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

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 причин появления трещин используют так называемые «маяки» или «щелемеры». Их закрепляют непосредственно в области дефекта для отслеживания динамики разрушения.
2. Краш-тесты. Различные материалы подвергают критическим нагрузкам, вследствие чего также появляются трещины.
3. 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 видеонаблюдения. Анализируя поток изображений с камеры, можно определять наличие дефектов и выделять сегменты трещин. Такая информация может оказаться полезной для расчета тех или иных метрических характеристик трещин.

В компьютерном зрении задача сегментации трещин на поверхности относится к семейству задач семантической сегментации [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). Цвет пикселя такой маски обычно устанавливают в 1, если на соответствующем пикселе входного изображения присутствует искомый целевой объект, и устанавливают в 0, если на соответствующем пикселе объект отсутствует.

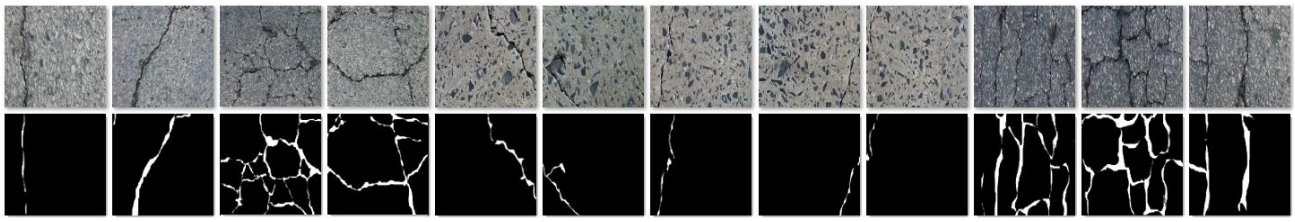

Рисунок 1. Пример сегментации трещин по картинке

Однако сегментация сама по себе не дает возможности получить однозначного ответа на вопрос «Есть ли трещина?». Для этого необходимо реализовывать дополнительные алгоритмы для анализа полученной бинарной маски. Такой подход требует дополнительных издержек как вычислительных, так и временных в следующих ситуациях:

1. Получение однозначного ответа на поставленный вопрос является более приоритетным, чем получение бинарной маски. Возможно, для негативного варианта такая маска вовсе не нужна.
2. Анализируемый поток информации по большей части не содержит трещин, и дополнительного анализа таких данных, в том числе и построения маски, вовсе не требуется.

В этой работе будет рассмотрен подход, который позволит определить наличие трещины на картинке без построения бинарной маски, делая это построение опциональным.

Постановка задачи

Целью данной работы является описание, реализация и обучение модели для определения и сегментации трещин на поверхности по изображению. Для начала необходимо сформулировать общие понятия каждой из задач.

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

1. Кодировщик. Сжимает входную информацию в так называемое «сжатое представление».
2. Декодер. Реконструирует сжатое представление к необходимому конечному виду.

Таким образом, имея входные данные X на выходе необходимо получить представление вида X' следующим образом:

$$E: X \to Z,$$
$$D: Z \to X',$$

где E – кодировщик, Z – сжатое представление, и D – декодер.

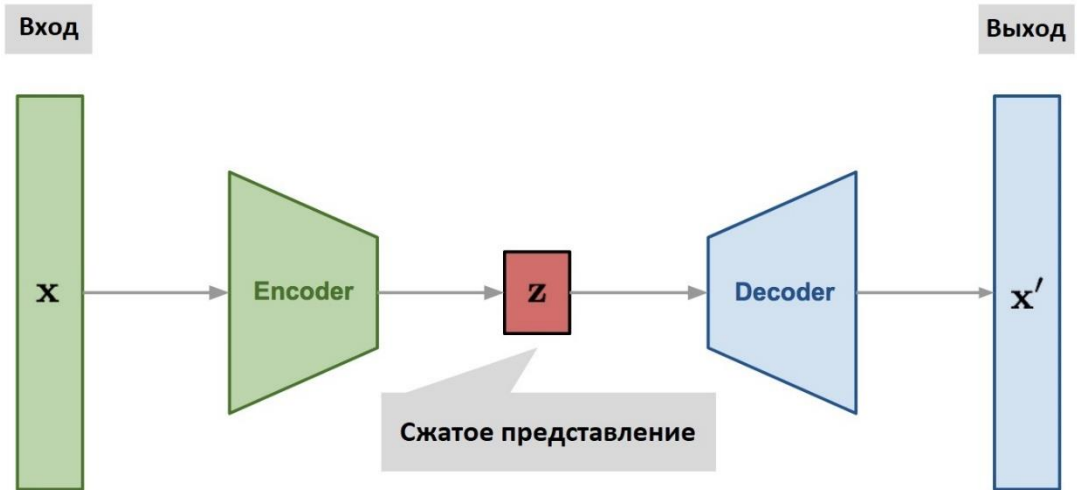

Рисунок 2. Принцип работы архитектур семейства «VAEs».

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 X из сжатого представления Z необходимо получить бинарное представление C:

$$F: Z \to P,$$

$$C = \begin{cases} 1, P > t \\ 0, P \leq t \end{cases},$$

$$P, t \in (0,1),$$

$$t - const,$$

где P – вероятность того, что на картинке присутствует трещина, t – 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» [3]. 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» [4] – 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 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 (запоминания набора данных), но и даёт возможность обучаемой сети познакомиться с большим набором уникальных ситуаций и выявить больше закономерностей. Процесс искусственного увеличения набора данных называется «аугментацией», и заключается в пропуске данных через различные, возможно случайные, операции обработки изображений. Рассмотрим примеры таких операций:

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

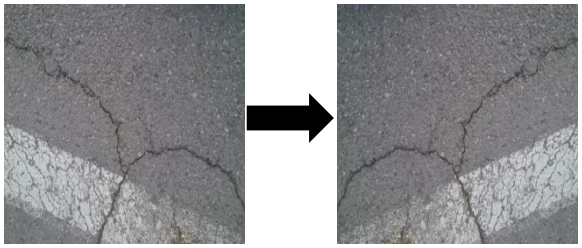

Рисунок 4. Горизонтальное отображение.

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

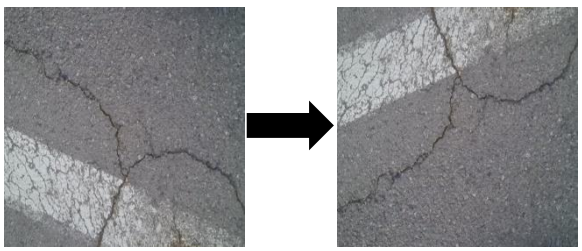

Рисунок 5. Вертикальное отображение.

3. Поворот на 0, 90, 180 и 270 градусов (Рисунок 6). Каждые 90 градусов получаются новые уникальные экземпляры.

