Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Control Engineering and Information Technology

Multi-view object detection using NeRF and YOLO

Paper for Scientific Students' Associations' Conference

Written by
Nándor Kőfaragó

Supervisor
Dr. Márton Szemenyei

November 2, 2023

Abstract

Deep neural networks have been applied to computer vision tasks for a considerable amount of time. Object detection is one of the most important applications in this field, with close-to-human performance of the latest models. Despite these achievements, there are still less explored realms of object detection. Most state-of-the-art models use single-view inputs and often perform poorly with partially visible or completely occluded objects. This leaves room for further exploration towards multi-view models.

On the other hand, there have been advancements in representing complex 3-dimensional scenes with neural networks (Neural Radiance Fields or NeRFs [12]). Initially, hundreds of input images and days of training were needed to create a 3D representation of a single scene. With further enhancements, these tasks can now be solved by generalizing networks using very few images and requiring only short inference times after initial training. However, these new models have not yet been proven very useful in practical applications.

The purpose of my paper for the Scientific Students' Associations' Conference is to combine these two areas of study utilizing a NeRF model for object detection. The goal of my research is to use multiple views to enhance the performance of object detection, especially in cases where the view of objects is partially or fully obstructed, and multiple views can be useful to take advantage of supplementary information. I have applied this new approach to a specific task: to improve object detection at road intersections for self-driving cars (and driver assistance systems).

To achieve this goal, I am using the proposed method of a recent paper, called PixelNeRF [21]. This model was originally designed to render novel views of a 3D scene from a few input images in a solely feed-forward manner. The same model architecture is used and the RGB output is replaced with the input features of a YOLO [18] object detection layer. This way, the neural 3D representation can be fed directly into an object detection network layer, which can then supposedly predict object bounding boxes with higher accuracy, even for objects that are completely occluded from the original view direction. This way, the model is designed to learn the 3D positions and bounding boxes of objects instead of the RGB visual appearances.

I have created a custom synthetic dataset for the task (designed to contain a high number of occluded objects) using Blender, implemented the new model (combined from the pixelNeRF and YOLO architectures), trained the model and evaluated the results. Finally, I have compared the results with a current state-of-the-art YOLO object detection model.

Absztrakt

Hungarian abstract

Mély neurális hálózatokat már régóta alkalmaznak gépi látási feladatokra. E terület egyik legfontosabb alkalmazása az objektumdetekció, ahol a legújabb modellek már megközelítik az emberi teljesítményt. Az objektumdetekciónak azonban vannak még kevésbé vizsgált területei. A legtöbb korszerű modell egyetlen nézetből származó bemenetet használ, és gyakran rosszul teljesít részben vagy teljesen eltakart objektumok esetén. Ezen gyengeségek javítására még nagyrészt felfedezetlen terület a több nézetet felhasználó modellek alkalmazása.

Máshonnan közelítve a problémát számos eredmény született komplex 3 dimenziós jelenetek neurális hálózatokkal történő reprezentálásában, az úgynevezett Neural Radiance Field vagy NeRF modellekkel. Ezen modelleknek kezdetben több száz bemeneti nézetre és több napnyi tanításra volt szükségük egyetlen jelent reprezentálásához. További fejlesztésekkel ilyen feladatok ma már általánosításra képes hálózatokkal oldhatók meg, amelyek kevés bemeneti nézetet használnak, és a kezdeti betanítás után csupán rövid következtetési időt igényelnek. Az új modellek azonban még nem igazán bizonyultak hasznosnak gyakorlati alkalmazások tekintetében.

A Tudományos Diákköri Konferenciára szánt munkámban arra törekszem, hogy kombináljam ezt a két módszert. Kutatásom célja egy NeRF modellre épített objektumdetekciós neurális hálóval és több bemeneti nézet felhasználásával javítani az objektumdetektálás teljesítményét, különösen olyan esetekben, amikor az objektumok részben vagy teljesen takarásban vannak, és a több nézet hasznos információval szolgálhat. Ezt az új megközelítést egy konkrét feladatra alkalmaztam: önvezető autók (és vezetéstámogató rendszerek) esetén a közúti kereszteződésekben történő objektumfelismerés javítására.

A cél eléréséhez egy nemrég publikált PixelNeRF című cikkben bemutatott módszert használok. Ezt a modellt arra tervezték, hogy néhány bemeneti képből új nézeteket rendereljen egy 3D-s jelenetről, kizárólag feed-forward módon. Megoldásomban ugyanezt a modellarchitektúrát használom, és az RGB kimenetet egy YOLO objektumdetektáló réteggel helyettesítem. Így a neurális 3D reprezentáció beköthető egy objektumdetektáló rétegbe, amely így feltételezhetően pontosabban tudja predikálni az objektumok bounding boxait, még az eredeti nézeti irányból teljesen kitakart objektumok esetében is. A modell célja, hogy az RGB vizualizáció helyett az objektumok térbeli helyzetét és méretét tanulja meg.

A feladathoz Blenderben létrehoztam egy szintetikusan generált adathalmazt (ami nagyszámú takarásban lévő objektumot tartalmaz), implementáltam az új modellt (a PixelNeRF és a YOLO architektúrákból kombinálva), betanítottam a modellt és kiértékeltem az eredményeket. Végül összehasonlítottam az eredményeket a jelenlegi legkorszerűbb YOLO objektumfelismerő modellel.

Contents

Chapter 1

Introduction

Deep learning [4][8][13] has been a prominent research area in recent decades, with computer vision at its core. The increasing deployment of technologies, including autonomous driving and industry 4.0 and 5.0, has resulted in numerous intelligent systems, and it is unsurprising that with it, object detection has become one of the most significant computer vision fields. YOLO object detection models are known for their exceptional performance in this field, with significant upgrades almost every year, and they have succeeded in exceeding human performance in real-time processing. However, there are still shortcomings in this field due to environmental limitations causing objects to be partially or fully occluded, preventing current state-of-the-art models from detecting them, even though this would be crucial in everyday scenarios.

While a multitude of research papers are dedicated to various facets of object detection, there has been no previous study on the use of multiple input views for the detection of objects in occlusion. Most studies concentrate on 3D detections or enhancing model architectures for single-view. The aim of my research is to address this problem by proposing a new model for multi-view object detection. This new model is based on a Neural Radiance Field (NeRF) and YOLO implementation to exploit both 3D understanding of complex scenes and object detection. The novel pixelNeRF-YOLO model is designed to aid object detection for autonomous vehicles and self-driving cars in complex urban settings. I have implemented and evaluated the new model and compared the results to a state-of-the-art YOLO model.

With the success of this novel approach, self-driving cars can receive an additional layer of safety, that is not only the perfected version of current solutions but leverages data that has never been used before. This model could not only enable autonomous vehicles to achieve the human-level performance, but also allow intelligent systems to use information in ways that is not possible for human drivers. Moreover, expanding the range of applications for this model may include not only the automotive industry, but also medical imaging, manufacturing of goods or initiatives for the preservation of the environment, by allowing the detection of details that were previously impossible.

My contributions to this research area are the following:

- I have created a synthetic dataset using Blender to serve as a foundation for training and evaluating multi-view object detection models. The dataset contains RGB, depth and semantic segmentation images of urban road intersections with a high number of occluded objects. This dataset was utilized for my research but can be employed as groundwork for subsequent models.
- I have implemented a new model which significantly outperformed the state-of-the-art YOLOv7 [20] object detection model on our dataset.
- I have presented a comparative evaluation for multi-view and single-view object detection, highlighting possible application areas of both models.

The paper is structured in the following way. In chapter 2, I present a brief overview of the relevant literature in this field, followed by a detailed description of the models that were used to implement the new multi-view model. Chapter 3 describes how the dataset was created. In chapter 4, I describe the high-level architecture and all the design details of my work, including training details of the model and hyperparameter optimization. Later, in chapter 5, some implementation details are disclosed, highlighting less trivial parts of the overall work. The paper concludes with an evaluation of the new pixelNeRF-YOLO model, with comparisons to a state-of-the-art benchmark YOLOv7 model.

Chapter 2

Related work

2.1 Current methods and models

In this section, I will outline some of the current methods and models that are being applied to tasks similar to my project. This will provide a brief literature review of research progress in this area.

2.1.1 3D object detection

One literature review [14] concentrates on publications that attempt to execute 3D object recognition, that is, identifying objects from their 3D representations, such as models or 3D scans. Within the category of 3D object detection models, the review precisely narrows down those that apply multiple views for detection (and other related tasks such as pose estimation). Although the logic is similar to my model, which involves using multiple input views for object detection, the aim of these publications is not to achieve better results with occluded objects.

2.1.2 Improved YOLO object detection

A different paper [10] aims to improve YOLO object detection to perform better with small and occluded objects. Although this publication strives for the same objective, it does not employ multiple input views to incorporate additional information. This method attains both remarkable accuracy and speed by altering the backbone network of YOLO, adding an attention mechanism and formulating a different loss function.

2.1.3 Detection improved with segmentation

Another paper [3] focuses on the detection of occluded objects. A more robust model is created by augmenting the bounding box with a set of binary variables, each corresponding to a cell indicating whether the pixels in the cell belong to the object. This segmentation-aware representation explicitly models and accounts for the supporting pixels for the object within the bounding box. Although this work is also impressive, it still does not incorporate multi-view inputs for recognition.

2.1.4 NeRF-YOLO

In a recent paper [19][1] that I co-authored with my supervisor, a multi-view object detection model was implemented, trained and evaluated on the dataset created for this paper. The concept is to project YOLO features from multiple input views to 3D space using camera parameters and the depth map corresponding to the views. This analytical approach is very precise and a lightweight addition to a regular YOLO network. It is also an intuitive approach to project object detection features back to 3D, to enable understanding of scenes with occlusions. However, a drawback is that it requires an accurate depth map to work, which is not needed by the model proposed in this paper.

Although these works are impressive, only the NeRF-YOLO directly addresses using multiple input views to enhance object detection for occluded objects. This provides an opportunity for further research

[1]The paper has already been presented at ISMCR 2023, but at the time of writing the proceedings are not yet publicly available.

in this field, where the outcomes hold significant importance for contemporary applications. In the following sections, I will detail the related work that I used directly in my implementation to create a multi-view object detection model based on NeRF and YOLO.

2.2 Neural Radiance Fields

Understanding and representing 3D scenes with machine learning algorithms has been a long-standing problem. However, a paper from 2021 presented a solution: to represent a 3D scene with a 5D continuous function, which is to be learnt by the model for every individual scene. This is called a Neural Radiance Field representation [12]. This was originally devised to synthesize novel views from a sparse set of input views (Figure 2.1). This work served as a foundation for numerous papers, and the technique has gained enormous popularity. Although I am not planning to reconstruct entire 3D scenes or render novel views, this approach can also serve as a base to enrich the input of an object detection algorithm. In this chapter, I will summarize how NeRFs represent 3D scenes and how they work for view synthesis.

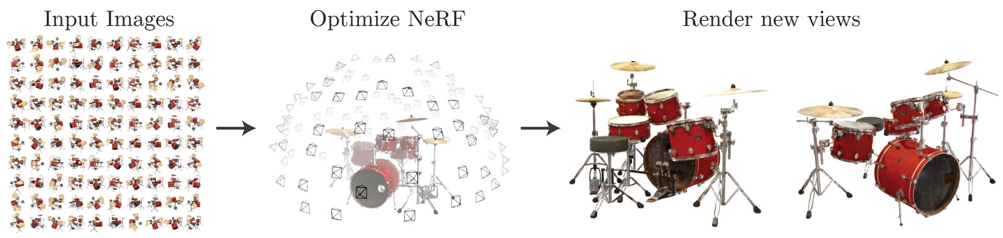


Figure 2.1: Original NeRF

2.2.1 NeRFs

NeRFs represent a continuous scene as a 5D vector-valued function (Figure 2.2) whose input is a 3D location $\boldsymbol{x} = (x, y, z)$ and a 2D viewing direction $\boldsymbol{d} = (\theta, \phi)$, and whose output is an emitted colour $\boldsymbol{c} = (r, g, b)$ and volume density σ. We can interpret this function as follows. For every continuous point in space, it assigns a colour (which is the visible colour of the object), and a density, which is the probability of that point absorbing a ray. This density is essentially the opacity of the object, otherwise 0 for “air”. This 5D function is not only location-dependent but also depends on viewing direction (Figure 2.3). This is to take special material surfaces into account, and correctly handle reflective and shiny surfaces. Therefore, the colour needs to depend on 3D spatial coordinates and the viewing (ray) direction.

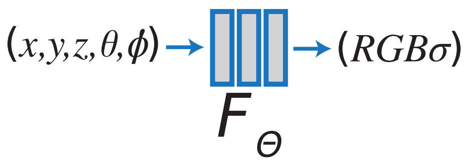


Figure 2.2: 5D function of NeRF

This continuous 5D scene representation is approximated with a fully connected MLP network $F_\Theta : (\boldsymbol{x}, \boldsymbol{d}) \longrightarrow (\boldsymbol{c}, \sigma)$ and its weights Θ are optimized to map from each input 5D coordinate to its corresponding volume density and directional emitted colour.

NeRFs encourage the representation to be multi-view consistent by restricting the network to predict the volume density σ as a function of only the location $\boldsymbol{x}$ while allowing the RGB colour $\boldsymbol{c}$ to be predicted as a function of both location and viewing direction. To accomplish this, the MLP F first processes the input 3D coordinate with fully connected layers and outputs the density and a feature vector. This feature vector is then concatenated with the camera ray's viewing direction and passed to one additional fully connected layer that outputs the view-dependent RGB colour.

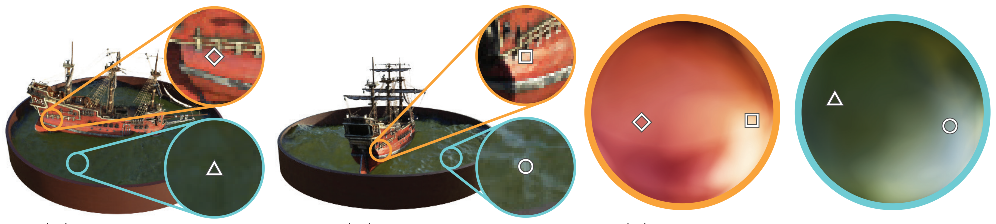

(a) View 1 (b) View 2 (c) Radiance Distributions

Figure 2.3: View direction dependence of colours

2.2.2 Volume rendering with radiance fields

The 5D neural radiance field represents a scene as the volume density and directional emitted radiance at any point in space. The colour of any ray passing through the scene is rendered using principles from classical volume rendering. The principles of volume rendering are known from computer graphics [7], but I will explain the part related to NeRFs.

