Building a Perceptron from Scratch: A Step-by-Step Guide with Python

In this post, I will show how to implement from scratch the most basic element of a neural network (the perceptron) and the math behind the fundamental block of Artificial Intelligence.

Marcus Sena · Follow

Published in Python in Plain English · 7 min read · Sep 13, 2023

Although terms like Artificial Intelligence and Machine Learning have become buzzwords and we hear or talk about these concepts on a daily basis, the mathematical concepts behind them have been around for quite a while (There are papers introducing Artificial Neural Networks — ANNs since the '40s).

But what makes the recent interest definitely more relevant? Well, there are a few good reasons:

- There is now a huge quantity of data available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.
- The tremendous increase in computing power since the 1990s now makes it possible to train large neural networks in a reasonable amount of time. This is in part due to Moore's Law, but also thanks to the gaming industry, which has pro- duced powerful GPU cards by the millions.
- The training algorithms have been improved. To be fair they are only slightly different from the ones used in the 1990s, but these relatively small tweaks have a huge positive impact.

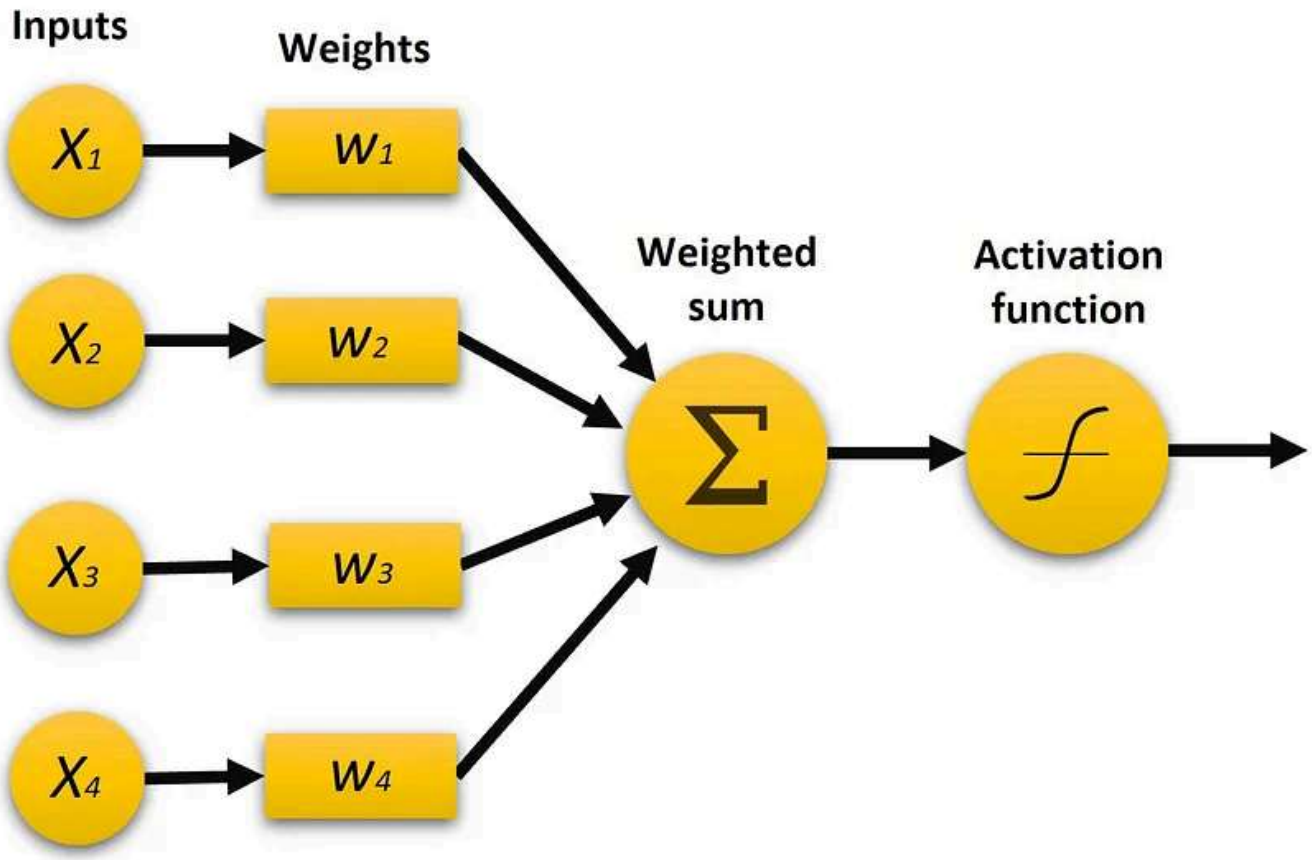
Perceptrons are the fundamental building blocks of neural networks, offering a simple yet powerful glimpse into the world of machine learning and artificial Intelligence. In this article, we'll explore the essence of perceptrons and learn how to implement them from scratch.

