МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное   
учреждение высшего образования   
«Южный федеральный университет»

Институт математики, механики   
и компьютерных наук им. И. И. Воровича

Кафедра теории упругости

Пандов Вячеслав Дмитриевич

ОПРЕДЕЛЕНИЕ И СЕГМЕНТАЦИЯ ТРЕЩИН НА ПОВЕРХНОСТИ ПРИ ПОМОЩИ СВЁРТОЧНОЙ НЕЙРОННОЙ СЕТИ

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА  
по направлению подготовки

01.03.02— Прикладная математика и информатика

**Научный руководитель** –   
доц. Карякин Михаил Игорьевич

Допущено к защите:  
заведующий кафедрой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Ватульян А.О.

Ростов-на-Дону – 2021

Оглавление

[Введение 2](#_Toc71241979)

[Постановка задачи 4](#_Toc71241980)

[Глава 1. Подготовка данных 8](#_Toc71241981)

[1.1 Описание набора данных 8](#_Toc71241982)

[Глава 2. Применение глубоких нейронных сетей 12](#_Toc71241983)

[2.1 Свёрточная нейронная сеть 12](#_Toc71241984)

[Глава 3. Эксперименты и результаты 19](#_Toc71241985)

[Выводы 20](#_Toc71241986)

[Заключение 21](#_Toc71241987)

[Список литературы 22](#_Toc71241988)

# Введение

Любой материал имеет свойство изнашиваться и впоследствии разрушаться. В результате на его поверхности появляются дефекты в виде трещин. Это может создавать разного рода проблемы и даже чрезвычайно опасные ситуации, скорость и методы решения которых в первую очередь зависят от сферы деятельности. Среди таких сфер можно выделить следующие.

1. Реконструкция фасадов зданий.

Для выявления причины появления трещин используют так называемые «маяки» (щелемеры), которые крепятся непосредственно в области разрушения и помогают отслеживать динамику его развития. Однако такие приспособления имеют ряд недостатков. «Простые» версии ограничены в своей информативности, а современные электронные аналоги часто подвержены краже и вандализму.

1. Краш-тесты.

Различные материалы подвергают критическим нагрузкам, вследствие чего также появляются трещины.

1. Отбраковка продукции на производстве.

При производстве газобетонных блоков, древесины и т. п. очень важно вовремя исключать из конвейера изделия с дефектами

Методы машинного обучения позволяют улучшить эффективность проведения описанных выше мероприятий используя системы фото и видеонаблюдение. Анализируя поток изображений с камеры, можно определять наличие дефектов и выделять сегменты трещин. Такой информации достаточно для расчетов всех необходимых метрических характеристик трещин.

В компьютерном зрении задача сегментации трещин на поверхности относится к семейству задач «Semantic Image Segmentation» [1]. Цель такого рода задач заключается в попиксельной локализации целевого объекта на изображении. Для каждой отдельно взятой картинки результатом локализации будет выступать так называемая «бинарная маска» — одноканальное бинарное изображение (Рисунок 1). Пиксели такой маски обычно обозначают 1, если на соответствующем пикселе входного изображения присутствует искомый целевой объект, и обозначают 0, если на соответствующем пикселе объект отсутствует.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 1. Пример сегментации трещин по картинке

Однако, для получения однозначного ответа на вопрос «Есть ли трещина?», необходимо реализовать дополнительный алгоритм, анализирующий полученную бинарную маску. Такой подход требует дополнительных издержек как вычислительных, так и временных, которые являются излишеством в следующих ситуациях.

1. Получение ответа на вопрос «Есть ли трещина?» является более приоритетным, чем получение бинарной маски. Возможно, в отрицательном случае такая маска вовсе не нужна.
2. Анализируемый поток информации по большей части не содержит «трещин», и дополнительного анализа таких данных, в том числе и построение маски, вовсе не требуется.

Дальше будет рассмотрен подход, позволяющий определить наличие трещины на картинке, не требующий построения бинарной маски, делая это самое построение опциональным.

# Постановка задачи

Целью данной работы является описание, реализация и обучение модели для определения и сегментации трещин на поверхности по изображению.

**Сегментация.** Для решения задачи сегментации в глубоком обучении используют различные архитектуры генеративных нейронных сетей, относящихся семейству «Variance Autoencoders (VAEs)» [2]. Такие архитектуры состоят из трёх основных компонент, которые взаимодействуют между собой следующим образом:

1. Кодировщик. Исходный набор картинок подается в энкодер. Задача энкодера определенным образом «сжать» информацию в так называемое «сжатое представление». Такое сжатие происходит путем многократного пропуска информации через последовательность обученных блоков.
2. Сжатое представление. На этом уровне никакого сжатия не происходит. Здесь сжатая информация дополнительно проходит через обученные блоки. Цель этого блока – обработать сжатое представление и подготовить его к обратной реконструкции, которой занимается декодер.
3. Декодер. Завершающий слой, реконструирующий сжатое представление к необходимому виду. Реконструкция происходит за счет обратных свёрточных преобразований.

Таким образом, имея входные данные на выходе необходимо получить представление вида следующим образом (Рисунок 2.):

где – блок кодировщика, – сжатое представление, – блок декодера.



Рисунок 2. Принцип работы архитектур семейства «VAEs».

**Классификация.** Задачу определения наличия трещины можно сформулировать как задачу классификации. С этой точки зрения класс 1 будет обозначать присутствие трещины на картинке, а класс 0 её отсутствие. Тогда рассматривая все те же входные данные из сжатого представления необходимо получить бинарное представление :

где – вероятность того, что на картинке присутствует трещина, – некоторый фиксированный порог строгости классификации.

