

# *Understanding the Effects of Distortion on Object Detection Models*

Alex Cohen  
DATS 6501

## ***Abstract***

While much work has been done to understand the effect of distorted visual input on image classification models, there has been little exploration of the same effects on object detection models. This study attempts to understand the impact of varying degrees of two common types of distortion (image blur and noise injection) on the ability to identify objects in the Microsoft Common Objects in Context dataset. This experiment involves applying distortions of two types and three different magnitudes and observing the changes in object detection task performance across different object sizes. Additionally, this study examines the impact that targeted fine-tuning has on the Faster R-CNN object detection model, and quantifies the improvements achieved by focusing on the feature extractor, Region Proposal Network and pooling layers, and overall model. This study shows immediate decreases in performance with the slightest injection of distortion, highlighting the limited robustness of common pretrained object detection models, and identifies the feature extraction layers as the component most easily fine-tuned to mitigate this decrease in performance at higher levels of distortion.

## ***Introduction***

From processing medical scans to look for the presence of diseases, reading license plates to automatically pay tolls, or reading handwriting to determine where a package should be sent, using computers to perform typically human-based vision tasks has seen an explosion of applications in the past few years. This field, attempting to replace visually based tasks typically performed by humans by machines capable of producing similar success metrics, is often referred to as computer vision. Early computer vision efforts began in the late 1960s with the desire to extract the three-dimensional structure from images, which served as the foundation for modern computer vision. However, this work was often limited by the constraints on computation power and available data. This changed in the early 2000s and 2010s with developments in deep learning and computing capabilities.

Computer vision work reached a turning point in the early 2010s with the introduction of deep convolutional neural networks in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC). The innovation was the use of a common, large-scale, and easily accessible set of everyday images from which image classifiers could be trained and evaluated. This challenge, which ended in 2017, pushed the idea that the data on which models are trained is just as important as the models themselves.<sup>[1]</sup> Parallel to this change, the entry of the AlexNet model in the 2012 competition spurred renewed interest in deep learning for image classification. AlexNet far outpaced the rest of the competition, winning the 2012 ImageNet challenge and producing an error rate over 10% lower than the second-place competitor.<sup>[2]</sup> With the demonstrated improvement that deep learning models could provide in visual tasks, the fields of computer vision, image classification, and object detection have seen tremendous strides in the sophistication of models, and the resulting capabilities in mirroring human performance in image-related tasks.

As the fields of computer vision and object detection continue to progress, much progress has been made on object detection models. From the R-CNN family of models to SSDs like the YOLO family of models, object detection has now become a nearly real-time capability.<sup>[3]</sup> However, little work has been done to date on how changes to the input images, specifically distortion, effect model performance. From faulty or cheap cameras, corrupted files, extreme temperatures, or just poor data processing, most images cannot be expected to be high resolution, as is often the case with training data. Distortion is likely to occur in images when used in real-time applications, with blur and noise being the most likely culprits. Blur is likely to occur when attempting to track a moving object or having an unfocused camera; noise is likely to occur with high temperatures, extreme lighting values, or corrupted video files. These potentially performance-inhibiting situations may be common when attempting to identify and or track objects in non-academic settings, however little work has been done to understand the effects these scenarios may cause on direct object detection capabilities. For this reason, this project attempts to explore the impact that these forms of distortion have on object detection model performance and identify potential techniques that mitigate the effect of distortion on object detection models.

### ***Problem Statement***

**How do different forms of input distortion, specifically noise injection and blur addition, impact performance in object detection models, and can different components of this model be refined to better respond to distorted input?**

### ***Related Work***

Prior work has been done using distortion when training deep neural networks for image classification tasks. In Deep Learning with Noise<sup>[4]</sup> Lou and Yang showed that more complex neural networks are better able to absorb input noise when comparing small noise injections into smaller machine learning classifiers – including Logistic Regression, a Multi-Layer Perceptron (MLP), and Convolutional Neural Networks. However, all models were better able to absorb this noise when it was injected earlier in the model, as opposed to distorting later feature maps or feature vectors.

While deep neural networks (DNNs) can perform many image classification and detection tasks at near-human levels of performance, Nguyen et al. have shown that they can be easily fooled and images can be created that are unidentifiable to humans, yet machines can classify them with over 99% confidence.<sup>[5]</sup> Using an evolutionary algorithm, they were able to create images that the ImageNet CNN was extremely confident about, yet humans could not recognize. While they were studying the kinds of images a computer could recognize that a human cannot, other projects have explored the opposite – what distorted images can a human easily identify that a computer has difficulty classifying.

A major study done on the effect that distortion has on image classification was performed by Dodge and Karam in 2016.<sup>[6]</sup> This paper looked to explore how susceptible various common CNN architectures are to distortions such as gaussian blur, noise injection, varying contrast levels, and JPEG image quality. They found that all CNN models in the experiment performed noticeably poorer with these higher levels of

distortion injection, with blur and noise injection seemingly having the most severe impacts. However, this paper only looked at the effect of distortion on image classifiers, not techniques to improve model performance.

One of the most comprehensive studies of image classification with distortion, upon which this project is an expansion, is the paper ON CLASSIFICATION OF DISTORTED IMAGES WITH DEEP CONVOLUTIONAL NEURAL NETWORKS by Zhou et al.<sup>[7]</sup> This paper looked to understand the impact that motion blur and gaussian noise injection have on the ability to classify images, as well as develop mitigation strategies to improve model performance when predicting on distorted images. The main difference between the prior work and this project, however, is that Zhou et al. studied the effect of distortion on image classification models; this project studies the effect of distortion on object detection models.

Some work has been done on adapting object detection models to non-traditional inputs. Wang and Lai studied the application of the Faster R-CNN model on 360° images and curved space, finding that traditional object detection methods do not perform as well and must be adapted to effectively function in curved images.<sup>[8]</sup> Yang et al. additionally investigated the effect of distortion on object detection, however in the context of pre-processing methods and attempting to identify and correct distorted images prior to prediction.<sup>[9]</sup> This technique has even been extended to distortion-correction in video, in which a technique is developed to identify image frames containing a moving object from a distorted video stream, in a paper by Uchida, Yamashita, and Asama.<sup>[10]</sup> Another relevant research effort, conducted by Hamzeh, El-Shair, and Rawashdeh, attempts to quantify the effect of rain and rain drops on the performance of object detection models in a self-driving car application, evaluating the amount of distortion caused by rain and the corresponding decrease in detector performance.<sup>[11]</sup> Further work has been conducted involving how potential training imbalances may effect Faster R-CNN performance. Chen et. al. showed that image-level and object-level class rebalancing can lead to more robust performance results<sup>[12]</sup>.

Finally, the Faster R-CNN model has been extended to detecting object masks using the Mask R-CNN, which adds a branch for predicting an object mask in addition to the bounding box regression and object classification<sup>[13]</sup>. This extension allows for the further localization of an object within a bounding box, predicting the specific outline instead of simply the occupied region of space.

