Toronto Metropolitan University

Department of Aerospace Engineering
Faculty of Engineering & Architectural Science

| | |
|---|---|
| **Semester (Term, Year)** | F2025 |
| **Course Code** | AER850 |
| **Course Section** | 02 |
| **Course Title** | Introduction to Machine Learning |
| **Course Instructor** | Dr. Reza Faieghi |
| **Submission** | 2 |
| **Submission No.** | 2 |
| **Submission Due Date** | November 5, 2025 |
| **Title** | Project 2 - NC-DCNN - Kai Stewart |
| **Submission Date** | November 5, 2025 |

| **Submission by (Name):** | **Student ID (XXXX1234)** | **Signature** |
|---|---|---|
| **Kai Stewart** | **29849** |  |

*By signing the above you attest that you have contributed to this submission and confirm that all work you contributed to this submission is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, and "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Academic Integrity Policy 60, which can be found at www.torontomu.ca/senate/policies/* *Aerospace Assignment Cover as of May 2022*

# 1.0 - Introduction

Project 2 of AER850 challenges students to develop a Deep Convolutional Neural Network (DCNN) capable of detecting three types of structural defects common in aerospace production environments. The three defects are cracks, missing heads (of fasteners) and paint delaminations. Students are challenges to build two variations of DCNN models which can classify non conformances into their appropriate classes. The name that I have given this tool is **NC-DCNN**, Non-Conformance - Deep Convolutional Neural Network.

**Github - Akkoxs - NC-DCNN**

# 2.0 - Neural Network Architecture Design & Iteration

Firstly, I would like to leave a note on optimization (of computational expense); as I am sure many others have experienced issues regarding this for this project. I have begun to see the flaws in training a neural network solely through using a CPU. For the next project, I will surely be looking into how I can apply NVIDIA's CUDA toolkit to utilize my GPU for training neural networks.

**CPU:** AMD Ryzen 5600X **[CURRENT]**
**GPU:** NVIDIA RTX 4070 Super

Furthermore, a few optimization tweaks have been made in the code in an attempt to further speed up training times. They are as follows:

- **tensorflow.data.AUTOTUNE:** Which automatically runs a diagnostic on my PC resources and returns an optimal number of threads to use for faster training times
- **.cache():** Saves data after the first EPOCH to RAM such that it can read directly from it and not the disk for subsequent EPOCHs
- **.prefetch():** Starts preparing the next batch (pre-processing) before the current batch is done training.

It should also be noted tha the following models are trained on two separate devices for concurrent training. The two devices are a Dell Lattitude 5510 with a 10th generation i7 CPU and a PC with the specs shown above:

- **Laptop:** NC-DCNN.v0
- **Laptop:** NC-DCNN.v0_ALT
- **Laptop:** NC-DCNN.v0_PRIME
- **Laptop:** NC-DCNN.v0_SECUNDUS **[BASELINE MODEL]**
- **Laptop:** NC-DCNN.v0_TERTIUS **[VARIANT MODEL]**
- **PC:** NC-DCNN.v1 **[FAILED]**
- **PC:** NC-DCNN.v2 **[FAILED]**

## 2.1 - NC-DCNN.v0

The first model that will be trained is more of an introductory step into the realm of DCNNs, and seeks to make an ultralight model simply for testing out our pipeline. The DCNN architecture can be seen below in Figure 1.

| Layer (type) | Output Shape | Param # |
|---|---|---|
| conv2d (Conv2D) | (None, 498, 498, 8) | 224 |
| max_pooling2d (MaxPooling2D) | (None, 249, 249, 8) | 0 |
| separable_conv2d (SeparableConv2D) | (None, 247, 247, 16) | 216 |
| max_pooling2d_1 (MaxPooling2D) | (None, 123, 123, 16) | 0 |
| separable_conv2d_1 (SeparableConv2D) | (None, 121, 121, 32) | 688 |
| global_average_pooling2d (GlobalAveragePooling2D) | (None, 32) | 0 |
| dropout (Dropout) | (None, 32) | 0 |
| dense (Dense) | (None, 32) | 1,056 |
| dense_1 (Dense) | (None, 3) | 99 |

**Figure 1:** NC-DCNN.v0 Architecture

As we can see, it consists of 3 convolutional layers, each using a convolution function such as Conv2D() or SeparableConv2D() and a MaxPooling2D() layer. Conv2D() convolves across all 3 input channels at once, while SeparableConv2D() works only on 1 input channel at once, and produces a feature map for each channel. then combines them. SeparableConv2D() was used because it results in overall less parameters than Conv2D(), which was desirable for an ultralight model and due to computational limitations.

When normally a model would undergo a Flatten() layer, this model chooses to opt for a GlobalAveragePooling2D() layer. Both layers effectively output a 1D vector such that it can be inputted into the fully connected layers below it, however, they achieve this in different ways. Flatten() quite literally stacks up all values to create a single 1D vector to feed into dense layers. GlobalAveragePooling2D() computes the average of each feature map to drastically reduce the number of parameters and thus computational expense.

There are also 2 dense layers, one of which is the output layer which must be 3 and using the activation function of softmax. All other operations in this model uses activation functions of ReLU.

**List of Key Design Parameters for NC-DCNN.v0:**

- **Epochs** = 20 [Only made it to Epoch 6]
- **Batch Size** = 32
- **Early Stop Patience** = 3
- **Optimizer** = Adam
- **Learning Rate** = 1e-3
- **Loss** = Categorical Cross-Entropy
- **Dropout** = 0.2
- **Convolution Activation Function** = ReLU
- **Fully Connected Activation Functions** = ReLU
- **Output Activation Function** = Softmax

**Table 1:** NC-DCNN.v0 Parameter Breakdown

| | **Total Params** | **Trainable Params** | **Non-Trainable Params** |
|---|---|---|---|
| **No. of Params** | 2,283 | 2,283 | 0 |
| **Size** | 8.92 KB | 8.92 KB | 0.00 B |

**Table 2:** NC-DCNN.v0 Evaluation Metrics at EPOCH = 6

| | **Training** | **Validation** |
|---|---|---|
| **Accuracy** | 0.4120 | 0.4780 |
| **Loss** | 1.0764 | 1.0696 |

