МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего
профессионального образования
«Пермский государственный национальный
исследовательский университет»

Физико-математический институт

УДК 004.6

Сегментация видеопотока: Изменяющийся и прячущийся объект. Решение задачи выявления лабораторного животного в сложной экспериментальной среде, используя нейронные сети

Отчет по научно исследовательской работе

Работу выполнил студент
группы ПМИ 1 курса магистратуры
физико-математического факультета
Санников Юрий Владиславович
«___» _____2025 г.
_____

Научный руководитель:
Директор института, доктор физико-
математических наук, доцент Марина
Александровна Барулина
«___» _____2025 г.
_____

Пермь 2025

РЕФЕРАТ

Научно исследовательская работа 42 с., 35 рис., 4 табл., 15 источн., 1 прил.

НЕЙРОННЫЕ СЕТИ, СЕГМЕНТАЦИЯ ОБЪЕКТА НА ВИДЕО, ДЕТЕКЦИЯ ОБЪЕКТА НА ИЗОБРАЖЕНИИ, U-NET АРХИТЕКТУРЫ, U-NET++.

Данная работа посвящена исследованию способов сегментации объекта на видео, с дальнейшей разработки модели используя нейронные сети.

Во введении представлено обоснование актуальности темы выявления лабораторного животного в сложной экспериментальной среде с использованием моделей сегментации изображений.

Первая глава содержит обзор сегментационных моделей и анализ предметной области.

СЕГМЕНТАЦИЯ ВИДЕОПОТОКА: ИЗМЕНЯЮЩИЙСЯ И ПРЯЧУЩИЙСЯ ОБЪЕКТ. РЕШЕНИЕ ЗАДАЧИ ВЫЯВЛЕНИЯ ЛАБОРАТОРНОГО ЖИВОТНОГО В СЛОЖНОЙ ЭКСПЕРИМЕНТАЛЬНОЙ СРЕДЕ, ИСПОЛЬЗУЯ НЕЙРОННЫЕ СЕТИ

Во второй главе содержится построение моделей и разработка класса для сегментации объекта в видеопотоке.

В третьей главе проводятся сравнения сегментации объекта на рассмотренных моделях и выбор лучшей из них.

В заключении подведен итог проделанной работы и приведены перспективы развития.

Приложение А содержит ссылку на репозиторий с исходным кодом программы.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Данная работа была предложена лабораторией экспериментальной фармакологии химического факультета Пермского университета. Они занимаются строительством систем наблюдения за живым объектом в лабораторных условиях.

Для поиска лекарств против болезней и старения иногда приходится использовать системы, которые наблюдают за поведением и состоянием животных в длительные интервалы времени. Данные получаемые за этот промежуток времени могут иметь вид звукозаписи, видеопотока или данных по специфике самого животного (поведение в темноте, отдельные химические процессы).

Эксперименты проводятся на группах животных, с целью определить подействовало ли лекарство на мышь. Процесс исследования можно разделить на три фазы:

1. фаза собирания данных – получение необработанных данных с датчиков за выделенный промежуток времени,
2. подготовительная фаза – фаза обработки полученных данных,
3. последняя фаза – количественная оценка и исследование обработанных данных.

В рамках текущей работы необходимо обработать полученный видеопоток передвижений лабораторной мыши по клетке.

Во время подготовительной фазы необходимо отделить силуэт исследуемого объекта от фона. Так как вручную выделять животное покадрово очень время затратное занятие, особенно учитывая, что продолжительность видео может занимать больше 16 часов, необходимо максимально автоматизировать этот процесс. Реализация модели для сегментации объекта с изображения решает эту проблему.

Задача сегментации тела животного осложняется множеством факторов, например, таких как неоднородность изображений, изменяемость фона, отражение животного в стенках установки и т.д.

Решение задачи этого проекта необходимо для перехода к количественной оценки встречаемости и оценки характеристик последовательностей поведенческих паттернов, что является важным звеном любой серии научных исследований, которая ориентирована на поиск лекарства против нейро-болезней, таких как болезни Альцгеймера или Паркинсона.

Объектом исследования являются архитектуры сегментации объекта на изображении.

Предметом исследования являются методы автоматизации и оптимизации препроцессинга видео.

Целью Научно исследовательской работы является исследование архитектур и методов их построения и разработка программы для сегментации тела животного на видео.

Для достижения поставленной цели необходимо было решить следующие задачи:

1. изучить необходимый материал для построения архитектуры модели,
2. разработать программу, для определения каждого пикселя на принадлежность объекту животного на видео (выделение животного одним цветом, фона другим),
3. протестировать и выбрать лучшую модель для сегментации объекта в видеопотоке.

Основная задача – надежное выделение контура животного.

Вспомогательная задача – оптимизация решения.

Ограничения и возможности:

1. использовать Tensorflow;
2. использование open source модели, для возможности иметь доступ к слоям модели;
3. показать выполняющийся workflow, обязательно показать в результатах белую мышь на черном фоне как результат сегментации;
4. видео для оценки работы, не должно состоять из кадров с обучающей выборки и должно быть предоставлено лабораторией;
5. не вносить изменения в датасеты (препроцессинг должен быть встроен в тело модели);
6. сложность и типы моделей не ограничены;
7. можно собирать полностью уникальные решения.

Инструменты, использованные при разработке программы: Python, Tensorflow, CUDA, Google Colab.

1. Анализ предметной области

Далее будут рассмотрены:

- сегментирование с использованием нейронных сетей,
- архитектуры Unet и Unet++,
- обзор библиотеки cv2 для работы с видео и изображениями,

1.1. Сегментирование с использованием нейронных сетей, обзор архитектур

С развитием нейросетей за последнее десятилетие, методы анализа изображений переживают ренессанс. Нейросетевые подходы позволили повысить уровень точности и эффективности в задачах компьютерного зрения. Новые методы превосходят традиционные в сложных задачах сегментации изображения и видео. Задача сегментации объекта на картинке, представляет собой классификацию по пикселям, определение принадлежности пикселя объекту или фону.

Современные методы сегментации изображений обычно адаптированы с архитектур свёрточных нейронных сетей (CNN – Convolution Neuro Network) или трансформеров [1]. Существует множество архитектур, разработанных для решения данной задачи. К наиболее распространенным архитектурам относятся:

- Сети U-Net, которые будут подробно рассмотрены в следующем пункте;
- DeepLab (2015) [2] архитектуры, используют ASPP (Atrous Spatial Pyramid Pooling) свёрточные сети, улавливающие контекст на разных масштабах, последние версии DeepLabV2 и DeepLabv3+ (2021) позволяют более точно восстанавливать пространственную информацию, выводя ее в число ведущих отраслей в этой сфере. Общая схема метода DeepLabV3 и DeepLabv3+ представлена на рисунке 1.

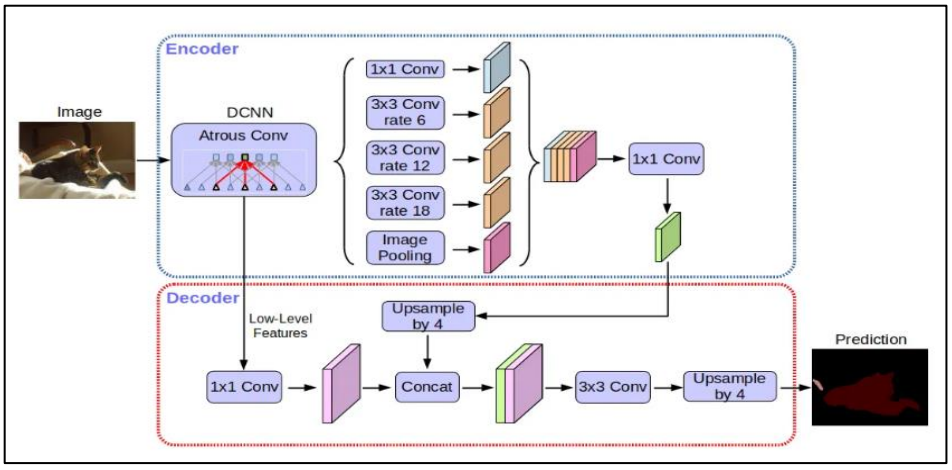

Рисунок 1. *Общая схема метода DeepLabV3 и DeepLabv3+.*

– Mask R-CNN (2017) [3], представляет собой улучшенную версию R-CNN (Region-based Convolutional Neural Network) [4]. Главное новшество Mask R-CNN заключается в внедрении механизма сегментации на уровне масок, который позволяет точно определить границы объектов, что делает возможным не только выделение классов объектов, но и их формы. Общая схема метода Mask R-CNN представлена на рисунке 2.

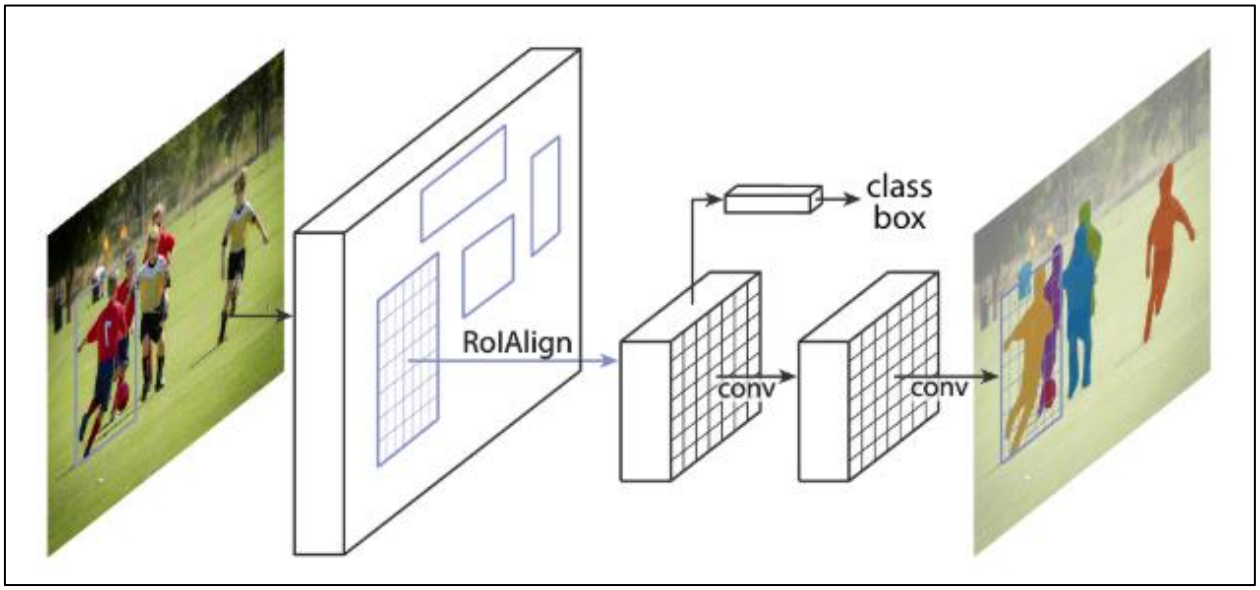

Рисунок 2. *Общая схема метода Mask R-CNN.*

– SegNet (2015) [5]. Была разработана для того, чтобы эффективно решать задачи сегментации с относительно низкими вычислительными затратами, особенно для приложений, требующих высоких разрешений изображений. Не смотря на свою эффективность и оптимизированность, главным недостатком такой модели является ее точность. Схема архитектуры SegNet представлена на рисунке 3.