# Глава 1. Подготовка данных

# 1.1 Описание набора данных

Для обучения и тестирования модели был взят набор данных под названием «Crack Segmentation» [\*]. Это большая база, состоящая из 11,298 пар изображений размера 448x448 пикселей. В каждой такой паре первая картинка представляет из себя цветную фотографию некоторой поверхности, а вторая картинка — это бинарная маска, в которой все пиксели, содержащие трещины из первой картинке, окрашены белым, а все остальные пиксели черным (Рисунок 3.).



Рисунок 3. Представление данных.

Техника «обучение с учителем» [\*] – когда модель обучается на заранее размеченном наборе данных, предполагает разделение всего набора данных на 2 части – тренировочную и проверочную. На тренировочной подвыборке модель изменяет своё состояние, то есть обучается, а на проверочной нет. Такой подход гарантирует честность и объективность подсчета метрик в процессе обучения. Описанный выше набор данных изначально разбит на 2 соответствующих каталога, из которых 9,603 экземпляров относятся к тренировочной подвыборке и 1,695 к проверочной.

1.2 Аугментация данных

Увеличение исходного набора данных оказывает положительное влияние на обучение модели. Это не только помогает избежать проблемы переобучения, но и даёт возможность обучаемой сети познакомиться с большим набором уникальных ситуаций и выявить больше закономерностей. Процесс искусственного увеличения набора данных называется «аугментацией». Такой эффект достигается путем пропуска данных через различные, в зачастую случайные, операции обработки изображений. Рассмотрим примеры таких операций:

1. Горизонтальное отображение (Рисунок 4). Позволяет из 1 уникального экземпляра изображения сделать 2 уникальных экземпляра.

Изображение выглядит как земля, внешний, строительный материал, камень

Автоматически созданное описание Изображение выглядит как земля, внешний, строительный материал, камень

Автоматически созданное описание

Рисунок 4. Горизонтальное отображение.

1. Вертикальное отображение (Рисунок 5). Аналогично горизонтальному. Используя оба вида отображения, можно из 1 уникального экземпляра сделать 4 уникальных.

Изображение выглядит как земля, внешний, строительный материал, камень

Автоматически созданное описание Изображение выглядит как земля, внешний, строительный материал, камень

Автоматически созданное описание

Рисунок 5. Вертикальное отображение.

1. Поворот на 0, 90, 180 и 270 градусов (Рисунок 6). Каждые 90 градусов дает новые уникальные экземпляры. В совокупности со всеми видами отображение можно из 1 уникального экземпляра сделать 16 уникальных.

Изображение выглядит как земля, внешний, строительный материал, камень

Автоматически созданное описание Изображение выглядит как земля, внешний, строительный материал, камень

Автоматически созданное описание Изображение выглядит как земля, внешний, строительный материал, камень

Автоматически созданное описание Изображение выглядит как земля, внешний, строительный материал, камень

Автоматически созданное описание

Рисунок 6. Поворот на 0, 90, 180 и 270 градусов.

Чередование случайным образом описанных операций позволяет увеличить исходный набор данных в 16 раз.

В рамках исследования размер картинок 448x448 пикселей является избыточным. Чем больше входная информация, тем мощнее должна быть модель, которая эту информацию будет обрабатывать. Мощность модели повышается за счет увеличения количества обучаемых параметров. Это значительно увеличивает требования к минимальному объему видеопамяти. Поэтому в процессе аугментации данных, каждая «аугментированная» картинка также будет разделена на 5 равных частей следующим образом (Рисунок 7).

Так как на части картинок трещины занимает меньше половины площади, операция разделения создаст изображения с отсутствием трещин, что даст необходимый набор данных для последующего решения задачи классификации.



Рисунок 7. Принцип разделения картинки на 5 частей.

# Глава 2. Применение глубоких нейронных сетей

# Свёрточная нейронная сеть

# Свёрточная нейронная сеть – архитектура искусственных нейронных сетей, которую впервые предложил Ян Лекун в 1998 году [11]. В общем случае такая архитектура устроена аналогичным образом, что и архитектура простой полносвязной нейронной сети: слой извлечения признаков, слой нормализации и слой активации. Полносвязный слой извлечения признаков, он же перцептрон, определяется следующим образом:

где – входной вектор данных размерности , – выходной вектор размерности , – матрица весов размерности , – вектор смещения размерности . Обработка двумерных данных обычными полносвязными (линейными) нейронными сетями имеет два критических недостатка. Рассмотрим их на примере задачи классификации на наборе данных MNIST (Рисунок 8.).



Рисунок 8. Набор данных MNIST [10].

Представленная база состоит из 10 наборов (классов) изображений рукописных цифр. Картинки представлены в одноканальном виде и имеют разрешение 28x28 пикселей. Перед подачей такого изображения на вход полносвязной нейронной сети его необходимо представить в одномерном виде, то есть растянуть в вектор длинной 284. При обработке такого вектора возникают следующие сложности:

1. Полносвязная нейронная сеть требует слишком большую входную размерность. Например, для обработки изображения размера 224х224 пикселей, необходимо 50,176 входных нейронов.
2. Нарушаются пространственные отношения на картинке. Полносвязная нейронная сеть лишена способности выявлять двумерные паттерны. Это приводит к дополнительным сложностям распознавания, когда объект, не изменив своих внешних характеристик, смещается в другие области изображения или изменяет свой масштаб.

Архитектура свёрточной нейронной сети решает описанные выше проблемы путем использования свёрточных слоев вместо полносвязных для извлечения признаков. Последовательность таких слоев формируют блоки архитектуры свёрточной нейронной сети. А последовательность таких блоков формируют непосредственно саму архитектуру. Рассмотрим принципы работы описанных слоев и пример основанной на них архитектуры для решения задачи классификации.

2.1.1 Свёрточный слой

Свёрточный слой определяется следующим образом:

где – тензор весов, – вектор смещений, и – входной и выходной тензоры, и – размеры ядра (фильтра) свёртки, и – высота и ширина входного изображения, а и – шаг по вертикали и горизонтали, а и – количество каналов входного и выходного изображения соответственно.



