**TEL AVIV UNIVERSITY**

School of Electrical Engineering

**Deep Neural Network Onboard an Unmanned Aerial Vehicle**

A project submitted toward the degree of

Master of Science in Electrical and Electronic Engineering

by

**Sagiv Dvash**

This research was carried out in The School of Electrical Engineering

Under the supervision of Mr. Yonatan Mandel and Prof. Ben-Zion Bobrovsky

August 2019

**Contents**

**Abstract 3**

**1. Introduction 4**

**2. Methods 6**

2.1 ROS 6

2.2 Deep Neural Networks 6

2.3 Dronet 6

2.4 Nvidia Jetson TX2 6

2.4.1 TensorRT

**3. Design 7**

3.1 Base Component 8

3.2 TensorRT Component 8

**4. Usage 9**

4.1 Hardware Setup 9

4.2 Software Setup 11

4.2.1 Keras Only

4.2.2 TensorRT

4.3 Running 12

4.3.1 Keras Only

4.3.2 TensorRT

**5. Results 14**

5.1 Performance 14

**6. Discussion and Conclusions 15**

**7. Future Work 16**

**8. Bibliography 17**

**A. Appendix: Working with the lab’s server 18**

**Abstract**

Unmanned Aerial Vehicles are becoming increasingly present in day to day life as well as research topics, most notably due to the advent of advanced computer vision algorithms. Recent hardware advances allows running such algorithms on the UAV itself, as opposed to remotely, enabling a step forward towards autonomous UAVs. In this project I developed a framework for using deep neural networks on an embedded module - Nvidia Jetson TX2. I also demonstrate how to use it for Dronet - an algorithm for drone navigation that is executed in real time, allowing the drone to navigate and avoid obstacles autonomously.

**1 Introduction**

Recent years have seen numerous advances in automating Unmanned Aerial Vehicles (UAVs, Drones). Some implementations are based on classical methods like SLAM [1] and some are using Deep Neural Networks (DNNs) [2,3,4]. There are also steps to unify these methods [5]. Until recently, all these methods were too heavy to be performed on embedded modules: SLAM requires tracking and mapping hundreds of points, while DNNs are known to require powerful hardware, specifically - GPUs, to run process video feeds in real time.

The solution for this issue was typically to perform these computations on a remote, more capable server. The input (e.g. video stream and other sensor data) would be transmitted through to the server where it would be processed and flight instruction would be sent back to the drone. As embedded chips are increasingly becoming more powerful, a natural transition will occur, moving the entire control block onto the UAV. Such transition will allow for the drone to be completely autonomous.

In this project I created a user-friendly framework\*, that allows rapid development of DNNs on the Nvidia Jetson TX2 platform as the UAV controller. To demonstrate, it is mounted and connected to a Parrot Bebop 2 drone and uses an implementation of the Dronet [2] deep learning navigation algorithm.

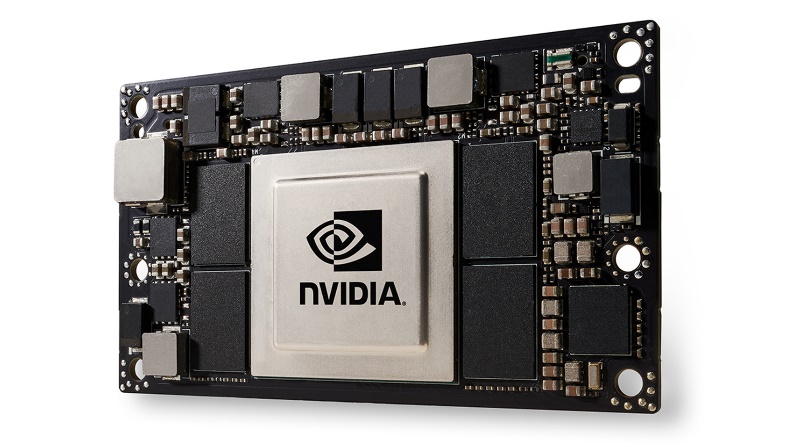


Figure 1: Nvidia Jetson TX2

\* Project repository hosted on Github at <https://github.com/tau-adl/DroNet_TX2>

**2 Methods**

**2.1 ROS**

ROS is the de-facto standard for robot software development. It is a collection of software frameworks that provides easy to use tools for integrating every part of the system - from sensor acquisition, through data processing, to control. It is not the most efficient system for such tasks, as it strives to be very generic, but it is the simplest and indeed the most widespread.

