Given the network and input, the output is a function of the network parameters.
- For each training input, we can define an *error* between the network output and the desired output
  - This is a function of network parameters

Examples of divergence functions

- For real-valued output vectors, the L_2 divergence is popular

$$Div(Y,d) = \|Y - D\|^2 = \sum_i (Y_i - d_i)^2$$

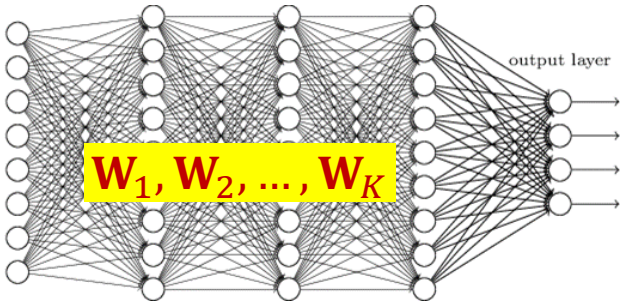
- For binary classifier with scalar output $\mathrm{Y} \in (0,1)$, d is $(0/1)$, the KL divergence is popular

$$Div(Y,d) = dlogY + (1-d)\log(1-Y)$$

- Multi-class classification: Y is a probability vector, d is a *one-hot* vector

$$Div(Y,d) = \sum_i d_i \log Y_i$$

Recap: Procedural outline



Actual output of network:

$$\boldsymbol{Y} = \boldsymbol{g}\left(\boldsymbol{X}; \left\{w_{i,j}^{(k)} \forall i, j, k\right\}\right)$$
$$= \boldsymbol{g}(\boldsymbol{X}; \mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K)$$

Desired output of network: $\boldsymbol{d}$

Error on t-th training input:

$$Div(\boldsymbol{Y_t}, \boldsymbol{d_t}; \mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K)$$

Total training error:

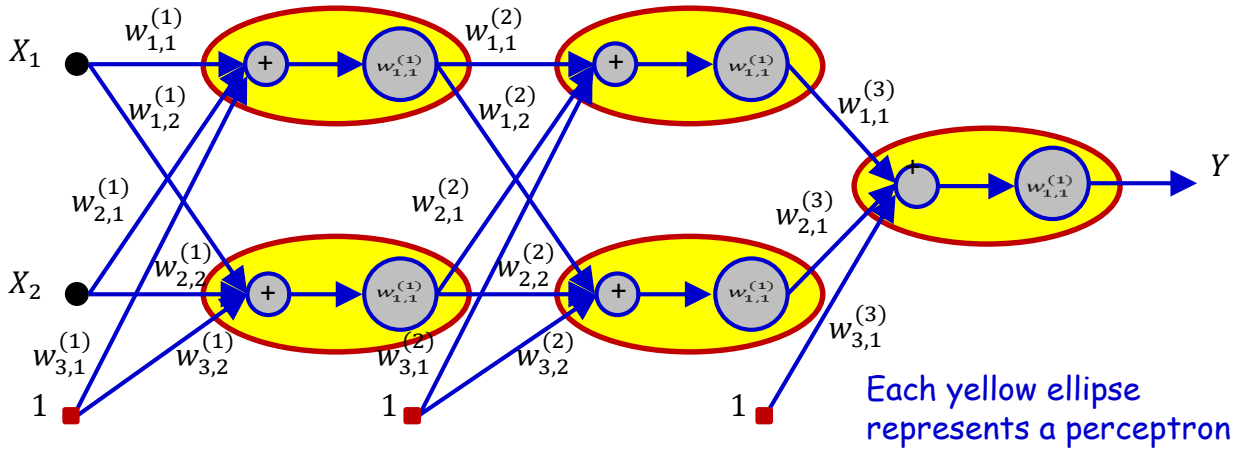
$$Err = \sum_{\boldsymbol{t}} Div(\boldsymbol{Y_t}, \boldsymbol{d_t}; \mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K)$$

- Error is a function of network parameters
- Find network parameters that minimize the total error over all training inputs
  - *With caveats..*

Gradient descent applied to MLPs

Training data

(5, 0)	(2, 1)
(2, 1)	(4, 0)
(0, 0)	(2, 1)



Total training error:

$$Err = \sum_{t} \boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t}; \mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K)$$

- Find the weights $\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K$ that minimize the total error Err

Recap: Problem Statement

- Minimize the following function

$$Err = \sum_{t} Div(\boldsymbol{Y_t}, \boldsymbol{d_t}; \mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K)$$

w.r.t $\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K$

- This is problem of function minimization
  - An instance of optimization
- We will solve this through gradient descent

Recap: Gradient Descent Algorithm

- In order to minimize any function $f(x)$ w.r.t. x
- Initialize:
  - x^0
  - $k = 0$
- While $\left|f\left(x^{k+1}\right) - f\left(x^k\right)\right| > \varepsilon$
  - $x^{k+1} = x^k - \eta^k \nabla f\left(x^k\right)^T$
  - $k = k + 1$

Training Neural Nets through Gradient Descent

Total training error:

$$Err = \sum_{t} Div(\boldsymbol{Y_t}, \boldsymbol{d_t}; \mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K)$$

- Gradient descent algorithm:
- Initialize all weights $\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K$
- Do:
  - For every layer k update:
    - $\mathbf{W}_k = \mathbf{W}_k - \eta \nabla_{\mathbf{W}_k} Err^{\mathsf{T}}$
- Until Err has converged

Training Neural Nets through Gradient Descent

Total training error:

$$Err = \sum_{t} Div(\boldsymbol{Y_t}, \boldsymbol{d_t}; \mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K)$$

- Gradient descent algorithm:
- Initialize all weights $\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K$
- Do:
  - For every layer k update:
    - $\mathbf{W}_k = \mathbf{W}_k - \eta \sum_t \nabla_{\mathbf{W}_k} Div(Y_t, d_t; \mathrm{W}_1, \mathrm{W}_2, \ldots, \mathrm{W}_K)^T$
- Until Err has converged

Training Neural Nets through Gradient Descent

Total training error:

$$Err = \sum_{t} Div(\boldsymbol{Y_t}, \boldsymbol{d_t}; \mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K)$$

- Gradient descent algorithm:
- Initialize all weights $\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K$
- Do:
  - For every layer k update:
    - $\mathbf{W}_k = \mathbf{W}_k - \eta \sum_t \nabla_{\mathbf{W}_k} Div(Y_t, d_t; \mathrm{W}_1, \mathrm{W}_2, \ldots, \mathrm{W}_K)^T$
- Until Err has converged

Computing the derivatives

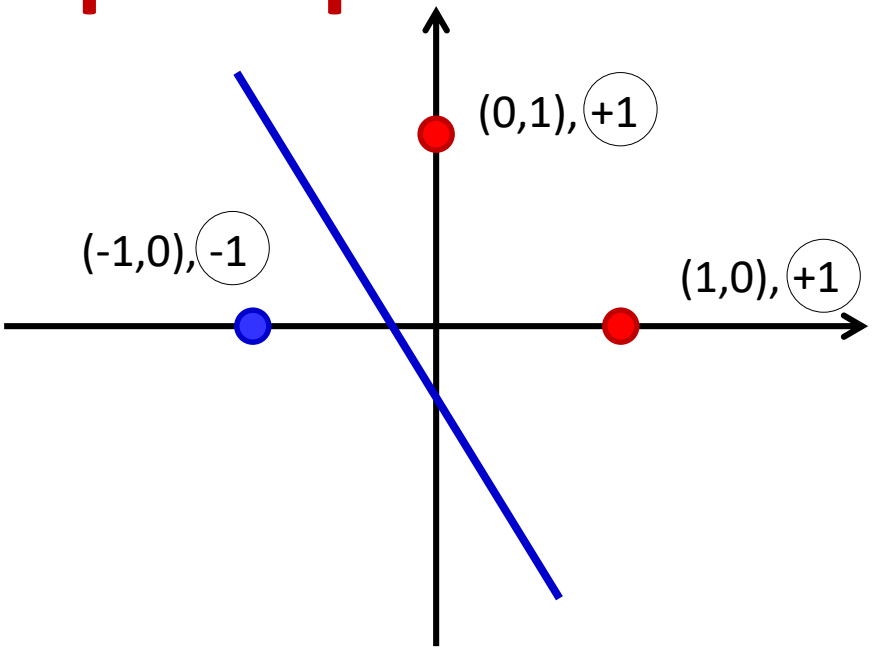
- We use back propagation to compute the derivative
- Assuming everyone knows how; will skip the details

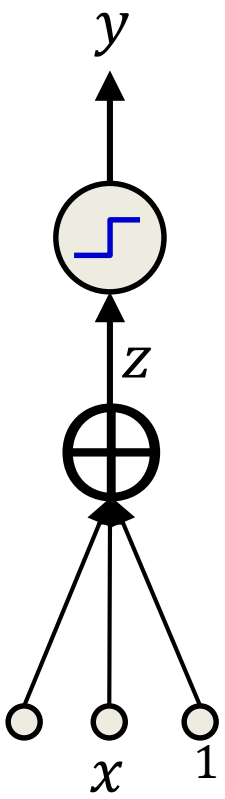
Does backprop do the right thing?

- **Is backprop always right?**
  - Assuming it actually find the global minimum of the divergence function?

- In classification problems, the classification error is a non-differentiable function of weights
- The divergence function minimized is only a *proxy* for classification error
- Minimizing divergence may not minimize classification error

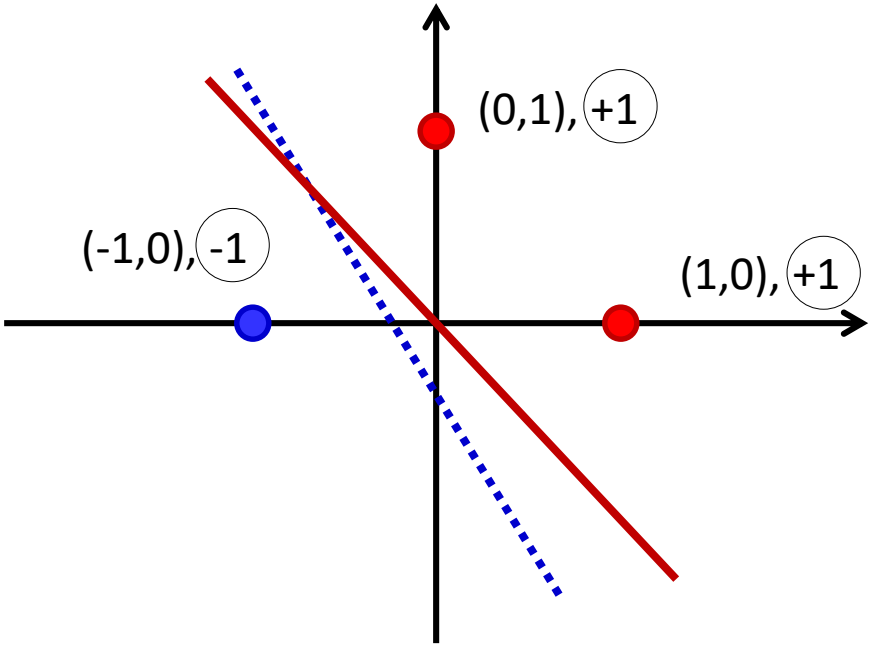
Backprop fails to separate where perceptron succeeds

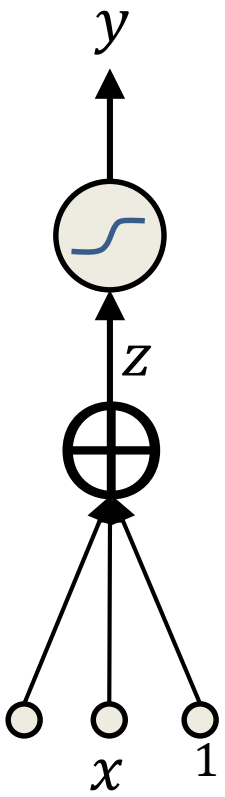




- Brady, Raghavan, Slawny, ’89
- Simple problem, 3 training instances, single neuron
- Perceptron training rule trivially find a perfect solution

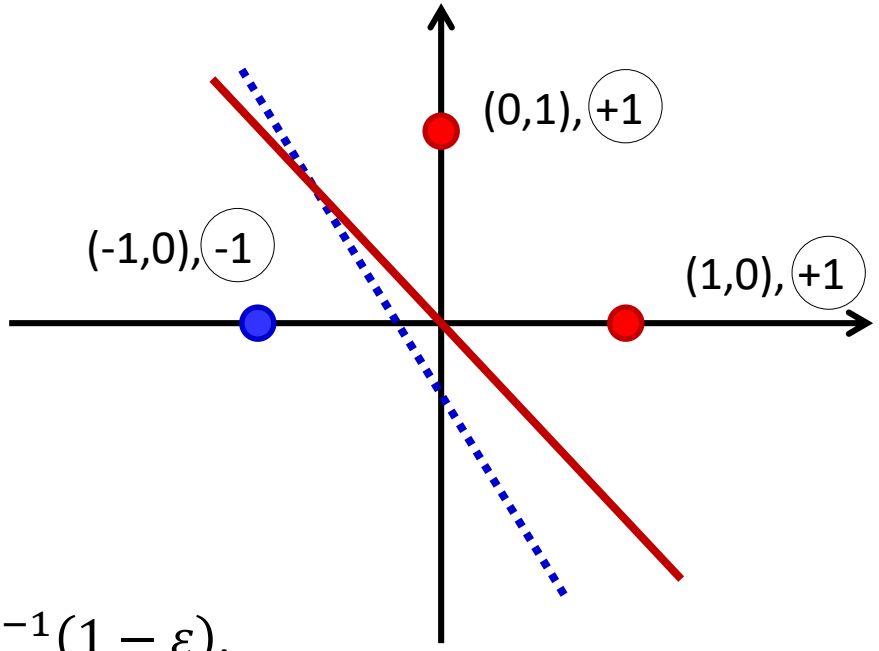
Backprop vs. Perceptron

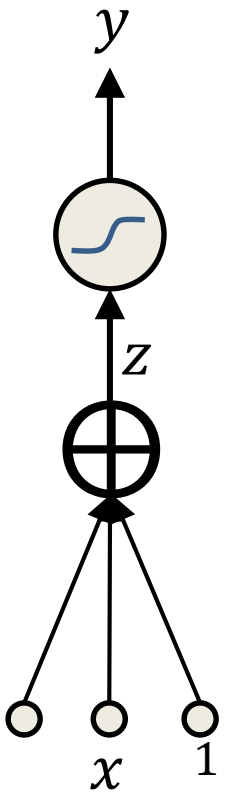




- Back propagation using logistic function and L_2 divergence $(Div = (y-d)^2)$
- Unique minimum trivially proved to exist, Backpropagation finds it

Unique solution exists



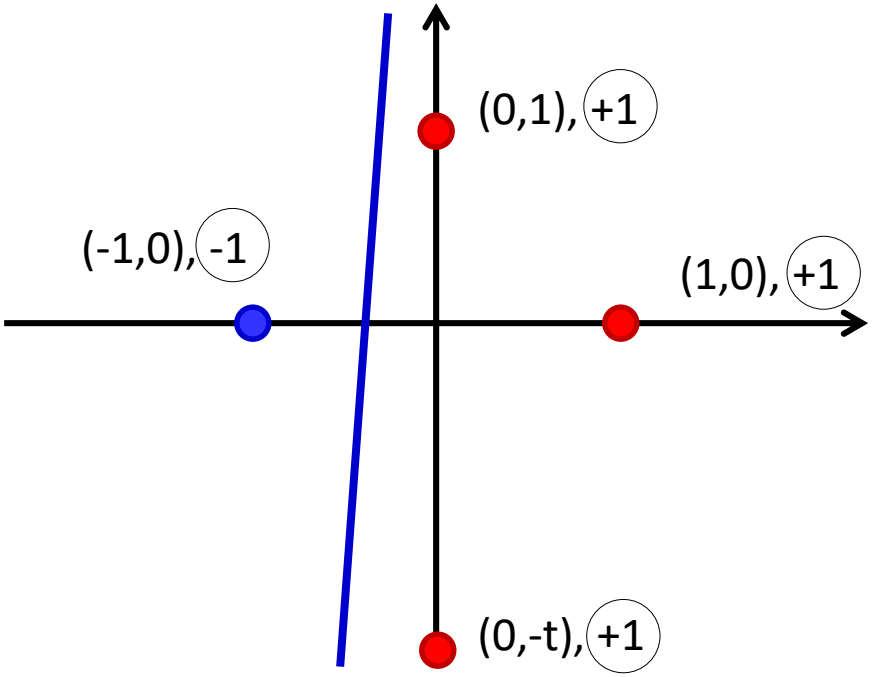


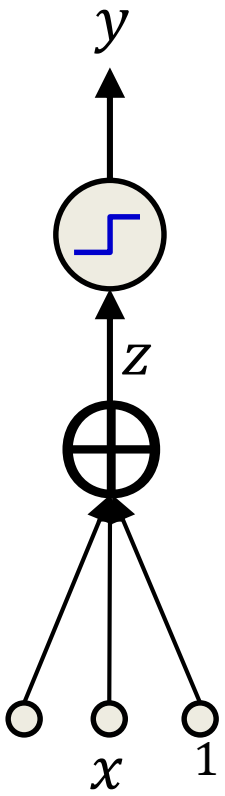
- Let $u = f^{-1}(1-\varepsilon)$.
- From the three points we get three independent equations:

$$w_x.1 + w_y.0 + b = u$$
$$w_x.0 + w_y.1 + b = u$$
$$w_x.-1 + w_y.0 + b = -u$$

- Unique solution $(w_x = u, w_x = u, b = 0)$ exists
  - represents a unique line regardless of the value of u

