Individual Final 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.

Individual Work

My unique contributions to the project of multi-label image classification for defect detection on aircraft fuselages amount to:

1. Initial setup of model architecture and importing json annotations using the labeled subset of our dataset.
2. Comprehensive logging of training metrics and artifacts for traceability.
3. Pseudo-labeling, where the trained model predicts labels for unlabeled data, which are then selectively manually graded by the user.

Adam and I also collaborated closely with each other, implementing new code and improvements throughout the duration of the project, often trying out each other's ideas to improve performance of the model.

Initial Training Setup

The network outputs logits $z = f(x)$, which are passed through a sigmoid activation for each class:

$$\hat{y}_i = \sigma(z_i) = \frac{1}{1+e^{-z_i}} \text{ for each class } i$$

Since it's a multi-label task, I used Binary Cross Entropy (BCE) loss per class:

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

Logging

During training and evaluation:

- Per-epoch metrics (F1 macro/micro, Hamming loss, accuracy, etc.) are computed and saved to metrics_log.json which shows metrics for each epoch.

All logs are stored in a structured timestamped directory:

- f1_score.png kernels.png, first_feature.png, feature_maps.png, loss_curve.png, and confusion matrices for data and performance visualization
- current version of model.pt and python files relevant to training whenever the training code is ran.
- hyperparameters.txt for training hyperparameters

Pseudo-Labeling

In pseudo-labeling, predictions on unlabeled data are used as new training targets if they exceed a confidence threshold:

$$Pseudo - Label: \hat{y}_i = \begin{cases} 1, & if\ \hat{y}_i \geq \tau_i \\ 0, & if\ \hat{y}_i \leq (1 - \tau_i) \end{cases}$$

A threshold of 0.4 was used in the labeling process. If no label met the 0.4 threshold for labeling than the class with the highest confidence value was labeled on the image (most often scratch). To avoid reinforcing errors, class-specific thresholds can be used by enabling manually.

Detailed Description

Groundwork

To establish the foundational infrastructure for our defect classification system, I developed a custom PyTorch Dataset class initially named AircraftFuselageDataset for processing the incoming data from the dataset. This class is designed to load paired image and annotation data, where images are stored in JPEG format and annotations follow a JSON schema containing object labels and corresponding image coordinates. The dataset supports multi-label classification by extracting all unique labels from the annotation files and constructing a binary label vector for each image based on the presence or absence of each class. A label-to-index mapping is built automatically unless a predefined label list is supplied. Images are loaded using the PIL library and converted to RGB format, while annotations are parsed using Python's json module. A basic preprocessing pipeline is defined using torchvision transforms, including resizing to 224×224 pixels and conversion to tensor format. The dataset was verified by loading a sample image-label pair and printing both the tensor shape and corresponding decoded label names, demonstrating the successful extraction and encoding of multi-label targets. Additionally, parameters such as batch size, learning rate, and input image dimensions were defined in preparation for subsequent model training.

After this groundwork was established, Adam and I collaborated in getting the model to perform how we wanted it to. While he mainly worked on improving the model performance and creating the masking techniques, I moved on to focus on the pseudo-label generation process.

Pseudo-Labeling

The bulk of my work was to establish a complete pipeline for applying a trained CNN model to the unlabeled aircraft fuselage images in order to generate pseudo-labels. The system starts by defining key parameters such as image resolution, batch size, class labels, and the file paths to the dataset. The CNN model architecture uses three parallel convolutional layers with 7×7, 5×5, and 3×3 kernels, followed by a shared convolutional block, max pooling, and two fully connected layers. This hybrid convolutional design captures visual patterns at different spatial scales, improving the model's capacity for detecting various surface-level damage types.

To mirror the dataset for the trained model I made a custom UnlabeledDataset class to load and transform unlabeled images with the same aggressive data augmentations, including flipping, cropping, jittering, and rotation, to simulate realistic imaging variations and improve robustness. The pipeline uses a ModelTester wrapper class to load pretrained weights, perform inference, and convert logits into class probabilities using a sigmoid activation function. Multi-label predictions are then generated by thresholding these probabilities at set value (0.4). If no class exceeds the threshold, the class with the highest activation is selected to ensure non-empty predictions. The predicted labels are stored in a structured CSV file, supporting downstream usage for validation via manual grading.

