Adam O'Connor
Jacob Whited

Final Project Report

Introduction

In the aviation industry, ensuring the structural integrity of aircraft is paramount to maintaining safety and regulatory compliance. Recent incidents involving fuselage cracks, panel detachments, and manufacturing defects have highlighted the limitations of traditional inspection methods, which are often labor-intensive, time-consuming, and susceptible to human error. As the global aircraft fleet continues to age and expand, the need for more precise, efficient, and automated defect detection systems has become critical.

In response to this challenge, our project focuses on developing a deep learning-based solution for aircraft fuselage defect detection. Utilizing the Aircraft_Fuselage_DET2023 dataset from IEEE DataPort, we aim to leverage convolutional neural networks (CNNs) to accurately classify surface defects across four defect types. We started with a small set of labeled images and used the CNN to train an initial model. To expand our dataset, we applied pseudo-labeling techniques to label the large pool of unlabeled images.

To track performance and ensure reproducibility, our PyTorch-based training pipeline logs hyperparameters, model architecture, evaluation metrics, and visualizes key aspects such as loss curves, learned kernels, and feature maps from early layers. By combining rigorous evaluation, thoughtful model design, and domain-specific best practices informed by FAA standards, this project aims to contribute toward safer and more reliable defect detection in the aviation industry.

Objectives

Develop an Accurate Classification Model:

Design and train a convolutional neural network capable of accurately classifying aircraft fuselage images into four categories based on defect type.

Address Class Imbalance:

Apply data augmentation and class weighting strategies to mitigate the effects of class imbalance in the dataset and improve overall model robustness.

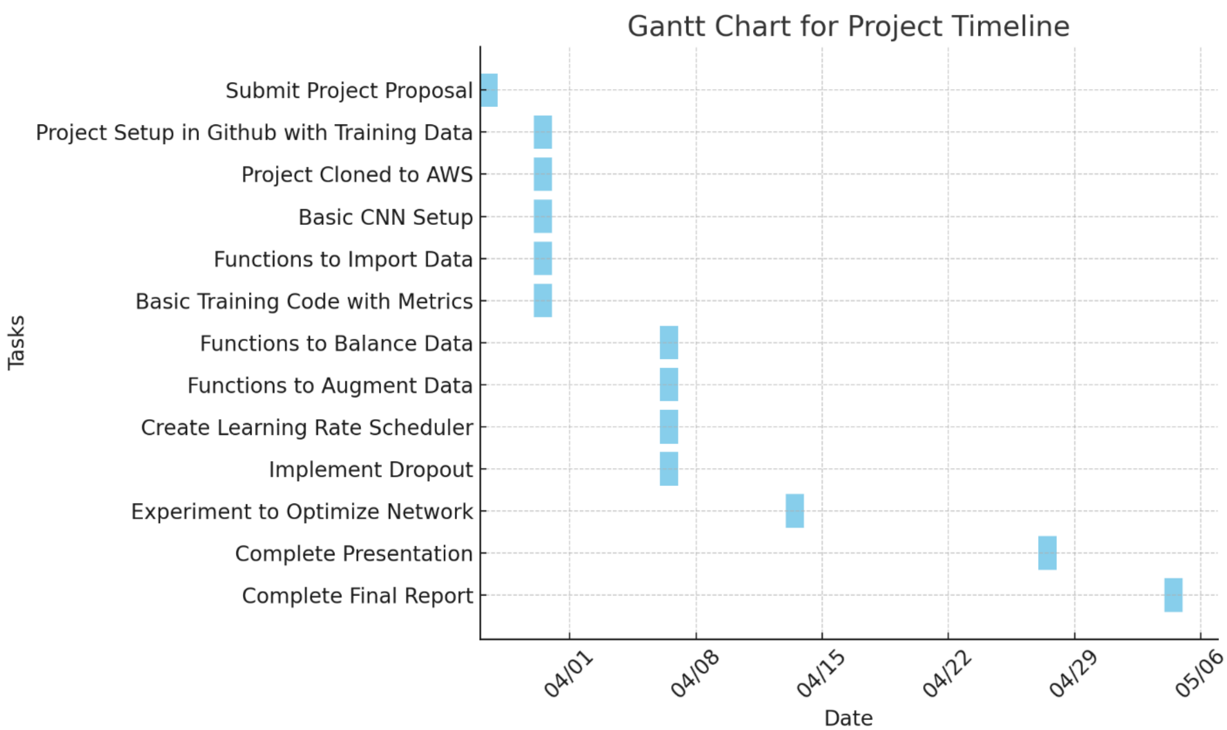
Optimize for Practical Deployment:

Evaluate and optimize the model based on performance metrics (accuracy, F1 scores), prediction time, and memory usage to ensure feasibility for real-world aircraft maintenance applications.

Expand dataset using Pseudo-labeling and manual grading techniques:

Expand the initial limited labeled dataset to include the expanse of unlabeled images included in the dataset and ensure the accuracy of the newly labeled images using manual grading.

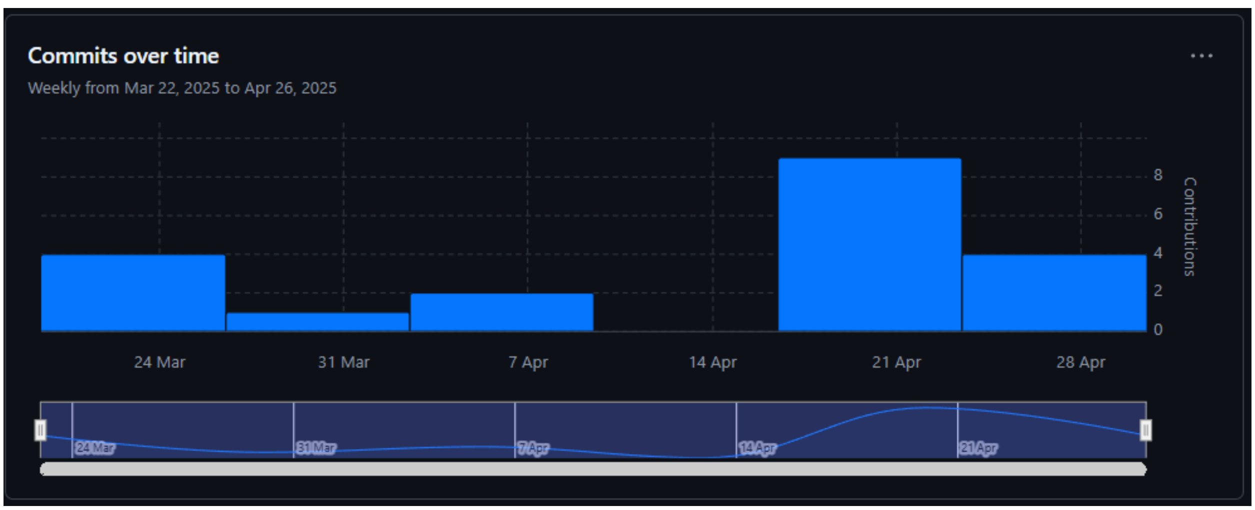
Project Schedule



The project followed a structured schedule, as outlined in the Gantt chart above. The above chart was a draft of what we would like to accomplish that was made at the beginning of the project whenever the project proposal was submitted. Early tasks matched up with the dates described above, while more time intensive tasks were pushed back to mid-April.

Key milestones included the submission of the project proposal and setting up the GitHub repository with the initial training data by early April. Shortly thereafter, the project was cloned to our GitHub to enable close collaboration in development. Foundational coding tasks, such as building a basic CNN architecture, developing functions for data import, and implementing metric-tracking during training, were completed between March 31st and April 7th. Subsequent efforts focused on balancing the dataset, applying data augmentation techniques, setting up our pseudo-labeling code, and masking techniques to increase classification accuracy were worked on from mid to late April. By the end of April, focus shifted toward preparing the final presentation and finalizing our best model.

Tasks were divided collaboratively among team members, with Jake Whited handling the initial data import functions, pseudo-labeling techniques, and manual grading, Adam O'Connor focusing on data balancing, augmentation, CNN modification, and image masking, and both members jointly conducted experiments, analysis, and work on final deliverables. Project progress was tracked using GitHub Projects, ensuring transparency and version control.



This graph from GitHub shows initial work at the beginning of April where we uploaded our groundwork code and dataset and the bulk of our work starting in mid-April.

Dataset

Our dataset is Aircraft_Fuselage_DET2023 dataset from IEEE DataPort. The dataset contains over 5600 high resolution images of various common defects on the fuselage of aircraft. It's a newly built dataset for developing defect detection models for advanced industrial defect detection systems. A high definition camera was used to photograph different parts of the aircraft fuselage in different lighting environments.