Backprop vs. Perceptron

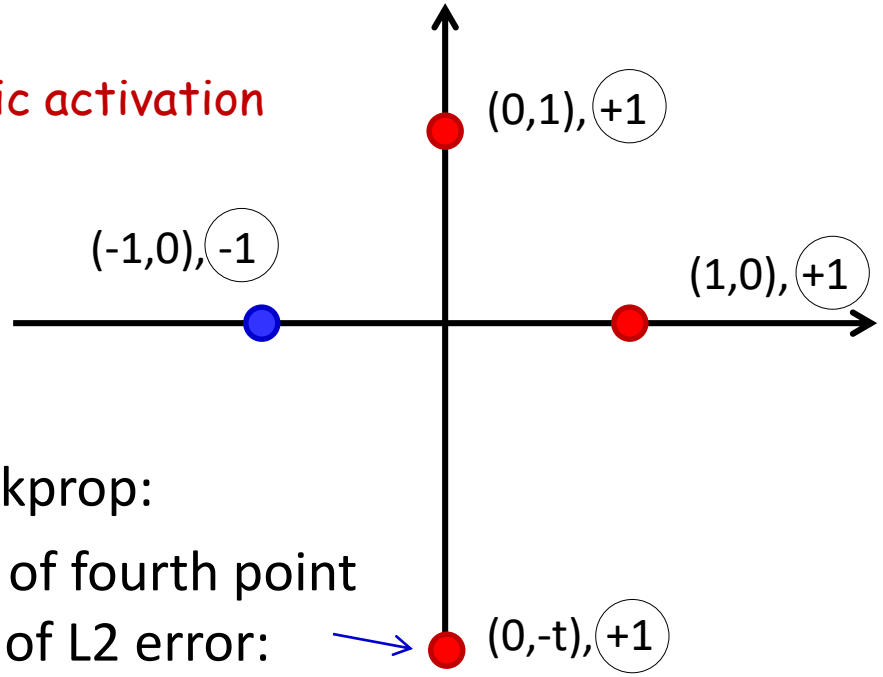


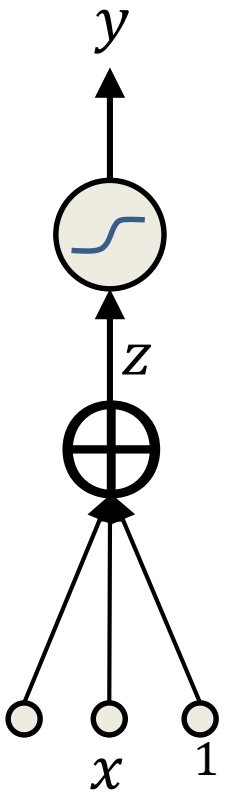


- Now add a fourth point
- t is very large (point near $-\infty$)
- Perceptron trivially finds a solution

Backprop

Notation:
$y = \sigma(z)$ = logistic activation





- Consider backprop:
- Contribution of fourth point to derivative of L2 error:

$$div_4 = \left(1 - \sigma(-w_y t + b)\right)^2$$

$$\frac{d\, div_4}{dw_y} = 2\left(1 - \sigma(-w_y t + b)\right)\sigma'(-w_y t + b)t$$

$$\frac{d\, div_4}{db} = -2\left(1 - \sigma(-w_y t + b)\right)\sigma'(-w_y t + b)$$

Backprop

Notation:
$y = \sigma(z)$ = logistic activation

$$div_4 = \left(1 - \sigma\left(-w_y t + b\right)\right)^2$$

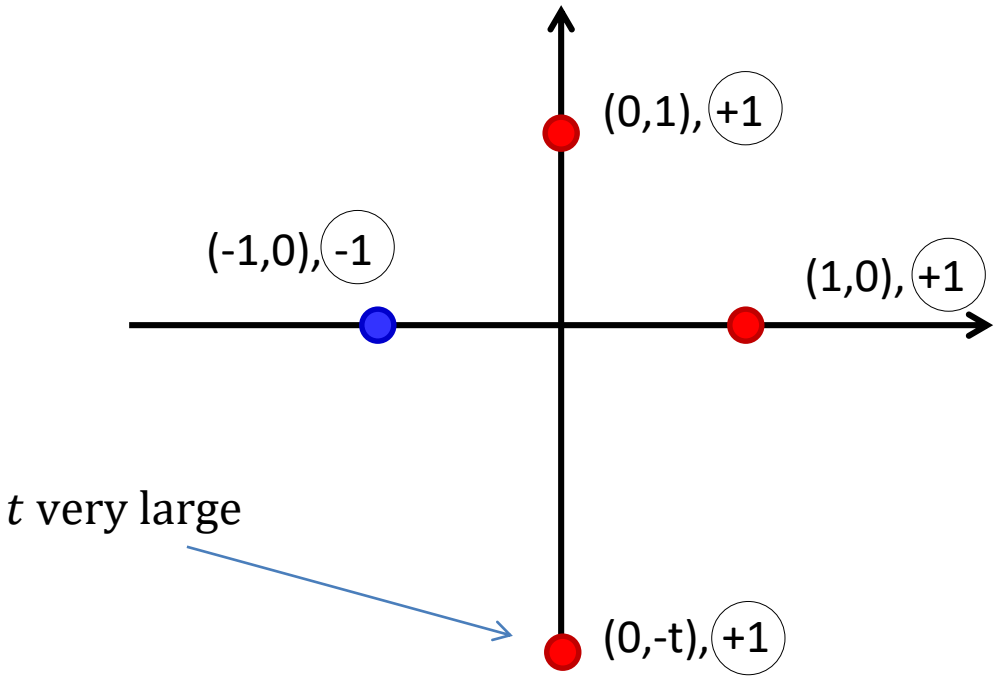
$$\frac{d\ div_4}{dw_y} = 2\left(1 - \sigma\left(-w_y t + b\right)\right)\sigma'\left(-w_y t + b\right)t$$

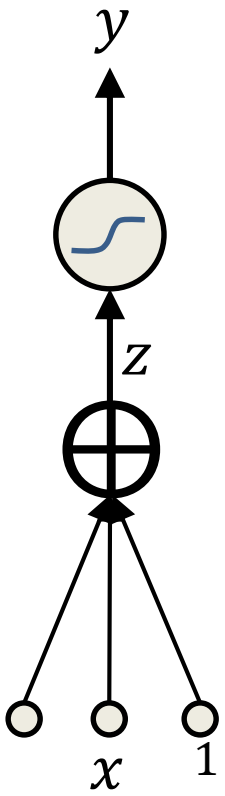
$$\frac{d\ div_4}{db} = 2\left(1 - \sigma\left(-w_y t + b\right)\right)\sigma'\left(-w_y t + b\right)t$$

- For very large positive t, $|w_y| > \epsilon$ (where $\mathbf{w} = [w_x, w_y, b]$)
- $\left(1 - \sigma\left(-w_y t + b\right)\right) \rightarrow 1$ as $t \rightarrow \infty$
- $\sigma'\left(-w_y t + b\right) \rightarrow 0$ *exponentially* as $t \rightarrow \infty$
- Therefore, for very large positive t

$$\frac{d\ div_4}{dw_y} = \frac{d\ div_4}{db} = 0$$

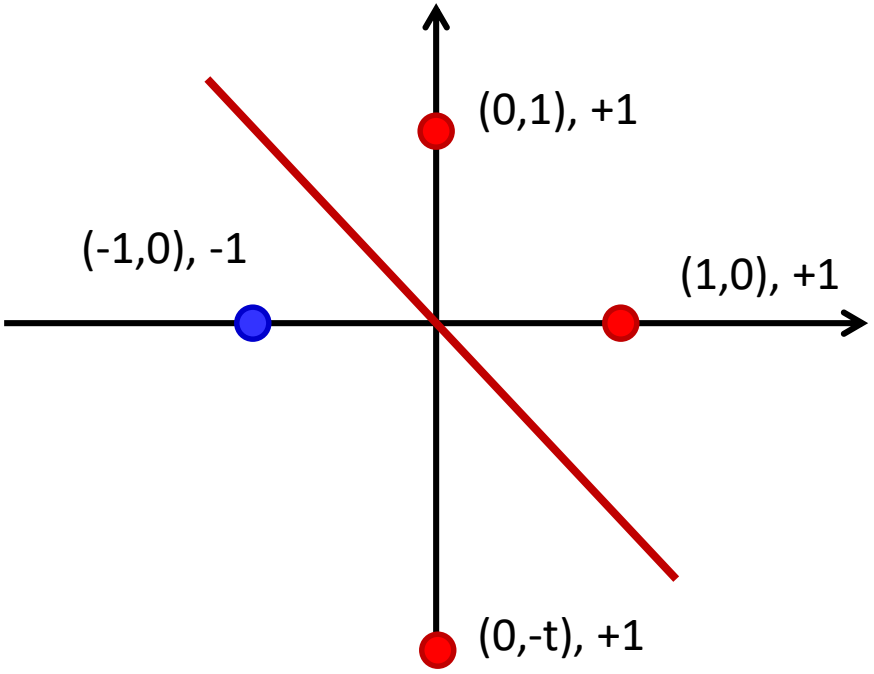
Backprop

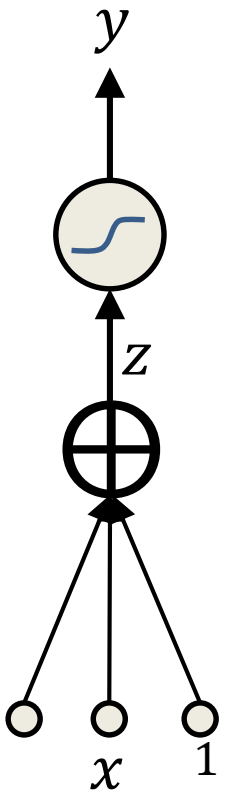




- The fourth point at $(0, -t)$ does not change the gradient of the L_2 divergence anywhere
- The optimum solution is the same as the optimum solution with only the first three points!

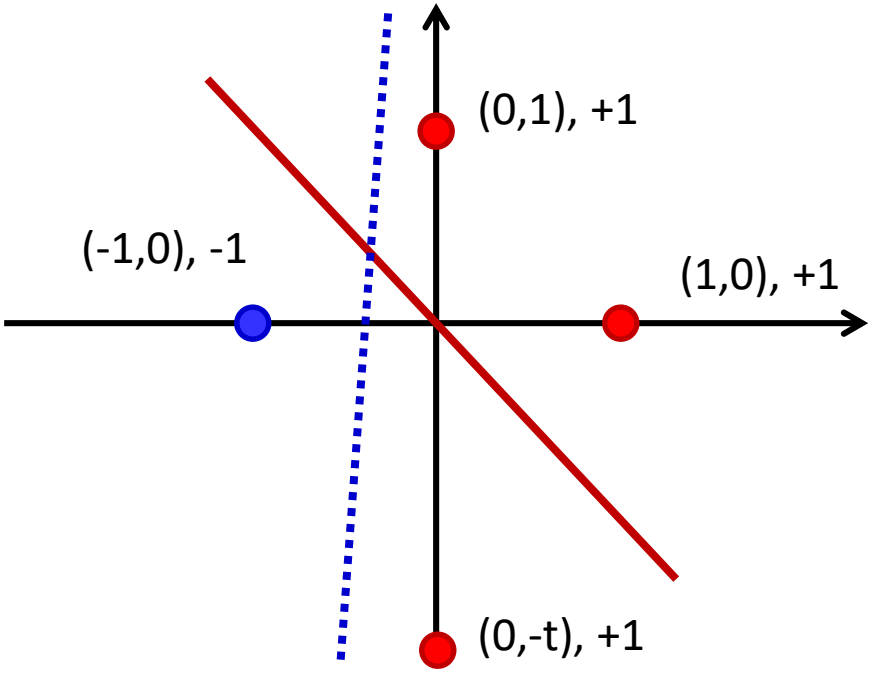
Backprop

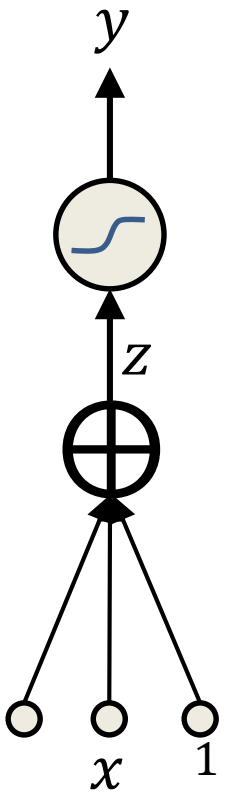




- Global optimum solution found by backprop
- Does not separate the points *even though the points are linearly separable!*

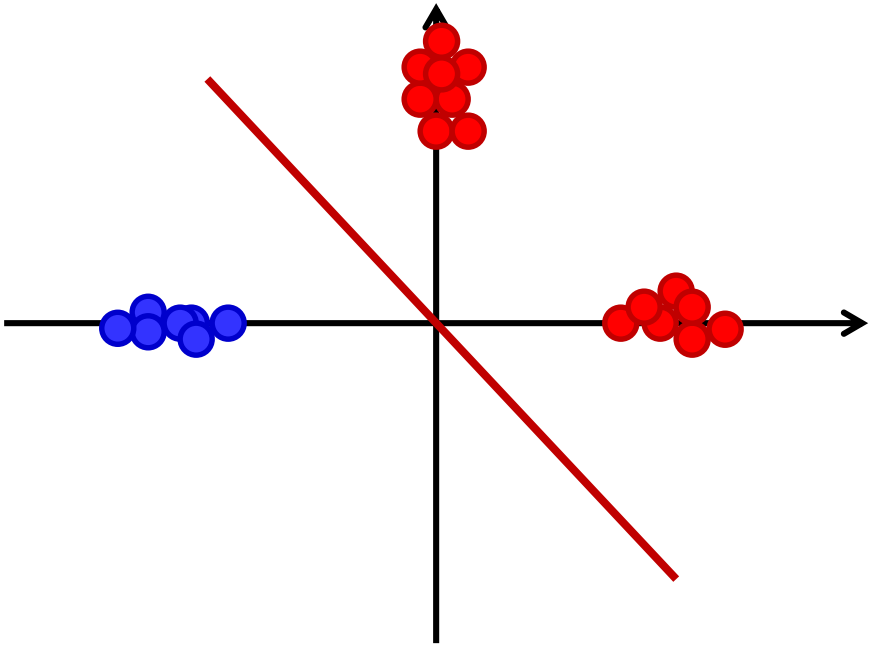
Backprop

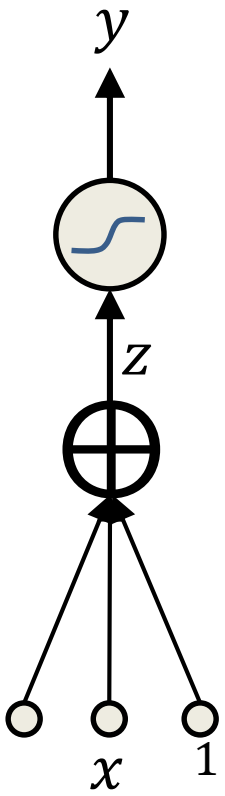




- Global optimum solution found by backprop
- Does not separate the points *even though the points are linearly separable!*
- Compare to the perceptron: *Backpropagation fails to separate where the perceptron succeeds*