While additional work has been done to both understand the impact of distortion on object detection models and further explore the capabilities of the Faster R-CNN model, this project is a novel effort to bridge the two areas.

### ***Description of the Dataset***

The images all come from the Microsoft Common Objects in Context (COCO) dataset, which was developed specifically to advance the state-of-the-art in object recognition by placing objects in their everyday scenes.<sup>[14]</sup> The entire dataset contains over 330,000 images and 1.5 million object instances across over 80 different object categories. The specific categories used in this project include:

*person, bicycle, car, motorcycle, airplane, bus,*

*train, truck, boat, traffic light, fire hydrant, stop sign,  
parking meter, bench, bird, cat, dog, horse, sheep, cow,  
elephant, bear, zebra, giraffe, backpack, umbrella,  
handbag, tie, suitcase, frisbee, skis, snowboard, sports ball,  
kite, baseball bat, baseball glove, skateboard, surfboard, tennis racket,  
bottle, wine glass, cup, fork, knife, spoon, bowl,  
banana, apple, sandwich, orange, broccoli, carrot, hot dog, pizza,  
donut, cake, chair, couch, potted plant, bed, dining table,  
toilet, tv, laptop, mouse, remote, keyboard, cell phone,  
microwave, oven, toaster, sink, refrigerator, book,  
clock, vase, scissors, teddy bear, hair drier, toothbrush*

#### COCO Object Categories

For this specific project, only the 2017 Train and Val image sets were used. The Train image set consists of 141,000 images, and the Val image set consists of 5,000 images. The Val set is used to determine the baseline object detection scores, the detection scores after distortions, and the scores on both unaltered and distorted images when using a fine-tuned model. The Train image set is used for refining the pretrained object detection models, with subsets of various sizes being taken to determine the optimal training parameters.

To locate and classify the objects with the images, the COCO dataset comes with accompanying JSON files with image annotations. These annotation files contain additional information about each image in the dataset, including the following fields:

*id, width, height, file\_name, license, flickr\_url, coco\_url,  
date\_captured, image\_id, category\_id, segmentation,  
area, bbox, iscrowd*

#### List of key COCO annotation fields

The main fields being used in this analysis are the *id*, *category\_id*, and *bbox* information. The *id* is used to maintain a constant identifier across the various programs. The *category\_id* is used in combination with the list of object categories above to identify the contents of a specific annotation. The *bbox* field is used to identify the specific pixel location within the image for the following values [*x1, y1, width, height*], which can be easily converted to [*x1, y1, x2, y2*] format as well. These annotations are used to both generate input for the model during training and evaluation or passed to the evaluator class when calculating summary metrics.



Figure 1 - Example COCO Image with Ground Truth Object Boxes

## Model Overview

The model used for this experiment is the PyTorch Faster R-CNN implementation, originally discussed by Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun in their paper *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*.<sup>[15]</sup> This work builds upon a series of object detection models, beginning with the R-CNN network architecture developed by Girshick et al. in 2014.<sup>[16]</sup> This initial model worked by using selective search to create around 2,000 proposed regions believed to contain an object, computing features for each of the proposed regions using a large convolutional neural network, then classifying the features using class-specific SVM models. This method, however, was slow to train and could not be used to evaluate in near real-time, so in 2015 Girshick et al. developed the Fast R-CNN.<sup>[17]</sup> The difference in this new model is that the computation of features using a CNN occurred only once, at the beginning of the training or evaluation process. Region proposals and object classes were then computed on top of this feature map, meaning that feature maps were not being calculated for each proposed region, which led to a significantly faster training and evaluation time.

The Fast R-CNN was improved again by Ren et al. in 2016, further reducing training and evaluation times. The main improvement was moving away from the use of a selective search algorithm to generate the proposed object regions and moving to a Region Proposal Network that is learned to predict the likely object regions. This process allows for object detection that is 30x faster than the Fast R-CNN, improving from 0.5 frames per second to 17 frames per second when using the ZF Net backbone.<sup>[9]</sup>

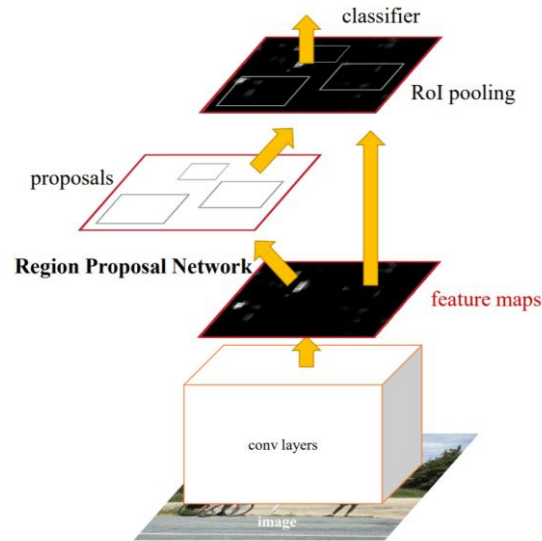


Figure 2 - Faster R-CNN Model Architecture <sup>[13]</sup>

The model is composed of multiple different components – the feature map backbone, the Region Proposal Network, and the RoI pooling/output layer, that all function together to create the Faster R-CNN. The backbone in the PyTorch implementation is a ResNet-50<sup>18</sup>, as shown below.

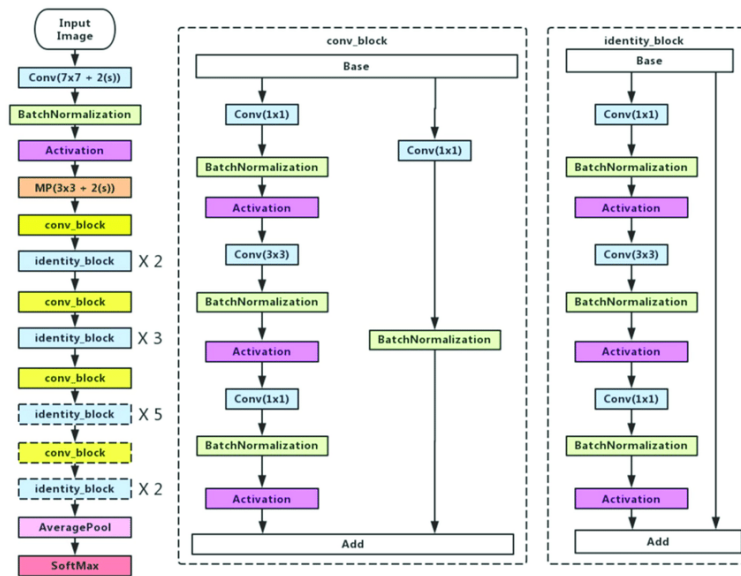


Figure 3 - Traditional ResNet50 Model Architecture <sup>[19]</sup>

This ResNet model serves to transform the input image into a series of feature maps, that can then be used in the Region Proposal Network to predict object classes and bounding box locations. The ResNet50 is composed of a few standard convolutional neural network layers, such as a Convolutional layer ( $n \times n$ ), batch normalization layers, max pooling and average pooling layers, and activation layers. What makes the ResNet50 (and the entire ResNet family of models) unique is the use of a specific convolutional block of neurons and layers, called both the identity and convolution blocks. Instead of using the convolutional

layers to transform the input feature maps directly into new feature maps, the layers are trained to compute the transformations between the original feature maps and the final feature maps, and then add this difference to the original set of features to create new feature maps. In a sense, the model is computing and optimizing the residual between the input and final feature map, as opposed to directly transforming the input feature maps, hence the name ResNet.<sup>[20]</sup> This ResNet backbone is used to generate the feature maps for use in the Region Proposal Network (RPN).

