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 №10

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)

Проверил:

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

_____

(подпись)

Ставрополь, 2025 г.

Тема: Разработка и исследование моделей семантической сегментации изображений на базе архитектур U-Net и PSPNet.

Цель: изучить методы семантической сегментации изображений с использованием сверточных нейронных сетей. Провести экспериментальное сравнение различных архитектур (SimpleUnet, U-Net, PSPNet) на примере базы изображений строительной тематики, а также выполнить модификацию модели под задачи с уменьшенным числом классов. Оценить качество сегментации и влияние различных параметров архитектуры на точность модели.

Ход работы:

Практика 1. Сегментация изображений (полное выполнение данной практики можно посмотреть в репозитории по ссылке в конце работы).

В данной практике рассматривается сегментация изображений. Задача состоит в составлении модели и обучении ее на наборе, состоящем из двух типов картинок:

- основное изображение;
- сегментированное изображение.

Обученная модель должна уметь находить различные классы на изображении и отмечать их разным цветом. Всего в базе содержатся объекты 16-ти различных классов.

Для этого выполняется импорт необходимых библиотек и загрузка базы изображений в разрешении 256x192. Далее выполняется распаковка архива и задаются параметры, также создаются функции загрузки выборки изображений из файлов в папке и функция для просмотра изображений из набора. Выполним загрузку входных изображений и их просмотр из обучающего набора (рис. 1).

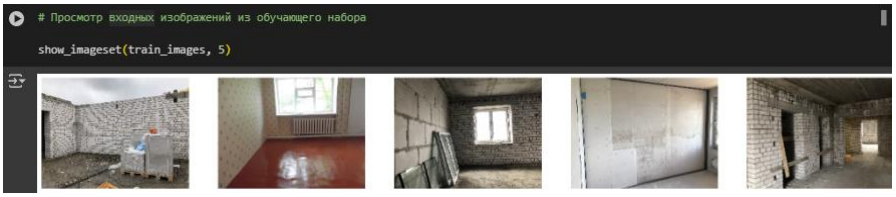


Рисунок 1. Просмотр входных изображений

Далее выполняется загрузка выходных (сегментированных) изображений и просмотр сегментированных изображений из обучающего набора (рис. 2).

Рисунок 2. Просмотр сегментированных изображений

Затем задаются цвета пикселей и все объединяется в один список меток классов. Далее необходимо выполнить создание выборки, для этого вначале рассмотрим сегментированное изображение (рис. 3).

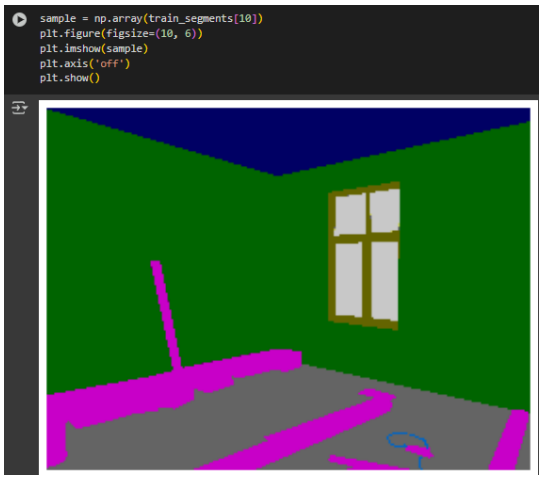


Рисунок 3. Сегментированное изображение

Далее выполняется создание нулевого выходного тензора и заполнение пикселей по цветам классов метками классов в выходном тензоре. Посмотрим на получившееся изображение (рис. 4).

Рисунок 4. Получившееся изображение

Отсюда была получена таже самая картинка, но в градациях серого, а объем выходного тензора стал в 16 (!) раз меньше.

Впоследствии добавляются две функции:

- перевод набора исходных сегментированых изображений в одноканальные изображения;
- перевод одноканальных изображений (результат работы модели) в цветные сегментированные изображения.

Далее происходит формирование проверочной и обучающей выборок. После формируется простая линейная модель нейронной сети, принимающую на вход оригинальное цветное изображение и возвращающую одноканальное изображение того же размера. Рассмотрим архитектуру созданной модели (рис. 5).

```
# Создание модели и вывод сводки по архитектуре

model_seq = sequential_segmentation_net(CLASS_COUNT,
                                        (IMG_WIDTH, IMG_HEIGHT, 3))

model_seq.summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 192, 256, 3)	0
block1_conv1 (Conv2D)	(None, 192, 256, 128)	3,584
batch_normalization (BatchNormalization)	(None, 192, 256, 128)	512
activation (Activation)	(None, 192, 256, 128)	0
block1_conv3 (Conv2D)	(None, 192, 256, 64)	73,792
batch_normalization_1 (BatchNormalization)	(None, 192, 256, 64)	256
activation_1 (Activation)	(None, 192, 256, 64)	0
block1_conv4 (Conv2D)	(None, 192, 256, 32)	18,464
batch_normalization_2 (BatchNormalization)	(None, 192, 256, 32)	128
activation_2 (Activation)	(None, 192, 256, 32)	0
block1_conv5 (Conv2D)	(None, 192, 256, 16)	4,624
batch_normalization_3 (BatchNormalization)	(None, 192, 256, 16)	64
activation_3 (Activation)	(None, 192, 256, 16)	0
conv2d (Conv2D)	(None, 192, 256, 16)	2,320

Total params: 103,744 (405.25 KB)
Trainable params: 103,264 (403.38 KB)
Non-trainable params: 480 (1.88 KB)

Рисунок 5. Архитектура созданной модели

Далее выполняется обучение созданной модели. Посмотрим на график процесса обучения модели (рис. 6).

Рисунок 6. График процесса обучения модели

Затем рассмотрим распознавание изображений, для этого определим функцию визуализации результатов работы модели. И выполним отображение результатов работы модели (рис. 7).

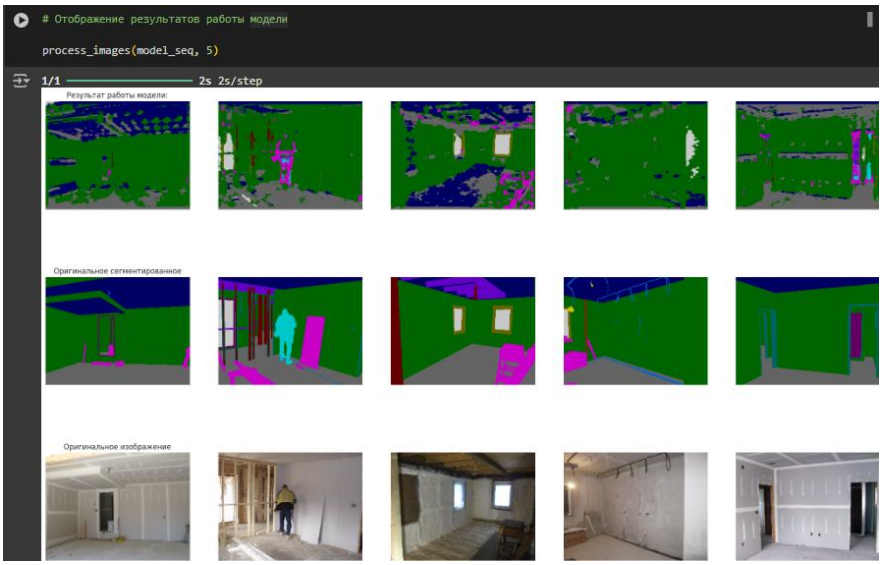


Рисунок 7. Отображение результатов работы модели

Отсюда можно сказать, что мы получили не самые плохие результаты. Где-то модель смогла обнаружить искомые объекты, где-то ошиблась. Но для первого эксперимента вполне достойно.

На следующем шаге мы уменьшим количество классов, объединив некоторые классы в один. Для этого посчитаем, сколько пикселей каждого класса есть в исходном датасете. Выполним отрисовку столбчатой диаграммы наполненности классов (рис. 8).

```
# Отрисовка столбчатой диаграммы наполненности классов

plt.figure(figsize = (10, 5))
plt.bar(np.arange(CLASS_COUNT), pix_count)
plt.show()
```

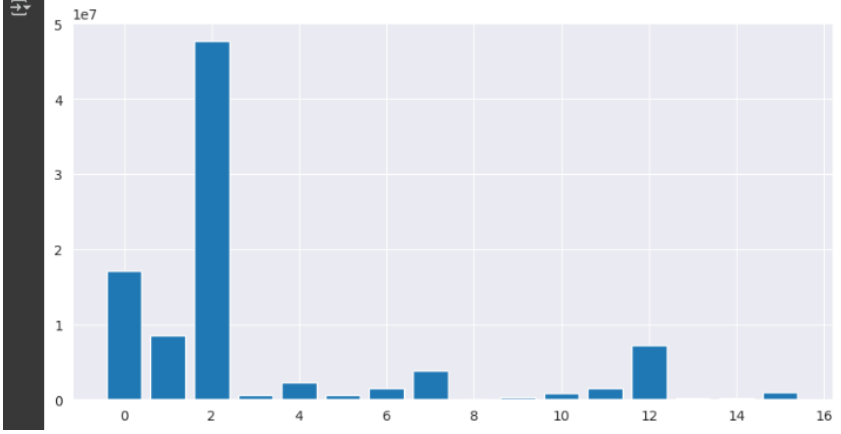


Рисунок 8. Отрисовка столбчатой диаграммы наполненности классов

Отсюда видно, что в нашей базе в большей степени присутствует класс 2: стены, классы, а пол, потолок и инвентарь представлены значительно меньше. Остальных классов совсем мало. Попробуем создать модель для сегментации на 5 классов: Стены; Пол; Потолок; Инвентарь; Остальное.

Для этого слегка изменим y_train и y_val, объединив соответствующие классы. Создадим, обучим и проверим новую модель той же архитектуре на исправленных данных (рис. 9).

```
# Создание модели для уменьшенного набора классов

model_seq5 = sequential_segmentation_net(5,
                                         (IMG_WIDTH, IMG_HEIGHT, 3))
model_seq5.summary()
```

Model: "functional_1"

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 192, 256, 3)	0
block1_conv1 (Conv2D)	(None, 192, 256, 128)	3,584
batch_normalization_4 (BatchNormalization)	(None, 192, 256, 128)	512
activation_4 (Activation)	(None, 192, 256, 128)	0
block1_conv3 (Conv2D)	(None, 192, 256, 64)	73,792
batch_normalization_5 (BatchNormalization)	(None, 192, 256, 64)	256
activation_5 (Activation)	(None, 192, 256, 64)	0
block1_conv4 (Conv2D)	(None, 192, 256, 32)	18,464
batch_normalization_6 (BatchNormalization)	(None, 192, 256, 32)	128
activation_6 (Activation)	(None, 192, 256, 32)	0
block1_conv5 (Conv2D)	(None, 192, 256, 16)	4,624
batch_normalization_7 (BatchNormalization)	(None, 192, 256, 16)	64
activation_7 (Activation)	(None, 192, 256, 16)	0
conv2d_1 (Conv2D)	(None, 192, 256, 5)	725

Total params: 102,149 (399.02 KB)
Trainable params: 101,669 (397.14 KB)
Non-trainable params: 480 (1.88 KB)

Рисунок 9. Создание модели

Далее выполним обучение модели. Посмотрим на график процесса обучения модели (рис. 10).

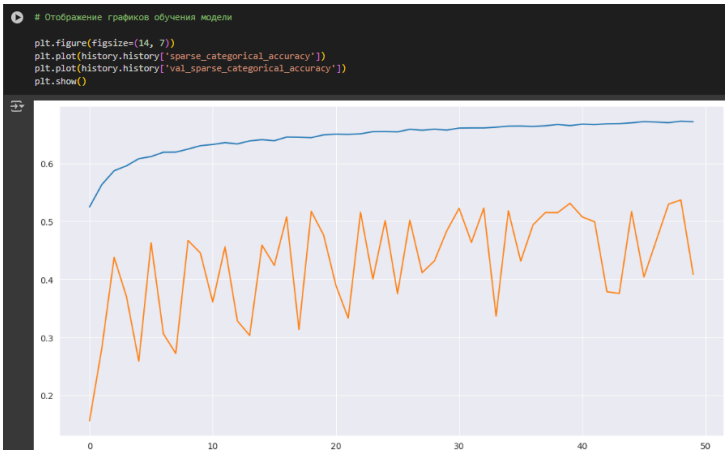


Рисунок 10. График процесса обучения модели

И также выполним отображение результатов работы модели (рис. 11).

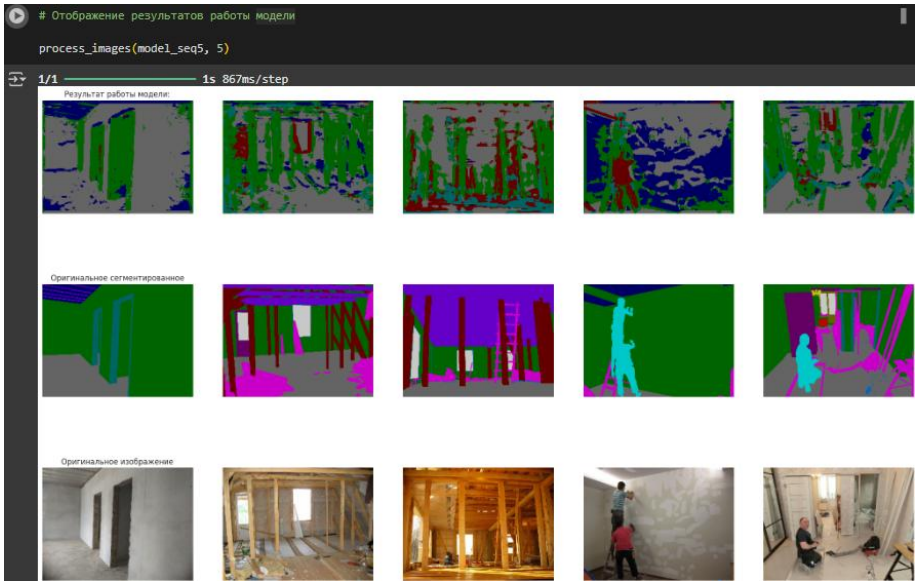


Рисунок 11. Отображение результатов работы модели

По численным значениям мы получаем чуть лучшие результаты, но в целом можно сказать, что у обеих моделей есть предел точности и они не могут "схватить" необходимые отличия классов.

Практика 2. Сегментация изображений (полное выполнение данной практики можно посмотреть в репозитории по ссылке в конце работы).

В данной практике мы также работаем с базой "Стройка". Построим модель архитектуры U-Net в различных вариациях. Решать задачу будем для полной базы на 16 классов.

Вначале повторим шаги из первого практического ноутбука с загрузкой библиотек и формированием выборок.

U-Net - сверточная нейронная сеть, разработанная Олафом Роннебергером, Филиппом Фишером, Томасом Броксом для выполнения семантической сегментации биомедицинских изображений в 2015 году и описанная в статье «U-Net: сверточные сети для сегментации биомедицинских изображений».

Архитектура U-Net оптимизирована для обеспечения наилучшей сегментации с меньшим количеством обучающих данных. Она построена без полносвязных слоев и относится к типу полностью сверточных сетей (FCNN) (рис. 12).

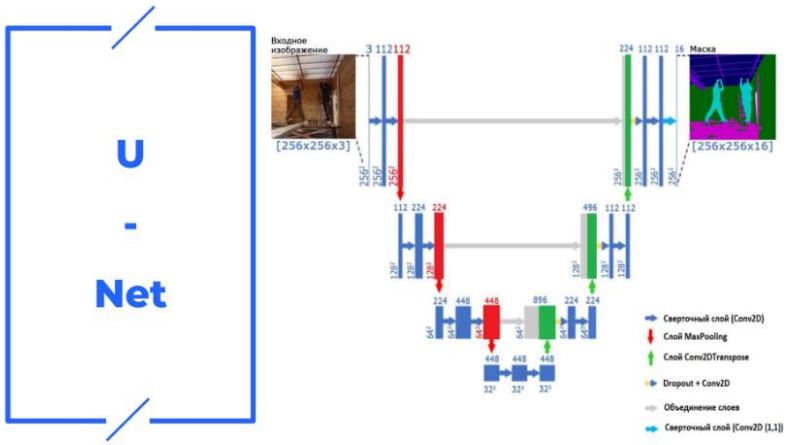
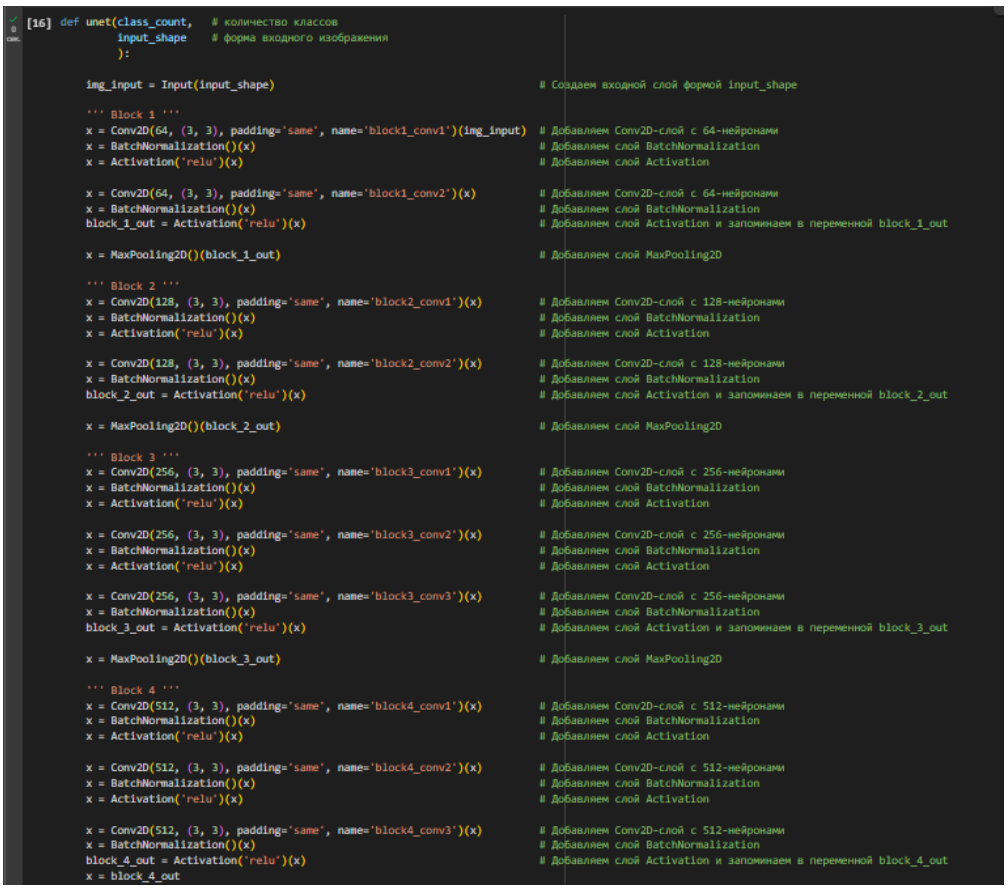


Рисунок 12. Архитектура U-Net

Архитектура U-Net представляет собой U-образную сеть, состоящую из трех частей:

- Путь сужения / понижения дискретизации;
- Узкое место - "бутылочное горлышко";
- Путь расширения / увеличения дискретизации.

Далее создадим модель U-Net (рис. 13 - 14).



