Wafer Defect Classification Using Convolutional Neural Networks, Autoencoder-Based Augmentation, and Transfer Learning

Janane Sivakumar
Department of Electrical Engineering
San José State University
San José, CA, USA
Email: janane.sivakumar@sjsu.edu

***Abstract*—Automated wafer defect inspection is essential for maintaining yield and reliability in semiconductor manufacturing. Traditional methods often fail because they cannot handle variability in defect patterns, low image resolution, or the need for extensive manual tuning. Deep learning offers a promising alternative, but its performance is limited by class imbalance and the lack of certain defect types. This study evaluates three convolutional neural network (CNN) approaches for wafer map defect classification using a highly imbalanced dataset of 14,366 images across nine defect categories. The models examined include: (1) a baseline CNN, (2) an augmented CNN that incorporates oversampling, class weighting, and autoencoder-based synthetic sample generation, and (3) a transfer learning model based on MobileNetV2. The baseline CNN achieved high overall accuracy due to the dominance of the "none" class but failed to detect minority defects. The augmented CNN improved minority-class recall and demonstrated successful detection of several rare defect types. These improvements indicate that balancing the dataset noticeably improves the model's classification performance. In contrast, the transfer learning model did not improve minority-class prediction. Its pretrained feature extractors were not well suited for the subtle geometric patterns found in low-resolution wafer maps. Overall, smaller CNNs with autoencoder augmentation perform better and are more efficient than large pretrained models for this application. These findings lay the foundation for more efficient wafer inspection systems. It also motivates future work using higher-resolution data, more advanced generative augmentation, and application-specific-aligned pretraining.**

***Index Terms*—Wafer defect classification, class imbalance, Auto-encoder-based augmentation, Oversampling, Convolutional Neural Networks, Transfer Learning**

I. Introduction

Automated wafer defect inspection is an essential component of semiconductor manufacturing, enabling early detection of yield-reducing defects and reducing reliance on manual visual inspection. Traditional rule-based image processing approaches struggle with variability in defect patterns and often require extensive parameter tuning. In contrast, deep learning-based classifiers have demonstrated strong generalization performance across diverse wafer map geometries, but their effectiveness is often limited by small dataset size, severe class imbalance, and low spatial resolution.

However, numerous challenges exist in the development of wafer classification CNN models. First of which is the severe class imbalance of the training data set, due to some of the defect types being more rare than other types. This introduces biases into the model, preventing it from accurately classifying those types of defects. In general, most wafer datasets are skewed towards the "none" class, as there is more data for this class. This results in the model classifying the wafers with this class more than others, even when defects are present. This issue is especially prominent in traditional CNNs since they struggle to generalize with the minority classes. This results in classifications that are difficult to interpret. The small wafer map resolution also affects the model classification. Its generally low resolution results in limited detail visibility. When trained on this model, the model is unable to clearly detect these features on the wafer, especially if they are small or subtle.

The purpose of this work is to explore improved classification methods for wafer defects using Convolutional Neural Networks (CNNs). In order to evaluate the effectiveness of the classification method, three CNN models were tested: a baseline CNN model without any classification modifications, augmented CNN model integrated with oversampling, class weighting, and autoencoding, and a transfer learning based CNN model. Three models were used to visualize the effects of the augmentation on the model efficiency. The standard CNN model was used to provide a baseline for efficiency comparison. The augmented model implements oversampling of minority classes and class weighting to reduce the bias towards classes with smaller sample sizes. An autoencoder was incorporated in addition to these techniques. A paper on the "Wafer Map Defect Classification Using Autoencoder-Based Data Augmentation and Convolutional Neural Network" reveals that the implementation of an autoencoder based data augmentation was found to effectively address the issue of data imbalance and increase class diversity. When tested on the WM-811K dataset, the augmented CNN was able to achieve 98.56% classification accuracy [1]. Hence, as a result of the effectiveness of this approach, this method was implemented into the current CNN classification. A transfer learning approach was also developed to evaluate the effect of model depth on the performance. Generally, it is known

that the addition of layers can increase the ability of the model to extract hierarchical features of the image and increase accuracy. By testing on the performance on a transfer learning model, it can be determined whether or not model depth can increase the accuracy of detection. For this test, MobileNet was used. Through this analysis, a practical, universal framework can be developed for industrial wafer inspection systems.

All model development, dataset preprocessing, CNN architecture implementation, autoencoder-based augmentation, transfer learning framework construction, performance evaluation, and result analysis were conducted by the student. Python code for all models, including training scripts, visualization utilities, and evaluation functions, was written, tested, and optimized independently by the student using Google Colab and CPU-based hardware. All plots, confusion matrices, and classification reports presented in this paper were generated directly from the implemented models.

II. Methodology & Data Preprocessing

A. Wafer Dataset:

The CNN models used in this study were tested using a wafers dataset consisting of 14,366 images, with 9 different defect classes. The classes included in the dataset are as follows: None, Edge-Loc, Loc, Center, Scratch, Random, Near Full, Edge-Ring, Donut. The None class images are images of wafers without any defects. The Edge-Loc class contains images with edge related defects, and the loc class contains wafer images with localized defects. The scratch class contains images of wafers with scratches. The Random and near full classes consist of images with randomly scattered defects and that are near full, without clear defects. The edge ring class contains wafer images with ring shaped anomalies and the donut class contains wafer images with a "donut" shaped defect at the center. Table I provides a summary of the number of images in each class within the test dataset.

All experiments in this study were performed using Python 3 within the Google Colab environment. The models were trained and evaluated using CPU-based hardware, which provided a consistent and accessible platform for running deep learning workflows without the need for specialized GPU acceleration. Google Colab also facilitated seamless integration with Google Drive for dataset management and allowed for efficient execution of all preprocessing, training, and analysis scripts.

A histogram of the samples per class in the given dataset was also generated to visualize the differences in the amount of data across different classes. The resulting histogram is presented in Figure 1.

Both the data in Table I and Figure 1 reveal that the dataset is highly imbalanced across classes. A majority of the samples belong to the "none" class. Aside from this category, the "Edge-loc" and "Loc" classes make up a sizable portion of the remaining data. In contrast, the rarer defect types contain very few samples. The "Donut" class, in particular, has the least representation across all categories, containing only a single sample.

TABLE I: Summary of Number of Images per Class for Wafer Dataset

Class Name	Number of Images per Class
None	13489
Edge-Loc	296
Loc	297
Center	90
Scratch	72
Random	74
Near-full	16
Edge-Ring	31
Donut	1

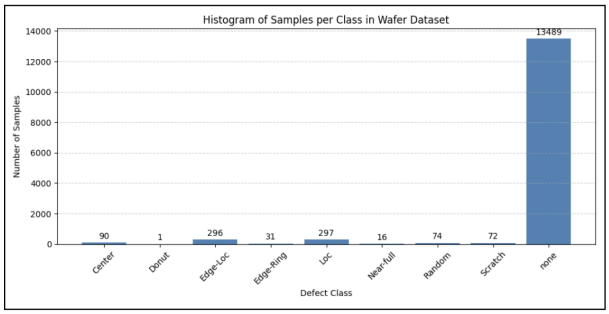


Fig. 1: Histogram of the Number of Samples per Defect Class

All images in the dataset are 26 x 26 pixels in size and in RGB format. Each image has a corresponding classification label associated with the image. All images present in the dataset are in binary form to simplify the data, highlight critical features, reduce noise, and decrease computational cost. The binarized images in the present dataset are presented in Figure 2.

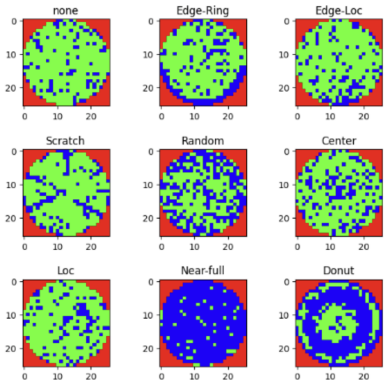


Fig. 2: Defect Wafer Images

B. Baseline CNN Model:

To build the baseline convolutional neural network (CNN) for wafer defect classification, the dataset was first loaded from a pickle file and organized into a Pandas DataFrame containing both wafer-map images and their corresponding labels. The image arrays were extracted and stacked into a single tensor. Each sample was reshaped into a ($26 \times 26 \times 1$) format to

maintain a uniform input structure. All pixel intensities were normalized to the [0,1] range to improve stability and prevent large gradient updates during training. The defect labels were cleaned and converted into integer class IDs using NumPy's label-encoding functions. This helped ensure that all nine defect categories were represented consistently.

Once preprocessing was complete, the dataset was split into training, validation, and testing subsets using randomized sampling. A 70%-15%-15% split was applied. The training set was randomized to avoid order-related bias, and each split was converted into a TensorFlow dataset. A batch size of 128 was used to speed up training and enhance stability Table II summarizes the split of the dataset for this model.

TABLE II: Split Dataset for Baseline CNN

Defect	Training Set	Validation Set	Test Set
Center	68	14	8
Donut	1	0	0
Edge-Loc	210	41	45
Edge-Ring	22	4	5
Loc	224	39	34
Near-full	14	1	1
Random	56	14	4
Scratch	50	12	10
none	9411	2030	2048

The baseline CNN architecture was intentionally kept compact to match the small spatial resolution of the wafer images. Figure 3 presents a block diagram of the CNN model used in the baseline model. The network consisted of three convolutional blocks. Each block included a 3×3 convolutional layer followed by a ReLU activation, batch normalization, and a max-pooling operation. The filters in these layers increased progressively from 8 to 16 and then 32. L2 kernel regularization with a coefficient of 5×10^{-4} was applied to all convolutional layers to reduce overfitting and control weight values. Max-pooling layers reduced the spatial dimensions after each feature extraction step, enabling the model to focus on the geometric structure of defects such as clusters, scratches, and donut patterns.

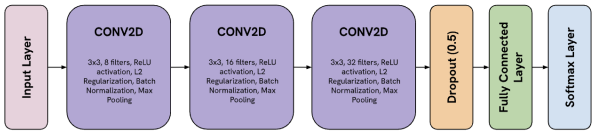


Fig. 3: Block Diagram of the Baseline CNN Model

After the convolutional blocks, the network applied a dropout layer with a rate of 0.5 to prevent the overfitting of the small dataset. The feature maps were then flattened and passed into a 64-unit dense layer with ReLU activation to prevent vanishing gradients and accelerate training. A second dropout layer with the same rate was used before the final classification stage. The output layer was a softmax classifier that produced probability scores for each defect class.

The model was compiled using stochastic gradient descent (SGD) with a learning rate of 1×10^{-4} and a momentum value of 0.9. These values were selected to ensure steady, stable convergence. This is especially important due to the high degree of class imbalance in the dataset. The model was trained for up to 30 epochs with early stopping enabled. A patience value of five epochs was used, which stopped training automatically when the validation loss failed to improve for five consecutive epochs. Throughout the training process, accuracy and loss values were recorded for both the training and validation sets. These results were later used to evaluate convergence behavior, identify possible overfitting, and analyze the overall performance of the baseline model.

C. Augmented CNN Model:

To develop the augmented CNN, the wafer dataset was first loaded from a pickle file and converted into a uniform tensor format. Each image was reshaped into a 26×26×1 structure, and the pixel values were normalized to the [0,1] range to maintain numerical stability during training. The labels were cleaned to remove list wrappers and were then mapped to integer class IDs. After preprocessing the dataset, a randomized 70%-15%-15% split was applied to create the training, validation, and test sets. These subsets were generated using a fixed random seed to ensure reproducibility. Once the splits were created, the training data was prepared for augmentation using a size-preserving autoencoder. Table III summarizes the split of the dataset for the augmented model.

TABLE III: Split Dataset for Augmented CNN Model

Defect	Training Set	Validation Set	Test Set
Center	272	14	8
Donut	4	0	0
Edge-Loc	210	40	45
Edge-Ring	88	4	5
Loc	224	39	34
Near-full	56	1	1
Random	224	14	4
Scratch	200	12	10
none	9411	40	2048

The autoencoder was designed to generate synthetic wafer images while maintaining the original spatial dimensions. Its encoder consisted of three convolutional layers with 32, 16, and 8 filters, respectively, each using 3×3 kernels and ReLU activation. These layers compressed the input into a lower-channel representation but preserved spatial resolution. The decoder mirrored the encoder structure and reconstructed the images using convolutional layers with 16 and 32 filters, followed by a final output layer with sigmoid activation. The autoencoder was trained for 20 epochs using mean-squared error loss. Once trained, it was used to reconstruct images from rare defect classes.

Rare classes were identified by counting the number of samples in each defect category within the training set. Any class with fewer than 100 samples was flagged as rare. For each rare class, the original images were repeated three times and passed through the trained autoencoder to produce synthetic reconstructions. These reconstructed samples were assigned

the same labels as their original inputs. The augmented datasets were gathered into new arrays that combined the original training data with the autoencoder-generated images.

To support fair validation, a balanced validation subset was constructed by selecting up to 40 samples per class. This approach ensured that validation performance was not biased toward majority classes. The augmented training set, balanced validation set, and original test set were then converted into TensorFlow Dataset frameworks, each using a batch size of 128 and preloading to enhance efficiency. After augmentation, class weights were computed to correct remaining label imbalances. These weights were calculated using scikit-learn's `compute_class_weight` function and applied during training.

The classifier architecture for the augmented CNN matched the structure used in the baseline model. The CNN architecture of this model is presented in Figure 4. It consisted of three convolutional blocks with 8, 16, and 32 filters, each using 3×3 kernels, ReLU activation, batch normalization, and max pooling. L2 regularization with a coefficient of 5×10^{-4} was applied to all convolutional layers. After feature extraction, a 0.5 dropout layer preceded a flattening layer and a 64-unit dense layer with ReLU activation. A second dropout layer was added before the final softmax classifier. The model was compiled using stochastic gradient descent (SGD) with a learning rate of 1×10^{-4} and a momentum value of 0.9, matching the settings used in the baseline model.

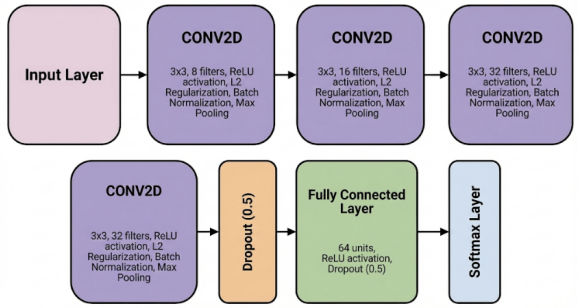


Fig. 4: Block Diagram of the Augmented CNN Model

Training was performed for up to 30 epochs with early stopping enabled. A patience value of five epochs allowed training to stop automatically when the validation loss failed to improve. Class weights were applied during optimization to further compensate for label imbalance. Throughout training, accuracy and loss values for both the training set and balanced validation set were recorded. After training, the model was evaluated on the original, imbalanced test set using accuracy, classification results, a confusion matrix, and per-class accuracy calculations.

D. Transfer learning-based CNN Model:

To develop the transfer learning model, the wafer dataset was first loaded from a pickle file and converted into a consistent tensor format. Each image was reshaped into a 26 × 26 × 1 structure, and all pixel intensities were normalized to the [0,1] range. The defect labels were cleaned to remove extra list brackets, and mapped to integer class IDs. After preprocessing, the dataset was split into training, validation, and test subsets using a 70%-15%-15% split with a fixed random seed to ensure reproducibility. Since MobileNetV2 expects three-channel RGB images of at least 96 × 96 resolution, a preprocessing process was developed. This process expanded the grayscale images into three channels, and resized them to 96 × 96. Then, it applied MobileNetV2's standardized preprocessing function, which scales pixel values to the range expected by the pretrained ImageNet feature extractor.

In order to evaluate the effect of the addition of additional layers to the CNN model performance, a transfer learning based CNN model was developed. This was developed using MobileNet. Figure 5 provides the block diagram for the model's processing framework.

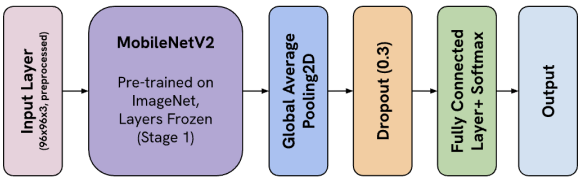


Fig. 5: Block Diagram of the Transfer Learning based CNN Model

MobileNetV2 was defined with pretrained ImageNet weights to leverage features, learned from large-scale natural image datasets. In the first training stage, all layers of the base model were frozen to preserve the pretrained feature representations. A custom classification head was added to adapt the model to wafer defect classification. The head consisted of a global average pooling layer that condensed the spatial feature maps into a single vector per image. A dropout layer with a rate of 0.3 for additional regularization, and a dense softmax layer that produced probability distributions across the defect classes, were also added. The model was compiled using the Adam optimizer with a learning rate of 1×10^{-3}. Training during this initial stage was limited to the newly added classification head, and the model was trained for up to 20 epochs with early stopping based on validation loss to prevent unnecessary overfitting.

Once the head was trained, the model entered a second stage dedicated to fine-tuning. During this stage, the MobileNetV2 base was partially unfrozen, beginning at layer index 100. This allowed the deeper, higher-level convolutional blocks to adapt to the unique geometric patterns present in wafer defect maps. It also enabled the early layers to remain fixed to preserve low-level feature extraction. The learning rate was reduced to 1×10^{-5} to ensure stable and incremental adjustments to the pretrained weights. The model was recompiled with these updated settings and trained for another 20 epochs with early stopping enabled. Throughout both training stages, the model tracked accuracy and loss for the training and validation

datasets to monitor convergence behavior and identify signs of under or overfitting.

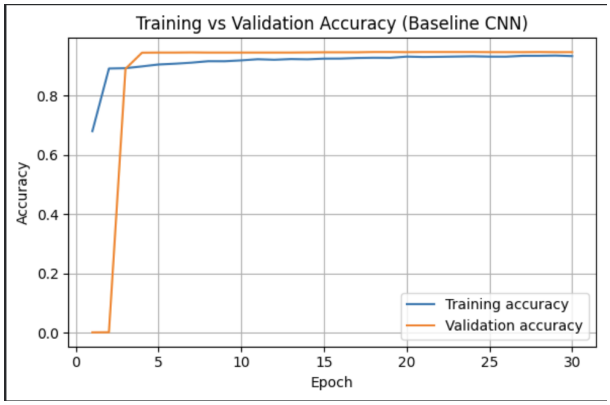
After training, the complete model was evaluated on the full test set. The evaluation included overall accuracy, a classification report summarizing precision, recall, and F1-scores for each defect class, and a confusion matrix to visualize misclassification patterns. These results provided a complete assessment of how effectively the transfer learning approach captured subtle wafer defect structures compared to non-pretrained CNN models.

III. Results & Discussion:

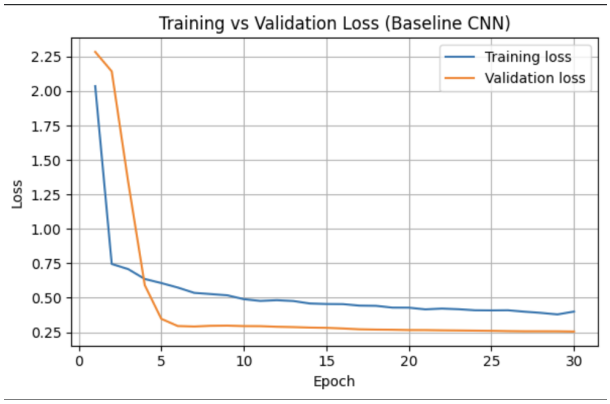
This study evaluated multiple deep learning strategies for addressing class imbalance in wafer map defect classification. Approaches evaluated in the study include a baseline convolutional neural network (CNN), a transfer learning approach, and an autoencoder-based augmentation method. Across all experiments, models were trained on the highly imbalanced wafer defect dataset, where the "none" class constitutes >90% of all samples. Some samples in the dataset contained several defect types, fewer than 10 examples. This imbalance strongly influenced prediction behavior and was a major determinant of model performance.

