Системы искусственного интеллекта

02.03.03 - Математическое обеспечение и администрирование информационных систем, направленность (профиль) - разработка и администрирование информационных систем
09.03.03 - Прикладная информатика, направленность (профиль) - прикладная информатика в экономике

http//vikchas.ru

Тема 1.Введение в искусственный интеллект и машинное обучение

Лекция 5 «Создание интеллектуальных агентов»

Часовских Виктор Петрович
д-р техн. наук, профессор кафедры ШИиКМ

ФГБОУ ВО «Уральский государственный экономический университет»

Екатеринбург 2023

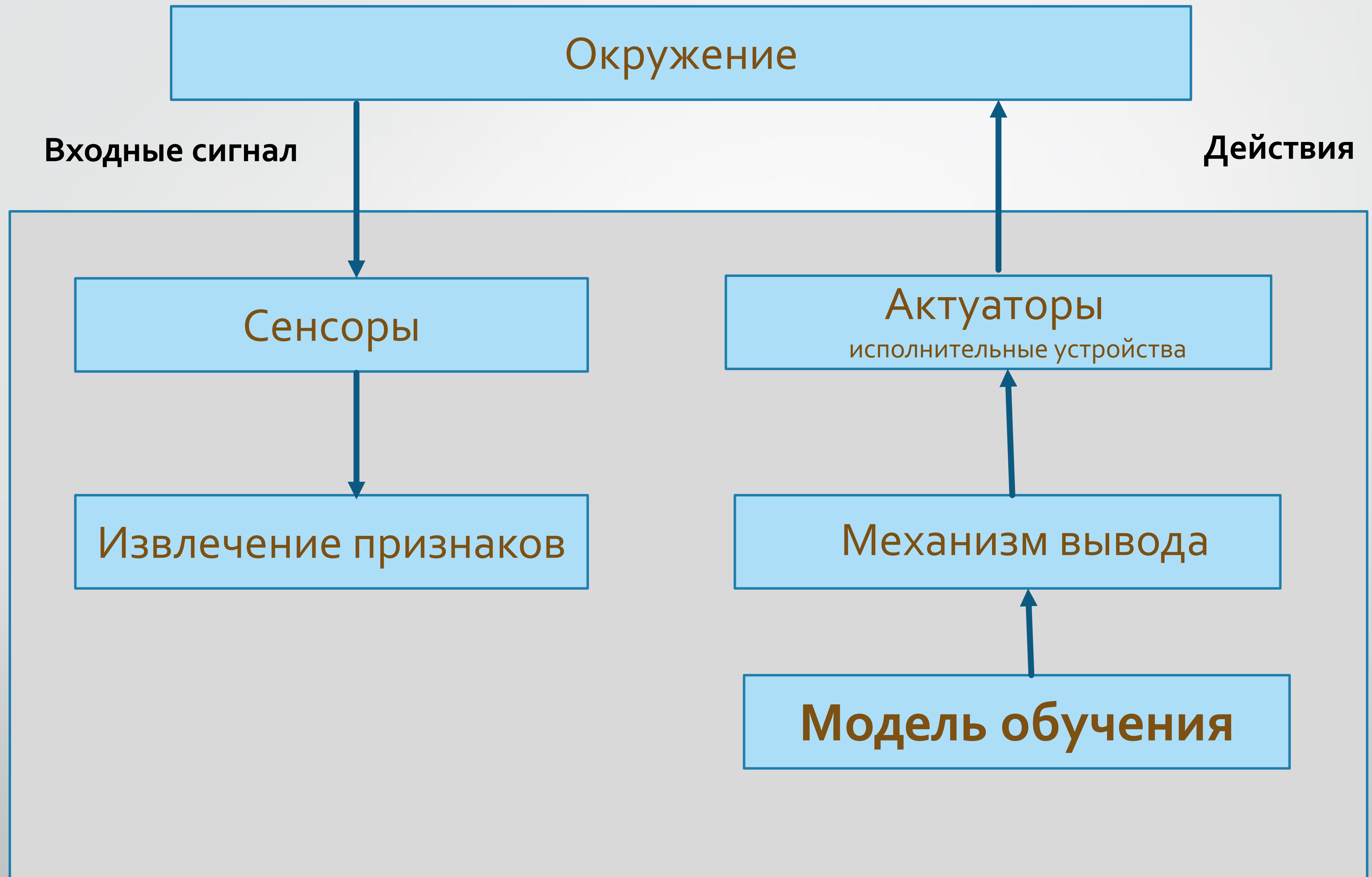
Создание интеллектуальных агентов

Существуют разные способы создания интеллектуальных агентов. К числу наиболее распространенных из них относятся **машинное обучение**, базы знаний, наборы правил и др.

Рассмотрим машинное обучение.

Данный метод предполагает наделение агента интеллектуальными способностями посредством тренировки (обучения) на **известных данных**.

Н



Окружение
Входные сигнал
Действия
Сенсоры
Актуаторы
исполнительные устройства
Извлечение признаков
Механизм вывода
Модель обучения
Интеллектуальный агент

Для применения машинного обучения необходимо подготовить размеченные данные, выбранные из базы данных (как правило BigData). Эти данные будут использоваться для формирования начальной модели, позволяющей распознавать образы и отношения между ними.

