

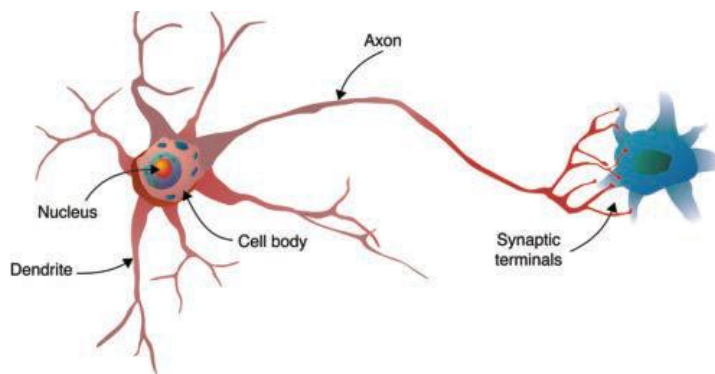
Unit – I

Perceptron and Gradient Based Learning

Perceptron: Two-Input Perceptron - The Perceptron Learning Algorithm -Limitations of the Perceptron -Combining Multiple Perceptrons - Bias Term. **Gradient-Based Learning:** Intuitive Explanation of the Perceptron Learning Algorithm - Derivatives and Optimization Problems - Solving a Learning Problem with Gradient Descent - Constants and Variables in a Network - Analytic Explanation of the Perceptron Learning Algorithm - Perceptron to Identify Patterns.

The Rosenblatt Perceptron

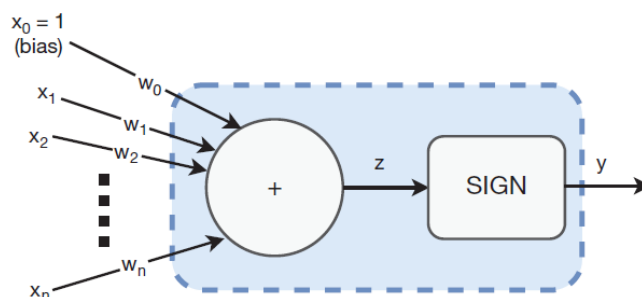
- The perceptron is an artificial neuron, that is, a model of a biological neuron.
- A biological neuron consists of one cell body, multiple dendrites, and a single axon.
- The connections between neurons are known as synapses.
- The neuron receives stimuli on the dendrites, and in cases of sufficient stimuli, the neuron fires (also known as getting activated or excited) and outputs stimulus on its axon, which is transmitted to other neurons that have synaptic connections to the excited neuron.
- Synaptic signals can be excitatory or inhibitory; that is, some signals can prevent a neuron from firing instead of causing it to fire.



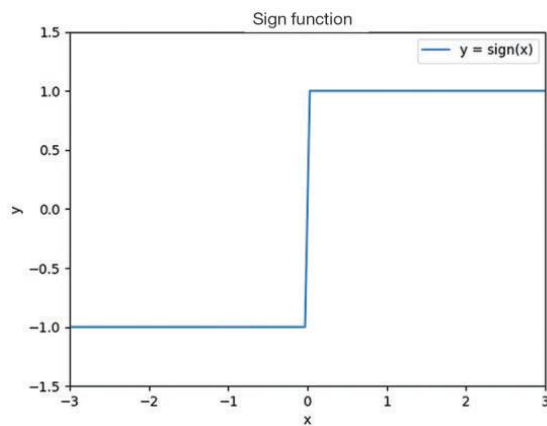
A biological neuron

Perceptron

- A **perceptron** is a type of **artificial neuron**. The perceptron consists of a computational unit, a number of inputs (one of which is a special *bias input*, which is detailed later in this chapter), each with an associated input weight and a single output.
- It sums up the inputs to compute an intermediate value z , which is fed to an **activation function**. The perceptron uses the **sign function** as an activation function, but **other artificial neurons use other functions**.



- The inputs are represented as x_0, x_1, \dots, x_n with n general inputs (x_0 being the bias input), and the output is named y .
- The inputs and output loosely correspond to the dendrites and the axon. Each input has an associated weight (w_i , where $i = 0, \dots, n$), which has been referred to as synaptic weight or input weight or weight.
- For the perceptron, the output can take on only one of two values, -1 or 1 . (Note: this constraint will be relaxed to a range of real values for other types of artificial neurons)
- The bias input is always 1 . Each input value is multiplied by its corresponding weight before it is presented to the computational unit which loosely corresponds to the cell body of a biological neuron.
- The computational unit computes the sum of the weighted inputs and then applies a so-called activation function, $y = f(z)$, where z is the sum of the weighted inputs.
- The activation function for a perceptron is the sign function, also known as the signum function, which evaluates to 1 if the input is 0 or higher and -1 otherwise.
- The sign function is shown below.



Sign (or signum) function

(Note: The figure uses variable names typically used for generic functions (y is a function of x). In our perceptron use case, the input to the signum function is not x but the weighted sum z .)

The perceptron will output -1 if the weighted sum is less than zero, and otherwise it will output 1 . Written as an equation,

$$y = f(z), \text{ where}$$

$$z = \sum_{i=0}^n w_i x_i$$

$$f(z) = \begin{cases} -1, & z < 0 \\ 1, & z \geq 0 \end{cases}$$

$$x_0 = 1 \text{ (bias term)}$$

Python Implementation of Perceptron Function

First element in vector x must be 1 .

Length of w and x must be n+1 for neuron with n inputs.

```
def compute_output(w, x):
    z = 0.0
    for i in range(len(w)):
        z += x[i] * w[i] # Compute sum of weighted inputs
    if z < 0: # Apply sign function
        return -1
    else:
        return 1
```

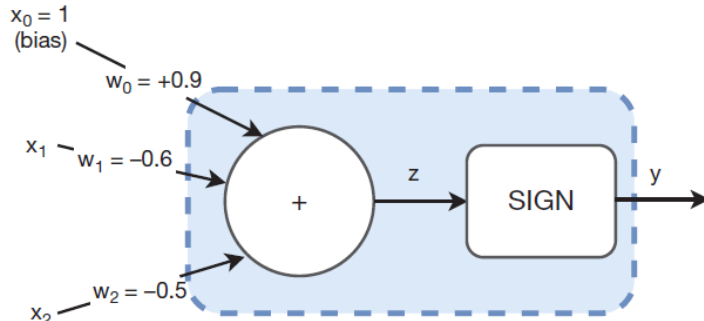
The first element of x represents the bias term and that must be set to 1 by the caller of the function.