Manul Grading

The manual grading function facilitates human-in-the-loop validation of the model's predictions. It randomly samples a subset of the pseudo-labeled images, displays them alongside

their predicted labels using matplotlib, and prompts the user to verify or correct these predictions through command-line input. For each sample, the user can either confirm the model's output or input corrected labels, which are then logged. The function computes the accuracy of the model against the manual corrections and saves a new CSV file containing both predicted and corrected labels. This enables direct assessment of model reliability and supports iterative improvement through semi-supervised training.

Figure and Log Creation

In the main training file, I incorporated a comprehensive system for logging and visualization. Upon initiation, it generates a dedicated logging directory with a timestamp to store outputs, model artifacts, and diagnostic plots. Within this directory, it systematically records the model architecture (model_architecture.txt), hyperparameter settings (hyperparameters.txt), and the absolute path to the saved model checkpoint (*_path.txt). It also archives all relevant Python scripts into a subdirectory to ensure experiment reproducibility.

As training progresses, the script logs performance metrics such as macro F1 score, accuracy, and loss for each epoch into a structured JSON file (metrics_log.json). Visual outputs include plots of the training loss curve (loss_curve.png) and the F1 score trend across epochs (f1_score.png). For model diagnostics, confusion matrices for each class are computed and saved as individual heatmaps inside a confusion matrices subfolder, enabling granular inspection of classification performance.

Additionally, from the labs, I incorporated scripts that visualize the learned convolutional kernels from each of the three parallel convolutional layers (e.g., kernels_7x7.png) and the corresponding feature maps generated from a sample image (feature_maps_7x7.png), offering

insight into how the model perceives different features during inference. These assets collectively support interpretability, performance monitoring, and reproducibility of the training process.

Results

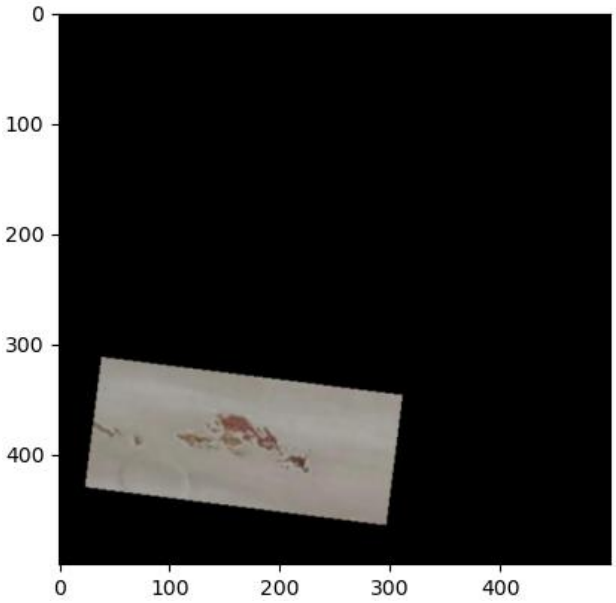
Initial Setup

Output of initial setup before training was implemented (goes through image augmentation):

