GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

Real-Time Egocentric Segmentation of Local Reality for Extended Reality Applications

Autor: Jorge Calvar Seco

Director: Ester González Sosa

Madrid

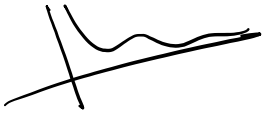
Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Real-Time Egocentric Segmentation of Local Reality

for Extended Reality Applications

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2022 - 2023 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Jorge Calvar Seco Fecha: 02/06/2023



Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Ester González-Sosa Fecha: 02/06/2023

GONZALEZ SOSA ESTER - 44735146P

Firmado digitalmente por GONZALEZ SOSA ESTER - 44735146P
Fecha: 2023.06.02 16:01:37 +02'00'

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

Real-Time Egocentric Segmentation of Local Reality for Extended Reality Applications

Autor: Jorge Calvar Seco

Director: Ester González Sosa

Madrid

SEGMENTACIÓN EGOCÉNTRICA EN TIEMPO REAL DE LA REALIDAD LOCAL PARA APLICACIONES DE REALIDAD EXTENDIDA

Autor: Calvar Seco, Jorge
Director: Gonzalez Sosa, Ester
Entidad Colaboradora: Nokia XR Lab

RESUMEN DEL PROYECTO

En este proyecto, aplicamos métodos de aprendizaje automático para sumergir a las personas en una realidad virtual. Nuestro objetivo es detectar qué píxeles de una imagen pertenecen al cuerpo humano y los objetos que dicha persona está tocando. Hemos implementado modelos de segmentación semántica de última generación y los hemos entrenado con varios conjuntos de datos. Los mejores resultados se han logrado con la arquitectura PIDNet entrenada con el conjunto de datos EgoHOS.

Palabras clave: segmentación semántica, visión artificial, realidad virtual

1. Introducción

La segmentación semántica es un problema de aprendizaje profundo que consiste en clasificar cada píxel de una imagen de entrada. Por lo tanto, la salida del modelo tiene el mismo ancho y alto que la entrada. Este campo ha mejorado constantemente desde el año 2015, con la publicación del modelo U-Net (Olaf Ronneberger, 2015). Este artículo propuso una arquitectura modelo de codificador-decodificador. En el codificador, el tamaño de entrada se reduce progresivamente a través de una red neuronal convolucional mientras aumenta la cantidad de canales, lo que se logra al establecer una gran cantidad de filtros en la capa convolucional. El proceso de codificación se invierte en el decodificador. Una contribución clave de U-Net es el uso de conexiones de salto, que se muestran en las flechas grises de la Figura 1. Estas conexiones consisten en concatenar en cada paso del decodificador el array en el paso correspondiente del codificador. Facilitan la regeneración de la salida del modelo con las características espaciales de la imagen de entrada.

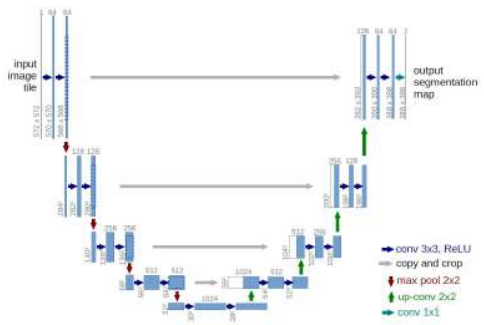


Figura 1: Arquitectura del modelo U-Net

En este proyecto, aplicaremos la segmentación semántica al campo de la realidad virtual. Nuestro objetivo es diferenciar qué partes de una imagen pertenecen al cuerpo humano y a los objetos que la persona está tocando. Las imágenes provendrán de una cámara

estéreo de unas gafas de realidad virtual (es decir, la cámara verá lo que ve la persona que la usa). Al segmentar al humano y al objeto, podremos eliminar el fondo y llevarlos a una realidad virtual.

Las aplicaciones de esta tecnología abarcan una variedad de industrias, incluida la educación, los juegos y la atención médica. Por ejemplo, podría ser utilizado por médicos para practicar operaciones en un escenario realista, pero sin riesgo.

2. Metodología

La implementación de los modelos de aprendizaje profundo se has realizado utilizando TensorFlow o PyTorch para Python 3.8. Todo el entrenamiento se ha realizado en un ordenador de sobremesa Ubuntu equipado con una NVIDIA RTX 3090 con 24Gb de RAM.

Los modelos con los que hemos experimentado son:

- Thundernet (Ester Gonzalez-Sosa, 2022): Tiene una arquitectura de codificador-decodificador. En el medio, hay un módulo de agrupación piramidal, que aplica un conjunto de operaciones de pooling con cuatro tamaños diferentes. Las salidas de esto están concatenadas. Esto permite que el modelo entienda objetos que vienen en diferentes tamaños.

- PIDNet (Jiacong Xu, 2022): Hemos optado por implementar este modelo debido a los excelentes resultados que ha logrado en otros conjuntos de datos famosos (cityscapes y camvid), tanto en términos de precisión como de tiempo de inferencia. Tiene tres ramas: una aprende información detallada, la otra aprende información de contexto y la última decide cómo se fusionarán las dos anteriores.

- YOLOv8 (Joseph Redmon, 2015): lo hemos usado con pesos preentrenados del conjunto de datos COCO.

Los conjuntos de datos egocéntricos que hemos utilizado para entrenar los modelos son:

- EgoBodies (Ester Gonzalez-Sosa, 2022): Es un conjunto de datos elaborado mediante la unión de otros conjuntos. Aunque el conjunto de datos original tiene muchos objetos, solo hemos conservado un subconjunto reducido de objetos de un entorno de oficina, incluidos un teléfono y una computadora portátil.

- EgoHOS (Lingzhi Zhang, 2022): Es un conjunto de datos público que contiene elementos con escenas de orígenes increíblemente diversos, como cocinar, hacer jardinería o trabajar con una computadora portátil. Lo hemos transformado a un tamaño de 640x480 y aplicado algunas otras transformaciones de datos. La clave de clase final son los brazos (clase 1) y cualquier objeto que se toque (clase 2), que es diferente de EgoBodies, que etiqueta objetos específicos.

Además, tenemos dos conjuntos de datos exocéntricos, filtrando imágenes con personas de COCO y ADE20K.

Para alimentar las imágenes a un modelo de aprendizaje profundo durante el entrenamiento, hemos creado un generador de datos, que se puede integrar a la perfección con ambos frameworks de aprendizaje profundo. Permite varias

transformaciones de datos, incluido el cambio de tamaño, la conversión a estéreo o la aplicación de cambios de clases. También puede aplicar técnicas de *data augmentation* o combinar múltiples conjuntos de datos.

La principal métrica utilizada para medir la calidad de los modelos es la intersección sobre la unión (IoU), que se calcula con las máscaras tanto de la realidad del terreno como de la predicción del modelo.

3. Experimentos y resultados

El experimento inicial, entrenar el modelo Thundernet con el conjunto de datos EgoBodies, logró un IoU de la clase humano del 54,84% y un IoU de objetos del 28,91%. Este es un IoU de objeto muy bajo y una disminución significativa de IoU humano con respecto al modelo sin objetos. Para tratar de mejorar los resultados, hemos realizado muchos experimentos. Los experimentos fallidos incluyeron:

- Reducción del muestreo del conjunto de datos eliminando imágenes donde solo estaba presente la clase humano (sin objetos). El objetivo era aumentar el porcentaje de imágenes con objetos. La suposición, que resultó incorrecta, era que esto aumentaría la calidad de la segmentación de objetos.

- Aumentar la tasa de aprendizaje.

- Aumento del tamaño de las imágenes. Como las imágenes se reducen de tamaño en el codificador del modelo, podríamos suponer que, si las imágenes son demasiado pequeñas, perderán demasiada información en este proceso.

- Aplicar técnicas de aumento de datos, incluidas transformaciones de volteo y color, como cambiar la saturación o el contraste.

Sin embargo, los resultados mejoraron con los siguientes experimentos:

- Incrementar el número de bloques de pooling en el PPM, de cuatro a seis. Al agregar dos nuevos tamaños de agrupación, el modelo pudo aprender más información de contexto, lo que mejoró significativamente la segmentación de objetos y humanos.

- Agregar una capa Attention U-Net, que da la capacidad al decodificador de enfocarse en las partes de una imagen que son más importantes para generar la salida, lo que aumenta la calidad de la segmentación.

El modelo PIDNet fue entrenado con el conjunto de datos EgoHOS. Se probaron varias configuraciones de tamaño, así como entrenamiento con imágenes estéreo y mono. En este caso, aumentar el tamaño de 640 x 480 a 1280 x 960 también mejoró la calidad de la segmentación, como se muestra en la Figura 2. Los mejores resultados finales son 90,95% de IoU de la clase humano y 54,35% de IoU de objetos. También realizamos una prueba de tiempo de inferencia para verificar que ejecutar PIDNet con imágenes de 1280x960 es lo suficientemente rápido para el procesamiento en tiempo real. Obtuvimos un tiempo de inferencia promedio de 12ms, lo que confirma que sí lo es.

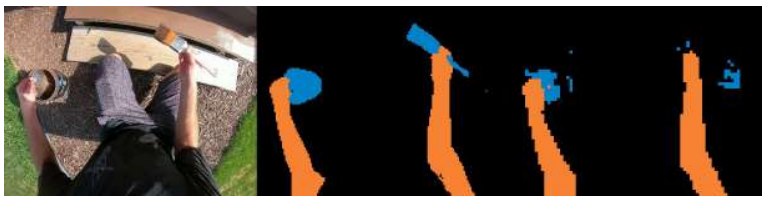


Figura 2: PIDNet entrenado con un tamaño de 1280x960 (izquierda) y un tamaño de 640x480 (derecha)

Finalmente, hemos probado el modelo YOLO segmentando los objetos laptop y phone. Consigue buenos resultados, aunque no tan buenos como Thundernet.

4. Conclusiones

Entre las conclusiones que extraemos del modelo de Thundernet se encuentra el hecho de que lograr buenos resultados en IoU de objetos tiene un costo en el IoU de la clase humano. También demostrado que pequeños cambios en la arquitectura, como agregar factores de agrupación al PPM o agregar una capa Attention U-Net, pueden generar mejoras significativas en la calidad de la segmentación.

Hemos hecho una comparación entre Thundernet y el modelo YOLOv8 preentrenado. YOLOv8 muestra resultados prometedores, especialmente en términos de calidad de segmentación cuando se identifican los objetos correctamente. Thundernet funciona mejor en términos de detección de objetos, pero tiene algunas imprecisiones en sus máscaras de segmentación.

No es posible una comparación directa con PIDNet, ya que las clases son diferentes. Sin embargo, PIDNet entrenado con EgoHOS obtuvo muy buenos resultados en la segmentación de objetos que interactúan, logrando un IoU humano del 90,95% y un IoU de objetos del 54,35%. La sensación de inmersión con PIDNet fue la más alta de todos los modelos probados.

En conclusión, hemos demostrado la alta viabilidad del uso de la segmentación semántica para aplicaciones de realidad virtual del mundo real. El principal obstáculo para obtener mejores resultados es la ausencia de suficientes datos etiquetados, lo que es muy costoso de obtener, ya que generalmente se etiqueta manualmente. Hemos explorado una forma de resolver esto, que es usar modelos de segmentación semántica que no son para tiempo real de última generación para etiquetar datos, que se pueden usar para entrenar modelos en tiempo real. Hemos mostrado el potencial de esta idea, aunque todavía tiene limitaciones.

5. Referencias

Ester Gonzalez-Sosa, A. G.-M. (2022). Real Time Egocentric Segmentation for Video-self Avatar in Mixed Reality. *arxiv*.

Jiacong Xu, Z. X. (2022). PIDNet: A Real-time Semantic Segmentation Network Inspired by PID Controllers. *CVPR2023*.

Joseph Redmon, S. D. (2015). You Only Look Once: Unified, Real-Time Object Detection. *arxiv*.

Lingzhi Zhang, S. Z. (2022). Fine-Grained Egocentric Hand-Object Segmentation: Dataset, Model, and Applications. *arxiv*.

Olaf Ronneberger, P. F. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. *MICCAI 2015*.

REAL-TIME EGOCENTRIC SEGMENTATION OF LOCAL REALITY FOR EXTENDED REALITY APPLICATIONS

Author: Calvar Seco, Jorge
Supervisor: González Sosa, Ester
Collaborating Entity: Nokia XR Lab

ABSTRACT

In this dissertation, we apply machine learning methods to immerse people into a virtual reality. We aim to detect which pixels of an image belong to the human body and the objects that the person is touching. We have implemented state-of-the-art semantic segmentation models and trained them with several datasets. The best results have been achieved with the PIDNet architecture trained with the EgoHOS dataset.

Keywords: semantic segmentation, computer vision, virtual reality

1. Introduction

Semantic segmentation is a deep learning problem that consists in classifying every pixel of an input image. Therefore, the model's output has the same width and height as the input. This field has steadily improved since the year 2015, with the publication of the U-Net model (Olaf Ronneberger, 2015). This paper proposed an encoder-decoder model architecture. In the encoder, the input size is reduced progressively through a convolutional neural network while the number of channels increases, which is achieved by setting a high number of filters in the convolutional layer. The encoding process is reversed in the decoder. A key contribution of U-Net is the use of skip connections, shown in the gray arrows from Figure 1. These connections consist in concatenating at each step in the decoder the array at the corresponding step in the encoder. They facilitate the regeneration of the model's output with the spatial features of the input image.

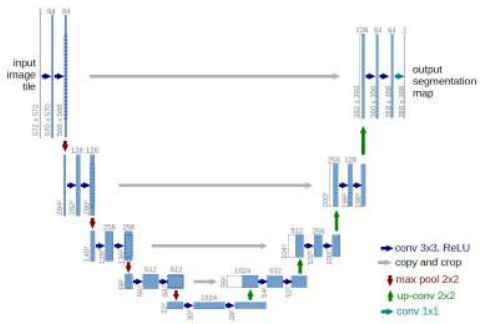


Figure 1: Architecture of the U-Net model

In this dissertation, we will apply semantic segmentation to the field of virtual reality. We aim to differentiate which parts of an image belong to the human body and to objects that the person is touching. The images will come from a stereo camera on a VR headset (i.e., the camera will see what the person wearing it is seeing). By segmenting the human and the object, we will be able to remove the background and bring them to a virtual reality.

The applications of this technology span across a variety of industries, including education, gaming, and healthcare. For example, it could be used by doctors to practice procedures in a realistic scenario but without risk.

2. Methodology

The implementation of the deep learning models has been done using either the TensorFlow or PyTorch frameworks for Python 3.8. All the training has been performed on an Ubuntu desktop computer equipped with an NVIDIA RTX 3090 with 24Gb RAM.

The models we have experimented with are:

- Thundernet (Ester Gonzalez-Sosa, 2022): It has an encoder-decoder architecture. In the middle, there is pyramid pooling module, which applies a set of average pooling operations with four different sizes. The outputs of this are concatenated. This allows the model to understand objects that come in different sizes.

- PIDNet (Jiacong Xu, 2022): We have chosen to implement this model because of the great results it had achieved on other famous datasets (cityscapes and camvid), both in terms of accuracy and inference time. It has three branches: one learns detailed information, the other learns context information, and the last one decides how the former two will be merged.

- YOLOv8 (Joseph Redmon, 2015): We have used it with pre-trained weights from the COCO dataset.

The egocentric datasets we have used to train the models are:

- EgoBodies (Ester Gonzalez-Sosa, 2022): It comprises 8005 training images and 1069 validation images, all in jpg format and 640x480 size. Although the original dataset has many objects, we have only kept a reduced set of objects from an office setting, including a phone and a laptop.

- EgoHOS (Lingzhi Zhang, 2022): It is a public dataset, which contains items with scenes from incredibly diverse backgrounds, including cooking, gardening, or working with a laptop. We have transformed it to 640x480 size and applied some other data transformations. The final class key is the arms (class 1) and any object being touched (class 2), which is different from EgoBodies, which labels specific objects.

Additionally, we have two exocentric datasets, filtering images with people from COCO and ADE20K.

To feed the images into a deep learning model, we have created a data generator, which can be seamlessly integrated with both deep learning frameworks. It allows for several data transformations, including changing size, converting to stereo, or applying class mappings. It can also apply data augmentation techniques or combine multiple datasets.

The main metric used to measure the quality of the models is the intersection over union (IoU), which is computed with the masks of both the ground truth and the model's prediction.

3. Experiments and results

The initial experiment training the Thundernet model with the EgoBodies dataset achieved a 54.84% human IoU and 28.91% object IoU. This is a very low object IoU and a significant decrease in human IoU with respect to the model without objects. To try to improve the results, we have run many experiments. Unsuccessful experiments included:

- Downsampling the dataset removing images where only a human (without objects) was present. The objective was to increase the percentage of images with objects. The assumption, which was proven incorrect, was that this would increase the quality of object segmentation.

- Increase the learning rate.

- Increasing the size of the images. As images are reduced in size in the encoder of the model, we could assume that if the images are too small, they will lose too much information in this process.

- Applying data augmentation techniques, including flip and color transformations, such as changing the saturation or contrast.

However, the results improved with the following experiments:

- Incrementing the number of pooling blocks in the PPM, from four to six. By adding two new pooling sizes, the model was able to learn more context information, which significantly improved the segmentation of objects and humans.

- Adding an Attention U-Net gate, which gives the ability to the decoder to focus on the parts of an image that are most important to generate the output, increasing the segmentation quality.

The PIDNet model was trained with the EgoHOS dataset. Several size configurations were tried, as well as training with both stereo and mono images. In this case, increasing size from 640x480 to 1280x960 also improved the segmentation quality, as shown in Figure 2: PIDNet trained with 1280x960 size (left) and 640x480 size (right). The final best results are 90.95% human IoU and 54.35% object IoU. We also run an inference time test to check that running PIDNet with 1280x960 images was fast enough for real-time processing. We obtained an average inference time of 12ms, confirming this.

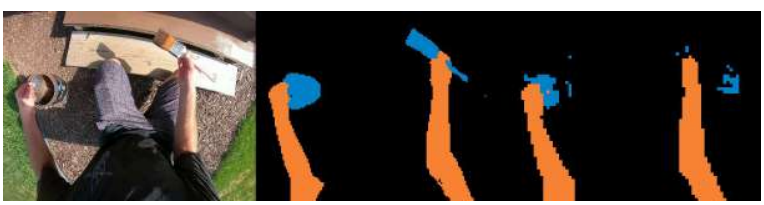


Figure 2: PIDNet trained with 1280x960 size (left) and 640x480 size (right)

Finally, we have tested the YOLO model segmenting the laptop and phone objects. It achieves good results, although not as good as Thundernet.

