Fast & Efficient Python Programming Workshop

Yazeed Balasmeh
Arpan Ghosal

Why Python is slow ?

Dynamic Typing in Python:

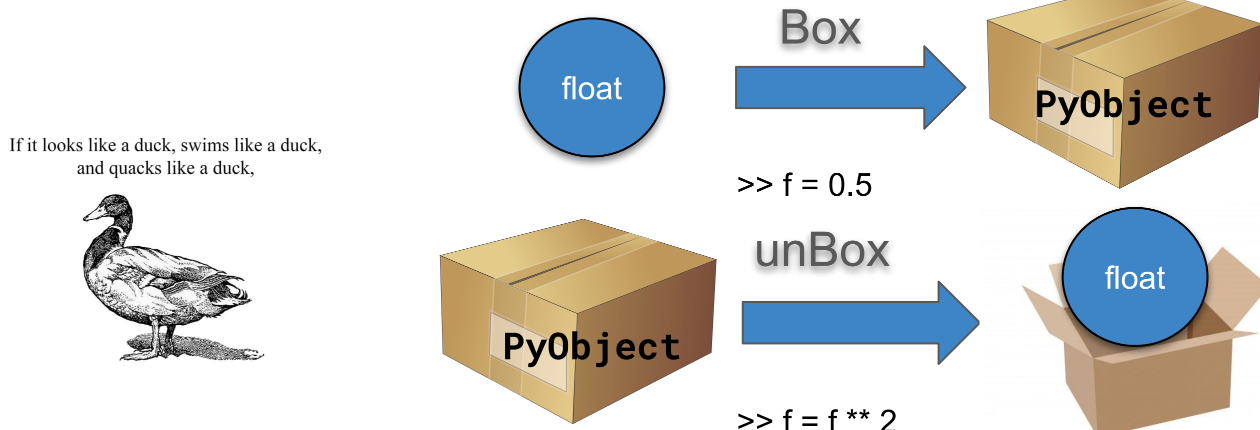
- **Variables are dynamically typed**: They get their type at runtime when values (`PyObject*`) are assigned.

- **Impact on Performance**:
  - This makes it difficult for the interpreter to optimize execution.
  - In contrast, compiled languages allow extensive analysis and optimization before runtime.

Why Python is slow ?

Dynamic Typing in Python:

- Python has only one data type, `PyObject*` with a pointer to its runtime type, which is yet another `PyObject*`.
- Python is a dynamically typed language (Duck Typing). It wraps and later unwraps objects (referred to as boxing/unboxing).



Why Python is slow ?

Flexible Data Structures in Python

- **Built-in Structures**: Python's built-ins (e.g., lists, dictionaries) are highly flexible and versatile.
- **Trade-offs**:
  - Generic nature, less efficient for numerical computations.
  - Perform well when processing diverse data types, but introduce significant overhead when handling large amounts of uniform data.

Flexible nature of Python

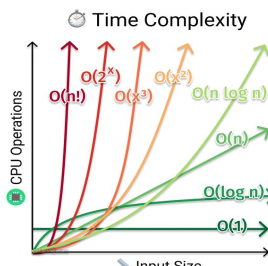
Enhances programmer productivity

Causes performance problems

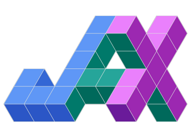
What to know, before optimising your code for acceleration …

- ❏ How do we measure performance?

- ❏ Navigating algorithmic complexity



- ❏ Tools that help achieve this



Performance Analysis

➔ **Execution Time**

A simple example:

```
def fact(n):
    product = 1
    for i in range(n):
        product = product * (i+1)
    return product
```

```
def fact2(n):
    if n == 0:
        return 1
    else:
        return n * fact2(n-1)
```

timeit

10 µs ± 183 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

30.5 µs ± 8 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

Performance Analysis

Why Execution Time Isn't the Best Metric for Algorithmic Complexity ?

- **Execution time** alone is not a reliable measure of algorithmic complexity because it depends on external factors like **hardware**, **system load**, and **compiler optimizations**.

- Objective complexity analysis requires a more standardized approach, such as evaluating the algorithm's asymptotic behavior (**Big O notation**), which focuses on **input size** rather than specific **execution time**.

