Enhancing Bearing Quality Control: A CNN-Based Approach for bearing defect classification.

Mengyuan Huang
Stanford University
mengy88@stanford.edu

Abstract

Convolutional Neural Networks (CNNs) have demonstrated significant potential for accurate surface defect detection across various materials and products. However, their success typically hinges on the availability of extensive labeled datasets, which are often costly and difficult to acquire in real-world industrial settings. This project addresses the challenge of defect classification with limited data by exploring various strategies using a relatively small dataset of bearing images. I first establish a shallow CNN model as a baseline and then investigate different techniques to mitigate overfitting inherent in deep learning processes when data is scarce.

1. Introduction

Bearings are vital components across countless industrial applications, facilitating smooth, efficient motion and minimizing friction and wear. In electric vehicles (EVs), for instance, bearings are crucial for the seamless operation of numerous moving parts, directly contributing to extended range and improved efficiency. Given their critical role, ensuring the quality of bearings is paramount.

1.1. Motivation

For this project, I collected bearing images from my family's factory in China to investigate the application of CNNs for bearing defect detection. Given the nature of the available images and the course's time constraints, I focused on bearing defect classification. My primary goal is to achieve better than baseline accuracy in detecting surface defects on bearings, even with a limited dataset. The bearings with defects are categorized into ten distinct types: {rust, scale, bruised, bruised-surface, abrasion, scratch, yellow-rust, dim, black, and holder-scratch}. Dataset also contains bearing images without any defects.

1.2. Problem Statement

The problem: using CNN model to achieve high accuracy on classifying bearing images into ten defect types with limited image dataset.

Inputs: 1500 * 1500 * 3 channels bearing images (total of 460 images) captured from industrial cameras, along with image labels represented as multi-hot encoding. The order of the defect types list defines the index for each defect category in the encoding. For example, $[1, 1, 0, ..., 0]$ represents the image has rust and scale defects. $[0, ..., 0]$ means the image is without any defects.

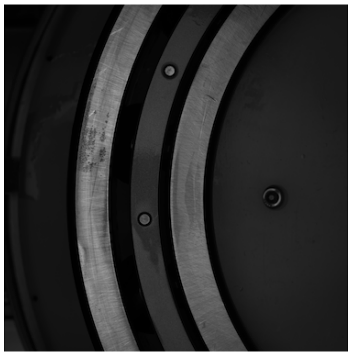


Figure 1: example bruised bearing surface

Output: predictions to categorize bearing surface defects into one or more of ten categories: {rust, scale, bruised, bruised-surface, abrasion, scratch, yellow-rust, dim, black, holder-scratch}. The classification output will be multi-labeled, as a single bearing image may exhibit multiple types of defects simultaneously. If no prediction in output, it means the bearing surface does not have any defects.

2. Related Work

Traditionally, bearing quality checks are performed manually, a process that is often time-consuming, subjective, and susceptible to human error. Fortunately, Convolutional

Neural Networks (CNNs) offer a promising solution for automating and enhancing the accuracy of surface defect detection in various industrial products [3]. This is an active and well-researched field. For example, a CNN with max pooling was proposed for faster steel defect detection [7].

Other research has demonstrated successful defect detection using a Faster R-CNN-based feature extraction module [6] and a Fast R-CNN method developed by Girshick to identify five types of defects [9]. Compared to its predecessor, Fast R-CNN generally consumes fewer computational resources [2]. While Fast R-CNN was a significant breakthrough in object detection, it has been surpassed by subsequent advancements in object detection. The overall object detection landscape has evolved to offer more efficient and often more accurate solutions. For example Single-Stage Detectors (YOLO[8] and SSD[5]), which perform both object localization and classification in a single pass, making them much faster and suitable for real-time applications.

For State-of-the-Art and highly effective defect detection now, fast R-CNN is still a very strong contender, especially when high accuracy and precise bounding box localization are critical. YOLO family (YOLOv7[12], YOLOv10[11], etc.) is also extremely popular due to their excellent balance of speed and accuracy. They are often preferred for real-time inspection systems where inference speed is paramount.