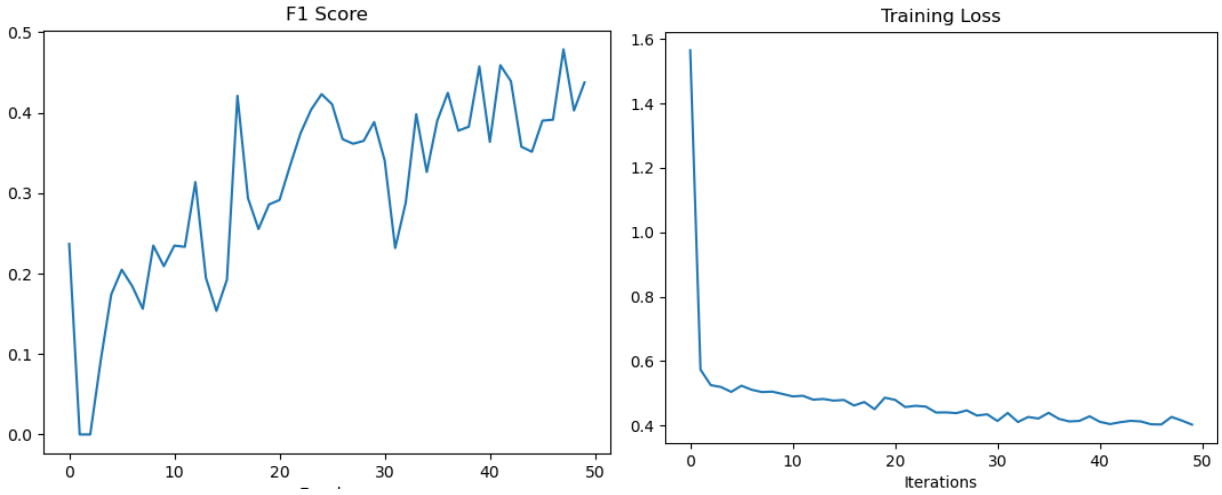
- Image shape: torch.Size([3, 224, 224])
- Multi-label vector: tensor([0., 0., 0., 1.])
- Label names: ['scratch']

