The Perceptron

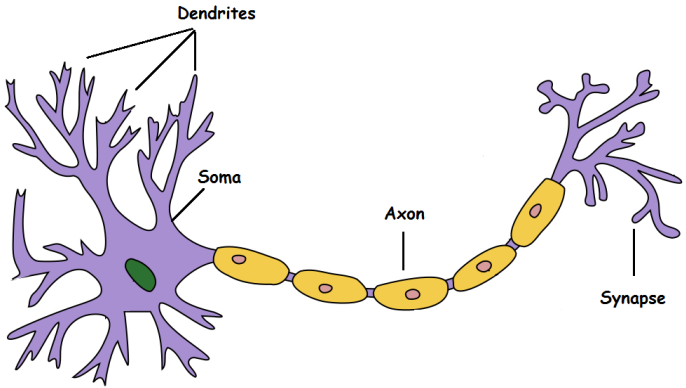
A **linear classifier** inspired by the human **neuron**

Faculty of Mathematics and Computer Science, University of Bucharest
and
Sparktech Software

Academic Year 2018/2019, 1st Semester

The Biological Neuron

- Electrical signals arrive at the neuron's **dendrites** (which can be considered *inputs*).
- The signals cause electrical potential to be accumulated in the neuron's *body* (the **soma**).
- When the potential reaches a certain *threshold*, a pulse is transmitted down the neuron's **axon** (the *output*).
- *Synapses* are connections from the *axon* to the *dendrites* of other neurons.



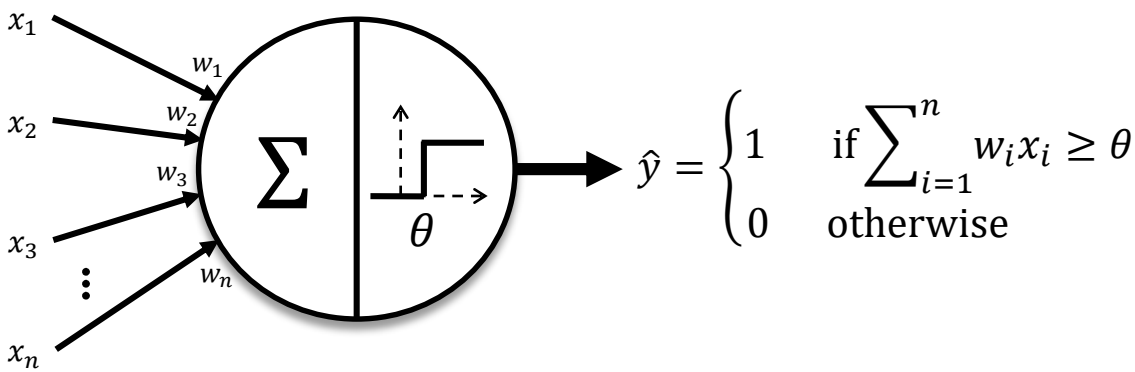
The Artificial Neuron

- The first artificial model of the neuron was proposed by **Warren MuCulloch** and **Walter Pitts** in 1943.
- It had *Boolean* inputs (either *present* or *not present*) and could be either *excitatory* or *inhibitory*.
    - If the number of present excitatory inputs were greater than the number of present inhibitory inputs by some threshold, then the neuron would fire.

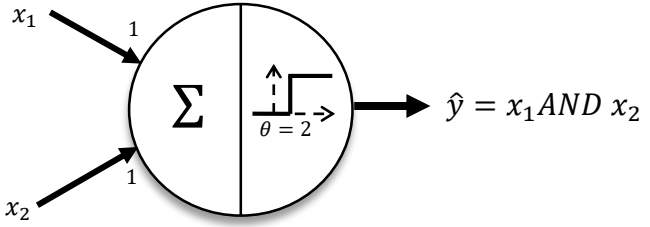
The Artificial Neuron

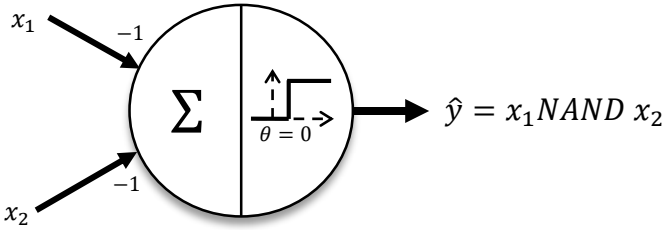
- The first artificial model of the neuron was proposed by **Warren MuCulloch** and **Walter Pitts** in 1943.
- It had *Boolean* inputs (either *present* or *not present*) and could be either *excitatory* or *inhibitory*.
  - If the number of present excitatory inputs were greater than the number of present inhibitory inputs by some threshold, then the neuron would fire.
- Mathematically, the neuron's output is 1 if the weighted sum of binary inputs would exceed a threshold and 0 otherwise.

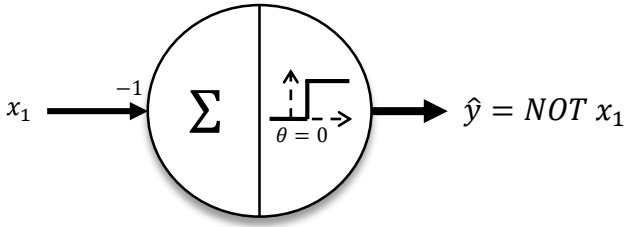
$$x_1, x_2, \dots, x_n \in \{0,1\}, \qquad w_1, w_2, \dots, w_n \in \{-1,1\}, \qquad \theta \in \mathbb{R}, \qquad \hat{y} \in \{0,1\}$$

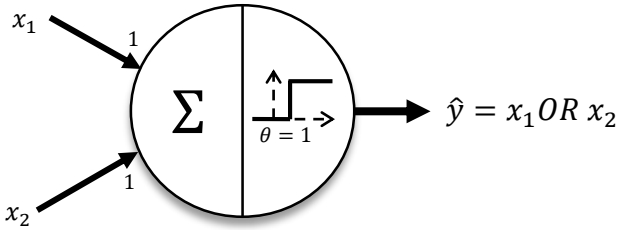


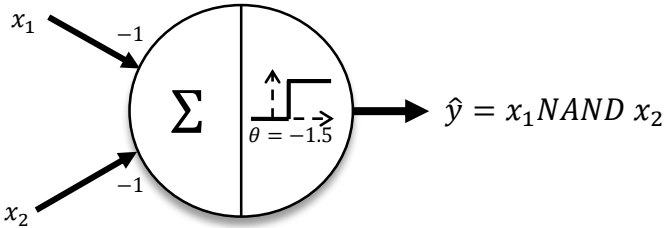
The Artificial Neuron











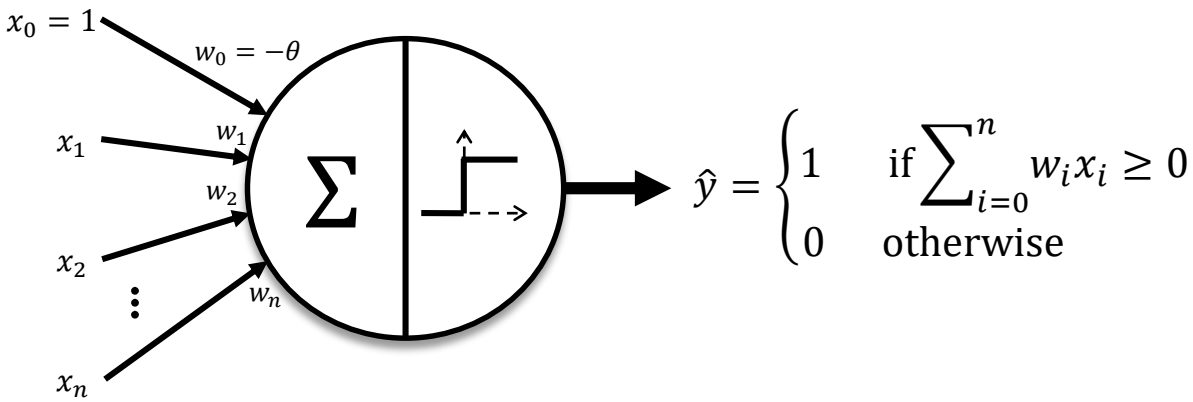
Hebb's Learning Rule

- **Hebb's rule**, postulated by *Donald Hebb* (psychologist) in 1949, states that the connections between two (biological) neurons might be strengthened, if a neuron often takes part in firing the other.
  - *"When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased".*
  - Often summarized as *"Cells that fire together wire together."*
- It gave rise to two ideas concerning the *MuCulloch-Pitts* artificial neuron model:
  - Inputs are not just *excitatory* or *inhibitory*, but have a *synaptic strength* (some connections between neurons are stronger than others).
  - It gave an indication of a method to update the synaptic strength (increase the strength of connection which is active when a neuron *should* fire).

The Perceptron

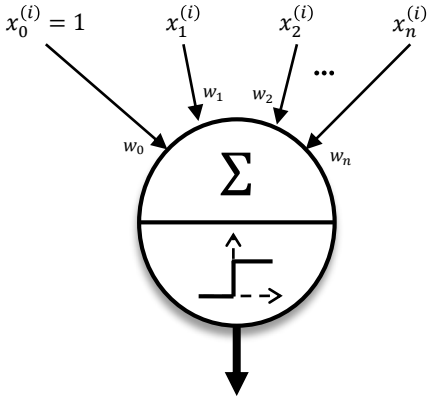
- Based on the *MuCulloch-Pitts* model and with Hebb's ideas in mind, **Frank Rosenblatt** invented, in 1957, a machine and an associated learning algorithm, which he called **"The Perceptron",** designed for image recognition.
  - It was intended to be a physical machine (with photocells as inputs, potentiometers as weights and electric motors which performed weight updates), although the first implementation was in software.

$$x_0 = 1, \qquad x_1, x_2, \dots, x_n \in \{0,1\}, \qquad w_0, w_1, w_2, \dots, w_n \in \mathbb{R}, \qquad \hat{y} \in \{0,1\}$$



Perceptron Learning Algorithm

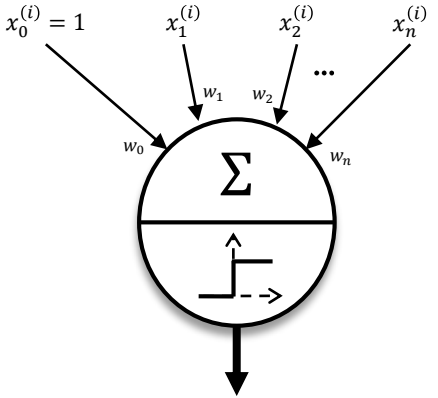
- Training Set: $E = \{(\vec{x}^{(1)}, y^{(1)}), \ldots, (\vec{x}^{(m)}, y^{(m)})\},\ \vec{x}^{(i)} \in \{0,1\}^{n+1}\left(x_0^{(i)} = 1, \forall i\right),\ y^{(i)} \in \{0,1\}$



$$\hat{y}^{(i)} = \begin{cases} 1 & \text{if } \sum_{j=0}^{n} w_j x_j^{(i)} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Perceptron Learning Algorithm

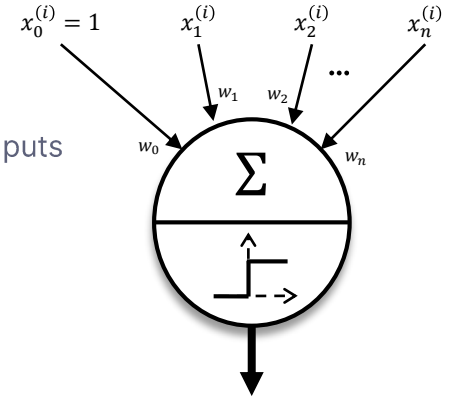
- Training Set: $E = \{(\vec{x}^{(1)}, y^{(1)}), \ldots, (\vec{x}^{(m)}, y^{(m)})\},\ \vec{x}^{(i)} \in \{0,1\}^{n+1}\left(x_0^{(i)} = 1, \forall i\right),\ y^{(i)} \in \{0,1\}$
- Weights w_j start off as 0.
- For every example $(\vec{x}^{(i)}, y^{(i)})$ in the dataset:
  - If $\hat{y}^{(i)} == y^{(i)}$ we don't change anything



$$\hat{y}^{(i)} = \begin{cases} 1 & \text{if } \sum_{j=0}^{n} w_j x_j^{(i)} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Perceptron Learning Algorithm

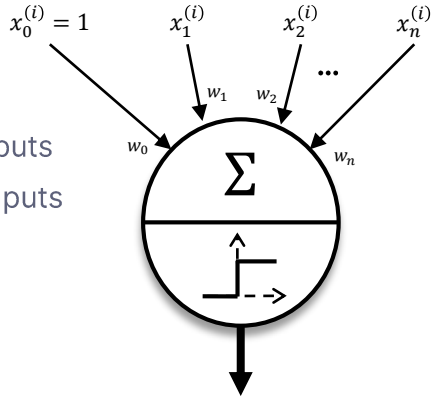
- Training Set: $E = \{(\vec{x}^{(1)}, y^{(1)}), \ldots, (\vec{x}^{(m)}, y^{(m)})\},\ \vec{x}^{(i)} \in \{0,1\}^{n+1}\left(x_0^{(i)} = 1, \forall i\right),\ y^{(i)} \in \{0,1\}$
- Weights w_j start off as 0.
- For every example $(\vec{x}^{(i)}, y^{(i)})$ in the dataset:
  - If $\hat{y}^{(i)} == y^{(i)}$ we don't change anything
  - If $\hat{y}^{(i)} == 0$ and $y^{(i)} == 1$ we need to increase the strength of all active inputs



$$\hat{y}^{(i)} = \begin{cases} 1 & \text{if } \sum_{j=0}^{n} w_j x_j^{(i)} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Perceptron Learning Algorithm

- Training Set: $E = \{(\vec{x}^{(1)}, y^{(1)}), \ldots, (\vec{x}^{(m)}, y^{(m)})\},\ \vec{x}^{(i)} \in \{0,1\}^{n+1}\left(x_0^{(i)} = 1, \forall i\right),\ y^{(i)} \in \{0,1\}$
- Weights w_j start off as 0.
- For every example $(\vec{x}^{(i)}, y^{(i)})$ in the dataset:
  - If $\hat{y}^{(i)} == y^{(i)}$ we don't change anything
  - If $\hat{y}^{(i)} == 0$ and $y^{(i)} == 1$ we need to increase the strength of all active inputs
  - If $\hat{y}^{(i)} == 1$ and $y^{(i)} == 0$ we need to decrease the strength of all active inputs



$$\hat{y}^{(i)} = \begin{cases} 1 & \text{if } \sum_{j=0}^{n} w_j x_j^{(i)} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Perceptron Learning Algorithm

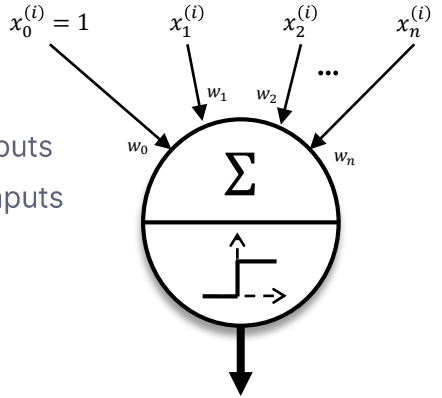
- Training Set: $E = \{(\vec{x}^{(1)}, y^{(1)}), \ldots, (\vec{x}^{(m)}, y^{(m)})\}$, $\vec{x}^{(i)} \in \{0,1\}^{n+1} \left(x_0^{(i)} = 1, \forall i\right)$, $y^{(i)} \in \{0,1\}$
- Weights w_j start off as 0.
- For every example $(\vec{x}^{(i)}, y^{(i)})$ in the dataset:
  - If $\hat{y}^{(i)} == y^{(i)}$ we don't change anything
  - If $\hat{y}^{(i)} == 0$ and $y^{(i)} == 1$ we need to increase the strength of all active inputs
  - If $\hat{y}^{(i)} == 1$ and $y^{(i)} == 0$ we need to decrease the strength of all active inputs
- Perceptron update rule:

$$\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right) \cdot x_j^{(i)}$$

Amount by which the strength is changed ("**learning rate**").

Sets the update direction.

Selects active inputs.



$$\hat{y}^{(i)} = \begin{cases} 1 & \text{if } \sum_{j=0}^{n} w_j x_j^{(i)} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Perceptron Learning Algorithm

```
1  def perceptron(X, y, n_epochs, η):
2      M, n = X.shape # number of samples, number of inputs
3      for j in range(n):
4          w_j = 0
5      for epoch in range(n_epochs): # an "epoch" is a run through all training data.
6          for i in range(m):        # a "training step" is one of update of the weights.
7              ŷ^(i) = unit_step_function(Σ_{j=0}^{n} w_j x_j^(i))
8              for j in range(n):
9                  w_j += η(y^(i) − ŷ^(i)) · x_j^(i)
```

Perceptron Learning Algorithm

- An alternative formulation is to consider inputs and outputs to be ± 1, instead of binary.

$$\vec{x}^{(i)} \in \{-1,1\}^{n+1}, \qquad y^{(i)} \in \{-1,1\}$$

- Prediction:

$$\hat{y}^{(i)} = \text{sign}\left(\sum_{j=0}^{n} w_j x_j^{(i)}\right)$$

- Update rule:

$$\Delta w_j = \eta \cdot y^{(i)} \cdot x_j^{(i)}$$

We only apply the update rule if $y^{(i)} \neq \hat{y}^{(i)}$

Example – Learning the OR function

	x_0	x_1	x_2	y
$\vec{x}^{(1)}$	1	0	0	0
$\vec{x}^{(2)}$	1	0	1	1
$\vec{x}^{(3)}$	1	1	0	1
$\vec{x}^{(4)}$	1	1	1	1

Example – Learning the OR function

	x_0	x_1	x_2
$\vec{x}^{(1)}$	1	0	0
$\vec{x}^{(2)}$	1	0	1
$\vec{x}^{(3)}$	1	1	0
$\vec{x}^{(4)}$	1	1	1

y
0
1
1
1

w_0	w_1	w_2
0	0	0

Accuracy: 25%

$\hat{y}$
0
0
0
0

$\eta = 1$

$\Delta w_j = \left(y^{(i)} - \hat{y}^{(i)}\right) \cdot x_j^{(i)}$

Example – Learning the OR function

	x_0	x_1	x_2
$\vec{x}^{(1)}$	1	0	0
$\vec{x}^{(2)}$	1	0	1
$\vec{x}^{(3)}$	1	1	0
$\vec{x}^{(4)}$	1	1	1

y
0
1
1
1

w_0	w_1	w_2
0	0	0

Accuracy: 25%

$\hat{y}$
0
0
0
0

	Δw_0	Δw_1	Δw_2
$\vec{x}^{(1)}$	0	0	0
$\vec{x}^{(2)}$			
$\vec{x}^{(3)}$			
$\vec{x}^{(4)}$			