4. Conclusions

Among the conclusions that we extract from the Thundernet model is that achieving good results in object IoU comes at the cost of human IoU. Additionally, we have shown that small changes in the architecture, such as adding pooling factors to the PPM or adding an Attention U-Net gate, can bring significant improvements in segmentation quality.

We have made a comparison between Thundernet and the pre-trained YOLOv8 model. YOLOv8 shows promising results, especially in terms of segmentation quality when correctly identifying objects. Thundernet performs better in terms of object detection but has some inaccuracies in its segmentation masks.

A direct comparison with PIDNet is not possible, as the classes are different. However, PIDNet trained with EgoHOS obtained very good results in segmenting interacting objects, achieving a human IoU of 90.95% and an objects IoU of 54.35%. The feeling of immersion with PIDNet was the highest of all models tried out.

In conclusion, we have shown the high viability of using semantic segmentation for real-world virtual reality applications. The main obstacle to obtaining better results is the absence of enough labeled data, which is very costly to obtain, as it is usually labeled manually. We have explored a way to solve this, which is using state-of-the-art not-for-real-time semantic segmentation models to label data, which can be used to train real-time models. We have shown the potential of this approach, although it still has limitations.

5. Referencias

Ester Gonzalez-Sosa, A. G.-M. (2022). Real Time Egocentric Segmentation for Video-self Avatar in Mixed Reality. *arxiv*.

Jiacong Xu, Z. X. (2022). PIDNet: A Real-time Semantic Segmentation Network Inspired by PID Controllers. *CVPR2023*.

Joseph Redmon, S. D. (2015). You Only Look Once: Unified, Real-Time Object Detection. *arxiv*.

Lingzhi Zhang, S. Z. (2022). Fine-Grained Egocentric Hand-Object Segmentation: Dataset, Model, and Applications. *arxiv*.

Olaf Ronneberger, P. F. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. *MICCAI 2015*.

Firstly, I would like to thank my family for always supporting me. I would also like to thank my director, Ester, for her guidance and invaluable help throughout the development of this project.

Abstract

This dissertation explores the application of machine learning methods to enable immersive virtual reality experiences. The goal is to detect and classify the pixels belonging to the human body and the objects being interacted with, using state-of-the-art semantic segmentation models. This technology has broad applications in fields such as education, gaming, and healthcare. Throughout the project we use various datasets and implement advanced architectures. An important limitation is that the model must be fast enough to run in real-time. The models explored include Thundernet, PIDNet, and YOLOv8, each with its unique strengths and applications. The training datasets used consist of egocentric and exocentric images, focusing on the human body and specific objects the person is interacting with. We run many experiments looking for quality improvements. The best results were achieved using the PIDNet model trained on the EgoHOS dataset, achieving a human IoU of 90.95% and an objects IoU of 54.35%. In conclusion, this research demonstrates the potential of semantic segmentation for real-world virtual reality applications. However, the need for labeled data remains a challenge, as manual labeling is costly and time-consuming. We slightly explore leveraging state-of-the-art models for labeling. This showed promising results but has its limitations.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Objectives . . . 3

2 Theoretical Framework and State of the art **5**
- 2.1 Fundamentals of deep learning . . . 5
  - 2.1.1 Layers . . . 5
  - 2.1.2 Loss functions . . . 8
  - 2.1.3 Optimizer . . . 9
- 2.2 Computer vision . . . 9
- 2.3 Semantic Segmentation . . . 11

3 Methodology **15**
- 3.1 Technical considerations . . . 15
- 3.2 Generating data . . . 16
- 3.3 Objective evaluation . . . 18
  - 3.3.1 Quality . . . 18
  - 3.3.2 Inference time . . . 19
- 3.4 Subjective evaluation . . . 20

4 Datasets **21**
- 4.1 Egocentric datasets . . . 21
  - 4.1.1 EgoBodies . . . 21
  - 4.1.2 EgoHOS . . . 24
- 4.2 Exocentric datasets . . . 25
  - 4.2.1 COCO . . . 26
  - 4.2.2 ADE20K . . . 26

5 Architectures **29**
- 5.1 Thundernet . . . 29
  - 5.1.1 Training environment . . . 30

5.2 PIDNet . . . 30
5.2.1 Training environment . . . 32
5.3 YOLO . . . 33

6 Experiments **35**
6.1 Training with EgoBodies (5 objects) . . . 35
6.1.1 Initial experiment . . . 35
6.1.2 Downsampled dataset . . . 36
6.1.3 Increasing learning rate . . . 37
6.1.4 Kernel regularizer . . . 37
6.1.5 Increasing size . . . 37
6.1.6 Training the resnet layer . . . 39
6.1.7 Data augmentation . . . 39
6.1.8 Conclusions . . . 40
6.2 Changes to the Thundernet architecture . . . 41
6.2.1 Modifying the PPM . . . 41
6.2.2 Adding an attention block . . . 43
6.3 Training Thundernet with EgoHOS . . . 46
6.4 Training PIDNet with EgoHOS . . . 46
6.4.1 Inference time . . . 48
6.5 YOLO . . . 49
6.6 Exploring exocentric segmentation . . . 51

7 Parallel problems **55**
7.1 Interpolation . . . 55
7.2 Solving data scarcity and invariance . . . 57

8 Conclusions **59**
8.1 Next steps . . . 60

Bibliography **63**

A Evolution of loss during training **67**

B Alignment with SDGs **71**

List of Figures

1.1 Example of a VR immersive experience [1] . . . 2
1.2 Example of an stereo image . . . 4

2.1 Examples of activation functions . . . 6
2.2 Convolution operation . . . 7
2.3 Example of max pooling operation . . . 7
2.4 Architecture of LeNet [9] . . . 10
2.5 Residual block . . . 11
2.6 U-Net architecture [14] . . . 12
2.7 Atrous convolution [17] . . . 13

3.1 Examples of Intersection over Union [22] . . . 19

4.1 Examples of the EgoBodies dataset . . . 21
4.2 Frequencies of the different classes of the EgoBodies dataset . . . 23
4.3 Examples of the EgoHOS dataset . . . 24
4.4 Labeled examples of EgoHOS after processing . . . 25
4.5 Distribution of the classes of the processed EgoHOS dataset . . . 26
4.6 Labeled examples of the selected images from both exocentric datasets after transformations . . . 27

5.1 Architecture of the Thundernet model [23] . . . 30
5.2 Architecture of the PIDNet model [29] . . . 31
5.3 Comparison of accuracy and inference speed [29] . . . 32
5.4 Part of the YOLO architecuture [30] . . . 34

6.1 Histogram of objects before (blue) and after (gray) downsampling . . . 36
6.2 Validation IoU of the model with and without kernel regularizer . . . 38
6.3 Images before and after increasing the size . . . 38
6.4 Examples of model with (left) and without (right) object segmentation . . . 41
6.5 Architecture of the new PPM (including the new pooling sizes) . . . 42
6.6 Implementation of the Attention Gate . . . 44

6.7 Applications of the attention gate to the Thundernet architecture . 45
6.8 Example of how PIDNet segments with the models trained at 1280x960 (left) and 640x480 (right) 48
6.9 Inference time of the PIDNet-L model for different input sizes . . . 50
6.10 Example of an image segmented with YOLOv8 51
6.11 Examples of images segmented with the PIDNet exocentric model . 54

7.1 Image (top left) and PIDNet prediction using nearest (top right), bilinear (bottom left) and bicubic (bottom right) interpolation . . . 56
7.2 Images of Figure 7.1 zoomed to a specific part 56
7.3 Images of the EgoBodies dataset and their predictions using a swin transformer 58

A.1 Evolution of validation IoU during the initial Thundernet experiment 68
A.2 Validation IoU of the Thundernet downsampled and initial models . 68
A.3 Validation IoU of the individual objects from the Thundernet downsampled model 69
A.4 Validation IoU of the fast lr and initial Thundernet models 69
A.5 Validation IoU of original (640x480) and increased-size (1280x480) Thundernet models 70
A.6 Validation IoU of the Thundernet model with (light gray) and without (dark gray) trainable resnet layer 70

List of Tables

3.1 Parameters of the `DataGenerator` 18

4.1 Original key of the Egobodies dataset 22
4.2 Class key of the 5-office-objects EgoBodies dataset 23
4.3 Original class key of the EgoHOS dataset [26] 24
4.4 Class key of the EgoHOS dataset after processing 25

5.1 Default training configuration for Thundernet 30
5.2 Number of parameters by model 32
5.3 Default training configuration for PIDNet 33

6.1 IoU (in %) of the initial Thundernet experiments with objects . . . 36
6.2 IoU (in %) of the initial and downsampled experiments 37
6.3 IoU (in %) of the experiments with fast and default learning rate . 37
6.4 IoU (in %) of the initial and increased-size models 39
6.5 IoU (in %) of the models with and without training the resnet layer 39
6.6 IoU (in %) for every class and transformation applied 40
6.7 IoU (in %) of the new PPM model 43
6.8 IoU (in %) of the experiments run with the attention layer (separated objects) 44
6.9 IoU (in %) of the experiments run with the attention layer (objects as one) 45
6.10 IoU (in %) of the experiments run with Thundernet and EgoHOS . 46
6.11 IoU (in %) of the experiments run with PIDNet and EgoHOS . . . 47
6.12 PIDNet-L inference time (rounded to nearest ms) for several sizes and GPUs (95% confidence interval) 49
6.13 IoU (in %) and inference time of the different sizes of the YOLOv8 model for the EgoBodies validation set (640x480) 50
6.14 Class key of the exocentric model 51
6.15 IoU (in %) of the experiment with exocentric humans 52

Chapter 1

Introduction

1.1 Motivation

In recent years, a new industry has emerged that will change how we perceive and interact with the world: virtual reality (VR). This technology offers immersive experiences with applications in many sectors, from entertainment to healthcare.

The potential of VR technology is huge and continues to grow exponentially. Some of the most prominent tech companies are investing heavily in the field. For example, Meta spends more than $10B annually on the Metaverse, which is their VR platform.

There are applications of VR in plenty of sectors, including:

- Education: VR can redefine learning experiences, making them more engaging and effective. Students can explore the solar system, or even dive into the human body, all from their classrooms. This immersive approach to learning can help improve comprehension and retention, making education more accessible and enjoyable.

- Healthcare: VR can offer tools for training, therapy, and visualization. Surgeons can rehearse complex procedures in a risk-free environment, therapists can treat phobias with controlled exposure therapy, and medical students can study anatomy in 3D. By pushing the boundaries of what's possible in healthcare, VR will improve outcomes and save lives.

- Gaming: Gamers will be active participants within the game's universe, able to manipulate objects, navigate environments, and engage with characters as if they were in the same physical space. This level of immersion has also paved the way for innovative game design, opening up unprecedented possibilities for storytelling, exploration, and player engagement.

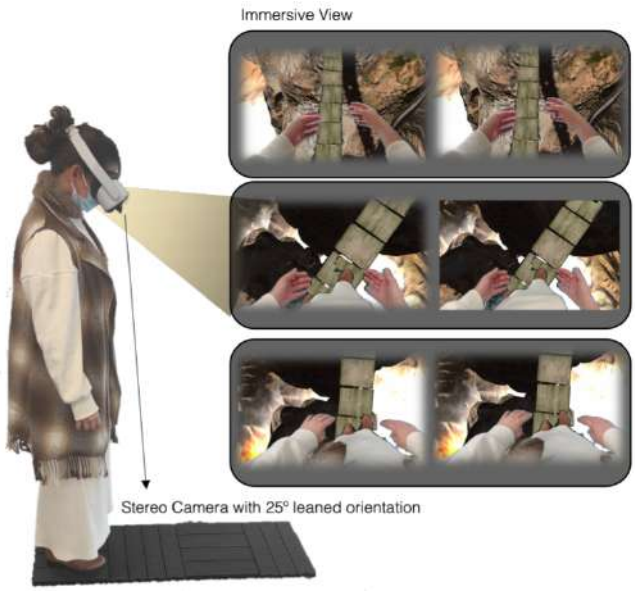


Figure 1.1: Example of a VR immersive experience [1]

There are many research lines to improve VR experiences. One is how to include self-avatars [1] in the virtual reality, which gives the user the possibility to observe his own body. Among the benefits of this is that the sense of presence is increased, the user is not a mere observer (as when watching tv), but instead is an entity who can interact with the virtual world [2]. Figure 1.1 shows an example of a VR immersive experience where the avatar is created using a camera located on the VR headset.

To include the human in the virtual reality (as in Figure 1.1), we must create an algorithm that differentiates which part of the image captured by the camera is the human body of the person wearing it. There are many approaches to achieve this. One of them is traditional chroma-based techniques, which depend on the user being located in a place where the background is always of the same color (normally green is used).

However, with the growth of machine learning, a task which has the potential to solve this problem has appeared: semantic segmentation. It is a computer vision technique that involves classifying each pixel in an image into predefined categories. The difference with a normal classification model is the output shape, which in the case of semantic segmentation has the same width and height of the input image.

Despite the progress in semantic segmentation, there are still numerous challenges to overcome. These include the accurate segmentation of objects in complex environments, the efficient processing of segmentation in real-time, and the robustness of models when facing varying lighting conditions and object orientations.

There has already been some research in applying semantic segmentation to virtual reality. Gonzalez-Sosa et al. [3] have achieved significant success in creating a system that is able to segment the human body in real time. They have trained a model inspired by Thundernet [4] with the THU-READ dataset [5] [6], achieving promising results. In this dissertation, we will continue this research, trying to improve the quality of the current algorithms and adding new features, such as the segmentation of objects.

1.2 Objectives

The broad objective of this project is to understand the current literature on real-time semantic segmentation with the intent to apply this technology for virtual reality use cases. We will place specific emphasys on the segmentation of objects in addition to the human body, and will explore and compare the different approaches.

The success of our algorithms will be measured both quantitatively and qualitatively. Measuring quantitatively consists of defining a set of metrics to track and optimize in our models. It will also let us compare and rank them to decide whether the new experiments have improved the quality or not.

However, it is also important to measure qualitatively, which consists of running the algorithm in real-time on our laptop and giving a subjective opinion of its performance. To do this, we will use a virtual reality headset with a camera, which sees the first-person view of the human wearing it. The images captured by the camera will be sent to our model, which will be responsible for detecting which parts of the image correspond to the human body of the wearer, along with certain objects. Then, we will create a mask that we will display on the screen. In this qualitative analysis, which is subjective, we must pay attention to the quality of the segmented parts and how close does the user consider his experience is to real life. The user will also have to report whether he feels that there is a lag in the images he sees in the virtual reality.

Another objective of the dissertation is comparing the different methods to segment objects, e.g., whether we are segmenting a specific object like a phone, or if we are segmenting any object that meets a property, such as touching the user's hand. We will analyze the different common approaches to these problems and present the results.

Lastly, we intent to discover what are the main bottlenecks that are impeding a faster development of these algorithms and make a recommendation on future lines of work.

There are several limitations to our work. One of the most important is that the algorithm is aimed to be used in real-time. It must be able to segment images at the same speed as they are produced by the camera. An important metric to track

Figure 1.2: Example of an stereo image

is the inference time. Many of the current state-of-the-art semantic segmentation models take too long to compute in real-time and, therefore, are not applicable to our use case. Another thing to consider is that the camera located in the VR headset is stereo, so we should take into account what is the best way to adapt our model to support stereo images, such as the one shown in Figure 1.2.

Chapter 2

Theoretical Framework and State of the art

2.1 Fundamentals of deep learning

Deep learning is a field of artificial intelligence, which uses neural networks to model complex functions. In this dissertation, we will be implementing several deep learning models. These networks are built by connecting a set of layers. We call the layers used and the way in which they are connected the network architecture. The parameters of those layers determine the relationship between the input and output, and we will need to optimize them to achieve the best possible result.

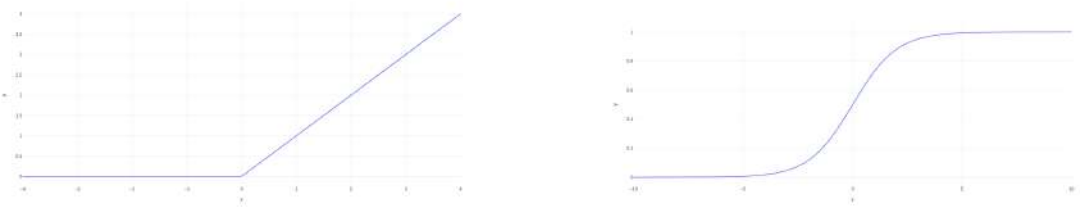
In this dissertation we are only concerned with supervised algorithms, which need labeled data to be trained. We will call these labels the ground truth, as they represent the output of the model should it be perfect.

2.1.1 Layers

In this subsection we briefly explain some of the most important layers that we will use.

Linear

A linear layer inputs and outputs a one-dimensional array, possibly of different sizes. Its characteristic is that there is a linear relationship between every output and all the inputs. Therefore, each output y_j can be modeled as $y_j = b_j + \sum_i w_i x_i$, where x_i and w_i are the values at position i of the input and weight arrays respectively. Because of this, the linear layer is also called fully-connected, as every input has the possibility of influencing every output.



(a) ReLU (b) Sigmoid

Figure 2.1: Examples of activation functions

Activation function

It can be demonstrated that if we apply any number of consecutive linear transformations, the resulting function will also be a linear transformation of the input. This is where the need for activation functions arises. They introduce non-linearity into the model. We will usually use activation functions after a linear or convolution layer.

Some of the most common activation functions, which we will use, are:

- ReLU: we use it between hidden layers of a neural network. The function is $f(x) = max(x, 0)$. ReLU outputs zero when the input is negative, and does not apply any modification otherwise. Shown in Figure 2.1a.
- Sigmoid: it is used as the output activation function of models that predict a probability. It is suitable for this use case because it maps the real domain to the $(0, 1)$ interval. The function is $f(x) = \frac{1}{1+e^{-x}}$, and is shown in Figure 2.1b.
- Softmax: it is used when we need to output a discrete probability distribution of more than two classes. It converts a one-dimensional input array x to a set of probabilities y that sum 1. The formula is $y_j = \frac{x_j}{\sum_i x_i}$.

Convolution

The convolution layer divides the input into smaller sections, which can be overlapping, and processes them separately. Each section is multiplied by a matrix of weights, which we call filter or kernel. The same filter is applied to all the sections. Depending on the dimension of the filters, we may have 1d, 2d, or 3d convolutions. However, we will only work with the 2d convolution, as we are processing images. In Figure 2.2 we observe the convolution operation being applied on one of the sections of an input. The output, which is the section multiplied by the filter (shown in red), corresponds to a pixel in the output array.