A Two-Input Perceptron

Let's examine a perceptron with two inputs and a bias input.

The weights are set as: $w_0=0.9$ (bias), $w_1=-0.6$, $w_2=-0.5$.

Let us see how this perceptron behaves for all input combinations assuming that each of the two inputs can take on only the values -1.0 and 1.0.



```
>>> compute_output([0.9, -0.6, -0.5], [1.0, -1.0, -1.0])
```

```
1
```

```
>>> compute_output([0.9, -0.6, -0.5], [1.0, -1.0, 1.0])
```

```
1
```

```
>>> compute_output([0.9, -0.6, -0.5], [1.0, 1.0, -1.0])
```

```
1
```

```
>>> compute_output([0.9, -0.6, -0.5], [1.0, 1.0, 1.0])
```

```
-1
```

The Perceptron Learning Algorithm

The perceptron learning algorithm is what is called a *supervised learning algorithm*. The notion of supervision implies that the *model* that is being trained (in this case, the perceptron) is presented with both the input data and the desired output data (also known as *ground truth*).

The algorithm works as follows:

1. Randomly initialize the weights.
2. Select one input/output pair at random.
3. Present the values x_1, \dots, x_n to the perceptron to compute the output y .
4. If the output y is different from the ground truth for this input/output pair, adjust the weights in the following way:
 - a. If $y < 0$, add ηx_i to each w_i .
 - b. If $y > 0$, subtract ηx_i from each w_i .
5. Repeat steps 2, 3, and 4 until the perceptron predicts all examples correctly.

The perceptron has certain limitations to what it can predict, so for some sets of input/output pairs, the algorithm will not converge. However, if it is possible to come up with a set of weights that enables the perceptron to represent the set of input/output pairs, then the algorithm is guaranteed to converge by finding these weights.

The learning rate is an example of a *hyperparameter*, which is not a parameter that is adjusted by the learning algorithm. The arbitrary constant η is known as the *learning rate* and can be set to 1.0, but setting it to a different value can lead to faster convergence of the algorithm.

For a perceptron, weights can be initialized to zero; however, for more complex neural networks, this approach is not effective. As a result, weights are typically initialized randomly to establish a better practice.

```
# Perceptron training loop
w = [0.2, -0.6, 0.25] # Initial weights
x_train = [[1, 1, -1], [1, -1, -1], [1, 1, 1], [1, -1, 1]] # Training inputs
y_train = [1, -1, 1, -1] # Expected outputs
LEARNING_RATE = 0.1
all_correct = False

while not all_correct:
    all_correct = True
    index_list = [0, 1, 2, 3]
    import random
```

```

random.shuffle(index_list) # Randomize order of examples

for i in index_list:
    x = x_train[i]
    y = y_train[i]

    # Compute perceptron output
    weighted_sum = 0
    for j in range(len(w)):
        weighted_sum += w[j] * x[j] # Calculate the weighted sum

    if weighted_sum > 0: # Apply the step function
        p_out = 1
    else:
        p_out = -1

    # Check if prediction is wrong
    if y != p_out:
        for j in range(len(w)):
            w[j] += (y * LEARNING_RATE * x[j]) # Update weights
        all_correct = False # Mark that the output wasn't correct
        print(f"w0 = {w[0]:.2f}, w1 = {w[1]:.2f}, w2 = {w[2]:.2f}") # Show updated
weights

```

The perceptron training loop. It is a nested loop in which the inner loop runs through all four training examples in random order. For each example, it computes the output and adjusts and prints the weights if the output is wrong. The weight adjustment line contains a subtle detail that makes it look slightly different.

The output will be something like the following:

w0 = 0.20 , w1 = -0.60 , w2 = 0.25

w0 = 0.30 , w1 = -0.50 , w2 = 0.15

w0 = 0.40 , w1 = -0.40 , w2 = 0.05

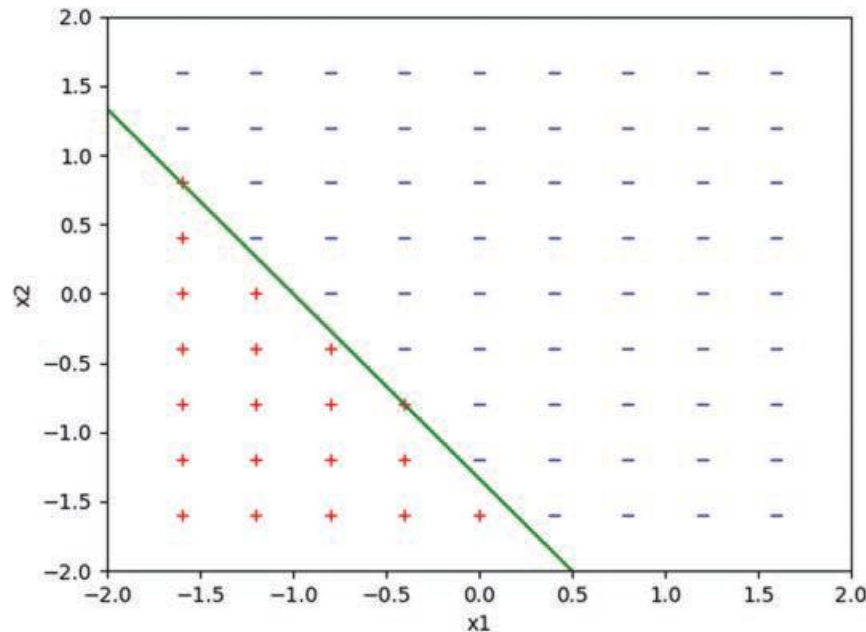
w0 = 0.30 , w1 = -0.50 , w2 = -0.05

$$w_0 = 0.40, w_1 = -0.40, w_2 = -0.15$$

Note: the weights are gradually adjusted from the initial values to arrive at weights that produce the correct output.

