MODERN TECHNIQUES IN MACHINE LEARNING WITH IMAGES

MODERNAS TÉCNICAS NO APRENDIZADO DE MÁQUINA COM IMAGENS

TÉCNICAS MODERNAS EN EL APRENDIZAJE AUTOMÁTICO CON IMÁGENES

https://doi.org/10.56238/sevened2026.019-048

Bruno Seki Schenberg[1], Rafael Colen de Almeida[2], Rogério de Oliveira[3]

ABSTRACT

This chapter presents modern Machine Learning techniques applied to computer vision, divided into two complementary fronts: the processing of 2D images and videos and the classification of three-dimensional data (3D images). The first has strong applications in security and product sales, while the second is fundamental for areas such as the parts manufacturing industry and medical image processing. Thus, the first part of the chapter explores the practical use of the YOLO ecosystem (specifically YOLO26) for object detection and scene classification, covering the implementation of inference pipelines, data annotation, fine-tuning, and data augmentation techniques to mitigate training overfitting. In the second part of the chapter, the focus is on the challenges involved in processing 3D spaces, detailing geometric representation processes (meshes, point clouds, and voxel grids). The chapter also presents the construction and preprocessing of a volumetric classifier from scratch, using the ResNet3D-18 architecture on the ModelNet10 dataset, along with a critical analysis of the model's performance.

Keywords: Computer Vision. Machine Learning. Detection and Classification. Video. 3D Images.

RESUMO

Este capítulo apresenta técnicas modernas de Aprendizado de Máquina aplicadas à visão computacional, dividindo-se em duas frentes complementares: o processamento de imagens e vídeos 2D e a classificação de dados tridimensionais (imagens 3D). A primeira, encontra forte aplicação em segurança e em vendas de produtos e, o segundo, fundamental para, por exemplo, a indústria de peças e o tratamento de imagens médicas. Desse modo, a primeira parte do capítulo explora o uso prático do ecossistema YOLO (especificamente o YOLO26) para detecção de objetos e classificação de cenas, abrangendo a implementação de pipelines de inferência, anotação de dados, fine-tuning e técnicas de data augmentation para

[1] Master's student in Applied Computing. Universidade Presbiteriana Mackenzie (UPM). Brazil.
E-mail: schenbergbruno@gmail.com Orcid: 0009-0002-8580-4827
Lattes: http://lattes.cnpq.br/5591023263193250
[2] Master's student in Applied Computing. Universidade Presbiteriana Mackenzie . Brazil.
E-mail: rafaeljbi@gmail.com Orcid: 0009-0000-3917-1016
Lattes: http://lattes.cnpq.br/3721492091376275
[3] Master's student in Applied Computing. Universidade Presbiteriana Mackenzie. Instituto Mauá de Tecnologia (IMT). Brazil. E-mail: rogerio.oliveira@{mackenzie.br Orcid: 0000-0001-5352-4420
Lattes: https://lattes.cnpq.br/3067732992972770

mitigar sobreajustes do treinamento. Na segunda parte do capítulo, o foco é sobre os desafios no tratamento de espaços 3D, detalhando os processos de representação geométrica (malhas, nuvens de pontos e grades de voxels). Apresenta-se aqui a construção e o pré-processamento de um classificador volumétrico do zero, utilizando a arquitetura ResNet3D-18 sobre o dataset ModelNet10, com uma análise crítica do desempenho do modelo.

Palavras-chave: Visão Computacional. Aprendizado de Máquina. Detecção e Classificação. Vídeo. Imagens 3D.

RESUMEN

Este capítulo presenta técnicas modernas de Aprendizaje Automático aplicadas a la visión por computadora, dividiéndose en dos frentes complementarios: el procesamiento de imágenes y videos 2D y la clasificación de datos tridimensionales (imágenes 3D). La primera tiene una fuerte aplicación en seguridad y en la venta de productos, mientras que la segunda es fundamental para, por ejemplo, la industria de piezas y el tratamiento de imágenes médicas. De este modo, la primera parte del capítulo explora el uso práctico del ecosistema YOLO (específicamente YOLO26) para la detección de objetos y la clasificación de escenas, abarcando la implementación de pipelines de inferencia, anotación de datos, fine-tuning y técnicas de aumento de datos para mitigar el sobreajuste durante el entrenamiento. En la segunda parte del capítulo, el enfoque se centra en los desafíos relacionados con el tratamiento de espacios 3D, detallando los procesos de representación geométrica (mallas, nubes de puntos y rejillas de vóxeles). También se presenta la construcción y el preprocesamiento de un clasificador volumétrico desde cero, utilizando la arquitectura ResNet3D-18 sobre el conjunto de datos ModelNet10, junto con un análisis crítico del desempeño del modelo.

Palabras clave: Visión por Computadora. Aprendizaje Automático. Detección y Clasificación. Video. Imágenes 3D.

1 INTRODUCTION

In the current scenario of computer vision, Machine Learning models based on deep convolutional neural networks (CNNs) almost completely predominate today [Bhatt et al. 2021]. But the transition from theory to successful real-world application of these models—such as analyzing *in-the-wild* videos—or processing geometries from industrial parts and medical examinations—requires more than just instantiating a basic architecture. It requires mastery of complementary techniques and robust workflows.

This chapter introduces the main concepts, provides intuition, and provides a *practical pipeline* of these modern techniques, as a starting point for the student or researcher who needs to use these techniques in a real problem or research project. We initially explore the state of the art in efficient (fast) image processing using the YOLO26 [Ultralytics 2026] ecosystem for images, demonstrating how to apply pre-trained models, specialize them for new data, and make them resilient to variations in the environment. Next, we extend the two-dimensional grid of 2D images to the processing of three-dimensional images in spatial models. From the ModelNet10 dataset [Wu et al. 2015], we illustrate the preprocessing pipeline required by 3D data and the adaptation of CNNs to operate over spatial volumes with ResNet3D-18 networks [Hara et al. 2018].

