# Подготовка данных для обучения

## Описание полученных данных

Набор данных получен на 18 кафедре НИЯУ «МИФИ» при анализе вихретоковым дефектоскопом НАЗВАНИЕ/ОПИСАНИЕ оболочки тепловыделяющего элемента реактора БН600.

Оболочка имеет два вида дефектов. С левого края находятся три расположенных подряд пропила. Далее следует пять отверстий, одно из которых сквозное.

После выполнения программным обеспечением функции обработки данных формируется таблица значений диагностического сигнала в формате

текстового файла с расширением \*.txt. Данные в рамках одного файла представляют собой двумерный массив. Количество различных файлов с массивами равно числу измерений в осевом направлении движения датчика. Вид массива показан на Рисунке 2.



Рисунок 2. Массив данных.

На Рисунке 2 ХR – реальное значение сопротивления в цепи, XL – мнимое. Значение n – число измерений в радиальном направлении движения датчика.

Массивы данных обрабатываются в программе Labview. Для преобразования использовалось только реальное значение сопротивление, использование мнимой компоненты было бы избыточным. ПРО ПРЕОБРАЗОВАНИЕ.

На выходе сигнал представляет собой цветное 2-Д изображение (Рисунок 3), на котором отчетливо видно различные дефекты.

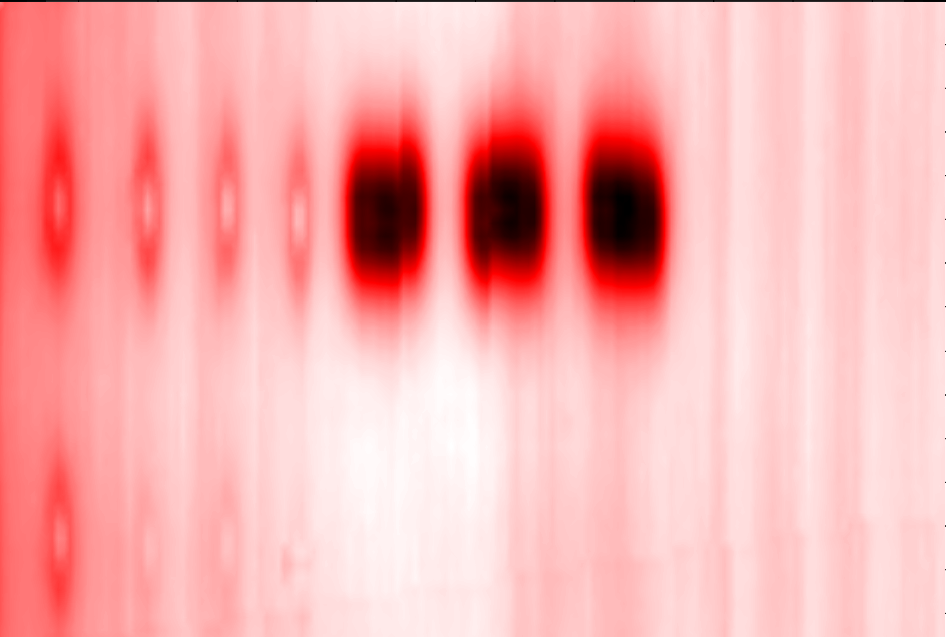


Рисунок 3. Пример сигнала дефектоскопа после обработки в Labview.

С целью предотвращения переобучения нейронной сети необходимо выбрать набор различных данных, иначе нейронная сеть запомнит определенный тип изображений и не сможет быть обобщена на детекцию дефектов в оболочках, отличных от данной.

Всего было получено 50 изображений, из которых 34 имели различные дефекты и использовались для дальнейшей разметки.

## Разметка изображений.

Для разметки изображений использовалась утилита LabelImg - инструмент для графического аннотирования изображений. [1] В LabelImg разметка хранится в виде XML файла в формате PASCAL VOC, который так же использовался для хранения разметки изображений в базе данных ImageNet. [2] Так же LabelImg поддерживает формат YOLO. Окно программы показано на Рисунке 4.

