Machine Learning on Small UAVs

Alison Cleary
The Charles Stark Draper Laboratory, Inc.
Cambridge, MA
acleary@draper.com

Kristopher Yoo
The Charles Stark Draper Laboratory, Inc.
Cambridge, MA
kyoo@draper.com

Paul Samuel
The Charles Stark Draper Laboratory, Inc.
Reston, VA
psamuel@draper.com

Sean George
The Charles Stark Draper Laboratory, Inc.
Reston, VA
sgeorge@draper.com

Fei Sun
The Charles Stark Draper Laboratory, Inc.
Cambridge, MA
fsun@draper.com

Steven A. Israel
The Charles Stark Draper Laboratory, Inc.
Reston, VA
israel@draper.com

2020 IEEE Applied Imagery Pattern Recognition Workshop (AIPR) | 978-1-7281-8243-8/20/$31.00 ©2020 IEEE | DOI: 10.1109/AIPR50011.2020.9425090

Abstract **— Commonly, machine learning (ML) workflows for training and inferencing occur in resource rich environments. Draper Laboratory is pushing ML to the edge. This paper shows the concept of operations (CONOPs), design parameters, and constraints the team faced for edge implementation. The overarching requirement is to fully integrate the machine learning element into the small unmanned aerial vehicle (UAV) or drone. Given the limited payload capacity and power available on small UAVs, integration of computing resources sufficient to host both ML and Autonomy functions is a challenge. Past efforts have relied on an Intel NUC as the primary processing unit. However, recent advances in GPUs provide greater computational power at low-SWaP, compatibility with ML algorithms, and sufficient CPU resources to host the UAVs autonomy element. More recently developed processing units, designed specifically for ML applications at the edge, enable scaled down variants of the algorithms for integration onto significantly smaller platforms. In this paper, we identify a common software architecting strategy that enables a micro UAV (~ 150 grams) supported by a traditional CPU and a small UAV (3 kg) configured with a GPU. Draper's automation strategy leverages the open-source Robotic Operating System (ROS). The ML models were built using open-source Python Pytorch libraries. We provide the flight test results for a vehicle detection algorithm. Future applications will include visual navigation and tracking.**

Keywords — UAV, Machine Learning, On-board Processing, Workflow, Robotic Operating System (ROS)

I. Introduction

Machine learning models have been integrated on Unmanned Aerial Vehicles (UAVs) for more than 10 years. In recent years, Deep Learning Neural Networks (DLNNs) have been utilized for various autonomous UAV functions due to the technology's "excellent capabilities for learning high-level representations from raw sensor data" [1]. However for many applications, machine learning algorithm development requires training activities on large, specialized datasets [2] and getting these models to edge devices has been challenging.

The divergence of the model development environment from the onboard software environment can typically lead to an *ad-hoc* model-UAV integration workflow. In the resource rich development environment, training & inferencing techniques are tailored to optimize model performance. The effort to parse and filter machine learning model outputs is an added burden on the already limited platform processor and power. To have the machine learning element fully integrated into the onboard software of a small UAV, the model's training, inferencing, and data-processing activities must be designed with the edge device's limitations in mind. This work provides a robust, repeatable workflow design for machine learning model integration on small UAVs. The design enables use of various UAV sensor packages, standardized data formats, technology refreshed processors, and regularly updated target models. For this analysis, we tackled the problem of having the same workflow to load an offline trained DLNN model onto a CPU or a GPU based system.

II. ML-UAV Design Space

A. Proof-of-Concept UAV Platform & Processor

A custom platform, built with COTS (Consumer-Off-The-Shelf) parts, served as the Machine Learning and Autonomy Flight Software integration testbed. The UAV is equipped with an EO camera, GPS, flight controller, and platform processor (*Fig. 1*). The role of the onboard UAV processor for this design is to host both flight software and machine learning algorithms capable of running on CPU or GPU.