For this project, I'll focus on a base CNN model specifically designed for multi-label image defect classification. More advanced models like Fast R-CNN aren't necessary since their object detection capabilities aren't required for my task.

I'll concentrate on improving accuracy and preventing overfitting given my limited dataset. There are multiple strategies to prevent overfitting in small dataset[1]. One strategy I'll employ is data augmentation to boost the model's generalization capabilities, which is especially helpful with small datasets. For instance, techniques like random image cropping and patching have shown promising results in deep CNNs [10]. Additionally, I explored early stopping as a method to combat overfitting on smaller datasets. This technique prevents the model from simply memorizing the training data, ultimately making overparameterized neural networks more robust to label noise [4].

3. Methods

For this project, I used a shallow 3-layer CNN for image surface defect detection where one image can have multiple types of defects (total 10 types). This is a multi-label classification problem and my data label is preprocessed to be multi-hot encoding.

3.1. Model Architecture

The core CNN architecture is similar to single-label classification. The key difference is the output layer and activation function. The output layer (fully connected) should have 10 neurons, one for each defect category.

```
--- Model Architecture ---
DefectDetectionCNN(
  (conv_layers): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3)
      , stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride
      =2, padding=0, dilation=1,
      ceil_mode=False)
    (3): Conv2d(32, 64, kernel_size=(3,
      3), stride=(1, 1), padding=(1, 1)
      )
    (4): ReLU()
    (5): MaxPool2d(kernel_size=2, stride
      =2, padding=0, dilation=1,
      ceil_mode=False)
    (6): Conv2d(64, 128, kernel_size=(3,
      3), stride=(1, 1), padding=(1, 1)
      )
    (7): ReLU()
    (8): MaxPool2d(kernel_size=2, stride
      =2, padding=0, dilation=1,
      ceil_mode=False)
  )
  (fc_layers): Sequential(
    (0): Linear(in_features=4476032,
      out_features=256, bias=True)
    (1): ReLU()
    (2): Linear(in_features=256,
      out_features=10, bias=True)
  )
)
```

3.1.1 Activation Function

For activation function, instead of traditional *softmax*, which is used for multi-class single-label classification where probabilities sum to 1, I choose to use the *sigmoid* 1 activation function for each of the 10 output neurons. *sigmoid* output s a probability between 0 and 1 independently for each neuron, which allow each neuron to predict the presence or absence of its corresponding defect type, regardless of the other defect types.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

3.1.2 Loss Function

The loss function is also crucial to the training. I decide to use Binary Cross-Entropy (BCE) Loss which is a common loss function used in multi-label classification problems. It measures the dissimilarity between the true labels and the predicted probabilities. BCE loss for this project treats each of the 10 defect categories as an independent binary classification problem 2. And the total loss over N samples would be the average of these 10 individual multi-label losses 3.

$$L_{BCE}^{(i)} = -\sum_{j=1}^{C}[y_{ij}log(\hat{y_{ij}}) + (1 - y_{ij})log(1 - \hat{y_{ij}})] \quad (2)$$

$$L_{BCE} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C}[y_{ij}log(\hat{y_{ij}}) + (1 - y_{ij})log(1 - \hat{y_{ij}})] \quad (3)$$

3.1.3 Training and Evaluation

After training, the *sigmoid* outputs will be probabilities. The final output layer chooses a threshold of the probability ($t = 5$ in this project) to convert these probabilities into binary predictions (0 or 1) for each defect type. An output greater than the threshold indicates the presence of that defect.

For metrics, I choose to use micro-averaging accuracy (Mean Binary Accuracy in the implemented code). It evaluate performance on a per-label basis and then aggregate these results using micro-averaging.

3.2. Reducing memory consumption

Given that my image data size is (1500 * 1500), which is significantly bigger than traditional image size (i.e. 512 * 512), one technical problem emerges during the training is memory exhaustion before training finished. For the above CNN architecture, the model will need 1,145,960,266 total learnable parameters, which is indeed massive and will almost certainly cause CUDA out of memory on consumer-grade GPUs. This parameter count is extremely high even for very deep, state-of-the-art models like large LLMs. Reducing the image size is impractical, so I need to focus on model architecture update.