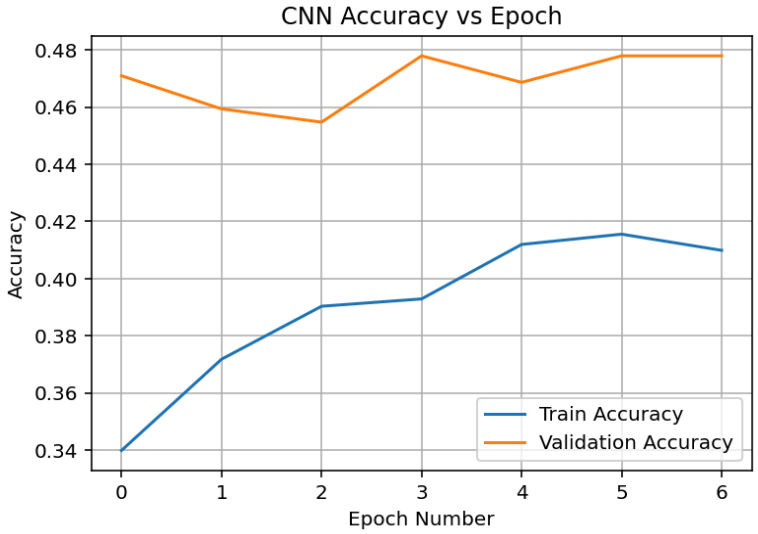


**Figure 2:** NC-DCNN.v0 Accuracy of Training/Validation vs Epoch

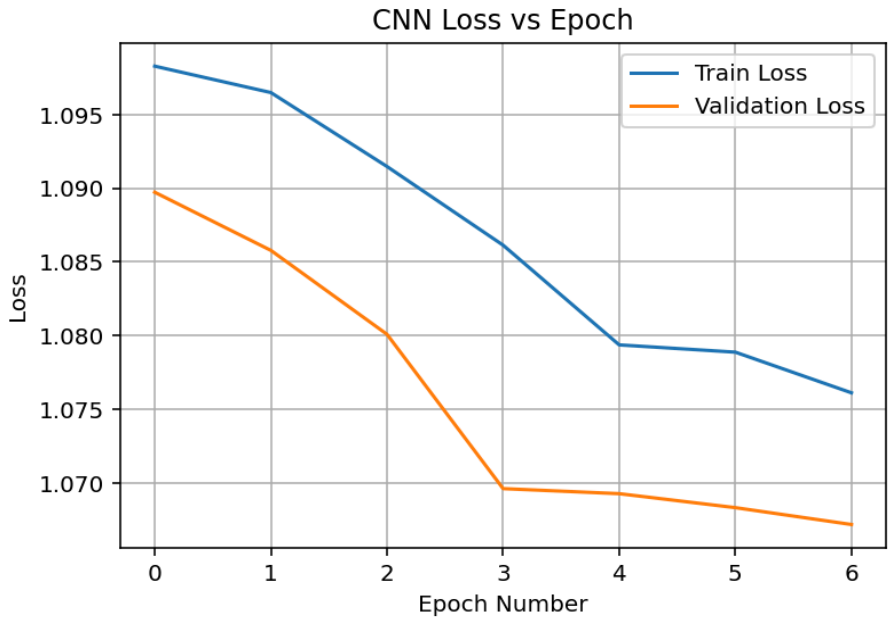


**Figure 3:** NC-DCNN.v0 Loss of Training/Validation vs Epoch

It should be noted that while the number of Epochs was set to 20, this model only trained until Epoch number 6. This is due to the early_stop object that was created, which monitors validation accuracy and will stop training the model if 3 Epochs pass without a significant change in validation_accuracy. Figure 2 and Figure 3 show us that the model was more or less headed in the right direction, despite slight fluctuations in the accuracy plot, which is to be expected. I believe this plot simply ended early because of a small patience score and thus, we will increase it for further iterations.

## 2.2 - NC-DCNN.v0_ALT

Another experimental lightweight model which experimented with some new layers, such as BatchNormalization() and Leaky ReLU. BatchNormalization() normalizes the outputs of layer above it, in this scenario, a Conv2D() or SeparableConv2D() layer. This realigns the data after each convolutional layer and helps to prevent overfitting.

| Layer (type) | Output Shape | Param # |
|---|---|---|
| conv2d_5 (Conv2D) | (None, 498, 498, 16) | 448 |
| batch_normalization_10 (BatchNormalization) | (None, 498, 498, 16) | 64 |
| leaky_re_lu_8 (LeakyReLU) | (None, 498, 498, 16) | 0 |
| max_pooling2d_10 (MaxPooling2D) | (None, 249, 249, 16) | 0 |
| conv2d_6 (Conv2D) | (None, 247, 247, 32) | 4,640 |
| batch_normalization_11 (BatchNormalization) | (None, 247, 247, 32) | 128 |
| leaky_re_lu_9 (LeakyReLU) | (None, 247, 247, 32) | 0 |
| max_pooling2d_11 (MaxPooling2D) | (None, 123, 123, 32) | 0 |
| separable_conv2d_8 (SeparableConv2D) | (None, 121, 121, 64) | 2,400 |
| batch_normalization_12 (BatchNormalization) | (None, 121, 121, 64) | 256 |
| leaky_re_lu_10 (LeakyReLU) | (None, 121, 121, 64) | 0 |
| max_pooling2d_12 (MaxPooling2D) | (None, 60, 60, 64) | 0 |
| separable_conv2d_9 (SeparableConv2D) | (None, 58, 58, 128) | 8,896 |
| batch_normalization_13 (BatchNormalization) | (None, 58, 58, 128) | 512 |
| leaky_re_lu_11 (LeakyReLU) | (None, 58, 58, 128) | 0 |
| max_pooling2d_13 (MaxPooling2D) | (None, 29, 29, 128) | 0 |
| separable_conv2d_10 (SeparableConv2D) | (None, 27, 27, 64) | 9,408 |
| batch_normalization_14 (BatchNormalization) | (None, 27, 27, 64) | 256 |
| global_average_pooling2d_3 (GlobalAveragePooling2D) | (None, 64) | 0 |
| dropout_3 (Dropout) | (None, 64) | 0 |
| dense_6 (Dense) | (None, 64) | 4,160 |
| dense_7 (Dense) | (None, 3) | 195 |

**Figure 4:** NC-DCNN.v0_ALT Architecture

LeakyReLU() is a variant of the activation function ReLU, which was used in the previous version, NC-DCNN.v0. The main difference being that LeakyReLU() allows for some negative values to 'leak' through, whereas ReLU would make those values 0. It is said that if ReLU works, LeakyReLU should not have to be used, as it is mainly used in cases where ReLU kills too many neurons by its nature. Here, it was just used experimentally to see if the model performance is what we would expect.