However, there are only 389 images that have defect annotations. The defects present were scratches, paint peeling, rivet damage, and rust. Annotations for the labeled images are provided in multiple formats (COCO .json, VOC .xml, and YOLO .txt), facilitating flexibility across different machine learning pipelines. For this project, we utilized the COCO format .json annotations. The labels in these files come along with image coordinates where each defect is located, sometimes containing multiple defects in one image. There are an additional 5212 images that we use for pseudo-labeling and testing our model. While this is a multi-label classification problem , the annotations only list a single type of defect, even if other defects exist in the image. For example, a lot of rivet damage also contains rust, but only rivet damage is listed in the annotations.

All images are high-resolution, RGB images, with a fixed size of **1120 × 960 pixels**.

The dataset is structured into four main folders:

- aircraft_fuselage_coco: COCO-style annotated data (images + .json labels)
- aircraft_fuselage_voc: VOC-style annotated data (images + .xml labels)

- aircraft_fuselage_yolo: YOLO-style annotated data (images + .txt labels)
- unlabel_aircraft_fuselage: Unlabeled images

The 389 annotated images were used for supervised training and validation. The 5212 unlabeled images were used for testing our model by assigning pseudo-labels to the images. Those labels were then evaluated by performing manual grading.

This dataset enabled comprehensive training and evaluation of the defect detection model, particularly in the presence of real-world industrial challenges like data imbalance and high-resolution inputs.

Description deep learning network and training algorithm.

Background

This project focuses on multi-label image classification, where each image can potentially belong to multiple classes simultaneously (e.g., a fuselage defect might fall under several categories). We developed a custom CNN model trained on a small, labeled dataset and later a larger set of pseudo-labeled data for detecting these defects.

The dataset was processed to extract annotations per image, with targets binarized using MultiLabelBinarizer.

Model Architecture

The proposed model is a convolutional neural network (CNN) designed for multi-label classification of aircraft fuselage images. The architecture begins with a series of convolutional layers that extract hierarchical spatial features from the input images. Each convolutional block consists of a convolutional layer followed by a ReLU activation function and a max-pooling layer for spatial downsampling. These convolutional layers progressively capture edge-level, texture-level, and shape-level features relevant for defect identification.

The output from the final convolutional block is flattened and passed through a fully connected (dense) layer with ReLU activation, which projects the learned features into a lower-dimensional latent space. This is followed by an output layer composed of sigmoid-activated neurons—one for each class label—to produce independent probabilities for each possible defect class. The use of sigmoid activation enables multi-label output, where multiple classes can be predicted simultaneously for a single image.

The final model used for this project is a custom convolutional neural network (CNN) designed to classify aircraft fuselage defects. It combines three parallel convolutional branches to capture features at multiple receptive field sizes, followed by standard convolutional and fully connected layers.

Architecture Overview

- **Input:** RGB image with shape (3, IMAGE_SIZE, IMAGE_SIZE)
- **Parallel Convolutional Branches:**
    - Conv2d(3, 8, kernel size=7, stride=1, padding=3)
    - Conv2d(3, 8, kernel size=5, stride=1, padding=2)

  - Conv2d(3, 8, kernel size=3, stride=1, padding=1)
  - Outputs from these branches are concatenated → total of 24 channels

- **Main Convolutional Path:**
  - Conv2d(24, 32, kernel size=3, padding=1) + ReLU
  - MaxPool2d(kernel size=2, stride=2)
  - MaxPool2d(kernel size=2, stride=2) (applied again for downsampling)

- **Fully Connected Layers:**
  - Flatten the output
  - Linear(flattened size, 128) + ReLU
  - Linear(128, num_classes)

Padding & Stride Calculation

The following equation calculates the output size of a convolution layer:

- For all initial conv layers: Padding = (kernel size - 1) // 2 → preserves spatial dimensions.
- MaxPooling reduces each spatial dimension by half.

Dynamic Flatten Size Calculation

Instead of hardcoding the flattened feature size for the first fully connected layer, the model creates a dummy input and passes it through _forward_conv_layers() to compute the correct shape. This makes the model adaptable to different input sizes and more maintainable.

Training Algorithm

The CNN is trained using a binary cross-entropy loss function (BCEWithLogitsLoss), which is appropriate for multi-label classification tasks. The loss is computed independently for each class and averaged across all classes. Formally, the model outputs a logit ziz_izi for each class iii, and the sigmoid activation transforms it into a probability σ(zi)\sigma(z_i)σ(zi). The loss for each label is calculated based on the ground truth label yiy_iyi and the predicted probability σ(zi)\sigma(z_i)σ(zi), penalizing both false positives and false negatives. During inference, class-wise thresholds are applied to the output probabilities to determine which labels are active, and these thresholds can be tuned per class to optimize F1 macro performance.

The training pipeline consists of a supervised phase where the model is trained using 300 manually labeled images and a subsequent semi-supervised phase incorporating pseudo-labeled data from 5,200 unlabeled samples. In the supervised phase, data augmentation and preprocessing are applied to enhance model robustness, and the model is optimized using Adam or SGD with early stopping and learning rate scheduling.

Key Equations

$$Loss = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \times \log\left(\sigma(x_i)\right) + (1 - y_i) \times \log\left(1 - \sigma(x_i)\right)\right]$$

Where:

- $y_i \in \{0,1\}$: ground truth for class i
- x_i: model output (logit)

- $\sigma(x) = \frac{1}{1+e^{-x}}$: sigmoid function
- N: number of labels per sample

$$F1_{macro} = \frac{1}{C}\sum_{i=1}^{C}\frac{2 \times Precision_i \times Recall_i}{Precission_i + Recall_i}$$

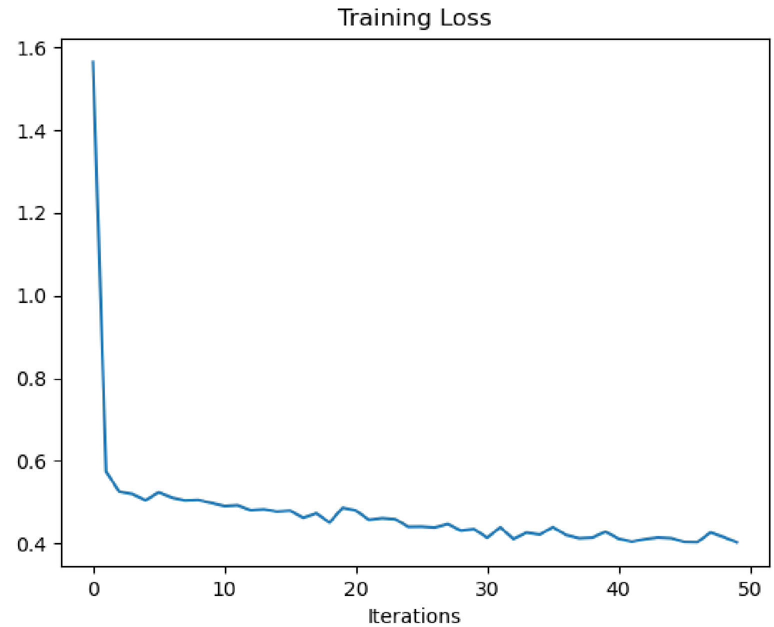
Key Figures

First Training Image and Label

Annotation: [{"image": "001.jpg", "annotations": [{"label": "scratch", "coordinates": {"x": 44.78313253012057, "y": 419.57228915662654, "width": 36.0, "height": 41.0}}, {"label": "scratch", "coordinates": {"x": 737.2831325301206, "y": 398.07228915662654, "width": 33.0, "height": 32.0}}]}]

Sample Loss Curve

This is a sample plot of the training loss curve taken from early on in our development.



