# Perceptron

Lecturer: Ngo Huy Bien
Software Engineering Department
Faculty of Information Technology
University of Science
Ho Chi Minh City, Vietnam
nhbien@fit.hcmus.edu.vn

## Objectives

- To *write* a machine learning program recognizing a handwritten digit using Perceptron
- To explain neural network *terminologies* via examples

## Contents



## References

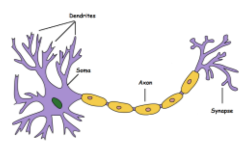
1. Raul Rojas (1996). Neural Networks: A Systematic Introduction. Springer.
2. Aurelien Geron (2017). Hands on Machine Learning with Scikit Learn and TensorFlow. O’Reilly Media.

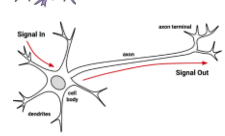
## Why Do We Smile?

## A Biological Neuron

- https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1

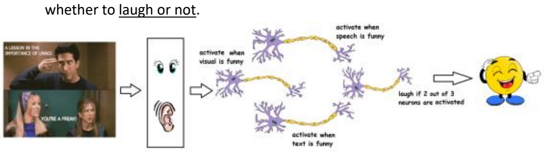
- *Dendrite*: Receives signals from other neurons
- *Soma*: Processes the information
- *Axon*: Transmits the output of this neuron
- *Synapse*: Point of connection to other neurons



## An Overly Simplified Illustration

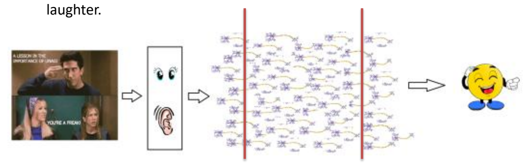
- Let's say you are watching Friends.
- Each neuron *only fires* when its intended criteria is met i.e., a neuron may perform a certain role to a certain stimulus.
- The "laugh or not" set of neurons that will help you *make a decision* on whether to laugh or not.




- In reality, it is *not* just a couple of neurons which would do the decision making.

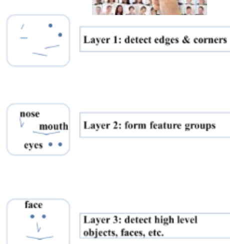
## Neural Network

- There is a *massively* parallel interconnected network of $10^{11}$ neurons (100 billion) in our brain.
- Now the sense organs pass the information to the *first/lowest layer* of neurons to process it.
- And the output of the processes is passed on to the *next layers* in a hierarchical manner, some of the neurons will fire and some won't and this process goes on until it results in a *final response* — in this case, laughter.

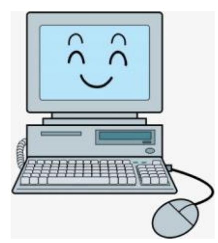


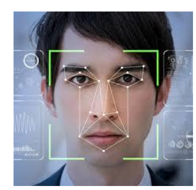
## How Do We Recognize People?

- It is believed that neurons are arranged in a hierarchical fashion (however, many credible alternatives with experimental support are proposed by the scientists) and each *layer* has its own role and responsibility.
- To detect a face, the brain could be relying on the *entire network* and not on a single layer.



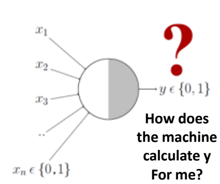

## Can Machine Smile or Recognize People?



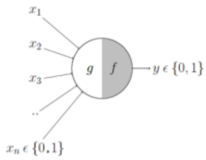


## How to Model a Neuron?

- Let's suppose that I want a machine to predict for me *whether* I should watch a random football game or not on TV.
- I give it 4 inputs:
  - $x_1$: *isPremierLeagueOn* (I like Premier League more): **1 or 0**
  - $x_2$: *isManUnitedPlaying* (I am a big Man United fan.): **1 or 0**
  - $x_3$: *hasFreeTime* (Do I have free time?): **1 or 0**
  - $x_4$: *isNotHome* (Can't watch it when I'm running errands. Can I?): **1 or 0**



