

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт перспективной инженерии
Департамент цифровых, робототехнических систем и электроники

ОТЧЕТ
По лабораторной работе №9
Дисциплины «Основы нейронных сетей»

Выполнил:

Евдаков Евгений Владимирович

3 курс, группа ИВТ-б-о-22-1,

09.03.01 «Информатика и
вычислительная техника (профиль)
«Программное обеспечение средств
вычислительной
техники и автоматизированных
систем», очная форма обучения

(подпись)

Руководитель практики:

Воронкин Р. А., доцент департамента
цифровых и робототехнических
систем и электроники и института
перспективной инженерии

(подпись)

Ставрополь, 2025 г.

Тема: Архитектура автокодировщика.

Цель: изучить архитектуру и принципы работы автокодировщиков, разработать и обучить модели для решения задач восстановления изображений, удаления шума, обнаружения аномалий и генерации данных.

Ход работы:

Практика 1. Базовый блок Архитектура автокодировщика Autoencoder.

В данном практическом задании рассматриваются автокодировщики. В данной практики необходимо выполнить следующее:

- построить модель автокодировщика;
- обучить его на картинках цифр (база mnist);
- необходимо посмотреть, что получится, если подать на вход обученной модели другие картинки (изображения одежды, база fashion_mnist);
- необходимо попробовать угадать порог ошибки автоэнкодера, отделяющий нормальные данные (картинки цифр) от выбросов (картинки с одеждой);
- написать алгоритм, подбирающий оптимальный порог;
- построить и обучить модель с двумерным латентным пространством;
- визуализировать латентное пространство.

Для выполнения данной практики выполняется загрузка всех необходимых библиотек. После рассматривается архитектура автокодировщиков, где выполняется функция сборки сверточного автокодировщика (рис. 1).

```
[2] # Функция сборки сверточного автокодировщика
def create_base_ae(in_shape):
    # Энкодер, вход нейросети
    img_input = Input(in_shape)

    # Энкодер, первый блок
    # 1.1. Двумерная свертка
    x = Conv2D(32, (3, 3), padding='same', activation='relu')(img_input)
    # 1.2. Нормализация
    x = BatchNormalization()(x)
    # 1.3. Двумерная свертка
    x = Conv2D(32, (3, 3), padding='same', activation='relu')(x)
    # 1.4. Нормализация
    x = BatchNormalization()(x)
    # 1.5. Сокращение размерности и обобщение данных
    x = MaxPooling2D()(x)

    # Энкодер, второй блок
    # 2.1. Двумерная свертка
    x = Conv2D(64, (3, 3), padding='same', activation='relu')(x)
    # 2.2. Нормализация
    x = BatchNormalization()(x)
    # 2.3. Двумерная свертка
    x = Conv2D(64, (3, 3), padding='same', activation='relu')(x)
    # 2.4. Нормализация
    x = BatchNormalization()(x)

    # На выходе кодировщика и на входе декодировщика z - вектор латентного пространства
    # 2.5. Сокращение размерности и обобщение данных
    z = MaxPooling2D()(x)
```

Рисунок 1. Функция сборки сверточного автокодировщика

Далее для выполнения задания применяются сверточные функции, такие как функция последовательного вывода нескольких изображений для сравнения. Затем выполняется загрузка данных из базы с готовой разбивкой на train/test. После выполняются следующие действия: приведение всех картинок к нужной форме; приведение всех картинок к нужному типу; нормализация пикселей в диапазон $[0,1]$; сборка автокодировщика для формы картинок датасета; сводка архитектуры автокодировщика (выполнения этих частей кода можно просмотреть по ссылке в репозитории). Далее выполняется визуализация схемы архитектуры модели (рис. 2).

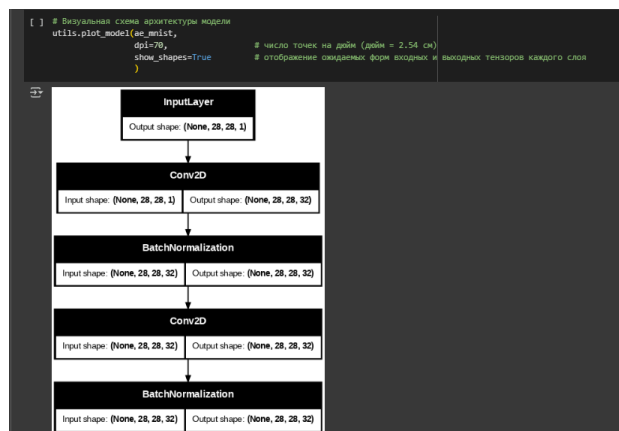


Рисунок 2. Визуализация схемы архитектурной модели

Затем выполняется обучение модели автокодировщика (рис. 3).

```

# Обучение модели автокодировщика, на входе и выходе одни и те же данные
history = ae_mnist.fit(x_train_mnist, x_train_mnist,
                      epochs=50,
                      batch_size=256,
                      validation_data = (x_test_mnist, x_test_mnist))
Epoch 5/50

```

Рисунок 3. Программа обучения автокодировщика

После смотрим на график процесса обучения модели (рис. 4).

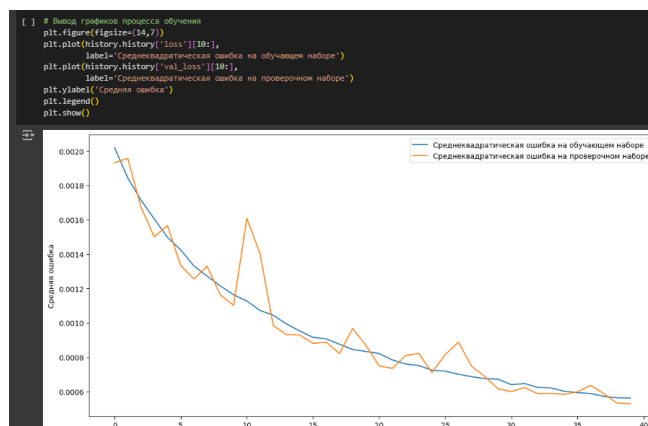


Рисунок 4. График процесса обучения модели

Отсюда можно сделать вывод, что результат хороший, т. к. ошибка уменьшается. Переобучение достаточно мало.

Далее попробуем задать на вход картинки, для этого выполняется код с указанием пути к директории хранения данных картинок. Далее выполняются следующие действия: сохранение/восстановление параметров модели; получение предсказания автокодировщика на тренировочной и тестовой выборках; сравнение исходных и восстановленных картинок из тестовой выборки; расчет количества пикселей изображения; загрузка и предобработка данных fashion_mnist; сравнение примеров mnist и fashion_mnist: очень разные картинки; получение предсказания автокодировщика на выборке fashion_mnist; сравнение исходных и восстановленных картинок из выборки fashion_mnist; сравнение среднеквадратических ошибок предсказания на mnist и fashion_mnist (выполнения этих частей кода можно просмотреть по ссылке в репозитории). Затем пишется функция для построения гистограммы, когда получим оптимальный порог (рис. 5).

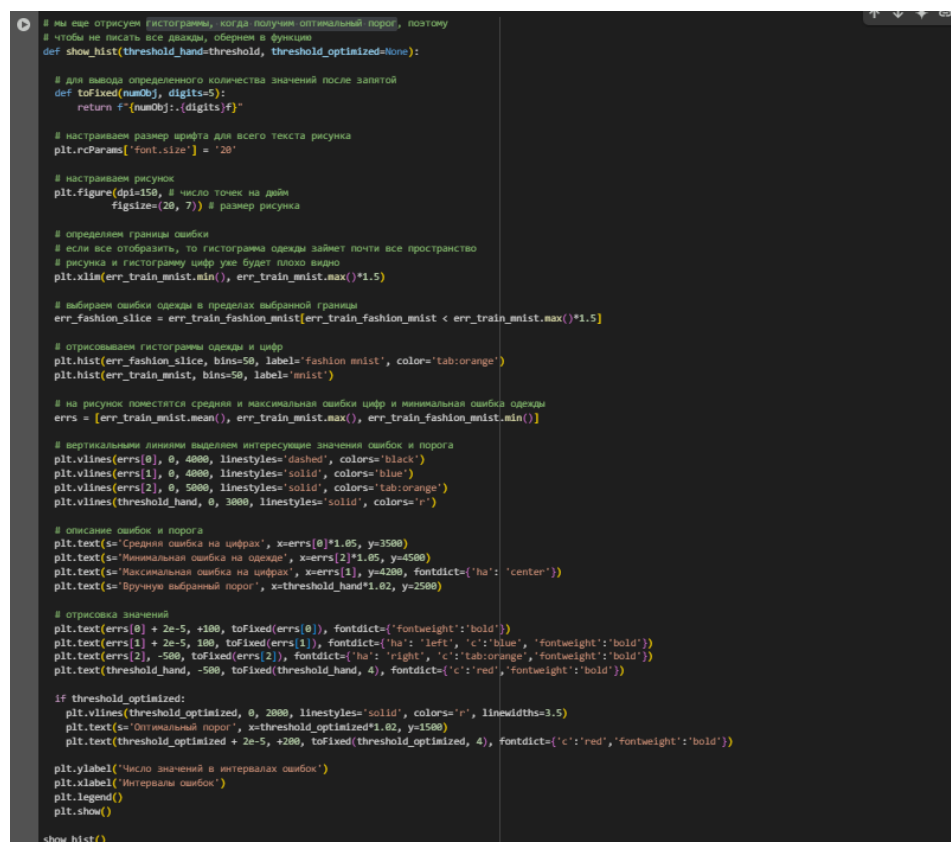


Рисунок 5. Функция для построения гистограммы

Посмотрим на результат выполнения данной функции (рис. 6).

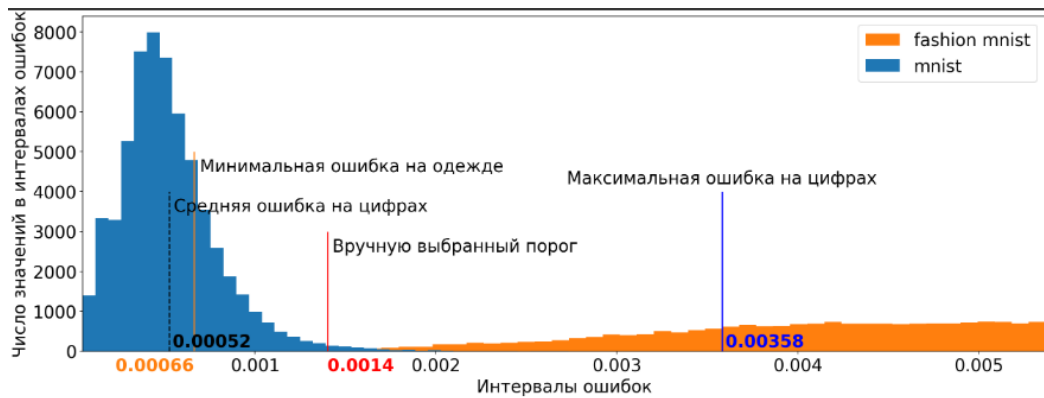


Рисунок 6. Полученная гистограмма

Отсюда видно, что гистограмма напоминает рисунок из теории. Только в нашем случае эти множественные выбросы не являются коллективными. Это точечные выбросы. Все левее порога считаем данными из mnist, все правее – данными из fashion_mnist (выбросы).

