Information systems research

UDC 004.438:67 doi: https://doi.org/10.20998/2522-9052.2024.1.05

Serhii Krivtsov[1], Yurii Parfeniuk[2], Kseniia Bazilevych[1], Ievgen Meniailov[2], Dmytro Chumachenko[1]

[1] National Aerospace University “Kharkiv Aviation Institute”, Kharkiv, Ukraine
[2] V.N. Karazin Kharkiv National University, Kharkiv, Ukraine

PERFORMANCE EVALUATION OF PYTHON LIBRARIES FOR MULTITHREADING DATA PROCESSING

Abstract: Topicality. The rapid growth of data in various domains has necessitated the development of efficient tools and libraries for data processing and analysis. Python, a popular programming language for data analysis, offers several libraries, such as NumPy and Numba, for numerical computations. However, there is a lack of comprehensive studies comparing the performance of these libraries across different tasks and data sizes. **The aim of the study.** This study aims to fill this gap by comparing the performance of Python, NumPy, Numba, and Numba.Cuda across different tasks and data sizes. Additionally, it evaluates the impact of multithreading and GPU utilization on computation speed. **Research results.** The results indicate that Numba and Numba.Cuda significantly optimizes the performance of Python applications, especially for functions involving loops and array operations. Moreover, GPU and multithreading in Python further enhance computation speed, although with certain limitations and considerations. **Conclusion.** This study contributes to the field by providing valuable insights into the performance of different Python libraries and the effectiveness of GPU and multithreading in Python, thereby aiding researchers and practitioners in selecting the most suitable tools for their computational needs.

Keywords: machine learning; Python; GPU; multithreading; numerical computations optimization.

Introduction

In solving problems in Data Science that depend on the availability of necessary data, processing large volumes of information becomes an increasingly important task [1]. However, with the increase in data volumes, there is a need for accelerated processing and multithreading data processing [2]. Python libraries, such as Numba [3], Numpy [4], CPython [5], Numba.Cuda [6], can be not only convenient but also an effective solution for such tasks.

Python is one of the most popular programming languages in the field of Data Science, and its main advantage is that it provides powerful tools for working with data. Libraries like Numpy and Pandas [7] provide various tools for working with data arrays and tables. However, multithreading processing may be necessary for processing large volumes of data.

This paper examines Python libraries, such as Numba, Numpy, CPython, and Numba.Cuda, for multithreading data processing in Data Science tasks. We will analyze their advantages and disadvantages and consider examples of using the mentioned libraries for data processing in Data Science tasks.

Particular attention should be paid to the use of these libraries for processing COVID-19 data, as the data is quite voluminous and reliable, and it is also essential that there is an opportunity to evaluate the processing results by comparing them with real values [8]. Assuming that this pandemic was not the last, and the processing and analysis of large volumes of data were critically essential tasks, the acceleration and optimization of this process have immense practical significance [9]. Python libraries can help speed up data processing and make it more efficient.

Overall, this article will provide readers with information on how the use of Python libraries can improve multithreading data processing in Data Science tasks and help assess the advantages and disadvantages of the mentioned libraries.

The primary aim of this research was to evaluate and compare the performance of different computational tools, namely Python, NumPy, Numba, and Numba.Cuda, in the context of specific Data Science applications. The study sought the most efficient tool for various tasks, considering different data sizes and computational requirements. The following tasks were formulated to achieve the aim:

- Implement solutions for three distinct applied tasks in Data Science using Python, NumPy, Numba, and Numba.Cuda. The selected tasks included regression analysis, image normalization, and activation function computation.
- Evaluate the performance of each implemented solution by measuring the execution time for different data sizes. This involved analyzing the execution time of each tool for varying sizes of datasets and determining the impact of data size on the performance of each tool.
- Compare the performance of the different tools based on the execution time and identify the most efficient tool for each specific task. This involved a detailed analysis of the results obtained from the performance evaluation to determine the optimal tool for each task and data size.
- Analyze the potential benefits and limitations of using multithreading and GPU computations in Python. This involved evaluating the impact of multithreading and GPU computations on the performance of the implemented solutions and identifying potential challenges and limitations associated with their use.

- Provide recommendations on the optimal computational tool for different Data Science applications based on the performance evaluation and comparison results. This involved synthesizing the study's findings to provide practical guidelines for researchers and practitioners in selecting the most appropriate tools for their specific tasks and computational requirements.

The research makes a significant and respectful contribution to the academic community and the field of Data Science by providing a comprehensive and systematic evaluation of widely used computational tools, including Python, NumPy, Numba, and Numba.Cuda. By implementing solutions for three distinct applied tasks, evaluating their performance across different data sizes, and analyzing the impact of multithreading and GPU computations, this study offers valuable insights into the optimal selection of computational tools for various applications. Furthermore, the research provides practical recommendations for researchers and practitioners, aiding in selecting the most appropriate tools for specific tasks and computational requirements. Ultimately, this work contributes to the ongoing efforts to optimize computational efficiency in Data Science applications, facilitating more robust and efficient analyses in this ever-evolving field.

Research is part of a complex intelligent information system for epidemiological diagnostics, the concept of which is discussed in [10].

Background

The exponential growth of data in various fields, such as finance [11], healthcare [12], sustainability [13] and social media [14], has posed significant challenges in data storage, processing, and analysis. The ability to quickly and efficiently process large volumes of data is crucial for timely decision-making and gaining insights from the data [15]. This has led to the development of various tools and libraries that aim to optimize computational performance and facilitate data analysis.

Due to its ease of use and extensive library support, Python has become the go-to language for many data scientists and researchers. NumPy, a foundational package for numerical computing in Python, has been widely adopted for its array object and related functionalities. However, as the size of the datasets and the complexity of the computations increase, the limitations of NumPy in terms of performance become more apparent [16]. This has led to developing other libraries, such as Numba, which aims to optimize Python code for better performance.

Numba leverages the LLVM compiler infrastructure to translate Python code into optimized machine code at runtime. This is particularly beneficial for functions that involve heavy use of loops and array operations, which are common in numerical computations [17]. Additionally, Numba. Cuda extends the capabilities of Numba by allowing developers to write CUDA code in Python, thereby enabling the utilization of NVIDIA GPUs for general-purpose processing [18]. This is a significant advancement as GPUs' parallel processing capabilities can significantly reduce computation time for specific tasks [19].

CUDA, on the other hand, is a parallel computing platform and API model developed by NVIDIA [20]. It provides a comprehensive set of programming tools and APIs that enable developers to write software that can leverage the parallel processing capabilities of NVIDIA GPUs. This is especially important for applications involving large datasets or computationally intensive tasks, as GPUs' parallel processing capabilities can significantly reduce computation time [21].

Despite the availability of these tools and libraries, there needs to be more comprehensive studies that compare their performance across different tasks and data sizes. Moreover, the impact of using GPU over CPU for computations and the effectiveness of multithreading in Python, are areas that require further exploration. This study aims to fill this gap by comparing the performance of Python, NumPy, Numba, and Numba.Cuda across different tasks and data sizes. Additionally, it will evaluate the impact of multithreading and GPU utilization on computation speed, thereby providing valuable insights for researchers and practitioners in data science and computational research.