Рисунок 9. Демонстрация работы свёртки.

Веса фильтров свёртки общие – это позволяет использовать относительно малое количество параметров, при этом выявлять паттерны, не зависящие от топологии изображения. В общем случае, вход и выход свёрточного слоя являются трёхмерными тензорами, где первые две размерности – это высота и ширина изображения, а третья размерность – это количество каналов, они же карты признаков. Такие карты позволяют использовать ядра свёртки для поиска различных паттернов (Рисунок 9.). Количество каналов свёрточного слоя – это параметр аналогичный количеству нейронов полносвязного слоя.

Также перед применением фильтра по краям карты признаков в зачастую добавляют нулевые контуры, или «padding». Такой прием полезен в следующих ситуациях:

1. Когда важные признаки могут находиться на краях изображениях или для коррекции размерности выходной карты признаков.
2. Когда необходимо обработать изображение свёрткой, при этом оставляя размерность выходной карты признаков такой же, как и входное изображение.



Рисунок 10. Добавление «padding».

2.1.2 Слой нормализации

При обучении глубоких нейронных сетей возникает проблема, когда распределение входных признаков каждого слоя меняется в связи с изменениями параметров предыдущего слоя во время обучения. В результате сам процесс обучения замедляется. Для решения данной проблемы используется дополнительный слой нормализации признаков [12] с обучаемыми параметрами, как часть архитектуры нейронной сети.

Также в процессе обучения на большом наборе данных возникает проблема, когда невозможно загрузить все данные на видеокарту в силу ограничений видеопамяти. Тогда прямо в процессе обучения, входные данные подаются модели партиями, которые принято называть «батчами». Слой нормализации применяется непосредственно к таким батчам, накапливая необходимую статистику для обобщения по всей выборке. Таким образом, слой батч нормализации связывает входные и выходные группы тензоров и следующим образом:

где индекс – обозначает -ый батч, и – накапливаемые статистики выборочного среднего и стандартного отклонения, посчитанные по всей подвыборке , и — обучаемые вектора параметров, используемые для масштабирования и смещения признаков к требуемому распределению, а — очень малое число-константа, необходимое для предотвращения деления на ноль.

2.1.3 Слой активации

В качестве функции активации в архитектурах свёрточных нейронных сетей можно использовать различные нелинейные функции. Однако в подавляющем большинстве случаев используют функцию ректификации (Рисунок 11.) под названием «ReLU» [13]:



Рисунок 11. График функции активации ReLU.

С точки зрения обратного распространения ошибки [14], производная данной функции активации имеет следующий физический смысл: ошибка распространяется неизменно по тем путям, где сигнал положителен и не распространяется вовсе, где сигнал отрицателен:

2.1.4 Слой субдискретизации

Слой субдискретизации – это нелинейное преобразование, использующее функции максимума (MAX), минимального (MIN), или среднего (AVG), действующие на группы компонентов тензора данных. Такой слой ещё называют слой пулинга, и используют его для нелинейного уплотнения карты признаков. Слой пулинга с функцией максимума определяется следующим образом.

где – входной тензор размера , – выходной тензор, – окно пулинга, а и – величины шага по вертикали и горизонтали соответственно. Такой слой субдискретизации уменьшает размерность выходных признаков, оставляя лишь наиболее «важные» в смысле максимума (Рисунок 12.).



Рисунок 12. Демонстрация работы макс пулинга.

2.2 Архитектура свёрточной нейронной сети для классификации

В качестве примера рассмотрим архитектуру свёрточной нейронной сети под названием VGG-13 [\*]. Представленная нейронная сеть демонстрирует эффективность свёрточных слоёв в решении задачи классификации на наборе данных ImageNet [\*]. Данную нейронную сеть можно разделить на две ключевые компоненты:

1. Кодировщик, состоящий из последовательности свёрточных блоков с нормализацией, функцией активации и субдискретизацией между блоками. На данном этапе из картинки выделяются ключевые признаки в виде сжатого представления, которые затем отправляются в полносвязный слой.
2. Полносвязный слой, анализирующий ключевые признаки из сжатого представления и интерпретирующий их в более низкоуровневые для конечной классификации.



Рисунок 13. Архитектура VGG-13.

Эффективность данного подхода показана в статье «Impact of Fully Connected Layers on Performance of Convolutional Neural Networks for Image Classification». Авторы провели эксперименты на различных архитектурах свёрточных нейронных сетей с различными наборами данных. Этот же принцип можно использовать и в задаче, рассматриваемой в данной работе. Для этого необходимо модифицировать описанную ранее архитектуру «VAEs», добавив сразу после кодировщика дополнительные полносвязные слои для классификации изображения на классы «трещина присутствует» и «трещина отсутствует». Данный подход позволит узнать есть ли трещина на картинке, не реконструируя сжатое представление декодером, сделав саму реконструкцию опциональной.

2.3 Архитектура генеративной свёрточной нейронной сети для сегментации

В качестве примера рассмотрим архитектуру U-Net [15] семейства «VAEs», которую разработали в 2015 году для сегментации биомедицинских изображений (Рисунок 14). Представленная нейронная сеть состоит из ранее описанных трёх компонент: кодировщик, сжатое представление и декодер.