The volume density σ can be interpreted as the differential probability of a ray terminating at an infinitesimal particle at location $\boldsymbol{x}$. The expected colour $C(\boldsymbol{r})$ of camera ray $\boldsymbol{r}(t) = \boldsymbol{o} + t\boldsymbol{d}$ with near and far bounds t_n and t_f is:

$$C(\boldsymbol{r}) = \int_{t_n}^{t_f} T(t)\sigma(\boldsymbol{r}(t))\boldsymbol{c}(\boldsymbol{r}(t), \boldsymbol{d})dt, \text{ where } T(t) = \exp\left(-\int_{t_n}^{t} \sigma(\boldsymbol{r}(s))ds\right) \tag{2.1}$$

The function $T(t)$ denotes the accumulated transmittance along the ray from tn to t, i.e., the probability that the ray travels from t_n to t without hitting any other particle. σ can be interpreted as the opacity dependent on the location and $\boldsymbol{c}$ is the colour dependent on the location and viewing direction.

This integral is the ideal representation of a single ray's colour. However, in practice, the integral is replaced by a finite sum of discrete queries to the radiance field (i.e. the NeRF MLP). Querying the MLP at fixed discrete locations would limit the resolution of the learnt function, therefore the ray is divided into N evenly spaced bins and one sample is drawn uniformly at random from each bin.

$$t_i \sim \mathcal{U}\left[t_n + \frac{i-1}{N}(t_f - t_n),\ \ t_n + \frac{i}{N}(t_f - t_n)\right] \tag{2.2}$$

The final integral is hence replaced by a finite discrete sum:

$$\hat{C}(\boldsymbol{r}) = \sum_{i=1}^{N} T_i(1 - \exp(-\sigma_i\delta_i))\boldsymbol{c}_i, \text{ where } T_i = \exp\left(-\sum_{j=i}^{i-1} \sigma_j\delta_j\right) \tag{2.3}$$

where $\delta_i = t_{i+1} - t_i$ is the distance between adjacent samples.

2.2.3 Training NeRFs

The above formula for rendering camera rays is differentiable. This enables the MLP to be trained using backpropagation. Instead of directly formulating a loss function for the outputs of the neural network, a loss is used for a sum of outputs. For every input view a ray is marched through each pixel and the ray colour is rendered with the method described above. The loss can then be determined by comparing the emitted colour of a ray with the ground truth colour of the given pixel (Figure 2.4).

2.2.4 Novel view synthesis with NeRF

To synthesise novel views from a NeRF model, the novel views' camera extrinsic and intrinsic parameters are needed. With known camera parameters, rays can be marched through 3D space. With discrete sampling along the rays, the 5D vectors are generated as inputs for the NeRF MLP. The outputs of RGB colour and density are then aggregated (rendered) along each ray, to produce the desired image. This makes it clear that the inputs for NeRFs are 5D vectors generated by camera rays through space and the outputs are colours and densities aggregated into output images.

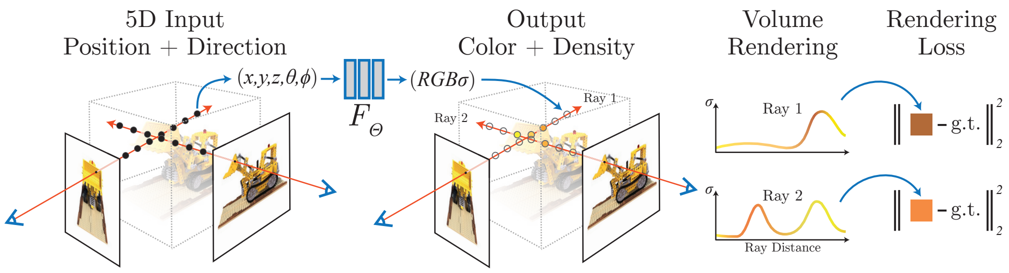


Figure 2.4: Training the NeRF model

2.2.5 Additional training details

The complexity of NeRF models requires some additional design details to make them capable of representing complex scenes. As these are not closely related to my field, I will just mention them with a brief reasoning for their use.

One of the techniques is called positional encoding, which is used to enable the network to represent high-frequency functions. Although neural networks are universal function approximators [6], they tend to be biased towards learning lower frequency functions (i.e., to represent only slow-changing details of the 3D scene) [15]. This can be problematic if we want to represent scenes with intricate details, like sharp edges (high frequencies).

The other technique is to use hierarchical volume sampling. Two networks (a coarse and a fine one) are trained together. The above-mentioned sampling technique is used to train the coarse network. Given the output of the coarse network, a more informed sampling is made to train the fine network. This makes the sampling locations biased towards relevant parts of the volume, enabling the fine network to capture finer details.

2.2.6 Evaluation of NeRFs

Neural radiance fields serve as a good foundation for representing 3D scenes with neural networks, which is beneficial for my research as well. However, regular NeRF models essentially overfit on a single scene and are only capable of rendering novel views for the scene that it was trained on (online retraining is required for each new scene). This is not useful for enhancing real-time object detection, as it is impossible to train a new network for every individual arising scene. In addition, there is no separate training and test time because the training is essentially the creation of the representation. The final downside of regular neural radiance fields is that they require hundreds of images to be trained on and an excessive amount of computing power and time. This also makes them impractical for object detection in real-time driver assistance systems.

2.3 PixelNeRF

A paper published in 2021, titled PixelNeRF [21] is meant to solve almost exactly those drawbacks of regular NeRFs which make them impractical for enhancing object detection. The creators of pixelNeRF set out to make neural radiance fields generalize from previously seen 3D scenes (pure offline training) and operate in a fully feed-forward manner on new scenes. This enables two significant advantages of this model: it requires far fewer input views for novel view synthesis, and the predictions for unseen scenes are orders of magnitude faster. In this chapter, I will introduce the pixelNeRF architecture and show how its approach enables it to be the perfect foundation for object detection.

2.3.1 Image conditioned NeRF

To overcome the NeRF representation's inability to share knowledge between scenes, pixelNeRF proposes an architecture to condition a NeRF on spatial image features. The model is comprised of two components: a fully convolutional image encoder, which encodes the input image into a pixel-aligned feature grid, and a NeRF network which outputs colour and density, given a spatial location and its corresponding encoded feature. The image encoder is a convolutional neural network, and it outputs the feature volume. This feature volume and the traditional inputs (3D coordinates and viewing direction) are fed into the NeRF network.

The model differs for single and multi-view cases because the features are extracted from a feature volume. Therefore, when there is only a single view, it can pass through simple a feed-forward network, but for multiple views, each input needs to be encoded and then processed through the network to finally give a better output for colour and density.

2.3.2 Single image pixelNeRF

Given an input image $\boldsymbol{I}$ of a scene, first the feature volume $\boldsymbol{W} = E(\boldsymbol{I})$ is extracted. Then, for a point on a camera ray $\boldsymbol{x}$, the corresponding image feature is retrieved by projecting $\boldsymbol{x}$ onto the image plane to the image coordinates $\pi(\boldsymbol{x})$ using known intrinsics, then bilinearly interpolating between the pixel-wise features to extract the feature vector $\boldsymbol{W}(\pi(\boldsymbol{x}))$ (see Figure 2.5). The image features are then passed into the NeRF network, along with the position and view direction (both in the input view coordinate system), as

$$f(\gamma(x), d; \boldsymbol{W}(\pi(x)) = (\sigma, \boldsymbol{c}) \tag{2.4}$$

where $\gamma(\cdot)$ is the positional encoding on $\boldsymbol{x}$, discussed in subsection 2.2.5 for NeRF models.

If the query view direction is closer to the input view orientation, the model can rely more directly on the input; if it is dissimilar, the model must leverage the learned prior. Moreover, in the multi-view case, view directions could serve as a signal for the relevance and positioning of different views. For this reason, view directions are inputs at the beginning of the NeRF network MLP.

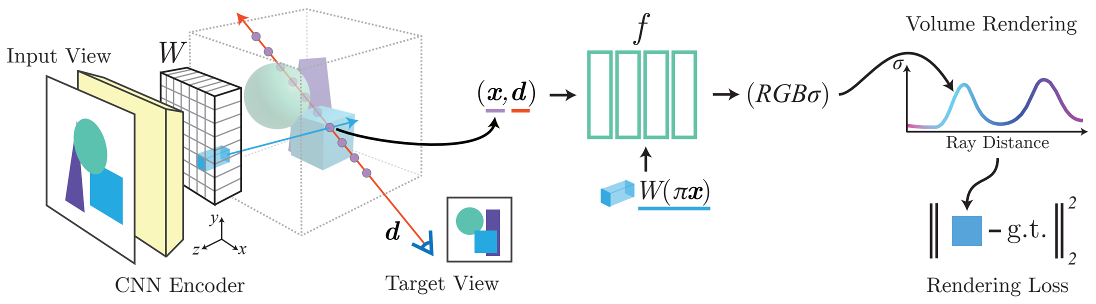


Figure 2.5: Architecture of pixelNeRF

We can interpret the architecture of pixelNeRF in the following way. The image encoder and the NeRF parts work together during training. The image encoder is responsible for providing concrete details (extracted features) for the NeRF network, while the neural radiance field part is responsible for generalizing on the characteristics of 3D scenes and how to render the input features. We might look at the neural radiance field as the part that understands 3D geometry, while the image encoder is the controller to define what should be the content of the scene. In comparison, a regular NeRF model only has the NeRF part, that overfits on a single scene. However, in pixelNeRF, the NeRF part does not overfit but learns to render the encoded features.

2.3.3 Novel view synthesis with pixelNeRF

To synthesise novel views from a pixelNeRF model, the novel views' camera extrinsic and intrinsic parameters are needed. In addition, the input views are also needed with known parameters. First, the input views are encoded by the image encoder. With camera parameters of the target, rays can be marched through 3D space. For every 3D point along the rays, the point is projected onto the encoded feature volumes. Now the input of the NeRF MLP is the 5D vector (same as regular NeRF) and the feature vector from the encoder. The outputs are the colours and densities aggregated (rendered) into output views. We can notice the similarities between pixelNeRF and a regular NeRF network. Here, the inputs of the NeRF MLP are not only the 5D vectors but also the features extracted from the input views.

2.3.4 Training pixelNeRF

Training the pixelNeRF model is almost the same as it is for regular NeRF models, concerning the loss and the backpropagation. However, the logistics are different. During the training, the model loops through the different scenes in the training data. For a training scene, some of the views are encoded (chosen randomly), and other views are rendered. These rendered views are compared with the ground truth RGB, and the loss can be backpropagated through the sum of outputs (the rendered rays). Notice the similarities between pixelNeRF training and NeRF training in Figure 2.5. However, this time, it is not only the NeRF MLP that is trained but also the image encoder. Through the inputs of the MLP, the loss is backpropagated all the way to the encoder.

2.3.5 Multiple image pixelNeRF

Incorporating multiple views into the pixelNeRF model is more complicated. For pixelNeRF, images are also inputs – while for regular NeRF, images are solely outputs – therefore handling multiple images is similar to handling differing-length inputs.

Multiple views provide additional information about the scene and resolve 3D geometric ambiguities inherent to the single-view case. The pixelNeRF model is capable of handling an arbitrary number of views at test time.

In the case when there are multiple input views of the scene, it is assumed that only the relative camera poses are known. For purposes of explanation, an arbitrary world coordinate system can be fixed for the scene. We denote the ith input image as $I^{(i)}$ and its associated camera transform from the world space to its view space as $\boldsymbol{P}^{(i)} = [\boldsymbol{R}^{(i)} \ \boldsymbol{t}^{(i)}]$.

For a target camera ray, we transform a query point $\boldsymbol{x}$, with view direction d, into the coordinate system of each input view i with the world to camera transform as

$$\boldsymbol{x}^{(i)} = \boldsymbol{P}^{(i)}\boldsymbol{x}, \quad \boldsymbol{d}^{(i)} = \boldsymbol{R}^{(i)}\boldsymbol{d} \tag{2.5}$$

Each input image is encoded into feature volume $\boldsymbol{W}^{(i)} = E(\boldsymbol{I}^{(i)})$ (the same way as for a single view). For the view-space point $\boldsymbol{x}^{(i)}$, we extract the corresponding image feature from the feature volume $\boldsymbol{W}^{(i)}$ at the projected image coordinate $\pi(\boldsymbol{x}^{(i)})$.

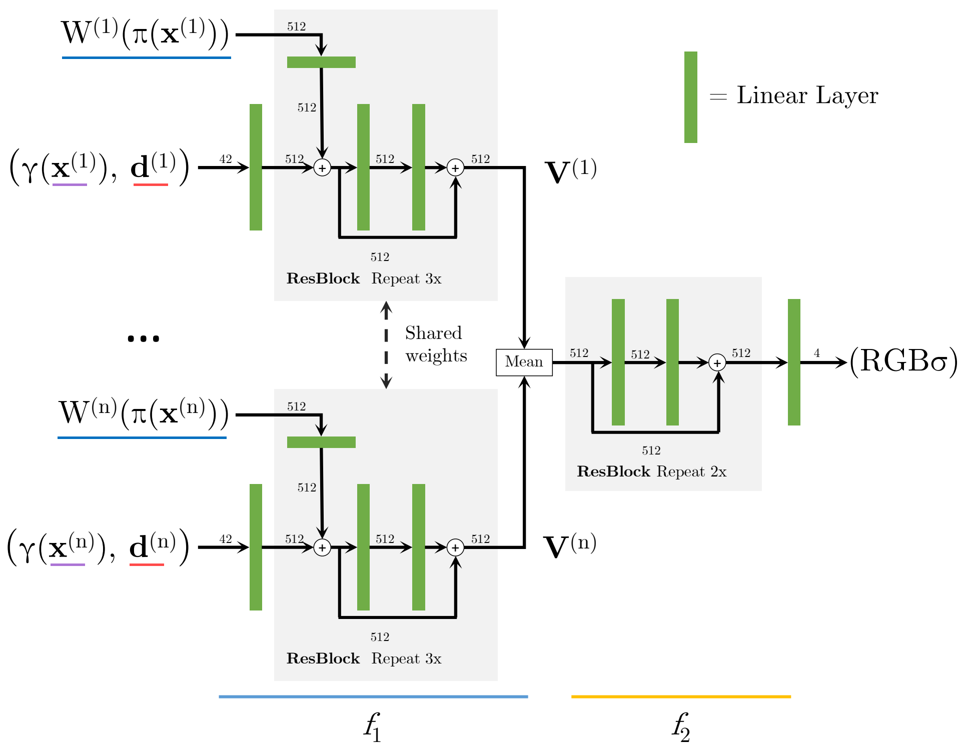
To obtain the output density and colour, the coordinates and corresponding features in each view coordinate frame are processed independently and aggregated across the views within the NeRF network. For ease of explanation, the initial layers of the NeRF network are denoted as f_1, which process inputs in each input view space separately, and the final layers as f_2, which process the aggregated views (see Figure 2.6). The extracted features are passed into f_1 to obtain intermediate vectors:

$$\boldsymbol{V}^{(i)} = f_1\left(\gamma(\boldsymbol{x}^{(i)}), \boldsymbol{d}^{(i)}; \boldsymbol{W}^{(i)}(\pi(\boldsymbol{x}^{(i)}))\right) \tag{2.6}$$

The intermediate $\boldsymbol{V}^{(i)}$ is then aggregated with the average pooling operator ψ and passed into the final layers, denoted as f_2, to obtain the predicted density and colour:

$$(\sigma, \boldsymbol{c}) = f_2\left(\psi(\boldsymbol{V}^{(1)}, \ldots, \boldsymbol{V}^{(n)})\right) \tag{2.7}$$

We can interpret the multi-view case the following way: most of the processing is done the same way as for the single-view case. Finally, in the last layer of the NeRF MLP, the intermediate vectors are aggregated by average pooling to provide the final RGB and density output.



