TEL AVIV UNIVERSITY

Deep Drone Racing

David Sriker
Tel Aviv University
Electrical Engineering
David.Sriker@gmail.com

Lidor Karako
Tel Aviv University
Biomedical Engineering
LidorKarako@gmail.com

February 2020

ABSTRACT

Autonomous drones is becoming of great interest in the last few years in various fields such as sports, agriculture, military etc. With the raising demand for good reliable autonomous drones the attempt to adjust in real time for dynamical changing environments, inaccurate state estimation and limited resources are the fundamental challenges that the field is currently involved in improving. Here we address the issue of vision based autonomous drone racing. A racing drone should be able to traverse a track, possibly changing environment, fast. The combination of state-of-the-art control system with the power of convolutional neural networks (CNN) perceptual awareness on images enabled the desired functioning. Except the advantage that the combination of those 2 systems are both platform and domain dependant, due to advances in the the field of deep learning we could use synthetic data only to train our model without a negative effect on the CNN performance; not only that due to the fact that the data was synthetic the CNN generalized better and showed a better adjustments capabilities to dynamical changing environments (illumination condition, gates appearance and background). This work demonstrate how to do *zero-shot simulation to reality* transfer in the field of agile drone flight.

1. INTRODUCTION

In the past few years there was an increase attention for drone racing sports, and it even have been televised. This sports involves professionals pilots that fly small quad-rotor drone based on first-person view (FPV) through a complex track at high speeds. Those skills can also be served in fields such as structure inspection in the military services or emergency responses where the drones can safely aids in rescuing wounded [1]. Due to the fact those tasks and more need professionals to fly the drones in the most efficient way it came to realize the autonomous driving/flying might be of great importance and became the interest of many research groups.

Developing a fully autonomous racing drone is a demanding task due to the amount of consideration that need to be accounted such as dynamic modeling, real-time perception, spatial localization, trajectory generation and optimal control. Due to those challenging aspects of autonomous drone racing there was a meteoric increase in the research community addressing this issue and also giving rise to several autonomous drone racing competitions [2, 3].

If flying an aerial robot through different gates along complex routs in an autonomous manner is not challenging enough, now days, the competitors are ought to use GPS-free techniques. one approach to autonomous racing is to compute the desired global trajectory in advance in account for state variations. This technique is not dynamic enough, and thus, participant teams attempted to localize the drone and predict the target's spatial trajectory, with methods such as Simultaneous Localization And Mapping (SLAM). This is consider as a chicken-or-egg problem, where the map is needed for localization and the pose estimation is needed for inferring a map. SLAM requires prior information about the scene, such as probabilistic approach for the control commands given relative observations, hence just as well missing the true purpose of autonomous meaning [4, 5]. In addition, while trying to account for minimum trajectory, SLAM methods attempt to recompute localization in order to reduce drift phenomenon, however by doing so the computational demands are increasing [5].

Moreover, the theoretical justifications and connections to different error metrics is a field that is not highly researched. SLAM pipelines enable navigation only in a predominantly-static world, where the waypoints and other parameters for collision-free trajectories are not defined for dynamic scenes [5]. There are already those whom suggest to justify the trajectory evaluation problem in SLAM with a probabilistic framework, where the error concepts are replaced with likelihood terms and the ground truth labels are modeled as random variables [6].

On this basis, and in order to improve the detection performance, deep-learning models are developed. This allows for increased robustness and better inference speed when compared to previous classic algorithm schemes. The learning models are often constructed with CNN architectures, which can be easily introduced with the modification needed in order to optimize the trade-off between accuracy and speed of

Unmanned Aerial Vehicle (UAV), such as drones [3, 6].
The trained neural network models can be deployed upon drones with supported hardware and executed with the appropriate PID controller. Thus, it is possible to have the same UAV robot performing a variety of different tasks, such as object detection models, fully offering all of the deep learning capabilities for the purpose of deep drone racing [4, 7].
Real-time on board processing is desired for many autonomous UAV applications, such as drone racing. Nevertheless, in the case of having prior information about the track trajectory and locations of the target gate, it is not a simple task to accomplish. The continues flight result in dealing with large streaming quantities of data and requires for complex calculations and demanding minimum-jerk trajectories optimization schemes. Urgent for this task of creating fully autonomous aerial experience, different models were constructed for navigation, based on variety of architectures such as LSTMs, autoencoders and CNNs [7]. The lastly mentioned is utilized for the purpose of our research, where we aim to use the fast processing capability of well-trained CNN based models in order to complete the task of GPS-less navigation for deep drone racing [5, 8].
Instead of relying on global state estimations, we adapt a CNN approach to identify waypoints in local frame coordinates. This eliminates the problem of drift and simultaneously enables robust system navigation through dynamic environments. These predicted waypoints are then fed to a state-of-the-art (SOTA) PID controller, which for a short trajectory segment generates motor commands for real-world navigation. Having output representations of discrete navigation commands predictions enables high robustness but leads to low agility. Instead, direct control can lead to high flight agility, but suffers from sample complexity as well as being less robust to dynamic scenes [8]. In this work the resulting system combines the perceptual awareness of trained CNNs with the precision to predict relative coordinates and speed for the controller to execute. Taking the best from both approaches, we can allow computations to run fully onboard [5, 8].
Falling into the niche of track specific is not desirable when taking on major drone racing competitions, as the human operator may end on top of this one. The perception system can be invariant to dynamically changing environments by training on non-photorealistic simulation images and the performance is well comparable to training process with real world data [5].
In this work we implement a learning based approach for agile drone navigation. The solution is based on zero-shot simulation-to-reality transfer, where our training process is applied with simulated data and the testing is performed on live stream images acquired by the drone. The existing implementation of the work we are based on was performed by team ASL/ADRL from the Robotics and Perception Group from the University of Zurich (UZH), Switzerland. Currently, the work suffers from a limited frame rate execution, which is only synthetically account for with the highly dynamic scene augmentation as further described in section 3. In addition, they are having much redundancy from the programming direction, on which for the perceptual part of the model they are using the **TensorFlow-v1** API (TF-1.x) [5], Sim2Real GitHub repository. With that, they are implementing a modification of the original architecture, *DroNet*, which was also developed in their lab which having different input image size and reducing the number of filters by incorporating a capacity factor of 0.5 [5, 9]. Until now, although this is one of the leading works in the field of deep drone racing, and have competed in the highest levels of drone racing challenges and was one of the top three, it still lacks optimizations schemes that can further improve the performance.
In this paper we introduce several modifications of our on. First, we present our work when using the effectiveness power of **TensorFlow-v2** API (TF-2.x), which integrates better with *Python* runtime using "Eager Execution", which is an imperative programming environment that evaluates operations immediately, without building graphs, operations return concrete values instead of constructing a computational graph to run later. Working in this mode deprecates usage of mechanisms that attempt to help users find their reusable variables. Since there is no graphs build process and was in older version of TF, there is no need to use the old cumbersome methods such as "session.run()" any more. Although, it still acts as a function, they are hidden operations from the user. In addition, in TF-2.x many redundant API's are removed. The bottom line is that when using "Eager Execution", the interface is much more intuitive for *Python* data structures, it is easier to debug with standard debugging tools and error traceback calls and the entire flow is simply more natural as for there is no need of graph control flow, which is more ideal for dynamically examining different models. Besides "cleaning" the code from unnecessary computationally complex consuming parts, we have further put use to the basic TF training flow, and obtained custom training via different callbacks. We aimed to do this in order to offer the possibility to convert our saved models to a **TensorFlow-Lite** (TF-lite). TF-lite includes quantization tools to assist users running their TF models on embedded systems, which fits well with on-device inference for deep drone racing. It's framework contains a model optimization toolkit to reduce the model's size while preserving the original efficiency with minimum deviation from the original unquantized model [10].
We demonstrate our method in real autonomous agile flight scenarios, using a vision-based quadrotor on a designated drone-racing track. Our suggested end-to-end solution is well tested and evaluated with the previously mentioned SOTA navigation work. We aim to offer similar performance with faster processing capabilities.