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

Здесь понятие «уникальности» обозначает следующее: если в результате некоторого преобразования получена картинка, которая хотя бы одним пикселем отличается от исходной, значит получена новая уникальная картинка. Таким образом, случайное чередование описанных выше операций позволяет искусственно увеличить исходный набор данных в 16 раз.

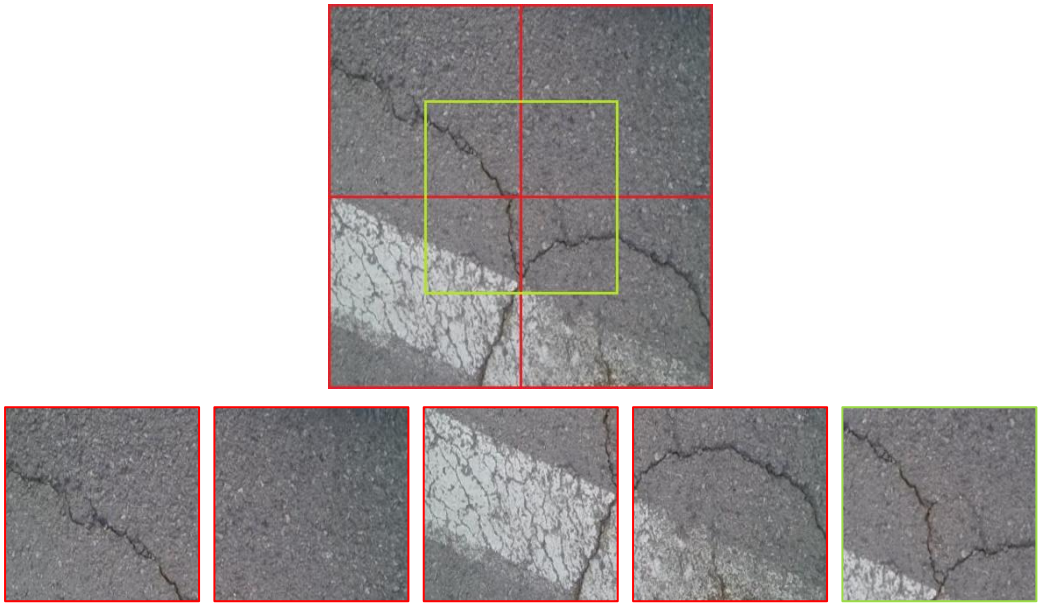

Рисунок 7. Принцип разделения картинки на 5 частей.

В рамках исследования размер картинок 448x448 пикселей является избыточным. Чем больше размер входного 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 занимают незначительную площадь, операция разделения гарантированно создаст изображения с отсутствием трещин, что даст необходимые экземпляры данных для задачи классификации. Будем считать, что на рассматриваемой картинке присутствует трещина, если площадь всех белых пикселей занимает больше или равно 0.1% площади всего изображения.

Глава 2. Применение глубоких нейронных сетей

2.1 Свёрточная нейронная сеть

Свёрточная нейронная сеть – архитектура искусственных нейронных сетей, которую впервые предложил Ян Лекун в 1998 году [5]. 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:

$$y = Wx + b$$

где x – входной вектор данных размерности n, y – выходной вектор размерности m, W – матрица весов размерности $n \times \mathrm{m}$, b – вектор смещения размерности m. Для обработки двумерных данных, например изображений, полносвязными нейронными сетями, данные необходимо приводить к одномерному виде. Например, одноканальную картинку размера 24x24 необходимо расправить в вектор длинной 284. Однако в обработке полученного вектора полносвязной нейронной сетью возникают следующие сложности:

1. Полносвязная нейронная сеть требует слишком большую входную размерность. Например, для обработки изображения размера 224x224 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 успешно справляется с этими проблемами путем использования свёрточных слоев вместо полносвязных для извлечения признаков. Рассмотрим принципы работы слоев свёрточной нейронной сети и примеры архитектур, основанные на них.

2.1.1 Свёрточный слой

Свёрточный слой определяется следующим образом:

$$y_{ijk} = \sum_{u=0}^{k_v} \sum_{v=0}^{k_h} \sum_{n=0}^{n(x)} W_{uvk} x_{(i+u)(j+v)n} + b_k$$

$$i \in \{as_v + 1 \mid a \in \{0, \dots, h(x)\ mod\ s_v\}\},$$

$$j \in \{as_h + 1 \mid a \in \{0, \dots, w(x)\ mod\ s_h\}\},$$

$$k \in \{1, \dots, n(y)\},$$

где W – тензор весов называемым ядром (фильтром) свёртки, b – вектор смещений, x и y – входной и выходной тензоры, k_v и k_h – размеры ядра (фильтра) свёртки, $h(x)$ и $w(x)$ – высота и ширина входного изображения, s_v и s_h – величина шага ядра по вертикали и горизонтали, $n(x)$ и $n(x)$ – количество каналов входного и выходного изображения соответственно, и mod – операция взятия остатка от деления.

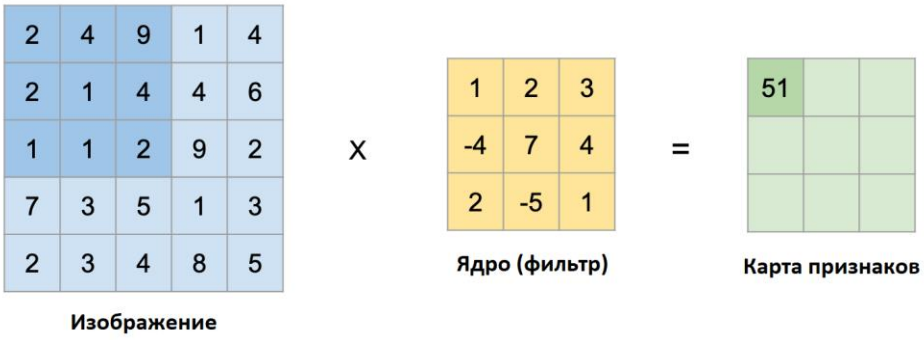

Рисунок 8. Демонстрация работы свёртки.

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 ядра, по краям карты признаков в зачастую добавляют нулевые контуры – «padding» различной толщины. 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 оставляя размерность выходной карты признаков как у входного изображения.