**List of Key Design Parameters for NC-DCNN.v0_ALT:**

- **Epochs** = 25 [Only made it to Epoch 6]
- **Batch Size** = 32
- **Early Stop Patience** = 3
- **Optimizer** = Adam
- **Learning Rate** = 1e-3
- **Loss** = Categorical Cross-Entropy
- **Dropout** = 0.2
- **Convolution Activation Function** = ReLU
- **Fully Connected Activation Functions** = ReLU
- **Output Activation Function** = Softmax

**Table 3:** NC-DCNN.v0_ALT Parameter Breakdown

| | **Total Params** | **Trainable Params** | **Non-Trainable Params** |
|---|---|---|---|
| **No. of Params** | 31, 363 | 30, 755 | 608 |
| **Size** | 122.51 KB | 120.14 KB | 2.38 KB |

**Table 4:** NC-DCNN.v0_ALT Evaluation Metrics at EPOCH = 5

| | **Training** | **Validation** |
|---|---|---|
| **Accuracy** | 0.5515 | 0.3921 |
| **Loss** | 0.9681 | 1.1512 |

With NC-DCNN.v0_ALT, I discovered an issue where both the validation accuracy and loss metrics seemed to plateau from the beginning of the training process. It did not improve despite the training accuracy/loss improving. This told me that I was seeing an overfitting scenario; for whatever reason, the model was able to correctly guess the class of the training data ~55% of the time, but could not generalize enough to also be applicable to the validation data set. This overfitting is abundantly clear in Figure 6 below, and the plateau can be seen in Figure 5.

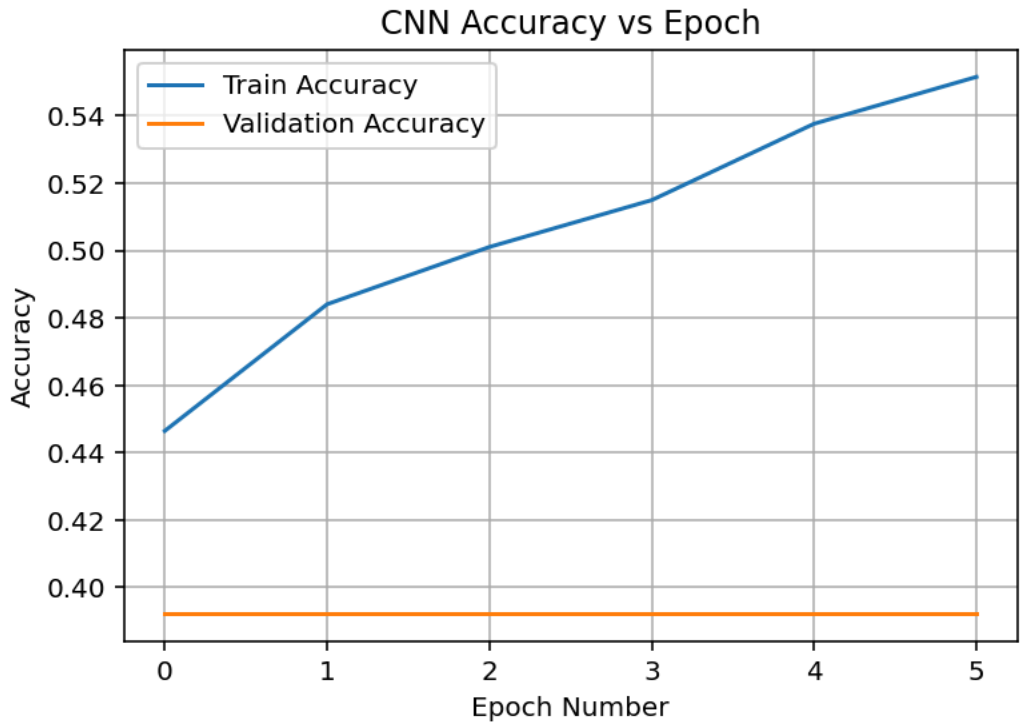


**Figure 5:** NC-DCNN.v0_ALT Accuracy of Training/Validation vs Epoch

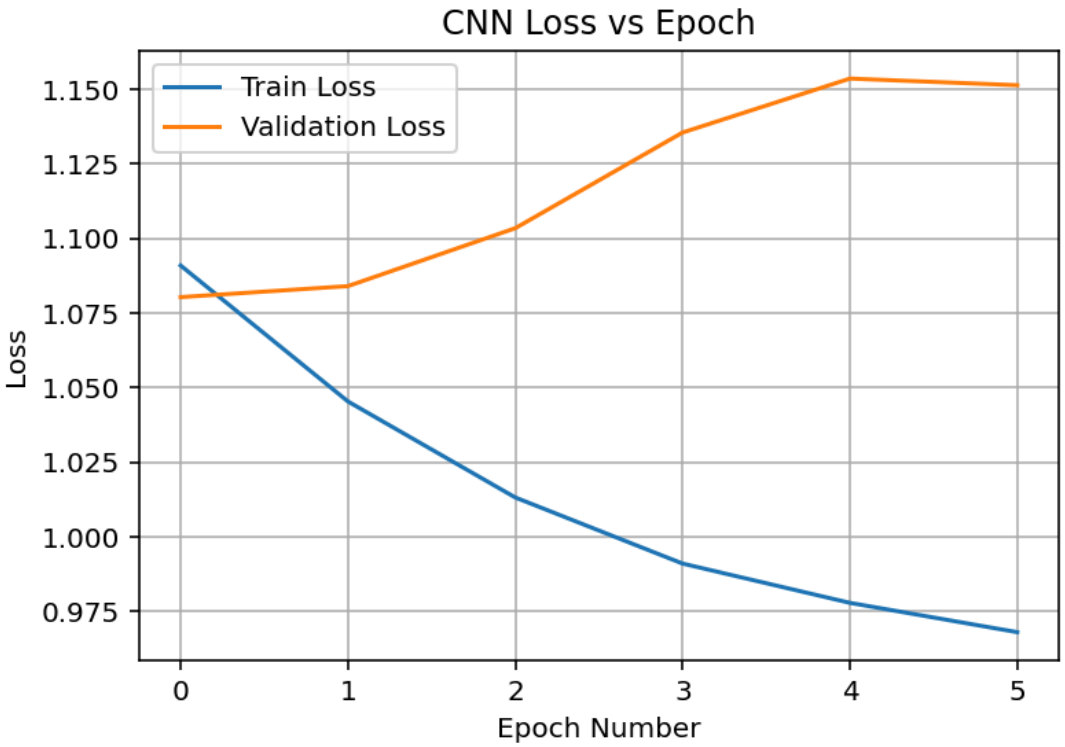


**Figure 6:** NC-DCNN.v0_ALT Accuracy of Training/Validation vs Epoch

## 2.3 - NC-DCNN.v0_PRIME

Knowing that NC-DCNN.v0_ALT was too complex of a model and resulted in overfitting, NC-DCNN.v0_PRIME will seek to make a much simpler model with only 3 convolutional layers, 2 of which use SeparableConv2D(), and GlobalAveragePooling2D() instead of the classic Flatten(). 1 dense layer followed up by a dropout layer and then the final output layer. See this below in Figure 7 below.