$W^{(1)}(\pi(\mathbf{x}^{(1)}))$
$(\gamma(\mathbf{x}^{(1)}), \mathbf{d}^{(1)})$
42
512
ResBlock Repeat 3x
$\mathbf{V}^{(1)}$
...
Shared weights
$W^{(n)}(\pi(\mathbf{x}^{(n)}))$
$(\gamma(\mathbf{x}^{(n)}), \mathbf{d}^{(n)})$
$\mathbf{V}^{(n)}$
Mean
ResBlock Repeat 2x
4
(RGBσ)
= Linear Layer
f_1
f_2

Figure 2.6: Architecture of pixelNeRF for multi-view

2.3.6 Model details

In practice, both NeRF and pixelNeRF models use the ResNet [5] architecture. The ResNet architecture is comprised of convolutional and fully connected layers in residual blocks. The residual blocks also contain skip connections that perform identity mappings, merged with the layer outputs by addition. This is mostly beneficial in very deep networks, as otherwise, the backpropagation may not reach the start of the network (the derivative is zeroed out through many layers). These residual connections backpropagate better because the derivative is preserved. This made the ResNet architecture one of the most popular models in neural image processing. It is ideal for both the pixelNeRF encoder and the NeRF MLP.

In pixelNeRF, the encoders start the training with a ResNet instance pre-trained on the ImageNet dataset [2]. This model is built into the PyTorch torchvision library. This implementation leverages similarities between the pixelNeRF encoder and other traditional image processing networks.

2.3.7 Evaluation of pixelNeRF

PixelNeRF is capable of novel view synthesis with one or very few images as inputs, while also providing reasonably good results. This is because it is capable of learning on multiple scenes of training data and generalising in 3D representations. The learnt prior is used at test time in a feed-forward manner without any further optimizations. This makes pixelNeRF much faster at test time (while only sacrificing training time). This construction is an ideal enhancement of NeRF models to serve as a basis for object detection.

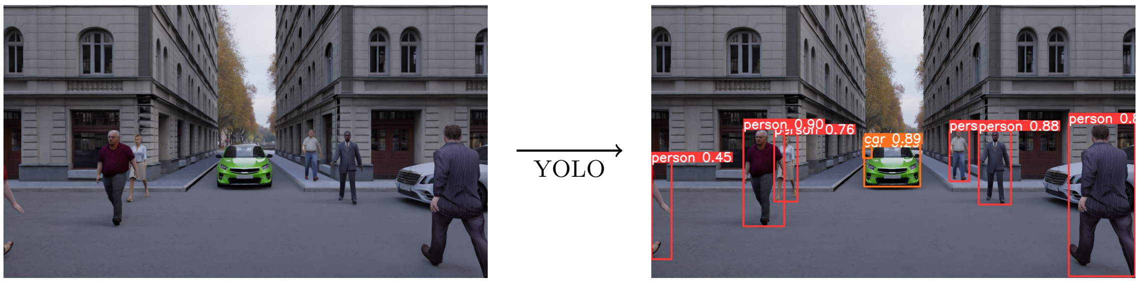
2.3.8 Applying pixelNeRF for object detection

Notice, that the outputs of pixelNeRF are only determined by the last layer. In the original model, it outputs 4 values: RGB colour and density. This can easily be swapped by an object detection layer. The original 4 values can be replaced by the features of a YOLO network (described in section 2.4), to enable object detection. The model is then trained by calculating object detection loss, instead of RGB loss. This way, the learnt features can be used for object detection, instead of colour emittance generation. This is the starting idea of my proposed model, detailed in chapter 4

2.4 YOLO

Object detection is one of the most prevalent areas in image processing. It has a vast number of use cases; therefore, it has been perfected throughout the years. YOLO [16] (You Only Look Once) is probably the most widely used object detection neural network model. Its core goal is to combine and simplify both bounding box generation and object classification into a single feed-forward architecture (Figure 2.7). Unlike other object detection models, YOLO networks propagate the input through the network only once – hence the name You Only Look Once. This makes YOLO networks outstandingly fast and substantially easier to train. This is ideal for real-time applications.

YOLO networks have been around for several years, hence there have been several major improvements made to the architecture. However, the core concepts have remained the same [17] [18] [1] [9] [20]. In this chapter, I will describe the general concept of YOLO networks, mostly known from the YOLOv3 [18] paper.



(a) Input (b) Bounding box detection

Figure 2.7: YOLO object detection

2.4.1 YOLO object detection

YOLO networks divide the input images into an $S \times S$ grid. Each grid cell predicts B bounding boxes. The grid cells are responsible for detecting bounding boxes with their centre inside the given grid cell. Every bounding box can be described by 6 different values: x, y, w, h, a confidence score and the class.

The x, y values represent the coordinates of the centre of the box relative to the bounds of the grid cell (between 0 and 1), see equations 2.8 and 2.9. The w, h values represent the bounding box dimensions. The confidence score combines two pieces of information: the likeliness of an object being present in the current grid cell and how accurate that box is for the object. Formally: $\Pr(\text{Object}) \cdot \text{IOU}^{\text{truth}}_{\text{pred}}$, where IOU is the intersection over union. If no object exists in that cell, the confidence score should be zero. Otherwise, the confidence score should equal the IOU between the predicted box and the ground truth. Finally, the class prediction is responsible for predicting the probability of the class of the object in a grid cell, supposing that there is one. These probabilities are conditioned on the grid cell containing an object: $\Pr(\text{Class}_i|\text{Object})$. The class prediction is not a single value in practice, but the more common one-hot encoding is used (creating N binary values for N classes). These 5 plus N class prediction values form the YOLO features.

It might make sense to predict the width and the height of the bounding box, but in practice, that leads to unstable gradients during training. Instead, most of the modern object detectors predict log-space transforms, or simply offsets to pre-defined default bounding boxes called anchors (see equations 2.10 and 2.11). Then, these transforms are applied to the anchor boxes to obtain the prediction. YOLOv3 has three anchors, which result in the prediction of three bounding boxes per cell (B).

$$\text{box}_x = \sigma(t_x) + c_x \tag{2.8}$$
$$\text{box}_y = \sigma(t_y) + c_y \tag{2.9}$$
$$\text{box}_w = p_w e^{t_w} \tag{2.10}$$
$$\text{box}_h = p_h e^{t_h} \tag{2.11}$$

Where box_x, box_y, box_w and box_h are the centre coordinates and the width and height of the bounding box. t_x, t_y, t_w and t_h are the network outputs. c_x and c_y are the top left coordinates of the grid cell.

Finally p_w and p_h are the anchor dimensions.

Another important detail of modern YOLO networks is that they combine multiple grid scales to predict bounding boxes more accurately. The same feature predictions are made on different scales, acquired from different layers of a convolutional neural network [11]. This leads to better predictions with both larger and smaller objects in the same image.

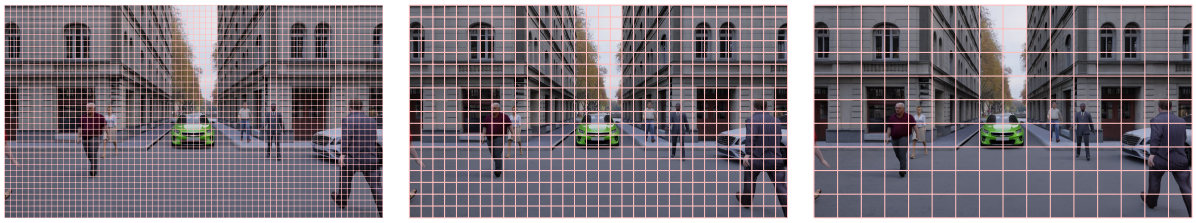


Figure 2.8: Different scales of YOLO grid

2.4.2 Training a YOLO network

While the above method describes an intuitive way to formulate an output for the YOLO object detection model, most training sets do not come in this format. Regular labelling of images mostly contains a varying number of bounding boxes, with no grid. Therefore, the labelled data needs to be transformed into the YOLO format before training. To achieve this, the following process is repeated for every object:

1. Order the anchor boxes by IOU with the object's bounding box.
2. Find the cell matching the centre of the bounding box.
3. At each scale, assign the bounding box to the best matching anchor box (that is not taken).
4. Assign the cell and anchor box probability of 1, and calculate x, y, w, h.

Finally, fill the remaining grid cells and anchors with 0 probability.

This intuitive method transforms the problem of predicting a variable number of bounding boxes per image, into predicting YOLO features on a fixed-size grid (of different scales). This approach can be implemented with a convolutional neural network, with images as inputs and grids of YOLO features as outputs.

To compute the loss, the ground truth and the prediction can be compared. Different losses are formulated for the different parts of the prediction. A mask is created based on the probabilities in the ground truth, which are either 0 or 1. When there is an object present in the target, 3 different losses are calculated. The first is the object loss, which is the mean squared error between the predicted probability and the target value. The target value is the intersection over union (IOU) of the predicted box with the target. The second is the box loss, which is obtained by calculating the mean squared error from the box coordinates and sizes, compared to the ground truth. The third is the class loss, computed as the cross-entropy loss between the predicted and target classes. Where no object is present in the target, the loss is the binary cross entropy between the predicted and target probabilities.

2.4.3 Interpreting the YOLO output

A YOLO network can generate hundreds of bounding box proposals for an image with varying probability scores and overlaps. Most of these boxes are irrelevant due to their low probability or duplicate detections. Therefore, the final step of the YOLO algorithm is to apply non-maximum suppression (NMS) to eliminate duplicate and irrelevant detections. Its input is a list of proposal boxes P, corresponding confidence scores C and overlap threshold N. The output is a list of filtered proposals D, initially empty. To perform NMS repeat 2 steps until P is empty:

1. Select the proposal with the highest confidence score, remove it from P and add it to the final proposal list D.
2. Compare this proposal with all the proposals – calculate the IOU of this proposal with every other proposal. If the IOU is greater than the threshold N, remove that proposal from P.

By the time P is empty, all duplicate and low-probability proposals have been removed. Although this is an efficient way of removing irrelevant and duplicate bounding boxes, highly overlapping objects' bounding boxes might be removed. There are similar implementations like soft-NMS, which handles overlaps better.

NMS is not part of the YOLO training process, it is only there to transform the YOLO grid back to a humanly interpretable output.

2.4.4 Evaluation of YOLO

YOLO is a powerful object detection model, that has become a standard in numerous real-time object detection scenarios. However, as it only takes single-view images as input, it often performs poorly with partially or totally occluded objects. Now that NeRF, pixelNeRF and YOLO have been introduced, it is quite intuitive to combine the pixelNeRF and YOLO models and replace the RGB output of the NeRF MLP with YOLO features. In the coming chapters, I will first introduce the synthetic dataset for this problem, and later the model built from pixelNeRF and YOLO.

Chapter 3

Synthetic data generation

3.1 Application scenario

So far, I have mostly addressed the techniques that were used to create a new pixelNeRF and YOLO-based neural network to enhance object recognition. Before discussing how I have generated training data, I will briefly address the reasons behind this field of application.

Real-time object detection has used YOLO networks for a long time. These networks have proven well in different scenarios and deliver outstanding performance in speed. However, they tend to struggle to detect overlapping objects or ones that are obstructed from the viewing direction. In autonomous driving reliable object detection is essential. Pedestrians and other cars need to be detected from unexpected angles. Different urban road intersections can pose great challenges to detecting road actors whose views are in partial or total occlusion. This field is the primary focus of my application. In this section, I will describe how I have generated synthetic data for this exact application.

3.2 Required data

As accessing a large amount of traffic and autonomous driving-related data from the real world is not feasible, synthetic data generation is the only viable solution. There are pre-built solutions to support self-driving car development, such as the Carla project. Because these environments are usually created to not only generate RGB image input but also to simulate driving, they are usually equipped with unnecessary features for my research. What is more, these virtual environments are usually hard to configure in the sense that there is little freedom in placing objects and modifying virtual intersections. Although these simulators, such as Carla would have perfectly fitted the needs of autonomous driving data, the special requirements of my research would have created an unnecessary overhead in such simulators.

My research needs many occluded objects to test the real performance of an improved algorithm, which is very hard to engineer in pre-built environments. These were the main motivations for deciding to build 3D intersections by hand. I used Blender and pre-made models to construct intersections that are carefully crafted to contain lifelike examples of objects blocking camera views. This way I can make sure to pose a challenge for conventional models and show if my improved algorithm performs better.

As all NeRF models operate with only image inputs and camera poses, these were the primary targets of synthetic data generation, namely the rendering. Every scene contains a camera for every car (forward facing), which are all rendered together with the cameras' extrinsic and intrinsic matrices. Besides these, I have also rendered a depth map and a semantic segmentation image from all views. These are not necessarily needed, but they might be useful in further training and testing. For each scene, I have also saved the bounding boxes for all actors (i.e., cars and pedestrians).

