Лабораторная работа №1

Знакомство с инструментальными средствами программирования для выполнения лабораторных работ по дисциплине «Системы искусственного интеллекта»

ОБЗОР БИБЛИОТЕК Python ДЛЯ РАБОТЫ С НЕЙРОННЫМИ СЕТЯМИ. ОСНОВЫ РАБОТЫ С МАССИВАМИ В БИБЛИОТЕКЕ NumPy

Язык программирования

С ростом объемов и сложности данных, повышается необходимость их обработки и анализа при помощи нейронных сетей. В сложных случаях нейронные сети могут дать гораздо более точные оценки и прогнозы, которые заметно повышают эффективность, увеличивают производительность и снижают расходы. Для создания, обучения и последующего использования нейросетевых моделей язык **Python** предлагает очень широкие возможности.

Python – это простой в освоении мощный язык программирования. Он имеет эффективные высокоуровневые структуры данных и простой, но эффективный подход к объектно-ориентированному программированию. Элегантный синтаксис и динамическая типизация Python, а также его интерпретируемый характер делают его идеальным языком для быстрой разработки приложений во многих областях на большинстве платформ.

Популярные библиотеки Python дают возможность готовить данные, отображать результаты, а главное – работать с моделями нейронных сетей на различных уровнях абстракции. Эти библиотеки чрезвычайно полезны, поскольку они экономят время разработчика и дополнительно предоставляют хорошо отлаженные функции, на которые можно смело опираться. Среди огромной коллекции библиотек Python для машинного обучения эти библиотеки следует рассмотреть в первую очередь. С их помощью можно использовать высокоуровневые аналитические функции даже при минимальных знаниях базовых алгоритмов, с которыми предстоит работать.

Среда программирования

Google Colaboratory (сокращенно **Colab**) - бесплатная среда, чтобы писать код в Jupyter notebook. Она функционирует по принципу облака, поэтому над одним проектом могут работать одновременно несколько человек. Программа предоставляет доступ к графическим процессорам GPU и TPU. Благодаря их мощности можно исследовать искусственный интеллект и развивать приложения на основе нейросетей. Сервис бесплатный. С

помощью Colab пишут код на Python прямо в браузере, и здесь же он выполняется. Всё, что требуется, - доступ к гугл-аккаунту. Colab позволяет использовать в одном файле исполняемый код, html-разметку, картинки. Всё будет храниться на гугл-диске. Этими файлами можно делиться: разрешать просматривать, редактировать и оставлять комментарии для совместной работы. Ссылка на хорошее руководство здесь .

Используемые библиотеки

NumPy

NumPy – это библиотека линейной алгебры, разработанная на Python. Почему большое количество разработчиков и экспертов предпочитают ее другим библиотекам Python для машинного обучения? Практически все пакеты Python, использующиеся при работе с нейронными сетями и в целом в машинном обучении, так или иначе опираются на NumPy.

В библиотеку входят функции для работы со сложными математическими операциями линейной алгебры, генерации случайных чисел, методы для работы с матрицами и n-мерными массивами.

Можно ли создавать нейронные сети средствами numpy? Да, можно. Но в этом случае Вы сами как-бы будете создавать отдельные строительные блоки, а потом строить из них дом.

Для того, чтобы использовать только лишь функции numpy, нужны базовые знания принципов работы с этой библиотекой, линейной алгебры, работы с матрицами, дифференциации и метода градиентного спуска.

В основном мы будем использовать эту библиотеку для подготовки данных для нейросети – обучающих (тренировочных) и тестовых.

Scikit-learn

Scikit-learn – отрытая библиотека машинного обучения Python, с широким спектром алгоритмов кластеризации, регрессии и классификации.

Несмотря на то, что библиотека прежде всего поддерживает классические и современные статистические и вероятностные алгоритмы, разработчик нейронных сетей может найти там полезные функции для предобработки данных, использующихся для обучения нейронных сетей:

- снижение размерности
- анализ и выбор признаков
- обнаружение и удаление выбросов
- классификация и кластеризация без учителя

Scikit-learn также отлично взаимодействует с NumPy и SciPy.

В scikit-learn массив NumPy - это основная структура данных. scikit-learn принимает данные в виде массивов NumPy. Любые данные, которые будут использоваться, должны быть преобразованы в массив NumPy. Базовые возможности NumPy - это класс ndarray, многомерный (n-мерный)

массив. Все элементы массива должны быть одного и того же типа. Объекты класса ndarray называют «массивами NumPy» или просто «массивами». Двумерный массив NumPy выглядит следующим образом (рисунок 1):

```
import numpy as np
x=np.array([[1,2,3 ],[4, 5,6]])
print("array x:\n",format(x))
```

```
array x:
 [[1 2 3]
 [4 5 6]]
```

Рисунок 1 – Двумерный массив NumPy

SciPy - это набор функций (библиотека) для научных вычислений в Python. Помимо всего прочего она предлагает продвинутые процедуры линейной алгебры, математическую оптимизацию функций, обработку сигналов, специальные математические функции и статистические функции. scikit-learn использует набор функций SciPy для реализации своих алгоритмов. Наиболее важной частью SciPy является пакет scipy.sparse: с помощью него получают разреженные матрицы (sparse matrices), которые представляют собой еще один формат данных, который используется в scikit-learn. Разреженные матрицы используются всякий раз, когда нужно компактно сохранить 2D массив, который содержит в основном нули. Большие разреженные данные обычно не уменьшаются в памяти, поэтому необходимо создавать разреженные матрицы. В примере приведенно создание матрицы в формате CSR (рисунок 2).

```
from scipy import sparse
# Создать 2D-массив NymPy с единицами на главной диагонали
# и нулями в остальных элементах
eye=np.eye(4)
print("array eye:\n",eye)
# Преобразуем массив в разреженную матрицу,
# которая хранит значения ненулевых элементов
# и их индексы (формат CSR)
sparse_matr=sparse.csr_matrix(eye)
print("sparse matrix for eye:\n",sparse_matr)
```

```
array eye:
 [[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
sparse matrix for eye:
   (0, 0)       1.0
  (1, 1)        1.0
  (2, 2)        1.0
  (3, 3)        1.0
```

Рисунок 2 – Создание CSR- матрицы

Еще один способ, которой позволяет создать такую же разреженную матрицу, что была приведена выше, но этот раз с использованием формата COO (рисунок 3).

```
data=np.ones (4) #генерируем одномерный массив единиц
print("array data:\n",data)
row_indices=np.arange(4) #генерируем одномерный массив индексов строк
col_indices=np.arange(4) #генерируем одномерный массив индексов столбцов
eye_coo=sparse.coo_matrix((data,(row_indices,col_indices )))
print("sparse coo-matrix :\n",eye_coo)
```

```
array data:
 [1. 1. 1. 1.]
sparse coo-matrix :
   (0, 0)       1.0
  (1, 1)        1.0
  (2, 2)        1.0
  (3, 3)        1.0
```

Рисунок 3 – Создание COO-матрицы

CSR (Compressed Sparse Row). В этом формате значения матрицы хранятся в одном массиве, строки индексируются с помощью второго массива, а индексы элементов в строке хранятся в третьем массиве.

CSC (Compressed Sparse Column). CSC - это альтернативный формат хранения разреженных матриц, в котором столбцы индексируются с помощью второго массива, а индексы элементов в столбце хранятся в третьем массиве. В отличие от CSR, CSC хорошо подходит для операций с матрицами, в которых основная цель - работа со столбцами, а не строками.

COO (Coordinate). COO - это простой формат хранения разреженных матриц, где для каждого ненулевого элемента хранятся его значение, строка и столбец. Этот формат не является оптимальным с точки зрения использования памяти, но обеспечивает эффективность при выполнении операций изменения размеров матрицы.

DOK (Dictionary of Keys). DOK - это формат хранения разреженных матриц, представляющий матрицу в виде словаря, где ключами служат пары координат (строка, столбец), а значениями - соответствующие элементы. DOK-формат обеспечивает быстрый доступ к элементам матрицы, но не так эффективен в использовании памяти, особенно для больших матриц.

Более подробную информацию о разреженных матрицах SciPy можно найти в SciPy Lecture Notes (на английском языке).

matplotlib - это основная библиотека для построения научных графиков в Python. Она включает функции для создания высококачественных визуализаций типа линейных диаграмм, гистограмм, диаграмм разброса и т.д.