All the codes and materials used in this chapter can be found in the https://github.com/Capitulo-Aprendizado-de-Maquina-Imagens/.github repository. They are an integral part of this chapter and it is recommended that the reader access this material in addition to the reading of the chapter.

2 YOLO AND IMAGE AND VIDEO PROCESSING

Introduced in 2015 by Joseph Redmon and his team, YOLO (*You Only Look Once*) brought a revolutionary approach to computer vision based on Convolutional Neural Networks (CNNs) [Redmon et al. 2016]. Instead of slicing the image and analyzing it into multiple cutouts like the classic models, YOLO applies a single neural network over the entire image at once. It divides the scene into a mathematical grid and simultaneously predicts where the objects are. In the most widely used detection models, this is done by means of bounding *boxes* that calculate the probability of each class.

This "single look" enabled real-time detection, a critical factor for video processing. In general terms, a video is a set of several images in sequence. These images, or frames, are called *frames*. Since a typical video runs between 30 and 60 *frames* per second (FPS), slow models generate high latency and "choke" processing. YOLO's optimized architecture allows it

to process tens of *frames* per second, making it possible to analyze *in-the-wild* videos (from the real world, without positioning or lighting control), following objects as the scene unfolds.

To achieve this efficiency, YOLO models are already pre-trained on the COCO (*Common Objects in Context*) dataset, a gigantic dataset, widely used by the AI community containing more than 330,000 annotated images with everyday objects divided into 80 categories or classes (such as cars, people, dogs, chairs, bottles, etc.) [Lin et al. 2014]. In other words, the model is born with a robust base for extracting characteristics, knowing the basic world.

2.1 YOLO ECOSYSTEM

The original architecture has evolved and today the YOLO family is an ecosystem capable of solving different computer vision problems, each with its own specialty. Table 1 summarizes how the network "sees" according to the chosen model.

It is important to note that advances in computer vision occur at a very accelerated pace and it is possible that by the time you are reading this chapter there are already newer models of YOLO available. Don't worry, the codes are all adaptable for use by any existing template.

Table 1

Comparison of YOLO26 models, main functions and ways of viewing.

Model	Main Function	How the network "sees"
YOLO26-cls	Classification	Evaluates the entire image and indicates which class predominates in the scene.
YOLO26	Object Detection	Answers what the object is and where it is, drawing bounding *boxes*.
YOLO26-sec	Instance Segmentation	Draws the exact outline of the object found, pixel by pixel.
YOLO26-pose	Pose Estimation	It identifies key points, focusing on structure and movement, such as human joints.
YOLO26-obb	Guided Detection	Draws rotated boxes, useful for aerial images where objects appear at varying angles.

In Table 1 we have seen five models and their applications. For the practical scope of this chapter, which is focused on extracting relevant information from real-world videos and generating an intuition about how YOLO works, we will explore the two fundamental models: the Classifier (YOLO26-cls) and the Detector (YOLO26).

2.2 AN INFERENCE PIPELINE WITH YOLO

The practical application of YOLO models follows a unified workflow, often using the official Ultralytics library [Ultralytics 2026]. The steps consist of instantiating the model, loading the video, and performing inference.

2.2.1 Example: Classifier, YOLO26-cls

Since this is a classifier model, it only tells you what's being found in the entire *frame*, i.e., it doesn't specify where the object is being found.

Code 2.1 is a simple example of implementing and using the classifier model directly on any video. You can access the directory of that project in the GitHub repository and run it on your machine locally using any video on your computer. A sample video can be found in the project directory.

Table 1

Code 2.1. Basic inference pipeline using the classifier model YOLO26n-cls

```
from ultralytics import YOLO
#1. Instantiate the pretrained model model = YOLO('yolo26n-cls.pt')
# 2. Upload the video (replace the file path below)
results = model(source=r'C:\caminho_do_video\video_exemplo.mp4', show=True, stream= True)
#3. Inferencia for frame in resultados:
# Returns the overall probability of the scene print(frame.probs.top1)
#**IMPORTANT**: To finish the video press the "Q" key.
```

Figure 2 demonstrates how the classification model runs, showing the probability of the classes in the upper left corner.

Figure 2

Example of video running with the ranking model

2.2.2 Example: Detector, YOLO26

With the detector model, the boxes point to the multiple objects within each frame of the video, which can be more interesting if the challenge is to know where the object is in the scene.

Using the same Code 2.1, let's adapt lines 4 and 13, calling the detector model and adding the *bounding boxes*, as we see in Code 2.2.

Table 2

Code 2.2. Basic inference pipeline using the YOLO26n detector model

```
from ultralytics import YOLO
#1. Instantiate the pretrained model (changed to the detector) model = YOLO('yolo26n.pt')
# 2. Upload the video (replace the file path below)
results = model(source=r'C:\caminho_do_video\video_exemplo.mp4', show=True, stream= True)
#3. Inferencia for frame in resultados:
# Returns the bounding box of the print(frame.boxes.xyxy) object
#**IMPORTANT**: To finish the video press the "Q" key.
```

Figure 3 shows how a detector model behaves, finding the objects and drawing the *bounding boxes* in the scene.

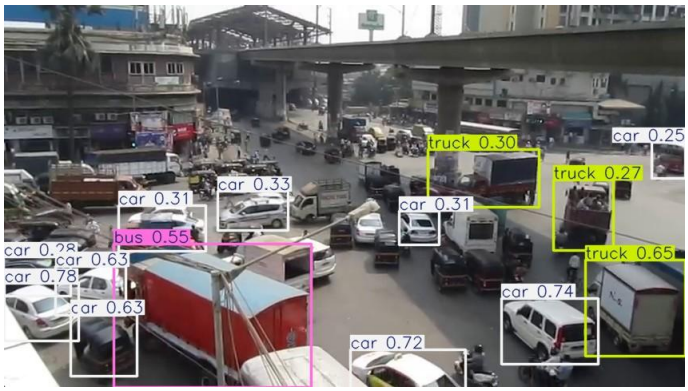
That's it, you've just applied two computer vision models and you can follow how they work along with the video!