Experimental Setup

Data Pre-Processing

For preprocessing, we split the labeled images into training and testing using an 80/20 split. We also balanced the dataset as the training set only contained a class count of 69 rust, 180 scratch, 55 rivet damage, and 85 paint peel images, so we balanced the data to have 180 instances of each class giving us a total training set of 720 images. We resized images to 250 by 250 pixels for faster training while keeping the image large enough to detect small defects like scratches and rust. We extracted labels from the COCO-format JSON files and applied multi-label binarization since one image could potentially have multiple defects. In addition, we performed data augmentations:

- Random horizontal and vertical flips
- Random resizing and cropping of the image

- Random color jitter altering brightness, contrast, saturation, and hue
- A random rotation between 1 and 90 degrees

We trained the model using the Adam optimizer and binary cross-entropy loss, appropriate for multi-label problems. We used a batch size of 30 and a learning rate of 0.001. We leveraged some of the code from the labs and exams such as saving the model with the best ~~f1 score and accuracy~~

Model Architecture Summary

Model Architecture

The defect detection model is based on a convolutional neural network (CNN) architecture specifically tailored for training on a relatively small dataset. The network design consists of:

Convolutional layers with ReLU activations

Max pooling layers for spatial downsampling

Fully connected layers followed by a SoftMax output for multi-class classification

The model was trained using the Adam optimizer and binary cross-entropy loss, which is appropriate for multi-label problems. The training process used a batch size of 30 and a learning rate of 0.001. Several elements from previous lab and exam exercises were incorporated into our approach—for example, we reused functionality for saving model checkpoints with the best F1 score and accuracy.

Masking Strategy for Cleaner Labels

One of the key challenges was that each image in the dataset contains annotations for only one defect type, even though other defects may be visible in other parts of the image. Simply feeding the full image to the model would risk penalizing it for correctly identifying defects that weren't labeled.

To avoid this, we implemented a masking strategy using the bounding box annotations to isolate only the labeled portion of the image:

A custom subclass of the Dataset class was created.

The __init__() method was extended to store not just file paths and labels, but also all available annotations and bounding boxes.

The __getitem__() method was modified to apply masking before any augmentation is performed, so that only the relevant portion of the image is visible to the model.

Bounding Box Bias and Augmentation

Although masking helped reduce false negatives, it introduced a new problem: the model could learn bounding box shapes and positions rather than defect-specific features. For example, it might associate square bounding boxes with rivet damage or rectangular ones with scratches.

To mitigate this risk, we introduced two key augmentations:

Aggressive Random Rotation: Images were randomly rotated up to 90 degrees to eliminate consistent orientation cues.

Random Bounding Box Expansion: Before applying the mask, the height and width of each bounding box were randomly expanded, resulting in varied sizes and shapes of the visible region.

Each time the dataset loads an image, these strategies ensure that the model sees a different version of the same sample, with a randomized mask position and shape. In tests using the "paint peel" class, for example, four consecutive views of the same image showed dramatically different bounding box patterns, helping prevent the model from learning the shape of the annotation rather than the defect itself.

Framework

The model was developed using **PyTorch**, taking advantage of its modular design, GPU acceleration capabilities, and utilities from the torchvision package for transformations and data loading.

Handling Overfitting and Data Imbalance

To reduce overfitting, we used aggressive data augmentations including:

Random rotations (up to 90°)

Flipping and color jittering

To address the imbalance between the four defect types (scratch, paint peel, rivet damage, rust), we:

[Insert specific strategy used—e.g., oversampling, weighted loss function, or stratified sampling.]

Data Loader

We built a custom CustomDataset class extending PyTorch's Dataset. This class loads image files, applies masking based on bounding box annotations, performs random augmentations, and encodes multi-label targets using MultiLabelBinarizer. The dataset also supports shuffling and split logic to assign samples to training and testing subsets.

Metrics

To assess the performance of our multi-label classification model, we employed a diverse set of evaluation metrics, each offering a unique perspective on the model's behavior. The macro-average F1 score is our primary metric, as it computes the F1 score independently for each class and then averages the results, treating all classes equally. This is particularly important given the class imbalance in our dataset, ensuring that rare defect classes are not overshadowed by more frequent ones.

We also measure accuracy, which indicates the overall proportion of correctly predicted labels, but in multi-label tasks it can be overly optimistic if some classes dominate. Hamming loss complements this by quantifying the fraction of labels that are incorrectly predicted—either falsely included or missed, making it a useful indicator of label-wise performance.

To evaluate the agreement between predicted and true class distributions beyond simple accuracy, we use Cohen's kappa score, which adjusts for the agreement occurring by chance. Since it requires single-label targets, we apply argmax to obtain the most likely class per sample. Similarly, Matthew's correlation coefficient (MCC)—also computed using argmax—provides a balanced measure of classification quality, even with imbalanced data, by considering true and false positives and negatives across all classes.

Together, these metrics offer a comprehensive evaluation framework that addresses the unique challenges of multi-label, imbalanced classification in our defect detection task.

Testing and Pseudo-Labels

In this function, we implement a multi-label pseudo-labeling framework to facilitate weakly supervised learning on unlabeled aircraft fuselage imagery. CNN, mirroring the one used for training the model, is designed with three parallel convolutional branches employing 7×7, 5×5, and 3×3 kernels to capture spatial features at multiple scales. The outputs of these branches are concatenated and passed through additional convolutional, pooling, and fully connected layers. A pre-trained model is loaded and deployed through a ModelTester module, which performs inference on unlabeled images using sigmoid-activated outputs and applies a fixed threshold, 0.4, to identify relevant defect classes. For ambiguous predictions below threshold, the most confident class is selected as a fallback. The resulting pseudo-labels are stored in a csv file and optionally refined via a manual grading interface, which allows a user to inspect, validate, and correct model predictions. This semi-automated approach balances model-driven annotation with expert feedback, enhancing label quality for downstream training.

Breakdown of classes

- Paint peel actually turned out to be pretty good at being recognized, with seemingly no false positives in the random grabs for manual grading.

- The rust labels seemed okay, but it's hard to tell if the model is actually picking up on the rust, or just the background color, as they all seem to be basically the same color.
- Rivet damage is another one where it seems to pick up on any rivets it finds period. However, this may be a misunderstanding with the dataset itself. Every image in the dataset is assumed to be a defect, therefore, any rivet in an image is assumed to contain rivet damage, even if it does not appear so from inspection.
- Scratch appears correct, although you could argue that pretty much every image could be labeled with a scratch.

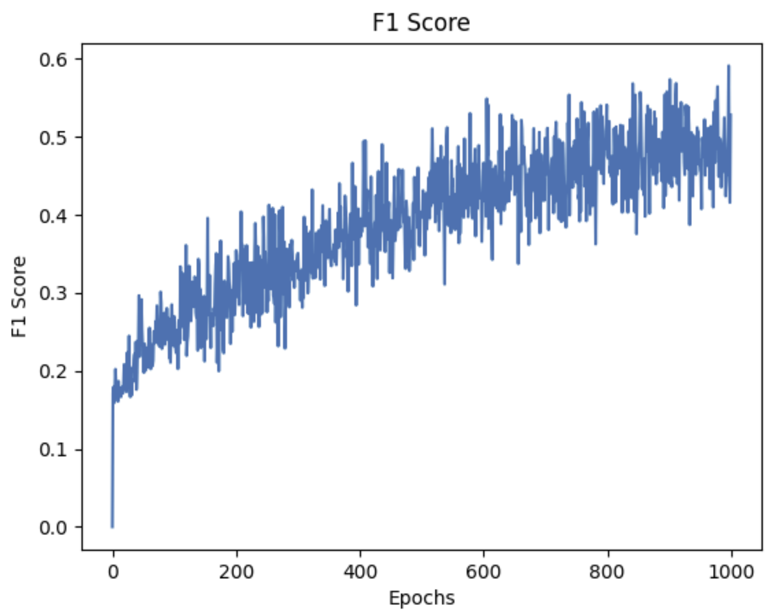
Results

We faced a few challenges: first, the small, labeled dataset, which limited our training effectiveness, however, we overcame this through data balancing and augmentation. Second, the lack of multi-class labels, which is not a problem for training, but might indicate a less accurate dataset, we address this by assigning confidence-based pseudo-labels. Third, the complexity of predicting sprawling defects, like paint peel and rust, which we handled by expanding our masking technique to incorporate more of the original image to provide context for the training by using varying kernel sizes.

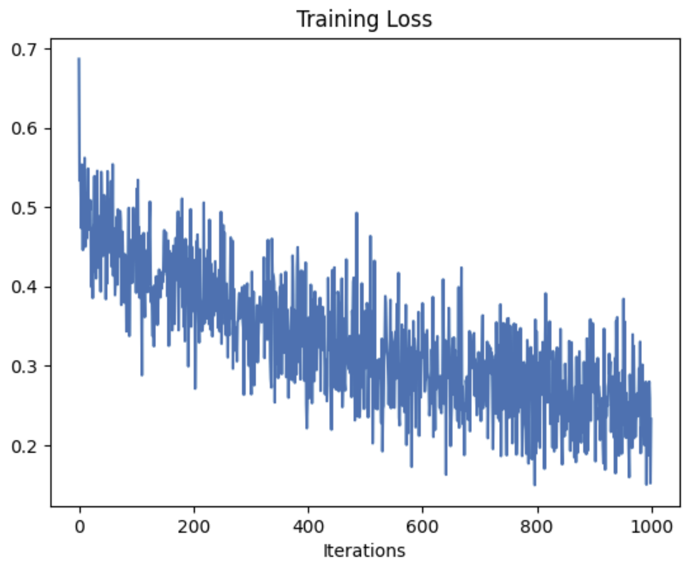
Our best model ended up being one with 3 parallel kernels of different sizes in the first layer, which get concatenated together before going into the second layer. This way model can compare different sized image features.