- Instead of timing, we need to consider a **line-by-line evaluation** of the algorithm’s structure, breaking down operations to assess how the number of steps grows as input size increases.

Performance Analysis

➔ **Line-by-line evaluation**

- Python provides the ability to evaluate your code line by line, allowing you to identify **bottlenecks**.
- By pinpointing these bottlenecks, we can optimize our code, leading to increased **efficiency** and **faster performance**.
- Python offers several profiling libraries. We will focus on two key libraries: **cProfile** and **line_profiler**.

Performance Analysis

➔ **Line-by-line evaluation**

Function which sorts a list of elements using the **bubble sort algorithm**.

```
import cProfile
#Here we are using CProfile

def bubble_sort(a):
    n = len(a)
    for i in range(n):
        for j in range(n - i - 1):
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]
    return a
```

Total time: 15.256 seconds

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	15.255	15.255	15.255	15.255	<ipython-input-25-a6ab47c635ba>:7(bubble_sort)
1	0.000	0.000	15.255	15.255	<string>:1(<module>)
1	0.000	0.000	15.256	15.256	{built-in method builtins.exec}
1	0.000	0.000	0.000	0.000	{built-in method builtins.len}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

Performance Analysis

➔ **Line-by-line evaluation**

Function which sorts a list of elements using the **bubble sort algorithm**.

```
from line_profiler import LineProfiler
#Here we are using Line_Profile

def bubble_sort(a):
    n = len(a)
    for i in range(n):
        for j in range(n - i - 1):
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]
    return a
```