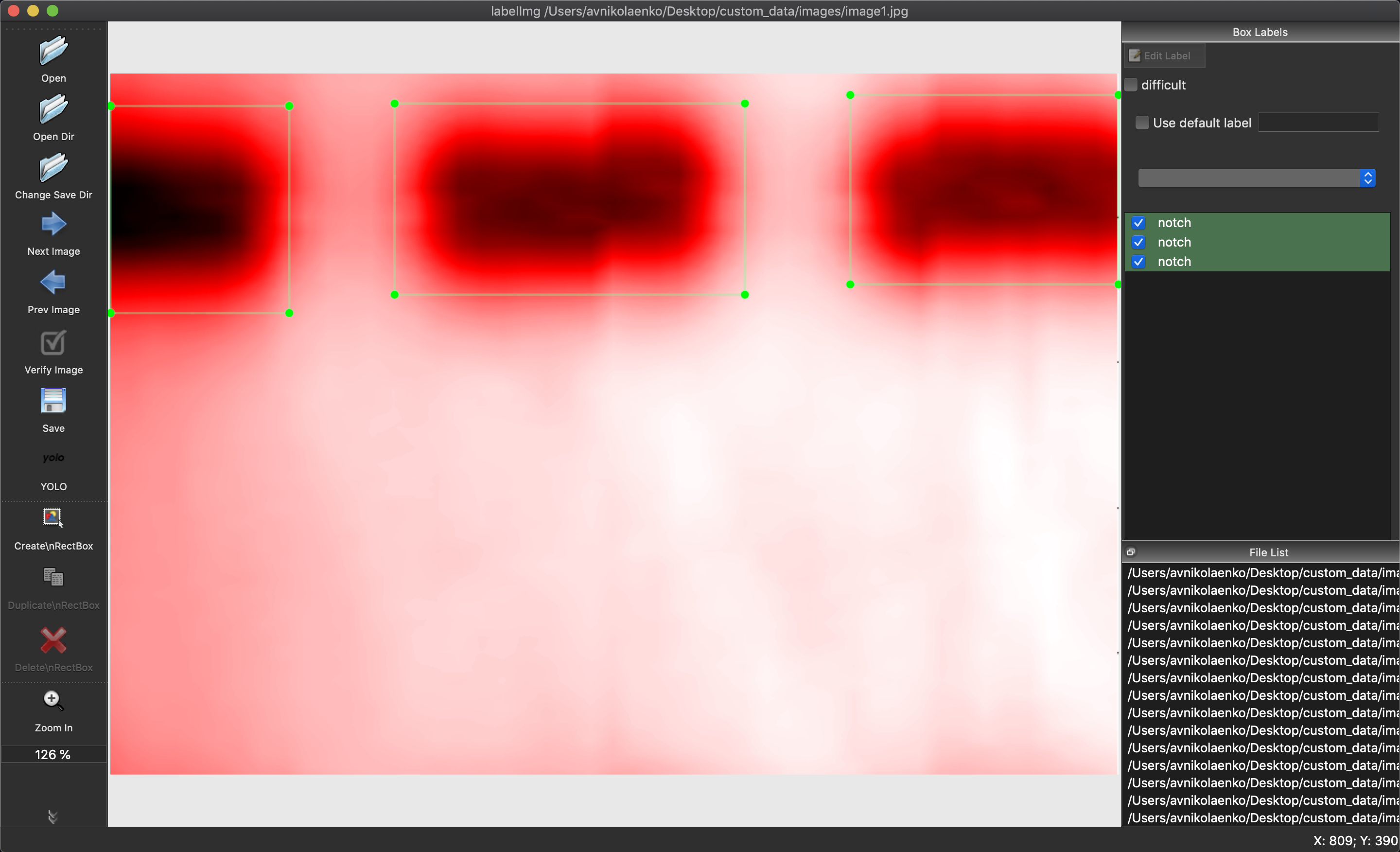


Рисунок 4. Окно программы LabelImg.

Разметка сохранялась в формате PASCAL VOC. На полученных изображениях удалось выделить два класса – отверстия и пропилы. Для каждого изображения дефекты помещались в соответствующие им обрамляющие окна, информация о расположении дефектов сохранялась в виде XML файла.