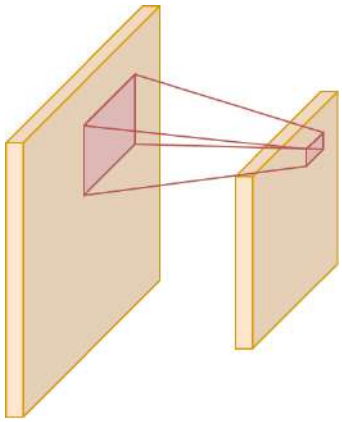


Figure 2.2: Convolution operation

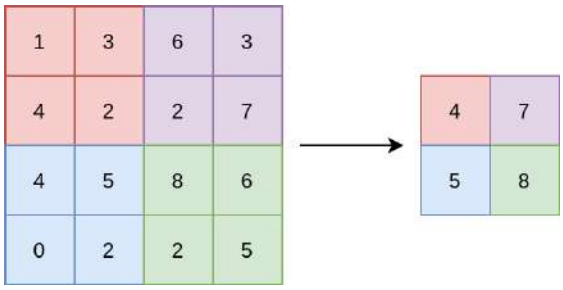


Figure 2.3: Example of max pooling operation

One of the properties of convolution layers is that they are translation invariant, which means that the result of applying a filter to a section of the input is the same, regardless of the location of that section. Additionally, they maintain the spatial relationships of the input. These properties make this layer incredibly useful for processing images.

Pooling

Pooling layers are often used after convolution layers to reduce the dimensions of the input. Similar to convolutions, they divide the input into sections, which in this case are normally not overlapping although they could be, and apply an operation to each section. The nature of this operation determines the type of pooling layer that we are using. The most common one is max pooling, in which the output of a section will be the maximum input value. If we divide an input image into non-overlapping sections of size 2x2, the output image will have half the width and height. An example of applying a max-pooling layer with sections of size 2x2 is shown in Figure 2.3.

Batch Normalization

Batch normalization is a layer whose introduction made training deep learning models significantly faster [7]. The problem addressed by this technique is the fact that, as the parameters of a layer change, the distribution of the inputs of the next layers also changes, making it very hard to adapt the weights. To solve this problem, the layer normalizes each mini-batch and applies a scale shift, as shown in the following equation:

$$y = \frac{x - \mathbb{E}[x]}{\sqrt{Var[x] + \epsilon}} \cdot \gamma + \beta \tag{2.1}$$

The value of ϵ is set very close to zero, as it used to avoid zero-division errors. γ and β are trainable parameters, which are used so that the layer can learn the mean and variance of the output distribution instead of forcing it to use zero-mean and unit-variance.

Upsample

This layer increases the size of an image using interpolation techniques, such as nearest or bilinear interpolation. It does not have parameters.

2.1.2 Loss functions

The training of a deep learning model is treated as an optimization problem. Therefore, we need to calculate a numerical value which we want to minimize. This value is called the loss, and it depends on the output of the model and the ground truth.

In our dissertation, the models will predict class probabilities. When we are creating a binary model (i.e., there are only two classes), we will use the binary cross-entropy loss, shown in Equation 2.2, where p_i is the output probability and y_i is the ground truth of batch element i. The ground truth may only be 0 or 1.

The logic behind this loss function is that, by taking the logarithm of the probability the model predicted for the correct class, the value of the loss will approach infinity as the output probability gets close to 0. Because of the non-linear nature of the logarithm function, it is much more beneficial to reduce the loss to change a probability from 0.1 to 0.2 than from 0.7 to 0.8. This causes the optimization algorithm to work harder to improve low-probability pixels than to get an even better result in high-probability ones.

$$L = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(p_i) + (1 - y_i)\log(1 - p_i)] \tag{2.2}$$

When we are predicting multiple classes, we will use the categorical cross-entropy loss (Equation 2.3). As we see, it is just an expansion of the binary loss but to account for an indefinite number of classes.

$$L = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{c} y_{i,j}\log(p_{i,j}) \tag{2.3}$$

2.1.3 Optimizer

An essential requirement of deep learning is that the loss of the model must be differentiable with respect to the parameters. This is because, to optimize, we will need to compute the gradient of the loss and subtract it from the parameters, as shown in Equation 2.4. This is called the gradient descent algorithm.

$$\theta_j = \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta) \tag{2.4}$$

The logic is that the gradient of a function at a certain point is in the direction where the slope is greatest. By subtracting the gradient, we are reducing the loss function as much as possible in that specific point. Something to be careful about is that, with this technique, we will be able to find a local minimum. However, our objective is to find a global minimum. If the function we are optimizing is not convex, we should not forget this consideration.

In particular, we will apply the stochastic gradient descent. As we cannot pass our whole dataset through the model at the same time because we do not have enough memory, we will divide it into smaller groups called batches, and update the weights using the optimization algorithm after each batch.

2.2 Computer vision

Computer vision is a field of machine learning that solves several tasks related to image recognition or generation. Some of the most common problems of computer vision are:

- **Classification:** it predicts whether an image contains a certain object.
- **Object detection:** it goes a step further and locates the object in the image using a bounding box.
- **Semantic segmentation:** it identifies the specific pixels in an image that belong to a specific object. It is similar to the classification problem but predicting a probability for each pixel and not the image as a whole.

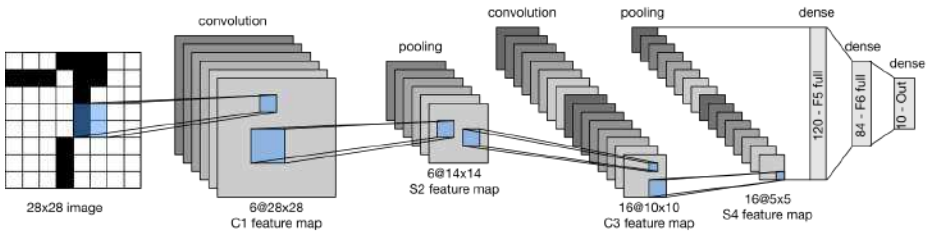


Figure 2.4: Architecture of LeNet [9]

The field of computer vision has advanced greatly in recent years. Convolutional neural networks (CNNs) have played a very important role in this. One of the earliest architectures was that proposed by Lecun et al. [8]. It is a simple combination of convolution and pooling layers, followed by dense layers, as shown in Figure 2.4. It was applied to handwritten digit recognition, in which it outperformed all previous methods.

In 2012, Krizhevsky, Sutskever, and Hinton presented a new model, Alexnet [10], with which they won the ImageNet competition. One of the innovations with respect to LeNet is the use of ReLU activation functions intead of tanh and sigmoid. Additionally, it included new layers, such as max pooling or dropout to regularize.

In 2014, a breakthrough was achieved with the proposal of VGGNet [11]. They showed that increasing the depth of CNNs and using small convolutional filters (3x3) significantly improved the performance. They created two networks, VGG-16 and VGG-19, where the number represents the depth. With them, they achieved the first and second positions in the 2014 ImageNet challenge.

Later, Google LeNet [12] outperformed VGGNet with the introduction of the Inception module. This module performs multiple convolutional operations in parallel on the same input, which are afterwards concatenated together.

The ResNet architecture [**resnet**], which stands for residual network, brought new advances to the field. They introduced the residual block, which contains skip connections. Basically, this means that the input of the block is summed to the output. This causes the model to learn variations to the input instead of a desired output. A schema of the residual block is shown in Figure 2.6. It has two weights layers (e.g., a convolution) and, between them, there is a ReLU and a batch normalization layer. ResNets have significantly influenced the development of computer vision, and they are heavily used in the models that we will implement in this dissertation.

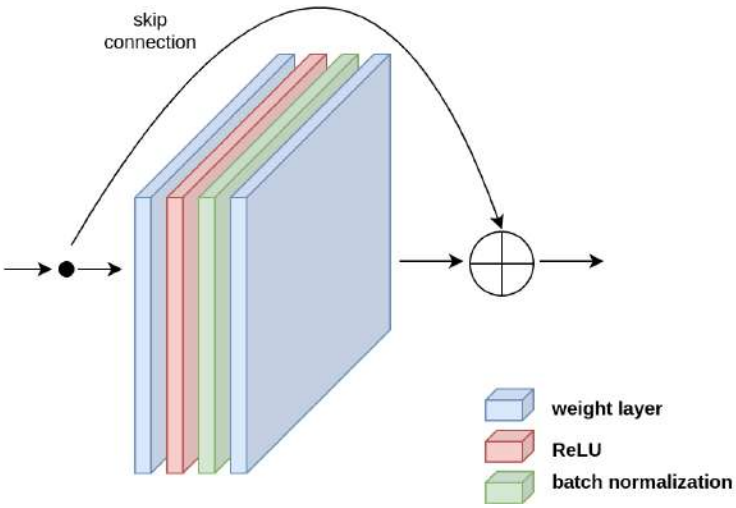


Figure 2.5: Residual block

2.3 Semantic Segmentation

Semantic segmentation is the computer vision task that classifies every pixel of an input image, as opposed to image classification, which classifies the image as a whole. It is highly relevant to this dissertation, as it mainly consists of implementing several semantic segmentation architectures. In this section, we briefly go over the main architectures that have marked the evolution and future of this field.

The first important step in the field was the presentation of Fully Convolutional Networks (FCN) by Long, Shelhamer, and Darrell [13]. They created CNNs whose output was the same size as the input by appending an upsampling layer at the end of the model.

U-Net [14] improved performance significantly with the introduction of the encoder-decoder architecture. In the encoder, the spatial dimensions (width and height) are progressively reduced while the number of features (i.e., the number of channels) increases. The decoder is located immediately afterward and, in it, the image size is increased through several layers. The architecture is shown in Figure 2.6.

An important addition of U-Net is the creation of skip connections. In the different stages of the decoder, the tensor at the corresponding step in the encoder is concatenated.

DeepLab [15] is another notable series of architectures. The first version introduced the atrous or dilated convolution. This refers to a convolution layer, where the kernel pixels are not adjacent, as shown in Figure 2.7. Later, DeepLabV2 extended on the first version by using several atrous convolutions in parallel with different dilation rates. Lastly, DeepLabV3 [16] included several improvements

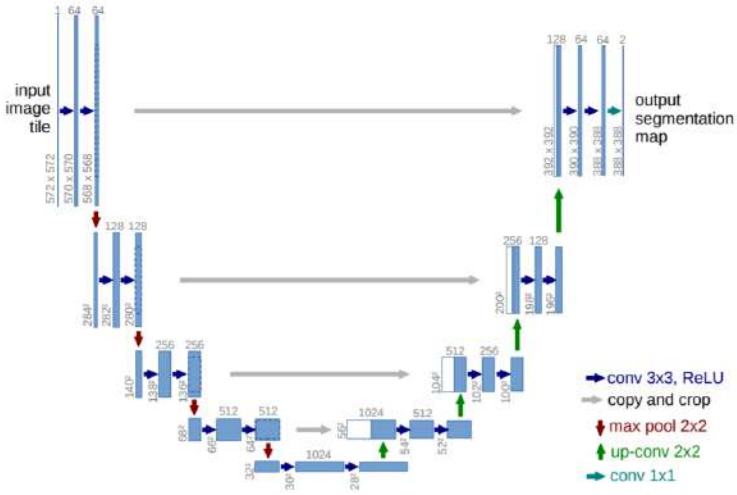


Figure 2.6: U-Net architecture [14]

which made it perform significantly better. It incorporated global context by using global average pooling and 1x1 convolutions. Additionally, it used batch normalization throughout the whole network.

Mask R-CNN [18], an extension of Fast R-CNN [19], proposes a completely new approach for semantic segmentation. It first applies object detection to an image, i.e., it obtains bounding boxes for the desired boxes. Then, it generates a binary mask for each of those objects.

Another influential architecture is the Pyramid Scene Parsing Network (PSPNet) [20], which introduced applying parallel pooling layers of different sizes, allowing the model to obtain more context information.

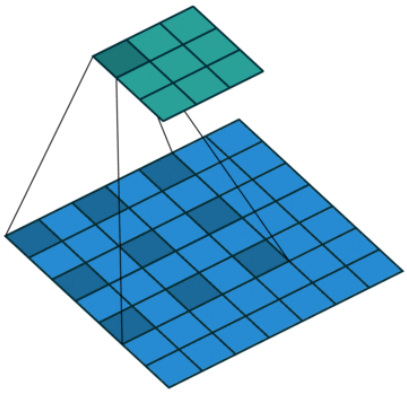


Figure 2.7: Atrous convolution [17]

Chapter 3

Methodology

This chapter explains the methodology we have followed to design, run, and evaluate our experiments. The actual datasets used for training will be explained in chapter 4, while in chapter 5 we will describe the architecture of the deep learning models used in this project.

3.1 Technical considerations

All the work in this dissertation has been performed on Linux, specifically with Ubuntu 18.04. All the code was been written in Python 3.8. The IDE used for the development is PyCharm Community Edition 2022.2.2. To manage the dependencies of our projects, we have used virtual environments. This has been extremely useful, as they allow to create identical environments in two different computers. Without using them, it is common to have errors because of missing libraries or incorrect versions. The packages installed in a virtual environment can be easily exported with the command `pip freeze > requirements.txt`. The newly created file can then be moved to any other computer and execute `pip install -r requirements.txt` to replicate the initial environment.

The implementation of the different models in this dissertation has been done using either TensorFlow [21] or PyTorch [**pytroch**]. These are the two most popular frameworks for programming deep learning models. Both of them are primarily made for Python and are open-source. TensorFlow was developed by Google Brain and the first version came out in 2015. On the other hand, PyTorch was first released in 2016 and has steadily become the preferred alternative for creating complex neural networks.

An important consideration is that both frameworks support the use of graphics cards. Our models will be trained on a desktop computer with an NVIDIA GeForce RTX 3090, which has 24Gb RAM. To be able to use this graphics card,

we will need to install the CUDA library, which includes a set of tools that are necessary to control the card. In particular, we have installed CUDA release 11.0. It is very important to create an environment variable in our project with key `LD_LIBRARY_PATH` and value of the location of the CUDA installation. In our case, CUDA has been installed in the default directory, `/usr/local/cuda`. Once these steps have been completed, we can check that the installation was correct and that we can use the NVIDIA card by executing the following line in Python: `torch.cuda.is_available()`.

The use of a graphics card reduces training and inference times considerably. This is because many of the operations performed in a neural network can be parallelized. The card has thousands of cores which can operate simultaneously, saving time.

3.2 Generating data

An essential part of training a model with images is to define the transformation techniques that take place from the moment the image is read from the disk until it is actually sent through the model for training and prediction. In this section, we will not explain the specific datasets that we have used or how we have created them, as we will talk about this later in chapter 4. Instead, we will talk about the implementation of the system to load and pre-process images.

To solve this problem, we will create a data generator, who will be in charge of reading the dataset from the disk and outputting both the ground truth and label. We will try to build a system with the following characteristics:

- Avoids data redundancy: often, we must transform the original data. These transformations include using stereo images, changing the size, or applying data augmentation techniques. We will try to avoid creating another dataset on the disk for each of these cases. Instead, we will specify a set of settings to the data generator, which will apply the necessary transformations on the go.

- Minimal storage consumption: we are working with datasets containing thousands of images, which can take up considerable space. We will avoid loading images until we actually need them, not when the data generator is instantiated. As models are trained and evaluated in batches, there will never be more than a small set of images that are loaded into memory simultaneously.

- Allow for mapping of label keys: we may want to modify the labeling keys of the dataset. For example, if we no longer want to train a particular class and want to convert it to the background. To solve this problem, the generator

will allow to specify a set of class mapping, which will be applied on the go, i.e., when the label is requested.

- Combination of datasets: at some point, we may want to merge multiple datasets. We must allow for this functionality without creating a new dataset on disk.

The file structure of the original datasets will always be the following:

```
<dataset name>
├── training
│   ├── images
│   │   └── <image name>.jpg
│   └── labels
│       └── <image name>.png
└── val
    ├── images
    └── labels
```

This code will be created in Python. We will define two classes: `DataGenerator` and `ImageHelper`. Two versions of the `DataGenerator` will be created for each of the two frameworks that we will use: PyTorch and TensorFlow. The TensorFlow `DataGenerator` will inherit from the `keras.utils.Sequence` class, while the PyTorch will from `torch.utils.data.Dataset`. The implementation of these classes forces us to implement the `__getitem__(self, idx)` and `__len__(self)` class methods, and will allow for our `DataGenerator` to be seamlessly used to train networks with the corresponding framework.

When instantiating a `DataGenerator`, we can specify all the parameters in Table 3.1. At that point, we will iterate over the specified directories, and a list of elements will be created. The number of elements will be the number of files in the directories multiplied by the number of data augmentation transformations (i.e., flip, saturation, contrast, brightness) we have applied.

The type of the previous elements will be of `ImageHelper`, which will have a `get(self)` method. When called, the image and label will be read from the disk, all transformations will be applied, and a tuple with image and label will be returned in the form of numpy arrays.

To access a batch of images, we will call the `__getitem__(self, idx)` method of the `DataGenerator`. This will load all the images of the batch through their respective `ImageHelpers`, and will apply the class mappings before returning the batch.

The benefit of this data pipeline implementation is that it is easily adaptable to new use cases and possible transformations that we will need. In fact, we did not

Parameter	Type	Description
images_path	path-like	Location of the images
labels_path	path-like	Location of the labels
n_classes	int	Number of classes
batch_size	int	Batch size
output_size	tuple	Output size of the images and labels
to_stereo	bool	Repeat images in the horizontal axis
flip	bool	Flip images along the horizontal axis
saturation	bool	Apply a random saturation transform
contrast	bool	Apply a random contrast transform
brightness	bool	Apply a random brightness transform
class_mappings	dict	Convert labels according to these mappings

Table 3.1: Parameters of the `DataGenerator`

build it at a single point in time, but instead, it has evolved with functionalities as we have needed them.

Lastly, we have created a helper class that allows us to merge different datasets. When instantiated, it receives a list of datasets and combines them. When using this merged dataset, we will treat it as any other dataset, using the same methods.

3.3 Objective evaluation

It is very important to define a set of metrics that measure how well our models are performing and allows us to objectively compare when testing out new hypothesis, or observe how the quality is improving as the number of training epochs increases.

3.3.1 Quality

To measure the quality of our models, we will use the intersection over union (IoU) metric, which is widely used in object detection and semantic segmentation. To compute it, we need two arrays of the same shape: ground truth (i.e., the true label of the image) and the model prediction. The metric is calculated as many times as classes we are predicting. To compute it for a certain class, we will obtain the mask of that class for the ground truth and the prediction by setting the pixels of the desired class to 1 and all others to 0. We will then apply the following formula from Equation 3.1, where $\cdot$ and $+$ are the logical 'and' and 'or' operators respectively.

