Universidad Zaragoza

Master's Thesis

Language-aware Neural Feature Fusion Fields for egocentric video

Author

Alejandro de Nova Guerrero

Supervisors

José Jesús Guerrero Campo

Lorenzo Mur Labadía

ESCUELA DE INGENIERÍA Y ARQUITECTURA

December 2024

Abstract

Environment understanding is a crucial characteristic in many applications, from navigation to robot manipulation. Scene comprehension is inherent to human beings but it is a very difficult task for machines. Egocentric videos represent the world seen from a first-person view in which an actor interacts with the environment and their objects. Neural rendering techniques allow learning scene geometric information but lacks of semantic understanding, losing a lot of environment comprehension. Some works focuses on egocentric videos in isolated or short clips or work on large tracks but lack of semantic information. In this project we present Dynamic Image-Feature Fields (DI-FF), a neural network able to reconstruct an egocentric video decomposing its three principal components: static objects, foreground objects and person moving around the scene. Furthermore, it incorporates language-image features that contribute to semantic understanding of the egocentric video. By computing relevancy maps, DI-FF is able to perform detailed dynamic object segmentation by open-vocabulary text queries, having knowledge of surrounding objects barely visible in the current viewpoint. It also enables affordance segmentation for simple action queries. To achieve this, the model is compound by three different neural network streams, one for each scene component: background, foreground and actor, which outputs the pixel color independently to be able to decompose the scene. Additionally, each of these networks predicts language CLIP embeddings to allow object recognition by text queries. With this implementation, DI-FF outperforms state-of-the-art projects, representing a robust basis for future work in egocentric video environment understanding.

Acknowledgements

I would to thank my supervisor, Josechu, for giving me the opportunity to work once again with him. To Lorenzo, thanks for your experience, your knowledge and for being always there for helping me out.

Contents

Chapter 1

Introduction

Scene understanding is inherent to human beings. By just a sight into a scene, we can get a whole picture of the environment, understanding where the relevant objects are in the scene and what actions we can perform with them. These actions or affordances are linked to our spatial memory of the scene, which can be understood as *environmental awareness*. It is strongly related to semantics, i.e., the objects category (mug, chair, sink...) and its properties (color, texture, opacity...), allowing an underlying open-set vocabulary to refer to each of the parts of the scene (brown pot, frying pan...).

Egocentric videos allow understanding the scene from a first-person human being perspective. In egocentric videos, a person with a camera on his head or chest moves around the scene, manipulating several objects along the video, providing interactions with the environment and adding a complexity that is not present in rigid scenes. This connection between actions and objects within a dynamic scene present challenges, but also opportunities to explore environment understanding in egocentric videos.

Methods working on egocentric environment understanding usually take a short or isolated clip ignoring the physical space of the scene [7, 26] or lack of semantic information [31, 18], which represents a fundamental part of the environmental awareness. Humans interact with the environment, combining a geometric and a semantic understanding to capture the context and attributes of the objects in the scene, which can be static or dynamic, and also the available actions to interact with them. Some works explore this idea, combining both semantic and geometric representations [32, 29]. By combining egocentric video spatial understanding with semantic information, these models allow a deep knowledge of the objects in the scene and their attributes.

Recently, with the increasing interest in neural rendering, several projects

dive into scene understanding in terms of spatial information for static scenes but also for dynamic videos. NeRF [21] provide an implicit representation of the geometry and appearance of a static scene. This representation supports semantic encoding, used in many applications such as scene editing [15], robot manipulation [41] or navigation [47]. Other works like [45] extend NeRF to be able to learn semantic information of the scene, by distilling 2D features from a teacher network into the NeRF model that acts as student, predicting semantic features in the 3D geometry. This semantic understanding is extended in works like [12], which supports object retrieving with open-vocabulary text queries, by volume rendering language embeddings.

1.1 Goals and contributions

The main objective of this project is to implement a neural renderer able to decompose the scene geometry, but also capture the semantic information of the scene elements in order to perform dynamic (active) object segmentation by text queries in egocentric videos. With this implementation, we aim to decompose an egocentric video into static background and transient foreground, differentiating between dynamic objects and the person recording. This allows reconstructing the video at different time periods with the possibility to get unseen viewpoints thanks to the flexibility of neural rendering. In addition, by combining it with 3D feature distillation and language embedding predictions, to learn semantic information, we can reconstruct the video focusing on the objects of the scene we want by a text query, performing object and action segmentation computing relevancy maps.

In order to reach our goal, we present the tasks developed and the contributions done during the project:

- We study and overview the state-of-the-art projects related to pure geometric 3D scene neural rendering, semantics feature distillation methods applied to traditional neural rendering and scene understanding in egocentric videos.
- We propose an approach to adapt feature fields to dynamic egocentric videos by dividing the radiance and feature fields depending on whether they are from the actor, foreground, or background elements.
- We distill image-language and semantic features into a neural rendering model using CLIP and DINO foundational models, respectively, to get robust semantic understanding over time. This achieve a consistent segmentation of dynamic objects along the egocentric video using text queries.

- We implement metrics to evaluate dynamic object segmentation task using open-vocabulary text queries.
- We study and evaluate training convergence speed and performance of the scene prediction in different ablation experiments of the neural rendering pipeline.
- We study and evaluate different semantic prediction pipelines for dynamic object segmentation and affordance segmentation tasks within the reconstructed videos.
- Our results show a good performance in scene semantic understanding by using text queries, achieving an improvement of +56.5% over state-of-the-art methods in dynamic object segmentation task. Furthermore, our model shows a strong connection between the objects in the scene and the three levels of scene decomposition, understanding the surroundings and occlusions in the video.

Chapter 2

Background

In this Chapter we describe the theoretical concepts behind neural rendering and some deep learning models that are used in this work. In Section 2.1 we explain the algorithms behind Multi-Layer Perceptron. Section 2.2 explores the most common neural network used for neural rendering tasks. Finally, in Section 2.3 we overview two neural networks that predict semantic information and language embeddings that serve as baseline of many projects in neural rendering field.

2.1 Multi-Layer Perceptron

A Multi-Layer Perceptron (MLP) is a neural network composed by at least three layers: input layer, output layer and one or more intermediate hidden layers. Each one of these layers contain perceptrons connected one to each other perceptron from the previous and next layers, as shown in Figure 2.1.

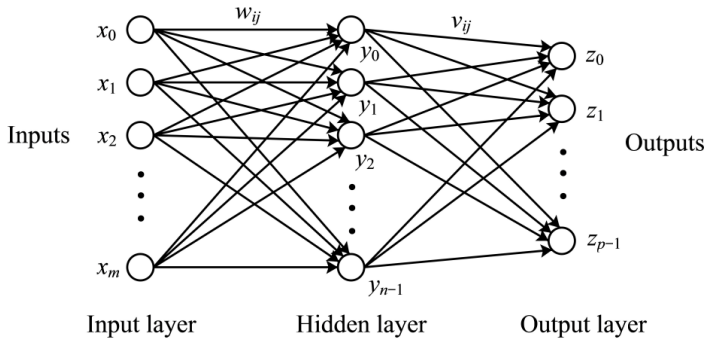

Figure 2.1: Graph of a Multi-Layer Perceptron neural network with one hidden layer. Figure from [40]

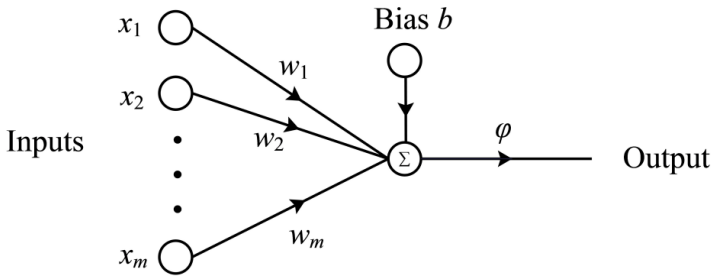

Figure 2.2: Graph of a perceptron. Figure from [40]

Each perceptron acts as a neuron inside the network, being its basic unit. Figure 2.2 illustrates how a perceptron works. It gets a set of inputs x_i that represent the interconnections with other previous neurons and perform a dot product by a set of weights w_i. Then, it sums up a bias b and applies an activation function φ, following Eq. 2.1, in order to predict an output value y which connects with following neurons in the network.

$$y = \varphi(b + \sum_{\forall i} x_i w_i) \tag{2.1}$$

Perceptrons learn through a training process in order to get a network output able to perform some required task, such as object classification. In order to do this training, from a set of inputs x, the neural network predicts an output $\hat{y} = F(x, w)$ that depends on the weights of each neuron. These weights are learnable and are updated each time we pass the input data through the net. This update is done comparing the prediction $\hat{y}$ with the target value y, obtaining a cost function $J(w)$ that represents the difference between prediction and target. Then, a back-propagation algorithm is applied starting from $J(w)$ and flowing back to the hidden layers of the neural network. During this back-propagation, the algorithm computes the gradients $\nabla_w J(w)$ of each weight with respect to the cost function, indicating in which direction should the weights vary to minimize the cost function value. Lower values of the cost function indicate that the model prediction is more similar to the desired target. Finally, an optimizer updates each weight value using the gradients computed and a learning rate. This step is repeated recursively, feeding the net with inputs, predicting an output and updating the weights using back-propagation, until the cost function converges.

2.2 Neural Radiance Fields

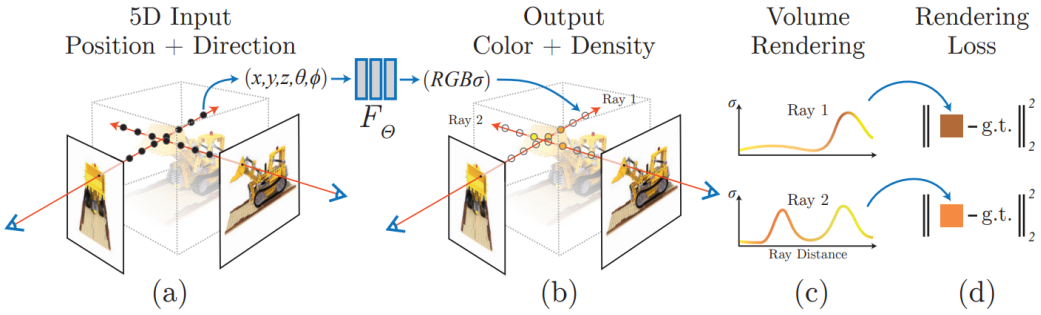

Figure 2.3: **NeRF overview.** It samples 5D coordinates (location and viewing direction) along a camera ray for each pixel (a), which are fed into an MLP to predict color and volume density (b). Using volumetric techniques, it compounds the pixel color value along the ray, compositing the rendered image (c). MLP optimization is done by minimizing the residuals between the predicted image and the ground truth image. Figure from [21]

Neural Radiance Fields (NeRF) [21] is an MLP that learns and optimizes the radiance field of a static scene from a set of input images and camera poses using the pipeline shown in Figure 2.3. This allows the synthesis of novel viewpoints of the scene that are not in the original samples.

A radiance field is a five-dimensional function that represents the emitted radiance of each 3D point $\boldsymbol{x} = (x, y, z)$ of the scene at each direction (θ, ϕ), represented by its 3D Cartesian unit vector $\boldsymbol{d} \in \mathbb{R}^3$. As output, NeRF obtains an RGB color $\boldsymbol{c} = (r, g, b)$ and volume density σ at each sampled point along the ray. This volume density controls the radiance accumulated by a ray with direction $\boldsymbol{d}$ passing through $\boldsymbol{x}$, i.e., it represents volume occupancy along the ray. On the one hand, color prediction is done using both 3D position and direction, so it is be view-dependent. On the other hand, volume density is predicted using only the 3D position to obtain a consistent multiview representation, as depicted in Figure 2.4.

