Pusan National University
Computer Science and Engineering
Technical Report 2023-10

철강 불량 검출 및 분류 인공지능 학습 모델 개발

저자1 강준우

저자2 김태경

저자3 여지수

지도교수 감진규

목 차

1. 서론

1.1. 연구 배경

대한민국은 철강산업과 함께 발전해 왔다고 말해도 과언이 아니다. 대한민국의 철강산업은 세계 조강생산량 6위(2022)를 차지하고 있으며, 대한민국 대표 철강회사인 포스코는 13년 연속 세계에서 가장 경쟁력있는 철강사 1위에 선정되기도 했다.

이러한 철강산업에서 가장 중요한 것은 무엇일까? 아마 품질이 먼저 떠오를 것이다. 그만큼 품질보증은 곧 철강회사의 자존심이자 신뢰 그 자체라 할 수 있다. 많은 회사가 품질을 보증하기 위해 불량률을 줄이고, 불량품 검출에 자본을 투자하고 있다.

철강은 압연 과정에서 결함을 검출하고 분류하는데, 이 과정에서 불량을 검출하지 못하고 후공정으로 넘어가면 품질 악화, 설비 사고 등이 일어날 수 있기 때문에 빠르고 높은 수준의 검출능력이 요구된다.[1]

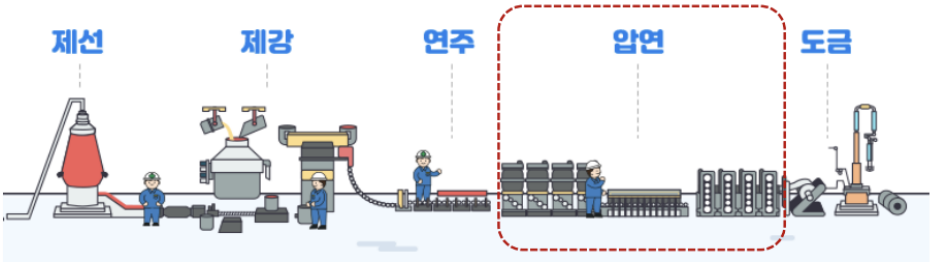


그림 1. 철강 제조 공정

압연 공정 중 철강 표면의 결함을 찾는 Surface Defect Detector(SDD)가 진행된다. SDD는 다음과 같은 과정으로 이루어지는데, 이 과정을 어플리케이션 환경으로 단순화 시켜 일반인들도 철강 결함 검출 과정을 한 눈에 이해할 수 있게 하고자 했다.

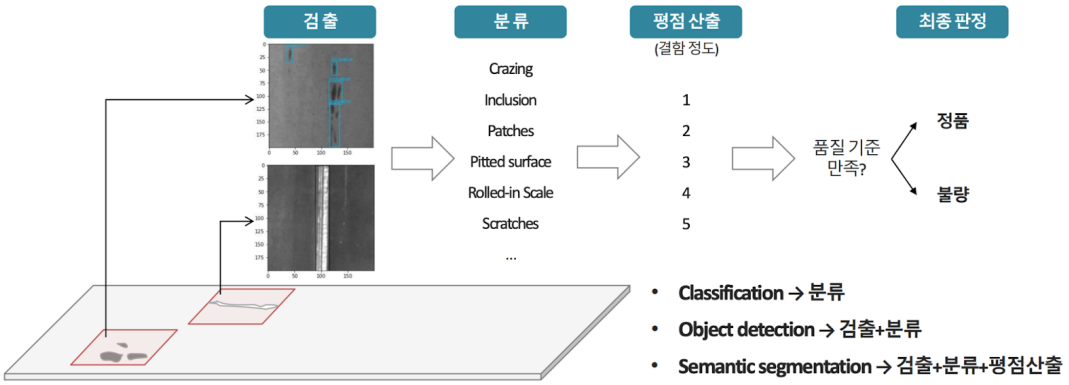


그림 2. 철강 표면 결함 탐지 과정

철강 이미지는 다른 이미지와는 다른 특성을 지닌다. 그래서 일반적인 이미지 기반 딥러닝 모델을 바로 적용할 수 없고, 직접 실험하지 않으면 어떤 딥러닝 모델이 철강 결함 이미지의 특성에 적합한지 알기 어렵다. 그리고 철강 표면의 결함 종류는 균일하게 발생하지 않는데, 이는 데이터 확보의 어려움과 불균형으로 인한 모델 학습의 질을 저하 시키는 주요 원인이 된다.

이번 연구에서는 다양한 딥러닝 모델을 분석하여 철강 결함 검출에 최적화된 학습 모델을 개발한다. Classification부터 Segmentation까지 각각 모델의 성능을 철강 이미지 특성에 맞게 개선시키고, 어플리케이션 환경으로 서비스한다. 또한 데이터 불균형으로 인한 부족한 이미지를 증강하는 모델을 연구해보고, 생성된 이미지의 품질을 향상 시키고자 한다.

1.2. 주요 문제점 분석

1.2.1. 철강 이미지 데이터의 불균형 문제

분류 성능을 위해서는 각 클래스 별로 고르게 분포되어 있는 데이터가 요구된다. 하지만 실제 공정에서 얻어지는 데이터는 특정 결함클래스가 편향되어 분포된 경우가 상당하다. 이러한 문제는 소량의 결함클래스가 다량의 결함클래스로 흡수되는 문제가 있으며, 불균형 데이터의 사용은 학습 결과 및 성능 저하의 요인이 된다.

1.2.2. 철강 데이터 특성의 문제

철강 이미지는 기본적으로 흑백이미지이며, 이미지의 형태가 일반적이지 않아 기존 모델로 학습하기 어렵다. 이를 철강 이미지의 특성에 맞게 모델을 수정하고, 전처리하여 성능을 향상 시키고자 한다.

1.3. 연구 목표

1.3.1. raw 이미지를 활용한 전처리 방법 개발

결함 위치를 나타내는 RLE 정보를 활용한 데이터 전처리 연구는 많이 선행되었다. 그러나 실제 산업 환경에서는 결함의 픽셀 값이 명시적으로 나타나는 경우가 드물다. 그렇기에 RLE

정보를 사용하지 않고 raw 이미지를 활용한 최적의 전처리 방법을 개발한다.

1.3.2. 소수 개체 데이터 증강 모델 개발

소수 개체의 철강 데이터를 확보하는 전처리 방법 및 철강 데이터의 특성을 반영한 최적의 GAN모델을 개발하고 정성적으로 평가하는 것을 목표로 한다.

1.3.3. 최적의 분류/검출 학습모델 개발 및 서비스

분류 및 검출 딥러닝 모델들을 비교하고, 철강 이미지 데이터의 특성에 맞게 성능을 개선시킨다. 이후 일반인도 철강 검출 과정을 쉽게 이해할 수 있도록 개발한 모델을 어플리케이션 환경으로 서비스한다.

2. 연구 배경

2.1. 생성 모델

2.1.1. DCGAN

GAN은 Generative Adversarial Network의 약자로 진짜 이미지를 학습해 가짜 이미지를 생성해내는 생성형 AI 모델이다. DCGAN은 이 GAN 구조를 딥러닝에서 효과적으로 사용할 수 있도록 CNN(Convolutional Neural Network)을 도입한 모델이다. 컨볼루션 레이어, 적절한 스트라이드 및 패딩, 배치 정규화, ReLU 활성화 함수, 그리고 시그모이드 출력 함수를 활용하여 안정적이고 고품질의 이미지를 생성한다. 생성자(G)와 판별자(D)가 경쟁적으로 이미지 생성과 판별을 하기 때문에 G-Loss값과 D-Loss값의 균형이 학습에 중요한 영향을 미친다.

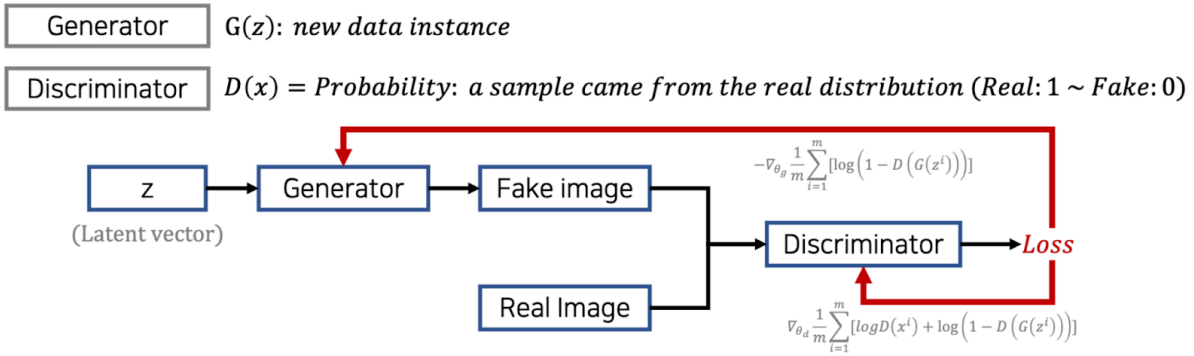


그림 3. DCGAN 생성자와 판별자의 역할

2.1.2. PGGAN

PGGAN의 PG는 Progressive Growing의 약자로 고해상도 이미지를 생성하는 데에 한계점을 개선한 생성형 모델이다. 고해상도의 이미지를 생성하도록 generator를 학습시키는 경우 학습 이미지의 distribution과 학습 결과 생성된 이미지의 distribution의 차이가 커진다. generator와 discriminator를 저해상도의 이미지로부터 고해상도의 이미지로까지 layer들을 추가하면서 점진적으로 커지게한다. 이를 통해 학습 속도를 향상시키고 고해상도에서도 안정적인 학습을 가능하게 한다.

2.2. 분류 모델

2.2.1. CNN

CNN (convolutional neural network)은 딥러닝은 한 종류로 주로 이미지를 인식하는데 사용된다. CNN은 수십 또는 수백 개의 계층을 가질 수 있으며, 각 계층은 영상의 서로 다른 특징을 검출한다. 밝기, 경계와 같이 매우 간단한 특징으로 시작하여 객체를 고유하게 정의하는 특징으로 복잡도를 늘려갈 수 있다.

2.2.2. ResNet

CNN에서 층이 깊어질 수록 학습이 잘 되지 않는 문제가 생겼고, ResNet은 이 문제를 해결하는 CNN 아키텍처이다. ResNet에서는 잔여 블록(residual block)을 이용한다는 것이 핵심이다. 실제로 학습하고자하는 매핑 함수인 H(x)를 곧바로 학습하는 것은 어려우므로 대신 F(x) = H(x)-x 를 학습한다. 이후 F(x)에 input값 x를 더함으로써 기존의 H(x)로 학습시키는 방식보다 훨씬 빠르고 정확하다.

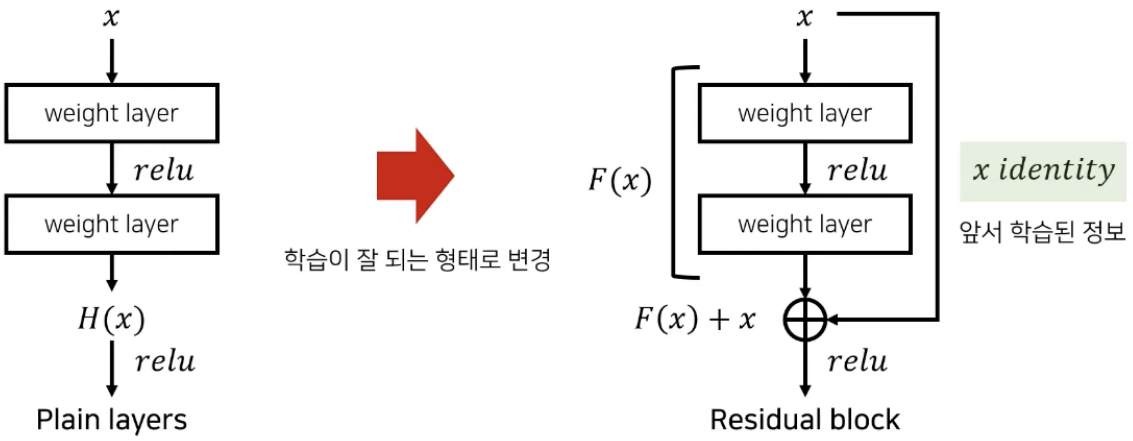


그림 4. CNN구조(왼쪽), ResNet구조(오른쪽)

2.2.3. Efficient Net

기존에 모델의 정확도를 높일 때, 일반적으로 수동으로 모델의 깊이, 너비, 입력 이미지의 크기를 조절했기에, 최적의 성능과 효율을 얻지 못했다. 이에 EfficientNet은 3가지를 효율적으로 조절할 수 있는 compound scaling 방법을 제안한다. 깊이, 너비, 입력 이미지 크기가 일정한 관계가 있다는 것을 실험적으로 찾아내고, 이 관계를 수식으로 만든다.

2.3. 세그멘테이션 모델

2.3.1. U-Net

U-Net은 Biomedical 분야에서 이미지 분할(Image Segmentation)을 목적으로 제안된 End-to-End 방식의 Fully-Convolutional Network 기반의 모델이다. 이미지의 전반적인 컨텍스트 정보를 얻기 위한 네트워크와 정확한 지역화(Localization)를 위한 네트워크가 대칭 형태로 구성되어 있다. Expanding Path의 경우 Contracting Path의 최종 특징 맵으로부터 보다 높은 해상도의 Segmentation 결과를 얻기 위해 몇 차례의 Up-sampling을 진행한다.

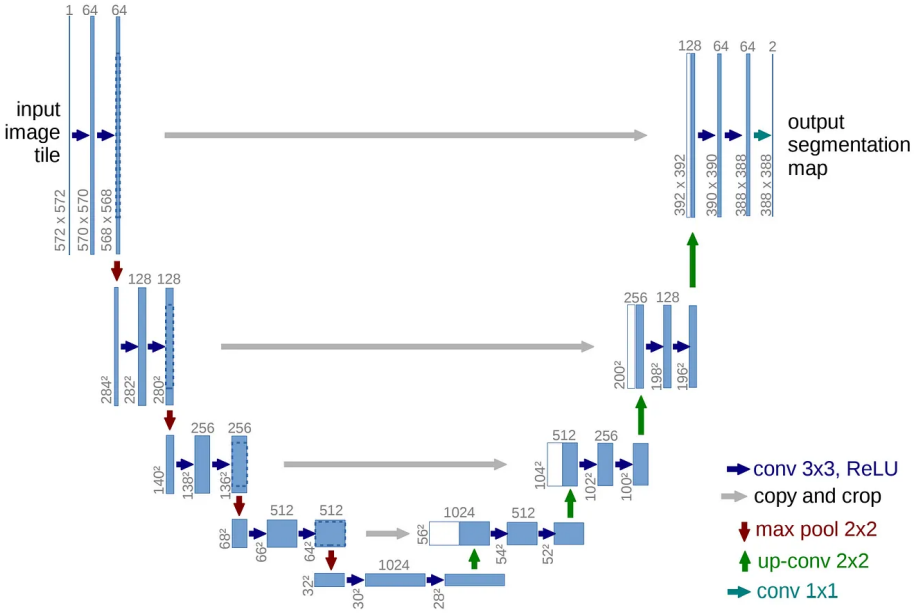


그림 5. U-Net의 구조

2.4. 기술 스택

2.4.1. Rest API

API는 애플리케이션이나 디바이스가 서로 간에 연결하여 통신할 수 있는 방법을 정의하는 규칙 세트이다. REST API는 REST(REpresentational State Transfer) 6가지의 디자인 원칙을 준수하는 API이다. HTTP 요청을 통해 통신함으로써 서버에 작성, 읽기, 업데이트 및 삭제 등의 표준 데이터베이스 기능을 수행한다.

2.4.2. Flutter

Flutter는 Google에서 개발 및 지원하는 오픈 소스 프레임워크이다. 개발자는 Flutter를 사용해 Android, IOS, Web 등 다수의 플랫폼에 대해 플랫폼에 관계 없이 사용자에게 일관된 시각적 경험을 제공한다. 이것이 가능한 이유는 플랫폼별 렌더링 도구를 사용하지 않고, Google의 자체 오픈 소스 그래픽 라이브러리를 사용하여 애플리케이션의 사용자 인터페이스(UI)를 렌더링하기 때문이다.

2.4.3. DBMS(MySQL)

MySQL은 오픈소스 관계형 데이터베이스 관리 시스템이다. MySQL은 전체 데이터베이스

구조를 변경하거나 기존 애플리케이션에 영향을 주지 않고 필요할 때마다 간편하게 테이블, 관계를 추가 또는 삭제하고 데이터를 변경할 수 있다.

2.4.4. Spring Data JPA

관계형 데이터베이스(MySQL 등)을 Java와 객체로 관리할 수 있도록 하는 JPA는 자바 진영에서 ORM(Object-Relational Mapping) 기술 표준으로 사용되는 인터페이스의 모음이다. JPA를 사용하면 데이터베이스를 객체로 다룰 수 있게 된다. 반복적인 입출력, 삭제가 발생하는 철강 검출 어플에서 유용하게 사용할 수 있다. 이번 연구에서 MySQL의 테이블과 객체를 Mapping하여 이미지 명세를 효율적으로 다루기 위해 사용하였다.

2.4.4. Docker

Docker는 컨테이너 기반으로 독립된 사용환경을 제공한다. 학과 AI서버도 Docker 컨테이너를 Kubernetes로 자동화하여 관리하고 있다. 이번 연구에서는 팀원간 라이브러리 등 환경세팅을 공유하기 위해 사용하고, MySQL서버를 독립된 환경에서 운용하기 위해 사용하였다.

3. 연구 내용

3.1. 개발 일정 및 역할

3.1.1. 개발 일정

개발 구분	세부 항목	5				6					7				8					9			
		1	2	3	4	1	2	3	4	5	1	2	3	4	1	2	3	4	5	1	2	3	4
기획	주제 선정																						
	사전 자료 조사																						
인공지능 모델 개발	콘다 가상 환경 세팅																						
	초기 모델 개발																						
	Raw 데이터 전처리																						
	후보 모델별 테스트																						
	모델 오차 분석																						
	모델 고도화 및 평가																						
	모델 추출 후 배포																						
DB 설계	DB 설계 및 구축																						
REST API 개발	DB와 모델 연동, 기능 개발																						
	REST API 배포																						
Application 개발 및 배포	REST API 연동, 기능 개발																						
	Application 배포																						

표 1. 상세 개발 일정

3.1.2. 구성원별 역할

이름	역할 분담
강준우	- 어플리케이션 제작(Flutter) - DBMS(mysql) 서버 구축 - DB 연동 Rest API(Spring JPA) 개발
김태경	- 데이터 분석 및 전처리 - 이미지 증강 모델(DCGAN, PGGAN) 연구 - 이미지 세그멘테이션 모델(U-Net) 연구 - 모델 학습을 위한 쿠버네티스 도커 환경 연구 - DBMS(mysql) 서버 구축 - DB 연동 Rest API(Spring JPA) 개발
여지수	- 데이터 분석 및 전처리 - 이미지 분류 모델(CNN, ResNet, EfficientNet) 연구 - 이미지 세그멘테이션 모델(U-Net) 연구 - 이미지 증강 모델 (DCGAN) 연구 - 모델 서빙 Rest API(Flask) 개발 - figma를 활용하여 UI 및 UX 디자인

표 2. 구성원별 상세 역할

3.2. 개발 환경 구축

3.2.1. 학과 AI서버 분석(쿠버네티스-도커 환경)

기존 Anconda를 통해 학과서버를 접속하여 사용하였고, 이후 학과서버가 Docker기반 AI 서버로 바뀌면서 새롭게 개발 환경을 구축하고자 했다.

학과 AI서버는 Kubernetes(쿠버네티스)를 통해 Pod를 생성하여 컨테이너를 관리한다. 여기서 Pod는 쿠버네티스의 작업 단위이며, 컨테이너는 독립된 하나의 가상화 환경이다. 정리하면 관리자(학과서버 관리자)는 쿠버네티스를 통해 자동으로 컨테이너의 생성주기를 관리하고, 사용자(여기서는 자두과자 팀)는 생성 규칙에 맞게 컨테이너를 생성하고 GPU사용 권한을 획득하여 사용한다. 작성한 AI 모델을 학습 데이터와 함께, GPU사용을 허가 받은 컨테이너에서 학습할 수 있다.

컨테이너는 Docker(도커)를 이용하여 생성하는데, 도커는 Docker File을 통해 생성하고자

하는 Docker Image(도커 이미지)의 명세를 작성할 수 있다. 이 명세를 기반으로 도커 이미지를 만들고 이 도커 이미지를 빌드하여 도커 컨테이너를 만든다. 도커 이미지는 틀과 같은 역할이며, 이 틀을 찍어서 만든 인스턴스가 컨테이너다. 기본적으로 Nvidia에서 Cuda코어를 이용할 수 있도록 nvidia/cuda 도커 이미지를 제공한다. 학과서버에서는 기본 옵션으로 **nvidia/cuda:11.4.1-base-ubuntu20.04**를 사용하여 컨테이너를 생성하고 있다. 학과 Kubernetes 작업 생성 명령어 실행시, 옵션은 다음과 같다.

-i : 사용할 도커 이미지 지정. 미지정시 기본값은 "nvidia/cuda:11.4.1-base-ubuntu20.04"로 되어 있다.

-g : 사용할 GPU 개수. 미지정시 기본값은 1

-n : 생성할 컨테이너 이름. 컨테이너 이름은 중복될 수 없고, 미지정시 '계정명 - autocreate - 랜덤숫자'로 작성된다.

-c : 실행할 명령어. 실행할 디렉토리로 이동 후 명령어 작성.

```
python3 k8s_create_job.py
-i nvidia/cuda:11.4.1-base-ubuntu20.04
-g 2
-n snack-pgan-1
-c "cd /home/snack && defaultpgan.py"
```

그림 6. 학과 AI서버 Kubernetes 명령 규칙

3.2.2. 시행착오

첫번째 문제는 **nvidia/cuda:11.4.1-base-ubuntu20.04**를 사용하여 Pod생성시 ImagePullBackOff가 발생했다. 이는 이미지를 가져올 수 없는 경우 발생하는 에러였다. 이를 nvidia 공식 GitLab에서 확인한 결과 더이상 온라인 배포가 되지 않아 자동으로 가져올 수 없어서 발생했다. 현 학과 서버 GPU 드라이버와 호환 가능한 11.4.3을 다운 받아 사용했고, 성공적으로 컨테이너를 생성할 수 있었다.

```
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  24s   default-scheduler  Successfully assigned snack/snack-pgan-3 to gpu-2
  Normal  Pulling    22s   kubelet            Pulling image "nvidia/cuda:11.4.3-base-ubuntu20.04"
  Normal  Pulled     15s   kubelet            Successfully pulled image "nvidia/cuda:11.4.3-base-ubuntu20.04" in 6.724849567s
  Normal  Created    15s   kubelet            Created container gpu-container
  Normal  Started    14s   kubelet            Started container gpu-container
```

그림 7. 생성한 Pod의 세부 단계

두번째 문제는 컨테이너는 생성되었지만, 명령어가 정상적으로 실행되지 않았다. 원인은 nvidia/cuda 이미지에는 python3, pytorch등 학습에 필요한 라이브러리가 기본 제공되지 않기 때문이었다. 쿠버네티스 Pod를 통해 컨테이너를 관리하기 때문에 생성된 컨테이너에 접속하여 사용할 수 없고, 초기 명령어만 실행하고 pod가 종료되기 때문에 이미지에 관련 라이브러리가 모두 설치되어 있어야 한다.