2.3 *FINE-TUNING* AND DATA PREPARATION

Fine-tuning is a technique that allows you to take advantage of the knowledge base acquired in the COCO dataset to teach new categories specific to YOLO. When we do this, there is a transfer of knowledge, but the original classes are lost. This means that when we apply *fine-tuning* we use the knowledge of the neural network, but specify it for the new classes or categories that we want our model to recognize. As a result, the way you prepare this data and train the neural network changes drastically depending on your goal: classifying the entire image in the scene or detecting specific objects within it.

Figure 3

Example of video running with the detection model



2.3.1 Classification Flow

In classification tasks, the neural network learns what an object is through images separated into categories.

i. Dataset Structure

Preparing data for classification does not require manual markings on the image. The very organization of the folders defines the classes. The industry standard is to perform a *split* (manual division of the data) separating it into Training, Validation and Testing, usually in a ratio of 80% for training and 20% for validation and testing, or 70/20/10 depending on the total volume.

Figure 4 shows the directory structure and hierarchy required by YOLO for classifier training.

ii. Training and Evaluation

With folders organized, training only requires the model to point to the main directory. In addition, it is at this stage that YOLO itself applies *Data Augmentation techniques* under the hood (such as rotations and brightness variations) to prevent the network from "decorating" the images. By default YOLO does this automatically, but it is possible to adjust the *Augmentation* parameters, as we will explain in section 1.6.

Code 2.3 shows how to train a classifier model with 50 epochs, 224px image size, and a batch of 16 simultaneously processed images. As training a neural network is a costly task for a standard computer, we recommend that *fine-tuning* be done in Google's Colab tool, where there are options to select environments more suitable for training neural networks (with options that use T4 or A100 GPUs). These environments will bring a higher training speed to your computer vision projects with images.

Figure 4

Recommended Directory Structure for Classification Model Training

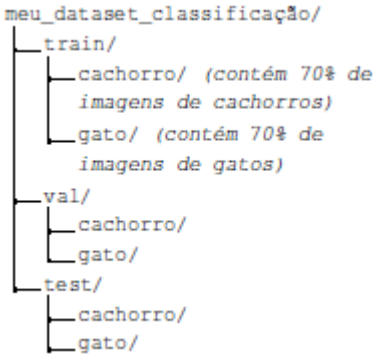


Table 3

Code 2.3. Fine tuning of classification model

```
from ultralytics import YOLO
# Loads pre-trained classification model = YOLO('yolo26n-cls.pt')
# Start training results = model.train(
data=r'caminho_do_dataset_classificacao', # Root folder with train/val/test
epochs=50, # Number of times the network will see the entire dataset
imgsz=224, # Image size for sorting batch=16 # Batch of images
processed simultaneously)
```

To evaluate the classifier model, we observed the drop in the *Loss Function* (which measures the mathematical error of the network), as the drop increases in each iteration, we understand that the model is converging. In addition, we track accuracy (Top-1 and Top-5), which indicates the percentage of times the network has registered the correct class. In this case, the higher the accuracy, the more correct answers the model recorded.

2.3.2 The Detection Flow

When we need to find the exact location of something in a scene, folders alone are not enough. In the case of detector models, we need to teach the coordinates of objects to the network.

i. *Annotation*, Roboflow

For detection, each image must be accompanied by a text file containing the coordinates of the bounding *boxes*. The legwork of drawing these boxes is called *Annotation*, and platforms

like Roboflow simplify this process. In the interface, we draw a very tight rectangle around the object, associate it with a class and the tool exports the dataset already structured and divided.

Figure 5 illustrates the annotation process on the platform. After the object is selected, the class name is entered.

ii. Training and Evaluation

Unlike the classifier that reads folders, the detector uses a data.yaml configuration file generated during annotation to locate the images and their labels. Code 2.4 shows how to *fine-tune* a model. Note that the main difference is that the dataset is referenced with the data.yaml file. In addition, detector models are usually more expensive than classifiers and their training can take longer, mainly because they require higher resolutions in training.

Figure 5

Example of annotation on the Roboflow platform

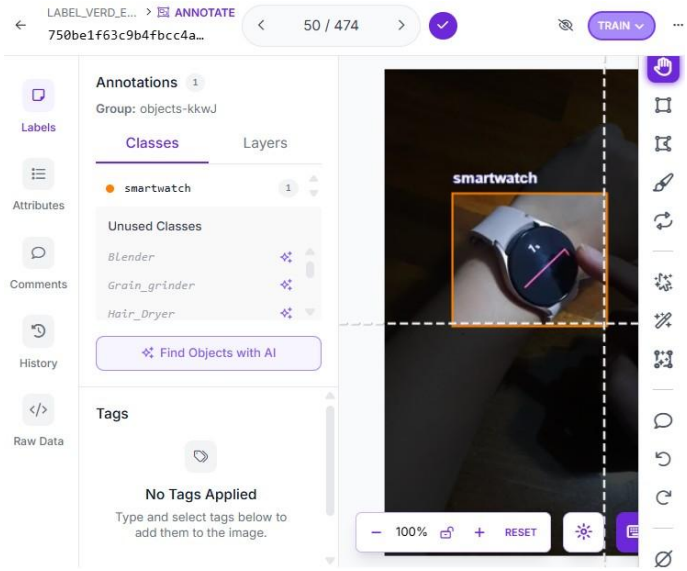


Table 4

Code 2.4. Fine tuning of Detection model

```
from ultralytics import YOLO
# Loads the pre-trained detection model = YOLO('yolo26n.pt')
# Start training results = model.train( data='path/data.yaml', # File generated in annotation epochs=50, imgsz=640, # Detectors require higher resolutions batch=16)
```

In detection, in addition *to the Loss Function*, the main evaluation metric is mAP (*Mean Average Precision*). mAP simultaneously evaluates whether the model has hit the class and whether the box it draws correctly overlaps the real object (another metric known as *Intersection over Union* - IoU).