The culprit is the transition from the convolutional layer to the fully connected layers. After 3 *MaxPool2d* layers, the feature map size is 128 channels * 187 height * 187 width. When *flatten()* this, the model has 128 * 187 * 187 = 4,408,072 features per each image. When feed this into a `nn.Linear()` it creates over 1.1 billion parameters in just one layer!

To solve this issue, I decided to update the architecture to use global average pooling (GAP). This is the most effective and common technique to drastically reduce the number of parameters in the classification head, especially with large input images. Instead of flattening the entire feature map (128 * 187 * 187) into a huge 1D vector, we use `nn.AdaptiveAvgPool2d((1, 1))` calculates the average value for each of the 128 feature maps. This reduces each 187 * 187 map to a single value. Therefore previous tensor `(Batch_size, 128, 187, 187)` becomes `(Batch_size, 128, 1, 1)`.

```
--- Modified Model Architecture ---
LightweightDefectDetectionCNN(
  (conv_layers): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3)
        , stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride
        =2, padding=0, dilation=1,
        ceil_mode=False)
    (3): Conv2d(32, 64, kernel_size=(3,
        3), stride=(1, 1), padding=(1, 1)
        )
    (4): ReLU()
    (5): MaxPool2d(kernel_size=2, stride
        =2, padding=0, dilation=1,
        ceil_mode=False)
    (6): Conv2d(64, 128, kernel_size=(3,
        3), stride=(1, 1), padding=(1, 1)
        )
    (7): ReLU()
    (8): MaxPool2d(kernel_size=2, stride
        =2, padding=0, dilation=1,
        ceil_mode=False)
  )
  (global_avg_pool): AdaptiveAvgPool2d(
      output_size=(1, 1))
  (fc_layer): Linear(in_features=128,
      out_features=10, bias=True)
)
```

After applying GAP, the total learnable parameters reduced from 1.1 billion to 94,538.

3.3. Preventing Overfitting

Given my limited datasets, I uses 80% as training samples, rest for testing/validation set. One concern with limited this limited dataset size is overfitting. It is crucial to add evaluation step within the training loop to detect overfitting.

3.3.1 Data Augmentation

From the research work, I choose to add data augmentation to artificially increase the size and diversity of the training data by applying random transformations. For this project, I applied flip and rotation to the original images. The model sees slightly different versions of the same image each epoch, making it more robust. The transformation is applied on-the-fly, which means they are dynamically added each time an image is loaded from the training dataset by the `DataLoader` during training. Because the transformation are random and applied every time the image is fetched, the model sees a slightly different version of the same original image in each epoch. Conceptually this adds extra training data, or more accurately, it significantly increases the effective size and diversity of the training set, but without physically more data to take up memory space.

3.3.2 Early Stop

Early stop is also applied so that learning stops when the model's performance on the validation set stops improving or start to degrade. This prevents the model from continuing to memorize the training data. At the end of each epoch, after the training loss is calculated, there is a dedicated model evaluation block that iterates over the `test_loader` to calculate `validation_epoch_loss`. Early stop method then applied to this validation loss and decides if the training should continue to prevent having an overfit model.

I also explored K-Fold cross-validation, which should conceptually provides the most reliable estimate for small datasets but does take significantly longer to run. Another alternatives is stratified sampling, but given that my dataset are imbalanced (e.g, in this project dataset contains significantly more "bruised" defects than all other types), a simple random split might result in some folds having very few or no examples of rare classes. Stratified splitting requires that each split maintains the same proportion of classes as the original dataset.