## McCulloch-Pitts Neuron: Mathematical Model of a Biological Neuron

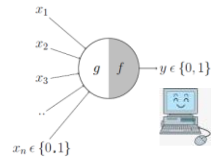



$$g(x_1, x_2, x_3, ..., x_n) = g(\mathbf{x}) = \sum_{i=1}^{n} x_i$$

$$y = f(g(\mathbf{x})) = 1 \quad if \quad g(\mathbf{x}) \geq \theta$$
$$= 0 \quad if \quad g(\mathbf{x}) < \theta$$

- **`g(x)`** is just doing a sum of the inputs — a simple aggregation.
- $x_1$ = 1, $x_2$ = 1, $x_3$ = 1, $x_4$ = 0. **`g(x) = ?`**
- ***`theta`*** here is called thresholding parameter. If I always watch the game when the sum turns out to be 2 or more, the theta is 2 here. This is called the *thresholding logic*.
- $x_1$ = 1, $x_2$ = 1, $x_3$ = 1, $x_4$ = 0, θ =2 ⇒ Should I what the football game?

## Telling a Machine To Laugh Using M-P Neuron




$$g(x_1, x_2, x_3, ..., x_n) = g(\mathbf{x}) = \sum_{i=1}^{n} x_i$$

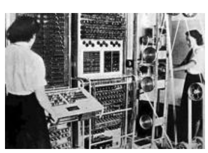
$$y = f(g(\mathbf{x})) = 1 \quad if \quad g(\mathbf{x}) \geq \theta$$
$$= 0 \quad if \quad g(\mathbf{x}) < \theta$$

- *In reality*, I laugh when having 10 inputs and the sum turns out to be 4 or more.
- Then I will code the machine with a function that sums 10 *inputs*, set *theta* to 4, and give it a "Laugh" icon.
- When I give the machine NEW input *values*, it will calculate the sum and if the sum turns out to be 4 or more, it will show the "Laugh" icon.

## Representing Boolean Functions Using M-P Neuron. Why?

| A | B | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

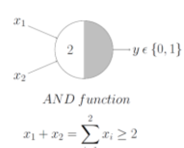
| Input | Input | Output |
|---|---|---|
| A | B | Y |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |



## Representing an AND Function

- *In reality*, I have 2 inputs and want to calculate the AND value of these inputs.
- How do I tell a machine to do this task for me?

**`g(x) = x₁ + x₂ ≥ ? (i.e. y = 1)`**



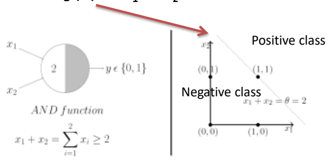

| Input | Input | Output |
|---|---|---|
| A | B | Y |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Find a threshold, then code the sum function and the threshold in the machine!

## Geometric Interpretation

- Here, all the input points that lie ON or ABOVE, just (1, 1), output 1 when passed through the M-P AND neuron.

**`g(x) = x₁ + x₂ = 2`**

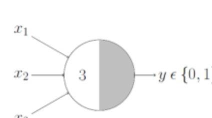



- The M-P AND neuron learnt a *linear decision boundary*!
- The M-P AND neuron is splitting the input sets into *two classes* — positive and negative.

## How About 3-Input AND Function?

- *In reality*, I have 3 inputs and want to calculate the AND value of these inputs.
- How do I tell a machine to do this task for me?

**`g(x) = x₁ + x₂ + x₃ ≥ ? (i.e. y = 1)`**



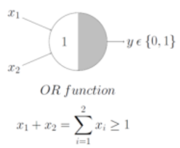

| A | B | C | Z |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

Find a threshold, then code the sum function and the threshold in the machine!

## Representing an OR Function

- *In reality*, I have <u>2 inputs</u> and want to calculate the OR value of these inputs.
- How do I tell a machine to do this task for me?

