МИНОБРНАУКИ РОССИИ

–––––––—————––––––––––––—————––––––––––––

Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И.Ульянова (Ленина)

——————————————————––––——

А. А. ЛИСС Т. Р. ЖАНГИРОВ Н. А. ЖУКОВА А. Н. СУББОТИН

# ИСКУССТВЕННЫЕ НЕЙРОННЫЕ СЕТИ. ПРАКТИЧЕСКИЕ ЗАДАЧИ

Учебно-методическое пособие

Санкт-Петербург   
Издательство СПбГЭТУ «ЛЭТИ»

2022

УДК: 004.032.26 (07)

ББК: З 818я7

И86

**Лисс А. А., Жангиров Т. Р., Жукова Н. А., Субботин А. Н.**

И86 Искусственные нейронные сети. Практические задачи: учеб.-метод. пособие. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2022. 00 с.

ISBN000-0-0000-0000-0

Представлены описания практических задач по дисциплине «Искусственные нейронные сети» и задания для решения на их основе.

Предназначено для студентов специальности 000000, также может быть полезно инженерно-техническим работникам этой области знаний.

УДК: 004.032.26 (07)

ББК: З 818я7

Рецензент:

Утверждено

редакционно-издательским советом университета

в качестве учебно-методического пособия

ISBN 000-0-0000-0000-0 © СПбГЭТУ «ЛЭТИ», 2022

## ПРЕДИСЛОВИЕ

В настоящем издании представлены 8 практический занятий и 4 лабораторные работы по искусственным нейронным сетям. Задания расположены по сложности выполнения и являются взаимосвязанными. Для выполнения практических занятий необходимо обладать базовыми знаниями в области программирования, уметь работать с командной строкой, иметь представление об интерпретаторе Python. Рекомендуется последовательно выполнить все практические занятия и задания, которые представлены в отдельных разделах данного пособия. Выполнение практических занятий и заданий обеспечивает базовую подготовку обучающегося, необходимую для выполнения лабораторных работ, позволяет сформировать представление о библиотеках и фреймворках машинного обучения и искусственных нейронных сетей. Выполнение предложенных практических заданий позволяет сформировать общее понимание нейросетевого подхода и его применения при решении целого рядя задач, что может являться полезным не только обучающимся, но и практикующим специалистам.

## ПРАКТИЧЕСКИЕ ЗАНЯТИЯ

* 1. Библиотека NumPy

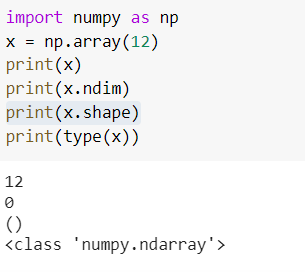
Современные системы машинного обучения работают с тензорами как основной структурой данных. В том числе и библиотека TensorflowKeras, что отражено в названии. Основная библиотека Pythonдля работы с тензорами это NumPy. По сути, тензор является контейнером для данных, и в основном числовых.

Любой тензор определяются тремя ключевыми атрибутами:

* Ранг (количество осей) – в NumPy за ранг отвечает параметр ndim.
* Форма – определяет количество измерения вдоль каждой оси тензора. В NumPy форма соответствует параметру shape.
* Тип данных – так как тензор является гомогенной структурой данных, то все значения принадлежат к одному типу данных. Например, это может быть float32, uint8, float64 и др. В области машинного обучения, обычно используются тензоры с числовыми типами данных. В NumPy форма хранится в параметре dtype.

Скаляры (тензор 0 ранга)

Тензор, хранящий одно единственное число, имеет ранг 0 (ndim = 0). Тензор представляющие скаляр практически не используются в чистом виде при работе с нейронными сетями. Пример подобного тензора:



import numpy as np

x = np.array(12)

print(x)

print(x.ndim)

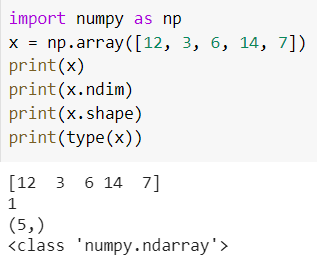
print(x.shape)

print(type(x))

Обратите внимание, что параметр shape содержит пустой кортеж, так как ранг равен 0 и в тензоре нет осей с измерениями.

Векторы (тензор 1 ранга)

Векторы представляют собой массив чисел и называются тензором ранга 1. В области машинного обучения, такими тензорам описывается один объект из предметной области. Пример создания вектора в NumPy:



import numpy as np

x = np.array([12, 3, 6, 14, 7])

print(x)

print(x.ndim)

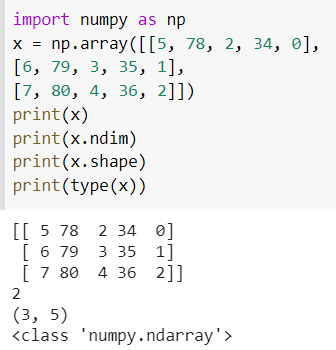
print(x.shape)

print(type(x))

В данном случае видно, что вектор имеет размерность (форму) равную (5, ), то есть в нем хранится 5 элементов.

Матрицы (тензоры 2 ранга)

Матрица – это вектор векторов, то есть это уже тензор ранга 2. Обычно оси матрицы называются строками и столбцами. Матрицы часто используются для описания монохромных изображений. Пример матрицы в NumPy:



import numpy as np

x = np.array([[5, 78, 2, 34, 0],

[6, 79, 3, 35, 1],

[7,80, 4, 36, 2]])

print(x)

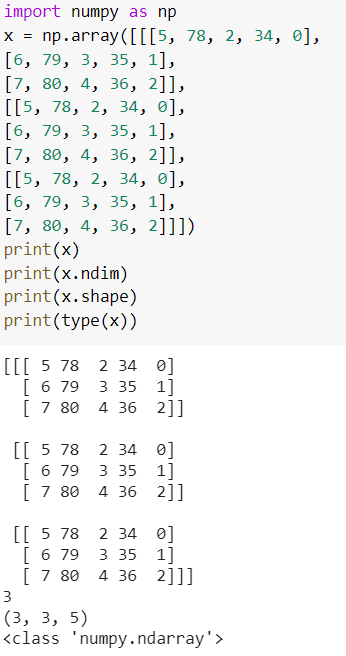
print(x.ndim)

print(x.shape)

print(type(x))

Тензоры третьего и высшего рангов

Если создать вектор, где каждый элемент вектора представлен в виде матрицы, то получится тензор ранга 3. Что может быть представлено в виде числового параллелепипед. Пример тензора ранга 3:



import numpy as np

x = np.array([[[5, 78, 2, 34, 0],

[6, 79, 3, 35, 1],

[7,80, 4, 36, 2]],

[[5, 78, 2, 34, 0],

[6, 79, 3, 35, 1],

[7,80, 4, 36, 2]]

[[5, 78, 2, 34, 0],

[6, 79, 3, 35, 1],

[7,80, 4, 36, 2]]])

print(x)

print(x.ndim)

print(x.shape)

print(type(x))

Помещая в вектор тензоры третьего ранга, можно получить тензор четвертого порядка. И если продолжать это процедуру, то можно получать тензоры высших рангов. Обычно в области машинного обучения ограничиваются тензорами ранга не более 4, в редких случаях применяются тензоры ранга 5.

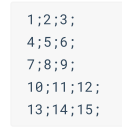
Основные тензоры с данными

При работе с системами машинного обучения, данные обрабатывается не по одному образцу, а сразу в виде пакета (несколько образцов, которые являются частью набора данных или сразу все набором данных). Данные, которые обычно необходимо обрабатывать при работе с нейронными сетями, обычно относятся к одной из следующих категорий:

* Векторные данные – набор таких данных представлен тензором второго порядка с формой (образцы, признаки). Самая часто встречаемая форма данных. Каждый образец представлен вектором признаков, и соответственно пакет изображается в виде матрицы. Принято считать, что первая ось (строки) отвечает за образцы, а вторая ось (столбцы) за признаки.
* Последовательности – обычно в качестве последовательности чаще всего рассматриваются временные ряды, то есть каждый образец описывает изменения признаков во времени. Такие данные используются, когда время или понятие упорядоченного ряда важны для решения поставленной задачи. В таком случае пакет данных рекомендуется использовать тензоры третьего ранга с формой (образцы, последовательность, признаками). По соглашению вторая ось всегда отвечает за хранение последовательности.
* Изображения – четырехмерные тензоры с формой (образцы, высота, ширина, цвет) или (образцы, цвет, высота, ширина). Изображение по отдельности имеет три оси, высота, ширина и цвет. Даже для черно-белых/монохромных изображений отдельности выделяется ось цвета с размерность 1, то есть отводится один канал под цвет.
* Видеопотоки – можно рассматривать как набор последовательностей изображений. То есть пакет видеопотоков хранится в тензоре с рангом 5 и формой (образцы, кадры, высота, ширина, цвет) или (образцы, кадры, цвет, высота, ширина).

Считывание тензоров из файла

NumPy представляет функционал для считывания данных из файла. Предположим, есть файл с содержимым:



Код для считывания данных из файла:



import numpy as np

x = np.fromfile(‘data1.csv’,dtype = ‘int’, sep = ‘;’)

print(x)

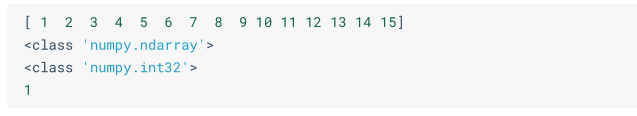
print(type(x))

print(type(x[0]))

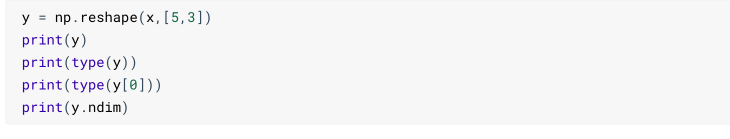
print(x.ndim)

В функции np.fromfile первый параметр отвечает за имя файла, dtype определяет тип данных элемента тензора, sep отвечает за разделитель между элементами в файле.

Если выполнить скрипт считывания, то будет следующий вывод:



Можно заметить, что данные считались в вектор (тензор 1 ранга). Для того, чтобы получить матрицу (тензор 2 ранга) такую как в файле, необходимо изменить форму тензора, для этого используется следующий скрипт:



y = np.reshape(x,[5,3])

print(y)

print(type(y))

print(type(y[0]))

print(type(y.ndim)

Функция np.reshape принимает тензор, который необходимо изменить и список размерностей нового тензора. После выполнения скрипта будет получен следующий вывод:



Изменять размерность каждый раз после считывания не всегда удобно, особенно если заранее не известны размеры тензора. Поэтому, есть более удобная функция для считывания из файла np.genfromtxt:



z = np.genfromtxt(‘data1.csv’,dtype = ‘int’ , delimiter=’;’)

print(z)

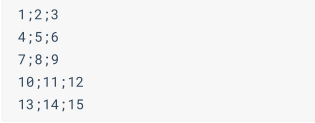
print(type(z))

print(z.ndim)

После выполнения скрипта будет следующий вывод:



Можно заменить, что считывание произошло сразу в двумерный тензор, но появилась лишняя колонка. Это связано с тем, что в конце каждой строки стоит символ ‘;’. Поэтому, необходимо изменить содержимое файла следующим образом:



Данный формат точно соответствует формату csv.

Теперь выполним следующий код:



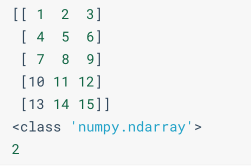
w = np.genfromtxt(‘data.csv’, dtype = ‘int’ , delimiter=’;’)

print(w)

print(type(w))

print(w.ndim)

Получим следующий вывод:



Как видно, данные сразу приобретают необходимую форму. К сожалению, у такого представления данных в файле есть ограничение, что таким образом можно представить данные в виде тензора не больше 2 ранга. В общем случае первая ось (ось с индексом 0, потому что нумерация начинается с 0) во всех тензорах, с которыми вам придется столкнуться в глубоком обучении, будет осью образцов (иногда ее называют измерением образцов).

Функции для работы с библиотекой numpy

arange

Функция аналогичная функции range в языке Python, но возвращает сразу массив numpy.ndarray



import numpy as np

a=np.arange(10)

print(type(A)) #<c;ass ‘numpy.ndarray’>

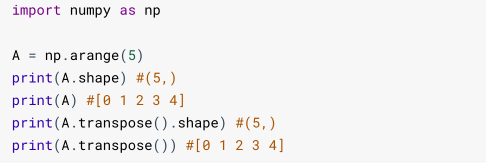
print(a.shape) #(10, )

print(a) #[0 1 2 3 4 5 6 7 8 9]

transpose

Фун кция транспонирования тензора

Для одномерного случая:



import numpy as np

a=np.arange(5)

print(A.shape) #(5, )

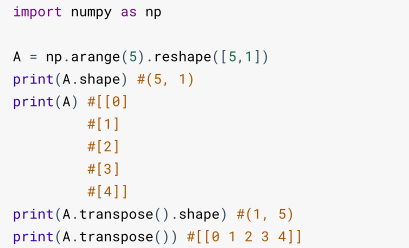
print(A) #[0 1 2 3 4]

print(A.transpose().shape) #(5, )

print(A.transpose()) #[0 1 2 3 4]

В данном случае, так как размерность одна, то транспонирования не происходит.

Если вектор представлен как матрица 5 на 1:



import numpy as np

a=np.arange(5).reshape([5,1])

print(A.shape) #[[0]

#[1]

#[2]

#[3]

#[4]]

print(A.transpose().shape) #(1, 5 )

print(A.transpose()) #[[0 1 2 3 4]]

То есть для тензоров с размерностью 2 транспонирование происходит по правилам транспонирования матриц.

Для тензоров большей размерности:



import numpy as np

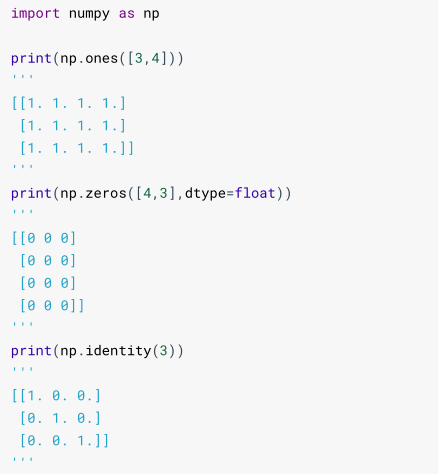
a=np.arange(120).reshape([2,3,4,5])

print(A.shape) #(2, 3, 4, 5)

print(A.transpose().shape) #(5, 4, 3, 2)

ones, zeros, identity

Функции позволяют создать матрицу, заполненную единицами, нулями, а также единичную соотвественно



import numpy as np

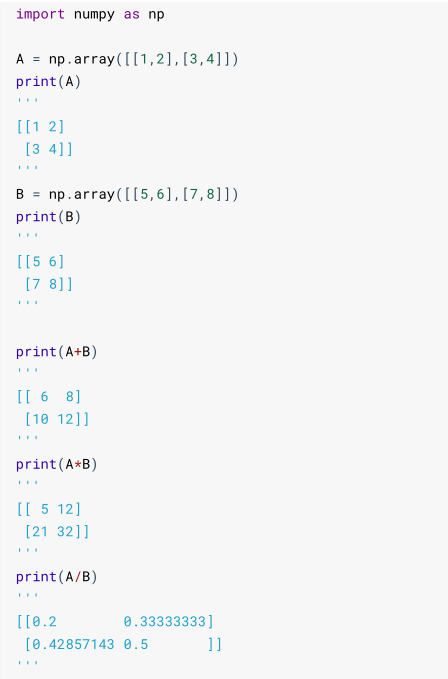
print(np.ones([3,4]))

print(np.zeros([4,3], dtype=float))

print(np.identity(3))

Математические операции

Для тензоров математические операции работают поэлементно



import numpy as np

a = np.array([[1,2],[3,4]])

print(A)

p = np.array([[5,6],[7,8]])

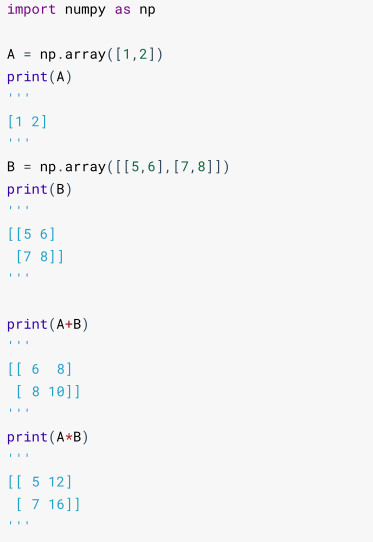
print(B)

print(A+B)

print(A\*B)

print(A/B)

Если размерности тензоров не совпадают, то меньший тензор будет преобразован для использования. Зачастую, он будет повторен несколько раз.



import numpy as np

a = np.array([1,2])

print(A)

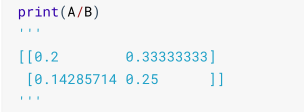
b = np.array([[5,6],[7,8]])

print(B)

print(A+B)

print(A\*B)

print(A/B)

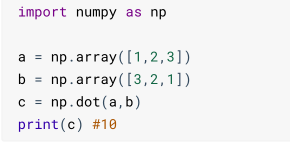


Отсюда видно, что можно выполнять математические операции тензора со скаляром.

Также, в numpy есть стандартные математические функции, такие как log, asb, sin, exp, и.т.д., которые можно применять поэлементно к тензорам.

dot и tensordot

NumPy обеспечивает много функций для работы с векторами и матрицами. Функция dot возвращает скалярное произведение векторов.



import numpy as np

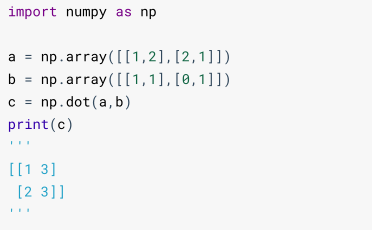
a = np.array([1,2,3])

b = np.array([3,2,1])

c = np.dot(a,b)

print(c) #10

Также, функцией dot можно перемножать матрицы



import numpy as np

a = np.array([[1,2],[2,1]])

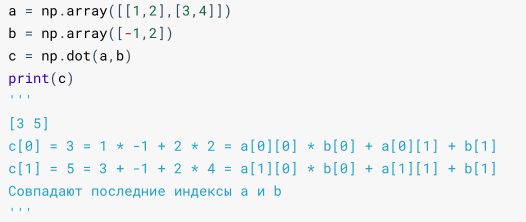
b = np.array([[1,1],[0,1]])

c = np.dot(a,b)

print(c)

Для функции dot выполняются следующие условия:

* Если a и b одномерные тензоры - выполняется скалярное произведение векторов
* Если a и b двумерные тензоры - выполняется матричное произведение, но рекомендуется использоваться функцию matmul или оператор a @ b
* Если a и b скаляры - выполняется обычное умножение, что равносильно функции multiply и оператору a \* b
* Если a N-мерный тензор и b 1-мерный тензор (вектор) - получается сумма произведений по последний оси a и b



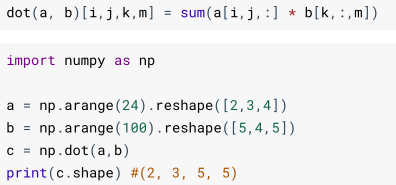
a = np.array([[1,2],[3,4]])

b = np.array([-1,2])

c = np.dot(a,b)

Print(c)

* Если a N-мерный тензор и b M-мерный тензор (M >= 2) - получается сумма произведений по последней оси a и предпоследней оси b



dot(a,b)[i,j,k,m] = sum(a[i,j,:] \* b [k,:,m])

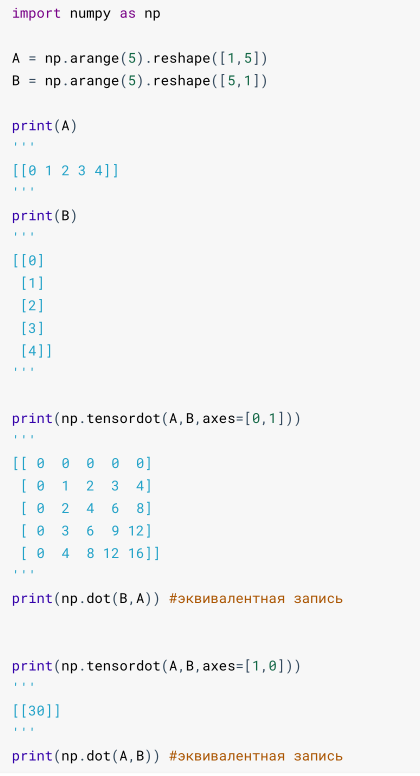
a = np.arange(24).reshape([2,3,4])

b = np.arange(100).reshape([5,4,5])

c = np.dot(a,b)

print(c.shape) #(2, 3, 5, 5)

Функция tensordot позволяет вручную задать по каким осям производить сумму произведений.



import numpy as np

a = np.arange(5).reshape([1,5])

b = np.arange(5).reshape([5,1])

print(A)

print(B)

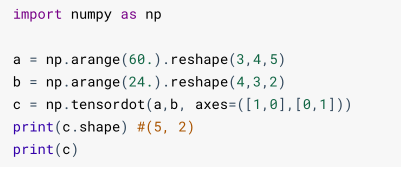
print(np.tensordot(A,B,axes=[0,1]))

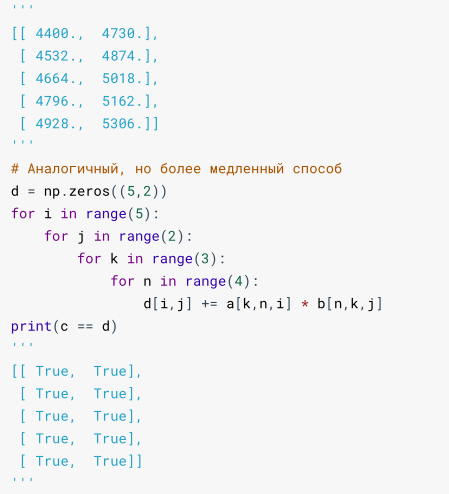
print(np.dot(B,A)) #эквивалентная запись

print(np.tensordot(A,B,axes=[1,0]))

print(np.dot(A,B)) #эквивалентная запись

Также можно задать сумму сразу по нескольким осям





import numpy as np

a = np.arange(60.).reshape(3,4,5)

b = np.arange(24.).reshape(4,3,2)

c = np.tensordot(a,b, axes=([1,0],[0,1]))

print(c.shape) #(5,2)

print(c)

#Аналогичный, но более медленный способ

d = np.zeros((5,2))

for I in range(5):

for j in range(2):

for k in range(3):

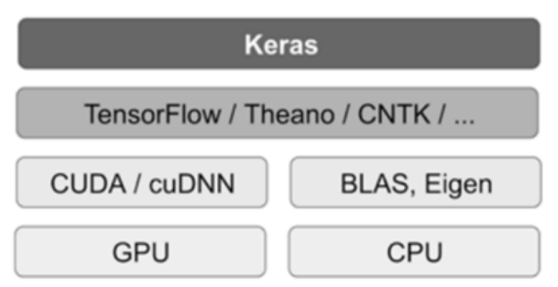
for n in range(4):

d[i,j] += a[k,n,i] \*b[n,k,j]

print(c == d)

* 1. Библиотека Keras

Keras – библиотека для работы с моделями глубоко обучения, обеспечивающая высокоуровневый APIдля создания и обучения искусственных нейронных сетей. Библиотека Kerasявляется надстройкой над другими более низкоуровневыми библиотеками такими как Tensorflow и Theano.



Таким образом, APIKeras’а предоставляет только высокоуровневый функционал для построения и обучения нейронных сетей, таким образом упрощая работу с моделями глубокого обучения. А саму внутреннюю реализацию нейронных сетей, процесс их обучения и так далее отводит низкоуровневым библиотекам, которые поддерживают работу с тензорами. Причем, код написанный с использование Keras можно переносить на любую низкоуровневую библиотеку.