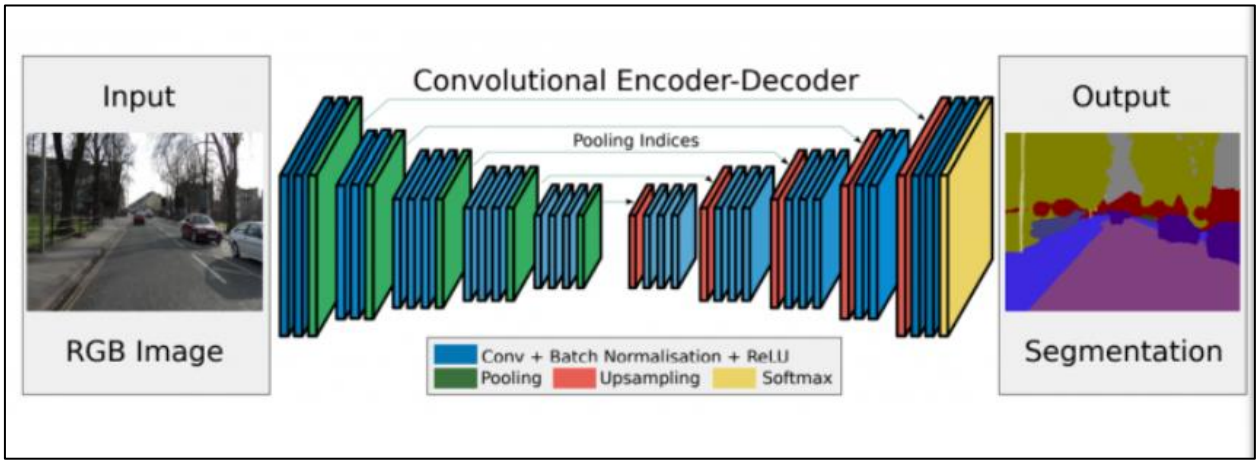

Рисунок 3. *Общая схема архитектуры SegNet.*

– Transformer-базированные архитектуры ViT (VisionTransformer) (2021) [6] представляют изображения в виде последовательных патчей 16x16 пикселей, которые равномерно преобразуются в векторные представления и подаются на вход трансформеру. В отличие от CNN, ViT не использует свертки, что позволяет захватывать более обширные контексты. К недостаткам метода можно отнести необходимость большого количества данных для обучения. Схема архитектуры Visual Transformer проиллюстрирована на рисунке 4.

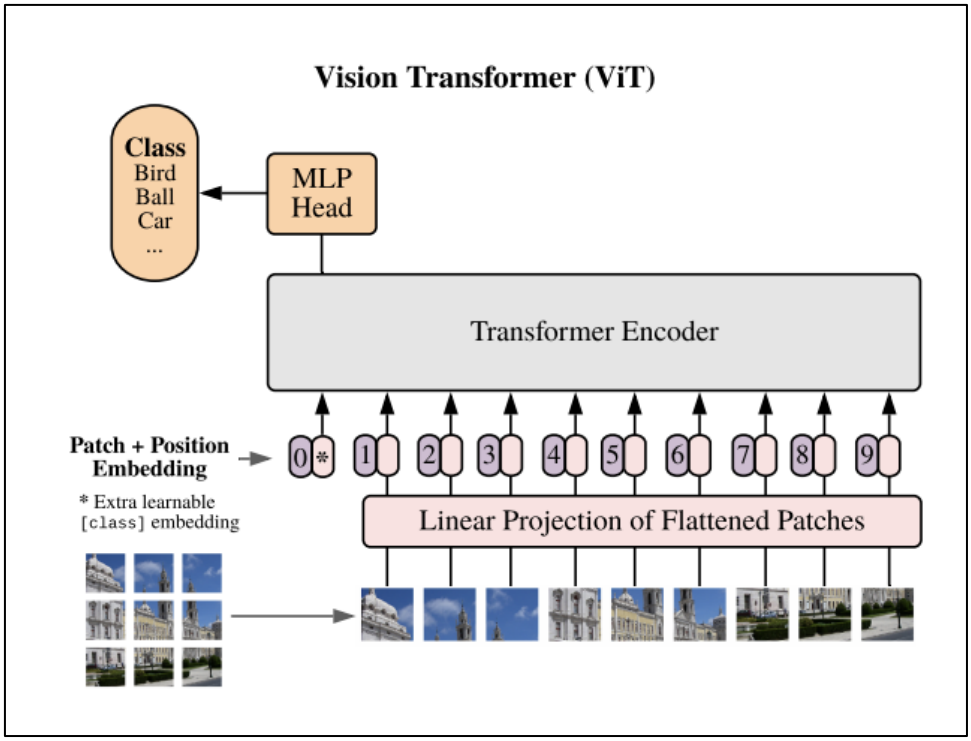

Рисунок 4. *Схема архитектуры Visual Transformer.*

Основными недостатками описанных выше методов, являются чрезмерные вычислительные затраты из-за сложных архитектур.

1.2. Архитектуры U-Net

U-Net является одной из стандартных архитектур CNN для задач сегментации изображений, использующаяся, для сегментации области изображения по классам. Для U-Net характерно получение хороших результатов в различных реальных задачах, при использовании небольшого количества данных. Данный метод был выбран для углубленного изучения, в связи с простотой построения архитектуры и ее понимания.

Будут рассмотрены:

– U-Net,

– U-Net++.

1.2.1. U-Net

Архитектура U-Net, изначально разработанная Олафом Роннебергером и др. в 2015 году [7], использовалась для медицинских целей в сегментации биомедицинских изображений. Свое название она получила, из-за U–образной формы схемы модели, представленной на рисунке 5.

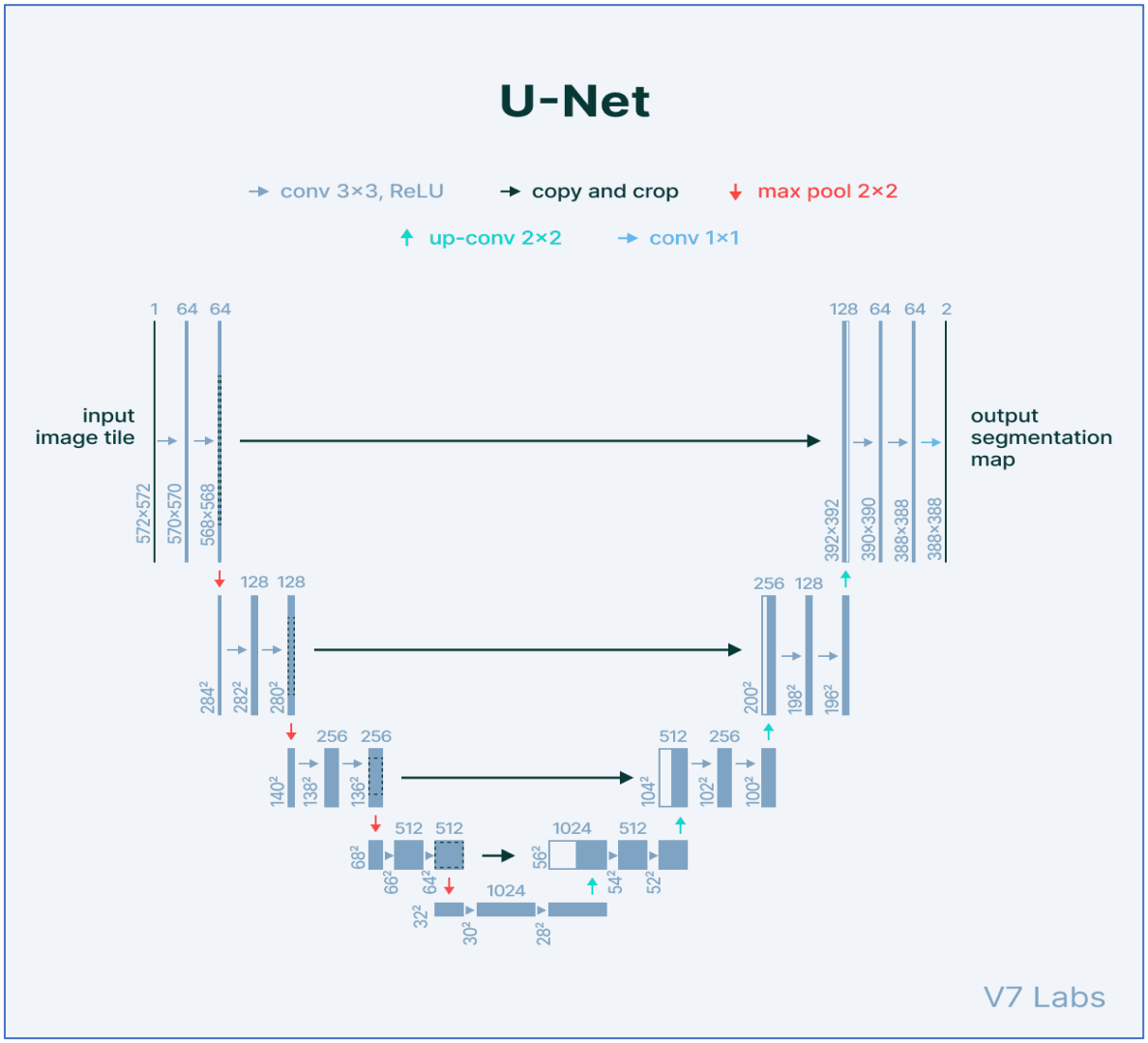

Рисунок 5. *Схема архитектуры U-Net.*

На рисунке 5, в левой части схемы располагается свёрточная сеть – Encoder, в правой стороне Decoder – расширяющийся путь [8]. Ключевой идеей схемы, заключается в том, что каждый промежуточный результат encoder-a конкатенируется с промежуточным результатом правой части архитектуры. Такое решение (skip-connection) было представлено для решения проблемы потери информации во время операции max-pooling.

Encoder работает следующим образом, если на вход подается изображение 572x572x1 и с помощью операции свертки (convolution 3x3) с функцией активации ReLU, и

нулевым отступом, для предотвращения потери информации на границах. Слой, создаваемый сверткой, позволяет изучать более локальные признаки, например, обнаружение глаза на изображении. Пример свертки 3x3 представлен на рисунке 6.

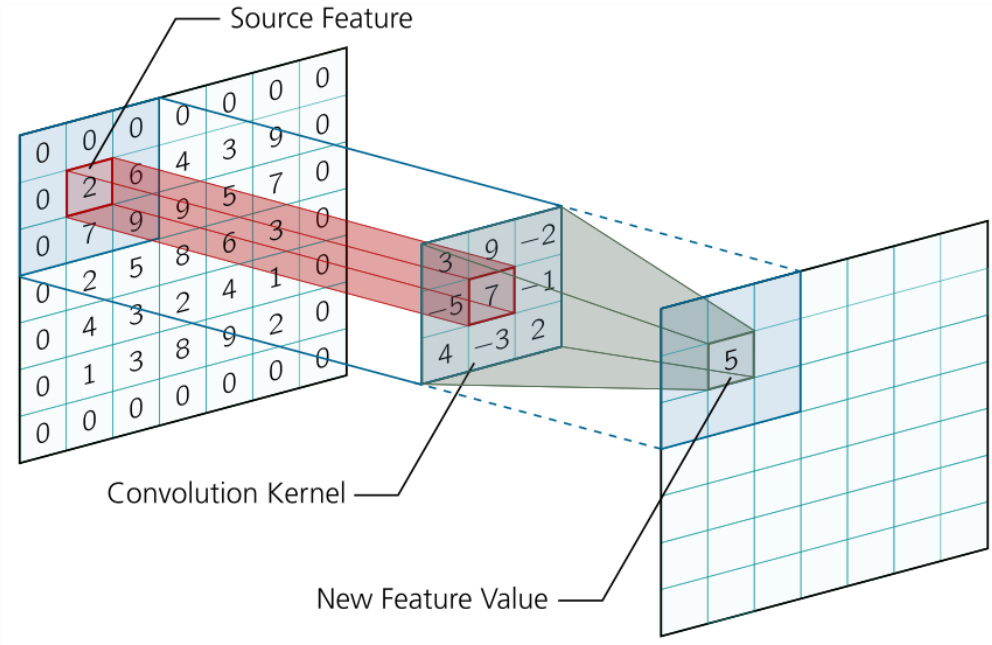

Рисунок 6. *Пример свертки 3x3.*

После данных манипуляций получаются тензоры 572x572x64. Следующим шагом применяется maxPooling 2x2, уменьшающий размер тензора по ширине и высоте вдвое. Помимо того, что данный шаг помогает уменьшить количество параметров модели, тем самым предотвратив переобучение, он также повышает надежность модели, предотвращая неточности при небольших изменениях во входных данных. Суть метода maxPooling 2x2 представлена на рисунке 7.

