Implementing Pet Detection on the NVIDIA Jetson Nano

1st William Chestnut
Computational Science strand
Governor's School for Science and Technology
Hampton, United States
william.chestnut@nhrec.org

2nd Jason Cox
Computational Science strand
Governor's School for Science and Technology
Hampton, United States
jason.cox@nhrec.org

***Abstract*—Embedded computing devices have emerged as a cost-effective way of implementing machine learning on a mobile platform for use in everyday life. This research compares multiple implementations of convolutional neural networks on the Nvidia Jetson Nano for the purpose of animal recognition. We ultimately compare the use of a YOLOv4-tiny algorithm with optimized Tensorflow implementations to both demonstrate the capability of this technology for this application as well as determine which of the two prominent approaches would work better in this scenario. Both algorithms are evaluated on a set of incoming still images for metrics as well as tested with real-time footage to determine successful recognition ability. We found that animal detection and identification is very much achievable on the NVIDIA Jetson with a satisfactory frame rate, supporting our hypothesis that there are affordable computing solutions in this domain. Though both algorithms produced viable results, we found that certain constraints change which algorithm would be desired in this context.**

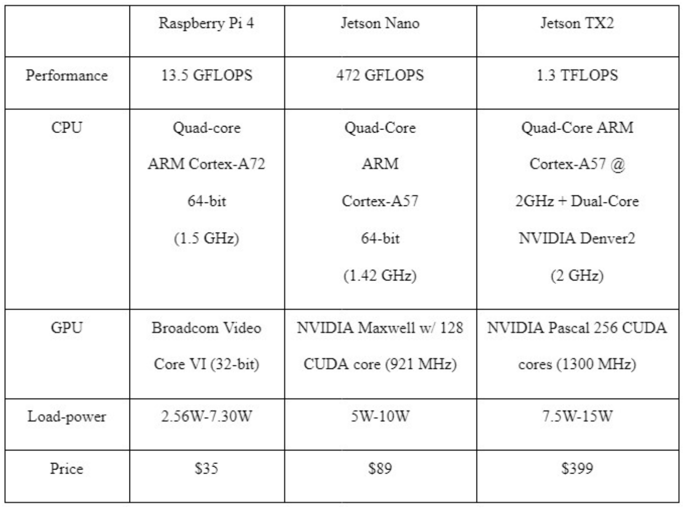
***Index Terms*—NVIDIA Jetson, machine learning, object detection**

I. Introduction

Machine learning is one of the largest and fastest growing areas of computer science as it develops into almost all of today's sectors such as agriculture, business, science, and many more. This trend will continue as researchers develop larger and more complex neural networks and models, but as this performance increases, so will the power consumption of these models. While power consumption may not be an issue on the typical personal computer, it is a large limitation for people wanting to develop mobile systems that employ machine learning algorithms. This includes applications such as self driving cars, autonomous robots, drone obstacle avoidance, camera recognition, and many more. While machine learning has the potential to drastically change the world, it's crucial that researchers pursue an efficient and effective way of developing machine learning [5].

This is where the NVIDIA Jetson lineup comes in. The NVIDIA Jetson lineup consists of embedded heterogeneous devices (contains a CPU and GPU) of different memory sizes designed for mobile low-power machine learning applications [4]. Since machine learning algorithms deal with matrix data structures, which the GPU, with its thousands of cores processors, is optimized to work well with, the Jetson lineup is perfect for machine learning applications. NVIDIA arguably provides the best options for these devices with a high price to performance and efficiency, such as the NVIDIA Jetson Nano. The NVIDIA Jetson Nano is priced at $90, but provides a performance of 472 GFLOPS, almost 40 times as much as the $35 Raspberry Pi. Furthermore, it maintains this performance at a load power draw of 5-10 watts [4].

Identify applicable funding agency here. If none, delete this.