A. Baseline CNN Results:

The baseline CNN showed a rapid improvement in both training and validation accuracy during the initial epochs. This indicates that the model was able to quickly learn distinguishable features from the wafer defect dataset. The accuracy and loss plots of the baseline CNN are presented in Figure 6.



(a) Training vs. Validation Accuracy (Baseline CNN)



(b) Training vs. Validation Loss (Baseline CNN)

Fig. 6: Accuracy and loss plots of the baseline CNN.

In the early phase of training, between Epochs 1 and 3, both training and validation accuracy increased very quickly. Validation accuracy rose from near zero to above 0.90, and validation loss dropped sharply from about 2.25 to below 0.3. These rapid changes indicate that the model quickly learned broad and easily separable features in the dataset. This is expected given the dominance of the "none" class, which allowed the model to achieve high accuracy early on.

During the middle phase of training, roughly between Epochs 4 and 10, the improvements became slower. The training and validation accuracy curves continued to rise, but at a more gradual rate, eventually reaching the 0.92–0.94 range. Training and validation loss also continued to decline more slowly. The close alignment of the curves in this period suggests that the model was still generalizing well. Dropout and L2 regularization likely helped maintain this stability.

In the later phase of training, from Epoch 10 onward, the curves began to level off. Training accuracy approached its maximum, and validation accuracy remained steady with only minor fluctuations. Training loss continued to decrease at a slow rate, while validation loss stayed low and stable. The lack of separation between the curves shows that the model did not overfit, even with extended training.

Overall, the curves indicate that the model converged efficiently. The rapid early rise and early plateau suggest that the model captured the main patterns in the data very quickly. However, this plateau also highlights the limiting effect of the strong class imbalance, which restricted further improvements. Most of the accuracy improvements resulted from correctly predicting the majority class. Minority defect types had little influence on the overall results.

These performance trends are reflected in the classification report for the full imbalanced test set (Figure 7). Out of the nine classes, the model correctly identified only the majority class (none), achieving a precision of 0.94, recall of 1.00, and an F-1 Score of 0.97.

All remaining defect classes—Center, Edge-Loc, Edge-Ring, Loc, Near-full, Random, and Scratch—received precision, recall, and F1-scores of 0.00. This outcome confirms that the model never predicted these classes. The macro-average F1-score was 0.19. In contrast, the weighted-average F1-score was 0.91. This is likely heavily inflated by the overwhelming proportion of none samples (2021 out of 2155 test images).

The confusion matrix, presented in Figure 8, further highlights this imbalance-related failure. Every defect-class sample was misclassified as none, while all none samples were correctly predicted. Several minority classes contained only 2–8 test samples, which made them particularly difficult for the unbalanced model to learn.

```
Classification Report (FULL test set, Baseline CNN):
              precision    recall  f1-score   support

      Center       0.00      0.00      0.00        15
    Edge-Loc       0.00      0.00      0.00        42
   Edge-Ring       0.00      0.00      0.00         8
         Loc       0.00      0.00      0.00        42
   Near-full       0.00      0.00      0.00         2
      Random       0.40      0.75      0.52         8
     Scratch       0.00      0.00      0.00        17
        none       0.94      1.00      0.97      2021

    accuracy                           0.94      2155
   macro avg       0.17      0.22      0.19      2155
weighted avg       0.89      0.94      0.91      2155
```

Fig. 7: Classification Report of the Baseline CNN

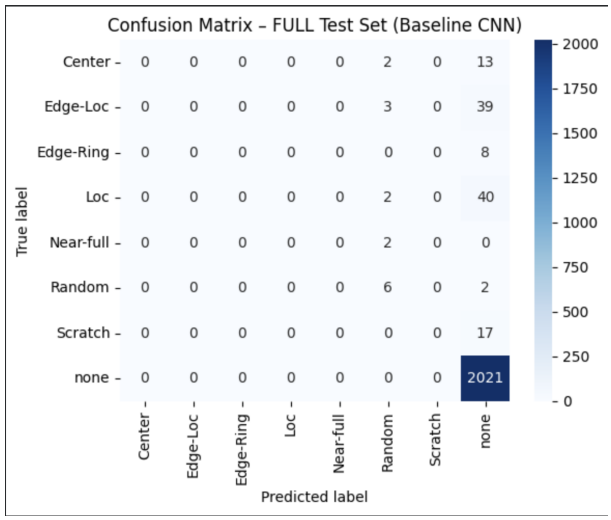


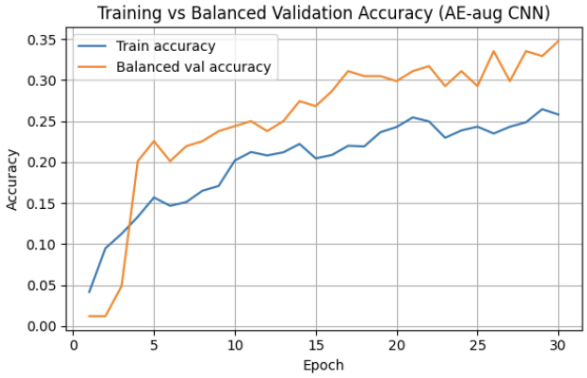
Fig. 8: Confusion Matrix of the Baseline CNN

Overall, these findings demonstrate that a standard CNN trained naively on this dataset will converge toward a degenerate classifier that predicts only the majority class. Although the model appears to achieve a high overall accuracy (94%), this metric is misleading and obscures its complete inability to detect any defect category. These results show the importance of balancing the data through oversampling, class weights, or autoencoder augmentation.

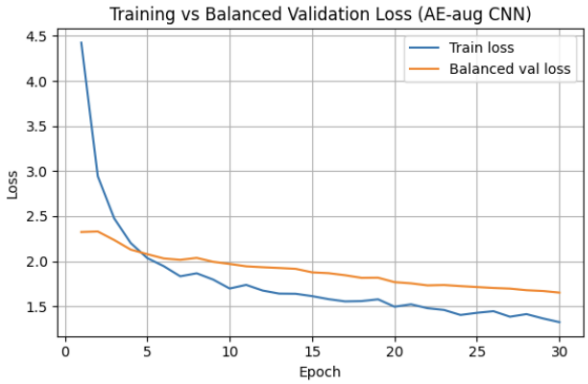
B. Augmented CNN Results:

The augmented CNN showed a different learning pattern than the baseline model because it was trained with oversampled minority classes, autoencoder-generated samples, and a class-balanced validation set. This is demonstrated by the accuracy and loss behavior of the model, as presented in Figure 9.

In the early training phase (Epochs 1–5), both training and balanced validation accuracy increased steadily. Validation accuracy rose more sharply than training accuracy and reached about 0.20 by Epoch 5. This shows the model learned faster because it had more rare defect examples to train on. Unlike the baseline CNN, which first learned only the 'none' class, the augmented model saw more diverse data right away.



(a) Training vs. validation accuracy (Augmented CNN).



(b) Training vs. validation loss (Augmented CNN).

Fig. 9: Accuracy and loss plots for the augmented CNN model.

During the middle phase of training (Epochs 5–15), the improvements became less consistent. Training accuracy increased gradually toward the 0.20–0.25 range, while balanced validation accuracy fluctuated before settling into an upward trend. These fluctuations reflect the increased difficulty of learning many low-frequency defect types. The validation set was intentionally balanced, so performance in this phase shows how well the model generalized across all classes rather than being driven by the majority class. The steady decline in both training and validation loss during this period indicates that the model continued to learn useful features.

In the later phase of training (Epochs 15–30), both accuracy curves showed slow but consistent improvement. Balanced validation accuracy reached roughly 0.35 by the final epoch, while training accuracy remained slightly lower. The small gap between the curves suggests that the model generalized reasonably well on the balanced validation set, even though the augmented training set contained noisy autoencoder reconstructions. The loss curves support this observation. Training loss decreased throughout training, while validation loss declined more gradually and remained higher overall. The stable behavior of both curves indicates that the model did not overfit.

Overall, the augmented CNN showed slower convergence and lower accuracy than the baseline model, but these results reflect the increased complexity of the task. The balanced

validation set required the model to perform well across all defect classes, not just the majority class. The upward trend in validation accuracy across epochs demonstrates that the augmentation strategy improved the model's ability to detect minority defects, which was the main objective of this approach.

Performance on the original, imbalanced test set also demonstrated a clear improvement in minority-class sensitivity. The classification report shown in Figure 10 highlights that the augmented model successfully detected several rare defect classes that the baseline CNN failed to recognize entirely. For instance, recall reached 80% for Edge-Ring, 75% for Random, and 50% for Scratch. Precision remained low for many defect categories due to the overwhelming dominance of the none class, but the model's ability to detect minority classes marked a substantial improvement over the baseline.

Classification Report (original test set):

	precision	recall	f1-score	support
Center	0.02	1.00	0.04	8
Edge-Loc	1.00	0.02	0.04	45
Edge-Ring	0.06	0.80	0.10	5
Loc	0.21	0.15	0.17	34
Near-full	1.00	1.00	1.00	1
Random	0.27	0.75	0.40	4
Scratch	0.01	0.50	0.02	10
none	0.98	0.50	0.66	2048
micro avg	0.49	0.49	0.49	2155
macro avg	0.44	0.59	0.30	2155
weighted avg	0.96	0.49	0.64	2155

Fig. 10: Classification Report for the Augmented CNN Model

The confusion matrix in Figure 11 further illustrates this shift. Instead of mapping almost all samples to the none category (as observed in the baseline CNN), the augmented model distributed predictions across multiple defect classes. Although many samples were misclassified as defects, this reflects the intended trade-off: improved defect sensitivity at the cost of reduced majority-class precision.

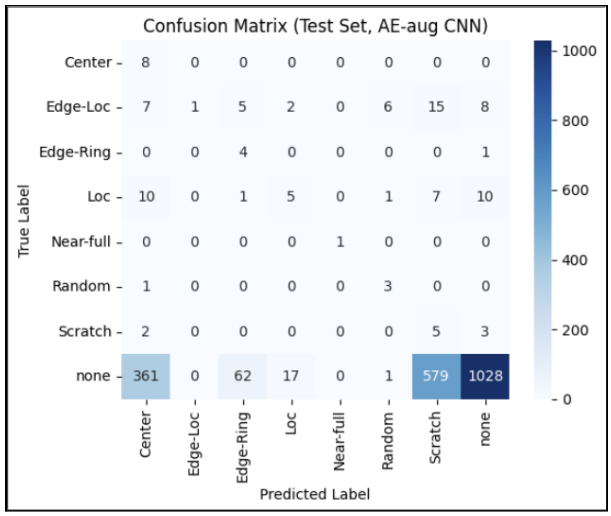


Fig. 11: Confusion Matrix of the Augmented CNN Model

The per-class accuracy results, presented in Figure 12, show that the augmented CNN achieved meaningful improvements for several minority defects. However, the overall performance was not the same across defect classes. The model reached 100% accuracy for *Center* and *Near-full* wafers and performed well on *Edge-Ring* (80%), *Random* (75%), and *Scratch* (50%). This indicates that augmentation helped the model learn distinctive patterns for these categories. In contrast, accuracy remained low for *Loc* (14.71%) and *Edge-Loc* (2.22%). This suggests that these defects remained difficult to distinguish despite additional synthetic samples. The "none" class achieved 50.20% accuracy, which indicates the distribution of predictions was more balanced rather than majority-class dependent. Overall, the model showed improved minority-class recognition, but several defect types still require more robust augmentation or additional real data.

```
Per-class accuracy on test set:
Center: 100.00% (n=8)
Donut: N/A (no samples in test set)
Edge-Loc: 2.22% (n=45)
Edge-Ring: 80.00% (n=5)
Loc: 14.71% (n=34)
Near-full: 100.00% (n=1)
Random: 75.00% (n=4)
Scratch: 50.00% (n=10)
none: 50.20% (n=2048)
```

Fig. 12: Per-class accuracy results for the augmented CNN model.

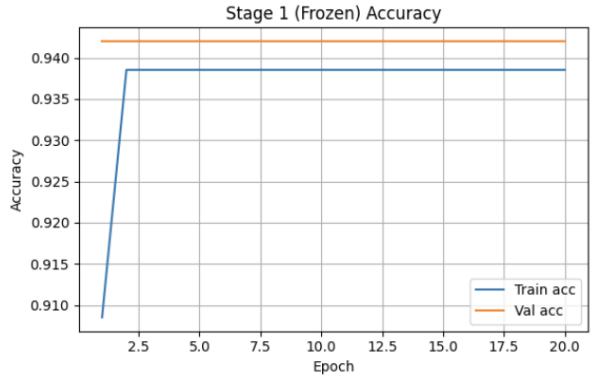
In all, the augmented CNN no longer collapses to predicting only the majority class. Instead, it demonstrates an improved recognition capability across a wider range of wafer defect patterns. Autoencoder-based augmentation and balanced validation sampling significantly improved minority-class detection, even though they reduced accuracy on defect-free samples. In manufacturing contexts where missing a defect is more costly than flagging a normal wafer, this trade-off is often desirable and justified.

C. Transfer Learning Results:

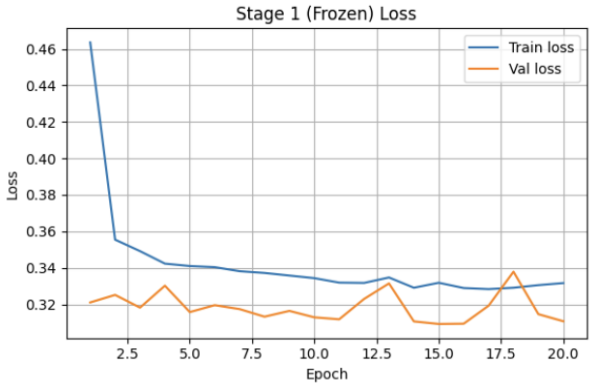
After assessing the baseline CNN and the augmented CNN, the transfer learning model was tested to determine whether pretrained feature extractors could further improve defect classification performance. To do so, MobileNetV2 was implemented as a two-stage transfer learning model.

The transfer learning model used a pretrained convolutional backbone followed by fine-tuning on the wafer map dataset. This approach was expected to improve feature extraction for rare defect types by leveraging filters learned from large-scale natural image datasets. The training behavior of the transfer learning model is shown in Figures 13 and 14.

In Stage 1, where the MobileNetV2 backbone remained frozen, the model reached a high level of accuracy almost immediately. Training accuracy increased from approximately 0.91 to 0.94 within the first two epochs and then remained stable. Validation accuracy showed a similar pattern and quickly plateaued just above 0.94. This rapid convergence

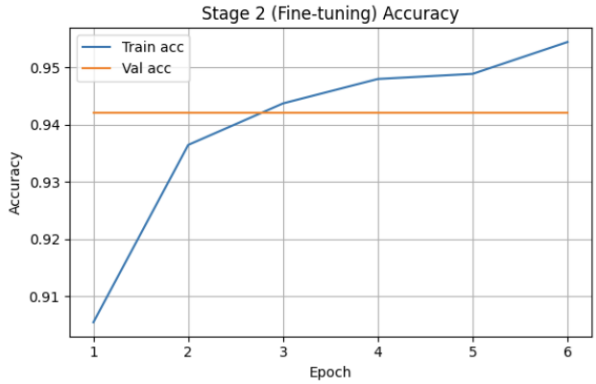


(a) Stage 1 (frozen) accuracy

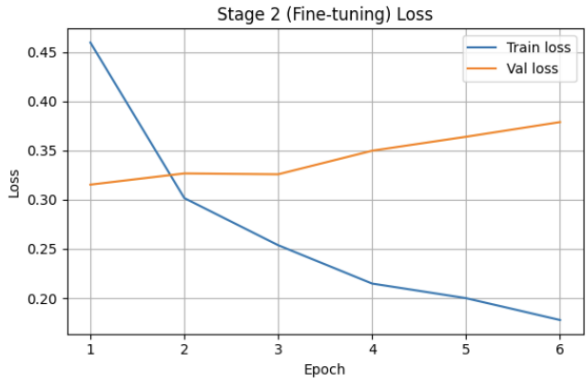


(b) Stage 1 (frozen) loss

Fig. 13: Training and validation accuracy and loss for Stage 1 (frozen MobileNetV2 backbone).



(a) Stage 2 (fine-tuning) accuracy



(b) Stage 2 (fine-tuning) loss

Fig. 14: Training and validation accuracy and loss for Stage 2 (fine-tuning MobileNetV2).

indicates that the pretrained ImageNet feature extractor was already well suited to capturing the coarse structural patterns in the wafer images, even though the images differ significantly from natural-image data. The loss curves also support this observation. Training loss decreased sharply during the first few epochs and then stabilized around 0.33. Validation loss fluctuated slightly across epochs but stayed within a narrow range. These small oscillations are typical when only the classification head is trainable, since the model cannot refine deeper feature representations to better match the target dataset. Overall, Stage 1 reflects fast stabilization with little room for improvement due to the frozen backbone.

In Stage 2 (Figure 14), the top portion of the MobileNetV2 backbone was unfrozen for fine-tuning Training accuracy increased steadily during this stage, rising from approximately 0.91 to above 0.95 by the final epoch. Validation accuracy, however, remained almost flat throughout fine-tuning and stayed near 0.94. This gap between the curves suggests that the fine-tuning process improved the model's fit to the training set independently of the validation set. The loss curves show a similar pattern. Training loss decreased consistently across all epochs, indicating that the unfrozen layers adapted to the training data. In contrast, validation loss increased slightly during later epochs. This trend suggests that the model began to over-specialize to the training distribution once deeper layers were allowed to update, and the refined features did not generalize as effectively to unseen data.

Overall, the two-stage behavior of the MobileNetV2 model shows that most of the benefit came from the pretrained feature extractor, not from fine-tuning. Stage 1 provided strong performance immediately, and Stage 2 offered limited improvement but introduced mild overfitting. These results reveal that although transfer learning can provide a strong starting point for wafer defect classification, the pretrained features may not align closely enough with fine-grained wafer defect patterns to improve classification accuracy.

Test-set performance is summarized in the classification results provided in Figure 15. Despite the improved training behavior and stable validation performance, the transfer learning model struggled to identify rare defect classes in the test set. Like the baseline CNN, it classified nearly all defect samples as "none", resulting in precision, recall, and F1-scores of 0.00 for most defect categories. The "none" class maintained extremely high recall. This indicates the dominance of this class in the training data. Although the pretrained base of the model captured general visual structures, it was unable to learn the subtle patterns associated with

the minority defect classes. This is likely due to the limited number of examples and extreme imbalance in the dataset.

```
Classification Report (FULL test set):
              precision    recall  f1-score   support

      Center       0.00      0.00      0.00        15
    Edge-Loc       0.00      0.00      0.00        42
   Edge-Ring       0.00      0.00      0.00         8
         Loc       0.00      0.00      0.00        42
   Near-full       0.00      0.00      0.00         2
      Random       0.00      0.00      0.00         8
     Scratch       0.00      0.00      0.00        17
        none       0.94      1.00      0.97      2021

    accuracy                           0.94      2155
   macro avg       0.12      0.12      0.12      2155
weighted avg       0.88      0.94      0.91      2155
```

Fig. 15: Classification Report of the Transfer Learning Model

The confusion matrix in Figure 16 further illustrates this effect. Nearly all minority-class samples resulted from the none category, which is similar to the behavior seen in the baseline CNN. This suggests that the limitations in achieving minority-class performance depends on the dataset rather than the model architecture and efficiency.The number of minority defect samples was too small for the model, even with transfer learning, to form meaningful representations of their features. Since transfer learning relies on fine-tuning high-level pretrained features, it remains limited when the target dataset lacks sample diversity, contains very few examples of each minority class, or exhibits severe class imbalance.