2. RELATED WORK

Winning a race requires taking the autonomous robotic system platform to the edge. Several technologies can be utilize in order to fundamentally challenge both perception and control. For example, different sensory modalities as LIDAR's or event-based cameras, are popular for navigation solutions to improve overall imaging experience. Nevertheless, they are neither that cheap nor accessible and are still considered bulky, which is on the contrary to the reason why in general drones today are at most popularity. In addition, from control perspective, there are many works regarding enabling of high-speed navigation, lately mainly for the autonomous industry. It is achieved either by hardware or by algorithmic improvements, but inherently it cannot be straightforward adapted for small, agile quadrators for onboard sensing and computing [5].

In this section we will further put into details the approaches that have been used to overcome many related problems in autonomous navigation.

2.1. Data-driven Algorithms for Autonomous Navigation

This type of approach focuses mainly on autonomous driving with the idea of end-to-end learning a navigation policy from large amount of annotated data. A recent work by Loquercio et al. [9] proposed to overcome this all-or-nothing approach by using data from ground vehicles in a transfer learning manner and another work by Gandhi et al. [11] offered to utilize data from the flying platform itself. Nevertheless, these data-hungry methods are suffering from either high sample complexity or either low generalization to variant scene conditions with training limited to specific scenarios. The perception and control are separated for efficient transfer of sensing data to variety of different task domains [5]. Our network enable high-speed, agile flight by making the output of our perception module compatible with fast and accurate model-based trajectory controllers.

Another approach, although stands alone from autonomous racing implementation at the moment, is the stereoscopic first person view (FPV) for the human operator. In this work the authors described the depth perception capabilities of their optical setup and mention the system can collect training data for machine learning algorithms in order to become fully autonomous [12]. We are aware the work of Loquercio et al. [5] that collecting real world ground truth labels for their deep learning model to be trained from was extremely difficult and had the authors working mainly with stimulative data [5, 8].

2.2. Gate Detection for Drone Racing

Drone racing competitions popularity keeps on drawing attention, with team MAV-lab of TU Delft from the Netherlands winning the last *Alpha AI* competition using a 72 gram drone and winning a 1 million dollar check [13]. Their solution is based on advanced computer vision algorithms with a note regarding future for deploying deep learning models. The main contribution is a proposed visual model-predictive localization (VML). Gate assignment is achieved via *snake-gate* algorithm, which detects proximal pixels with similar color and consequently finds the gate corners and color. A prior of known gate size is applied to determine the drone's relative position to the gate. In addition, they estimate which gate on the racing track has the highest likelihood for being in the field-of-view (FOV), and transform the relative position to the gate to a global position measurement. Their sensor can provide high-frequency, and quite accurate attitude estimation by means of an Attitude and Heading Reference System (AHRS). The VML approach predicts lateral acceleration based on attitude estimates. It fuses the low-frequency onboard gate detections and high-frequency onboard sensor readings to estimate the position and the velocity of the drone. Simulation and real-world flight experiments show that VML can provide robust estimates with sparse visual measurements and large outliers. The work of the USRG group from KAIST, Korea, is utilizing object detection deep learning model for real time implementation on NVIDIA Jetson TX2 [4]. In their work they propose the development of ADRNet to deal with detection of moving gates. It is composed of a SSD network with a modified backbone, composed of variant of AlexNet, instead of the originally VGG-16. As their primary objective is to perform single gate detection, they reduced the number of unnecessary high-level features in the AlexNet module. Utilizing the information about the center point of the gate they were able to guide and control the drone to pass through the gate. The draw back in using this simplified object detection schemes is that the average precision is reduced dramatically.

2.3. Transfer from Simulation to Reality

The field of few-shot and zero-shot learning is intensively researched due to its compatibility with variety of real world problems, ranging from classification to object detection and synthetic data generation [5]. The work of Balaji et al. [14] combined zero-shot simulation-to-reality and reinforcement learning (RF) into their autonomous racing car training policy. They used an algorithm that utilizes two neural networks in the training phase. The first is a *policy* network that selects which action to take and the second is a *value* network that estimates the expected cumulative reward given input image. They initialize the model using random actions and the result is an interaction with the simulation environment that maximize the actions that gain a higher reward. They claim to obtain high speed that is not with model based controller, but rather with RF methods that allow for high performance even in a short time training period. This is a promising approach as the authors of [5] themselves mentioned combination of RF in their training as a future plan.

3. DATA

The entire training data was synthetically generated. In order to collect a diverse training data from simulation we performed visual scene randomization while fixing up to small perturbations the track layout. To get as much generalization as possible we randomized the following visual scene properties:

- ***Texture of the background, floor and gates.*** Background and floor textures were picked randomly from a pool of 30 diverse synthetic textures, while the gate texture was drawn from a pool of 10 texture (mainly red/orange in order to better distinguish them from the environment).
- ***Gates shape.*** Gate's shape was chosen from a pool of 6 different shapes that each is approximately the same size as the original starting gate.
- ***Illumination conditions.*** Illumination was separated to 2 components:
  - ***Ambient light*** is constant lighting. It is the light an object gives even in the absence of strong light. It is constant in all directions and it colors all pixels of an object the same.
  - ***Emissive light*** is light that is emitted by an object.

  Both illumination elements are drawn from uniform distribution, for the ambient light the support is [0,1] and for the emissive light the support is [0,0.3], per each of the textures (background, floor and gates).

The simulation is highly dynamic and allows to create various in-flight FOV perspectives of the drone regarding the gates. Figure 1 shows a sample of images from the training and validation data sets, where one can easily observe the diversity in training data.

Although the simulation accounts for numerous scenarios that should allow zero-shot training schemes, it may be possible to increase efficiency of the training process with additional real-world data. As mentioned in [5], it is not easy to collect and align real-world data images, but using their already annotated validation dataset, one can account for the gap between going from zero-shot learning to testing on non-simulative data. Having natural images in the training loop introduces physical features that will be more specific to the description of real-world drone racing track. This approach will enrich the dataset and will open the possibility to few-shot training, using only a few real-world images. We attempt to go in this direction, as will be further detailed in section 4.