За счет этого библиотека Kerasимеет следующие особенности:

* Один и тот же код можно выполнять на CPU и на GPU
* Встроенная поддержка сверточных и рекуррентных сетей, а также их комбинаций
* Поддержка любых сетевых архитектур

Далее рассматриваться будет Keras с библиотекой TensorFlow 2.0, так как на данный момент TensorFlow 2.0. включает в себя KerasAPI.

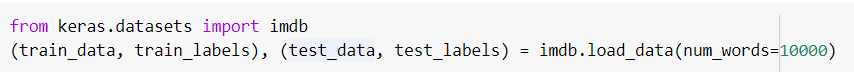
Полносвязные нейронные сети

Бинарная классификация

Знакомство с библиотекой Keras начнется с решения задачи бинарной классификации, так как это одна из самых простых задач с точки зрения построения и обучения модели.

Бинарная классификация – это классификация объектов по двум классам. В данном примере рассмотрим решение задачи на классификации отзывов к фильмам с IMDBна положительные и отрицательные. Этот датасет, как и множество других, поставляется вместе с библиотекой Keras, и содержит 50 000 отзывов. Разбиение на обучающую и тестовую выборку произведено в отношении 1:1.

Первым делом подключим датасет и проинициализируем переменные *train\_data* и *test\_data*, которые будут содержать сами отзывы, и переменные *train\_labels* и *test\_labels*, содержащие метки о том, положительный или отрицательный отзыв.

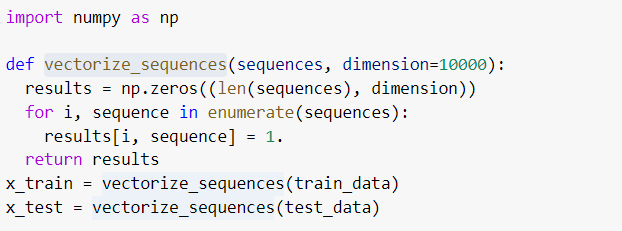


from keras.datasets import imdb

(train\_data, train\_labels), (test\_data, test\_labels) = imdb.load\_data(num\_words=10000)

Переменные *train\_labels*и*test\_labels*являются списками из 0 и 1, где 0 обозначает отрицательный отзыв, а 1 положительный отзыв. Переменные *train\_data* и *test\_data* содержат список, состоящий из другого списка хранящий индексы слов, которые содержатся в обзоре. Так как нейронная сеть обычно работает с вещественными числами, а также будет показывать лучший результат, если все подаваемые значения будут принадлежать одному диапазону значений, то необходимо провести векторизацию данных. Это означает, что необходимо изменить представление данных. После преобразования вектор [10 243] будет преобразован к вектору длины 10 000 заполненный нулями кроме элементов с индексами 10 и 243, которые содержат единицы. Число 10 000 выбрано по той причине, что при загрузке датасета были выбраны 10 000 самых встречающихся слов среди всех обзоров. После такого преобразования будет учитываться только то, какие слова содержаться в обзоре, но не их порядок.

Для векторизации будет использоваться функция *vectorize\_sequences*



import numpy as np

def vectorize\_sequences(sequences, dimension=10000):

results – np.zeros((len(sequences), dimension))

for i, sequence in enumerate(sequences):

results[i,sequence] = 1

return results

x\_train = vectorize\_sequences(train\_data)

x\_test = vectorize\_sequences(test\_data)

Также, приведем метки классов к вещественному типу



y\_train = np.asarray(train\_labels).astype(‘float32)

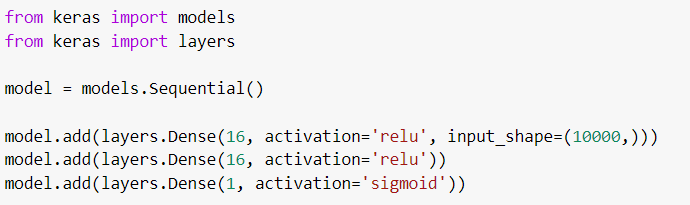
y\_test = np.asarray(test\_labels).astype(‘float32’)

При построении любой модели размер входного и выходного слоя зависят от размерности входных и выходных данных. В этой задаче входными данными являются вектора длины 10 000, соответственно на входном слое будет количество нейронов равное размеру вектора. Выходными данными являются скаляры, поэтому выходной слой имеет 1 нейрон.

Остальные параметры архитектуры поддаются гибкой настройки. Для решения задачи, будет использовать нейронную сеть с двумя скрытыми слоями и 16 нейронами на каждом. Функцию активации выберем relu, которая активно применяется в подобного рода задачи. Формула reluимеет следующий вид , то есть она обнуляет отрицательные значения и не меняет положительные.

На выходном слое будет использоваться функция сигмоида, которая принимает значения от 0 до 1. Использование этой функции позволяет интерпретировать результат не просто как жесткое определение класса, а как вероятность того, что обзор является положительным.

Построение модели в библиотеке Kerasвыглядит следующим образом:



from keras import models

from keras import layers

model = models.sequential()

model.add(layers.Dense(16, activation=’relu’, input\_shape(10000,)))

model.add(layers.Dense(16, activation=’relu’))

model.add(layers.Dense(1, activation=’sigmoid’))

Здесь *Sequential* обозначает то, что в нейронной сети сигнал распространяется последовательно от слоя к слою начиная со входного и заканчивая выходным. Функция *add* добавляет очередной слой. *Dense* обозначает полносвязный слой, то есть сигналы с предшествующего слоя распространяются на каждый нейрон слоя. Основными параметрами данного слоя являются количество нейронов (первый параметр) и используемая функция активации (параметр *activation*). Также для самого первого слоя указывается значение параметра *input\_shape*, отвечающее за размер входных данных. Для остальных слоев это значение рассчитывается автоматически.

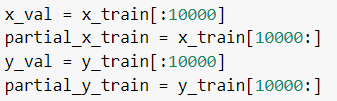
После создания модели необходимо указать оптимизатор (каким методом будет проводиться обучение нейронной сети), функцию потерь (она используется для оценки качества модели и коррекции весов), а также дополнительные метрики, которые не влияют на коррекцию весов. Чтобы задать параметры, используется функция *compile*.



model.compile(optimizer=’rmsprop’, loss=’binary\_crossentropy’, metrics=[‘accuracy’])

Для решения задачи был выбран оптимизатор *RMSProp*, функция потерь бинарная кросс-энтропия, а в качестве дополнительной метрики точность. Бинарная кросс-энтропия классическая функция потерь в задачах бинарной классификации, если результат может быть представлен в виде вероятности. Точность отслеживается по той причине, что значение ошибки анализировать в чистом виде крайне затруднительно.

В процессе обучения необходимо контролировать реакцию модели на данные, которые не участвуют в обучении, чтобы избежать эффекта переобучения. Для этого, из 25 000 объектов обучающей выборки отделим 10 000 в отдельную контрольную выборку. Тестовая выборка, которая была получена при загрузке датасета, будет использована уже после обучения модели для валидации результата.



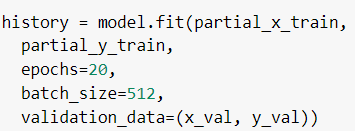
x\_val = x\_train[:10000]

partial\_x\_train = x\_train[10000:]

y\_val = y\_train[:10000]

partial\_y\_train = y\_train[10000:]

После того, как датасет разбит на обучающую, контрольную и тестовую выборку, и создана модель, то можно обучать нейронную сеть. Для обучения используется функция *fit*, которая в качестве первых двух параметров принимает набор входных и выходных данных для обучения. Обучение будем проводить в течении 20 эпох, за это отвечает параметр *epochs* функции *fit*. Контрольные данные подаются в виде кортежа через параметр *validation\_data*. Также важным параметром является *batch\_size*, который задает размер пакета обучения. В этой задаче разобьём обучающие данные на пакеты размером 512.



history = model.fit(partial\_x\_train,

partial\_y\_train,

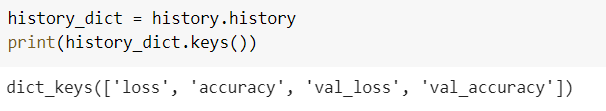
epochs=20,

batch\_size=512,

validation\_data=(x\_val, y\_val))

Можно заменить, что функция *fit* имеет возвращаемое значение. Это объект *History*, и в нем в поле *history* хранится словарь с изменениями каждой метрики в процессе обучения для каждой выборки (обучающей и контрольной).

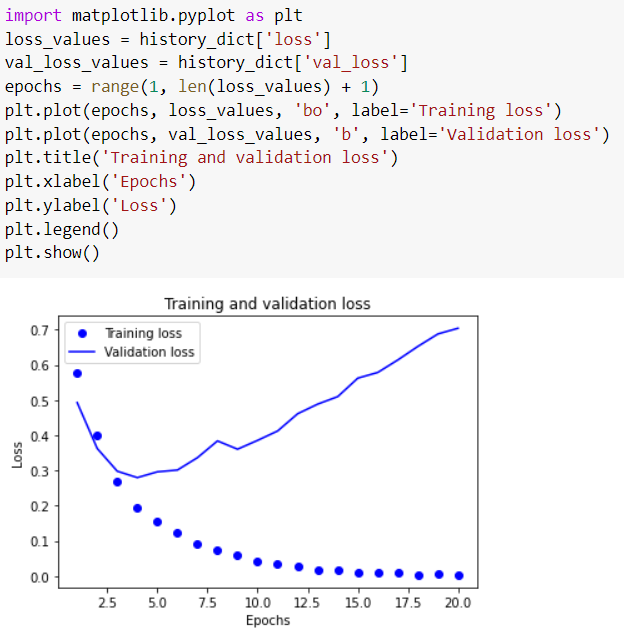
Чтобы узнать, какие метрики хранятся в словаре, достаточно просто вывести список всех ключей словаря:



history\_dict = history.history

print(history\_dict.keys())

Выведем графики ошибок (функции потерь) для обучающей и контрольной выборки. Для этого воспользуется библиотекой matplotlib:



import matplotlib.pyplot as plt

loss\_values = history\_dict[‘loss’]

val\_loss\_values = history\_dict[‘val\_loss’]

epochs = range(1, len(loss\_values) +1)

plt.plot(epochs, loss\_values, ‘bo’, label=’Training loss’)

plt.plot(epochs, val\_loss\_values, ‘b’, label=’Validation loss’)

plt.title(‘Training and vavlidation loss’)

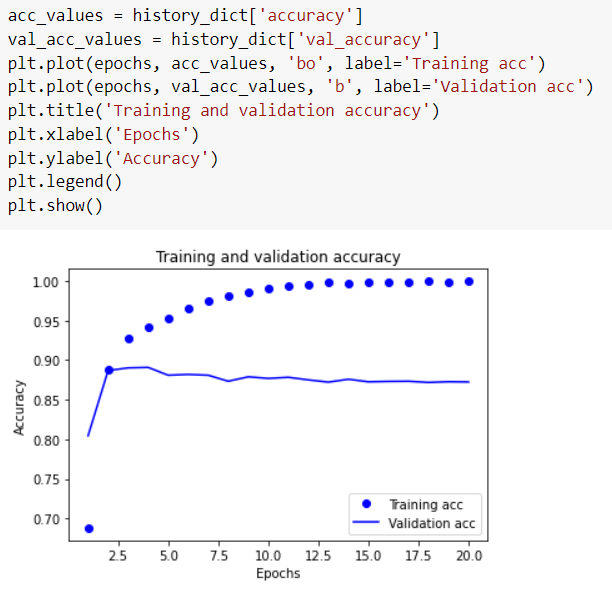
plt.xlabel(‘Epochs’)

plt.ylabel(‘Loss’)

plt.legend()

plt.show()

Аналогично выведем изменение точности в процессе обучения:



acc\_values = history\_dict[‘accuracy’]

val\_acc\_values = history\_dict[‘val\_accuracy’]

plt.plot(epochs, acc\_values, ‘bo’, label=’Training acc’)

plt.plot(epochs, val\_acc\_values, ‘b’, label=’Validation acc’)

plt.title(‘Training and vavlidation accuracy’)

plt.xlabel(‘Epochs’)

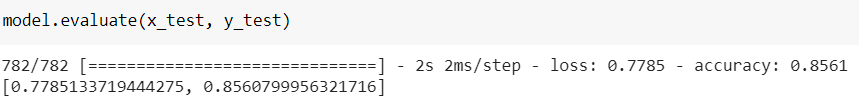
plt.ylabel(‘Accuracy’)

plt.legend()

plt.show()

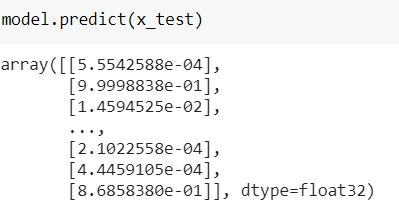
Из графиков видно, что значение ошибки постепенно уменьшалось, что и ожидалось от оптимизации градиентным спуском. Но для контрольной выборки максимум значения функции потерь был достигнут на 4 эпохе. Это может говорить о том, что начиная с этой эпохи началось переобучение.

Для оценки модели на тестовых данных используется функция *evaluate*:

model.evaluate(x\_test, y\_test)

Как видно, значение ошибки равно 0.778 и точность классификации имеет значение 0.856 (85,6%). Данные значения близки к значениям на контрольной выборке в конце обучения.

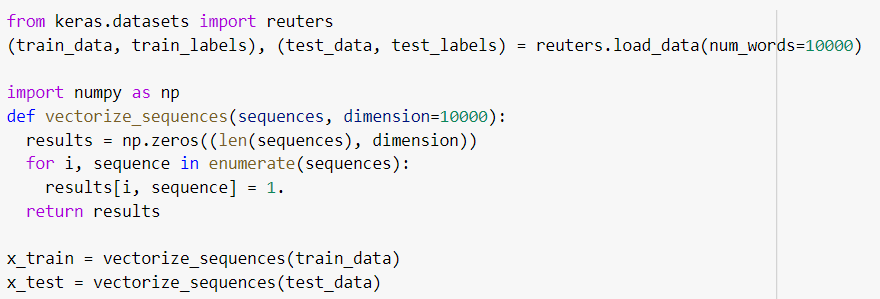
После того как сеть обучена, ее можно применять на практике, подавая любые данные, которые соответствуют формату входных данных. Для получения выходного значения используется функция *predict*:



model.predict(x\_test)

Классификация нескольких классов

Отличие данной задачи от бинарной классификации заключается в том, что количество классов, к которым может быть отнесен объект больше 2. Решение этой задачи будет рассмотрено на датасете Reuters. Этот датасет содержит набор заметок публиковавшихся агенством Reuters. Эти заметки имеют 46 разных тем, что соответствует 46 классам. Датасет Reuters как и датасет IMDB представлен в виде текста, поэтому загрузка и предобработка данных ничем не отличается:



from keras.datasets import reuters

(train\_data, train\_labels), (test\_data, test\_labels) = reuters.load\_data(nim\_words=10000)

import numpy as np

def vectorize\_sequences(sequences, dimension=10000):

result = np.zeros((len(sequences), dimension))

for i, sequence in enumerate(sequences):

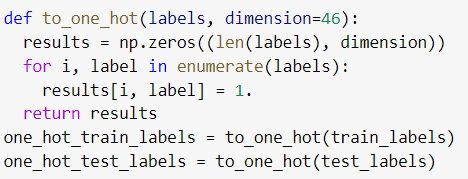
results[i, sequences] = 1.

return results

x\_train = vectorize\_sequences(train\_data)

x\_test = vectorize\_sequences(test\_data)

Так как классов больше чем 2, то метки классов хранят номер класса (целые числа от 0 до 45), и в данном случае уже недостаточно перевести скаляры к вещественным числам. Поэтому метки по аналогии с текстом надо векторизовать. После векторизации каждая метка будет представлена в виде вектора размер которого равен количеству классов. Этот вектор содержит нули кроме элемента с индексом, соответствующим номеру класса. Элемент с индексом класса имеет значение 0. Такое кодирование называется OneHotEncoding.



def to\_one\_hot(labels, dimension=46):

results = np.zeros((len(labels), dimension))

for i, label in enumerate(labels):

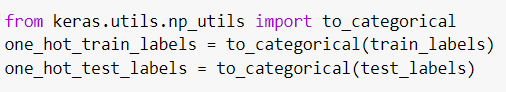
results[i, label] = 1.

Return results

one\_hot\_train\_labels = to\_one\_hot(train\_labels)

one\_hot\_test\_labels = to\_one\_hot(test\_labels)

Прописывать эту функцию самостоятельно не обязательно, так как данная функция уже содержится в библиотеке Keras.



from keras.utils.np\_utils import to\_categorical

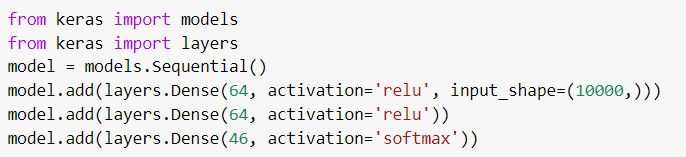
one\_hot\_train\_labels = to\_categorical(train\_labels)

one\_hot\_test\_labels = to\_categorical(test\_labels)

Построение архитектуры для данной задачи не сильно отличается от построения в бинарной классификации. Отличие заключается в том, что выходной слой должен быть адаптирован под 46 классов. Поэтому, выходной слой будет содержать 46 нейронов, и в ходе обучения модель будет пытаться преобразовать данные так, чтобы только на одном нейроне было значение 1, а на остальных ноль. Также, чтобы можно было интерпретировать значения как вероятности принадлежности объекта к классу, на выходном слое будет использоваться функция активации Softmax. Данная функция преобразует значения так, что каждое значение принадлежит диапазон от 0 до 1, и сумма всех выходных значений равно 1.

При решении задачи бинарной классификации на скрытых слоях было по 16 нейронов, то есть после входного слоя размер проходящего сигнала менялся с 10 000 до 16. В случае, когда было 2 класса нейронная сеть с каждым слоем могла выделять только важную для классификации информацию, но так как в этой задаче 46 классов, модели может оказаться тяжело провести классификацию из сигнала размером 16. Это проблема также называется «бутылочным горлышком». Чтобы избежать этого, будет использоваться модель с двумя скрытыми слоями и 64 нейронами на них.

Построение модели в итоге выглядит следующим образом:



from keras import models

from keras import layers

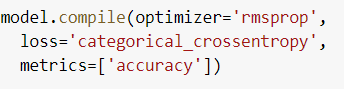
model = models.Sequential()

model. Add(layers. dense(64, activation =’relu’,inport\_shape=(10000,)))

model. Add(layers. dense(64, activation =’relu’))

model. Add(layers. dense(64, activation =’softmax’))

Так как изменилось представление результатов нейронной сети, то в качестве функции потерь будет использоваться категориальная кросс-энтропия. И аналогично предыдущей задаче, будет отслеживаться метрика точности.

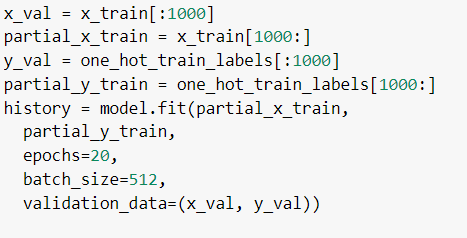


model.compile(optimizer=’rmsprop’,

loss=’categorical\_crossentropy’,

metrics=[‘accuray’]

Обучение проведем с теми же параметрами. Отличие будет лишь только в том, что в контрольную выборку выделим 1000 наблюдений.



x\_val=x\_train[:100]

partial\_x\_train=x\_train[1000:]

y\_val=one\_hot\_tarin\_labels[:1000]

partial\_y\_train=one\_hot\_train\_labels[1000:]

history=model.fit(partial\_x\_train,

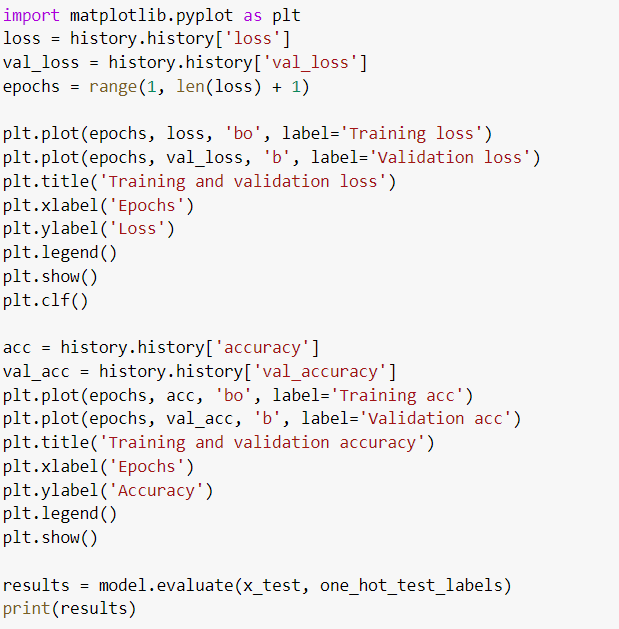
partial\_y\_train=x\_tarin,

epochs=20,

batch\_size=512,

validation\_data=(x\_val,y\_val))

Построение графиков и проверку модели на тестовых данных проводится по аналогии с тем, как это делалось в задаче бинарной классификации



import matplotib.pyplot as plt

loss = histoty.histoty[‘loss’]

val\_loss=historu.history[‘val\_loss’]

epochs=range(1, len(loss)+1)

plt.plot(epochs, loss, ‘bo’, label=’training loss’)

plt.plot (epochs, loss, ‘bo’, label=’validation loss’)

plt.title(‘training and validation loss’)

plt.slabel(‘epochs’)

plt.ylabel(‘loss’)

plt.legend()

plt.show()

plt.clf()

acc=history.history[‘accuracy’]

val\_acc=history.history[‘val\_accurscy’]

plt.plot(epochs, loss, ‘bo’, label=’training loss’)

plt.plot (epochs, loss, ‘bo’, label=’validation loss’)

plt.title(‘training and validation loss’)

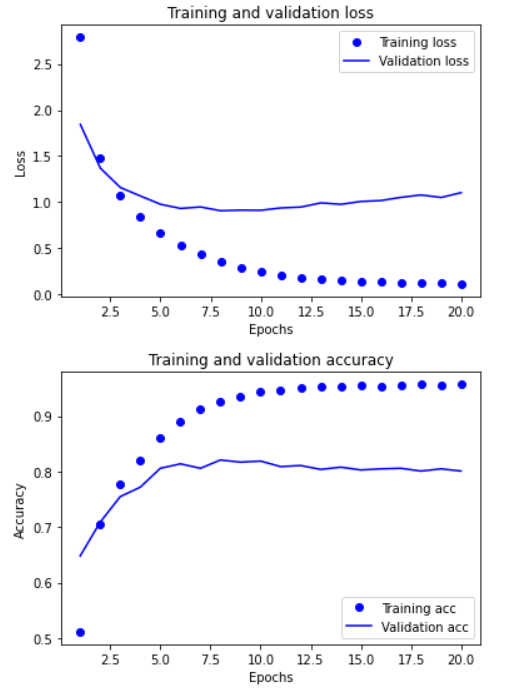
plt.slabel(‘epochs’)

plt.ylabel(‘accuracy’)

plt.legend()

plt.show()

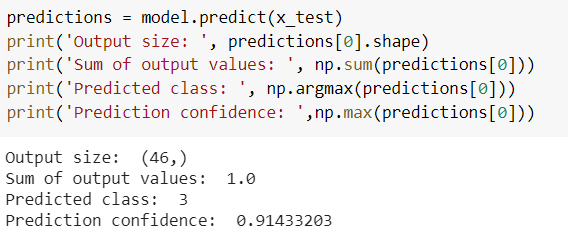
Полученные графики



По графикам видно, что возникло переобучение после 8 эпохи, но не такое сильное, как в случае бинарной классификации.

Проверка на тестовых данных показало ошибку равную 1.22 и точность 0.788 (78.8%).