We have restricted making each input take on just one of two values (either -1 or 1). One way to illustrate this is to make a chart of a 2D coordinate system where one axis represents the first input (x_1), and the other axis represents the second input (x_2).

For each point in this coordinate system, we can write a “+” or a “-” depending on what value the perceptron outputs.



Output of a perceptron as a function of two inputs x_1 and x_2

The perceptron divides the 2D space into two regions, separated by a straight line, where all input values on one side of the line produce the output -1 , and all input values on the other side of the line produce the output $+1$.

The boundary is exactly where the weighted sum of the inputs is zero, because the sign function will change its value when its input is zero. That is, we have

$$w_0x_0 + w_1x_1 + w_2x_2 = 0$$

x_2 is a function of x_1 , because x_2 is plotted on the y-axis, and normally when plotting a straight line, we do $y = f(x)$. We insert 1 for x_0 , solve the equation for x_2 ,

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{w_0}{w_2}$$

It is a straight line with slope $-w_1/w_2$ and a y-intercept of $-w_0/w_2$.

Limitations of the Perceptron

The two-input perceptron learns how to draw a straight line between two groups of data points.

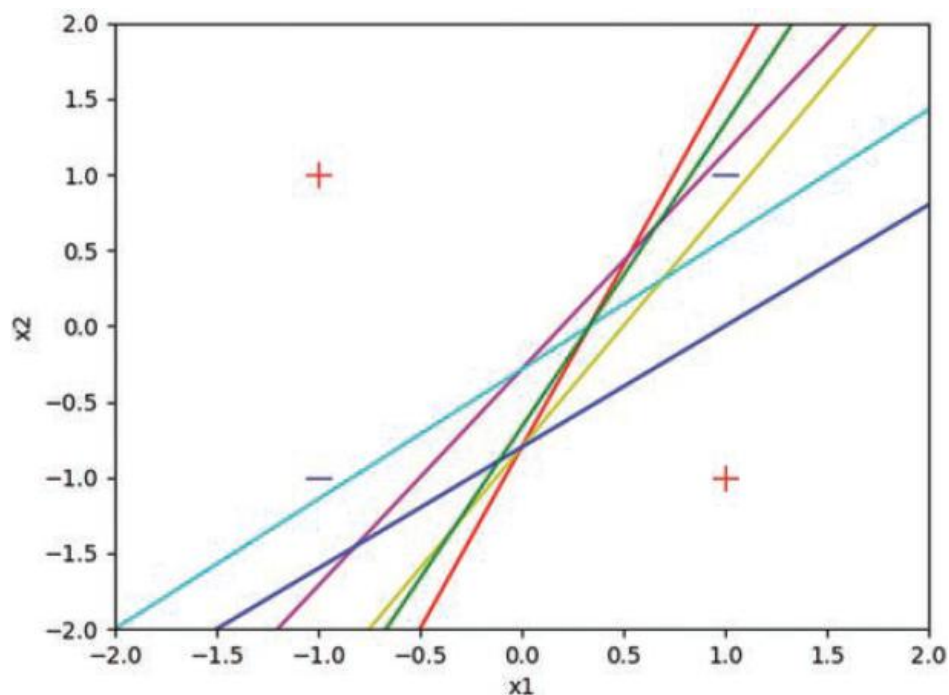
But what happens if a straight line cannot separate the data points? To explore this scenario, a different Boolean function called exclusive OR (XOR) is used.

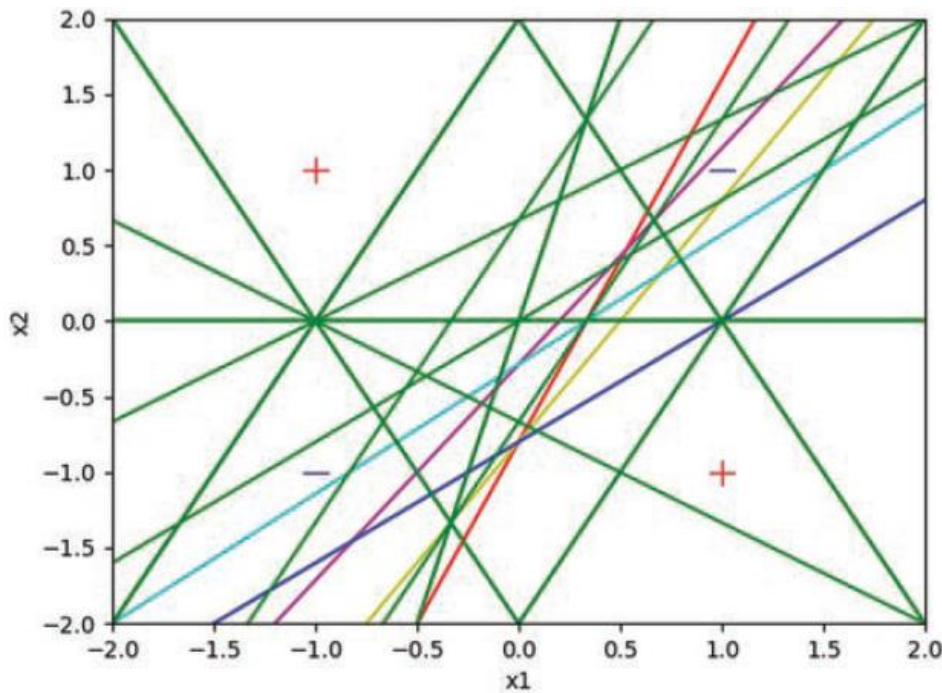
Truth Table for a Two-Input XOR Gate

X_0	X_1	Y
False	False	False
True	False	True
False	True	True
True	True	False

The below figure shows these four data points on the same type of chart, illustrating how the algorithm tries to learn how to draw a line between the plus and minus signs.

The top chart shows what it looks like after 6 weight updates and the bottom chart, after 30 weight updates.





Perceptron attempting to learn XOR. Top: After 6 weight adjustments.

Bottom: After 30 weight adjustments.

It is trivial to solve the problem with a curved line but is not possible with a straight line. This is one of the key limitations of the perceptron.

It can solve classification problems only where the classes are linearly separable, which in two dimensions (two inputs) means that the data points can be separated by a straight line.