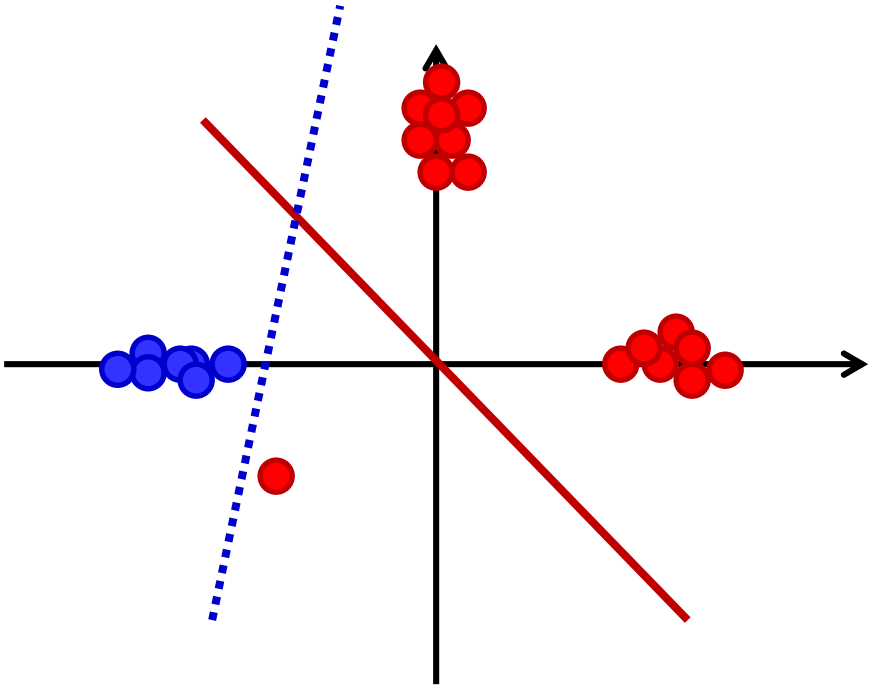
A more complex problem

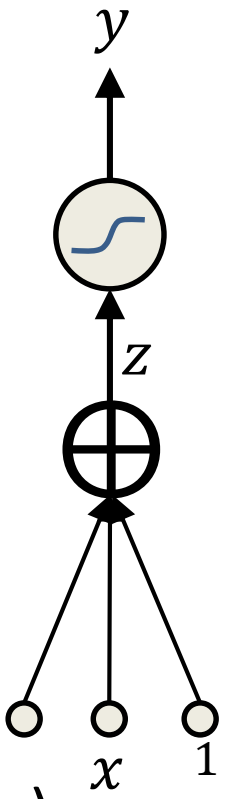




- *Several* linearly separable training examples
- Simple setup: both backprop and perceptron algorithms find solutions

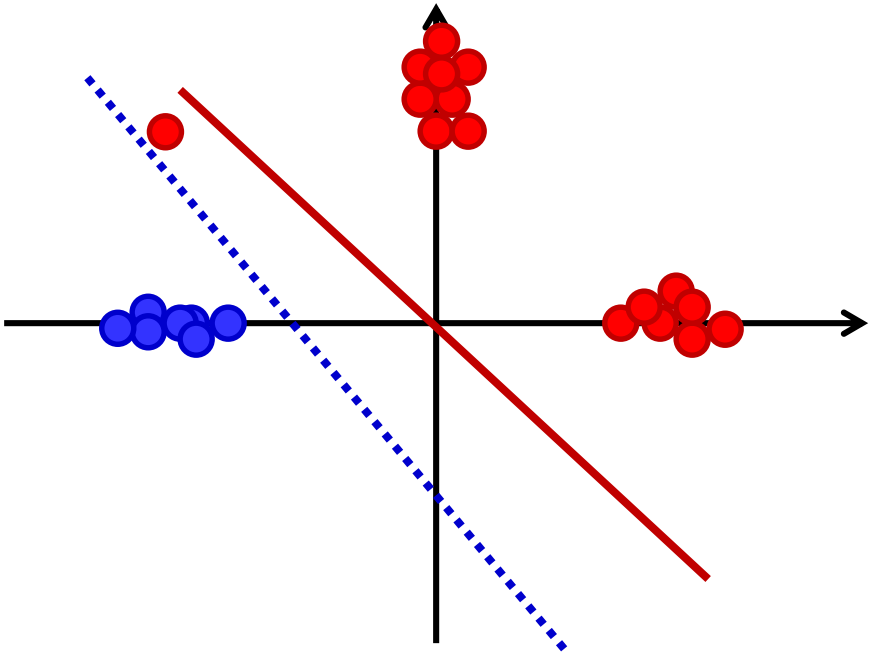
A more complex problem

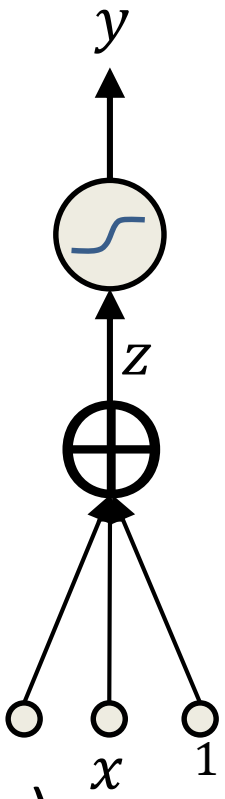




- Adding a “spoiler” (or a small number of spoilers)
  - Perceptron finds the linear separator,
  - Backprop does not find a separator
    - A single additional input does not change the loss function significantly

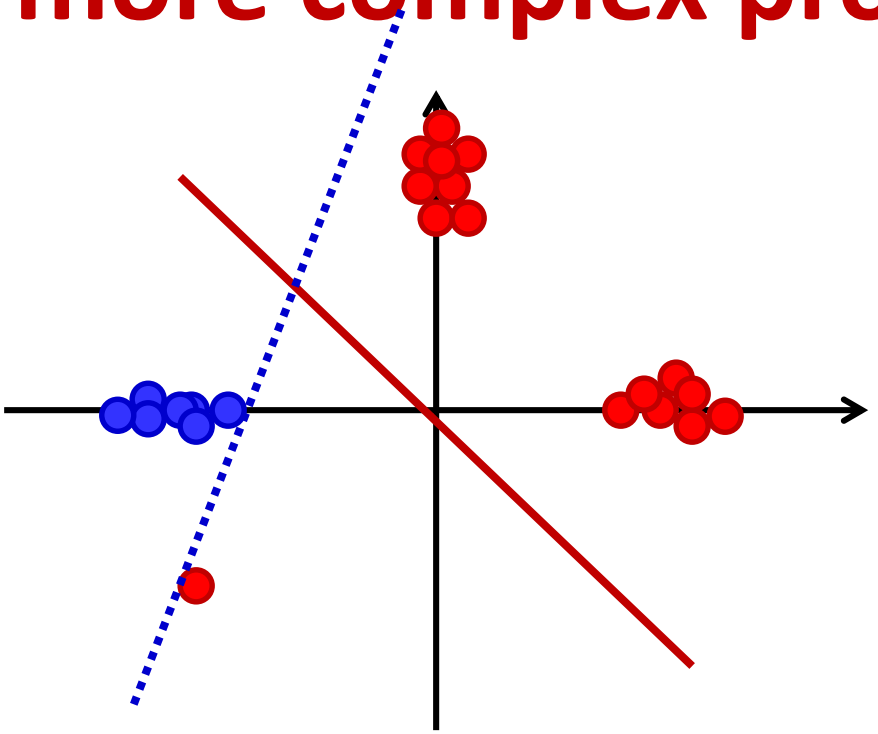
A more complex problem

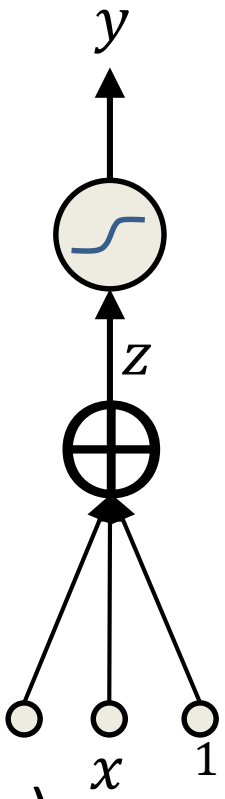




- Adding a “spoiler” (or a small number of spoilers)
  - Perceptron finds the linear separator,
  - Backprop does not find a separator
    - A single additional input does not change the loss function significantly

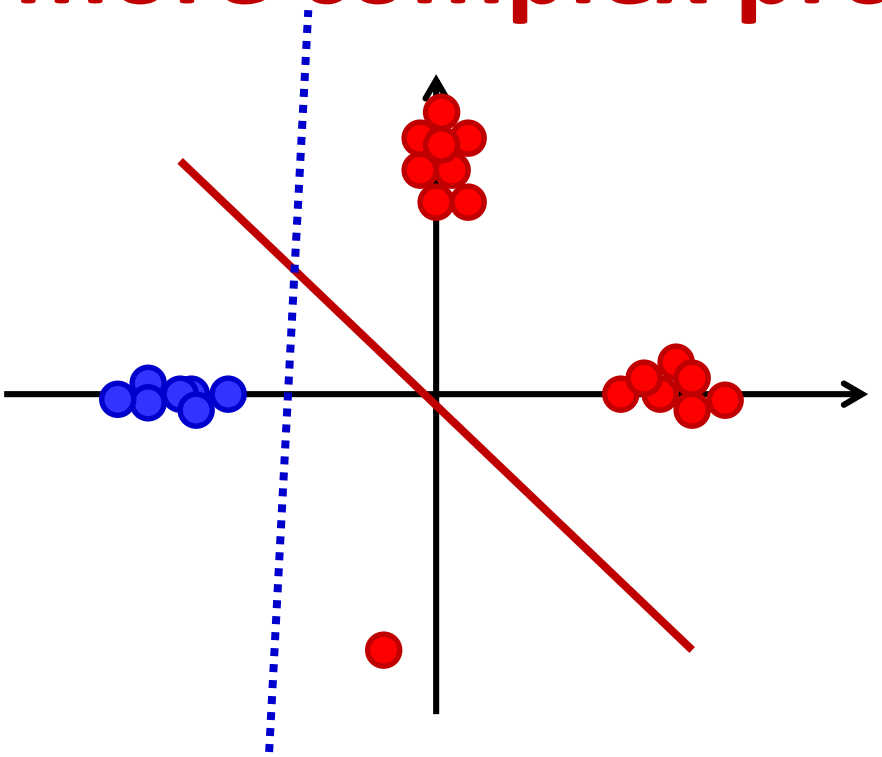
A more complex problem

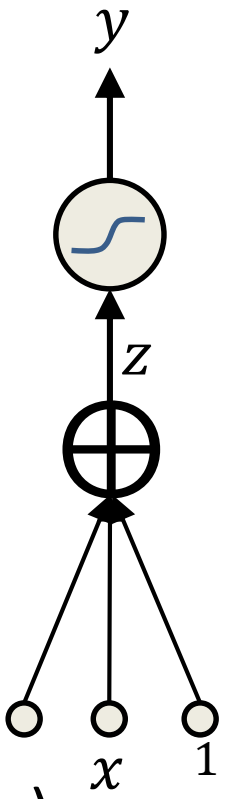




- Adding a “spoiler” (or a small number of spoilers)
  - Perceptron finds the linear separator,
  - Backprop does not find a separator
    - A single additional input does not change the loss function significantly

A more complex problem



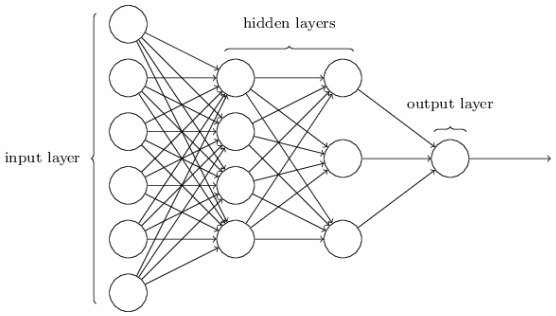


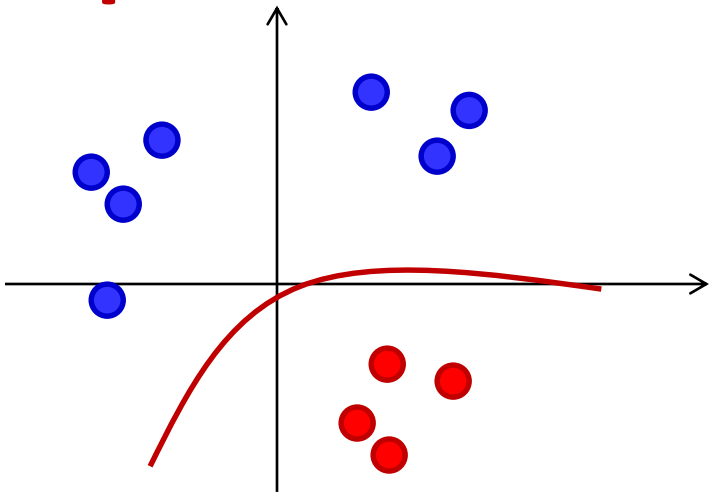
- Adding a “spoiler” (or a small number of spoilers)
  - Perceptron finds the linear separator,
  - Backprop does not find a separator
    - A single additional input does not change the loss function significantly

So what is happening here?

- The perceptron may change greatly upon adding just a single new training instance
  - But it fits the training data well
  - The perceptron rule has *low bias*
    - Makes no errors if possible
  - But high variance
    - Swings wildly in response to small changes to input
- Backprop is minimally changed by new training instances
  - Prefers consistency over perfection
  - It is a *low-variance* estimator, at the potential cost of bias

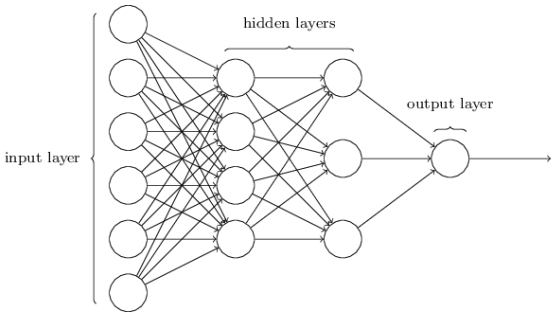
Backprop fails to separate even when possible

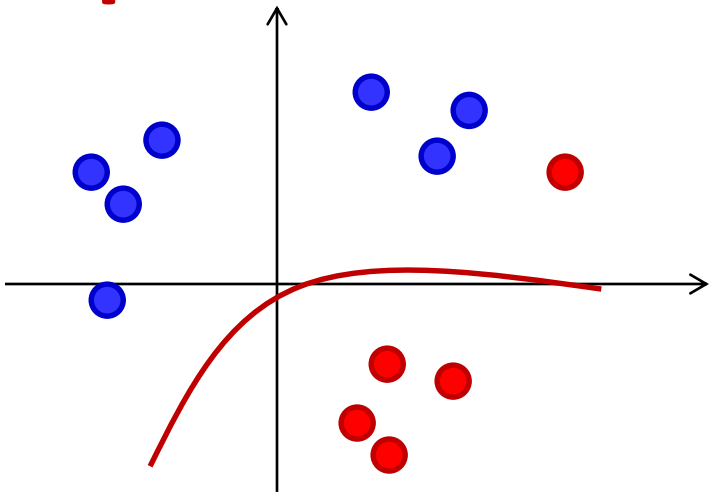




- This is not restricted to single perceptrons
- In an MLP the lower layers “learn a representation” that enables linear separation by higher layers
  - More on this later
- Adding a few “spoilers” will not change their behavior