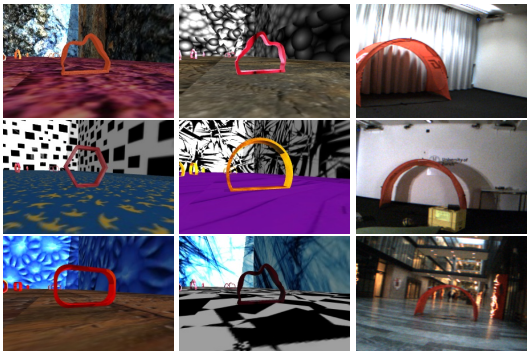


Figure 1: *Two left columns: an example of 6 different simulation training images. Right column: real world images used during the validation process. The different background textures and gate shape, as well as different illumination conditions can be seen*

4. METHODS

We address the problem of robust, agile flight of a quadrotor in a dynamic environment, tested in high frame rates. Our approach makes use of two separated subsystems: perception and control. The perception system uses a CNN based predictor to obtain direction to the next gate in local image coordinates, along with the desired navigation speed. This is obtained from a single image collected by a forward facing camera. The control system uses the perception system's predictions in order to generate a minimum jerk trajectory that is tracked by a controller. Along this sections we describe the different details regarding the two subsystems, where the contribution of our work is among the scope of of the perception system.

Perception system. We implement the perception system part by a CNN that analyzes an input 300×200 RGB simulation image and provides a $\{\vec{x}, v\}$ tuple, where $\vec{x} \in [-1, 1]^2$ is a two-dimensional vector in normalized image coordinates, that encodes the direction to the next gate, and $v \in [0, 1]$ is the desired relative speed, normalized by the maximum possible velocity.

Control system. For the inference part of the model, we have created an un-normalization function, built for the purpose of restoring the normalized triplet tuple coordinate and speed for on demand costume visualizations. The controller part repeats this in a similar manner, where it takes an input $\{\hat{x}, v\}$ tuple, converts the two-dimensional normalized image coordinate to three-dimensional local frame coordinates by means of back-propagation. It is performed along the camera projection ray and thus derive the goal point at a depth equal to the prediction horizon d, between the drone's coordinate and the desired position.

4.1. Training Policy

We train the perception system in a supervised manner, in order to imitate automatically learning for automatic generation of global optimal trajectories. In order to obtain the trajectories, we take on two assumptions:

1. During training time the location of each gate on the track is known. Although in reality the inference time dictates the frame-time a gate will appear at.

2. We additionally consider that the quadrator has access to accurate state estimates with respect to latter reference frame, meaning the drone is always in the range of being accurate to the target.

Expert policy. The global trajectory t_g is obtained for the case that it passes through all of the track's gates. The training data is generated for the perception network by simulation that implements an expert policy. During simulation flight, the closest point on the trajectory is computed with respect to the quadrator position. The target coordinate that our perception network learns is $\vec{p}_{c'} \in \mathbb{R}^3$, which is the projection of the drone's location at distance $d \in \mathbb{R}$ from the optimal point on the trajectory. In practice, as we can see in figure 2 and as in verification that will be furthered shown in the *Experiments* part, the ground truth (GT) point will not always point to the next gate, but rather always direct to the projection on the image for which the point, $\vec{p}_c \in \mathbb{R}^3$, is at distance d from the quadrator on the optimal trajectory. Intuitively, always pointing to the next gate is not the an optimal strategy, in particular if the racing track reference trajectories contain narrow curves such as passing close to a wall.

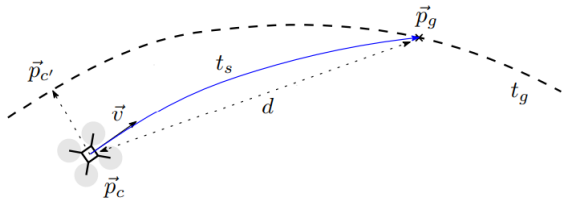


Figure 2: *The global trajectory curve and the drone's mapping toward the optimal target coordinate [5]. The pose $\vec{p}_c$ is the quadrator is projected on global trajectory, t_g, with the purpose of finding point $\vec{p}_{c'}$. A point, $\vec{p}_g$, on the global trajectory point that is at a distance d from the location of the quadrator is the desired goal position. At each frame the drone is pushed toward the current part of the global trajectory, or the reference trajectory, by means of short trajectory segment, t_s*

Objective function. Our loss is consistent of velocity prediction loss and coordinate prediction loss, for a combined total loss of weighted mean-squared-error (MSE):

$$L = \frac{1}{N}\sum_{i=1}^{N} \|\vec{x}_i - \vec{x}_{g_i}\|^2 + \gamma \cdot \|v_i - v_{g_i}\|^2 \qquad (1)$$

where $\vec{x}_{g_i}$ denotes for the GT image coordinate, v_{g_i} is the GT velocity value of sample i and γ is a weight factor.

Experiment plans. In our first experiments, we compared the training results from the original suggested model, compiled with four different optimizers [15]:

- *Adam.* It is one of the most popular optimizers used in neural network training today. It was considered a fast converging optimizer that generalize worse in comparison to SGD and SGD with momentum. On the parctical note, it may be simpler to achieve any convergence at all with Adam as it is most insensitive to hyperparameters selection and to initializations. This optimizer utilizes two decay factors for the first and second moments, and we used $\beta_1 = 0.7$ and $\beta_2 = 0.9$, where both are used in momentum approach by using moving average of the gradient and the squared gradients to adaptively scale the learning rate.

- *SGD.* Often implemented with step-size learning rate scheduals, the stochastic gradient descent (SGD) is a very strong optimizer in terms of generalization power. Even though it is easy to implement, it accounts only to the current gradient state, and tuning it for convergence may be a difficult task, as it is not an adaptive technique. That is why it is often implemented with momentum technique, which accounts for past gradients.

- *Adagrad.* Designed to deal with sparse and highly unbalanced data. If we use the same learning rate for all every feature-resolved CNN, then the infrequent features will be updated too slowly compared to the frequent ones. Might not be ideally as specifically stitching a solution per training phase, but in order to avoid using multiple learning rate, AdaGrad can be used for sparse data. It performs large learning rate scaling for more sparse parameters and smaller updates for less sparse parameters. One drawback of AdaGrad is that it treats all past gradients equally and mainly focus on the first order. In addition, there is still a hand tuned global learning rate, which can make the method sensitive to initial large gradients. If this occurs the learning rates will remain low for the rest of the training and thus increasing the global learning rate will be required.

- *Adadelta.* While SGD, momentum and Adagrad are sensitive to the learning rate selection, Adadelta is not. Although derived from Adagrad, this method dynamically adapts over time using only first order information and requires no manual tuning of a learning rate. It is robust to noisy gradient information, different model architecture choices, various data modalities and selection of hyperparameters. The method utilizes a single decay factor, ρ.

All models were initially trained with default γ value of 0.1. After tuning the best optimizer over 100 epochs, we have moved forward and trained the selected model with various values of γ-s for a range of epochs. Each model was evaluated for its loss and RMSE, visualization vs the GT labels and frame rates. Hence, we could compare different selected models and examine their matching TF-Lite variations.
We then trained 2 additional models where our plan was to reduce the model size significantly while preserving the robustness and ability of the perception block to generalize which is described more thourouly in appendix 7.2. In those additional models we took the optimizer and γ that we found after fine-tuning the original network as described at section 5.