Thus, we need a different model of a neuron or combine multiple of them to solve the problem.

Combining Multiple Perceptrons

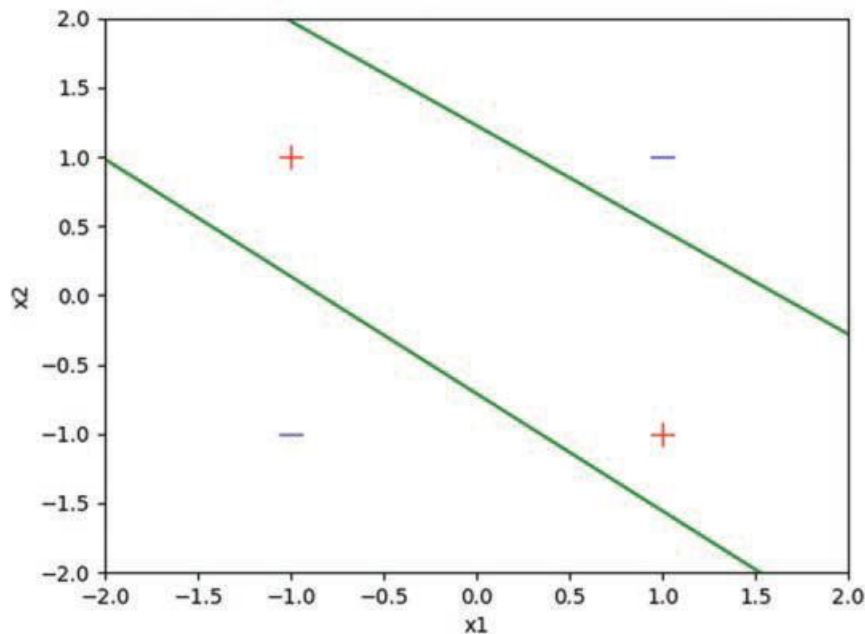
A single perceptron can separate the chart into two regions, illustrated by drawing a straight line on the chart. That means that if we add another perceptron, we can draw another straight line.

Figure below shows one such attempt: One line separates one of the minuses from all other data points. Similarly, the other line separates the other minus also from all other data points.

The challenge is to correctly classify the points between these two lines.

To do this, we need to output **1** only for the data points between the lines, which represents the desired XOR function output. This is achieved by combining the outputs of the first two perceptrons using a third perceptron that performs an **AND** operation. The third

perceptron takes the outputs of the first two as inputs and outputs **1** only when both perceptrons produce **1**, effectively solving the XOR problem.

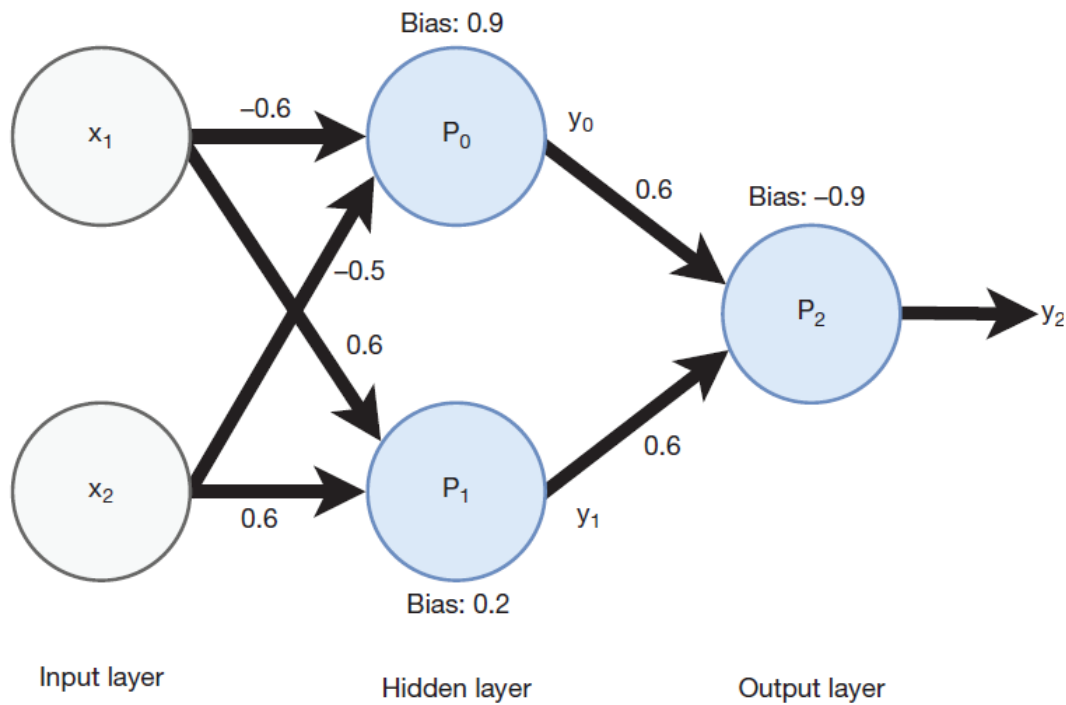


XOR output values isolated by two lines

Another way to look at it is that each of the two perceptrons will fire correctly for three out of four data points; that is, both of them almost do the right thing. They both incorrectly categorize one data point.

If we could combine the output of the two, and output 1 only when both of them compute the output as a 1, then we would get the right result. So, we want to do an AND of their outputs.

We just add another perceptron that uses the outputs of the two previous perceptrons as its inputs.



Two-level feedforward network implementing XOR function

This neural network is one of the simplest examples of a fully connected feedforward network. *Fully connected* means that the output of each neuron in one layer is connected to all neurons in the next layer.

Feedforward means that there are no backward connections, or, using graph terminology, it is a directed acyclic graph (DAG).

A multilevel neural network has an input layer, one or more hidden layers, and an output layer. The input layer does not contain neurons but contains only the inputs themselves; that is, there are no weights associated with the input layer.

A feedforward network is also known as a *multilevel perceptron* even when it is built from neuron models that are not perceptrons.

In a **fully connected network**, a neuron in one layer receives inputs from all other neurons in the immediately preceding layer. A **feedforward network** or **multilevel perceptron** has no cycles. The **input layer** has no neurons. The outputs of neurons in a **hidden layer** are not visible (they are hidden) outside of the network. DNNs have **multiple hidden layers**. The **output layer** can have multiple neurons.