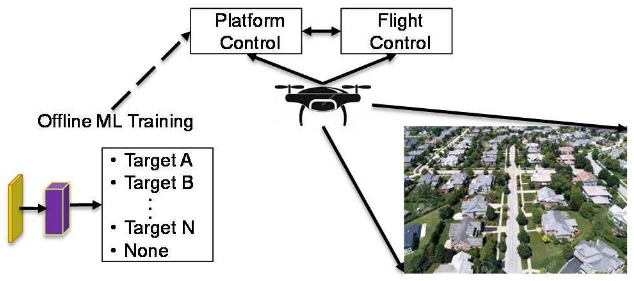


Fig. 1. System Workflow

Throughout the ML workflow development phase, the onboard UAV platform processor underwent multiple iterations to maintain a competitive computing advantage over similar low-SWaP and low cost candidates on the market. The processor refresh required upgrading the current onboard processor from an Intel NUC8i7BEH to an NVIDA Jetson Xavier NX.

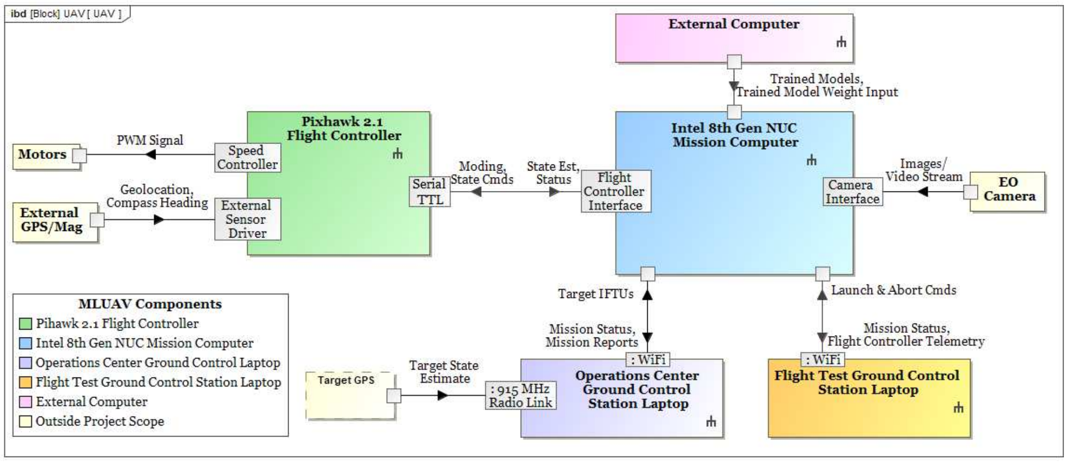

ibd [Block] UAV [UAV]
External Computer
Trained Models, Trained Model Weight Input
Pixhawk 2.1 Flight Controller
Motors
PWM Signal
Speed Controller
External GPS/Mag
Geolocation, Compass Heading
External Sensor Driver
Serial TTL
Moding, State Cmds
State Est, Status
Flight Controller Interface
Intel 8th Gen NUC Mission Computer
Camera Interface
Images/ Video Stream
EO Camera
Target IFTUs
Launch & Abort Cmds
Mission Status, Mission Reports
Mission Status, Flight Controller Telemetry
: WiFi
: WiFi
MLUAV Components
Pihawk 2.1 Flight Controller
Intel 8th Gen NUC Mission Computer
Operations Center Ground Control Laptop
Flight Test Ground Control Station Laptop
External Computer
Outside Project Scope
Target GPS
Target State Estimate
: 915 MHz Radio Link
Operations Center Ground Control Station Laptop
Flight Test Ground Control Station Laptop

Fig. 2. MLUAV Flight Software Block Diagram

An NVIDIA Jetson AGX Xavier developer kit, emulating an NVIDA Jetson Xavier NX, supported the standardization of software installation procedures and was implemented as a development processor. Leveraging free and open-source (FOS) Python-based machine learning software allowed the proposed workflow design to grow with the addition of new models, software updates, and bug fixes released by the larger machine learning communities. When paired with an open source ROS workspace, the FOS software packages need to be integrated into the workflow design to track installation procedures and updates.

The Robotic Operating System provided an interface for onboard sensors, algorithms, and flight controller to communicate. Draper autonomy software, hosted on the mission computer, offers an interface for mission scheduling as well as the collection of ML model outputs. All Domain Planning and Execution Technology (ADEPT), developed at Draper, is the foundation for Draper's autonomy framework. The framework employs a multi-level, hierarchical decomposition approach to problem solving [3]. *Fig. 2* presents the workflow of a flight tested autonomy and flight software architecture *without* an integrated machine learning component. The goal of this work is to provide a well-defined, extensible interface between the ML component and the flight-tested software architecture.