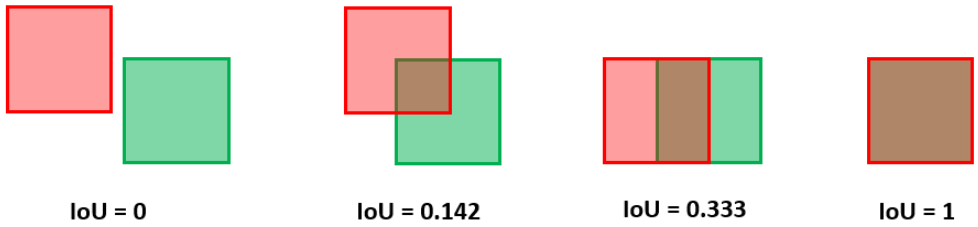


Figure 3.1: Examples of Intersection over Union [22]

$$IoU(label, prediction) = \frac{sum(label \cdot prediction)}{sum(label + prediction)} \tag{3.1}$$

Figure 3.1 gives us some example to help us understand the logic behind IoU. It may take a value from 0 to 1. When there is no overlap between ground truth and prediction, IoU is 0, because the intersection of both shapes is zero pixels. The opposite case is when there is a complete overlap, then the intersection and union are both the same, and IoU is 1. The two examples the middle represent the most common case, which is when there is a partial overlap.

Note that the examples from Figure 3.1 are for object detection, but we will be calculating IoU for semantic segmentation. The only difference is that the label and prediction will not be rectangular, but may take any form.

3.3.2 Inference time

As we are concerned with real-time models, we will also need to compute the inference time. To achieve this, we will predict the whole validation set while keeping track of the duration of each prediction. We will measure the duration using Python's `time.perf_counter()` function. Additionally, we will also obtain the standard deviation, with whom we will be able to create confidence intervals.

To implement this, we will create a class `AverageMetric`, which will have a method `add_value(self, value)`. This class will keep track of the average and standard deviation of the values that are added through that method. To do this, it will have two class attributes (`sum_values` and `sum_squared_values`), which will be updated every time the method is called. An attribute `count` will also be used to keep track of how many values have been added. The final confidence interval can be calculated after applying the formulas from Equation 3.2. It would be $[mean\_value - confidence\_error, mean\_value + confidence\_error]$.

$$\begin{aligned} mean\_value &= sum\_values/count \\ variance &= sum\_squared\_values/count - mean\_value^2 \\ std\_value &= \sqrt{variance} \\ confidence\_error &= std\_value \cdot z_{\alpha/2}/\sqrt{count} \end{aligned} \tag{3.2}$$

An important consideration is that, when using the PyTorch framework with GPU, it is important to call `torch.cuda.synchronize()` before measuring the time. This is because tensor operations are computed asynchronously, which means that they may not necessarily be completed when we are measuring the time. By calling this function, the code will block and the next line will not execute until all tensor operations on the GPU device have finished.

3.4 Subjective evaluation

Finally, we will do a qualitative evaluation of how the models work. We will use a VR headset with a stereo camera for this. Images will be captured in 1280x480, which will be separated in two 640x480 frames, one for each camera. These frames will be forwarded through the model. Then, with the prediction, we will create a mask so that the user may only see his body and certain objects.

The user will then do a subjective evaluation of what he sees considering, among others, the following aspects:

- How good is the quality of the segmentation? The user will observe if there are many false positives or false negatives. He will consider the impact of these two different types of errors.
- How good are the edges? An important part of the segmentation are the borders that separate the positive and negative pixels. The user will observe how precise they are, and whether they are fine or coarse.
- How responsive is the segmentation? The user will analyze whether he feels any lag in the segmentation or if it is fast enough to be close to a real experience.
- How real does the experience feel? Finally, the user will consider the overall performance of the model and the immersiveness of his experience.

This subjective evaluation is highly important because it may sometimes give us additional insights or unexpected results to those obtained from the objective evaluation.

Chapter 4

Datasets

In this chapter, we explain the characteristics of the datasets we used to train our models, and we explore different transformation techniques (i.e., feature engineering) we applied to the images and labels. The performance results will be presented later in chapter 6.

4.1 Egocentric datasets

There are widely known datasets, such as Pascal VOC, COCO, or Cityscapes, that are used by most researchers to train, test, and compare their semantic segmentation algorithms. However, the specific requirement of egocentric segmentation makes them unsuitable to our use case. We need images that are taken from the point of view of a human (i.e., a camera located on a VR headset). Additionally, the images must be labeled so that the egocentric human body is differentiated from other humans.

4.1.1 EgoBodies

The EgoBodies dataset is a subset of the dataset presented by Gonzalez-Sosa et al. [23]. Figure 4.1 shows some examples. It is composed of images from 3 different datasets:

Figure 4.1: Examples of the EgoBodies dataset

- **EgoHuman:** This is a semi-synthetic dataset. It was created recording the human body with a chroma (i.e., a green background) and then setting several realistic backgrounds.
- **THU-READ:** This dataset was presented by Tang et al. [24]. It contains RGB-D egocentric images, which means that, in addition to color information, the images contain a fourth variable, representing depth. However, in our algorithms, we ignore this channel and only use 3-channel RGB images.
- **EgoOffices:** This dataset comprises several recordings in an office setting. It was also created taking IMU and depth information. However, as with the previous one, we will only use the RGB values.

The final EgoBodies dataset is composed of 8005 training images and 1069 validation images, all in jpg format and 640x480 size. These images are accompanied by their corresponding ground truths. These ground truths, also called labels, are in png format and have the same size as the image they refer to. The pixel value is an integer from 0 to 12. In total, there are 13 possible values, representing the class of the object to which that pixel belongs. For example, 0 refers to the background, 1 to a person or any body part, 4 to a keyboard, and 9 to a plate. All the classes are shown in Table 4.1.

Key	Class
0	Background
1	Human body
2	Laptop
3	Screen
4	Keyboard
5	Mouse
6	Notebook / Paper
7	Pen
8	Mobile phone
9	Food
10	Cutlery
11	Drinking cup
12	Other

Table 4.1: Original key of the Egobodies dataset

With the objective of simplifying our task, we transformed the labels, keeping only the human body and 5 objects. The reason for this filtering of objects is to

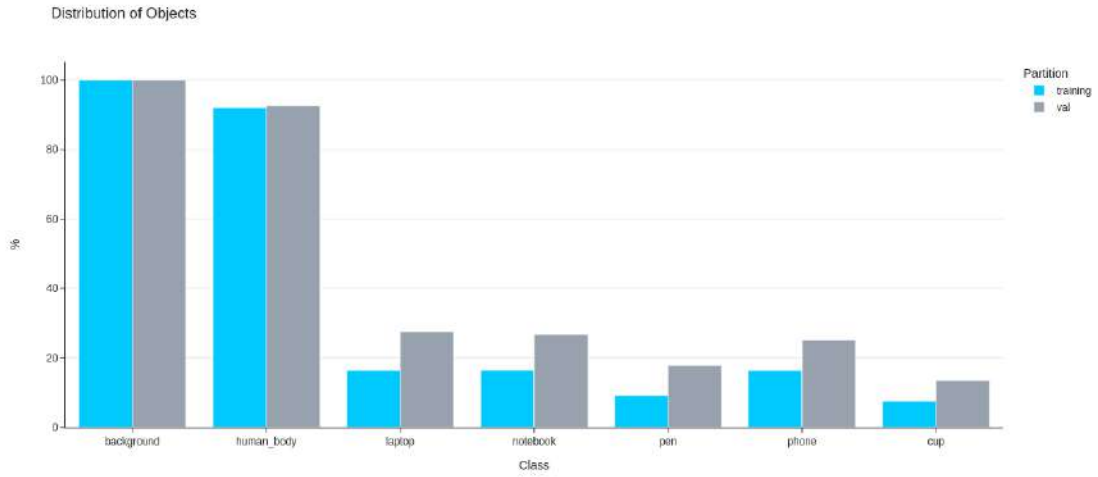


Figure 4.2: Frequencies of the different classes of the EgoBodies dataset

run a minimum viable experiment without adding too much complexity. If we start from scratch trying to segment more than 10 objects, we will probably achieve poor results, and it would be harder to debug what are the causes. All these objects have in common that they can be found in an office setting. The remaining objects were all converted to the background by changing their label value to 0.

The class key of the resulting dataset can be found in Table 4.2. Figure 4.2 shows the count of the different objects in the EgoBodies dataset. We observe that the human body is a lot more represented than the five objects.

Key	Class
0	Background
1	Human
2	Laptop
3	Notebook
4	Pen
5	Phone
6	Cup

Table 4.2: Class key of the 5-office-objects EgoBodies dataset

Another variation of the dataset was created instead, keeping only two office objects: the laptop and the phone, which received the keys 2 and 3, respectively.

Figure 4.3: Examples of the EgoHOS dataset

4.1.2 EgoHOS

This dataset was presented by Zhang et al. [25]. It contains 8993 training images and 1124 validation images of different sizes. Figure 4.3 shows some examples. The items contain scenes from incredibly diverse backgrounds, including cooking, gardening, or working with a laptop.

The images have been obtained from multiple sources, including Ego4D, EPIC-KITCHEN, THU-READ, and videos collected by the authors. They have taken the videos obtained from the previous datasets and sampled a frame every 3 seconds. They have filtered the frames to make sure they contain an interacting between a hand and an object. To obtain the ground truths, they have used human labeling. This information is found in the supplemental materials to the paper [25].

The ground truth format of this dataset is different in many ways from the one of EgoBodies. Instead of segmenting the whole body, EgoHOS only labels the arms (hands included). Moreover, only interacting objects are segmented, i.e., objects that are touching the hands (1st order interacting order), or objects that are touching an object touching the hands (2nd order interacting object). EgoHOS also assigns different class keys to left and right hands, and objects touching the left hand, the right hand, or both. The full original class key of this dataset is shown in Table 4.3.

Key	Class
0	Background
1	Left hand
2	Right hand
3	1st order interacting object by left hand
4	1st order interacting object by right hand
5	1st order interacting object by both hands
6	2nd order interacting object by left hand
7	2nd order interacting object by right hand
8	2nd order interacting object by both hands

Table 4.3: Original class key of the EgoHOS dataset [26]

Figure 4.4: Labeled examples of EgoHOS after processing

The dataset was processed, applying several transformations. Firstly, images were transformed to 640x480 size. This was done by center cropping the image to the corresponding aspect ratio. Then, the image was resized to 640x480.

The labels were also resized in the same way as the images, and the class keys were modified for simplicity, as shown in Table 4.4. Both hands were combined in a single class (1). All first-order interacting objects were combined in class 2. Second-order interacting objects were converted to the background. Figure 4.4 shows some labeled images. Black represents the background, orange depicts the human arms, and blue is used for the interacting objects.

Key	Class
0	Background
1	Hands
2	Interacting object

Table 4.4: Class key of the EgoHOS dataset after processing

In Figure 4.5, we can observe the distribution of the three classes of the processed EgoHOS dataset in both the training and validation subsets. We observe that all images contain the human class, while most contain an interacting object too.

4.2 Exocentric datasets

While most of our work will be done with the egocentric datasets, we will also attempt to segment exocentric humans. For this, we will assign two different classes for the human body, depending on whether it is seen from a first-person or third-person point of view.

To achieve this goal, we will need to mix several datasets containing both egocentric and exocentric humans. We already have egocentric datasets. The positive thing is that exocentric human images are substantially easier to find. In fact, many common datasets have a class defined for people. In this section, we have explored two renowned and publicly-available datasets and obtained a subset of them with images of people.

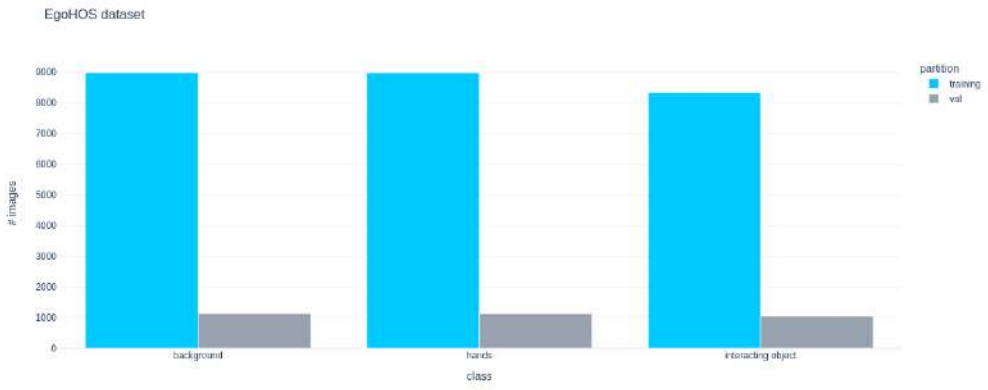


Figure 4.5: Distribution of the classes of the processed EgoHOS dataset

4.2.1 COCO

The COCO (Common Objects in Context) [27] dataset was presented in 2014 with the objective of helping advance the state-of-the-art in the tasks of object detection and segmentation. It contains 328k images and labels 91 object types. To simplify the process of downloading and exploring this dataset, we have used `fiftyone`, a Python module that provides a tool called Zoo for using several famous datasets out of the box. We have used the COCO-2017 subset. All the labels are found together in a labels.json file. However, it is easy to process them using the `pycocotools` module.

We have manually selected 1010 images and separated them between training and validation with a 90:10 ratio. We have mainly selected images in which there were people, and they were clearly visible. We have avoided images where the person was too small (i.e., it was very far from the camera) or where there was a huge number of people (e.g., a stadium). The selected images have been resized to 640x480 to have the same format as those from the egocentric datasets. Additionally, the annotations (i.e., the labels) have been transformed to remove all classes except the person class, which took the value. Some examples of the selected images are shown in Figure 4.6a.

4.2.2 ADE20K

The ADE20K dataset was created by researchers from MIT CSAIL [28]. It contains 25574 training images and 2000 validation images, all fully annotated with 150 classes. We have filtered the dataset to only keep images in which the class human was present. The resulting subset contains 2340 images for training and 250 images for validation. We have applied the same transformations as with the COCO

(a) COCO (b) ADE20K

Figure 4.6: Labeled examples of the selected images from both exocentric datasets after transformations

dataset: conversion to 640x480 size and keeping only the person class. Some examples are shown in Figure 4.6b.

Chapter 5

Architectures

5.1 Thundernet

The Thundernet architecture was developed by Gonzalez-Sosa et al. [23] inspired by the namesake object detection model [4]. Figure 5.1 shows a schema of the architecture. The model can be divided into the following parts:

- **Encoder blocks:** In the encoder, the image size is reduced through a set of three blocks, throughout which the network learns semantic information while reducing spatial information. Each block consists of two equal parts, which we will call residual blocks. Each residual block has two parts, which are formed by a convolutional layer, a batch normalization, and a ReLU activation layer. The main characteristic of the residual block is that the output (just before the ReLU) is added to the input of the block, and then that goes through the activation function. This is called a skip connection.

- **Pyramid pooling module:** In the middle of the model, we find the pyramid pooling module (PPM). This layer applies a set of average pooling operations with four different sizes. All the outputs go through a convolutional layer and are then concatenated. The objective of this is that the model must be able to understand images of many different sizes.

- **Decoder blocks:** Finally, the decoder blocks reverse the encoding process. This is achieved by using convolutional transpose layers, which are the exact opposite of a convolutional layer. A very important part of this section of the model is to use skip connections.

Thundernet has a total of 5,648,452 parameters.

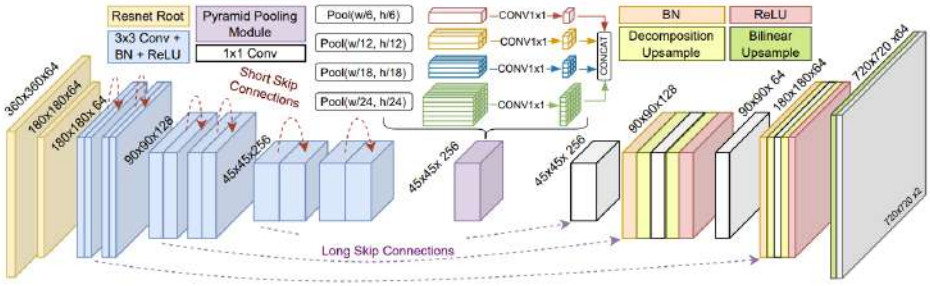


Figure 5.1: Architecture of the Thundernet model [23]

Parameter	Value
optimizer	adam
learning rate	0.0001
batch size	4
kernel regularizer	0
epochs	20
n_classes	depends on the dataset
stereo	True
data augmentation	None
size	640x480

Table 5.1: Default training configuration for Thundernet

5.1.1 Training environment

The Thundernet model was developed in TensorFlow. The default configuration used for training is shown in Table 5.1. Throughout the experiments, we have changed some of these values, searching for performance improvements.

The data generators are easily created with our `DataGenerator`, explained in section 3.2. During training, we save a checkpoint after each epoch along with the following metrics: IoU and loss for both training and validation sets. These are useful, as we can check whether the loss is decreasing to see if the network is learning.

5.2 PIDNet

Another architecture we have implemented is PIDNet [29]. We decided to try it because of the great results it had achieved on other famous datasets (cityscapes

and camvid), both in terms of accuracy and inference time.

The architecture of the model is shown in Figure 5.2. The model is inspired by a PID controller, which has three components: a proportional, an integral, and a derivative one. Similarly, PIDNet has three branches with different purposes:

- P (Detail): this branch learns detailed information, which refers to pixel-level information. The layers keep the same size. A pixel in the output tensor is not influenced by a pixel in the input tensor that is located far from it.
- I (Context): as opposed to the previous branch, this one learns context information, which means that it learns patterns about a big part of the image. This is achieved by reducing in size the image by setting the stripe parameter in the convolution layer to more than 1. A pixel in the output tensor is a function of a big area of the input tensor.
- D (Mix): this branch is used to decide how the output of the previous two branches will be merged. The output of this branch is passed through a sigmoid function, taking values in the $[0, 1]$ interval, where 0 would mean only keep detail information and 1 only context information.

As there are not linear layers, the input image can be of any shape. The sides of the output will have one eighth of the input's shape. For instance, if the input is 640x480, the output will be 80x60.

