**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ**

**Кафедра технологий программирования**

# СЕМАНТИЧЕСКАЯ СЕГМЕНТАЦИЯ РЕЗУЛЬТАТОВ КОМПЬЮТЕРНОЙ ТОМОГРАФИИ С ИСПОЛЬЗОВАНИЕМ СВЕРТОЧНЫХ НЕЙРОННЫХ СЕТЕЙ

Курсовая работа

Черепенникова Романа Михайловича
студента 3 курса, 8 группы
специальность «прикладная математика»
Научный руководитель:
старший преподаватель кафедры ТП
Карпович Наталья Александровна

Минск, 2022

# РЕФЕРАТ

Курсовая работа содержит: 68 страниц, 13 иллюстраций (рисунков), 13 использованных литературных источников.

**Ключевые слова**: СЕГМЕНТАЦИЯ, КОМПЬЮТЕРНАЯ ТОМОГРАФИЯ, ГЛУБОКОЕ ОБУЧЕНИЕ, НЕЙРОННЫЕ СЕТИ

**Объектом исследования является** — подходы и методы сегментации изображений со снимков компьютерной томографии.

**Цель курсовой работы –** создание программного модуля, минимизирующего работу человека, и в то же время качественно решающего поставленные задачи сегментации на уровне, пригодном к применению на практике.

**Результатами являются –** разработан программный модуль, предоставляющий функционал для сегментации изображений со снимков компьютерной томографии

**Методы исследования –** теоретические: изучение литературы, посвященной алгоритмам сегментации и обработке трёхмерных моделей, а также изучение документации выбранного инструментария. Практические: применение изученных алгоритмов на реальных примерах томографий, проектирование программного модуля для сегментации.

**Область применения –** в настоящий момент полученный программный модуль находит применение в планировании хирургических операций. Создаваемые трёхмерные модели загружаются в очки дополненной реальности Microsoft Hololens, в которых врачи наносят предоперационную разметку.

Курсовая работа выполнена автором самостоятельно.

# РЭФЕРАТ

Курсавая праца змяшчае: 68 старонак, 13 ілюстрацый (малюнкаў), 13 выкарыстаных літаратурных крыніц.

**Ключавыя словы**: СЕГМЕНТАЦЫЯ, КАМП'ЮТАРНАЯ ТАМАГРАФІЯ, ГЛЫБОКАЕ НАВУЧАННЕ, НЕЙРОННЫЯ СЕТКІ.

**Аб'ект даследавання –** падыходы і метады сегментацыі здымкаў камп'ютарнай тамаграфіі.

**Мэта курсавой работы –** стварэнне праграмнага модуля, мінімізуючага працу чалавека, і ў той жа час якасна вырашаючага пастаўленыя задачы сегментацыі на ўзроўні, прыдатным да ўжывання на практыцы.

**Вынікамі з'яўляюцца** - — распрацаваны праграмны модуль, які прадстаўляе функцыянал для сегментацыі здымкаў камп'ютарнай тамаграфіі.

**Метады даследавання** – тэарэтычныя: вывучэнне літаратуры, прысвечанай алгарытмах сегментацыі і апрацоўцы трохмерных мадэляў, а таксама вывучэнне дакументацыі абранага інструментара. Практычныя: прымяненне вывучаных алгарытмаў на рэальных прыкладах тамаграфій, праектаванне праграмнага модуля для сегментацыі

**Вобласць прымянення** — ў сапраўдны момант атрыманы праграмны модуль знаходзіць прымяненне ў планаванні хірургічных аперацый. Ствараемыя трохмерныя мадэлі загружаюцца ў акуляры дапоўненай рэальнасці Microsoft Hololens, у якіх лекары наносяць перадаперацыйную разметку.

Курсавая праца выканана аўтарам самастойна.

# ESSAY

Coursework contains: 68 pages, 13 illustrations, 13 used literary sources.

**Keywords:** SEGMENTATION, COMPUTED TOMOGRAPHY, DEEP LEARNING, NEURAL NETWORKS

**The object of study** — the approaches and methods for segmenting images from computed tomography images.

**The purpose of the work** — create a software module that minimizes human work, and at the same time, qualitatively solves the problems of segmentation at a level suitable for practical use.

**As a result** — the software module that provides functionality for segmenting computed tomography images.

**The research methods** — Theoretical: studying the literature on segmentation algorithms and processing three-dimensional models, as well as studying the documentation of the selected tools. Practical: the application of the studied algorithms on real tomography examples, designing a software module for segmentation and the subsequent reconstruction of three-dimensional models.

**Scope** — Currently, the resulting software module is used in planning surgical operations. The created three-dimensional models are loaded into Microsoft Hololens augmented reality glasses, in which doctors apply preoperative markup.

The course work was carried out by the author independently.

# СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

Планирование оперативного вмешательства – важный и неотъемлемый этап хирургии. Данный процесс определяет особенности клинического случая, позволяет выявить возможные проблемы во время проведения операции. Планирование включает в себя стандартное обследование пациента, изучение данных лабораторных исследований, а также методы визуализации.

Одним из наиболее популярных методов исследования – компьютерная томография. В отличие от одиночного рентгеновского снимка, компьютерная томография, позволяет сделать множество снимков, являющихся срезами участка тела, отстоящих друг от друга на фиксированных интервалах. Имея множество срезов объемного тела, можно воссоздать на их основе трехмерную модель, точность которой будет зависеть лишь от интервалов между снимками. Такую модель хирург может использовать во время планирования операции для составления более точной картины проблемы.

На сегодняшний день большинство оперативных вмешательств проводится без использования современного технологического потенциала. Например, перед оперированием хирург может составлять план и наносить маркером разметку на часть тела пациента, основываясь на прощупывании и лежащем рядом рентгеновском снимке. В данный момент уже существуют программные модули, позволяющие создавать трехмерные модели на основе снимков компьютерной томографии, однако зачастую они требуют дополнительной ручной работы врача по выделению на срезах структур, представляющих интерес для исследования.

В данной работе будут рассмотрены способы автоматической семантической сегментации изображений, основанные на применении сверточных нейронных сетей. В практической части будет решена задачи сегментации печени и желчного пузыря на снимках, полученных при помощи компьютерной томографии.

# ГЛАВА 1. ВВЕДЕНИЕ В ПРЕДМЕТНУЮ ОБЛАСТЬ. ЗАДАЧА СЕМАНТИЧЕСКОЙ СЕГМЕНТАЦИИ

## 1.1 Компьютерная томография

*Компьютерная томография* – метод неразрушающего послойного исследования внутреннего строения предмета, который был предложен в 1972 году Годфри Хаунсфилдом и Алланом Кормаком. Метод основан на измерении и компьютерной обработке ослабления рентгеновского излучения различными по плотностям тканями. В настоящее время рентгеновская компьютерная томография является одним из основных методов исследования внутренних органов человека с использованием рентгеновского излучения. [1]

### 1.1.1 Шкала Хаунсфилда

Для визуальной и количественной оценки плотности визуализируемых методом компьютерной томографии структур используется шкала ослабления рентгеновского излучения, получившая название *шкалы Хаунсфилда* (её визуальным отражением на мониторе аппарата является черно-белый спектр изображения). Диапазон единиц шкалы («*денситометрических показателей*, англ. *Hounsfield units*»), соответствующих степени ослабления рентгеновского излучения анатомическими структурами организма, составляет от −1024 до +3071, то есть 4096 чисел ослабления. Показатель 0 в шкале Хаунсфилда (0 HU) соответствует плотности воды, отрицательные величины шкалы соответствуют воздуху и жировой ткани, положительные — мягким тканям, костной ткани и более плотным веществам (металл). В практическом применении измеренные показатели ослабления могут несколько отличаться на разных аппаратах.

**Таблица 1.1 Средние денситометрические показатели**

| Вещество | HU |
|---|---|
| Воздух | -1000 |
| Жир | -120 |
| Вода | 0 |
| Мягкие ткани | +40 |
| Кости | +400 и выше |

### 1.1.2 Процесс обследования

При компьютерной томографии пациент ложится на кушетку, которая затем заезжает в томограф, в котором находится кольцеобразный контур, являющийся источником рентгеновского излучения (рис 1.1). Пропуская это излучение через человека и фиксируя его ослабление, можно делать выводы о том, какие ткани являются более или менее «плотными» для рентгеновских волн.

**Рисунок 1.1 Томограф**

Результатом работы томографа является последовательность (серия) снимков (рис. 1.2), сделанных параллельно друг другу, но в отстоящих друг от друга плоскостях.

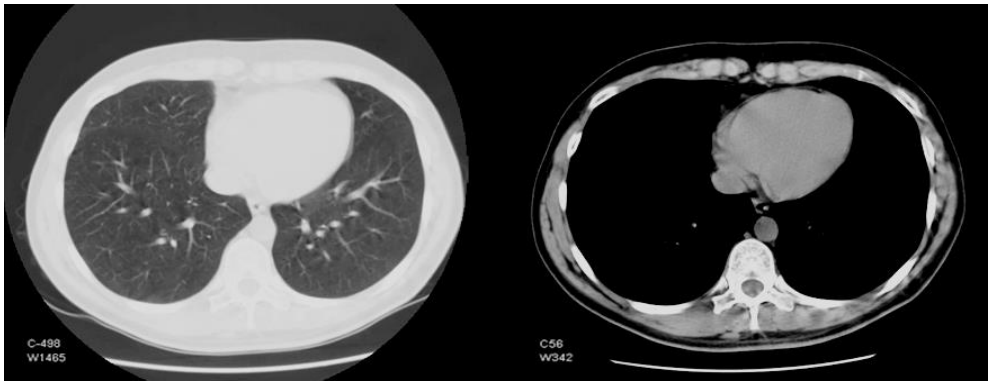


**Рисунок 1.2 Пример снимков, полученных с помощью компьютерной томографии**

## 1.2 Семантическая сегментация

В контексте машинного обучения задача сегментации объектов на изображении называется *семантической сегментацией.* Она заключается в классификации каждого пикселя входного изображения. С помощью такой классификации можно принять решение, расположен ли некоторый пиксель на области, где изображена, например, печень.

В результате классификации всех пикселей входного изображения можно получить бинарную маску, повторяющую форму искомого объекта на снимке (рис. 1.3).

Семантическая сегментация — одна из фундаментальных задач компьютерного зрения, поскольку к ней сводятся другие важные задачи. Если известны маски всех объектов на изображении, то становятся тривиальными задачи, например, детектирования объектов определённой категории. С другой стороны, получение семантической сегментации в явном виде требуется в прикладных задачах, таких как автономная навигация автомобилей, оценка позы человека или восстановление трехмерной структуры сцены.

**Рисунок 1.3 Пример семантической сегментации**

Существует два главных критерия оценки качества алгоритмов семантической сегментации. Вычислительная сложность и точность алгоритма. Так как в работе рассматривается задача, связанная с медициной, производительность не играет ключевой роли, будем рассматривать метрики, связанные с точностью полученных результатов.

Для описания метрик необходимо ввести важную концепцию для описания этих метрик в терминах ошибок классификации – *матрицу ошибок* (англ. *confusion matrix*).

Пусть алгоритму необходимо предсказывать относится ли объект к определенному классу. Если объект принадлежит к этому классу, то будем

считать, что правильный ответ на этом объекте равен 1, а если не принадлежит – 0. Тогда матрица ошибок классификации будет иметь следующий вид.

| | $y = 1$ | $y = 0$ |
|---|---|---|
| $\bar{y} = 1$ | True Positive (TP) | False Positive (FP) |
| $\bar{y} = 0$ | False Negative (FN) | True Negative (TN) |

Здесь $\bar{y}$ – ответ алгоритма на объекте, а $y$ – правильный ответ на данном объекте. Таким образом ошибки бывают двух типов: *ложноотрицательный* (FN) результат и *ложноположительный* результат (FP).

- *Accuracy (аккуратность)*

  Самой простой метрикой и интуитивно понятной метрикой является является accuracy(аккуратность), которая вычисляется как доля верных ответов алгоритма:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

  Здесь и далее через TP, TN, FP, FN обозначается количество истинноположительных, истинноотрицательных, ложноположительных и ложноотрицательных ответов алгоритма соответственно.

  Существенным недостатком данной метрики является, невозможность ее применения в задачах с несбалансированными классами. Рассмотрим на примере почему это так.

  Пусть имеется картинка из 110 пикселей, на 100 пикселях нет объектов интересующего нас класса, из которых алгоритм определил верно 90 (TN = 90, FP = 10), и 10 на которых необходимый класс присутствует, 5 из которых определены верно (TP = 5, FN = 5), тогда accuracy:

$$accuracy = \frac{5 + 90}{5 + 90 + 10 + 5} = 86.4$$

  Однако, если предсказывать, что на картинке нет объектов интересующего класса, то получим более высокий accuracy:

$$accuracy = \frac{0 + 100}{0 + 100 + 10 + 0} = 90.9$$

При этом алгоритм не будет обладать никакой предсказательной силой, так как изначально нашей целью было определять какие пиксели относятся к интересующему классу.

Преодолеть это поможет переход с общей для всех классов метрики к отдельным показателям качества классов.

- *Precision (точность)*

  Точностью называется доля объектов действительно принадлежащих к данному классу относительно всех объектов относительно всех объектов, которые система отнесла к этому классу.

$$precision = \frac{TP}{TP + FP}$$

  Эта метрика не позволяет относить все объекты к одному классу, так в этом случае получим рост FP, а следовательно и ухудшение значения метрики.

- *Recall (полнота)*

  Полнотой определяется как доля истинно положительных классификаций. Полнота показывает, какую долю объектов реально относящихся к положительному классу предсказана верно и способность алгоритма вообще обнаруживать данный класс.

$$recall = \frac{TP}{TP + FN}$$

- *F-score (F-мера)*

  Precision и recall не зависят, в отличие от accuracy, от соотношения классов и поэтому применимы в условиях несбалансированных классов. Чем выше и полнота и точность, тем лучше алгоритм. В реальной практике максимальная точность и полнота недостижимы, поэтому стоит задача найти оптимальный(в зависимости от задачи) баланс между этими двумя метриками. F-мера как раз и является такой метрикой. Она представляет собой гармоническое среднее precision и recall, и стремится к нулю, если precision или recall стремятся к нулю.

$$F = \frac{2precision \times recall}{precision + recall}$$

Данная формула придает одинаковый вес точности и полноте. Однако ее можно обобщить так, чтобы придавать больший вес точности или полноте в зависимости от задачи:

$$F_\beta = \frac{(1 + \beta^2)precision \times recall}{\beta^2 precision + recall}$$

Добавление параметра $\beta$ позволяет придать различный вес точности и полноте, в зависимости от того, что важнее в нашей задаче. При $0 < \beta < 1$ приоритет отдается точности, а при $\beta > 1$ – полноте.

- *Intersection over Union*

Одной из наиболее используемых метрик в задачах сегментации является Intersection over Union, которая определяется как отношение площади пересечения полученной и правильной масок к площади их объединения (рис.1.4).