```
[16] def unet(class_count,   # количество классов
              input_shape    # форма входного изображения
              ):

        img_input = Input(input_shape)                                          # Создаем входной слой формой input_shape

        ''' Block 1 '''
        x = Conv2D(64, (3, 3), padding='same', name='block1_conv1')(img_input)  # Добавляем Conv2D-слой с 64-нейронами
        x = BatchNormalization()(x)                                             # Добавляем слой BatchNormalization
        x = Activation('relu')(x)                                               # Добавляем слой Activation

        x = Conv2D(64, (3, 3), padding='same', name='block1_conv2')(x)          # Добавляем Conv2D-слой с 64-нейронами
        x = BatchNormalization()(x)                                             # Добавляем слой BatchNormalization
        block_1_out = Activation('relu')(x)                                     # Добавляем слой Activation и запоминаем в переменной block_1_out

        x = MaxPooling2D()(block_1_out)                                         # Добавляем слой MaxPooling2D

        ''' Block 2 '''
        x = Conv2D(128, (3, 3), padding='same', name='block2_conv1')(x)         # Добавляем Conv2D-слой с 128-нейронами
        x = BatchNormalization()(x)                                             # Добавляем слой BatchNormalization
        x = Activation('relu')(x)                                               # Добавляем слой Activation

        x = Conv2D(128, (3, 3), padding='same', name='block2_conv2')(x)         # Добавляем Conv2D-слой с 128-нейронами
        x = BatchNormalization()(x)                                             # Добавляем слой BatchNormalization
        block_2_out = Activation('relu')(x)                                     # Добавляем слой Activation и запоминаем в переменной block_2_out

        x = MaxPooling2D()(block_2_out)                                         # Добавляем слой MaxPooling2D

        ''' Block 3 '''
        x = Conv2D(256, (3, 3), padding='same', name='block3_conv1')(x)         # Добавляем Conv2D-слой с 256-нейронами
        x = BatchNormalization()(x)                                             # Добавляем слой BatchNormalization
        x = Activation('relu')(x)                                               # Добавляем слой Activation

        x = Conv2D(256, (3, 3), padding='same', name='block3_conv2')(x)         # Добавляем Conv2D-слой с 256-нейронами
        x = BatchNormalization()(x)                                             # Добавляем слой BatchNormalization
        x = Activation('relu')(x)                                               # Добавляем слой Activation

        x = Conv2D(256, (3, 3), padding='same', name='block3_conv3')(x)         # Добавляем Conv2D-слой с 256-нейронами
        x = BatchNormalization()(x)                                             # Добавляем слой BatchNormalization
        block_3_out = Activation('relu')(x)                                     # Добавляем слой Activation и запоминаем в переменной block_3_out

        x = MaxPooling2D()(block_3_out)                                         # Добавляем слой MaxPooling2D

        ''' Block 4 '''
        x = Conv2D(512, (3, 3), padding='same', name='block4_conv1')(x)         # Добавляем Conv2D-слой с 512-нейронами
        x = BatchNormalization()(x)                                             # Добавляем слой BatchNormalization
        x = Activation('relu')(x)                                               # Добавляем слой Activation

        x = Conv2D(512, (3, 3), padding='same', name='block4_conv2')(x)         # Добавляем Conv2D-слой с 512-нейронами
        x = BatchNormalization()(x)                                             # Добавляем слой BatchNormalization
        x = Activation('relu')(x)                                               # Добавляем слой Activation

        x = Conv2D(512, (3, 3), padding='same', name='block4_conv3')(x)         # Добавляем Conv2D-слой с 512-нейронами
        x = BatchNormalization()(x)                                             # Добавляем слой BatchNormalization
        block_4_out = Activation('relu')(x)                                     # Добавляем слой Activation и запоминаем в переменной block_4_out
        x = block_4_out
```

Рисунок 13. Первая часть кода создания модели

```
        ''' UP 2 '''
        x = Conv2DTranspose(256, (2, 2), strides=(2, 2), padding='same')(x)     # Добавляем слой Conv2DTranspose с 256 нейронами
        x = BatchNormalization()(x)                                             # Добавляем слой BatchNormalization
        x = Activation('relu')(x)                                               # Добавляем слой Activation

        x = concatenate([x, block_3_out])                                       # Объединяем текущий слой со слоем block_3_out
        x = Conv2D(256, (3, 3), padding='same')(x)                              # Добавляем слой Conv2D с 256 нейронами
        x = BatchNormalization()(x)                                             # Добавляем слой BatchNormalization
        x = Activation('relu')(x)                                               # Добавляем слой Activation

        x = Conv2D(256, (3, 3), padding='same')(x)
        x = BatchNormalization()(x)                                             # Добавляем слой BatchNormalization
        x = Activation('relu')(x)                                               # Добавляем слой Activation

        ''' UP 3 '''
        x = Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='same')(x)     # Добавляем слой Conv2DTranspose с 128 нейронами
        x = BatchNormalization()(x)                                             # Добавляем слой BatchNormalization
        x = Activation('relu')(x)                                               # Добавляем слой Activation

        x = concatenate([x, block_2_out])                                       # Объединяем текущий слой со слоем block_2_out
        x = Conv2D(128, (3, 3), padding='same')(x)                              # Добавляем слой Conv2D с 128 нейронами
        x = BatchNormalization()(x)                                             # Добавляем слой BatchNormalization
        x = Activation('relu')(x)                                               # Добавляем слой Activation

        x = Conv2D(128, (3, 3), padding='same')(x)                              # Добавляем слой Conv2D с 128 нейронами
        x = BatchNormalization()(x)                                             # Добавляем слой BatchNormalization
        x = Activation('relu')(x)                                               # Добавляем слой Activation

        ''' UP 4 '''
        x = Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same')(x)      # Добавляем слой Conv2DTranspose с 64 нейронами
        x = BatchNormalization()(x)                                             # Добавляем слой BatchNormalization
        x = Activation('relu')(x)                                               # Добавляем слой Activation

        x = concatenate([x, block_1_out])                                       # Объединяем текущий слой со слоем block_1_out
        x = Conv2D(64, (3, 3), padding='same')(x)                               # Добавляем слой Conv2D с 64 нейронами
        x = BatchNormalization()(x)                                             # Добавляем слой BatchNormalization
        x = Activation('relu')(x)                                               # Добавляем слой Activation

        x = Conv2D(64, (3, 3), padding='same')(x)                               # Добавляем слой Conv2D с 64 нейронами
        x = BatchNormalization()(x)                                             # Добавляем слой BatchNormalization
        x = Activation('relu')(x)                                               # Добавляем слой Activation

        x = Conv2D(class_count, (3, 3), activation='softmax', padding='same')(x)  # Добавляем Conv2D-Слой с softmax-активацией на class_count-нейронов

        model = Model(img_input, x)                                             # Создаем модель с входом 'img_input' и выходом 'x'

        # Компилируем модель
        model.compile(optimizer=Adam(learning_rate=1e-3),
                      loss='sparse_categorical_crossentropy',
                      metrics=['sparse_categorical_accuracy'])

        # Возвращаем сформированную модель
        return model
```

Рисунок 14. Вторая часть кода создания модели

Затем выполним обучение модели и отображение графика процесса обучения модели (рис. 15).

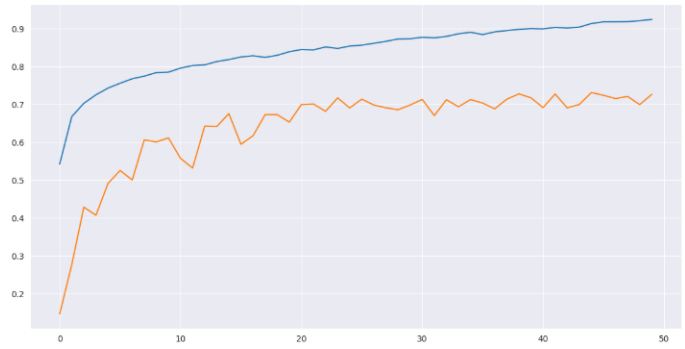


Рисунок 15. График процесса обучения модели

График обучения позволяет заключить, что имеет смысл продолжить обучение модели, поскольку рост точности продолжается. Далее дообучим модель еще на 50 эпох и посмотрим на график процесса обучения (рис. 16).

Рисунок 16. График процесса обучения модели

Выполним отображение результатов работы модели (рис. 17).

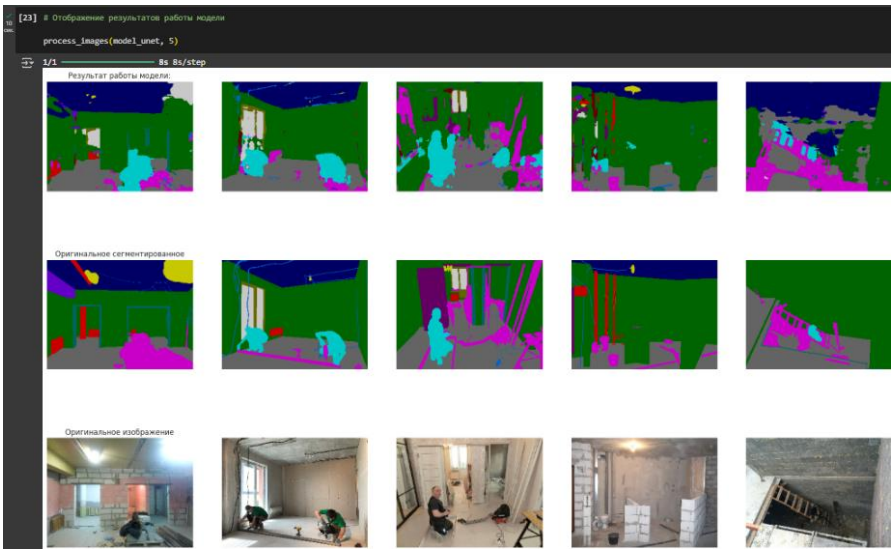


Рисунок 17. Отображение результатов работы модели

Отсюда получен Стандартная модель U-Net способна находить контуры и позиции объектов. А в некоторых случаях даже довольно точно обнаруживать элементы.

Далее рассматривается упрощенная архитектура U-net. Выполним создание модели и вывод архитектуры данной модели (рис. 18).

```
[25] # Создание упрощенной модели и вывод сводки

model_simple_unet = simple_unet(CLASS_COUNT,
                                (IMG_WIDTH, IMG_HEIGHT, 3))

model_simple_unet.summary()
```

Model: "functional_1"

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 192, 256, 3)	0
block1_conv1 (Conv2D)	(None, 192, 256, 32)	896
activation_19 (Activation)	(None, 192, 256, 32)	0
block1_conv2 (Conv2D)	(None, 192, 256, 32)	9,248
activation_20 (Activation)	(None, 192, 256, 32)	0
max_pooling2d_3 (MaxPooling2D)	(None, 48, 64, 32)	0
block2_conv1 (Conv2D)	(None, 48, 64, 64)	18,496
activation_21 (Activation)	(None, 48, 64, 64)	0
block2_conv2 (Conv2D)	(None, 48, 64, 64)	36,928
activation_22 (Activation)	(None, 48, 64, 64)	0
max_pooling2d_4 (MaxPooling2D)	(None, 12, 16, 64)	0
conv2d_transpose_3 (Conv2DTranspose)	(None, 48, 64, 64)	65,600
activation_23 (Activation)	(None, 48, 64, 64)	0
conv2d_7 (Conv2D)	(None, 48, 64, 64)	36,928
activation_24 (Activation)	(None, 48, 64, 64)	0
conv2d_8 (Conv2D)	(None, 48, 64, 64)	36,928
activation_25 (Activation)	(None, 48, 64, 64)	0
conv2d_transpose_4 (Conv2DTranspose)	(None, 192, 256, 32)	32,800
activation_26 (Activation)	(None, 192, 256, 32)	0
conv2d_9 (Conv2D)	(None, 192, 256, 32)	9,248
activation_27 (Activation)	(None, 192, 256, 32)	0
conv2d_10 (Conv2D)	(None, 192, 256, 32)	9,248
activation_28 (Activation)	(None, 192, 256, 32)	0
conv2d_11 (Conv2D)	(None, 192, 256, 16)	4,624

Total params: 260,844 (1019.31 KB)
Trainable params: 260,844 (1019.31 KB)
Non-trainable params: 0 (0.00 B)

Рисунок 18. Архитектура модели

Затем выполним обучение модели и посмотрим на график процесса обучения данной модели (рис. 19).

Рисунок 19. График процесса обучения модели

Выполним отображение результатов работы модели (рис. 20).

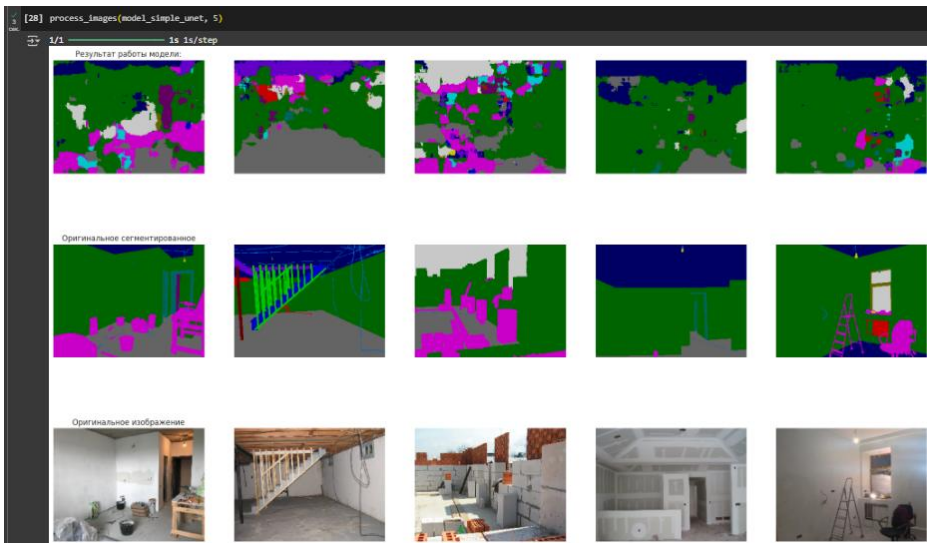
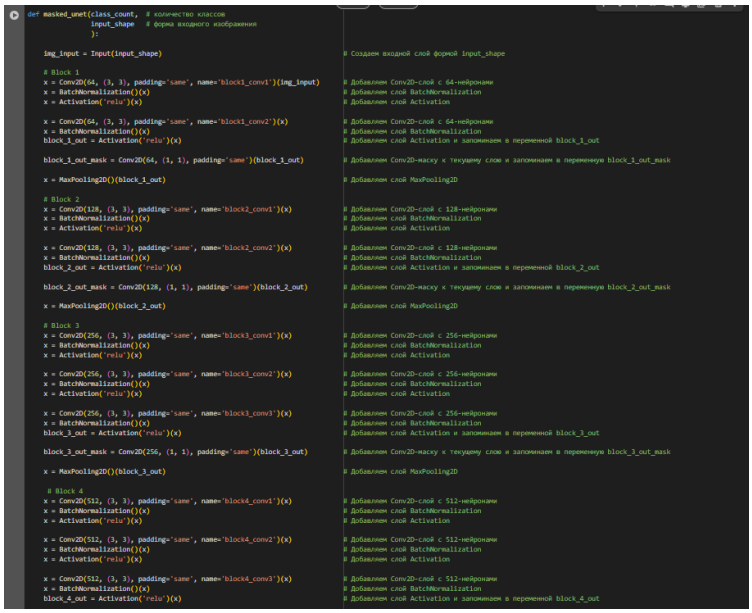


Рисунок 20. Отображение результатов работы модели

Отсюда видно, что модель обучилась довольно быстро, но результаты заметно хуже первой версии.

Далее рассматривается расширенная архитектура U-net. Выполним создание модели и вывод архитектуры данной модели (рис. 21 - 23).