우리가 사용하고자 하는 라이브러리 및 환경을 모두 가진 nvidia/cuda 이미지를 찾을 수 없어 직접 docker file을 작성하였다.

```
FROM nvidia/cuda:11.4.3-base-ubuntu20.04

# 빌드중 상호작용방지
ENV DEBIAN_FRONTEND=noninteractive

# lib설치 : 필요한 라이브러리를 추가한다.
RUN apt-get update && apt-get install -y \
    libsm6 \
    libxext6 \
    libxrender-dev \
    python3-pip \
    python3-pandas \
    python3-seaborn \
    python3-tqdm \
    python3-scipy \
 && apt-get clean \
 && rm -rf /var/lib/apt/lists/*

# Python 패키지 설치
RUN pip3 install numpy opencv-python-headless matplotlib seaborn pillow keras scikit-learn torch torchvision

# 철강데이터
COPY ./dataset /app/dataset

# 작업 디렉터리
WORKDIR /app
```

그림 8. 직접 작성한 Docker file

base image를 기반으로 필요한 라이브러리를 작성하였고, pod생성시 이미지 파일을 추가할 수 없기 때문에 학습에 필요한 철강 데이터는 작업 디렉터리에 함께 첨부하여 빌드하였다. 커스텀한 이미지는 도커 허브[1]에서 확인할 수 있다.

이후 커스텀한 도커 이미지를 통해 Pod를 생성하였지만 에러가 발생했다. kubectl describe

[1] Docker Hub : tigerfriend/repository

pod 명령어로 분석해도 원인을 찾을 수 없었고, 학과 서버 환경 설정을 확인할 권한이 없었다. 또한 Pod 생성을 통해 GPU 사용 권한을 획득해야 학습할 수 있기 때문에 서버에서 직접 컨테이너를 생성해서 학습할 수는 없다.

하지만 실행이 된다고 하더라도, 매번 GB단위의 이미지 데이터를 도커 이미지로 빌드하여 실행하기에는 빌드에 지나치게 많은 시간이 소요되었고, 데이터 set을 수정할 때마다 새롭게 빌드해야 했다. Pod 생성 시 초기화 컨테이너(Init Container)로 데이터 set을 저장한 컨테이너를 만들 수 있지만 데이터 set을 자주 수정하기 때문에 데이터 수정에 용이한 Google Colab을 사용하여 학습하기로 하였다.

3.2.3. 정리

비록 학과 AI 서버를 제대로 이용하지 못했지만, 서버 환경을 분석하고 에러를 해결하기 위해 연구하는 과정에서 자동화 컨테이너 관리와 Docker 환경에 대해 깊이 있게 학습할 수 있었다. 이 경험을 토대로 이번 철강 불량 검출 및 분류 모델 서비스를 운용하는 서버를 Docker 환경에서 구축하여 관련 지식을 활용하였다.

3.3. 데이터 분석

사용한 데이터셋은 kaggle의 Severstal: Steel Defect Detection 이다. 데이터셋에는 test_images 폴더, train_images 폴더, 그리고 train.csv로 구성되어있고, 각 폴더 안에는 철강 이미지 파일들이 들어있다. 한 장의 이미지는 1600x256 사이즈이고, train image는 12600장, test image는 5506장으로 구성되어 있다. train.csv 파일에는 train image에 대한 정보가 적혀 있다. 총 세가지 칼럼이 있는데, ImageId에는 이미지 파일명이 적혀 있고, ClassId에는 결함의 라벨이, EncodedPixels에는 어떤 픽셀에 손상부위가 있는 것인지 encoding 되어 있다. 결함의 라벨로는 총 4종류 (결함1, 결함2, 결함3, 결함4)가 제공된다.

ClassId 별로 결함 개수는 그림9와 같이 파악된다. 세로축에는 결함의 label이 표시되어있다. 두가지 이상의 label이 들어있는 것을 보아, 데이터셋에는 다중 결함 이미지가 포함되어있다. 그리고 결함3은 4759개인 반면, 결함 2는 195개로 데이터 불균형이 심하다는 것을 파악할 수 있다.

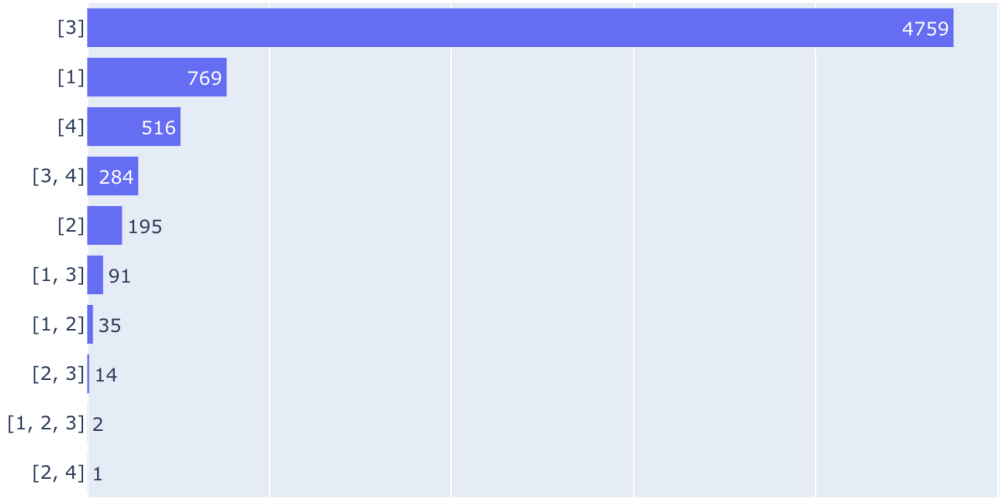


그림 9. Label 별 이미지 데이터 개수

표 3은 결함 유형별 이미지를 정리한 표이다. 결함이 있는 부분들에 대해 하나씩 살펴보고 결함의 유형들을 분석해보았다. defect 1은 pitted surface로 표면에 작은 구멍이나 홈이 생기는 결함 유형이다d. defect 2는 crazing으로 표면 위의 매우 작고 얕은 균열이다. defect 3은 scratches로 표면에 무언가가 긁혀서 생긴 선 모양의 손상이다. defect 4는 patches로 표면이 부풀어 오르고 울퉁불퉁하게 돌출되어 있는 등 산화물 조합에 불량이 있는 결함 유형이다.

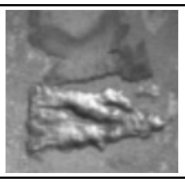
defect 1	defect 2	defect 3	defect 4
			

표 3. 결함 유형별 이미지

표 4는 결함이 없는 철강 표면들에 대해 각각 정리한 표이다. 다양한 형태의 철강 표면이 존재한다는 것을 파악할 수 있었다. 첫번째, 두번째 표면은 주름 도장 기술이 처리된 것으로 추정된다. 첫번째 강판의 표면에는 일정한 간격으로 돌기가 위치해 있으며, 두번째 강판에는 일정한 간격으로 다이아몬드 형태의 기하학적 패턴이 존재한다. 그리고 강판의 군데 군데에 검은색이 보이는데 이는 산화 층 (Oxide Layer)이나 블루잉(Bluing)에 의한 것일 수 있다. 모두 결함이 없는 표면의 이미지만 가져온 것이고, 이러한 형태를 결함으로 인식되어 학습되지 않도록 주의해야할 것이다. 또한 증강 모델을 활용하여 이 다양한 표면들을

학습시켜 데이터를 추가로 확보해야 할 것이다.

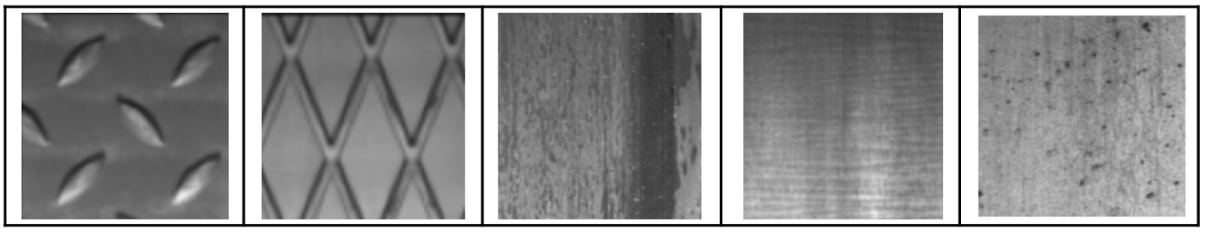


표 4. 철강 표면 유형별 이미지

3.4. 생성 모델

3.4.1. DCGAN

- 모델 구조

가중치 초기화 및 생성자(G), 판별자(D) 구조는 Ian Goodfellow의 DCGAN논문[2][3]을 참고하였고, 가중치 초기화 값은 논문에서 제시한 mean = 0, standard deviation = 0.02인 정규분포로 무작위로 초기화 하여 진행하였다. 학습시간을 고려해 먼저 입력 64x64이미지로 생성자는 합성곱 계층마다 기울기 소실문제를 해결하기 위해 배치 정규화와 ReLu 활성함수를 한 쌍으로 묶어서 배치하였다. 작성한 모델과 파라미터를 구조화 하면 다음과 같다.

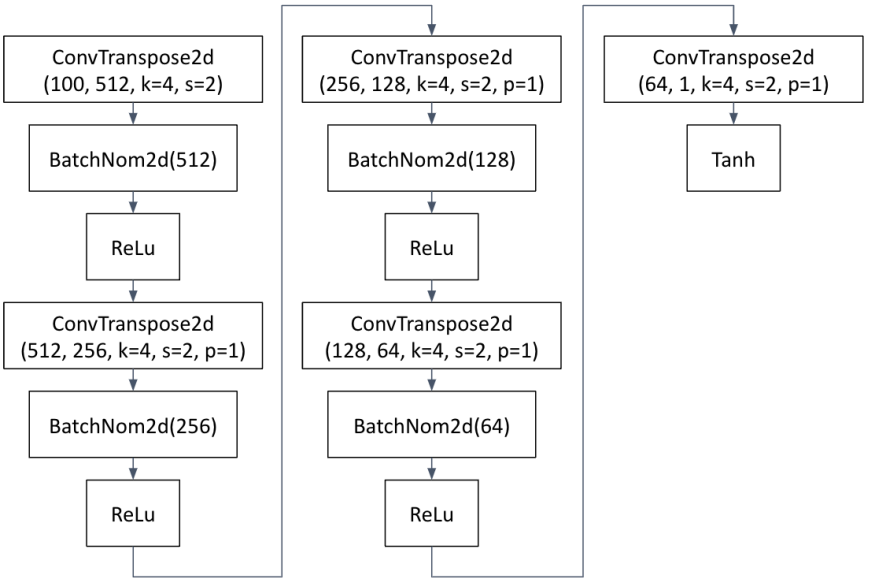


그림 10. 생성자(G)구조 (64x64)

판별자는 배치 정규화와 LeakyReLu를 계층 사이 마다 배치하였다.

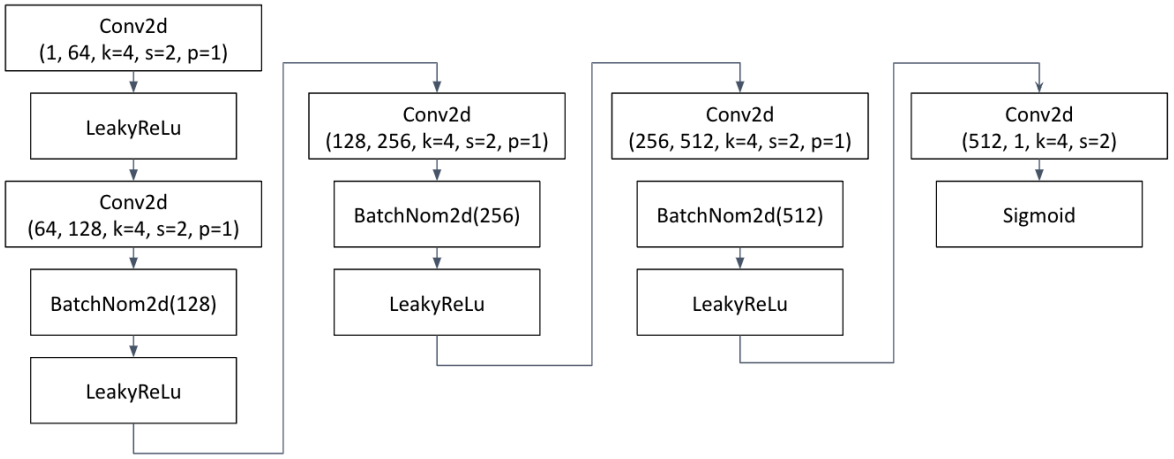


그림 11. 판별자(D)구조 (64x64)

작성한 모델이 제대로 작동하는지 전처리 없이 전체 철강 데이터를 학습시켜 테스트 해보았다.

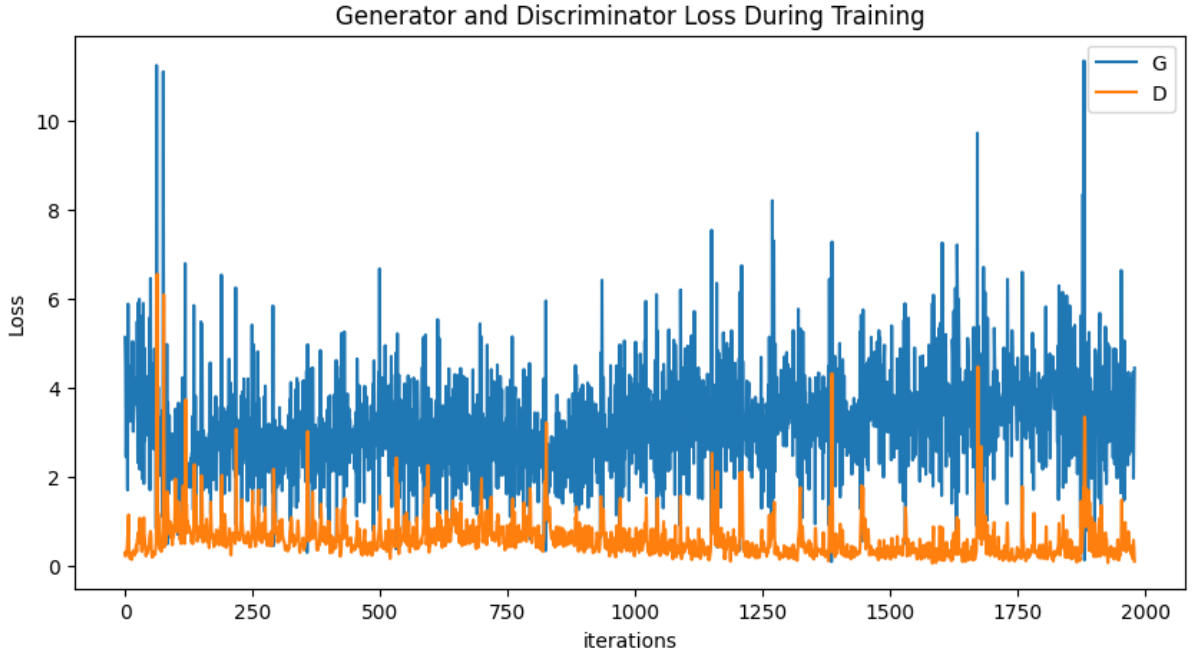


그림 12. G와 D의 Loss값 (전처리x, epochs 50)

G-loss는 생성자가 만든 가짜 데이터가 판별자에 의해 얼마나 진짜로 인식되는지를

측정하며, 이 값이 높을수록 생성자가 만든 가짜 데이터가 가짜같다는 의미로 잘 만들지 못하는 경우다.

D-loss는 판별자가 얼마나 잘 진짜 데이터와 가짜 데이터를 구분하는지를 측정하며, 이 값이 낮을수록 판별자가 잘 구분하고 있다는 것을 의미한다. 하지만 속여야 의미 있는 학습이 되기에 너무 낮으면 학습이 제대로 되지 않는다.

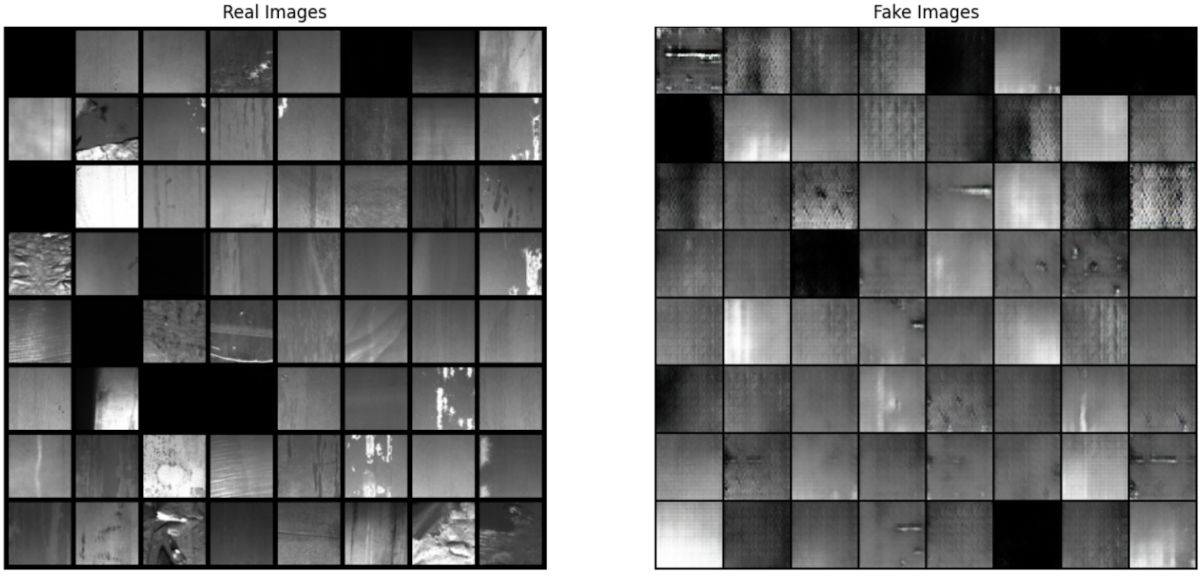


그림 13. 전체 이미지로 학습한 생성 결과(전처리X, epochs 50)

학습 횟수가 낮아 제대로 생성되지 않았고, 생성 이미지(Fake Images)에서 노이즈가 눈에 띄게 보인다.

이후 교수님께서 주신 피드백(결함 별로 나눈 후 데이터가 적은 클래스 증강, epochs 횟수 증가)을 반영하여 defect label별로 데이터를 분류하고 전처리 하면서 Loss값 분석과 생성결과를 확인하였다.

- 전처리 및 fake 이미지 생성

먼저, 클래스 결함별로 이미지를 정리하였고 결함 이미지별로 DCGAN 학습을 진행했다. 하지만 Loss값이 극단적이 되어 생성자는 제대로 가짜이미지를 만들지 못하고, 판별자가 가짜 이미지를 너무 잘 분별하였다. 또한 의미있는 이미지 생성에도 실패했다.

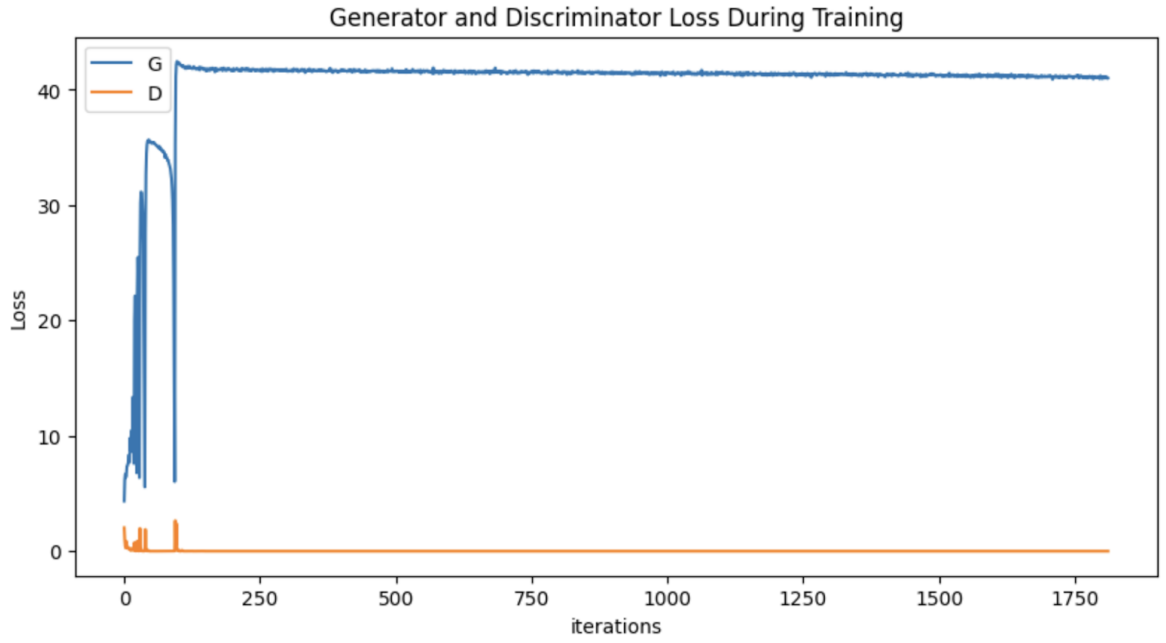


그림 14. Loss그래프 : G와 D의 균형이 깨진 상태

원인을 분석한 결과, 1600x256 크기의 많은 철강 이미지가 빈 공간(검정색 부분)을 포함하고 있었다. 그렇기 때문에 이를 바로 학습하면 검정 배경이 학습될 뿐만 아니라, resize 과정에서 유의미한 데이터가 소실된다.