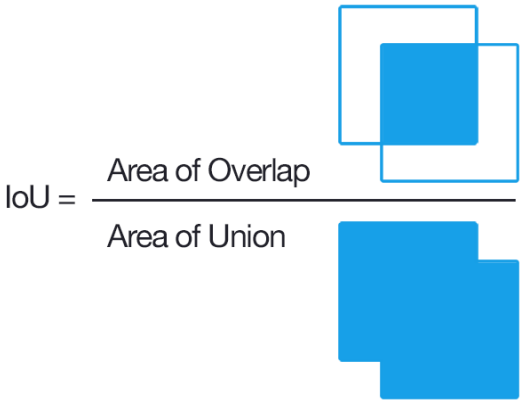


**Рисунок 1.4 Графическая иллюстрация метрики IoU**

В терминах матрицы ошибок классификации данная метрика может быть вычислена следующим образом:

$$IoU = \frac{TP}{TP + FP + FN}$$

Мы рассмотрели метрики для случая бинарной семантической сегментации. Теперь перейдем к изучению способов оценки качества в много классовом случае.

В многоклассовых задачах, как правило, стараются свести подсчет качества к одной из рассмотренных выше метрик для бинарного случая. Выделяют два подхода к такому сведению: микро- и макро-усреднение.

Пусть выборка состоит из $K$ классов. Рассмотрим $K$ бинарных задач, каждая из которых заключается в отделении своего класса от остальных, то есть

целевые значения для *k-й* задачи вычисляются как $y_i^k = [y_i = k]$ . Для каждой из них можно вычислить значения $TP, FP, TN, FN$ алгоритма. При микро-усреднении сначала эти характеристики суммируются по всем классам, а затем вычисляется итоговая бинарная метрика – например, точность, полнота или F-мера. При макро-усреднении сначала вычисляется итоговая метрика для каждого класса, а затем результаты усредняются по всем классам.

Если классы отличаются по мощности, то при микро-усреднении некоторые (маленькие) классы практически не будут влиять на итоговый результат, поскольку их вклад в итоговые значения $TP, FP, TN, FN$ будет незначительным. В случае же с макро-вариантом усреднение проводится для величин, которые уже не чувствительны к соотношению размеров классов, и поэтому каждый класс внесет равный вклад в итоговую метрику [2].

## ВЫВОДЫ

В данной главе была кратко рассмотрена предметная область, а также рассмотрена задача семантической сегментации и способы оценки полученных при решении задачи результатов.

# ГЛАВА 2. АРХИТЕКТУРЫ СВЕРТОЧНЫХ НЕЙРОННЫХ СЕТЕЙ ДЛЯ ЗАДАЧИ СЕМАНТИЧЕСКОЙ СЕГМЕНТАЦИИ

Использование нейронных сетей для решения задач компьютерного зрения, не является новой идеей, например, еще в 1989 году Я. Лекун в своей работе “Backpropagation Applied to Handwritten Zip Code Recognition” [3] использовал сверточные нейронные сети для распознавания, написанных от руки, почтовых индексов. Однако наибольшей популярности нейронные сети достигли относительно недавно, это стало возможно благодаря росту вычислительных мощностей, что позволило обучать глубокие нейросети, которые в некоторых задачах могут превосходить по точности даже человека.

В данной главе будут рассмотрены нейронные сети прямого распространения, в качестве общего примера работы нейронных сетей, сверточные нейронные сети, в том числе архитектуры предназначенные для задач семантической сегментации.

## 2.1 Нейронные сети прямого распространения

*Нейронные сети прямого распространения*, которые также называют *многослойными перцептронами* – самые типичные примеры моделей глубокого обучения. Цель сети прямого распространения – аппроксимировать некоторую неизвестную функцию $f^*(x)$. Например, в случае задачи классификации, $y = f^*(x)$ отображает входные данные $x$ в категорию $y$. Сеть прямого распространения определяет отображение $y = f(x, w)$ и путем обучения находит значения параметров w, дающих наилучшую аппроксимацию $f^*$ .

Слова прямое распространение означают, что распространение информации начинается с $x$, проходит через промежуточные вычисления, необходимые для определения $f$ , и заканчивается выходом $y$. Не существует обратных связей, по которым выходы модели подаются ей на вход. Обобщенные нейронные сети, включающие такие обратные связи, называются *рекуррентными* и в данной работе не рассматриваются.

В ходе обучения нейронной сети необходимо приблизить $f(x, w)$ к $f^*(x)$. Обучающие данные – это зашумленные приближенные примеры $f^*(x)$. Каждый пример $x$ сопровождается меткой y $f^*(x)$. Обучающие примеры напрямую указывают, что в выходном слое должно соответствовать, тому что поступило на входной слой. Поведение остальных слоев напрямую обучающими данными не определяется. Алгоритм обучения должен решить, как использовать эти слои для

порождения желаемого выхода, но обучающие данные ничего не говорят о том, что должен делать каждый слой. [4, 150].

Атомарным элементом вычислений в нейронных сетях является *нейрон*. Структурно он состоит из следующих частей:

- Несколько входов, принимающих входные сигналы $x_i$
- Вычислительный центр, агрегирующий входные сигналы с использованием весов $w_i$, которые являются изменяющимися параметрами, именно они и настраиваются на этапе обучения нейронной сети
- Несколько выходов, которые распространяют полученный сигнал $y$

В качестве функции-агрегатора используют линейную функцию:

$$w_0 + \sum_i^n w_i x_i$$

Источником входных сигналов выступают как изучаемые данные, так и выходы других нейронов. У одного нейрона, которому на вход поступает $n$ сигналов имеется $(n\ +\ 1)$ весов $w_i$.

Затем нейроны объединяются в *слои.* Один слой нейронной сети состоит из нескольких несоединенных между собой нейронов, однако, нейроны одного слоя разделяют входные сигналы и приемников их выходных сигналов (рис. 2.1).

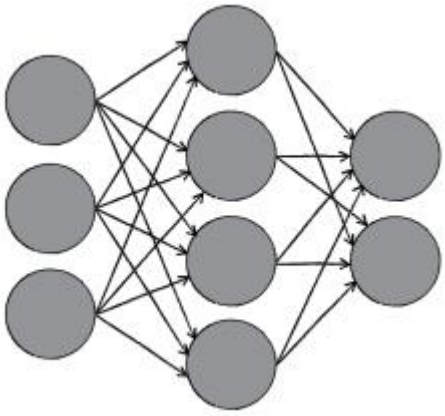

**Рисунок 2.1 Схема простейшей нейронной сети**

В нейронных сетях выделяют следующие типы слоев. *Входной* (первый) слой используется исключительно как распространитель сигналов к следующему слою и не производит никаких вычислительных операций. *Выходной* (последний) слой определяет размерность функции. Все остальные слои принято называть *скрытыми.* Таким образом в слое из m нейронов и выходными данными размерности $n\ +\ 1$ всего $m(n\ +\ 1)$ обучаемых параметров (весов).

Соединяя несколько слоев с одинаковым или различным числом нейронов получаем нейронную сеть. Она, как результат композиции функций, тоже является функцией, которая на вход принимает данные размерности входного слоя, а результатом этой функции также является вектор, только уже размерности выходного слоя.

У такой конфигурации нейронной сети есть существенный недостаток. Так как результирующая функция есть композиция линейных функций, то и она сама является линейной функции. Таким образом, объединение нескольких слоев не дает никаких результатов, так как один слой мог бы выполнять такое же преобразование, как и построенная нейронная сеть.

Далеко не все процессы можно описать линейной функцией, которой в текущей конфигурации является наша нейронная сеть. Решением данной проблемы являются *функции активации.* Суть работы функций активации заключается в следующем: добавить нелинейных преобразований в нейронную сеть, что и позволит нам аппроксимировать в том числе и нелинейные функции. Также функция активации позволяет ограничить диапазон выходных значений нейрона (например, можно ограничить выходные значения диапазоном [0; 1] и в задачах классификации интерпретировать полученный результат как вероятность принадлежности входных данных к некоторому классу). После добавления функции активации выход нейрона будет выглядеть следующим образом:

$$y = \phi(w_0 + \sum_i^n w_i x_i)$$

Где $\phi(x)$ – некоторая функция активации. Вообще говоря, в качестве функции активации может выступать любая нелинейная функция. Однако на практике, в большинстве случаев, используют один из следующих вариантов:

- ReLU (англ. Rectified Linear unit)
$$\phi(x) = \max\{0, x\}$$
- Сигмоидная функция
$$\phi(x) = \frac{1}{1 + \exp(-x)}$$
- Гиперболический тангенс
$$\phi(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$
- Арктангенс
$$\phi(x) = arctg(x)$$
- Leaky ReLU
$$\phi(x, \alpha) = \max\{\alpha x, x\}$$

- SoftPlus

$$\phi(x) = \log(1 + \exp(\mathrm{x}))$$

На рисунке 2.2 представлены графики функций активации в окрестности точки 0. Отметим, что эти функции что эти функции не имеют обучаемых параметров, а также всюду дифференцируемы, за исключением ReLU и Leaky ReLU (их производные обычно просто доопределяют в точке 0).

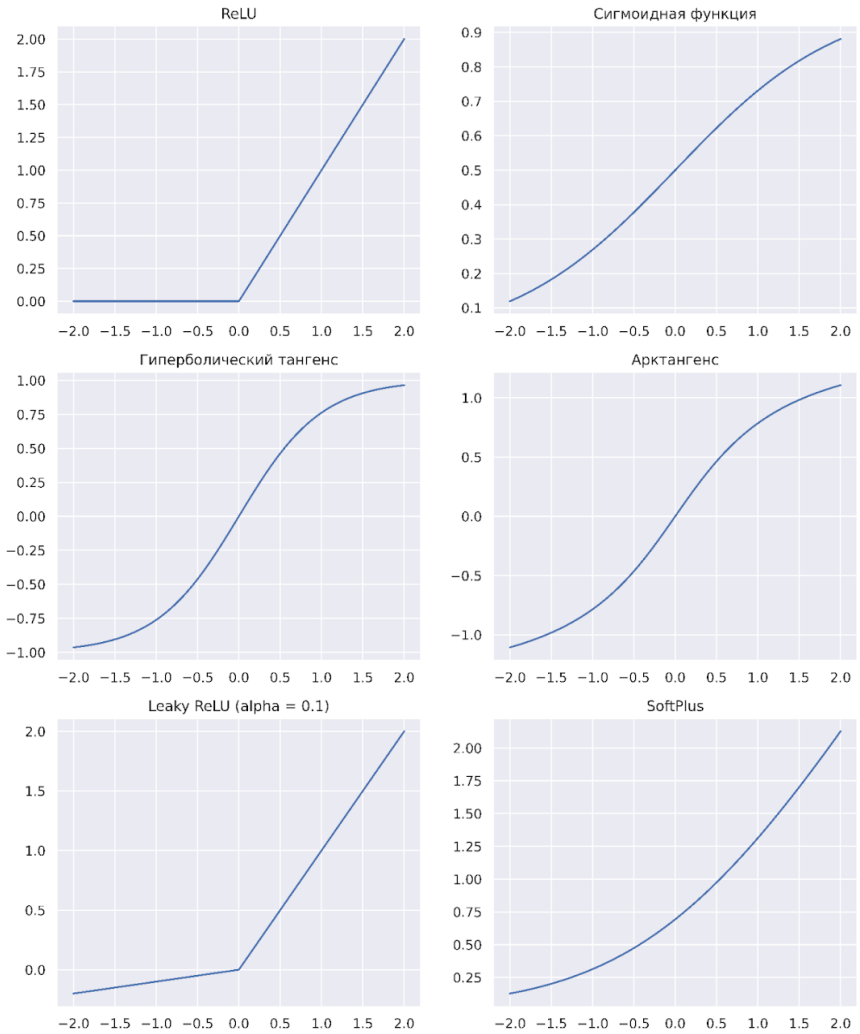


**Рисунок 2.2 Функции активации**

## 2.2 Сверточные нейронные сети

Нейронные сети прямого распространения являются полносвязными. Такие сети имеют один заметный недостаток: при большом размере входных данных либо получится большое количество параметров, либо сеть будет недостаточно хорошо приближать целевую функцию, если в ней существуют сложные зависимости. Такая проблема автоматически исключила этот тип сетей из использования в анализе изображений, так как там входные данные всегда имеют солидный размер. Для решения этой проблемы существует другой тип нейронных сетей, называемых *сверточными*.

Свое название сверточные сети получили из-за наличия сверточных слоев, работающих по следующему принципу:

- Входные данные представляются в виде матрицы
- Методом скользящего окна матрица преобразуется в новую матрицу меньшего размера, используя, так называемые, *ядра свертки.*
- Уменьшенная выходная матрица проходит через функцию активации и подается дальше.

Кроме сверточных слоев в сверточных нейронных сетях используются следующие типы слоев:

- *Слой пуллинга (pooling layer)*

  Этот слой принимает на вход матрицу или вектор и по некоторому принципу удаляет определенную долю значений. Слой пуллинга, как и сверточный слой, направлен на уменьшение размерности входных данных, что позволяет сократить дальнейшее число вычислений. Помимо этого, данный слой позволяет сети быть более устойчивой по отношению к небольшим сдвигам объектов на изображении. Сама операции пуллинга может быть совершенно разной, можно выбирать максимальное значение (*max pooling*), минимальное значение (*min pooling*) или, например, вычислять среднее значение (*average pooling*). [5]

- *Слой пакетной нормализации (batch-normalization layer)*

  Суть данного слоя заключается в том, чтобы передать следующему слою данные имеющие нулевую математическое ожидание и единичную дисперсию. Данный слой позволяет повысить производительность и стабилизировать работу искусственных нейронных сетей за счет борьбы с внутренним ковариантным сдвигом (явлением, в результате которого происходит изменение среднего значения и дисперсии входных данных во внутренних слоях во время обучения) [6]

- *Dropout слой*

Основным назначением данного слоя, является борьба с переобучением нейронной сети. Главная идея заключается в обучении нескольких нейронных сетей вместо одной, а затем усреднить результаты. Причем сеть для обучения получается исключением из исходной сети некоторых нейронов в сети с вероятностью $p$. Исключенные нейроны не вносят свой вклад в процесс обучения, поэтому исключение хотя бы одного нейрона равносильно обучению новой нейронной сети. По словам авторов, [7] “В стандартной нейронной сети производная, полученная каждым параметром, сообщает ему, как он должен измениться, чтобы, учитывая деятельность остальных блоков, минимизировать функцию конечных потерь. Поэтому блоки могут меняться, исправляя при этом ошибки других блоков. Это может привести к чрезмерной совместной адаптации (co-adaptation), что, в свою очередь, приводит к переобучению, поскольку эти совместные адаптации невозможно обобщить на данные, не участвовавшие в обучении. Мы выдвигаем гипотезу, что Dropout предотвращает совместную адаптацию для каждого скрытого блока, делая присутствие других скрытых блоков ненадежным. Поэтому скрытый блок не может полагаться на другие блоки в исправлении собственных ошибок.”