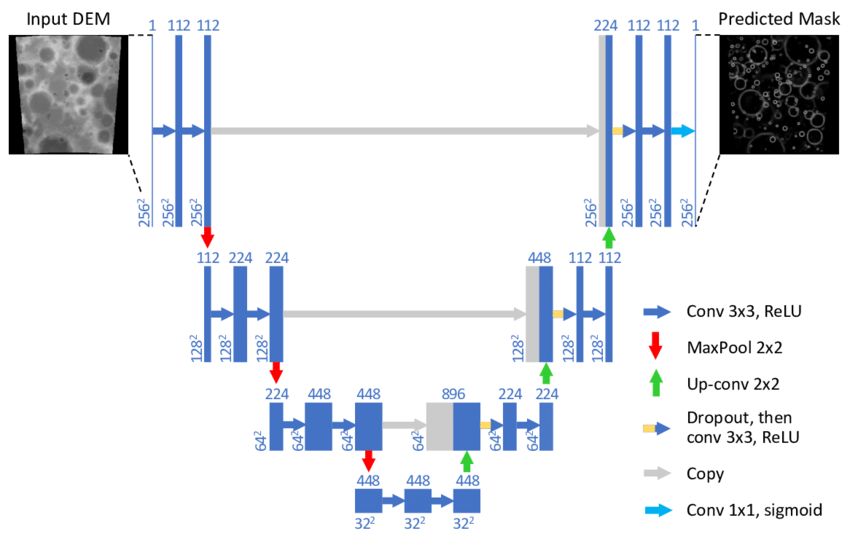


Рисунок 14. Архитектура U-Net.

В данной архитектуре кодировщик использует блоки кодировщика ранее упомянутой архитектуры VGG-13. Операцию «реконструкции» сжатого представления в бинарную маску выполняют слои обратной (транспонированной) свёртки.

2.3.1 Слой транспонированной свёртки.

Данная операция использует описанный ранее «padding», добавляя более широкий контур. Дополнительно можно увеличить шаг ядра по вертикали и горизонтали – и , чтобы заполнить «нулями» пространство между пикселями карты признаков. В результате выходная карта признаков получается большего размера, чем входная (Рисунок 15.).

Изображение выглядит как визитка

Автоматически созданное описание

Рисунок 15. Транспонированная свёртка с шагом 1 и с шагом 2.

Также при реконструкции используется подход под названием «skip connection». Визуально блоки архитектуры разделены на уровни таким образом, чтобы каждому блоку энкодера на каждом уровне был соответствующий блок декодера той же размерности. Смысл подхода заключается в соединении выходного тензора блока энкодера с соответствующим входным тензором блока декодера. После чего объединенная информация обрабатывается свёрточными слоями. Такой подход объединяет высокоуровневые признаки с более низкоуровневыми, что впоследствии позволяет с лучшей точностью реконструировать изображение.

* 1. Критерии оптимизации

Добавив после кодировщика U-Net полносвязные нейронные слои, была получена архитектура нейронной сети, имеющая один вход и два выхода. Таким образом полученная модель решает две задачи одновременно: сегментацию и классификацию. Для каждой из задач необходимо определить соответствующие критерии оптимизации, интерпретируемые как ошибка, которую необходимо минимизировать в процессе обучения. Однако одновременный поиск минимума двух функций от разных параметров для нейронной сети с двумя различными выходами является задачей нетривиальной. В обход данной проблемы было решено обучать описанную модель в два этапа:

1. В первую очередь решить задачу сегментации. Во время обучения проблема классификации не рассматривается. Необходимо обучить сеть генерировать бинарные маски трещин.
2. Затем решить задачу классификации. На этом этапе задача сегментации не рассматривается. Перед обучением, к обученному на прошлом этапе кодировщику добавляются полносвязные нейронные слои. И непосредственно в самом обучении обучаются только добавочные слои, не изменяя состояние уже обученного кодировщика. Таким образом формулируется гипотеза о том, что полносвязные слои смогут интерпретировать сжатое представление из кодировщика в целевой класс.

Для оптимизации критериев была использована модификация градиентного спуска – стохастический градиентный спуск с импульсом [\*].

2.4.1 Критерий для задачи классификации

Для вероятностной интерпретации выходов нейронной сети классификатора выходы такой сети должны гарантированно принадлежать диапазону . Для этого выходы сети дополнительно обрабатываются сигмоидной функцией активации:

В качестве критерия ошибки рассмотрим для начала функцию квадрата ошибки:

где – это выход из нейронной сети, а – это истинное значение класса. Затем найдем производную от этой функции по выходу сети – аргументу :

У получившейся производной есть большая проблема, которая называется «паралич сигмоидной нейронной сети», которая заключается в следующем: даже если значение такой функции ошибки очень большое, а следовательно, достаточно большое значение , все равно возможна ситуация, когда производная будет «равна» нулю. Из-за чего обучать такую нейронную сеть при помощи градиентного спуска становиться невозможным. Проблему можно увидеть в следующем уравнении:

Поэтому использовать квадратичную ошибку в качестве функции потерь вместе с сигмоидной функцией активации на выходе из сети очень плохая идея.

В данной ситуации в качестве критерия используют функцию под названием бинарная кросс-энтропия, которая формулируется следующим образом:

Аргумент логарифма всегда принадлежит положительному диапазону , следовательно никогда не возникает ситуации, когда в качестве аргумента логарифма выступают нулевые или отрицательные числа. Рассмотрим производную данной функции все по тому же аргументу :

Таким образом производная бинарной кросс энтропии по выходу сети не содержит никаких домножений на или , вследствие чего не происходит ранее описанного зануления производной. Поэтому для задачи бинарной классификации в качестве критерия можно использовать описанную функцию. Для подсчета ошибки по всем экземплярам будем брать среднее значение ошибки по всем экземплярам данных:

,

где – это выход сети для -го экземпляра, а – это истинное значение класса для -го экземпляра.

2.4.2 Критерий для задачи сегментации

Для формулирования критерия для задачи сегментации рассмотрим меру под названием коэффициент Сёренсена [\*], которая по своей сути является операцией над множествами:

Данный коэффициент позволяет количественно описать меру сходства между двумя множествами и , и равен 1, если множества полностью совпадают и 0 если не пересекаются. Данную аналогию можно провести и для двух бинарных масок, рассматривая их в виде двух множеств (Рисунок 16.).



Рисунок 16. Применение метрики Dice к бинарным маскам.