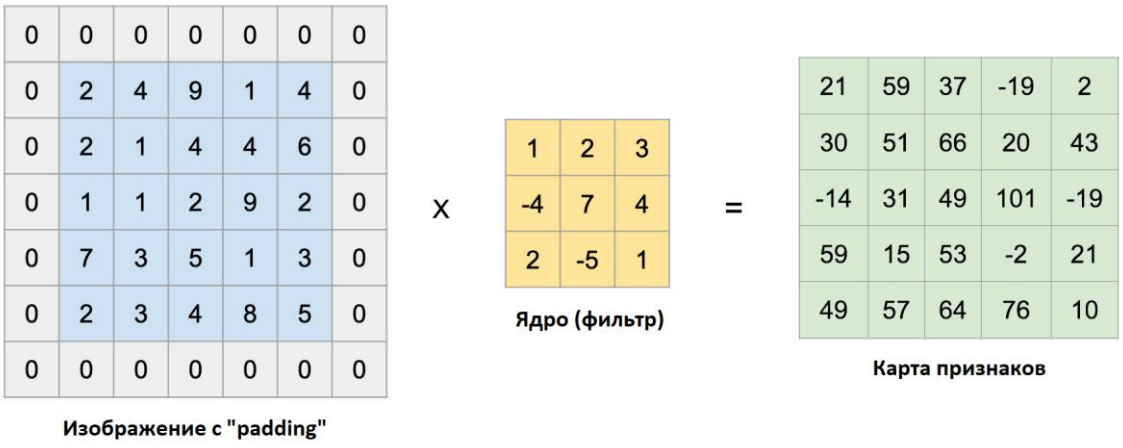

Рисунок 9. Добавление «padding».

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, как часть архитектуры нейронной сети, используется дополнительный слой нормализации признаков [6] с обучаемыми параметрами, действующий на так называемые «батчи данных». Батчами называют партию входных данных, которые подаются модели во время обучения. Разбиение на партии используется в работе большими наборами данных, которые невозможно полностью загрузить на видеокарту в силу ограничений видеопамяти. 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 $\{x\}_{i=1}^{L}$ и $\{y\}_{j=1}^{L}$ следующим образом:

$$y_{ijk}^{l} = \frac{x_{ijk}^{l} - \mu_{ijk}}{\sqrt{\sigma_{ijk}^{2} + \varepsilon}}\gamma + \beta_k,$$

$$i \in \{1, \dots, h(x)\},$$

$$j \in \{1, \dots, w(x)\},$$

$$k \in \{1, \dots, n(x)\},$$

$$l \in \{1, \dots, L\},$$

где индекс l – обозначает l-ый батч, μ и σ – накапливаемые статистики выборочного среднего и стандартного отклонения, посчитанные по всей подвыборке $\{x\}_{i=1}^{L}$, γ и β — обучаемые вектора параметров, используемые для масштабирования и смещения признаков к требуемому распределению, а ε — очень малое число-константа, необходимое для предотвращения деления на ноль.

2.1.3 Слой функции 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» [7]:

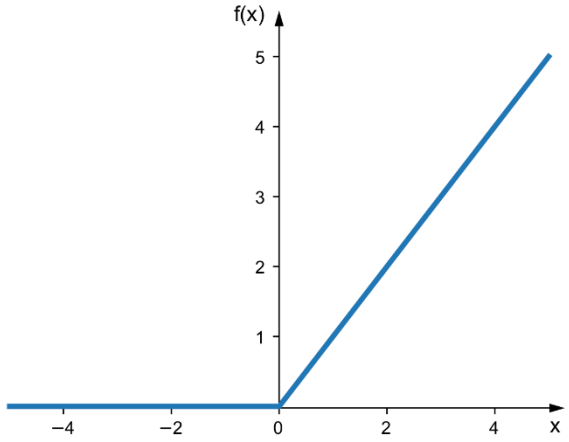

Рисунок 10. График функции активации ReLU.

С точки зрения обратного распространения ошибки [8], производная данной функции активации имеет следующий физический смысл: ошибка распространяется неизменно по тем путям, где сигнал положителен и не распространяется вовсе, где сигнал отрицателен:

$$ReLU(x) = x^{+} = \max(0, x) = \begin{cases} x, x \geq 0 \\ 0, x < 0 \end{cases}$$

$$\frac{dReLU(x)}{dx} = \begin{cases} 1, x \geq 0 \\ 0, x < 0 \end{cases}$$

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 для уплотнения карты

признаков. Слой субдискретизации с функцией максимума определяется следующим образом:

$$y_{ijk} = \max\{x_{(i+u)(j+v)k} \mid u, v \in \{0, \dots, m\}\},$$
$$i \in \{as_v + 1 \mid a \in \{0, \dots, h(x)\ mod\ s_v\}\},$$
$$j \in \{as_h + 1 \mid a \in \{0, \dots, w(x)\ mod\ s_h\}\},$$
$$k \in \{1, \dots, n(x)\},$$

где x – входной тензор размера $h(x) \times w(x) \times n(x)$, y – выходной тензор, $m \times m$ – размер окна, а s_v и s_h – величины шага по вертикали и горизонтали соответственно. Такая операция уменьшает размерность карты признаков, оставляя лишь наиболее «важные» в смысле максимума (Рисунок 12.).

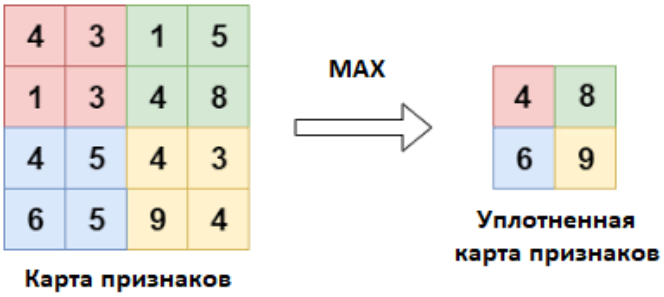

Рисунок 11. Демонстрация работы макс пулинга.

2.2 Архитектура свёрточной нейронной сети для классификации

В качестве примера рассмотрим архитектуру свёрточной нейронной сети под названием VGG-13 [9], которая продемонстрировала свою эффективность в решении задачи классификации на наборе данных ImageNet [10]. Данную архитектуру можно разделить на две ключевые компоненты:

1. Кодировщик, состоящий из последовательности ранее описанных слоёв. На данном этапе из картинки выделяются ключевые признаки, которые затем отправляются в полносвязный слой.
2. Полносвязный слой, анализирующий ключевые признаки из сжатого представления и интерпретирующий их в более низкоуровневые для конечной классификации.

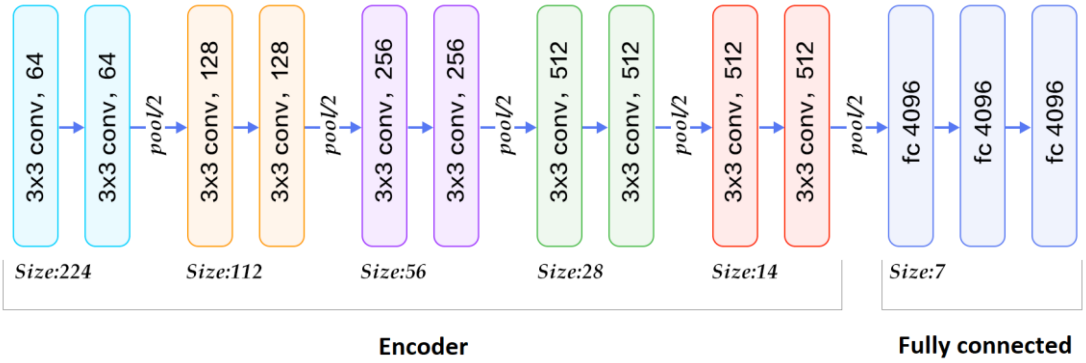

Рисунок 12. Архитектура VGG-13.