3.3 Rendering pipeline

Blender is mostly designed for 3D artists and less for engineering a deep learning dataset. However, it provides an easy-to-use Python API, which can be used to automate all the rendering and bounding box exports into a clean process. I have created a script, utilizing the API to render all cameras with RGB, depth and semantic segmentation, and save all the bounding box information as NumPy objects.

Blender could not provide all the camera parameters in the desired format, so this script also collected the required information and compiled them into the extrinsic and intrinsic matrices.

In addition, another script was responsible for labelling all the actors automatically before the export, because, by default, Blender does not contain any semantic segmentation capabilities. If the proper Blender project structure is created, this script automatically labels cars, pedestrians, and every other object in the scene. These labels can then be rendered into an image, containing the object IDs as pixel values.

Pure pixel values are not easy to distinguish by the human eye – such as pedestrians with IDs 13 and 14, because the values are too close to each other. Therefore, I have created a visualizer to randomly colour these IDs and make them easy to interpret. The sample data provided in this paper in Figure 3.1 is created this way.

The bounding box data is raw 3D coordinates, so to train or test any models, the bounding boxes need to be projected onto all the desired views. Besides the dataset, I have also created a script that projects the bounding boxes to all the views in each scene. This was done by the pinhole camera model, known from 3D graphics. To project a point from the world to the image, the following transformation is needed:

$$\begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} f & 0 & c_x & 0 \\ 0 & f & c_y & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} R_{3\times3} & t_{3\times1} \\ 0_{1\times3} & 1_{1\times1} \end{pmatrix} \begin{pmatrix} X_{\text{world}} \\ Y_{\text{world}} \\ Z_{\text{world}} \\ 1 \end{pmatrix} \tag{3.1}$$

$$\text{where} \quad f = \text{focal length} \cdot \frac{\text{image width}}{\text{sensor width}}$$

and c_x and c_y is the camera centre pixel coordinates. The two matrices are the intrinsic and extrinsic matrices, also referred to as the camera extrinsic and intrinsic parameters.

However, this poses two more additional factors to consider. The first one is simple: if the object is behind the virtual camera, the pinhole model could still project it onto the image. Therefore, we need to check if it is in front of the camera. This can be done in the camera coordinate system, by checking the Z coordinate value. The second factor needs more careful handling. If an object is in front of the camera, but not in the field of view, it should not be projected onto the image. This is straightforward if the object is a single point in space, we can just clip the bounds of the image with known resolution. However, with objects that have spatial extent, some parts (even the projected centres) might be off the image after projection. This meant, that I had to introduce a threshold for the area of the projected bounding box. If the part visible in the image was over the threshold, the bounding box was kept.

Unfortunately, this meant that some partially visible objects (mostly cars) were omitted, because of perspective distortion. Mathematically only an insignificant part was visible (well under the threshold), but the ratio of the non-visible part was amplified by the perspective. This is very rare in the dataset, so I decided to go on with it.

3.4 Designing the data

All rendered scenes are photorealistic and are created to contain numerous blocking objects. This was done to test the real improvements of a new algorithm capable of detecting objects with obstructed views, and the scenes are interesting enough for this purpose.

Because scenes take a considerable time to construct by hand (to make sure that they are interesting enough), I have used a data augmentation method. All scenes have been rendered with a couple of different HDRs. HDRs define how a 3D scene – an intersection – is lit, reproducing real-world lighting situations, such as cloudy, daytime with sharp shadows and sunsets. This method completely changes the look of a scene; therefore, it creates additional data with minimal effort.

I have also made sure to simplify the pipeline to create these scenes. All the models, like cars, humans and buildings were pre-built to make the modelling faster. I have also implemented the automations described above, to make the exports swifter. This was also done to future-proof the dataset, and if any more scenes are needed, it takes much less to produce them.

I have rendered 12 different scenarios with different HDRs to produce a total number of 32 input scenes (together with exporting all corresponding data mentioned before). This might not seem like much, but the original pixelNeRF model was also trained with much less than a hundred scenes.

3.5 Utilizing the data for a different model

It is quite laborious to create a synthetic dataset like this. However, it was not only used for this paper but was also used to train and evaluate a different multi-view YOLO model, in a shared paper with my supervisor [19]. That model utilized the depth maps provided for each view to enable geometric aggregation. I believe that this dataset is a good foundation for these papers, but it will also be valuable for future models and projects.

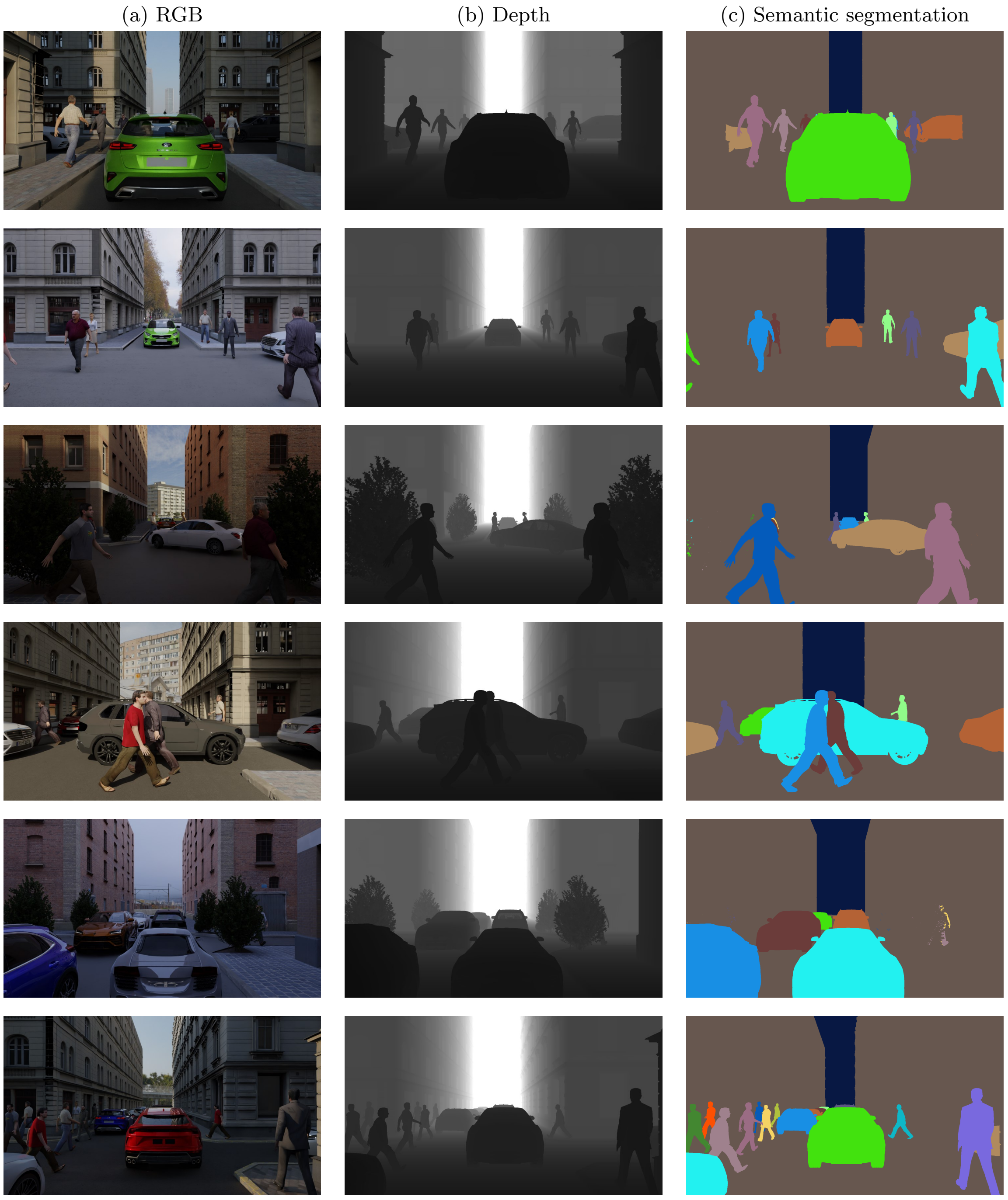


Figure 3.1: Sample images from different views

Chapter 4

Methodology

4.1 Replacing RGB with YOLO features

The main concept of using the pixelNeRF model for object detection is to replace the RGB output with YOLO features and render (i.e. aggregate) the YOLO features along camera rays to get object detection output for the input images. Finally, train this model with YOLO loss instead of RGB loss.

This may seem like replacing a few layers of a network, but in practice, it required at least partial redesign of most components of the pixelNeRF model. In this chapter, I will describe the design details of the new pixelNeRF-YOLO network and how it was trained. I will also present some of the hyperparameters of the architecture and how they were optimized.

The original pixelNeRF model was implemented in Python using the PyTorch library. I used the original code base as a starting point and created a fork on GitHub for my redesign.

4.1.1 Forward propagation

The easiest way to understand how this model works is to follow the data flow through the forward propagation, from loading the dataset to the object detection output.

Loading the dataset

The dataset described in chapter 3 contains all the information needed to train and test the model. The images and the intrinsic and extrinsic matrices can be loaded directly into the model. However, bounding boxes are different. They are provided in a list of projected bounding boxes for each view. These lists need to be converted to YOLO format: $S_{\text{scale}} \times S_{\text{scale}}$ grids on different scales, with each bounding box assigned to a cell by its centre and the closest matching anchor by IOU. The dataset is implemented by a class derived from the PyTorch Dataset class. In one request, it returns all the information for a single scene: RGB images, camera intrinsic and extrinsic matrices, and the bounding box data transformed into YOLO format.

Preparing to render

When a scene is loaded, three images are selected to be encoded in the pixelNeRF network. The three images are randomly selected during training and can be manually specified for testing. These images are sent through the encoder described in section 2.3. Features are later extracted from the different layers of this network. I have tested two different encoders: the ResNet from the original pixelNeRF and the backbone network of YOLOv7.

After encoding the images, camera rays are generated for the three encoded views using known camera parameters and the grid cells. One ray is marched through the centre of each grid cell. This implies that camera rays should be generated separately for all scales (with different grid resolutions). The rays are described by 8 values: xyz coordinates for the starting point, xyz values for the direction (represented by a vector), and near and far values. The near and far values represent the limits of the camera rays.

In the pixelNeRF-YOLO model, camera rays are generated for each grid cell of all three input views. This approach differs from the original pixelNeRF model. There, rays are generated for **all** the views in the scene (even for those that were not encoded), and a random selection is used during training. This makes sense in the original approach: novel view synthesis needs to learn how to reproduce unencoded images. However, in pixelNeRF-YOLO, the goal is not novel view synthesis, but object detection for **known** views. Therefore, pixelNeRF-YOLO only generates rays for the encoded views, where the RGB value is known. The generated rays are then passed to the renderer. To see how the rays are generated refer to section 5.3.

Model forward calls

The renderer generates xyz coordinates along the camera rays between the near and far bounds, dividing them into a predefined number of discrete points. This is the ray resolution. The model is then called with these points in 3D space, along with the corresponding view directions.

$$\begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} f & 0 & c_x & 0 \\ 0 & f & c_y & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} R_{3\times3} & t_{3\times1} \\ 0_{1\times3} & 1_{1\times1} \end{pmatrix} \begin{pmatrix} X_{\text{world}} \\ Y_{\text{world}} \\ Z_{\text{world}} \\ 1 \end{pmatrix} \tag{4.1}$$

The model projects these 3D points to all the encoded views (Equation 4.1). The features are extracted from different layers of the encoder network for each encoded view, indexed by the projected points. If the 3D point is projected outside the encoded view, or it is behind the given view, the features are replaced by all zeros. Since the feature map size is not the same in the different layers, the smaller feature maps are upscaled via bilinear interpolation. This feature extraction is called indexing in the model. The number of features produced by the encoder depends on the used implementation (size of the ResNet or the YOLO backbone). Larger encoders produce more features. The number of layers from which features are extracted can also be specified. By default, the ResNet loads pre-trained weights from the PyTorch library, trained on the ImageNet dataset, while the YOLO backbone loads pre-trained weights, published with the paper.

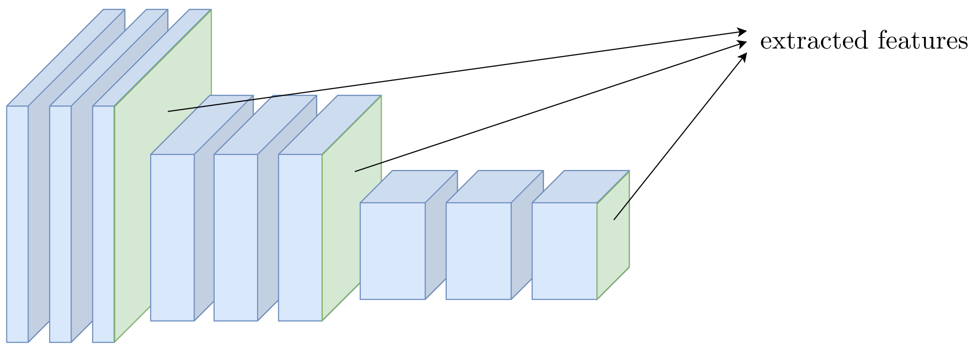


Figure 4.1: Extracting features from different layers of the encoder

The thicker boxes represent layers with more channels. As the input is scaled down, the number of channels increases.

After indexing, the features are concatenated with the x, z, y coordinates and view directions (in the corresponding input view space) and fed into the NeRF MLP. The NeRF MLP is a custom implementation of the ResNet architecture. The difference is that this custom implementation feeds the data through the first layers of the ResNet separately for the input views. The individual extracted features are also input at every residual block. Finally, the separately processed input view features are aggregated by average pooling, and the aggregated features are passed through the final layers of the network.

The output of the NeRF MLP is 7×3, which is the YOLO output features $\times$ the number of anchors per scale. The YOLO features are the confidence score, x, y, w, h, and the class (as 2 values for car and human). This output simply replaces the original RGB and density output by replacing the last linear layer of the network with a larger one.

Rendering

The 7×3 output values of the NeRF MLP are returned to the renderer (with the same shape as the inputs along the rays). These raw values need to be aggregated along each camera ray. In the original model, how these values are aggregated (rendered) is defined in computer graphics (described in subsection 2.2.2). In the pixelNeRF-YOLO model, however, this is not actual rendering because there are no RGB values. Therefore, I have designed a different pipeline.