**g(x) = x$_1$ + x$_2$ ≥ ? (i.e. y = 1)**

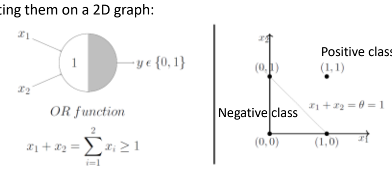



| A | B | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Find a threshold, then code the sum function and the threshold in the machine!

## Geometric Interpretation

- The inputs are obviously boolean, so only 4 combinations are possible — (0,0), (0,1), (1,0) and (1,1).
- Plotting them on a 2D graph:

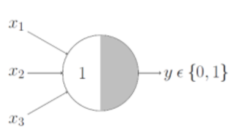



- The M-P OR neuron learnt a *linear decision boundary*!
- The M-P OR neuron is splitting the <u>input sets</u> into *two classes* — <u>positive</u> and <u>negative</u>.

## How About 3-Input OR Function?

- *In reality*, I have <u>3 inputs</u> and want to calculate the OR value of these inputs.
- How do I tell a machine to do this task for me?

**g(x) = x$_1$ + x$_2$ + x$_3$ ≥ ? (i.e. y = 1)**

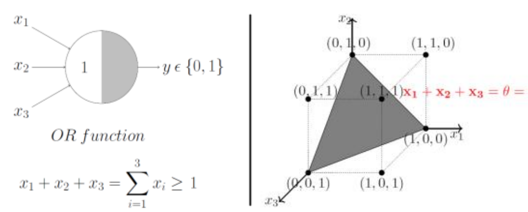



| A | B | C | Z |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Find a threshold, then code the sum function and the threshold in the machine!

## Geometric Interpretation

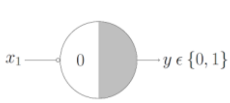
- All the points that lie ON or ABOVE that plane (positive half space) will result in output 1 when passed through the OR function M-P unit and all the points that lie BELOW that plane (negative half space) will result in output 0.




## Representing a NOT Function

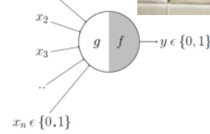
- *In reality*, I have <u>1 input</u> and want to calculate the NOT value of this input.
- How do I tell a machine to do this task for me?

**g(x) = x$_1$ ≥ ?**




| Input | Output |
|---|---|
| A | Y |
| 0 | 1 |
| 1 | 0 |

## Inhibitory Inputs




- These inputs can either be *excitatory* or *inhibitory*.
- *Excitatory* inputs are NOT the ones that will make the neuron fire on their own but they might fire it when combined together.
- *Inhibitory* inputs are those that have maximum effect on the decision making irrespective of other inputs.

- Let's suppose that I want a <u>machine</u> to <u>predict</u> for me *whether* I should watch a random football game or not on TV.
- I give it 4 inputs:
  - ***x$_1$***: *isPremierLeagueOn* (I like Premier League more): **1 or 0**
  - ***x$_2$***: *isManUnitedPlaying* (I am a big Man United fan.): **1 or 0**
  - ***x$_3$***: *hasFreeTime* (Do I have free time?): **1 or 0**
  - ***x$_4$***: *isNotHome* (Can't watch it when I'm running errands. Can I?): **1 or 0**

## Representing a NOT Function

- *In reality*, I have 1 input and want to calculate the NOT value of this input.
- How do I tell a machine to do this task for me?
- Take the input as an *inhibitory input*, i.e. set below rule

  **if $x_1$ = 1 then y = 0**

  then set the thresholding parameter to ?,

  **$g(x) = x_1 \geq$ ?**


| Input | Output |
|---|---|
| A | Y |
| 0 | 1 |
| 1 | 0 |

Find inhibitory rules and a threshold,
then code the inhibitory rules and the threshold in the machine!

