Learning and practicing NumPy with examples and exercises

Neural Networks and Deep Learning

Prof. José Francisco Ruiz Muñoz
jfruizmu@unal.edu.co
Prof. Richard Rios Patiño
rriospa@unal.edu.co

November 27th, 2024

NumPy, an essential Python library, empowers numerical computation and data manipulation by introducing the ndarray data structure, enabling multi-dimensional arrays and versatile mathematical operations. It facilitates efficient numerical computations through vectorized operations, supports advanced indexing and slicing, and seamlessly integrates with other libraries for data analysis and scientific computing. With its array-centric approach, NumPy serves as a foundation for various applications, from scientific research to machine learning, providing a versatile toolkit for efficient handling of numerical data and computations. It is usually imported as `import numpy as np`

1 Examples: NumPy

1. Basic Indexing:

```
arr = np.array([10, 20, 30, 40, 50])
print(arr[2])  # Output: 30
```

2. Slicing:

```
arr = np.array([10, 20, 30, 40, 50])
print(arr[1:4])  # Output: [20 30 40]
```

3. Indexing 2D Arrays:

```
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr[1, 2])  # Output: 6
```

4. Slicing 2D Arrays:

```
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr[:2, 1:])  # Output: [[2 3] [5 6]]
```

5. 3D Tensor Indexing:

```
tensor = np.array([[[1, 2, 3], [4, 5, 6]],
                   [[7, 8, 9], [10, 11, 12]],
                   [[13, 14, 15], [16, 17, 18]]])

# Access specific elements
print(tensor[1, 0, 2])  # Output: 9

# Access an entire "slice" (2D submatrix)
print(tensor[1])  # Output: [[ 7  8  9] [10 11 12]]
```

6. Slicing a 3D Tensor:

```
tensor = np.arange(27).reshape(3, 3, 3)

# Slice along the first dimension
print(tensor[1])  # Output: 3x3 submatrix

# Slice along the second dimension
print(tensor[:, 1, :])  # Output: 3x3 submatrix

# Slice along the third dimension
print(tensor[:, :, 2])  # Output: 3x3 submatrix
```

7. 4D Tensor Indexing:

```
tensor = np.arange(48).reshape(2, 3, 2, 4)

# Access specific elements
print(tensor[1, 2, 0, 3])  # Output: 47

# Access a whole "block" (3D sub-tensor)
print(tensor[1, 2])  # Output: 2x4x3 sub-tensor
```

8. Slicing a 4D Tensor:

```
tensor = np.arange(48).reshape(2, 3, 2, 4)

# Slice along the first dimension
```

```
print(tensor[1])  # Output: 3x2x4 sub-tensor

# Slice along the second dimension
print(tensor[:, 2, :, :])  # Output: 2x2x4 sub-tensor

# Slice along the third dimension
print(tensor[:, :, 0, :])  # Output: 2x3x4 sub-tensor
```

9. Matrix Addition:

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
result = A + B
print(result)
```

10. Matrix Multiplication:

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
result = np.dot(A, B)
print(result)
```

11. Transpose of a Matrix:

```
A = np.array([[1, 2], [3, 4]])
result = np.transpose(A)
print(result)
```

12. Matrix Determinant and Inverse:

```
A = np.array([[1, 2], [3, 4]])
det = np.linalg.det(A)
inverse = np.linalg.inv(A)
print("Determinant:", det)
print("Inverse:\n", inverse)
```

13. Eigenvalues and Eigenvectors:

```
A = np.array([[1, -1], [1, 3]])
eigenvalues, eigenvectors = np.linalg.eig(A)
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:\n", eigenvectors)
```

14. Solving Linear Equations:

```
A = np.array([[2, 1], [1, 1]])
b = np.array([3, 2])
solution = np.linalg.solve(A, b)
print("Solution:", solution)
```

15. **Gradient descent optimization:** Given the dataset X and y:

$$X = \begin{bmatrix} x^{(1)} \\ x^{(2)} \\ \vdots \\ x^{(m)} \end{bmatrix}, \quad y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

Where m is the number of samples.

The linear regression model is defined as:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

We add a bias term to the features, creating X_b:

$$X_b = \begin{bmatrix} 1 & x^{(1)} \\ 1 & x^{(2)} \\ \vdots & \vdots \\ 1 & x^{(m)} \end{bmatrix}$$

The goal is to minimize the mean squared error:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Gradient Descent updates the parameters θ using the gradients:

$$\theta := \theta - \alpha \nabla J(\theta)$$

Where α is the learning rate.

The gradient $\nabla J(\theta)$ with respect to θ is calculated as:

$$\nabla J(\theta) = \frac{1}{m} X_b^T (X_b \theta - y)$$

We iterate this process for a specified number of iterations.