**2.2 Deep Neural Network**

Neural Networks, a family of machine learning algorithms, were re-introduced as Deep Neural Networks (DNNs) or as "Deep Learning" several years ago [6], when the then-shallow neural networks were deepened and re-implemented on GPUs to the required heavy computations. The concept is to present a set of learning neurons numerous samples (e.g. images) and matching desired result (e.g. "dog"/"cat" tags) during the training session, and back-propagate the error gradient until the neurons learn to generalize the features and predict the correct result.

**Tensorflow** (TF) is an open source library developed by Google to streamline the development of DNNs. **Keras** is another open source library, which works on top of TF to ease its use. Currently it is used as Tensorflow's official high level API. Using such libraries allow quick prototyping and easy development of even complex NNs.

**2.3 Dronet**

Dronet [2] is a custom ResNet[7]-based DNN, which was trained for drone navigation. The DNN input is a single image and the output is two scalars - a steering angle (-pi,pi) and a collision probability (0,1). It was trained on two datasets - the first is the Udacity [8] dataset of driving vehicles, which was used to estimate the desired steering angle of the UAV. The tags are taken from the actual steering sensors on the vehicles. The second dataset was custom made by the Dronet team by filming bicycle rides and was used to estimate the collision probability. It was hand-tagged with 0 probability when the bicycle path was clear and 1 when an object was close.

Aside from the DNN itself, the algorithm uses LPFs to smooth the speed changes and rotation commands of the UAV. The algorithm performs moderately in outside environment, following the general direction of the road and stopping when close to an object. It was not designed for indoor environment but the authors stated that in their tests it performed well in such scenarios.

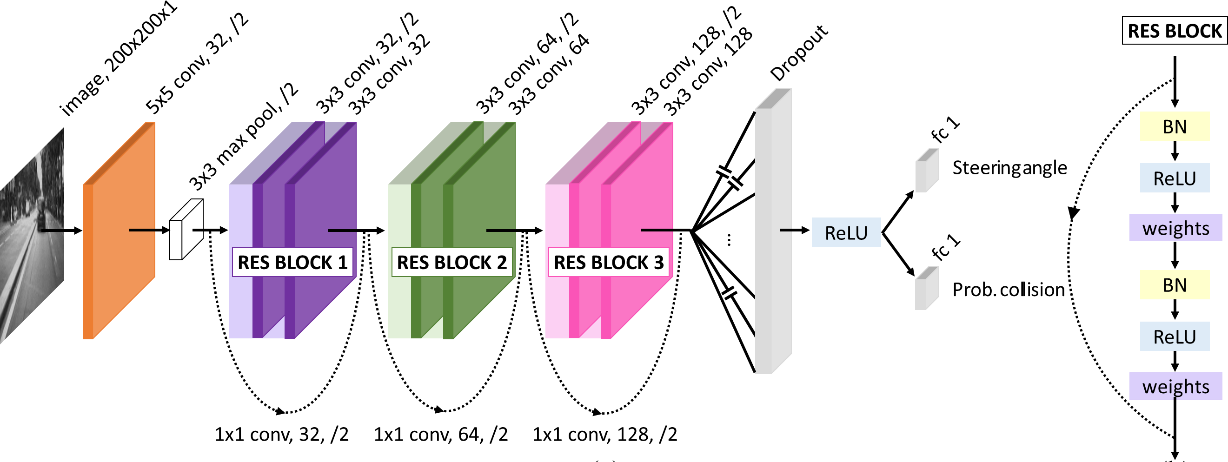


Figure : Dronet Structure

**2.4 Nvidia Jetson TX2**

Nvidia Jetson TX2 is an embedded System-on-Module (SoM) with ARM cortex A57, 8GB 128-bit LPDDR4 Memory and integrated 256-core Pascal GPU, all packed inside a 50x87mm package weighing 85 gramms. When optimized it takes up to 7.5 Watts and when fully utilized uses up to 15 Watts. It runs Ubuntu 16 with CUDA drivers out of the box and can easily support ROS. CUDA is required to utilize the GPU, and is used by the DNN frameworks. The size, weight and power consumption allows it to be mounted stright on a medium-sized UAV and connected to its power source.

**2.4.1 TensorRT**

TensorRT is an optimization software by Nvidia for DNN. It can be used to optimize a NN model on a specific GPU in order to achive one of the following: maximum throughput (inferences per second), maximum efficiency (performance per watt), minimum latency (time per inference), Accuracy (correctness of answers) or memory usage capping.