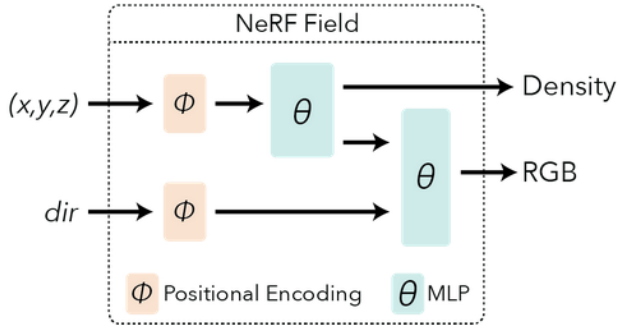

Figure 2.4: NeRF field overview. Figure from [1]

For training the model, NeRF employs a set of input images with known camera pose and calibration. For each image it traces a ray $\boldsymbol{r}(t) = \boldsymbol{o} + t\boldsymbol{d}$ from the camera origin $\boldsymbol{o}$ towards each pixel direction $\boldsymbol{d}$, getting a ray per pixel in the image. By sampling several 3D points along each ray $\boldsymbol{r}$, NeRF is fed-forward to obtain the color and volume density of each point. Then, by using classic volume rendering techniques [10], NeRF estimates the color of each pixel along each ray. The estimated color $C(\boldsymbol{r})$ of each ray $\boldsymbol{r}$ following these volume rendering techniques is:

$$C(\boldsymbol{r}) = \int_{t_n}^{t_f} T(t)\sigma(\boldsymbol{r}(t))\boldsymbol{c}(\boldsymbol{r}(t), \boldsymbol{d})\, dt\,, \; T(t) = \exp\left(-\int_{t_n}^{t} \sigma(\boldsymbol{r}(t))\, dt\right) \quad (2.2)$$

where t_n and t_f are the near bound and far bound of the rendered volume, respectively, $\sigma(\boldsymbol{r}(t))$ is the volume density at a point of the ray $\boldsymbol{r}$, and $\boldsymbol{c}(\boldsymbol{r}(t), \boldsymbol{d})$ is the color at a point of the ray towards a viewing direction. $T(t)$ represents the accumulated transmittance of a ray from t_n to t, i.e., the probability of the ray not hitting any particle from t_n to t.

To represent high-frequency variation in color and geometry in a better way, NeRF maps the inputs to a higher dimensional space, using high frequency functions. Since deep neural networks are biased to learn low-frequency functions [37], this input mapping before feeding the network provides a better fitting of data with high frequency variation. The model uses a function (positional encoding) that maps the input from $\mathbb{R}$ to $\mathbb{R}^{2L}$. It is applied separately to the point location $\boldsymbol{x}$ and to the Cartesian viewing direction unit vector $\boldsymbol{d}$. The positional encoding function γ is computed as:

$$\gamma(p) = (sin(2^0\pi p), cos(2^0\pi p), ..., sin(2^{L-1}\pi p), cos(2^{L-1}\pi p)), \quad (2.3)$$

[1] https://docs.nerf.studio/nerfology/methods/nerf.html

where p is the input coordinate that is mapped and L is the frequency to map into, which is set empirically.

In order to avoid recursively sampling points at free space or occluded areas, NeRF optimizes two different models: a coarse and a fine model. They use stratified sampling to get N_c 3D points $\boldsymbol{x_i}$ along each ray $\boldsymbol{r}$ to evaluate the coarse model, obtaining the estimated color of the pixel $\hat{C}_c(\boldsymbol{r})$ along the ray by summing up the color of each sampled point, weighted by the probability that a photon reaches the camera from that point, as follows:

$$\hat{C}_c(\boldsymbol{r}) = \sum_{i=1}^{N_c} v_i(1 - T_i)\boldsymbol{c}_i, \text{ where } v_i = \left(\prod_{j=1}^{i-1} T_j\right), \tag{2.4}$$

where $\boldsymbol{c_i}$ is the predicted color for each sample and T_i represents the transmittance for each sample, calculated as:

$$T_i = e^{-\sigma_i \delta_i} \tag{2.5}$$

In this equation, σ_i is the predicted volume density for each sample and δ_i is the distance between two adjacent samples.

From these samples, they perform a more informed sampling using a piecewise Probability Density Function (PDF) along the ray, obtaining N_f samples in the relevant parts of the scene. These samples are evaluated through the fine model to obtain the estimated color of the ray $\hat{C}_f(\boldsymbol{r})$ in the same way as 2.4 but summing up along $N_c + N_f$, being the final pipeline of the model represented in Figure 2.5

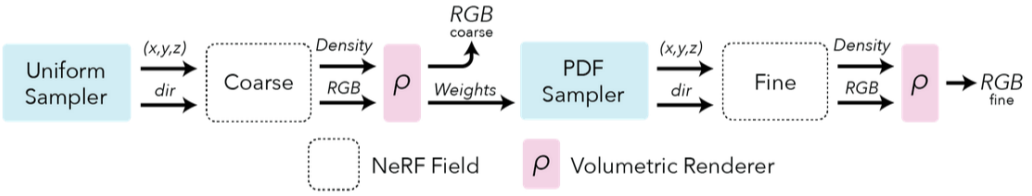

Figure 2.5: **NeRF model pipeline.** NeRF samples uniformly a set of points along each ray corresponding to each pixel of a frame. The 3D positions and the viewing direction pass through a coarse model, predicting the color and volume density of each sample. With the volume density, it samples more points following a PDF towards zones of interest in the scene. These samples pass through the fine model to get the color of each sample. By volumetric rendering it computes the final pixel color. Figure from [2]

[2]https://docs.nerf.studio/nerfology/methods/nerf.html

2.3 Foundational models

Foundational models are large neural networks trained on vast datasets, so they can be used in many different applications. Some NeRF-based models employ foundational models able to extract semantic information to achieve semantic understanding of the scene, exploiting its versatility. We overview two different foundational models: DINO and CLIP.

2.3.1 DINO

DINO [2] is a self-supervised learning (SSL) contrastive model. SSL consists in training a model with its own generated data, without any ground truth label. DINO is trained with two different neural networks: one of them predicts features for the whole input image and the other one predicts features of cropped images, trying to fit the latter with the global image representation.

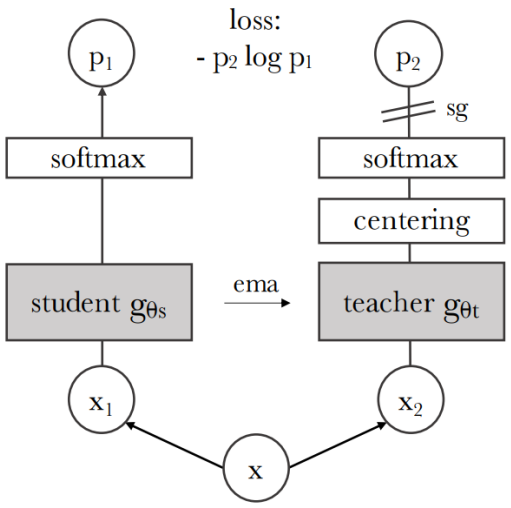

Figure 2.6: DINO architecture overview. Figure from [2]

The model consists on a teacher-student architecture, each one using a Vision Transformer (ViT) [6] as backbone, in order to achieve a self-supervised learning, as shown in Figure 2.6. To build the teacher network they use past iterations of the student network during training. The training is done on ImageNet [39] dataset and they feed both models with the input image. For the teacher network, they use the global view of the image, but for the student network they use several cropped images of the input image, enhancing local-to-global correspondences. By using different ViT models as backbone they achieve different performances in relation to the computational cost,

as well as different features output dimension. In the end, DINO is able to learn class-specific features, obtaining self-attention maps that really cover the class instance with a high performance, as shown in Figure 2.7. DINO features are widely used in neural rendering field [45, 12] to obtain semantic information, due to its object decomposition properties, making it a better alternative to features extracted from classic pre-trained Convolutional Neural Networks (CNNs).

Recently, a new version of DINO was released, named DINOv2 [30], which improves the performance of the first version by combining different novel techniques. The main contribution of DINOv2 is using curated data for training the model. DINO training is self-supervised using a vast amount of uncurated data, which is non-filtered data that can be wrongly annotated or imbalanced. The use of this data leads to a worse performance of the model. To avoid this, DINOv2 generates curated data to feed the model and train it using a good quality dataset.

Figure 2.7: **DINO self-attention maps.** The maps show class-specific features learned by self-supervised learning, achieving object segmentation without any label. Figure from [2]

2.3.2 CLIP

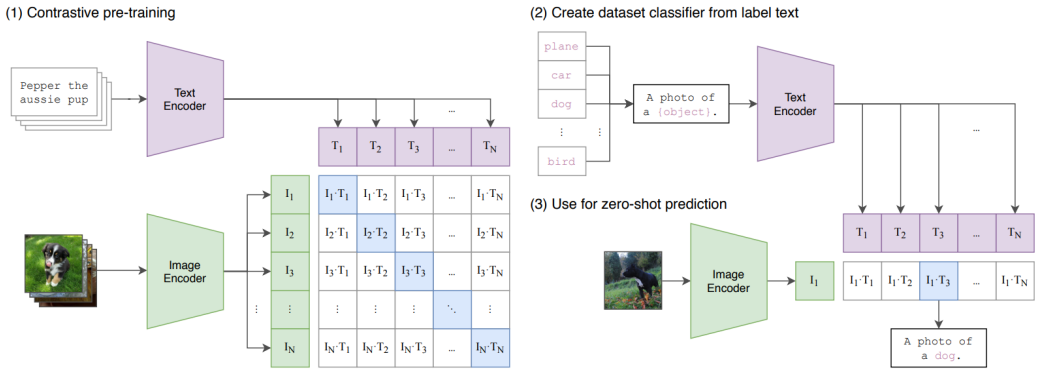

Figure 2.8: **CLIP overview.** (1) By contrastive learning, CLIP learns to match (image, text) pairs by computing embeddings for both the images and texts. (2) During inference, it embeds a list of texts and (3) perform zero-shot classification by computing a probability distribution with respect to the embedding of the input image. Figure from [36]

CLIP [36] is a neural network composed by an image encoder and a text encoder, trained to pair image and text, as shown in Figure 2.8. It uses contrastive learning to extract representations from both the images and texts, contrasting positive and negative pairs of data. In order to do this, CLIP computes image embeddings using a ViT and text embeddings using a Transformer [46]. During training, CLIP maximizes the cosine similarity between the image and text embeddings for the real (image, text) pairs and minimize the cosine similarity for the rest of incorrect pairs. The cosine similarity between two embeddings is calculated as:

$$loss(x_1, x_2, y) = \begin{cases} 1 - cos(x_1, x_2), & \text{if } y = 1 \\ max(0, cos(x_1, x_2) - margin), & \text{if } y = -1 \end{cases} \tag{2.6}$$

where x_1, x_2 are the embeddings we are comparing and y is the target (1 to maximize the similarity and -1 otherwise). In addition, there is a margin to modify the behavior of the loss.

During inference, the image encoder computes the feature embedding of the input image and the feature embeddings of a list of possible texts to match with the given image. Then, the model calculates the cosine similarity between the text embeddings and the image embedding and normalize it, creating a probability distribution using a softmax function. In the end, this is the behavior of a linear classifier, but predicting from a set of different

texts which one is the most likely to match the given image. This allows CLIP to perform zero-shot classification, i.e., predicting on unseen classes during training, relating a given text with an image.

Chapter 3

Related work

In this Chapter we overview the state of the art related to our project. In Section 3.1 we review neural renderers for both static and dynamic scenes. Then, we introduce works that combine feature distillation and language predictions with neural rendering in Section 3.2. Finally, in Section 3.3 we focus on methods that enhances environment understanding in egocentric videos.