Logging

First Feature

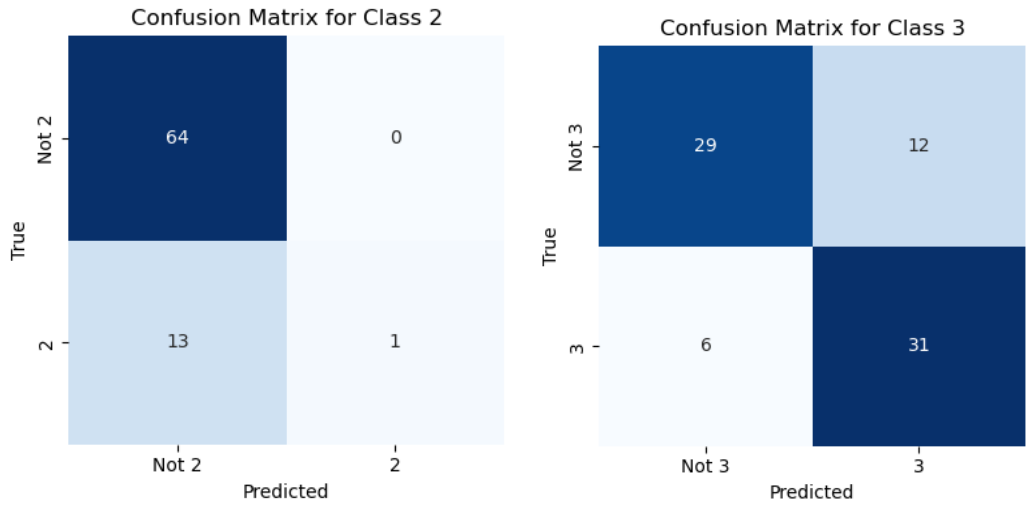


Training Curves



Classification Report & Confusion Matrices

	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



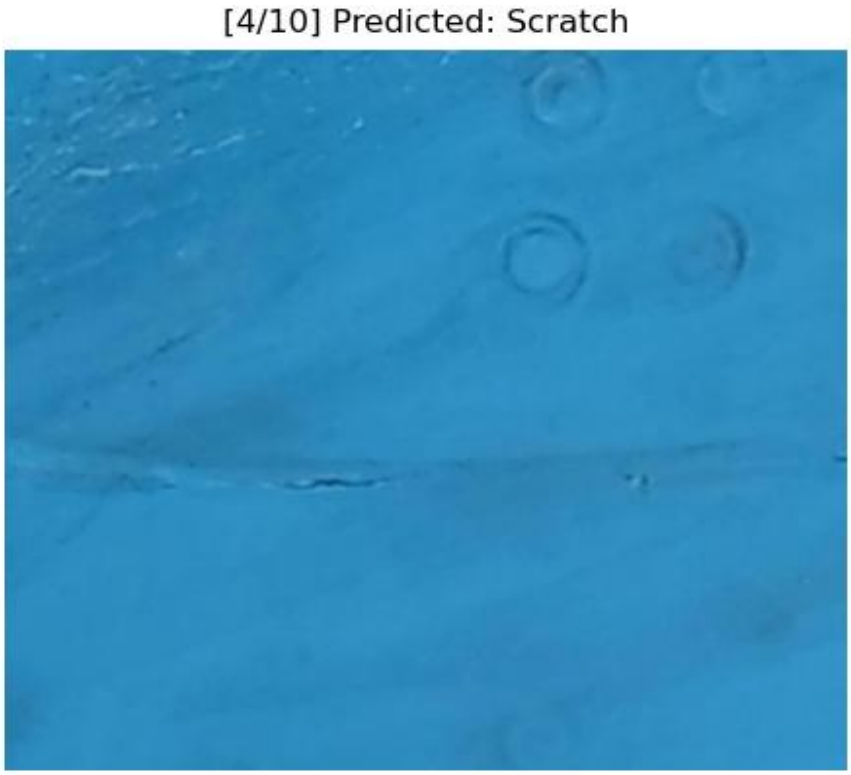
Metrics

This is a sample of one of our later runs taken from the output json file:


```
{
  "epoch": 1,
  "loss": 19.357697069644928,
  "f1_macro": 0.23695652173913043,
  "accuracy": 0.0,
  "hamming_loss": 0.4230769230769231,
  "cohen_kappa": 0.0,
  "matthews_corrcoef": 0.0
},
{
  "epoch": 5,
  "loss": 5.603519558906555,
  "f1_macro": 0.17415730337078653,
  "accuracy": 0.3974358974358974,
  "hamming_loss": 0.22115384615384615,
  "cohen_kappa": 0.18571428571428583,
  "matthews_corrcoef": 0.2037727693371
},
{
  "epoch": 50,
  "loss": 3.316991835832596,
  "f1_macro": 0.43772409695330583,
  "accuracy": 0.5384615384615384,
  "hamming_loss": 0.19230769230769232,
  "cohen_kappa": 0.3275862068965518,
  "matthews_corrcoef": 0.3436320669303881
}
```

Pseudo Labeling

Here is an example of what the user would see when they ran the pseudo-label program and got to the manual grading portion of the function:



The user would then be asked to input all the defects that they see. That input would then be compared to the generated labels of the image to evaluate accuracy.

Summary and Conclusions

My contribution to this project included laying the groundwork for our training model by importing annotations through the json file and setting up the initial model architecture. I also implemented a robust logging system so that each run is saved to a timestamped directory, which

includes model weights, architecture, logs, hyperparameters, and the full source code. This ensures reproducibility and clear tracking of all experiments. To expand further, during training, the model's performance was evaluated using F1 macro, F1 micro, Hamming loss, accuracy, Cohen’s kappa, and Matthew’s correlation coefficient. Per-epoch results were logged and visualized, providing a clear view of the model’s progress and convergence. The system also output a wide range of visual diagnostics, including training curves, confusion matrices, and classification reports. These aid in understanding the internal behavior of the model and increase trust in its predictions.

I also implemented pseudo-labeling to leverage a larger pool of unlabeled images. After the initial training phase on labeled data, the model was used to generate predicted labels for unlabeled samples. These predictions were filtered using per-class confidence thresholds. In addition, I developed a manual grading mode to assist in qualitative analysis. Selected predictions were displayed with both predicted and ground truth labels, allowing for human verification. This was particularly useful for refining threshold settings and identifying systematic errors.

Overall, I consider my contributions to this project to add significant value and help in paving a way for future development. I want to particularly the pseudo-label system which not only greatly expands the labeled dataset but also transforms it into a true multi-label dataset.

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