Epoch 1

$\eta = 1$

$\Delta w_j = \left(y^{(i)} - \hat{y}^{(i)}\right) \cdot x_j^{(i)}$

Example – Learning the OR function

	x_0	x_1	x_2
$\vec{x}^{(1)}$	1	0	0
$\vec{x}^{(2)}$	1	0	1
$\vec{x}^{(3)}$	1	1	0
$\vec{x}^{(4)}$	1	1	1

y
0
1
1
1

w_0	w_1	w_2
0	0	0

$\hat{y}$
0
0
0
0

Accuracy: 25%

	Δw_0	Δw_1	Δw_2
$\vec{x}^{(1)}$	0	0	0
$\vec{x}^{(2)}$	1	0	1
$\vec{x}^{(3)}$			
$\vec{x}^{(4)}$			

Epoch 1

$\eta = 1$

$\Delta w_j = \left(y^{(i)} - \hat{y}^{(i)}\right) \cdot x_j^{(i)}$

Example – Learning the OR function

	x_0	x_1	x_2
$\vec{x}^{(1)}$	1	0	0
$\vec{x}^{(2)}$	1	0	1
$\vec{x}^{(3)}$	1	1	0
$\vec{x}^{(4)}$	1	1	1

y
0
1
1
1

w_0	w_1	w_2
1	0	1

Accuracy: 75%

$\hat{y}$
1
1
1
1

	Δw_0	Δw_1	Δw_2
$\vec{x}^{(1)}$	0	0	0
$\vec{x}^{(2)}$	1	0	1
$\vec{x}^{(3)}$			
$\vec{x}^{(4)}$			

Epoch 1

$\eta = 1$

$\Delta w_j = \left(y^{(i)} - \hat{y}^{(i)}\right) \cdot x_j^{(i)}$

Example – Learning the OR function

	x_0	x_1	x_2
$\vec{x}^{(1)}$	1	0	0
$\vec{x}^{(2)}$	1	0	1
$\vec{x}^{(3)}$	1	1	0
$\vec{x}^{(4)}$	1	1	1

y
0
1
1
1

w_0	w_1	w_2
1	0	1

Accuracy: 75%

$\hat{y}$
1
1
1
1

	Δw_0	Δw_1	Δw_2
$\vec{x}^{(1)}$	0	0	0
$\vec{x}^{(2)}$	1	0	1
$\vec{x}^{(3)}$	0	0	0
$\vec{x}^{(4)}$			

Epoch 1

$\eta = 1$

$\Delta w_j = \left(y^{(i)} - \hat{y}^{(i)}\right) \cdot x_j^{(i)}$

Example – Learning the OR function

	x_0	x_1	x_2
$\vec{x}^{(1)}$	1	0	0
$\vec{x}^{(2)}$	1	0	1
$\vec{x}^{(3)}$	1	1	0
$\vec{x}^{(4)}$	1	1	1

y
0
1
1
1

w_0	w_1	w_2
1	0	1

$\hat{y}$
1
1
1
1

Accuracy: 75%

	Δw_0	Δw_1	Δw_2
$\vec{x}^{(1)}$	0	0	0
$\vec{x}^{(2)}$	1	0	1
$\vec{x}^{(3)}$	0	0	0
$\vec{x}^{(4)}$	0	0	0

Epoch 1

$\eta = 1$

$\Delta w_j = \left(y^{(i)} - \hat{y}^{(i)}\right) \cdot x_j^{(i)}$

Example – Learning the OR function

	x_0	x_1	x_2
$\vec{x}^{(1)}$	1	0	0
$\vec{x}^{(2)}$	1	0	1
$\vec{x}^{(3)}$	1	1	0
$\vec{x}^{(4)}$	1	1	1

y
0
1
1
1

w_0	w_1	w_2
1	0	1

Accuracy: 75%

$\hat{y}$
1
1
1
1

	Δw_0	Δw_1	Δw_2
$\vec{x}^{(1)}$	-1	0	0
$\vec{x}^{(2)}$			
$\vec{x}^{(3)}$			
$\vec{x}^{(4)}$			

Epoch 2

$\eta = 1$

$\Delta w_j = \left(y^{(i)} - \hat{y}^{(i)}\right) \cdot x_j^{(i)}$

Example – Learning the OR function

	x_0	x_1	x_2
$\vec{x}^{(1)}$	1	0	0
$\vec{x}^{(2)}$	1	0	1
$\vec{x}^{(3)}$	1	1	0
$\vec{x}^{(4)}$	1	1	1

y
0
1
1
1

w_0	w_1	w_2
0	0	1

Accuracy: 75%

$\hat{y}$
0
1
0
1

	Δw_0	Δw_1	Δw_2
$\vec{x}^{(1)}$	-1	0	0
$\vec{x}^{(2)}$			
$\vec{x}^{(3)}$			
$\vec{x}^{(4)}$			

Epoch 2

$\eta = 1$

$\Delta w_j = \left(y^{(i)} - \hat{y}^{(i)}\right) \cdot x_j^{(i)}$

Example – Learning the OR function

	x_0	x_1	x_2
$\vec{x}^{(1)}$	1	0	0
$\vec{x}^{(2)}$	1	0	1
$\vec{x}^{(3)}$	1	1	0
$\vec{x}^{(4)}$	1	1	1

y
0
1
1
1

w_0	w_1	w_2
0	0	1

Accuracy: 75%

$\hat{y}$
0
1
0
1

	Δw_0	Δw_1	Δw_2
$\vec{x}^{(1)}$	-1	0	0
$\vec{x}^{(2)}$	0	0	0
$\vec{x}^{(3)}$			
$\vec{x}^{(4)}$			

Epoch 2

$\eta = 1$

$\Delta w_j = \left(y^{(i)} - \hat{y}^{(i)}\right) \cdot x_j^{(i)}$

Example – Learning the OR function

	x_0	x_1	x_2
$\vec{x}^{(1)}$	1	0	0
$\vec{x}^{(2)}$	1	0	1
$\vec{x}^{(3)}$	1	1	0
$\vec{x}^{(4)}$	1	1	1

y
0
1
1
1

w_0	w_1	w_2
0	0	1

$\hat{y}$
0
1
0
1

Accuracy: 75%

	Δw_0	Δw_1	Δw_2
$\vec{x}^{(1)}$	-1	0	0
$\vec{x}^{(2)}$	0	0	0
$\vec{x}^{(3)}$	1	1	0
$\vec{x}^{(4)}$			

Epoch 2

$\eta = 1$

$\Delta w_j = \left(y^{(i)} - \hat{y}^{(i)}\right) \cdot x_j^{(i)}$

Example – Learning the OR function

	x_0	x_1	x_2
$\vec{x}^{(1)}$	1	0	0
$\vec{x}^{(2)}$	1	0	1
$\vec{x}^{(3)}$	1	1	0
$\vec{x}^{(4)}$	1	1	1

y
0
1
1
1

w_0	w_1	w_2
1	1	1

Accuracy: 75%

$\hat{y}$
1
1
1
1

	Δw_0	Δw_1	Δw_2
$\vec{x}^{(1)}$	-1	0	0
$\vec{x}^{(2)}$	0	0	0
$\vec{x}^{(3)}$	1	1	0
$\vec{x}^{(4)}$			

Epoch 2

$\eta = 1$

$\Delta w_j = \left(y^{(i)} - \hat{y}^{(i)}\right) \cdot x_j^{(i)}$

Example – Learning the OR function

	x_0	x_1	x_2
$\vec{x}^{(1)}$	1	0	0
$\vec{x}^{(2)}$	1	0	1
$\vec{x}^{(3)}$	1	1	0
$\vec{x}^{(4)}$	1	1	1

y
0
1
1
1

w_0	w_1	w_2
1	1	1

$\hat{y}$
1
1
1
1

Accuracy: 75%

	Δw_0	Δw_1	Δw_2
$\vec{x}^{(1)}$	-1	0	0
$\vec{x}^{(2)}$	0	0	0
$\vec{x}^{(3)}$	1	1	0
$\vec{x}^{(4)}$	0	0	0

Epoch 2

$\eta = 1$

$\Delta w_j = \left(y^{(i)} - \hat{y}^{(i)}\right) \cdot x_j^{(i)}$

Example – Learning the OR function

	x_0	x_1	x_2
$\vec{x}^{(1)}$	1	0	0
$\vec{x}^{(2)}$	1	0	1
$\vec{x}^{(3)}$	1	1	0
$\vec{x}^{(4)}$	1	1	1

y
0
1
1
1

w_0	w_1	w_2
1	1	1

Accuracy: 75%

$\hat{y}$
1
1
1
1

	Δw_0	Δw_1	Δw_2
$\vec{x}^{(1)}$	-1	0	0
$\vec{x}^{(2)}$			
$\vec{x}^{(3)}$			
$\vec{x}^{(4)}$			

Epoch 3

$\eta = 1$

$\Delta w_j = \left(y^{(i)} - \hat{y}^{(i)}\right) \cdot x_j^{(i)}$

Example – Learning the OR function

	x_0	x_1	x_2
$\vec{x}^{(1)}$	1	0	0
$\vec{x}^{(2)}$	1	0	1
$\vec{x}^{(3)}$	1	1	0
$\vec{x}^{(4)}$	1	1	1

y
0
1
1
1

w_0	w_1	w_2
0	1	1

Accuracy: 100%

$\hat{y}$
0
1
1
1

	Δw_0	Δw_1	Δw_2
$\vec{x}^{(1)}$	-1	0	0
$\vec{x}^{(2)}$			
$\vec{x}^{(3)}$			
$\vec{x}^{(4)}$			

Epoch 3

$\eta = 1$

$\Delta w_j = \left(y^{(i)} - \hat{y}^{(i)}\right) \cdot x_j^{(i)}$

Example – Learning the OR function

	x_0	x_1	x_2
$\vec{x}^{(1)}$	1	0	0
$\vec{x}^{(2)}$	1	0	1
$\vec{x}^{(3)}$	1	1	0
$\vec{x}^{(4)}$	1	1	1

y
0
1
1
1

w_0	w_1	w_2
0	1	1

$\hat{y}$
0
1
1
1

Accuracy: 100%

	Δw_0	Δw_1	Δw_2
$\vec{x}^{(1)}$	-1	0	0
$\vec{x}^{(2)}$	0	0	0
$\vec{x}^{(3)}$	0	0	0
$\vec{x}^{(4)}$	0	0	0

Epoch 3

$\eta = 1$

$\Delta w_j = \left(y^{(i)} - \hat{y}^{(i)}\right) \cdot x_j^{(i)}$

Example – Learning the OR function

	x_0	x_1	x_2
$\vec{x}^{(1)}$	1	0	0
$\vec{x}^{(2)}$	1	0	1
$\vec{x}^{(3)}$	1	1	0
$\vec{x}^{(4)}$	1	1	1

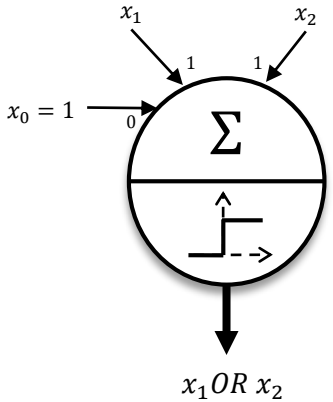
y
0
1
1
1

w_0	w_1	w_2
0	1	1

Accuracy: 100%

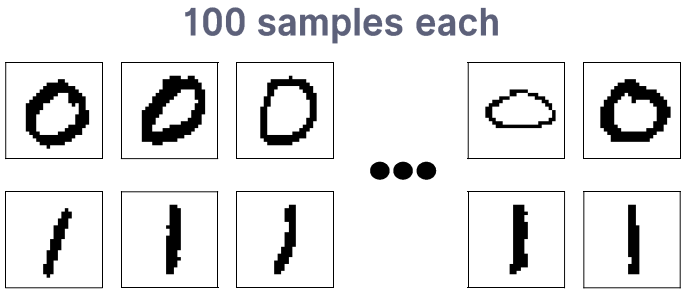
$\hat{y}$
0
1
1
1

The *Perceptron* learned the OR function in 3 epochs.



Example – Recognizing Digits

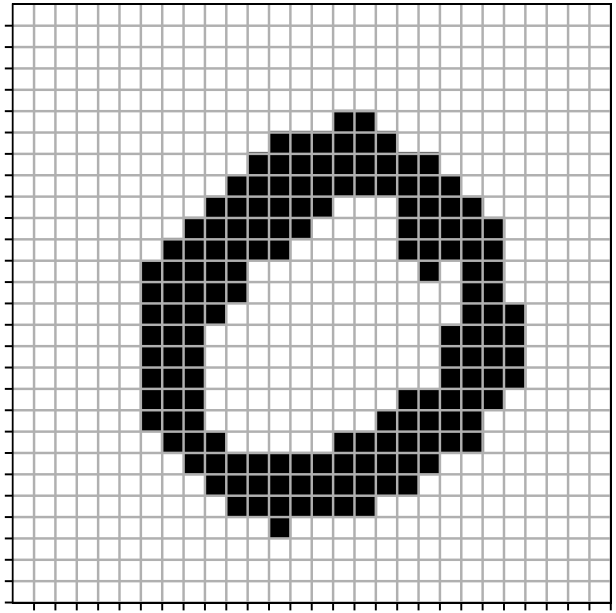
Can a *perceptron* be trained to distinguish handwritten pictures of 0s from pictures of 1s?



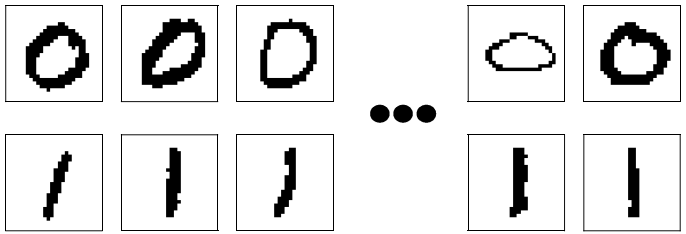
Example – Recognizing Digits

Can a *perceptron* be trained to distinguish handwritten pictures of 0s from pictures of 1s?

28x28 pixels



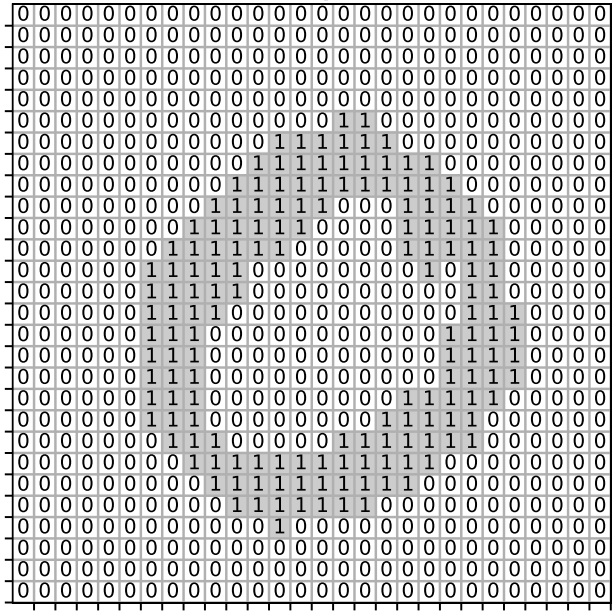
100 samples each



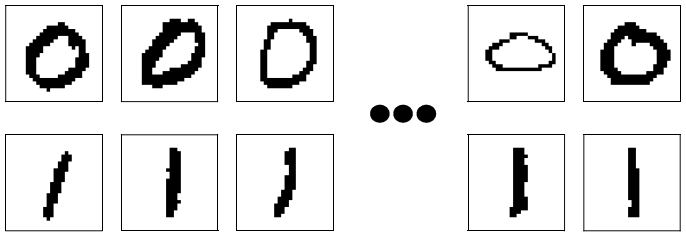
Example – Recognizing Digits

Can a *perceptron* be trained to distinguish handwritten pictures of 0s from pictures of 1s?

28x28 binary features



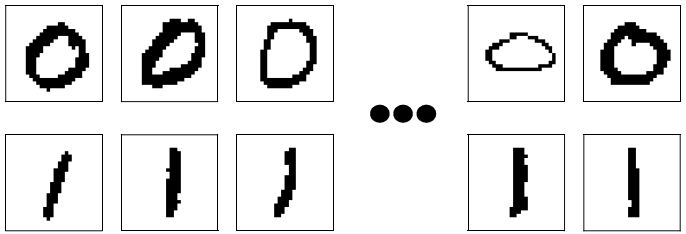
100 samples each



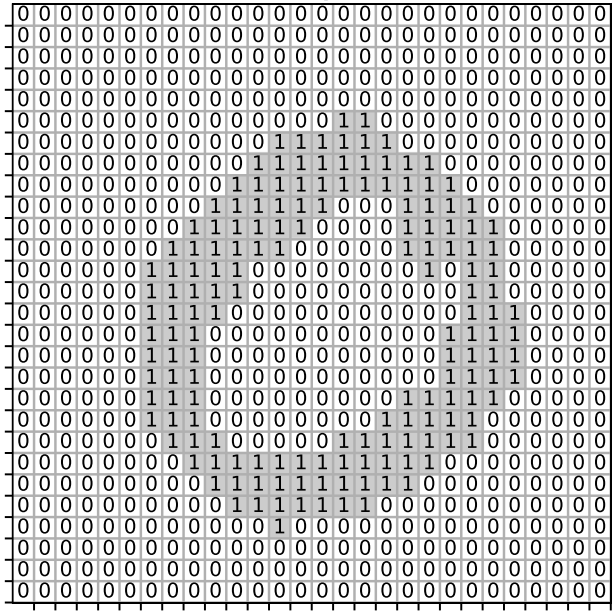
Example – Recognizing Digits

Can a *perceptron* be trained to distinguish handwritten pictures of 0s from pictures of 1s?

