Вестник Сыктывкарского университета.
Серия 1: Математика. Механика. Информатика. 2023.
Выпуск 1 (46)
Bulletin of Syktyvkar University.
Series 1: Mathematics. Mechanics. Informatics. 2023; 1 (46)

ИНФОРМАТИКА

INFORMATICS

Научная статья

УДК 378.14, 004.432
https://doi.org/10.34130/1992-2752_2023_1_14

ПРИМЕНЕНИЕ БИБЛИОТЕКИ NUMPY ДЛЯ ВЕКТОРИЗАЦИИ КОДА PYTHON

Надежда Николаевна Бабикова
Сыктывкарский государственный университет
им. Питирима Сорокина, valmasha@mail.ru

Аннотация. Векторизация кода — процесс перехода от операций над отдельными элементами массивов к операциям, происходящим над целыми массивами или их частями. В статье рассматриваются инструменты библиотеки NumPy, позволяющие векторизовать код на языке Python: векторные функции, укладывание, маскирование, прихотливая индексация. Эффективность применения этих инструментов продемонстрирована на примере двух задач машинного обучения.

Ключевые слова: NumPy, Python, векторизация, многомерные массивы, циклы

Для цитирования: Бабикова Н. Н. Применение библиотеки NumPy для векторизации кода Python // *Вестник Сыктывкарского университета. Сер. 1: Математика. Механика. Информатика.* 2023. Вып. 1 (46). С. 14–29. https://doi.org/10.34130/1992-2752_2023_1_14

© Бабикова Н. Н., 2023.

Article

Using NumPy to vectorization of Python code

Nadezhda N. Babikova
Pitirim Sorokin Syktyvkar State University, valmasha@mail.ru

Abstrakt. Code vectorization is the process of moving from operations on individual elements of arrays to operations that occur on entire arrays or their parts. The NumPy library tools that allow to vectorize Python code are discussed in the article: vector functions, broadcasting, masking, fancy indexing. The effectiveness of these tools is demonstrated on the example of two machine learning problems.

Keywords: NumPy, Python, vectorization, multidimensional arrays, loops

For citation: Babikova N. N. Using NumPy to vectorization of Python code . *Vestnik Syktyvkarskogo universiteta. Seriya 1: Matematika. Mekhanika. Informatika* [Bulletin of Syktyvkar University, Series 1: Mathematics. Mechanics. Informatics], 2023, no 1 (46), pp. 14–29. https://doi.org/10.34130/1992-2752_2023_1_14

Введение

Библиотека Numerical Python, сокращенно NumPy, – это основа научной экосистемы языка Python. NumPy лежит в основе почти каждой библиотеки Python, предназначенной для научных или числовых вычислений, включая SciPy, Matplotlib, Pandas, Scikit-learn и Scikit-image [1, с. 357]. Библиотека NumPy предоставляет удобный и гибкий интерфейс для оптимизированных вычислений над массивами данных. Векторизованные операции в NumPy делегируют выполнение цикла высоко оптимизированным функциям C и Fortran, что делает код Python более чистым и быстрым.

Векторизация кода – процесс перехода от скалярных операций над отдельными элементами массивов к векторным операциям, происходящим над целыми массивами или их частями. С точки зрения сложности реализации, а также изучения или обучения способы векторизации кода можно разделить на непосредственные и опосредованные. *Непосредственная векторизация* заключается в прямой замене поэлементной обработки массивов в циклах на векторные операции и использовании *универсальных функций* (universal function) NumPy. *Опо-*

средованная векторизация подразумевает дополнительно к этому использование специальных возможностей библиотеки NumPy, таких как *укладывание* (broadcasting), *маскирование* (masking) и *прихотливое индексирование* (fancy indexing). Цель статьи – показать на примере задач машинного обучения эффективность применения инструментов векторизации NumPy с точки зрения повышения производительности кода Python.

Непосредственная векторизация

Рассмотрим непосредственную векторизацию кода. Основой библиотеки NumPy является объект *ndarray* для представления однородного n-мерного массива. У любого массива есть атрибут *shape* (форма) – кортеж, описывающий размер по каждому измерению, атрибут *dtype* – объект, определяющий тип данных в массиве, атрибут *strides* – кортеж, который задает шаги перемещения к следующему элементу данных по каждому измерению. Синтаксис индексации и доступа к подмассивам с помощью *срезов* (slicing) библиотеки NumPy аналогичен синтаксису для стандартных списков языка Python.