Также для проверки результата, выведем размер выходного вектора, сумму значений выходного вектора, предсказанный класс и уверенность в предсказании для первого наблюдения из тестовой выборки:



predictions=model.predict(x\_test)

print(‘output size:’, predictions[0].shape)

print(‘sum of output values:’, np.sum(predictions[0]))

print(‘predicted class:’, np.argmax(predictions[0]))

print(‘prediction confidence:’, np.max(predictions[0]))

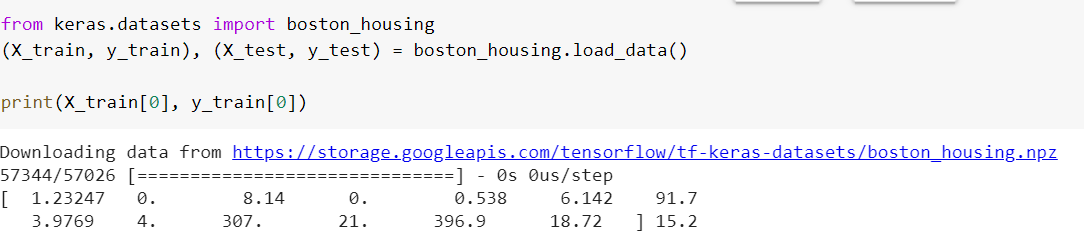
Как видно, размер совпадает с количеством класса, сумма значений равно 1, как и должно быть при использовании функции Softmax. Наблюдение было отнесено к классу с номером 3 с уверенностью (вероятностью) 0.91 (91%).

Регрессия

Основное отличие задачи регрессии от классификации заключается в формате данных, которые необходимо предсказать. Если в задачах классификации необходимо предсказать дискретную величину из конечного множества, то в регрессии предсказываемая величина является непрерывной и зачастую не ограничена.

Применение нейронных сетей для решения задачи регрессии будет рассматриваться на примере задачи предсказания стоимости домов в Бостоне на основе их характеристик.

Данный датасет содержится в библиотеке Keras.

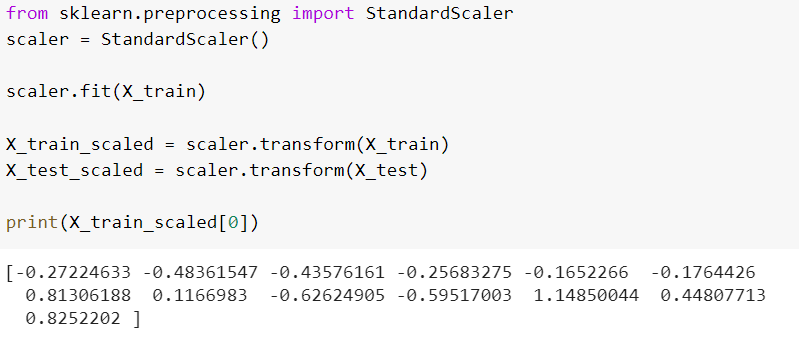


from keras.datasets import boston\_housing

(x\_train,y\_train), (x\_test, y\_test) + boston\_housing.load\_data()

print(x\_train[0], y\_train[0])

После загрузки датасета, если вывести значения, видно, что разные признаки сильно различаются. При рассмотрении задач классификации работа велась с текстовыми закодированными данными, и нормализация данных не требовалась. В данной задаче, для улучшения качества результата необходимо все данные нормировать. Для этого будет использовать стандартизатор из библиотеки SKLearn



from sklearn.preprocessing import StandardScaler

scaler = StandardSkaler()

scaler.fit(X\_train)

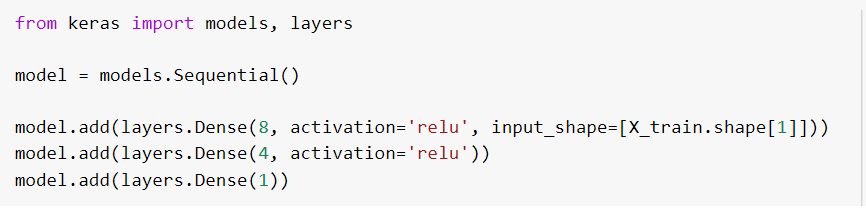
X\_train\_scaled = scaler.transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

print(X\_train\_scaled[0])

После стандартизации данных (приведение данных к виду, чтобы мат. ожидание было 0 и дисперсия 1) признаки имеют близкие значения.

Принцип построения самой модели аналогичен построению в задачах классификации.



from keras import models, layers

model = models.Sequential()

model.add(layers.Dense(8, activation=’relu’, input\_shape=[X\_train.shape[1]]))

model.add(layers.Dense(4, activation=’relu’))

model.add(layers.Dense(1))

Единственное различие заключается в том, что на выходном слое не указывается функция активации. В данном случае значения параметра *activation* равно None, что соответствует линейной функции активации.

Выбранная модель имеет 2 скрытых слоя с 8 и 4 нейронами на них и функцией активации RElu.

Так как предсказывается непрерывное значение, то использование кросс-энтропии уже не даст хорошего результата. Поэтому используется MSE (среднеквадратическое отклонение), которое оценивает, насколько близко предсказанное значение к реальному. В качестве дополнительное метрики указано MAE (среднее абсолютное отклонение). Так как стоит задача предсказать цену на дом, то квадрат ошибки анализировать не удобно, а MAEпокажет среднее отклонение в тех же единицах, что и предсказываемое значение.



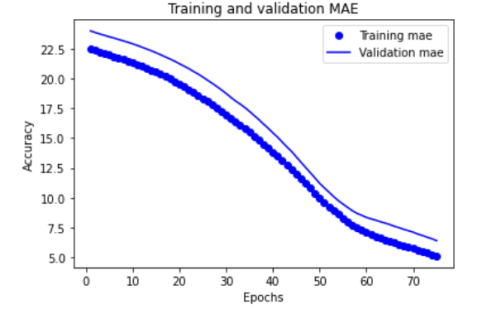
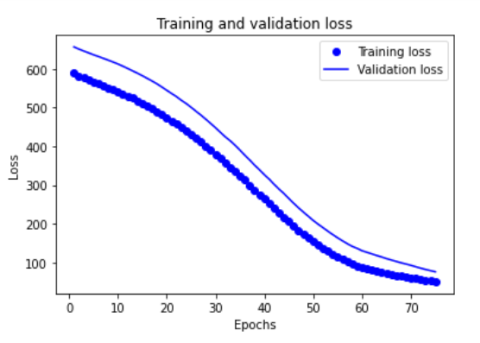
model.compile(optimizer=’rmsprop’, loss=’mse’, metrics=[‘mae’])

Как можно было заметить, при решении задачи не выделялся отдельно контрольный датасет. Создание контрольного датасета можно сделать, указав в функции *fit*значению параметру *validation\_split.* При таком способе, в качестве контрольной выборки будут выбран указанный процент наблюдений из начала обучающей выборки.

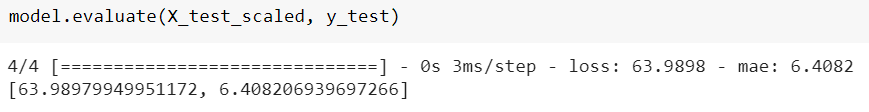


history = model.fit(X\_train\_scaled, y\_train, validation\_split=0.2, epochs=75)

После построения графиков ошибки и отслеживаемой метрики видно, что модель не переобучилась. Более того, видно, что значения ошибки не вышли на «плато» и сеть может быть недообучена. В данном случае процесс обучения можно было продлить.



Оценка на тестовых данных показывает, что значение ошибки равно 63.98 и значение МАЕ равно 6.40. Данное значение МАЕ по уровню практически совпадает со значениями на обучающей выборке. Значение 6.40 в данном случае можно интерпретировать как то, что при предсказании цены модель в среднем ошибается на 6 400$.

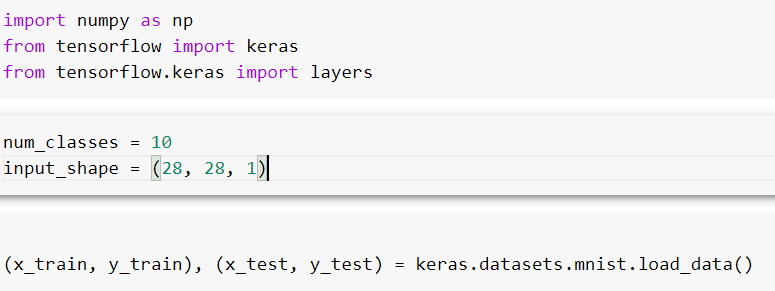


model.evaluate(X\_test\_scaled, y\_test)

Сверточные сети

Основное применение сверточных сетей это обработка изображений. При обработке изображений самые распространенные задачи — классификация, сегментация и иногда регрессия. Пример создания сверточной последовательной сети будет рассмотрен в рамках решения задачи классификации рукописных цифр из датасета MNIST. Этот датасет состоит из 70 000 монохромных изображений размером 28 на 28 пикселей.

Также как и в уже рассмотренных задачах, сначала необходимо загрузить датасет. Также сразу зададим параметр количества классов (10 различных цифр) и параметр размера входных данных (тензор размером 28х28х1).



import numpy as np

from tensorflow import keras

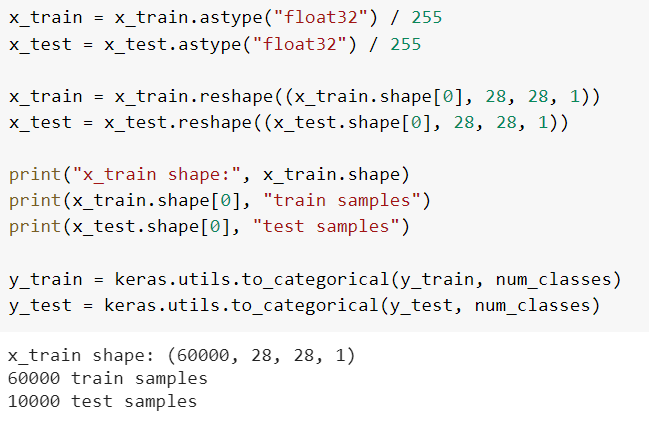
from tensorflow.keras import layers

num\_classes = 10

input\_shape = (28, 28, 1)

(x\_train, y\_train), (x\_test, y\_test) = keras.datasets.mnist.load\_data()

Так как в датасете каждый пиксель кодируется целым числом от 0 до 255, то необходимо их привести к вещественному числу, а также нормировать значения пикселей к диапазону [0; 1]. Все метки класса необходимо векторизовать.



x\_train = x\_train.astype(“float32”) / 255

x\_test = x\_test.astype(“float32”) / 255

x\_train = x\_train.reshape((x\_train.shape[0], 28, 28, 1))

x\_test = x\_test.reshape((x\_test.shape[0], 28, 28, 1))

print(“x\_train shape:”, x\_train.shape)

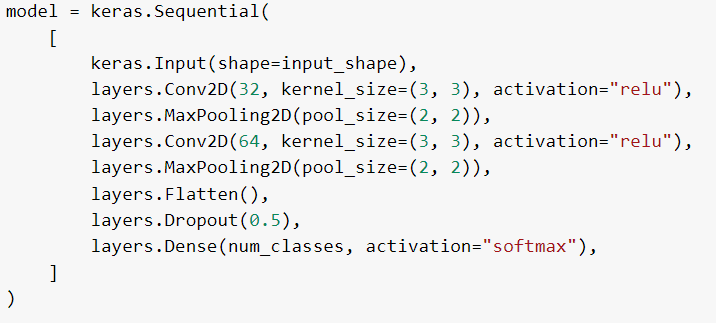
print(x\_train.shape[0], “train samples”)

print(x\_test.shape[0], “test samples”)

y\_train = keras.utils.to\_categorical(y\_train, num\_classes)

y\_test = keras.utils.to\_categorical(y\_test, num\_classes)

После предварительной обработки данных создаем модель.



model = keras.Sequential(

[

keras.Input(shape=input\_shape),

layers.Conv2D(32, kernel\_size=(3, 3), activation=”relu”),

layers.MaxPooling2D(pool\_size(2, 2)),

layers.Conv2D(64, kernel\_size=(3, 3), activation=”relu”),

layers.MaxPooling2D(pool\_size(2, 2)),

layers.Flatten(),

layers.Dropout(0.5),

layers.Dense(num\_classes, activation=”softmax”),

]

)

В данном случае формирование последовательной модели происходило путем указания всех слоев в списке параметра класса *Sequential*.

В данной модели слой *Input*отвечает за входной слой и в нем задается форма входных данных. Этот слой можно использовать вместо параметра *input\_shape*первого скрытого слоя.

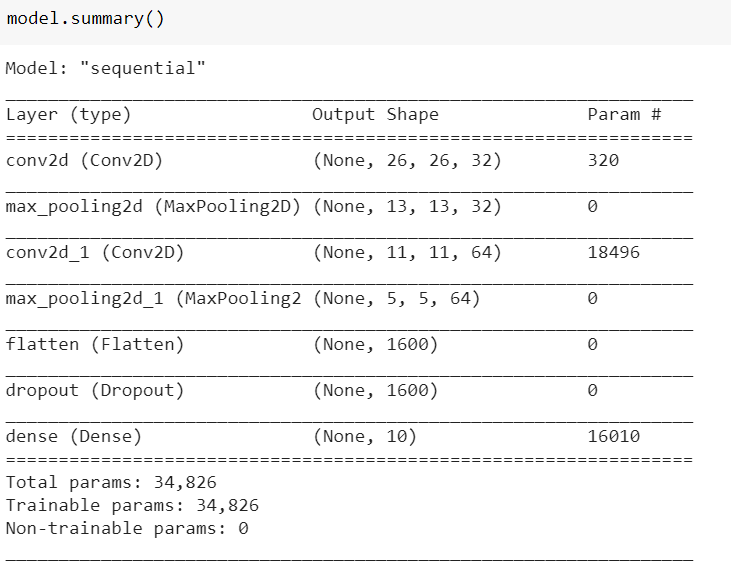
Слой *Conv2D* отвечает за саму свертку. Первый параметр слоя определяет количество фильтров на слое, параметр *kernel\_size*определяет форму ядра свертки (обычно используется свертки 3х3 и 5х5). Параметр *activation*определяет функцию активации, которая применяется после свертки.

Слой *MaxPooling2D* определяет слой пуллинга. Данный слой не имеет весовых коэффициентов, а также для не задается количество фильтров/нейронов. Основным параметром этого слоя является параметр *pool\_size*, определяющим область пуллинга.

Слой *Flatten*изменяет форму и ранг тензора таким образом, что выходной тензор имеет ранг 1, то есть вектор. Количество элементов данного вектора равна общему количеству элементов во входном тензоре. Этот слой в основном используется для того, чтобы связаться сверточные слои с полносвязными, и как слой *MaxPooling2D* не имеет весовых коэффицентов.

Слой *Dropout*добавляет прореживание в данных, не имеет весовых коэффициентов, и определяется одним значением. Это значение должно изменяться в диапазоне от 0 до 1 и определяет вероятность обнуления значения.

Для того, чтобы просмотреть архитектуру модели и оценить ее, используется функция *summary*.

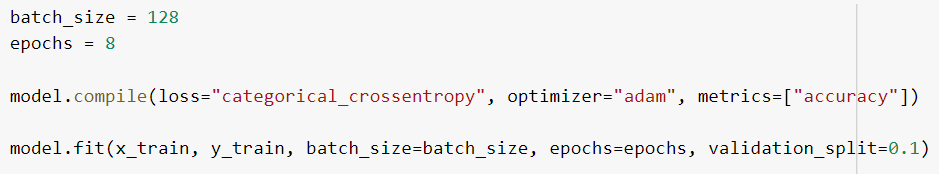


model.summary()

Это функция выводит таблицу с 3 колонками. Первая колонка хранит название и тип слоя. Вторая колонка отображает форму выходного тензора. Как можно видеть, в выводимых формах на первом месте стоит значение None. Это означает то, что может быть подано любое количество образцов, и это же количество образцов будет возвращено. Также можно заметить, что после слоя *Flatten*ранг тензора изменился (1600 = 5\*5\*64). Третья колонка показывается количество весовых коэффициентов на слое. В этой колонке для некоторых слоев указано 0, то есть на этих слоях не весовых коэффициентов.

В самом конце таблицы указано общее количество весовых коэффициентов, а также сколько из их будут обучаться, а сколько не будут.

Далее проводим обучение. Размер пакета возьмем равным 128, обучение будет длиться 8 эпох, а в качестве оптимизатора используем Adam. Под контрольную выборку выделим 10% обучающей выборки.



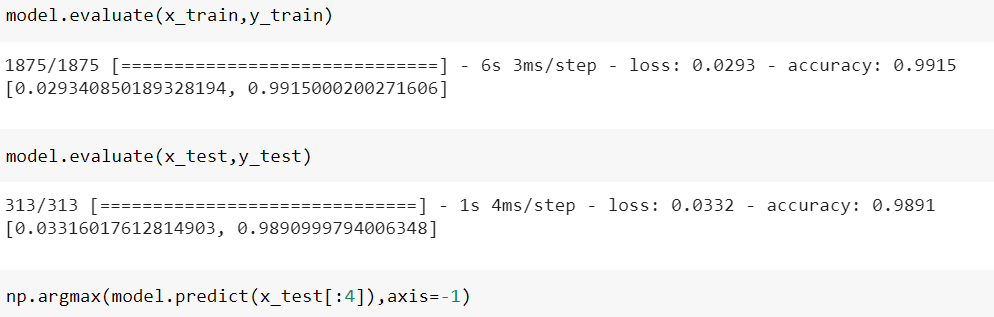
batch\_size = 128

epochs = 8

model.compile(loss=”categorical\_crossentropy”, optimizer=”adam”, metrics=[“accuracy”])

model.fit(x\_train, y\_train, batch\_size=batch\_size, epochs=epochs, validation\_cplit=0.1)

После обучения оценим модель на обучающей и тестовой выборке.

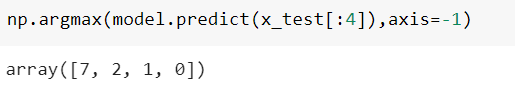
model.evaluate(x\_train, y\_train)

model.evaluate(x\_test, y\_test)

np.argmax(model.predict(x\_test[:4]),axis=-1)

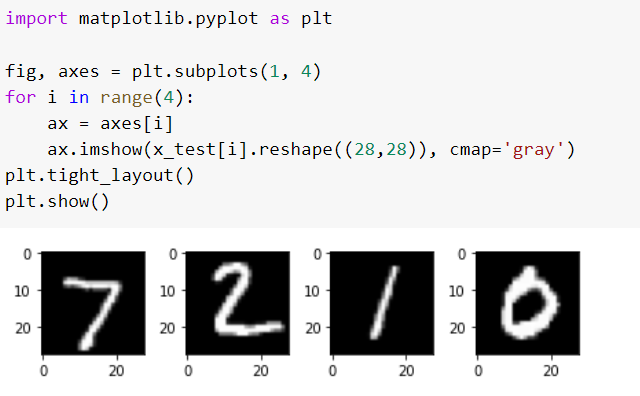
Как видно, значение ошибки и точности для двух выборок практически совпадают, что может говорить о том, что обучение прошло корректно и сеть не начала переобучаться.

Для проверки результатов. Выведем предсказание для первых четырех изображений.



np.argmax(model.predict(x\_test[:4]), axis=-1

Модель предсказала, что это цифры 7, 2, 1 и 0. Чтобы убедиться, что сеть предсказала цифры правильно, выведем сами изображения с цифрами.



import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, 4)

for i in range(4):

ax = axes[i]

ax.imshow(x\_test[i].reshape((28, 28),cmap=’gray’))

plt.tight\_layout()

plt.show()

Видно, что модель правильно определила изображенные цифры. Учитывая значения точности и ошибки, можно сказать, что модель обучилась корректно. С учетом того, что изображения довольно простые и не очень большие, то выбранная архитектура может являться избыточной.

Рекуррентные сети

Для обработки данных, представленных в виде последовательностей, можно использовать обычные полносвязнные нейронные сети или сверточные нейронные сети с одномерной сверткой. В таком случае, нейронная сеть анализирует всю последовательность за раз без учета информации о том, какой элемент за каким идет. Например, такой подход может быть использован при обработке строк, когда нужно анализировать только то, какие слова встречаются в строке, но не их порядок. Понятно, что такие сети решают узкий круг задач, и будут плохо справляться с такой задачей как прогнозирование последовательности.

Если стоит необходимость обработки последовательности с учетом порядка элементов, то применяются рекуррентные нейронные сети. Эти сети анализируют последовательность итеративно, сначала первый элемент, затем второй, и так далее до конца. Причем, что отличает рекуррентные сети от остальных, это наличие внутреннего состояние, которое извлекает и хранит информацию об уже просмотренных элементах. Таким образом, одно и то же значение может быть обработано рекуррентной нейронной сетью по-разному в зависимости от ее состояния.

Пример работы рекуррентной нейронной сети будет рассмотрен на примере решения задачи предсказания температуры. Рассматриваемого датасета нет в библиотеке Keras, поэтому его нужно загрузить отдельно.



import pandas as pd

import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow import keras

from zipfile import ZipFile

import os

uri = <https://storage.googleapis.com/tensorflow/tf-keras-datasets/jena_climate_2009_2016.csv.zip>

zip\_path = keras.utils.get\_file(origin=uri, fname=”jena\_climate\_2009\_2016.csv.zip”)

zip\_file = ZipFile(zip\_path)

zip\_file.extractall()

csv\_path = “jena\_climate\_2009\_2016.csv”

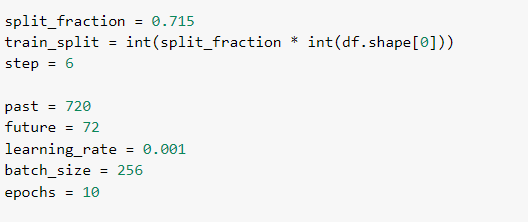
df = pd.read\_csv(csv\_path)

Датасет содержит значения по 14 признакам, описывающим погоду. Признаки снимались каждые 10 минут начиная с 10 января 2009 г. и заканчивая 31 декабрем 2016 г.

Загруженный датасет представлен в виде таблицы, в которой каждый столбец отвечает за признак, а каждая строка определяет временную отметку. Для того, чтобы можно было обучать рекуррентную сеть на этих данных, ее нужно разбить на небольшие последовательности. Сначала определим процент разбиения выборки на обучающую и контрольную через параметр *split\_fraction* равным 71,5%. Параметры для разбиения датасета на последовательности:

* Шаг (step) = 6 – определяет с каким шагом будут учитываться временные метки. Значение 6 в данном случае показывается, что будет браться значение раз в час.
* Длина последовательности (past) = 720. Задает количество временных интервалов, в этой задаче это обозначает, что предсказание будет проводится на основании 120 часов.
* Через какой интервал будет предсказываться значение (future) = 72. Означает то, что предсказываем значение, которое будет через 12 часов после конца последовательности.

Также сразу зададим параметры для обучения: скорость обучения 0.001, размер пакета 256, количество эпох 10.



split\_fraction = 0.715

train\_split = int(split\_fraction \* int(df.shape[0]))

step = 6

past = 720

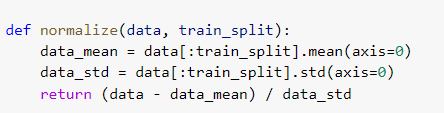
future = 72

learning\_rate = 0.001

batch\_size = 256

epoch = 10

Помимо разбиения на последовательности, данные необходимо будет нормализовать. Поэтому введем функцию для этого.