	Raspberry Pi 4	Jetson Nano	Jetson TX2
Performance	13.5 GFLOPS	472 GFLOPS	1.3 TFLOPS
CPU	Quad-core ARM Cortex-A72 64-bit (1.5 GHz)	Quad-Core ARM Cortex-A57 64-bit (1.42 GHz)	Quad-Core ARM Cortex-A57 @ 2GHz + Dual-Core NVIDIA Denver2 (2 GHz)
GPU	Broadcom Video Core VI (32-bit)	NVIDIA Maxwell w/ 128 CUDA core (921 MHz)	NVIDIA Pascal 256 CUDA cores (1300 MHz)
Load-power	2.56W-7.30W	5W-10W	7.5W-15W
Price	$35	$89	$399

Fig. 1. Device Specs [4]

Machine learning ranges from many different types of applications and algorithms such as association rule mining or reinforcement learning, however, this paper focuses on computer vision and object detection algorithms, one of the more intriguing sections of machine learning. In computer vision, an image is taken as input and the algorithm returns outlines or boxes for objects it sees, along with its confidence in these detections. This opens the door for computers to complete tasks such as self-driving, pedestrian detection, and in the case of this project, animal detection. Computer vision works through neural networks and convolutional neural networks, both of which are trained to detect distinct patterns in image data. It continues training until it is able to develop a general sense of which patterns indicate different objects and can apply these to future images.

II. Research Problem

To demonstrate the ability of the NVIDIA Jetson Nano, this project will focus on the implementation of a pet detection algorithm, which will print out any dog breed detections it makes from testing images. For our project, we included a specific pet prediction class to compare against other general dog breeds. If successful, this software can be implemented into a future product that alerts the owner if their pet is outside. This can then be utilized to open the dog door remotely along with other operations. This application will not only bring convenience for the owner but it will also address the issue of dogs being accidentally left out in the cold or rain. For example, there were 37 dog deaths in 2019 from them being left out in cold weather [1]. The software system in this project can help address this problem and bring more convenience to pet owners.

III. Research Questions

In an effort to add to the statistics and project knowledge pool surrounding the Jetson, our project will investigate a new role of the Jetson as a pet alert control system. Our project will detect if an animal is sitting outside of a door and classify it as different breeds of dogs, along with the inputted unique pet. If it is the unique pet or the same species as it, the Jetson will output an alert which can later be turned into a phone notification if needed.

IV. Literature Review

A. Jetson and Algorithm Improvements

While the NVIDIA Jetson provides a great platform for mobile machine learning, there are still some improvements researchers have suggested, both at the circuit and microarchitecture level. For circuit level changes, many researchers have recommended using smaller feature sizes in processors to increase integration density and employing non-volatile memories in an effort to achieve near-zero idle power [3]. The primary recommended microarchitecture improvement is a closer integration between the CPU and GPU in an effort to reduce data-transfer overheads [3].

While these improvements may reap noticeable performance gains, they are not feasible for users to complete, but there are many options in regards to optimizing algorithms to run on the NVIDIA Jetson. First, it's crucial to choose the right algorithm for the project. Computer vision models come in a range of different input resolutions and layer depths. An increase in resolution and layer depth will enable the algorithm to pick up on and learn more details, but will slow the algorithm down because of the increase in data it must process [3]. This is why it's important to choose the optimal size for input resolution and layer depth that best balances accuracy and speed for the algorithm.

B. Past Projects

Since the applications for the Jetson are practically endless, there is a plethora of research papers and projects covering the applications and performance of the Nvidia Jetson, along with optimizations they recommend taking to improve the algorithm's performance. For example, a research team in Hong Kong utilized the NVIDIA Jetson Nano to translate Hong-Kong sign language to allow deaf people to better communicate with others [5]. As expected, a high power computer was not an option for such a portable application, so the researchers decided to utilize the NVIDIA Jetson Nano. With the application of a 3D cNN, the researchers were able to achieve an accuracy of 93.3% and a total response time of 5.82 seconds [5]. This project along with others such as an autonomous robot using LiDAR, completed at the University of Padova, show how the Jetson can revolutionize the world of computing with its portability and low power consumption [2].