III. ML Framework

The focus of the machine learning effort was to create a target detection model to demonstrate functionality interfacing with a common Draper UAV flight software. The codebase to perform detections was written in a flexible and extensible way such that the processes and source code could be applied to other UAV systems within Draper's portfolio. The developed process gives the option for splitting training and inferencing steps into processor agnostic steps– i.e. provide the option to train and inference on the same processor or on disparate processors, depending on the computing resources available. Consideration was given to the fact that UAV processors may or may not include GPU compute power.

A. Integration with Free and Open Source Tools

To create an ML based target detection capability, a software stack capable of integrating with packages enabling neural network and deep learning computations was chosen. A preference was given toward utilizing FOS programming languages and tools because of their online community support, thorough documentation, and accessibility. Python was selected as the top-level language due to the extensive number of state-of-the-art (SOTA) machine learning packages and resources, namely Google's TensorFlow and Facebook's PyTorch, available with Python APIs. The Rospy API, discussed in Section IV, allowed for easy integration with the ROS based flight software. For the MLUAV software stack, a Python – PyTorch approach was chosen for its lack of additional overheard required in creating a prototype over other packages. Utilizing PyTorch also gives the option to leverage Detectron2, an advanced object detection API that provides an easy framework for performing object detection tasks, a typically difficult integration task. The object detection task was thus chosen for the workflow's proof-of-concept ML model application.

Detectron2 is Facebook's next generation software system for implementing object detection algorithms [1]. With Detectron2, a number of common pre-trained backbones are provided in their model zoo such as Faster R-CNN (Region Based Convolutional Neural Network), RetinaNet, and RPN (Regional Proposal Network). The strength of Detectron2 lies in its ability to provide a quick method for using canned algorithms, retraining a training model from scratch, or applying transfer learning for a specific application.

B. ML Detection Model Training

For the MLUAV effort, transfer learning was applied to a Faster R-CNN backbone with images taken from the UAV's camera positioned on top of a tall structure to capture street level traffic. Training and test data were manually labeled using LabelImg to indicate the location of objects of interest within the image. Additionally, the data were augmented using standard image transformation techniques such as rotation, flip, and mirroring to increase the number of images and views of the objects. A training process and codebase was developed and documented to make this process repeatable and extensible to other ROS based UAV applications. The focus of codebase development was to define a process which gives flexibility to how models are trained and deployed. For example, in situations in which there are sufficient computing resources, the training pipeline could be deployed directly on the candidate processor. This can be completed whether the candidate processor has GPU or just CPU availability. Conversely, the training pipeline could be deployed remotely, such as on a development laptop, and then the model weights and metadata transferred to the candidate processor.

Regardless of the specifics of the training – i.e the processor on which the training was performed, whether training was applied from scratch or whether transfer training was applied – the deployment of the model to perform inferences has been standardized through the creation of *MLDetectionClass*, as shown in *Fig. 3.*

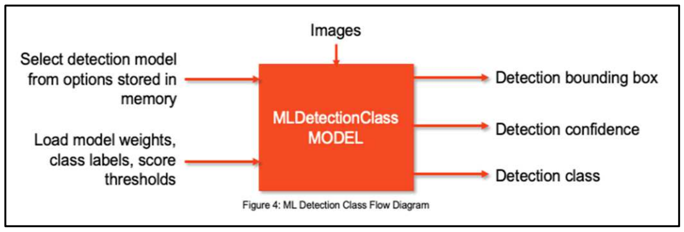
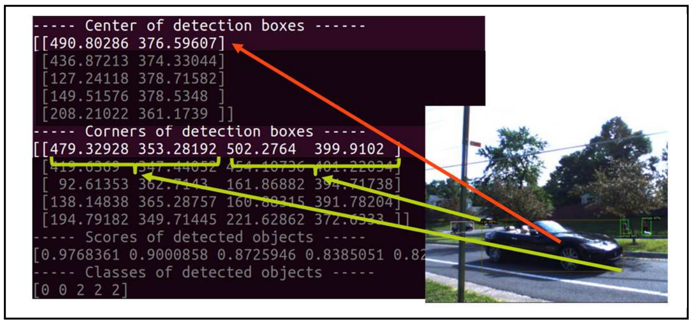

