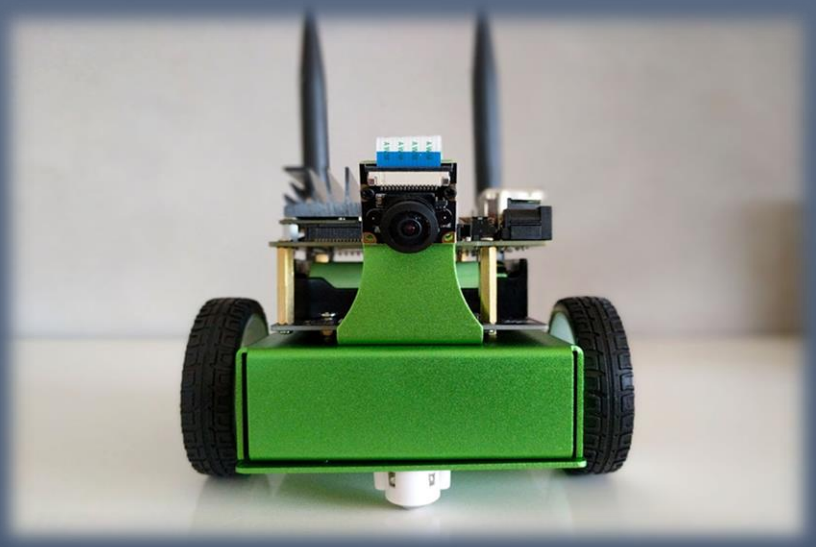
2021
/2022



Jetson Nano
MINI-ROBOT DE TYPE SOURIS

REALISE AUX HALL TECHNOLOGIQUE
CAMPUS D'ILKIRCH, API.

UNIVERSITE DE STRASBOURG |Master 2 Mécatronique et Energie

Année universitaire **2021-2022**

MINI-ROBOTS DE TYPE SOURIS

Présenté par **Ali MARZAK et Sourena MOHIT-TABATABAIE**

CLASSE : **MASTER 2 Mécatronique et Energie**

Sous la direction de **Dominique KNITTEL**, Responsable du Master Mécatronique et Energie

Projet présenté le 11/01/2022. Devant un jury composé

Dominique KNITTEL, Responsable Master et Professeur

Philippe CELKA, Professeur

Joel FRITSCH, Professeur

Projet de Master 2 à l'université de STRASBOURG

RESUME/ABSTRACT

A. RESUME

L'Intelligence Artificielle est une avancée majeure de ces dernières décennies. En effet, la société cherche de plus en plus à rendre la vie simple. Pour cela, la recherche se portent sur des algorithmes permettant de rendre des robots autonomes. On pourrait citer les voitures autonomes ou en encore la reconnaissance d'image qui permettent d'effectuer un travail monstrueux pour l'humain en peu de temps.

Ainsi nous avons vu la notion de DEEP Learning qui est un domaine de l'Intelligence Artificielle. Cette notion nous permet de travail avec des modèles de réseaux neurones qui permettent d'effectuer du traitement d'image. Cette capacité nous a permis de faire fonctionner notre robot Jetbot sur un circuit pour qu'il suive une ligne par régression mais aussi qu'il évite les obstacles par classification. Grâce au DEEP Learning et son modèle ResNet, nous avons pu permettre au Jetbot de suivre la ligne sans sortir du circuit.

B. ABSTRACT

Object detection and recognition are core AI techniques that find applications in self-driving cars, robotics, and traffic management. This technique is only meant for coarse detections and real-world scenarios often need more precision and higher resolution in their detections. The image segmentation technique suffices such requirements and is capable of pixel-level detection of objects and their further classifications.

Artificial Intelligence is a major breakthrough in recent decades. Indeed, society is increasingly seeking to make life simple. For this, research is focused on algorithms to make robots autonomous. We could cite self-driving cars or even image recognition that can do phenomenal work for humans in a short time.

So, we saw the notion of DEEP Learning which is a field of Artificial Intelligence. This notion allows us to work with models of neural networks that allow us to perform image processing. This ability allowed us to operate our Jetbot robot on a circuit to follow a line by regression but also to avoid obstacles by classification. Thanks to DEEP Learning and its ResNet model, we were able to allow the Jetbot to follow the line without leaving the circuit.

REMERCIEMENTS

Nous tenons à exprimer tout d'abord nos remerciements aux membres du jury, qui ont accepté d'évaluer notre projet de deuxième année de Master.

Nous voudrions adresser notre reconnaissance à notre responsable de formation Monsieur Dominique KNITTEL qui a été notre tuteur durant ce projet et qui a nous permis de se donner les moyens de réussir ce projet.

Nous associons à ces remerciements notre respect à Joel FRITSCH et Philippe CELKA, professeur à la Faculté de Physique & Ingénierie, pour la confiance et la sympathie qu'ils nous ont témoigné durant ce projet.

Nous tenons à remercier notre ami et collègue Mohamed CHAKROUNE qui a été disponible lorsque nous avions besoin d'aide, vidéo et conseils, durant la réalisation et la préparation de ce projet.

Enfin nous n'oublions pas de remercier toute l'équipe pédagogique de l'Université de Strasbourg qui, pendant cette crise sanitaire majeure du Covid-19, assure le meilleur apprentissage possible. Merci pour leur dévouement, leur disponibilité et leur réactivité. Vous avez tous notre respect et notre gratitude.

SOMMAIRE

II. HARDWARE

A. Jetson Nano NVIDIA

Today, the power of modern artificial intelligence is available to all manufacturers, programming learners, and developers everywhere.

The Jetson Nano Developer Kit is a small, powerful computer that allows the user to run multiple neural networks in parallel for applications such as image classification, object recognition, and speech processing.

All of these features are included in a platform that is easy to use and requires only 5 watts of power to operate.

From Nvidia developer website: *"NVIDIA® Jetson Nano™ makes it possible to bring incredible new capabilities to millions of small, power-efficient AI systems."*

1. But what does one actually do with all of that artificial brain power?

Well, the easiest thing is just to use your Nano as a basic PC. Most other **SBCs** either have no GUI and can only execute a pre-flashed program, or have a pretty basic, low-resolution GUI. Nano, on the other hand, runs pretty much full Ubuntu Linux, with the exception of the kernel being compiled for the ARM processor, and some extra libraries Nvidia decided were important to have on their install image.

2. What is the SBC?

Single board computers

3. What are single-board computers (SBCs) like Raspberry Pi, etc., and what are their uses?

Single Board Computer In short, SBCs are very small computers about the size of the palm of your hand that have the minimum components of a computer such as CPU, GPU, RAM, and connection ports. These computers are much cheaper than other computers and of course have very different uses.

Single-board computers are used in programming, Internet of Things, and so on. Although these computers are much lower in terms of processor and memory (for example, 512 MB of RAM), they always have a lot of input and output ports. SBCs usually have 2 USB ports, one HDMI, 3.5mm jack, Ethernet port, camera cable input, micro-SD port and other inputs such as SPI and GPIO.

If you hear the word SOM, this board is not the only difference with BSCs, I / O´s. That is, it has only the main components such as CPU and RAM, and allows us to implement ports as needed.

The most popular and famous single-board computer is Raspberry Pi, and Arduino can be ranked second.

4. Performance: why Jetson nano is so powerful?

The end of Moore's law is no longer just a prediction, it's a fact the industry has been dealing with for years now, and it has driven the rise of the general-purpose GPU.

Nvidia said: "The NVIDIA Jetson Nano… delivers ×3 to ×4 higher AI performance than platforms such as the Intel Neural Compute Stick 2"

As we can see in the next figure: a GPU isn't just for graphics any more.

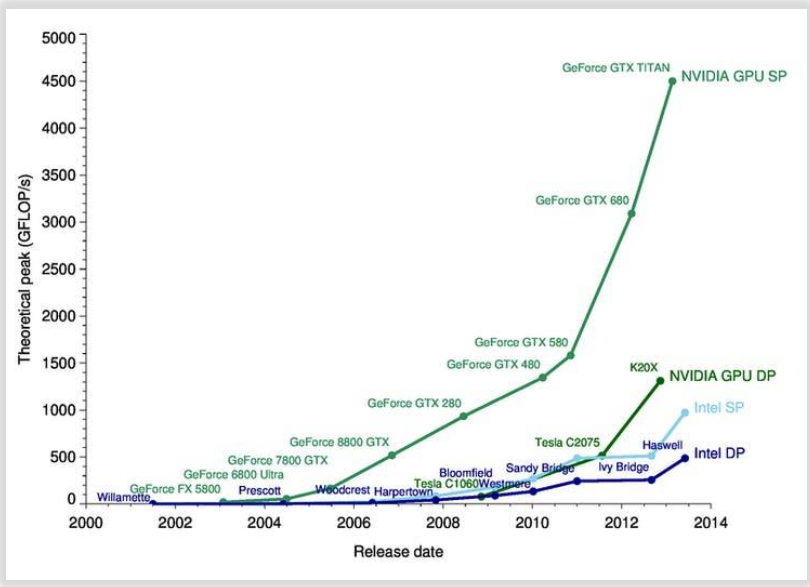


Figure 1 CPU vs GPU performance, Michael Galloy

Now, Jetson nano comes in two versions, the 2-GB and 4-GB developer kits, and is one of the most popular boards to compete with the Raspberry Pi.

In the university labs, we have the 4GB version.

B. Raspberry pi Vs Jetson Nano

For our project, we have 2 major options so:

1. Jetson Nano or Raspberry Pi 4, that is the question.

Raspberry Pi 4: short introduction

Raspberry Pi was first released in 2012. With it, you can enter the heart of the computer by launching an operating system and then connecting the wires and circuits directly to the pins on the board!

Also, it is designed to teach students about programming.

	Raspberry PI 4	Nvidia Jetson Nano
CPU	Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC	64-bit Quad-core ARM Cortex-A57
GPU	Broadcom VideoCore VI (Mali-T880) @ 500MHz	128-core Maxwell GPU @ 921MHz
Clock frequency	1.5GHz	1.43 GHz
System memory	1GB, 2GB, or 4GB LPDDR4–3200 SDRAM 25.6 GB/s	4GB 64-bit LPDDR4 @ 1600MHz 25.6 GB/s

2. The Common Section: The most important sections

1. 40 GPIO pins
2. Ethernet support
3. Linux-based operating system
4. it’s also possible to use existing Raspberry Pi accessories with the Jetson Nano.

3. The Big difference:

The biggest difference lies in the graphics capabilities between the two boards, specifically their graphical processing units (GPU). NVIDIA Jetson Nano has a 128-core Maxwell 128 GPU processor clocked at 921 MHz Compared to each other, the Jetson Nano has a much more powerful graphics processor than the Raspberry Pi 4.

4. Conclusion

The Jetson Nano has a much more powerful GPU. For machine learning and artificial intelligence applications, the Jetson Nano remains the better choice.

Raspberry Pi 4 supports dual display whereas the Jetson Nano does not. So, if you’re after a home entertainment system.

The Cortex-A72 on the Raspberry Pi 4 is a newer generation than the Cortex-A57 on the NVIDIA Jetson Nano. This CPU offers higher performance and higher clock speed. But for deep learning and artificial intelligence, it may not offer enough performance benefits.

5. Is Nvidia Jetson Nano better than Raspberry Pi 4?

Compared to Raspberry Pi 4B Jetson Nano is expensive in the same RAM configuration. For day-to-day computing activities and embedded work and projects, Raspberry Pi is a better value for money. Only when projects demand GPU usage or ML or AI applications that can benefit from CUDA cores you should consider Jetson Nano.

C. Why Python is Best for AI ?

A great library ecosystem	A good selection of libraries is one of the main reasons why Python is AI's most popular programming language. A library is a module or group of modules released from different sources (PyPi). It includes a pre-written code segment that allows a user to use a particular function or perform various operations. The Python library provides base-level items, so developers do not have to write code from scratch every time. Machine learning requires continuous data processing, and Python libraries allow you to access, process, and transform your data. These are some of the most extensive libraries available for AI and ML.
Platform independence	Python is easy to use, learn, and it is versatile too. It means that Python, which is used to develop machine learning, can run on all platforms, including Windows, Linux, Unix, macOS, and 21 others.
Good visualization options	We have mentioned that Python comes with many libraries, some of which are great visualization tools. However, AI developers need to point out that it is vital to represent data in a human-readable format in AI, deep learning, and machine learning.
Versatility	Python is easy to use and supports various libraries and frameworks, making the language more versatile. However, it works in two categories. 1. Web development 2. Machine learning
Readability	Python is easy to read and understand, so Python developers have no problem understanding, modifying, copying, or pasting peer code.
Simple and Consistent	Python code is easy to understand and read. ML and AI support complex algorithms and common workflows, but Python's ease of use allows developers to create reliable systems.

III. ETAT DE L'ART

A. DEEP Learning

Le DEEP Learning ou apprentissage profond est un domaine du Machine Learning, qui est capable d'apprendre par elle-même à partir d'exemple. Elle correspond à un ensemble de méthodes d'apprentissage automatique basée sur des approches mathématiques et où l'on développe des réseaux de neurones artificielles.

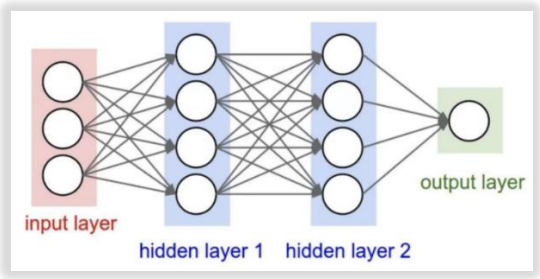


Figure 2 https://www.jedha.co/blog/

Il existe plusieurs types de réseaux de neurone tel que le réseau de neurones à convolution CNN, qui est la plus répandue puisqu'elle s'ouvre à de larges applications, que ce soit la reconnaissance d'image ou vidéo.

Nous pouvons catégoriser l'utilisation du DEEP Learning en 2 phases :

- Phase 1 : Apprentissage
- Phase 2 : Déploiement

Dans la phase d'Apprentissage, il s'agit d'entrainer notre machine pour la rendre fonctionnel pour la phase 2. Le principe de cet entrainement se base sur 3 étapes. Comme illustré sur la figure, nous commençons par fournir des données à la machine puis nous définissons le modèle de réseau neuronal profond que nous voulons utiliser (Exemple : ResNet, AlexNet, VGGNet, …).

Lorsque nous avons les données et le modèle, nous pouvons entraîner notre machine à partir d'un Algorithme d'optimisation qui permettra d'ajuster le modèle en fonction des données. Pour cette optimisation, nous avons l'Algorithme du gradient qui est un algorithme d'optimisation différentiable.

Cette phase d'Apprentissage permettra d'améliorer les paramètres de chaque neurone et ainsi renforcer notre réseau en minimisant les erreurs.

Dans la phase de déploiement, il s'agit de la phase de prédiction qui nous permettra d'observer si notre machine est bien entrainée ou pas. Pour commencer, il suffit de fournir une donnée à la machine entrainée et observer le résultat fourni par la machine. Plus le résultat correspond aux attentes, plus la machine est bien entrainée et est fonctionnel pour une application.

Comme nous pouvons le constater sur l'image, la particularité du DEEP Learning est que les Algorithmes évoluent avec les données contrairement aux DEEP Learning.

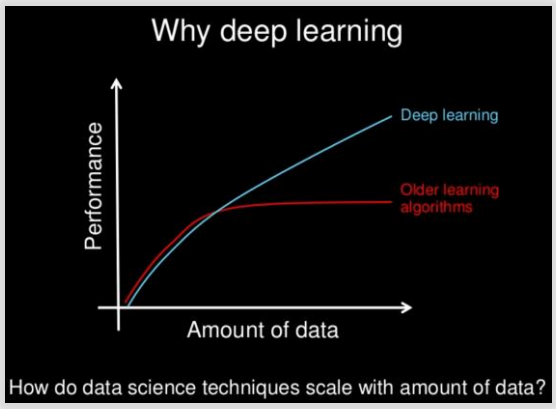


Figure 3 Slide by Andrew Ng, all rights reserved.

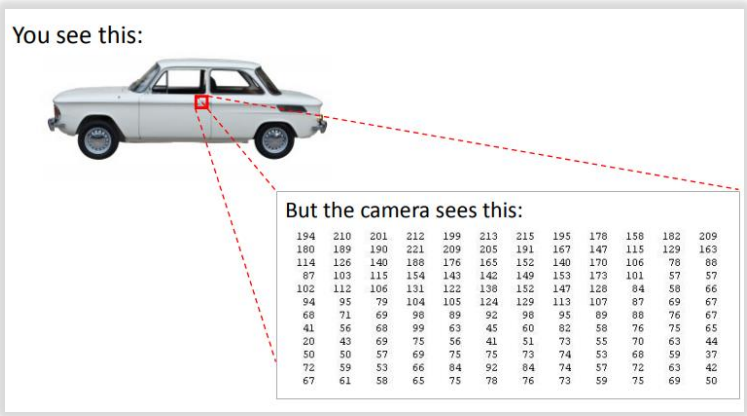
B. Neural Networks: Representation

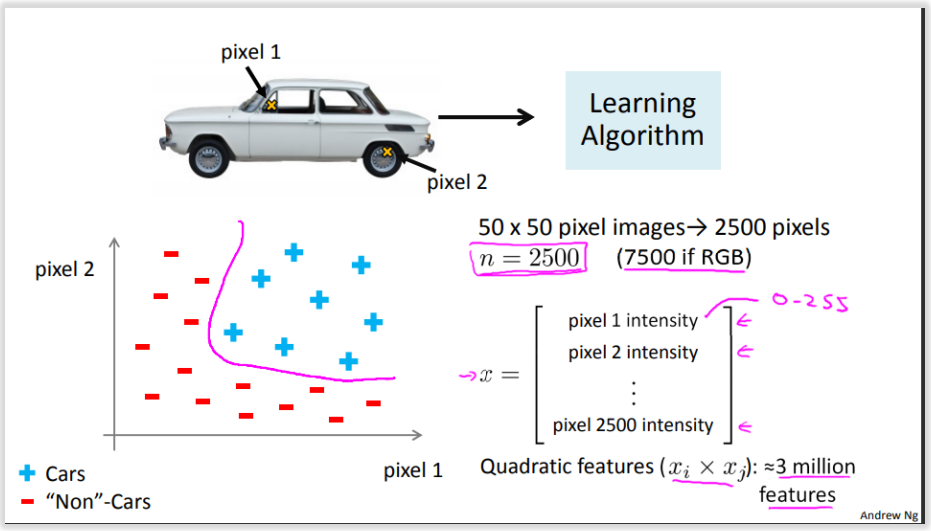
I want to focus in this theory because all the CNN based algorithms and all Deep learning bases come from this method; so, it is important to fully understand the basics of this algorithm.

Neural Networks Origins: Algorithms that try to mimic the brain. Was very widely used in 80s and early 90s; popularity diminished in late 90s.

Computer vision: when we look in to a picture, we will see just a recorded moment in the color but the camera sees the picture in other way, in shape of the one matrix that contains numbers represented the brightness.

So, with a picture we have a Non-linear case and with each number in matrix our pixel we have new feature, for example in a picture with 50*50 pixels and range of the 0-255 for brightness number, our quadratic features will be near the 3M,just in gray scale mode so if we give a test image , with the normal machine learning algorithm and when we have a video streaming it is possible that we get an overfitting situation.





Neuron model: Let's examine how we will represent a hypothesis function using neural networks. At a very simple level, neurons are basically computational units that take inputs (dendrites) as electrical inputs (called "spikes") that are channeled to outputs (axons). In our model, our dendrites are like the input features $X_1 \ldots X_n$, and the output is the result of our hypothesis function. In this model our x_0 input node is sometimes called the "bias unit." It is always equal to 1. In neural networks, we use the same logistic function as in classification: $\frac{1}{1+e^{-\theta ix}}$ In this situation, our "theta" parameters are sometimes called "weights".

Visually, a simplistic representation looks like:

$$[x_0 x_1 x_2] \rightarrow [\ ] \rightarrow h_\theta(x)$$

Our input nodes (layer 1), also known as the "input layer", go into another node (layer 2), which finally outputs the hypothesis function, known as the "output layer".

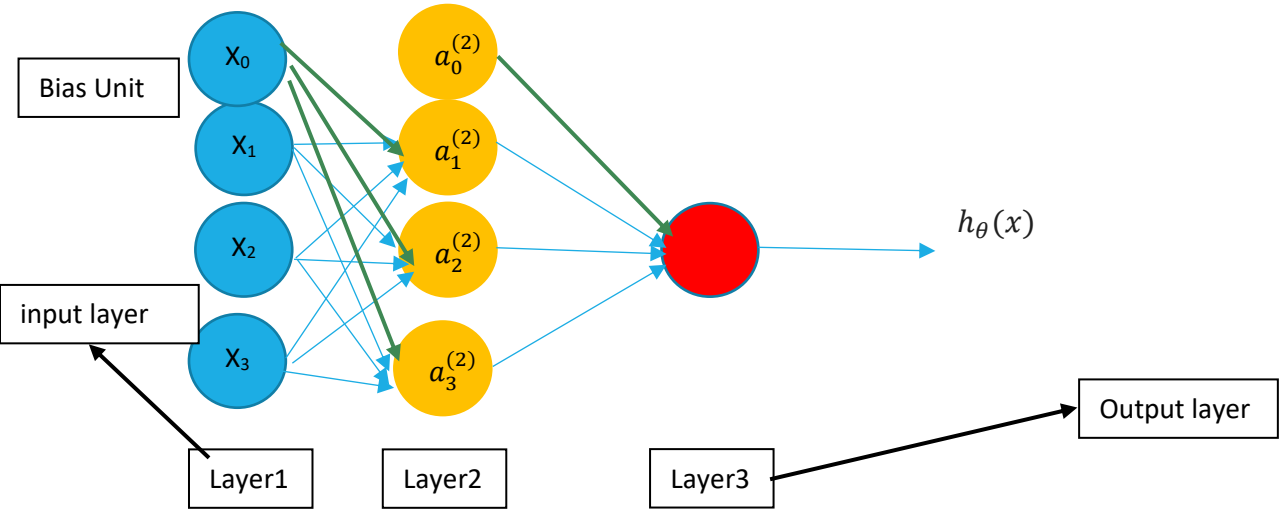
We can have intermediate layers of nodes between the input and output layers called the "hidden layers."

