Machine Intelligence

8. Machine Learning: Neural Networks

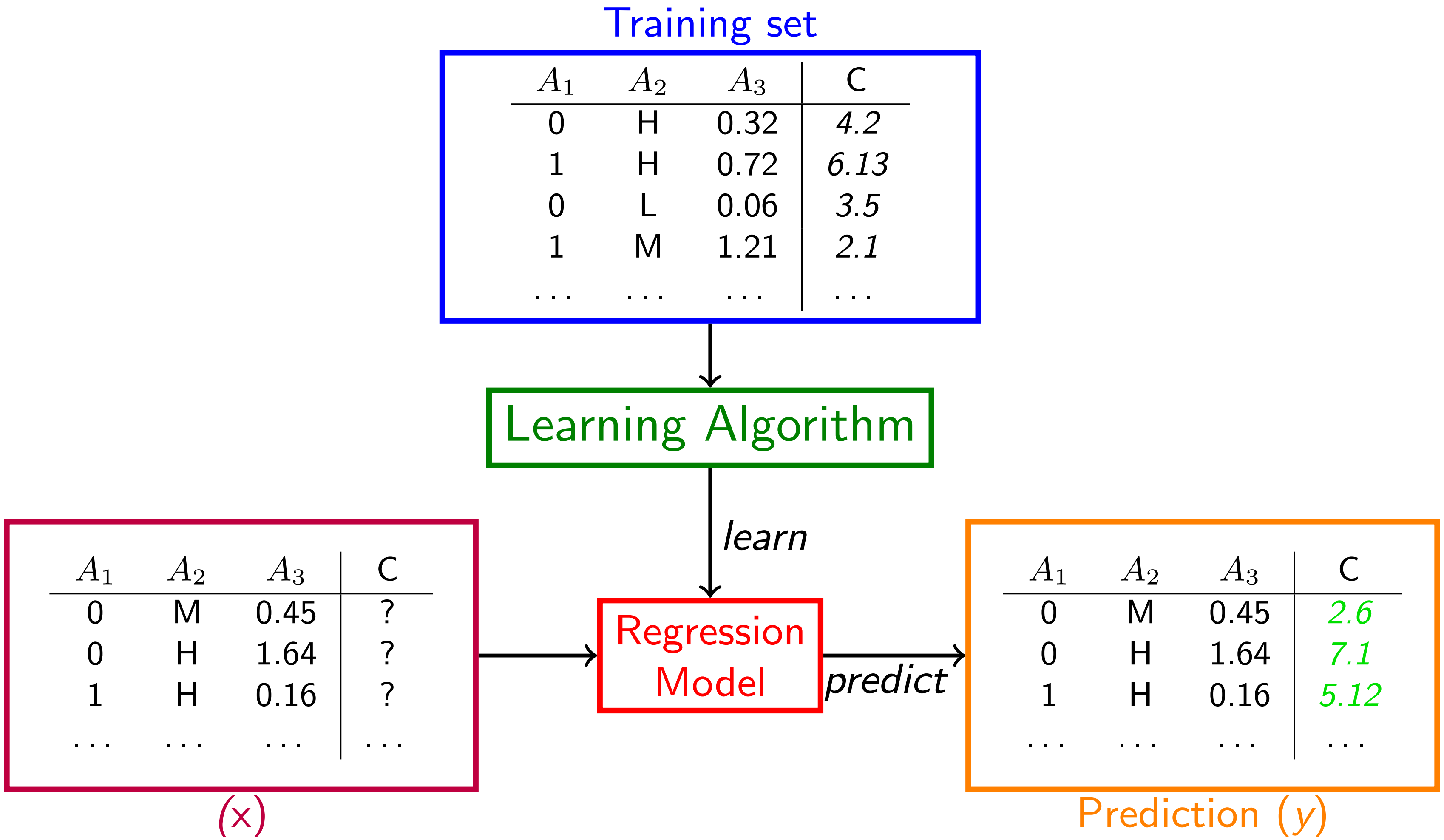
Álvaro Torralba

AALBORG UNIVERSITET

Fall 2022

Thanks to Thomas D. Nielsen and Jörg Hoffmann for slide sources

Reminder: Regression



Training set

A_1	A_2	A_3	C
0	H	0.32	*4.2*
1	H	0.72	*6.13*
0	L	0.06	*3.5*
1	M	1.21	*2.1*
...	...	...	...

(x)

A_1	A_2	A_3	C
0	M	0.45	?
0	H	1.64	?
1	H	0.16	?
...	...	...	...

Prediction (y)

A_1	A_2	A_3	C
0	M	0.45	*2.6*
0	H	1.64	*7.1*
1	H	0.16	*5.12*
...	...	...	...

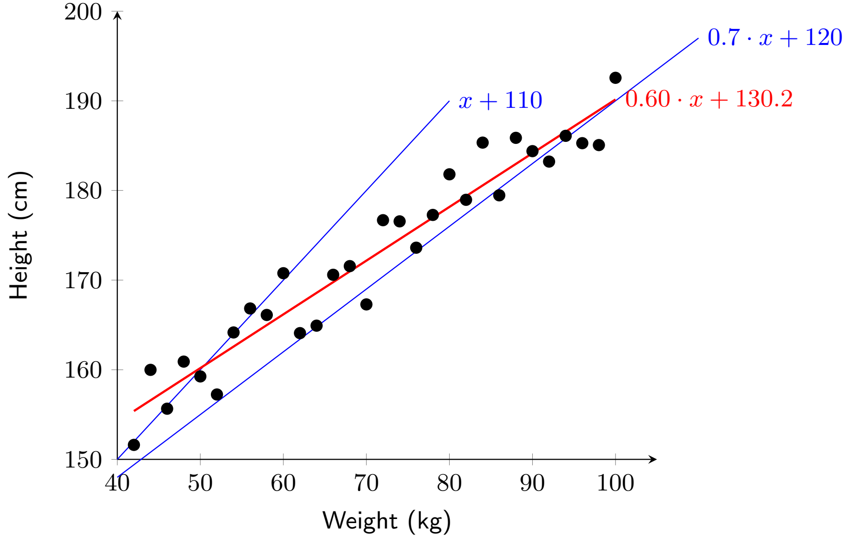
→In this lecture we will see a learning algorithm to obtain the function f that predicts the correct answer

Reminder: Linear Regression

Assumes the target attribute is a linear combination of the input attributes:

$$\hat{y} = \sum_{i=1}^{n} w_i x_i + w_0$$

Parameters: $W = w_0, \dots w_n$
Find the weights W that minimize the error: $\sum (y - \hat{y})^2$



Two questions we will try to answer today:

1. **How to find these weights?** $\rightarrow$ Gradient Descent!
2. **How to represent more complex functions** $\rightarrow$ Neural Networks!

Our Agenda for This Chapter

- **Gradient Descent (for Linear Regression)**: How the learning happens.

  $\rightarrow$ **A basic method that we can use to learn in linear regression and later in neural networks.**

- **Neural Networks: Multi-Layer Percepton**: The power of connecting simple units of computation

  $\rightarrow$ **The type of neural networks we will consider**

- **Backpropagation**: How to apply gradient descent in neural networks

  $\rightarrow$ **All the math behind the scenes!**

- **Discrete Input/Outputs**: What if some attributes are not numeric?

  $\rightarrow$ **It's easy to adapt Neural Networks for discrete variables too!**

- **Expressive Power**: What functions can we represent?