Срезы массивов возвращают *представления* (views), а не *копии* (copies) данных массива. Представления NumPy – это метод доступа к данным массива, при котором изменяются только метаданные dtype и/или strides, но не меняется буфер данных. Один и тот же массив в NumPy может иметь разные представления и математически и программно обрабатываться по-разному. При изменении значений в представлении изменяются данные в исходном массиве. Использование представлений полезно при векторизации кода – создание копии занимает больше времени и памяти, чем создание представления [2].

Векторизованные вычисления в библиотеке NumPy реализованы посредством универсальных функций. Универсальной (*u*-функцией) называется функция, которая выполняет поэлементные операции над данными, хранящимися в объектах ndarray. Можно считать, что это векторные обертки вокруг простых функций. Универсальные функции библиотеки NumPy очень просты в использовании, поскольку применяют нативные арифметические операторы языка Python. Все арифметические операции – удобные адаптеры для встроенных функций библиотеки NumPy. Например, оператор + является адаптером для функции add().

В табл. 1 приводится сравнение средней скорости выполнения поэлементной обработки массивов, списков Python и векторной опера-

ции NumPy. Проверка проводилась в среде IDLE (Python 3.10 64 bit), NumPy 1.22.3, процессор Intel Core i5-5200U (ноутбук), 4GB DDR3 L Memory.

Таблица 1

Среднее время выполнения циклов и векторных операций

Двумерные массивы		

`# Векторная операция`
`l = np.ones((n,n), dtype=np.int32)`
`l += 10`
	n= 100
	n=1000
	n=5000
		t= 0.0000
		t= 0.0009
		t= 0.0211

`# Циклы по списку`
`l = [[1.0 for i in range(n)]`
`for j in range(n)]`
`for i in range(n):`
`for j in range(n):`
`l[i][j] += 10.0`
	n= 100
	n=1000
	n=5000
		t= 0.0019
		t= 0.2545
		t= 6.0717

`# Циклы по массиву`
`l = np.ones((n,n), dtype=np.int32)`
`for i in range(n):`
`for j in range(n):`
`l[i, j] += 10`
	n= 100
	n=1000
	n=5000
		t= 0.0081
		t= 0.7472
		t=20.6330

Одномерные массивы		
`l = np.array(range(n), dtype=np.int32)` `l += 10`	n= 100000 n=10000000	t= 0.0001 t= 0.0098

`l = list(range(n))`
`for i in range(n):`
`l[i] += 10`
	n= 100000
	n=10000000
		t= 0.0217
		t= 2.4403

`l = np.array(range(n), dtype=np.int32)`
`for i in range(n):`
`l[i] += 10`
	n= 100000
	n=10000000
		t= 0.0675
		t= 6.5120

Таблица 2

Среднее время выполнения циклов с проверкой условия и векторной функции

# Векторная операция 	
a = np.random.randint(1, 10000,
size = (n,n), dtype=np.int32)
z = np.where(a > 5, a, a+11)
	n= 100
	n=1000
	n=5000
		t= 0.00015
		t= 0.0059
		t= 0.1468

# Циклы по списку 	
aa = list(np.random.randint(1, 10000,
size = (n,n), dtype=np.int32)
z = list(np.zeros((n,n),dtype=np.int32))
for i in range(n):
for j in range(n):
if aa[i][j]>5:
z[i][j]=aa[i][j]
else:
z[i][j]=aa[i][j]+11
	n= 100
	n=1000
	n=5000
		t= 0.0092
		t= 0.9637
		t=24.7588

# Циклы по массиву 	
a = np.random.randint(1, 10000,
size = (n,n), dtype=np.int32)
z = np.zeros((n,n), dtype=np.int32)
for i in range(n):
for j in range(n):
if a[i, j]>5:
z[i, j]=a[i, j]
else:
z[i ,j]=a[i, j]+11
	n= 100
	n=1000
	n=5000
		t= 0.0107
		t= 1.1528
		t=29.6370

В табл. 2 приводится сравнение средней скорости выполнения поэлементной обработки массивов, списков Python при наличии условного оператора и функции NumPy. Функция numpy.where – это векторный вариант тернарного выражения «x if condition else y».

Преимущество векторных операций очевидно, но удивительно то, что в больших циклах использование массивов ndarray менее эффективно, чем использование списков чистого Python.