4. Dataset and Features

The given dataset is coming from a Chinese factory which my family runs to supply bearing parts for some EV factories in China. I obtained the bearing images from industrial camera in the factory and manually labeled with defect types and defect locations in corresponding json files. I then resized the image to be 1500 * 1500 with 3 channels. Given the time constraint, I only acquired 460 images with 10 defect types: {rust, scale, bruised, bruised-surface, abrasion, scratch, yellow- rust, dim, black, and holder-scratch}. Following distributions of each defect types:

```
--- Defect Type Distribution ---
                  Count  Percentage
No Defects          249      54.13%
bruised              69       15.0%
bruised-surface      61      13.26%
scratch              38       8.26%
black                36       7.83%
abrasion             15       3.26%
scale                14       3.04%
yellow-rust          11       2.39%
rust                  8       1.74%
holder-scratch        8       1.74%
dim                   6        1.3%
```

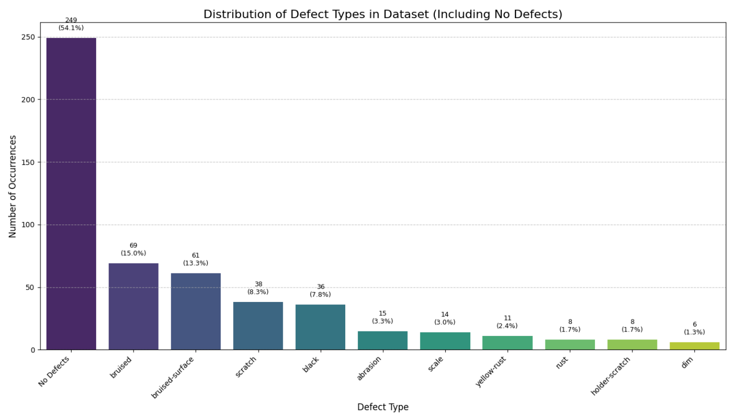
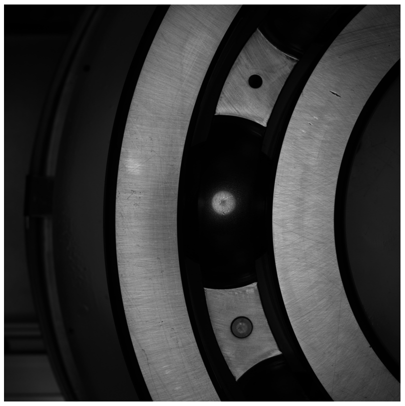
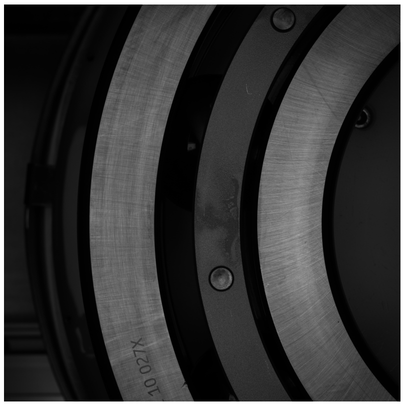


Figure 2: Dataset defect types distributions

There are total of two types of bearing style, but all dataset images are in the same size after preprocessing:



(a) Style 1



(b) Style 2

Figure 3: Two basic bearing styles

I choose to use the basic 80/20 split for training and testing/validation sets. Therefore I have 368 training samples and 92 testing/validation samples.

In 3.3.1 I have explained the concern for overfitting on limited datasets and decided to use data augmentation in training per each epoch. The data augmentation is applied on-the-fly per each epoch but will not physically add extra images to the original datasets.

5. Experiments/Results/Discussion

5.1. Hyperparameters

I used `batch_size = 8`. This is a smaller batch size because the image input is quite large so I need to manage GPU memory. I set `learning_rate = 0.001` . This is a quite common learning rate since I chose to use Adam optimizer. It comes from empirical observations and the design of the Adam algorithm itself. 0.001 with Adam has been found to be a good balance, and from the training process I think the result achieved reasonable accuracy. I chose Adam optimizer for the CNN model because it is a very common and often effective choice in deep learning. Adam is an adaptive learning rate optimization algorithm, and especially a good pick for my model because it allows me to quickly get a working model and see initial results. Another good reason for me to choose Adam is because it is less sensitive to the exact global learning rate value than plain SGD.

