

Image Stitching using OpenCV — A Step-by-Step Tutorial



Paulson Premsingh · Follow

11 min read · 5 days ago



Listen



Share

<https://www.linkedin.com/in/paulsonpp/>

Overview : In this article, we'll look at how to use OpenCV to stitch together images. In order to create a bigger composite image, such as a panorama, image stitching entails joining several overlapping photos. We'll go over each phase of the image stitching process, including feature matching, homography estimate, and blending. You will be able to use OpenCV to combine numerous images into spectacular panoramas at the end of this tutorial.

Table of Contents:

1. Introduction
2. Image Stitching Pipeline
3. Feature Detection and Extraction
4. Feature Matching
5. Homography Estimation
6. Warping and Blending
7. Conclusion

Introduction

Image stitching is a method for creating composite images, including panoramic views, from several overlapping photographs. More information can be captured and a larger field of view can be created than what can be done with just one image. Numerous industries, including photography, virtual reality, and robotics, use image stitching.

Through the use of image stitching, panoramas can be produced that provide viewers a seamless perspective of expansive landscapes or architectural marvels. They provide a means of preserving and capturing a scene's beauty, which cannot be properly conveyed in a single frame. Photographers can use image stitching to display the magnificence of a mountain range, the size of an open field, or the minute details of an architectural wonder.

Large-scale maps, 3D scene reconstruction, and even medical imaging, where overlapping scan slices are stitched together, can all be made using image stitching. Image stitching can help roboticists create panoramic maps for navigation or visual localization.

We may explore a variety of uses for this technology and unlock the potential to produce magnificent panoramic photographs by comprehending the image stitching procedure and employing OpenCV's capabilities. Whether you are an engineer working on robotics or mapping projects, a computer vision researcher, or a photography lover, understanding picture stitching with OpenCV can broaden your creative options and open doors to fascinating opportunities.

Image Stitching Pipeline

A pipeline that converts numerous separate photos into a seamless composite can be formed by breaking down the image stitching procedure into a number of steps. To achieve precise alignment and blending, each stage is essential. In this section,

we will outline the fundamental ideas and methods used in the step-by-step image stitching pipeline.

The key concepts and techniques involved in image stitching include:

Keypoint detection: Identifying distinctive features in images.

Descriptors: Representing keypoints with numerical descriptors for matching.

Feature matching: Finding correspondences between keypoints across images.

Homography: The transformation matrix that describes the geometric relationship between images.

RANSAC: An algorithm for estimating the homography matrix robustly.

Image warping: Transforming images based on the estimated homography.

Blending: Merging images smoothly to create a seamless composite.

For successful image stitching, it is crucial to comprehend these ideas and methods. We will examine each stage in depth in the ensuing sections, with real-world examples and OpenCV code samples. You may produce eye-catching panoramic photographs and master the skill of image stitching by using this pipeline and the proper procedures.

Feature Detection and Extraction

Feature extraction and detection is the first and most important stage in the image stitching pipeline. This stage entails locating different keypoints in each image and calculating their descriptions. Descriptors give numerical representations of these keypoints for matching across multiple photos, while keypoints reflect distinctive features or points of interest in the image. We may use OpenCV's several feature detection techniques, including SIFT, SURF, and ORB, to extract keypoints and descriptors from images.

Feature Detection

- The goal of feature identification algorithms is to locate intriguing and recognizable focal areas in an image.

- These keypoints could stand in for locations with substantial texture differences, corners, or edges.
- For the purpose of detecting features, algorithms such as SIFT (Scale-Invariant Feature Transform), SURF (Speeded-Up Robust Features), and ORB (Oriented FAST and Rotated BRIEF) are frequently employed.
- These algorithms take into account the keypoints' scale and orientation to make them resistant to changes in scale and rotation.