Тогда операции над множествами заменяются на арифметические над пикселями:

где и – соответственно ширина и высота изображения бинарной маски, – целевое значение пикселя маски, – предсказанное значение пикселя маски. Здесь в числитель и знаменатель также была добавлена 1, так как значение функции не определено, если все .

Как и в случае с бинарной классификацией, в качестве выхода нейронной сети для каждого пикселя также предсказывается вероятность того, что на данном пикселе присутствует трещина. Для этого необходимо аналогично обрабатывать выходы нейронной сети сигмоидной функцией активации:

Исходя из области значений данной функции можно сформулировать критерий ошибки:

Однако данная функция не является выпуклой [\*], что добавляет сложности для оптимизации. В качестве решения автор статьи предлагает дополнить критерий раннее описанной функцией бинарной кросс энтропии:

где параметр подбирается эмпирическим путем. Таким образом каждый пиксель маски будет минимизироваться сразу по двум критериям.

Глава 3. Эксперименты и результаты

Для реализации описанных моделей необходимо определиться с набором инструментов. В первую очередь нужно выбрать такой язык программирования, на котором вся разработка сведется к сбору, систематизации и анализу данных. После чего на основе полученной информации можно будет легко создавать и настраивать алгоритмы для глубокого обучения. Такой язык также должен обладать библиотеками и фреймворками, которые позволяют производить математические расчеты на видеокартах, активно развиваются и поддерживаются, а также имеют крупное сообществом разработчиков. Среди популярных языков для анализа данных можно выделить MATLAB, R и Python. Однако всем этим критериям удовлетворяет язык Python. Из подходящих фреймворков можно выделить два наиболее крупных и известных: PyTorch и TensorFlow. TensorFlow, в отличии от PyTorch, имеет более простые инструменты для создания и обучения модели. Однако такая простота усложняет работу в случаях, когда нужно совершить более низкоуровневые настройки моделей. PyTorch предоставляет более низкоуровневый интерфейс для настройки, что повышает порог входа для изучение фреймворка в начале и облегчает работу в дальнейшем за счет гибкости, которую дает низкоуровневость.

Оба рассмотренных фреймворка дают возможность создавать модели глубоких нейронных сетей и производить все вычисления на видеокартах. Но они не имеют интерфейса для обучения самих моделей. Процесс обучения можно определить как отдельную задачу, т. к. он имеет свои параметры и проблемы, которые также решаются различными техниками. Поэтому для каждого из перечисленных фреймворков были разработаны дополнительные фреймворки, которые содержат весь необходимый набор инструментов для обучения. Для TensorFlow есть официальное решение под названием Keras, в то время как для PyTorch существует несколько неофициальных решений, разработанные сообществом фреймворка. Самым крупным, с точки зрения предоставляемых возможностей, является фреймворк под названием Catalyst, разработанный отечественными энтузиастами. Помимо полного набора инструментов и удобного интерфейса для обучения, данный фреймворк содержит в себе все необходимое метрики и функции потерь, которые будут необходимы на этапе обучения модели.

3.1 Программная обработка набора данных

Каталог набора данных разбит на два каталога: «train», содержащий обучающие экземпляры данных, и «valid», содержащий проверочные экземпляры данных. Каждый такой каталог также разбит на два каталога: «images» содержащий RGB картинки поверхностей, и «masks» содержащий изображения бинарных масок соответствующих картинок. Имена файлов цветной картинки и её маски идентичны.

3.1.1 Чтение данных

Имея каталог с набором данных, необходимо разработать программное средство для чтения данных из файловой системы, их аугментации и перевода в тензоры. В фреймворке PyTorch имеются программные интерфейсы для этого. Реализация представлена в приложении 1. Данный код реализует класс для индексации по экземплярам данных и совершения всех необходимых операций над ними. В конструкторе класса инициализируется преобразователь для дальнейшей аугментации экземпляров данных (строка 30) и инициализируются списки путей ко всем цветным и бинарным изображениям (строки 31-34). Для чтения конкретных изображений из файлов системы определены соответствующие методы (строки 39-43). Все это используется в методе, определенном для индексации по экземплярам данных (строки 45-56). Данный метод считывает изображения из файловой системы (строки 46-47) и применяет к ним операции аугментации и переводит изображения в тензоры (строка 48). Здесь же каждой паре изображения присваивается соответствующий класс, обозначающий наличие трещины (строка 49). Наличие трещины определяется следующим образом: если соотношение количества белых пикселей к черным пикселям больше или равно 0.1%, значит трещина присутствует (строки 59-65). В результате после всех операций метод индексации возвращает словарь с тремя значениям: тензор из 5 изображений, тензор из 5 бинарных масок и тензор из 5 меток классов.

В процессе обучения такие экземпляры группируются в батчи, которые подаются непосредственно в модель. Такую группировку осуществляет класс фреймворка PyTorch под названием DataLoader. В классе также описан метод, возвращающий экземпляр этого класса (строки 81-89). Принцип, по которому набор экземпляров группируется в батч также описан в методе (строки 68-79).

3.1.2 Аугментация данных

Непосредственно код операций аугментации описан в приложении 2. Каждая отдельная операция описана в отдельном классе и действует одновременно на пару изображений. Для обработки изображений используются методы фреймворков Pillow [\*] и PyTorch. Таким образом в аугментации тренировочной и проверочной выборках использованы следующие операции:

Листинг 1. Аугментация выборок

|  |  |
| --- | --- |
| Аугментация тренировочной выборки | Аугментация проверочной выборки |
| Compose(  SingleChannel()  RandomVerticalFlip(p=0.5)  RandomHorizontalFlip(p=0.5)  RandomRotation()  FiveCrop(size=(224, 224))  ToTensor()  ) | Compose(  SingleChannel()  FiveCrop(size=(224, 224))  ToTensor()  ) |

Данный программный стиль аугментации повторяет стиль модуля для аугментации torchvision.transforms. Некоторые классы являются наследниками соответствующих классов данного модуля. Такое решение было принято в связи с тем, что родительские классы не поддерживают одновременную обработку пары изображений. Данная проблема была решена переопределением методов таких классов.