Опосредованная векторизация

Рассмотрим опосредованную векторизацию кода на примере применения метода кластеризации k-means для уменьшения количества цветов в изображении. В листинге 1 приведен код считывания трехканального изображения и подготовки данных для кластеризации. Далее в цикле по количеству итераций выполняется вычисление квадрата расстояния от каждого пикселя изображения до центра каждого кластера, для каждого пикселя определяется номер центра с минимальным расстоянием (листинг 2). Затем рассчитываются средние арифметические RGB для каждого кластера, если эти значения отличаются от центров кластеров менее чем на заданную точность, то цикл завершается. Иначе вычисленные значения становятся новыми центрами. После завершения цикла перекрашиваем все пиксели каждого кластера в цвет центра.

Листинг 1

```
from time import time
import matplotlib.pyplot as plt
path_to_png_file = r"C:\0110.png"
import matplotlib.image as mpimg
import numpy as np
# функция для вычисления квадрата расстояния
def dist(s1,s2):
    d = 0
    for si,sj in zip(s1,s2):
        d += (si-sj)**2
    return d
k = 5 # количество кластеров
max_iter = 10
# оригинальный рисунок
img = mpimg.imread(path_to_png_file)
```

```
# определяем высоту и ширину изображения
img_x = img.shape[0]
img_y = img.shape[1]
# массив размера img_x*img_y - номера кластеров для пикселей
z = np.zeros((img_x, img_y), dtype=np.int32)
# выбор центров кластеров, массив k*3
cl_center = np.zeros((k,3))
for i in range(3):
    cl_center [:, i] = np.linspace(0.25, 0.95, k)
```

Листинг 2

```
iter = 0
while iter != max_iter:
# вычисление квадратов расстояний до центров кластеров и
#                                 номера ближайшего центра
    for x in range(img_x):
        for y in range(img_y):
            z[x, y] = np.argmin([dist(img[x, y], cl_center[j])
                                 for j in range(k)])
# вычисление средних значений RGB в каждом кластере
    sum = np.zeros((k, 3))    # средние значения компонент RGB
                              #              в каждом кластере
    sum_n = np.zeros(k, dtype=np.int32)  # количество пикселей
                                         #   в каждом кластере
    for x in range(img_x):
        for y in range(img_y):
            sum_n[z[x, y]] +=1
            sum[z[x, y], :] += img[x, y, :]
    for i in range(k):
        if sum_n[i] != 0:
            sum[i, :] = sum[i, :]/sum_n[i]
# проверка достижения точности
    if np.allclose(sum, cl_center, 0.02):
        print('ups')
        break
# назначаем новые центры
    cl_center = np.copy(sum)
    iter += 1
```

```
# перекрашиваем все пиксели каждого кластера в цвет центра
for x in range(img_x):
    for y in range(img_y):
        img[x, y, :] = cl_center[z[x, y], :]
```

В программе много циклов, которые невозможно реализовать непосредственно как векторные операции. Среднее время выполнения одной итерации цикла для изображения размером (582, 800) и 5 кластеров составляет около 62 секунд (рис. 1, 2).

Очевидно, самый тяжеловесный цикл в программе – это вычисление расстояний от каждого пикселя до каждого центра кластера и определения номера ближайшего кластера. Для того чтобы избавиться от него, потребуется укладывание (транслирование) – способ выполнения арифметических операций над массивами разной формы. Это очень мощный механизм, но даже опытные пользователи иногда испытывают затруднения с его пониманием. Автор книги «Python и анализ данных» Уэс Маккинли пишет: «Даже я, опытный пользователь Numpy, иногда вынужден рисовать картинки, чтобы понять, как будет применяться правило укладывания» [3]. В последних версиях библиотеки Numpy появилась функция np.broadcast_shapes(), которая позволяет проверить правильность предполагаемой процедуры укладывания. Эта функция принимает на вход кортежи с размерностями исходных массивов и возвращает размерность результирующего массива после укладывания.

Укладывание в библиотеке NumPy следует строгому набору правил, определяющему взаимодействие двух массивов [4]:

1. Если размерность двух массивов отличается, форма массива с меньшей размерностью дополняется единицами с ведущей (левой) стороны.

2. Если форма двух массивов не совпадает в каком-то измерении, массив с формой, равной 1 в данном измерении, растягивается вплоть до соответствия форме другого массива.

3. Если в каком-либо измерении размеры массивов различаются и ни один не равен 1, генерируется ошибка.