Feature Extraction

- The local picture data surrounding each keypoint is computed using the keypoint descriptors that have been identified.
- The keypoints' surrounding patches are numerically represented as descriptors, which encode their look or texture.
- Descriptors include details on the neighborhood's texture, color, and gradient characteristics.
- Feature descriptor extraction techniques are also included in the SIFT, SURF, and ORB algorithms.

Let's have a practice using SIFT for feature detection and extraction.

```
import cv2

# Load the images
image1 = cv2.imread('image1.jpg')
image2 = cv2.imread('image2.jpg')

# Initialize the SIFT feature detector and extractor
sift = cv2.SIFT_create()

# Detect keypoints and compute descriptors for both images
keypoints1, descriptors1 = sift.detectAndCompute(image1, None)
keypoints2, descriptors2 = sift.detectAndCompute(image2, None)

# Draw keypoints on the images
image1_keypoints = cv2.drawKeypoints(image1, keypoints1, None)
image2_keypoints = cv2.drawKeypoints(image2, keypoints2, None)

# Display the images with keypoints
```

```
cv2.imshow('Image 1 with Keypoints', image1_keypoints)
cv2.imshow('Image 2 with Keypoints', image2_keypoints)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

This code snippet loads images *image1* and *image2* first. Then, using *cv2.SIFT_create()*, we initialize the SIFT feature detector and extractor.

Next, we calculate descriptors for both pictures using the *detectAndCompute* method of the SIFT object to identify keypoints. While the descriptors are kept in the *descriptors1* and *descriptors2* variables, the keypoints are kept in the *keypoints1* and *keypoints2* variables.

The *drawKeypoints* function is used to draw the keypoints on the images in order to view them. The variables *image1_keypoints* and *image2_keypoints* contain the keypoints for the final images.

This piece of code demonstrates the SIFT algorithm's feature detection and extraction process. By simply swapping out the '*sift*' object with the appropriate detector and extractor objects (for example, *cv2.SURF_create()* for SURF or *cv2.ORB_create()* for ORB) and modifying the code accordingly, you may experiment with alternative feature detection techniques, such as SURF or ORB.

To get the best results, keep in mind to try out several algorithms and modify their parameters according to your unique use case.

Feature matching

A crucial stage in image stitching is feature matching, which creates correspondences between keypoints in various images. In order to ensure accurate alignment and registration of the images, it seeks to identify keypoints that match based on their descriptors. Numerous feature matching methods, including FLANN (Fast Library for Approximate Nearest Neighbors) based matching and brute-force matching, are offered by OpenCV, each with unique properties and benefits.

Brute-force matching

- A straightforward and easy method of feature matching is brute-force matching.
- Every descriptor from one image is compared to every descriptor from the second image.
- Calculating distances or similarities between descriptors (such the Euclidean distance or cosine similarity) is a step in the matching process.
- The OpenCV *BFMatcher* class allows for the use of brute-force matching.
- Even though it is straightforward, it can be computationally expensive, particularly for big feature sets.

FLANN (Fast Library for Approximate Nearest Neighbors) Matching

- A practical closest neighbor search algorithm for effective feature matching is FLANN matching.
- To expedite the search, it employs tree-based indexing structures like KD-trees or randomized trees.
- Particularly for big feature sets, FLANN can be quicker than brute-force matching.
- It offers choices for selecting the search algorithm and configuring the search criteria.
- The *FlannBasedMatcher* class in OpenCV can be utilized to conduct FLANN matching.

This code snippet demonstrates how to perform feature matching using both brute-force matching and FLANN matching algorithms in OpenCV

```
import cv2

# Load the images
image1 = cv2.imread('image1.jpg', cv2.IMREAD_GRAYSCALE)
image2 = cv2.imread('image2.jpg', cv2.IMREAD_GRAYSCALE)

# Initialize the feature detector and extractor (e.g., SIFT)
sift = cv2.SIFT_create()

# Detect keypoints and compute descriptors for both images
```

```

keypoints1, descriptors1 = sift.detectAndCompute(image1, None)
keypoints2, descriptors2 = sift.detectAndCompute(image2, None)