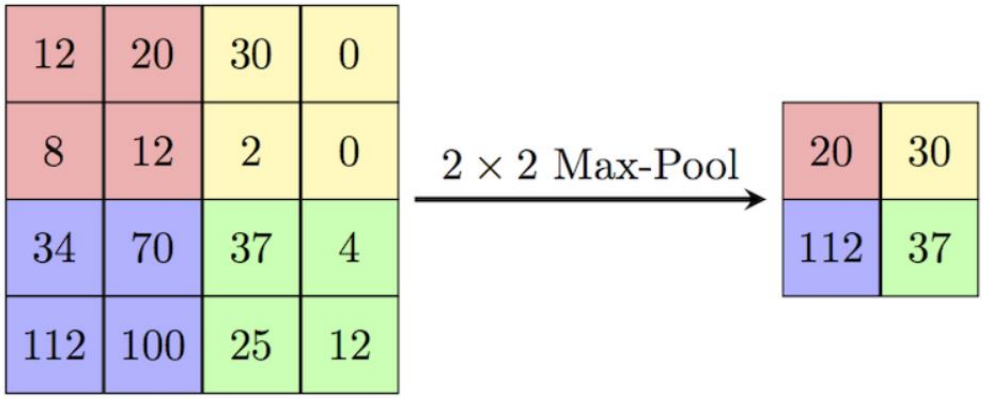

Рисунок 7. *Пример maxPooling 2x2.*

Эти два шага повторяются до момента, когда тензор не станет размером 28x28x1024.

Основной целью декодера, является реконструкция карты сегментации высокого разрешения из изученных признаков, тем самым восстановив пространственную детализацию. Так как при max-pooling терялась важная информация, при выполнении восстановления изображения с помощью up-sampling результат получался не точным, из-за потери контуров объектов, поэтому для приведения тензора к целевому размеру используется up-convolution 2x2 или conv2dTranspose 2x2, после чего берется тензор 64x64 с предыдущего этапа и конкатенируется с полученным результатом up-conv, чтобы размеры тензоров совпадали, от предыдущий тензор обрезают до 56x56. Данная операция позволяет восстановить признаки, утраченные во время сжатия до тензора 28x28. На рисунке 8 представлена часть схемы с переходом от свёрточной части к восстанавливающей.

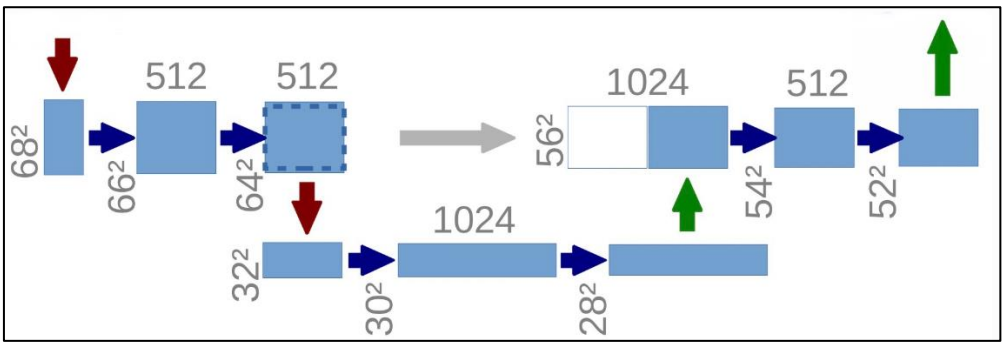

Рисунок 8. *Переход от encoder части к decoder*.

1.2.2. U-Net++

U-Net++, представленная в 2018 году является модификацией существующей модели U-Net [9]. Для улучшения показателей традиционной U-Net было предложено использование плотных пропускных соединений (skip-connections). На рисунке 9 представлена схема U-Net++.

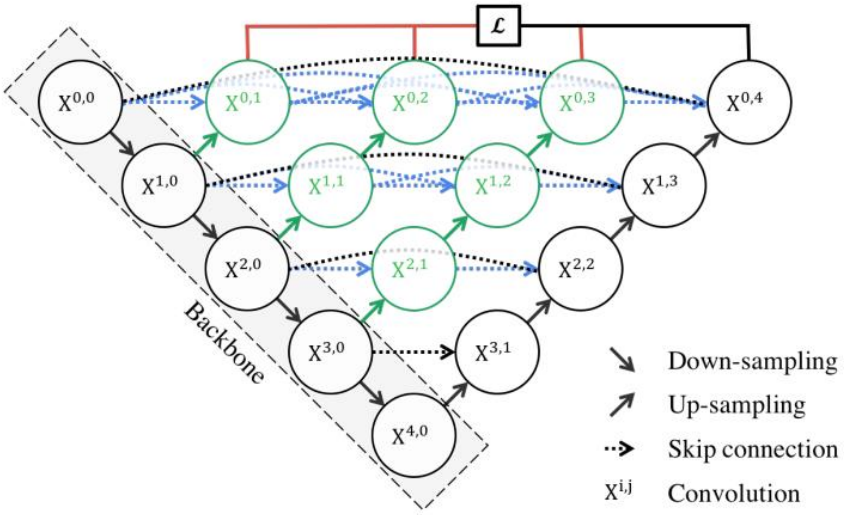

Рисунок 9. *Архитектура U-Net++*.

Обновленный дизайн архитектуры предлагает проводить up-convolution с предыдущими параметрами на каждом шаге свертки, полученный skip-connection передается далее, пока не упирается в декодер. Каждый такой мост строится за счет соединения (concatenate) предыдущих узлов up-sampling с свертыванием на пропускном соединении (рисунок 10). Такой подход позволяет более точно сократить семантический разрыв при восстановлении изображения.

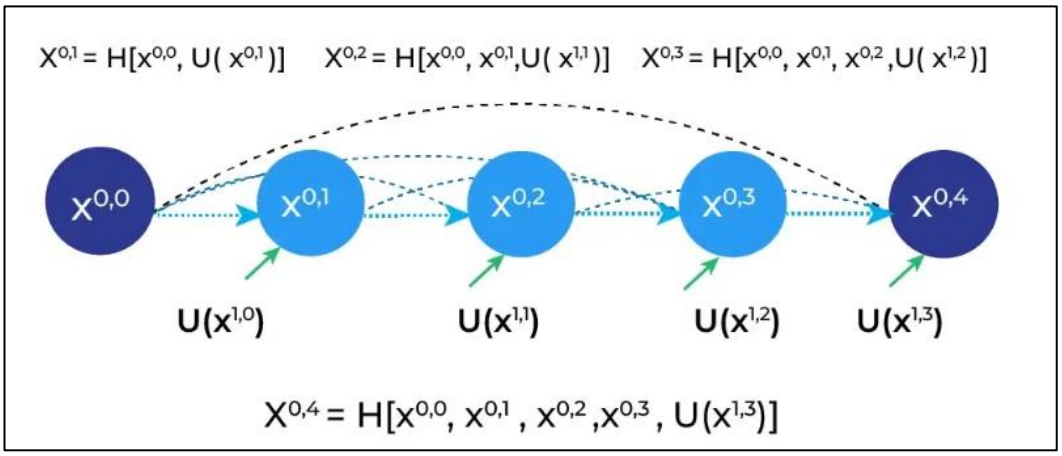

Рисунок 10. *Оптимизация skip-connection с использованием комбинирования узлов.*

Количество шагов вниз и вверх в этих моделях обозначаются букой L. Экспериментальным путём было доказано, что архитектура U-Net++ L3 при только 3-х слоях обучается на 32,2% быстрее чем U-Net++L4, при минимальном отклонении в итоговом результате [10]. Схема U-Net++L3 представлена на рисунке 11.

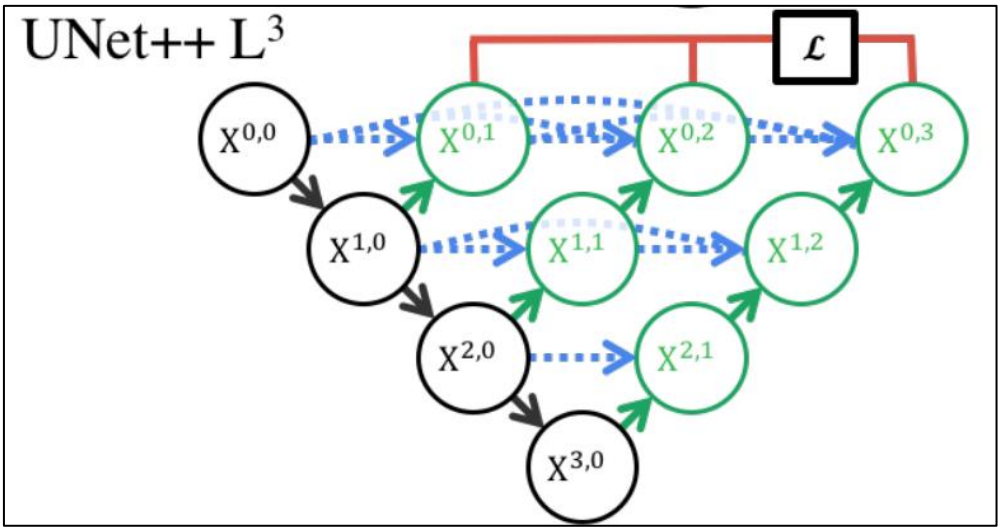

Рисунок 11. *Архитектура U-Net++L3.*

1.3. Обзор библиотеки cv2

Cv2 это модуль OpenCV в Python - открытой библиотеки для работы с алгоритмами компьютерного зрения [11]. В рамках работы используется для преобразований изображений и видео. Хотя одним из пунктов нужно было реализовать препроцессинг изображений внутри модели, из-за неудачных попыток имплементации этого способа на протяжении пары недель, было решено использовать для преобразований данную библиотеку.

В этом пункте будут рассмотрены:

- методы для работы с изображением,
- методы для работы с видео.