Далее разметка конвертировалась в формат COCO при помощи программы voc2coco [3]. Для этого были выполнены следующие команды в командной строке:

pip install lxml

python voc2coco.py xmllist.txt \ /Users/avnikolaenko/Desktop/custom\_data/Annotations annotation.json

В результате выполнения команд исходные файлы с разметкой в формате XML были сохранены в один файл JSON формата для дальнейшего применения при построении датасета.

# Построение модели.

## Выбор языка программирования.

Одним из самых популярных высокоуровневых динамических языков программирования является Python. Код на данном языке удобен в использовании и прост для чтения. Ниже на Рисунке 5 представлен график, на котором показано предсказание модели по использованию языков программирования в 2020 году, на основании статистики предыдущих лет. Так же, Python легко взаимодействует с другими языками, особенно с C и C++.

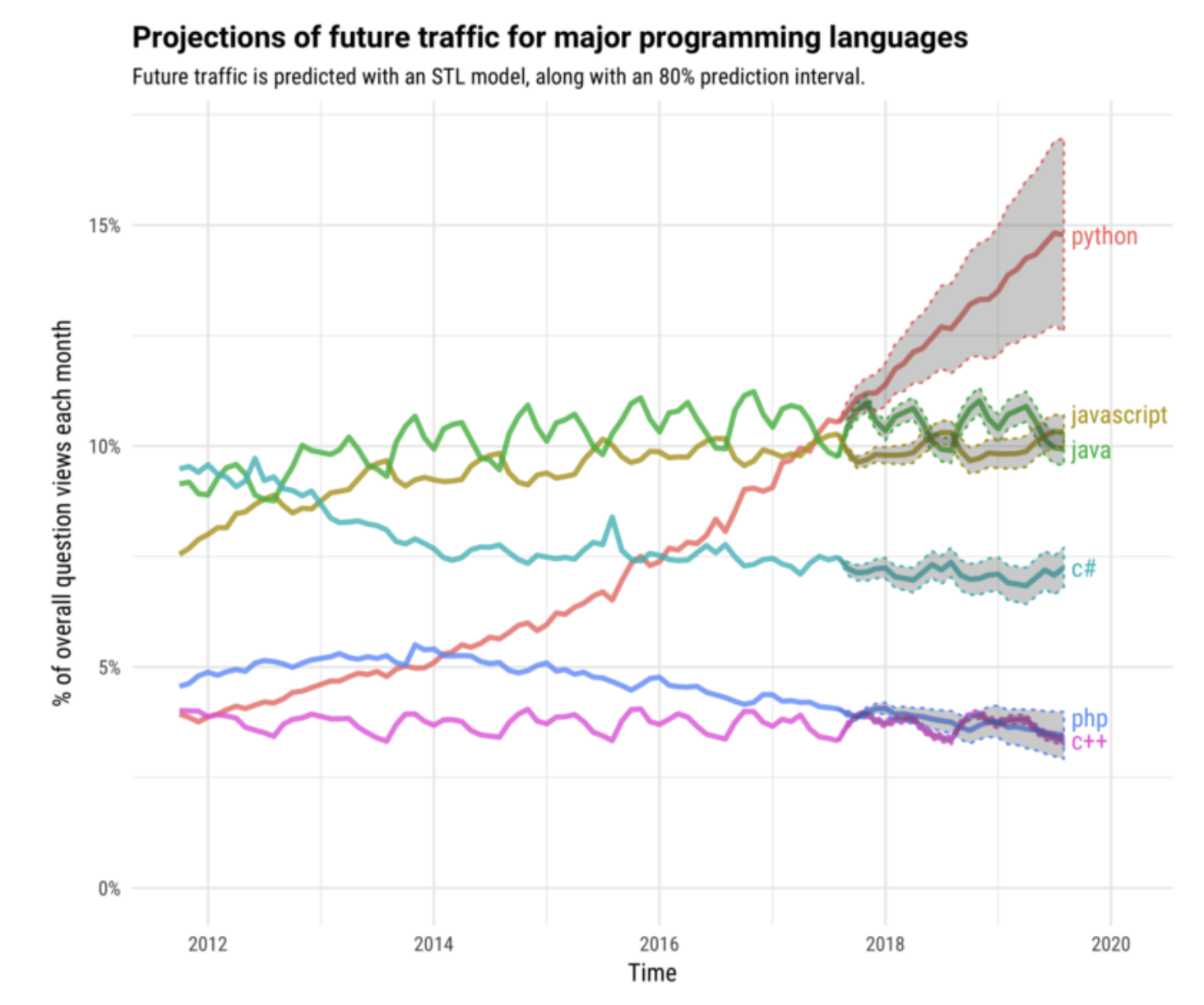


Рисунок 5. Популярность языков программирования.