Рассматриваемый цикл можно заменить одной строкой кода (листинг 3). Массив img имеет размерность (img_x, img_y, 3), массив

Рис. 1. Исходное изображение

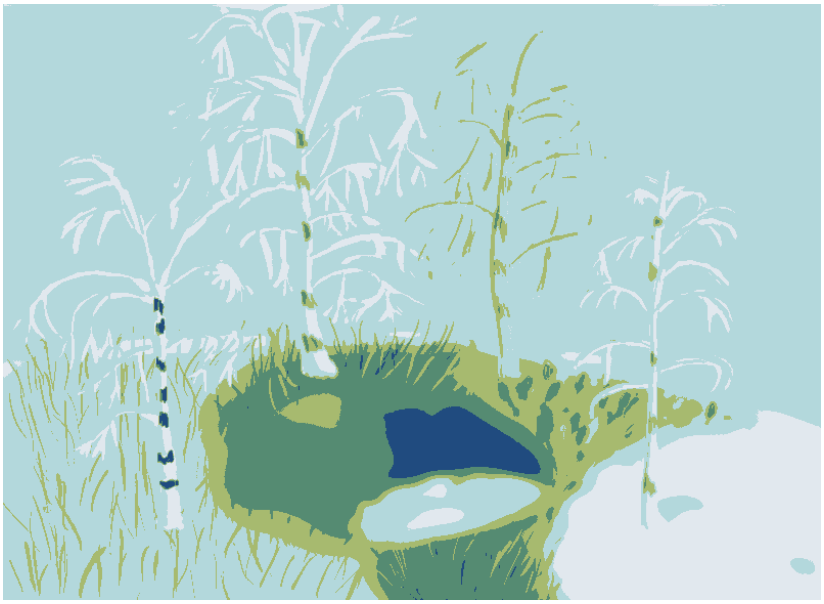

Рис. 2. Изображение после выполнения одной итерации

cl_center – (k, 3), запись cl_center[:, None, None, :] подразумевает размерность (k, 1, 1, 3). Размерность разности массивов (img – cl_center) определяется по правилам укладывания – (k, img_x, img_y, 3) (табл. 3). Ключевое слово None (синоним np.newaxis) вводит дополнительное измерение. Ключевое слово axis задает измерение массива, которое будет «схлопнуто».

Листинг 3

```
iter = 0
while iter != max_iter:
# вычисление расстояний до центров кластеров и номера
#                                    ближайшего центра
   z = np.argmin(np.sum((img-cl_center[:, None, None, :])**2,
                        axis=3), axis=0)
# вычисление средних значений RGB в каждом кластере
   sum = np.zeros((k,3)) # средние значения компонент RGB
                         # в каждом кластере
   for i in range(k):
     for j in range(3):
       sum[i, j] = np.sum(img[:, :, j][z==i])/(np.sum(z==i)+1)
# проверка достижения точности
   if np.allclose(sum, cl_center, 0.02):
     print('ups')
     break
# назначаем новые центры
   cl_center = np.copy(sum)
   iter += 1
# перекрашиваем все пиксели каждого кластера в цвет центра
   img = cl_center[z]
```

Таблица 3

Применение правила укладывания

img	1	Img_x	Img_y	3
cl_center	k	1	1	3
img – cl_center	k	Img_x	Img_y	3

После замены тройного цикла для расстояний на векторную операцию среднее время выполнения одной итерации цикла составляет около 4.5 секунд, т. е. сокращается приблизительно в 15 раз.

Вычисление средних значений компонент цвета для каждого кластера в первоначальной версии программы содержит два цикла. Использование маскирования (наложения маски) позволит заменить их одним (листинг 3). Маскирование заключается в индексировании по булеву массиву.

В результате применения к массиву z операции сравнения получаем булев массив (z==i) той же размерности, что и массив z. Вычисляем np.sum(z==i) – сумма истинных значений в булевом массиве, т. е. количество пикселей в кластере с номером i. Вырезаем j компонент RGB из матрицы изображения и накладываем маску, тем самым получаем все элементы, которые относятся к кластеру i: img[:, :, j][z==i].

Конечно, можно было бы еще больше упростить код, используя функцию np.mean() для вычисления средних значений. Но возможна ситуация, когда в некоторый кластер не попадет ни один пиксель, а функция np.mean() в этом случае выдает сообщение об ошибке.

После замены циклов для вычисления средних значений время выполнения одной итерации цикла составляет около 1.5 секунд, т. е. сокращается еще приблизительно в 3 раза.