Materials and methods

Let us consider three applied tasks from the field of Data Science, based on which the above-described libraries and processing methods will be tested. All these tasks are also used in the medical field, for example, in predicting a patient's diagnosis [22] or transforming an X-ray image for more qualitative diagnostics [23].

The first is the regression task – correlating a set of features with some number predicting a continuous variable. In mathematical terms, regression by the gradient descent method optimizes the target function L. In the current task, the function J can be defined as the sum of squared errors (SSE):

$$L(w) = \frac{1}{2}(y\varphi(z)). \qquad (1)$$

Based on gradient descent, weights can be updated by calculating the gradient of function J. The weight change is defined as $w = w + \Delta w$.

The value of the weight change Δw is defined as $\Delta w = n \Delta L(w)$. The gradient of function L is the partial derivative of L concerning each weight w_i.

Let us consider taking the derivative concerning each weight:

$$\frac{dL}{dw} = \frac{d}{dw2}\big(y - \varphi(z)\big) =$$
$$= \frac{dL}{dw} = \frac{1d}{2dw}\big(y - \varphi(z)\big) = \qquad (2)$$
$$= \big(y - \varphi(z)\big)\frac{d}{dw}(y - wx) = (y - \varphi(z))x.$$

Thus, the weight update is carried out according to the rule:

$$\Delta w = \sum_{i=1}^{n}(y - \varphi(z))x_j. \qquad (3)$$

Thus, we can iteratively update the weights to minimize the cost function, knowing the function's derivative. In this process, there are some nuances, such as overfitting, underfitting, the curse of dimensionality, etc., but their discussion is beyond the scope of this article. Testing of technologies will be carried out as follows: for a generated set of values, which has the form (BATCHSIZE, SAMPLESIZE), perform a single weight update. The dependence of the computation time on BATCHSIZE will be the final result.

The second task is image normalization. Medical data is often represented in images (X-rays, ultrasound results, etc.). For a more successful application of Convolutional Neural Network (CNN) - the primary method of image processing in deep learning, the image (or rather its representation as an array of numbers) needs to be normalized, that is, to bring all values in the array to the range [0, 1]. For images, the normalization process is significantly simplified - it consists of dividing each array value by 255 (in the RGB format, 255 is the maximum value of the color channel).

If an image of size 1024x1024 has three channels (red, green, blue), then the array, which represents the image in numerical form, will have the shape (1024, 1024, 3). Similarly, an array of such image-size 100 will have the shape (100, 1024, 1024, 3).

The dependence of the execution time of the image array normalization on the number of images in the array is the final result, based on which any conclusions can be made.

The last task is using the activation function for an array of numbers. The activation function of a neural network layer (also used, for example, in logistic regression) is often a nonlinear transformation of spaces to highlight new features. In this section, we will use the sigmoid function (the inverse function of the logit function), which is used in classification problems due to the ease of calculating its derivative. This function looks as follows:

$$\mathrm{f(x)} = \frac{1}{1+e^{-x}}. \tag{4}$$

This function takes values from 0 to 1 since:

$$\lim_{x\to\infty} f(x) = \frac{1}{1+\infty} = 0; \tag{5}$$

$$\lim_{x\to\infty} f(x) = \frac{1}{1+0} = 0. \tag{6}$$

We will compute this function from an array of numbers vectorized, that is, in the following way:

$$M = (m_1, m_2, \dots, m_n); \tag{7}$$

$$f(M) = (f(\mathrm{m}_1), f(m_2), \dots, f(m_n)). \tag{8}$$

The result of the study is the dependence of the execution time of the vectorized operation on the size of the array.

There are several tools for working with such data.

CPython is the standard Python language interpreter distributed on the official website python.org. All aspects of this interpreter's internal workings are beyond this article's scope, but the general concept needs to be stated. Python uses the Everything-Is-Object system, as well as dynamic typing. Code execution is carried out by interpreting the program line by line and building bytecode, which is then converted into machine code and executed on the user's machine. All of the above significantly slows down the speed of work (which is compensated by other advantages), making Python unsuitable for large calculations. However, the conciseness of the language and the ability to create a high-quality and simple API led to the creation of various compiling libraries based on C, Fortran, and LLVM. Since the publication's research subject is functions and their execution time, it is necessary to provide an example of implementing a function in the Python language. The logit(p) function is implemented in the example below:

```
From math import log
def logit(p:float) -> float: return log(p / (1 - p))
```

Importantly, annotating the arguments of the function and the returned value is optional (that is, it is not typing) and exists only to improve the quality of the code. The function is called standardly, using round brackets (in the internal implementation of Python for this purpose, the "magic" method call() is used):

```
>>> logit(0.5)
-0.47712125472
```

It is also worth considering how to use the logit function vectorized. For this, list comprehension or genexp (generator expression) is used. An example of these tools for a list and a tuple is given below:

```
>>>  user_list:list  =  [1,  2,  3,  4]
>>> [logit(p) for  p in  user_list] [0.0, 0.693147,
1.098612, 1.386294]
>>>  user_tuple:tuple  =  (1,  2,  3,  4)
```

Next, let us consider working with NumPy (Numeric Python) – an open-source library for Python based on functions implemented in C and Fortran languages. NumPy provides the ability to perform vectorized operations on numpy.ndarray – the main class of the library, representing an array with various dimensions. NumPy functions that work with the array vectorized are called u-funcs. For the study, it is worth considering only the functions div, mul, add, and exp, which are replacements for the division, multiplication, addition, and exponent operators. Moreover, NumPy provides the ability for very fast calculations due to its implementation. Let us consider an example of a function that calculates the exponent of the inverse argument:

```
import numpy as np
def rev_exp(x:np.ndarray) -> np.ndarray: return np.exp(-x)
```

As you can see, there are no special differences in implementing functions. However, now we can perform operations on arrays of numbers vectorized, that is:

```
>>> import numpy as np
>>> user_arr:np.ndarray = np.arange(1, 6, 1)
>>> user_arr
array([1, 2, 3, 4, 5])
>>> rev_exp(user_arr)
array([0.367879, 0.135335, 0.049787, 0.018315, 0.006737])
```

Also, NumPy provides a convenient interface for changing the shape of the array. For example, to create a column vector from a one-dimensional array of numbers, you can use the following method:

```
>>> np.reshape(user_arr, (-1, 1)) array([[1], [2], [3], [4], [5]])
```

It is also worth noting that functions such as np.add, np.reshape, and many others are alternatives to the methods of the np.ndarray class .add(), .reshape(), etc.

However, calling methods implies calling the library's function, so this method of calling functions will be used in testing.