Perceptrons hold historical significance in the development of artificial intelligence, laying the groundwork for modern deep learning models. Their conceptual simplicity makes them an ideal starting point for anyone looking to grasp the basics of neural networks.

In the following sections, we'll dive into the architecture and inner workings of perceptrons. We'll provide code examples and practical insights to ensure you gain a solid understanding of this fundamental concept. Hence, we will use the following python packages in order to implement the perceptron.

```
import numpy as np
import matplotlib.pyplot as plt
import random
import math
```

Perceptron Architecture



Components of a Perceptron.

A perceptron consists of the following core components:

1. Input Features

Input features represent the raw data that the perceptron uses to make decisions. In this article, we will define two inputs *(X1, X2)* with fixed values to simplify the neural network. However, in real applications, the neuron may use many inputs in order to represent diverse input information.

```
X = [1, 0]
```

2. Weights

Here we will generate the weights to compose our neural network. We will initially generate small random weights in this example, but it is important to note that these weights will be updated when the network is trained. The code below shows how to generate two random weights between -1 and 1 (one for each input).

```
w = []
nInputs = 2
for i in range(nInputs):
    w.append(random.uniform(-1, 1))
```

3. Bias

The bias term is an additional input to the perceptron, which serves as an adjustable threshold. It allows the perceptron to make decisions even when all input features are zero. The bias helps control the activation of the perceptron. Similarly we will set the initial bias as a small random number.

```
b = random.uniform(-1,1)
```