그림 15. 2번 class 결함 이미지 : 1600x256

또한 defect label 2번, 4번 데이터는 각각 196, 516장으로 4759장인 label 3번에 비해 부족한 양이다. 제대로 학습하기 위해선 충분한 양의 의미 있는 데이터를 확보해야했고, 의미 있는 데이터를 추출과 부족한 데이터 확보 문제를 다음 세 단계로 해결하였다.

1) 1600x256 이미지를 266x256으로 6개로 자른다.
2) 자른 이미지의 임계값을 계산하여 일정 임계값 이하의 이미지를 제외한다. (검정 이미지 삭제)
3) 통과한 이미지를 256x256으로 resize한다.

단순히 6개로 나누면 의미없는 검정 배경 이미지가 생성된다. 이를 grayscale을 기준으로

픽셀값 0(검정)~255(흰색) 중 평균 10이하의 값을 가지는 이미지를 검정 이미지로 판별하여 제외하였다. 임계값을 10으로 정한 이유는 노이즈로 인해 0이 아니지만 0에 가까운 값을 가지는 이미지를 검출하기 위해 실험을 통해 적정값을 설정했다. 전처리에 사용된 모든 코드는 utils.py로 따로 관리하여 한번의 코드로 다른 label이미지를 전처리 할 수 있도록 하였다.

이를 통해 데이터를 약 3~5배 가량 확보함과 동시에 불필요한 검정 배경 이미지를 제거할 수 있었다.

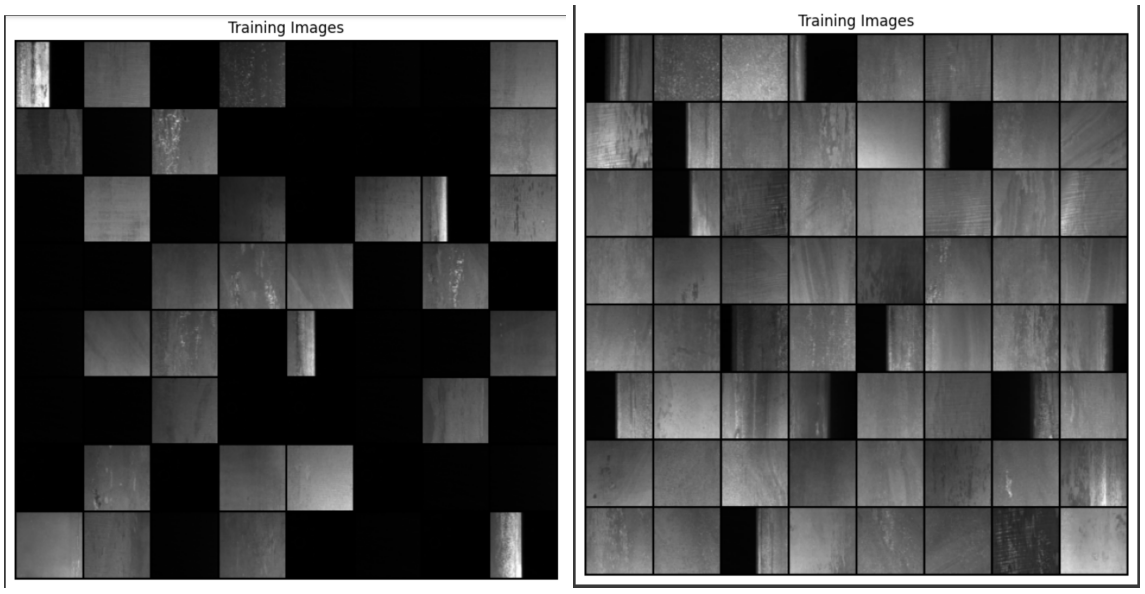


그림 16. 2번 class 결함 이미지 전처리 전/후

이후 소수 개체 데이터인 defect 1, 2, 4 Label을 각각 학습하였고, Label별 결함 특징을 담은 이미지를 생성할 수 있었다. 특히 Label 4에서 눈에 띄는 성과를 확인할 수 있었다.

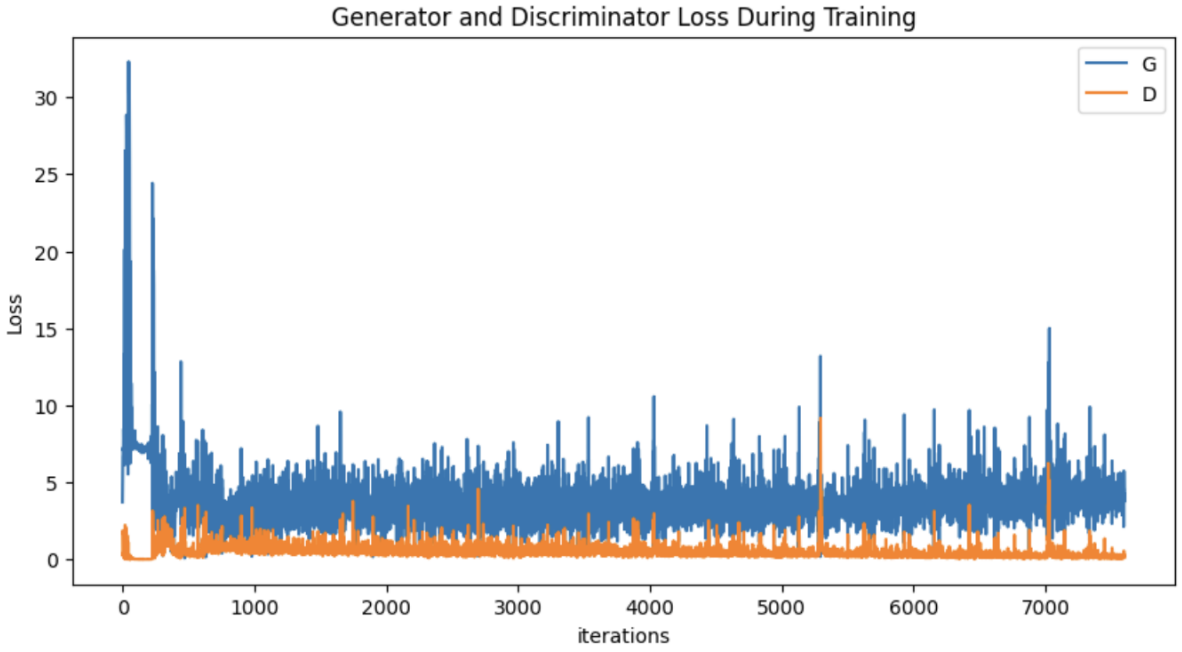


그림 17. 4번 클래스 결함 Loss 그래프 (epochs :300)

그래프를 보면 D와 G가 서로 한쪽이 압도하지 않고 균형있게 경쟁하는 모습을 볼 수 있다. 그래프가 안정적일 수록 fake 이미지는 real 이미지에 가까웠다.

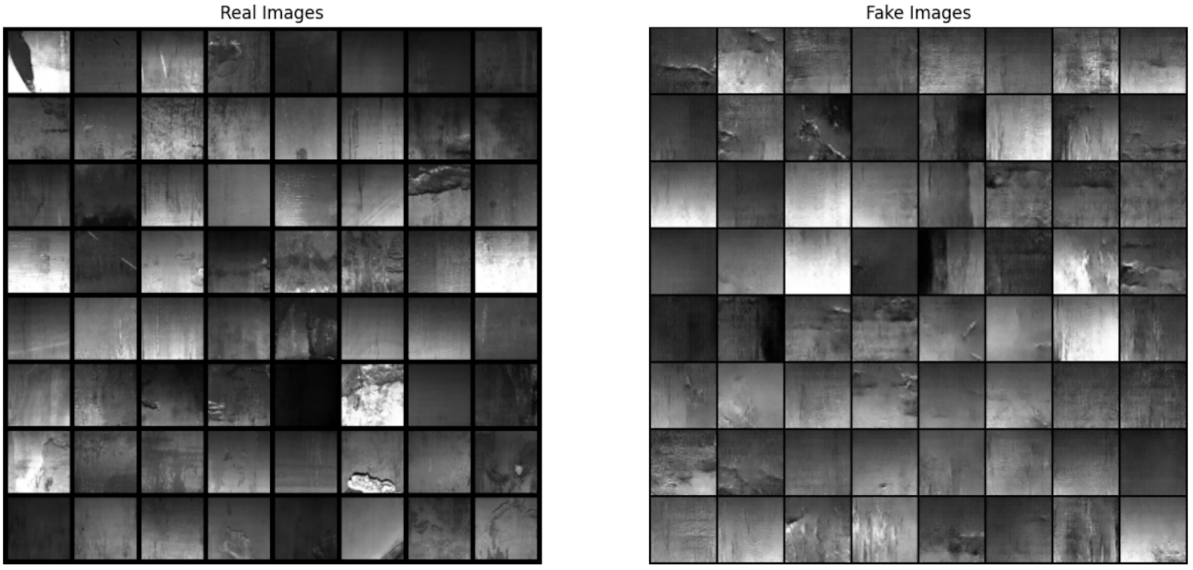


그림 18. 4번 class 결함 이미지 Real/Fake (epochs : 300)

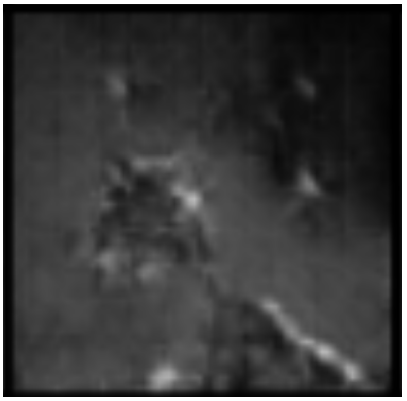


그림 19. 4번 class 결함 Fake이미지 (epochs : 300)

생성한 fake 이미지는 real 이미지의 특징을 상당히 잘 담고 있었고, 4번 결함의 특징인 Patches처럼 표면처리가 되어 있는 것을 확인할 수 있다. 이를 통해 부족한 철강 데이터 확보의 가능성을 확인할 수 있었다. 하지만 64x64 이미지로 생성했기 때문에 바로 학습 모델에 사용하기에는 해상도가 낮아 정확한 결함을 학습하기엔 어렵다. 그래서 모델을 한 단계 발전시켜 128x128 이미지를 생성하도록 수정하였다.

- 생성 이미지 해상도 향상

해상도를 올리기 위해선 하이퍼 파라미터 조절과 계층을 수정해야한다. 파라미터를 64x64와 동일하게 가져갈 수 없기에 여러 실험이 필요했고, 생성자와 판별자의 계층을 추가한 후 실험을 통해 최적의 파라미터 값을 찾아나갔다.

128x128 해상도의 이미지를 얻기 위해 CNN구조를 기반으로 생성자와 판별자를 구성하기 때문에 Layer를 각각 하나씩 추가하였다. 하지만 batch사이즈와 채널 크기를 적절히 조절하지 않으면 Loss그래프가 한쪽으로 치우쳐서 제대로 값이 나오지 않았다. 실험 중 ngf(생성자 채널 크기)와 ndf(판별자 채널 크기)의 값 조절이 학습에 영향을 유의미한 영향을 줬는데, 아래는 채널값 비율을 ngf/4 = ndf로 설정시 얻은 결과 값이다.

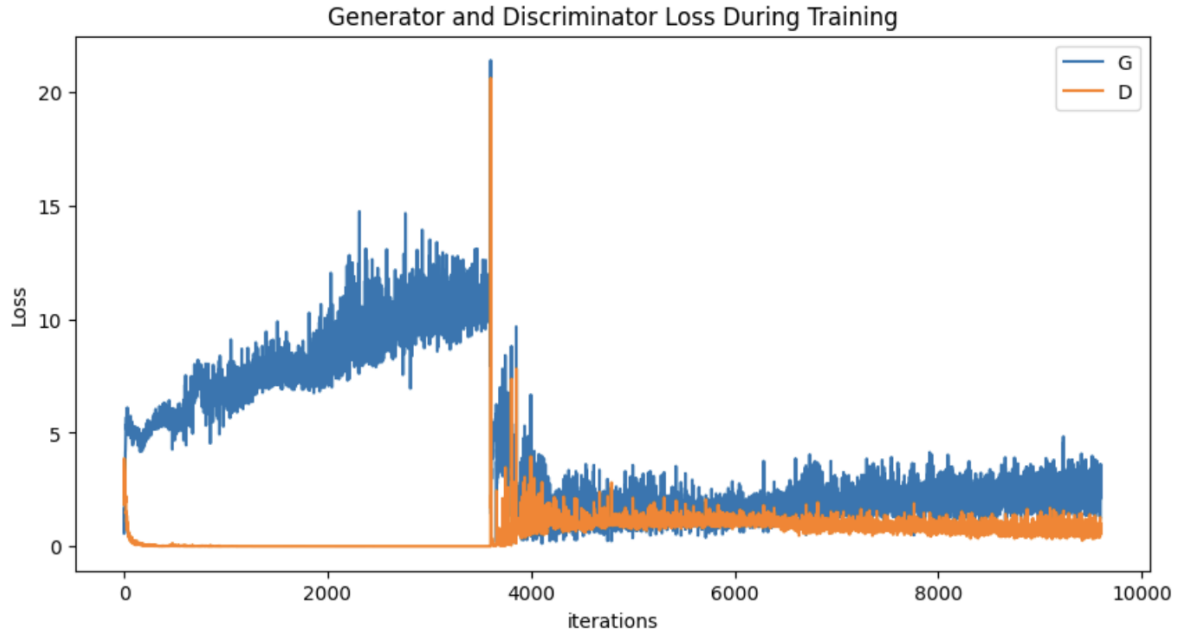


그림 20. 4번 class 결함 Loss 그래프 (epochs : 400)

ngf/4 = ndf 값으로 학습시 약 150 epochs까지는 Loss값이 불안정 하였고, 이후 안정됨을 확인할 수 있었다.

128x128이미지 생성에서는 기존 값인 ngf=ndf에서 학습에 실패했기 때문에, 생성자 대비 판별자의 채널 값을 줄이는 방법이 안정적인 학습을 진행하게 하였다. 이는 일반적으로 생성자에 비해 판별자가 승리(가짜 이미지 판별)하기 쉬운 것이 이유로 추측된다. 다만, 64x64 이미지 입력과는 다르게 적은 epochs로는 학습이 잘 이루어지지 않아서 epochs 200에서 1000으로 증가시켜 학습하였다.

표 5는 epochs 200부터 1000까지 생성된 이미지를 나열한 것이다. 충분히 많은 epochs가 필요함을 알 수 있다.

표 5. 4번 class 결함 epochs별 생성 변화

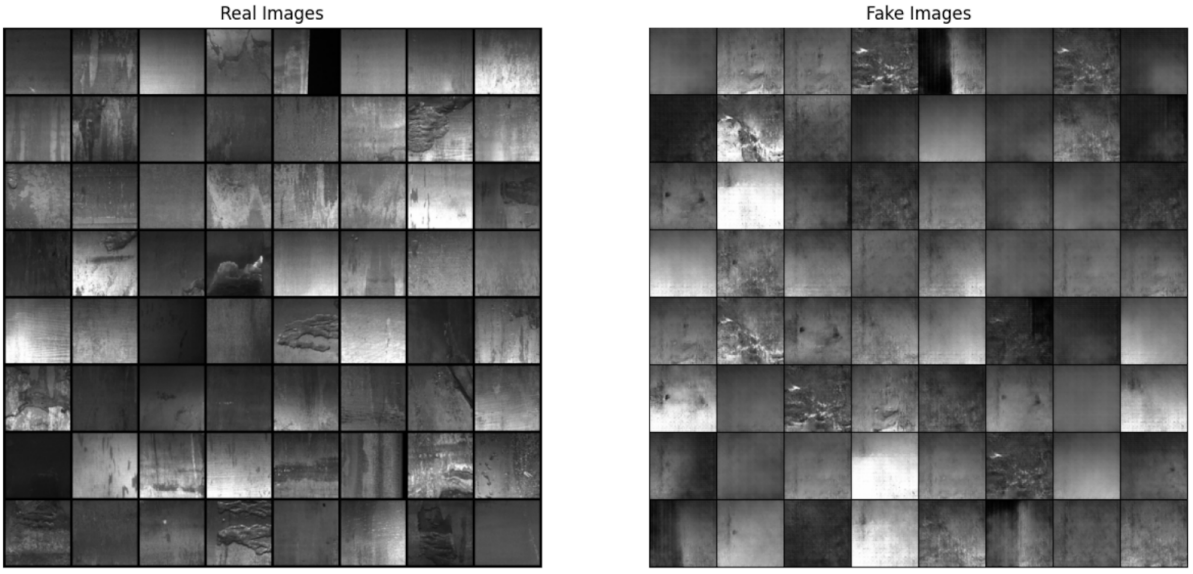


그림 21. 4번 class 결함 이미지 Real/Fake (epochs : 1000)

이렇게 DCGAN을 이용해 64x64 이미지, 128x128로 소수 개체 이미지를 활용하여 fake 이미지를 생성해 보았다. 이 연구를 통해 적은 이미지로도 GAN모델을 활용하여 이미지 생성할 수 있는 가능성을 확인하였다.

DCGAN은 해상도를 올리기 위해 조정해야할 파라미터가 많고 학습이 어려워 고해상도 이미지 생성에는 무리가 있다고 판단된다. 이에 낮은 해상도부터 학습하여 고해상도의 이미지를 생성하는 PGGAN을 연구해 보았다.

3.4.2. PGGAN

- 모델 구조 및 전처리

PGGAN은 DCGAN에서 가장 성능이 좋았던 다음 세 단계 전처리를 사용했다.

1) 1600x256 이미지를 266x256으로 6개로 자른다.
2) 자른 이미지의 임계값을 계산하여 일정 임계값 이하의 이미지를 제외한다. (검정 이미지 삭제)
3) 통과한 이미지를 256x256으로 resize한다.

위 전처리 방식이 원본 철강이미지의 손상이 가장 적으면서 생성모델에서 다루기 좋았다. PGGAN은 저해상도부터 학습하고 이 학습결과를 반영하여 해상도를 단계별로 상승시키는 모델이다. DCGAN의 128x128해상도 보다 높은 해상도인 256x256 크기의 이미지를 얻기 위해 설계하였다.

DCGAN을 연구하면서, 생성모델 이미지 품질과 학습률을 확인 하는 단서 중 하나는 G-Loss와 D-Loss를 안정적으로 유지시키는 것이다. 생성자와 판별자의 생성과 판별이 도전적인 난이도를 가지지만 균형이 깨지지 않아야 향상된 이미지를 생성할 수 있었다. 이에 PGGAN은 생성자와 판별자를 4x4의 낮은 해상도에서 생성 및 판별을 시작해 Loss를 안정적으로 유지하며 다음 단계로 성장하는 방식이다.

PGGAN 논문[4]을 참고하여 구조와 활성 함수를 설정했다. 다만, 철강 이미지는 흑백이기 때문에 RGB채널을 Grayscale에 맞게 in_channel 값을 변경하고 추가적인 전처리를 진행하였다.

PGGAN은 train데이터와 valid데이터가 필요하여 8:2로 분할하였고, 소수 개체 데이터 중 defect label 4를 사용하여 학습을 진행하였다. PGGAN은 각 해상도 별로 epochs 30으로 학습하였다.

- 실험 결과

학습 결과 낮은 해상도에서는 G와 D가 1 미만의 Loss값을 가졌다. 하지만 32x32 epochs 25에서 갑자기 G-Loss가 0으로 떨어지고 D-Loss가 50으로 상승하였다. 이후 이 값이

고정되어 나타났고, 학습에 실패했다.

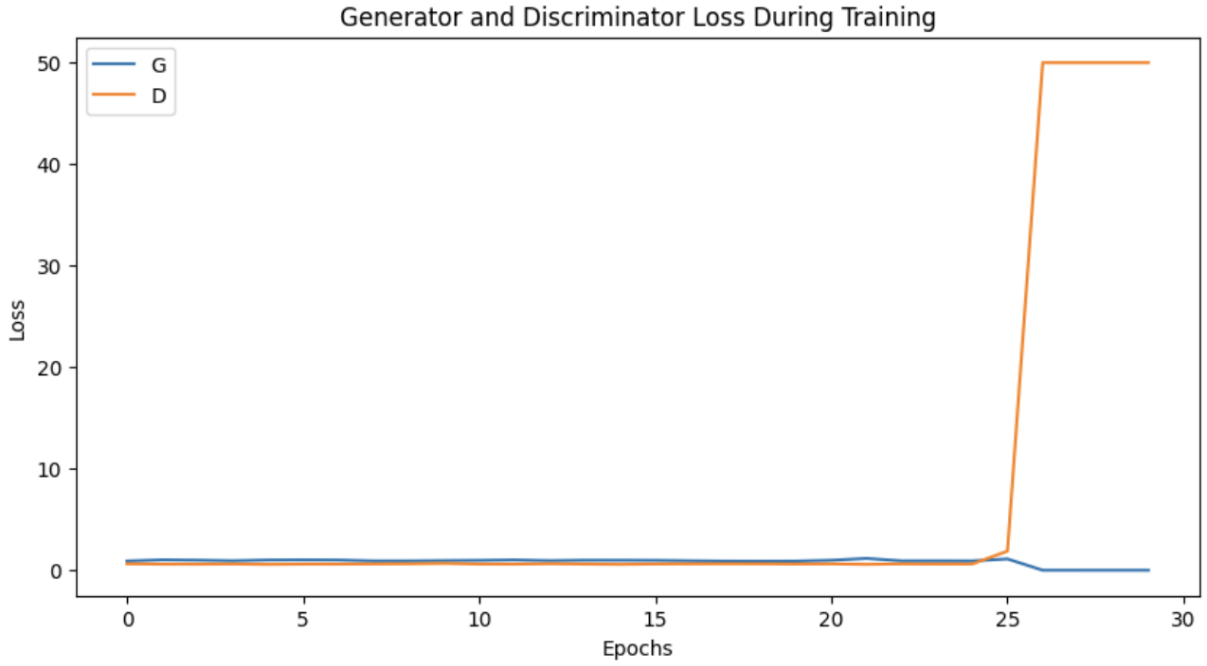


그림 22. 4번 class 결함 32x32 Loss 그래프

각 해상도 별 생성이미지는 다음과 같다. 결함의 특성으로 보이는 포인트를 확인할 수 있다.