Backprop fails to separate even when possible





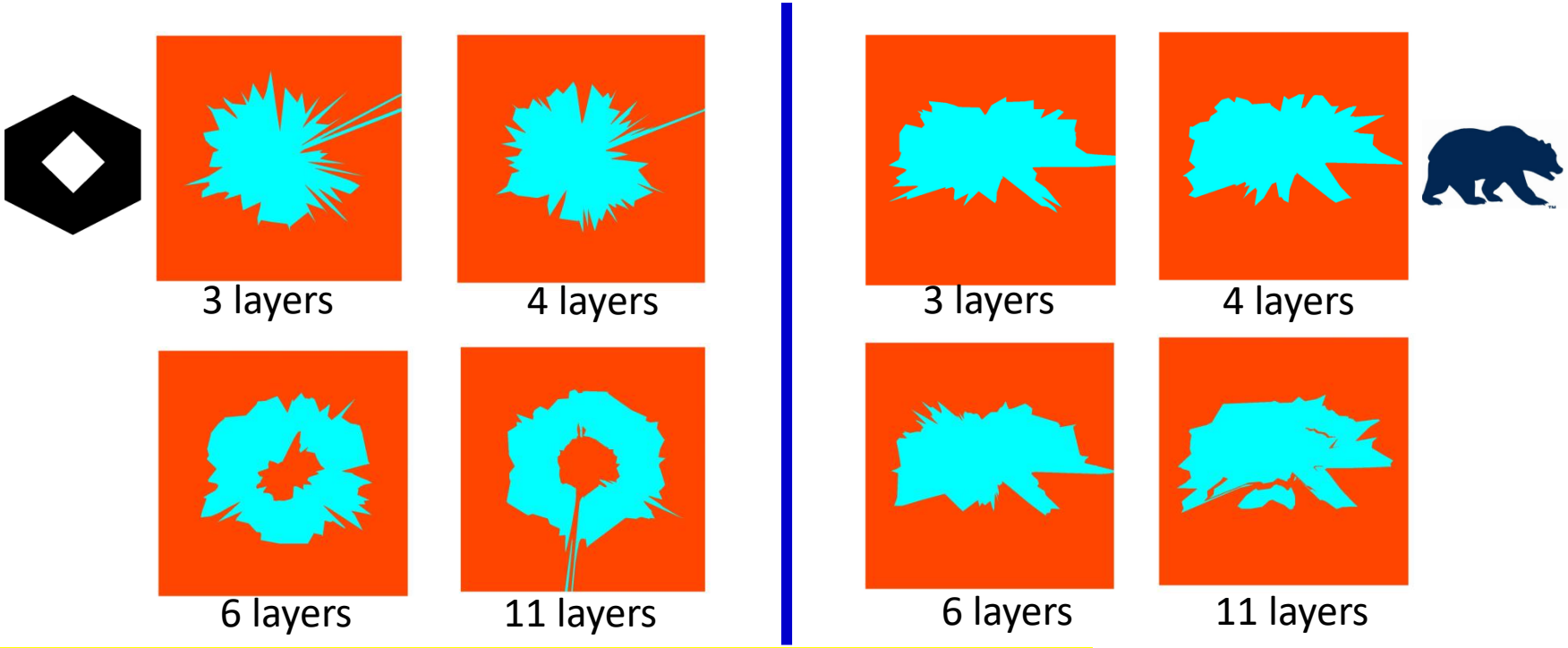
- This is not restricted to single perceptrons
- In an MLP the lower layers “learn a representation” that enables linear separation by higher layers
  - More on this later
- Adding a few “spoilers” will not change their behavior

Backpropagation

- Backpropagation will often not find a separating solution ***even though the solution is within the class of functions learnable by the network***

- This is because the separating solution is not an optimum for the loss function
- One resulting benefit is that a backprop-trained neural network classifier has lower variance than an optimal classifier for the training data

Variance and Depth



- Dark figures show desired decision boundary (2D)
  - 1000 training points, 660 hidden neurons
  - Network heavily overdesigned even for shallow nets
- **Anecdotal: Variance decreases with**
  - Depth
  - Data

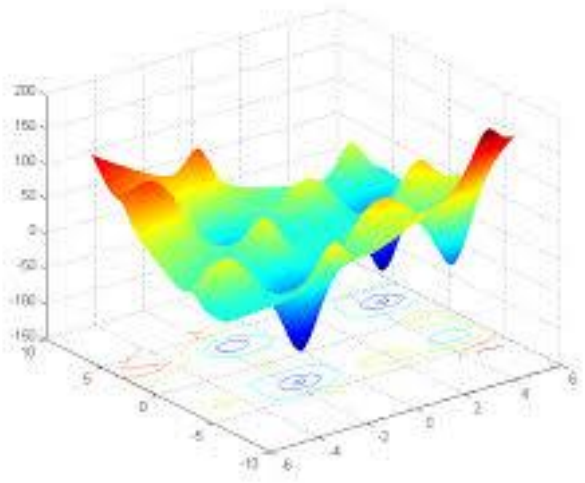
10000 training instances

Convergence

- How fast does it converge?
  - And where?

The Error Surface

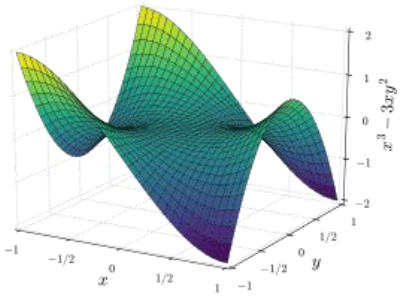
- The example (and statements) earlier assumed the loss objective had a single global optimum that could be found
  - Statement about variance is assuming global optimum

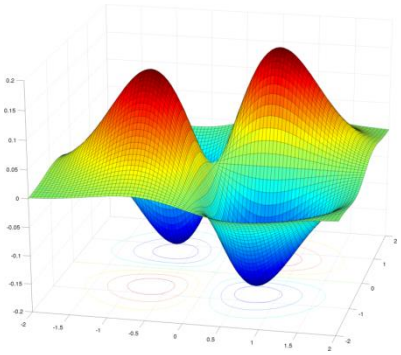


- What about local optima

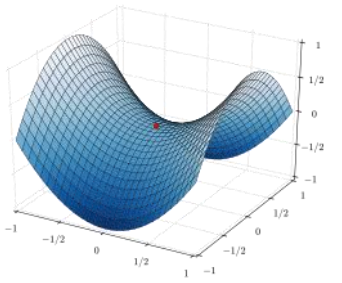
The Error Surface

- **Popular hypothesis**:
  - In large networks, saddle points are far more common than local minima
    - Frequency exponential in network size
  - Most local minima are equivalent
    - And close to global minimum
  - This is not true for small networks





- **Saddle point:** A point where
  - The slope is zero
  - The surface increases in some directions, but decreases in others
    - Some of the Eigenvalues of the Hessian are positive; others are negative
  - Gradient descent algorithms like saddle points



The Controversial Error Surface

- **Baldi and Hornik (89), "***Neural Networks and Principal Component Analysis: Learning from Examples Without Local Minima*" : An MLP with a *single* hidden layer has only saddle points and no local Minima

- **Dauphin et. al (2015),** "*Identifying and attacking the saddle point problem in high-dimensional non-convex optimization*" : An exponential number of saddle points in large networks

- **Chomoranksa et. al (2015)**, "*The loss surface of multilayer networks*" : For large networks, most local minima lie in a band and are equivalent
  - Based on analysis of spin glass models

- **Swirscz et. al. (2016)**, "Local minima in training of deep networks", In networks of finite size, trained on finite data, you *can* have horrible local minima
- Watch this space…

Story so far

- Neural nets can be trained via gradient descent that minimizes a loss function
- Backpropagation can be used to derive the derivatives of the loss
- Backprop *is not guaranteed* to find a "true" solution, even if it exists, and lies within the capacity of the network to model
  - The optimum for the loss function may not be the "true" solution
- For large networks, the loss function may have a large number of unpleasant saddle points
  - Which backpropagation may find

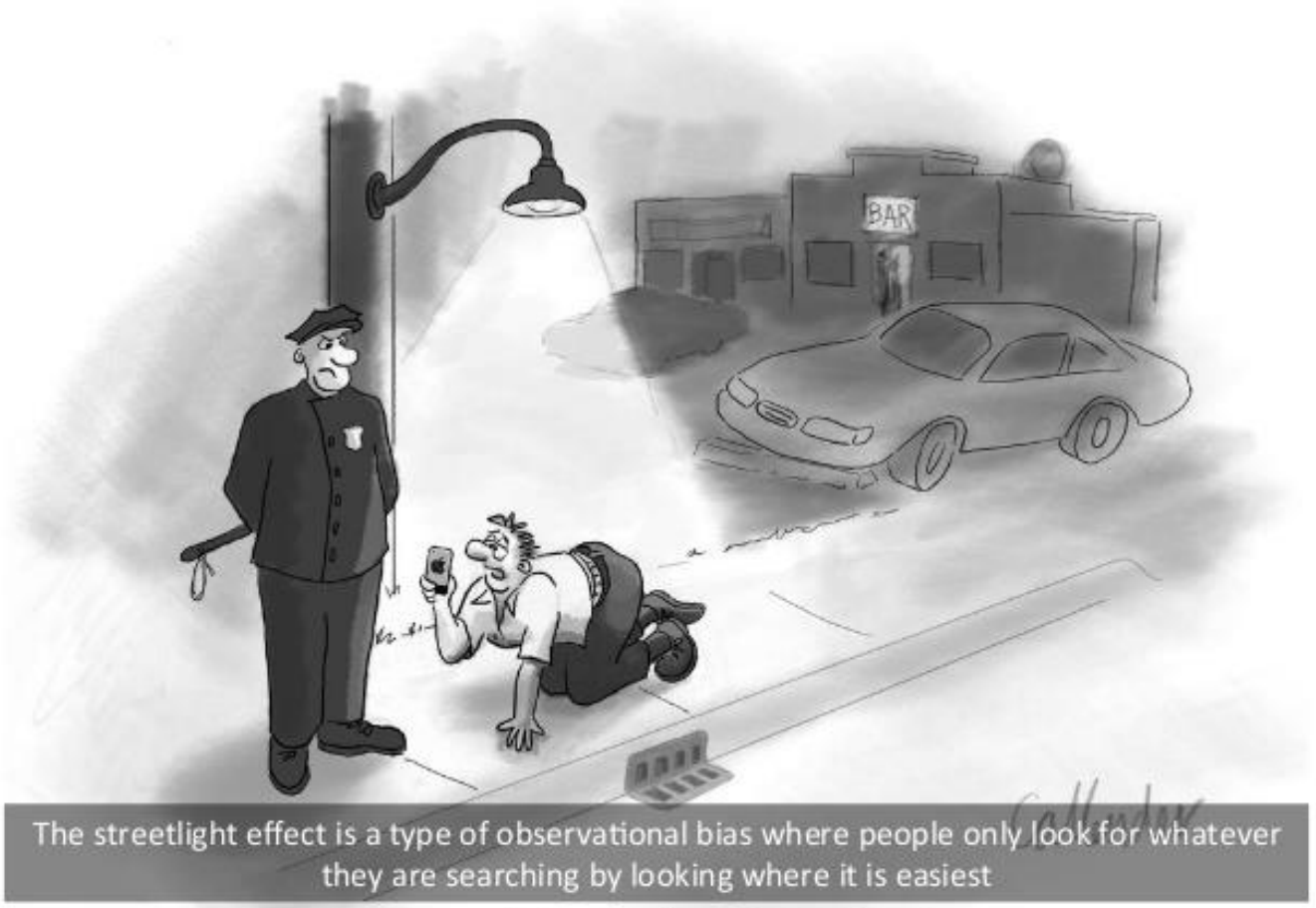
Convergence

- In the discussion so far we have assumed the training arrives at a local minimum

- Does it always converge?
- How long does it take?

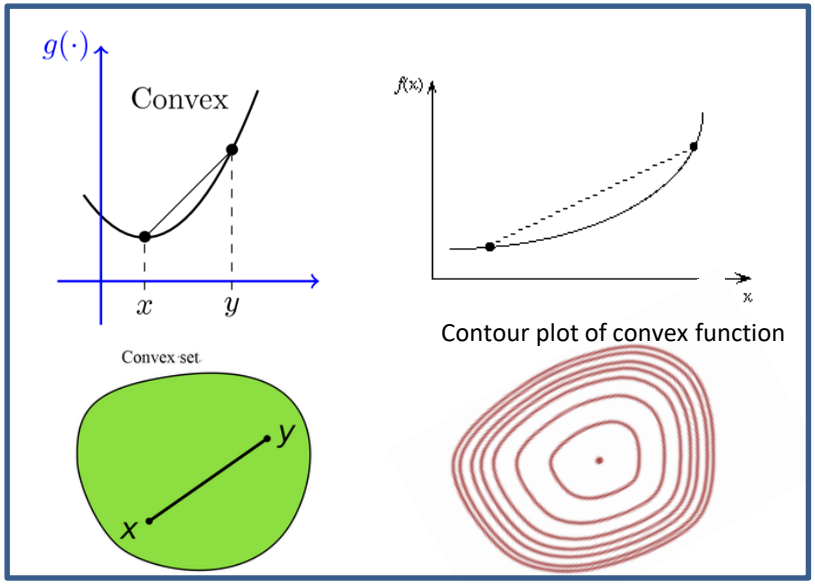
- Hard to analyze for an MLP, so lets look at convex optimization instead

A quick tour of (convex) optimization

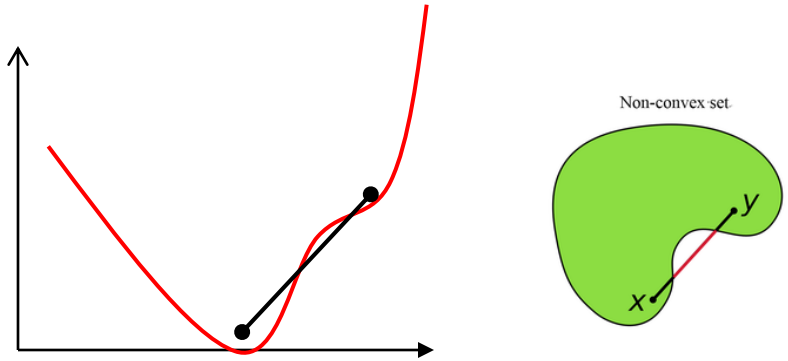


Convex Loss Functions

- A surface is “convex” if it is continuously curving upward
  - We can connect any two points above the surface without intersecting it
  - Many mathematical definitions that are equivalent

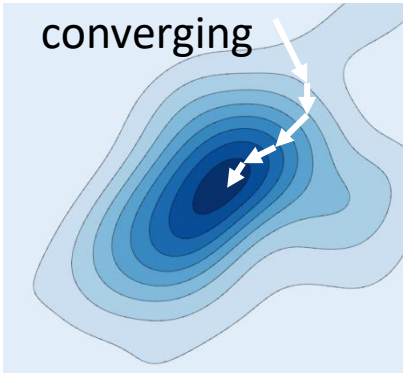


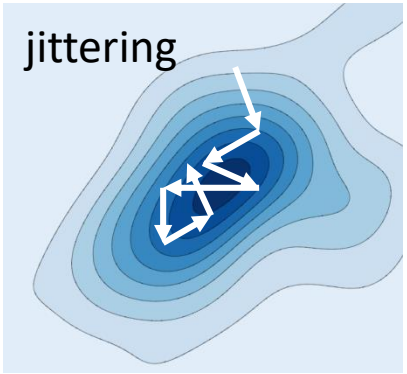
- Caveat: Neural network error surface is generally not convex
  - Streetlight effect

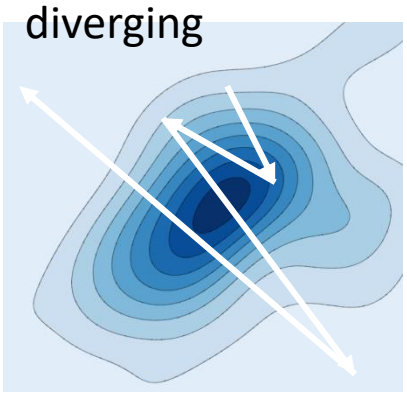


Convergence of gradient descent

- An iterative algorithm is said to *converge* to a solution if the weight updates arrive at a fixed point
  - Where the gradient is 0 and further updates do not change the weight
- The algorithm may not actually converge
  - It may jitter around the local minimum
  - It may even diverge
- Conditions for convergence?