The confidence values are fed through a sigmoid (Equation 4.2). Then x, y, w, h and class values are weighted (multiplied) by the confidence at each queried 3D point. Then these multiplied values are averaged along the camera rays to produce the final output x, y, w, h and class values for the given ray (weighted average by the confidence score, Equation 4.3). Finally, the output confidence is the maximum confidence value along each ray (Equation 4.4). This aggregation is designed to achieve the following: The NeRF MLP only needs to accurately predict x, y, w, h and class values where it is confident that an object is present in 3D space. Where there are no objects, the predictions do not alter the output significantly (if the confidence is correctly low). The confidence is chosen as the maximum along each ray because if there is an object along the ray (even if it is behind other objects) it should predict a high confidence. Ideally, the confidence will be high (close to 1) in 3D space where there is an object with accurate x, y, w, h and class values. Where there is no object in space, the confidence should be low and the remaining values do not matter.

$$\text{confidence}_i = \sigma(\text{confidence}_i) \tag{4.2}$$

$$(x, y, w, h, \text{class})_{\text{ray}} = \frac{\sum_i \big((x, y, w, h, \text{class})_i \cdot \text{confidence}_i\big)}{\sum_i \text{confidence}_i} \tag{4.3}$$

$$\text{confidence}_{\text{ray}} = \max_i(\text{confidence}) \tag{4.4}$$

Because the NeRF MLP directly predicts bounding box values, there is no way to account for perspective distortion during aggregation (making objects further away appear smaller). However, the input to the MLP contains the xyz values in the input view space, where the z coordinate is the distance from the camera. This way, the MLP can learn, how distance affects the size of the bounding boxes, and accurately predict them for the queried view.

Interpreting the results

After the rendering, the output of the model matches the output of a YOLO network. During testing, this can be visualized, by converting the grid back to bounding boxes, applying non-maximum suppression, and then drawing the bounding boxes onto the input views.

Loss

When the forward propagation's output is used to train the model, the standard YOLO loss can be applied to the renderer's output. This trains the model in the same way as the original RGB loss in pixelNeRF.

4.1.2 Renderer

Although the above-described method to render (aggregate) the output of the NeRF MLP seems straightforward, the weighted average and taking the confidence score maximum along the camera rays might prevent the proper backpropagation of the gradient. Taking the maximum only backpropagates the gradient to a single value, which may decrease the effectiveness of the training. In addition, initial values could play a very important role in the maximum and the weighted average, making the training much less predictable.

These motivations led me to experiment with a much simpler renderer: aggregating the values by average along each camera ray. Although this does not support the original concept well, it is a good way to test if the renderer prevents the gradients from reaching the start of the network.

After testing with many different settings, this renderer proved to be inferior, so I stuck with the original intuitive approach in the final model.

4.1.3 Imbalanced data

In the pixelNeRF model, each ray is equally important, since each of the queried rays contains an equal amount of colour information. However, in the pixelNeRF-YOLO model, not all rays contain objects, in fact, the number of rays containing an object is usually around 1-5% of all rays. This makes the dataset skewed and requires special attention. It is easy to see, that the training can minimize the loss by predicting no bounding boxes at all, and consequently only missing in 1-5% of the predictions.

Although this was foreseen, it still caused some setbacks in setting the optimal parameters for training. I found two different ways to address this imbalance, both of which had to be constantly tested throughout the training experiments. The first is to introduce weights for the YOLO loss. If missing a bounding box is penalised more than incorrectly predicting one, the model will be biased towards producing higher confidence scores and predicting more bounding boxes. This is an obvious solution, but there is no well-defined method for finding the exact weights. I will discuss this with other hyperparameters in subsection 4.3.2.

The second option is to modify the scales of the YOLO grids. By increasing the size of the grid cells, the same number of boxes are distributed into a smaller number of cells, therefore artificially increasing the ratio of object to no object rays. While this is a good solution for a skewed dataset, the downside is that by increasing the size of the grid cells, the resolution and performance of YOLO can be lost. Fine-tuning this and predicting on multiple scales is also a hyperparameter and will be discussed in more detail in subsection 4.3.2 and in chapter 5.

4.2 Training

The complexity of the pixelNeRF-YOLO model presents several challenges during training. This section will detail the training process, outlining design decisions and the discovery of optimal solutions. Notably, the model and its training have a considerable number of hyperparameters; therefore the final section of this chapter addresses the fine-tuning of these parameters.

4.2.1 Loss

In the pixelNeRF-YOLO model, a YOLO loss function was utilised to compare the outputs with the ground truth data. As previously noted, both the predictions and ground truth are available in YOLO grid format. The exact implementation of the loss function differs amongst various YOLO models. I used the one described below.

The input of the loss function is the prediction, the target and the anchor boxes. A mask is created based on the probabilities in the target, which are either 0 or 1. When there is an object present in the target, 3 different losses are calculated. The first is the object loss, which is the mean squared error between the predicted probability and the target value (Equation 4.5). The target value is the intersection over union (IOU) of the predicted box with the target. The second is the box loss, which is obtained by calculating the mean squared error from the box coordinates and sizes, compared to the ground truth (Equation 4.5). The third is the class loss, computed as the cross-entropy loss between the predicted and target classes (Equation 4.6). Where no object is present in the target, the loss is the binary cross entropy between the predicted and target probabilities (Equation 4.7).

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4.5}$$

$$\mathrm{CrossEntropy} = -\frac{1}{n}\sum_{i=1}^{n}\left(\log\frac{\exp\left(y_{i,\hat{y}_i}\right)}{\sum_{c=1}^{C}\exp\left(y_{i,c}\right)}\right) \tag{4.6}$$

$$\mathrm{BinaryCrossEntropy} = -\frac{1}{n}\sum_{i=1}^{n}\left(y\log(\hat{y}) + (1-y)\log(1-\hat{y})\right) \tag{4.7}$$

where y is the prediction and $\hat{y}$ is the ground truth. For the cross-entropy, y is a C long vector of probabilities and $\hat{y}$ is an integer from $[0, C[$.

These four losses describe various aspects of the predictions. For optimal performance of the model, all losses must remain low simultaneously. If any of the losses are comparatively large, it will significantly affect the model's performance. For instance, if the box loss is excessive, but the other losses are low, it indicates that the bounding boxes were accurately detected, but they do not match the ground truth. These predictions are considered incorrect as they do not match closely enough. This indicates that a smaller overall loss may result in poorer performance if the different losses are not balanced.

To account for the skewed dataset, I have introduced weights for the four losses. All losses are multiplied by a corresponding weight, which are hyperparameters, and the gradients are calculated as the total of the weighted losses. This concept is often used in YOLO loss functions, but my dataset and application required these weights to be tuned slightly differently to regular YOLO networks.

4.2.2 Visualization

The training loop of the original pixelNeRF model comprises multiple components, the main one being the loss and gradient calculations used in the backpropagation process. In addition, it periodically performs an evaluation and a visualization. The evaluation uses the same loss calculation as the training but on the test data without any backpropagation. This, however, does not provide adequate information about performance. It is useful to tell if the model overfits or how far the training has progressed but tells very little about how it actually performs. For this reason, the visualization proves valuable. Formerly, the visualization entailed a sample render of a random target view from the test set. This has now been replaced with object detection on a random target from the test set. The process randomly selects three views from the test data, encodes them into the model, and performs object detection on one of the input views. It also visualizes the object detection, by performing the NMS algorithm and drawing the bounding boxes on the target view image. The selected views, along with the ground truth and predicted boxes, are saved for easy tracking of training progress. Evaluation and visualization occur at predetermined intervals that can be set before each training begins.

4.2.3 Metrics

In addition to evaluation and visualization, it is advantageous to quantify the performance of the model. While the loss function is not very effective for this purpose, other metrics can be computed. Typically, precision and recall are used to gauge efficiency in detection tasks. I have implemented a function that takes the predictions and the ground truth values and counts true positives (TP), false positives (FP) and false negatives (FN). (It is evident that quantifying the true negatives is not feasible.) The precision (Equation 4.8) and recall (Equation 4.9) can be derived from these values, along with the F_1 score (Equation 4.10), to objectively rate the model's performance. These metrics are regularly calculated during training to monitor progress.

To calculate the TP, FP and FN values, we need to compare the prediction boxes with the targets. If a predicted box has an IOU greater than a predefined threshold with a target box and labels the same class, it is counted as a true positive. Conversely, if no matching boxes are present with the IOU greater than the threshold and the same assigned class, the prediction is classified as a false positive. Any remaining targets are labelled as false negatives, as no corresponding predictions were found for them.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{4.8}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{4.9}$$

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2 \cdot TP}{2 \cdot TP + FP + FN} \tag{4.10}$$

4.2.4 Overfitting

Complex models are frequently tested initially by overfitting them on a small portion of the training data. Overfitting involves training and testing the model on the same data to solely minimise the loss for selected inputs. This way it can be tested that all components function well. However, if the model can overfit, it solely shows its capacity to learn the training data, and does not offer any indication as to how well it can generalise on unseen data. This is a promising preliminary evaluation to verify the validity of the code, and also to know if the approach is well-founded and capable of learning. Overfitting is a beneficial procedure to reduce the risk of the model's failure, but it does not provide direct insight into its actual performance.

The testing commenced by overfitting the model on a single scene. I identified multiple bugs in the code and was able to prove that this architecture can learn object detection from multiple input views. I will present the findings on overfitting in the evaluation section, as outlined in chapter 6.

4.2.5 Logging, comparisons

Due to the intricacies involved in the pixelNeRF-YOLO model, numerous hyperparameters, settings and training results need to be tested, so it is important to keep track of these results and track the progress of the research. I have created a framework to document all information that might later be useful to compare runs and settings.

When a training session starts, the model outputs all the configuration parameters. Throughout the training, the system periodically logs the data mentioned earlier, specifically, the losses and metrics. This information is saved as a text output, with the numerical values also stored as NumPy arrays. These arrays can be utilized later for visualization purposes, in order to acquire more in-depth insights from the training process. Several of these visualizations will be presented in chapter 6. Subsequent to each successful training session, all of the logs, visualizations, and model weights were archived. From this data, any training can be resumed or analysed later. The configurations and results of the training have been recorded in an Excel spreadsheet to monitor the experiments that led to better outcomes. This was necessary due to the large number of parameters to tune, which otherwise would have been impossible to keep track of. The insights gathered from this will be shared in chapter 6.

4.3 Hyperparameters

Most deep learning models have a significant number of hyperparameters. Hyperparameters are those parameters that are not optimized during the training. These parameters often involve the model architecture (where the number of layers or the layer sizes are parameters), the learning rate and other similar configurations. These parameters are often hard to optimize, with no defined way of doing so. As a result, optimization usually relies on heuristics and trial-and-error approach. Unfortunately, combining the pixelNeRF and the YOLO models implied building the model on a lot of hyperparameters. This section will outline the most important hyperparameters of my model and discuss how I optimised them. In the evaluation in chapter 6, I will share the results of the best-performing combination of hyperparameters.

4.3.1 YOLO grid size

The size of the YOLO grid turned out to be one of the most important hyperparameters. The YOLOv3 model, upon which my YOLO object detection is based, uses cell sizes of 16, 32, and 64 for the three different scales. With the images used in the original paper, the grid sizes are 52×52, 26×26 and 13×13. Because my images have a different size and width-height ratio, I had to use different cell sizes. The images in my dataset are rendered at a size of 1920×1080, however they are downscaled to 25% and 50% for different training experiments. As an initial configuration, I used a cell size of 15. This cell size is close to the original YOLO implementation, and it is also a common divisor for the width and the height – so that the number of cells is integer. For model development and training, I started by using only one scale and later experimented with more.

Initially, it seemed a good idea to start with the highest resolution (and smallest cell size), but after training the model with otherwise unchanged configurations, it turned out that the cell size of 15 did not give good results. I conducted experiments with a cell size of 30, which turned out to give much better results than 15, while requiring less training time, because a lower grid resolution means fewer rays to render. From that point on I only experimented with lower grid resolutions and multiple scales.

The optimal cell size being 30 or more may correspond with relevant comments shared in subsection 4.1.3. This cell size appears to strike the ideal balance between achieving high enough resolution and avoiding an excessive number of empty cells, which do not contain any objects.

4.3.2 YOLO loss weights

I have already explained the significance of YOLO weights in balancing the various losses that contribute to the overall YOLO loss. Based on estimations, I have found that 1-5% of cells, and therefore rays, contain bounding boxes, so it seems logical to set the object loss to 100-20. This would exactly compensate for the ratio between cells containing and not containing boxes. Based on my experiments, values above

20 do not have a significant impact on the predicted confidence scores, while values around 10 or lower produce extremely low confidence scores resulting in very few predictions. Therefore, I have determined that an object loss weight of 20 consistently yields the best results.

While I did experiment with adjusting the weights for box and class loss, these modifications did not appear to significantly influence the training outcome. This could be explained by the fact that confidence scores and box and class values require separate optimization, and minimizing losses in one area does not have a detrimental effect on the others.

4.3.3 Ray resolution and limits

The ray resolution is the number of samples that are taken between the near and far limits of a ray. At these sampling points, the NeRF MLP is queried. The initial value used in the pixelNeRF model was 64. I have adjusted the near and far limits to roughly match the size of the scenes, which is roughly 30 to 40 metres. Initial assumptions did not yield satisfactory results. However, modifying the sample size to 128 and limiting ray length to a range between 1 and 13 produced superior results, surpassing all other attempts. A resolution of 256 was also trialled, resulting in almost identical results. The resolution has a linear effect on the training time required (because it determines the number of 3D point inputs). The 128 resolution and near and far bounds of 1 and 13 proved to be optimal, balancing accuracy and training times. The ray hyperparameters and grid resolution were found to be the most crucial factors in achieving desired results.

4.3.4 Network size and encoder type

In terms of network architecture, larger networks generally lead to improved accuracy, granted sufficient training data. After attempting to use larger networks (with bigger ResNets for the encoder and additional residual blocks for the NeRF MLP), it became evident that this had no effect on the output whatsoever, even when overfitting on a single scene. One explanation for this could be that the network size originally used for the pixelNeRF implementation is already capable of processing all the information available in a given scene, and that performance cannot be improved by using a larger network.

Another hyperparameter is the type of network used. I have tried replacing the ResNet encoder with the backbone network of YOLOv7 (using the implementation published together with the paper[1]). The decision was motivated by the fact that the ResNet was trained on the ImageNet dataset, which excludes images of humans. Although both networks have the potential to perform well, noticeable differences were observed between the results. The evaluation of these two encoders will be presented in Chapter 6.