  $\rightarrow$ **A quick theoretical look at what can be done with neural networks**

Notation

Bold letters represent vectors (list of values)

We have a **training set**:

- X is the input matrix with M rows and N columns
  - M is the number of examples in the training set
  - N is the number of attributes
  - Each example $(\mathbf{x}_1, \ldots, \mathbf{x}_M)$ is an input vector
- $\mathbf{y} = (y_1, \ldots, y_M)$ vector of target values

Name		Calories	Protein	Sugars	Rating ($\mathbf{y}$)	Output ($\mathbf{o}$)
($\mathbf{x}_1$) All-Bran	1	70	4	5	(y_1) 59.42	(o_1) 44
($\mathbf{x}_2$) Almond_Delight	1	110	2	8	(y_2) 34.38	(o_2) 80
($\mathbf{x}_3$) Apple_Jacks	1	110	2	14	(y_3) 33.17	(o_3) 68
($\mathbf{x}_4$) Basic_4	1	130	3	8	(y_4) 37.03	(o_4) 99
($\mathbf{x}_5$) Bran_Chex	1	90	2	6	(y_5) 49.12	(o_5) 64
($\mathbf{x}_6$) Bran_Flakes	1	90	3	5	(y_6) 53.31	(o_6) 65
($\mathbf{x}_7$) Cap_n_Crunch	1	120	1	12	(y_7) 18.04	(o_7) 83
($\mathbf{x}_8$) Cheerios	1	110	6	1	(y_8) 50.76	(o_8) 90
Parameters ($\mathbf{w}$):	$w_0 = -12$	$w_1 = 1$	$w_2 = -1$	$w_3 = -2$		

In **Linear Regression** $\mathbf{o}_i = \mathbf{w} \cdot \mathbf{x}_i$ (i.e., of $o_i = \sum_{j=0}^{N} w_j x_{i,j}$ for all examples $i \in [1, M]$)

- $\mathbf{w} = (w_0, \ldots, w_N)$ vector of parameters
- $\mathbf{o} = (o_1, \ldots, o_M)$ vector of current outputs (o and $\hat{y}$ are synonyms)

$\rightarrow$ **We request to find values $\mathbf{w}^*$ of the parameters yielding $\mathbf{o} = \mathbf{y}$**

Sum Squared Error as a Function of the Parameters

We want to find the value of the parameters $\mathbf{w}$ that minimize the SSE:

$$SSE = \sum_{i=1}^{M}(y_i - o_i)^2 = \sum_{i=1}^{M}\left(y_i - (\sum_{j=1}^{N} x_{i,j}w_j)\right)^2 =$$

We define the error-function as:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{M}\left(y_i - (\sum_{j=1}^{N} x_{i,j}w_j)\right)^2$$

Note:

1. We multiply by $\frac{1}{2}$ because it simplifies the derivative. This is fine because the mimimum is the same
2. $x_{i,j}$ and y_i are just the values on our dataset so $E(\mathbf{w})$ is just a quadratic equation:

Name		Calories	Protein	Sugars	Rating ($\mathbf{y}$)
($\mathbf{x}_1$) All-Bran	1	70	4	5	(y_1) 59.42
($\mathbf{x}_2$) Almond_Delight	1	110	2	8	(y_2) 34.38
($\mathbf{x}_3$) Apple_Jacks	1	110	2	14	(y_3) 33.17
($\mathbf{x}_4$) Basic_4	1	130	3	8	(y_4) 37.03
($\mathbf{x}_5$) Bran_Chex	1	90	2	6	(y_5) 49.12
($\mathbf{x}_6$) Bran_Flakes	1	90	3	5	(y_6) 53.31
($\mathbf{x}_7$) Cap_n_Crunch	1	120	1	12	(y_7) 18.04
	w_0	w_1	w_2	w_3	

$$E(w_0,w_1,w_2,w_3) = (59.42 - w_0 - 70w_1 - 4w_2 - 5w_3)^2 + (34.38 - w_0 - 110w_1 - 2w_2 - 8w_3)^2 + \ldots$$

Idea of Gradient Descent

1. Initialize weights randomly to any value
2. While there is error, slightly change the parameters to reduce error

The weights are NOT updated randomly.
We decide how to change them in order to reduce the error $\nabla E = \mathbf{y} - \mathbf{o}$:

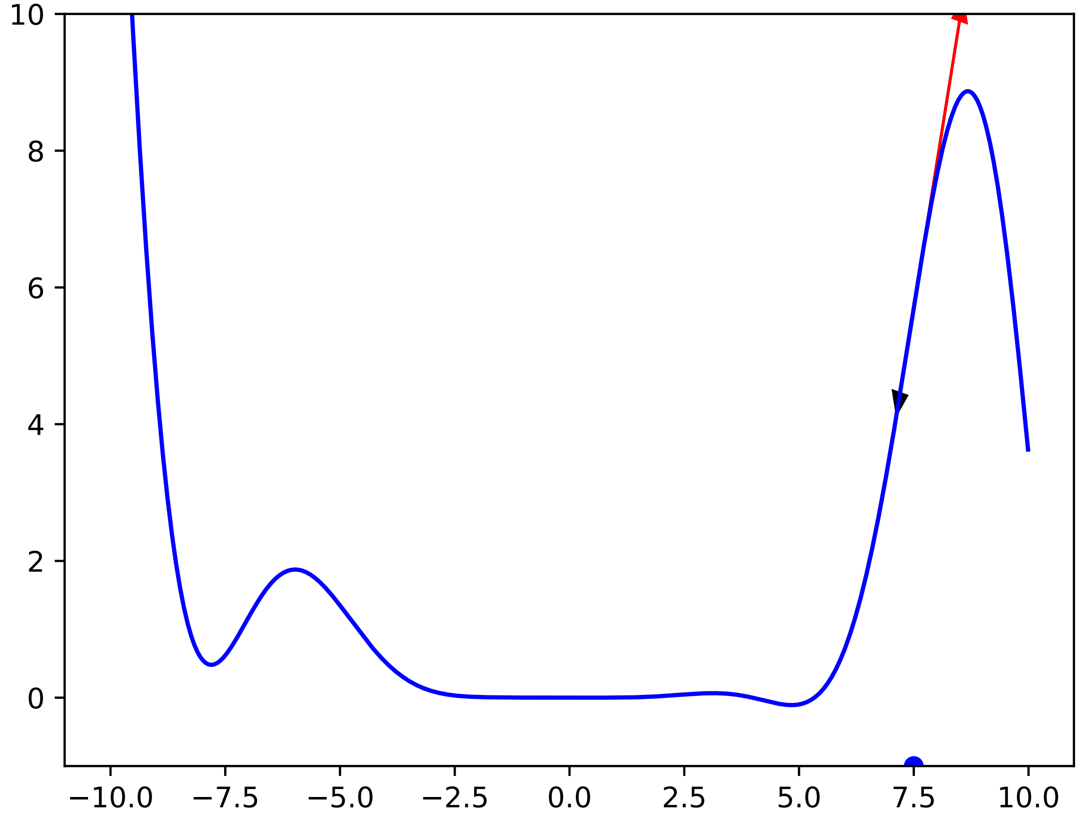
Intuition to Update the Weights

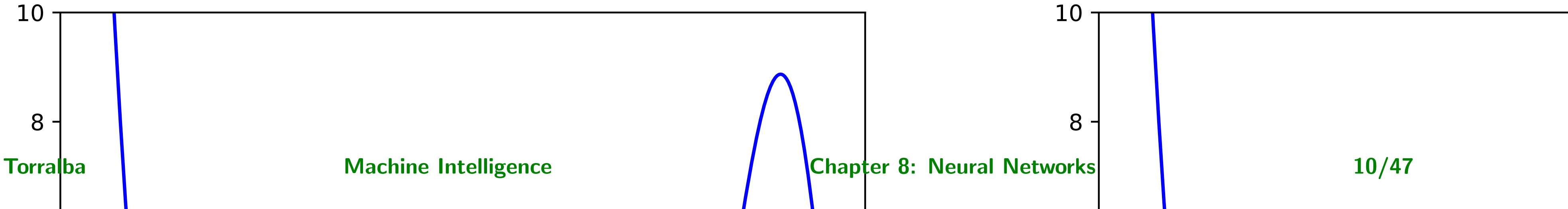
- $\nabla E > 0 \Rightarrow o$ shall be increased $\Rightarrow \mathbf{x} \cdot \mathbf{w}$ up $\Rightarrow \mathbf{w} := \mathbf{w} + \alpha \nabla E \mathbf{x}$
- $\nabla E < 0 \Rightarrow o$ shall be decreased $\Rightarrow \mathbf{x} \cdot \mathbf{w}$ down $\Rightarrow \mathbf{w} := \mathbf{w} + \alpha \nabla E \mathbf{x}$

Hyperparameter α is called the **learning rate**:
$\rightarrow$ α controls how much we update the parameters at each iteration

$\rightarrow$we can find in which direction to change the weights by looking at the slope of the error function!