Key in our research is that NumPy can perform operations on arrays of different but compatible dimensions. For example, we can add two arrays with dimensions 5x5 and 5:

```
>>> arr1:np.ndarray = np.arange(1, 26, 1).reshape((5, 5)
>>> arr1
array([[ 1, 2, 3, 4, 5],
[ 6, 7, 8, 9, 10],
[11, 12, 13, 14, 15],
[16, 17, 18, 19, 20],
[21, 22, 23, 24, 25]])
>>> arr2:np.ndarray = np.arange(1, 5)
>>> arr2
array([1, 2, 3, 4, 5])
>>> np.add(arr1, arr2) array([[ 2, 4, 6, 8, 10],
[ 7, 9, 11, 13, 15],
[12, 14, 16, 18, 20],
[17, 19, 21, 23, 25],
[22, 24, 26, 28, 30]])
```

Also, by specifying the axis of operation, you can calculate the sum of each column of the array.

```
>>> sum_arr:np.ndarray = np.add(arr1, arr2)
>>> np.sum(sum_arr, axis=1) array([ 30, 55, 80, 105, 130])
```

We will use this technique when testing tools for gradient descent.

Moreover, finally, Numba is a library for optimized execution of Python functions. It supports almost all built-in language features and NumPy functions and operations. Numba translates Python functions into optimized machine code during execution using the standard LLVM compiler library. The function is compiled at the first call (Just-In-Time compilation), and then the compiled version is called, which runs much faster. The study considers two decorators from Numba – jit and vectorized. They are used very simply:

```
from numba import jit
@jit
def f(x:float)  ->  float: return x**3 - 2 * x**2 + 5
```

The function *f* will be compiled at the first call and work faster after that. We will use the time module from the standard Python library for measurements.

```
from numba import jit
from time import time
import numpy as np
@jit
def f(x:float) -> float:
    return x**3 - 2 * x**2 + 5
arr:np.ndarray  =  np.arange(1,  10000,  0.05)
print(arr.shape)
start = time()
f(arr)
print(time() - start)
start = time()
f(arr)
print(time() - start)
The program output is as follows:
scssCopy code
(199980,)
0.6352410316467285
0.0010001659393310547
```

Also, Numba allows you to create your own u-funcs, working on the principle of NumPy functions. For example, let us implement a function that calculates the square root of each array element.

```
from numba import vectorize
@vectorize
def f(x:float) -> float: return x * x
Test the function on a NumPy array:
>>> arr:np.ndarray = np.arange(1, 6, 1)
>>> arr
array([1, 2, 3, 4, 5])
>>> f(arr)
array([ 1, 4, 9, 16, 25])
```

Vectorized functions are the subject of research in image normalization and the use of activation functions.

Special mention should be made of CUDA and Numba.Cuda. Computations on GPUs are a relatively new concept in Computer Science, but despite this, they have made significant changes in Machine Learning and Deep Learning. When comparing two types of processors used for calculations, the deciding factor is the number of floating-point operations per second (flops). At the moment, the value of this metric for graphic cards is about ten times higher than for CPUs.

The graphic processor has many computing cores (in modern realities, thousands or tens of thousands), which are combined into blocks, which imposes some restrictions. Thus, the high computing power of graphic cards is a consequence of a special architecture (which, of course, imposes its restrictions). Initially, graphic processors were created for rendering textures, creating graphic objects, etc., so the main direction for GPUs became large parallel calculations.

An important point when working with a GPU is that all cores during calculations perform the same set of instructions (SIMD), which relates to the disadvantages of using calculations on graphic processors. Also, graphic processors are used to calculate large volumes of data, but not in complex algorithms with many conditional constructs-loops, etc.

One of the objects of this study in the field of computations is CUDA (Compute Unified Device Architecture) – an SDK from NVidia, that allows the use of video cards of this company for computations.

CUDA SDK has a very convenient interface, many add-ons for various programming languages, and quality documentation. Within the Python programming language, we will focus on implementing the add-on over CUDA in the Numba library. The Numba.cuda submodule provides a simple interface, requiring almost nothing from the user. Using the example of a function that raises each element of the array to a square, we will analyze the mechanism of interaction with CUDA:

```
import numpy as np
from numba import cuda
@cuda.jit
def power2(arr:np.ndarray) -> None:
i:int = cuda.grid(1)
if i < arr.shape[0]:
arr[i] = arr[i] * arr[i]
```

Here, the expression cuda.grid(1) may cause difficulty. This function returns the index for the array element being processed now. This opens up the possibility for parallel calculations, simultaneously changing the passed array in real-time. To run the function, you also need to specify the number of blocks of processor cores used and the number of threads in one block of cores. The product of these two values is equal to the total number of threads used in the calculations. Also, to prevent Out-Of-Bounds Error (in Numba – Cuda.APIError), we check whether the index is valid for a given array.

Let us set the number of threads to 32, and the number of blocks will be calculated according to the recommended formula:

```
threadsperblock = 32
blockspergrid = (len(array) + (threadsperblock - 1))
// threadsperblock
```

Now we call the function from the previously created array of numbers as follows (square brackets when calling are part of the Numba.cuda interface). Putting the array into the graphic processor's memory is also important. Previous experiments show that calculations using Numba.cuda are significantly slowed down without this action.

```
user_arr:np.ndarray = np.random.random(size=(100, ))
user_arr:np.ndarray = cuda.to_device(user_arr)
power2[blockspergrid, threadsperblock](user_arr)
```

At the first call, the function is compiled similarly to numba.jit and changes the source array, raising all its elements to a square. CUDA also supports processing multidimensional arrays (cuda.grid() can return up to 3 values for the index) and some Python and NumPy features. For example, when normalizing images, we will use the range() function (the definition "function" is indicated only because of its prevalence; in fact, range in Python is a class), since images, in our case, are represented as an array of numbers with four dimensions.

Implementation using various tools

This section presents implementations of the tasks described above using the tools we are considering. Almost all functions work in-place; they modify the passed array rather than returning a new one. This is done only to simplify code fragments; this approach is not recommended (at least from the point of view of constructing processing pipelines).

Regression. The software implementation of gradient descent (this optimization method is described above) is very simple and is just a notation of the formula for updating weights. With NumPy, the situation is even easier, as this library inherently supports operations on arrays of different shapes, and we will use this.

Python. In the implementation of pure CPython, the process of updating weights is most explicit, as it is written in the form of two loops (calculating the model's prediction and subsequent updating of weights). Here, we also use the global variable LEARNING_RATE (also only for convenience; using global variables is not good practice) to control learning. LEARNING_RATE is a double-precision floating-point number. The software implementation is as follows:

```
def python_version(weights:list, samples:list, target:list) -> None:
for i in range(len(samples)):
sample:list = samples[i]
value:float = target[i]
predicted:float = 0
for j in range(len(weights)):
predicted += weights[j] * sample[j]
for j in range(len(weights)):
weights[j] -= sample[j] * LEARNING_RATE * (value - predicted)
```

NumPy. Using vectorized operations, we can move away from loops and write the weight update in matrix form (representing the set of samples for training as a matrix and the weights and values of the target variable as row vectors). It looks like this:

```
def numpy_version (weights:np.ndarray,
samples:np.ndarray, target:np.ndarray) -> None:
dw:np.ndarray = np.reshape((target - np.sum
(weights * samples)), (-1, 1))
weights -= LEARNING_RATE * np.sum(samples * dw)
```

All aspects of this implementation (using NumPy functions, changing the shape of the array, etc.) were considered when analyzing work with NumPy.