Images
Select detection model from options stored in memory
Load model weights, class labels, score thresholds
MLDetectionClass MODEL
Detection bounding box
Detection confidence
Detection class
Figure 4: ML Detection Class Flow Diagram

Fig. 3. ML Detection Model

C. *ML Detection Model Architecture*

Though this specific CONOP pursued an object detection capability, the *MLDetectionClass* structure (*Fig. 5*) is inherently extensible to other missions or use cases and therefore serves well as a template. The behavior of the module changes based on the specific algorithm and data expected, but the underlying framework remains the same. The model weights and metadata for new use cases are loaded into the *MLDetectionClass*, appropriate data are ingested, and the corresponding inferencing/processing is performed. The necessary outputs are communicated to the ROS API and sent to the required parts of the UAV and/or ground station.

The *MLDetectionClass* lives onboard the candidate processor on the UAV. There may be one or more models stored within the processor memory that could correspond to different missions, flight modes, etc. The desired detection model is selected from those available in memory. The corresponding model weights, class labels, and score thresholds are loaded to instantiate the *MLDetectionClass*. As will be discussed further in Section IV on ROS System Integration, images are passed to the *MLDetectionClass* as standard Python NumPy arrays. This ensures that the format of information being passed to the ML detection model is agnostic of the specific sensors that may be onboard the UAV. Inferencing is performed on the ingested images. For the MLUAV target detection task, detection bounding box pixel locations, detection confidences, and detection classes are outputted for each of the objects detected within each image. This information is presented in a standard form and outputted to the ground station and/or flight controller onboard the UAV.



```
----- Center of detection boxes ------
[[490.80286 376.59607]
 [436.87213 374.33044]
 [127.24118 378.71582]
 [149.51576 378.5348 ]
 [208.21022 361.1739 ]]
----- Corners of detection boxes -----
[[479.32928 353.28192 502.2764  399.9102 ]
 [419.6369  347.44052 454.10736 401.22034]
 [ 92.61353 362.7143  161.86882 394.71738]
 [138.14838 365.28757 160.88315 391.78204]
 [194.79182 349.71445 221.62862 372.6333 ]]
----- Scores of detected objects -----
[0.9768361 0.9000858 0.8725946 0.8385051 0.82
----- Classes of detected objects -----
[0 0 2 2 2]
```

IV. ROS System Integration

The overall system integration goals of the workflow design problem can be broken down into three key categories:

1) Train and Save models: All machine learning model metadata (weights, names, etc.) should be maintained in memory to be loaded for specific applications and CONOPs.

2) Deploy flight software and ML algorithm: On the same processor, all autonomy, guidance, and necessary flight functions should be able to run with the additional load of the machine learning algorithm.

3) Relay detections to a GCS (Ground Control Station) Operator: To verify the workflow design, a flight test should confirm that the lab-trained machine learning model is able to output inferencing metadata real-time in a useable format for the rest of the flight system.

Due to distributed resources and limited available flight time, initial proof-of-concept ML models were trained in the lab and moved to the UAV.

A. *Development Process*

The goals of creating an integration pipeline are to standardize software installation processes, standardize model inputs, and aggregate inferencing outputs. To quickly integrate the ML framework with an existing ROS-based flight software framework, there are a few constraints and required interfaces that can shape the development process.

- Python versions, drivers, and packages necessary to run the ML model must be compatible with any existing package versions defined as a workspace dependency.
- Python versions, drivers, and packages must be compatible with the candidate processor, operating system, and both CPU and GPU computing.
- The Rospy client API must be used to interface with Python models, as implementation speed is favored to potentially better runtime performance.

In this proof-of-concept architecture, a ROS Melodic workspace, built with the default Python 2.7 target environment, was configured to work with Python 3.6 ROS nodes and deployed on the UAV's processor. With the release of newer ROS distributions and increasing support for Python 3, this process can be expected to evolve and update.

To standardize model input and inferencing output formats, both open source and custom ROS message packages were used to clearly define expected data types for both the ML model and flight software messages. Standard ROS message conversions provide modularity and flexibility to different CONOPS, requirements, and ML models. For a detection model, a camera driver is required to conform outputs to a common ROS sensor message package for image data (*sensor_msgs/Image*). Accounting for both varying encoding and image size, the image messages were then converted to a standard NumPy array representation for use with the MLDetector model. A custom ROS message to support varying detected object metadata (*Fig. 4*) was used to communicate object pixel location, classification, and confidence with the onboard autonomy and guidance software that runs in parallel with the ML algorithm.