Gradient descent: an example





Gradient Descent Learning

The gradient is the vector of partial derivatives:

$$\nabla E[\mathbf{w}] = \left(\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n} \right)$$

The partial derivatives are (with a linear activation function):

$$\frac{\partial E}{\partial w_k} = \sum_{i=1}^{M} (\mathbf{y}_i - \mathbf{w} \cdot \mathbf{x}_i)(-x_{i,k}).$$

Gradient descent rule:

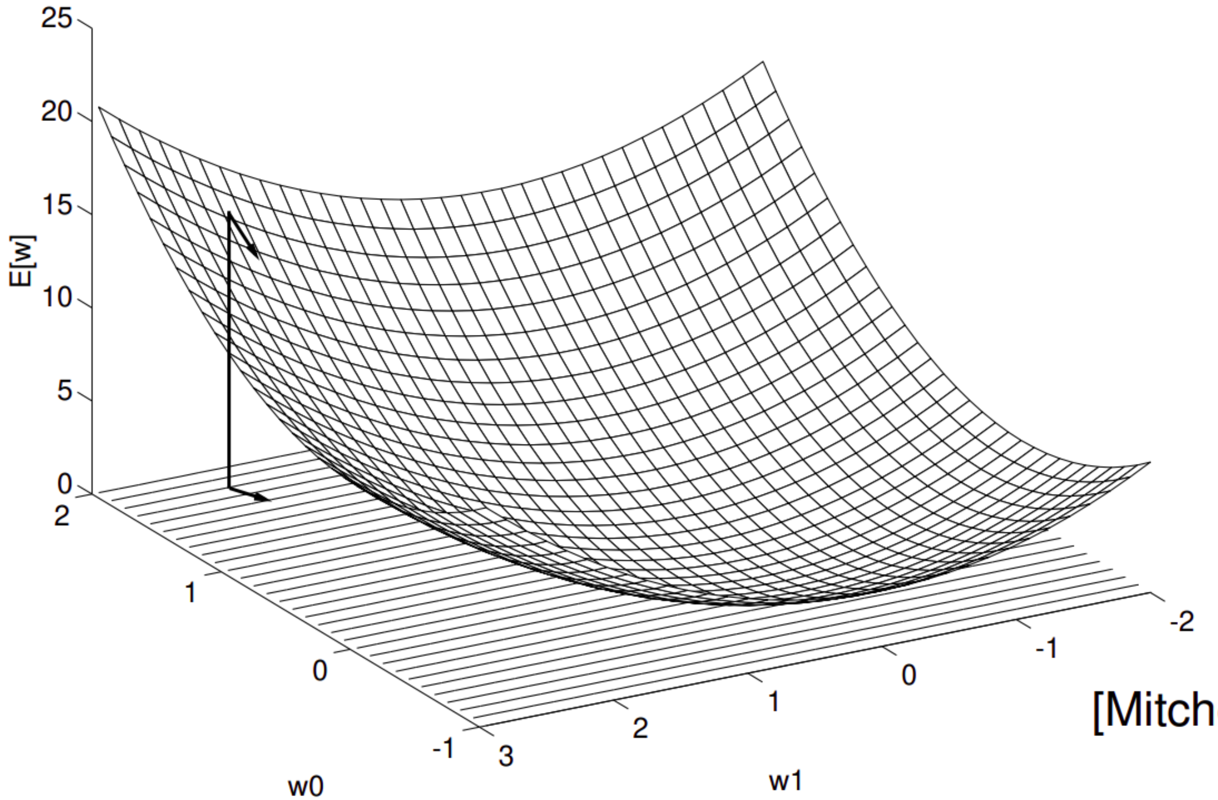
Initialize $\mathbf{w}$ with random values
repeat
$$\mathbf{w} := \mathbf{w} - \alpha \nabla E(\mathbf{w})$$
until $\nabla E(\mathbf{w}) \approx \mathbf{0}$

(α is again a small constant, the learning rate).

Properties of Gradient Descent for SSE and Linear Regression

- Gradient Descent can be applied to optimize any function (as long as the error function is differentiable) but it may converge to a local minima
- In Linear Regression it conveges to the optimal value (global minima) because the SSE is always a smooth, convex function of the weights!

Example for $N = 1$ (and a linear activation function):



[Mitchell, Fig.4.4]

$\rightsquigarrow$ weights $\mathbf{w}$ that minimize $E(\mathbf{w})$ can be found by <u>gradient descent</u>.

Computational Cost of Gradient Descent

$$\frac{\partial E}{\partial w_k} = \sum_{i=1}^{M} (y_i - \mathbf{w} \cdot \mathbf{x}_i)(-x_{i,k}).$$

Question

What is the computational cost of computing the gradient?

Stochastic Gradient Descent

Variation of gradient descent: instead of following the gradient computed from the whole dataset:

Gradient Descent (like in previous slides)

$$\frac{\partial E}{\partial w_i} = \sum_{k=1}^{M} (y_k - \mathbf{w} \cdot \mathbf{x}_k)(-x_{k,i}),$$

iterate through the data instances one by one (or by batches), and in one iteration follow the gradient defined by a single data instance $(\mathbf{x}_k, y_k)$:

Stochastic Gradient Descent

$$\frac{\partial E}{\partial w_i} = (y_k - \mathbf{w} \cdot \mathbf{x}_k)(-x_{k,i}),$$

$\rightarrow$ This still tends to converge towards a local minima (or global minima if the error function is convex like in linear regression)

The Issue with Linear Regression

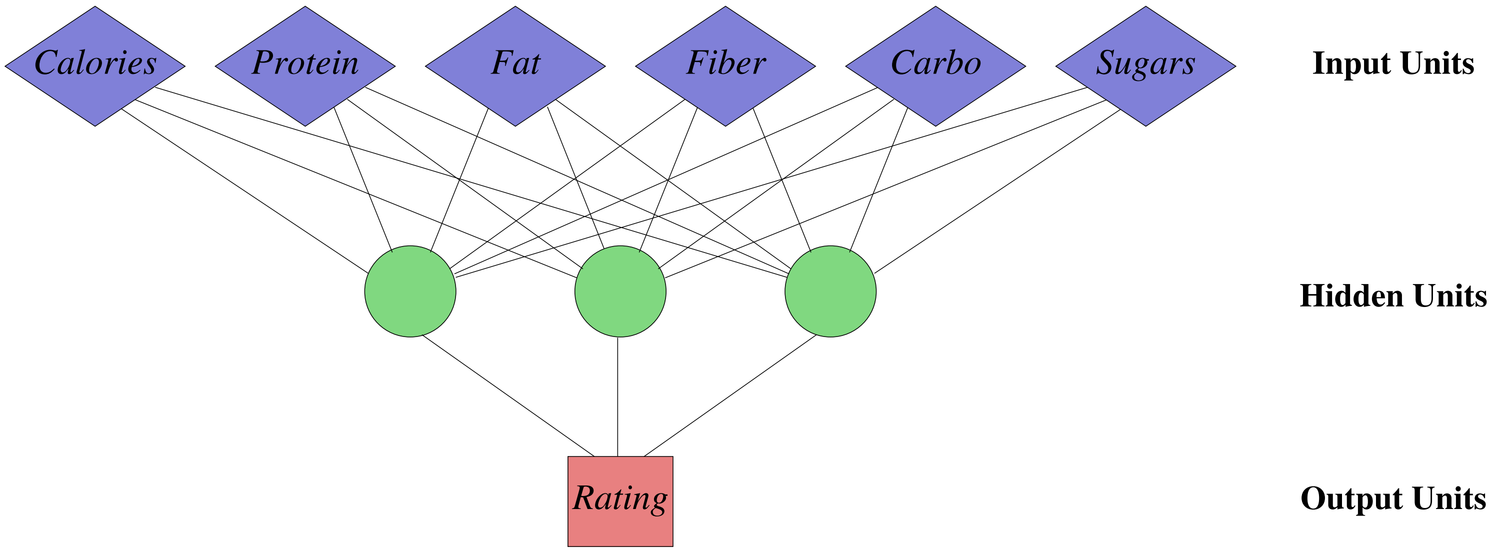
Question

What is the issue with Linear Regression?