Numba. For the Numba library, for application in the gradient descent task, we used two implementations: the first is based on pure Python, and the second is implemented based on NumPy.

The functions repeat the implementations given above, except for the use of the @jit decorator; however, for a complete understanding, we provide their source:

```
@jit('(f8[:], f8[:, :], f8[:])')
def numba_python_version (weights:np.ndarray,
samples:np.ndarray, target:np.ndarray) -> None:
for i in range(samples.shape[0]):
sample:list = samples[i]
predicted:float = 0
for j in range(weights.shape[0]):
predicted += weights[j] * sample[j]
for j in range(weights.shape[0]):
weights[j] -= sample[j] * LEARNING_RATE * (target[i]
- predicted)
@jit('(f8[:], f8[:, :], f8[:])')
def numba_numpy_version(weights:np.ndarray,
samples:np.ndarray, target:np.ndarray) -> None:
dw:np.ndarray = np.reshape((target - np.sum(weights
* samples)), (-1, 1))
weights -= LEARNING_RATE * np.sum(samples * dw)
```

CUDA. The implementation based on GPU computations is based on the idea of implementation on CPython. However, we do not iterate over all sets of features but select one using cuda.grid() and update the weights with each selected set. The implementation using CUDA looks like this:

```
pythonCopy code
@cuda.jit('(f8[:], f8[:, :], f8[:])')
def cuda_version (weights:np.ndarray,
samples:np.ndarray, target:np.ndarray) -> None:
i:int = cuda.grid(1)
if i < samples.shape[0]:
pred:float = 0
for j in range(weights.shape[0]):
pred = pred + weights[j] * samples[i, j]
for j in range(weights.shape[0]):
weights[j] = weights[j] - LEARNING_RATE *
samples[i, j] * (target[i] - pred)
```

Image normalization. From a software implementation perspective, image normalization is much simpler, as it requires no additional computations. Here, we divide each element of the 4-D array by 255. Almost all functions listed below also work in place, modifying the passed array.

Python. The main difficulty in Python implementation is calculating the image's dimensions (since the images in the array may have different sizes). Using built-in functions is very costly, so expecting this method to work efficiently is not advisable.

```
def python_version(images:list) -> None:
for i1 in range(len(images)):
for i2 in range(len(images[i1])):
for i3 in range(len(images[i1][i2])):
for i4 in range(len(images[i1][i2][i3])):
images[i1][i2][i3][i4] /= 255
```

NumPy. Since NumPy provides vectorized operations, we can use the division operator (in this case, the "magic" method idiv()) to achieve the goal.

```
def numpy_version(images:np.ndarray) -> None:
images /= 255
```

Numba. Since we use vectorized operations for the current problem, there will be three variants in the implementation using Numba: pure CPython, NumPy, and the vectorize decorator. The first two repeat the source functions of CPython and NumPy, and the third returns a value equal to the passed argument divided by 255. Implementation using Python:

```
@jit
def numba_python_version (images:np.ndarray) -> None:
d1, d2, d3, d4 = images.shape
for i in range(d1):
for j in range(d2):
for k in range(d3):
for p in range(d4):
images[i][j][k][p] /= 255
Implementation using NumPy functions:
@jit
def numba_numpy_version(images:np.ndarray) -> None:
return np.divide(images, 255)
```

The following function is based on the Numba interface, which was discussed earlier. In the example, a squaring function was implemented; now, a division by 255 functions is implemented. Numba implementation using vectorization:

```
@vectorize
def numba_vectorize_version(x:float) -> float:
return x / 255
```

CUDA. In the CUDA implementation from Numba, the cuda.grid() function can return up to 3 values. In our task, the image array has four dimensions, which means using an additional loop to perform the task. However, CUDA is optimized for such constructions, unlike conditional branching. It is also essential to prevent an out-of-bounds error for each image, as the image size may vary (in NumPy, subarrays can have different sizes when dtype='object'). Implementation using the cuda.jit decorator:

```
@cuda.jit()
def cuda_version(images:np.ndarray) -> None:
i1, i2, i3 = cuda.grid(3)
if (i1 < images.shape[0] and
i2 < images[i1].shape[0] and
i3 < images[i1, i2].shape[0]):
for i4 in range(images.shape[3]):
images[i1][i2][i3][i4] /= 255
```

Using the in-place division operator ensures changes to the array elements (CUDA works in this way, changing the state of the array).

Layer activation using activation function. Next, we present implementations of functions that compute the value of the activation function (in our case – the sigmoid function, used in logistic regression). This is a vectorization task; however, unlike the previous task, we consider a row vector, not a 4-D array, and the calculations are somewhat more complex, as exponentiation from this point of view is much "heavier" than ordinary division.

Python. In the Python implementation, the idea is simple: use a loop to iterate over the elements of the x array, replacing them with the value 1-x. The function works in place:

```
pythonCopy code
def python_version(values:list) -> None:
for i in range(len(values)):
values[i] = 1 / (1 + exp(-values[i]))
```

NumPy. Using the u-func of the NumPy library, we can remove the iteration of elements and write a pure mathematical expression. Here, the function works with a return, as it is impossible to change the elements of the array in place in this way without using multiple constructions. However, using the return construction will not affect the execution time of the function.

```
def numpy_version(values:np.ndarray) -> np.ndarray:
return 1 / (1 + np.exp(-values))
```

Numba. As in the previous task, this subsection presents three implementations using different approaches and decorators. They have the same properties as the corresponding implementations without using just-in-time compilation.

```
@jit('(f8[:])')
def numba_python_version(values:np.ndarray) -> None:
for i in range(len(values)):
values[i] = 1 / (1 + exp(-values[i]))
@jit('f8[:] (f8[:])')
def numba_numpy_version (values:np.ndarray) -> np.ndarray:
return 1 / (1 + np.exp(-values))
```

The function generated by the vectorize decorator also works:

```
@vectorize
def numba_vectorize_version(x:float) -> float:
return 1 / (1 + np.exp(-x))
```

CUDA. In the implementation of calculations on the GPU, we select one index for the array element and change this element by index using the sigmoid function.

```
pythonCopy code
@cuda.jit()
def cuda_version(values:np.ndarray) -> None:
i:int = cuda.grid(1)
if i < values.shape[0]:
values[i] = 1 / (1 + exp(-values[i]))
```

Results

It should be noted in advance that for the purity of the experiment, all functions were implemented using Numba and Numba.Cuda is precompiled using a call. This is because the Numba interface allows you to save compiled functions to a separate source file, which can be imported and used in the future without compilation at the first call. Therefore, only the pure execution time should be evaluated.