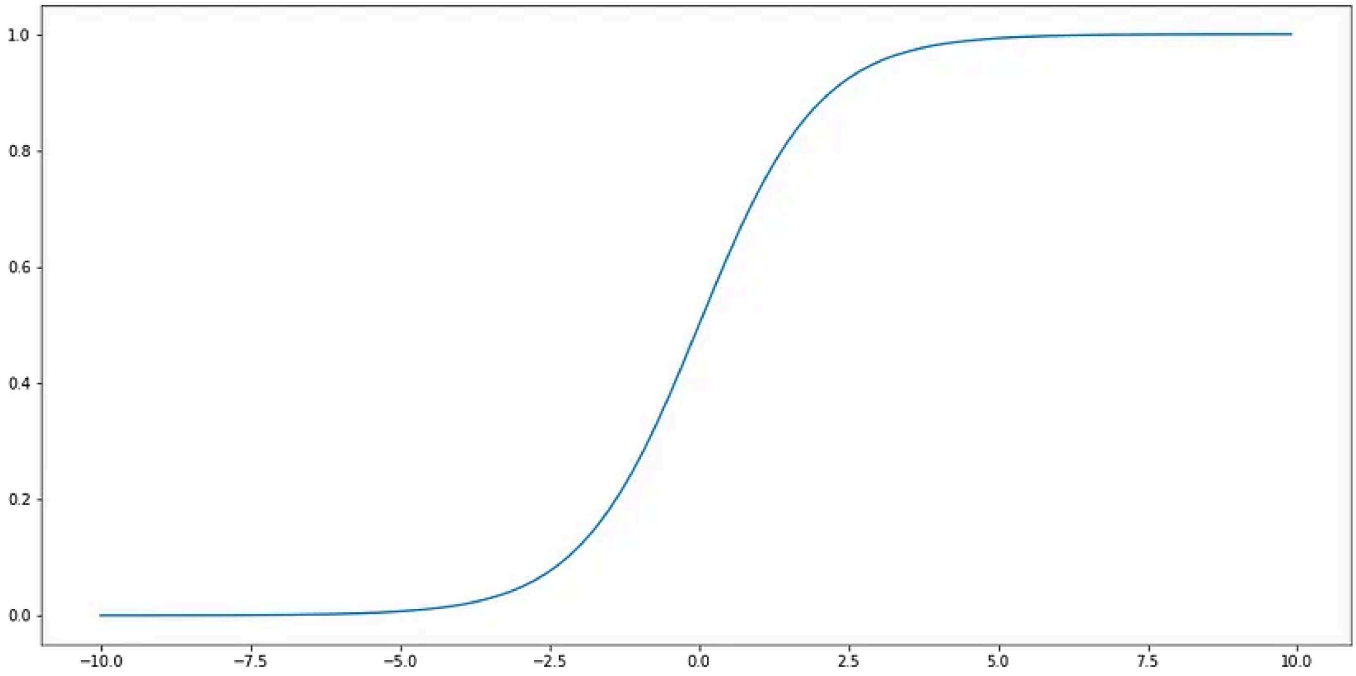
4. Activation function

The activation function transforms the weighted sum of inputs into a binary output. This output is the perceptron's prediction, representing a decision or classification. Often, we want the output of the activation function to be 0 or 1.

Another crucial attribute is its differentiability, which plays a vital role in network training. The differentiability of the function is an indispensable requirement for adjusting the network's weights during the training process, enabling the network to learn from data and improve over time.

A very common function used with that purpose is the sigmoid function, due to its differentiability, smoothness and have a simple gradient.

```
def sigmoid(x):
    return 1/(1+math.exp(-x))
```



Sigmoid function plot.

5. Feed forward

Now we got the basic elements to design the perceptron, we must pass the input data through the perceptron. This step is called **feed forward**. First we calculate the weighted sum of the inputs and then pass the result through the activation function.

The weighted sum is just the dot product between the weights vector and the inputs, then we pass it through the sigmoid function:

```
#weighted sum = w1*x1 + w2*x2 ... + b
z = sigmoid(np.dot(X, w) + b)
```

Finally we must apply the activation to predict the output as 0 or 1:

```
def activation(z):
    return 1 if z >0.5 else 0
```

Training the perceptron

Until now we only have been able to pass the input data through the perceptron without knowing if the perceptron is correct or not. The training phase is crucial to make the perceptron more accurate.

The training process involves updating the weights of the perceptron for a number of iterations (epochs). In each epoch, we calculate the error of the output relative to the real value of the target in order to update the weights and minimize the error:

```
y_pred = activation(z)
error = y - y_pred

lr = 0.1
w_new = []
for wi, xi in zip(w, X):
    w_new.append(wi + lr*error*xi)

b_new = b + lr*error
```

Where lr is the *learning rate* and we set its value to 0.1 but it could another value depending on the problem.

Hence, we now have updated weights the we would repeat the feed forward step for a number of epochs. By repeatedly adjusting the weights and biases based on the gradients, the network gradually learns to make better

predictions, reducing the error over time. That step is also known as **backpropagation**.

Final step: Applying to the Iris Dataset

Until now, we looked at the parts of the perceptron and how they worked. Next, we will gather all that we learned to create the class Perceptron and use a real dataset to verify how the perceptron works.

```
class Perceptron():

    def __init__(self,input_size = 2, lr = 0.01, epochs = 20):
        #setting default parameters
        self.lr = lr
        self.epochs = epochs
        self.input_size = input_size
        self.w = np.random.uniform(-1, 1, size=(input_size))
        self.bias = random.uniform(-1,1)
        self.misses = []

    def predict(self, X):
        w = self.w
        b = self.bias
        z = sigmoid(np.dot(X,self.w) + b)

        if z > 0.5:
            return 1
        else:
            return 0

    def fit(self, X, y):

        for epoch in range(self.epochs):
            miss = 0
            for yi, xi in zip(y, X):
                y_pred = self.predict(xi)
                #update the weights to minimize error
                error = yi - y_pred
                self.w += self.lr*error*xi
                self.bias += self.lr*error
                miss += int(error != 0.0)
```

```
            #get the number of missclassifications of each epoch
            self.misses.append(miss)
```