4.3.5 Input image size

The input image size is a simple hyperparameter: a bigger image size can never impact the results negatively (because of the additional) information, but the bigger the images, the more VRAM is required during the encoding. This makes it easy to find the optimum: find the largest tolerable image size that still improves the output. In my case, this was a straightforward process; the model produced the best results with an image scale of 0.5. Any scale above 0.5 always ran out of CUDA memory on any of the available devices, even with 24 GB of VRAM available. This prevented the testing of higher resolutions. Future improvements may include evaluating the model with increased input image resolutions.

4.3.6 Learning rate

As a hyperparameter of deep learning models, the learning rate is critical. The key is that setting the learning rate too high could prevent the model from converging, while having it too low will result in prolonged training. Initially, I used the same learning rate of 10^{-4} as the original pixelNeRF model. That seemed to work well, so I experimented with smaller learning rates (like 10^{-5}) if they helped better convergence. I discovered that I was able to attain slightly better results by lowering the learning rate to 10^{-5}. To achieve the optimal result, I ultimately settled on implementing a learning rate decay, which reduces the learning rate by a predetermined factor at the end of each epoch. I started from 10^{-4}, with a factor of 0.9. This way, the model starts to converge fast, and if there is any chance that a lower learning rate helps the training, I just need to let the training run for longer and wait for the decay.

[1]**github.com/WongKinYiu/yolov7**

Chapter 5

Implementation details

In this chapter, I aim to provide informative details about the implementation, focusing on less straightforward parts of the code, and technological limitations. The code is publicly available on GitHub[1] for concrete implementation.

5.1 Custom data loader

In the previous chapter, I described how the data was loaded through a custom data loader class, and how the bounding boxes are required in the YOLO format. Although the pixelNeRF-YOLO model was built by using pre-existing implementations of two models, the data format that is suitable in one model might not be appropriate in the other model. Transforming between the formats of the two models constituted a formidable challenge, especially in the entire data flow and data loader.

The data loader of YOLOv3 generates $S_{\text{scale}} \times S_{\text{scale}}$ grids for all input views. As the grids do not match in size, they cannot be combined into a single tensor. The original implementation used tuples instead. However, I require these YOLO grids for all the views in my model. Thus, the tuples must be collected in a list for all views. Consequently, for a scene with NV views and 3 scales, the data structure consists of an NV long list of 3 long tuples containing the grids, wherein every grid cell contains 6 values of bounding box data for 3 anchors. The complexity is due to the grid cells of differing sizes, preventing stacking into one tensor. Loading batches of more than one scene would have complicated matters further, but memory restrictions prevented this from becoming a problem.

Data can be loaded as described above but the pixelNeRF model is not equipped to render on multiple scales. Therefore, it was necessary to amend the training function in order to loop through all scales.

Although this may seem like an excessive and unnecessary task, it arises from the combination of two models, neither of which was designed to be compatible with the other. Possible solutions included rewriting one or both models for compatibility or using more complex data formats. I have decided not to rewrite any implementations and to handle the transformation of the data between the two.

5.2 CUDA memory

All contemporary deep learning models are trained on GPUs in order to take advantage of fast tensor operations and parallel computing. While GPUs enable faster model training than CPUs, they have hardware limitations, such as lower memory capacity (VRAM). The pixelNeRF model is rather large (not in terms of industry state-of-the-art models, but for the consumer world) with over 24 million trainable parameters with the ResNet encoder and over 43 million with the YOLOv7 backbone. What is more, the parallel encoding of all the input images requires to store intermediary feature vectors, which produces an additional load for the VRAM. The model has been trained using various GPUs, including Nvidia Titan XP, Nvidia RTX 2080 Super and Nvidia RTX 3060 Ti.

Unfortunately, most of the hyperparameter optimisations that improved the model's performance made it more hardware-intensive. Increasing the resolution of rays or the YOLO grid necessitates querying more points and rays, which subsequently increases the computational load. While these can be processed simultaneously, parallelization requires more VRAM. Furthermore, the processing of larger input images

[1] github.com/kofinandi/pixel-nerf-yolo

results in the generation of larger feature maps and also requires more graphics memory. Due to these factors, modifications were required to enable the model to function with limited memory capacity.

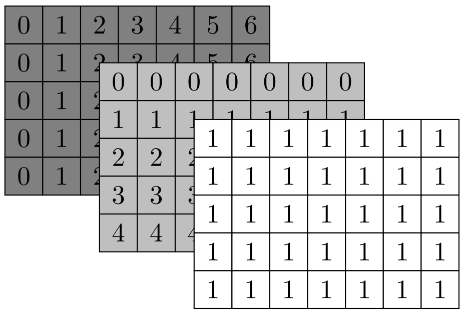
During loss calculation and backpropagation, the rays for rendering were separated into sub-batches. For a given scene, the losses are computed for subsets of all rays. Backpropagation is performed in multiple steps, enabled by the PyTorch library storing gradients. This allows the model to fit into the GPU memory but at the expense of significantly slower training times.

Despite optimizing all components to make the best use of resources, certain training configurations remained untestable due to hardware limitations. Raising the ray resolution to 256 or higher leads to training times of over 10 hours, even when overfitting to a single scene. Likewise, reducing the grid cell size and increasing grid resolution also prolongs training, to about 40 hours. When using an image scale greater than 0.5, the model cannot be started with any sub-batch size on 24 GB of VRAM or less due to a lack of CUDA memory. These areas are unfortunately out of my reach to test; however, they might hold valuable insights on how to improve the model even further.

5.3 Poses

The original pixelNeRF model was trained and tested on multiple datasets, each featuring different methods of storing intrinsic and extrinsic camera parameters, but no documentation was provided regarding these formats. For my dataset, I have used the representation that was provided by Blender. Although I was aware of the disparity between the pixelNeRF coordinate system (X right, Y up, Z out) and the Blender coordinate system (X right, Y in, Z up), the extrinsic tensors were not only in distinct coordinate systems but also in completely different formats. These matrices that hold the extrinsic parameters are referred to as poses in the pixelNeRF implementation. Due to a lack of documentation, the necessary transformations needed to convert from the Blender extrinsic representation to pixelNeRF were unclear. As a result, the projection had to be replaced entirely to conform to the Blender data format. This allowed for direct use of my dataset.

With the transition to the Blender data format, I also had to reimplement the ray generation. Rays can be generated from extrinsic and intrinsic parameters in the following way. The intrinsic and extrinsic matrices are inverted and two grids are created filled with the cell X and Y indices. The grids are concatenated with an additional grid filled with ones (for the homogeneous coordinate) to create a $W_{\text{grid}} \times H_{\text{grid}} \times 3$ tensor. Then, this tensor is multiplied by the inverted intrinsic matrix, to get the directions in the camera space. (It is easy to validate this logic reversed.) Finally, these directions are multiplied by the inverse extrinsic matrix to transform them into the world space (see Equation 5.1).



$$\times \quad \boldsymbol{M}_{\text{intrinsic}}^{-1} \quad \times \quad \boldsymbol{M}_{\text{extrinsic}}^{-1} \tag{5.1}$$

Although this transition made the code unambiguous, realizing the differences and testing the correctness was a great challenge. With no guide from the original code, any parts that were not implemented by me had to be directly tested due to these subtle differences.

The unfortunate feature of using the PyTorch library is that if the model works mathematically, it might still not have the correct functionality. Projection is mathematically the same for different pose matrices, which is just a simple vector-matrix multiplication. However, if the data is not in the correct format, the projection will be incorrect, and the acquired features will be inconsistent. This does not give any errors or visible signs, only that the model is unable to learn.

For the original research paper or the code, it would have been helpful to provide details on concrete formulas and data representations used for reproducibility.

5.4 Debugging the model

As previously mentioned, distinguishing between an incorrect implementation and a poorly performing model can be challenging in many cases. There are several causes of this difficulty. The first aspect to consider is that mathematical operability does not guarantee logical correctness.

The second major challenge in ensuring that the code works correctly comes from the data format. Most of the data in the model is represented by multi-dimensional tensors, which are hard to interpret. The dimensions can be checked, but the contained data is often just the output of the MLP, which is not human-readable. Therefore, there is no way of validating the data flowing through the system, and if the model underperforms, it is hard to trace the causes.

To mitigate the possibility of the code containing such errors, I have designed unit tests to separately test the components. This way even if the model underperformed, it was easier to narrow down the possible sources. I was able to find errors in the projections and the loss calculations this way.

5.5 Saving backups

The original pixelNeRF model was well-prepared for unexpected events during training. The model is always saved by first creating a copy, to prevent the saved weights from becoming corrupted. However, redesigning many parts of the code introduced numerous new possibilities for errors. I have created a system that saves a backup of the model weights at the end of every epoch. This way, the states can be restored even if the training somehow overwrites the model weights with incorrect data. What is more, if there is an interesting point during the training – such as an unexpectedly high or low metric value – the model can be tested with the weights saved at the nearest checkpoint.

This provided an additional analytics tool and the safety to let trainings run for a long time without any supervision.

5.6 Mid-training checks

There are numerous reasons why a deep learning model may produce NaN (Not a Number) values during training. Unfortunately, during the development of the pixelNeRF-YOLO model, I also encountered bugs that produced NaN values. These were very hard to debug because the root cause was hard to identify.

The initial NaN issue in the model occurred due to zero division in the renderer. After hours of training, the sum of probabilities along rays was able to reach zero, and the weighted sum became a NaN. The second NaN problem arose from the new implementation of the projection. When query points were too close to one of the cameras, the projected point could have a near-zero value, which produced infinite values. To prevent these errors from corrupting trainings and make the causes easier to find, I have added frequent NaN and infinite value checks into the model. When the values were encountered, it would print the location and other details and stop the training from saving the incorrect values.

This technique helped to prevent other errors but caused an overhead to the training time. I believe that this was a good solution, but it was important to consider and evaluate the trade-off between frequent checks and longer training times.

5.7 Testing, metrics and visualization

I have already introduced the visualizations and different metrics that are calculated during training in sections 4.2.2 and 4.2.3. I have made sure to implement these parts of the code so that they can be utilized outside of the training. This way any model can be tested after training and new evaluations can be conducted on older results. By making sure that the evaluation during and after training uses the same code, I can guarantee that the results I see while the training is running will be reproducible later. This is a significant part of the continuous testing and improvement.

Chapter 6

Evaluation

In this chapter, the evaluation process of my model will be introduced, followed by the presentation of the results obtained from various configurations. Finally, a comparison between my model and the state-of-the-art YOLOv7 model will be discussed.

6.1 Evaluation method

The metrics used to assess the performance of the model were previously introduced in subsection 4.2.3. Although loss provides a good measure of how different trainings compare, it provides minimal insight into the performance of the model. To evaluate the performance of pixelNeRF-YOLO in object detection, objective measurements including precision, recall and F_1 score can be computed. These metrics provide a means of comparing the model with other object detection models. To calculate precision, recall and F_1, the predicted and ground truth bounding boxes must be compared. To accept a predicted box as correct, an intersection over union (IOU) threshold must be defined. This match IOU threshold is critical. Using lower values accepts erroneous boxes, while higher values mark seemingly correct boxes as incorrect. I have suggested 0.4 IOU as a good standard to provide a good basis for evaluation. However, as I will explain later in this chapter, a lower IOU can also be valid. For fully occluded objects, a good guess with 0.2 IOU is still a major improvement to not predicting any boxes at all. Although these IOUs seem low, they mean that there is a significant overlap between the prediction and the target, or total containment with only size differences, which means almost perfect localization. See Figure 6.1 to visualise what these IOU values mean. When an object is not visible good localization is the key rather than predicting perfect bounding box sizes.

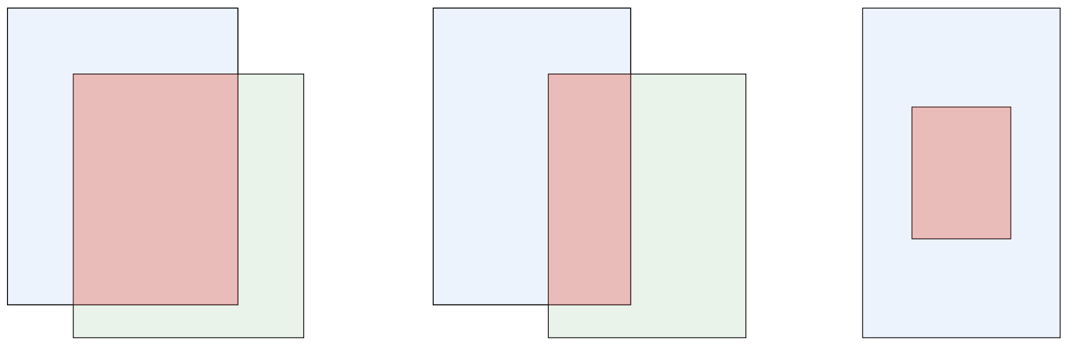

(a) IOU 0.4 (b) IOU 0.2 (c) IOU 0.2

Figure 6.1: Visualising different IOUs

Alongside numerical comparison of different configurations and models, I will share visualizations to highlight strengths and weaknesses. I assert that combining numeric and visual evaluations provides a more comprehensive comprehension of performance.

In addition to the match IOU threshold, several other evaluation parameters must be accounted for when conducting non-maximum suppression on predicted bounding boxes. The initial threshold involves filtering for confidence, with any bounding box below this value ignored. Reducing this value generally results in increased recall and decreased precision. The objective is to determine the optimal value that attains high precision and recall to maximise F_1.

A similar parameter is the overlap IOU threshold. If two boxes have an IOU over the threshold, the one with the lower confidence is discarded. Keeping this value low discards more bounding boxes, therefore recall might decrease and precision increase, but I found that this value does not have a significant effect in most cases.

These two values are comparable to the single threshold implemented in binary classification for generating the ROC curve. The optimization process is also analogous, but due to this model having more parameters, I have utilised a heuristic approach.

I will share the results for all experiments by providing all the optimal parameters and how they were found, in addition to the metrics and visualisations.

6.2 Development approach

My development approach after finishing the implementation was as follows. First, test the model by overfitting on a single scene. This proves the correct functionality and provides the opportunity to correct any logical errors. After the implementation functions well, optimise the hyperparameters until the model gives good results for overfitting (as it was severely underfitting in the beginning).

After these results, move on to real training with training and test sets. With this approach I was able to narrow down the functioning models and optimal hyperparameters. This is advantageous because training on the whole dataset takes about 30 times longer than overfitting on a single scene, and by knowing which models work well in overfitting, only a much smaller number of them need to be tested on the whole dataset. This chapter presents both the overfitting and regular training results in the order in which I progressed through the stages of improvement.