I did not implement cross validation in the model due to the time constraint as a single-person project. Implementing k-fold cross-validation significantly change the overall training loop structure. It's more complex to implement because it wraps the entire training and evaluation process. However, I think cross validation is really important to provide robust evaluation.

5.2. Primary metrics

The primary metrics for the defect detection model are not just a single accuracy score, because standard accuracy can be very misleading in multi-label scenarios. In this project, the image surface can have zero or multiple defects. Standard "accuracy" is defined as subset accuracy, which gives the percentage of samples for which the entire set of predicted labels exactly match the entire set of true labels. However, this definition is too strict. For example, if an image has "rust" and "scratch", but the model predicts "rust" and "dim", it gets 0% subset accuracy, even though one label is correct. Another reason is that half of the data are no defects, a model that predicts "no defect" would always have a very high subset accuracy, which is misleading.

I choose to evaluate performance on per-label basis and then aggregate results with micro-averaging (Mean Binary Accuracy). It calculates the global true positives, false positives and false negatives by summing across all individual labels and all samples. Then it computes per label correctness from these global counts. It gives a quick overall sense of performance across all instances regardless of class distributions.

Another metrics I monitored in the model is validation loss to detect overfitting and trigger early stopping if possible.

5.3. More memory management

In addition to reducing GPU memory consumption caused by large learnable parameters mentioned in 3.2 with GAP, the model still faces CUDA out of memory issue during training. This means the bottleneck is now firmly in the convolutional layers as a 1500 * 1500 image genuinely large for CNN models on my environment. I applied another aggressive strategy to use depthwise separable convolutions. This introduce a single filter that applied to each input channel independently, which is depthwise convolution. Then a 1 * 1 convolution is used to combine the outputs across the channels. This mixes channel information. This factorization drastically reduces the number of parameters compared to standard convolution, making the model much lighter. The total parameter count drops further from 94,538 to 12,074 in the end. Then the training can perfectly fit into my current GPU memory.

5.4. Results

I trained model for 50 epoch with following results. Based on the result below both training and validation loss generally decrease over the 50 epochs, indicating that the model is learning and improving. the validation loss converges to a value relatively close to the training loss in the end, which I think suggesting a good generalization to some extent.

Initial rapid decrease happened around epoch 1 - 5, where train loss starts at 0.4188 and quickly drops to around 0.19 - 0.20. this indicates that the model is rapidlly learning the basic patterns in the training data. For validation loss, it shows a rapid decrease from 0.2491 to around 0.18 - 0.19. This is a positive sign I believe, showing the model's initial learning is effectively generalizing to unseen data.

In epoch 6 - 20, both training and validation losses continue to decrease, but the rate of decreases slows down. The training loss gradually falls from around 0.19 to 0.17. The validation loss also continues to decrease, but with some fluctuations. For example, slight increase at epoch 8, 11, 13, 17, 19, 20. This indicates the model is fine-tuning its parameters.

In the rest epochs, it seems the model have fluctuations and plateauing. The train loss continue to decrease, but slowly. In the end it reaches around 0.15. This suggests the model is still finding ways to fit the training data better. For validation loss, it shows more visible fluctuations in the rest epochs. For example, 0.1619 at epoch 21 and 0.1609 at epoch 25, then 0.1576 at epoch 29. It also experienced significant jumps upwards at epoch 23 which has loss of 0.1850. These upward spikes are the cause of early stopping counter that frequently triggered.

Overall the consistent decrease in both training and validation losses, particularly in the early stages, which I believe demonstrates that the model is effectively learning the underlying pattern.

