NumPy for Machine Learning Beginners: A Hands-On Guide Using Google Colab

February 21, 2025

1 Introduction: Why NumPy?

Imagine you're trying to teach a computer to recognize cats in pictures. You need to feed it tons of images, and each image is broken down into millions of numbers representing the color of each pixel. Now, imagine trying to store and manipulate all those numbers using regular Python lists. It would be incredibly slow and inefficient!

That's where NumPy comes in. NumPy, short for "Numerical Python," is a powerful tool that makes working with large amounts of numerical data in Python much faster and easier. It's the foundation upon which many machine learning libraries are built. In this book, we'll start from scratch and learn how to use NumPy in Google Colab to prepare our data for machine learning models.

2 Lists in Python - A Quick Review

Before diving into NumPy, let's quickly revisit lists in Python. You've probably used them before, but let's make sure we're all on the same page.

A list is simply a collection of items. You create a list using square brackets `[]` and separate the items with commas:

```
my_list = [1, 2, 3, 4, 5]
print(my_list) # Output: [1, 2, 3, 4, 5]
```

You can access individual items in a list using their index (position), starting from 0:

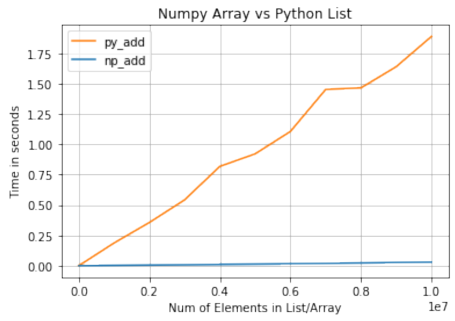
```
print(my_list[0]) # Output: 1
print(my_list[2]) # Output: 3
```

One interesting feature of lists is that they can hold different data types:

```
mixed_list = [1, "hello", 3.14, True]
print(mixed_list) # Output: [1, 'hello', 3.14, True]
```

This flexibility is useful in some cases, but it comes with a performance cost when dealing with large numerical datasets.

3 The Need for Speed: Why NumPy Arrays?



In machine learning, we often work with huge datasets containing mostly numbers. Lists can store these numbers, but they are generally slow and computationally expensive for large datasets. Imagine trying to perform a simple arithmetic operation on millions of numbers in a list!

This is where NumPy arrays shine. NumPy arrays are like lists, but they are designed specifically for efficient numerical operations. They are much faster and more powerful than lists when it comes to handling numerical data.

4 Introducing NumPy Arrays

Think of NumPy arrays as super-charged lists specifically designed for numbers. They are the core data structure for numerical computations in Python.

4.1 Importing NumPy

To use NumPy, you first need to import it into your Python environment. A standard practice is to import NumPy with the alias `np`:

```
import numpy as np
```

This allows you to refer to NumPy functions and objects using the shorter `np` prefix.

4.2 Creating NumPy Arrays

You can create a NumPy array from a Python list using the `np.array()` function:

```
my_list = [1, 2, 3, 4, 5]
my_array = np.array(my_list)
print(my_array) # Output: [1 2 3 4 5]
```

NumPy arrays can only store data of the *same* type. This might seem like a disadvantage compared to lists, but it's a key reason why they are so efficient for numerical computations.

4.3 Accessing Elements and Shape

Accessing elements in a NumPy array is similar to lists:

```
print(my_array[0]) # Output: 1
print(my_array[3]) # Output: 4
```

To determine the "size" of a NumPy array, you use the `.shape` attribute:

```
print(my_array.shape) # Output: (5,)
```

This tells you the array has one dimension and 5 elements.

5 NumPy's Power: Creating Arrays with Ease

One of NumPy's strengths is its ability to create arrays quickly and easily.

5.1 `np.arange()`: Generating Number Sequences

The `np.arange()` function is similar to Python's `range()` function, but it creates a NumPy array instead of a list.

```
numbers = np.arange(0, 10) # Numbers from 0 to 9
print(numbers) # Output: [0 1 2 3 4 5 6 7 8 9]

even_numbers = np.arange(2, 21, 2) # Even numbers from 2 to 20
print(even_numbers) # Output: [ 2 4 6 8 10 12 14 16 18 20]

reverse_numbers = np.arange(20, 9, -3) # Reverse array with step
    size of -3
print(reverse_numbers) # Output: [20 17 14 11]
```

5.2 `np.zeros()` and `np.ones()`: Arrays Filled with Zeros or Ones

Often, you need to create arrays filled with a specific value, like zeros or ones. NumPy provides `np.zeros()` and `np.ones()` for this:

```
zeros_array = np.zeros(10) # Array of 10 zeros
print(zeros_array) # Output: [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

ones_array = np.ones(5) # Array of 5 ones
print(ones_array) # Output: [1. 1. 1. 1. 1.]
```