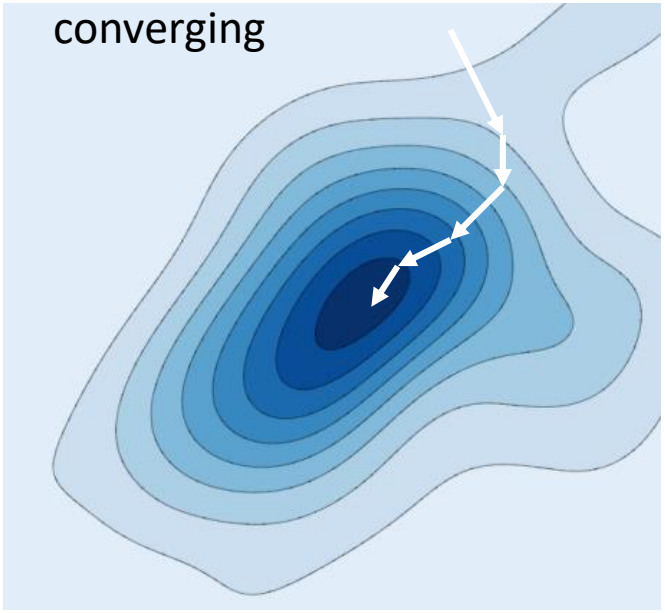


Convergence and convergence rate

- Convergence rate: How fast the iterations arrive at the solution
- Generally quantified as

$$R = \frac{\left|f\left(x^{(k+1)}\right) - f(x^*)\right|}{\left|f(x^{(k)}) - f(x^*)\right|}$$

  - $x^{(k+1)}$ is the k-th iteration
  - x^* is the optimal value of x



- If R is a constant (or upper bounded), the convergence is *linear*
  - In reality, its arriving at the solution exponentially fast

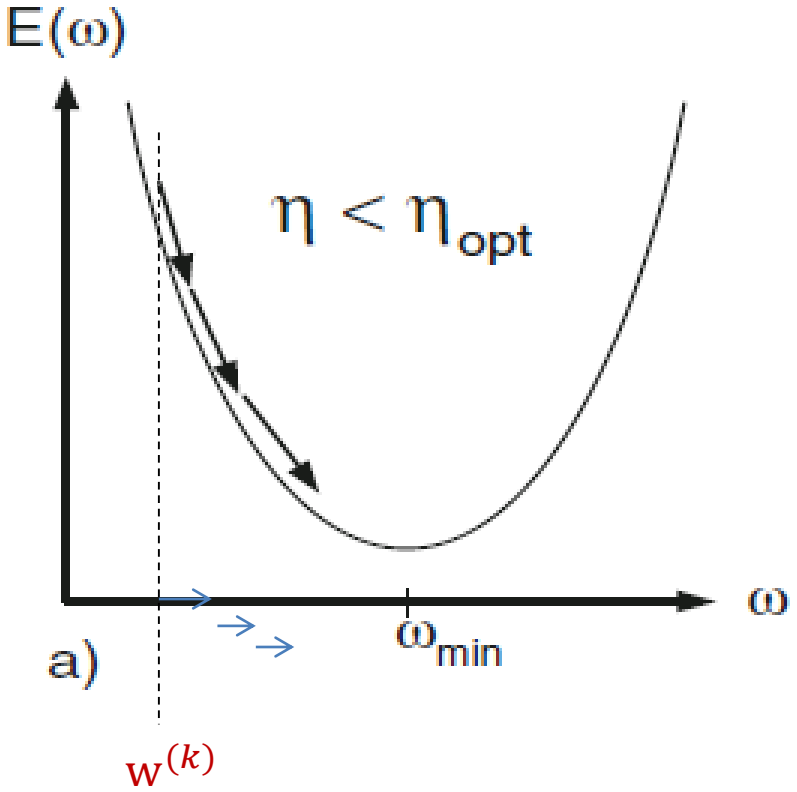
$$\left|f\left(x^{(k)}\right) - f(x^*)\right| = c^k \left|f\left(x^{(0)}\right) - f(x^*)\right|$$

Convergence for quadratic surfaces

$$Minimize\ E = \frac{1}{2}aw^2 + bw + c$$

$$\mathrm{w}^{(k+1)} = \mathrm{w}^{(k)} - \eta \frac{dE\left(\mathrm{w}^{(k)}\right)}{d\mathrm{w}}$$

Gradient descent with fixed step size η to estimate *scalar* parameter w

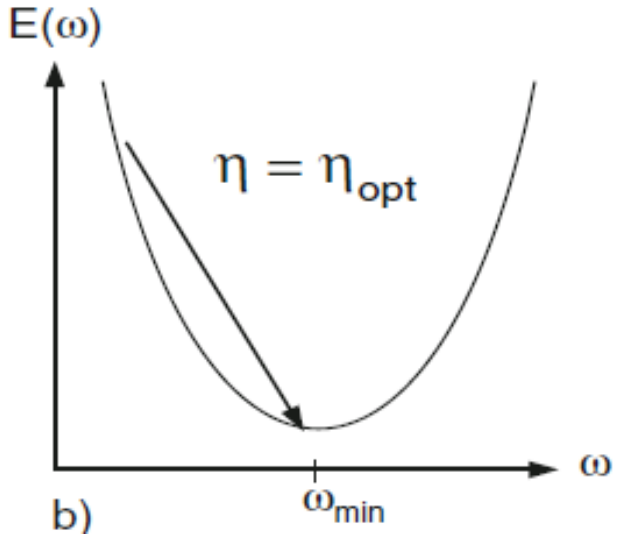


- Gradient descent to find the optimum of a quadratic, starting from $\mathrm{w}^{(k)}$
- Assuming fixed step size η
- What is the optimal step size η to get there fastest?

Convergence for quadratic surfaces

$$E = \frac{1}{2}aw^2 + bw + c$$

$$\mathrm{w}^{(k+1)} = \mathrm{w}^{(k)} - \eta \frac{dE\left(\mathrm{w}^{(k)}\right)}{d\mathrm{w}}$$



- Any quadratic objective can be written as

$$E = E\left(\mathrm{w}^{(k)}\right) + b\left(w - \mathrm{w}^{(k)}\right) + \frac{1}{2}a\left(w - \mathrm{w}^{(k)}\right)^2$$

  - Taylor expansion

- Minimizing w.r.t w, we get

$$w_{min} = \mathrm{w}^{(k)} - a^{-1}b$$

- Note:

$$\frac{dE\left(\mathrm{w}^{(k)}\right)}{d\mathrm{w}} = b$$

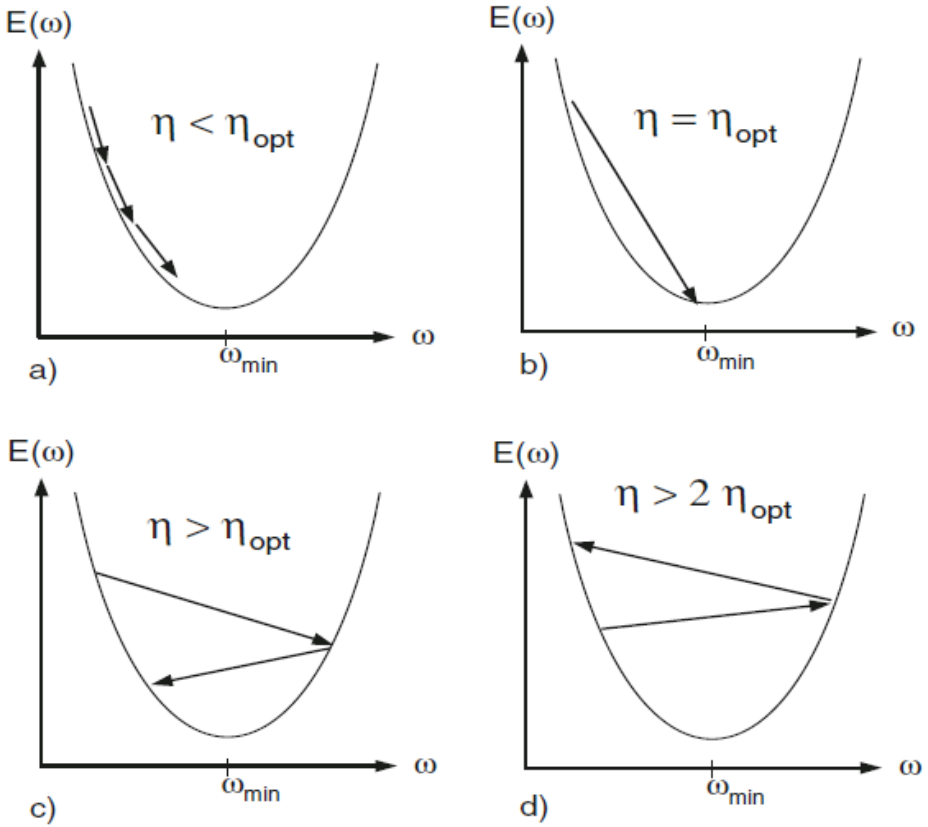
- Comparing to the gradient descent rule, we see that we can arrive at the optimum in a single step using the optimum step size

$$\eta_{opt} = a^{-1}$$

With non-optimal step size

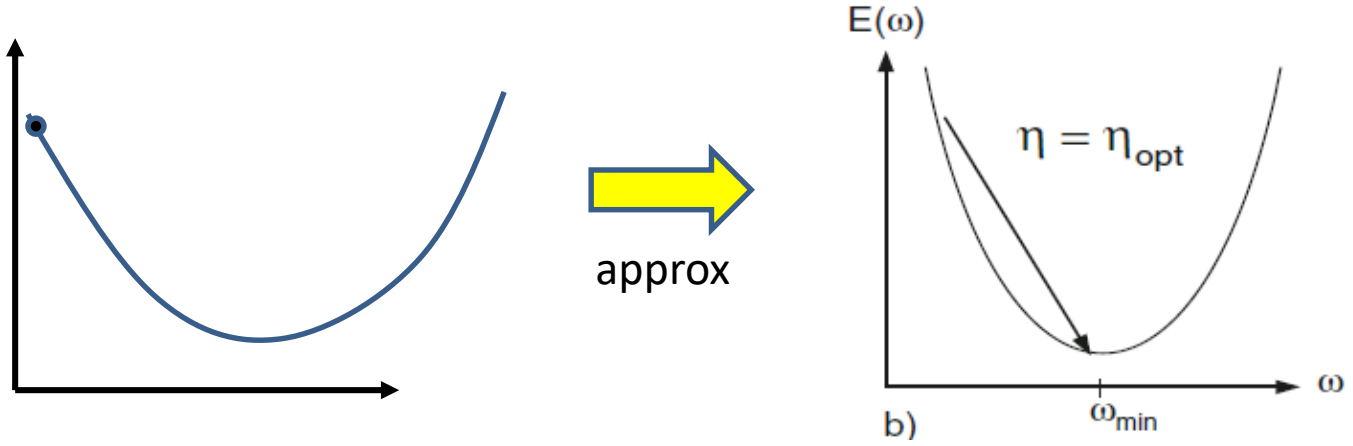
$$w^{(k+1)} = w^{(k)} - \eta \frac{dE\left(w^{(k)}\right)}{dw}$$

Gradient descent with fixed step size η to estimate scalar parameter w



- For $\eta < \eta_{opt}$ the algorithm will converge monotonically
- For $2\eta_{opt} > \eta > \eta_{opt}$ we have oscillating convergence
- For $\eta \geq \eta_{opt}$ we get divergence

For generic differentiable convex objectives



- Any differentiable objective $E(w)$ can be approximated as

$$E \approx E\left(\mathrm{w}^{(k)}\right) + \left(w - \mathrm{w}^{(k)}\right)\frac{dE\left(\mathrm{w}^{(k)}\right)}{dw} + \frac{1}{2}\left(w - \mathrm{w}^{(k)}\right)^2 \frac{d^2E\left(\mathrm{w}^{(k)}\right)}{dw^2} + \cdots$$

  - Taylor expansion
- Using the same logic as before, we get

$$\eta_{opt} = \left(\frac{d^2E\left(\mathrm{w}^{(k)}\right)}{dw^2}\right)^{-1}$$

- We can get divergence if $\eta \geq 2\eta_{opt}$

For functions of *multivariate* inputs

$E = g(\mathbf{w})$, $\mathbf{w}$ is a vector $\mathbf{w} = [w_1, w_2, \ldots, w_N]$

- Consider a simple quadratic convex (paraboloid) function

$$E = \frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{w}^T\mathbf{b} + c$$

  - Since $E^T = E$ (E is scalar), $\mathbf{A}$ can always be made symmetric
    - For **convex** E, $\mathbf{A}$ is always positive definite, and has positive eigenvalues

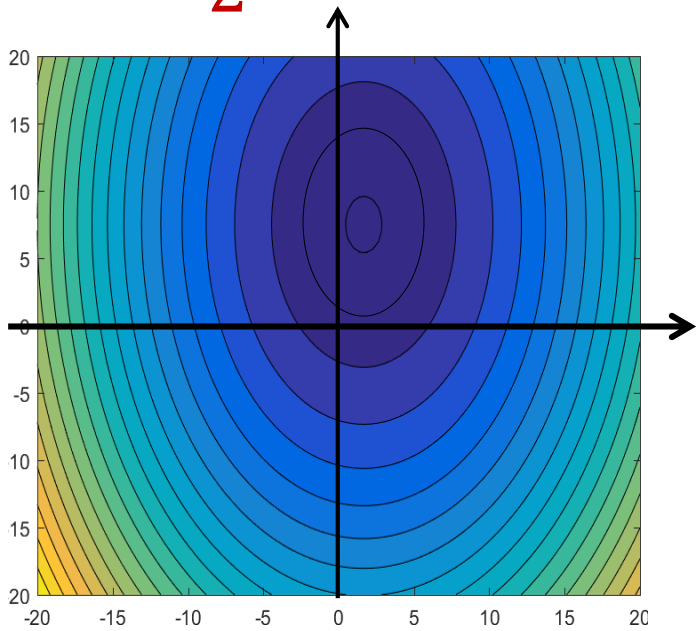
- When $\mathbf{A}$ is diagonal:

$$E = \frac{1}{2}\sum_i a_{ii}w_i^2 + \sum_i b_i w_i + c$$

  - The w_is are *uncoupled*
  - For *convex* (paraboloid) E, the a_{ii} values are all positive
  - Just an sum of N independent quadratic functions

Multivariate Quadratic with Diagonal $\mathbf{A}$

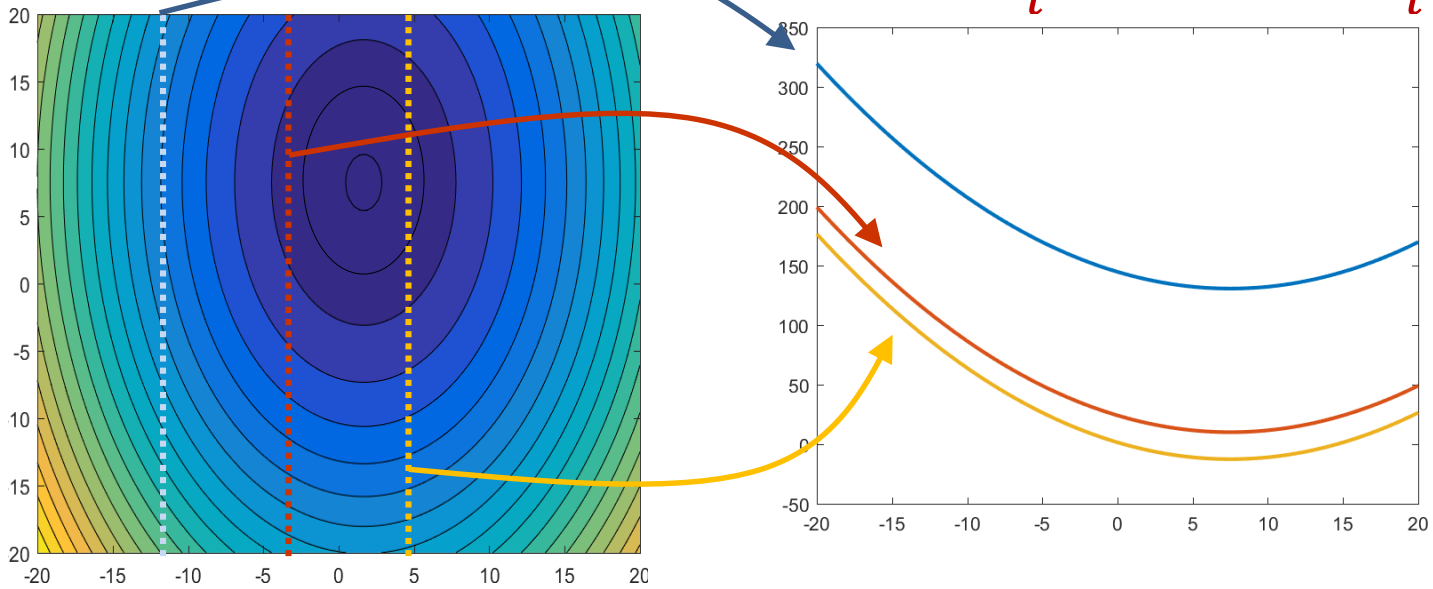
$$E = \frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{w}^T\mathbf{b} + c = \frac{1}{2}\sum_i a_{ii}w_i^2 + \sum_i b_i w_i + c$$