The RPN is composed of two parts: the classifier and the regressor, which are used to predict the “objectness” of a proposed region, and the adjustments to the bounding box coordinates needed to define a certain segment of the image. This is done in a series of steps. The first step is to generate a series of anchor points in the image, which will serve as the center of the proposed regions. These anchor points then have multiple predicted boxes placed over top at varying aspect ratios (1:1, 2:1, 1:2) and sizes (128, 256, 512). The anchor boxes are then checked against the ground truth boxes to determine if there is an object present (calculating the IoU of the anchor box with the ground truth box). Those with the highest IoU scores (in the original paper this is defined as any box with an IoU above 0.7, which are further subset using non-maximum suppression, then reduced to the top 2000 proposals) are considered regions of interest, and then passed to the Region of Interest (RoI) pooling layer.

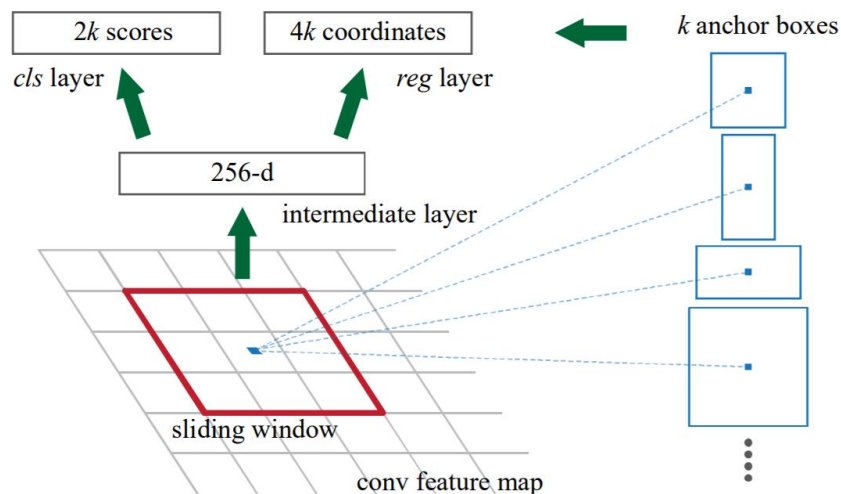


Figure 4 - Region Proposal Network Structure<sup>[13]</sup>

The RoI pooling layer takes the proposed regions of interest generated in the RPN and the feature maps generated by the backbone network to make predictions about the underlying objects. The pooling layer takes the section of the feature map that corresponds with the proposed region of interest, and then breaks that section of the feature map into  $n \times n$  boxes. From each of these boxes, the maximum value is taken – in essence, it is a max-pooling layer applied to specific regions of interest with an output of a constant size. This fixed-sized output can be passed to the final fully-connected layers before the final layers are used to classify an object within a proposed region and determine any necessary adjustments to the bounding box coordinates.



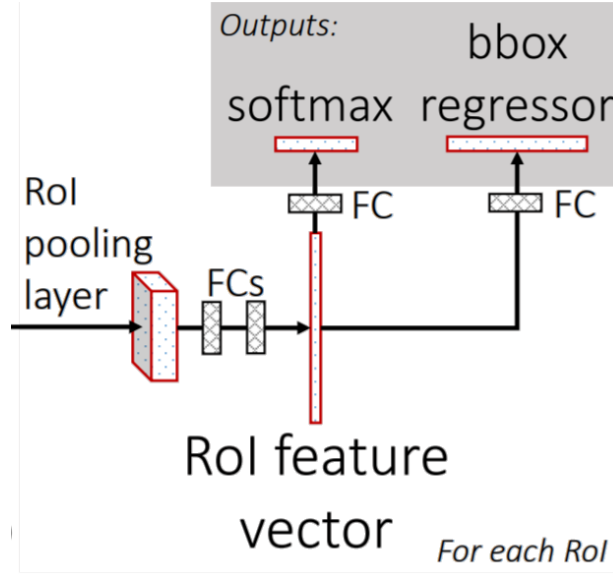


Figure 5 - Fast R-CNN Model Head <sup>[15]</sup>

The region proposal network is trained with the loss function seen below, which is a weighted combination of the classification error and the regression error on bounding box locations.

$$L(\{p_i\}, \{t_i\}) = \frac{1}{N_{cls}} \sum_i L_{cls}(p_i, p_i^*) + \lambda \frac{1}{N_{reg}} \sum_i p_i^* L_{reg}(t_i, t_i^*).$$

Figure 6 - RPN Loss Function <sup>[13]</sup>

This multi-task loss function was first used in the Fast R-CNN model, and is a weighted combination of the average classification loss and the inflated regression loss. In the Fast R-CNN and Faster R-CNN model, the  $L_{cls}(p_i, p_i^*)$  component is the log-loss (or categorical cross-entropy) of the likelihood of anchor box  $i$  being an object of each predicted class with the actual ground truth “objectness” of anchor box  $i$ . The regression loss is a multiplication of  $p_i^*$  (in order to only activate the regression loss for positive object anchors – i.e. not the image background) and the  $L_{reg}(t_i, t_i^*)$  regression component, which is the smooth  $L_1$  loss over the four coordinates of the bounding box offsets. Each term is then normalized by the number of anchor locations (during regression), as well as the number of images (during classification). The  $\lambda$  represents a weighting factor, as there are more bounding box coordinates than there are images during training. In the Faster R-CNN paper, this weight factor was set so  $\lambda=10$ , as the minibatch size of 256 was approximately 10x smaller than the number of anchor boxes generated (2400). This multi-task loss function allowed for the Region Proposal Network to be trained using stochastic gradient descent (SGD).

This multi-task loss function functions similarly to the loss calculated for the final object class prediction and bounding box regression tasks performed after the ROI pooling calculations. Two additional losses are calculated, the classification loss when predicting the object class within a region of interest, as well as the bounding box regression coordinates for the final object bounding box coordinates. This process



uses the same multi-task loss function as the region proposal network, computing losses for both the classification error (using log-loss or categorical cross-entropy), however the difference being the prediction of the specific object class as opposed to the “objectness” of the object within the bounding box. The regression component operates in a similar manner across both the RPN and final classification/regression layers.