Input and Output Values Showing That the Network Implements the XOR Function

X_0	X_1	X_2	Y_0	Y_1	Y_2
1	-1 (False)	-1 (False)	1.0	-1.0	-1.0 (False)
1	1 (True)	-1 (False)	1.0	1.0	1.0 (True)
1	-1 (False)	1 (True)	1.0	1.0	1.0 (True)
1	1 (True)	1 (True)	-1.0	1.0	-1.0 (False)

Bias Term

The perceptron includes the use of a bias term.

$y=f(z)$, where

$$z = \sum_{i=0}^n w_i x_i$$

$$f(z) = \begin{cases} -1, & z < 0 \\ 1, & z \geq 0 \end{cases}$$

$$x_0 = 1 \text{ (bias term)}$$

The output values of -1 and 1 and the threshold of 0 seem like they have been chosen somewhat arbitrarily.

The threshold, which is often denoted by θ (Greek letter theta). We could make our perceptron more general by replacing the activation function with the following, where the threshold is simply a parameter θ :

$$f(z) = \begin{cases} -1, & z < \theta \\ 1, & z \geq \theta \end{cases}$$

For the output to take on the value 1 , $z \geq \theta$.

This can be rewritten as

$$Z - \theta \geq 0$$

As we subtract the threshold from z , we can keep our implementation that uses 0 as its threshold.

The rationale for the bias term in the first place was to make the perceptron implement an adjustable threshold value.

The associated weight w_0 , can be both positive and negative. By adjusting w_0 , the perceptron can be made to implement any arbitrary threshold θ . If we want the perceptron to use a threshold of θ , then we set w_0 to be $-\theta$.

If we focus only on the weighted sum z and ignore the activation function, the bias term acts like the intercept b in a straight-line equation:

$$y = mx + b$$

Gradient-Based Learning

Intuitive Explanation of the Perceptron Learning Algorithm

Weight Update Step of Perceptron Learning Algorithm

for i in range(len(w)):

w[i] += (y * LEARNING_RATE * x[i])

w - array representing the weight vector

x - array representing the input vector

y - desired output

If an example is presented to the perceptron and the perceptron **correctly predicts** the output, we **do not adjust any weights**.

When the perceptron predicts the **outputs incorrectly**, we need to **adjust the weights**.

The weight adjustment is computed by combining the desired y value, the input value, and a parameter known as **LEARNING_RATE**.

Let us consider three different training examples where x_0 represents the bias input that is always 1:

Training example 1: $x_0 = 1, x_1 = 0, x_2 = 0, y = 1$

Training example 2: $x_0 = 1, x_1 = 0, x_2 = 1.5, y = -1$

Training example 3: $x_0 = 1, x_1 = -1.5, x_2 = 0, y = 1$

For training example 1,

The z-value (the input to the signum function) for the perceptron is computed as

$$z = w_0x_0 + w_1x_1 + w_2x_2$$

For training example 1, the result is

$$z = w_0 \cdot 1 + w_1 \cdot 0 + w_2 \cdot 0 = w_0$$

w_1 and w_2 do not affect the result, so the only weight that can be adjusted is w_0 .

If the desired output value is positive ($y = 1$), then we want to increase the value of w_0 . If the desired output value is negative ($y = -1$), then we want to decrease the value of w_0 .

For training example 2,

$$z = w_0 \cdot 1 + w_1 \cdot 0 + w_2 \cdot 1.5 = w_0 + 1.5w_2$$

Both w_0 and w_2 are adjusted negatively because $y = -1$. The magnitude of adjustment for w_2 is greater than w_0 because $x_2 = 1.5$.

For training example 3,

$$z = w_0 \cdot 1 + w_1 \cdot -1.5 + w_2 \cdot 0 = \mathbf{w_0 - 1.5w_1}$$

w_0 is adjusted **positively**. w_1 is adjusted **negatively** because $x_1 = -1.5$.

The weight adjustment for each training example is computed using a learning rate of **0.1**.

Adjustment Values for Each Weight for the Three Training Examples

	<i>w_0 Change</i>	<i>w_1 Change</i>	<i>w_2 Change</i>
Example 1	$1 * 1 * 0.1 = 0.1$	$1 * 0 * 0.1 = 0$	$1 * 0 * 0.1 = 0$
Example 2	$(-1) * 1 * 0.1 = -0.1$	$(-1) * 0 * 0.1 = 0$	$(-1) * 1.5 * 0.1 = -0.15$
Example 3	$1 * 1 * 0.1 = 0.1$	$1 * (-1.5) * 0.1 = -0.15$	$1 * 0 * 0.1 = 0$

The adjustment of the bias weight depends only on the desired output value and will be determined by whether the majority of the training examples have positive or negative desired outputs.

For a given training example, only the weights that affect the output will see a significant adjustment, because the adjustments are proportional to the input values. When an input value is 0 for a training example, its corresponding weight will see zero adjustment.

This makes much sense, **if more than 50%** of the training examples have the same output, adjusting the bias toward that output helps the perceptron perform better.

Derivatives and Optimization Problems

Given a function

$$Y = f(x)$$

The derivative of y with respect to x tells how much the value of y changes given a small change in x . A few common notations are:

$$y', f'(x), \frac{dy}{dx}$$

The below figure plots the value of an arbitrary function $y = f(x)$.

Given an initial value x and its correspondingly, the sign of the derivative indicates in what direction to adjust x to reduce the value of y . Similarly, if we know how to solve x for 0, we will find an extreme point (minimum, maximum, or saddle point) of y .

We can compute two partial derivatives:

$$\frac{\partial y}{\partial x_0} \text{ and } \frac{\partial y}{\partial x_1}$$

A partial derivative is just like a normal derivative, when computing a partial derivative with respect to one variable, we **treat all other variables as constants** and differentiate only with respect to the only variable that is not treated as a constant.

