Escuela
Politécnica
Superior

Semantic Segmentation and Sim2Real

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Jordi Amorós Moreno
Tutor/es:
José García-Rodriguez
Nombre Apellido1 Apellido2 (tutor2)

Julio 2019

Universitat d'Alacant
Universidad de Alicante

Semantic Segmentation and Sim2Real

Subtítulo del proyecto

Autor
Jordi Amorós Moreno

Tutor/es
José García-Rodriguez
Departamento de Tecnología Informática y Computación
Nombre Apellido1 Apellido2 (tutor2)
Departamento de Tecnología Informática y Computación

Grado en Ingeniería Informática

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Julio 2019

Abstract

“TODO”

Aknowledgements

TODO

TODO

TODO

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 will give an overview of the whole thesis. Section 1.2 will describe the motivations of this research. Finally, in Section 1.3 we will point out the main proposal as well as the goals for this work.

1.1 Overview

In this bachelor's thesis we will be researching the Sim-2-Real field, specifically tackling the object detection problem from a semantic segmentation perspective. In the following chapters we will review and analyze the current state of art of some of the most important datasets, architectures and techniques used for semantic segmentation up to this day. We will also dive into the Sim-2-Real field, analyze some of the latest works as well as try to demonstrate how synthetic data can help data-driven algorithms solve problems in real-world domains.

This document is structured as follows: This first Chapter 1 and Chapter 2 will go over the related works and State of Art of the Sim-2-Real and Semantic Segmentation field, as well as some works that served as inspiration while developing this project. Chapter 3 will describe the studied materials and methodologies used in this work. In Chapter 4 we will describe the process of expansion of the UnrealROX framework as well as the Semantic Segmentation implementations. Finally, in Chapter 6 we go over the conclusions of this research.

1.2 Motivation

The main motivation behind this project is to further investigate the Sim-2-Real and object detection field, specifically focusing on the UnrealROX project, and semantic segmentation techniques, concentrating our efforts in developing a user friendly framework for researches to easily generate synthetic data sequences in order to apply deep learning algorithms to real world environments.

This document was motivated by a collaboration with the Department of Information Technologies and Computing (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.

UnrealROX is a project directly correlated to *The Robotrix: An eXtremely Photorealistic and Very-Large-Scale Indoor Dataset of Sequences with Robot Trajectories and Interactions* Garcia-Garcia et al. (2019) which was presented at the IROS conference in 2018. Funded by *Ministerio de Economia y Competitividad* of the Spanish Government and directed by Jose Garcia-Rodriguez and Miguel Angel Cazorla-Quevedo, UnrealROX was the framework developed in order to generate said dataset.

In the last decade, data driven algorithms have improved tremendously and huge high quality datasets have been created in order to improve the accuracy of such algorithms.

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

However it is still extremely expensive, both in time and resources, to create such datasets. This is where the Sim-2-Real field and the synthetic data generation comes into play, in this work we will specifically focusing on *UnrealROX: An eXtremely Photorealistic Virtual Reality Environment for Robotics Simulations and Synthetic Data Generation* by Martinez-Gonzalez et al. (2018).

1.3 Proposal and Goals

The main proposal for this work is to develop an extension to the UnrealROX project 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 was to establish a new type of *Agent* in the framework that was not user-controlled, this would allow to generate data sequences without the need of a Virtual Reality (VR) Headset and user input, this provides a faster and more convenient way to obtain datasets.

The second main objective of this work was to test if such data would prove useful in real world problems. In order to demonstrate this, we will have to develop data driven algorithms and verify their accuracy with real-world datasets.

2 State of Art

As we previously stated, semantic segmentation is a extremely important task in the field of computer vision due to its enormous value towards complete scene understanding. This chapter is organized as follows: Section 2.1 will give a brief introduction to the Semantic segmentation problem. In section 2.2 we will 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, in fact it is fundamentally the same, it just adds an extra layer of complexity towards a more fine-grained solution. 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.

2.2 Sim 2 Real

In the last decade, data driven algorithms have vastly surpassed traditional techniques for computer vision problems, these algorithms, although can be tuned and improved in many different ways, still require vast amounts of 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 will allow us to workaround the previously mentioned restrictions, there's 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 will present discrepancies between them and the real world, to overcome this and properly

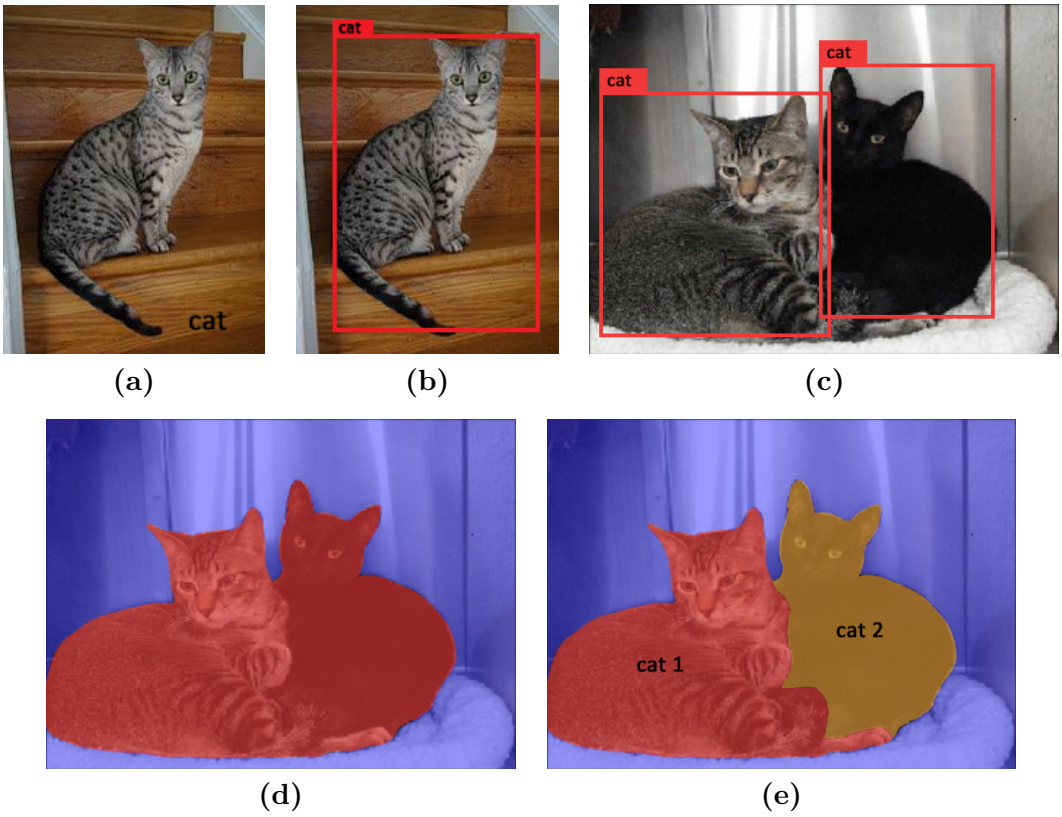


Figure 2.1: **(a)** Object detection **(b)** Object localization **(c)** Multiple object localization **(d)** Semantic segmentation **(e)** Instance segmentation.

transfer the knowledge to real problems there are two 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 innovation 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.