| Layer (type) | Output Shape | Param # |
|---|---|---|
| conv2d (Conv2D) | (None, 498, 498, 16) | 448 |
| max_pooling2d (MaxPooling2D) | (None, 249, 249, 16) | 0 |
| separable_conv2d (SeparableConv2D) | (None, 247, 247, 32) | 688 |
| max_pooling2d_1 (MaxPooling2D) | (None, 123, 123, 32) | 0 |
| separable_conv2d_1 (SeparableConv2D) | (None, 121, 121, 64) | 2,400 |
| max_pooling2d_2 (MaxPooling2D) | (None, 60, 60, 64) | 0 |
| global_average_pooling2d (GlobalAveragePooling2D) | (None, 64) | 0 |
| dense (Dense) | (None, 64) | 4,160 |
| dropout (Dropout) | (None, 64) | 0 |
| dense_1 (Dense) | (None, 3) | 195 |

**Figure 7:** NC-DCNN.v0_PRIME Architecture

It should also be noted that in the fully connected dense layer, we use a kernel regularizer using Tensorflows L2 Regularizer with a penalty of 1e-4, this is a measure taken to attempt to prevent overfitting. The L2 regularizer encoruages the model to use smaller weights as it punishes the loss function attributed to this layer (ReLU) proportional to the sum of squares of its weights.

**List of Key Design Parameters for NC-DCNN.v0_PRIME:**

- **Epochs** = 25 [Only made it to Epoch 5]
- **Batch Size** = 32
- **Early Stop Patience** = 3
- **Optimizer** = Adam
- **Learning Rate** = 1e-3
- **Loss** = Categorical Cross-Entropy
- **Dropout** = 0.5
- **Convolution Activation Function** = ReLU
- **Fully Connected Activation Functions** = ReLU
- **L2 Regularizer Penalty** = 1e-4
- **Output Activation Function** = Softmax

**Table 5:** NC-DCNN.v0_PRIME Parameter Breakdown

| | **Total Params** | **Trainable Params** | **Non-Trainable Params** |
|---|---|---|---|
| **No. of Params** | 7,891 | 7,891 | 0 |
| **Size** | 30.82 KB | 30.82 KB | 0.00 B |

**Table 6:** NC-DCNN.v0_PRIME Evaluation Metrics

| | **Training** | **Validation** |
|---|---|---|
| **Accuracy** | 0.3980 | 0.4617 |
| **Loss** | 1.0816 | 1.0539 |

NC-DCNN.v0_PRIME accomplished the goals of our experiment compared to NC-DCNN.v0_ALT in the sense that the validation scores did not stay flat and we successfully jumped the hurdle of overfitting. However, I think we jumped it by too much because the early stoppage function with a patience of 3 stopped the training of this model early at the 5th EPOCH. I believe this is due to underfitting, the model failed to increase validation scores by a meaningful amount in 3 EPOCHS and was terminated by the early stoppage function. It is possible that given more patience, the model would have converged to something more. See Figure 8 and Figure 9 below.