Далее выполняется определение оптимального порога, для этого была писана функция поиска оптимального порога и команда для определения оптимального порога для mnist и fashion_mnist, после чего был получен график (рис. 7).

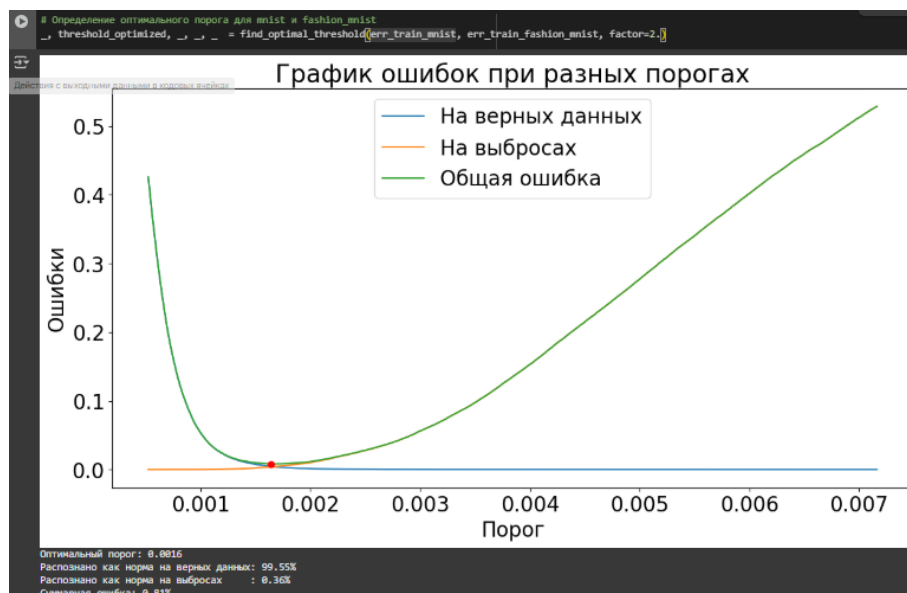


Рисунок 7. График оптимального порога

Отсюда видно, что с повышением порога ошибка на нормальных данных падает. И это логично. С увеличением порога все больше объектов нормальных данных становятся по левую сторону от порога, а значит, предсказываются как нормальные данные. Ошибка на верных данных

уменьшается. И такая же ситуация с выбросами: с увеличением порога все больше выбросов подходят под этот критерий нормальных данных. Ошибка на выбросах увеличивается.

Теперь снова отобразите гистограммы, но уже с добавлением нового порога (рис. 8).

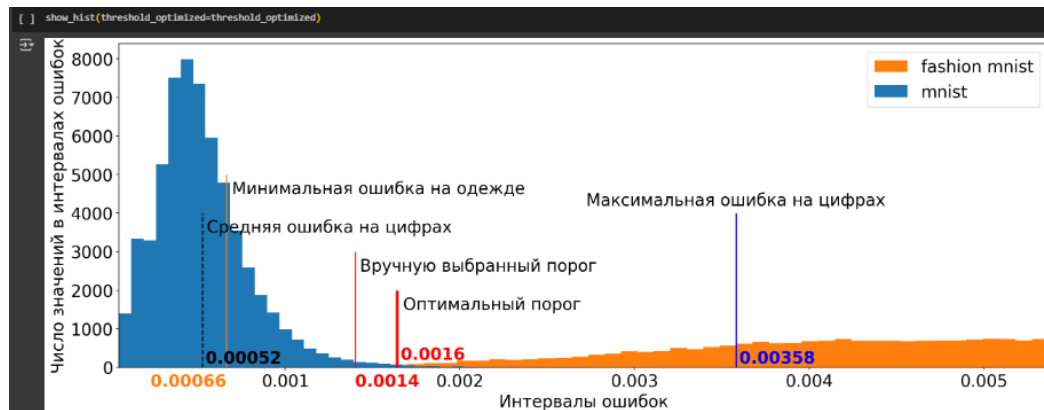


Рисунок 8. Гистограмма с добавлением нового порога

Отсюда видно, что вручную был почти угадан оптимальный порог, но перебрать 1 000 значений самостоятельно было бы трудно. Функция, которая была написана, ускорила процесс.

Далее будет выполнена визуализация динамики латентного пространства в процессе обучения, для этого выполняется создание и обучение автокодировщика. Для этого выполняются следующие действия: выполнение функции сборки автокодировщика по частям; создание и вывод трех моделей - составных частей автокодировщика и самого автокодировщика (рис. 9).

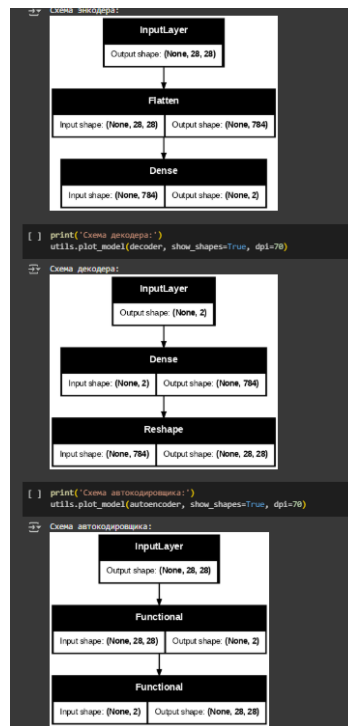


Рисунок 9. Вывод трех моделей

Далее визуализируем представление точек латентного пространства, для этого выполняются следующие действия: выбор случайного изображения из тренировочной выборки и показ его; получение предсказания энкодера (двумерная точка в латентном пространстве); задание цифр для визуализации латентного пространства и получение предсказания энкодера для отобранных примеров (рис. 10).

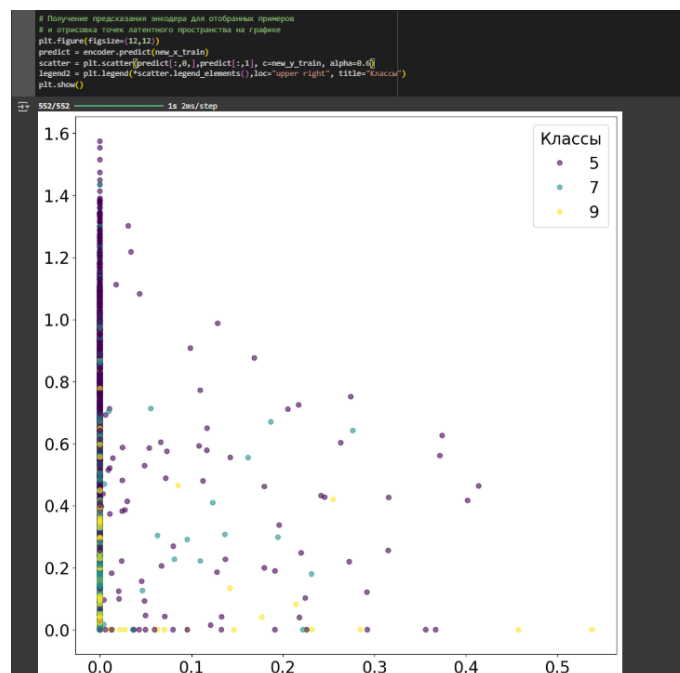


Рисунок 10. Получение предсказания энкодера

Далее выполняется визуализация динамики латентного пространства в процессе обучения автокодировщика, для этого выполняются следующие действия: назначения функции-коллбэка в конце эпохи; компиляция модели с выбранным оптимизатором и функцией потерь и обучение автокодировщика с протоколом работы (рис. 11).

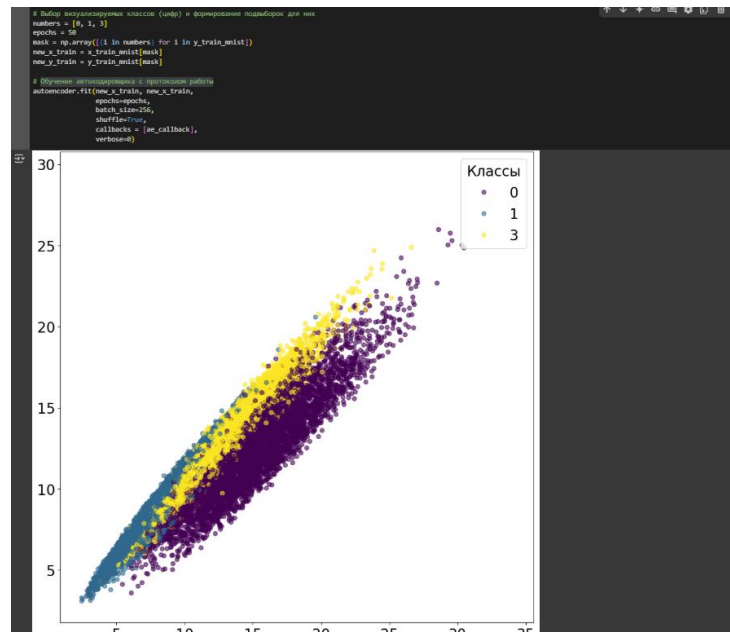


Рисунок 11. Обучение модели

Далее когда обучение НС закончено, возьмем любую точку из скрытого пространства, и подадим ее в декодер и посмотрим на результат (рис. 12).

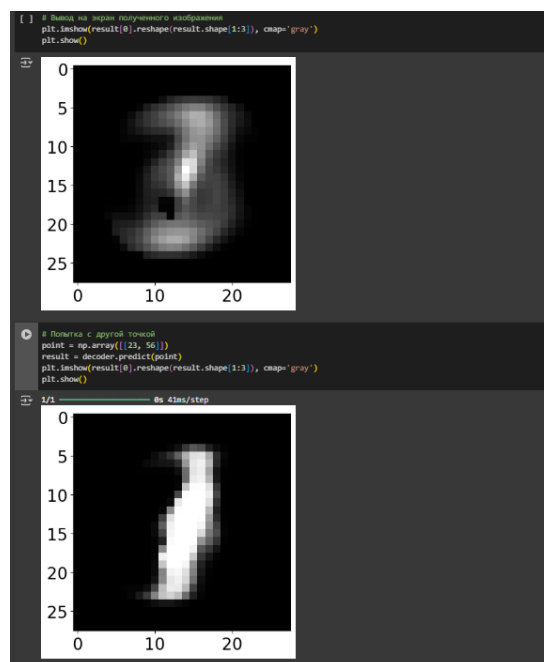


Рисунок 12. Вывод на экран полученного изображения

Затем при помощи библиотеки `imageio` создадим анимацию динамики скрытого пространства, используя графики, которые сохранены в `callback` (рис. 13).