```
def masked_unet(class_count,  # количество классов
                input_shape   # форма входного изображения
                ):

    img_input = Input(input_shape)                                  # Создаем входной слой формой input_shape

    # Block 1
    x = Conv2D(64, (3, 3), padding='same', name='block1_conv1')(img_input)  # Добавляем Conv2D-слой с 64-нейронами
    x = BatchNormalization()(x)                                     # Добавляем слой BatchNormalization
    x = Activation('relu')(x)                                       # Добавляем слой Activation

    x = Conv2D(64, (3, 3), padding='same', name='block1_conv2')(x)  # Добавляем Conv2D-слой с 64-нейронами
    x = BatchNormalization()(x)                                     # Добавляем слой BatchNormalization
    block_1_out = Activation('relu')(x)                             # Добавляем слой Activation и запоминаем в переменной block_1_out

    block_1_out_mask = Conv2D(64, (1, 1), padding='same')(block_1_out)  # Добавляем Conv2D-маску к текущему слою и запоминаем в переменную block_1_out_mask

    x = MaxPooling2D()(block_1_out)                                 # Добавляем слой MaxPooling2D

    # Block 2
    x = Conv2D(128, (3, 3), padding='same', name='block2_conv1')(x) # Добавляем Conv2D-слой с 128-нейронами
    x = BatchNormalization()(x)                                     # Добавляем слой BatchNormalization
    x = Activation('relu')(x)                                       # Добавляем слой Activation

    x = Conv2D(128, (3, 3), padding='same', name='block2_conv2')(x) # Добавляем Conv2D-слой с 128-нейронами
    x = BatchNormalization()(x)                                     # Добавляем слой BatchNormalization
    block_2_out = Activation('relu')(x)                             # Добавляем слой Activation и запоминаем в переменной block_2_out

    block_2_out_mask = Conv2D(128, (1, 1), padding='same')(block_2_out)  # Добавляем Conv2D-маску к текущему слою и запоминаем в переменную block_2_out_mask

    x = MaxPooling2D()(block_2_out)                                 # Добавляем слой MaxPooling2D

    # Block 3
    x = Conv2D(256, (3, 3), padding='same', name='block3_conv1')(x) # Добавляем Conv2D-слой с 256-нейронами
    x = BatchNormalization()(x)                                     # Добавляем слой BatchNormalization
    x = Activation('relu')(x)                                       # Добавляем слой Activation

    x = Conv2D(256, (3, 3), padding='same', name='block3_conv2')(x) # Добавляем Conv2D-слой с 256-нейронами
    x = BatchNormalization()(x)                                     # Добавляем слой BatchNormalization
    x = Activation('relu')(x)                                       # Добавляем слой Activation

    x = Conv2D(256, (3, 3), padding='same', name='block3_conv3')(x) # Добавляем Conv2D-слой с 256-нейронами
    x = BatchNormalization()(x)                                     # Добавляем слой BatchNormalization
    block_3_out = Activation('relu')(x)                             # Добавляем слой Activation и запоминаем в переменной block_3_out

    block_3_out_mask = Conv2D(256, (1, 1), padding='same')(block_3_out)  # Добавляем Conv2D-маску к текущему слою и запоминаем в переменную block_3_out_mask

    x = MaxPooling2D()(block_3_out)                                 # Добавляем слой MaxPooling2D

     # Block 4
    x = Conv2D(512, (3, 3), padding='same', name='block4_conv1')(x) # Добавляем Conv2D-слой с 512-нейронами
    x = BatchNormalization()(x)                                     # Добавляем слой BatchNormalization
    x = Activation('relu')(x)                                       # Добавляем слой Activation

    x = Conv2D(512, (3, 3), padding='same', name='block4_conv2')(x) # Добавляем Conv2D-слой с 512-нейронами
    x = BatchNormalization()(x)                                     # Добавляем слой BatchNormalization
    x = Activation('relu')(x)                                       # Добавляем слой Activation

    x = Conv2D(512, (3, 3), padding='same', name='block4_conv3')(x) # Добавляем Conv2D-слой с 512-нейронами
    x = BatchNormalization()(x)                                     # Добавляем слой BatchNormalization
    block_4_out = Activation('relu')(x)                             # Добавляем слой Activation и запоминаем в переменной block_4_out
```

Рисунок 21. Первая часть кода создания модели

```
x = MaxPooling2D()(block_4_out)                                  # Добавляем слой MaxPooling2D

# Block 5
x = Conv2D(512, (3, 3), padding='same', name='block5_conv1')(x)  # Добавляем Conv2D-слой с 512-нейронами
x = BatchNormalization()(x)                                      # Добавляем слой BatchNormalization
x = Activation('relu')(x)                                        # Добавляем слой Activation

x = Conv2D(512, (3, 3), padding='same', name='block5_conv2')(x)  # Добавляем Conv2D-слой с 512-нейронами
x = BatchNormalization()(x)                                      # Добавляем слой BatchNormalization
x = Activation('relu')(x)                                        # Добавляем слой Activation

x = Conv2D(512, (3, 3), padding='same', name='block5_conv3')(x)  # Добавляем Conv2D-слой с 512-нейронами
x = BatchNormalization()(x)                                      # Добавляем слой BatchNormalization
x = Activation('relu')(x)                                        # Добавляем слой Activation

for_pretrained_weight = MaxPooling2D()(x)                        # Добавляем слой MaxPooling2D

# UP 1
x = Conv2DTranspose(512, (2, 2), strides=(2, 2), padding='same')(x)  # Добавляем слой Conv2DTranspose с 512 нейронами
x = BatchNormalization()(x)                                          # Добавляем слой BatchNormalization
x = Activation('relu')(x)                                            # Добавляем слой Activation

x = concatenate([x, block_4_out, block_4_out_mask])              # Объединяем текущий слой со слоем block_4_out и слоем-маской block_4_out_mask
x = Conv2D(512, (3, 3), padding='same')(x)                       # Добавляем слой Conv2D с 512 нейронами
x = BatchNormalization()(x)                                      # Добавляем слой BatchNormalization
x = Activation('relu')(x)                                        # Добавляем слой Activation

x = Conv2D(512, (3, 3), padding='same')(x)                       # Добавляем слой Conv2D с 512 нейронами
x = BatchNormalization()(x)                                      # Добавляем слой BatchNormalization
x = Activation('relu')(x)                                        # Добавляем слой Activation

# UP 2
x = Conv2DTranspose(256, (2, 2), strides=(2, 2), padding='same')(x)  # Добавляем слой Conv2DTranspose с 256 нейронами
x = BatchNormalization()(x)                                          # Добавляем слой BatchNormalization
x = Activation('relu')(x)                                            # Добавляем слой Activation

x = concatenate([x, block_3_out, block_3_out_mask])              # Объединяем текущий слой со слоем block_3_out и слоем-маской block_3_out_mask
x = Conv2D(256, (3, 3), padding='same')(x)                       # Добавляем слой Conv2D с 256 нейронами
x = BatchNormalization()(x)                                      # Добавляем слой BatchNormalization
x = Activation('relu')(x)                                        # Добавляем слой Activation

x = Conv2D(256, (3, 3), padding='same')(x)                       # Добавляем слой Conv2D с 256 нейронами
x = BatchNormalization()(x)                                      # Добавляем слой BatchNormalization
x = Activation('relu')(x)                                        # Добавляем слой Activation

# UP 3
x = Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='same')(x)  # Добавляем слой Conv2DTranspose с 128 нейронами
x = BatchNormalization()(x)                                          # Добавляем слой BatchNormalization
x = Activation('relu')(x)                                            # Добавляем слой Activation

x = concatenate([x, block_2_out, block_2_out_mask])              # Объединяем текущий слой со слоем block_2_out и слоем-маской block_2_out_mask
x = Conv2D(128, (3, 3), padding='same')(x)                       # Добавляем слой Conv2D с 128 нейронами
x = BatchNormalization()(x)                                      # Добавляем слой BatchNormalization
x = Activation('relu')(x)                                        # Добавляем слой Activation

x = Conv2D(128, (3, 3), padding='same')(x)                       # Добавляем слой Conv2D с 128 нейронами
x = BatchNormalization()(x)                                      # Добавляем слой BatchNormalization
x = Activation('relu')(x)                                        # Добавляем слой Activation
```

Рисунок 22. Вторая часть кода создания модели

```
# UP 4
x = Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same')(x)  # Добавляем слой Conv2DTranspose с 64 нейронами
x = BatchNormalization()(x)                                         # Добавляем слой BatchNormalization
x = Activation('relu')(x)                                           # Добавляем слой Activation

x = concatenate([x, block_1_out, block_1_out_mask])             # Объединяем текущий слой со слоем block_1_out и слоем-маской block_1_out_mask
x = Conv2D(64, (3, 3), padding='same')(x)                       # Добавляем слой Conv2D с 128 нейронами
x = BatchNormalization()(x)                                     # Добавляем слой BatchNormalization
x = Activation('relu')(x)                                       # Добавляем слой Activation

x = Conv2D(64, (3, 3), padding='same')(x)                       # Добавляем слой Conv2D с 128 нейронами
x = BatchNormalization()(x)                                     # Добавляем слой BatchNormalization
x = Activation('relu')(x)                                       # Добавляем слой Activation

x = Conv2D(class_count, (3, 3), activation='softmax', padding='same')(x)  # Добавляем Conv2D-Слой с softmax-активацией на class_count-нейронов

model = Model(img_input, x)                                     # Создаем модель с входом 'img_input' и выходом 'x'

# Компилируем модель
model.compile(optimizer=Adam(learning_rate=1e-3),
              loss='sparse_categorical_crossentropy',
              metrics=['sparse_categorical_accuracy'])

# Возвращаем сформированную модель
return model
```

Рисунок 23. Третья часть кода создания модели

Затем выполним обучение модели и посмотрим на график процесса обучения данной модели (рис. 24).