Последний цикл в программе перекрашивает каждую точку изображения в цвет соответствующего центра кластера. Для векторизации цикла можно использовать прихотливую (причудливую) индексацию (листинг 3). Суть прихотливой индексации заключается в передаче целочисленного массива индексов с целью одновременного доступа к нескольким элементам массива. В случае прихотливой индексации форма результата отражает форму массива индексов, а не форму индексируемого массива.

После замены цикла перекрашивания пикселей среднее время выполнения одной итерации цикла составляет около 0.25 секунды, т. е. сокращается еще приблизительно в 5 раз.

Векторизация задачи

В некоторых случаях векторизация кода невозможна без предварительной векторизации задачи [5; 6], т. е. переосмысления алгоритма ее решения. Рассмотрим векторизацию задачи на примере фильтрации изображения. К изображению в оттенках серого применяется верти-

кальный фильтр Собеля (листинг 4), размер ядра 3 на 3, страйд равен 1. Исходное изображение имеет размер 400 на 400 пикселей, в результате фильтрации получаем изображение размером 398 на 398. Ядро скользит по матрице изображения, на каждом шаге вычисляется сумма попарных произведений соответствующих элементов ядра и текущего окна изображения, равного по форме ядру. Программа выполняется приблизительно 2.3 секунды. Стандартная функция cv2.filter2D() библиотеки OpenCV выполняется за наносекунды, результаты фильтрации идентичны (рис. 3, 4).

Листинг 4

```
import cv2
import numpy as np
from time import time
img = cv2.imread('C:\\804.png', cv2.IMREAD_GRAYSCALE)
img = img/255
=========================================================
yadro = np.array([[-1,0,1],[-2,0,2],[-1,0,1]])
res = np.zeros((398,398))
# фильтрация 1
for i in range(398):
   for j in range(398):
      res[i, j] = np.sum(yadro*img[i: i+3, j: j+3])
=========================================================
#использование библиотечной функции фильтра
im = cv2.filter2D(img, -1, yadro)
=========================================================
cv2.imshow('NO FILTER ', img)
cv2.waitKey(0)
cv2.imshow('LIBRARY FILTER', im)
cv2.waitKey(0)
cv2.imshow('MY FILTER', res)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рис. 3. Исходное изображение

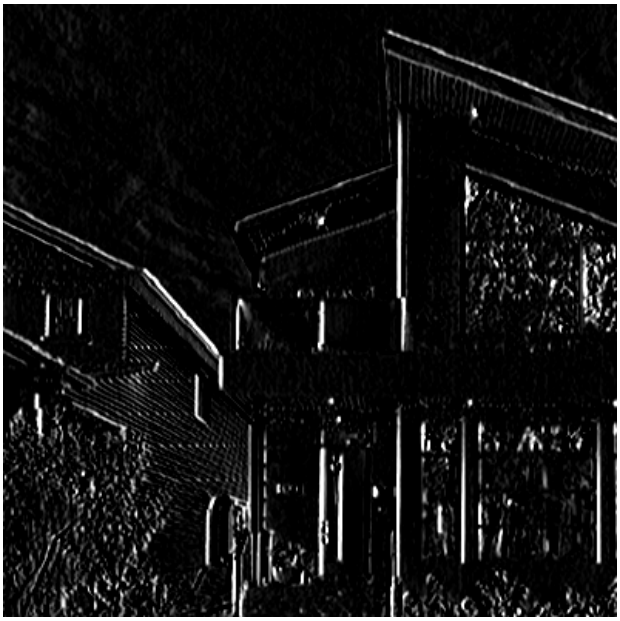

Рис. 4. Изображение после фильтрации

Индексы среза меняются на каждом шаге цикла, поэтому от цикла нельзя избавиться. Но слова «сумма попарных произведений» любого математика наводят на мысль о произведении матриц: если бы элементы ядра были строкой матрицы, а элементы окон – столбцами матрицы (или наоборот), то можно было бы за одну операцию выполнить все вычисления. Для этого нужно предварительно сформировать массив из всех окон. Имеет место проблема пространственно-временного компромисса: размер массива всех окон будет в 9 раз больше (для ядра 3 на 3) размера исходного изображения. Цикл формирования массива окон занимает меньше времени, чем цикл вычисления результата для каждого окна (листинг 5). В этом случае алгоритм фильтрации выполняется приблизительно за 0.5 секунды.

Идея скользящего (сканирующего) окна используется не только в алгоритме фильтрации, но и во многих других алгоритмах обработки изображений. Начиная с версии 1.20.00, в Numpy появилась функция np.lib.stride_tricks.sliding_window_view() [2], которая строит массив из всех возможных окон изображения (в общем случае произвольного массива) (листинг 5). Применение этой функции сокращает время выполнения алгоритма до приблизительно 0.016 секунды.

Листинг 5