From the code above:

- lr: learning rate
- input_size: number of features of input data
- epochs: number of iterations of the training
- X: input data (sample, features)
- y: target data (labels 0 or 1)

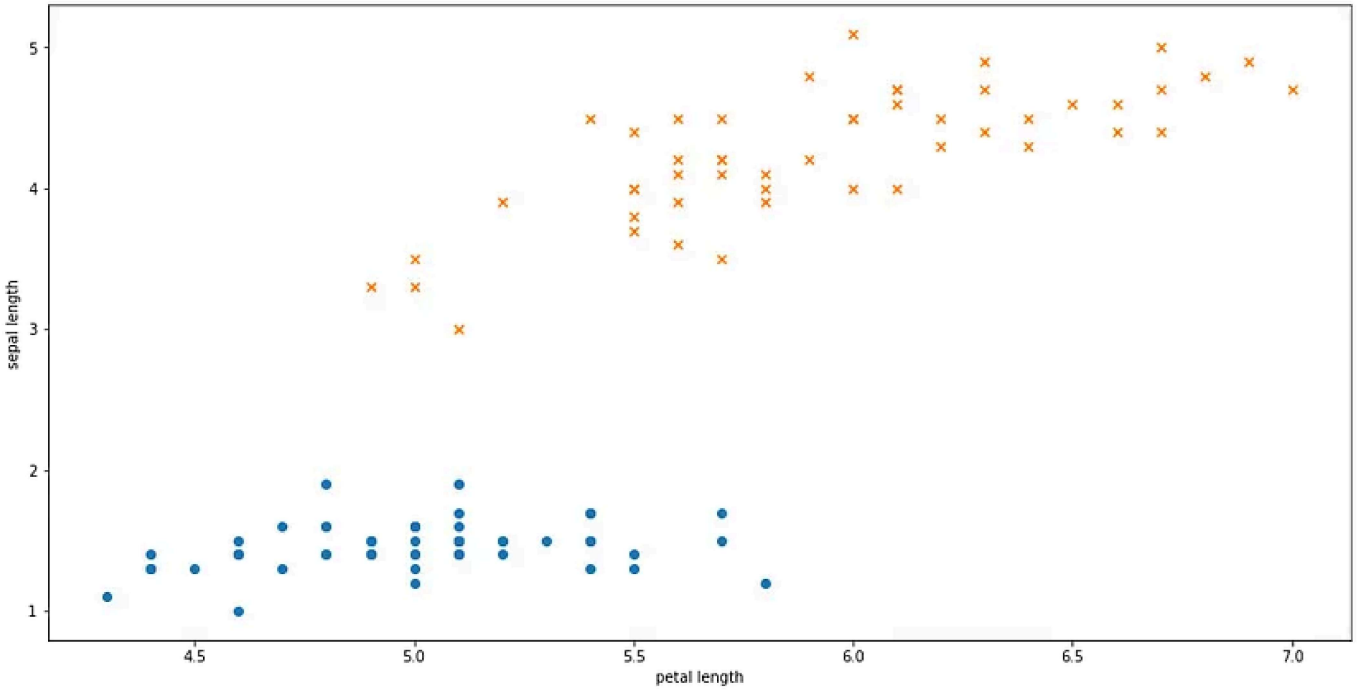
Now, we'll use the implementation of the perceptron over the iris dataset from scikit-learn library:

```
from sklearn import datasets
iris = datasets.load_iris()
X = iris.data
y = iris.target
```