```
# Отображение графиков обучения модели

plt.figure(figsize=(14, 7))
plt.plot(history.history['sparse_categorical_accuracy'])
plt.plot(history.history['val_sparse_categorical_accuracy'])
plt.show()
```

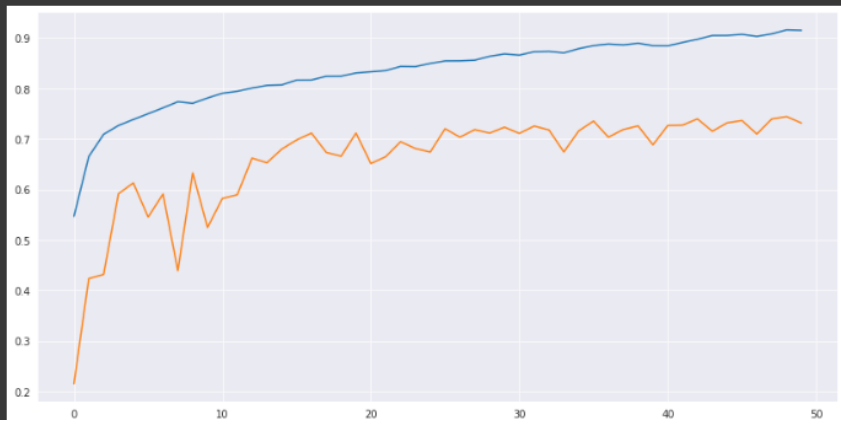


Рисунок 24. График процесса обучения модели

Выполним отображение результатов работы модели (рис. 25).

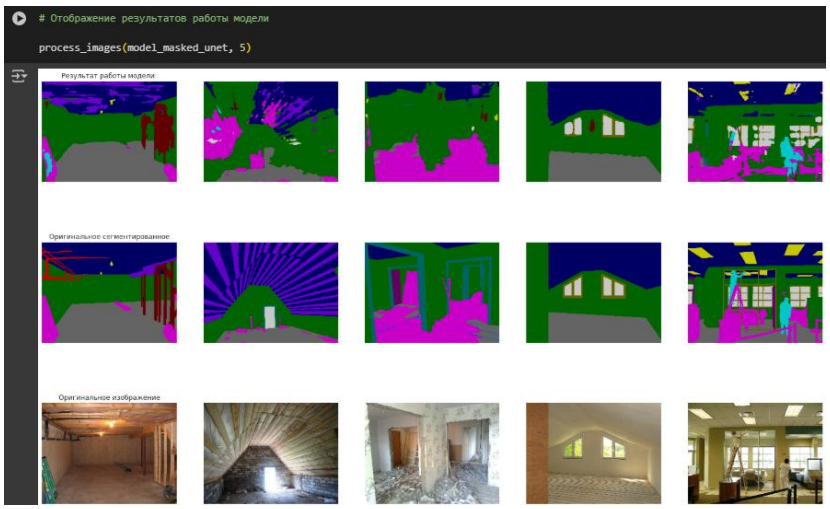


Рисунок 25. Отображение результатов работы модели

Практика 3. Сегментация изображений (полное выполнение данной практики можно посмотреть в репозитории по ссылке в конце работы).

В данной практике мы создадим сегментриующую модель на базе изображений самолетов. Эта задача значительно проще предыдущей, поскольку на оригинальном изображении будет всего два класса объектов: самолет и фон.

Вначале повторим шаги из первого практического ноутбука с загрузкой библиотек и формированием выборок. Далее после формирования выборок, рассмотрим простую линейную (последовательную) архитектуру модели. Выполним создание модели и посмотрим на ее архитектуру (рис. 26).

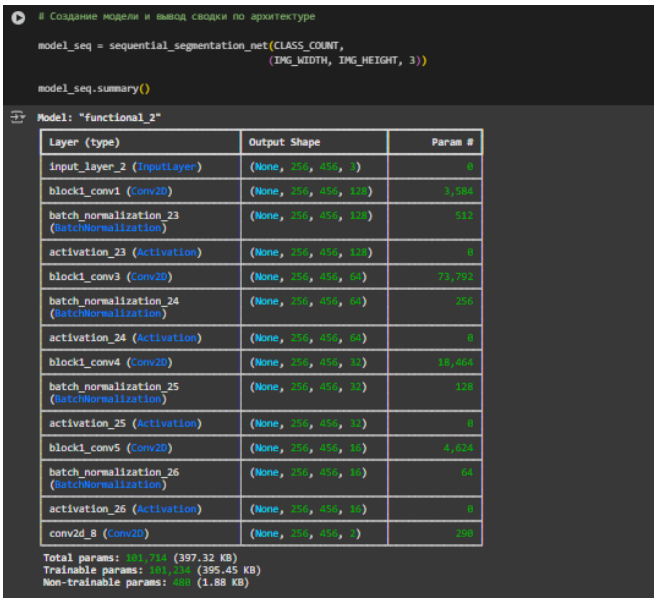


Рисунок 26. Архитектура модели

Затем выполним обучение созданной модели и посмотрим на график процесса ее обучения (рис. 27).

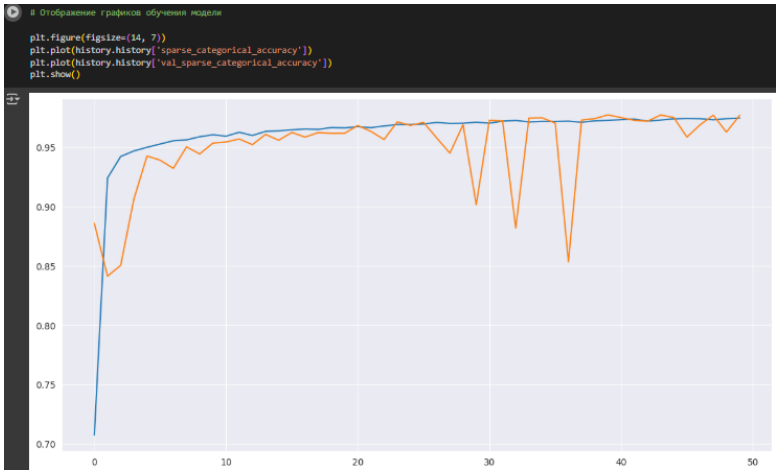


Рисунок 27. График процесса обучения

Отсюда, были получены довольно неплохие результаты. 0.97 - очень хороший показатель для проверочной выборки.

Теперь взглянем на результат работы нашей модели, для этого выполним функцию визуализации процесса сегментации изображений, модифицированная для вывода по списку индексов и выполним отображение результатов работы модели (рис. 28).

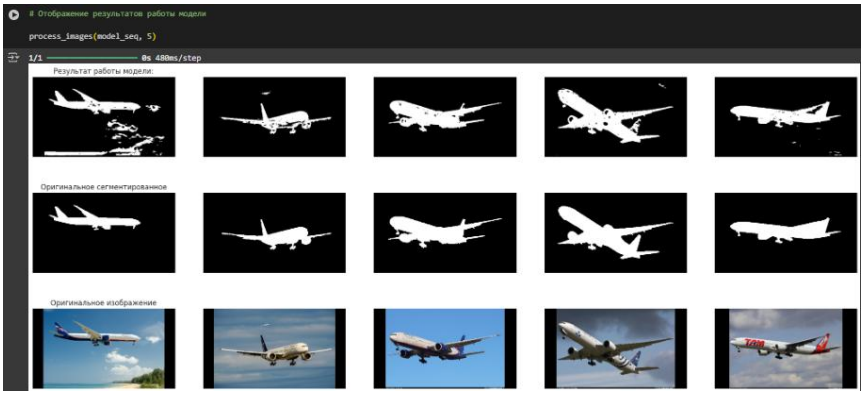


Рисунок 28. Отображение работы модели

Результат не плохой, теперь, мы вправе расчитывать, что получим отличные результаты с более мощной архитектурой, например, U-Net.

Далее рассмотрим архитектуру U-Net. Для этого создадим модель такую же как и во второй практике. Выполним обучение модели и посмотрим на график процесса обучения модели (рис. 29).

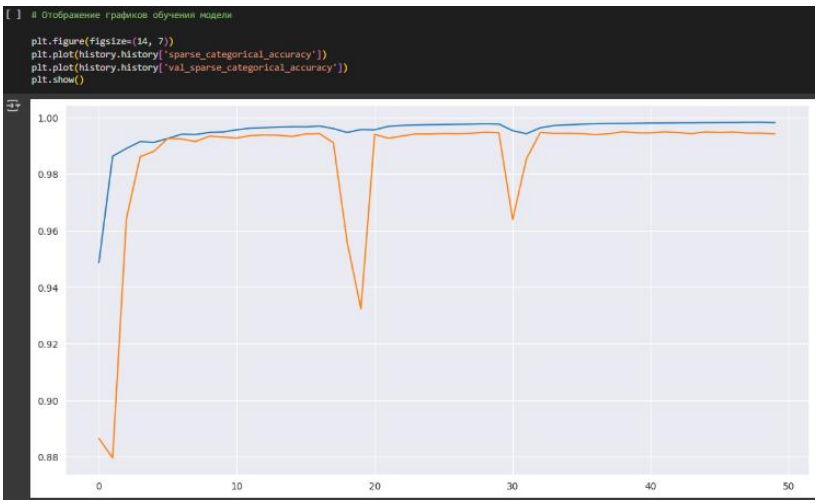


Рисунок 29. График процесса обучения модели

Выполним отображение результатов работы модели (рис. 30).

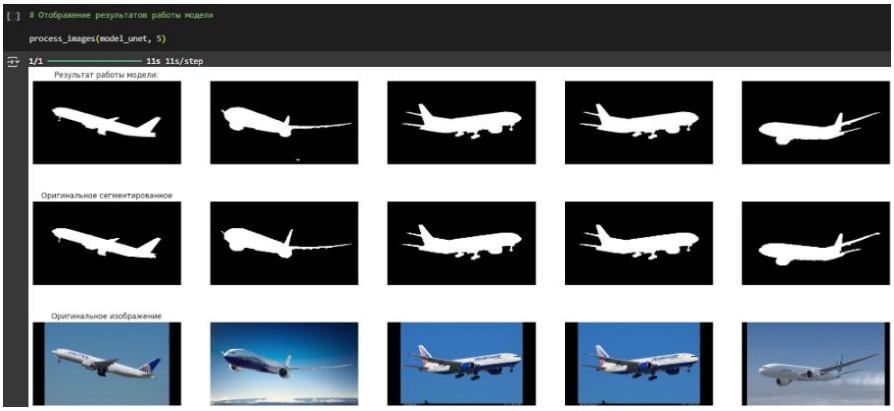


Рисунок 30. Отображение результатов работы модели

Полученный результат выглядит очень хорошо, модель сегментирует даже самые мелкие детали. Теперь выведем 5 самых плохих по точности изображений проверочной выборки (рис. 31).

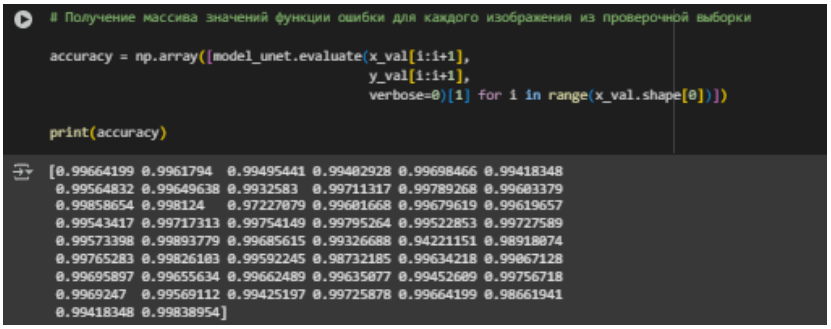


Рисунок 31. Самые плохие по точности изображения

Был получен массив accuracy, в котором сохранены значения val_sparse_categorical_accuracy для каждого изображения проверочной

выборки. Выберем отсюда 5 худших значений. А после выведем на экран результаты работы модели на этих изображениях (рис. 32).

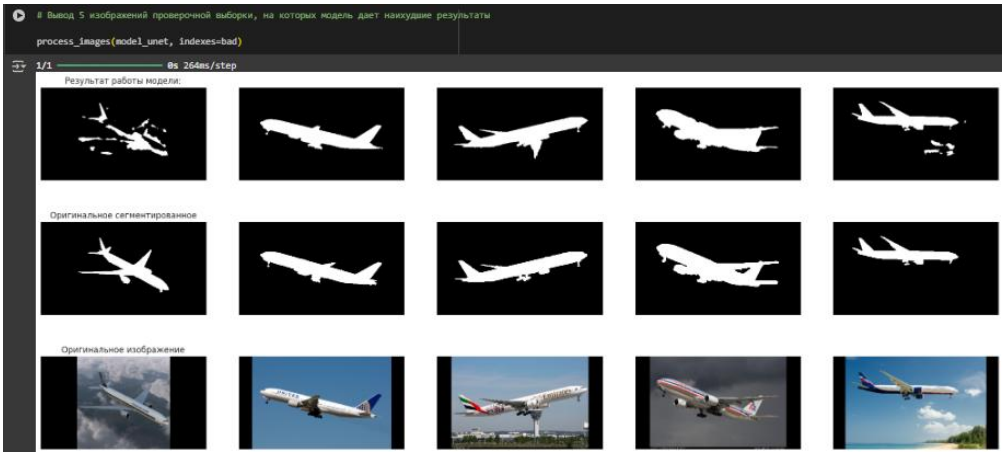


Рисунок 32. Вывод 5 изображений проверочной выборки

Выполнение индивидуальных заданий:

Задание 1.

Условие: необходимо на основе учебного ноутбука провести финальную подготовку данных. Изменить количество сегментирующих классов с 16 на 5.

Провести суммарно не менее 10 экспериментов и визуализировать их результаты (включая точность обучения сетей на одинаковом количестве эпох, например, на 7):

- изменив filters в сверточных слоях
- изменив kernel_size в сверточных слоях
- изменив активационную функцию в скрытых слоях с relu на linear или/и selu, elu.

Для выполнения данного задания вначале запустим раздел “Подготовка”. Для этого выполним импорт необходимых библиотек (рис. 33).

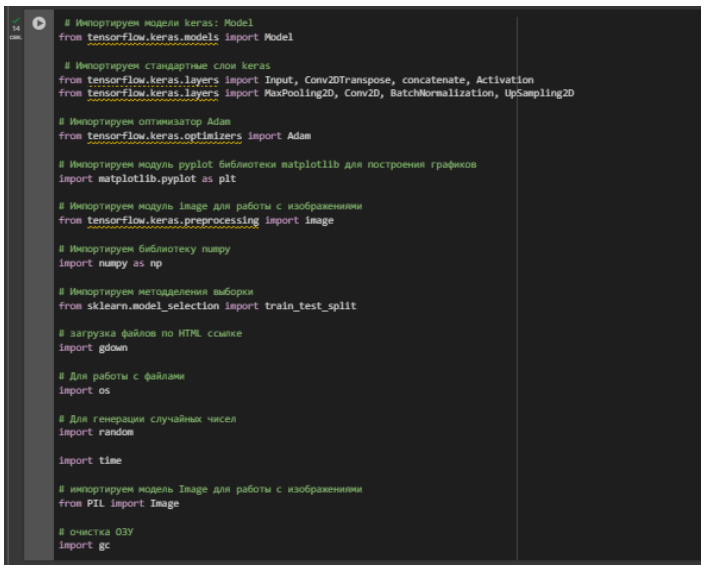


Рисунок 33. Импорт библиотек

Затем загрузим и распакуем архив картинок (рис. 34).

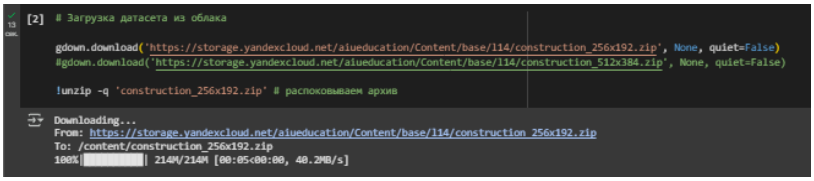


Рисунок 34. Загрузка датасета

Далее зададим глобальные параметры (рис. 35).

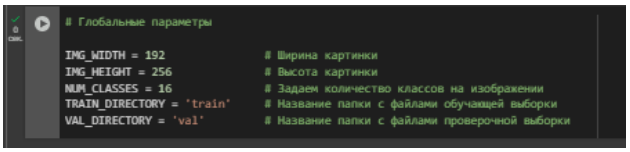


Рисунок 35. Глобальные параметры

Загрузим оригинальные изображения (код из лекции) (рис. 36).

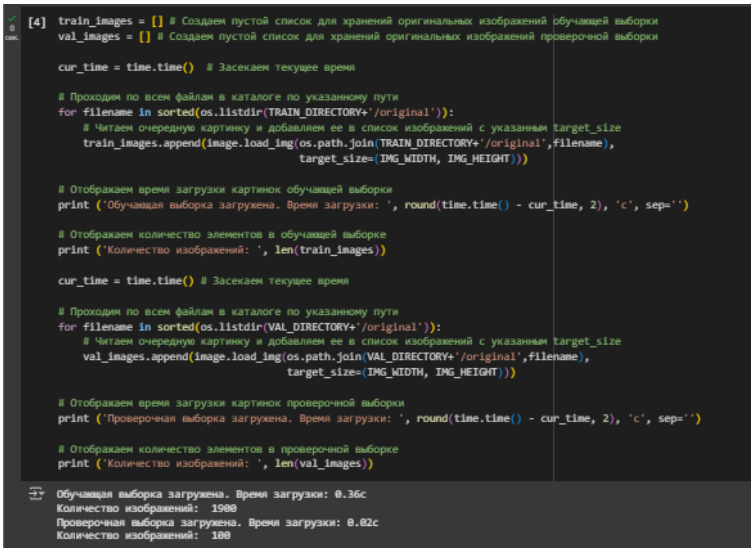


Рисунок 36. Загрузка оригинальных изображений

Загрузим сегментированные изображения (код из лекции) (рис. 37).

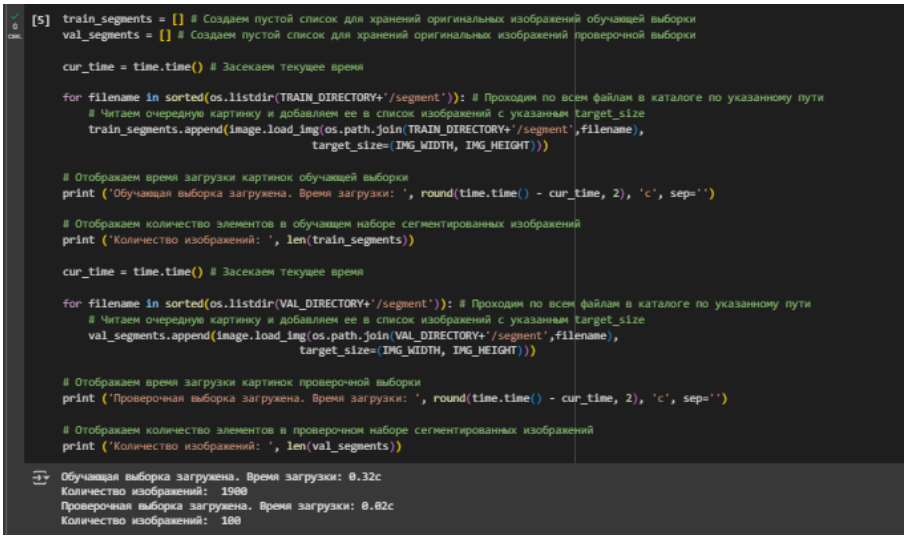


Рисунок 37. Загрузка сегментированных изображений

Далее перейдем к написанию кода для решения задания, для этого напишем функцию для обработки сегментов и выполним ее (рис. 38).

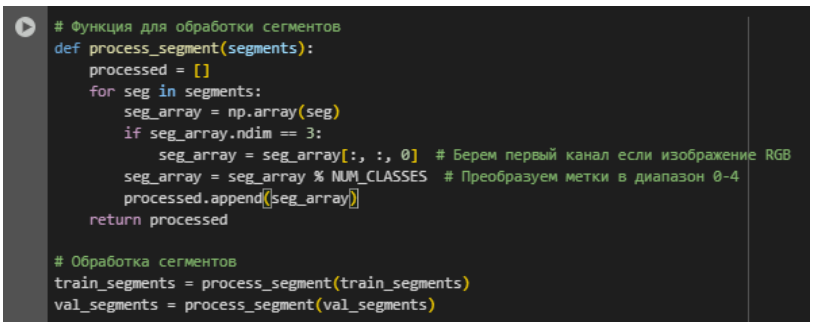


Рисунок 38. Обработка сегментов

Выполним преобразование изображений и меток в numpy массивы (рис. 39).

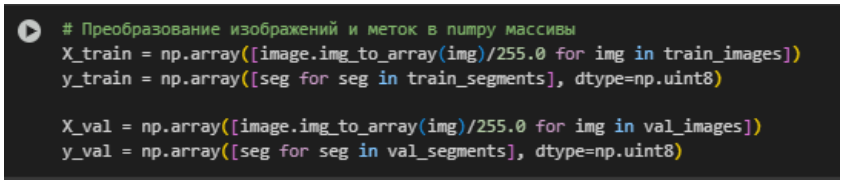
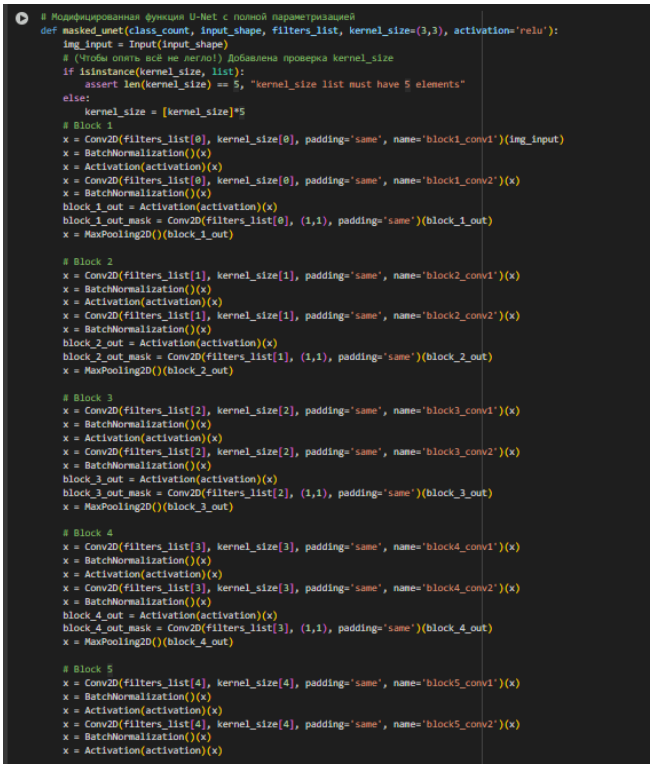


Рисунок 39. Преобразование изображений и меток в numpy массивы

Затем напишем функцию для U-Net с полной параметризацией (рис. 40 - 41).



```python
# Модифицированная функция U-Net с полной параметризацией
def masked_unet(class_count, input_shape, filters_list, kernel_size=(3,3), activation='relu'):
    img_input = Input(input_shape)
    # (Чтобы опять всё не легло!) Добавлена проверка kernel_size
    if isinstance(kernel_size, list):
        assert len(kernel_size) == 5, "kernel_size list must have 5 elements"
    else:
        kernel_size = [kernel_size]*5
    # Block 1
    x = Conv2D(filters_list[0], kernel_size[0], padding='same', name='block1_conv1')(img_input)
    x = BatchNormalization()(x)
    x = Activation(activation)(x)
    x = Conv2D(filters_list[0], kernel_size[0], padding='same', name='block1_conv2')(x)
    x = BatchNormalization()(x)
    block_1_out = Activation(activation)(x)
    block_1_out_mask = Conv2D(filters_list[0], (1,1), padding='same')(block_1_out)
    x = MaxPooling2D()(block_1_out)

    # Block 2
    x = Conv2D(filters_list[1], kernel_size[1], padding='same', name='block2_conv1')(x)
    x = BatchNormalization()(x)
    x = Activation(activation)(x)
    x = Conv2D(filters_list[1], kernel_size[1], padding='same', name='block2_conv2')(x)
    x = BatchNormalization()(x)
    block_2_out = Activation(activation)(x)
    block_2_out_mask = Conv2D(filters_list[1], (1,1), padding='same')(block_2_out)
    x = MaxPooling2D()(block_2_out)

    # Block 3
    x = Conv2D(filters_list[2], kernel_size[2], padding='same', name='block3_conv1')(x)
    x = BatchNormalization()(x)
    x = Activation(activation)(x)
    x = Conv2D(filters_list[2], kernel_size[2], padding='same', name='block3_conv2')(x)
    x = BatchNormalization()(x)
    block_3_out = Activation(activation)(x)
    block_3_out_mask = Conv2D(filters_list[2], (1,1), padding='same')(block_3_out)
    x = MaxPooling2D()(block_3_out)

    # Block 4
    x = Conv2D(filters_list[3], kernel_size[3], padding='same', name='block4_conv1')(x)
    x = BatchNormalization()(x)
    x = Activation(activation)(x)
    x = Conv2D(filters_list[3], kernel_size[3], padding='same', name='block4_conv2')(x)
    x = BatchNormalization()(x)
    block_4_out = Activation(activation)(x)
    block_4_out_mask = Conv2D(filters_list[3], (1,1), padding='same')(block_4_out)
    x = MaxPooling2D()(block_4_out)

    # Block 5
    x = Conv2D(filters_list[4], kernel_size[4], padding='same', name='block5_conv1')(x)
    x = BatchNormalization()(x)
    x = Activation(activation)(x)
    x = Conv2D(filters_list[4], kernel_size[4], padding='same', name='block5_conv2')(x)
    x = BatchNormalization()(x)
    x = Activation(activation)(x)
```

Рисунок 40. U-Net с полной параметризацией (первая часть кода)

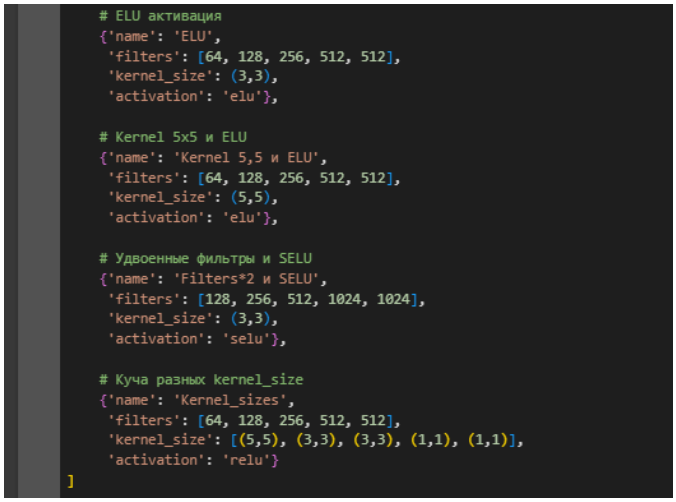
```python
    # Декодинг
    # UP 1
    x = Conv2DTranspose(filters_list[3], (2,2), strides=(2,2), padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation(activation)(x)
    x = concatenate([x, block_4_out, block_4_out_mask])
    x = Conv2D(filters_list[3], kernel_size[3], padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation(activation)(x)

    # UP 2
    x = Conv2DTranspose(filters_list[2], (2,2), strides=(2,2), padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation(activation)(x)
    x = concatenate([x, block_3_out, block_3_out_mask])
    x = Conv2D(filters_list[2], kernel_size[2], padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation(activation)(x)

    # UP 3
    x = Conv2DTranspose(filters_list[1], (2,2), strides=(2,2), padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation(activation)(x)
    x = concatenate([x, block_2_out, block_2_out_mask])
    x = Conv2D(filters_list[1], kernel_size[1], padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation(activation)(x)

    # UP 4
    x = Conv2DTranspose(filters_list[0], (2,2), strides=(2,2), padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation(activation)(x)
    x = concatenate([x, block_1_out, block_1_out_mask])
    x = Conv2D(filters_list[0], kernel_size[0], padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation(activation)(x)

    # Последний слой
    x = Conv2D(class_count, (3,3), activation='softmax', padding='same')(x)

    model = Model(img_input, x)
    model.compile(optimizer=Adam(learning_rate=1e-3),
                  loss='sparse_categorical_crossentropy',
                  metrics=['sparse_categorical_accuracy'])
    return model
```

Рисунок 41. U-Net с полной параметризацией (вторая часть кода)

Зададим список словарей experiments (рис. 42 - 43), каждый из которых — это отдельный эксперимент с разной конфигурацией слоев CNN. Эти эксперименты можно использовать в цикле, чтобы автоматически обучить и сравнить разные архитектуры модели.