```
yadro1 = np.array([-1,0,1,-2,0,2,-1,0,1])
=======================================================
# фильтрация 2
k=0
res2 = np.zeros((9,158404))
for j in range(398):
   for i in range(398):
      res2[0:9, k] = img[i: i+3, j: j+3].flatten()
      k+=1
res2 = np.dot(yadro1, res2)
res2 = np.transpose(res2.reshape((398,398)))
=======================================================
# фильтрация 3
v = np.lib.stride\_tricks.sliding\_window\_view(img, (3, 3))
v = v.reshape(398,398,9)
res1 = np.dot(v, yadro1.T)
res1 = res1.reshape((398,398))
```

Заключение

Библиотека NumPy хорошо документирована и имеет довольно простой и продуманный синтаксис, что важно в процессе обучения. Изучение библиотеки студентами направления подготовки «Прикладная информатика» происходит в рамках дисциплин «Интеллектуальный анализ данных» и «Математические методы в экономике». Опыт показал, что студенты легко справляются с освоением универсальных функций и маскирования. Затруднения возникают при необходимости использования механизмов укладывания и прихотливой индексации, что не удивительно – использование этих механизмов требует элементов творчества и, конечно, опыта.

Список источников

1. **Harris C. R., Millman K. J., van der Walt S. J. et al.** Array programming with NumPy // *Nature.* 2020. No. 585. Pp. 357–362. https://doi.org/10.1038/s41586-020-2649-2.

2. NumPy documentation. Version: 1.25.dev0. URL: https://numpy.org/devdocs/user/basics.copies.html (дата обращения: 07.02.2023).

3. **Уэс Маккинли.** Python и анализ данных / пер. с англ. А. А. Слинкин М.: ДМК Пресс, 2015. 482 с.

4. **Плас Дж. Вандер.** Python для сложных задач: наука о данных и машинное обучение. СПб.: Питер, 2018. 576 с.

5. **Nicolas P. Rougier.** From-python-to-numpy. URL: https://www.labri.fr/perso/nrougier/from-python-to-numpy/#code-vectorization (дата обращения: 07.02.2023).

6. **Shenoy A.** How Are Convolutions Actually Performed Under the Hood. URL: https://towardsdatascience.com/how-are-convolutions-actually-performed-under-the-hood-226523ce7fbf (дата обращения: 07.02.2023).

References

1. **Harris C. R., Millman K. J., van der Walt S. J. et al.** Array programming with NumPy. *Nature*, 2020, no. 585. pp. 357–362. https://doi.org/10.1038/s41586-020-2649-2.

2. *NumPy documentation. Version: 1.25.dev0.* Available at: https://numpy.org/devdocs/user/basics.copies.html (accessed: 07.02.2023).

3. **Ues Makkinli.** *Python i analiz dannyh / per. s angl. A. A. Slinkin* [Python for data analysis] M.: DMK Press, 2015. 482 p.

4. **Plas Dzh. Vander.** *Python dlya slozhnyh zadach: nauka o dannyh i mashinnoe obuchenie* [Python for complex tasks. Data Science and Machine Learning]. SPb.: Piter, 2018. 576 p. (In Russ.)

5. **Nicolas P. Rougier.** *From-python-to-numpy.* Available at: https://www.labri.fr/perso/nrougier/from-python-to-numpy/#code-vectorization (accessed: 07.02.2023).

6. **Shenoy A.** *How Are Convolutions Actually Performed Under the Hood.* Available at: https://towardsdatascience.com/how-are-convolutions-actually-performed-under-the-hood-226523ce7fbf (accessed: 07.02.2023).

Сведения об авторе / Information about author
Бабикова Надежда Николаевна / Nadezhda N. Babikova
к.пед.н., доцент, доцент кафедры прикладной информатики / PhD (Pedagogy), Associate Professor, Associate Professor of Applied Informatics Department
Сыктывкарский государственный университет им. Питирима Сорокина / Pitirim Sorokin Syktyvkar State University
167001, Россия, г. Сыктывкар, Октябрьский пр., 55 / 167001, Russia, Syktyvkar, Oktyabrsky Ave., 55

Статья поступила в редакцию / The article was submitted 02.03.2023
Одобрено после рецензирования / Approved after reviewing 06.03.2023
Принято к публикации / Accepted for publication 10.03.2023