100 samples each

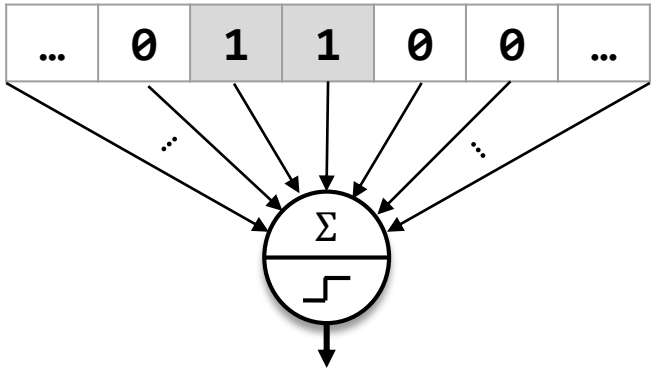


28x28 binary features

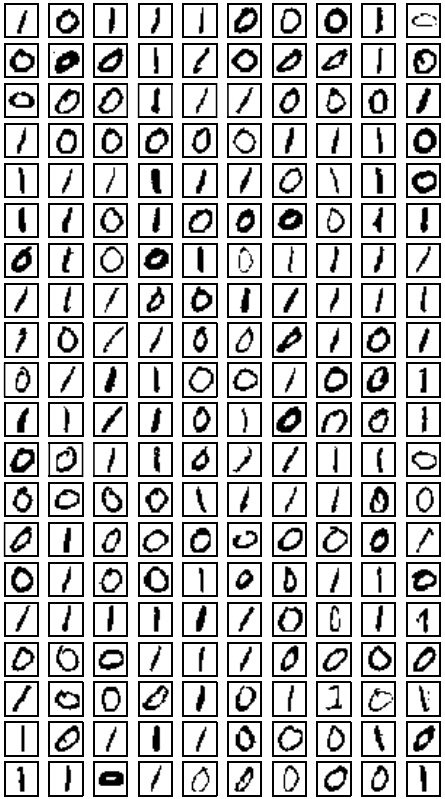


Flatten

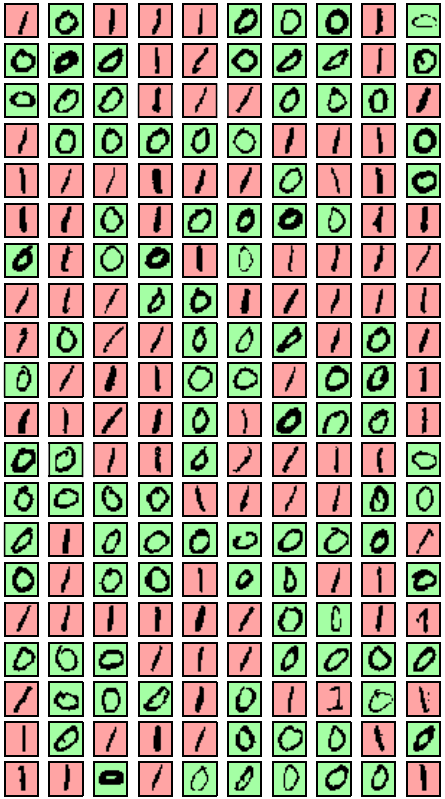
724 features (inputs)