3.1 Scene Neural Rendering

If we refer to 3D scene neural rendering from 2D images, Neural Radiance Fields (NeRF) [21] is the model that set the basics. The authors present a learning-based method able to generate novel 3D views of a static scene using a camera pose as input. In order to do that, they use an MLP fed by 3D points and a viewing direction along a ray to predict the volume density and color of the correspondent pixel by learning a radiance field from classic volume rendering techniques. NeRF-W [20] extends NeRF to handle the representation of transient objects that occlude the static scene. They add a learned appearance term to model the photometric variation of the static scene, as well as a transient embedding linked to a transient MLP to predict the color and density variation over time. D-NeRF [34] extends NeRF to allow rendering novel views at arbitrary timestamps of a dynamic scene by adding a temporal component to the input. NSFF [16] learns the 3D scene flow and a disocclusion term over temporal points of a dynamic scene using NeRF as backbone. Nerfacto, described in [42] combines different techniques applied to neural rendering among different papers to create a novel model, more optimized and efficient in static scenes than the original NeRF. The authors achieve this modifying the classic NeRF pipeline, adding

pose refinement to the input data and changing positional encoding by more efficient input encodings, such as hash encoding [23]. NeuralDiff [43] focuses on dynamic scene reconstruction in egocentric videos. The authors decompose the scene into static background and dynamic foreground, dividing it into transient objects and the actor that manipulates the objects, using a triple-stream neural rendering network.

3.2 Feature Fusion Fields

Classic NeRF-based models are thought just to recover 3D scenes with geometric and, in certain cases, temporal understanding. However, they lack of semantic information to perform other tasks such as object segmentation or object retrieval. By extending NeRF, 3D feature distillation methods [8, 15, 17] transfer 2D image features from a teacher DINO [2] network into a 3D neural renderer student that learns to predict those features. N3F [45] follows the same strategy but it is also adapted to dynamic scenes, expanding NeuralDiff. LERF [12] combines neural rendering with CLIP [36] language embeddings in handheld recorded videos, extracting 3D relevancy maps for language queries in real time. The authors combine CLIP embeddings with DINO features to regularize the CLIP predictions. N2F2 [1] supports scene rendering at different scales by learning an unified feature field, encoding the semantic information into a high-dimensional feature space. LangSplat [35] extends 3D Gaussians Splatting [11] and combines CLIP features with multi-scale SAM [14] masks, improving the segmentation quality. EgoLifter [9] augments 3D Gaussian Splatting with instance features from egocentric videos, getting only the static components of the scene by filtering out the actor and the dynamic objects.

3.3 Egocentric environment understanding

Several projects that capture environment understanding in egocentric videos consider an isolated or short clip. Some of these works achieve semantic information, supporting action anticipation [26, 7] or object segmentation [44]. However, they ignore the physical space of the scene. EgoEnv [28] enhances egocentric video understanding by incorporating awareness of the surrounding environment into video representations. The authors define a local environment state, including the positions and distances of objects relative to the person recording, providing a semantic and geometric understanding of the surrounding physical space. Ego-Topo [27] decompose the scene into a

topological map, where the nodes represent environment zones with a coherent set of interactions linked by their spatial proximity. Ramakrishnan et al. [38] capture environment-level representations, learning an environment predictive coding, which applies later for navigation, predicting trajectories.

By contrast to these methods, some works in egocentric videos build a semantic explicit representation from indoor scenes using SLAM (Simultaneous Localization And Mapping). Semantic MapNet [3] encodes each egocentric frame, projects the egocentric features into appropriate locations and then, decodes the semantic features into a top-down 2D map. Liu et al. [18] recognize and localize actions and activities in an existing 3D voxel representation from an egocentric video using a hierarchical volumetric representation to capture geometric and semantic information.

With the recent apparition of egocentric video datasets that contain known camera poses, novel works were developed. Mur-Labadia et al. [25] extract 2D affordance segmentation maps to build a point cloud of the environment encoding those labels. Plizzari et al. [32] track active objects through their appearance and spatial consistency in the 3D scene, even when they are out of view.

Chapter 4

Method

Our approach aims to combine the capability of neural rendering to reconstruct the geometry of egocentric videos with the semantic information of the scene obtained with 3D feature distillation to get a model able to differentiate objects within the video with just a text query. We base our implementation on NeuralDiff [43] model, explained in Section 4.1, for the geometric representation. On top of it, we implement the semantic prediction parts, following LERF [12] strategy, with key modifications to adapt the implementation to our baseline, as presented in Section 4.2 . With this, we jointly train a model able to reconstruct a dynamic egocentric video from different viewpoints, decomposing the scene into static and dynamic components, and compute object relevancy maps by a single text query to perform object and action segmentation. We name our model DI-FF (Dynamic Image-Feature Fields).

4.1 Dynamic Neural Rendering

We define an egocentric video x as a set $(x_t)_{t\in[0,\dots,T-1]}$ of T frames over the time t, being each of them an RGB image $x_t \in \mathbb{R}^{H\times W\times 3}$. Each frame is dependent on the camera pose $g_t \in SE(3)$, where $SE(3)$ is the group of Euclidean transformations, due to the person movement, as well as on the static background B_t and the moving foreground F_t, so we can define each frame as a function $x_t = h(B_t, F_t, g_t)$. Classic NeRF-based models work on static scenes, converting the frame function to $x_t = h(B_t, g_t)$. However, egocentric videos are dynamically variable, with a moving person manipulating the objects in the scene in every frame, making it more complex than a static scene. NeuralDiff maps $h(B_t, F_t, g_t) \rightarrow f(g_t, t)$, so it only needs the camera pose g_t and a temporal point t to reconstruct the video at frame x_t, being

able to render unseen views of the original video.

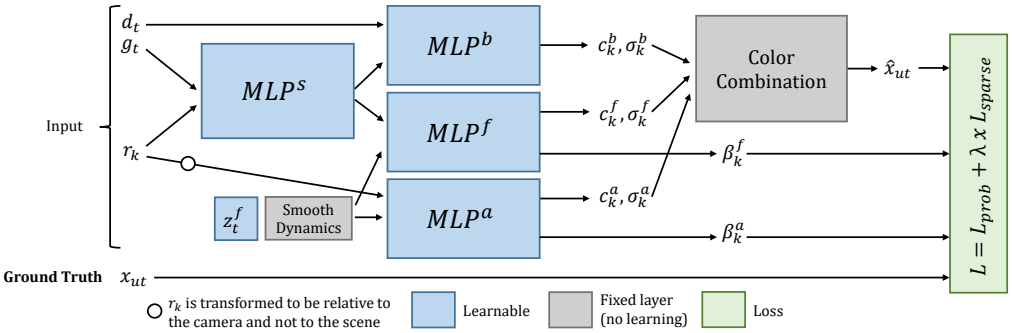

Figure 4.1: **NeuralDiff overview.** Given a camera pose g_t, a viewing direction d_t and a frame latent code z_t^f, the model learns to predict the pixel color $\hat{x}_{ut}$ associated to each ray r_k, by combining the information from three different streams for background, foreground and actor components of the scene.

The architecture of NeuralDiff model (See Figure 4.1) consists of three different MLP networks. Unlike the original NeRF model that only has one MLP for the static scene, NeuralDiff decompose the scene into static background, moving foreground and actor (person). The background network MLP^b computes the color $\boldsymbol{c_k^b} \in \mathbb{R}^3$ and the volume density $\sigma_k^b \in \mathbb{R}_+$ for each sampled 3D point in world frame coordinates along a ray $\boldsymbol{r_k}$ towards each pixel of the frame, given a camera pose g_t and a unit-norm viewing direction $\boldsymbol{d_t} \in \mathbb{R}^3$, so $(\boldsymbol{c_k^b}, \sigma_k^b) = MLP^b(g_t \boldsymbol{r_k}, \boldsymbol{d_t}, z_t^b)$. Additionally, to account for the photometric variation of the scene, it uses a frame-specific appearance code $z_t^b \in \mathbb{R}^B$, which encodes each frame into a B-dimensional vector using learnable parameters. The final color of the pixel is obtained from the average color of the points along the ray, weighted by the probability of a photon reaching the camera from that point. For this calculation, the followed strategy is the one explained in Section 2.2, using a stratified sampling plus a piecewise PDF sampling to optimize two different networks (coarse and fine), as well as positional encoding for the 3D points and viewing directions. It is important to notice that the final volume density calculation over each ray is done in the same way as the estimated color for each pixel, as described in Eq. 2.4, substituting $\boldsymbol{c_i}$ by σ_i.

In order to predict the foreground and actor appearance, the strategy used is different. In addition to color and volume density, both foreground network MLP^f and actor network MLP^a predicts an uncertainty value β_k^f, β_k^a for each point along the ray $\boldsymbol{r_k}$. This prediction captures the aleatoric uncertainty in the input data, i.e., the inherent noise. In this case, this uncertainty

refers to the observed pixel color, so by regularizing the loss function with these parameters, the model is adapted to ignore unreliable pixels. In the same way as the color and the volume density, the uncertainty along each ray is calculated following Eq. 2.4, changing $\boldsymbol{c_i}$ by β_i. By contrast to the background network, both foreground and actor networks are optimized only through the fine model, in order to render the transient part using only the most relevant samples in which is more likely to have objects. Dynamic foreground appearance is time-dependent, so we need to add this dependency into the model to represent correctly the color $\boldsymbol{c_k^f} \in \mathbb{R}^3$ and the volume density $\sigma_k^f \in \mathbb{R}_+$ along time. In order to model this temporal dependency, the model incorporates a frame-specific latent code $z_t^f \in \mathbb{R}^D$ to encode the foreground variation over time, so $(\boldsymbol{c_k^f}, \sigma_k^f, \beta_k^f) = MLP^f(g_t \boldsymbol{r_k}, z_t^f)$. In egocentric videos, most of the foreground objects remain static in most of the frames of the video. Because of this, z_t^f is computed as a low-rank expansion of the trajectory of states, being $z_t^f = B(t)\Gamma$, where $B(t) \in \mathbb{R}^P$ is a fixed basis such that $B(t)$ is an harmonic encoding of time, equivalent to Eq. 2.3 with frequency P, and the motion $\Gamma \in \mathbb{R}^{PxD}$ is a set of coefficients that can be learned during training. It is important to notice that MLP^b and MLP^f share the weights of their initial layers, which are defined as MLP^s.

Finally, for modeling the actor movement, MLP^a is in charge of capturing the parts of the person that appear over the frames. It works in the same way as MLP^f but with the key difference of using the input 3D points in camera frame reference instead of in world frame reference, so $(\boldsymbol{c_k^a}, \sigma_k^a, \beta_k^a) = MLP^a(\boldsymbol{r_k}, z_t^a)$. This is because the camera is fixed to the person's body, moving together with it, so there is a variation of the camera frame reference over the video. By contrast, background objects are invariant in world frame reference. The same happens with the foreground objects when they are not manipulated, which happens in most of the frames. Since this model predicts three different components (background, foreground and actor), it presents a color composition strategy, using the volume density σ_k^b, σ_k^f, σ_k^a and the color $\boldsymbol{c_k^b}$, $\boldsymbol{c_k^f}$, $\boldsymbol{c_k^a}$ of each component for each ray $\boldsymbol{r_k}$, in order to obtain a better scene representation.

Photometric loss function. The model is supervised according to [43], computing a photometric loss $\mathcal{L}_{photo}$. This loss function is divided into three individual loss functions: coarse model color loss $\mathcal{L}_{rgb}^c$, uncertainty loss $\mathcal{L}_{prob}$ and sparse loss $\mathcal{L}_{sparse}$.

For the coarse model color loss, the calculation is simply a Mean Squared Error (MSE) loss between the ground truth color $\boldsymbol{c_u^{gt}}$ of each pixel u of a frame (M pixels) and the predicted color $\boldsymbol{c_u^c}$ by the coarse model for each

pixel, as follows:

$$\mathcal{L}^{c}_{rgb} = \frac{1}{M}\sum_{u=1}^{M}(\boldsymbol{c^{c}_{u}} - \boldsymbol{c^{gt}_{u}})^2 \tag{4.1}$$

The uncertainty loss, referred to the fine model color loss is calculated in a different way than the coarse model one. In order to have into account the uncertainty of the pixel color for the transient MLPs prediction, we compose the loss function using the uncertainty values predicted for each pixel u as:

$$\mathcal{L}_{prob} = \frac{1}{M}\sum_{u=1}^{M}\left(\frac{(\boldsymbol{c^{f}_{u}} - \boldsymbol{c^{gt}_{u}})^2}{2\beta_u^2} + log\beta_u^2\right) \tag{4.2}$$

The first term of the equation represents the MSE between the ground truth pixel color $\boldsymbol{c^{gt}_{u}}$ and the predicted pixel color $\boldsymbol{c^{f}_{u}}$, normalized by the variance β_u^2, calculated summing up the predicted uncertainty values by the transient and the actor networks. This uncertainty term achieves a lower penalty when β_u is higher, which means that the model gives less importance to pixels that are more likely to belong to transient objects. The second term acts as a regularizer of the first term to prevent the model to predict large uncertainty values.