The nominal time must not interest us; the main object of analysis is the law of growth of execution time and the ratio between the indicators of different functions. In the last section, we will discuss all aspects related to the results of execution, as there are many of them.

Calculations were performed on a machine with the following configuration:

- AMD Ryzen 7 (8/16) 3.0-3.7 GHz (AM4 Socket)
- MSI GTX 1060 TI (3 GB)
- SSD 256 GB

Regression. A graph of the dependence of the execution time of various functions on the number of feature sets in the considered matrix is presented in Fig. 1.

This graph shows that the function's implementation in pure Python is significantly faster than implementations in NumPy and its variations with just-in-time compilation. As expected, CUDA implementations are unparalleled in calculations (acceleration relative to Python is slightly less than a million times).

Table 1 is presenting execution times for various tools, depending on the number of rows in the dataset.

Table 1 – **Regression task execution time**

Tool	**Number of samples in dataset**				
	5 000	**50 000**	**100 000**	**150 000**	**200 000**
Python	0.13095	1.3231	2.5853	3.7579	5.1502
Numba (Python)	0.00012	0.00100	0.00216	0.00304	0.00404
Numba (NumPy)	0.00036	0.00668	0.01344	0.02245	0.02713
NumPy	0.00024	0.00784	0.01668	0.02737	0.03421
CUDA	7.932e-05	8.149e-05	8.355e-05	8.481e-05	8.566e-05

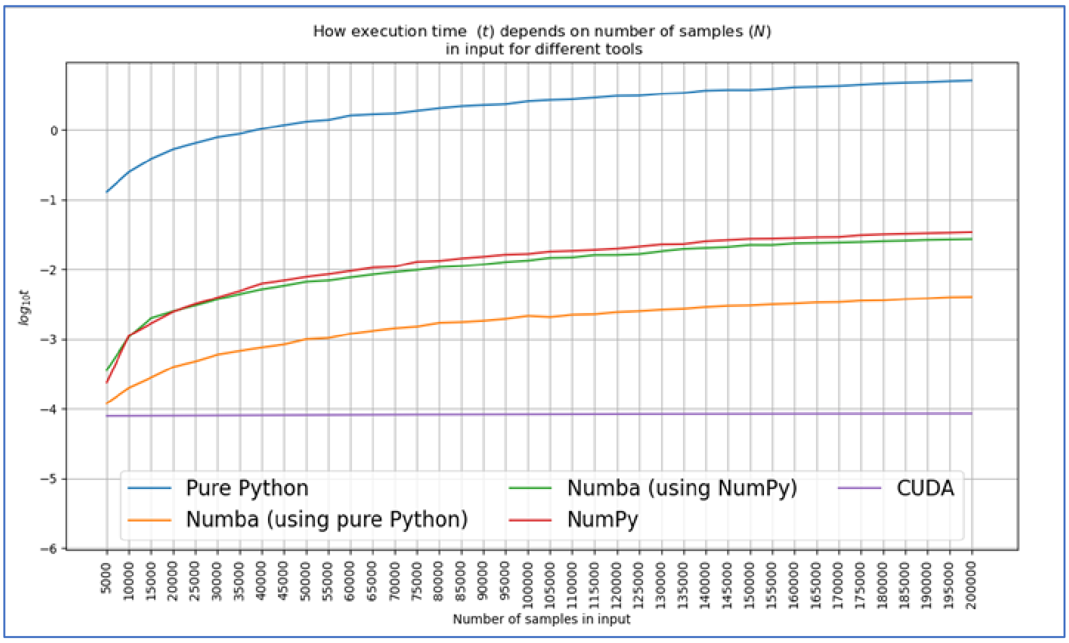

Fig. 1. A graph of the dependence of the execution time of various functions on the number of feature sets in the considered matrix

Image normalization. When normalizing images, as can be seen from the graph, Numba in any of its variations is better than NumPy, giving an acceleration of about six times. The execution speed of the function using CUDA practically does not change from the number of images in the array. A significant result is that we achieved acceleration of the function precisely with the implementation of Numba, based on standard Python constructions, which speaks of extremely high optimization of cyclic operations (Fig. 2).

Table 2 shows the dependence of normalization execution time on the number of images in the set.

Activation function. Let us analyze the execution results for calculating the activation function. This is an essential aspect, as such a task often arises in Machine Learning and Deep Learning spheres, where the speed of calculations plays a crucial role in the development process.

A graph of the dependence of the execution time of various functions on the number of feature sets in the considered matrix is presented in Fig. 3.

Table 3 shows the execution times for some amounts of values in one-dimensional arrays for which the activation function is calculated:

Table 2 – **Image normalization execution time**

Tool	**Number of samples in dataset**				
	500	**2500**	**5000**	**7500**	**9500**
Python	0.28126	1.37804	2.78959	3.93409	4.99358
Numba (Python)	0.00024	0.00131	0.00256	0.00376	0.0048
Numba (NumPy)	0.00055	0.00292	0.00576	0.00908	0.01196
Numba (vectorized)	0.00031	0.00232	0.00492	0.00772	0.01028
NumPy	0.00035	0.00228	0.00496	0.00792	0.01044
CUDA	3.999e-05	4.539e-05	5.236e-05	6.002e-05	6.34e-05

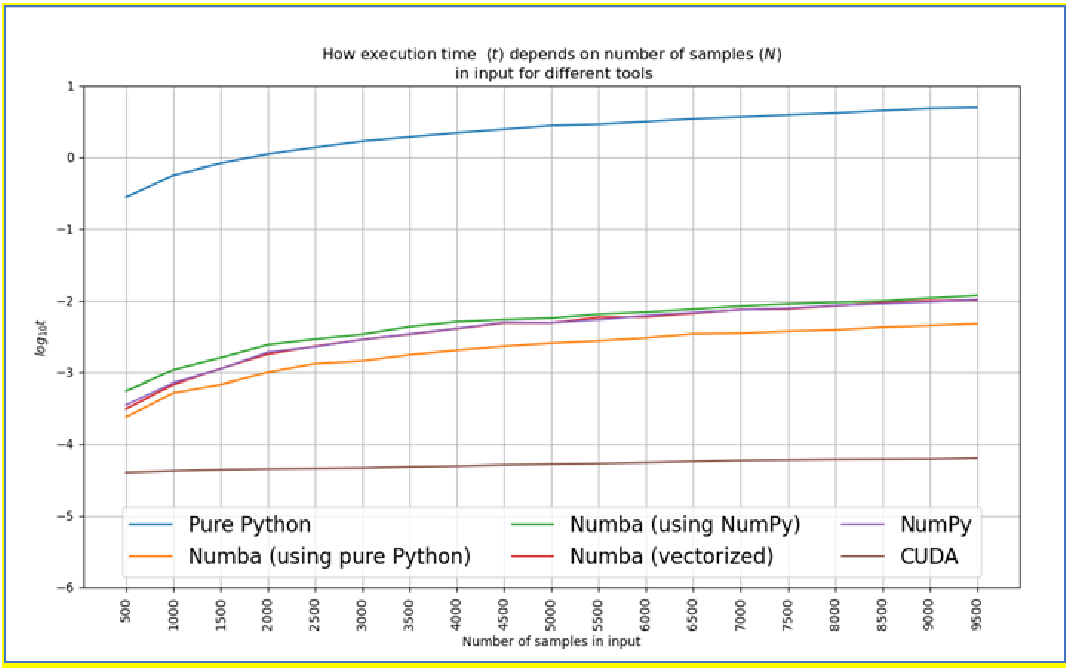

Fig. 2. A graph of the dependence of the execution time of various functions on the number of feature sets **in** the considered matrix