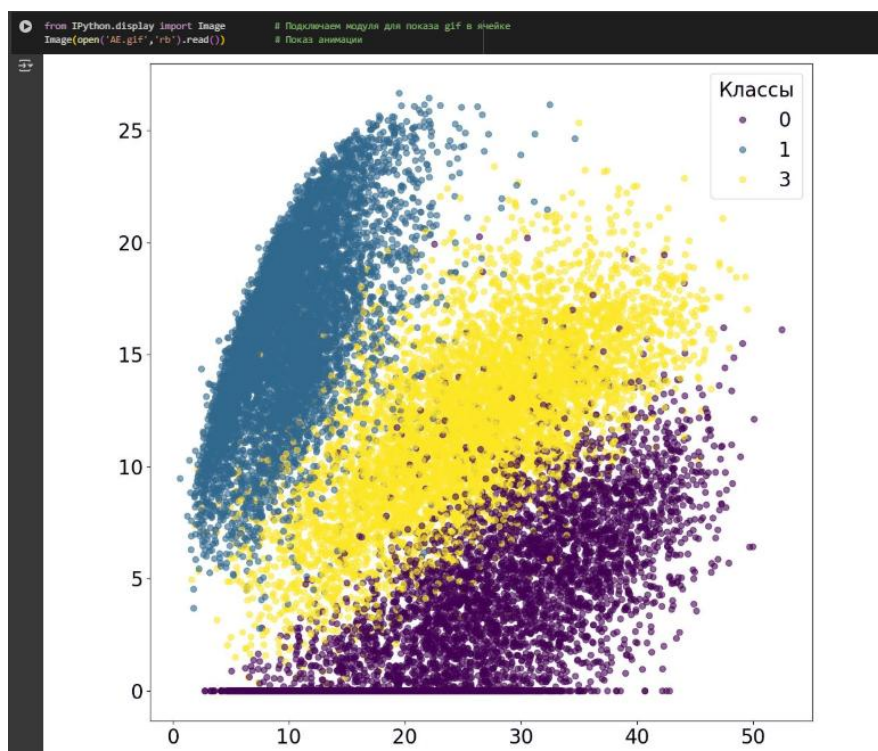


Рисунок 13. Показ анимации

Отсюда, можно сделать вывод, что суть автокодировщика состоит в восстановлении исходного изображения из вектора скрытого пространства. Для этого нужно, чтобы кодировщик правильно выделил ключевые признаки, декодировщик умел восстановить из этих признаков само изображение. Также нужно, конечно, достаточно сложное скрытое пространство, способное вместить эти ключевые признаки.

В нашем случае в скрытом пространстве всего 2 измерения. Соответственно в них кодировщик пытается выделить 2 самых ярких, самых главных признака датасета. Под признаками имеется в виду не одно из чисел изображения. Признак в данном случае – это абстрактное понятие, известное лишь самому автокодировщику.

Как видно из гифки, по этим двум признакам уже можно различить классы. По крайней мере большую часть изображений этих трех классов друг от друга.

Практика 2. Базовый блок Архитектура автокодировщика Autoencoder.

В данной практике также рассматриваются автокодировщики. Здесь необходимо выполнить следующее:

- построить и обучить модель автокодировщика на изображениях с лицами людей;
- написать алгоритм генерации шума на этих изображениях;
- узнать, как на практике модель удаляет шум с изображений;
- построить и обучить модель для распознавания мошеннических транзакций.

Решение данной практики начинается с импорта необходимых библиотек для работы. Затем рассмотрим процесс удаления шума на изображениях лиц, для этого напишем архитектуру автокодировщика (рис. 14).

```
def create_base_ae(in_shape):
    inputs = Input(in_shape)

    # 3 блока свертки перед скатием
    x = Conv2D(12, 3, padding='same', activation='relu')(inputs)
    x = BatchNormalization()(x)

    # используем разрежение (dilation_rate) для выявления мелких деталей изображения
    x = Conv2D(24, 3, dilation_rate=2, padding='same', activation='relu')(x)
    x = BatchNormalization()(x)

    x = Conv2D(48, 3, padding='same', activation='relu')(x)
    x = BatchNormalization()(x)

    # Сжатие 1
    x = Conv2D(96, 3, strides=2, padding='same', activation='relu')(x)
    x = BatchNormalization()(x)

    x = Conv2D(96, 3, padding='same', activation='relu')(x)
    x = BatchNormalization()(x)

    # Сжатие 2
    x = Conv2D(128, 3, strides=2, padding='same', activation='relu')(x)
    x = BatchNormalization()(x)

    x = Conv2D(256, 3, padding='same', activation='relu')(x)

    # Скрытое пространство
    z = BatchNormalization()(x)

    x = Conv2DTranspose(128, 3, strides=2, padding='same', activation='relu')(z)
    x = BatchNormalization()(x)

    x = Conv2D(128, 3, padding='same', activation='relu')(x)
    x = BatchNormalization()(x)

    x = Conv2DTranspose(96, 3, strides=2, padding='same', activation='relu')(x)
    x = BatchNormalization()(x)

    x = Conv2D(48, 3, padding='same', activation='relu')(x)
    x = BatchNormalization()(x)

    x = Conv2D(24, 3, padding='same', activation='relu')(x)
    x = BatchNormalization()(x)

    x = Conv2D(8, 3, padding='same', activation='relu')(x)
    x = BatchNormalization()(x)

    # финальный слой свертки, выход модели
    outputs = Conv2D(1, 2, dilation_rate=2, padding='same', activation='sigmoid')(x)

    # Сборка модели. На входе оригинальное изображение, на выходе - сжатое-восстановленное
    model = Model(inputs, outputs)

    # компиляция модели
    model.compile(optimizer=Adam(1e-4),
                  loss='mse')

    return model
```

Рисунок 14. Архитектура автокодировщика

Далее были написаны сервисные функции, такие как функция последовательного вывода нескольких изображений для сравнения. Далее будут выполнены следующие действия: функция поиска оптимального порога; загрузка датасета и подготовка данных, настройка констант; функция загрузки изображения; загрузка датасета в память; сжатие диапазона $[0, 255]$ значений к диапазону $[0, 1]$; разбиение на обучающие и проверочные целевые изображения; выбор индекса случайного изображения из тренировочных; удаление последней оси для корректной работы plt (выполнения этих частей кода можно просмотреть по ссылке в репозитории). Далее выполним вывод примера в черно-белом формате (рис. 15).

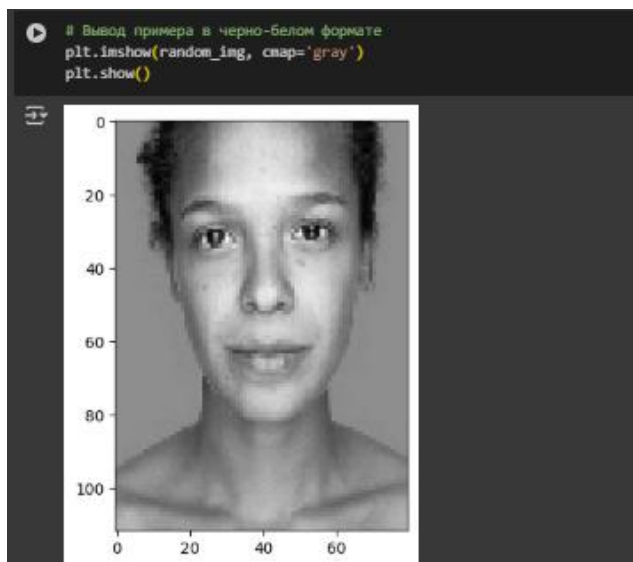


Рисунок 15. Вывод примера в черном-белом формате

Далее выполним исследование шума. Для генерации шумных картинок, необходимо выполнить следующие действия: генерация тензора случайных значения сразу равных по количеству и форме исходных данных; создание зашумленных изображений. Затем создаем полотно и отрисовываем изображения (рис. 16).



Рисунок 16. Создание полотна и отрисовка изображений

Далее создадим и обучим автокодировщик, для этого сначала выполним отображение модели (рис. 17).

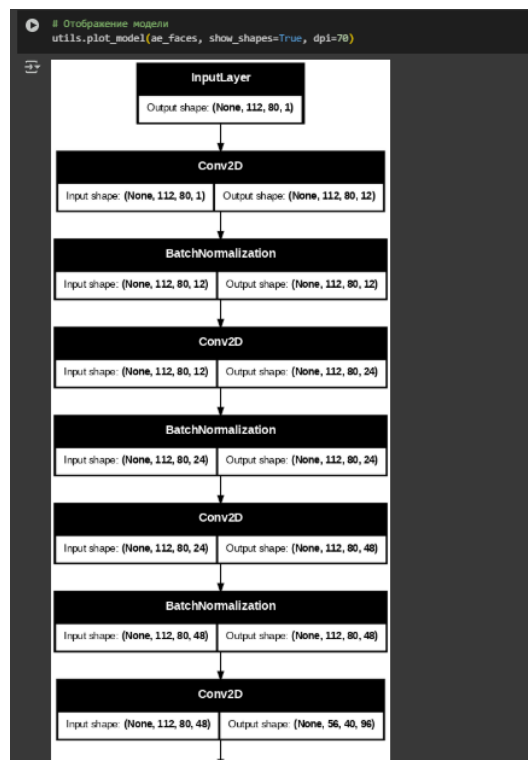


Рисунок 17. Отображение модели

Затем выполняется обучение модели на датасете лиц. Результат обучения можно считать отличным так как, MSE loss на тесте $< 10 - 3$ за ~ 20 мин. Переобучение не наблюдается. Далее выполняется сохранение данных в папку и прописываются следующие операции: сохранение/восстановление

всей модели; получение предсказания автокодировщика по обучающей выборке. Далее выполним сравнение исходных и восстановленных картинок из обучающей выборки (рис. 18).

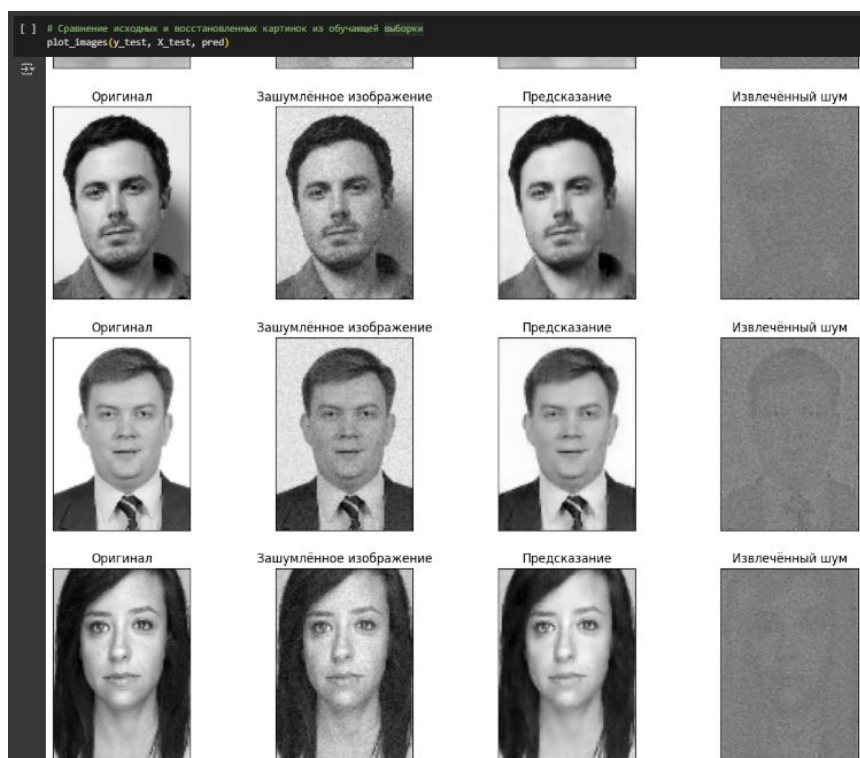


Рисунок 18. Сравнение исходных и восстановленных картинок

Отсюда, видно, что изображения на выходе модели на вид очень похожи на оригинальные. Шум в значительной степени убран с изображений, сами лица восстановлены с большей частью деталей. Если приглядеться, можно увидеть небольшое размытие на предсказаниях. Некоторые детали (например, щетина) нейросеть распознала как шум и попыталась убрать.