After making these updates, we saw some, but not drastic improvements to our F1 score, and the paint peel still hovered around 10 to 20, however applying the model to the unlabeled set, where no masking is used, we could see that it went from labeling near 0 as paint peel, to 763 out of the 5000. Even though we can't get an exact score from the unlabeled data, since there is no ground truth and even manual grading leaves room for error, it was clear from manually reviewing the images our model labeled that this model was generalizing significantly better than the other models.

F1 Score

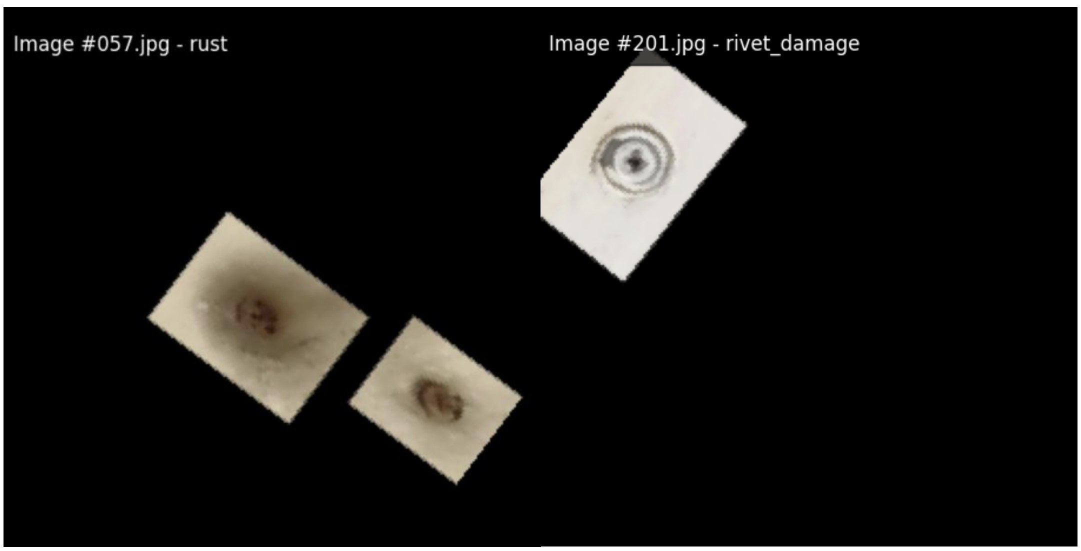


Loss vs Epoch



As you can see our model, while noisy, decreased in loss over iterations.