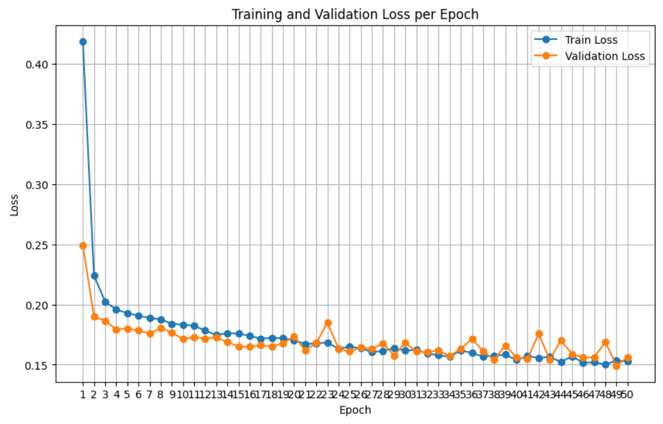


Figure 4: Train and Validation Loss per Epoch

We can notice that there are some spike in validation loss, which align with the early stop triggered in the same epoch. This indicates that those are the key signs of overfitting. For epochs that triggered early stopping, they are likely reached a point where further training on the current dataset would lead to a decrease in its ability to generalize to new data, so they should be stopped early. The validation loss generally tracks the training loss well, and the final gap between them is not excessively large. For reference, the train loss is 0.15 and the best validation loss is 0.1492. This indicates that the model is generalizing reasonably well to unseen data and not severely overfitting.

The graph below illustrates the value of early stopping. Without it, training loss will continue to decrease till epoch 50, the validation loss might have ended up higher than its minimum at epoch 49.

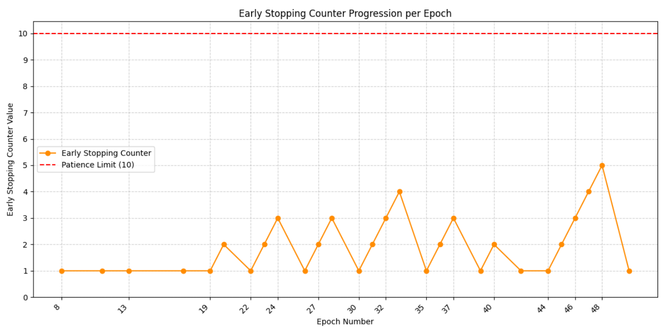


Figure 5: Early Stopping Counter Progression per Epoch

Validation accuracy (Mean binary accuracy) per epoch stays quite stable. The results are all around 0.95. I think the stable accuracy doesn't necessarily mean the model doesn't improve much over time, given that the validation loss is still decreasing. The accuracy is a discrete measure and can sometimes be less sensitive to small improvements or changes in the model's confidence. The loss on the other hand can decreases significantly with a stable accuracy. I think this could probably demonstrate cases like "model prediction for a certain defect type from a probability of 0.51 to 0.99". In this scenario, the accuracy remains pretty much the same as long as the probability pass the threshold and categorized as the correct label. But the loss will decrease significantly. The decreasing in validation loss indicates that the model is still learning and making better and more confident predictions.