1.3.1. Методы для работы с изображением

В текущем подпункте будут рассмотрены методы для работы с изображением, использовавшиеся во время разработки:

– cv2.imread (path, cv2.IMREAD_GRAYSCALE) чтение картинки, по указанному пути, в данной работе изображения сразу считывались как черно-белые с помощью cv2.IMREAD_GRAYSCALE;

– cv2.resize (img, x, y) – изменяет масштаб картинки, принимает на вход изображение и аргументы ширины с высотой;

– cv2.GaussianBlur (img, (13,13), 0) – размывает изображение, вторым аргументом подается квадратная матрица (ядро свертки), где пиксель в центре матрицы устанавливается как взвешенное среднее значение у соседних пикселей, чем больше матрица, тем больше размытие, третьим аргументом устанавливается стандартное отклонение ядра Гаусса;

– cv2.threshold(blur, 100, 255, cv2.THRESH_BINARY) - позволяет применять порог к каждому пикселю изображения. Она принимает четыре параметра: исходное изображение, значение порога, максимальное значение и тип thresholding, в работе используется порог cv2.THRESH_BINARY, пиксели выше порога получают максимальное значение (255 – белый), пиксели ниже порога получают значение 0 (черный), в рамках текущей работы, блюр изображения с помощью GaussiznBlur, в связке с применением фиксированного порога используются, чтобы убрать лишние шумы с предсказанной маски;

– cv2.addWeighted(mask, alpha, original_image, 1-alpha, 0, original_image) – используется для наложения на предсказанное изображение исходного с некоторой

прозрачностью, используется в работе модели, для предсказания маски у единственного изображения;

– cv2.cvtColor(img, cv2.COLOR_BGR2RGB) - преобразует изображение из одного цветового пространства в другое, так-как в OpenCV формат цвета по умолчанию имеет формат цвета BGR, для корректной работы рекомендуется преобразовать его в RGB с помощью cv2.COLOR_BGR2RGB;

– cv2_imshow(img, name) - создаёт окно с заданным именем и отрисовывает в нём изображение;

– cv2.imwrite (save_image_path, img) – сохраняет изображение по указанному пути.

1.3.2. Методы для работы с видео

В подпункте будут рассмотрены методы для работы с видеорядом, использовавшиеся во время разработки:

– cv2.VideoCapture(fileName) – функция для захвата видео записывается в значение cap, на вход подается путь к файлу, далее в цикле while True: с помощью cap.read() считывается кадр видео, данная функция возвращает 2 значения (логическое значение корректности считывания кадра и сам кадр);

– cap.get(cv2.CAP_PROP_FPS) – используется для получения кадров в секунду у обрабатываемого видео, fps (frames per second) понадобятся для записи полученных изображений в видео;

– cv2.waitKey(1) - функция, прерывающая выполнение бесконечного цикла, ждет нажатия клавиши 1 миллисекунду, чтобы проверить нужно ли освобождать захват видео с помощью cv2.destroyAllWindows() и выходить из цикла;

– cv2.VideoWriter(pathToVideo, cv2.VideoWriter_fourcc(*'DIVX'), fps, (width, height)) - функция для записи видео в файл, на вход подается путь до файла для сохранения, переменная fourcc - 4-байтовый код, который используется для указания видеокодека, количество кадров в секунду и размер видео в пикселях;

– video.write(cv2.imread(os.path.join(pathToVideoImages, image))) – используется для конвертации папки изображений в видеоряд, в рамках работы программы, во время сегментации, обработанные изображения сохраняются в папку, чтобы в случае вылета программы полученный результат можно было склеить в один видеоряд.

2. Проектирование и разработка

Во время разработки я столкнулся с рядом ограничений, среди перечня технических требований к реализации проекта, необходимо было иметь:

1. Компьютер, оснащенный:

– Видеокартой фирмы Nvidia с объемом внутренней оперативной памяти не менее 6 Гб и поддержкой технологии параллельных вычислений CUDA;

– Оперативной памятью, объемом не менее 16 Гб;

– Свободной постоянной памятью объемом не менее 150 Гб.

2. Программное обеспечение:

– Язык программирования Python 3 с дистрибутивом Anaconda3;

– Программная библиотека для машинного обучения «TensorFlow» с высокоуровневым API «Keras»;

– Библиотека для работы с компьютерным зрением «OpenCV»;

– Программное обеспечение для проведения параллельных вычислений «CUDA» версии, соответствующей драйверу используемой видеокарты;

– Библиотека с поддержкой GPU примитивов для глубоких нейронных сетей «CuDNN» версии, соответствующей версии «CUDA»;

– Среду разработки JupyterLab.

По моим характеристикам мой ноутбук не удовлетворял требованиям:

– Видеокарта AMD Radeon Vega Graphics, без поддержки технологии параллельных вычислений CUDA;

– Оперативная память объемом 6 гб;

– Свободная память объемом 25 гб;

– Из-за отсутствия CUDA, программная библиотека для машинного обучения «TensorFlow» с высокоуровневым API «Keras» не позволяет обучать модели.

В связи с этим вместо среды разработки Jupiter Lab было решено использовать Google Colab - облачный сервис от Google, который позволяет писать и выполнять Python-код в браузере без установки дополнительных программ [12]. В Colab есть возможность подключения к удаленной среде выполнения с графическим аппаратным ускорением T4, что позволяет использовать библиотеку Tensorflow для обучения нейросетевых моделей. Характеристики графического процессора представлены на рисунке 12.

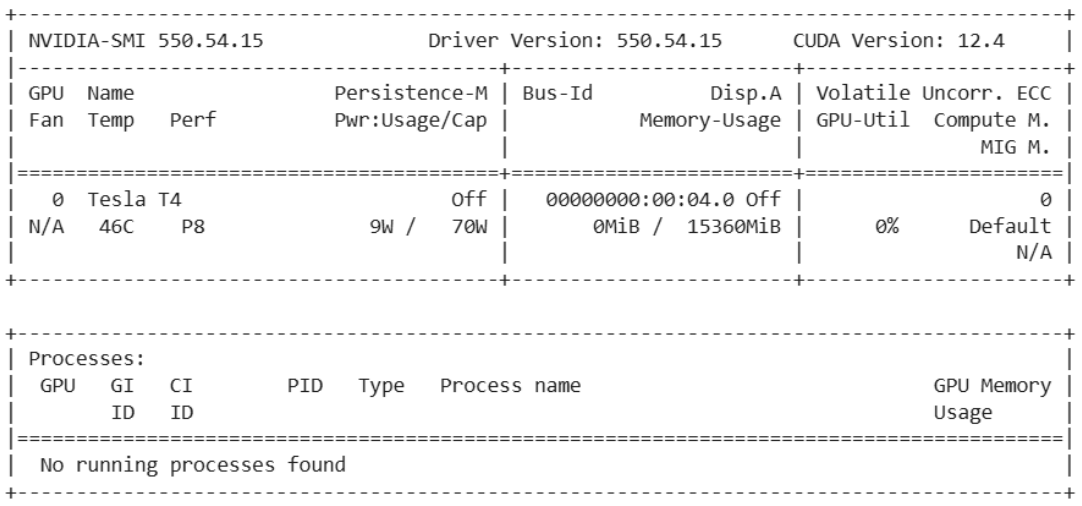

Рисунок 12. *Характеристики графического процессора T4.*

В качестве файлового менеджера был выбран Google-disk, так-как Google Colab не сохраняет переменные и файлы текущего сеанса после выхода. Функции для работы с обучением модели, были импортированы из библиотеки tensorflow.keras. К таковым относятся ModelCheckpoint, ReduceLROnPlateau, CSVLogger и т.д. из tensorflow.keras.callbacks.

Далее будут рассмотрены:

- обзор класса MiceSegmentationClass для сегментации видеопотока,
- реализация архитектуры моделей U-Net,
- функции для работы с видео.

2.1. Обзор класса MiceSegmentationClass для сегментации видеопотока

Класс MiceSegmentationClass был разработан для удобной работы с видео и обучением моделей. К неизменяемым переменным относятся:

- globalShapeSize = 256 (дефолтный размер обрабатываемого изображения),
- unet = "U-Net" (название модели Unet),
- unetModified = "U-Net-modified" (название модели UnetModifiedModel),
- unetPlusPlusModified = "U-Net-plus-plus-modified" (название модели UnetPlusPlusModifiedModel),
- fileExt = ".keras" (расширение загружаемых сохраняемых моделей),
- cropValue = 5 (число пикселей которые нужно обрезать по краям изображения),
- lastVideoImages = 'videoImg' (папка с изображениями от последней сегментации видео).

Во время инициализации класса, входными параметрами можно указать путь до папки с данными до обучения (pathToFolder), уже обученную модель U-Net (UnetModel), уже обученную модель U-Net_modified (UnetModifiedModel), уже обученную модель U-Net++ (UnetModelPlusPlusModifiedModel). Тело инициализатора выглядит следующим образом (рисунок 13):

```
def __init__(self, pathToFolder =
'/content/drive/MyDrive/miceSegmentation/', UnetModel = [],
UnetModifiedModel = [], UnetPlusPlusModifiedModel = []):
        self.UnetModel = UnetModel
        self.UnetModifiedModel = UnetModifiedModel
        self.UnetPlusPlusModifiedModel = UnetPlusPlusModifiedModel
        self.pathToFolder = pathToFolder # путь до рабочей папки
        self.databasePath = pathToFolder # путь до папки с датасетами
        # путь до папки, для сохранения отдельно предсказанных изображений
        self.savedImagesPath = pathToFolder+'save_images'
        # датасет для обучения
        self.train_dataset = []
        # датасет для валидации
        self.valid_dataset = []
        # количество кадров в секунду
        self.fps = 24
        # Модель использующаяся для сегментации
        self.currentModel = ''
```

Рисунок 13. *Тело инициализатора класса MiceSegmentationClass.*

Функция тренировки модели разбита на три части. Сначала с помощью вызова функции self.trainModel, формируется модель для обучения, количество эпох обучения, количество шагов для тренировки и валидации, колбеки для пользовательского изменения обучения модели. Вторым шагом происходит обучения с вызовом функции model.fit(args). Обученная модель сохраняется в классе и ее можно будет использовать для сегментации. Функция тренировки модели представлена на рисунке 14:

```
    def trainAndFitModel(self, modelType = ''):
      if not self.checkCorrectModelType(modelType):
        return
      if (len(self.databasePath) > 0):
        self.currentModel = modelType
        model, epochs, train_steps, valid_steps, callbacks =
self.trainModel(self.databasePath, modelType)
        print(model)
        self.fitModel(model, epochs, train_steps, valid_steps, callbacks)
        if (modelType == self.unet):
          self.UnetModel = model
        if (modelType == self.unetModified):
          self.UnetModifiedModel = model
        if (modelType == self.unetPlusPlusModified):
          self.UnetPlusPlusModifiedModel = model
      else:
        print("Choose correct folder with Train/Test data")
```

Рисунок 14. *Тело функции для тренировки модели.*

Подробнее рассмотрим функцию trainModel. Сначала методом подбора были найдены оптимальные гиперпараметры для каждой модели:

- batch_size (количество изображения, подаваемых модели на одном шаге обучения, для U-Net = 8, для U-Net++ = 16),
- epochs (количество полных обходов по всему набору данных, для обучения модели, для U-Net = 14, для U-Net++ = 5),
- lr (темп обучения, подаваемый в Adam optimizer из библиотеки Tensorflow, для минимизации функции потерь и повышения производительности, для U-Net и U-Net++ = 0.0001, для U-Net++ = 16).