4.2. Model Training and TF-2 Methodologies

In order to train our models and also perform the code conversion to a lighter tensorflow model we used TF-2.0.0rc1 version with CUDA 10.2, cudnn 7.6.2 running on a Linux Ubuntu 18.04.1 LTS laboratory server docker environment. The training process utilized the docker's T4 16GB GPU for a near 10 minutes per epoch running time.
TF-2 syntax. Focusing only on the section of code that was used to train the model, with tensorflow v2 syntax we reduced tremendous amount of unnecessary placeholder calls and memory demanding visualizations.
TF-Lite Compression. TF-Lite is an open-source deep learning framework for on-devices inference. It appears that the TF-Lite model file we had created weights approximately 3.5 times less than the original TF model.

Our models were trained using batch size of 128 images, learning-rate of $1e-4$ for all the optimizers, besides the Adam optimizer who was set with learning-rate of $1e-6$. We validate the inference running time for all different models with respect to the TF and the TF-Lite saved models. Additional information is given on appendix 7.1.

4.3. Trajectory Generation by the Controller

This paper highlights the perceptual part of the system, as so, we do not describe thoroughly about the controller part. Even-though, some details are given regarding its design and usage of the learned parameters according the guidelines in the work of Locauercio et al.[5]. First, the global trajectory is generated using the predicted coordinates, calculating the path to the center of the gate. The velocity is set by the predicted relative speed and normalization factor of the maximum speed possible. Following, the motor parameters are set and finally the desired three-dimensional location is obtained. During training a minimum distance from the optimal trajectory is used, and during inference from practical reasons a larger distance is set.

We first used a DroNet like architecture, as mention in section 1. We apply a capacity factor of 1 and discard the use of batch normalization (BN) layers as the original authors claim they do not contribute to an improved performance. Removing the

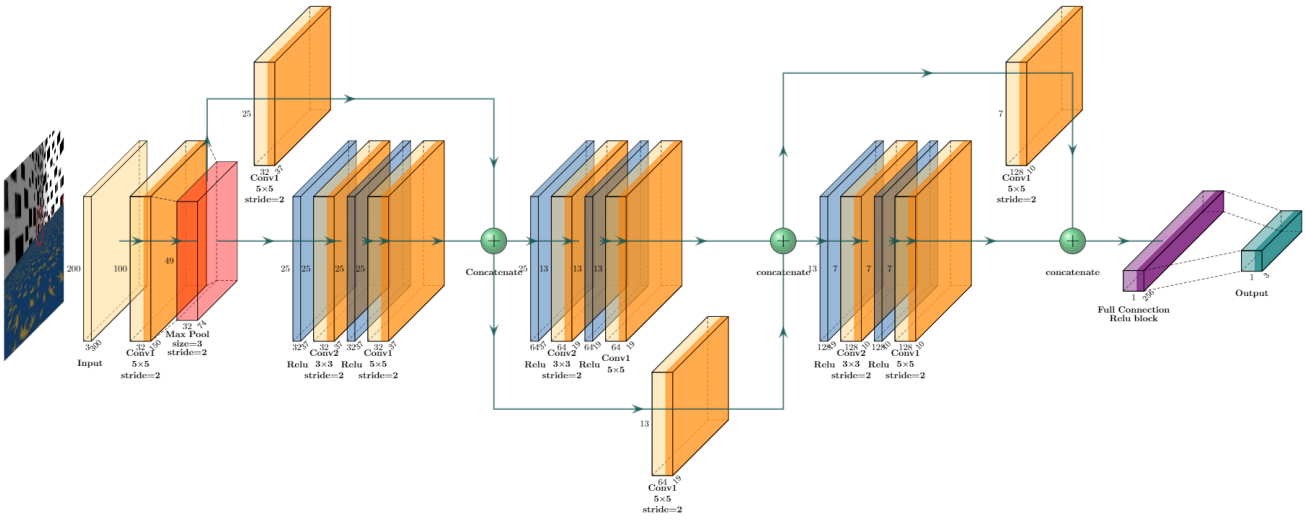


Figure 3: *Original architecture as presented in [5]*

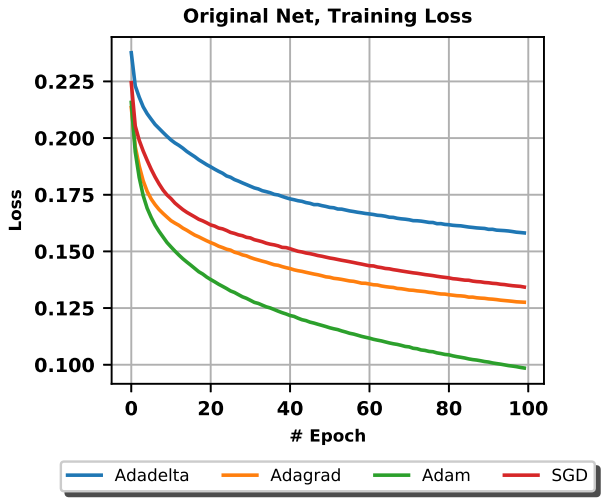
BN should allow increase in inference frequency. Although residual blocks reduce the problem of vanishing gradients and thus allow to have a deeper model, we limited the model to only 3 resnet-blocks for high speed onboard performance. The architecture contains about 5 million parameters and can be seen in figure 3.

5. EXPERIMENTS

We have implemented several experiments in order to obtain number of possible models to run on-board and from them to choose the best fit with respect to performance and execution time.

Each model was trained for a total of 100 epochs and its weighted-loss function was calculated with a weight factor of $\gamma = 0.1$ as was used in the original paper [5]. The training loss convergence plot is shown in figure 4. We can observe that the Adam optimizer reached less than 10[%] loss. The validation loss and Root Mean Squared Error (RMSE) trends are shown on figure 5 and figure 6 respectively.

The validation results offer the Adam optimizer as the best optimizer. The Adam optimizer shows the highest potential among all examined optimizers as even though it has not yet been fully converged, it has the lowest loss and RMSE values. Moreover, we can see that using Adam the model better generalized as the slope in the plots is negative both for the loss and RMSE. The loss and RMSE values from the last epoch for each optimizer experiment are presented in table 1.