**3 Design**

As mentioned, the HW setup is comprised of a Parrot Bebop 2 with Nvidia Jetson TX2 mounted on top of it. The Orbitty carrier board is used to power and allow interface with the TX2 (full system in fig. 8). The Bebop is used as a WiFi Access Point (AP) to which the TX2 connects as a client. Other clients (e.g. a controller's laptop) can connect to the AP as well and SSH to the TX2 through the drone in order to control it.

On the SW side, ROS is used to connect the different parts of the system (see fig. 3). Bebop Autonomy [9] is a handy ROS component that provides a full API for the Bebop drone over WiFi. In this project, it is used to receive the video feed and forward it through ROS as single frames to the DNN. ROS then takes the DNN output, sends it to another thread that apply the control algorithms and sends the commands back to the drone through the same Bebop Autonomy library.

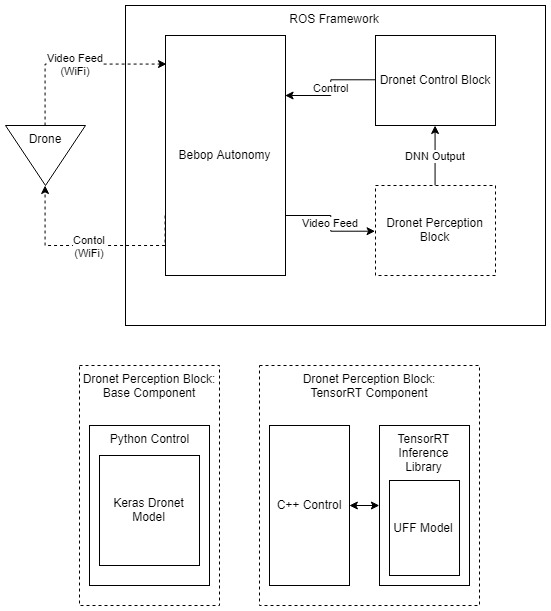


Figure 3: Software Framework Diagram

During the development, it came as a surprise that the Dronet DNN is not a bottleneck on the TX2, and no optimizations were required on this specific DNN for it to perform in real time (i.e. 30 FPS). This is described below as "Base Component". In order to support deeper and more sophisticated DNNs, a "TensorRT Component" was also developed.

**3.1 Base Component**

This component uses *rospy*, an implementation of the ROS API in python. It takes the frames efficiently from the video feed and uses the Keras model to get the output for every frame.

**3.2 TensorRT Component**

This component uses *roscpp*, an implementation of the ROS API on C++. It is required since TensorRT is not supported natively on python. It behaves exactly the same as the python implementation, except that the DNN is utilized through the TensorRT API instead of Keras.

In order to keep the code simple, the TensorRT DNN is implemented as a standalone dynamic library (.so) that is linked to the ROS component in build time. This means that a different network can easily be used with very few parameter changes and no changes to the ROS component. To prevent performance issues, no data is actually copied between the modules - only pointers are passed.

**4 Usage**

**4.1 Hardware Setup**

1. Remove the bebop battery and remove the screws holding the bebop cap, then remove the cap itself (fig. 4)
2. Remove the screws holding the GPS and compass module: (fig. 5)
3. Solder wires to the 11.4V and GND terminals of the battery (fig. 6) and pass them out of the cap through the bottom (fig.7)
4. Finally, attach the TX2 with the Orbitty carrier board to the drone and connect the power (fig. 8)