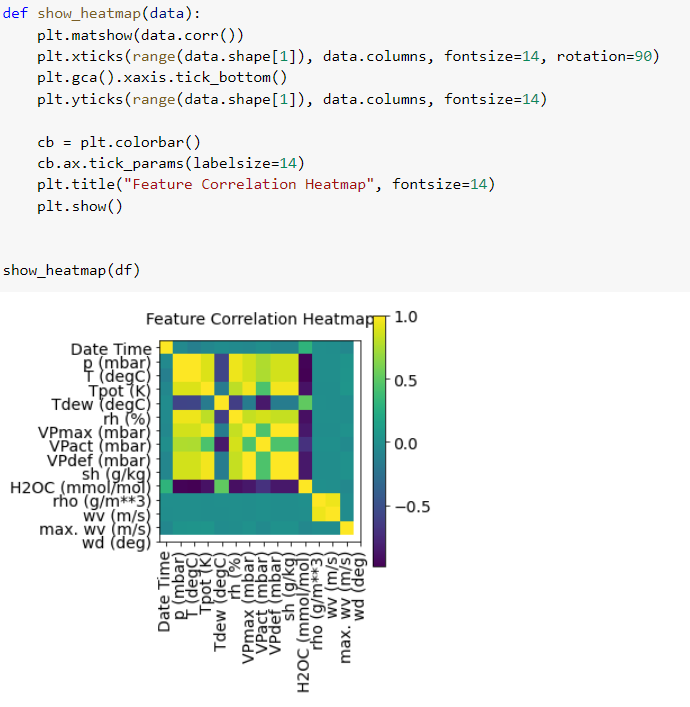
def normalize(data, train\_split):

data\_mean = data[:train\_split].mean(axis=0)

data\_std = data[:train\_split].std(axis=0)

return(data – data\_mean) / data\_std

Так как датасет содержит признаки, которые сильно связаны друг с другом (например, температура в Цельсиях и в Кельвинах), то такие признаки не несут полезной информации, а в некоторых случаях могут только ухудшить решение задачи. Поэтому, построим тепловую карту корреляций признаков.



def show\_heatmap(data):

plt.matshow(data.corr())

plt.xticks(range(data.shape[1]), data.columns, fontsize=14, rotation= 90)

plt.gva().xaxis.tick\_bottom()

plt.yticks(range(data.shape[1]), data.columns, fontsize=14)

cb = plt.colorbar()

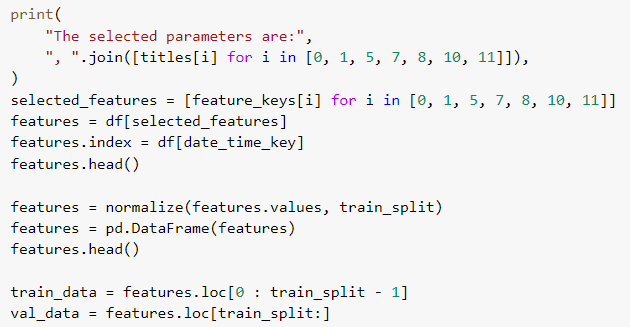
cb.ax.tick\_params(labelsize = 14)

plt.title(“Feature Correlation Heatmap”, fomtsize=14)

plt.show()

show\_heatmap(df)

Поэтому отберем те признаки, которые слабо коррелированы между собой, сформируем обучающую выборку и стандартизируем значения на основе статистики обучающей выборки.



print(

“The selected parameters are:”,

“, ”.join([titles[i] for i in [0, 1, 5, 7, 8, 10, 11]]),

)

selected\_features = [feature\_keys[i] for i in [0, 1, 5, 7, 8, 10, 11]]

features = df[selected\_features]

features.index = df[date\_time\_key]

features.head()

features = normalize(features.values, train\_split)

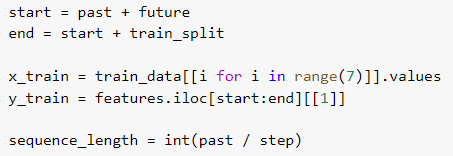
features = pd.DataFrame(features)

features.head()

train\_data = features.loc[0 : train\_split - 1]

val\_data = features.loc[train\_split:]

Определим диапазоны меток (значения, которые необходимо предсказать) в обучающей выборке. В нашем случае метки начинаются с индекса 792 (720 + 72). Также, так как сейчас данные хранятся в датафреме из библиотеки Pandas, выделим чистые значения.



start = past + future

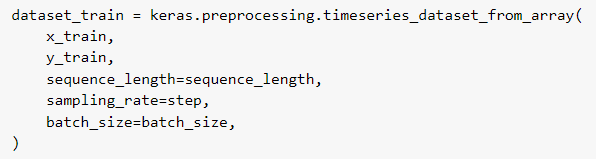
end = start + train\_split

x\_train = train\_data[[i for in range(7)]].values

y\_train = features.iloc[start:end][[1]]

sequence\_lenght = int(past / step)

Используя функцию *timeseries\_dataset\_from\_array,* сгенерируем последовательности для обучения. В эту функцию подается массив с данными для обучения, массив с метками, длинной последовательности (в итоге она равна 120 = 720/6), каким шагом брать временные метки и размером пакета.



dataset\_train = keras.preprocessing.timeseries\_dataset\_from\_array(

x\_train,

y\_train,

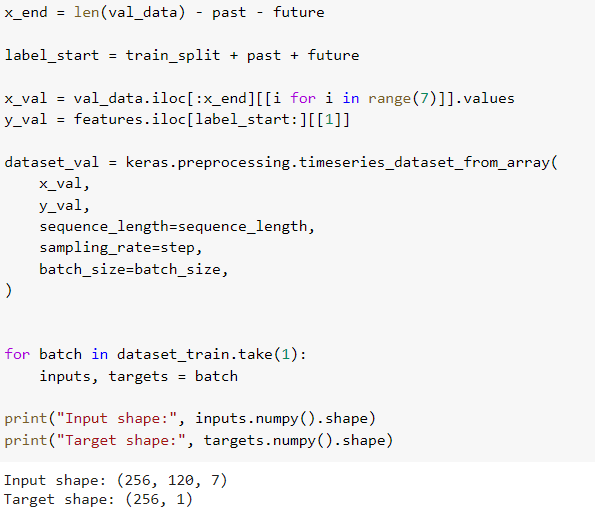
sequence\_lenght=sequence\_lenght,

sampling\_rate=step,

batch\_size=barch\_size,

)

Аналогично сформируем контрольную выборку.



x\_end = len(val\_data) – past – future

label\_start = train\_split = train\_split + past + future

x\_val

val\_data.iloc[:x\_end][[i for in range(7)]].values

y\_val = features.iloc[label\_start:][[1]]

dataset\_val keras.preprocessing.timeseries\_dataset\_from\_array(

x\_val,

y\_val,

sequence\_lenght=sequence\_lenght,

sampling\_rate=step,

batch\_size=batch\_size,

)

for batch in dataset\_train.take(1):

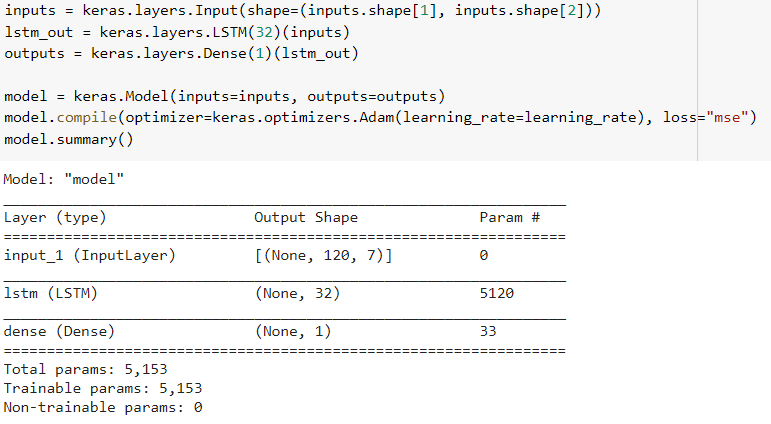
inputs, targets = batch

print(“Input shape:”, inputs.numpy().shape)

print(“Target shape:”, targets.numpy().shape)

Также в конце вывели один пакет. Как видно, размер пакета 256, длина последовательности 120, количество признаков 7, количество значений для предсказания 1.

Далее создадим саму модель. В библиотеке Kerasесть три типа рекуррентных слоев SimpleRNN (стандартный рекуррентный слой), GRUи LSTM. Так как в задаче используются довольно длинные последовательности (120 временных меток), то модель будет содержать слой LSTM.



inputs = keras.layers.Input(shape=(inputs,shape[1], inputs.shape[2]))

lstm\_out = keras.layers.LSTM(32)(inputs)

outputs = keras.layers.Danse(1)(lstm\_out)

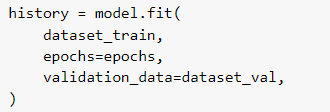
model = keras.Model(inputs=inputs, outputs=outputs)

model.compile(optimizer=keras.optimizer.Adam(learning\_rate), loss=”mse”)

model.summary()

Как можно заметить, сначала создавались слои таким образом, что предшествующий слой подавался в качестве аргумента следующему, и только потом формировался нейронная сеть с использованием класса Model. Данный подход будет подробнее разобран в следующем разделе «Сети произвольной архитектуры».

После того как модель сформирована, обучаем ее.



history = model.fit(

dataset\_train,

epochs=epochs,

validation\_data=dataset\_val,

)

Причем, размер пакета не указывается, так как данные изначально были разбиты на пакеты.

После обучения строим график ошибки, чтобы проверить наличие переобучения.



def visualize\_loss(history, title):

loss = history.history[“loss”]

val\_loss = history.history[“val\_loss”]

epochs = range (len(loss))

plt.figure()

plt.plot(epochs, loss, “b”, label=”Training loss”)

plt.plot(epochs, val\_loss, “r”, label=”Validation loss”)

plt.title(title)

plt.xlabel(“Epochs”)

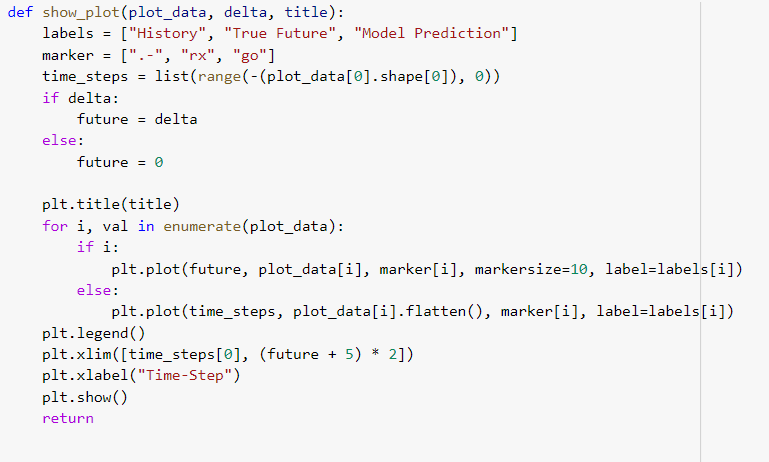
plt.ylabel(“Loss”)

plt.legend()

plt.show()

visualize\_loss(history, “Training and Validation Loss”)

Также, напишем функцию для визуализации последовательности (только температуры) и предсказанного значения температуры.



def show\_plot(plot\_data,delta, title):

labels = [“History”, “True Future”, “Model Prediction”]

marker = [“.-”, “rx”, “go”]

time\_steps = list(range(-(plot\_data[0].shape[0]), 0))

if delta:

future = delta

else:

future = 0

plt.title(title)

for i, val in enumerate(plot\_data):

if i:

plt.plot(future, plot\_data[i], marker[i], markersize=10, label=labels[i])

else:

plt.plot(time\_steps, plot\_data[i].flatten(), marker[i], label=labels[i])

plt.legend()

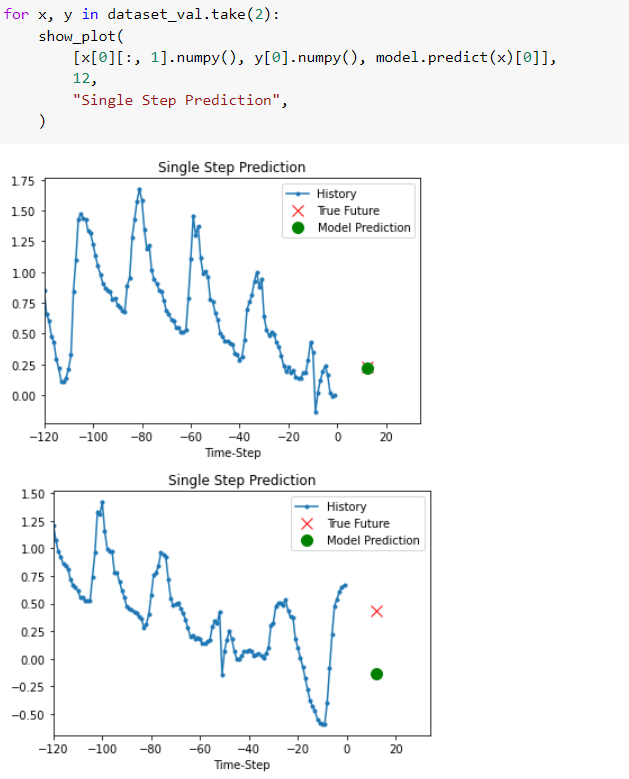
plt.xlim([time\_steps[0], (future + 5) \* 2])

plt.xlabel(“Time-Step”)

plt.show()

return

Выведем результаты предсказания для первых двух последовательностей.



for x, y in dataset\_val.take(2):

show\_plot(

[x][0][:, 1].numpy(), model.predict(x)[0]],

12,

“Single Step Prediction”,

)

Как видно, в случае первой последовательности значение было предсказано близко, а во втором случае наблюдается сильное отклонение.

При более тонкой настройке модели можно достигнуть лучших результатов.

Сети произвольной архитектуры

Хоть и большой спектр задач решается последовательными нейронными сетями с одним входом и выходом, существуют задачи, в которых такие модели не применить. Примеры таких задач:

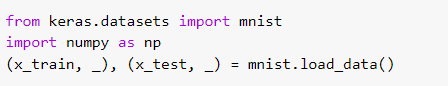
* Автокодирование – в этих задачах используются сети, которые обучаются кодировать данные, а потом их декодировать. После обучения, такие сети часто разделяется на две меньших модели. Первая модель кодирует данные и возвращает вектор-кодировку. Вторая модель принимает вектор-кодировку и восстанавливает данные.
* Ответ на вопрос относительно изображения – в данном случае потребуется нейронная сеть, которая на вход принимает изображение и строку, и возвращает строку с ответом. Для этого модель должна иметь два входа.
* Выделение из изображения объекта и фона – для этого модели необходимо два входа, чтобы вернуть два изображения.

Для создания таких моделей используется FunctionalAPIиз библиотеки Keras. Этот APIможет быть использован для создания моделей с несколькими входными и выходными слоями, нелинейной структурой, слоями, которые разделяются между разными нейронными сетями.

Автокодировщики

Первым примером использования FunctionalAPIбудет реализация простого авктокодировщика для датасетаMNIST.

Загрузим датасет, причем для данной задачи требуются только изображения без меток классов.

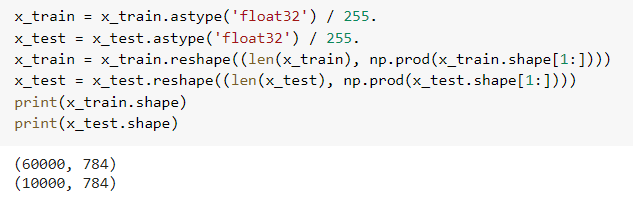


from keras.datasets import mnist

import numpy as np

(x\_train, \_), (x\_test, \_) mnist.load\_data()

После загрузки нормируем данные.



x\_train = x\_train.astype(‘float32’) / 255.

x\_test = x\_test.astype(‘float32’) / 255.

x\_train = x\_train.reshape((len(x\_train), np.prod(x\_train.shape[1:])))

x\_test = x\_test.reshape((len(x\_test), np.prod(x\_test.shape[1:)])))

print(x\_train.shape)

print(x\_test.shape)

Далее создается модель.



import keras

from keras import layers

encoding\_dim = 32

input\_img = keras.Input(shape=(784,))

encoded = layers.Dense(encoding\_dim, activations=’relu’, activity\_regularizer=regularizers.l1(10e-5))(input\_img)

decoded\_layers.Dense(784, activation=’sigmoid’)(encoded)

autoencoder = keras.Model(input\_img, decoded)

В приведенном коде сначала задаем параметр *encoding\_dim*, которые отвечает за размер вектора-кодировки, которые будет получается. В данном случае будет использоваться размер равный 32. Далее создаются слои.

Сначала создается слой *Input (переменная input\_img)*, который отвечает за получение данных и передачи их дальше. Особенность этого слоя заключается в том, что он находится в подмодуле *keras*, а не *keras*.*layers* как все остальные слои.

Следующим создается полносвязный слой *(переменная encoded)*. Данный слой будет принимать изображение, представленное в виде вектора и возвращать вектор-кодировку. Можно обратить внимание, что после указания конструктора слоя, далее передается аргумент *(inpit\_img)*. Таким образом, мы этому слою передаем слой, который должен находится перед ним. Это можно делать по той причине, что при создании слоев на самом деле не создается объект слоя, а возвращается функция, производящая необходимые вычисления. И в библиотеке Keras, функции создающие слои создают функции высшего порядка, то есть такие функции, которые могут принимать другие функции и возвращать новую функцию. Поэтому данный подход к созданию архитектуры нейронных сетей в библиотеке Kerasназывается FunctionalAPI.

Также, при создании слоя *encoded*, указывался параметр *activity\_regularizer*. Этому параметру была назначена L1 регуляризация с интенсивностью 10-5. Таким образом была добавлена L1 регуляризация к выходным значениям слоя. Добавления подобной регуляризации делает автокодировщик разреженным, и улучшает интерпретируемость вектора-кодировки. Помимо регуляризации выходных значений на слое, можно добавить регуляризацию весов слоя и весов сдвига через параметры *kernel\_regularization*и *bias\_regularization* соответственно. Помимо L1 регуляризации (*l1*), можно применять L2 регуляризацию (*l2*) и L1 и L2 регуляризации одновременно (*l1\_l2*).

Последним создается полносвязный слой *(переменная decoded)*, который будет декодировать представление, возвращаемое слоем *encoded*. Поэтому его размер должен соответствовать размерности входных данных.

После того как все слои созданы и указаны связи между ними, они объединяются в одну модель. Для этого необходимо вызвать *keras.Model*, передав туда в качестве первого аргумента слой или список слоев типа Input, а в качестве второго выходной слой или список выходных слоев.

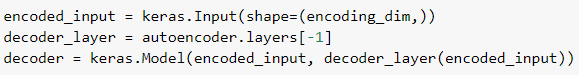
В итоге получилась полная модель автокодировщика, которая принимает изображение в виде вектора, кодирует его и сразу декодирует.

Далее, из этих же слоев нужно получить кодирующую сеть и декодирующую сеть по отдельности. Для того, чтобы получить кодировщик, необходимо создать модель, но в качестве выходного слоя указать не *decoded*, а *encoded*.



encoder = keras.Model(input\_img, encoded)

Получить декодировщик аналогичным способом не получится, так как при создании модели первым аргументом необходимо указывать слой типа Input. Но у слоя *decoded*, входным слоем по сути является слой *encoded*, который имеет тип Dense. Для того, чтобы решить эту проблему необходимо создать новый слой Input, который будет принимать данные, по форме совпадающие с выходом слоя *encoded*. После этого можно будет связать новый слой Inputи слой *decoded*.

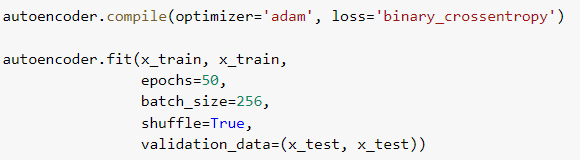


encoded\_input = keras.Input(shape=(encoding\_dim,))

decoder\_layer = autoencoder.layers[-1]

decoder = keras.Model(encoded\_input, decoder\_layer(encoded\_input))

После этих операций, получены три переменные *autoencoder, encoder*и *decoder*, которые, по сути, соответствуют трем разным нейронным сетям, но которые между собой разделяют определенные слои. И теперь достаточно обучить только *autoencoder*.



autoencoder.compile(optimizer=’adam’, loss=’binary\_crossentropy’)

autoencoder.fit(x\_train, x\_train,

epochs=50,

batch\_size=256,

shuffle=True,

validation\_data(x\_test, x\_test))

Для модели кодер-декодер в качестве входных и выходных данных указываются одни и те же данные.

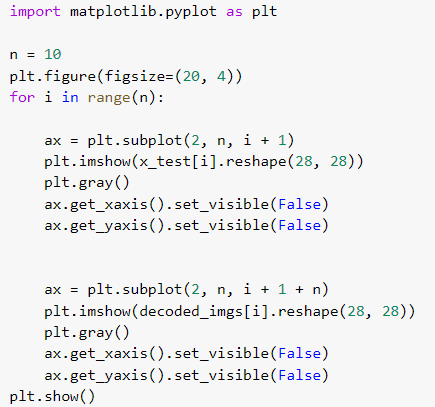
После обучения, можно отдельно кодировать данные, а затем декодировать уже используя вектор-кодировку.



encoded\_imgs = encoder.predict(x\_test)

decoded\_imgs = decoder.predict(encoded\_imgs)

Проверим работу кодирования и декодирования сравнив исходные и восстановленные изображения. Для этого выполним следующий код.



import matplotlib.pyplot as plt

n = 10

plt.figure(figsize=20,4))

for i in range(n):

ax = plt.subplot(2, n, i + 1)

plt.imshow(x\_test[i].reshape(28, 28))

plt.gray()

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

ax = plt.subplot(2, n, i + 1 + n)

plt.imshow(decoded\_imgs[i].reshape(28, 28))

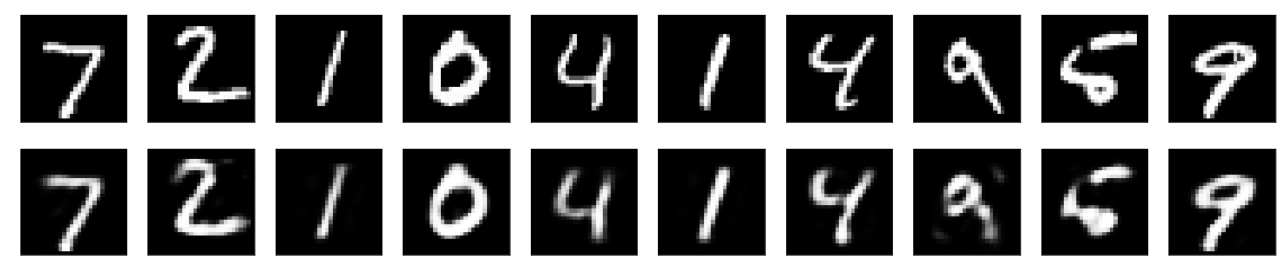
plt.gray()

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

plt.show()

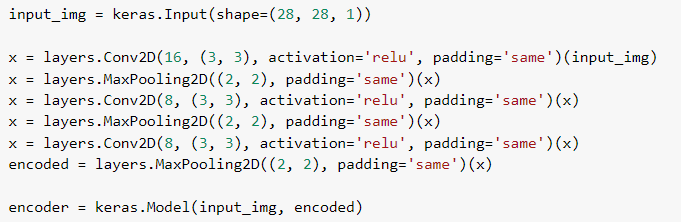
В результате получим результаты для первых 10 цифр из тестовой выборки.



Можно заметить, что после восстановления цифры на изображениях распознаются. Для изображений наблюдается небольшое размытие, и в некоторых случаях небольшие искажения. Это связано с тем, что при понижении размерности данных и их восстановления практически всегда теряется часть информации.

Рассмотрим более сложный автокодировщик, который имеет больше скрытых слоев и использует сверточные слои для обработки изображений. Такой автокодировщик называется глубоким сверточным. Для построения такой модели кодер-декодер сначала по отдельности создадим кодировщик и декодировщик, а затем объединим их в одну модуль, которую можно обучить.