The data is composed of 4 features (Sepal Length, Sepal Width, Petal Length and Petal Width) and 150 samples divided in 3 classifications (Setosa, Versicolour, and Virginica). For simplification we'll only use 2 features (Sepal Length and Petal Length) and the Setosa and Versicolor samples:

```
X = X[:100, [0,2]]
y = y[y<2]
```

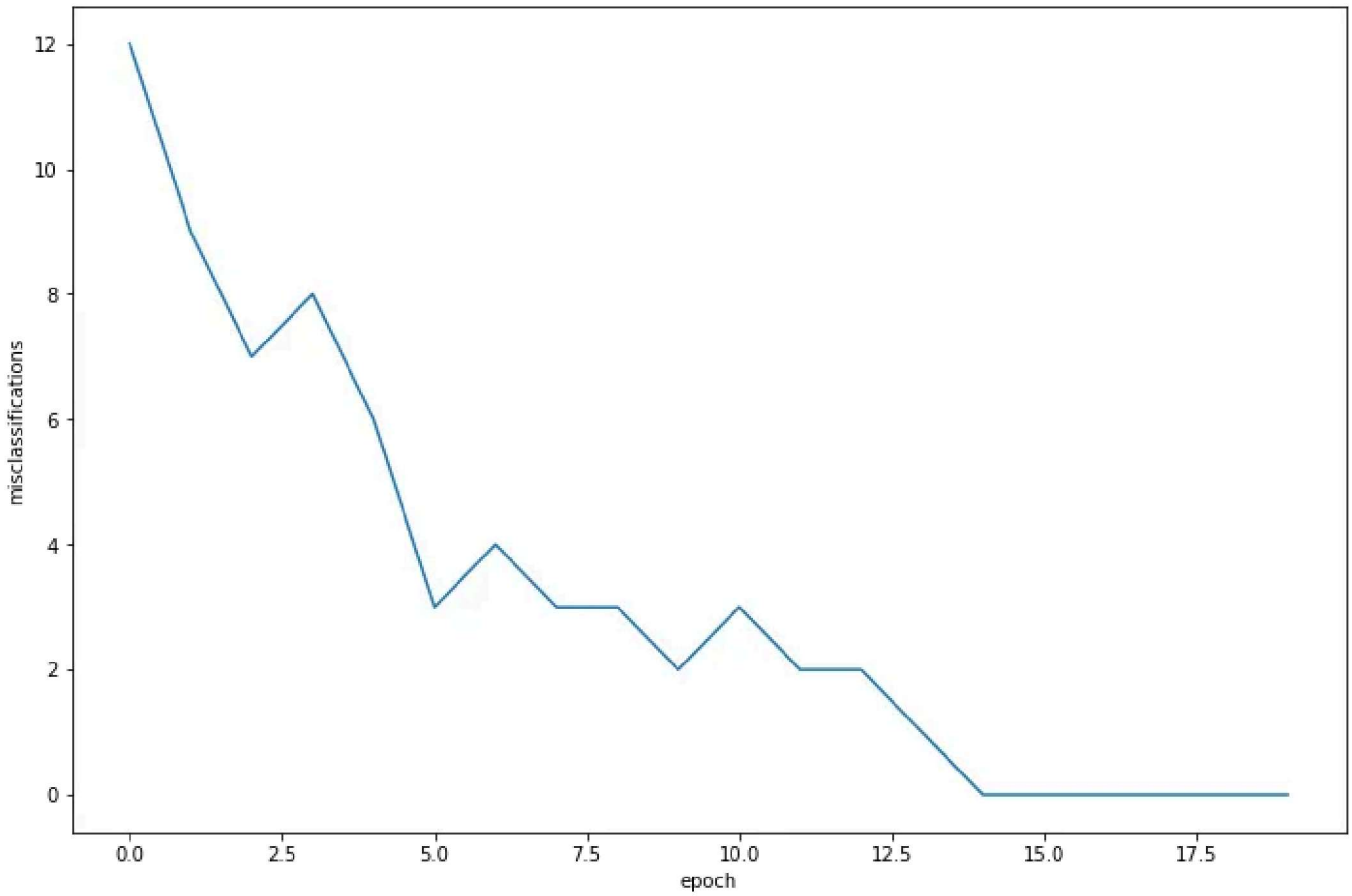
```
# plot the data
plt.figure(figsize=(16,8))
plt.scatter(X[:50,0], X[:50,1], marker='o', label='setosa')
plt.scatter(X[50:,0], X[50:,1], marker='x', label='virginica')
plt.ylabel('sepal length')
plt.xlabel('petal length')
plt.show()
```