  Normally, Ray-Tracing techniques are performed offline since they are quite heavy in

[1] `https://devblogs.nvidia.com/vulkan-raytracing`

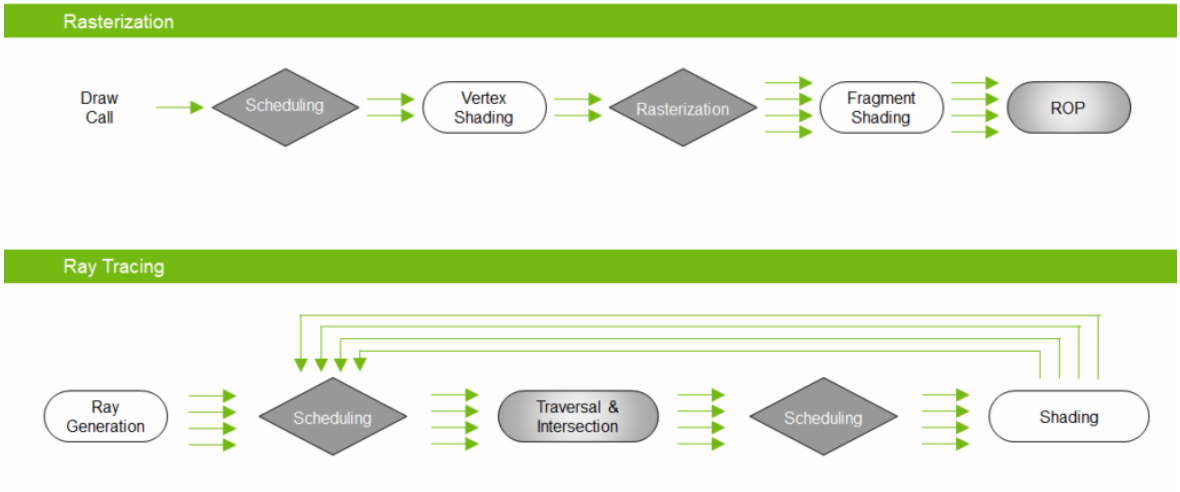


Figure 2.2: Traditional rasterization pipeline in contrast to the Ray Tracing pipeline. Extracted from NVIDIA devblog[1].

terms of computation times. 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 and worth keeping an eye on it.

- **Domain randomization:** This is the main alternative to photorealism when trying to cover the reality gap between the real world and the synthetic environments. This technique is based on showing 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 will appear as just another variation which will allow the model to generalize. Figure 2.3 shows an example of randomized training inputs.

  Another interesting approach to reduce the reality gap was researched in a 2018 paper by Y. Chen et al. (2018). In this work they present **Domain Adaptation**, the main takeaway behind this idea is to transfer the real-world style of images into the synthetic ones. This is done by a Generative Adversarial Network (GAN), the image transform network generates an image taking the synthetic RGB image, the segmentation masks of said image, as well as the depth map. The discriminator will have to differentiate between the real image and the one generated by the network.

  Also a second adaptation pass is done at the output level. The Task Network is the responsible for predicting the outputs of both real and synthetic images. Then it goes through another discriminator which has to discern if the outputs are predicted from a real or the synthetic adapted image. Figure 2.4 shows their proposed architecture.

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

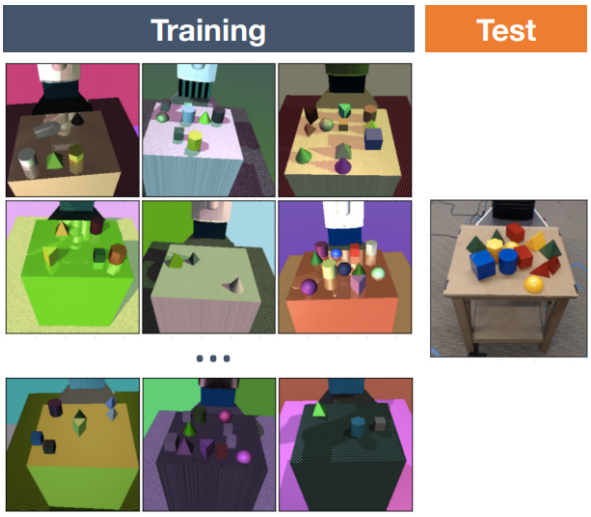


Figure 2.3: Low-fidelity images with random variations in camera angle, lightning and positions are used to train an object detector. Testing is done in the real world. Image from Tobin et al. (2017).

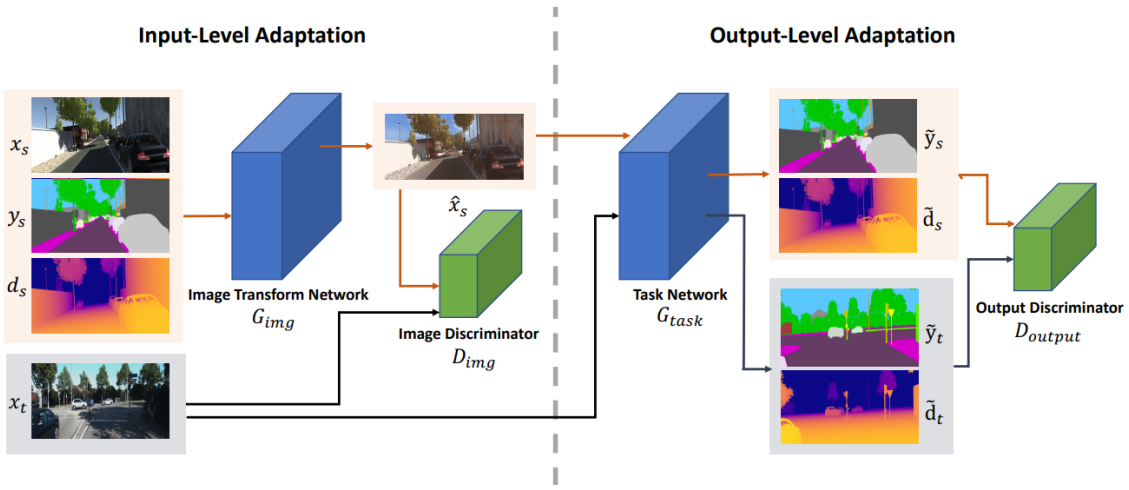


Figure 2.4: Overview of the Input-Output adaptation network form Y. Chen et al. (2018)

2.2.1 VirtualHome

VirtualHome 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 was necessary, 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 had workers describe in natural language all 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.

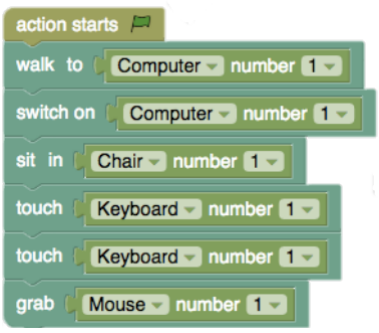


Figure 2.5: List of actions represented with the scratch interface, where the user can manually add, modify and change the arguments of every action.

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

2.2.2 UnrealROX

UnrealROX 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 higher level, we will also require more complex architectures. Although the models we will be reviewing in this section work properly for image recognition and detection, some modifications will 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 being used as the basic building blocks of segmentation architectures.

2.3.1 AlexNet

AlexNet 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, easily surpassing the its competitors by a huge margin. The architecture itself was pretty straight forward. It consisted of five convolution + pooling layers followed by three fully connected ones as seen in figure 2.6.