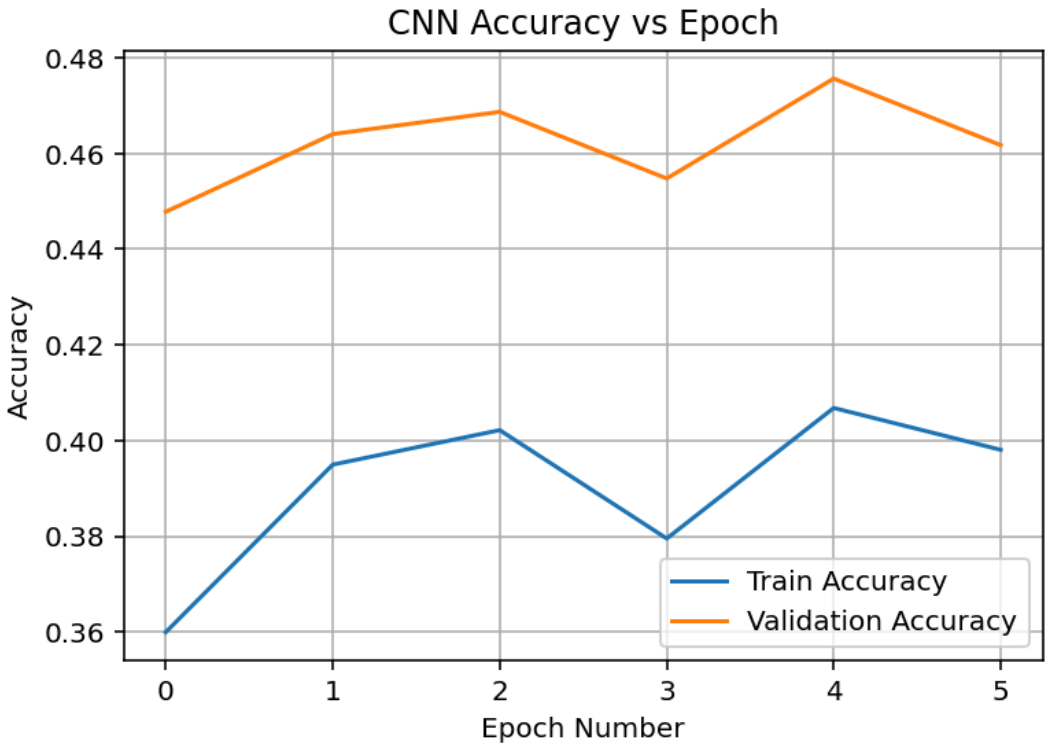


**Figure 8:** NC-DCNN.v0_PRIME Accuracy of Training/Validation vs Epoch

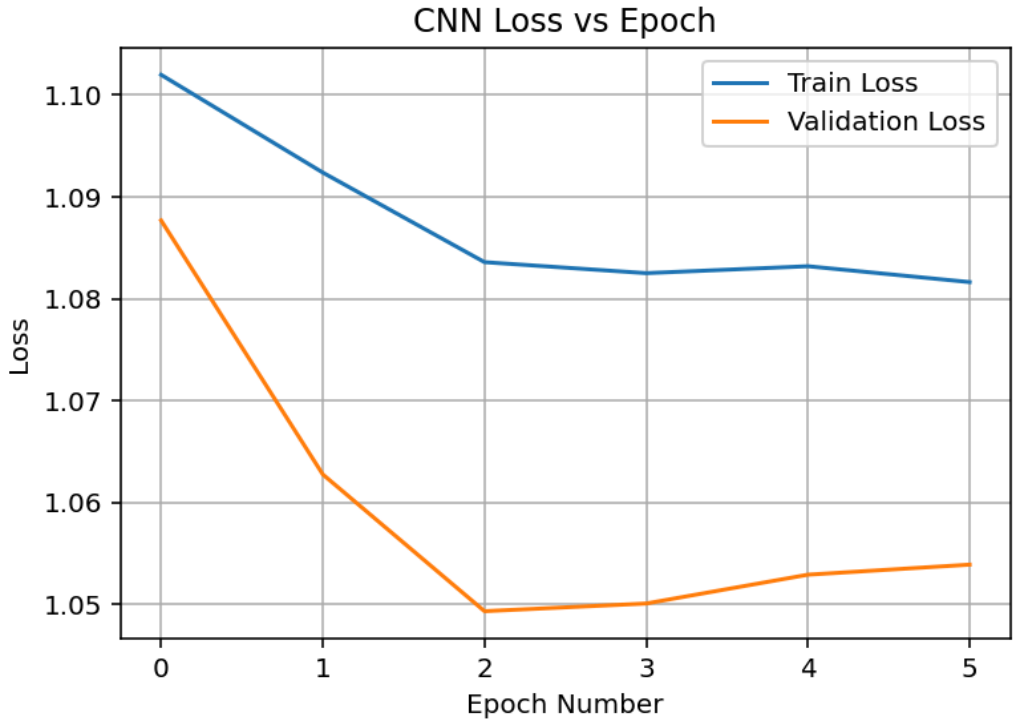


**Figure 9:** NC-DCNN.v0_PRIME Loss of Training/Validation vs Epoch

## 2.4 - NC-DCNN.v0_SECUNDUS

Due to a lot of issues of non convergence and constant overfitting, I decided to return back to the same functions that we use in the lecture notes such as Conv2D() in place of SeparableConv2D() and Flatten() instead of GlobalAveragePooling2D(). This resulted in better performance of the model. We decided to go with 4 convolutional layers and 3 dense layers (including the output layer) with dropouts of 0.25 after each. We continued to use L2 regularization in the first dense layer.

| Layer (type) | Output Shape | Param # |
|---|---|---|
| conv2d_6 (Conv2D) | (None, 498, 498, 16) | 448 |
| max_pooling2d_28 (MaxPooling2D) | (None, 249, 249, 16) | 0 |
| conv2d_7 (Conv2D) | (None, 247, 247, 32) | 4,640 |
| max_pooling2d_29 (MaxPooling2D) | (None, 123, 123, 32) | 0 |
| conv2d_8 (Conv2D) | (None, 121, 121, 64) | 18,496 |
| max_pooling2d_30 (MaxPooling2D) | (None, 60, 60, 64) | 0 |
| conv2d_9 (Conv2D) | (None, 58, 58, 128) | 73,856 |
| max_pooling2d_31 (MaxPooling2D) | (None, 29, 29, 128) | 0 |
| flatten_1 (Flatten) | (None, 107648) | 0 |
| dense_20 (Dense) | (None, 64) | 6,889,536 |
| dropout_13 (Dropout) | (None, 64) | 0 |
| dense_21 (Dense) | (None, 32) | 2,080 |
| dropout_14 (Dropout) | (None, 32) | 0 |
| dense_22 (Dense) | (None, 3) | 99 |

**Figure 10:** NC-DCNN.v0_SECUNDUS Architecture

**List of Key Design Parameters for NC-DCNN.v0_SECUNDUS:**

- **Epochs** = 25 [Only made it to Epoch 16]
- **Batch Size** = 32
- **Early Stop Patience** = 7
- **Optimizer** = Adam
- **Learning Rate** = 1e-3
- **Loss** = Categorical Cross-Entropy
- **Dropout** = 0.25
- **Convolution Activation Function** = ReLU
- **Fully Connected Activation Functions** = ReLU
- **Output Activation Function** = Softmax

The model hasd the smallest validation loss at the EPOCH = 8, and thus this was saved in the .KERAS file that will eventually be tested. Table 7 below shows the performance metrics and Figure 11 and Figure 12 show the point at which the model started to overfit well at EPOCH 7 and after.

**Table 7:** NC-DCNN.v0_SECUNDUS Evaluation Metrics at EPOCH = 8

| | **Training** | **Validation** |
|---|---|---|
| **Accuracy** | 0.6617 | 0.7352 |
| **Loss** | 0.7367 | 0.6914 |