Figure 4: Remove Cap Screws

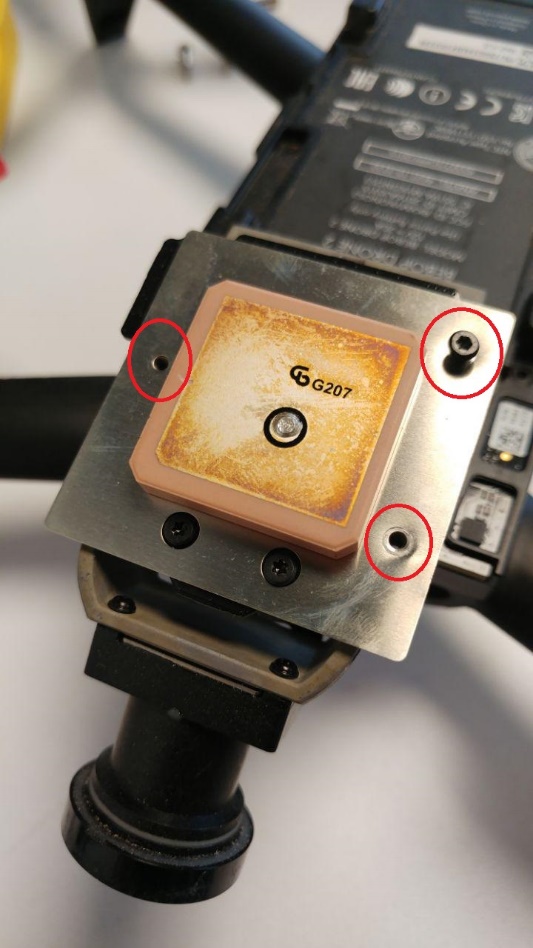


Figure 5: Remove Navigation Module Screws

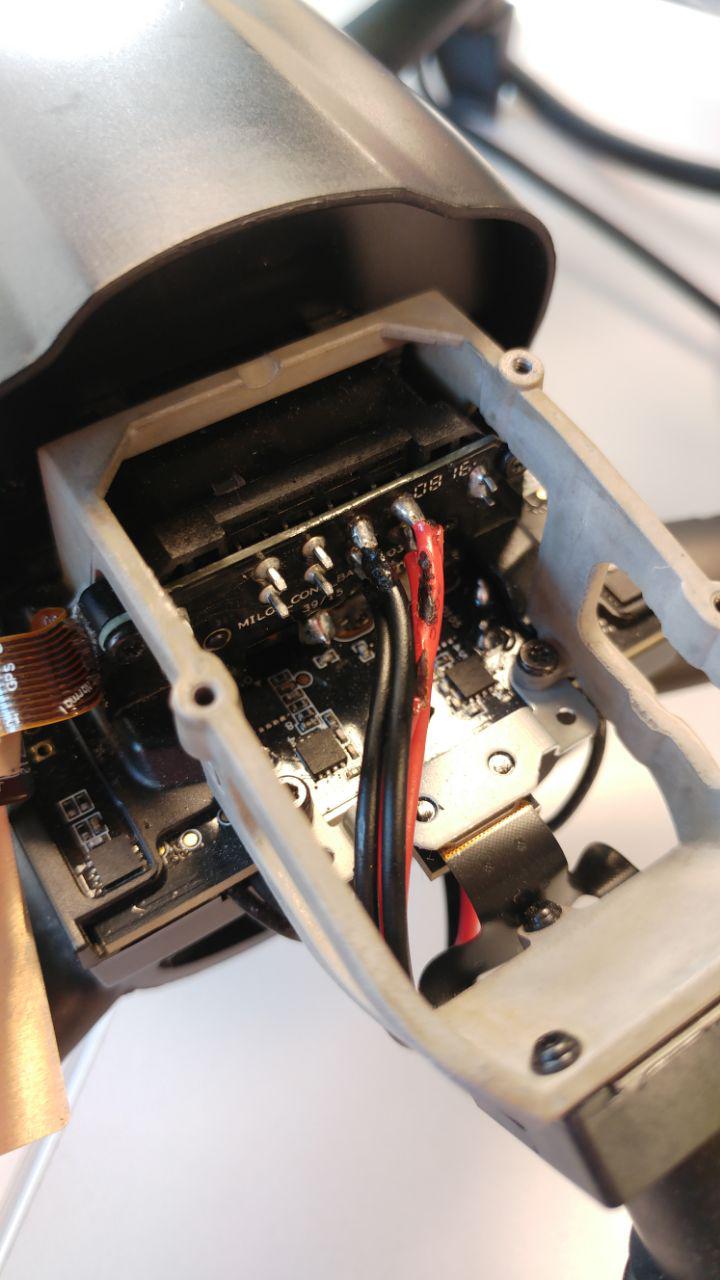
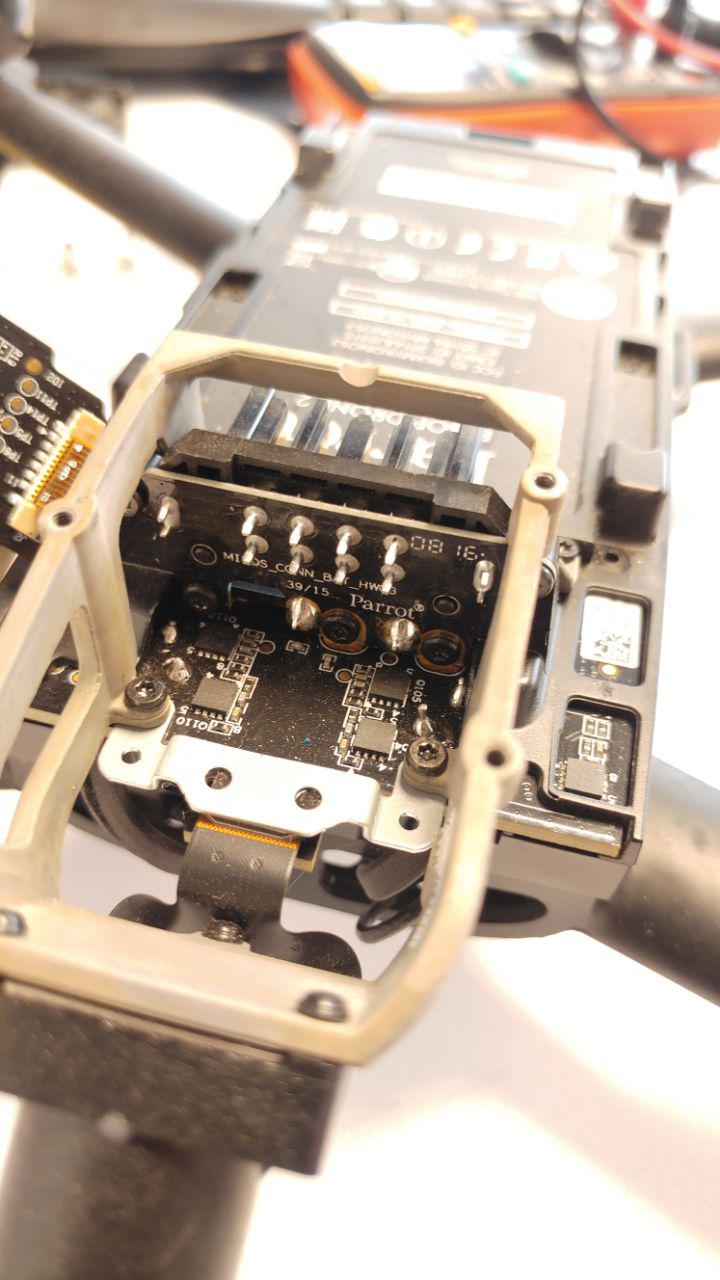