→ **When the function is not linear, we cannot represent it!**

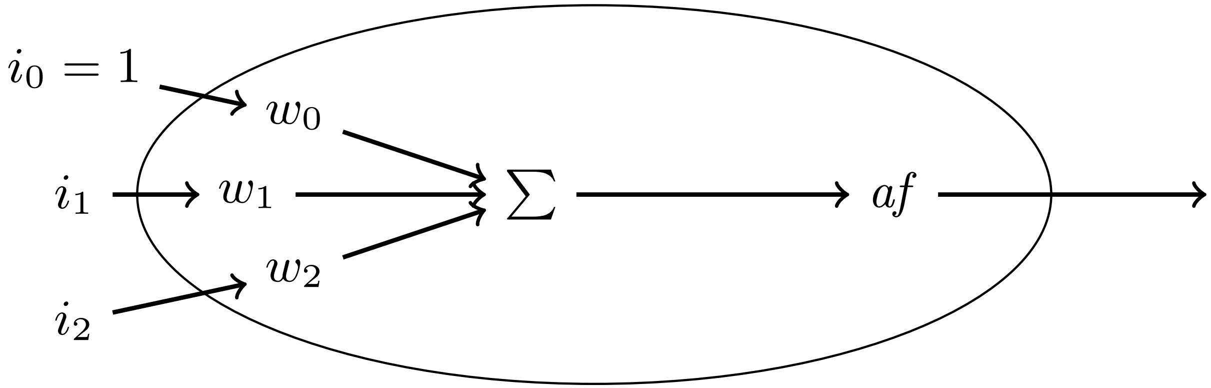
Neural Networks: Overview

Name	Calories	Protein	Fat	Fiber	Carbo	Sugars	Rating
All-Bran	70	4	1	9	7	5	59.42
Almond_Delight	110	2	2	1	14	8	34.38
Apple_Jacks	110	2	0	1	11	14	33.17
Basic_4	130	3	2	2	18	8	37.03
Bran_Chex	90	2	1	4	15	6	49.12
Bran_Flakes	90	3	0	5	13	5	53.31
Cap_n_Crunch	120	1	2	0	12	12	18.04
Cheerios	110	6	2	2	17	1	50.76
...	...	...	...	...	...	...	...



- Layered network of computational **units** (or *neurons*)
- Each unit has outputs of all units in preceding layer as inputs
- With each connection in the network there is an associated **weight**

Single Neuron



$$af(\sum_j \mathbf{i}_i \cdot \mathbf{w}_j)$$

Two step computation:

- Combine inputs as *weighted sum*
- Compute output by **activation function** of combined inputs

Remember our convention: the input i_0 is always the constant value 1

So:

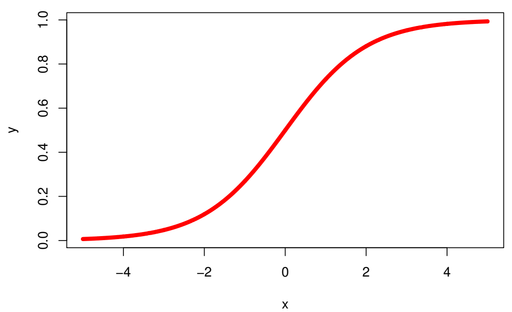
1. **Each neuron applies a linear function (like in linear Regression) and then the activation function**
2. And combine the result of multiple neurons, each with their own weights!

Activation Functions

The most common activation functions are:

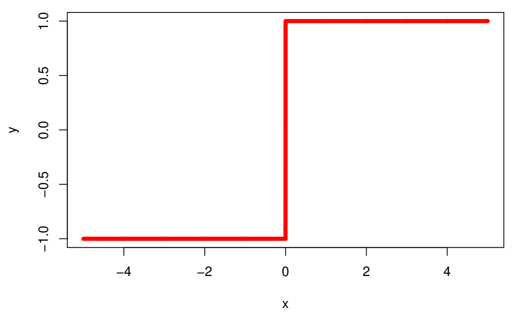
Sigmoid

$$af(x) = \sigma(x) = 1/(1+e^{-x})$$



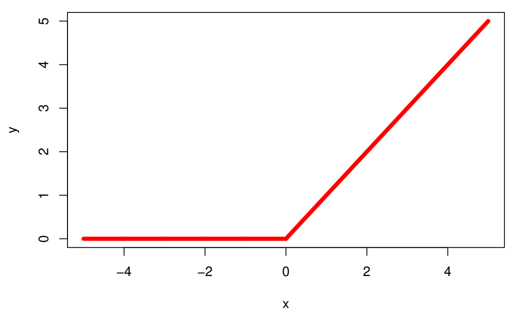
Sign

$$af(x) = sign(x)$$



Relu

$$af(x) = \max(0, x)$$



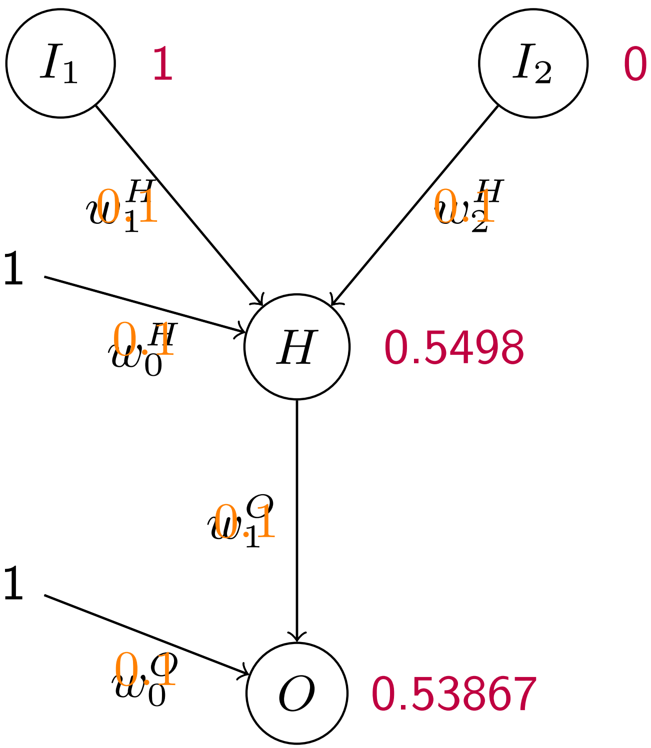
- If activation function is sigmoid, i.e. $out = \sigma(\sum_j i_j \cdot w_j)$, we also talk of *squashed linear function*.
- For the output neuron also the **identity function** is used: $af(x) = id(x) = x$

$\rightarrow$ As we will see in the next section, it is important for activation functions to be differentiable almost everywhere

Propagation in Neural Networks

- Put value of the input on input neurons
- Compute the output for each neuron, layer by layer

Example with all weights set to 0.1, activation function σ, and an input of $I_1 = 1$ and $I_2 = 0$:



The output of neuron H is:

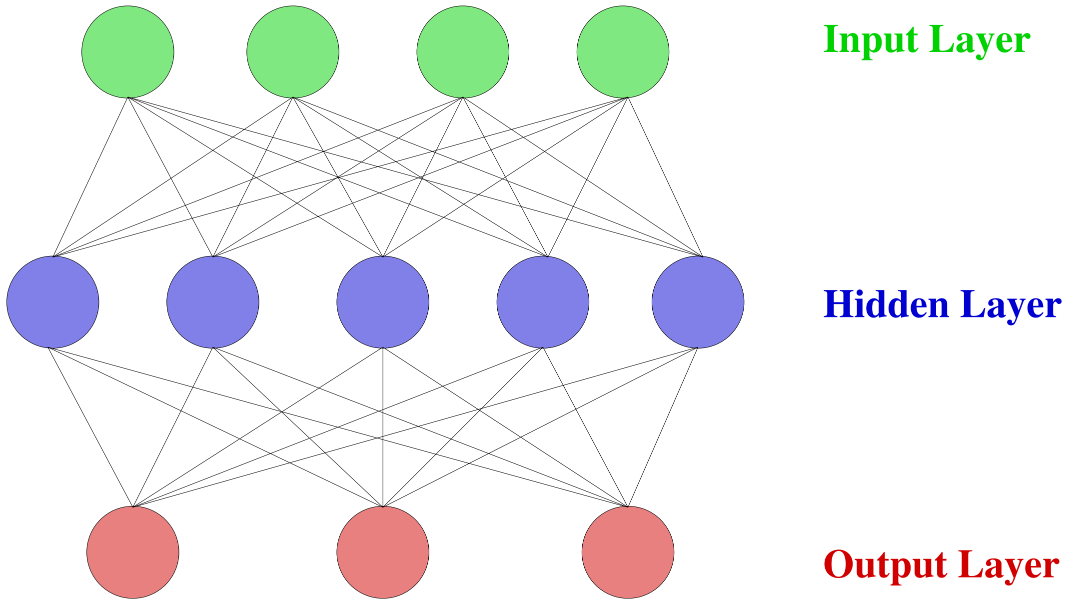
$$o_H = \sigma(1 \cdot 0.1 + 0 \cdot 0.1 + 1 \cdot 0.1) = 0.5498$$

The output of neuron O is:

$$o_O = \sigma(1 \cdot 0.1 + 0.5498 \cdot 0.1) = 0.53867$$

$\rightarrow$So the NN with these weights represents a function where $f(1, 0) = 0.5498$

Neural Network Semantics



Given

- the network structure,
- the weights associated with links/nodes,
- the activation function (usually the same for all hidden/output nodes)

a neural network with n input and k output nodes defines k real-valued functions on continuous input attributes:

$$o_i(a_1, \dots, a_n) \in \mathbb{R} \qquad (i = 1, \dots, k).$$

The Task of Learning Neural Networks

Given: structure and activation functions. To be learned: weights.
Goal: given the training examples, find the weights that minimize the *sum of squared errors (SSE)*

Note: When NN have multiple output neurons we are learning multiple functions at the same time!