Визуализация данных и различных аспектов анализа данных может дать исследователю важную информацию.

```
import matplotlib.pyplot as plt
# Генерируем последовательность чисел от -10 до +10 со 100 шагами
x=np.linspace(-10,10,100) # ато массив аргументов
# Создаем массив функции синуса каждого из 100 аргументов
y=np.sin(x)
# строим линейный график
plt.plot(x,y,marker="x")
```

```
[<matplotlib.lines.Line2D at 0x7a56027046d0>]
```

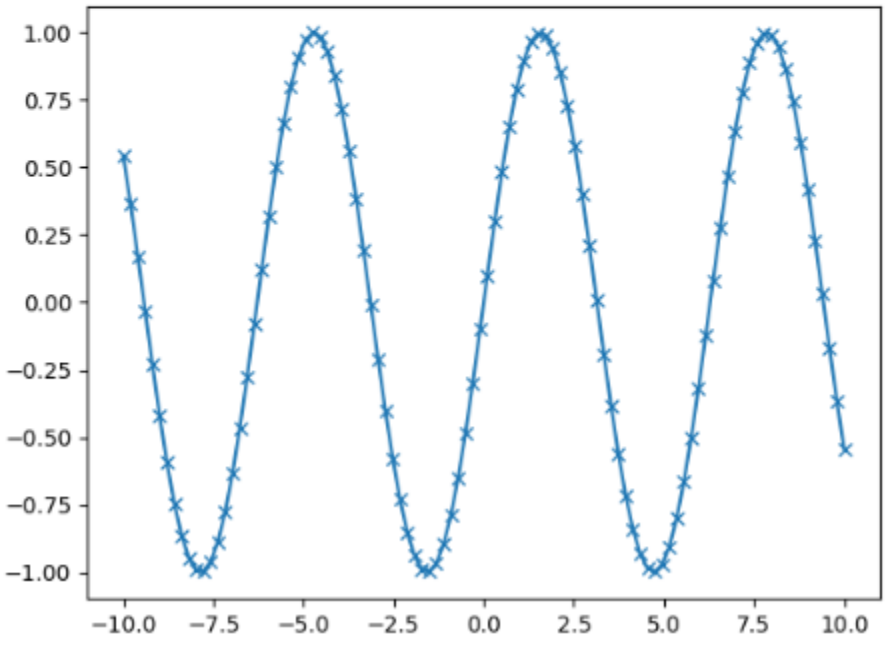


Рисунок 4 – Иллюстрация вывода графика функцией plot

Keras

Keras – одна из основных библиотек Python с открытым исходным кодом, написанная для построения нейронных сетей и проектов машинного обучения. Keras может работать совместно с Deeplearning4j, MXNet, Microsoft Cognitive Toolkit (CNTK), Theano или TensorFlow. В этой библиотеке реализованы практически все автономные модули нейронной сети, включая оптимизаторы, нейронные слои, функции активации слоев, схемы инициализации, функции затрат и модели регуляризации. Это позволяет строить новые модули нейросети, просто добавляя функции или классы. И поскольку модель уже определена в коде, разработчику не приходится создавать для нее отдельные конфигурационные файлы.

Keras особенно удобна для начинающих разработчиков, которые хотят проектировать и разрабатывать собственные нейронные сети. Также Keras

можно использовать при работе со свёрточными нейронными сетями. В ней реализованы алгоритмы нормализации, оптимизации и активации слоев.

Keras не является ML-библиотекой полного цикла (то есть, исчерпывающей все возможные варианты построения нейронных сетей). Вместо этого она функционирует как очень дружелюбный, расширяемый интерфейс, увеличивающий модульность и выразительность (в том числе других библиотек).

Pandas

В проектах по работе с нейронными сетями значительное время уходит на подготовку данных, а также на анализ основных тенденций и нейросетевых моделей. Именно здесь Pandas может предложить свои функции. Python Pandas – это библиотека с открытым исходным кодом, которая предлагает широкий спектр инструментов для обработки и анализа данных. С ее помощью можно читать данные из широкого спектра источников, таких как CSV, базы данных SQL, файлы JSON и Excel.

Эта библиотека позволяет производить сложные операции с данными помощью всего одной команды. Python Pandas поставляется с несколькими встроенными методами для объединения, группировки и фильтрации данных и временных рядов.

Но Pandas не ограничивается только решением задач, связанных с данными; он служит лучшей отправной точкой для создания более сфокусированных и мощных инструментов обработки данных.

от NumPy, который требует, чтобы все записи в массиве были одного и того же типа, в pandas каждый столбец может иметь отдельный тип (например, целые числа, даты, числа с плавающей точкой и строки). Еще одним преимуществом библиотеки pandas является ее способность работать с различными форматами файлов и баз данных, например, с файлами SQL, Excel и CSV.

```
import pandas as pd
# Создание простого набора данных
data = {'Name':["experiment_1","experiment_2","experiment_3","experiment_4"],
        'Duration':[23, 28, 45, 33],
        'Voltage':[1000,1500, 2000,1700]}
data_pandas=pd.DataFrame(data)
# Красивая печать
display(data_pandas)
```

	Name	Duration	Voltage
0	experiment_1	23	1000
1	experiment_2	28	1500
2	experiment_3	45	2000
3	experiment_4	33	1700

Рисунок 5 – Фрейм pandas

Существует несколько способов осуществить запрос к таблице. Например:

```
display(data_pandas[data_pandas.Voltage>1500])
```

	Name	Duration	Voltage
2	experiment_3	45	2000
3	experiment_4	33	1700

Рисунок 6 – Часть фрейма pandas

mglearn - библиотека, которая включает в себя разные полезные функции. Библиотека написана для того, чтобы не перегружать листинги подробной информацией о построении графиков и загрузке данных. Детали mglearn не очень важны для понимания материала (интеллектуальных методов обработки данных). Вызов mglearn в программном коде свидетельствует о быстром способе построить красивую картинку или загрузить некоторые интересные данные. В Colab необходимо установить библиотеку командой:

```
!pip install mglearn
```

Seaborn

Seaborn – мощная библиотека визуализации, основанная на библиотеке Matplotlib. Для нейросетевых проектов важны и описание данных, и их визуализация, поскольку для выбора подходящей нейросетевой модели часто бывает необходим зондирующий анализ набора данных. Seaborn предлагает высокоуровневый интерфейс для создания сложной статистической графики на основе набора данных.

С помощью этой библиотеки машинного обучения легко создавать определенные типы графиков, такие как временные ряды, тепловые карты (heat map) и графики «скрипками» (violin plot). По функционалу Seaborn превосходит Pandas и MathPlotLib благодаря функциям статистической оценки данных в процессе наблюдений и визуализации пригодности статистических моделей для этих данных.

Tensor Flow

TensorFlow – библиотека сквозного машинного обучения Python для выполнения высококачественных численных вычислений. С помощью TensorFlow можно построить глубокие нейронные сети для распознавания образов и рукописного текста и рекуррентные нейронные сети для NLP (обработки естественных языков).

Этот фреймворк имеет отличную архитектурную поддержку, позволяющую с легкостью производить вычисления на самых разных платформах, в том числе на десктопах, серверах и мобильных устройствах.

Основной козырь TensorFlow это абстракции. Они позволяют разработчикам сфокусироваться на общей логике приложения, а не на мелких деталях реализации тех или иных алгоритмов. С помощью этой библиотеки разработчики могут легко использовать нейронные сети для создания уникальных адаптивных приложений, гибко реагирующих на пользовательские данные, например на выражение лица или интонацию голоса.

Theano

Theano – это научная математическая библиотека, которая позволяет определять, оптимизировать и вычислять математические выражения, в том числе и в виде многомерных массивов. В основу большинства нейросетевых приложений положено многократное вычисление заковыристых математических выражений. Theano позволяет проводить подобные вычисления в сотни раз быстрее, вдобавок она отлично оптимизирована под графический процессор (GPU), а также предлагает широкие возможности для тестирования кода.

Когда речь идет о производительности, Theano – отличная библиотека, поскольку она может работать с очень большими нейронными сетями. Её целью является снижение времени разработки и увеличение скорости выполнения приложений, в частности, основанных на алгоритмах глубоких нейронных сетей. Ее единственный недостаток — не слишком простой синтаксис по сравнению с TensorFlow, особенно для новичков.