This model architecture creates four individual losses which are then back-propagated through the network:

- Negative Log Likelihood: classification error of final class predictions (identified as 'loss\_classifier')
- Kullback-Leibler Divergence: error of bounding box regression in final predictions (identified as 'loss\_box\_reg')
- Binary Cross Entropy with Logits: classification error of “objectness” of RPN bounding box proposal (identified as 'loss\_objectness')
- Kullback-Leibler Divergence: error of bounding box regression in RPN (identified as 'loss\_rpn\_box\_reg')

## *Methodology*

There are two phases to this project: the exploration and measurement of baseline values, and then the attempt to improve upon those metrics using transfer learning to fine-tune and improve model performance on distorted images.

The initial phase of baseline performance measurement requires passing images with varying levels of distortion to the pretrained Faster R-CNN model, and capturing the mAP across all validation pictures. The specific images used for this task were the VAL 2017 set of Microsoft COCO images – containing 5,000 pictures. The first file used in this workflow is the `gen_predictions.py` python file, which contains the prediction component of the project. The first step in the pipeline is creating a dataset object that would grab each individual image present in the provided file path, load the validation annotations (the ground truth object classes and bounding box locations), convert the image to a tensor object for GPU computations, and return a dictionary containing the image array and the image ID. This dataset object could also be passed the `AddNoise` or `AddBlur` classes to inject distortions, which will be further discussed below. A PyTorch data loader object is then used to iterate through each of the 5,000 images in the dataset – passing them iteratively to the pretrained Faster R-CNN model for prediction which will return a set of predicted classes, bounding boxes, and confidence scores for each object in the image. This output would be transformed into a dictionary and appended to the existing output dictionary. Once the model iterated through each of the 5,000 images, the dictionary would be written as a JSON object for use in the `evaluate_coco.py` script.

The `AddNoise` and `AddBlur` classes, which are located in the `data_loaders.py` file, exist to inject distortion into the images prior to being passed to the model for prediction. The `AddBlur` transformation is more straightforward and takes an input image tensor and applies gaussian blur to the image to increase the level of distortion. Gaussian blur works by replacing the value of each pixel with a weighted average of the pixels within a given radius, decreasing in weight as you move farther from the center pixel out to the

edge of the affected area (determined by the radius) based on the standard deviation of the pixel values within the range. This serves to “blur” the image by reducing the sharp edges between pixels, since now those color channels are smoothed by neighboring values during the weighted average computation.

The AddNoise class applies a mask of random pixel value fluctuations to the original input image. First, the input image is passed as a three-dimensional array ( $w \times h \times \text{channels}$ ). From there, three equally sized masks are created using a numpy random normal array of size  $w \times h$ , mean of zero, and standard deviation of  $\sigma$ , and a mask is added to each of the original image channels. To avoid overflow (since colors can only have values between 0 and 255), if the resulting sum for each pixel is less than zero or more than 255, the addition is flipped to instead subtract the pixel mask value, effectively “clipping” the noise injection. This limits the resulting image pixel values to be between 0 and 255, which still providing pixel updates of the same magnitude. This updated image, with injected pixel variation, is then passed back to the data loader as a distorted image.

For both the AddNoise and AddBlur transformations, the user gets to define the amount of noise injected into the image by controlling the radius of the gaussian blur (when using the AddBlur transformation) or the standard deviation of the random pixel mask applied to the image (when using the AddNoise transformation). The radii used to generate the varying levels of blur were 1 px (low), 2 px (medium) and 5 px (high); the standard deviation of pixel values used to generate the varying levels of noise were 5 (low), 10 (medium), and 25 (high). When reporting results, the subscript used records the specific level of distortion used (e.g. `blur05` represents a blur distortion with a radius of 5 px, `noise25` represents a noise injection with a mask standard deviation of 25). Example transformations can be seen below.



Original Image



Blur<sub>01</sub>



Blur<sub>02</sub>



Blur<sub>05</sub>



Noise<sub>05</sub>



Noise<sub>10</sub>

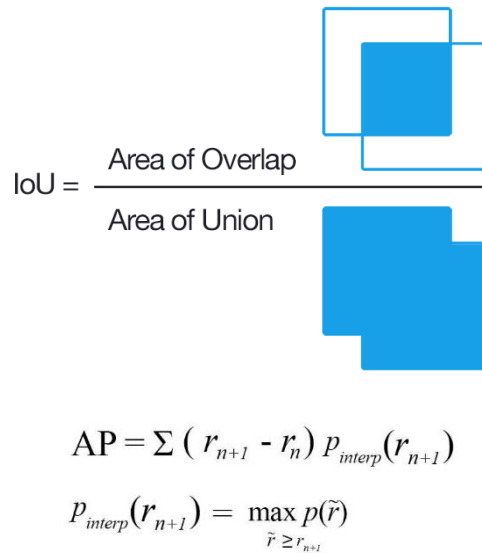


Noise<sub>25</sub>

Figure 7 - Examples of Image Distortion

The `gen_predictions.py` file takes a given model and cycles through all 5000 validation images to generate predicted bounding boxes and object classes. After generating the predictions, it transforms them to a dictionary, and this output dictionary is then saved as a JSON object for use in the evaluation stage. The `evaluate_coco.py` script uses this JSON prediction file created in the `gen_predictions.py` script and the ground-truth annotation file to calculate the precision and recall metrics for the model, as well as the mAP calculations (which are discussed further below). The first step is to create a COCO object and initialize it with the ground truth file. Next, a COCO object is created for the results file, utilizing a similar process. Finally, an evaluator class is created and passed the ground truth object and the prediction object to calculate the precision and recall values for the model, providing metrics regarding its object detection capabilities.

Determining results involves calculating the Average Precision of object detection and classification tasks for the given model. The metric used to score these results is Average Precision (or mean Average Precision – mAP), which is the metric by which object detection models are scored in annual COCO challenge. This process involves calculating the precision and recall of the predicted bounding boxes across differing areas of overlap, calculated using the Intersect over Union (IoU) metric. The IoU metric required for inclusion was varied between 0.50 and 0.95 at increments of 0.05, meaning there are 10 categories of predictions across all 90 object categories (not counting the background class) that are all averaged to produce the results. The precision-recall curve is then interpolated across different values, meaning that a single recall value is replaced with the maximum recall value to the “right” in the precision-recall curve. This serves to “smoothen” the precision-recall curve across the 101 different regions (for each percentile in the curve). The average precision for a given class and IoU threshold is then calculated by adding the width of a percentile (0.01) multiplied by the interpolated precision at that percentile across the precision-recall curve. The mean average precision is then this AP score across all classes and IoU thresholds.<sup>21</sup>



$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

$$\text{AP} = \sum (r_{n+1} - r_n) p_{\text{interp}}(r_{n+1})$$

$$p_{\text{interp}}(r_{n+1}) = \max_{\tilde{r} \geq r_{n+1}} p(\tilde{r})$$

Figure 8 - IoU and mAP calculations

Once the baseline metrics are calculated for varying levels of noise injection on the pretrained model, it was then fine-tuned in order to see if additional training on blurred and noise-injected images would

improve object detection capabilities when attempting to detect objects in distorted input. This was done in the `train_fasterRCNN_distort.py` script, which facilitates the actual model training/fine-tuning. Different components of the model would have their associated weights frozen during training, to limit weight updates to specific layers within the model and better understand component effects. This was done by creating a dataset object which would grab training images from the provided directory, apply the transformations passed to the object (including injecting noise and blur), and return each image as a tensor. Unlike during validation, each image had to be resized so that a minibatch size greater than 1 could be used during training. Each image was resized to 224 x 224, and bounding boxes were reshaped and moved accordingly during the data loading process. The pretrained Faster R-CNN model was then loaded, and the distorted training images would be used to update model weights using stochastic gradient descent and the loss function described in the model overview section. Various combination of hyperparameters were used to differing levels of success – which are discussed further in the results section. Once the additional model training was completed, the model weights would then be saved for use in the `gen_predictions.py` and `evaluate_coco.py` files, as outlined above. The results of the pretrained model and the model with additional training on distorted images would then be compared to determine if the baseline model performance can be improved.