3.2 Программная реализация нейронных сетей

Программная реализация всех моделей нейронных сетей также осуществлялась средствами PyTorch. Слои моделей, блоки моделей и непосредственно сами модели описаны в модульном виде как отдельные классы. Данный подход позволяет конструировать архитектуру сети в виде иерархии из таких модулей. Были разработаны следующие модули:

1. «VGGBlock» модуль, реализующий блок архитектуры VGG.
2. «VGGEncoder» модуль, реализующий кодировщик архитектуры VGG, состоящий из последовательности блоков «VGGBlock».
3. «Upconv2d» модуль-наследник класса «ConvTranspose2d», отвечающий за операцию обратной свёртки. Данный модуль используется в декодере, а переопределенный метод «forward» в качестве второго аргумента принимает тензор из соответствующего обратной свёртки блока кодировщика, реализуя ранее описанный подход «skip connection».
4. «Decoder» модуль, реализующий принцип декодера, состоящий из последовательности блоков «Upconv2d» и «VGGBlock».
5. «UNet» непосредственно сама архитектуры U-Net, состоящий из ранее описанных модулей «VGGEncoder» и «Decoder».
6. «LinearBlock» модуль, реализующий блок полносвязной нейронной сети для задачи классификации.
7. «Classifier» – модуль полносвязной нейронной сети для задачи классификации, состоящий из последовательности блоков «LinearBlock» и модуля слоя субдискретизации «AvgPool2d», добавленный для усреднения карты признаков из кодировщика, что впоследствии уменьшает объем входных тензоров и общий объем самой модели-классификатора.
8. «VGGEncoderClassifier» модель, реализованная по принципу архитектуры VGG, состоящий из «VGGEncoder» и «Classifier». Данный модуль фиксирует состояние кодировщика, благодаря чего во время обучения изменяется состояние только модуля-классификатора.
9. «UNetClassifier» модель, совмещающая архитектуру U-Net с классификатором. Это позволяет одновременно получать бинарную маску трещины и выдавать класс, соответствующий наличию трещины.

3.3 Обучение моделей

Обучение проводилось в среде «Google Colab» с подпиской «Pro», который предлагает видеокарты двух видов: «NVIDIA Tesla P100» и «NVIDIA Tesla T4» объемом 16 гигабайт. В программной реализации обучения были использованы средства фреймворка Catalyst. Был реализован класс-наследник от catalyst.dl.Runner (приложение 4), который содержит метод «train», принимающий все необходимые для обучения компоненты и запускающий непосредственно само обучение. Данное решение позволило свести весь процесс обучения в переопределение метода «\_handle\_batch», который выполняет следующие операции:

1. Получение входных и целевых тензоров данных из батча.
2. Изменение состояния модели для обучения и тестирования.
3. Получение предсказаний модели на основе входных тензоров из батча и подсчет функции ошибки между предсказанными и целевыми тензорами.
4. Подсчет градиента функции ошибки и соответствующий шаг градиентного спуска.
5. Подсчет всех необходимых метрик.

Так как набор данных для каждого этапа один и тот же, был переопределен сам метод «train», в котором выполняется его автоматическая подстановка. Это позволяет избежать дублирования кода инициализации данных на обоих этапах.

В общем и целом, процесс обучения на обоих этапах идентичен. А уникальные операции для непосредственного подсчета функции ошибки и метрик были вынесены в абстрактные методы, обязательные для переопределения. Таким образом, на каждом этапе обучения реализуется класс-наследник класса «Trainer», в котором необходимо переопределить только уникальные операции.

3.3.1 Обучение модели для сегментации

…

3.3.2 Обучение модели для классификации

…

# Выводы

По результатам исследования могут быть сделаны следующие выводы:

1. …
2. …

# Заключение

В рамках исследования были выполнены все поставленные задачи и подтверждены выдвинутые гипотезы.

# Список литературы

[1] <https://arxiv.org/abs/2001.05566>

[2] <https://arxiv.org/abs/1906.02691>

[3] <https://www.mathworks.com/help/matlab/language-fundamentals.html>

[4] <https://www.r-project.org/>

[5] <https://www.python.org/>

[6] <https://pytorch.org/>

[7] <https://www.tensorflow.org/>

[8] <https://keras.io/>

[9] <https://catalyst-team.github.io/catalyst/>

[10] <http://yann.lecun.com/exdb/mnist/>

[11] <http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf>

[12] <https://arxiv.org/abs/1502.03167>

[13] <https://en.wikipedia.org/wiki/Rectifier_(neural_networks)>

[14] <https://arxiv.org/ftp/arxiv/papers/1708/1708.06243.pdf>

[15] <https://arxiv.org/abs/1505.04597>

[\*] <https://en.wikipedia.org/wiki/Supervised_learning>

[\*] <https://arxiv.org/abs/1409.1556>

[\*] <https://arxiv.org/abs/1902.02771> (Impact of Fully Connected Layers on Performance of Convolutional Neural Networks for Image Classification)

[\*] <https://arxiv.org/abs/2102.13653> (On the Generalization of Stochastic Gradient Descent with Momentum)

[\*] <https://arxiv.org/pdf/2006.14822.pdf> (A survey of loss functions for semantic

Segmentation)