Masking Features



Kernels

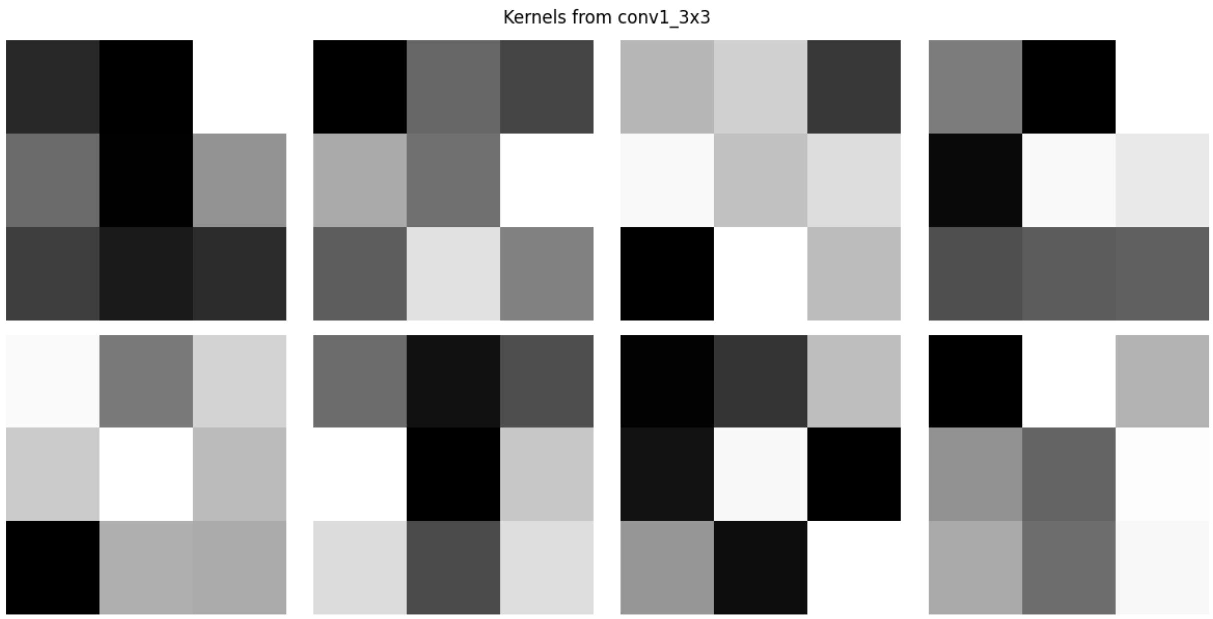


Figure 1 3x3 Kernels from Initial Layer

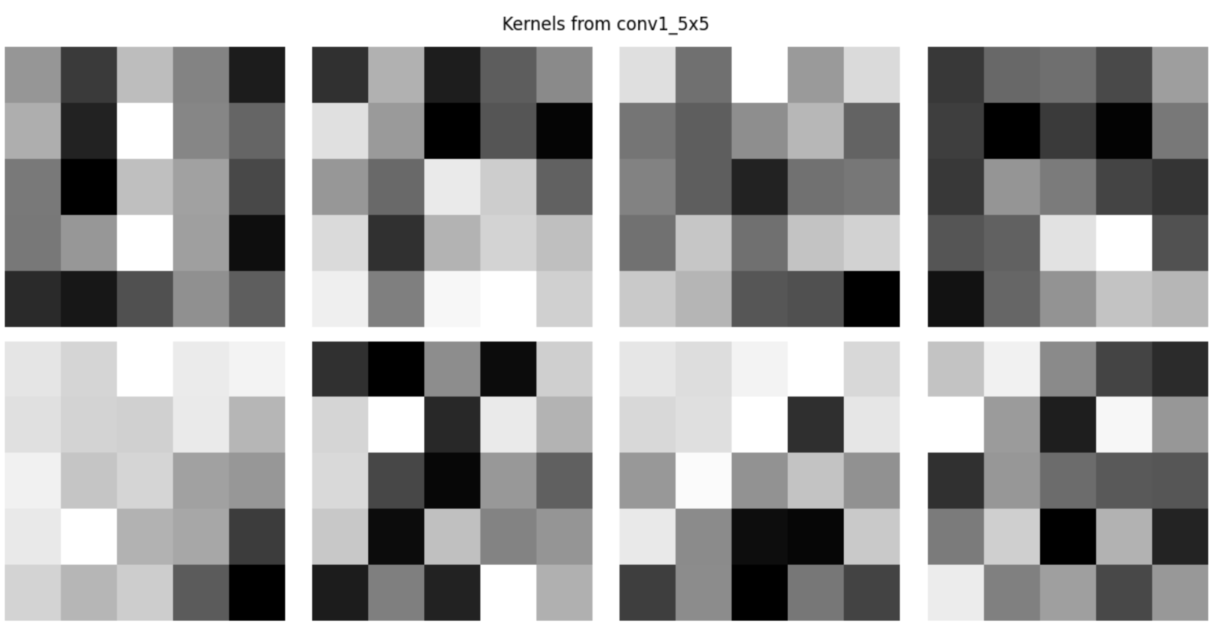


Figure 2 5x5 Kernels from Initial Layer

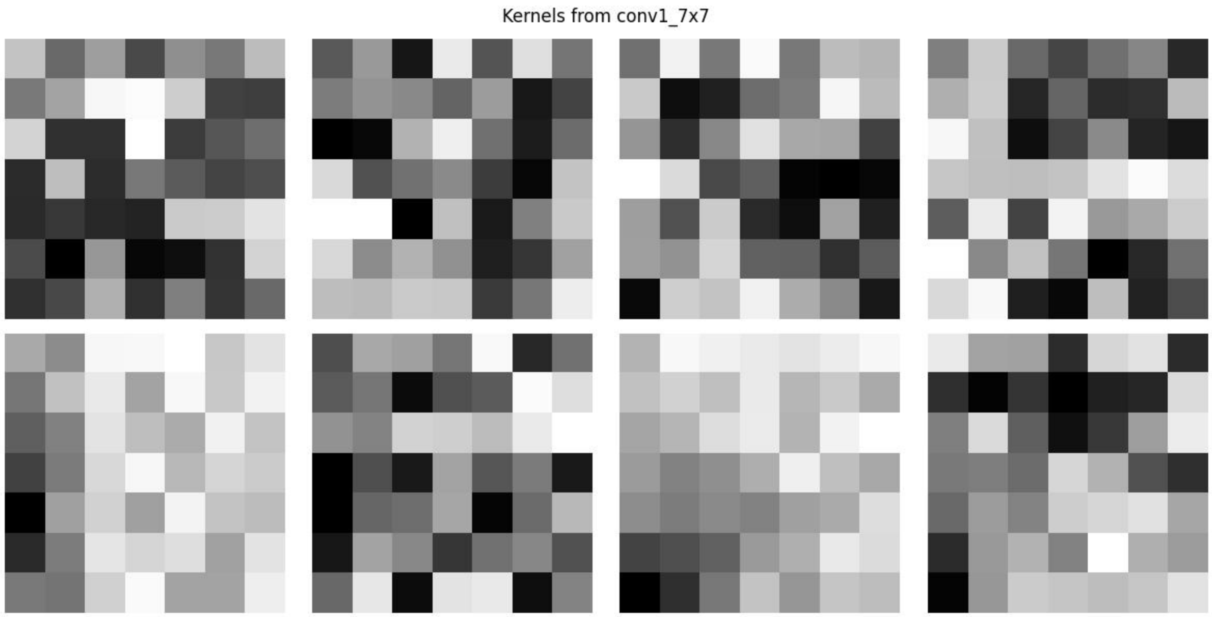


Figure 37x7 Kernels from Initial Layer

Classification Report

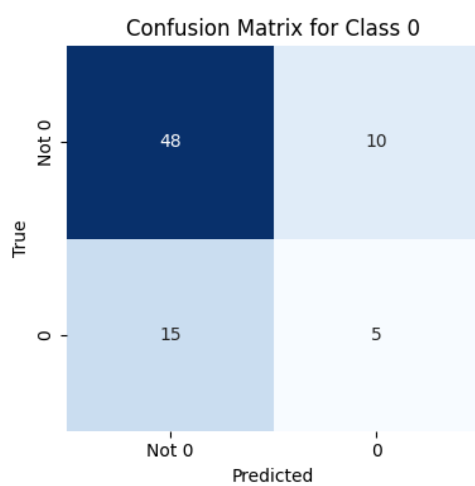
	precision	recall	f1-score	support
Paint peel	1.00	0.10	0.18	20
Rivet damage	0.50	0.71	0.59	7
rust	0.50	0.14	0.22	14
scratch	0.66	0.89	0.76	37
micro avg	0.64	0.54	0.58	78

macro avg	0.67	0.46	0.44	78
weighted avg	0.70	0.54	0.50	78
samples avg	0.54	0.54	0.54	78

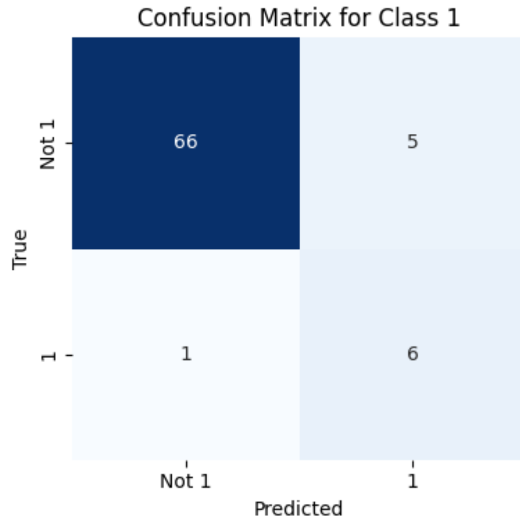
This is our classification report based on our testing set which was composed of 78 images. While our model does very well at classifying scratches and rivet damage, we have a fairly low recall and f1 score for paint peel and rust. We believe this is because paint peel and rust need the context of the entire image to properly classify, as they can sprawl out, and our masking blacks out the entire image except for the defect features. This makes small individual defects easy to classify, but is the opposite for larger, often broken up defects.

Confusion Matrixes

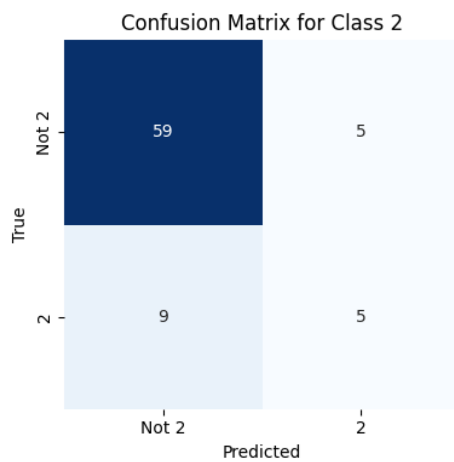
- Class 0: Scratch:



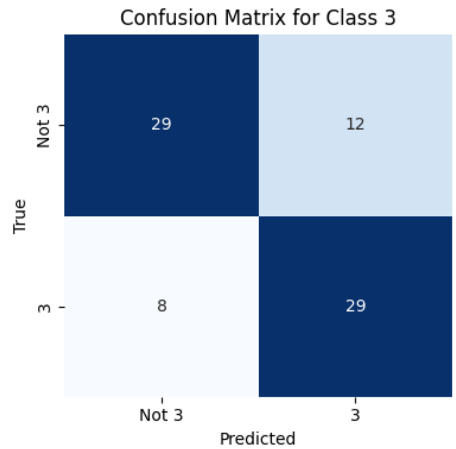
- Class 1:



- Class 2:



- Class 3:



Summary

This project successfully developed a convolutional neural network (CNN)-based solution for multi-label classification of aircraft fuselage defects, addressing key limitations of traditional inspection methods. By leveraging the Aircraft_Fuselage_DET2023 dataset and implementing a two-phase training pipeline—initial supervised learning followed by semi-supervised learning through pseudo-labeling—we were able to significantly expand our training data and improve model generalization.

Our final model architecture, which incorporates three parallel convolutional branches with varying kernel sizes, demonstrated the ability to extract features at multiple spatial resolutions, improving its robustness across defect types. Data augmentation, class balancing, and a custom masking strategy were critical to reducing overfitting and improving performance in the presence of noisy and incomplete annotations. Although the model exhibited strong performance in classifying localized defects such as scratches and rivet damage, it showed lower recall and F1 scores for more complex, context-dependent defects like rust and paint peel—likely due to the masking approach limiting available spatial information.

Despite these challenges, the model's ability to generalize effectively to unlabeled data and its integration with a manual grading pipeline for refining pseudo-labels highlights the practical potential of this approach in real-world aircraft maintenance settings. Through thoughtful model design, rigorous evaluation, and adherence to domain-specific best practices, this project contributes a scalable and reproducible framework for automated fuselage defect detection in aviation safety applications.