```
experiments = [
    # Базовый вариант (оригинальные параметры)
    {'name': 'Basic',
     'filters': [64, 128, 256, 512, 512],
     'kernel_size': (3,3),
     'activation': 'relu'},

    # Фильтры в 2 раза больше
    {'name': 'Filters*2',
     'filters': [128, 256, 512, 1024, 1024],
     'kernel_size': (3,3),
     'activation': 'relu'},

    # Фильтры в 2 раза меньше. При сохранении весов из-за / была ошибка
    {'name': 'Filters_del_2',
     'filters': [32, 64, 128, 256, 256],
     'kernel_size': (3,3),
     'activation': 'relu'},

    # Увеличен kernel
    {'name': 'Kernel 5, 5',
     'filters': [64, 128, 256, 512, 512],
     'kernel_size': (5,5),
     'activation': 'relu'},

    # Уменьшен kernel
    {'name': 'Kernel 1,1',
     'filters': [64, 128, 256, 512, 512],
     'kernel_size': (1,1),
     'activation': 'relu'},

    # Линейная активация
    {'name': 'Linear',
     'filters': [64, 128, 256, 512, 512],
     'kernel_size': (3,3),
     'activation': 'linear'},

    # SELU активация
    {'name': 'SELU',
     'filters': [64, 128, 256, 512, 512],
     'kernel_size': (3,3),
     'activation': 'selu'},
```

Рисунок 42. Список словарей (первая часть кода)



```
    # ELU активация
    {'name': 'ELU',
     'filters': [64, 128, 256, 512, 512],
     'kernel_size': (3,3),
     'activation': 'elu'},

    # Kernel 5x5 и ELU
    {'name': 'Kernel 5,5 и ELU',
     'filters': [64, 128, 256, 512, 512],
     'kernel_size': (5,5),
     'activation': 'elu'},

    # Удвоенные фильтры и SELU
    {'name': 'Filters*2 и SELU',
     'filters': [128, 256, 512, 1024, 1024],
     'kernel_size': (3,3),
     'activation': 'selu'},

    # Куча разных kernel_size
    {'name': 'Kernel_sizes',
     'filters': [64, 128, 256, 512, 512],
     'kernel_size': [(5,5), (3,3), (3,3), (1,1), (1,1)],
     'activation': 'relu'}
]
```

Рисунок 43. Список словарей (вторая часть кода)

Далее напишем цикл для обучения модели, которая будет создавать модель с нужной для эксперимента архитектурой, затем обучать ее, сохранять веса и выводить графики точности и потерь (рис. 44).

```python
# Цикл обучения
histories = []
for exp in experiments:
    print(f"\nStarting experiment: {exp['name']}")
    print(f"Parameters: filters={exp['filters']}, kernel={exp['kernel_size']}, activation={exp['activation']}")

    # Создание модели
    model = masked_unet(
        class_count=NUM_CLASSES,
        input_shape=(IMG_WIDTH, IMG_HEIGHT, 3),
        filters_list=exp['filters'],
        kernel_size=exp['kernel_size'],
        activation=exp['activation']
    )

    # Обучение
    history = model.fit(
        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=7,
        batch_size=8,
        verbose=1
    )

    # Сохранение истории обучения
    histories.append({
        'name': exp['name'],
        'history': history.history,
        'params': exp
    })
    # Сохранение весов модели
    safe_name = exp['name'].replace(' ', '_').lower()
    weights_filename = f"{safe_name}.weights.h5"
    model.save_weights(weights_filename)
    print(f"Веса в {weights_filename}")
    # Визуализация для текущей модели
    plt.figure(figsize=(20, 10))

    # График точности
    plt.subplot(1, 2, 1)
    plt.plot(history.history['sparse_categorical_accuracy'], '--', label='Training Accuracy')
    plt.plot(history.history['val_sparse_categorical_accuracy'], '-', label='Validation Accuracy')
    plt.title(f'Accuracy: {exp["name"]}')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend()

    # График потерь
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], '--', label='Training Loss')
    plt.plot(history.history['val_loss'], '-', label='Validation Loss')
    plt.title(f'Loss: {exp["name"]}')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend()
    plt.tight_layout()
    plt.show()

    # Очистка памяти
    del model
    gc.collect()
    print(f"Experiment {exp['name']} completed.\n")
```

Рисунок 44. Цикл обучения моделей

Посмотрим на пример одного из обучения данной (рис. 45) (полное выполнение данного обучения можно посмотреть в репозитории по ссылке в конце работы).

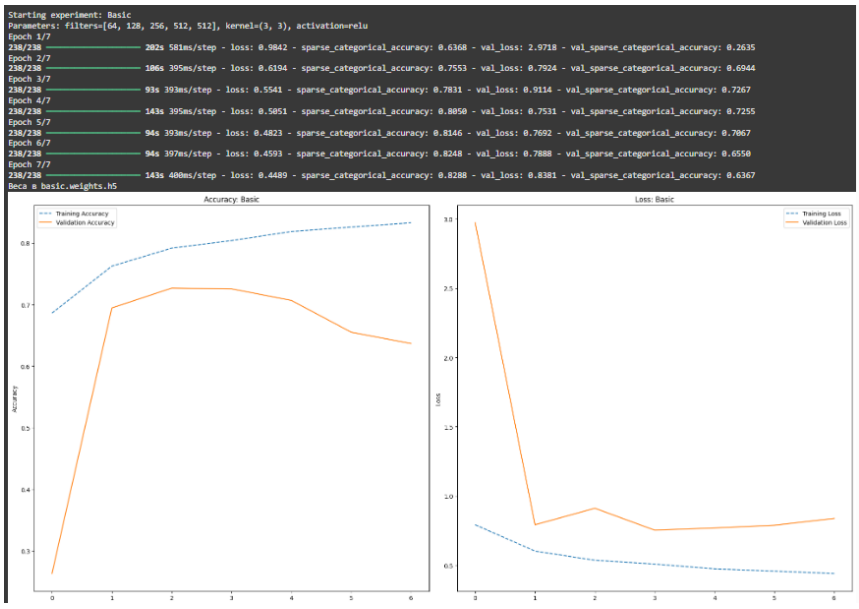


Рисунок 45. Обучение базовой модели

Далее выполним команду для пропуска обучения моделей, которые уже обучены (рис. 46).

```
skip_list = ["Basic", "Filters*2", "Filters_del_2", "Kernel 5, 5"]
```

Рисунок 46. Пропуск обученных моделей

Выполним тот же цикл, что и на рисунке 44, но для оставшихся моделей, это выполняется из-за того, что после первой части обучения заканчивается ОЗУ и мы не можем продолжать обучение, поэтому поменяли среду и продолжили обучение оставшихся моделей. Посмотрим на процесс обучения одной из них (рис. 47) (полное выполнение данного обучения можно посмотреть в репозитории по ссылке в конце работы).

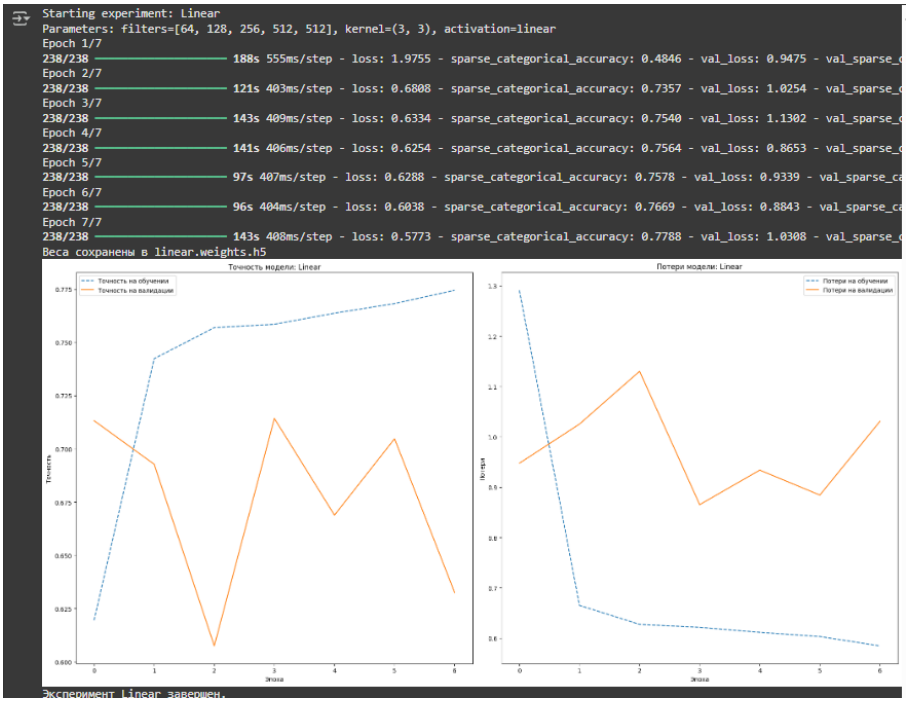


Рисунок 47. Добавление линейной активации

Затем выполним визуализацию всех моделей на графиках точности и потери (рис. 48).

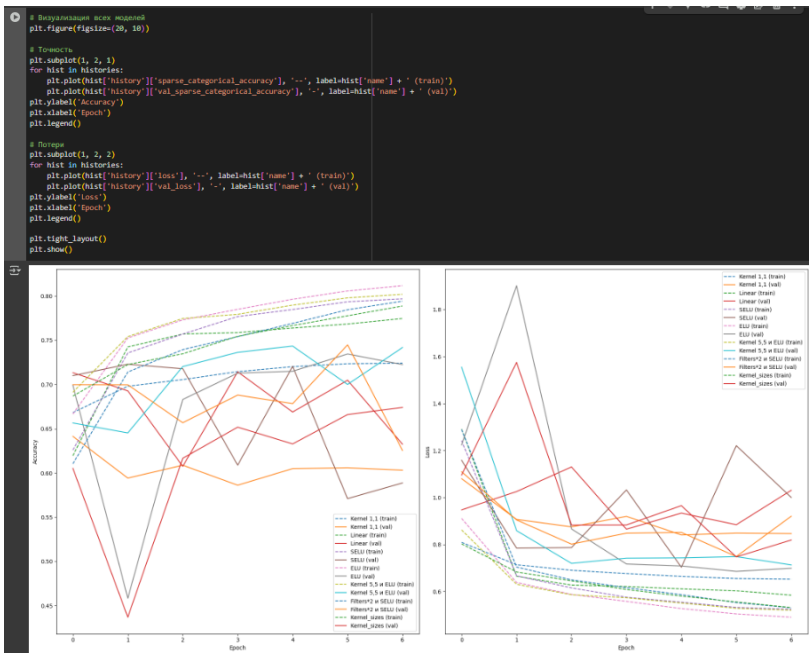


Рисунок 48. Визуализация всех моделей на графиках точности и потери

Отсюда, сделаем некоторые выводы по заданию: количество сегментирующих классов было изменено с 16 на 5, что упростило задачу и уменьшило вычислительную сложность.

Было проведено 11 экспериментов с различными параметрами модели, включая изменение количества фильтров, размера ядра свертки и активационных функций. Эксперименты показали, что увеличение количества фильтров (например, до 128, 256, 512, 1024) может улучшить точность модели, но требует больше вычислительных ресурсов.

Изменение размера ядра свертки влияет на способность модели выделять признаки: большие ядра лучше улавливают глобальные особенности, а маленькие — локальные.

Замена активационной функции с ReLU на линейную, SELU или ELU показала, что ReLU остается эффективным выбором, но SELU и ELU могут быть полезны в некоторых сценариях.

Наилучшие результаты были достигнуты с увеличением количества фильтров и использованием активационной функции ReLU. Комбинация увеличенных фильтров и функции активации SELU также показала хорошие результаты, что может быть связано с её свойством самомасштабирования.

Эксперименты с линейной активацией оказались менее эффективными, что подтверждает важность нелинейностей в глубоком обучении.

Задание 2.

Условие: необходимо на основе учебного ноутбука провести финальную подготовку данных. Изменить количество сегментирующих классов с 16 на 7:

- 0_класс - FLOOR
- 1_класс - CEILING
- 2_класс - WALL
- 3_класс - APERTURE, DOOR, WINDOW
- 4_класс - COLUMN, RAILINGS, LADDER
- 5_класс - INVENTORY
- 6_класс - LAMP, WIRE, BEAM, EXTERNAL, BATTERY, PEOPLE

Изучить внимательно особенности U-net, определить в чем принципиальное отличие U-net и simpleUnet из учебного ноутбука.

Доработать simpleUnet с учетом особенностей U-net. Обучить модель на 100 эпохах и визуализировать результат.

В начале выполнения данного задания запустим программу “Подготовка”. Для этого выполним импорт нужных библиотек (рис. 49).

```
[2]  # Импортируем модели keras: Model
     from tensorflow.keras.models import Model

     # Импортируем стандартные слои keras
     from tensorflow.keras.layers import Input, Conv2DTranspose, concatenate, Activation
     from tensorflow.keras.layers import MaxPooling2D, Conv2D, BatchNormalization, UpSampling2D

     # Импортируем оптимизатор Adam
     from tensorflow.keras.optimizers import Adam

     # Импортируем модуль pyplot библиотеки matplotlib для построения графиков
     import matplotlib.pyplot as plt

     # Импортируем модуль image для работы с изображениями
     from tensorflow.keras.preprocessing import image

     # Импортируем библиотеку numpy
     import numpy as np

     # Импортируем методделения выборки
     from sklearn.model_selection import train_test_split

     # загрузка файлов по HTML ссылке
     import gdown

     # Для работы с файлами
     import os

     # Для генерации случайных чисел
     import random

     import time

     # импортируем модель Image для работы с изображениями
     from PIL import Image

     # очистка ОЗУ
     import gc
```

Рисунок 49. Импорт библиотек

Далее выполним загрузку датасета из облака и распакуем архив (рис. 50).

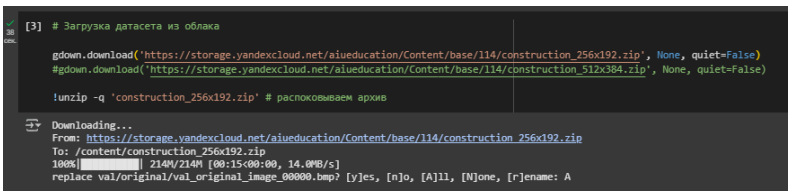


Рисунок 50. Загрузка датасета

Затем зададим глобальные параметры (рис. 51).

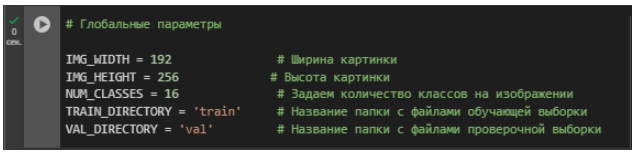


Рисунок 51. Глобальные параметры

Потом загрузим оригинальные изображения (код из лекции) (рис. 52).

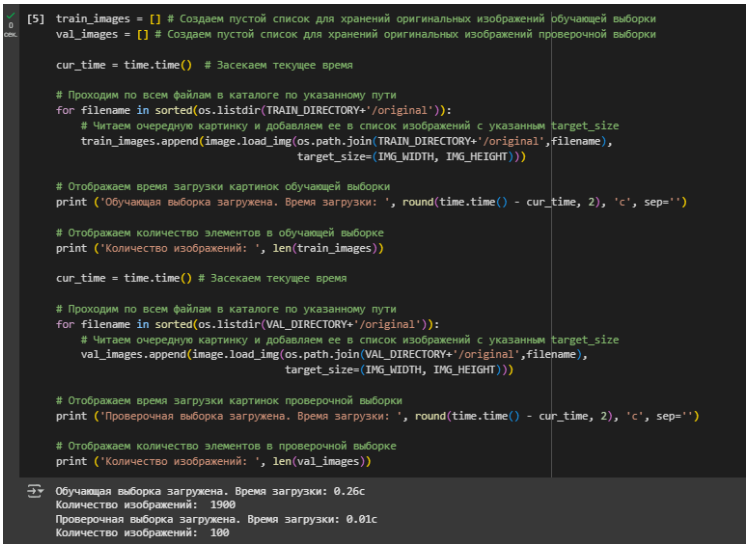


Рисунок 52. Загрузка оригинальных изображений

Далее загрузим сегментированные изображения (код из лекции) (рис. 53).

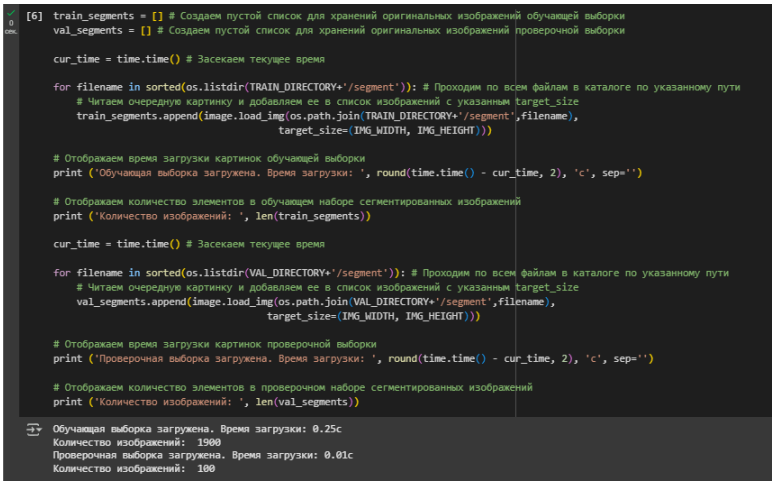


Рисунок 53. Загрузка сегментированные изображений

Затем перейдем к выполнению заданию, для этого сначала зададим все новые классы (рис. 54).