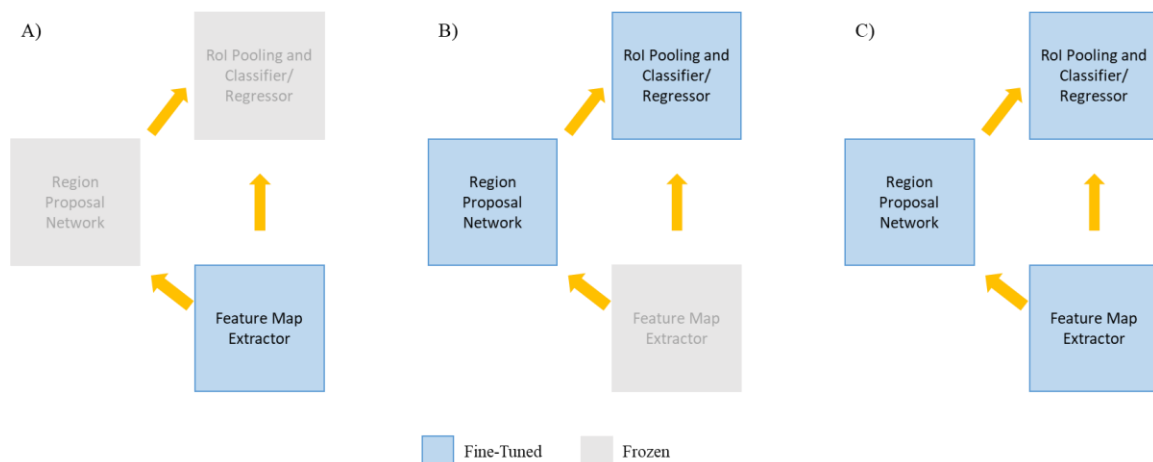


Figure 9 - Fine-Tuning Methodology including: A) Isolated Feature Extractor B) Isolated RPN/Model Head C) Entire Model

The hyperparameters for model training are as follows:

- Batchsize: 4
- Number of training batches: 500 (specific deviations are highlighted in the results section)
- Total images used: 2000 (specific deviations highlighted in the results section)
- Optimizer: Stochastic Gradient Decent
- Gradient Clipping Factor: 0.5
- Learning Rate: 0.005
- Momentum: 0.9
- Epochs: 10

All training and evaluation is done on an AWS EC2 g3.4xlarge instance, which has an associated GPU and 16 vCPUs to further improve training times. Throughout the code, many computations were performed

on this GPU in order to reduce code run-time, and a GPU and instance with a large amount of memory (8GiB GPU memory in the g3.4xlarge) should be used due to image sizes and volume.

## Results

The evaluation methods described above are further broken down by object size, measured in the number of pixels in the image the object occupies. All four of these metrics (average precision, and average precision across small, medium, and large objects) are reported for the baseline model (without any fine tuning or additional training) across the undistorted images, and images distorted varying amounts.

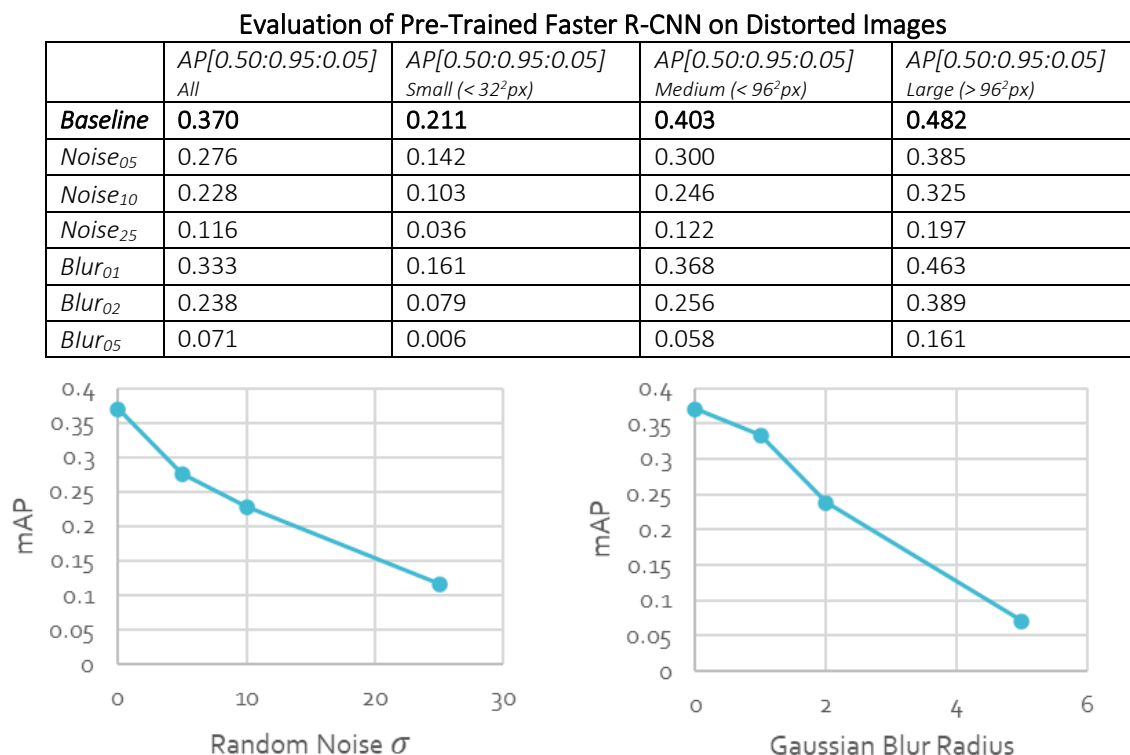


Figure 10 - Baseline Distortion Results

The first thing to notice is the rapid decrease in performance upon the slightest injection of distortion. With minimal noise injection, the overall performance decrease by about 25% (0.37 to 0.276), and this is amplified for smaller objects in images (decrease of approximately 33%). A similar pattern is observed for the blur transformations; however, the decrease is not as drastic both overall and for each object size, with an overall decrease of just 9% of overall performance (decrease of 24% in performance for small objects and nearly no decrease for medium and large objects). While minor noise seems to have a larger effect than minor blur, the highest level of noise and blur injections show the opposite property – the maximum level of blur distortion used showed an overall performance decrease of 81%, whereas the maximum level of noise injection showed a precision decrease of just 69% across the 5,000 validation images. Especially in the case of blur, smaller objects are almost undetectable, with a mAP of just 0.006 with the blur<sub>05</sub> transformation.