Input				Output			
X_1	X_2	$\ldots$	X_N	Y_1	Y_2	$\ldots$	Y_L
$x_{1,1}$	$x_{2,1}$	$\ldots$	$x_{N,1}$	$y_{1,1}$	$y_{2,1}$	$\ldots$	$y_{L,1}$
$x_{1,2}$	$x_{2,2}$	$\ldots$	$x_{N,2}$	$y_{1,2}$	$y_{2,2}$	$\ldots$	$y_{L,2}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$x_{1,M}$	$x_{2,M}$	$\ldots$	$x_{N,M}$	$y_{1,M}$	$y_{2,M}$	$\ldots$	$y_{L,M}$

$$\sum_{i=1}^{M}\sum_{j=1}^{L}\left(y_{j,i}-o_{j,i}\right)^2,$$

where $o_{j,i}$ is the value of the jth output neuron for the ith data instance.

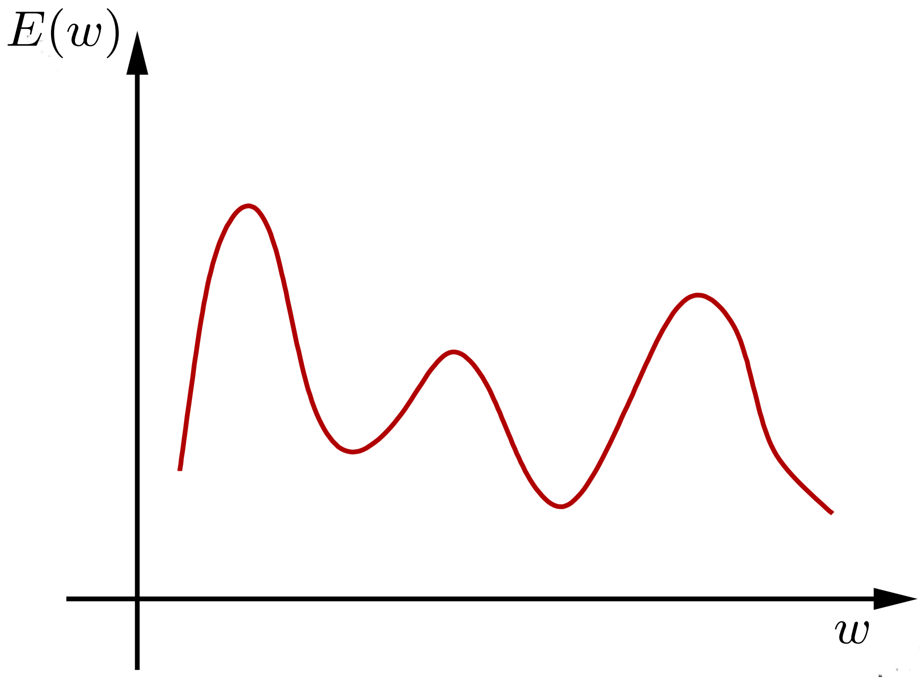
Gradient Descent for Multilayer NN

As for perceptron with SSE error:

- Error is smooth function of weights $\mathbf{w}$
- Can use gradient descent to optimize weights

but:

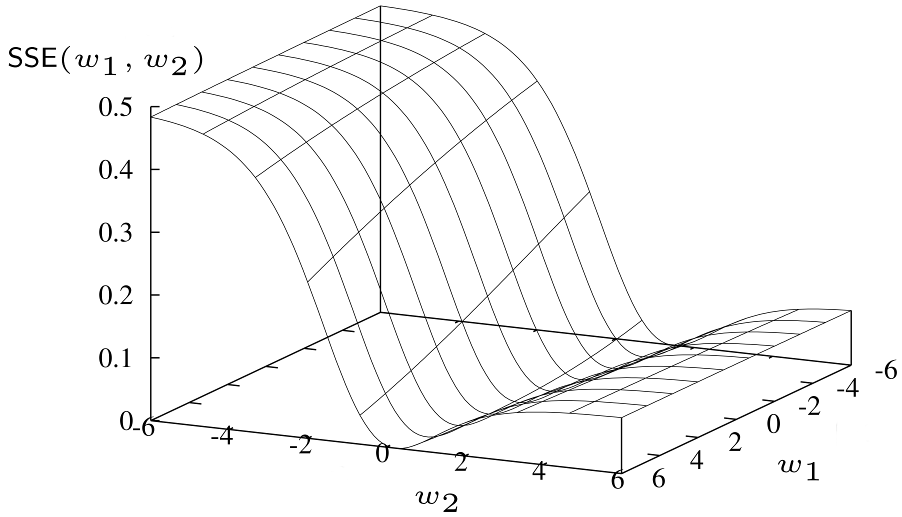
- Error no longer convex, can have multiple local minima:



- Partial derivatives more difficult to compute

Gradient Descent in Neural Networks

Basic principle: Same as in Gradient Descent for Linear Regression. *SSE* is a differentiable function of the weights (for differentiable activation functions such as the sigmoid function!). Use *gradient descent* to optimize SSE:



$$\nabla SSE(\mathbf{w}) = \left(\frac{\partial SSE}{\partial w_0}, \dots, \frac{\partial SSE}{\partial w_n}\right)$$

specifies the direction of steepest increase in SSE.

Hence, our new training rule becomes:

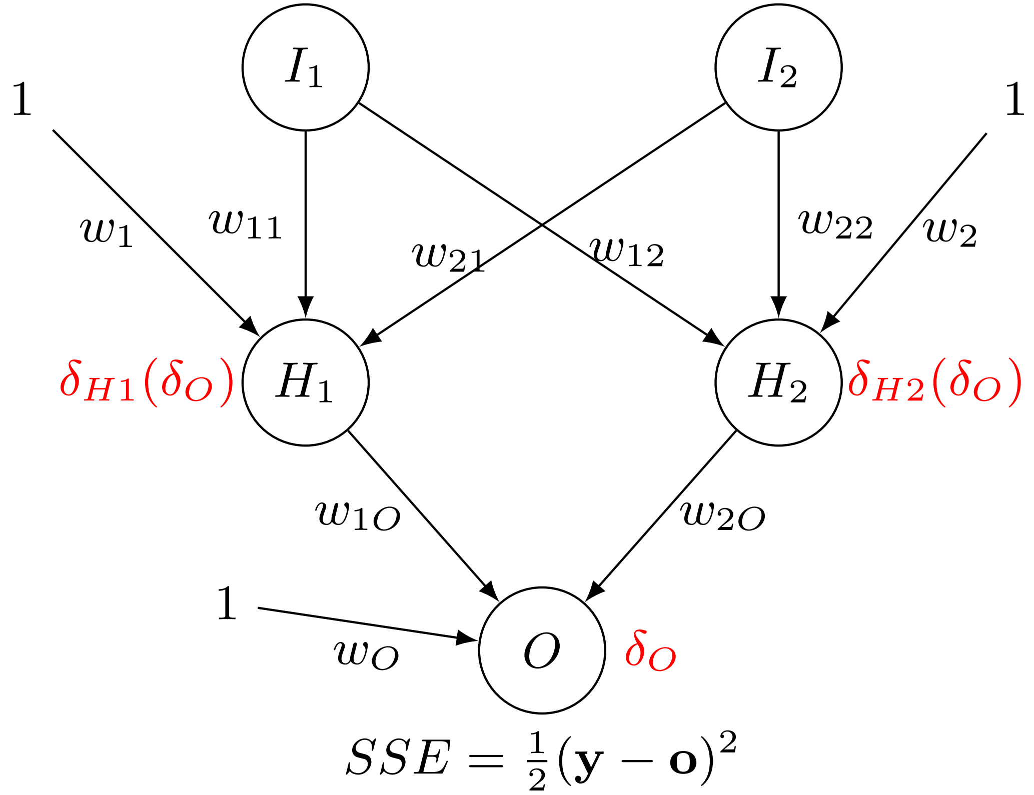
$$w_i := w_i + \Delta w_i,$$

where

$$\Delta w_i = -\alpha \frac{\partial SSE}{\partial w_i}$$

How to Update Weights in Neural Networks: Backpropagation

Issue: Training examples provide target values for only network outputs, so no target values are directly available for indicating the error of the hidden units' values.