Total time: 58.4391 s

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     8                                           def bubble_sort(a):
     9                                               """
    10                                               Perform bubble sort on a list of elements.
    11                                               """
    12         1       3056.0   3056.0      0.0      n = len(a)
    13     10001    3809427.0    380.9      0.0      for i in range(n):
    14  50005000        1e+10    290.5     25.8          for j in range(n - i - 1):
    15  49995000        2e+10    481.2     42.7              if a[j] > a[j + 1]:
    16  24963051        2e+10    711.5     31.5                  a[j], a[j + 1] = a[j + 1], a[j]
    17         1        523.0    523.0      0.0      return a
```

Performance Analysis

➔ **Line-by-line evaluation**

Profiling Overhead

- **CProfile**: This profiler operates with **lower overhead**, giving a general overview of function execution times. It allows for efficient profiling without significantly affecting the performance of the code being analyzed.
- **Line Profiler**: In contrast, this profiler incurs higher overhead because it tracks execution time for each individual line of code. As a result, the reported execution times may be **longer due to the additional processing required**.

Measurement Focus

- **CProfile**: This tool **measures the total execution time of function calls**, which **includes the time spent in any sub-functions**. Consequently, this can create the impression that functions are faster than they actually are since it aggregates all execution time into a single measurement.
- **Line Profiler**: This profiler concentrates on the **time taken by each line of code**, offering detailed insights into performance inefficiencies. It is particularly **useful for identifying bottlenecks within loops and complex sections** of code.

CProfile provides broad overview of function performance, **Line Profiler** delivers in-depth insights into execution time of each line of code.

Performance Analysis

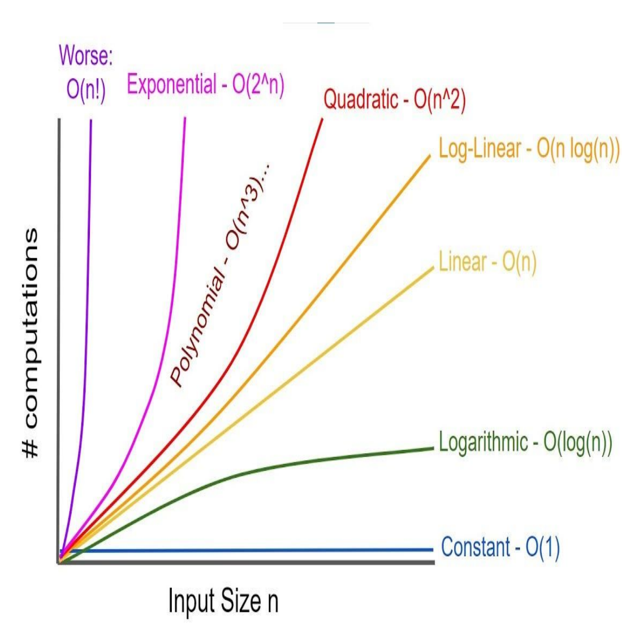
➔ **Big-O Notation**

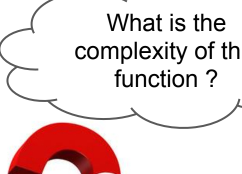
Big-O notation describes the relationship between the size of the input to an algorithm and the number of steps required to execute it.

Unlike measuring performance for a specific instance (such as calculating `fact(50)`), Big-O focuses on how well an algorithm scales with:

1. **Increasing Input Size**: How the algorithm's performance changes as the input size grows.
2. **Type of Input**: How the algorithm's efficiency varies with different types of input data.

This approach provides a more comprehensive evaluation metric than assessing concrete execution time for a specific case.

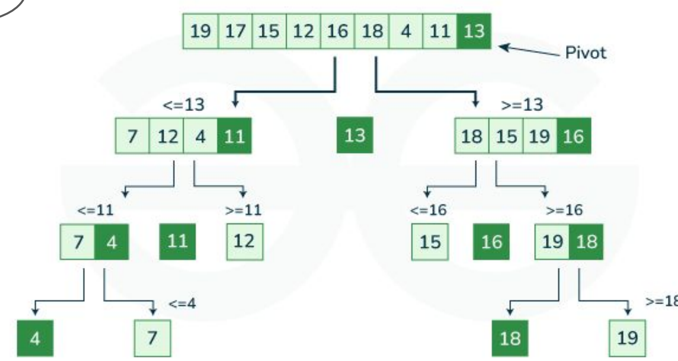




Performance Analysis

➔ **Big-O Notation**

Simple Example: **Quick Sort**



At each level of recursion, we make comparisons and partition the array, which takes **O(n)** time. The recursion proceeds for approximately **log(n)** levels, because the **array size halves with each step**.

Therefore, the time complexity of Quicksort is **O(n log n)**.

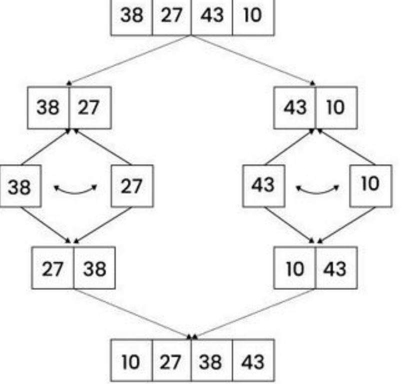
```
def QuickSort(arr):
        if len(arr) <= 1:
                return arr
        pivot = arr[len(arr) // 2]
        left = [x for x in arr if x < pivot]
        middle = [x for x in arr if x == pivot]
        right = [x for x in arr if x > pivot]
        return QuickSort(left) + middle + QuickSort(right)
```

Performance Analysis

➔ **Big-O Notation**

Simple Example: **Merge Sort**

Merge Sort works by **dividing** the array into two halves.
Each half is **recursively sorted**.
The two sorted halves are then **merged** back together into a single sorted array.
It's stable and guarantees O(n log n) time complexity but requires extra space for merging.




```
def merge_sort(arr):
    if len(arr) > 1:
    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]

    # Recursively sort both halves
    merge_sort(left)
    merge_sort(right)

    i = j = k = 0

    # Merge the two halves
    while i < len(left) and j < len(right):
    if left[i] < right[j]:
    arr[k] = left[i]
    i += 1
    else:
    arr[k] = right[j]
    j += 1
    k += 1