Эффективность данного подхода показана в статье [11], где 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: добавив сразу после кодировщика архитектуры «VAEs» дополнительные полносвязные нейронные слои для классификации изображения на классы «трещина присутствует» и «трещина отсутствует». Данный подход позволит определить наличие трещины на картинке, не реконструируя сжатое представление декодером, сделав саму реконструкцию опциональной.

2.3 Архитектура генеративной свёрточной нейронной сети для сегментации

Рассмотрим для примера архитектуру U-Net [12] семейства «VAEs», которую разработали в 2015 году для сегментации биомедицинских изображений. В данной архитектуре кодировщик использует блоки кодировщика ранее упомянутой архитектуры VGG-13.

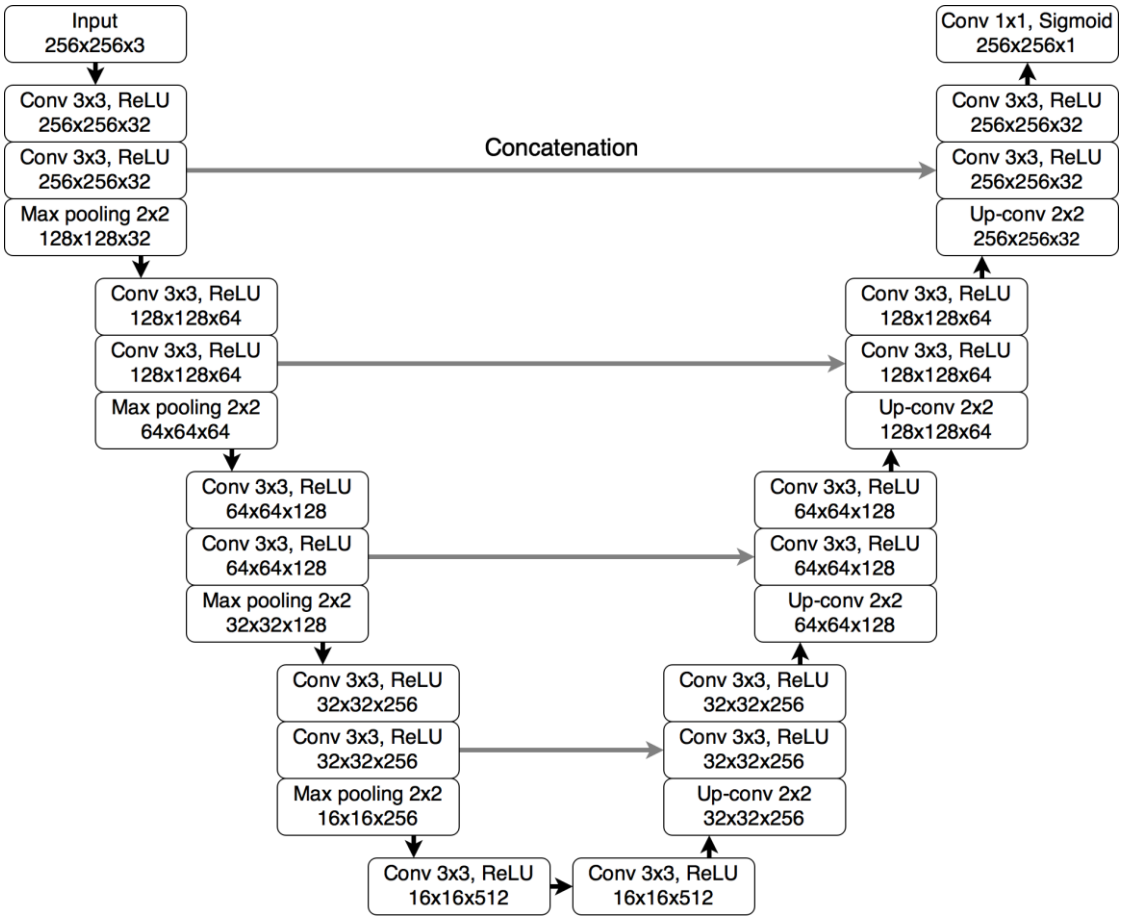

Рисунок 13. Архитектура U-Net.

При реконструкции изображение применяется подход под названием «skip connection». Визуально блоки архитектуры разделены на уровни таким образом, чтобы каждому блоку кодировщика на каждом уровне был соответствующий блок декодера той же размерности (Рисунок 14). Смысл подхода заключается в

соединении таких соответствующих блоков в определенных местах. Такой подход объединяет высокоуровневые признаки с более низкоуровневыми, что впоследствии позволяет с лучшей точностью реконструировать изображение. Саму «реконструкцию» сжатого представления в бинарную маску выполняют слои обратной (транспонированной) свёртки.

2.3.1 Слой транспонированной свёртки.

Данная операция использует описанный ранее «padding». Дополнительно можно заполнить «нулями» пространство между пикселями карты признаков. В результате выходная карта признаков получается большего размера, чем входная (Рисунок 15.).

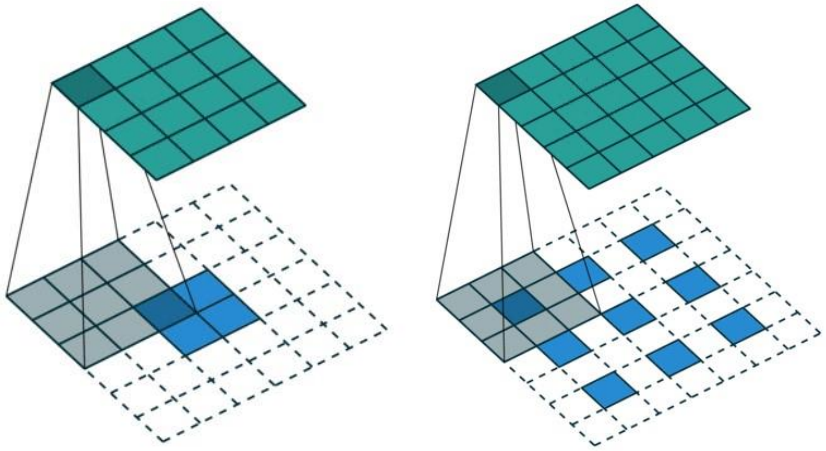

Рисунок 14. Транспонированная свёртка.