The sparsity loss, calculated over the volume density of the transient components (foreground and actor), is an L_1 regularizer that penalizes the volume occupancy of the transient objects, biasing it towards sparsity. It is calculated as:

$$\mathcal{L}_{sparse} = \frac{1}{M}\sum_{u=1}^{M}(\sigma_u^f + \sigma_u^a), \tag{4.3}$$

where M is the total amount of pixels of a frame and σ_u^f,σ_u^a are the volume densities predicted by the foreground MLP and actor MLP, respectively, for each pixel u.

The final loss for the neural renderer L_{photo} is calculated as:

$$\mathcal{L}_{photo} = \mathcal{L}^{c}_{rgb} + \mathcal{L}_{prob} + \lambda\mathcal{L}_{sparse}, \tag{4.4}$$

where λ is a weight parameter set to 0.01.

Input data encoding. NeuralDiff [43] baseline employs positional encoding for the input data, both 3D locations and viewing directions. However, other works such as [42], propose using other encoding methods that can lead

to better results and faster convergence speed. Therefore, we also include in our model these encodings: hash and spherical harmonics.

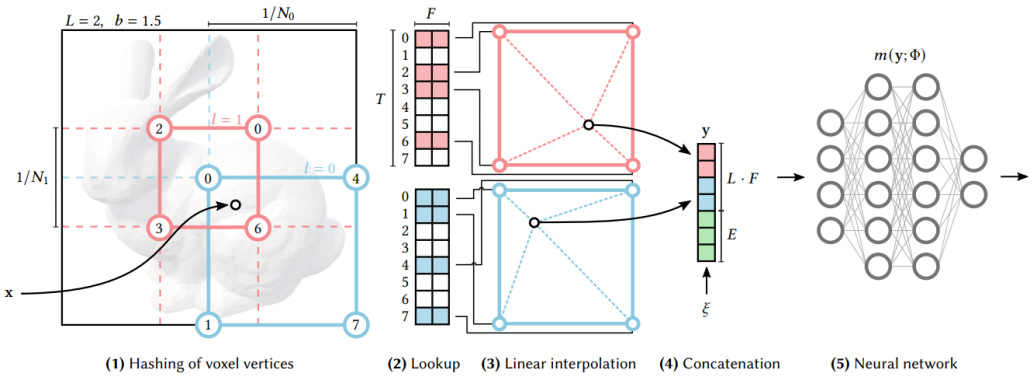

Figure 4.2: **Hash encoding overview.** (1) It assigns each input coordinate x to surrounding voxels at L resolution levels by hashing its coordinates. (2) By a look-up table, it takes the F-dimensional feature of each assigned corner. (3) With linear interpolation of the values according to the relative position of x within the respective l-th voxel, (4) it concatenates the result of each level with auxiliary inputs $\xi \in \mathbb{R}^E$, producing the encoded output $y \in \mathbb{R}^{LF+E}$, (5) which is used as MLP input. Figure from [24].

Hash encoding [24] is a learnable type of encoding used to encode input 3D points in NeRF-based models. The trainable parameters are divided into levels L, each one containing up to T feature vectors with dimensionality F. Each level has an associated resolution N_l calculated from a minimum resolution N_{min} and a maximum resolution N_{max} as follows:

$$N_L = N_{min} b^l, \text{ where } b = \exp\left(\frac{ln(N_{max}) - ln(N_{min})}{L-1}\right) \tag{4.5}$$

As shown in Figure 4.2, input data passes through each level, which is independent to the others. Each input coordinate is scaled by the level resolution N_l and rounded up and down to get a voxel with those two values. By hashing the coordinates of the voxel, its corners are assigned to voxels of the current level. Then, with a look-up table, it gets the features for each of that corners from the feature vector of that level. Finally, by interpolating those features with the relative position of the input coordinate with respect to the assigned corners, it gets the features for that input at that level. By concatenating the features along every level it computes the encoded input for feeding a neural network.

Spherical harmonics [4] represent functions defined on the surface of a sphere, so it is very common to use this kind of functions to encode direc-

tional information. Spherical harmonics are based on harmonic functions of a specific order, so the higher the order the higher is the frequency covered (similarly to positional encoding).

4.2 Dynamic Image-Feature Fields (DI-FF)

Using NeuralDiff [43] architecture as backbone, we extend it to be able to learn the semantic information of the scene, grounding language embeddings to allow text queries, into a model we refer to as DI-FF (Dynamic Image-Feature Fields), illustrated in Figure 4.3. Following LERF [12], we implement a multi-scale CLIP embedding rendering together with the volumetric rendering done by NeuralDiff. In [12], the model is able to work only in static scenes, so they perform classic volumetric rendering for the scene geometry, following a single neural renderer pipeline. Then, they get a subset of sampled points, looking for the ones with more probability to be occupied by objects, using the predicted volume density of the sampled points along the ray. By encoding those 3D points using a hash encoding, they feed a language embedding field that renders CLIP embeddings over a volume centered at each of those points, as well as DINO features for each point.

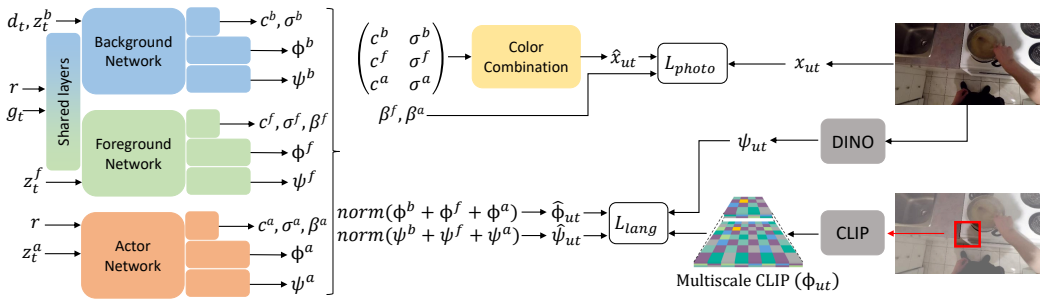

Figure 4.3: **DI-FF overview.** Given a camera pose g_t, a viewing direction d_t, a frame latent code z_t^f, z_t^a and a frame appearance code z_t^b, our model predicts the pixel color $\hat{x}_{ut}$ associated to each ray r, combining the information from three different streams for background, foreground and actor components of the scene. In addition, it renders a CLIP embedding $\hat{\phi}_{ut}$ along the ray, which is supervised with a multi-scale CLIP ground truth and regularized predicting DINO features $\hat{\psi}_{ut}$.

4.2.1 Semantics prediction with CLIP

To allow semantic understanding, we employ CLIP in order to predict language embeddings. In DI-FF, the baseline used has three different MLPs

for each one of the parts of the video (background, foreground and actor). For this reason, we decide to extend each one of these MLPs with an extra MLP network that predicts the CLIP embeddings for each sampled point along each ray $\boldsymbol{r_k}$, optimizing it only through the fine model. In order to calculate these D-dimensional embeddings $F_{lang}(g_t\boldsymbol{r_k}, s_k) \in \mathbb{R}^D$, we need also a scale s_k. This scale corresponds to the side length in world frame reference of a cube centered at each point of the ray $\boldsymbol{r_k}$. With this, we can render a volume around the sampled point to be coherent with the CLIP property of being image-aligned instead of pixel-aligned. For each ray $\boldsymbol{r_k}$, we select a random initial scale s_{img}. This scale is used as basis for each sampled point along $\boldsymbol{r_k}$, so analogous to [12] we define a function $s_k(t)$ to increase the scale proportionally to the focal length f_y, the image height h_{img} and the distance sampled t from the ray origin, as follows:

$$s_k(t) = \frac{s_{img} h_{img} t}{f_y} \tag{4.6}$$

CLIP embeddings supervision is done comparing the predicted image language embedding with pre-extracted ground truth CLIP embeddings. Following [12], we supervise each predicted CLIP embedding for each pixel with a ground truth CLIP embedding computed for a crop of the rendered image, since CLIP features are image-aligned. In this case, we select an image patch of size s_{img} centered at the image pixel associated with the sampled ray. In order to train the model to be precise at different scales, we compute an image pyramid, dividing the image in different crop sizes using scales that vary from s_{min} to s_{max}. Then, we store the CLIP embeddings for these crops at each pyramid level. During training, the random scale s_{img} selected for each ray lies in range (s_{min}, s_{max}), so we compare the predicted language embedding for the corresponding pixel ϕ_u with the ground truth language embedding ϕ_u^{gt} for that pyramid level. Since the pixel of the ray we are sampling could not lie in the center of a crop in that pyramid level, we perform trilinear interpolation between the embeddings from the four nearest crops for the scale above and below in the pyramid to compute ϕ_u^{gt}, as done in [12].

As explained in Section 4.1, our model baseline has shared layers MLP^s for background and transient networks. In addition, each one of the three networks employ different input data, so we keep consistent with that for computing the CLIP embeddings. For each stream network from the baseline, the final output is fed into different heads for predicting the color, volume density and uncertainty. Analogously, this output is used as input for the CLIP MLP, that predicts the language-image embedding centered on each pixel of the image. However, since the fine model samples a large amount of points per ray, we only predict CLIP embeddings for a subset of best samples,

in order to reduce the memory usage and focus on the occupied volumes of the scene. This selection is done using the volume density predicted for each sample through the fine model. For computing the final CLIP embedding over each ray for the corresponding pixel we use Eq. 2.4, substituting $\boldsymbol{c_i}$ by F_{lang_i}, so we calculate the embeddings using the same volumetric rendering equation that we use for the color, volume density and uncertainty predictions in the baseline. By doing this for each of the three CLIP networks, we compute each pixel CLIP embedding for the background ϕ^b, foreground ϕ^f and actor ϕ^a. Finally, we sum up these contributions and normalize it to obtain the CLIP embedding prediction ϕ_u for each pixel u of the frame.

4.2.2 Semantics regularization with DINO

Predicting CLIP embeddings through our three different MLPs for each sampled point is not enough to obtain good results in terms of relevancy maps when querying text. This is because CLIP embeddings are image-aligned instead of pixel-aligned. As proposed in [12], we add a DINO MLP (using DINOv2) to distill 2D features from the input images into the 3D field of each one of our baseline MLPs, analogously to our CLIP implementation. In the same way than in Section 4.2.1, we propose an extra MLP for predicting DINO embeddings $F_{dino}(g_t\boldsymbol{r_k}) \in \mathbb{R}^d$ for each sampled 3D point along a ray $\boldsymbol{r_k}$, optimized through the fine model. The objective of this prediction is regularizing the CLIP embedding predictions, since DINO features are pixel-aligned. This characteristic makes the DINO network to not need a scale as input, by contrast to our CLIP networks. As explained before, these MLPs takes the output from the geometric appearance MLPs as input, predicting a DINO feature embedding for each sampled point, using only the best samples. In order to compute the final DINO embedding over each ray for the corresponding pixel we use Eq. 2.4, substituting $\boldsymbol{c_i}$ by F_{dino_i}, in the same way we compute the CLIP embeddings. By doing this for each of the three DINO networks, we compute each pixel DINO embedding for the background ψ^b, foreground ψ^f and actor ψ^a. Alike CLIP embeddings, we sum up these contributions and normalize it to obtain the DINO embedding prediction ψ_u for each pixel u of the frame. Since DINO features are pixel-aligned we can supervise the predictions comparing them directly with the ground truth embeddings ψ_u^{gt}.