# Initialize the feature matcher using brute-force matching
bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)

# Match the descriptors using brute-force matching
matches_bf = bf.match(descriptors1, descriptors2)

# Sort the matches by distance (lower is better)
matches_bf = sorted(matches_bf, key=lambda x: x.distance)

# Draw the top N matches
num_matches = 50
image_matches_bf = cv2.drawMatches(image1, keypoints1, image2, keypoints2, matches_bf, num_matches)

# Initialize the feature matcher using FLANN matching
index_params = dict(algorithm=0, trees=5)
search_params = dict(checks=50)
flann = cv2.FlannBasedMatcher(index_params, search_params)

# Match the descriptors using FLANN matching
matches_flann = flann.match(descriptors1, descriptors2)

# Sort the matches by distance (lower is better)
matches_flann = sorted(matches_flann, key=lambda x: x.distance)

# Draw the top N matches
image_matches_flann = cv2.drawMatches(image1, keypoints1, image2, keypoints2, matches_flann, num_matches)

# Display the images with matches
cv2.imshow('Brute-Force Matching', image_matches_bf)
cv2.imshow('FLANN Matching', image_matches_flann)
cv2.waitKey(0)
cv2.destroyAllWindows()

```

Starting with the L2 norm and *crossCheck=True* for feature matching, we initialize the brute-force matcher using *cv2.BFMatcher()*. The *match()* function is then used to look for similarities between the two images' descriptions. On the basis of distance, the matches are arranged.

Similar to this, we set up the index settings and search parameters for the FLANN matcher using *cv2.FlannBasedMatcher()*. Using the FLANN algorithm, we invoke the *match()* method to look for matches between the descriptors. Once more, the matches are arranged according to their distance.

We draw the matches on the images using the *drawMatches()* function to visualize the matches. For clarity, we only display the first N matches (as determined by *num_matches*).

Homography Estimation

In the process of stitching together several photos, the fundamental step of homography estimation is essential for aligning and integrating the images. In order to transfer similar points from one image to another, a homography matrix is used to express the transformation between two image planes. With the help of matched keypoints received from feature matching, OpenCV offers techniques for estimating the homography matrix.

Understanding Homography and its Role in Image Stitching

- Homography, which depicts a planar mapping between two images, is a 3x3 transformation matrix.
- It shows the correspondence between two images' corresponding points' pixel coordinates.
- Geometric changes including translation, rotation, scaling, and perspective distortion are possible using homography.
- Homography is a technique used in image stitching to align and transform the images to a single coordinate system.

Estimating Homography Matrix using Matched Keypoints

- The *findHomography()* function in OpenCV can be used to calculate the homography matrix.
- The estimated homography matrix is returned by the *findHomography()* method, which accepts the matching keypoints as input.
- RANSAC (Random Sample Consensus), a reliable estimate procedure, is used to handle outliers and provide an accurate homography matrix.

- RANSAC estimates the homography matrix by picking a subset of the matched keypoints iteratively. The optimal homography matrix with the most inliers is chosen after assessing the number of inliers that suit the predicted model.

Here's a code snippet that demonstrates homography estimation using matched keypoints:

```
import cv2
import numpy as np

# Load the images
image1 = cv2.imread('image1.jpg')
image2 = cv2.imread('image2.jpg')

# Convert the images to grayscale
gray1 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
gray2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)

# Initialize the feature detector and extractor (e.g., SIFT)
sift = cv2.SIFT_create()

# Detect keypoints and compute descriptors for both images
keypoints1, descriptors1 = sift.detectAndCompute(gray1, None)
keypoints2, descriptors2 = sift.detectAndCompute(gray2, None)