PyTorch

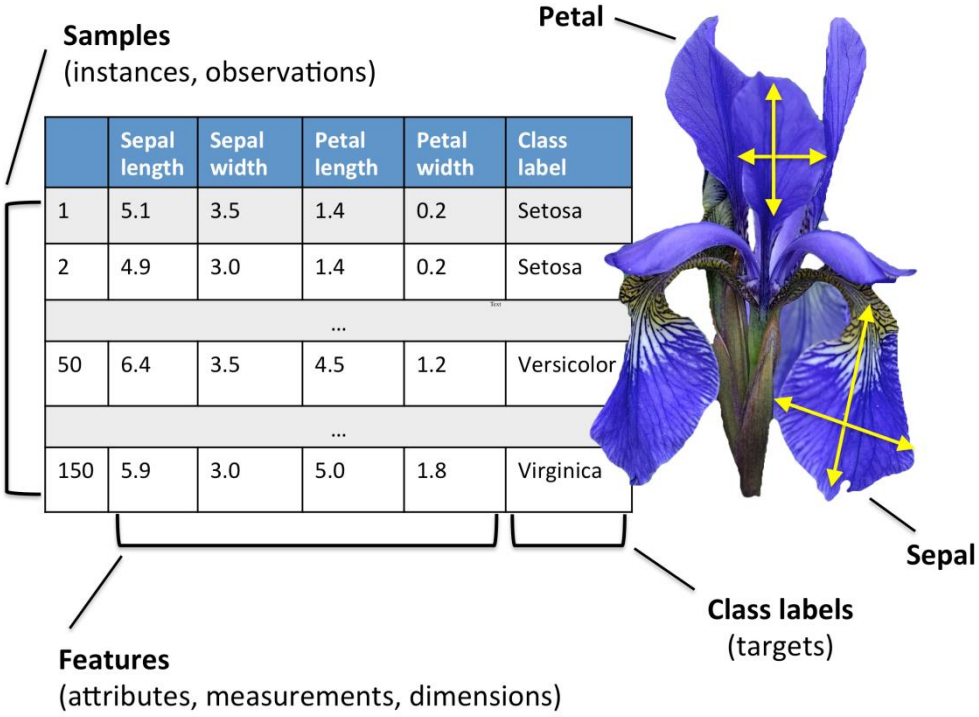
PyTorch – это полностью готовая к работе библиотека машинного обучения Python с отличными примерами, приложениями и вариантами использования, поддерживаемая высокопрофессиональным сообществом. PyTorch отлично адаптирована к графическому процессору GPU, что обеспечивает оптимизацию и масштабирование распределенных задач обучения как в области исследований, так и в области создания ПО.

Глубокие нейронные сети и тензорные вычисления с ускорением на GPU – два основных отличительных преимущества PyTorch. Библиотека также включает в себя компилятор машинного обучения под названием Glow, который серьезно повышает производительность фреймворков глубокого обучения.

Разберем пример с набором данных «Ирисы Фишера».

Данные, которые используются для этого примера, - это набор данных Iris классический набор данных в машинном обучении и статистике. Набор Iris включен в модуль datasets библиотеки scikit-learn. Массив содержит измерения для 150 различных цветов (рисунок 7,8), по 4-м признакам. Вспомним, что в машинном обучении отдельные элементы называются ***примерами*** **(samples)**, а их свойства – характеристиками или атрибутами или измерениями или ***признаками*** **(feature)**. ***Форма*** **(shape)** массива данных определяется количеством примеров, умноженным на количество признаков. Это является общепринятым соглашением в scikit-learn.

Рисунок 7 – Три сорта ирисов



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
			...		
50	6.4	3.5	4.5	1.2	Versicolor
			...		
150	5.9	3.0	5.0	1.8	Virginica

Рисунок 8 – Иллюстраниция набора данных

Загрузка данных

Сначала загрузим и изучим исходные данные задачи:

```
#Пример с Ирисами Фишера
from sklearn.datasets import load_iris
# Загрузка набора данных
iris_dataset = load_iris()

# Объект iris, возвращаемый функцией load_iris, является объектом Bunch,
# который очень похож на словарь.
# Он содержит ключи и значения, посмотри на них
print("Ключи iris_dataset: \n{}".format(iris_dataset.keys()))
print("\n**********************************************************")

# Напечатаем первые 230 символов ключа DESCR - это краткое описание набора
данных
print(iris_dataset['DESCR'][:230] + "\n...")
print("\n**********************************************************")

# ключ target_names - это массив строк, содержащий сорта цветов, которые
мы хотим предсказать:
# ключ feature_names - это список строк с описанием каждого признака
# data - массив NumPy, который содержит количественные измерения длины
чашелистиков,
# ширины чашелистиков, длины лепестков и ширины лепестков

print("Названия признаков: {}".format(iris_dataset['feature_names']))
print("Названия ответов: {}".format(iris_dataset['target_names']))
print("Тип массива data: {}".format(type(iris_dataset['data'])))
print("Форма массива data: {}".format(iris_dataset['data'].shape))
print("Первые пять строк массива
data:\n{}".format(iris_dataset['data'][:5]))
print("\n**********************************************************")

# target - одномерный массив, содержащий сорта цветов, тоже записанные в
виде массива NumPy
# Сорта кодируются как целые числа от 0 до 2
print("Тип массива target: {}".format(type(iris_dataset['target'])))
print("Форма массива target: {}".format(iris_dataset['target'].shape))
print("Ответы:\n{}".format(iris_dataset['target']))
```

```
Ключи iris_dataset:
dict_keys(['data','target','frame','target_names','DESCR','feature_names','filename','data_module'])

**************************************************
.. _iris_dataset:

Iris plants dataset
--------------------

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
...
**************************************************
Названия признаков: ['sepal length (cm)','sepal width (cm)','petal length (cm)','petal width (cm)']
Названия ответов: ['setosa' 'versicolor' 'virginica']
Тип массива data: <class 'numpy.ndarray'>
Форма массива data: (150, 4)
Первые пять строк массива data:
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]]

**************************************************
Тип массива target: <class 'numpy.ndarray'>
Форма массива target: (150,)
Ответы:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

Рисунок 9 – Результат работы кода

Разделение данных на тестовое и обучающее множества (наборы)

На основе этих данных необходимо построить классификационную модель, которая относится к типу моделей машинного обучения, способную предсказывать сорта ириса для новых объектов. Но прежде, чем применить построенную модель к новому набору, необходимо убедиться в том, что модель на самом деле работает и ее прогнозам можно доверять.

Для оценки качества модели нельзя использовать данные, которые были взяты для построения модели. Это обусловлено тем, что модель просто запомнит весь обучающий набор и поэтому она всегда будет предсказывать правильную метку (ответ, номер класса) для любой точки данных в обучающем наборе. Это «запоминание» ничего не говорит об обобщающей способности модели. Другими словами, неизвестно, будет ли эта модель так же хорошо работать на новых данных. Для оценки эффективности построенной модели следует предъявить ей новые размеченные данные (объекты, для которых известен его класс, но которые использовались при создании модели). Обычно это делается путем разбиения собранных размеченных данных (в данном случае 150 цветов) на две неравнозначные

части. Одна часть данных используется для построения модели машинного обучения и называется обучающими данными (training data) или обучающим набором (training set). Другая часть данных будет использована для оценки качества модели. Этц часть называют тестовыми данными (test data), тестовым набором (test set) или контрольным набором (hold-out set).

В библиотеке scikit-learn есть функция train_test_split, которая перемешивает набор данных и разбивает его на две части. Эта функция сначала перемешивает весь набор, а потом отбирает в обучающий набор 75% строк данных с соответствующими метками. Оставшиеся 25% данных с метками объявляются тестовым набором. Вопрос о том, сколько данных отбирать в обучающий и тестовый наборы, является дискуссионным, однако использование тестового набора, содержащего 25% данных, является хорошим правилом.

В scikit-learn данные, как правило, обозначаются заглавной X, тогда как метки обозначаются строчной y. Это навеяно стандартной математической формулой f(x)=y, где x является аргументом функции, а y - выводом. В соответствии с некоторыми математическими соглашениями используется заглавный символ X, потому что данные представляют собой двумерный массив (матрицу) и строчную y, потому что целевая переменная - это одномерный массив (вектор). Чтобы в точности повторно воспроизвести полученный результат, следует воспользоваться генератором псевдослучайных чисел с фиксированным стартовым значением, которое задается с помощью параметра random_state. Это позволит сделать результат воспроизводим - т.е. при каждом запуске будет генерироваться один и тот же результат. При отладке всегда рекомендуется задавать random_state в случае использования рандомизированных процедур. Выводом функции train_test_split являются X_train, X_test, y_train и y_test, которые все являются массивами NumPy. X_train содержит 75% строк набора данных, а X_test содержит оставшиеся 25%:

```
# функция train_test_split разделит данные на обучающие и тестовые
# зададим обучающие данные, обучающие метки, тестовые данные, тестовые
метки

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test= train_test_split(iris_dataset['data'],
iris_dataset['target'], random_state=0)
print("Форма массива X_train: {}".format(X_train.shape))
print("Форма массива y_train: {}".format(y_train.shape))
print("Форма массива X_test: {}".format(X_test.shape))
print("Форма массива y_test: {}".format(y_test.shape))
```