Одна из основных причин, использования Python для машинного обучения в том, что у него есть множество фреймворков, которые упрощают процесс написания кода и сокращают время на разработку.

Существует фреймворк для Python, разработанный специально для машинного обучения — это PyTorch.

Модель для задачи строится при помощи библиотеки PyTorch. PyTorch — библиотека машинного обучения для языка Python с открытым исходным кодом, созданная на базе Torch. Используется для решения различных задач: компьютерное зрение, обработка естественного языка. [4]

## Построение датасета.

Библиотека PyTorch предоставляет стандартный класс torch.utils.data.Dataset, от которого будет наследоваться класс, преобразующий набор данных в работе. Класс будет предоставлять два основных метода: \_\_len\_\_ и \_\_getitem\_\_.

\_\_getitem\_\_ должен возвращать:

1. image: изображение в формате PIL размера (H, W)
2. target: словарь, содержащий следующие поля:

* boxes (FloatTensor[N, 4]): координаты N ограничивающих прямоугольников (bounding box) в формате [x0, y0, x1, y1], в диапазоне от 0 до W и от 0 до H
* labels (Int64Tensor [N]): метки классов для каждого ограничивающего прямоугольника.
* image\_id (Int64Tensor [1]).: идентификатор изображения. Он должен быть уникальным для всех изображений в наборе данных.
* area (Tensor [N]): площадь каждого прямоугольника, использующаяся во время оценки.
* iscrowd (UInt8Tensor [N]): фон. Экземпляры с iscrowd = True будут игнорироваться во время оценки.

Фрагмент кода для построения класса DefectDataset представлен в Приложении.

## Архитектура модели.

Как описывалось ранее, Faster R-CNN обладает рядом преимуществ. Одно из них – точность. Так как в нашей задаче гораздо важнее точность, чем быстродействие, для реализации была выбрана именно эта архитектура.

Так как размер датасета в нашей задаче очень мал, мы возьмем ранее предобученную модель и дообучим ее на наши классы – отверстия и пропилы.

Предобученные нейронные сети умеют распознавать образы и не требуют долгой подготовки сети к работе. Такие сети уже натренированы на большом количестве изображений и умеют распознавать простейшие объекты. Плюс предобучения в том, что веса для модели не будут настраиваться с нуля.

Для этого необходимо определить объект стандартного класса

model= torchvision.models.detection.fasterrcnn\_resnet50\_fpn(pretrained=True)

с параметром pretrained=True, который отвечает за загрузку предобученной модели. Экземпляр предварительно обученной модели загрузит ее веса в кэш. Все предобученные модели ожидают на вход одинаково нормализованные изображения, а именно трехканальные изображения (3 x W x H), где W –ширина изображения, H – высота. Параметры H и W должны иметь значения не меньше 224. Значение каждого пикселя должно лежать в диапазоне [0,1]. При нормализации должны быть использованы значения среднего - [0,485, 0,456, 0,406], и значения стандартных отклонений [0,229, 0,224, 0,225].

Для дальнейшего определения модели необходимо установить параметр in\_features, который отвечает за количество нейронов входного слоя. Данный параметр оставим как в предобученной модели. Для этого необходимо обратиться к атрибуту in\_features загруженной модели.

in\_features = model.roi\_heads.box\_predictor.cls\_score.in\_features

Количество выходных нейронов должно определяться количеством детектируемых классов в задаче. В нашем случае таких классов три - пропилы, отверстия и фон. Определение модели вынесем в отдельную функцию get\_model, которая на вход будет получать один параметр – количество классов в задаче. Пример вызова функции:

get\_model(num\_classes)

Фрагмент кода для построения модели представлен в Приложении.