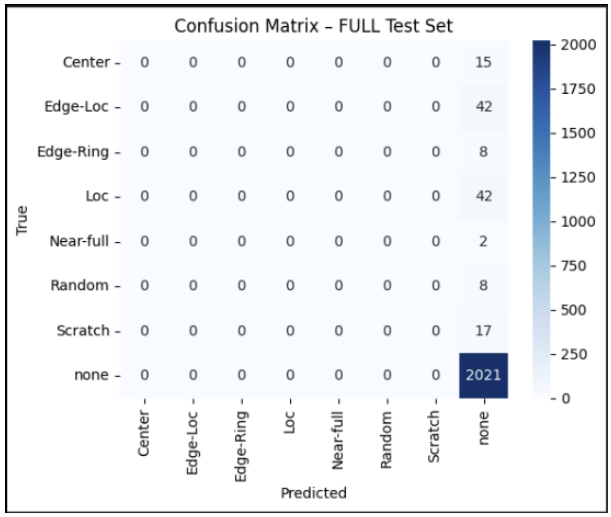


Fig. 16: Confusion Matrix of the Transfer Learning Model

Overall, these results indicate that transfer learning is insufficient to overcome the inherent data limitations. Although the model optimized and maintained validation performance, it did not improve the detection of rare defects. Compared with the augmented CNN, which demonstrated substantial improvements in minority-class recall, the transfer learning model showed minimal improvement. This highlights the importance of implementing strategies to balance datasets, such as augmentation, synthetic sample generation, and balanced sampling. This is especially important when attempting to classify wafer patterns, which can have extremely unbalanced distributions.

D. Performance Comparison across the Three Models:

The three models demonstrated distinct behaviors across all evaluation results. The baseline CNN achieved strong overall accuracy and showed stable convergence in both its accuracy and loss curves. However, its confusion matrix revealed that nearly all predictions were assigned to the "none" class. Its classification report confirmed zero precision and recall for every minority defect type. This pattern demonstrates that the baseline model relied heavily on the "none" class and failed to learn meaningful representations for rare defects. The augmented CNN displayed slower and noisier learning in its accuracy and loss plots due to the oversampling and autoencoder-based generation of data. Despite this, its confusion matrix showed substantial improvement in identifying minority classes, It displayed correct predictions appearing for categories such as Center, Edge-Ring, Loc, Random, and Scratch. Its classification report also showed increased recall for several rare defects. These results indicate that the augmented model learned on a more balanced dataset, even though its overall accuracy was lower due to the difficulty of balancing the validation dataset.

The transfer learning model reached high accuracy quickly during the frozen stage, and its training and validation curves stabilized early. Fine-tuning reduced the training loss further but did not improve validation accuracy and eventually caused the validation loss to rise slightly. Its confusion matrix resembled that of the baseline CNN, showing that all minority-class samples were misclassified as "none." The classification report also showed zero precision, recall, and F1-scores for every defect type except the majority class. These findings demonstrate that MobileNetV2's pretrained features did not transfer effectively to the defect patterns found in wafer maps. The accuracy/loss trends and confusion matrices reveal that the baseline and transfer learning approaches produced high global accuracy but failed to detect minority defects. Whereas, the augmented CNN achieved lower overall accuracy but demonstrated meaningful improvements in minority-class recognition.

A summary of the performance of all three models is provided in Table IV.

IV. Limitations:

The models presented in this study exhibited several limitations that constrained the model's overall classification performance. Both the baseline CNN and the transfer learning model were limited by the extreme class imbalance in the dataset. This prevented them from learning the patterns for minority defect classes. As a result, their performance collapsed almost entirely into the majority "none" class, as reflected by the zero precision and recall across all rare categories. The augmented CNN partially addressed this issue, but its effectiveness was

TABLE IV: Performance Summary Across All Three Models

Model	Minority-Class Detection	Majority-Class Bias	Overall Findings
Baseline CNN	Failed (0% recall)	Extremely high	Memorized "none"; no defect recognition.
Augmented CNN	Substantial improvement	Reduced	Successfully detected several defect types.
Transfer Learning	Failed (0% recall)	Extremely high	Pretrained features not aligned with wafer maps.

limited by the small number of available samples in the minority class and the restricted diversity of the autoencoder-generated samples. These reconstructions captured only general patterns and did not fully capture the subtle differences in the patterns for the minority classes. The low spatial resolution of the wafer maps further constrained all three models, as the 26×26 pixel inputs restricted the extraction of finer details of the patterns that distinguish similar defect types. The transfer learning model faced additional challenges due to its reliance on ImageNet-derived features. These high-level features were poorly matched to the binary structure of the wafer maps images , which reduced the usefulness of fine-tuning and led to overfitting without performance improvement.

Beyond this, the use of a fixed train–validation–test split also limited the efficiency of the evaluation, as performance may vary across different partitions of such an imbalanced dataset. The augmentation method used relied on a single autoencoder architecture and did not incorporate more expressive generative methods. If implemented, methods, such as diffusion models or GANs, may have produced higher-quality synthetic defects. These limitations highlight the need for more diverse data sources, higher-resolution inputs, stronger augmentation methods in future work.

V. Conclusion

This study evaluated three deep learning approaches for wafer map defect classification: a baseline CNN, an augmented CNN incorporating autoencoder-based oversampling and balanced sampling, and a transfer learning model. The baseline CNN achieved high accuracy on the majority class but failed to detect any defect categories due to extreme class imbalance and limited feature diversity. The augmented CNN addressed these issues by generating synthetic samples for rare classes and applying balancing strategies during training. This approach substantially improved minority-class recognition, yielding measurable recall across several rare defect modes. It also significantly increased overall performance. In contrast, the transfer learning model failed to improve detection, as the depth and High-level representation of the pretrained backbone suppressed the fine patterns essential for wafer defect identification.

These findings demonstrate that lighter architectures combined with targeted data augmentation can effectively enhance defect classification in imbalanced wafer datasets. At the same time, it also highlights the limitations of directly applying deep pretrained natural-image models to structured industrial data.

Future work should explore alternative methods to further improve defect classification performance. Increasing wafer map resolution may enable models to distinguish subtle geometric variations that are lost at 26×26 resolution. More advanced generative approaches, such as GANs, VAEs, or diffusion models, could create more diverse synthetic defect patterns. Designing loss functions that account for defect patterns, including focal loss or class-balanced weighting schemes, may mitigate the model's tendency to overpredict the "none" class. Hybrid architectures that use less convolutional layers with feature-weighting could preserve low-level defect patterns while improving selectivity. Finally, evaluating these models within realistic industrial inspection conditions will help determine their stability, computational efficiency, and suitability for use in manufacturing processes.

References

[1] Y.-Y. Bao, E.-C. Li, H.-Q. Yang, and B.-B. Jia, "Wafer map defect classification using autoencoder-based data augmentation and convolutional neural network," *arXiv [cs.CV]*, 17-Nov-2024.

Brief Author Biography:

Janane Sivakumar is a graduate student pursuing an M.S. in Biomedical Engineering. Her academic background includes biomedical signal processing, machine learning, and diagnostic development. Her research interests include deep learning for diagnostic systems, point-of-care biosensing, and computational modeling of biological signals.

Acknowledgements:

AI tools such as ChatGPT, and Gemini were used to assist in the development of the code and refine the paper.

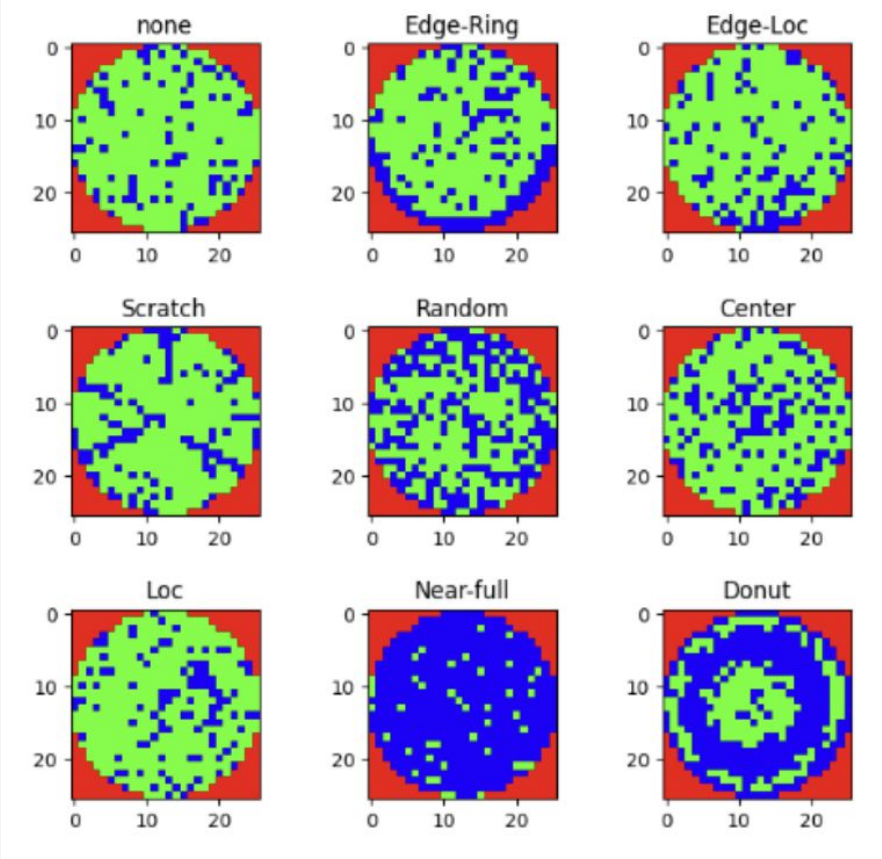
Mini Project II: CNN-Based Wafer Defect Classification

By: Janane Sivakumar
EE 267- Computer Vision w/ AI Applications
Dr. Chang Choo
December 5, 2025

Dataset Overview & Preprocessing

- **Overview:** 14,366 wafer maps, 9 classes
  - "None" class: **13,489 samples**
  - Rare classes: Donut = 1 sample, Near-Full = **16 samples**
  - Images: **26×26**, binarized
  - Minority defect patterns are extremely underrepresented.

- **Preprocessing:**
  - Load wafer maps from pickle file, convert to NumPy
  - Normalize pixels to [0,1] for CNN handling
  - Encode labels into integers
  - 70/15/15 train–val–test split



Models & Model Architectures

Baseline CNN

3 Convolutional layers, ReLU + Batch Norm + Max Pooling, L2 Regularization and Dropout (0.5), Fully connected 64-unit layer → Softmax output; SGD optimizer (LR = 1e-4, momentum 0.9), Early stopping (patience 5)

Augmented CNN

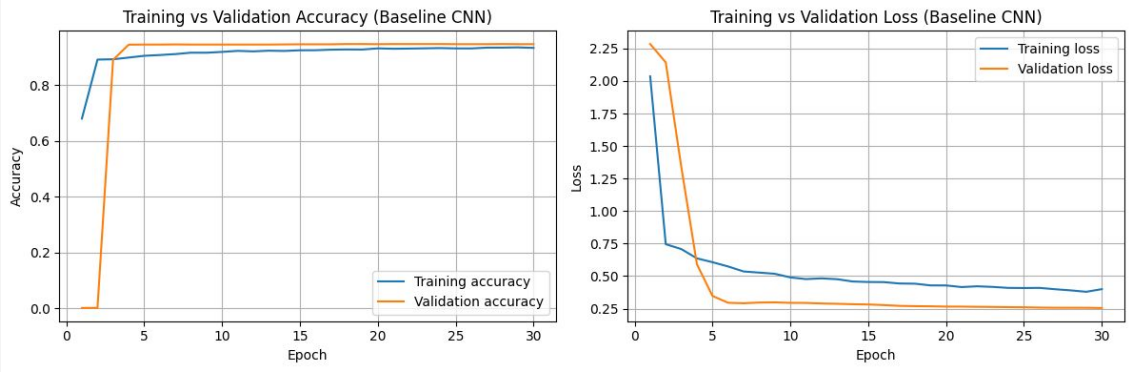
Same architecture as baseline, Autoencoder-generated synthetic samples, Oversampling rare classes, Class weighting, Balanced validation set (equal per-class sample count)

Transfer Learning-Based CNN

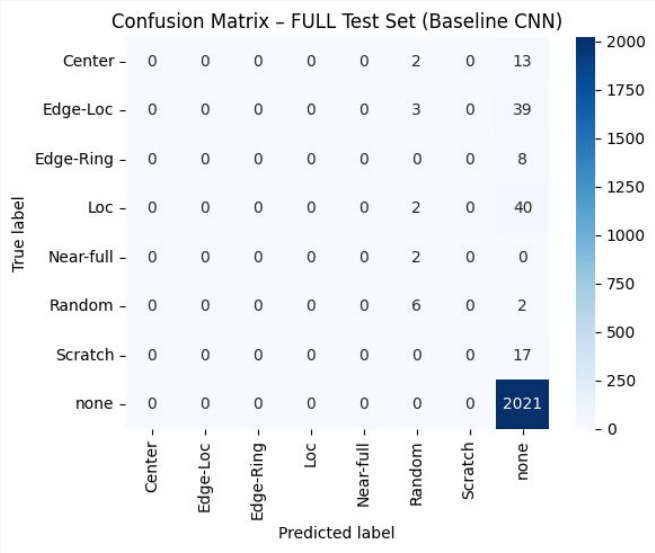
MobileNetV2 pre trained on ImageNet, Upscale wafer maps 26×26 → 96×96 RGB, Two-stage training (Stage 1: Freeze backbone, Stage 2: Fine-tune deeper layers at low LR) Adam optimizer + early stopping

Model Results: Baseline CNN

- Achieves 94% overall accuracy, but 0% defect detection due to class imbalance collapse.
- **Training & Validation Behavior**
  - Training accuracy: 0.68 → 0.91
  - Validation accuracy: 0.00 → 0.90 (due to majority-class bias)
- **Test-Set Performance**
  - None class: Precision 0.94, Recall 1.00
  - All defect classes: Recall 0.00
- **Confusion Matrix Summary**
  - 2021/2021 “none” samples predicted correctly
  - 0/134 defect samples detected

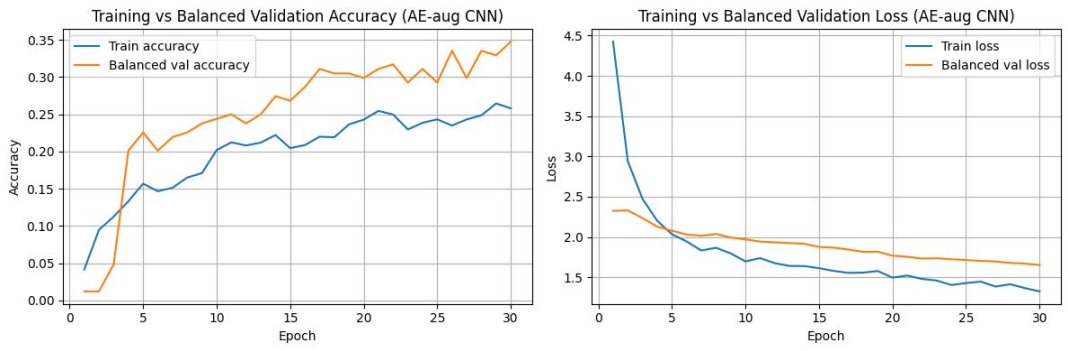



```
Classification Report (FULL test set, Baseline CNN):
              precision    recall  f1-score   support

      Center       0.00      0.00      0.00        15
    Edge-Loc       0.00      0.00      0.00        42
   Edge-Ring       0.00      0.00      0.00         8
         Loc       0.00      0.00      0.00        42
   Near-full       0.00      0.00      0.00         2
      Random       0.40      0.75      0.52         8
     Scratch       0.00      0.00      0.00        17
        none       0.94      1.00      0.97      2021

    accuracy                           0.94      2155
   macro avg       0.17      0.22      0.19      2155
weighted avg       0.89      0.94      0.91      2155
```



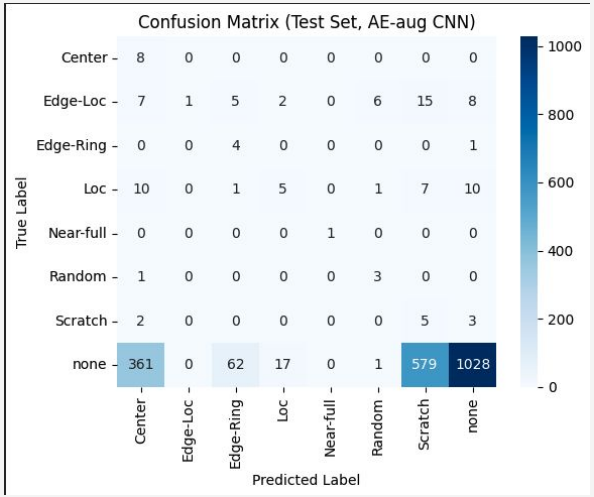
Model Results: Augmented CNN

- Successfully detect multiple rare defects
- Autoencoder augmentation + balancing solves minority-class exclusion
- **Training & Balanced Validation**
  - Balanced validation accuracy: 1% → 35%
  - Loss decreases smoothly (no overfitting)
- **Minority-Class Recall (Major Improvement)**
  - Center: 100%
  - Edge-Ring: 80%
  - Random: 75%
  - Scratch: 50%
  - Near-full: 100%
- **None-Class Adjustment**
  - None-class recall drops to 50% (improves defect sensitivity)



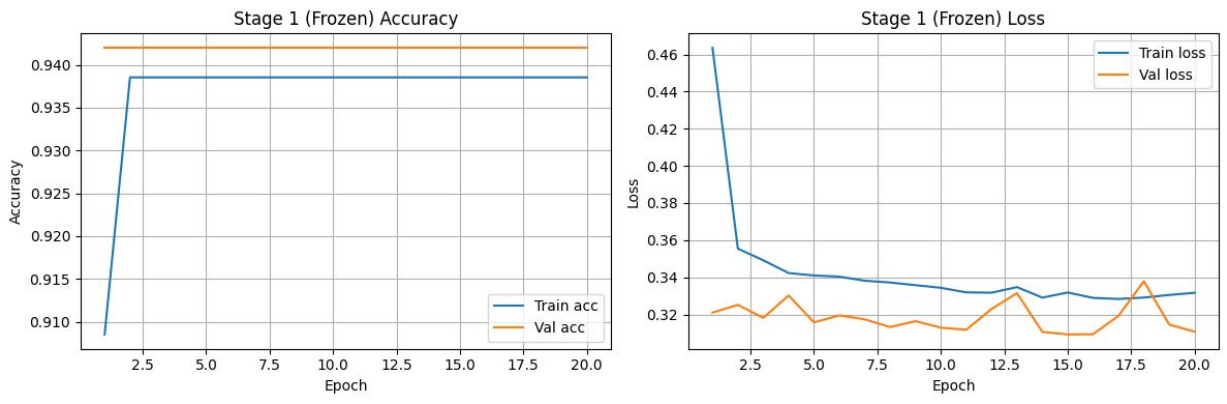

```
Classification Report (original test set):
              precision    recall  f1-score   support

      Center       0.02      1.00      0.04         8
    Edge-Loc       1.00      0.02      0.04        45
   Edge-Ring       0.06      0.80      0.10         5
         Loc       0.21      0.15      0.17        34
   Near-full       1.00      1.00      1.00         1
      Random       0.27      0.75      0.40         4
     Scratch       0.01      0.50      0.02        10
        none       0.98      0.50      0.66      2048

   micro avg       0.49      0.49      0.49      2155
   macro avg       0.44      0.59      0.30      2155
weighted avg       0.96      0.49      0.64      2155
```

