Performance Metrics Compared: YV8 vs ViT

Anthony J. Vasquez
Research Data Scientist
Inficon Inc.
2 Technology Pl,
East Syracuse, NY, 13057
Email: avasque1@jh.edu

Editor: Anthony J. Vasquez

Abstract

This document investigates performance metrics of object detectors that are no longer cutting edge but are still relevant as they are still widely used solutions. The metrics considered for this study include, parameter count, training GPU memory usage, testing GPU memory usage, time to train, time to inference, mean-average precision (mAP) at 0.5 intersection-over-union (IOU), mAP at 0.5 to 0.95 IOU, and average recall at 0.5 to 0.95 IOU, of the validation set at the end of the tenth training epoch of each network. The two architectures considered are the Single-Shot YoloV8 object detector with CSPNet backbone (Wang Chien-Yao, 2019), and the region proposal network FasterRCNN with a Nested Hierarchical Transformer (NesT) vision transformer (ViT) backbone.

1 Introduction

Object detection is a widely used machine learning area of interest that has grown exponentially over the last ten years. Due to the increase in compute power, as well as the increase in optimization techniques used, many open-source off-the-shelf solutions have been developed such as the Hugging Face framework for object detection (ViTDet, 2023), and YOLOV8 (Glenn, 2023). The latter has been around since Joseph Redmond first published You Only Look Once: Unified, Real-Time Object Detection (Redmon Joseph, 2016). Since then, YOLO has gone from being an object detection algorithm, to an object detection framework expanded upon by Alexey Bochkovskiy (AlexeyAB) and turned into a framework by Ultralytics.

The transformer architecture was first conceived in the now infamous "Attention is All You Need" (Vaswani, 2017). In this hallmark paper, the authors showed that attention mechanism for a new transformer-based model could be used in lieu of the then innovative process intensive CNN encoder/decoder architectures. Though the architecture was initially used for natural language processing, it found use in other fields, including computer vision, for classification and object detection. One example is the Hugging Face framework that has an extensive and robust library of vision transformer networks that clarified how to implement a pretrained ViT backbone. For that reason, the NesT ViT pretrained weights were used for this study.

The question that should be answered by this study is, does the vision transformer inference faster than a similar sized CNN? YoloV8 is well known for inferencing exceptionally fast when compared to other CNN based object detectors, but can it come close to the inference speed of ViT? Furthermore, can ViT perform as good or better on common performance metrics, such as recall, precision, and mAP scores. The final question is, what are the implications on hardware requirements for integration?

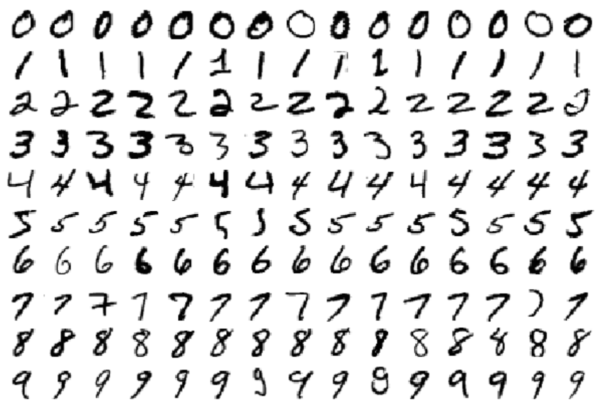
2 Method

Experiment methods are discussed in this section.

2.1 Data Acquisition

The Numbers MNIST (Modified National Institute of Standards and Technology) dataset (Deng, 2012) was downloaded from the popular data science website Kaggle (KhodaBakhsh, 2018). This dataset was first developed by LeCun et al., in 1994 and is commonly used as a machine learning benchmark dataset (Chavi, 2019). The train set includes 60k samples each with 784 features at 8-bit depth in CSV format. The test set includes 10k samples in the same format as the train set. The dataset was selected because of its extensive use over the past 25 years, and because the quality of the data is well established. Furthermore, the size of the dataset was sufficient yet small enough to overlay on larger noisy images, a task that is covered in the data preprocessing section.

Figure 1. Shown here are samples from the Numbers MNIST dataset (Yashwanth, 2020).



2.2 Data Preprocessing

The train digits were opened from a CSV file, converted to a Numpy array (Millman, 2020) and each reshaped to 28x28 pixels. The samples were randomly selected from the train set and had no limits to the number of times the samples could be used. A random number between 0-10 was selected, and that number of random images were sampled from the train set. A larger 256x256 pixel image was created and populated with random gaussian noise. The smaller randomly selected images were put at random locations throughout the images; however, no overlap was allowed at the image boundaries to avoid padding complications. However, overlapping was allowed between the number images, this was allowed to preserve the simplicity of the code. However, there is one logic step that ensures that the center of the images do not overlap, though it was observed that most of the large images were absent of overlapping samples.