With this implementation, we are substituting the hash encoding for the 3D points used in [12] by the encoding done by early layers of the main MLPs. Another important difference with respect to [12] is that we are sharing the weights of the neural renderer layers with the networks that predict CLIP and DINO embeddings, meanwhile in the original implementation there are

no shared weights from the layers that predict the scene appearance and the language embedding field.

Language loss function. Language embeddings supervision is done using both CLIP and DINO features. Following [12] we implement the language embedding loss $\mathcal{L}_{lang}$ as:

$$\mathcal{L}_{lang} = \mathcal{L}_{dino} + \mathcal{L}_{clip} \tag{4.7}$$

In the case of DINO loss $\mathcal{L}_{dino}$, we calculate the function as a MSE between the predicted DINO embeddings ψ_u and ground truth DINO embeddings ψ_u^{gt} of each pixel u along a frame with M pixels, as:

$$\mathcal{L}_{dino} = \frac{1}{M}\sum_{u=1}^{M}(\psi_u - \psi_u^{gt})^2 \tag{4.8}$$

By contrast to $\mathcal{L}_{dino}$, CLIP loss $\mathcal{L}_{clip}$ is calculated using Huber loss $\mathcal{L}_{Huber}$, which penalizes less the outliers. This loss is defined as follows:

$$\mathcal{L}_{Huber}(x, y, \delta) = \begin{cases} 0.5(x-y)^2, & \text{if } |x-y| < \delta) \\ \delta(|x-y| - 0.5\delta), & \text{Otherwise} \end{cases}, \tag{4.9}$$

where x is the predicted value, y is the target value and δ is a parameter set empirically to change the behavior of the function.

Then, $\mathcal{L}_{clip}$ is calculated by averaging the Huber loss ($\delta = 1.25$) between the predicted CLIP embeddings ϕ_u and ground truth CLIP embeddings ϕ_u^{gt} of each pixel u along a frame with M pixels, as:

$$\mathcal{L}_{clip} = \frac{1}{M}\sum_{u=1}^{M}\mathcal{L}_{Huber}(\phi_u, \phi_u^{gt}, \delta) \tag{4.10}$$

By summing up the language loss together with the photometric loss we compute the total loss function of our model.

Language embeddings dimension reduction. CLIP and DINO embeddings are high-dimensional features, depending on the model used for computing them. The lowest output dimension for DINO features is 384, meanwhile that for CLIP embeddings, the minimum output dimension is 512. In order to reduce the GPU memory footprint, we reduce the dimensionality of the ground truth embeddings using Principal Component Analysis (PCA) [33], a widely used method in machine learning to convert high-dimensional data into low-dimensional data without losing relevant information.

DINO embeddings are computed using a ViT, so we can just perform PCA on the ground truth data to reduce the dimensionality and predict an output with the same dimension in each of the DINO MLPs. However, CLIP embeddings have two parts: a ViT for computing the image embeddings and a Transformer to calculate the text embeddings. Ground truth data refers to the image embeddings, which can be passed through a PCA to reduce the dimensionality. Nonetheless, text embeddings have the same dimension as the image embeddings so they can be related properly during zero-shot predictions. When we use DI-FF for doing object recognition from text queries, the text embeddings still have the original dimension even if we predict the image embeddings with a lower dimension to accomplish for the PCA reduced ground truth. In order to leverage the output dimension of the text embeddings to the dimension of the image embeddings we save the weights used by the PCA algorithm during the ground truth dimensionality reduction and apply them to the text embeddings during inference.

Chapter 5

Experimental procedure

In this Chapter we present the experimental setup used for the conducted experiments, as well as the different tests done. In Section 5.1 we expose the experiment settings for our model. Then, in Section 5.2 we introduce the dataset used for training and evaluation. Sections 5.3 and 5.4 are dedicated to explain the experiments performed, meanwhile Section 5.5 is focused on the metrics used for evaluating the results.

5.1 Implementation details

First, we introduce the standard setup we use for all our experiments. We run all the experiments with an NVIDIA RTX 4090 GPU using images of 256x456 pixels from different EPIC-KITCHENS [5] dataset scenes that we downsample to 128x228 pixels. Each scene is split into training, validation and test sets, representing around 88%, 6% and 6% of the total number of frames, respectively. We use a batch size of 1024 rays, sampling 64 points along each ray in the coarse model and 64 additional points in the fine model. CLIP embeddings are calculated using the best 32 samples from the fine model. For the neural renderer, the layers of the MLPs have 128 units, except for the shared MLP between the static model and the transient model, which has 256 units. The number of layers for the MLPs are 8, 1, 4 and 4, for MLP^s, MLP^b, MLP^f and MLP^a, respectively. The transient encoding for both foreground and actor models is the same, such that $z_t^f = z_t^a \in \mathbb{R}^{17}$. The frame appearance latent code is a 48-dimensional embedding, such that $z_t^b \in \mathbb{R}^{48}$. The positional encoding for the 3D positions and viewing directions uses 10 and 4 frequencies, respectively. For the semantic heads, on top of the static, transient and actor models, we use 6 layers of 128 units for each CLIP network and 3 layers of 128 units for each DINO network before the

final layer of 64 units. The DINOv2 architecture used is ViT-S/14 trained on LVD-142M dataset, meanwhile the CLIP architecture used is ViT-B/16 trained on LAION-2B dataset. We reduce the dimension of CLIP and DINO ground truth data using PCA to 128 and 64 dimensions, respectively, and extract the data from the images at 256x456 pixels to get a better feature representation. CLIP ground truth image pyramid is computed using $s_{min} = 0.05$ and $s_{max} = 0.5$ in 7 levels. For training the models, we first train the neural renderer (without CLIP and DINO MLPs) for 10 epochs. Then, we train the model enabling CLIP and DINO MLPs during 3 more epochs, freezing the rest of layers during the first 5000 steps. Every model is trained using Adam optimizer [13] with an initial learning rate of 0.0005 using a cosine annealing schedule [19] to reduce it during the training.

5.2 EPIC-KITCHENS dataset

EPIC-KITCHENS [5] is an extensive dataset compound of several scenes of egocentric videos, offering information of people interacting with objects in kitchen environments. Additionally, this dataset is labeled with actions and object interactions annotations, as well as binary masks for background and dynamic components of the scene (See Figure 5.1), resulting in a very extended benchmark for different tasks such as action recognition, hand-object segmentation or single-object tracking. For our experiments, we select 10 scenes from the EPIC-Diff dataset [43], which is a subset of EPIC-KITCHENS. Each scene contains 900 calibrated frames in average, with known camera pose, representing 14 minutes of egocentric video with multiple viewpoints and manipulated objects.

Figure 5.1: **EPIC-Diff dataset.** Example of ground truth image (Left), dynamic object segmentation map (Center) and dynamic components binary mask (Right).

5.3 Neural rendering speed-up and performance

According to [45], the training of NeuralDiff lasts 24 hours using an NVIDIA Tesla P40 GPU, including fine-tuning. Since this model serves as baseline for our project, we aim to reduce the training time, and also improve the results obtained. As described in [42], they propose substituting the positional encoding of the input coordinates by hash encoding [24] for the 3D points and spherical harmonics [4] for the viewing directions to increase the convergence speed and performance. These encodings are implemented within a framework called Tiny CUDA Neural Networks (TCNN) [22], which contains several tools for training and querying neural networks. This framework is implemented in C++ and presents also a fast MLP that works parallelizing the forward pass through the net.

Moving on to the experiment, we substitute the standard MLPs for the static, transient and person predictions by TCNN fast MLPs. In addition, we remove the positional encoding of the inputs and add a hash encoding for the 3D points and spherical harmonics encoding for the viewing directions. Hash encoding is done using a hashgrid of 16 layers from a resolution that goes from 16 to 512, and a hash table size of 2^{19} and feature dimension of 2. For the spherical harmonics we use 4 levels to compute the encoding. We train the neural renderer standalone (without semantic embedding networks) and compare it to the baseline model from NeuralDiff [43] that employs standard PyTorch MLPs and positional encoding.

5.4 DI-FF language embedding field ablation

In order to evaluate our model in tasks such as dynamic object segmentation or affordance segmentation, we propose two different architectures for the language embeddings predictions. The first one follows the architecture proposed in [12], which has a single semantic MLP, composed by a CLIP MLP and a DINO MLP, using as input a set of best sampled 3D points along the ray using the predicted volume density. This inputs are encoded using hash encoding with a hashgrid of 32 layers from a resolution that goes from 16 to 512, a hash table size of 2^{21} and feature dimension of 8. The second proposal is our model DI-FF, using three different semantic heads, one for static, transient and person components, respectively, and using as encoding an intermediate output of each one of the static, transient and person MLPs.

5.5 Evaluation metrics

We evaluate our experiments using four different metrics. For the speed-up of the model training, we compare the training speed in terms of iterations per second, being an iteration the total steps from getting a batch of data until computing the loss after a forward pass through the model. To evaluate if changing the input encodings leads to a better performance, we compare the scene prediction using PSNR (Peak Signal to Noise Ratio) and mAP (Mean Average Precision) metrics, following [43]. PSNR is used to evaluate the whole geometry reconstruction, meanwhile that mAP evaluates the dynamic components reconstruction using annotated binary masks for transient and actor components. Finally, for dynamic object segmentation we use mIoU (mean Intersection over Union).

Geometry reconstruction. PSNR is a classic metric used to compare the fidelity of an image in a pixel-wise way. The metric is calculated as:

$$PSNR(pred, gt) = -10log_{10}(MSE(pred, gt)), \tag{5.1}$$

where $pred$ is the predicted image, gt is the ground truth image and MSE is the Mean Squared Error between the predicted image and the ground truth image, calculated as:

$$MSE(pred, gt) = \frac{1}{wh}\sum_{i=0}^{h-1}\sum_{j=0}^{w-1}(pred(i,j) - gt(i,j))^2, \tag{5.2}$$

where w is the image width and h is the image height.

Dynamic components reconstruction. Average Precision (AP) is a common evaluation metric in object detection and segmentation tasks. It is calculated as the value under the curve Precision-Recall, which is a curve of precision values calculated along every recall level. Precision and recall are metrics calculated out of three variables associated to the model prediction for a specific class:

- True Positive (TP): The model prediction for a class is correct.
- False Positive (FP): The model prediction for a class is incorrect.
- False Negative (FN): The model does not predict a class instance that belongs to the ground truth.

Precision represents the percentage of correct predictions over the total number of predictions done for a specific class, computed as:

$$Precision = \frac{TP}{TP + FP} \tag{5.3}$$

Recall makes reference to the percentage of correct predictions for a class over the total samples of that class, calculated as:

$$Recall = \frac{TP}{TP + FN} \tag{5.4}$$

By applying this method, we can compute the AP for each class i we are predicting, and by computing the average AP over all classes M we can compute the mAP as follows:

$$mAP = \frac{1}{M} \sum_{i=1}^{M} AP_i \tag{5.5}$$

Moving on to our model, we can treat the background color and transient (foreground and actor) color predictions as two different binary masks. Then, we compare the transient prediction mask with respect to a ground truth mask that represent the pixels that belongs to transient parts to compute the AP of the transient component. By computing the transient AP over all the scenes of the dataset we can compute the mAP for the transient prediction.