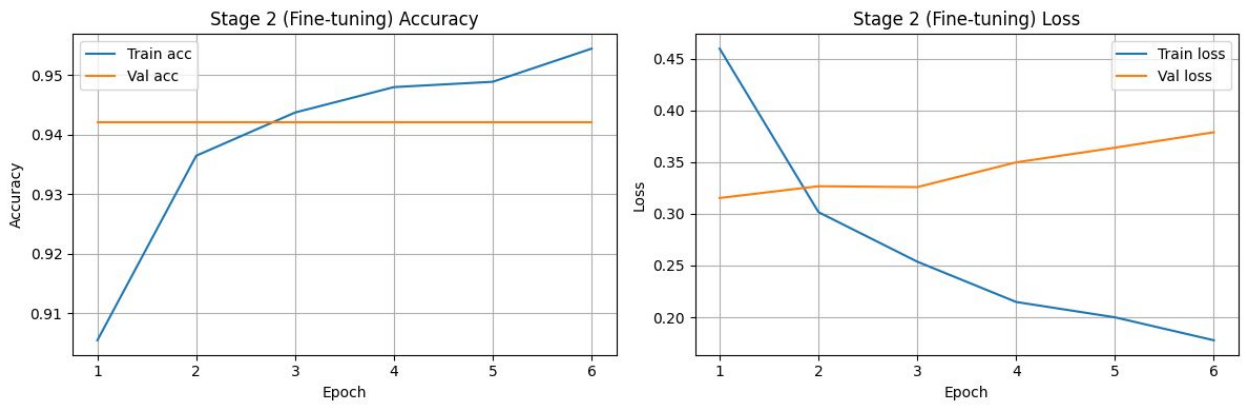
```
Per-class accuracy on test set:
Center: 100.00% (n=8)
Donut: N/A (no samples in test set)
Edge-Loc: 2.22% (n=45)
Edge-Ring: 80.00% (n=5)
Loc: 14.71% (n=34)
Near-full: 100.00% (n=1)
Random: 75.00% (n=4)
Scratch: 50.00% (n=10)
none: 50.20% (n=2048)
```



Model Results: Transfer-Learning CNN

- Transfer learning fails to detect defects, mirroring the baseline failure
- Pretrained features too abstract for low-resolution (26×26) wafer maps
- **Training Behavior**
  - Training accuracy: 0.91 → 0.95
  - Validation accuracy: stable at 0.94
  - Validation loss unchanged → no real improvement

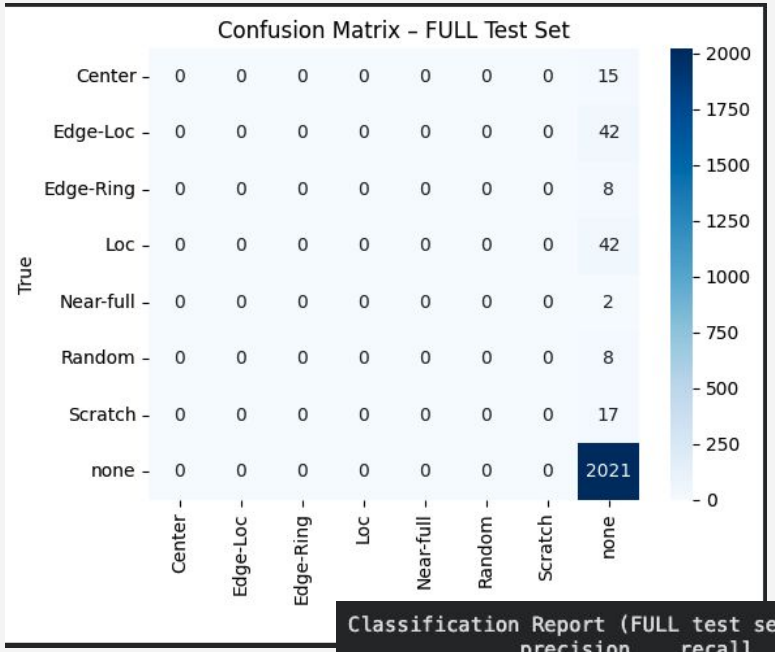




Model Results: Transfer-Learning CNN (cont'd)

- Classification report shows 0% recall for all defect types
- Confusion matrix resembles the baseline → predicts only "none"
- **Why it failed:**
  - MobileNetV2 is **too deep** for 26×26 symbolic wafer maps
  - Pretrained ImageNet filters suppress subtle defect features
  - Dataset too imbalanced for deep models to adapt

- Transfer learning is not suitable for this domain under current constraints.




```
Classification Report (FULL test set):
              precision    recall  f1-score   support

      Center       0.00      0.00      0.00        15
    Edge-Loc       0.00      0.00      0.00        42
   Edge-Ring       0.00      0.00      0.00         8
         Loc       0.00      0.00      0.00        42
   Near-full       0.00      0.00      0.00         2
      Random       0.00      0.00      0.00         8
     Scratch       0.00      0.00      0.00        17
        none       0.94      1.00      0.97      2021

    accuracy                           0.94      2155
   macro avg       0.12      0.12      0.12      2155
weighted avg       0.88      0.94      0.91      2155
```

Model Comparison & Limitations

- **Model Comparison:**
  - <u>Baseline CNN</u>
    - No minority detection
    - Extremely high bias toward “none”
  - <u>Augmented CNN</u>
    - Best minority-class detection
    - Improved recall & meaningful recognition
  - <u>Transfer Learning (MobileNetV2)</u>
    - Stable training but poor defect detection
    - Depth + domain mismatch → feature suppression
  - **Overall:** Data augmentation outperforms deeper pretrained models for wafer maps.

- **Limitations:**
  - Extremely Low Image Resolution
  - Residual Class Imbalance even after Data Augmentation
  - Synthetic Data Limitations due to artifacts and inability to capture true variability
  - Resolution and Defect oversuppression of transfer learning models
  - Limited test data diversity and skewed performance metrics

Conclusion & Future Directions

- **Conclusion:**
  - Baseline CNN fails due to imbalance.
  - Augmented CNN significantly improves rare-defect recall.
  - Transfer learning does not generalize due to model depth and data mismatch.
  - Lightweight CNNs + data augmentation are more effective for wafer defect detection.

- **Future Work:**
  - Higher-resolution wafer maps
  - GAN / VAE / diffusion-based generation for richer defects
  - Loss functions tailored to imbalance (focal loss, class-balanced loss)
  - Hybrid CNN + attention models
  - Domain-specific pretraining on synthetic wafers
  - Deployment testing in real industrial inspection pipelines

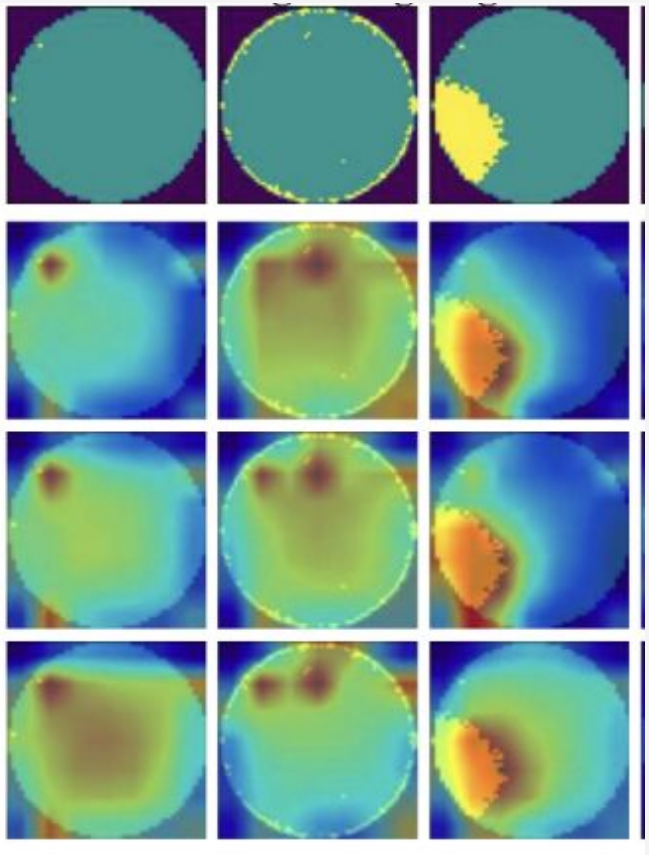


Image: (Xu, Yu, and Essaf, 2022)

References

1. Y.-Y. Bao, E.-C. Li, H.-Q. Yang, and B.-B. Jia, “Wafer map defect classification using autoencoder-based data augmentation and convolutional neural network,” *arXiv [cs.CV]*, 17-Nov-2024.
2. Q. Xu, N. Yu, and F. Essaf, “Improved wafer map inspection using attention mechanism and cosine normalization,” *Machines*, vol. 10, no. 2, p. 146, Feb. 2022.

Thank you for Listening!

Baseline_CNN

December 6, 2025

[ ]:
```
# Baseline CNN Model
#Importing Necessary Modules:
import os
import random
import datetime

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

import tensorflow as tf
from tensorflow import keras
from keras import layers, regularizers

from sklearn.metrics import confusion_matrix, classification_report
from sklearn.model_selection import train_test_split

# Setting seeds for saving model weights (ensures the results can be reproduced)
seed = 42
os.environ["PYTHONHASHSEED"] = str(seed)
random.seed(seed)
np.random.seed(seed)
tf.random.set_seed(seed)

print("TF version:", tf.__version__)

# Loading Data from Pickle File

# Mounting Drive
from google.colab import drive
drive.mount('/content/drive')

DATA_PKL = "/content/waferImg26x26.pkl"
df = pd.read_pickle(DATA_PKL)
print("DataFrame shape:", df.shape)
```

```python
# Extracting the Images and their Labels
X_list = df["images"].to_numpy()
X = np.stack(X_list).astype("float32")
print("Raw X shape:", X.shape)

# Checking if shape is (N, C, H, W), transpose to (N, H, W, C)
if X.ndim == 4 and X.shape[1] in (1, 3):
    X = np.transpose(X, (0, 2, 3, 1))
    print("Transposed X shape (NHWC):", X.shape)

# Ensuring correct channel dimension exists
if X.ndim == 3:
    X = X[..., np.newaxis]
    print("Added channel dim:", X.shape)
else:
    print("X already has channels:", X.shape)

IMG_SIZE = X.shape[1]
channels = X.shape[-1]

# Normalizing Pixel Intensity Values to [0,1]
if X.max() > 1.0:
    X = X / 255.0
print("Pixel range after normalization:", X.min(), "to", X.max())

# Cleaning the Image Labels
labels_raw = df["labels"].to_numpy()

labels_clean = []
for v in labels_raw:
    if isinstance(v, (list, np.ndarray)) and len(v) > 0:
        labels_clean.append(v[0])
    else:
        labels_clean.append(v)

y_str = np.array(labels_clean)
print("Example labels:", y_str[:10])

# Encoding the string labels to integers
unique_labels, y_int = np.unique(y_str, return_inverse=True)
class_names = [str(lbl) for lbl in unique_labels]
y = y_int.astype("int32")
num_classes = len(class_names)

print("Classes:", class_names)
print("num_classes:", num_classes)
```

```python
# Splitting the Dataset into Training, Testing and Validation Datasets
N = len(X)
print("Total samples:", N)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.30,␣
↪random_state=42, shuffle=True)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.50,␣
↪random_state=42, shuffle=True)

print("Train:", X_train.shape, y_train.shape)
print("Val:  ", X_val.shape,   y_val.shape)
print("Test: ", X_test.shape,  y_test.shape)

BATCH_SIZE = 128

train_ds = tf.data.Dataset.from_tensor_slices((X_train, y_train)) \
                          .shuffle(1000, seed=seed,␣
↪reshuffle_each_iteration=False) \
                          .batch(BATCH_SIZE)

val_ds   = tf.data.Dataset.from_tensor_slices((X_val,   y_val)) \
                          .batch(BATCH_SIZE)

test_ds  = tf.data.Dataset.from_tensor_slices((X_test,  y_test)) \
                          .batch(BATCH_SIZE)


# Defining the Baseline CNN Model
def make_model(input_shape, num_classes):
    inputs = keras.Input(shape=input_shape)

    # 1st Convolutional Layer
    x = layers.Conv2D(8, 3, padding="same", activation="relu",␣
↪kernel_regularizer=regularizers.l2(5e-4))(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPooling2D()(x)

    # 2nd Convolutional Layer
    x = layers.Conv2D(16, 3, padding="same", activation="relu",␣
↪kernel_regularizer=regularizers.l2(5e-4))(x)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPooling2D()(x)

    # 3rd Convolutional Layer
    x = layers.Conv2D(32, 3, padding="same", activation="relu",␣
↪kernel_regularizer=regularizers.l2(5e-4))(x)
    x = layers.BatchNormalization()(x)
```

```
    x = layers.MaxPooling2D()(x)

    # Dropout and Output Layer
    x = layers.Dropout(0.5)(x)
    x = layers.Flatten()(x)
    x = layers.Dense(64, activation="relu")(x)
    x = layers.Dropout(0.5)(x)
    outputs = layers.Dense(num_classes, activation="softmax")(x)

    return keras.Model(inputs, outputs, name="wafer_cnn_baseline")

input_shape = (IMG_SIZE, IMG_SIZE, channels)
print("Model input_shape:", input_shape)

model = make_model(input_shape, num_classes)
model.summary()

#Compiling and Training the Model
model.compile(optimizer=keras.optimizers.SGD(learning_rate=1e-4, momentum=0.
 ↪9),loss="sparse_categorical_crossentropy", metrics=["accuracy"],)

early_stop = keras.callbacks.EarlyStopping( monitor="val_loss", patience=5,␣
 ↪restore_best_weights=True)

history = model.fit(train_ds, epochs=30, validation_data=val_ds,␣
 ↪callbacks=[early_stop], verbose=1,)


# Plotting the Accuracy and Loss Plot of the CNN
hist = history.history
acc = hist.get("accuracy", hist.get("acc"))
val_acc = hist.get("val_accuracy", hist.get("val_acc"))
loss = hist["loss"]
val_loss = hist["val_loss"]
epochs = range(1, len(loss) + 1)

plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(epochs, acc, label="Training accuracy")
plt.plot(epochs, val_acc, label="Validation accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.title("Training vs Validation Accuracy (Baseline CNN)")
plt.legend()
plt.grid(True)
```

```
plt.subplot(1, 2, 2)
plt.plot(epochs, loss, label="Training loss")
plt.plot(epochs, val_loss, label="Validation loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Training vs Validation Loss (Baseline CNN)")
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()


# Evaluating the Model on the Test Dataset
print("\nBaseline CNN: Evaluation on FULL test set")
test_loss, test_acc = model.evaluate(test_ds, verbose=1)
print(f"Test loss: {test_loss:.4f}  |  Test accuracy: {test_acc:.4f}")

# Generating the Classification Report
y_pred_probs = model.predict(test_ds)
y_pred = np.argmax(y_pred_probs, axis=1)
y_true = y_test  # for clarity

labels_present = np.unique(y_true)
target_names_present = [class_names[i] for i in labels_present]

print("\nClassification Report (FULL test set, Baseline CNN):")
print(classification_report(y_true, y_pred, labels=labels_present,␣
 ↪target_names=target_names_present, zero_division=0,))

# Generating the Confusion matrix
cm = confusion_matrix(y_true, y_pred, labels=labels_present)
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",␣
 ↪xticklabels=target_names_present, yticklabels=target_names_present,)
plt.xlabel("Predicted label")
plt.ylabel("True label")
plt.title("Confusion Matrix - FULL Test Set (Baseline CNN)")
plt.tight_layout()
plt.show()

# Saving the Model and its Predictions for Future Retrieval
run_id = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
base_dir = "/content/drive/MyDrive/wafer_baseline_runs"
run_dir = os.path.join(base_dir, f"run_{run_id}")
os.makedirs(run_dir, exist_ok=True)
print(f"\nSaving this run to: {run_dir}")
```

```python
# Saving the model
model_path = os.path.join(run_dir, "model_baseline.keras")
model.save(model_path)
print(f"Model saved to: {model_path}")

# Saving the model predictions and labels
np.save(os.path.join(run_dir, "y_pred.npy"), y_pred)
np.save(os.path.join(run_dir, "y_true.npy"), y_true)
print("Predictions and labels saved.")

# Saving the Classification Report as a CSV File
report_dict = classification_report( y_true, y_pred, labels=labels_present,␣
↪target_names=target_names_present, output_dict=True, zero_division=0,)
report_df = pd.DataFrame(report_dict).T
report_csv_path = os.path.join(run_dir, "classification_report.csv")
report_df.to_csv(report_csv_path)
print(f"Classification report saved to: {report_csv_path}")
```

```
TF version: 2.19.0
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
DataFrame shape: (14366, 2)
Raw X shape: (14366, 3, 26, 26)
Transposed X shape (NHWC): (14366, 26, 26, 3)
X already has channels: (14366, 26, 26, 3)
Pixel range after normalization: 0.0 to 1.0
Example labels: ['none' 'none' 'none' 'none' 'none' 'none' 'none' 'none' 'none'
'none']
Classes: ['Center', 'Donut', 'Edge-Loc', 'Edge-Ring', 'Loc', 'Near-full',
'Random', 'Scratch', 'none']
num_classes: 9
Total samples: 14366
Train: (10056, 26, 26, 3) (10056,)
Val:   (2155, 26, 26, 3) (2155,)
Test:  (2155, 26, 26, 3) (2155,)
Model input_shape: (26, 26, 3)
```

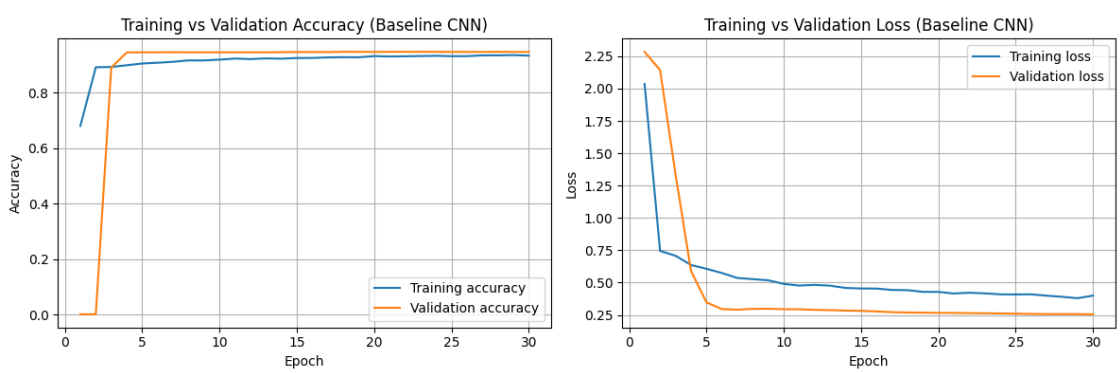
Model: "wafer_cnn_baseline"

Layer (type)	Output Shape	Param #
input_layer_5 (InputLayer)	(None, 26, 26, 3)	0
conv2d_15 (Conv2D)	(None, 26, 26, 8)	224
batch_normalization_15	(None, 26, 26, 8)	32

```
 (BatchNormalization)

 max_pooling2d_15 (MaxPooling2D)   (None, 13, 13, 8)                0

 conv2d_16 (Conv2D)                (None, 13, 13, 16)            1,168

 batch_normalization_16            (None, 13, 13, 16)               64
 (BatchNormalization)

 max_pooling2d_16 (MaxPooling2D)   (None, 6, 6, 16)                 0

 conv2d_17 (Conv2D)                (None, 6, 6, 32)              4,640

 batch_normalization_17            (None, 6, 6, 32)               128
 (BatchNormalization)

 max_pooling2d_17 (MaxPooling2D)   (None, 3, 3, 32)                 0

 dropout_10 (Dropout)              (None, 3, 3, 32)                 0

 flatten_5 (Flatten)               (None, 288)                      0

 dense_10 (Dense)                  (None, 64)                  18,496

 dropout_11 (Dropout)              (None, 64)                       0

 dense_11 (Dense)                  (None, 9)                      585


 Total params: 25,337 (98.97 KB)

 Trainable params: 25,225 (98.54 KB)

 Non-trainable params: 112 (448.00 B)

Epoch 1/30
79/79              8s 80ms/step -
accuracy: 0.4169 - loss: 3.9889 - val_accuracy: 0.0019 - val_loss: 2.2846
Epoch 2/30
79/79              8s 99ms/step -
accuracy: 0.8978 - loss: 0.7299 - val_accuracy: 0.0023 - val_loss: 2.1432
Epoch 3/30
79/79              6s 75ms/step -
accuracy: 0.8967 - loss: 0.6867 - val_accuracy: 0.8910 - val_loss: 1.3380
Epoch 4/30
79/79              8s 100ms/step -
```