## 2.3 Архитектуры нейронных сетей для семантической сегментации

С момента появления сверточных нейронных сетей возник интерес к использованию признаков, которые они умеют выделять из изображения, для решения задачи семантической сегментации. Первые опубликованные подходы были попытками преобразовать сети, предназначенные для классификации, такие как VGG и Alex-Net путем изменения их полносвязных слоев. Основной проблемой такого подхода к решению задачи является быстрое переобучение подобных сетей и чрезмерная сложность обучения полносвязной части, за счет большой размерности выходного слоя. Таким образом, обучение подобных систем, в силу вычислительной сложности, было нецелесообразно. Кроме того, используемые нейросети не были достаточно глубокими, что не позволяло им выделять признаки, способные описать семантику изображения.

В работе Fully Convolutional Networks for Semantic Segmentation [8] для решения описанных проблем, в качестве альтернативы нейросетям состоящим из сверточных и полносвязных частей, была предложена архитектура,

получившая название “*Полностью сверточные нейронные сети*”(англ. – *Fully convolutional networks*) (рис. 2.3).

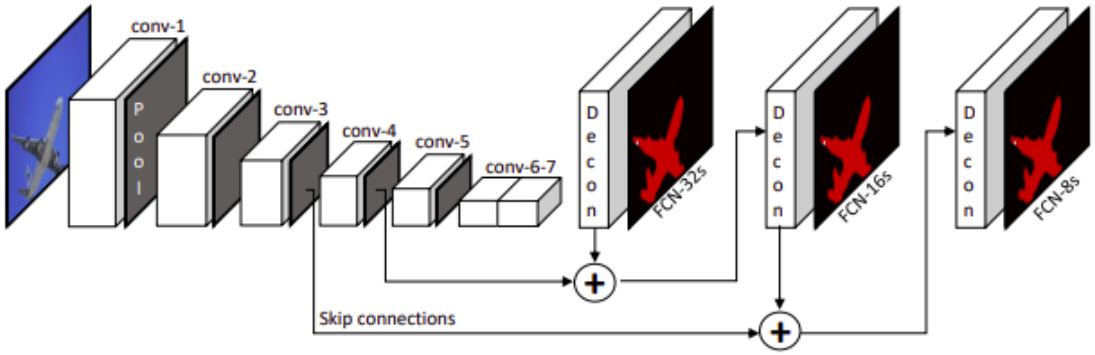


**Рисунок 2.3 Полностью сверточная нейронная сеть**

Архитектура полностью сверточных нейронных сетей считалась революционной во многих аспектах. Прежде всего, за счет отказа от полносвязных слоев, процесс обучения стал занимать существенно меньше времени. Это объясняется тем, что, по сравнению с полносвязным слоями, сверточные имеют существенно меньшее количество обучаемых параметров. Вторым важным новшеством стало то, что такая архитектура не накладывает ограничений на размер входного изображения. Третьим, и возможно самым главным, стало использование так называемых *skip-соединений.* Подобные соединения позволяют использовать при сегментации информацию, которая в противном случае могла бы быть потеряна из-за слоев пулинга. Операции пулинга помогают сети создать иерархию признаков, но в то же время приводят к потери локальной информации, которая представляет ценность для семантической сегментации, особенно важную роль такая информация играет при определении границ объектов. Skip-соединения сохраняют эту информацию и передают ее более глубоким слоям, позволяя объединить локальные признаки ранних слоев с иерархическими признаками глубоких слоев.

В 2015 году для решения задач сегментации биомедицинских изображений была предложена нейронная сеть, получившая название Unet[9]. Архитектура сети представляет собой полностью сверточную нейронную сеть, модифицированную таким образом, чтобы она могла работать с меньшим количеством примеров (обучающих данных) и делала более точную сегментацию. Отсутствие больших объемов данных является частой практикой для задач, связанных с медициной. Это объясняется тем, что для сбора таких наборов необходимо привлечение квалифицированных специалистов.

Сеть содержит сжимающий путь (слева) и расширяющий путь (справа), поэтому архитектура похожа на букву U (рис. 2.4), что и отражено в названии.

Сжимающий путь является типичной сверточной сетью, он содержит два подряд идущих сверточных слоя с размером ядра$3 \times 3$, после которых применяется функция активации ReLU и операция максимального пулинга с ядром $2 \times 2$ и шагом 2.

Каждый шаг расширяющего пути содержит слой обратный пуллингу, используемый для увеличения размеров карты признаков, после которого следует свертка $2 \times 2$, которая уменьшает количество каналов (карт признаков). После идет конкатенация с обрезанной картой признаков соответствующего размера из сжимающего пути сети и две свертки $3 \times 3$ после каждой из которых применяется функция активации ReLU. Обрезка карт признаков из сжимающего пути необходима так как при каждой операции свертки мы теряем граничные пиксели изображения. На последнем слое используются свертка размера 11 для приведения каждого 64-компонентного вектора признаков до требуемого количества классов. Всего нейросеть имеет 23 сверточных слоя.

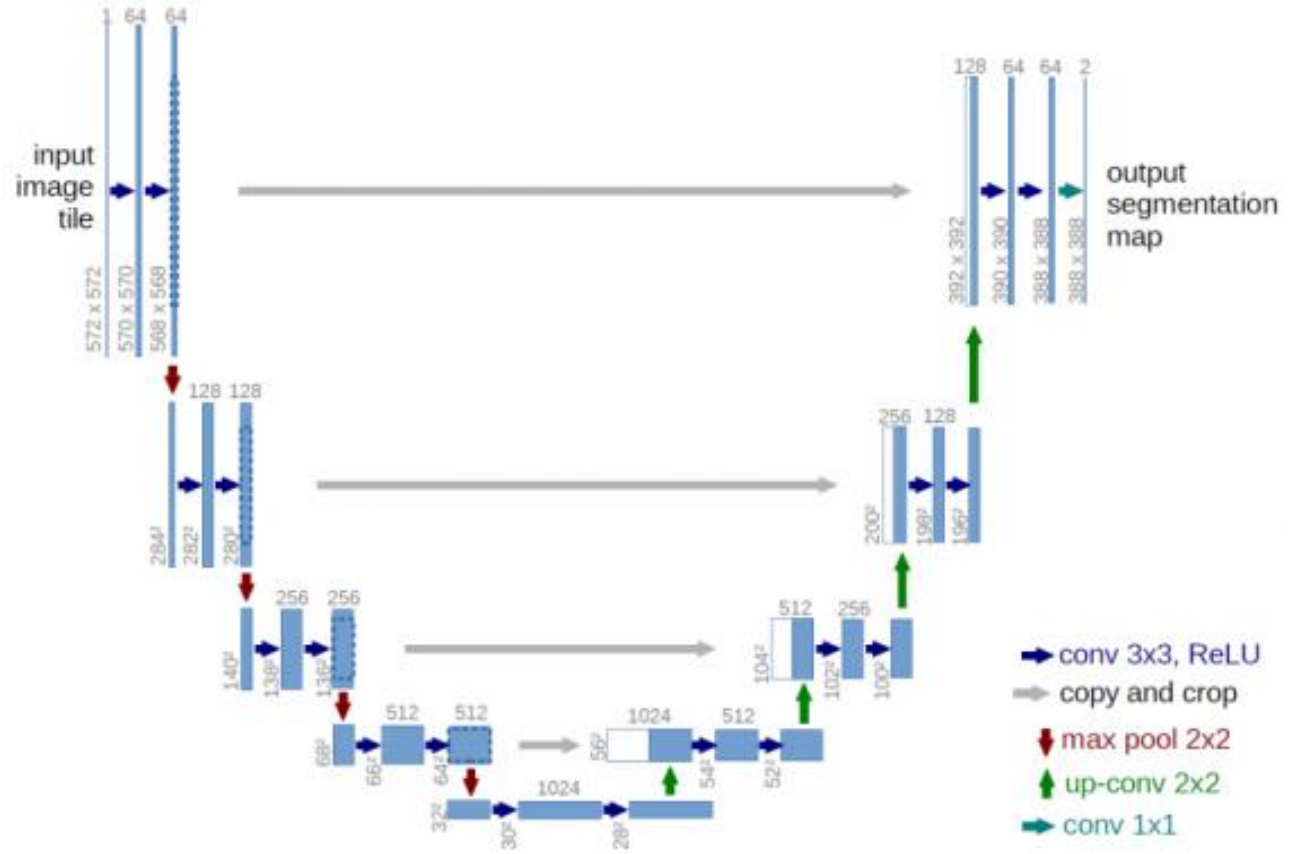


**Рисунок 2.4 Архитектура UNet**

Еще одним вариантом архитектуры предназначенной для сегментации является Feature Pyramid Network(FPN)[10] (рис. 2.5).

Основной особенностью данной архитектуры является использование промежуточных состояний (карт признаков) нейронной сети непосредственно

для финального предсказания. Это, как и в случае с Unet, позволяет объединять иерархические и локальные признаки. А также решает другую проблему: иерархические признаки по мере продвижения вглубь нейронной сети охватывают все большие и большие части исходного изображения, что затрудняет сегментацию относительно небольших объектов. В случае архитектуры FPN использование карт признаков со всех слоев сети позволяет одинаково хорошо решать задачу сегментации как больших, так и маленьких объектов.

К отличительным особенностям FPN можно отнести и то, что при объединении карт признаков одинаковых размеров используется операция сложения, а не конкатенации. Такое решение способствует увеличению скорости обучения нейронной сети, так как существенно уменьшает вероятность столкнуться с проблемой затухающих градиентов.

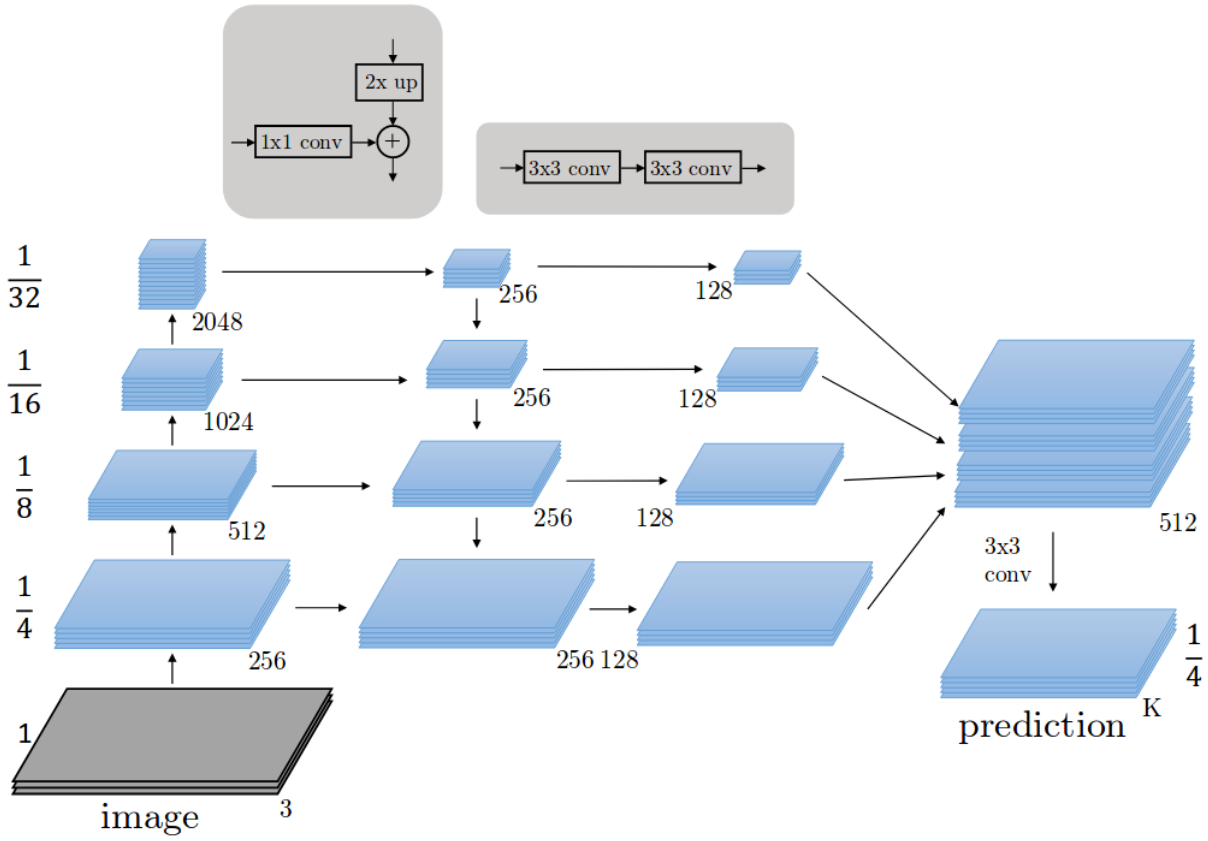


**Рисунок 2.5 Архитектура FPN**

Несмотря на то, что авторами архитектур были предложены определенные слои для выделения признаков, можно заменить эти части на сверточные части нейронных сетей, предназначенных для задач классификации, таких как VGG[11], ResNet [12], ResNeXt [13] и др.

## ВЫВОДЫ

Во второй главе были рассмотрены нейронные сети прямого распространения, в качестве общего примера работы нейронных. Изучены основные элементы сверточных нейронных сетей, применяемых для работы с изображениями, а также архитектуры сверточных сетей, предназначенные для задачи семантической сегментации.

# ГЛАВА 3. РЕШЕНИЕ ЗАДАЧИ СЕГМЕНТАЦИИ РЕЗУЛЬТАТОВ КОМПЬЮТЕРНОЙ ТОМОГРАФИИ

## 3.1 Выбор инструментов

В последнее время появилось большое количество программных систем машинного обучения. К их числу относятся программные библиотеки, расширения языков и даже самостоятельные языки программирования. Все эти системы позволяют использовать готовые алгоритмы создания обучения нейронных сетей.

Правильный выбор инструмента – важная задача, позволяющая достичь необходимого результата с наименьшими затратами сил и за наименьшее время. Рассмотрим наиболее популярные платформы для обучения нейронных сетей.

*TensorFlow* — это комплексная платформа для машинного обучения с открытым исходным кодом. Она была разработана командой Google Brain как продолжение закрытой системы машинного обучения DistBelief, однако в ноябре 2015 года компания передумала и открыла фреймворк для свободного доступа.

Как и большинство фреймворков глубокого обучения, TensorFlow имеет API на Python поверх механизма C и C ++, что ускоряет его работу.

TensorFlow имеет гибкую экосистему инструментов, библиотек и ресурсов сообщества. Это позволяет исследователям использовать самые современные МО-технологии, а разработчикам — создавать и развертывать приложения на базе машинного обучения.

Стоит отметить, что фреймворк постоянно развивается за счёт открытого исходного кода и огромного сообщества энтузиастов. Также за счет его популярности есть множество уже решённых задач, что существенно упрощает жизнь новоиспеченным разработчикам.