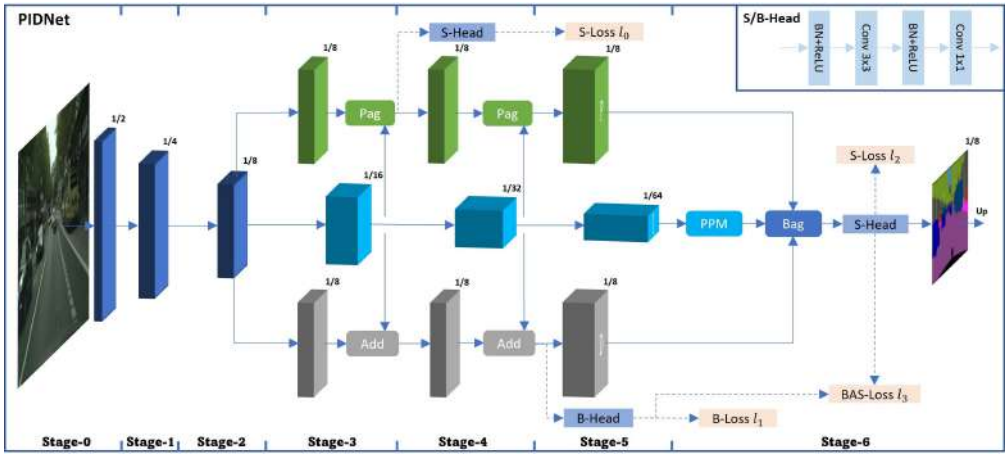


Figure 5.2: Architecture of the PIDNet model [29]

The PIDNet model is offered in three different sizes: S, M, and L. The number of parameters is shown in Table 5.2. It is surprising that even the small PIDNet model has more parameters than Thundernet. This increase in the number of

parameters of PIDNet with respect to Thundernet, without an increase in inference time, can be explained by the three-branch architecture of PIDNet. By having three branches, there is a high level of parallelization, which does not incrase inference time with a powerful enough GPU.

Model	# parameters
Thundernet	5 648 452
PIDNet S	7 623 651
PIDNet M	28 536 451
PIDNet L	36 935 459

Table 5.2: Number of parameters by model

Figure 5.3 [29] shows a comparison between accuracy and inference speed for several real-time semantic segmentation models. As expected, there is an inverse relationship between those variables for the PIDNet models. Normally, the greater the number of parameters, the higher the accuracy and the lower the inference speed. The reason why this does not always hold is that a model could have more tasks that can be run in parallel.

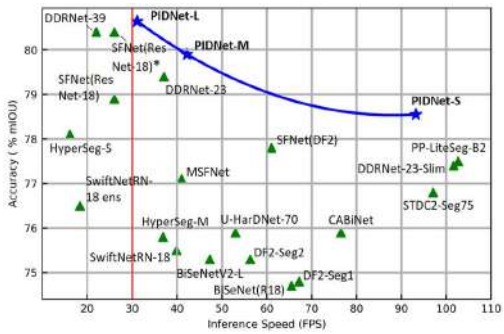


Figure 5.3: Comparison of accuracy and inference speed [29]

5.2.1 Training environment

The PIDNet model has been implemented in PyTorch. The training configuration is shown in Table 5.3. As we see, the maximum number of epochs is huge. We have implemented a checkpoint function that allows us to restart training from the last epoch if it stops for any reason. Additionally, we will often not train the model for the full 500 and will stop earlier if the model seems to have converged.

Parameter	Value
optimizer	sgd
learning rate	0.01
batch size	4
max epochs	500
n_classes	depends on the dataset
stereo	False
data augmentation	None
size	640x480

Table 5.3: Default training configuration for PIDNet

5.3 YOLO

The YOLO (You Look Only Once) was presented in 2016 by Redmon et al. [30] and was a breakthrough for real-time object detection. The architecture is shown in Figure 5.4, consisting of 24 convolutional layers and two fully-connected layers. Before YOLO, object detection was performed through classification: you cropped a specific part of an image, run it through a classifier, and with the score obtained you can decide on a final output.

The contribution of YOLO is that they take a new approach: treating the object detection problem as a regression problem where the aim is to predict the coordinates and the class probabilities. One of the best characteristics about YOLO is that it is incredibly fast and efficient, which makes it suitable for an application such as ours, where we need real-time prediction.

Throughout the years, several versions have been presented [31] [32], which have improved on the previous performances. A company called ultralytics has developed a set of open-source respositories that make the use of YOLO a very easy task. We have used their YOLOv8 model, which includes a model that was adapted for semantic segmentation.

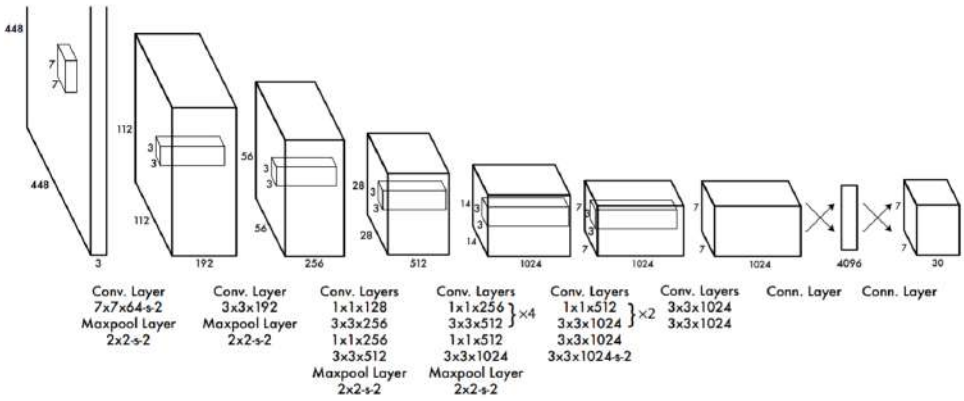


Figure 5.4: Part of the YOLO architecuture [30]

Chapter 6

Experiments

6.1 Training with EgoBodies (5 objects)

All experiments in this section have been performed by training the Thundernet model with the EgoBodies dataset.

6.1.1 Initial experiment

The first experiment we have carried out involves just training the baseline model (i.e., the model that only segments the human body) but adding new classes for objects. We no longer have a binary classification problem (segmenting human body and background) but a multi-classification problem. Therefore, we only need to change the loss function to categorical cross-entropy. This model was trained with all the 5 objects from the EgoBodies dataset: laptop, phone, cup, book, and pen.

We will train two variations of the experiment, depending on whether the objects are grouped or not in a single class. To implement this functionality, we only need to change the class mappings of the data generator.

- **Separated objects:** every object has its individual class. As we have five objects, they are represented by classes 2 to 6. Classes 0 and 1 are the background and the human body, respectively. We do not need to specify any class mappings as we are using the dataset as is.

- **Objects as one:** we merge the masks of all objects into a single class, which will be class 2. To make this change, we can specify the following class mappings: {`1: 1, 2: 2, 3: 2, 4: 2, 5: 2, 6: 2`}. It is important to set the mapping of the human class because all classes not specified will be converted to the background (class 0).

model	background	human	laptop	notebook	pen	phone	cup	objects
baseline	92.74	60.72						
separated objects	82.34	54.84	28.96	11.98	2.19	24.71	8.34	28.91
objects as one	80.22	46.72	-	-	-	-	-	32.17

Table 6.1: IoU (in %) of the initial Thundernet experiments with objects

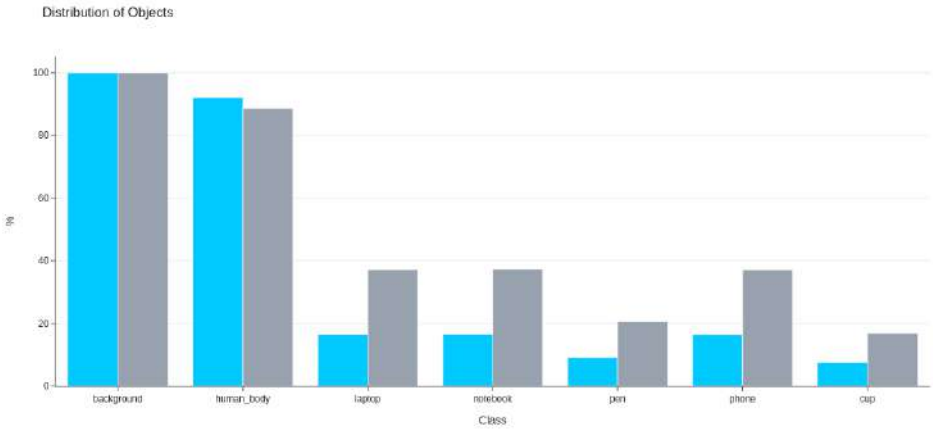


Figure 6.1: Histogram of objects before (blue) and after (gray) downsampling

Table 6.1 shows the results of the experiment. In the table, we also show the baseline model for comparison. This is the model that already and only segmented the human body. Firstly, we see that the object IoU is around 30% in both cases, which is quite low. The human IoU has decreased below what we had obtained in the baseline model. Lastly, the background IoU has also decreased substantially. Note that in the objects-as-one scenario, the model achieves a better objects IoU at the cost of human IoU.

6.1.2 Downsampled dataset

One of the possible causes of the low performance of object segmentation is the fact that objects are underrepresented in the dataset in comparison with the human class. A way to address this is to increase the percentage of images where objects appear by removing images where only the human class is present. After applying this filter to the dataset, we can observe the new distribution in Figure 6.1.

We have trained the Thundernet model with the new downsampled dataset, and the results can be observed in Table 6.2. The quality of segmentation is poor and worse than the one obtained in the previous experiment.

If we analyze the object IoU, it is noticeable that it is greater for the bigger and

model	background	human	laptop	notebook	pen	phone	cup	objects
not downsampled	82.34	54.84	28.96	11.98	2.19	24.71	8.34	28.91
downsampled	78.67	42.78	37.49	9.78	1.22	16.39	11.51	24.04

Table 6.2: IoU (in %) of the initial and downsampled experiments

model	background	human	laptop	notebook	pen	phone	cup	objects
default lr	78.67	42.78	37.49	9.78	1.22	16.39	11.51	24.04
fast lr	76.83	33.79	26.73	11.17	0.04	17.75	7.16	23.51

Table 6.3: IoU (in %) of the experiments with fast and default learning rate

more represented objects (phone and laptop) than for the smaller ones (pen, cup, and notebook). For the former, the model achieves an IoU of around 20% and 30%, respectively, while the other three objects obtain significantly lower results. It is surprising that the pen, which is the smallest object, does not even reach 3% IoU.

6.1.3 Increasing learning rate

In our quest to try to improve the model performance, we have tried to increase the learning rate. Normally, we used 1e-4 but in this experiment, we will use 1e-3. The results are shown in Figure A.4. The change in the learning rate has not improved the quality of the results. In fact, the IoU for most classes has decreased.

6.1.4 Kernel regularizer

The kernel regularizer applies a penalty to the weights of the convolutional layers. The penalty used in Thundernet is the l2 loss function. We have run two experiments with (1) and without (2) the kernel regularizer. When we used it, the weight applied to the loss is 2e-4. In Figure 6.2, we can observe the results, and we see that there is no significant difference between both scenarios.

6.1.5 Increasing size

Another logical hypothesis to think about is that the size of the input images is not big enough for the model to be able to learn the characteristics of the objects. This is because, in the first part of the model (the encoder), the images go through a set of layers that reduce the image size. In this part, semantic information is learned, but spatial information is lost. If, after the encoder, the object is too small and only covers a small part of the vector, it follows that it will be harder for the

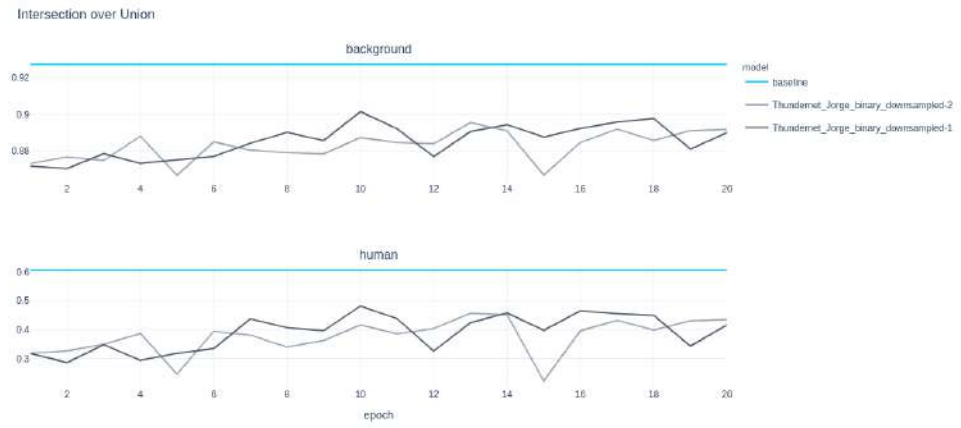


Figure 6.2: Validation IoU of the model with and without kernel regularizer

(a) Image used previously (640x480)

(b) Image used in this experiment (1280x480)

Figure 6.3: Images before and after increasing the size

model to learn to distinguish it.

One way to test this hypothesis is to increase the size of the model. Our initial size was 640x480, while this test was run in 1280x640. The original images from our dataset are also 640x480. What this means is that, when converting the image to stereo, we were squeezing the images on the horizontal axis while applying no size reduction in the height. In this test, by increasing only the horizontal axis, we are also testing whether it is detrimental to model quality to apply unsymmetrical size transformations in each axis. In Figure 6.3, we can see an example of an image from the dataset with the smaller and larger size.

The results of the experiment are shown in Table 6.4. We observe that there is not a significant difference between both models.

One limitation of this approach is that when the input image size increases, so does the inference time. This is because the number of operations in a forward run

size	objects	background	human	laptop	notebook	pen	phone	cup	objects
640x480	separated	82.34	54.84	28.96	11.98	2.19	24.71	8.34	28.91
640x480	as-one	80.22	46.72	-	-	-	-	-	32.17
1280x480	separated	79.76	48.02	33.20	9.56	1.53	19.94	11.04	25.99
1280x480	as-one	78.55	48.86	-	-	-	-	-	25.14

Table 6.4: IoU (in %) of the initial and increased-size models

model	background	human	laptop	notebook	pen	phone	cup	objects
resnet not trained	79.76	48.02	33.20	9.56	1.53	19.94	11.04	25.99
resnet trained	77.98	46.57	33.24	12.03	0.45	16.57	6.11	24.10

Table 6.5: IoU (in %) of the models with and without training the resnet layer

increases. Note that the number of parameters of the model will not necessarily increase. The reason is that the number of parameters of a convolutional layer depends on the kernel size and input and output channels, but it does not depend on the image size.

As a result, if we had decided to implement this solution, we should also have run a performance test, so that the model is still compatible with the real-time requirements we initially set.

6.1.6 Training the resnet layer

Another experiment we run is training the resnet layer. This layer forms the initial part of the model and its function is to encode the image. The weights of the resnet blocks are initially loaded from a resnet18 model that was previously trained. In previous experiments, we assumed that this resnet encoder is already optimized to obtain the best possible representations of the images it encodes. Therefore, in those layers, we set the parameter `trainable=False` so that the weights were not modified during training. However, in this experiment, we test the hypothesis that the model performance can be improved by training those layers. The results are shown in Table 6.5. Unfortunately, training the resnet layer does not bring a significant improvement to the segmentation quality.

6.1.7 Data augmentation

Another possibility that would explain the low quality of our model is that the training set is not big enough. This is especially true for the objects, as they appear way less frequently than humans. In this experiment, we have tried to apply data augmentation techniques to increase the presence of objects.

transformation	background	human	laptop	notebook	pen	phone	cup	objects
none	82.92	60.50	41.18	16.90	2.45	31.61	15.02	37.20
flip	82.69	49.58	49.81	16.11	2.00	22.93	13.99	36.12
saturation	82.36	54.38	54.52	16.10	2.47	29.72	19.37	40.11
contrast	83.40	54.51	51.56	13.24	2.21	27.95	11.90	38.59
brightness	82.75	55.51	49.10	13.77	2.98	30.20	15.39	37.83
all transformation	81.37	52.81	48.59	16.32	3.88	32.21	14.20	39.65

Table 6.6: IoU (in %) for every class and transformation applied

To apply the transformations, we have used the functions built in the `tensorflow.image` module. We have implemented the following:

- **Flip:** flipping the image along the width dimension.
- **Saturation:** adjusts the saturation multiplying it by a random factor taken from the $U[0.5, 1.5)$ distribution.
- **Contrast:** adjusts the contrast multiplying it by a random factor taken from the $U[0.5, 1.5)$ distribution.
- **Brightness:** adjusts the brightness by a random factor taken from the $U[-0.3, 0.3)$ distribution.

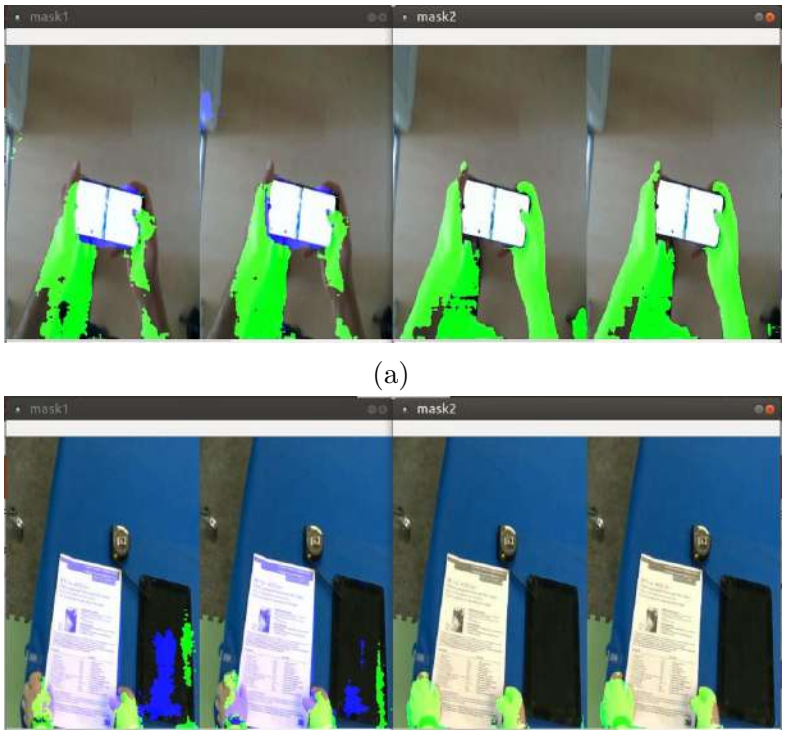
We have run five experiments. In the first four, we have run a single transformation type in each experiment. In the last one, we have applied all transformations. When applying a transformation, the training set is expanded with the transformed images, but the original images are not removed. Therefore, when applying a certain number of transformation, the training set size is multiplied by that number. Note that transformations are never applied to the validation set.

In Table 6.6 we can observe the results of each experiment in addition to the case where no transformations were applied. The results are very disappointing: human IoU decreases substantially in all cases. Regarding the objects, the only one where we see an improvement is the laptop, where there is a clear increase in the IoU. For all the other objects, the changes are not significant.