References

"A Semi-Supervised Aircraft Fuselage Defect Detection Network with Dynamic Attention and Class-aware Adaptive Pseudo-Label Assignment"

Xiaoyu Zhang, Jinping Zhang, Jiusheng Chen, Runxia Guo, Jun Wu, "Aircraft_Fuselage_DET2023: An Aircraft Fuselage Defect Detection Dataset", IEEE Dataport, August 14, 2023, doi:10.21227/3ref-ex71

Appendix

Train_Model.py

```python
import numpy as np
import torch.nn as nn
from torch.utils import data
from sklearn.preprocessing import MultiLabelBinarizer
from sklearn.metrics import multilabel_confusion_matrix, classification_report
import matplotlib.pyplot as plt
from datetime import datetime
from config import *
from dataset_loader import CustomDataset, load_json_annotations
from model import CNN
from metrics import evaluate_metrics
import shutil
import inspect
import json
import seaborn as sns

# Main training loop
def train_model():  1 usage  aoco3995 +1
    timestamp = datetime.now().strftime("%Y_%m_%d_%H:%M:%S")
    log_dir = os.path.join(BASE_DIR, "training_logs", f"{NICKNAME}_{timestamp}")
    os.makedirs(log_dir, exist_ok=True)
    script_dir = os.path.join(log_dir, "scripts")
    os.makedirs(script_dir, exist_ok=True)

    # Save model
    model_path = os.path.join(log_dir, f"model_{NICKNAME}.pt")

    # Save full path to a text file
    path_txt_file = os.path.splitext(model_path)[0] + "_path.txt"  # same name, but "_path.txt"

    with open(path_txt_file, "w") as f:
        f.write(os.path.abspath(model_path))  # write full absolute path

    source_files = [
        inspect.getfile(inspect.currentframe()),  # this script
        os.path.join(BASE_DIR, "config.py"),
        os.path.join(BASE_DIR, "dataset_loader.py"),
        os.path.join(BASE_DIR, "metrics.py"),
        os.path.join(BASE_DIR, "model.py")
    ]

```

```python
    for src_path in source_files:
        if os.path.exists(src_path):
            dst_path = os.path.join(script_dir, os.path.basename(src_path).replace(".py", "_script.txt"))
            shutil.copy2(src_path, dst_path)
            print(f"Copied {src_path} to {dst_path}")
        else:
            print(f"Warning: {src_path} not found and was not copied.")

    # Save hyperparameters
    hyperparams = {
        "IMAGE_SIZE": IMAGE_SIZE,
        "BATCH_SIZE": BATCH_SIZE,
        "LR": LR,
        "N_EPOCH": n_epoch,
        "THRESHOLD": THRESHOLD,
        "DEVICE": device
    }

    with open(os.path.join(log_dir, "hyperparameters.txt"), "w") as f:
        for key, val in hyperparams.items():
            f.write(f"{key}: {val}\n")

    df = load_json_annotations(JSON_FOLDER)

    df = (
        df
        .groupby("id")
        .agg({
            "target": lambda x: ",".join(sorted(set(x))),
            "data":   "first"
        })
        .reset_index()
    )

    #df = df.groupby('id')['target'].apply(lambda x: ','.join(sorted(set(x)))).reset_index()

    df['split'] = ['train' if i % 5 != 0 else 'test' for i in range(len(df))]

    mlb = MultiLabelBinarizer()
    class_names = sorted(set(','.join(df['target']).split(',')))
    label_map = {label: idx for idx, label in enumerate(class_names)}
```

```python

    train_df = df[df['split'] == 'train'].reset_index(drop=True)
    test_df = df[df['split'] == 'test'].reset_index(drop=True)

    train_set = CustomDataset(train_df, label_map)
    test_set = CustomDataset(test_df, label_map)

    # Length of Training Data
    print('The number of training examples is:', str(len(train_set)))
    # Length of Testing Data
    print('The number of testing examples is:', str(len(test_set)))
    # The shape of the first feature
    print('The shape of the first feature is:')
    print(train_set[0][0].shape)
    # The first label
    print('The first label is:' + str(train_set[0][1]))
    plt.figure()
    plt.imshow(train_set[0][0][0], cmap='gray')
    plt.savefig(os.path.join(log_dir, "first_feature.png"))
    plt.close()

    train_loader = data.DataLoader(train_set, batch_size=BATCH_SIZE, shuffle=True)
    test_loader = data.DataLoader(test_set, batch_size=BATCH_SIZE, shuffle=False)

    model = CNN(num_classes=len(label_map), image_size=IMAGE_SIZE).to(device)

    print(model)
    # Save model architecture to txt
    model_info_path = os.path.join(log_dir, "model_architecture.txt")
    with open(model_info_path, "w") as f:
        f.write("Model Architecture:\n")
        f.write(str(model))
        f.write("\n\nTotal Parameters: {}\n".format(
            sum(p.numel() for p in model.parameters())
        ))
        f.write("Trainable Parameters: {}\n".format(
            sum(p.numel() for p in model.parameters() if p.requires_grad)
        ))
    print(f"Saved model architecture to: {model_info_path}")

    optimizer = torch.optim.Adam(model.parameters(), lr=LR)
    criterion = nn.BCEWithLogitsLoss()

```

```python
        best_f1 = 0
        metrics_log = []  # Store metrics for each epoch
        total_loss = []
        ind = []
        f1_scores = []
        for epoch in range(n_epoch):
            model.train()
            running_loss = 0
            for batch_idx, (inputs, targets) in enumerate(train_loader):
                inputs, targets = inputs.to(device), targets.to(device)
                optimizer.zero_grad()
                outputs = model(inputs)
                loss = criterion(outputs, targets)
                loss.backward()
                optimizer.step()
                running_loss += loss.item()
                if batch_idx % 100 == 0:
                    total_loss.append(loss.item())
                    ind.append(batch_idx + epoch * len(train_loader) / BATCH_SIZE)
                    print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                        epoch, batch_idx * len(inputs), len(train_loader.dataset),
                               100. * batch_idx / len(train_loader), loss.item()))

            model.eval()
            preds, reals = [], []
            with torch.no_grad():
                for inputs, targets in test_loader:
                    inputs = inputs.to(device)
                    outputs = model(inputs)
                    probs = torch.sigmoid(outputs).cpu().numpy()
                    preds.extend((probs > THRESHOLD).astype(int))
                    reals.extend(targets.numpy())

            metrics = evaluate_metrics(np.array(reals), np.array(preds))
            f1_scores.append(metrics['f1_macro'])
```

```python
        epoch_metrics = {
            "epoch": epoch + 1,
            "loss": running_loss,
            "f1_macro": metrics["f1_macro"],
            "accuracy": metrics["accuracy"],
            "f1_micro": metrics.get("f1_micro"),
            "f1_weighted": metrics.get("f1_weighted"),
            "hamming_loss": metrics.get("hamming"),
            "cohen_kappa": metrics.get("cohen"),
            "matthews_corrcoef": metrics.get("mcc")
        }
        metrics_log.append(epoch_metrics)

        # Calculate multilabel confusion matrices
        conf_matrices = multilabel_confusion_matrix(np.array(reals), np.array(preds))

        # Create a directory to save confusion matrices if not exists
        conf_dir = os.path.join(log_dir, "confusion_matrices")
        os.makedirs(conf_dir, exist_ok=True)


        # Generate classification report
        report = classification_report(np.array(reals), np.array(preds), target_names=list(label_map.keys()),
                                       zero_division=0)

        # Save classification report to a text file
        with open(os.path.join(log_dir, "classification_report.txt"), "w") as f:
            f.write(report)

        print(
            f"Epoch {epoch + 1}: Loss={running_loss:.4f}, F1={metrics['f1_macro']:.4f}, Accuracy={metrics['accuracy']:.4f}")
```

```python

        if metrics['f1_macro'] > best_f1 and SAVE_MODEL:
            best_f1 = metrics['f1_macro']
            torch.save(model.state_dict(), model_path)
            print("Model saved!")

            # Save each class confusion matrix separately
            for class_idx, cm in enumerate(conf_matrices):
                plt.figure(figsize=(4, 4))
                sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
                            xticklabels=[f'Not {class_idx}', f'{class_idx}'],
                            yticklabels=[f'Not {class_idx}', f'{class_idx}'])
                plt.title(f'Confusion Matrix for Class {class_idx}')
                plt.xlabel('Predicted')
                plt.ylabel('True')
                plt.tight_layout()
                plt.savefig(os.path.join(conf_dir, f"confusion_matrix_class_{class_idx}_epoch_{epoch + 1}.png"))
                plt.close()

    with open(os.path.join(log_dir, "metrics_log.json"), "w") as f:
        json.dump(metrics_log, f, indent=4)
    print(f"Saved metrics log to {os.path.join(log_dir, 'metrics_log.json')}")

    plt.figure()
    plt.plot(total_loss)
    plt.title('Training Loss')
    plt.xlabel('Iterations')
    plt.savefig(os.path.join(log_dir, "loss_curve.png"))
    plt.close()

    plt.figure()
    plt.plot(f1_scores)
    plt.title('F1 Score')
    plt.xlabel('Epochs')
    plt.ylabel('F1 Score')
    plt.savefig(os.path.join(log_dir, "f1_score.png"))
    plt.close()