```
Форма массива X_train: (112, 4)
Форма массива y_train: (112,)
Форма массива X_test: (38, 4)
Форма массива y_test: (38,)
```

Рисунок 10 – Результат работы кода

Исследования данных

Перед тем как обучать нейросеть, неплохо было бы исследовать данные, чтобы понять, можно ли легко решить поставленную задачу без обучения сети или содержится ли нужная информация в данных. Кроме того, исследование данных - это хороший способ обнаружить аномалии и особенности. Например, вполне возможно, что некоторые из ирисов измерены в дюймах, а другие - в сантиметрах. В реальном мире нестыковки в данных и неожиданности очень распространены.

Один из лучших способов исследовать данные - визуализировать их. Это можно сделать, используя диаграмму рассеяния (scatter plot). В диаграмме рассеяния один признак откладывается по оси x, а другой признак - по оси y, каждому наблюдению соответствует точка. На экране компьютера можно разместить только два или, возможно, три признака одновременно. Один из способов решения этой проблемы - построить матрицу диаграмм рассеяния (scatterplot matrix) или парные диаграммы рассеяния (pair plots), на которых будут изображены все возможные пары признаков. Если имеет место небольшое число признаков, например, четыре, как в рассматриваемом примере, то использование матрицы диаграмм рассеяния будет вполне разумным. Однако, следует помнить, что матрица диаграмм рассеяния не показывает взаимодействие между всеми признаками сразу, поэтому некоторые интересные аспекты данных не будут выявлены с помощью этих графиков.

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import pandas as pd
import mglearn
from IPython.display import display
import matplotlib.font_manager as fm

# создаем data frame из данных в массиве X_train
# маркируем столбцы, используя строки в iris_dataset.feature_names
iris_dataframe = pd.DataFrame(X_train, columns=iris_dataset.feature_names)
# посмотрим на созданный фрейм
print(iris_dataframe)

font = fm.FontProperties(family='Arial', size=10)
# создаем матрицу рассеяния из data frame, цвет точек задаем с помощью
y_train
grr = pd.plotting.scatter_matrix(iris_dataframe, c=y_train, figsize=(15,
15),
                                 marker='o',hist_kwds={'bins': 20}, s=60,
                                 alpha=.8, cmap=mglearn.cm3)
```

Рисунок 12 представляет собой матрицу диаграмм рассеяния для признаков обучающего набора. Точки данных окрашены в соответствии с

сортами ириса, к которым они относятся. Чтобы построить диаграммы, мы сначала преобразовываем массив **NumPy** в DataFrame (основной тип данных в библиотеке **pandas**). В **pandas** есть функция для создания парных диаграмм рассеяния под названием **scatter_matrix**. По диагонали этой матрицы располагаются гистограммы каждого признака

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.9	3.0	4.2	1.5
1	5.8	2.6	4.0	1.2
2	6.8	3.0	5.5	2.1
3	4.7	3.2	1.3	0.2
4	6.9	3.1	5.1	2.3
..	...	...	...	...
107	4.9	3.1	1.5	0.1
108	6.3	2.9	5.6	1.8
109	5.8	2.7	4.1	1.0
110	7.7	3.8	6.7	2.2
111	4.6	3.2	1.4	0.2

[112 rows x 4 columns]

Рисунок 11 – Фреймовое представление данных об ирисах

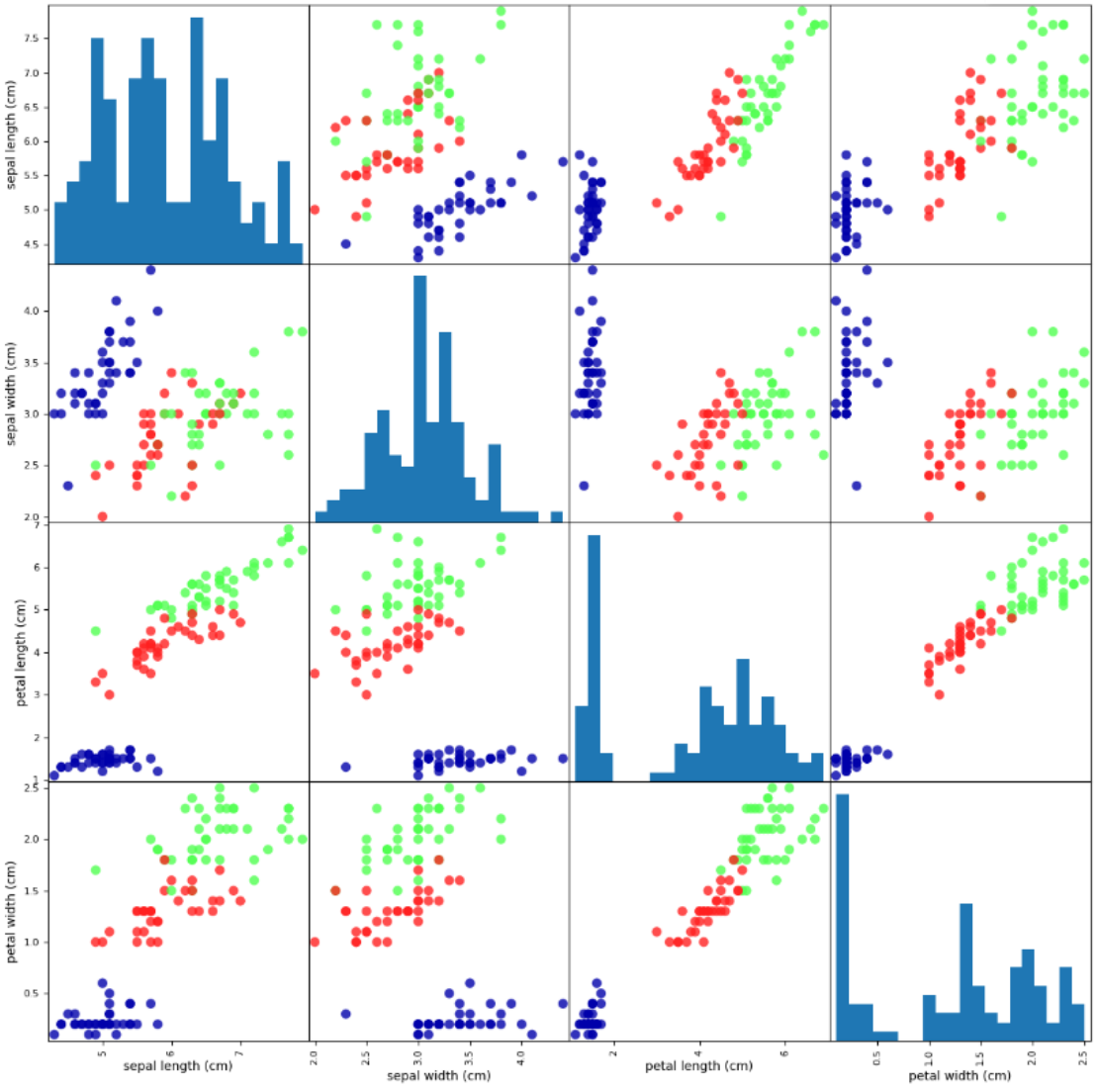


Рисунок 12 – Матрица диаграмм рассеяния

Из представленного рисунка видно, что измерения чашелистиков и лепестков позволяют относительно хорошо разделить три класса. Класс setosa линейно отделим от двух других классов. Классы Versicolor и Virdginica линейно неразделимы. Однако нейронная сеть, при удачном выборе архитектуры и настройках алгоритма обучения, вероятно, сможет научиться разделять их.

Примером ***синтетического набора данных*** для двухклассовой классификации является набор данных **forge**, который содержит два признака. Программный код, приведенный ниже, создает диаграмму рассеяния (рисунок 13), визуализируя все точки данных в этом наборе.

```
import mglearn
import matplotlib.pyplot as plt
# генерируем набор данных
X, y = mglearn.datasets.make_forge()
# строим график для набора данных
%matplotlib inline
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.legend(["Класс 0", "Класс 1"], loc=4)
plt.xlabel("Первый признак")
plt.ylabel("Второй признак")
print("форма массива X: {}".format(X.shape))
```

```
форма массива X: (26, 2)
```

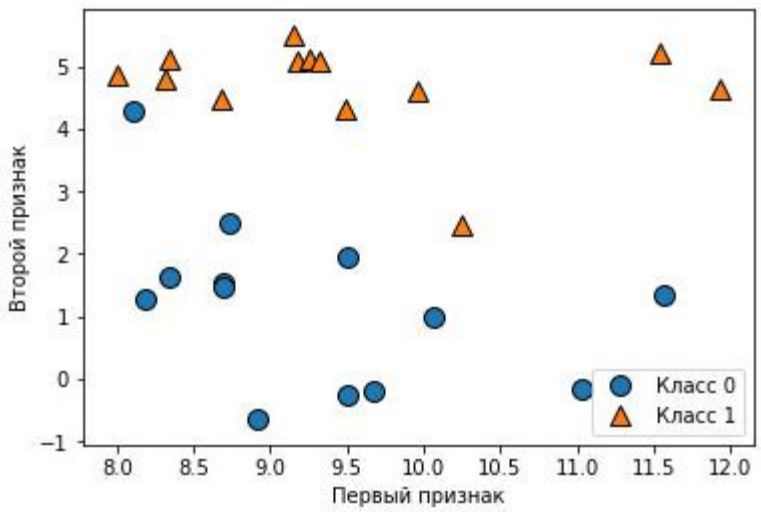


Рисунок 13 – Диаграмма рассеяния для набора данных **forge**

На графике первый признак отложен на оси **x**, а второй - по оси **y**. Как это всегда бывает в диаграммах рассеяния, каждая точка данных представлена в виде одного маркера. Цвет и форма маркера указывает на класс, к которому принадлежит точка.

Как видно из сводки по массиву **X**, этот набор состоит из 26 точек данных и 2 признаков.