V. Methods and Materials

A. Leveraging Convolutional Neural Networks

One of the main tools used to improve image classification algorithms, especially with animals, is Convolutional Neural Networks [4]. A Convolutional Neural Network (CNN) reduces image size while helping identify patterns in the image. For example, a CNN may look for a triangle pattern similar to the top of a dog's ear, in which it would produce an image with high values mainly near the dog's ear in the picture given. This also creates the opportunity to use layers of CNNs that look for different patterns, which identifies more features, making the input data more useful and accuracy higher [4]. This project will be utilizing CNNs prebuilt and pre-trained in the models YOLO Tinyv4 and ssd_mobilenet_v2 (run on Tensorflow).

While Convolutional Neural Networks help greatly increase the performance of computer vision algorithms, they can also cause an increase in processing requirement with their multitude of layers. This is why it's important to choose an optimal resolution and layer count for the intended task. Reducing the resolution of each layer can greatly help with increasing performance because it reduces the total amount of data the neural network must analyze [3]. However, this may reduce the accuracy of the algorithm because some data may be lost, but this effect is rarely large. The other primary approach to increasing performance of a CNN is by reducing the amount of layers, which similar to reducing resolution, will increase performance while potentially decreasing accuracy [3].

B. YOLO

Most work with object detection and CNNs use classifiers to perform detection, in which regions of interest in an image are selected and then convolutional neural networks are run on each region of the image. This generally results in a slow algorithm because predictions need to be run sometimes thousands of times on a single image. YOLO, or "You Only Look Once," however, frames detection as a regression problem. A regression algorithm detects bounding boxes and class probabilities for the whole image in one run of the algorithm. Because of this, YOLO is able to be extremely fast, while maintaining comparable if not superior accuracy over other implementations. This speed is very important for our application as we are attempting to create a camera that needs to be able to make decisions in real time as a pet approaches the door. Additionally, implementing an algorithm onto the NVIDIA Jetson Nano, with its limited processing power, requires a fast algorithm that can be easily run on the small computer. This single prediction approach to object detection also benefits the algorithm by allowing the prediction to be based on the global context of the image rather than separated points of interest in the image.

In order to use the algorithm contained in YOLO for pet detection, we require pre-trained weights obtained by running pictures with their correlated annotations through a training command. The data must first be obtained in the right format. A collection of 2,576 photographs of pets annotated by their species or breed from the University of Oxford was downloaded online in the darknet TXT format to serve as a baseline dataset. Additional photos were added to designate the specific pet that is to be recognized by the door. These photos were manually labeled using the labelimg software that allowed us to draw bounding boxes around an object to be recognized, and label that object as whatever we wish. This was used to label exactly where the face of the pet is in the photo.

After uploading these photos and their labels to the device, a config file was downloaded and updated for information such as the image width and height, batches, steps, and number of classes. Next, additional files were created that the program used to determine which of the uploaded files are to be used in the testing and training parts of the algorithm. This is done through a processing python script that randomly copies the title of each file into a text document labeled either "train.txt" or "test.txt". When the training program runs, it uses the files with names in the training file to check the weights and update on them, and then checks them again. After the training is complete, files with names in the test file were used to evaluate the performance of the newly created weights with fresh data. Additionally, files and directories were created to direct the training program to all of the information it needs, including the correct files to use, which classes are being looked for, and the correct classes within each of the photos.

With all of the data uploaded, and relayed to the YOLOv4-tiny command, the pre-built training algorithm can begin to run. This program ran through the files in the testing set, randomly adjusting the weights until it reached the set number of batches, or randomizations, specified in the configuration file. This program returns files containing the weights for the neural network at every thousandth iteration as well as the weights that it believes to be the best. However, sometimes these weights can become "over-trained" and become too good at detecting the features of the data in the training set, but will not be applicable to other data that will be used in the future. To solve for this, each set of weights and the thousandth interval were tested for mean average precision on the testing data and graphed on a line chart. If the line ever begins to fall after a certain point, this is proof of over-training and the peak weights will be used for the final algorithm instead.