## Обучение модели.

Обучение модели будет происходить на графическом процессоре (GPU), так как GPU обладают высокой производительностью – графические процессоры эффективно справляются с большим количеством несложных однотипных задач благодаря архитектуре своего ядра. Обучение нейронных сетей на CPU занимает в несколько раз больше времени, чем на видеокартах. Графические процессоры используют преимущества параллелизма, а нейронные сети, в тоже время, параллельные алгоритмы. Также видеокарты оптимизированы для матричных операций и ускоряют их – они необходимы нейронным сетям для получения результата. Для обучения на GPU создадим переменную device:

device = torch.device('cuda:0')

Для того, чтобы отправить модель на GPU, воспользуемся стандартным методом модели - to и передадим в него параметр device.

model.to(device)

Для организации набора данных запишем путь к изображениям и аннотациям к ним в отдельные переменные train\_data\_dir и train\_coco. Эти параметры используются при определении класса DefectDataset.

train\_data\_dir = 'data/images'

train\_coco = 'data/annotations.json'

Вместе с этими параметрами передается параметр transforms, который отвечает за нормализацию отдельных изображений.

Разделим исходные изображения на два набора: набор для обучения и валидационный набор, на которым мы будем смотреть метрики во время обучения. Для валидации оставим 7 случайно выбранных изображений, остальные поместим в обучающую выборку.

torch.optim - пакет, реализующий различные алгоритмы оптимизации. В рамках данной задачи для оптимизации использовался стохастический градиентный спуск (SGD). SGD - метод итерации для оптимизации целевой функции с подходящими свойствами гладкости (например, дифференцируемость). Его можно расценивать как стохастическую аппроксимацию оптимизации методом градиентного спуска, поскольку он заменяет реальный градиент (вычисленный из полного набора данных) путём оценки такового (вычисленного из случайно выбранного подмножества данных). Особенно в приложениях обработки больших данных это сокращает вычислительные ресурсы, достигая более быстрые итерации в обмен на более низкую скорость сходимости [5]. Определение оптимизатора:

torch.optim.SGD(params, lr=0.005, momentum=0.9, weight\_decay=0.0005)

Где params – параметры текущей модели:

params = [p for p in model.parameters() if p.requires\_grad]

Для начала было принято решение обучить модель на 10 эпохах. Эпоха - одна итерация в процессе обучения, включающая предъявление всех примеров из обучающего множества и проверку качества обучения на валидационном наборе данных.

Для обучения и валидации использовались функции из пакета engine.py. [6].

train\_one\_epoch(model, optimizer, data\_loader, device, epoch, print\_freq=10)

evaluate(model, data\_loader\_test, device=device)

Каждую эпоху в стандартный поток вывода записывались значения метрик на валидации. Пример вывода на одной эпохе показан на Рисунке 6.