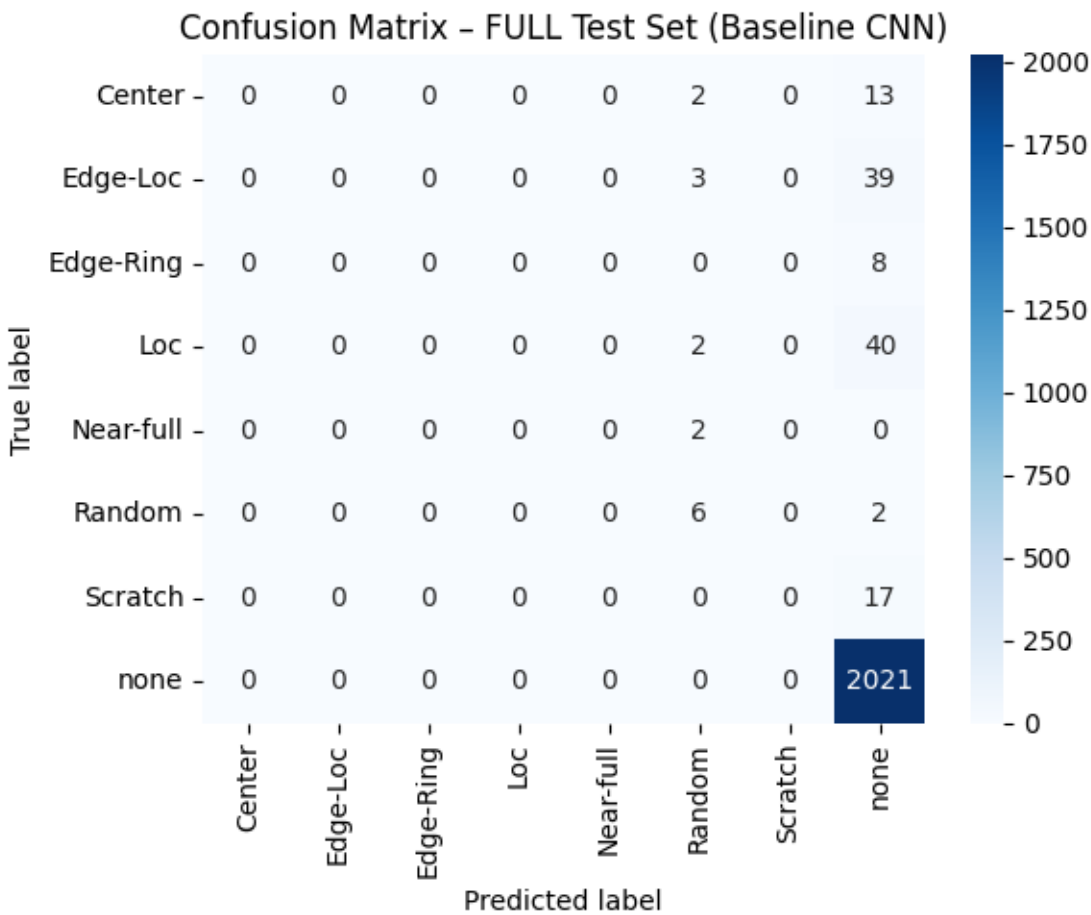
```
accuracy: 0.9003 - loss: 0.6099 - val_accuracy: 0.9452 - val_loss: 0.5917
Epoch 5/30
79/79              8s 75ms/step -
accuracy: 0.9073 - loss: 0.5860 - val_accuracy: 0.9457 - val_loss: 0.3476
Epoch 6/30
79/79              8s 100ms/step -
accuracy: 0.9092 - loss: 0.5626 - val_accuracy: 0.9457 - val_loss: 0.2949
Epoch 7/30
79/79              10s 92ms/step -
accuracy: 0.9146 - loss: 0.5039 - val_accuracy: 0.9462 - val_loss: 0.2914
Epoch 8/30
79/79              7s 85ms/step -
accuracy: 0.9204 - loss: 0.5064 - val_accuracy: 0.9457 - val_loss: 0.2967
Epoch 9/30
79/79              7s 88ms/step -
accuracy: 0.9212 - loss: 0.4894 - val_accuracy: 0.9457 - val_loss: 0.2979
Epoch 10/30
79/79              7s 86ms/step -
accuracy: 0.9247 - loss: 0.4631 - val_accuracy: 0.9457 - val_loss: 0.2944
Epoch 11/30
79/79              6s 80ms/step -
accuracy: 0.9290 - loss: 0.4530 - val_accuracy: 0.9457 - val_loss: 0.2939
Epoch 12/30
79/79              7s 93ms/step -
accuracy: 0.9255 - loss: 0.4548 - val_accuracy: 0.9457 - val_loss: 0.2895
Epoch 13/30
79/79              6s 75ms/step -
accuracy: 0.9283 - loss: 0.4472 - val_accuracy: 0.9457 - val_loss: 0.2872
Epoch 14/30
79/79              8s 98ms/step -
accuracy: 0.9257 - loss: 0.4376 - val_accuracy: 0.9462 - val_loss: 0.2839
Epoch 15/30
79/79              6s 75ms/step -
accuracy: 0.9290 - loss: 0.4444 - val_accuracy: 0.9466 - val_loss: 0.2820
Epoch 16/30
79/79              8s 98ms/step -
accuracy: 0.9290 - loss: 0.4366 - val_accuracy: 0.9466 - val_loss: 0.2772
Epoch 17/30
79/79              6s 75ms/step -
accuracy: 0.9301 - loss: 0.4129 - val_accuracy: 0.9466 - val_loss: 0.2713
Epoch 18/30
79/79              8s 98ms/step -
accuracy: 0.9328 - loss: 0.4224 - val_accuracy: 0.9476 - val_loss: 0.2690
Epoch 19/30
79/79              6s 75ms/step -
accuracy: 0.9312 - loss: 0.4074 - val_accuracy: 0.9476 - val_loss: 0.2679
Epoch 20/30
79/79              11s 80ms/step -
```

```
accuracy: 0.9368 - loss: 0.4085 - val_accuracy: 0.9471 - val_loss: 0.2657
Epoch 21/30
79/79              12s 98ms/step -
accuracy: 0.9368 - loss: 0.3955 - val_accuracy: 0.9476 - val_loss: 0.2655
Epoch 22/30
79/79              10s 91ms/step -
accuracy: 0.9356 - loss: 0.4016 - val_accuracy: 0.9476 - val_loss: 0.2639
Epoch 23/30
79/79              7s 82ms/step -
accuracy: 0.9374 - loss: 0.3908 - val_accuracy: 0.9476 - val_loss: 0.2628
Epoch 24/30
79/79              11s 98ms/step -
accuracy: 0.9366 - loss: 0.3903 - val_accuracy: 0.9476 - val_loss: 0.2615
Epoch 25/30
79/79              6s 75ms/step -
accuracy: 0.9380 - loss: 0.3834 - val_accuracy: 0.9471 - val_loss: 0.2603
Epoch 26/30
79/79              10s 77ms/step -
accuracy: 0.9375 - loss: 0.3825 - val_accuracy: 0.9471 - val_loss: 0.2581
Epoch 27/30
79/79              12s 99ms/step -
accuracy: 0.9393 - loss: 0.3847 - val_accuracy: 0.9471 - val_loss: 0.2568
Epoch 28/30
79/79              6s 78ms/step -
accuracy: 0.9395 - loss: 0.3727 - val_accuracy: 0.9476 - val_loss: 0.2568
Epoch 29/30
79/79              8s 99ms/step -
accuracy: 0.9411 - loss: 0.3568 - val_accuracy: 0.9471 - val_loss: 0.2565
Epoch 30/30
79/79              6s 75ms/step -
accuracy: 0.9374 - loss: 0.3739 - val_accuracy: 0.9471 - val_loss: 0.2549
```




```
Baseline CNN: Evaluation on FULL test set
```

```
17/17              0s 23ms/step -
accuracy: 0.9394 - loss: 0.2898
Test loss: 0.2941  |  Test accuracy: 0.9406
17/17              1s 29ms/step

Classification Report (FULL test set, Baseline CNN):
              precision    recall  f1-score   support

      Center       0.00      0.00      0.00        15
    Edge-Loc       0.00      0.00      0.00        42
   Edge-Ring       0.00      0.00      0.00         8
         Loc       0.00      0.00      0.00        42
   Near-full       0.00      0.00      0.00         2
      Random       0.40      0.75      0.52         8
     Scratch       0.00      0.00      0.00        17
        none       0.94      1.00      0.97      2021

    accuracy                           0.94      2155
   macro avg       0.17      0.22      0.19      2155
weighted avg       0.89      0.94      0.91      2155
```



```
Saving this run to:
/content/drive/MyDrive/wafer_baseline_runs/run_20251206_013031
Model saved to: /content/drive/MyDrive/wafer_baseline_runs/run_20251206_013031/m
odel_baseline.keras
Predictions and labels saved.
Classification report saved to: /content/drive/MyDrive/wafer_baseline_runs/run_2
0251206_013031/classification_report.csv
```

Augmented & Transfer Learning CNN Code and Output

December 6, 2025

[ ]:
```
# Augmented CNN
# Importing the Necessary Modules:
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras
from keras import layers, regularizers

import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.metrics import confusion_matrix, classification_report
from sklearn.utils.class_weight import compute_class_weight

# Loading the Data from the Pickle File
from google.colab import drive
drive.mount('/content/drive')

DATA_PKL = "/content/waferImg26x26.pkl"

df = pd.read_pickle(DATA_PKL)
print("Loaded object type:", type(df))
print(df.head())

# Extracting the images and their labels
X_list = df["images"].to_numpy()
X = np.stack(X_list).astype("float32")
print("Raw X shape:", X.shape)

# Checking if shape is (N, C, H, W), transpose to (N, H, W, C)
if X.ndim == 4 and X.shape[1] in (1, 3):
    X = np.transpose(X, (0, 2, 3, 1))
    print("Transposed X shape (NHWC):", X.shape)

# Ensuring correct channel dimension exists
if X.ndim == 3:
    X = X[..., np.newaxis]
```

```
    print("Added channel dim:", X.shape)
else:
    print("X already has channels:", X.shape)

IMG_SIZE = X.shape[1]
channels = X.shape[-1]
print("IMG_SIZE:", IMG_SIZE, "channels:", channels)

# Normalizing Pixel Intensity Values to [0,1]
if X.max() > 1.0:
    X = X / 255.0
print("X range:", X.min(), "to", X.max())

# Cleaning the Image Labels
labels_raw = df["labels"].to_numpy()

labels_clean = []
for v in labels_raw:
    if isinstance(v, (list, np.ndarray)) and len(v) > 0:
        labels_clean.append(v[0])
    else:
        labels_clean.append(v)

y_str = np.array(labels_clean)
print("Example labels:", y_str[:10])

# Encoding the string labels to integers
unique_labels, y_int = np.unique(y_str, return_inverse=True)
class_names = [str(lbl) for lbl in unique_labels]
y = y_int.astype("int32")
num_classes = len(class_names)

print("Classes:", class_names)
print("num_classes:", num_classes)

# Splitting the Dataset into Training, Testing and Validation Datasets via␣
↪random permutations (ensures that the samples are randomly distributed␣
↪across classes)
N = len(X)
rng = np.random.default_rng(seed=42)
perm = rng.permutation(N)

train_end = int(0.7 * N)
val_end   = int(0.85 * N)

idx_train = perm[:train_end]
idx_val   = perm[train_end:val_end]
```

```
idx_test  = perm[val_end:]

X_train, y_train = X[idx_train], y[idx_train]
X_val,   y_val   = X[idx_val],   y[idx_val]
X_test,  y_test  = X[idx_test],  y[idx_test]

print("Train:", X_train.shape, y_train.shape)
print("Val:  ", X_val.shape,   y_val.shape)
print("Test: ", X_test.shape,  y_test.shape)

# Creating the autoencoder for data augmentation
# Building the dataset needing augmenting
BATCH_AE = 128
ae_train_ds = (tf.data.Dataset.from_tensor_slices((X_train, X_train)).
 ↪shuffle(1000).batch(BATCH_AE).prefetch(tf.data.AUTOTUNE))

# Defining the autoencoder:

ae_inputs = keras.Input(shape=(IMG_SIZE, IMG_SIZE, channels)) # defining the␣
 ↪autoencoder inputs

# Creating the encoder for the images
x = layers.Conv2D(32, 3, activation="relu", padding="same")(ae_inputs)
x = layers.Conv2D(16, 3, activation="relu", padding="same")(x)
encoded = layers.Conv2D(8, 3, activation="relu", padding="same",␣
 ↪name="encoded")(x)

# Creating the decoder for the images
x = layers.Conv2D(16, 3, activation="relu", padding="same")(encoded)
x = layers.Conv2D(32, 3, activation="relu", padding="same")(x)
ae_outputs = layers.Conv2D(channels, 3, activation="sigmoid", padding="same")(x)

# Constructing and compiling the final autoencoder
autoencoder = keras.Model(ae_inputs, ae_outputs, name="wafer_autoencoder")
autoencoder.compile(optimizer="adam", loss="mse")
autoencoder.summary()

print("\nTraining autoencoder for data augmentation...")
ae_history = autoencoder.fit(ae_train_ds,epochs=20) # training the autoencoder␣
 ↪with the images
print("Autoencoder training complete.")


# Applying the Autoencoder to the Minority Classes in the Dataset

#Identifying the Minority classes in the wafer dataset
classes_train, counts_train = np.unique(y_train, return_counts=True)
```

```python
print("\nTrain class counts:")
for c, cnt in zip(classes_train, counts_train):
    print(f"  {class_names[c]} (label {c}): {cnt}")

# Setting the threshold used to identify the minority classes
min_count = 100
rare_labels = [c for c, cnt in zip(classes_train, counts_train) if cnt <␣
 ↪min_count]

print("\nRare classes selected for AE augmentation (count < "f"{min_count}):")
for c in rare_labels:
    print(f"  {class_names[c]} (label {c})")

# Reconstructing the Minority Class Images using the Autoencoder in Batches
def ae_generate_batch(images, batch_size=128):
    all_out = []
    for i in range(0, len(images), batch_size):
        batch = images[i:i+batch_size]
        recon = autoencoder.predict(batch, verbose=0)
        all_out.append(recon)
    return np.concatenate(all_out, axis=0)

# For each rare class, oversample with AE reconstructions
# Oversampling the autoencoder reconstructed images to reduce class bias
augmented_X_list = [X_train]
augmented_y_list = [y_train]

n_aug_per_sample = 3 # setting the number of synthetic samples to recreate for␣
 ↪each minority class in the original dataset

for c in rare_labels:
    idx_c = np.where(y_train == c)[0]
    X_c = X_train[idx_c]

    if len(X_c) == 0:
        continue

    print(f"\nAugmenting class {class_names[c]} (n={len(X_c)}) with␣
 ↪"f"{n_aug_per_sample}x AE reconstructions...")

    # Sampling each image 3 times then reconstruct
    X_c_repeat = np.repeat(X_c, n_aug_per_sample, axis=0)
    X_c_aug = ae_generate_batch(X_c_repeat)

    y_c_aug = np.full(len(X_c_aug), c, dtype="int32")

    augmented_X_list.append(X_c_aug)
```

```
    augmented_y_list.append(y_c_aug)

# Stacking the final, reconstructed images
X_train_aug = np.concatenate(augmented_X_list, axis=0)
y_train_aug = np.concatenate(augmented_y_list, axis=0)

print("\nAugmented train shape:", X_train_aug.shape, y_train_aug.shape)

# Balancing the Validation Dataset:
def make_balanced_subset(X, y, max_per_class=40):
    X_bal_list = []
    y_bal_list = []
    classes = np.unique(y)
    for c in classes:
        idx_c = np.where(y == c)[0]
        local_rng = np.random.default_rng(seed=123 + int(c))
        local_rng.shuffle(idx_c)
        chosen = idx_c[:max_per_class]
        X_bal_list.append(X[chosen])
        y_bal_list.append(y[chosen])
    return np.concatenate(X_bal_list, axis=0), np.concatenate(y_bal_list,␣
 ↪axis=0)

X_val_bal, y_val_bal = make_balanced_subset(X_val, y_val, max_per_class=40)

print("\nBalanced val shape:", X_val_bal.shape, y_val_bal.shape)
u_bal, c_bal = np.unique(y_val_bal, return_counts=True)
print("Balanced val class counts:")
for c, cnt in zip(u_bal, c_bal):
    print(f"  {class_names[c]}: {cnt}")

# Creating the Training, Validation, and Test Datasets
BATCH_SIZE = 128
train_ds = (tf.data.Dataset.from_tensor_slices((X_train_aug, y_train_aug)).
 ↪shuffle(2000).batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE))
val_ds_bal = (tf.data.Dataset.from_tensor_slices((X_val_bal, y_val_bal)).
 ↪batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE))
test_ds = (tf.data.Dataset.from_tensor_slices((X_test, y_test)).
 ↪batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE))

# Computing the Class weights after data augmentation was applied (helps gauge␣
 ↪whether the augmenation was successful at removing the imbalance)
classes_aug = np.unique(y_train_aug)
class_weights_arr = compute_class_weight(class_weight="balanced",␣
 ↪classes=classes_aug, y=y_train_aug)
```

```python
class_weights = {int(c): float(w) for c, w in zip(classes_aug,␣
↪class_weights_arr)}
print("\nClass weights (after AE augmentation):")
for c, w in class_weights.items():
    print(f"  {class_names[c]}: {w:.3f}")

# Creating the CNN Model
def make_cnn(input_shape, num_classes):
    inputs = keras.Input(shape=input_shape)

    # First Convolutional Layer
    x = layers.Conv2D(8, 3, padding="same", activation="relu",␣
↪kernel_regularizer=regularizers.l2(5e-4))(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPooling2D()(x)

    # Second Convolutional Layer
    x = layers.Conv2D(16, 3, padding="same", activation="relu",␣
↪kernel_regularizer=regularizers.l2(5e-4))(x)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPooling2D()(x)

    # Third Convolutional Layer
    x = layers.Conv2D(32, 3, padding="same", activation="relu",␣
↪kernel_regularizer=regularizers.l2(5e-4))(x)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPooling2D()(x)

    # Dropout and Output Layer
    x = layers.Dropout(0.5)(x)
    x = layers.Flatten()(x)
    x = layers.Dense(64, activation="relu")(x)
    x = layers.Dropout(0.5)(x)
    outputs = layers.Dense(num_classes, activation="softmax")(x)

    return keras.Model(inputs, outputs, name="wafer_cnn_ae_aug")

input_shape = (IMG_SIZE, IMG_SIZE, channels)
print("\nCNN input_shape:", input_shape)
model = make_cnn(input_shape, num_classes)
model.summary()

model.compile(optimizer=keras.optimizers.SGD(learning_rate=1e-4, momentum=0.9),␣
↪loss="sparse_categorical_crossentropy", metrics=["accuracy"],)

# Training the Constructed CNN using the augmented dataset
```

```
early_stop = keras.callbacks.EarlyStopping(monitor="val_loss", patience=5,␣
↪restore_best_weights=True)
history = model.fit(train_ds, epochs=30, validation_data=val_ds_bal,␣
↪class_weight=class_weights, callbacks=[early_stop],)

# Plotting the Accuracy and Loss of the Model
hist = history.history
acc = hist.get("accuracy", hist.get("acc"))
val_acc = hist.get("val_accuracy", hist.get("val_acc"))
loss = hist["loss"]
val_loss = hist["val_loss"]

epochs_range = range(1, len(loss) + 1)

plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label="Train accuracy")
plt.plot(epochs_range, val_acc, label="Balanced val accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.title("Training vs Balanced Validation Accuracy (AE-aug CNN)")
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label="Train loss")
plt.plot(epochs_range, val_loss, label="Balanced val loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Training vs Balanced Validation Loss (AE-aug CNN)")
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

# Evaluating the Model on the Original, Imbalanced Dataset
print("\nEvaluating on ORIGINAL (imbalanced) test set...")
test_loss, test_acc = model.evaluate(test_ds)
print(f"Test loss: {test_loss:.4f}  |  Test accuracy: {test_acc:.4f}")

# Obtaining the Model Predictions
y_pred_probs = model.predict(test_ds)
y_pred = np.argmax(y_pred_probs, axis=1)
y_true = y_test
```

```python
# Generating the Classification Report:
labels_present = np.unique(y_true)
target_names_present = [class_names[i] for i in labels_present]

print("\nClassification Report (original test set):")
print(classification_report(y_true, y_pred, labels=labels_present,␣
↪target_names=target_names_present, zero_division=0))

# Generating the Confusion Matrix:
cm = confusion_matrix(y_true, y_pred, labels=labels_present)
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",␣
↪xticklabels=target_names_present, yticklabels=target_names_present)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix (Test Set, AE-aug CNN)")
plt.tight_layout()
plt.show()
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
Loaded object type: <class 'pandas.core.frame.DataFrame'>
                                                 images  labels
0  [[[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0…  [none]
1  [[[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0…  [none]
2  [[[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0…  [none]
3  [[[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0…  [none]
4  [[[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0…  [none]
Raw X shape: (14366, 3, 26, 26)
Transposed X shape (NHWC): (14366, 26, 26, 3)
X already has channels: (14366, 26, 26, 3)
IMG_SIZE: 26 channels: 3
X range: 0.0 to 1.0
Example labels: ['none' 'none' 'none' 'none' 'none' 'none' 'none' 'none' 'none'
'none']
Classes: ['Center', 'Donut', 'Edge-Loc', 'Edge-Ring', 'Loc', 'Near-full',
'Random', 'Scratch', 'none']
num_classes: 9
Train: (10056, 26, 26, 3) (10056,)
Val:   (2155, 26, 26, 3) (2155,)
Test:  (2155, 26, 26, 3) (2155,)
```