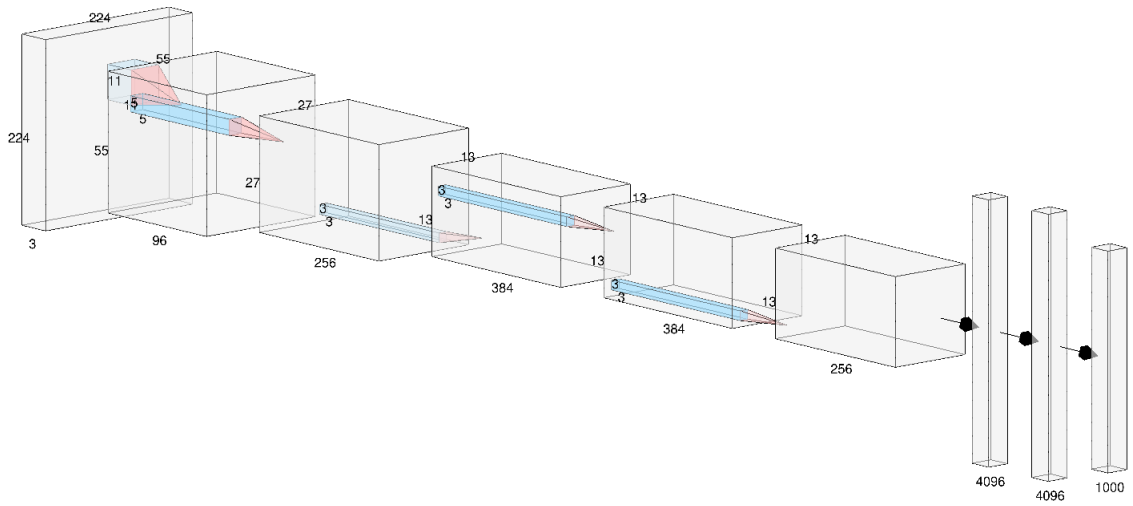


Figure 2.6: AlexNet architecture reproduced from Krizhevsky et al. (2012)

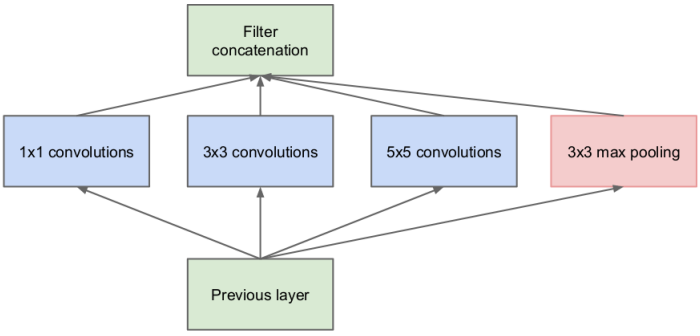


Figure 2.7: Inception module extracted from Szegedy et al. (2014)

2.3.2 VGG

VGG (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. The VGG-16 achieved 92.7% TOP-5 test accuracy.

The structure of VGG-16 is also quite simple and not too deep, as its own name hints, it consists of 16 convolutional layers, and just like AlexNet, uses three fully connected layers for classification. The main improvement over AlexNet was made 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 (cite) and was submitted to ILSVRC 2014, winning with a TOP-5 test accuracy of 93.3%. GoogLeNet architecture is rather complex, it introduced the inception module (shown in Figure 2.7) which consisted of a new approach where convolution layers were not stacked in just sequential order but instead had some of them compute 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 He et al. (2015) was first introduced by Microsoft research and it got very popular after achieving 96.4% in ILSVRC 2016. This CNN is extremely deep with 152 layers (although shallower variations do exist) and it introduced a new concept called residual blocks. Since AlexNet was released, CNN have become increasingly deeper, this 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 compute output is a combination of X and $F(X)$ as depicted in Figure 2.8. This helps reducing the vanishing

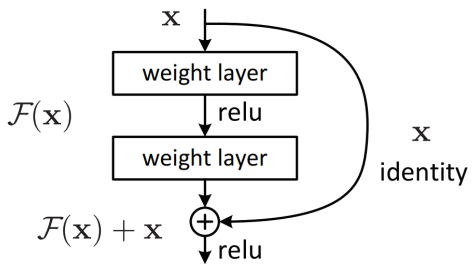


Figure 2.8: Residual block extracted from He et al. (2015)

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

Multi Dimensional Recurrent Neural Network (MDRNN) are a variation of regular RNN that allow them to work with d spatio-temporal dimensions of the data. The architecture proposed by (cite) used regular RNN instead of MDRNN, where every convolution + pooling layer is replaced by four RNN's that sweep the image across in four directions.

2.3.6 Semantic Segmentation Methods

All the image recognition problem solutions based on convolutional architectures, whether it is recognition, detection, localization or segmentation, they all share a big common module, which is the convolution layers that will 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 uses 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) downsample the input image in order to learn features, this downsampling 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 upsampled 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 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 upsampling layers that correspond to the max-pooling ones in the encoder, the whole architecture can be seen in figure 2.9. SegNet is a very simple architecture which yields very good results and is relatively fast, which makes it a good starting point of any semantic segmentation problem.

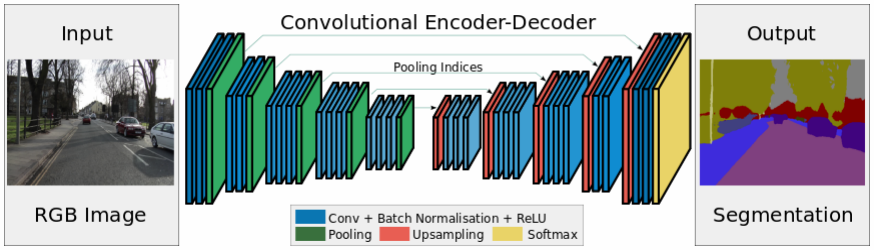


Figure 2.9: Segnet architecture graph extracted from Badrinarayanan et al. (2015)

2.3.6.2 Dilated Convolutions

As we previously mentioned, CNN's 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 will control the up-sampling factor of the filters. That way, a 1-dilated convolution would be 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 multi-context aggregation by Yu & Koltun (2015) and DeepLab by L. Chen et al. (2016).

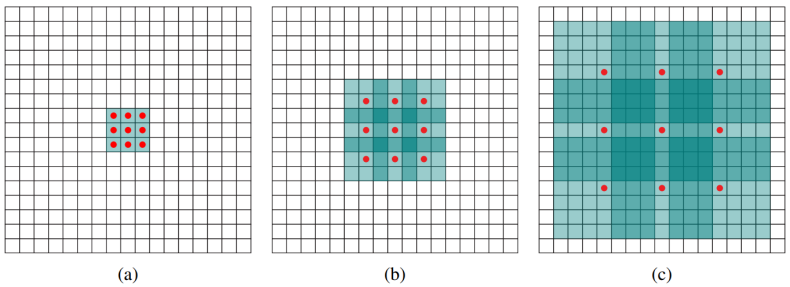


Figure 2.10: (a) 1x1 receptive fields, 1-dilated, (b) 3x3 receptive fields, 2-dilated, (c) 7x7 receptive fields, 3-dilated. Figure extracted from Yu & Koltun (2015).