By default, these functions create arrays of floating-point numbers. You can specify the data type using the `dtype` argument:

```
integer_zeros = np.zeros(5, dtype=int)
print(integer_zeros) # Output: [0 0 0 0 0]
```

5.3 Multidimensional Arrays

You can create two-dimensional (or higher) arrays using `np.zeros()` and `np.ones()` by passing a tuple as the shape:

```
matrix_of_zeros = np.zeros((3, 5)) # 3 rows, 5 columns
print(matrix_of_zeros)
```

Output:

```
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

This creates a 3×5 matrix (3 rows and 5 columns) filled with zeros.

Another way to interpret this is as a “stack” of 3 arrays, each containing 5 elements.

Similarly:

```
three_d_array = np.zeros((2,3,5)) # 2 "Layers", 3 rows, 5 columns
print(three_d_array)
```

This creates a 3D array of shape $(2, 3, 5)$. You can think of it as two layers where each layer is the matrix created earlier.

The `.shape` attribute gives you the dimensions of the array:

```
print(matrix_of_zeros.shape) # Output: (3, 5)
print(three_d_array.shape) # Output: (2,3,5)
```

6 NumPy’s Powerful Functions

NumPy provides a wide range of functions for performing mathematical operations on arrays. These functions are highly optimized for performance.

6.1 Basic Mathematical Operations

```
my_array = np.array([1, 16, 23, 45, 67, 21, 5])

print(np.max(my_array)) # Output: 67 (Maximum element)
print(np.min(my_array)) # Output: 1 (Minimum element)
print(np.sqrt(25)) # Output: 5.0 (Square root)
print(np.abs(-10)) # Output: 10 (Absolute value)
print(np.exp(0)) # Output: 1.0 (Exponential e^0)
print(np.mean(my_array)) # Output: 25.428
```

6.2 Applying Functions to Entire Arrays

One of NumPy's most powerful features is the ability to apply functions to entire arrays at once:

```
numbers = np.array([1, 4, 9, 16, 25])
square_roots = np.sqrt(numbers)
print(square_roots) # Output: [1. 2. 3. 4. 5.]
```

6.3 Universal Functions (ufuncs)

NumPy's "universal functions" (ufuncs) are functions that operate element-wise on arrays. You can find a comprehensive list of ufuncs in the NumPy documentation. Some examples include: `power`, `lcm`, `gcd`, trigonometric functions (`sin`, `cos`, `tan`), comparison functions, and logical functions. You can find these in the universal function documentation. Just search "NumPy Universal Functions" on Google.

6.4 Sorting

Sorting arrays is also incredibly easy with NumPy:

```
unsorted_array = np.array([5, 2, 8, 1, 9])
sorted_array = np.sort(unsorted_array)
print(sorted_array) # Output: [1 2 5 8 9]
```

Sorting can also be applied to strings. Lowercase are ordered after uppercase, following ASCII tables.

```
string_array = np.array(["Aman","vidae","khushu", "Shashan", "
    Harman"])
print(np.sort(string_array))
```

On 2D arrays, each individual array is sorted, but not together.

```
two_d_array = np.array([[4,2,3],[6,10,4]])
print(np.sort(two_d_array))
```

For arrays containing boolean values, `False` comes before `True` because Python represents `False` as 0 and `True` as 1.

7 Array Arithmetic and Comparison

NumPy makes it easy to perform arithmetic and comparison operations on arrays.

7.1 Scalar Operations

You can perform arithmetic operations between an array and a single number (a "scalar"):

```
my_array = np.array([1, 2, 3, 4, 5])
result = my_array + 2
print(result) # Output: [3 4 5 6 7]

result2 = my_array * 6
print(result2) # Output: [ 6 12 18 24 30]

print(my_array % 6) # Output: [1 2 3 4 5]
```

7.2 Element-wise Operations

You can also perform arithmetic operations between two arrays of the same shape. The operations are performed element-wise:

```
array1 = np.array([1, 2, 3, 4])
array2 = np.array([1, 0, 1, 0])
result = array1 + array2
print(result) # Output: [2 2 4 4]
```

Similarly:

```
print(array1 > array2) # Output: [False True True True]
```

7.3 Matrix Multiplication

NumPy offers two ways to do multiplication. As seen earlier, it can be performed element-wise with the `*` symbol.

However, we can also conduct proper matrix multiplication with `@`:

```
matrix1 = np.array([[1,2,3],[4,5,6],[7,8,9]])
matrix2 = np.array([[1,2,3],[4,5,6],[1,2,3]])
print(matrix1 @ matrix2)
```

8 Conclusion: NumPy - Your Foundation for Machine Learning

NumPy is a powerful library that provides efficient tools for working with numerical data in Python. It's an essential foundation for machine learning, allowing you to store and manipulate large datasets quickly and easily. By understanding the concepts covered in this guide, you'll be well-equipped to tackle more advanced machine learning tasks.