# Initialize the feature matcher using brute-force matching
bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)

# Match the descriptors using brute-force matching
matches = bf.match(descriptors1, descriptors2)

# Extract the matched keypoints
src_points = np.float32([keypoints1[m.queryIdx].pt for m in matches]).reshape((-1, 2))
dst_points = np.float32([keypoints2[m.trainIdx].pt for m in matches]).reshape((-1, 2))

# Estimate the homography matrix using RANSAC
homography, mask = cv2.findHomography(src_points, dst_points, cv2.RANSAC, 5.0)

# Print the estimated homography matrix
print("Estimated Homography Matrix:")
print(homography)
```

Warping and Blending

The following steps entail warping the images depending on the homography and combining them to generate a seamless composite image after estimating the homography matrix in the previous step. The stitched photos will be properly aligned and have seamless transitions between them thanks to this procedure.

Warping Images using the Estimated Homography

- After estimating the homography matrix, we may use it to convert one image's coordinate system to that of the other.
- The `cv2.warpPerspective()` function in OpenCV allows you to warp an image by applying the homography transformation.
- The estimated homography matrix, the size of the output picture, and the input image are all input parameters for the function.
- The size of the output image will match that of the reference image (or destination image).

Blending the Warped Images to Create a Seamless Composite

- To produce a smooth composite image, we must blend the warped images together.
- Simple picture concatenation could produce obvious seams or jarring changes in the images.
- Several blending methods, including alpha blending, feathering, and multi-band blending, can be used to produce a smoother blending effect.
- To achieve a seamless and realistic outcome, these strategies try to progressively transition the pixel intensities between the images.
- The individual needs and qualities of the photos being stitched determine the best blending technique to use.

Here's a code snippet that demonstrates the warping and blending of images using the estimated homography:

```
import cv2
import numpy as np
```

```

# Load the images
image1 = cv2.imread('image1.jpg')
image2 = cv2.imread('image2.jpg')

# Convert images to grayscale
gray1 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
gray2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)

# Initialize the feature detector and extractor (e.g., SIFT)
sift = cv2.SIFT_create()

# Detect keypoints and compute descriptors for both images
keypoints1, descriptors1 = sift.detectAndCompute(gray1, None)
keypoints2, descriptors2 = sift.detectAndCompute(gray2, None)

# Initialize the feature matcher using brute-force matching
bf = cv2.BFMatcher()

# Match the descriptors using brute-force matching
matches = bf.match(descriptors1, descriptors2)

# Select the top N matches
num_matches = 50
matches = sorted(matches, key=lambda x: x.distance)[:num_matches]

# Extract matching keypoints
src_points = np.float32([keypoints1[match.queryIdx].pt for match in matches]).r
dst_points = np.float32([keypoints2[match.trainIdx].pt for match in matches]).r

# Estimate the homography matrix
homography, _ = cv2.findHomography(src_points, dst_points, cv2.RANSAC, 5.0)

# Warp the first image using the homography
result = cv2.warpPerspective(image1, homography, (image2.shape[1], image2.shape[0]))

# Blending the warped image with the second image using alpha blending
alpha = 0.5 # blending factor
blended_image = cv2.addWeighted(result, alpha, image2, 1 - alpha, 0)