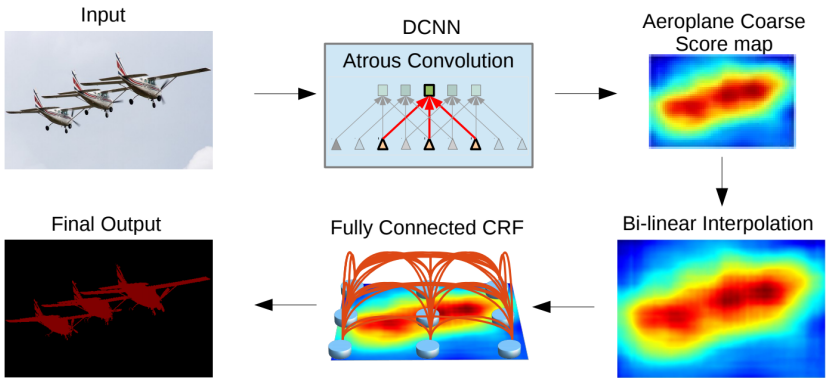


Figure 2.11: Illustration of the DeepLab proposed architecture, using a deep CNN for pixel-wise classification and a fully connected CRF to refine the output.

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, CRF. CRFs make use of both low level pixel interaction as well as the multi-class inference pixel prediction of high level models.

The DeepLab model 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 is one of the most popular 2D datasets 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 such as PASCAL Context which provides pixel-level classification for the entire original dataset classes although only 59 of them are usually taken into account when working with classification problems since the rest of them are too rare, so they are normally considered as background. 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 could be decomposed into wheels, chassis, headlights and windows, figure 2.12 shows more examples of different classes.

2.4.2 Semantic Boundaries Dataset

This dataset 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.

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

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 is a urban dataset mainly used for instance and semantic segmentation. It contains over 25000 images and 30 different classes was recorded in 50 cities during different times of the day and year.

2.4.4 KITTI

The KITTI dataset Geiger et al. (2013) was recorded from a vehicle on a urban environment. It includes camera 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.

2.4.5 COCO

COCO is yet 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

This chapter is organized as follows: Section 3.1 will analyze 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 will review our hardware equipment used for this project.

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 were 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 UE4 potential to run extremely realistic scenes, such as the one shown in figure 3.1, in real time made it perfect for this purpose.
- **Blueprints:** Blueprints are one of the tools that UE4 offers and it allows for quick behavior definitions in the editor without writing a single line of code. This makes it perfect for prototyping and testing.
- **Community:** UE4 is currently one of the most popular game engines and it has a rather big community, the official forums and other platforms are very active and the documentation is well maintained. The developing team is very active 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 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 comes integrated with IntelliSense, which allows for code completion, quick information of variables and methods, amongst others 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.

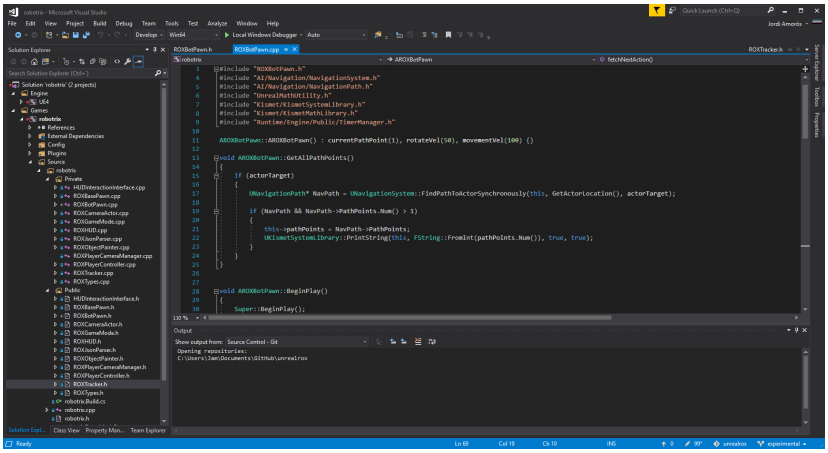


Figure 3.2: Visual Studio IDE.

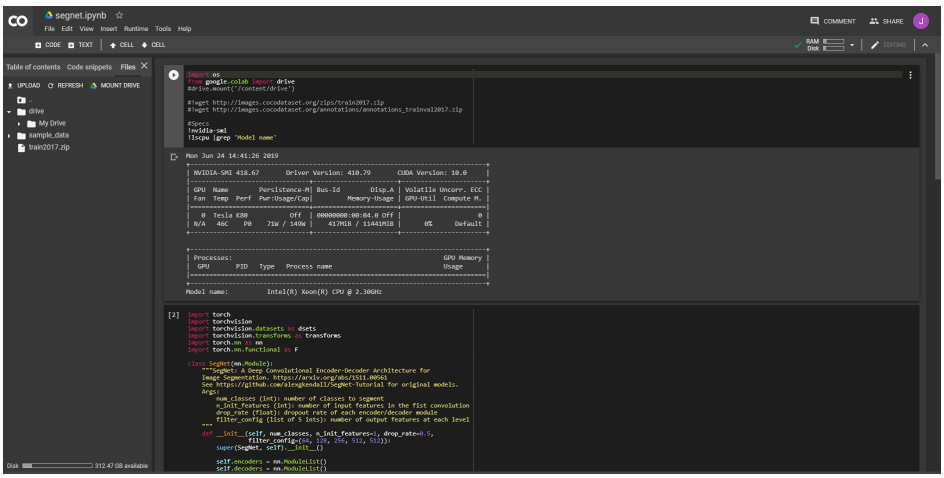


Figure 3.3: Google Colab web interface.

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

TODO

3.1.5 Frameworks

Deep Learning (DL) has been arising in popularity in the last decade and most current state-of-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 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 3.4 illustrates an example of a flow graph representation.

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

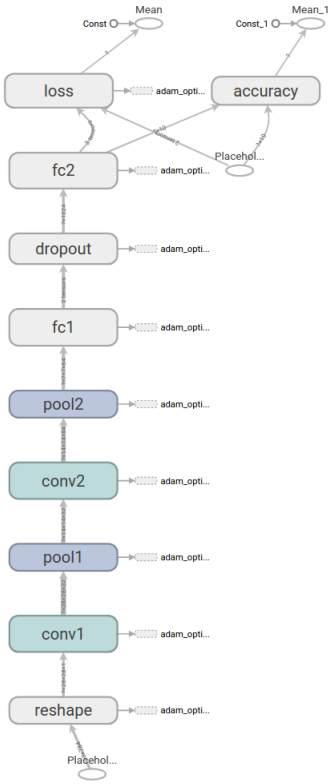


Figure 3.4: Tensorflow graph example depicting a simple two-layer convolution + pooling with a fully connected layer for classification at the end.

TensorFlow was created by the researches 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 distribution on the CPU-GPU.

In a first approach, TensorFlow was going to be the main framework for this project, but was finally discarded since high level frameworks will ease the work and the low level implementation of the networks falls out of the scope of this project.

3.1.5.2 Keras

Keras is a high level framework written in Java 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 of some of the more complex low level tasks with a very user friendly interface. This also makes Keras a very good entry framework for beginners that still don't have a solid foundation on deep learning.

Keras provides two different APIs for different model building approaches. The Sequential API which allows to simply stack layer after layer, allowing for a very simple and easy to use interface for models with a input to output data flow. The Functional API however allows for more complex models by understanding each layer as a node graph, allowing for different, more complex and non sequential models.

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. We discarded it for the end project since we believe there are better alternatives that allow for more flexibility.

3.1.5.3 PyTorch

PyTorch is an open source, Python-based, GPU-Ready computing package and machine learning framework, just like other frameworks, it provides Tensors, which can be moved both to the Central Processing Unit (CPU) and GPU, as well as all the algebra and math operations for them, they can also be converted from NumPy and other Tensor libraries.