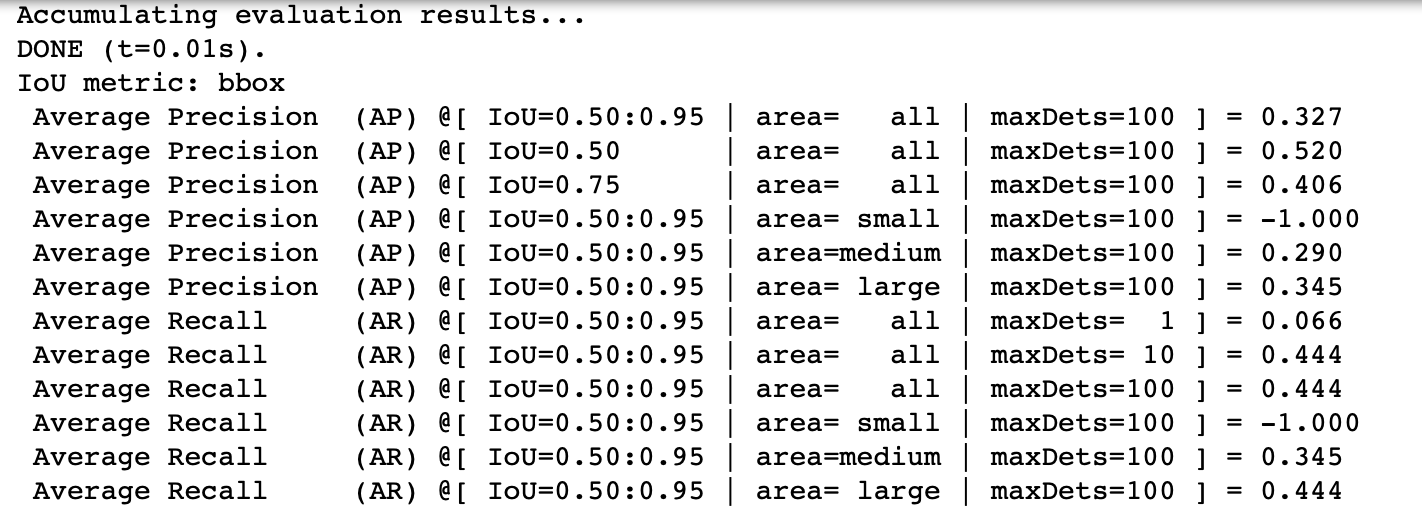


Рисунок 6. Промежуточные результаты обучения.

Состояния модели на каждой эпохе записывались в отдельный файл, чтобы состояние с самыми лучшими метриками можно было воспроизвести.

torch.save(model.state\_dict(), f"model\_{epoch}")

## Тестирование качества модели.

Промежуточное тестирование модели на каждой эпохе производилось с помощью валидационного набора данных, как показано на Рисунке 6. После десяти эпох обучения была выбрана эпоха, значение метрик на которой было наилучшим.

Для визуализации была дополнительно написана отдельная функция detect, которая принимает на вход обученную модель, изображение, на котором будут детектироваться дефекты и минимальный порог. Пример вызова функции:

detect(model, img, 0.5)

Ответ сети состоит из координат обрамляющих окон, меток классов для каждого окна и вероятностей, определяющих уверенность сети в том или ином окне. Если значение уверенности ниже минимального порога, обрамляющее окно не строится. Таким образом, порог служит для регулировки уверенности сети в том или ином обрамляющем окне.

Внутри функции исходное изображение нормализуется и приводится к формату, который требует нейронная сеть. Каждой метке класса сопоставляется название класса и цвет обрамляющего окна. Если изображение не содержит объектов детектируемых классов, функция возвращает исходное изображение. При наличии дефекта при помощи модуля ImageDraw библиотеки Pillow отрисовываются границы обрамляющего окна. На Рисунке 7 изображен пример работы функции detect.

На Рисунке 7 и на других изображениях сеть работает недостаточно хорошо, некоторые дефекты не помещаются в обрамляющие окна, а также совсем не определяется класс hole. Больше примеров работы модели после данном этапе работы можно посмотреть в Приложении.