```

```
    # Access the three parallel conv layers
    conv_layers = {
        "7x7": model.conv1_7x7,
        "5x5": model.conv1_5x5,
        "3x3": model.conv1_3x3
    }

    # --- Visualize kernels ---
    for name, conv in conv_layers.items():
        weights = conv.weight.data.cpu().numpy()  # Shape: (out_channels, in_channels, k, k)
        num_filters = weights.shape[0]
        fig, axes = plt.subplots(nrows=num_filters // 4 + 1, ncols=4, figsize=(12, 3 * (num_filters // 4 + 1)))
        axes = axes.flatten()

        for i in range(num_filters):
            axes[i].imshow(weights[i, 0], cmap='gray')  # Show 1st channel of each filter
            axes[i].axis('off')
        for i in range(num_filters, len(axes)):
            axes[i].axis('off')  # Hide unused subplots

        plt.suptitle(f"Kernels from conv1_{name}")
        plt.tight_layout()
        plt.savefig(os.path.join(log_dir, f"kernels_{name}.png"))
        plt.close()

```

```
    # --- Visualize feature maps on a sample image ---
    model.eval()
    with torch.no_grad():
        first_batch = next(iter(train_loader))
        image_tensor = first_batch[0][0].unsqueeze(0).to(device)  # Add batch dim

        for name, conv in conv_layers.items():
            feature_maps = conv(image_tensor).cpu().squeeze(0)  # Shape: (C, H, W)

            num_maps = feature_maps.shape[0]
            fig, axes = plt.subplots(nrows=num_maps // 4 + 1, ncols=4, figsize=(12, 3 * (num_maps // 4 + 1)))
            axes = axes.flatten()

            for i in range(num_maps):
                axes[i].imshow(feature_maps[i], cmap='gray')
                axes[i].axis('off')
            for i in range(num_maps, len(axes)):
                axes[i].axis('off')

            plt.suptitle(f"Feature maps from conv1_{name}")
            plt.tight_layout()
            plt.savefig(os.path.join(log_dir, f"feature_maps_{name}.png"))
            plt.close()


if __name__ == '__main__':
    train_model()
```

Dataset_loader.py

```python
from torch.utils import data
import json
import cv2
import pandas as pd
from torchvision import transforms
from config import *
import torch
import math
import random
import numpy as np

# Load JSON annotations
def load_json_annotations(json_folder):  aoco3995
    rows = []

    for fname in os.listdir(json_folder):
        if fname.endswith(".json"):
            with open(os.path.join(json_folder, fname), "r") as f:
                data = json.load(f)
                for item in data:
                    image = item["image"]
                    for ann in item["annotations"]:
                        label = ann["label"]
        rows.append({"id": image, "target": label, "data": data[0]})  # Corrected: inside the loop

    # Sort rows by "id"
    rows = sorted(rows, key=lambda x: x['id'])

    # Count current samples per class
    class_counts = {
        'rust': 0,
        'scratch': 0,
        'rivet_damage': 0,
        'paint_peel': 0
    }

    for row in rows:
        if row['target'] in class_counts:
            class_counts[row['target']] += 1

    print(f"Current class counts: {class_counts}")

    # Find the maximum class size
    max_count = max(class_counts.values())
```

```
    # Oversample: duplicate rows for underrepresented classes
    balanced_rows = []

    for target_class in class_counts.keys():
        # Extract all rows with this target
        class_rows = [row for row in rows if row['target'] == target_class]

        # Duplicate as needed
        if len(class_rows) < max_count:
            multiplier = max_count // len(class_rows)
            remainder = max_count % len(class_rows)
            balanced_class_rows = class_rows * multiplier + random.sample(class_rows, remainder)
        else:
            balanced_class_rows = class_rows

        balanced_rows.extend(balanced_class_rows)

    print(f"Balanced dataset size: {len(balanced_rows)}")

    rows = balanced_rows

    class_counts = {
        'rust': 0,
        'scratch': 0,
        'rivet_damage': 0,
        'paint_peel': 0
    }

    for row in rows:
        if row['target'] in class_counts:
            class_counts[row['target']] += 1

    print(f"Current class counts: {class_counts}")

    # Shuffle for randomness
    random.shuffle(balanced_rows)

    return pd.DataFrame(balanced_rows)
```

```python

# Dataset Class
class CustomDataset(data.Dataset):
    def __init__(self, df, label_map):
        self.df = df
        self.label_map = label_map
        self.transform = transforms.Compose([
            transforms.ToPILImage(),
            transforms.RandomHorizontalFlip(p=0.5),          # Randomly flip horizontally
            transforms.RandomVerticalFlip(p=0.2),            # Randomly flip vertically (less common)
            transforms.RandomResizedCrop(
                size=(IMAGE_SIZE, IMAGE_SIZE),
                scale=(0.8, 1.0),   # Random zoom between 80% and 100%
                ratio=(0.9, 1.1)    # Allow a little squishing/stretching
            ),
            transforms.ColorJitter(
                brightness=0.2,
                contrast=0.2,
                saturation=0.2,
                hue=0.02           # Slight color variations
            ),
            transforms.RandomRotation(degrees=90),             # Small random rotations
            transforms.ToTensor()
        ])

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

    def __getitem__(self, idx):
        row = self.df.iloc[idx]
        img_path = os.path.join(DATA_DIR, row['id'])
        image = cv2.imread(img_path)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        # Save a copy of the *original* image to pull from
        original_image = image.copy()

        change = 2

        black_image = np.zeros_like(image)

        collected_targets = set()
```

```
        for ann in row['data']['annotations']:
            if change > 1:
                x = int(math.floor(ann['coordinates']['x']))
                y = int(math.floor(ann['coordinates']['y']))
                current_target = row['target']
            else:
                x = random.randint( a: 0, image.shape[1])
                y = random.randint( a: 0, image.shape[0])
                current_target = 'none'

            w = int(ann['coordinates']['width']) * (1 + random.randint( a: 5,  b: 30) * 0.1)
            h = int(ann['coordinates']['height']) *  (1 + random.randint( a: 5,  b: 30) * 0.1)

            y1 = max(int(y - h / 2), 0)
            y2 = min(int(y + h / 2), image.shape[0])
            x1 = max(int(x - w / 2), 0)
            x2 = min(int(x + w / 2), image.shape[1])

            # Important: Always copy from the original unmodified image
            black_image[y1:y2, x1:x2, :] = original_image[y1:y2, x1:x2, :]

            for label in current_target.split(','):
                collected_targets.add(label)

        image = black_image

        image = self.transform(image)

        target = [0] * len(self.label_map)
        for label in collected_targets:
            if label in self.label_map:
                target[self.label_map[label]] = 1

        return image, torch.FloatTensor(target)#, original_image, row['id']
```

Model.py

```
import torch.nn as nn
import torch.nn.functional as F

# CNN Model
class CNN(nn.Module):  aoco3995
    def __init__(self, num_classes, image_size):  aoco3995
        super(CNN, self).__init__()

        # === PARALLEL CONVS (input: 3 channels) ===
        self.conv1_7x7 = nn.Conv2d( in_channels: 3,  out_channels: 8, kernel_size=7, stride=1, padding=3)
        self.conv1_5x5 = nn.Conv2d( in_channels: 3,  out_channels: 8, kernel_size=5, stride=1, padding=2)
        self.conv1_3x3 = nn.Conv2d( in_channels: 3,  out_channels: 8, kernel_size=3, stride=1, padding=1)
        # Total output after concat: 8+8+8 = 24 channels

        # === CONV + POOL ===
        self.conv2 = nn.Conv2d( in_channels: 24,  out_channels: 32, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d( kernel_size: 2,  stride: 2)

        # === DYNAMIC FC CALCULATION ===
        dummy_input = torch.zeros(1, 3, image_size, image_size)
        x = self._forward_conv_layers(dummy_input)
        flattened_size = x.view(1, -1).size(1)

        self.fc1 = nn.Linear(flattened_size,  out_features: 128)
        self.fc2 = nn.Linear( in_features: 128, num_classes)

    def _forward_conv_layers(self, x):  2 usages  aoco3995
        # Apply all 3 in parallel
        out_7x7 = F.relu(self.conv1_7x7(x))
        out_5x5 = F.relu(self.conv1_5x5(x))
        out_3x3 = F.relu(self.conv1_3x3(x))

        # Concatenate along the channel dimension
        x = torch.cat( tensors: (out_7x7, out_5x5, out_3x3), dim=1)  # Shape: (B, 24, H, W)

        # Continue through rest of CNN
        x = self.pool(F.relu(self.conv2(x)))  # Downsample
        x = self.pool(x)  # Downsample again
        return x

    def forward(self, x):  aoco3995
        x = self._forward_conv_layers(x)
        x = x.view(x.size(0), -1)  # Flatten
        x = F.relu(self.fc1(x))
        return self.fc2(x)
```