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 Application Program Interface (API). Also it has 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 easily outperforms other high level frameworks such as Keras.

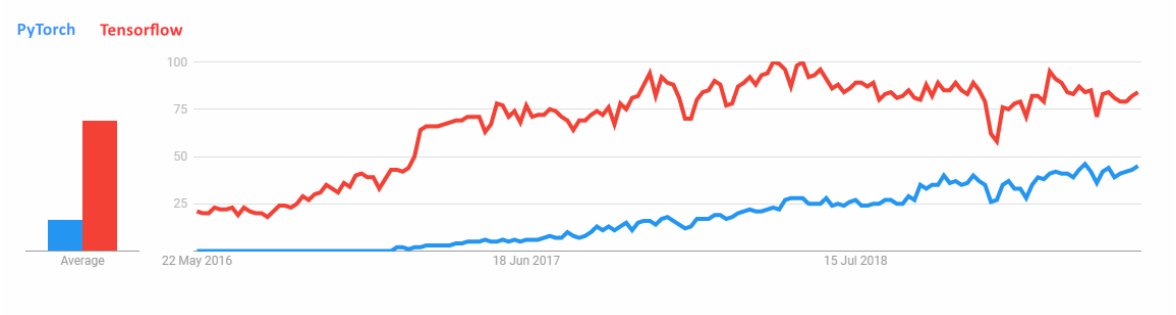


Figure 3.5: Worldwide PyTorch and Tensorflow popularity comparison in Google Search.

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 mainly used in this work.

3.2.1 Clarke

The structure of neural networks where multiple data streams are organized in layers allow for very easy paralelization. Because of this GPU's 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 will only be used for visualization purposes. The full hardware specification for the Clarke server is shown in Figure 3.1.

As for the software, Asimov runs Ubuntu 16.04 with Linux kernel 4.4.0-21-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.

Also, it is worth mentioning Asimov 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.

Asimov	
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, my 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 the 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 my 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 Desarrollo

This chapter develops the main work of this thesis and is organized as follows. Section 4.1 will describe the process of automating the UnrealROX Actor. Section 4.2 will go through the process of recording sequences. Finally, Section 4.3 will cover all 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 section 1 one of the main goals on 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 will 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. In this work we have built the basic framework for the programmer to include its own actions and execute them in a sequential way, much like other frameworks such as VirtualHome 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 introduce a robot to the scene and manually move and interact with the objects in a scene. We will use this class as our parent for the implementation.

4.1.2 The ROXBotPawn Class

The ROXBotPawn class inherits from ROXBasePawn and will handle all the logic for the automation of tasks of any *Pawn* within a scene. In order to model all the different actions and interactions the Enum $EActionType$ was created, here 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 $EActionType$, the structure is shown in listing 4.1.

Listing 4.1: FROXAction struct

```
1  USTRUCT(BlueprintType)
2  struct FROXAction
3  {
4      GENERATED_USTRUCT_BODY()
5      UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = Pathfinding)
6      AActor* target;
```

```
7
8       UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = Pathfinding)
9       EActionType action;
10
11      FROXAction() : target(nullptr), action(EActionType::MoveTo) {};
12      FROXAction(AActor* tg, EActionType t) : target(tg), action(t) {};
13  }
```

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 way, 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 making 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

```
1   UFUNCTION(BlueprintCallable)
2   void AROXBotPawn::doAction(AActor* actor, EActionType type)
3   {
4       actions.Add(FROXAction(actor, type));
5   }
```