Кодировщик будет принимать изображение размером 28х28 пикселей, проводить свертку без изменения размера (для этого в сверточных слоях необходимо задать параметр *paddint=’same’*), и выдавать тензор с формой 4х4х8.



input\_img = keras.Input(shape(28, 28, 1))

x = layers.Conv2D(16, (3, 3), activations=’relu’, padding=’same’)(input\_img)

x = layers.MaxPooling2D((2, 2), padding=’same’)(x)

x = layers.Conv2D(8, (3, 3), activations=’relu’, padding=’same’)(x)

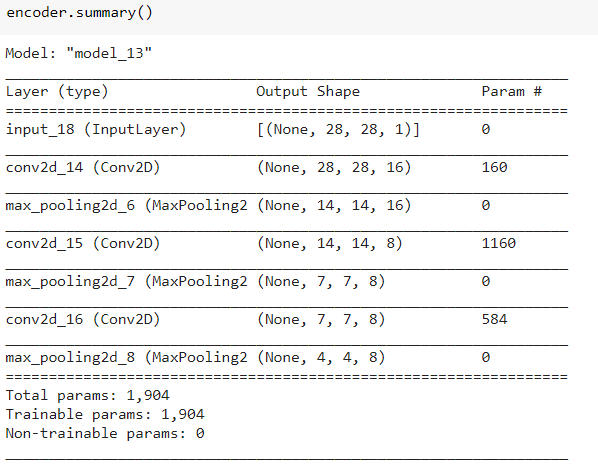
x = layers.MaxPooling2D((2, 2), padding=’same’)(x)

x = layers.Conv2D(8, (3, 3), activations=’relu’, padding=’same’)(x)

endcoded = layers.MaxPooling2D((2, 2), padding=’same’)(x)

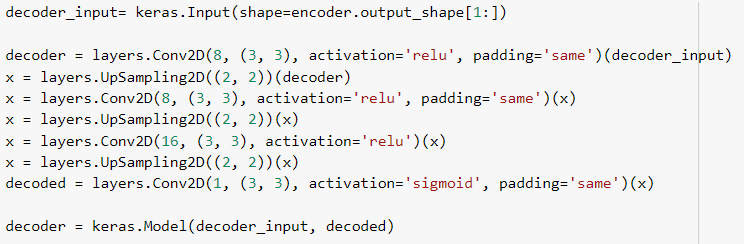
encoded = keras.Model(input\_img, encoded)

Если вывести структуру сети, то видно, что после слоев свертки размер карты признаков не меняется, и модель имеет 1904 весовых коэффициента.



encoder.summary()

По аналогии создается декодировщик. Для того, чтобы задать размер входного слоя корректно, достаточно узнать размер выходного тензора у кодировщика. Для того, чтобы из тензора меньших размеров получить тензор большего, используется слой *UpSampling2D*, который повторяет значение



decoder\_input= keras.Input(shape=encoder.output\_shape[1:])

decoder = layers.Conv2D(8, (3, 3), activations=’relu’, padding=’same’)(decoder\_input)

x = layers.UpSampling2D((2, 2))(decoder)

x = layers.Conv2D(8, (3, 3), activation=’relu’, padding=’same’)(x)

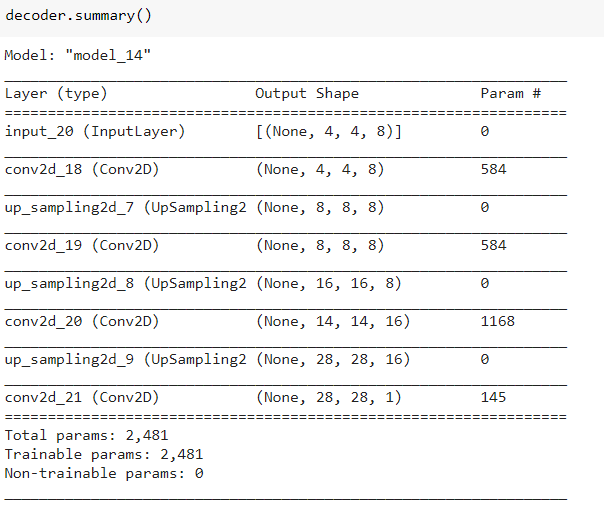
x = layers.UpSampling2D((2, 2))(x)

x = layers.Conv2D(16, (3, 3), activation=’relu’)(x)

x = layers.UpSampling2D((2, 2))(x)

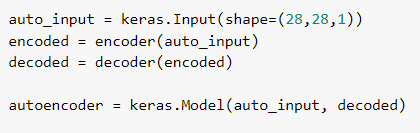
decoder = layers.Conv2D(1, (3, 3), activations=’sigmoid’, padding=’same’)(x)

После вывода структуры декодировщика видно, что выходной тензор совпадает по форме с изображением, но на промежуточных слоя размеры отличаются, так как не всегда возможно повторить операции в обратном порядке. Из-за этого количество весовых коэффициентов модели больше, чем у кодировщика, и их количество равно 2481.



decoder.summary()

После того, как созданы модели, их можно объединить в одну нейронную сеть. При создании надо снова задать входной слой, так как в функцию *Model*нельзя передать другую модель.



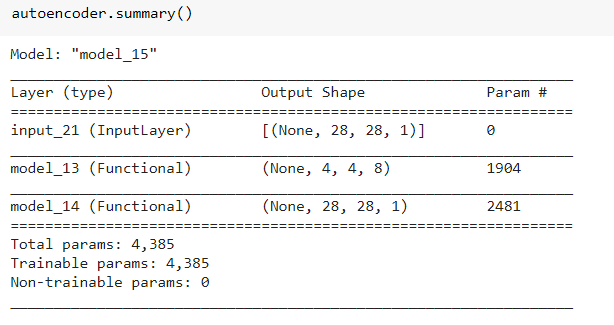
auto\_input = keras.Input(shape=(28, 28, 1))

encoded = encoder(auto\_input)

decoded = decoder(encoded)

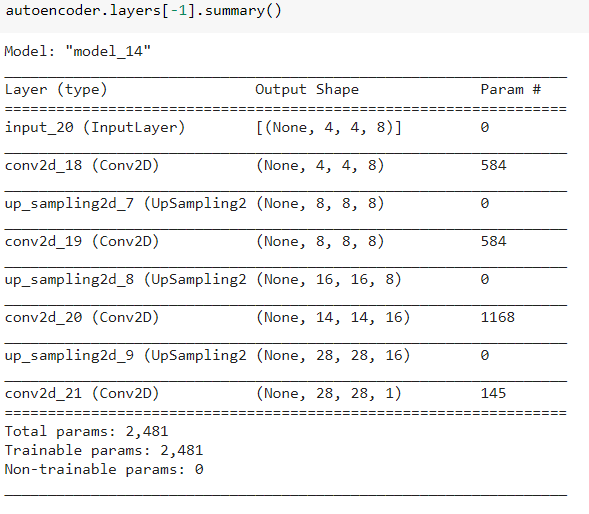
autoencoder = keras.Model(auto\_input, decoded)

Если вывести структуру полученной нейронной сети, то можно увидеть, что вместо слоев указаны уже модели с пометкой *Functional*. Причем, название моделей и количество весовых коэффициентов соответствует ранее созданным нейронным сетям.



autoencoder.summary()

Например, если вывести структуру последнего слоя, то она будет идентична структуру декодировщика.

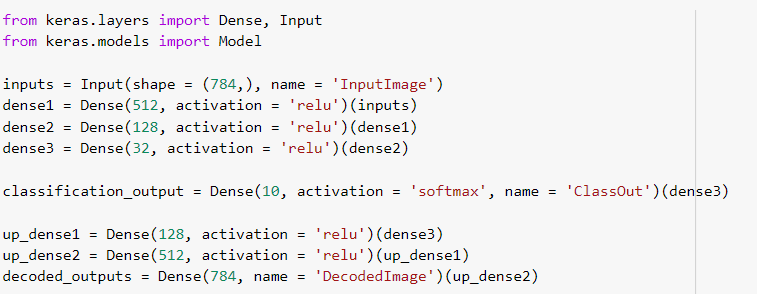


autoencoder.layers[-1].summary()

Это возможно за счет того, что в библиотеке Kerasсама модель может вызываться также как и слой, и соответственно использоваться в архитектуре других моделей. Это позволяет быстро разрабатывать модели глубокого обучения, используя уже готовые модели, которые эффективно решают схожую задачу. Например, для классификации изображений с людьми и без можно использовать сеть, которая умеет качественно кодировать изображения с подходящим содержимым, и дополнить такую модель лишь слоями отвечающими непосредственно за классификацию. То есть необходимо будет решить задачу классификации векторов-представлений, а не самих изображений.

Нейронные сети с несколькими выходными/входными слоями

Рассмотрим построение нейронной сети, которая будет принимать изображение в виде вектора, кодировать его, а затем сможет классифицировать и восстанавливать его. Для создание такой сети, необходимо создать необходимые слои и слой возвращающий вектор-кодировку передать в 2 разных слоя.



from keras.layers import Dense, Input

from keras.models import Model

inputs = input(shape = (784,), name = ‘InputImage’)

dense1 = Dense(512, activations = ‘relu’)(inputs)

dense2 = Dense(128, activations = ‘relu’)(dense1)

dense3 = Dense(32, activations = ‘relu’)(dense2)

classification\_output = Dense(10, activation = ‘softmax’, name = ‘ClassOut’)(dense3)

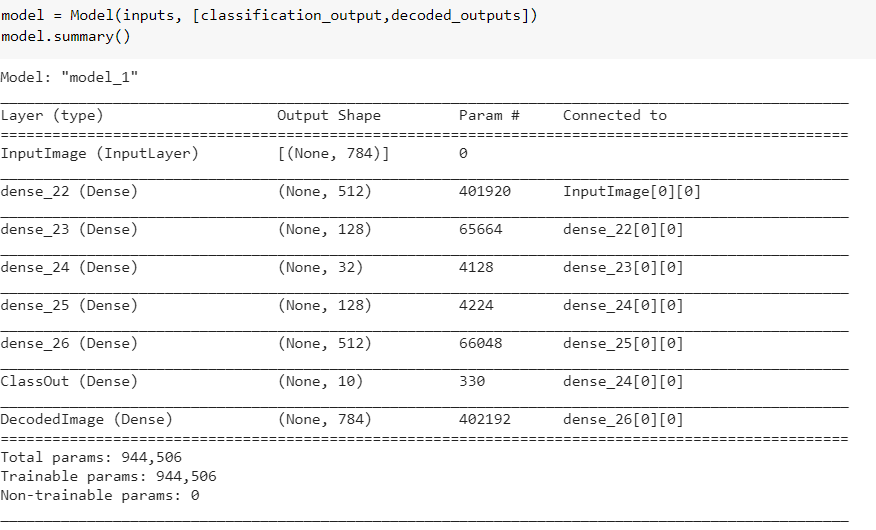
up\_dense1 = Dense(128, activations = ‘relu’)(dense3)

up\_dense2 = Dense(512, activations = ‘relu’)(up\_dense1)

decode\_outputs = Dense(728, name = ‘DecodedImage’)(up\_dense2)

В данном случае, слой *dense3*возвращает закодированное представление данных, а затем он передается слою *classification\_output*и *up\_dense1*. Слой *classification\_output*отвечает за классификацию, а слой *up\_dense1* является первым скрытым слое декодировщика. Также всем входным и выходным слоя задали имена через параметр *name*. Использовать пользовательских имен слоев удобно при просмотре структуры сетей со сложной топологией и/или большим количеством слоев.

Для того, чтобы задать модель с несколькими выходами, необходимо вместо одного выходного слоя указать список выходных слоев.

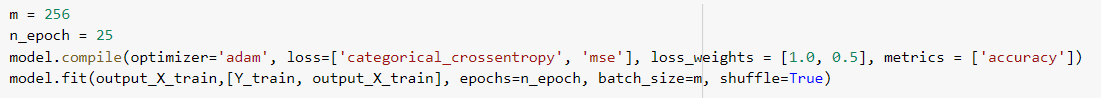


model = Model(inputs, [classification\_output,decode\_outputs])

model.summary()

Как видно, после вывода структуры, выходные слои располагаются в конце в том порядке, в каком они были переданы в функцию *Model*.

При подготовке к обучению, в функции *compile*в качестве параметра *loss*необходимо задать отдельно функции ошибок для каждого из выходных слоевв виде списка. Порядок функций ошибок соответствует порядку слоев, который был указан при построении модели. Также, можно указать параметр *loss\_weights*, который принимает список весовых коэффициентов. Этот параметр определяет степень влияния ошибки каждого слоя на обучение. При обучении, в функцию *fit*, выходные метки также задаются в виде списка.



m = 256

n\_epoch = 25

model.compile(optimizer=’adam’, loss=[‘categorical\_crossentropy’, ‘mse’], loss\_weight = [1.0, 0.5], metrics = [‘accuracy’])

model.fit(output\_X\_train, [Y\_train, output\_X\_train], eponchs=n\_epoch, batch\_size=m, shuffle=True)

Создание нейронной сети с несколькими входными слоями происходит по аналогии с построением сети с несколькими выходными слоями. Рассмотрим создание модели, которая на выход принимает монохромное изображение и изображение в цветовой схеме RGB, и на основе этой информации проводится классификацию.



from keras.layers import Conv2D, MaxPolling2D, Flatten, concatenate

inputs = Input(shape = 28, 28, 1), name = ‘GreyImage’

conv1 = Conv2D(16, (3,3), activation = ‘relu’, padding = “SAME”)(inputs)

pool1 = MaxPooling2D(poll\_size = (2, 2), strides = 2)(conv1)

conv2 = Conv2D(32, (3,3), activation = ‘relu’, padding = “SAME”)(pool1)

pool2 = MaxPooling2D(poll\_size = (2, 2), strides = 2)(conv2)

flat\_1 = Flatten()(pool2)

inputs\_2 = Input(shape = (28, 28, 3), name = ‘RGBImage’)

conv1\_2 = Conv2D(16, (3,3), activation = ‘relu’, padding = “SAME”)(inputs\_2)

pool1\_2 = MaxPooling2D(poll\_size = (2, 2), strides = 2)(conv1\_2)

conv2\_2 = Conv2D(32, (3,3), activation = ‘relu’, padding = “SAME”)(pool1\_2)

pool2\_2 = MaxPooling2D(poll\_size = (2, 2), strides = 2)(conv2\_2)

flat\_2 = Flatten(poll2\_2)

concat = comcatenate([flat\_1,flat\_2])

dense1 = Dense(512, activation = ‘relu’)(concat)

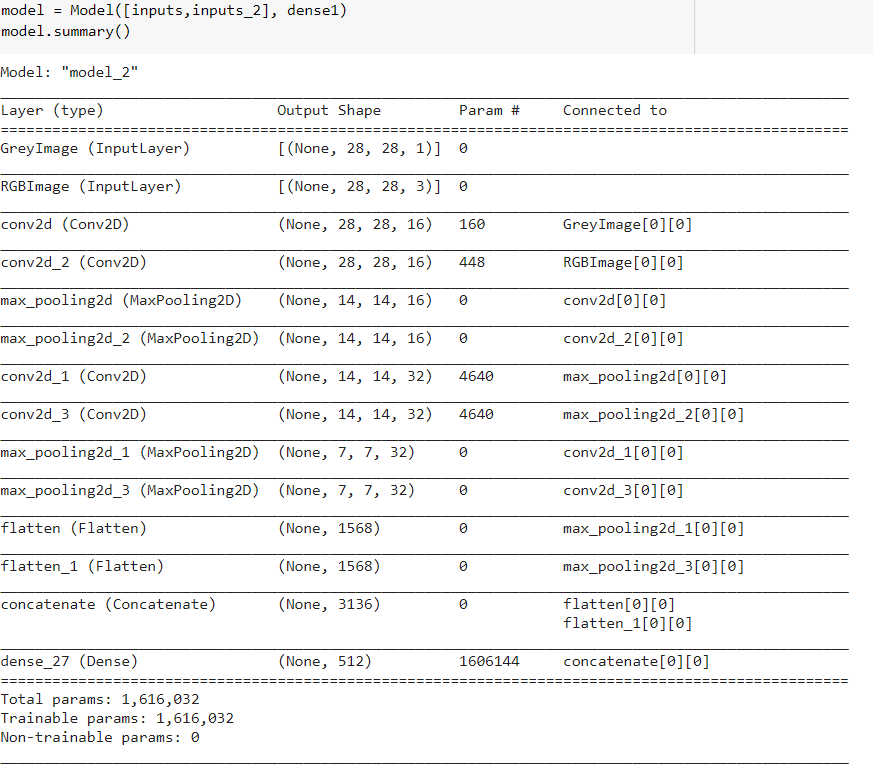
dense2 = Dense(128, activation = ‘relu’)(dense1)

dense3 = Dense(32, activation = ‘relu’)(dense2)

output = Dense(10, activation = ‘softmax’, name = ‘ClassOut’)(dense3)

Единственное отличие от создания нейронной сети с несколькими выходами заключается в том, как обрабатываются данные в области ветвления архитектуры. Если случае нескольких выходных слоев достаточно было передать слой нескольким разным, то в данном случае нужно объединить несколько слоев в один. В данном примере для этого используется слой *concatenate*, который располагает два тензора рядом. Также, в библиотеке Keras можно сложить, перемножить, вычесть, усреднить два тензора, найти минимум или максимум или рассчитать скалярное произведение двух тензоров.

При создании самой модели, достаточно вместо одного входного слоя указать список слоев, которые имеют тип *Input*.После вывода структуры нейронной сети видно, что перед слоем *concatenate*находятся два слоя *Flatten*, которые возвращают вектора длиной 1568. Эти два слоя передают данные на слой *concatenate*, и выход слоя объединения уже возвращает вектор с длиной 3136 = 1568 + 1568.



model = Model([inputs, inputs\_2], dense1)

model.summary()

Подготовка к обучению происходит как обычно, так как здесь всего один выходной слой, то нет необходимости указывать несколько функций потерь и их степени влияния. А при вызове функции *fit*, необходимо уже задать список выходных данных.

*FunctionalAPI* позволяет: создать модель с несколькими входами и выходами одновременно, причем их количество может быть различно; создать модель с быстрыми соединениями как в ResNet или создать фрактальную нейронную сеть, которая в качестве слое использует саму себя.

Обратные вызовы в Keras

Во всех рассмотренных ранее примерах не происходило никакого взаимодействия с моделью во время ее обучения. Поэтому, могло возникнуть переобучение модели или модель могла не дойти до точки минимума. Чтобы устранить эти проблемы, необходимо контролировать процесс обучения напрямую. Для такого контроля в библиотеке Keras есть механизм обратных вызовов (*Callback*). Механизмы обратного вызова имеет доступ к данным, на которых модель обучается и контролируется, параметрам обучения, и самой модели.

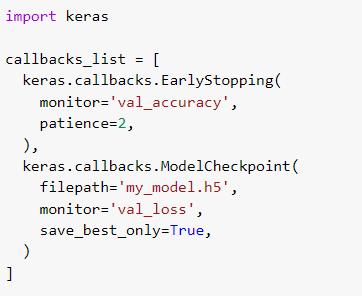
Стандартные обратные вызовы

В библиотеке Keras есть набор уже реализованных обратных вызовов:

* *ModelCheckpoint* – сохраняет модель на определенном этапе обучения. Обычно используется, чтобы сохранить наилучшую модель.
* *EarlyStoping* – останавливает обучение если какая-то метрика перестает улучшаться.
* *LearningRateScheduler* – изменяет, обычно уменьшает, скорость обучения согласно заданной в качестве параметра функции. Обычно функция меняет скорость обучения в зависимости от эпохи.
* *ReduceLROnPlateau* – уменьшает скорость обучения если отслеживаемая метрика перестает меняться.
* *TerminateOnNaN* – останавливает обучение, если в ходе вычислений было получено значение NaN.
* *CSVLogger* – записывает результат каждой эпохи в файл.

Рассмотрим применение обратных вызов на примере задачи бинарной классификации, в которой наблюдалось сильное переобучение. Поэтому, для улучшения получаемого результата будет сохраняться лучшая модель, а также будет использоваться ранняя остановка.

Создание модели полностью идентично тому, как это делалось в самом первом примере. Перед началом обучения создадим список из двух обратных вызовов *EarlyStopping* и *ModelCheckpoint*.



import keras

callbacks\_list = [

keras.callbacks.EarlyStopping(

monitor=”val\_accuracy”,

patience=2,

),

keras.callbacks.ModelChekpoint(

filepath=’my\_model.h5’,

monitor+’val\_loss’,

save\_best\_only= True,

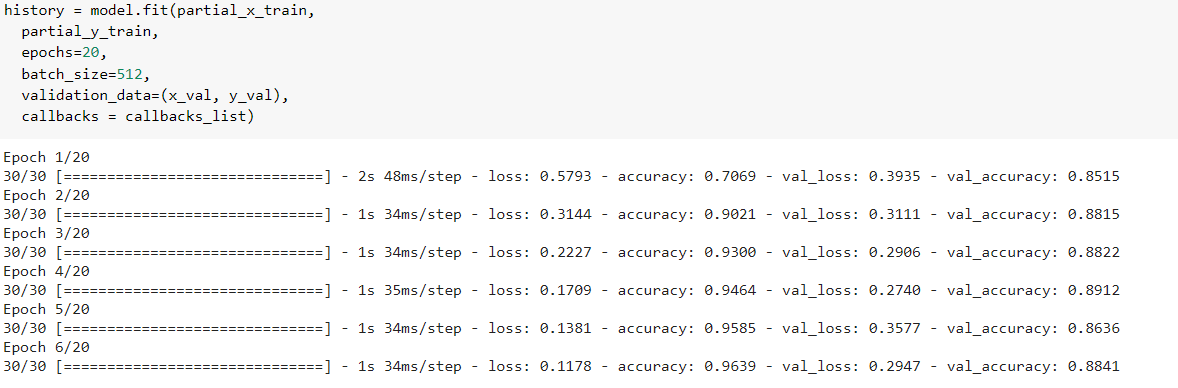
)

]

Для обратного вызова *EarlyStopping*зададим в качестве отслеживаемой метрики (*monitor*) точность на контрольной выборке. Параметр *patience*определяет в течении какого количества эпох отслеживаемая метрика не изменяется. В нашем случае остановка обучения будет происходить, если точность на контрольной выборке не менялась в течении двух эпох.

Для обратного вызова *ModelCheckpoint*задается название файла, в который будет сохраняться модель (формат h5 является стандартным для хранения модели), параметр *monitor*также задает метрику, на основе которой выбирается лучшая модель, параметр *save\_best\_only*равным True означает, что сохраняется только самая лучшая модель. Если параметр *save\_best\_only*имеет значение False, то при нахождении очередной более лучшей модели будет также сохраняться предыдущая лучшая модель.

Далее необходимо в функции *fit* через параметр *callbacks* указать список обратных вызовов, которые будут срабатывать во время обучения.



history = model.fit(partial\_x\_train,

partial\_y\_train,

epochs

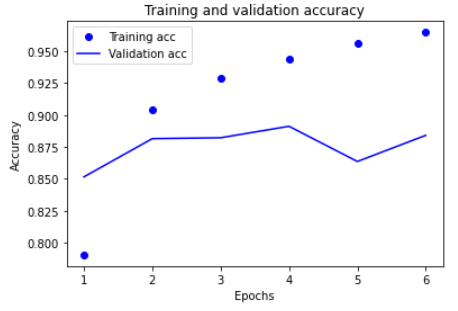
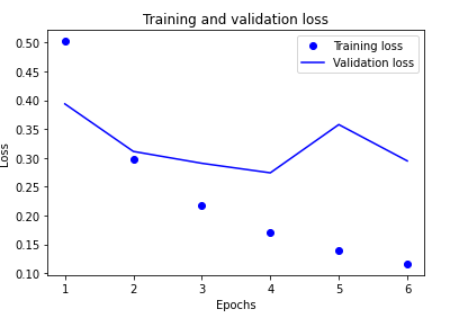
batch\_size=512,

validation\_data=(x\_val, y\_val),

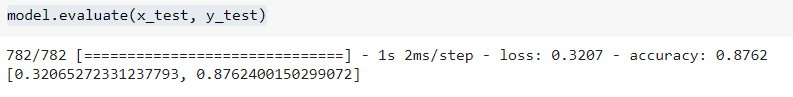
callbacks = callbacks\_list)

Как видно, из указанных 20 эпох, модель обучалась только 6.