The final optimized parameters are:

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$$

The gradient descent optimization process can be coded in Python as follows:

```
# Generate some sample data
np.random.seed(0)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Add bias term to X
X_b = np.c_[np.ones((100, 1)), X]

# Hyperparameters
learning_rate = 0.1
n_iterations = 1000

# Initialize the parameters
theta = np.random.randn(2, 1)

# Gradient Descent
for iteration in range(n_iterations):
    gradients = 2 / 100 * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - learning_rate * gradients

print("Final theta:", theta)
```

2 Examples: Python functions and classes

1. Simple Function:

```
def greet(name):
    return "Hello, " + name + "!"

print(greet("Ana"))  # Output: Hello, Ana!
```

2. Function with Parameters and Return:

```
def add_numbers(a, b):
    sum = a + b
    return sum

result = add_numbers(5, 7)
print(result)  # Output: 12
```

3. Function with Default Parameter:

```
def power(base, exponent=2):
    return base ** exponent

print(power(3))        # Output: 9
print(power(2, 3))     # Output: 8
```

4. Function with Multiple Returns:

```
def min_max(numbers):
    return min(numbers), max(numbers)

nums = [4, 9, 2, 7, 5]
minimum, maximum = min_max(nums)
print(minimum, maximum)  # Output: 2 9
```

5. Function with Docstring:

```
def area_of_rectangle(length, width):
    """
    Calculates the area of a rectangle.

    Args:
        length (float): The length of the rectangle.
        width (float): The width of the rectangle.

    Returns:
        float: The area of the rectangle.
    """
    return length * width
```

6. Creating a Simple Class:

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print(f"{self.name} is barking!")

# Create an instance of the Dog class
dog1 = Dog("Buddy", 3)
dog1.bark()  # Output: Buddy is barking!
```

7. Adding Methods and Attributes

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.grades = []

    def add_grade(self, grade):
        self.grades.append(grade)

    def get_average_grade(self):
        return sum(self.grades) / len(self.grades)

# Create a Student instance and add grades
student1 = Student("Alice", 18)
student1.add_grade(85)
student1.add_grade(92)
average_grade = student1.get_average_grade()
print(f"{student1.name}'s average grade: {average_grade}")
```

8. Inheritance

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass  # Placeholder for subclasses to override

class Cat(Animal):
    def speak(self):
        print(f"{self.name} says Meow!")

class Dog(Animal):
    def speak(self):
        print(f"{self.name} says Woof!")

# Create instances of Cat and Dog
cat = Cat("Whiskers")
dog = Dog("Buddy")
cat.speak()  # Output: Whiskers says Meow!
dog.speak()  # Output: Buddy says Woof!
```

3 Exercises

3.1 Indexing Arrays in NumPy

1. Given the NumPy array `arr = np.array([3, 7, 1, 9, 5])`, perform the following indexing operations:

a) Retrieve the third element of the array.

b) Slice the array to get elements from index 1 to index 3.

c) Change the value of the fourth element to 12.

2. Create a 2D NumPy array `matrix` with the following values:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

a) Retrieve the element in the second row and third column.

b) Slice the matrix to get the submatrix consisting of the first two rows and all columns.

3.2 Indexing and Slicing Higher-Dimensional Tensors

Consider the following 3D tensor representing a stack of matrices:

$$\text{tensor} = \begin{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}, \\ \begin{bmatrix} 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \end{bmatrix}, \\ \begin{bmatrix} 25 & 26 & 27 & 28 \\ 29 & 30 & 31 & 32 \\ 33 & 34 & 35 & 36 \end{bmatrix} \end{bmatrix}$$

1. Retrieve the element at the position $(1, 0, 2)$ within the tensor.

2. Slice the tensor to obtain the 3×4 sub-tensor that corresponds to the second "matrix" (slice) along the first dimension.

3. Slice the tensor to obtain the 2×4 sub-tensor that corresponds to the last row of the last two "matrices."

4. Retrieve the element at the position $(2, 1, 3)$ within the tensor.

5. Slice the tensor to obtain the 2×4 sub-tensor that corresponds to the first row of the las two "matrices."

Hint: Remember that in NumPy indexing, the first index corresponds to the outermost dimension, the second index corresponds to the next inner dimension, and so on.

3.3 Defining Functions in Python

1. Write a Python function `calculate_average` that takes a list of numbers as input and returns the average of those numbers.

2. Define a function `find_power` that calculates the result of a number raised to a given exponent. The function should have two parameters: `base` (the base number) and `exponent` (default value should be 2).

3.4 Implementing Gradient Descent Optimization

1. Consider the linear regression problem where you have the following data points:

$$X = [1, 2, 3, 4, 5]$$

$$y = [3, 6, 8, 11, 13]$$

   Implement a Python program using NumPy to perform gradient descent optimization to find the parameters θ_0 and θ_1 for the linear regression model $h_\theta(x) = \theta_0 + \theta_1 x$. Use a learning rate of 0.01 and iterate for 1000 steps.

2. Draw in a plot the progress of the θ_0 and θ_1 parameters through the steps.

3. Explain in your own words the purpose of the learning rate in gradient descent optimization.