    # Check if any element was left in the left half
    while i < len(left):
    arr[k] = left[i]
    i += 1
    k += 1

    # Check if any element was left in the right half
    while j < len(right):
    arr[k] = right[j]
    j += 1
    k += 1
```

Performance Analysis

➔ **Big-O Notation**

1. Memory Usage:

- **Quick Sort** is an **in-place sorting algorithm**, meaning it doesn't need extra memory for temporary arrays, while **Merge Sort** requires additional space for merging. This makes Quick Sort more efficient in terms of memory usage, especially for large datasets.

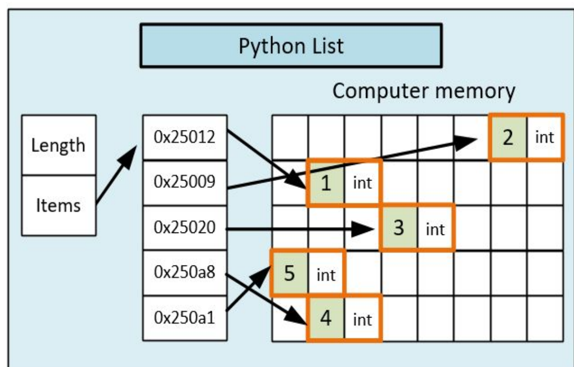
2. Cache Efficiency:

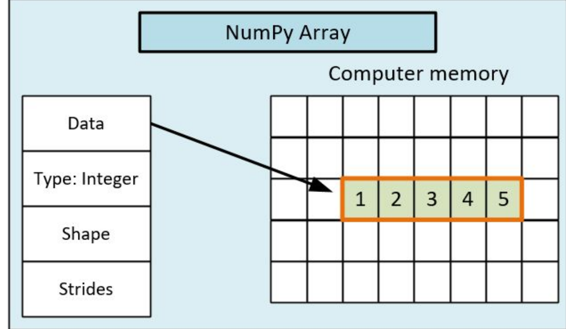
- **Quick Sort** has better **cache locality**, accessing memory sequentially and utilizing modern CPU cache more effectively. **Merge Sort** accesses memory in a scattered way, leading to more cache misses and slower performance.

MergeSort vs QuickSort Time Complexity
QuickSort Time Complexity O(n log n)
MergeSort Time Complexity O(n log n)
Time (seconds)
0
1
2
3
4
5
0
100000
200000
300000
400000
500000
Array Size

A bit about Numpy

- Stores **data in continuous memory blocks** for high performance.
- Allows operations on entire arrays/lists without explicit loops for increased speed (**Vectorization**).
- Supports operations on arrays of different sizes without manual size adjustment (**Broadcasting**).
- Provides fast, element-wise array operations (**ufuncs**).