# Display the blended image
cv2.imshow('Blended Image', blended_image)
cv2.waitKey(0)
cv2.destroyAllWindows()

```

Conclusion

In this tutorial, we have explored the image stitching process using OpenCV. Here is a summary of the steps involved:

- *Image Loading*: Load the individual images you want to combine.
- *Feature Extraction and Matching*: To find keypoints and generate descriptors, use a feature detection and extraction method (such as SIFT or SURF). To discover correspondences between keypoints in various photos, use feature matching.
- *Homography Estimation*: Using the matched keypoints, estimate the homography matrix. The geometric transformation between the images is represented by the homography matrix.
- *Warping and Blending*: To align one image with the other, warp it using the approximated homography. To produce a smooth composite image, blend the distorted image with the second image.

Throughout the tutorial, we have covered key concepts and techniques involved in image stitching, including feature matching, homography estimation, and image warping. We have also provided code snippets to illustrate the implementation using OpenCV.

Further Exploration and Potential Improvements:

- To examine the effectiveness of various feature extraction and detection techniques for image stitching, conduct experiments.
- For more reliable homography estimate, look into cutting-edge techniques like RANSAC (Random Sample Consensus).
- For more seamless transitions between photos, consider mixing techniques like multi-band blending and gradient-based blending.
- Utilizing sophisticated algorithms and techniques, handle image stitching difficulties such as parallax, perspective distortion, and occlusions.