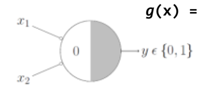
## Representing NOR Function

- *In reality*, I have 2 inputs and want to calculate the NOR value of these inputs.
- How do I tell a machine to do this task for me?
- For a NOR neuron to fire, we want ALL the inputs 1 to be 0 so we take them all as *inhibitory inputs*, i.e. set below rule:

  **if $x_1$ = 1 or $x_2$ = 1 then y = 0**

  then set the thresholding parameter to 0,

  **$g(x) = x_1 + x_1 \geq$ ?**




| Inputs | | Outputs |
|---|---|---|
| X | Y | Z |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

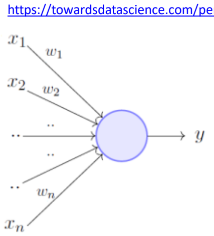
Find inhibitory rules and a threshold,
then code the inhibitory rules and the threshold in the machine!

## Limitations Of M-P Neuron

- What about *non-boolean* (say, real) inputs?
- Do we always need to *hand code* the threshold?
- Are all inputs *equal*? What if we want to assign *more importance* to some inputs?
- What about functions which are *not linearly separable*? Say XOR function.

## Perceptron

https://towardsdatascience.com/perceptron-the-artificial-neuron-4d8c70d5cc8d




Perceptron Model (Minsky-Papert in 1969)

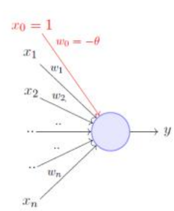
$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i \geq \theta$$
$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i < \theta$$

Rewriting the above,

$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0$$
$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0$$

It overcomes some of the limitations of the M-P neuron by introducing the concept of *numerical weights* (a measure of importance) for inputs, and a mechanism for *learning those weights*.

## A Convention

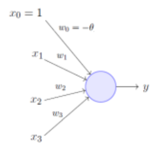



A more accepted convention,

$$y = 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0$$
$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

$$where, \quad x_0 = 1 \quad and \quad w_0 = -\theta$$

## An Example




$x_1 = isPremierLeagueOn$
$x_2 = isManUnitedPlaying$
$x_3 = isFriendlyGame$

$$y = 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0$$

$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

$$where, \quad x_0 = 1 \quad and \quad w_0 = -\theta$$

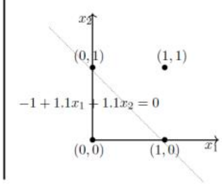
***$x_1$*= 1.2, *$x_2$* = 1.3, *$x_3$* = 1.7**
***$w_0$*= 3 *$w_1$*= 1.2, *$w_2$* = 1.3, *$w_3$* = 1.7**
What is the decision made by the machine?

## Another Example: OR Function

| $x_1$ | $x_2$ | OR | |
|---|---|---|---|
| 0 | 0 | 0 | $w_0 + \sum_{i=1}^{2} w_i x_i < 0$ |
| 1 | 0 | 1 | $w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$ |
| 0 | 1 | 1 | $w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$ |
| 1 | 1 | 1 | $w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$ |

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

One possible solution is
$w_0 = -1, \ w_1 = 1.1, \ w_2 = 1.1$




How do I tell a machine to calculate the OR function for me?

## Perceptron vs. McCulloch-Pitts Neuron

McCulloch Pitts Neuron
(assuming no inhibitory inputs)

$$y = 1 \quad if \sum_{i=0}^{n} x_i \geq 0$$
$$= 0 \quad if \sum_{i=0}^{n} x_i < 0$$

Perceptron

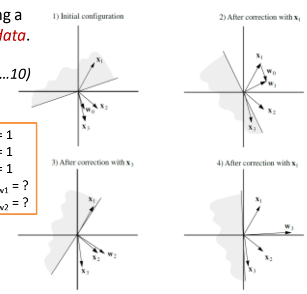
$$y = 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0$$
$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

- A *single Perceptron* can only be used to implement linearly separable functions, just like the M-P neuron.
- Then what is the *difference*?
- Here, the weights, including the threshold can be *learned* and the inputs can be *real* values. **What does this mean?**