Глава 3. Критерии для 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 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 использован модифицированный метод стохастического градиентного спуска ADAM [13]. Каждая нейронная сеть обучалась на протяжении 64 эпох. При этом скорость обучения равнялась 10^{-2} и уменьшалась в 10 раз после 2, 32 и 48 эпох.

3.1 Критерий для задачи классификации

Для вероятностной интерпретации выходов нейронной сети классификатора выходы такой сети должны гарантированно принадлежать диапазону $(0, 1)$. Для этого выходы сети дополнительно обрабатываются сигмоидной функцией активации:

$$\sigma(x) = \frac{1}{1 + e^{-x}},$$

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\big(1 - \sigma(x)\big),$$

$$\sigma(x) \in (0,1), \forall x \in \mathbb{R}$$

В качестве критерия ошибки возьмем для начала функцию квадрата ошибки:

$$SE = (p - t)^2,$$

$$SE \in (0, +\infty),$$

$$SE \to min,$$

$$p = \sigma(y), \qquad y \in \mathbb{R}, \qquad t \in \{0,1\},$$

где y – это выход из нейронной сети, а t – это истинное значение класса. Рассмотрим производную этой функции по выходу сети – аргументу y:

$$\frac{\partial SE}{\partial y} = \frac{\partial SE}{\partial p} \cdot \frac{\partial p}{\partial y} = 2 \cdot (p - t) \cdot p \cdot (1 - p),$$

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 $\sigma(y) - t$, все равно возможна ситуация, когда значение производной будет равно нулю. Из-за чего обучать такую нейронную сеть при помощи градиентного спуска становиться невозможным. Проблему можно увидеть в следующем уравнении:

$$p(1-p) \approx 0, \qquad y \to -\infty,$$
$$p(1-p) \approx 0, \qquad y \to +\infty,$$

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:

$$BCE = -t\log(p) + (1-t)\log(1-p),$$
$$BCE \in (0, +\infty),$$
$$BCE \to min,$$
$$p = \sigma(y),$$

Аргумент логарифма всегда принадлежит положительному диапазону $(0,1)$, следовательно никогда не возникает ситуации, когда логарифм берется от нулевого или отрицательного числа. Рассмотрим производную данной функции все по тому же аргументу y:

$$\frac{\partial BCE}{\partial y} = \frac{\partial BCE}{\partial p} \cdot \frac{\partial p}{\partial y} =$$
$$= -\frac{t}{p} \cdot p \cdot (1-p) + \frac{1-t}{1-p} \cdot p \cdot (1-p) =$$
$$= -t \cdot (1-p) + (1-t) \cdot p =$$
$$= -t + t \cdot p + p - t \cdot p =$$
$$= p - t$$

Таким образом производная бинарной кросс энтропии по выходу сети не содержит никаких умножений на p или $1-p$, вследствие чего не происходит ранее описанного зануления производной. Поэтому для задачи бинарной классификации в качестве критерия можно использовать описанную функцию.

Также такая производная имеет физический смысл: производная критерия равна разности между предсказанным и истинным значениями.

3.2 Критерий для задачи сегментации

Для формулирования критерия для задачи сегментации рассмотрим меру под названием коэффициент Сёренсена [14], которая по своей сути является операцией над множествами:

$$Dice = \frac{2|A \cap B|}{|A| + |B|}$$

Данный коэффициент позволяет количественно описать меру сходства между двумя множествами A и B, 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 15. Применение метрики Dice к бинарным маскам.

Тогда операции над множествами заменяются на арифметические:

$$Dice = \frac{2pt + 1}{p + t + 1},$$

$$t \in \{0,1\}, \qquad p = \sigma(y), \qquad y \in \mathbb{R}$$

где y и t – предсказанное и целевое значения пикселя бинарной маски соответственно. Здесь в числитель и знаменатель добавлена 1, так как значение функции не определено, если $y = 0$ и $t = 0$. 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 выходы нейронной сети y аналогично обрабатываются сигмоидной функцией активации. Исходя из области значений данной функции можно сформулировать критерий ошибки:

$$Loss_{segmentation} = 1 - Dice,$$
$$Loss_{segmentation} \to min$$

Однако функция $Dice$ не является выпуклой [15], что создает сложности при оптимизации. В качестве решения автор статьи предлагает дополнить критерий раннее описанной функцией бинарной кросс энтропии:

$$Loss_{segmentation} = \alpha BCE + (1 - \alpha)(1 - Dice),$$
$$Loss_{segmentation} \to min,$$
$$\alpha \in (0,1),$$

где параметр α подбирается эмпирическим путем.

Глава 4. 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 3. Однако всем этим критериям удовлетворяет язык Python 3. Из подходящих фреймворков можно выделить два наиболее крупных и известных: PyTorch [16] и TensorFlow [17]. 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.

4.1 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.

4.1.1 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 29) и инициализируются списки путей ко всем цветным и бинарным изображениям (строки 30–33). Для чтения конкретных изображений из файлов системы определены соответствующие методы (строки 38–42). Все это используется в методе,

определенном для индексации по экземплярам данных (строки 44–55). Данный метод считывает изображения из файловой системы и применяет к ним операции аугментации и переводит изображения в тензоры (строки 45–47). Здесь же каждой паре изображения присваивается соответствующий класс, обозначающий наличие трещины (строка 48). Наличие трещины определяется следующим образом: если соотношение количества белых пикселей к черным пикселям больше или равно 0.1%, значит трещина присутствует (строки 57–64). 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 81–88). Принцип, по которому набор экземпляров группируется в батч также описан в методе (строки 66–79).

4.1.2 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:

Таблица 1. Аугментация выборок.

Аугментация тренировочной выборки	Аугментация проверочной выборки
Compose(RandomVerticalFlip(p=0.5)	Compose(FiveCrop(size=(224, 224))

RandomHorizontalFlip(p=0.5) RandomRotation() FiveCrop(size=(224, 224)) ToTensor())	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.

4.2 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.

4.3 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, оптимизатор и «планировщик» обучающего шага для каждого этапа одни и те же, был переопределен сам метод «train», в котором выполняется их автоматическая подстановка. Это позволяет избежать дублирования кода 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.

4.3.1 Обучение модели для сегментации

Реализация функции бинарной кросс-энтропия уже имеется в библиотеке PyTorch. Аналогично с критерием ошибки с использованием Dice, реализация которого представлена в библиотеке Catalyst. Для итогового критерия сегментации эмпирическим путем был выбран параметр $\alpha = 0.9$. Вся программная реализация представлена в листинге 5 приложения в строках 20–24.

В качестве дополнительных метрик рассматривались по отдельности компоненты оптимизируемого критерия: значения ошибки бинарной кросс-энтропии и значение метрики Dice. Здесь же была добавлена альтернативная метрика под названием «Intersection over Union», или коротко «IoU», для расчета схожести двух бинарных масок:

$$IoU = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

Реализация всех метрик также имеется в описанных библиотеках.