Figure 6: Solder Power Wires



Figure 7: Pass the wire out of the cap

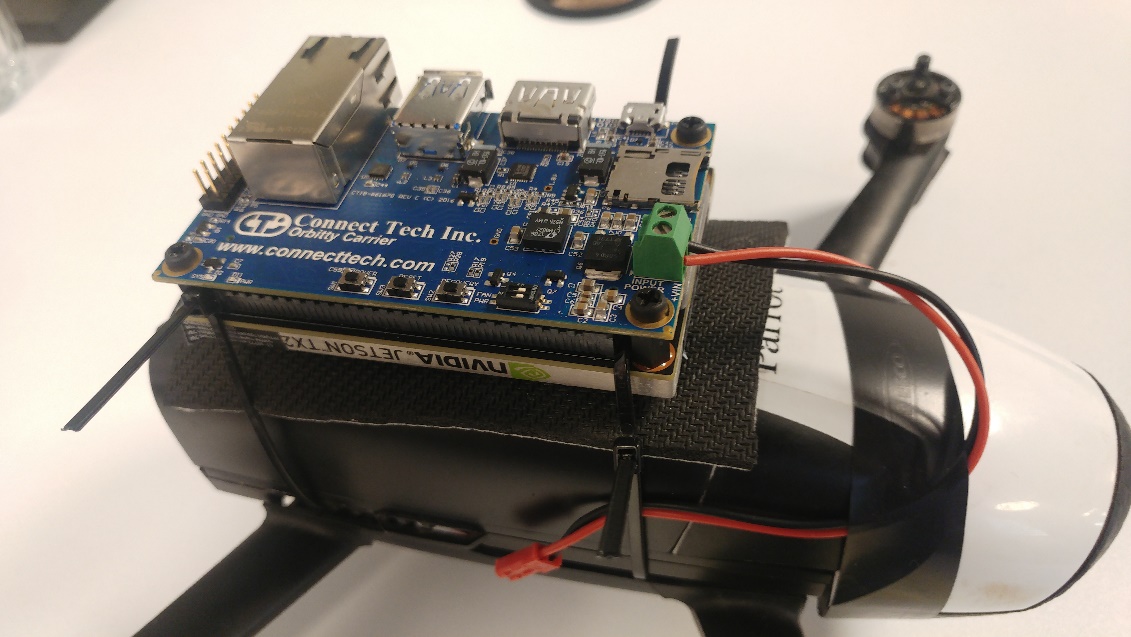


Figure 8: Final Setup

**4.2 Software Installation**

As already mentioned, I provide two ways to run NN on the TX2. One using regular trained keras models, and one using optimized TensorRT models. The latter require additional installations and setup stages. Both are explained below:

**4.2.1 Keras Only**

* Jetson TX2 installed using Jetpack 3.3 (already includes CUDA, CUDNN drivers)
* Install python 2 with Numpy (optional - matplotlib)
* Install Tensorflow + Keras with CUDA support
* Install ROS Kinetic (desktop version)
  + Install ROS and setup the workspace according to the dronet installation notes [2]
  + Get this project's [source](https://github.com/tau-adl/DroNet_TX2) and overwrite the dronet\_perception component.
  + Compile ROS packages: dronet\_perception, dronet\_control and bebop\_autonomy from dronet github

**4.2.1 TensorRT**

On host PC:

* Install Python and Keras with Tensorflow backend. CUDA is required, but a GPU is not.
* Jetpack 3.3
* Get the PC-component project [source](https://github.com/tau-adl/DroNet_TX2) code.
* Using the scripts included, translate the Keras HDF5 model to a Tensorflow PB model, using the included application:

python keras\_to\_tensorflow.py --input\_model="path/to/keras/model.h5"  
--input\_model\_json="path/to/keras/model.json"  
--output\_model="path/to/save/model.pb"

* Translate the Tensorflow PB model to UFF model: python pb\_to\_uff.py  
  If a different model than the default is used, the contents of the script should be altered to match it.

On Jetson TX2:

* Jetson TX2 installed using Jetpack 3.3 (already includes CUDA, CUDNN drivers and the TensorRT software)
* Install Python 2 with numpy (option- matploblib)
* Install ROS Kinetic (desktop version)
  + Install ROS and setup the workspace according to the dronet installation notes [2]
  + Get this project's [source](https://github.com/tau-adl/DroNet_TX2) and copy over the dronet\_perception\_trt package
  + Compile ROS packages: dronet\_perception\_trt, dronet\_control and bebop\_autonomy
* Get the [project](https://github.com/tau-adl/DroNet_TX2) TensorRT inference library and:
  + Compile the TensorRT inference library (trtinference) by running make solibs in the base source directory.
  + Setup the paths for the trtinference.h and the newly created libtrtinference.so in the CMakeLists file.

Alternatively, flash a cloned image with the entire project/prerequisites already installed on the TX2 EMMC. This requires Jetpack to be installed on the host. See [instructions here](https://elinux.org/Jetson/TX2_Cloning).

**4.3 Running**

**4.3.1 Keras Only**

* Launch dronet\_bebop.launch, dronet\_launch.launch (from dronet\_perception) and deep\_navigation.launch (in this order). This will connect to the drone and stream the video feed into the neural network, displaying its predictions. The prediction will then be picked up by the control block that can send commands to the drone (OFF by default)
* To issue the drone to start to fly, send the command

rostopic pub --once /bebop/takeoff std\_msgs/Empty}

**WARNING: THE DRONE WILL NOW START TO HOVER**

* To issue the autonomous navigation, send the command   
  rostopic pub --once /bebop/state\_change std\_msgs/Bool "data: true"

**WARNING: THE DRONE WILL NOW SELF NAVIGATE**

* To stop the autonomous navigation, send the command   
  rostopic pub --once /bebop/state\_change std\_msgs/Bool "data: false"
* To order the drone to land, send the command  
  rostopic pub --once /bebop/land std\_msgs/Empty

**4.3.2 TensorRT**

NOTE: There is an issue with the TensorRT inference model, which outputs different values than the regular (Keras) model. This is under investigation with Nvidia (see [discussion here](https://devtalk.nvidia.com/default/topic/1055217/tensorrt/model-accuracy-penalty-with-tensorrt-on-jetson-tx2/)).

On the Jetson TX2:

* Launch dronet\_bebop.launch, dronet\_launch.launch (from dronet\_perception\_trt) and deep\_navigation.launch (in this order). This will connect to the drone and stream the video feed into the neural network, displaying its predictions. The prediction will then be picked up by the control block that can send commands to the drone (OFF by default)
* To issue the drone to start to fly, send the command

rostopic pub --once /bebop/takeoff std\_msgs/Empty}

**WARNING: THE DRONE WILL NOW START TO HOVER**

* To issue the autonomous navigation, send the command   
  rostopic pub --once /bebop/state\_change std\_msgs/Bool "data: true"

**WARNING: THE DRONE WILL NOW SELF NAVIGATE**

* To stop the autonomous navigation, send the command   
  rostopic pub --once /bebop/state\_change std\_msgs/Bool "data: false"
* To order the drone to land, send the command  
  rostopic pub --once /bebop/land std\_msgs/Empty

**5 Results**

The different implementations + models were first tested against previously recorded video feeds of a car that were not seen by the NN ("validation datasets"). Qualitatively, the regular Keras model performed well, keeping the collision prediction and steering value as expected with respect to the video feed content. The TensorRT model has shown different results that seemed unrelated to the content of the video.