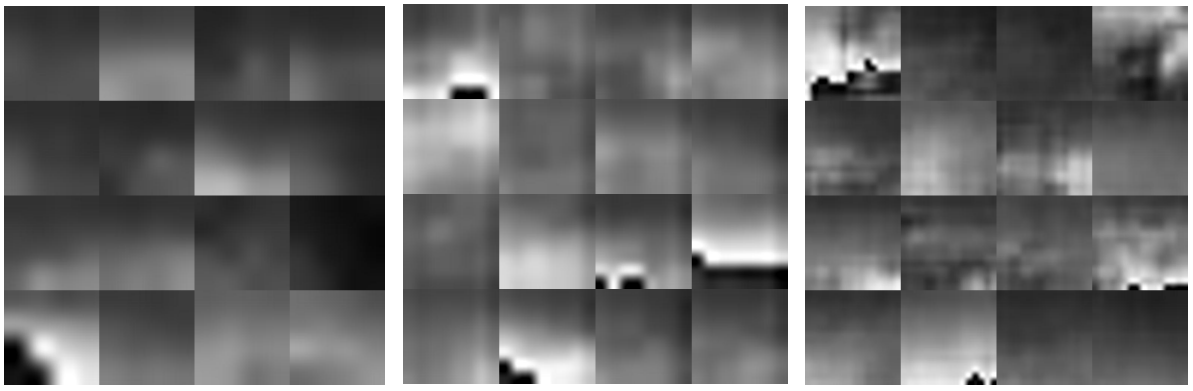


그림 23. 4번 class 결함 fake 이미지

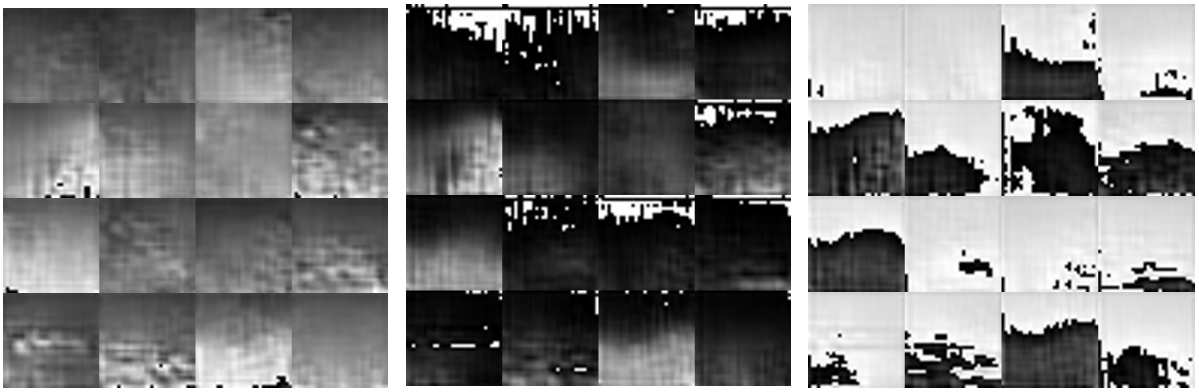


그림 24. 4번 class 결함 fake 이미지 : 32x32

해상도 32x32의 epochs 25를 기점으로 이미지가 바뀌었다. PGGAN은 해상도가 높아지면서 판별자가 제대로 가짜 이미지를 검출하지 못하는 것으로 보인다.

현재 G-loss 값 대비 D-loss값의 급증의 원인을 학습률 값으로 가정하고 학습률 값을 조정하고 있다.

3.5. 분류 모델

3.5.1. CNN

데이터 불균형을 해소하기 위해 결함 label이 1,2,4에 대해서 원래 이미지에 대해 좌우 반전과 상하 반전을 시켜 특정 label에 대해 데이터셋을 3배로 늘렸다. 참고로 회전을 하지 않은 이유는 도메인에 맞는 증강을 하기 위해서였다. 예를 들어 수직 방향으로만 생기는 철강의 결함에 대해 rotation을 하여 사선 방향이나 수평 방향으로 생성된다면 도메인 지식에 어긋난다.

이렇게 증강 시킨 데이터셋으로 학습을 시켰고, validation 데이터의 accuracy가 오를 때 학습 모델을 저장하여 갱신하였다. 처음에는 학습시킬 때는 원본 이미지를 224 by 224로 resize하였다. 하지만 가로 세로 비율이 100:16인 원본 이미지를 정방이미지로 압축을 시키면 defect2와 같은 결함의 경우에는 특히 손실이 많다. 그렇기에 원본 이미지 비율이 유지될 수 있도록 학습을 진행했다.

그림 25는 CNN 모델의 구조이다. Keras를 사용하여 layer를 쌓아 정의하였다. 모델의 입력

이미지는 가로 세로 비율이 유지되도록 (32, 200, 3) 형태로 넣었다. 효과적인 이미지 특징 추출과 분류를 위해 다양한 레이어를 조합하였다.

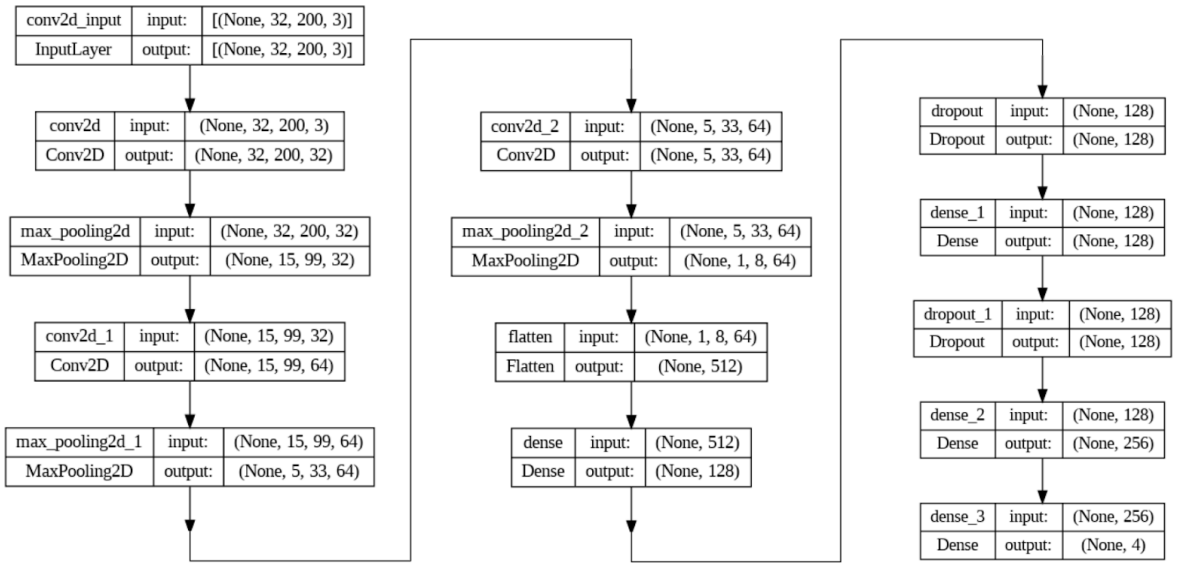
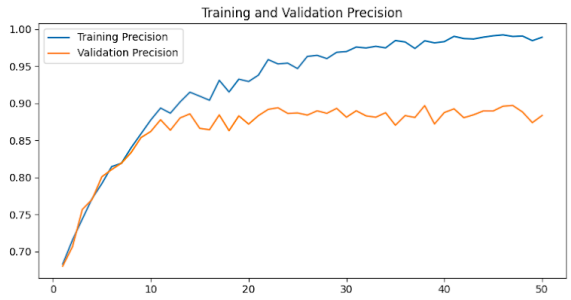


그림 25. 설계한 CNN 모델의 구조

train : validation : test 데이터를 8:1:1로 나누어 50 에포크로 학습을 진행했다. 그림 26의 좌측 그래프 "Training and Validation Loss"를 보면 validation loss가 에포크 12이후 증가하여 더이상의 학습은 진행하지 않았다.

그림 26. Train data와 validation data의 loss(좌), accuracy(우)



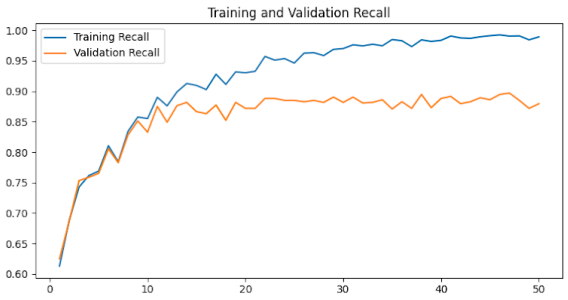
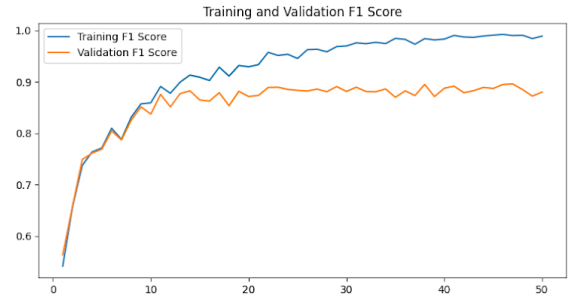


그림 27. Train data와 validation data의 Precision(좌), Recall(우)



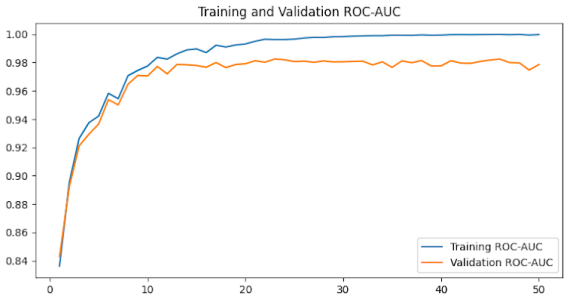


그림 28. Train data와 validation data의 F1 score(좌), ROC-AUC(우)

표 6은 test 데이터에 대해 CNN 모델의 전체 성능을 평가한 지표이다.

accuracy	precision	recall	f1 score	roc_auc
0.888	0.891	0.888	0.888	0.9791

표 6. CNN 모델의 평가 지표 값

결함 별로 성능을 살펴보면 아래와 같다. 그림29는 CNN 모델의 Confusion Matrix를 그려본 결과이다. defect 2에서 분류가 다소 안되는 모습을 보이나, 행렬이 사선 방향으로 색깔이 진하고 값이 큰 부분이 주로 대각성 상에 위치한 것으로 보아 예측이 잘 되었다고 판단이 된다.

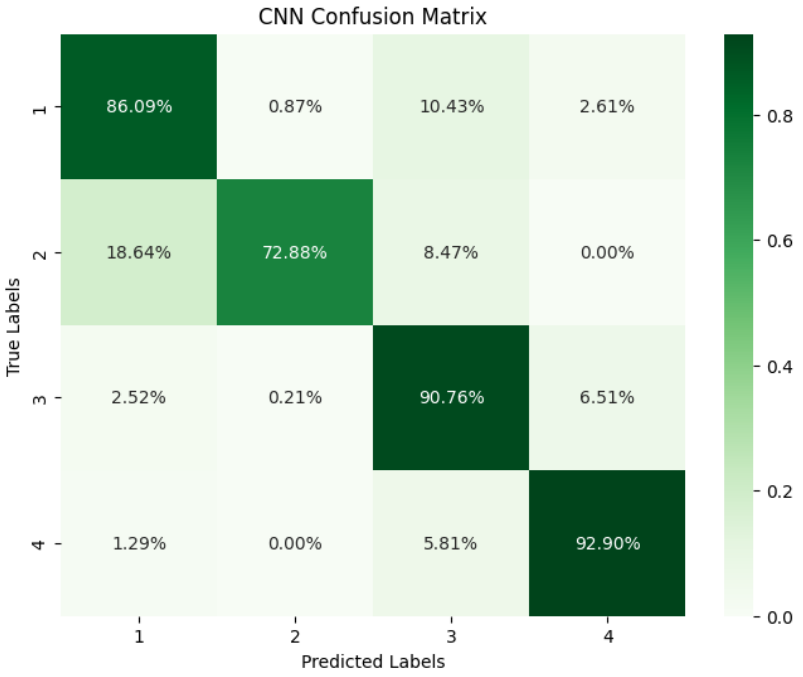


그림 29. CNN 모델의 Confusion Matrix

3.5.2. ResNet

ResNet 아키텍처의 원래 버전인 ResNet-50 등은 주로 ImageNet과 같은 대형 데이터셋에서 사전 훈련되었고, 이 데이터셋은 224x224 크기의 이미지를 사용했다. 따라서 사전 훈련된 ResNet 모델 사용을 위해 철강 이미지의 크기를 정방 사이즈로 맞추고자 했다. 처음에는 기존의 전체 이미지 크기를 정방으로 resize 했으나 성능이 좋지 못하여 커스텀 Steel Dataset 클래스를 별도로 정의하였다. 이미지는 텐서로 변환한 후 정규화하는 방식으로 전처리를 진행했다. 이미지 데이터가 들어왔을 때 해당 이미지의 너비를 6등분 한 후 (224,224)로 각각을 resize하여 image list가 return하도록 구현했다.

ResNet에는 layer의 깊이에 따라 여러 버전으로 나뉜다. 표7은 논문[5]에서 발췌한 것으로 ResNet 모델의 구조를 표현했다.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

표 7. ResNet 버전별 Layer의 깊이

우리는 각 모델별로 성능을 비교하고자한다. 사전 훈련된 torchvision의 ResNet을 사용했고, 철강 데이터로 재훈련을 진행했다. 훈련 방식으로는 6가지의 이미지 부분을 모델에 통과시키고 나온 각각의 출력을 취합하여 평균을 내는 방식을 취했다. 더 상세하게 기술하자면, 6등분된 이미지의 출력을 쌓아서 텐서를 만든 후, 이 텐서를 새로운 차원을 기준으로 평균을 구하여 새로운 텐서를 생성했다. 이로써 전체 손실을 계산하여 역전파가 수행되고 이를 통해 모델의 가중치가 조정되어 학습이 이루어진다.

표8은 ResNet에 재훈련을 진행한 후 train 데이터와 test 데이터를 사용해 모델의 성능을 평가한 지표이다.

	resnet 18		resnet 34		resnet 50		resnet 101		resnet 152	
	train	test	train	test	train	test	train	test	train	test
Accuracy	0.8379	0.8373	0.7868	0.7740	0.8193	0.8061	0.7862	0.7909	0.7736	0.7772
Precision	0.8275	0.8262	0.7821	0.7415	0.7955	0.7906	0.7546	0.7591	0.6934	0.6981
Recall	0.8379	0.8373	0.7868	0.7740	0.8193	0.8061	0.7862	0.7909	0.7736	0.7772
F1 score	0.8194	0.8175	0.7291	0.7121	0.8014	0.7924	0.7462	0.7553	0.6871	0.6940

표 8. ResNet 버전별 test 데이터 평가 지표

ResNet-18은 상대적으로 더 간단한 구조를 가지고 있어서, 현재 사용하는 작은 양의 철강 데이터셋에서 가장 일반화가 잘된 것으로 추정된다. 그리고 resnet 101과 resnet 152의 경우

train 데이터의 성능 평가 지표보다 test 데이터의 성능평가 지표가 더 우수하게 나오는 기이한 현상이 발견된다. 해당 원인은 데이터 불일치 문제로 추정된다. 훈련 데이터와 테스트 데이터가 서로 다른 분포를 가져서 테스트 데이터에 대해 유난히 성능이 더 높게 나온 것이다. 이러한 문제점을 통해 각 클래스의 샘플이 훈련 데이터에서 골고루 나타나도록 샘플링을 조절하였고, 훈련 데이터셋을 더 다양하게 만들기 위한 데이터 증강 기법을 연구한다.

그림 30은 ResNet 버전 중에서 가장 성능이 좋았던 ResNet18의 confusion Matrix이다. defect별 성능을 살펴본다면, defect 2와 defect 4의 경우 성능이 좋지 않음을 판단할 수 있다.

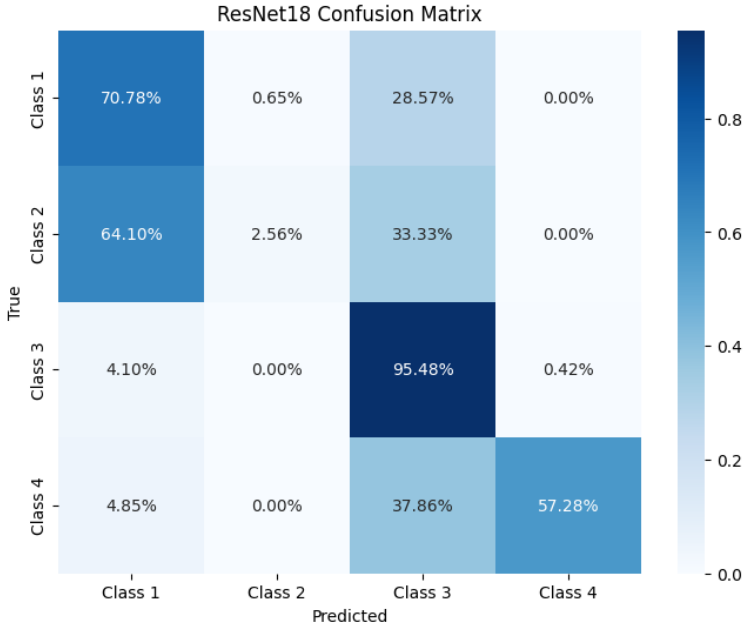


그림 30. ResNet 18의 Confusion Matrix

3.5.3. EfficientNet

사전훈련된 ResNet 모델을 사용할 때 원본 이미지 크기를 유지하여 학습을 시키는데에 어려움을 겪었다. 그리하여 원본 이미지 크기의 비율을 유지할 수 있는 EfficientNet B0와 EfficientNet B3로 학습을 진행했다. TensorFlow의 Keras API에서 제공하는 클래스인 ImageDataGenerator를 사용하여 이미지 반전 증강을 했고, 이미지 크기는 너비 625 픽셀, 높이 100 픽셀로 줄여 입력으로 넣었다. 처음에는 multi label이 포함된 데이터셋으로

학습을 시켰고, 성능을 더 높이기 위해 multi label을 제외한 데이터셋 또한 학습을 시켜서 비교해보았다.

그림 31은 EfficientNet B0의 전체 모델 구조를 그린 것이다. 크게 Stem, Blocks, Head, Pooling & FC 부분으로 나눌 수 있다. Stem은 모델의 시작 부분으로 입력 이미지에 대한 전처리가 수행되고, 초기 특징이 추출된다. Blocks에서는 특징을 계층적으로 추출하며, 네트워크의 깊이에 따라 다양한 수의 블록을 사용한다. EfficientNet B0는 7개의 반복 블록을, EfficientNet B3는 10개의 반복 블록을 가진다. Head에서는 입력 이미지에서 추출한 특징을 더욱 강화하고, Swish 활성화 함수를 통해 비선형성을 부여하여 네트워크의 표현력을 향상시킨다. Adaptive Average Pooling에서는 입력된 특징 맵의 크기를 동적으로 조절하여 고정된 크기의 출력을 생성하고, Global Average Pooling은 특징 맵의 각 채널별로 평균 값을 계산한다. 평균 값을 계산한 결과로 얻은 특징 벡터는 Fully Connected Layer에 입력으로 사용되어 최종적인 출력을 얻을 수 있게 된다.

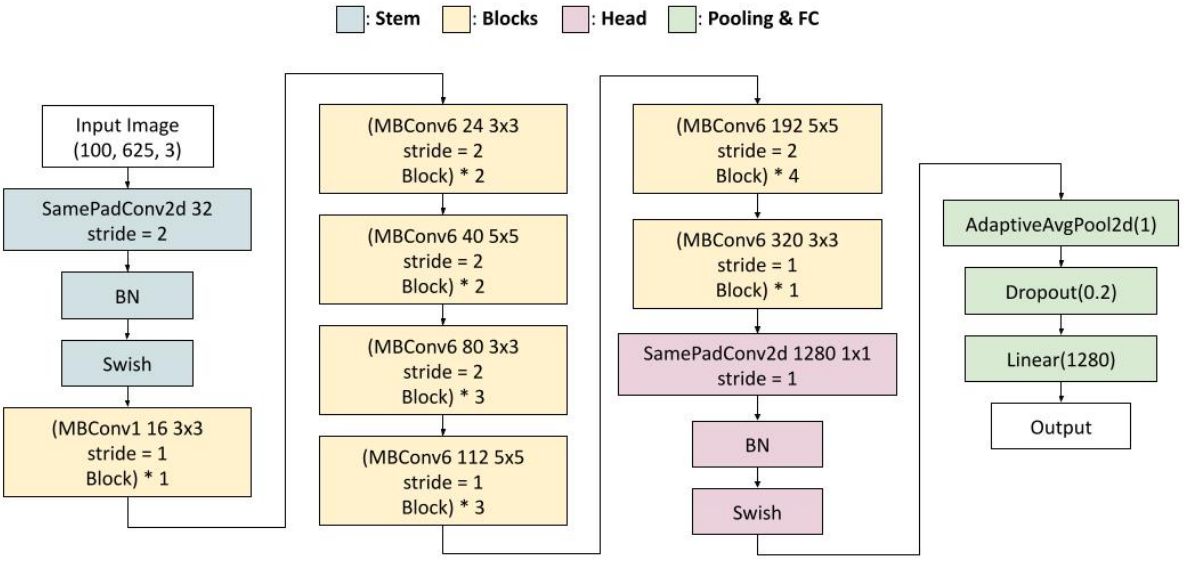


그림 31. EfficientNet B0의 전체 모델 구조

그림 32는 multi-label을 제외한 데이터셋에 대해 EfficientNet B3로 학습을 시킨 결과이다. training 데이터의 평가 지표는 안정적으로 나오나, validation 데이터는 다소 흔들리는 것처럼 보인다. 이는 모델이 새로운 데이터에 대해 아직 학습 과정에서 일부 미세 조정을 하고 있는 것으로 판단된다.