Unlabeled_Test_Mars.py

```
import os
import cv2
import torch
import torch.nn as nn
import pandas as pd
from torchvision import transforms
from tqdm import tqdm
from sklearn.preprocessing import MultiLabelBinarizer
import numpy as np
from torch.utils import data
import matplotlib.pyplot as plt
import torch.nn.functional as F

IMAGE_SIZE = 250
NUM_CLASSES = 4
NICKNAME = "MARS"
device = 'cuda:0' if torch.cuda.is_available() else 'cpu'

LABEL_MAP = {
    "Scratch": 0,
    "Paint Peel": 1,
    "Rivet Damage": 2,
    "Rust": 3,
    "none": 4
}  # Your label -> index dictionary
BASE_DIR = os.path.dirname(os.path.abspath(__file__))
DATA_DIR = os.path.join(BASE_DIR, "Dataset", "Aircraft_Fuselage_DET2023", "unlabel_aircraft_fuselage")

UNLABELED_IMAGES = sorted([
    f for f in os.listdir(DATA_DIR)
    if f.lower().endswith(('.jpg', '.jpeg', '.png'))
])

n_epoch = 1
BATCH_SIZE = 30
LR = 0.001

mlb = MultiLabelBinarizer()
THRESHOLD = 0.4
SAVE_MODEL = True
```

```python
#------------------------------------------------------------------------------------------------
#---- Define the model ---- #

class CNN(nn.Module):  aoco3995 +1
    def __init__(self, num_classes, image_size):  aoco3995 +1
        super(CNN, self).__init__()

        # === PARALLEL CONVS (input: 3 channels) ===
        self.conv1_7x7 = nn.Conv2d( in_channels: 3,  out_channels: 8, kernel_size=7, stride=1, padding=3)
        self.conv1_5x5 = nn.Conv2d( in_channels: 3,  out_channels: 8, kernel_size=5, stride=1, padding=2)
        self.conv1_3x3 = nn.Conv2d( in_channels: 3,  out_channels: 8, kernel_size=3, stride=1, padding=1)
        # Total output after concat: 8+8+8 = 24 channels

        # === CONV + POOL ===
        self.conv2 = nn.Conv2d( in_channels: 24,  out_channels: 32, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d( kernel_size: 2,  stride: 2)

        # === DYNAMIC FC CALCULATION ===
        dummy_input = torch.zeros(1, 3, image_size, image_size)
        x = self._forward_conv_layers(dummy_input)
        flattened_size = x.view(1, -1).size(1)

        self.fc1 = nn.Linear(flattened_size,  out_features: 128)
        self.fc2 = nn.Linear( in_features: 128, num_classes)

    def _forward_conv_layers(self, x):  2 usages  aoco3995
        # Apply all 3 in parallel
        out_7x7 = F.relu(self.conv1_7x7(x))
        out_5x5 = F.relu(self.conv1_5x5(x))
        out_3x3 = F.relu(self.conv1_3x3(x))

        # Concatenate along the channel dimension
        x = torch.cat( tensors: (out_7x7, out_5x5, out_3x3), dim=1)  # Shape: (B, 24, H, W)

        # Continue through rest of CNN
        x = self.pool(F.relu(self.conv2(x)))  # Downsample
        x = self.pool(x)  # Downsample again
        return x

    def forward(self, x):  aoco3995 +1
        x = self._forward_conv_layers(x)
        x = x.view(x.size(0), -1)  # Flatten
        x = F.relu(self.fc1(x))
        return self.fc2(x)
```

```python
# Dataset Class
class UnlabeledDataset(data.Dataset):
    def __init__(self, image_list, DATA_DIR):
        self.image_list = image_list
        self.DATA_DIR = DATA_DIR
        self.transform = transforms.Compose([
            transforms.ToPILImage(),
            transforms.RandomHorizontalFlip(p=0.5),           # Randomly flip horizontally
            transforms.RandomVerticalFlip(p=0.2),             # Randomly flip vertically (less common)
            transforms.RandomResizedCrop(
                size=(IMAGE_SIZE, IMAGE_SIZE),
                scale=(0.8, 1.0),   # Random zoom between 80% and 100%
                ratio=(0.9, 1.1)    # Allow a little squishing/stretching
            ),
            transforms.ColorJitter(
                brightness=0.2,
                contrast=0.2,
                saturation=0.2,
                hue=0.02           # Slight color variations
            ),
            transforms.RandomRotation(degrees=10),             # Small random rotations
            transforms.ToTensor()
        ])

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

    def __getitem__(self, idx):
        img_name = self.image_list[idx]
        img_path = os.path.join(self.DATA_DIR, img_name)
        image = cv2.imread(img_path)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        image = self.transform(image)

        return image, img_name
```

```python
class ModelTester:
    def __init__(self, model, model_path, label_map, threshold=0.5, image_size=100, batch_size=32, device=None):
        self.model = model
        self.model_path = model_path
        self.label_map = LABEL_MAP
        self.rev_label_map = {v: k for k, v in label_map.items()}
        self.threshold = THRESHOLD
        self.image_size = IMAGE_SIZE
        self.batch_size = BATCH_SIZE
        self.device = device or ('cuda' if torch.cuda.is_available() else 'cpu')
        self._load_model()

    def _load_model(self):
        self.model.load_state_dict(torch.load(self.model_path, map_location=self.device))
        self.model.to(self.device)
        self.model.eval()

    def run(self, image_list, image_dir, save_csv=None):
        dataset = UnlabeledDataset(image_list, image_dir)
        dataloader = data.DataLoader(dataset, batch_size=self.batch_size, shuffle=False)

        results = []

        with torch.no_grad():
            for images, img_names in tqdm(dataloader, desc="Running Inference"):
                images = images.to(self.device)
                outputs = torch.sigmoid(self.model(images)).cpu().numpy()

                for i, output in enumerate(outputs):
                    preds = [self.rev_label_map[j] for j, prob in enumerate(output) if prob > self.threshold]
                    if not preds:  # fallback: use the most confident class
                        max_index = np.argmax(output)
                        preds = [self.rev_label_map[max_index]]
                    results.append({"id": img_names[i], "target": ",".join(preds)})

        df_results = pd.DataFrame(results)
        if save_csv:
            df_results.to_csv(save_csv, index=False)
        return df_results
```

```python
def manual_grading(df, num_samples=10, save_corrected_csv='corrected_labels.csv'):  1 usage  Jake
    correct = 0
    results = []

    sampled = df.sample(n=num_samples).reset_index(drop=True)

    for i, row in sampled.iterrows():
        img_path = os.path.join(DATA_DIR, row['id'])
        image = cv2.imread(img_path)
        image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        # Show the image and predicted label
        plt.imshow(image_rgb)
        plt.title(f"[{i + 1}/{num_samples}] Predicted: {row['target']}")
        plt.axis('off')
        plt.show()

        # Prompt user for input
        user_input = input(
            "Enter correct label(s) (comma-separated), or press Enter to accept prediction: ").strip()

        if user_input:
            user_labels = ",".join([label.strip() for label in user_input.split(",")])
        else:
            user_labels = row['target']  # Accept prediction

        predicted_set = set(row['target'].split(","))
        actual_set = set(user_labels.split(","))

        is_correct = predicted_set == actual_set
        if is_correct:
            correct += 1
        else:
            print(f"❌ Mismatch → Model: {predicted_set} | You: {actual_set}")

        results.append({"id": row['id'], "predicted": row['target'], "corrected": user_labels})

    # Save corrected results
    df_corrected = pd.DataFrame(results)
    df_corrected.to_csv(save_corrected_csv, index=False)

    print(f"\n✅ Manual grading complete. Accuracy: {correct}/{num_samples} = {correct / num_samples * 100:.2f}%")
    print(f"📝 Corrected labels saved to: {save_corrected_csv}")
```

```python

    # ------------------------------------------------------------------------------------------------------------------

if __name__ == '__main__':
    model = CNN(num_classes=NUM_CLASSES, image_size=IMAGE_SIZE)
    tester = ModelTester(model=model,
                         model_path='../Code/Saved_logs/Final_model/model_MARS.pt',
                         label_map=LABEL_MAP,
                         threshold=THRESHOLD,
                         image_size=IMAGE_SIZE,
                         batch_size=BATCH_SIZE,
                         device=device)

    df_pseudo_labels = tester.run(
        image_list=UNLABELED_IMAGES,
        image_dir=DATA_DIR,
        save_csv='pseudo_labels_{}.csv'.format(NICKNAME)
    )

    print("✅ Saved pseudo-labels for {} images".format(len(df_pseudo_labels)))

    manual_grading(df_pseudo_labels, num_samples=10, save_corrected_csv='corrected_labels_MARS.csv')
```