Example – Recognizing Digits



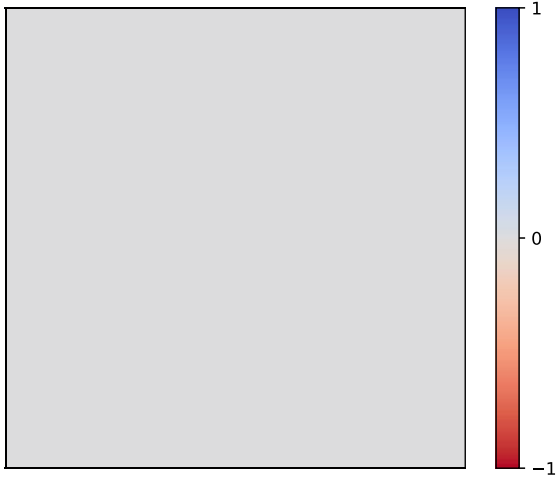
Example – Recognizing Digits



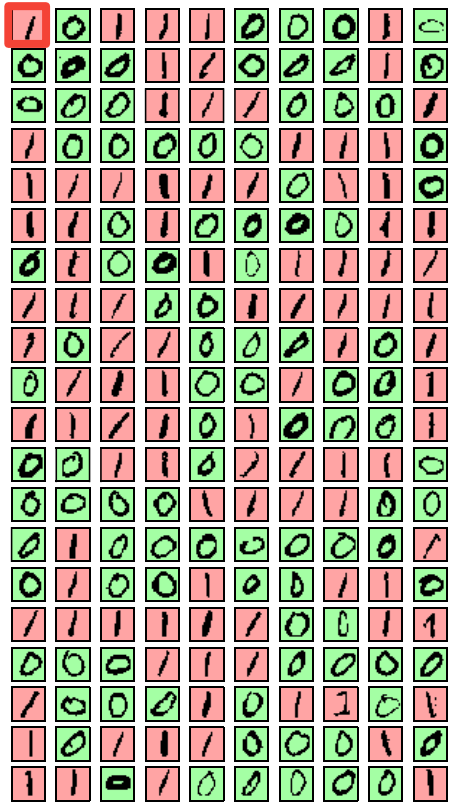
Training step 0

50% Accuracy

Weights as a 28x28 picture



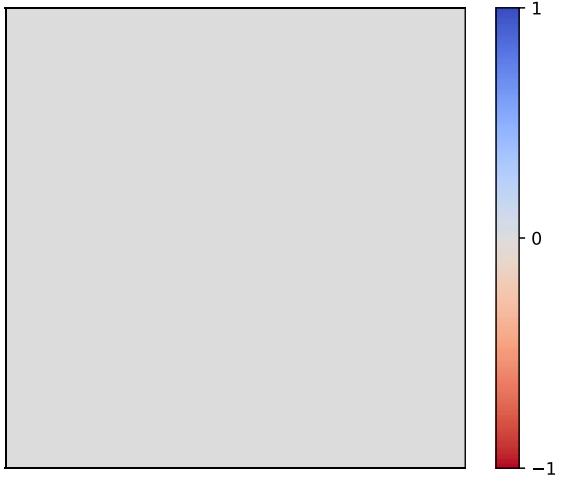
Example – Recognizing Digits



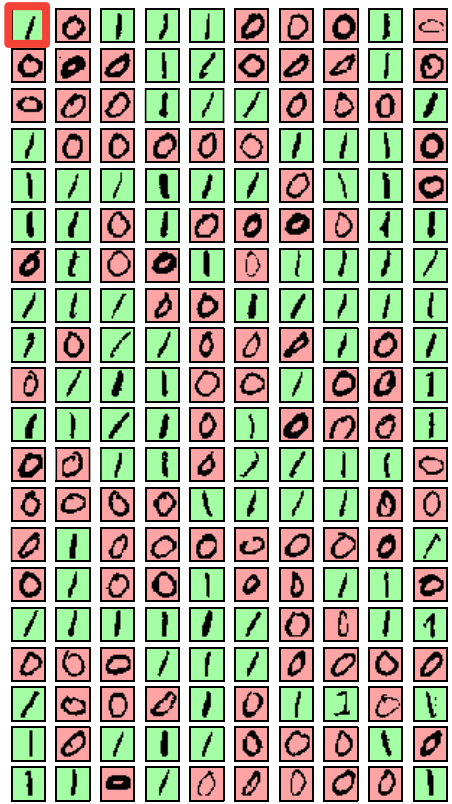
Training step 1

50% Accuracy

Weights as a 28x28 picture



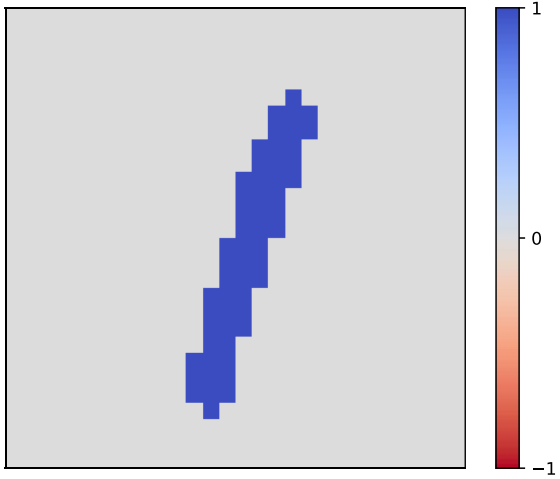
Example – Recognizing Digits



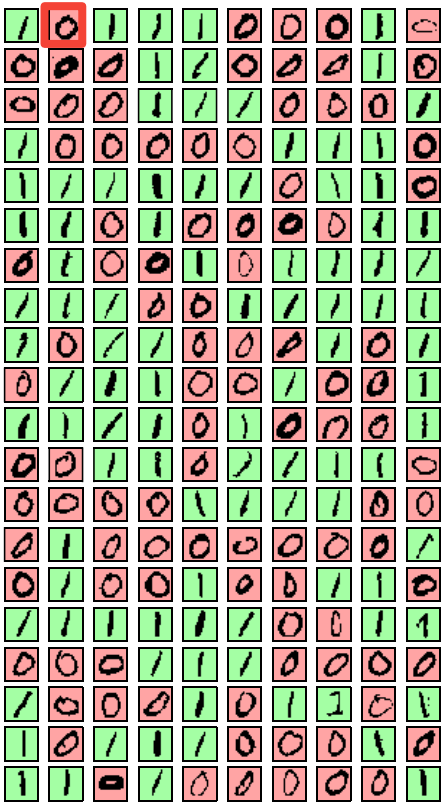
Training step 1

50% Accuracy

Weights as a 28x28 picture



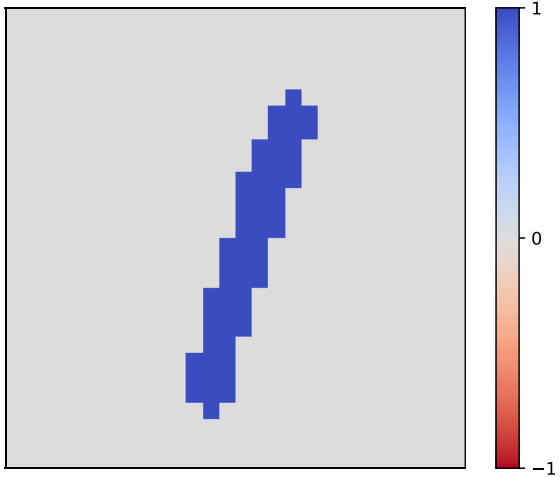
Example – Recognizing Digits



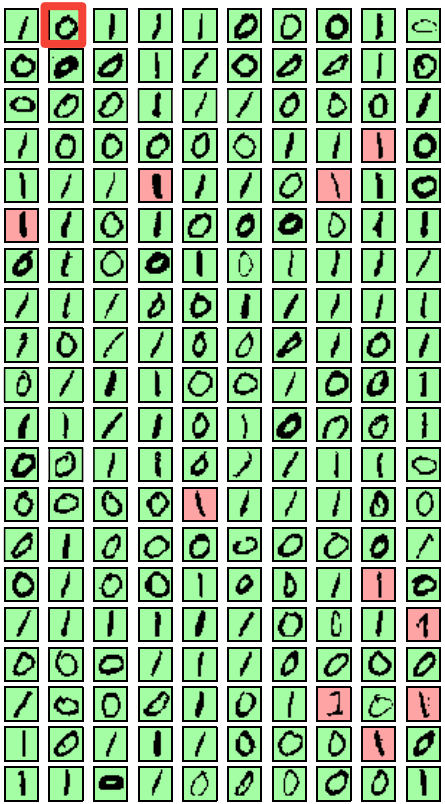
Training step 2

50% Accuracy

Weights as a 28x28 picture



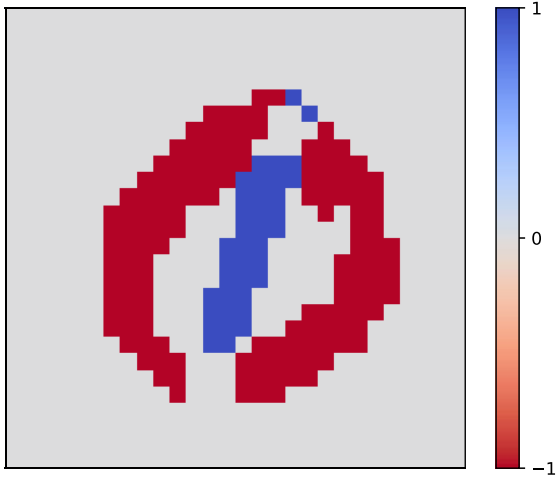
Example – Recognizing Digits



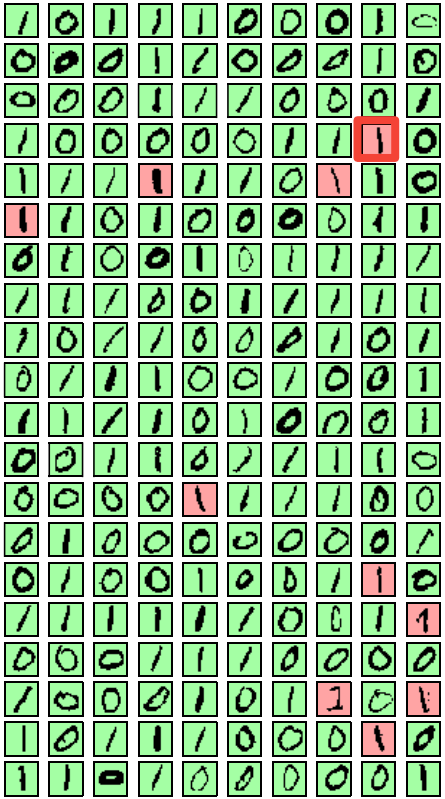
Training step 2

95% Accuracy

Weights as a 28x28 picture



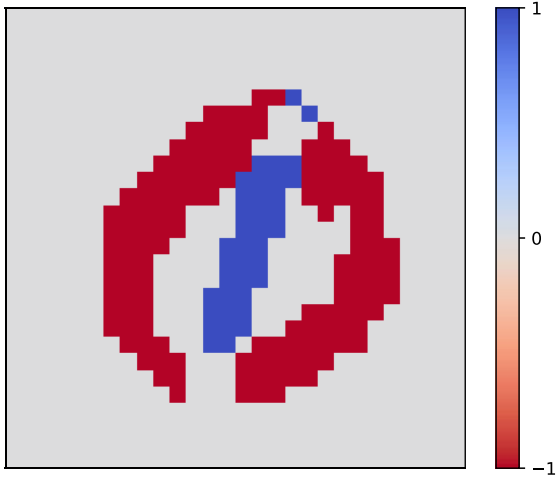
Example – Recognizing Digits



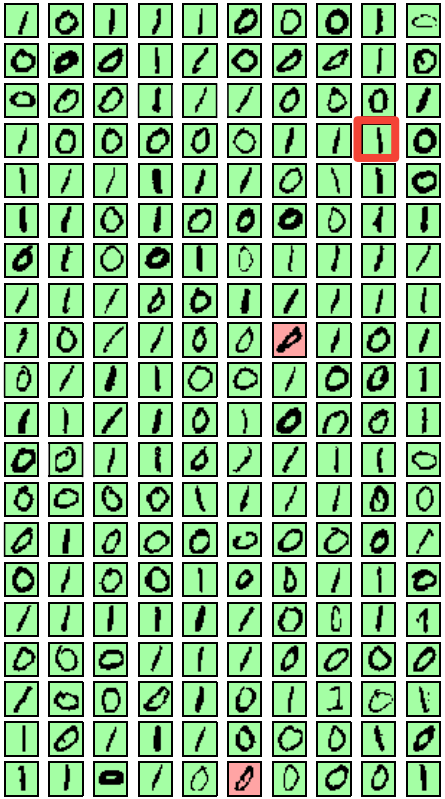
Training step 39

95% Accuracy

Weights as a 28x28 picture



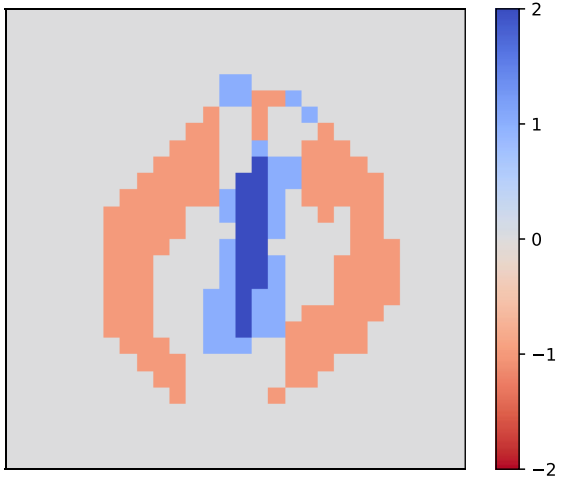
Example – Recognizing Digits



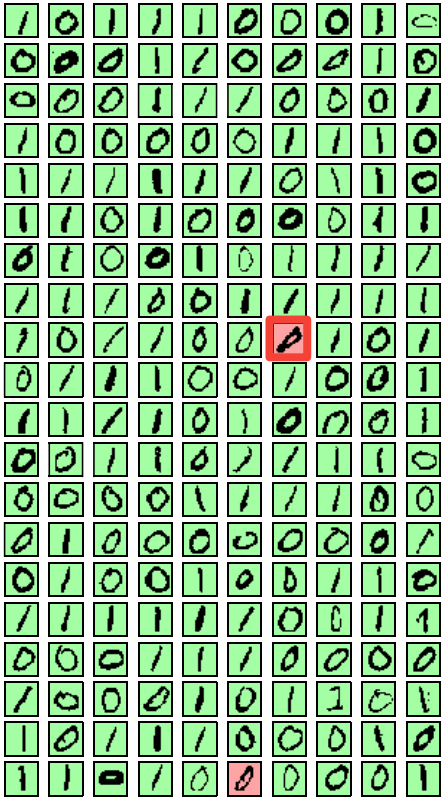
Training step 39

99% Accuracy

Weights as a 28x28 picture



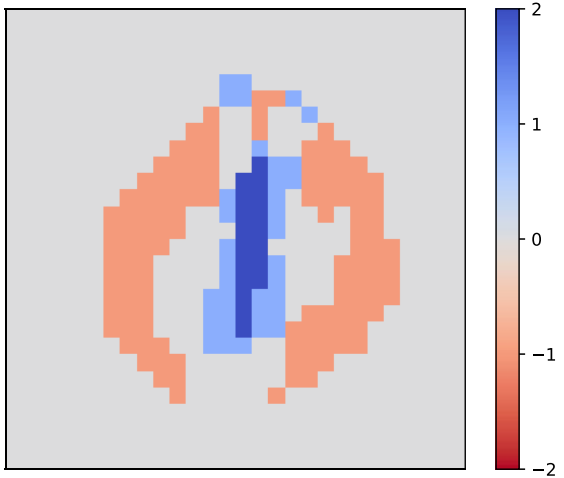
Example – Recognizing Digits



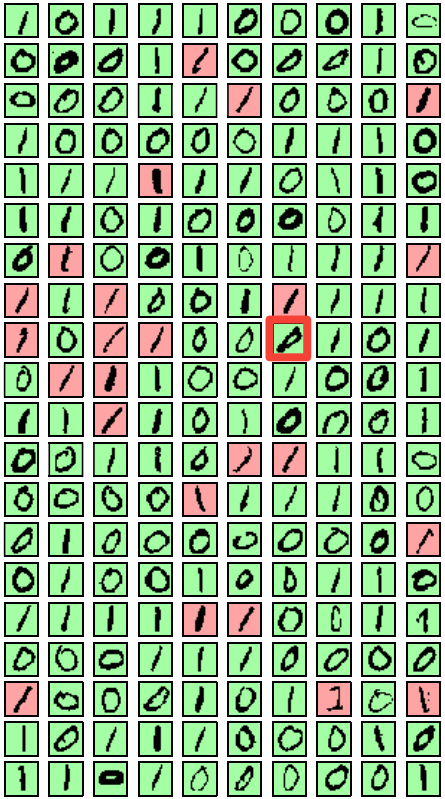
Training step 87

99% Accuracy

Weights as a 28x28 picture



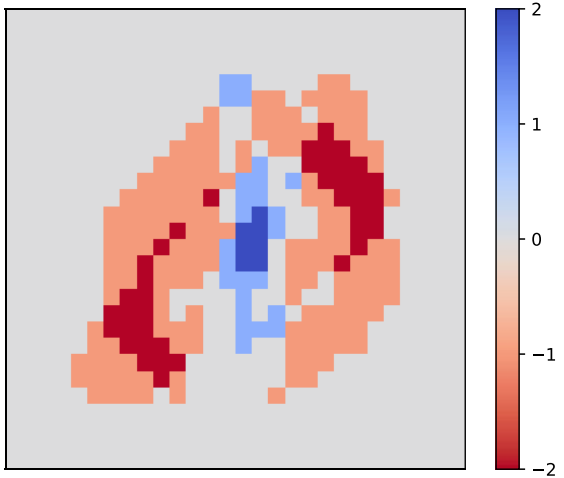
Example – Recognizing Digits



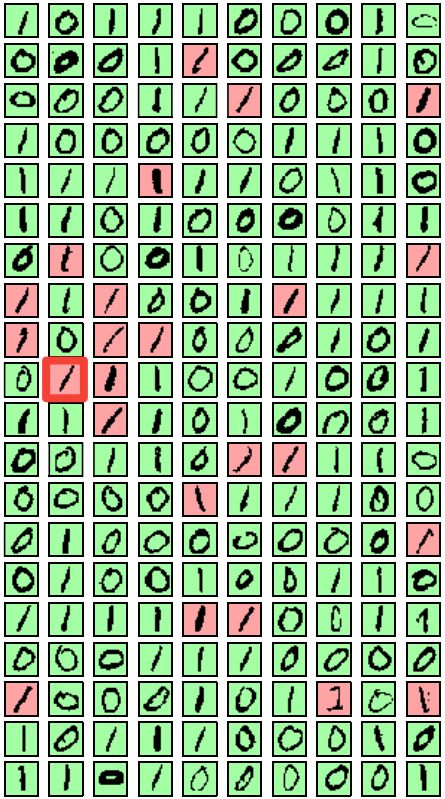
Training step 87

88% Accuracy

Weights as a 28x28 picture

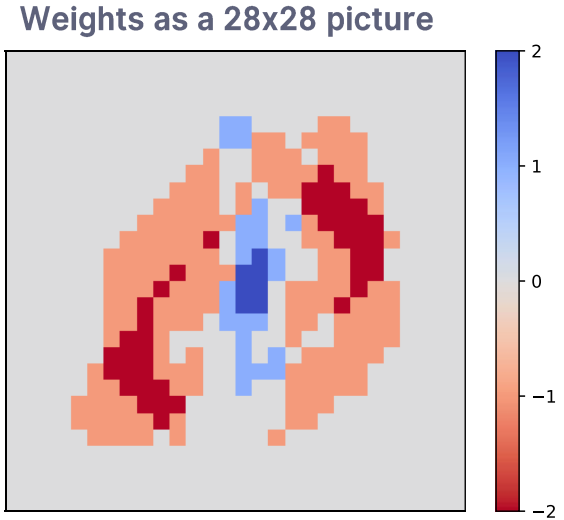


Example – Recognizing Digits

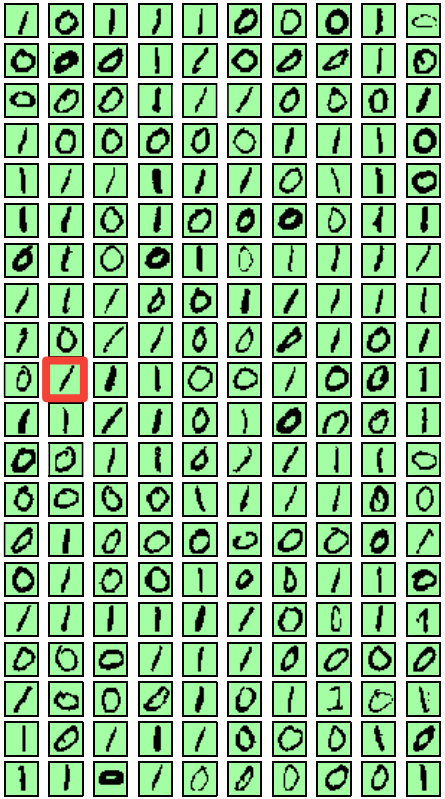


Training step 92

88% Accuracy



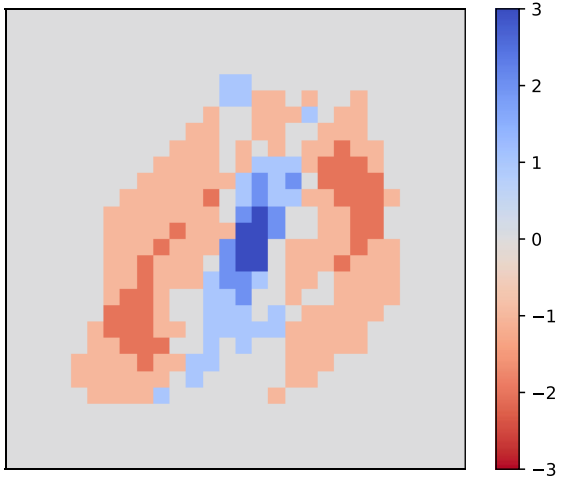
Example – Recognizing Digits



Training step 92

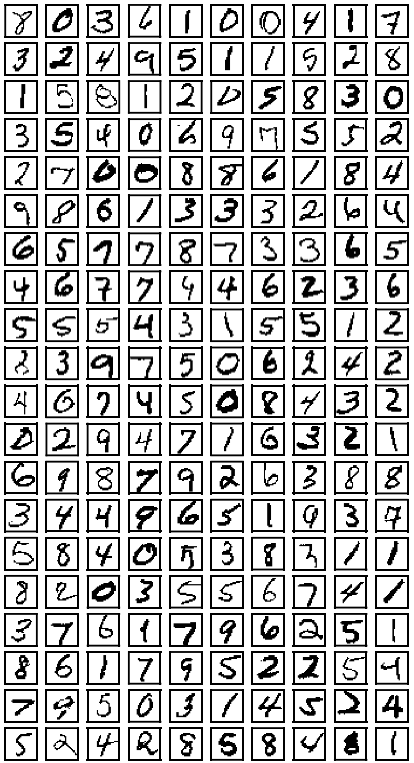
100% Accuracy

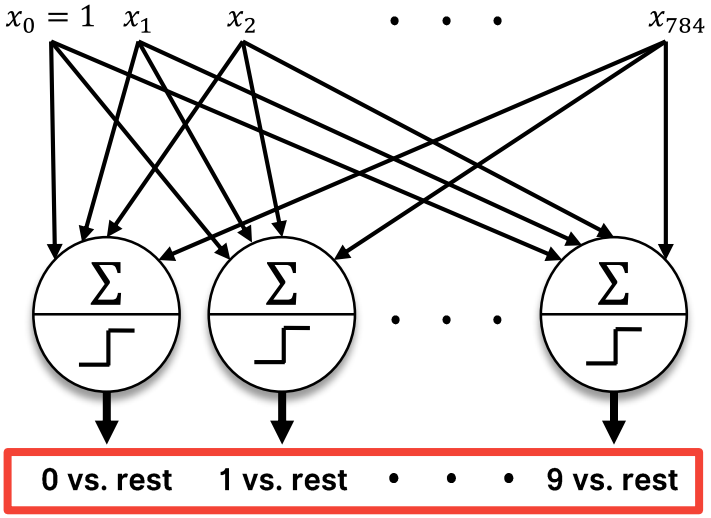
Weights as a 28x28 picture



Example – Recognizing All Digits

100 samples of each digit



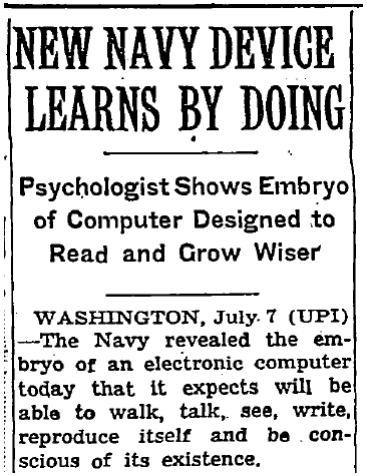


One-hot encoding of predicted digit

- The perceptron can predict multiple classes by using a separate neuron for each class.
- It easily achives ~99% accuracy.
- Pretty impressive, right? Rosenblatt certainly thought so… ☺

Overconfidence in the Perceptron

Excerpts from **The New York Times**, July 8, 1958



NEW NAVY DEVICE LEARNS BY DOING

Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser

WASHINGTON, July 7 (UPI) —The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

The service said it would use this principle to build the first of its Perceptron thinking machines that will be able to read and write. It is expected to be finished in about a year at a cost of $100,000.

Dr. Frank Rosenblatt, designer of the Perceptron, conducted the demonstration. He said the machine would be the first device to think as the human brain. As do human beings, Perceptron will make mistakes at first, but will grow wiser as it gains experience, he said.

The Navy said the perceptron would be the first non-living mechanism "capable of receiving, recognizing and identifying its surroundings without any human training or control."

The "brain" is designed to remember images and information it has perceived itself. Ordinary computers remember only what is fed into them on punch cards or magnetic tape.

Later Perceptrons will be able to recognize people and call out their names and instantly translate speech in one language to speech or writing in another language, it was predicted.

- The article was published after statements made by Frank Rosenblatt in press conference held by the US Navy.
- http://jcblackmon.com/wp-content/uploads/2018/01/MBC-Rosenblatt-Perceptron-NYT-article.jpg.pdf

Recap

- **MuCulloch-Pitts model** – In 1943, *MuCulloch* and *Pitts* proposed a model for an *artificial neuron* which mimicked the behavior of the *biological neuron*.
  - The biological neuron has dendrites, a soma (body) and an axon.
  - The neuron “fires” (sends a pulse through its axon) if enough electrical potential has accumulated in the soma though its dendrites.
- **Hebb’s rule** – In 1949, *Donald Hebb* published the idea that connections between neurons have an associated *strength* and that strength changes based on how often one neuron is involved in firing the other.
- **The Perceptron** – In 1957, *Frank Rosenblatt* proposed an algorithm (and a machine) which was based on the *MuCulloch-Pitts model* and *Hebb’s rule* which could learn to recognize images.

Perceptron Limitations

Perceptron Limitations

- It turns out that the *Perceptron* had no trouble learning AND/OR and recognizing digits because they are not very hard problems.

Perceptron Limitations

- It turns out that the *Perceptron* had no trouble learning AND/OR and recognizing digits because they are not very hard problems.
- Let's consider the **XOR** (exclusive OR) function:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Perceptron Limitations

- It turns out that the *Perceptron* had no trouble learning AND/OR and recognizing digits because they are not very hard problems.
- Let's consider the **XOR** (exclusive OR) function:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

For a Perceptron to learn XOR, it needs:

$$w_0 + 0 \cdot w_1 + 0 \cdot w_2 < 0$$
$$w_0 + 0 \cdot w_1 + 1 \cdot w_2 \geq 0$$
$$w_0 + 1 \cdot w_1 + 0 \cdot w_2 \geq 0$$
$$w_0 + 1 \cdot w_1 + 1 \cdot w_2 < 0$$

Perceptron Limitations

- It turns out that the *Perceptron* had no trouble learning AND/OR and recognizing digits because they are not very hard problems.
- Let's consider the **XOR** (exclusive OR) function:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

For a Perceptron to learn XOR, it needs:

$$w_0 + 0 \cdot w_1 + 0 \cdot w_2 < 0 \Rightarrow w_0 < 0$$
$$w_0 + 0 \cdot w_1 + 1 \cdot w_2 \geq 0 \Rightarrow w_0 \geq -w_2$$
$$w_0 + 1 \cdot w_1 + 0 \cdot w_2 \geq 0 \Rightarrow w_0 \geq -w_1$$
$$w_0 + 1 \cdot w_1 + 1 \cdot w_2 < 0 \Rightarrow w_0 < -w_1 - w_2$$

Perceptron Limitations

- It turns out that the *Perceptron* had no trouble learning AND/OR and recognizing digits because they are not very hard problems.
- Let's consider the **XOR** (exclusive OR) function:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

For a Perceptron to learn XOR, it needs:

$$w_0 + 0 \cdot w_1 + 0 \cdot w_2 < 0 \Rightarrow w_0 < 0$$
$$\left.\begin{aligned} w_0 + 0 \cdot w_1 + 1 \cdot w_2 \geq 0 \Rightarrow w_0 \geq -w_2 \\ w_0 + 1 \cdot w_1 + 0 \cdot w_2 \geq 0 \Rightarrow w_0 \geq -w_1 \end{aligned}\right\} 2w_0 \geq -w_1 - w_2$$
$$w_0 + 1 \cdot w_1 + 1 \cdot w_2 < 0 \Rightarrow w_0 < -w_1 - w_2$$

Perceptron Limitations

- It turns out that the *Perceptron* had no trouble learning AND/OR and recognizing digits because they are not very hard problems.
- Let's consider the **XOR** (exclusive OR) function:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

For a Perceptron to learn XOR, it needs:

$$w_0 + 0 \cdot w_1 + 0 \cdot w_2 < 0 \Rightarrow w_0 < 0$$
$$\left.\begin{array}{l} w_0 + 0 \cdot w_1 + 1 \cdot w_2 \geq 0 \Rightarrow w_0 \geq -w_2 \\ w_0 + 1 \cdot w_1 + 0 \cdot w_2 \geq 0 \Rightarrow w_0 \geq -w_1 \end{array}\right\} 2w_0 \geq -w_1 - w_2$$
$$\left.\begin{array}{l} \\ \\ w_0 + 1 \cdot w_1 + 1 \cdot w_2 < 0 \Rightarrow w_0 < -w_1 - w_2 \end{array}\right\} 2w_0 > w_0$$

Perceptron Limitations

- It turns out that the *Perceptron* had no trouble learning AND/OR and recognizing digits because they are not very hard problems.
- Let's consider the **XOR** (exclusive OR) function:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

For a Perceptron to learn XOR, it needs:

$$w_0 + 0 \cdot w_1 + 0 \cdot w_2 < 0 \Rightarrow \boldsymbol{w_0 < 0}$$

$$\left.\begin{array}{l} w_0 + 0 \cdot w_1 + 1 \cdot w_2 \geq 0 \Rightarrow w_0 \geq -w_2 \\ w_0 + 1 \cdot w_1 + 0 \cdot w_2 \geq 0 \Rightarrow w_0 \geq -w_1 \end{array}\right\} 2w_0 \geq -w_1 - w_2 \left.\begin{array}{l} \\ \\ \end{array}\right\} 2w_0 > w_0 \Rightarrow \boldsymbol{w_0 > 0}$$

$$w_0 + 1 \cdot w_1 + 1 \cdot w_2 < 0 \Rightarrow w_0 < -w_1 - w_2$$

Contradiction

$\Rightarrow$ The Perceptron cannot possibly learn this function, no matter how many training steps it takes.

Perceptron Limitations

- It turns out that the *Perceptron* had no trouble learning AND/OR and recognizing digits because they are not very hard problems.
- Let's consider the **XOR** (exclusive OR) function:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

For a Perceptron to learn XOR, it needs:

$$w_0 + 0 \cdot w_1 + 0 \cdot w_2 < 0 \Rightarrow \boldsymbol{w_0 < 0}$$

$$\left.\begin{array}{l} w_0 + 0 \cdot w_1 + 1 \cdot w_2 \geq 0 \Rightarrow w_0 \geq -w_2 \\ w_0 + 1 \cdot w_1 + 0 \cdot w_2 \geq 0 \Rightarrow w_0 \geq -w_1 \end{array}\right\} 2w_0 \geq -w_1 - w_2$$

$$w_0 + 1 \cdot w_1 + 1 \cdot w_2 < 0 \Rightarrow w_0 < -w_1 - w_2$$

Contradiction

$$2w_0 > w_0 \Rightarrow \mathbf{w_0 > 0}$$

$\Rightarrow$ The Perceptron cannot possibly learn this function, no matter how many training steps it takes.

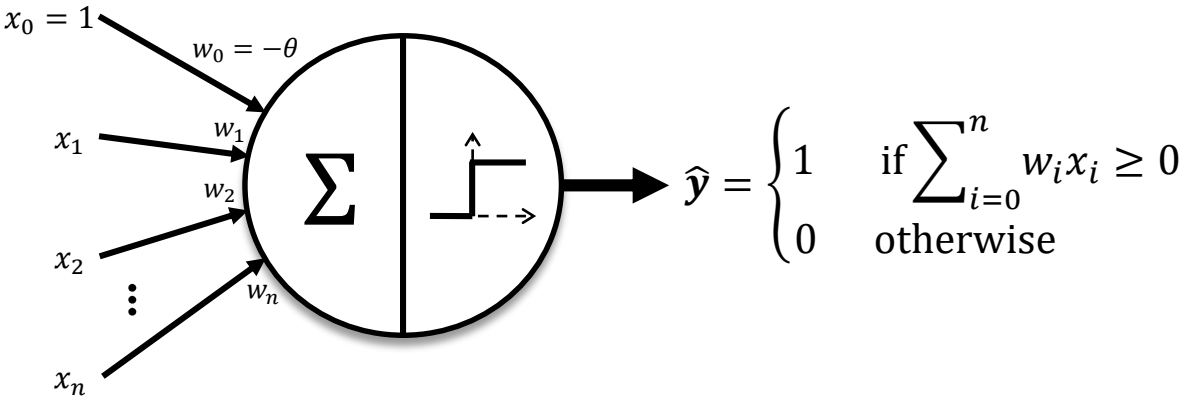
- In fact, the *Perceptron* can only learn the class of problems known as **linearly separable**.

AI Winter

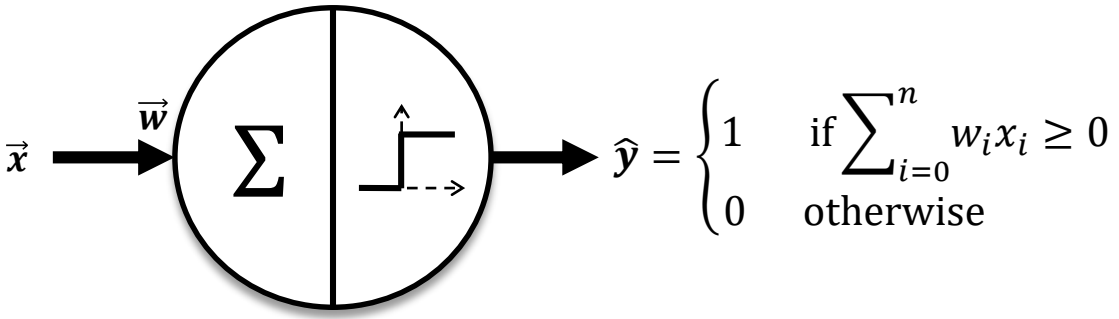
- Early 50s to mid 60s were a period of very strong optimism in the field of AI.
  - Distinguished figures, such as **Allan Turing, Claude Shannon, Frank Rosenblatt**, talked about human-level AI within a few years.
- The promised breakthroughs did not come as quickly as promised, so mid to late 60s saw an abrupt decline in interest and funding for AI research.
  - **Marvin Minsky** and **Seymour Papert**'s 1969 book **"Perceptrons: an introduction to computational geometry"**, which discussed the limitations of perceptrons in detail, is regarded as the main cause for the lack of funding in the field (especially for neural networks) for over a decade.
  - Mathematician **Sir James Lighthill** also criticized the utter failure of AI to achieve its "grandiose objectives.", mentioning that many of AI's supposedly successful algorithms were only suitable for „toy" problems.
- The period from late 60s to late 80s is known as the (first) **AI Winter**.