Idea: Calculate an error term δ_h for each hidden unit by taking the weighted sum of the error terms, δ_k, for each output units it influences.

$\rightarrow$ **Backpropagation**: The error terms are (back-)propagated from the output layer towards the input layer

Updating Rules with Sigmoid Activation Function

When using a sigmoid activation function we can derive the following updating rule to update w_{ab} the weight between neurons a and b:

$$w_{ab}^{\text{new}} := w_{ab}^{\text{current}} + \alpha \cdot \delta_b \cdot x_{ab},$$

learning rate

input to the neuron b via that link

error term for neuron b

where

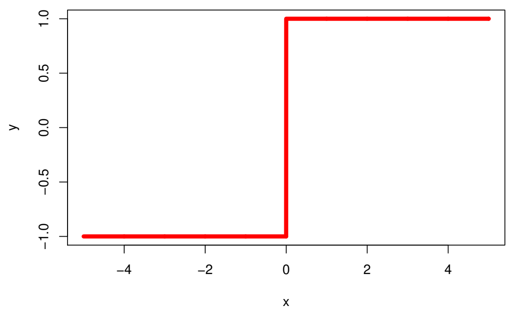
$$\delta_b = \begin{cases} o_b(1 - o_b)(y - o_b) & \text{for output nodes,} \\ o_b(1 - o_b)\sum_c w_{bc}\delta_c & \text{for hidden nodes.} \end{cases}$$

Question!

Where this comes from?

Updating Rules with Sign Activation Function

The sign activation function has 0 as derivative, so it is not suitable for gradient descent.



If this is used only in the output neuron, we can ignore the sign function, applying the intuitive rule in slide 9.

$$w_{ab}^{\text{new}} := w_{ab}^{\text{current}} + \alpha \cdot (y - o_b) \cdot x_{ab},$$

learning rate

input to the neuron b via that link

error term for neuron b

Backpropagation: Example

- Propagate value forward (as in slide 20)
- Compute error terms, propagating them backwards
- Update weights

Discrete Attributes

So far, all are attributes were numeric. What if we have discrete attributes?
Example: Manufacturer

Calories	*Protein*	*Sugars*	*Vitamins*	*Manufacturer*	*Rating*
70	105	8	135	Kellogs	59.3
110	80	23	99	Nabisco	43.6
...	...	...	...	...	...

They could be an input or a target attribute

Neural networks can also handle discrete attributes!
Next, we see two different ways of encoding discrete attributes:

1. Numerical Encoding
2. Indicator Variables

Numerical Encoding of Discrete Attributes

Translate values into numbers, e.g.:

- *True*, *False* $\mapsto$ 1,0
- *Low*, *Medium*, *High* $\mapsto$ 0,1,2

Example: Manufacturer

Calories	*Protein*	*Sugars*	*Vitamins*	*Manufacturer*	*Rating*
70	105	8	135	Kellogs0	59.3
110	80	23	99	Nabisco1	43.6
...	...	...	...	...	...

- *Kellogs*, *Nabisco*, *Bells* $\mapsto$ 0,1,2
- *Red*, *Green*, *Blue*, *Pink*, ... $\mapsto$ 0,1,2,3,...

Not a great way in these cases because *blue* is not "two times *green*"
$\rightarrow$ When a numerical encoding is not sensible, indicator variables are preferred

Indicator Variables

Replace discrete attributes with a 0-1-valued **indicator variables** for each possible value:

- For each value x_i of X with domain $\{x_1, \ldots, x_k\}$ introduce a binary feature *X_is_*x_i with values 0,1.
- Encode input $X = x_i$ by inputs

$$\textit{X\_is\_}x_0 = 0, \ldots, \textit{X\_is\_}x_{i-1} = 0, \textit{X\_is\_}x_i = 1, \textit{X\_is\_}x_{i+1} = 0, \ldots, \textit{X\_is\_}x_k = 0$$

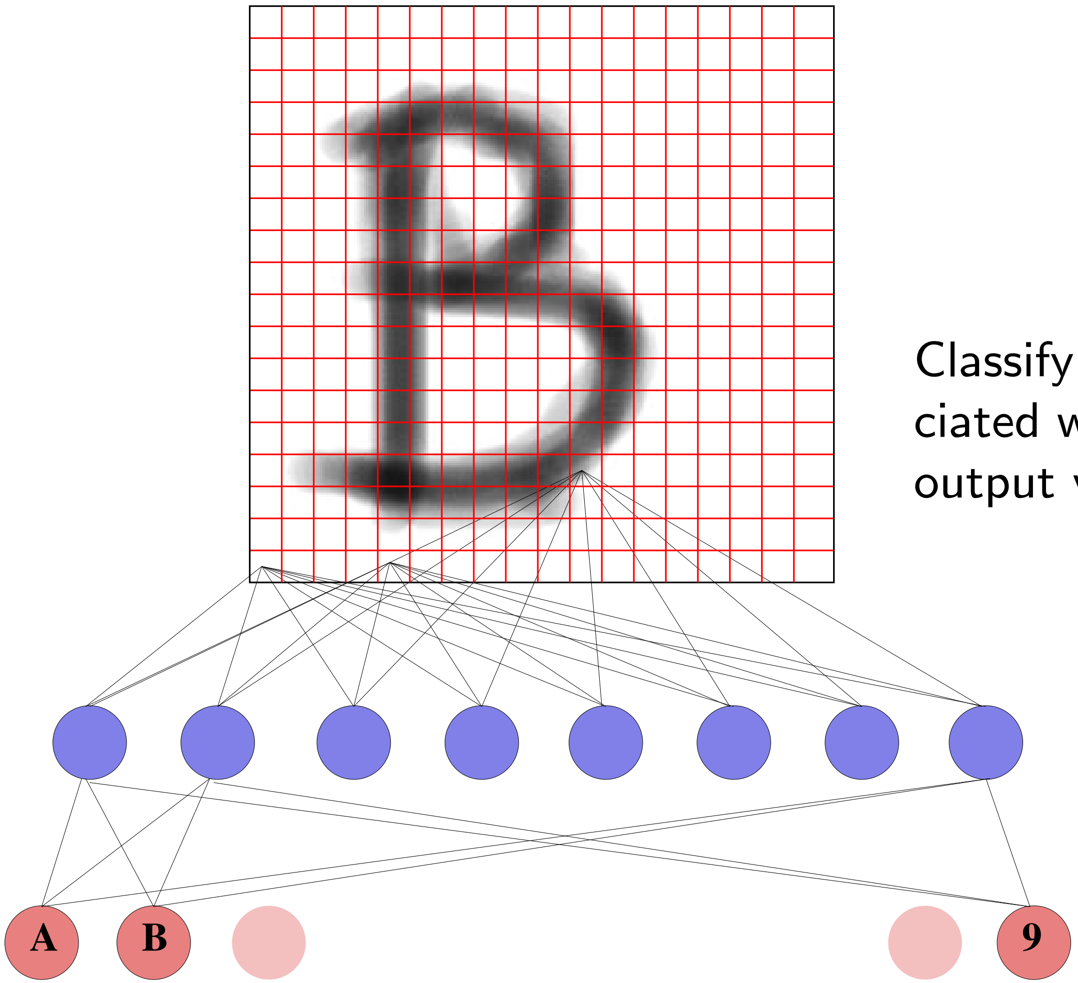
Example:

Calories	*Protein*	*Sugars*	*Vitamins*	*M_Kellogs*	*M_Nabisco*	*M_xxx*	*Rating*
70	105	8	135	1	0	...	59.3
110	80	23	99	0	1	...	43.6
...	...	...	...	...	...	...	...