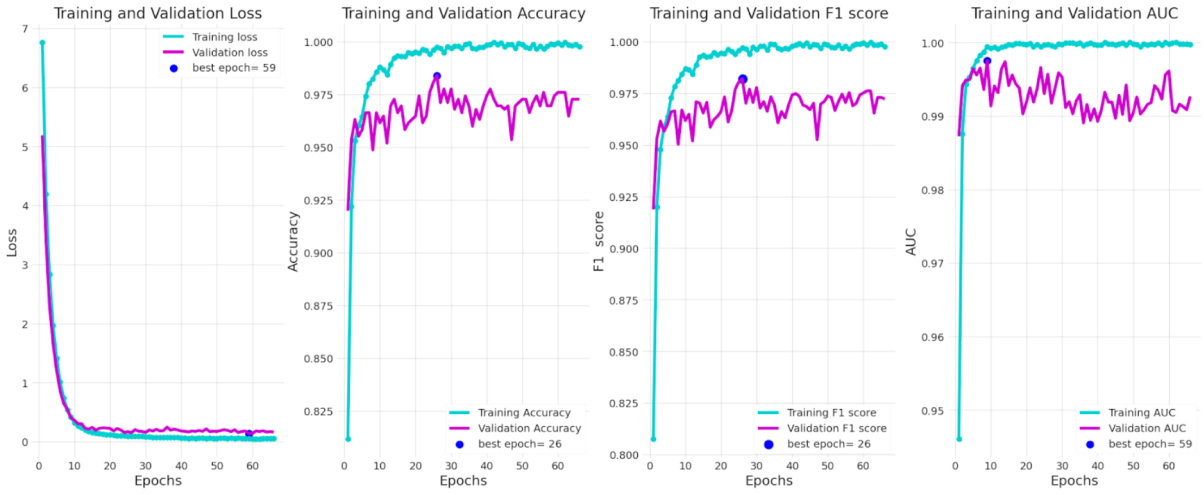


그림 32. EfficientNet B3의 학습 결과 (multi-label 제외)

표 9는 실제 test 데이터셋에서의 성능을 기록한 표이다. multi label을 제외한 데이터셋으로 학습을 시켰을 때 분류를 더 잘하며, EfficientNet B0보다는 B3에서 성능이 더 좋게 나온다. 이는 B3기 B0보다 더 깊고 넓으며 더 높은 해상도의 입력 이미지를 처리할 수 있기 때문으로 판단된다. 그렇기에 B3가 더 복잡한 패턴을 학습할 수 있었던 것이다.

model	multi label	accuracy	f1 score	roc_auc
Efficient Net B0	O	0.8561	0.8547	0.9566
	X	0.9647	0.9655	0.9947
Efficient Net B3	O	0.8674	0.8669	0.9639
	X	0.9744	0.9735	0.9904

표 9. 모델별 multi-label 유무에 따른 평가 지표 값

그림 33은 EfficientNet 버전 중에서 가장 성능이 좋았던 Efficient Net B3의 confusion Matrix이다. defect별 성능을 살펴본다면, defect2에 대해서는 모델이 잘 학습하지 못함을 확인할 수 있다.

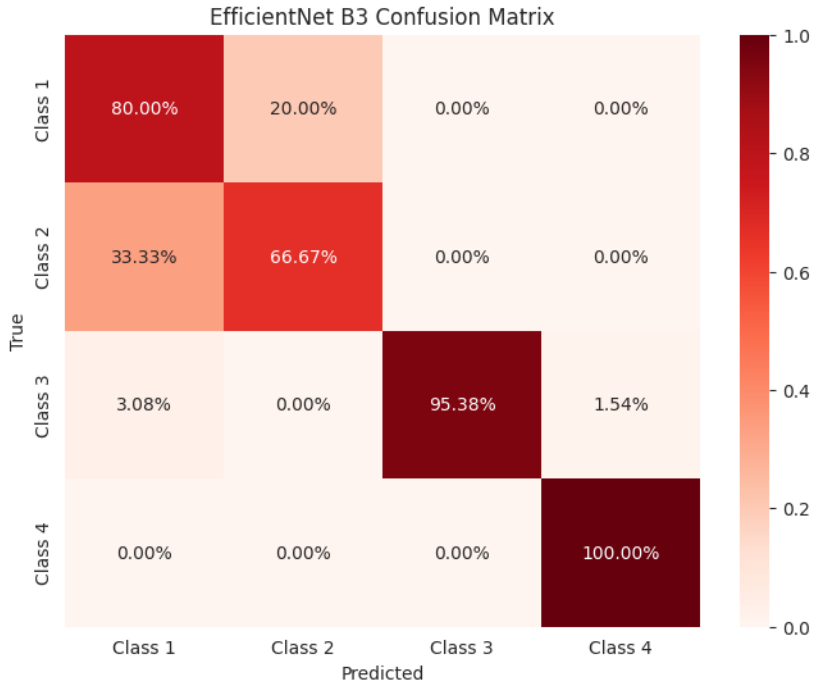


그림 33. EfficientNet B3의 Confusion Matrix

3.6. 세그멘테이션 모델

검출 모델에 들어가기 앞서, 한 가지 짚고 넘어갈 것이 있다. 현재 철강 데이터셋에는 이미지 내에 결함이 있는 부분이 인코딩된 픽셀로 존재한다. 그러나 실제 산업 환경에서는 결함이 픽셀 값으로 명시적으로 나타나는 경우가 드물다. 그렇기에 우리는 인코딩된 픽셀을 사용하지 않고 결함을 분류할 수 있는 좋은 성능의 모델 연구를 해왔다. 다만, 우리의 최종 목표인 서비스 배포에서 단순 분류 모델만을 보이기에는 UX(User Experience) 측면에서 사용성과 유용성이 떨어진다고 판단했다. 구체적인 결함 위치를 보여준다면, 사용자는 어떤 부분이 문제인지 더 쉽게 이해할 수 있을 것이고, 사용자에게 더 많은 정보를 제공하게 되어 유용성이 향상될 것이다. 그리하여, 추가적으로 Semantic Segmentation 모델에 대한 연구를 하였다.

3.6.1. U-Net

데이터 구성 방식은 모델이 multi-label과 마스크 처리에 유리할 수 있도록 변경하였다. 현재 철강 데이터셋 들어있는 이미지 ID, 결함 유형, 그리고 결함 위치를 나타내는 RLE(run-length encoding) 문자열(EncodedPixels) 정보를 활용하여 새롭게 데이터셋을 재구성했다. 각

이미지 ID에 대해 4개의 결함 유형별 엔트리를 생성하여 처리 과정에서 각 이미지와 결함 유형 조합에 따른 마스크 정보를 쉽게 조회할 수 있도록 설계했다.

그림 34는 설계한 모델의 구조이다. input size는 원본 이미지의 크기로 설정했고, 학습시 원본 이미지 크기의 비율이 유지된다. 모델은 5단계의 인코더와 디코더로 구성했으며 활성화함수로 ELU를 사용했다.

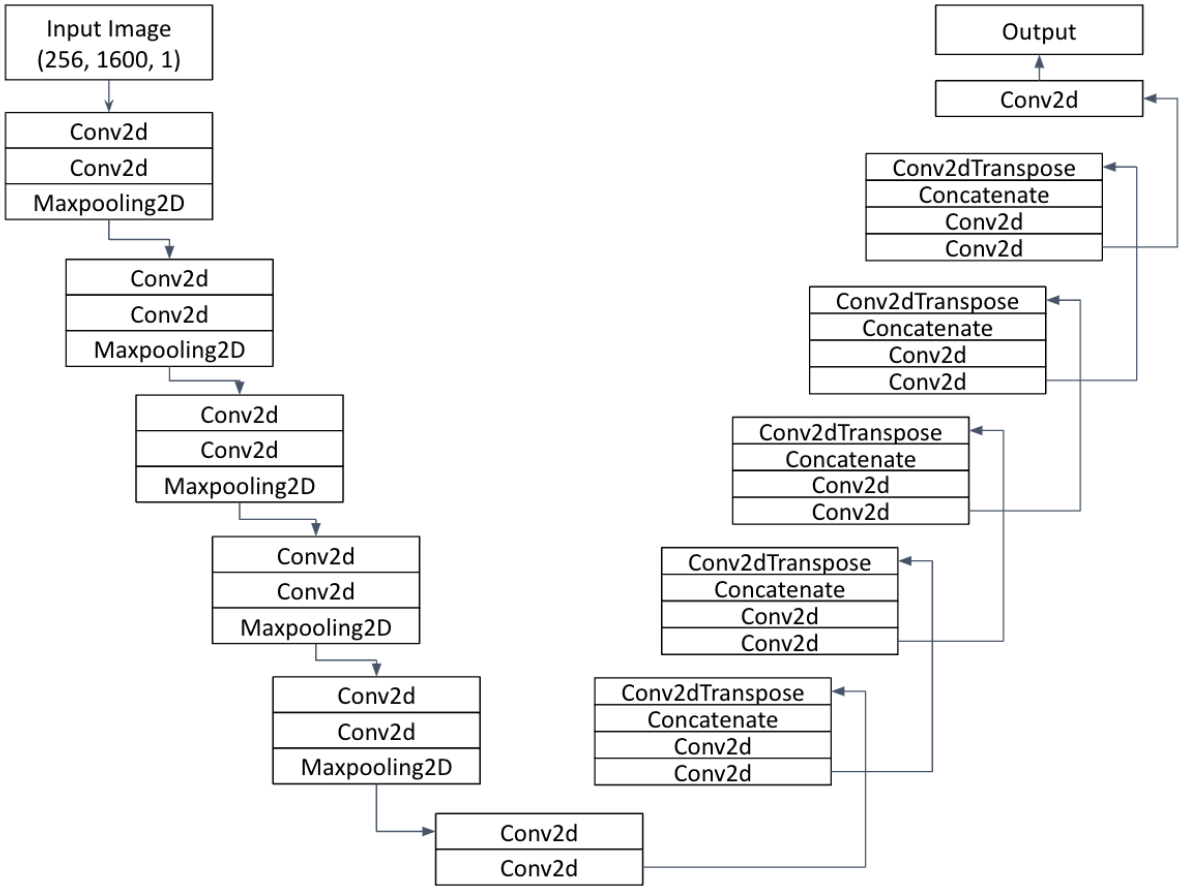


그림 34. 설계한 U-Net의 구조

처음에는 4단계의 MaxPool2d를 사용한 다운 샘플링과 3단계의 Umsample 업샘플링으로 모델을 구성했으며 각 단계에서는 Conv2D와 ReLU 활성화 함수를 사용하는 2개의 연속적인 합성곱 계층을 사용했다. 하지만 이 모델은 dice coefficient 값이 0.01로 나오며 마스크 값이 제대로 도출되지 않는다는 문제점이 있었다. 그리하여 데이터 구성 방식, 모델의 구조와 사용되는 파라미터 값을 변경함으로써 개선시킬 수 있었다.

표 10은 모델에서 사용된 설정 값들을 나타낸 것이다. 업샘플링을 할 때 Upsampling으로

단순히 이미지 크기를 증가시키는 것보다 Transposed convolution을 통해 역전파 과정 중에서도 그래디언트 정보를 보존하는 것이 성능 개선에 도움이 될 것이라 판단했다. 그리고 손실 함수로 단순히 binary cross entropy만 사용한다면, 모델은 각각의 픽셀들이 어느 클래스에 속해야 하는 지만 학습하게 되기에 dice loss와 결합하여 사용했다. dice coefficient 메트릭을 함께 사용하여 전체 이미지 수준에서의 성능 개선에 중점을 뒀다. 활성화 함수로는 Relu의 그래디언트 소실 문제를 보완해주는 Elu로 변경했다. 그리고 계층 수를 늘려서 보다 복잡한 패턴을 학습하는 데에 효과적으로 설정했으며, 출력 채널을 1개에서 4개로 늘림으로써 multi-label 데이터셋에 적합하도록 변경했다.

	설정 값
CNN 모델	U-Net
옵티마이저	Adam
배치 사이즈	16
다운샘플링 연산자	MaxPooling2D
업샘플링 연산자	Upsample → Conv2DTranspose
손실 함수	BCE(Binary Cross Entropy) → BCE + Dice Coefficient
활성화 함수	ReLu → ELU
계층 수	14개 → 38개
출력 채널 수	1개 → 4개

표 10. 사용된 hyper-parameters

그림35는 100 epoch 만큼 학습을 시켰을 때 나온 loss 그래프와 dice coefficient 그래프이다. train loss가 계속 줄어들고 있는 것으로 보아 모델이 훈련 데이터를 잘 학습하고 있다는 것을 확인할 수 있었다. 다만, validation의 loss와 dice coefficient가 어느 순간부터 일정하게 유지가 되고 있어 모델의 일반화 성능이 한계에 도달한 것으로 판단되었다.

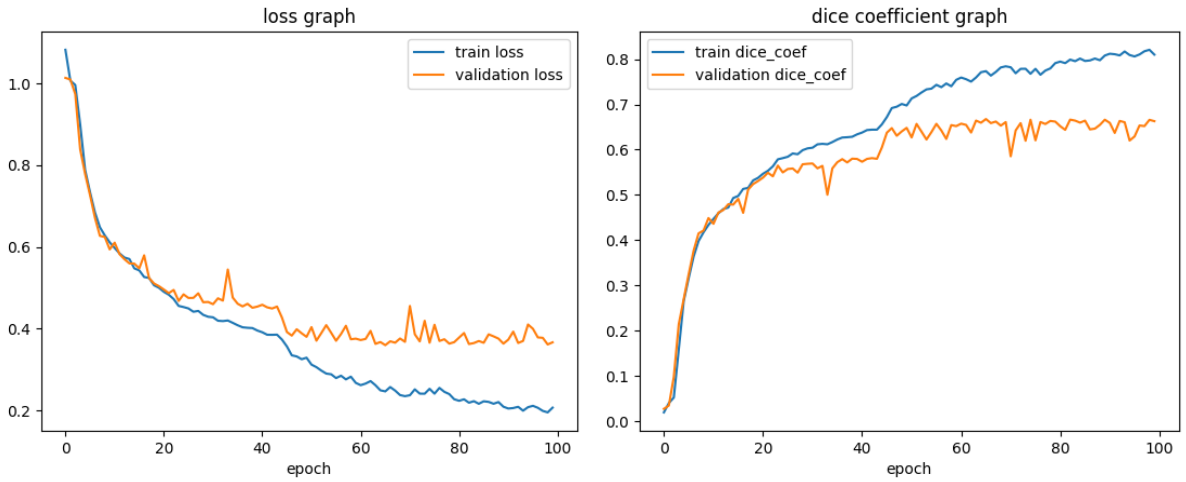


그림 35. loss graph(좌), dice coefficient graph(우)

학습된 모델에 test 데이터를 넣고 평가해보았고, dice coefficient값이 0.6749로 나왔다. 이미지 위에 mask를 표시하는 함수를 사용하여 원본 마스크와 예측된 마스크를 비교해보았다.

표 11은 defect 3(보라색), defect 4(빨간색)를 가진 철강 이미지에 대한 예측 결과이다. 모델이 다중 결함을 잘 처리하고 있음을 파악할 수 있었다.

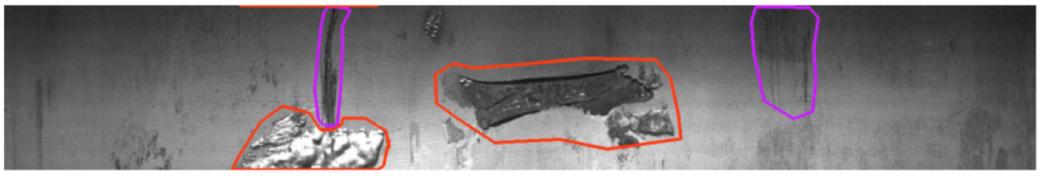
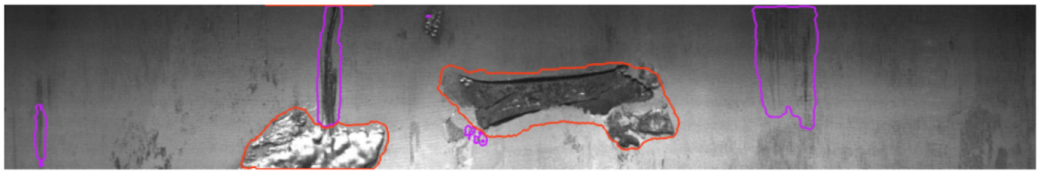
original mask	
predicted mask	

표 11. defect 3(보라색), defect 4(빨간색)를 가진 철강 이미지와 예측된 마스크

표 12는 defect 4(빨간색)를 가진 철강 이미지에 대한 예측 결과이다. 결함이 아닌 부분에 대해서 defect 3 으로 인식되는 경향이 있다. 해당 이유는 두가지로 추정되는데, 첫번째로는 전체 데이터셋에서 defect 3이 차지하는 비율이 크기 때문이다. 두번째로는 defect 3 생김새 중에서 검은색 세로선처럼 생긴 결함이 많이 학습이 되었고, 그것과 비슷하게 생긴 표면을 defect 3이라 인식하기 때문으로 추정된다. 하지만 육안으로 보았을 때 충분히 결함으로

판단할 정도로 강판 표면에 결함이 있어 보이고, 제공된 데이터에서 검출하지 못한 불량을 모델이 검출해낸 것일 수도 있다. 그렇기에 segmentation이 오히려 잘 됐다고 평가할 수 있다.

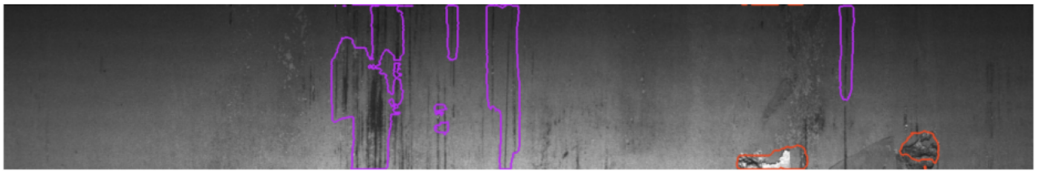
original mask	
predicted mask	

표 12. defect 4(빨간색)를 가진 철강 이미지와 예측된 마스크

표 13은 defect 1(노란색), defect 2(하늘색)를 가진 철강 이미지에 대한 예측 결과이다. defect 1과 defect 2가 들어있는 여러장의 이미지에 대한 예측된 마스크 결과값들을 보았을 때, 예측이 잘 되지 않는 것을 파악했다. 학습된 데이터셋들을 살펴보았을 때 defect 2의 경우, 이미지 크기에 비해 마스크된 부분의 크기가 작다는 문제가 있다. defect 1의 경우 데이터셋 개수가 압도적으로 적다는 점과 데이터 품질이 좋지 않다는 점이 원인으로 파악된다. defect 1 데이터셋들을 검토해보면, 결함이 매우 작고 얕아서 모델이 패턴을 학습하는 데에 어렵다고 판단된다.

original mask	
predicted mask	

표 13. defect 1(노란색), defect 2(하늘색)를 가진 철강 이미지와 예측된 마스크

3.7. 서비스 구조 설계

3.7.1. 서비스 구조도

그림 36은 이 연구에서 개발한 철강 불량 검출 서비스 앱의 구조도이다.

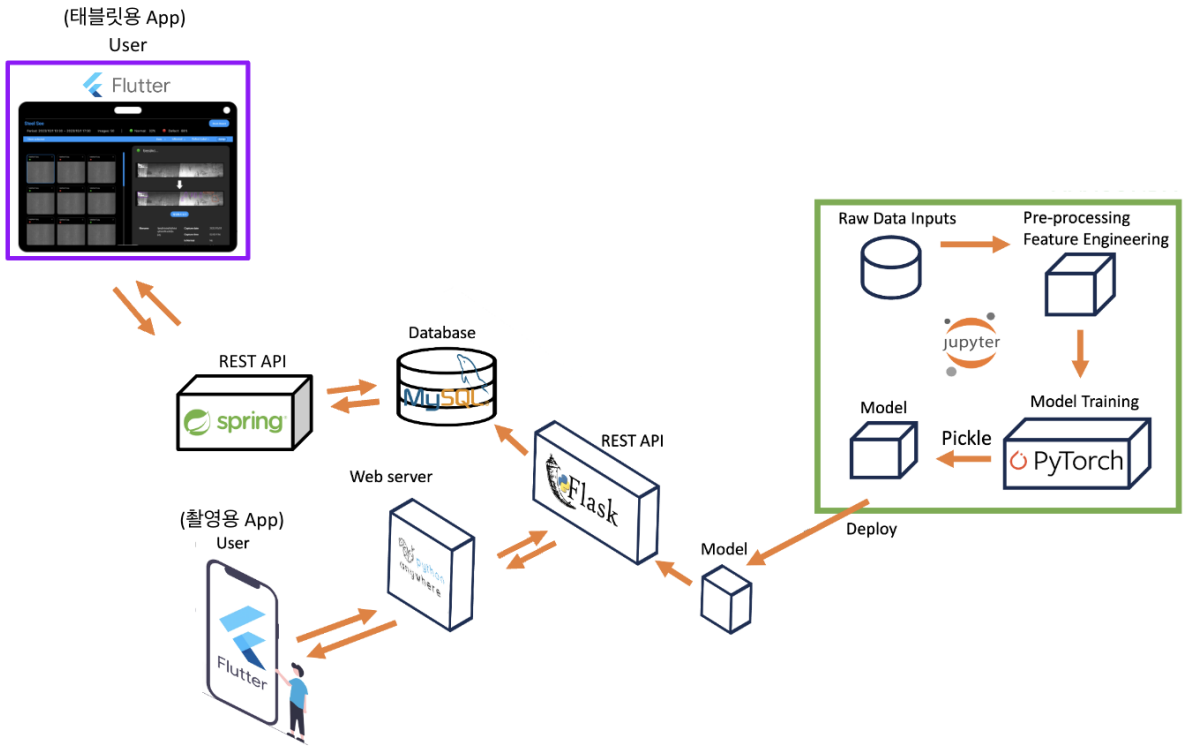


그림 36. 서비스 앱 구조도

구조를 간단히 설명하자면, 먼저 사용자는 촬영용 어플리케이션을 이용하여 철강 사진을 촬영하여 업로드한다. 해당 사진은 웹 서버에 올라간 Flask 기반 REST API로 전달된다. API는 서버에 저장된 학습 모델에 사용자의 사진을 input으로 넣어 예측을 한 후, 결함이 검출 및 분류 된 데이터들을 데이터베이스에 저장한다.

이후, 사용자는 태블릿용 어플리케이션에서 자신이 업로드했던 사진을 조회한다. 이때 Spring 기반 REST API가 데이터베이스에 저장된 사용자가 업로드한 사진과 결함이 검출된 사진, 그리고 결함 class 번호 등의 데이터들을 가져오게 된다. 해당 데이터들은 태블릿용 어플리케이션 화면상에서 보여지게 되어, 사용자는 자신이 올린 철강 사진에 대한 결함 정보를 얻을 수 있게 된다.

3.7.2. DBMS 구축

데이터베이스는 관계형데이터베이스(RDB)기반 관리 시스템인 MySQL을 채택하였다. 도커 컨테이너에 mysql 서버를 띄워서 독립된 환경에서 배포할 수 있도록 구현하였다. DB구성은 Spring Data JPA로 하였고, API 호출로 데이터를 사용한다.