Dynamic object segmentation. We compute the mIoU between the predicted dynamic object segmentation maps and the ground truth segmentation maps. Intersection over Union (IoU) is a widely used metric in object detection and object segmentation tasks. It consists on calculating the number of True Positives (TP), False Positives (FP) and False Negatives (FN) for each segmented object. These values are commonly represented in a Confusion Matrix, which is a NxN matrix with N object classes. This matrix represents the ground truth labels in the rows axis and the predicted labels in the columns axis, so the values in the diagonal represents the TP for each class, the sum of values in each row (not counting the diagonal value) represents the FP for each class and the sum of values in each column (not counting the diagonal value) represents the FN for each class. IoU for each possible label is calculated as:

$$IoU = \frac{TP}{TP + FP + FN} \tag{5.6}$$

The mIoU value is calculated as the average IoU value for each class i over the total number of possible labels N as follows:

$$mIoU = \frac{1}{N} \sum_{i=1}^{N} IoU_i \tag{5.7}$$

In order to obtain a segmentation map for each class, we compute a relevancy map for each text query. This relevancy map is calculated as the combination of the relevancy score of all pixels in the frame, following [12] implementation. Relevancy score is calculated for each pixel by computing the CLIP text embedding ϕ_{query} for a given text query along with a set of canonical phrases ϕ^i_{canon}, which are standard queries that could be done, such as *"thing"* or *"stuff"*. We compute the relevancy score RS for a CLIP image embedding prediction ϕ_{lang} as:

$$RS = min_i \frac{exp(\phi_{lang}\phi_{query})}{exp(\phi_{lang}\phi^i_{canon}) + exp(\phi_{lang}\phi_{query})} \tag{5.8}$$

For rendering the language embedding of a text query during inference, we select a range of scales between 0 and 2 meters with 30 increments, as done in [12]. By computing the relevancy score for all pixels with that query, we select the scale that yields the higher relevancy score and re-compute the relevancy score of all pixels using that best scale. The relevancy map of the text query is normalized in range $[-1, 1]$ for a better representation. By selecting a threshold, we keep the most relevant pixels of the map, generating with them the segmentation map for that query.

Chapter 6

Results

In this Chapter we analyse the results obtained for the explained experiments and perform a qualitative and quantitative evaluation of our model performance in dynamic object segmentation and affordance segmentation tasks.

6.1 Neural renderer evaluation

We start evaluating the results for the NeuralDiff pipeline ablation described in Section 5.3.

6.1.1 Scene prediction performance

Since we aim to test the performance in terms of scene prediction when using hash and spherical harmonics (SH) encoding instead of positional encoding, we train the model with standard PyTorch modules but changing the encoding pipeline. In Table 6.1 we can see that the results obtained with positional encoding are far better than the ones yield by hash encoding and spherical harmonics encoding. Both PSNR and mAP values are quite better when using positional encoding, representing an improvement of +9.8% and +14.7%, respectively, demonstrating that hash encoding and spherical harmonics usage is not efficient in our model. This could be because of the complexity of the neural network, since we have a triple MLP for the different parts of the scene. This could affect to the improvement that hashgrid and spherical harmonics should aport. Furthermore, our model is based on the original NeRF [21] model, by contrast to [42], which uses a whole different pipeline in addition to hash encoding and spherical harmonics. The difference in the results could be also related to the different architecture, so these encodings

lead to a better performance when used with the model proposed in [42].

Encoding Method	mPSNR	mAP
Hash + SH	20.55	60.68
Positional	**22.56**	**69.58**

Table 6.1: **Scene prediction results by NeuralDiff baseline.** We present the mAP for foreground segmentation and mPSNR of the full scene over the 10 scenes of EPIC-Diff dataset, comparing the results using positional encoding or hash and spherical harmonics encoding for the model inputs.

We can observe in Figure 6.1 that predictions done when using positional encoding represent better the foreground and actor components of the scene. When using hash and spherical harmonics encodings, the background seems clearer in general but transient objects appear more blurred and sometimes together with strange artifacts, since the model is not learning correctly. Since hash encoding provides a very large amount of learnable parameters, the complexity of the model with three different MLPs can lead to overfitting.

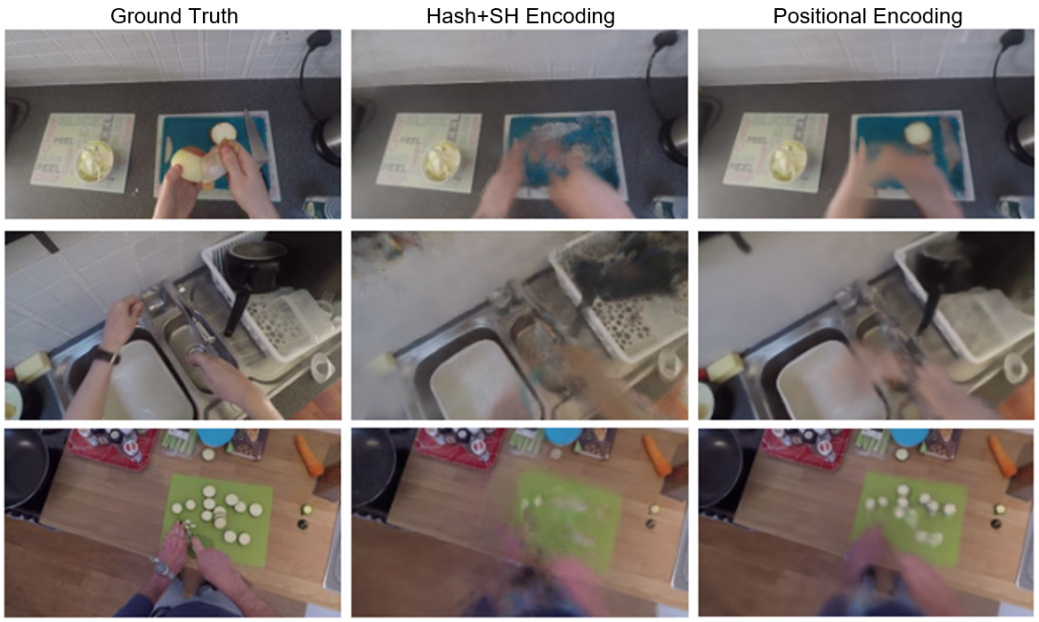

Figure 6.1: **Qualitative scene prediction results.** Comparison between using positional encoding and hash and spherical harmonics encoding in NeuralDiff baseline. Positional encoding captures better the transient objects and movement, having less artifacts.

6.1.2 Convergence speed

When changing the main PyTorch MLPs of the architecture by TCNN MLPs we appreciate loss instability, so the model loss becomes infinity after a few epochs of training with the standard training settings. Probably, this is due to the TCNN module behavior, leading to exploding gradient. This idea is built upon the fact that TCNN modules work with half precision (16-bit) floating point, meanwhile PyTorch modules work with double precision (32-bit). Since the model pipeline works with double precision, when mixing the architecture using half precision (TCNN MLPs) we need to cast several times the intermediate results from 16-bit precision to 32-bit precision and vice-versa. This can lead to information loss and divisions by very small values during gradient computation in the back-propagation algorithm, producing exploding gradient issues. In order to avoid this behavior, we test training the model adding weight decay to the optimizer. This weight decay acts as a penalty to the loss function, avoiding large weights values, preventing a growth out of control and avoiding exploding gradient. However, even though the weight decay works fine, the model still diverges to infinity after half of the training is complete.

Instability issues apart, we still compare the training speed of the two models. We can see in Figure 6.2 that using TCNN MLPs leads to a faster training speed overall, being the largest difference when using a batch size of 1024 rays (20 vs 10 iterations/s). However, since the training is quite unstable as explained before, we decide to not use TCNN fast MLP and keep PyTorch MLP in our implementation.

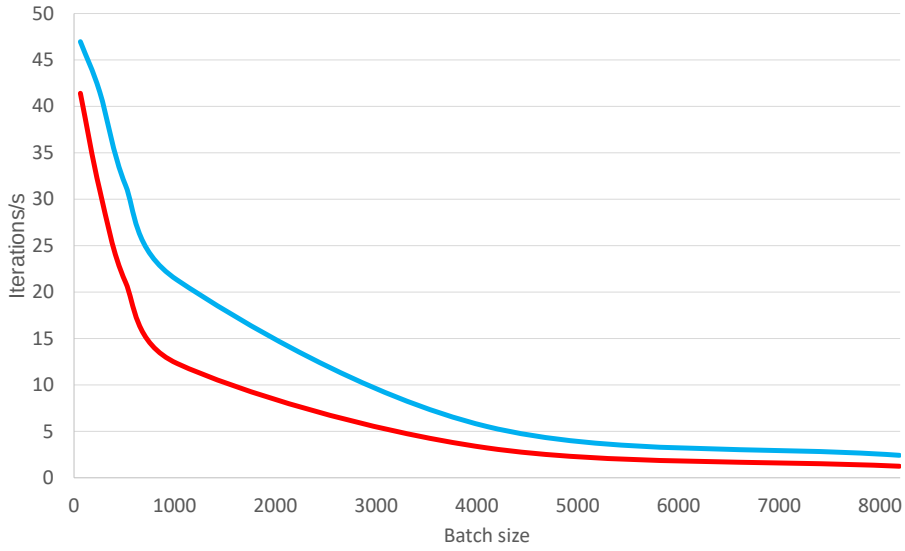

Figure 6.2: **Training speed results.** We present a comparison in terms of iterations per second with respect to the batch size when using TCNN fast MLP (blue) and PyTorch MLP (red) in NeuralDiff baseline.

6.2 Semantic understanding evaluation

After testing our baseline and deciding the best method to keep to do the training, we move forward to evaluating the results of our DI-FF model in semantic understanding for different tasks, as described in Section 5.4.

6.2.1 Dynamic object segmentation

In order to study dynamic object segmentation task, we take some test frames of each scene and predict the image output together with the language embedding ϕ_{img} for each text query. The text queries used for each scene are the ones of the dynamic objects that appear on those frames, which are labeled with its corresponding segmentation mask. The language embeddings are used to compute the relevancy score of that query according to Eq. 5.8, using as canonical phrases ϕ^i_{canon}: *"object", "things", "stuff", "texture"* and *"hands"*, following [12] implementation. Since we have an actor network that predicts person's hands we add this part of the body as canonical phrase, in order to reduce relevancy maps of occluded objects extending through the hands. With this relevancy score for each query, we compute a segmentation mask for each object by getting each pixel in which the relevancy score is

greater than 0.58, which is set empirically in the validation set.

In Table 6.2 we report the results for mIoU in different scenes comparing our model DI-FF that uses a triple MLP for semantic predictions with the pipeline used in LERF [12], with a single language embedding MLP (NeuralDiff+LERF). We observe that DI-FF (19.78 mIoU) outperforms NeuralDiff+LERF (12.65 mIoU), obtaining an improvement of +56.5%. This higher performance relies on the fact of having three different MLPs for grounding language embeddings, one for each part of the scene (background, foreground, actor), which allows a better understanding of the environment, obtaining sharper and more defined boundaries within the retrieved objects, as shown in Figure 6.3. Having a better relevancy map for the given text queries impact directly on the mIoU value. Since DI-FF relevancy maps are more focused on each query, it increases the number of true positives and decrease the number of false positives, which leads to a higher mIoU than the achieved by NeuralDiff+LERF implementation.

Scene	NeuralDiff + LERF	DI-FF
P01_01	18.84	**26.9**
P03_04	11.58	**21.7**
P04_01	7.34	**18.3**
P05_01	6.57	**16.8**
P06_03	14.04	**18.1**
P08_01	11.50	**24.9**
P09_02	14.15	**17.3**
P13_03	10.75	**12.3**
P16_01	15.13	**17.9**
P21_01	17.15	**23.6**
Average mIoU	12.65	**19.78** (+56.5%)

Table 6.2: **Dynamic Object Segmentation by CLIP image-language feature field.** Comparison between NeuralDiff+LERF pipeline and DI-FF pipeline. We compute relative improvement against NeuralDiff+LERF method. Refer to Appendix A to see the text queries used.