- To increase the accuracy of feature matching, look into automatic picture alignment approaches like scale-invariant feature transform (SIFT) or robust matching algorithms.

Virtual reality, picture mosaics, and panoramic photography are just a few of the exciting uses for image stitching. You may improve the precision and quality of stitched images with additional research and development, providing a seamless and immersive visual experience.

Feel free to experiment and explore the vast possibilities of image stitching using OpenCV and other related libraries. *Happy stitching!*

About Author :

I'm Paulson Premsingh, a graduate student of National University of Singapore in AI. I have a strong passion for Vision Systems, 3D Sensing, and Robotics and Mobility. With a focus on these areas, I'm eager to explore the intersection of artificial intelligence and computer vision to develop innovative solutions for robotic perception, object recognition, and autonomous navigation. I'm excited to be part of the AI community and contribute to advancements in these fascinating fields.

Image Stitching

Opencv



Follow



Written by Paulson Premsingh

0 Followers

Recommended from Medium





 Turgay Ceylan


Image Processing in Python with OpenCV—Morphological Operations

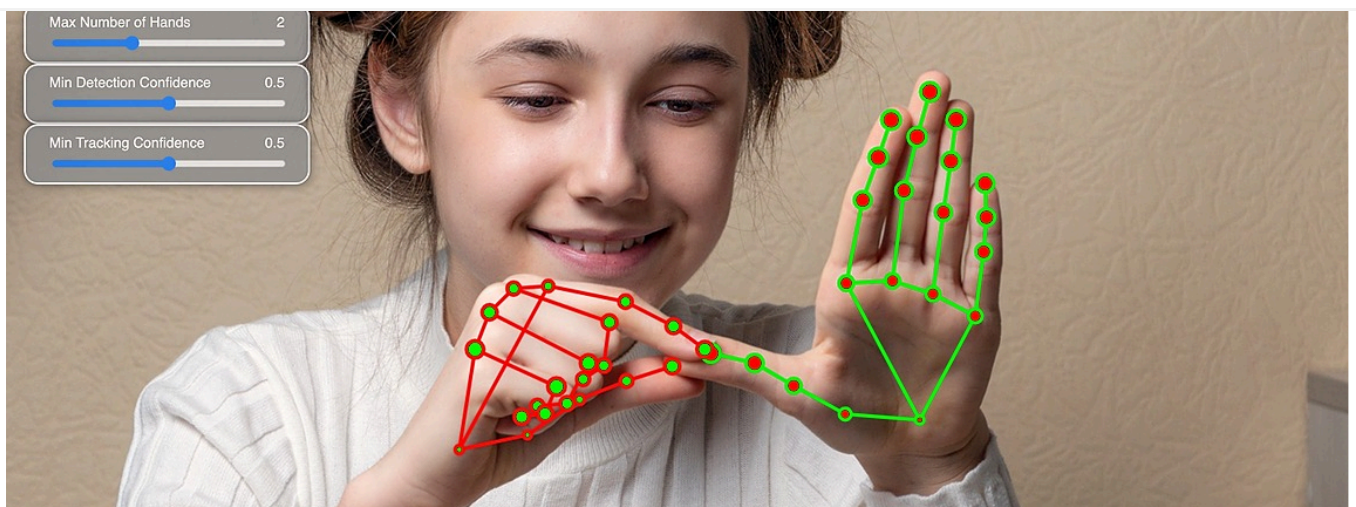
Let's learn morphological image processing techniques and examine how to use them with examples.

4 min read · Dec 24, 2022

 --  1



 Search Medium

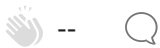


 Keshanth Jude in Dev Genius

Object Detection with MediaPipe