$a_i^{(j)} =$ "***activation*** *of* ***unit*** i ***in layer*** j

$\theta^{(j)}$ =matrix of weights controlling function mapping from layer j to layer j + 1

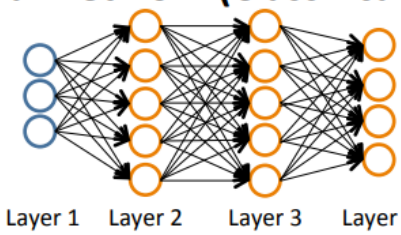
The values for each of the "activation" nodes is obtained as follows:

$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$
$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$



If we have more output , for example if we want to train a picture that can be a car or truck or bicycle , we have:

Neural Network (Classification)



$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$

$L =$ total no. of layers in network

$s_l =$ no. of units (not counting bias unit) in layer l

Cost function

Logistic regression:

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2$$

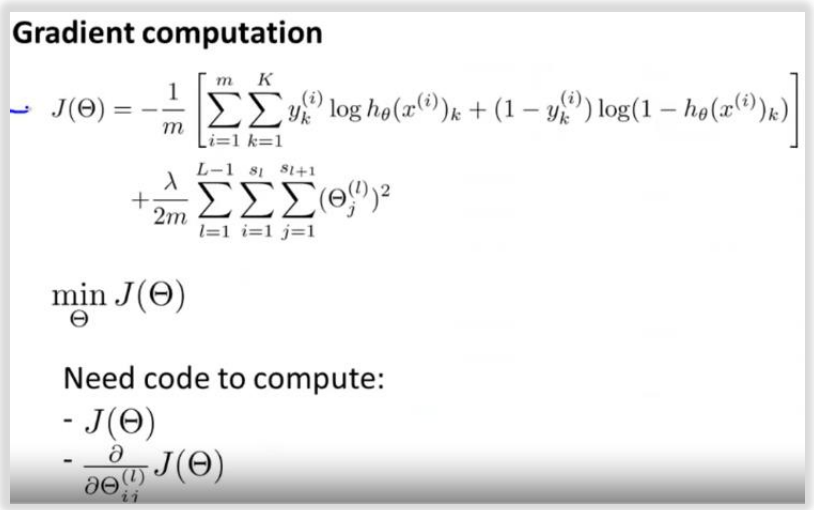
Neural network:

$$h_\Theta(x) \in \mathbb{R}^K \quad (h_\Theta(x))_i = i^{th} \text{ output}$$

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right]$$
$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

Our cost function for neural networks is going to be a generalization of the one we used for logistic regression. Recall that the cost function for regularized logistic regression was and We have added a few nested summations to account for our multiple output nodes. In the first part of the equation, before the square brackets, we have an additional nested summation that loops through the number of output nodes.

In the regularization part, after the square brackets, we must account for multiple theta matrices. The number of columns in our current theta matrix is equal to the number of nodes in our current layer (including the bias unit). The number of rows in our current theta matrix is equal to the number of nodes in the next layer (excluding the bias unit). As before with logistic regression, we square every term. After that we must use the Gradient for minimize our cost function so we have:



Forward propagation (or *forward pass*) refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer

"Backpropagation" is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression. Our goal is to compute:

$\min_\Theta J(\Theta)$

That is, we want to minimize our cost function J using an optimal set of parameters in theta. In this section we'll look at the equations we use to compute the partial derivative of J(Θ):

$$\frac{\partial}{\partial \Theta_{i,j}^{(l)}} J(\Theta)$$

Intuitively, $\delta j^{(l)}$ is the "error" $aj^{(l)}$ (unit j in layer l). More formally, the delta values are actually the derivative of the cost function:

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} cost(t)$$

Putting it Together

First, pick a network architecture; choose the layout of your neural network, including how many hidden units in each layer and how many layers in total you want to have.

- Number of input units = dimension of features $x^{(i)}$
- Number of output units = number of classes
- Number of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units)
- Defaults: 1 hidden layer. If you have more than 1 hidden layer, then it is recommended that you have the same number of units in every hidden layer.

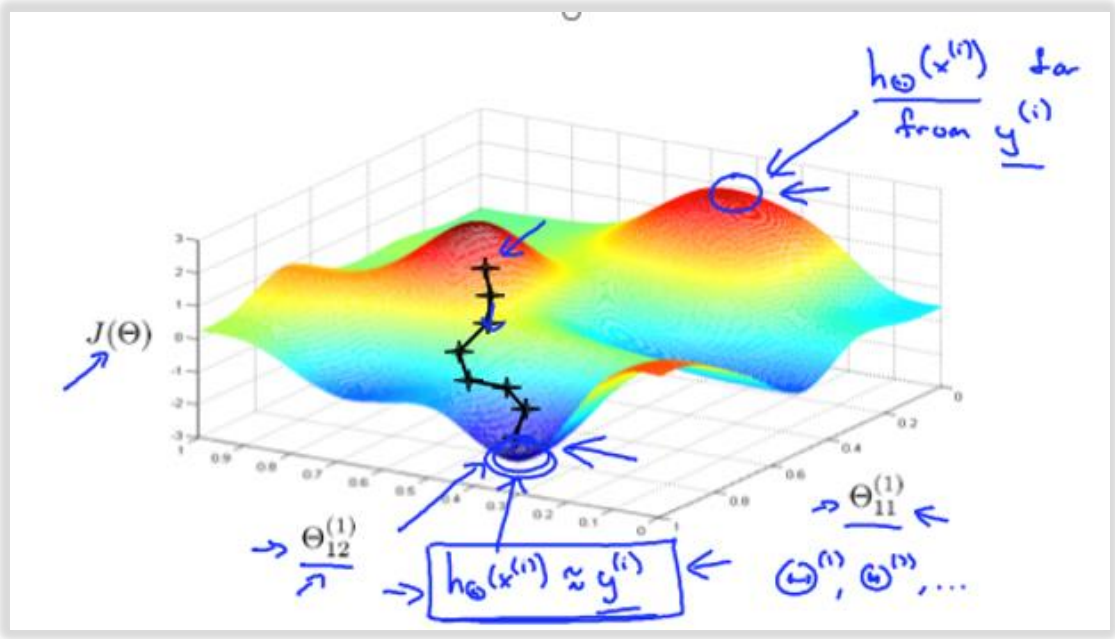
Training a Neural Network

1. Randomly initialize the weights
2. Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
3. Implement the cost function
4. Implement backpropagation to compute partial derivatives
5. Use gradient checking to confirm that your backpropagation works. Then disable gradient checking.
6. Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.

for i = 1:m,
Perform forward propagation and backpropagation using example (x(i),y(i))
(Get activations a(l) and delta terms d(l) for l = 2,...,L

7. Ideally, you want $h\Theta(x^{(i)}) \approx y(i)$. This will minimize our cost function. However, keep in mind that $J(\Theta)$ is not convex and thus we can end up in a local minimum instead.

As shown in figure: in 2 level examples (simple)



C. Types de réseaux neuronaux

Précédemment, nous avons dit qu'il existait différents types de réseaux de neurones. En effet, nous pouvons les regrouper en 4 grandes familles, il y a les réseaux de neurones :

- **Feed foward :** il s'agit d'un réseau où la donnée traverse le réseau de l'entrée à la sortie sans retour d'information.

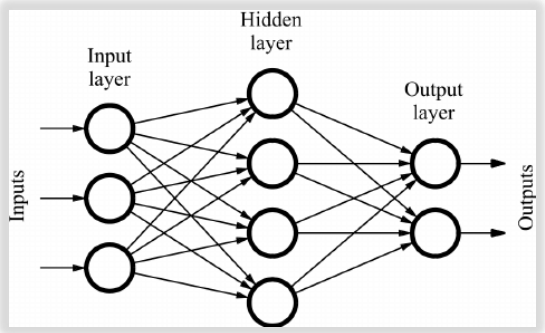


Figure 4 https://www.researchgate.net/

Dans cette famille, nous avons les réseaux monocouches possédant 2 couches (entrées et sorties) les réseaux multicouches possédant des couches « cachés » entre les couches d'entrées et sorties.

Pour les informations complexes, il existe des réseaux neuronaux dites convolutifs (CNN) qui correspond à une succession de réseaux de neurones correspondant chacune à une partie différente des informations.

Très souvent utilisé pour le traitement d'image et de vidéos.

- **Récurrent (RNN) :** Contrairement au réseau précédent, celle-ci possède un retour d'information au sein du réseau. Ainsi la sortie dépend des entrées mais aussi des sorties précédentes.

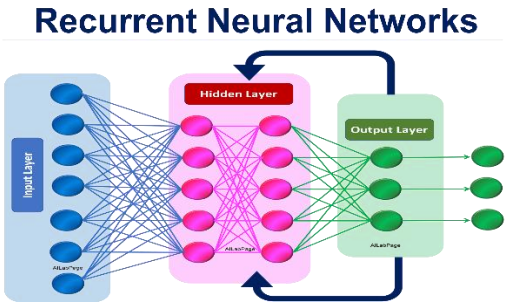


Figure 5 https://morioh.com/p/

Dans cette famille, les 2 réseaux les plus connus sont le LSTM (Long Short-Term Memory) et GRU (Gated Recurrent Unit).

Cette famille est adaptée aux données séquentielles.

- **à résonance :** Il s'agit comme précédemment d'un réseau avec retour mais avec oscillation. En effet, le retour d'activation des neurones aux neurones précédents provoque une oscillation

- **de Kohonen :** cette famille de réseau permet d'étudier la répartition de données dans un espace à grande dimensions.

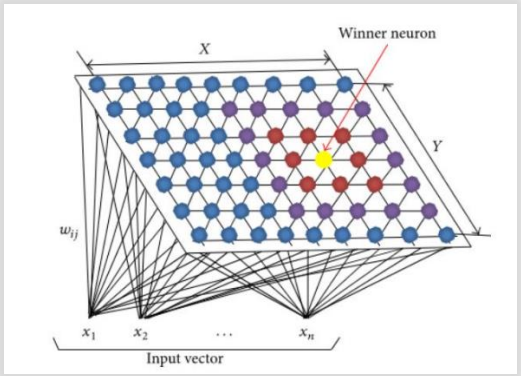


Figure 6 https://www.analyticsvidhya.com/blog/

Elle est caractérisée par un réseau de neurones en forme de grille où chaque nœud correspond à un neurone.

Développement : L'avenir du DEEP Learning pourrait bien se reposer sur une nouvelle génération des réseaux de neurones, appelé « SPIKING NEURAL NETWORKS ».

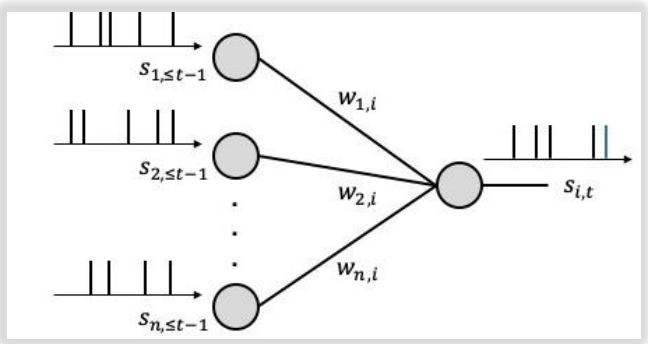


Figure 7 https://blogs.kcl.ac.uk/

Ce réseau de neurone se rapproche beaucoup plus des neurones biologiques humaines. Ce réseau consiste à traiter des évènements ponctuels plutôt que des événements continue.

Fonctionnement : Lorsque le potentiel d'un neurone dépasse une certaine valeur, celle-ci retombe et envoie un signal.

Pour notre projet, les réseaux qui correspondent le mieux à nos domaines d'applications sont les réseaux neuronaux convolutifs.

D. Frameworks & Modèle de réseau

1. Frameworks

Pour entraîner notre réseau neuronal, nous avons besoin d'un modèle. Bien sûr, chercher et utiliser le modèle peut s'avérer un peu long et peu productif. Pour résoudre ce problème, nous allons passer par des bibliothèques de programmation Python, programmés en Open Source dits « Frameworks » permettant d'utiliser des modèles complexes avec seulement quelques lignes.

Parmi ces Frameworks, il y en 3 qui sont populaires :

- Pytorch, développé par Facebook
- Keras, développé par François Chollet chez Google
- Tensorflow, développé par Google

Nous allons ainsi voir les différences entre les librairies à partir du tableau ci-dessous :

	Popularité	Vitesse	Datasets utilisables	Architecture	Débuggage	Niveau d'API
Keras (intégré à TensorFlow)	1er plus populaire	moyenne, basse performance	taille moyenne	lisible, simple	réseaux simples, pas de débuggage nécessaire	haut
TensorFlow	2ème plus populaire	rapide, haute performance	grande taille	difficile à utiliser	difficile à débugger (même s'il y a maintenant une "eager execution")	bas ou haut
PyTorch	3ème plus populaire	rapide, haute performance	grande taille	complexe	bonnes capacités de débuggage	bas

Figure 8 https://datascientest.com/pytorch-tout-savoir

Il faut savoir que Tensorflow est capable d'utiliser Keras, ce qui lui a permis de devenir la référence dans le DEEP Learning. Ainsi il est plus intéressant de comparer Tensorflow et Pytorch.

Malgré sa Tensorflow reste un Frameworks avec une documentation assez compliquée. Contrairement à Pytorch qui permet une accessibilité assez simplifiée et un débogage rapide.

Dans notre cas, nous allons choisir le Frameworks Pytorch.

2. Modèle de réseau neuronale

Il existe plusieurs modèles de réseaux neuronaux. Mais avant de voir ces modèles, il est impératif d'évoquer l'une des premières structures de réseaux neuronales, qui est le modèle LeNet ou LeNet-5. Il s'agit d'un modèle, proposé par Yann LeCun et Co. en 1998, conçu pour la reconnaissance de caractères manuscrits et imprimés à la machine aux années 1990. C'est une architecture très simple qui avait pour base de données des images de chiffres en noir et blanc.

Pour revenir aux modèles de réseaux, nous pouvons en cité 4 qui sortent du lot (du plus ancien au plus récent) :

- **AlexNet** est une architecture de réseaux de neurones convolutifs de 8 couches de profondeur (5 couches convolutifs et 3 couches associés).

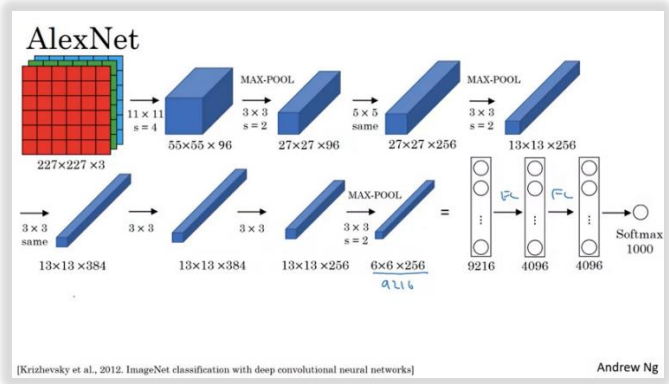


Figure 9 https://medium.com/

Ce modèle est semblable au modèle LeNet mais avec une plus grande profondeur et un meilleur filtrage. De plus, il s'agit du premier modèle basé sur GPU ce qui lui apportait une vitesse plus rapide que les modèles précédents.

Un des inconvénients d'AlexNet est son grand nombre d'hyper-paramètres.

- **VGGNet** est un réseau qui introduit le concept de regroupement de plusieurs couches de convolution avec des petits noyaux au lieu d'une couche de convolution avec un grand noyau. Ce modèle comporte 19 couches.

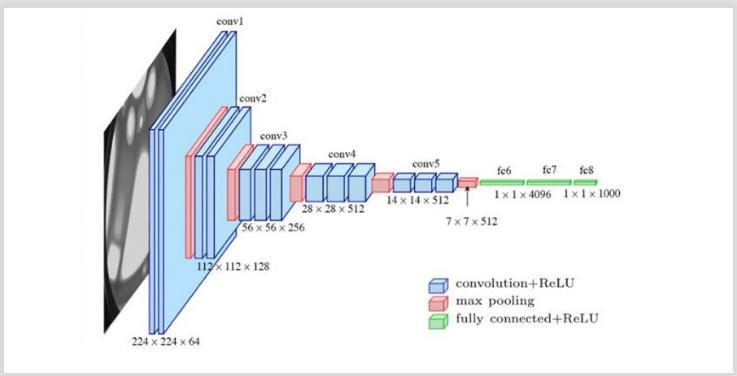


Figure 10 https://medium.com/

Cette amélioration a permis de résoudre l'inconvénient précédemment cité du modèle AlexNet. Malgré cette amélioration, ce modèle comporte un inconvénient majeur qui est le problème du gradient de fuite.

On parle de gradient d'explosion ou de disparition, lorsque la valeur du gradient devient trop importante ou nulle. En effet, plus une architecture surpasse la précédente, plus elle utilise de couche pour réduire les erreurs, ce qui nous provoque ce phénomène.

- **InceptionNet** ou GoogleLe Net est un réseau de neurone qui se composent de composants répétés appelés « Inception modules ». Ce module possède 28 couches avec les couches de mise en commun.

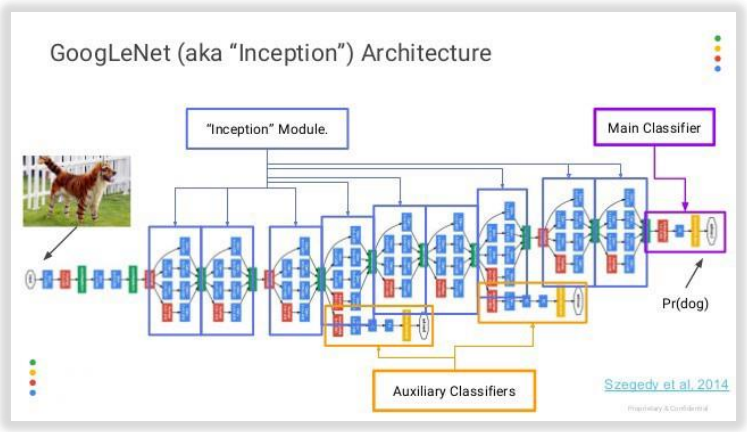


Figure 11 https://medium.com/

- **ResNet :** Pour pallier les problèmes du gradient, cette architecture utilise le concept de Réseau résiduel.

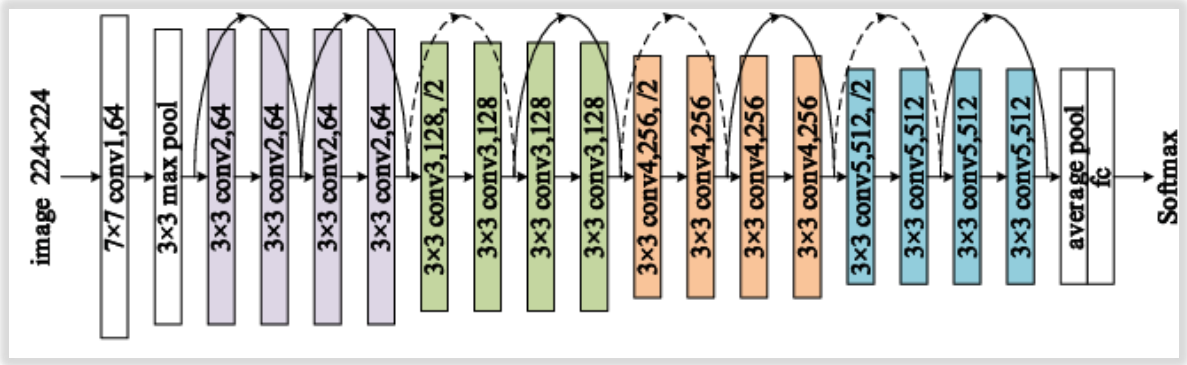


Figure 12 https://www.semanticscholar.org/

Dans ce réseau, nous avons des sauts de connexions qui permettent aux couches introduites de ne pas dégrader le réseau de neurones et ainsi de ne pas avoir de pertes de la précision d'entrainements malgré l'augmentation du nombre de couches.

Comparison					
Network	Year	Salient Feature	top5 accuracy	Parameters	FLOP
AlexNet	2012	Deeper	84.70%	62M	1.5B
VGGNet	2014	Fixed-size kernels	92.30%	138M	19.6B
Inception	2014	Wider - Parallel kernels	93.30%	6.4M	2B
ResNet-152	2015	Shortcut connections	95.51%	60.3M	11B

Figure 13 https://towardsdatascience.com/the-w3h-of-alexnet-vggnet-resnet-and-inception-7baaaecccc96

A partir du tableau ci-dessus, on peut se permettre de comparer les différents modèles de réseaux de neurones artificielles à partir du taux de précision, du nombre de paramètres pouvant être entrainée et du nombre d'opérations en virgule flottante par seconde (FLOP).

Tout d'abord, on remarque que VGGNet est le modèle dont les données des paramètres et les FLOP sont les plus élevées avec une efficacité de 92,3%.

Malgré un taux d'erreur très bas, le modèle InceptionNet comporte un nombre de paramètres et de FLOP bas.

On constate que ResNet est celui qui possède un taux de précision très élevée, s'élevant à 95,51% avec un nombre de paramètre et de FLOP très appréciable.

Dans notre cas, nous avons décidé de choisir le modèle de réseau de neurone ResNet18 possédant 18 couches profondes.

IV. TRAVAIL REALISE

A. ROAD FOLLOWING

Pour notre projet, nous allons entrainer le robot pour qu'il suive une ligne. Pour cela, nous allons entrainer notre réseau neuronal de telle sorte qu'elle prédît les valeurs X et Y de l'étiquette. Nous pouvons comparer cette application comme un fromage suspendu sur un bâton et le Jetbot comme une souris voulant attraper le fromage.

Comme nous l'avons vu précédemment, nous avons 3 grandes étapes pour atteindre notre objectif : la collecte de données, l'entrainement des données et le chargement du modèle déformé.

Collecte de données :

Sous Jupyter Notebook et son langage Python, nous commençons tout d'abord par importer toutes les bibliothèques qui nous serons utiles tel que OpenCV permettant de visualiser et enregistrer les images avec les étiquettes, ou encore uuid et datetime permettant de nommer les images en fonction de la date. Il existe d'autres bibliothèques comme ipywidgets qui sera pour nous permettre de cliquer un point sur l'image et de prendre les coordonnés.

Lorsque nous avons importé les bibliothèques, il est temps de collecter les données. Pour cela, nous allons créer un dossier qui accueillera les données. Ensuite on affiche la caméra avec l'ipywidget et configure le nom des données.

Pour la prise de données, on positionne le robot dans des positions différentes et à chaque fois on annote avec un point vert en cliquant sur l'endroit voulu comme représenté ci-dessous.

La question qu'on peut se poser est : Où doit-on placer le point sur la caméra ?

Bien évidemment, nous plaçons le point sur la ligne que nous voulons suivre. Lorsque nous sommes en ligne droite, nous pouvons mettre le point loin du robot contrairement aux virages où il est préférable de mettre les points plus proches du robot.

Entrainement des données

Dans cette étape, nous allons former notre réseau neuronal ResNet18 de telle sorte qu'à partir d'une image, nous obtenons une cible caractérisée par des valeurs X et Y.

L'étape première de l'entrainement est de diviser les données en 2 groupes : Entrainement et Test.

Durant notre projet, nous avons tout d'abord entrainer le réseau avec 10% de données Test et ensuite avec 30% de données Test.

En Intelligence artificielle, les données sont très souvent divisées pour 30% de Test puisqu'il s'agit d'une moyenne où lorsqu'on augmente, le réseau sous-entrainée et lorsqu'on diminue le réseau est surentraînée.

Ensuite on créer des chargeurs de données pour charger des données par lots en utilisant la classe « DataLoader ».

Enfin on définit notre réseau, ici ResNet18, qu'on entraine sur un nombre d'époque défini et nous choisirons le meilleur modèle avec la perte la plus basse.

Chargement du modèle entrainé :

Dans cette étape, on charge le modèle entrainé.

Nous transférons les poids du modèle au GPU puisqu'elle se situe aux CPU. Nous utiliserons torch2trt, préalablement installé, qui permettra de convertir et optimiser le modèle entrainé en un modèle TRT plus rapide. Une fois que nous avons notre nouveau modèle TRT, on doit le charger mais le suivi de ligne n'est pas encore fonctionnel.

En effet, nous devons avoir un format de caméra similaire à l'entrée du réseau neuronal, ce qui n'est pas encore notre cas.

Il s'agit d'un prétraitement en 4 étapes :

- Conversion du format HWC en format CHW
- Exécution du réseau neuronal
- Calcul de la valeur approximative de la direction
- Contrôle des moteurs à l'aide de correcteur PD (Proportionnel/Dérivé)

Pour le suivi de ligne, nous avons entrainé 2 modèles : l'un avec un groupe de TEST à 10% et un autre avec un groupe de TEST à 30%. L'objectif initial était de comparer les 2 modèles.