Загрузка и препроцессинг изображений осуществляются сначала с помощью функции load_data, на вход которой подаётся значение batch_size, функция собирает список всех тренировочных и валидационных данных (путей до изображений), округляет их количество относительно batch_size по формуле (floor(<длина массива>/batch_size)-1)*batch_size, где floor – округление в нижнюю сторону, и перемешивает полученный массив. На рисунке 15 представлена функция load_data.

```
 # загрузка данных
 def load_data(self, batch_size):
     dataset_path = self.databasePath
     # Загрузка изображений
   train_images = sorted(glob(os.path.join(dataset_path, "images/*")))
   train_masks = sorted(glob(os.path.join(dataset_path, "masks/*")))
   test_images = sorted(glob(os.path.join(dataset_path, "t_images/*")))
   test_masks = sorted(glob(os.path.join(dataset_path, "test_masks/*")))
   # Длина тренировочных и тестовых данных
   lenTrain = len(train_images)
   lenTest = len(test_images)
   print(len(train_images), len(train_masks), len(test_images),
len(test_masks))
   # Получить реальную длину данных
   countTrain = (math.floor(lenTrain/batch_size)-1)*batch_size
   countTest = (math.floor(lenTest/batch_size)-1)*batch_size
   print(countTrain, countTest)
   # Перемешать тренировочные и тестовые данные
   train_x, train_y = shuffle(train_images[:countTrain],
train_masks[:countTrain], random_state=42)
   test_x, test_y = shuffle(test_images[:countTest],
test_masks[:countTest], random_state=42)
   return (train_x, train_y), (test_x, test_y)
```

Рисунок 15. *Функция load_data для получения путей до изображений обучающей и проверяющей выборок.*

Далее полученные массивы передаются в функцию self.tf_dataset, для препроцессинга изображений и формирования датасетов. Функция перемешивает данные с помощью shuffle и делит датасет на батчи. Препроцессинг изображений происходит при

загрузке с помощью функции preprocessingModelImages. Маскам и оригинальным изображениям меняют размер до 256x256px, обрезая крайние 10px изображения, после чего устанавливают шейп (кортеж размеров изображения) [256, 256, 3] и [256, 256, 1] для оригинальных и масок. Установка шейпа необходима для корректной работы алгоритма по обучению модели.

Для компиляции модели подаются функция потерь binary crossentropy, оптимизатор Adam, и метрики, mean IoU (среднее значение, оценивающее степень пересечения между предсказанной и реальной областями, рассчитывается на полном наборе данных, чем выше значение, тем лучше сегментация), Recall (полнота, показывает какую долю реальных положительных случаев модель смогла правильно предсказать, чем выше значение, тем лучше сегментация), Precision (точность, показывает какая доля предсказанных моделью положительных случаев действительно является положительной). Компиляция модели представлена на рисунке 16.

```
model.compile(
            loss="binary_crossentropy",
            optimizer=tf.keras.optimizers.Adam(lr),
            metrics=[
                tf.keras.metrics.MeanIoU(num_classes=2),
                tf.keras.metrics.Recall(),
                tf.keras.metrics.Precision()
            ]
        )
```

Рисунок 16. *Компиляция модели.*

Для регулирования обучения указываются ряд функций-колбэков:

– ReduceLROnPlateau (метод планирования обучения, уменьшающий скорость обучения, когда контролируемый показатель перестаёт улучшаться в течение определённого количества эпох),

– ModelCheckpoint (сохраняет модель и ее веса по указанному пути, после каждой эпохи),

– CSVLogger (записывает результаты метрик в csv-файл после каждой эпохи по указанному пути),

– EarlyStopping (останавливает обучение, когда указанная метрика перестает улучшаться).

– Функции-колбэки представлены на рисунке 17.

```
 callbacks = [ModelCheckpoint(model_path,
monitor="val_loss",save_best_only=True, verbose=0),
             ReduceLROnPlateau(monitor="val_loss", patience=5,
factor=0.1, verbose=1),
             CSVLogger(csv_path),
             EarlyStopping(monitor="val_loss", patience=10),]
```

Рисунок 17. *Функции-колбэки.*

2.2. Реализация архитектуры моделей U-net

Функции для работы со слоями, использующиеся при реализации архитектуры моделей были импортированы из библиотеки tensorflow.keras. К таковым относятся UpSampling2D,Conv2D, Dropout и т.д из tensorflow.keras.layers. Были разработаны:

- функции для работы со свёрточными слоями,
- реализация архитектуры U-net,
- реализация архитектуры U-net_modified,
- реализация архитектуры U-net++L3,
- реализация архитектуры U-net++L4,

2.2.1. Функции для работы со свёрточными слоями

Для свертки стандартной модели U-Net используется функция conv_block, для U-Net++ conv_block_plus_plus. Функции идентичны, дважды добавляются три слоя - свертка с ядром 3x3 и одинаковыми отступами по краям картинки, батч-нормализация (слой, нормализирующий входные данные сети, для оптимизации обучения модели и борьбы с переобучением за счет стандартизации входов каждого из слоев) и функция активации, в случае с U-Net это relu. На рисунке 18 представлена реализация.

```
    def conv_block(self, inputs, num_filters):
      x = Conv2D(num_filters, 3, padding='same')(inputs)
      x = BatchNormalization()(x)
      x = Activation("relu")(x)

      x = Conv2D(num_filters, 3, padding='same')(x)
      x = BatchNormalization()(x)
      x = Activation("relu")(x)

      return x
    def conv_block_plus_plus(self, inputs, num_filters):
      x = tf.keras.Sequential([
        # Convolutional Layer
        Conv2D(num_filters, 3, padding='same'),
        # Batch Normalization Layer
        BatchNormalization(),
        # Activation Layer
        Activation('relu'),
        # Convolutional Layer
        Conv2D(num_filters, 3, padding='same'),
        # Batch Normalization Layer
        BatchNormalization(),
        # Activation Layer
        Activation('relu')
    ])(inputs)
      return x
```

Рисунок 18. *Функции для свертки в архитектуре моделей U-Net и U-Net++.*

Описанные функции используются блоках encoder и в decoder. Шаг сжатия изображения описан в функции encoder_block. Для U-Net он состоит из конволюции предыдущего слоя с помощью self.conv_block и MaxPool с ядром 2x2. Реализация для U-Net++ отличается тем, что MaxPool производится с использованием strides(2,2), шаг смещения равен 2 по x и y вместо 1. На рисунке 19 описана функция encoder_block.

```
    def encoder_block(self, inputs, num_filters, isPlusPlus = False):
      if isPlusPlus:
        x = self.conv_block_plus_plus(inputs, num_filters)
        p = MaxPool2D((2, 2), strides=(2, 2))(x)
        return x, p
      else:
        x = self.conv_block(inputs, num_filters)
        p = MaxPool2D((2, 2))(x)
        return x, p
```

Рисунок 19. *Функция encoder_block.*

Decoder, восстанавливающий пространство изображения, увеличивая его разрешение, описан в функции decoder_block. Для U-Net он состоит из транспонирования предыдущего слоя с шагом (2,2) и одинаковыми отступами, далее происходит конкатенирование с промежуточным результатом правой части, для восстановления

потерянных при свертке данных, в конце шага с помощью self.conv_block происходит свертка. Реализация для U-Net++ отличается тем, что вместо транспонирования используется upSampling с ядром (2,2) для увеличения размера изображения. На рисунке 20 описана функция decoder_block.

```
    def decoder_block(self, inputs, skip_features, num_filters,
isPlusPlus = False):
      if isPlusPlus:
        x = UpSampling2D((2, 2))(inputs)
        toConcat = skip_features + [x]
        x = concatenate(toConcat)
        x = self.conv_block_plus_plus(x, num_filters)
      else:
        x = Conv2DTranspose(num_filters, (2, 2), strides=2,
padding='same')(inputs)
        x = concatenate([x, skip_features])
        x = self.conv_block(x, num_filters)
      return x
```

Рисунок 20. *Функция decoder_block.*

2.2.2. Реализация архитектуры U-Net

Согласно архитектуре, обычный U-Net состоит из 4 encoder_block и 4 decoder_block. В конце вызывается Conv2D с ядром (1,1) одинаковыми отступами и сигмоидальной функцией активации. На рисунке 21 представлена реализация U-Net архитектуры.

```
def build_unet(self, input_shape):
      tf.keras.backend.clear_session()
      inputs = Input(input_shape)

      """ Encoder """
      s1, p1 = self.encoder_block(inputs, 64)
      s2, p2 = self.encoder_block(p1, 128)
      s3, p3 = self.encoder_block(p2, 256)
      s4, p4 = self.encoder_block(p3, 512)
      b1 = self.conv_block(p4, 1024)

      """ Decoder """
      d1 = self.decoder_block(b1, s4, 512)
      d2 = self.decoder_block(d1, s3, 256)
      d3 = self.decoder_block(d2, s2, 128)
      d4 = self.decoder_block(d3, s1, 64)

      """ Output """
      outputs = Conv2D(1, (1, 1), padding="same",
activation="sigmoid")(d4)
      return Model(inputs, outputs, name=self.unet)
```

Рисунок 21. *Реализация U-Net архитектуры.*

2.2.3. Реализация архитектуры U-Net_modified

Для реализации модифицированной архитектуры U-Net использовались 5 encoder_block и 5 decoder_block. На рисунке 22 представлена реализация U-Net_modified архитектуры.

```
    def build_unet_modified(self, input_shape):
      tf.keras.backend.clear_session()
      inputs = Input(input_shape)

      """ Encoder """
      s1, p1 = self.encoder_block(inputs, 64)
      s2, p2 = self.encoder_block(p1, 128)
      s3, p3 = self.encoder_block(p2, 256)
      s4, p4 = self.encoder_block(p3, 512)
      s5, p5 = self.encoder_block(p4, 1024)
      b1 = self.conv_block(p5, 2048)

      """ Decoder """
      d0 = self.decoder_block(b1, s5, 1024)
      d1 = self.decoder_block(d0, s4, 512)
      d2 = self.decoder_block(d1, s3, 256)
      d3 = self.decoder_block(d2, s2, 128)
      d4 = self.decoder_block(d3, s1, 64)

      """ Output """
      outputs = Conv2D(1, (1, 1), padding="same",
activation="sigmoid")(d4)
      return Model(inputs, outputs, name=self.unetModified)
```

Рисунок 22. *Реализация U-Net_modified архитектуры.*

2.2.4. Реализация архитектуры U-Net++L3

Для реализации архитектуры U-Net++L3 использовались 3 encoder_block и 3 decoder_block. В отличие от обычной U-Net для получения более точных промежуточных результатов добавились переменные s1_0, s2_0 и s1_1. В эти переменные записывались результаты decoder_block со следующего шага и для конкатенации в декодере передавался массив из полученных ранее на шаге промежуточных результатов. На рисунке 23 представлена реализация U-Net++L3 архитектуры.

```
def build_unet_plus_plus_modified(self, input_shape):
      tf.keras.backend.clear_session()

      inputs = Input(input_shape)
      inputs = Lambda(lambda x: x/255)(inputs)
      """ Encoder """
      s1, p1 = self.encoder_block(inputs, 64, True)
      s2, p2 = self.encoder_block(p1, 128, True)
      s3, p3 = self.encoder_block(p2, 256, True)
      b1 = self.conv_block_plus_plus(p3, 512)

      """ Inbetween """
      s1_0 = self.decoder_block(s2, [s1], 64, True)
      s2_0 = self.decoder_block(s3, [s2], 128, True)


      s1_1 = self.decoder_block(s2_0, [s1, s1_0], 64, True)

      """ Decoder """
      d1 = self.decoder_block(b1, [s3], 256, True)
      d2 = self.decoder_block(d1, [s2, s2_0], 128, True)
      d3 = self.decoder_block(d2, [s1, s1_0, s1_1], 64, True)

      """ Output """
      outputs = Conv2D(1, (1, 1), padding="same", kernel_initializer =
'he_normal', activation="sigmoid")(d3)
      return Model(inputs, outputs, name=self.unetPlusPlusModified)
```