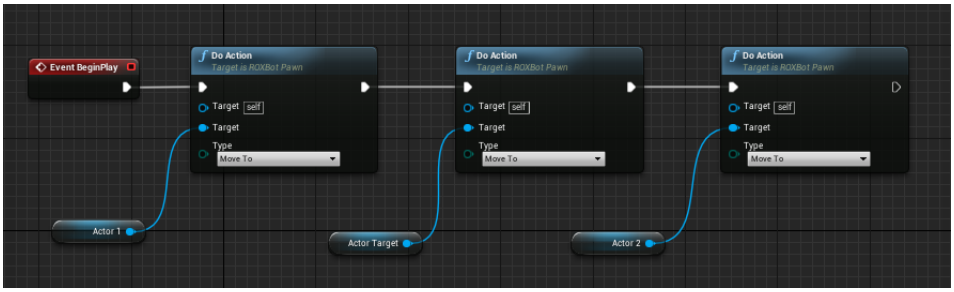


Figure 4.1: Example of queuing 3 "MoveTo" actions from the editor

The $doAction()$ method creates a $FROXActions$ 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 and allows to gather data from the environment. It 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

```
1 FVector nextPos = FMath::VInterpConstantTo(this->ActorLocation(), FVector(pathPoints[i].X, ↩
      ↪ pathPoints[i].Y, ActorLocation().Z), DeltaTime, vel);
2 FRotator nextRot = UKismetMathLibrary::RInterpTo_Constant(ActorRotation(), UKismetMathLibrary::↩
      ↪ FindLookAtRotation(ActorLocation(), nextPos), DeltaTime, rotateVel);
3 SetActorRotation(nextRot);
4 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 the algorithms to properly infer the knowledge and transfer it to the real world. In this section we will 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 the assets, in our case we will be using the default walk and idle animation from the UnrealROX framework. In order to have smooth transitions between different animations i.e 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 Figures 4.3 and 4.4.

With the event graph and the BlendSpaces created all that is left to do is create the state machine which will contain 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 is shown in Figure 4.5.

The logic for the transitions between states are very simple, we will simply check if 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.

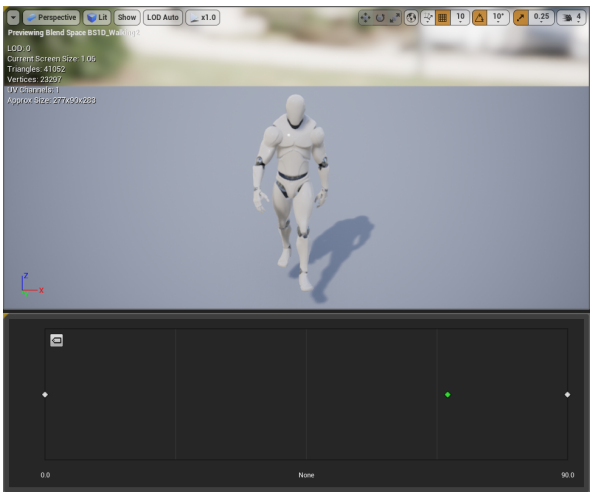


Figure 4.2: Blend Space asset which samples the transition from idle walking state animation, 0 speed would translate into a complete idle, while 90 would be walking forward.

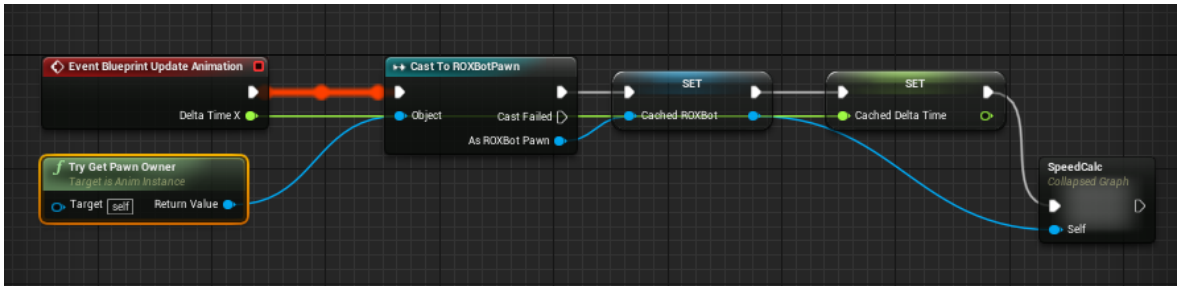


Figure 4.3: Animation event graph which will obtain the Bot data, the SpeedCalc module is expanded in Figure 4.4.

4.2 Recording sequences with UnrealROX

In the previous section, we developed the process of automatization of the framework in order to record sequences without the need of manually doing the actions with a VR Headset. In this section we will 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 that has knowledge of the whole scene. In order to use it, we just need to search for it in the contextual menu (as shown in Figure 4.6) and drag it to our scene. While in record mode, the ROXTracker will be able to store all of the information needed in order to rebuild the sequence offline as a TXT or JSON file, this will allow then to run the sequence in playback mode and generate frame by frame, ground truth annotated images such as the segmentation masks, depth and normal maps, as well as the RGB images.

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

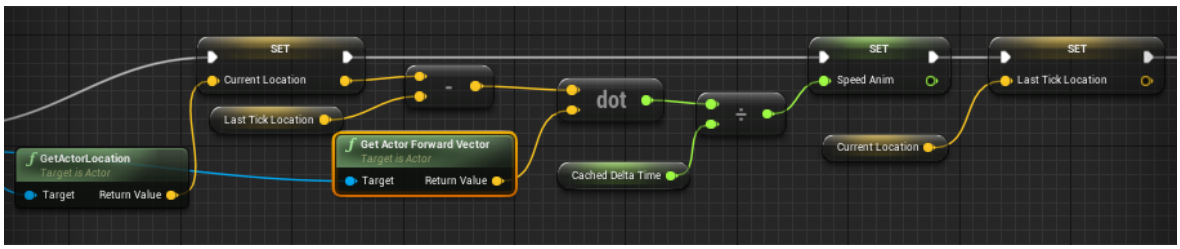


Figure 4.4: Blueprint sub-module which computes the instantaneous speed of the Bot.

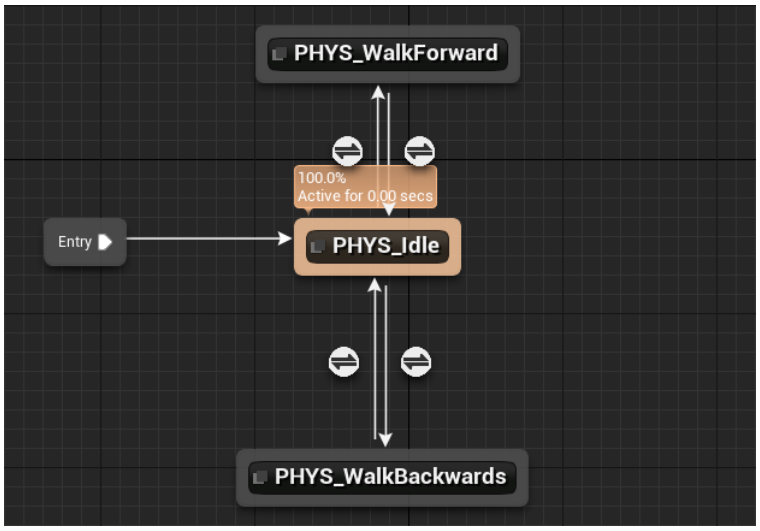


Figure 4.5: Animation state machine with idle, walk forward and walk backwards states and the transitions logic.

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 Secene Folder with name corresponding to the field **Input Scene TXT File Name** and will generate its equivalent JSON file with the name on the field **Output Scene Json File Name** i.e looking at Figure 4.7, the Tracker will search for a *scene.txt* file inside the *unrealrox/RecordedSequences* folder and generate a *scene.json* file.

4.2.1 Recording mode

Before start generating sequences we have to make some tweaks to the recording settings of the ROXTracker, which can be seen in Figure 4.8 and further explained as follows:

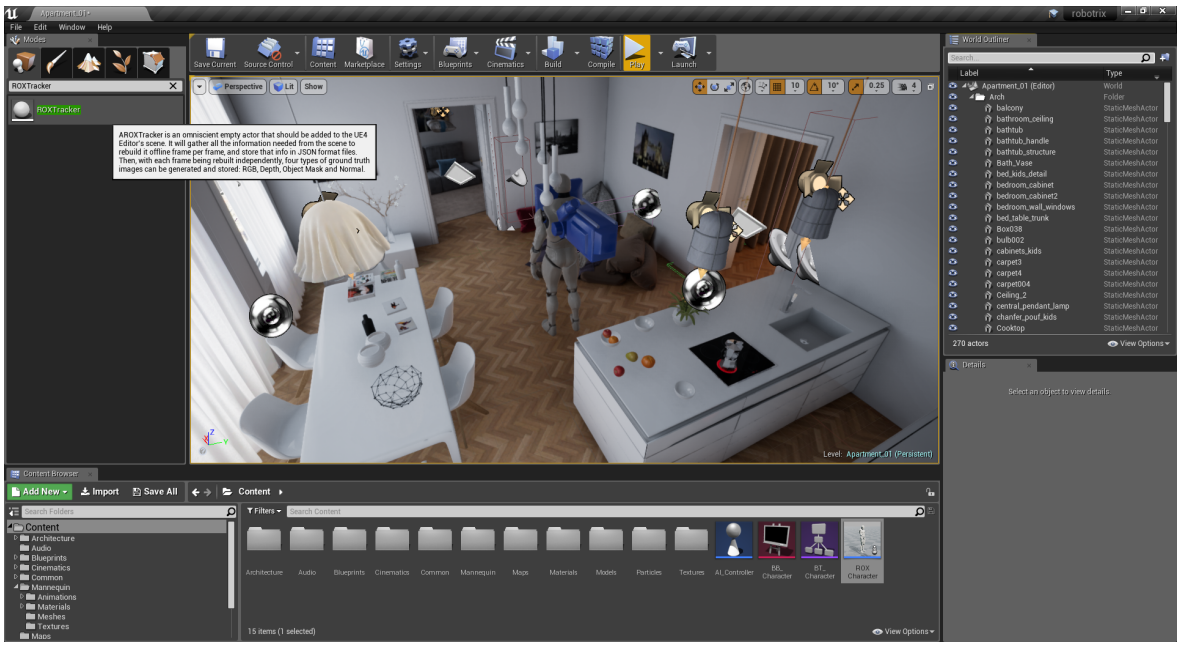


Figure 4.6: ROXTracker Object in the UE4 contextual menu.

- **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.

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/stop recording the user needs press "R", a red **RECORDING** message will display at the top of the screen, an example of the recorder running can be seen inf Figure 4.9.

When the sequence finishes, we can stop recording. If we take a look at our designated Scene Folder for the data generation we can se our recording raw data in TXT format. We now need, in order to get it ready for playback, parse it to JSON, we can do this with the **Generate Sequence Json** utility we have seen in Figure 4.7.

4.2.2 Playback mode

With our recording data in JSON format we are now able to play the sequences in the UE4 editor, but first we will go through some of the configuration settings for the playback mode, they can be seen in Figure 4.10 and are further explained below:

Figure 4.7: ROXTracker settings in the UE4 editor.

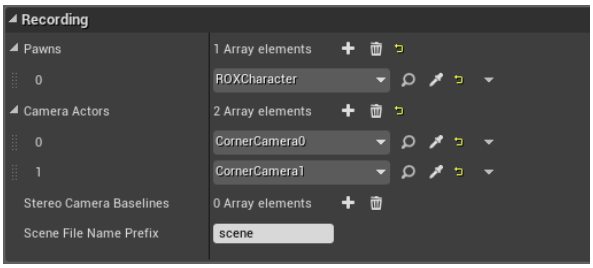


Figure 4.8: ROXTracker recording settings.

- **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.
- **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 screenshot folder.
- **Width/Height:** Output resolution for the generated images.

Figure 4.9: Example of a running scene being recorded.

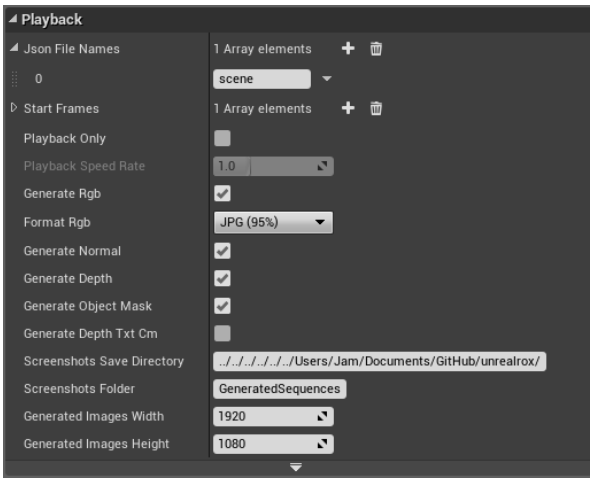


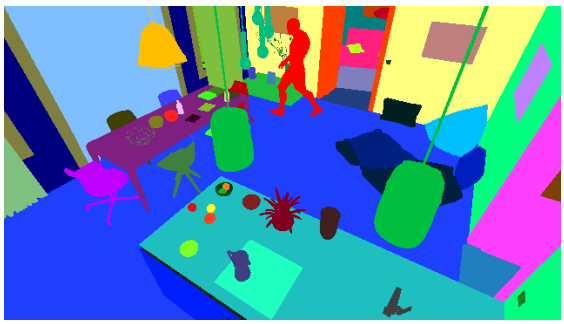
Figure 4.10: ROXTracker playback settings.

Once the desired configuration is set and the Record Mode is disabled , we can hit 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 4.11.

4.3 Implementing a SegNet using PyTorch

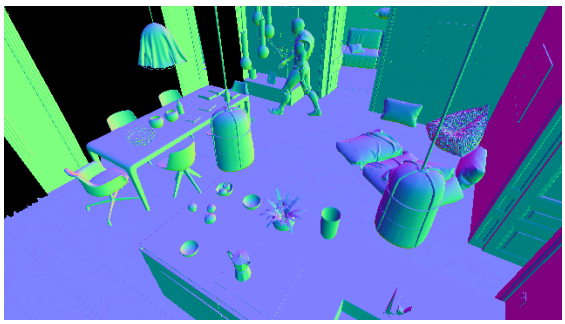
As we previously mentioned, one of the main goals of this work is to study how synthetic data can help semantic segmentation algorithms. For this purpose, a SegNet has been implemented and trained with a real-world human-pose dataset, this section will cover all the data processing, as well as the design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

(a) Output RGB image.



(b) Output Segmentation Masks.

(c) Output Depth image.



(d) Output Normals image.

Figure 4.11: 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 Unite The People (UTP) dataset. 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.

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