Еще один синтетический наборо **wave**. Набор данных имеет единственный входной признак и непрерывную целевую переменную или ***отклик*** **(response)**, который мы хотим смоделировать. На рисунке, построенном здесь (рисунок 14), по оси **x** располагается единственный признак, а по оси **y** - целевая переменная (ответ).

```
X, y = mglearn.datasets.make_wave(n_samples=40)
plt.plot(X, y, 'o')
plt.ylim(-3, 3)
plt.xlabel("Признак")
plt.ylabel("Целевая переменная")
```

```
Text(0, 0.5, 'Целевая переменная')
```

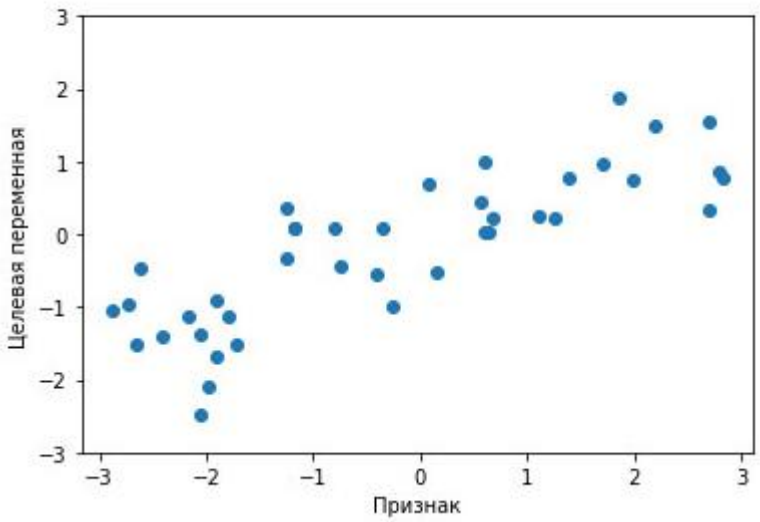


Рисунок 14 - График для набора данных wave, по оси **x** отложен признак, по оси **y** - целевая переменная

Будут использоваться очень простые, низкоразмерные наборы данных, потому что их легко визуализировать - печатная страница имеет два измерения, и данные, которые содержат более двух признаков, графически представить трудно. Вывод, полученный для набора с небольшим числом признаков или ***низкоразмерном*** **(low-dimensional)** наборе, возможно, не подтвердится для набора данных с большим количеством признаков или ***высокоразмерного*** **(high-dimensional)** набора. Об этом следует помнить, однако проверка алгоритма на низкоразмерном наборе данных может оказаться очень полезной для понимания работы метода и алгоритма.

Еще два реальных набора, которые включены в **scikit-learn**. Один из них - набор данных по раку молочной железы Университета Висконсин (**breast_cancer** или **cancer** для краткости), в котором записаны клинические измерения опухолей молочной железы. Каждая опухоль обозначается как **"benign"** (***"доброкачественная"***, для неагрессивных опухолей) или **"malignant"** (***"злокачественная"***, для раковых опухолей), и задача состоит

в том, чтобы на основании измерений ткани дать прогноз, является ли опухоль злокачественной. Набор данных включает 569 точек данных и 30 признаков. Из 569 точек данных 212 помечены как злокачественные, а 357 как доброкачественные. Данные можно загрузить из **scikit-learn** с помощью функции **load_breast_cancer**:

```
import numpy as np
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
print("Ключи cancer(): \n{}". format(cancer.keys()))
print("Форма массива data для набора cancer:
{}".format(cancer.data.shape))

print("Количество примеров для каждого класса:\n{}".format(
    {n: v for n, v in zip(cancer.target_names,
np.bincount(cancer.target))}))
print("Имена признаков:\n{}".format(cancer.feature_names))
```

```
Ключи cancer():
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR',
'feature_names', 'filename', 'data_module'])
Форма массива data для набора cancer: (569, 30)
Количество примеров для каждого класса:
{'malignant': 212, 'benign': 357}
Имена признаков:
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

Рисунок 15 – Ключи и форма набора данных cancer

***Примечание**. Наборы данных, которые включены в **scikit-learn**, обычно хранятся в виде объектов **Bunch**, которые содержат сами фактические данные, а также некоторую информацию о наборе данных. Все, что вам нужно знать об объектах **Bunch** - это то, что они похожи на словари, с тем преимуществом, что можно прочитать значения, используя точку (**bunch.key** вместо **bunch['key']**).*

Более подробную информацию о данных можно получить, прочитав **cancer.DESCR**.

Второй реальный набор данных - набор данных **Boston Housing**. Задача, связанная с этим набором данных, заключается в том, чтобы спрогнозировать медианную стоимость домов в нескольких районах Бостона в 1970-е годы на основе такой информации, как уровень преступности,

близость к **Charles River**, удаленность от радиальных магистралей и т.д. Набор данных содержит 506 точек данных и 13 признаков. Из этических соображений набор был удален из последних версий библиотеки. Однако, вы можете загрузить набор данных напрямую из интернета, используя следующий код:

```
!wget https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data
import pandas as pd
data = pd.read_csv('housing.data', delim_whitespace=True, header=None)
```

Этот код загрузит файл housing.data в текущую директорию. Затем можно использовать pandas для чтения данных из файла. В этом примере данные будут загружены в переменную data в формате DataFrame из библиотеки pandas.

Второй вариант загрузки:

```
import pandas as pd
data_url = "http://lib.stat.cmu.edu/datasets/boston"
raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
target = raw_df.values[1::2, 2]
```

Существуют и альтернативные бостонскому наборы данных по жилищному строительству Калифорнии и набор данных о жилье Эймса. Их можно загрузить данных следующим образом:

```
from sklearn.datasets import fetch_california_housing
housing = fetch_california_housing()

from sklearn.datasets import fetch_openml
housing = fetch_openml(name="house_prices", as_frame=True)
```

Опять же, можно получить более подробную информацию о наборе данных, прочитав атрибут **DESCR**. В данном случае мы более детально проанализируем набор данных, учтя не только 13 измерений в качестве входных признаков, но и приняв во внимание все ***взаимодействия*** **(interactions)** между признаками. Иными словами, мы будем учитывать в качестве признаков не только уровень преступности и удаленность от радиальных магистралей по отдельности, но и взаимодействие уровень преступности-удаленность от радиальных магистралей. Включение в набор данных производных признаков называется ***конструированием признаков*** **(feature engineering)**, которое мы более подробно рассмотрим позже. Набор данных с производными признаками можно загрузить с помощью функции **load_extended_boston**:

```
X, y = mglearn.datasets.load_extended_boston()
print("Форма массива X: {}".format(X.shape))
```

```
Форма массива X: (506, 104)
```

Содержание работы

1. Ознакомиться со средой программирования Colab (можно пользоваться любым дистрибутивом Python по своему усмотрению)
2. Ознакомиться со способом задания массивов данных для обучения нейронных сетей (входные данные-выходные данные) и способом их деления на обучающее и тестовое множества.
3. Выделить числовые данные из набора **breast_cancer** или **Boston Housing (**или же любой другой набор**,** содержащий числовые и категориальные данные). Произвести варианты разделения этих данных на обучающее и тестовое множества.
4. Оформить отчет по лабораторной работе.

Содержание отчета

1. Цель работы.
2. Формулировка задания (для пунктов 2 и 3 из Содержания работы)
3. Программный код с комментариями
4. Результат
5. Выводы по лабораторной работе

Ниже представлена справочная информация для работы с NymPy-массивами

СПРАВОЧНЫЙ МАТЕРИАЛ

К ЭТОМУ МАТЕРИАЛУ ВЫ СМОЖЕТЕ ОБРАЩАТЬСЯ ПО МЕРЕ ПОДГОТОВКИ ОБУЧАЮЩИХ И ТЕСТОВЫХ ДАННЫХ

Основы работы с библиотекой NumPy

Мы рассмотрим базовые приёмы работы с многомерными массивами в NumPy, потому что этот материал потребуется нам как для низкоуровневой реализации нейронных сетей, так и для подготовки данных.

Понятие однородного многомерного массива

Основным объектом NumPy является *однородный многомерный массив*. Однородный – состоящий из однотипных элементов. Это таблица элементов (обычно чисел) одного типа, индексированных кортежем неотрицательных целых чисел.

В NumPy измерения называются *осями*. Понять использование осей в массиве Numpy не очень просто. Ось Numpy очень похожа на оси в декартовой системе координат. Оси numpy работают по-разному для одномерных (1D) и n-мерных массивов. Массивы 1D отличаются тем, что они имеют только одну ось. Оси Numpy нумеруются так же, как индексы Python, то есть они начинаются с 0. Поэтому в массиве 1D первой и единственной осью является ось 0.

Для 2D-и многомерных массивов:

Ось 0 (Направление вдоль строк) – Ось 0 называется первой осью массива Numpy. Эта ось 0 проходит вертикально вниз вдоль строк многомерных массивов Numpy, то есть выполняет операции по столбцам.

Ось 1 (Направление вместе со столбцами) – Ось 1 называется второй осью многомерных массивов Numpy. В результате ось 1 суммируется по горизонтали вместе со столбцами массивов. Он выполняет операции по строкам.

Ось Numpy – это тип направления, через которое начинается итерация. Каждая операция в numpy имеет определенный итерационный процесс, через который она проходит. Такие операции, как numpy sum (), np mean() и concatenate () достигаются путем передачи осей numpy в качестве параметров. Мы также можем перечислять данные массивов по их строкам и столбцам с помощью оси numpy.

Класс массива NumPy называется ***ndarray***. Он также известен под псевдонимом *array*. Обратите внимание, что numpy.array это не то же самое, что класс стандартной библиотеки Python array.array, который обрабатывает только одномерные массивы и предлагает меньшую функциональность. Наиболее важными атрибутами ndarrayобъекта являются:

ndarray.ndim – количество осей (размеров) массива.

ndarray.shape – размеры массива. Это кортеж целых чисел, указывающий размер массива в каждом измерении. Для матрицы с n строками и m столбцами shapebудет (n,m). Таким образом, длина shape кортежа равна количеству осей ndim.

ndarray.size – общее количество элементов массива. Это равно произведению элементов shape.

ndarray.dtype – объект, описывающий тип элементов в массиве. Можно создать или указать dtype, используя стандартные типы Python. Кроме того, NumPy предоставляет собственные типы. Например, numpy.int32, numpy.int16 и numpy.float64.

ndarray.itemsize – размер в байтах каждого элемента массива. Например, массив элементов типа float64имеет itemsize8 (=64/8), а один тип complex32имеет itemsize4 (=32/8). Это эквивалентно ndarray.dtype.itemsize.

ndarray.data – буфер, содержащий фактические элементы массива. Обычно нам не нужно использовать этот атрибут, потому что мы будем обращаться к элементам в массиве, используя средства индексирования.

Пример (курсивом печатаются ответы python-интерпрететора):