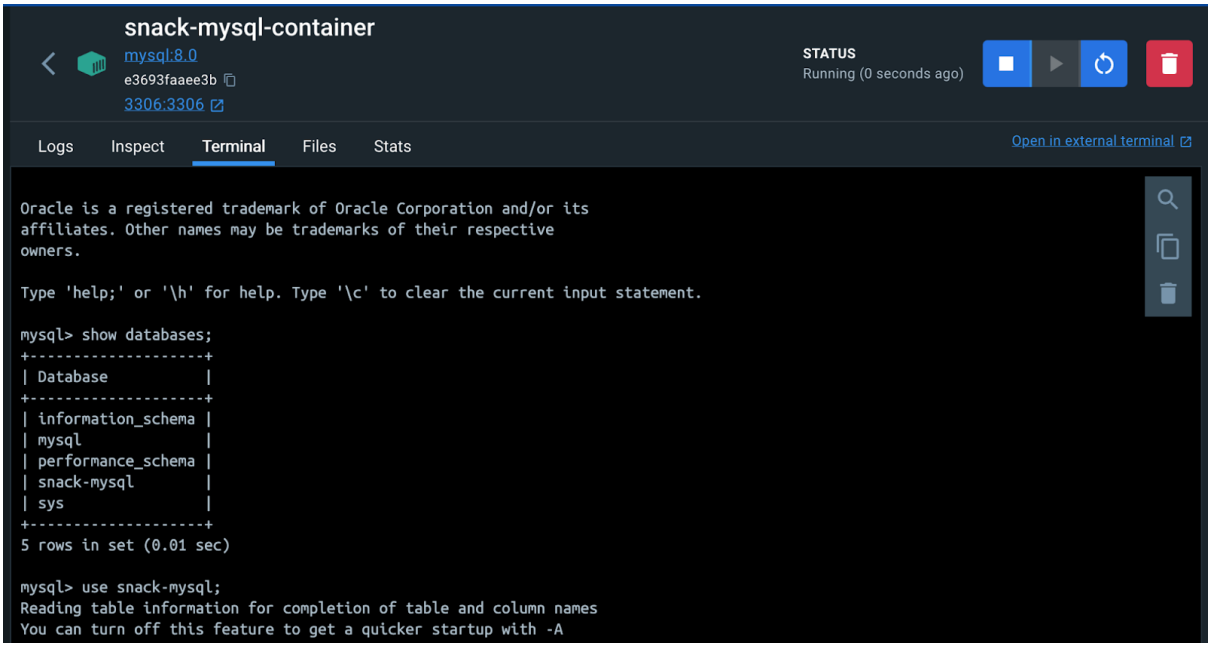


그림 37. Docker로 동작중인 mysql 서버

3.7.3. Rest API 설계

Rest API는 두가지로 나누어 제작했다.

먼저, 인공지능 모델을 호출하여 예측을 수행한 후 데이터베이스에 검출된 이미지와 불량 label을 저장하는 Rest API를 제작했다. 우리가 개발한 분류 모델과 세그멘테이션 모델 두가지를 pickle하여 서버에 업로드하였고, 이미지 파일이 들어오면 예측 모델에 넣고 결과값을 데이터베이스에 저장될 수 있도록 하였다. 인공지능 모델을 모두 파이썬 언어로 작성했기에 파이썬으로 작성된 웹 프레임워크인 Flask로 개발을 진행했다. 완성된 API는 웹 호스팅을 가능하게 해주는 서비스인 pythonanywhere에 업로드하여, 앱에서 접근 가능한 API로 만들어냈다.

다음은 mysql 데이터베이스에 접근을 통해 이미지 조회 등의 기능을 하는 Rest API를 제작했다. mysql은 관계형 데이터베이스 관리시스템(DBMS)으로 데이터 명세를 기준으로

테이블간의 관계를 표현한다. 데이터 저장방식은 mysql로 했지만 이에 접근하고 관리하기 위해 Spring Data JPA를 사용했는데, 테이블을 자바 객체로 mapping하여 관리할 수 있다. 객체로 관리하기 때문에 앱에서 필요한 정보를 요청할 때, 매번 쿼리문을 작성하지 않고 함수형 명령어로 입출력, 삭제를 자동화할 수 있는 장점이 있다.

3.7.4. Application 개발

Application을 개발하기 전, UI/UX 디자인 프로그램인 figma를 활용하여 어플의 UI를 구상했다. 그림 38은 처음 구성했던 모바일용 어플의 figma이다.

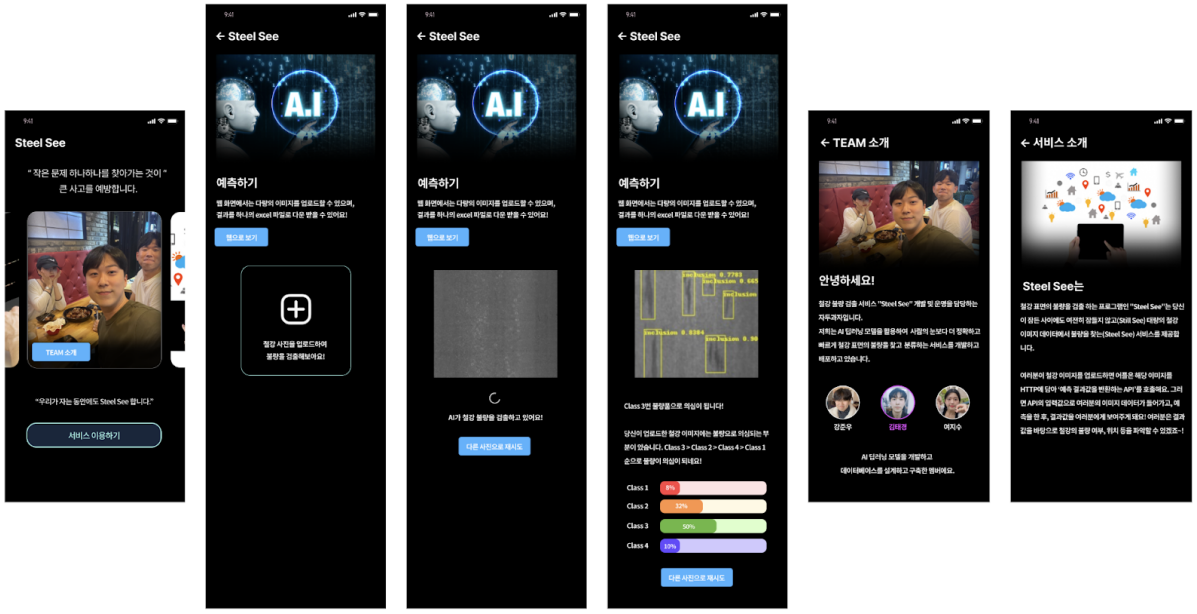


그림 38. 처음 구상한 UI

중간 발표 때 테블릿 버전 어플에 대한 필요성과 이미지 확대 기능 및 결함 부분에 대한 마스킹 표시 기능에 대한 피드백 등을 바탕으로 그림39과 같이 UI를 새롭게 재구성하였다.

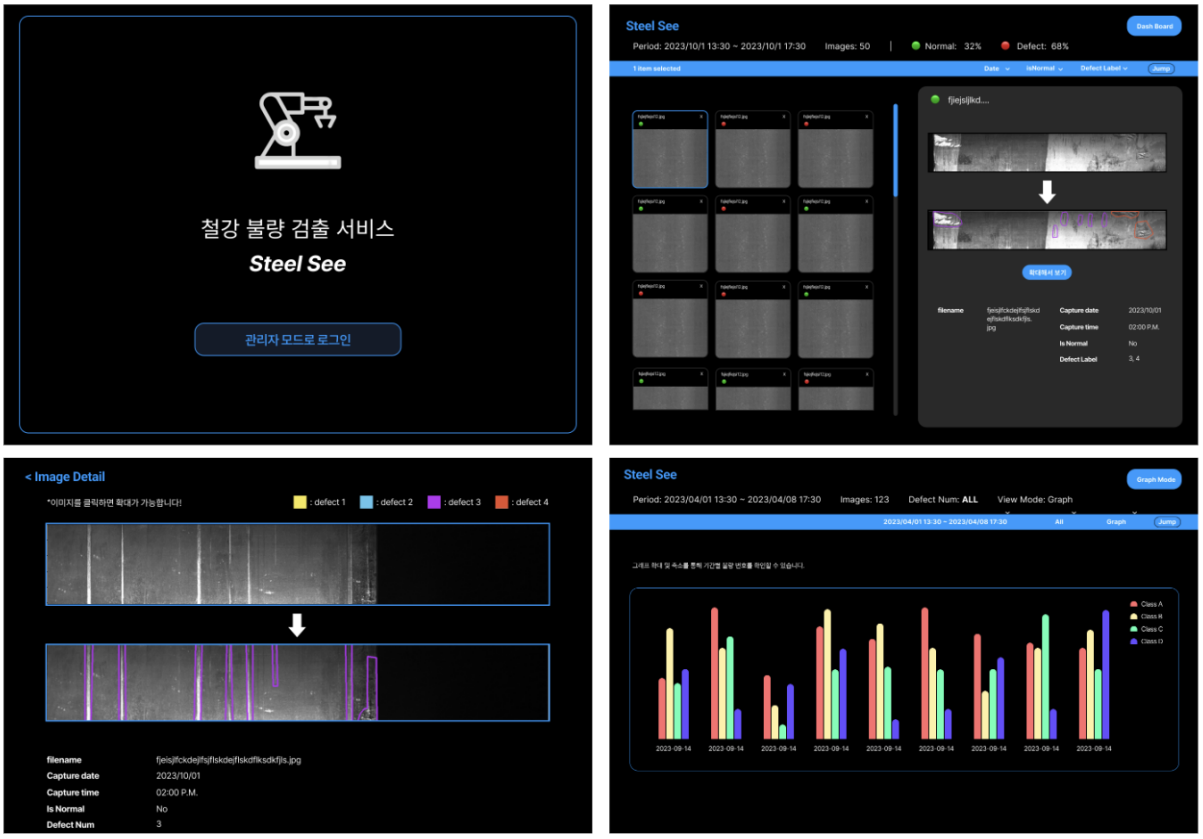


그림 39. 현재의 UI

리뉴얼된 UI를 바탕으로 Flutter 프레임워크를 활용해 새롭게 어플리케이션을 제작하였다. 어플리케이션 동작 과정에 관해서는 4장 연구 결과 분석 및 평가의 서비스 부분에서 설명하겠다.

4. 연구 결과 분석 및 평가

4.1. 생성 모델

4.1.1. DCGAN

표 14는 800, 800, 300 epochs만큼 학습했다. Defect4의 경우 300이 넘어가면서 Loss그래프가 불안정해지면서 생성 이미지의 반복된 패턴이 발생하는 Mode Collapse가 발생했다. 이에 300 epochs의 이미지를 첨부했다. 결함별로 분석해보면 상대적으로 특징이 도드라지는 “patches” 결함(defect 4)이 보다 잘 생성됨을 확인할 수 있다.

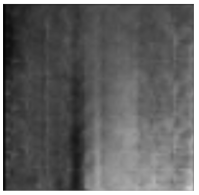
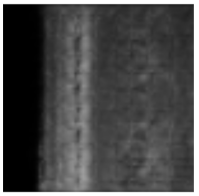
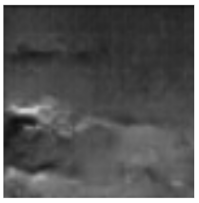
Defect 1	Defect 2	Defect 4
		
		

표 14. DCGAN으로 생성된 Label별 가짜 이미지들 (epoch: 800, 800, 300)

표 15는 각각 300, 1000 epoch만큼 학습했다. 128x128로 해상도를 올렸을때 보다 표면 결함이 세부적으로 표현됨을 확인할 수 있다. 아쉬운점은 모든 DCGAN 모델 학습을 결함별로 epochs 1000이상 학습하여 비교하고자 했으나, 구글 코랩 런타임 환경의 제약으로 인해 epochs 횟수를 통제하지 못했다.

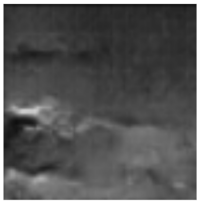
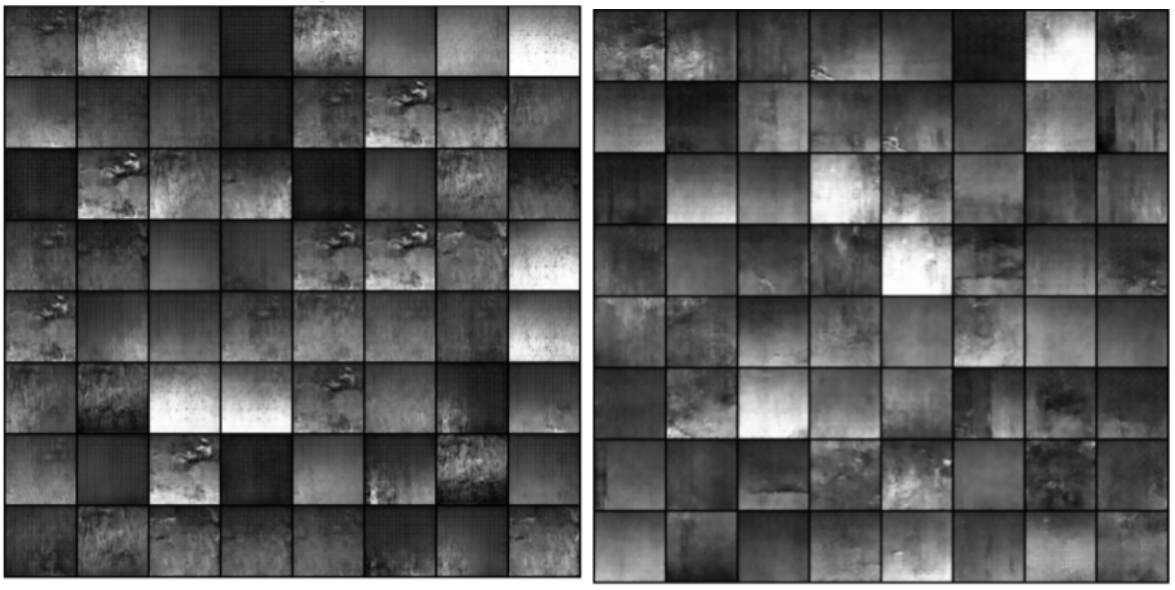
Resolution	Defect 4
64 x 64	 
128 x 128	 

표 15. DCGAN으로 생성된 해상도별 가짜 이미지들 (epoch: 300, 1000)

생성자와 판별자의 convolution 하이퍼 파라미터로 사용되는 ngf(생성자 채널 크기)와 ndf(판별자 채널 크기)의 상대적 비율에 따라 학습된 이미지에서 특이사항을 발견할 수 있었다.

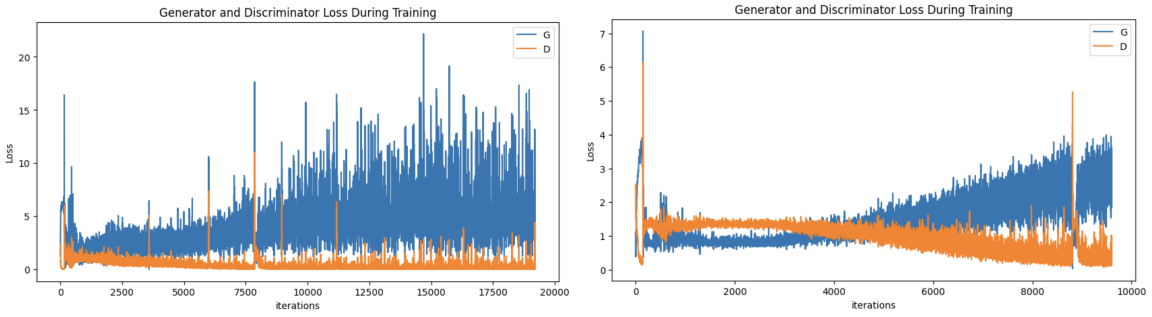
ngf/2=ndf에서 epochs가 일정 수준이상 높아질 수록 G-Loss값이 상승하는 구간이 있는데,

이 구간에서는 이미지 생성에 패턴이 생겨 비슷한 패턴의 결함 이미지가 생성되는 Mode Collapse 현상을 발견하였다. ngf/4=ndf로 설정하자, G-Loss 상승폭이 줄어들면서 Mode Collapse 현상이 다소 해소되었다.



ngf/2 = ndf(좌)　　ngf/4 = ndf(우)

그림 40. 4번 결함 생성 이미지 - 좌측 이미지에 비슷한 패턴의 결함이 보인다.



ngf/4 = ndf(좌)　　ngf/2 = ndf(우)

그림 41. G/D Loss 그래프

ngf/2=ndf(좌)의 G-Loss값이 불안정하게 15까지 올라간다. ngf/2=ndf(우)는 G-Loss값이 안정적인 증가 추이를 보인다.

이는 아래 목적함수 식을 대입하여 해석하면 생성자는 V(D,G)를 최소화 하기 위해 log(D(G(z))를 최대화 하기위해 ndf대비 ngf값이 클수록 복잡한 이미지를 생성하여

판별자가 분류하기 위한 계산이 많아져 D-Loss이 높아지는 경향을 보이는 것으로 추측된다.

$$\min_G \max_D V(D,G) = E_{x\sim p_{data}(x)}[logD(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

그림 42. GAN Loss function: minimax function 수식

실제로 128x128크기의 경우 ngf=ndf로 학습시, D-Loss값이 0이되어 판별자가 너무 쉽게 가짜이미지를 분별하여 학습이 이루어 지지않았다.

다만, 이러한 ngf와 ndf 비율조정이 Mode Collapse에 핵심적인 영향을 주는지에 대해서는 추가적인 연구가 필요해 보인다.

- 정리

DCGAN은 CNN신경망을 활용하여 안정적인 결과값을 내면서 모든 GAN모델의 기본이 되고 있다. 또한 이후 모든 GAN모델은 DCGAN을 기본으로 쌓아올렸다고 해도 과언이 아니다.

> *"Most GANs today are at least loosely based on the DCGAN architecture."*
>
> ***- NIPS 2016 Tutorial by Ian Goodfellow***

그림 43. Ian Goodfellow 인용문

기존 계획은 DCGAN을 시작으로 최신 GAN모델을 분석하여 생성하는 것이었지만, 생각보다 DCGAN의 내용이 깊었고, DCGAN을 공부하면 할수록 파고 들어가야할 내용과 실험을 통해 검증해야할 내용이 많았다. 그래서 기존 계획을 수정하여 얕게 분석하기보다 하나의 기본이 되는 모델을 깊게 공부하면서 성능을 개선하고자 했다.

결과적으로 철강 데이터의 특성에 맞는 전처리 방식과 하이퍼 파라미터 값 등을 찾을 수 있었다.

추후 채널 크기(ngf, ndf)와 해상도 up-scaling(128x128, 256x256)에 관한 상관관계를 연구하여 이미지 생성에 주는 영향을 분석하고자 한다. 그리고 이번 연구에서 얻은 지식을 StyleGAN 등 최근 각광받는 GAN 모델에 접목하여 이미지 생성 모델에 대한 분석과 성능개선을 시도할 계획이다.

4.1.2. PGGAN

PGGAN으로 16x16까지 이미지를 생성하는데 성공하였지만 32x32 해상도를 출력하던 중 G-Loss가 0, D-Loss가 50이 되며 생성에 실패했다. 4x4, 8x8, 16x16은 모두 낮은 Loss값을 생성했고, 모두 G-Loss와 D-Loss 값의 차이가 1미만 이었다.

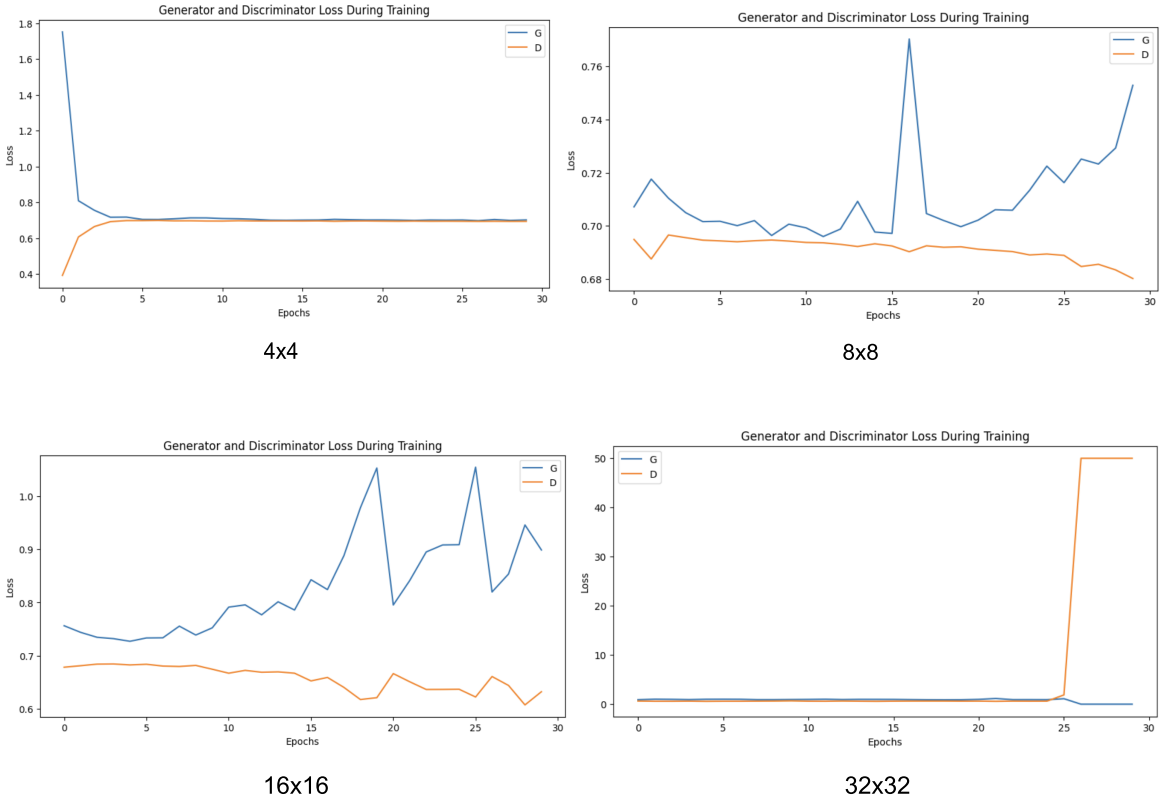


그림 44. 4번 결함 Loss 그래프

PGGAN의 경우 DCGAN에 비해 연구기간이 짧아 많은 결과를 만들어 내지는 못했다. 현재 생성자와 판별자의 Loss값 경향을 통해, 해당 값에 영향을 줄 것으로 예상되는 학습률 값을 조정하고 있고, 추후 채널 값 등을 조정하여 실험할 계획이다.