After the massive release of ChatGPT, the world has grown curious (and scared) of Artificial Intelligence and Machine Learning. This blog...

4 min read · Jan 3



Lists



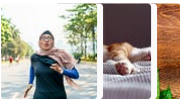
Staff Picks

320 stories · 81 saves



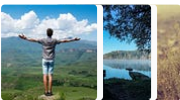
Stories to Help You Level-Up at Work

19 stories · 40 saves



Self-Improvement 101

20 stories · 87 saves



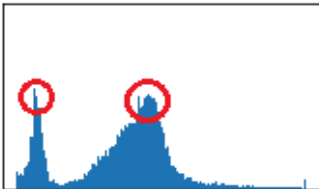
Productivity 101

20 stories · 88 saves

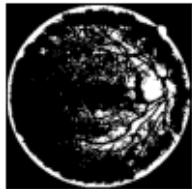
Original Noisy Image



Histogram



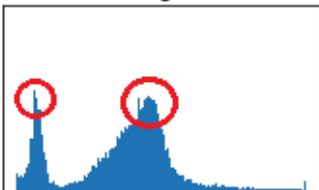
Global Thresholding ($\tau=127$)



Original Noisy Image



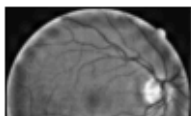
Histogram



Otsu's Thresholding



Gaussian filtered Image



Histogram



Otsu's Thresholding

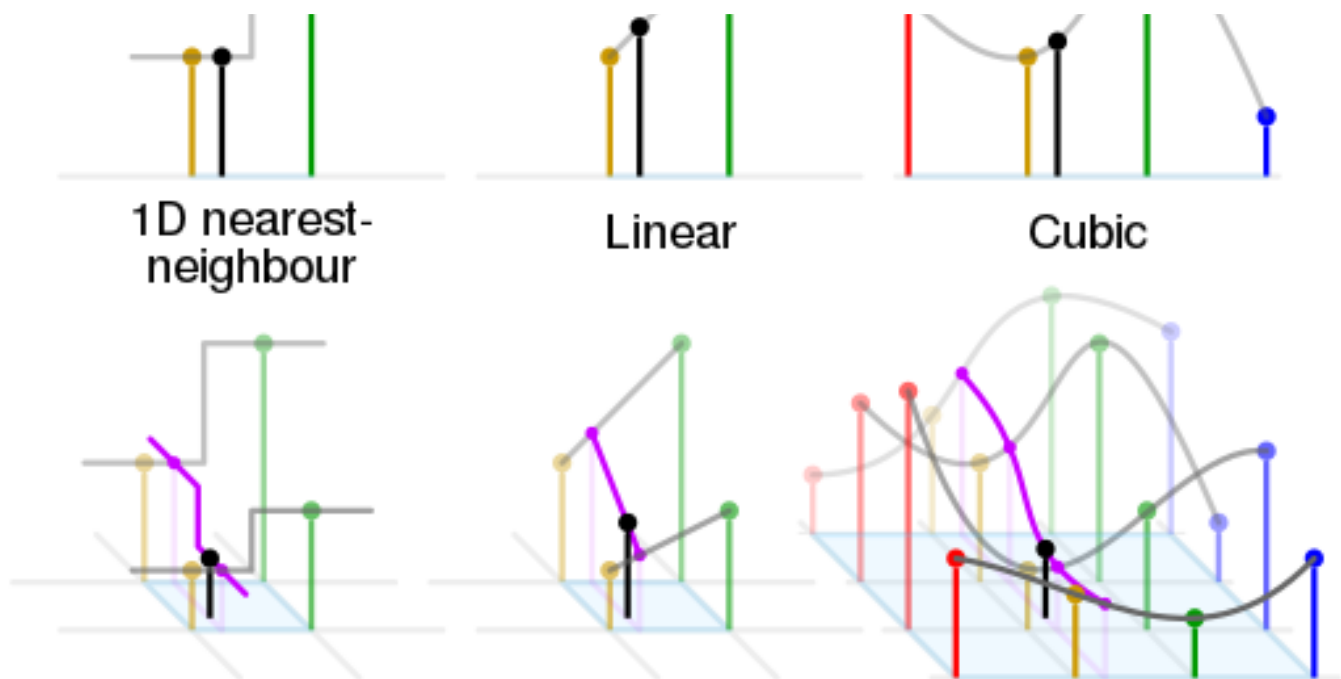


Amit Chauhan

OpenCV: Adaptive and Otsu Threshold in Image Processing with Python

Image pre-processing techniques in artificial intelligence

★ · 3 min read · Mar 20



 Zahid Parvez

Image interpolation in OpenCV

Some form of image interpolation is always taking place when manipulating digital images—whether it's resizing them to increase or...

🌟 · 3 min read · Jan 8



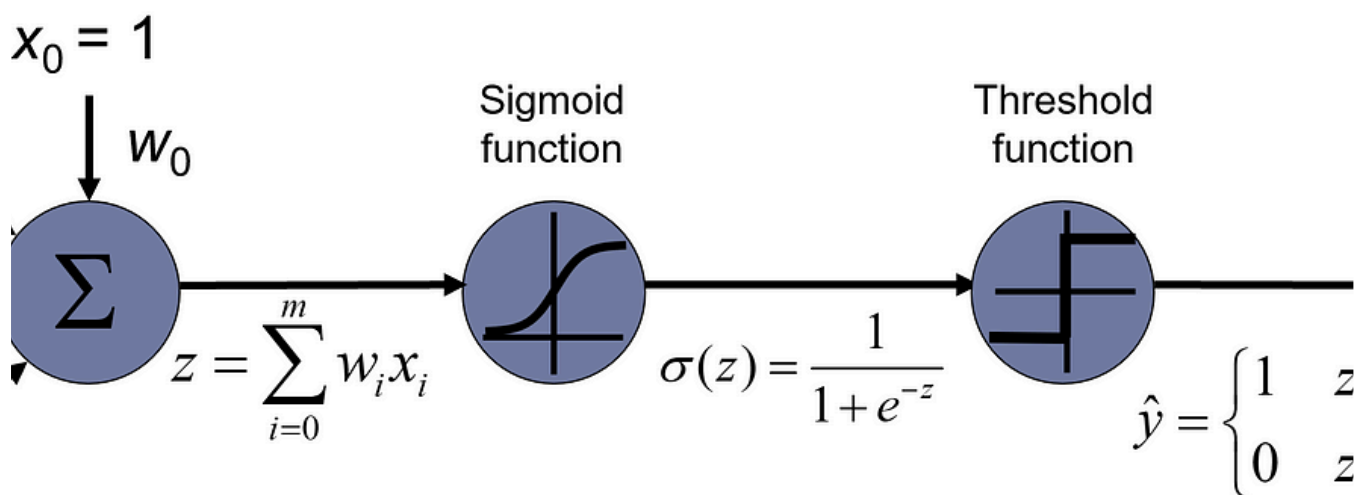
 Gabe Araujo, M.Sc.  in Level Up Coding

Introducing PandasAI: The Generative AI Python Library

Pandas AI is an additional Python library that enhances Pandas, the widely-used data analysis and manipulation tool, by incorporating...

★ · 9 min read · May 16

 --  7

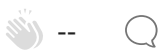


 Dr. Roi Yehoshua in Towards Data Science

Mastering Logistic Regression

From theory to implementation in Python

🌟 · 17 min read · 6 days ago



See more recommendations