```
>>> import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<class 'numpy.ndarray'>
>>> b = np.array([6, 7, 8])
>>> b
array([6, 7, 8])
>>> type(b)
<class 'numpy.ndarray'>
```

Создание массива

Существует несколько способов создания массивов.

Например, можно создать массив из обычного списка или кортежа Python, используя функцию array(). Тип результирующего массива выводится из типа элементов в последовательностях.

```
>>> import numpy as np
```

```
>>> a = np.array([2, 3, 4])
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int64')
>>> b = np.array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

Частая ошибка заключается в вызове `array`с несколькими аргументами вместо предоставления одной последовательности в качестве аргумента.

```
>>> a = np.array(1, 2, 3, 4)    # WRONG
Traceback (most recent call last):
  ...
TypeError: array() takes from 1 to 2 positional arguments but 4
were given
>>> a = np.array([1, 2, 3, 4])  # RIGHT
```

Функция array()преобразует последовательности последовательностей в двумерные массивы, последовательности последовательностей последовательностей в трехмерные массивы и так далее.

```
>>> b = np.array([(1.5, 2, 3), (4, 5, 6)])
>>> b
array([[1.5, 2. , 3. ],
       [4. , 5. , 6. ]])
```

Тип массива также может быть явно указан во время создания:

```
>>> c = np.array([[1, 2], [3, 4]], dtype=complex)
>>> c
array([[1.+0.j, 2.+0.j],
       [3.+0.j, 4.+0.j]])
```

Часто элементы массива изначально неизвестны, но известен его размер. Следовательно, NumPy предлагает несколько функций для создания

массивов с начальным содержимым-заполнителем. Это сводит к минимуму необходимость выращивания массивов, что является дорогостоящей операцией.

Функция zeros() создает массив, полный нулей, функция onesсоздает массив, полный единиц, и функция empty создает массив, начальное содержимое которого является случайным и зависит от состояния памяти. По умолчанию dtype создаваемого массива — float64, но его можно указать через аргумент ключевого слова dtype.

```
>>> np.zeros((3, 4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> np.ones((2, 3, 4), dtype=np.int16)
array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],

       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=int16)
>>> np.empty((2, 3))
array([[3.73603959e-262, 6.02658058e-154, 6.55490914e-260],  # may
vary
       [5.30498948e-313, 3.14673309e-307, 1.00000000e+000]])
```

Для создания последовательностей чисел NumPy предоставляет функцию arange(), аналогичную встроенной в Python range(), но возвращающую массив.

```
>>> np.arange(10, 30, 5)
array([10, 15, 20, 25])
>>> np.arange(0, 2, 0.3)  # it accepts float arguments
array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])
```

Когда arange() используется с аргументами с плавающей запятой, как правило, невозможно предсказать количество полученных элементов из-за конечной точности с плавающей запятой. По этой причине обычно лучше использовать функцию linspace(), которая получает в качестве аргумента количество элементов, которое мы хотим, вместо шага:

```
>>> from numpy import pi
>>> np.linspace(0, 2, 9)                   # 9 numbers from 0 to 2
array([0.  , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.  ])
>>> x = np.linspace(0, 2 * pi, 100)        # useful to evaluate
function at lots of points
>>> f = np.sin(x)
```

Подобные функции: array, zeros, zeros_like, ones, ones_like, empty, empty_like, arange, numpy.random.Generator.rand, numpy.random.Generator.randn, linspace_fromfunctionfromfile

Печать массивов

Когда вы печатаете массив, NumPy отображает его аналогично вложенным спискам, но со следующим макетом:

- последняя ось печатается слева направо,
- предпоследний печатается сверху вниз,
- остальные также печатаются сверху вниз, при этом каждый фрагмент отделяется от следующего пустой строкой.

Затем одномерные массивы печатаются как строки, двумерные — как матрицы, а трехмерные — как списки матриц.

```
>>> a = np.arange(6)                         # 1d array
>>> print(a)
[0 1 2 3 4 5]
>>>
>>> b = np.arange(12).reshape(4, 3)          # 2d array
>>> print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>>
>>> c = np.arange(24).reshape(2, 3, 4)       # 3d array
>>> print(c)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]
```

```
 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

Если массив слишком велик для печати, NumPy автоматически пропускает центральную часть массива и печатает только углы:

```
>>> print(np.arange(10000))
[   0    1    2 ... 9997 9998 9999]
>>>
>>> print(np.arange(10000).reshape(100, 100))
[[   0    1    2 ...   97   98   99]
 [ 100  101  102 ...  197  198  199]
 [ 200  201  202 ...  297  298  299]
 ...
 [9700 9701 9702 ... 9797 9798 9799]
 [9800 9801 9802 ... 9897 9898 9899]
 [9900 9901 9902 ... 9997 9998 9999]]
```

Чтобы отключить это поведение и заставить NumPy печатать весь массив, можно изменить параметры печати с помощью `set_printoptions`:

```
np.set_printoptions(threshold=sys.maxsize)  # sys module should be
imported
```

Основные операции

Арифметические операторы к массивам применяются поэлементно. Новый массив создается и заполняется результатом.

```
a = np.array([20, 30, 40, 50])
>>> b = np.arange(4)
>>> b
array([0, 1, 2, 3])
>>> c = a - b
>>> c
array([20, 29, 38, 47])
```

```
>>> b**2
array([0, 1, 4, 9])
>>> 10 * np.sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
>>> a < 35
array([ True,  True, False, False])
```

В отличие от многих матричных языков, оператор продукта «*» работает с массивами NumPy поэлементно. Произведение матриц можно выполнить с помощью оператора «@» (в python >=3.5) или функции dot() или метода:

```
>>> A = np.array([[1, 1],
...               [0, 1]])
>>> B = np.array([[2, 0],
...               [3, 4]])
>>> A * B      # elementwise product
array([[2, 0],
       [0, 4]])
>>> A @ B      # matrix product
array([[5, 4],
       [3, 4]])
>>> A.dot(B)   # another matrix product
array([[5, 4],
       [3, 4]])
```

Некоторые операции, такие как +=и *=, действуют на месте для изменения существующего массива, а не для создания нового.

```
>>> rg = np.random.default_rng(1)  # create instance of default
random number generator
>>> a = np.ones((2, 3), dtype=int)
>>> b = rg.random((2, 3))
>>> a *= 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
>>> b += a
>>> b
```

```
array([[3.51182162, 3.9504637 , 3.14415961],
       [3.94864945, 3.31183145, 3.42332645]])
>>> a += b  # b is not automatically converted to integer type
Traceback (most recent call last):
    ...
numpy.core._exceptions._UFuncOutputCastingError: Cannot cast ufunc
'add' output from dtype('float64') to dtype('int64') with casting
rule 'same_kind'
```

При работе с массивами разных типов тип результирующего массива соответствует более общему или точному типу (поведение, известное как восходящее приведение).