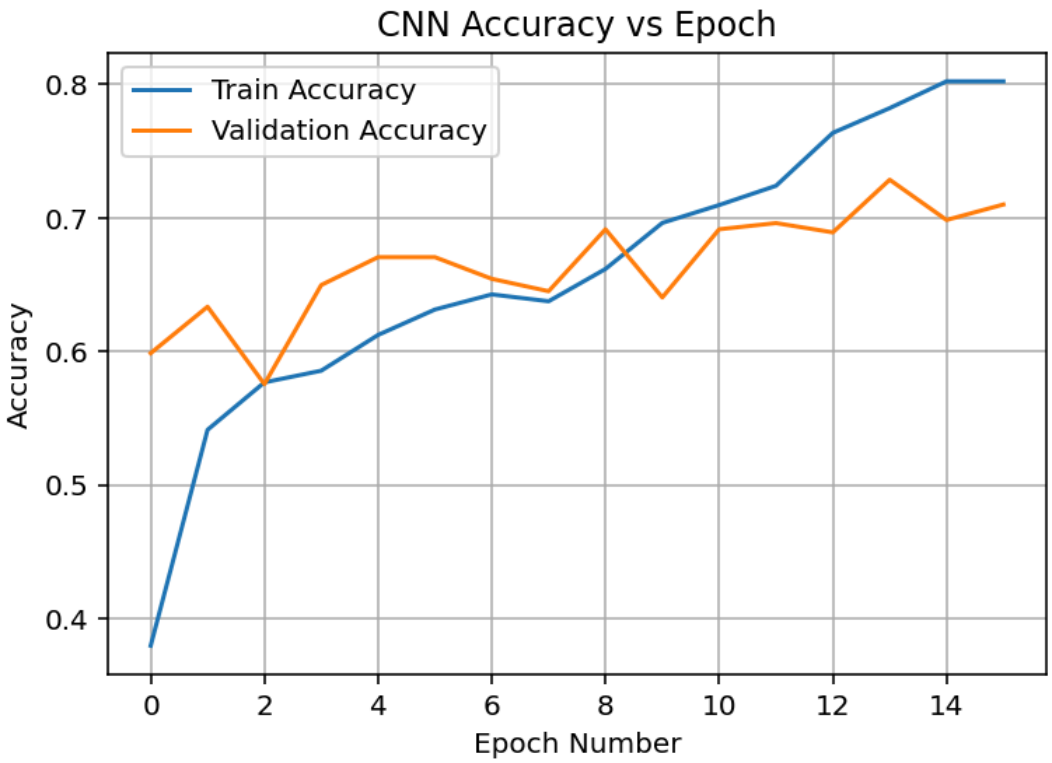


**Figure 11:** NC-DCNN.v0_SECUNDUS Accuracy of Training/Validation vs Epoch

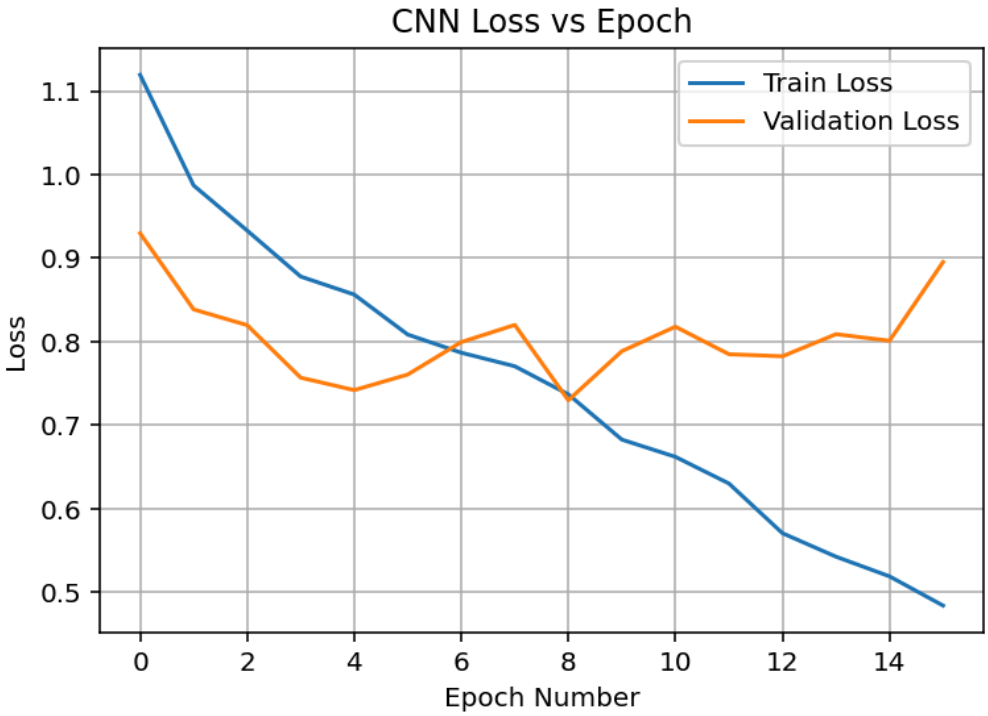


**Figure 12:** NC-DCNN.v0_SECUNDUS Loss of Training/Validation vs Epoch

## 2.4 - NC-DCNN.v0_TERTIUS

Now that a baseline model has been created with an adequate accuracy, let us now create our variant model which will decrease the complexity of its architecture by cutting each of the filters in half but changing the activation functions of all of the layers to ‘eLU’. It should also be noted that a different Optimizer was also used using ‘RMSprop’, which is supposedly better for some categorical classification tasks. It should also be noted that Dropout was increased by 0.10.

| Layer (type) | Output Shape | Param # |
|---|---|---|
| conv2d (Conv2D) | (None, 498, 498, 8) | 224 |
| max_pooling2d (MaxPooling2D) | (None, 249, 249, 8) | 0 |
| conv2d_1 (Conv2D) | (None, 247, 247, 16) | 1,168 |
| max_pooling2d_1 (MaxPooling2D) | (None, 123, 123, 16) | 0 |
| conv2d_2 (Conv2D) | (None, 121, 121, 32) | 4,640 |
| max_pooling2d_2 (MaxPooling2D) | (None, 60, 60, 32) | 0 |
| conv2d_3 (Conv2D) | (None, 58, 58, 64) | 18,496 |
| max_pooling2d_3 (MaxPooling2D) | (None, 29, 29, 64) | 0 |
| flatten (Flatten) | (None, 53824) | 0 |
| dense (Dense) | (None, 64) | 3,444,800 |
| dropout (Dropout) | (None, 64) | 0 |
| dense_1 (Dense) | (None, 32) | 2,080 |
| dropout_1 (Dropout) | (None, 32) | 0 |
| dense_2 (Dense) | (None, 3) | 99 |

**Figure 13:** NC-DCNN.v0_TERTIUS Architecture

**List of Key Design Parameters for NC-DCNN.v0_TERTIUS:**

- **Epochs** = 25 [Only made it to Epoch 15]
- **Batch Size** = 32
- **Early Stop Patience** = 7
- **Optimizer** = RMSprop
- **Learning Rate** = 1e-4
- **Loss** = Categorical Cross-Entropy
- **Convolution Activation Function** = eLU
- **Fully Connected Activation Functions** = eLU
- **Dropout** = 0.35
- **Output Activation Function** = Softmax

This model has the lowest validation loss at EPOCH = 7, which outputted marginally better scores in the training dataset but a little bit worse performance against the validation dataset. The model weights were saved to a .KERAS file. See Figure 14 and Figure 15 below for the performance as the EPOCHS went on.

**Table 8:** NC-DCNN.v0_TERTIUS Parameter Breakdown

| | **Total Params** | **Trainable Params** | **Non-Trainable Params** |
|---|---|---|---|
| **No. of Params** | 3,471,507 | 3,471,507 | 0 |
| **Size** | 13.24 MB | 13.24 MB | 0.00 B |

**Table 9:** NC-DCNN.v0_TERTIUS Evaluation Metrics at EPOCH = 7

| | **Training** | **Validation** |
|---|---|---|
| **Accuracy** | 0.7364 | 0.6636 |
| **Loss** | 0.6585 | 0.8076 |