X

Figure 4: *The training loss per epoch for a given optimizer for* $\gamma = 0.1$

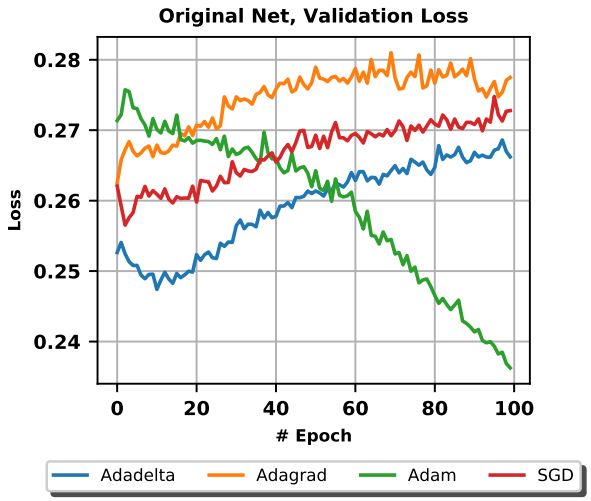


Figure 5: *Loss per epoch for a given optimizer for* $\gamma = 0.1$

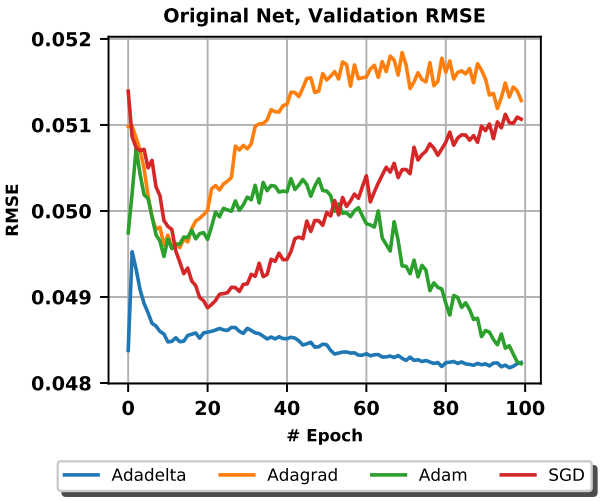


Figure 6: *RMSE per epoch for a given optimizer for* $\gamma = 0.1$

Table 1: *Results for the original model trained for 100 epochs with 4 different optimizers with a weight factor* $\gamma = 0.1$

Optimizer	Loss [%]	RMSE [%]
SGD	27.2	5.11
Adadelta	26.7	4.83
Adagrad	27.6	5.13
Adam	23.7	4.83

In addition to evaluate the performance of the network we also asses the execution times of the models with both TF and TF-lite models the effect of the optimizer on the frame per

second (FPS) for both types of models can be seen in figure 7. We use the most simplified quantization algorithm for our model without affecting the performance and gain improved speed as can be seen on table 2.

Table 2: *FPS results for ResNet8* ***TF*** *and* ***TF-Lite*** *models* per oprimizer

Optimizer	TF [fps]	TF-Lite [fps]
SGD	83.9	164.9
Adadelta	61.5	162.0
Adagrad	83.7	163.7
Adam	82.8	164.7

As shown in figure 7 the results are with no doubt a proof that the TF-Lite model ran twice as fast then the original TF. Both models are practically the same in terms of prediction power. We have calculated the loss value for each model incorporated with different optimizer and there is no significant difference when one another, which means the TF-Lite model can be used for getting inference results on an actual drone.

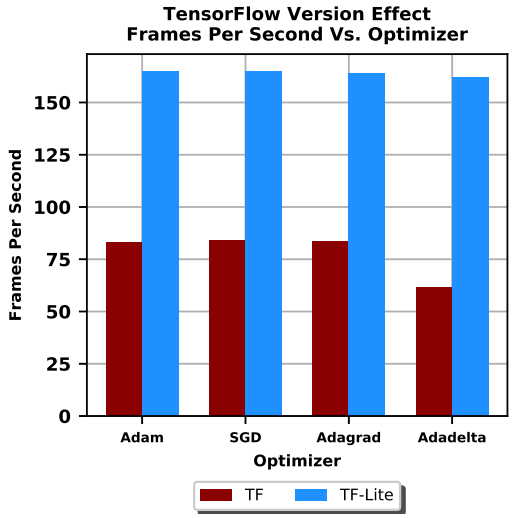


Figure 7: *Frames per seconds (FPS) for different values of optimizers. Evaluated using γ=0.1, while the network is trained for 100 epochs*

We have selected γ=0.1 to train every selected model, compiled with the Adam optimizer, due to its robust performance in longer training processes. The selected loss as function of epoch for the training and validation datasets is shown on figure 8, and presented for the selected weight factor and optimizer.

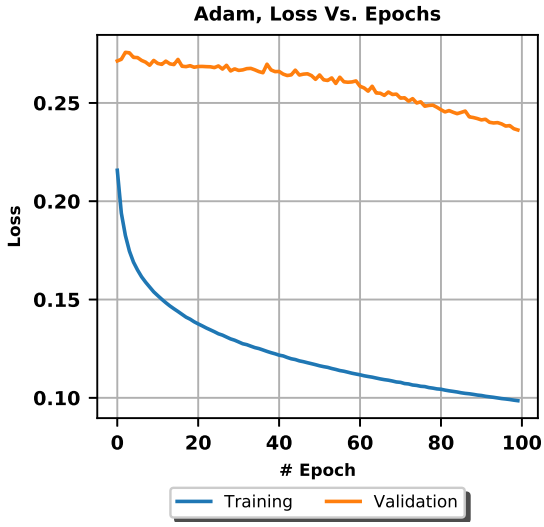


Figure 8: *The training and validation loss value per epoch for the Adam optimizer. The model was trained with weight factor of γ=0.1.*

After concluding that it is best to continue with the Adam optimizer, we arranged for 4 different γ values experiment to check the influence of the weight factor on the performance. Each value was evaluated for 20, 40, 60, 80 and 100 epochs each time. The bar chart in figure 9 shows the results by means of RMSE on the validation set.

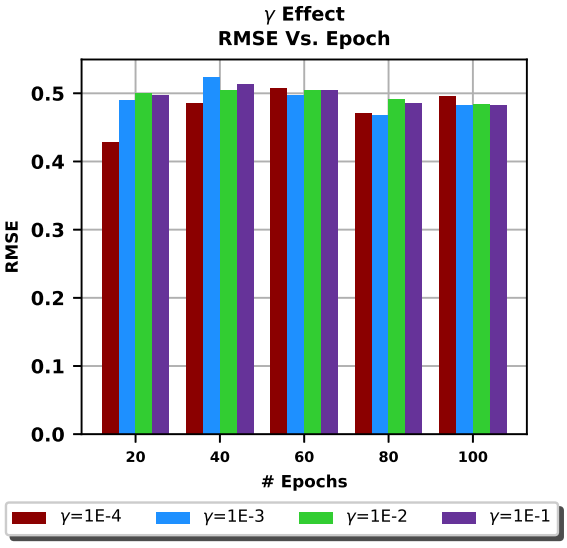


Figure 9: *RMSE for different values of γ. For each γ, the network is trained each time for a different amount of epochs. This results in 20 experiments for ResNet8 using the Adam optimizer. The RMSE value is recorded from the last validation epoch and presented at each bar*

As can be seen from figure 9 the γ value has little influence on the generalization of the network.
After selecting the best possible model for only limited amount from the training data and analyzing it for small run of 100 epochs, we experimented with larger training data and for longer training period. The training loss improved and it is now approximately less than $7[\%]$, while the validation loss, shown for the original architecture in figure 10, reduced to be less than $10[\%]$. Comparing both training processes, when we doubled the number of epochs we more than doubled our final validation loss. Comparing for 100 epochs on both validation fit, we notice that the error is lower regarding the new training, it is smoother and converges faster. This is due to the fact that we not only validated on more larger validation dataset, but in addition planted a few real-world images in the training dataset. We of course did not test our models on neither the training data or similar data and all testing were performed on different drone run images to avoid any overfitting scenario.