Improving the Perceptron

Simplifying Notation

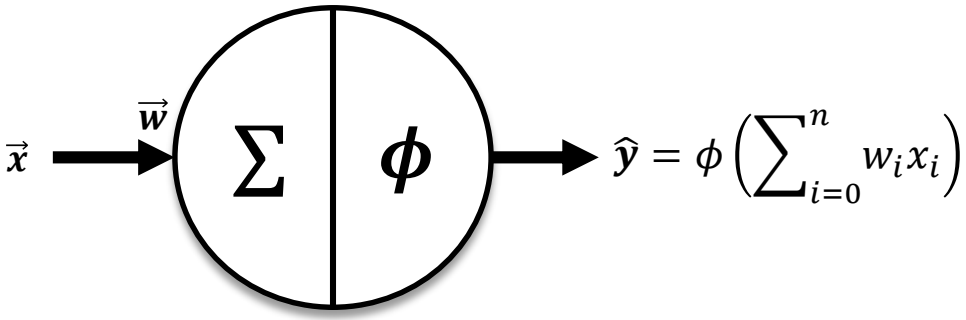


Simplifying Notation



Inputs and weights in vector format.

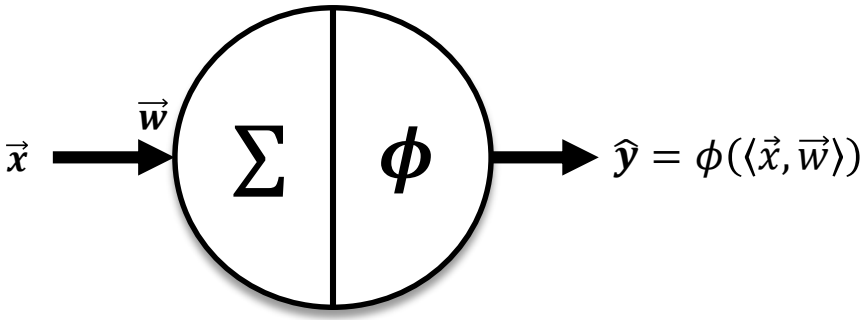
Simplifying Notation



Inputs and weights in vector format.

Denote the *unit step function* by ϕ (and call it an "**activation function**")

Simplifying Notation



Inputs and weights in vector format.

Denote the *unit step function* by ϕ (and call it an "**activation function**")

Weighted sum of inputs as *dot product.*

Better Justification for the Learning Rule

- Perceptron output:

$$\hat{y} = \phi(\langle \vec{x}, \vec{w} \rangle)$$

- *Squared-error* of the perceptron (for one example):

$$E(\vec{x}) = \frac{1}{2}(y - \hat{y})^2$$

Better Justification for the Learning Rule

- Perceptron output:

$$\hat{y} = \phi(\langle \vec{x}, \vec{w} \rangle)$$

- *Squared-error* of the perceptron (for one example):

$$E(\vec{x}) = \frac{1}{2}(y - \hat{y})^2$$

- **Gradient descent** (*Cauchy*, 1847) is a method of finding the minimum of a function by taking steps in the direction of the negative gradient.

$$\frac{\partial E(\vec{x})}{\partial w_j} =$$

We want to find the minimum of the error w.r.t. weights.

Better Justification for the Learning Rule

- Perceptron output:

$$\hat{y} = \phi(\langle \vec{x}, \vec{w} \rangle)$$

- *Squared-error* of the perceptron (for one example):

$$E(\vec{x}) = \frac{1}{2}(y - \hat{y})^2$$

- **Gradient descent** (*Cauchy*, 1847) is a method of finding the minimum of a function by taking steps in the direction of the negative gradient.

$$\frac{\partial E(\vec{x})}{\partial w_j} = \frac{\partial E(\vec{x})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_j}$$

E is function of $\hat{y}$ and $\hat{y}$ is a function of $w_j \Rightarrow$ we apply **chain rule**.

Better Justification for the Learning Rule

- Perceptron output:

$$\hat{y} = \phi(\langle \vec{x}, \vec{w} \rangle)$$

- *Squared-error* of the perceptron (for one example):

$$E(\vec{x}) = \frac{1}{2}(y - \hat{y})^2$$

- **Gradient descent** (*Cauchy*, 1847) is a method of finding the minimum of a function by taking steps in the direction of the negative gradient.

$$\frac{\partial E(\vec{x})}{\partial w_j} = \frac{\partial E(\vec{x})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_j} = \frac{\partial E(\vec{x})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \langle \vec{x}, \vec{w} \rangle} \cdot \frac{\partial \langle \vec{x}, \vec{w} \rangle}{\partial w_j}$$

Chain rule again.

Better Justification for the Learning Rule

- Perceptron output:

$$\hat{y} = \phi(\langle \vec{x}, \vec{w} \rangle)$$

- *Squared-error* of the perceptron (for one example):

$$E(\vec{x}) = \frac{1}{2}(y - \hat{y})^2$$

- **Gradient descent** (*Cauchy*, 1847) is a method of finding the minimum of a function by taking steps in the direction of the negative gradient.

$$\frac{\partial E(\vec{x})}{\partial w_j} = \frac{\partial E(\vec{x})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_j} = \frac{\partial E(\vec{x})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \langle \vec{x}, \vec{w} \rangle} \cdot \frac{\partial \langle \vec{x}, \vec{w} \rangle}{\partial w_j}$$

$$\frac{\partial E(\vec{x})}{\partial \hat{y}} = \qquad \frac{\partial \hat{y}}{\partial \langle \vec{x}, \vec{w} \rangle} = \qquad \frac{\partial \langle \vec{x}, \vec{w} \rangle}{\partial w_j} =$$

Better Justification for the Learning Rule

- Perceptron output:

$$\hat{y} = \phi(\langle \vec{x}, \vec{w} \rangle)$$

- *Squared-error* of the perceptron (for one example):

$$E(\vec{x}) = \frac{1}{2}(y - \hat{y})^2$$

- **Gradient descent** (*Cauchy*, 1847) is a method of finding the minimum of a function by taking steps in the direction of the negative gradient.

$$\frac{\partial E(\vec{x})}{\partial w_j} = \frac{\partial E(\vec{x})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_j} = \frac{\partial E(\vec{x})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \langle \vec{x}, \vec{w} \rangle} \cdot \frac{\partial \langle \vec{x}, \vec{w} \rangle}{\partial w_j}$$

$$\frac{\partial E(\vec{x})}{\partial \hat{y}} = -(y - \hat{y})$$

$$\frac{\partial \hat{y}}{\partial \langle \vec{x}, \vec{w} \rangle} =$$

$$\frac{\partial \langle \vec{x}, \vec{w} \rangle}{\partial w_j} =$$

Better Justification for the Learning Rule

- Perceptron output:

$$\hat{y} = \phi(\langle \vec{x}, \vec{w} \rangle)$$

- *Squared-error* of the perceptron (for one example):

$$E(\vec{x}) = \frac{1}{2}(y - \hat{y})^2$$

- **Gradient descent** (*Cauchy*, 1847) is a method of finding the minimum of a function by taking steps in the direction of the negative gradient.

$$\frac{\partial E(\vec{x})}{\partial w_j} = \frac{\partial E(\vec{x})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_j} = \frac{\partial E(\vec{x})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \langle \vec{x}, \vec{w} \rangle} \cdot \frac{\partial \langle \vec{x}, \vec{w} \rangle}{\partial w_j}$$

$$\frac{\partial E(\vec{x})}{\partial \hat{y}} = -(y - \hat{y}) \qquad \frac{\partial \hat{y}}{\partial \langle \vec{x}, \vec{w} \rangle} = \phi'(\langle \vec{x}, \vec{w} \rangle) \qquad \frac{\partial \langle \vec{x}, \vec{w} \rangle}{\partial w_j} =$$

Better Justification for the Learning Rule

- Perceptron output:

$$\hat{y} = \phi(\langle \vec{x}, \vec{w} \rangle)$$

- *Squared-error* of the perceptron (for one example):

$$E(\vec{x}) = \frac{1}{2}(y - \hat{y})^2$$

- **Gradient descent** (*Cauchy*, 1847) is a method of finding the minimum of a function by taking steps in the direction of the negative gradient.

$$\frac{\partial E(\vec{x})}{\partial w_j} = \frac{\partial E(\vec{x})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_j} = \frac{\partial E(\vec{x})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \langle \vec{x}, \vec{w} \rangle} \cdot \frac{\partial \langle \vec{x}, \vec{w} \rangle}{\partial w_j}$$

$$\frac{\partial E(\vec{x})}{\partial \hat{y}} = -(y - \hat{y}) \qquad \frac{\partial \hat{y}}{\partial \langle \vec{x}, \vec{w} \rangle} = \phi'(\langle \vec{x}, \vec{w} \rangle) \qquad \frac{\partial \langle \vec{x}, \vec{w} \rangle}{\partial w_j} = \frac{\partial \left(\sum_{j=0}^{n} w_j x_j \right)}{\partial w_j} = x_j$$

Better Justification for the Learning Rule

- Perceptron output:

$$\hat{y} = \phi(\langle \vec{x}, \vec{w} \rangle)$$

- *Squared-error* of the perceptron (for one example):

$$E(\vec{x}) = \frac{1}{2}(y - \hat{y})^2$$

- **Gradient descent** (*Cauchy*, 1847) is a method of finding the minimum of a function by taking steps in the direction of the negative gradient.

$$\frac{\partial E(\vec{x})}{\partial w_j} = \frac{\partial E(\vec{x})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_j} = \frac{\partial E(\vec{x})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \langle \vec{x}, \vec{w} \rangle} \cdot \frac{\partial \langle \vec{x}, \vec{w} \rangle}{\partial w_j} = -(y - \hat{y}) \cdot \phi'(\langle \vec{x}, \vec{w} \rangle) \cdot x_j$$

Better Justification for the Learning Rule

- Perceptron output:

$$\hat{y} = \phi(\langle \vec{x}, \vec{w} \rangle)$$

- *Squared-error* of the perceptron (for one example):

$$E(\vec{x}) = \frac{1}{2}(y - \hat{y})^2$$

- **Gradient descent** (*Cauchy*, 1847) is a method of finding the minimum of a function by taking steps in the direction of the negative gradient.

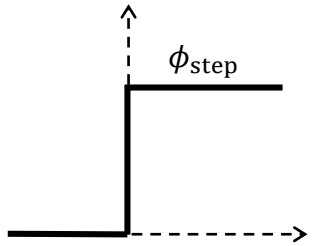
$$\frac{\partial E(\vec{x})}{\partial w_j} = \frac{\partial E(\vec{x})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_j} = \frac{\partial E(\vec{x})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \langle \vec{x}, \vec{w} \rangle} \cdot \frac{\partial \langle \vec{x}, \vec{w} \rangle}{\partial w_j} = -(y - \hat{y}) \cdot \phi'(\langle \vec{x}, \vec{w} \rangle) \cdot x_j$$

$$\Delta w_j = \eta \cdot (y - \hat{y}) \cdot \phi'(\langle \vec{x}, \vec{w} \rangle) \cdot x_j$$

Delta rule

Applying the Delta Rule to the Perceptron

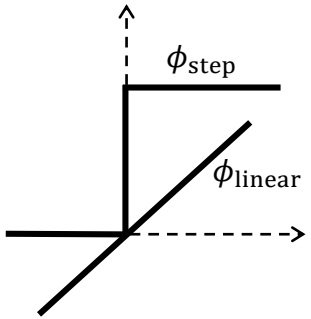
- Delta rule: $\Delta w_j = \eta \cdot (y - \hat{y}) \cdot \phi'(\langle \vec{x}, \vec{w} \rangle) \cdot x_j$



The perceptron uses a *step activation function* which is not *differentiable*.

Applying the Delta Rule to the Perceptron

- Delta rule: $\Delta w_j = \eta \cdot (y - \hat{y}) \cdot \phi'(\langle \vec{x}, \vec{w} \rangle) \cdot x_j$

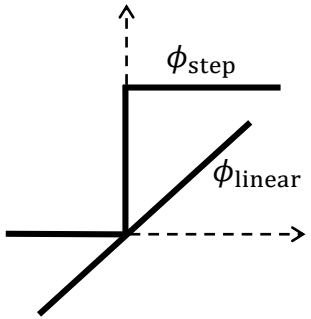


The perceptron uses a *step activation function* which is not *differentiable*.

- The **ADALINE** (**Adaptive Linear Neuron**) is a variant of the Perceptron, proposed by Bernard Widrow in 1960, which uses a *linear activation* function while training, and a *step activation* function afterwards.
  - Linear activation means the derivative is constant: $\phi(x) = x \Rightarrow \phi'(x) = 1$

Applying the Delta Rule to the Perceptron

- Delta rule: $\Delta w_j = \eta \cdot (y - \hat{y}) \cdot \phi'(\langle \vec{x}, \vec{w} \rangle) \cdot x_j$



The perceptron uses a *step activation function* which is not *differentiable*.

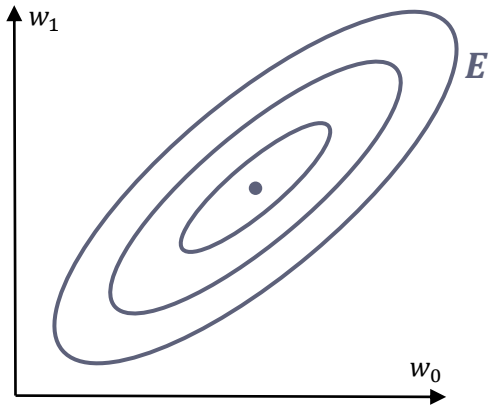
- The **ADALINE** (**Adaptive Linear Neuron**) is a variant of the Perceptron, proposed by Bernard Widrow in 1960, which uses a *linear activation* function while training, and a *step activation* function afterwards.
  - Linear activation means the derivative is constant: $\phi(x) = x \Rightarrow \phi'(x) = 1$

- Adaline delta rule: $\Delta w_j = \eta \cdot (y - \hat{y}) \cdot x_j$

Very similar to the Perceptron update rule, but $\hat{y}$ is now real and unbounded.

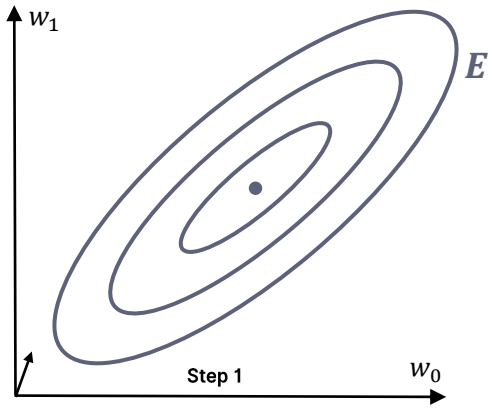
Batch Update

- We are interested in finding weights to minimize the error over *all training samples*.
- In the original perceptron algorithm, each misclassified training sample caused an update in the weights.
  - Changing the weights in the direction indicated by a single example might not be optimal in terms of the whole training set.



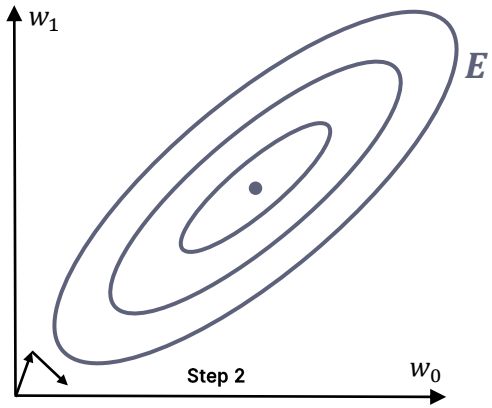
Batch Update

- We are interested in finding weights to minimize the error over *all training samples*.
- In the original perceptron algorithm, each misclassified training sample caused an update in the weights.
  - Changing the weights in the direction indicated by a single example might not be optimal in terms of the whole training set.



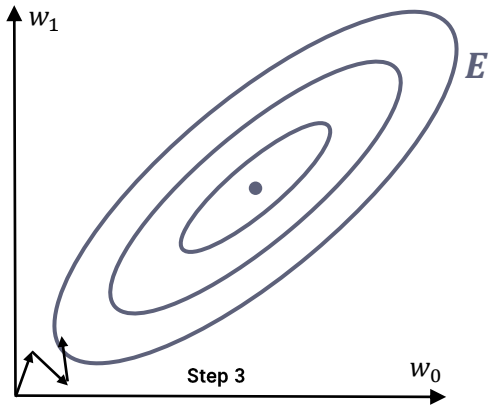
Batch Update

- We are interested in finding weights to minimize the error over *all training samples*.
- In the original perceptron algorithm, each misclassified training sample caused an update in the weights.
  - Changing the weights in the direction indicated by a single example might not be optimal in terms of the whole training set.



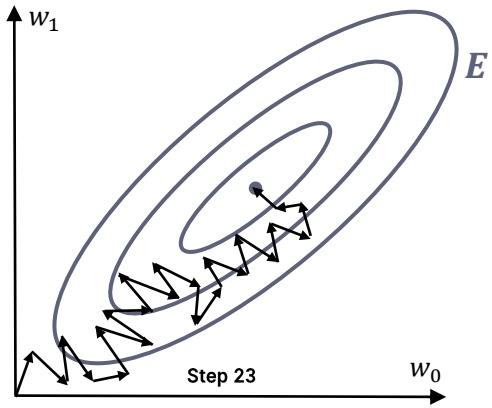
Batch Update

- We are interested in finding weights to minimize the error over *all training samples*.
- In the original perceptron algorithm, each misclassified training sample caused an update in the weights.
  - Changing the weights in the direction indicated by a single example might not be optimal in terms of the whole training set.



Batch Update

- We are interested in finding weights to minimize the error over *all training samples*.
- In the original perceptron algorithm, each misclassified training sample caused an update in the weights.
  - Changing the weights in the direction indicated by a single example might not be optimal in terms of the whole training set.