2.4 DATA AUGMENTATION

One of the biggest challenges in machine learning is *overfitting*, which occurs when the model becomes an expert at recognizing only the exact images of its training set, but fails miserably when finding the same object at a different angle or under a shadow in the real world.

To mitigate this, we use *Data Augmentation* . This technique consists of applying random mathematical transformations to the original images during the training process, creating infinite variations without having to take new photos or make new notes [Géron 2019].

2.4.1 Types of *Data Augmentation*

There are numerous possibilities for increasing data, but we will address the most common ones in this subsection.

- Geometric Transformations: Flips, rotations, and zooms. This teaches the model that a product upside down or seen from afar remains the same product.
- Color Transformations: Changes in brightness, contrast, and saturation. Essential for *in-the-wild* videos, where lighting is not controlled like in a studio.
- Quality Filters: Addition of Gaussian noise or blur. This prepares the network to handle low-resolution cameras or fast movements that blur the frame.

2.4.2 Implementation in YOLO

The Ultralytics library already includes native *augmentation* settings that are automatically applied during *fine-tuning* of a model. However, we can customize the intensity of these transformations directly in the hyperparameters of the train() command.

In your training code, you can add the parameters from Code 2.5 to make the model more resilient.

Table 5

Code 2.5. Custom Data Augmentation Application Example

```
        # Example of fine-tuning with augmentation parameterization
results=model.train( data='data.yaml', epochs=50, degrees=15.0, # Rotates the image up to 15 degrees flipud=0.5, # 50% chance to rotate the image vertically brightness=0.3, # Varies brightness by up to 30%
        blur=0.1,       # Applies random blur to simulate motion device=0)
```

This technique is what allows your model to detect products in a video from a supermarket aisle or on a traffic camera, even if the camera is swaying or the light is blown out in some spots.

2.5 APPLYING THE CUSTOM MODEL

After training (both a classification and detection model), YOLO automatically saves a copy of the network's best moment, often referred to as best.pt.

To apply this newly acquired knowledge in the real world, we went back to the inference pipeline seen in 1.4 and adapted in Code 2.6. The only difference is that instead of loading the factory model from Ultralytics, we load our file.

Table 6

Code 2.6. Inference with trained models

```
        from ultralytics import YOLO
        # We replaced the original model with our newly trained model
modelo_customizado = YOLO(r'C:\caminho_do_modelo_treinado\best.pt')
        # We apply it to real-world video
        results                                                    =
modelo_customizado(source=r'C:\caminho_do_video\video_exemplo.mp4',
show=
        True, stream=True)
        for frame in results:
        print(frame.boxes.xyxy) # For detection #print(frame.probs.top1) # For classification
```

2.6 NEXT STEPS WITH YOLO

Throughout these sections you have seen how YOLO26 simplifies and powers computer vision projects, from model loading to specialization and practical application in videos. Despite its innovations and efficiency, it is important to remember that, like any technology, YOLO26 can present limitations in very specific scenarios or highly specialized tasks, requiring adjustments or complementary solutions. Still, its modern architecture and ease of use open doors to innovative applications, making you able to explore new possibilities and create increasingly intelligent and impactful solutions.

3 3D IMAGE CLASSIFICATION

So far we have worked with images and videos, that is, with data organized in 2D pixel grids. But a growing share of modern computer vision applications must deal with objects in three-dimensional space: CT scans and MRIs in the medical field, industrial parts scanned for quality control, environments captured by LiDAR sensors in self-driving cars, and CAD models used in engineering and architecture. In all these cases, "flattening" the data in a single photo means losing the very information that matters: the shape of the object in space.

For this kind of problem, we need models that see in three dimensions. In this section we will build a 3D object classifier using the ResNet3D-18 architecture on top of the ModelNet10 dataset, a classic benchmark containing CAD models of everyday furniture and objects [Wu et al. 2015]. The path we are going to take involves an extra step that does not exist in the 2D world: before reaching the neural network, the 3D data needs to be converted into a format that CNN can process. It is precisely this pre-processing — transforming a mesh of triangles into a grid of voxels — that occupies a good part of our pipeline.

3.1 3D REPRESENTATIONS

There are three main ways to represent a three-dimensional object on the computer, each with advantages and disadvantages:

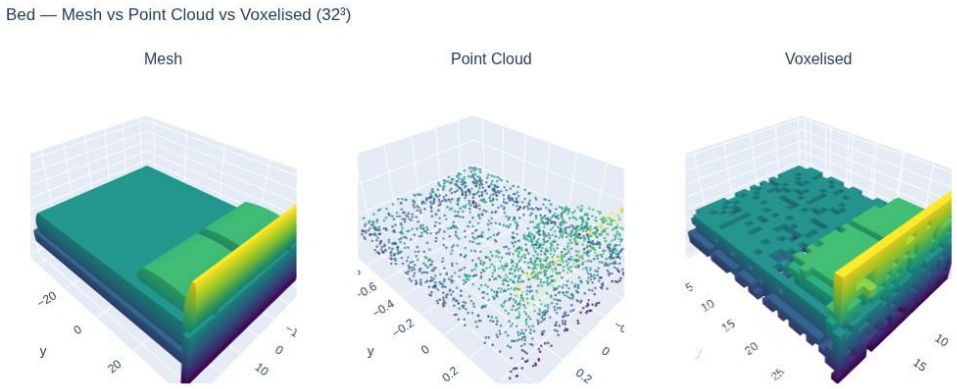
- Mesh: The object is described by vertices (points) connected by triangular faces. It is the native format of 3D modeling software and .off, .obj, and .stl files. Compact and precise, yet irregular — each model has a different number of vertices and faces.
- Point *cloud*: The object is represented only by the coordinates (*x,y,z*) of points on its surface, with no connectivity information. It is the natural output format of LiDAR sensors and 3D scanners.