```
1    csv = pd.read_csv(csv_path)
```

```
2
3   while x < len(csv):
4       instances = 0
5
6       for id in range(x, len(csv)):
7           if csv['mpii_id'][id] == csv['mpii_id'][id+1]:
8               instances += 1
9           else:
10              break
```

With the amount of instances and having 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, this method takes two inputs, the image to be pasted and the mask which contains the pixels that are to be copied into the our 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

```
1   first_image = Image.open(img_path + str(x))
2
3   for n in range(1, instances + 1):
4       next_image = Image.open(img_path + str(x + n))
5       first_image.paste(next_image, (0,0), next_image.convert('L'))
6
7   new_id += 1
8   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, this way we will be able to create our data loaders and 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

```
1   class UTPDataset(Dataset):
2       def ___init___(self, img_dir, transform=None):
3           self.transform = transform
4           self.image_root_dir = img_dir
5           self.img_extension = '_full.png'
6           self.mask_extension = '_segmentation_full.png'
7
8       def ___getitem___(self, index):
9           image_id = str(index).zfill(5)
10          image_path = os.path.join(self.image_root_dir, image_id, self.img_extension)
11          mask_path = os.path.join(self.image_root_dir, image_id, self.mask_extension)
12
13          image = self.load_image(path=image_path)
14          mask = self.load_mask(path=mask_path)
15
16          data = {
17                  'image': torch.FloatTensor(image),
18                  'mask' : torch.LongTensor(mask)
19              }
```

```
20
21          return data
```

The ___*getitem*___ method will return 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 will load and preprocess the data. Finally we insert it into a small dictionary and return it.

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

```
1   PALETTE = {
2       (0,  0,  0)  : 0,  #Human
3       (255, 255, 255) : 1,  #Background
4   }
5
6   def load_image(self, path=None):
7       raw_image = Image.open(path)
8       raw_image = np.transpose(raw_image.resize((224,224)), (2,1,0))
9       imx_t = np.array(raw_image, dtype=np.float32)/255.0
10
11      return imx_t
12
13  def load_mask(self, path=None):
14      raw_image = Image.open(path)
15      raw_image = raw_image.resize((224,224))
16      imx_t = np.array(raw_image)
17      label_seg = np.zeros((2,224,224), dtype=np.int)
18
19      for k in PALETTE:
20          label_seg[PALETTE[k]][(imx_t==k).all(axis=2)] = 1
21
22      return label_seg
```

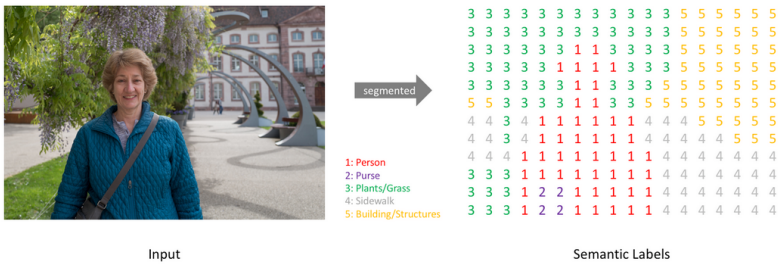
It is important to remark that the input data format for the masks is $(N \times C \times H \times W)$ where N is the batch size, C the number of classes and $H \times W$ the height and width of the image. The segmentation masks are one-hot encoded, which means each class has a $(1 \times H \times W)$ 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, we iterate 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 4.12.

Also, when loading images from the UnrealROX dataset, segmentation masks are slightly different since we have more than 30 classes. Listing 4.8 shows a small script that encodes the segmentation masks so that it only contains the human and background class. Result can be seen in Figure 4.13.

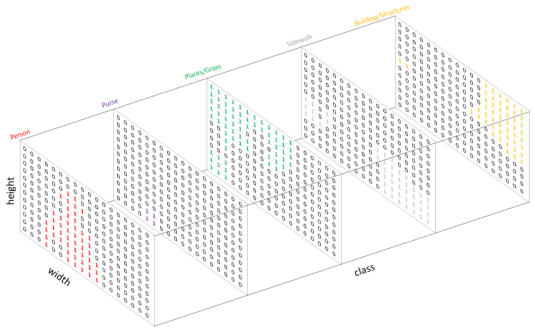
Listing 4.8: Preprocessing the UnrealROX segmentation masks

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

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



(a) Regular segmentation mask codification.