Table 3 – **Activation function execution time**

Tool	Number of samples in dataset				
	50000	**250000**	**500000**	**750000**	**950000**
Python	0.03597	0.16328	0.32821	0.48553	0.63951
Numba (Python)	0.00036	0.00194	0.00378	0.00574	0.00738
Numba (NumPy)	0.00016	0.00108	0.0022	0.00356	0.00472
Numba (vectorized)	0.00014	0.00112	0.00256	0.00364	0.00492
NumPy	0.0007	0.00438	0.00854	0.01342	0.01748
CUDA	4.038e-05	4.08e-05	4.218e-05	4.656e-05	4.997e-05

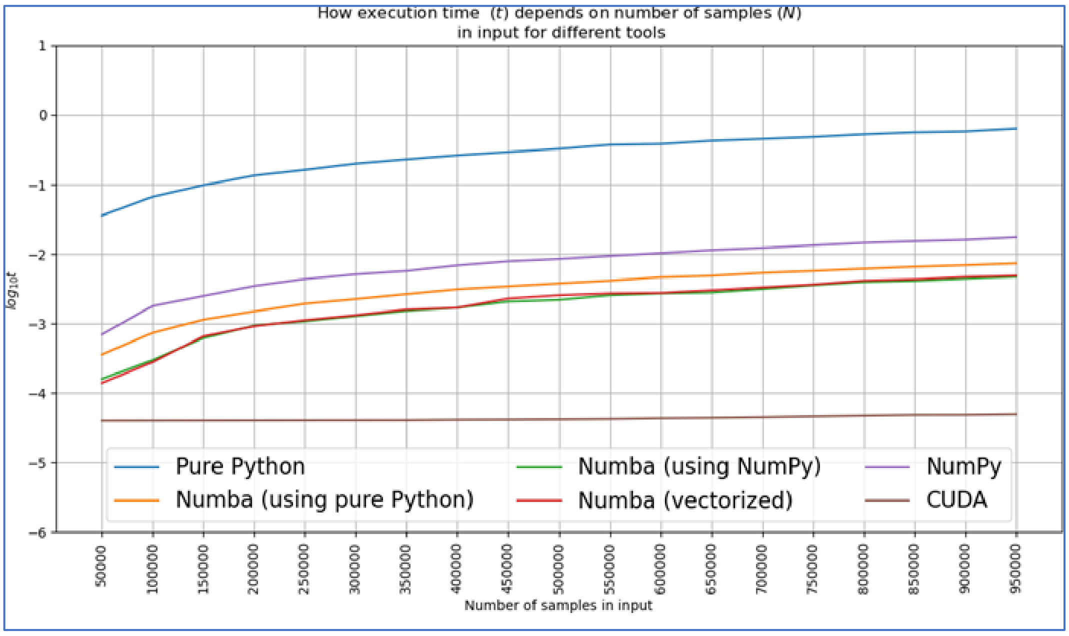

Fig. 3. A graph of the dependence of the execution time of various functions on the number of feature sets in the considered matrix

Discussion

PyTorch [24] and Keras [25] can use both CPU and GPU for computations. Therefore, comparing CUDA and other implementations here is particularly meaningful (unlike the previous task, as the primary data preprocessing pipeline for models is performed on the CPU).

As we can see from the graph, implementations using Numba again increase execution speed compared to regular Python or NumPy.

The study presents the software implementation of solutions for three applied tasks from the field of Data Science, using Python, NumPy, Numba, and Numba.Cuda. Based on the execution of the implemented programs, correspondences of the program's runtime, using one or another tool, were compiled depending on the amount of data to be processed. An analysis was carried out on the rationality of using various tools for a specific task using the obtained data presented in the work in the form of graphs and tables.

Even though computations on video cards are the fastest option for calculations, they should be used in specific situations.

With a small dataset, due to the memory exchange between the main processor and the graphic processor, the computation speed may be the same as when performed on the main processor. Computations on video cards should be used in high-load systems or with a large dataset (sufficiently large means a high difference between the execution speed on the processor and the graphic processor).

It is also worth considering the economic aspect of the issue. Graphic processors on the modern market are expensive, and even old models (for example, video cards up to the RTX generation at NVidia) may seem overpriced for some users.

However, based on the experiment results, using Numba seems optimal for calculations on the CPU, as it practically does not require anything from the user. In the regression task, we achieved a speed increase using the for...in construction compared to NumPy. Unfortunately, not all NumPy functions work accelerated when combined with Numba, so using the latter library is useful with a very good understanding of the developer's specific task.

It is worth considering separately the possibility of implementing multithreaded data processing, which, in this case, can significantly accelerate the data processing process itself.

Parallel calculations in Python without external libraries can be done using several modules: threading – which provides the ability to manage threads; queue - which is responsible for organizing queues; and multiprocessing – which manages processes. In this case, we are mainly interested in the first module. To start working with it, you need to import the class:

```
from threading import Thread
```

After the import, the Thread() function will be available – with its help, we will create threads.

For example, like this:

```
variable = Thread(target=function_name, args=(arg1, arg2,))
```

For convenience and to avoid confusion during debugging, assigning a name to the threads is advisable. To perform calculations in this case, creating a separate class that inherits from Thread from the threading module is advisable. Moreover, prescribe the program of actions in the run() method. This need is dictated by the fact that the thread's behavior will be quite complex. The implementation can be as follows:

```
import threading
class MyThread(threading.Thread):
def __init__(self, num):
super().__init__(self, name="threddy" + num)
self.num = num
def run(self):
print ("Thread ", self.num),
thread1 = MyThread("1")
thread2 = MyThread("2")
thread1.start()
thread2.start()
thread1.join()
thread2.join()
```

A similar implementation is found in many languages.

For calculations using multithreading, and not on the CPU, but on the GPU, the NUMBA library is used in conjunction with NumPy. Considering the computing power of modern GPUs and their ability to work in multithreading mode, we will get a multiple reduction in the time required for processing.