Il faut savoir que pour le modèle n°1, nous avons utilisé 200 images du circuit (100 images avec une pièce éclairée et 100 images avec une pièces sombres).

Et pour le modèle n°2, nous avons utilisé 100 images (50 images avec une pièce éclairée et 50 avec une pièce sombre).

B. COLLISION AVOIDANCE

Contrairement aux suiveurs de lignes qui utilise une approche de régression, l'évitement de collision utilise une approche de classification. L'évitement de collision permettra au Jetbot de tourner sur lui-même lorsqu'il se retrouve face à une situation dangereuse.

Il s'agit du même principe qui est d'entrainer un réseau de neurones (ResNet18) mais la collecte de données est différente.

En effet, nous avons créé deux dossiers qu'on appellera « Libre » et « Bloqué » qui accueillera :

- Dans le premier tous les images où nous jugeons que le Jetbot peut continuer son chemin
- Dans le deuxième tous les images où nous jugeons que le Jetbot doit tourner sur lui-même pour éviter l'obstacle.

C. ROAD FOLLOWING & COLLISION AVOIDANCE

Il s'agit tout simplement de la combinaison du Suiveur de ligne et de l'Evitement de collision. On charge les 2 modèles entrainées précédemment et on ajoute quelques lignes de codes qui permettront au Jetbot de suivre une ligne et de s'arrêter simultanément sans tourner sur lui-même face à un obstacle.

V. Difficultés rencontrées

Durant ce projet, nous avons rencontré plusieurs difficultés, quel soit grave ou minime.

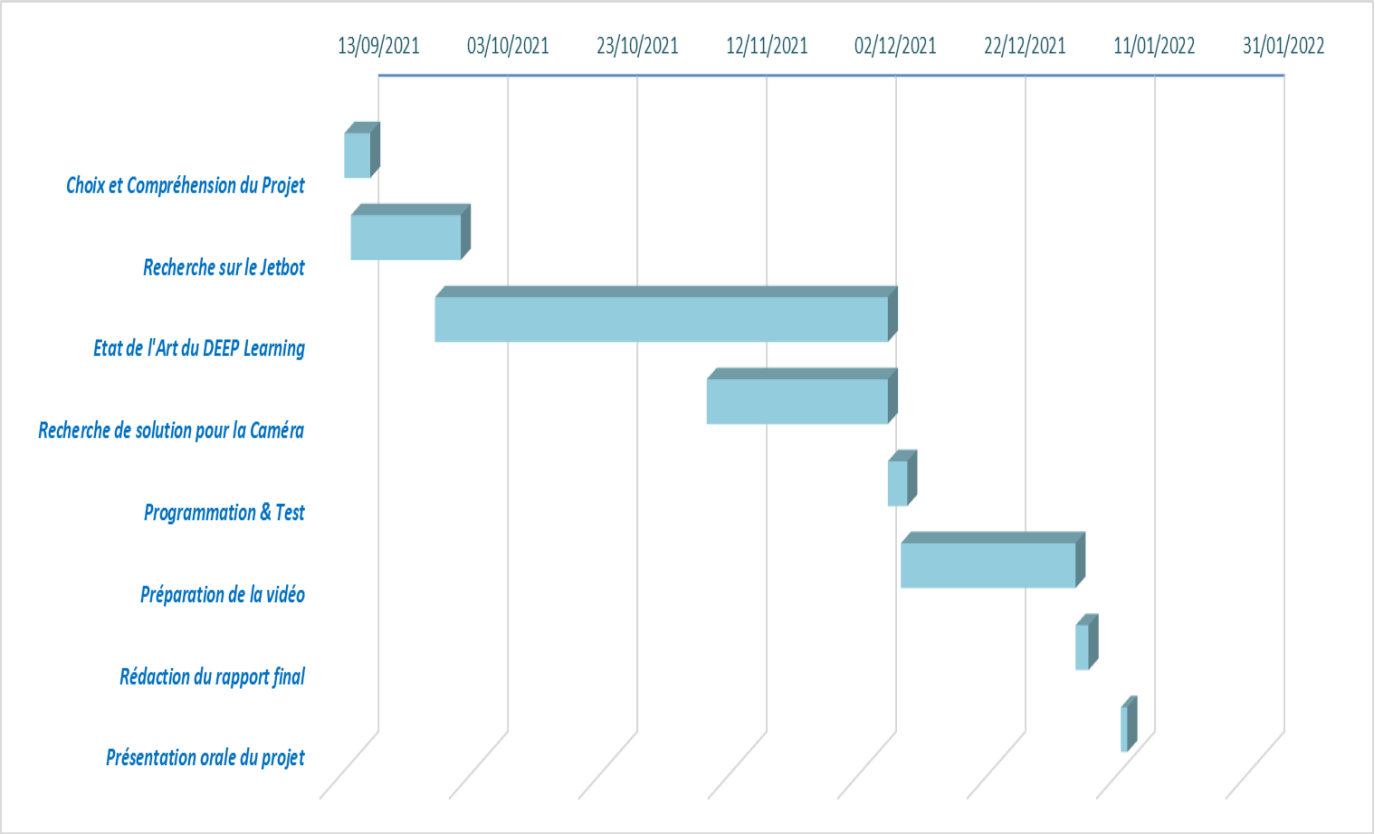
Le problème majeur durant le projet a été la caméra Raspberry pi. En effet, au bout de la 4ème semaine, la caméra ne fonctionnait plus, ce qui nous a empêché de continuer le projet puisque notre projet est basé sur la caméra. A cause de cet imprévu, nous avons pu continuer le projet que 1 semaine et demie avant la fin du projet.

Ensuite un autre problème moins pénalisant que la dernière a été la batterie. En effet, le Jetbot est un robot qui fonctionne à l'aide d'une batterie. Dans notre cas, nous avons eu le Jetbot avec une batterie non fonctionnelle et qu'avec le COVID, il était impossible de commander une batterie dans les temps. Pour le modèle de Suiveur de ligne, cela n'a pas été dérangeant que le Jetbot soit branché directement au secteur mais lorsque nous avons voulu essayer de placer le Jetbot dans un Labyrinthe, nous avons compris qu'il serait très difficile de le faire bouger dans un endroit étroit comme notre Labyrinthe.

D'autres problèmes, qui sont minimes et qui permettrait une optimisation de l'environnement, est la qualité de notre circuit de Suiveur de ligne. En effet, le circuit a été fait à partir de scotch ce qui nous donne un circuit avec une précision de ligne assez mauvais. De plus, à la suite du rangement de la maquette, on se retrouve très souvent avec une maquette pliée, ce qui bloque le Jetbot lorsqu'il est en fonctionnement puisque le Jetbot est très proche du sol.

VI. Diagramme de GANTT et Répartition des taches

A. Diagramme de GANTT



B. Répartition des tâches

Tout d'abord, nous avons commencé le projet chacun de son côté comme notre tuteur nous l'as demandé. Nous avons su qu'à la fin que nous devions faire la présentation et le rapport en binôme.

Malgré cela, nous avons su répartir les tâches. En effet, **Sourena** s'est occupé d'effectuer la recherche approfondie sur le Jetbot et Jetson nano circuit pour véhicules autonomes, le cours de machine Learning propose par Stanford University sur la Platform Coursera (réalisation de Neural Network et programmation python), le Hardware et toute la partie initialisation du robot. Et pour **Ali**, il s'agissait d'effectuer les recherches sur le DEEP Learning, les bibliothèques et les modèles de réseaux neuronaux. Toutefois, chacun d'entre nous a aussi approfondies ces connaissances en effectuant des recherches sur la partie du binôme.

VII. Amélioration future

Le Jetbot est un outil pédagogique qui permet de s'exercer dans le DEEP learning. Or nous connaissons la richesse de développement de ces dernières années du DEEP learning.

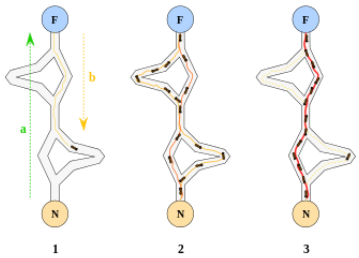
Pour la **première amélioration**, il serait intéressant d'intégrer, sur notre Suiveur de ligne et Evitement de collision, un détecteur d'objet qui permettra de reconnaitre instantanément le type d'objet qui se trouve devant lui (Humain, Tasse, etc…).

Figure 14 https://www.youtube.com/watch?v=2XMkPW_sIGg&ab_channel=NVIDIADeveloper

Malheureusement durant notre projet, nous n'avons pas pu l'intégrer puisque la mémoire était saturée et que l'installation de ce détecteur est couteuse en mémoire.

Pour l'utilisation de ce détecteur, il faut installer une librairie d'image qui correspond aux objets qu'on souhaite détecter.

Pour la **2ème amélioration**, il serait intéressant d'utiliser l'Evitement de collision dans un Labyrinthe et d'optimiser celui-ci avec un algorithme ACO. L'algorithme ACO est un algorithme qui s'inspire des colonies de fourmis et qui permet de trouver le chemin le plus rapide. En effet les fourmis ont la particularité de libérer de la phéromone qui permet aux fourmis de suivre le chemin car elle devient attractive. Ainsi plus les fourmis prennent un chemin particulier, plus ce chemin devient attractif.



Pour notre projet, il serait intéressant de laisser le robot tourner dans le labyrinthe et de remplacer la phéromone des fourmis par les images de caméra que nous allons enregistrer à temps régulier. Et ainsi, à partir d'une programmation, permettre au robot de retrouver un chemin plus rapide en le laissant découvrir le labyrinthe.

CONCLUSION

L'objectif de ce projet a été de se familiariser avec le Jetbot et son environnement et de comprendre le DEEP Larning.

Pour cela, nous avons effectué une recherche poussée sur le matériel qu'on nous a fourni pour comprendre les fonctionnalités du Jetbot et de s'apercevoir de ses limites. Ensuite nous nous sommes occupés de l'Etat de l'art du DEEP Learning pour comprendre et assimiler les méthodes nous permettant de profiter du Jetbot à son maximum de ces capacités.

Durant cet Etat de l'art, nous avons pu comprendre les différents types de réseaux puis les différentes bibliothèques, qu'on appelle Frameworks, qui nous permettent d'utiliser les différents réseaux, qui ont permis de choisir celui qui nous convient le mieux, c'est-à-dire **Pytorch**. Ensuite nous avons effectué les recherches sur les différents modèles de réseau neuronale pour nous permettre de choisir celui qui convient le mieux pour notre projet. Ainsi nous sommes partis sur **ResNet18**.

Malgré un petit souci de caméra qui nous a pénalisé durant plusieurs semaines, nous avons tout de même pu faire fonctionner le Jetbot sur un circuit lui permettant de **suivre une ligne** et **d'éviter les obstacles**.

Ce projet nous a appris plusieurs compétences clés comme l'esprit d'équipe, la capacité à effectuer des activités de recherche et de trouver l'information pertinente, l'autonomie, la capacité d'innover ainsi que la rigueur et l'organisation qui sont des compétences importantes pour un ingénieur dans le monde professionnel.

BIBLIOGRAPHIE

[1] Das, Biplab. "AAEON PICO-WHU4 an Alternative of PI With Intel 8th Gen SOC and 16GB RAM." Etechwall.com, 6 Apr. 2020, etechwall.com/2020/04/aaeon-pico-whu4-an-alternative-of-pi-with-intel-8th-gen-soc-and-16gb-ram/.

[2] Khnissi, Khaled, et al. "A Smart Mobile Robot Commands Predictor Using Recursive Neural Network." Robotics and Autonomous Systems, North-Holland, 12 June 2020 www.sciencedirect.com/science/article/abs/pii/S0921889020304334.

[3] He, Kaiming, et al. Deep Residual Learning for Image Recognition. www.cv-foundation.org/openaccess/content_cvpr_2016/papers/He_Deep_Residual_Learning_CVPR_2016_paper.pdf.

[4] Raspberry Pi. "Raspberry Pi 4 Model B Specifications." Raspberry Pi, www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/.

[5] Nvidia-Ai-Iot. "NVIDIA-AI-IOT/Jetbot." GitHub, github.com/NVIDIA-AI-IOT/jetbot/wiki/software-setup.

[6] "Getting Started With Jetson Nano Developer Kit." NVIDIA Developer, 12 Apr. 2020, developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit#write.

[7] "Abstract." NVIDIA Developer Documentation, docs.nvidia.com/deeplearning/frameworks/install-tf-jetson-platform/index.html.

*[8] "*Introducing the NVIDIA Jetson Nano*.", www.*Hackster.io

*[9] "*A Comparison of NVIDIA's Embedded AI Computing Platform Options*." Gumstix, Inc.* www.altium.com

*[10] "*Understanding the VGG19 Architecture." www.opengenus.org

[11] "Abstract." NVIDIA Developer Documentation, docs.nvidia.com/deeplearning/frameworks/install-tf-jetson-platform/index.html.

[12] "Real-time Face detection on Jetson Nano using OpenCV." Nvidia Jetson,
Real time Lane detection and Object detection using multiple neural networks Optimized with TensorRT - Jetson & Embedded Systems / Jetson Projects - NVIDIA Developer Forums

[13] OpenCV:" Face Detection using Haar Cascades."
docs.opencv.org/4.x/d7/d9f/tutorial_linux_install.html

[14] "How to Set Up a Camera for NVIDIA Jetson Nano." Automatic Addison

automaticaddison.com/how-to-set-up-a-camera-for-nvidia-jetson-nano

[15] Jetson Zoo, www.eLinux.org

[16] Hello AI World guide to deploying deep-learning inference networks and deep vision primitives with TensorRT and NVIDIA Jetson.
jetson-inference/aux-docker.md at master · dusty-nv/jetson-inference · GitHub

[17] ": Machine learning course by Stanford university" platform coursera
www.coursera.org

[18]": Machine Learning A-Z™: Hands-On Python & R In Data Science."
www.udemy.com

[19] "Neural Networks: Forward and Backpropagation" TJHSST Machine Learning Club, tjmachinelearning.com.

[20] " Machine learning : comprendre les réseaux de neurones." juripredis
www.juripredis.com/fr/blog/id-19-demystifier-le-machine-learning-partie-2-les-reseaux-de-neurones-artificiels.

[21] " Image Classification on ImageNet." /paperswithcode.com/sota/image-classification-on-imagenet

*[22] "*A *practical experiment for comparing LeNet, AlexNet, VGG and ResNet models with their advantages and disadvantages." tejasmohanayyar.medium.com/a-practical-experiment-for-comparing-lenet-alexnet-vgg-and-resnet-models-with-their-advantages-d932fb7c7d17*

*[23] "*A *Simple Guide to the Versions of the Inception Network." towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202*

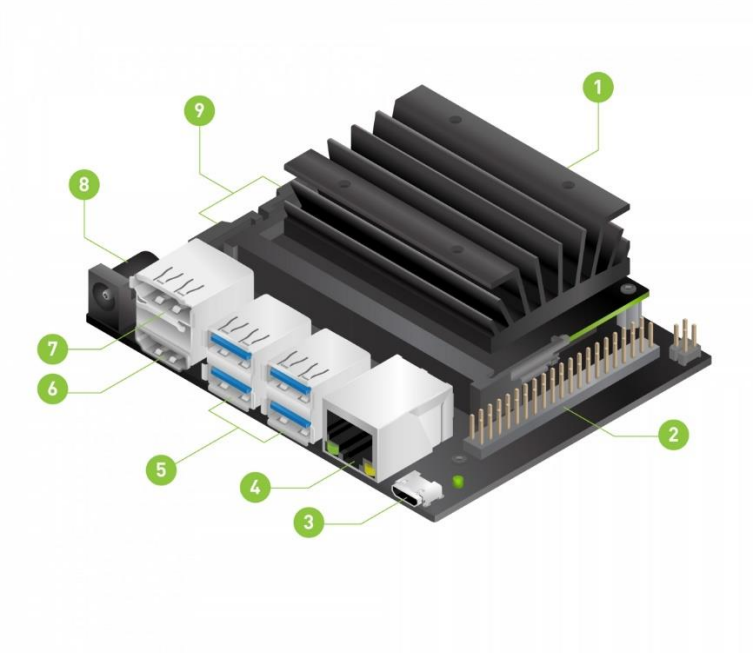
[24] " Difference between AlexNet, VGGNet, ResNet, and Inception" towardsdatascience.com/the-w3h-of-alexnet-vggnet-resnet-and-inception-7baaaecccc96

[25] " Jetson AI Fundamentals." JetBot Collision Avoidance www.youtube.com/watch?v=mwOVFXkc1WI&t=751s&ab_channel=NVIDIADeveloper

ANNEXES

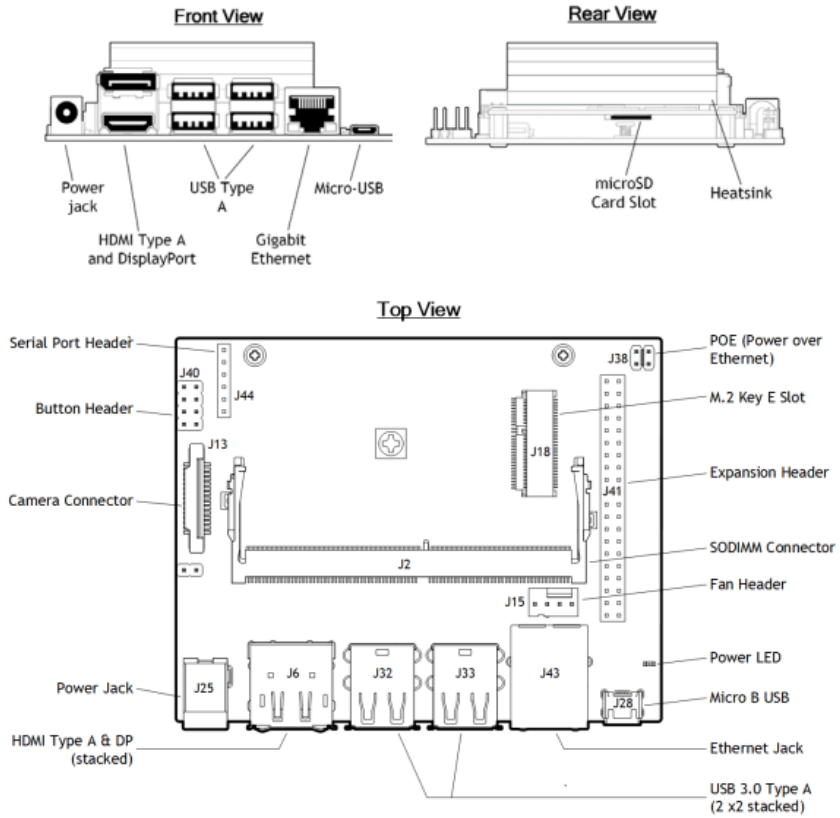
A. Appendix A: Technical details

Now, we have a figure that shows us the different parts of the Jetson nano board:



1. microSD card slot for main storage
2. 40-pin expansion header
3. Micro-USB port for 5V power input, or for Device Mode
4. Gigabit Ethernet port
5. USB 3.0 ports (x4)
6. HDMI output port
7. DisplayPort connector
8. DC Barrel jack for 5V power input
9. MIPI CSI-2 camera connectors

And from jetson nano user guide we have:



If we want to give a short detail in terms of components and electronics:

- **NVIDIA Maxwell graphics processor with 128 CUDA cores**
- **ARM Cortex-A57 quad-core processor**
- **RAM 4 GB LPDDR4**
- **16 GB eMMC 5.1 flash memory**
- **Connection:**
- **12-way camera connection (3 x 4 or 4 2 2) MIPI CSI-1.1 DPHY 18 (XNUMX Gbps)**
- **Gigabit Ethernet Network (RJ-45)**
- **HDMI 2.0 or DP 1.2 monitor connection eDP 1.4 | DSI (1 2 2) XNUMX simultaneously**
- **1/2/4 PCIE ports, 1 USB 3.0, 3 USB 2.0**
- **Additional I / O: 1 SDIO / 2 SPI / 4 I2C / 2 I2S / GPIO**
- **Size: 69.6mm x 45mm**
- **Consumption: 5-10 watts**
- **Linux operating system**

Side view of the Jetson Nano Developer Kit showing most of the available ports. (: NVIDIA)

B. Appendix B: Jetson nano setup

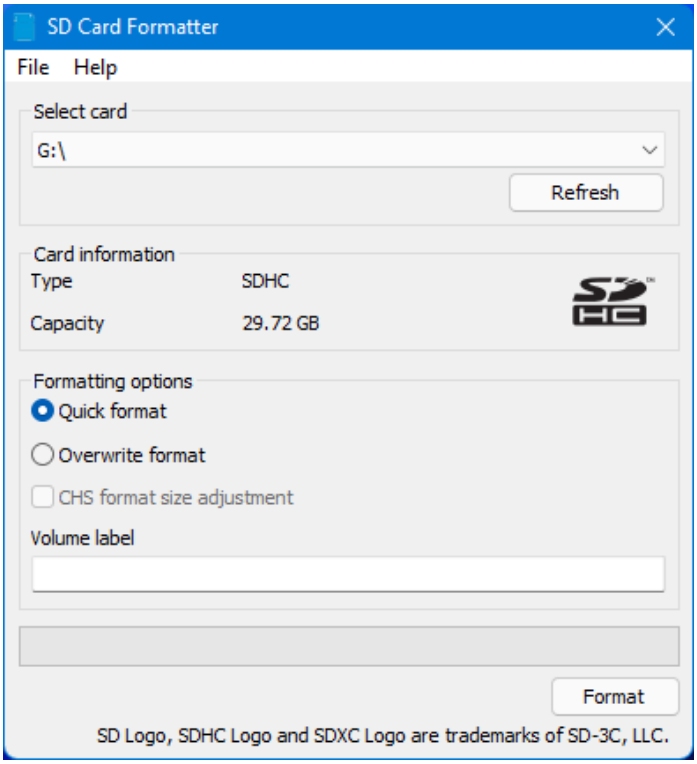
First: Items for Getting Started:

- **microSD Card:** The Jetson Nano Developer Kit uses a microSD card as a boot device and for main storage. It's important to have a card that's fast and large enough for your projects; the minimum recommended is a 32 GB UHS-1 card
- **Micro-USB Power Supply:** power supply that can deliver 5V⎓2A.

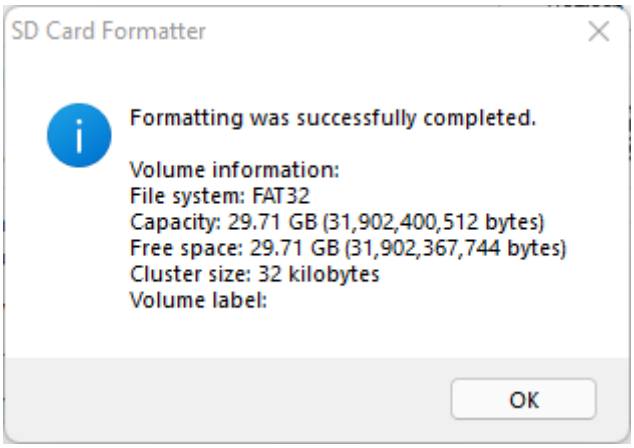
Second: Write Image to the microSD Card

We must download the Jetson Nano Developer Kit SD Card Image , and it is important to download the right version of it, because there is one for 4GB board version and another one for 2GB.