Далее определим мошеннические операции на датасете транзакций по кредитным картам. Для этого выполним следующие действия: загрузка датасета и подготовка данных; чтение данных в таблицу; выделение данных нормальных и мошеннических операций; разбивка нормальных данных на обучающую и тестовую выборки. А также выполняется завершение подготовки данных. Затем выполним создание и обучение модели автокодировщика и выполним просмотр графика обучения процесса (рис. 19).

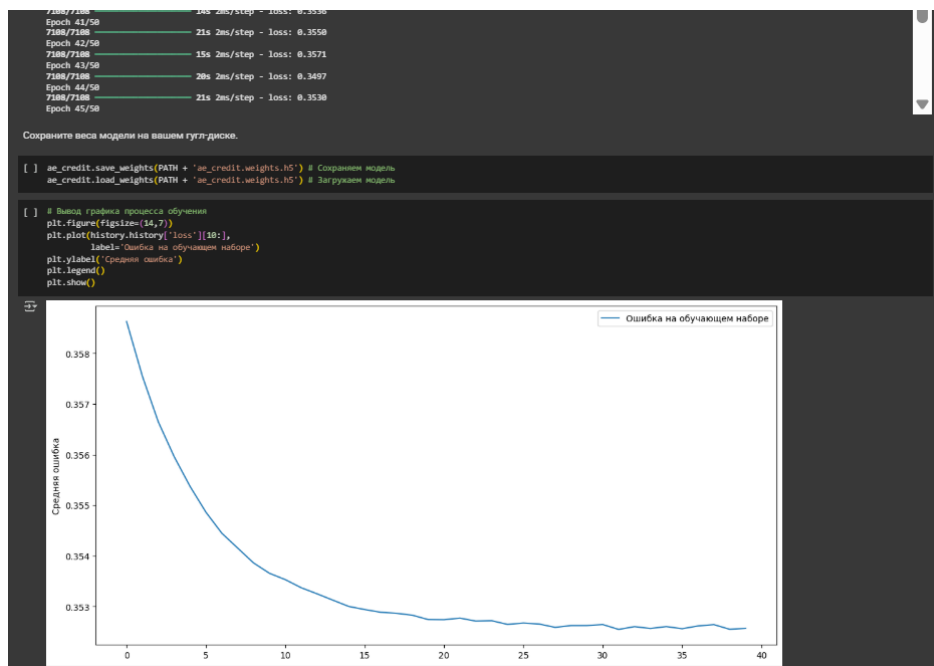


Рисунок 19. График обучения процесса обучения

Отсюда видно по графику, ошибка на обучающей выборке имеет скачкообразный характер, но тренд нисходящий.

Далее выполняется определение выбросов, для этого выполняются следующие операции: получение предсказания автокодировщика на тестовой выборке (нормальные + мошеннические транзакции); вычисление среднеквадратических ошибок на тестовой выборке. Выполним вывод результатов (рис. 20).

```
[ ] print('Минимальная ошибка нормальных транзакций: {:.5f}'.format(err_normal.min()))
    print('Максимальная ошибка нормальных транзакций: {:.5f}'.format(err_normal.max()))
    print('Средняя ошибка нормальных транзакций: {:.5f}'.format(err_normal.mean()))

Минимальная ошибка нормальных транзакций: 0.84168
Максимальная ошибка нормальных транзакций: 69.97714
Средняя ошибка нормальных транзакций: 0.35429

[ ] print('Минимальная ошибка мошеннических транзакций: {:.5f}'.format(err_fraud.min()))
    print('Максимальная ошибка мошеннических транзакций: {:.5f}'.format(err_fraud.max()))
    print('Средняя ошибка мошеннических транзакций: {:.5f}'.format(err_fraud.mean()))

Минимальная ошибка мошеннических транзакций: 0.15935
Максимальная ошибка мошеннических транзакций: 145.79848
Средняя ошибка мошеннических транзакций: 20.31876
```

Рисунок 20. Вывод результатов

Затем выполним определение порога для нормальных и мошеннических транзакций (рис. 21).

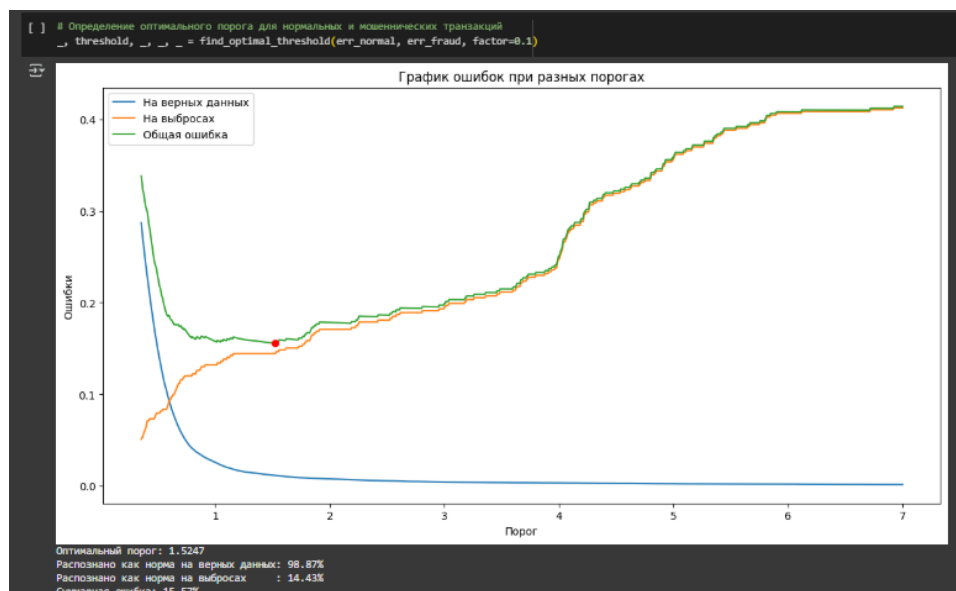


Рисунок 21. Определение оптимального порога

Затем вычислим доли верно распознанных нормальных и мошеннических транзакций (рис. 22).

```
# Вычисление доли верно распознанных нормальных и мошеннических транзакций
found_normal = (err_normal < threshold).mean()
found_fraud = (err_fraud >= threshold).mean()

# Вывод статистик
print('Распознано нормальных транзакций: {:.2f}%'.format(100. * found_normal))
print('Распознано мошеннических транзакций: {:.2f}%'.format(100. * found_fraud))
print('Средняя точность распознавания: {:.2f}%'.format(50. * (found_normal + found_fraud)))
```

Распознано нормальных транзакций: 98.87%
 Распознано мошеннических транзакций: 85.57%
 Средняя точность распознавания: 92.22%

Рисунок 22. Вывод статистик

Отсюда подведем итог, создана и обучена модель автокодировщика с целью отделения мошеннических от нормальных транзакций. В целом задача решена успешно – средняя точность распознавания ~92%. В этой задаче была взята максимально простая модель, что оставляет большое пространство для экспериментов.

Выполнение индивидуальных заданий:

Задание 1.

Условие: необходимо добиться на автокодировщике с 2-мерным скрытым пространством на 3-х цифрах: 0, 1 и 3 – ошибки $MSE < 0.034$ на скорости обучения 0.001 на 10-й эпохе.

Выполнение данного задания начнем с импорта необходимых библиотек (рис. 23).


```

[23] # Работа с операционной системой
import os

# Отрисовка графиков
import matplotlib.pyplot as plt

# Операции с путями
import glob

# Работа с массивами данных
import numpy as np

# Слои
from tensorflow.keras.layers import Dense, Flatten, Reshape, Input, Conv2DTranspose, concatenate, Activation, MaxPooling2D, Conv2D,

# Модель
from tensorflow.keras import Model

# Загрузка модели
from tensorflow.keras.models import load_model

# Датасет
from tensorflow.keras.datasets import mnist

# Оптимизатор для обучения модели
from tensorflow.keras.optimizers import Adam

# Коллбеки для выдачи информации в процессе обучения
from tensorflow.keras.callbacks import LambdaCallback

%matplotlib inline

```

Рисунок 23. Импорт библиотек

Затем выполним удаление изображений. Это стоит применять при обучении новой модели, чтобы не было путаницы в картинках. Для этого напишем следующую функцию (рис. 24).

```

[24] def clean():
    # Получение названий всех картинок
    paths = glob.glob('*.jpg')

    # Удаление всех картинок по полученным путям
    for p in paths:
        os.remove(p)

    # Удаление всех картинок
    clean()

```

Рисунок 24. Функция удаления всех картинок

Далее выполняется загрузка и подготовка данных (рис. 25).

```

[48] # Загрузка и подготовка данных
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = (X_train.astype('float32') / 255.).reshape(-1, 784) # Формат (None, 784)
X_test = (X_test.astype('float32') / 255.).reshape(-1, 784)

# Выбор только цифр 0, 1, 3
numbers = [0, 1, 3]
mask = np.array([i in numbers for i in y_train])
X_train = X_train[mask]
y_train = y_train[mask]

```

Рисунок 25. Загрузка и подготовка данных

Далее выполним загрузку необходимых библиотек, которые так же будут нужны для задания (рис. 26).

```

[38] import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Reshape, Conv2DTranspose, UpSampling2D, LeakyReLU, Dropout, GaussianNoise
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.datasets import mnist
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.regularizers import l1_l2
from tensorflow.keras.regularizers import l2

```

Рисунок 26. Загрузка библиотек

Затем напишем функцию-коллбэк, которая будет отрисовывать объекты в скрытом пространстве (рис. 27).

```
[50] def ae_on_epoch_end(epoch, logs):
    print('_____')
    print(f'*** ЭПОХА: {epoch+1}, loss: {logs["loss"]} ***')
    print('_____')

    # Получение картинки латентного пространства в конце эпохи и запись в файл
    # Задание числа пикселей на дюйм
    plt.figure(dpi=100)

    # Предсказание энкодера на тренировочной выборке
    predict = encoder.predict(X_train)

    # Создание рисунка: множество точек на плоскости 3-х цветов (3-х классов)
    scatter = plt.scatter(predict[:,0],predict[:,1], c=y_train, alpha=0.6, s=5)

    # Создание легенды
    legend2 = plt.legend(*scatter.legend_elements(), loc='upper right', title='Классы')

    # Сохранение картинки с названием, которого еще нет
    paths = glob.glob('*.jpg')
    plt.savefig(f'image_{str(len(paths)).zfill(4)}.jpg')

    # Отображение. Без него рисунок не отрисуется
    plt.show()

    ae_callback = LambdaCallback(on_epoch_end=ae_on_epoch_end)
```

Рисунок 27. Функция-коллбэк

Далее выполним создание архитектуры автокодировщика (рис. 28) и выполним компиляцию.

```
[51] # Архитектура автокодировщика
input_img = Input(shape=(28, 28, 1))

[52] from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model

# Размерность входных данных (784 = 28*28)
input_dim = 784
encoding_dim = 2 # 2D скрытое пространство

# Энкодер
input_layer = Input(shape=(input_dim,))
encoder_layer = Dense(128, activation='relu')(input_layer)
encoder_layer = Dense(64, activation='relu')(encoder_layer)
encoder_output = Dense(encoding_dim)(encoder_layer)

# Декодер
decoder_layer = Dense(64, activation='relu')(encoder_output)
decoder_layer = Dense(128, activation='relu')(decoder_layer)
decoder_output = Dense(input_dim, activation='sigmoid')(decoder_layer)

# Полная модель
autoencoder = Model(inputs=input_layer, outputs=decoder_output)
encoder = Model(inputs=input_layer, outputs=encoder_output)

# Компиляция
autoencoder.compile(optimizer=Adam(learning_rate=0.001), loss='mse')
```

Рисунок 28. Архитектура автокодировщика

Затем выполним обучение для созданной модели автокодировщика (рис. 29).

```
[53] history = autoencoder.fit(
    X_train, X_train,
    epochs=10,
    batch_size=128,
    validation_data=(X_test, X_test),
    verbose=1,
    callbacks=[ae_callback]
)
```

Рисунок 29. Обучение модели

После посмотрим на результаты обучения на каждой эпохе, так же посмотрим на ошибку MSE и на картинки латентного пространства (рис. 30 - 40).