The metrics of mean average precision, average intersection over union, F1 score, and the framerate were calculated on a uniform dataset including one unique pet, and a comparison can take place between alternative methods to finally evaluate the implementation of the most effective algorithm in the use case of opening a pet door.

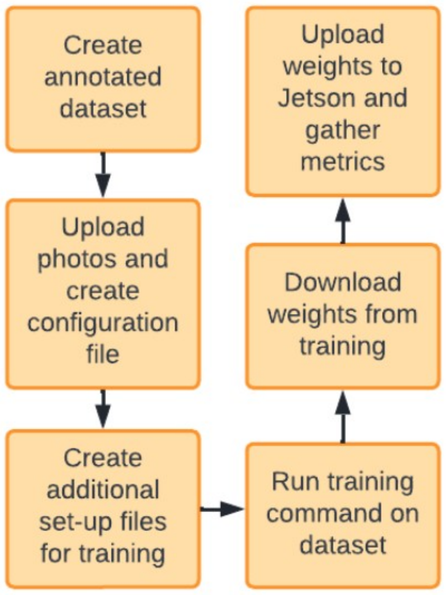

Create annotated dataset
Upload photos and create configuration file
Create additional set-up files for training
Run training command on dataset
Download weights from training
Upload weights to Jetson and gather metrics

Fig. 2. YOLO workflow

C. *Tensorflow*

One would be correct to assume that object detection algorithms are on the more complex side of machine learning, but with Google's Tensorflow API and python module, creating a neural network for object detection can be done with a couple lines of code. Google also provides pre-trained state-of-the-art models of varying speed and accuracy in the Tensorflow model garden. These pretrained models make it much easier to create and train complex object detection models, but they have the downside of not being as customizable and optimizable as a model from scratch. This project will be employing the ssd_mobilenet_v2 model on Tensorflow and its optimization modules. This model stands for single-shot-detector in which it takes one pass over the image to detect objects, making it optimal for speed and algorithms for mobile devices, and though it takes one look at the image, the ssd_mobilenet_v2 model still performs with rather high accuracy.

While Tensorflow makes machine learning development much easier, the models are still very difficult to run and consume a lot of memory. To help with this, Tensorflow has provided optimization tools such as TensorRT, which are able to greatly reduce the computational load of the model while maintaining almost all of its accuracy. Tensorflow provides an easy to use interface with a multitude of models to choose from, along with seemingly endless support resources, which is why it will be employed for this project.

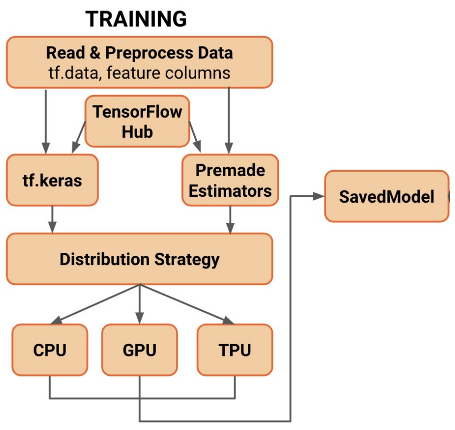


Fig. 3. Tensorflow workflow

D. TensorRT

Because Tensorflow uses a very large amount of memory and resources, optimization modules are required to simplify the models to be run on smaller devices. One of these optimization modules is TensorRT. TensorRT optimizes a model to be a bit simpler, but focuses more on having the model run as efficiently as possible on NVIDIA GPUs, which is why it's so important for the Jetson.