В настоящее время машинное обучение находит множество применений.

Оно используется для распознавания образов и речи, прогнозирования рыночных тенденций, в робототехнике и других областях.

Чтобы понять сущность машинною обучения и научиться создавать полные решения необходимо концептуально понимать распознавание образов, искусственные нейронные сети, добыча данных, статистика и т.п. и их математическую базу.

В ИИ существует два типа моделей: **аналитические и обучаемые**.

Аналитические модели основываются на математических формулировках, представляющих собой, по сути, описание последовательных шагов, которые требовалось выполнить для получения окончательного уравнения. Проблемой этого подхода является то, что он зависит от суждений человека. Как следствие, подобные модели были упрощенными и страдали неточностью, обусловленной недостаточно большим количеством параметров.

Появление компьютеров позволило использовать обучаемые модели.

Такие модели создаются посредством тренировки. Для получения уравнения в процессе тренировки просматривает множество примеров входных и соответствующих выходных данных.

Подобным обучаемым моделям свойственны сложность и высокая точность, поскольку в них учитываются тысячи параметров. Это приводит к тому, что результирующее уравнение, управляющее данными, оказывается чрезвычайно сложным.

Методы машинного обучения позволяют получать такие обучаемые модели, которые могут быть использованы в механизме вывода.

Наиболее благоприятным для нас следствием этого факта является то, что в данном случае мы избавлены от необходимости выводить базовые математические формулы. От нас не требуется владение сложным математическим аппаратом, поскольку компьютер извлекает эти формулы на основании данных. Все, что мы должны сделать, — предоставить соответствующие списки входных и выходных значений.

Обученная модель, которую мы при этом получаем, всего лишь отражает отношения между маркированными входными и желаемыми выходными значениями.

Математические функции и их производные

Каждое математическое представление будем рассматривать с трех сторон:

- математическое представление в виде **формулы** или набора уравнений;
- **код**, по возможности содержащий минимальное количество дополнительного синтаксиса (для этой цели идеально подходит язык Python);
- **рисунок** или схема, иллюстрирующие происходящий процесс.

Благодаря такому подходу мы сможем исчерпывающе понять, как и почему работают вложенные математические функции.

Функции

Математическое представление

Рассмотрим два примера функций в математической форме записи:

- $f_1(x) = x^2$
- $f_2(x) = \max(x,0)$

Записи означают, что функция f_1 преобразует входное значение x в x^2, а функция f_2 возвращает наибольшее значение из набора $(x, 0)$.

Визуализация

Еще один способ представления функций:

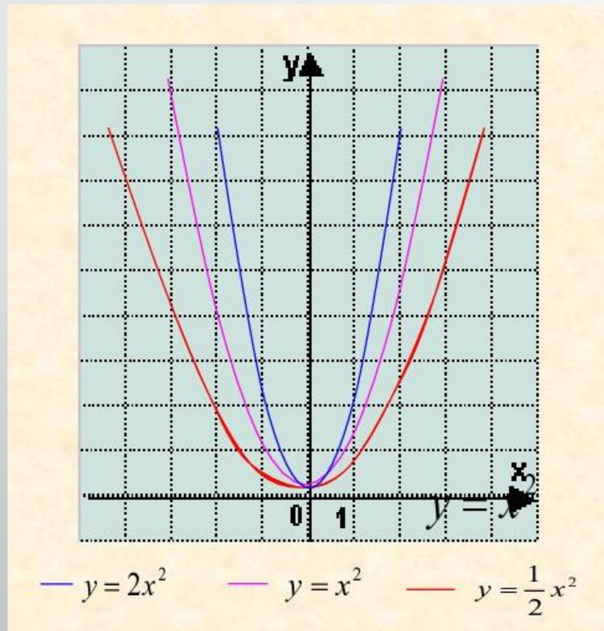
1. Нарисовать плоскость xy (где x соответствует горизонтальной оси, а y — вертикальной).

2. Нарисовать на этой плоскости набор точек, x-координаты которых (обычно равномерно распределенные) соответствуют входным значениям функции, а y-координаты — ее выходным значениям.

3. Соединить эти точки друг с другом. Получим графики.