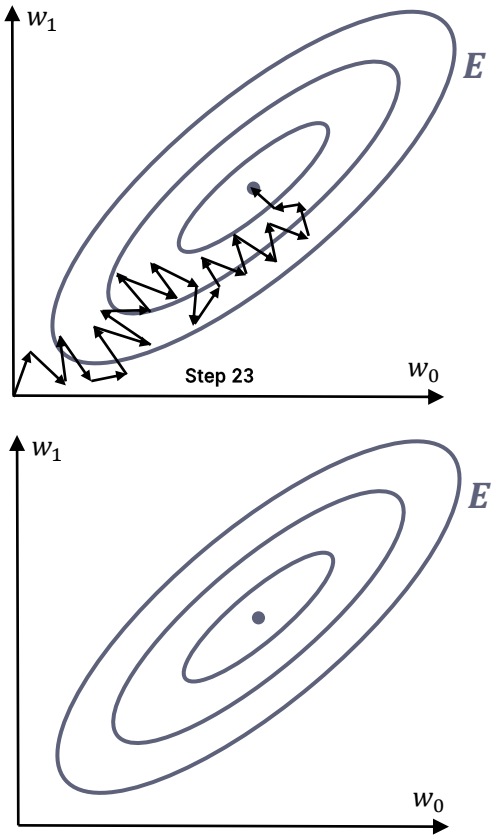


Batch Update

- We are interested in finding weights to minimize the error over *all training samples*.
- In the original perceptron algorithm, each misclassified training sample caused an update in the weights.
  - Changing the weights in the direction indicated by a single example might not be optimal in terms of the whole training set.
- **Batch update** means updating the weights only once with an average gradient over multiple training samples.

$$\Delta w_j = \frac{\eta}{\mathbf{b}} \sum_{i_b=0}^{b-1} \left(y^{(i+i_b)} - \hat{y}^{(i+i_b)} \right) \cdot x_j^{(i+i_b)}$$

b – Batch Size
i_b - Batch Iterator

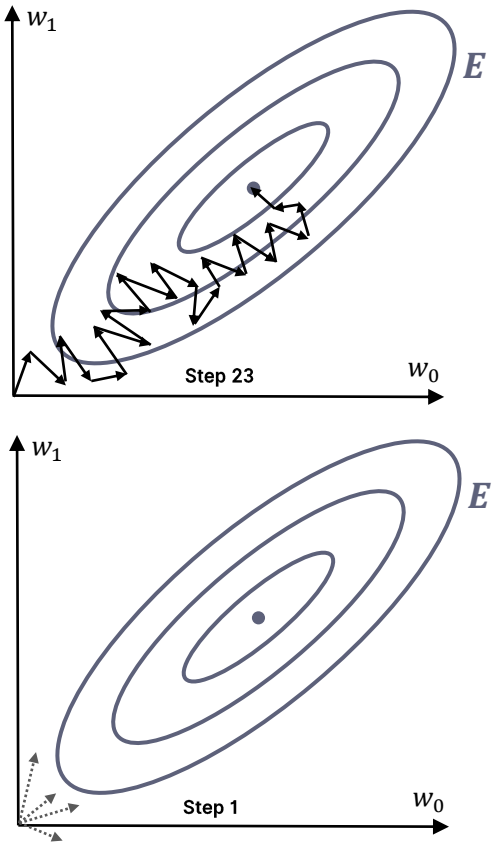


Batch Update

- We are interested in finding weights to minimize the error over *all training samples*.
- In the original perceptron algorithm, each misclassified training sample caused an update in the weights.
  - Changing the weights in the direction indicated by a single example might not be optimal in terms of the whole training set.
- **Batch update** means updating the weights only once with an average gradient over multiple training samples.

$$\Delta w_j = \frac{\eta}{\mathrm{b}} \sum_{i_b=0}^{b-1} \left(y^{(i+i_b)} - \hat{y}^{(i+i_b)} \right) \cdot x_j^{(i+i_b)}$$

b – Batch Size
i_b - Batch Iterator

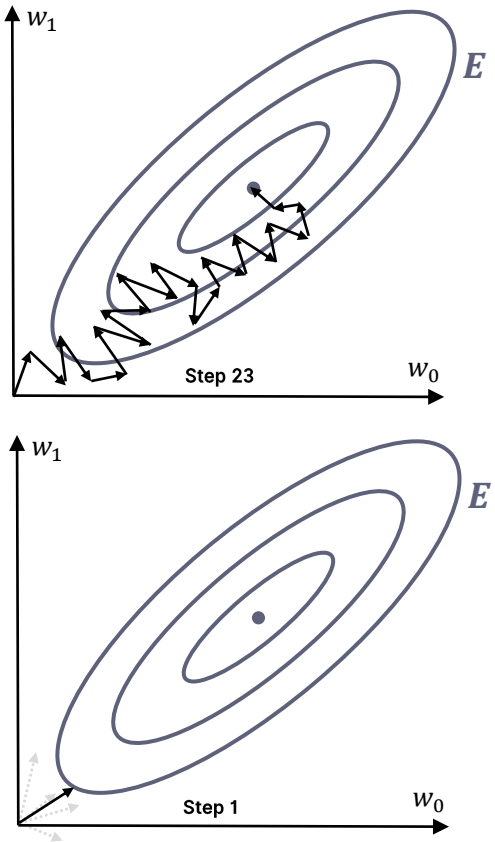


Batch Update

- We are interested in finding weights to minimize the error over *all training samples*.
- In the original perceptron algorithm, each misclassified training sample caused an update in the weights.
  - Changing the weights in the direction indicated by a single example might not be optimal in terms of the whole training set.
- **Batch update** means updating the weights only once with an average gradient over multiple training samples.

$$\Delta w_j = \frac{\eta}{\mathbf{b}} \sum_{i_b=0}^{b-1} \left(y^{(i+i_b)} - \hat{y}^{(i+i_b)} \right) \cdot x_j^{(i+i_b)}$$

b – Batch Size
i_b - Batch Iterator

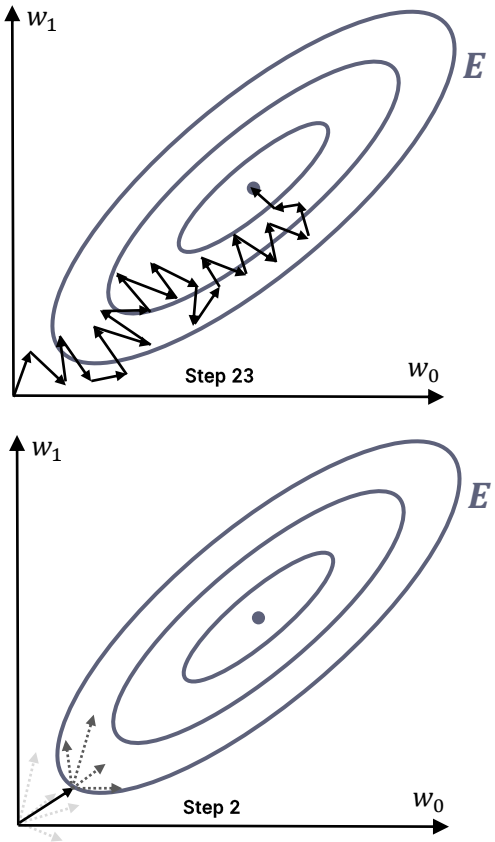


Batch Update

- We are interested in finding weights to minimize the error over *all training samples*.
- In the original perceptron algorithm, each misclassified training sample caused an update in the weights.
  - Changing the weights in the direction indicated by a single example might not be optimal in terms of the whole training set.
- **Batch update** means updating the weights only once with an average gradient over multiple training samples.

$$\Delta w_j = \frac{\eta}{\mathbf{b}} \sum_{i_b=0}^{b-1} \left(y^{(i+i_b)} - \hat{y}^{(i+i_b)} \right) \cdot x_j^{(i+i_b)}$$

b – Batch Size
i_b - Batch Iterator

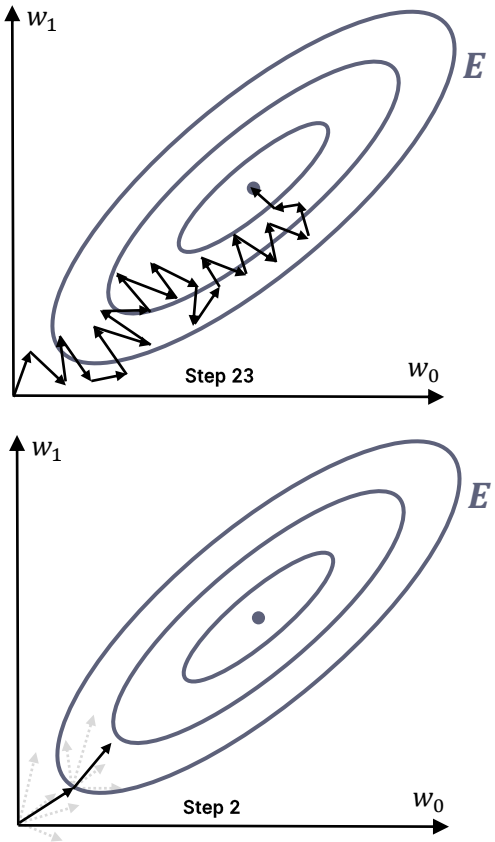


Batch Update

- We are interested in finding weights to minimize the error over *all training samples*.
- In the original perceptron algorithm, each misclassified training sample caused an update in the weights.
  - Changing the weights in the direction indicated by a single example might not be optimal in terms of the whole training set.
- **Batch update** means updating the weights only once with an average gradient over multiple training samples.

$$\Delta w_j = \frac{\eta}{\mathbf{b}} \sum_{i_b=0}^{b-1} \left(y^{(i+i_b)} - \hat{y}^{(i+i_b)} \right) \cdot x_j^{(i+i_b)}$$

b – Batch Size
i_b - Batch Iterator

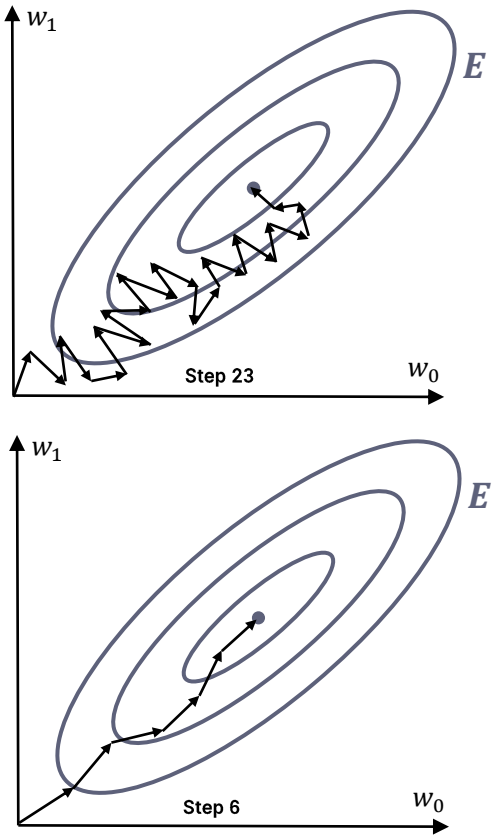


Batch Update

- We are interested in finding weights to minimize the error over *all training samples*.
- In the original perceptron algorithm, each misclassified training sample caused an update in the weights.
  - Changing the weights in the direction indicated by a single example might not be optimal in terms of the whole training set.
- **Batch update** means updating the weights only once with an average gradient over multiple training samples.

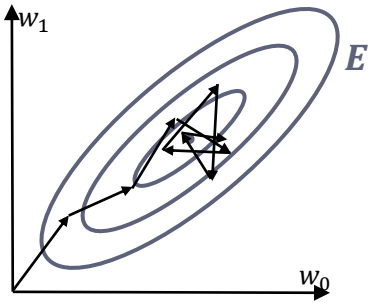
$$\Delta w_j = \frac{\eta}{\mathrm{b}} \sum_{i_b=0}^{b-1} \left(y^{(i+i_b)} - \hat{y}^{(i+i_b)} \right) \cdot x_j^{(i+i_b)}$$

b – Batch Size
i_b - Batch Iterator

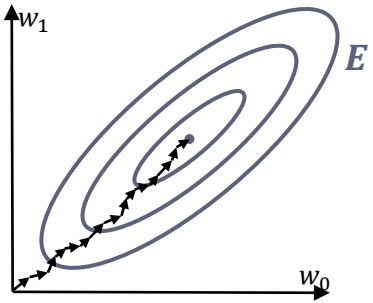


Influence of Learning Rate

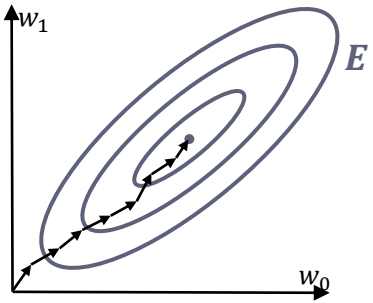
Learning rate too large
(It may fail to converge)



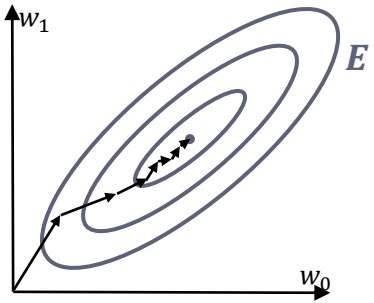
Learning rate too small
(It can take very long to converge)



Proper learning rate
(Compromise between speed and convergence)



Learning rate decay
(Start with a larger value and decrease it as training progresses)



Multilayer Perceptron

Building a **network** of **Perceptrons**

Faculty of Mathematics and Computer Science, University of Bucharest
and
Sparktech Software

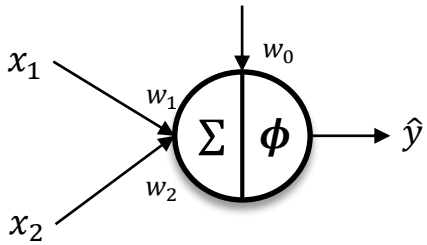
Academic Year 2018/2019, 1st Semester

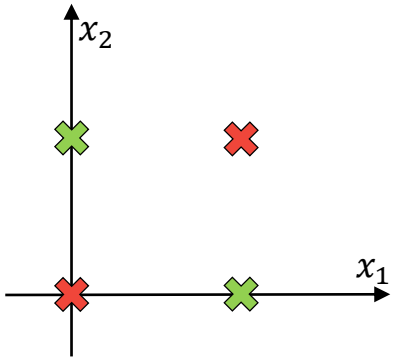
XOR with Perceptrons

- A single Perceptron cannot learn the XOR function because it is not linearly separable.

XOR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



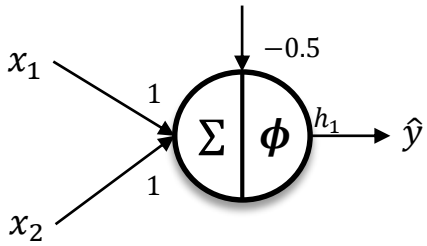


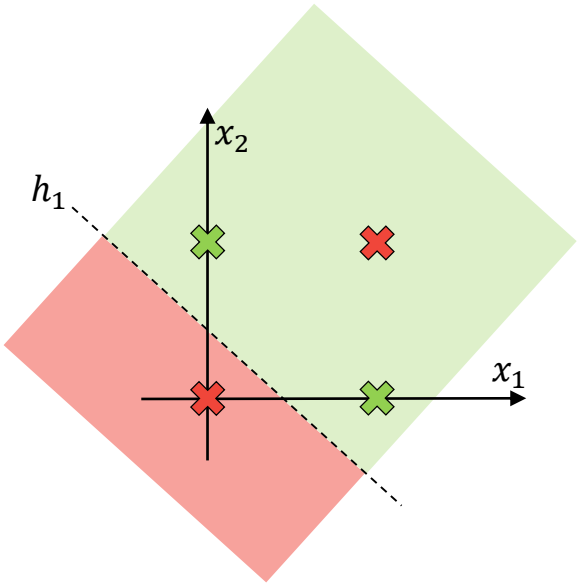
XOR with Perceptrons

- A single Perceptron cannot learn the XOR function because it is not linearly separable.

XOR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



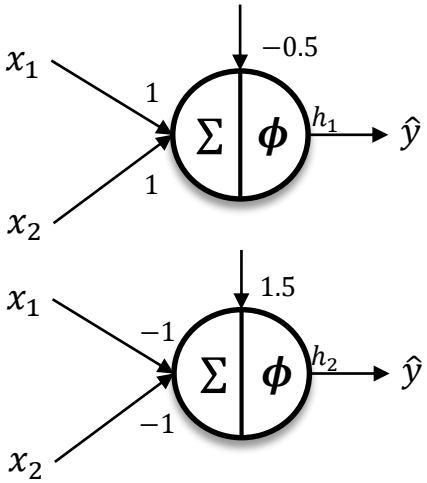


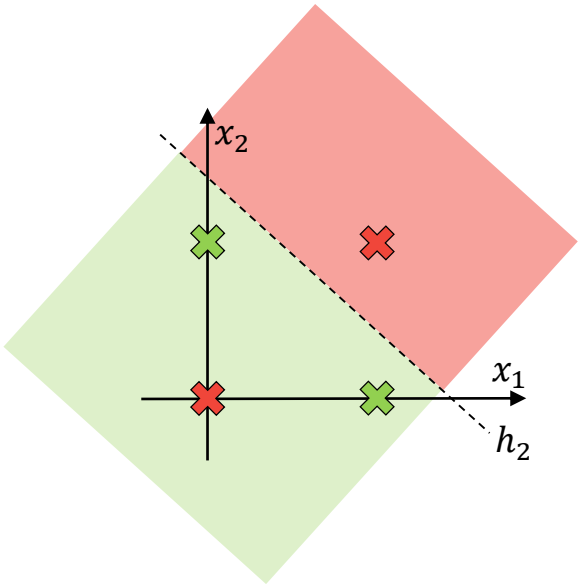
XOR with Perceptrons

- A single Perceptron cannot learn the XOR function because it is not linearly separable.

XOR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



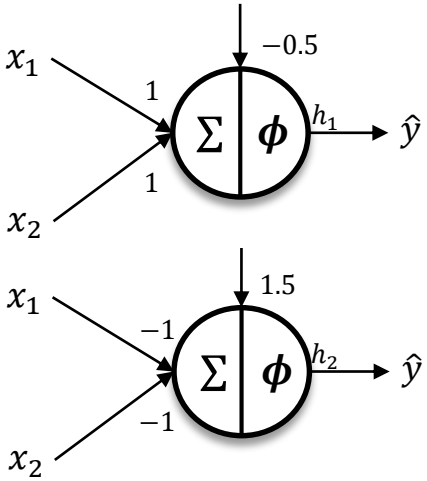


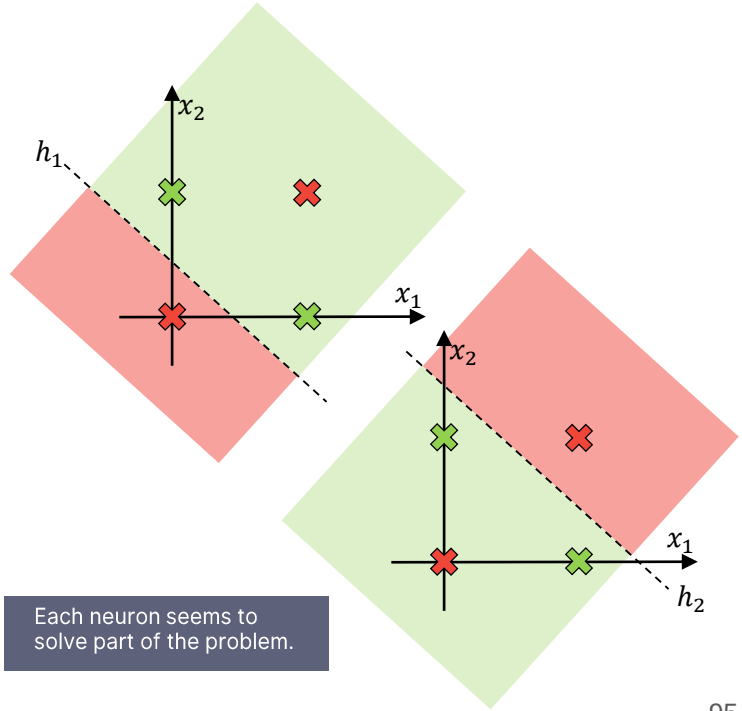
XOR with Perceptrons

- A single Perceptron cannot learn the XOR function because it is not linearly separable.