TensorRT utilizes multiple optimization techniques for the model. It first starts by reducing the precision of the numbers within a model. This involves things such as changing from floating point 32 numbers to floating point 16 numbers. This makes the model simpler to run while theoretically only sacrificing a small percentage of performance. It then takes multiple methods including Layer and Tensor Fusion, Kernel auto-tuning, and Dynamic Tensor Memory to help optimize the model to run more efficiently on GPU memory and faster on GPU processing.

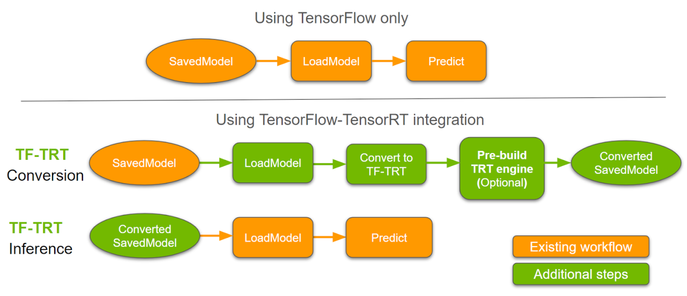


Fig. 4. TensorRT workflow

E. tfLite

Similar to tensorRT, tfLite is a Tensorflow model optimization module, however, it optimizes the model to be as simple as possible to run on the single core of a CPU, which is why it's much more common for systems such as the Raspberry Pi.

With the pure goal of reducing the size and load of the model, tfLite employs three primary optimization techniques. It first reduces the precision of the numbers within the model (ex: going from FP32 to FP16). It then goes through the pruning process by removing minor parameters, resulting in more efficient model compression. Lastly, to further simplify each layer, it replaces the layer weights by clustering them and replacing them with the centers of these clusters, thus reducing the amount of neurons in the layer, which reduces the model complexity.

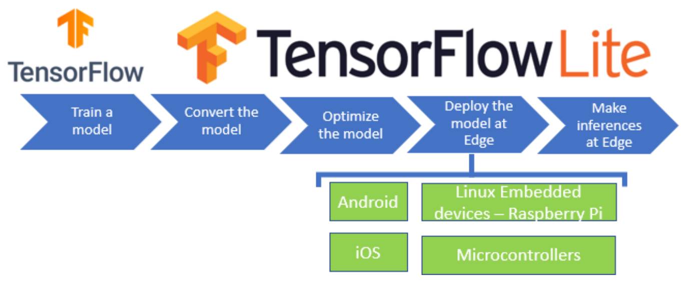


Fig. 5. tfLite workflow

VI. Results

Algorithm	FPS	mAP	AP for Bella	Memory usage (GB)	GPU usage	CPU usage
Yolo	3.8	0.497	0.8474	1.026	100%	30% (Average for cores)
tfLite	0.78	0.620	0.420	0.127	0%	25% (100% one core)
TensorRT	3.82	0.846	0.848	**4.9	100%	50%
*Tensorflow		0.751	0.384			

* only run on desktop
** used Swap memory past 2GB

VII. Discussion

A. FPS and processing usage

TensorRT and YOLO both achieved nearly 4 fps, which is plenty for the application, while tfLite only achieved 0.78 fps. However, tfLite used significantly less resources, which may make it viable if the rest of the application needs a lot of resources to run.

B. Average Precisions

In terms of performance between YOLO and TensorRT, both had high average precision for the unique dog (in this case Bella), which can be explained by how her breed was not in the dataset, making her the only dog of her class while the other breed classes had different looking dogs of the same breed. However, the models' average precision varied much more for the other breeds with TensorRT maintaining a rather high AP and YOLO falling lower. What's more surprising is that TensorRT beat Tensorflow by a large margin, which is very unexpected since they should lose a little performance in the optimization process. This result between Tensorflow, tfLite, and TensorRT, along with YOLO and TensorRT should be investigated in future research to determine if it's a repeatable occurrence or a lucky event.