Однако фреймворк не лишён недостатков. Компания Google известна своей любовью к созданию собственных стандартов, что коснулось и фреймворка.

*Keras* — открытая среда глубокого обучения, написанная на Python. Она была разработана инженером из Google Франсуа Шолле и представлена в марте 2015 года.

Фреймворк нацелен на оперативную работу с нейросетями и является компактным, модульным и расширяемым. Подходит для небольших проектов, так как создать что-то масштабное на нём сложно, и он явно будет проигрывать в производительности нейросетей тому же TensorFlow.

Keras работает поверх TensorFlow, CNTK и Theano и предоставляет интуитивно понятный API.

Фреймворк содержит многочисленные реализации широко применяемых строительных блоков нейронных сетей, таких как слои, целевые и передаточные функции, оптимизаторы, а также множество инструментов для упрощения работы с изображениями и текстом.

*PyTorch* — это среда машинного обучения на языке Python с открытым исходным кодом, обеспечивающая тензорные вычисления с GPU-ускорением. Она была разработана компанией Facebook и представлена в октябре 2016 года, а открыта для сторонних разработчиков — в январе 2017 года. Фреймворк подходит для быстрого прототипирования в исследованиях, а также для любителей и небольших проектов.

Фреймворк предлагает динамические графы вычислений, которые позволяют обрабатывать ввод и вывод переменной длины, что полезно, например, при работе с рекуррентными нейронными сетями. Если коротко, то за счёт этого инженеры и исследователи могут менять поведение сети «налету».

В отличие от TensorFlow, PyTorch менее гибок в поддержке различных платформ. Также в нём нет родных инструментов для визуализации данных, но есть сторонний аналог, называемый tensorboardX.

Также при развёртке сетей на GPU PyTorch самостоятельно займёт только необходимую видеопамять.

Для решения задачи был выбран PyTorch. Он прост в освоении, удобен и гибок в плане обучения моделей. Также для задач сегментации существует библиотека Segmentation Models PyTorch, в которой реализовано большое количество архитектур.

## 3.2 Постановка задачи

В качестве задачи для экспериментального исследования была выбрана задача сегментации печени и желчного пузыря на результатах компьютерной томографии.

Сама задача была рассмотрена в главе 1 данной работы.

Для решения задачи использовался набор данных содержащий более 100 результатов компьютерной томографии.

В качестве обучающей выборки использовалось 80% имеющихся данных, то есть 84 компьютерные томограммы, для валидации – 10% (11 КТ), для теста – 10%(11 КТ).

Каждый из результатов компьютерной томографии представляет из себя около 250 отдельных снимков размера 512512. Примеры исходных данных и ожидаемых результатов (бинарных масок органов) представлены на рисунках 3.1 и 3.2.

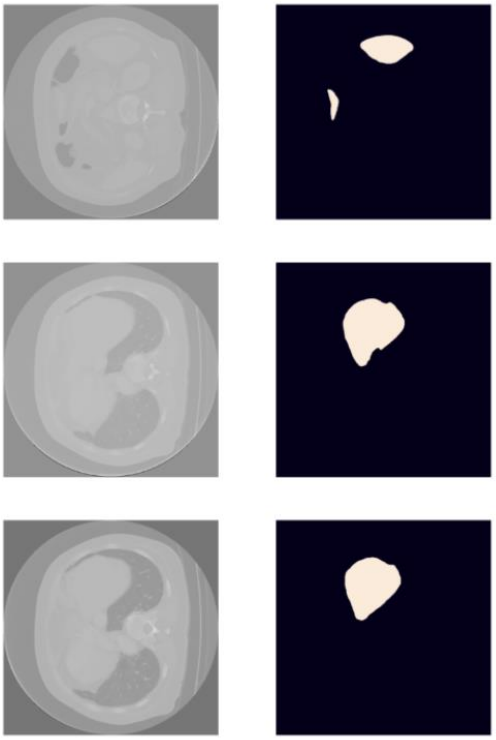

**Рисунок 3.1 Пример данных для сегментации печени**

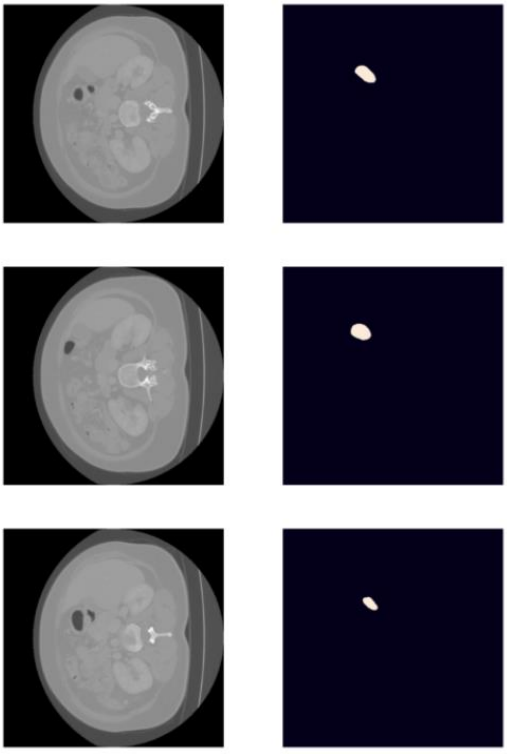

**Рисунок 3.2 Пример данных для сегментации желчного пузыря**

### 3.3 Метод решения

Так как входные данные нейронной сети (результаты КТ) имеют большой размер то обучение моделей, работающих с ними как с 3D объектами, является затруднительным, в силу вычислительной сложности. В связи с этим было принято решение работать с отдельными срезами КТ и решать задачу сегментации двумерных изображений.

Для ускорения работы нейронных сетей при использовании в конечном приложении сначала с помощью классификатора предсказывается наличие интересующего органа, и лишь затем, если орган присутствует на снимке, к этому снимку применяется нейронная сеть предназначенная для сегментации.

Также нейросети “специализирующиеся” на одном, а не нескольких органах, зачастую работают лучше. Поэтому для из органов использовались отдельные модели.

Таким образом, при использующемся подходе, для работы с n органами необходимо обучить 2n моделей (классификатор и сегментатор для каждого из органов).

### 3.4 Обучение моделей

Для выбора используемых моделей было проведено более 20 экспериментов, с их результатами можно ознакомиться в приложении А. В данной части будут рассмотрены общие подходы, которые использовались при решении задачи.

Код использованный для обучения моделей приведен в приложении Б.

*Функции потерь*

Для задач классификации самой популярной функцией потерь является бинарная кросс энтропия(BCE):

$$BCELoss(y, \bar{y}) = -[\beta y log(\bar{y}) + (1-\beta)(1-y)\log(1-\bar{y})]$$

Где параметр $\beta$ отвечает за вес положительных и негативных примеров.

Обобщением этой функции потерь является Focal Loss. Введем

$$p_t = \bar{y}, \text{при } y = 1$$

$$p_t = 1 - \bar{y}, \text{иначе}$$

Аналогичным образом вводится $\alpha_t$. Тогда:

$$FocalLoss = -\alpha_t(1 - p_t)^{\gamma} log(p_t)$$

Здесь параметр $\alpha$отвечает за вес положительных и негативных примеров. А параметр $\gamma$ используется для того чтобы сильнее штрафовать модель за большие ошибки.

Так как задача сегментации по своей сути является задачей классификации каждого пикселя на принадлежность к классу, то при обучении нейросетей можно использовать функции потерь для задачи классификации. В то же время существуют функции потерь предназначенные для задач сегментации. Например, Dice Loss:

$$DiceLoss = 1 - \frac{2y\bar{y} + 1}{y + \bar{y} + 1}$$

Все предложенные функции потерь можно комбинировать, суммируя их с различными весами.

*Аугментации*

При обучении нейронных сетей можно столкнуться с ситуацией, когда набор данных имеет ограниченный размер. Но для получения лучших результатов модели необходимо иметь больших данных, в том числе различные вариации этих данных. В таком случае прибегают к приему, называемому аугментацией данных. Аугментация данных – методика создания дополнительных данных из уже имеющихся. Кроме того использование аугментаций позволяет улучшить обобщающую способность сети, выступая регуляризатором. Наиболее популярными аугментациями являются: отображение по вертикали или горизонтали, поворот изображения на определенный угол, добавление шума, вырезание части изображения, манипуляции с цветом и различные комбинации описанных выше подходов.

При выборе аугментаций необходимо учитывать специфику данных, чтобы не вносить смещение в распределение, так как это вместо ожидаемого повышения точности работы нейронной сети, может наоборот ухудшить точность.

В связи со спецификой решаемой задачи для изображений печени использовались случайные повороты изображения на угол до 15 и вырезание части изображения случайным образом (до размера входа нейронной сети). Для изображений желчного пузыря из-за маленького размера органа использовались лишь случайные повороты, так как при случайных вырезаниях, в большинстве случаев, сам желчный пузырь не попадал бы в изображение.

*Label smoothing*

Еще одной проблемой в машинном обучении является тенденция алгоритмов делать слишком уверенные предсказания, которые в свою очередь приводят к слишком долгому обучению нейронной сети. Для борьбы с этим предназначена техника, получившая название сглаживание меток (англ. label smoothing).

При ее использовании метки в обучающей выборке заменяются следующим образом:

$$\bar{y} = (1 - \alpha)y + \frac{\alpha}{K}$$

Где $\alpha$ – гиперпараметр отвечающий за новые значения меток, а $K$ – количество классов в рассматриваемой задаче.

## 3.3 Полученные результаты

В результате были разработаны нейронные сети способные сегментировать печень и желчный пузырь. Было замерено качество полученных моделей (таблица 3.1).

**Таблица 3.1 Метрики качества полученных моделей**

| | IoU | Recall | Precision | F1 |
|---|---|---|---|---|
| Печень | 0.958 | 0.979 | 0.978 | 0.978 |
| Желчный пузырь | 0.817 | 0.887 | 0.911 | 0.898 |

Также был разработан API с использованием веб-фреймворка Flask, который может быть использован для внедрения модели в веб-сервисы (реализация приведена в приложении В).

Ниже представлены примеры полученных масок сегментации для различных срезов из тестового набора данных.

## ВЫВОДЫ

В данной главе была кратко рассмотрена предметная область, а также рассмотрена задача семантической сегментации и способы оценки полученных при решении задачи результатов.

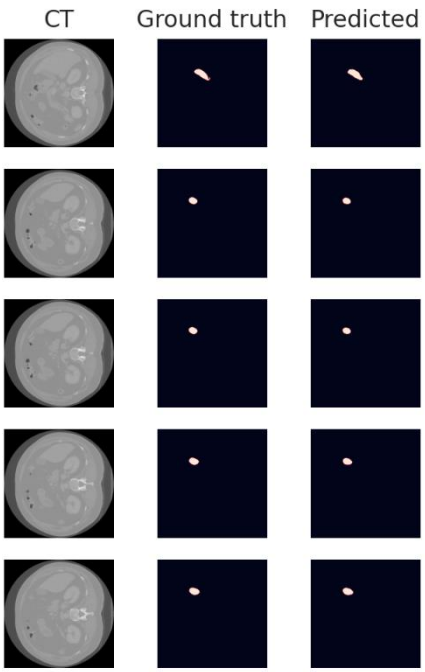


**Рисунок 3.1 Примеры результатов сегментации желчного пузыря**

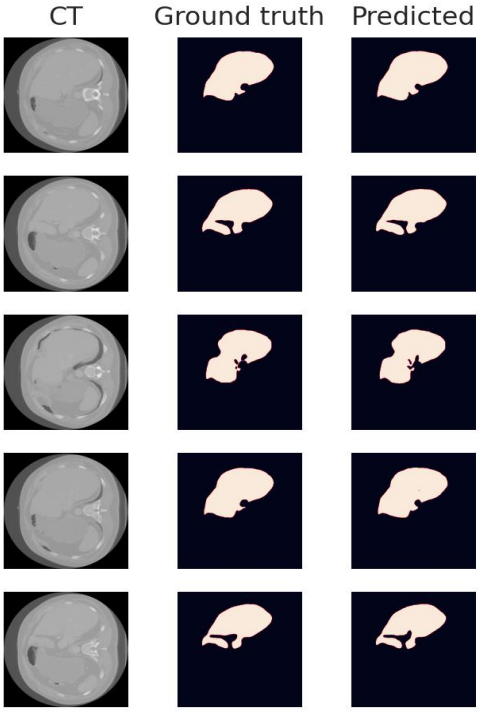


**Рисунок 3.2 Примеры результатов сегментации печени**

## ВЫВОДЫ

В третьей главе в качестве практической задачи была решена задача семантической сегментации снимков компьютерной томографии, были рассмотрены и использованы различные техники обучения нейронных сетей. Также был разработан модуль, позволяющий встроить обученные нейронные сети в веб приложение.

# ЗАКЛЮЧЕНИЕ

В работе были рассмотрены методы семантической сегментации изображений, основанные на применении нейронных сетей.

В ходе выполнения практической задачи по сегментации снимков компьютерной томографии были разработаны модули, позволяющие проводить различные эксперименты по обучению моделей изменяя лишь конфигурационные файлы, и не меняя при этом код используемый для обучения. Благодаря такому подходу удается проверять большое количество гипотез, связанных с работой нейросетей, что в свою очередь позволяет добиваться наилучшего качества решения практической задачи.

Также был разработан модуль, позволяющий легко интегрировать обученные модели в веб-сервисы.

Стоит отметить, что приложение “AIbolit.3D”, использующее разработанный модуль, не только принесло опыт и пользу в качестве дипломной работы, но и успешно используется в настоящий момент на практике. AIbolit.3D является инструментом для врачей, с помощью которого они могут в считанные минуты, пользуясь браузером, создавать трёхмерные модели органов по КТ пациентов. Построенные модели загружаются в очки дополненной реальности Microsoft Hololens, в которых доктор может наложить голограмму перелома на оперируемый участок тела и эффективно провести разметку маркером, планируя ход операции и, таким образом, минимизируя повреждения мягких тканей человека.

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ


1. Компьютерная томография – Википедия [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D1%82%D0%BE%D0%BC%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%8F

2. Соколов Е. Семинары по выбору моделей [Электронный ресурс]. – Режим доступа: http://www.machinelearning.ru/wiki/images/1/1c/sem06_metrics.pdf

3. LeCun Y. Backpropagation Applied to Handwritten Zip Code Recognition [Electronic resource]. – Mode of access: http://yann.lecun.com/exdb/publis/pdf/lecun-89e.pdf

4. Я. Гудфеллоу Глубокое обучение / пер. с англ. А. А. Слинкина – 2-е изд., испр. – М.: ДМК Пресс, 2018 – 652 с.: цв. ил.