- Integrates well with **Pandas** and **SciPy**.
- Able to perform complex mathematical computations such as linear algebra and Fourier transforms.
- Uses **less memory compared to traditional Python lists**, with precise control over data types.
- **Consistent** over different platforms and OS.
- Can be extended with C or Fortran for performance-critical tasks.



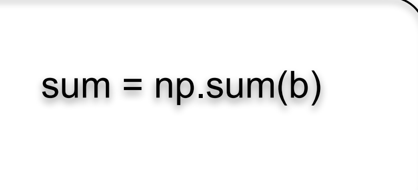


Performance Analysis

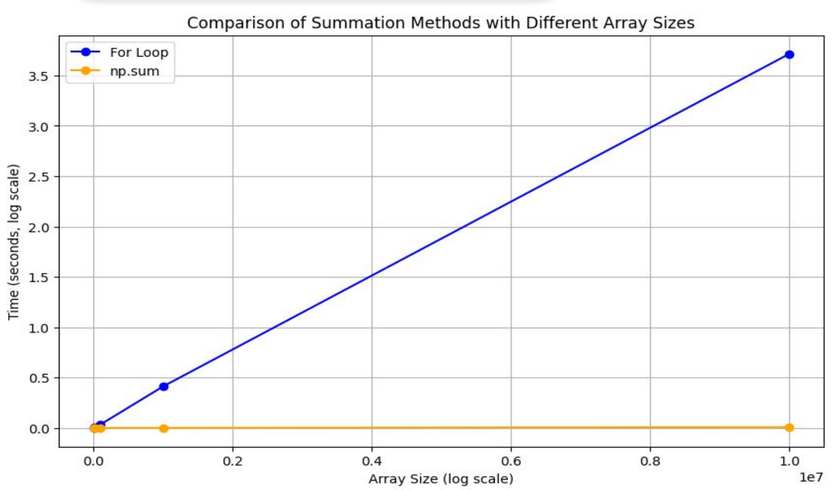
➔ **CPU Usage**

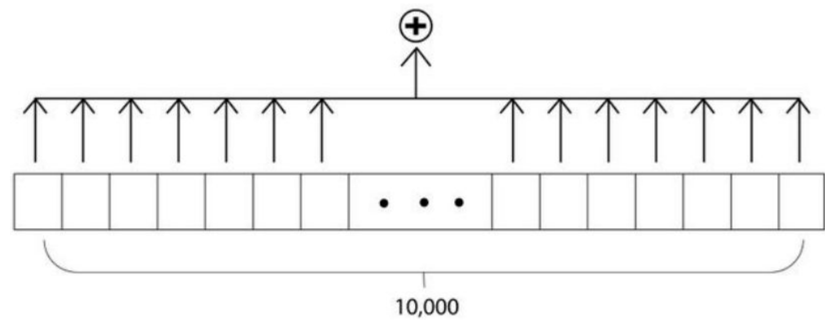
```
sum = 0
for i in range(b.shape[0]):
        sum += i
```

versus



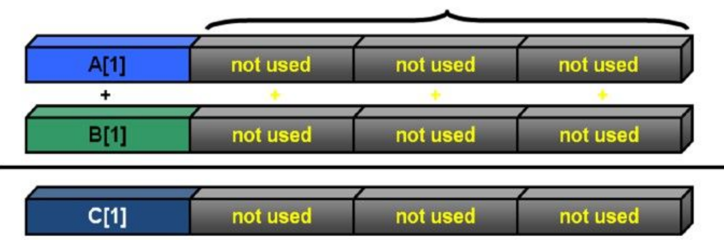
```
sum = np.sum(b)
```



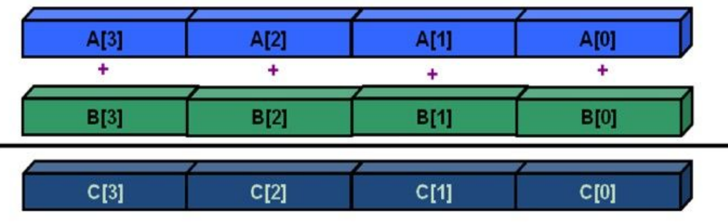


Vectorization

Vectorization refers to the process of converting operations that typically process one element at a time, such as those in a loop, into operations that process multiple elements simultaneously.



Non-Vectorized



Vectorized

- **Key aspects:**
  - **Data parallelism** utilizing SIMD architecture (as implemented in GPUs).
  - Contiguous memory management.
- **Benefits:**
  - Performance improvements by reducing the number of interpreted loops in high-level languages.
  - Lower memory footprint by minimizing temporary variable storage.
  - Cleaner and more concise code, better readability.
  - Utilized in tons of libraries (some we will discuss today).

Loop-wise versus Vectorization