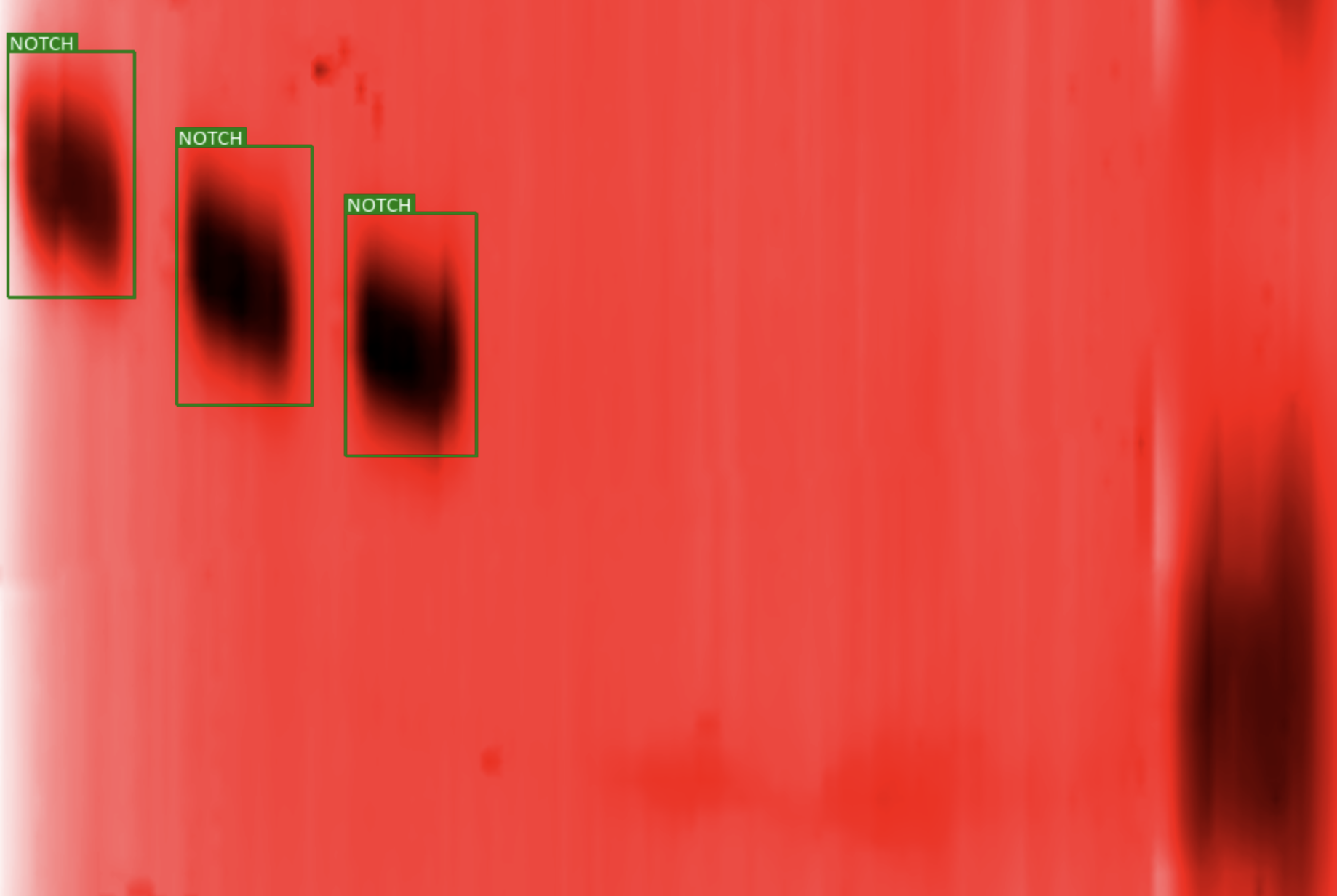
****

Рисунок 7. Пример работы функции detect.

## Аугментация данных.

Низкое качество детекции на данном этапе связно с тем, что набор данных для обучения очень мал.

Аугментация данных - способ создания дополнительных данных для обучения на основе существующих данных, с целью увеличения исходного набора. Чтобы достичь хороших результатов, глубокие нейронные сети должны быть обучены на очень большом объеме данных. Следовательно, если исходный обучающий набор содержит ограниченное количество изображений, для улучшения качества работы модели необходимо увеличить набор данных. Наиболее популярными вариантами аугментации являются следующие: горизонтальное отражение, случайная обрезка и изменение цветовой гаммы. Так же одновременно могут применяться различные комбинации, например, вращение и масштабирование.

В работе для аугментации данных использовалась библиотека albumentations, которая предоставляет множество способов аугментации для любых типов задач машинного обучения. [7] Пример применения библиотеки показан на Рисунке 8.