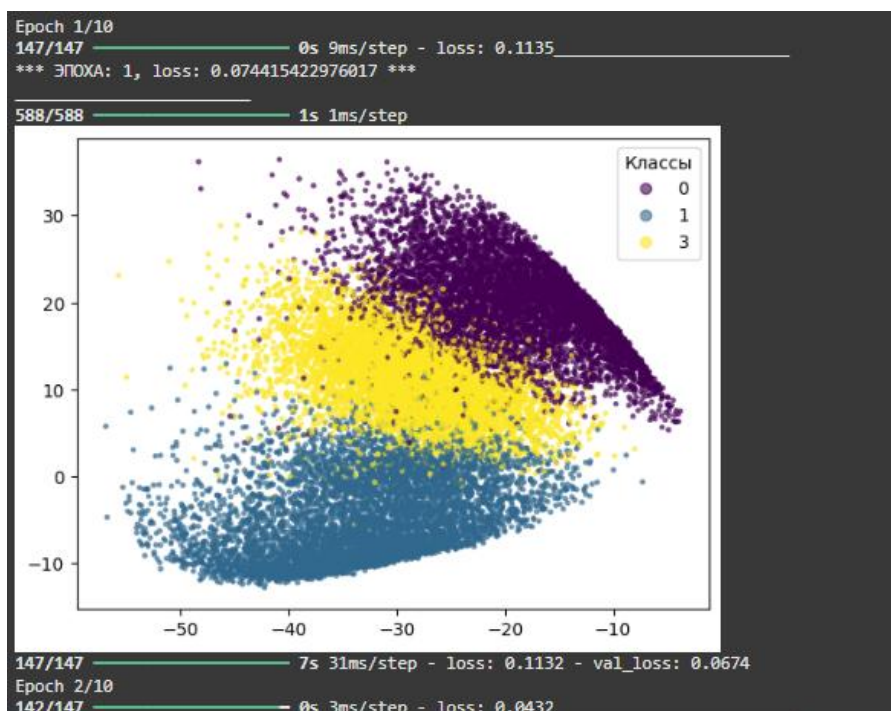


Рисунок 30. Процесс обучения – 1 эпоха

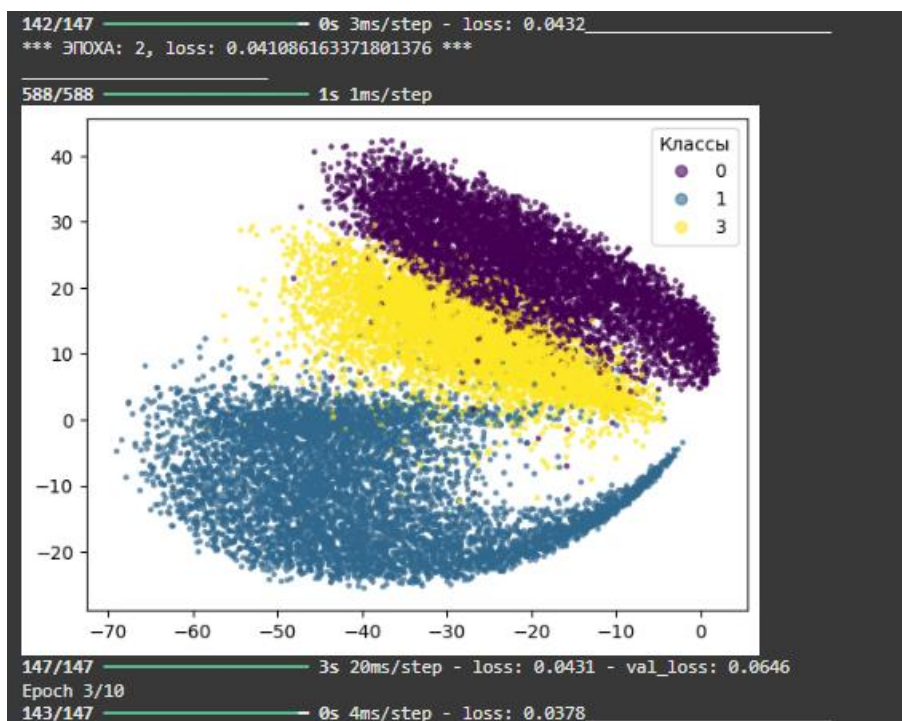


Рисунок 31. Процесс обучения – 2 эпоха

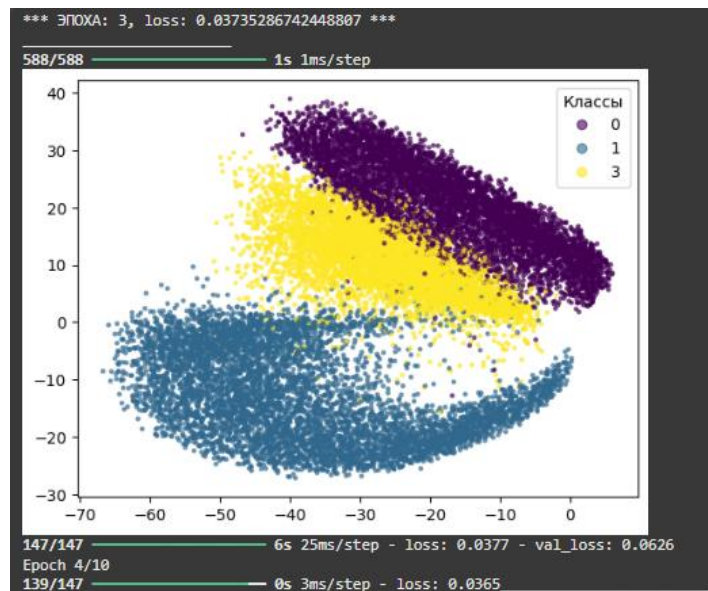


Рисунок 33. Процесс обучения – 3 эпоха

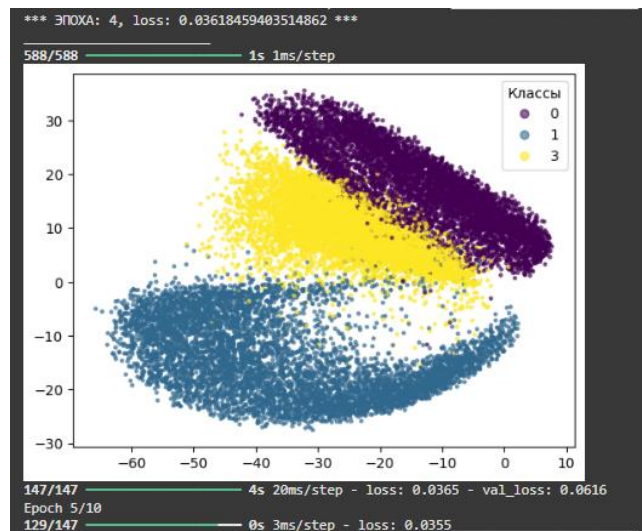


Рисунок 34. Процесс обучения – 4 эпоха

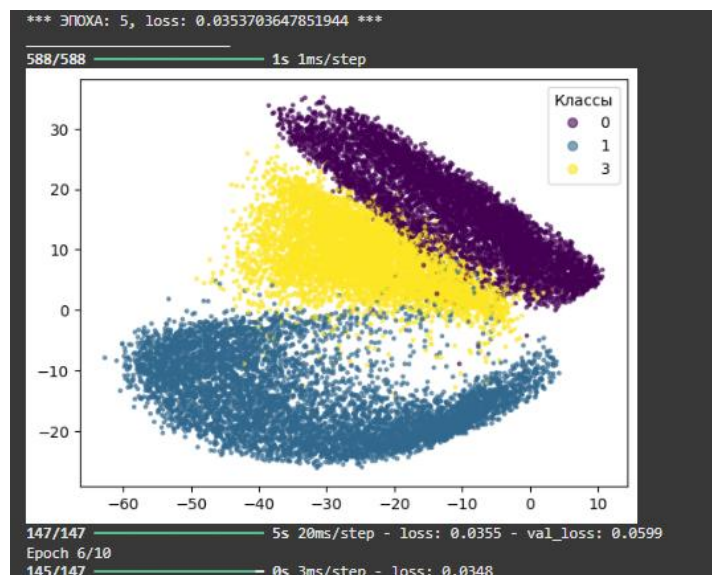


Рисунок 35. Процесс обучения – 5 эпоха

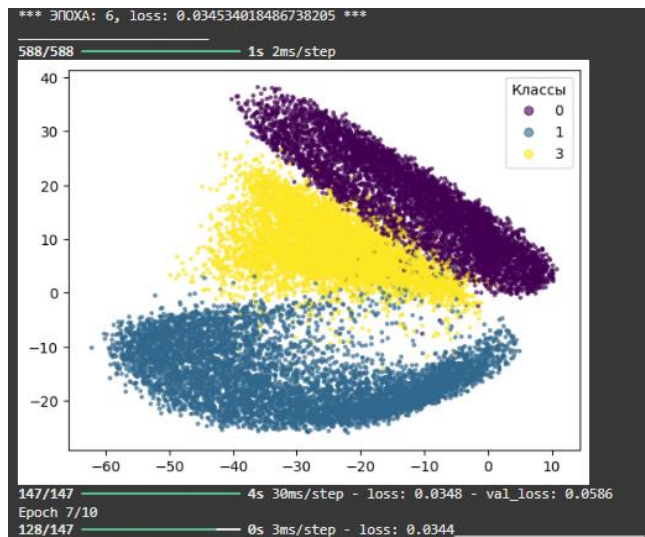


Рисунок 36. Процесс обучения – 6 эпоха

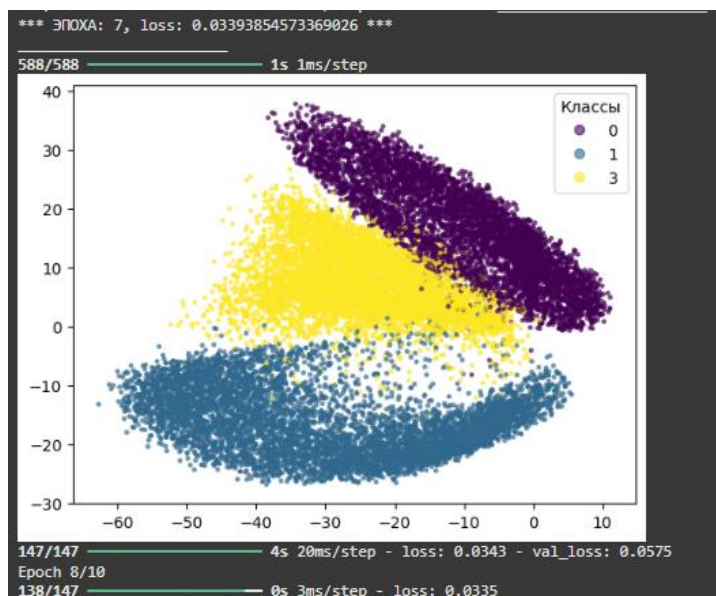


Рисунок 37. Процесс обучения – 7 эпоха

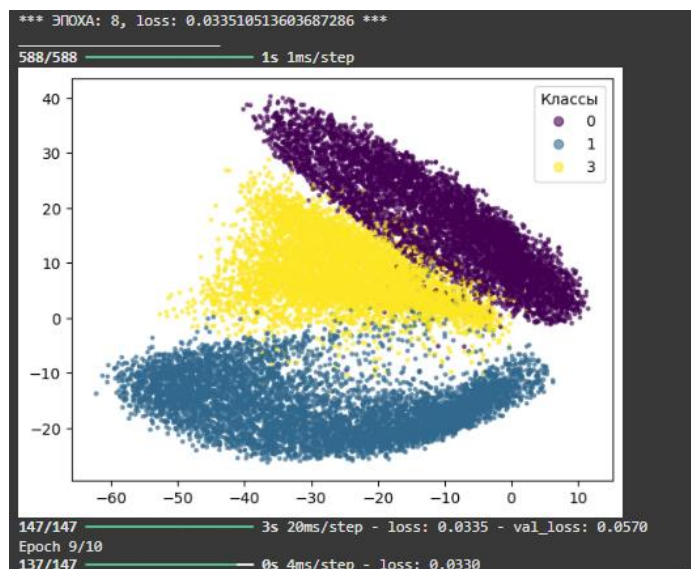


Рисунок 38. Процесс обучения – 8 эпоха

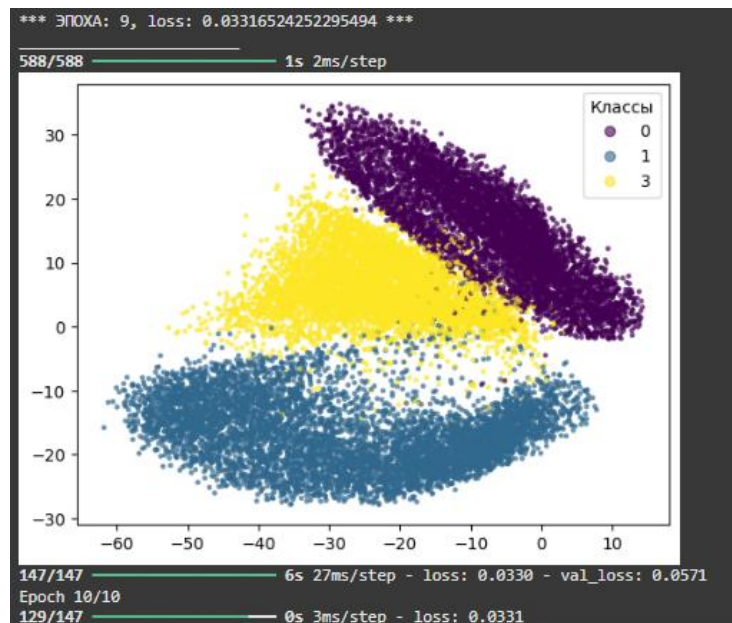


Рисунок 39. Процесс обучения – 9 эпоха

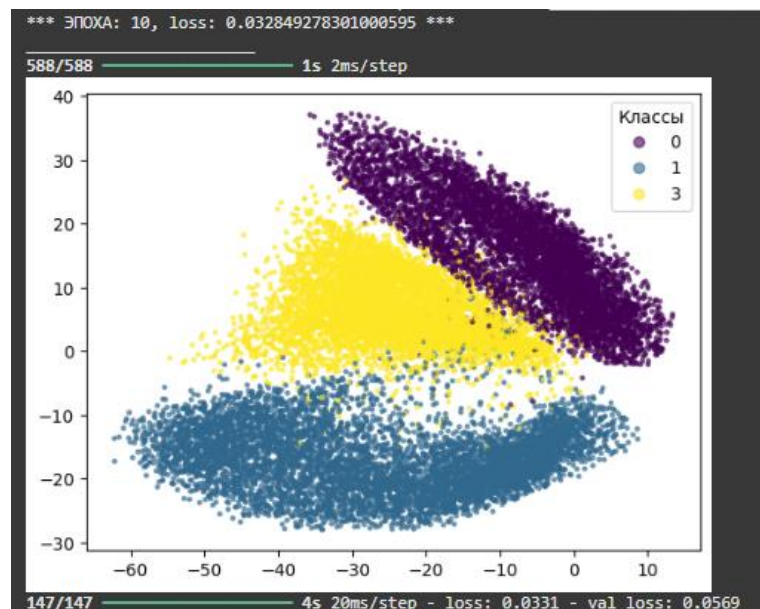


Рисунок 40. Процесс обучения – 10 эпоха

Далее посмотрим на результаты ошибки на 10 эпохе (рис. 41).

```
print(f"Final Train MSE: {history.history['loss'][-1]:.5f}")
print(f"Final Val MSE: {history.history['val_loss'][-1]:.5f}")
```

Final Train MSE: 0.03285

Final Val MSE: 0.05690

Рисунок 41. Результаты ошибки

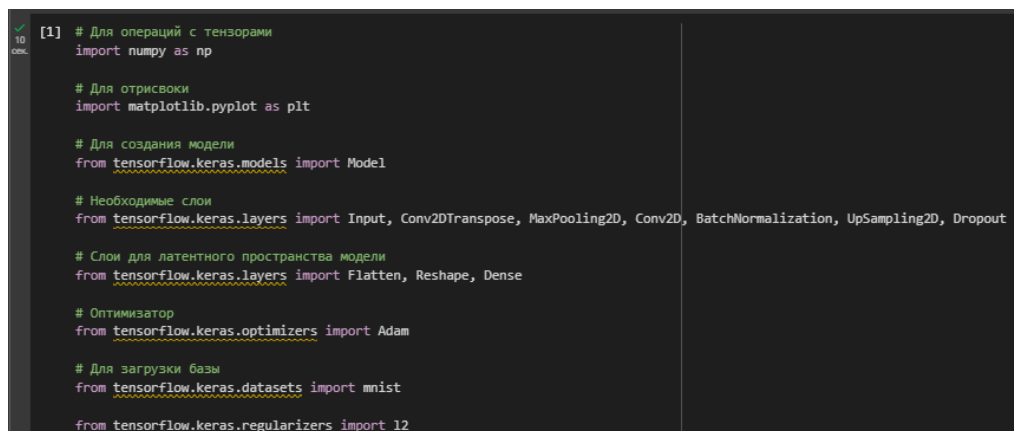
Так как условие требовало $MSE < 0.034$ на 10-й эпохе для обучающей выборки, тогда данная модель успешно достигла этого ($0.03253 < 0.034$). И

следовательно решение полностью соответствует условию задачи ($MSE < 0.034$ на 10-й эпохе).

Задание 2. Необходимо выполнить следующие:

1. Выбрать 10 самых красивых по мнению пятерок в тренировочной выборке mnist.
2. Создать датасет, где объекты – это все пятерки из тренировочной части mnist, а метки – это случайные пятерки из "красивого" набора.
3. Создать автокодировщик и проверить, совпадают ли у него размеры выхода и входа.
4. Обучить автокодировщик.
5. Добиться ошибки MSE на тренировочной выборке < 0.05 .
6. Посмотреть, как выглядят пятерки из тестовой выборки после обученного автокодировщика.

Для выполнения данного задания, сначала импортируем все необходимые библиотеки для работы (рис. 42).



```
[1] # Для операций с тензорами
import numpy as np

# Для отрисовки
import matplotlib.pyplot as plt