Although it was not the main part of the project, the navigation was also tested in various experiments "in the field". The navigation part of the model did not perform well, especially in close quarters (e.g. corridors) and had varying success outside. It was most successful when placed in environment that resembles an actual road. The collision avoidance worked relatively well in both environments. No real life experiments were performed with the TensorRT model due to the above issues.

**5.1 Performance**

Both ROS components (Keras and TensorRT) run inference on the bebop video feed with 30 FPS, which allows both to run in real time. In both cases the CPU and GPU are not fully utilized, allowing for additional logic and algorithms to run in parallel. The TensorRT model takes on average half the time of the Keras model, which for both models takes the bulk of the processing time of each frame (other parts being frame copy and image preprocessing).

**6 Discussion and Conclusions**

As mentioned, the actual navigation capabilities of the Dronet algorithm is mediocre, but the TX2 platform along with the TensorRT and the entire developed framework allows much more complex algorithms to be evaluated.

Moreover, additional efforts could be directed to optimizing the TX2, which can focus in either maximizing performance by increasing the clock speeds of the different processors (using jetson\_clocks) or to minimize power consumption by shutting down some of the CPU cores (using nvpmodel) as well as throttle maximum clock speeds.

The Parrot Bebop 2, while providing a relatively easy-to-use framework along with the Bebop Autonomy ROS component, is not perfect. The software running on the drone is not open-source, and restricts the user to certain usage patterns (e.g. Bebop must be AP, no easy way to use wired connection).

I also encountered technical problems during the development due to the fragile nature of the video feed coming over WiFi to the ROS system and into the NN node. I assume using a more robust channel would mitigate this issue (e.g. physical wire instead of wifi). In addition, the drone can only operate as a WiFi Access Point, which forced the TX2 to be used as client in order to get the video feed. This in turn forced me to also connect with the controlling laptop as a client to the drone and route my activation command through the drone - which is not preferable.

Using the TX2 while mounted on the drone provides great processing power for complex algorithms - much more than required by the "simple" Dronet.

**7 Future Work**

From DNN perspective, many algorithms can be tested with the framework. This includes, but not limited to, different network structures, networks trained on other datasets and combination of several networks ("ensembles").

It is also possible to use other ("classic") algorithms for computer vision and navigation, as these can also benefit from the GPU parallel computing capabilities.

From HW point of view - the TX2 has an excess amount of computing power when compared to the task performed here. It is possible to try and implement the same NN and setup on a smaller, cheaper and less power-hungry SoM like the Jetson Nano. On the other hand, this shows that heavier, smarter algorithms can be performed on the same HW. Furthermore, for allowing readers who are new the field to better understand the paper, we have gathered relevant terms that are essential for this purpose.

In summary, after reviewing this paper, the reader is able to follow the PTAM logic, to install and debug its code and to understand how to utilize and modify it for his use.

**8 Bibliography**

[1] Nitin J. Sanket et al. “Gapflyt: Active vision based minimalist structure-less gap detection for quadrotor flight”. In:IEEE Robotics and Automation Letters3.4 (2018), pp. 2799–2806.8

AUGUST17, 2019

[2] Antonio Loquercio et al. “Dronet: Learning to fly by driving”. In:IEEE Robotics and Automation Letters3.2(2018), pp. 1088–1095.

[3] Dhiraj Gandhi, Lerrel Pinto, and Abhinav Gupta. “Learning to fly by crashing”. In: 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2017, pp. 3948–3955.

[4] Ram Prasad Padhy et al. “Deep Neural Network for Autonomous UAV Navigation in Indoor Corridor Environments”. In: Procedia computer science 133 (2018), pp. 643–650.

[5] Rong Kang et al. “DF-SLAM: A Deep-Learning Enhanced Visual SLAM System based on Deep Local Features”.In: arXiv preprint arXiv: 1901.07223 (2019).

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “Imagenet classification with deep convolutional neural networks”. In: Advances in neural information processing systems. 2012, pp. 1097–1105.

[7] Kaiming He et al. “Deep residual learning for image recognition”. In: Proceedings of the IEEE conference oncomputer vision and pattern recognition. 2016, pp. 770–778.

[8] Udacity. “Udacity Self-Driving Car Driving Dataset”. In: (2016). URL: <https://github.com/udacity/self-driving-car>.