Рисунок 23. *Реализация U-Net++L3 архитектуры.*

2.2.5. Реализация архитектуры U-net++L4

Для реализации стандартной архитектуры U-Net++L4 в отличие от U-Net++L3 использовались 4 encoder_block и 4 decoder_block. На рисунке 24 представлена реализация U-Net++L4 архитектуры.

```
def build_unet_plus_plus_modified(self, input_shape):
      tf.keras.backend.clear_session()

      inputs = Input(input_shape)
      inputs = Lambda(lambda x: x/255)(inputs)
      """ Encoder """
      s1, p1 = self.encoder_block(inputs, 32, True)
      s2, p2 = self.encoder_block(p1, 64, True)
      s3, p3 = self.encoder_block(p2, 128, True)
      s4, p4 = self.encoder_block(p3, 256, True)
      b1 = self.conv_block_plus_plus(p4, 512)

      """ Inbetween """
      s1_0 = self.decoder_block(s2, [s1], 32, True)
      s2_0 = self.decoder_block(s3, [s2], 64, True)
      s3_0 = self.decoder_block(s4, [s3], 128, True)

      s1_1 = self.decoder_block(s2_0, [s1, s1_0], 32, True)
      s2_1 = self.decoder_block(s3_0, [s2, s2_0], 64, True)
      s1_2 = self.decoder_block(s2_1, [s1, s1_0, s1_1], 32, True)
      """ Decoder """
      d1 = self.decoder_block(b1, [s4], 256, True)
      d2 = self.decoder_block(d1, [s3, s3_0], 128, True)
      d3 = self.decoder_block(d2, [s2, s2_0, s2_1], 64, True)
      d4 = self.decoder_block(d3, [s1, s1_0, s1_1, s1_2], 32, True)

      """ Output """
      outputs = Conv2D(1, (1, 1), padding="same", kernel_initializer =
'he_normal', activation="sigmoid")(d3)
      return Model(inputs, outputs, name=self.unetPlusPlusModified)
```

Рисунок 24. *Реализация U-Net++L4 архитектуры.*

2.3. Функции для работы с видео

Для сегментации объекта на видео использовалась функция maskVideo, на вход которой подавалось видео, которое нужно было обработать и модель, которую нужно использовать для сегментации, перед применением нужно иметь уже обученную модель. Все кадры, обработанные функцией, сохраняются в папку videoImg, каждое использование этой функции очищает содержимое папки с предыдущего использования. С помощью cv2.VideoCapture и cap.read считывается текущий кадр, после чего происходит его предобработка (изменение масштаба и обрезка краев). Полученное изображение подается в модель для сегментации, далее с помощью размытия Гаусса и фиксированного порога убираются лишние шумы и изменяется размер изображения на исходный. С помощью str(curIm).zfill(32) задается имя изображения, где curIm – номер кадра в видео, zfill

заполняет строку слева 0-ми, пока она не будет равна длине указанной в скобках. Функцию можно посмотреть в приложении А.

Конвертация изображений в видео происходит с помощью вызова функции convertToVid, где на вход подается название результирующего видео, которое будет сохранено в рабочую папку, указанную при инициализации класса. Сначала в один массив собираются пути до изображений, полученных во время работы предыдущей функции. Затем, с помощью функции video = cv2.VideoWriter() и video.write() изображения формируются в один видеоряд. Реализация функции представлена на рисунке 25.

```
 # конвертация изображений в видео
    def convertToVid(self, video_name = 'miceSeg.avi'):
      pathToVideoImages = self.pathToFolder+self.lastVideoImages
      if not os.path.exists(pathToVideoImages):
        print("Try run Mask Video first")
      else:

        images = [img for img in os.listdir(pathToVideoImages) if
img.endswith((".jpg", ".jpeg", ".png"))]
        print("Images:", images)

        # Set frame from the first image
        frame = cv2.imread(os.path.join(pathToVideoImages, images[0]))
        height, width, layers = frame.shape

        # Video writer to create .avi file
        pathToVideo = self.pathToFolder + video_name
        video = cv2.VideoWriter(pathToVideo,
cv2.VideoWriter_fourcc(*'DIVX'), self.fps, (width, height))

        # Appending images to video
        for image in images:
            video.write(cv2.imread(os.path.join(pathToVideoImages, image)))
        # Release the video file
        video.release()
        cv2.destroyAllWindows()
        print("Video generated successfully!")
```

Рисунок 25. *Функция для формирования видео из изображений.*

3. Тестирование и выбор лучшей из сегментационных моделей

Будут рассмотрены:

- результаты обучения,
- результаты сегментации,
- выбор лучшей модели для сегментации видео.

В качестве результатов обучения будут представлены результаты метрик и их сравнение. По результатам сегментации будет выбрана лучшая модель для сегментации видео. Будет приведен пример сегментации на изображении, представленном на рисунке 26.

Рисунок 26. *Изображение для сегментации.*

3.1. Результаты обучения

Результаты метрик, собранных во время обучения модели U-Net представлены в таблице 1.

Таблица 1. *Результаты метрик тестовой и валидационной выборок после обучения на U-Net.*

epoch	learning_rate	loss	mean_io_u	precision	recall	val_loss	val_mean_io_u	val_precision	val_recall
0	1.00E-04	-161.756	0.50122	0.15727	0.93592	2.989	0.4999	0.3021	0.2768
1	1.00E-04	-215.924	0.50118	0.41899	0.97399	-134.376	0.5027	0.7004	0.8638
2	1.00E-04	-239.651	0.50156	0.53821	0.98306	-201.896	0.5033	0.7711	0.9486
3	1.00E-04	-260.273	0.50159	0.60149	0.98568	-264.845	0.5026	0.6837	0.9727
4	1.00E-04	-281.227	0.50184	0.64916	0.98823	-280.885	0.5032	0.7618	0.9749
5	1.00E-04	-302.147	0.50196	0.67874	0.99040	-308.033	0.5034	0.7724	0.9719
6	1.00E-04	-323.050	0.50199	0.69510	0.99166	-325.307	0.5028	0.7778	0.9733
7	1.00E-04	-344.389	0.50205	0.70646	0.99310	-373.524	0.5024	0.7467	0.9769
8	1.00E-04	-366.196	0.50210	0.72051	0.99389	-417.529	0.5008	0.6795	0.9883
9	1.00E-04	-388.613	0.50222	0.73570	0.99478	-421.650	0.5002	0.6556	0.9882
10	1.00E-04	-411.500	0.50237	0.75391	0.99549	-420.508	0.5028	0.8007	0.9792
11	1.00E-04	-434.784	0.50247	0.75867	0.99582	-445.956	0.5020	0.7866	0.9748
12	1.00E-04	-458.962	0.50263	0.77622	0.99645	-488.104	0.5016	0.7789	0.9724
13	1.00E-04	-476.254	0.50116	0.69237	0.99208	-553.635	0.4939	0.4248	0.9908

Результаты метрик, собранных во время обучения модели U-Net_modified представлены в таблице 2.

Таблица 2. *Результаты метрик тестовой и валидационной выборок после обучения на U-Net_modified.*

epoch	learning_rate	loss	mean_io_u	precision	recall	val_loss	val_mean_io_u	val_precision	val_recall
0	1.00E-04	-124,2896	0,50186	0,21549	0,91346	-42,7508	0,49999	0,52839	0,70351
1	1.00E-04	-171,7194	0,50192	0,43842	0,97969	-121,9121	0,50120	0,66048	0,86442
2	1.00E-04	-191,5632	0,50224	0,55908	0,98366	-163,0349	0,50229	0,75837	0,94641
3	1.00E-04	-209,4066	0,50226	0,61143	0,98483	-206,5336	0,50118	0,62312	0,98089
4	1.00E-04	-227,8135	0,50226	0,65002	0,98751	-224,7719	0,50267	0,75949	0,97369
5	1.00E-04	-246,7217	0,50246	0,68528	0,98929	-257,8942	0,50307	0,77291	0,97653
6	1.00E-04	-265,5846	0,50253	0,70470	0,99063	-281,7148	0,50291	0,73484	0,97517
7	1.00E-04	-284,5923	0,50255	0,71350	0,99163	-288,8844	0,50234	0,71491	0,97864
8	1.00E-04	-302,5352	0,50214	0,69853	0,99021	-399,6726	0,37525	0,07404	0,99065
9	1.00E-04	-321,5272	0,50189	0,68698	0,99053	-376,4975	0,50140	0,64512	0,98902
10	1.00E-04	-343,7540	0,50241	0,72796	0,99391	-362,8318	0,50285	0,77794	0,98017
11	1.00E-04	-365,3047	0,50257	0,74708	0,99494	-397,6070	0,50247	0,77058	0,97037
12	1.00E-04	-386,9136	0,50273	0,75977	0,99574	-421,2573	0,50300	0,77224	0,97262
13	1.00E-04	-408,8769	0,50274	0,76753	0,99633	-440,8224	0,50245	0,77575	0,97471

Результаты метрик, собранных во время обучения модели U-Net++L3 на 5 эпохах представлены в таблице 3.

Таблица 3. *Результаты метрик тестовой и валидационной выборок после обучения на U-Net++L3.*

epoch	learning_rate	loss	mean_io_u	precision	recall	val_loss	val_mean_io_u	val_precision	val_recall
0	1.00E-04	-67,7703	0,50278	0,10310	0,95837	5,9693	0,49999	0,44572	0,14159
1	1.00E-04	-88,5164	0,50002	0,18656	0,99283	4,5429	0,49999	0,24182	0,13617
2	1.00E-04	-90,4698	0,49871	0,21907	0,99572	-22,0816	0,49999	0,52713	0,56461
3	1.00E-04	-91,1579	0,49797	0,25335	0,99715	-62,1069	0,49999	0,57415	0,82788
4	1.00E-04	-89,5654	0,49919	0,22374	0,99367	-78,3743	0,49998	0,09330	0,98643

Результаты метрик, собранных во время обучения модели U-Net++L4 на 10 эпохах представлены в таблице 4.

Таблица 4. *Результаты метрик тестовой и валидационной выборок после обучения на U-Net++L4.*

epoch	learning_rate	loss	mean_io_u	precision	recall	val_loss	val_mean_io_u	val_precision	val_recall
0	1.00E-04	-85,2765	0,50242	0,17466	0,91245	1,5288	0,49999	0,25042	0,29004
1	1.00E-04	-114,2972	0,50224	0,31066	0,97279	4,2751	0,49999	0,46732	0,14518
2	1.00E-04	-122,9709	0,50278	0,39020	0,98276	-9,7083	0,49999	0,79142	0,51184
3	1.00E-04	-129,6167	0,50364	0,49546	0,98541	-60,1894	0,49999	0,75788	0,78334
4	1.00E-04	-135,7413	0,50386	0,56864	0,98755	-102,3098	0,50182	0,70550	0,90963
5	1.00E-04	-141,3896	0,50398	0,61203	0,99005	-113,2647	0,50344	0,69042	0,95835
6	1.00E-04	-146,6775	0,50453	0,63647	0,99123	-127,7173	0,50243	0,75716	0,95495
7	1.00E-04	-151,7814	0,50433	0,64678	0,99256	-161,6665	0,50207	0,45538	0,99090
8	1.00E-04	-157,4634	0,50452	0,67582	0,99396	-166,6554	0,50266	0,71718	0,97306
9	1.00E-04	-162,9235	0,50482	0,69277	0,99488	-158,4954	0,50269	0,71849	0,97634

Для наглядного сравнения каждой из метрик по отдельности в программе Excel были построены соответствующие диаграммы.

По результатам метрики Mean IoU [13], значения для тестовых и валидационных выборок у каждой модели были близки к 0.5. Это означает, что отношение пересечения маски и оригинального изображения к их объединению примерно одинаковое. На рисунке 27 представлены изменения метрики Mean IoU для тренировочной и валидационных выборок во время обучения.