Fig. 4. ML Detection Metadata

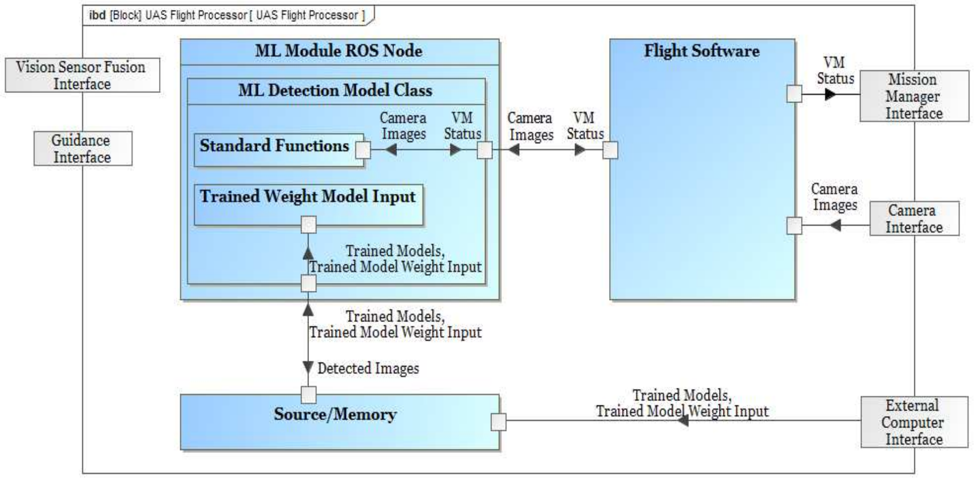

ibd [Block] UAS Flight Processor [UAS Flight Processor]
Vision Sensor Fusion Interface
Guidance Interface
ML Module ROS Node
ML Detection Model Class
Standard Functions
Trained Weight Model Input
Camera Images
VM Status
Camera Images
VM Status
Trained Models, Trained Model Weight Input
Trained Models, Trained Model Weight Input
Detected Images
Source/Memory
Flight Software
VM Status
Mission Manager Interface
Camera Images
Camera Interface
Trained Models, Trained Model Weight Input
External Computer Interface

Fig. 5. ML ROS Node Block Diagram

B. *ML Module ROS Node Architecture*

To call inferencing functions from an ML detector model and relay the resulting detections to the rest of the components of the flight system, a ROS wrapper was structured as both a "talker" and "listener" node (*Fig. 5).* Within the node, an instance of the MLDetector model is created and serves as an interface to the trained ML model's inputs and outputs. By calling ML model functions from an abstracted, higher level, the ROS node is able to provide a transparent, extensible framework for interfacing with a ML detector algorithm:

- Create an instance of the MLDetector Class detector model in the ML ROS node.
- Call inferencing, image-saving, and detection metadata functions as needed for flight application.
- Save detection images to memory or display through ROS image messages.

To deploy a new type of ML model, a developer would need to define all necessary ROS input/output messages, create a python class using the MLDetector Class template, and modify the ML Module ROS wrapper functions (as needed) to transform inputs and/or outputs into the right format for the new model's functions.

C. *System Integration and Test*

The feasibility and repeatability of the workflow design was evaluated and verified with detection data from an EO camera both on the ground and in the air. The proof-of-concept UAV platform was retrofitted with the candidate processor, a NVIDIA NX Development Kit, and other necessary mechanical interfaces to mount the processor. The aircraft power system was re-designed to support the new processor, and the flight controller firmware and camera drivers were updated to match current stable releases.

Flight test goals were simplified to quick, basic camera data collections and flight checkouts in order to give the ML team the chance to perform iterations on both models and software after receiving a flight test dataset. Data were collected on different days and with different lighting conditions to account for variability in training images. For both ground and air tests, the ML module facilitated object detection, returned detection images as ROS messages for a GCS operator to view, and saved images with detections to onboard memory.

In initial ground tests, the EO camera was positioned to capture pedestrians and cars in a residential area. The pre-trained detector model reported a large number of both false positives and false negatives like those shown in *Fig. 6.* To tailor the detector to this proof-of-concept application, and to exercise the iterative nature of the proposed ML-UAV workflow, the model was retrained offline with transfer learning. Additional UAV collected images were included in the training image dataset. The newly trained model was then copied over to the UAV's onboard processor and to be ready for use.