Neural Networks for Classification

We can use indicator variables and learning multiple functions: one output node for each class label!

Example Task: hand-written character recognition. Predictor attributes: (continuous) grey-scale values for 18×18 grid cells. Class label: one of A,. . . ,Z,0,. . . ,9.



Classify instance by class label associated with output node with highest output value.

Expressive Power in Classification

Question

What are Neural Networks capable of?

To address this question, we consider the following simplified setting:

- Assume all inputs are Boolean (-1 or 1)
- Assume we have a single output (-1 or 1)

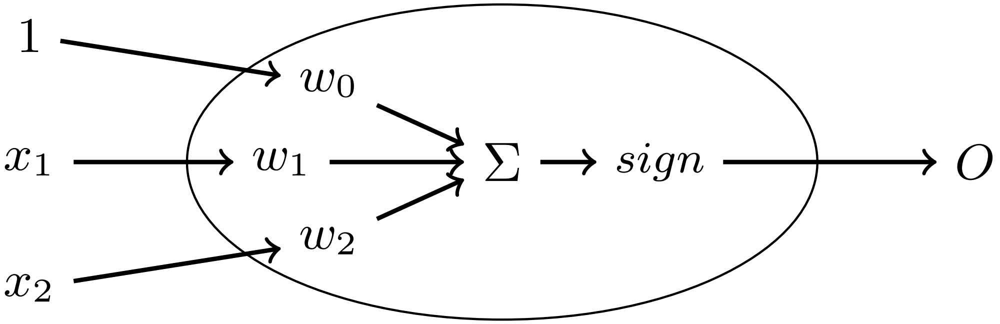
$\rightarrow$**This corresponds to representing arbitrary logical formulas!**

Questions

1. Are Neural Networks capable of representing any logical formula?
2. Do we need more than one neuron to do that?

The perceptron

The perceptron is an algorithm for supervised learning of binary classifiers.

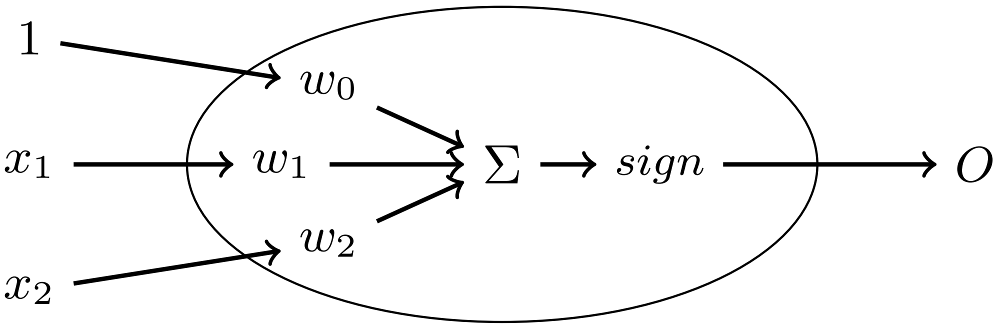


- No hidden layer
- One output neuron o
- *sign* activation function

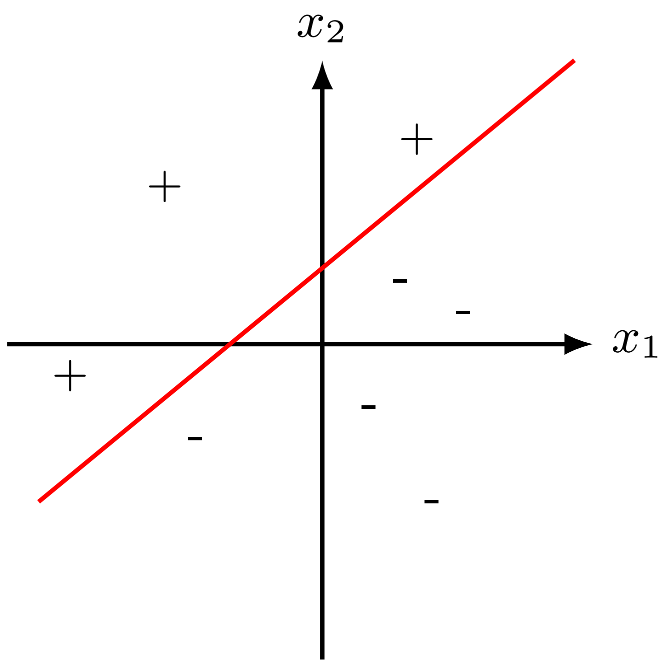
Function computed:

$$O(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \ldots w_n x_n > 0 \\ -1 & \text{otherwise} \end{cases}$$

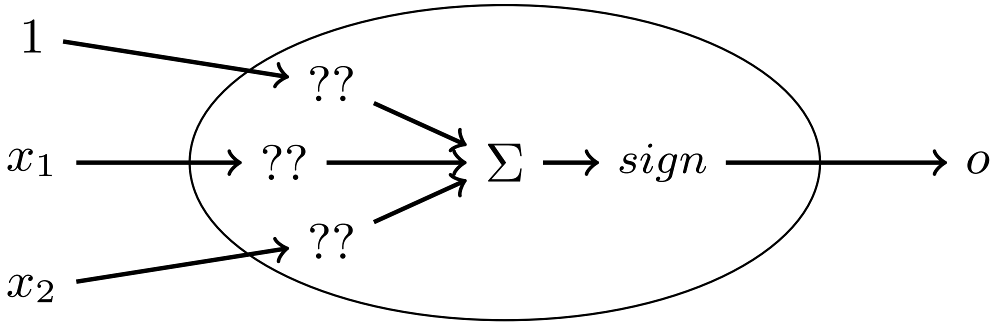
The perceptron for Classification tasks



The decision surface of a two-input perceptron $a(x_1, x_2) = \text{sign}(x_1 \cdot w_1 + x_2 \cdot w_2 + w_0)$ is given by a straight line, separating positive and negative examples.



Expressive power: Representing the conjuntion formula



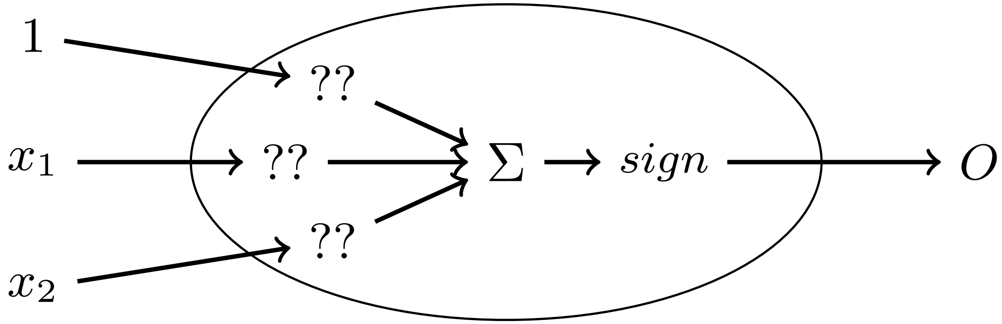
		$x_1 = -1$	$x_1 = 1$
x_2	-1	-1	-1
	1	-1	1

$x_1 \wedge x_2$

Question

Can the perceptron represent the conjunction Boolean function $x_1 \wedge x_2$?

Expressive power: Representing the disjuntion formula



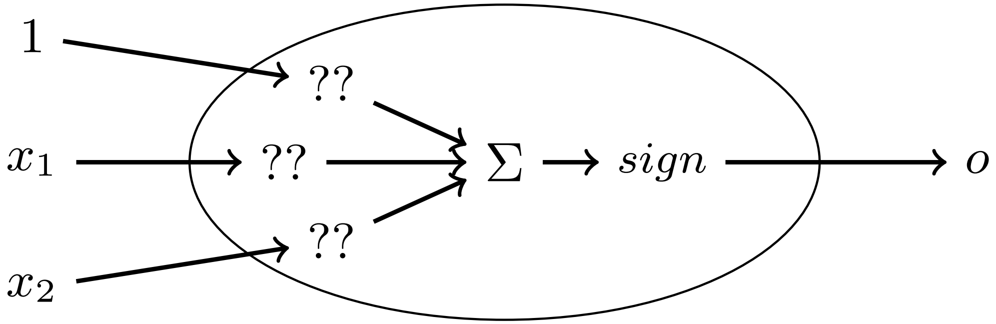
		x_1	
		-1	1
x_2	-1	-1	1
	1	1	1

$x_1 \lor x_2$

Question

Can the perceptron represent the disjunction Boolean function $x_1 \lor x_2$?