Now, we must train the perceptron using the labeld data (X, y) and see how it performs:

```
perceptron = Perceptron()
perceptron.fit(X,y)
print(perceptron.w, perceptron.bias)
#(array([0.09893529, 0.09323132]), -0.763184789232628)
```

That way we obtained the optimal weights and bias of the training. But how well the perceptron performed in terms of misclassifications? For that end, we'll plot the misses parameter of the perceptron class:



Number of misclassifications of the perceptron during training.

As we can see, the perceptron misses 12 times in the first epoch and continously learns until make no mistake in the end of the training.

Conclusions & Next steps

We successfully were able to implement and train a perceptron and evaluate its performance with a real dataset.

Although the perceptron is the founding block of neural networks, it have its limitations as we saw along the article:

- It worked well in a 2-class dataset;
- The problem was linearly separable. In other words, they can only classify data that can be separated into two classes by a straight line or hyperplane;
- We have dealt with a clean dataset with little or no noise, what made possible to the perceptron converge with ease.

One way to overcome these limitations is to use more complex models like multi-layer perceptrons (MLPs) with nonlinear activation functions (which are the subject of the following posts). MLPs can learn non-linear decision boundaries, allowing them to tackle more complex problems.

Please feel free to comment, make suggestions and connect with me on LinkedIn, X and Github.

[1] The Iris dataset, available: https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html

[2] Sebastian Raschka (2015), Python Machine Learning.

[3] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain.

In Plain English

Thank you for being a part of our community! Before you go:

- *Be sure to **clap** and **follow** the writer!* 👏
- *You can find even more content at **PlainEnglish.io*** 🚀
- *Sign up for our **free weekly newsletter**.* 🗞️
- *Follow us on **Twitter(X)**, **LinkedIn**, **YouTube**, and **Discord**.*

Machine Learning | Artificial Intelligence | Data Science | Python | Perceptron

Written by Marcus Sena

521 Followers · Writer for Python in Plain English

Follow

More from Marcus Sena and Python in Plain English

Marcus Sena in Towards Data Science

Principal Component Analysis Made Easy: A Step-by-Step Tutorial

Implement the PCA algorithm from scratch with Python

Jun 8 193 1

Code geass in Python in Plain English

8 Python Performance Tips I Discovered After Years of Coding ...

Hey Everyone!! I wanted to share these Python Performance tips, that i feel that...

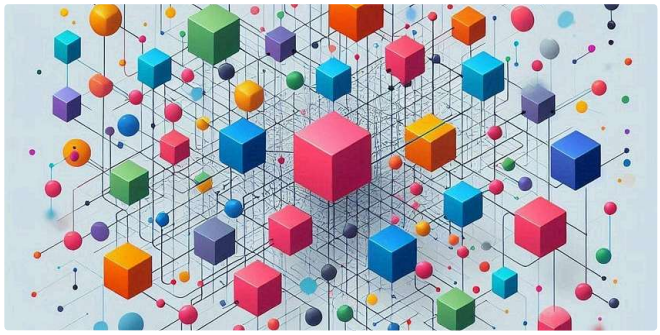
Oct 12 625 6

Bryson Meiling in Python in Plain English

What is Pydantic and how can it help your next python project?

For anyone that has been either living under a programming rock you may not have heard ...

Oct 6 456 9



Marcus Sena in Towards Data Science

Mastering K-Means Clustering

Implement the K-Means algorithm from scratch with this step-by-step Python tutorial

May 22 224 2

Recommended from Medium

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

Here, x_i are the activations in the mini-batch, μ_B is the mean, σ_B^2 is the variance, and m is the mini-batch size.