4.2. 분류 모델

4.2.1. 세가지 모델 비교

표 16은 세가지 이미지 분류 모델의 결과를 비교한 지표이다. 전체적인 성능만 살펴보았을 때는 EfficientNetB3의 성능이 가장 좋게 나온 것을 확인할 수 있다.

Model	CNN	ResNet18	EfficientNetB3
accuracy	0.8934	0.8373	0.9744
f1 score	0.8925	0.8175	0.9735
precision	0.8932	0.8262	0.9739
recall	0.8934	0.8373	0.9744

표 16. 분류 모델 평가 지표

그림 45는 세가지 이미지 분류 모델의 Confusion Matrix를 비교한 지표이다. 결함 별로 살펴보았을 때 defect 1과 defect2는 CNN이, defect3과 defect4에 대해서는 EfficientNetB3가 더 잘 분류했음을 파악할 수 있다. ResNet은 성능이 좋지 않았는데 그 이유를 분석하자면, 우선 CNN과 EfficientNet과 달리 ResNet에서는 데이터 전처리를 할 때 원본 이미지 크기의 비율을 유지시키기 어려웠다. 6등분된 이미지의 텐서를 평균낸 것이 문제였던 것으로 판단되고, ResNet 모델에서는 결함 부분만 정방 이미지로 잘라내어 학습을 시켰더라면 더 좋은 성능을 가져왔을 것이다.

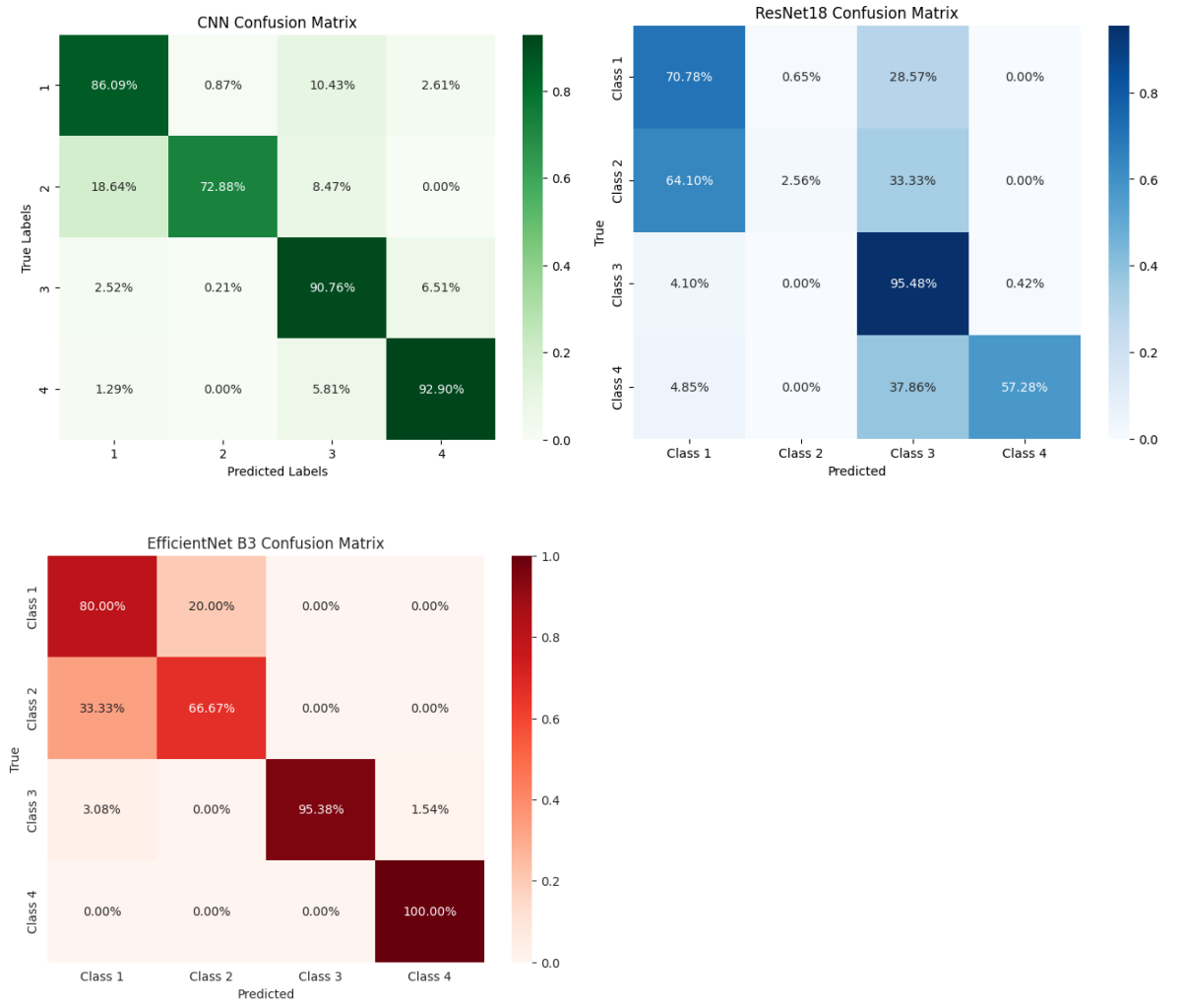


그림 45. 세가지 이미지 분류 모델의 Confusion Matrix

4.2.2. CNN과 EfficientNetB3 성능 평가

CNN과 EfficinetNetB3의 성능을 조금 더 자세하게 비교하고자, 각 결함 유형별로 성능 평가를 다시 진행해보았다.

앞서 그림 45에서 살펴본 CNN의 전체 혼동 행렬에서는 defect 1,2,4에 대해 모델이 잘 분류한 것처럼 보이나, 그림 46과 같이 해당 결함을 유형별로 혼동 행렬을 살펴보면 True Positive값은 크지만 True Negative 값은 굉장히 작다는 것을 확인할 수 있었다. 이는 불균형한 성능을 시사하기 때문에 CNN 모델의 성능이 좋다고 보기는 힘들다.

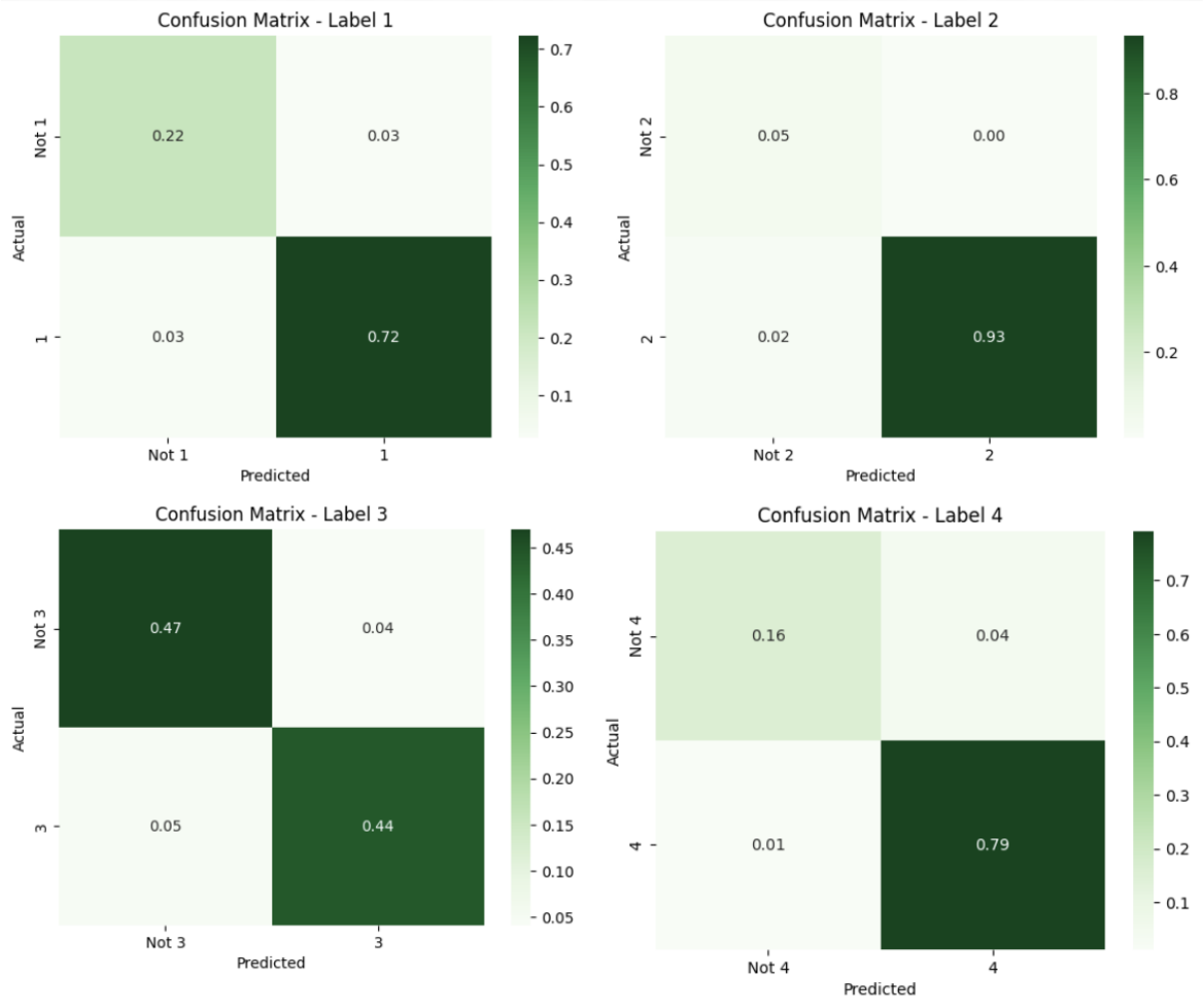


그림 46. test 데이터에 대한 CNN의 결함 유형별 혼동 행렬

표17과 그림47는 EfficientNetB3 모델에서 각 결함 유형별로 성능 평가를 한 것이다. 결함 3번, 4번에서는 높은 성능을 보이나, 결함 1번, 2번은 성능이 낮은 편이다. 특히 결함 2번은 실제 결함인 것에 대해 다른 결함으로 판단할 확률이 33.3%나 되는 것으로 보아, 해당 모델은 결함 2번에 대해서는 잘 분류하지 못한다고 볼 수 있다. EfficientNetB3에 학습된 결함 1,2번의 데이터 양이 상대적으로 작으며, 결함의 크기가 작고 주변 철강 표면과 대비되지 않기 때문인 것으로 판단된다.

결함 유형	accuracy	precision	recall	f1 score
defect 1	0.948	0.8934	0.8925	0.666
defect 2	0.974	0.666	0.666	0.666
defect 3	0.961	1.0	0.953	0.976
defect 4	0.987	0.833	1.0	0.909

표 17. test 데이터에 대한 EfficientNetB3의 결함 유형별 성능 평가 지표

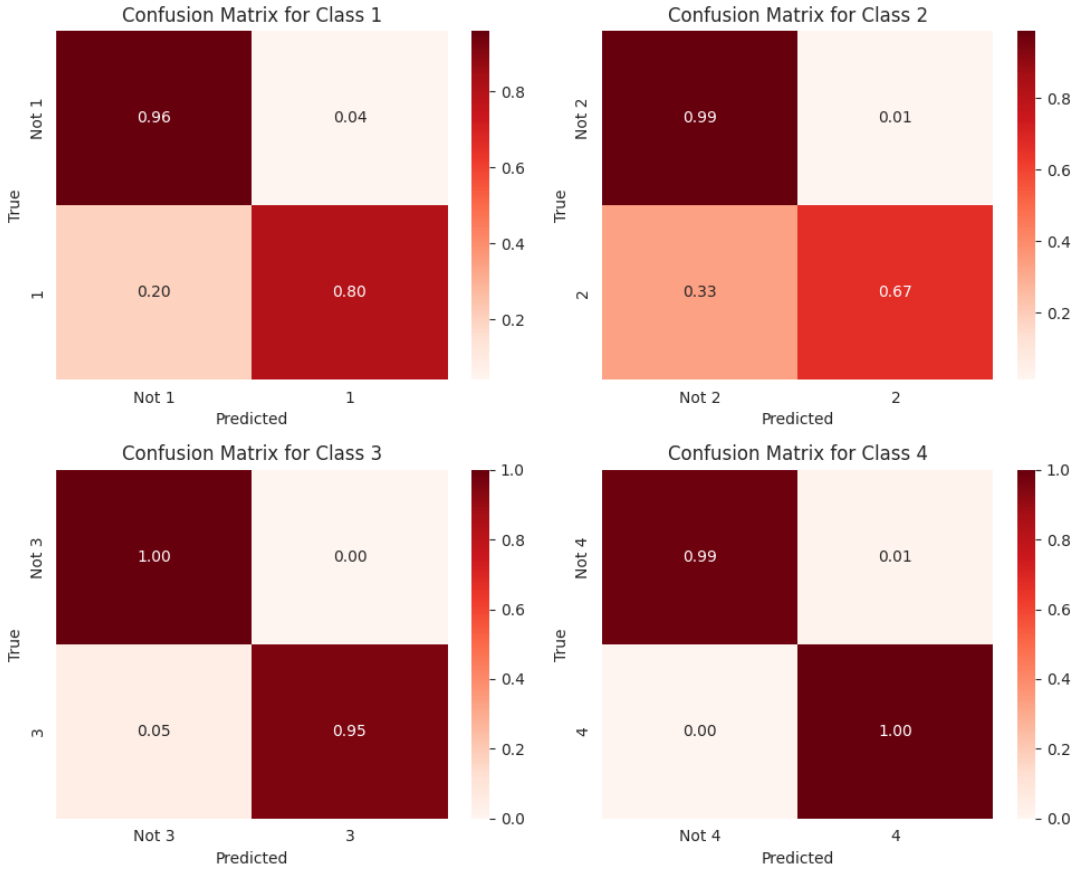


그림 47. test 데이터에 대한 EfficientNetB3의 결함 유형별 혼동 행렬

- 결론

위 세가지 모델 중 EfficientNetB3가 가장 균형 있게 예측이 잘된 모델이라고 판단이 된다. 더 다양하고 균형 잡힌 데이터셋으로 모델을 학습을 시킨다면 defect2의 성능도 높일 수 있을 것이다.

4.3. 세그멘테이션 모델

우리의 목표는 결함의 RLE 인코딩 픽셀 값을 사용하지 않고 전처리를 하여 최적의 성능을 내는 모델을 개발하는 것이었다. 그렇기에 앞선 분류 모델에서는 결함 픽셀 값을 사용하지 않고 연구해왔다.

하지만 서비스 이용자들에게 정교한 검출 정보를 전달하기 위해 RLE 데이터와 U-Net을 활용한 세그멘테이션 연구를 돌입했다. 처음 모델의 구조를 구성했을 때는 성능이 좋지 못했으나, 모델의 설정 값을 변경해가며 개선해나간 끝에 test 데이터 기준 dice coefficient값이 0.6749로 나온 모델을 추출해낼 수 있었다. 얕고 크기가 작은 결함에 대해서는 학습이 잘되지 않는다는 한계를 가지고 있지만, 그 외의 결함에 대해서는 학습이 잘 되었으며 multi-label 데이터에 대한 세그멘테이션이 잘된다는 장점을 가진다.

4.4. 서비스

4.4.1. 태블릿 환경에서의 동작 시연

이 앱은 실제 제철소 현장에서의 사용자 환경을 고려하여 설계하였다. 또한, 이 부분에 대한 멘토링에서의 피드백을 반영하여 입력 이미지 사이즈를 통일하고 동일한 사용자 환경(촬영 디바이스 및 포맷 통일 등)을 만들어 주기 위하여 시연에서는 카메라 위치를 고정하여 철강 이미지를 입력할 계획이다.

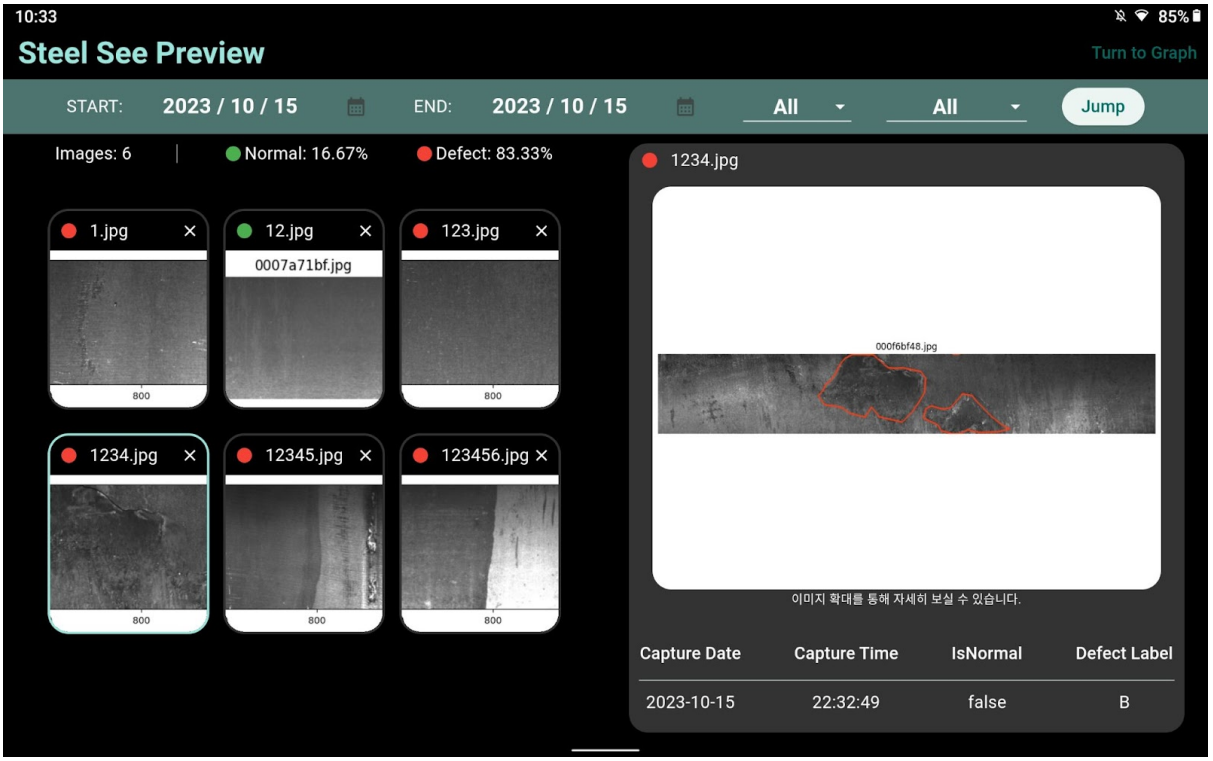


그림 48. 메인 화면 UI (앱 內)

앱의 시작 화면으로써 사용자(공장 관리자)가 제일 많이 보게 될 메인 화면이다.

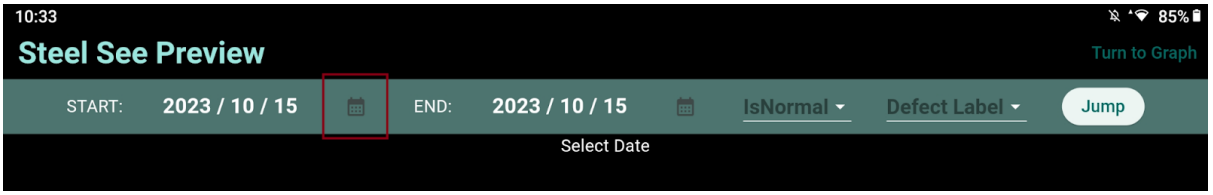


그림 49. 필터 설정 - 날짜 입력 버튼

조회하고 싶은 날짜 설정을 빨간색 테두리의 버튼으로 할 수 있다.

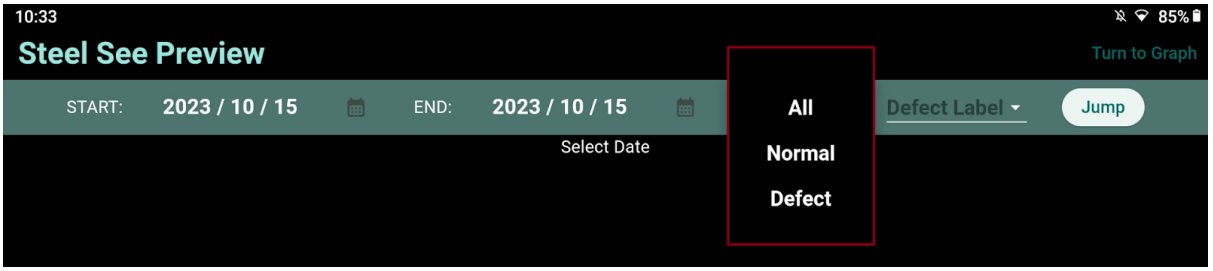


그림 50. 필터 설정 - 결함 유무에 대한 필터

결함 유무 및, 결함 Label 검색에 대한 필터 설정을 할 수 있다.

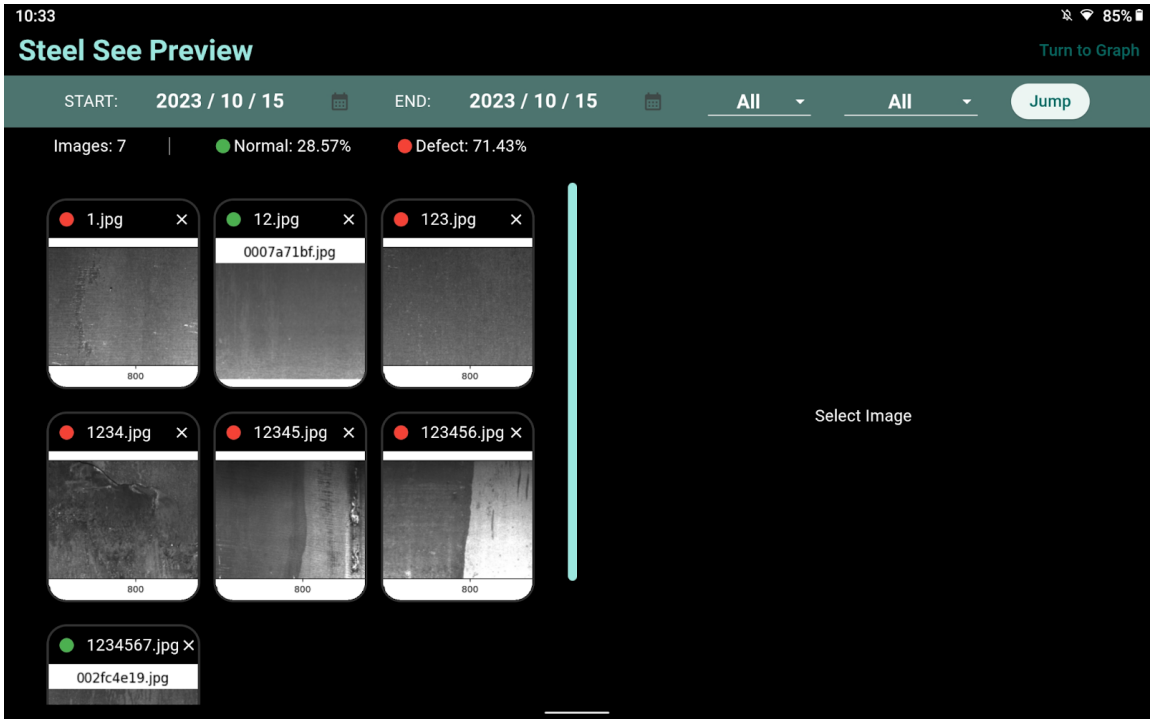


그림 51. 조회 후 화면 - 조회된 이미지, 총 이미지 갯수, 결함 비율 등을 알 수 있다.