```
# Loop-based calculation

# Create two arrays of size 1,000,000
a = np.random.rand(1_000_000)
b = np.random.rand(1_000_000)

# Loop-wise multiplication
def loop_wise_multiplication(a, b):
    result = np.empty_like(a)  # Initialize an empty array with the same shape as 'a'
    for i in range(len(a)):
        result[i] = a[i] * b[i]
    return result

# Time the loop-wise multiplication
%timeit c = loop_wise_multiplication(a, b)

# Display the first 5 elements of the result
c = loop_wise_multiplication(a, b)  # Call the function to get the result
print("First 5 elements of the result:", c[:5])
```

```
246 ms ± 11.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
First 5 elements of the result: [0.37046711 0.0320318  0.58237442 0.07603239 0.03643322]
```

```
# Vectorization on small set

# Create two arrays of size 10,000
a = np.random.rand(10000)
b = np.random.rand(10000)

# Vectorized multiplication
%timeit c = a * b  # This line is vectorized

# Display the first 5 elements of the result
print("First 5 elements of the result:", c[:5])
#print(timeit.time())
```

```
4.02 µs ± 243 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
First 5 elements of the result: [0.26792758 0.61447971 0.01041451 0.07903875 0.05245871]
```

```
# Vectorization on larger set

# Create two arrays of size 1,000,000
a = np.random.rand(1_000_000)
b = np.random.rand(1_000_000)

# Vectorized multiplication
%timeit c = a * b  # This line is vectorized

# Display the first 5 elements of the result
print("First 5 elements of the result:", c[:5])
#print(timeit.time())
```

```
1.39 ms ± 90.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
First 5 elements of the result: [0.26792758 0.61447971 0.01041451 0.07903875 0.05245871]
```

Broadcasting

Broadcasting simplifies the handling of arrays with different dimensions by automatically 'broadcasting' the smaller array across the larger one so that they have compatible shapes.

(4, 3)

0	0	0
1	1	1
2	2	2
3	3	3

\+

(3,)

1	2	3
1	2	3
1	2	3
1	2	3

=

(4, 3)

1	2	3
2	3	4
3	4	5
4	5	6

```
# Define a 3x3 matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Define a 1x3 vector
vector = np.array([1, 0, 100])

# Add the vector to each row of the matrix using broadcasting
result = matrix + vector

print("\n--Example code:--\n")
print("\nTake 3x3 matrix:\n", matrix)
print("\nAdd 1-D vector: \n", vector)
print("\nOuput after broadcasting:\n",result)
```

```
--Example code:--


Take 3x3 matrix:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]

Add 1-D vector:
 [  1   0 100]

Ouput after broadcasting:
 [[  2   2 103]
 [  5   5 106]
 [  8   8 109]]
```

- **Key aspects:**
  - Avoids explicit data replication, thus minimizing memory usage.
  - **Smaller array is "broadcast" across the larger array** so that they appear to have the same shape.
- **Rules for broadcasting:**
  - To deem which two arrays are suitable for operations, NumPy compares the shape of the two arrays dimension-by-dimension starting from the trailing dimensions, working it's way forward. (from right to left)
  - Two dimensions are said to be compatible if **both of them are equal, or either one of them is 1.**
  - If both the dimensions are unequal and neither of them is 1, then NumPy will throw an error and halt.

Auto Differentiation

Automatic differentiation (AD or autograd) is a set of techniques to evaluate the derivative of a function with a mix of numerical and symbolic approaches.

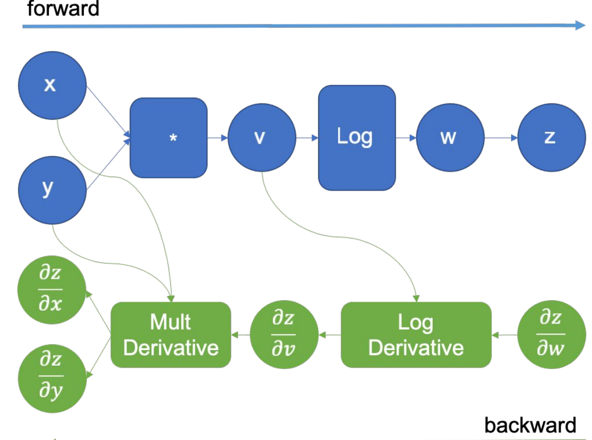
Every computer program executes a sequence of elementary arithmetic operations and elementary functions.

By **applying the chain rule repeatedly** to these operations, derivatives of arbitrary order can be computed automatically and efficiently.

Widely implemented in libraries - JAX (uses XLA to run on GPUs), integrated autograd systems in Tensorflow and PyTorch etc.

How Autograd Works?

- Autograd can be implemented using two main methods:
  - **Forward mode AD**: Computes derivatives from the input towards the output. Suitable when there are fewer inputs than outputs.
  - **Reverse mode AD** (often used in deep learning): Computes derivatives from the output back to the inputs. Suitable when there are fewer outputs than inputs, as in the case of a loss function in neural networks.
- **Computation Graph**: AD involves constructing a computation graph where **nodes represent operations or variables** and **edges represent dependencies between these operations**. Forward pass computes the values, and the backward pass propagates derivatives.



Autodifferentiation example using TensorFlow

Minimizing a Quadratic Function with TensorFlow

```python
# Define a variable for our starting point
x = tf.Variable(0.0, name='x')
```

```python
# Define the function f(x) = (x - 3)^2
def f(x):
    return (x - 3) ** 2
```