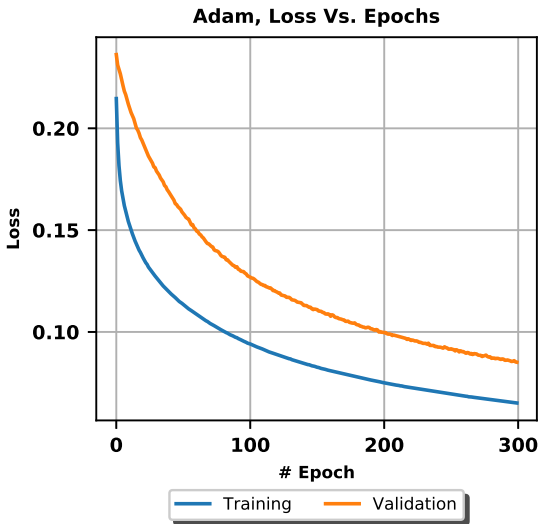


Figure 10: *The training and validation loss value per epoch for the Adam optimizer. The model was trained with weight factor of γ=0.1. for 300 epochs. We validated on larger data, and inserted a very few amount of real-world images to the training dataset.*

Jetson TX2 experiment. We were able to activate our code on top of the Jetson TX2 after installing the appropriate jetpack version via the NVIDIA-SDK and adjusting it to work with tensorflow 2 which is only supported with python 3. Due to the fact that the entire communication with the Parrot drone was implemented in advance in python 2, it is currently not supporting combination with our implementation. Although the mentioned issue we nevertheless performed a simulation on top of the Jetson where we read images and put the prediction with the GT and we observed that indeed the network performed well. It is currently presented in Deep Learning TAU repository.

In figure 11 we present several images regarding the detection capabilities with the selected Adam optimizer and weight factor γ=0.1, where we plot the image coordinates of the GT and the prediction from the selected ResNet8 trained model. The predictions were performed over several images from the validation dataset.

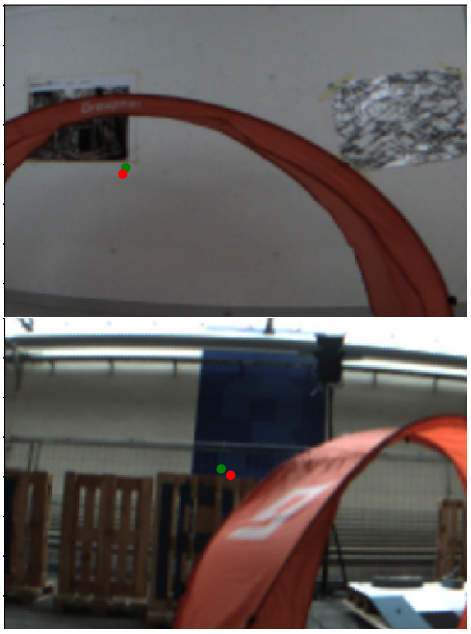


Figure 11: *Selected ResNet8 model prediction results (in red) and GT labels (in Green).*

In appendix 7.2 We address the additional networks that we trained, as mentioned in the *methods* section, both using the same optimizer, weight factor, while the training was performed for 300 epochs. First, a shallower ResNet model was trained. This model is kept with the same layers properties as the original architectures. Next we trained a variant, the TC-ResNet8 model, which was built with different layers properties and is not comparable in the morphology.

6. DISCUSSION AND CONCLUSION

Our vision-based deep drone racing method uses a CNN models to rapidly predict a desired waypoint in the image plane and relative velocity value straight out raw images. We investigated several models capabilities by their loss, rmse and fps and performed parameter tuning regarding the loss weight factor and the optimizer that has the best convergence-to-generalization power over the given simulative data. Our implemented approach shows that we can learn real-world trajectory insight from simulation inputs and perform in a

robust manner. In addition, we executed extensive experimental schemes, during of which we showed that adding only a few real-world data samples to the training process enhance capabilities with respect to the original from simulation to real-world zero-shot learning.

Future plans. In the original DroNet architecture, the authors had different output. Using the *steering angle* and the *collision* value they navigate their drone racer continuously without crashing into obstacles. These parameters can be considered as a future beneficial output to not only obtain coordinate and velocity predictions but also output a number regarding the collision probability of the drone to crash into a wall or any other object than the gate to be passed through. This must be factored with an objectiveness score, which will call whether the drone is heading toward an object or not. In the current implementation performed by the UZH group they only set a distance threshold, which is not robustly adapted during flight.

References

[1] Guang Zhong Yang et al. "The grand challenges of science robotics". English (US). In: *Science Robotics* 3.14 (Jan. 2018). ISSN: 2470-9476. DOI: 10.1126/scirobotics.aar7650.

[2] M. Schönheits et al. "IROS 2018 Fan Challenge - Team DLR Augsburg". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Oct. 2018, pp. 4158–4163. DOI: 10.1109/IROS.2018.8593792.

[3] Hyungpil Moon et al. "Challenges and implemented technologies used in autonomous drone racing". In: *Intelligent Service Robotics* 12.2 (Jan. 2019), pp. 137–148. DOI: 10.1007/s11370-018-00271-6. URL: http://dx.doi.org/10.1007/s11370-018-00271-6.

[4] S. Jung et al. "Perception, Guidance, and Navigation for Indoor Autonomous Drone Racing Using Deep Learning". In: *IEEE Robotics and Automation Letters* 3.3 (July 2018), pp. 2539–2544. ISSN: 2377-3774. DOI: 10.1109/LRA.2018.2808368.

[5] Antonio Loquercio et al. "Deep Drone Racing: From Simulation to Reality with Domain Randomization". In: *CoRR* abs/1905.09727 (2019). arXiv: 1905.09727. URL: http://arxiv.org/abs/1905.09727.

[6] Zichao Zhang and Davide Scaramuzza. "Rethinking Trajectory Evaluation for SLAM: a Probabilistic, Continuous-Time Approach". In: *CoRR* abs/1906.03996 (2019). URL: http://arxiv.org/abs/1906.03996.

[7] Adrian Carrio et al. "A Review of Deep Learning Methods and Applications for Unmanned Aerial Vehicles". In: *J. Sensors* 2017 (2017), 3296874:1–3296874:13. DOI: 10.1155/2017/3296874. URL: https://doi.org/10.1155/2017/3296874.

[8] Elia Kaufmann et al. "Deep Drone Racing: Learning Agile Flight in Dynamic Environments". In: *CoRR* abs/1806.08548 (2018). arXiv: 1806.08548. URL: http://arxiv.org/abs/1806.08548.

[9] Antonio Loquercio et al. "DroNet: Learning to Fly by Driving". In: *IEEE Robotics and Automation Letters* PP (Jan. 2018), pp. 1–1. DOI: 10.1109/LRA.2018.2795643.

[10] *Effective TensorFlow 2 : TensorFlow Core*. Software available from tensorflow.org. Jan. 2020. URL: https://www.tensorflow.org/guide/effective_tf2.

[11] D. Gandhi, L. Pinto, and A. Gupta. "Learning to fly by crashing". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Sept. 2017, pp. 3948–3955. DOI: 10.1109/IROS.2017.8206247.

[12] Nikolai Smolyanskiy and Mar Gonzalez-Franco. "Stereoscopic First Person View System for Drone Navigation". In: *Frontiers in Robotics and AI* 4 (2017), p. 11. ISSN: 2296-9144. DOI: 10.3389/frobt.2017.00011. URL: https://www.frontiersin.org/article/10.3389/frobt.2017.00011.