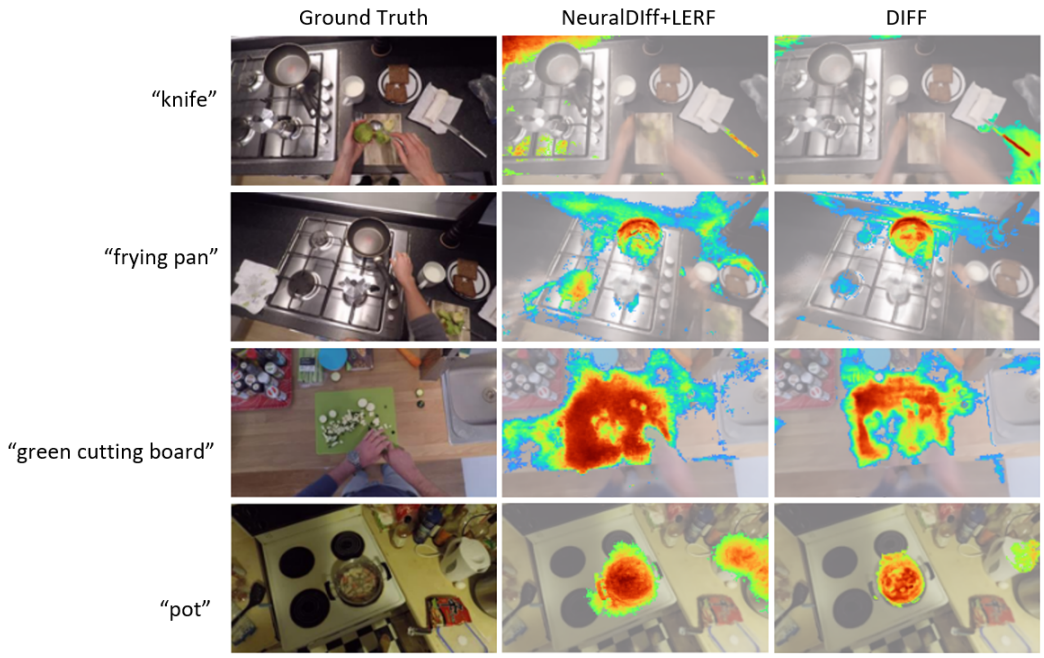

Figure 6.3: **Dynamic object relevancy maps in novel views.** Comparison between NeuralDiff+LERF and DI-FF models dynamic object relevancy maps with different queries in novel views. DI-FF predicts more accurate relevancy maps, focused on the target query.

We can see that DI-FF differentiates better the dynamic objects of the scene. NeuralDiff+LERF pipeline gets correct relevancy maps pointing to the object of interest but it also generates high relevancy scores around objects that do not correspond to the text query. By contrast, DI-FF relevancy maps are more accurate to the queried texts, even though it points out to incorrect objects sometimes. Since our model renders CLIP embeddings along three different MLPs for background, foreground and actor components, it is able to get a better understanding of the scene. In addition, the language embedding layers are trained sharing weights with the layers that predict the scene geometry, so both semantic and geometric information are connected. On the other hand, NeuralDiff+LERF pipeline only renders a single batch of language embeddings, without any geometric differentiation. The language embedding layers are also trained without sharing weights with the neural renderer layers, so we can clearly observe that our method DI-FF outperforms NeuralDIff+LERF method.

Regarding the predictions for some text queries, we can see that the relevancy maps are not always good, since DI-FF miss-predicts some queries or points out to multiple similar objects that are not the same, such as *”pot”*

and *"kettle"*. We appreciate in these cases that DI-FF struggles with objects that are semantically similar, but the relevancy score is still higher on the target query than on the other similar objects. In addition, long-tail queries often lead to a low quality relevancy map, since CLIP embeddings work many times as a "bag-of-words" more than as a cohesive phrase. This behavior can be seen in queries like *"frying pan"* in which the relevancy map extends over the cooking (frying) zones apart from the pan itself, or *"green cutting board"* in which the relevancy score is high for some green vegetables and part of the table (or board). This behavior affects directly to the computed segmentation maps. In Figure 6.4 we observe that the segmentation maps computed out of the relevancy maps are not so precise, with strange artifacts due to the relevancy maps extending along more objects than it should. We can see, also, that DI-FF segmentation maps are quite more reliable than the ones from NeuralDIff+LERF, as discussed before.

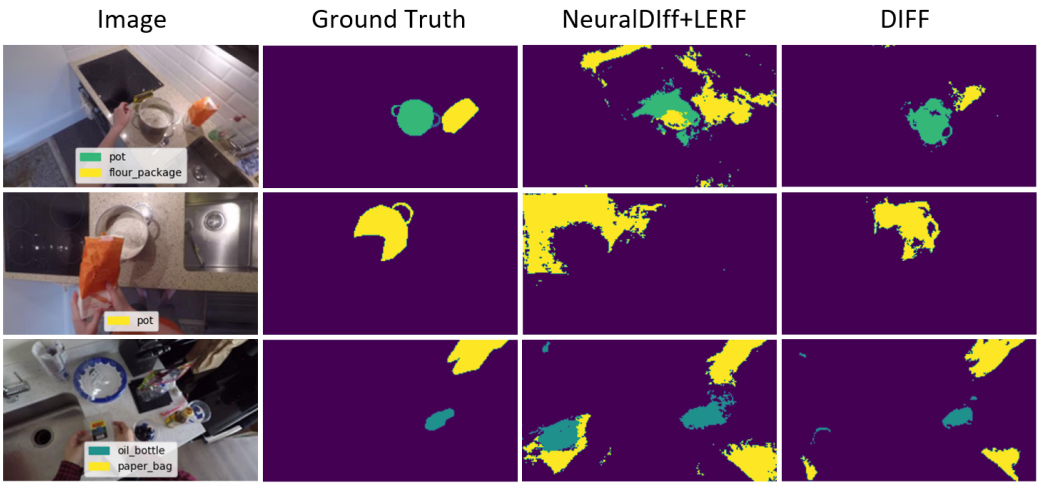

Figure 6.4: **Dynamic object segmentation maps in novel views.** Comparison between NeuralDiff+LERF and DI-FF models dynamic object segmentation with different queries in novel views. DI-FF segmentates better the objects, with less artifacts and wrong predictions.

We can also notice that some objects occluded behind the person's arms are not segmented correctly, as it happens with *"green cutting board"* query. The triple MLP stream we use in DI-FF allows us to remove the person contribution to the CLIP language embeddings, obtaining a better relevancy map of the occluded object, as represented in Figure 6.5.

Figure 6.5: **Amodal Scene Segmentation.** Comparison of relevancy map for *"green cutting board"* query when DI-FF predicts the three components of the scene and when DI-FF removes the actor component. We appreciate that the occluded object relevancy map is much better when removing the person.

Another key feature of our model is that it is able to retrieve queried objects that barely appears within the current frame, having full environment awareness of its position when the camera does not point directly to that item. In Figure 6.6 we can see some examples of this behavior.

Figure 6.6: **Surrounding Awareness.** DI-FF has information about surrounding objects in novel views, allowing segmentation of objects barely visible withing the image.

After discussing the results obtained and see that our model DI-FF performs better, we present in Figure 6.7 more relevancy maps from different text queries for both dynamic and static objects in the scene.

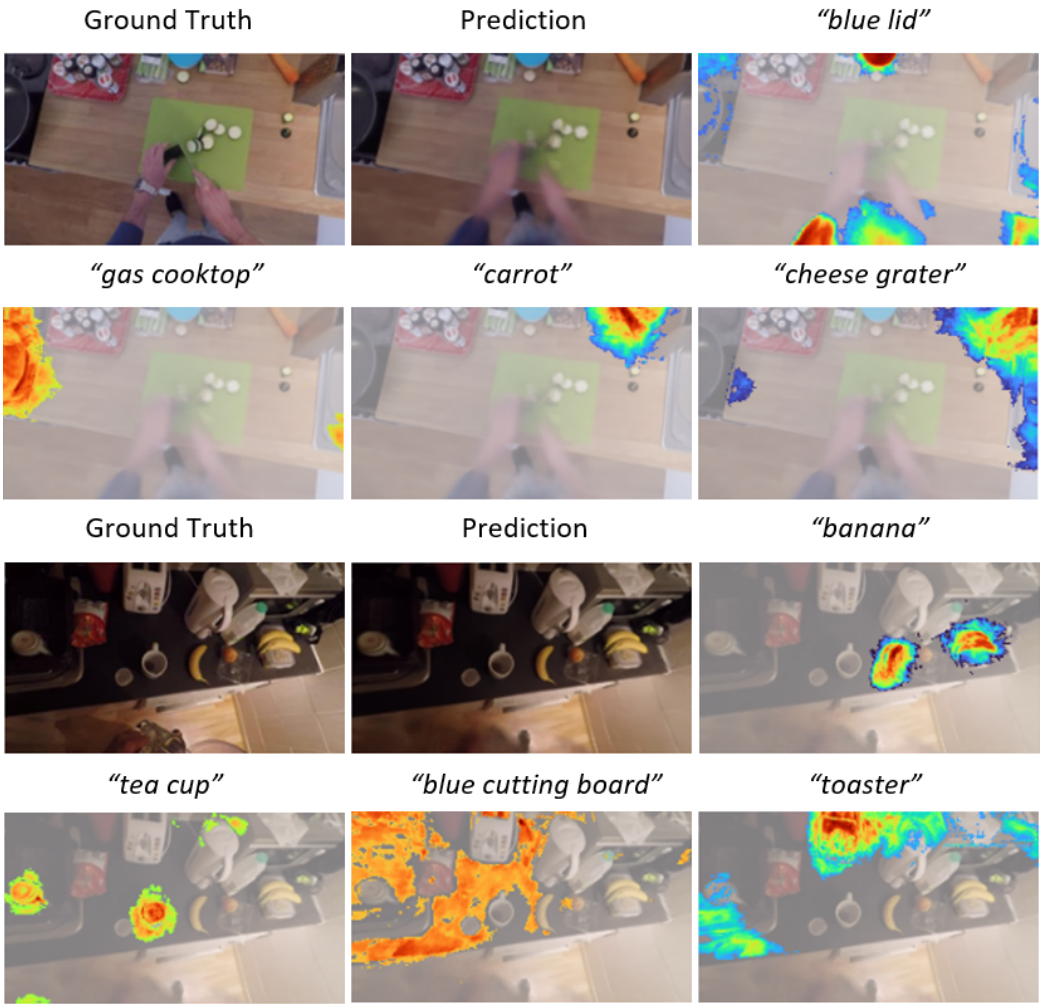

Figure 6.7: **DI-FF extra qualitative results.** Scene prediction and relevancy maps for dynamic and static object text queries. We observe very good results (*"banana"* or *"gas cooktop"*) but also failure cases (*"blue lid"* or *"blue cutting board"*).

6.2.2 Affordance segmentation

We also study the performance of DI-FF when querying actions instead of specific objects. In order to evaluate affordance (action) segmentation we query some actions and obtain the relevancy map for each one of them using our model DI-FF. Since we do not have ground truth action segmentation masks we only do a qualitative evaluation of this task. For computing the relevancy score for each action query we use as canonical phrases ϕ^i_{canon}: *"general task"*, *"indistinct movement"* and *"unclear action"*. We can see in

Figure 6.8 that our model struggles when querying complex actions that do not refer to a specific object. Due to the "bag-of-words" behavior from CLIP, querying large phrases that involve an abstract situation in the video, such as *"wash a kitchen utensil"* makes DI-FF segment poorly the scene, pointing to objects that could be understood as *"utensils"* or objects belonging to *"kitchen"* without establishing a connection within the entire query. This is due to the fact that CLIP is trained with object descriptions, not actions, so it is not able to generate correct relevancy maps for action text queries.

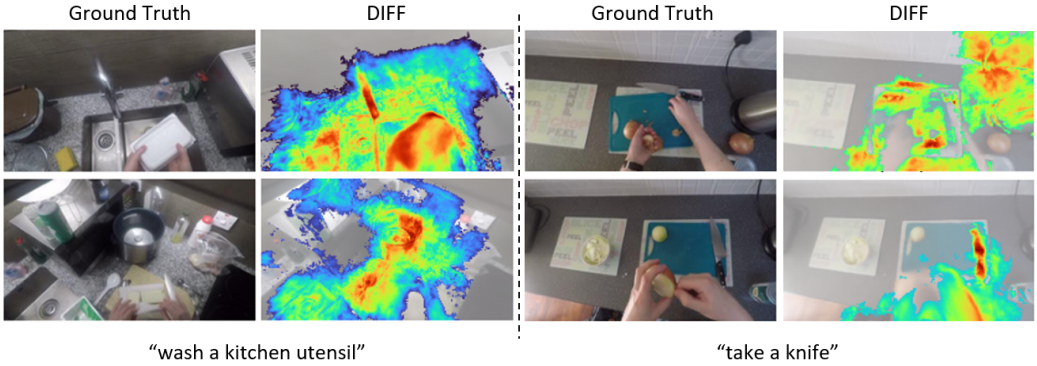

Figure 6.8: **Affordance relevancy maps in novel views.** We present qualitative results using DI-FF with different action queries. We notice that the model is unable to segment properly actions that point out to several objects or none in particular.