## A Problem

Estimate if I will be watching a movie based on *historical data*.
*$x_1$: imdb rating (–10 … 10)*
$x_2$: *rotten tomato rating (–10 …10)*

| | |
|---|---|
| $\mathbf{x}_1(x_1 = 2, x_2 = 4)$ | $y_1 = 1$ |
| $\mathbf{x}_2(x_1 = 4, x_2 = -3)$ | $y_2 = 1$ |
| $\mathbf{x}_3(x_1 = -0.5, x_2 = -4)$ | $y_3 = 1$ |
| $\mathbf{x}_{new1}(x_1 = 1, x_2 = 1)$ | $y_{new1}$ = ? |
| $\mathbf{x}_{new2}(x_1 = -1.5, x_2 = -1.5)$ | $y_{new2}$ = ? |




## Perceptron Learning Algorithm [1]

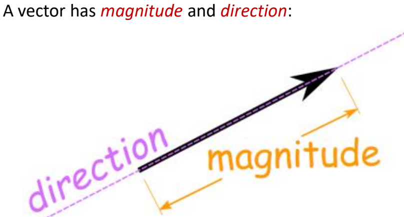
- The *training set* consists of two sets (containing m elements), P and N, in n+1 - dimensional extended input space: **x** = [$x_0$, $x_1$, $x_2$, …, $x_n$]
- We *look for* a vector **ω** = [$\omega_0$, $\omega_1$, $\omega_2$, …, $\omega_n$] capable of absolutely separating both sets, so that all vectors in P belong to the open positive half-space and all vectors in N to the open negative half-space of the linear separation.

Batch Perceptron

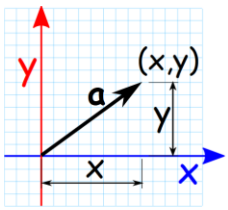
**input:** A training set $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$
**initialize:** $\mathbf{w}^{(1)} = (0, \ldots, 0)$
**for** $t = 1, 2, \ldots$
  **if** $(\exists\, i \text{ s.t. } y_i \langle \mathbf{w}^{(t)}, \mathbf{x}_i \rangle \leq 0)$ **then**
    $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + y_i \mathbf{x}_i$
  **else**
    **output** $\mathbf{w}^{(t)}$

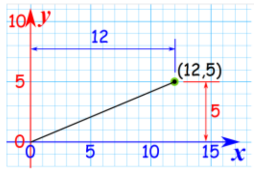
## Vector Review

- https://www.mathsisfun.com
- A vector has *magnitude* and *direction*:

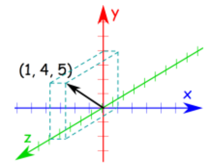


## Vector **a** in Cartesian Coordinates




## More Than 2 Dimensions




A *vector* is a list of numbers (can be in a row or column).

Vector

$[2\ \ -8\ \ 7]$ row

or column $\begin{bmatrix}2\\-8\\7\end{bmatrix}$

## Dot Product

- The Dot Product gives *a number* as an answer (a "scalar", not a vector).
- *Method 1 (definition)*:

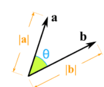
  **a** · **b** = |**a**| × |**b**| × cos(θ)

  Where:

  |**a**| is the magnitude (length) of vector **a**

  |**b**| is the magnitude (length) of vector **b**

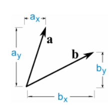
  θ is the angle between **a** and **b**




- *Method 2 (definition)*:

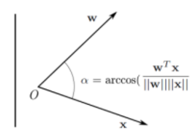
  **a** · **b** = $a_x \times b_x + a_y \times b_y$

  So we multiply the x's, multiply the y's, then add.




## In A n+1 Dimensional Space

$$\mathbf{w}^T\mathbf{x} = \|\mathbf{w}\|\|\mathbf{x}\| \cos\alpha \qquad \cos\alpha = \frac{\mathbf{w}^T\mathbf{x}}{||\mathbf{w}||||\mathbf{x}||}$$




$$\alpha = \arccos\left(\frac{\mathbf{w}^T\mathbf{x}}{||\mathbf{w}||||\mathbf{x}||}\right)$$

$$\mathbf{w} = [w_0, w_1, w_2, ..., w_n]$$
$$\mathbf{x} = [1, x_1, x_2, ..., x_n]$$
$$\mathbf{w} \cdot \mathbf{x} = \mathbf{w}^\mathbf{T}\mathbf{x} = \sum_{i=0}^{n} w_i * x_i$$

- When the dot product of two vectors is 0, they are *perpendicular* to each other.