```
>>> a = np.ones(3, dtype=np.int32)
>>> b = np.linspace(0, pi, 3)
>>> b.dtype.name
'float64'
>>> c = a + b
>>> c
array([1.        , 2.57079633, 4.14159265])
>>> c.dtype.name
'float64'
>>> d = np.exp(c * 1j)
>>> d
array([ 0.54030231+0.84147098j, -0.84147098+0.54030231j,
       -0.54030231-0.84147098j])
>>> d.dtype.name
'complex128'
```

Многие унарные операции, такие как вычисление суммы всех элементов массива, реализованы как методы `ndarray` класса.

```
>>> a = rg.random((2, 3))
>>> a
array([[0.82770259, 0.40919914, 0.54959369],
       [0.02755911, 0.75351311, 0.53814331]])
>>> a.sum()
3.1057109529998157
>>> a.min()
0.027559113243068367
```

```
>>> a.max()
0.8277025938204418
```

По умолчанию эти операции применяются к массиву, как если бы он был списком чисел, независимо от его формы. Однако, указав `axis` параметр, можно применить операцию вдоль указанной оси массива:

```
>>> b = np.arange(12).reshape(3, 4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> b.sum(axis=0)      # sum of each column
array([12, 15, 18, 21])
>>>
>>> b.min(axis=1)      # min of each row
array([0, 4, 8])
>>>
>>> b.cumsum(axis=1)   # cumulative sum along each row
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

Универсальные функции

NumPy предоставляет знакомые математические функции, такие как sin, cos и exp. В NumPy они называются «универсальными функциями» (`ufunc`). В NumPy эти функции работают с массивом поэлементно, создавая массив в качестве вывода.

```
>>> B = np.arange(3)
>>> B
array([0, 1, 2])
>>> np.exp(B)
array([1.        , 2.71828183, 7.3890561 ])
>>> np.sqrt(B)
array([0.        , 1.        , 1.41421356])
>>> C = np.array([2., -1., 4.])
>>> np.add(B, C)
```

```
array([2., 0., 6.])
```

Смотрите также функции:
all, **any**, **apply_along_axis**, **argmax**, **argmin**, **argsort**, **average**, **bincount**, **ceil**, **clip**, **conj**, **corrcoef**, **cov**, **cross**, **cumprod**, **cumsum**, **diff**, **dot**, **floor**, **inner**, **invert**, **lexsort**, **max**, **maximum**, **mean**, **median**, **min**, **minimum**, **nonzero**, **outer**, **prod**, **re**, **round**, **sort**, **std**, **sum**, **trace**, , **transpose**_ _ _ **varvdotvectorizewhere**

Индексирование, нарезка и итерация

Одномерные массивы можно индексировать, нарезать и повторять, как списки и другие последовательности Python.

```
>>> a = np.arange(10)**3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[2]
8
>>> a[2:5]
array([ 8, 27, 64])
>>> # equivalent to a[0:6:2] = 1000;
>>> # from start to position 6, exclusive, set every 2nd element to
1000
>>> a[:6:2] = 1000
>>> a
array([1000,    1, 1000,   27, 1000,  125,  216,  343,  512,  729])
>>> a[::-1]  # reversed a
array([ 729,  512,  343,  216,  125, 1000,   27, 1000,    1, 1000])
>>> for i in a:
...     print(i**(1 / 3.))
...
9.999999999999998
1.0
9.999999999999998
3.0
9.999999999999998
4.999999999999999
5.999999999999999
6.999999999999999
7.999999999999999
8.999999999999998
```

Многомерные массивы могут иметь один индекс на ось. Эти индексы приведены в кортеже, разделенном запятыми:

```
>>> def f(x, y):
...     return 10 * x + y
...
>>> b = np.fromfunction(f, (5, 4), dtype=int)
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> b[2, 3]
23
>>> b[0:5, 1]  # each row in the second column of b
array([ 1, 11, 21, 31, 41])
>>> b[:, 1]    # equivalent to the previous example
array([ 1, 11, 21, 31, 41])
>>> b[1:3, :]  # each column in the second and third row of b
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

Когда указано меньше индексов, чем количество осей, отсутствующие индексы считаются полными срезами:

```
>>> b[-1]   # the last row. Equivalent to b[-1, :]
array([40, 41, 42, 43])
```

Выражение в квадратных скобках `b[i]` рассматривается как за `i` которым следует столько экземпляров, `:` сколько необходимо для представления остальных осей. NumPy также позволяет вам писать это, используя точки как `.b[i, ...]`

Точки ( ) представляют столько двоеточий, сколько необходимо для создания полного кортежа индексации `. . .` Например, если `x` это массив с 5 осями, то

`x[1, 2, ...]` эквивалентно `x[1, 2, :, :, :]`
`x[..., 3]` эквивалентно `x[:, :, :, :, 3]`
`x[4, ..., 5, :]` эквивалентно `x[4, :, :, 5, :]`

```
>>> c = np.array([[[  0,  1,  2],  # a 3D array (two stacked 2D
arrays)
...                [ 10, 12, 13]],
...               [[100, 101, 102],
...                [110, 112, 113]]])
>>> c.shape
(2, 2, 3)
>>> c[1, ...]  # same as c[1, :, :] or c[1]
array([[100, 101, 102],
       [110, 112, 113]])
>>> c[..., 2]  # same as c[:, :, 2]
array([[  2,  13],
       [102, 113]])
```

Перебор многомерных массивов выполняется относительно первой оси:

```
>>> for row in b:
...     print(row)
...
[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

Однако, если кто-то хочет выполнить операцию над каждым элементом массива, можно использовать атрибут `flat`, который является итератором для всех элементов массива:

```
>>> for element in b.flat:
...     print(element)
...
0
1
2
3
10
```

```
11
12
13
20
21
22
23
30
31
32
33
40
41
42
43
```

Изменение формы массива

Массив имеет форму, заданную количеством элементов вдоль каждой оси:

```
>>> a = np.floor(10 * rg.random((3, 4)))
>>> a
array([[3., 7., 3., 4.],
       [1., 4., 2., 2.],
       [7., 2., 4., 9.]])
>>> a.shape
(3, 4)
```

Форму массива можно изменить с помощью различных команд. Обратите внимание, что все следующие три команды возвращают измененный массив, но не изменяют исходный массив:

```
>>> a.ravel()  # returns the array, flattened
array([3., 7., 3., 4., 1., 4., 2., 2., 7., 2., 4., 9.])
>>> a.reshape(6, 2)  # returns the array with a modified shape
array([[3., 7.],
       [3., 4.],
       [1., 4.],
       [2., 2.],
```

```
       [7., 2.],
       [4., 9.]])
>>> a.T  # returns the array, transposed
array([[3., 1., 7.],
       [7., 4., 2.],
       [3., 2., 4.],
       [4., 2., 9.]])
>>> a.T.shape
(4, 3)
>>> a.shape
(3, 4)
```

Порядок элементов в массиве, полученном из ravel(), обычно соответствует «стилю C», то есть крайний правый индекс «меняется быстрее всего», поэтому элемент после . Если массив преобразуется в какую-либо другую форму, массив снова обрабатывается как «C-стиль». NumPy обычно создает массивы, хранящиеся в этом порядке, поэтому обычно не нужно копировать его аргумент, но если массив был создан путем взятия фрагментов другого массива или создан с необычными параметрами, его может потребоваться скопировать. Функциям и также можно указать, используя необязательный аргумент, использовать массивы в стиле FORTRAN, в которых самый левый индекс изменяется быстрее всего. `a[0, 0]a[0, 1]ravelravelreshape`

Функция reshape() возвращает свой аргумент с измененной формой, тогда как **`ndarray.resize`**метод изменяет сам массив:

```
>>> a
array([[3., 7., 3., 4.],
       [1., 4., 2., 2.],
       [7., 2., 4., 9.]])
>>> a.resize((2, 6))
>>> a
array([[3., 7., 3., 4., 1., 4.],
       [2., 2., 7., 2., 4., 9.]])
```

Если размер задан как `-1`в операции изменения формы, другие размеры рассчитываются автоматически:

```
>>> a.reshape(3, -1)
array([[3., 7., 3., 4.],
```

```
       [1., 4., 2., 2.],
       [7., 2., 4., 9.]])
```

Смотрите также: **ndarray.shape**, **reshape**, **resize**, **ravel**

Объединение разных массивов вместе

Несколько массивов могут быть сложены вместе по разным осям:

```
>>> a = np.floor(10 * rg.random((2, 2)))
>>> a
array([[9., 7.],
       [5., 2.]])
>>> b = np.floor(10 * rg.random((2, 2)))
>>> b
array([[1., 9.],
       [5., 1.]])
>>> np.vstack((a, b))
array([[9., 7.],
       [5., 2.],
       [1., 9.],
       [5., 1.]])
>>> np.hstack((a, b))
array([[9., 7., 1., 9.],
       [5., 2., 5., 1.]])
```

Функция column_stack() объединяет одномерные массивы в виде столбцов в двухмерный массив. Это эквивалентно hstack() только для 2D-массивов:

```
>>> from numpy import newaxis
>>> np.column_stack((a, b))  # with 2D arrays
array([[9., 7., 1., 9.],
       [5., 2., 5., 1.]])
>>> a = np.array([4., 2.])
>>> b = np.array([3., 8.])
>>> np.column_stack((a, b))  # returns a 2D array
array([[4., 3.],
       [2., 8.]])
```

```
>>> np.hstack((a, b))         # the result is different
array([4., 2., 3., 8.])
>>> a[:, newaxis]  # view `a` as a 2D column vector
array([[4.],
       [2.]])
>>> np.column_stack((a[:, newaxis], b[:, newaxis]))
array([[4., 3.],
       [2., 8.]])
>>> np.hstack((a[:, newaxis], b[:, newaxis]))  # the result is the
same
array([[4., 3.],
       [2., 8.]])
```

С другой стороны, функция **row_stack**эквивалентна **vstack** для любых входных массивов. На самом деле, **row_stack**это псевдоним для **vstack**:

```
>>> np.column_stack is np.hstack
False
>>> np.row_stack is np.vstack
True
```

В общем, для массивов с более чем двумя измерениями, **hstack** стекируется по их вторым осям, **vstack** складывается по их первым осям и **concatenate** допускает необязательные аргументы, задающие номер оси, вдоль которой должно происходить объединение.