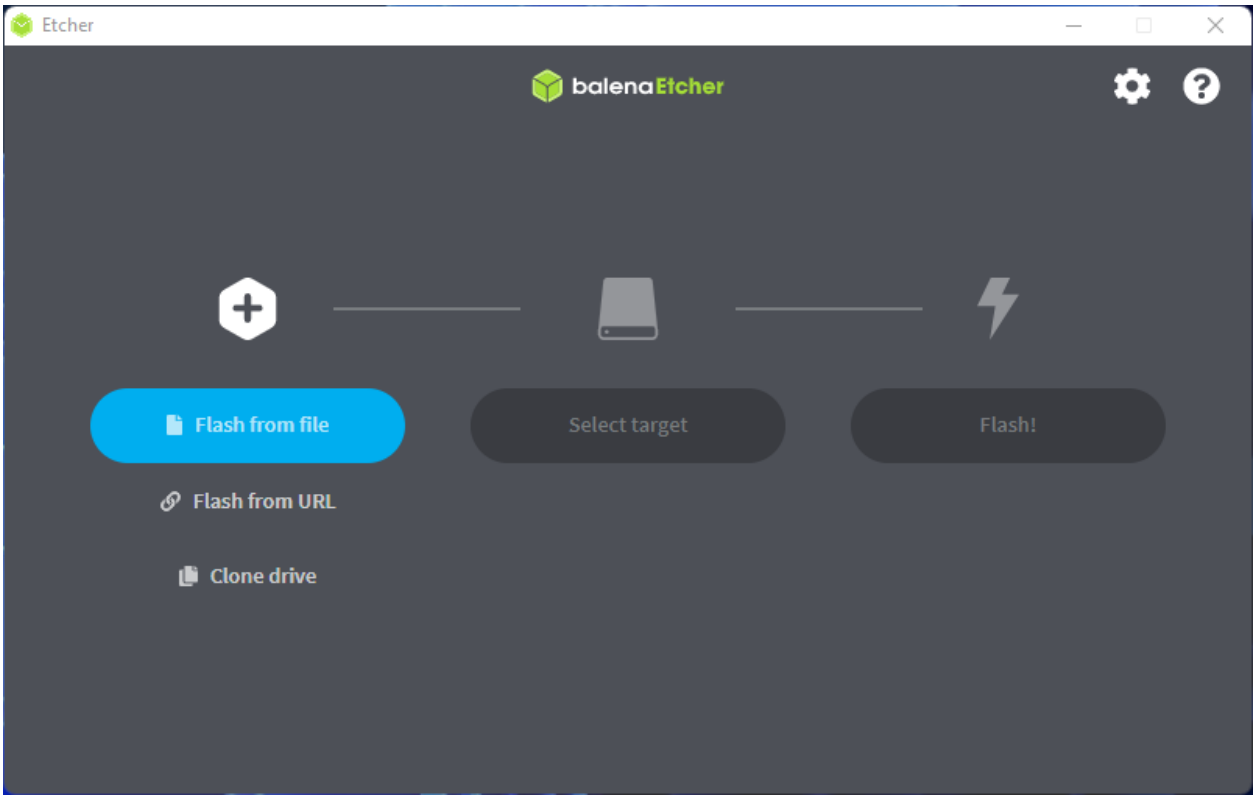
After that we format our SD card: I use the SD Memory Card Formatter



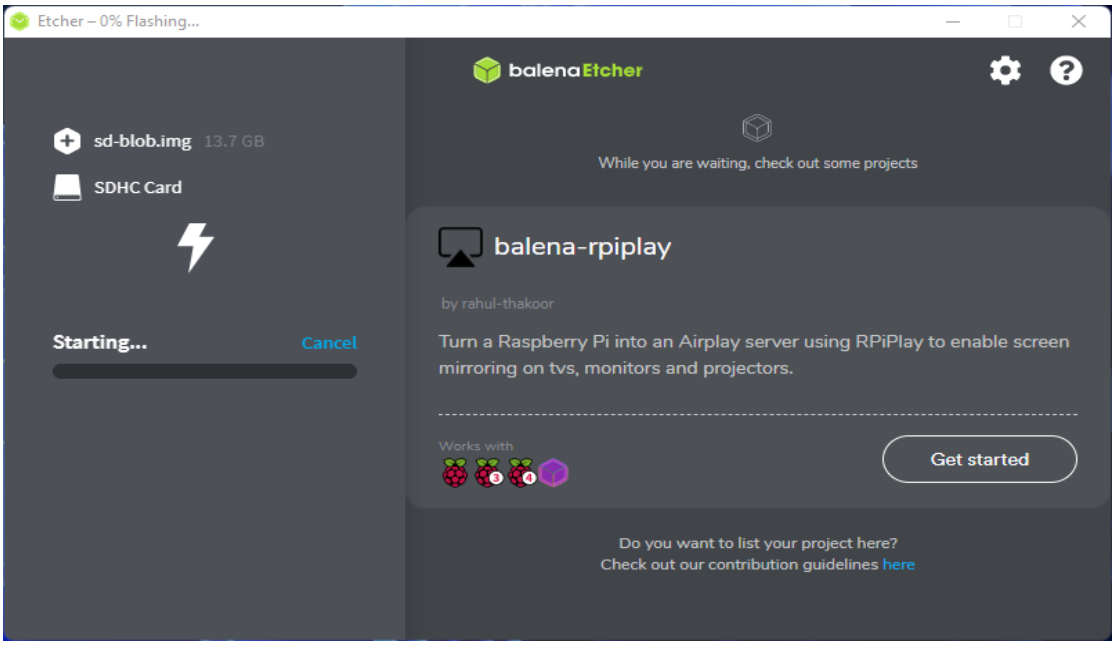
After that we obtained this window:



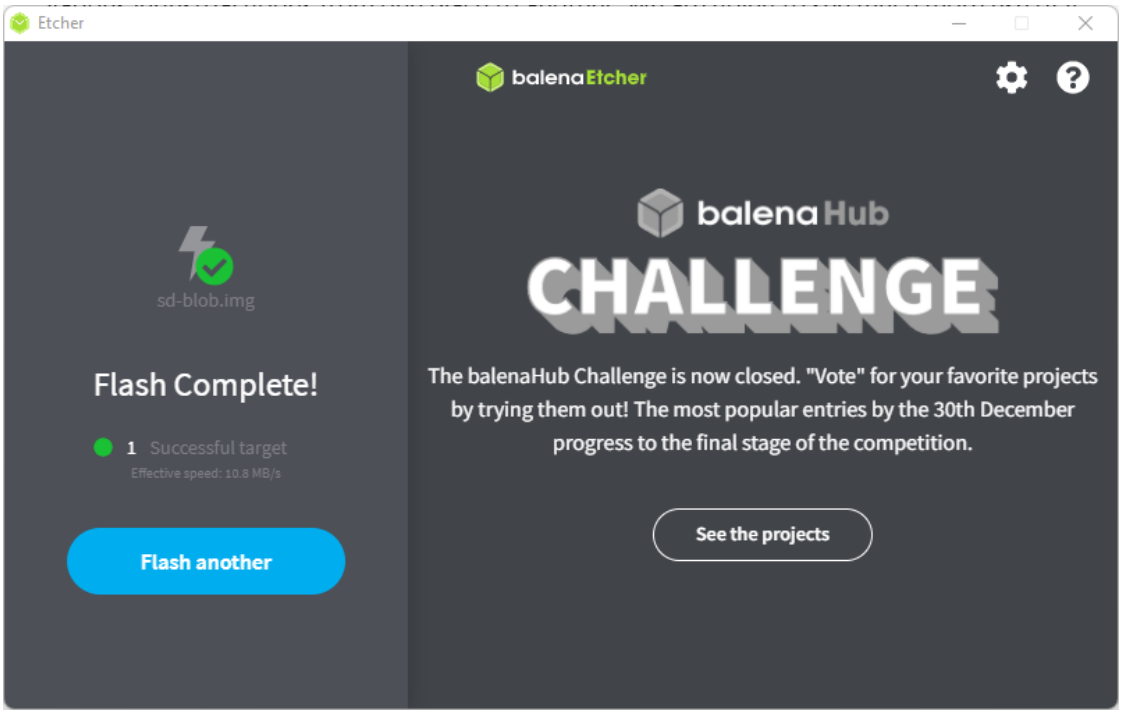
The next step is to write our image file that we already downloaded to our microSD, for this task we use Etcher:



After selecting our SD and Image file we have:



After a while, our SD card successfully flashed.



Now we have all requirements for first Boot.

Setup and First Boot:

There are two ways to interact with the developer kit:

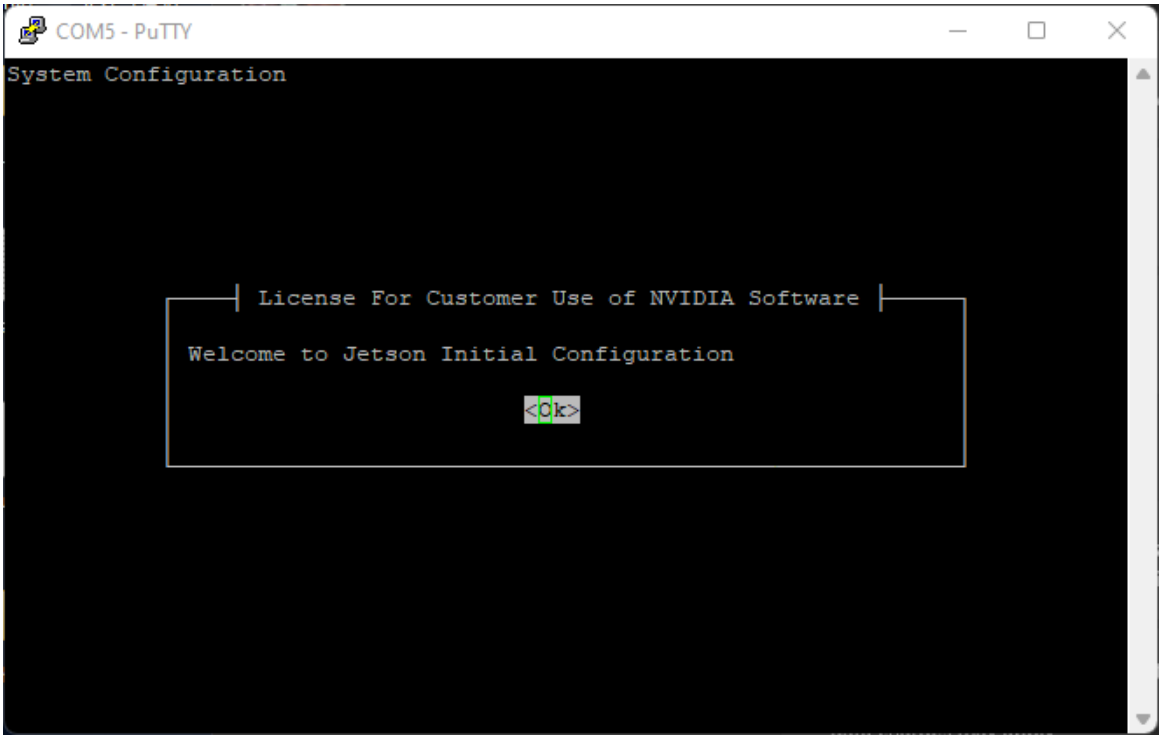
	Initial setup with display attached	Initial setup in headless mode
Monitor, keyboard and mouse	Required	Not required
Extra computer	Not required	Required
Power options	Either Micro-USB or DC power supply can be used	DC power supply is needed

Initial Setup Headless Mode:

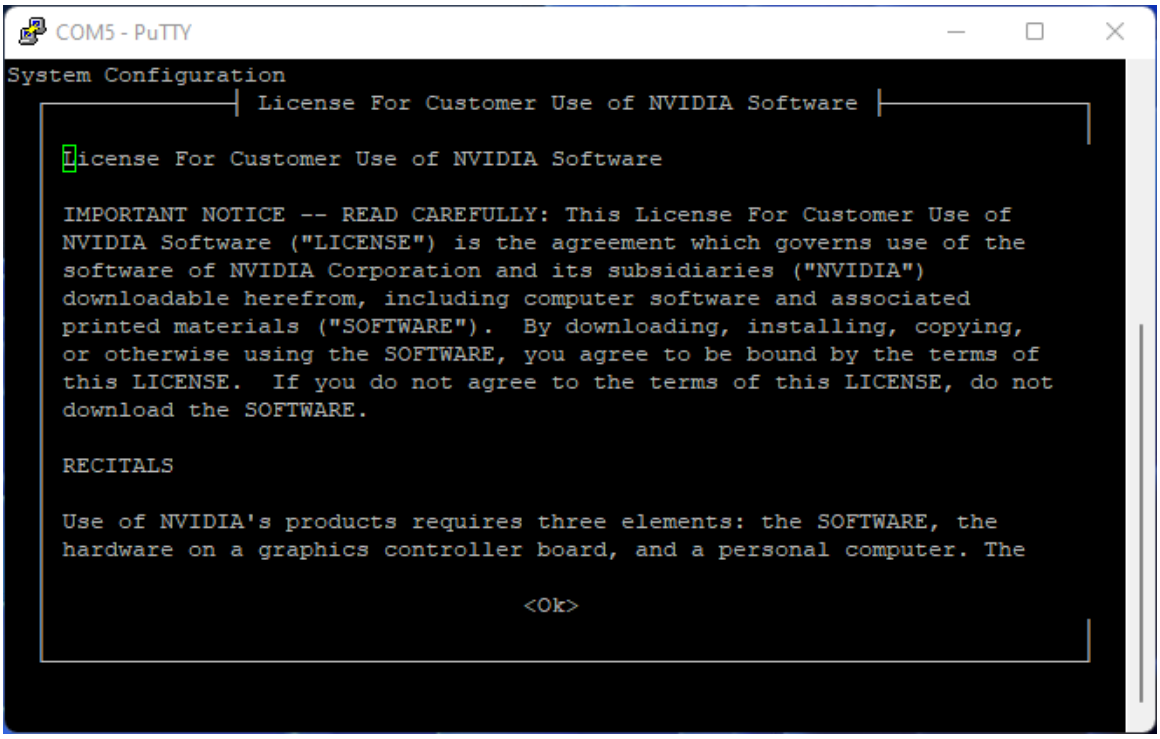
From NVIDIA: *"To complete setup when no display is attached to the developer kit, you'll need to connect the developer kit to another computer and then communicate with it via a terminal application (e.g., PuTTY) to handle the USB serial communication on that other computer."*

1. Insert the microSD
2. Connect your other computer to the developer kit's Micro-USB port.
3. Connect your USB-C power supply (5V⎓3A). The developer kit will power on automatically.
4. Connect your other computer to the developer kit's Micro-USB port.
5. Allow 1 minute for the developer kit to boot.

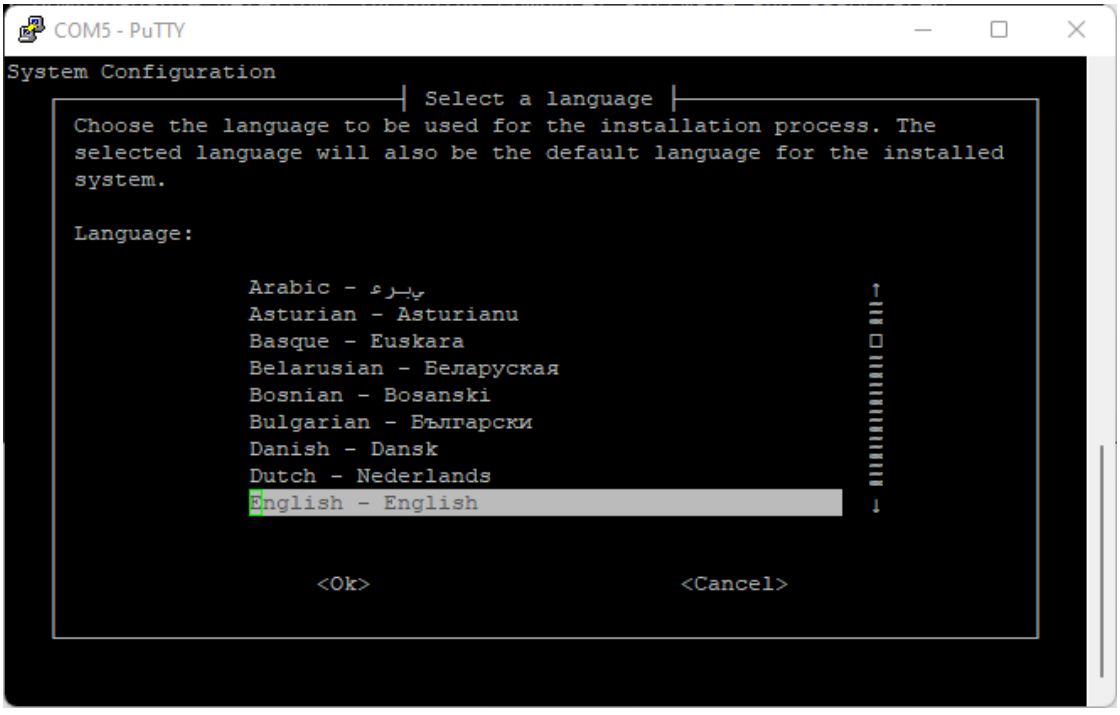
After all, we use putty in windows (screen in Linux and mac) for connection over USB driver.



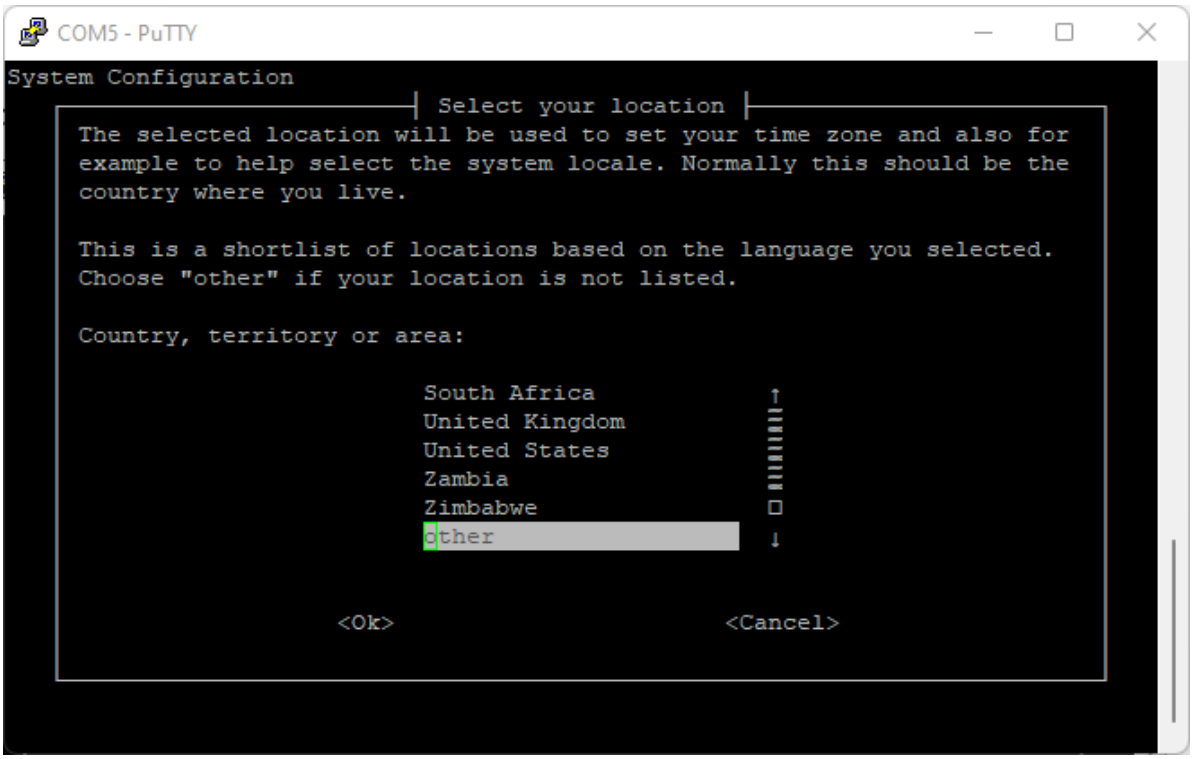
We can see that Linux works correctly, we pressed ok.



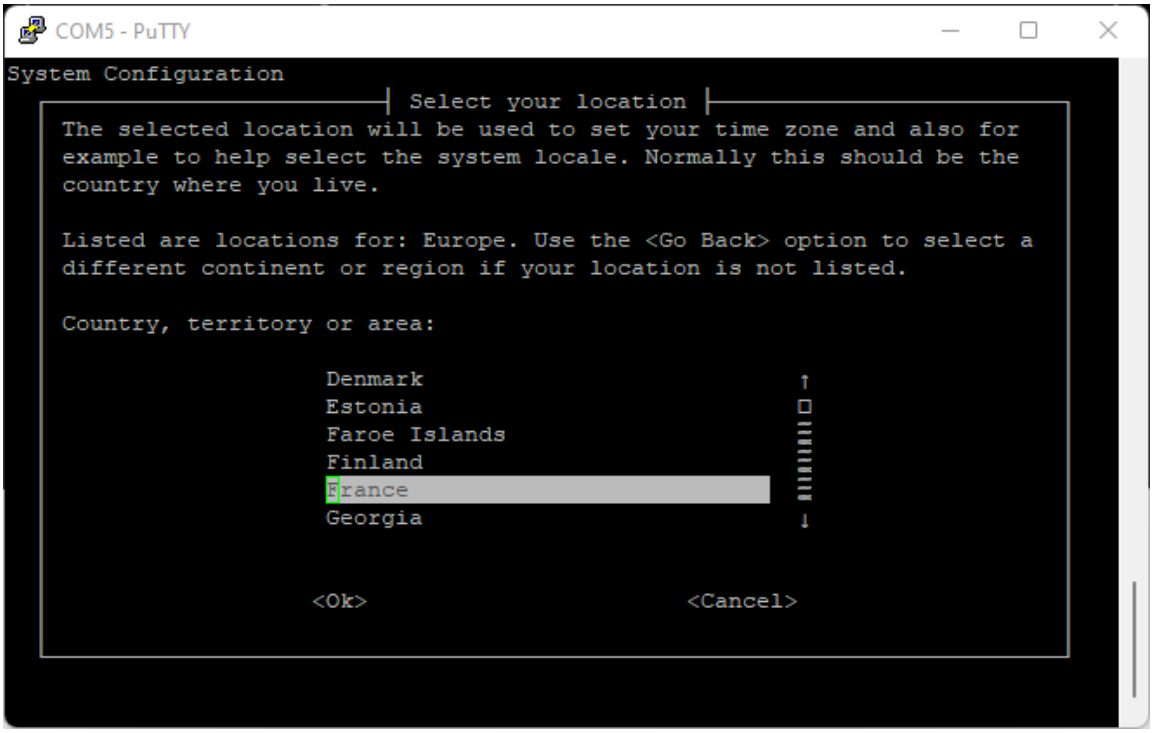
Now we have to choose language:



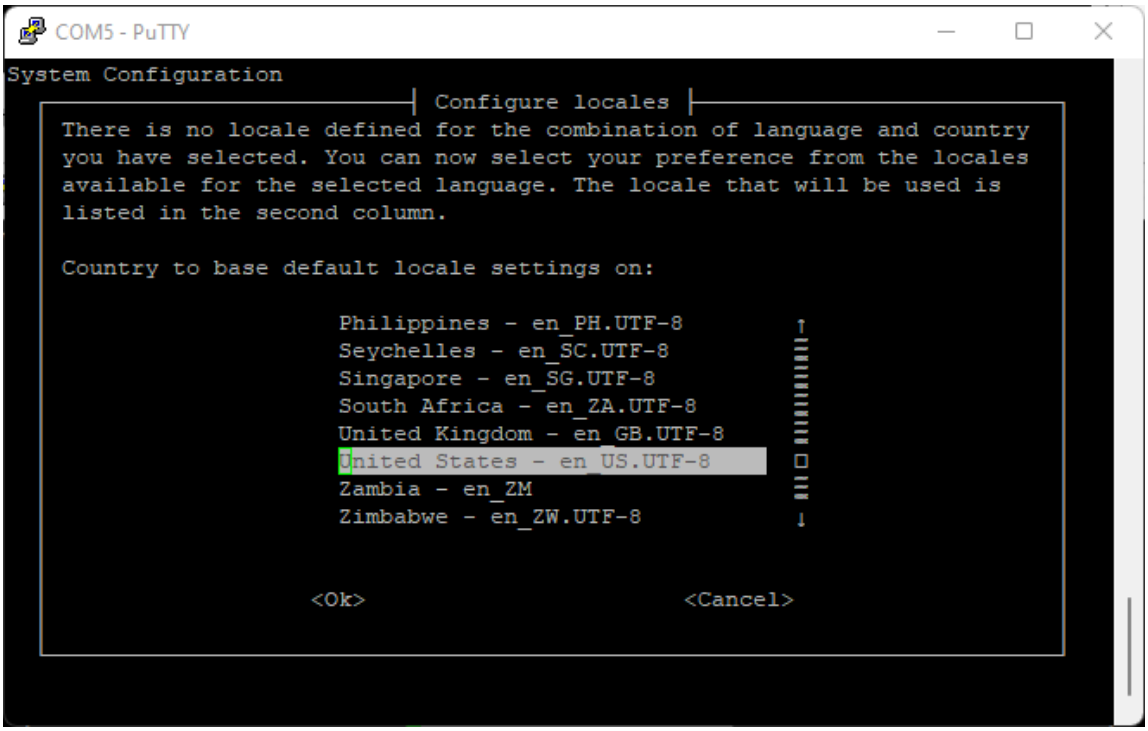
After that we must choose the territory , but France is not in the list , so I choose others .



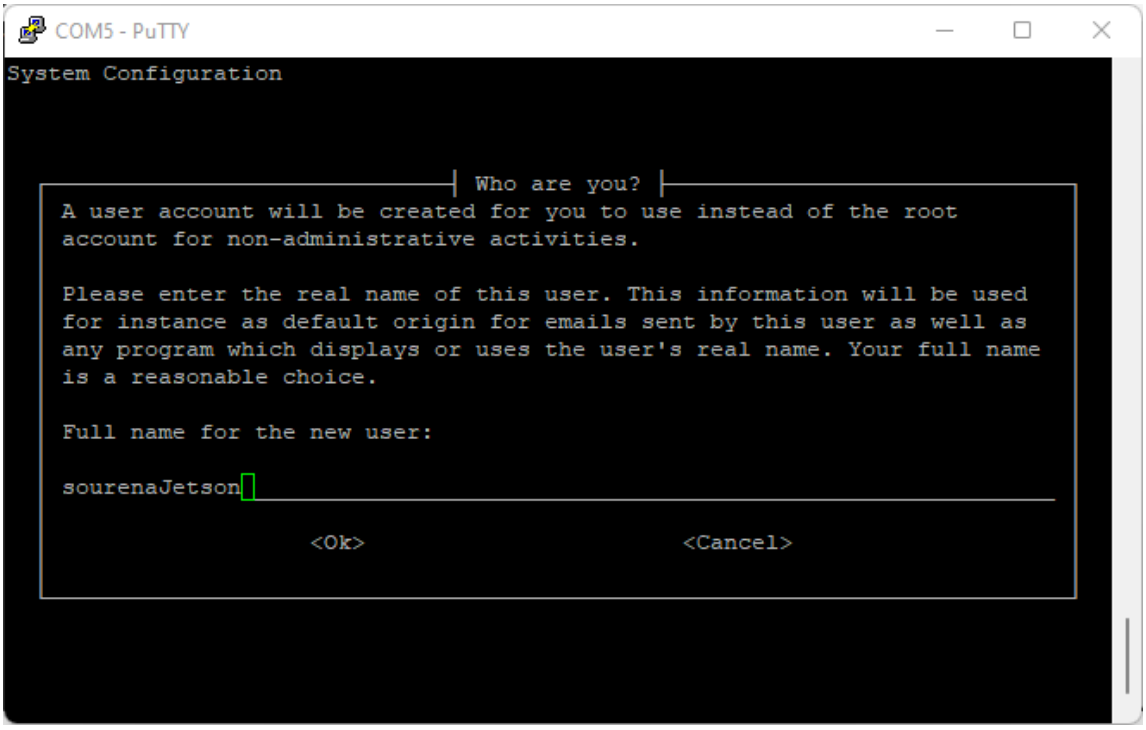
Therefor we must select the region: Europe plus we can find the France.



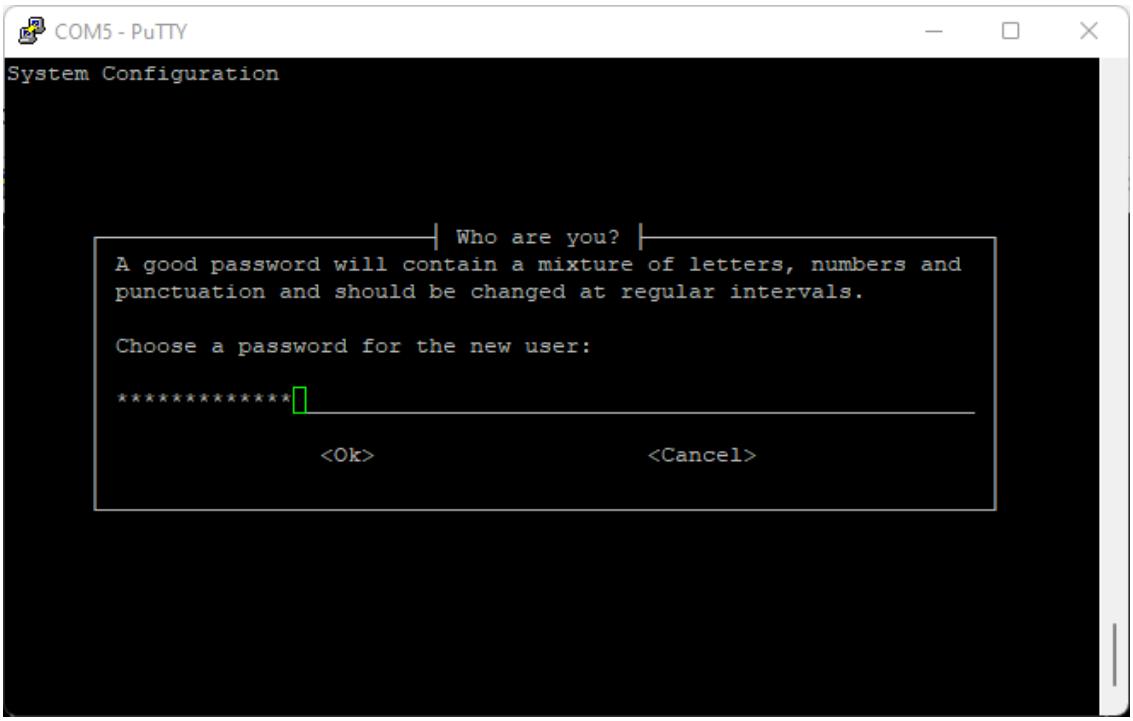
Now very simple , we choose the UTF8 from USA as default setting as common standard.



Now it is time to choose the user's name of our device:



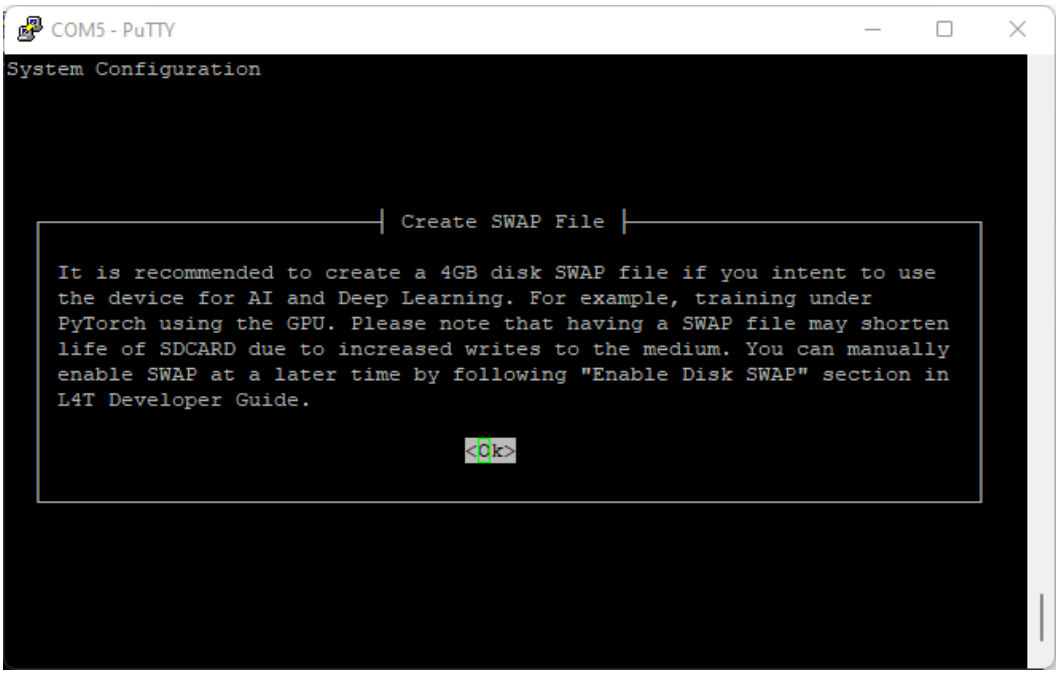
After that password:



After that we have: Partition size

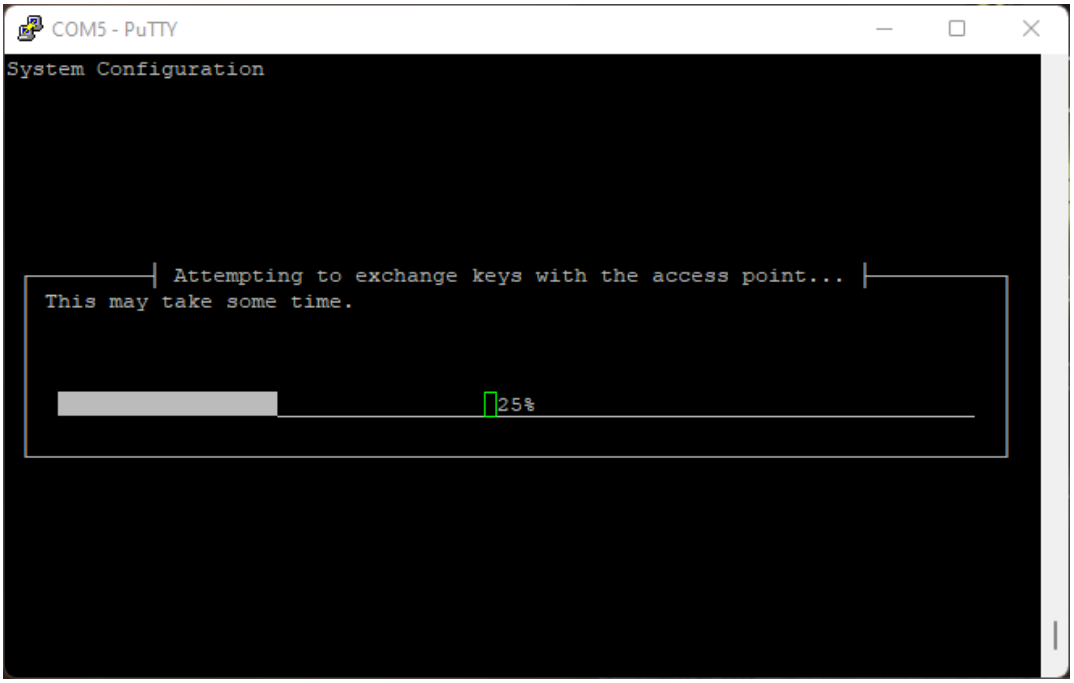
We choose the maximum size value.

For the next step we better to create a SWAP file for Deep Learning

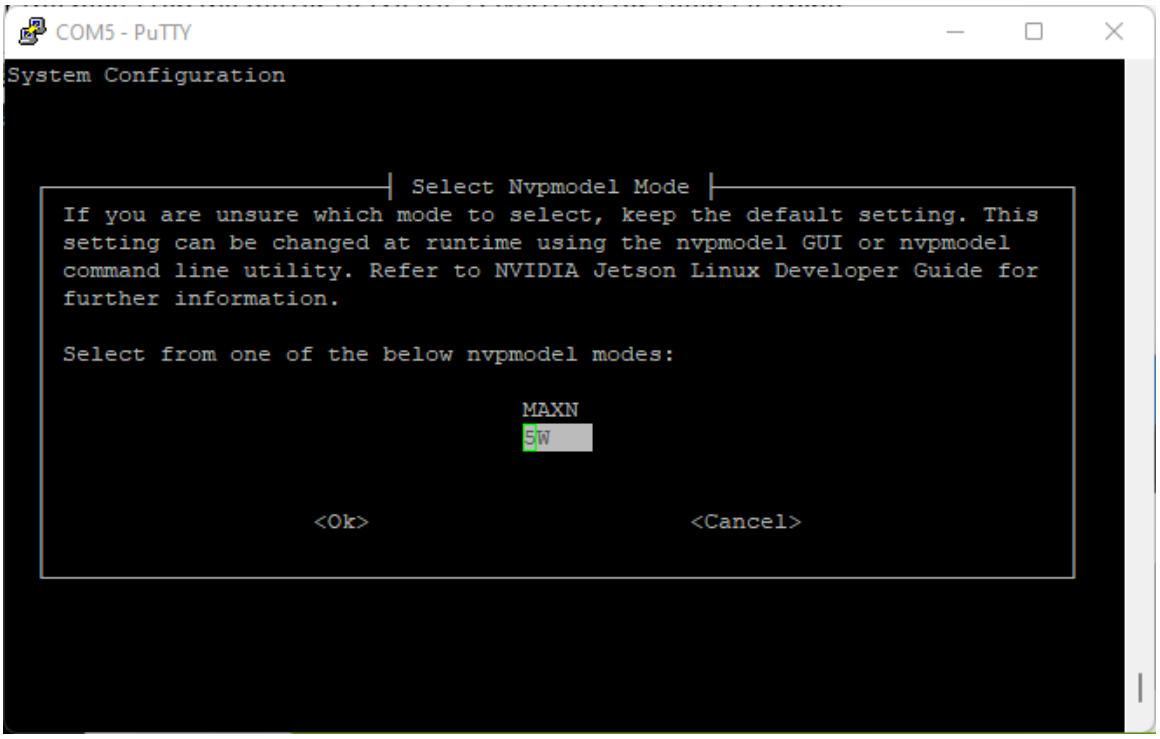


We continue to the end of initialization setup for our first Boot, now after 2minutes:

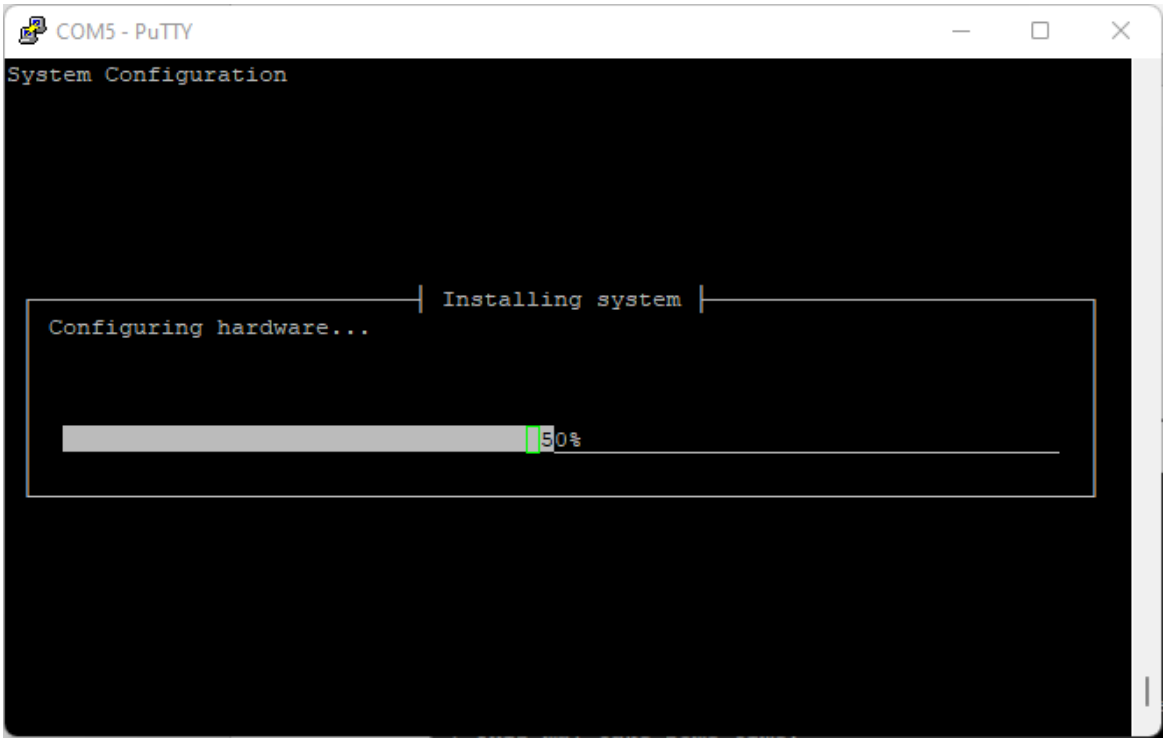
Connecting to internet:Network config



Another issue is decided between the 5W and MAXN, here we choose 5W because we always use the UCB C adaptor or micro-USB so we don't need such power (MAXN is 10W) and according to the NVIDIA, if we choose badly, it causes several problems or lags later.

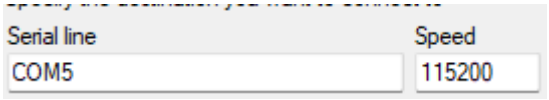


Installing process:

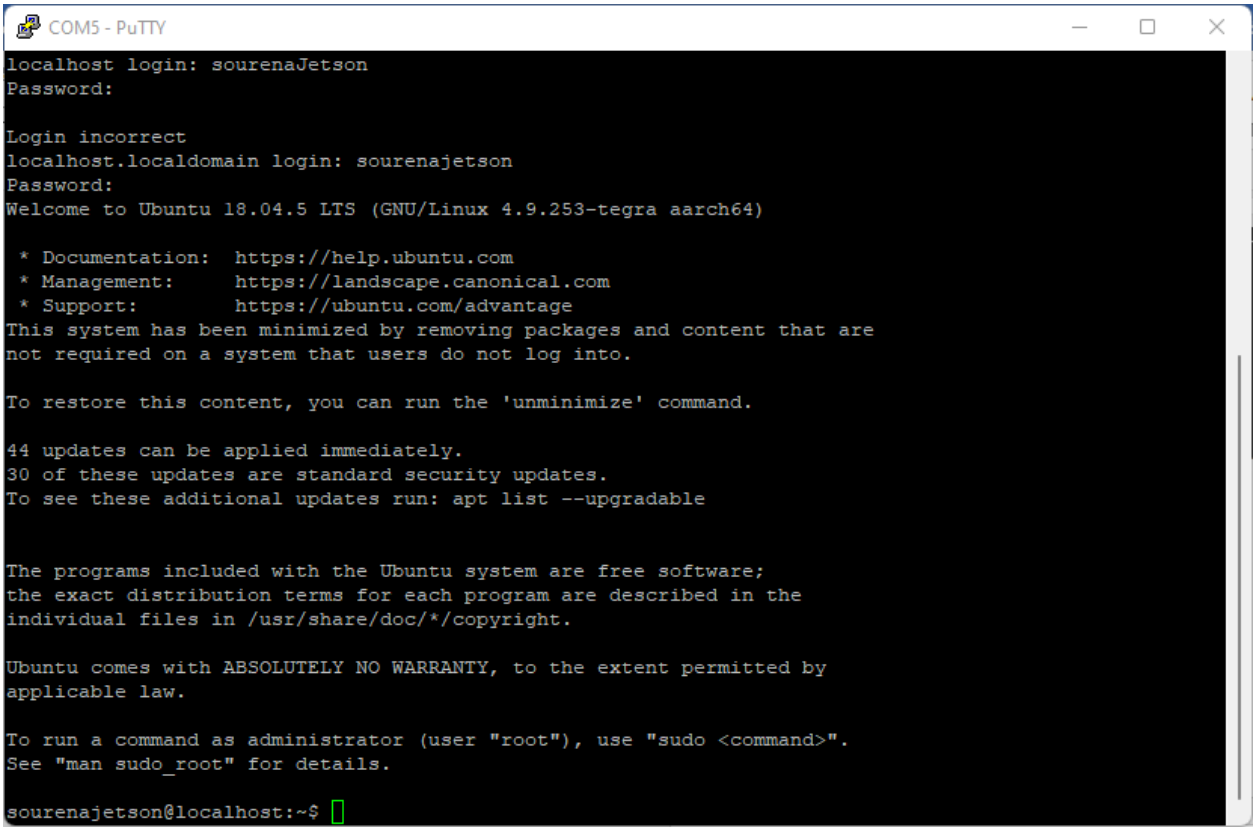


After this, installing process is done but we can see that we have a fatal error in putty, so we must re Run our putty:

We set putty in 115200 not 9600. According to Jetson nano datasheet:



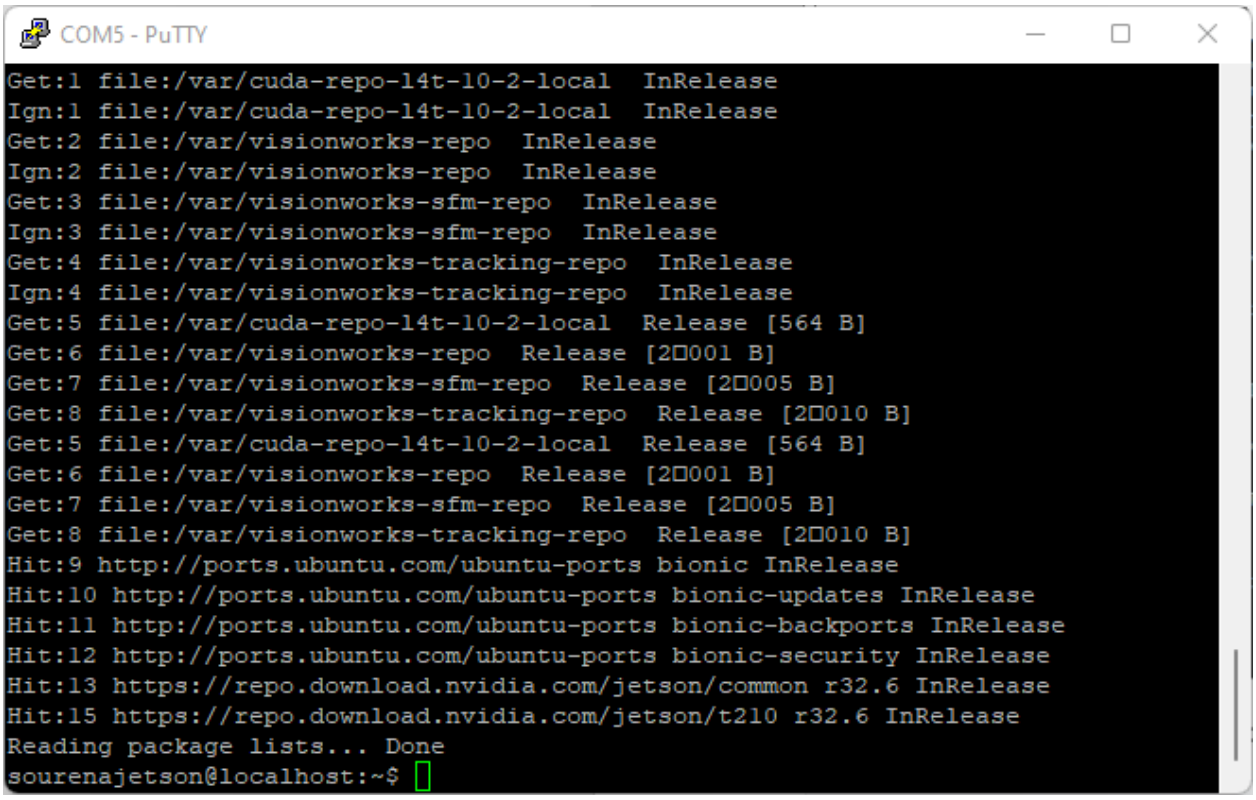
And finally, our system Boost is done.



Now we update the system like: we entered these commands

```
sudo apt-get update
sudo apt-get upgrade
```

and we got:



And we reboot the system after the update:

```
sudo reboot
```

C. Appendix C : Jetson nano : Configure VNC server

We do not have a monitor or mouse and keyboard to connect with jetson nano, **so how should we connect to the board?**

One way is using a free SSH and Telnet client like putty on windows, bud if we want to have a graphical view, we must use VNC server.

Virtual **N**etwork **C**omputing is a connection system that allows you to use your keyboard and mouse to interact with a graphical desktop environment on a remote server.

on **Ubuntu 18.04** on the Jetson Nano platform, we use **Xfce** as the desktop environment:

1. *Connect to the putty for working we the linux inside of the board*
2. *Install xfce4:*

```
sudo apt install xfce4 xfce4-goodies
```

3. *Install TightVNC server:*

```
sudo apt install tightvncserver
```

4. *Run VNC server once to create a configuration file:*

```
vncserver
```

5. *Kill the VNC server:*

```
vncserver -kill :1
```

6. *Edit the configuration file:*

```
#!/bin/bash
unset SESSION_MANAGER
unset DBUS_SESSION_BUS_ADDRESS
xrdb $HOME/.Xresources
startxfce4 &
```

7. *Restart VNC server:*

```
vncserver :1 -geometry 1280x720
```

8. *Create a file:*

```
sudo vim /etc/systemd/system/vncserver@.service
```

9. *Adding code to the file:*

```
[Unit]
Description=Start TightVNC server at startup
After=syslog.target network.target

[Service]
Type=forking
User=!!!user_name!!!
Group=!!!user_name!!!
WorkingDirectory=/home/!!!user_name!!!

PIDFile=/home/!!!user_name!!!/.vnc/%H:%i.pid
ExecStartPre=-/usr/bin/vncserver -kill :%i > /dev/null 2>&1
ExecStart=/usr/bin/vncserver -depth 24 -geometry 1280x720 :%i
ExecStop=/usr/bin/vncserver -kill :%i

[Install]
WantedBy=multi-user.target
```

We must change the username with our actual username .

10. Make the system aware of the new file:

```
sudo systemctl daemon-reload
```

11. Enable it:

```
sudo systemctl enable vncserver@1.service
```

12. Stop the current VNC server if it is running:

```
vncserver -kill :1
```

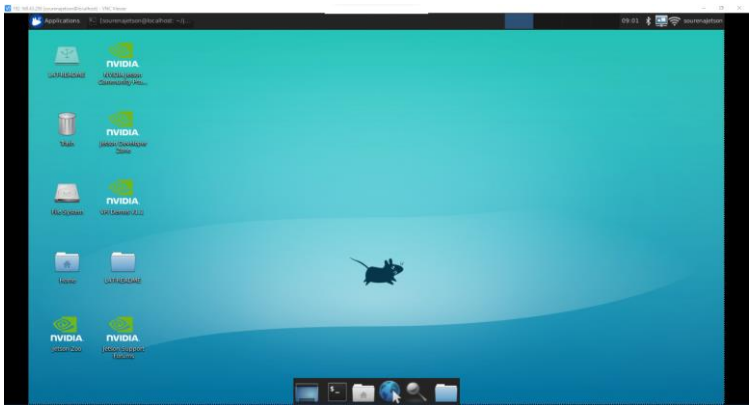
13. Start the system server:

```
sudo systemctl start vncserver@1
```

14. Verify the status:

```
sudo systemctl status vncserver@1
```

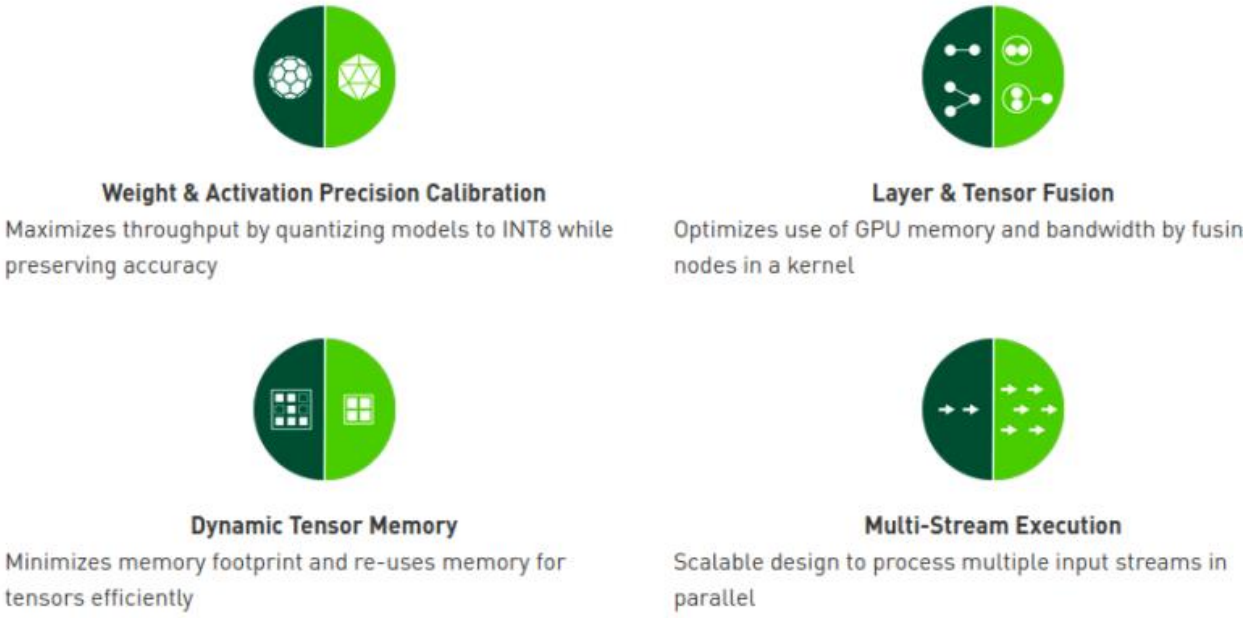
Demo:



D. Appendix D: Introduction to TensorRT on Jetson Nano

The Nvidia JetPack has in-built support for TensorRT (a deep learning inference runtime used to boost CNNs with high speed and low memory performance), cuDNN (CUDA-powered deep learning library), OpenCV, and other developer tools.

TensorRT SDK is provided by Nvidia for high-performance deep learning inference. It has an inference optimizer that runs deep learning models with low latency even in real-time situations.



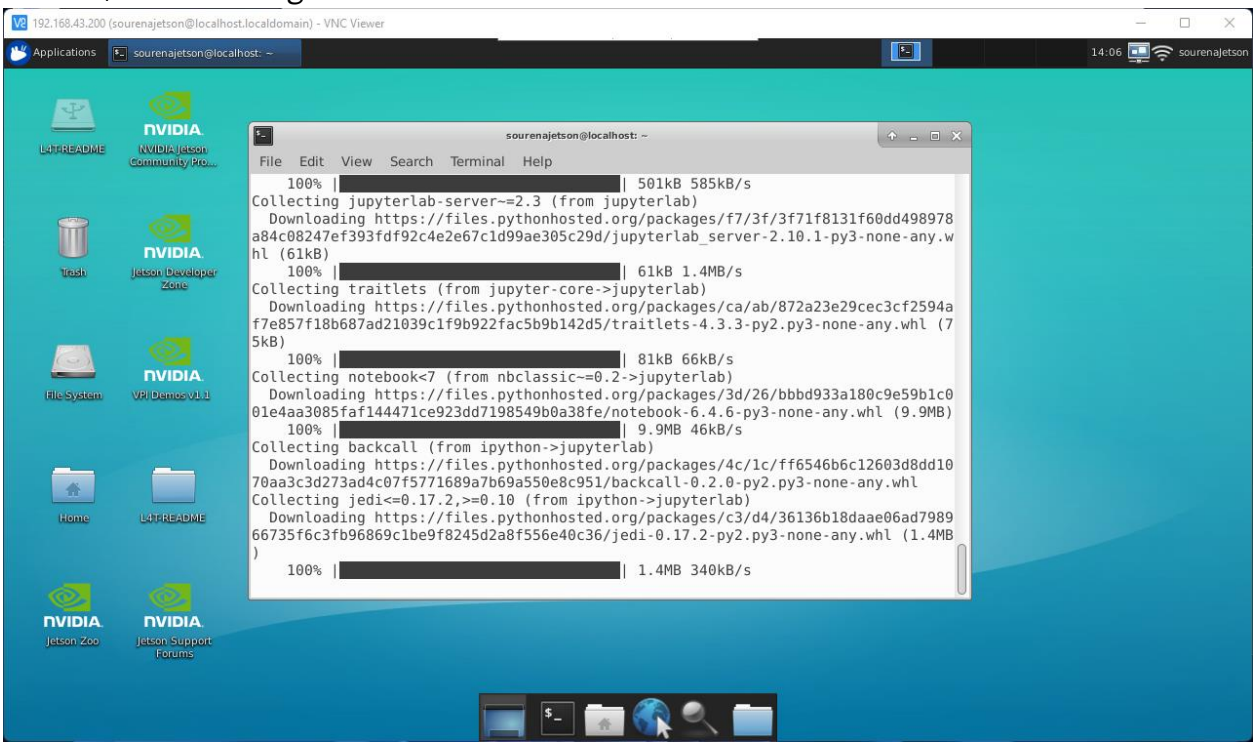
TensorRT optimizations. Image Source: TensorRT website

E. Appendix E: Installing Pre-trained Models on our Jetson nano

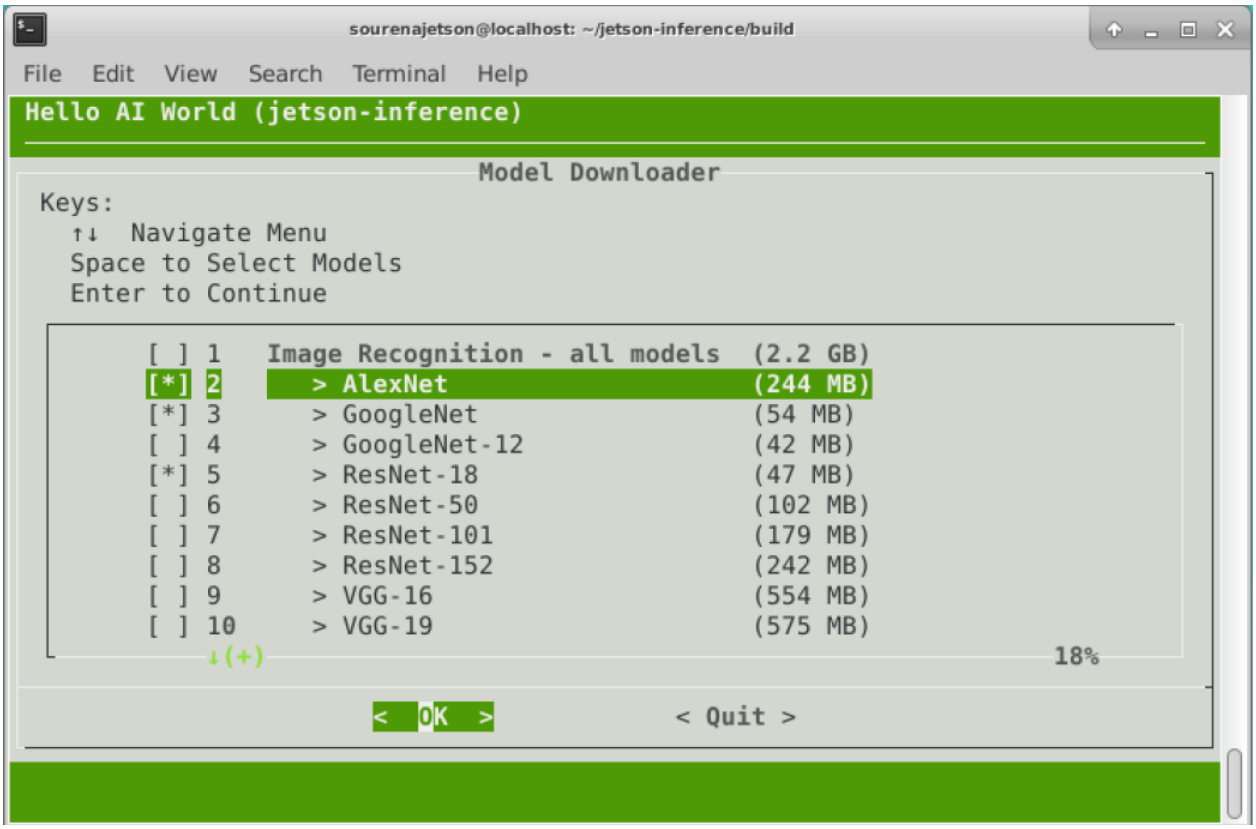
Provided with the repo is a library of TensorRT-accelerated deep learning networks for image recognition, object detection with localization (i.e. bounding boxes), and semantic segmentation. This inferencing library (libjetson-inference) is intended to be run on the Jetson, and includes support for both C++ and Python.

We can see the code for cloning to the NVIDIA developer repository in git hub and download that for different AI purposes.

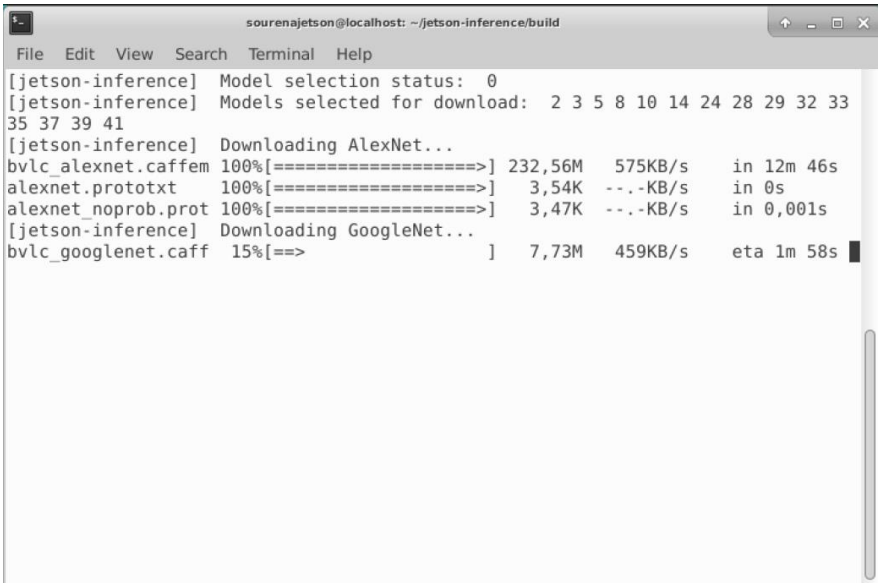
- $ sudo apt-get update
- $ sudo apt-get install git cmake libpython3-dev python3-numpy
- $ git clone --recursive https://github.com/dusty-nv/jetson-inference
- $ cd jetson-inference
- $ mkdir build
- $ cd build
- $ cmake ../
- $ make -j$(nproc)
- $ sudo make install
- $ sudo ldconfig



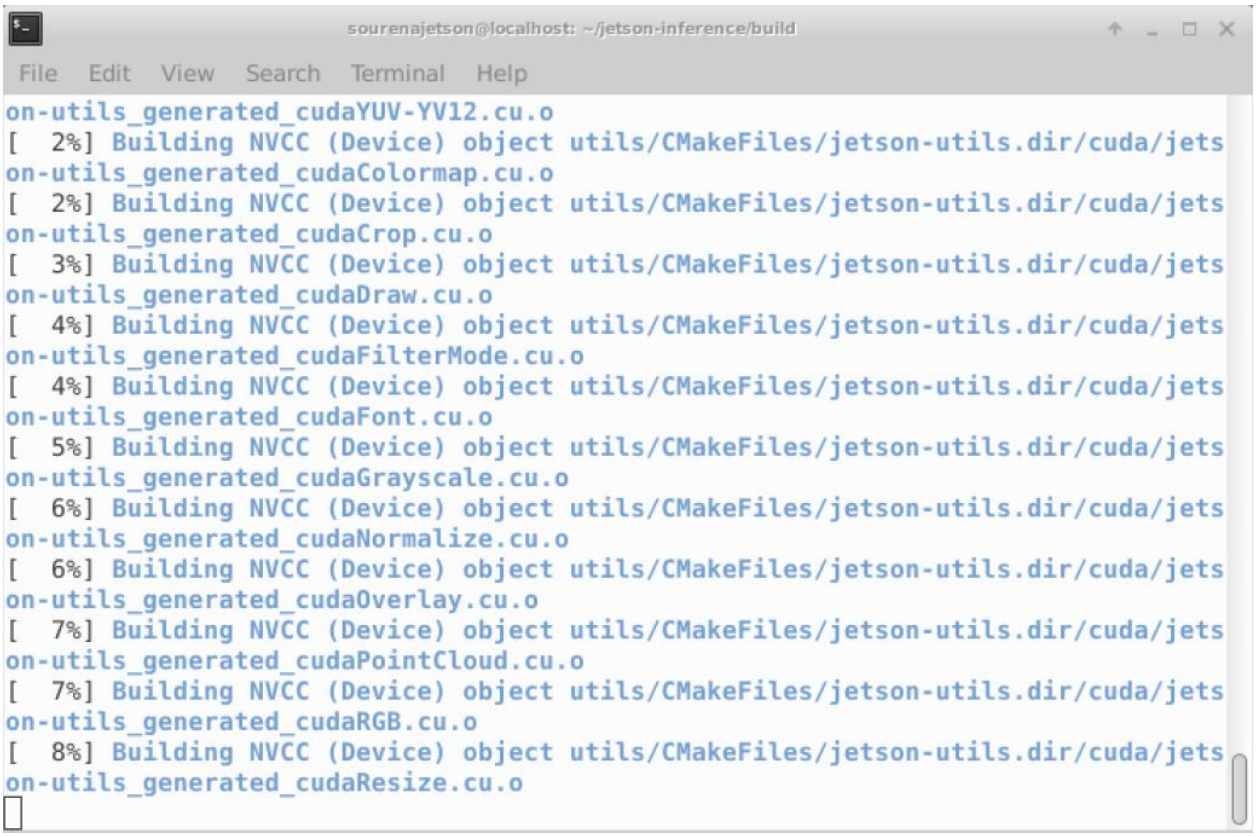
After that in the process we have to choose the pre-trained models that we want to download on the Jetson Nano board. Like :



And we downloaded them:



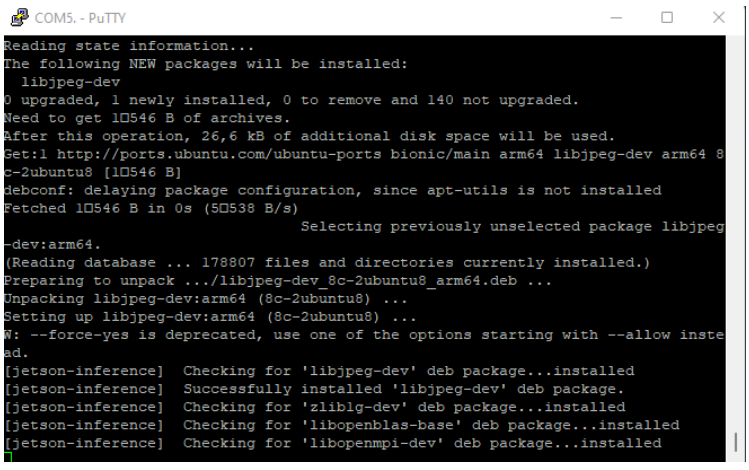
In the end we can install all the files in to our board:



The project will be built to jetson-inference/build/aarch64, with the following directory structure:

- |-build
- \aarch64
- \bin where the sample binaries are built to
- \networks where the network models are stored
- \images where the test images are stored
- \include where the headers reside
- \lib where the libraries are built to

In the end we install library Pytorch :



The code explications in details :

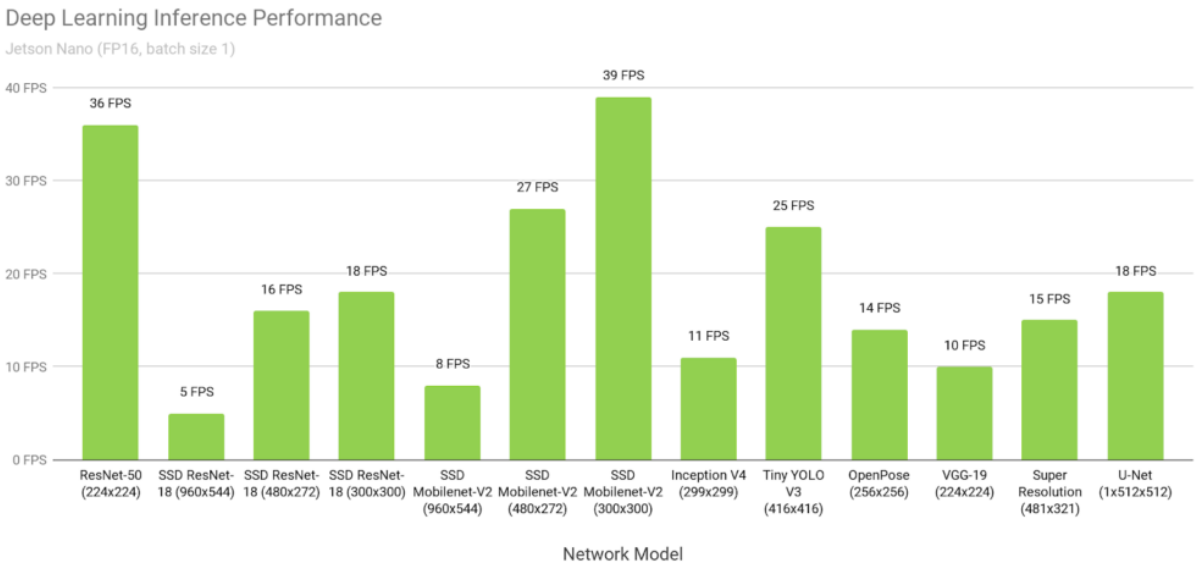
Below is the output of the Python imageNet object from the jetson.inference package:

```
jetson.inference.imageNet = class imageNet(tensorNet)
 |  Image Recognition DNN - classifies an image
 |
 |  __init__(...)
 |       Loads an image recognition model.
 |
 |       Parameters:
 |         network (string) -- name of a built-in network to use
 |                             values can be:  'alexnet', 'googlenet', 'googlenet-
12', 'resnet-18`, ect.
 |                             the default is 'googlenet'
 |
 |         argv (strings) -- command line arguments passed to imageNet,
 |                           for loading a custom model or custom settings
 |
 |  Classify(...)
 |      Classify an RGBA image and return the object's class and confidence.
 |
 |      Parameters:
 |        image  (capsule) -- CUDA memory capsule
 |        width  (int) -- width of the image (in pixels)
 |        height (int) -- height of the image (in pixels)
 |
 |      Returns:
 |        (int, float) -- tuple containing the object's class index and confidence
 |
 |  GetClassDesc(...)
 |      Return the class description for the given object class.
 |
 |      Parameters:
 |        (int) -- index of the class, between [0, GetNumClasses()]
 |
 |      Returns:
 |        (string) -- the text description of the object class
 |
 |  GetNumClasses(...)
 |      Return the number of object classes that this network model is able to
classify.
 |
 |      Parameters:  (none)
 |
 |      Returns:
 |        (int) -- number of object classes that the model supports
```

While previously we manually created a GStreamer pipeline to interact with the video stream from the Raspberry Pi camera, jetson.utils already provides a pre-configured pipeline. It also provides a display handle. Finally, the code captures every image in the video stream and runs the detection algorithm and then draws translucent bounding boxes around the objects.

F. Appendix F: Benchmarks: interfaces

Jetson Nano's flexible software and full framework support, memory capacity, and unified memory subsystem, make it able to run a myriad of different networks up to full HD resolution, including variable batch sizes on multiple sensor streams concurrently. These benchmarks represent a sampling of popular network.

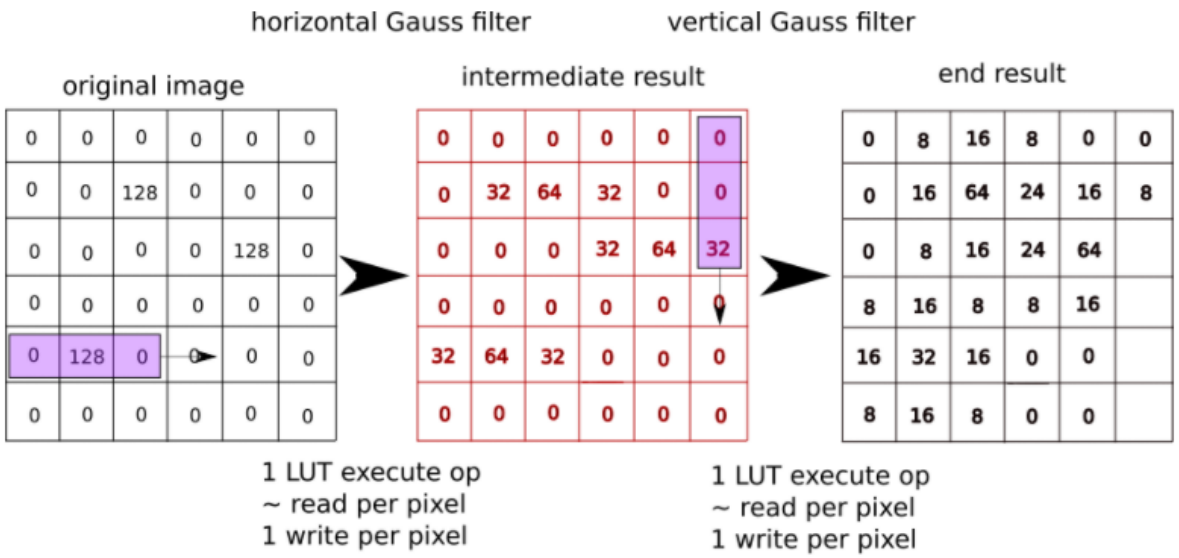


G. Appendix G: Lane Recognition goal in the Future: Autonomous vehicles

Autonomous vehicles are slowly becoming an important part of the automotive industry. Many believe fully autonomous vehicles will soon be driving alongside humans, and technology companies are in a race to deploy fully autonomous vehicles.

The first step of our lane detection system is being able to read live images from the camera:

1. First installing the library for our first step: camera detection:
   - **Pip3 install nanocamera**
   - **Import nanocamera as nano**
   - **Camera = nano.Camera(flip=0,width=1280 ,height =800, fps=30)**
   - **Frame = camera.read()**
2. We getting the live image from the code above .
3. We must remove the noise Blur to get the smoother and better image and remove unwanted noise like a virtual filter , hopefully python has many powerful libraries in the AI aspect so we have : This involves calculating each pixel's value as a weighted average of the surrounding pixels. We can apply this we the code below :



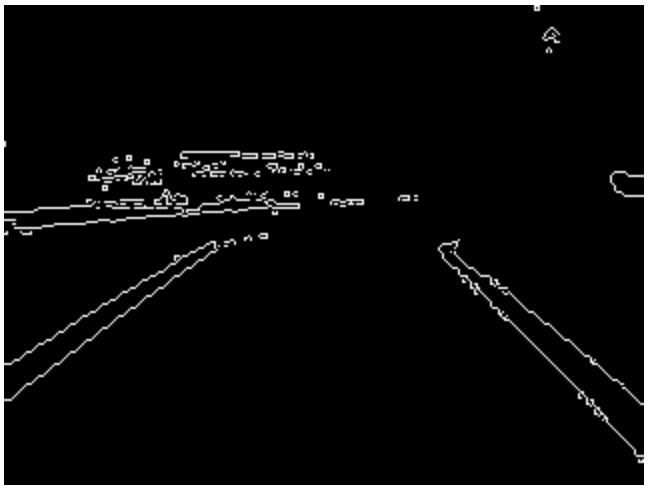
- 3 bit depth
- comparison with standard moving window algorithm:
  exact needs more read ops (3 per pixel, but better locality)
  and 7 additions as well as 3 FPU multiplications

Gaussian blur algorithm for image smoothing. [Image source]

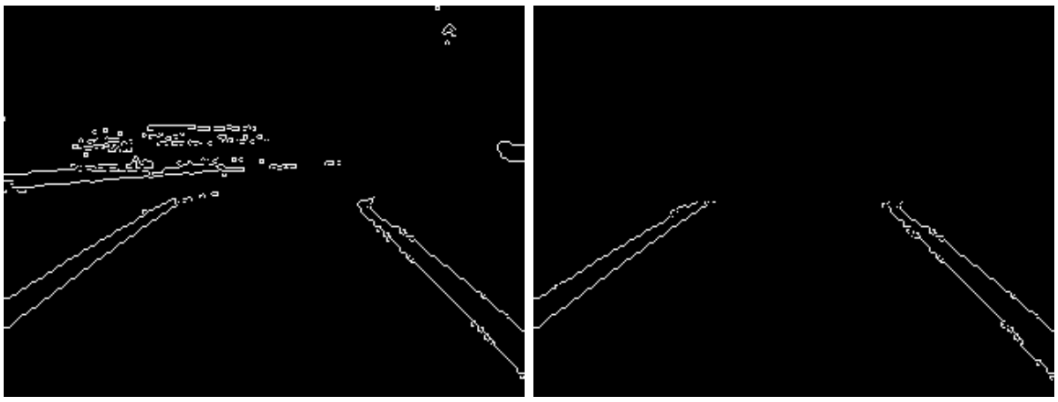
- Kernel_size =(3,3)
- Gauss_image = cv.GaussianBlur(fram,kernel_size,0)

4. Color Transformation is the 4th step: the image is still in RGB and must be transformed into HSV color space. Color separation will be used to remove unwanted colors from the image. So, grayscale range is necessary for our cause.
    - To define a range of black color :
        - Lower_black = np.array([0,0,0,])
        - Upper_black =np.array([227,100,70])

5. Boundaries in pictures : Dilation adds pixels to the boundaries of objects in an image. Applying dilation to the image will help close any loose spaces in the line images.
    - Dilate the threshold image:
        - Thresh = cv.dilate(thresh_1,rectKernel,iterations=1)

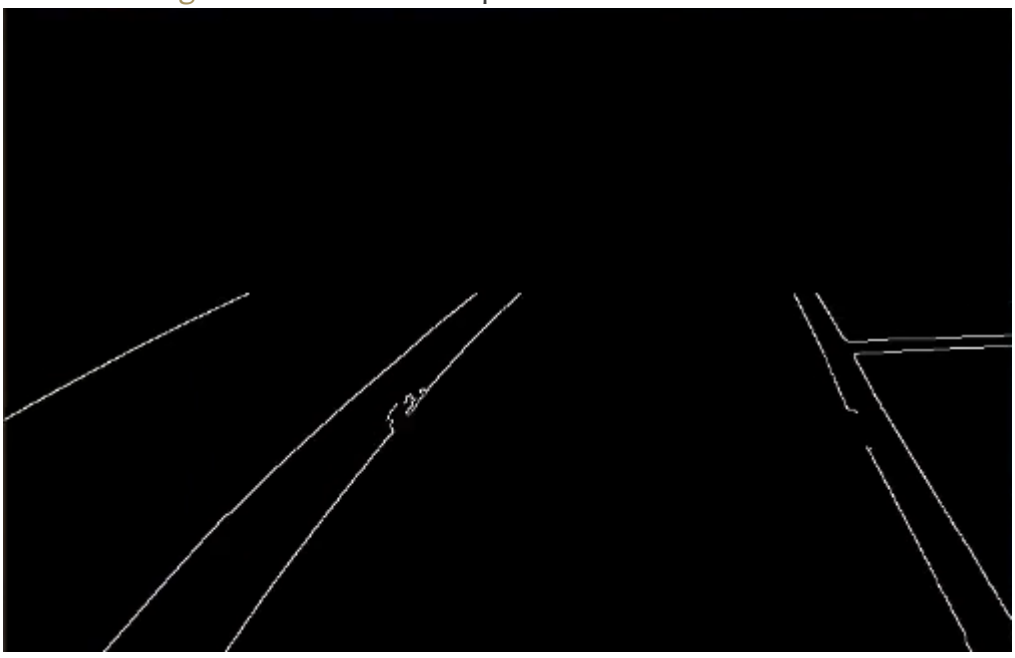
6. Edge detection for detecting the lane: edge detection can be used to extract useful structural information from an object. The OpenCV library provides a Canny edge detection function that can be used to detect edges in an image.
    - Apply canny edge detection from OpenCV
        - Low_threshold =200
        - High_threshold =400
        - Canny_edges =cv.Canny(thresh,low_threshold,high_threshold)
        - 

7. Output image contrains somme noise. All information other than lanes in the image can be isolated by reducing the region of interest.



Edges (left) and Cropped Edges (right)

8. Detect Line: finally after the region of interest was reduced, the lanes are clearly visible with four distinct lines. However, the computer doesn't know that these lines represent the boundaries of two lanes. So, we need a way to extract the coordinates for these lane lines. A Hough transform is a technique used in image processing to extract features like lines, circles, and ellipses. It can be to find straight lines from a number of pixels that seem to form a line. This can be done with the HoughLinesP function in OpenCV.



9. Line Segments combination: All the left lane lines have a positive slope, and the line segments belonging to the right lane line have a negative slope. Based on these two new groups, we can take the average of the slopes and intercepts of the line segments to get the slopes and intercepts of the left and right lane lines.

Conclusion:

he output from this function would be input to another algorithm for controlling the speed and heading of the vehicle. In order to do this, a steering angle can be computed from the detected lane lines and then passed to our AI deep learning algorithm in the **Jetbot** to minimize the steering angle and keep the vehicle centered.

Code of the programmed project above:

```
import cv2 as cv

import numpy as np
from time import gmtime, strftime, sleep, time
import logging
import nanocamera as nano

def weighted_img(img, initial_img, α=0.8, β=1., λ=0.):
    """
    `img` is the output of the hough_lines(), An image with lines drawn on it.
    Should be a blank image (all black) with lines drawn on it.
    `initial_img` should be the image before any processing.
    The result image is computed as follows:
    initial_img * α + img * β + λ
    NOTE: initial_img and img must be the same shape!
    """
    return cv.addWeighted(initial_img, α, img, β, λ)


def detect_line_segments(cropped_edges):
    # tuning min_threshold, minLineLength, maxLineGap is a trial and error process by hand
    rho = 1  # distance precision in pixel, i.e. 1 pixel
    angle = np.pi / 180  # angular precision in radian, i.e. 1 degree
    min_threshold = 10  # minimal of votes
    line_segments = cv.HoughLinesP(cropped_edges, rho, angle, min_threshold,
                                   np.array([]), minLineLength=10, maxLineGap=15)

    return line_segments

def make_points(frame, line):
    height, width, _ = frame.shape
    slope, intercept = line
    y1 = height  # bottom of the frame
    y2 = int(y1 * 1 / 2)  # make points from middle of the frame down

    # bound the coordinates within the frame
    x1 = max(-width, min(2 * width, int((y1 - intercept) / slope)))
    x2 = max(-width, min(2 * width, int((y2 - intercept) / slope)))
    return [[x1, y1, x2, y2]]

def average_slope_intercept(frame, line_segments):
    """
    This function combines line segments into one or two lane lines
    If all line slopes are < 0: then we only have detected left lane
    If all line slopes are > 0: then we only have detected right lane
    """
```

```
    lane_lines = []
    if line_segments is None:
        logging.info('No line_segment segments detected')
        return lane_lines

    height, width, _ = frame.shape
    left_fit = []
    right_fit = []

    boundary = 1/3
    left_region_boundary = width * (1 - boundary)  # left lane line segment should be on left 2/3 of the screen
    right_region_boundary = width * boundary # right lane line segment should be on left 2/3 of the screen

    for line_segment in line_segments:
        for x1, y1, x2, y2 in line_segment:
            if x1 == x2:
                logging.info('skipping vertical line segment (slope=inf): %s' % line_segment)
                continue
            fit = np.polyfit((x1, x2), (y1, y2), 1)
            slope = fit[0]
            intercept = fit[1]
            if slope < 0:
                if x1 < left_region_boundary and x2 < left_region_boundary:
                    left_fit.append((slope, intercept))
            else:
                if x1 > right_region_boundary and x2 > right_region_boundary:
                    right_fit.append((slope, intercept))

    left_fit_average = np.average(left_fit, axis=0)
    if len(left_fit) > 0:
        lane_lines.append(make_points(frame, left_fit_average))

    right_fit_average = np.average(right_fit, axis=0)
    if len(right_fit) > 0:
        lane_lines.append(make_points(frame, right_fit_average))

    logging.debug('lane lines: %s' % lane_lines)  # [[[316, 720, 484, 432]], [[1009, 720, 718, 432]]]
    return lane_lines

def display_lines(frame, lines, line_color=(0, 255, 255), line_width=20):
    line_image = np.zeros_like(frame)
    if lines is not None:
        for line in lines:
            for x1, y1, x2, y2 in line:
                cv.line(line_image, (x1, y1), (x2, y2), line_color, line_width)
    line_image = cv.addWeighted(frame, 0.8, line_image, 1, 1)
    return line_image

def region_of_interest(edges):
    height, width = edges.shape
    mask = np.zeros_like(edges)

    # only focus bottom half of the screen
    polygon = np.array([[
        (0, height * 1 / 2),
```

```
        (width, height * 1 / 2),
        (width, height),
        (0, height),
    ]], np.int32)
    cv.fillPoly(mask, polygon, 255)
    cropped_edges = cv.bitwise_and(edges, mask)
    return cropped_edges

# define a range of black color in HSV

lower_black = np.array([0, 0, 0])
upper_black = np.array([227, 100, 70])

# Rectangular Kernel
rectKernel = cv.getStructuringElement(cv.MORPH_RECT,(7,7))

# Create the Camera instance for 640 by 480
camera = nano.Camera()
if __name__=='__main__':

    print('Started')
    print ("Beginning Transmitting to channel: Happy_Robots")
    now = time()
    # commencing subtraction
    while True:
        try:
            # fetching each frame
            frame = camera.read()

            if frame is None:
                break

            # apply some gaussian blur to the image
            kenerl_size = (3, 3)
            gauss_image = cv.GaussianBlur(frame, kenerl_size, 0)

            # here we convert to the HSV colorspace
            hsv_image = cv.cvtColor(gauss_image, cv.COLOR_BGR2HSV)

            # apply color threshold to the HSV image to get only black colors
            thres_1 = cv.inRange(hsv_image, lower_black, upper_black)


            # dilate the the threshold image
            thresh = cv.dilate(thres_1, rectKernel, iterations=1)

            # apply canny edge detection
            low_threshold = 200
            high_threshold = 400
            canny_edges = cv.Canny(gauss_image, low_threshold, high_threshold)
            # get a region of interest
            roi_image = region_of_interest(canny_edges)
            line_segments = detect_line_segments(roi_image)
            lane_lines = average_slope_intercept(frame, line_segments)
            # overlay the line image on the main frame
            line_image = display_lines(frame, lane_lines)
```

```
        # display both the current frame and the fg masks
        cv.imshow('Frame', frame)
        cv.imshow('New Image', roi_image)
        cv.imshow('Line Image', line_image)

        keyboard = cv.waitKey(30)
        if keyboard == 'q' or keyboard == 27:
            break
    except KeyboardInterrupt:
        break

# cleanup
camera.release()
cv.destroyAllWindows()
del camera
print('Stopped')
```

H. Appendix H: Introduction: CNN

Convolutional Neural Networks (CNN):

Convolutional Neural Network (CNN) is a neural network which extracts or identifies a feature in a particular image. This forms one of the most fundamental operations in Machine Learning and is widely used as a base model in majority of Neural Networks like GoogleNet, VGG19 and others for various tasks such as Object Detection, Image Classification and others.

CNN has the following five basic components:

- **Convolution** : to detect features in an image
- **ReLU** : to make the image smooth and make boundaries distinct
- **Pooling** : to help fix distored images
- **Flattening** : to turn the image into a suitable representation
- **Full connection** : to process the data in a neural network

Convolution is the fundamental mathematical operation that is highly useful to detect features of an image. Convolution preserves the relationship between pixels by learning image features using small squares of input data. It is a mathematical operation that takes two inputs:

- image matrix
- a filter

In mathematical formula :

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

ReLU:

An image, in general, is highly non-linear that is it has **varied pixel values**, especially if it has many features to be detected.

Thus to decrease the non linearity in the convolution layer, we add the **rectifier function** say **max(x, 0)**, so that in case if there are negative pixel values, they will be replaced by zeroes.

Pooling

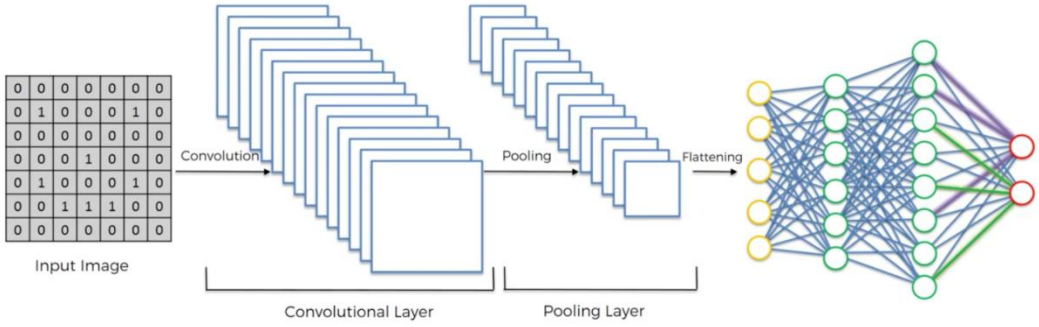
Pooling ensures that even if the features are a little distorted, or not very similar in different images, it still has the flexibility to identify the feature.

Flattening

In the flattening procedure, we basically take the elements in a pooled feature map and put them in a vector form. This becomes the input layer for the upcoming ANN.

Full Connection

Once the image is convolved, max pooled and flattened, the result is a vector. This vector acts as the input layer for an ANN which then works normally to detect the image.



I. Appendix I: Intoduction to Image processing :

What is image processing?

Image processing means manipulating the image to improve its condition or extract information from it. There are two ways to process an image:

Analog image processing is used to process physical images, prints, and other hard copies of images.

Digital image processing is used to manipulate digital images with the help of computer algorithms.

In both cases the input is an image. In analog image processing, the output is always an image. In digital image processing, the output can be an image or information about that image, including data about features, specifications, image frames, or overlays.

Today, image processing is widely used in medical imaging, biometrics, driverless vehicles, gaming, monitoring, law enforcement, and more. Here are some key pointers in moving your image forward:

Visualization: Shows processed data in ways that are understandable and visualizes objects that are not visible.

Image sharpening and restoration: Improving the quality of processed images.

Image Recovery: Help in image search.

Object Measurement: Measuring objects in an image.

Pattern Recognition: Distinguish and classify objects in an image, identify their positions, and understand the scene of the image.

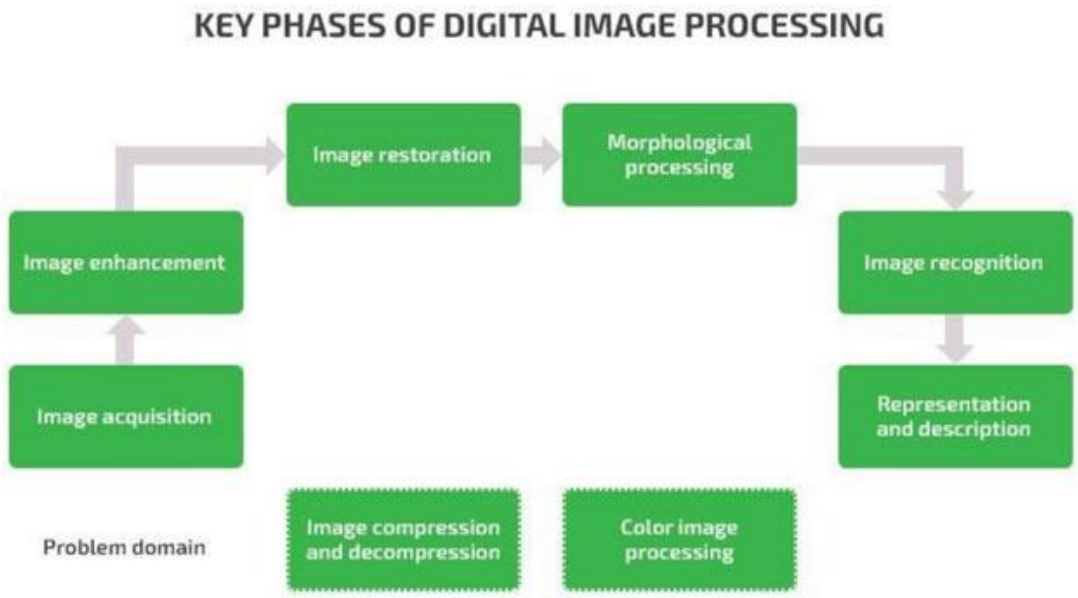


Image processing techniques, tools and methods

Most images taken with conventional receivers require reprocessing because their focus may be incorrect or too much noise. Edge filtering and detection are two of the most common digital image processing methods.

Filtering is used to enhance and modify the input image. With the help of various filters you can enhance or remove certain features of an image, reduce image noise and do other things. Popular filtering techniques include linear filtering, middle filtering, and Wiener filtering.

Edge Detection uses filters to split the image and extract data. This method helps to find meaningful edges of objects in the processed images by detecting cases where the brightness of the image is turned off. Mineral edge detection, subel edge detection and Roberts edge detection are some of the most popular edge detection techniques.

J. Appendix J : AlexNet & ResNet & VGG19 :

AlexNet :

AlexNet was not the first fast GPU-implementation of a CNN to win an image recognition contest. A CNN on GPU by K. Chellapilla et al. (2006) was 4 times faster than an equivalent implementation on CPU. They also significantly improved on the best performance in the literature for multiple image databases.

ResNet :

Deep residual networks like the popular ResNet-50 model is a convolutional neural network (CNN) that is 50 layers deep. A residual neural network (ResNet) is an artificial neural network (ANN) of a kind that stacks residual blocks on top of each other to form a network.

ResNet stands for Residual Network. It is an innovative neural network that was first introduced by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun in their 2015 computer vision research paper titled 'Deep Residual Learning for Image Recognition'.

residual network or ResNet was a major innovation that has changed the training of deep convolutional neural networks for tasks related to computer vision. While the original Resnet had 34 layers and used 2-layer blocks, other advanced variants such as the Resnet50 made the use of 3-layer bottleneck blocks to ensure improved accuracy and lesser training time.

AlexNet :

AlexNet came out in 2012 and it improved on the traditional Convolutional neural networks, So we can understand **VGG as a successor of the AlexNet** but it was created by a different group named as **Visual Geometry Group** at **Oxford**'s and hence the name VGG, It carries and uses some ideas from it's predecessors and improves on them and uses deep Convolutional neural layers to improve accuracy.

K. Appendix K: OpenCV & TensorFlow:

OpenCV:

OpenCV is a library of programming functions first developed by Intel, later supported by Willow Garage and then Itseez. OpenCV's open source library is cross-platform and free to use under a BSD license.

The library supports various platforms including Windows, Linux, Android, iOS, Mac and contains more than 2500 optimized algorithms.

OpenCV is a library of programming functions for **real-time image processing.**

This library uses the BSD license and is therefore free for academic and commercial use.

TensorFlow:

TensorFlow, developed by Google Elite, is an open-source library for numerical computing and large-scale machine learning. TensorFlow integrates machine learning and deep learning models and algorithms (aka neural networks) and makes them useful and usable. The library uses Python to provide a well-built API for building applications that run at a high level of performance. TensorFlow can deploy deep neural networks for handwritten digit classification, image recognition, recurrent neural networks, sequential models for machine translation, natural language processing, and partial differential equations (PDE). Equation train and execute.

TensorFlow allows developers to create dataflow graphs structures that describe how data flows through a graph or create a set of processing nodes. Each node in this diagram represents a mathematical operation, and each connection or edge between nodes represents a multidimensional data array or a tensor. TensorFlow provides all these features to programmers in the Python programming language. Learning and working with Python is easy and provides good ways to express how high-level abstractions combine. The nodes and tensors in TensorFlow are Python objects, and the TensorFlow applications themselves are Python applications. However, real mathematical operations are not performed in Python. The libraries offered by TensorFlow are written in the powerful C ++ language. Python only directs traffic between components and provides high-level programming abstracts to connect them.

TensorFlow applications can be run on most existing platforms such as a local machine, a cluster in the cloud, Android and iOS devices, as well as CPUs and GPUs. If you use Google's proprietary cloud, you have the ability to run TensorFlow on Google's TensorFlow Processing Unit, which is a special purpose integrated circuit, for added acceleration. TPU is a programmable artificial intelligence accelerator designed to provide high throughput in low-precision computing.

L. APPENDIX L: ROAD FOLLOWING

1. DATA COLLECTION

In [ ]:

```
# IPython Libraries for display and widgets
import ipywidgets
import traitlets
import ipywidgets.widgets as widgets
from IPython.display import display

# Camera and Motor Interface for JetBot
from jetbot import Robot, Camera, bgr8_to_jpeg

# Basic Python packages for image annotation
from uuid import uuid1
import os
import json
import glob
import datetime
import numpy as np
import cv2
import time
```

In [ ]:

```
from jupyter_clickable_image_widget import ClickableImageWidget

DATASET_DIR = 'dataset_xy'

# we have this "try/except" statement because these next functions can throw an error if the directories exist already
try:
    os.makedirs(DATASET_DIR)
except FileExistsError:
    print('Directories not created because they already exist')

camera = Camera()

# create image preview
camera_widget = ClickableImageWidget(width=camera.width, height=camera.height)
snapshot_widget = ipywidgets.Image(width=camera.width, height=camera.height)
traitlets.dlink((camera, 'value'), (camera_widget, 'value'), transform=bgr8_to_jpeg)

# create widgets
count_widget = ipywidgets.IntText(description='count')
# manually update counts at initialization
count_widget.value = len(glob.glob(os.path.join(DATASET_DIR, '*.jpg')))

def save_snapshot(_, content, msg):
    if content['event'] == 'click':
        data = content['eventData']
        x = data['offsetX']
        y = data['offsetY']

        # save to disk
        #dataset.save_entry(category_widget.value, camera.value, x, y)
        uuid = 'xy_%03d_%03d_%s' % (x, y, uuid1())
        image_path = os.path.join(DATASET_DIR, uuid + '.jpg')
        with open(image_path, 'wb') as f:
            f.write(camera_widget.value)

        # display saved snapshot
        snapshot = camera.value.copy()
        snapshot = cv2.circle(snapshot, (x, y), 8, (0, 255, 0), 3)
        snapshot_widget.value = bgr8_to_jpeg(snapshot)
        count_widget.value = len(glob.glob(os.path.join(DATASET_DIR, '*.jpg')))

camera_widget.on_msg(save_snapshot)

data_collection_widget = ipywidgets.VBox([
    ipywidgets.HBox([camera_widget, snapshot_widget]),
    count_widget
])

display(data_collection_widget)
```

In [ ]:

```
camera.stop()
```

2. TRAIN MODEL

Entrée [ ]:
```
import torch
import torch.optim as optim
import torch.nn.functional as F
import torchvision
import torchvision.datasets as datasets
import torchvision.models as models
import torchvision.transforms as transforms
import glob
import PIL.Image
import os
import numpy as np
```

Entrée [ ]:
```
def get_x(path, width):
    """Gets the x value from the image filename"""
    return (float(int(path.split("_")[1])) - width/2) / (width/2)

def get_y(path, height):
    """Gets the y value from the image filename"""
    return (float(int(path.split("_")[2])) - height/2) / (height/2)

class XYDataset(torch.utils.data.Dataset):

    def __init__(self, directory, random_hflips=False):
        self.directory = directory
        self.random_hflips = random_hflips
        self.image_paths = glob.glob(os.path.join(self.directory, '*.jpg'))
        self.color_jitter = transforms.ColorJitter(0.3, 0.3, 0.3, 0.3)

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        image_path = self.image_paths[idx]

        image = PIL.Image.open(image_path)
        width, height = image.size
        x = float(get_x(os.path.basename(image_path), width))
        y = float(get_y(os.path.basename(image_path), height))

        if float(np.random.rand(1)) > 0.5:
            image = transforms.functional.hflip(image)
            x = -x

        image = self.color_jitter(image)
        image = transforms.functional.resize(image, (224, 224))
        image = transforms.functional.to_tensor(image)
        image = image.numpy()[::-1].copy()
        image = torch.from_numpy(image)
        image = transforms.functional.normalize(image, [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

        return image, torch.tensor([x, y]).float()

dataset = XYDataset('dataset_xy', random_hflips=False)
```

Entrée [ ]:
```
test_percent = 0.1
num_test = int(test_percent * len(dataset))
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [len(dataset) - num_test, num_test])
```

Entrée [ ]:
```
train_loader = torch.utils.data.DataLoader(
    train_dataset,
    batch_size=8,
    shuffle=True,
    num_workers=0
)

test_loader = torch.utils.data.DataLoader(
    test_dataset,
    batch_size=8,
    shuffle=True,
    num_workers=0
)
```

Entrée [ ]:
```
model = models.resnet18(pretrained=True)
```

Entrée [ ]:
```
model.fc = torch.nn.Linear(512, 2)
device = torch.device('cuda')
model = model.to(device)
```

Entrée [ ]:
```
NUM_EPOCHS = 70
BEST_MODEL_PATH = 'best_steering_model_xy.pth'
best_loss = 1e9

optimizer = optim.Adam(model.parameters())

for epoch in range(NUM_EPOCHS):

    model.train()
    train_loss = 0.0
    for images, labels in iter(train_loader):
        images = images.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = F.mse_loss(outputs, labels)
        train_loss += float(loss)
        loss.backward()
        optimizer.step()
    train_loss /= len(train_loader)

    model.eval()
    test_loss = 0.0
    for images, labels in iter(test_loader):
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        loss = F.mse_loss(outputs, labels)
        test_loss += float(loss)
    test_loss /= len(test_loader)

    print('%f, %f' % (train_loss, test_loss))
    if test_loss < best_loss:
        torch.save(model.state_dict(), BEST_MODEL_PATH)
        best_loss = test_loss
```

3. LOAD MODEL

Entrée [ ]:
```
import torchvision
import torch

model = torchvision.models.resnet18(pretrained=False)
model.fc = torch.nn.Linear(512, 2)
model = model.cuda().eval().half()
```

Entrée [ ]:
```
model.load_state_dict(torch.load('best_steering_model_xy.pth'))
```

Entrée [ ]:
```
device = torch.device('cuda')
```

Entrée [ ]:
```
from torch2trt import torch2trt

data = torch.zeros((1, 3, 224, 224)).cuda().half()

model_trt = torch2trt(model, [data], fp16_mode=True)
```

Entrée [ ]:
```
torch.save(model_trt.state_dict(), 'best_steering_model_xy_trt.pth')
```

Entrée [ ]:
```
import torch
device = torch.device('cuda')
```

Entrée [ ]:
```
import torch
from torch2trt import TRTModule

model_trt = TRTModule()
model_trt.load_state_dict(torch.load('best_steering_model_xy_trt.pth'))
```

Entrée [ ]:
```
import torchvision.transforms as transforms
import torch.nn.functional as F
import cv2
import PIL.Image
import numpy as np

mean = torch.Tensor([0.485, 0.456, 0.406]).cuda().half()
std = torch.Tensor([0.229, 0.224, 0.225]).cuda().half()

def preprocess(image):
    image = PIL.Image.fromarray(image)
    image = transforms.functional.to_tensor(image).to(device).half()
    image.sub_(mean[:, None, None]).div_(std[:, None, None])
    return image[None, ...]
```

Entrée [ ]:
```
from IPython.display import display
import ipywidgets
import traitlets
from jetbot import Camera, bgr8_to_jpeg

camera = Camera()
```

Entrée [ ]:
```
image_widget = ipywidgets.Image()

traitlets.dlink((camera, 'value'), (image_widget, 'value'), transform=bgr8_to_jpeg)

display(image_widget)
```

Entrée [ ]:
```
from jetbot import Robot

robot = Robot()
```

Entrée [ ]:
```
speed_gain_slider = ipywidgets.FloatSlider(min=0.0, max=1.0, step=0.01, description='speed gain')
steering_gain_slider = ipywidgets.FloatSlider(min=0.0, max=1.0, step=0.01, value=0.2, description='steering gain')
steering_dgain_slider = ipywidgets.FloatSlider(min=0.0, max=0.5, step=0.001, value=0.0, description='steering kd')
steering_bias_slider = ipywidgets.FloatSlider(min=-0.3, max=0.3, step=0.01, value=0.0, description='steering bias')

display(speed_gain_slider, steering_gain_slider, steering_dgain_slider, steering_bias_slider)
```

Entrée [ ]:
```
x_slider = ipywidgets.FloatSlider(min=-1.0, max=1.0, description='x')
y_slider = ipywidgets.FloatSlider(min=0, max=1.0, orientation='vertical', description='y')
steering_slider = ipywidgets.FloatSlider(min=-1.0, max=1.0, description='steering')
speed_slider = ipywidgets.FloatSlider(min=0, max=1.0, orientation='vertical', description='speed')

display(ipywidgets.HBox([y_slider, speed_slider]))
display(x_slider, steering_slider)
```

Entrée [ ]:
```
angle = 0.0
angle_last = 0.0

def execute(change):
    global angle, angle_last
    image = change['new']
    xy = model_trt(preprocess(image)).detach().float().cpu().numpy().flatten()
    x = xy[0]
    y = (0.5 - xy[1]) / 2.0

    x_slider.value = x
    y_slider.value = y

    speed_slider.value = speed_gain_slider.value

    angle = np.arctan2(x, y)
    pid = angle * steering_gain_slider.value + (angle - angle_last) * steering_dgain_slider.value
    angle_last = angle

    steering_slider.value = pid + steering_bias_slider.value

    robot.left_motor.value = max(min(speed_slider.value + steering_slider.value, 1.0), 0.0)
    robot.right_motor.value = max(min(speed_slider.value - steering_slider.value, 1.0), 0.0)

execute({'new': camera.value})
```

Entrée [ ]:
```
camera.observe(execute, names='value')
```

Entrée [ ]:
```
import time

camera.unobserve(execute, names='value')

time.sleep(0.1)  # add a small sleep to make sure frames have finished processing

robot.stop()
```

Entrée [ ]:
```
camera.stop()
```

M. APPENDIX M: COLLISION AVOIDANCE

1. DATA COLLECTION

Entrée [ ]:
```
import traitlets
import ipywidgets.widgets as widgets
from IPython.display import display
from jetbot import Camera, bgr8_to_jpeg

camera = Camera.instance(width=224, height=224)

image = widgets.Image(format='jpeg', width=224, height=224)
camera_link = traitlets.dlink((camera, 'value'), (image, 'value'), transform=bgr8_to_jpeg)

display(image)
```

Entrée [ ]:
```
import os

blocked_dir = 'dataset/blocked'
free_dir = 'dataset/free'


try:
    os.makedirs(free_dir)
    os.makedirs(blocked_dir)
except FileExistsError:
    print('Directories not created because they already exist')
```

Entrée [ ]:
```
button_layout = widgets.Layout(width='128px', height='64px')
free_button = widgets.Button(description='add free', button_style='success', layout=button_layout)
blocked_button = widgets.Button(description='add blocked', button_style='danger', layout=button_layout)
free_count = widgets.IntText(layout=button_layout, value=len(os.listdir(free_dir)))
blocked_count = widgets.IntText(layout=button_layout, value=len(os.listdir(blocked_dir)))

display(widgets.HBox([free_count, free_button]))
display(widgets.HBox([blocked_count, blocked_button]))
```

Entrée [ ]:
```
from uuid import uuid1

def save_snapshot(directory):
    image_path = os.path.join(directory, str(uuid1()) + '.jpg')
    with open(image_path, 'wb') as f:
        f.write(image.value)

def save_free():
    global free_dir, free_count
    save_snapshot(free_dir)
    free_count.value = len(os.listdir(free_dir))

def save_blocked():
    global blocked_dir, blocked_count
    save_snapshot(blocked_dir)
    blocked_count.value = len(os.listdir(blocked_dir))

# attach the callbacks, we use a 'lambda' function to ignore the
# parameter that the on_click event would provide to our function
# because we don't need it.
free_button.on_click(lambda x: save_free())
blocked_button.on_click(lambda x: save_blocked())
```

Entrée [ ]:
```
display(image)
display(widgets.HBox([free_count, free_button]))
display(widgets.HBox([blocked_count, blocked_button]))
```

Entrée [ ]:
```
camera.stop()
```

2. TRAINING MODEL

Entrée [ ]:
```
import torch
import torch.optim as optim
import torch.nn.functional as F
import torchvision
import torchvision.datasets as datasets
import torchvision.models as models
import torchvision.transforms as transforms
```

Entrée [ ]:
```
dataset = datasets.ImageFolder(
    'dataset',
    transforms.Compose([
        transforms.ColorJitter(0.1, 0.1, 0.1, 0.1),
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
)
```

Entrée [ ]:
```
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [len(dataset) - 50, 50])
```

Entrée [ ]:
```
train_loader = torch.utils.data.DataLoader(
    train_dataset,
    batch_size=8,
    shuffle=True,
    num_workers=0,
)

test_loader = torch.utils.data.DataLoader(
    test_dataset,
    batch_size=8,
    shuffle=True,
    num_workers=0,
)
```

Entrée [ ]:
```
model = models.resnet18(pretrained=True)
```

Entrée [ ]:
```
model.fc = torch.nn.Linear(512, 2)
```

Entrée [ ]:
```
device = torch.device('cuda')
model = model.to(device)
```

Entrée [ ]:
```
NUM_EPOCHS = 30
BEST_MODEL_PATH = 'best_model_resnet18.pth'
best_accuracy = 0.0

optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

for epoch in range(NUM_EPOCHS):

    for images, labels in iter(train_loader):
        images = images.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = F.cross_entropy(outputs, labels)
        loss.backward()
        optimizer.step()

    test_error_count = 0.0
    for images, labels in iter(test_loader):
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        test_error_count += float(torch.sum(torch.abs(labels - outputs.argmax(1))))

    test_accuracy = 1.0 - float(test_error_count) / float(len(test_dataset))
    print('%d: %f' % (epoch, test_accuracy))
    if test_accuracy > best_accuracy:
        torch.save(model.state_dict(), BEST_MODEL_PATH)
        best_accuracy = test_accuracy
```

3. LOAD MODEL

Entrée [ ]:
```
import torch
import torchvision

model = torchvision.models.resnet18(pretrained=False)
model.fc = torch.nn.Linear(512, 2)
model = model.cuda().eval().half()
```

Entrée [ ]:
```
model.load_state_dict(torch.load('best_model_resnet18.pth'))
```

Entrée [ ]:
```
device = torch.device('cuda')
```

Entrée [ ]:
```
from torch2trt import torch2trt

data = torch.zeros((1, 3, 224, 224)).cuda().half()

model_trt = torch2trt(model, [data], fp16_mode=True)
```

Entrée [ ]:
```
torch.save(model_trt.state_dict(), 'best_model_trt.pth')
```

Entrée [ ]:
```
import torch
device = torch.device('cuda')
```

Entrée [ ]:
```
import torchvision
from torch2trt import TRTModule

model_trt = TRTModule()
model_trt.load_state_dict(torch.load('best_model_trt.pth'))
```

Entrée [ ]:
```
import torchvision.transforms as transforms
import torch.nn.functional as F
import cv2
import PIL.Image
import numpy as np

mean = torch.Tensor([0.485, 0.456, 0.406]).cuda().half()
std = torch.Tensor([0.229, 0.224, 0.225]).cuda().half()

normalize = torchvision.transforms.Normalize(mean, std)

def preprocess(image):
    image = PIL.Image.fromarray(image)
    image = transforms.functional.to_tensor(image).to(device).half()
    image.sub_(mean[:, None, None]).div_(std[:, None, None])
    return image[None, ...]
```

Entrée [ ]:
```
import traitlets
from IPython.display import display
import ipywidgets.widgets as widgets
from jetbot import Camera, bgr8_to_jpeg

camera = Camera.instance(width=224, height=224)
image = widgets.Image(format='jpeg', width=224, height=224)
blocked_slider = widgets.FloatSlider(description='blocked', min=0.0, max=1.0, orientation='vertical')
speed_slider = widgets.FloatSlider(description='speed', min=0.0, max=0.5, value=0.0, step=0.01, orientation='horizontal')

camera_link = traitlets.dlink((camera, 'value'), (image, 'value'), transform=bgr8_to_jpeg)

display(widgets.VBox([widgets.HBox([image, blocked_slider]), speed_slider]))
```

Entrée [ ]:
```
from jetbot import Robot

robot = Robot()
```

Entrée [ ]:
```
import torch.nn.functional as F
import time

def update(change):
    global blocked_slider, robot
    x = change['new']
    x = preprocess(x)
    y = model_trt(x)
    #print(y)


    y = F.softmax(y, dim=1)
    #print(y)

    prob_blocked = float(y.flatten()[0])
    #print(prob_blocked)

    blocked_slider.value = prob_blocked

    if prob_blocked < 0.5:
        robot.forward(speed_slider.value)
    else:
        robot.left(speed_slider.value)

    time.sleep(0.001)

update({'new': camera.value})  # we call the function once to initialize
```

Entrée [ ]:
```
camera.observe(update, names='value')
```

Entrée [ ]:
```
import time

camera.unobserve(update, names='value')

time.sleep(0.1)  # add a small sleep to make sure frames have finished processing

robot.stop()
```

Entrée [ ]:
```
camera_link.unlink()
```

Entrée [ ]:
```
camera_link.link()
```

Entrée [ ]:
```
camera.stop()
```

N. APPENDIX N: ROAD FOLLOWING AND COLLISION AVOIDANCE

Entrée [ ]:
```
import torch
import torchvision

model = torchvision.models.resnet18(pretrained=False)
model.fc = torch.nn.Linear(512, 2)
model = model.cuda().eval().half()
```

Entrée [ ]:
```
model.load_state_dict(torch.load('best_model_resnet18.pth'))
```

Entrée [ ]:
```
device = torch.device('cuda')
```

Entrée [ ]:
```
from torch2trt import torch2trt

data = torch.zeros((1, 3, 224, 224)).cuda().half()

model_trt = torch2trt(model, [data], fp16_mode=True)
```

Entrée [ ]:
```
torch.save(model_trt.state_dict(), 'best_model_trt.pth')
```

Entrée [ ]:
```
import torch
device = torch.device('cuda')
```

Entrée [ ]:
```
import torchvision
from torch2trt import TRTModule

model_trt = TRTModule()
model_trt.load_state_dict(torch.load('best_model_trt.pth'))
```

Entrée [ ]:
```
import torchvision.transforms as transforms
import torch.nn.functional as F
import cv2
import PIL.Image
import numpy as np

mean = torch.Tensor([0.485, 0.456, 0.406]).cuda().half()
std = torch.Tensor([0.229, 0.224, 0.225]).cuda().half()

normalize = torchvision.transforms.Normalize(mean, std)

def preprocess(image):
    image = PIL.Image.fromarray(image)
    image = transforms.functional.to_tensor(image).to(device).half()
    image.sub_(mean[:, None, None]).div_(std[:, None, None])
    return image[None, ...]
```

Entrée [ ]:
```
import traitlets
from IPython.display import display
import ipywidgets.widgets as widgets
from jetbot import Camera, bgr8_to_jpeg

camera = Camera.instance(width=224, height=224)
image = widgets.Image(format='jpeg', width=224, height=224)
blocked_slider = widgets.FloatSlider(description='blocked', min=0.0, max=1.0, orientation='vertical')
speed_slider = widgets.FloatSlider(description='speed', min=0.0, max=0.5, value=0.0, step=0.01, orientation='horizontal')

camera_link = traitlets.dlink((camera, 'value'), (image, 'value'), transform=bgr8_to_jpeg)

display(widgets.VBox([widgets.HBox([image, blocked_slider]), speed_slider]))
```

Entrée [ ]:
```
from jetbot import Robot

robot = Robot()
```

Entrée [ ]:
```
import torch.nn.functional as F
import time

def update(change):
    global blocked_slider, robot
    x = change['new']
    x = preprocess(x)
    y = model_trt(x)
    #print(y)


    y = F.softmax(y, dim=1)
    #print(y)

    prob_blocked = float(y.flatten()[0])
    #print(prob_blocked)

    blocked_slider.value = prob_blocked

    if prob_blocked < 0.5:
        robot.forward(speed_slider.value)
    else:
        robot.left(speed_slider.value)

    time.sleep(0.001)

update({'new': camera.value})  # we call the function once to initialize
```

Entrée [ ]:
```
camera.observe(update, names='value')
```

Entrée [ ]:
```
import time

camera.unobserve(update, names='value')

time.sleep(0.1)  # add a small sleep to make sure frames have finished processing

robot.stop()
```

Entrée [ ]:
```
camera_link.unlink()
```

Entrée [ ]:
```
camera_link.link()
```

Entrée [ ]:
```
camera.stop()
```