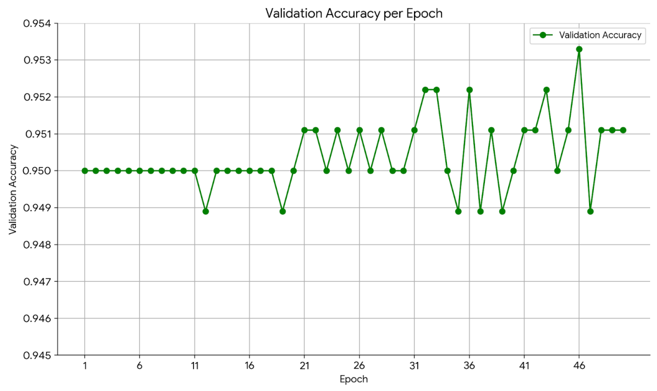


Figure 6: Validation Accuracy per Epoch

Using 0.5 as threshold and output probability samples with mean binary accuracy across all labels and samples on test set as shown below. I think the sample shows that the model predicts pretty well on the no defect cases, which also takes more than half of the total dataset. Therefore I can achieve a 95% high accuracy.

```
--- Sample Predictions vs. True Labels (First
    3 Samples from Test Set) ---
Sample 1 (correct):
True Labels:        [0 0 0 0 0 0 0 0 0 0]
Probabilities:      [0.008 0.019 0.108 0.145
    0.026 0.051 0.017 0.011 0.072 0.018]
Predicted Labels:   [0 0 0 0 0 0 0 0 0 0]
----------------------------------------
Sample 2 (correct):
True Labels:        [0 0 0 0 0 0 0 0 0 0]
Probabilities:      [0.005 0.014 0.089 0.126
    0.02  0.039 0.012 0.007 0.058 0.013]
Predicted Labels:   [0 0 0 0 0 0 0 0 0 0]
----------------------------------------
Sample 3 (wrong):
True Labels:        [0 0 1 0 0 0 0 0 0 0]
```

```
Probabilities:       [0.003 0.008 0.067 0.103
    0.014 0.027 0.006 0.004 0.042 0.008]
Predicted Labels:    [0 0 0 0 0 0 0 0 0 0]
-----------------------------------------
```

6. Conclusion/Future Work

I think in general this CNN defect detection model achieves pretty reasonable accuracy with the limited dataset. With help of GAP and depthwise separable convolution layer, I can successfully run the training with the large image size that fits into a limited GPU memory. I also applied early stopping and data augmentation to prevent overfitting introduce by the limited amount of data.

In the future, I think increasing the dataset size will definitely help to give the model a more reasonable evaluation. Another prevent overfitting strategy that I think is important to have is k-fold cross validation, which should provide a more reliable and less biased estimate of the model's performance on unseen data.

7. Contributions and Acknowledgments

This is a single person project. I did all the work described in this report. Data coming from a Chinese factory that produces bearings for an EV manufacturer.

References

[1] R. Caruana, S. Lawrence, and C. L. Giles. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *Advances in Neural Information Processing Systems (NIPS)*, volume 13, pages 402–408, 2000.

[2] A. Howard, M. Sandler, B. Chen, W. Wang, L.-C. Chen, M. Tan, G. Chu, V. Vasudevan, Y. Zhu, R. Pang, et al. Searching for mobilenetv3. In *Proceedings of the 2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1314–1324, Seoul, Republic of Korea, Oct–Nov 2019.

[3] D. Li, Y. Zhao, X. Liu, C. Yuan, G. Song, and H. Yan. A surface defect detection based on convolutional neural network. In *2017 13th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, pages 1939–1944, 2017.

[4] M. Li, M. Soltanolkotabi, and S. Oymak. Gradient descent with early stopping is provably robust to label noise for overparameterized neural networks. In S. Chiappa and R. Calandra, editors, *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, volume 108 of *Proceedings of Machine Learning Research*, pages 4313–4324. PMLR, 2020.

[5] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[6] Z. Liu, L. Wang, C. Li, and Z. Han. A high-precision loose strands diagnosis approach for isoelectric line in high-speed railway. *IEEE Transactions on Industrial Informatics*, 14(3):1067–1077, 2017.

[7] J. Masci, U. Meier, D. Ciresan, J. Schmidhuber, and G. Fricout. Steel defect classification with max-pooling convolutional neural networks. In *2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–6, Brisbane, Australia, June 2012.

[8] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[9] S. Ren, K. He, R. Girshick, and J. Sun. Apple surface defect detection method based on weight comparison transfer learning with mobilenetv3. In *Advances in Neural Information Processing Systems*, pages 91–99, 2025.

[10] R. Takahashi, T. Matsubara, and K. Uehara. Ricap: Random image cropping and patching data augmentation for deep cnns. In J. Zhu and I. Takeuchi, editors, *Proceedings of The 10th Asian Conference on Machine Learning*, volume 95 of *Proceedings of Machine Learning Research*, pages 786–798. PMLR, 14–16 Nov 2018.

[11] A. Wang, H. Chen, L. Liu, K. Chen, Z. Lin, J. Han, and G. Ding. Yolov10: Real-time end-to-end object detection. In *Advances in Neural Information Processing Systems*, 2024.

[12] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao. Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. *arXiv preprint arXiv:2207.02696*, 2022.