Приложение

Листинг 1. Исходный код файла src/dataset.py

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89 | from typing import Tuple, List, Dict  from torch import Tensor  from PIL.Image import Image  import torch  from torch.utils.data import Dataset, DataLoader  from torchvision.datasets.folder import default\_loader  from pathlib import Path  from os import cpu\_count  from src import augmentations  class CracksDataset(Dataset):      TRANSFORM = {          'train': augmentations.Compose([              augmentations.SingleChannel(),              augmentations.RandomVerticalFlip(),              augmentations.RandomHorizontalFlip(),              augmentations.RandomRotation(),              augmentations.FiveCrop(224),              augmentations.ToTensor(),          ]),          'valid': augmentations.Compose([              augmentations.SingleChannel(),              augmentations.FiveCrop(224),              augmentations.ToTensor(),          ]),      }      def \_\_init\_\_(self, mode: str):          self.transform = self.TRANSFORM[mode]          self.images = list(Path(f'dataset/{mode}/images')\              .rglob('\*.jpg'))          self.masks = list(Path(f'dataset/{mode}/masks')\              .rglob('\*.jpg'))      def \_\_len\_\_(self):          return len(self.images)      def image(self, index: int) -> Image:          return default\_loader(self.images[index])      def mask(self, index: int) -> Image:          return default\_loader(self.masks[index])      def \_\_getitem\_\_(self, index: int) -> Dict[str, Tensor]:          image = self.image(index)          mask = self.mask(index)          images, masks = self.transform(image, mask)          cracks = self.is\_cracks\_exists(masks)          masks[cracks == 0] =\              torch.zeros\_like(masks[cracks == 0])          return {              'images': images,              'masks': masks,              'cracks': cracks,          }      @staticmethod      def is\_cracks\_exists(          masks: Tensor,          threshold: float = 0.001) -> Tensor:          h, w = masks.shape[-2:]          scale = masks.sum(dim=[1, 2, 3])          return (scale / (h \* w) >= threshold).int()      @staticmethod      def \_collate\_fn(          batch: List[Dict[str, Tensor]]) -> Dict[str, Tensor]:          images, masks, cracks = zip(\*(              (b['images'], b['masks'], b['cracks'])              for b in batch          ))          return {              'images': torch.cat(images),              'masks': torch.cat(masks),              'cracks': torch.cat(cracks).unsqueeze(1).float(),          }      def get\_loader(self, \*\*kwargs) -> DataLoader:          kwargs['num\_workers'] = kwargs.pop('num\_workers',\              cpu\_count())          return DataLoader(              dataset=self,              collate\_fn=self.\_collate\_fn,              pin\_memory=torch.cuda.is\_available(),              \*\*kwargs,          ) |

Листинг 2. Исходный код файла src/augmentations.py

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87 | from typing import Tuple, Union  from torch import Tensor  from PIL.Image import Image  import torch  from torchvision import transforms as T  from torchvision.transforms import functional as F  class Compose(T.Compose):      def \_\_call\_\_(self,          image: Union[Tensor, Image],          mask: Union[Tensor, Image],          ) -> Tuple[Union[Tensor, Image]]:          for t in self.transforms:              image, mask = t(image, mask)          return image, mask  class Transform(object):      def \_\_repr\_\_(self):          return self.\_\_class\_\_.\_\_name\_\_ + '()'  class SingleChannel(Transform):      def \_\_call\_\_(self,          image: Image,          mask: Image)-> Tuple[Image]:          mask = mask.convert('1')          return image, mask  class RandomHorizontalFlip(T.RandomHorizontalFlip):      def forward(self,          image: Image,          mask: Image) -> Tuple[Image]:          if torch.rand(1) < self.p:              image = F.hflip(image)              mask = F.hflip(mask)          return image, mask  class RandomVerticalFlip(T.RandomVerticalFlip):      def forward(self,          image: Image,          mask: Image)-> Tuple[Image]:          if torch.rand(1) < self.p:              image = F.vflip(image)              mask = F.vflip(mask)          return image, mask  class RandomRotation(Transform):      def \_\_call\_\_(self,          image: Image,          mask: Image) -> Tuple[Image]:          angle = 90 \* torch.randint(0, 4, (1,)).item()          image = F.rotate(image, angle)          mask = F.rotate(mask, angle)          return image, mask  class FiveCrop(T.FiveCrop):      def forward(self,          image: Image,          mask: Image) -> Tuple[Tuple[Image]]:          images = F.five\_crop(image, self.size)          masks = F.five\_crop(mask, self.size)          return images, masks  class ToTensor(T.ToTensor):      def \_\_call\_\_(self,          images: Tuple[Image],          masks: Tuple[Image]) -> Tuple[Tensor]:          images = torch.stack([              F.to\_tensor(image) for image in images])          masks = torch.stack([              F.to\_tensor(mask) for mask in masks])          return images, masks |