Model: "wafer_autoencoder"

Layer (type)	Output Shape	Param #

input_layer_3 (InputLayer)	(None, 26, 26, 3)	0
conv2d_8 (Conv2D)	(None, 26, 26, 32)	896
conv2d_9 (Conv2D)	(None, 26, 26, 16)	4,624
encoded (Conv2D)	(None, 26, 26, 8)	1,160
conv2d_10 (Conv2D)	(None, 26, 26, 16)	1,168
conv2d_11 (Conv2D)	(None, 26, 26, 32)	4,640
conv2d_12 (Conv2D)	(None, 26, 26, 3)	867

Total params: 13,355 (52.17 KB)

Trainable params: 13,355 (52.17 KB)

Non-trainable params: 0 (0.00 B)

```
Training autoencoder for data augmentation…
Epoch 1/20
79/79              34s 375ms/step -
loss: 0.1597
Epoch 2/20
79/79              29s 367ms/step -
loss: 0.0314
Epoch 3/20
79/79              29s 369ms/step -
loss: 0.0013
Epoch 4/20
79/79              41s 525ms/step -
loss: 2.5550e-04
Epoch 5/20
79/79              70s 373ms/step -
loss: 9.5458e-05
Epoch 6/20
79/79              41s 367ms/step -
loss: 4.8237e-05
Epoch 7/20
79/79              28s 359ms/step -
loss: 2.8932e-05
Epoch 8/20
79/79              30s 378ms/step -
```

```
loss: 1.9621e-05
Epoch 9/20
79/79              28s 355ms/step -
loss: 1.4255e-05
Epoch 10/20
79/79              41s 356ms/step -
loss: 1.0716e-05
Epoch 11/20
79/79              41s 357ms/step -
loss: 8.3279e-06
Epoch 12/20
79/79              28s 359ms/step -
loss: 6.7344e-06
Epoch 13/20
79/79              28s 357ms/step -
loss: 5.6042e-06
Epoch 14/20
79/79              28s 358ms/step -
loss: 4.6784e-06
Epoch 15/20
79/79              28s 356ms/step -
loss: 3.9456e-06
Epoch 16/20
79/79              41s 357ms/step -
loss: 3.4027e-06
Epoch 17/20
79/79              28s 358ms/step -
loss: 2.9087e-06
Epoch 18/20
79/79              41s 356ms/step -
loss: 2.5400e-06
Epoch 19/20
79/79              28s 355ms/step -
loss: 2.2323e-06
Epoch 20/20
79/79              41s 356ms/step -
loss: 2.1440e-06
Autoencoder training complete.

Train class counts:
  Center (label 0): 68
  Donut (label 1): 1
  Edge-Loc (label 2): 210
  Edge-Ring (label 3): 22
  Loc (label 4): 224
  Near-full (label 5): 14
  Random (label 6): 56
  Scratch (label 7): 50
```

```
  none (label 8): 9411

Rare classes selected for AE augmentation (count < 100):
  Center (label 0)
  Donut (label 1)
  Edge-Ring (label 3)
  Near-full (label 5)
  Random (label 6)
  Scratch (label 7)

Augmenting class Center (n=68) with 3x AE reconstructions…

Augmenting class Donut (n=1) with 3x AE reconstructions…

Augmenting class Edge-Ring (n=22) with 3x AE reconstructions…

Augmenting class Near-full (n=14) with 3x AE reconstructions…

Augmenting class Random (n=56) with 3x AE reconstructions…

Augmenting class Scratch (n=50) with 3x AE reconstructions…

Augmented train shape: (10689, 26, 26, 3) (10689,)

Balanced val shape: (164, 26, 26, 3) (164,)
Balanced val class counts:
  Center: 14
  Edge-Loc: 40
  Edge-Ring: 4
  Loc: 39
  Near-full: 1
  Random: 14
  Scratch: 12
  none: 40

Class weights (after AE augmentation):
  Center: 4.366
  Donut: 296.917
  Edge-Loc: 5.656
  Edge-Ring: 13.496
  Loc: 5.302
  Near-full: 21.208
  Random: 5.302
  Scratch: 5.938
  none: 0.126

CNN input_shape: (26, 26, 3)
```

Model: "wafer_cnn_ae_aug"

Layer (type)	Output Shape	Param #
input_layer_4 (InputLayer)	(None, 26, 26, 3)	0
conv2d_13 (Conv2D)	(None, 26, 26, 8)	224
batch_normalization (BatchNormalization)	(None, 26, 26, 8)	32
max_pooling2d_2 (MaxPooling2D)	(None, 13, 13, 8)	0
conv2d_14 (Conv2D)	(None, 13, 13, 16)	1,168
batch_normalization_1 (BatchNormalization)	(None, 13, 13, 16)	64
max_pooling2d_3 (MaxPooling2D)	(None, 6, 6, 16)	0
conv2d_15 (Conv2D)	(None, 6, 6, 32)	4,640
batch_normalization_2 (BatchNormalization)	(None, 6, 6, 32)	128
max_pooling2d_4 (MaxPooling2D)	(None, 3, 3, 32)	0
dropout (Dropout)	(None, 3, 3, 32)	0
flatten (Flatten)	(None, 288)	0
dense (Dense)	(None, 64)	18,496
dropout_1 (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 9)	585

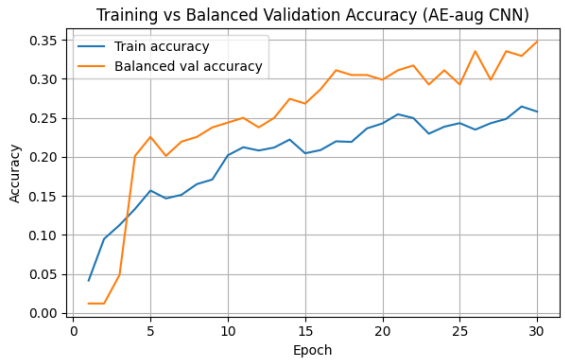
Total params: 25,337 (98.97 KB)

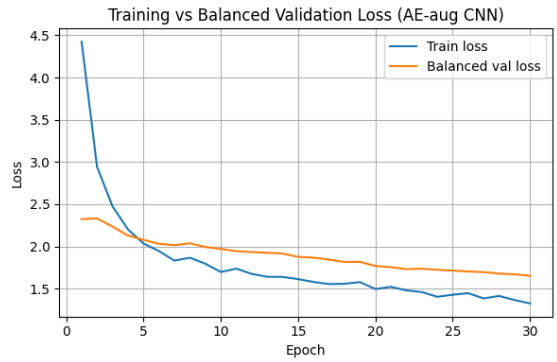
Trainable params: 25,225 (98.54 KB)

Non-trainable params: 112 (448.00 B)

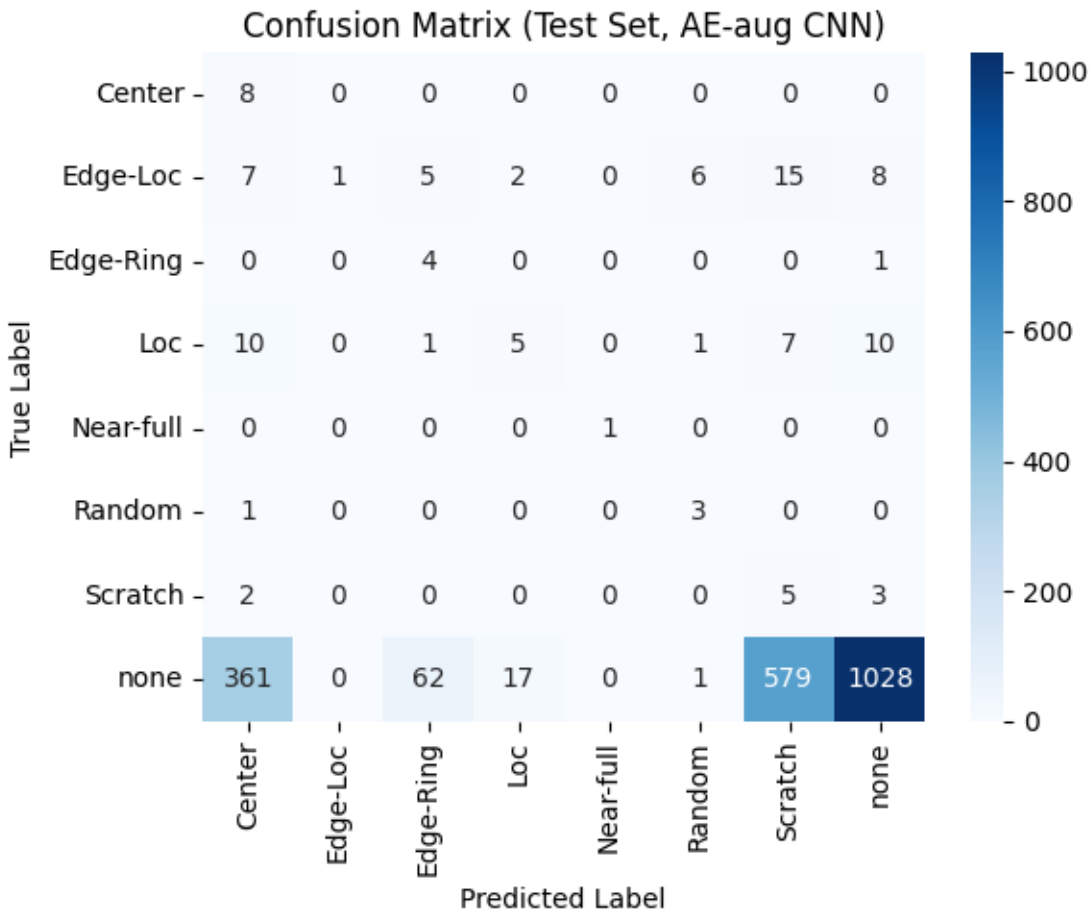
```
Epoch 1/30
84/84              9s 87ms/step -
accuracy: 0.0231 - loss: 3.6254 - val_accuracy: 0.0122 - val_loss: 2.3243
Epoch 2/30
84/84              5s 61ms/step -
accuracy: 0.0790 - loss: 1.9491 - val_accuracy: 0.0122 - val_loss: 2.3304
Epoch 3/30
84/84              11s 70ms/step -
accuracy: 0.0999 - loss: 1.6881 - val_accuracy: 0.0488 - val_loss: 2.2348
Epoch 4/30
84/84              5s 62ms/step -
accuracy: 0.1145 - loss: 1.4805 - val_accuracy: 0.2012 - val_loss: 2.1287
Epoch 5/30
84/84              7s 81ms/step -
accuracy: 0.1393 - loss: 1.3632 - val_accuracy: 0.2256 - val_loss: 2.0785
Epoch 6/30
84/84              9s 70ms/step -
accuracy: 0.1295 - loss: 1.3582 - val_accuracy: 0.2012 - val_loss: 2.0309
Epoch 7/30
84/84              6s 72ms/step -
accuracy: 0.1298 - loss: 1.2836 - val_accuracy: 0.2195 - val_loss: 2.0153
Epoch 8/30
84/84              5s 62ms/step -
accuracy: 0.1530 - loss: 1.2569 - val_accuracy: 0.2256 - val_loss: 2.0371
Epoch 9/30
84/84              7s 80ms/step -
accuracy: 0.1512 - loss: 1.2889 - val_accuracy: 0.2378 - val_loss: 1.9935
Epoch 10/30
84/84              5s 62ms/step -
accuracy: 0.1782 - loss: 1.1718 - val_accuracy: 0.2439 - val_loss: 1.9696
Epoch 11/30
84/84              7s 80ms/step -
accuracy: 0.1942 - loss: 1.1750 - val_accuracy: 0.2500 - val_loss: 1.9433
Epoch 12/30
84/84              5s 62ms/step -
accuracy: 0.1852 - loss: 1.1593 - val_accuracy: 0.2378 - val_loss: 1.9331
Epoch 13/30
84/84              11s 74ms/step -
accuracy: 0.1961 - loss: 1.1701 - val_accuracy: 0.2500 - val_loss: 1.9249
Epoch 14/30
84/84              11s 81ms/step -
accuracy: 0.2066 - loss: 1.1453 - val_accuracy: 0.2744 - val_loss: 1.9155
Epoch 15/30
84/84              5s 62ms/step -
accuracy: 0.1853 - loss: 1.1294 - val_accuracy: 0.2683 - val_loss: 1.8765
Epoch 16/30
84/84              6s 74ms/step -
accuracy: 0.1882 - loss: 1.1407 - val_accuracy: 0.2866 - val_loss: 1.8672
```

```
Epoch 17/30
84/84              6s 70ms/step -
accuracy: 0.2003 - loss: 1.0947 - val_accuracy: 0.3110 - val_loss: 1.8445
Epoch 18/30
84/84              11s 80ms/step -
accuracy: 0.1994 - loss: 1.0863 - val_accuracy: 0.3049 - val_loss: 1.8155
Epoch 19/30
84/84              5s 62ms/step -
accuracy: 0.2185 - loss: 1.1242 - val_accuracy: 0.3049 - val_loss: 1.8172
Epoch 20/30
84/84              7s 79ms/step -
accuracy: 0.2257 - loss: 1.0692 - val_accuracy: 0.2988 - val_loss: 1.7678
Epoch 21/30
84/84              5s 63ms/step -
accuracy: 0.2323 - loss: 1.0946 - val_accuracy: 0.3110 - val_loss: 1.7545
Epoch 22/30
84/84              5s 64ms/step -
accuracy: 0.2217 - loss: 1.0608 - val_accuracy: 0.3171 - val_loss: 1.7317
Epoch 23/30
84/84              10s 62ms/step -
accuracy: 0.2025 - loss: 1.0855 - val_accuracy: 0.2927 - val_loss: 1.7358
Epoch 24/30
84/84              7s 82ms/step -
accuracy: 0.2148 - loss: 1.0145 - val_accuracy: 0.3110 - val_loss: 1.7238
Epoch 25/30
84/84              5s 63ms/step -
accuracy: 0.2153 - loss: 1.0040 - val_accuracy: 0.2927 - val_loss: 1.7134
Epoch 26/30
84/84              11s 68ms/step -
accuracy: 0.2085 - loss: 1.0713 - val_accuracy: 0.3354 - val_loss: 1.7032
Epoch 27/30
84/84              5s 62ms/step -
accuracy: 0.2092 - loss: 0.9852 - val_accuracy: 0.2988 - val_loss: 1.6961
Epoch 28/30
84/84              10s 63ms/step -
accuracy: 0.2120 - loss: 1.0054 - val_accuracy: 0.3354 - val_loss: 1.6782
Epoch 29/30
84/84              7s 81ms/step -
accuracy: 0.2315 - loss: 1.0048 - val_accuracy: 0.3293 - val_loss: 1.6697
Epoch 30/30
84/84              5s 62ms/step -
accuracy: 0.2179 - loss: 1.0066 - val_accuracy: 0.3476 - val_loss: 1.6517
```






```
Evaluating on ORIGINAL (imbalanced) test set…
17/17              1s 36ms/step -
accuracy: 0.4821 - loss: 1.5664
Test loss: 1.5657  |  Test accuracy: 0.4896
17/17              1s 41ms/step

Classification Report (original test set):
              precision    recall  f1-score   support

      Center       0.02      1.00      0.04         8
    Edge-Loc       1.00      0.02      0.04        45
   Edge-Ring       0.06      0.80      0.10         5
         Loc       0.21      0.15      0.17        34
   Near-full       1.00      1.00      1.00         1
      Random       0.27      0.75      0.40         4
     Scratch       0.01      0.50      0.02        10
        none       0.98      0.50      0.66      2048

   micro avg       0.49      0.49      0.49      2155
   macro avg       0.44      0.59      0.30      2155
weighted avg       0.96      0.49      0.64      2155
```




```
Per-class accuracy on test set:
Center: 100.00% (n=8)
Donut: N/A (no samples in test set)
Edge-Loc: 2.22% (n=45)
Edge-Ring: 80.00% (n=5)
Loc: 14.71% (n=34)
Near-full: 100.00% (n=1)
Random: 75.00% (n=4)
Scratch: 50.00% (n=10)
none: 50.20% (n=2048)
```