6.1.8 Conclusions

We obtain several conclusions from the previous experiments:

- Achieving good results in object segmentation quality with the Thundernet model will come at the cost of quality of the human class. In the images from Figure 6.4 we can observe the difference qualitatively. The images from



(a)

(b)

Figure 6.4: Examples of model with (left) and without (right) object segmentation

the left segment both the human and objects classes, while in the right only the human class is segmented. We observe that the human segmentation is clearly better in the second images. Additionally, object segmentation quality is very poor.

- There is no benefit in segmenting the objects as-one. Therfore, when possible, we should segment the objects separately. We will always be able to merge them later if we want.

6.2 Changes to the Thundernet architecture

6.2.1 Modifying the PPM

In this new set of experiments, we have applied a change to the Thundernet architecture, specifically to the pyramid pooling module. As explained previously, this module applies a set of pooling layers with different sizes to the encoded image. After these pooling layers, the output goes through a convolutional layer,

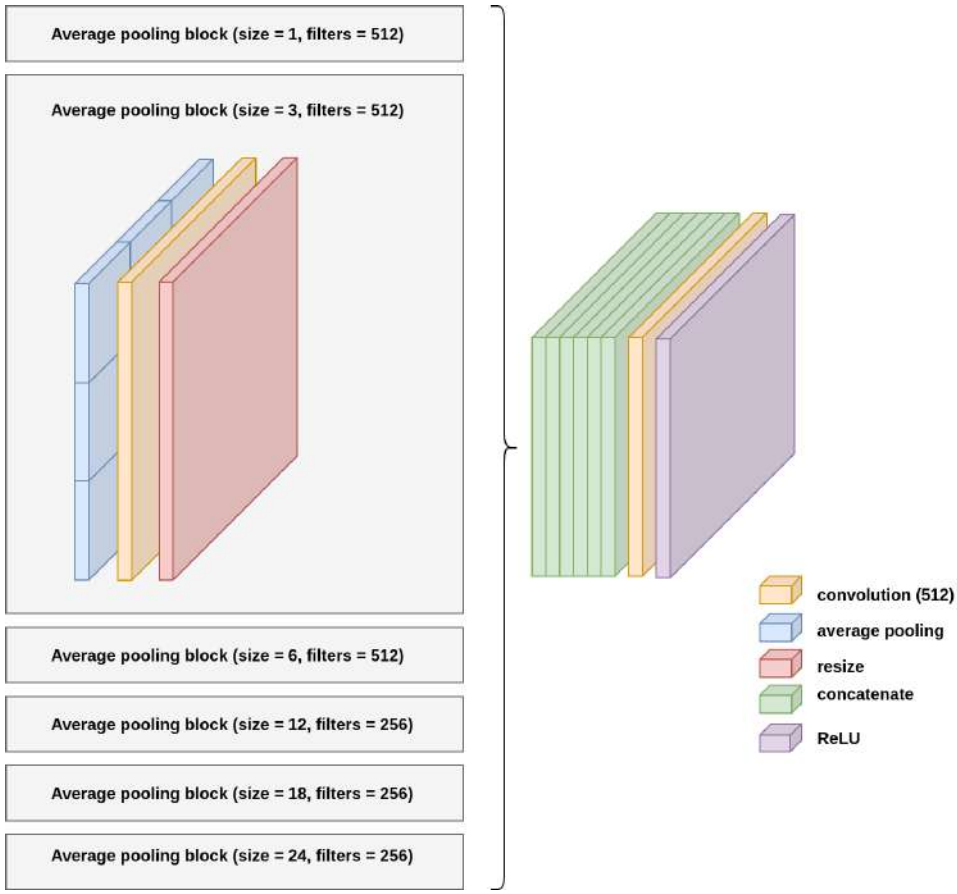


Figure 6.5: Architecture of the new PPM (including the new pooling sizes)

it is resized, and then concatenated. Lastly, the output of the concatenation goes through a convolution and an activation. This architecture is shown in Figure 6.5.

In the experiments from the previous section, the PPM had four pooling blocks with sizes 6, 12, 18, and 24. In the first two, the convolution had 512 filters while, in the later, it had 256. It is a good idea to use a higher number of filters for the output of smaller sizes because, as the size will be smaller, applying more filters will come at a lesser computational cost during inference time.

The hypothesis that we will test in this section is that adding more average pooling blocks will improve the quality of the segmentation. To do this, we have added two new blocks corresponding to sizes 1 and 3. The final architecture after these modifications is the one shown in Figure 6.5.

Additionally, we also made a change to the dataset. In the previous section, we used the EgoBodies dataset with the five selected objects. Throughout these experiments, however, we will use only two: the laptop and the phone. The reason

model	background	human	laptop	phone	objects
binary	94.83	75.27	-	-	-
separated objects	92.69	74.37	59.75	44.84	65.08
objects as one	92.15	73.13	-	-	55.30
human + objects as one	92.34	-	-	-	76.76

Table 6.7: IoU (in %) of the new PPM model

is that we observed that segmentation quality was higher for the laptop and phone objects, probably because of their size and that they are found in a big enough number of images.

The results of the experiments we launched are shown in Table 6.7. We observe that the human IoU is very good, above 70% in all cases. The object segmentation has also improved, with an IoU of 59.75% and 44.84% for the laptop and phone, respectively. This is a significant improvement from the results obtained in the previous section. It is highly surprising that the objects IoU in the as-one model is lower than when the objects were trained in separate classes. This was probably related to the randomness of the training.

Lastly, we have trained a model in which the human and the objects are the same class. In this case, the IoU obtained in that class is 76.76%, which is above the maximum IoU we had for both humans and objects. The model is able to segment better when it only has to predict a single class. However, this has the limitation that, in a real-life use case, we would not be able to create different masks depending on which objects (if any) we want to segment.

6.2.2 Adding an attention block

In this section, we try a new architectural change: adding an attention gate. Attention U-Net [33] is a new layer, which gives the ability for the model to focus on features that may come in different shapes and sizes. The authors have applied this technique to the segmentation of medical images, and they have achieved an improvement in prediction accuracy with a minimal increase in computational overhead. This is in line with our objective to increase quality without making inference time greater.

Our implementation of the attention block is shown in Figure 6.6. The attention block is applied in the last part of the model, the decoder, in parallel with the decoder blocks. It receives two inputs, one of which has double the size of the other. The smaller input (g) comes from the output of the last decoder layer, while the other input (x) is a skip connection from the encoder section of the model.

The attention layers must be used in the decoder part of the architecture. We

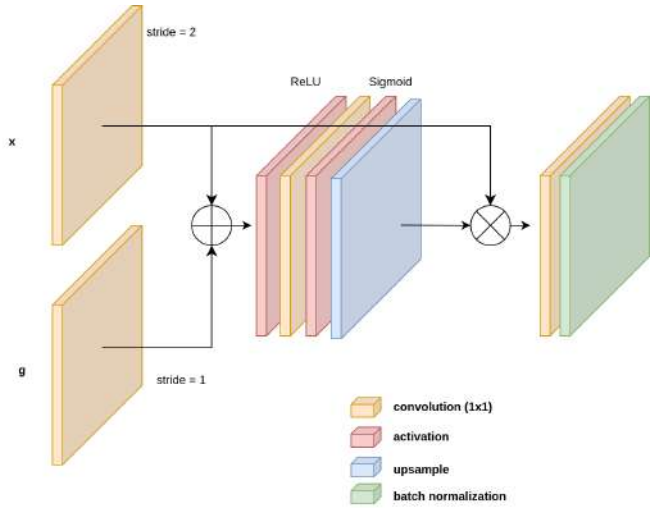


Figure 6.6: Implementation of the Attention Gate

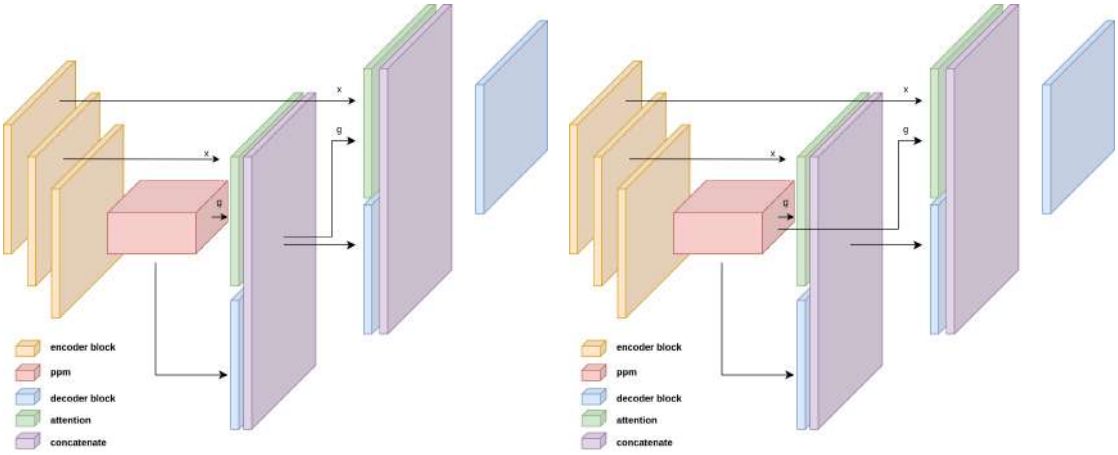
model	background	human	laptop	phone	objects
original	92.69	74.37	59.75	44.84	65.08
attention-v1	91.77	71.10	64.39	41.30	67.49
attention-v2	91.86	72.88	69.03	45.83	70.12

Table 6.8: IoU (in %) of the experiments run with the attention layer (separated objects)

have integrated them into our Thundernet model in two different ways, as shown in Figure 6.7. The distinction between both implementations is where the gate input to the attention layer comes from. In the first one, it comes from the output of the previous layer, while in the second, it always comes from the output of the PPM block.

The results are shown in Table 6.8 for the models trained with separated objects and Table 6.9 for the objects as one. We see that, in both cases, the attention models perform better than the original model when segmenting objects. On the other hand, we see that human segmentation has also decreased in all cases, with the exception of attention-v2 with objects as one, where both human IoU and objects IoU are better than the original model. Regardless of this, we see that the increase in the objects IoU is always more than the decrease in IoU.

Additionally, it is not clear which attention implementation is better, as the first implementation performs better for the separated objects scenario, while the second is superior when objects are the same class.



(a) Input g comes from the previous block (b) Input g comes from PPM

Figure 6.7: Applications of the attention gate to the Thundernet architecture

model	background	human	objects
original	92.15	73.13	55.30
attention-v1	92.10	72.55	69.26
attention-v2	92.23	73.35	61.22

Table 6.9: IoU (in %) of the experiments run with the attention layer (objects as one)

model	background	human	objects
original	92.65	71.91	18.15
attention-v1	92.73	73.10	23.51
attention-v2	92.84	74.67	23.43

Table 6.10: IoU (in %) of the experiments run with Thundernet and EgoHOS

6.3 Training Thundernet with EgoHOS

In this section, we have run an experiment training the Thundernet model with the EgoHOS dataset. We have trained three variations: the original one and both implementations of the Attention U-Net layer. As a reminder, the EgoHOS does not segment the human body, but only the arms, and it segments the objects the human is interacting with (i.e., touching).

The results of the experiment are shown in Table 6.10. We observe a high IoU in the human class, above 70% for all models, but the objects IoU is very poor. One interesting thing is that this experiment also confirms the positive impact of the attention layer, as in both cases, it is able to improve the IoU of the objects with respect to the original model. Surprisingly, this does not come at the cost of human IoU, as it is still greater than that of the original model.

6.4 Training PIDNet with EgoHOS

We have trained the PIDNet model with the EgoHOS dataset. As we know, the input images to the PIDNet model do not have to be of any particular shape. Therefore, in order to decide what the best training size is, we have run the following experiments:

- **Mono 1280x960:** the images have been upsampled to double their size along both axes.
- **Mono 640x480:** the images are taken as they are from the dataset.
- **Stereo 1280x960:** the images have been converted to stereo and then upsampled to the specified size. Because of the stereo conversion, the images are actually in their original size along the horizontal axis while they have stretched to double their height.

Note that the original images are always of size 640x480. Before training, the images were resized to the input size specified above. Therefore, we are increasing the size of the image but the amount of information is the same.

model	prediction size	background	human	objects
mono 1280x960	1280x960	95.67	90.95	54.35
mono 640x480	1280x960	92.47	83.35	39.15
mono 640x480	640x480	91.69	77.94	32.98
stereo 1280x960	1280x960	95.13	87.44	52.13

Table 6.11: IoU (in %) of the experiments run with PIDNet and EgoHOS

As seen in Table 6.11, the training size of the images must not necessarily be the same as the prediction size of them. The reason is the same as stated above: as this model does not have fully-connected layers, it can take a three-dimensional (height, width, channel) input of any shape. Therefore, we can decide to train the model with a shape and then predict with another. We could even train the model with images of different shapes.

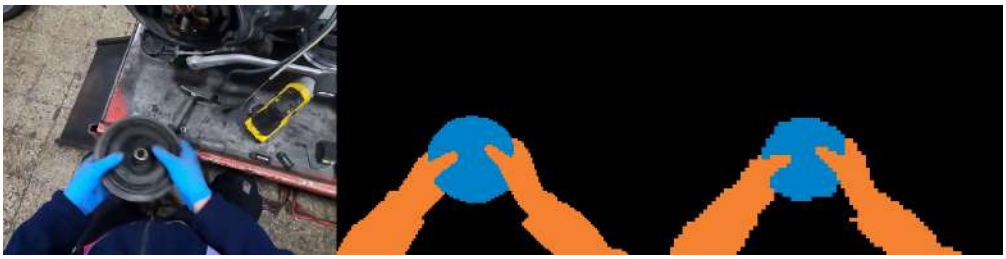
The results of the training are shown in Table 6.11. We see that they are very positive, obtaining the best results in the Mono 1280x960 model: above 90% in human IoU and above 50% in objects IoU. In particular, we can extract the following conclusions from the results:

- It is better to train the network with larger images. Even if the original size of the image is small, applying a simple interpolation technique to resize it to a larger size may bring an improvement in model performance.

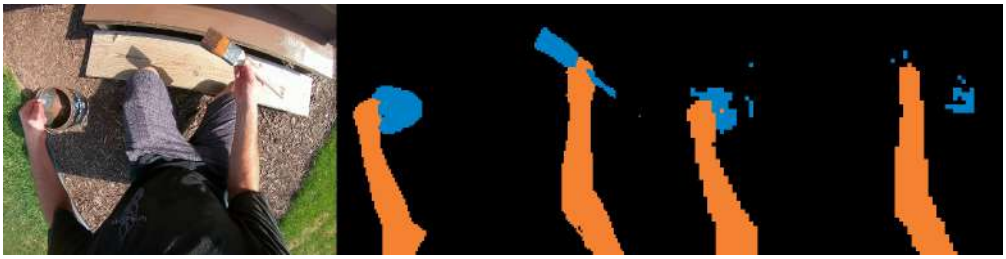
- The impact of training the network with stereo images is very small. We observe that, although there is a reduction in IoU from the Mono 1280x960 to the Stereo 1280x960, it is very slight. If it servers our purpose to train the network with stereo images, we can do so knowing there will not be a significant drop in performance.

- Even if the model has been trained with images of smaller size, we can achieve better results by upsampling images before inference. This is shown in the comparison of both training scenarios of the Mono 640x480 model.

One important thing to consider with regard to the above conclusions is that the PIDNet model output shape is one eighth of the model input shape. This could be the reason why we achieve the best results when upsampling the images before training or predicting. As the output shape is one-eighth of the input, this means that, in the output mask, every group of adjacent 8x8 pixels will be assigned the same predicted class. By upsampling the images before running them through the model, we obtain a more fine and higher-resolution output.



(a)



(b)

Figure 6.8: Example of how PIDNet segments with the models trained at 1280x960 (left) and 640x480 (right)

In the images displayed in Figure 6.8, we can qualitatively assess the quality of the PIDNet segmentation and compare the different training sizes. As confirmed by the metrics from Table 6.11, the model which was trained at 1280x960 performs significantly better. This is especially so in cases where the objects are small, such as Figure 6.8b. As the images go through a series of encoder layers which extract semantic information, if the object is too small, the model will not be able to identify its unique characteristics correctly.

Another thing that we observe from the images is that we can clearly observe the pixels in both images. The reason is that the PIDNet model output is one-eighth of the input. Having so coarse edges clearly degrades the user experience. In a later section, we will explore the different possibilities for resizing images.

6.4.1 Inference time

As it is crucial for us to run our models in real time, we have analyzed the inference time of the PIDNet-L network for different input sizes. To run this experiment, we have processed the entire validation set of the EgoHOS dataset and computed the average duration the model took to make a prediction. We have done this in two different computers, one with an NVIDIA GeForce GTX 1060, which has 6Gb RAM, and the other with an NVIDIA GeForce RTX 3090, which has a 24Gb RAM.

It is important to note that, in this experiment, we are only measuring the

Input size	GTX 1060	RTX 3090
160x120	14 ± 1	11 ± 1
320x240	21 ± 1	11 ± 0
640x480	58 ± 3	11 ± 0
1280x960	180 ± 7	12 ± 0
1600x1200	271 ± 6	32 ± 0
1920x1440	384 ± 2	45 ± 0
2240x1680	520 ± 20	58 ± 1
2560x1920	657 ± 20	72 ± 1

Table 6.12: PIDNet-L inference time (rounded to nearest ms) for several sizes and GPUs (95% confidence interval)

time taken by the model to make a prediction. To obtain the actual number of FPS our application will be able to run on, we would need to take into account the computational overhead of other actions that need to be performed, such as the time taken to send and receive the image from the inference server, or possible transformations like resizing that need to be done before or after predicting.

The results are shown in Table 6.12. Firstly, we notice a clear and expected difference between the inference times taken on each GPU, as the RTX 3090 is a far more advanced model. Another insight is that there is a minimum amount of time it takes to make an inference which will not be reduced even if the input size is reduced. This is inferred from the fact that the first three rows of the inference times of the RTX 3090 take the same value.

We have decided to test the hypothesis that the inference time will increase linearly with the number of pixels that the model is predicting. To test this, we have represented both variables in a scatter plot, as shown in Figure 6.9, where $\#pixels = width * height$. In the figure, we have drawn the trendline obtained using ordinary least squares, which goes right through the data point. We can therefore confirm our hypothesis: the inference time increases approximately linearly.

6.5 YOLO

The last model we have experimented with is YOLO. As opposed to the previous ones, which we have run and trained ourselves, the YOLO model already comes with a set of pre-trained weights which gives us a working model out of the box. This model segments 80 classes from different categories, including vehicles, office items, kitchen items, and food. However, we will only analyze two of them: the laptop and the cell phone.