- Equal-value contours will be parallel to the axis

Multivariate Quadratic with Diagonal $\mathbf{A}$

$$E = \frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{w}^T\mathbf{b} + c = \frac{1}{2}\sum_i a_{ii}w_i^2 + \sum_i b_i w_i + c$$

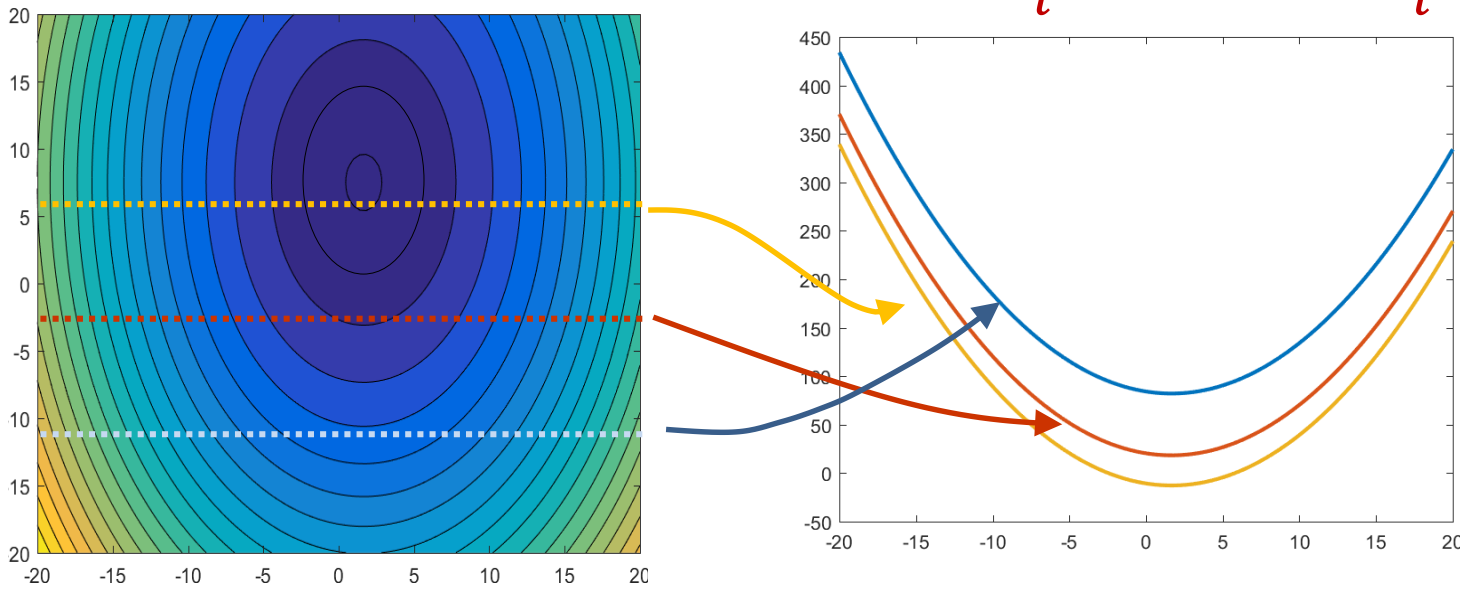


- Equal-value contours will be parallel to the axis
  - All "slices" parallel to an axis are shifted versions of one another

$$E = \frac{1}{2}a_{ii}w_i^2 + b_i w_i + c + C(\neg w_i)$$

Multivariate Quadratic with Diagonal A

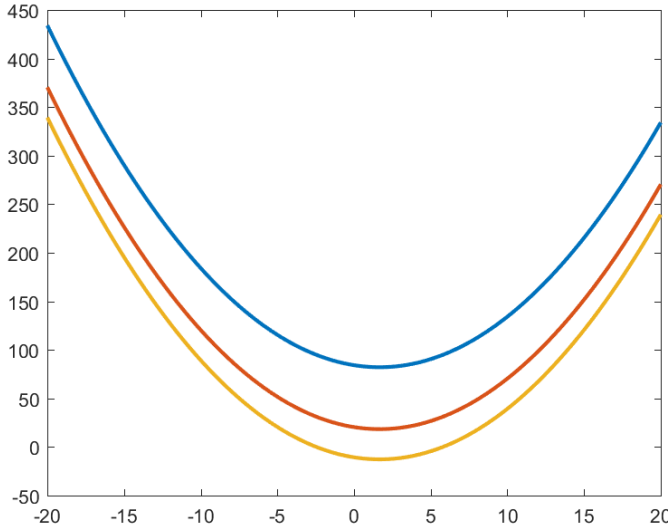
$$E = \frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{w}^T\mathbf{b} + c = \frac{1}{2}\sum_i a_{ii}w_i^2 + \sum_i b_i w_i + c$$

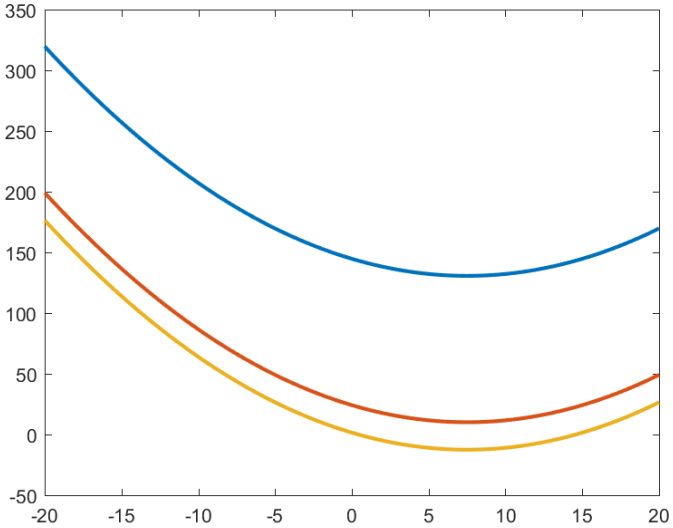


- Equal-value contours will be parallel to the axis
  - All "slices" parallel to an axis are shifted versions of one another

$$E = \frac{1}{2}a_{ii}w_i^2 + b_i w_i + c + C(\neg w_i)$$

“Descents” are uncoupled





$$E = \frac{1}{2}a_{11}w_1^2 + b_1 w_1 + c + C(\neg w_1)$$

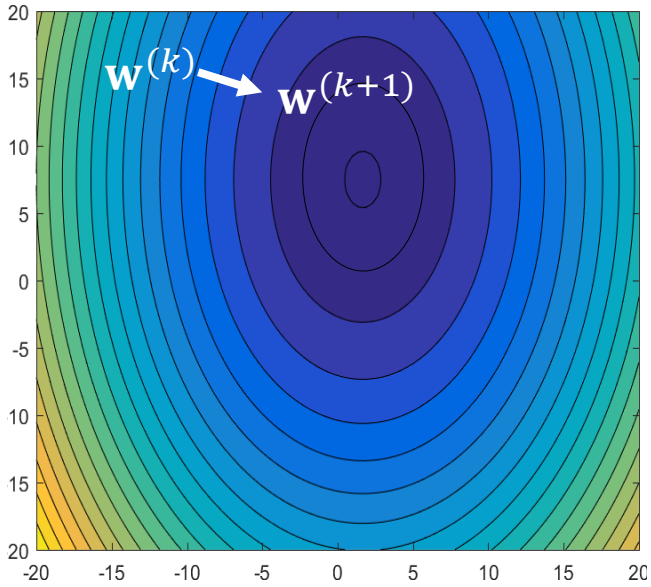
$$\eta_{1,opt} = a_{11}^{-1}$$

$$E = \frac{1}{2}a_{22}w_2^2 + b_2 w_2 + c + C(\neg w_2)$$

$$\eta_{2,opt} = a_{22}^{-1}$$

- The optimum of each coordinate is not affected by the other coordinates
  - I.e. we could optimize each coordinate independently
- **Note: Optimal learning rate is different for the different coordinates**

Vector update rule



$$\mathbf{w}^{(k+1)} \leftarrow \mathbf{w}^{(k)} - \eta \nabla_{\mathbf{w}} E$$

$$w_i^{(k+1)} = w_i^{(k)} - \eta \frac{dE\left(w_i^{(k)}\right)}{d\mathrm{w}}$$

- Conventional vector update rules for gradient descent: update entire vector against direction of gradient
  - Note : Gradient is perpendicular to equal value contour
  - The same learning rate is applied to all components

Problem with vector update rule

$$\mathbf{w}^{(k+1)} \leftarrow \mathbf{w}^{(k)} - \eta \nabla_{\mathbf{w}} E^T$$

$$w_i^{(k+1)} = w_i^{(k)} - \eta \frac{dE\left(w_i^{(k)}\right)}{d\mathrm{w}}$$

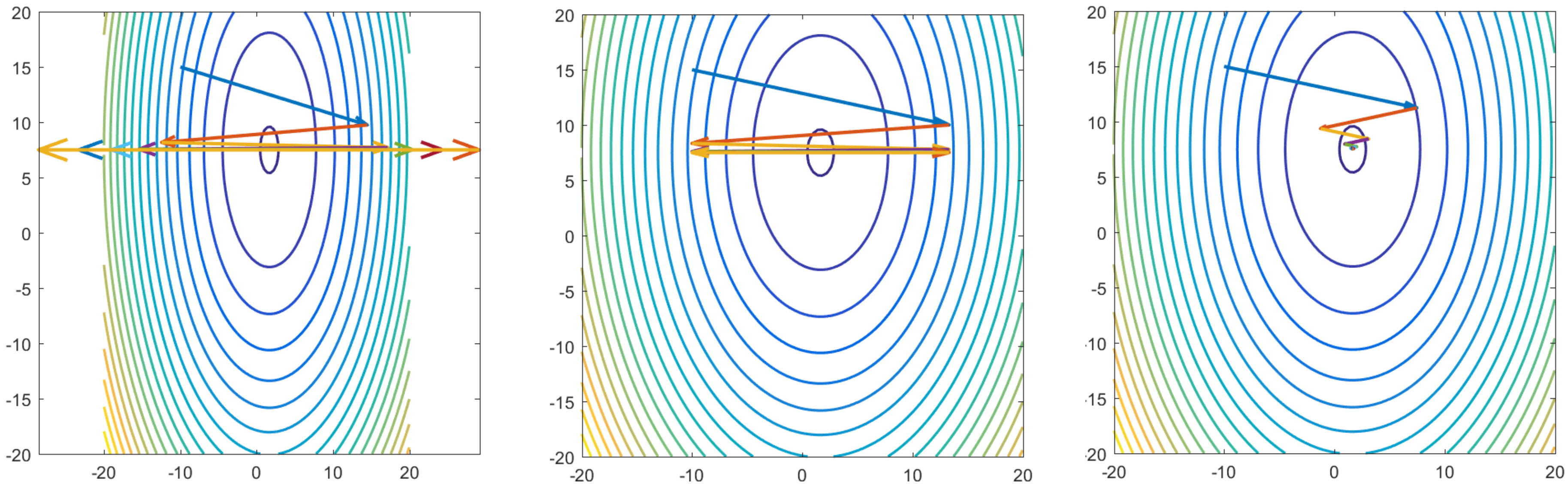
$$\eta_{i,opt} = \left(\frac{d^2 E\left(w_i^{(k)}\right)}{dw_i^2}\right)^{-1} = a_{22}^{-1}$$

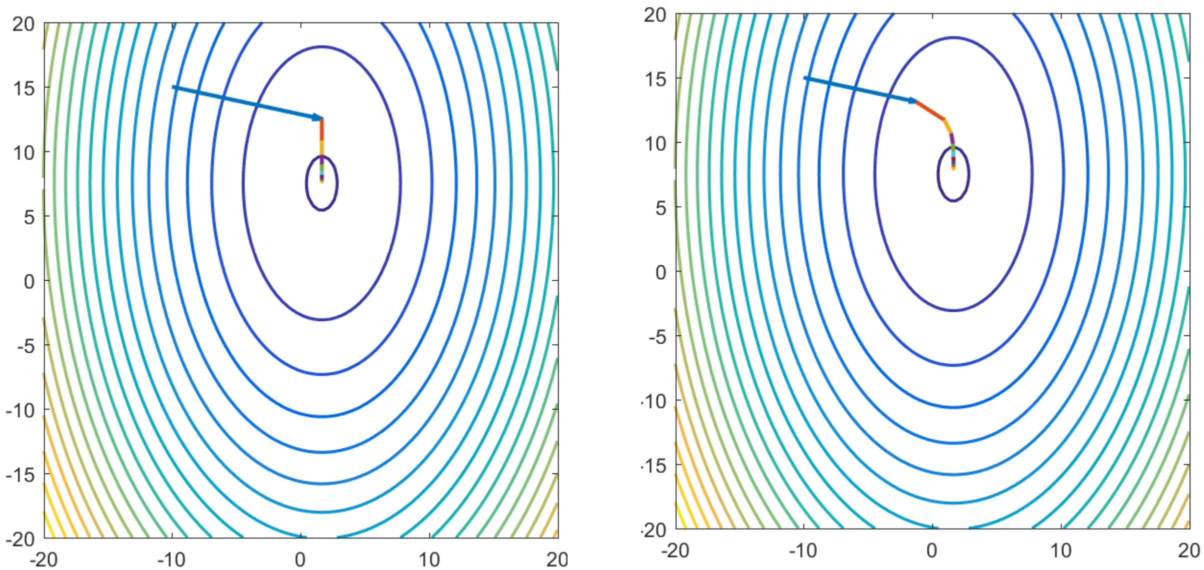
- The learning rate must be lower than twice the *smallest* optimal learning rate for any component

$$\eta < 2 \min_i \eta_{i,opt}$$

  - Otherwise the learning will diverge
- This, however, makes the learning very slow
  - And will oscillate in all directions where $\eta_{i,opt} \leq \eta < 2\eta_{i,opt}$

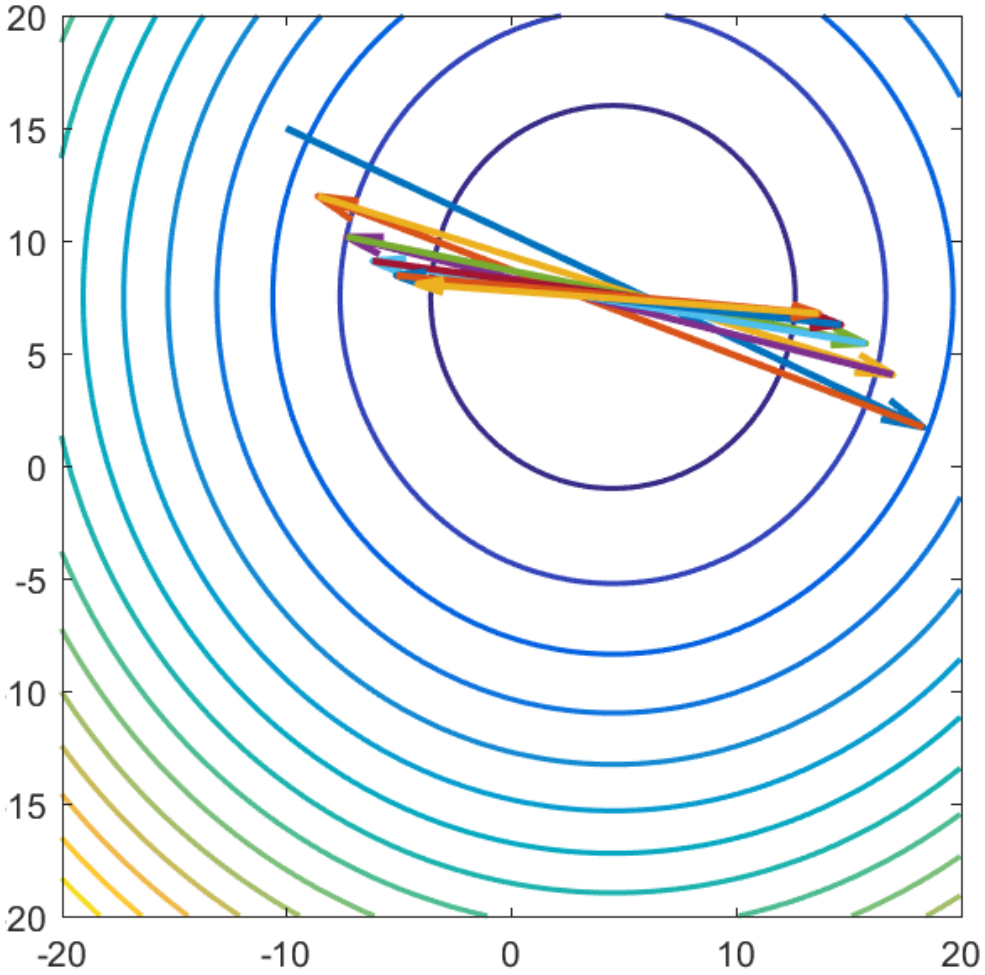
Dependence on learning rate





- $\eta_{1,opt} = 1;\ \eta_{2,opt} = 0.33$
- $\eta = 2.1\eta_{2,opt}$
- $\eta = 2\eta_{2,opt}$
- $\eta = 1.5\eta_{2,opt}$
- $\eta = \eta_{2,opt}$
- $\eta = 0.75\eta_{2,opt}$