Если теперь вывести графики ошибки и точности, то видно, что переобучения как такового не наблюдается.



Протестировав модель на тестовых данных, заметим, что значение ошибки равно 0.32, а значение точности равно 0.876 (87.6%). То есть полученная модель лучше, чем та, которая была раньше. Старое значение ошибки 0.78, точность была равна 0.856 (85.6%).



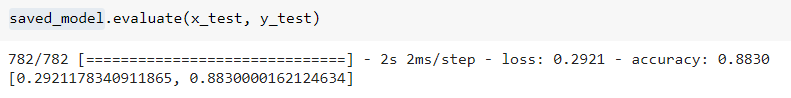
model.evaluate(x\_test, y\_test)

Также по графикам можно заметить, что на 4 эпохе наблюдался минимум ошибки и максимум точности. На этой эпохе модель должны была сохраниться. Загрузим ее.



saved\_model = keras.models.load\_model(‘my\_model.h5’)

И проверим ее на тестовых значениях.



saved\_model.evaluate(x\_test, y\_test)

Видим, что значение ошибки еще меньше (0.292), а итоговая точность классификации 0.883 (88.3%). То есть таким образом мы смогли получить наилучшую модуль без слишком долгого обучения.

Написание пользовательского обратного вызова

Иногда может потребоваться выполнять какие-то особенные действия во время обучения, который отсутствуют среди стандартных. Это может быть замена весов модели определенным принципом, построение графиков на определенных эпохах, и.т.д.

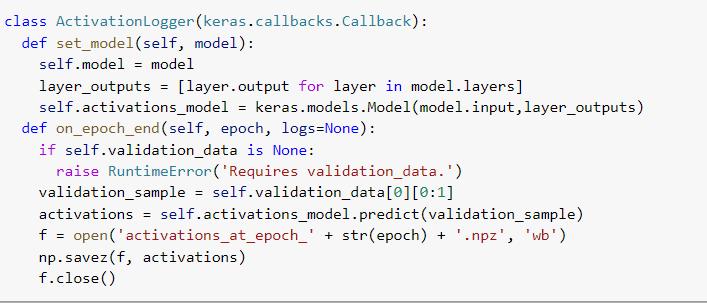
Пользовательские вызовы можно реализовать черзе наследование от класса *keras.callbacks.Callback*. Данный класс является базовым для всех обратных вызовов.

Для определения функционала, который будет выполняться в определенный момент обучения, необходимо переопределить одну или более функций:

* *on\_epoch\_begin* – вызывается в начале эпохи
* *on\_epoch\_end*– вызывается в конце эпохи
* *on\_batch\_begin*– вызываетсяпри передачи очередного пакета
* *on\_batch\_end*– вызываетсяпосле изменения на очередном пакете
* *on\_train\_begin*–вызывается в начале обучения
* *on\_train\_end*–вызывается в конце обучения

Все эти функции вызываются с номером пакета, эпохи (если они имеются) и параметром *logs*, который хранит текущую информацию об обучении в виде словаря.

Создадим пользовательский обратный вызов, который будет сохранять все активации слоев после каждой эпохи. Активация будет рассчитываться на первом объекте из контрольной выборки.



class ActivationLogger(keras.callbacks.Callback):

def set\_model(self, model):

self.model = model

layers\_outputs = [layer.output for layer in model.layers]

self.activations\_model = keras.model.Model(model.input,layers\_outputs)

def on\_epoch\_end(self, epoch, logs=None):

if self.validation\_data is None

raise RuntimeError(‘Requires validation\_data.’)

validation\_sample = self.validation\_data[0][0:1]

activations = self.activations\_model.predict(validation\_sample)

f = open(‘activations\_at\_epoch\_’ + str(epoch) + ‘.npz’, ‘wb’)

np.savez(f, activations)

f.close()

В данном примере класс имеет два метода. Первый метод *set\_model* является обязательным, так как через обратный вызов имеет доступ к самой модели. Второй метод *on\_epoch\_end* описывает логику того, что будет происходить в конце эпохи. В нашем случае сначала идет проверка того, что контрольные данные вообще есть, затем получаем активации после каждого слоя, и сохраняем их в файле, в названии которого указывается эпоха. После того, как обратный вызов написан его можно использовать во время обучения.

## ЗАДАНИЯ

* 1. Библиотека NumPy

Задача 1

Дан список чисел, каждое число имеет значение от 0 до 255. Необходимо преобразовать список в матрицу бинарных представлений.

**Пример**

*Входные данные:*

1 10 255 0 151

*Выходные данные:*

0 0 0 0 0 0 0 1

0 0 0 0 1 0 1 0

1 1 1 1 1 1 1 1

0 0 0 0 0 0 0 0

1 0 0 1 0 1 1 1

Задача 2

На выход подается 4 числа *n, m, a, b*. Необходимо сгенерировать матрицу размером *n*на *m*, заполненную значениями *a*и *b* в шахматном порядке.

**Пример**

*Входные данные:*

3 4 9 11

*Выходные данные:*

9 11 9 11

11 9 11 9

9 11 9 11

Задача 3

Дана матрица *n* на *m*. Необходимо получить список всех уникальных строчек матрицы.

**Пример**

*Входные данные:*

4 3

1 1 0 1 0 1 1 0 1 1 1 0

‘’’

Соответствует матрице

1 1 0

1 0 1

1 0 1

1 1 0

‘’’

*Выходные данные:*

1 0 1

1 1 0

Задача 4

Дана матрица *n* на *m*. Необходимо получить матрицу со стандартизованными значениями.

**Пример**

*Выходные данные:*

2 2

1 2 3 4

*Выходные данные:*

-1.34164079 -0.4472136

0.4472136 1.34164079

Задача 5

Дано *p* квадратных матриц размером n и *p*векторов длинной *n*. Необходимо рассчитать сумму векторов, получаемых после произведения матрицы с индексом *i* на вектор с соответствующим индексом *i*(*I*= 1, 2, … *p*).

**Пример**

*Входные данные:*

3 4

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 30 31

32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47

0 1 2 3

4 5 6 7

8 9 10 11

*Выходные данные:*

1682 1946 2210 2475

**Примечание**

Сначала идут два числа *p* и *n*. Затем в следующих *p* строчка идет список чисел длинной *n\*n*, представляющий матрицу выписанную построчно. После этого в следующих *p* строчках идут списки из *n* чисел, соответствующие векторам.

2.2. Построение простой нейронной сети

Необходимо реализовать нейронные сети следующим образом:

1. С использованием библиотеки Keras
2. Как функцию, использующую тензорные операции из библиотеки NumPy
3. Как функцию, выполняющая поэлементные операции над тензорами

Затем обучить модель реализованную через KerasAPI, получить значение всех весовых матриц, и сравнить результаты с двумя оставшимся функциями, передав туда данные и весовые матрицы обученной модели.

Для получения весовых матриц каждого слоя можно использовать функцию *get\_weights()*. Данная функция возвращает список из матриц. Первая матрица соответствует весовым коэффициентам между слоями, вторая матрица хранит веса для нейронов сдвига.

При построении модели рекомендуется ограничится 2-3 скрытыми слоями, с не более чем 16 нейронами на них. В качестве функций на скрытых слоя используйте RElu (*max(0, x)*), а на выходном слое сигмоид (*1 / (1 + exp(-x)*).

Генерацию данных проводить согласно представленным в таблице логическим функций.

|  |  |
| --- | --- |
| Вариант | Фукнция |
| 1 | (a and b) or (a and c) |
| 2 | (a or b) xornot(b and c) |
| 3 | (a and b) or c |
| 4 | (a or b) and (b or c) |
| 5 | (a xor b) and (b xor c) |
| 6 | (a and not b) or (c xor b) |
| 7 | (a or b) and (a xor not b) |
| 8 | (a and c and b) xor (a or not b) |

*Примечание: так как множество всех наблюдений ограничено, то обучение проводить можно на всем датасете без контрольной выборки*.

* 1. Построение бинарного классификатора

Необходимо реализовать нейронную сеть, которая будет проводить бинарную классификацию сгенерированных точек в декартовом двумерном пространстве.

Код, который необходимо дополнить:

import numpy as np

from tensorflow.keras import layers

from tensorflow.keras import models

def genData(size=500):

    #Функцию выбрать в зависимости от варианта

(train\_data, train\_label), (test\_data, test\_label) = genData()

#В данном месте необходимо создать модель и обучить ее

#Получение и вывод результатов на тестовом наборе

results = model.evaluate(test\_data, test\_label)

Функции для генерации данных находятся в приложении А.

Функция для проверки результатов:

import matplotlib.pyplot as plt

import matplotlib.colors as mclr

def drawResults(data, label, prediction):

    p\_label = np.array([round(x[0]) for x in prediction])

    plt.scatter(data[:, 0], data[:, 1], s=30, c=label[:, 0], cmap=mclr.ListedColormap(['red', 'blue']))

    plt.scatter(data[:, 0], data[:, 1], s=10, c=p\_label, cmap=mclr.ListedColormap(['red', 'blue']))

    plt.grid()

    plt.show()

#Вывод результатов бинарной классификации

all\_data = np.vstack((train\_data, test\_data))

all\_label = np.vstack((train\_label, test\_label))

pred = model.predict(all\_data)

drawResults(all\_data, all\_label, pred)

* 1. Построение сверточной нейронной сети

Необходимо построить сверточную нейронную сеть, которая будет классифицировать черно-белые изображения с простыми геометрическими фигурами на них.

Для генерации данных необходимо вызвать функцию gen\_data, которая возвращает два тензора:

* Тензор с изображениями ранга 3
* Тензор с метками классов

Варианты для генерации изображений находятся в приложении Б.

Обратите внимание:

* Выборки не перемешаны, то есть наблюдения классов идут по порядку
* Метки классовописываютсястрокой
* Выборка изначально не разбита на обучающую, контрольную и тестовую
  1. Построение рекуррентной нейронной сети

Необходимо построить рекуррентную нейронную сеть, которая будет прогнозировать значение некоторого периодического сигнала. Варианты с генерацией последовательностей находятся в приложении В.

Так как генерирующие функции возвращают сигнал в виде набора пар точек, то необходимо:

* Преобразовать последовательность в датасет, который можно подавать на вход нейронной сети.
* Разбить датасет на обучающую, контрольную и тестовую выборку

Пример преобразования последовательности, представленной зашумленным синусом, к виду подходящему для нейронной сети виду:

def gen\_sequence(seq\_len = 1000):

    seq = [math.sin(i/10) + random.normalvariate(0, 0.09) for i in range(seq\_len)]

    return np.array(seq)

def gen\_data\_from\_sequence(seq\_len = 1000, lookback = 10):

    seq = gen\_sequence(seq\_len)

    past = np.array([[[seq[j]] for j in range(i,i+lookback)] for i in range(len(seq) - lookback)])

    future = np.array([[seq[i]] for i in range(lookback,len(seq))])

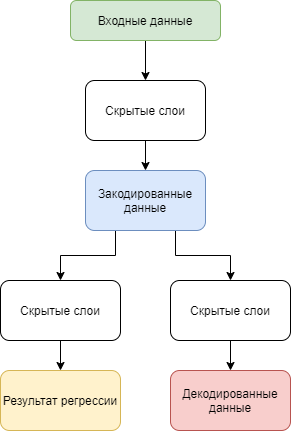
    return (past, future)

data, labels = gen\_data\_from\_sequence()

В примере, функция *gen\_sequence*генерирует непрерывный сигнал. Функция*gen\_data\_from\_sequence*генерируетсигналдлинной равной *seq\_len*, и разбивает так, что по количеству точек равным *lookback* будет предсказываться 1 точка. Затем возвращает два тензора – входные и выходные данные. В примере размер входных данных равен (990, 10, 1), 990 наблюдений длинной 10 с количеством признаков равным 1. Размер выходных данных равен (990, 1).

* 1. Построение автокодировщика и регрессионной модели

Необходимо, с использованием FunctionalAPI, построить модель, которая будет совмещать в себе автокодировщик и регрессионую модель. Схема архитектуры представлена на следующем рисунке:



Для генерации данных необходимо сгенерировать набор случайных чисел *X*по нормальному закону, затем рассчитать значений шести признаков и одно предсказываемое значение. После этого к полученным признакам нужно добавить белый шум*e*. Параметры для генерации данных представлены в таблице. Белый шум генерируется по закону N(0,0.3).

|  |  |
| --- | --- |
| Вариант | Распределение для  генерации данных |
| 1 | N(3,10) |
| 2 | N(0,10) |
| 3 | N(0,10) |

Формулы, по которым рассчитывать признаки представлены в таблице:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Вар. | Признаки | | | | | | Метка |
| 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | X^2+e | sin(X/2)+e | cos(2x)+e | X-3+e | -X+e | |X|+e | (X^3)/4+e |
| 2 | -X^3+e | ln(|X|)+e | sin(X-PI\4)+e | -X^3+e | -X/4+e | X^2+X+e | sin(3X)+e |
| 3 | cos(X)+e | -X+e | sin(X)\*X+e | sqrt(|X|)+e | X^2+e | -|X|+4 | X-(X^2)/5+e |

После обучения, необходимо убедиться, что закодированные данные не имеют пустых столбцов (все значения равны 0) и в одном столбце все значения не повторяются. Также, декодированные данные должны быть близки по значениям к исходным.

* 1. Написание обратных вызовов

Необходимо реализовать обратный вызов согласно варианту. Для проверки обратного вызова можно использовать модели уже рассматриваемые в рамках раздела 2 или полученные во время выполнения заданий.

Вариант 1

Сохранение трех наилучших моделей. Название файлов с моделями должны иметь следующий вид:<текущая дата>\_<префикс, задаваемый пользователем>\_<номер модели>

Вариант 2

Построение и сохранение карты признаков на заданных пользователем эпохах. Карта признаков - ядро свертки, представленное в виде изображения. Название карты признака должно иметь: вид <номер слоя>\_<номер ядра в слое>\_<номер эпохи>

Вариант 3

Построение и сохранение столбчатых диаграмм, отображающихсреднюю точность классификации наблюдений для каждого класса на заданных пользователем эпохах. Названия файлов с изображениями должны иметь название, соответствующее эпохе, на которой была построена диаграмма.

Вариант 4

Сохранение моделей на заданных пользователем эпохах. Название файлов с моделями должна иметь следующий вид <текущая дата>\_<префикс, задаваемый пользователем>\_<номер эпохи>

Вариант 5

На каждой эпохе расчет количества наблюдений, для которых точность классификации меньше 90%. В конце обучения построения и сохранение графика изменения рассчитываемой величины в зависимости от эпохи.

Вариант 6

Построение и сохранение таблицы со следующими данными: номер эпохи, номер наблюдения с наименьшей точностью классификации на заданной эпохе, к какому классу принадлежит наблюдение, точность классификации, значение ошибки.

Каждая строчка должна рассчитываться с заданным пользователем интервалом начиная с 0 эпохи, а также на самой последней.

## ЛАБОРАТОРНЫЕ РАБОТЫ

## Многоклассовая классификация цветов

## Цель

Реализовать классификацию сортов растения ирис (IrisSetosa - 0, IrisVersicolour - 1, IrisVirginica - 2) по четырем признакам: размерам пестиков и тычинок его цветков.

## Задачи

* Ознакомиться с задачей классификации
* Загрузить данные
* Создать модель ИНС в Keras
* Настроить параметры обучения
* Обучить и оценить модель

## Выполнение работы

Задача многоклассовой классификации является одним из основных видов задач, для решения которых применяются нейронные сети. В листинге 1 представлен пример данных.

Листинг 1 - Пример данных

|  |
| --- |
| 5.1,3.5,1.4,0.2,Iris-setosa  4.9,3.0,1.4,0.2,Iris-setosa  4.7,3.2,1.3,0.2,Iris-setosa  4.6,3.1,1.5,0.2,Iris-setosa  5.0,3.6,1.4,0.2,Iris-setosa |

Набор данных доступен для скачивания http://archive.ics.uci.edu/ml/machine-learning-databases/iris/ (файл iris.data). Скачанный файл необходимо переименовать в “iris.csv” и поместить в директорию своего проекта.

Импортируем необходимые для работы классы и функции. Кроме Keras понадобится Pandas для загрузки данных и scikit-learn для подготовки данных и оценки модели (листинг 2).

Листинг 2 - Подключение модулей

|  |
| --- |
| import pandas  from tensorflow.keras.layers import Dense  from tensorflow.keras.models import Sequential  from tensorflow.keras.utils import to\_categorical  from sklearn.preprocessing import LabelEncoder |

Набор данных загружается напрямую с помощью pandas. Затем необходимо разделить атрибуты (столбцы) на входные данные(X) и выходные данные(Y) (листинг 3).

Листинг 3 - Загрузка данных

|  |
| --- |
| dataframe = pandas.read\_csv("iris.csv", header=None)  dataset = dataframe.values  X = dataset[:,0:4].astype(float)  Y = dataset[:,4] |

При решении задач многоклассовой классификации хорошей практикой является преобразование выходных атрибутов из вектора в матрицу к виду представленных в листинге 4.

Листинг 4 - Представление данных

|  |
| --- |
| Iris-setosa, Iris-versicolor, Iris-virginica  1, 0, 0  0, 1, 0  0, 0, 1 |

Для этого необходимо использовать функцию to\_categorical()(Листинг 5)

Листинг 5 - Переход от текстовых меток к категориальному вектору

|  |
| --- |
| encoder = LabelEncoder()  encoder.fit(Y)  encoded\_Y = encoder.transform(Y)  dummy\_y = to\_categorical(encoded\_Y) |

Теперь можно задать базовую архитектуру сети (листинг 6)

Листинг 6 - Создание модели

|  |
| --- |
| model = Sequential()  model.add(Dense(4, activation=’relu’))  model.add(Dense(3, activation=’softmax’)) |

Основным строительным блоком нейронных сетей является слой (или уровень), модуль обработки данных, который можно рассматривать как фильтр для данных. Он принимает некоторые данные и выводит их в более полезной форме. В частности, слои извлекают представления из подаваемых в них данных, которые, как мы надеемся, будут иметь больше смысла для решаемой задачи. Фактически методика глубокого обучения заключается в объединении простых слоев, реализующих некоторую форму поэтапной очистки данных. Модель глубокого обучения можно сравнить с ситом, состоящим из последовательности фильтров все более тонкой очистки данных — слоев.

В данном случае наша сеть состоит из последовательности двух слоев Dense, которые являются тесно связанными (их еще называют полносвязными) нейронными слоями. Второй (и последний) слой — это 3-переменный слой потерь (softmaxlayer), возвращающий массив с 3 оценками вероятностей (в сумме дающих 1). Каждая оценка определяет вероятность принадлежности текущего изображения к одному из 3 классов цветов.

Чтобы подготовить сеть к обучению, нужно настроить еще три параметра для этапа компиляции:

1. функцию потерь, которая определяет, как сеть должна оценивать качество своей работы на обучающих данных и, соответственно, как корректировать ее в правильном направлении;
2. оптимизатор — механизм, с помощью которого сеть будет обновлять себя, опираясь на наблюдаемые данные и функцию потерь;
3. метрики для мониторинга на этапах обучения и тестирования — здесь нас будет интересовать только точность (доля правильно классифицированных изображений).

Листинг 6 - Инициализация параметров обучения

|  |
| --- |
| model.compile(optimizer='adam',loss='categorical\_crossentropy', metrics=['accuracy']) |

Теперь можно начинать обучение сети (листинг 7), для чего в случае использования библиотеки Keras достаточно вызвать метод fit сети — он пытается адаптировать (fit) модель под обучающие данные.

Листинг 7 - Обучение сети

|  |
| --- |
| model.fit(X, dummy\_y, epochs=75, batch\_size=10, validation\_split=0.1) |

В процессе обучения отображаются четыре величины: потери сети на обучающих данных и точность сети на обучающих данных, а также потери и точность на данных, не участвовавших в обучении.

## Требования

1. Изучить различные архитектуры ИНС (Разное кол-во слоев, разное кол-во нейронов на слоях)
2. Изучить обучение при различных параметрах обучения (параметры ф-цийfit)
3. Построить графики ошибок и точности в ходе обучения
4. Выбрать наилучшую модель.

## Бинарная классификация отраженных сигналов радара

## Цель

Реализовать классификацию между камнями (R) и металлическими цилиндрами (M) на основе данных об отражении сигналов радара от поверхностей.

60 входных значений показывают силу отражаемого сигнала под определенным углом. Входные данные нормализованы и находятся в промежутке от 0 до 1.

## Задачи

* Ознакомиться с задачей бинарной классификации
* Загрузить данные
* Создать модель ИНС в tf.Keras
* Настроить параметры обучения
* Обучить и оценить модель
* Изменить модель и провести сравнение. Объяснить результаты

## Выполнение работы

Ниже представлены первые 2 строки из набора данных.

Листинг 1:

|  |
| --- |
| 0.0200,0.0371,0.0428,0.0207,0.0954,0.0986,0.1539,0.1601,0.3109,0.2111,0.1609,0.1582,0.2238,0.0645,0.0660,0.2273,0.3100,0.2999,0.5078,0.4797,0.5783,0.5071,0.4328,0.5550,0.6711,0.6415,0.7104,0.8080,0.6791,0.3857,0.1307,0.2604,0.5121,0.7547,0.8537,0.8507,0.6692,0.6097,0.4943,0.2744,0.0510,0.2834,0.2825,0.4256,0.2641,0.1386,0.1051,0.1343,0.0383,0.0324,0.0232,0.0027,0.0065,0.0159,0.0072,0.0167,0.0180,0.0084,0.0090,0.0032,R  0.0453,0.0523,0.0843,0.0689,0.1183,0.2583,0.2156,0.3481,0.3337,0.2872,0.4918,0.6552,0.6919,0.7797,0.7464,0.9444,1.0000,0.8874,0.8024,0.7818,0.5212,0.4052,0.3957,0.3914,0.3250,0.3200,0.3271,0.2767,0.4423,0.2028,0.3788,0.2947,0.1984,0.2341,0.1306,0.4182,0.3835,0.1057,0.1840,0.1970,0.1674,0.0583,0.1401,0.1628,0.0621,0.0203,0.0530,0.0742,0.0409,0.0061,0.0125,0.0084,0.0089,0.0048,0.0094,0.0191,0.0140,0.0049,0.0052,0.0044,R |

Набор данных доступен для скачивания https://archive.ics.uci.edu/ml/machine-learning-databases/undocumented/connectionist-bench/sonar/ (файл sonar.all-data). Скачанный файл необходимо переименовать в “sonar.csv” и поместить в директорию своего проекта.

Импортируем необходимые для работы классы и функции. Кроме Keras понадобится Pandas для загрузки данных и scikit-learn для подготовки данных и оценки модели.

Листинг 2:

|  |
| --- |
| import pandas  from tensorflow.keras.layers import Dense  from tensorflow.keras.models import Sequential  from tensorflow.keras.utils import to\_categorical  from sklearn.preprocessing import LabelEncoder |

Набор данных загружается напрямую с помощью pandas. Затем необходимо разделить атрибуты (столбцы) на 60 входных параметров (X) и 1 выходной (Y).

Листинг 4:

|  |
| --- |
| dataframe = pandas.read\_csv("sonar.csv", header=None)  dataset = dataframe.values  X = dataset[:,0:60].astype(float)  Y = dataset[:,60] |

Выходные параметры представлены строками (“R” и “M”), которые необходимо перевести в целочисленные значения 0 и 1 соответственно. Для этого применяется LabelEncoder из scikit-learn.

Листинг 5:

|  |
| --- |
| encoder = LabelEncoder()  encoder.fit(Y)  encoded\_Y = encoder.transform(Y) |

Теперь можно задать базовую архитектуру сети.