Квадратичная функция

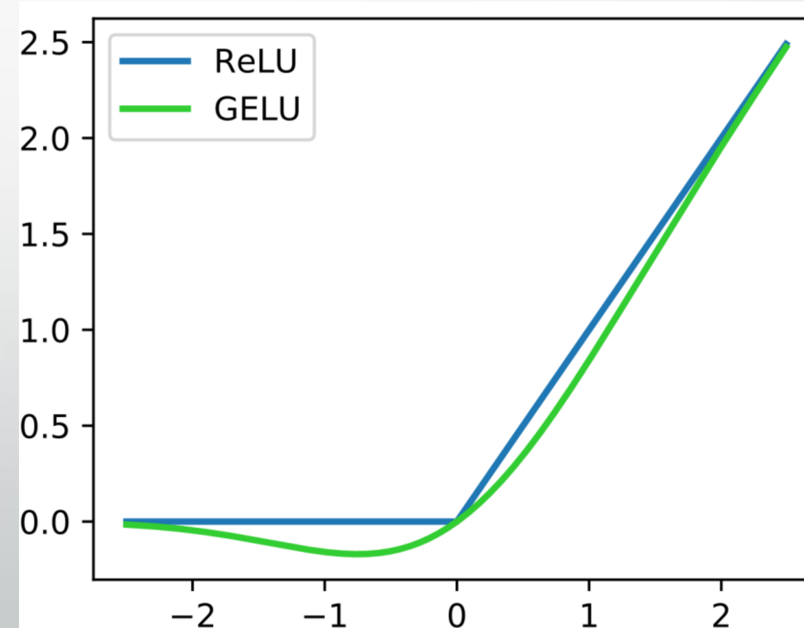
Выходные данные



Входные данные

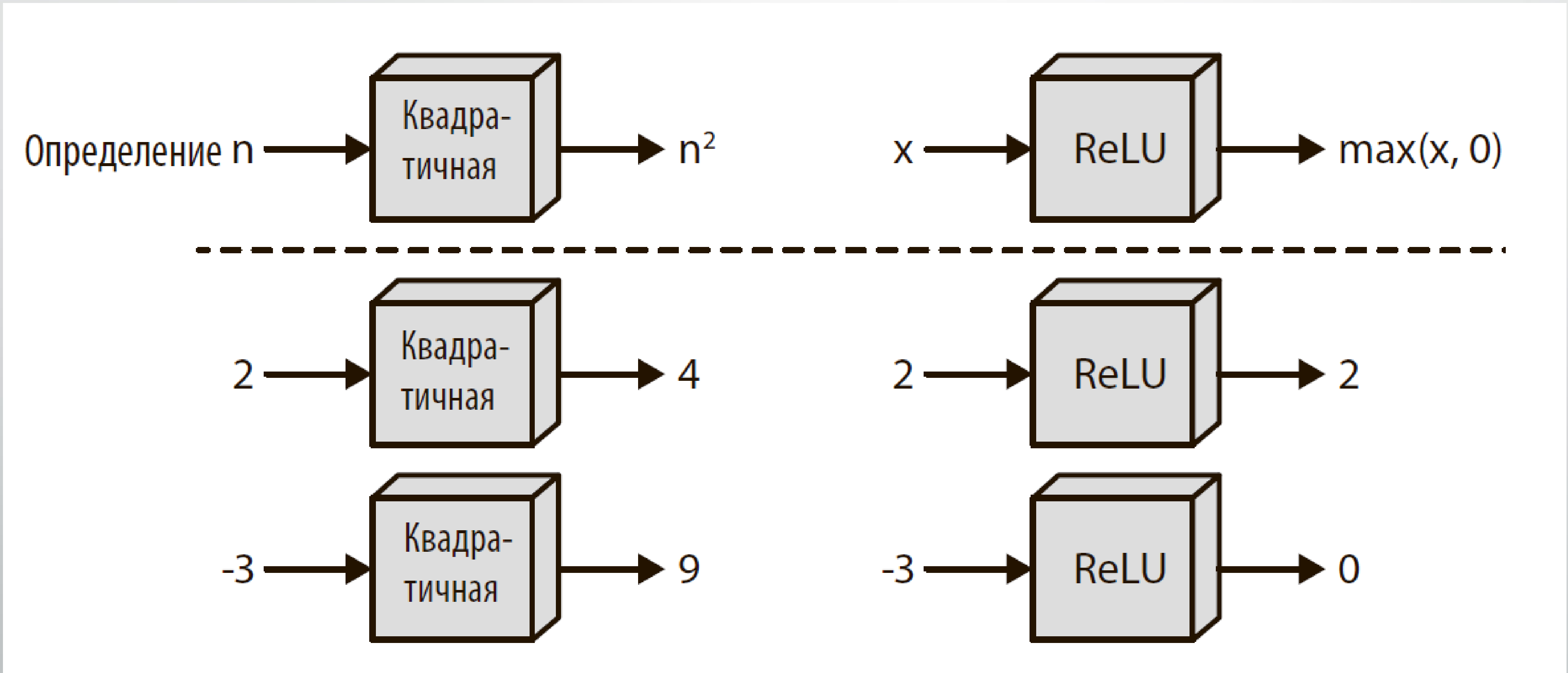
Нелинейная (активационная) функция ReLU и GELU

Выходные данные



Входные данные

Иной способ представления тех же функций



Для машинного обучения и нейронных сетей необходимо применять билиотеку Python **NumPy.** NumPy сокращен от Numeric Python, это самая универсальная библиотека.

Используя этот инструмент, можно легко и комфортно работать с многомерными массивами и матрицами.

Такие функции, как операции линейной алгебры и числовые преобразования также доступны.

Библиотека числового Python (NumPy) в Python одна из самых важных библиотек, когда дело доходит до численных вычислений, связанных со статистикой, и, поскольку большая часть науки о данных и машинного обучения вращается вокруг статистики, становится гораздо важнее иметь практическую работу с библиотекой.

Объекты **ndarray** из библиотеки **NumPy** дают возможность интуитивно и быстро работать с этими массивами.

Основной элемент библиотеки NumPy — объект **ndarray** (что значит N-размерный массив). Этот объект является многомерным однородным массивом с заранее заданным количеством элементов. Однородный — потому что практически все объекты в нем одного размера или типа. На самом деле, тип данных определен другим объектом NumPy, который называется **dtype** (тип-данных). Каждый ndarray ассоциирован только с одним типом dtype.