```
[8]  # Новые классы
     FLOOR = (100, 100, 100)          # Пол (Серый)
     CEILING = (0, 0, 100)            # Потолок (Синий)
     WALL = (0, 100, 0)               # Стена (Зелёный)
     APERTURE = (0, 100, 100)         # Проем (Темно-бирюзовый)
     DOOR = (100, 0, 100)             # Дверь (Бордовый)
     WINDOW = (100, 100, 0)           # Окно (Золотой)
     COLUMN = (100, 0, 0)             # Колонна (Красный)
     RAILINGS = (0, 200, 0)           # Перила (Светло-зелёный)
     LADDER = (0, 0, 200)             # Лестница (Светло-синий)
     INVENTORY = (200, 0, 200)        # Инвентарь (Розовый)
     LAMP = (200, 200, 0)             # Лампа (Жёлтый)
     WIRE = (0, 100, 200)             # Провод (Голубой)
     BEAM = (100, 0, 200)             # Балка (Фиолетовый)
     EXTERNAL = (200, 200, 200)       # Внешний мир (Светло-серый)
     BATTERY = (200, 0, 0)            # Батареи (Светло-красный)
     PEOPLE = (0, 200, 200)           # Люди (Бирюзовый)
```

Рисунок 54. Новые классы

Изменим количество сегментирующих классов с 16 на 7 (рис. 55):

- 0_класс - FLOOR
- 1_класс - CEILING
- 2_класс - WALL
- 3_класс - APERTURE, DOOR, WINDOW
- 4_класс - COLUMN, RAILINGS, LADDER
- 5_класс - INVENTORY
- 6_класс - LAMP, WIRE, BEAM, EXTERNAL, BATTERY, PEOPLE

```
[9]  # Список меток семи классов
     NEW_CLASS_LABELS = (FLOOR, CEILING, WALL, APERTURE, COLUMN, INVENTORY, LAMP)
     NUM_CLASSES = 7  # Теперь классов 7

     # Функция преобразования цветного изображения в метки классов (7 классов)
     def rgb_to_labels_7class(image_list):
         result = []
         for d in image_list:
             sample = np.array(d)
             y = np.zeros((IMG_WIDTH, IMG_HEIGHT, 1), dtype='uint8')
             # FLOOR
             y[np.where(np.all(sample == FLOOR, axis=-1))] = 0
             # CEILING
             y[np.where(np.all(sample == CEILING, axis=-1))] = 1
             # WALL
             y[np.where(np.all(sample == WALL, axis=-1))] = 2
             # APERTURE, DOOR, WINDOW
             y[np.where(np.all(sample == APERTURE, axis=-1))] = 3
             y[np.where(np.all(sample == DOOR, axis=-1))] = 3
             y[np.where(np.all(sample == WINDOW, axis=-1))] = 3
             # COLUMN, RAILINGS, LADDER
             y[np.where(np.all(sample == COLUMN, axis=-1))] = 4
             y[np.where(np.all(sample == RAILINGS, axis=-1))] = 4
             y[np.where(np.all(sample == LADDER, axis=-1))] = 4
             # INVENTORY
             y[np.where(np.all(sample == INVENTORY, axis=-1))] = 5
             # LAMP, WIRE, BEAM, EXTERNAL, BATTERY, PEOPLE
             y[np.where(np.all(sample == LAMP, axis=-1))] = 6
             y[np.where(np.all(sample == WIRE, axis=-1))] = 6
             y[np.where(np.all(sample == BEAM, axis=-1))] = 6
             y[np.where(np.all(sample == EXTERNAL, axis=-1))] = 6
             y[np.where(np.all(sample == BATTERY, axis=-1))] = 6
             y[np.where(np.all(sample == PEOPLE, axis=-1))] = 6
             result.append(y)
         return np.array(result)
```

Рисунок 55. Изменение количества сегментирующих классов

Напишем функцию преобразования семантических масок (7 классов) в RGB-изображение с цветовой кодировкой (рис. 56).

```
[10] # Функция преобразования семантических масок (7 классов) в RGB-изображение с цветовой кодировкой
     def labels_to_rgb_7class(image_list):
         result = []
         for y in image_list:
             temp = np.zeros((IMG_WIDTH, IMG_HEIGHT, 3), dtype='uint8')
             # FLOOR (Серый)
             temp[np.where(np.all(y==0, axis=-1))] = FLOOR
             # CEILING (Синий)
             temp[np.where(np.all(y==1, axis=-1))] = CEILING
             # WALL (Зеленый)
             temp[np.where(np.all(y==2, axis=-1))] = WALL
             # APERTURE (Тёмно-бирюзовый)
             temp[np.where(np.all(y==3, axis=-1))] = APERTURE
             # COLUMN (Красный)
             temp[np.where(np.all(y==4, axis=-1))] = COLUMN
             # INVENTORY (Розовый)
             temp[np.where(np.all(y==5, axis=-1))] = INVENTORY
             # LAMP (Жёлтый)
             temp[np.where(np.all(y==6, axis=-1))] = LAMP
             result.append(temp)
         return np.array(result)
```

Рисунок 56. Функция преобразования семантических масок

Далее выполним создание улучшенной U-Net архитектуры для семантической сегментации (рис. 57 - 58).

```
[12] # Улучшенная реализация U-Net архитектуры для семантической сегментации
     def improved_simple_unet(class_count, input_shape):
         img_input = Input(input_shape)
         # Block 1
         x = Conv2D(32, (3, 3), padding='same', name='block1_conv1')(img_input)
         x = BatchNormalization()(x)
         x = Activation('relu')(x)
         x = Conv2D(32, (3, 3), padding='same', name='block1_conv2')(x)
         x = BatchNormalization()(x)
         block_1_out = Activation('relu')(x)
         x = MaxPooling2D((2, 2))(block_1_out)

         # Block 2
         x = Conv2D(64, (3, 3), padding='same', name='block2_conv1')(x)
         x = BatchNormalization()(x)
         x = Activation('relu')(x)
         x = Conv2D(64, (3, 3), padding='same', name='block2_conv2')(x)
         x = BatchNormalization()(x)
         block_2_out = Activation('relu')(x)
         x = MaxPooling2D((2, 2))(block_2_out)

         # Block 3
         x = Conv2D(128, (3, 3), padding='same', name='block3_conv1')(x)
         x = BatchNormalization()(x)
         x = Activation('relu')(x)
         x = Conv2D(128, (3, 3), padding='same', name='block3_conv2')(x)
         x = BatchNormalization()(x)
         block_3_out = Activation('relu')(x)
         x = MaxPooling2D((2, 2))(block_3_out)

         # Block 4
         x = Conv2D(256, (3, 3), padding='same', name='block4_conv1')(x)
         x = BatchNormalization()(x)
         x = Activation('relu')(x)
         x = Conv2D(256, (3, 3), padding='same', name='block4_conv2')(x)
         x = BatchNormalization()(x)
         block_4_out = Activation('relu')(x)
         x = MaxPooling2D((2, 2))(block_4_out)
```

Рисунок 57. Создание U-Net архитектуры первая часть кода

```
    # UP 1
    x = Conv2DTranspose(256, (2, 2), strides=(2, 2), padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = concatenate([x, block_4_out])
    x = Conv2D(256, (3, 3), padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    # UP 2
    x = Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = concatenate([x, block_3_out])
    x = Conv2D(128, (3, 3), padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    # UP 3
    x = Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = concatenate([x, block_2_out])
    x = Conv2D(64, (3, 3), padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    # UP 4
    x = Conv2DTranspose(32, (2, 2), strides=(2, 2), padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = concatenate([x, block_1_out])
    x = Conv2D(32, (3, 3), padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = Conv2D(class_count, (3, 3), activation='softmax', padding='same')(x)
    model = Model(img_input, x)
    model.compile(optimizer=Adam(),
                  loss='sparse_categorical_crossentropy',
                  metrics=['sparse_categorical_accuracy'])
    return model
```

Рисунок 58. Создание U-Net архитектуры первая вторая кода

Преобразуем входные изображения в numpy массивы и сегментированные изображения в метки семи классов (рис. 59).

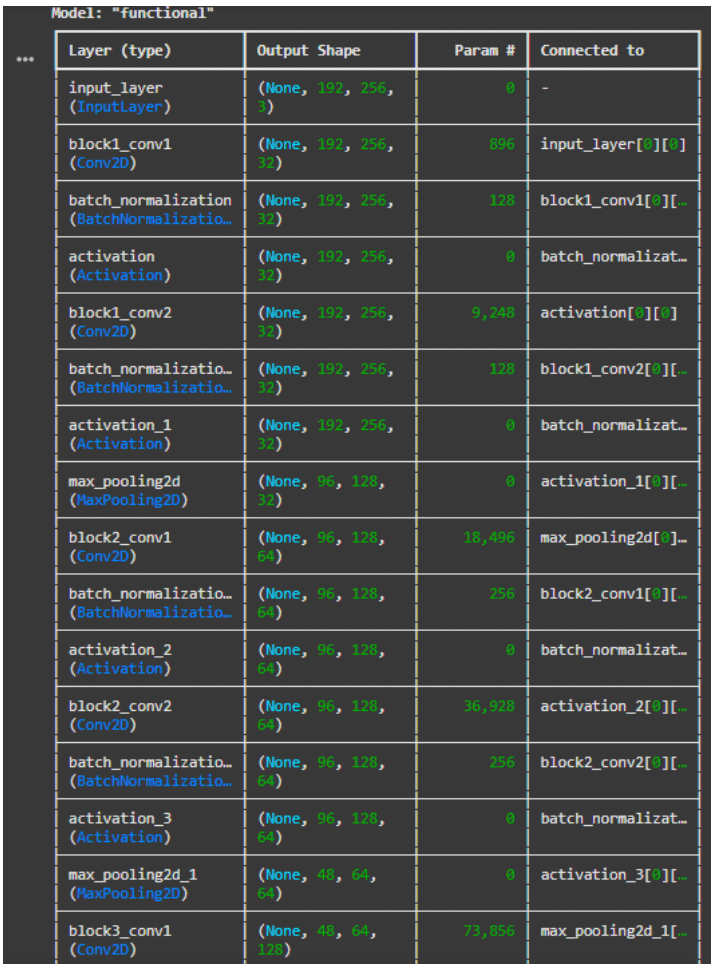
```
[13] # Входные изображения в numpy массивы
     x_train = np.array([image.img_to_array(img) for img in train_images])
     x_val = np.array([image.img_to_array(img) for img in val_images])

     # Сегментированные изображения в метки семи классов
     y_train = rgb_to_labels_7class(train_segments)
     y_val = rgb_to_labels_7class(val_segments)
```

Рисунок 59. Преобразование изображений

Далее рассмотрим модель simpleUnet и выполним вывод архитектуры модели (рис. 61 - 64). А также выполним обучение модели (рис. 60).

```
import tensorflow as tf
with tf.device('/GPU:0'):
    # Улучшенная модель
    model_improved = improved_simple_unet(NUM_CLASSES, (IMG_WIDTH, IMG_HEIGHT, 3))
    model_improved.summary()
    # Обучение
    history = model_improved.fit(x_train, y_train,
                                 epochs=100, batch_size=32,
                                 validation_data=(x_val, y_val))
```

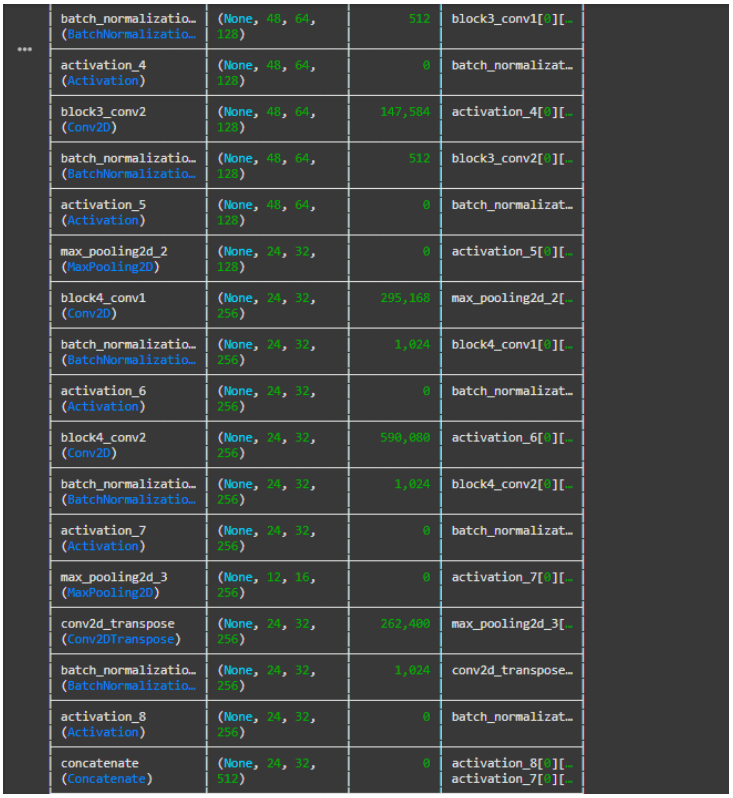
Рисунок 60. Обучение модели



Model: "functional"

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 192, 256, 3)	0	-
block1_conv1 (Conv2D)	(None, 192, 256, 32)	896	input_layer[0][0]
batch_normalization (BatchNormalizatio…	(None, 192, 256, 32)	128	block1_conv1[0][…
activation (Activation)	(None, 192, 256, 32)	0	batch_normalizat…
block1_conv2 (Conv2D)	(None, 192, 256, 32)	9,248	activation[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 192, 256, 32)	128	block1_conv2[0][…
activation_1 (Activation)	(None, 192, 256, 32)	0	batch_normalizat…
max_pooling2d (MaxPooling2D)	(None, 96, 128, 32)	0	activation_1[0][…
block2_conv1 (Conv2D)	(None, 96, 128, 64)	18,496	max_pooling2d[0]…
batch_normalizatio… (BatchNormalizatio…	(None, 96, 128, 64)	256	block2_conv1[0][…
activation_2 (Activation)	(None, 96, 128, 64)	0	batch_normalizat…
block2_conv2 (Conv2D)	(None, 96, 128, 64)	36,928	activation_2[0][…
batch_normalizatio… (BatchNormalizatio…	(None, 96, 128, 64)	256	block2_conv2[0][…
activation_3 (Activation)	(None, 96, 128, 64)	0	batch_normalizat…
max_pooling2d_1 (MaxPooling2D)	(None, 48, 64, 64)	0	activation_3[0][…
block3_conv1 (Conv2D)	(None, 48, 64, 128)	73,856	max_pooling2d_1[…

Рисунок 61. Архитектура модели (часть 1)



batch_normalizatio… (BatchNormalizatio…	(None, 48, 64, 128)	512	block3_conv1[0][…
activation_4 (Activation)	(None, 48, 64, 128)	0	batch_normalizat…
block3_conv2 (Conv2D)	(None, 48, 64, 128)	147,584	activation_4[0][…
batch_normalizatio… (BatchNormalizatio…	(None, 48, 64, 128)	512	block3_conv2[0][…
activation_5 (Activation)	(None, 48, 64, 128)	0	batch_normalizat…
max_pooling2d_2 (MaxPooling2D)	(None, 24, 32, 128)	0	activation_5[0][…
block4_conv1 (Conv2D)	(None, 24, 32, 256)	295,168	max_pooling2d_2[…
batch_normalizatio… (BatchNormalizatio…	(None, 24, 32, 256)	1,024	block4_conv1[0][…
activation_6 (Activation)	(None, 24, 32, 256)	0	batch_normalizat…
block4_conv2 (Conv2D)	(None, 24, 32, 256)	590,080	activation_6[0][…
batch_normalizatio… (BatchNormalizatio…	(None, 24, 32, 256)	1,024	block4_conv2[0][…
activation_7 (Activation)	(None, 24, 32, 256)	0	batch_normalizat…
max_pooling2d_3 (MaxPooling2D)	(None, 12, 16, 256)	0	activation_7[0][…
conv2d_transpose (Conv2DTranspose)	(None, 24, 32, 256)	262,400	max_pooling2d_3[…
batch_normalizatio… (BatchNormalizatio…	(None, 24, 32, 256)	1,024	conv2d_transpose…
activation_8 (Activation)	(None, 24, 32, 256)	0	batch_normalizat…
concatenate (Concatenate)	(None, 24, 32, 512)	0	activation_8[0][… activation_7[0][…

Рисунок 62. Архитектура модели (часть 2)

conv2d (Conv2D)	(None, 24, 32, 256)	1,179,904	concatenate[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 24, 32, 256)	1,024	conv2d[0][0]
activation_9 (Activation)	(None, 24, 32, 256)	0	batch_normalizat…
conv2d_transpose_1 (Conv2DTranspose)	(None, 48, 64, 128)	131,200	activation_9[0][…
batch_normalizatio… (BatchNormalizatio…	(None, 48, 64, 128)	512	conv2d_transpose…
activation_10 (Activation)	(None, 48, 64, 128)	0	batch_normalizat…
concatenate_1 (Concatenate)	(None, 48, 64, 256)	0	activation_10[0]… activation_5[0][…
conv2d_1 (Conv2D)	(None, 48, 64, 128)	295,040	concatenate_1[0]…
batch_normalizatio… (BatchNormalizatio…	(None, 48, 64, 128)	512	conv2d_1[0][0]
activation_11 (Activation)	(None, 48, 64, 128)	0	batch_normalizat…
conv2d_transpose_2 (Conv2DTranspose)	(None, 96, 128, 64)	32,832	activation_11[0]…
batch_normalizatio… (BatchNormalizatio…	(None, 96, 128, 64)	256	conv2d_transpose…
activation_12 (Activation)	(None, 96, 128, 64)	0	batch_normalizat…
concatenate_2 (Concatenate)	(None, 96, 128, 128)	0	activation_12[0]… activation_3[0][…
conv2d_2 (Conv2D)	(None, 96, 128, 64)	73,792	concatenate_2[0]…
batch_normalizatio… (BatchNormalizatio…	(None, 96, 128, 64)	256	conv2d_2[0][0]
activation_13 (Activation)	(None, 96, 128, 64)	0	batch_normalizat…

Рисунок 63. Архитектура модели (часть 3)

conv2d_transpose_3 (Conv2DTranspose)	(None, 192, 256, 32)	8,224	activation_13[0]…
batch_normalizatio… (BatchNormalizatio…	(None, 192, 256, 32)	128	conv2d_transpose…
activation_14 (Activation)	(None, 192, 256, 32)	0	batch_normalizat…
concatenate_3 (Concatenate)	(None, 192, 256, 64)	0	activation_14[0]… activation_1[0][…
conv2d_3 (Conv2D)	(None, 192, 256, 32)	18,464	concatenate_3[0]…
batch_normalizatio… (BatchNormalizatio…	(None, 192, 256, 32)	128	conv2d_3[0][0]
activation_15 (Activation)	(None, 192, 256, 32)	0	batch_normalizat…
conv2d_4 (Conv2D)	(None, 192, 256, 7)	2,023	activation_15[0]…

Total params: 3,183,815 (12.15 MB)
Trainable params: 3,179,975 (12.13 MB)
Non-trainable params: 3,840 (15.00 KB)

Рисунок 64. Архитектура модели (часть 4)

Далее посмотрим на график процесса обучения модели (рис. 65).