6.3 Benchmark model

The primary objective of this study is to advance object detection for obstructed objects through the use of multiple views. As this was the desired goal, first the benchmark model had to be tested. The Ultralytics YOLOv8 model was selected for testing purposes (available on GitHub[1]). This model is actually a pre-trained object detection package based on the YOLOv7 paper. As foreseen, this implementation achieved very high precision but lacked recall. This is a result of the model's design: it will only detect visible objects and will never identify occluded objects. I have tested this model on the same test set, as I tested the pixelNeRF-YOLO model. The results can be found in Table 6.1 for different match IOU thresholds. It is important to note that precision is high and increases as match IOU is decreased. However, recall - and consequently F_1 - does not increase significantly as this model cannot detect occluded objects.

Match IOU threshold	**Precision**	**Recall**	$\boldsymbol{F_1}$
0.6	0.5309	0.2529	0.3426
0.5	0.6790	0.3254	0.4400
0.4	0.8272	0.4012	0.5403
0.3	0.8765	0.4465	0.5917
0.2	0.9753	0.5097	0.6695
0.1	1.0	0.6090	0.7570

Table 6.1: YOLOv8 evaluation results

[1]github.com/ultralytics/ultralytics

6.4 Evaluation results

6.4.1 Adjusting YOLO resolution

After implementing and thoroughly testing the pixelNeRF-YOLO model to verify that every component is logically correct, I have started tuning the hyperparameters. I experimented with various YOLO loss weights (Table 6.2), ray resolutions, and network sizes, but none of these adjustments led to significant improvements. These tests were conducted to overfit a single scene since the F_1 score remained significantly low, and I was uncertain whether the model was suitable for the problem.

Object loss weight	**Precision**	**Recall**	$\boldsymbol{F_1}$
1	0.0563	0.0325	0.0412
10	0.0622	0.1967	0.0945
20	0.0588	0.2459	0.0949

Table 6.2: Overfitting and object loss weight

The first improvement was achieved by reducing the grid resolution and adjusting the ray limits and resolution. The cell size was modified to 30×30, with a grid size of 16×9 for an image size of 480×270. The ideal ray length was found to be between 1 to 13 metres with a resolution of 128. The best achieved F_1 score is 0.4737 (details are presented in Table 6.3).

This result was achieved by employing the same model size as the pixelNeRF model, which comprises resnet34 with four layers, and five blocks in the NeRF MLP with average pooling after the third layer. The YOLO weights were only modified for the object loss to 20, whereas the others were left unchanged.

Match IOU thres.	**NMS thres.**	**NMS IOU thres.**	**Precision**	**Recall**	$\boldsymbol{F_1}$
0.40	0.54	0.40	0.3564	0.7059	**0.4737**

Table 6.3: Metrics with adjusted YOLO resolution at match IOU 0.4

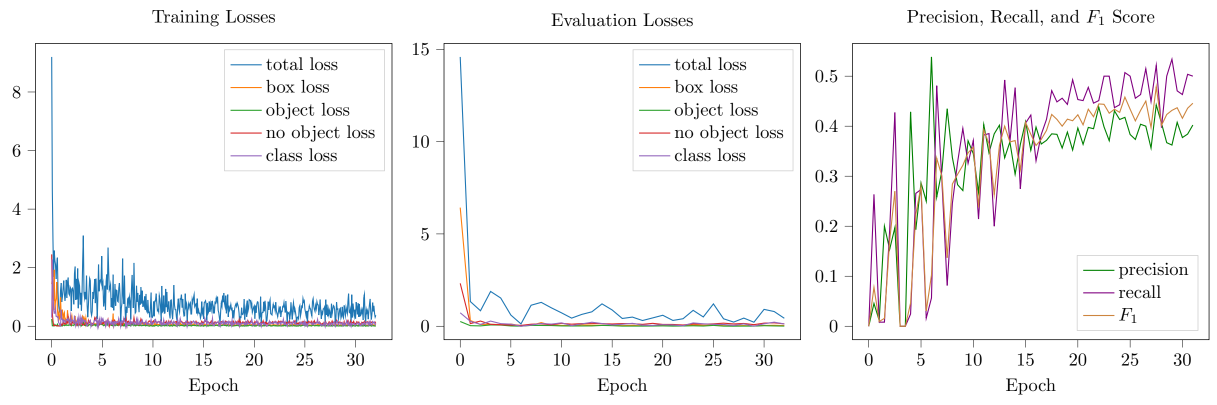


Figure 6.2: Training with adjusted YOLO resolution

The charts in Figure 6.2 show that the losses fell to a minimum between the 10th and the 15th epoch, while the other metrics reached their peak between the 15th and 20th epoch. (For the overfit, the evaluation and the train losses are essentially the same.) All graphs indicate a considerable fluctuation, which is an unfortunate trend in all experiments. The visualizations illustrate that the model was able to learn the scene with very accurate predictions for occluded objects, however, it often lacks precision resulting in many false detections (Figure A.1).

6.4.2 Overfitting and input image size

The most effective enhancement for the overfit training was resizing the input image to 960×540 (50% of its original size) while maintaining the cell size at 30×30. This experiment gave the best results of all the trainings, with an F_1 score of 0.5681 (see details in Table 6.4). Although I would have liked to carry out experiments with bigger input images, VRAM constraints made it unfeasible.

Match IOU thres.	NMS thres.	NMS IOU thres.	Precision	Recall	F_1
0.40	0.60	0.40	0.5530	0.5870	**0.5681**
0.40	0.55	0.40	0.4163	0.7029	0.5229
0.40	0.50	0.40	0.3602	0.8333	0.5030

Table 6.4: Metrics with adjusted input image resolution at match IOU 0.4

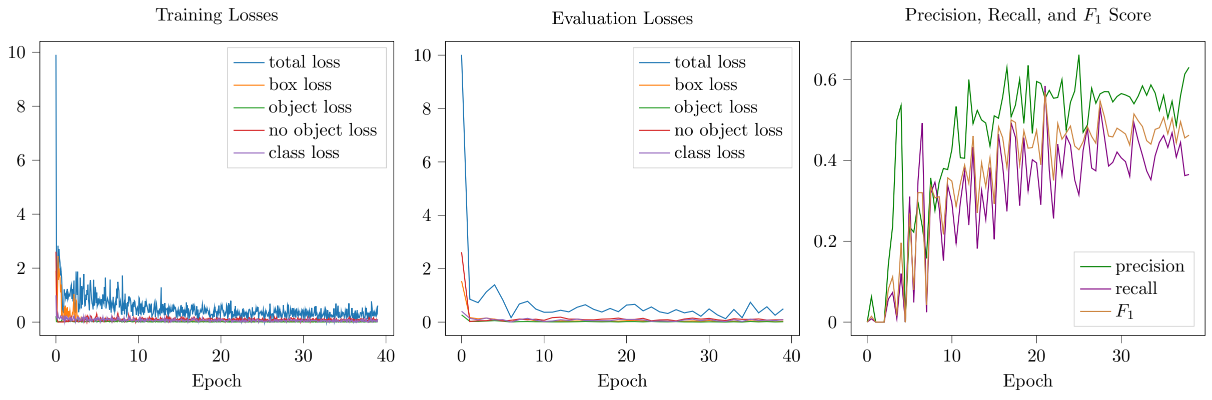


Figure 6.3: Training with adjusted input image resolution

The charts in Figure 6.3 are closely resembling those of the previous experiment. The visuals indicate an improvement in accuracy, though still lacking in some cases (Figure A.2).

6.4.3 Overfitting and additional layer, multiple YOLO scales

Adding an extra layer after the renderer could simplify YOLO feature prediction for the model. This way, the renderer does not need to directly output YOLO features, only the appropriate outputs for the final layer, which can transform them into YOLO features. Therefore, the next experiments targeted this additional layer. As this did not bring any improvement (Table 6.5), I have continued with different experiments.

Match IOU thres.	NMS thres.	NMS IOU thres.	Precision	Recall	F_1
0.40	0.60	0.40	0.6667	0.0161	0.0315

Table 6.5: Best metrics with additional layer

I have implemented object detection for multiple YOLO scales, but the test outcome was unsatisfactory. The possible reason is that I query a single NeRF MLP for all the scales. With different anchors on each scale, but only one MLP, the same network should output different YOLO features for indistinguishable rays, which is impossible to learn. Next, I tried using the same anchors on different grid resolutions, which did not improve the performance either (Table 6.6). This might be due to the fact, that the 30×30 cell size is sufficient, and more scales do not provide additional information. With all the experiments underachieving, I continued with other modifications.

Match IOU thres.	NMS thres.	NMS IOU thres.	Precision	Recall	F_1
0.40	0.42	0.80	0.3650	0.4620	0.4078

Table 6.6: Best metrics with multiple YOLO scales

6.4.4 Training on the entire dataset with ResNet encoder

At this stage, the model delivered satisfactory results with overfitting, so I moved on to real training on the full dataset and evaluation on the test dataset. The same parameters were used as in the overfitting phase, using a resnet34 encoder with 4 layers, 5 blocks in the NeRF MLP, and average pooling after the 3rd layer. The input image size was 50%, and the cell size was 30×30. An additional enhancement was incorporating colour jitter for the input images. Random modifications are applied to the images by altering their brightness, contrast, saturation, and hue. This augmentation technique usually enhances the model's generalization ability while preventing overfitting on the training set.

The training losses reveal that the model achieved low training losses (very close to the previous overfitting performance). However, the evaluation loss did not reduce as significantly, yet was relatively impressive. Precision, recall and F_1 metrics improved during the training but with substantial fluctuations (Figure 6.4).

However, the training presented some challenges, including an increase in training time to 5 hours per epoch, which made it difficult to iterate through different configurations. The final training in this experiment ran for over 15 hours on an Nvidia Titan XP GPU. Fortunately, the most effective overfitting configurations proved to be beneficial for normal training too. The model attained an F1 score of 0.76 at a match IOU threshold of 0.2 (Table 6.7), which surpassed the benchmark of 0.67 at the same threshold (Table 6.8).

The visuals demonstrate that the model frequently identifies objects accurately in full occlusion. However, it occasionally lacks accuracy, and predicts false positives (Figure A.3).

Match IOU threshold	**Precision**	**Recall**	F_1
0.6	0.0373	0.3121	0.0666
0.5	0.0968	0.6029	0.1669
0.4	0.2222	0.8538	0.352
0.3	0.4003	0.9514	0.5635
0.2	0.6159	0.9877	**0.7587**
0.1	0.8299	0.9991	**0.9066**

Table 6.7: PixelNeRF-YOLO metrics with ResNet encoder at NMS threshold 0.4, NMS IOU threshold 0.75

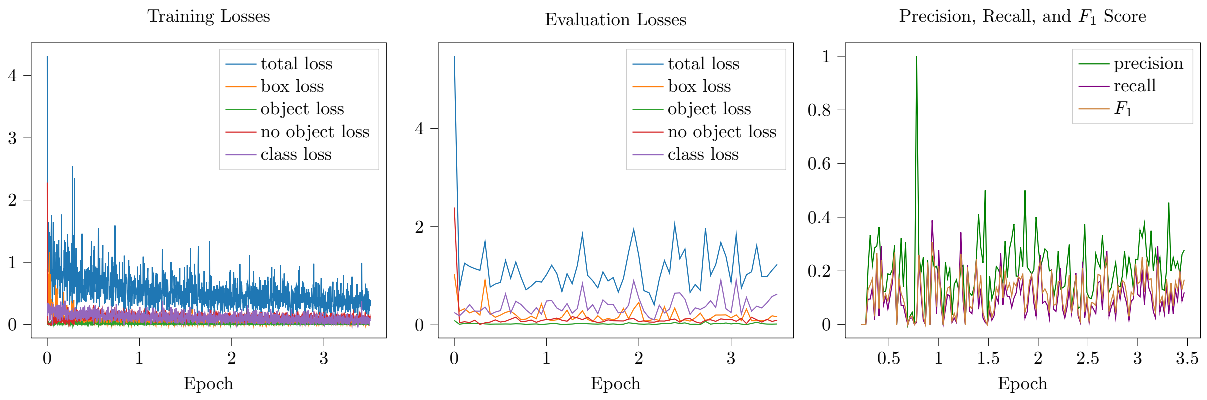


Figure 6.4: Training with ResNet encoder

Model	**Precision**	**Recall**	F_1
Benchmark YOLOv8	0.9753	0.5097	0.6695
PixelNeRF-YOLO	0.6159	0.9877	**0.7587**

Table 6.8: PixelNeRF-YOLO with ResNet encoder compared to benchmark YOLOv8 at match IOU 0.2

6.4.5 Training on the entire dataset with YOLO encoder

The ultimate stage in refining the pixelNeRF-YOLO model involved replacing the encoder with the backbone network of a YOLO model. Specifically, I used the YOLOv7 implementation, the same as in the shared paper with my supervisor (the code is available on GitHub[2]). I utilized the features extracted from the different layers of the network, similarly to ResNet. However, the challenge was that the YOLO backbone network is exceedingly large, preventing me from being able to train (fine-tune) due to memory constraints. Nevertheless, by training the NeRF MLP alone, without training the YOLO backbone (loading pre-trained weights), both the benchmark model and the pixelNeRF-YOLO model with ResNet encoder were outperformed. This model achieved an F_1 score of 0.81 at a match IOU threshold of 0.2 (Table 6.9), surpassing the ResNet encoder which scored 0.76 and the benchmark which scored 0.67 (Table 6.10).

The losses and metrics during the training show almost the same as the model with the ResNet encoder. The training losses reveal that the model achieved low training losses (very close to the previous overfitting performance). However, the evaluation loss did not reduce as significantly, yet was relatively impressive. Precision, recall and F_1 metrics improved during the training but with substantial fluctuations (Figure 6.5).

The visuals demonstrate that the model frequently identifies objects accurately in full occlusion. However, it occasionally lacks accuracy, and predicts false positives, resembling the model with the ResNet encoder (Figure A.4).