*Примечание***.** В сложных случаях **r_**и **c_**полезны для создания массивов путем укладки чисел вдоль одной оси. Они позволяют использовать литералы диапазона:

```
>>> np.r_[1:4, 0, 4]
array([1, 2, 3, 0, 4])
```

При использовании с массивами в качестве аргументов **r_**и **c_**аналогичны **vstack**и **hstack**в своем поведении по умолчанию, но допускают необязательный аргумент, указывающий номер оси, вдоль которой выполняется конкатенация.

Смотрите также: **hstack**, **vstack**, **column_stack**, **concatenate**, **c_**, **r_**

Разделение одного массива на несколько меньших

Используя **hsplit**, можно разбить массив вдоль его горизонтальной оси, либо указав количество возвращаемых массивов одинаковой формы, либо указав столбцы, после которых должно происходить деление:

```
>>> a = np.floor(10 * rg.random((2, 12)))
>>> a
array([[6., 7., 6., 9., 0., 5., 4., 0., 6., 8., 5., 2.],
       [8., 5., 5., 7., 1., 8., 6., 7., 1., 8., 1., 0.]])
>>> # Split `a` into 3
>>> np.hsplit(a, 3)
[array([[6., 7., 6., 9.],
       [8., 5., 5., 7.]]), array([[0., 5., 4., 0.],
       [1., 8., 6., 7.]]), array([[6., 8., 5., 2.],
       [1., 8., 1., 0.]])]
>>> # Split `a` after the third and the fourth column
>>> np.hsplit(a, (3, 4))
[array([[6., 7., 6.],
       [8., 5., 5.]]), array([[9.],
       [7.]]), array([[0., 5., 4., 0., 6., 8., 5., 2.],
       [1., 8., 6., 7., 1., 8., 1., 0.]])]
```

vsplit разбивается по вертикальной оси и **array_split** позволяет указать, по какой оси разбиваться.

Копии и представления

При работе с массивами их данные иногда копируются в новый массив, а иногда нет. Это часто является источником путаницы для начинающих. Есть три случая:

1. Никакого копирования

Простые присваивания не копируют объекты или их данные.

```
>>> a = np.array([[ 0,  1,  2,  3],
...               [ 4,  5,  6,  7],
...               [ 8,  9, 10, 11]])
>>> b = a            # no new object is created
>>> b is a           # a and b are two names for the same ndarray
object
True
```

Python передает изменяемые объекты как ссылки, поэтому вызовы функций не копируют.

```
>>> def f(x):
...     print(id(x))
...
>>> id(a)  # id is a unique identifier of an object
148293216  # may vary
>>> f(a)
148293216  # may vary
```

2. Просмотр или поверхностное копирование

Одни и те же данные могут использоваться разными объектами массива. Метод `view`создает новый объект массива, который просматривает те же данные.

```
>>> c = a.view()
>>> c is a
False
>>> c.base is a                        # c is a view of the data owned by a
True
>>> c.flags.owndata
False
>>>
>>> c = c.reshape((2, 6))  # a's shape doesn't change
>>> a.shape
(3, 4)
>>> c[0, 4] = 1234                     # a's data changes
>>> a
array([[   0,    1,    2,    3],
       [1234,    5,    6,    7],
       [   8,    9,   10,   11]])
```

Нарезка массива возвращает его представление:

```
>>> s = a[:, 1:3]
>>> s[:] = 10  # s[:] is a view of s. Note the difference between s
= 10 and s[:] = 10
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
```

```
       [   8,   10,   10,   11]])
```

3. Глубокое копирование

Метод `copy` делает полную копию массива и его данных.

```
>>> d = a.copy()  # a new array object with new data is created
>>> d is a
False
>>> d.base is a  # d doesn't share anything with a
False
>>> d[0, 0] = 9999
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

Иногда `copy` следует вызывать после нарезки, если исходный массив больше не требуется. Например, предположим, что `a` это огромный промежуточный результат, а конечный результат `b` содержит лишь небольшую часть , при построении с нарезкой `a` следует сделать глубокую копию :`b`

```
>>> a = np.arange(int(1e8))
>>> b = a[:100].copy()
>>> del a  # the memory of ``a`` can be released.
```

Если вместо этого используется, на него ссылается и будет сохраняться в памяти, даже если выполняется.`b = a[:100]abdel a`

Обзор функций и методов

Вот список некоторых полезных функций и методов NumPy, упорядоченных по категориям.

Создание массива:

arange, **array**, **copy**, **empty**, **empty_like**, **eye**, **fromfile**, **fromfunction**, **identity**, **linspace**, **logspace**, **mgrid**, **ogrid**, **ones**, **ones_like**, **r_**, **zeros_ zeros_like**

Конверсии

ndarray.astype, **atleast_1d**, **atleast_2d**, **atleast_3d**, **mat**

Манипуляции
array_split, column_stack, concatenate, diagonal, dsplit, dstack, hsplit, hstack, ndarray.item, newaxis, ravel, repeat, reshape, resize, squeeze, swapaxes, take, transpose, vsplit_ vstack

Вопросы
all, any, nonzero, where

Заказ
argmax, argmin, argsort, max, min, ptp, searchsorted, sort

Операции
choose, compress, cumprod, cumsum, inner, ndarray.fill, imag, prod, put, putmask, real, sum

Основная статистика
cov, mean, std, var

Базовая линейная алгебра
cross, dot, outer, linalg.svd, vdot

«Автоматическое» изменение формы

Чтобы изменить размеры массива, вы можете опустить один из размеров, который затем будет выведен автоматически:

```
>>> a = np.arange(30)
>>> b = a.reshape((2, -1, 3))  # -1 means "whatever is needed"
>>> b.shape
(2, 5, 3)
>>> b
array([[[ 0,  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, 26],
        [27, 28, 29]]])
```

Наложение векторов

Как построить двумерный массив из списка векторов-строк одинакового размера? В MATLAB это довольно просто: если `x` и `y` два вектора одинаковой длины, нужно только сделать `m=[x;y]`. В NumPy это работает с помощью функций `column_stack()` и `dstack()` в зависимости от измерения, в котором должно выполняться наложение. Например: `hstackvstack`

```
>>> x = np.arange(0, 10, 2)
>>> y = np.arange(5)
>>> m = np.vstack([x, y])
>>> m
array([[0, 2, 4, 6, 8],
       [0, 1, 2, 3, 4]])
>>> xy = np.hstack([x, y])
>>> xy
array([0, 2, 4, 6, 8, 0, 1, 2, 3, 4])
```

Логика этих функций в более чем двух измерениях может быть странной.

Если приведенной информации Вам не достаточно, можно обратиться к официальной документации: NumPy: абсолютные основы для начинающих: **https://numpy.org/doc/stable/user/absolute_beginners.html**

Заключение

Использование высокоуровневых фреймворков, таких как Keras, TensorFlow, PyTorch и других, позволяет быстро создавать очень сложные нейросетевые модели. Тем не менее, стоит потратить время, чтобы заглянуть внутрь нейросетей и понять основные концепции. Поэтому далее мы сначала попытаемся использовать базовые знания о принципах работы и обучения нейронных сетей и построить работоспособные нейронные сети, используя только NumPy.

А затем перейдем к реализации нейросетей в отдельных высокоуровневых библиотеках.

Подведем итоги:

- Вы ознакомились с перечнем наиболее популярных библиотек, которые можно использовать для создания нейросетевых приложений.
- Вы научились понимать разницу между одномерными, двумерными и n-мерными массивами в NumPy;

- Вы ознакомились с применением некоторых операций линейной алгебры к n-мерным массивам без использования циклов for;
- Вы ознакомились со свойствами оси и формы для n-мерных массивов.
- Вы получили базовые сведения по работе с numpy-массивами.