Exploring fine-tuning techniques for weed detection: A technical report on implementation and performance analysis of various Super Gradients object detection models

Project by: Rohan Anil Marar
Date: 11th February 2024

Object Detection on weed datasets: training YOLO NAS and other similar models.
My choice for a custom use-case of this project was weed detection.
I had developed this project as a part of an in-house Jetson Nano project in my college, where the final inference and testing was to be done on the Jetson Nano Tegra model. Due to technical difficulties, the inference part on Jetson Nano could not be carried out. However, I have made sure to document all my workings and ideas in the form of this document.

This article serves as a record of all experiments and final working results for my project on exploring state-of-the-art object detection models. I made use of various versions of YOLO NAS and other facilities provided by SuperGradients.Datasets were taken from the Roboflow website. Training of the final models was done, using weed dataset images in YOLO format, on standard T4 GPU of google Colaboratory notebooks.

I considered using YOLO NAS for a few important reasons. Traditional YOLO models are ready-to-use with simple syntax, where no transfer learning or fine tuning is immediately required if we want to train the model on a custom dataset. And if the programmer does want to perform transfer learning for additional benefits, then Ultralytics provides a very simple command syntax that takes care of all training steps.I felt this was a limitation and wanted to be in charge of all training steps myself and SuperGradients provides a perfect way to do that for YOLO NAS. I could take care of all hyperparameters, customization, configuration files.

Second reason was that YOLO NAS essentially is much better suited for edge devices,and provides much faster inference time, along with greater accuracy.

My main purpose for this project was to thoroughly explore the state-of-the-art(SOTA) models used in the industry for object detection, and all pre-trained checkpoints provided by Super gradients serve as a solid foundation from which the custom training can be done. Another goal was to test the limits of maximum inference speeds that can be achieved on Jetson Nano for practical real-time applications. Almost always, the pre-trained checkpoints provided by such libraries are not fine-tuned for maximum performance efficiency, with their focus being on procedural aspects rather than achieving the highest possible inference speed or accuracy.Many such models are well-suited for such edge device applications, and there are always tradeoffs to be made between accuracy, inference speeds and model sizes. In total, I ran experiments on four models thoroughly, which were: yolonas small, yolonas medium , yolo_x_small, and ppyoloe_medium. In particular, we found that yoloNAS-small and yoloNAS-medium are both almost equally good for our project. Yolonas-medium provided slightly better accuracy and performance, and even the tradeoff for its greater latency, as compared to yoloNAS-small, is only 2 millisecond which is negligible. In contrast, yolo_x_small and ppyoloe_medium both showed worse results, where the latter performed slightly better than the former.

My novel approach:

While I was building the project, I thought of a technique which could potentially be used to make things easier and better. Traditionally, transfer learning(including frozen layers) is done in computer vision for classification models, where the pre-trained model’s weights might get adversely changed and result in false outputs. However, technically frozen layers is not needed a lot for pre-trained object detection models about to be trained on a new custom dataset, as the model’s weights do not get changed for the worse, and most of the computation is involved in feature

extraction, kernels and the complex mechanism inside a convolutional neural network.

I came up with a new novel approach to training the models repetitively in order to keep on improving performance. I like to call it : “ Iterative fine-tuning after manually freezing layers”. The training steps are as follows: using a custom code snippet I proceed to freeze the backbone parameters which serve as feature extractors,since they are already sufficiently trained and require no further fine-tuning. Normally yolo models can be trained directly on a custom dataset, and the model weights get fine-tuned while still not fluctuating a lot. Training like this, using one small or medium sized dataset, is fine. However if the developer wants to make the model generalize more by training it on similar but newer and bigger datasets, then he can choose to freeze the backbone parameters, and train the model’s neck and head on the new dataset,essentially performing the process normally done during transfer learning,but in this case, being done on the same type of datasets repetitively. In this manner, a lot of training time and computation cost is saved, and the model can reach a better convergence faster. This same procedure can be kept on repeating again and again, each time by loading the model from its last best checkpoint, freezing the weights, and using a new dataset every time. As we found out, doing this incrementally improves model performance across all important metrics, while reducing time that would have been spent if the entire model was trained. This procedure also avoids the risk of overfitting due to repetitive fine-tuning, and enables the model to adapt over different data distributions in a better way.

The developer can then further choose to freeze other parameters of the model as well, such as the model’s neck, and try out variations to see how it affects the performance. I ran one experiment where I froze both the backbone and neck parameters, which resulted in much faster training(training time reduces by a huge factor), and a very slight improvement in metrics), as when trained by only freezing the backbone. Hence one can save a lot of computational cost and training time by trying out different

ways to manually freeze the layers and train the model on new datasets, as the end result depends mostly on the end head layers of such models. This can preserve information more efficiently as well, and prevent risks of overfitting or worsening of model performance after repetitive fine-tuning. This use case of repetitive fine-tuning using different datasets becomes more important when the real world applications that such models are intended for, are very diverse and require different perspectives. In the case of weed detection itself, different datasets contain different types of images with various backgrounds and different weed structures and crops, and hence training a model on different datasets repetitively, ensures that the model becomes robust and generalized, which is the whole intent of my experiment.