Листинг 3. Исходный код файла src/model.py

|  |  |
| --- | --- |
| 0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179 | from typing import Tuple  from torch import Tensor  import torch  from torch import nn  class VGGBlock(nn.Sequential):      def \_\_init\_\_(self,          in\_channels: int,          out\_channels: int,          bias: bool):          super().\_\_init\_\_(              nn.Conv2d(in\_channels, out\_channels,                  kernel\_size=3, padding=1, bias=bias),              nn.BatchNorm2d(out\_channels),              nn.ReLU(),              nn.Conv2d(out\_channels, out\_channels,                  kernel\_size=3, padding=1, bias=bias),              nn.BatchNorm2d(out\_channels),              nn.ReLU(),          )  class VGGEncoder(nn.Module):      def \_\_init\_\_(self,      in\_channels: int,      num\_features: int,      bias: bool):          super().\_\_init\_\_()          self.conv1 = VGGBlock(              in\_channels, num\_features, bias)          self.conv2 = VGGBlock(              num\_features, num\_features \* 2, bias)          self.conv3 = VGGBlock(              num\_features \* 2, num\_features \* 4, bias)          self.conv4 = VGGBlock(              num\_features \* 4, num\_features \* 8, bias)          self.bottleneck = VGGBlock(              num\_features \* 8, num\_features \* 16, bias)          self.pool = nn.MaxPool2d(2)      def forward(self, x: Tensor) -> Tuple[Tensor]:          x1 = self.conv1(x)          x2 = self.conv2(self.pool(x1))          x3 = self.conv3(self.pool(x2))          x4 = self.conv4(self.pool(x3))          x = self.bottleneck(self.pool(x4))          return x, x1, x2, x3, x4  class Upconv2d(nn.ConvTranspose2d):      def forward(self, x: Tensor, y: Tensor) -> Tensor:          x = super().forward(x)          x = torch.cat((x, y), dim=1)          return x  class Decoder(nn.Module):      def \_\_init\_\_(self, num\_features: int, bias):          super().\_\_init\_\_()          self.deconv4 = Upconv2d(              num\_features \* 16, num\_features \* 8,              kernel\_size=2, stride=2)          self.conv4 = VGGBlock(              num\_features \* 16, num\_features \* 8, bias)          self.deconv3 = Upconv2d(              num\_features \* 8, num\_features \* 4,              kernel\_size=2, stride=2)          self.conv3 = VGGBlock(              num\_features \* 8, num\_features \* 4, bias)          self.deconv2 = Upconv2d(              num\_features \* 4, num\_features \* 2,              kernel\_size=2, stride=2)          self.conv2 = VGGBlock(              num\_features \* 4, num\_features \* 2, bias)          self.deconv1 = Upconv2d(              num\_features \* 2, num\_features,              kernel\_size=2, stride=2)          self.conv1 = VGGBlock(              num\_features \* 2, num\_features, bias)          self.header = nn.Sequential(              nn.Conv2d(num\_features, 1, kernel\_size=1),              nn.Sigmoid(),          )      def forward(self,          x: Tensor,          x4: Tensor,          x3: Tensor,          x2: Tensor,          x1: Tensor) -> Tensor:          x = self.deconv4(x, x4)          x = self.deconv3(self.conv4(x), x3)          x = self.deconv2(self.conv3(x), x2)          x = self.deconv1(self.conv2(x), x1)          x = self.header(self.conv1(x))          return x  class UNet(nn.Module):      def \_\_init\_\_(self,          in\_channels: int = 3,          out\_channels: int = 1,          init\_features: int = 32,          bias: bool = False):          super().\_\_init\_\_()          self.VGGEncoder = VGGEncoder(              in\_channels, init\_features, bias)          self.decoder = Decoder(init\_features, bias)      def forward(self, x: Tensor) -> Tensor:          x, x1, x2, x3, x4 = self.VGGEncoder(x)          x = self.decoder(x, x4, x3, x2, x1)          return x  class LinearBlock(nn.Sequential):      def \_\_init\_\_(self, in\_features: int, out\_features: int):          super().\_\_init\_\_(              nn.Linear(in\_features, out\_features),              nn.BatchNorm1d(out\_features),              nn.ReLU(),          )  class Classifier(nn.Module):      def \_\_init\_\_(self):          super().\_\_init\_\_()          self.avgpool = nn.AvgPool2d(2)          self.header = nn.Sequential(              nn.Flatten(1),              LinearBlock(1024 \* 7 \* 7, 2048),              LinearBlock(2048, 1024),              nn.Linear(1024, 1),              nn.Sigmoid(),          )      def forward(self, x: Tensor) -> Tensor:          x = self.avgpool(x)          x = self.header(x)          return x  class VGGEncoderClassifier(nn.Module):      def \_\_init\_\_(self,          in\_channels: int = 3,          init\_features: int = 32,          bias: bool = False):          super().\_\_init\_\_()          self.VGGEncoder = VGGEncoder(              in\_channels, init\_features, bias)          self.VGGEncoder.requires\_grad\_(False)          self.classifier = Classifier()      def forward(self, x: Tensor) -> Tuple[Tensor]:          x = self.VGGEncoder(x)[0]          x = self.classifier(x)          return x  class UNetClassifier(UNet):      def \_\_init\_\_(self,          in\_channels: int = 3,          init\_features: int = 32,          bias: bool = False):          super().\_\_init\_\_()          self.classifier = Classifier()      def forward(self, x: Tensor) -> Tensor:          x, x1, x2, x3, x4 = self.VGGEncoder(x)          x = self.decoder(x, x4, x3, x2, x1)          y = self.classifier(x)          return x, y |

Листинг 4. Исходный код файла src/trainer.py

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66 | import torch  from src import dataset  from torch import Tensor  from catalyst.dl import Runner  from typing import Dict  class Trainer(Runner):      def \_\_init\_\_(self,          input\_key: str,          target\_key: str,          \*args, \*\*kwargs):          super().\_\_init\_\_(\*args, \*\*kwargs)          self.input\_key = input\_key          self.target\_key = target\_key      def \_calc\_loss(self,          outputs: Tensor,          targets: Tensor,          ) -> Tensor:          raise NotImplementedError      def \_calc\_metrics(self,          outputs: Tensor,          targets: Tensor) -> Dict[str, Tensor]:          raise NotImplementedError      def \_handle\_batch(self, batch: Tensor):          inputs = batch[self.input\_key]          targets = batch[self.target\_key]          self.model.train(self.is\_train\_loader)          with torch.set\_grad\_enabled(self.is\_train\_loader):              outputs = self.model(inputs)              loss = self.\_calc\_loss(outputs, targets)              if self.is\_train\_loader:                  self.optimizer.zero\_grad()                  loss.backward()                  self.optimizer.step()              outputs = outputs.detach()              self.batch\_metrics.update({                  'loss': loss.detach(),                  \*\*self.\_calc\_metrics(outputs, targets),              })      def train(self, \*args, \*\*kwargs):          datasets = {              'train': dataset.CracksDataset('train'),              'valid': dataset.CracksDataset('valid'),          }          batch\_size = kwargs.pop('batch\_size', 1)          loaders = {              'train': datasets['train'].get\_loader(                  batch\_size=batch\_size,                  shuffle=True,                  drop\_last=True),              'valid': datasets['valid'].get\_loader(                  batch\_size=batch\_size),          }          kwargs['loaders'] = loaders          super().train(\*args, \*\*kwargs) |