Additionally, it can be observed that there is a more immediate decrease in performance with minimal noise injection when compared to using blur to distort the image (noise<sub>05</sub> mAP: 0.276 vs blur<sub>01</sub> mAP: 0.333). This appears to indicate that the specific pixel color values are more important when identifying an object, however this idea is flipped with higher levels of distortion. At the maximum levels of image distortion, blur<sub>05</sub> and noise<sub>25</sub>, the Faster R-CNN model is better able to detect noisy objects than blurred objects across all object sizes, however large objects are identified at a very similar rate.

In an attempt to mitigate this decrease in performance, model fine-tuning was performed using distorted images as input. The fine-tuning was performed in three stages: fine-tuning the feature extractor (ResNet-50 backbone), fine-tuning the Region Proposal Network (RPN) and ROI Pooling/final classification layers, and fine-tuning the entire model. The results of each are shown below:

#### Evaluation of Refined Faster R-CNN on Distorted Images

2000 images, 10 epochs, Fine-Tune Backbone

	<i>AP[0.50:0.95:0.05] All</i>	<i>AP[0.50:0.95:0.05] Small (&lt; 32<sup>2</sup>px)</i>	<i>AP[0.50:0.95:0.05] Medium (&lt; 96<sup>2</sup>px)</i>	<i>AP[0.50:0.95:0.05] Large (&gt; 96<sup>2</sup>px)</i>
<b>Baseline</b>	<b>0.337</b>	<b>0.188</b>	<b>0.375</b>	<b>0.438</b>
Noise <sub>05</sub>	0.279	0.141	0.309	0.379
Noise <sub>10</sub>	0.240	0.111	0.265	0.341
Noise <sub>25</sub>	0.134	0.045	0.144	0.217
Blur <sub>01</sub>	0.284	0.139	0.316	0.397
Blur <sub>02</sub>	0.203	0.070	0.223	0.315
Blur <sub>05</sub>	0.082	0.011	0.074	0.168

#### Evaluation of Refined Faster R-CNN on Distorted Images

2000 images, 10 epochs, Fine-Tune ROI Pooling and RPN Layers

	<i>AP[0.50:0.95:0.05] All</i>	<i>AP[0.50:0.95:0.05] Small (&lt; 32<sup>2</sup>px)</i>	<i>AP[0.50:0.95:0.05] Medium (&lt; 96<sup>2</sup>px)</i>	<i>AP[0.50:0.95:0.05] Large (&gt; 96<sup>2</sup>px)</i>
<b>Baseline</b>	<b>0.347</b>	<b>0.193</b>	<b>0.388</b>	<b>0.439</b>
Noise <sub>05</sub>	0.256	0.131	0.284	0.342
Noise <sub>10</sub>	0.211	0.098	0.231	0.299
Noise <sub>25</sub>	0.109	0.035	0.118	0.175
Blur <sub>01</sub>	0.316	0.153	0.357	0.423
Blur <sub>02</sub>	0.228	0.079	0.248	0.361
Blur <sub>05</sub>	0.071	0.009	0.059	0.153

Evaluation of Refined Faster R-CNN on Distorted Images  
2000 images, 10 epochs, Fine-Tune Entire Model

	$AP[0.50:0.95:0.05]$ All	$AP[0.50:0.95:0.05]$ Small ( $< 32^2px$ )	$AP[0.50:0.95:0.05]$ Medium ( $< 96^2px$ )	$AP[0.50:0.95:0.05]$ Large ( $> 96^2px$ )
<b>Baseline</b>	<b>0.352</b>	<b>0.196</b>	<b>0.391</b>	<b>0.448</b>
$Noise_{05}$	0.260	0.134	0.285	0.350
$Noise_{10}$	0.214	0.096	0.234	0.303
$Noise_{25}$	0.109	0.035	0.117	0.178
$Blur_{01}$	0.319	0.157	0.358	0.429
$Blur_{02}$	0.230	0.079	0.248	0.365
$Blur_{05}$	0.071	0.008	0.059	0.156

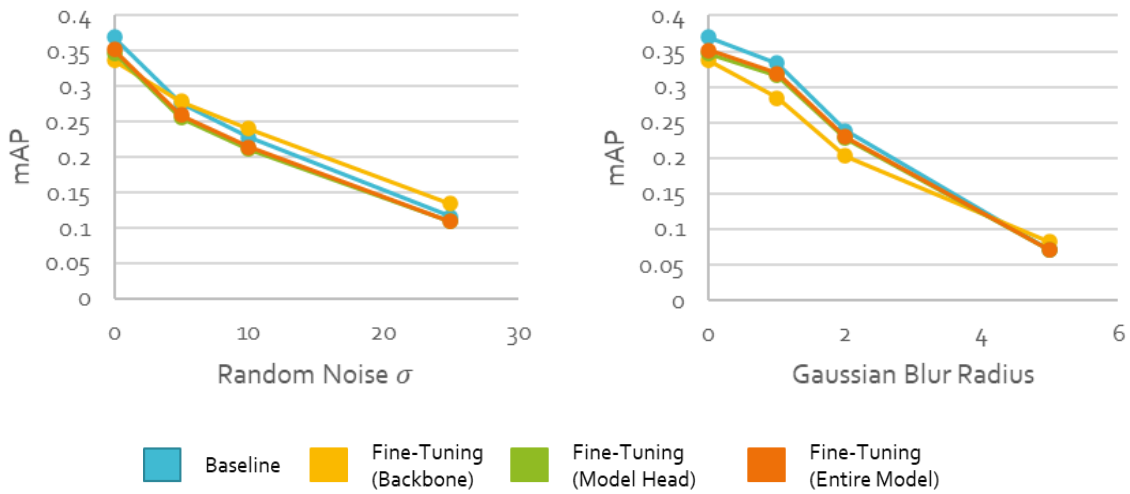


Figure 11 - Fine-Tuned Distortion Results

Across the different fine-tuning methods, each done with constant training parameters, differences begin to emerge as various components of the model are more sensitive to refinement than others, as well as the two different distortion techniques. Overall, refinement of the model backbone (the ResNet-50 feature extractor in the PyTorch implementation), when fine-tuned, is best able to mitigate the decrease in performance associated with random noise injection at levels of distortion above zero. This refinement outperforms the baseline model (without fine-tuning on distorted input) across all levels of noise injection, except for the baseline images without any distortion. Fine-tuning the backbone is the only refinement technique that outperforms the baseline model when noise is injected, as the full-model refinement and model head refinement both perform worse than the unrefined model, albeit close performance at lower noise levels. It is interesting to note that the full-model refinement performs worse than the baseline model, as well as the model where just the feature map backbone was refined. The conclusion that can be drawn from this is that refining the model head (the RPN and then pooling/final predictive layers) is the component of the model that responds most poorly to noise injection and causes a larger decrease in performance when fine-tuned on distorted images, however would require additional investigation to conclusively prove.