- Voxel *grid*: space is divided into small cubes (the "3D pixels") and each cube receives the value 1 if it contains the object's surface and 0 if it is empty. It is the closest representation to a digital image, and precisely for this reason it is the one that fits directly into a CNN.

Figure 3.5 illustrates these three representations for the same ModelNet10 object. Notice how the mesh preserves fine details with little data, the point cloud samples the unstructured surface, and the voxel grid "pixelizes" the object into a regular cube.

Figure 6

Comparison between mesh, point cloud and voxel grid for the same ModelNet10 3D model



The choice of representation determines the type of neural network we can use. Networks that operate directly over point clouds, such as PointNet, exist and are quite powerful, but they require special architectures. For classifiers based on conventional CNNs, the most direct path is what the literature calls a volumetric approach: voxelizing the object and treating the resulting cube as a "3D image" [Maturana and Scherer 2015]. That is the approach we will take here.

3.2 THE MODELNET10 DATASET

ModelNet10 is a subset of the ModelNet project, launched by Princeton University in 2015 [Wu et al. 2015]. It brings together about 5,000 clean and aligned CAD models, distributed in 10 categories of furniture and household objects: bathtub, bed, chair, desk, dresser, monitor, bedside table, sofa, table and toilet. Because of its manageable scale and balance between classes, it is often called the "MNIST of 3D" — a didactic benchmark before moving on to larger datasets such as ModelNet40 or ShapeNet.

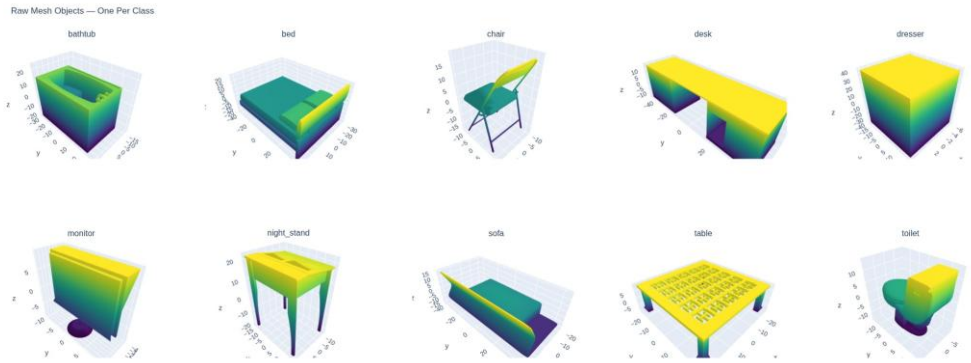
Each model is stored in OFF (*Object File Format*) format, a plain text file: the first line is the OFF header, followed by the vertex, face, and edge counters. Then come the coordinates of each vertex and, finally, the definition of each triangular face, indicating which vertices compose it.

The dataset already comes with the division between training and test ready, organized into subfolders by class (one train/ and one test/ folder within each category). Since there is no separate validation folder, the validation split is done from the training set before training. The big difference in relation to the 2D case seen so far is the content of the files: instead of .jpg images or .png ready to be fed to the network, we have meshes that need to be processed.

Figure 3.6 shows an example of each of the 10 classes in the dataset, rendered from the original .off meshes. Notice the variety of geometries: thin-walled objects (chair, table), massive volumes (dresser, monitor), shapes with cavities (bathtub, toilet) and flat tops on supports (table, desk). This diversity explains why some classes will be easy for the network to learn and others not so much—we'll come back to this point in error analysis.

Figure 7

One example per ModelNet10 class, rendered directly from the original .off meshes. The colors encode the height (z) of each vertex



3.3 MESH AND VOXEL: 3D DATA PREPROCESSING

As we have seen, a mesh cannot be fed directly to a CNN: the number of vertices varies from model to model, and the network expects a fixed-size input arranged in a regular grid. We then need a pre-processing pipeline that converts the mesh into a grid of voxels with standardized resolution. This pipeline has three steps: point-on-surface sampling, normalization, and voxelization.

3.3.1 Surface Point Sampling

The first step is to generate a point cloud from the mesh, with a fixed number of points for all models (in our case, 3,000). The naïve way to do this would be to choose vertices at random, but this would pose a problem: regions with many small triangles would be overrepresented, while large, flat regions would be almost empty.

The solution is area-weighted sampling: we draw triangles with probability proportional to their area, and for each triangle chosen, we generate a random point within it using barycentric coordinates. This ensures even coverage of the surface, regardless of how it has been tessellated.

3.3.2 Standardisation

CAD models come in arbitrary scales and positions—one can be centered at the origin with 1 unit in height, another offset, and with 100 units. In order for the network to learn the shape of the object and not its absolute scale, we normalize the point cloud to fit into a unit cube: we subtract the minimum from each axis (so the box starts at the origin) and divide it by the longest length (so the largest dimension becomes 1.0).

3.3.3 Voxelization

With the cloud normalized to $[0,1]^3$, the voxelization itself is straightforward. We define a resolution R (e.g., 32 voxels per edge) and map each point (*x,y,z) to an integer index* (i, j,k) *by multiplying its coordinates by* R−1 and rounding. We mark each cell visited with 1.0 and leave the others at 0.0. The result is an *occupancy grid*: a binary cube where 1 indicates the presence of a surface and 0 indicates empty space.