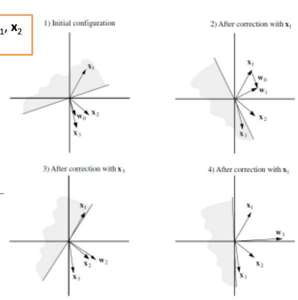
## Why Would The Update Rule Work?

$y_1 = 1$
$y_2 = 1$
$y_3 = 1$

Fin w (a line) so that $\mathbf{x}_1$, $\mathbf{x}_2$ and $\mathbf{x}_3$ is in a side.

**Batch Perceptron**

**input:** A training set $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$
**initialize:** $\mathbf{w}^{(1)} = (0, \ldots, 0)$
**for** $t = 1, 2, \ldots$
 **if** $(\exists\, i \text{ s.t. } y_i\langle \mathbf{w}^{(t)}, \mathbf{x}_i\rangle \le 0)$ then
  $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + y_i\mathbf{x}_i$
 **else**
  **output** $\mathbf{w}^{(t)}$




## Linear Threshold Unit [2]

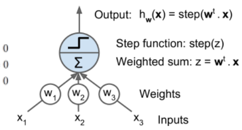
- The LTU computes a *weighted sum* of its inputs:

  $z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \mathbf{w}^T \cdot \mathbf{x}$
- Then applies a *step function* to that sum and outputs the result:

  $h_w(\mathbf{x}) = \text{step}(z) = \text{step}(\mathbf{w}^T \cdot \mathbf{x})$.

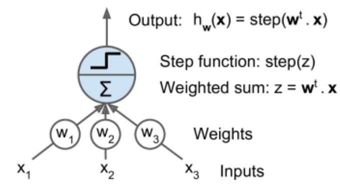
***Common step functions:***

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \qquad \text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$



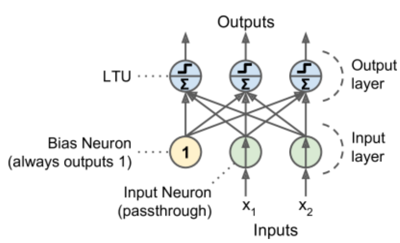

## Training An LTU

- Training an LTU means finding the right values for $w_0$, $w_1$, and $w_2$.
- The training algorithm is discussed shortly.




## Perceptron With Multiple Outputs

- A *Perceptron* is simply composed of a single layer of LTUs.




## How is a Perceptron Trained?

- The Perceptron is fed *one training instance* at a time, and for each instance it makes its predictions.
- For every output neuron that produced a *wrong prediction*, it reinforces the connection weights from the inputs that would have contributed to the correct prediction.
- Perceptron *learning rule* (weight update):

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta\left(\hat{y}_j - y_j\right)x_i$$

  - $w_{i,j}$ is the *connection weight* between the $i^{th}$ input neuron and the $j^{th}$ output neuron.
  - $x_i$ is the $i^{th}$ input value of the *current training instance*.
  - $y_j$ is the *target output* of the $j^{th}$ output neuron for the current training instance.
  - $\eta$ is the *learning rate*.

## The MNIST Dataset

- The *MNIST dataset*, which is a set of 70,000 small images of digits handwritten by high school students and employees of the US Census Bureau.
- Each image is *labeled* with the digit it represents.
- *Scikit-Learn* provides many helper functions to download popular datasets. MNIST is one of them.