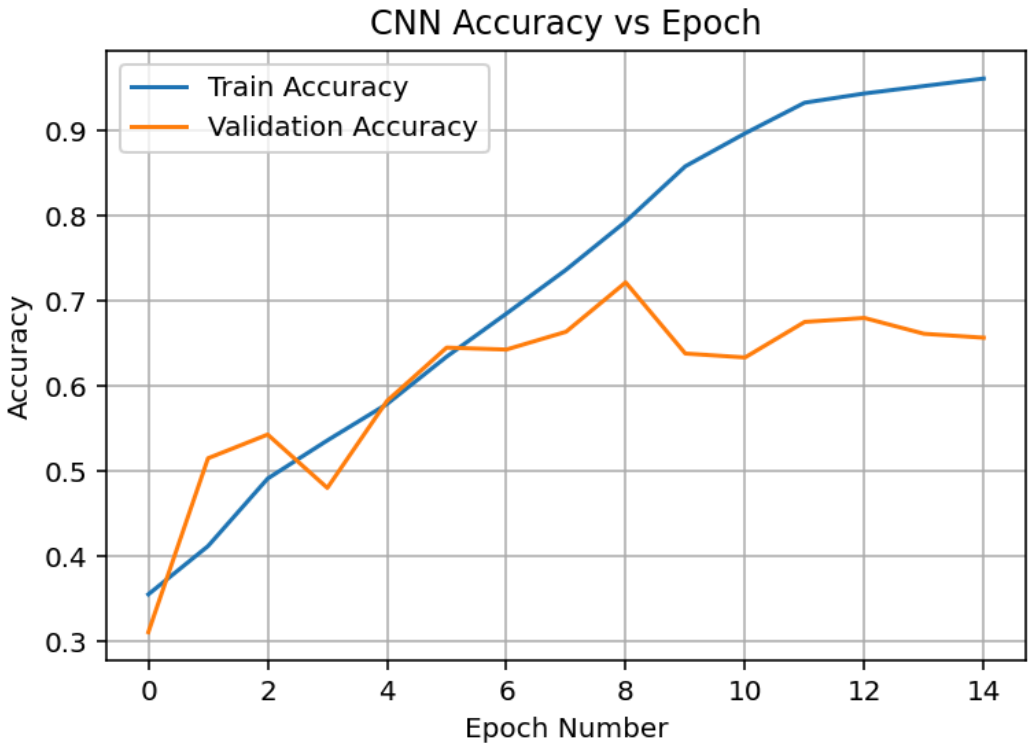


**Figure 14:** NC-DCNN.v0_TERTIUS Accuracy of Training/Validation vs Epoch

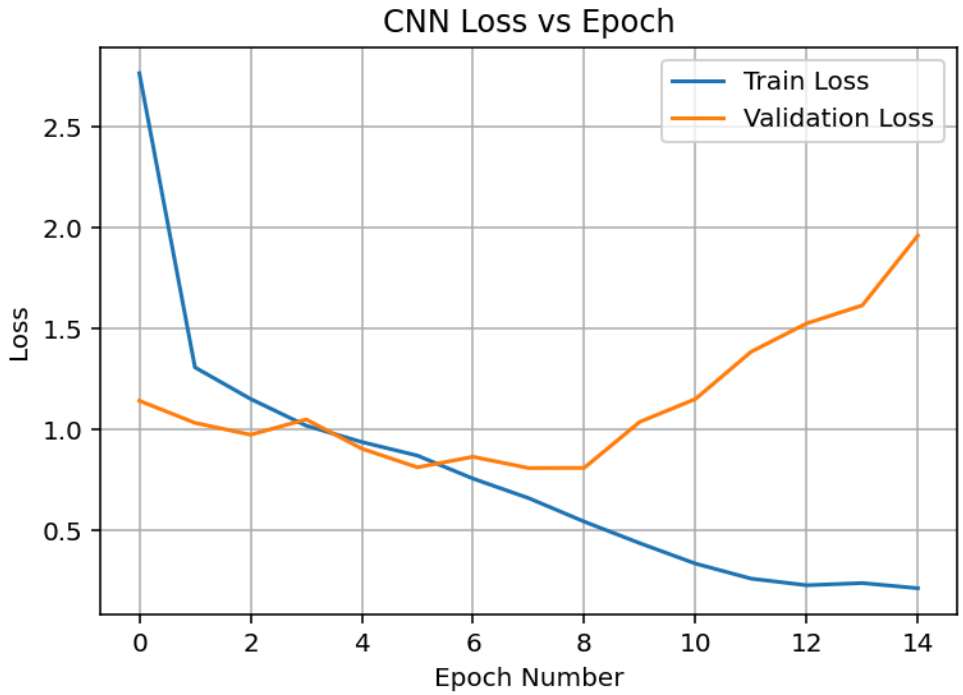


**Figure 15:** NC-DCNN.v0_TERTIUS Loss of Training/Validation vs Epoch

# 3.0 - Model Testing

Now it comes time to test the models against the testing dataset.

**Table 10:** Baseline and Variant Model Accuracies

| Model | Accuracy |
|---|---|
| **Baseline Model** | 0.6549 |
| **Variant Model** | 0.6549 |

I had to double check the results of Table 10 many times in my code, but I believe it is correctly telling me the accuracies of both models. Since the models were so similar in performance, it seems that both models got the same results in the accuracy test by getting 353 out of the 539 total test images correct. This is about ⅔ of the time, better than absolute random but there is still lots of room for improvement.

**Table 11:** Baseline Model Per Class Accuracies

| **Baseline Model - NC-DCNN.v0_SECUNDUS** | |
|---|---|
| **Crack** | 0.7014 |
| **Missing Head** | 0.6950 |
| **Paint Off** | 0.5156 |

**Table 12:** Variant Model Per Class Accuracies

| **Variant Model - NC-DCNN.v0_TERTIUS** | |
|---|---|
| **Crack** | 0.6209 |
| **Missing Head** | 0.7450 |
| **Paint Off** | 0.5703 |

Table 11 and Table 12 above showcase the per-class score and we do in fact prove that our model performances are different in some regard. We can see that the baseline model is better at predicting cracks but worse at predicing missing heads and paint offs than the variant model. It seems that for both models, they struggled the most with paint offs.