These conclusions appear nearly opposite the results shown when locating objects in images transformed with Gaussian blur. With distortion, the model backbone fine-tuning appears to accelerate the decrease



in performance as lower-levels of blur when compared to the baseline model (e.g. mAP of 0.284 for fine-tuned backbone vs 0.333 with  $\text{blur}_{01}$ ). The performance of the model-head-refined model and fully-refined model both appear closer (albeit slightly worse) than the baseline model, which is an opposite result to the noise-injected images. However, at the highest levels of distortion recorded ( $\text{blur}_{05}$ ), the model with the refined backbone outperforms all other refinement techniques as well as the baseline model, proving to mitigate more of the performance decrease than other techniques. This result means there appears to be a trade-off with blurred object detection: improved performance at lower levels of distortion with a minimally refined model (or simply using the baseline model), and improved performance with the backbone-refined model at higher levels of distortion. This appears to be a counter-result to the noise-injection results, where the backbone-refined model outperforms all other models at all levels of distortion (with comparable or slightly decreased performance with no distortion). For this reason, the impact of additional backbone fine-tuning is also explored.

### Evaluation of Pre-Trained Faster R-CNN on Distorted Images

20,000 images, 10 epochs, Fine-Tune Model Backbone

	$AP[0.50:0.95:0.05]$ <i>All</i>	$AP[0.50:0.95:0.05]$ <i>Small (&lt; 32<sup>2</sup>px)</i>	$AP[0.50:0.95:0.05]$ <i>Medium (&lt; 96<sup>2</sup>px)</i>	$AP[0.50:0.95:0.05]$ <i>Large (&gt; 96<sup>2</sup>px)</i>
<b>Baseline</b>	<b>0.298</b>	<b>0.157</b>	<b>0.335</b>	<b>0.392</b>
<i>Noise<sub>05</sub></i>	0.264	0.127	0.292	0.363
<i>Noise<sub>10</sub></i>	0.237	0.104	0.261	0.338
<i>Noise<sub>25</sub></i>	0.146	0.047	0.152	0.237
<i>Blur<sub>01</sub></i>	0.235	0.104	0.266	0.333
<i>Blur<sub>02</sub></i>	0.190	0.068	0.212	0.289
<i>Blur<sub>05</sub></i>	0.104	0.020	0.102	0.189

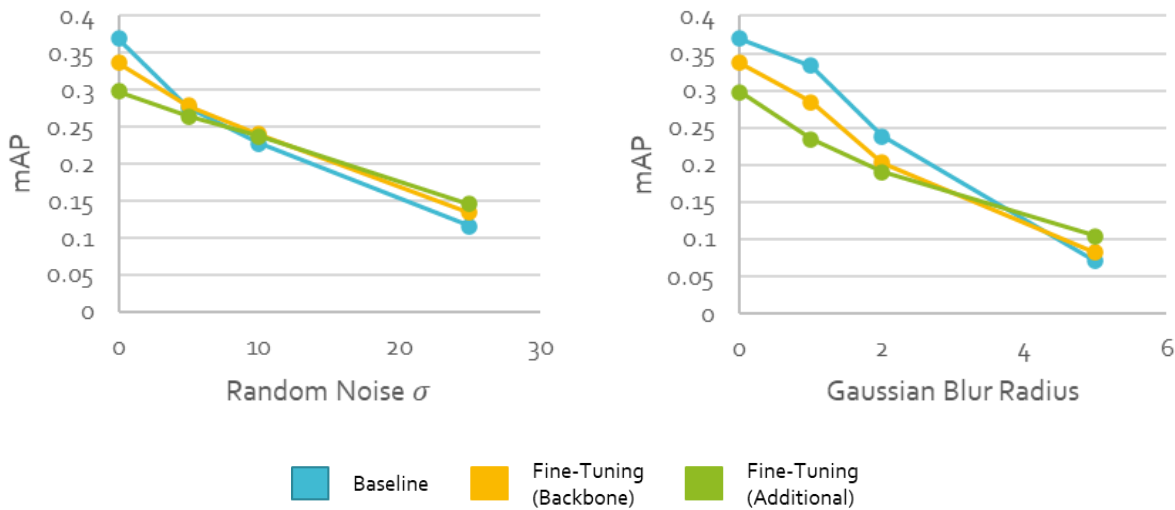


Figure 12 - Fine-Tuned Model Backbone Results

Additional fine-tuning of the model backbone only (when importing the entire pretrained PyTorch implementation) shows interesting results. For the noise-injected data, the increased training (10x the training data) appears to cause the model to greatly underperform at lower levels of distortion and outperform other techniques at higher levels of distortion. This runs counter to the previous results, which found that refining the backbone caused comparable results at lower levels of noise injection and

improved performance at higher distortion levels. This could be due to the model over-fitting on the distorted training images, thus causing it to underperform when presented with images with low to no distortion.

This pattern is mirrored in the model performance when detecting objects in images that have been distorted with Gaussian blur. Similar underperformance is noted at lower levels of distortion, however the degree to which the model underperforms is only increased with the additional refinement. Without distortion, the mAP value drops from 0.37 (baseline) and 0.337 (slight refinement) to 0.298 (additional refinement), a decrease of slightly over 20%. This decrease in performance continues at low levels of distortion ( $\text{blur}_{01}$  – 0.333 vs 0.284 vs 0.235, respectively), however becomes comparable to the slightly-refined model at medium levels of distortion. At the highest levels of image blur recorded ( $\text{blur}_{05}$ ), the model further improves its performance over the moderately-refined model, improving the mAP from 0.082 to 0.104 – an increase of just over 20%. The additional refinement further enforces the idea that there is a tradeoff in model performance – the more fine-tuned models tend to outperform the pretrained/baseline model at these higher levels of distortion, whereas they underperform or only perform comparably to that baseline at lower levels of distortion.

## ***Conclusions***

In conclusion, evaluating the performance of the Faster R-CNN model on distorted images shows that it is not very robust to the two most common types of noise injection, Gaussian blur representing motion blur and noise injection representing temperature/lighting/processing corruption. The ability for the model to detect objects within images immediately decreases with the slightest addition of distortion, and only continues to decrease with higher and higher levels. When attempting to mitigate this decrease in performance, fine-tuning the feature extraction backbone with distorted input proved to perform equally well as the unrefined model at lower levels of noise injection, and outperform the baseline model at higher levels. Conversely, focus on the feature extractor showed the worst drop in performance when detecting objects distorted with blur at lower levels, yet outperformed all other types of refinement and baseline models at higher levels of blur. Additionally, it was shown that additional fine-tuning of the feature extraction backbone would amplify this pattern: further decreased performance compared to the baseline model at lower levels of distortion but improved performance at higher levels. This work shows that it is possible to improve the performance of the Faster R-CNN model at higher levels of distortion through the additional fine-tuning of the feature extractor backbone, but the best-performing model at low-to-no distortion levels is still the pretrained implementation.