If we have the function $y = ax_0 + bx_1$ our two partial derivatives become

$$\frac{\partial y}{\partial x_0} = a$$

$$\frac{\partial y}{\partial x_1} = b$$

If we arrange these partial derivatives in a vector, we get:

$$\nabla y = \begin{pmatrix} \frac{\partial y}{\partial x_0} \\ \frac{\partial y}{\partial x_1} \end{pmatrix}$$

which is called the *gradient* of the function - that is, the gradient is an extension of the derivative for functions with multiple variables.

Solving a Learning Problem with Gradient Descent

The goal of learning in a neural network is to **identify the weights** so that the network's predicted output (\hat{y}) matches the actual desired output (y).

Mathematically, this is the same as solving the following equation:

$$y - \hat{y} = 0$$

In reality, we do not have just a single training example (data point), but we have a set of training examples.

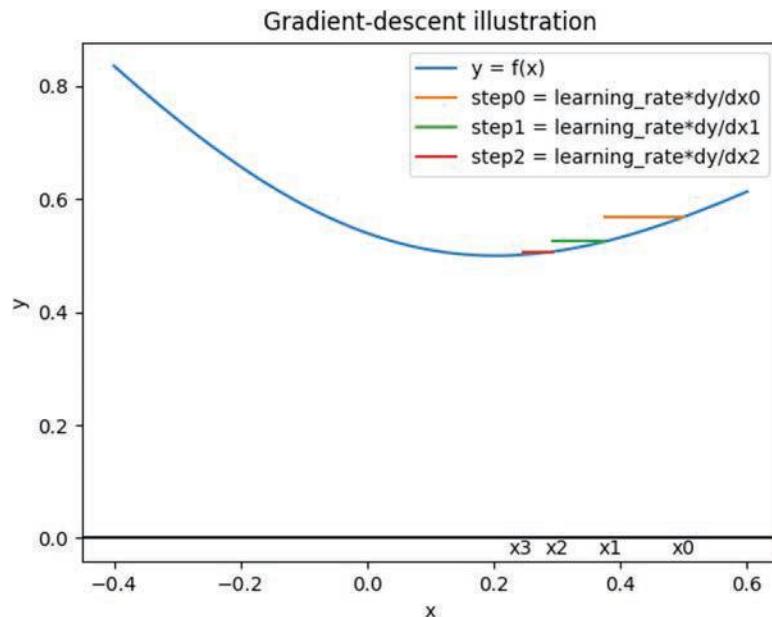
We can combine these multiple training examples into a single error metric by computing their mean squared error (MSE):

$$\frac{1}{m} \sum_{i=1}^m (y^{(i)} - \hat{y}^{(i)})^2$$

MSE measures how far predictions are from actual values. It is **strictly greater than 0**, so we **cannot solve for MSE = 0** directly.

Instead, we will treat our problem as an optimization problem, in which we try to find weights that *minimize the value* of the error function.

In most deep learning (DL) problems, it is not feasible to find a closed form solution to this minimization problem. Instead, a numerical method known as **gradient descent** is used. It is an iterative method in which we start with an initial guess of the solution and then gradually refine it.



We start with an initial guess x_0 . We can insert this value into $f(x)$ and compute the corresponding y .

The sign of the derivative indicates whether we should increase or decrease x_0 . If the slope is **positive**, decrease x . If the slope is **negative**, increase x . We can iteratively refine the solution by repeatedly doing small adjustments to x .

Gradient descent uses this property to decide how much to adjust x . The update formula:

$$x_{n+1} = x_n - \eta f'(x_n)$$

The step size depends on both the learning rate and the derivative, so the step size will decrease as the derivative decreases. As the algorithm converges at the minimum point, (i.e. the derivative approaches 0) implies that the step size also approaches 0.

If the learning rate is too large, gradient descent might overshoot the solution and fail to converge, and if it's too small, it may take too long to converge.

Gradient Descent For Multidimensional Functions

- ✚ Neural networks are functions of many variables, so we need the ability to minimize multidimensional functions.
- ✚ A gradient is a vector consisting of partial derivatives and indicates the direction in the input space that results in the steepest ascent for the function value.
- ✚ The negative gradient is the direction of steepest descent, or the direction of the quickest path to reducing the function value.
- ✚ If the current point $\mathbf{x} = (\mathbf{x}_0, \mathbf{x}_1)$ and the goal is to minimize \mathbf{y} , the next point is chosen as

$$\begin{pmatrix} x_0 \\ x_1 \end{pmatrix} - \eta \nabla y$$

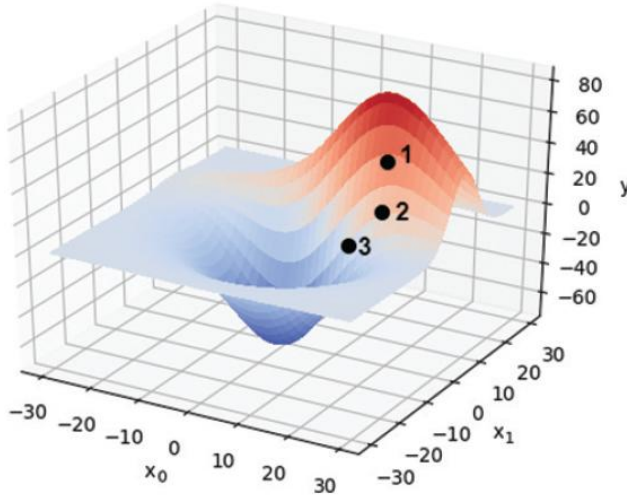
- ✚ where $\nabla \mathbf{y}$ is the gradient. For a function of n variables, the gradient consists of n partial derivatives, and the next step is computed as

$$\mathbf{x} - \eta \nabla y$$

where both \mathbf{x} and ∇y are vectors consisting of n elements.

- Below figure shows gradient descent for a function of two input variables. The function value y gradually decreases as we move from point 1 to point 2 and 3.

$$y = f(x_0, x_1)$$



Gradient descent for a function of two variables

Constants and Variables in a Network

When applying gradient descent to our neural network, we consider input values (\mathbf{x}) to be constants, with the goal to adjust the weights (\mathbf{w}), including the bias input weight (w_0).

During learning, not the inputs (\mathbf{x}) but the weights (\mathbf{w}) are considered to be the variables in our function.