5. [Электронный ресурс]. – Режим доступа: https://programforyou.ru/poleznoe/convolutional-network-from-scratch-parttwo-pooling-layer

6. Пакетная нормализация - Batch normalization [Электронный ресурс]. – Режим доступа: https://livepcwiki.ru/wiki/Batch_normalization

7. Hinton G. Improving neural networks by preventing co-adaptation of feature detectors [Electronic resource]. – Mode of access: https://arxiv.org/abs/1207.0580.pdf

8. Shelhammer E., Long J. Fully Convolutional Networks for Semantic Segmentation [Electronic resource]. – Mode of access: https://arxiv.org/pdf/1605.06211.pdf

9. Ronneberger O. U-Net: Convolutional Networks for Biomedical Image Segmentation [Electronic resource]. – Mode of access: https://arxiv.org/pdf/1505.04597.pdf

10. Seferbekov S. Feature Pyramid Network for Multi-Class Land Segmentation [Electronic resource]. – Mode of access:

https://openaccess.thecvf.com/content_cvpr_2018_workshops/papers/w4/Seferbekov_Feature_Pyramid_Network_CVPR_2018_paper.pdf

11. Simonyan K., Zisserman A. Very Deep Convolutional Networks for Large-Scale Image Recognition [Electronic resource]. – Mode of access: https://arxiv.org/pdf/1409.1556.pdf

12. He K. Deep Residual Learning for Image Recognition [Electronic resource]. – Mode of access: https://arxiv.org/pdf/1512.03385.pdf

13. Xie S. Aggregated Residual Transformations for Deep Neural Networks [Electronic resource]. – Mode of access: https://arxiv.org/pdf/1611.05431.pdf

# ПРИЛОЖЕНИЯ

## ПРИЛОЖЕНИЕ А

**Таблица А1 – Результаты проведенных экспериментов по выбору модели для классификации печени**

| Архитектура | Функция потерь | Оптимизатор | Learning rate | Количество эпох | Recall | f1 |
|---|---|---|---|---|---|---|
| VGG16 | Focal $\gamma = 2$ | SGD | 0.001 | 15 | 0.9407 | 0.9212 |
| ResNet18 | BCE $\beta = 0.67$ | Adam | 0.003 | 15 | 0.9803 | 0.9405 |
| ResNet18 | Focal $\gamma = 2$ | Adam | 0.001 | 15 | 0.95 | 0.9276 |
| ResNet18 | BCE | SGD | 0.001 | 15 | 0.9602 | 0.933 |

**Таблица А2 – Результаты проведенных экспериментов по выбору модели для классификации желчного пузыря**

| Архитектура | Функция потерь | Оптимизатор | Learning rate | Нормализация входного изображения | Количество эпох | Recall | F1 | Примечания |
|---|---|---|---|---|---|---|---|---|
| ResNeXt50 | Focal Loss $\gamma = 2$ | Adam | 0.0001 | [-100, 200] | 30 | 0.9512 | 0.9237 | Использовался label smoothing c =0.2 |
| ResNeXt50 | BCE | Adam | 0.001 С понижением в 5 раз каждые 5 эпох | [-100, 200] | 25 | 0.9896 | 0.9120 | Использовался label smoothing c =0.2 |
| DenseNet121 | Focal Loss $\gamma = 2$ | Adam | 0.001 С понижением в 5 раз каждые 5 эпох | [-150, 220] | 20 | 0.9282 | 0.8208 | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ResNet 34 | BCE | Adam | 0.0001 | [-150, 220] | 30 | 0.8854 | 0.8035 | |
| ResNet 18 | BCE | Adam | 0.0001 | - | 17 | 0.9013 | 0.8521 | Входное изображение повторялось 3 раза, для использования предобученных весов |

**Таблица А3 – Результаты проведенных экспериментов по выбору модели для сегментации печени**

| Архитектура | Энкодер | Функция потерь | Оптимизатор | Learning rate | Нормализация входного изображения | Количество эпох | IoU на валидационной части датасета | Примечания |
|---|---|---|---|---|---|---|---|---|
| FPN | ResNet 34 | Focal Loss $\gamma = 2$ | Adam | 256x256: 0.001 384x384: 0.0001 512x512: 0.0001 С понижением в 2 раза каждые 5 эпох для каждого из размеров входного изображения | [-200, 350] | 48 | 0.9707 | 36 эпох обучение проводилось на размере 256x256 и еще по 6 на 384x384 и 512x512 |
| UNet | VGG-19 | Focal Loss $\gamma = 2$ | Adam | 0.001 | [-200, 200] | 20 | 0.8237 | - |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| FPN | VGG-19 | Focal Loss $\gamma = 2$ | Adam | 0.003 | [-200, 200] | 20 | 0.913 | - |
| UNet | ResNeXt 101 32x8d | BCE | Adam | 0.001 | [-200, 200] | 25 | 0.95 | |
| UNet | ResNet 18 | BCE | SGD | 0.0005 | - | 15 | 0.9334 | - |

**Таблица А4 – Результаты проведенных экспериментов по выбору модели для сегментации желчного пузыря**

| Архитектура | Энкодер | Функция потерь | Оптимизатор | Learning rate | Нормализация входного изображения | Количество эпох | IoU на валидационной части датасета | Примечания |
|---|---|---|---|---|---|---|---|---|
| UNet | ResNeXt 101 32x16d | Focal Loss $\gamma = 2$ weight=0.8 | SGD | 0.001 | [-100, 200] | 20 | 0.8551 | К входному изображению добавлялась маска печени Глубина энкодера – 4 |
| UNet | ResNeXt 101 32x16d | DiceFocal Loss $\gamma = 2$ focal_weigth=0.8 | Adam | 0.001 С понижением в 10 раз каждые 7 эпох | [-100, 200] | 27 | 0.8707 | К входному изображению добавлялась маска печени Проводилась попытка дообучить сеть |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | на картинках размера (384x384) в течении 10 эпох, но она не привела к успеху |
| UNet | ResNeXt 101 32x16d | Focal Loss $\gamma = 2$ | Adam | 0.003 С понижением в 10 раз каждые 7 эпох | [-100, 200] | 20 | 0.8488 | К входному изображению добавлялась маска печени |
| FPN | SEResNet 50 | DiceFocal Loss $\gamma = 2$ focal_weigth=0.8 | Adam | 0.001 С понижением в 5 раз каждые 5 эпох | [-100, 200] | 19 | 0.8481 | К входному изображению добавлялась маска печени |
| FPN | ResNeXt 101 32x16d | BCE | SGD | 0.001 | [-150, 220] | 20 | 0.794 | - |
| FPN | ResNeXt 101 32x16d | BCE | Adam | Сегментационная часть сети: 0.001 С понижением в 10 раз каждые 10 эпох Энкодер: | [-150, 220] | 15 | 0.8513 | Входное изображение повторялось 3 раза, для использования предобученных весов |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | 0.00001 | | | | |
| FPN | ResNet34 | Focal Loss $\gamma = 2$ | Adam | 0.003 | [-150, 220] | 15 | 0.8264 | - |
| LinkNet | SE ResNeXt 101 32x4d | Focal Loss $\gamma = 2$ | Adam | 0.001 С понижением в 5 раз каждые 5 эпох | [-150, 220] | 32 | 0.8514 | - |
| UNet | VGG 19-BN | BCE | SGD | Сегментационная часть сети: 0.001 С понижением в 10 раз каждые 10 эпох Энкодер: 0 | [-100, 200] | 30 | 0.86 | Входное изображение повторялось 3 раза, для использования предобученных весов |
| UNet | ResNet101 | BCE | Adam | 0.001 | [-150, 220] | 15 | 0.7996 | - |

Файл config.py:

```
import json
import typing
from abc import ABC

import addict
import monai
import segmentation_models_pytorch as smp
import torch


class ConfigException(Exception):
    pass


class AbstractConfig(ABC):
    _optimizers = {
        "adadelta": torch.optim.Adadelta,
        "adagard": torch.optim.Adagrad,
        "adam": torch.optim.Adam,
        "adamw": torch.optim.AdamW,
        "sparseadam": torch.optim.SparseAdam,
        "adamax": torch.optim.Adamax,
        "asgd": torch.optim.ASGD,
        "lbfgs": torch.optim.LBFGS,
        "rmsprop": torch.optim.RMSprop,
        "rprop": torch.optim.Rprop,
        "sgd": torch.optim.SGD,
    }

    _losses = {
        "dice": monai.losses.DiceLoss,
        "generalized dice": monai.losses.GeneralizedDiceLoss,
        "dice_ce": monai.losses.DiceCELoss,
        "dice_focal": monai.losses.DiceFocalLoss,
        "focal": monai.losses.FocalLoss,
        "tversky": monai.losses.TverskyLoss,
        "contrastive": monai.losses.ContrastiveLoss,
        "bce_with_logits": torch.nn.BCEWithLogitsLoss,
        "smp_focal": smp.losses.FocalLoss,
    }

    @classmethod
    def from_json(cls, json_path: str):
        with open(json_path) as config_file:
            config_dict = json.load(config_file)
        return cls(addict.Dict(config_dict))

    def get_batch_size(self) -> int:
        return self.batch_size

    def get_num_epochs(self) -> int:
        return self.epochs

    def get_checkpoint(self) -> typing.Optional[str]:
        return self.ckpt_path

    # LOSS
    @property
```

```
def loss_fn(self):
    return AbstractConfig._losses[self.loss_info.name.lower()]

@property
def loss_params(self) -> typing.Dict[str, typing.Any]:
    return self.loss_info.params.to_dict()

# DATA
@property
def csv_path(self) -> str:
    return self.data_info.path

@property
def train_cts(self) -> typing.List[int]:
    return self.data_info.train_cts

@property
def validation_cts(self) -> typing.List[int]:
    return self.data_info.validation_cts

@property
def organ_name(self) -> str:
    return self.data_info.organ

@property
def normalization(self) -> typing.Optional[typing.Tuple[int, int]]:
    norm = self.data_info.normalization
    if norm is None or len(norm) != 2:
        return None
    return tuple(norm)

@property
def image_size(self) -> typing.Optional[typing.Tuple[int, int]]:
    size = self.data_info.image_size
    if size is None or len(size) != 2:
        return None
    return tuple(size)

@property
def save_dir(self) -> str:
    return self.data_info.save_dir

@property
def label_smoothing(self):
    return self.data_info.label_smoothing

@property
def extra_organs(self) -> typing.Optional[typing.List]:
    return self.data_info.extra_organs

# OPTIMIZER
@property
def optimizer(self) -> torch.optim.Optimizer:
    return AbstractConfig._optimizers[self.optimizer_info.name.lower()]

@property
def optimizer_params(self) -> typing.Dict[str, typing.Any]:
    return self.optimizer_info.params.to_dict()

@property
def is_encoder_frozen(self) -> bool:
    return self.froze_encoder
```

```
    # WANDB INFO
    @property
    def project(self) -> str:
        return self.wandb_info.project

    @property
    def run(self) -> str:
        return self.wandb_info.name


class SegmentationTrainingConfig(AbstractConfig):
    def __init__(self, config_dict: addict.Dict):
        self.model_info = config_dict.model
        self.optimizer_info = config_dict.optimizer
        self.loss_info = config_dict.loss
        self.data_info = config_dict.data
        self.wandb_info = config_dict.wandb
        self.batch_size = config_dict.batch_size
        self.epochs = config_dict.epochs
        self.seed = config_dict.random_seed
        self.use_amp = config_dict.use_amp
        self.ckpt_path = config_dict.ckpt_path
        self.froze_encoder = config_dict.forze_encoder

    @property
    def model_arch(self) -> str:
        return self.model_info.arch

    @property
    def model_params(self) -> typing.Dict:
        return self.model_info.to_dict()

    @property
    def encoder_name(self) -> str:
        return self.model_info.encoder_name

    @property
    def only_organ_present(self) -> bool:
        return self.data_info.only_organ_present

    def set_encoder_name(self, encoder_name: str):
        self.model_info.encoder_name = encoder_name

    def set_model_arch(self, arch: str):
        self.model_info.arch = arch


class ClassificationTrainingConfig(AbstractConfig):
    def __init__(self, config_dict: addict.Dict):
        self.model_info = config_dict.model
        self.optimizer_info = config_dict.optimizer
        self.loss_info = config_dict.loss
        self.data_info = config_dict.data
        self.wandb_info = config_dict.wandb
        self.batch_size = config_dict.batch_size
        self.epochs = config_dict.epochs
        self.seed = config_dict.random_seed
        self.use_amp = config_dict.use_amp
        self.ckpt_path = config_dict.ckpt_path
        self.monitor_metric = config_dict.monitor_metric

    @property
    def model_params(self):
```

```
        return self.model_info.to_dict()

    @property
    def model_arch(self):
        return self.model_info.arch

    def get_monitor_metric(self):
        return self.monitor_metric
```