# Для создания модели
from tensorflow.keras.models import Model

# Необходимые слои
from tensorflow.keras.layers import Input, Conv2DTranspose, MaxPooling2D, Conv2D, BatchNormalization, UpSampling2D, Dropout

# Слой для латентного пространства модели
from tensorflow.keras.layers import Flatten, Reshape, Dense

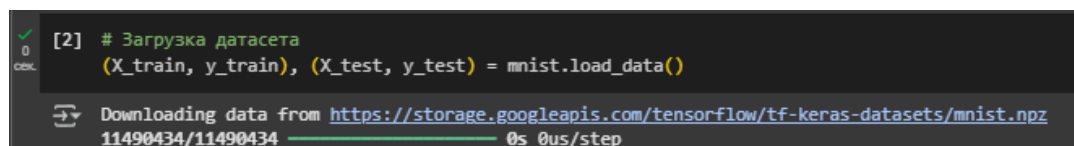
# Оптимизатор
from tensorflow.keras.optimizers import Adam

# Для загрузки базы
from tensorflow.keras.datasets import mnist

from tensorflow.keras.regularizers import l2
```

Рисунок 42. Импорт библиотек

Затем выполним загрузку данных, а именно загрузку датасета (рис. 43).



```
[2] # Загрузка датасета
(X_train, y_train), (X_test, y_test) = mnist.load_data()

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 — 0s 0us/step
```

Рисунок 43. Загрузка датасета

Далее выполним нормализацию данных и приведение формы к удобной для Keras (рис. 44).

```

[3] # Нормализация данных
X_train = X_train.astype('float32')/255.
X_test = X_test.astype('float32')/255.

[4] # Приведение формы к удобной для Keras
X_train = X_train.reshape(-1, 28, 28, 1)
X_test = X_test.reshape(-1, 28, 28, 1)

```

Рисунок 44. Нормализация данных

Затем выполним отбор пятерок нужных для обучающих и тестовых данных (рис. 45).

```

[5] # Отбор пятерок
mask = y_train == 5
X_train = X_train[mask]
y_train = y_train[mask]

[6] # Аналогично для тестирования
mask = y_test == 5
X_test = X_test[mask]
y_test = y_test[mask]

```

Рисунок 45. Отбор пятерок

Потом выполним отбор красивых пятерок, для этого выберем конкретные пятерки по указанным индексам (рис. 46).

```

[7] # Выбираем конкретные пятерки по указанным индексам
selected_indices = [2, 3, 8, 9, 10, 15, 16, 19, 22, 25, 30]
beautiful_fives = X_test[selected_indices]

```

Рисунок 46. Отбор пятерок по индексам

Далее создадим метки для красивых пятерок (рис. 47).

```

[8] # Создание меток - случайные "красивые" пятерки
np.random.seed(42)
labels = beautiful_fives[np.random.randint(0, len(selected_indices), size=len(X_train))]

```

Рисунок 47. Создание меток

Затем выполним визуализацию отобранных красивых пятерок (рис. 48).