The locations of each of the samples were recorded along with the class name, which were all written to a text document in the form of the class, upper left x-coordinate pixel, upper left y-coordinate pixel, width, and height of image. For example, each large image with overlaid samples would contain a list of class and coordinates like “0 200 50, 28, 28”. The example shows that a “0” is located at location (x=200, y=50) and is of size (w=28, h=28). Each of the text files could have from 1 to 10 such annotations. YOLOV8 requires that the coordinates be normalized according to the size of the image, which translates to dividing the coordinate values by the size of the large noisy image of 256. Once the samples were overlaid on the larger image, SciPy’s implementation of the single-dimension-Gaussian filter (Virtanen, 2020) was applied to the pixel values to smooth any obvious boundaries around the samples. One text file for each dataset (test and train sets), was written to disk and had the path to each image that could be read by the Custom Pytorch data-loader (Paszke, 2019).

Figure 2. Shown here is a large noisy 8-bit 256x256 image overlaid with multiple 28x28 MNIST samples.

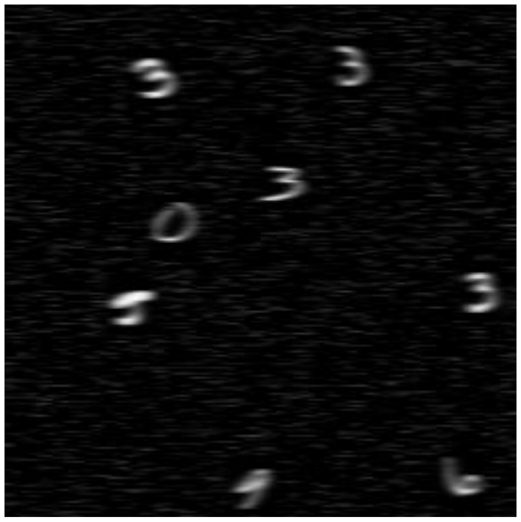
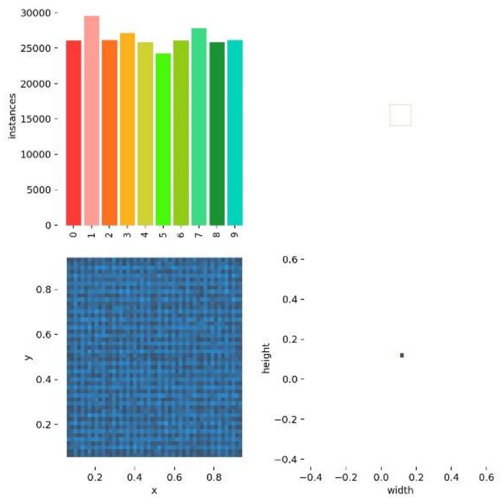


Figure 3. The following set of figures shows the distribution of classes to be mostly uniform. The bottom left figure shows the x and y pixel location distributions to be uniform which is expected because the coordinates were randomly sampled and overlaid on the large noisy image. Finally, the bottom right figure shows that the samples were of size 28x28 (W-by-H).



2.3 Dataset Split

The dataset was pre-split, which is to say that only train samples were used from the original Numbers MNIST train set split. Likewise, the test set only included test images from the original test set split.

2.4 Training YOLOV8 (YV8)

The YV8 network used the yolov8n.pt pretrained weights downloaded directly from Ultralytics (Jocher, 2023). The pretrained weights were trained on the well-known COCO dataset (Microsoft, 2017) composed of 330k images, 1.5 million object instances, and 80 unique object classes. As mentioned earlier, there are different architecture sizes that can be used with YV8 with each increase in size is a deeper network. For this study, YV8 nano weights were used for pretrained weights which includes a CSPNet with about three million trainable parameters (Wang Chien-Yao, 2019). The Adam optimizer was used with a learning rate of 0.0001. At the end of 10 epochs, the trained model was tested on the validation set.

2.5 Training NesT (ViT)

Initially, the object detector SSDLite320 Mobilenet V3 Large was used which is a similar architecture in size along with a ViT backbone. Unfortunately, the model did not generalize well and seemed to underfit the data. Instead, FasterRCNN Mobilenet V3 Large 320 FPN object detector was used with the NesT ViT backbone (Zhang, 2021). This network was not ideal, as it had a combined number of trainable parameters of nineteen million, about six times larger than YV8. Nevertheless, it seemed to generalize well to the dataset and offered competitive metric scores to YV8.