[9] Bebop Autonomy ROS library.https://bebop- autonomy.readthedocs.io/en/latest/. Accessed: 2019-07-09.

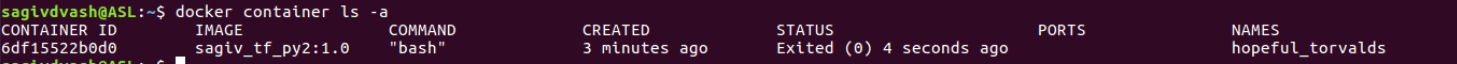
**Appendix A: Working with the lab’s server**

During the project, a server was acquired in order to allow students train their NN models. I compiled a basic How-To guide to help new users start up. We use Docker to separate the environments and applications of different users. A shared directory was created to allow users to share datasets. Each student that needs access to the server should receive access to the university VPN as well as personal user on the lab's server.

**First Setup**

After setting up and connecting to the university VPN ([explained here](https://www.tau.ac.il/cc/helpdesk/communication/sslvpn/sslvpn.html)):

* Use SSH to remote-access the server:  
  ssh <username>@asl.tau.ac.il
* Create your own Docker container with the environment that fits your need (your desired NN Framework/version, specific Python version, specific Python packages, etc.).  
  Example for Python 2 with latest Tensorflow version:  
  docker run --runtime=nvidia -it tensorflow/tensorflow:latest-gpu-jupyter bash  
  Example for Python 3 with latest Tensorflow version:  
  docker run --runtime=nvidia -it tensorflow/tensorflow:latest-gpu-py3-jupyter bash
* (optional) Install packages you want inside the container. e.g. pip install keras.
* Exit the container by running: exit  
  (it should now be the most recently terminated container)
* Check your newly-created container\_id number:  
  docker container ls -a  
  Example output:



* Save the image with your name:  
  docker commit -a <yourname> <container\_id> <image\_name>:<version>  
  e.g. docker commit -a sagiv 6df15522b0d0 sagiv\_tf\_py2:1.0
* You should now see it in the docker images:  
  docker images

**Common Workflow**

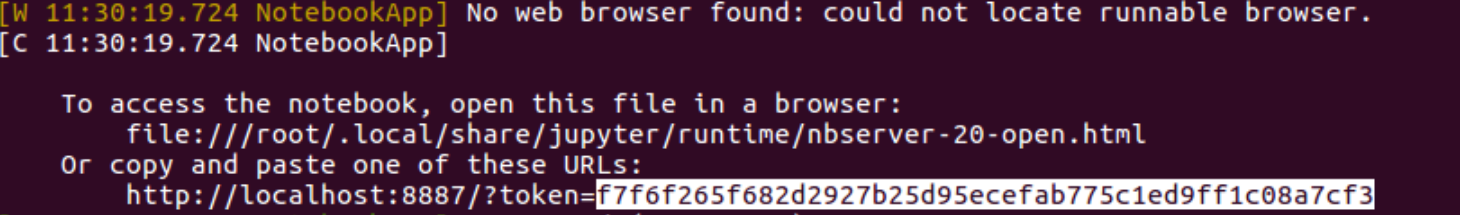
* **Load your saved container:**as super user  
  docker run --network host --runtime=nvidia -it <image\_name>:<version> bash  
  as regular user:  
  docker run -u $(id -u):$(id -g) --network host --runtime=nvidia -it <image\_name>:<version> bash

To use shared directory (e.g. datasets directory) from host, add -v /var/datasets:/dataset

Be careful not to use this with super user to avoid changing the files from within the container.

* **Saving your container:**Exit the container and save it with Docker commit (see First Setup). You can overwrite previous images in this way. **NOTE: if you will not save your container after you exit, changes from the loaded images will NOT be saved!**
* **Cleaning up unused container:**Every time a container is launched, all changes are saved until cleaned. This history log is not necessarily needed and takes storage space. It can be cleaned by running:  
  docker container prune  
  **WARNING: SAVE YOUR CONTAINER BEFORE THIS STEP**

* **Use jupyter to edit files inside the container from you local PC:**Inside docker:  
  (first time only) jupyter notebook --generate-config  
  cd /your/code  
  jupyter notebook --port 8887 --no-browser  
  You will need to use the token provided. e.g.



On local PC, run:  
ssh -N -L localhost:8888:localhost:8887 <your\_username>@ASL.tau.ac.il   
(you will be prompted to enter password, and nothing will happen after. This is ok)  
  
Open your browser and enter:  
localhost:8888  
Use the token you got from the jupyter server.