Suppose it is necessary to implement the addition of one-dimensional arrays, element by element. Let us implement it like this:

```
def arr_sum (x , y):
result_arr = nupmy.empty_like ( x)
for i in range (len (x)) :
result_arr [i ] = x[i ] + y[i ]
return result_arr
```

To speed up the execution of the code, it makes sense to import the jit class from the numba module and add the @jit annotation at the beginning of the code:

```
from numba import jit @jit def arr_sum(x,y)
```

Thus, the processing will be significantly accelerated.

However, multithreading in the program can have negative consequences, with incorrect implementation or suboptimal use.

For example, when using multithreading, it is not advisable to use more threads than processor cores available for use; it should be noted that some processors have different virtualization and parallelization technologies.

Interrelated calculations will add dependency between data streams, leading to performance problems and, in case of an error, to the program's termination.

Conclusions

The study aimed to analyze the performance of various computational tools, including Python, NumPy, Numba, and Numba.Cuda, in the context of Data Science applications.

The experiment used the tools above to implement solutions for three applied tasks: regression analysis, image normalization, and activation function computation. The performance of each tool was evaluated based on the execution time for different data sizes.

The main results indicate that while computations on video cards (GPUs) are the fastest, they are most effective in high-load systems or with a sufficiently large dataset. Specifically, the use of Numba.Cuda showed significant speed improvements compared to traditional Python or NumPy implementations. However, the experiment also highlighted that GPUs may not always be the most efficient approach, especially for small datasets, due to the memory exchange between the main and graphic processors. Additionally, the study revealed that using Numba seems optimal for CPU calculations as it does not require significant modifications to the existing Python code and provides a notable speed increase compared to NumPy.

The contribution of this study to the field lies in its comprehensive analysis of the performance of different computational tools in specific Data Science applications. It provides valuable insights into the optimal use of these tools based on the size of the dataset and the nature of the computations involved. Moreover, the study sheds light on the potential benefits and limitations of using multithreading and GPU computations in Python. This can guide researchers and practitioners in selecting the most appropriate tools for their tasks.

Future work should focus on expanding the scope of the experiment to include additional computational tasks and tools.

Moreover, it would be beneficial to explore the impact of different hardware configurations on the performance of the tools analyzed in this study. Additionally, further research is needed to investigate multithreading's potential negative consequences and develop strategies for optimizing its use in different applications.

Ultimately, a more comprehensive understanding of the performance characteristics of various computational tools will contribute to developing more efficient and effective solutions in the field of Data Science.

Acknowledgements

The study was funded by the National Research Foundation of Ukraine in the framework of the research project 2020.02/0404 on the topic “Development of intelligent technologies for assessing the epidemic situation to support decision-making within the population biosafety management”.

References

1. Motamarri, S., Akter, S., Yanamandram, V. and Wamba, S. F. (2017), “Why is Empowerment Important in Big Data Analytics?”, *Procedia Computer Science*, vol. 121, pp. 1062–1071, doi: https://doi.org/10.1016/j.procs.2017.11.136
2. Zhang, J., Cui, Y., Fan, X. and Ren, J. (2023), “Asynchronous Multithreading Reinforcement Control Decision Method for Unmanned Surface Vessel,” *IEEE Internet of Things Journal*, Vol. 10, Is. 24, doi: https://doi.org/10.1109/jiot.2023.3305387
3. (2018), “Numba: A High-Performance Python Compiler,” *Pydata.org*, available at: https://numba.pydata.org/
4. (2009), “NumPy”, *Numpy.org*, available at: https://numpy.org/
5. (2021), “python/cpython”, *GitHub*, available at: https://github.com/python/cpython
6. (2023), “Numba for CUDA GPUs — Numba 0.50.1 documentation”, *numba.pydata.org*, available at: https://numba.pydata.org/numba-doc/latest/cuda/index.html
7. (2018), “Python Data Analysis Library — pandas: Python Data Analysis Library,” *Pydata.org,* available at: https://pandas.pydata.org/
8. Haleem, A., Javaid, Mohd., Khan, I. H. and Vaishya, R.(2020), “Significant Applications of Big Data in COVID-19 Pandemic”, *Indian Journal of Orthopaedics*, Vol. 54, No. 4, pp. 1–3, doi: https://doi.org/10.1007/s43465-020-00129-z
9. Garattini, C., Raffle, J., Aisyah, D. N., Sartain, F. and Kozlakidis, Z. (2017), “Big Data Analytics, Infectious Diseases and Associated Ethical Impacts”, *Philosophy & Technology,* Vol. 32, No. 1, pp. 69–85, doi: https://doi.org/10.1007/s13347-017-0278-y
10. Yakovlev S., Bazilevych, K., Chumachenko, D., Chumachenko, T., Hulianytskyi, L., Meniailov, Ie. and Tkachenko, A. (2020), “The Concept of Developing a Decision Support System for the Epidemic Morbidity Control”, *CEUR Workshop Proceedings*, Vol. 2753, pp. 265–274, available at: https://ceur-ws.org/Vol-2753/paper19.pdf
11. Hasan, Md. M., Popp, J. and Oláh, J. (2020), “Current landscape and influence of big data on finance,” *Journal of Big Data*, Vol. 7, No. 1, pp. 1–17, doi: https://doi.org/10.1186/s40537-020-00291-z
12. Izonin, I., Tkachenko, R., Verhun, V. and Zub, K. (2021), “An approach towards missing data management using improved GRNN-SGTM ensemble method”, *Engineering Science and Technology, an International Journal*, Vol. 24, No. 3, pp. 749–759, doi: https://doi.org/10.1016/j.jestch.2020.10.005
13. Davidich, N., Chumachenko, I., Davidich, Y., Taisiia, H., Artsybasheva, N. and Tatiana, M. (2020), “Advanced Traveller Information Systems to Optimizing Freight Driver Route Selection”, *2020 13th International Conference on Developments in eSystems Engineering (DeSE)*, doi: https://doi.org/10.1109/dese51703.2020.9450763
14. Chew, A. M. K. and Gunasekeran, D. V. (2021), “Social Media Big Data: The Good, The Bad, and the Ugly (Un)truths”, *Frontiers in Big Data*, Vol. 4, doi: https://doi.org/10.3389/fdata.2021.623794
15. Ahmed, R., Shaheen, S. and Philbin, S. P. (2022), “The role of big data analytics and decision-making in achieving project success”, *Journal of Engineering and Technology Management,* Vol. 65, 101697, doi: https://doi.org/10.1016/j.jengtecman.2022.101697
16. Harris, C. R., Millman, K. Ja., Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, Ju., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H., Brett, M., Haldane, A., Río, Ja. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C. and Oliphant T. E. (2020), “Array programming with NumPy,” *Nature*, Vol. 585, No. 7825, pp. 357–362, doi: https://doi.org/10.1038/s41586-020-2649-2
17. Lam, S. K., Pitrou, A. and Seibert, S. (2015), “Numba: A LLVM-based Python JIT Compiler”, *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC - LLVM ’15*, doi: https://doi.org/10.1145/2833157.2833162
18. Oden, L. and Saidi, T. (2021), “Implementation and Evaluation of CUDA-Unified Memory in Numba,” *Springer eBooks*, pp. 197–208, Jan. 2021, doi: https://doi.org/10.1007/978-3-030-71593-9_16
19. Nguyen G., Dlugolinsky S., Bobák M., Tran V., García, Á. L., Heredia, I., Malík, P. and Hluch, L. (2019), “Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: a survey”, *Artificial Intelligence Review*, Vol. 52, No. 1, pp. 77–124, Jan. 2019, doi: https://doi.org/10.1007/s10462-018-09679-z