```python
optimizer = tf.optimizers.SGD(learning_rate=0.1) # Stochastic Gradient Descent as optimizer

steps = 50         # steps or iterations for gradient descent


x_values, f_values = [], [] # Lists to store the values of x and f(x) for plotting
```

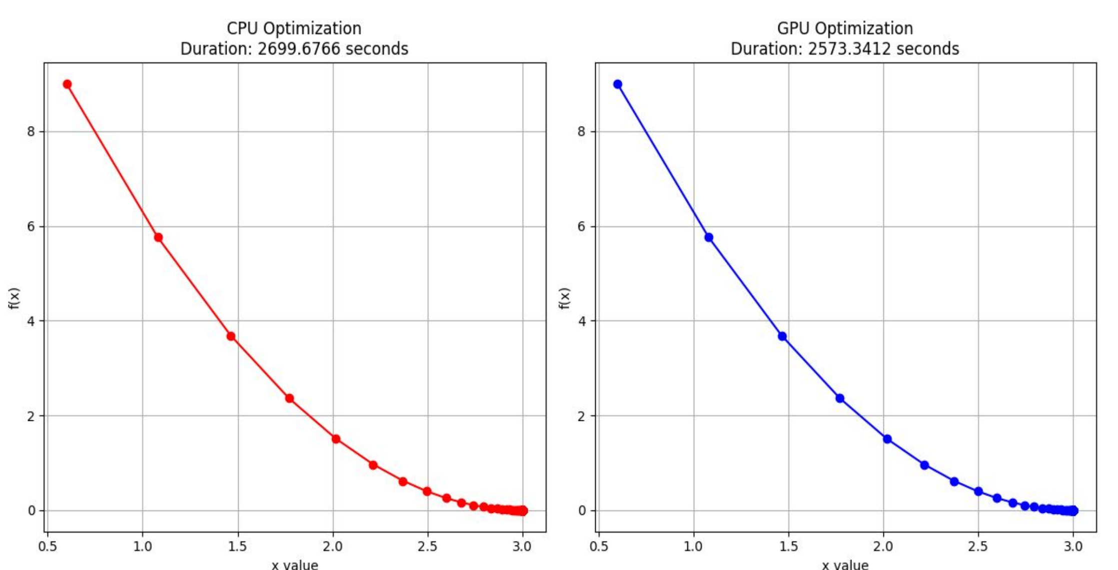
tf.GradientTape(): context manager records the operations performed on TensorFlow variables for automatic differentiation

```python
# Perform gradient descent
for step in range(steps):
    with tf.GradientTape() as tape:
        # Monitor  'x'
        tape.watch(x)
        y = f(x)

    # Get the gradient of f with respect to x
    grad = tape.gradient(y, x)

    # Update x to minimize f(x)
    optimizer.apply_gradients([(grad, x)])
    # Store the values for visualization
    x_values.append(x.numpy())
    f_values.append(y.numpy())