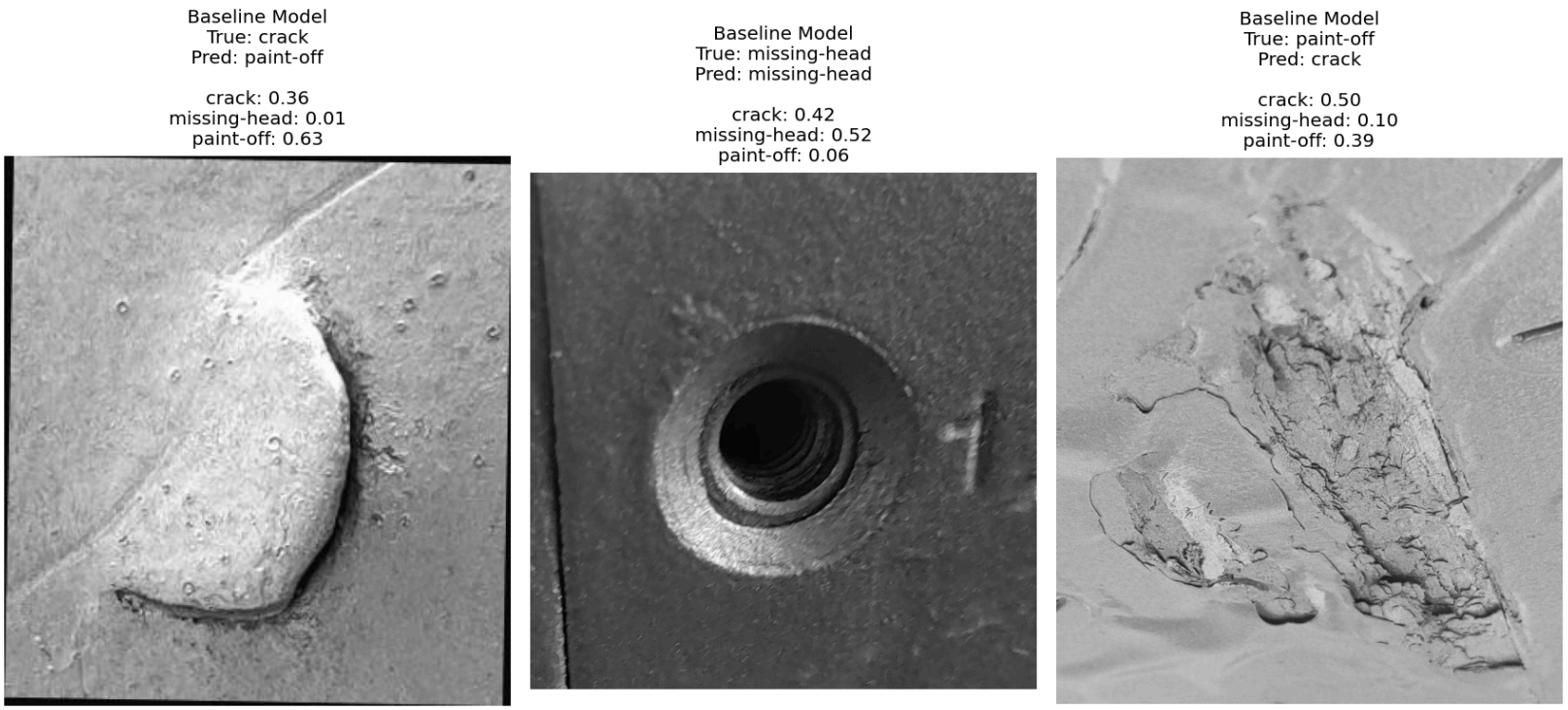


**Figure 16:** Final Test Images of Baseline Model

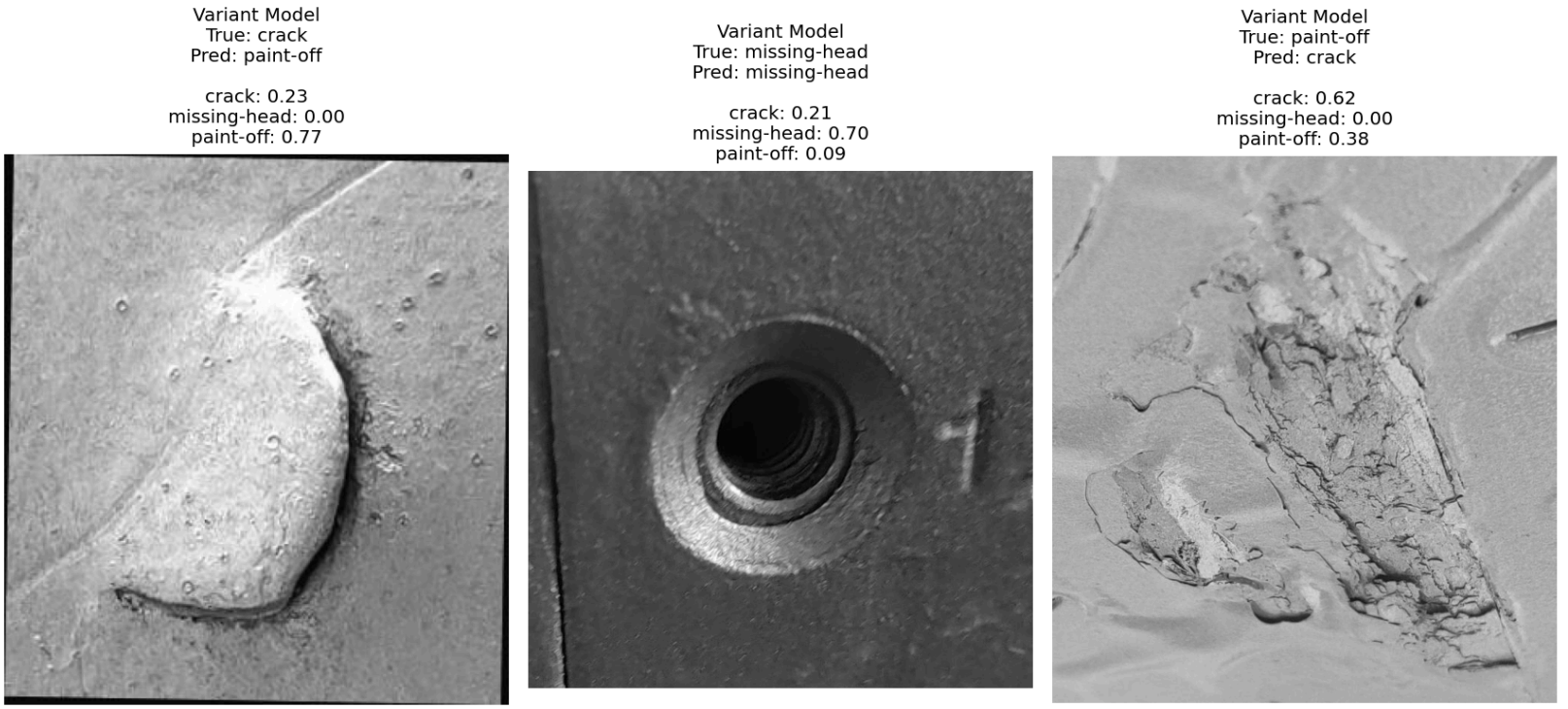


**Figure 17:** Final Test Images of Variant Model

Figure 16 and Figure 17 above showcase the performance of each of the models on the testing images, both of the models incorrectly predict the crack and paint off, except the variant model more confidently gets them incorrect. These models both have a lot of performance issues and can be fine tuned further, but in this case, it appears that the baseline model is the best even despite the activation function and optimizer changes for the variant. It should also be noted that the baseline model had more filters and neurons than the variant.

This concludes my report.