[ ]:
```python
# Transfer Learning (MobileNetV2) Model
# Importing the Necessary Modules
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras
```

```python
from keras import layers
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report
from keras.applications import MobileNetV2
from keras.applications.mobilenet_v2 import preprocess_input

print("TensorFlow version:", tf.__version__)  # Printing the Tensorflow Version

# Loading the Dataset from the Pickle File:
from google.colab import drive
drive.mount('/content/drive')

DATA_PKL = "/content/waferImg26x26.pkl"

df = pd.read_pickle(DATA_PKL)
print("DataFrame shape:", df.shape)
print(df.head())

# Extracting the Images and the Labels
X_list = df["images"].to_numpy()
X = np.stack(X_list).astype("float32")
print("Raw X shape:", X.shape)

# Checking if shape is (N, C, H, W), transpose to (N, H, W, C)
if X.ndim == 4 and X.shape[1] in (1, 3):
    X = np.transpose(X, (0, 2, 3, 1))
    print("Transposed to NHWC:", X.shape)

# Ensuring correct channel dimension exists
if X.ndim == 3:
    X = X[..., np.newaxis]
    print("Added channel dimension:", X.shape)

IMG_SIZE = X.shape[1]
channels = X.shape[-1]
print("IMG_SIZE:", IMG_SIZE, "channels:", channels)

# Normalizing Pixel Intensity Values to [0,1]
if X.max() > 1.0:
    X = X / 255.0

# Cleaning the Image Labels
labels_raw = df["labels"].to_numpy()
labels_clean = []
for v in labels_raw:
```

```
    if isinstance(v, (list, np.ndarray)) and len(v) > 0:
        labels_clean.append(v[0])
    else:
        labels_clean.append(v)

y_str = np.array(labels_clean)
print("Example raw labels:", y_str[:10])

# Encoding the string labels to integers
unique_labels, y_int = np.unique(y_str, return_inverse=True)
class_names = [str(lbl) for lbl in unique_labels]
y = y_int.astype("int32")
num_classes = len(class_names)
print("Classes:", class_names)
print("num_classes:", num_classes)

# Splitting the Dataset into Training, Testing and Validation Datasets
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.30,␣
 ↪random_state=42, shuffle=True)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.50,␣
 ↪random_state=42, shuffle=True)
print("Train:", X_train.shape, y_train.shape)
print("Val:  ", X_val.shape,   y_val.shape)
print("Test: ", X_test.shape,  y_test.shape)

# Formatting the Images to input them into the Transfer Learning Model
BATCH_SIZE = 64
TL_IMG_SIZE = 96
print("Original train image shape:", X_train.shape[1:])

# Creatining the Modifed Dataset to Input into the Model
def make_tl_dataset(X, y, shuffle=False):
    ds = tf.data.Dataset.from_tensor_slices((X, y))
    if shuffle:
        ds = ds.shuffle(2000)

    def _preprocess(x, label):
        x = tf.cast(x, tf.float32)
        if channels == 1:
            x = tf.tile(x, [1, 1, 3])
        x = tf.image.resize(x, (TL_IMG_SIZE, TL_IMG_SIZE))
        x = preprocess_input(x)
        return x, label

    ds = ds.map(_preprocess, num_parallel_calls=tf.data.AUTOTUNE)
    ds = ds.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
    return ds
```

```python
# Create training, validation, and test datasets
train_tl_ds = make_tl_dataset(X_train, y_train, shuffle=True)
val_tl_ds   = make_tl_dataset(X_val,   y_val,   shuffle=False)
test_tl_ds  = make_tl_dataset(X_test,  y_test,  shuffle=False)

# Creating the Transfer Learning Model
# Freezing the Base Model
base_model = MobileNetV2(input_shape=(TL_IMG_SIZE, TL_IMG_SIZE, 3),␣
 ↪include_top=False,weights="imagenet")
base_model.trainable = False

# Defining the Classification Head of the Model
inputs = keras.Input(shape=(TL_IMG_SIZE, TL_IMG_SIZE, 3))
x = base_model(inputs, training=False)
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dropout(0.3)(x)
outputs = layers.Dense(num_classes, activation="softmax")(x)

tl_model = keras.Model(inputs, outputs, name="wafer_mobilenetv2_tl")
tl_model.summary()

# Compiling and Training the Transfer Learning Model
tl_model.compile(optimizer=keras.optimizers.Adam(learning_rate=1e-3),␣
 ↪loss="sparse_categorical_crossentropy", metrics=["accuracy"],)
early_stop = keras.callbacks.EarlyStopping(monitor="val_loss",␣
 ↪patience=5,restore_best_weights=True)

print("\n=== Stage 1: Training feature extractor (frozen base) ===")
history_tl = tl_model.fit(train_tl_ds, epochs=20, validation_data=val_tl_ds,␣
 ↪callbacks=[early_stop], verbose=1)

# Unfreezing the top layer of the model to fine-tune the performance:
base_model.trainable = True
fine_tune_at = 100 # setting the layer from which the unfreezing should start

for i, layer in enumerate(base_model.layers):
    if i < fine_tune_at:
        layer.trainable = False
    else:
        layer.trainable = True

# Recompiling with a lower learning rate
tl_model.compile(optimizer=keras.optimizers.Adam(learning_rate=1e-5),␣
 ↪loss="sparse_categorical_crossentropy", metrics=["accuracy"],)

print("\n=== Stage 2: Fine-tuning top of MobileNetV2 ===")
```

```python
history_ft = tl_model.fit(train_tl_ds, epochs=20, validation_data=val_tl_ds,␣
↪callbacks=[early_stop], verbose=1)

# Plotting the Training accuracy and Loss Plots of the Model
def plot_history(hist, title_prefix=""):
    h = hist.history
    acc = h.get("accuracy", h.get("acc"))
    val_acc = h.get("val_accuracy", h.get("val_acc"))
    loss = h["loss"]
    val_loss = h["val_loss"]
    epochs = range(1, len(loss) + 1)

    plt.figure(figsize=(12, 4))

    plt.subplot(1, 2, 1)
    plt.plot(epochs, acc, label="Train acc")
    plt.plot(epochs, val_acc, label="Val acc")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.title(f"{title_prefix} Accuracy")
    plt.legend()
    plt.grid(True)

    plt.subplot(1, 2, 2)
    plt.plot(epochs, loss, label="Train loss")
    plt.plot(epochs, val_loss, label="Val loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title(f"{title_prefix} Loss")
    plt.legend()
    plt.grid(True)

    plt.tight_layout()
    plt.show()

plot_history(history_tl, "Stage 1 (Frozen)")
plot_history(history_ft, "Stage 2 (Fine-tuning)")

# Evaluting the Model Performance on the Original, Imbalanced Dataset
print("\n=== Evaluation on FULL test set ===")
test_loss, test_acc = tl_model.evaluate(test_tl_ds, verbose=1)
print(f"Test loss: {test_loss:.4f}  |  Test accuracy: {test_acc:.4f}")

# Getting the Model's Image Predictions
y_pred_probs = tl_model.predict(test_tl_ds)
y_pred = np.argmax(y_pred_probs, axis=1)
y_true = y_test
```

```
labels_present = np.unique(y_true)
target_names_present = [class_names[i] for i in labels_present]

# Generating the Classification Report for the Model
print("\nClassification Report (FULL test set):")
print(classification_report(y_true, y_pred, labels=labels_present,␣
 ↪target_names=target_names_present, zero_division=0))

# Generating a Confusion matrix
cm = confusion_matrix(y_true, y_pred, labels=labels_present)
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",␣
 ↪xticklabels=target_names_present, yticklabels=target_names_present)
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix - FULL Test Set")
plt.tight_layout()
plt.show()
```

```
TensorFlow version: 2.19.0
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
DataFrame shape: (14366, 2)
                                                  images  labels
0  [[[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0…  [none]
1  [[[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0…  [none]
2  [[[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0…  [none]
3  [[[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0…  [none]
4  [[[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0…  [none]
Raw X shape: (14366, 3, 26, 26)
Transposed to NHWC: (14366, 26, 26, 3)
IMG_SIZE: 26 channels: 3
Example raw labels: ['none' 'none' 'none' 'none' 'none' 'none' 'none' 'none'
'none' 'none']
Classes: ['Center', 'Donut', 'Edge-Loc', 'Edge-Ring', 'Loc', 'Near-full',
'Random', 'Scratch', 'none']
num_classes: 9
Train: (10056, 26, 26, 3) (10056,)
Val:   (2155, 26, 26, 3) (2155,)
Test:  (2155, 26, 26, 3) (2155,)
Original train image shape: (26, 26, 3)
```

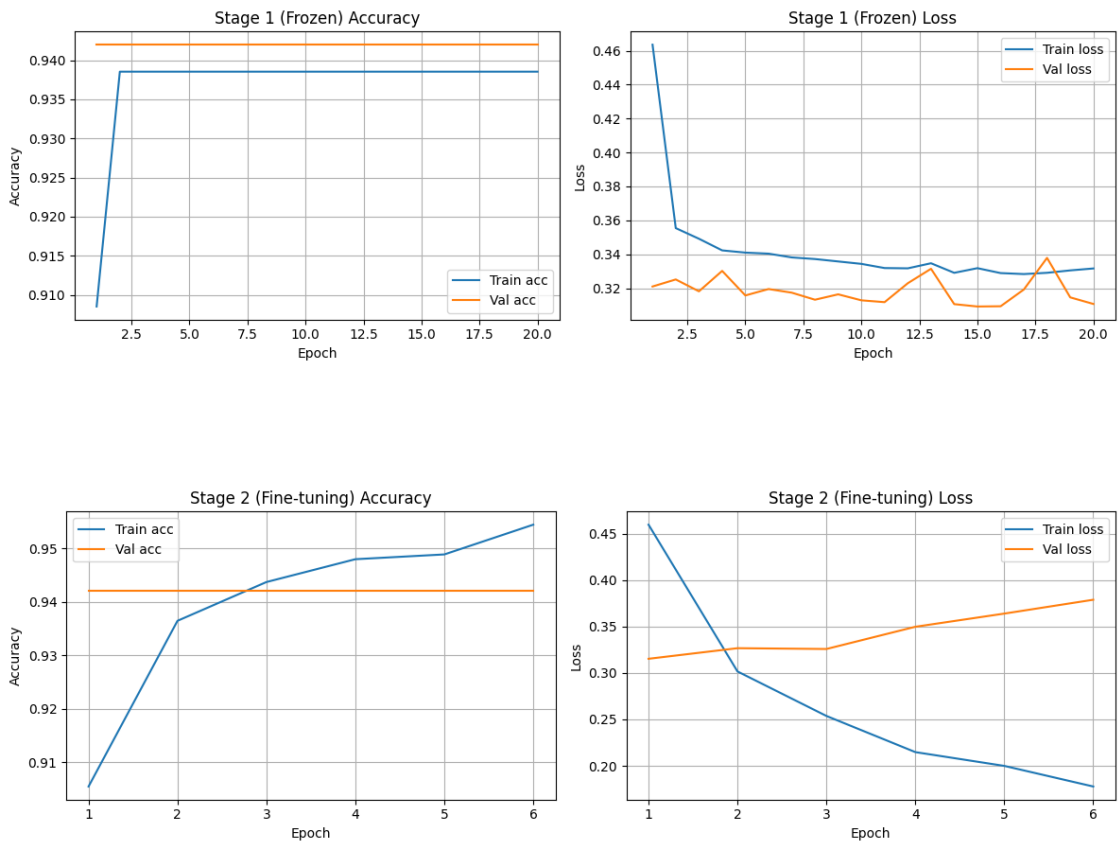
Model: "wafer_mobilenetv2_tl"

Layer (type)	Output Shape	Param #

```
 input_layer_27 (InputLayer)      (None, 96, 96, 3)                  0

 mobilenetv2_1.00_96              (None, 3, 3, 1280)          2,257,984
 (Functional)

 global_average_pooling2d_6       (None, 1280)                       0
 (GlobalAveragePooling2D)

 dropout_16 (Dropout)             (None, 1280)                       0

 dense_28 (Dense)                 (None, 9)                     11,529


 Total params: 2,269,513 (8.66 MB)


 Trainable params: 11,529 (45.04 KB)


 Non-trainable params: 2,257,984 (8.61 MB)


=== Stage 1: Training feature extractor (frozen base) ===
Epoch 1/20
158/158            94s 552ms/step -
accuracy: 0.8167 - loss: 0.7421 - val_accuracy: 0.9420 - val_loss: 0.3210
Epoch 2/20
158/158            145s 573ms/step -
accuracy: 0.9448 - loss: 0.3266 - val_accuracy: 0.9420 - val_loss: 0.3252
Epoch 3/20
158/158            141s 567ms/step -
accuracy: 0.9443 - loss: 0.3268 - val_accuracy: 0.9420 - val_loss: 0.3182
Epoch 4/20
158/158            89s 565ms/step -
accuracy: 0.9439 - loss: 0.3168 - val_accuracy: 0.9420 - val_loss: 0.3302
Epoch 5/20
158/158            90s 568ms/step -
accuracy: 0.9432 - loss: 0.3178 - val_accuracy: 0.9420 - val_loss: 0.3157
Epoch 6/20
158/158            137s 535ms/step -
accuracy: 0.9427 - loss: 0.3162 - val_accuracy: 0.9420 - val_loss: 0.3195
Epoch 7/20
158/158            89s 563ms/step -
accuracy: 0.9445 - loss: 0.3102 - val_accuracy: 0.9420 - val_loss: 0.3174
Epoch 8/20
158/158            84s 533ms/step -
accuracy: 0.9450 - loss: 0.3097 - val_accuracy: 0.9420 - val_loss: 0.3132
Epoch 9/20
```

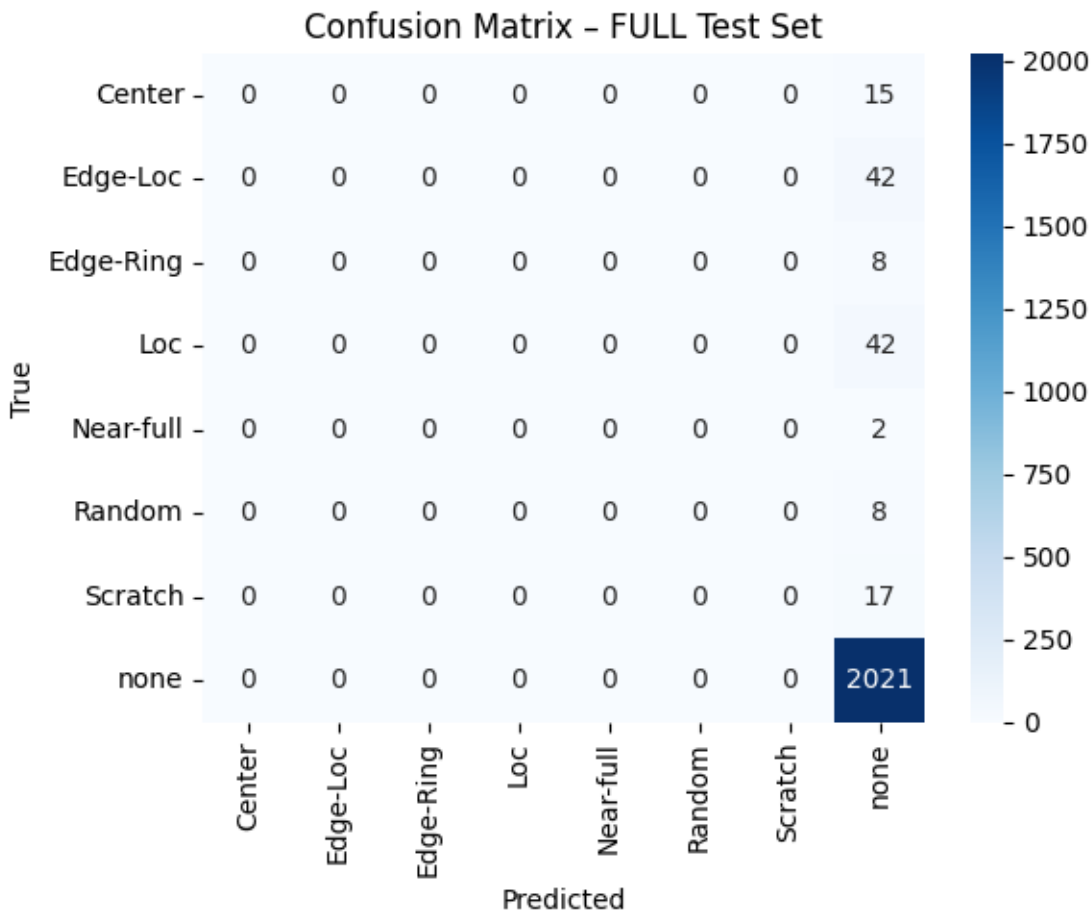
```
158/158              148s 569ms/step -
accuracy: 0.9418 - loss: 0.3186 - val_accuracy: 0.9420 - val_loss: 0.3164
Epoch 10/20
158/158              90s 568ms/step -
accuracy: 0.9408 - loss: 0.3246 - val_accuracy: 0.9420 - val_loss: 0.3129
Epoch 11/20
158/158              142s 569ms/step -
accuracy: 0.9450 - loss: 0.3026 - val_accuracy: 0.9420 - val_loss: 0.3118
Epoch 12/20
158/158              90s 571ms/step -
accuracy: 0.9437 - loss: 0.3087 - val_accuracy: 0.9420 - val_loss: 0.3229
Epoch 13/20
158/158              90s 570ms/step -
accuracy: 0.9432 - loss: 0.3136 - val_accuracy: 0.9420 - val_loss: 0.3315
Epoch 14/20
158/158              84s 530ms/step -
accuracy: 0.9457 - loss: 0.2976 - val_accuracy: 0.9420 - val_loss: 0.3106
Epoch 15/20
158/158              149s 577ms/step -
accuracy: 0.9435 - loss: 0.3109 - val_accuracy: 0.9420 - val_loss: 0.3092
Epoch 16/20
158/158              136s 539ms/step -
accuracy: 0.9432 - loss: 0.3066 - val_accuracy: 0.9420 - val_loss: 0.3094
Epoch 17/20
158/158              149s 582ms/step -
accuracy: 0.9459 - loss: 0.2947 - val_accuracy: 0.9420 - val_loss: 0.3192
Epoch 18/20
158/158              90s 573ms/step -
accuracy: 0.9459 - loss: 0.2971 - val_accuracy: 0.9420 - val_loss: 0.3379
Epoch 19/20
158/158              142s 570ms/step -
accuracy: 0.9436 - loss: 0.3144 - val_accuracy: 0.9420 - val_loss: 0.3146
Epoch 20/20
158/158              136s 533ms/step -
accuracy: 0.9442 - loss: 0.3040 - val_accuracy: 0.9420 - val_loss: 0.3107

=== Stage 2: Fine-tuning top of MobileNetV2 ===
Epoch 1/20
158/158              180s 1s/step -
accuracy: 0.8670 - loss: 0.5907 - val_accuracy: 0.9420 - val_loss: 0.3153
Epoch 2/20
158/158              197s 991ms/step -
accuracy: 0.9399 - loss: 0.3052 - val_accuracy: 0.9420 - val_loss: 0.3267
Epoch 3/20
158/158              204s 1s/step -
accuracy: 0.9462 - loss: 0.2592 - val_accuracy: 0.9420 - val_loss: 0.3258
Epoch 4/20
158/158              217s 1s/step -
```

```
accuracy: 0.9495 - loss: 0.2114 - val_accuracy: 0.9420 - val_loss: 0.3498
Epoch 5/20
158/158              177s 940ms/step -
accuracy: 0.9474 - loss: 0.2068 - val_accuracy: 0.9420 - val_loss: 0.3639
Epoch 6/20
158/158              208s 980ms/step -
accuracy: 0.9570 - loss: 0.1745 - val_accuracy: 0.9420 - val_loss: 0.3788
```




```
=== Evaluation on FULL test set ===
34/34              16s 452ms/step -
accuracy: 0.9364 - loss: 0.3445
Test loss: 0.3409  |  Test accuracy: 0.9378
34/34              18s 480ms/step

Classification Report (FULL test set):
              precision    recall  f1-score   support

      Center       0.00      0.00      0.00        15
    Edge-Loc       0.00      0.00      0.00        42
   Edge-Ring       0.00      0.00      0.00         8
```

```
          Loc       0.00      0.00      0.00        42
    Near-full       0.00      0.00      0.00         2
       Random       0.00      0.00      0.00         8
      Scratch       0.00      0.00      0.00        17
         none       0.94      1.00      0.97      2021

     accuracy                           0.94      2155
    macro avg       0.12      0.12      0.12      2155
weighted avg       0.88      0.94      0.91      2155
```