Dependence on learning rate



- $\eta_{1,opt} = 1;\ \eta_{2,opt} = 0.91;$ $\eta = 1.9\,\eta_{2,opt}$

Convergence

- Convergence behaviors become increasingly unpredictable as dimensions increase

- For the fastest convergence, ideally, the learning rate η must be close to both, the largest $\eta_{i,opt}$ and the smallest $\eta_{i,opt}$
  - To ensure convergence in every direction
  - Generally infeasible

- Convergence is particularly slow if $\frac{\max_i \eta_{i,opt}}{\min_i \eta_{i,opt}}$ is large
  - The "condition" number is small (more on this shortly)

More Problems

- For quadratic (strongly) convex functions, gradient descent is exponentially fast
  - Linear convergence
  - Assuming learning rate is non-divergent
- For generic (Lipschitz Smooth) convex functions however, it is very slow

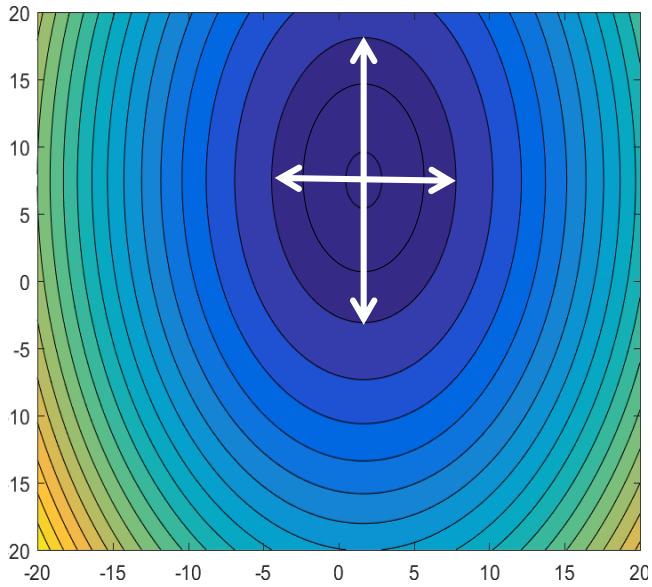
$$\left|f\left(w^{(k)}\right) - f(w^*)\right| \propto \frac{1}{k}\left|f\left(w^{(0)}\right) - f(w^*)\right|$$

  - And inversely proportional to learning rate

$$\left|f\left(w^{(k)}\right) - f(w^*)\right| \leq \frac{1}{2\eta k}\left|w^{(0)} - w^*\right|$$

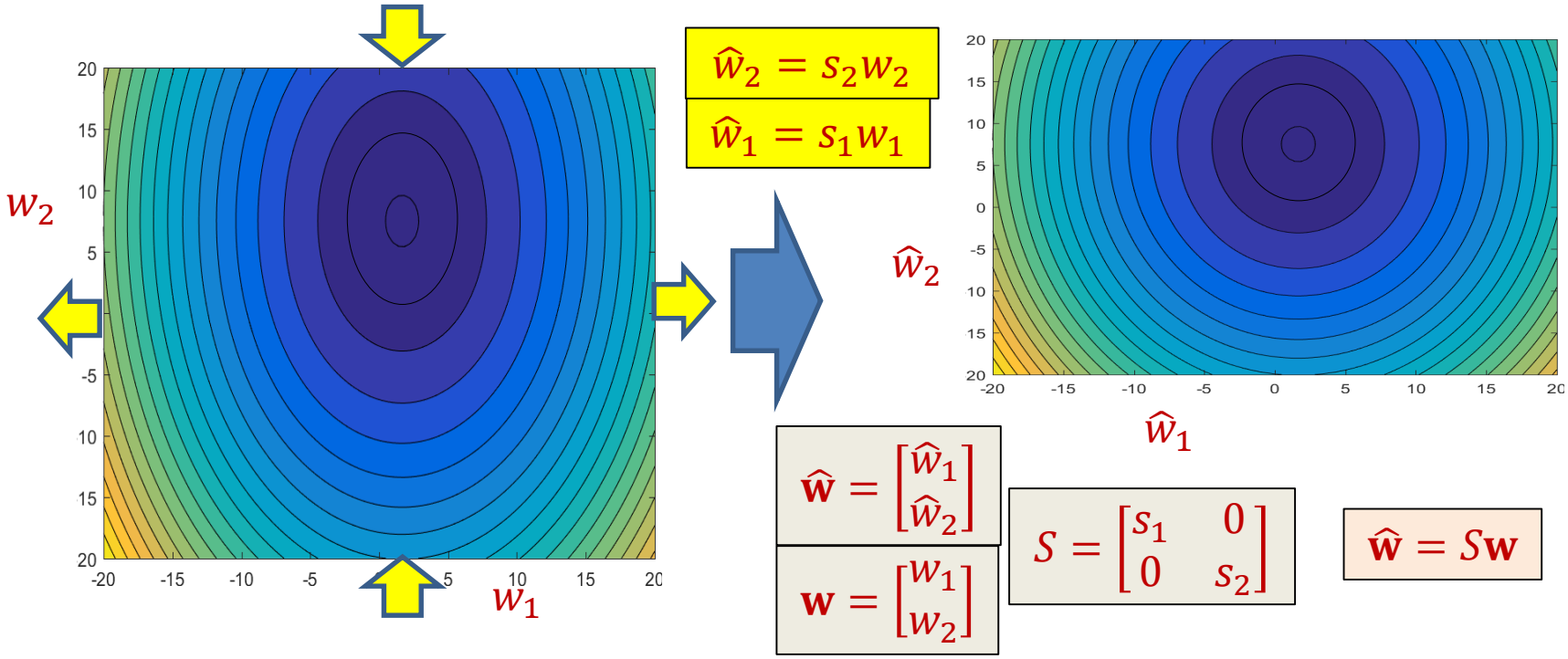
  - Takes $O(1/\epsilon)$ iterations to get to within ϵ of the solution
- An inappropriate learning rate will destroy your happiness

The reason for the problem



- The objective function has different eccentricities in different directions
  - Resulting in different optimal learning rates for different directions
- Solution: *Normalize* the objective to have identical eccentricity in all directions
  - Then all of them will have identical optimal learning rates
  - Easier to find a working learning rate

Solution: Scale the axes



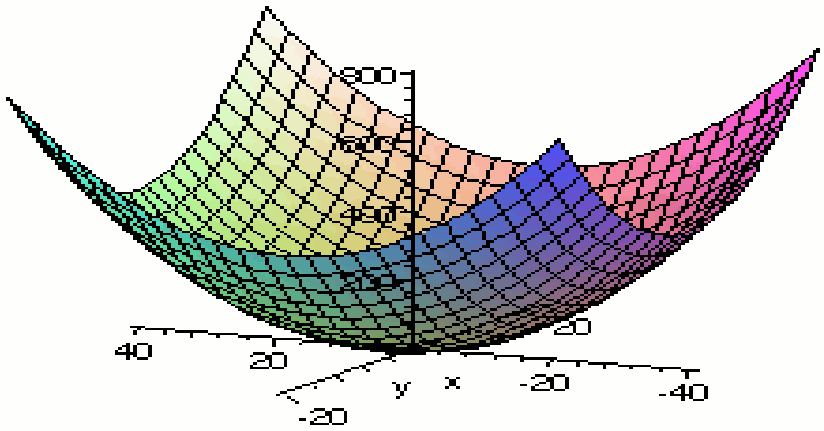
- Scale the axes, such that all of them have identical (identity) "spread"
  - Equal-value contours are circular
- **Note:** equation of a quadratic surface with circular equal-value contours can be written as

$$E = \frac{1}{2}\hat{\mathbf{w}}^T\hat{\mathbf{w}} + \hat{\mathbf{b}}^T\hat{\mathbf{w}} + c$$

Scaling the axes

- Finding the right way to scale the axes is an art in itself
- Instead, we will modify things differently: *allow different step sizes in different directions*

The Momentum Methods



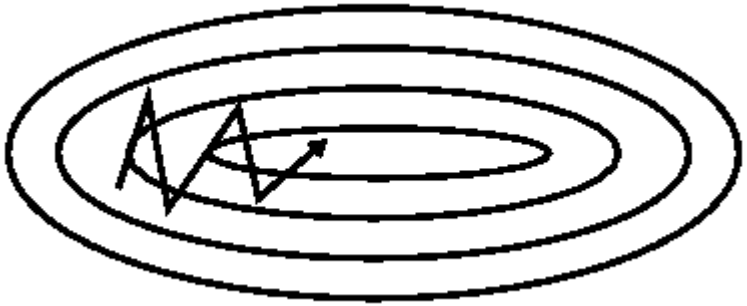
Figures from Sebastian Ruder

- Conventional gradient descent follows the immediate gradient
  - Against *local* direction of steepest ascent
  - Constantly follows *current local trend*
- This largely ignores *global* trend
  - Can take forever to get to optimum
- Global trend is generally also a component of local trends
  - Momentum methods: Increase contribution of global trends to step size by averaging across steps

The Momentum Method

Plain gradient update

With acceleration



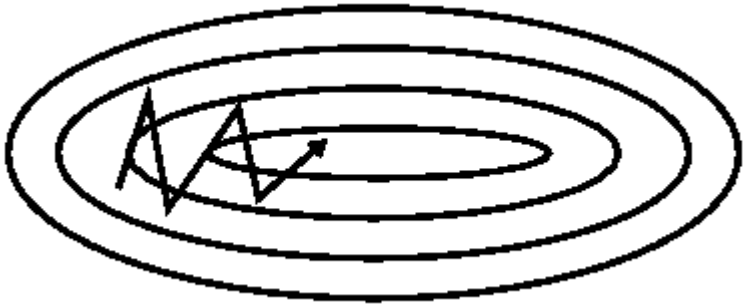
- "Remember" the previous step
- The actual step is a linear combination of the previous step and the current batch gradient

$$\Delta W_k = \beta \Delta W_{k-1} + \eta \nabla D(W_{k-1})$$

$$W_k = W_{k-1} - \Delta W_k$$

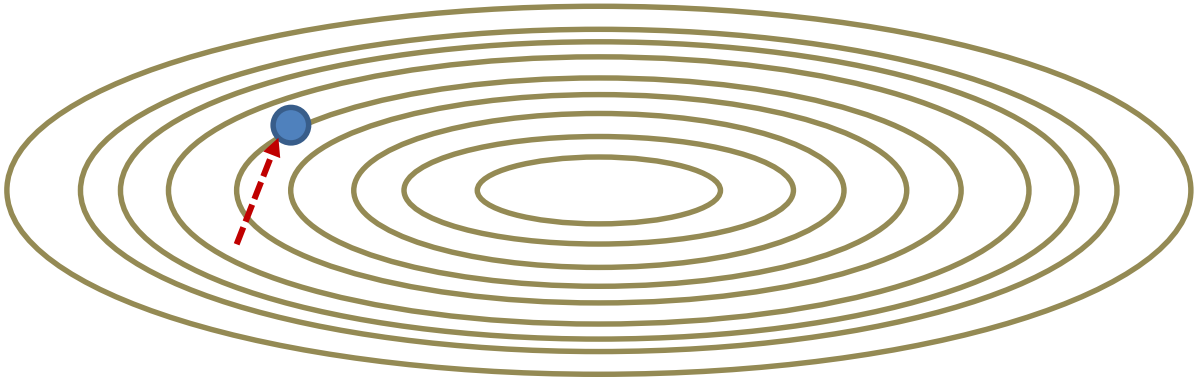
- Typical β value is 0.9
- Steps
  - Get longer in directions where gradient stays in the same sign
  - Become shorter where the sign keeps flipping

Nestorov's method



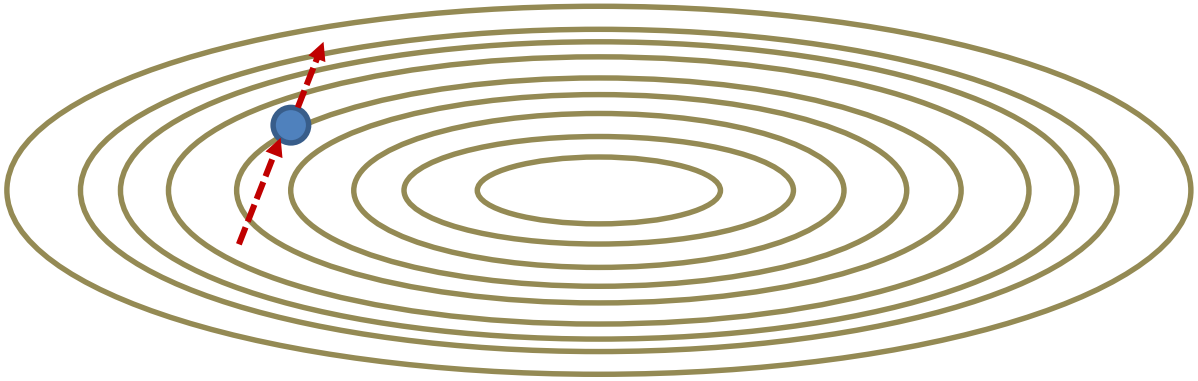
- Simple gradient descent: only look at current trend
- Momentum: Look at current trend *and* past trend
- Better still: Look at the *future*
  - Where you will go if you follow the momentum strategy

Nestorov's method



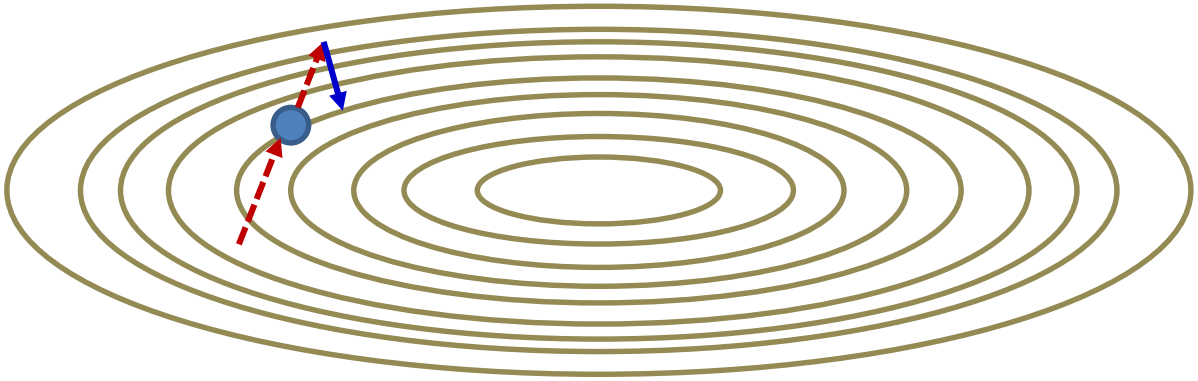
- Location at t and direction v_t to come here

Nestorov's method



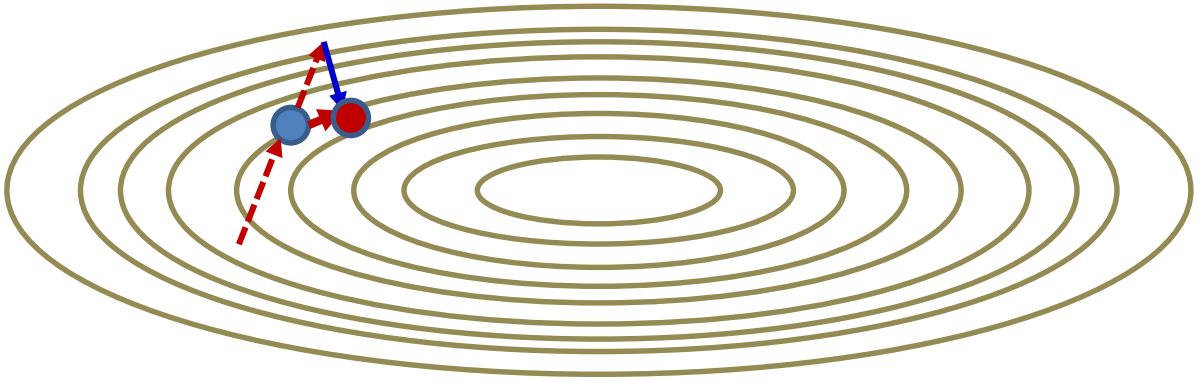
- Location at t and direction v_t to come here
- Guess to where we would go if we continued the way we came