Файл data.py:

```
import typing

import numpy as np
import pandas as pd
import torch


class OrganClassificationDataset(torch.utils.data.Dataset):
    def __init__(
        self,
        path_to_csv: str,
        organ_name: str,
        CTs: typing.Optional[typing.Sequence[int]] = None,
        transform: typing.Optional[typing.Callable] = None,
        augmentations: typing.Optional[typing.Callable] = None,
        label_smoothing: float = 0.0,
        normalization: typing.Optional[typing.Tuple[int, int]] = None,
    ) -> None:
        """
        Params:
             path_to_csv: str
                Path to csv file with name of files with slices and
corresponding label
            organ_name: str
                Organ name to extract labels for
            CTs: Optional[Sequence[int]], default: None
                Names of CTs to be present in dataset, if None all available CTs
from csv will be used
            transform: Optional[Callable] ,default: None
                Transformation to perform on image
            augmentations: Optional[Callable], default: None
                Augmentations from ALBUMENTATIONS LIBRARY TO PERFORM
                Applied after transformations (transform param)
            label_smoothing: float, default: 0.0
                Label will be: (1 - label_smoothing) * ground_truth_label +
label_smoothing / 2
            normalization: Optional[Tuple[int, int]], default: None
                If None, normalization is not performed.
                If tuple it should be (lower bound, upper bound). All pixels
that < lower bound will be equal lower bound
                and all pixels > upper bound will be equal upper bound. Then
MinMaxScaling is performed.
        """
        buffer_df = pd.read_csv(path_to_csv)
        self.transform = transform

        # Leave only required part of dataset
        if CTs is not None:
            buffer_df["IS_OK"] = buffer_df["CT"].apply(lambda i: i in CTs)
            buffer_df = buffer_df[buffer_df["IS_OK"] == True]

        self.data_frame = pd.DataFrame()
```

```
        self.data_frame["path"] = buffer_df["path"]
        self.data_frame["label"] = buffer_df[organ_name.lower()]
        self.data_frame = self.data_frame.reset_index(drop=True)
        self.label_smoothing = label_smoothing
        self.augmentations = augmentations
        self.normalization = normalization

    def __len__(self):
        return len(self.data_frame)

    def __getitem__(self, idx):
        image_path = self.data_frame["path"].iloc[idx]
        with open(image_path, "rb") as f:
            image = np.load(f).astype(np.float32)

        if self.normalization is not None:
            image = np.where(
                image < self.normalization[0], self.normalization[0], image
            )
            image = np.where(
                image > self.normalization[1], self.normalization[1], image
            )
            image = (image - self.normalization[0]) / (
                self.normalization[1] - self.normalization[0]
            )

        if self.transform is not None:
            image = self.transform(image)

        if self.augmentations is not None:
            image = self.augmentations(image=image)["image"]

        label = torch.Tensor([self.data_frame["label"].iloc[idx]])
        label = (1 - self.label_smoothing) * label + self.label_smoothing / 2
        return image, label


class OrganSegmentationDataset(torch.utils.data.Dataset):
    def __init__(
        self,
        path_to_csv: str,
        organ_name: str,
        organ_mapping: int,
        CTs: typing.Optional[typing.Sequence[int]] = None,
        transform: typing.Optional[typing.Callable] = None,
        augmentations: typing.Optional[typing.Callable] = None,
        label_smoothing: float = 0.0,
        normalization: typing.Optional[typing.Tuple[int, int]] = None,
        only_organ_present: bool = True,
        extra_organs: typing.Optional[typing.List[int]] = None,
    ) -> None:
        """
        Params:
            path_to_csv: str
                Path to csv file with name of files with slices and
corresponding label
            organ_name: str
                Organ name to extract labels for
            organ_mapping: int
                Code of organ on segmented images
            CTs: Optional[Sequence[int]], default: None
                Names of CTs to be present in dataset, if None all available CTs
from csv will be used
```

```
            transform: Optional[Callable] ,default: None
                Transformation to perform on image
            augmentations: Optional[Callable], default: None
                Augmentations from ALBUMENTATIONS LIBRARY TO PERFORM
                Applied after transformations (transform param)
            label_smoothing: float, default: 0.0
                Label will be: (1 - label_smoothing) * ground_truth_label +
label_smoothing / 2
            normalization: Optional[Tuple[int, int]], default: None
                If None, normalization is not performed.
                If tuple it should be (lower bound, upper bound). All pixels
that < lower bound will be equal lower bound
                and all pixels > upper bound will be equal upper bound. Then
MinMaxScaling is performed.
            only_organ_present: bool, default: True
                If True images where organ is not present are removed from
datatset
            extra_organs: Optional[List[int]], default: None
                If list is passed then, masks of corresponding organ_ids will be
passed as input info. If None only CT image is passed
        """

        buffer_df = pd.read_csv(path_to_csv)

        # Leave only required part of dataset
        if CTs is not None:
            buffer_df["IS_OK"] = buffer_df["CT"].apply(lambda i: i in CTs)
            buffer_df = buffer_df[buffer_df["IS_OK"] == True]

        if only_organ_present:
            buffer_df = buffer_df[buffer_df[organ_name.lower()] == 1]

        self.data_frame = pd.DataFrame()
        self.data_frame["image_path"] = buffer_df["path"]
        self.data_frame["mask_path"] = buffer_df["segmented_path"]
        self.data_frame = self.data_frame.reset_index(drop=True)

        self.transform = transform
        self.augmentations = augmentations
        self.organ_mapping = organ_mapping
        self.label_smoothing = label_smoothing
        self.normalization = normalization
        self.extra_organs = extra_organs

    def __len__(self):
        return len(self.data_frame)

    def __getitem__(self, idx):
        image_path = self.data_frame["image_path"].iloc[idx]
        with open(image_path, "rb") as f:
            image = np.load(f).astype(np.float32)
        image = np.expand_dims(image, -1)

        mask_path = self.data_frame["mask_path"].iloc[idx]
        with open(mask_path, "rb") as f:
            original_mask = np.load(f).astype(np.float32)
        mask = np.expand_dims(original_mask, -1)
        mask = np.where(mask == self.organ_mapping, 1, 0).astype(np.float32)

        # perform image normalization if needed
        if self.normalization is not None:
            image = np.where(
                image < self.normalization[0], self.normalization[0], image
```

```
            )
            image = np.where(
                image > self.normalization[1], self.normalization[1], image
            )
            image = (image - self.normalization[0]) / (
                self.normalization[1] - self.normalization[0]
            )

        # add info from other organs as input
        # TODO: naming
        if self.extra_organs is not None:
            for organ_id in self.extra_organs:
                additional_mask = np.where(original_mask == organ_id, 1, 0)
                additional_mask = np.expand_dims(additional_mask, -1)
                image = np.concatenate((image, additional_mask), axis=-
1).astype(
                    np.float32
                )

        if self.transform is not None:
            image = self.transform(image)
            mask = self.transform(mask)

        if self.augmentations is not None:
            d = self.augmentations(image=image, mask=mask)
            image = d["image"]
            mask = d["mask"]

        mask = (1 - self.label_smoothing) * mask + self.label_smoothing / 2

        return image, mask
```

Файл models.py:

```
import pytorch_lightning as pl
import segmentation_models_pytorch as smp
import torch
import torchvision
from config import SegmentationTrainingConfig


class SegmentationModel(pl.LightningModule):
    def __init__(
        self,
        config: SegmentationTrainingConfig,
    ):
        super().__init__()
        self.config = config
        self.model = smp.create_model(**self.config.model_params)

        self.loss_fn = config.loss_fn(**self.config.loss_params)

    def set_loss(self, loss):
        self.loss_fn = loss

    def forward(self, x):
        mask = self.model(x)
        return mask

    def _step(self, batch, stage):
        image = batch[0]
        mask = batch[1]
        logits_mask = self.model(image)
```

```
        loss = self.loss_fn(logits_mask, mask)

        if stage in ["train", "val"]:
            self.log(
                f"{stage} loss", loss, on_step=False, on_epoch=True,
prog_bar=False
            )

        prob_mask = logits_mask.sigmoid()
        pred_mask = (prob_mask > 0.5).float()

        tp, fp, fn, tn = smp.metrics.get_stats(
            pred_mask.long(), mask.long(), mode="binary"
        )

        return {"loss": loss, "tp": tp, "fp": fp, "tn": tn, "fn": fn}

    def _epoch_end(self, outputs, stage):
        tp = torch.cat([x["tp"] for x in outputs])
        fp = torch.cat([x["fp"] for x in outputs])
        tn = torch.cat([x["tn"] for x in outputs])
        fn = torch.cat([x["fn"] for x in outputs])

        per_image_iou = smp.metrics.iou_score(
            tp, fp, fn, tn, reduction="micro-imagewise"
        )
        dataset_iou = smp.metrics.iou_score(tp, fp, fn, tn, reduction="micro")

        self.log(f"{stage} per image iou", per_image_iou, prog_bar=True)
        self.log(f"{stage} dataset iou", dataset_iou, prog_bar=True)

    def training_step(self, batch, batch_idx):
        return self._step(batch, "train")

    def validation_step(self, batch, batch_idx):
        return self._step(batch, "val")

    def test_step(self, batch, batch_idx):
        return self._step(batch, "test")

    def training_epoch_end(self, outputs):
        self._epoch_end(outputs, "train")

    def validation_epoch_end(self, outputs):
        self._epoch_end(outputs, "val")

    def test_epoch_end(self, outputs):
        self._epoch_end(outputs, "test")

    def configure_optimizers(self):
        if self.config.is_encoder_frozen:
            optim = self.config.optimizer(
                self.model.decoder.parameters(), **self.config.optimizer_params
            )
        else:
            optim = self.config.optimizer(
                self.model.parameters(), **self.config.optimizer_params
            )
        scheduler = torch.optim.lr_scheduler.StepLR(optim, step_size=10,
gamma=0.1)
        return [optim], [scheduler]
```

```
def create_classification_model(
    arch: str,
    in_channels: int,
    classes: int,
    pretrained: bool = False,
) -> torch.nn.Module:
    available_models = [
        "vgg16",
        "vgg16_bn",
        "vgg19",
        "vgg19_bn",
        "resnet18",
        "resnet34",
        "resnet50",
        "resnet101",
        "densenet121",
        "densenet169",
        "resnext50",
        "resnext101",
    ]
    if arch.lower() not in available_models:
        raise ValueError(
            f"Unknown model architecture {arch}. Use one of {available_models}"
        )
    arch = arch.lower()

    if arch.startswith("vgg"):
        if arch == "vgg16":
            model = torchvision.models.vgg16(pretrained=pretrained)
        elif arch == "vgg16_bn":
            model = torchvision.models.vgg16_bn(pretrained=pretrained)
        elif arch == "vgg19":
            model = torchvision.models.vgg19(pretrained=pretrained)
        elif arch == "vgg19_bn":
            model = torchvision.models.vgg19_bn(pretrained=pretrained)

        model.features[0] = torch.nn.Conv2d(
            in_channels, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)
        )
        model.classifier[6] = torch.nn.Linear(
            in_features=4096, out_features=classes, bias=True
        )

    elif arch.startswith("resnet"):
        if arch == "resnet18":
            model = torchvision.models.resnet18(pretrained=pretrained)
            model.fc = torch.nn.Linear(in_features=512, out_features=classes,
bias=True)
        elif arch == "resnet34":
            model = torchvision.models.resnet34(pretrained=pretrained)
            model.fc = torch.nn.Linear(in_features=512, out_features=classes,
bias=True)
        elif arch == "resnet50":
            model = torchvision.models.resnet50(pretrained=pretrained)
            model.fc = torch.nn.Linear(
                in_features=2048, out_features=classes, bias=True
            )
        elif arch == "resnet101":
            model = torchvision.models.resnet101(pretrained=pretrained)
            model.fc = torch.nn.Linear(
                in_features=2048, out_features=classes, bias=True
            )
```

```
        model.conv1 = torch.nn.Conv2d(
            in_channels,
            64,
            kernel_size=(7, 7),
            stride=(2, 2),
            padding=(3, 3),
            bias=False,
        )

    elif arch.startswith("densenet"):
        if arch == "densenet121":
            model = torchvision.models.densenet121(pretrained=pretrained)
            model.classifier = torch.nn.Linear(
                in_features=1024, out_features=classes, bias=True
            )
        elif arch == "densenet169":
            model = torchvision.models.densenet169(pretrained=pretrained)
            model.classifier = torch.nn.Linear(
                in_features=1664, out_features=classes, bias=True
            )

        model.features.conv0 = torch.nn.Conv2d(
            in_channels,
            64,
            kernel_size=(7, 7),
            stride=(2, 2),
            padding=(3, 3),
            bias=False,
        )

    elif arch.startswith("resnext"):
        if arch == "resnext50":
            model = torchvision.models.resnext50_32x4d(pretrained=pretrained)
        elif arch == "resnext101":
            model = torchvision.models.resnext101_32x8d(pretrained=pretrained)

        model.conv1 = torch.nn.Conv2d(
            in_channels,
            64,
            kernel_size=(7, 7),
            stride=(2, 2),
            padding=(3, 3),
            bias=False,
        )
        model.fc = torch.nn.Linear(in_features=2048, out_features=classes,
bias=True)

    return model
```