XOR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0





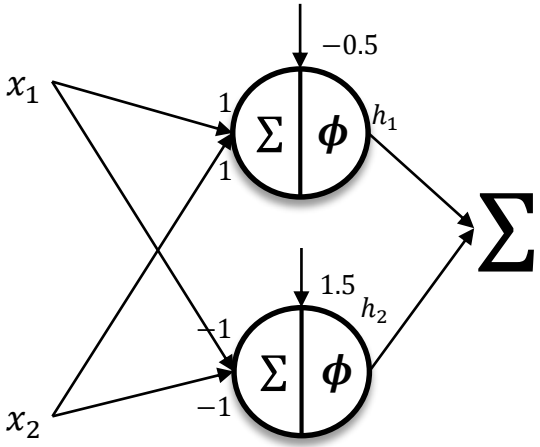
Each neuron seems to solve part of the problem.

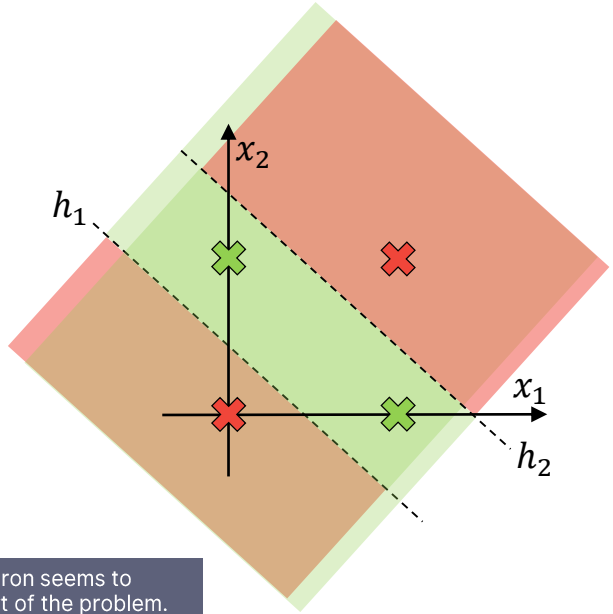
XOR with Perceptrons

- A single Perceptron cannot learn the XOR function because it is not linearly separable.
- This limitation can be overcome by “combining” the outputs of multiple Perceptrons.

XOR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0





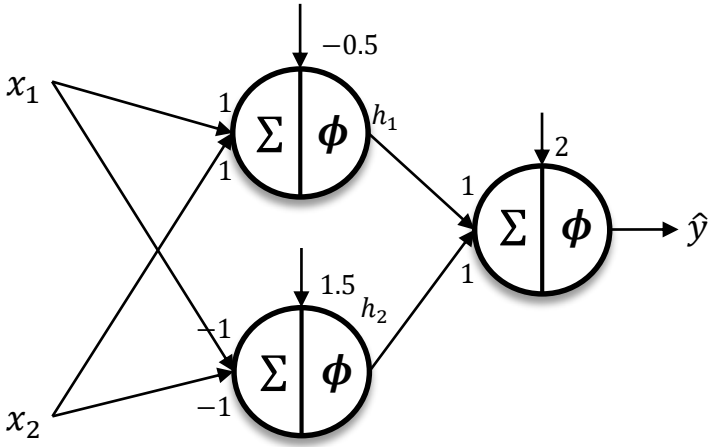
Each neuron seems to solve part of the problem.

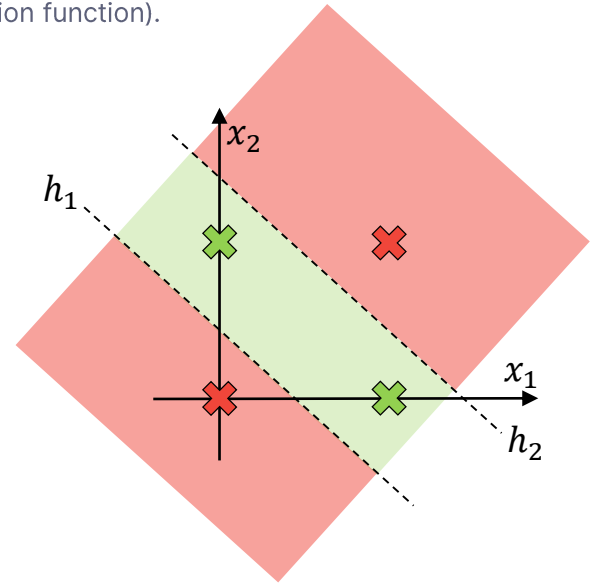
XOR with Perceptrons

- A single Perceptron cannot learn the XOR function because it is not linearly separable.
- This limitation can be overcome by "combining" the outputs of multiple Perceptrons.
  - We can combine them by using another Perceptron (weighed sum + activation function).

XOR

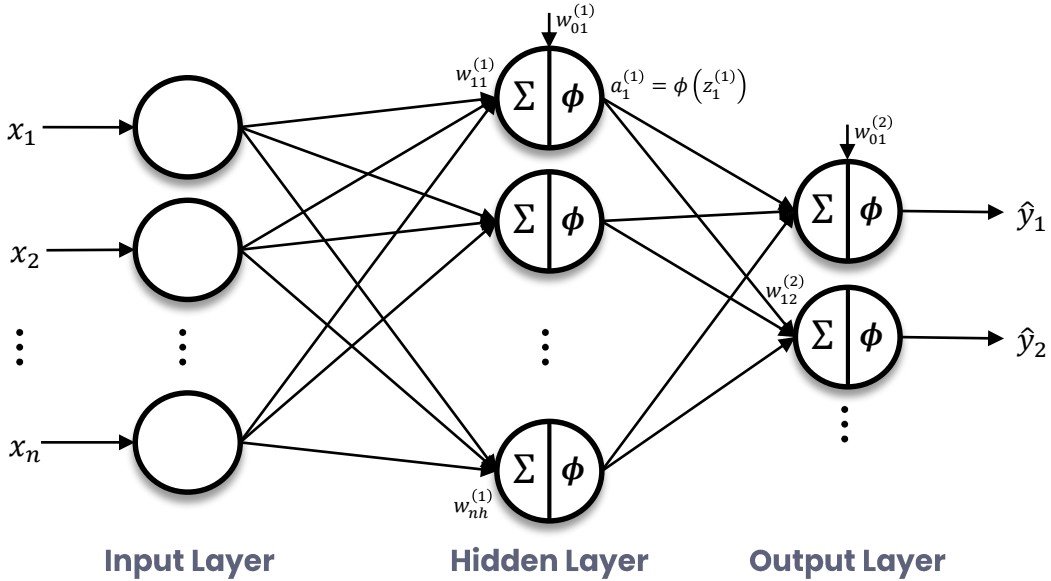
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0





The Multilayer Perceptron

- A **Multilayer Perceptron (MLP)** is a *feedforward artificial neural network* which has an *input layer*, one or more *hidden layers* and an *output layer*.
  - All neurons, excepts those from the input layer, apply an activation function to the weighted sum of inputs.
  - Each pair of neurons from consecutive layers has an associated weight.



- $w_{ij}^{(l)}$ is the weight from neuron i on layer $l-1$ to neuron j on layer l (input is layer 0).
- $w_{0j}^{(l)}$ is the bias of neuron j on layer l.
- $a_i^{(l)}$ is the output of neuron i on layer l.
- $z_i^{(l)}$ is the output before activation.

MLP in matrix format

$$\vec{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \qquad W^{(1)} = \begin{bmatrix} w_{01}^{(1)} & w_{11}^{(1)} & \cdots & w_{n1}^{(1)} \\ w_{02}^{(1)} & w_{12}^{(1)} & \cdots & w_{n2}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{0h}^{(1)} & w_{1h}^{(1)} & \cdots & w_{nh}^{(1)} \end{bmatrix}_{h \times n+1}$$

$$W^{(1)}\vec{x} = \begin{bmatrix} \sum_{i=0}^{n} x_i w_{i1}^{(1)} \\ \sum_{i=0}^{n} x_i w_{i2}^{(1)} \\ \vdots \\ \sum_{i=0}^{n} x_i w_{ih}^{(1)} \end{bmatrix} = \begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \\ \vdots \\ z_h^{(1)} \end{bmatrix} = \vec{z}^{(1)}$$

$$\phi\left(\vec{z}^{(1)}\right) = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_h^{(1)} \end{bmatrix} \qquad \vec{a}^{(1)} = \begin{bmatrix} 1 \\ a_1^{(1)} \\ \vdots \\ a_h^{(1)} \end{bmatrix} \qquad \hat{y} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_o \end{bmatrix} = \phi\left(W^{(2)}\phi\left(W^{(1)}x\right)\right)$$

$$\vec{x} \rightarrow \vec{z}^{(1)} = W^{(1)}\vec{x} \rightarrow \vec{a}^{(1)} = \left[1; \phi\left(\vec{z}^{(1)}\right)\right] \rightarrow \vec{z}^{(2)} = W^{(2)}\vec{a}^{(1)} \rightarrow \hat{y} = \vec{a}^{(2)} = \phi\left(\vec{z}^{(2)}\right)$$

Training an MLP

- A multilayer perceptron is trained with **stochastic gradient descent**.
  - "*Stochastic*" because the gradient is computed only with respect to a single training example or a batch, not the entire dataset.
  - We need to compute the gradient of the error function with respect to each weight of the network and update the weights correspondingly.

$$E(\vec{x}) = \mathcal{L}(y, \hat{y})$$

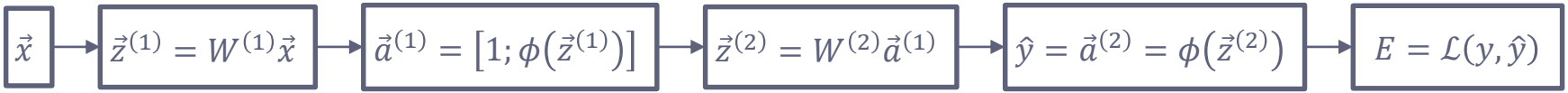
$$\Delta w_{ij}^{(l)} = -\eta \frac{\partial E}{\partial w_{ij}^{(l)}}$$

$\mathcal{L}$ is the loss (a function which should be small if $\hat{y}$ is close to y and larger otherwise).

  - Or in matrix format:

$$\Delta W^{(l)} = -\eta \frac{\partial E}{\partial W^{(l)}}$$

The Chain Rule



- The error is a function which depends on all the weights of the network.

$$E(\vec{x}) = \mathcal{L}\left(y, \phi\left(W^{(2)}\phi\left(W^{(1)}\vec{x}\right)\right)\right)$$

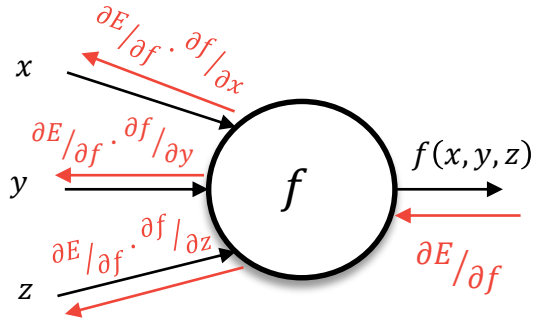
- We could pick any weight $w_{ij}^{(l)}$ and use the chain rule to compute the formula for $\frac{\partial E}{\partial w_{ij}^{(l)}}$.
- In matrix format:

$$\frac{\partial E}{\partial W^{(1)}} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \vec{z}^{(2)}}\frac{\partial \vec{z}^{(2)}}{\partial \vec{a}^{(1)}}\frac{\partial \vec{a}^{(1)}}{\partial \vec{z}^{(1)}}\frac{\partial \vec{z}^{(1)}}{\partial W^{(1)}}$$

$$\frac{\partial E}{\partial W^{(2)}} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \vec{z}^{(2)}}\frac{\partial \vec{z}^{(2)}}{\partial W^{(2)}}$$

Backpropagation of error

- Computing the derivative *formula* (symbolic differentiation) for every weight in the network is both inefficient and very complex for larger networks.
- We are only interested in the numerical evaluation of the derivatives, we can focus on one gate at a time and we can used previously computed values.

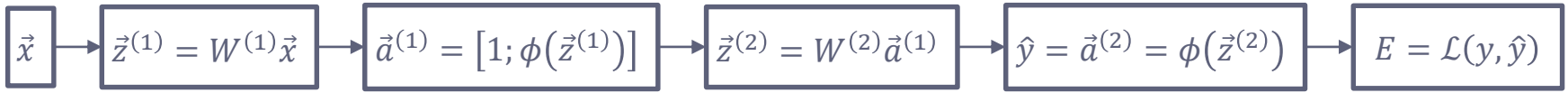


$\partial E/\partial f$ is already computed numerically.

- This method is known as "**backpropagation**".
  - **„Learning representations by back-propagating errors"**

David E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams, 1986

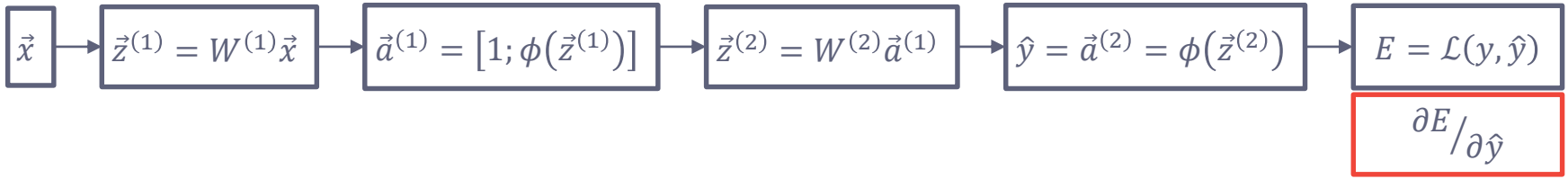
Backpropagation of error



$$\frac{\partial E}{\partial W^{(2)}} =$$

$$\frac{\partial E}{\partial W^{(1)}} =$$

Backpropagation of error

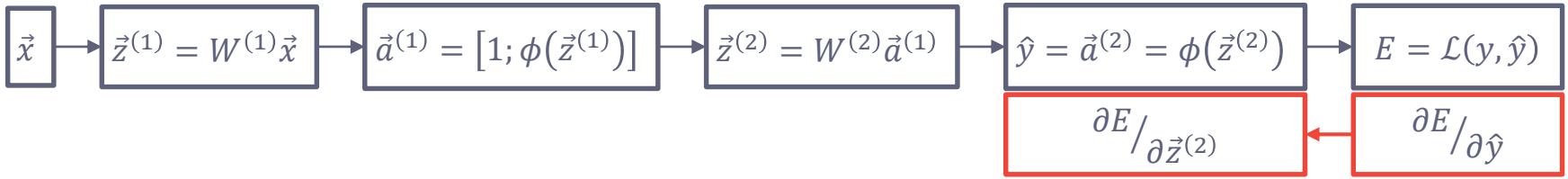


$$\frac{\partial E}{\partial W^{(2)}} =$$

$$\frac{\partial E}{\partial W^{(1)}} =$$

$$\boldsymbol{\frac{\partial E}{\partial \hat{y}} = \mathcal{L}'(y, \hat{y})}$$

Backpropagation of error



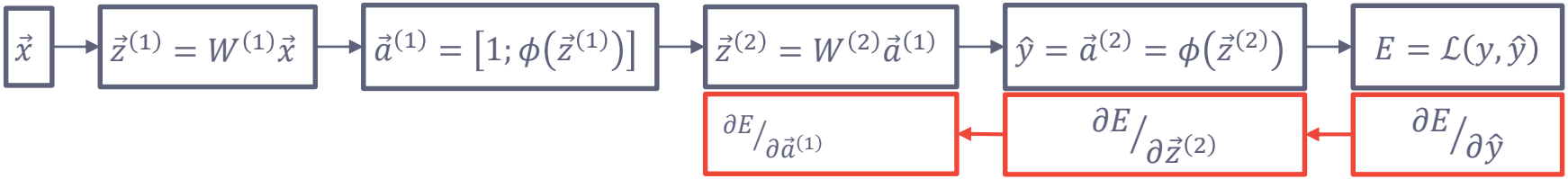
$$\frac{\partial E}{\partial W^{(2)}} =$$

$$\frac{\partial E}{\partial W^{(1)}} =$$

$$\frac{\partial E}{\partial \hat{y}} = \mathcal{L}'(y, \hat{y})$$

$$\boldsymbol{\frac{\partial E}{\partial \vec{z}^{(2)}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \vec{z}^{(2)}} = \frac{\partial E}{\partial \hat{y}} \phi'(\vec{z}^{(2)})}$$

Backpropagation of error



$$\frac{\partial E}{\partial W^{(2)}} =$$

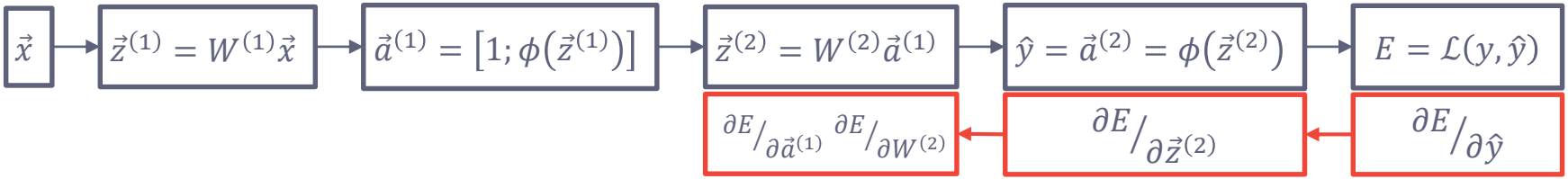
$$\frac{\partial E}{\partial W^{(1)}} =$$

$$\frac{\partial E}{\partial \hat{y}} = \mathcal{L}'(y, \hat{y})$$

$$\frac{\partial E}{\partial \vec{z}^{(2)}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \vec{z}^{(2)}} = \frac{\partial E}{\partial \hat{y}} \phi'(\vec{z}^{(2)})$$

$$\boldsymbol{\frac{\partial E}{\partial \vec{a}^{(1)}} = \frac{\partial E}{\partial \vec{z}^{(2)}} \frac{\partial \vec{z}^{(2)}}{\partial \vec{a}^{(1)}} = \frac{\partial E}{\partial \vec{z}^{(2)}} W^{(2)}}$$

Backpropagation of error



$$\frac{\partial E}{\partial W^{(2)}} = \frac{\partial E}{\partial \vec{z}^{(2)}} \frac{\partial \vec{z}^{(2)}}{\partial W^{(2)}} = \frac{\partial E}{\partial \vec{z}^{(2)}} \vec{a}^{(1)}$$