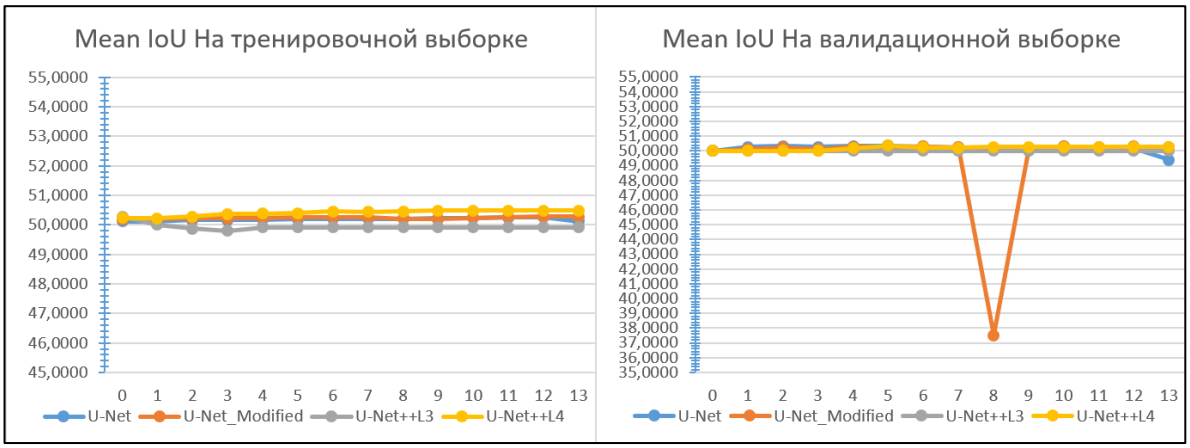

Рисунок 27. *Диаграмма с изменяющейся метрикой MeanIoU для тренировочной и валидационной выборок.*

Метрика «точность» [14] на тестовой выборке, почти для всех моделей к концу обучения приблизилась к 0.9. У U-Net++L3 метрика осталась на уровне 0,2. На валидационной выборке результаты похожие. Эта метрика показывает на процент пикселей модель посчитала принадлежащими мыши на картинке и которые оказались верным предсказанием. Хуже всего проявила себя U-Net++L3, а лучшие значения оказались у модели U-Net_modified. На рисунке 28 представлены изменения метрики Precision для тренировочной и валидационных выборок во время обучения.

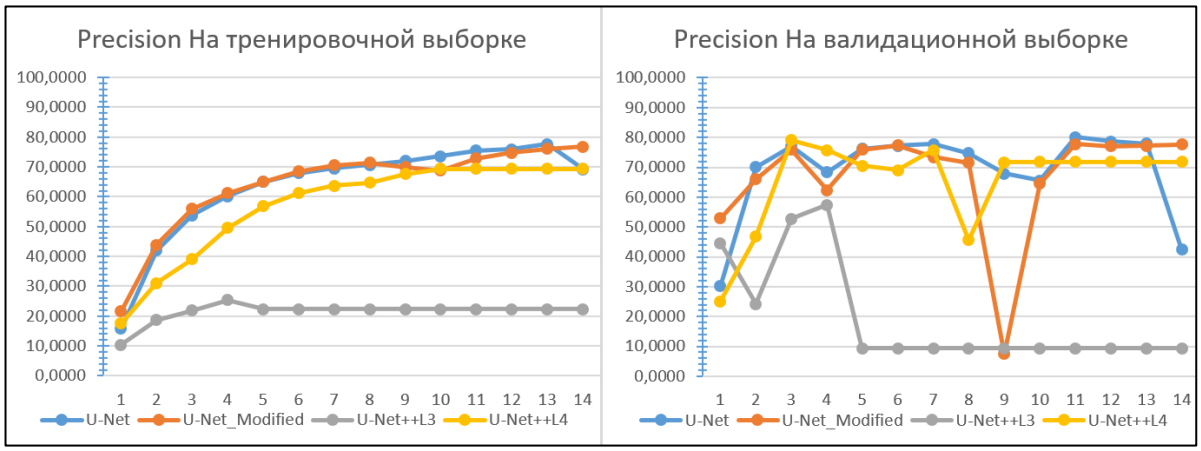

Рисунок 28. *Диаграмма с изменяющейся метрикой Precision для тренировочной и валидационной выборок.*

Метрика «полнота» [14] показывает насколько полно модель различает мышь на фоне. Для тестовой и валидационной выборок, метрика Recall к концу обучения доходила до 99%-100%. Лучшие значения оказались у модели на U-Net. На рисунке 29 представлены изменения метрики Recall для тренировочной и валидационных выборок во время обучения.

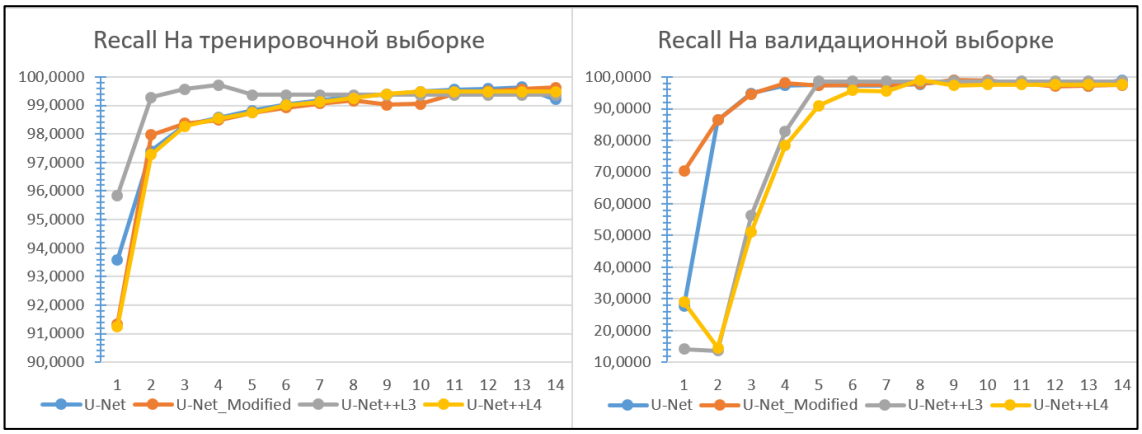

Рисунок 29. *Диаграмма с изменяющейся метрикой Recall для тренировочной и валидационной выборок.*

По метрике «переобучения» [15], на тестовой и валидационной выборках лучше всего показала себя модель U-Net++L3. К концу обучения у моделей U-Net и U-Net_modified начиналось заметное переобучение, когда у архитектур на основе U-Net++, значение не превышало -200. На рисунке 30 представлены изменения метрики Loss для тренировочной и валидационных выборок во время обучения.

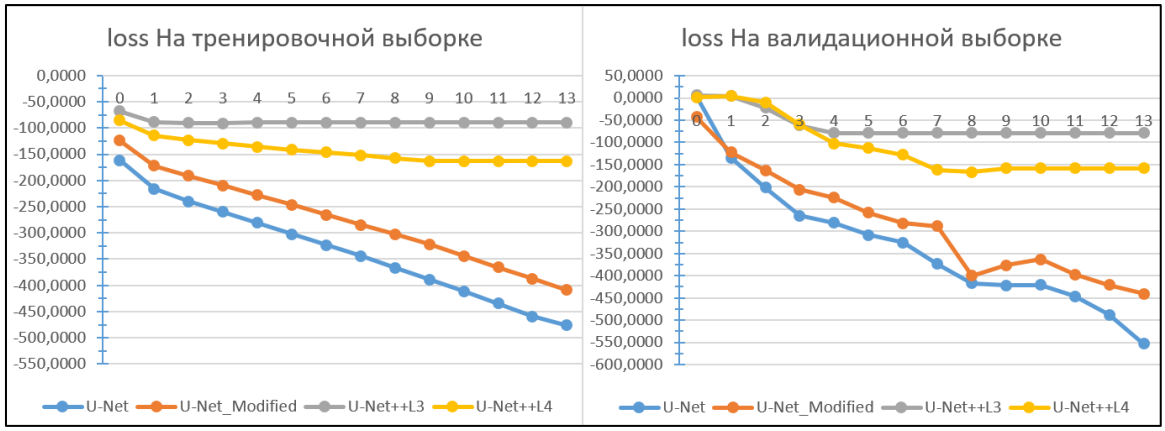

Рисунок 30. *Диаграмма с изменяющейся метрикой Loss для тренировочной и валидационной выборок.*

По полученным результатам, лучшей архитектурой для сегментации оказалась U-Net_modified и U-Net++L4.

3.2. Результаты сегментации

По результатам сегментации можно заметить, что модель не смогла точно распознать силуэт мыши на картинке. На полученной маске присутствует много шумов как на полу, так и на стенках. Результат сегментации на модели U-Net на 14 эпохах обучения представлен на рисунке 31.

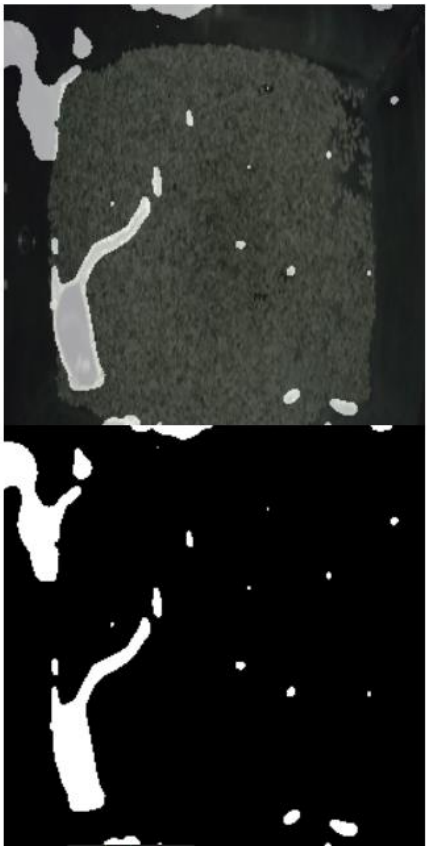

Рисунок 31. *Результат сегментации модели на архитектуре U-Net с 14 эпохами обучения.*

По результатам сегментации модели U-Net_modified шумы практически отсутствуют, как и более мелкие части мыши, например, хвост. Результаты сегментации на модели U-Net_modified Net на 14 эпохах обучения представлен на рисунке 32.

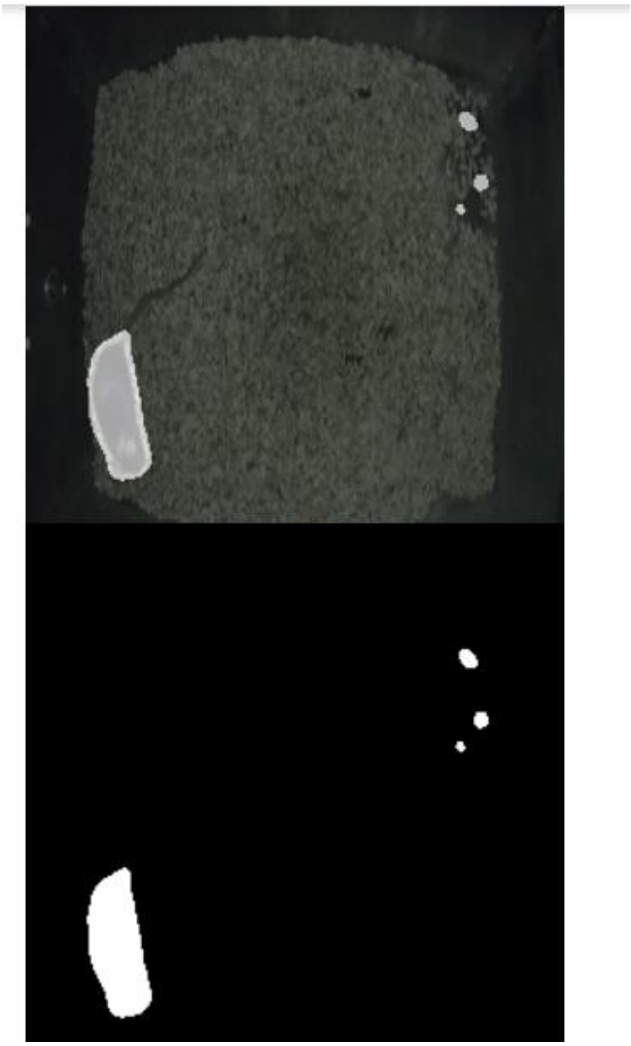

Рисунок 32. *Результат сегментации модели на архитектуре U-Net_modified с 14 эпохами обучения.*

Результаты сегментации U-Net и U-Net_modified, субъективно неудовлетворительные.