[13] Philipp Duernay Shuo Li and Erik van der Horst and, Christophe De Wagter, and Guido C. H. E. de Croon. "Visual Model-predictive Localization for Computationally Efficient Autonomous Racing of a 72-gram Drone". In: *CoRR* abs/1905.10110 (2019). arXiv: 1905.10110. URL: http://arxiv.org/abs/1905.10110.

[14] Bharathan Balaji et al. "DeepRacer: Educational Autonomous Racing Platform for Experimentation with Sim2Real Reinforcement Learning". In: *arXiv e-prints*, arXiv:1911.01562 (Nov. 2019), arXiv:1911.01562. arXiv: 1911.01562 [cs.LG].

[15] Ruoyu Sun. "Optimization for deep learning: theory and algorithms". In: *arXiv preprint arXiv:1912.08957* (2019).

[16] Seungwoo Choi et al. "Temporal Convolution for Real-time Keyword Spotting on Mobile Devices". In: *CoRR* abs/1904.03814 (2019). arXiv: 1904.03814. URL: http://arxiv.org/abs/1904.03814.

7. APPENDICES

7.1. Appendix 1: Code & Implementations

The supplementary code is available at the following GitHub repository Deep Learning TAU.

Our implementation was based on TensorFlow-2, where we tried to keep the code as elegant as possible while incorporating the advantages TensorFlow-2 has over TensorFlow-1. Although there is an automatic conversion tool supplied to easily convert the code it was not sufficient in our case due to the fact that it only convert from TensorFlow-v1 to TensorFlow-v2 compatible which kept the code bulky with the old TensorFlow-1 convention such as "session", "placeholder" etc. Overall our new implementation is much more cleaner and can easily be adopted to use with different networks architecture, optimizers etc. Moreover, once the code was transformed completely to TensorFlow-2 we could immediately export our models to TensorFlow-Lite and apply different optimization to quantize our model; In more details, TensorFlow Lite supports converting all model values, such as weights and activations, to 8-bit integers. We used 32-bit float variable and the conversion is significantly improve the results in terms of model size and performance. This technique statically quantizes all weights and activations during model conversion. In conclusion the Deep Learning part that was originally implemented by [5] is currently keep the overall logic but not only that it is completely changed its much shorter and efficient.

Moreover, our data loading scheme, where we divide our data to batches performance was twice as fast then the original implemented code this was achieved by incorporating vector operations instead for the original redundant use of for-loops.

Examination of different models is extremely reachable with our efficient implementation that can build and iterator that generates batches of data and feed it to the model in an automated and almost seamlessly way and does not require any considerable changes in order to initiate the end-to-end training process or a testing scheme of the saved model. Moreover, our scripts are compatible with automatic training and testing of various scenarios, for example when we wanted to check the influence of the type of optimizer on the model we could easily run the appropriate experiment from the "Train.py" script.

Regarding the activation of our code on top of the Jetson Tx2, we could easily do it be using the NVIDIA-SDK manager where one should install an enviroment which will be comptible with TensorFlow-2; We installed Ubuntu 18.04 with Jetpack suit 4.3 and examined both our TensorFlow-2 regular model and the TensorFlow-Lite model and indeed activate and created a simulation of the running drones (a sample git can be seen in our repository Deep Learning TAU); We could not run it with the Parrot drone due to the fact that currently the communication is done via Python-2 while the Jetpack suit 4.3 with TensorFlow-2 needes python-3.

7.2. Appendix 2: Additional Models

We had trained and tested two variant models of the original architecture 3 and both can be seen schematically in figures 16, 17. In the first variant we removed the third residual block from the original architecture and reduce the last fully-connected layer to a vector of size 32 instead of 256, which resulted in 5 times reduction in the amount of parameters. The additional network implemented is based on the work of Choi et al. [16]. This architecture uses temporal convolutions (TC) and different number of filters than in the original design, resulted in approximately 40 times less parameters.
We utilized the validation dataset for comparison of all three networks with respect to the loss, rmse and fps values.
Each network modification to the original architecture was performed for achieving reduced computational complexity. The *ResNet7* model contains less parameters, and takes on the same kind of input. Naturally, we tried to train it to perform close to the original *ResNet8*, intuitively done as a deeper network does not always suggests better results. On similar note, we built the *TC-ResNet8* model, which has resemblance in the number of layers used, but not like the original model, *TC-ResNet8* is much lighter. This is achieved thanks to smart deployment of smaller convolution layers, for which we were able to get a deep model architecture with less parameters used, hence should have had the similar computing capability without longer training. The results are shown on figures 12,13 and 14 and a comparison table regarding the running times and the number of parameters is available on table 3.

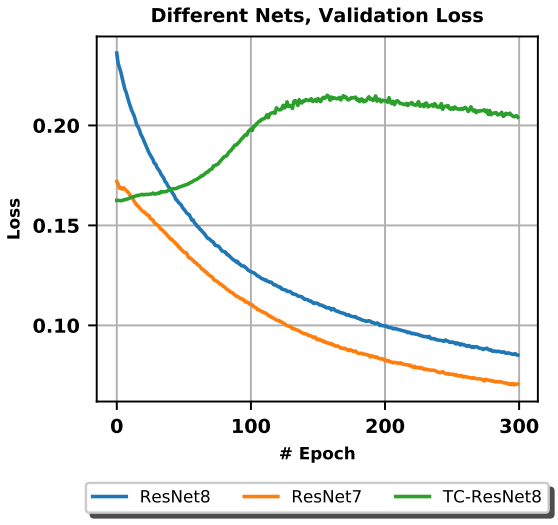


Figure 12: *Comparison of the loss value per epoch for ResNet8, ResNet7 and TC-ResNet8 models.*

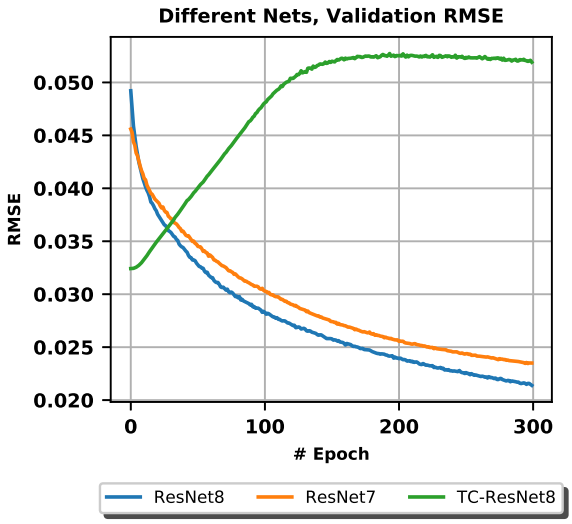


Figure 13: *Comparison of the RMSE value per epoch for ResNet8, ResNet7 and TC-ResNet8 models.*

We can see that the the original architecture convergence in a monotonic decreasing trend, while *TC-ResNet* explodes, and the *ResNet7* performs earlier and faster convergence as shown on the loss plot. The RMSE value per epoch is decreasing better in the original architecture, and outperforms the two variant models, however the *ResNet7* model shows similar values. In addition, even though the original architecture has also a stronger generalization power as seen in terms of RMSE value, the narrower architecture is a suitable alternative.