For final ground tests and a first flight test, the detector model was able to more accurately and frequently detect the target objects in images.

V. Outcomes

This work presented a workflow design and architecture comprised of Python modules and an associated ROS wrapper package that can be easily integrated into an existing ROS workspace. This design reduces development time, allows for faster and more iterations on model training, and deploys easily into an existing UAV flight software. The workflow can be easily adapted to other ROS-based systems, and the training pipeline can be modified for other projects outside of unmanned aerial vehicles.

Proposed future work on the ML framework involves model investigation, model tradeoff analyses, and additional training image data collection. Although a type of CNN was used for this application, the framework can be expanded to support varying families of detector models. Within individual families of models, trades between network size and performance can be used to characterize models that are optimized to run within an allowable processing budget. The addition of more application-specific training images (overhead, airborne, etc.) can help reduce processing load and errors in accuracy.

Proposed future work on the overall System Integration includes leveraging GPU cores, optimizing CPU and GPU communication, and performing processing load tradeoff analyses. For this application, flight software and machine learning software were both running on the CPU; future applications plan to migrate the ML model and framework to the GPU. Using data collected in flight tests, a tradeoff analysis between the processor load available to the detector model and the camera sampling rate can be done to assess overall mission performance.

Acknowledgment

The team would like to thank Draper's System Engineering Division for funding this research. The funding enabled Draper to build a common workflow to incorporate ML models across multiple UAV programs.

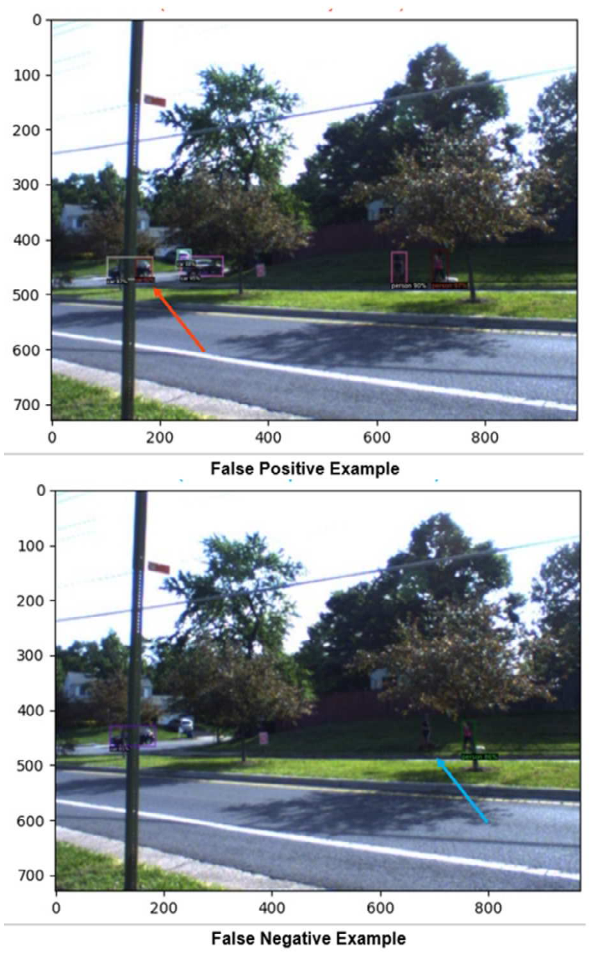


Fig. 5 MLUAV Sample Output

References

[1] A. Carrio, C. Sampedro, A. Rodriguez-Ramos and P. Campoy, "A Review of Deep Learning Methods and Applications for Unmanned Aerial Vehicles," *Journal of Sensors,* vol. 2017, no. 3296874 , 2017.

[2] P. Zhu, L. Wen, X. Bian, H. Ling and Q. Hu, "Vision meets drones: A challenge," *arXiv: 1804.07437,* p. 11, 2018.

[3] M. Ricard and S. Kolitz, "The ADEPT framework for intelligent autonomy," The Charles Stark Draper Laboratory, Inc. Cambridge, MA, 2013.

[4] PyTorch, 09 10 2020. [Online]. Available: https://pytorch.org/.