Jo Wang

Deep Learning Part 5 -How to prevent overfitting

Techniques used to prevent overfitting in deep learning models:

Jun 29 1 1

Sai kumaresh 🚀

Python Basics: From Zero to Hero in Simple Steps 🐍⭐

Python Setup, Basics, Data Types, Control Structures, Data Structures, Functions, OOP...

Sep 1 145

Lists

Predictive Modeling w/ Python

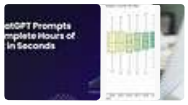
20 stories · 1620 saves

Practical Guides to Machine Learning

10 stories · 1977 saves

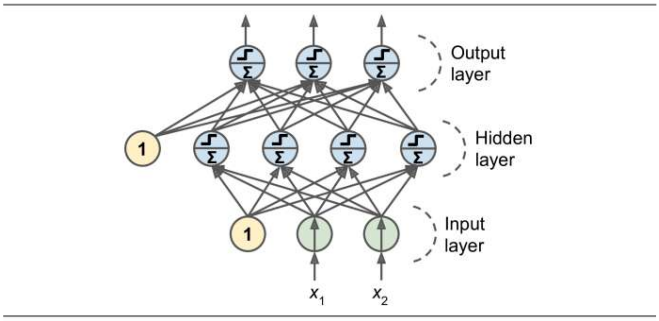
Natural Language Processing

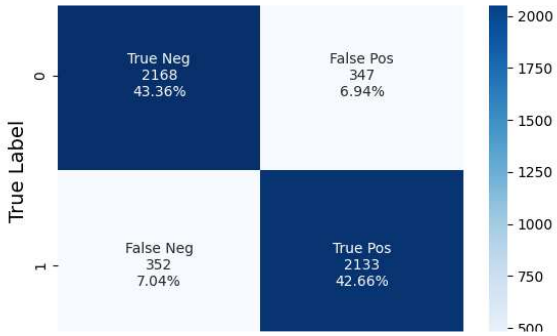
1779 stories · 1383 saves



ChatGPT prompts

50 stories · 2149 saves





Sanjay Dutta

Multi-Layer Perceptron and Backpropagation: A Deep Dive

The Multi-Layer Perceptron (MLP) is a cornerstone in the field of artificial neural...

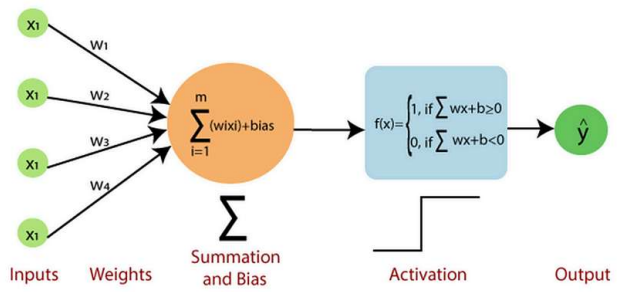
Jun 3 10

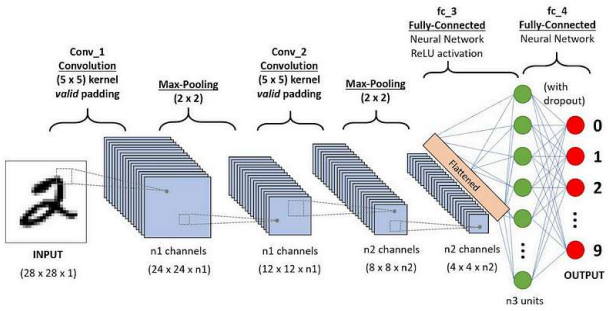
Nivedita Bhadra

Build a machine learning model for sentiment analysis in Python

An NLP project on IMDB movie reviews

Oct 17 1





Abhishek Jain

Perceptron vs neuron, Single layer Perceptron and Multi Layer...

In deep learning, the terms “perceptron” and “neuron” are related but have distinct...

Sep 21 13 1

Luqman Zaceria in Lumos

Understanding Convolutional Neural Networks (CNNs)

Hey everyone! We’re going to explore one of the most influential and powerful tools in the...

Aug 5 73

See more recommendations