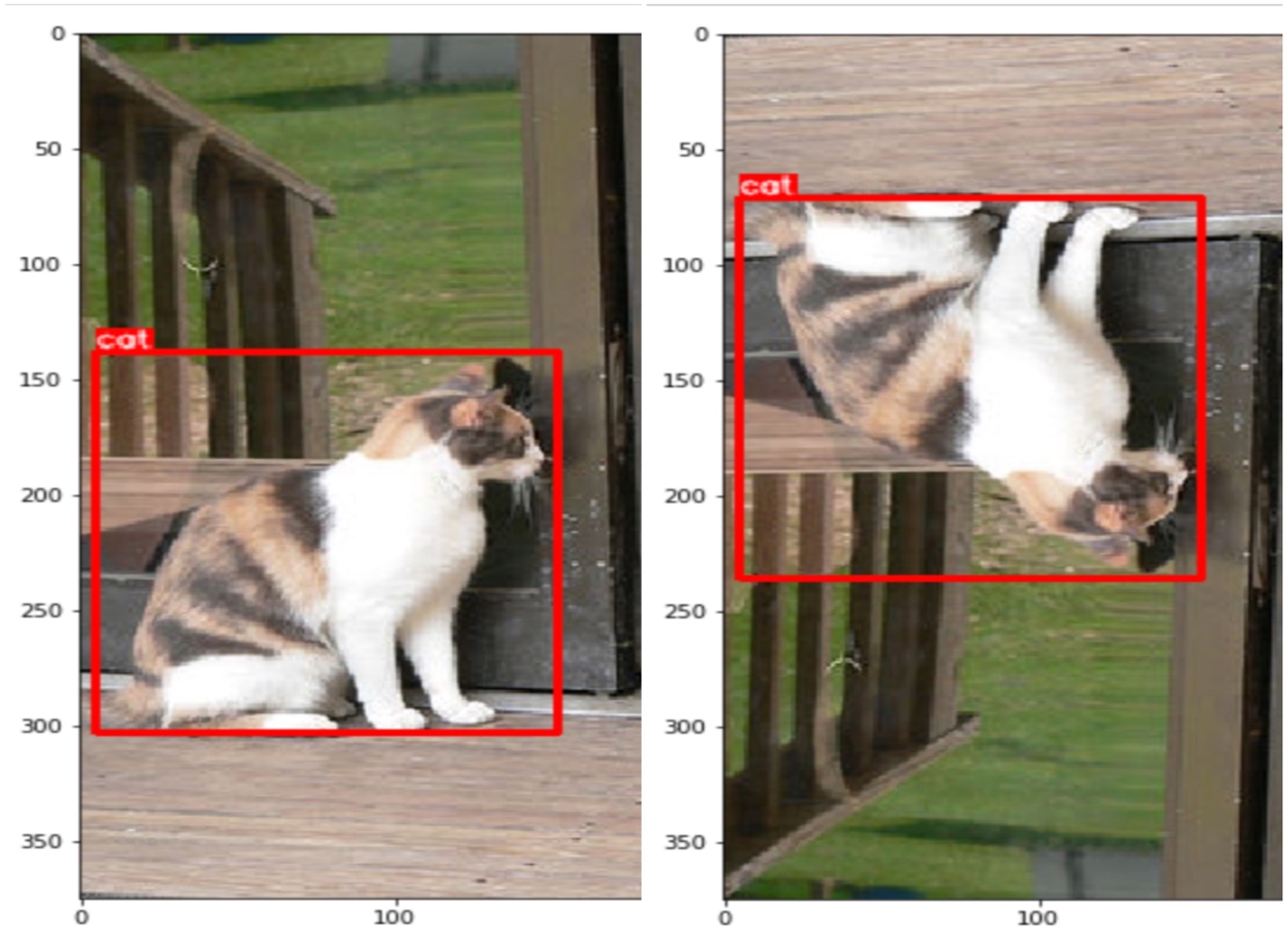


Рисунок 8. Пример работы albumentations.

В ранее определенный класс DefectDataset добавлен параметр augmentation. Если этот параметр принимает значение True, к исходному изображению применяются различные преобразования, такие как отражение, вращение и т.д, с определенной ранее указанной вероятностью. В работе примерялись такие преобразования:

* HorizontalFlip
* VerticalFlip
* HueSaturationValue
* RandomRotate90

С помощью

rand\_sampler = torch.utils.data.RandomSampler(dataset, num\_samples=1000, replacement=True)

из небольшого числа исходных изображений было получено 1000 изображений.

[1] LabelImg

[2] ImageNet

[3] https://github.com/shiyemin/voc2coco

[4] <https://ru.wikipedia.org/wiki/PyTorch>

[5] <https://ru.wikipedia.org/wiki/Стохастический_градиентный_спуск#cite_note-_e5afba02f6feec50-1>

[7] <https://github.com/albumentations-team/albumentations>

[6] https://github.com/pytorch/vision/blob/master/references/detection/engine.py