```
# Графики обучения
plt.figure(figsize=(14, 7))
plt.plot(history.history['sparse_categorical_accuracy'])
plt.plot(history.history['val_sparse_categorical_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

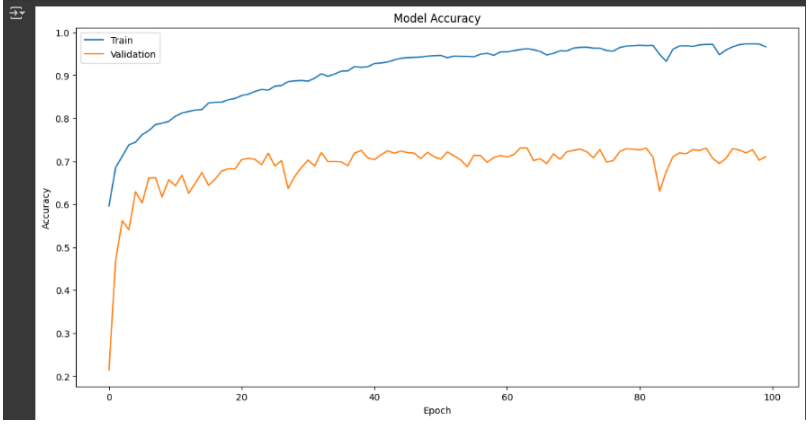


Рисунок 65. График процесса обучения модели

Напишем функцию для визуализации сегментации (рис. 66).

```
# Визуализация
def process_images_7class(model, count=1):
    indexes = np.random.randint(0, len(x_val), count)
    predict = np.argmax(model.predict(x_val[indexes]), axis=-1)
    orig = labels_to_rgb_7class(predict[..., None])
    fig, axs = plt.subplots(3, count, figsize=(25, 15))
    for i in range(count):
        axs[0, 0].set_title('Результат работы модели:')
        axs[0, i].imshow(orig[i])
        axs[0, i].axis('off')
        axs[1, 0].set_title('Оригинальное сегментированное')
        axs[1, i].imshow(val_segments[indexes[i]])
        axs[1, i].axis('off')
        axs[2, 0].set_title('Оригинальное изображение')
        axs[2, i].imshow(val_images[indexes[i]])
        axs[2, i].axis('off')
    plt.show()
```

Рисунок 66. Функция для визуализации сегментации

Затем посмотрим на результаты сегментации (рис. 67).

```
# Результаты сегментации
process_images_7class(model_improved, 5)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 5s 5s/step
```

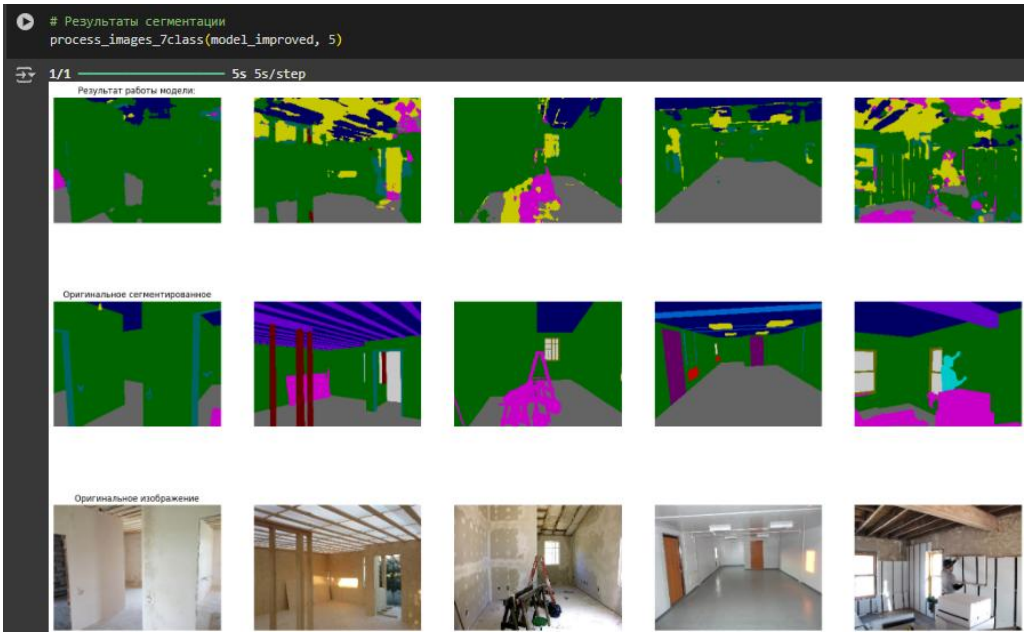


Рисунок 67. Результаты сегментации

Отсюда по заданию сделаем выводы: была проведена подготовка данных. Основной задачей на этом этапе стало объединение исходных 16 классов сегментации в 7 обобщённых категорий. Такое объединение позволило упростить задачу и повысить обобщающую способность модели при сохранении ключевых объектов в архитектурной среде.

Далее был проведён анализ архитектуры U-net и её упрощённой версии — simpleUnet. Основное отличие классической U-net заключается в симметричной структуре и наличии пропусков между слоями энкодера и декодера, что позволяет эффективно сохранять пространственные признаки и повышает точность сегментации.

С учётом этих особенностей базовая модель simpleUnet была доработана: в архитектуру были добавлены skip connections, увеличено число фильтров и глубина сети. После модификации модель была обучена на протяжении 100 эпох. Визуализация результатов показала стабильное улучшение качества сегментации, а также более точное выделение границ между объектами по сравнению с исходным вариантом сети.

В процессе обучения возникали сложности, связанные с ограничениями вычислительных ресурсов, в частности с риском переполнения оперативной памяти. Для их устранения были предприняты меры по оптимизации: уменьшен размер батча, снижено разрешение входных изображений и реализована очистка памяти между эпохами.

Задание 3.

Условие: необходимо на основе учебного ноутбука, провести финальную подготовку данных. Изменить количество сегментирующих классов с 16 на 7:

- 0_класс - FLOOR
- 1_класс - CEILING
- 2_класс - WALL
- 3_класс - APERTURE, DOOR, WINDOW
- 4_класс - COLUMN, RAILINGS, LADDER

– 5_класс - INVENTORY

– 6_класс - LAMP, WIRE, BEAM, EXTERNAL, BATTERY, PEOPLE

Реализовать сегментацию базы Стройка на основе модели PSPnet.

Для начала выполним реализацию раздела подготовка, для этого выполним импорт библиотек (рис. 68).

```
# Импортируем модели keras: Model
from tensorflow.keras.models import Model
import tensorflow as tf
# Импортируем стандартные слои keras
from tensorflow.keras.layers import Input, Conv2DTranspose, concatenate, Activation
from tensorflow.keras.layers import MaxPooling2D, Conv2D, BatchNormalization, UpSampling2D

# Импортируем оптимизатор Adam
from tensorflow.keras.optimizers import Adam

# Импортируем модуль pyplot библиотеки matplotlib для построения графиков
import matplotlib.pyplot as plt

# Импортируем модуль image для работы с изображениями
from tensorflow.keras.preprocessing import image

# Импортируем библиотеку numpy
import numpy as np

# Импортируем методделения выборки
from sklearn.model_selection import train_test_split

# загрузка файлов по HTML ссылке
import gdown

# Для работы с файлами
import os

# Для генерации случайных чисел
import random

import time

# импортируем модель Image для работы с изображениями
from PIL import Image

# очистка ОЗУ
import gc

from tensorflow.keras import backend as K

from tensorflow.keras.layers import Input, Conv2D, BatchNormalization, Activation, MaxPooling2D

from tensorflow.keras.layers import AveragePooling2D, UpSampling2D, concatenate

from tensorflow.keras.layers import Cropping2D, ZeroPadding2D

from tensorflow.keras.models import Model

from tensorflow.keras.optimizers import Adam

import os
```

Рисунок 68. Импорт библиотек

Далее выполним загрузку датасета и распаковку архива (рис. 69).

```
# Загрузка датасета из облака

gdown.download('https://storage.yandexcloud.net/aiueducation/Content/base/114/construction_256x192.zip', None, quiet=False)
#gdown.download('https://storage.yandexcloud.net/aiueducation/Content/base/114/construction_512x384.zip', None, quiet=False)

!unzip -q 'construction_256x192.zip' # распоковываем архив
```

```
Downloading...
From: https://storage.yandexcloud.net/aiueducation/Content/base/114/construction_256x192.zip
To: /content/construction_256x192.zip
100%|██████████| 214M/214M [00:18<00:00, 11.7MB/s]
```

Рисунок 69. Загрузка датасета

Перейдем к выполнению задания, для этого для начала напишем функцию преобразования RGB в метки (написана для 7 классов) (рис. 70).

```python
# Функция преобразования RGB в метки (адаптирована под 7 классов)
def rgb_to_labels(image_list):
    result = []
    for img in image_list:
        sample = np.array(img)
        y = np.zeros((IMG_HEIGHT, IMG_WIDTH, 1), dtype='uint8')

        # FLOOR
        y[np.where(np.all(sample == (100, 100, 100), axis=-1))] = 0
        # CEILING
        y[np.where(np.all(sample == (0, 0, 100), axis=-1))] = 1
        # WALL
        y[np.where(np.all(sample == (0, 100, 0), axis=-1))] = 2
        # APERTURE/DOOR/WINDOW
        y[np.where(np.all(sample == (0, 100, 100), axis=-1))] = 3
        y[np.where(np.all(sample == (100, 0, 100), axis=-1))] = 3
        y[np.where(np.all(sample == (100, 100, 0), axis=-1))] = 3
        # COLUMN/RAILINGS/LADDER
        y[np.where(np.all(sample == (100, 0, 0), axis=-1))] = 4
        y[np.where(np.all(sample == (0, 200, 0), axis=-1))] = 4
        y[np.where(np.all(sample == (0, 0, 200), axis=-1))] = 4
        # INVENTORY
        y[np.where(np.all(sample == (200, 0, 200), axis=-1))] = 5
        # LAMP/...
        y[np.where(np.all(sample == (200, 200, 0), axis=-1))] = 6
        y[np.where(np.all(sample == (0, 100, 200), axis=-1))] = 6
        y[np.where(np.all(sample == (100, 0, 200), axis=-1))] = 6
        y[np.where(np.all(sample == (200, 200, 200), axis=-1))] = 6
        y[np.where(np.all(sample == (200, 0, 0), axis=-1))] = 6
        y[np.where(np.all(sample == (0, 200, 200), axis=-1))] = 6

        result.append(y)
    return np.array(result)
```

Рисунок 70. Функция преобразования RGB в метки

Затем определим количество сегментирующих классов (7 классов) (рис. 71).

```python
# Определение цветов классов (7 классов)
FLOOR = (100, 100, 100)         # 0 класс - Пол (серый)
CEILING = (0, 0, 100)           # 1 класс - Потолок (синий)
WALL = (0, 100, 0)              # 2 класс - Стена (зеленый)
APERTURE = (0, 100, 100)        # 3 класс - Проем, Дверь, Окно (объединяем)
COLUMN = (100, 0, 0)            # 4 класс - Колонна, Перила, Лестница (объединяем)
INVENTORY = (200, 0, 200)       # 5 класс - Инвентарь (розовый)
LAMP = (200, 200, 0)            # 6 класс - Лампа, Провод, Балка, Внешний мир, Батареи, Люди (объединяем)
```

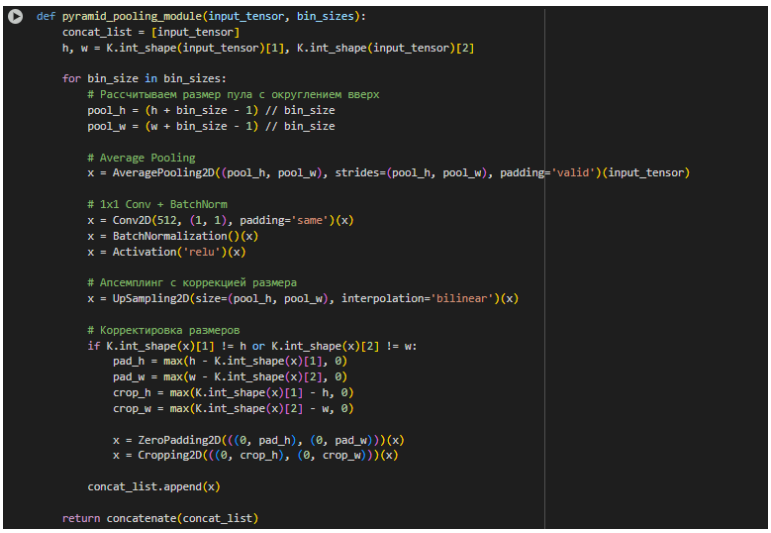
Рисунок 71. Определение цветов

Конвертируем массив изображений с метками классов в RGB-изображения с цветовым кодированием (рис. 72).

```python
# Конвертирует массив изображений с метками классов в RGB-изображения с цветовым кодированием
def labels_to_rgb(image_list):
    result = []
    for y in image_list:
        temp = np.zeros((y.shape[0], y.shape[1], 3), dtype='uint8')

        # 0 класс - FLOOR
        temp[np.where(y[..., 0] == 0)] = FLOOR
        # 1 класс - CEILING
        temp[np.where(y[..., 0] == 1)] = CEILING
        # 2 класс - WALL
        temp[np.where(y[..., 0] == 2)] = WALL
        # 3 класс - APERTURE/DOOR/WINDOW
        temp[np.where(y[..., 0] == 3)] = APERTURE
        # 4 класс - COLUMN/RAILINGS/LADDER
        temp[np.where(y[..., 0] == 4)] = COLUMN
        # 5 класс - INVENTORY
        temp[np.where(y[..., 0] == 5)] = INVENTORY
        # 6 класс - LAMP/...
        temp[np.where(y[..., 0] == 6)] = LAMP

        result.append(temp)
    return np.array(result)
```

Рисунок 72. Функция для конвертации изображений в RGB-изображения

Далее выполним функцию, которая выполняет анализ изображений в разных масштабах и комбинирует эту информацию, чтобы улучшить понимание сцены (рис. 73).



```
def pyramid_pooling_module(input_tensor, bin_sizes):
    concat_list = [input_tensor]
    h, w = K.int_shape(input_tensor)[1], K.int_shape(input_tensor)[2]

    for bin_size in bin_sizes:
        # Рассчитываем размер пула с округлением вверх
        pool_h = (h + bin_size - 1) // bin_size
        pool_w = (w + bin_size - 1) // bin_size

        # Average Pooling
        x = AveragePooling2D((pool_h, pool_w), strides=(pool_h, pool_w), padding='valid')(input_tensor)

        # 1x1 Conv + BatchNorm
        x = Conv2D(512, (1, 1), padding='same')(x)
        x = BatchNormalization()(x)
        x = Activation('relu')(x)

        # Апсемплинг с коррекцией размера
        x = UpSampling2D(size=(pool_h, pool_w), interpolation='bilinear')(x)

        # Корректировка размеров
        if K.int_shape(x)[1] != h or K.int_shape(x)[2] != w:
            pad_h = max(h - K.int_shape(x)[1], 0)
            pad_w = max(w - K.int_shape(x)[2], 0)
            crop_h = max(K.int_shape(x)[1] - h, 0)
            crop_w = max(K.int_shape(x)[2] - w, 0)

            x = ZeroPadding2D(((0, pad_h), (0, pad_w)))(x)
            x = Cropping2D(((0, crop_h), (0, crop_w)))(x)

        concat_list.append(x)

    return concatenate(concat_list)
```

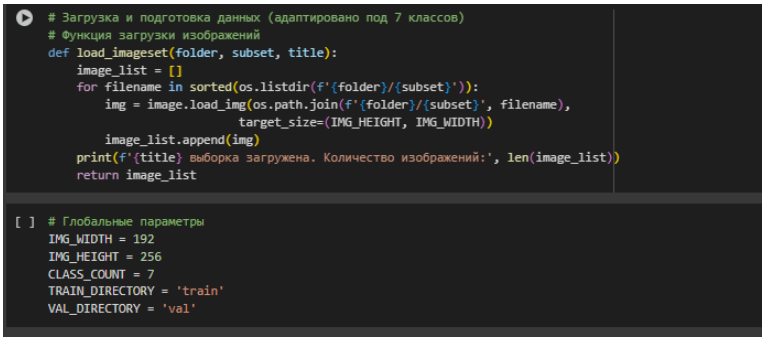
Рисунок 73. Функция обработки изображений

Затем напишем архитектуру модели PSPNet (рис. 74).

```
def PSPNet(input_shape, n_classes):
    inputs = Input(input_shape)

    # Базовая CNN
    x = Conv2D(64, 3, activation='relu', padding='same')(inputs)
    x = Conv2D(64, 3, padding='same')(x)
    x = BatchNormalization()(x)
    x = MaxPooling2D(2)(x)

    x = Conv2D(128, 3, activation='relu', padding='same')(x)
    x = Conv2D(128, 3, padding='same')(x)
    x = BatchNormalization()(x)
    x = MaxPooling2D(2)(x)

    x = Conv2D(256, 3, activation='relu', padding='same')(x)
    x = Conv2D(256, 3, padding='same')(x)
    x = BatchNormalization()(x)
    x = MaxPooling2D(2)(x)

    x = Conv2D(512, 3, activation='relu', padding='same')(x)
    x = Conv2D(512, 3, padding='same')(x)
    x = BatchNormalization()(x)

    # Pyramid Pooling
    x = pyramid_pooling_module(x, [1, 2, 3, 6])

    # Финал
    x = Conv2D(512, 3, padding='same', activation='relu')(x)
    x = Conv2D(n_classes, 1)(x)
    x = UpSampling2D(8, interpolation='bilinear')(x)
    outputs = Activation('softmax')(x)

    return Model(inputs, outputs)
```

Рисунок 74. Создание модели

Далее выполним загрузку и подготовку данных для 7 классов и зададим глобальные параметры (рис. 75).



```
# Загрузка и подготовка данных (адаптировано под 7 классов)
# Функция загрузки изображений
def load_imageset(folder, subset, title):
    image_list = []
    for filename in sorted(os.listdir(f'{folder}/{subset}')):
        img = image.load_img(os.path.join(f'{folder}/{subset}', filename),
                             target_size=(IMG_HEIGHT, IMG_WIDTH))
        image_list.append(img)
    print(f'{title} выборка загружена. Количество изображений:', len(image_list))
    return image_list
```

```
[ ] # Глобальные параметры
    IMG_WIDTH = 192
    IMG_HEIGHT = 256
    CLASS_COUNT = 7
    TRAIN_DIRECTORY = 'train'
    VAL_DIRECTORY = 'val'
```

Рисунок 75. Глобальные параметры

Выполним загрузку данных и их преобразование, а также выведем формы данных (рис. 76).

```
# Загрузка данных
train_images = load_imageset(TRAIN_DIRECTORY, 'original', 'Обучающая')
val_images = load_imageset(VAL_DIRECTORY, 'original', 'Проверочная')
train_segments = load_imageset(TRAIN_DIRECTORY, 'segment', 'Обучающая сегментированная')
val_segments = load_imageset(VAL_DIRECTORY, 'segment', 'Проверочная сегментированная')

# Преобразование данных
x_train = np.array([image.img_to_array(img) for img in train_images])
x_val = np.array([image.img_to_array(img) for img in val_images])
y_train = rgb_to_labels(train_segments)
y_val = rgb_to_labels(val_segments)

# Проверка форм данных
print("Форма x_train:", x_train.shape)
print("Форма y_train:", y_train.shape)
print("Форма x_val:", x_val.shape)
print("Форма y_val:", y_val.shape)
```

```
Обучающая выборка загружена. Количество изображений: 1900
Проверочная выборка загружена. Количество изображений: 100
Обучающая сегментированная выборка загружена. Количество изображений: 1900
Проверочная сегментированная выборка загружена. Количество изображений: 100
Форма x_train: (1900, 256, 192, 3)
Форма y_train: (1900, 256, 192, 1)
Форма x_val: (100, 256, 192, 3)
Форма y_val: (100, 256, 192, 1)
```

Рисунок 76. Вывод форм данных

Далее выполним компиляцию созданной модели (рис. 77).

```
# Очистка памяти
del train_images, val_images, train_segments, val_segments
gc.collect()
model = PSPNet((IMG_HEIGHT, IMG_WIDTH, 3), CLASS_COUNT)
model.compile(
    optimizer=Adam(learning_rate=1e-4),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
model.summary()
```

Рисунок 77. Обучение модели

Затем посмотрим на архитектуру созданной модели (рис. 78 - 80).