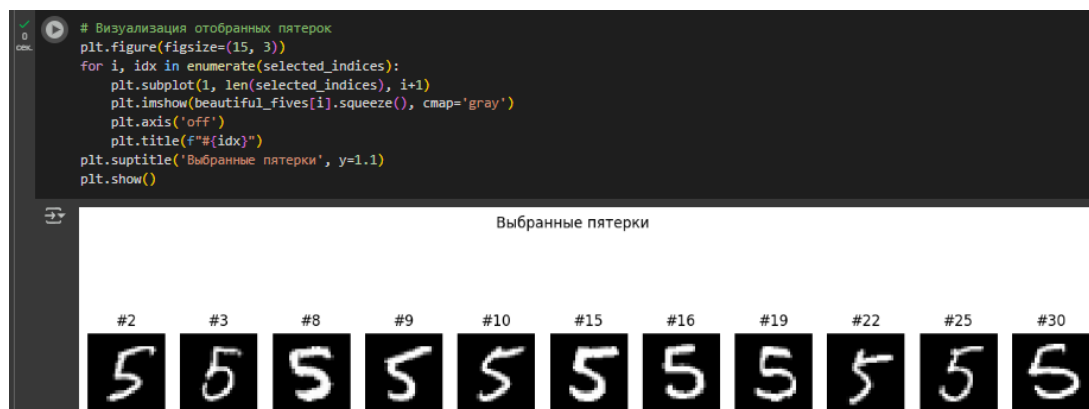


Рисунок 48. Визуализация отобранных пятерок

Далее выполним создание модели автокодировщика (рис. 49).

```
[10] # Входной слой
input_img = Input(shape=(28, 28, 1))

# Энкодер
x = Conv2D(32, (3, 3), strides=2, padding='same', activation='relu', kernel_regularizer=l2(0.0001))(input_img)
x = BatchNormalization()(x)
x = Conv2D(64, (3, 3), strides=2, padding='same', activation='relu', kernel_regularizer=l2(0.0001))(x)
x = BatchNormalization()(x)
x = Conv2D(128, (3, 3), padding='same', activation='relu', kernel_regularizer=l2(0.0001))(x)
encoded = BatchNormalization()(x)

# Декодер
x = Conv2DTranspose(128, (3, 3), padding='same', activation='relu')(encoded)
x = BatchNormalization()(x)
x = Conv2DTranspose(64, (3, 3), strides=2, padding='same', activation='relu')(x)
x = BatchNormalization()(x)
x = Conv2DTranspose(32, (3, 3), strides=2, padding='same', activation='relu')(x)
x = BatchNormalization()(x)
decoded = Conv2D(1, (3, 3), padding='same', activation='sigmoid')(x)

# Модель
autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer=Adam(learning_rate=0.001), loss='mse')
```

Рисунок 49. Создание модели автокодировщика

Дальше проверим совпадают ли у созданного автокодировщика размеры выхода и входа (рис. 50).

```
# Проверка размеров
print("Размер входа:", autoencoder.input_shape)
print("Размер выхода:", autoencoder.output_shape)

Размер входа: (None, 28, 28, 1)
Размер выхода: (None, 28, 28, 1)
```

Рисунок 50. Проверка размеров

Отсюда можно заметить, что размер входа и выхода совпадает. Далее выполним обучение созданной модели (рис. 51). И выполним проверку ошибки MSE, она должна быть меньше 0,05.

```
history = autoencoder.fit(
    X_train, labels,
    epochs=15,
    batch_size=64,
    shuffle=True,
    validation_data=(X_test, X_test),
    callbacks=[
        EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True),
        ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5)
    ],
    verbose=1
)

# Результаты
train_mse = history.history['loss'][-1]
val_mse = history.history['val_loss'][-1]
print(f"Final Train MSE: {train_mse:.5f}") # Должно быть < 0.05
```

```
Epoch 1/15 12s 61ms/step - loss: 0.1438 - val_loss: 0.1026 - learning_rate: 0.0010
Epoch 2/15 3s 8ms/step - loss: 0.0633 - val_loss: 0.0958 - learning_rate: 0.0010
Epoch 3/15 1s 7ms/step - loss: 0.0569 - val_loss: 0.0973 - learning_rate: 0.0010
Epoch 4/15 1s 6ms/step - loss: 0.0543 - val_loss: 0.1002 - learning_rate: 0.0010
Epoch 5/15 1s 7ms/step - loss: 0.0525 - val_loss: 0.0955 - learning_rate: 0.0010
Epoch 6/15 1s 7ms/step - loss: 0.0518 - val_loss: 0.0856 - learning_rate: 0.0010
Epoch 7/15 1s 7ms/step - loss: 0.0515 - val_loss: 0.0711 - learning_rate: 0.0010
Epoch 8/15 1s 7ms/step - loss: 0.0510 - val_loss: 0.0717 - learning_rate: 0.0010
Epoch 9/15 1s 7ms/step - loss: 0.0511 - val_loss: 0.0734 - learning_rate: 0.0010
Epoch 10/15 1s 7ms/step - loss: 0.0507 - val_loss: 0.0742 - learning_rate: 0.0010
Epoch 11/15 1s 7ms/step - loss: 0.0498 - val_loss: 0.0716 - learning_rate: 0.0010
Epoch 12/15 1s 7ms/step - loss: 0.0490 - val_loss: 0.0727 - learning_rate: 0.0010
Epoch 13/15 1s 7ms/step - loss: 0.0484 - val_loss: 0.0705 - learning_rate: 5.0000e-04
Epoch 14/15 1s 7ms/step - loss: 0.0474 - val_loss: 0.0714 - learning_rate: 5.0000e-04
Epoch 15/15 1s 8ms/step - loss: 0.0470 - val_loss: 0.0724 - learning_rate: 5.0000e-04
Final Train MSE: 0.04726
```

Рисунок 51. Обучение модели

Ошибка составила 0,04726, что меньше 0,05 и следовательно данная часть задания выполнена. Далее посмотрим, как выглядят пятерки из тестовой выборки после обученного автокодировщика.

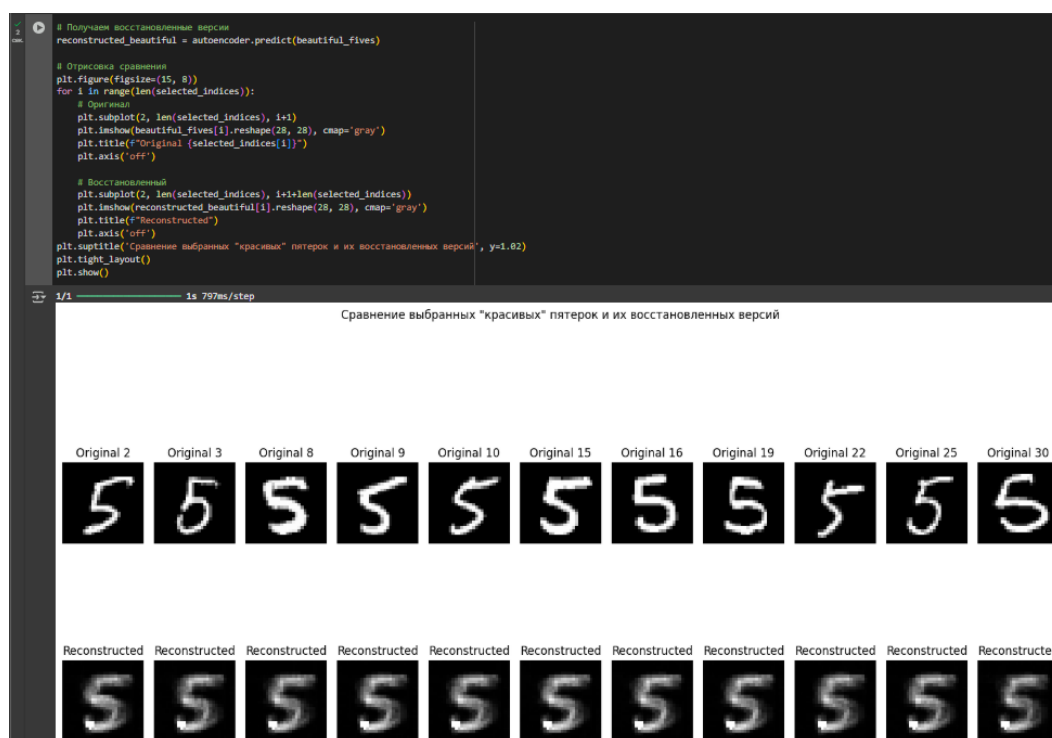


Рисунок 52. Вывод пятерок после обучения

Условие:

Задание 3.

Условие: необходимо создать автокодировщик, удаляющий черные квадраты в случайных областях изображений.

Алгоритм действий:

1. Взять базу картинок Mnist.
2. На картинках в случайных местах сделать чёрные квадраты размера 8 на 8.
3. Создать и обучить автокодировщик восстанавливать оригинальные изображения из "зашумленных" квадратом изображений.
4. Добиться $MSE < 0.0070$ на тестовой выборке

Выполнение данного задания начнем с загрузки необходимых библиотек для работы (рис. 53).

```

[1] # Отображение
import matplotlib.pyplot as plt

# Для работы с тензорами
import numpy as np

# Класс создания модели
from tensorflow.keras.models import Model

# Для загрузки данных
from tensorflow.keras.datasets import mnist

# Необходимые слои
from tensorflow.keras.layers import Input, Conv2DTranspose, MaxPooling2D, Conv2D, BatchNormalization, Dropout, Cropping2D, Concatenate, LeakyReLU

# Оптимизатор
from tensorflow.keras.optimizers import Adam

from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D

from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau, ModelCheckpoint

from tensorflow.keras.regularizers import l2

```

Рисунок 53. Импорт библиотек

Далее выполним загрузку данных (рис. 54).

```

[2] # Загрузка данных
(X_train, y_train), (X_test, y_test) = mnist.load_data()

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 2s 0us/step

```

Рисунок 54. Загрузка данных

Затем выполним нормализацию данных и изменение формы входных данных для подачи в сверточную нейронную сеть (рис. 55).

```

# Нормализация данных
X_train = X_train.astype('float32')/255.
X_test = X_test.astype('float32')/255.

# Изменение формы входных данных для подачи в сверточную нейронную сеть
X_train = X_train.reshape((-1, 28, 28, 1))
X_test = X_test.reshape((-1, 28, 28, 1))

```

Рисунок 55. Нормализация данных

Далее напишем функцию для добавления черных квадратов как сказано по заданию (рис. 56).

```

[4] # Функция для добавления черных квадратов
def add_black_squares(images, square_size=8):
    noisy_images = np.copy(images)
    for i in range(len(noisy_images)):
        x = np.random.randint(0, 28 - square_size)
        y = np.random.randint(0, 28 - square_size)
        noisy_images[i, x:x+square_size, y:y+square_size, :] = 0
    return noisy_images

```

Рисунок 56. Функция для добавления черных квадратов

Затем создадим зашумленные данные (рис. 57).

```

[5] # Создаем зашумленные данные
noisy_X_train = add_black_squares(X_train)
noisy_X_test = add_black_squares(X_test)

```

Рисунок 57. Создание зашумленных данных

После выполним визуализацию оригинальных и зашумленных изображений (рис. 58).