    # Print the current value of x and f(x)
    print(f"Step {step + 1}: x = {x.numpy()}, f(x) = {y.numpy()}")


# Final output
print(f"\nOptimal x found: {x.numpy()}")
```

```
Step 1: x = 0.6000000238418579, f(x) = 9.0
Step 2: x = 1.0800000429153442, f(x) = 5.760000228881836
Step 3: x = 1.4639999866485596, f(x) = 3.6863999366760254
Step 4: x = 1.7711999416351318, f(x) = 2.3592960834503174
Step 5: x = 2.0169599056243896, f(x) = 1.5099495649337769
Step 6: x = 2.2135679721832275, f(x) = 0.9663678407669067
Step 7: x = 2.370854377746582, f(x) = 0.6184753179550171
Step 8: x = 2.4966835975646973, f(x) = 0.39582422375679016
Step 9: x = 2.597346782684326, f(x) = 0.2533273994922638
Step 10: x = 2.677877426147461, f(x) = 0.16212961077690125
Step 11: x = 2.7423019409179688, f(x) = 0.10376295447349548
Step 12: x = 2.793841600418091, f(x) = 0.06640829145908356
Step 13: x = 2.835073232650757, f(x) = 0.042501285672187805
Step 14: x = 2.868058681488037, f(x) = 0.027200838550925255
Step 15: x = 2.894446849822998, f(x) = 0.01740851067006588
Step 16: x = 2.915557384490967, f(x) = 0.011141467839479446
Step 17: x = 2.932446002960205, f(x) = 0.007130555342882872
Step 18: x = 2.9459567070007324, f(x) = 0.004563542548567057
Step 19: x = 2.9567654132843018, f(x) = 0.0029206774197518826
Step 20: x = 2.9654123783111572, f(x) = 0.0018692294834181666
Step 21: x = 2.97232985496521, f(x) = 0.0011963035212829710
Step 22: x = 2.9778637886047363, f(x) = 0.0007656369125470519
Step 23: x = 2.9822909832000732, f(x) = 0.0004900118801742792
Step 24: x = 2.985832691192627, f(x) = 0.0003136092855129391
Step 25: x = 2.98866605758667, f(x) = 0.0002007126458920538‌4
Step 26: x = 2.9909329414367676, f(x) = 0.00012845825403928757
Step 27: x = 2.992746353149414, f(x) = 8.221155439969152e-05
Step 28: x = 2.994197130203247, f(x) = 5.261539263301529e-05
Step 29: x = 2.9953577518463135, f(x) = 3.367329918546602e-05
Step 30: x = 2.996286153793335, f(x) = 2.155046786356251‌7e-05
Step 31: x = 2.9970288276672363, f(x) = 1.3792653589916881e-05
Step 32: x = 2.9976229667663574, f(x) = 8.827864803606644e-06
Step 33: x = 2.998098373413086, f(x) = 5.650286766467616e-06
Step 34: x = 2.998478651046753, f(x) = 3.616183676058426‌5e-06
Step 35: x = 2.9987828731536865, f(x) = 2.3145025807025377e-06
Step 36: x = 2.999026298522949, f(x) = 1.4813977031735703e-06
Step 37: x = 2.999221086502075, f(x) = 9.480945664108731e-07
Step 38: x = 2.9993767738342285, f(x) = 6.067062372494547e-07
Step 39: x = 2.9995014667510986, f(x) = 3.884108537022257e-07
Step 40: x = 2.999601125717163, f(x) = 2.4853540026015253e-07
Step 41: x = 2.999680995941162, f(x) = 1.5910069350866252e-07
Step 42: x = 2.9997448921203613, f(x) = 1.0176358955504838e-07
Step 43: x = 2.999795913696289, f(x) = 6.50800302537391e-08
Step 44: x = 2.9998366832733154, f(x) = 4.16512193623930‌2e-08
Step 45: x = 2.9998693466186523, f(x) = 2.667235321496264e-08
Step 46: x = 2.9998955726623535, f(x) = 1.707030605757609e-08
Step 47: x = 2.9999165534973145, f(x) = 1.0905068847932853e-08
Step 48: x = 2.9999332427978516, f(x) = 6.963318810448982e-09
Step 49: x = 2.9999465942382812, f(x) = 4.456524038687348e-09
Step 50: x = 2.999957323074341, f(x) = 2.852175384759903e-09

Optimal x found: 2.999957323074341
```

Auto Differentiation - Comparison

Grad descent calculated over 1M steps.
GPU starts to performs better as number of steps increases.




```python
# Function to perform gradient descent with timing
def optimize_function(use_gpu=True):
    # Optionally place computation on the GPU
    device = '/GPU:0' if use_gpu else '/CPU:0'
    with tf.device(device):
        # Define a variable for our starting point
        x = tf.Variable(0.0, name='x')

        # Define the function f(x) = (x - 3)^2
        def f(x):
            return (x - 3) ** 2

        # Set up the optimizer
        optimizer = tf.optimizers.SGD(learning_rate=0.01)

        # Number of steps for gradient descent
        steps = 1_000_000

        # Lists to store the values of x and f(x) for plotting
        x_values, f_values = [], []

        # Timing start
        start_time = time.time()

        # Perform gradient descent
        for step in range(steps):
            with tf.GradientTape() as tape:
                tape.watch(x)
                y = f(x)

            grad = tape.gradient(y, x)
            optimizer.apply_gradients([(grad, x)])

            # Store the values for visualization
            x_values.append(x.numpy())
            f_values.append(y.numpy())

        # Timing end
        duration = time.time() - start_time

    return x_values, f_values, duration
```