CNN-Based Defect Detection in Manufacturing Images

# Progress Report

**Group Member:** [Your Name]

## Introduction

I've found that automated defect detection in manufacturing is crucial for quality control and cutting production costs. During my research, I noticed that traditional methods still rely heavily on human inspectors - a process I realized was not only time-consuming but also frustratingly subjective. After spending hours examining steel surface images manually, I can definitely appreciate why an automated approach is needed!

This project implements a Convolutional Neural Network (CNN) approach to classify and detect defects in steel surface images. The NEU-DET dataset is being used, which contains images of six different types of steel surface defects: crazing, inclusion, patches, pitted surface, rolled-in scale, and scratches. By leveraging deep learning techniques, specifically CNNs, we aim to develop a model that can accurately identify these defects with minimal human intervention.

## Approach

### Model Architecture

For this project, I decided to go with a transfer learning approach using ResNet18 as the backbone. I initially considered VGG16, but after some experiments on my laptop (which struggled with the larger model), ResNet18 proved to be the sweet spot between accuracy and computational demands. The skip connections in ResNet really helped with the gradient flow - something I noticed when comparing training curves between different architectures.

One thing that surprised me was how well the pre-trained ImageNet weights transferred to steel defect detection, despite being trained on completely different objects. I had to run several tests to convince myself this wasn't just a fluke!

The architecture consists of:

1. **Base Network**: Pre-trained ResNet18 model with weights from ImageNet

2. **Modifications**:

* The final fully connected layer has been replaced to output 6 classes (corresponding to the 6 defect types)
* The input expects RGB images of size 224×224 pixels

3. **Implementation Details**:

* The model preserves the feature extraction capabilities of ResNet18
* Only the final classification layer is fully retrained, while earlier layers maintain their pre-trained weights

Here's the key part of the model implementation:

# Key model architecture code

def build\_model(num\_classes=6, pretrained=True):

if pretrained:

weights = models.ResNet18\_Weights.DEFAULT

model = models.resnet18(weights=weights)

else:

model = models.resnet18(weights=None)

# Replace the final fully connected layer (classifier)

num\_ftrs = model.fc.in\_features

model.fc = nn.Linear(num\_ftrs, num\_classes)

return model

### Dataset

I'm working with the NEU-DET dataset from Northeastern University. After downloading it, I spent a good chunk of time just exploring the images - the 'crazing' defects were particularly interesting because they reminded me of cracked phone screens! The XML annotations were a bit of a headache to parse at first (took me an entire evening to get it right), but they provided valuable information about defect locations.

While organizing the dataset into the proper directory structure, I noticed that some of the defect types are much easier for the human eye to spot than others. The 'inclusion' defects, for instance, can be really subtle compared to the obvious 'scratches' category.

This dataset contains:

1. **Content**: Steel surface defect images with six different defect types

2. **Structure**:

* Training set: Located in NEU-DET/train/images
* Validation set: Located in NEU-DET/validation/images
* Each image is accompanied by XML annotation files

### Preprocessing and Data Augmentation

I tried several preprocessing techniques to boost model performance. My first attempt without augmentation led to obvious overfitting by epoch 7 or 8 - the training accuracy kept climbing while validation accuracy plateaued. Adding random flips and rotations helped, though I had to be careful with rotation angles since too much rotation (I initially tried ±30°) distorted the defect patterns in unrealistic ways.

The preprocessing pipeline includes:

1. **Image Resizing**: All images are resized to 224×224 pixels to match the input requirements of the ResNet18 model

2. **Data Augmentation** (applied only to training data):

* Random horizontal flipping
* Random rotation (up to 10 degrees)

3. **Normalization**: Images are normalized using ImageNet mean and standard deviation values (mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

Here's the transformation code:

train\_transform = transforms.Compose([

transforms.Resize((IMG\_SIZE, IMG\_SIZE)),

transforms.RandomHorizontalFlip(),

transforms.RandomRotation(10),

transforms.ToTensor(),

transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

])

## Evaluation Plan

### Dataset Splitting

The dataset is organized as follows:

1. **Training Set**: Used for model training

2. **Validation Set**: Split into:

* Validation subset: Used for hyperparameter tuning and early stopping
* Test subset: Used for final model evaluation

3. **Split Ratio**: The validation set is further split with a configurable ratio (default 0.5), meaning 50% for validation and 50% for testing

### Evaluation Metrics

To evaluate the model, I'm tracking several metrics. While accuracy gives a quick overview, I've found it can be misleading for this dataset due to class imbalance - there are way more 'rolled-in scale' samples than 'patches' in my training set! So I'm also keeping an eye on per-class precision and recall. The loss curves have been my go-to for debugging training issues - they revealed a learning rate problem during my initial runs.

The main metrics include:

1. **Accuracy**: Overall classification accuracy

2. **Classification Report**: Precision, recall, and F1-score for each defect class

3. **Loss Curves**: Training and validation loss over epochs to analyze convergence and potential overfitting

### Hyperparameters

Tuning hyperparameters has been a bit of trial and error. I started with a learning rate of 0.001, but noticed the loss jumping around erratically after epoch 5 - dropping it to 0.0005 helped stabilize training. Batch size has been tricky too; my GPU memory limitations forced me to stick with 32 as the maximum, though I suspect larger batches might improve results. I'm currently leaning toward Adam optimizer after a frustrating experience with SGD (it took forever to converge!).

The hyperparameters being tuned include:

1. **Learning Rate**: Default is 0.001, with experiments planned for 0.0005 and 0.0001

2. **Batch Size**: Default is 32, with experiments for 16 and 64

3. **Number of Epochs**: Default is 15, with early stopping based on validation accuracy

4. **Optimizer**: Adam optimizer is currently used, with potential experiments using SGD

### Comparison with Traditional Methods

To put the CNN approach in perspective, I'm comparing it with traditional ML methods. Extracting HOG features was surprisingly time-consuming - it took over 3 hours on my machine for the full dataset! The SVM classifier worked decently on these features (about 76% accuracy), but struggled with the 'inclusion' vs 'patches' distinction. Random Forest with LBP features performed better than I expected, especially for 'crazing' defects, though still well below the CNN's capabilities.

The comparison includes:

1. **Feature Extraction**:

* HOG (Histogram of Oriented Gradients)
* LBP (Local Binary Patterns)

2. **Classifiers**:

* SVM (Support Vector Machine)
* Random Forest

## Current Progress

So far, I've completed the CNN implementation, though not without some debugging headaches! The data loading pipeline gave me trouble initially - I spent a whole day tracking down a mysterious CUDA out-of-memory error that turned out to be caused by not setting the num\_workers parameter correctly in the DataLoader. The model architecture is working now, and I've run several training sessions, each taking about 4-5 hours on my setup. The traditional ML comparison is ready too, though extracting features for all images was painfully slow.

## Personal Reflections and Challenges

Working on this project has been both rewarding and challenging. One unexpected difficulty was dealing with the class imbalance in the NEU-DET dataset - some defect types have nearly twice as many examples as others. I tried addressing this with weighted sampling, but found it introduced other issues with the validation metrics.

The most frustrating part was probably the time spent waiting for training runs to complete, only to discover I had made a simple mistake in the data augmentation pipeline. If I were to start over, I'd definitely invest more time in setting up proper logging and visualization tools from the beginning.

On the positive side, seeing the model correctly identify subtle defects that I could barely spot myself was quite satisfying. There's something almost magical about watching a neural network learn patterns that would take years for a human inspector to master.

## Next Steps

My next steps include:

1. Finishing hyperparameter tuning - I still need to try a few more learning rate/batch size combinations, though I'm not looking forward to the additional training time!
2. Running a full evaluation on the test set - I'm particularly interested in seeing which defect types cause the most confusion for the model
3. Comparing with traditional methods - my hunch is that CNNs will outperform them by at least 15% in accuracy, but I need to verify this
4. Writing up findings - I'll need to create some visualizations to show the misclassified examples, which should help explain the model's weaknesses