Code 3.7 shows the implementation of this final step: the inputs are a point cloud ($N,3$) and the desired resolution, and the output is a *numpy array* (R,R,R) ready to be stacked in batches and sent to the network. The logic is simple — multiply the coordinates by R−1, round to integer, and mark each cell visited with 1.0.

Table 7

Code 3.7. Voxelization of a point cloud into a fixed-resolution occupancy grid.

```
        def voxelize_point_cloud(points, resolution):
        """Converts a cloud (N, 3) into a binary voxel grid (R, R, R)."""" grid =
np.zeros((resolution, resolution, resolution), dtype=np.
        float32)
        if points.shape[0] == 0:
        return grid
        #1. Translated to the origin (each axis starts at 0) shifted = points -
points.min(axis=0)
        #2. Scale to fit [0, 1]^3 (largest dimension = 1.0) max_extent =
float(shifted.max()) if max_extent == 0.0:
        Return Grid # Degenerate Cloud (All Points Equal)
        normalized = shifted / max_extent
        #3. Quantizes on integer indexes and marks occupied cells indexes =
(normalized * (resolution - 1)).astype(np.int32) indexes = np.clip(indexes, 0,
resolution - 1)
                grid[indices[:, 0], indices[:, 1], indices[:, 2]] = 1.0 return grid
```

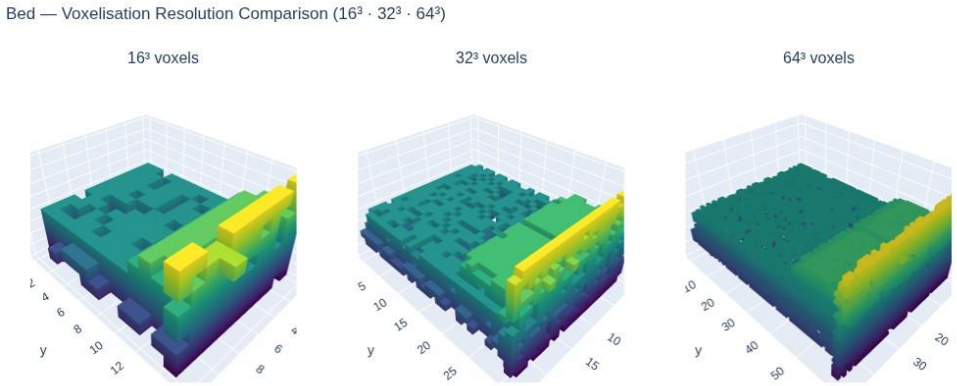
3.4 THE RESOLUTION TRADE-OFF

The grid resolution is the most important hyperparameter of this step, and it involves a *severe tradeoff*. With R = 16, each model fits in only 163 = 4.096 voxels — quick to process, but fine details (chair legs, table legs) may disappear. With R = 64, we gain 643 = 262.144 voxels is much more detail, but memory consumption and training time increase in $O(R3)$. Doubling the resolution multiplies the cost by eight.

Figure 3.7 shows the same voxelized object at three different resolutions. For ModelNet10, values between 323 and 643 are the most common equilibrium point in the literature [Maturana and Scherer 2015]. In the rest of this chapter we will use 323, enough to distinguish the 10 classes of the dataset with good accuracy on modest hardware.

Figure 8

The same object voxelized in three resolutions: 163, 323 and 643. Higher resolutions preserve more geometric detail at the cost of more memory and processing time.



3.5 THE RESNET3D-18 ARCHITECTURE

With the data ready in volumetric format, we now need a neural network that operates on voxel cubes in the same way that a conventional CNN operates on images. The choice here is ResNet3D-18, a three-dimensional version of the classic ResNet by He et al. [He et al. 2016]. Before explaining the adaptation, it's worth a quick pause on what makes ResNet special.

Before ResNet, very deep networks were difficult to train — after a certain number of layers, the gradient that walks back during backpropagation became so small that the first layers practically stopped learning (the famous *vanishing gradient* problem). ResNet solved this with a simple idea: instead of each layer block calculating an output from scratch, it only calculates the *difference* (the waste) in relation to the input, and that input is added to the result via a *skip connection*. This shortcut creates a direct path to the gradient, allowing you to train networks with 18, 50, or even 152 layers without degradation.

ResNet-18 is the lightest variant in the family, with 18 layers and about 11 million parameters. It's the natural choice when we have a modestly sized dataset like ModelNet10 — larger networks would tend to overfit with no real gain in accuracy.

3.5.1 From 2D to 3D: Swapping the *Kernels*

The transition from a 2D CNN to a 3D one is conceptually straightforward: where there was a 2D operation (convolution, *pooling*, *batch normalization*), we exchange it for the 3D equivalent.

In code, it's literally swapping Conv2d for Conv3d, MaxPool2d for MaxPool3d, and so on. What changes is the shape of the *kernel*: instead of a 3×3-type 2D window that slides across the image, we have a 3×3×3 3D window that slides across the voxel cube, capturing geometric patterns in all directions simultaneously.

Table 8 summarizes the ResNet3D-18 architecture that we will use. At each *stage* (layer1 to layer4) the cube is halved in each dimension and the number of channels doubles — the same pyramidal pattern as in 2D ResNets. In the end, the *global average pooling* collapses cube 2×2×2 into a single 512 feature vector, which is then projected into 10 values (one probability for each class of ModelNet10).