At first sight, for the two-input perceptron, it seems like x_1 and x_2 would be considered input values that minimize a function. But the goal is different: we want to adjust the weights so the network produces the correct output for a given input.

The purpose of our learning algorithm is to adjust the weights (w_0, w_1, w_2) for a given fixed input (x_1, x_2) so that the output matches the desired value.

That is, we treat x_1 and x_2 as constants (x_0 as well, but that is always the constant 1, as stated earlier), while we treat w_0, w_1 , and w_2 as variables that we can adjust.

Example:

If we are training a network to distinguish between a dog and a cat, the input values are the pixel values of the image. If the network misclassifies a dog as a cat, we don't change the image to make it look more like a cat. Instead, we adjust the network's weights to improve its ability to correctly classify images in the future.

Analytic Explanation of the Perceptron Learning Algorithm

The perceptron learning algorithm is based on gradient descent.

Starting with the two-input perceptron, we have the following variables:

$$\mathbf{w} = \begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}, \quad y$$

The weight vector \mathbf{w} is initialized with arbitrary values.

The input vector, $\mathbf{x} = (x_0, x_1, x_2)$, where $x_0 = 1$ and the target output (ground truth) is y .

Let us first consider the case where the current weights result in an output of +1 but the ground truth is -1. This means that the z -value is positive and we want to drive it down toward (and below) 0.

We can do this by applying gradient descent to the following function:

$$z = w_0x_0 + w_1x_1 + w_2x_2$$

where x_0, x_1, x_2 are constants and the weights are treated as variables. First, we need to compute the gradient, which consists of the three partial derivatives with respect to w_0, w_1 and w_2 .

When computing a partial derivative, we treat all the variables except the one that we are taking the derivative are constants, so the gradient calculation will be:

$$\nabla z = \begin{pmatrix} \frac{\partial z}{\partial w_0} \\ \frac{\partial z}{\partial w_1} \\ \frac{\partial z}{\partial w_2} \end{pmatrix} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

To update the weight vector \mathbf{w} using gradient descent, we adjust it in the opposite direction of the gradient to reduce the error:

$$\mathbf{w} = \mathbf{w} - \eta \nabla z$$

Expanding this for each weight component:

$$\begin{pmatrix} w_0 - \eta x_0 \\ w_1 - \eta x_1 \\ w_2 - \eta x_2 \end{pmatrix}$$

This is the update rule for the perceptron learning algorithm. That is, the perceptron learning algorithm is equivalent to applying gradient descent to the perceptron function.

If the perceptron predicts -1 but the true label is +1, we multiply all terms by -1 to make it a minimization problem.

The only difference will be that the gradient will have a different sign, which again makes gradient descent equivalent to the perceptron learning algorithm.

The above process is stochastic gradient descent (SGD).

Gradient descent is an optimization algorithm used to train neural networks by minimizing the loss function, which measures the difference between the predicted output and the actual target values. It adjusts the model's weights and biases iteratively to reduce this error and improve the model's performance.

Types of Gradient Descent:

1. **Batch Gradient Descent:**
 - Updates weights using the entire dataset in one iteration.
 - Computationally expensive for large datasets.
2. **Stochastic Gradient Descent (SGD):**
 - Updates weights using a single sample at a time.
 - Faster but noisier convergence.
3. **Mini-Batch Gradient Descent:**
 - Updates weights using a small batch of samples.
 - Balances the efficiency of batch GD and the speed of SGD.

Here's a differentiation of **Batch Gradient Descent**, **Stochastic Gradient Descent (SGD)**, and **Mini-Batch Gradient Descent** with a simple example:

Example Scenario

Suppose we are training a neural network to predict house prices using a dataset of **1000 samples**. The model has two parameters: **weights (ww)** and **bias (bb)**.

1. Batch Gradient Descent

- **How it works:**
 - Uses the entire dataset (all 1000 samples) to compute gradients and update the parameters in one step.
- **Pros:** Stable convergence.
- **Cons:** Slow and memory-intensive for large datasets.

2. Stochastic Gradient Descent (SGD)

- **How it works:**
 - Uses only one randomly selected sample at each iteration to compute gradients and update the parameters.
- **Pros:** Faster updates; can escape local minima.
- **Cons:** Noisy convergence.

3. Mini-Batch Gradient Descent

- **How it works:**
 - Uses a small batch of samples (e.g., 32, 64) at each iteration to compute gradients.

- **Pros:** Balances speed and convergence stability.
- **Cons:** Still requires moderate memory.

Summary Table

Algorithm	Data Used	Update Frequency	Convergence Speed	Stability
Batch GD	Entire dataset	Once per epoch	Slow	Stable
Stochastic GD (SGD)	Single sample	After every sample	Fast	Noisy
Mini-Batch GD	Small batch	After each batch	Moderate	Balanced

Perceptron to Identify Patterns

This is an example of binary classification. An important case for this is to use the perceptron to identify a specific pattern.

We use the perceptron to classify inputs as belonging to a specific class of interest or not belonging to that class.

Example:

A perceptron could act as a **cat identifier**. If we present an image of a cat to the perceptron, it will fire, but if we present any other image to the perceptron, it will not fire. If the perceptron does not fire, the only thing we know is that the image was not of a cat.

The limitation of a single perceptron is it is not possible to build a good cat identifier from a single perceptron. We need to combine multiple perceptrons, as we did to solve the XOR problem.

Let us consider just a single perceptron and use it to identify some simpler image patterns.

We analyze a small part of a larger image. We arbitrarily choose to look at 9 pixels that are arranged in a 3 x 3 grid. We assume that a pixel can take one of three intensities:

White (1.0), gray (0.0), or black (-1.0)

This is done to limit the number of training examples. A training example will only consist of combinations of black and white pixels or black and gray pixels.

There are no training examples with a combination of gray and white pixels or with black, gray, and white pixels. That is, we have $2^9 = 512$ examples that are black and white and $2^9 = 512$ examples that are black and gray.

These two sets overlap at only one place: each contains an image of all-black pixels. So, to be precise, we end up with 1,023 unique training examples.

The perceptron's task is to output **+1** for a specific target pattern and **-1** for all others.

To illustrate this, we trained five perceptrons. Each perceptron was trained to identify a specific pattern.