20. (2013), "CUDA Toolkit," *NVIDIA Developer*, Jul. 02, 2013, available at: https://developer.nvidia.com/cuda-toolkit
21. Pala, A. and Sadecki, J. (2018), "Application of the Nvidia CUDA Technology to Solve the System of Ordinary Differential Equations", *Advances in intelligent systems and computing (AISC)*, Vol. 720, Jan. 2018, pp. 207–217, doi: https://doi.org/10.1007/978-3-319-75025-5_19.
22. Dash, S., Shakyawar, S. K., Sharma, M. and Kaushik, S. (2019), "Big data in healthcare: management, analysis and future prospects", *Journal of Big Data*, Vol. 6, No. 1, Jun. 2019, pp. 1–25, doi: https://doi.org/10.1186/s40537-019-0217-0
23. Packhäuser, K., Gündel, S., Münster, N., Syben, C., Christlein, V. and Maier, A. (2022), "Deep learning-based patient re-identification is able to exploit the biometric nature of medical chest X-ray data," *Scientific Reports*, Vol. 12, No. 1, Sep. 2022, doi: https://doi.org/10.1038/s41598-022-19045-3
24. (2023), "PyTorch," *Pytorch.org*, available at: https://pytorch.org/
25. (2019), "Home - Keras Documentation," *Keras.io*, 2019, available at: https://keras.io/

Надійшла (received) 25.11.2023
Прийнята до друку (accepted for publication) 13.02.2024

ВІДОМОСТІ ПРО АВТОРІВ / ABOUT THE AUTHORS

Кривцов Сергій Олегович – аспірант кафедри математичного моделювання та штучного інтелекту, Національний аерокосмічний університет ім. М. Є. Жуковського «Харківський авіаційний інститут», Харків, Україна;
Serhii Krivtsov – PhD Student of Department of Mathematical Modeling and Artificial Intelligence, National Aerospace University "Kharkiv Aviation Institute", Kharkiv, Ukraine;
e-mail: krivtsovpro@gmail.com; ORCID ID: https://orcid.org/0000-0001-5214-0927;
Scopus ID: https://www.scopus.com/authid/detail.uri?authorId=57214220648.

Парфенюк Юрій Леонідович – доктор філософії, старший викладач кафедри теоретичної та прикладної інформатики, Харківський національний університет ім. В.Н. Каразіна, Харків, Україна;
Yurii Parfeniuk – PhD, Senior Lecturer of Department of Theoretical and Applied Informatics, V.N. Karazin Kharkiv National University, Kharkiv, Ukraine
e-mail: parfuriy.l@gmail.com; ORCID ID: https://orcid.org/0000-0001-5357-1868;
Scopus ID: https://www.scopus.com/authid/detail.uri?authorId=57204619131.

Базілевич Ксенія Олексіївна – кандидат технічних наук, доцент, доцент кафедри математичного моделювання та штучного інтелекту, Національний аерокосмічний університет ім. М.Є. Жуковського «Харківський авіаційний інститут», Харків, Україна;
Kseniia Bazilevych – Candidate of Technical Sciences, Associate Professor, Associate Professor of Department of Mathematical Modeling and Artificial Intelligence, National Aerospace University "Kharkiv Aviation Institute", Kharkiv, Ukraine;
e-mail: ksenia.bazilevich@gmail.com; ORCID ID: https://orcid.org/0000-0001-5332-9545;
Scopus ID: https://www.scopus.com/authid/detail.uri?authorId=57202239038.

Меняйлов Євген Сергійович – кандидат технічних наук, доцент, в.о. завідувача кафедри теоретичної та прикладної інформатики, Харківський національний університет ім. В.Н. Каразіна, Харків, Україна;
Ievgen Meniailov - Candidate of Technical Sciences, Associate Professor, Acting Head of Department of Theoretical and Applied Informatics, V.N. Karazin Kharkiv National University, Kharkiv, Ukraine;
e-mail: evgenii.menyailov@gmail.com; ORCID ID: https://orcid.org/0000-0002-9440-8378;
Scopus ID: https://www.scopus.com/authid/detail.uri?authorId=57202229519.

Чумаченко Дмитро Ігорович – кандидат технічних наук, доцент, доцент кафедри математичного моделювання та штучного інтелекту, Національний аерокосмічний університет ім. М.Є. Жуковського «Харківський авіаційний інститут», Харків, Україна;
Dmytro Chumachenko – Candidate of Technical Sciences, Associate Professor, Associate Professor of Department of Mathematical Modeling and Artificial Intelligence, National Aerospace University "Kharkiv Aviation Institute", Kharkiv, Ukraine;
e-mail: dichumachenko@gmail.com; ORCID ID: https://orcid.org/0000-0003-2623-3294;
Scopus ID: https://www.scopus.com/authid/detail.uri?authorId=58194260300.

Оцінка продуктивності бібліотек Python для обробки даних з використанням багатопотоковості

С. О. Кривцов, Ю. Л. Парфенюк, К. О. Базілевич, Є. С. Меняйлов, Д. І. Чумаченко

Анотація. **Актуальність.** Швидке зростання даних у різних доменах потребує розробки ефективних інструментів та бібліотек для обробки та аналізу даних. Python, популярна мова програмування для аналізу даних, пропонує кілька бібліотек, таких як NumPy та Numba, для чисельних обчислень. Однак, існує нестача всебічних досліджень, які порівнюють продуктивність цих бібліотек у різних задачах та з різними розмірами даних. **Мета дослідження.** Це дослідження має на меті заповнити цей пробіл, порівнюючи продуктивність Python, NumPy, Numba та Numba.Cuda в різних задачах та з різними розмірами даних. Крім того, воно оцінює вплив багатопотоковості та використання GPU на швидкість обчислень. **Результати дослідження.** Результати вказують, що Numba та Numba.Cuda значно оптимізують продуктивність додатків Python, особливо для функцій, що включають цикли та операції з масивами. Більше того, використання GPU та багатопотоковості в Python додатково підвищує швидкість обчислень, хоча і з певними обмеженнями та міркуваннями. **Висновок.** Це дослідження вносить вклад у галузь, надаючи цінні висновки щодо продуктивності різних бібліотек Python та ефективності використання GPU та багатопотоковості в Python, тим самим допомагаючи дослідникам та практикам у виборі найбільш підходящих інструментів для їхніх обчислювальних потреб.

Ключові слова: машинне навчання; Python; GPU; багатопотоковість; оптимізація чисельних обчислень.