Embedded Machine Learning

Autonomous Vehicle with NVIDIA Jetson Nano

Damien Dam

Enzo Durand

April 27, 2022

0. Introduction

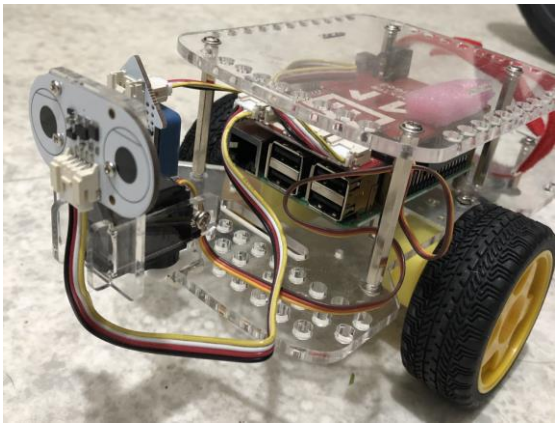
During our studies in Machine Learning (ML) and Computer Sciences, we have been exposed to mid-level to powerful machines capable of running highly complex tasks that require a considerable amount of computational power. We have been able to explore endless possibilities of convolutional neural network (CNN) architectures whose jobs range from simple digit classification to live object detection in real time.

However, in this project, we place ourselves in the shoes of autonomous vehicle engineers whose goal is to teach a resource-limited onboard computer to drive itself without much human intervention. This is an ambitious objective so we do not aim to build a Tesla by the end of the semester. Instead, we took the time to benchmark and explore the provided hardware in order to find the most suitable approach to tackle the tasks while staying within the constraints of the embedded system. Specifically, for this project, we are given the following hardware:

- A NVIDIA Jetson Nano,
- A Raspberry Pi,
- A GoPiGo by Dexter Industries,
- Batteries, wifi adapter, and other accessories.

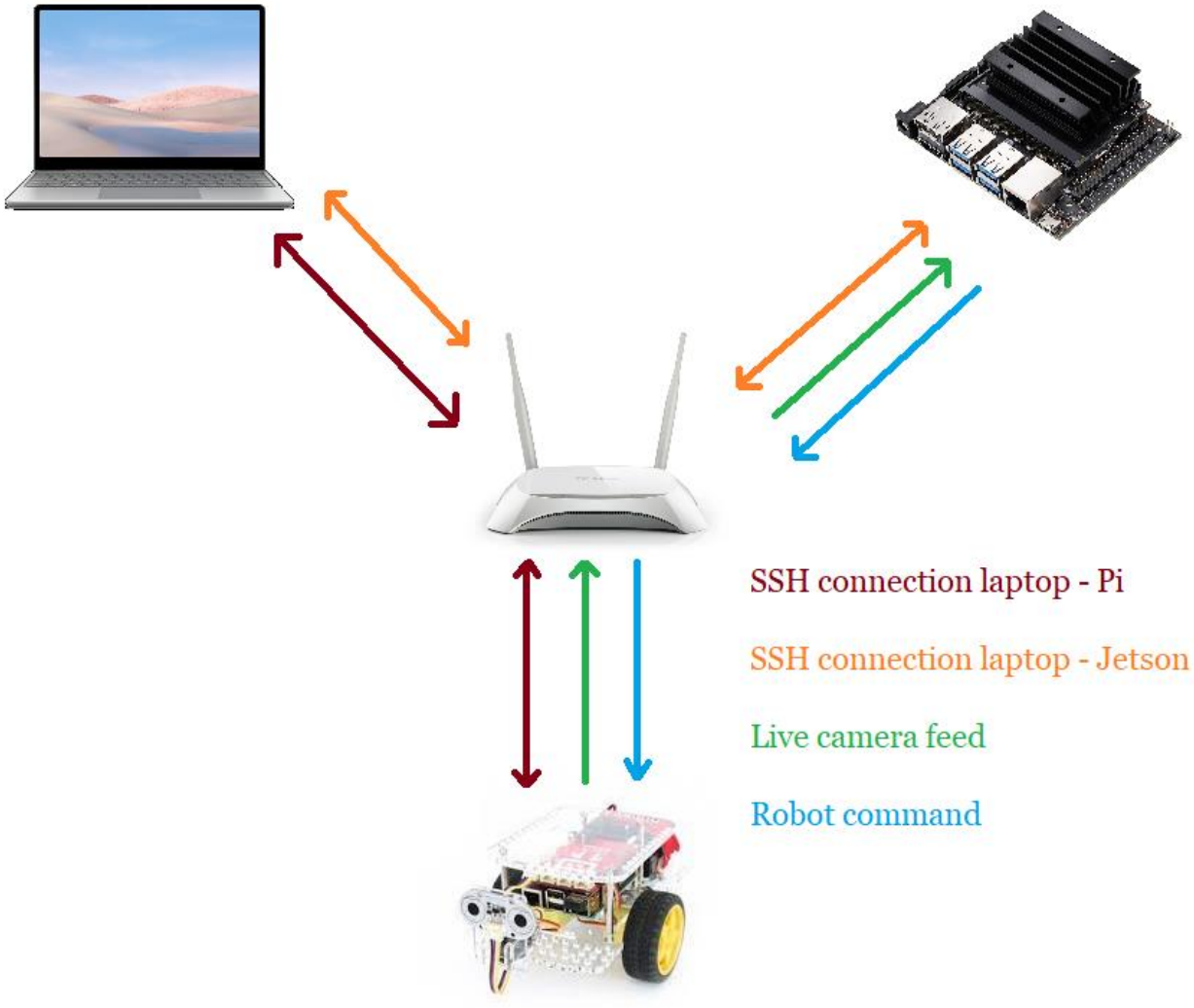
The Jetson Nano, being the center of the project, acts as the brain where one or a few trained ML models run. It consists of a 128-core graphical processing unit (GPU), a quad-core processor (CPU) and 4GB of memory (RAM), plus a Ubuntu environment for development. Inside the Jetson we put the model binaries and a Python environment where frameworks and libraries (such as PyTorch, OpenCV) are pre-installed to support model execution as well as external communication (as explained in later part).

The Raspberry Pi and the GoPiGo are attached and assembled together as shown in the picture below.



These two pieces act as the body which physically drives around, transmits live images from the front camera to the Jetson Nano (the brain), then receives and executes commands from the Jetson Nano (drive straight, reverse, turn, stop, etc.).

Since the brain and the body are physically apart, communications must be done over a wireless local network to which both pieces are connected. To facilitate control of both devices, we use a computer with a physical screen and a keyboard (can be a laptop) and establish an SSH connection to each one in order to access their command line control. On the Jetson Nano, we launch a server-side Python script that we had programmed which waits for a TCP connection from the Pi. On the Pi, we then launch the client-side script which connects to the server's address and starts transmitting live camera feed. Once the Pi is connected, the Jetson Nano must analyze each received image to decide whether it must reply with a specific command. If it is the case, a textual response is sent back to the Pi to be executed. The illustration below shows roughly the described infrastructure.



With the help of GoPiGo's official documentation and some online tutorials, it was not hard for us to program the client-side script that transmits live images, receives text commands and parses the text into the robot's movements. The real difficulties lie in the construction, training, and testing the CNN that would go into the Jetson Nano.

First, we must define a feasible task. We considered the following choices:

- Follow-me. In this task, the robot must recognize its owner and follow him/her around if detected;
- Follow-it. This task is similar to the one above, but instead of a person, the robot follows a specific object like a tennis ball;
- Controlled-by-hand. As a traffic controller, a person can stand in front of the robot and issue various commands using his/her hand.

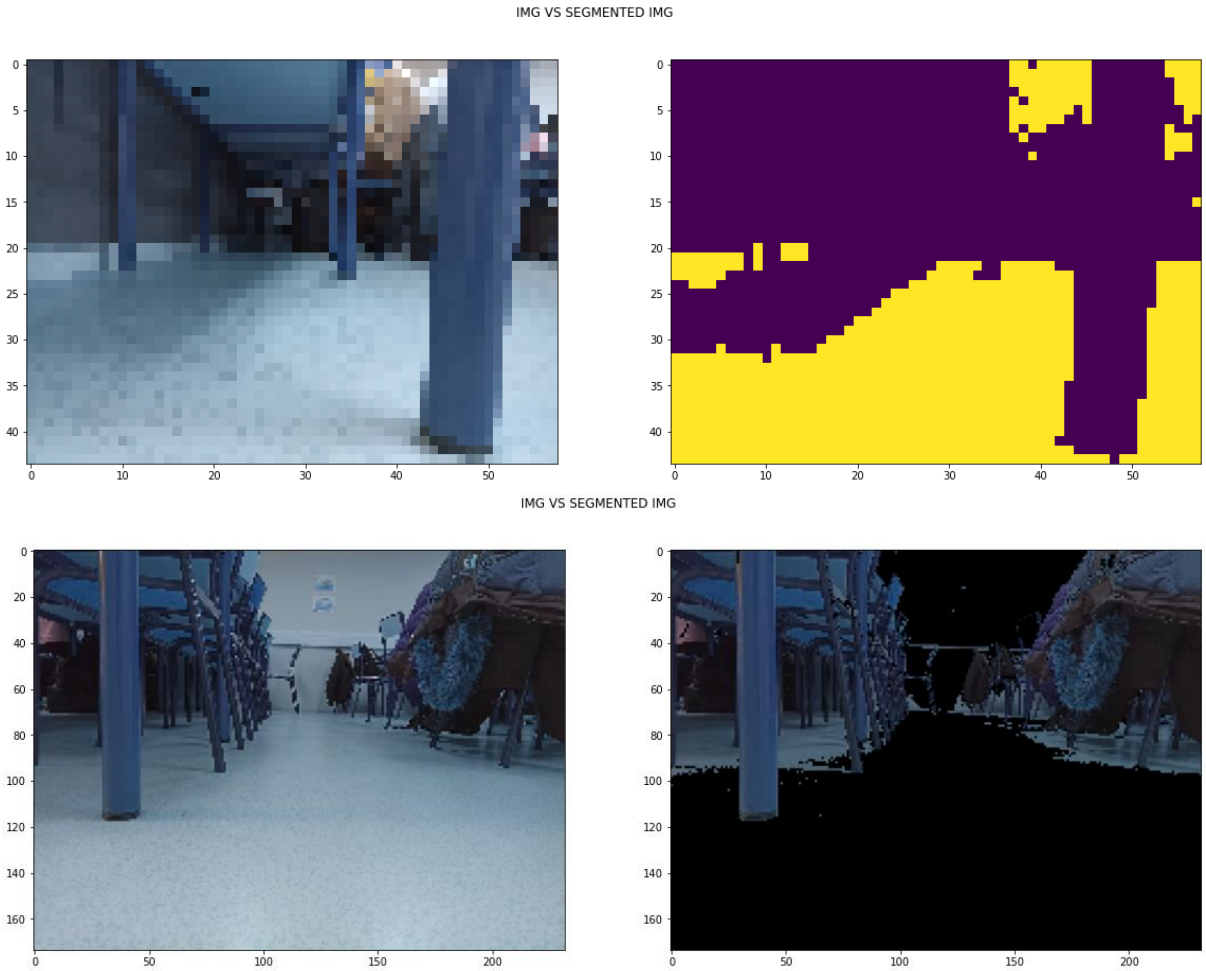
We chose to work on the last task as it is moderately challenging among the choices.

In the later parts of this report, we are going in depth on how we proceeded with the task in hand, the problems that we faced, as well as our reflections along the way to finding solutions to tackle these obstacles.

1. Researching

1.1. Image processing

In order to start our project with appropriate tools, we did some research on several subjects. Before thinking about ML techniques, we first looked into image processing. As a matter of fact, building a deep learning (DL) pipeline is useless if classical approaches can solve the problem. We looked at image recognition and segmentation techniques such as K-means, OSU, etc...



The problem with these techniques is that they are dependent on some parameters and therefore generalization is not great. Indeed, certain values of k in the K-means algorithm would not work in lots of situations. After a few tests, we decided to approach the problem using ML instead.

1.2. CNN and PyTorch

First thing first, we needed to understand convolutional neural networks since we are working on computer vision. We found what we needed in books such as

Quand la machine apprend (When machine learns) by Yann LeCun or *Deep Learning* by Ian Goodfellow, Yoshua Bengio and Aaron Courville. For implementation with code, we use PyTorch for its robustness and ease in building and experimenting different neural network architectures, so we obviously looked into its documentation to understand how we were going to proceed.

1.3. Transfer learning

Aware of CNN and PyTorch, we needed to train our models for our different tasks. In order to speed up the process, we decided to take advantage of the transfer learning technique where one would continue to train a pre-trained general model to fit his needs, instead of building it from scratch which would take too much time and resources. The PyTorch documentation helped us a lot to understand how transfer learning works, and their *model zoo* had the materials that we needed.

1.4. Beginning the project

We understood that fine-tuning hyperparameters and regularization terms is done by experiments but there are rules that we had to follow in order to build our pipeline without wasting too much time. We decided to follow these recipes for training neural networks written by Andrej Karpathy which gives a simple set of instructions to avoid bugs and save time.

2. Benchmarking

2.1. Pipeline

Before building a pipeline to experiment with the models for our tasks, we had to know which type of task that could work on the Jetson Nano.

Using CIFAR10 and CALTECH101 as datasets, we experimented inference time of some popular pre-trained models available online such as AlexNet, ResNet, YOLO, etc. Each model performs a specific task including classification (AlexNet, VGG, Resnet...) or segmentation (FCNResNet50). For each model, we wrote a Python script that imports necessary libraries, loads the model and the dataset, passes the images in the dataset to the model, and computes the average time taken to produce the output.

Launching the scripts and obtaining the results from our personal computer were straightforward. However, doing the same thing on the Jetson Nano proved to be extremely challenging! During benchmarking, the machine's GPU must run at full power while at the same time, its CPU and RAM must also work hard to process and store any output produced by the GPU. The Jetson Nano, being an embedded machine, had a hard time performing as well as a normal computer. Most of the time, when launching a script, it returned the infamous *Segmentation Fault* message indicating a memory-related error. Other times, the machine froze and must be hard-rebooted.

After many failed attempts, we realized that powering the Jetson Nano using the micro-USB port tended to underdeliver the power needed for heavy-lifting tasks. A quick search on NVIDIA's website showed that in order to maintain the current's stability, the Jetson Nano must be powered using the DC jack. Indeed, after switching to DC power, the machine was much more stable and we obtained the benchmarking results in the table below.

CLASSIFICATION

AlexNet	Computer	Jetson
Time per image	0,006	0,032
FPS possible	166,6666667	31,25
TPI using ONNX	0,0033	0,024
FPS using ONNX	303,030303	41,66666667

CLASSIFICATION

VGG	Computer	Jetson
Time per image	0,143	0,796
FPS possible	6,993006993	1,256281407
TPI using ONNX	0,108	0,551
FPS using ONNX	9,259259259	1,814882033

CLASSIFICATION

ResNet	Computer	Jetson
Time per image	0,02183	0,13
FPS possible	45,80852038	7,692307692
TPI using ONNX	0,012	0,059
FPS using ONNX	83,33333333	16,94915254

SEGMENTATION

FCNResNet50	Computer	Jetson
Time per image	0,49	1,836
FPS possible	2,040816327	0,5446623094
TPI using ONNX	0,17	0,887
FPS using ONNX	5,882352941	1,127395716

CLASSIFICATION + BOUDING BOX

YOLOV5s	Computer	Jetson
Time per image	0,0192	0,098
FPS possible	52,08333333	10,20408163
TPI using ONNX	0,0109	0,068
FPS using ONNX	91,74311927	14,70588235

NETWORKS SPECS

Specs	AlexNet	VGG16	ResNet	FCNResNet50	YOLOv5s
input size	224	224	224	224	224
n parameters	61M	138M	23M	23M	7M

We see that the inference time depends on the number of parameters of the model and on the problem being solved. For instance, the segmentation task requires lots of computation which leads to a low inference FPS, while we need to have a certain number of frames per seconds to be functional. On the other hand, in a classification problem, the Jetson Nano would use several sequential images to decide which action to take by doing a hard vote.

2.2. Doubling the inference performance

Previously, we used the classical *.pth* and .h5 format to save and load our models for inference, but we found out that the *.onnx* format speeds up the computation by a factor of two. Thus, benchmarking was done using *.onnx* format and after testing it on our personal GPU, we produced several scripts to benchmark the performance of the Jetson Nano with different tasks and architectures.

2.3. Choosing the right model

While inference time is one of the major factors, we also had to take into account the limitation of the Jetson, specifically its 4GB of shared memory between the RAM and the GPU against some very good hardware that we personally own: the RTX 2060 with 6GB of memory and the RTX 3090 with 24GB. Although this is suitable for most simple tasks, driving an autonomous vehicle is not one of them.

After a few rounds of benchmarking, we chose the VGG16 model as it was the most appropriate compromise between the capability and the number of hyperparameters. This model was going to be used for transfer learning afterwards.

3. The experiments

3.1. Collecting data

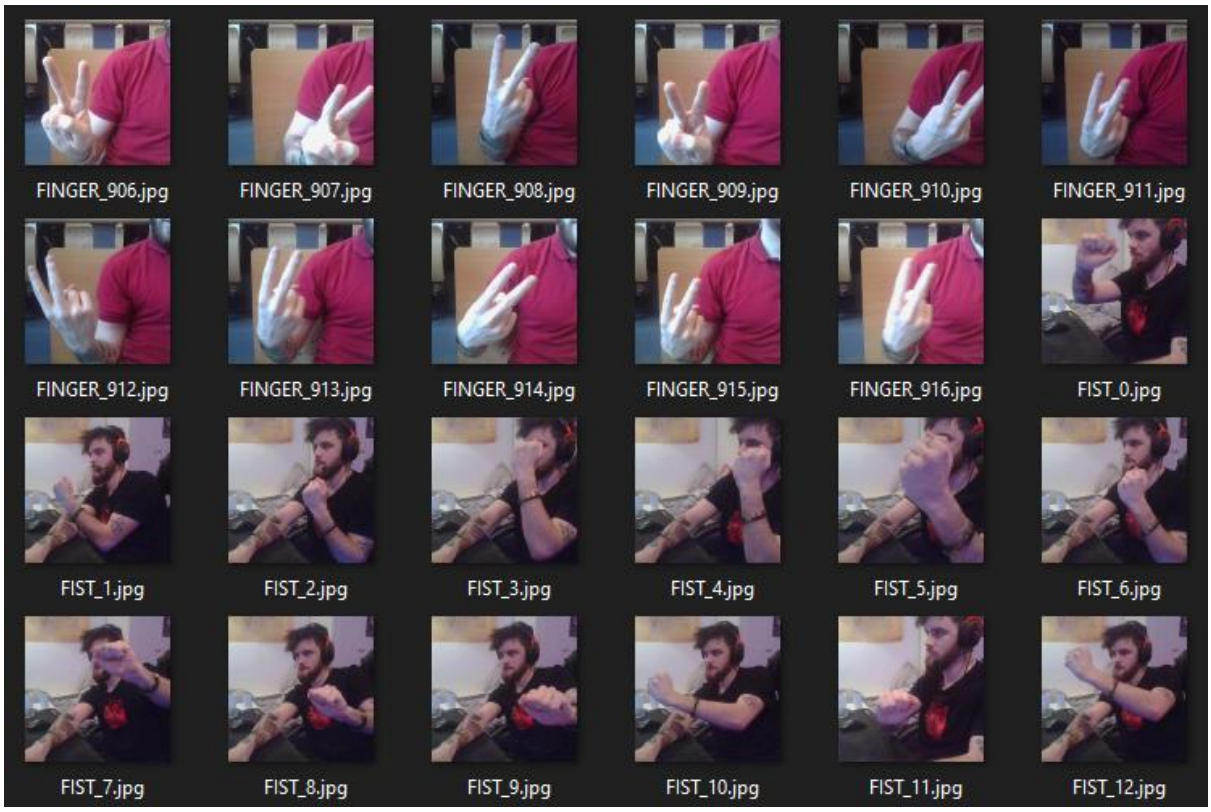
After defining the task *"Controlled-by-hand"* where posing our hand in front of the robot controls its movements, we considered two possible solutions:

1. Building a one-stage model that directly detects and classifies the hand gesture from the whole image as input whose dimensions are 224 by 224 pixels with 3 color channels. This model outputs the probability for each label where the most probable label is chosen as the model's prediction;
2. Building a two-stage model whose output is identical to the one above but differs in the inner structure. To be more precise, the model consists of two small CNNs: the first detects and outputs the position of the hand within a 400 by 400 by 3 image, which would then be cropped, resized to 100 by 100 by 3, and passed into the second CNN that classifies the gesture.

Since this kind of image is not widely available online for model training, we must create them ourselves!

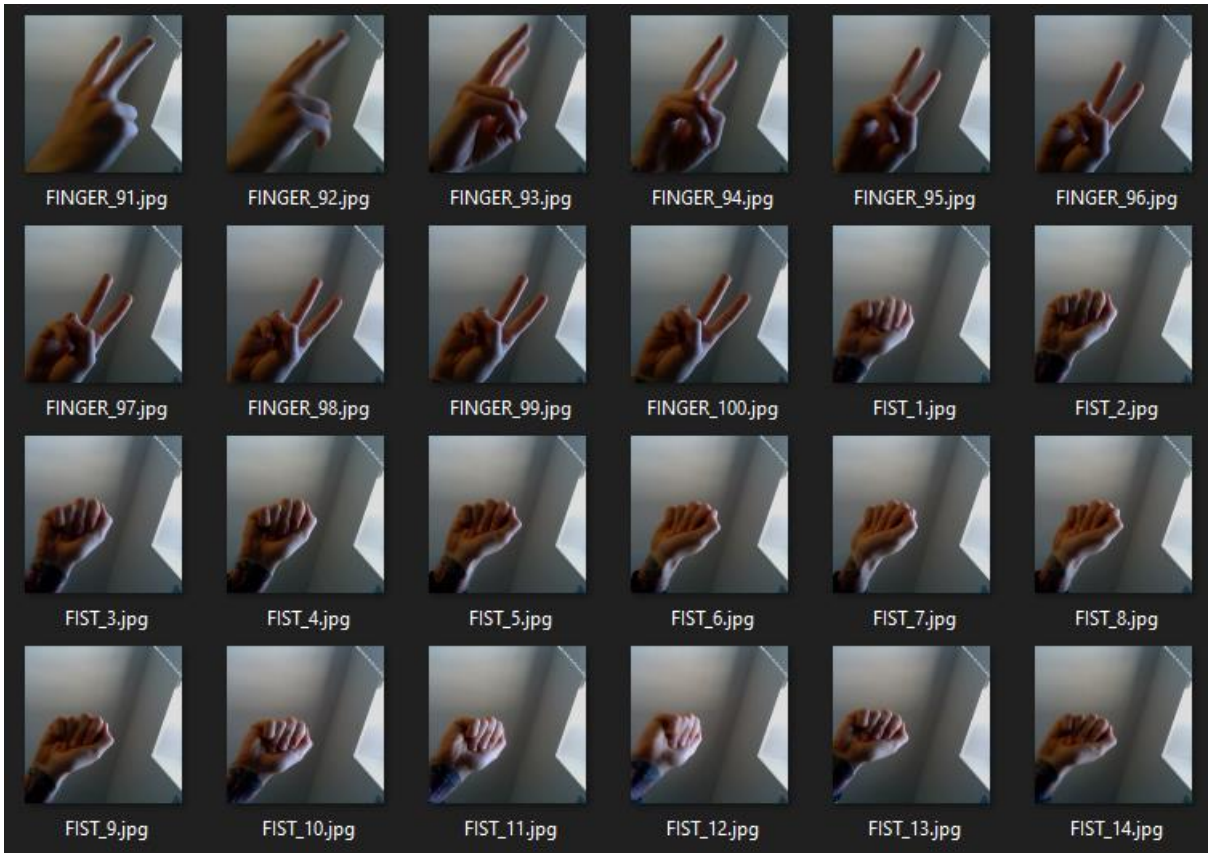
To explore each solution, we programmed a few scripts producing three different sets of data for training whose labels are specified in the file names. Besides using the scripts to create the images ourselves, we also called for help from our classmates who spent a few minutes taking their photos as well which greatly diversified the data that we had. The result was nearly 10,000 original images ready to be used for model training. They were divided into three sets:

1. The first set includes 4903 training and 1994 testing images corresponding to the first model mentioned above (one-stage classifier);



2. The second set consists of 2144 training and 602 testing images which are similar to those above but contain in their names four integers representing the coordinates of a square in the image surrounding the hand (also known as *bounding box*). This set is used to train the first CNN of the two-stage classifier;

3. The last set contains images cropped from the one above as well as images from other sources showing only the hand. They are consumed by the second CNN of the two-stage classifier. Their names contain the correct gesture: *Finger*, *Palm*, *Fist*, *Left* or *Right*.

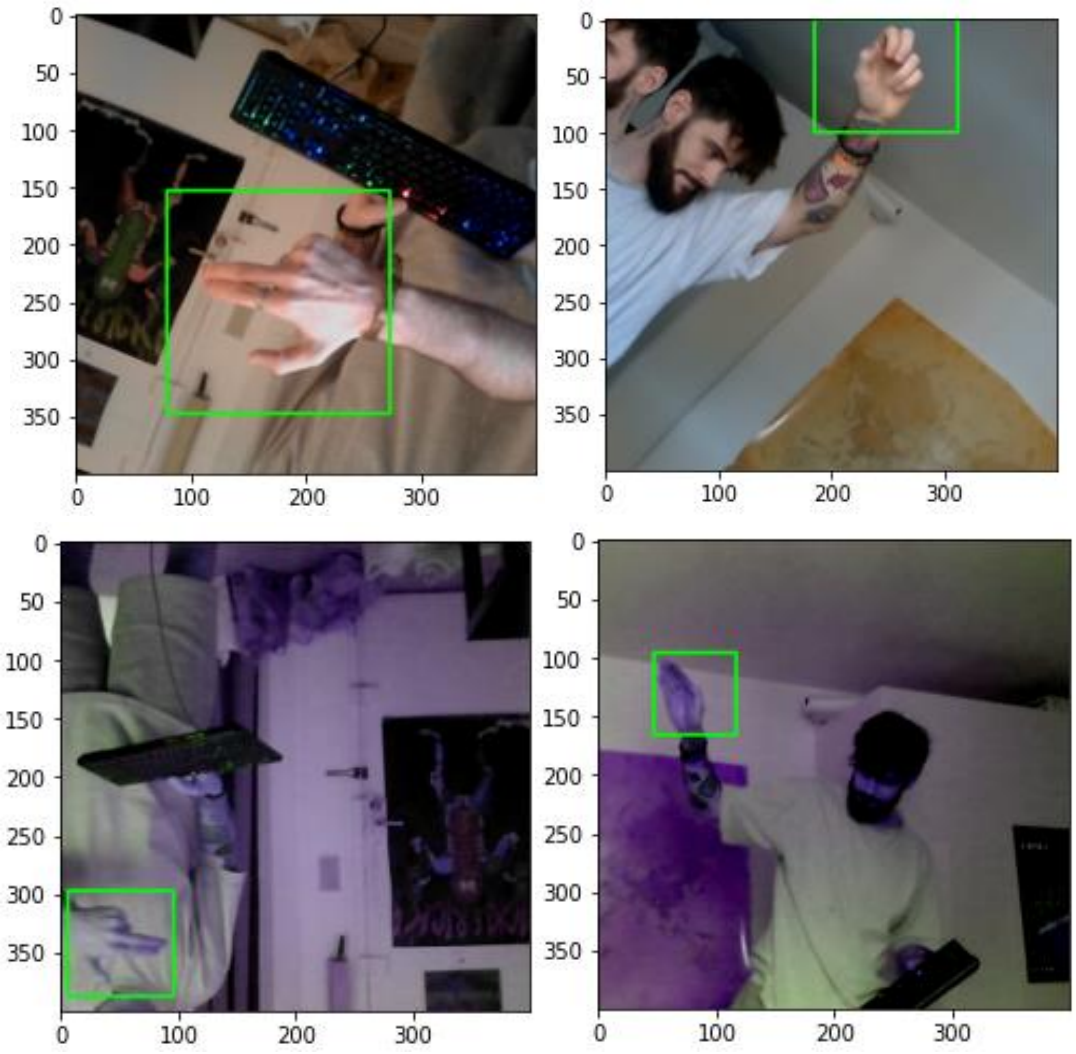


Having decided which labels to use, we then had to find the best physical gestures to match the gestures. The goal was to have each pose *significantly* different from the rest in order to help the model outputs more accurately. Indeed, at first try, our poses for *Left, Right* and *Finger* were somewhat similar and therefore indifferent to the computer.

After choosing a pose for each label, we proceeded to the data augmentation step. This is the process of generating new training images from the originals in order to "trick" the model into thinking that the images are indeed different. There are pros and cons to this method. It helps us save time by obtaining several times more images for training depending on how many augmentation techniques are being applied. By tweaking the original image's color and brightness for example, the network would also work better in various lighting conditions should the robot operate in different environments at different times of the day. However, if done excessively and carelessly, data augmentation can lead to severe overfitting which hurts inference accuracy.

Precisely, we used PyTorch's built-in augmentation tool and applied a number of techniques such as changing color's temperature, polarizing, color saturation, altering brightness and contrast, adding Gaussian noise, etc. We also utilized some positional transformations such as flipping, rotating, or slight cropping, but with

careful consideration: flipping an image having a *Left* label would turn it into *Right* and vice versa. For images containing a bounding box label, we looked into the albumentation library in order to correctly transform them without falsifying the correct label.

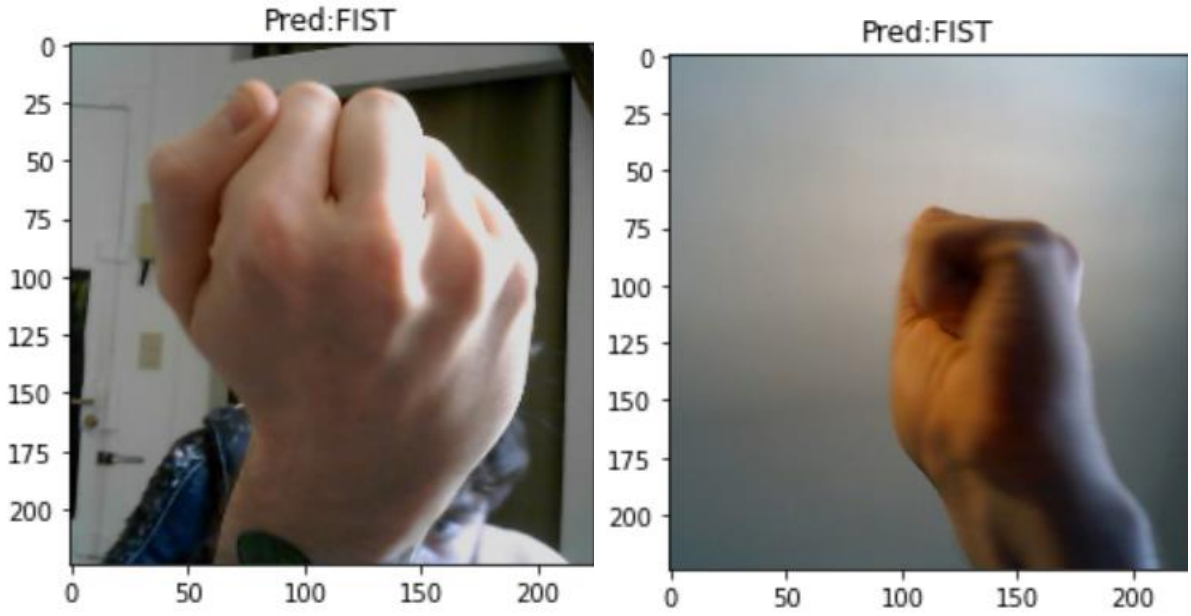


Despite our efforts, overfitting was still a significant problem while we were training our models. Although loss value was minimized over training, validation loss was quite high and accuracy was low, which was a big disappointment. We later discovered that by better separating our test set from the training set, the models achieved much better performance. In fact, we used images taken by Enzo for training and those taken by Damien for validation. By having two completely different sets, the models were better at generalizing and therefore saved us a lot of time.

On top of that, we had to make sure that no gestures were better classified than the others by using the classification score function from the scikit-learn library. The following metrics were computed from the two-stage classifier model showing a nice balance and non-biased predictions among the classes.

	precision	recall	f1-score	support
0	0.98	1.00	0.99	49
1	0.98	0.98	0.98	50
2	1.00	0.98	0.99	50
3	0.97	1.00	0.99	38
4	1.00	0.98	0.99	45
accuracy			0.99	232
macro avg	0.99	0.99	0.99	232
weighted avg	0.99	0.99	0.99	232

By visualizing some predictions by the model, we got a better idea of how it worked. First, we looked specifically at images classified as *Fist* to make sure that the model did not mistake it with *Palm* which was possible since they are more similar than any other pairs of labels. We then noticed that the model misclassified quite often while the hand is further away, which led us to adding more training images only having a hand far away from the camera.



3.2. Model training

After pre-processing the images, we started training models using VGG16 as the baseline of transfer learning for our three CNNs of the two solutions mentioned in the last part. The goal is to freeze the feature extracting layers and to only train the classifier.

Each problem required a unique output to which we must adapt, for example the hand gesture classifier should output a vector of five elements corresponding to the five classes that we defined, while the bounding box CNN should output four numbers indicating the coordinates of the box.

To proceed with the training, we started with a simple model with only a few layers and neurons and without any data augmentation. We then gradually added more neurons and layers to the network's architecture in order to complexify the task, which in return helped us not only catch bugs but also approach overfitting. Once overfitted, we backed up by juggling with regularization techniques: either fine-tuning the hyperparameters or adding a dropout layer or augmenting the training data.

To save time while testing numerous possible courses, we simultaneously launched several Jupyter notebooks on Google Colab and Kaggle and compared the findings once finished.

The network's output value was also one of the problems that we had to consider. Initially, we chose Softmax as the activation function for the output layer to better highlight the label that stood out in prediction. However, we had overlooked an important aspect of this function: it cannot tell if the input image belongs to none of the defined classes. This led to cases during inference testing when the model predicted one of the hand gestures while there is no hand showing in the image! To correct this, we could either add a *None* class along with new training images, or use Sigmoid instead of Softmax for activation function and set an arbitrary threshold value below which we considered as *None*. To avoid having to redo the dataset and retrain the network, we chose the second approach which worked out nicely.

In the end, the one-stage classification model did not manage to perform well so we decided to focus on the two-stage model. The classifier network was easier to train since the dataset contained only hand gestures without much background and other distractions. It quickly gave over 95 percent of accuracy on all the labels and also worked well in inference. The bounding box network was harder to get right but by the time of writing this paper, it can produce correct bounding boxes about 70% of the time tested.

4. What next?

To finally assemble every element that we have done, we produced the last script that loads the trained model of the two-stage classifier, receives and passes the live images from the robot to the networks, interprets the output, and issues the appropriate commands. Unfortunately, due to the prediction accuracy still being low, the robot's performance is left to be desired.

If given more time, we could try simple techniques such as increasing the size of the bounding box. However, to improve upon what we have built, more data is required. Furthermore, we could also unfreeze the baseline model VGG and retrain the whole network to better fit our needs, but it would take much more time and resources. We could also try another optimizer than the current one, Adam, which could be more suitable.

Another approach that we could explore is a one-stage object detection model such as YOLO.

Last but not least, we could also completely change our objective by picking a simpler task such as *Follow-it*. By choosing a distinguishable object like a black tennis ball, our network could easily recognize its location within the image and issue correct commands to the robot.

5. Conclusion

Despite not achieving all of our technical goals, the project has led us to exploring many interesting aspects of embedded systems, CNN optimization, transfer learning, etc. We have also learned that machine learning and neural networks are not at all magic that can solve any problem. Without careful consideration and patience during each step, we would not be able to achieve results nearly as good.

We often hear that in real life's machine learning, processing data is much more difficult than building models. Indeed, we have discovered it by experimenting throughout this project with trials and errors. The difference between the training data and the real-time inference can be overwhelming, so this project has allowed us to discover an interesting application of machine learning with several key problems.

Bibliography

[1] Dexter Industries. 2022. GoPiGo3 Documentation. Retrieved February 8, 2022 from https://readthedocs.org/projects/gopigo3/downloads/pdf/latest/

[2] Ahamed Mollick Faruque. *Capture and save webcam video in Python using OpenCV*. Retrieved February 8, 2022 from https://www.codespeedy.com/save-webcam-video-in-python-using-opencv/

[3] Omar M Faruque Sarker. 2014. Python network programming cookbook.

[4] Nathan Jennings. 2022. Socket Programming in Python (Guide). *Real Python*. Retrieved February 22, 2022 from https://realpython.com/python-sockets/

[5] 2020. Jetson Nano Developer Kit. Retrieved February 8, 2022 from https://developer.nvidia.com/embedded/jetson-nano-developer-kit

[6] 2021. GoPiGo OS v3.0.1 · GoPiGo.io. *GoPiGo.io*. Retrieved February 8, 2022 from https://gopigo.io/gopigo-os-v-3-0-1/

[7] 2022. Install PyTorch on Jetson Nano. *Q-engineering*. Retrieved February 8, 2022 from https://qengineering.eu/install-pytorch-on-jetson-nano.html

[8] Start Locally | PyTorch. *PyTorch*. Retrieved February 8, 2022 from https://pytorch.org/get-started/locally/

[9] GoPiGo3 Tutorials and Documentation for your Raspberry Pi Robot Car. *Dexter Industries*. Retrieved February 8, 2022 from https://www.dexterindustries.com/gopigo3-tutorials-documentation/

[10] Connect to the GoPiGo3 with Raspbian for Robots. *Dexter Industries*. Retrieved February 8, 2022 from https://www.dexterindustries.com/GoPiGo/get-started-with-the-gopigo3-raspberry-pi-robot/2-connect-to-the-gopigo-3/raspbian-for-robots-operating-system/

[11] Vincent Dumoulin and Visin Francesco. 2018. A guide to convolution arithmetic for deep learning. Retrieved from https://arxiv.org/pdf/1603.07285.pdf

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. The MIT Press, Cambridge, Massachusetts.

[13] Yann Le Cun, Caroline Brizard, and Caroline Brizard. 2019. *Quand la machine apprend: la révolution des neurones artificiels et de l'apprentissage profond*. Odile Jacob, Paris.

[14] Training a Classifier — PyTorch Tutorials. *PyTorch*. Retrieved February 8, 2022 from https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

[15] Transfer learning with PyTorch. *PyTorch*. Retrieved February 8, 2022 from https://pytorch.org/vision/stable/models.html

[16] Finetuning Torchvision Models. *PyTorch*. Retrieved February 8, 2022 from https://pytorch.org/tutorials/beginner/finetuning_torchvision_models_tutorial.html

[17] ONNX with PyTorch. *PyTorch*. Retrieved February 8, 2022 from https://pytorch.org/docs/stable/onnx.html