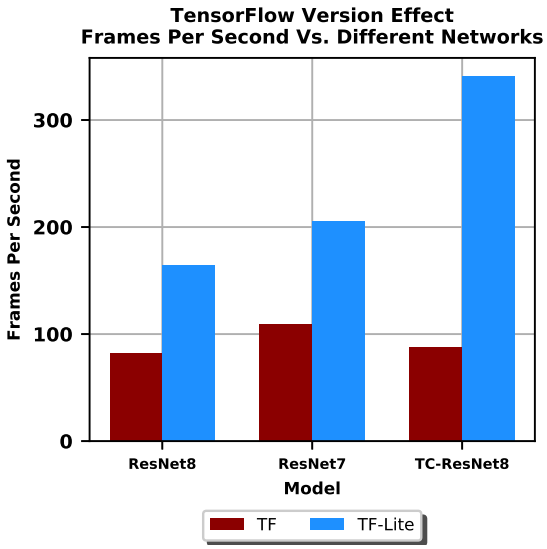


Figure 14: Running time bar chart comparison for ResNet8 with two of its variant architectures: ResNet7 and TC-ResNet.

Table 3: *FPS and number of parameters comparison table for ResNet8, ResNet7 and TC-ResNet8* ***TF*** *and* ***TF-Lite*** *models* with selected optimizer and weight factor

Optimizer	TF [fps]	TF-Lite [fps]	#Param [mil]
ResNet8	81.9	164.6	5.0
ResNet7	109.0	205.3	1.1
TC-ResNet8	87.9	341.0	0.1

Lastly, in terms of running time the original architecture has the lowest FPS rate, then the *ResNet7* and the *TC-ResNet8* models.
It is interesting to observe that the TF-Lite converted model for the *TC-ResNet8* is the fastest among all TF-Lite models, eventhough the *ResNet7* TF model was faster among all the TF models. This may be due to the fact that the temporal convolution model contained a fair amount of activation layers at the downstream, combined with the overall network structure that contained various shortcut convolutions of 1x1 that allow for straightforward quantization.
In addition, the following figure 15 emphasizes the regression model quality in approximate the GT coordinate.

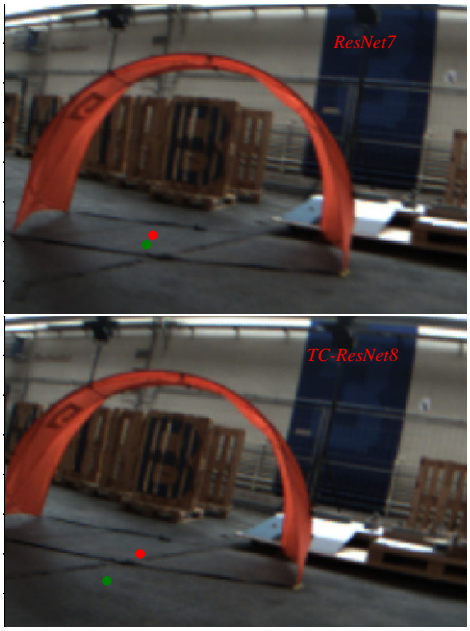


Figure 15: *Selected ResNet7 and TC-ResNet8 models prediction results (in red) and GT labels (in Green), respectively.*

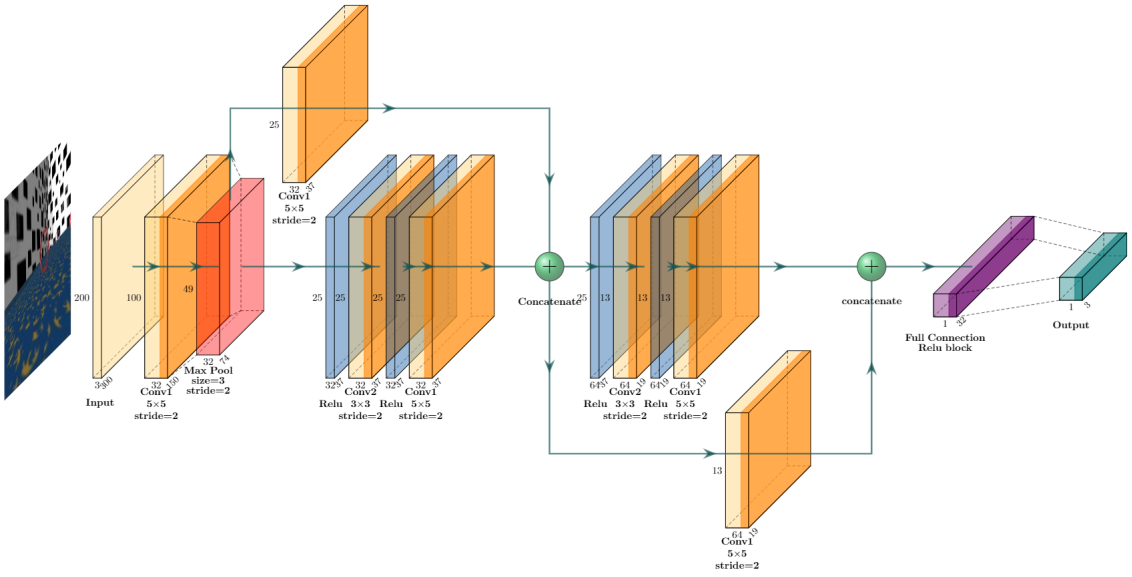


Figure 16: *ResNet7 architecture.*

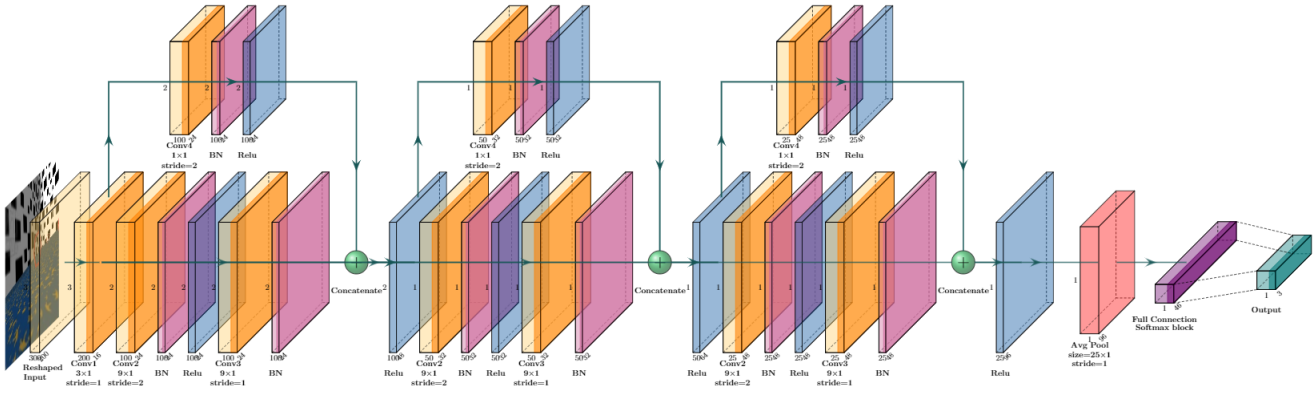


Figure 17: *TC-ResNet8 architecture.*