4.3.2 Обучение модели для классификации

Во время обучения непосредственно самой модели, веса модуля кодировщика берутся из кодировщика, обученного на первом этапе. Программная реализация представлена в листинге 6 приложения в строках 6–11.

Перед переходом к самим метрикам для задачи бинарной классификации необходимо обозначить важную концепцию для описания этих метрик в терминах ошибок классификации — матрица ошибок. Допустим, имеются два класса и алгоритм, предсказывающий принадлежность каждого входного объекта к одному из классов. Тогда матрица ошибок такой классификации будет выглядеть следующим образом:

Таблица 2. Концепция матрицы ошибок.

	$t = 1$	$t = 0$
$y = 1$	True Positive (TP)	False Positive (FP)
$y = 0$	False Negative (FN)	True Negative (TN)

где y – это предсказание алгоритма на объекте, а t – истинная метка класса на этом объекте. Используя данную концепцию, можно вывести следующие метрики качества классификации:

1. **Accuracy**. Самая интуитивно понятная, очевидная и почти неиспользуемая метрика, которая интерпретируется как «доля правильных ответов алгоритма»:

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

Однако использование такой метрики бесполезно в случае неравных объемов выборок классов. Например, пусть имеются 100 картинок с отсутствием трещин, 90 из которых классификатор определил верно ($TN = 9, FP = 10$), и 10 картинок с трещиной, 5 из которых классификатор определил верно ($TP = 5, FN = 5$). Тогда такая метрика равна:

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

Если допустить, что классификатор не обладает никакой предсказательной силой, и будет предсказывать для абсолютно всех изображений отсутствие трещин ($TP = 0, FP = 0$), метрика будет несправедливо высокой:

$$accuracy = \frac{0 + 90}{0 + 90 + 0 + 5} \approx 90.9$$

Преодолеть описанную проблему помогает переход с общей для всех классов метрики к отдельным показателям качества классов.

2. **Precision** и **Recall**. Precision можно интерпретировать как долю объектов, названных классификатором положительными и при этом действительно являющимися положительными, а recall показывает, какую долю объектов положительного класса из всех объектов положительного класса нашел алгоритм.

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

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

Именно введение precision не позволяет записывать все объекты в один класс, так как в этом случае получается рост значения FP в знаменателе. Recall демонстрирует способность алгоритма обнаруживать данный класс вообще, а precision — способность отличать этот класс от других классов. Данные метрики не зависят от соотношения классов и потому применимы в условиях несбалансированных выборок.

3. **F-мера**. Существует способ агрегировать precision и recall в единый критерий качества:

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

где β определяет вес precision в метрике, и при $\beta = 1$, F_1 – это среднее гармоническое. Данная мера достигает максимума при полноте и точности, равными единице, и близка к нулю, если один из аргументов близок к нулю.

Реализация всех описанных метрик имеется в библиотеке Catalyst.

4.4 Результаты

На рисунке 16 приведены графики функций ошибок на обоих этапах:

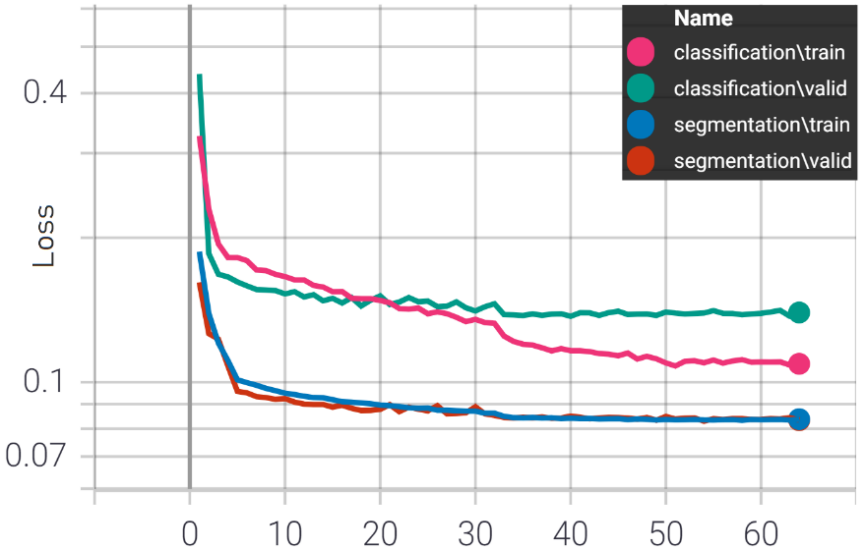

Рисунок 16. График функций ошибок.

Результаты лучших эпох обучения нейронных сетей с точки зрения минимальной ошибки на проверочной подвыборке приведены в таблице 3.

Таблица 3. Результаты проверочной выборки на лучших эпохах.

SEGMENTATION			
LOSS	0.083		
BCE	0.053		
DICE	0.647		
IOU	0.481		
CLASSIFICATION			
	Class 0	Class 1	Mean
PRECISION	0.920	0.966	0.943

RECALL	0.947	0.948	0.947
F1	0.933	0.957	0.945
LOSS		0.137	

На рисунке 17 представлены результаты одновременных предсказаний бинарных масок и классов на экземплярах данных, которые не участвовали в обучении.

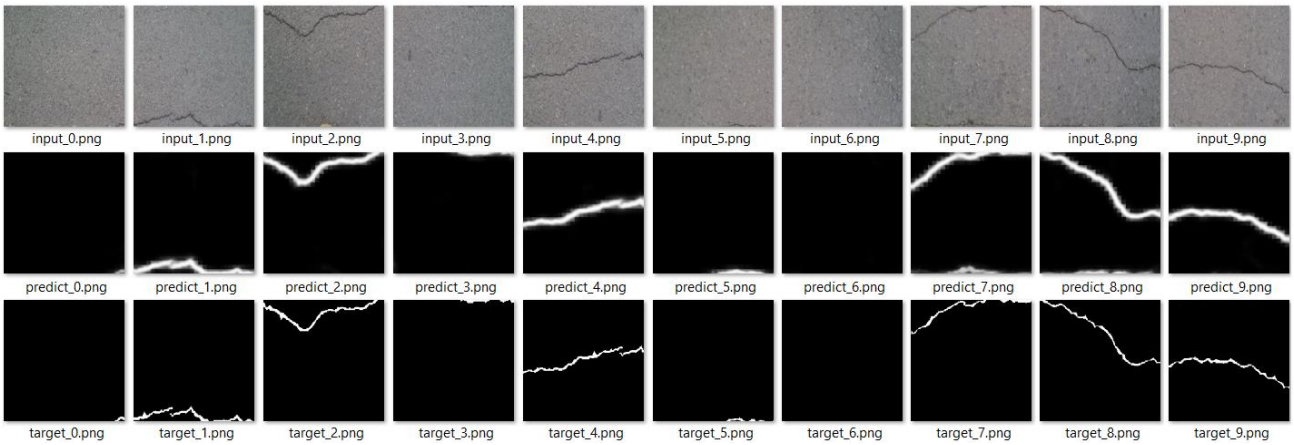

Рисунок 17. Тестирование бинарной маски.