Table 8

Layers of ResNet3D-18 adapted for ModelNet10. B is the lot size, and D = H = W = 32 is the voxel resolution

Camada	Operação	Saída
Entrada	Cubo de voxels	$(B, 1, 32, 32, 32)$
`stem`	Conv3d + BN + ReLU	$(B, 64, 16, 16, 16)$
`layer1`	2 blocos residuais	$(B, 64, 16, 16, 16)$
`layer2`	2 blocos residuais (*stride* 2)	$(B, 128, 8, 8, 8)$
`layer3`	2 blocos residuais (*stride* 2)	$(B, 256, 4, 4, 4)$
`layer4`	2 blocos residuais (*stride* 2)	$(B, 512, 2, 2, 2)$
`avgpool`	*Average pooling* global	$(B, 512)$
`fc`	Linear $\rightarrow$ 10 classes	$(B, 10)$

3.5.2 Adapting the *torchvision model*

PyTorch already offers a ready-made implementation of ResNet3D-18 through the torchvision.models.video.r3d_18 function. The name of the module (*video*) indicates the origin of the model: it was originally proposed for recognition of actions in videos, where the third dimension is time [Tran et al. 2018]. Despite this temporal origin, architecture is purely spatial — nothing in it is time-specific — and it works equally well when the third dimension is a spatial coordinate, as in our case. To use the model in ModelNet10 we need to make only two adjustments:

- Input layer (**stem[0]**): the original version expects 3 channels (R, G, B of the video). Our voxel grids only have 1 channel (busy/empty), so we swapped the first convolution to accept 1 input channel.
- End layer (**fc**): The original version classifies into 400 actions of the Kinetics dataset. We swapped it out for a Linear layer (512, 10) to produce the 10 classes of ModelNet10.

Code 3.8 shows this adaptation in *PyTorch*. Notice how minimalist it is: we instantiate the *torchvision* model without pre-trained weights (weights=None), replace the first convolution

preserving the other hyperparameters (*kernel*, *stride*, *padding*), and swap the final fully connected layer. Everything else in the architecture—the four *stages* of residual blocks, the *global average pooling*—is repurposed directly.

Table 9

Code 3.8. Adapted torchvision's ResNet3D-18 for 1-channel, 10-class input.

```
import torch.nn as nn import torchvision
def build_resnet3d18_classifier(num_classes, in_channels=1):
"""ResNet3D-18 adapted for 1-channel volumetric input."""  model =
torchvision.models.video.r3d_18(weights=None)
# 1. Replaces the first convolution: 3 channels (RGB) -> 1 channel (voxel)
original_conv = model.stem[0] model.stem[0] = nn. Conv3d(
in_channels=in_channels, out_channels=original_conv.out_channels,
kernel_size=original_conv.kernel_size, stride=original_conv.stride,
padding=original_conv.padding, bias=False,
)
#2. Replaces the final layer: 400 actions (Kinetics) -> num_classes
in_features = model.fc.in_features model.fc = nn. Linear(in_features,
num_classes) return model
model = build_resnet3d18_classifier(num_classes=10, in_channels=1)
```

It is worth noting that we are training the model from scratch, that is, without carrying pre-trained weights — unlike what was done with YOLO at the beginning of this chapter. For a dataset with the simplicity of ModelNet10 (only 10 balanced classes, with very different geometries), training from scratch achieves decent results in a few epochs, without the need for the prior knowledge that pre-training would bring.

3.6 3D CLASSIFIER TRAINING PIPELINE

With the pre-processing and architecture defined, it remains to integrate everything into a training pipeline. Unlike YOLO — which has a high-level API where you just point to a folder and call model.train() — here we use *PyTorch* directly, which gives you more control over each step of the process.

3.6.1 Dataset and DataLoader

PyTorch organizes training data around two abstractions: the Dataset, which knows how to load an individual sample, and the DataLoader, which groups samples into batches and delivers them to the training loop. For ModelNet10, the Dataset encapsulates the entire pipeline from the previous section: it takes the path of an .off file, reads the mesh, samples the points, normalizes the cloud, and voxels it, returning the voxel cube in the format (*1,R,R,R*) along with the entire class label.

3.6.2 *Data Augmentation* in 3D

Section 1.6 introduced *data augmentation* in the 2D context, with transformations such as rotations, *flips,* and brightness variations. In 3D, the idea is the same — to apply random geometric transformations so that the network doesn't memorize the exact poses of the workout — but the transformations themselves are different, and not all ideas from the 2D world make sense here.

The most important difference involves the vertical axis. The ModelNet10 models are all aligned upright: the Z-axis points upwards consistently across the entire dataset. This means that:

- Rotation on the Z axis is safe: Rotating a chair around the vertical axis produces another valid chair — it's equivalent to looking at the same object from another angle, simulating cameras positioned in different directions around the object.
- Rotation on the X or Y axes is risky: Turning a chair upside down produces a scene that the network will never see in the real world, and can confuse the model during training.

In addition to rotation, we added a bit of Gaussian noise to the coordinates of the points before voxelization, simulating small scanning imperfections and making the network more robust to noisy real-world data. These two transformations are applied before voxelization, still in the point cloud.

Like any form of *augmentation*, they are only applied to the training set, never to the test — we want to evaluate the model on "clean" data that reflects the actual expected performance. Code 3.9 shows the implementation of these two transformations.

3.6.3 Training

With data, model, and *augmentation* ready, the training follows the classic *PyTorch pattern*: each season, the model goes through all the batches of the training set, calculates the error between its predictions and the actual labels with the *cross-entropy*, and updates its

weights with the Adam optimizer. Each season we measure the accuracy in the test set and save the model if it beats the previous best result — this strategy ensures that the final evaluation is always done with the best weights, and not with those of the last season.

On modest hardware (Google Colab's GPU T4), the full ModelNet10 training at 323 resolution takes between 15 and 30 minutes for 10 epochs, achieving about 83% accuracy in the test set. The training curves in Figure 9 show this behavior: the *loss* drops consistently at each epoch, while the validation accuracy stabilizes around 82% around the 5th epoch.