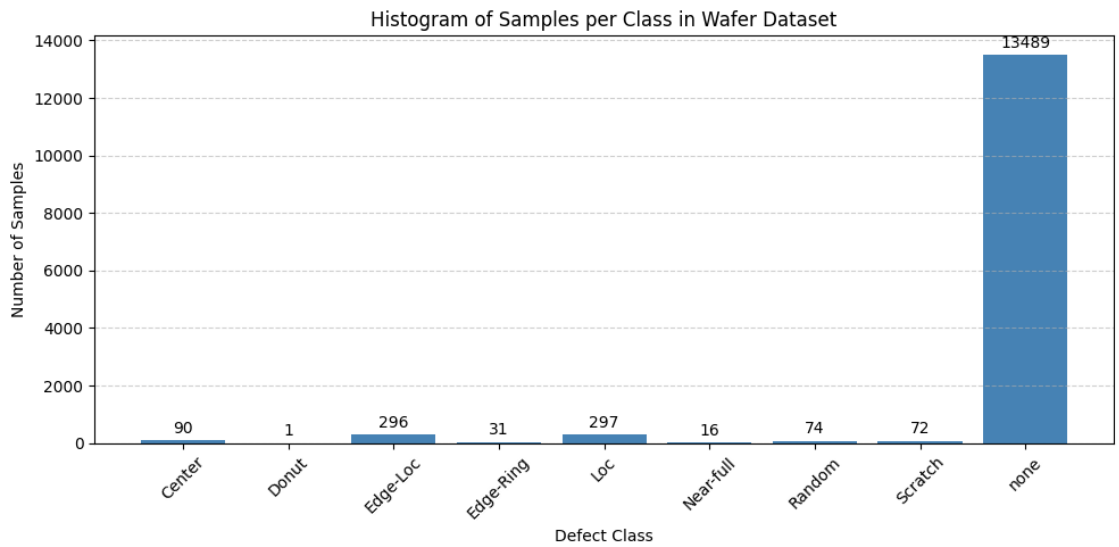
[3]:
```
# Code for Getting the Data Distribution after Splitting on Raw Data

# Importing the Necessary Modules
import numpy as np
import pandas as pd
from collections import Counter
import matplotlib.pyplot as plt
```

```python
# Loading the data from the Pickle File
DATA_PKL = "/content/waferImg26x26.pkl"

df = pd.read_pickle(DATA_PKL)

# Extracting Images and their Labels
X_list = df["images"].to_numpy()
X = np.stack(X_list).astype("float32")

labels_raw = df["labels"].to_numpy()

labels_clean = []
for v in labels_raw:
    if isinstance(v, (list, np.ndarray)) and len(v) > 0:
        labels_clean.append(v[0])
    else:
        labels_clean.append(v)

y_str = np.array(labels_clean)
print("Example labels:", y_str[:10])

# Encoding the string labels to integers
class_names = ["Center", "Donut", "Edge-Loc", "Edge-Ring", "Loc", "Near-full",␣
 ↪"Random", "Scratch", "none"]
label_to_int = {name: idx for idx, name in enumerate(class_names)}

# Converting string labels to integer codes
y = np.array([label_to_int[str(lbl)] for lbl in y_str], dtype=int)

# Split the Dataset into Training, Validation, and Testing sets
N = len(X)
rng = np.random.default_rng(seed=42)
perm = rng.permutation(N)

train_end = int(0.7 * N)
val_end   = int(0.85 * N)

idx_train = perm[:train_end]
idx_val   = perm[train_end:val_end]
idx_test  = perm[val_end:]

y_train = y[idx_train]
y_val   = y[idx_val]
y_test  = y[idx_test]

print("Train size:", len(y_train))
print("Val size:  ", len(y_val))
```

```python
print("Test size: ", len(y_test))

# Obtaining the number of samples in each dataset
train_counts = Counter(y_train)
val_counts   = Counter(y_val)
test_counts  = Counter(y_test)

# Creating a table containing the number of samples in each dataset, in each␣
↪class
rows = []
for idx, cls_name in enumerate(class_names):
    rows.append({"Defect": cls_name, "Training Set":  train_counts.get(idx, 0),␣
↪"Validation Set": val_counts.get(idx, 0), "Test Set": test_counts.get(idx,␣
↪0),})

df_table = pd.DataFrame(rows, columns=["Defect", "Training Set", "Validation␣
↪Set", "Test Set"])

print("\nTable 1. Raw Data Set Distribution\n")
print(df_table.to_string(index=False))

# Saving the table as a CSV file
df_table.to_csv("Raw_Data_Distribution.csv", index=False)
print("\nSaved table as Raw_Data_Distribution.csv")

# Creating a Histogram of the number of samples per class:
# Finding the number of samples per class
unique_classes, counts = np.unique(labels_clean, return_counts=True)

# Plotting the Histogram as a Bar Chart:
plt.figure(figsize=(10, 5))
plt.bar(unique_classes, counts, color='steelblue')
plt.xlabel("Defect Class")
plt.ylabel("Number of Samples")
plt.title("Histogram of Samples per Class in Wafer Dataset")
plt.xticks(rotation=45)
plt.grid(axis='y', linestyle='--', alpha=0.6)

# Displaying sample counts above each bar
for i, count in enumerate(counts):
    plt.text(i, count + max(counts)*0.01, str(count), ha='center', va='bottom')

plt.tight_layout()
plt.show()
```

```
Example labels: ['none' 'none' 'none' 'none' 'none' 'none' 'none' 'none' 'none'
'none']
```

```
Train size: 10056
Val size:   2155
Test size:  2155

Table 1. Raw Data Set Distribution

   Defect  Training Set  Validation Set  Test Set
   Center            68              14         8
    Donut             1               0         0
 Edge-Loc           210              41        45
Edge-Ring            22               4         5
      Loc           224              39        34
Near-full            14               1         1
   Random            56              14         4
  Scratch            50              12        10
     none          9411            2030      2048

Saved table as Raw_Data_Distribution.csv
```



[7]:
```python
# Code for Getting the Data Distribution after Splitting for␣
↪Autoencoding-Enhanced Data
# Importing the necessary modules
import numpy as np
import pandas as pd
from collections import Counter
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras
```

```python
from keras import layers, regularizers
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.utils.class_weight import compute_class_weight

# Loading the Dataset from Drive
from google.colab import drive
drive.mount('/content/drive')

DATA_PKL = "/content/waferImg26x26.pkl"

df = pd.read_pickle(DATA_PKL)
print("Loaded object type:", type(df))
print(df.head())

# Extracting the images and their labels
X_list = df["images"].to_numpy()
X = np.stack(X_list).astype("float32")
print("Raw X shape:", X.shape)

# Checking if shape is (N, C, H, W), transpose to (N, H, W, C)
if X.ndim == 4 and X.shape[1] in (1, 3):
    X = np.transpose(X, (0, 2, 3, 1))
    print("Transposed X shape (NHWC):", X.shape)

# Ensuring correct channel dimension exists
if X.ndim == 3:
    X = X[..., np.newaxis]
    print("Added channel dim:", X.shape)
else:
    print("X already has channels:", X.shape)

IMG_SIZE = X.shape[1]
channels = X.shape[-1]
print("IMG_SIZE:", IMG_SIZE, "channels:", channels)

# Normalizing Pixel Intensity Values to [0,1]
if X.max() > 1.0:
    X = X / 255.0
print("X range:", X.min(), "to", X.max())

# Cleaning the Image Labels
labels_raw = df["labels"].to_numpy()

labels_clean = []
for v in labels_raw:
```

```
    if isinstance(v, (list, np.ndarray)) and len(v) > 0:
        labels_clean.append(v[0])
    else:
        labels_clean.append(v)

y_str = np.array(labels_clean)
print("Example labels:", y_str[:10])

# Encoding the string labels to integers
unique_labels, y_int = np.unique(y_str, return_inverse=True)
class_names = [str(lbl) for lbl in unique_labels]
y = y_int.astype("int32")
num_classes = len(class_names)

print("Classes:", class_names)
print("num_classes:", num_classes)

# Splitting the Dataset into Training, Testing and Validation Datasets
N = len(X)
rng = np.random.default_rng(seed=42)
perm = rng.permutation(N)

train_end = int(0.7 * N)
val_end   = int(0.85 * N)

idx_train = perm[:train_end]
idx_val   = perm[train_end:val_end]
idx_test  = perm[val_end:]

X_train, y_train = X[idx_train], y[idx_train]
X_val,   y_val   = X[idx_val],   y[idx_val]
X_test,  y_test  = X[idx_test],  y[idx_test]

print("Train:", X_train.shape, y_train.shape)
print("Val:  ", X_val.shape,   y_val.shape)
print("Test: ", X_test.shape,  y_test.shape)

# Creating the autoencoder for data augmentation
# Building the dataset needing augmenting
BATCH_AE = 128
ae_train_ds = (tf.data.Dataset.from_tensor_slices((X_train, X_train)).
 ↪shuffle(1000).batch(BATCH_AE).prefetch(tf.data.AUTOTUNE))

# Defining the autoencoder:

ae_inputs = keras.Input(shape=(IMG_SIZE, IMG_SIZE, channels))
```

```
# Creating the encoder for the images
x = layers.Conv2D(32, 3, activation="relu", padding="same")(ae_inputs)
x = layers.Conv2D(16, 3, activation="relu", padding="same")(x)
encoded = layers.Conv2D(8, 3, activation="relu", padding="same",␣
 ↪name="encoded")(x)

# Creating the decoder for the images
x = layers.Conv2D(16, 3, activation="relu", padding="same")(encoded)
x = layers.Conv2D(32, 3, activation="relu", padding="same")(x)
ae_outputs = layers.Conv2D(channels, 3, activation="sigmoid", padding="same")(x)

# Constructing and compiling the final autoencoder
autoencoder = keras.Model(ae_inputs, ae_outputs, name="wafer_autoencoder")
autoencoder.compile(optimizer="adam", loss="mse")
autoencoder.summary()
print("\nTraining autoencoder for data augmentation...")
ae_history = autoencoder.fit(ae_train_ds,epochs=20)
print("Autoencoder training complete.")

# Applying the Autoencoder to the Minority Classes in the Dataset

#Identifying the Minority classes in the wafer dataset
classes_train, counts_train = np.unique(y_train, return_counts=True)
print("\nTrain class counts:")
for c, cnt in zip(classes_train, counts_train):
    print(f"  {class_names[c]} (label {c}): {cnt}")

# Setting the threshold used to identify the minority classes
min_count = 100
rare_labels = [c for c, cnt in zip(classes_train, counts_train) if cnt <␣
 ↪min_count]

print("\nRare classes selected for AE augmentation (count < "f"{min_count}):")
for c in rare_labels:
    print(f"  {class_names[c]} (label {c})")

# Reconstructing the Minority Class Images using the Autoencoder in Batches
def ae_generate_batch(images, batch_size=128):
    all_out = []
    for i in range(0, len(images), batch_size):
        batch = images[i:i+batch_size]
        recon = autoencoder.predict(batch, verbose=0)
        all_out.append(recon)
    return np.concatenate(all_out, axis=0)

# For each rare class, oversample with AE reconstructions
# Oversampling the autoencoder reconstructed images to reduce class bias
```

```
augmented_X_list = [X_train]
augmented_y_list = [y_train]

n_aug_per_sample = 3 # setting the number of synthetic samples to recreate for␣
↪each minority class in the original dataset

for c in rare_labels:
    idx_c = np.where(y_train == c)[0]
    X_c = X_train[idx_c]

    if len(X_c) == 0:
        continue

    print(f"\nAugmenting class {class_names[c]} (n={len(X_c)}) with␣
↪"f"{n_aug_per_sample}x AE reconstructions...")

    # Sampling each image 3 times then reconstruct
    X_c_repeat = np.repeat(X_c, n_aug_per_sample, axis=0)
    X_c_aug = ae_generate_batch(X_c_repeat)

    y_c_aug = np.full(len(X_c_aug), c, dtype="int32")

    augmented_X_list.append(X_c_aug)
    augmented_y_list.append(y_c_aug)

# Stacking the final, reconstructed images
X_train_aug = np.concatenate(augmented_X_list, axis=0)
y_train_aug = np.concatenate(augmented_y_list, axis=0)

print("\nAugmented train shape:", X_train_aug.shape, y_train_aug.shape)

# Balancing the Validation Dataset:
def make_balanced_subset(X, y, max_per_class=40):
    X_bal_list = []
    y_bal_list = []
    classes = np.unique(y)
    for c in classes:
        idx_c = np.where(y == c)[0]
        local_rng = np.random.default_rng(seed=123 + int(c))
        local_rng.shuffle(idx_c)
        chosen = idx_c[:max_per_class]
        X_bal_list.append(X[chosen])
        y_bal_list.append(y[chosen])
    return np.concatenate(X_bal_list, axis=0), np.concatenate(y_bal_list,␣
↪axis=0)

X_val_bal, y_val_bal = make_balanced_subset(X_val, y_val, max_per_class=40)
```

```
print("\nBalanced val shape:", X_val_bal.shape, y_val_bal.shape)
u_bal, c_bal = np.unique(y_val_bal, return_counts=True)
print("Balanced val class counts:")
for c, cnt in zip(u_bal, c_bal):
    print(f"  {class_names[c]}: {cnt}")

# Creating the Training, Validation, and Test Datasets
BATCH_SIZE = 128
train_ds = (tf.data.Dataset.from_tensor_slices((X_train_aug, y_train_aug)).
 ↪shuffle(2000).batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE))
val_ds_bal = (tf.data.Dataset.from_tensor_slices((X_val_bal, y_val_bal)).
 ↪batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE))
test_ds = (tf.data.Dataset.from_tensor_slices((X_test, y_test)).
 ↪batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE))

# Computing the Class weights after data augmentation was applied (helps gauge␣
 ↪whether the augmenation was successful at removing the imbalance)
classes_aug = np.unique(y_train_aug)
class_weights_arr = compute_class_weight(class_weight="balanced",␣
 ↪classes=classes_aug, y=y_train_aug)
class_weights = {int(c): float(w) for c, w in zip(classes_aug,␣
 ↪class_weights_arr)}
print("\nClass weights (after AE augmentation):")
for c, w in class_weights.items():
    print(f"  {class_names[c]}: {w:.3f}")

# Obtaining the dataset split information
def make_split_table(y_train, y_val, y_test, class_names, title="Data Set␣
 ↪Distribution"):

    train_counts = Counter(y_train)
    val_counts   = Counter(y_val)
    test_counts  = Counter(y_test)

    rows = []
    for cls_idx, cls_name in enumerate(class_names):
        rows.append({"Defect":cls_name, "Training Set": train_counts.
 ↪get(cls_idx, 0), "Validation Set": val_counts.get(cls_idx, 0), "Test Set":␣
 ↪test_counts.get(cls_idx, 0),})

    df_table = pd.DataFrame(rows, columns=["Defect", "Training Set",␣
 ↪"Validation Set", "Test Set"])

    print(f"\n{title}\n")
    print(df_table.to_string(index=False))
```

```
    return df_table

class_names = ["Center", "Donut", "Edge-Loc", "Edge-Ring", "Loc", "Near-full",␣
 ↪"Random", "Scratch", "none"]

# Creating a table containing the number of samples in each dataset, per class
df_aug_table = make_split_table(y_train=y_train_aug, y_val=y_val_bal,␣
 ↪y_test=y_test, class_names=class_names, title="Augmented CNN Data Set␣
 ↪Distribution")

# Saving the generated table as a CSV file
df_aug_table.to_csv("Augmented_Data_Distribution.csv", index=False)
print("\nSaved table as Augmented_Data_Distribution.csv")
```

```
Mounted at /content/drive
Loaded object type: <class 'pandas.core.frame.DataFrame'>
                                                 images  labels
0  [[[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0…  [none]
1  [[[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0…  [none]
2  [[[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0…  [none]
3  [[[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0…  [none]
4  [[[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0…  [none]
Raw X shape: (14366, 3, 26, 26)
Transposed X shape (NHWC): (14366, 26, 26, 3)
X already has channels: (14366, 26, 26, 3)
IMG_SIZE: 26 channels: 3
X range: 0.0 to 1.0
Example labels: ['none' 'none' 'none' 'none' 'none' 'none' 'none' 'none' 'none'
'none']
Classes: ['Center', 'Donut', 'Edge-Loc', 'Edge-Ring', 'Loc', 'Near-full',
'Random', 'Scratch', 'none']
num_classes: 9
Train: (10056, 26, 26, 3) (10056,)
Val:   (2155, 26, 26, 3) (2155,)
Test:  (2155, 26, 26, 3) (2155,)
```

Model: "wafer_autoencoder"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 26, 26, 3)	0
conv2d (Conv2D)	(None, 26, 26, 32)	896
conv2d_1 (Conv2D)	(None, 26, 26, 16)	4,624

```
 encoded (Conv2D)                (None, 26, 26, 8)              1,160

 conv2d_2 (Conv2D)               (None, 26, 26, 16)             1,168

 conv2d_3 (Conv2D)               (None, 26, 26, 32)             4,640

 conv2d_4 (Conv2D)               (None, 26, 26, 3)                867


 Total params: 13,355 (52.17 KB)

 Trainable params: 13,355 (52.17 KB)

 Non-trainable params: 0 (0.00 B)


Training autoencoder for data augmentation…
Epoch 1/20
79/79            32s 377ms/step -
loss: 0.1538
Epoch 2/20
79/79            30s 375ms/step -
loss: 0.0289
Epoch 3/20
79/79            41s 373ms/step -
loss: 8.6597e-04
Epoch 4/20
79/79            30s 374ms/step -
loss: 2.0020e-04
Epoch 5/20
79/79            31s 388ms/step -
loss: 6.4526e-05
Epoch 6/20
79/79            40s 374ms/step -
loss: 2.9870e-05
Epoch 7/20
79/79            30s 381ms/step -
loss: 1.7887e-05
Epoch 8/20
79/79            41s 378ms/step -
loss: 1.1898e-05
Epoch 9/20
79/79            41s 375ms/step -
loss: 9.0525e-06
Epoch 10/20
79/79            45s 432ms/step -
```

```
loss: 6.6769e-06
Epoch 11/20
79/79            30s 379ms/step -
loss: 5.1021e-06
Epoch 12/20
79/79            30s 375ms/step -
loss: 4.2058e-06
Epoch 13/20
79/79            30s 375ms/step -
loss: 3.5933e-06
Epoch 14/20
79/79            41s 375ms/step -
loss: 2.9512e-06
Epoch 15/20
79/79            42s 386ms/step -
loss: 2.5048e-06
Epoch 16/20
79/79            30s 378ms/step -
loss: 2.0566e-06
Epoch 17/20
79/79            53s 533ms/step -
loss: 1.7815e-06
Epoch 18/20
79/79            30s 376ms/step -
loss: 1.5454e-06
Epoch 19/20
79/79            30s 375ms/step -
loss: 1.2641e-06
Epoch 20/20
79/79            29s 374ms/step -
loss: 1.0894e-06
Autoencoder training complete.

Train class counts:
  Center (label 0): 68
  Donut (label 1): 1
  Edge-Loc (label 2): 210
  Edge-Ring (label 3): 22
  Loc (label 4): 224
  Near-full (label 5): 14
  Random (label 6): 56
  Scratch (label 7): 50
  none (label 8): 9411

Rare classes selected for AE augmentation (count < 100):
  Center (label 0)
  Donut (label 1)
  Edge-Ring (label 3)
```

```
  Near-full (label 5)
  Random (label 6)
  Scratch (label 7)

Augmenting class Center (n=68) with 3x AE reconstructions…

Augmenting class Donut (n=1) with 3x AE reconstructions…

Augmenting class Edge-Ring (n=22) with 3x AE reconstructions…

Augmenting class Near-full (n=14) with 3x AE reconstructions…

Augmenting class Random (n=56) with 3x AE reconstructions…

Augmenting class Scratch (n=50) with 3x AE reconstructions…

Augmented train shape: (10689, 26, 26, 3) (10689,)

Balanced val shape: (164, 26, 26, 3) (164,)
Balanced val class counts:
  Center: 14
  Edge-Loc: 40
  Edge-Ring: 4
  Loc: 39
  Near-full: 1
  Random: 14
  Scratch: 12
  none: 40

Class weights (after AE augmentation):
  Center: 4.366
  Donut: 296.917
  Edge-Loc: 5.656
  Edge-Ring: 13.496
  Loc: 5.302
  Near-full: 21.208
  Random: 5.302
  Scratch: 5.938
  none: 0.126

Augmented CNN Data Set Distribution

   Defect  Training Set  Validation Set  Test Set
   Center           272              14         8
    Donut             4               0         0
 Edge-Loc           210              40        45
Edge-Ring            88               4         5
      Loc           224              39        34
```

```
Near-full            56               1         1
   Random           224              14         4
  Scratch           200              12        10
     none          9411              40      2048

Saved table as Augmented_Data_Distribution.csv
```