C. Memory Usage

The last notable result is the drastic difference in memory usage between tensorRT and YOLO, in which tensorRT utilized 4.9 GB of memory, resulting in it having to offload 2.9 GB to Swap memory. This is because the project is utilizing TF-TRT rather than dedicated TensorRT, which makes the program load the entirety of Tensorflow in the process, thus resulting in the large memory usage and a 20+ minute load time. Using dedicated TensorRT should have a more comparable memory to YOLO and similar prediction performance to the TF-TRT method utilized.

VIII. Conclusions

As the results show, tensorRT and YOLO can both provide adequate performance for computer vision on the NVIDIA Jetson Nano, however, we found that YOLO development was easier for beginners compared to Tensorflow. While tfLite does set aside resources that could be used for other tasks in an application, it's performance and lack of GPU utilization makes it unsatisfactory unless the application specifically needs it. In all, this application of a pet doorbell system is very feasible on the NVIDIA Jetson with both Tensorflow's tensorRT and YOLO's Tinyv4 algorithms providing adequate speed and accuracy for the task.

IX. Limitations

There were many limitations with our methods both with some of the algorithms and the dataset. Our dataset was limited in that all the pictures of the custom pet/dog were posed pictures, meaning that the dog was looking at the camera in each one. This could have potentially caused some overfitting where the models would not see the dog in the actual application because the dog will not always be looking at the camera.

There was also a limitation with the implementation of TensorRT. TF-TRT was used for this project, but it is slower compared to the other implementation method for TensorRT. This comes from TF-TRT having to load the entirety of Tensorflow and still running with unoptimized Tensorflow input and output.

X. Future Research

In future research, it would be beneficial to both improve upon the implementation of this project and try other methods. To improve the implementation of this project, it would be best to first better diversify the dataset of the custom pet and look at implementing the faster version of TensorRT. It may also be beneficial to look into other possible methods for the project such as facial detection algorithms, which could potentially be better suited for this task.

Aside from increasing performance and accuracy of the project, future research can investigate why there was such a large performance gap between Tensorflow and TensorRT. It's expected that TensorRT optimized models lose accuracy compared to Tensorflow, however, the TensorRT model made significant accuracy gains in all classes for this project. There are some possible mechanisms for this, such as dropout, but a more in-depth investigation is needed to be more certain.

XI. Acknowledgements

A special acknowledgement to Gavin Alberghini at the MITRE corporation for providing expert guidance and encouragement as we took a deep dive into the complex field of machine learning and computer vision. We acknowledge him for creating an environment that allowed us to get involved in and finish a project like this with such little experience. Another acknowledgement goes to Ms. Vobrak and the New Horizons Governor's School for Science and Technology for giving us this opportunity and setting up the program. This mentorship project would have probably never happened without them and we are very grateful for it.

References

[1] Animal companions are freezing to death in cold temps. PETA. (2022, April 20). Retrieved May 18, 2022, from https://www.peta.org/features/animal-companions-winter-freeze-death/

[2] Giubilato, R., Chiodini, S., Pertile, M., Debei, S. (2019). "An Evaluation of ROS-compatible Stereo Visual SLAM Methods on a NVIDIA Jetson TX2." Industrial Engineering Dept., University of Padova

[3] Mattal, Sparsh. (2018). "A Survey on Optimized Implementation of Deep Learning Models on the NVIDIA Jetson Platform." Indian Institute of Technology Roorkee.

[4] Süzen, B. Duman and B. Şen, (2020). "Benchmark Analysis of Jetson TX2, Jetson Nano and Raspberry PI using Deep-CNN," Isparta University of Applied Sciences.

[5] Zhenxing Zhou, Yisiang Neo, King-Shan Lui, Vincent W.L. Tam, Edmund Y. Lam, and Ngai Wong. (2020). "A Portable Hong Kong Sign Language Translation Platform with Deep Learning and Jetson Nano." The University of Hong Kong.