Across all the four models that were tested, the best overall performance was shown by the yolonas_medium model. In one experiment I loaded the model's pretrained weights, and trained it normally on a new dataset. Once sufficiently trained, I then re-loaded it, manually froze the backbone and the neck parameters, and then re-trained this model on a different and slightly bigger dataset, which improved its performance a lot across many metrics, while reducing training time by a huge factor. Most noticeably, the mAP@0.50 score went from 0.71 to 0.85 ! Then,I repeated this same procedure for a slightly bigger and different weed dataset, and once the training began, initial mAP scores dropped, but then increased consistently until finally stopping at 0.84 in the 10th epoch, along with similar values for different metrics, which shows that no matter the changes in datasets, the model quickly adapts to the data distribution via this fine-tuning technique of mine.

As an example of what I refer to in terms of repetitive fine-tuning and model generalization across different datasets, the model was initially trained on datasets with training images of this type:

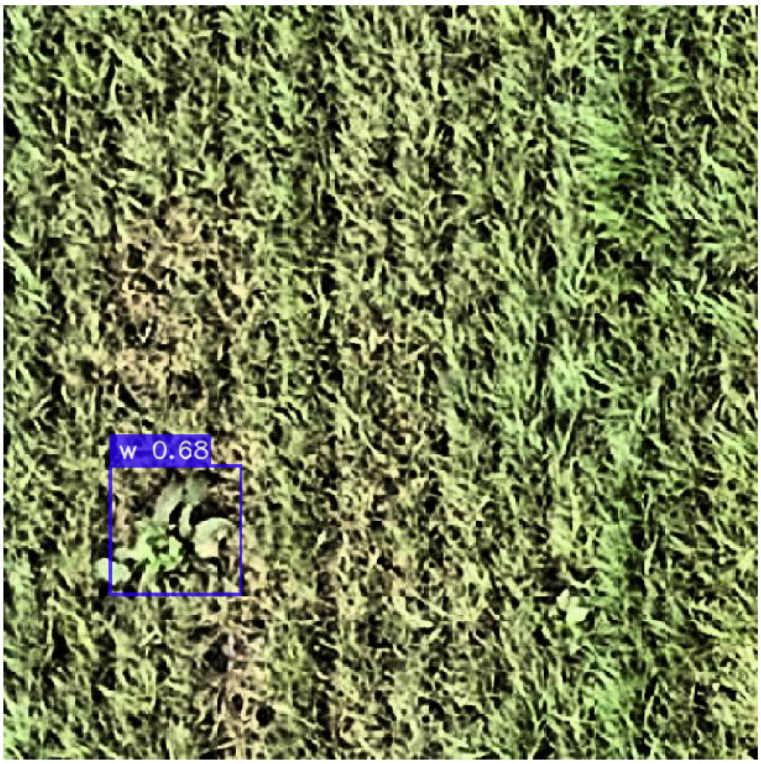
The model then works perfectly fine,when tested on images of similar type, and this is also perfectly valid in real-world scenarios where some imaginary robot is tasked with weed detection and removal on grounds with barren lands and prominent weeds. However, in the case of a real-world scenario where the robot is tasked with working in agricultural fields or local gardens with plenty of plantations everywhere, the difficulty level quickly increases.

In particular, I then proceeded with training and fine-tuning with frozen layers ,the once-trained model, on another dataset with different training images, such as the following:

Upon following this procedure, the model's mAP scores drop a bit while training, averaging out slightly above 0.55. But once tested, the model shows very impressive performance across different types of testing images, especially when the confidence scores of bounding boxes are set to 0.5 or slightly more.

In particular, for one type of testing images on which I had not trained it,the model still showed remarkable predictions which proved that it had most definitely generalized efficiently. These images were very difficult to draw predictions from ,such as the following:

The model shows valid performance, for very minute protrusions and hard-to-see extra weed growth, especially for confidence score settings greater than 0.5, such as the following:



The above prediction was indeed accurate, according to the default label coordinates for this test image, when in fact the model was not accustomed to the dataset at all. This makes it well-suited for practical real-world applications.

Furthermore, this same procedure can be used again and again, any time the developer wants to train the model on different custom datasets and use-cases.

Future suggestions for people actively contributing to this area is that, similar to how standard yolo models that we can use using the Ultralytics documentation steps, have features like hyperparameter evolution, a similar method could be further miniaturized and developed for YOLO NAS as well, where the evolution steps complete within epochs set manually by user, rather than being set to 500 or 1000 or more epochs. If the inherent process of genetic mutation and evolution algorithms themselves are made faster and shorter, and are then integrated into libraries that can be used on YOLO NAS models,such that the expensive nature of computing the base

scenarios was reduced and made faster and easier, then it would be an amazing thing. I believe hyperparameter evolution is an amazing feature, and its usage for fast YOLO NAS training would be a lot beneficial to the community.