조회 버튼을 누르면 자신이 설정한 날짜에 해당하는 촬영된 원본 이미지들을 볼 수 있다.

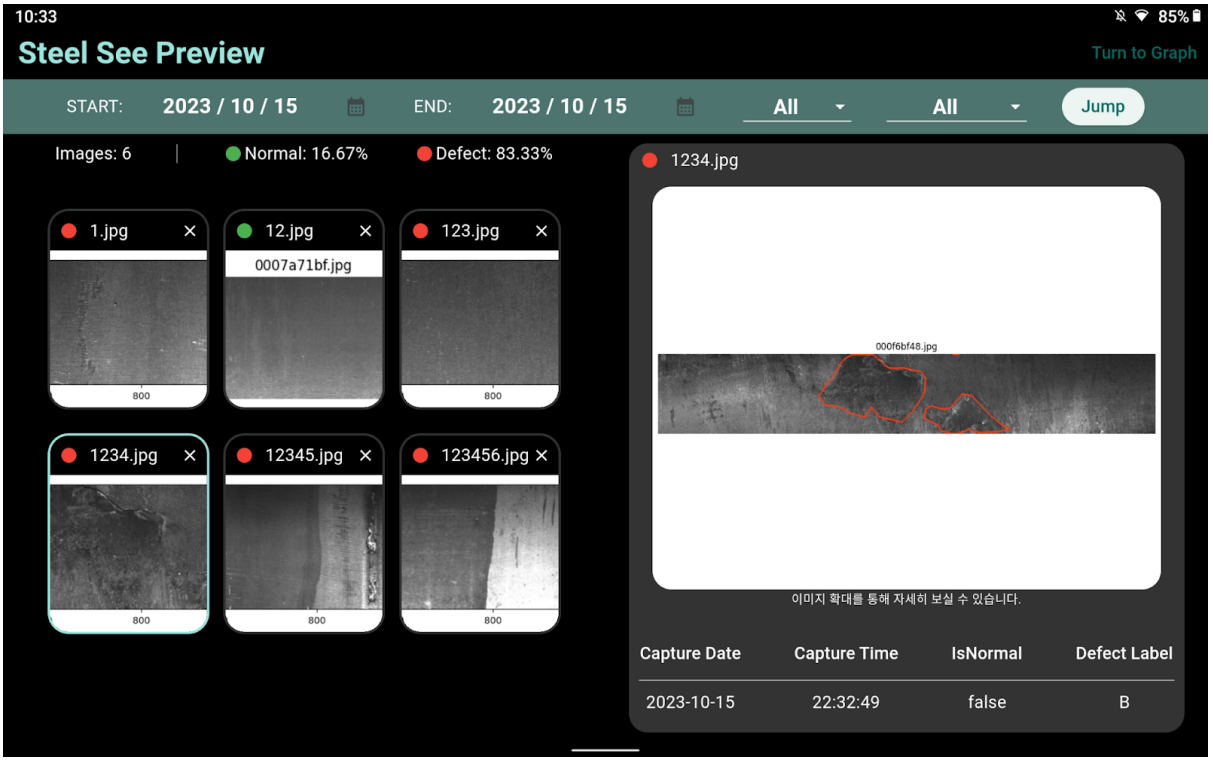


그림 52. 이미지를 선택한 뒤 화면 - 촬영된 날짜, 시간 등 데이터의 자세한 명세를 확인할 수 있다.

이미지 선택하여 detected 이미지와 이미지의 촬영된 날짜, 시간 등 자세한 명세등을 확인할 수 있다. 또한 확대 및 이동하여 이미지를 자세하게 볼 수 있다.

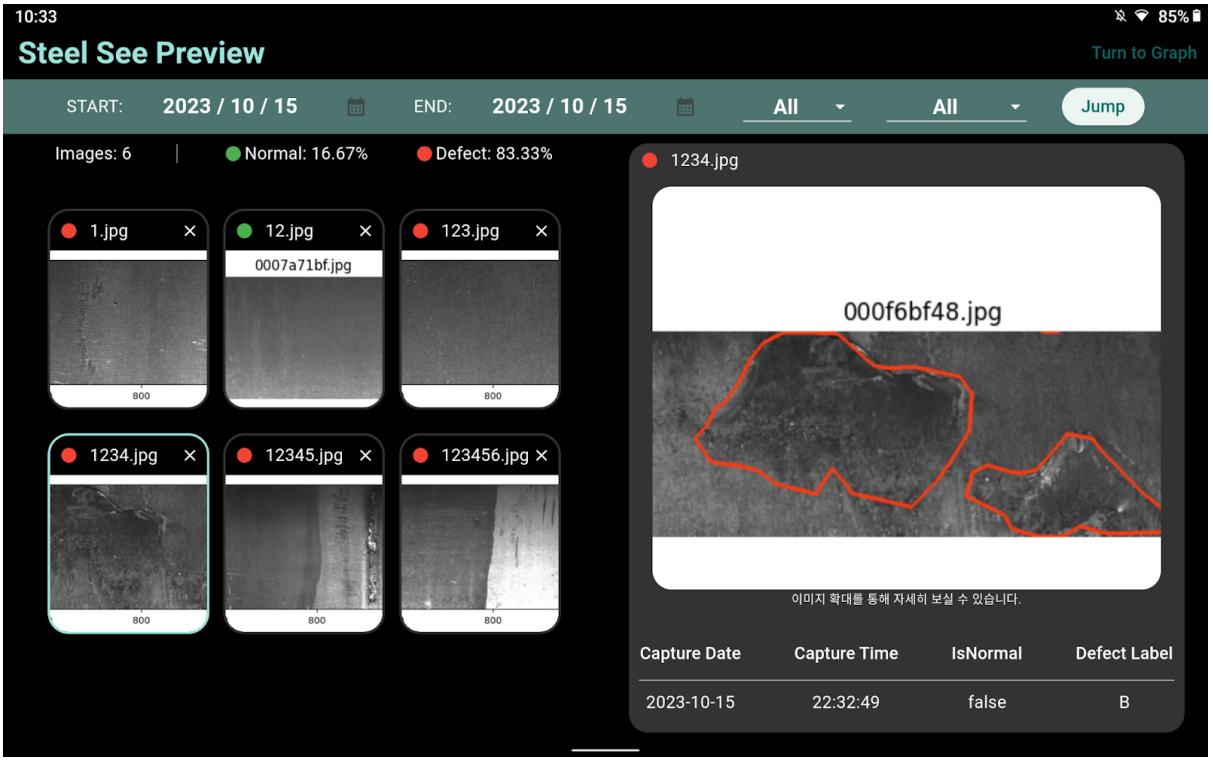


그림 53. Detected 이미지 확대 후 모습

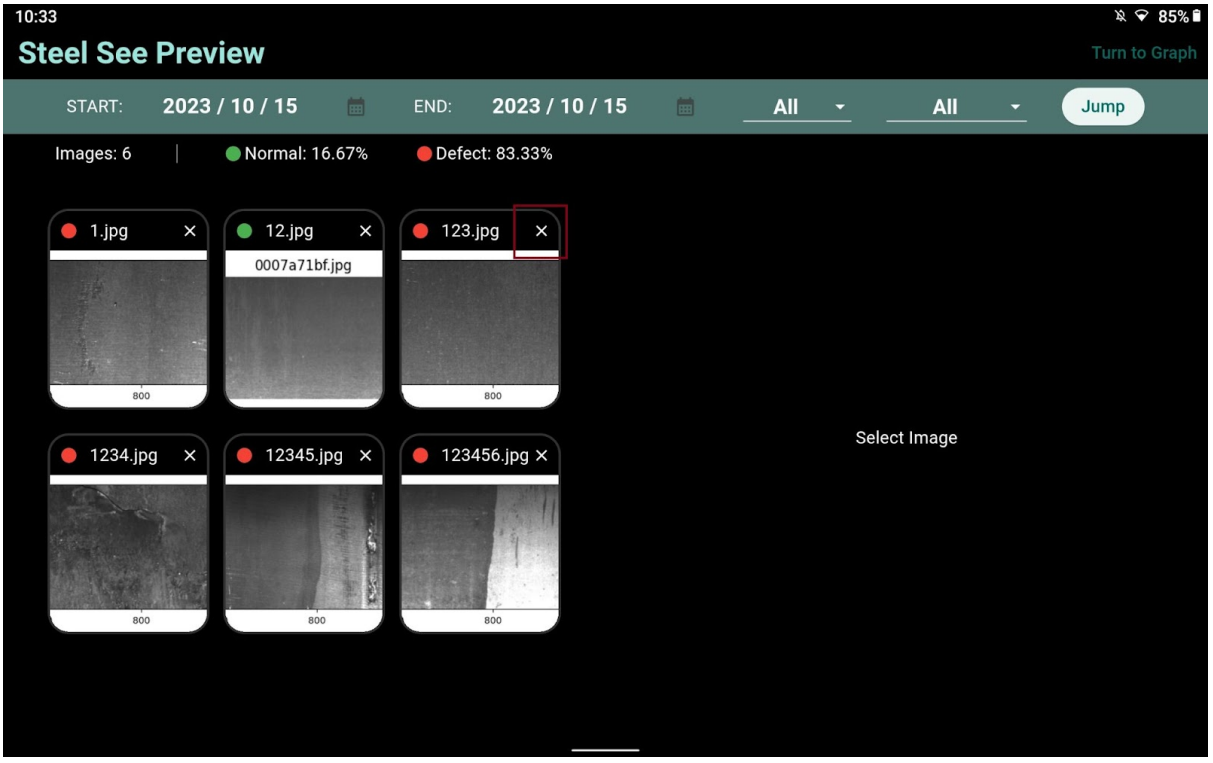


그림 54. 빨간색 테두리의 버튼을 눌러 이미지가 사라진 모습

조회하고 싶지 않은 이미지를 지울 수 있다. 되돌리기와 같은 사용자를 위한 추가적인 기능들도 구현 예정에 있다.

그림 55. 저장된 데이터의 날짜별, Label 별 그래프

대시보드와 마찬가지로 조회하고 싶은 날짜를 설정하고 그에 따른 이미지들을 그래프로 표현하여 한눈에 데이터의 변화들을 볼 수 있다. 또한 각각의 막대들을 누르면 해당 Label의 데이터들만 비교할 수 있다.

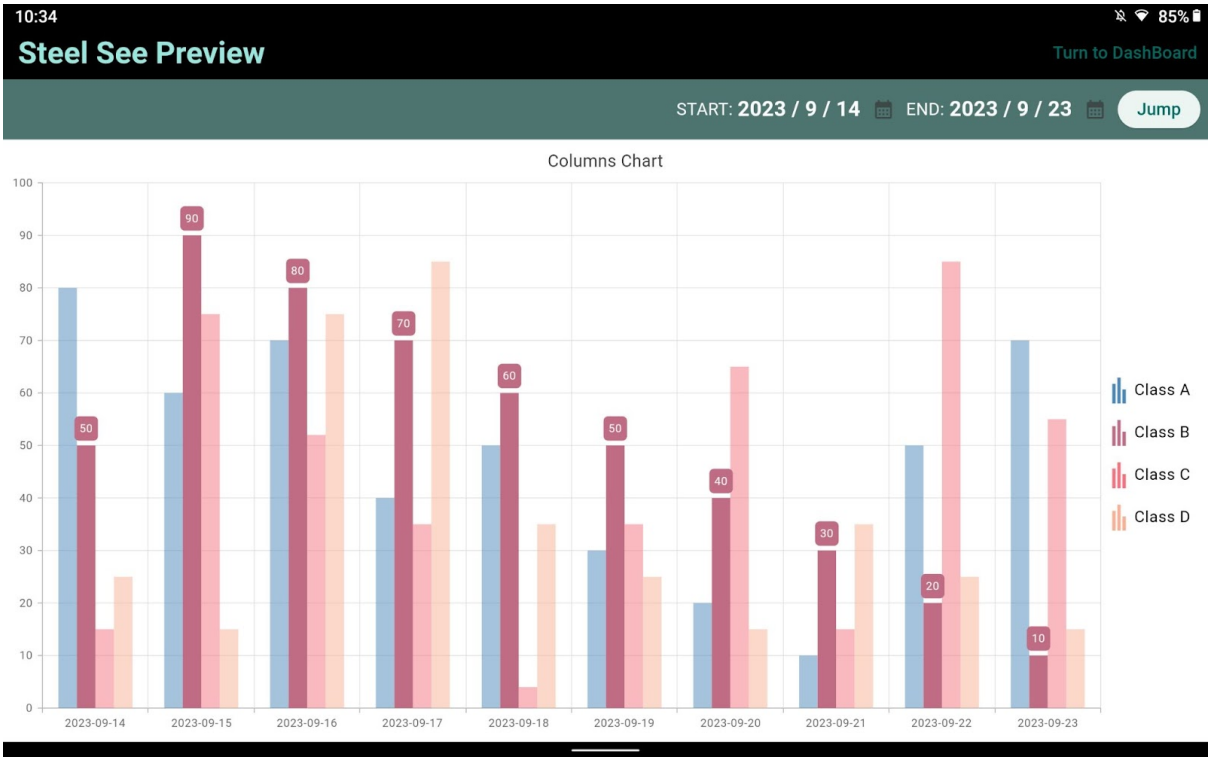


그림 56. Class B를 선택한 상태 - Class B를 제외한 나머지는 흐려진 모습

5. 멘토 의견서 반영 및 시연계획

5.1. 멘토 의견서 반영

멘토 의견서에 작성된 내용을 토대로 긍정적으로 검토하신 부분을 더 보강하고, 제시하신 의견을 모두 반영하였다.

5.1.1. 의견1 : 철판의 영상을 활용하여 불량검출율을 높이기 위해 복수개의 모델을 활용

- 다양한 모델의 특성을 분석하고 성능을 개선하기 위해 파라미터 수정 및 전처리를 적용하였다. 그 결과 CNN, ResNet, EfficientNet, Unet, DCGAN, PGGAN등 총 6개의 모델을 연구하여 정리하였다.

5.1.2. 의견2 : 전처리 단계에서의 문제를 발견하였고 이를 해결하기 위한 방안을 제시하였음

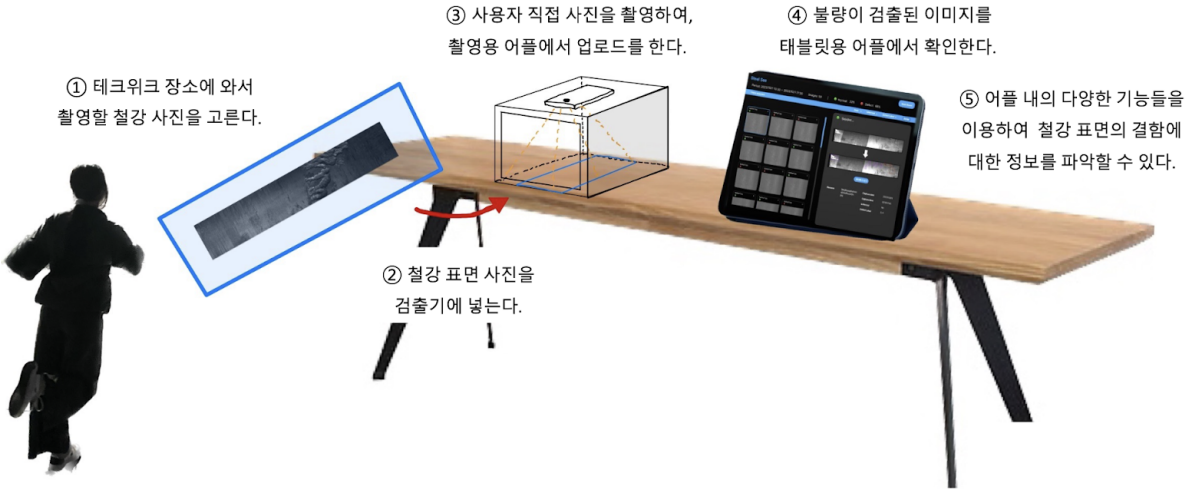
- 철강 이미지의 특수성과 모델별 특성을 고려하여 모델별 최적의 전처리를 적용하고자 하였고, 마주한 문제를 해결하는 과정에서 겪은 시행착오와 분석내용을 정리하였다.

5.1.3. 의견3 : 서비스 구조도를 통해, 실제 제철소 현장에서의 사용자환경을 고려하여 설계하였고 갱신된 과제 추진계획을 통해, CNN 성능개선, 사용자 UI 개선 및 데이터 문제를 해결하고자 하였음, 입력 이미지 사이즈를 통일하기 위해 동일한 사용자 환경(촬영디바이스 및 포맷 통일 등)을 만들어주는것도 하나의 방법으로 보임

- 실제 제철소 검출 현장을 모티브로 어플개발과 시연을 구상하였고, 입력 이미지 사이즈를 통일하기 위해 사전 조정된 카메라 위치에서 철강 이미지 촬영할 수 있게 하였다. (시연 계획 참고) 시연 단계에서 참여자의 특성을 고려하여 태블릿을 활용한 어플리케이션으로 접근성을 높이고자 하였으며, 시연 과정에 관객이 직접 참여할 수 있도록 구성하여 낯선 철강산업에 대한 진입장벽을 낮추고 이해도를 높이고자 하였다.

5.2. 시연 계획

5.2.1. 시연 시나리오



5.2.2. 참고사항

- 낮시간대는 조도 등의 요인으로 정밀한 촬영이 어려워 작은 결함 이미지는 검출이 어려울 수 있다. 이에 시연용 철강 이미지는 결함의 특성이 뚜렷한 이미지를 사용한다.
- 촬영용 카메라는 검출기의 역할로 고정되어 있고, 참여자는 고정된 카메라에서 직접 촬영버튼을 누를 수 있다.
- 철강 검출이란 낯선 분야에 대한 진입장벽을 낮추고 참여도를 높이기 위해 실제 검출 과정을 모티브로 구성하였고, 직접 촬영한 이미지를 태블릿에서 터치하며 검출 결과를 확인할 수 있다.
- 시연보다는 체험운영 방식으로 진행하며 팀원이 각 단계를 안내하며 가이드하는 방식으로 부스 체험을 진행한다.

6. 결론 및 향후 연구 방향

DCGAN을 사용하여 결함 특징을 담은 이미지를 생성할 수 있었고, 해상도를 높여 결함의 세부적인 특징을 담은 이미지를 생성할 수 있었다. 이미지 저장과정에서 노이즈가 발생하여 실제 분류모델에 사용할 수 없었다는 한계가 있지만, 소수 개체 데이터 문제를 해결하는 전처리 방법과 철강 이미지 특성에 맞는 생성 모델을 개발함으로써 이와 같은 접근법이 다른 GAN모델에 적용 가능하다는 가능성을 확인했다. 이번 연구에서 얻은 전처리 방법으로 향후 PGGAN, StyleGAN 등 각광받는 GAN 모델에 적용하여 최적의 이미지 생성 모델을 개발할 계획이다.

대표적인 분류모델들의 성능을 비교하여 철강 이미지 특성에 맞는 학습모델을 개발하였다. 분류 모델에서는 대규모의 데이터로 사전 학습된 모델을 사용함으로써 모델의 성능을 높일 수 있었다. 세그멘테이션 모델에서는 모델의 설정 값들을 변경하면서 mask 값을 추출하는데에 성공했다. EfficientNetB3 분류 모델과 U-Net 세그멘테이션 모델을 이용하여 철강 검출 및 분류 서비스 형태로 배포했다.

분류 및 세그멘테이션 모델에서 DCGAN을 활용하지 못했다는 점이 많은 아쉬움으로 남지만, 해당 부분을 성공시켜 향후 최적의 불량 검출 및 분류 모델을 개발하고 논문으로 작성해볼 예정이다.

7. 참고 문헌

[1] I. B. Ann (2021. May 7) Introduction to Steel Surface Defects Detection
[2] J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville and Y. Bengio, “Generative Adversarial Nets," in NIPS, 2014.
[3] J. Goodfellow, Alec Radford, Luke Metz, “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks” ,7 Jan 2016
[4] Tero Karras, Timo Aila, Samuli Laine, Jaakko Lehtinen, “Progressive Growing of GANs for Improved Quality, Stability, and Variation”, 26 Feb 2018
[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, “Deep Residual Learning for Image Recognition”, 10 Dec 2015
[6] Tajeddine Benbarrad, Lamiae Eloutouate ,Mounir Arioua, Fatiha Elouaai, Driss Laanaoui, “Impact of Image Compression on the Performance of Steel Surface Defect Classification with a CNN”, 16 Dec 2021
[7] W. Choi, Y. Kim, J. Jo, D. Lee, S. Kim, S. Park, J. Kang, J.K Gahm, “Steel surface defect classification using ResNet50”,, Dec 2021
[8] aladdinpersson, “Machine-Learning-Collection : ProGAN” [Online]. Available: https://github.com/aladdinpersson/Machine-Learning-Collection/tree/master/ML/Pytorch/GANs/ProGAN
[9] H. S. Kim, H. S. Lee, “Generative Adversarial Networks based Data Generation Framework for Overcoming Imbalanced Manufacturing Process Data”, 1 Feb 2019
[10] Severstal: Steel Defect Detection [Online]. Available: https://www.kaggle.com/competitions/severstal-steel-defect-detection/data?select=train_images
[11] Mingxing Tan, Quoc V. Le, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks”, 11 Sep 2020
[12] Olaf Ronneberger, Philipp Fischer, and Thomas Brox, “U-Net: Convolutional Networks for Biomedical Image Segmentation", 18 May 2015