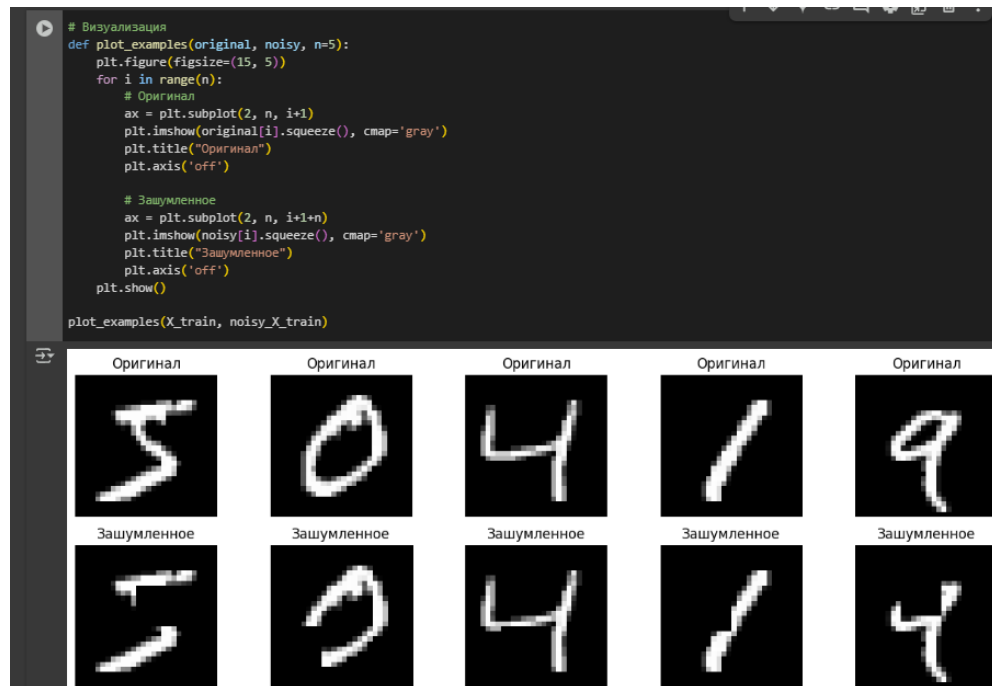


Рисунок 58. Визуализация оригинальных и зашумленных изображений

Далее выполним создание модели автокодировщика (рис. 59).

Выполним компиляцию модели и выведем архитектуру модели (рис. 60).

```

[9] # Архитектура автокодировщика
def create_advanced_autoencoder(input_shape):
    inputs = Input(shape=input_shape)

    # Энкодер
    x1 = Conv2D(64, (3, 3), activation='relu', padding='same')(inputs)
    x1 = BatchNormalization()(x1)
    x1 = LeakyReLU(alpha=0.1)(x1)
    p1 = MaxPooling2D((2, 2), padding='same')(x1)

    x2 = Conv2D(128, (3, 3), activation='relu', padding='same')(p1)
    x2 = BatchNormalization()(x2)
    x2 = LeakyReLU(alpha=0.1)(x2)
    p2 = MaxPooling2D((2, 2), padding='same')(x2)

    # Центральный блок
    x3 = Conv2D(256, (3, 3), activation='relu', padding='same')(p2)
    x3 = BatchNormalization()(x3)
    x3 = LeakyReLU(alpha=0.1)(x3)
    x3 = Dropout(0.3)(x3)

    # Декодер с правильными размерами для конкатенации
    u1 = UpSampling2D((2, 2))(x3)
    u1 = Conv2D(128, (2, 2), activation='relu', padding='same')(u1) # Изменение размера
    u1 = Concatenate()([u1, x2]) # Теперь размеры совпадают

    x4 = Conv2D(128, (3, 3), activation='relu', padding='same')(u1)
    x4 = BatchNormalization()(x4)
    x4 = LeakyReLU(alpha=0.1)(x4)

    u2 = UpSampling2D((2, 2))(x4)
    u2 = Conv2D(64, (2, 2), activation='relu', padding='same')(u2) # Изменение размера
    u2 = Concatenate()([u2, x1]) # Теперь размеры совпадают

    x5 = Conv2D(64, (3, 3), activation='relu', padding='same')(u2)
    x5 = BatchNormalization()(x5)
    x5 = LeakyReLU(alpha=0.1)(x5)

    outputs = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x5)

    model = Model(inputs, outputs)
    model.compile(optimizer=Adam(learning_rate=0.0005), loss='mse')
    return model

model = create_advanced_autoencoder((28, 28, 1))
model.summary()

```

Рисунок 59. Создание модели автокодировщика

Model: "functional_1"

Layer (type)	Output Shape	Param #	Connected to
input_layer_1 (InputLayer)	(None, 28, 28, 1)	0	-
conv2d_8 (Conv2D)	(None, 28, 28, 64)	640	input_layer_1[0][0]
batch_normalization_5 (BatchNormalization)	(None, 28, 28, 64)	256	conv2d_8[0][0]
leaky_re_lu_5 (LeakyReLU)	(None, 28, 28, 64)	0	batch_normalization_5_
max_pooling2d_2 (MaxPooling2D)	(None, 14, 14, 64)	0	leaky_re_lu_5[0][0]
conv2d_9 (Conv2D)	(None, 14, 14, 128)	73,856	max_pooling2d_2[0][0]
batch_normalization_6 (BatchNormalization)	(None, 14, 14, 128)	512	conv2d_9[0][0]
leaky_re_lu_6 (LeakyReLU)	(None, 14, 14, 128)	0	batch_normalization_6_
max_pooling2d_3 (MaxPooling2D)	(None, 7, 7, 128)	0	leaky_re_lu_6[0][0]
conv2d_10 (Conv2D)	(None, 7, 7, 256)	295,168	max_pooling2d_3[0][0]
batch_normalization_7 (BatchNormalization)	(None, 7, 7, 256)	1,024	conv2d_10[0][0]
leaky_re_lu_7 (LeakyReLU)	(None, 7, 7, 256)	0	batch_normalization_7_
dropout_1 (Dropout)	(None, 7, 7, 256)	0	leaky_re_lu_7[0][0]
up_sampling2d_2 (UpSampling2D)	(None, 14, 14, 256)	0	dropout_1[0][0]
conv2d_11 (Conv2D)	(None, 14, 14, 128)	131,200	up_sampling2d_2[0][0]
concatenate_2 (Concatenate)	(None, 14, 14, 256)	0	conv2d_11[0][0], leaky_re_lu_6[0][0]
conv2d_12 (Conv2D)	(None, 14, 14, 128)	295,040	concatenate_2[0][0]
batch_normalization_8 (BatchNormalization)	(None, 14, 14, 128)	512	conv2d_12[0][0]
leaky_re_lu_8 (LeakyReLU)	(None, 14, 14, 128)	0	batch_normalization_8_
up_sampling2d_3 (UpSampling2D)	(None, 28, 28, 128)	0	leaky_re_lu_8[0][0]
conv2d_13 (Conv2D)	(None, 28, 28, 64)	32,832	up_sampling2d_3[0][0]
concatenate_3 (Concatenate)	(None, 28, 28, 128)	0	conv2d_13[0][0], leaky_re_lu_5[0][0]
conv2d_14 (Conv2D)	(None, 28, 28, 64)	73,792	concatenate_3[0][0]
batch_normalization_9 (BatchNormalization)	(None, 28, 28, 64)	256	conv2d_14[0][0]
leaky_re_lu_9 (LeakyReLU)	(None, 28, 28, 64)	0	batch_normalization_9_
conv2d_15 (Conv2D)	(None, 28, 28, 1)	577	leaky_re_lu_9[0][0]

Total params: 905,665 (3.45 MB)
Trainable params: 904,385 (3.45 MB)
Non-trainable params: 1,280 (5.00 KB)

Рисунок 60. Архитектура модели

Далее пропишем коллбэки (рис. 61).

```
[10] # Коллбэки
callbacks = [
    EarlyStopping(monitor='val_loss', patience=25, restore_best_weights=True),
    ModelCheckpoint('best_model.h5', monitor='val_loss', save_best_only=True),
    ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=10, min_lr=1e-6)
]
```

Рисунок 61. Добавление коллбэков

Затем обучим модель восстанавливать оригинальные изображения из "зашумленных" квадратом изображений (рис. 62).

```
[11] # Обучение модели
history = model.fit(
    noisy_X_train, X_train,
    epochs=5,
    batch_size=256,
    shuffle=True,
    validation_data=(noisy_X_test, X_test),
    callbacks=callbacks
)

Epoch 1/5
235/235 — 0s 115ms/step - loss: 0.0267WARNING:absl:You are saving your model as an HDF5 file vi
235/235 — 49s 137ms/step - loss: 0.0266 - val_loss: 0.0758 - learning_rate: 5.0000e-04
Epoch 2/5
235/235 — 58s 87ms/step - loss: 1.5916e-04 - val_loss: 0.0876 - learning_rate: 5.0000e-04
Epoch 3/5
234/235 — 0s 85ms/step - loss: 6.2842e-05WARNING:absl:You are saving your model as an HDF5 file
235/235 — 41s 89ms/step - loss: 6.2764e-05 - val_loss: 2.3832e-04 - learning_rate: 5.0000e-04
Epoch 4/5
234/235 — 0s 83ms/step - loss: 3.4333e-05WARNING:absl:You are saving your model as an HDF5 file
235/235 — 21s 88ms/step - loss: 3.4300e-05 - val_loss: 1.2726e-05 - learning_rate: 5.0000e-04
Epoch 5/5
234/235 — 0s 81ms/step - loss: 2.1566e-05WARNING:absl:You are saving your model as an HDF5 file
235/235 — 41s 87ms/step - loss: 2.1550e-05 - val_loss: 7.6705e-06 - learning_rate: 5.0000e-04
```

Рисунок 62. Обучение модели

Далее посмотрим на MSE на тестовой выборке, она должна быть меньше 0,0070 (рис. 63).

```
[31] # Определение ошибки
noisy_X_test = add_black_squares(X_test) # Ваша функция
test_loss = model.evaluate(noisy_X_test, X_test, verbose=0)
print(f"New Test MSE: {test_loss:.7f}")

New Test MSE: 0.0033364
```

Рисунок 63. Вычисление MSE

Отсюда видно, что MSE составила 0,0033364, что меньше 0,0070, следовательно задание выполнено. Далее выполним визуализацию данных после обучения (рис. 64).



Рисунок 64. Визуализация данных после обучения

Ссылка на гитхаб с файлами: <https://github.com/EvgenyEvdakov/NS-9>

Вывод: в ходе выполнения лабораторной работы была изучена архитектура и принципы работы автокодировщиков, были разработаны и обучены модели для решения задач восстановления изображений, удаления шума, обнаружения аномалий и генерации данных.