Например, если сохранить данные в виде обычного или многомерного списка, обычный синтаксис языка Python не позволит выполнить поэлементное сложение или умножение списков, зато эти операции прекрасно реализуются с помощью объектов **ndarray**:

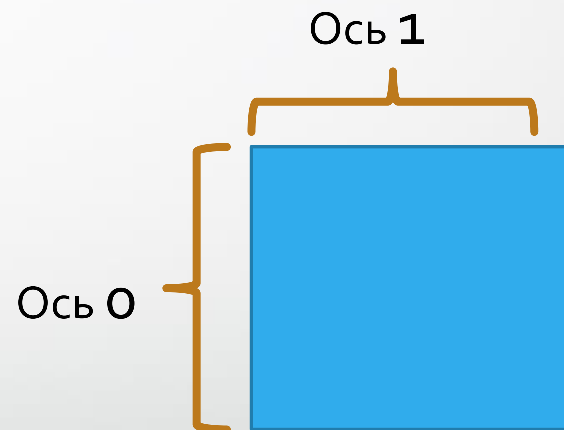
```
print("операции со списками на языке Python:")
a = [1,2,3]
b = [4,5,6]
print("a+b:", a+b)
try:
    print(a*b)
except TypeError:
    print("a*b не имеет смысла для списков в
языке Python")
print()
print("операции с массивами из библиотеки
numpy:")
a = np.array([1,2,3])
b = np.array([4,5,6])
print("a+b:", a+b)
print("a*b:", a*b)
```

```
операции со списками на языке Python:
a+b: [1, 2, 3, 4, 5, 6]
a*b не имеет смысла для списков в языке Python

 операции с массивами из библиотеки numpy:
 a+b: [5 7 9]
 a*b: [ 4 10 18]
```

Объект **ndarray** обладает и таким важным для работы с многомерными массивами атрибутом, как количество измерений. Измерения еще называют осями. Их нумерация начинается с 0, соответственно первая ось будет иметь индекс 0, вторая — 1 и т. д. В частном случае двумерного массива нулевую ось можно сопоставить строкам, а первую — столбцам,

```
print('a:')
print(a)
print('a.sum(axis=0):', a.sum(axis=0))
print('a.sum(axis=1):', a.sum(axis=1))
a:
[[1 2]
[3 4]]
a.sum(axis=0): [4 6]
a.sum(axis=1): [3 7]
```



Двумерный массив из библиотеки NumPy, в котором ось с индексом 0 соответствует строкам, а ось с индексом 1 — столбцам

Объект **ndarray** поддерживает такую операцию, как сложение с одномерным массивом. Например, к двумерному массиву *a*, состоящему из *R* строк и *C* столбцов, можно прибавить одномерный массив *b* длиной *C*, и библиотека NumPy выполнит сложение для элементов каждой строки массива a:

```
a = np.array([[1,2,3],
[4,5,6]])
b = np.array([10,20,30])
print("a+b:\n", a+b)
```

```
a+b:
[[11 22 33]
[14 25 36]]
```

Для примера слайда 8 напишем КОД

```
def square(x: ndarray) -> ndarray:
'''
Возведение в квадрат каждого элемента объекта ndarray.
'''
return np.power(x, 2)
def leaky_relu(x: ndarray) -> ndarray:
'''
Применение функции "Leaky ReLU" к каждому элементу ndarray.
'''
return np.maximum(0.2 * x, x)
```

Библиотека **NumPy** позволяет применять многие функции к объектам ndarray двумя способами: np.function_name (ndarray) или ndarray.function_name. Например, функцию relu можно было написать как x.clip (min = 0). В дальнейшем мы будем пользоваться
записью вида np.function_name (ndarray). И даже когда альтернативная запись короче, как, например, в случае транспонирования двумерного объекта ndarray, мы будем писать не ndarray.T, а np. trans pose (ndarray, (1, 0)).

Производные

Понятие производной функции, скорее всего, многим из вас уже знакомо. Производную можно определить как скорость изменения функции в рассматриваемой точке. Мы подробно рассмотрим это понятие с разных сторон.

Математическое представление

Математически производная определяется как предел отношения приращения функции к приращению ее аргумента при стремлении приращения аргумента к нулю:

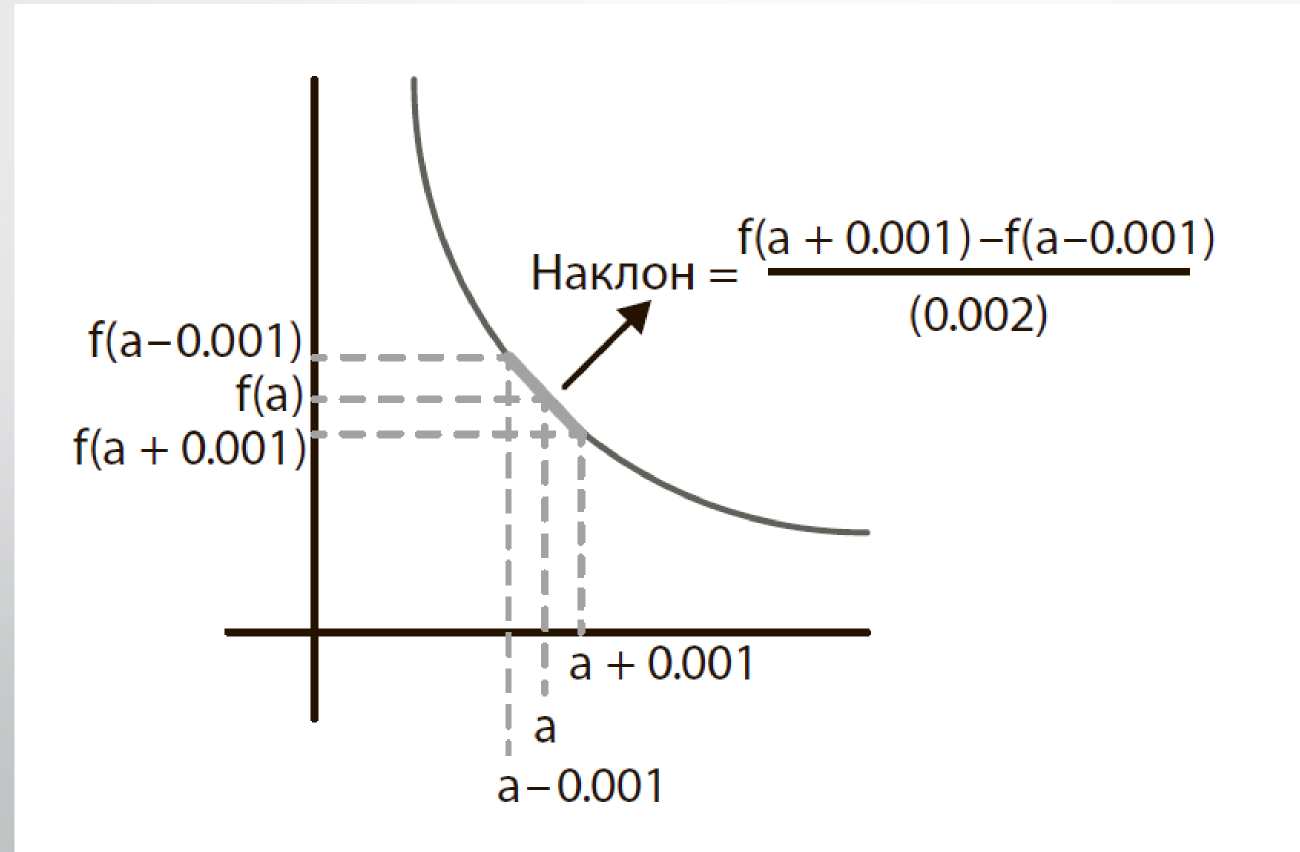
$$\frac{df}{du}(a) = \lim_{\Delta \to 0} \frac{f(a+\Delta) - f(a-\Delta)}{2 * \Delta}$$

Можно численно оценить этот предел, присвоив переменной Δ маленькое значение, например 0.001:

$$\frac{df}{du}(a) = \lim_{\Delta \to 0.001} \frac{f(a+0.001) - f(a-0.001)}{2 * 0.001}$$

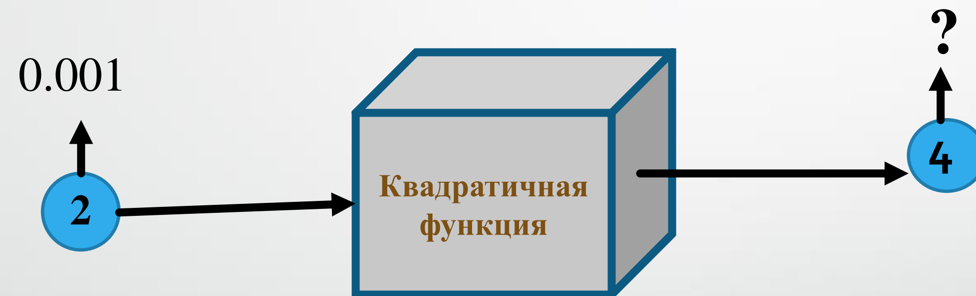
Визуализация

Начнем с общеизвестного способа: если нарисовать касательную к декартову представлению функции f, производная функции в точке касания будет равна угловому коэффициенту касательной. Вычислить этот коэффициент, или тангенс угла наклона прямой, можно, взяв разность значений функции f при $a - 0.001$ и $a + 0.001$ и поделив на величину приращения.



Производная как угловой коэффициент

На рисунке производную можно представить в виде множителя, кратно которому меняется выходное значение функции при небольшом изменении подаваемого на вход значения. Фактически мы меняем значение входного параметра на очень маленькую величину и смотрим, как при этом поменялось значение на выходе.