2.6 Evaluation

The metrics tested include, total trainable parameter count, training GPU memory used, testing GPU memory used, time to train, time to inference, mAP at 0.5 IOU, mAP at 0.5 to 0.95 IOU, and average recall at 0.5 ti 0.95 IOU. These metrics were all accessible through the Torchvision tutorial on object detection (Pytorch, 2023).

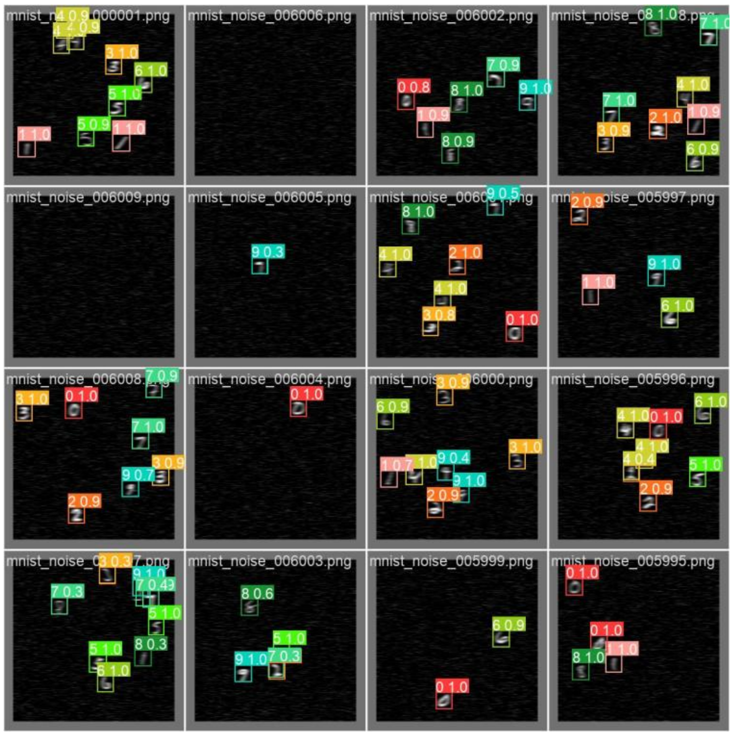
3 Results

This section discusses the individual results as well as a summary comparison of the two object detection networks. All training and testing was done on an R15 Alienware gaming laptop with Python-3.8.18 torch-2.1.1 CUDA:0 (NVIDIA GeForce RTX 3080 Ti Laptop GPU, 16384MiB). Additionally, all testing was done sequentially, which means that no two processes were done at the same time.

3.1 YOLOV8

YV8 trained using 4.1GB of GPU memory and took 3075 seconds or just under an hour to train. The inference time per image averaged 0.018 seconds over 9k test images. The mAP score at 0.5 IOU was 96.8 percent, and mAP at 0.5 to 0.95 IOU was 78.3 percent. The average recall at 0.5 to 0.95 IOU was 91.2 percent. Overall, these are satisfactory performance metrics considering the complexity of the task.

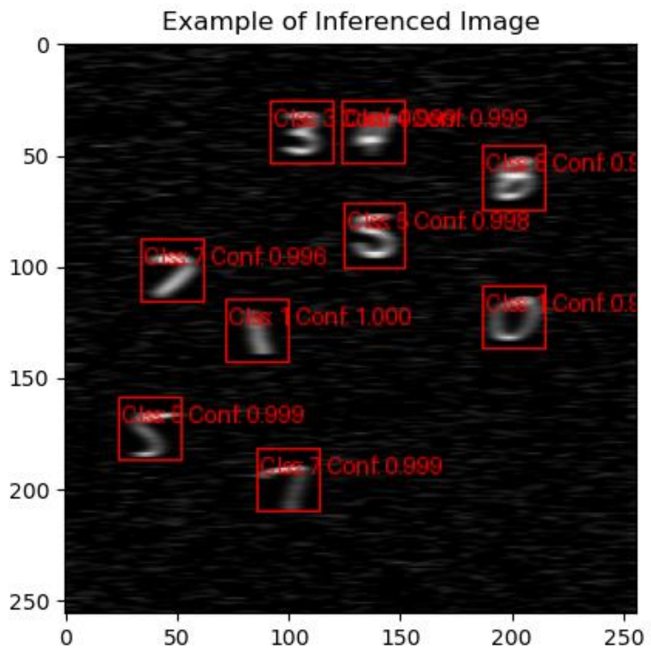
Figure 4. YV8 framework comes with a fancy mosaic functionality that is likely derived from Torchvision. YV8 seems perform well with only 10 epochs of training.