Match IOU threshold	**Precision**	**Recall**	**F_1**
0.6	0.0252	0.1135	0.0412
0.5	0.0891	0.3563	0.1425
0.4	0.2450	0.6780	0.3600
0.3	0.4877	0.8753	0.6264
0.2	0.7028	0.9558	**0.8100**
0.1	0.8649	0.9946	**0.9252**

Table 6.9: PixelNeRF-YOLO metrics with YOLO encoder at NMS threshold 0.45, NMS IOU threshold 0.75

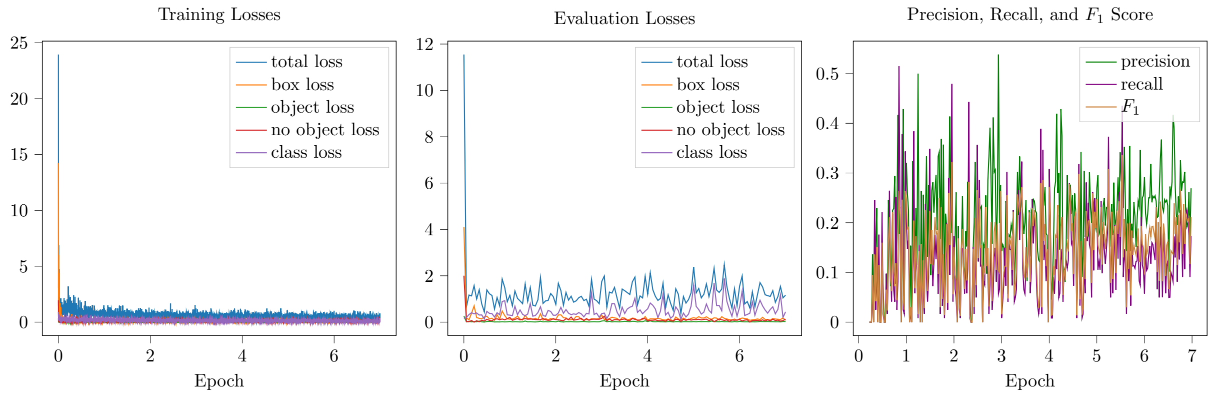


Figure 6.5: Training with YOLO encoder

Model	**Precision**	**Recall**	**F_1**
Benchmark YOLOv8	0.9753	0.5097	0.6695
PixelNeRF-YOLO, ResNet encoder	0.6159	0.9877	0.7587
PixelNeRF-YOLO, YOLO encoder	0.7028	0.9558	**0.8100**

Table 6.10: PixelNeRF-YOLO with YOLO encoder compared to benchmark YOLOv8 at match IOU 0.2

[2] github.com/szemenyeim/NeRF-YOLO

6.5 Comparative analysis

Single-view object detection is an already solved problem. Although improved and faster models are continuously being developed, the fundamentals are already successfully functioning. Accordingly, my objective is not to create a superior object detection model but to expand the field to multi-view detection. Although high precision can be achieved with single-view models, they often face difficulties in recall when dealing with occluded objects. The purpose of the pixelNeRF-YOLO model was to enhance current state-of-the-art object detection networks by detecting occluded objects.

The primary objective when dealing with fully obstructed objects is to recognise them and enable the larger system, such as an autonomous vehicle, to prepare for unforeseen events. The accuracy of the bounding box is not a priority, provided that the model correctly identifies the objects. Therefore, multi-view detection may have a lower match IOU compared to precise single-view detection.

The findings of my work show that the pixelNeRF-YOLO significantly outperforms the state-of-the-art YOLOv8 model, with a match IOU threshold of 0.2 or less Figure 6.6. This shows that the object detection and localization was considerably improved with the new model, with the recall close to 1, meaning almost no missed objects. Meanwhile, standard YOLO performs satisfactorily in high-precision – high match IOU threshold – areas.

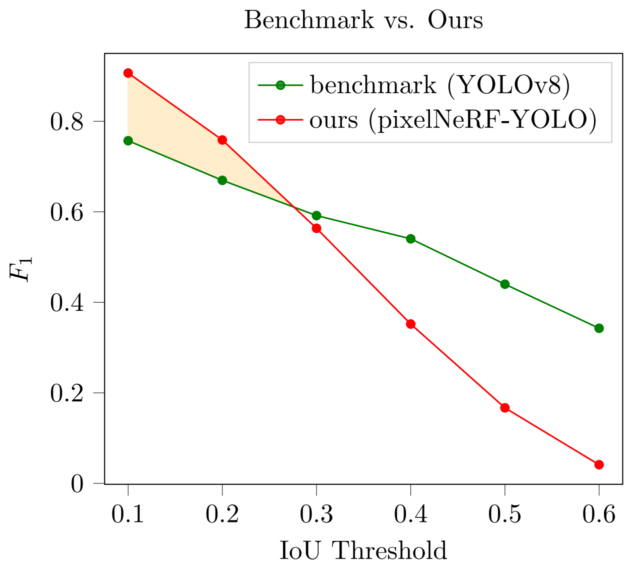


Figure 6.6: Comparing the benchmark YOLOv8 to our model pixelNeRF-YOLO

This novel multi-view approach shows great promise as a research direction since the traditional YOLO models will never improve recall for objects that are not visible. Additionally, I experimented with the pixelNeRF-YOLO model by merging predictions with the YOLOv8 model, which yielded superior results at a 0.3 match IOU threshold compared to deploying YOLOv8 alone (Table 6.11).

Model	**Precision**	**Recall**	**F_1**
Benchmark YOLOv8	0.8765	0.4465	0.5917
PixelNeRF-YOLO + YOLOv8	0.4677	0.8736	**0.6092**

Table 6.11: PixelNeRF-YOLO compared to YOLOv8 at 0.3 match IOU

Using the pixelNeRF-YOLO model involves increased computational load as a consequence of NeRF rendering, in comparison to standard YOLO networks. Therefore, for practical implementation, it would be advantageous to operate the multi-view model periodically in challenging scenarios where additional views are available to support object detection performance and continue to rely on the traditional YOLO model in simple situations.

Chapter 7

Conclusion

In summary, my study consisted of three parts. Firstly, a synthetic dataset was generated featuring a high number of occluded objects to train and test new models. Secondly, the multi-view pixelNeRF-YOLO object detection model was implemented, trained, and optimized. This new model surpasses the existing state-of-the-art YOLOv7, yielding a F_1 score increase of 0.14 by identifying occluded objects. Although my model's precision may not be as advanced as single-view models, it is a valuable addition to the current object detection frameworks. Moreover, I have conducted a comprehensive analysis comparing it to YOLOv7 (YOLOv8) and demonstrated the benefits of using a combined model.

This new model clearly showed that it is capable of outperforming the current state-of-the-art solution. Therefore, it can be used for multi-view object detection with a significantly increased F_1, and can also be utilized as a companion for current implementations, by aiding crucial weaknesses. All in all, this new model's performance makes it an important research contribution.

The synthetic dataset also serves as a good foundation for subsequent models, while the pixelNeRF-YOLO model shows a promising direction in multi-view object detection, with the research contribution of outperforming the current state-of-the-art implementation.

7.1 Future work

By outperforming current models, this study demonstrates the potential of multi-view object detection as a promising research direction. Nonetheless, it represents only an initial step in this area, so there are many opportunities for future work and potential improvements. Herein, I outline the most prominent opportunities that emerged during the development of this work.

1. Further explore the model towards larger input images.
   Due to resource limitations in this project, it remains a possibility that the model's performance could be improved by using higher-resolution input images.

2. Train and test the model using real-world data.
   Currently, there is a lack of real-world data for multi-view object detection, as it is a tedious task to compile such a dataset. As a result, future research should focus on advancing in this direction.

3. Replace the weighted average along camera rays with a transformer model.
   This way the simple, yet not necessarily most optimal aggregation can be enhanced, thereby expectedly increasing precision. By employing an attention model, the original maximum aggregation can be refined and the outputs can be improved.

4. Develop a protocol for autonomous vehicles to share data for multi-view object detection.
   While the model's test performance is impressive, its utility in driver assistance systems is a separate matter. For such systems to function effectively, the vehicles must be capable of communicating the data required for multi-view object detection.

5. Extend the model to predict distances to detected objects.
   The underlying NeRF MLP could not only predict the bounding boxes but also their corresponding distances, thereby leading to a more comprehensive understanding of complex 3D scenes and object detection.

Chapter 8

Bibliography

[1] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *CoRR*, abs/2004.10934, 2020.

[2] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[3] Tianshi Gao, Benjamin Packer, and Daphne Koller. A segmentation-aware object detection model with occlusion handling. In *CVPR 2011*, pages 1361–1368, 2011.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[6] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[7] James T. Kajiya and Brian P Von Herzen. Ray tracing volume densities. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, page 165–174, New York, NY, USA, 1984. Association for Computing Machinery.

[8] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[9] Chuyi Li, Lulu Li, Hongliang Jiang, Kaiheng Weng, Yifei Geng, Liang Li, Zaidan Ke, Qingyuan Li, Meng Cheng, Weiqiang Nie, Yiduo Li, Bo Zhang, Yufei Liang, Linyuan Zhou, Xiaoming Xu, Xiangxiang Chu, Xiaoming Wei, and Xiaolin Wei. Yolov6: A single-stage object detection framework for industrial applications, 2022.

[10] Yongjun Li, Shasha Li, Haohao Du, Lijia Chen, Dongming Zhang, and Yao Li. Yolo-acn: Focusing on small target and occluded object detection. *IEEE Access*, 8:227288–227303, 2020.

[11] Tsung-Yi Lin, Piotr Dollar, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[12] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020.

[13] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458, 2015.

[14] Shaohua Qi, Xin Ning, Guowei Yang, Liping Zhang, Peng Long, Weiwei Cai, and Weijun Li. Review of multi-view 3d object recognition methods based on deep learning. *Displays*, 69:102053, 2021.

[15] Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred Hamprecht, Yoshua Bengio, and Aaron Courville. On the spectral bias of neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 5301–5310. PMLR, 09–15 Jun 2019.

[16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[17] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[18] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[19] Márton Szemenyei and Nándor Kőfaragó. Nerf-yolo: Detecting occluded objects via multi-view geometric aggregation. In *2023 International Symposium on Measurement and Control in Robotics (ISMCR)*, 2023.

[20] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7464–7475, June 2023.

[21] Alex Yu, Vickie Ye, Matthew Tancik, and Angjoo Kanazawa. pixelNeRF: Neural radiance fields from one or few images. In *CVPR*, 2021.

Appendix A

Training visuals

The visualizations contain images from left to right: 3 encoded views, the selected target view, ground truth object detection, prediction.

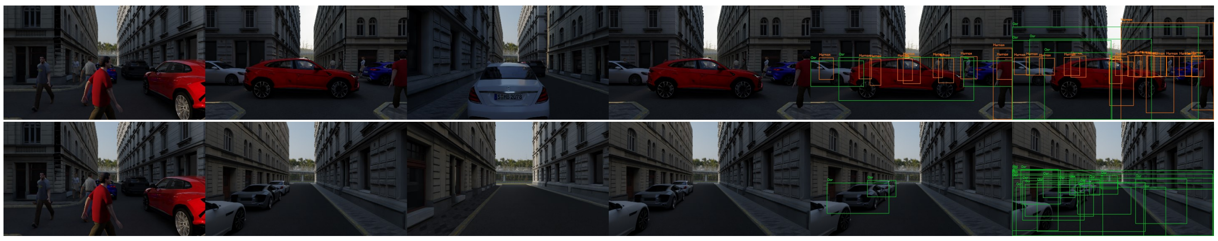


Figure A.1: Visualizations from overfitting pixelNeRF-YOLO after adjusting YOLO resolution

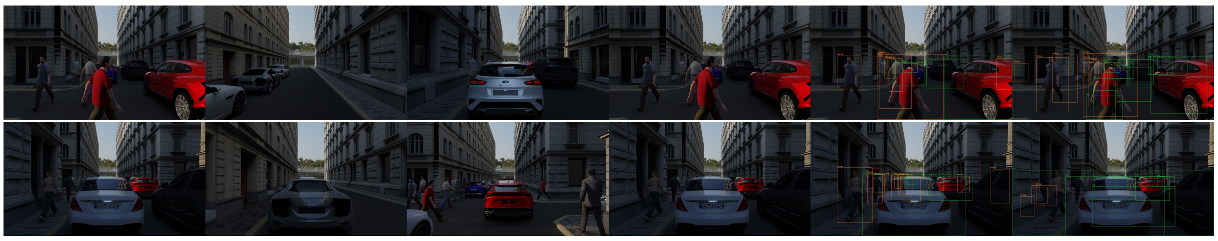


Figure A.2: Visualizations from overfitting pixelNeRF-YOLO after adjusting input image resolution

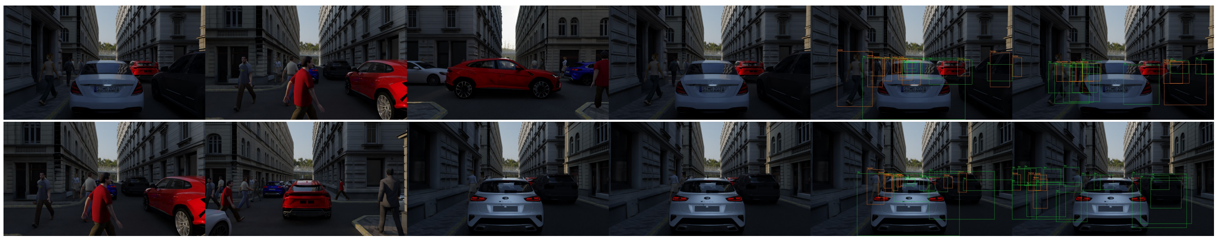


Figure A.3: Visualizations from pixelNeRF-YOLO with ResNet encoder

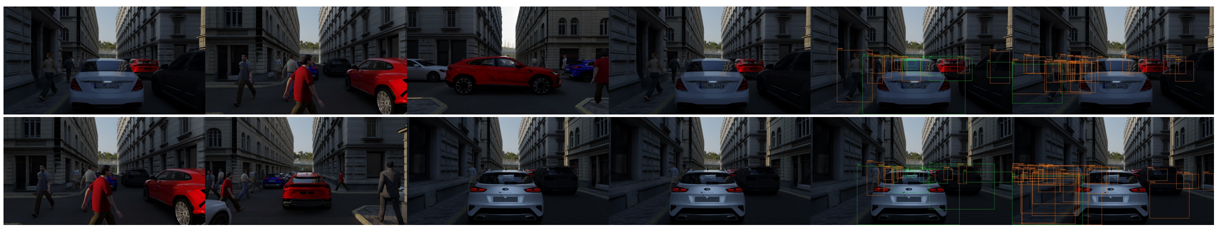


Figure A.4: Visualizations from pixelNeRF-YOLO with YOLO encoder