The biggest challenge of this project was identifying the proper hyperparameters during training that could be used to improve the performance of the Faster R-CNN model on the distorted images. Due to the large volume of training data, computation limitations, and other factors, each set of hyperparameters would require multiple hours to properly train, thus limiting the number of combinations that could be investigated. To alleviate this challenge, once a well-performing set of hyperparameters were found, these values were then kept constant across the three different refinement processes (isolating refinement to the feature extractor, isolating refinement to the model head, refining the complete model). This allowed for a much quicker gathering of results. However, once these baseline performance numbers were found attempts any additional attempts to further improve

performance would be slow-developing, and often to no avail as overall performance would frequently decrease.

There are a few aspects of this project that have the potential to further improve the final results. Additional training time or improved computing power would have assisted in the more comprehensive evaluation of hyperparameter combinations, as one could spend either more time training different models, or train models in a more efficient manner. While the EC2 instance used to perform the training and analysis was equipped with a GPU, there was limited memory meaning that, even with a data loader for memory management, the maximum batch size that could be used during training was 4 images. This limited the speed at which models could be refined. Additionally, as a single model may not be best equipped to detect objects in both extremely distorted and minimally distorted images, an ensemble technique could be used to combine the predictions of different models all trained to detect objects at different levels of distortion. This could prove especially promising if paired with techniques mentioned in [10] to identify distorted objects or sections of an image within the larger picture, allowing for more targeted model application.

Additional work that could be undertaken in the future would revolve around different model configurations and generalizing these results to other types of object detection models. First, the PyTorch implementation featured a ResNet-50 as the feature extraction backbone, however it is possible to substitute this model for any other type of CNN model that can generate feature maps. A logical next step to this work would be to swap the ResNet50 for another backbone, such as a model from the VGG family or DenseNet family and compare the results to identify the backbone which is best able to identify distorted objects. Additionally, determining if these results still hold while using a different type of object detection model, such as one from the YOLO family of single-shot detectors, would provide robustness to these results. Being able to understand if all object detection models are impacted by distortion in a similar manner would better allow for the selection of the best performing model when attempting to perform distorted object detection tasks in production systems. As object detection models become a more integral part of daily life, these next steps likely define a promising area of additional research into improved performance and closer to near-human capabilities in the computer vision field.

## Works Cited

---

- [1] Russakovsky, Olga, et al. "ImageNet Large Scale Visual Recognition Challenge." *International Journal of Computer Vision*, vol. 115, no. 3, 2015, pp. 211–252., doi:10.1007/s11263-015-0816-y.
- [2] Krizhevsky, Alex, et al. "ImageNet Classification with Deep Convolutional Neural Networks." *Communications of the ACM*, vol. 60, no. 6, 2017, pp. 84–90., doi:10.1145/3065386.
- [3] Liu, Wei, et al. "SSD: Single Shot MultiBox Detector." *Computer Vision – ECCV 2016 Lecture Notes in Computer Science*, 2016, pp. 21–37., doi:10.1007/978-3-319-46448-0\_2.
- [4] Luo, Yixin and Fan Yang. "Deep Learning With Noise." (2014).
- [5] Nguyen, Anh, et al. "Deep Neural Networks Are Easily Fooled: High Confidence Predictions for Unrecognizable Images." *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, doi:10.1109/cvpr.2015.7298640.
- [6] Dodge, Samuel, and Lina Karam. "Understanding How Image Quality Affects Deep Neural Networks." *2016 Eighth International Conference on Quality of Multimedia Experience (QoMEX)*, 2016, doi:10.1109/qomex.2016.7498955.
- [7] Zhou, Yiren, et al. "On Classification of Distorted Images with Deep Convolutional Neural Networks." *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017, doi:10.1109/icassp.2017.7952349.
- [8] Wang, Kuan-Hsun, and Shang-Hong Lai. "Object Detection in Curved Space for 360-Degree Camera." *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2019, doi:10.1109/icassp.2019.8683093.
- [9] Yang, Ruiduo; Shih, Yichang; Liang, Chia-Kai; and Hasinoff, Sam, "Improved Object Detection in an Image by Correcting Regions with Distortion", *Technical Disclosure Commons*, (April 01, 2020) [https://www.tdcommons.org/dpubs\\_series/3090](https://www.tdcommons.org/dpubs_series/3090)
- [10] Daisuke Uchida, Atsushi Yamashita, and Hajime Asama "Detecting image frames which contain a moving object from a severely distorted video stream using dynamic mode decomposition", *Proc. SPIE 11515, International Workshop on Advanced Imaging Technology (IWAIT) 2020, 1151510 (1 June 2020)*; <https://doi.org/10.1117/12.2566797>
- [11] Hamzeh, Y., El-Shair, Z., and Rawashdeh, S., "Effect of Adherent Rain on Vision-Based Object Detection Algorithms," *SAE Technical Paper 2020-01-0104*, 2020, <https://doi.org/10.4271/2020-01-0104>.
- [12] Chen, Yuhua, et al. "Domain Adaptive Faster R-CNN for Object Detection in the Wild." *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, doi:10.1109/cvpr.2018.00352.
- [13] He, Kaiming, et al. "Mask R-CNN." *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, doi:10.1109/iccv.2017.322.
- [14] Lin, Tsung-Yi, et al. "Microsoft COCO: Common Objects in Context." *Computer Vision – ECCV 2014 Lecture Notes in Computer Science*, 2014, pp. 740–755., doi:10.1007/978-3-319-10602-1\_48.

- 
- [15] Ren, Shaoqing, et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 6, 2017, pp. 1137–1149., doi:10.1109/tpami.2016.2577031.
- [16] Girshick, Ross, et al. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation." *2014 IEEE Conference on Computer Vision and Pattern Recognition*, 2014, doi:10.1109/cvpr.2014.81.
- [17] Girshick, Ross. "Fast R-CNN." *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, doi:10.1109/iccv.2015.169.
- [18] "Torchvision.models." *Torchvision.models - PyTorch 1.6.0 Documentation*, pytorch.org/docs/stable/torchvision/models.html.
- [19] [https://cv-tricks.com/wp-content/uploads/2019/07/ResNet50\\_architecture-1.png](https://cv-tricks.com/wp-content/uploads/2019/07/ResNet50_architecture-1.png)
- [20] He, Kaiming, et al. "Deep Residual Learning for Image Recognition." *ArXiv.org*, 10 Dec. 2015, arxiv.org/abs/1512.03385.
- [21] <https://github.com/Cartucho/mAP>

## ***Appendix – Code***

Due to the volume of code, it would make the PDF too large, so it can all be found at:

<https://github.com/alexjcohen/Capstone/tree/master/Code>