3.2 ViT

ViT trained using 15.8 GB of GPU memory and took 8590 seconds or about 2.4 hours to train. The inference time per image averaged 0.1 seconds over 9k test images. The mAP score at 0.5 IOU was 94.0 percent, and mAP at 0.5 to 0.95 IOU was 85.2 percent. The average recall at 0.5 to 0.95 IOU was 89.3 percent.

Figure 5. Unfortunately, a mosaic presentation was not created due to time constraints, but the below image shows that ViT also performs well on test data.



3.3 Results Summary

YV8 score better on most of the performance metrics. It had a lower trainable parameter count, used less GPU memory during training, took less than half the time to train, inferenced faster per image, scored a better mAP score at 0.5 IOU, and score a better average recall score at 0.5 to 0.95 IOU. They both inferenced with the same GPU memory. ViT scored better on mAP at 0.5 to 0.95 IOU by seven percent. This might have been the result of having a deeper network than YV8. Another noteworthy comment is that ViT scored only about one percent less than Yv8 on average recall, and only 2.8 percent less on mAP at 0.5 IOU.

Table 1. The performance table of YV8 vs ViT shows the comparison of the two archtectures.

Architecture	YV8	ViT
Parameter Count	3007598	19386354
Training GPU Memory (GB)	4.1	15.8
Testing GPU Memory (GB)	1.6	1.6
Train Time (s)	3075	8590
Inference Time Per Sample (s)	0.018	0.1
mAP @0.5 IOU	96.8	94
mAP @0.5 to 95 IOU	78.3	85.2
Average Recall @0.5 to 0.95 IOU	91.2	89.3

4 Conclusion

This section summarizes this work.

4.1 Conclusion

YV8 and ViT networks were trained on a reframed object detection version of Numbers MNIST. The two networks generalized well to the data, by YV8 performed better on most performance metrics. However, YV8 is a highly optimized network that is not entirely free to use, since if a company wants to make money using the YV8 framework by Ultralytics, royalties must be paid to the Ultralytics. On the other side, the ViT network is open source and can be used for all purposes by a user or their company. A hyperparameter search was not done on either network to achieve the results, which could be a subject for further investigation. Either way, at a high-level, both networks performed well and are both useable for both model development and real-time applications depending on their use-cases and customer requirements.

5 References

Chavi, Y. E. (2019). Cold Case: the Lost MNIST Digits. *Neural Information Processing Systems* . Vancouver: New York University.

Deng, L. (2012). The Mnist Database of Handwritten Digit Images for Machine Learning Research. *IEEE Signal Processing Magazine*, 141-142.

Glenn, J. E. (2023, 12 5). *github.com*. Retrieved from ultralytics: https://github.com/ultralytics/ultralytics

Jocher, G. E. (2023). *Detect*. Retrieved from Ultralytics: https://docs.ultralytics.com/tasks/detect/

KhodaBakhsh, H. (2018). *Kaggle.com*. Retrieved from MNIST Dataset: https://www.kaggle.com/datasets/hojjatk/mnist-dataset

Microsoft. (2017). *COCO: Common Objects in Context*. Retrieved from cocodataset.org: https://cocodataset.org/#home

Millman, C. R. (2020). Array programming with {NumPy}. *Nature*, 357-362.

Paszke, A. E. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32* (pp. 8024-8035). Curran Associates, Inc.

Pytorch. (2023). *Torchvision Object Detection Finetuning Tutorial*. Retrieved from pytorch.org: https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html

Redmon Joseph, E. a. (2016, May 09). You Only Look Once: Unified, Real-Time Object Detection. Washington.

Vaswani, A. (2017). Attention is All You Need.

Virtanen, P. E. (2020). Scipy 1.0: Fundamental Algorithms for Scientific. *Nature Methods*, 261-272.

ViTDet. (2023, 12). Retrieved from Hugging Face: https://huggingface.co/docs/transformers/model_doc/vitdet

Wang Chien-Yao, E. e. (2019). CSPNet: A New Backbone that Can Enhance Learning Capability of CNN. *Arxiv*.

Yashwanth. (2020). *MNIST Handwritten Digits Recognition using PyTorch*. Retrieved from Github.com: https://github.com/NvsYashwanth/machinelearningmaster

Zhang, Z. E. (2021). Nested Hierarchical Transformer: Towards Accurate, Data-Efficient and. *Arxiv*.