Файл train.py:

```
import typing
import warnings

import numpy as np
import torch
import wandb
from sklearn.metrics import (
    classification_report,
    f1_score,
    precision_score,
    recall_score,
)
```

```
from tqdm.autonotebook import tqdm


class BCTrainer:
    # TODO: add support of amp
    """
    Class for Binary Classification train loop
    """

    def __init__(self, model: torch.nn.Module, logits_predictions: bool = True):
        """
        Params:
            model: torch.nn.Module
                Model to train
            logits_predictions: bool, default: True
                If model returns logits prediction use True, else use False
        """

        self.model = model
        self.logits_predictions = logits_predictions
        self.per_epoch_stats = {}

    def _bc_train_step(
        self,
        train_dl: torch.utils.data.DataLoader,
        validation_dl: torch.utils.data.DataLoader,
        optimizers: typing.Sequence[torch.optim.Optimizer],
        loss_fn,
        device: str,
    ) -> typing.Dict[str, typing.Any]:
        """
        Training step for binary classification problem
        """
        stats = {}
        train_loss = 0.0
        predictions = []
        real_labels = []
        self.model.train()
        for batch in tqdm(train_dl, total=len(train_dl), desc="Train",
leave=False):
            for optimizer in optimizers:
                optimizer.zero_grad()
            x = batch[0].to(device)
            y = batch[1].to(device)
            y_pred = self.model(x)

            if self.logits_predictions:
                predictions.append(
                    (y_pred.sigmoid() > 0.5).int().detach().cpu().numpy()
                )
                real_labels.append((y > 0.5).int().detach().cpu().numpy())
            else:
                predictions.append((y_pred > 0.5).int().detach().cpu().numpy())
                real_labels.append((y > 0.5).int().detach().cpu().numpy())

            loss = loss_fn(y_pred, y)
            loss.backward()
            for optimizer in optimizers:
                optimizer.step()

            train_loss += loss.item()

        predictions = np.concatenate(predictions)
```

```
        predictions = predictions.flatten()
        real_labels = np.concatenate(real_labels)
        real_labels = real_labels.flatten()
        stats["train_f1"] = f1_score(real_labels, predictions)
        stats["train_recall"] = recall_score(real_labels, predictions)
        stats["train_precission"] = precision_score(real_labels, predictions)
        stats["train_loss"] = train_loss / len(train_dl)

        self.model.eval()
        val_loss = 0.0
        predictions = []
        real_labels = []
        for batch in tqdm(
            validation_dl, total=len(validation_dl), desc="Validation",
leave=False
        ):
            with torch.no_grad():
                x = batch[0].to(device)
                y = batch[1].to(device)
                y_pred = self.model(x)

                if self.logits_predictions:
                    predictions.append(
                        (y_pred.sigmoid() > 0.5).int().detach().cpu().numpy()
                    )
                    real_labels.append((y > 0.5).int().detach().cpu().numpy())
                else:
                    predictions.append((y_pred >
0.5).int().detach().cpu().numpy())
                    real_labels.append((y > 0.5).int().detach().cpu().numpy())

                loss = loss_fn(y_pred, y)
                val_loss += loss.item()

        predictions = np.concatenate(predictions)
        predictions = predictions.flatten()
        real_labels = np.concatenate(real_labels)
        real_labels = real_labels.flatten()
        stats["val_f1"] = f1_score(real_labels, predictions)
        stats["val_recall"] = recall_score(real_labels, predictions)
        stats["val_precission"] = precision_score(real_labels, predictions)
        stats["val_loss"] = val_loss / len(train_dl)

        return stats

    def train(
        self,
        train_dl: torch.utils.data.DataLoader,
        validation_dl: torch.utils.data.DataLoader,
        optimizers: typing.Sequence[torch.optim.Optimizer],
        loss_fn,
        epochs: int,
        lr_schedulers: typing.Optional[typing.Sequence[typing.Any]] = None,
        device: str = "cuda",
        save_directory: str = "./",
        save_name: typing.Optional[str] = None,
        log_to_wandb: bool = True,
        monitor_metric: str = "val_f1",
    ) -> typing.Dict:
        """
        Params:
            train_dl: torch.utils.data.DataLoader
                Training dataloader
```

```
            validation_dl: torch.utils.data.DataLoader
                Validation dataloader
            optimizers: Sequence[torch.optim.Optimizer]
                Search space optimizers
            loss_fn:
                Loss function
            epochs: int
                Number of epochs to perform
            lr_schedulers: Optional[Sequence], default: None
                Learning rate schedulers, called at the end of epoch. None if
schedulers are not used.
            device: str, default: 'cuda'
                'cuda' or 'cpu'. Device used for training.
            save_directory: str, default: './'
                Directory to save models
            save_name: Optional[str], default: None
                Name of file model will be saved in. If none type(model) will be
used
            log_to_wandb: bool, default: True
                If true, training info is logged to wandb, and you have to call
wandb.init() before.
            monitor_metric: str, default: 'val_f1'
                Metric used to compare model perfomance and decide whether to
save model.
        """

        if save_name is None:
            save_name = f"{type(self.model).__name__}"

        if device != "cpu" and not torch.cuda.is_available():
            warnings.warn(f"{device} is not available, cpu will be used")
            device = "cpu"

        self.model.to(device)

        stats = {}

        for epoch in tqdm(range(epochs), desc="Epochs", postfix=stats):
            epoch_target_metric = 0.0
            best_target_metric = 0.0

            epoch_stats = self._bc_train_step(
                train_dl, validation_dl, optimizers, loss_fn, device
            )
            epoch_target_metric = epoch_stats[monitor_metric]
            self.per_epoch_stats[f"{epoch}"] = epoch_stats
            best_target_metric = stats.get(f"best_{monitor_metric}", -1.0)

            # stats['best_train_f1'] = max(stats.get('best_train_f1', -1.0),
epoch_stats['train_f1'])
            stats["best_val_f1"] = max(
                stats.get("best_val_f1", -1.0), epoch_stats["val_f1"]
            )

            # stats['best_train_precission'] =
max(stats.get('best_train_precission', -1.0), epoch_stats['train_precission'])
            stats["best_val_precission"] = max(
                stats.get("best_val_precission", -1.0),
epoch_stats["val_precission"]
            )

            # stats['best_train_recall'] = max(stats.get('best_train_recall', -
1.0), epoch_stats['train_recall'])
```

```
            stats["best_val_recall"] = max(
                stats.get("best_val_recall", -1.0), epoch_stats["val_recall"]
            )

            stats["train_f1"] = epoch_stats["train_f1"]
            stats["val_f1"] = epoch_stats["val_f1"]

            stats["train_precission"] = epoch_stats["train_precission"]
            stats["val_precission"] = epoch_stats["val_precission"]

            stats["train_recall"] = epoch_stats["train_recall"]
            stats["val_recall"] = epoch_stats["val_recall"]

            stats["train_loss"] = epoch_stats["train_loss"]
            stats["val_loss"] = epoch_stats["val_loss"]

            if lr_schedulers is not None:
                for scheduler in lr_schedulers:
                    scheduler.step()

            if log_to_wandb:
                wandb.log(stats)

            if epoch_target_metric > best_target_metric:
                self.model.to("cpu")
                torch.save(
                    self.model,
                    f"{save_directory}{save_name}_{int(epoch_target_metric *
1000)}.pt",
                )
                self.model.to(device)
        return self.per_epoch_stats

    def _bc_test(self, test_dl, device, threshold):
        predictions = []
        real_labels = []
        for batch in tqdm(
            test_dl,
            total=len(test_dl),
            desc=f"Test",
        ):
            with torch.no_grad():
                x = batch[0].to(device)
                y = batch[1].to(device)
                y_pred = self.model(x)

                if self.logits_predictions:
                    predictions.append(
                        (y_pred.sigmoid() >
threshold).int().detach().cpu().numpy()
                    )
                else:
                    predictions.append(
                        (y_pred > threshold).int().detach().cpu().numpy()
                    )
                real_labels.append((y > 0.5).int().detach().cpu().numpy())

        predictions = np.concatenate(predictions)
        predictions = predictions.flatten()
        real_labels = np.concatenate(real_labels)
        real_labels = real_labels.flatten()
        print(classification_report(real_labels, predictions))
```

```
    def test(
        self,
        test_dl: torch.utils.data.DataLoader,
        device: str = "cuda",
        threshold: float = 0.5,
    ) -> None:

        if device != "cpu" and not torch.cuda.is_available():
            device = "cpu"

        self.model.to(device)
        self.model.eval()
        if self.mode == "BC":
            self._bc_test(test_dl, device, threshold)
```

Файл pipeline.py:

```
import argparse
import os
import random

import albumentations as A
import cv2
import numpy as np
import pytorch_lightning as pl
import torch
import wandb
from albumentations.pytorch import ToTensorV2
from config import ClassificationTrainingConfig, SegmentationTrainingConfig
from data import OrganClassificationDataset, OrganSegmentationDataset
from train import BCTrainer

from models import SegmentationModel, create_classification_model

organ_mapping = {
    "gallbladder": 34,
    "liver": 36,

}


def set_determenistic(seed: int = 17) -> None:
    if seed is None:
        return
    np.random.seed(seed)
    random.seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
    torch.manual_seed(seed)


def segmentation_pipline(args):
    config = SegmentationTrainingConfig.from_json(args.config)

    set_determenistic(config.seed)

    train_dataset = OrganSegmentationDataset(
        path_to_csv=config.csv_path,
        organ_name=config.organ_name,
        organ_mapping=organ_mapping[config.organ_name.lower()],
        CTs=config.train_cts,
        normalization=config.normalization,
        augmentations=A.Compose(
```

```
        [
            # A.RandomCrop(config.image_size[0], config.image_size[1]),
            A.Resize(
                config.image_size[0],
                config.image_size[1],
                interpolation=cv2.INTER_NEAREST,
            ),
            A.Rotate(limit=20, p=0.5),
            ToTensorV2(transpose_mask=True),
        ]
    ),
    only_organ_present=config.only_organ_present,
    label_smoothing=config.label_smoothing,
    extra_organs=config.extra_organs,
)

validation_dataset = OrganSegmentationDataset(
    path_to_csv=config.csv_path,
    organ_name=config.organ_name,
    organ_mapping=organ_mapping[config.organ_name.lower()],
    CTs=config.validation_cts,
    normalization=config.normalization,
    augmentations=A.Compose(
        [
            A.Resize(
                config.image_size[0],
                config.image_size[1],
                interpolation=cv2.INTER_NEAREST,
            ),
            ToTensorV2(transpose_mask=True),
        ]
    ),
    extra_organs=config.extra_organs,
)

n_cpu = os.cpu_count()
train_dataloader = torch.utils.data.DataLoader(
    train_dataset,
    batch_size=config.get_batch_size(),
    num_workers=n_cpu,
    shuffle=True,
)
validation_dataloader = torch.utils.data.DataLoader(
    validation_dataset,
    batch_size=config.get_batch_size(),
    num_workers=n_cpu,
    shuffle=False,
)

if config.get_checkpoint() is None:
    model = SegmentationModel(config)
else:
    model = SegmentationModel.load_from_checkpoint(
        config.get_checkpoint(), config=config
    )

logger = pl.loggers.WandbLogger(
    project=config.project,
    name=config.run,
    config={
        "loss": config.loss_info.to_dict(),
        "optimizer": config.optimizer_info.to_dict(),
        "model": config.model_info.to_dict(),
```

```
            "epochs": config.get_num_epochs(),
            "batch_size": config.get_batch_size(),
            "random_seed": config.seed,
            "image_size": (config.image_size[0], config.image_size[1]),
            "normalization": config.normalization,
            "label_smoothing": config.label_smoothing,
            "froze_encoder": config.is_encoder_frozen,
        },
    )

    ckpt = pl.callbacks.ModelCheckpoint(
        config.save_dir,
        config.model_arch + config.encoder_name,
        mode="max",
        monitor="val dataset iou",
    )

    if config.use_amp:
        trainer = pl.Trainer(
            gpus=1,
            logger=logger,
            callbacks=ckpt,
            max_epochs=config.get_num_epochs(),
            precision=16,
            amp_backend="native",
        )
    else:
        trainer = pl.Trainer(
            gpus=1, logger=logger, callbacks=ckpt,
max_epochs=config.get_num_epochs()
        )

    trainer.fit(model, train_dataloader, validation_dataloader)

    wandb.finish()


def classification_pipeline(args):
    config = ClassificationTrainingConfig.from_json(args.config)

    set_determenistic(config.seed)

    train_dataset = OrganClassificationDataset(
        path_to_csv=config.csv_path,
        organ_name=config.organ_name,
        CTs=config.train_cts,
        normalization=config.normalization,
        augmentations=A.Compose(
            [
                A.Resize(
                    config.image_size[0],
                    config.image_size[1],
                    interpolation=cv2.INTER_NEAREST,
                ),
                A.Rotate(limit=20, p=0.5),
                ToTensorV2(),
            ]
        ),
        label_smoothing=config.label_smoothing,
    )

    validation_dataset = OrganClassificationDataset(
        path_to_csv=config.csv_path,
```

```
    organ_name=config.organ_name,
    CTs=config.validation_cts,
    normalization=config.normalization,
    augmentations=A.Compose(
        [
            A.Resize(
                config.image_size[0],
                config.image_size[1],
                interpolation=cv2.INTER_NEAREST,
            ),
            ToTensorV2(),
        ]
    ),
)

n_cpu = os.cpu_count()
train_dataloader = torch.utils.data.DataLoader(
    train_dataset,
    batch_size=config.get_batch_size(),
    num_workers=n_cpu,
    shuffle=True,
)
validation_dataloader = torch.utils.data.DataLoader(
    validation_dataset,
    batch_size=config.get_batch_size(),
    num_workers=n_cpu,
    shuffle=False,
)

if config.get_checkpoint() is None:
    model = create_classification_model(**config.model_params)
else:
    model = torch.load(config.get_checkpoint())

trainer = BCTrainer(model)

wandb.init(
    project=config.project,
    name=config.run,
    config={
        "loss": config.loss_info.to_dict(),
        "optimizer": config.optimizer_info.to_dict(),
        "model": config.model_info.to_dict(),
        "epochs": config.get_num_epochs(),
        "batch_size": config.get_batch_size(),
        "random_seed": config.seed,
        "image_size": (config.image_size[0], config.image_size[1]),
        "normalization": config.normalization,
        "label_smoothing": config.label_smoothing,
    },
)

loss = config.loss_fn(**config.loss_params)
optim = [config.optimizer(model.parameters(), **config.optimizer_params)]

trainer.train(
    train_dataloader,
    validation_dataloader,
    optim,
    loss,
    epochs=config.get_num_epochs(),
    save_directory=config.save_dir,
    monitor_metric=config.get_monitor_metric(),
```

```
        save_name=config.model_arch,
    )

    wandb.finish()


def main():
    parser = argparse.ArgumentParser()
    subparsers = parser.add_subparsers(title="Available commands")

    # segmentation subparser
    parser_segm = subparsers.add_parser("segm", help="Train segmentation model")
    parser_segm.add_argument(
        "--config", "-c", required=True, help="Path to config file"
    )
    parser_segm.set_defaults(func=segmentation_pipline)

    # clasification without using pytorch lightning
    parser_class = subparsers.add_parser("class", help="Train classification
model")
    parser_class.add_argument("--config", "-c", help="Path to config file")
    parser_class.set_defaults(func=classification_pipeline)

    args = parser.parse_args()
    args.func(args)


if __name__ == "__main__":
    main()
```

Файл model_config.py:

```
import json
from typing import Any, Dict, List, Optional, Tuple, Union


class ConfigException(Exception):
    pass


class Config:
    def __init__(self, config_dict: Dict[str, Any], model_name: str):
        self._dict = Config._clear_config_dict(config_dict, model_name)
        self._model_name = model_name

    @staticmethod
    def _clear_config_dict(
        config_dict: Dict[str, Any], model_name: str
    ) -> Dict[str, Any]:
        """
        Config dict contains info about different models, for example
classificators for different organs.
        Some of these models potentially be used by this model (segmentator can
use classificator to run only on slices
        where organ present, or segmentator, can use mask from other model as
it's input). Others won't be used by model
        This function just leaves in dictionary only info about models that this
model need.
        """
        model_config = config_dict.get(model_name, None)
        if model_config is None:
            raise ConfigException(
                f"Model with name {model_name} is not specified in config"
            )

        model_config = {model_name: model_config}

        if model_config[model_name]["type"] == "classificator":
            return model_config

        # segmentator can use classificator first
        classificator_name = model_config[model_name]["configuration"][
            "pipeline_params"
        ].get("classificator_name", None)
        if classificator_name is not None:
            classificator_dict = config_dict.get(classificator_name, None)
            # check if everything is correct
            if classificator_dict["type"] != "classificator":
                raise ConfigException(
                    f"{classificator_name} is used as classificator by
{model_name}"
                    + f'but its type {classificator_dict["type"]}'
                )
            if classificator_dict is None:
                raise ConfigException(
                    f"Model with name {classificator_name} used by {model_name}
is not specified in config"
                )
            model_config[classificator_name] = classificator_dict
```

```
        # also, segmentator can use outputs of other segmentation models
        additional_channels_models = model_config[model_name]["configuration"][
            "pipeline_params"
        ]["preprocessing"].get("additional_channels", None)

        if additional_channels_models is not None:
            for model in additional_channels_models:
                # here we can go to endless recursion if models use each other
                model_config = {
                    **model_config,
                    **Config._clear_config_dict(config_dict, model),
                }

        return model_config

    @property
    def config_dict(self):
        return self._dict

    @property
    def model_path(self) -> str:
        path = self._dict[self._model_name]["configuration"].get("model_path",
None)
        if path is None:
            raise ConfigException(f"model_path is not specified for
{self.model_name}")
        return path

    @property
    def model_type(self) -> str:
        type_ = self._dict[self._model_name].get("type", None)
        if type_ is None:
            raise ConfigException(f"type is not specified for
{self.model_name}")
        return type_

    @property
    def model_name(self) -> str:
        return self._model_name

    # preprocessing info
    @property
    def normalization(
        self,
    ) -> Union[Tuple[float, float], Tuple[List[float], List[float]], None]:
        return self._dict[self._model_name]["configuration"]["pipeline_params"][
            "preprocessing"
        ].get("normalization", None)

    @property
    def input_size(self) -> Tuple[int, int]:
        result =
self._dict[self.model_name]["configuration"]["pipeline_params"][
            "preprocessing"
        ].get("input_size", None)
        if result is None:
            raise ConfigException(f"Input size is not specified for
{self.model_name}")
        return result

    # slice postprocessing info
    @property
    def activation(self) -> Optional[str]:
```

```
        return self._dict[self._model_name]["configuration"]["pipeline_params"][
            "postprocessing"
        ].get("activation", None)

    @property
    def threshold(self) -> float:
        threshold =
self._dict[self._model_name]["configuration"]["pipeline_params"][
            "postprocessing"
        ].get("threshold", None)
        if threshold is None:
            raise ConfigException(f"threshold is not specified for
{self.model_name}")
        return threshold

    @classmethod
    def from_json(cls, file_name, model_name):
        with open(file_name) as f:
            config_dict = json.load(f)
        return cls(config_dict, model_name)


class ClassificatorConfig(Config):
    def __init__(self, config_dict: Dict[str, Any], model_name: str):
        super(ClassificatorConfig, self).__init__(config_dict, model_name)
        if self.model_type != "classificator":
            raise ConfigException(
                f"Tried create classification config with model, which type
{self.model_type}"
            )


class SegmentatorWithClassificatorConfig(Config):
    def __init__(self, config_dict: Dict[str, Any], model_name: str):
        super(SegmentatorWithClassificatorConfig, self).__init__(
            config_dict, model_name
        )
        if self.model_type != "segmentator":
            raise ConfigException(
                f"Tried create segmentation config with model, which type
{self.model_type}"
            )

    @property
    def classificator_name(self) -> Optional[str]:
        return
self._dict[self.model_name]["configuration"]["pipeline_params"].get(
            "classificator_name", None
        )

    @property
    def mapping(self) -> int:
        return self._dict[self.model_name]["configuration"]["pipeline_params"][
            "postprocessing_ct"
        ].get("mapping", 1)

    @property
    def leave_only_biggest_connected_component(self) -> bool:
        return self._dict[self.model_name]["configuration"]["pipeline_params"][
            "postprocessing"
        ].get("leave_only_biggest_connected_component", False)

    @property
```

```
    def additional_channels(self) -> Optional[List[str]]:
        return self._dict[self.model_name]["configuration"]["pipeline_params"][
            "preprocessing"
        ].get("additional_channels", None)

    @property
    def to_numpy(self) -> bool:
        return self._dict[self.model_name]["configuration"]["pipeline_params"][
            "postprocessing_ct"
        ].get("to_numpy", False)
```