Далее рассмотрим сегментацию на модели U-Net++L3. После десяти эпох обучения, на полученной маске, при наличии минимальных шумов, четко видны части мыши включая ее хвост. Результат сегментации на модели U-Net++L3 на 10 эпохах обучения представлен на рисунке 33.

WARNING:tensorflow:5 out of the last 5 c
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step

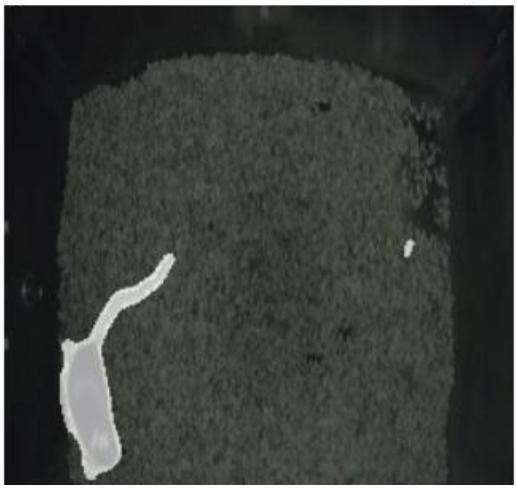

Рисунок 33. *Результат сегментации модели на архитектуре U-Net++L3 с 10 эпохами обучения.*

Так-как на 10 эпохах, у силуэта мыши появились странные выпуклости в районе хвоста и мордочки предположительно из-за переобучения, было решено сократить количество эпох обучения до 5, тем самым улучшив предыдущий результат. Сегментация на модели U-Net++L3 на 5 эпохах обучения представлена на рисунке 34.

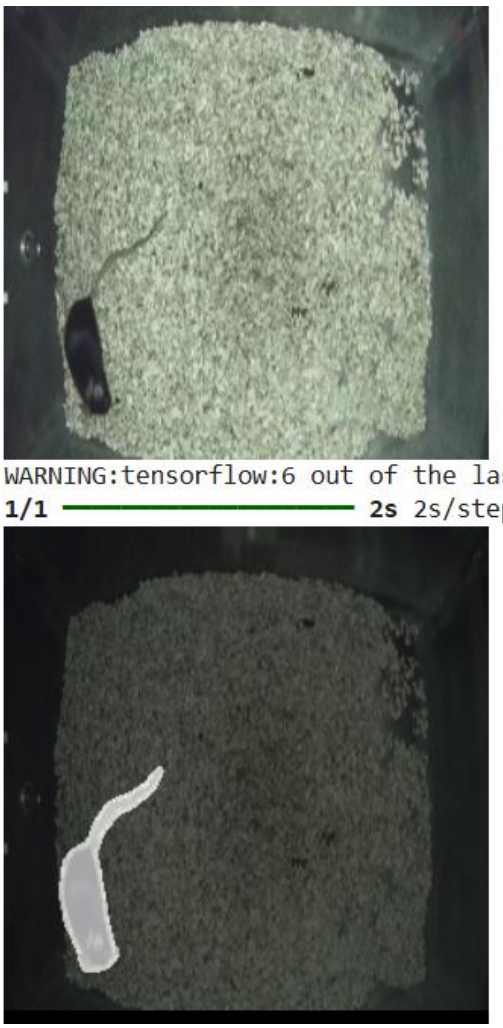

Рисунок 34. *Результат сегментации модели на архитектуре U-Net++L3 с 5 эпохами обучения.*

Последняя модель дала худшие результаты из рассмотренных выше. После перебора количества эпох, результат улучшить не получилось. Сегментация на модели U-Net++L4 на 10 эпохах обучения представлена на рисунке 35.

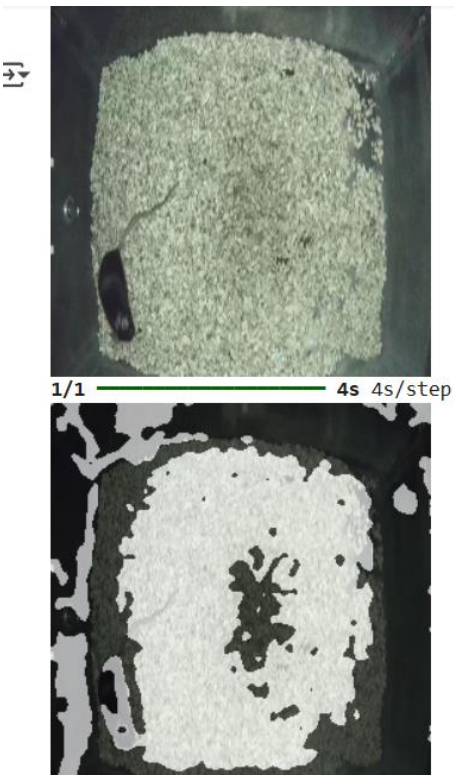

Рисунок 35. *Результат сегментации модели на архитектуре U-Net++L4 с 10 эпохами обучения.*

3.3. Выбор лучшей модели для сегментации видео

По результатам обучения и выборочной сегментации, лучше всего проявила себя модель с архитектурой U-Net++L3 на 5 эпохах обучения. Выбранная модель использовалась для сегментации видео длиной 9 минут. Время потраченное на сегментацию видео без использования графического процессора составило 8 часов, с графическим процессором 45 минут. Результат сегментации можно будет найти в папке Result перейдя по ссылке в приложении А.

ЗАКЛЮЧЕНИЕ

В результате выполнения научно исследовательской работы был изучен материал, необходимый для построения архитектуры модели. Также был разработан класс, для работы с сегментационными моделями и видео с целью определения каждого пикселя на принадлежность к объекту животного в видеопотоке. В заключении, полученные модели были протестированы и по результатам, лучшая модель использовалась для сегментации объекта на видео. Разработанная программа позволяет сегментировать контур животного за относительно короткое время.

Были реализованы задачи, а именно:

- подготовлена система к выполнению задания,
- написана программ для подготовки данных, обучения модели и инфраструктуры для использования модели,
- проведены циклы обучения, валидации и тестирования модели, оптимизация процесса обучения, параметров выборок, параметров систем использования модели,
- получены и оценок до возможного предела формальные и субъективные оценки модели.

Таким образом, цель и поставленные задачи были достигнуты.

Для дальнейшего развития необходимо:

- реализовать обход по сетке, для поиска лучших коэффициентов для обучения,
- добавить слой препроцесссинга внутрь модели,
- поисследовать лучше способы оптимизации сегментации.

ГЛОССАРИЙ

Workflow - это процесс, который включает последовательность действий.

Препроцессинг - это процесс предварительной обработки данных.

CNN – (Convolutional Neural Network) это тип алгоритма глубокого обучения, предназначенный для обработки визуальных данных.

Тензор - это представление данных, которое используется для решения задач сегментации изображений или временных рядов. Например, в задаче семантической сегментации изображений, элементы тензора отвечают о принадлежности каждого пикселя к определённому классу.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 В. А. Офицеров, А. С. Конушин. Нейросетевые методы сегментации изображений высокого разрешения [Научная статья] // cyberleninka.ru [Сайт]. URL: https://cyberleninka.ru/article/n/neyrosetevye-metody-segmentatsii-izobrazheniy-vysokogo-razresheniya/viewer (дата последнего обращения 10.06.2025).

2 Liang-Chieh Chen, George Papandreou, Florian Schroff, Hartwig Adam. Rethinking Atrous Convolution for Semantic Image Segmentation [Научная статья] // arxiv.org [Сайт]. URL: https://arxiv.org/pdf/1706.05587 (дата последнего обращения 10.06.2025).

3 Kaiming He, Georgia Gkioxari, Piotr Dollar, Ross Girshick. Mask R-CNN, 2018 [Научная статья] // arxiv.org [Сайт]. URL: https://arxiv.org/pdf/1703.06870 (дата последнего обращения 10.06.2025).

4 Abirami Vina. What is R-CNN? A quick overview [Электронный ресурс] // www.ultralitics.com [Сайт]. URL: https://www.ultralytics.com/ru/blog/what-is-r-cnn-a-quick-overview (дата последнего обращения 10.06.2025).

5 Vijay Badrinarayanan, Alex Kendall, SegNet: A Deep Convolutional EncoderDecoder Architecture for Image Segmentation, 2016 [Научная статья] // arxiv.org [Сайт]. URL: https://arxiv.org/pdf/1511.00561 (дата последнего обращения 10.06.2025).

6 Визуальные трансформеры (ViT) [Электронный ресурс] // habr.com [Сайт]. URL: https://habr.com/ru/companies/otus/articles/849756/ (дата последнего обращения 10.06.2025).

7 Olaf Ronneberger, Philipp Fischer, U-Net: Convolutional Networks for Biomedical Image Segmentation, 2015. [Научная статья] // arxiv.org [Сайт]. URL: https://arxiv.org/pdf/1505.04597 (дата последнего обращения 10.06.2025).

8 U-Net: нейросеть для сегментации изображений [Электронный ресурс] // neurohive.io [Сайт]. URL: https://neurohive.io/ru/vidy-nejrosetej/u-net-image-segmentation/ (дата последнего обращения 10.06.2025).

9 Unet++ Architecture Explained [Электронный ресурс] // www.geeksforgeeks.org [Сайт]. URL: https://www.geeksforgeeks.org/machine-learning/unet-architecture-explained/ (дата последнего обращения 10.06.2025).

10 Zongwei Zhou, Md Mahfuzur Rahman Siddiquee, Nima Tajbakhsh, and Jianming Liang. UNet++: A Nested U-Net Architecture for Medical Image Segmentation. [Научная статья] // arxiv.org [Сайт]. URL: https://arxiv.org/pdf/1807.10165v1 (дата последнего обращения 10.06.2025).

11 Open Source Computer Vision [Электронный ресурс] // docs.opencv.org [Сайт]. URL: https://docs.opencv.org/3.4/index.html (дата последнего обращения 10.06.2025) или https://opencv.org/ (дата последнего обращения 15.06.2025).

12 Google Colab вместо Jupyter Notebook: плюсы и особенности работы для новичков [Электронный ресурс] // habr.com [Сайт]. URL: https://habr.com/ru/companies/yandex_praktikum/articles/825754/ (дата последнего обращения 10.06.2025).

13 MeanIoU. [документация] // haibal.com [Сайт]. URL: https://haibal.com/documentation/metric-mean-iou/ (дата последнего обращения 10.06.2025).

14 Метрики классификации и регрессии. [Электронный ресурс] // education.yandex.ru [Сайт]. URL: https://education.yandex.ru/handbook/ml/article/metriki-klassifikacii-i-regressii дата последнего обращения 10.06.2025).

15 Shruti Jadon. A survey of loss functions for semantic segmentation, 2020 [Научная статья] // arxiv.org [Сайт]. URL: https://arxiv.org/pdf/2006.14822 (дата последнего обращения 10.06.2025).

Приложение А

ссылка на репозиторий с кодом программы и результатами сегментации

Репозиторий проекта можно найти, перейдя по ссылке:

https://github.com/Uranus28/miceSegmentation: НИР (github.com)