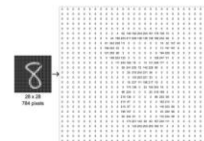
# Downloading The Dataset

```
import numpy as np
from sklearn.datasets import fetch_openml

def sort_by_target(mnist):
    reorder_train = np.array(sorted([(target, i) for i, target in
enumerate(mnist.target[:60000])]))[:, 1]
    reorder_test = np.array(sorted([(target, i) for i, target in
enumerate(mnist.target[60000:])]))[:, 1]
    mnist.data[:60000] = mnist.data[reorder_train]
    mnist.target[:60000] = mnist.target[reorder_train]
    mnist.data[60000:] = mnist.data[reorder_test + 60000]
    mnist.target[60000:] = mnist.target[reorder_test + 60000]

try:
    mnist = fetch_openml('mnist_784', version=1, cache=True)
    mnist.target = mnist.target.astype(np.int8) # fetch_openml() returns targets
as strings
    sort_by_target(mnist) # fetch_openml() returns an unsorted dataset
except ImportError:
    from sklearn.datasets import fetch_mldata
    mnist = fetch_mldata('MNIST original')
```

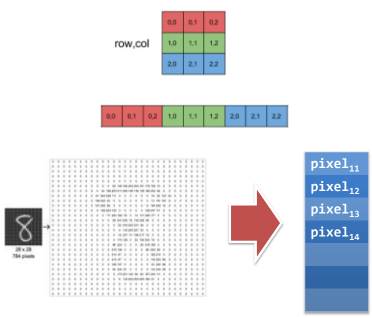
# Loading Data

```
X, y = mnist["data"], mnist["target"]
print('X.shape:', X.shape)
print('y.shape:', y.shape)
```

```
X.shape: (70000, 784)
y.shape: (70000,)
```



# Reshaping Matrix To Vector




# Visualizing The Data

```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
print(y[36000])
some_digit = X[36000]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap =
matplotlib.cm.binary,
interpolation="nearest")
plt.axis("off")
plt.show()
```

# Preparing Training Data

- The MNIST dataset is actually already split into a *training set* (the first 60,000 images) and a *test set* (the last 10,000 images):

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000],
y[60000:]

# Shuffle the training set;
# this will guarantee that all cross-validation folds will be similar
import numpy as np
shuffle_index = np.random.permutation(60000)
X_train, y_train = X_train[shuffle_index], y_train[shuffle_index]
print ('X_train.shape:', X_train.shape)
print ('y_train.shape:', y_train.shape)
print ('X_test.shape:', X_test.shape)
print ('y_test.shape:', y_test.shape)
```

```
X_train.shape: (60000, 784)
y_train.shape: (60000,)
X_test.shape: (10000, 784)
y_test.shape: (10000,)
```

# Normalizing Image Inputs

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled =
scaler.fit_transform(X_train.astype(np.float64))
X_test_scaled = scaler.fit_transform(X_test.astype(np.float64))
print ('X_train_scaled.shape:', X_train_scaled.shape)
print ('X_test_scaled.shape:', X_test_scaled.shape)
```

```
X_train_scaled.shape: (60000, 784)
X_test_scaled.shape: (10000, 784)
```

## Preparing Targets for Number 5

```
# Only try to identify one digit—for example, the number 5
y_train_5 = (y_train == 5) # True for all 5s, False for all
other digits.
y_test_5 = (y_test == 5)
print ('y_train:', y_train)
print ('y_train_5:', y_train_5)
```

```
y_train: [2 1 7 ... 5 9 8]
y_train_5: [False False False ...  True False False]
```

## Training a **Perceptron** for Number 5

```
from sklearn.linear_model import Perceptron
per_clf_5 = Perceptron(max_iter = 1000, tol = 3,
random_state=42)
per_clf_5.fit(X_train_scaled, y_train_5)
```

```
Perceptron(alpha=0.0001, class_weight=None,
early_stopping=False, eta0=1.0,
      fit_intercept=True, max_iter=1000,
n_iter=None, n_iter_no_change=5,
      n_jobs=None, penalty=None, random_state=42,
shuffle=True, tol=3,
      validation_fraction=0.1, verbose=0,
warm_start=False)
```

## Making Prediction

```
# The classifier guesses that this image does not
represents a 5 (Correct).
print ('Prediction:',
per_clf_5.predict([X_test_scaled[4000]]))
```

```
Prediction: [False]
```