Файл processing.py:

```
from copy import deepcopy
from typing import Optional, Sequence, Union

import cc3d
import numpy as np
import torch
from monai.transforms import KeepLargestConnectedComponent


def min_max_scale(
    input_: Union[torch.Tensor, np.ndarray],
    min_: Union[Sequence[Union[int, float]], Union[int, float]],
    max_: Union[Sequence[Union[int, float]], Union[int, float]],
    ignore_channels: Optional[Sequence[int]] = None,
) -> Union[torch.Tensor, np.ndarray]:
    """
    Perform min_max scaling:
    Every input element greater than max_ replaced by max_ and every element
greater than min_ replaced by min_ and then
    (input - min_) / (max_ - min_)
    Args:
        input_: Union[torch.Tensor, np.ndarray]
            BCHW or CHW image
        min_: Union[Sequence[float], float]
            If float same value used for all channels, else it should be length
of:  C - length of ignore_channels
        max_: Union[Sequence[float], float]
            If float same value used for all channels, else it should be length
of:  C - length of ignore_channels
        ignore_channels: Optional[Sequence[int]], default: None
            Indexes of channels to ignore. If scaling is performed over all
channels
    Returns: Union[torch.Tensor, np.ndarray]
        Scaled image
    """

    # check if all arguments are correct
    ndim = input_.ndim
    if ndim < 3 or ndim > 4:
        raise ValueError(
            f"Expected BCHW or CHW image, but got {ndim} dimensional input"
        )

    n_channels = input_.shape[1] if ndim == 4 else input_.shape[0]
    if isinstance(min_, float) or isinstance(min_, int):
        min_ = [min_ for i in range(n_channels)]
    if isinstance(max_, float) or isinstance(max_, int):
        max_ = [max_ for i in range(n_channels)]
    if len(max_) != n_channels:
        raise ValueError(
```

```
            f"Expected len(max_) to be same as number of image channels. But got
len(max_):{len(max_)}, channels:{n_channels}"
        )
    if len(min_) != n_channels:
        raise ValueError(
            f"Expected len(min_) to be same as number of image channels. But got
len(max_):{len(min_)}, channels:{n_channels}"
        )

    result = deepcopy(input_)

    for channel in range(n_channels):
        if ignore_channels is not None and channel in ignore_channels:
            continue

        if ndim == 4:
            channel_result = input_[:, channel, :, :]
            channel_result[channel_result < min_[channel]] = min_[channel]
            channel_result[channel_result > max_[channel]] = max_[channel]
            channel_result = (channel_result - min_[channel]) / (
                max_[channel] - min_[channel]
            )
            result[:, channel, :, :] = channel_result
        else:
            channel_result = input_[channel, :, :]
            channel_result[channel_result < min_[channel]] = min_[channel]
            channel_result[channel_result > max_[channel]] = max_[channel]
            channel_result = (channel_result - min_[channel]) / (
                max_[channel] - min_[channel]
            )
            result[channel, :, :] = channel_result

    return result


def remove_artifacts(input_: np.ndarray):
    """
    Remove artifacts of segmentation caused by wrong classifier predictions by
leaving the biggest connected component
    """
    input_ = torch.Tensor(input_)
    input_ = input_.unsqueeze(0)
    # labels to apply for, assume that all labels and positive and on 1st
position one will be 0
    labels = torch.unique(input_)[1:]
    result = KeepLargestConnectedComponent(labels)(input_)
    result = np.array(result)
    return result


def merge_ct_predictions(segmentations: Sequence[Union[np.ndarray,
torch.Tensor]]):
    """
    Args:
        segmentations: segmentations to merge, in order of their priority
        It means that if we have same pixel marked as true part of organ in
segmentations[0] and segmentations[1],
        in result pixel will be in class represented by segmentations[0]

    Returns:

    """
    if len(segmentations) == 0:
```

```
        raise ValueError("Expected at least one segmentation")

    for i, _ in enumerate(segmentations[1:]):
        if segmentations[0].shape != segmentations[i].shape:
            raise ValueError("Expected all segmentations to be same shape")

    result_type = np.array if isinstance(segmentations[0], np.ndarray) else
torch.Tensor

    result = torch.zeros(segmentations[0].shape)

    for segmentation in reversed(segmentations):
        segmentation = torch.Tensor(segmentation)
        result[segmentation != 0] = segmentation[segmentation != 0]

    return result_type(result)
```

Файл inference_models.py:

```
from abc import ABC, abstractmethod
from typing import Callable, List, Union

import numpy as np
import torch
from model_config import ClassificatorConfig, Config,
SegmentatorWithClassificatorConfig
from processing import min_max_scale, remove_artifacts
from torchvision.transforms import InterpolationMode, Resize

# TODO: add cuda support, cause now it takes quite a long time for pipeline for
liver and gallbladder(more than 2 minutes)


class AbstractModel(ABC):
    @abstractmethod
    def __init__(self, config: Config):
        self.preprocessing = AbstractModel._build_preprocessing_function(config)
        self.postprocessing =
AbstractModel._build_postprocessing_function(config)

    @abstractmethod
    def predict_slice(self, ct_slice):
        pass

    @staticmethod
    def _build_preprocessing_function(config: Config) -> Callable:
        def preprocessing(image):
            model_input = torch.Tensor(image)
            if model_input.ndim > 4:
                raise ValueError(
                    f"Input should be maximum 4D(BCHW), 3D(CHW) or 2D (HW), but
got {image.ndim}D input "
                )
            if model_input.ndim == 2:
                model_input = model_input.unsqueeze(0)
            if model_input.ndim == 3:
                model_input = model_input.unsqueeze(0)

            norm = config.normalization
            if norm is not None:  # normalization need to be performed
                if isinstance(norm[0], float) or isinstance(norm[0], int):
                    min_ = norm[0]
                    max_ = norm[1]
```

```
                else:  # using different normalization for different channels
                    min_ = [n[0] for n in norm]
                    max_ = [n[1] for n in norm]
                model_input = min_max_scale(model_input, min_, max_)

            model_input = Resize(
                config.input_size, interpolation=InterpolationMode.NEAREST
            )(model_input)
            return model_input

        return preprocessing

    @staticmethod
    def _build_postprocessing_function(config: Config) -> Callable:
        def postprocessing(model_output: torch.Tensor):
            if config.activation is not None:
                if config.activation == "sigmoid":
                    model_output = model_output.sigmoid()
                elif config.activation == "softmax":
                    model_output = model_output.softmax()

            model_output = torch.where(model_output > config.threshold, 1, 0)

            return model_output

        return postprocessing


class ClassificationModel(AbstractModel):
    def __init__(self, config: ClassificatorConfig):
        super(ClassificationModel, self).__init__(config)
        self.model = torch.load(config.model_path)

    def predict_slice(self, ct_slice):
        model_input = self.preprocessing(ct_slice)
        model_output = self.model(model_input)
        result = self.postprocessing(model_output)
        return result


class SegmentationWithClassificationModel(AbstractModel):
    def __init__(self, config: SegmentatorWithClassificatorConfig):
        super(SegmentationWithClassificationModel, self).__init__(config)
        self.postprocessing_ct = (

SegmentationWithClassificationModel._build_ct_postprocessing_function(
                config
            )
        )
        # build main model
        self.model = torch.load(config.model_path)
        # build classificator model
        self.classificator = (
            ClassificationModel(
                ClassificatorConfig(config.config_dict,
config.classificator_name)
            )
            if config.classificator_name is not None
            else None
        )
        # build models for additional channels
        if config.additional_channels is None:
            self.additional_channels_models = None
```

```
        else:
            self.additional_channels_models = [
                SegmentationWithClassificationModel(
                    SegmentatorWithClassificatorConfig(config.config_dict,
model_name)
                )
                for model_name in config.additional_channels
            ]

    @staticmethod
    def _build_ct_postprocessing_function(
        config: SegmentatorWithClassificatorConfig,
    ) -> Callable[[List[torch.Tensor]], Union[torch.Tensor, np.ndarray]]:
        def postprocess(ct: [List[torch.Tensor]]) -> Union[np.ndarray,
torch.Tensor]:
            # 1x1x512x512
            ct = torch.cat(ct)
            # n_slices x 1 x 512 x 512
            ct = ct.squeeze()
            # n_slices x 512 x 512
            ct = ct.permute(1, 2, 0)
            # 512 x 512 x n_slices
            ct = ct.detach().cpu().numpy()

            if config.leave_only_biggest_connected_component:
                ct = remove_artifacts(ct)

            ct = np.where(ct == 1, config.mapping, 0)

            if not config.to_numpy:
                ct = torch.Tensor(ct)
            return ct

        return postprocess

    def predict_slice(self, ct_slice):
        ct_slice = torch.Tensor(ct_slice)
        model_input = [
            ct_slice,
        ]

        if (self.classificator is not None) and (
            not self.classificator.predict_slice(ct_slice)
        ):
            result = torch.zeros(ct_slice.shape)
            if result.ndim == 2:
                result = result.unsqueeze(0)
            if result.ndim == 3:
                result = result.unsqueeze(0)
        else:

            if self.additional_channels_models is not None:
                for model in self.additional_channels_models:
                    model_out = model.predict_slice(ct_slice)
                    model_input.append(model_out.reshape(ct_slice.shape))

            model_input = torch.stack(model_input)

            model_input = self.preprocessing(model_input)

            result = self.model(model_input)
            result = self.postprocessing(result)
```

```
        result = Resize(ct_slice.shape[-2:],
interpolation=InterpolationMode.NEAREST)(
            result
        )
        return result

    def predict_ct(self, ct):
        res = []
        for i in range(ct.shape[2]):
            res.append(self.predict_slice(ct[:, :, i]))
        return self.postprocessing_ct(res)
```

Файл utils.py:

```
import gzip
from io import BytesIO

import nibabel as nib
from werkzeug.datastructures import FileStorage


def file_storage_nii_to_nifti(file: FileStorage) -> nib.Nifti1Image:
    fh = nib.FileHolder(fileobj=BytesIO(file.read()))
    image = nib.Nifti1Image.from_file_map({'header': fh, 'image': fh})
    return image


def file_storage_nii_gz_to_nifti(file: FileStorage) -> nib.Nifti1Image:
    gzf = gzip.GzipFile('', 'rb', 9, file)
    rr = gzf.read()
    bb = BytesIO(rr)
    fh = nib.FileHolder(fileobj=bb)
    img = nib.Nifti1Image.from_file_map({'header': fh, 'image': fh})
    return img


def file_storage_to_nifti(file: FileStorage) -> nib.Nifti1Image:
    ext = file.filename.split('.')[-1]
    if not is_file_type_correct(file):
        raise ValueError(
            f'Incorrect file type, expected .nii.gz or .nii; but got {ext}'
        )
    if ext == 'nii':
        return file_storage_nii_to_nifti(file)
    else:
        return file_storage_nii_gz_to_nifti(file)


def is_file_type_correct(file: FileStorage) -> bool:
    filename = file.filename
    if filename.split('.')[-1] == 'nii':
        return True
    if filename.split('.')[-1] == 'gz' and filename.split('.')[-2] == 'nii':
        return True
    return False


def nifti_to_binary_io(img: nib.Nifti1Image) -> BytesIO:
    bio = BytesIO()
    file_map = img.make_file_map({'image': bio, 'header': bio})
    img.to_file_map(file_map)
    data = BytesIO(gzip.compress(bio.getvalue()))
    return data
```

Файл main.py:

```
import json

import nibabel as nib
from flask import Flask, request, send_file
from utils import file_storage_to_nifti, nifti_to_binary_io

from inference import (
    ConfigException,
    SegmentationWithClassificationModel,
    SegmentatorWithClassificatorConfig,
    merge_ct_predictions,
)

MODELS_CONFIG_PATH = "/home/raman/Work/ct/inference/models_config.json"
DEBUG = True

app = Flask(__name__)


@app.route('/predict/', methods=["POST"])
def predict():
    if request.method == "POST":
        if 'file' not in request.files:
            return 'You must provide file', 400

        file = request.files['file']

        try:
            ct = file_storage_to_nifti(file)
        except ValueError as e:
            return str(e), 400

        ct_data = ct.get_fdata()
        models = request.form.get('models', None)
        if models is None:
            return "Models are not provided", 400
        models = json.loads(models)

        predictions = []
        for model_name in models:
            try:
                gallbladder_config =
SegmentatorWithClassificatorConfig.from_json(
                    MODELS_CONFIG_PATH, model_name
                )
                gallbladder_model = SegmentationWithClassificationModel(
                    gallbladder_config
                )
                predictions.append(gallbladder_model.predict_ct(ct_data))
            except ConfigException as e:
                return str(e), 400

        if len(predictions) == 0:
            return "No segmentations were requested", 400

        segm_result = merge_ct_predictions(predictions)
        res = nib.Nifti1Image(segm_result, ct.affine)

        return send_file(nifti_to_binary_io(res),
download_name='segmentation.nii.gz')
```

```
if __name__ == "__main__":
    app.run()
```