Заключение

По результатам исследования могут быть сделаны следующие выводы:

1. Для решения задачи сегментации была реализована и успешно обучена модель архитектуры U-Net.
2. Кодировщик, обученный на задаче сегментации, может успешно применяться в задаче классификации, если к нему добавить полносвязные нейронные слои.
3. Полносвязная нейронная сеть, добавленная после уже обученного кодировщика, успешно анализирует информацию, исходящую из кодировщика и способна интерпретировать её для решения задачи классификации.

В рамках исследования были выполнены все поставленные задачи и подтверждены выдвинутые гипотезы. Систему определения трещины на поверхности можно действительно интегрировать в обученную «VAEs» архитектуру как отдельный модуль нейронной сети.

Список литературы

1. Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image Segmentation Using Deep Learning: A Survey. // arXiv:2001.05566. – 2019.
2. Diederik P. Kingma, Max Welling. An Introduction to Variational Autoencoders. // arXiv:1906.02691. – 2019. – № 3. – P. 15-16.
3. Набор данных «Crack Segmentation». – URL: https://github.com/khanhha/crack_segmentation (дата обращения: 05.06.2021).
4. Обучение с учителем (Supervised learning). – URL: https://en.wikipedia.org/wiki/Supervised_learning (дата обращения: 06.06.2021).
5. Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based Learning Applied to Document Recognition. // Proceedings of the IEEE. – 1998. – Т. 86. – № 11. – P. 2278-2324.
6. Sergey Ioffe, Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. // arXiv:1502.03167. – 2015.
7. Rectified Linear Unit activation function. – URL: https://en.wikipedia.org/wiki/Rectifier_(neural_networks) (дата обращения: 08.06.2021).
8. Backpropagation. – URL: https://en.wikipedia.org/wiki/Backpropagation (дата обращения: 08.06.2021).
9. Karen Simonyan, Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. // arXiv:1409.1556. – 2015.

10. ImageNet database. – URL: https://www.image-net.org/ (дата обращения: 10.06.2021).
11. S.H. Shabbeer Basha, Shiv Ram Dubey, Viswanath Pulabaigari, Snehasis Mukherjee, Impact of Fully Connected Layers on Performance of Convolutional Neural Networks for Image Classification. // arXiv:1902.02771. – 2019.
12. Olaf Ronneberger, Philipp Fischer, Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. // arXiv:1505.04597. – 2015.
13. Diederik P. Kingma, Jimmy Lei Ba. Adam: A method for stochastic optimization. // arXiv:1412.6980. – 2015.
14. Коэффициент Сёренсена (Dice coefficient). – URL: https://en.wikipedia.org/wiki/Sørensen–Dice_coefficient https://www.image-net.org/ (дата обращения: 10.06.2021).
15. Shruti Jadon. A survey of loss functions for semantic segmentation. // arXiv:2006.14822. – 2020. – № 4. P. 3.
16. Adam Paszke et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. // arXiv:1912.01703. – 2019.
17. Martín Abadi et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. // arXiv:1603.04467. – 2016.

Приложение

Листинг 1. Исходный код файла src/dataset.py

```
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.RandomVerticalFlip(),
            augmentations.RandomHorizontalFlip(),
            augmentations.RandomRotation(),
            augmentations.FiveCrop(224),
            augmentations.ToTensor(),
        ]),
        'valid': augmentations.Compose([
            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]).convert('1')
```

```
43
44      def __getitem__(self, index: int) -> Dict[str, Tensor]:
45          image = self.image(index)
46          mask = self.mask(index)
47          images, masks = self.transform(image, mask)
48          cracks = self.is_cracks_exists(masks)
49          masks[cracks == 0] =\
50              torch.zeros_like(masks[cracks == 0])
51          return {
52              'images': images,
53              'masks': masks,
54              'cracks': cracks,
55          }
56
57      @staticmethod
58      def is_cracks_exists(
59          masks: Tensor,
60          threshold: float = 0.001) -> Tensor:
61
62          h, w = masks.shape[-2:]
63          scale = masks.sum(dim=[1, 2, 3])
64          return (scale / (h * w) >= threshold).int()
65
66      @staticmethod
67      def _collate_fn(
68          batch: List[Dict[str, Tensor]],
69          ) -> Dict[str, Tensor]:
70
71          images, masks, cracks = zip(*(
72              (b['images'], b['masks'], b['cracks'])
73              for b in batch
74          ))
75          return {
76              'images': torch.cat(images),
77              'masks': torch.cat(masks),
78              'cracks': torch.cat(cracks).unsqueeze(1).float(),
79          }
80
81      def get_loader(self, **kwargs) -> DataLoader:
82          kwargs['num_workers'] = kwargs.pop('num_workers',\
83              cpu_count())
84          return DataLoader(
85              dataset=self,
86              collate_fn=self._collate_fn,
87              pin_memory=torch.cuda.is_available(),
88              **kwargs)
```

Листинг 2. Исходный код файла src/augmentations.py

```
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 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

```

```
47
48 class RandomRotation(Transform):
49     def __call__(self,
50         image: Image,
51         mask: Image) -> Tuple[Image]:
52
53         angle = 90 * torch.randint(0, 4, (1,)).item()
54         image = F.rotate(image, angle)
55         mask = F.rotate(mask, angle)
56         return image, mask
57
58
59 class FiveCrop(T.FiveCrop):
60     def forward(self,
61         image: Image,
62         mask: Image) -> Tuple[Tuple[Image]]:
63
64         images = F.five_crop(image, self.size)
65         masks = F.five_crop(mask, self.size)
66         return images, masks
67
68
69 class ToTensor(T.ToTensor):
70     def __call__(self,
71         images: Tuple[Image],
72         masks: Tuple[Image]) -> Tuple[Tensor]:
73
74         images = torch.stack([
75             F.to_tensor(image) for image in images])
76         masks = torch.stack([
77             F.to_tensor(mask) for mask in masks])
78         return images, masks
```