Chapter 7

Conclusions and future work

7.1 Conclusions

In this project we propose a novel NeRF-based model, DI-FF (Dynamic-Image Feature Fields), able to decompose an egocentric video into background, moving foreground and actor components that also enhances semantic understanding of the scene. The combination of these features makes our model able to perform dynamic object segmentation with certain robustness, being able to remove the actor presence to recognize occluded objects along the video. Furthermore, scene understanding allows DI-FF to retrieve objects that lie within the margins of the frame, supporting a surrounding understanding. Even though DIFF struggles when trying to identify actions that not necessarily point to an specific object or when querying long-tail object descriptions, it is a good baseline for future work towards getting a NeRF-based model able to yield a better environment understanding in terms of possible actions and also specific objects.

7.2 Future work

Although the proposed model accomplishes the main objective of this project, as shown in the results, the model lacks on affordance understanding and object-action connection. In addition, its "bag-of-words" behavior leads to miss-predicting long-tail queries. Future work could expand over this field, trying to improve the performance of the model in these cases. As done in some of the state-of-the-art literature, we could extend the model adding language video-based embeddings alongside the language image embeddings that we already use. By adding the information belonging to the video features, the model could learn action-oriented context, being able to do

not only a better affordance segmentation but also a better dynamic object segmentation.

Bibliography

[1] Yash Bhalgat et al. "N2F2: Hierarchical Scene Understanding with Nested Neural Feature Fields". In: *arXiv preprint arXiv:2403.10997* (2024).

[2] Mathilde Caron et al. "Emerging properties in self-supervised vision transformers". In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2021, pp. 9650–9660.

[3] Vincent Cartillier et al. "Semantic mapnet: Building allocentric semantic maps and representations from egocentric views". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 2. 2021, pp. 964–972.

[4] Feng Dai and Yuan Xu. "Spherical Harmonics". In: *Springer Monographs in Mathematics* (Apr. 2013). DOI: 10.1007/978-1-4614-6660-4_1.

[5] Dima Damen et al. "Scaling Egocentric Vision: The EPIC-KITCHENS Dataset". In: *European Conference on Computer Vision (ECCV)*. 2018.

[6] Alexey Dosovitskiy et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *ArXiv* abs/2010.11929 (2020). URL: https://api.semanticscholar.org/CorpusID:225039882.

[7] Rohit Girdhar and Kristen Grauman. "Anticipative video transformer". In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2021, pp. 13505–13515.

[8] Rahul Goel et al. "Interactive Segmentation of Radiance Fields". In: *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2022), pp. 4201–4211. URL: https://api.semanticscholar.org/CorpusID:255186143.

[9] Qiao Gu et al. "Egolifter: Open-world 3d segmentation for egocentric perception". In: *European Conference on Computer Vision*. Springer. 2025, pp. 382–400.

[10] James Kajiya and Brian von herzen. "Ray Tracing Volume Densities". In: *ACM SIGGRAPH Computer Graphics* 18 (July 1984), pp. 165–174. DOI: 10.1145/964965.808594.

[11] Bernhard Kerbl et al. "3D Gaussian Splatting for Real-Time Radiance Field Rendering." In: *ACM Trans. Graph.* 42.4 (2023), pp. 139–1.

[12] Justin Kerr et al. "LERF: Language Embedded Radiance Fields". In: *International Conference on Computer Vision (ICCV)*. 2023.

[13] Diederik Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations* (2014).

[14] Alexander Kirillov et al. "Segment Anything". In: *2023 IEEE/CVF International Conference on Computer Vision (ICCV)* (2023), pp. 3992–4003. URL: https://api.semanticscholar.org/CorpusID:257952310.

[15] Sosuke Kobayashi, Eiichi Matsumoto, and Vincent Sitzmann. "Decomposing NeRF for editing via feature field distillation". In: *Proceedings of the 36th International Conference on Neural Information Processing Systems*. 2024.

[16] Zhengqi Li et al. "Neural Scene Flow Fields for Space-Time View Synthesis of Dynamic Scenes". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2021.

[17] Yiqing Liang et al. "Semantic Attention Flow Fields for Monocular Dynamic Scene Decomposition". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. Oct. 2023.

[18] Miao Liu et al. "Egocentric activity recognition and localization on a 3d map". In: *European Conference on Computer Vision*. Springer. 2022, pp. 621–638.

[19] Ilya Loshchilov and Frank Hutter. "SGDR: Stochastic Gradient Descent with Warm Restarts". In: *arXiv: Learning* (2016). URL: https://api.semanticscholar.org/CorpusID:14337532.

[20] Ricardo Martin-Brualla et al. "NeRF in the Wild: Neural Radiance Fields for Unconstrained Photo Collections". In: *CVPR*. 2021.

[21] Ben Mildenhall et al. "NeRF: representing scenes as neural radiance fields for view synthesis". In: *Communications of the ACM* 65 (Jan. 2022), pp. 99–106. DOI: 10.1145/3503250.

[22] Thomas Müller. *tiny-cuda-nn*. Version 1.7. Apr. 2021. URL: https://github.com/NVlabs/tiny-cuda-nn.

[23] Thomas Müller et al. “Instant neural graphics primitives with a multiresolution hash encoding”. In: *ACM Transactions on Graphics* 41.4 (July 2022), pp. 1–15. ISSN: 1557-7368. DOI: 10.1145/3528223.3530127. URL: http://dx.doi.org/10.1145/3528223.3530127.

[24] Thomas Müller et al. “Instant neural graphics primitives with a multiresolution hash encoding”. In: *ACM Transactions on Graphics* 41.4 (July 2022), pp. 1–15. ISSN: 1557-7368. DOI: 10.1145/3528223.3530127. URL: http://dx.doi.org/10.1145/3528223.3530127.

[25] Lorenzo Mur-Labadia, Jose J Guerrero, and Ruben Martinez-Cantin. “Multi-label affordance mapping from egocentric vision”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2023, pp. 5238–5249.

[26] Lorenzo Mur-Labadia et al. “AFF-ttention! Affordances and Attention models for Short-Term Object Interaction Anticipation”. In: *arXiv preprint arXiv:2406.01194* (2024).

[27] Tushar Nagarajan et al. “Ego-topo: Environment affordances from egocentric video”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 163–172.

[28] Tushar Nagarajan et al. “EgoEnv: Human-centric environment representations from egocentric video”. In: *NeurIPS*. 2023.

[29] Tushar Nagarajan et al. “EgoEnv: Human-centric environment representations from egocentric video”. In: *Advances in Neural Information Processing Systems* 36 (2024).

[30] Maxime Oquab et al. *DINOv2: Learning Robust Visual Features without Supervision*. 2023.

[31] Hyun Soo Park et al. “Egocentric future localization”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 4697–4705.

[32] Chiara Plizzari et al. “Spatial Cognition from Egocentric Video: Out of Sight, Not Out of Mind”. In: *arXiv preprint arXiv:2404.05072* (2024).

[33] “Principal component analysis”. In: *Chemometrics and Intelligent Laboratory Systems* 2.1 (1987). Proceedings of the Multivariate Statistical Workshop for Geologists and Geochemists, pp. 37–52. ISSN: 0169-7439. DOI: https://doi.org/10.1016/0169-7439(87)80084-9. URL: https://www.sciencedirect.com/science/article/pii/0169743987800849.

[34] Albert Pumarola et al. “D-NeRF: Neural Radiance Fields for Dynamic Scenes”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020.

[35] Minghan Qin et al. “Langsplat: 3d language gaussian splatting”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2024, pp. 20051–20060.

[36] Alec Radford et al. “Learning Transferable Visual Models From Natural Language Supervision”. In: *International Conference on Machine Learning*. 2021. URL: https://api.semanticscholar.org/CorpusID:231591445.

[37] Nasim Rahaman et al. “On the Spectral Bias of Neural Networks”. In: *International Conference on Machine Learning*. 2018. URL: https://api.semanticscholar.org/CorpusID:53012119.

[38] Santhosh Kumar Ramakrishnan and Tushar Nagarajan. “Environment predictive coding for visual navigation”. In: *ICLR 2022* (2022).

[39] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[40] Changpeng Shao. “A Quantum Model for Multilayer Perceptron”. In: *arXiv: Quantum Physics* (2018). URL: https://api.semanticscholar.org/CorpusID:119427804.

[41] William Shen et al. “Distilled Feature Fields Enable Few-Shot Language-Guided Manipulation”. In: *7th Annual Conference on Robot Learning*. 2023. URL: https://openreview.net/forum?id=Rb0nGIt_kh5.

[42] Matthew Tancik et al. “Nerfstudio: A Modular Framework for Neural Radiance Field Development”. In: *Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Proceedings*. 2023. DOI: 10.1145/3588432.3591516. URL: http://dx.doi.org/10.1145/3588432.3591516.

[43] Vadim Tschernezki, Diane Larlus, and Andrea Vedaldi. “NeuralDiff: Segmenting 3D objects that move in egocentric videos”. In: *2021 International Conference on 3D Vision (3DV)*. IEEE. 2021, pp. 910–919.

[44] Vadim Tschernezki et al. “Epic fields: Marrying 3d geometry and video understanding”. In: *Advances in Neural Information Processing Systems* 36 (2024).

[45] Vadim Tschernezki et al. “Neural Feature Fusion Fields: 3D Distillation of Self-Supervised 2D Image Representations”. In: *Proceedings of the International Conference on 3D Vision (3DV)*. 2022.

[46] Ashish Vaswani et al. “Attention Is All You Need”. In: *Advances in neural information processing systems, pp. 5998–6008* (June 2017). DOI: 10.48550/arXiv.1706.03762.

[47] Abdelrhman Werby et al. “Hierarchical Open-Vocabulary 3D Scene Graphs for Language-Grounded Robot Navigation”. In: *First Workshop on Vision-Language Models for Navigation and Manipulation at ICRA 2024*. 2024.

List of Figures

List of Tables

Appendices

Appendix A

Dynamic object segmentation queries

Scene	Dynamic object text queries
P01_01	green cutting board, blue lid, cheese grater, saucepan, pot
P03_04	bowl, blue cutting board, knife, white pot
P04_01	spoon, transparent bottle, white bottle, pan
P05_01	cup, electric kettle, apple, banana, bottle of milk
P06_03	pot, flour package, jug, orange bag
P08_01	frying pan, cup, pepper cellar, cutting board, plate
P09_02	package, white cutting board, pan, cup, pan turner
P13_03	plate, blue colander, pesto jar, pasta, scissors
P16_01	pot, package, egg, knife protector, cutting board
P21_01	plate, package, oil bottle, blue plate, paper bag

Table A.1: Text queries used during the dynamic object segmentation evaluation experiments.

Appendix B

Project tools and management

B.1 Project tools

For the project realization, we use Python v3.9 as main programming language, with some tests done using a C++ framework called TCNN v1.7. Implementation of the model is done using PyTorch v1.13 with some other standard deep learning libraries. To speed-up the model we use both CUDA v11.7 and cuDNN v8.5 together with an NVIDIA RTX 4090 GPU available thanks to RoPeRT research group from the University of Zaragoza.

B.2 Project management

Table B.1 shows the estimated time spent in each of the project parts.

Task	Time Spent (h)
Literature review	90
NeuralDiff baseline understanding	60
TCNN implementation	60
NeuralDiff ablation and performance experiments	80
DIFF implementation	190
DIFF ablation and performance experiments	100
Meetings	30
Thesis Document	90
TOTAL	**700**

Table B.1: Project management.