Листинг 6:

|  |
| --- |
| model = Sequential()  model.add(Dense(60, input\_dim=60, init=’normal’, activation=’relu’))  model.add(Dense(1, init=’normal’, activation=’sigmoid’)) |

Чтобы подготовить сеть к обучению, нужно настроить еще три параметра для этапа компиляции:

1. функцию потерь, которая определяет, как сеть должна оценивать качество своей работы на обучающих данных и, соответственно, как корректировать ее в правильном направлении; Для задач бинарной классификации применяется функция binarycrossentropy.
2. оптимизатор — механизм, с помощью которого сеть будет обновлять себя, опираясь на наблюдаемые данные и функцию потерь;
3. метрики для мониторинга на этапах обучения и тестирования — здесь нас будет интересовать только точность (доля правильно классифицированных изображений).

Листинг 7:

|  |
| --- |
| model.compile(optimizer='adam',loss='binary crossentropy', metrics=['accuracy']) |

Теперь можно начинать обучение сети, для чего в случае использования библиотеки Keras достаточно вызвать метод fit сети — он пытается адаптировать (fit) модель под обучающие данные.

Листинг 8:

|  |
| --- |
| model.fit(X, encoded\_Y, epochs=100, batch\_size=10, validation\_split=0.1) |

В процессе обучения отображаются четыре величины: потери сети на обучающих данных и точность сети на обучающих данных, а также потери и точность на данных, не участвовавших в обучении.

В представленном наборе данных присутствует некоторая избыточность, т.к. с разных углов описывается один и тот же сигнал. Вероятно, что некоторые углы отражения сигнала имеют большую значимость, чем другие. Изменение количества нейронов во входном слое напрямую влияет на количество признаков, с которыми будет работать нейронная сеть.

* Необходимо уменьшить размер входного слоя в два раза и сравнить с результатами первоначальной архитектуры.

Нейронная сеть с несколькими слоями позволяет находить закономерности не только во входных данных, но и в их комбинации. Также, дополнительные слои позволяют ввести нелинейность в сеть, что позволяет получать более высокую точность.

* Необходимо добавить промежуточный (скрытый) слой Dense в архитектуру сети с 15 нейронами и проанализировать результаты.

## Требования

1. Изучить влияние кол-ва нейронов на слое на результат обучения модели.
2. Изучить влияние кол-ва слоев на результат обучения модели
3. Построить графики ошибки и точности в ходе обучения
4. Провести сравнение полученных сетей, объяснить результат

## Регрессионная модель изменения цен на дома в Бостоне

## Цель

реализовать предсказание медианной цены на дома в пригороде Бостона в середине 1970-х по таким данным, как уровень преступности, ставка местного имущественного налога и т. д.

Данный набор содержит относительно немного образцов данных: всего 506, разбитых на 404 обучающих и 102 контрольных образца. И каждый признак во входных данных (например, уровень преступности) имеет свой масштаб. Например, некоторые признаки являются пропорциями и имеют значения между 0 и 1, другие — между 1 и 12 и т. д.

*Не путайте регрессию с алгоритмом логистической регрессии. Как ни странно, логистическая регрессия не является регрессионным алгоритмом — это алгоритм классификации.*

## Задачи

* Ознакомиться с задачей регрессии
* Изучить отличие задачи регрессии от задачи классификации
* Создать модель
* Настроить параметры обучения
* Обучить и оценить модели
* Ознакомиться с перекрестной проверкой

## Выполнение работы

Набор данных присутствует в составе Keras.

Листинг 1:

|  |
| --- |
| import numpy as np  from tensorflow.keras.layers import Dense  from tensorflow.keras.models import Sequential  from tensorflow.keras.utils import to\_categorical  from tensorflow.keras.datasets import boston\_housing  (train\_data, train\_targets), (test\_data, test\_targets) = boston\_housing.load\_data()  print(train\_data.shape)  print(test\_data.shape)  print(test\_targets) |

404 обучающих и 102 контрольных образца, каждый с 13 числовыми признаками.

Цены в основном находятся в диапазоне от 10 000 до 50 000 долларов США.

Было бы проблематично передать в нейронную сеть значения, имеющие самые разные диапазоны. Сеть, конечно, сможет автоматически адаптироваться к таким разнородным данным, однако это усложнит обучение. На практике к таким данным принято применять нормализацию: для каждого признака во входных данных (столбца в матрице входных данных) из каждого значения вычитается среднее по этому признаку, и разность делится на стандартное отклонение, в результате признак центрируется по нулевому значению и имеет стандартное отклонение, равное единице. Такую нормализацию легко выполнить с помощью Numpy.

Листинг 2:

|  |
| --- |
| mean = train\_data.mean(axis=0)  train\_data -= mean  std = train\_data.std(axis=0)  train\_data /= std  test\_data -= mean  test\_data /= std |

Определим функцию build\_model():

Листинг 3:

|  |
| --- |
| def build\_model():  model = Sequential()  model.add(Dense(64, activation='relu', input\_shape=(train\_data.shape[1],)))  model.add(Dense(64, activation='relu'))  model.add(Dense(1))  model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])  return model |

Сеть заканчивается одномерным слоем, не имеющим функции активации (это линейный слой). Это типичная конфигурация для скалярной регрессии (целью которой является предсказание одного значения на непрерывной числовой прямой). Применение функции активации могло бы ограничить диапазон выходных значений: например, если в последнем слое применить функцию активации sigmoid, сеть обучилась бы предсказывать только значения из диапазона между 0 и 1.

В данном случае, с линейным последним слоем, сеть способна предсказывать значения из любого диапазона.

Обратите внимание на то, что сеть компилируется с функцией потерь mse — meansquarederror (среднеквадратичная ошибка), вычисляющей квадрат разности между предсказанными и целевыми значениями. Эта функция широко используется в задачах регрессии. Также добавлен новый параметр на этапе обучения: mae — meanabsoluteerror (средняя абсолютная ошибка). Это абсолютное значение разности между предсказанными и целевыми значениями. Например, значение MAE, равное 0,5, в этой задаче означает, что в среднем прогнозы отклоняются на 500 долларов США.

Чтобы оценить качество сети в ходе корректировки ее параметров (таких, как количество эпох обучения), можно разбить исходные данные на обучающий и проверочный наборы, как это делалось в предыдущих примерах. Однако так как у нас и без того небольшой набор данных, проверочный набор получился бы слишком маленьким (скажем, что-нибудь около 100 образцов). Как следствие, оценки при проверке могут сильно меняться в зависимости от того, какие данные попадут в проверочный и обучающий наборы: оценки при проверке могут иметь слишком большой разброс. Это не позволит надежно оценить качество модели.

Хорошей практикой в таких ситуациях является применение перекрестной проверки по K блокам (K-foldcross-validation). Суть ее заключается в разделении доступных данных на K блоков (обычно K = 4 или 5), создании K идентичных моделей и обучении каждой на K—1 блоках с оценкой по оставшимся блокам. По полученным K оценкам вычисляется среднее значение, которое принимается как оценка модели. В коде такая проверка реализуется достаточно просто.

Листинг 4:

|  |
| --- |
| k = 4  num\_val\_samples = len(train\_data) // k  num\_epochs = 100  all\_scores = []  for i in range(k):  print('processing fold #', i)  val\_data = train\_data[i \* num\_val\_samples: (i + 1) \* num\_val\_samples]  val\_targets = train\_targets[i \* num\_val\_samples: (i + 1) \* num\_val\_samples]  partial\_train\_data = np.concatenate([train\_data[:i \* num\_val\_samples], train\_data[(i + 1) \* num\_val\_samples:]],  axis=0)  partial\_train\_targets = np.concatenate(  [train\_targets[:i \* num\_val\_samples], train\_targets[(i + 1) \* num\_val\_samples:]], axis=0)  model = build\_model()  model.fit(partial\_train\_data, partial\_train\_targets, epochs=num\_epochs, batch\_size=1, verbose=0)  val\_mse, val\_mae = model.evaluate(val\_data, val\_targets, verbose=0)  all\_scores.append(val\_mae)  print(np.mean(all\_scores)) |

Разные прогоны действительно показывают разные оценки, от 2,6 до 3,2. Средняя (3,0) выглядит более достоверно, чем любая из оценок отдельных прогонов, — в этом главная ценность перекрестной проверки по K блокам. В данном случае средняя ошибка составила 3000 долларов, что довольно много, если вспомнить, что цены колеблются в диапазоне от 10 000 до 50 000 долларов.

* Необходимо уменьшить или увеличить количество эпох обучения и проанализировать полученные результаты

## Требования

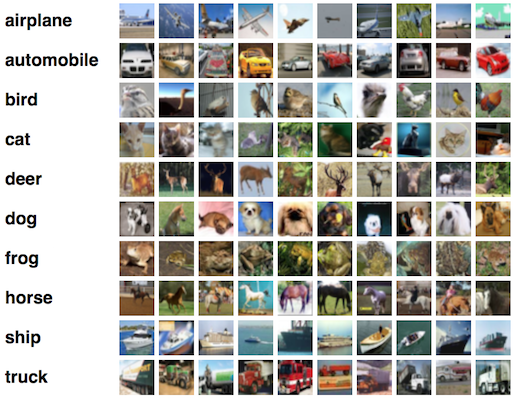
1. Объяснить различия задач классификации и регрессии
2. Изучить влияние кол-ва эпох на результат обучения модели
3. Исследовать влияние различных оптимизаторов, а также их параметров, на процесс обучения
4. Выявить точку переобучения
5. Построить графики ошибки и точности во время обучения для моделей

## Распознавание объектов на фотографиях

## Цель

Распознавание объектов на фотографиях (Object RecognitioninPhotographs)

CIFAR-10 (классификация небольших изображений по десяти классам: самолет, автомобиль, птица, кошка, олень, собака, лягушка, лошадь, корабль и грузовик).



## Задачи

* Ознакомиться со сверточными нейронными сетями
* Изучить построение модели в Keras в функциональном виде
* Изучить работу слоя разреживания (Dropout)

## Выполнение работы

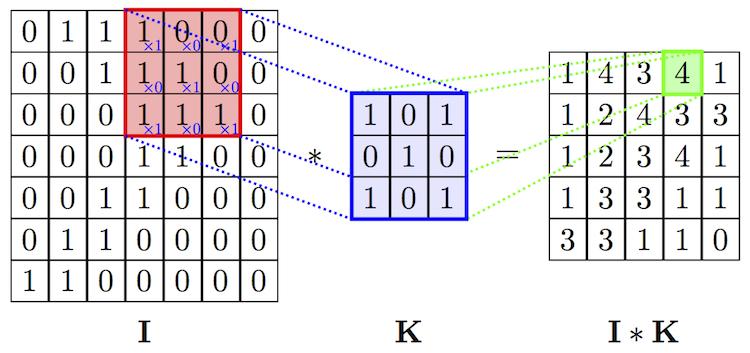
В лабораторной 4 архитектуру “многослойный перцептрон” (MLP) применили к MNIST. Но все же полносвязный перцептрон обычно не выбирают для задач, связанных с распознаванием изображений — в этом случае намного чаще пользуются преимуществами сверточных нейронных сетей (ConvolutionalNeural Networks, CNN).

Посмотрим, что происходит с количеством параметров (весов) в модели MLP, когда ей на вход поступают необработанные данные. Например, CIFAR-10 содержит 32 x 32 x 3 пикселей, и если мы будем считать каждый канал каждого пикселя независимым входным параметром для MLP, каждый нейрон в первом скрытом слое добавляет к модели около 3000 новых параметров. И с ростом размера изображений ситуация быстро выходит из-под контроля, причем происходит это намного раньше, чем изображения достигают того размера, с которыми обычно работают пользователи реальных приложений.

Одно из популярных решений — понижать разрешение изображений до той степени, когда MLP становится применим. Тем не менее, когда мы просто понижаем разрешение, мы рискуем потерять большое количество информации, и было бы здорово, если бы можно было осуществлять полезную первичную обработку информации еще до применения понижения качества, не вызывая при этом взрывного роста количества параметров модели.

Существует весьма эффективный способ решения этой задачи, который обращает в нашу пользу саму структуру изображения: предполагается, что пиксели, находящиеся близко друг к другу, теснее “взаимодействуют” при формировании интересующего нас признака, чем пиксели, расположенные в противоположных углах. Кроме того, если в процессе классификации изображения небольшая черта считается очень важной, не будет иметь значения, на каком участке изображения эта черта обнаружена.

Введем понятие оператора свертки. Имея двумерное изображение I и небольшую матрицу K размерности h\times w (так называемое ядро свертки), построенная таким образом, что графически кодирует какой-либо признак, мы вычисляем свернутое изображение I \* K, накладывая ядро на изображение всеми возможными способами и записывая сумму произведений элементов исходного изображения и ядра. Результат применения операции свертки (с двумя разными ядрами) к изображению с целью выделить контуры объекта:





Сверточные и субдискретизирующие слои

Оператор свертки составляет основу сверточного слоя (convolutionallayer) в CNN. Слой состоит из определенного количества ядер \vec{K} (с аддитивными составляющими смещения \vec{b} для каждого ядра) и вычисляет свертку выходного изображения предыдущего слоя с помощью каждого из ядер, каждый раз прибавляя составляющую смещения. В конце концов ко всему выходному изображению может быть применена функция активации \sigma. Обычно входной поток для сверточного слоя состоит из d каналов, например, red/green/blue для входного слоя, и в этом случае ядра тоже расширяют таким образом, чтобы они также состояли из d каналов; получается следующая формула для одного канала выходного изображения сверточного слоя, где K — ядро, а b — составляющая смещения:



Обратите внимание, что так как все, что мы здесь делаем — это сложение и масштабирование входных пикселей, ядра можно получить из имеющейся обучающей выборки методом градиентного спуска, аналогично вычислению весов в многослойном перцептроне (MLP). На самом деле MLP мог бы в совершенстве справиться с функциями сверточного слоя, но времени на обучение (как и обучающих данных) потребовалось бы намного больше.

Заметим также, что оператор свертки вовсе не ограничен двухмерными данными: большинство фреймворков глубокого обучения (включая Keras) предоставляют слои для одномерной или трехмерной свертки прямо “из коробки”.

Стоит также отметить, что, хотя сверточный слой сокращает количество параметров по сравнению с полносвязным слоем, он использует больше гиперпараметров — параметров, выбираемых до начала обучения.

В частности, выбираются следующие гиперпараметры:

Глубина (depth) — сколько ядер и коэффициентов смещения будет задействовано в одном слое;

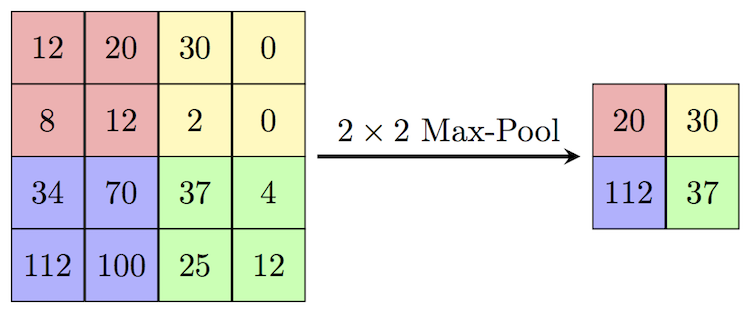
Высота (height) и ширина (width) каждого ядра;

Шаг (stride) — на сколько смещается ядро на каждом шаге при вычислении следующего пикселя результирующего изображения. Обычно его принимают равным 1, и чем больше его значение, тем меньше размер выходного изображения;

Отступ (padding): заметим, что свертка любым ядром размерности более, чем 1х1 уменьшит размер выходного изображения. Так как в общем случае желательно сохранять размер исходного изображения, рисунок дополняется нулями по краям.

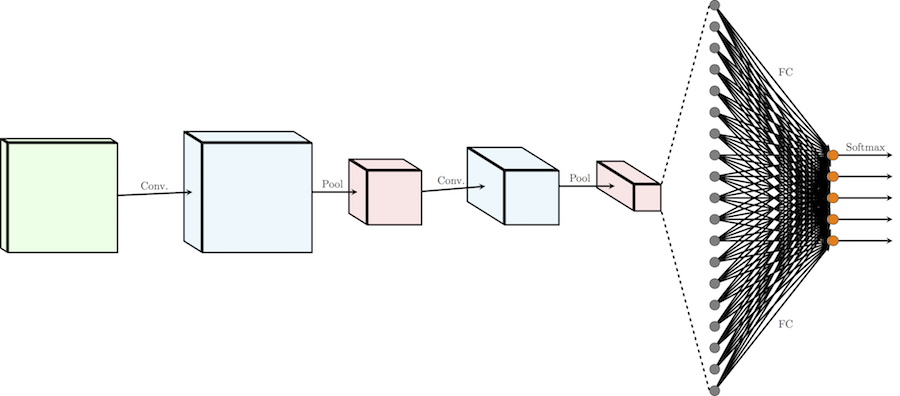
Операции свертки — не единственные операции в CNN (хотя существуют многообещающие исследования на тему “чисто-сверхточных” сетей); они чаще применяются для выделения наиболее полезных признаков перед субдискретизацией (downsampling) и последующей обработкой с помощью MLP.

Популярный способ субдискретизации изображения — слой подвыборки (также называемый слоем субдискретизации, по-английски downsampling или poolinglayer), который получает на вход маленькие отдельные фрагменты изображения (обычно 2х2) и объединяет каждый фрагмент в одно значение. Существует несколько возможных способов агрегации, наиболее часто из четырех пикселей выбирается максимальный. Этот способ схематически изображен ниже.



Итого: обычная CNN

Теперь, когда у нас есть все строительные блоки, давайте рассмотрим, как выглядит обычная CNN целиком.



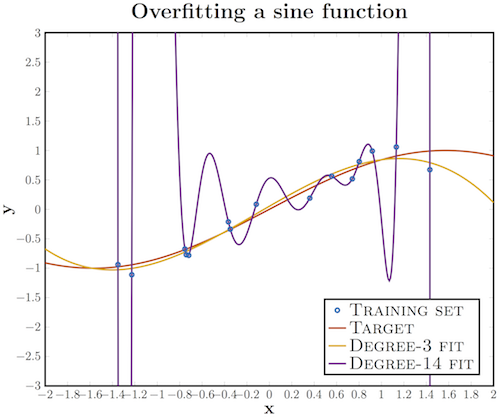
Обычную архитектуру CNN для распределения изображений по k классам можно разделить на две части: цепочка чередующихся слоев свертки/подвыборкиConv -> Pool (иногда с несколькими слоями свертки подряд) и несколько полносвязных слоев (принимающих каждый пиксель как независимое значение) с слоем softmax в качестве завершающего. Я не говорю здесь о функциях активации, чтобы наша схема стала проще, но не забывайте, что обычно после каждого сверточного или полносвязного слоя ко всем выходным значениям применяется функция активации, например, ReLU.

Один проход Conv -> Pool влияет на изображение следующим образом: он сокращает длину и ширину определенного канала, но увеличивает его значение (глубину).

Softmax и перекрестная энтропия более подробно рассмотрены на предыдущем уроке. Напомним, что функция softmax превращает вектор действительных чисел в вектор вероятностей (неотрицательные действительные числа, не превышающие 1). В нашем контексте выходные значения являются вероятностями попадания изображения в определённый класс. Минимизация потерь перекрестной энтропии обеспечивает уверенность в определении принадлежности изображения определенному классу, не принимая во внимание вероятность остальных классов, таким образом, для вероятностных задач softmax предпочтительней, чем, например, метод квадратичной ошибки.

Переобучение, регуляризация и dropout:

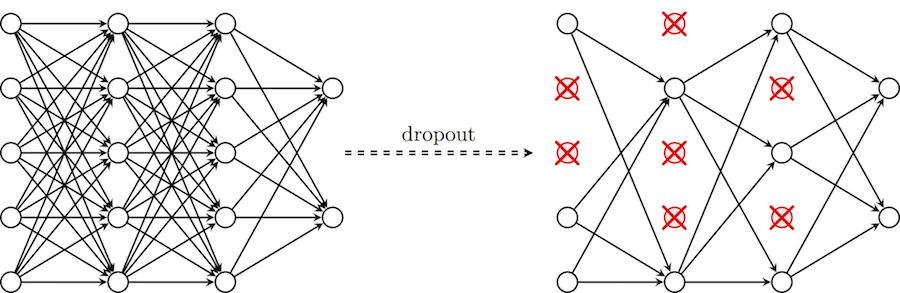
Переобучение — это излишне точное соответствие нейронной сети конкретному набору обучающих примеров, при котором сеть теряет способность к обобщению. Другими словами, наша модель могла выучить обучающее множество (вместе с шумом, который в нем присутствует), но она не смогла распознать скрытые процессы, которые это множество породили. В качестве примера рассмотрим задачу аппроксимации синусоиды с аддитивным шумом.



У нас есть обучающее множество (синие кружки), полученное из исходной кривой синуса, с некоторым количеством шума. Если мы приложим к этим данным график многочлена третьей степени, мы получим хорошую аппроксимацию исходной кривой. Кто-то возразит, что многочлен 14-й степени подошел бы лучше; действительно, так как у нас есть 15 точек, такая аппроксимация идеально описала бы обучающую выборку. Тем не менее, в этом случае введение дополнительных параметров в модель приводит к катастрофическим результатам: из-за того, что наша аппроксимация учитывает шумы, она не совпадает с исходной кривой нигде, кроме обучающих точек.

У глубоких сверточных нейронных сетей масса разнообразных параметров, особенно это касается полносвязных слоев. Переобучение может проявить себя в следующей форме: если у нас недостаточно обучающих примеров, маленькая группа нейронов может стать ответственной за большинство вычислений, а остальные нейроны станут избыточны; или наоборот, некоторые нейроны могут нанести ущерб производительности, при этом другие нейроны из их слоя не будут заниматься ничем, кроме исправления их ошибок.

Чтобы помочь нашей сети не утратить способности к обобщению в этих обстоятельствах, мы вводим приемы регуляризации: вместо сокращения количества параметров, мы накладываем ограничения на параметры модели во время обучения, не позволяя нейронам изучать шум обучающих данных. Здесь я опишу прием dropout, который сначала может показаться “черной магией”, но на деле помогает исключить ситуации, описанные выше. В частности, dropout с параметром p за одну итерацию обучения проходит по всем нейронам определенного слоя и с вероятностью p полностью исключает их из сети на время итерации. Это заставит сеть обрабатывать ошибки и не полагаться на существование определенного нейрона (или группы нейронов), а полагаться на “единое мнение” (consensus) нейронов внутри одного слоя. Это довольно простой метод, который эффективно борется с проблемой переобучения сам, без необходимости вводить другие регуляризаторы. Схема ниже иллюстрирует данный метод.



**Реализация сети**

В качестве практической части построим глубокую сверточную нейронную сеть и применим ее к классификации изображений из набора CIFAR-10.

Импорты:

|  |
| --- |
| from keras.datasets import cifar10  from keras.models import Model  from keras.layers import Input, Convolution2D, MaxPooling2D, Dense, Dropout, Flatten  from keras.utils import np\_utils  import numpy as np |

Как уже говорилось, обычно CNN использует больше гиперпараметров, чем MLP. В этом руководстве мы все еще будем использовать заранее известные “хорошие” значения, но не будем забывать, что в последующей лекции я расскажу, как их правильно выбирать.

Зададим следующие гиперпараметры:

batch\_size — количество обучающих образцов, обрабатываемых одновременно за одну итерацию алгоритма градиентного спуска;

num\_epochs — количество итераций обучающего алгоритма по всему обучающему множеству;

kernel\_size — размер ядра в сверточных слоях;

pool\_size — размер подвыборки в слоях подвыборки;

сonv\_depth — количество ядер в сверточных слоях;

drop\_prob (dropoutprobability) — мы будем применять dropout после каждого слоя подвыборки, а также после полносвязного слоя;

hidden\_size — количество нейронов в полносвязном слое MLP.

|  |
| --- |
| batch\_size = 32 # in each iteration, we consider 32 training examples at once  num\_epochs = 200 # we iterate 200 times over the entire training set  kernel\_size = 3 # we will use 3x3 kernels throughout  pool\_size = 2 # we will use 2x2 pooling throughout  conv\_depth\_1 = 32 # we will initially have 32 kernels per conv. layer...  conv\_depth\_2 = 64 # ...switching to 64 after the first pooling layer  drop\_prob\_1 = 0.25 # dropout after pooling with probability 0.25  drop\_prob\_2 = 0.5 # dropout in the dense layer with probability 0.5  hidden\_size = 512 # the dense layer will have 512 neurons |

Загрузка и первичная обработка CIFAR-10 осуществляется ровно так же, как и загрузка и обработка MNIST, где Keras выполняет все автоматически. Единственное отличие состоит в том, что теперь мы не рассматриваем каждый пиксель как независимое входное значение, и поэтому мы не переносим изображение в одномерное пространство. Мы снова преобразуем интенсивность пикселей так, чтобы она попадала в отрезок [0,1] и используем прямое кодирование для выходных значений.