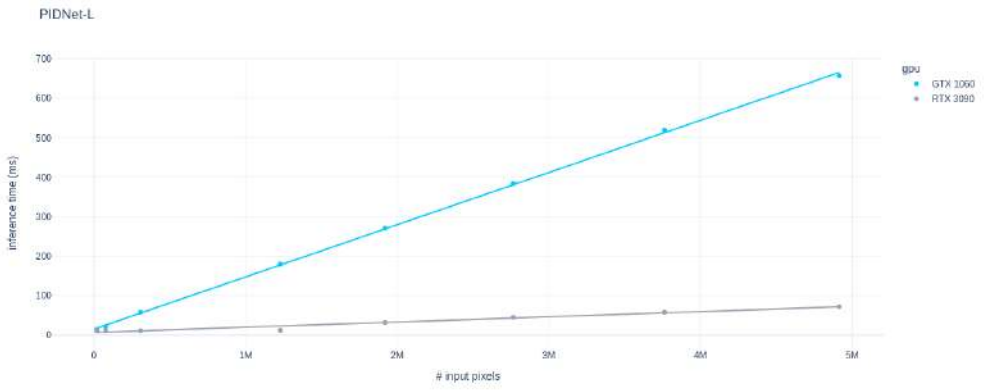


Figure 6.9: Inference time of the PIDNet-L model for different input sizes

model size	background	human	laptop	phone	mean inference time
n	81.67	39.40	46.32	26.81	22ms
s	82.71	47.28	51.24	28.93	27ms
m	84.99	50.45	56.70	34.03	46ms
l	84.86	54.05	56.55	32.52	73ms
x	85.49	56.19	55.95	34.22	105ms

Table 6.13: IoU (in %) and inference time of the different sizes of the YOLOv8 model for the EgoBodies validation set (640x480)

The YOLO model comes in 5 different sizes. We have evaluated each of them using the validation set of the EgoBodies dataset. The results are shown in Table 6.13. We observe that the model achieves decent results. However, they are worse than Thundernet.

Additionally, we observe that the segmentation quality slightly increases with the size of the model, and so does the inference time. Given our real-time constraint, we believe the biggest model that we would be able to use is the small (s) version.

In Figure 6.10, we observe an example of a stereo image segmented with YOLO. We observe that the phone is detected correctly, but the hand is not. Additionally, we observe that the chair is detected correctly on the lower image (right camera) but not on the one above (left camera). This is one of the main problems with YOLO: it creates masks of incredibly quality but it often fails to detect the object altogether. On the other hand, Thundernet detects the presence of the objects most of the time, but the masks are very poor.

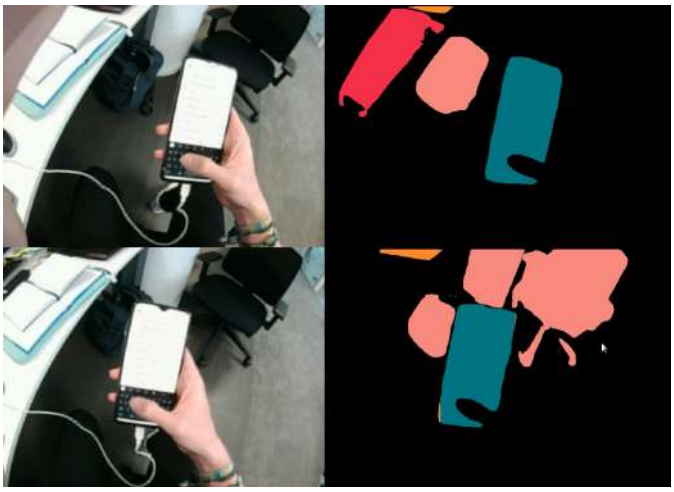


Figure 6.10: Example of an image segmented with YOLOv8

Class key	Value
0	background
1	egocentric human
2	interacting objects
3	exocentric human

Table 6.14: Class key of the exocentric model

6.6 Exploring exocentric segmentation

One of the problems of the experiments run in the previous sections is that our models do not differentiate between egocentric and exocentric people, i.e., the person who is wearing the camera and other people in his field of vision.

This is a very relevant for some applications. We may want to segment other people to immerse them into the virtual reality or we may want to exclude them completely. In this section, we will run an experiment in which we will train the PIDNet architecture adding a new class to predict: an exocentric human. The rest of the classes will be the same we have used previously to train PIDNet with EgoHOS. The new class key is shown in Table 6.14. Note that the interacting object class refers only to objects which the egocentric human is touching. If an object is being held by an exocentric human, it should not be segmented.

To train this model, we need labeled data of exocentric humans. Thankfully, images showing people from this perspective are easier to come by than egocentric images. To do this, we have selected a subset of the ADE20K and COCO datasets. The specific process we have used to select these images is explained in chapter 4.

Class	IoU
background	95.74
egocentric human	90.72
interacting object	55.49
exocentric human	69.38

Table 6.15: IoU (in %) of the experiment with exocentric humans

We have trained a model merging the three datasets: EgoHOS, COCO, and ADE20K. In the data generator, we have had to specify the following class mappings to COCO and ADE20K: {`1: 3`}

What we achieve with this is transforming the labeled human class of the datasets to the new class, exocentric human. On the other hand, no class mapping was specified for the EgoHOS dataset, as their original datasets correspond to the one of this experiment.

The results are shown in Table 6.15. We can extract the following conclusions:

- The IoU of the classes 'egocentric human' and 'interacting objects' is the same as the one obtained in the previous section training with only these two classes. Therefore, adding a new 'exocentric human' class has had no negative effect on the previously achieved results.

- The IoU of the 'exocentric human' class is 69.38%, which is quite satisfactory. However, the variability of this class is extremely high. There are images where the human is really small, appears only partly in the image, or is performing any kind of strange action. Therefore, to better assess this result, we should try the model qualitatively.

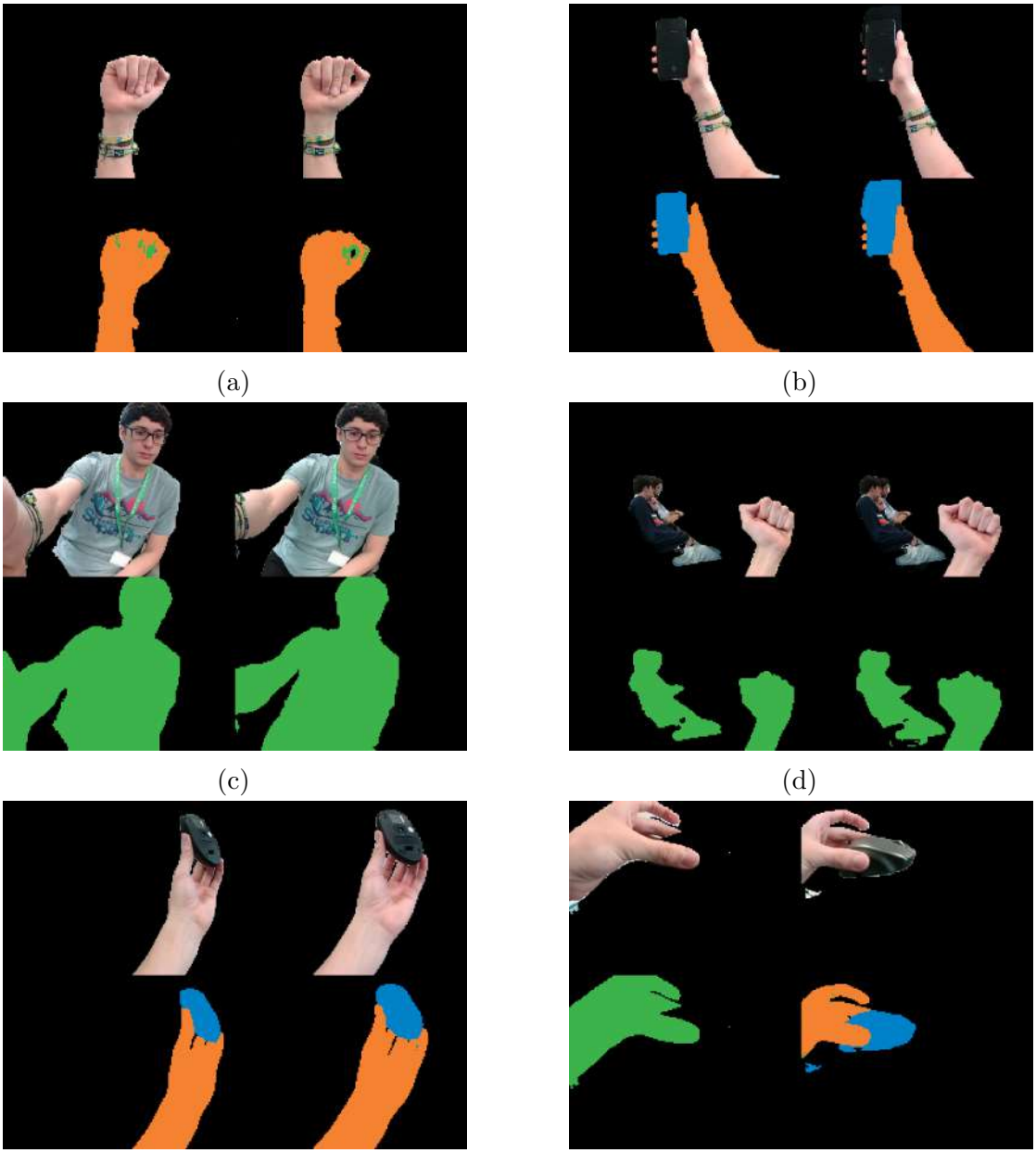
Using the stereo camera, we have done a qualitative evaluation of the model. Some images are shown in Figure 6.11. Classes 1, 2 and 3 are colored in orange, blue and green respectively. From these tests, we can conclude the following:

- When the image only contains one human or a part of him, the model, in general, does a very good job at differentiating egocentric and exocentric humans.

- When the image contains both an exocentric and egocentric human, the model performs poorly and tends to classify everything as an exocentric human. This can be clearly seen in Figure 6.11d.

- The model mostly detects correctly the objects that are in the egocentric human's hand. It does not incorrectly classify objects with whom the exocentric human is interacting. In Figure 6.11f, we observe that when the model incorrectly classifies the human as exocentric (part of the left), it will also not segment the interacting object.

The main problem of this experiment is the fact that the model is unable to segment an egocentric human and an exocentric human simultaneously, which is what we wanted to apply this model too. The most probable reason behind this is the quality of the training data, we have no images where there are both an egocentric and an exocentric human. This is because each of these two classes comes from different datasets, EgoHOS for the former, and COCO and ADE20K for the latter.



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

Figure 6.11: Examples of images segmented with the PIDNet exocentric model

Chapter 7

Parallel problems

7.1 Interpolation

A factor that affects segmentation quality is the output size of the model. PIDNet, with which we have obtained the best results so far, outputs a size one-eighth of the input size. This causes the edges of the segmented class to be very coarse.

The reason behind this is the upsampling technique used, which is called interpolation. This is a mathematical operation that, when increasing the size of an image, assigns the values of the newly created pixels as a function of the pixels that are close. The three most common interpolation algorithms are:

- Nearest: it assigns each new pixel the value of the closest pixel.
- Bilinear: to find the value of a new pixel, it computes the weighted mean between the two nearest pixels along one axis. Then, it repeats the same process of the other axis.
- Bicubic: it is similar to the bilinear case, but instead of using only the two closest samples (one on each side), it uses the four closest ones. With the polynomial obtained from fitting to these points, the new pixel value can be obtained. This process is repeated for each axis.

To test the difference between these techniques, we have taken the prediction of an image through the PIDNet model and interpolated them with each algorithm. We have used the `cv2.resize` function of the opencv package. To set the interpolation algorithm to be used, we just need to set the 'interpolation' keyword argument to its corresponding value.

The results can be seen in image Figure 7.1. To better appreciate the differences, we have zoomed each image to a specific part, which are shown in Figure 7.2. We observe that there is a clear improvement between the nearest interpolation and

(a) (b)

Figure 7.1: Image (top left) and PIDNet prediction using nearest (top right), bilinear (bottom left) and bicubic (bottom right) interpolation

(a) (b)

Figure 7.2: Images of Figure 7.1 zoomed to a specific part

the linear one, as the edges are substantially smoother. On the other hand, the difference between bilinear and bicubic interpolation is barely visible. Therefore, we conclude that bilinear interpolation is the best option. The almost imperceptible improvement of bicubic interpolation is not worth the huge computational overhead. However, the results are far from perfect, as the user is still able to see the coarse edges of the segmentation.

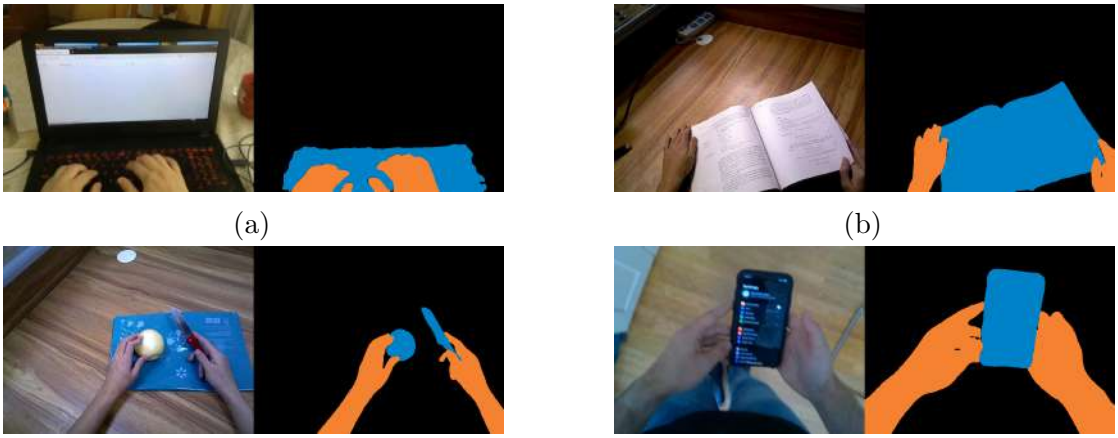
7.2 Solving data scarcity and invariance

One of the most critical problems that we have faced in this dissertation is data scarcity and invariance. We do not have enough training data to achieve the best possible results. Acquiring training data for semantic segmentation is very costly, as a human must manually segment each image, which takes significant time. To give an estimate of the costs, assuming that it takes 5 minutes to label an image (which is a very conservative estimate), we would need more than 800 hours of human labor to obtain 10000 labeled images (around the size of our EgoBodies dataset).

In order to tackle this problem, we had the idea of trying to obtain labeled images through the use of state-of-the-art not-for-real-time semantic segmentation models. There are models that have achieved amazing results in semantic segmentation. Unfortunately, we cannot use them to solve our main problem because their inference time is considerable. However, we could use them to label images, which we will then use to train our existing models.

The first experiment we have run is trying to label our EgoBodies dataset using the class key of the EgoHOS dataset. For this, we have taken a swin transformers model with pre-trained weights on EgoHOS, so we did not have to train the model ourselves. The next step was running the whole dataset through the model and saving the results, some of which are shown in Figure 7.3.

We observe that, in general, the quality of the segmentation is very high. In the example images we have shown, only the one in Figure 7.3a is labeled incorrectly, as it has only detected the keyboard and not the whole laptop. Doing a brief exploration of the new dataset, we realize that, although the incorrectly labeled images do not represent a big percentage, it could be enough to cause problems to our model. Therefore, this dataset should probably be manually filtered before use.



(a) (b)

(c) (d)

Figure 7.3: Images of the EgoBodies dataset and their predictions using a swin transformer

Chapter 8

Conclusions

Some of the conclusions we can extract from this dissertation are:

- We have analyzed the different ways to segment objects for virtual reality applications. The first is with the semantic meaning of the object, and the second is considering all objects which are being touched by the user. We have proved that both approaches can be implemented, and the choice of one over the other should be made based on the use case.

- We have trained a model named Thundernet with the EgoBodies dataset, aiming to segment the human body and a set of objects. We have run many experiments to try to improve the results. Experiments that have been unsuccessful include downsampling the data to increase the percentage of objects, increasing the learning rate, adding a kernel regularizer, increasing the size of the images, training the resnet layer, and applying data augmentation. However, the addition of more PPM factors brought a significant improvement, as did including an attention U-Net layer.

- We have experimented with a pre-trained model named YOLOv8 and compared it with Thundernet. Among the classes that YOLO segments are included the laptop and the phone. YOLO presents promising results. A significant difference between YOLO and Thundernet is that the quality of the YOLO segmentation is a lot higher when the object is identified correctly, i.e., YOLO either detects the object or it does not, but when it does, the segmentation mask is incredibly good. On the other hand, Thundernet may make fewer errors of missing an object, but its segmentation masks could be better, with regions of false positives and false negatives, and rough borders.

- We have trained a really fast segmentation model named PIDNet with a publicly available egocentrcic dataset named EgoHOS. This dataset labels

objects that are interacting with the user irrespective of what they are. We have achieved very good results with this experiment, creating an algorithm that is close to being able to be used in production. We have experimented with different sizes and formats of the images to be trained and discovered that larger images in mono format achieved the best results. We have run inference time tests to confirm that, with the increased size, the model can be run in real time.

- We have identified one of the main obstacles to achieving high-quality results: needing more labeled data. Labeled data is expensive to obtain because it must be manually created, which is costly. We have explored the possibility of using state-of-the-art not-for-real-time semantic segmentation models, which have extremely high values of IoU, to label data, which will be used to train real-time models.

- We have created two small datasets with exocentric humans and trained a PIDNet model that is able to differentiate between egocentric and exocentric people. We have observed that the model performed really well when the image only contained either an egocentric or an exocentric human. However, when it contained both, results were very poor. We believe the cause for this is that the final training dataset contained no images with both an exocentric and an egocentric human. Therefore, the model is not able to correctly predict this type of scenarios. This further proves the fact that the dataset quality has a huge influence in the final results.

8.1 Next steps

In the future, one interesting solution to explore is the use of time data. Sometimes segmenting an object from an image can be hard because the object is too small, it is blurry or other reasons. If we were able to use time data, the model would receive not only the last frame produced by the camera, but also some of the previous one up to a certain window size. This will allow the model to learn not only from the aspect of objects but also from their movement patterns. This has many advantages: for example, if an object is clearly visible in an older frame, the model would be able to track it and still segment it in the future, even if it becomes hard to identify. Naturally, the viability of this approach depends on the computation overhead that it would bring, which could be significant.

Another interesting solution would be to try to combine real-time models with state-of-the-art models that have high inference times. As seen during the development of the project, there are state-of-the-art semantic segmentation models which achieve amazing quality scores, although this comes at a very high computational

cost. However, there are ways in which we can use this information. As the images we are segmenting come from videos, there is significant learning that can be done from an image at a certain time with the images from the previous points in time. The difference with the approach of the previous paragraph is that, in this one, we will run two models in parallel: (1) a real-time and low-quality model, and (2) a high inference time and high-quality model. The real-time model will not only take the last frame captured by the camera but will also receive the frame that was captured by the camera at time $t - \Delta t$, where t is the current time and Δt is the inference time of the high-quality model. To sum up, the real-time model would receive three inputs: (1) the frame captured at time t, (2) the frame captured at time $t - \Delta t$, and (3) the output of the high-quality model of the frame captured at $t - \Delta t$. Therefore, the real-time model can find similarities between the current frame and the older one and take advantage of the high-fidelity segmentation of the better model.

The limitation of this approach is that there must be a relation between the image at time t and the one at time $t - \Delta t$. As Δt increases, the chances of this become smaller. This solution will not work in cases where the user is rapidly moving his head, as there will not be any relation between images captured at close points in time. However, in most scenarios, there will be a relation, especially considering we are trying to segment the human body, which is unlikely to change considerably even if the perspective of the user changes.

Another limitation of the two approaches proposed above is that to train the model, we would need segmented videos and not individual images. This could come at a high economic cost, as the segmentation of images for supervised training is normally done manually. In this case, the number of images needed for a high-quality algorithm would increase significantly. Additionally, to the extent of my knowledge, there are no widespread publicly available datasets of segmented videos.

Bibliography

[1] Diego Gonzalez Morin et al. "Full body video-based self-avatars for mixed reality: from E2E system to user study". In: *Virtual Reality* (Apr. 2023). ISSN: 1359-4338. DOI: 10.1007/s10055-023-00785-0.

[2] Benjamin Lok et al. "Effects of Handling Real Objects and Self-Avatar Fidelity on Cognitive Task Performance and Sense of Presence in Virtual Environments". In: *Presence: Teleoperators and Virtual Environments* 12 (6 Dec. 2003), pp. 615–628. ISSN: 1054-7460. DOI: 10.1162/105474603322955914.

[3] E. Gonzalez-Sosa et al. "Real Time Egocentric Object Segmentation for Mixed Reality: THU-READ Labeling and Benchmarking Results". In: IEEE, Mar. 2022, pp. 195–202. ISBN: 978-1-6654-8402-2. DOI: 10.1109/VRW55335.2022.00048.

[4] Zheng Qin et al. "ThunderNet: Towards Real-time Generic Object Detection". In: (Mar. 2019).

[5] Yansong Tang et al. "Action recognition in RGB-D egocentric videos". In: IEEE, Sept. 2017, pp. 3410–3414. ISBN: 978-1-5090-2175-8. DOI: 10.1109/ICIP.2017.8296915.

[6] Yansong Tang et al. "Multi-Stream Deep Neural Networks for RGB-D Egocentric Action Recognition". In: *IEEE Transactions on Circuits and Systems for Video Technology* 29 (10 Oct. 2019), pp. 3001–3015. ISSN: 1051-8215. DOI: 10.1109/TCSVT.2018.2875441.

[7] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: (Feb. 2015).

[8] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86 (11 1998), pp. 2278–2324. ISSN: 00189219. DOI: 10.1109/5.726791.

[9] Aston Zhang et al. "Dive into Deep Learning". In: *arXiv preprint arXiv:2106.11342* (2021).

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Communications of the ACM* 60 (6 May 2017), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386.

[11] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: (Sept. 2014).

[12] Christian Szegedy et al. "Going Deeper with Convolutions". In: (Sept. 2014).

[13] Jonathan Long, Evan Shelhamer, and Trevor Darrell. "Fully Convolutional Networks for Semantic Segmentation". In: (Nov. 2014).

[14] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation". In: vol. 9351. 2015. DOI: 10.1007/978-3-319-24574-4_28.

[15] Liang-Chieh Chen et al. "DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs". In: (June 2016).

[16] Liang-Chieh Chen et al. "Rethinking Atrous Convolution for Semantic Image Segmentation". In: (June 2017).

[17] Paul-Louis Pröve. *An introduction to different types of convolutions in deep learning*. Feb. 2018. URL: https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d.

[18] Kaiming He et al. "Mask R-CNN". In: (Mar. 2017).

[19] Ross Girshick. "Fast R-CNN". In: (Apr. 2015).

[20] Hengshuang Zhao et al. "Pyramid Scene Parsing Network". In: (Dec. 2016).

[21] Martín Abadi et al. "TensorFlow: A system for large-scale machine learning". In: (May 2016).

[22] *Intersection over Union for Object Detection*. May 2023. URL: https://www.baeldung.com/cs/object-detection-intersection-vs-union.

[23] Ester Gonzalez-Sosa et al. "Real Time Egocentric Segmentation for Video-self Avatar in Mixed Reality". In: (July 2022). URL: http://arxiv.org/abs/2207.01296.

[24] Yansong Tang et al. "Action recognition in RGB-D egocentric videos". In: IEEE, Sept. 2017, pp. 3410–3414. ISBN: 978-1-5090-2175-8. DOI: 10.1109/ICIP.2017.8296915.

[25] Lingzhi Zhang et al. "Fine-Grained Egocentric Hand-Object Segmentation: Dataset, Model, and Applications". In: (Aug. 2022).

[26] Lingzhi Zhang et al. *Owenzlz/EgoHOS: Github of the EgoHOS dataset*. URL: https://github.com/owenzlz/EgoHOS.

[27] Tsung-Yi Lin et al. "Microsoft COCO: Common Objects in Context". In: (May 2014).

[28] Bolei Zhou et al. "Scene Parsing through ADE20K Dataset". In: IEEE, July 2017, pp. 5122–5130. ISBN: 978-1-5386-0457-1. DOI: 10.1109/CVPR.2017.544.

[29] Jiacong Xu, Zixiang Xiong, and Shankar P. Bhattacharyya. "PIDNet: A Real-time Semantic Segmentation Network Inspired from PID Controller". In: (June 2022).

[30] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection". In: (June 2015).

[31] Joseph Redmon and Ali Farhadi. "YOLO9000: Better, Faster, Stronger". In: (Dec. 2016).

[32] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: (Apr. 2018).

[33] Ozan Oktay et al. "Attention U-Net: Learning Where to Look for the Pancreas". In: (Apr. 2018).

[34] United Nations Department of Economic and Social Affairs. *The 17 Goals*. URL: https://sdgs.un.org/goals.

[35] United Nations Department of Economic and Social Affairs. *Goal 4. Quality Education*. URL: https://sdgs.un.org/goals/goal4.

[36] United Nations Department of Economic and Social Affairs. *Goal 8. Decent Work and Economic Growth*. URL: https://sdgs.un.org/goals/goal8.

[37] United Nations Department of Economic and Social Affairs. *Goal 9. Industry, Innovation and Infrastructure*. URL: https://sdgs.un.org/goals/goal9.

Appendix A

Evolution of loss during training

In this appendix, we present a set of graphs where we show the evolution of validation loss during the training of some of the models.

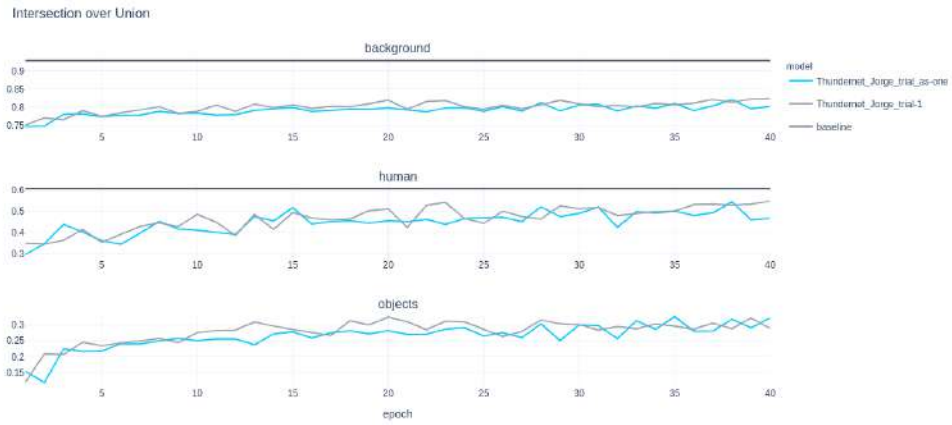
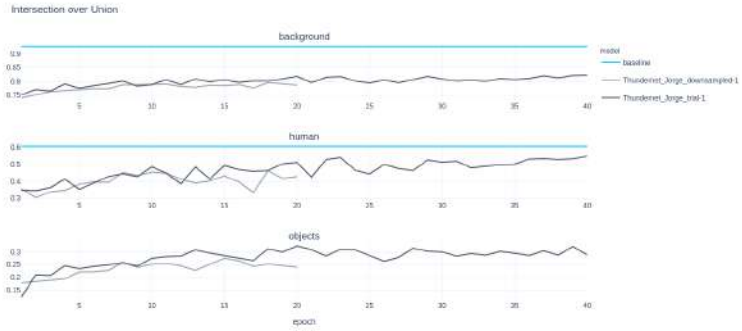
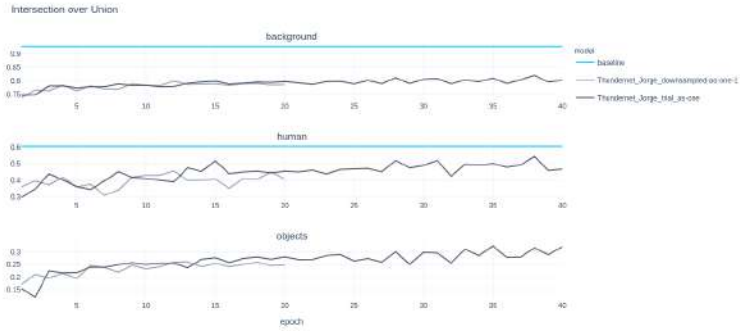


Figure A.1: Evolution of validation IoU during the initial Thundernet experiment



(a) Separated objects



(b) Objects as one

Figure A.2: Validation IoU of the Thundernet downsampled and initial models

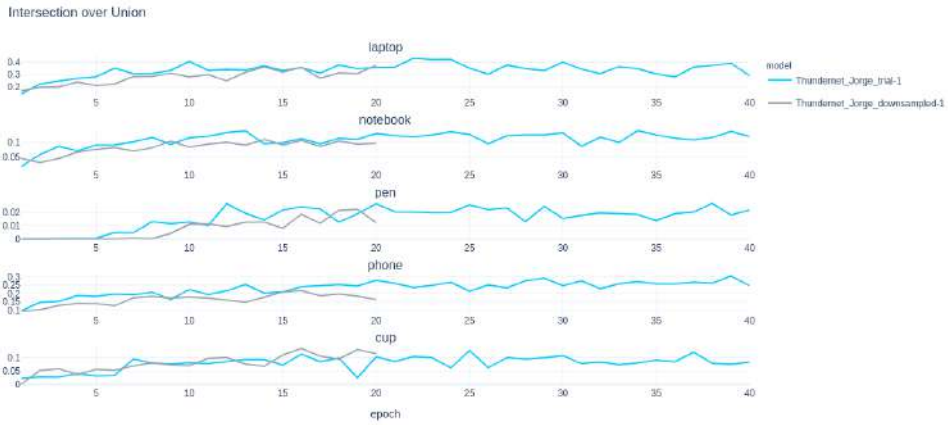
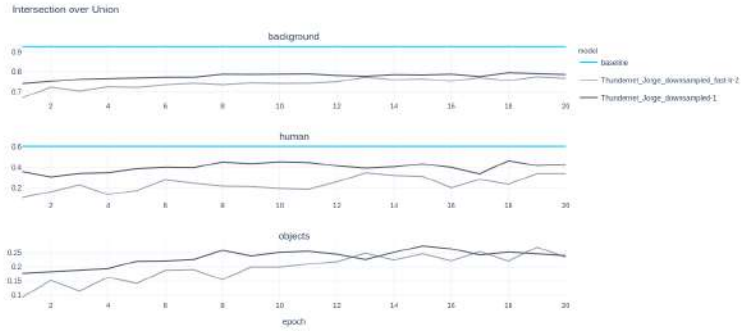
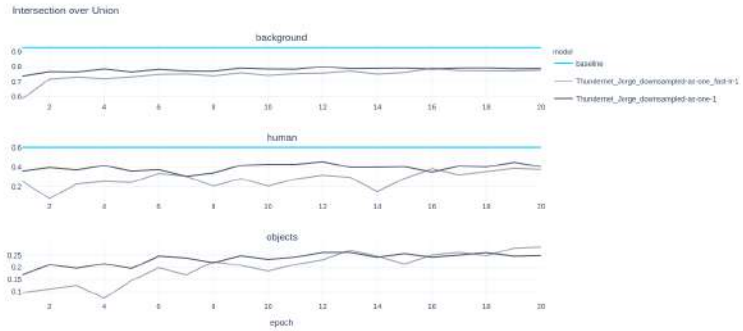


Figure A.3: Validation IoU of the individual objects from the Thundernet downsampled model

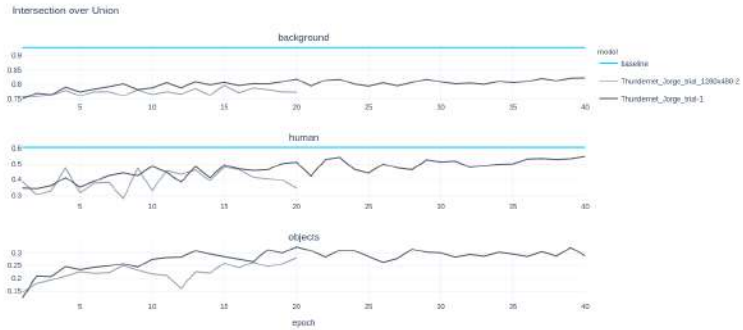


(a) Separated objects

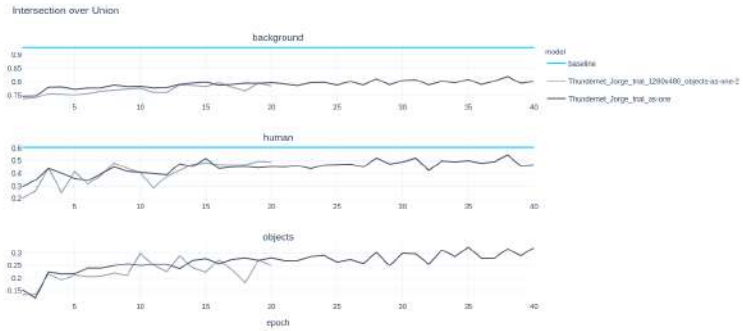


(b) Objects as one

Figure A.4: Validation IoU of the fast lr and initial Thundernet models



(a) Separated objects



(b) Objects as one

Figure A.5: Validation IoU of original (640x480) and increased-size (1280x480) Thundernet models

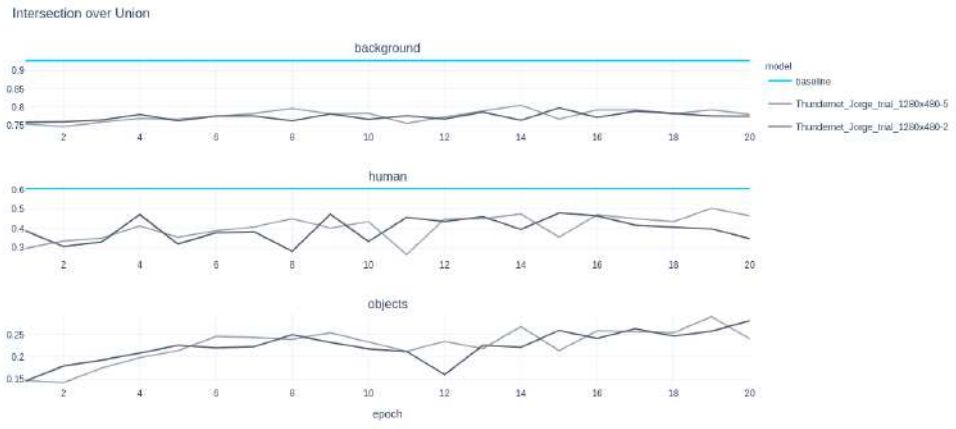


Figure A.6: Validation IoU of the Thundernet model with (light gray) and without (dark gray) trainable resnet layer

Appendix B

Alignment with SDGs

In the globalized and constantly-evolving world of the 21st century, it is essential to consider the impact of everything we do, both professionally and personally, on the world. In particular, the United Nations has proposed a set of seventeen goals to advance toward peace and prosperity in all societies on our planet [34]. These goals are part of the UN 2030 Agenda for Sustainable Development, which all UN member states have adopted. The UN believes that we must all take urgent action within our capabilities to work in line with the SDGs.

In this appendix, we will consider how the project we have developed aligns with several SDGs:

- **Goal 4: Quality Education.** This goal aims to ensure inclusive and equitable access to education throughout the globe. The pandemic has disrupted the education sector, and millions of educational institutions stopped in-person classes. Quality has many advantages apart from learning knowledge and skills. For example, it serves as a lifeline for children who are vulnerable or have difficult situations at home [35]. The development of virtual reality technology may contribute greatly to this SDG. The main benefit is that it gives remote learners a very similar experience to in-person instruction. It will provide access to high-quality education to children growing up in remote areas, as the teacher may be elsewhere.

- **Goal 8: Decent Work and Economic Growth.** Among the targets of this SDG is to increase productivity or guarantee jobs with fair working conditions [36]. Through the use of VR, we can help achieve these objectives. VR can be used in many industry sectors to increase productivity. In healthcare, doctors may practice how to perform a surgery in an environment very similar to the real one. This will reduce the amount of training needed, increasing productivity. Another example is that of maintenance workers of public infrastructure, such as the electricity grid or the fiber optic network. Their

work often involves several risks. The electricity line maintenance worker must make sure that his harness is well attached, and he must avoid touching a powered line. VR technology will allow workers to practice risky tasks in a safe environment to make sure that they are well-prepared for real-world situations and to face a crisis. This contributes to achieving decent work for everyone.

- **Goal 9: Industry, Innovation and Infrastructure.** Among the targets of this goal is to enhance scientific research and increase technological capabilities [37]. This has many benefits, including the fact that highly-technological industries are less impacted by economic crises. The development of VR technology contributes to this goal because of all the research lines that it creates, such as the one we have pursued in this dissertation. Additionally, it also increases digitalisation in all industries in which it is adopted, making these sectors more robust to adverse circumstances.