Expressive power: The XOR case



		x_1	
		-1	1
x_2	-1	-1	1
	1	1	-1

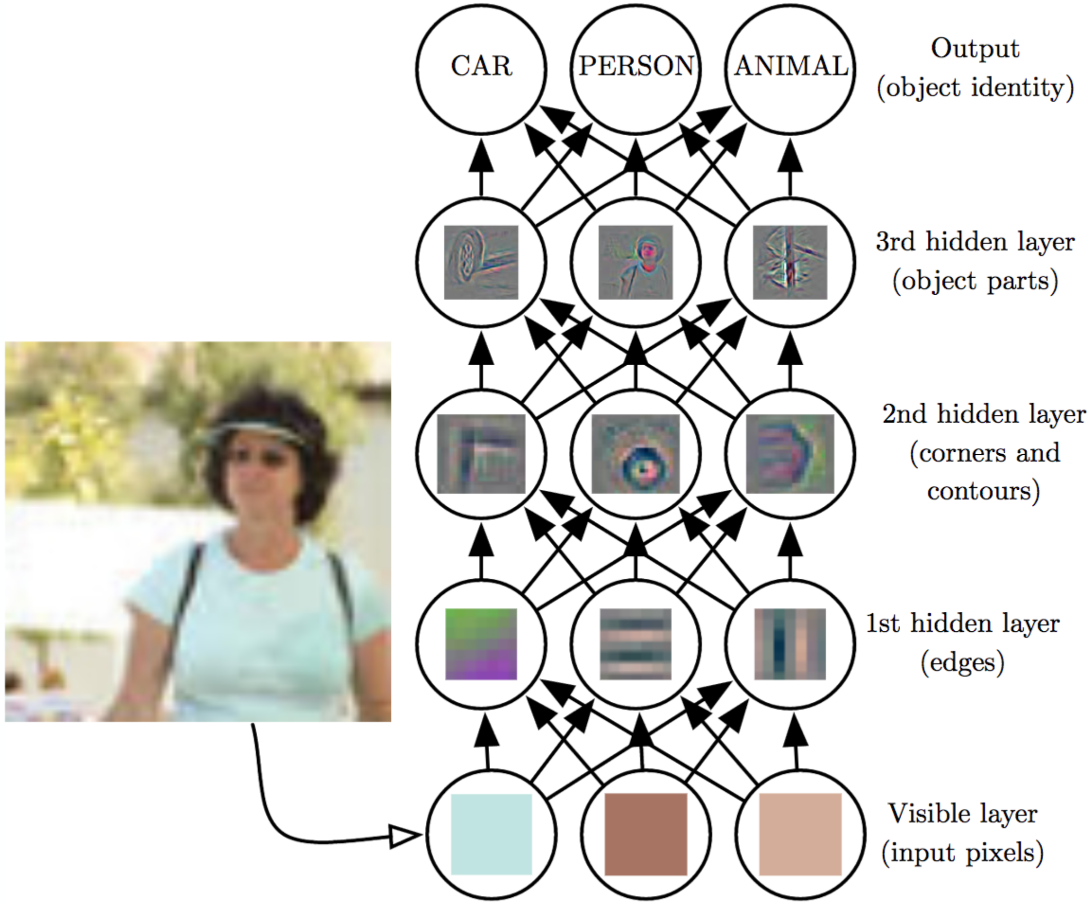
x_1 xor x_2

Question

Can the perceptron represent the disjunction Boolean function $x_1 \oplus x_2$?

Representing XOR with Neural Networks

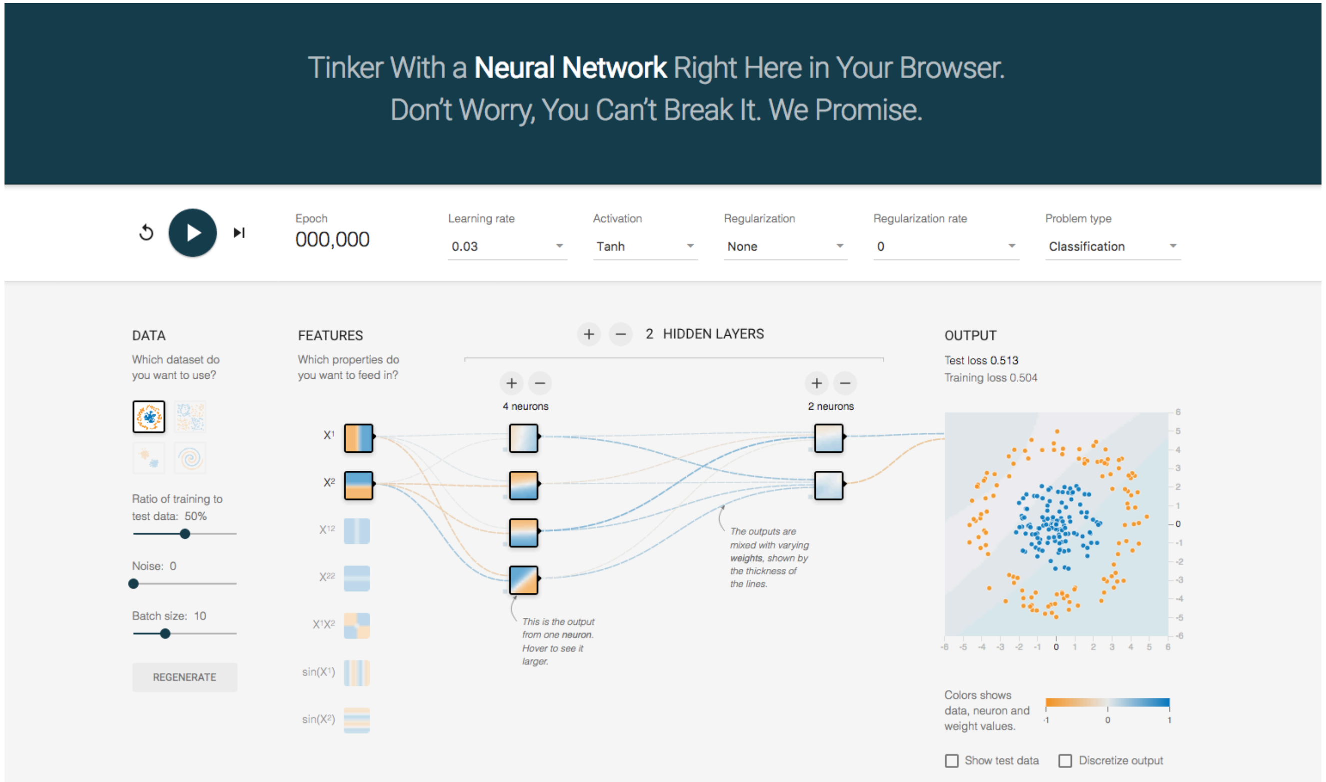
Depth: repeated composition



- Neurons in hidden layers represent more complex features
- Neurons in hidden layers are shared among multiple output functions:

  →Something this eases learning: features that help to predict an output, also help for another

Playground



playground.tensorflow.org

Summary

- **Neural Networks** can represent arbitrary functions by repeteadly combining individual networks.
- **Multilayer perceptron** has an input layer, a number of hidden layer and an output layer.
- We can find the function that best approximates our examples (reduces the SSE error) by using **gradient descent**.
- **Gradient descent** adjusts the parameters/weights iteratively by going in the direction that reduces the error.
- **Backpropagation** can be used to know how to adjust the weight in hidden neurons, by backpropagating the error they will cause on subsequent neurons.

Reading

- *Chapter 7: Supervised Machine Learning* from the book "Artificial Intelligence:Foundations of Computational Agents (2nd edition)
  In particular:
  - *Chapter 4.9.2 Local Search for Optimization* (Gradient Descent)
  - *Chapter 7.3.2 Linear Regression and Classification*
  - *Chapter 7.5: Neural Networks and Deep Learning*
- Extra Reading: To go further, you can read the Lecture Notes of the Stanford Course in Machine Learning, Chapter 7.
- Also, it is very recommendable the video series by 3Blue1Brown: `https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi`