Escuela Politécnica Superior

Automation of Robotic Agents and Generation of Synthetic Data for Semantic Segmentation.

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Jordi Amorós Moreno
Tutors:
José García-Rodriguez
Sergiu Ovidiu-Oprea

Universitat d'Alacant
Universidad de Alicante

Septiembre 2019

Automation of Robotic Agents and Generation of Synthetic Data for Semantic Segmentation.

Bachelor's Thesis

Autor
Jordi Amorós Moreno

Tutor/es
José García-Rodriguez
Departamento de Tecnología Informática y Computación
Sergiu Ovidiu-Oprea
Departamento de Tecnología Informática y Computación

Grado en Ingeniería Informática

Escuela Politécnica Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Septiembre 2019

Abstract

Semantic segmentation is one of the most relevant techniques in the object detection field since it provides highly valuable scene information and context. Nowadays, data driven algorithms, specifically deep learning, dictate the current state-of-the-art. Although these algorithms can be extremely accurate, they require vast amounts of data. Additionally, this data needs to be highly variable, which is fundamental in order for these models to properly achieve a general solution. On top of that, obtaining such data requires human operators for both the capture and labeling process. All of these constraints present a major drawback in terms of time, cost and resources.

The Sim-To-Real field offers an alternative by synthetically generating the data for these algorithms. In this thesis, we propose a modification of a data generation framework, which is aimed towards the automation of action sequences. Then, we use the generated data in order to train and evaluate the performance of semantic segmentation architectures.

Resumen

La segmentación semántica es una de las técnicas más relevantes en el campo de detección de objetos, ya que provee información muy valiosa del contexto de una escena. Hoy en día, los algoritmos basados en datos, especificamente el deep learning, dominan el actual estado del arte. Aunque estos algoritmos pueden ser extremadamente precisos, requieren grandes cantidades de datos. Además, es fundamental que sean altamente variables para que los modelos sean capaces de obtener una solución general. Adicionalmente, la obtención de dichos datos requiere de operadores humanos tanto para el proceso de captura como el de etiquetado. Todas estas limitaciones suponen un gran inconveniente en términos de tiempo, coste y recursos.

El campo Sim-To-Real ofrece una alternativa al proceso clásico de obtencion de datos, generándolos de forma sintética y etiquetándolos automáticamente. En esta tesis, proponemos una modificación de un framework de generación de datos, centrándonos en la automatización de la generación de acciones. A continuación, utilizamos los datos generados para entrenar y evaluar el rendimiento de arquitecturas de segmentación semántica.

Acknowledgements

First, and most of all, I would like to thank my tutor Jose Garcia-Rodriguez for his endless patience and guidance, and for giving me the opportunity to work in such a great project. I would also like to thank all the members of the Department of Information Technologies and Computing (DTIC), specially my co-tutors Sergiu Ovidiu-Oprea, Albert Garcia-Garcia, John Castro-Vargas and Pablo Martinez-Gonzalez for their constant support and suggestions.

I would also like to thank my friends and family for their relentless support through the good and the hard times. Without them this thesis would have not been possible.

To my family, who showed me the true meaning of perseverance.

"I don't know everything, I just know what I know."

Nishio Ishin

Contents

List of Figures

List of Tables

Listings

List of Acronyms

AI	Artificial Inteligence.
API	Application Program Interface.
CNN	Convolutional Neural Network.
CPU	Central Processing Unit.
CRF	Conditional Random Fields.
CUDA	Compute Unified Device Architecture.
DL	Deep Learning.
DTIC	Department of Information Technologies and Computing.
FCN	Fully Convolutional Network.
GAN	Generative Adversarial Network.
GPU	Graphics Processing Unit.
IDE	Integrated Development Environment.
ILSVRC	ImageNet Large Scale Visual Recognition Challenge.
NN	Neural Network.
RNN	Recurrent Neural Network.
TFG	Trabajo Final de Grado.
UE4	Unreal Engine 4.
UTP	Unite The People.
VGG	Visual Geometry Group.
VR	Virtual Reality.

1 Introduction

In this first chapter we go over the main ideas of this work. In Section 1.1 we give an overview of the whole thesis. Section 1.2 describes the motivations of this research. Finally, in Section 1.3 we point out the main proposal as well as the goals for this work.

1.1 Overview

In the last decade, data driven algorithms have improved tremendously and large-scale high quality datasets have been created in order to improve the accuracy of such algorithms. However it is still extremely expensive, both in time and resources, to create such datasets. The Sim-To-Real field offers a promising alternative by synthetically generating and automatically annotating the data necessary for the aforementioned algorithms.

In this bachelor's thesis we propose a modification for a specific data-generation framework which allows for a more automatic approach of the data generation process. Additionally, with the purpose of demonstrating that synthetic data is a viable alternative to real-world datasets, we research some of the latest semantic segmentation techniques in order to verify that such data will properly transfer to real-world domains.

This document is structured as follows: Chapter 1 gives an overview of the whole thesis and goes through the motivations and proposals for this work. Chapter 2 goes over the related works and State of the Art of the Sim-To-Real and Semantic Segmentation fields, as well as some works that served as inspiration and motivated this project. Chapter 3 describes the materials and methodologies used in this work. In Chapter 4 we describe the process of expansion of the synthetic framework as well as the Semantic Segmentation implementations. Chapter 5 describes the experimentation process and its results. Finally, in Chapter 6 we go over the conclusions of this research.

1.2 Motivation

Although semantic segmentation has become increasingly popular and new datasets have emerged, the manual labeling of such data is still incredibly time consuming. Because of this, the Sim-To-Real field could prove extremely useful to the semantic segmentation problem and is one of the key motivations of this work. Additionally, most current synthetic data generation frameworks require user inputs in order to generate sequences, for this reason, we concentrate our efforts in developing a user-friendly framework for researchers to easily generate synthetic data sequences.

UnrealROX: *An eXtremely Photorealistic Virtual Reality Environment for Robotics Simulations and Synthetic Data Generation* by Martinez-Gonzalez et al. (2018) is a Virtual Reality (VR) environment used to generate *The Robotrix: An eXtremely Photorealistic and*

Figure 1.1: Snapshots of the Robotrix dataset extracted from Garcia-Garcia et al. (2019).

Very-Large-Scale Indoor Dataset of Sequences with Robot Trajectories and Interactions by Garcia-Garcia et al. (2019) which was presented at the IROS conference in 2018.

This work was motivated by a collaboration with the DTIC and the *3D Perception Lab* group, which mainly focuses on the fields of 3D Computer Vision, Machine Learning and Graphics Processing Unit (GPU) Computing. This work was carried out in the context of the COMBAHO Spanish National Project funded by *Ministerio de Economia y Competitividad* of the Spanish Government and directed by Jose Garcia-Rodriguez and Miguel Angel Cazorla-Quevedo.

1.3 Proposal and Goals

In UnrealROX, a VR headset is used in order to control the character or *Agent*, this means that in order to generate data an operator must manually perform the required actions for a sequence. This presents a major drawback both in time and equipment requirements. The main proposal for this work is to develop an extension for the UnrealROX framework in order to automatize the synthetic data generation process, as well as conducting a study on how semantic segmentation architectures can transfer the knowledge of such synthetic data into the real-world domain.

As for the main objectives of this work, one of the first tasks is to establish a new type of *Agent* in the framework that is not user-controlled. This would allow users to generate data sequences without the need of a VR headset and user input, providing a faster and more convenient way to obtain datasets.

The second main objective of this work is to prove how synthetic data can be applied to real world problems. This is, in other words, use synthetically generated datasets with the intention of transfer learning to real data. In order to demonstrate this, we develop data driven algorithms and verify their effectivity with real-world datasets.

2 State of the Art

Semantic segmentation is an extremely important task in the field of computer vision due to its enormous value towards complete scene understanding. Because of this, many works on this matter have been published. In this chapter we analyze some of the most relevant ones and it is organized as follows: Section 2.1 give a brief introduction to the Semantic segmentation problem. In section 2.2 we delineate the importance of the Sim-To-Real field, as well as review some of the latest works on the matter. In Section 2.3 we cover several of the most important and recent deep network architectures. Finally in Section 2.4 we take a look at some of the most important data-sets and frameworks that tackle the semantic segmentation problem.

2.1 Introduction

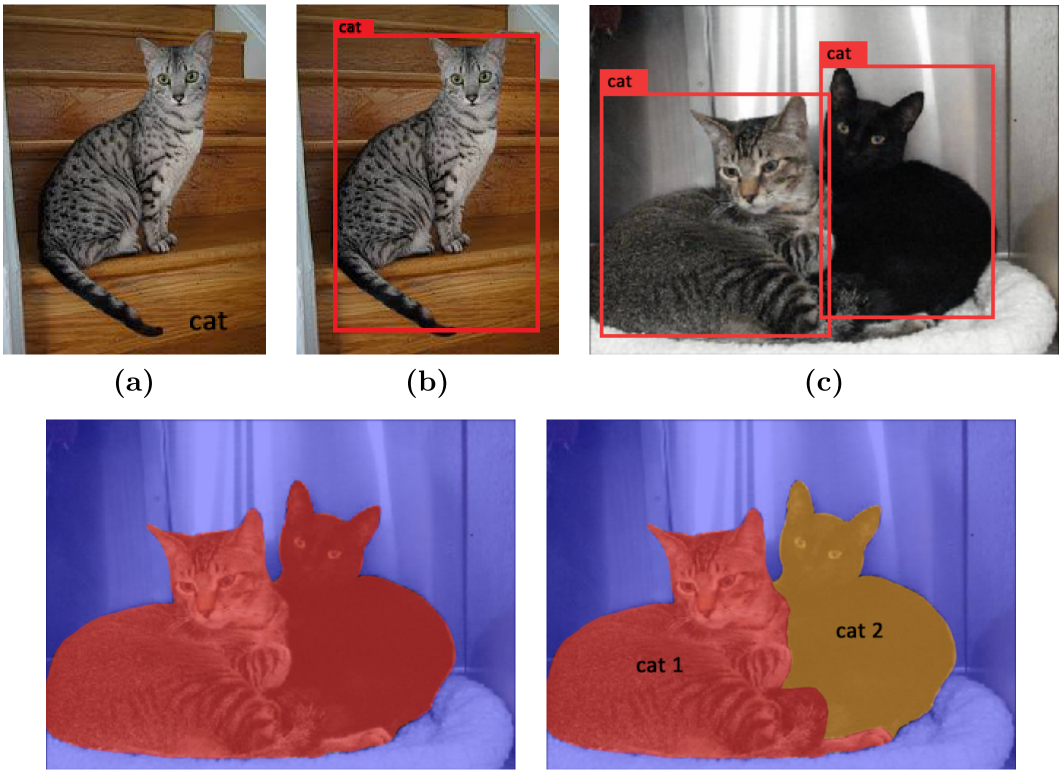
Before we dive into the next sections it is important to understand the semantic segmentation problem and where it comes from. Semantic segmentation is a natural evolution of the object recognition problem, the goal is to infer the class for every pixel on the image, obtaining a pixel-by-pixel labeled output. Semantic segmentation is not so different from classic object recognition, it just adds an extra layer of complexity towards a more fine-grained solution. With semantic segmentation, not only we are able to infer what are the objects in a certain scene, but we also gain knowledge of its localization and exact boundaries. We could go further and try to differentiate instances of the same class, that would be instance segmentation. Figure 2.1 shows the different object recognition solutions from less to more complex.

In the following sections, we review some of the current state-of-the-art works in the fields of semantic segmentation and Sim-To-Real.

2.2 Sim-To-Real

In the last decade, data driven algorithms have vastly surpassed traditional techniques for computer vision problems (Walsh et al., 2019). These algorithms, although can be tuned and improved in many different ways, still require vast amounts of high-quality, precisely annotated data in order to yield good results. In the real world environment, there are quite a few limitations to the quantity and quality of the data that can be produced. For instance, we could be limited to the number of cameras and annotators, moving physical objects to setup scenes could also be difficult and time consuming, and dangerous situations could be risky to set up properly, i.e., trying to get an autonomous car to learn to avoid pedestrians when there is no time to brake.

Sim2Real is a specific section of the data science field that mainly focuses on the automatic generation and ground-truth annotation of synthetic data by simulating the real world in a virtual environment. Although a virtual environment allows us to workaround the previously



(a) (b) (c) (d) (e)

Figure 2.1: **(a)** Object detection **(b)** Object localization **(c)** Multiple object localization **(d)** Semantic segmentation **(e)** Instance segmentation.

mentioned restrictions, there is still a reality gap that must be covered in order for the synthetic data to be transferred to real life situations. Most synthetic data scenarios present discrepancies between them and the real world, to overcome this and properly transfer the knowledge to real problems there are three known approaches that have been proven to be effective:

- **Photorealism:** This is the most intuitive approach and the main idea behind it is to simply generate extremely realistic environments as close to reality as possible. To achieve this, multiple techniques can be applied and the current state of the art in this field has grown substantially in the last decade. Such techniques include rendering very high and photo-realistic textures, models and lightning or simulating the noise of real cameras by applying filters and post-process effects.

  One of the most recent and promising innovations is real-time ray tracing, which is a technique that substitutes the traditional rasterization step of the classic rendering pipeline, Figure 2.2 illustrates both pipelines. Instead of discretizing the scene and assigning the pixel values, ray tracing simulates the behavior of the lightning by casting (tracing) the path of light as pixels, simulating effects such as reflection, refraction and scattering. This allows for higher precision in the pixel values since it takes into account all of the materials of the scene where the light bounced.

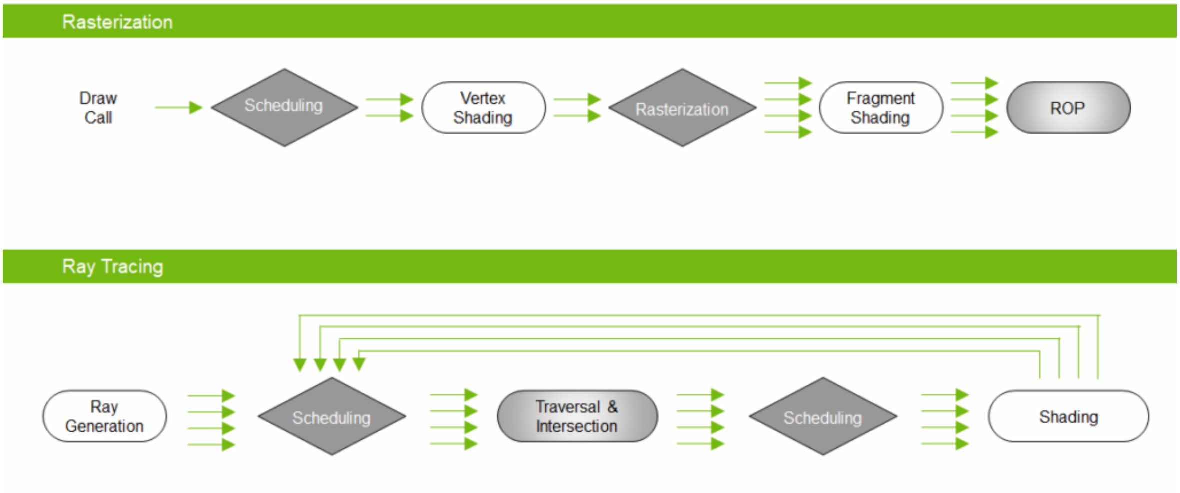


Figure 2.2: Traditional rasterization pipeline in contrast to the ray tracing pipeline. Extracted from NVIDIA devblog[1].

Normally, ray tracing techniques are performed offline since they are quite demanding in terms of computation time. However, the recent RTX NVIDIA Graphic cards [2] feature a new type of processing unit, the RT cores, which specialize in ray tracing computing and are able to accelerate such process, allowing for real time ray tracing. Although this technique is still relatively recent, it is very promising.

- **Domain randomization:** This is the main alternative to photorealism (Tobin et al., 2017) when trying to cover the reality gap between the real world and the synthetic environments. This technique is based on feeding the model a large amount of randomized variations of certain synthetic object or environment. This is done by randomizing certain inputs like the materials, lightning, animations or meshes. With this method even if the data is not represented with extreme fidelity, the variability of the multiple range of slightly different samples will make up for it. With enough randomization, the real world sample will appear as just another variation, which will allow the model to generalize. Figure 2.3 shows an example of randomized training inputs.

- **Domain Adaptation:** This is another interesting approach to reduce the reality gap presented by Y. Chen et al. (2018). The main takeaway behind this idea is to transfer the real-world style of images into the synthetic ones. This is done by an architecture based on the Generative Adversarial Network (GAN) (see Figure 2.4). First, the *Image Transform Network* outputs a new generated image by using as inputs the synthetic RGB image, the segmentation mask and also the depth map. The *Image Discriminator* has to differentiate between the real image and the one generated by the network.

  Then, the *Task Network* performs a second adaptation pass at the output level, predicting the outputs of both real and synthetic images. A second discriminator has to discern if the outputs are predicted from a real or the synthetic adapted image.

[1] `https://devblogs.nvidia.com/vulkan-raytracing`
[2] `https://www.nvidia.com/es-es/geforce/20-series/`

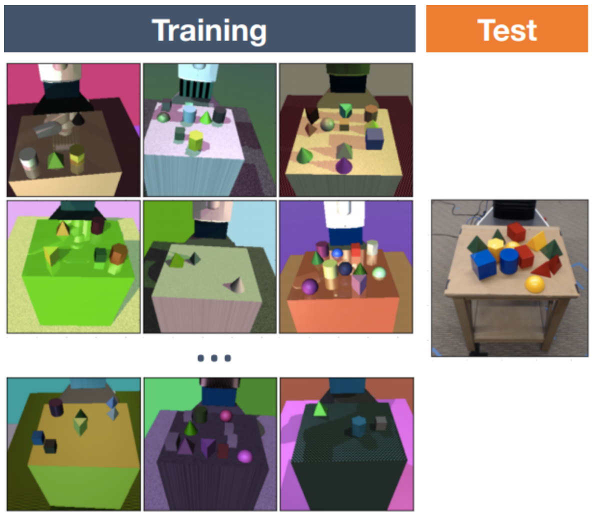


Figure 2.3: Low-fidelity images with random variations in camera angle, lightning and positions are used to train an object detector. Testing is done in the real world. Image from Tobin et al. (2017).

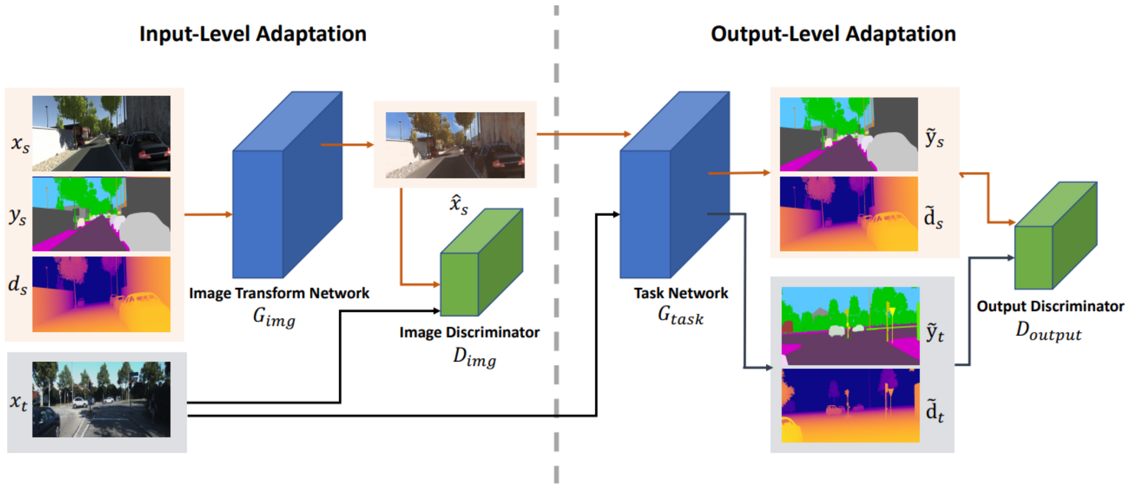


Figure 2.4: Overview of the Input-Output adaptation network form Y. Chen et al. (2018)

2.2.1 VirtualHome

VirtualHome by Puig et al. (2018) is a three-dimensional environment built in the Unity game engine. The main goal is to model complex tasks in a household environment as sequences of more atomic and simple instructions.

In order to perform this task, a big database describing activities composed by multiple atomic steps is necessary, as in the human natural language there is a lot of information that is common knowledge and is usually omitted, however, for a robot or agent this information has to be provided in order to fully understand a task. For this purpose an interface to formalize this tasks was built on top of the Scratch [3] MIT project. Then all of this atomic actions and interactions were implemented using the Unity3D game engine.

For the data collection, they crowdsourced the natural language description of these tasks and then built them using the Scratch interface. Every task is composed by a sequence of steps where every step is a Scratch block, and every block defines a syntactic frame and a list of arguments for the different interactions that they may have.

Every step t in the program can be written as:

$$step_t = [action_t](object_{t,1})(id_{t,1})...(object_{t,n})(id_{t,n})$$

Where id is an identifier to differentiate instances of the same object. An example program to "watch tv" would look like this:

$step_1$ = [Walk] (TELEVISION)(1)
$step_2$ = [SwitchOn] (TELEVISION)(1)
$step_3$ = [Walk] (SOFA)(1)
$step_4$ = [Sit] (SOFA)(1)
$step_5$ = [Watch] (TELEVISION)(1)

Another example this time with the scratch block interface can be seen in Figure 2.5.

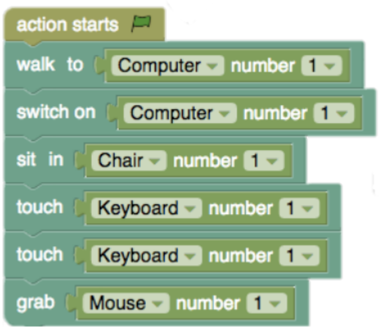


Figure 2.5: List of actions represented with the scratch interface, where the user can manually add, modify and change the arguments of every action.

[3] https://scratch.mit.edu/

2.2.2 UnrealROX

UnrealROX by Martinez-Gonzalez et al. (2018) is a photo-realistic 3D virtual environment built in UE4 capable of generating synthetic, ground-truth annotated data. Unlike VirtualHome, the main method to record sequences is to actually control the actors manually making use of the VR headset and controllers, this will be further explained in the following chapters.

2.3 Common Architectures

As we previously stated, semantic segmentation is a natural step towards the more fine-grained image recognition problem, since the information that we are trying to infer is of a higher level, we also require more complex architectures. Although the models we review in this section work properly for image recognition and detection, some modifications have to be made in order to adapt them for segmentation problems. However, they have made such significant contributions to the field that they are still used as the basic building blocks of segmentation architectures.

2.3.1 AlexNet

AlexNet by Krizhevsky et al. (2012) was the first deep network architecture that successfully surpassed traditional machine learning approaches, winning the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012 with a 84.6% TOP-5 test accuracy, surpassing its competitors by a considerable margin. The architecture itself consists of five convolution + pooling layers followed by three fully connected ones as seen in Figure 2.6.

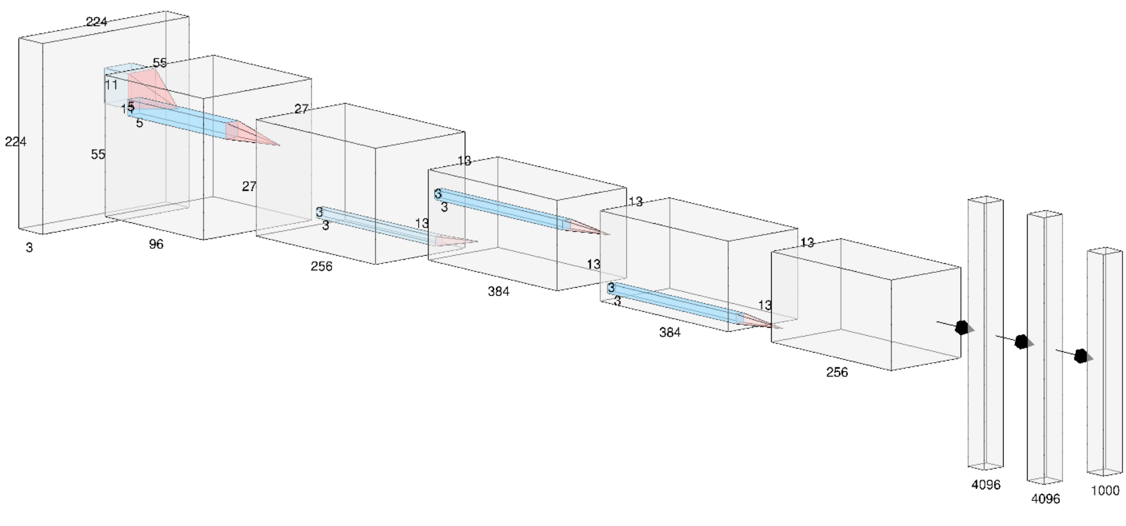


Figure 2.6: AlexNet architecture reproduced from Krizhevsky et al. (2012)

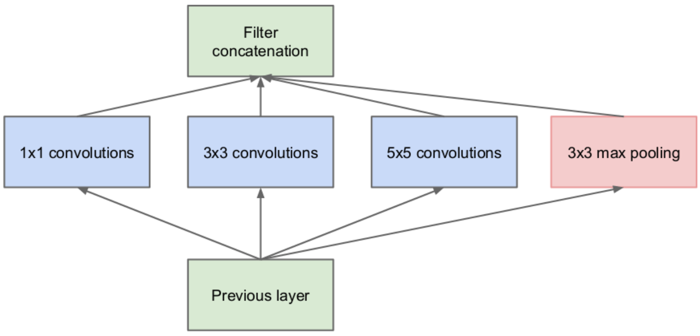


Figure 2.7: Inception module extracted from Szegedy et al. (2014)

2.3.2 VGG

VGG by Simonyan & Zisserman (2015) is also a deep network model introduced by the Visual Geometry Group (VGG), one of the various model configurations proposed was submitted to the ILSVRC 2013, concretely VGG-16, which achieved 92.7% TOP-5 test accuracy.

The structure of VGG-16 consists of 16 convolutional layers, and just like AlexNet, uses three fully connected layers for classification. The main improvement over AlexNet was made by substituting the first large kernel sizes in the first few layers with multiple 3x3 sequential kernel filters.

2.3.3 GoogLeNet

GoogLeNet (also known as Inception) was introduced by Szegedy et al. (2014) and was submitted to ILSVRC 2014, winning with a TOP-5 test accuracy of 93.3%. The architecture of GoogLeNet introduced the inception module (shown in Figure 2.7) which is a new approach where convolution layers were not stacked in just sequential order but instead had some of them computed in parallel, which substantially reduced computational cost. The outputs of the different layers where then concatenated and moved towards the next module.

Ever since their first version, Inception v1, they have been constantly releasing new iterations of the network with constant performance improvements, up to their last Inception v4 release.

2.3.4 ResNet

ResNet by He et al. (2015) was first introduced by Microsoft research and it received increasing attention after winning ILSVRC 2015 with a TOP-5 test accuracy rate of 96.4%. This CNN has 152 layers (although shallower variations do exist) and it introduced a new concept called residual blocks. Its greater number of layers makes the network more prone to the vanishing gradient problem (the backpropagated gradients gets infinitely small and the network performance falls off). The new residual module allowed the network inputs to skip layers and copy the values onto deeper layers, in a way that the computed output is a combination of both the skipped inputs and the forward propagated ones (see Figure 2.8).

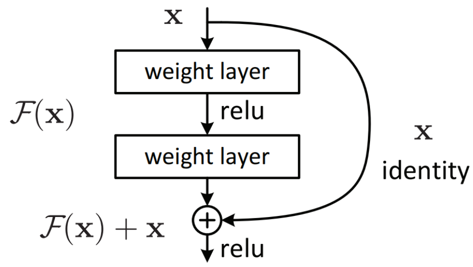


Figure 2.8: Residual block extracted from He et al. (2015)

This helps to reduce the vanishing gradient problem in deep networks and allows each layer with a residual input to learn something new since the inputs are both the encoded values from the previous layer as well as the unchanged inputs.

2.3.5 ReNet

The Multi Dimensional Recurrent Neural Network (MDRNN) by Graves et al. (2007) is a variation of regular Recurrent Neural Networks (RNN) that allows them to work with d spatio-temporal dimensions of the data. However, the architecture proposed by Visin et al. (2015) used regular RNNs instead of MDRNN, the main intuition behind their proposal is that every convolution + pooling layer is replaced by four RNNs that sweep the image across in four directions.

2.3.6 Semantic Segmentation Methods

All the image recognition approaches are based on convolutional architectures, whether it is recognition, detection, localization or segmentation, they all share a big common module, which is the convolution layer that extract the features of any image, then the feature maps can be applied to any classification network structure depending on the desired output format.

Today, almost every semantic segmentation architecture is based on the Fully Convolutional Network (FCN) by Long et al. (2014). The idea behind this is to replace the classic fully connected layers of CNNs with FCNs in order to obtain a spatial map instead of classification outputs. This way we can obtain pixel-wise classification while still using the inferred knowledge and power of the CNNs to extract the features. However, a new problem arises when using CNNs for semantic segmentation, since convolutional layers (convolution + pooling) down-sample the input image in order to learn features. This down-sampling does not matter when applied to classification problems, however, when doing pixel-wise classification, we require the output image to have the same size as the input. To overcome this problem, spatial maps are then up-sampled by using deconvolution layers as shown in Zeiler et al. (2011).

2.3.6.1 Decoder Variant

The decoder variant is another method to adapt networks that were initially made for classification. In this variant, the network after removing the fully connected layers is normally called encoder and it outputs a low-resolution feature map. The second part of this variant is called decoder and the main idea behind it is to up-sample those feature maps to obtain a full resolution pixel-wise classification.

One of the most known examples of this encoder-decoder architecture is SegNet by Badrinarayanan et al. (2015), the encoder part is fundamentally the same as a VGG-16 without the fully connected layers at the very end, while the decoder part consists of a combination of convolution and up-sampling layers that correspond to the max-pooling ones in the encoder, the whole architecture can be seen in Figure 2.9. SegNet is capable of achieving very good results while being relatively fast, which makes it a good starting point for any semantic segmentation problem.

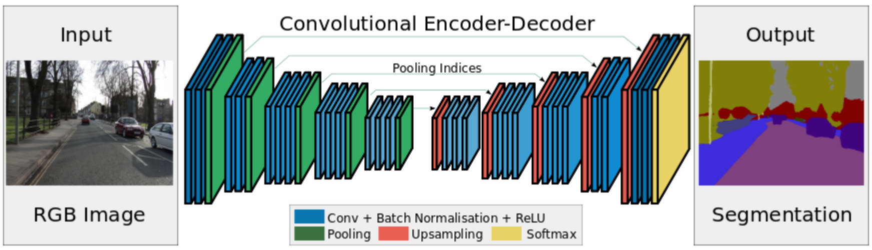


Figure 2.9: Segnet architecture graph extracted from Badrinarayanan et al. (2015)

2.3.6.2 Dilated Convolutions

As we previously mentioned, CNNs generate significantly reduced spacial feature maps. To overcome this spatial reduction, dilated convolutions (also known as *atrous* convolutions) can be used in order to aggregate multi-scale contextual information without down-scaling.

The dilation rate l controls the up-sampling factor of the filters. That way, a 1-dilated convolution is just a regular convolution where every element has a receptive field of 1x1, in a 2-dilated every element has a 3x3 receptive field, in a 3-dilated every element has a 7x7, this is depicted in Figure 2.10. This way, the receptive field grows in a exponential way, while the parameters have a linear growth.

Some of the most important works that make use of this technique are the aforementioned

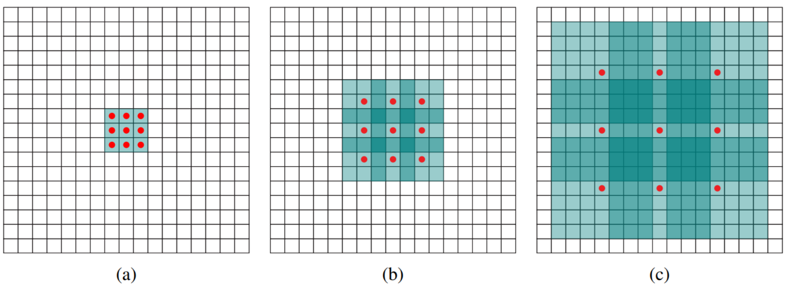


Figure 2.10: (a) 1x1 receptive fields, 1-dilated, (b) 3x3 receptive fields, 2-dilated, (c) 7x7 receptive fields, 3-dilated. Figure extracted from Yu & Koltun (2015).

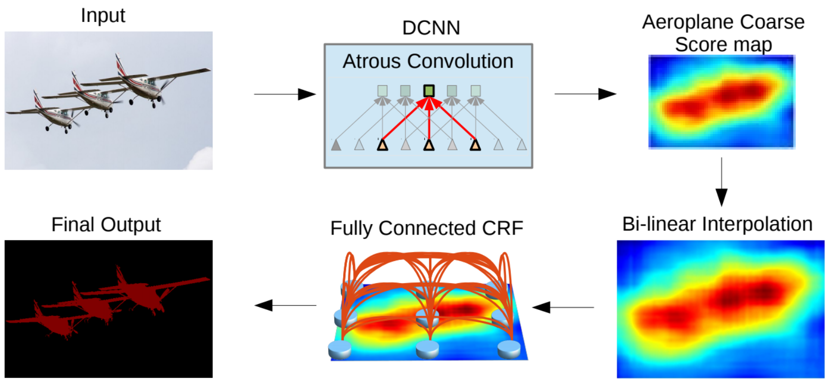


Figure 2.11: Illustration of the DeepLab proposed architecture, using a deep CNN for pixel-wise classification and a fully connected CRF to refine the output.

multi-context aggregation by Yu & Koltun (2015) and DeepLab by L. Chen et al. (2016).

2.3.6.3 Conditional Random Fields

Deep CNNs applied to semantic segmentation excel at classification tasks, however they still lack precision when it comes to spacial information and struggle to properly delineate the boundaries of objects. To overcome this, a last post-processing step in order to refine the output can be applied, for instance, Conditional Random Fields (CRF). This technique makes use of both low level pixel interaction as well as the multi-class inference pixel prediction of high level models.

The DeepLab model proposed by L. Chen et al. (2016) makes use of CRF to refine their output. An overview of the model can be seen in Figure 2.11.

2.4 Datasets

In this section we will review some of the most important datasets that are commonly used to train semantic segmentation architectures.

2.4.1 PASCAL

PASCAL Visual Object Classes by Everingham et al. (2015) is one of the most popular 2D benchmark for semantic segmentation. The challenge consists of 5 different competitions, 21 ground-truth annotated classes and a private test set to verify the accuracy of submitted models. Also there are a few extensions of this dataset. One of them is PASCAL Context, which provides pixel-level classification for the entire original dataset. Another extension for the PASCAL dataset that is worth mentioning is PASCAL Part, which further decomposes the instances in smaller classes. For instance, a car is decomposed into wheels, chassis, headlights and windows. Figure 2.12 shows more examples of different classes.

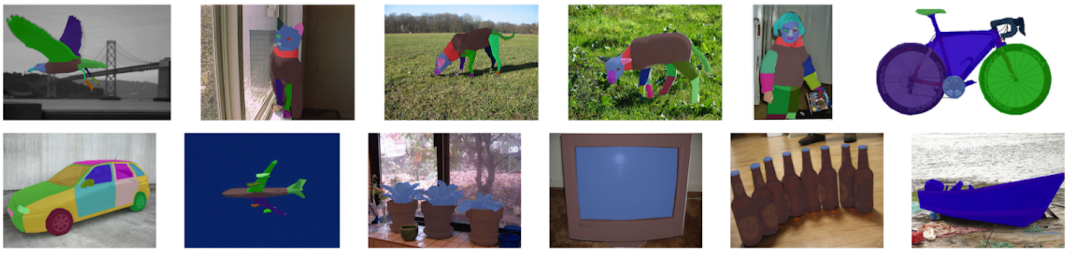


Figure 2.12: PASCAL Part examples of ground truth annotated parts for different classes.

2.4.2 Semantic Boundaries Dataset

SBD by Hariharan et al. (2011) is an extension of the PASCAL dataset that provides semantic segmentation ground-truth annotations for all the images that were not labeled in the original dataset. These annotations contain class, instance and boundaries information. SBD greatly increases the amount of data from the original PASCAL and, because of this, is commonly used for deep learning architectures.

2.4.3 Cityscapes

Cityscapes by Cordts et al. (2016) is a urban dataset mainly used for instance and semantic segmentation. It contains over 25000 images and 30 different classes that were recorded in 50 cities during different times of the day and year.

2.4.4 KITTI and Virtual KITTI

The KITTI dataset by Geiger et al. (2013) was recorded from a vehicle on an urban environment. It includes RGB images, laser scans, and precise GPS measurements. Despite being very popular for autonomous driving, it does not contain ground-truth annotations for semantic segmentation. To work around this, some researchers manually annotated parts of the dataset to fit their necessities.

Virtual KITTI by Gaidon et al. (2016) is a synthetic dataset based on KITTI. In their work, they propose an efficient real-to-virtual world cloning method, which was used in order to generate their dataset. Virtual KITTI offers highly accurate ground-truth for object detection, semantic and instance segmentation, depth and tracking.

2.4.5 COCO

COCO is another image recognition and segmentation dataset by Lin et al. (2014) which mainly focuses on everyday scenes and common objects. The dataset contains 91 different classes and a total of 328.000 images and the labeling methods contain both bounding boxes as well as semantic segmentation.

3 Materials and Methods

In this chapter we go over the different materials and methods that were considered in this work. It is organized as follows: Section 3.1 analyzes some of the software specification used in this thesis, focusing on the 3D Game engine framework and working environments. Then, in Section 3.2 we present our hardware equipment used for this work.

3.1 Software

In order to carry out this project, it was necessary to carefully choose our working environments and programming tools. In this section, we go through some of our software of choice as well as giving a brief explanation on why it was chosen.

3.1.1 Unreal Engine 4

UE4 is a very powerful, highly portable game engine, written in C++ and developed by Epic Games[1]. The main advantages UE4 offers over other game engines and the reason UnrealROX was built using it are listed as follows:

- **Virtual reality support:** VR was a key point when developing the ROX framework since one of the main goals was to allow the user to completely interact with the environment.
- **Photorealism:** Realism is a key factor when it comes to synthetic data and the potential of UE4 to run extremely realistic scenes in real time, such as the one shown in Figure 3.1, made it suitable for this purpose.
- **Blueprints:** Blueprints are one of the tools that UE4 offers, they allow for quick behavior definitions within the editor. This makes it apt for prototyping and testing.
- **Community:** UE4 is currently one of the most popular game engines and it has a rather populous community, the official forums and other platforms are very active and the documentation is well maintained. The developing team is heavily involved and they continuously release new versions and bug fixes.

[1] https://www.unrealengine.com/en-US/
[2] https://ue4arch.com/projects/viennese-apartment/

Figure 3.1: Snapshot of the Viennese Apartment by UE4Arch[2]

3.1.2 Visual Studio 2017

Visual Studio is an Integrated Development Environment (IDE) developed by Microsoft and used for software development. It supports a variety of programming languages, although it mainly focuses on C++, C# and the .NET framework. It includes a code editor, file browser and debugger. It also includes plugins, support for syntax highlighting and integration with IntelliSense, which allows for code completion, quick information of variables and methods, amongst other features.

However, the main reason Visual Studio was chosen as the main IDE for this project is the integration with UE4. The UE4 editor has options to quickly visualize any object from the context menu or the scene in Visual Studio, allowing to make quick changes, recompile and launch in very little time.

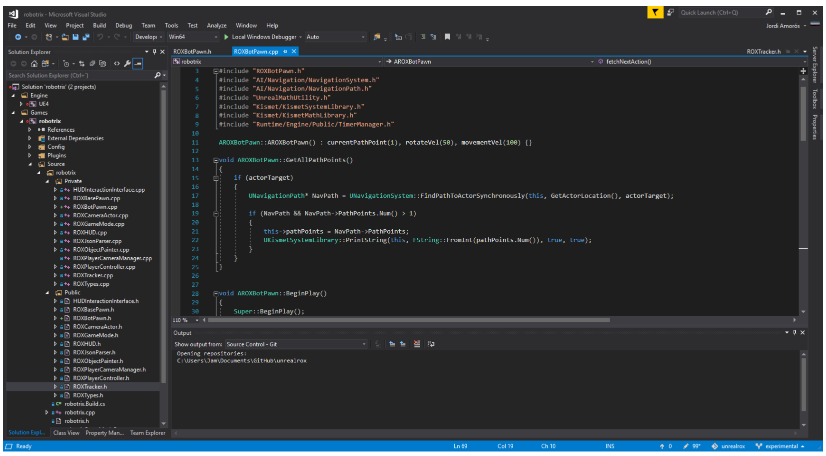


Figure 3.2: Visual Studio IDE.

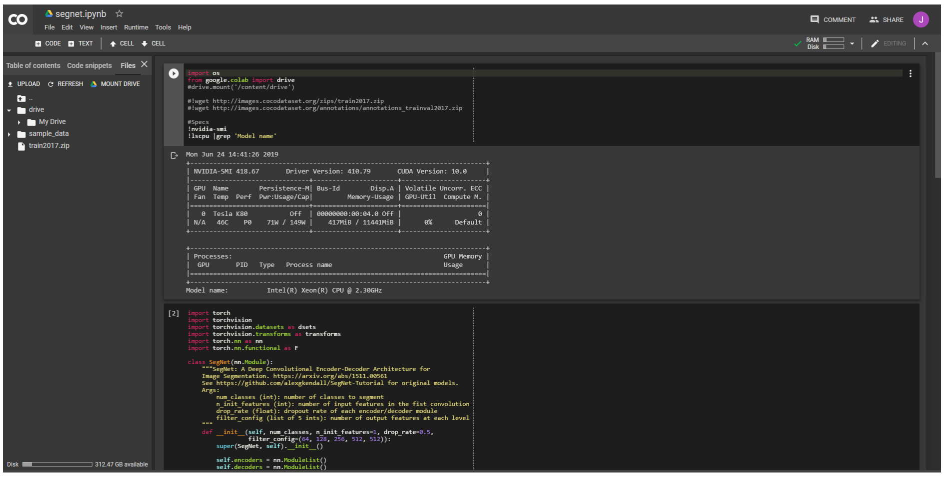


Figure 3.3: Google Colab web interface.

3.1.3 Google Colab

Google Colaboratory[3] is a free Jupyter[4] notebook environment that runs entirely on the cloud, requires no setup and is powered by Google. Jupyter notebook is a cell-based environment that allows to create and share documents containing live code, equations, images and text. It is mostly used with Python, however it has integration with other languages such as R, C++, Scheme or Ruby. Its integration within Google Colab makes it a very easy-to-use tool for prototyping and quick testing. Figure 3.3 shows the main user interface.

Since it is a free environment one may think its not to powerful hardware wise, however this is not the case, it provides more than enough to run deep architectures with ease, although it is not a tool intended for long-running background computation and the system will free resources every 12 hours. The full hardware specification is shown in Table 3.3.

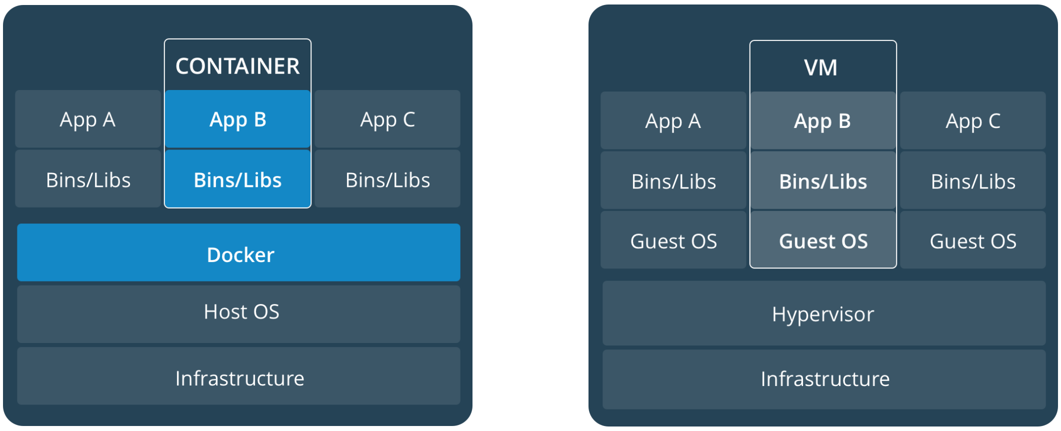
3.1.4 Docker

Docker[5] is a virtualization software that allows to develop, deploy and run applications within containers. The Docker platform is based on images, which are packages containing all the necessary libraries, environment variables and configuration files needed to run a container. These containers are runtime instances of such images and they can run natively on linux, sharing the kernel of the host machine with other containers. Figure 3.4 shows the architectural differences between a conventional VM and Docker.

We used Docker for this thesis since is very lightweight compared to other virtualization software, which allows to run multiple instances on the same machine. This way, we can use any version CUDA, CUDNN and other Deep Learning (DL) libraries without modifying the installed versions on the host operating system.

Docker also allows to bind ports, so the applications running on the container can be accessed from the outside. Specifically, we ran a Ubuntu a 16.0.4 container with a Jupyter

[3] `https://colab.research.google.com/`
[4] `https://jupyter.org`
[5] `https://www.docker.com`



(a) (b)

Figure 3.4: (a) Docker architecture. **(b)** Conventional VM architecture.

notebook server, which is then available via HTTP.

3.1.5 Frameworks

DL has been arising in popularity in the last decade and the most current state-of-the-art Artificial Inteligence (AI) algorithms are based on deep architectures. Because of this, multiple DL frameworks have been developed to ease the low level implementation of these algorithms.

3.1.5.1 TensorFlow

TensorFlow[6] is a open source library for numerical computation based on the idea of data flow graphs. In TensorFlow, the graph nodes represent the mathematical operations, while the edges represent the multidimensional data arrays (or tensors) flowing between them, Figure 3.5 illustrates an example of a flow graph representation.

TensorFlow was created by the researchers at Google Brain for the purpose of conducting machine learning and deep neural network research, its low level nature allows for a very fine-grained framework that can be use to build any architecture from the ground up and the tensor-graph structure also allows for very easy data distribution on the CPU-GPU.

In a first approach, TensorFlow was considered for its use as the main framework for this project, but it was finally discarded since high level frameworks ease the work and a low level implementation of the networks falls out of the scope of this project.

3.1.5.2 Keras

Keras[7] is a high level framework written in Python that can use TensorFlow, CNTK or Theano as backend. It was developed to be an easy-to-use framework, allowing for very fast experimentation and prototyping, abstracting the user from some of the more complex low

[6] `https://www.tensorflow.org/overview`
[7] `https://keras.io`

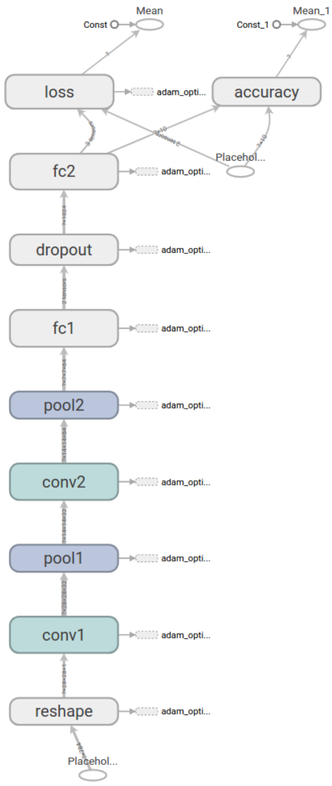


Figure 3.5: TensorFlow graph example depicting a simple two-layer convolution + pooling with a fully connected layer for classification at the end.

level tasks with a very user friendly interface. This also makes Keras a very good entry framework for beginners that still do not have a solid foundation on deep learning.

Keras provides two different Application Program Interface (API)s for different model building approaches. The Sequential API allows to simply stack layer after layer with a very simple and easy-to-use interface. This makes it ideal for models with an input to output data flow. The Functional API, however, provides a more flexible way for defining models. With this API instances of different layers can be created attached to the model, with this approach, more complex and non-sequential models can be defined.

At the start of this project, Keras was used in order to implement simple neural networks with educational purposes since it is a very easy and intuitive framework. In the end, we discarded it since we believe there are better alternatives that allow for more flexibility.

3.1.5.3 PyTorch

PyTorch[8] is an open source, Python-based, GPU-Ready computing package and machine learning framework, just like other frameworks, it provides a Tensor datatype together with all the operations to handle them in both the GPU and CPU. This data structure is also compatible with NumPy and other Tensor libraries, which makes it very compatible and easy

[8] `https://pytorch.org`

to integrate.

Another PyTorch feature that is worth mentioning is the modularity, writing new modules for a Neural Network (NN) is very straightforward and they can be written in both native Python and other NumPy based libraries or with the torch API. It also counts with multiple pretrained networks, datasets and well documented examples.

PyTorch is also in continuous development and has been steadily gaining popularity ever since its release back in 2016. In terms of performance, PyTorch is just slightly behind TensorFlow, and outperforms[9] other high level frameworks such as Keras.

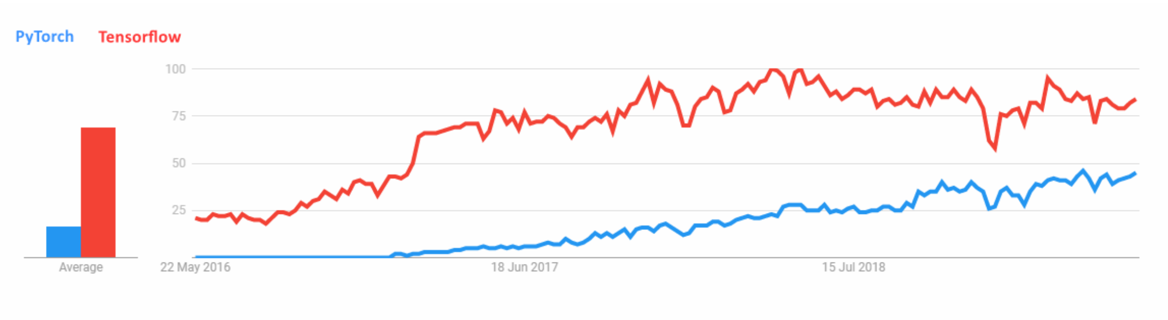


Figure 3.6: Worldwide PyTorch and TensorFlow popularity comparison in Google Search.

PyTorch was the framework of choice for this project since it allows for easy prototyping without losing the flexibility to make architectural modifications to the networks. Its syntax is also very easy for anyone that has experience with Python which made it perfect for this project.

3.2 Hardware

It is widely known how computationally demanding DL algorithms are, specially when dealing with large amounts of data. Also, in order to smoothly run UE4 while recording and generating all the output images we need a mid to high end computer. In this section we review the ones that have been used in this work.

3.2.1 Clarke

The structure of neural networks where multiple data streams are organized in layers allows for very easy parallelization. Because of this GPUs are extremely powerful when executing said algorithms.

The Clarke server was deployed with this in mind and features three different NVIDIA GPUs. The most powerful of them, the Titan X, is aimed towards DL computing, the Tesla K40 is also used for computational purposes. The last of them is a Quadro 2000 that is only used for visualization purposes. The full hardware specification for the Clarke server is shown in Table 3.1.

[9] `https://wrosinski.github.io/deep-learning-frameworks/`

As for the software, Clarke runs Ubuntu 16.04 with Linux kernel 4.15.0-39-generic for x86_64 architecture. It also runs Docker, which allows any user to configure its own container with any CUDA / CUDNN version and DL framework.

It is also worth mentioning that Clarke was configured for remote access using SSH with public/private key pair authentication. The installed versions are OpenSSH 7.2p2 with OpenSSL 1.0.2 and X11 forwarding was configured for visualization purposes.

Clarke	
Motherboard	Asus X99-A Intel X99 Chipset 4x PCIe 3.0/2.0 x 16(x16, x16/ x16, x16/ x16/ x8)
CPU	Intel(R) Core(TM) i7-6800K CPU @ 3.4GHz 3.4 GHz (3.8 GHz Turbo Boost) 6 cores (12 threads) 140 W TDP
GPU (visualization)	NVIDIA GeForce Quadro 2000 192 CUDA cores 1 GiB of DDR5 Video Memory PCIe 2.0 62 W TDP
GPU (deep learning)	NVIDIA GeForce Titan X 3072 CUDA cores 12 GiB of GDDR5 Video Memory PCIe 3.0 250 W TDP
GPU (compute)	NVIDIA Tesla K40c 2880 CUDA cores 12 GiB of GDDR5 Video Memory PCIe 3.0 235 W TDP
RAM	2 x 8 GiB G.Skill X DDR4 2400 MHz CL15
Storage (Data)	(RAID1) Seagate Barracuda 7200rpm 3TiB SATA III HDD
Storage (OS)	Samsung 850 EVO 250 GiB SATA III SSD

Table 3.1: Hardware specification for Clarke.

3.2.2 Personal Computer

During the developing of this work, a personal computer was used in order to run UE4 and UnrealROX, as well as to generate the data used for the deep learning experimentation. Table

3.2 shows its full hardware specification.

	Personal Computer
Motherboard	Asus STRIX X370-F Amd X370 Chipset 2 x PCIe 3.0/2.0 x16 (x16 or dual x8)
CPU	AMD Ryzen™ 5 1600 CPU @ 3.2GHz 3.2 GHz (3.6 GHz Turbo Boost) 6 cores (12 threads) 140 W TDP
GPU	NVIDIA GeForce GTX960 1024 CUDA cores 2048 MiB of DDR5 Video Memory PCIe 3.0 120 W TDP
RAM	2 x 8 GiB G.Skill Trident Z DDR4 3200 MHz CL15
Storage (Data)	Seagate Barracuda 7200rpm 2TiB SATA HDD
Storage (OS)	Samsung 960 EVO 250GiB NVMe M.2 SSD

Table 3.2: Hardware specification for the personal computer.

3.2.3 Google Colab

As explained in Subsection 3.1.3, the Google Colaboratory environment was used in the prototyping and testing process. The hardware specification where this environment runs is shown in Table 3.3. Since Colab is run in the cloud and assigns the user a virtual machine, the exact specifications are not known, although it is enough to get an idea of its computational power.

	Google Colab
CPU	Intel(R) Core(TM) Xeon CPU @ 2.3GHz 2.3 GHz (No Turbo Boost) 1 core (2 threads) 45MB Cache
GPU	NVIDIA Tesla K80 2496 CUDA cores 12 GiB of GDDR5 Video Memory PCIe 3.0 300 W TDP
RAM	12.6 GiB
Storage (Data)	320 GiB

Table 3.3: Hardware specification for Google Colab instances.

4 Data Generation and Semantic Segmentation

This chapter develops the main work of this thesis and is organized as follows. Section 4.1 describes the process of automating the UnrealROX Actor. Section 4.2 goes through the process of recording sequences. Finally, Section 4.3 covers the deep learning work such as the network implementation and the data pre-processing.

4.1 Expanding the UnrealROX Framework

As we previously mentioned in Chapter 1, one of the main goals of this work is to expand the UnrealROX framework in order to automatize the generation of synthetic data without the need of a VR Headset and user input. In this section we further detail the framework itself along with the data generation process.

UnrealROX can automatically generate and annotate data from a recorded sequence, but manually recording can be tedious and time consuming. Additionally, UnrealROX was mainly oriented towards first-person interaction, so it lacks third-person tools to generate data. In this work we have built the basic framework for the programmer to include their own actions and execute them in a sequential way, much like other frameworks such as VirtualHome by Puig et al. (2018).

4.1.1 The ROXBasePawn Class

This is the main class that contains all the logic for the character controller (movement, animations, grasping) of any robot pawn. It allows for the user to introduce a robot to the scene and manually move and interact with the objects in a scene. We use this as our parent class for the implementation.

4.1.2 The ROXBotPawn Class

The ROXBotPawn class inherits from ROXBasePawn and handles all the logic for the automation of tasks of any *Actor* within a scene. In order to model all the different actions and interactions, the Enum *EActionType* was created, where the programmer can add any type of action to be built into the system.

Also, in order to model the actions themselves, the *FROXAction* struct was built, containing a pointer to the target, as well as the type of action *EActionType*. This structure is shown in Listing 4.1.

Listing 4.1: FROXAction struct

```
USTRUCT(BlueprintType)
```

```
struct FROXAction
{
    GENERATED_USTRUCT_BODY()
    UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = Pathfinding)
    AActor* target;

    UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = Pathfinding)
    EActionType action;

    FROXAction() : target(nullptr), action(EActionType::MoveTo) {};
    FROXAction(AActor* tg, EActionType t) : target(tg), action(t) {};
}
```

In order for the programmer to add actions and queue them from the UE4, editor we built the $doAction(AActor*, EActionType)$ (seen in Listing 4.2) and made it BlueprintCallable, this way, in a simple manner, actions can be queued from the editor and the *Pawn* will execute them in a sequential order as seen in Figure 4.1. The target actor can be picked from the editor by creating a new variable and the type of action can be selected with a drop-down menu in the body of the blueprint function.

Listing 4.2: doAction function which queues a new FROXAction to the system

```
UFUNCTION(BlueprintCallable)
void AROXBotPawn::doAction(AActor* actor, EActionType type)
{
    actions.Add(FROXAction(actor, type));
}
```

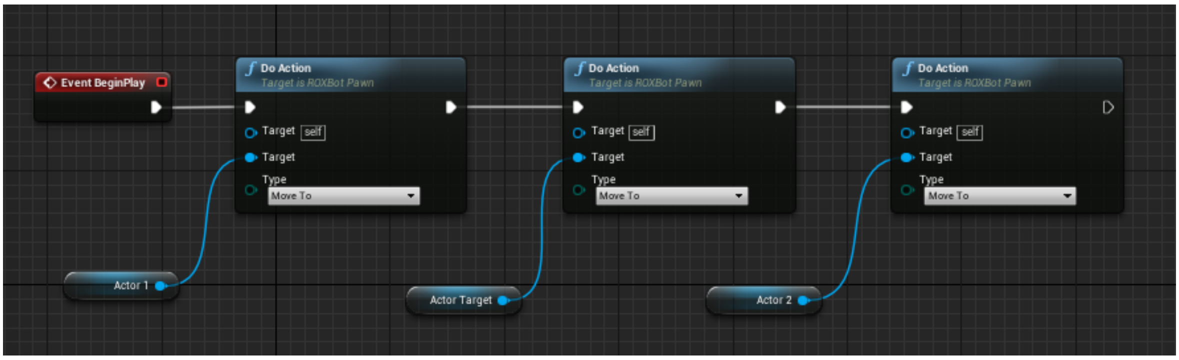


Figure 4.1: Example of queuing 3 "MoveTo" actions from the editor

The $doAction()$ method creates a $FROXAction$ and pushes it to the queue. Whenever the queue contains an action and the *Pawn* is not executing one, it will run the $fetchNextAction()$ method. This will pop the action from the queue and execute the method corresponding to that action.

Additionally, pathfinding and movement logic was implemented in order to define the $MoveTo$ action. In a first approach, the MoveToLocation method of the default UE4 Actor class was used, but the idea was discarded since we needed to work with the ROXBasePawn instead of the default Actor. Another option that was studied was Environment Query Systems (EQS), which is an experimental feature within the AI system in UE4 which allows to gather data from the environment, e.g., the distance from one object to another or whether they are in line of sight or not. With this data we can perform queries and move accordingly to our goal, however, this method was finally discarded since the complexity of the technology was greater than that of the task we needed to solve. This is due to the fact that we only need the position of a certain actor in order to travel towards them, and this can be achieved without the need of more complex objects.

With the previous options discarded, we decided to go with a custom implementation of the movement. In order to accomplish this, the NavigationMesh component of UE4 was used along with the *FindPathToActorSynchronously* method, which returns a *UNavigationPath* containing all the path-points from one actor to another. Once we obtain the path-points the *VInterpConstantTo* from the FMath library and the *RInterpTo_Constant* from the UKismeMathLibrary are used in order to obtain the next vector transformation for both position and rotation of the *Pawn*, this movement logic can be seen in Listing 4.3. These methods interpolate the current location with the next path-point location in order to achieve a smooth transition and make the movement more natural.

Listing 4.3: Movement logic for the pathfinding algorithm

```
FVector nextPos = FMath::VInterpConstantTo(this->ActorLocation(), FVector(pathPoints[i].X, pathPoints[i].Y, ActorLocation().Z), DeltaTime, vel);
FRotator nextRot = UKismetMathLibrary::RInterpTo_Constant(ActorRotation(), UKismetMathLibrary::FindLookAtRotation(ActorLocation(), nextPos), DeltaTime, rotateVel);
SetActorRotation(nextRot);
SetActorLocation(nextPos);
```

4.1.3 Animating the ROXBotPawn

As we previously mentioned in Section 2.2, simulating the 3D environment with extreme detail is a must in order for DL algorithms to properly infer the knowledge and transfer it to the real world. In this section we take a look into the process of creating a new animation for the ROXBotPawn using UE4 blueprints.

The ROXBasePawn already provides a default walking animation for our Actor, however it is thought to work with a VR Headset, therefore, it takes into account the pose and movement of both hand controllers and the headset itself in order to move accordingly to the user. Since our ROXBotPawn does not require such data, we will change the blueprint animation in order to fit our needs.

First of all, we need animation assets, in our case we will be using the default walk and idle animation from the UnrealROX framework since they fit our needs. In order to have smooth transitions between different animations, e.g., from idle to walking, we have used UE4 BlendSpaces, which are special assets that can be sampled in the Animation Graph and allow for blended transitions based on one or more inputs. In our case, we will blend the animation based on the speed of the Bot. A preview of said asset can be seen in Figure 4.2.

In order to use the BlendSpaces, we need to create our Event Graph and compute the Bot speed, we do this by obtaining its position in the current and last tick, therefore obtaining the travel distance in one tick. We can now apply the dot operator with the forward vector and divide by the delta time, obtaining the current speed. The complete logic of this function as well as the full Event Graph can be seen in Figure 4.3.

With the event graph and the BlendSpaces created, all there is left to do is to create the state machine, which contains the state and transition logic of the different animations. In our case, we just need an idle, walk forward and walk backwards states. This state machine

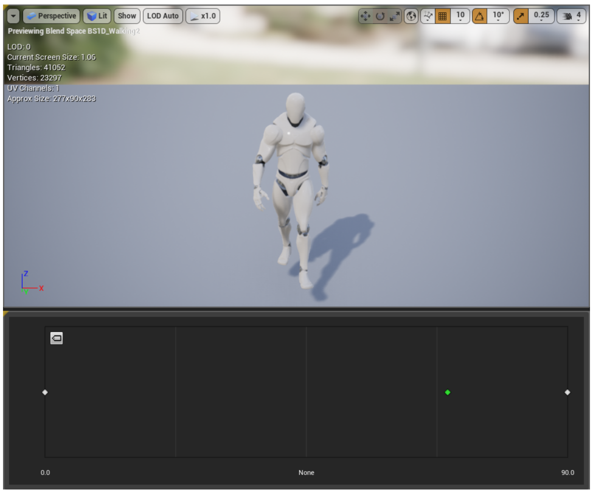
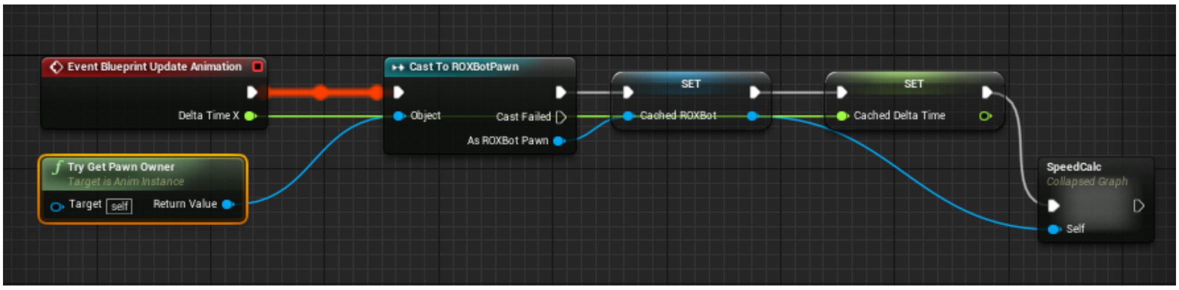
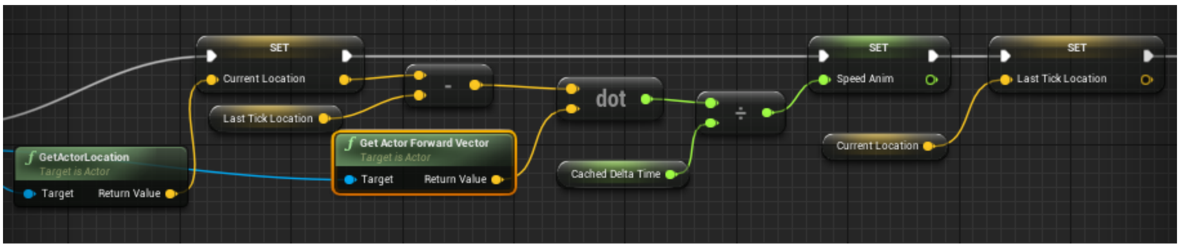


Figure 4.2: Blend Space asset which samples the transition from idle walking state animation, 0 speed would translate into a complete idle, while 90 would be walking forward.



(a) Animation event graph.



(b) SpeedCalc sub-module.

Figure 4.3: a) Event graph that obtains the data of the Bot. b) Blueprint sub-module which computes the instantaneous speed of the Bot.

is shown in Figure 4.4.

The logic for the transitions between states is implemented by checking whether the speed surpasses a certain threshold. For instance, if the *SpeedAnim* is greater than 0.1, we will transition from the *Idle* state to the *WalkForward*. In the same manner, when the speed value falls under negative 0.1, we will transition towards the *WalkBackwards* state.

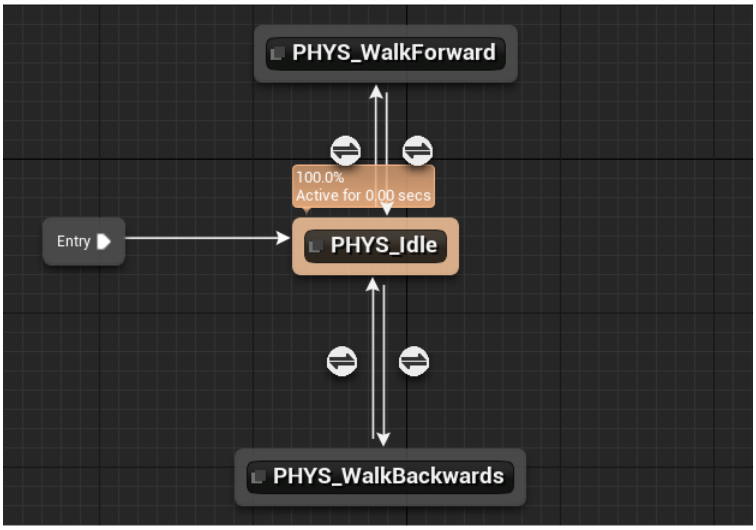


Figure 4.4: Animation state machine with idle, walk forward and walk backwards states and the transitions logic.

4.2 Recording sequences with UnrealROX

In this section we go through the ROXTracker class and how to use it in order to record sequences (Subsection 4.2.1) as well as how to play them and generate the data (Subsection 4.2.2).

The ROXTracker is an empty *Actor* which means that it has no mesh or physical appearance in the world. However, this actor has knowledge of the whole scene and is able to determine the pose, rotation and materials of every other *Actor* of the scene. In order to use it, we just need to search for it in the contextual menu (as shown in Figure 4.5) and drag it into our scene. While in record mode, the ROXTracker is be able to store all of the information needed in order to rebuild the sequence as a TXT or JSON file. This information is then parsed in order to run the sequence in playback mode. While the sequence is being reproduced, the ROXTracker generates frame by frame the ground truth annotated images such as the segmentation masks, depth and normal maps.

Once we have our ROXTracker in the World Outliner, we can tweak its behavior and change certain settings, some of the most important ones are listed as follows:

- **Record Mode:** When checked, the ROXTracker will operate in record mode, this means that if the user presses the record key, it will start gathering and writing all of the necessary data to a TXT file.
- **Scene Save Directory:** As its own name implies, this variable stores the path where the data will be saved.
- **Scene Folder:** Name of the folder inside the Scene Save Directory path where all the data files will be stored.
- **Generate Sequence Json:** When pressed, will look for a TXT file inside the Scene Folder with a name corresponding to the field **Input Scene TXT File Name**. Then

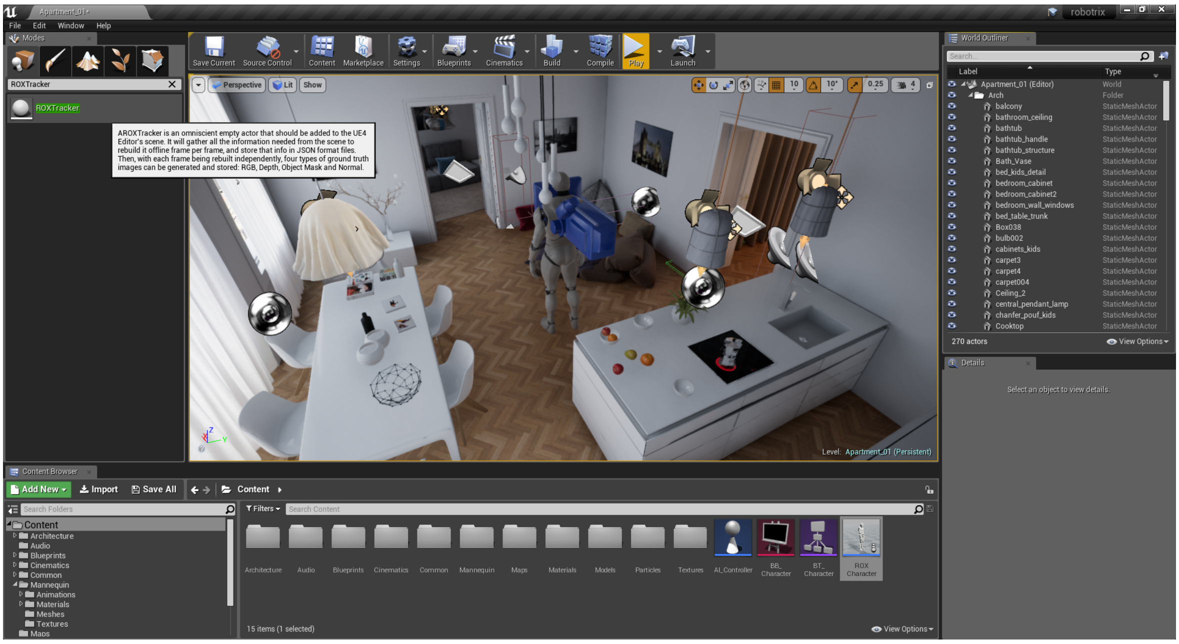


Figure 4.5: ROXTracker Object in the UE4 contextual menu.

it will generate its equivalent JSON file with the name on the field **Output Scene Json File Name**. In other words, looking at Figure 4.6, the Tracker will search for a *scene.txt* file inside the *unrealrox/RecordedSequences* folder and generate a *scene.json* file.

Figure 4.6: ROXTracker settings in the UE4 editor.

4.2.1 Recording mode

Before we start generating sequences, we need to make some tweaks to the recording settings of the ROXTracker, which can be seen in Figure 4.7 and further explained as follows:

- **Pawns:** Array that contains a reference to every actor that the user wants to keep track of.
- **Camera Actors:** Array containing the cameras that will be tracked.
- **Stereo Camera Baselines:** Array that stores the focal distance (baseline) between the corresponding camera in the **CameraActors** array. It can be left empty if there are none.
- **Scene File Name Prefix:** Every generated file of raw scene data will share this prefix in its filename.

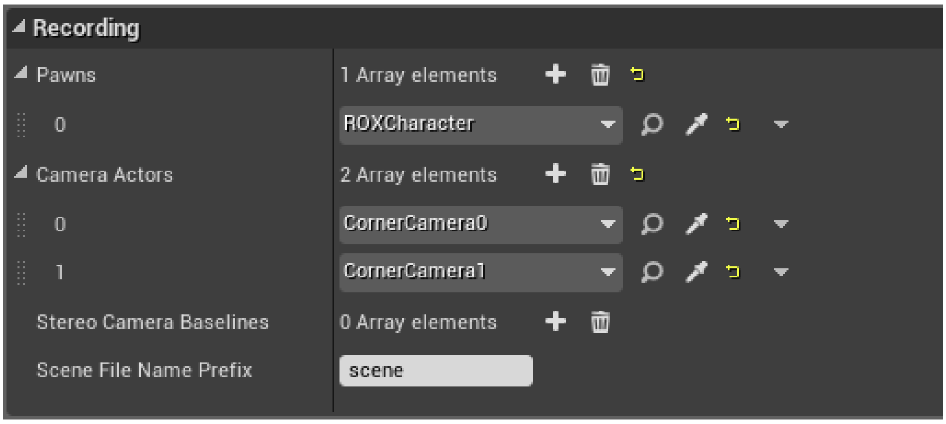


Figure 4.7: ROXTracker recording settings.

Once all of the fields are filled with the desired actors and cameras to be tracked and the filenames are set, we can start recording a sequence, to do this, we simply have to set the ROXTracker in record mode and run the scene by hitting play on the editor. To begin or stop the recording, the user needs to press ”R”, a red **RECORDING** message will then be displayed at the top of the screen. An example of the recorder running can be seen in Figure 4.8.

When the sequence finishes, we can stop recording. If we take a look at our designated Scene Folder for the data generation, we can see our raw recording data in TXT format. We now need, in order to get it ready for playback, to parse it to JSON, we can do this with the **Generate Sequence Json** utility displayed in Figure 4.6.

4.2.2 Playback mode

Before we are able to play the sequences in the UE4 editor, we have to go through some of the configuration settings for the playback mode, which can be seen in Figure 4.9 and are further explained below:

- **Json File Names:** Array containing all the JSON filenames that we want to playback.
- **Start Frames:** Array that contains the starting frames for each JSON. The index in this array will directly correlate to the one in the **Json File Names** array.

Figure 4.8: Example of a running scene being recorded.

- **Playback Only:** When active, it will only play the sequence, skipping the data generation process.
- **Playback Speed Rate:** As its own name indicates, allows to set the speed of the playback, although it can only be used in **Playback Only** mode.
- **Generate RGB, Depth, Object Mask, Normal:** It will generate the RGB (format can be adjusted in the **Format RGB** option), Depth, Segmentation Masks and Normal maps for each frame and camera.
- **Generate Depth Txt Cm:** Generates an equivalent TXT file to the Depth image, where the depth values are stored as plain text.
- **Screenshot Save Directory:** Base path where the Screenshot folder will be located.
- **Screenshot Folder:** Name of the folder where the screenshots will be saved.
- **Width/Height:** Output resolution of the generated images.

Once the desired configuration is set and the Record Mode is disabled, we can press play in the editor. The ROXTracker object will start parsing the sequence JSON file and generating the output images in our designated folder, an example of the four different outputs is shown in Figure 4.10.

4.3 Implementing a SegNet using PyTorch

As we previously mentioned, one of the main goals of this work is to study how synthetic data can help semantic segmentation algorithms. For this purpose, a SegNet has been implemented and trained with a real-world human-pose dataset. This Section will cover all the data processing, as well as the design and development process of such network. All of the implementations in this section can be found at GitHub[1].

[1] `https://github.com/byFlowee/human-segmentation`

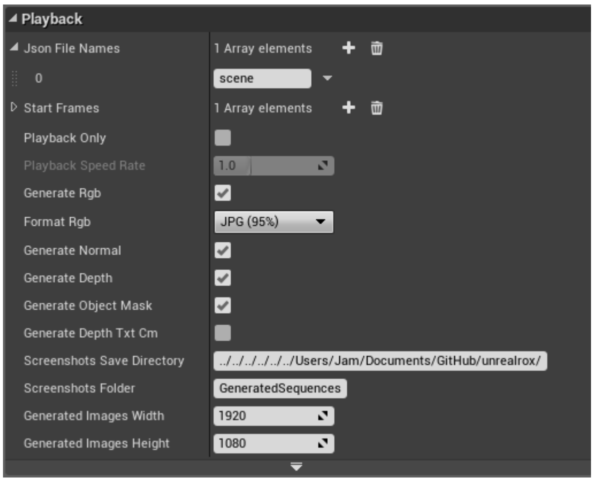
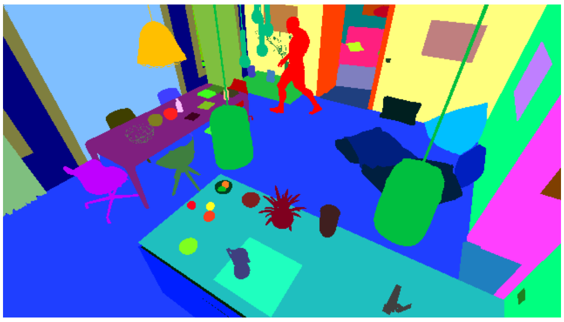


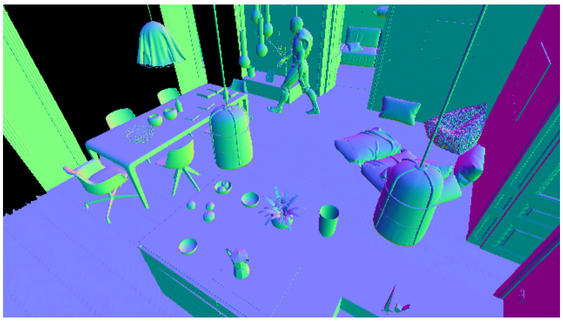
Figure 4.9: ROXTracker playback settings.

(a) Output RGB image.



(b) Output Segmentation Masks.

(c) Output Depth image.



(d) Output Normals image.

Figure 4.10: Output examples of the Tracker in playback mode.

4.3.1 Preprocessing the dataset

In Section 2.4 we mentioned a few of the most important datasets in the field, however, all of them are general purpose oriented, and for this work we needed a human pose dataset. Because of this, we decided to use the UTP dataset by Lassner et al. (2017). Most of the images from this dataset come from the MPII Human Pose Dataset and contains both the

RGB and the Segmentation Mask image.

However, some data pre-processing will be needed in order to fit the data to our needs, this is further detailed in the following Subsections.

4.3.1.1 Merging the segmentation masks

The UTP dataset is divided by segmentation instances. This means that a full image with different persons is divided in different images, each with its unique segmentation mask. For our purpose, we wanted the full image without the instance information. The dataset provides a CSV file which contains the image ID for every segmentation instance in sequential order. This means that we can get all the instances that share the same image ID by iterating through the CSV file. Listing 4.4 shows how we obtain the amount of instances for a single image ID.

Listing 4.4: Obtaining the number of instances for a single image

```
csv = pd.read_csv(csv_path)

while x < len(csv):
    instances = 0

    for id in range(x, len(csv)):
        if csv['mpii_id'][id] == csv['mpii_id'][id+1]:
            instances += 1
        else:
            break
```

Having obtained the amount of instances and with our iterator x pointing to the first image, we can now obtain all the following images and merge them as shown in Listing 4.5. To do this, we use the *Paste* method from the *Pillow* library, which takes two inputs: the image to be pasted and the mask which contains the pixels that are to be copied into the first image. In our case, the mask already fits our purpose since it is the mask that we want to copy.

Listing 4.5: Merging the instance masks into a single image

```
first_image = Image.open(img_path + str(x))

for n in range(1, instances + 1):
    next_image = Image.open(img_path + str(x + n))
    first_image.paste(next_image, (0,0), next_image.convert('L'))

new_id += 1
x += instances + 1
```

4.3.1.2 Creating the dataset class

Before we can train our network, we need to design a data loader class that will fetch our dataset and transform the data to the proper format. For this purpose, we used the PyTorch Dataset class, which allows us to create our data loaders, this way all the data and batch processing will be handled by the framework. Listing 4.6 shows the main structure of the UTPDataset class.

Listing 4.6: UTPDataset definition

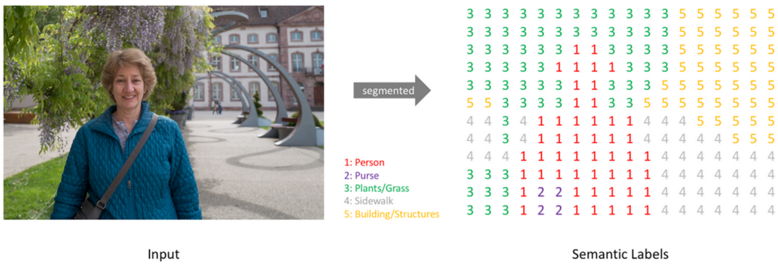
```python
class UTPDataset(Dataset):
    def __init__(self, img_dir, transform=None):
        self.transform = transform
        self.image_root_dir = img_dir
        self.img_extension = '_full.png'
        self.mask_extension = '_segmentation_full.png'

    def __getitem__(self, index):
        image_id = str(index).zfill(5)
        image_path = os.path.join(self.image_root_dir, image_id, self.img_extension)
        mask_path = os.path.join(self.image_root_dir, image_id, self.mask_extension)

        image = self.load_image(path=image_path)
        mask = self.load_mask(path=mask_path)

        data = {
                    'image': torch.FloatTensor(image),
                    'mask' : torch.LongTensor(mask)
                }

        return data
```

The *__getitem__* method returns the rgb-mask pair when given an index. In order to load the image, we compute the filename using the $zfill$ method, this will add leading zeros to the index so it fits the image name. We then call the *load_image* and *load_mask* methods shown in Listing 4.7, which loads and processes the data. Finally, we insert it into a small dictionary and return it.

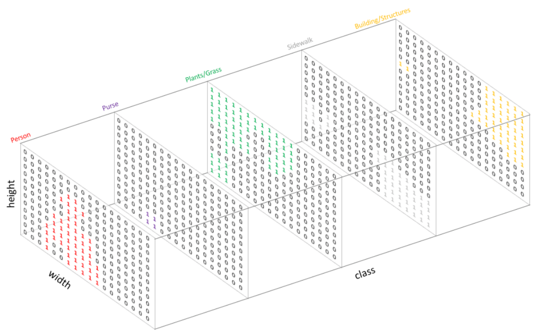
Listing 4.7: UTPDataset rgb and mask load and pre-processing

```python
PALETTE = {
    (0,  0,  0)  : 0,  #Human
    (255, 255, 255) : 1,  #Background
}

def load_image(self, path=None):
    raw_image = Image.open(path)
    raw_image = np.transpose(raw_image.resize((224,224)), (2,1,0))
    imx_t = np.array(raw_image, dtype=np.float32)/255.0

    return imx_t

def load_mask(self, path=None):
    raw_image = Image.open(path)
    raw_image = raw_image.resize((224,224))
    imx_t = np.array(raw_image)
    label_seg = np.zeros((2,224,224), dtype=np.int)

    for k in PALETTE:
        label_seg[PALETTE[k]][(imx_t==k).all(axis=2)] = 1

    return label_seg
```

It is important to remark that the input data format for the masks is $(N \times C \times H \times W)$ where N is the batch size, C the number of classes and $H \times W$ the height and width of the image. The segmentation masks are one-hot encoded, which means that each class has, omitting the batch size, a $(1 \times H \times W)$ image where the pixels belonging to the C class are stored as 1 and the rest as 0. The method *load_mask* in Listing 4.7 performs such encoding by iterating through all the classes stored on the $PALLETE$ dictionary, where the RGB values for each class are stored. For every class and starting with a zero-filled matrix, we write only on the pixel coordinates of the mask that corresponds to the RGB value of the current class. This encoding is depicted in Figure 4.11.



(a) Regular segmentation mask codification.



(b) One-hot encoded masks.

Figure 4.11: One-hot encoding format from a regular segmentation mask. Extracted from Jeremy Jordan semantic segmentation post[2].

4.3.1.3 From UnrealROX to UTP

When loading images from the UnrealROX dataset, segmentation masks are slightly different since we have more than 30 classes. However, our network expected classes are only background and human. To overcome this problem we built two different scripts. The first one is shown in Listing 4.8 and its purpose is to encode the segmentation masks so that it only contains the human and background class.

Listing 4.8: Preprocessing the UnrealROX segmentation masks

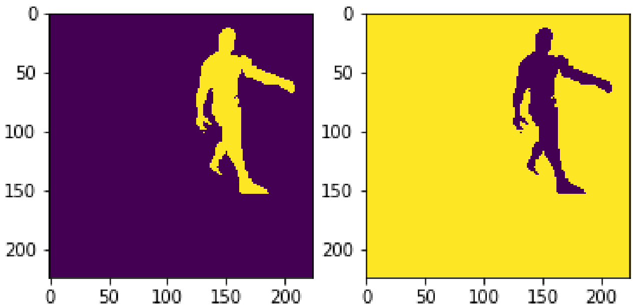
```
imx_t = np.array(raw_image)
imx_t = imx_t[:,:,:3]
label_seg = np.zeros((2,224,224), dtype=np.int)

label_seg[0][(imx_t==[255,0,0]).all(axis=2)] = 1
label_seg[1][np.where(label_seg[0] == 0)] = 1
```

It works in a similar manner to the one-hot encoding on the *load_mask* method (Listing

[2]https://www.jeremyjordan.me/semantic-segmentation/

4.7). In the same fashion, we create a $(2 \times H \times W)$ zero-filled array. Then for the first layer we fill with ones the [255,0,0] RGB value since it corresponds to the human class, as for the second layer, we simply invert the first in order to obtain the background. The result can be seen in Figure 4.12.

(a) Sample UnrealROX segmentation mask.



(b) One-hot two-class segmentation mask.

Figure 4.12: UnrealROX segmentation masks before and after the pre-process pass.

The second script is shown in Listing 4.9. Its main purpose is to adapt the data generated with UnrealROX to the UTP format, in such a way that the UTP dataset class can process the ROX data without extra logic. In order to do this, we iterate through the directory structure of the generated data (Figure 4.13), process each image and move them to our UTP dataset folder. The outermost for loop will iterate through the cameras of the scene. For each camera, we iterate through all the images in both the *rgb* and *mask* folders. The RGB images are simply renamed and moved to the dataset folder, however, the segmentation masks require some pre-processing. As seen in the previous script, we need to get rid of the unnecessary classes. This time, we want the image in the UTP RGB format, which is [255,255,255] for the person class, and [0,0,0] for the background class. Listing 4.10 shows the *process_mask* method which performs such processing.

Listing 4.9: UnrealROX data to UTP format

```
directory = 'scene1/'
dataset_path = 'dataset/dataset/'

new_id = len(os.listdir(dataset_path)) // 2

for camera in ['CornerCamera0/', 'CornerCamera1/', 'MainCamera/']:
    for idx in range(1,len(os.listdir(os.path.join(directory, 'rgb', camera)))):
        if idx % 10 == 0:
            filename = str(idx).zfill(6) + '.png'
            os.rename(directory + 'rgb/' + camera + filename, dataset_path + str(new_id).zfill(5) + '_full.png')

            raw_image = Image.open(directory + 'mask/' + camera + filename)
            raw_image = raw_image.convert('RGB')
            raw_image = process_mask(raw_image)

            raw_image.save(dataset_path + str(new_id).zfill(5) + '_segmentation_full.png')
            new_id += 1
```

Listing 4.10: process_mask method whithin the script

```
pixels = raw_image.load()
for i in range(raw_image.size[0]):
    for j in range(raw_image.size[1]):
        if pixels[i,j] == (255, 0, 0):
            raw_image.putpixel((i,j), (255, 255, 255))
        else:
            raw_image.putpixel((i,j), (0, 0, 0))
```

```
Scene1
├─ rgb
│  ├─ Camera1
│  ├─ Camera2
│  └─ ...
└─ mask
   ├─ Camera1
   ├─ Camera2
   └─ ...
```

Figure 4.13: Directory structure of the generated UnrealROX data.

4.3.2 Training the network

In this Subsection we go through the network implementation and the training script.

4.3.2.1 SegNet Model

In Section 2.3.6 we described the encoder-decoder variant used on CNNs for semantic segmentation, specifically, we described the SegNet architecture. This Subsection describes the model implementation we used[3] for this work.

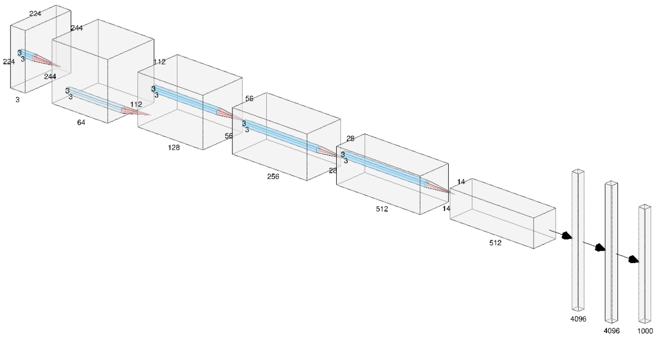


Figure 4.14: Illustration of the VGG-16 architecture.

The encoder and decoder layout equivalent in PyTorch is shown in Listings 4.11 and 4.12. The encoder fundamentally consists of stacked convolutional layers. At the decoder, the convolutions are replaced with *Convolutional Transposed Layers*, which applies a transposed convolution that upsamples the output. Additionally, every convolutional layer is followed by a batch normalization one. These layers standardize their inputs, in a way that for each mini-batch, their mean is 0 and their variance is 1. This can have slight regularization effects and can help to increase the learning process.

Listing 4.11: First layers of the SegNet encoder

```
self.encoder_conv_00 = nn.Sequential(*[nn.Conv2d(in_channels=self.input_channels, out_channels=64, kernel_size=3, padding=1), nn.BatchNorm2d(64) ])

self.encoder_conv_01 = nn.Sequential(*[ nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1), nn.BatchNorm2d(64)])

...

self.encoder_conv_42 = nn.Sequential(*[nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, padding=1), nn.BatchNorm2d(512)])
```

Listing 4.12: First layers of the SegNet decoder

```
self.decoder_convtr_42 = nn.Sequential(*[nn.ConvTranspose2d(in_channels=512, out_channels=512, kernel_size=3, padding=1), nn.BatchNorm2d(512)])

self.decoder_convtr_41 = nn.Sequential(*[nn.ConvTranspose2d(in_channels=512, out_channels=512, kernel_size=3, padding=1), nn.BatchNorm2d(512)])

...

self.decoder_convtr_00 = nn.Sequential(*[nn.ConvTranspose2d(in_channels=64, out_channels=self.
```

[3] https://github.com/Sayan98/pytorch-segnet/blob/master/src/model.py

```
↪ output_channels, kernel_size=3, padding=1)])
```

Then, we need to define the forward function. This module will be perform a forward pass on the network given an input and is shown in Listings 4.13 and 4.14. On the encoder, as with every convolutional network, we apply the activation function between layers and pooling at the end of the convolution. On the decoder, however, the order is reversed, we first apply the reverse pooling function or unpooling, then the activation functions and, finally, in the output layer, we apply a softmax classifier in order to get pixel-wise predictions.

Listing 4.13: Forward function on the encoder

```
dim_0 = input_img.size()
x_00 = F.relu(self.encoder_conv_00(input_img))
x_01 = F.relu(self.encoder_conv_01(x_00))
x_0, indices_0 = F.max_pool2d(x_01, kernel_size=2, stride=2, return_indices=True)

...

dim_4 = x_3.size()
x_40 = F.relu(self.encoder_conv_40(x_3))
x_41 = F.relu(self.encoder_conv_41(x_40))
x_42 = F.relu(self.encoder_conv_42(x_41))
x_4, indices_4 = F.max_pool2d(x_42, kernel_size=2, stride=2, return_indices=True)
```

Listing 4.14: Forward function on the decoder

```
x_4d = F.max_unpool2d(x_4, indices_4, kernel_size=2, stride=2, output_size=dim_4)
x_42d = F.relu(self.decoder_convtr_42(x_4d))
x_41d = F.relu(self.decoder_convtr_41(x_42d))
x_40d = F.relu(self.decoder_convtr_40(x_41d))
dim_4d = x_40d.size()

...

x_0d = F.max_unpool2d(x_10d, indices_0, kernel_size=2, stride=2, output_size=dim_0)
x_01d = F.relu(self.decoder_convtr_01(x_0d))
x_00d = self.decoder_convtr_00(x_01d)
dim_0d = x_00d.size()

x_softmax = F.softmax(x_00d, dim=1)
```

4.3.2.2 Training script

In order to train our model, a training script was built. To do this, we create our data loaders and split the data into train and validation splits. Listing 4.15 shows how the PyTorch DataLoader[4] class can be use to create our data loaders.

When training, we can not only use the training loss as a metric for evaluation, since the network can over-fit the training samples, which means that it will not be good at generalization. Because of that, we use a split of the dataset for validation purposes.

Listing 4.15: Data loaders and train-val split

```
full_dataset = UTPDataset(img_dir='dataset/dataset')

train_size = int(0.8 * len(full_dataset))
val_size = len(full_dataset) - train_size
```

[4] https://pytorch.org/docs/stable/data.html

```

train_dataset, val_dataset = torch.utils.data.random_split(full_dataset, [train_size, val_size])

train_dataloader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=6)
val_dataloader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=6)

data_loaders = {"train": train_dataloader, "val": val_dataloader}
```

In order to create two different data loaders for training and validation, we split the dataset with the *random_split* function. Then, we create the two data loaders and insert them into a dictionary, this allows us to easily differentiate the phase in every epoch.

Listing 4.16: Model criterion and optimizer definition

```
model = SegNet(input_channels=3, output_channels=2).cuda()
criterion = torch.nn.CrossEntropyLoss().cuda()
optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)
```

Listing 4.16 shows the definition of our model and the criterion, which is the metric that is used to determine how far our predictions are from the ground-truth annotation, in our case, we used Cross Entropy Loss. Finally we define our optimizer, which handles how we modify the network weights based on different criteria. In our case, we used the Adam optimizer.

With the initialization ready, we can now train the network. Algorithm 1 shows a high-level version of the training script. For every epoch, we process the whole dataset, which is divided in two different loaders. The *phase* loop will determine which phase is currently being processed. Before iterating the data loaders, we need to set our model to the proper phase by using the *train*() and *eval*() methods. Then the inner most loop will iterate through all the batches of the loader. In this inner loop, we obtain the input RGB images and the target segmentation masks, perform the forward propagation or prediction and then compute the loss with respect to our prediction and the desired one. It is important to remark that, if we are in the training phase, we also need to compute the gradients based on our loss, as well as update the weights using our optimizer. These two steps are not needed in the validation phase since we are just doing it as a metric to know how the network is performing during training.

In order to prevent our model from overfitting, model checkpoints were implemented. This method allows us to keep the best model obtained during training with respect to any measure, in our case, we kept the model with the lowest loss value on the validation set. Listing 4.17 shows this implementation, where *prev_loss* is our best loss in the validation pass and *running_loss* our current validation loss. This is computed after every epoch. The *torch.save* method persists the *mode.state_dict*() which is a dictionary containing the learnable parameters of the model (weights and biases).

Listing 4.17: Model checkpoints

```
if running_loss < prev_loss:
    torch.save(model.state_dict(), os.path.join('.', 'epoch-{}.pth'.format(epoch+1)))
    prev_loss = running_loss
```

Algorithm 1: Training script

Input: model, data_loaders, criterion
Output: model
for $epoch \leftarrow 0$ **to** $EPOCHS$ **do**
  for *phase* ***in*** *['train', 'val']* **do**
    $running\_loss \leftarrow 0$
    if *phase = 'train'* **then**
      model.train()
    else
      model.eval()
    for *batch* ***in*** *data_loaders[phase]* **do**
      $input \leftarrow batch['image']$
      $target \leftarrow batch['mask']$
      $output \leftarrow model(input)$
      $loss \leftarrow criterion(output, target)$
      if *phase = 'train'* **then**
        $loss.backward()$
        $optimizer.step()$
      $running\_loss \leftarrow running\_loss + loss$

4.3.2.3 Loading a trained model

In order to load the model once it has been trained, PyTorch provides the *load_state_dict* method. An example of its use is shown in Listing 4.18.

Listing 4.18: Load model checkpoint

```
model = SegNet(input_channels=3, output_channels=2)
model.load_state_dict(torch.load(PATH))
model.eval()
```

It is important to note that this only works for prediction and not for training, since there are multiple parameters that are not saved on the *state_dict* such as the optimizer or the loss metrics. However, PyTorch does provide us with a custom *save* method that allows to save a custom *dict* and thus to save all the necessary parameters so that it can be used to resume training.

5 Results

This chapter goes through the results of our experimentation with the previously described implementations. Section 5.1 gives an overview of the evaluation methods. Then, Section 5.2 reviews the results when training with and without synthetic data.

5.1 Methodology

In order to evaluate the accuracy of our model, several decisions had to be taken, such as the evaluation metric or the size of our test set.

Regarding the evaluation criteria, multiple methods have been proposed, however, Intersection Over Union (IoU) stands out since it represents the accuracy of the models remarkably well while being very simple.

IoU computes the ratio between the intersection and the union of the semantic prediction and the ground-truth annotation.

$$IoU = \frac{prediction \cap target}{prediction \cup target}$$

Where $prediction \cap target$ can be computed as the number of true positives and $prediction \cup target$ as the sum of the true positives, false negatives and false positives.

$$IoU = \frac{truePositives}{truePositives + falsePositives + falseNegatives}$$

As we previously described in Subsection 4.3.1, our segmentation masks are arrays marked with 1 for each pixel that belongs to the class and 0 for the rest. However, the network output is softmaxed, which means that the values are probabilities instead of discrete predictions. We use a boolean operator on the *Numpy* array in order to check (for each pixel) if the prediction is over a certain threshold. Figure 5.1 shows the result of such discretization.

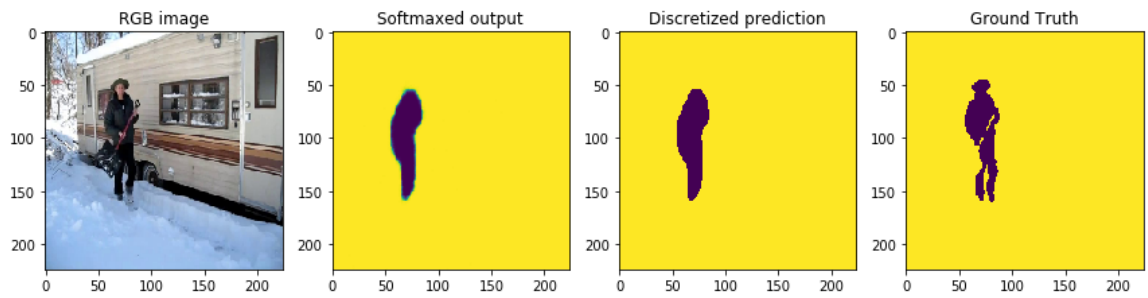


Figure 5.1: Sample output of the softmaxed and discretized predictions of the network.

Once the outputs are discretized, we are able to use the *logical_and*() and *logical_or*() methods of the *Numpy* library in order to get the intersection and union sets respectively. Along with the IoU score, recall and precision metrics were also computed. The precision represents the ability of the model to detect only the relevant pixels, i.e., it computes the ratio of correct pixels among all the predicted ones:

$$Precision = \frac{truePositives}{truePositives + falsePositives}$$

On the other hand, recall represents the ability of the model to find all the relevant pixels within the ground truth mask.

$$Recall = \frac{truePositives}{truePositives + falseNegatives}$$

The implementation of these metrics is shown in Listing 5.1

Listing 5.1: Function that computes the IoU score

```
def compute_iou(prediction, mask):
    intersection = np.logical_and(prediction, mask)
    union = np.logical_or(prediction, mask)

    iou = np.sum(intersection) / np.sum(union)
    precision = (np.sum(intersection) / np.sum(prediction)) if np.sum(prediction) else 0
    recall = np.sum(intersection) / np.sum(mask)

    return iou, precision, recall
```

It is important to remark that these metrics are computed for each class and then averaged, however, since we are only segmenting the human class this is not needed.

Additionally, we set up a testing set which was not used neither for training nor validation. The script shown in Listing 5.2 performs the IoU score for the whole test set and outputs the average score.

Listing 5.2: Testing script

```
test_dataloader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=6)

accumulated_iou = accumulated_precision = accumulated_recall = 0

for batch_idx, batch in enumerate(test_dataloader):
    image, mask = batch['image'].cuda(), batch['mask']

    output = model(image)

    prediction_batch = output[1].cpu()

    for img_idx, img in enumerate(prediction_batch):
        current_prediction = prediction_batch.data.numpy()[img_idx][0]
        current_prediction = 1 - (current_prediction > 0.8).astype(int)
        current_mask = 1 - mask.data.numpy()[img_idx][0]

        score = compute_iou(current_prediction, current_mask)

        accumulated_iou += score[0]
        accumulated_precision += score[1]
        accumulated_recall += score[2]
```

```
total_iou = 100 * (accumulated_iou / test_dataset.__len__())
total_precission = 100 * (accumulated_precision / test_dataset.__len__())
total_recall = 100 * (accumulated_recall / test_dataset.__len__())
```

As for the data split for the different experiments, 6 different models were trained with different number of samples. All of them used the same split ratio, 80% for training and 20% for validation. The test set is made of 1000 samples never for the training process.

5.2 Network convergence and results

The first experimentation was carried out using the whole dataset in order to establish a baseline. Figure 5.2 shows the loss per batch progression through the training process, both for the validation and training sets. It is important to remark that all the models were trained using the Adam optimizer and the Cross Entropy Loss.

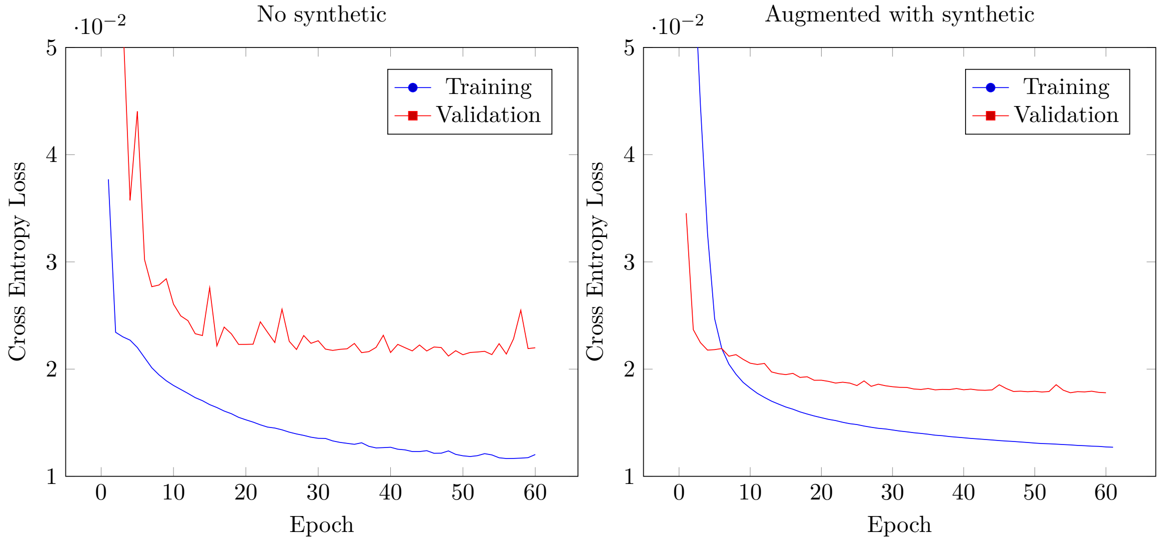


Figure 5.2: Training and validation Cross Entropy Loss without synthetic data (left) and augmented with $\approx$ 10% of synthetic data (right).

When training with synthetic data the validation loss is much more stable. This is due to the real data having more variation than the synthetic one. Although different environments and camera angles were used when obtaining the dataset from UnrealROX, we still lack the variability the real world offers. This downside can be overcome by adding more realistic models and environments. However, due to time restrictions and the environmental modeling process being out of the scope of this project, this was not possible.

Finally, Table 5.1 shows the IoU scores for both the synthetic and non synthetic models, the distribution of the data samples being *real data* + *synthetic data* and the IoU score being displayed as a percentage.

For the next experiment we halved the amount of real samples, the model was trained using 5000 samples from the UTP dataset. In the same fashion, another model was trained after adding 1000 synthetic samples from UnrealROX. As with the previous experiment, Figure 5.3

#	Data samples	IoU Score
1	10000 + 0	32.17%
2	10000 + 1000	30.72%

Table 5.1: IoU score for the model trained with the whole UTP dataset.

shows the loss values during the training phase and Table 5.2 the IoU score for each model.

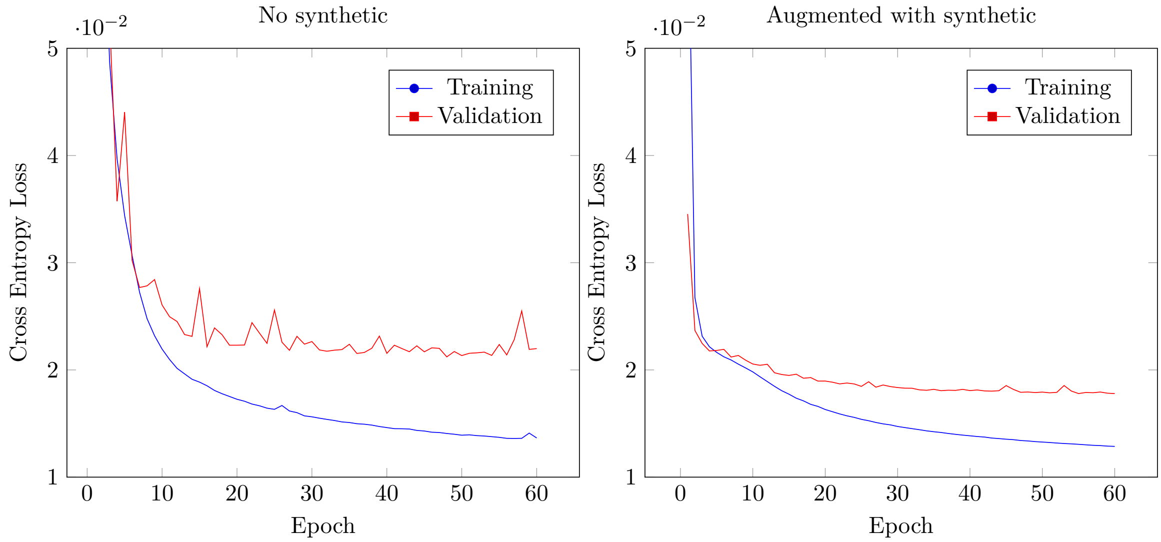


Figure 5.3: Training and validation loss without synthetic data (left) and augmented with $\approx 20\%$ of synthetic data (right).

#	Data samples	IoU Score
1	5000 + 0	9.85%
2	5000 + 1000	9.12%

Table 5.2: IoU score for the model trained with half of the UTP dataset.

The last experiment was performed with a very small set, specifically 2000 samples or 20% of the original UTP dataset. Then it was augmented with 1000 extra synthetic samples. Just as the other experiments, the results are displayed in the same manner in Figure 5.4 and Table 5.3.

#	Data samples	IoU Score
1	2000 + 0	6.34%
2	2000 + 1000	7.25%

Table 5.3: IoU score for the model trained with a small part of the UTP dataset.

Finally, Table 5.4 shows the final results, including precision and recall, for all the 6 models.

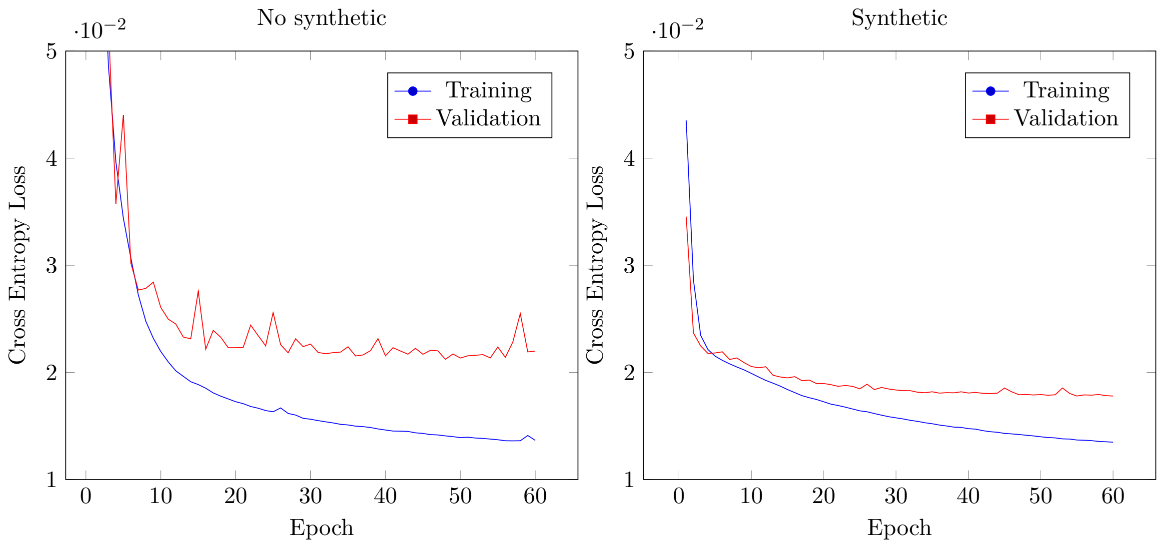


Figure 5.4: Training and validation loss without synthetic data (left) and augmented with $\approx 50\%$ of synthetic data (right).

#	Data samples	IoU Score	Precision	Recall
1	2000 + 0	6.34%	23.30%	8.05%
2	2000 + 1000	7.25%	26.35%	9.20%
3	5000 + 0	9.85%	30.14%	12.49%
4	5000 + 1000	9.12%	28.03%	11.73%
5	10000 + 0	32.17%	52.00%	38.92%
6	10000 + 1000	30.72%	50.53%	38.15%

Table 5.4: IoU, precision and recall for all the trained models.

As it is to be expected, the overall score increases when training with more data samples. Having access to a higher variability dataset allows the network to generalize better. We can also observe that synthetic data seems to decrease the overall performance of the model when the real dataset is big. However, when working with few samples, it can help increase the score significantly.

It is also important to remark how the precision is notably higher than the IoU score. This indicates that the network is able to infer where a person might be although its ability to properly define their boundaries is not that good. This is partly due to the network being relatively shallow. However, this also allows for very fast inferring times, making it a very interesting alternative for real time systems. A small script was built in order to test the average time of inference. Table 5.5 shows such times for different image sizes.

Another interesting point to highlight is how the recall is notably lower than the precision. This is, again, because the network is not good at defining the boundaries, thus the amount of false negatives is high.

Lastly, a small comparison with our results and the ones published by the authors of SegNet was made. They trained their network using the SUN RGB-D benchmark, which is a 37 class,

#	Image Resolution	Avg. Inference time	Avg. FPS
1	256 x 512	0.012 s	83.33
2	512 x 1024	0.042 s	23.81
3	1024 x 2048	0.173 s	5.78

Table 5.5: Inference times of the SegNet for different image resolutions.

indoor scene dataset. Table 5.6 shows a comparison between their accuracy on the human class and ours.

Model	Human Class IoU
SegNet (SUN RGB-D)	27.27%
SegNet (UTP) (ours)	32.17%

Table 5.6: IoU Accuracy comparison with the SegNet results from Badrinarayanan et al. (2015).

In this case, our model was slightly better, showing a 4.9% increase in IoU accuracy. However, it is important to remark that their model was trained to learn 37 different classes. Nonetheless, their mean IoU score for all the classes was 31.84% which is only slightly below our best result.

6 Conclusions

This chapter summarizes the conclusions of this work. Section 6.1 gives an overview of this thesis while analyzing the conclusions. Section 6.2 highlights some of the possible future research lines.

6.1 Summary

In this thesis we made an extensive review of the Sim-To-Real and Semantic Segmentation fields, analyzing some of the most important state-of-the-art architectures used for segmentation, as well as the main methods used in order to generate synthetic data.

Then, we proposed a modification for the UnrealROX framework, in order to ease the sequence generation process. With this framework, we generated a synthetic dataset for testing purposes.

Finally, we implemented a segmentation model and tested its accuracy with different data combinations. After some experimentation, we have concluded that synthetic data can help augment models trained with small datasets, helping its accuracy and overall score. This could be extremely helpful on areas where large-scale datasets are not common. Additionally, we found out that simulators still lack variability, the data is too general and the models struggle to generalize. This can be compensated by adding more environments, models and animations, but this is not the only limitation, as simulators still have to cover the reality gap between them and the real world.

As we previously covered in Section 2.2, the photo-realism field is quickly evolving, while the Sim-To-Real field is still in very early stages. Because of this, we believe the field has a lot of room to grow and it presents as a very interesting alternative to generate data, specially in areas where obtaining real data can be troublesome, expensive or dangerous.

6.2 Future research

As we have stated several times, Sim-To-Real is still a young field and, due to time restrictions, there are multiple research lines that we were not able to fully implement in this thesis. This section summarizes some of the most notable ones:

- **Domain Randomization:** As we described in Section 2.2, Domain Randomization is a very promising technique that could be integrated to the framework. By procedurally generating different meshes and materials with randomized parameters we could obtain multiple models for the *Agent*. This would add more variation to the datasets, which could have a positive impact on the ability of the network to generalize.

- **Adding generated actions from video sequences:** In this thesis, we built a framework in order to integrate actions. However, generating those actions was not possible due to time constraints. By adding new actions we could increase the variability of the dataset, and thus, improving the accuracy of the model when training with synthetic data.

- **Integrating multiple agents:** The current state of the UnrealROX framework only allows to specify the behavior for a single *Agent*. We could add multiple agents and define their (inter)actions. Again, this would allow for more realistic and variable datasets.
- **Pose recognition with synthetic data:** Although UnrealROX, and specifically the *ROXTracker* has knowledge of the *Agent* pose, there is no tool to extract such data. We could have the *ROXTracker* export the position and rotation of each joint in order to train pose estimation networks.

Bibliography

Badrinarayanan, V., Kendall, A., & Cipolla, R. (2015). Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *CoRR*, *abs/1511.00561*. Retrieved from http://arxiv.org/abs/1511.00561

Chen, L., Papandreou, G., Kokkinos, I., Murphy, K., & Yuille, A. L. (2016). Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *CoRR*, *abs/1606.00915*. Retrieved from http://arxiv.org/abs/1606.00915

Chen, Y., Li, W., Chen, X., & Gool, L. V. (2018). Learning semantic segmentation from synthetic data: A geometrically guided input-output adaptation approach. *CoRR*, *abs/1812.05040*. Retrieved from http://arxiv.org/abs/1812.05040

Cordts, M., Omran, M., Ramos, S., Rehfeld, T., Enzweiler, M., Benenson, R., … Schiele, B. (2016). The cityscapes dataset for semantic urban scene understanding. *CoRR*, *abs/1604.01685*. Retrieved from http://arxiv.org/abs/1604.01685

Everingham, M., Eslami, S. M. A., Van Gool, L., Williams, C. K. I., Winn, J., & Zisserman, A. (2015, January). The pascal visual object classes challenge: A retrospective. *International Journal of Computer Vision*, *111*(1), 98–136.

Gaidon, A., Wang, Q., Cabon, Y., & Vig, E. (2016). Virtual worlds as proxy for multi-object tracking analysis. *CoRR*, *abs/1605.06457*. Retrieved from http://arxiv.org/abs/1605.06457

Garcia-Garcia, A., Martinez-Gonzalez, P., Oprea, S., Castro-Vargas, J. A., Orts-Escolano, S., Rodríguez, J. G., & Jover-Alvarez, A. (2019). The robotrix: An extremely photorealistic and very-large-scale indoor dataset of sequences with robot trajectories and interactions. *CoRR*, *abs/1901.06514*. Retrieved from http://arxiv.org/abs/1901.06514

Geiger, A., Lenz, P., Stiller, C., & Urtasun, R. (2013). Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*.

Graves, A., Fernández, S., & Schmidhuber, J. (2007). Multi-dimensional recurrent neural networks. *CoRR*, *abs/0705.2011*. Retrieved from http://arxiv.org/abs/0705.2011

Hariharan, B., Arbelaez, P., Bourdev, L., Maji, S., & Malik, J. (2011). Semantic contours from inverse detectors. In *International conference on computer vision (iccv)*.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep residual learning for image recognition. *CoRR*, *abs/1512.03385*. Retrieved from http://arxiv.org/abs/1512.03385

Krizhevsky, A., Sutskever, I., & E. Hinton, G. (2012, 01). Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, *25*. doi: 10.1145/3065386

Lassner, C., Romero, J., Kiefel, M., Bogo, F., Black, M. J., & Gehler, P. V. (2017, July). Unite the people: Closing the loop between 3d and 2d human representations. In *Ieee conf. on computer vision and pattern recognition (cvpr).* Retrieved from `http://up.is.tuebingen.mpg.de`

Lin, T., Maire, M., Belongie, S. J., Bourdev, L. D., Girshick, R. B., Hays, J., … Zitnick, C. L. (2014). Microsoft COCO: common objects in context. *CoRR*, *abs/1405.0312*. Retrieved from `http://arxiv.org/abs/1405.0312`

Long, J., Shelhamer, E., & Darrell, T. (2014). Fully convolutional networks for semantic segmentation. *CoRR*, *abs/1411.4038*. Retrieved from `http://arxiv.org/abs/1411.4038`

Martinez-Gonzalez, P., Oprea, S., Garcia-Garcia, A., Jover-Alvarez, A., Orts-Escolano, S., & Rodríguez, J. G. (2018). Unrealrox: An extremely photorealistic virtual reality environment for robotics simulations and synthetic data generation. *CoRR*, *abs/1810.06936*. Retrieved from `http://arxiv.org/abs/1810.06936`

Puig, X., Ra, K., Boben, M., Li, J., Wang, T., Fidler, S., & Torralba, A. (2018). Virtualhome: Simulating household activities via programs. In *Computer vision and pattern recognition (cvpr).*

Ronneberger, O., Fischer, P., & Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. *CoRR*, *abs/1505.04597*. Retrieved from `http://arxiv.org/abs/1505.04597`

Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *3rd international conference on learning representations, ICLR 2015, san diego, ca, usa, may 7-9, 2015, conference track proceedings.* Retrieved from `http://arxiv.org/abs/1409.1556`

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., … Rabinovich, A. (2014). Going deeper with convolutions. *CoRR*, *abs/1409.4842*. Retrieved from `http://arxiv.org/abs/1409.4842`

Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., & Abbeel, P. (2017). Domain randomization for transferring deep neural networks from simulation to the real world. *CoRR*, *abs/1703.06907*. Retrieved from `http://arxiv.org/abs/1703.06907`

Visin, F., Kastner, K., Cho, K., Matteucci, M., Courville, A. C., & Bengio, Y. (2015). Renet: A recurrent neural network based alternative to convolutional networks. *CoRR*, *abs/1505.00393*. Retrieved from `http://arxiv.org/abs/1505.00393`

Walsh, J., O' Mahony, N., Campbell, S., Carvalho, A., Krpalkova, L., Velasco-Hernandez, G., … Riordan, D. (2019, 04). Deep learning vs. traditional computer vision..

Yu, F., & Koltun, V. (2015). *Multi-scale context aggregation by dilated convolutions.*

Zeiler, M. D., Taylor, G. W., & Fergus, R. (2011, Nov). Adaptive deconvolutional networks for mid and high level feature learning. In *2011 international conference on computer vision* (p. 2018-2025). doi: 10.1109/ICCV.2011.6126474