## Evaluating The **Perceptron** Model

```
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import precision_score, recall_score
y_train_pred_per = cross_val_predict(per_clf_5, X_train_scaled,
y_train_5, cv=3)
print('Precision:', precision_score(y_train_5,
y_train_pred_per))
print('Recall:', recall_score(y_train_5, y_train_pred_per))
```

```
Precision: 0.7415597569209993
Recall: 0.8103670909426305
```

## Under The Hood

```
from sklearn.linear_model import SGDClassifier
# Stochastic Gradient Descent (SGD) classifier
sgd_clf_5 = SGDClassifier(max_iter = 1000, tol = 3,
random_state = 42, loss="perceptron",
learning_rate="constant", eta0=1, penalty=None)
sgd_clf_5.fit(X_train_scaled, y_train_5)
```

```
SGDClassifier(alpha=0.0001, average=False, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=1, fit_intercept=True,
       l1_ratio=0.15, learning_rate='constant', loss='perceptron',
       max_iter=1000, n_iter=None, n_iter_no_change=5, n_jobs=None,
       penalty=None, power_t=0.5, random_state=42, shuffle=True, tol=3,
       validation_fraction=0.1, verbose=0, warm_start=False)
```

## Evaluating The **SGDClassifier** Model

```
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import precision_score, recall_score
y_train_pred_sgd = cross_val_predict(sgd_clf_5, X_train_scaled,
y_train_5, cv=3)
print('Precision:', precision_score(y_train_5,
y_train_pred_sgd))
print('Recall:', recall_score(y_train_5, y_train_pred_sgd))
```

```
Precision: 0.7415597569209993
Recall: 0.8103670909426305
```

## Perceptron from Scratch (I)

```
import numpy as np

class Perceptron(object):

    def __init__(self, n_features, n_iter=100, learning_rate=0.01):
        self.n_iter = n_iter
        self.learning_rate = learning_rate
        self.weights = np.zeros(n_features + 1)

    def fit(self, X, y):
        unique, counts = np.unique(y, return_counts=True)
        total_pos = counts[1]
        total_neg = counts[0]
        for _iter in range(1, self.n_iter):
            true_pos, true_neg = self.update_weights(X, y)
            self.display_info(_iter, true_pos, true_neg, total_pos,
total_neg)
```

## Perceptron from Scratch (II)

```
    def update_weights(self, X, y):
        true_pos = 0
        true_neg = 0
        for digit, label in zip(X, y):
            predicted = self.predict(digit)
            # Update weights
            self.weights[1:] += self.learning_rate * (label - predicted) * digit
            self.weights[0] += self.learning_rate * (label - predicted)
            if (predicted==label) and (1 == label):
                true_pos += 1
            if (predicted==label) and (0 == label):
                true_neg += 1
        return true_pos, true_neg
```

```
    def predict(self, digit):
        # Compute activation for the digit
        weighted_sum = np.dot(digit, self.weights[1:]) + self.weights[0]
        if weighted_sum > 0:
            activation = 1
        else:
            activation = 0
        return activation
```

## Perceptron from Scratch (III)

```
    def display_info(self, _iter, true_pos, true_neg, total_pos, total_neg):
        if 0 == _iter % 10:
            print ('Iteration:', _iter)
            print ('weights[0]:', self.weights[0])
            print ('true_pos/total_pos:', true_pos*100/total_pos)
            print ('true_neg/total_neg:', true_neg*100/total_neg)

np_per_clf = Perceptron(784)
np_per_clf.fit(X_train_scaled, y_train_5)
print ('Prediction:', np_per_clf.predict([some_digit]))
```

```
Iteration: 10
weights[0]: -11.729999999999794
true_pos/total_pos: 78.56484043534404
true_neg/total_neg: 97.85998277725865
Iteration: 20
weights[0]: -12.209999999999784
true_pos/total_pos: 78.93377605607822
true_neg/total_neg: 97.89662690778505
```

## Thank You for Your Time