Nestorov's method



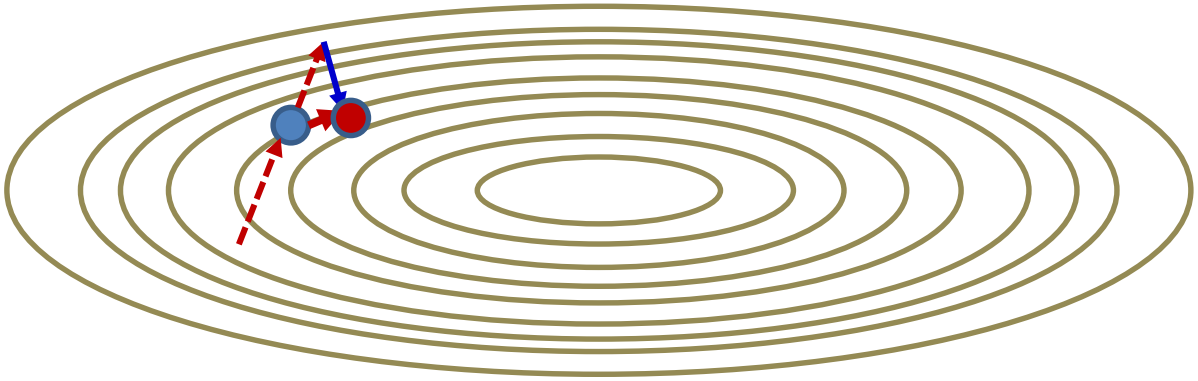
- Location at t and direction v_t to come here
- Guess to where we would go if we continued the way we came
- The gradient at that point
  - How we would update through gradient descent from this *future* point

Nestorov's method



- Location at t and direction v_t to come here
- Guess to where we would go if we continued the way we came
- The gradient at that point
  - How we would update through gradient descent from this *future* point
- Final update

Nestorov's Accelerated Gradient



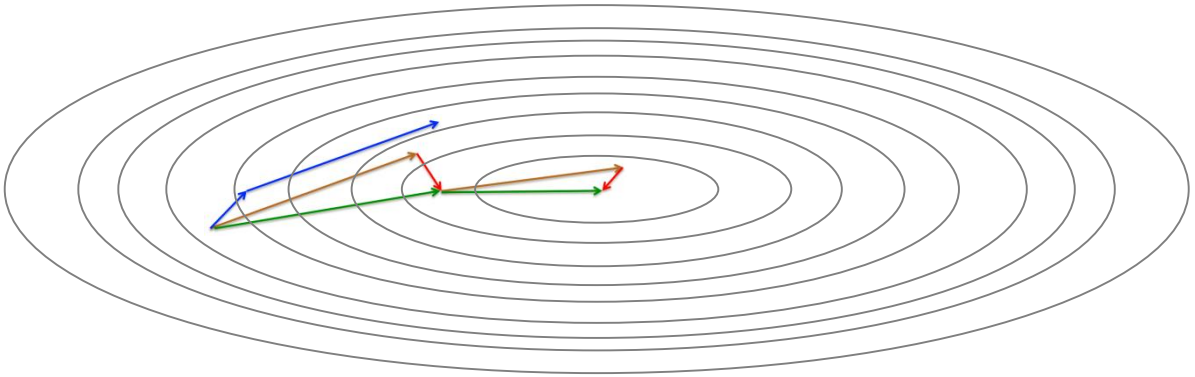
- Nestorov's method

$$\Delta W_k = \beta \Delta W_{k-1} + \eta \nabla D(W_{k-1} - \beta \Delta W_{k-1})$$

$$W_k = W_{k-1} - \Delta W_k$$

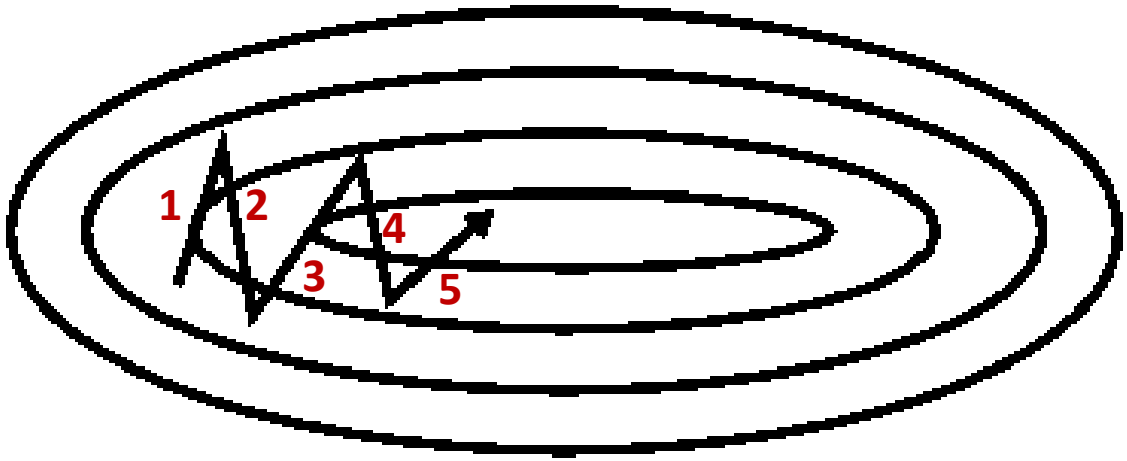
- Based on guessing the future

Nestorov's Accelerated Gradient



- From Hinton
  - Instead of
    1. Computing current gradient
    2. Taking a (negative) step in the direction of the current gradient
    3. Adding (negative) acceleration term from there
  - We
    1. Take a (negative) step in the direction of acceleration
    2. Compute the gradient there
    3. Take a (negative) step in the direction of the gradient
- Converges much faster

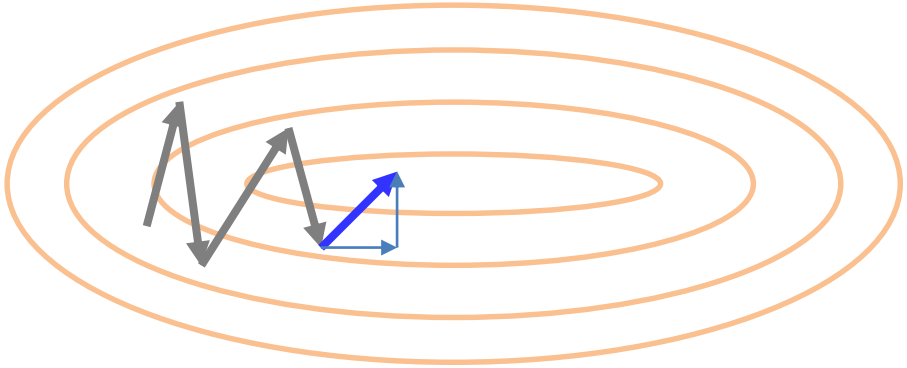
Smoothing the trajectory

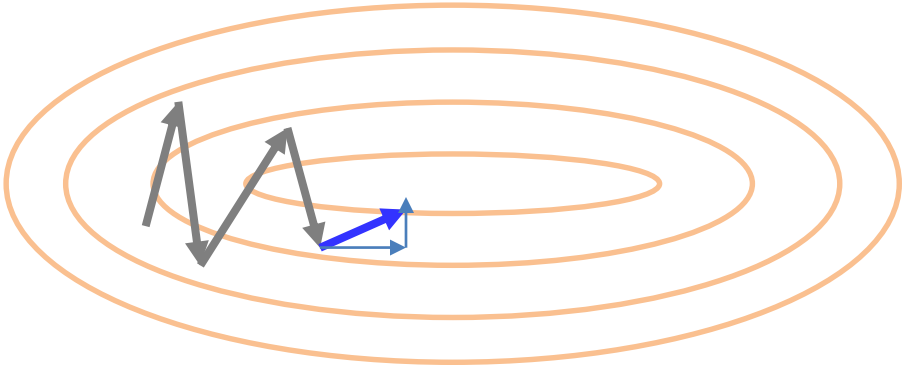


Step	X component	Y component
1	1	+2.5
2	1	-3
3	3	+2.5
4	1	-2
5	2	1.5

- Simple gradient and acceleration methods still demonstrate oscillatory behavior in some directions
- Observation: Steps in "oscillatory" directions show large total movement
  - In the example, total motion in the vertical direction is much greater than in the horizontal direction
- Improvement: Dampen step size in directions with high motion

Variance-normalized step





- In recent past
  - Total movement in *Y* component of updates is high
  - Movement in *X* components is lower
- Current update, modify usual gradient-based update:
  - Scale *down* Y component
  - Scale *up* X component
- A variety of algorithms have been proposed on this premise
  - We will see a popular example

RMS Prop

- Notation:
  - Updates are ***by parameter***
  - Sum derivative of divergence w.r.t any individual parameter w is shown as $\partial_w D$
  - The ***squared*** derivative is $\partial_w^2 D = (\partial_w D)^2$
  - The ***mean squared*** derivative is a running estimate of the average squared derivative. We will show this as $E[\partial_w^2 D]$
- Modified update rule: We want to
  - scale down updates with large mean squared derivatives
  - scale up updates with large mean squared derivatives

RMS Prop

- This is a variant on the *basic* mini-batch SGD algorithm

- **Procedure:**
  - Maintain a running estimate of the mean squared value of derivatives for each parameter
  - Scale update of the parameter by the *inverse* of the *root mean squared* derivative

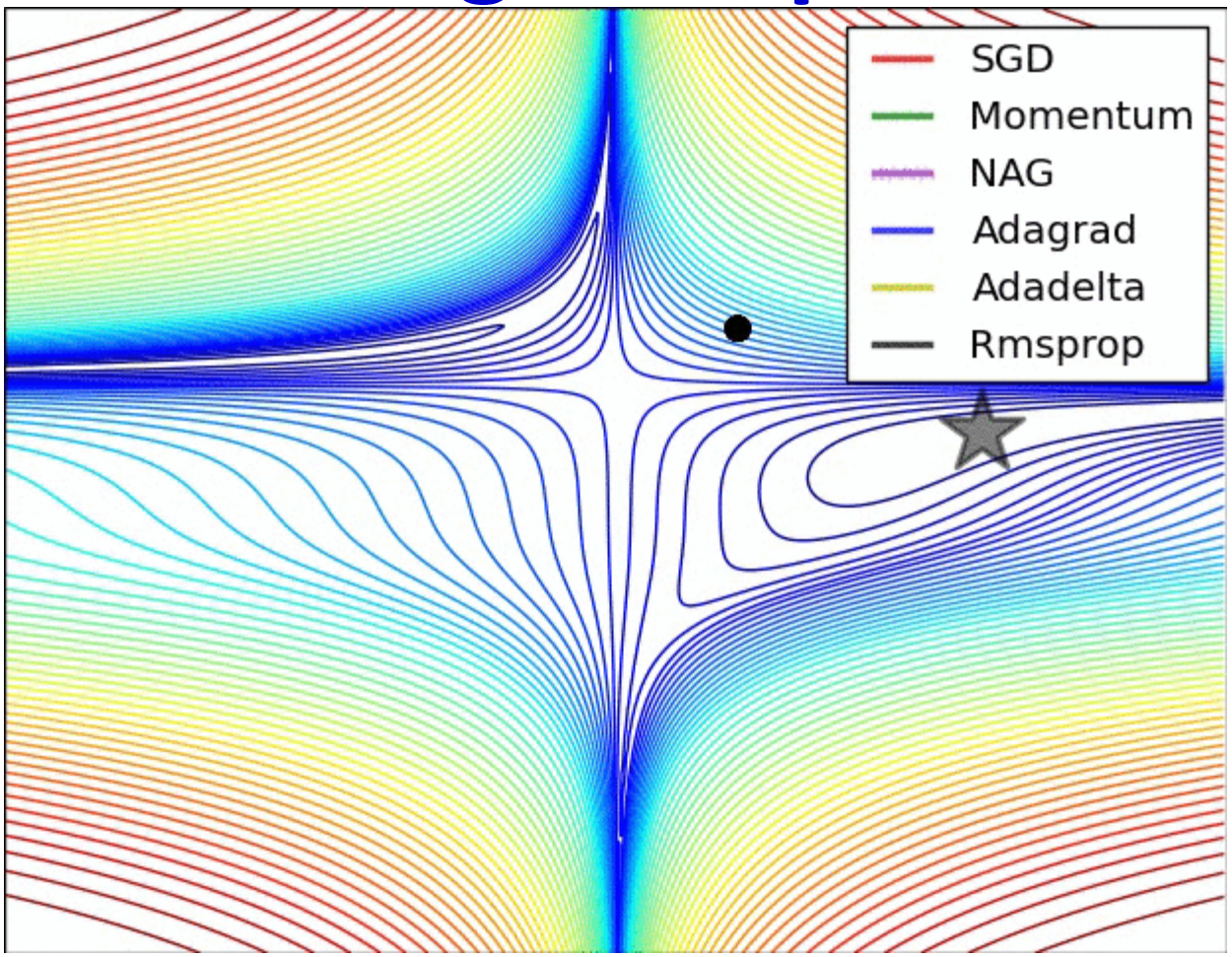
$$E[\partial_w^2 D]_k = \gamma E[\partial_w^2 D]_{k-1} + (1-\gamma)(\partial_w^2 D)_k$$

$$w_{k+1} = w_k - \frac{\eta}{\sqrt{E[\partial_w^2 D]_k + \epsilon}} \partial_w D$$

Fast convergence remains a challenge

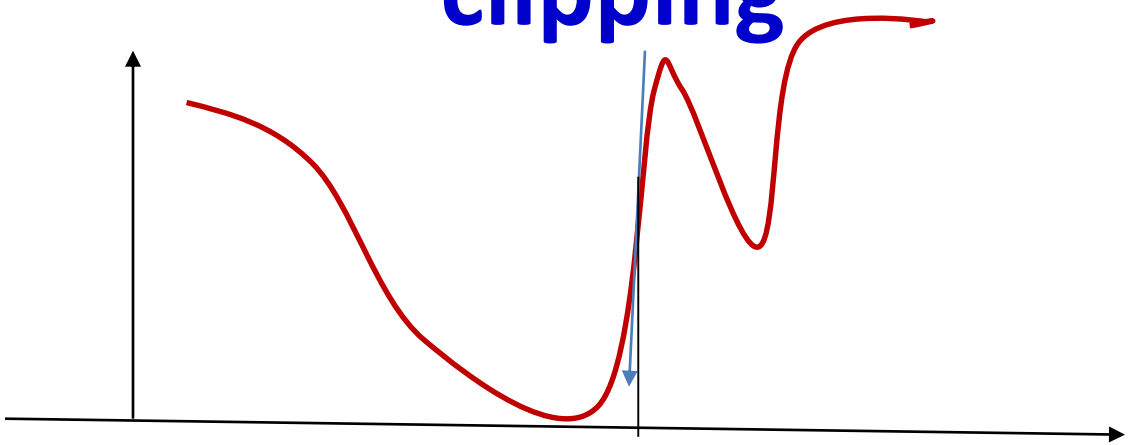
- Newer methods:
  - Adagrad
  - AdaDelta
  - RMS Prop
  - ADAM

Visualizing the optimizers



- http://sebastianruder.com/optimizing-gradient-descent/index.html

A few additional tricks: Gradient clipping



- Often the derivative will be too high
  - When the divergence has a steep slope
  - This can result in instability
- **Gradient clipping**: set a ceiling on derivative value

$$if\ \partial_w D > \theta\ then\ \partial_w D = \theta$$

  - Typical θ value is 5

Several other tricks are often required for good performance

- Batch normalization
- Dropout

- Other issues not covered:
- Objective functions, speed up algorithms, generalization..

Far from complete

- Efficient training remains a challenge with much scope for advances..