Альтернативный способ визуализации концепции производной

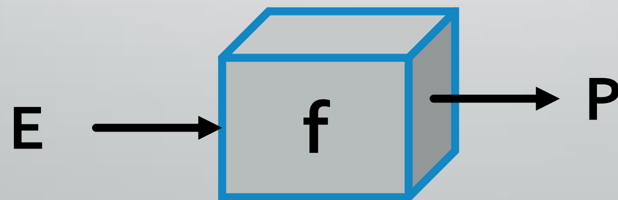
Код

Код для вычисления приблизительного значения производной:

```
from typing import Callable

def deriv(func: Callable[[ndarray], ndarray], input_: ndarray,
                          delta: float = 0.001) -> ndarray:
'''
Вычисление производной функции "func" в каждом элементе массива
"input_".
'''
return (func(input_ + delta) – func(input_ – delta)) / (2 * delta)
```

Рассмотрим выражение «*P* — это функция *E*» (я намеренно использую тут случайные символы) означает, что некая функция *f* берет объекты *E* и превращает в объекты *P*, как показано на рисунке. Другими словами, *P* — это результат применения функции *f* к объектам *E*:



Соответствующий код:

```
def f(input_: ndarray) -> ndarray:
# Какое-то преобразование
return output
```

P = f(E)

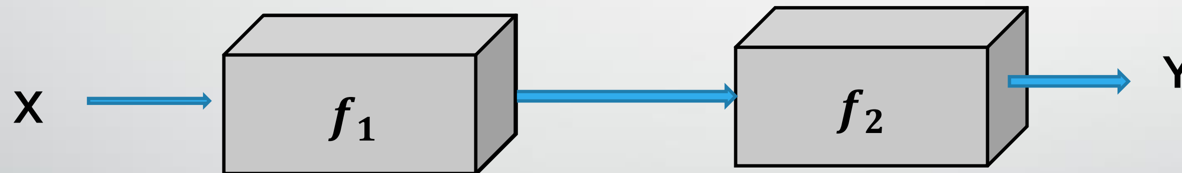
Вложенные функции

Вложенные, или составные, функции это концепция, которая важна для понимания нейронных сетей.

Очевидно, что две функции f_1 и f_2 можно связать друг с другом таким образом, что выходные данные одной функции станут входными для другой.

Визуализация

Наглядно представить концепцию вложенной функции можно следующим образом:



Данные подаются в первую функцию, преобразуются, выводятся и становятся входными данными для второй функции, которая и дает окончательный результат

Математическое представление

В математической нотации вложенная функция выглядит так:

$$f_2(f_1(x)) = y$$

Такое представление уже сложно назвать интуитивно понятным, потому что читать эту запись нужно не по порядку, а изнутри наружу. Хотя, казалось бы, это должно читаться как «функция f_2 функции f_1 переменной x», но на самом деле мы вычисляем f_1 от переменной x, а затем — f_2 от полученного результата.

Код

Чтобы представить вложенные функции в виде кода, для них первым делом нужно определить тип данных:

```
from typing import List

# Function принимает в качестве аргумента объекты ndarray и выводит
# объекты ndarray
Array_Function = Callable[[ndarray], ndarray]

# Chain — список функций
Chain = List[Array_Function]
```

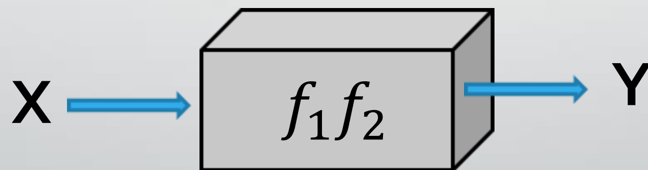
Теперь определим прохождение данных по цепочке из двух функций:

```
    def chain_length_2(chain: Chain, a: ndarray) -> ndarray:
    '''
    Вычисляет подряд значение двух функций в объекте "Chain“
'''
assert len(chain) == 2, \
"Длина объекта 'chain' должна быть равна 2“

f1 = chain[0]
f2 = chain[1]

return f2(f1(x))
```

Так как составная функция, по сути, представляет собой один объект, ее можно представить в виде $f_1 f_2$,



Функции нескольких переменных

Функции, с которыми приходится иметь дело в машинном обучении, часто имеют целый набор входных данных, которые в процессе обработки складываются, умножаются или комбинируются каким-то другим способом.

Вычислить производную такой функции тоже несложно. В качестве примера рассмотрим простой сценарий: функция двух переменных, которая вычисляет их сумму и затем передает в другую функцию.

Математическое представление

Пусть входные данные представляют переменные x и y. Действие функции можно разбить на два этапа. Сначала функция, которую мы обозначим греческой буквой α, выполняет сложение входных данных.

Греческие буквы и дальше будут использоваться в качестве имен функций.

Результат действия функции обозначим a. С формальной точки зрения все очень просто:

$$a = \alpha(x,y) = x + y$$

Затем передадим *a* в некую функцию σ (это может быть любая непрерывная функция, например квадратичная функция или другая функция по вашему выбору). Результат ее работы обозначим переменной *s*:

$$s = \sigma(a)$$

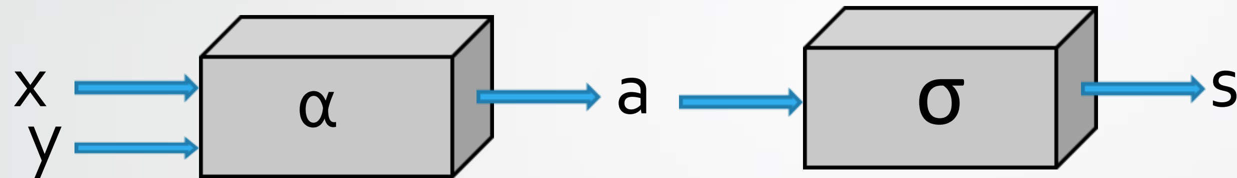
При этом ничто не мешает обозначить всю функцию *f* и написать:

$$f(x,y) = \sigma(x + y)$$

С математической точки зрения это более точная запись, но она не дает представления о том, что на самом деле мы последовательно выполняем две операции. Лучше всего это будет видно на следующим рисунке:

Визуализация

Функции нескольких переменных, окончательно определили то, что наши схемы со стрелками, указывающими на порядок выполнения операций, представляют собой *вычислительные графы* (computational graphs). Нашу функцию f можно представить



Мы видим, что в функцию α подаются два элемента входных данных, а результат ее работы *a* передается в функцию σ.

Код

Код вычислений в данном случае выглядит очень просто, но обратите внимание на дополнительный проверочный оператор:

```
def multiple_inputs_add(x: ndarray,
y: ndarray,
sigma: Array_Function) -> float:

'''

Функция сложения двух переменных, прямой
проход.
'''

assert x.shape == y.shape

a = x + y
return sigma(a)
```

В предыдущих случаях наши функции по отдельности обрабатывали каждый элемент объекта **ndarray**, то теперь мы сначала проверяем форму этих объектов, чтобы удостовериться, что с ними можно проводить указанную в коде операцию. Такие проверки всегда выполняются для функций нескольких переменных

Функции нескольких переменных с векторными аргументами

В глубоком машинном обучении используются функции, на вход которых подаются ***векторы*** или ***матрицы***. Эти объекты можно складывать, перемножать, а для векторов существует еще и такая операция, как скалярное произведение.

Основная цель глубокого обучения — **подбор модели**, наилучшим образом описывающей данные.

Фактически мы ищем функцию, которая точнее всего сможет сопоставить результаты наблюдений (служащие ее входными данными) с определенным шаблоном (который играет роль выходных данных). Входные данные удобно представлять в виде матриц, строки которых содержат результаты наблюдений, а столбцы —соответствующие количественные характеристики.

Необходимо научиться вычислять производные сложных функций, в которых происходит скалярное умножение векторов или умножение матриц.

Математическое представление

При работе с нейронными сетями единицы информации или результаты наблюдений обычно представляют в виде набора признаков. Каждый признак обозначается как x_1, x_2 и т. д., до последнего признака x_n:

$$\mathrm{X} = [\, x_1, x_2, \dots \dots, x_n \,]$$

Например, построим нейронную сеть, которая будет прогнозировать цену на жилье; в этом примере x_1, x_2 и далее — это числовые характеристики, такие как площадь дома или его расстояние до ближайшей школы.

Создание новых признаков из уже существующих

Возможно, самая распространенная операция в нейронных сетях — определение **взвешенной суммы** признаков.

Именно этот параметр усиливает определенные признаки и снимает акцент с других. Его можно рассматривать как новую функцию, полученную комбинированием существующих. Математически это скалярное произведение вектора характеристик и имеющего такой же размер вектора весов: w_1, w_2 и дальше до w_n.

Математическое представление

Если определить вектор весов отдельных признаков как

$$W = \begin{bmatrix} w_1 \\ \cdot \\ \cdot \\ w_n \end{bmatrix}$$

Скалярное произведение двух векторов будет записано как:

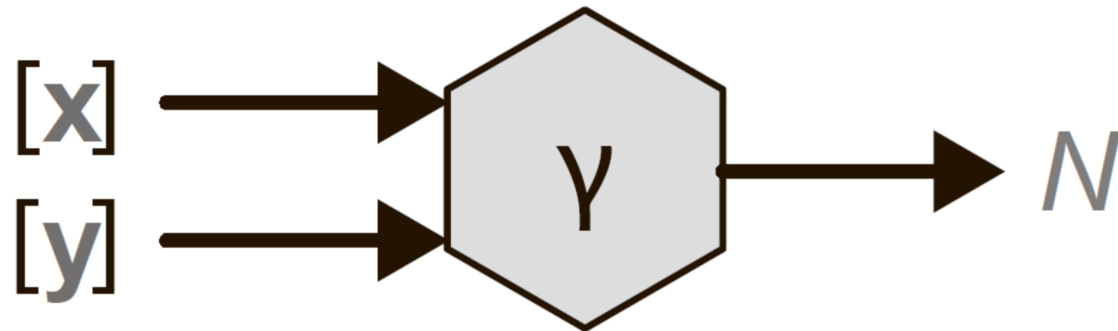
$N = v(X,W) = X*W = x_1 * w_1 + x_2 * w_2 + \cdots + x_n * w_n$

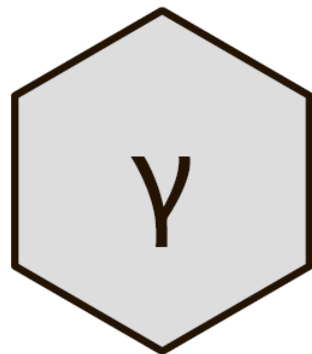
Фактически мы умножает вектор-строку X на вектор-столбец W

Визуализация

Простой способ изображения скалярного произведения следующий:

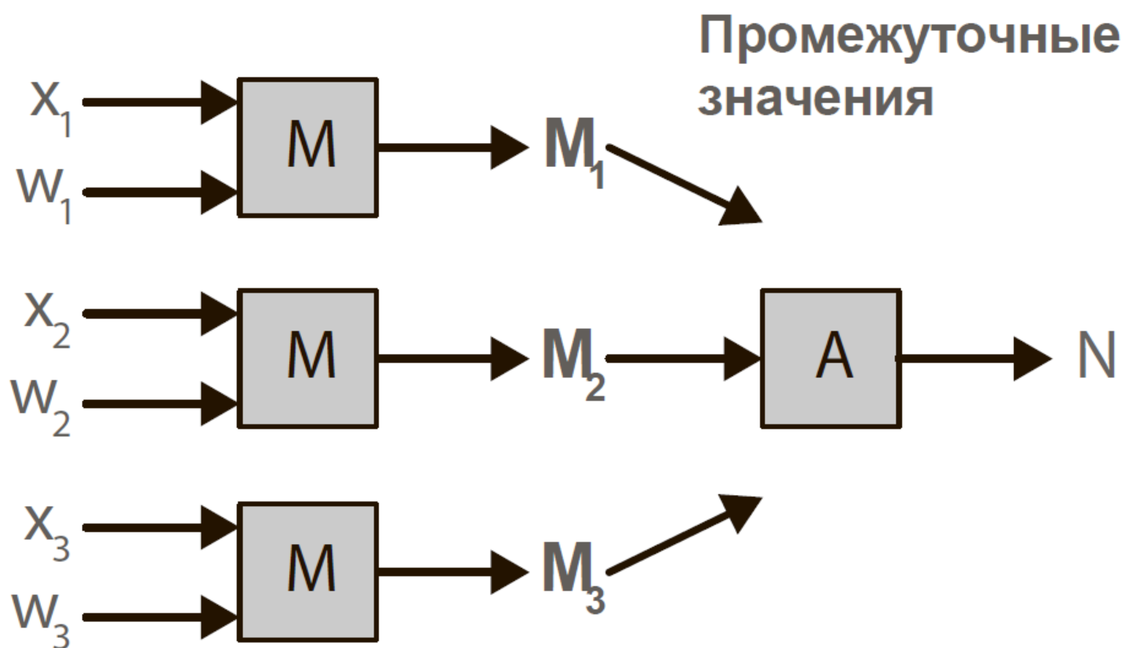
Входные данные *Выходные*



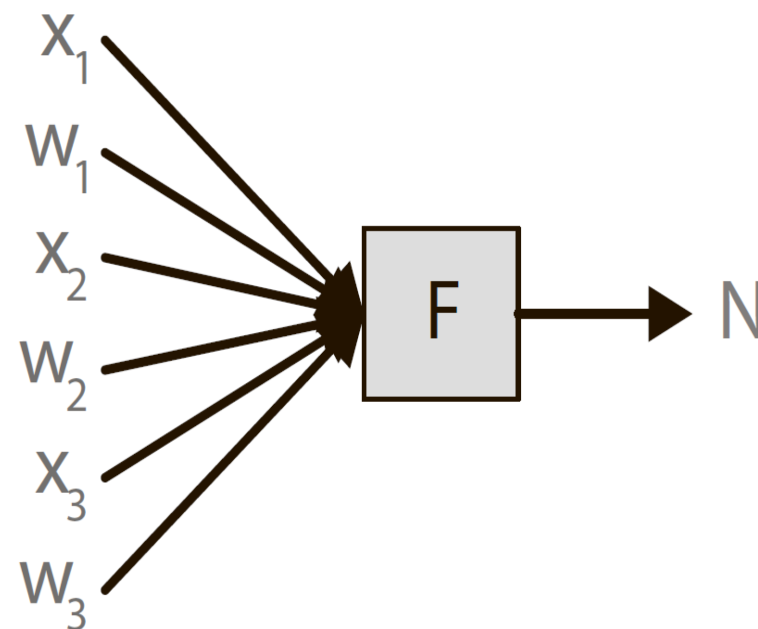


= Умножение матриц

Мы видим, что функция принимает два аргумента, роль которых могут играть объекты **ndarray**, и превращает их в один объект **ndarray**. Но работу функций нескольких переменных можно представить и другими способами. Например, выделив отдельные операции и входные данные, как показано



Второй способ представления операции умножения матриц



Третий способ представления операции умножения матриц

Код

Несложный код, реализующий операцию умножения:

```
def matmul_forward(X: ndarray,
W: ndarray) -> ndarray:
'''
Прямой проход при умножении матриц.
'''
assert X.shape[1] == W.shape[0], \
'''
Для операции умножения число столбцов первого массива должно
совпадать с числом строк второго; у нас же число столбцов
первого массива равно {0}, а число строк второго равно {1}.
'‘’
.format(X.shape[1], W.shape[0])

# умножение матриц
N = np.dot(X, W)

return N
```

Этот код содержит оператор проверки, гарантирующий допустимость операции умножения. Раньше такая проверка не требовалась, так как мы имели дело с объектами **ndarray** одинакового размера, операции над которыми выполнялись поэлементно.

Благодарю за внимание!