Листинг 3. Исходный код файла src/model.py

```
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,
```

```
94          x1: Tensor) -> Tensor:
95
96          x = self.deconv4(x, x4)
97          x = self.deconv3(self.conv4(x), x3)
98          x = self.deconv2(self.conv3(x), x2)
99          x = self.deconv1(self.conv2(x), x1)
100         x = self.header(self.conv1(x))
101         return x
102
103
104 class UNet(nn.Module):
105     def __init__(self,
106         in_channels: int = 3,
107         out_channels: int = 1,
108         init_features: int = 32,
109         bias: bool = False):
110         super().__init__()
111         self.encoder = VGGEncoder(
112             in_channels, init_features, bias)
113         self.decoder = Decoder(init_features, bias)
114
115     def forward(self, x: Tensor) -> Tensor:
116         x, x1, x2, x3, x4 = self.encoder(x)
117         x = self.decoder(x, x4, x3, x2, x1)
118         return x
119
120
121 class LinearBlock(nn.Sequential):
122     def __init__(self, in_features: int, out_features: int):
123         super().__init__(
124             nn.Linear(in_features, out_features),
125             nn.BatchNorm1d(out_features),
126             nn.ReLU(),
127         )
128
129
130 class Classifier(nn.Module):
131     def __init__(self):
132         super().__init__()
133         self.avgpool = nn.AvgPool2d(2)
134         self.header = nn.Sequential(
135             nn.Flatten(1),
136             LinearBlock(512 * 7 * 7, 2048),
137             LinearBlock(2048, 1024),
138             nn.Linear(1024, 1),
139             nn.Sigmoid(),
140         )
141
```

```
142     def forward(self, x: Tensor) -> Tensor:
143         x = self.avgpool(x)
144         x = self.header(x)
145         return x
146 
147 
148 class VGGEncoderClassifier(nn.Module):
149     def __init__(self,
150         in_channels: int = 3,
151         init_features: int = 32,
152         bias: bool = False):
153 
154         super().__init__()
155         self.encoder = VGGEncoder(
156             in_channels, init_features, bias)
157         self.encoder.requires_grad_(False)
158         self.classifier = Classifier()
159 
160     def forward(self, x: Tensor) -> Tuple[Tensor]:
161         x = self.encoder(x)[0]
162         x = self.classifier(x)
163         return x
164 
165 
166 class UNetClassifier(UNet):
167     def __init__(self,
168         in_channels: int = 3,
169         init_features: int = 32,
170         bias: bool = False):
171 
172         super().__init__()
173         self.classifier = Classifier()
174         self.eval()
175 
176     def forward(self, x: Tensor) -> Tensor:
177         z = torch.zeros_like(x)
178         x, x1, x2, x3, x4 = self.encoder(x)
179         y = self.classifier(x)
180         idx = torch.where(y > 0.5)[0]
181         z[idx] = self.decoder(
182             x[idx], x4[idx], x3[idx], x2[idx], x1[idx])
183         return z, y
```

Листинг 4. Исходный код файла src/trainer.py

```
import torch
from src.dataset import CracksDataset, DataLoader
from torch import Tensor
from torch.nn import Module
from torch.optim import Optimizer
from torch.optim.lr_scheduler import _LRScheduler
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(),
```

```
46                  'lr': self.scheduler.get_last_lr()[0],
47                  **self._calc_metrics(outputs, targets),
48              })
49
50      def _get_datasets(self) -> Dict[str, CracksDataset]:
51          return {
52              'train': CracksDataset('train'),
53              'valid': CracksDataset('valid'),
54          }
55
56      def _get_loaders(self,
57          batch_size: int) -> Dict[str, DataLoader]:
58
59          datasets = self._get_datasets()
60          return {
61              'train': datasets['train'].get_loader(
62                  batch_size=batch_size,
63                  shuffle=True,
64                  drop_last=True),
65              'valid': datasets['valid'].get_loader(
66                  batch_size=batch_size),
67          }
68
69      def _get_optimizer(self, model: Module) -> Optimizer:
70          return torch.optim.Adam(model.parameters(), lr=1e-2)
71
72      def _get_scheduler(self,
73          optimizer: Optimizer) -> _LRScheduler:
74
75          return torch.optim.lr_scheduler.MultiStepLR(
76              optimizer, milestones=[2, 32, 48], gamma=0.1)
77
78      def train(self, *args, **kwargs):
79          batch_size = kwargs.pop('batch_size', 1)
80          loaders = self._get_loaders(batch_size)
81          model = kwargs.pop('model')
82          optimizer = self._get_optimizer(model)
83          scheduler = self._get_scheduler(optimizer)
84          kwargs['loaders'] = loaders
85          kwargs['model'] = model
86          kwargs['optimizer'] = optimizer
87          kwargs['scheduler'] = scheduler
88          super().train(*args, **kwargs)
89
90      def on_epoch_end(self, runner):
91          super().on_epoch_end(runner)
92          self.scheduler.step()
```

Листинг 5. Исходный код файла segmentation.py

```
import torch
import torch.nn.functional as F
from src import dataset, model, trainer
from catalyst import utils, dl, contrib, metrics
utils.set_global_seed(17)

net = model.UNet()
criterion = {
    'dice': contrib.nn.DiceLoss(activation=None),
    'bce': torch.nn.BCELoss(),
}
alpha = 0.9


class Trainer(trainer.Trainer):
    def _calc_loss(self, outputs, targets):
        loss_bce = self.criterion['bce'](outputs, targets)
        loss_dice = self.criterion['dice'](outputs, targets)
        loss = alpha * loss_bce + (1 - alpha) * loss_dice
        return loss

    def _calc_metrics(self, outputs, targets):
        return {
            'bce': F.binary_cross_entropy(outputs, targets),
            'dice': metrics.dice(outputs, targets),
            'iou': metrics.iou(outputs, targets),
        }

runner = Trainer(input_key='images', target_key='masks')
runner.train(
    model=net,
    criterion=criterion,
    batch_size=12,
    num_epochs=64,
    logdir=logdir,
    verbose=True)
```

Листинг 6. Исходный код файла classification.py

```
import torch
from src import dataset, model, trainer
from catalyst import utils, dl, contrib
utils.set_global_seed(17)

net = model.VGGEncoderClassifier()
state_dict = torch.load(
    'logs/segmentation/1/checkpoints/best.pth',
    map_location='cpu')
net.load_state_dict(
    state_dict['model_state_dict'], strict=False)
criterion = torch.nn.BCELoss()


class Trainer(trainer.Trainer):
    def _calc_loss(self, outputs, targets):
        loss = self.criterion(outputs, targets)
        return loss

    def _calc_metrics(self, outputs, targets):
        logits = (outputs > 0.5).float()
        precision, recall, f1, _ = metrics\
            .precision_recall_fbeta_support(logits, targets)

        return {
            'precision/0': precision[0],
            'recall/0': recall[0],
            'f1/0': f1[0],
            'precision/1': precision[1],
            'recall/1': recall[1],
            'f1/1': f1[1],
            'precision/mean': precision.mean(),
            'recall/mean': recall.mean(),
            'f1/mean': f1.mean(),
        }

runner = Trainer(input_key='images', target_key='cracks')
runner.train(
    model=net,
    criterion=criterion,
    batch_size=12,
    num_epochs=64,
    logdir=logdir,
    verbose=True)
```