Model: "functional"

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 256, 192, 3)	0	-
conv2d (Conv2D)	(None, 256, 192, 64)	1,792	input_layer[0][0]
conv2d_1 (Conv2D)	(None, 256, 192, 64)	36,928	conv2d[0][0]
batch_normalization (BatchNormalizatio…	(None, 256, 192, 64)	256	conv2d_1[0][0]
max_pooling2d (MaxPooling2D)	(None, 128, 96, 64)	0	batch_normalizat…
conv2d_2 (Conv2D)	(None, 128, 96, 128)	73,856	max_pooling2d[0]…
conv2d_3 (Conv2D)	(None, 128, 96, 128)	147,584	conv2d_2[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 128, 96, 128)	512	conv2d_3[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 64, 48, 128)	0	batch_normalizat…
conv2d_4 (Conv2D)	(None, 64, 48, 256)	295,168	max_pooling2d_1[…
conv2d_5 (Conv2D)	(None, 64, 48, 256)	590,080	conv2d_4[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 64, 48, 256)	1,024	conv2d_5[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 32, 24, 256)	0	batch_normalizat…
conv2d_6 (Conv2D)	(None, 32, 24, 512)	1,180,160	max_pooling2d_2[…
conv2d_7 (Conv2D)	(None, 32, 24, 512)	2,359,808	conv2d_6[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 32, 24, 512)	2,048	conv2d_7[0][0]

Рисунок 78. Архитектура модели (часть 1)

average_pooling2d_2 (AveragePooling2D)	(None, 2, 3, 512)	0	batch_normalizat…
average_pooling2d_3 (AveragePooling2D)	(None, 5, 6, 512)	0	batch_normalizat…
conv2d_10 (Conv2D)	(None, 2, 3, 512)	262,656	average_pooling2…
conv2d_11 (Conv2D)	(None, 5, 6, 512)	262,656	average_pooling2…
average_pooling2d (AveragePooling2D)	(None, 1, 1, 512)	0	batch_normalizat…
average_pooling2d_1 (AveragePooling2D)	(None, 2, 2, 512)	0	batch_normalizat…
batch_normalizatio… (BatchNormalizatio…	(None, 2, 3, 512)	2,048	conv2d_10[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 5, 6, 512)	2,048	conv2d_11[0][0]
conv2d_8 (Conv2D)	(None, 1, 1, 512)	262,656	average_pooling2…
conv2d_9 (Conv2D)	(None, 2, 2, 512)	262,656	average_pooling2…
activation_2 (Activation)	(None, 2, 3, 512)	0	batch_normalizat…
activation_3 (Activation)	(None, 5, 6, 512)	0	batch_normalizat…
batch_normalizatio… (BatchNormalizatio…	(None, 1, 1, 512)	2,048	conv2d_8[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 2, 2, 512)	2,048	conv2d_9[0][0]
up_sampling2d_2 (UpSampling2D)	(None, 22, 24, 512)	0	activation_2[0][…
up_sampling2d_3 (UpSampling2D)	(None, 30, 24, 512)	0	activation_3[0][…
activation (Activation)	(None, 1, 1, 512)	0	batch_normalizat…
activation_1 (Activation)	(None, 2, 2, 512)	0	batch_normalizat…

Рисунок 79. Архитектура модели (часть 2)

zero_padding2d (ZeroPadding2D)	(None, 32, 24, 512)	0	up_sampling2d_2[…
zero_padding2d_1 (ZeroPadding2D)	(None, 32, 24, 512)	0	up_sampling2d_3[…
up_sampling2d (UpSampling2D)	(None, 32, 24, 512)	0	activation[0][0]
up_sampling2d_1 (UpSampling2D)	(None, 32, 24, 512)	0	activation_1[0][…
cropping2d (Cropping2D)	(None, 32, 24, 512)	0	zero_padding2d[0…
cropping2d_1 (Cropping2D)	(None, 32, 24, 512)	0	zero_padding2d_1…
concatenate (Concatenate)	(None, 32, 24, 2560)	0	batch_normalizat… up_sampling2d[0]… up_sampling2d_1[… cropping2d[0][0], cropping2d_1[0][…
conv2d_12 (Conv2D)	(None, 32, 24, 512)	11,796,992	concatenate[0][0]
conv2d_13 (Conv2D)	(None, 32, 24, 7)	3,591	conv2d_12[0][0]
up_sampling2d_4 (UpSampling2D)	(None, 256, 192, 7)	0	conv2d_13[0][0]
activation_4 (Activation)	(None, 256, 192, 7)	0	up_sampling2d_4[…

Total params: 17,548,615 (66.94 MB)
Trainable params: 17,542,599 (66.92 MB)
Non-trainable params: 6,016 (23.50 KB)

Рисунок 80. Архитектура модели (часть 3)

Далее выполним обучение созданной модели (рис. 81).

```
# Обучение модели
history = model.fit(
    x_train, y_train,
    batch_size=4,
    epochs=50,
    validation_data=(x_val, y_val)
)
```

```
Epoch 6/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 82s 128ms/step - accuracy: 0.7706 - loss: 0.6518 - val_accuracy: 0.6136 - val_loss: 1.1070
Epoch 7/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 82s 128ms/step - accuracy: 0.7800 - loss: 0.6251 - val_accuracy: 0.6569 - val_loss: 1.1042
Epoch 8/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 82s 129ms/step - accuracy: 0.7937 - loss: 0.5886 - val_accuracy: 0.6646 - val_loss: 1.0736
Epoch 9/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 82s 129ms/step - accuracy: 0.8035 - loss: 0.5538 - val_accuracy: 0.6612 - val_loss: 0.9810
Epoch 10/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 82s 129ms/step - accuracy: 0.8136 - loss: 0.5263 - val_accuracy: 0.5327 - val_loss: 1.3458
Epoch 11/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 82s 129ms/step - accuracy: 0.8132 - loss: 0.5282 - val_accuracy: 0.6457 - val_loss: 1.0607
Epoch 12/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 81s 128ms/step - accuracy: 0.8175 - loss: 0.5176 - val_accuracy: 0.6692 - val_loss: 0.9628
Epoch 13/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 60s 127ms/step - accuracy: 0.8356 - loss: 0.4637 - val_accuracy: 0.6891 - val_loss: 0.9531
Epoch 14/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 83s 129ms/step - accuracy: 0.8301 - loss: 0.4774 - val_accuracy: 0.6687 - val_loss: 1.0180
Epoch 15/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 60s 127ms/step - accuracy: 0.8499 - loss: 0.4256 - val_accuracy: 0.6453 - val_loss: 1.1125
Epoch 16/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 83s 129ms/step - accuracy: 0.8380 - loss: 0.4531 - val_accuracy: 0.6999 - val_loss: 0.9114
Epoch 17/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 61s 128ms/step - accuracy: 0.8611 - loss: 0.3864 - val_accuracy: 0.6653 - val_loss: 1.2338
Epoch 18/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 82s 128ms/step - accuracy: 0.8650 - loss: 0.3784 - val_accuracy: 0.6952 - val_loss: 0.9392
Epoch 19/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 82s 129ms/step - accuracy: 0.8655 - loss: 0.3762 - val_accuracy: 0.6365 - val_loss: 1.0396
Epoch 20/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 82s 129ms/step - accuracy: 0.8754 - loss: 0.3444 - val_accuracy: 0.7085 - val_loss: 0.9867
Epoch 21/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 82s 129ms/step - accuracy: 0.8820 - loss: 0.3281 - val_accuracy: 0.6285 - val_loss: 1.1581
Epoch 22/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 81s 128ms/step - accuracy: 0.8811 - loss: 0.3319 - val_accuracy: 0.6926 - val_loss: 1.0486
Epoch 23/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 82s 128ms/step - accuracy: 0.8935 - loss: 0.2970 - val_accuracy: 0.6938 - val_loss: 1.0034
Epoch 24/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 82s 129ms/step - accuracy: 0.8876 - loss: 0.3134 - val_accuracy: 0.7090 - val_loss: 0.9518
Epoch 25/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 81s 128ms/step - accuracy: 0.8988 - loss: 0.2815 - val_accuracy: 0.7071 - val_loss: 0.9786
Epoch 26/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 82s 129ms/step - accuracy: 0.9059 - loss: 0.2622 - val_accuracy: 0.7061 - val_loss: 1.0240
Epoch 27/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 82s 129ms/step - accuracy: 0.9066 - loss: 0.2596 - val_accuracy: 0.7115 - val_loss: 0.9373
Epoch 28/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 81s 128ms/step - accuracy: 0.9115 - loss: 0.2460 - val_accuracy: 0.7122 - val_loss: 0.9933
Epoch 29/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 60s 127ms/step - accuracy: 0.9077 - loss: 0.2560 - val_accuracy: 0.6862 - val_loss: 1.3164
Epoch 30/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 83s 128ms/step - accuracy: 0.9143 - loss: 0.2373 - val_accuracy: 0.7029 - val_loss: 1.0237
Epoch 31/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 82s 128ms/step - accuracy: 0.9186 - loss: 0.2270 - val_accuracy: 0.7145 - val_loss: 1.0868
Epoch 32/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 83s 130ms/step - accuracy: 0.9061 - loss: 0.2611 - val_accuracy: 0.7106 - val_loss: 0.9793
Epoch 33/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 60s 127ms/step - accuracy: 0.9167 - loss: 0.2309 - val_accuracy: 0.7191 - val_loss: 0.9867
Epoch 34/50
475/475 ━━━━━━━━━━━━━━━━━━━━ 61s 127ms/step - accuracy: 0.9239 - loss: 0.2106 - val_accuracy: 0.7147 - val_loss: 1.0650
Epoch 35/50
```

Рисунок 81. Обучение модели

Далее напишем функцию, которая визуализирует результаты семантической сегментации модели, сравнивая предсказания с оригинальными изображениями и разметкой (рис. 82).

```
def process_images(model,        # обученная модель
                   count = 1,    # количество случайных картинок для сегментации
                   show_original=True,  # показывать оригинальное изображение
                   show_gt=True         # показывать ground truth (разметку)
                   ):

    # Генерация случайного списка индексов
    indexes = np.random.randint(0, len(x_val), count)

    # Вычисление предсказания сети
    predict = model.predict(x_val[indexes])
    predict_labels = np.argmax(predict, axis=-1)  # получаем метки классов

    # Подготовка цветов классов для отрисовки предсказания
    pred_rgb = labels_to_rgb(predict_labels[..., None])

    # Определяем количество строк для отображения
    rows = 1
    if show_gt:
        rows += 1
    if show_original:
        rows += 1

    fig, axs = plt.subplots(rows, count, figsize=(25, 5 * rows))

    # Если только 1 изображение, делаем axs двумерным для единообразия
    if count == 1:
        axs = axs.reshape(-1, 1)

    # Отображение результатов
    for i in range(count):
        row_idx = 0

        # Предсказание модели
        axs[row_idx, i].imshow(pred_rgb[i])
        axs[row_idx, i].set_title('Предсказание модели')
        axs[row_idx, i].axis('off')
        row_idx += 1

        # разметка
        if show_gt:
            axs[row_idx, i].imshow(val_segments[indexes[i]])
            axs[row_idx, i].set_title('Оригинальная разметка')
            axs[row_idx, i].axis('off')
            row_idx += 1

        # Оригинальное изображение
        if show_original:
            axs[row_idx, i].imshow(val_images[indexes[i]])
            axs[row_idx, i].set_title('Оригинальное изображение')
            axs[row_idx, i].axis('off')

    plt.tight_layout()
    plt.show()
```

Рисунок 82. Функция визуализации результатов

И напишем функцию для визуализации сегментации (рис. 83).

```python
def process_images(model, count=1):
    indexes = np.random.randint(0, len(x_val), count)
    predict = model.predict(x_val[indexes])
    predict_labels = np.argmax(predict, axis=-1)

    pred_rgb = labels_to_rgb(predict_labels[..., np.newaxis])
    val_rgb = labels_to_rgb(y_val[indexes])

    fig, axs = plt.subplots(3, count, figsize=(20, 10))
    if count == 1:
        axs = axs.reshape(3, 1)

    for i in range(count):
        axs[0,i].imshow(pred_rgb[i])
        axs[0,i].set_title('Предсказание')
        axs[0,i].axis('off')

        axs[1,i].imshow(val_rgb[i])
        axs[1,i].set_title('Истинная сегментация')
        axs[1,i].axis('off')

        axs[2,i].imshow(x_val[indexes[i]].astype('uint8'))
        axs[2,i].set_title('Оригинал')
        axs[2,i].axis('off')

    plt.tight_layout()
    plt.show()
```

Рисунок 83. Функция для визуализации сегментации

Затем посмотрим на результаты сегментации (рис. 84).

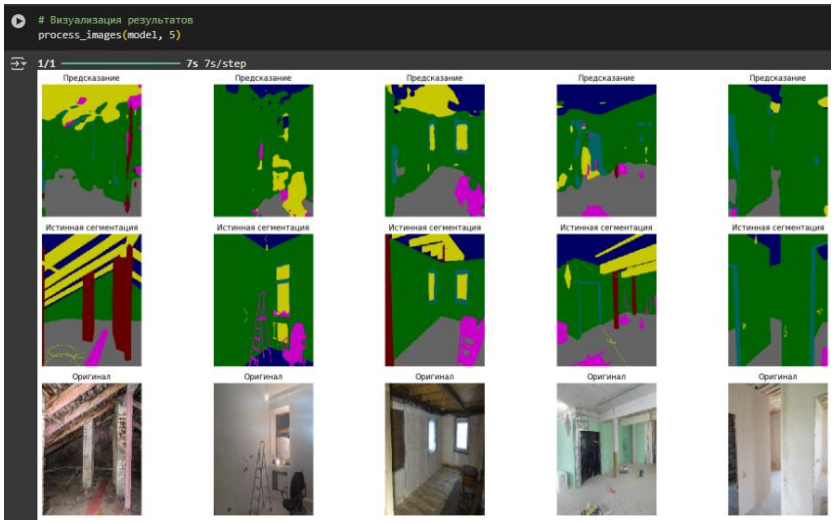


Рисунок 84. Визуализация результатов

Отсюда по заданию сделаем выводы: была реализована задача семантической сегментации строительной базы с использованием модели PSPNet. Основной целью стало сокращение исходного числа классов объектов с шестнадцати до семи, что позволило упростить структуру задачи и адаптировать модель под ключевые категории.

На этапе подготовки данных была реализована функция преобразования масок изображений из формата RGB в числовые метки классов. Это позволило модели корректно интерпретировать входные данные и обучаться на них.

Маски были созданы таким образом, чтобы объединять смежные категории в более общие группы, что повысило устойчивость модели к ошибкам классификации.

Сама модель сегментации была построена на основе архитектуры PSPNet, которая использует пирамидальную агрегацию признаков для захвата контекстной информации на разных масштабах изображения. Такой подход позволяет достичь более точного разделения объектов, особенно в сложных сценах с пересекающимися или частично закрытыми элементами.

Для повышения устойчивости обучения и предотвращения сбоев, связанных с ограничением оперативной памяти в среде выполнения, была использована ручная сборка мусора с помощью вызова gc.collect() в процессе обучения модели.

Результаты сегментации, полученные на выходе модели, продемонстрировали уверенное разделение объектов по заданным классам. Визуальный анализ показал, что модель успешно справляется с задачей распознавания ключевых элементов строительной базы.

Ссылка на гитхаб с файлами: https://github.com/EvgenyEvdakov/NS-10

Вывод: в ходе выполнения лабораторной работы были изучены методы семантической сегментации изображений с использованием сверточных нейронных сетей. Были проведены экспериментальное сравнение различных архитектур (SimpleUnet, U-Net, PSPNet) на примере базы изображений строительной тематики, а также была выполнена модификация модели под задачи с уменьшенным числом классов. Была выполнена оценка качества сегментации и влияние различных параметров архитектуры на точность модели.