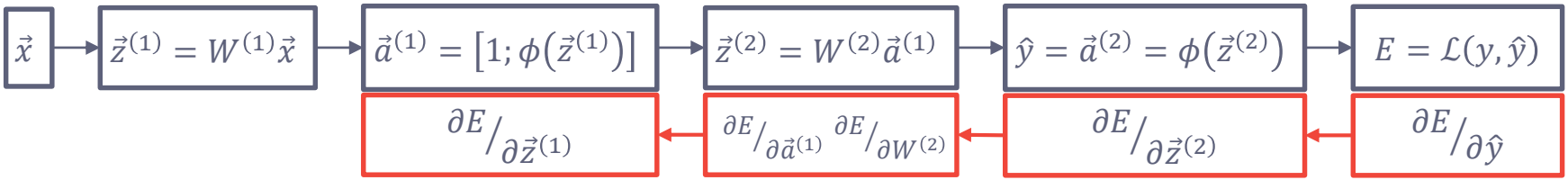
$$\frac{\partial E}{\partial W^{(1)}} =$$

$$\frac{\partial E}{\partial \hat{y}} = \mathcal{L}'(y, \hat{y})$$

$$\frac{\partial E}{\partial \vec{z}^{(2)}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \vec{z}^{(2)}} = \frac{\partial E}{\partial \hat{y}} \phi'(\vec{z}^{(2)})$$

$$\frac{\partial E}{\partial \vec{a}^{(1)}} = \frac{\partial E}{\partial \vec{z}^{(2)}} \frac{\partial \vec{z}^{(2)}}{\partial \vec{a}^{(1)}} = \frac{\partial E}{\partial \vec{z}^{(2)}} W^{(2)}$$

Backpropagation of error



$$\frac{\partial E}{\partial W^{(2)}} = \frac{\partial E}{\partial \vec{z}^{(2)}} \frac{\partial \vec{z}^{(2)}}{\partial W^{(2)}} = \frac{\partial E}{\partial \vec{z}^{(2)}} \vec{a}^{(1)}$$

$$\frac{\partial E}{\partial W^{(1)}} =$$

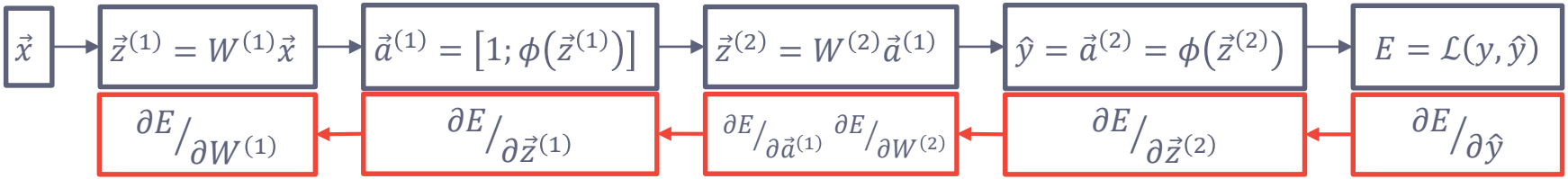
$$\frac{\partial E}{\partial \hat{y}} = \mathcal{L}'(y, \hat{y})$$

$$\frac{\partial E}{\partial \vec{z}^{(2)}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \vec{z}^{(2)}} = \frac{\partial E}{\partial \hat{y}} \phi'(\vec{z}^{(2)})$$

$$\frac{\partial E}{\partial \vec{a}^{(1)}} = \frac{\partial E}{\partial \vec{z}^{(2)}} \frac{\partial \vec{z}^{(2)}}{\partial \vec{a}^{(1)}} = \frac{\partial E}{\partial \vec{z}^{(2)}} W^{(2)}$$

$$\boldsymbol{\frac{\partial E}{\partial \vec{z}^{(1)}} = \frac{\partial E}{\partial \vec{a}^{(1)}} \frac{\partial \vec{a}^{(1)}}{\partial \vec{z}^{(1)}} = \frac{\partial E}{\partial \vec{a}^{(1)}} \phi'(\vec{z}^{(1)})}$$

Backpropagation of error



$$\frac{\partial E}{\partial W^{(2)}} = \frac{\partial E}{\partial \vec{z}^{(2)}} \frac{\partial \vec{z}^{(2)}}{\partial W^{(2)}} = \frac{\partial E}{\partial \vec{z}^{(2)}} \vec{a}^{(1)}$$

$$\boldsymbol{\frac{\partial E}{\partial W^{(1)}} = \frac{\partial E}{\partial \vec{z}^{(1)}} \frac{\partial \vec{z}^{(1)}}{\partial W^{(1)}} = \frac{\partial E}{\partial \vec{a}^{(1)}} \vec{x}}$$

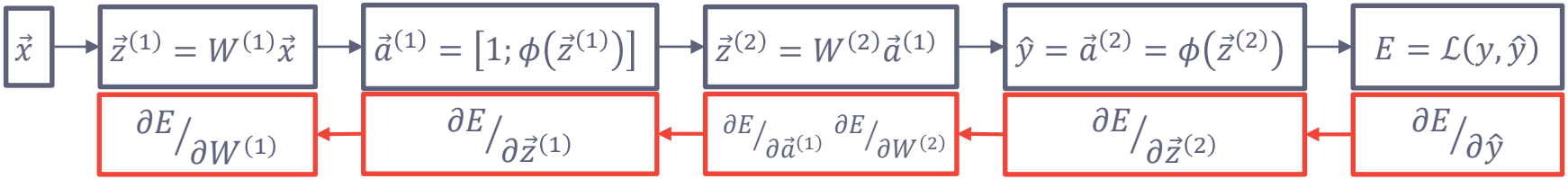
$$\frac{\partial E}{\partial \hat{y}} = \mathcal{L}'(y, \hat{y})$$

$$\frac{\partial E}{\partial \vec{z}^{(2)}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \vec{z}^{(2)}} = \frac{\partial E}{\partial \hat{y}} \phi'(\vec{z}^{(2)})$$

$$\frac{\partial E}{\partial \vec{a}^{(1)}} = \frac{\partial E}{\partial \vec{z}^{(2)}} \frac{\partial \vec{z}^{(2)}}{\partial \vec{a}^{(1)}} = \frac{\partial E}{\partial \vec{z}^{(2)}} W^{(2)}$$

$$\boldsymbol{\frac{\partial E}{\partial \vec{z}^{(1)}} = \frac{\partial E}{\partial \vec{a}^{(1)}} \frac{\partial \vec{a}^{(1)}}{\partial \vec{z}^{(1)}} = \frac{\partial E}{\partial \vec{a}^{(1)}} \phi'(\vec{z}^{(1)})}$$

Backpropagation of error



$$\frac{\partial E}{\partial W^{(2)}} = \frac{\partial E}{\partial \vec{z}^{(2)}}\frac{\partial \vec{z}^{(2)}}{\partial W^{(2)}} = \frac{\partial E}{\partial \vec{z}^{(2)}}\vec{a}^{(1)}$$

$$\frac{\partial E}{\partial W^{(1)}} = \frac{\partial E}{\partial \vec{z}^{(1)}}\frac{\partial \vec{z}^{(1)}}{\partial W^{(1)}} = \frac{\partial E}{\partial \vec{a}^{(1)}}\vec{x}$$

$$\boldsymbol{\Delta W^{(l)} = \eta \frac{\partial E}{\partial W^{(l)}}}$$

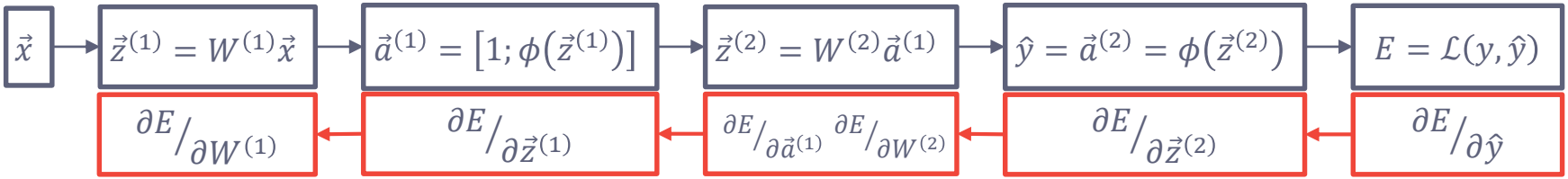
$$\frac{\partial E}{\partial \hat{y}} = \mathcal{L}'(y, \hat{y})$$

$$\frac{\partial E}{\partial \vec{z}^{(2)}} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \vec{z}^{(2)}} = \frac{\partial E}{\partial \hat{y}}\phi'(\vec{z}^{(2)})$$

$$\frac{\partial E}{\partial \vec{a}^{(1)}} = \frac{\partial E}{\partial \vec{z}^{(2)}}\frac{\partial \vec{z}^{(2)}}{\partial \vec{a}^{(1)}} = \frac{\partial E}{\partial \vec{z}^{(2)}}W^{(2)}$$

$$\frac{\partial E}{\partial \vec{z}^{(1)}} = \frac{\partial E}{\partial \vec{a}^{(1)}}\frac{\partial \vec{a}^{(1)}}{\partial \vec{z}^{(1)}} = \frac{\partial E}{\partial \vec{a}^{(1)}}\phi'(\vec{z}^{(1)})$$

Backpropagation of error



$$\frac{\partial E}{\partial W^{(2)}} = \frac{\partial E}{\partial \vec{z}^{(2)}}\frac{\partial \vec{z}^{(2)}}{\partial W^{(2)}} = \frac{\partial E}{\partial \vec{z}^{(2)}}\vec{a}^{(1)}$$

$$\frac{\partial E}{\partial W^{(1)}} = \frac{\partial E}{\partial \vec{z}^{(1)}}\frac{\partial \vec{z}^{(1)}}{\partial W^{(1)}} = \frac{\partial E}{\partial \vec{z}^{(1)}}\vec{x}$$

$$\boldsymbol{\Delta W^{(l)} = \eta \frac{\partial E}{\partial W^{(l)}}}$$

$$\frac{\partial E}{\partial \hat{y}} = \mathcal{L}'(y, \hat{y})$$

Derivative of activation function.

$$\frac{\partial E}{\partial \vec{z}^{(2)}} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \vec{z}^{(2)}} = \frac{\partial E}{\partial \hat{y}}\phi'(\vec{z}^{(2)})$$

$$\frac{\partial E}{\partial \vec{a}^{(1)}} = \frac{\partial E}{\partial \vec{z}^{(2)}}\frac{\partial \vec{z}^{(2)}}{\partial \vec{a}^{(1)}} = \frac{\partial E}{\partial \vec{z}^{(2)}}W^{(2)}$$

$$\frac{\partial E}{\partial \vec{z}^{(1)}} = \frac{\partial E}{\partial \vec{a}^{(1)}}\frac{\partial \vec{a}^{(1)}}{\partial \vec{z}^{(1)}} = \frac{\partial E}{\partial \vec{a}^{(1)}}\phi'(\vec{z}^{(1)})$$

Choosing an activation function

- In order to do backpropagation we need to compute the *derivative of the activation function*.
  - The *unit step function* of the perceptron is not differentiable.
  - The *linear activation* used for *Adaline* does not benefit from multiple layers:

$$\hat{y} = \phi_{\text{linear}}\left(W^{(2)}\phi_{\text{linear}}\left(W^{(1)}x\right)\right) = W^{(2)}W^{(1)}x = W'x$$

- We are looking for a step-like differentiable function.

Choosing an activation function

- In order to do backpropagation we need to compute the *derivative of the activation function*.
  - The *unit step function* of the perceptron is not differentiable.
  - The *linear activation* used for *Adaline* does not benefit from multiple layers:

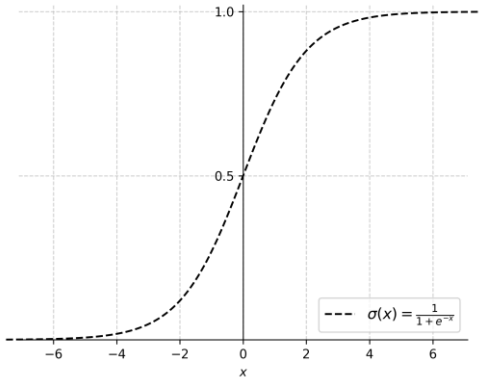
$$\hat{y} = \phi_{\text{linear}}\left(W^{(2)}\phi_{\text{linear}}\left(W^{(1)}x\right)\right) = W^{(2)}W^{(1)}x = W'x$$

- We are looking for a step-like differentiable function.
  - **Standard Logistic Function**

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

$$\sigma'(x) = \sigma(x)\left(1-\sigma(x)\right)$$

Usually referred to as "the" **sigmoid**.



Choosing a loss function

- Adaline used a **least squares loss** function:

$$\mathcal{L}(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2 \qquad \mathcal{L}'(y, \hat{y}) = -(y - \hat{y})$$

Choosing a loss function

- Adaline used a **least squares loss** function:

$$\mathcal{L}(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2 \qquad \mathcal{L}'(y, \hat{y}) = -(y - \hat{y})$$

- The **Cross-entropy loss** is much more common in practice.

$$\mathcal{L}(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}) \qquad \mathcal{L}'(y, \hat{y}) = -\frac{y}{\hat{y}} - \frac{1 - y}{1 - \hat{y}}$$

- Typically, the output layer uses **Softmax** activation, instead of a *sigmoid* for each neuron, to make it suitable for probabilistic interpretation.

$$\hat{y}_j = \frac{e^{z_j^{(2)}}}{\sum_{k=1}^{o} e^{z_k^{(2)}}} \qquad \text{instead of} \qquad \hat{y}_j = a_j^{(2)} = \sigma\left(z_j^{(2)}\right)$$

Universal approximation theorem

- The **universal approximation theorem** states that a multilayer perceptron with a single hidden layer containing a finite number of neurons is sufficient to represent **any function** given appropriate parameters.
  - However, it does not say anything about the learnability of those parameters.
- This implies that the model which the MLP learns can be *arbitrarily complex*, which can rapidly lead to **overfitting**.

MLP
Regularization Techniques

Weight Decay

- Similarly to *Ridge Regression* and *SVMs*, keeping the weights small makes the represented function smoother and less prone to overfitting.

$$E = \mathcal{L}(y, \hat{y}) + \frac{\lambda}{2} \|\vec{w}\|^2$$

Tikhonov regularization.

Weight Decay

- Similarly to *Ridge Regression* and *SVMs*, keeping the weights small makes the represented function smoother and less prone to overfitting.

$$E = \mathcal{L}(y, \hat{y}) + \frac{\lambda}{2} \|\vec{w}\|^2$$

Tikhonov regularization.

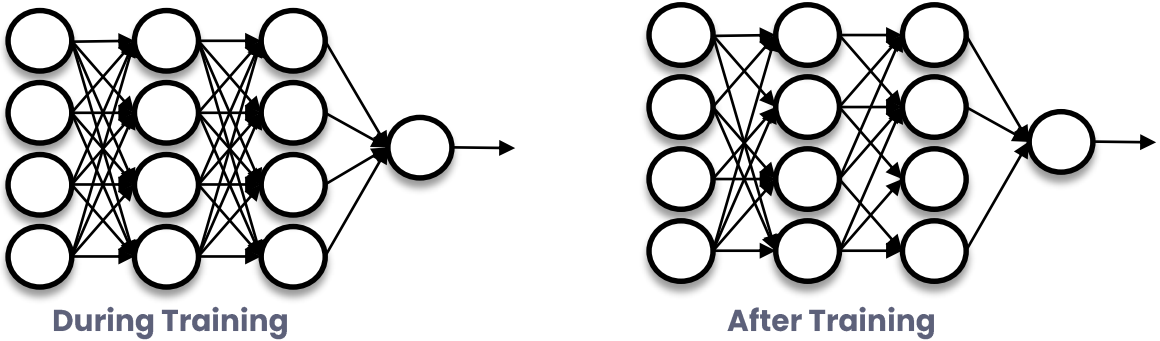
- By applying *gradient descent* to the new error function we obtain:

$$\Delta w_{ij}^{(l)} = -\eta \frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} - \eta \lambda w_{ij}^{(l)}$$

Subtracting a faction of the weight.

Optimal Brain Damage

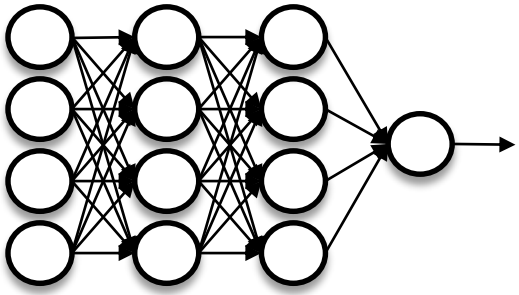
- After training, completely remove some weights (set them to 0) if it does not affect training error too much.
  - Similar to pruning decision trees.
  - **“Optimal Brain Damage”** Yann LeCun, John S. Denker and Sara A. Solla, 1990



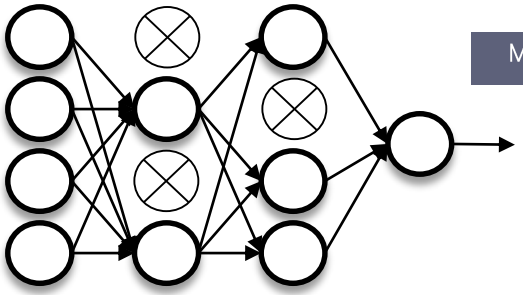
Dropout

- **Dropout** is a regularization technique which randomly disables certain neurons during some steps of the training phase.
  - **"Dropout: A Simple Way to Prevent Neural Networks from Overfitting"**

    N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, 2014



Without Dropout



With Dropout

Much newer technique!

- Each neuron has a probability $p_i^{(l)}$ of being dropped.
- After training, during inference, all neurons are used.

Popularity of Neural Networks

- During the 90s and early 2000s, NNs were not very popular in comparison to other algorithms (especially *SVMs*).
  - Neural networks have a lot of parameters to learn.
    - They requires **lots of training data** and lots of **computational resources**.
  - They also have many *hyperparameters* to tune.
    - Learning rate, Batch size, Regularization strength, Dropout probability
    - But most importantly, the **architecture of the network itself**.
  - Simply put, they were slow, hard to train and did not work as well as other techniques
- There are 3 key factors which determined the large increase in performance and, hence, popularity of neural networks, under the name "**Deep Learning**", in the late 2000s and early 2010s:
  - **Training Data**, **Computing Power** and **Algorithms**.

Popularity of Neural Networks

- **Training Data**
  - Large neural networks require huge amounts of data for training.
  - Many large, high-quality datasets have been published over the past few years, which were not available to researchers in the 90s.
  - More data is generated everyday nowadays than was generated in years a decade ago.
- **Computing Power**
  - The use of GPUs and dedicated architectures, like TPUs, for training has significantly improved the speed and allowed the use of much larger neural networks.
  - *“a training run that takes one day on a single TPU device would have taken a quarter of a million years on an 80486 from 1990”.* - Shane Legg, cofounder of Google DeepMind.
- **Algorithms**
  - Improvements in network structure (convolutions, recurrent Networks), activation functions (ReLu) and optimizers (RMSprop, ADAM) have greatly improved the performance of NNs.

Keywords

Perceptron

McCulloch-Pitts Neuron

Hebb's Rule

Linear Separability

AI Winter

Learning Rate

Batch Update

Gradient Descent

Chain Rule

Delta Rule

Multilayer Perceptron

MLP

Hidden Layer

Backpropagation

Universal approximation theorem

Weight Decay

Optimal Brain Damage

Dropout

Warren McCulloch

Walter Pits

Donald Hebb

Frank Rosenblatt

Marvin Minsky

Seymour Papert

Bernard Widrow

Ted Hoff

Geoffrey Hinton