(b) One-hot encoded masks.

Figure 4.12: One-hot encoding format from a regular segmentation mask. Extracted from Jeremy Jordan semantic segmentation post[1].

```
4
5 label_seg[0][(imx_t==[255,0,0]).all(axis=2)] = 1
6 label_seg[1][np.where(label_seg[0] == 0)] = 1
```

It works similarly to the one-hot encoding on the *load_mask* method on Listing 4.7, 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, then for the second layer we simply invert the first layer to obtain the background.

4.3.2 Training the network

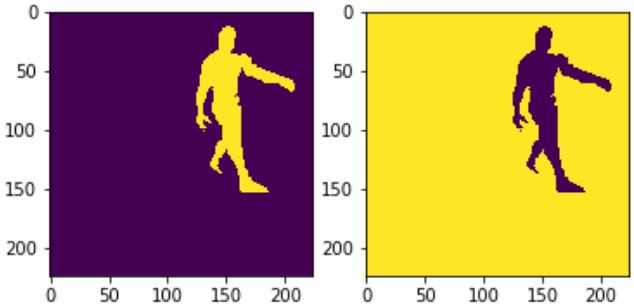
In this Subsection we will go through the network implementation (Subsection 4.3.2.1) and the training script (Subsection 4.3.2.2).

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, to sum it up, the encoder part is exactly the same as the VGG-16 shown in Figure 4.14 while the decoder performs the reverse up-convolution as de decoder. This Subsection describes the model implementation we used[2] for this work.

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

(a) Sample UnrealROX segmentation mask.



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

Figure 4.13: UnrealROX segmentation masks before and after the pre-process pass.

The encoder and decoder layout equivalent in PyTorch is shown in Listings 4.9 and 4.10. The decoder fundamentally consists of convolutional layers followed by a batch normalization. At the decoder, the convolutions are replaced by Convolutional Transposed layer, which applies transposed convolution that upsamples the outputs, batch normalization is also applied in every layer.

Listing 4.9: 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)])
```

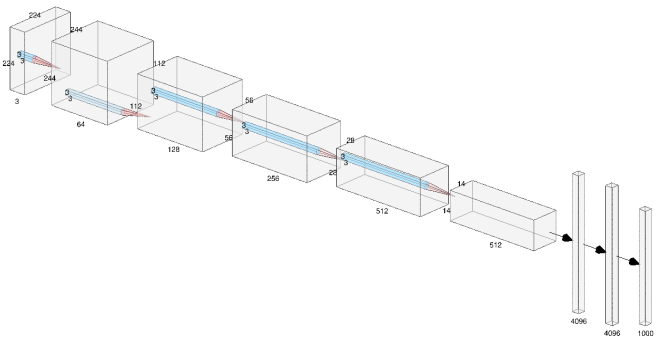


Figure 4.14: Illustration of the VGG-16 architecture.

Listing 4.10: 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.↩
    ↪ output_channels, kernel_size=3, padding=1)])
```

Then, the forward function of the model is shown in Listings 4.11 and 4.12. On the decoder, as with every convolutional network, we apply the activation function between layers and pooling at the end of the convolution. In 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 4.11: 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 4.12: 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 small training script was built. To do this, we first need to create our data loaders and split the data into train and validation splits. When training, we can not only use the training loss as a metric, since the network can over-fit the training samples, meaning that it will not be good at generalization. Listing 4.13 shows how the PyTorch DataLoader[3] class can be use to create a data loader.

Listing 4.13: 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

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 will allow us to easily differentiate the phase in every epoch.

Listing 4.14: 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 4.14 shows the definition of our model, the criterion, which is the metric that will be 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 will be handling how we modify the network weights based on different criteria, in our case, we used the Adam optimizer.

With all of the initialization ready, we can now start training, Algorithm 1 shows a simplified version of the training script. For every epoch, we will process the whole dataset, which is divided in two different loaders, the *phase* loop will determine which phase is currently being processed and we change our model to the proper phase using the *train* and *eval* methods,

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

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 the loss is computed based on 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.

Algorithm 1: Training script

```
for epoch < EPOCHS do
    for phase in ['train', 'val'] do
        if phase == 'train' then
            model.train()
        else
            model.eval()
        for batch in data_loaders[phase] do
            input, target = batch['image'], batch['mask']
            output = model(input)
            loss = criterion(output, target)
            if phase == 'train' then
                loss.backward()
                optimizer.step()
            running_loss += loss
```

In order to prevent our model to overfit, model checkpoints were implemented. This method, although very simple, allows us to keep the best model obtained during training. Listing 4.15 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 will write to disk the *mode.state_dict*() which is a dictionary containing the model's learnable parameters (weights and biases).

Listing 4.15: Model checkpoints

```
if running_loss < prev_loss:
    torch.save(model.state_dict(), os.path.join('.', 'epoch-{}.pth'.format(epoch+1)))
    prev_loss = running_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, Listing 4.16 shows an example usage. It is important to note that this will only work for inference and not for training, since there are multiple parameters like the optimizer or the loss metrics that are not saved on the *state_dict*, however, PyTorch does provide with a custom *save* method that allows to save a custom *dict* and thus allows to save all the necessary parameters so that it can be used to resume training.

Listing 4.16: Load model checkpoint

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

5 Results

This chapter will go through the results of our experimentation with the previously described implementations. Section 5.1 will review the results when training without synthetic data. Section 5.2 will focus on the synthetic data experimentation.

5.1 Training with no synthetic data

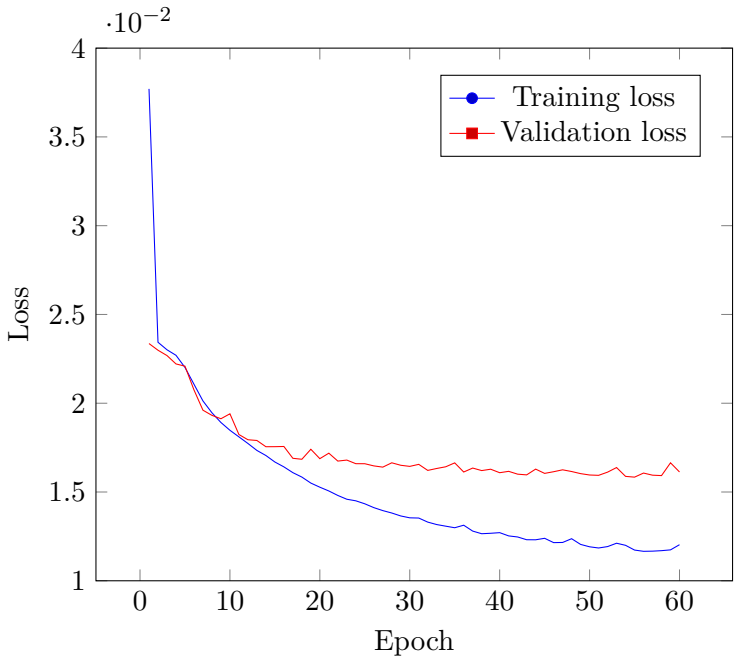


Figure 5.1: Training and validation loss without synthetic data.

5.2 Adding synthetic data to the training dataset

6 Conclusions

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`

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)*.

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

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)*.

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`

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