The training process consisted of repeatedly presenting all input patterns in random order to the perceptron.

We used a ground truth of +1 for the pattern that we wanted the perceptron to learn to identify and -1 for all other examples.

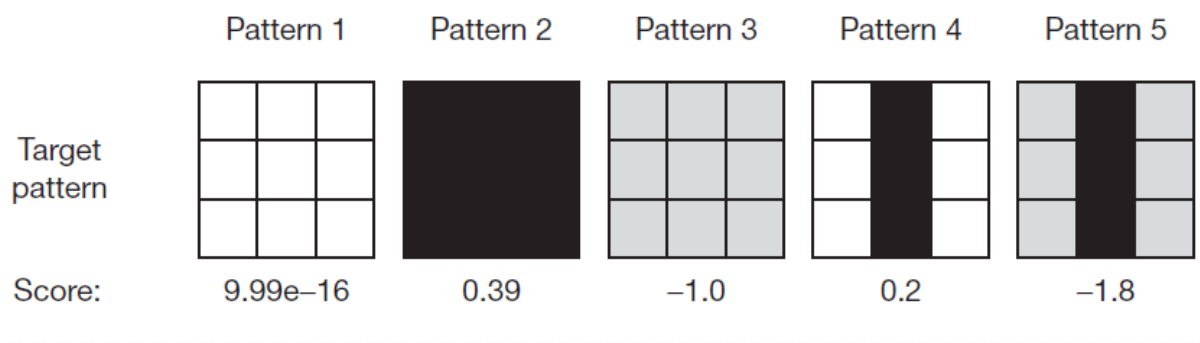
The five columns correspond to the five different perceptrons.

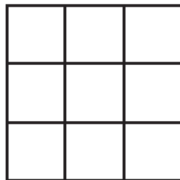


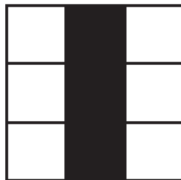


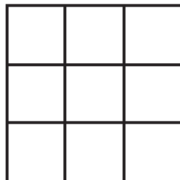



The top row in the figure shows the pattern that we wanted each perceptron to identify. The score under each pattern is the input value to the sign function in the perceptron after it had been trained (i.e., the weighted sum of the inputs when we present the target pattern to the trained perceptron).

The second row shows the highest scoring pattern (we simply presented all combinations to the trained neuron and recorded which patterns resulted in the highest score) and its score. If the highest-scoring pattern is identical to the target pattern, then our classifier is successful at identifying the target pattern. This does not necessarily mean that it does not make any mistakes. For example, a perceptron could output a high score on all patterns, in which case it signals many false positives.

The third row shows the lowest-scoring pattern and its score.

The bottom row shows the weights after training, including the bias weight.



Highest scoring pattern																																																		
Score:	9.99e-16	0.39	-0.2	0.2	-1.2																																													
<hr/>																																																		
Lowest scoring pattern																																																		
Score:	-5.2	-8.8	-1.8	-7.0	-2.8																																													
<hr/>																																																		
Weigh matrix	<table><tr><td>0.6</td><td>0.2</td><td>0.2</td></tr><tr><td>0.2</td><td>0.2</td><td>0.2</td></tr><tr><td>0.6</td><td>0.2</td><td>0.2</td></tr></table>	0.6	0.2	0.2	0.2	0.2	0.2	0.6	0.2	0.2	<table><tr><td>-0.4</td><td>-0.6</td><td>-0.4</td></tr><tr><td>-1</td><td>-0.6</td><td>-0.4</td></tr><tr><td>-0.4</td><td>-0.4</td><td>-0.4</td></tr></table>	-0.4	-0.6	-0.4	-1	-0.6	-0.4	-0.4	-0.4	-0.4	<table><tr><td>-0.2</td><td>0</td><td>0</td></tr><tr><td>0.2</td><td>0</td><td>0</td></tr><tr><td>0</td><td>-0.2</td><td>0.2</td></tr></table>	-0.2	0	0	0.2	0	0	0	-0.2	0.2	<table><tr><td>0.2</td><td>-0.2</td><td>0.8</td></tr><tr><td>0.8</td><td>-0.2</td><td>0.4</td></tr><tr><td>0.4</td><td>-0.4</td><td>0.2</td></tr></table>	0.2	-0.2	0.8	0.8	-0.2	0.4	0.4	-0.4	0.2	<table><tr><td>0.2</td><td>0</td><td>0</td></tr><tr><td>0.2</td><td>-0.2</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0.2</td></tr></table>	0.2	0	0	0.2	-0.2	0	0	0	0.2
0.6	0.2	0.2																																																
0.2	0.2	0.2																																																
0.6	0.2	0.2																																																
-0.4	-0.6	-0.4																																																
-1	-0.6	-0.4																																																
-0.4	-0.4	-0.4																																																
-0.2	0	0																																																
0.2	0	0																																																
0	-0.2	0.2																																																
0.2	-0.2	0.8																																																
0.8	-0.2	0.4																																																
0.4	-0.4	0.2																																																
0.2	0	0																																																
0.2	-0.2	0																																																
0	0	0.2																																																
Bias weight:	-2.6	-4.2	-1.0	-3.4	-2.0																																													

Five example patterns, the resulting weights, and highest- and lowest scoring patterns

The observations are:

- The perceptron is successful at identifying the black-and-white patterns (1, 2, and 4). Because the weights of the perceptron resemble the structure of the input pattern. If a pixel is white (1.0), we want to multiply it by a positive value and if a pixel is black (-1.0), we want to multiply it by a negative value to end up with a high score.
- For all-white case, the target example brings the score above 0. If an example has many—but not all—white pixels, its score will remain below 0. As a result, the perceptron limits false positives, ensuring that only fully white patterns (or very close to it) are classified as positive while others do not.
- Even if the input pattern is perfectly symmetric (e.g., all white or all black), the resulting weight pattern can be asymmetric. This is a result of the random initialization of the weights and the algorithm selects one out of many solutions.
- The perceptron is not successful in perfectly identifying cases in which some of the pixels are gray. In these cases, the algorithm never converged, and we stopped after a fixed number of iterations. These are not linearly separable from the other examples. To identify such patterns, we will need to combine multiple neurons as we did for the XOR problem.