Тем не менее, в этот раз этот этап будет выполнен для более общего случая, что позволит проще приспосабливаться к новым наборам данных: размер будет не жестко задан, а вычислен из размера набора данных, количество классов будет определено по количеству уникальных меток в обучающем множестве, а нормализация будет выполнена путем деления всех элементов на максимальное значение обучающего множества.

|  |
| --- |
| (X\_train, y\_train), (X\_test, y\_test) = cifar10.load\_data() # fetch CIFAR-10 data  num\_train, depth, height, width = X\_train.shape # there are 50000 training examples in CIFAR-10  num\_test = X\_test.shape[0] # there are 10000 test examples in CIFAR-10  num\_classes = np.unique(y\_train).shape[0] # there are 10 image classes  X\_train = X\_train.astype('float32')  X\_test = X\_test.astype('float32')  X\_train /= np.max(X\_train) # Normalise data to [0, 1] range  X\_test /= np.max(X\_train) # Normalise data to [0, 1] range  Y\_train = np\_utils.to\_categorical(y\_train, num\_classes) # One-hot encode the labels  Y\_test = np\_utils.to\_categorical(y\_test, num\_classes) # One-hot encode the labels |

Настало время моделирования! Наша сеть будет состоять из четырех слоев Convolution\_2D и слоев MaxPooling2D после второй и четвертой сверток. После первого слоя подвыборки мы удваиваем количество ядер (вместе с описанным выше принципом принесения высоты и ширины в жертву глубине). После этого выходное изображение слоя подвыборки трансформируется в одномерный вектор (слоем Flatten) и проходит два полносвязных слоя (Dense). На всех слоях, кроме выходного полносвязного слоя, используется функция активации ReLU, последний же слой использует softmax.

Для регуляризации нашей модели после каждого слоя подвыборки и первого полносвязного слоя применяется слой Dropout. Здесь Keras также выделяется на фоне остальных фреймворков: в нем есть внутренний флаг, который автоматически включает и выключает dropout, в зависимости от того, находится модель в фазе обучения или тестирования.

|  |
| --- |
| inp = Input(shape=(depth, height, width)) # N.B. depth goes first in Keras  # Conv [32] -> Conv [32] -> Pool (with dropout on the pooling layer)  conv\_1 = Convolution2D(conv\_depth\_1, kernel\_size, kernel\_size, border\_mode='same', activation='relu')(inp)  conv\_2 = Convolution2D(conv\_depth\_1, kernel\_size, kernel\_size, border\_mode='same', activation='relu')(conv\_1)  pool\_1 = MaxPooling2D(pool\_size=(pool\_size, pool\_size))(conv\_2)  drop\_1 = Dropout(drop\_prob\_1)(pool\_1)  # Conv [64] -> Conv [64] -> Pool (with dropout on the pooling layer)  conv\_3 = Convolution2D(conv\_depth\_2, kernel\_size, kernel\_size, border\_mode='same', activation='relu')(drop\_1)  conv\_4 = Convolution2D(conv\_depth\_2, kernel\_size, kernel\_size, border\_mode='same', activation='relu')(conv\_3)  pool\_2 = MaxPooling2D(pool\_size=(pool\_size, pool\_size))(conv\_4)  drop\_2 = Dropout(drop\_prob\_1)(pool\_2)  # Now flatten to 1D, apply Dense ->ReLU (with dropout) ->softmax  flat = Flatten()(drop\_2)  hidden = Dense(hidden\_size, activation='relu')(flat)  drop\_3 = Dropout(drop\_prob\_2)(hidden)  out = Dense(num\_classes, activation='softmax')(drop\_3)  model = Model(input=inp, output=out) # To define a model, just specify its input and output layers  model.compile(loss='categorical\_crossentropy', # using the cross-entropy loss function  optimizer='adam', # using the Adam optimiser  metrics=['accuracy']) # reporting the accuracy  model.fit(X\_train, Y\_train, # Train the model using the training set...  batch\_size=batch\_size, nb\_epoch=num\_epochs,  verbose=1, validation\_split=0.1) # ...holding out 10% of the data for validation  model.evaluate(X\_test, Y\_test, verbose=1) # Evaluate the trained model on the test set! |

## Требования

1. Построить и обучить сверточную нейронную сеть
2. Исследовать работу сеть без слоя Dropout
3. Исследовать работу сети при разных размерах ядра свертки

Приложение А – Функции генерации данных в декартовом пространстве

Вариант 1

def genData(size=500):

    data = np.random.rand(size, 2)\*2 - 1

    label = np.zeros([size, 1])

    for i, p in enumerate(data):

        if (p[0] + .5 >= p[1]) and (p[0] - 0.5 <= p[1]):

            label[i] = 1.

        else:

            label[i] = 0.

    div = round(size\*0.8)

    train\_data = data[:div, :]

    test\_data = data[div:, :]

    train\_label = label[:div, :]

    test\_label = label[div:, :]

    return (train\_data, train\_label), (test\_data, test\_label)

Вариант 2

def genData(size=500):

    data = np.random.rand(size, 2)\*2 - 1

    label = np.zeros([size, 1])

    for i, p in enumerate(data):

        if p[0]\*p[1] >= 0:

            label[i] = 1.

        else:

            label[i] = 0.

    div = round(size\*0.8)

    train\_data = data[:div, :]

    test\_data = data[div:, :]

    train\_label = label[:div, :]

    test\_label = label[div:, :]

    return (train\_data, train\_label), (test\_data, test\_label)

Вариант 3

def genData(size=500):

    data = np.random.rand(size, 2)\*2 - 1

    label = np.zeros([size, 1])

    for i, p in enumerate(data):

        if (p[0]+0.2)\*\*2 + (0.6\*p[1])\*\*2 >= 0.25:

            label[i] = 0.

        else:

            label[i] = 1.

    div = round(size\*0.8)

    train\_data = data[:div, :]

    test\_data = data[div:, :]

    train\_label = label[:div, :]

    test\_label = label[div:, :]

    return (train\_data, train\_label), (test\_data, test\_label)

Вариант 4

def genData(size=500):

    size1 = size//2

    size2 = size - size1

    t1 = np.random.rand(size1)

    x1 = np.asarray([i \* math.cos(i\*2\*math.pi) + (np.random.rand(1)-1)/2\*i for i in t1])

    y1 = np.asarray([i \* math.sin(i\*2\*math.pi) + (np.random.rand(1)-1)/2\*i for i in t1])

    data1 = np.hstack((x1, y1))

    label1 = np.zeros([size1, 1])

    div1 = round(size1\*0.8)

    t2 = np.random.rand(size2)

    x2 = np.asarray([-i \* math.cos(i\*2\*math.pi) + (np.random.rand(1)-1)/2\*i for i in t2])

    y2 = np.asarray([-i \* math.sin(i\*2\*math.pi) + (np.random.rand(1)-1)/2\*i for i in t2])

    data2 = np.hstack((x2, y2))

    label2 = np.ones([size2, 1])

    div2 = round(size2\*0.8)

    div = div1 + div2

    order = np.random.permutation(div)

    train\_data = np.vstack((data1[:div1], data2[:div2]))

    test\_data = np.vstack((data1[div1:], data2[div2:]))

    train\_label = np.vstack((label1[:div1], label2[:div2]))

    test\_label = np.vstack((label1[div1:], label2[div2:]))

    return (train\_data[order, :], train\_label[order, :]), (test\_data, test\_label)

Вариант 5

def genData(size=500):

    size1 = size//2

    size2 = size - size1

    x1 = np.random.rand(size1, 1)\*1.3 - 0.95

    y1 = np.asarray([3.5\*(i+0.2)\*\*2 - 0.8 + (np.random.rand(1)-0.5)/3 for i in x1])

    data1 = np.hstack((x1, y1))

    label1 = np.zeros([size1, 1])

    div1 = round(size1\*0.8)

    x2 = np.random.rand(size2, 1)\*1.3 - 0.35

    y2 = np.asarray([-3.5\*(i-0.2)\*\*2 + 0.8 + (np.random.rand(1)-0.5)/3 for i in x2])

    data2 = np.hstack((x2, y2))

    label2 = np.ones([size2, 1])

    div2 = round(size2\*0.8)

    div = div1 + div2

    order = np.random.permutation(div)

    train\_data = np.vstack((data1[:div1], data2[:div2]))

    test\_data = np.vstack((data1[div1:], data2[div2:]))

    train\_label = np.vstack((label1[:div1], label2[:div2]))

    test\_label = np.vstack((label1[div1:], label2[div2:]))

    return (train\_data[order, :], train\_label[order, :]), (test\_data, test\_label)

Приложение Б – Функции генерации изображений с простыми фигурами

Функции генерации данных основываются на следующих функциях. После выбора варианта рекомендуется выбрать только необходимые, и разместить их в файле gens.py.

import numpy as np

#Прямоугольник

def gen\_rect(size=50):

    img = np.zeros([size, size])

    x = np.random.randint(0, size)

    y = np.random.randint(0, size)

    w = np.random.randint(size // 10, size // 2)

    h = np.random.randint(size // 10, size // 2)

    img[x:x + w, y:y + h] = 1

    return img

#Круг

def gen\_circle(size=50):

    img = np.zeros([size, size])

    x = np.random.randint(0, size)

    y = np.random.randint(0, size)

    r = np.random.randint(size // 10, size // 3)

    for i in range(0, size):

        for j in range(0, size):

            if (i-x)\*\*2 + (j-y)\*\*2 <= r\*\*2:

                img[i, j] = 1

    return img

#Кольцо

def gen\_empty\_circle(size=50):

    img = np.zeros([size, size])

    x = np.random.randint(0, size)

    y = np.random.randint(0, size)

    r = np.random.randint(size // 10, size // 3)

    dr = np.random.randint(1, 10) + r

    for i in range(0, size):

        for j in range(0, size):

            if r\*\*2 <= (i - x) \*\* 2 + (j - y) \*\* 2 <= dr \*\* 2:

                img[i, j] = 1

    return img

#Горизонтальная линия

def gen\_h\_line(size=50):

    img = np.zeros([size, size])

    x = np.random.randint(10, size-10)

    y = np.random.randint(10, size-10)

    l = np.random.randint(size // 8, size // 2)

    w = 1

    img[x-w:x+w, y-l:y+l] = 1

    return img

#Вертикальная линия

def gen\_v\_line(size=50):

    img = np.zeros([size, size])

    x = np.random.randint(10, size - 10)

    y = np.random.randint(10, size - 10)

    l = np.random.randint(size // 8, size // 2)

    w = 1

    img[x - l:x + l, y - w:y + w] = 1

    return img

#Крест

def gen\_cross(size=50):

    img = np.zeros([size, size])

    x = np.random.randint(10, size - 10)

    y = np.random.randint(10, size - 10)

    l = np.random.randint(size // 8, size // 5)

    w = 1

    img[x-l:x+l, y-w:y+w] = 1

    img[x-w:x+w, y-l:y+l] = 1

    return img

Вариант 1

Классификация изображений с прямоугольником или кругом.

import gens

import numpy as np

def gen\_data(size=500, img\_size=50):

    c1 = size // 2

    c2 = size - c1

    label\_c1 = np.full([c1, 1], 'Square')

    data\_c1 = np.array([gens.gen\_rect(img\_size) for i in range(c1)])

    label\_c2 = np.full([c2, 1], 'Circle')

    data\_c2 = np.array([gens.gen\_circle(img\_size) for i in range(c2)])

    data = np.vstack((data\_c1, data\_c2))

    label = np.vstack((label\_c1, label\_c2))

    return data, label

Вариант 2

Классификация изображений с закрашенным или не закрашенным кругом

import gens

import numpy as np

def gen\_data(size=500, img\_size=50):

    c1 = size // 2

    c2 = size - c1

    label\_c1 = np.full([c1, 1], 'Empty')

    data\_c1 = np.array([gens.gen\_empty\_circle(img\_size) for i in range(c1)])

    label\_c2 = np.full([c2, 1], 'Not Empty')

    data\_c2 = np.array([gens.gen\_circle(img\_size) for i in range(c2)])

    data = np.vstack((data\_c1, data\_c2))

    label = np.vstack((label\_c1, label\_c2))

    return data, label

Вариант 3

Классификация изображений с горизонтальной или вертикальной линией

import gens

import numpy as np

def gen\_data(size=500, img\_size=50):

    c1 = size // 2

    c2 = size - c1

    label\_c1 = np.full([c1, 1], 'Horizontal')

    data\_c1 = np.array([gens.gen\_h\_line(img\_size) for i in range(c1)])

    label\_c2 = np.full([c2, 1], 'Vertical')

    data\_c2 = np.array([gens.gen\_v\_line(img\_size) for i in range(c2)])

    data = np.vstack((data\_c1, data\_c2))

    label = np.vstack((label\_c1, label\_c2))

    return data, label

Вариант 4

Классификация изображений с крестом или с линией (может быть горизонтальной или вертикальной)

import gens

import numpy as np

def random\_line(img\_size=50):

    p = np.random.random([1])

    if p < 0.5:

        return gens.gen\_v\_line(img\_size)

    else:

        return gens.gen\_h\_line(img\_size)

def gen\_data(size=500, img\_size=50):

    c1 = size // 2

    c2 = size - c1

    label\_c1 = np.full([c1, 1], 'Cross')

    data\_c1 = np.array([gens.gen\_cross(img\_size) for i in range(c1)])

    label\_c2 = np.full([c2, 1], 'Line')

    data\_c2 = np.array([random\_line(img\_size) for i in range(c2)])

    data = np.vstack((data\_c1, data\_c2))

    label = np.vstack((label\_c1, label\_c2))

    return data, label

Вариант 5

Классификация изображений по количеству крестов на них. Можетбыть 1, 2 или 3

import gens

import numpy as np

def gen\_k\_cross(k, img\_size=50):

    img = np.zeros([img\_size, img\_size])

    for i in range(k):

        img += gens.gen\_cross(img\_size)

    img[np.nonzero(img)] = 1

    return img

def gen\_data(size=500, img\_size=50):

    c1 = size // 3

    c2 = c1

    c3 = size - (c1 + c2)

    label\_c1 = np.full([c1, 1], 'One')

    data\_c1 = np.array([gen\_k\_cross(1, img\_size) for i in range(c1)])

    label\_c2 = np.full([c2, 1], 'Two')

    data\_c2 = np.array([gen\_k\_cross(2, img\_size) for i in range(c2)])

    label\_c3 = np.full([c3, 1], 'Three')

    data\_c3 = np.array([gen\_k\_cross(3, img\_size) for i in range(c3)])

    data = np.vstack((data\_c1, data\_c2, data\_c3))

    label = np.vstack((label\_c1, label\_c2, label\_c3))

    return data, label

Вариант 6

Классификация изображений по количеству линий на них. Можетбыть 1, 2 или 3

import gens

import numpy as np

def random\_line(img\_size=50):

    p = np.random.random([1])

    if p < 0.5:

        return gens.gen\_v\_line(img\_size)

    else:

        return gens.gen\_h\_line(img\_size)

def gen\_k\_lines(k, img\_size=50):

    img = np.zeros([img\_size, img\_size])

    for i in range(k):

        img += random\_line(img\_size)

    img[np.nonzero(img)] = 1

    return img

def gen\_data(size=500, img\_size=50):

    c1 = size // 3

    c2 = c1

    c3 = size - (c1 + c2)

    label\_c1 = np.full([c1, 1], 'One')

    data\_c1 = np.array([gen\_k\_lines(1, img\_size) for i in range(c1)])

    label\_c2 = np.full([c2, 1], 'Two')

    data\_c2 = np.array([gen\_k\_lines(2, img\_size) for i in range(c2)])

    label\_c3 = np.full([c3, 1], 'Three')

    data\_c3 = np.array([gen\_k\_lines(3, img\_size) for i in range(c3)])

    data = np.vstack((data\_c1, data\_c2, data\_c3))

    label = np.vstack((label\_c1, label\_c2, label\_c3))

    return data, label

ПриложениеВ – Функции генерации последовательностей

Вариант 1

import numpy as np

import random

def gen\_sequence(seq\_len = 1000):

    seq = [math.sin(i/5)/2 + math.cos(i/3)/2 + random.normalvariate(0, 0.04) for i in range(seq\_len)]

    return np.array(seq)

Вариант 2

import numpy as np

import random

def gen\_sequence(seq\_len = 1000):

    seq = [math.sin(i/5)\*math.sin(i/10+0.5) + random.normalvariate(0, 0.04) for i in range(seq\_len)]

    return np.array(seq)

Вариант 3

import numpy as np

import random

def func(i):

    i = i % 21

    return abs(i - 10)/4

def gen\_sequence(seq\_len = 1000):

    seq = [math.cos(i/4) + func(i) + random.normalvariate(0, 0.04) for i in range(seq\_len)]

    return np.array(seq)

Вариант 4

import numpy as np

import random

def func(i):

    i = i % 31

    return ((i-15) \*\* 2)/100 - 4

def gen\_sequence(seq\_len = 1000):

    seq = [math.cos(i/2) + func(i) + random.normalvariate(0, 0.04) for i in range(seq\_len)]

    return np.array(seq)

Вариант 5

import numpy as np

import random

def gen\_sequence(seq\_len = 1000):

    seq = [math.cos(i/5)/(1.1 + math.sin(i/6))/6 + random.normalvariate(0, 0.04) for i in range(seq\_len)]

    return np.array(seq)

Вариант 6

import numpy as np

import random

def func(i):

    return (i % 16 + 1) / 16

def gen\_sequence(seq\_len = 1000):

    seq = [math.cos(i/10) \* func(i) + random.normalvariate(0, 0.04) for i in range(seq\_len)]

    return np.array(seq)

Вариант 7

import numpy as np

import random

def func(i):

    i = abs((i/2 % 27) - 13)

    return (math.log(i+1))/2 - 0.5

def gen\_sequence(seq\_len = 1000):

    seq = [ func(i) + random.normalvariate(0, 0.04) for i in range(seq\_len)]

    return np.array(seq)

Вариант 8

import numpy as np

import random

def func(i):

    return ((i % 20) + 1) / 20

def gen\_sequence(seq\_len = 1000):

    seq = [abs(math.sin(i/20)) + func(i) + random.normalvariate(0, 0.04) for i in range(seq\_len)]

    return np.array(seq)

Оглавление

[ИСКУССТВЕННЫЕ НЕЙРОННЫЕ СЕТИ. ПРАКТИЧЕСКИЕ ЗАДАЧИ 1](#_Toc95398486)

[ПРЕДИСЛОВИЕ 3](#_Toc95398487)

[1. ПРАКТИЧЕСКИЕ ЗАНЯТИЯ 4](#_Toc95398488)

[1.1. Библиотека NumPy 4](#_Toc95398489)

[Скаляры (тензор 0 ранга) 4](#_Toc95398490)

[Векторы (тензор 1 ранга) 4](#_Toc95398491)

[Матрицы (тензоры 2 ранга) 5](#_Toc95398492)

[Тензоры третьего и высшего рангов 5](#_Toc95398493)

[Основные тензоры с данными 6](#_Toc95398494)

[Считывание тензоров из файла 7](#_Toc95398495)

[Функции для работы с библиотекой numpy 9](#_Toc95398496)

[1.2. Библиотека Keras 16](#_Toc95398497)

[Полносвязные нейронные сети 17](#_Toc95398498)

[Сверточные сети 27](#_Toc95398499)

[Рекуррентные сети 30](#_Toc95398500)

[Сети произвольной архитектуры 37](#_Toc95398501)

[Обратные вызовы в Keras 46](#_Toc95398502)

[2. ЗАДАНИЯ 51](#_Toc95398503)

[2.1. Библиотека NumPy 51](#_Toc95398504)

[Задача 1 51](#_Toc95398505)

[Задача 2 51](#_Toc95398506)

[Задача 3 51](#_Toc95398507)

[Задача 4 52](#_Toc95398508)

[Задача 5 52](#_Toc95398509)

[2.2. Построение простой нейронной сети 52](#_Toc95398510)

[2.3. Построение бинарного классификатора 53](#_Toc95398511)

[2.4. Построение сверточной нейронной сети 54](#_Toc95398512)

[2.5. Построение рекуррентной нейронной сети 54](#_Toc95398513)

[2.6. Построение автокодировщика и регрессионной модели 55](#_Toc95398514)

[2.7. Написание обратных вызовов 57](#_Toc95398515)

[Вариант 1 57](#_Toc95398516)

[Вариант 2 57](#_Toc95398517)

[Вариант 3 57](#_Toc95398518)

[Вариант 4 57](#_Toc95398519)

[Вариант 5 57](#_Toc95398520)

[Вариант 6 57](#_Toc95398521)

[3. ЛАБОРАТОРНЫЕ РАБОТЫ 58](#_Toc95398522)

[3.1. Многоклассовая классификация цветов 58](#_Toc95398523)

[Цель 58](#_Toc95398524)

[Задачи 58](#_Toc95398525)

[Выполнение работы 58](#_Toc95398526)

[Требования 61](#_Toc95398527)

[3.2. Бинарная классификация отраженных сигналов радара 61](#_Toc95398528)

[Цель 61](#_Toc95398529)

[Задачи 61](#_Toc95398530)

[Выполнение работы 61](#_Toc95398531)

[Требования 64](#_Toc95398532)

[3.3. Регрессионная модель изменения цен на дома в Бостоне 64](#_Toc95398533)

[Цель 64](#_Toc95398534)

[Задачи 64](#_Toc95398535)

[Выполнение работы 65](#_Toc95398536)

[Требования 67](#_Toc95398537)

[3.4. Распознавание объектов на фотографиях 68](#_Toc95398538)

[Цель 68](#_Toc95398539)

[Задачи 68](#_Toc95398540)

[Выполнение работы 68](#_Toc95398541)

[Требования 78](#_Toc95398542)

[Приложение А – Функции генерации данных в декартовом пространстве 79](#_Toc95398543)

[Вариант 1 79](#_Toc95398544)

[Вариант 2 79](#_Toc95398545)

[Вариант 3 79](#_Toc95398546)

[Вариант 4 80](#_Toc95398547)

[Вариант 5 80](#_Toc95398548)

[Приложение Б – Функции генерации изображений с простыми фигурами 82](#_Toc95398549)

[Вариант 1 83](#_Toc95398550)

[Вариант 2 83](#_Toc95398551)

[Вариант 3 84](#_Toc95398552)

[Вариант 4 84](#_Toc95398553)

[Вариант 5 85](#_Toc95398554)

[Вариант 6 85](#_Toc95398555)

[ПриложениеВ – Функции генерации последовательностей 87](#_Toc95398556)

[Вариант 1 87](#_Toc95398557)

[Вариант 2 87](#_Toc95398558)

[Вариант 3 87](#_Toc95398559)

[Вариант 4 87](#_Toc95398560)

[Вариант 5 87](#_Toc95398561)

[Вариант 6 88](#_Toc95398562)

[Вариант 7 88](#_Toc95398563)

[Вариант 8 88](#_Toc95398564)

Лисс Анна Александровна

Жангиров Тимур Рафаилович

Жукова Наталия Александровна

Субботин Алексей Николаевич

**Искусственные нейронные сети. Практические задачи**

Учебно-методическое пособие

Редактор

––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

Подписано в печать 00.00.0000. Формат 60×84 1/16.

Бумага офсетная. Печать цифровая. Печ. л. 5,0.

Гарнитура «Times New Roman». Тираж 83 экз. Заказ

––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

Издательство СПбГЭТУ «ЛЭТИ»

197376, С.-Петербург, ул. Проф. Попова, 5