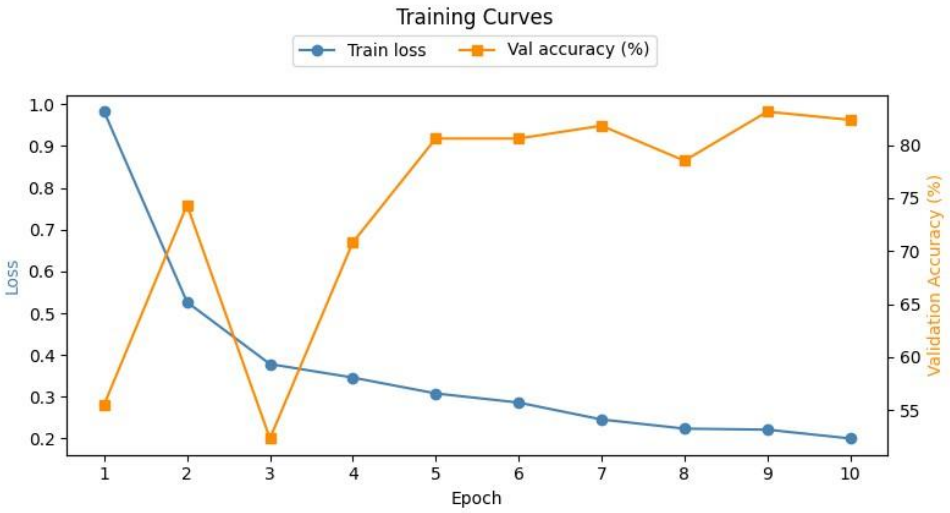
Table 10

Code 3.9. Data augmentation applied to the point cloud before voxelization: random rotation on the Z-axis and Gaussian noise

```
        def augment_point_cloud(points):
        """Random rotation on the Z-axis + Gaussian noise."""
        # Rotation around the vertical axis (Z) by a random angle theta =
random.random() * 2.0 * math.pi cos_t, sin_t = math.cos(theta), math.sin(theta)
rotation_matrix = np.array(
        [[cos_t, -sin_t, 0],
        [sin_t, cos_t, 0],
            [       0,          0, 1]],
             dtype=np.float32,
             ) rotated = (rotation_matrix@ points. T). T
             # Gaussian noise to simulate scan imperfections noise =
      np.random.normal(loc=0.0, scale=0.02, size=rotated.shape) return rotated
      + noise.astype(np.float32)
```

Figure 9

Training curves of ResNet3D-18 over ModelNet10 over 10 epochs. The training loss (blue) decreases monotonically while the validation accuracy (orange) stabilizes at around 82%



3.7 EVALUATION AND ANALYSIS OF RESULTS

Training a model is only half the work—the other half is understanding what it has learned. A single metric of overall accuracy can mask serious problems: a model that gets 83% right on the test set might be getting almost 100% right in some classes and less than 60% in others. In this section, we explore three tools that help diagnose this type of behavior.

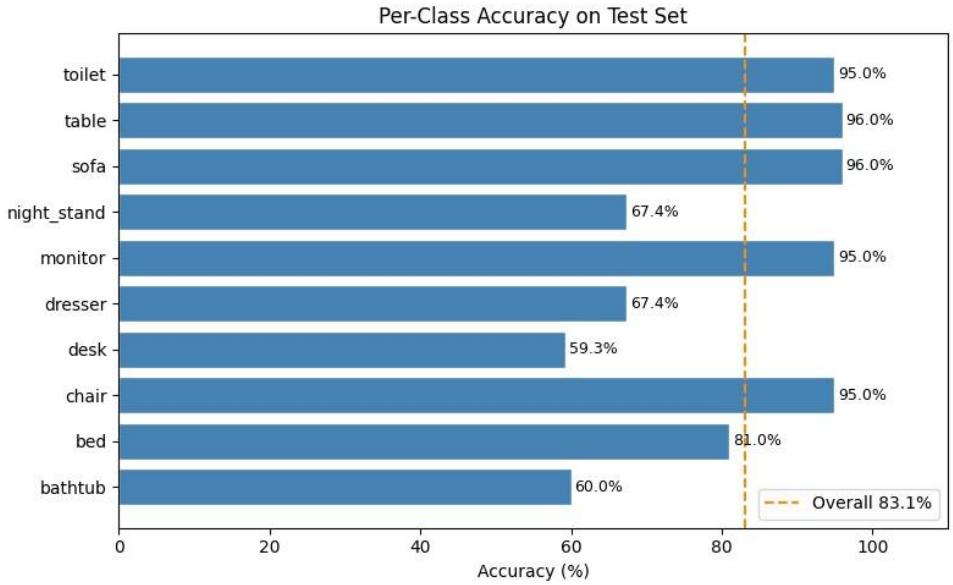
3.7.1 Overall and Class Accuracy

The overall accuracy of our 10-epoch-trained model is 83.1%. That's a good number of starts, but not enough on its own — especially since some classes have far more test samples than others, which means that errors in smaller classes weigh little in the overall average.

Accuracy by class reveals the full spectrum. Figure 10 shows that classes with very different volumetric geometries — *table*, *sofa*, *chair*, *monitor,* and *toilet* — reach an accuracy of around 95%, while *bathtub* (60%), *desk* (59.3%), *dresser,* and *night_stand* (67.4%) are well below the average and are the main responsible for pulling it down.

Figure 10

Accuracy per class in the ModelNet test set10. The dashed line indicates an overall accuracy of 83.1%. Classes with distinct volumetric geometry achieve accuracy close to 95%, while geometrically ambiguous classes are below average



3.7.2 Confusion Matrix

When a class has low accuracy, the next natural question is, "What is it being confused with?" The confusion matrix responds to this. It is a $C \times C$ *table* in which the entry in row *i* and column *j* indicates the proportion of samples of true class *i* that have been classified as *j*. The main diagonal contains the hits; everything outside the diagonal are mistakes.

Figure 11 shows the normalized matrix of our model. Four patterns of confusion stand out:

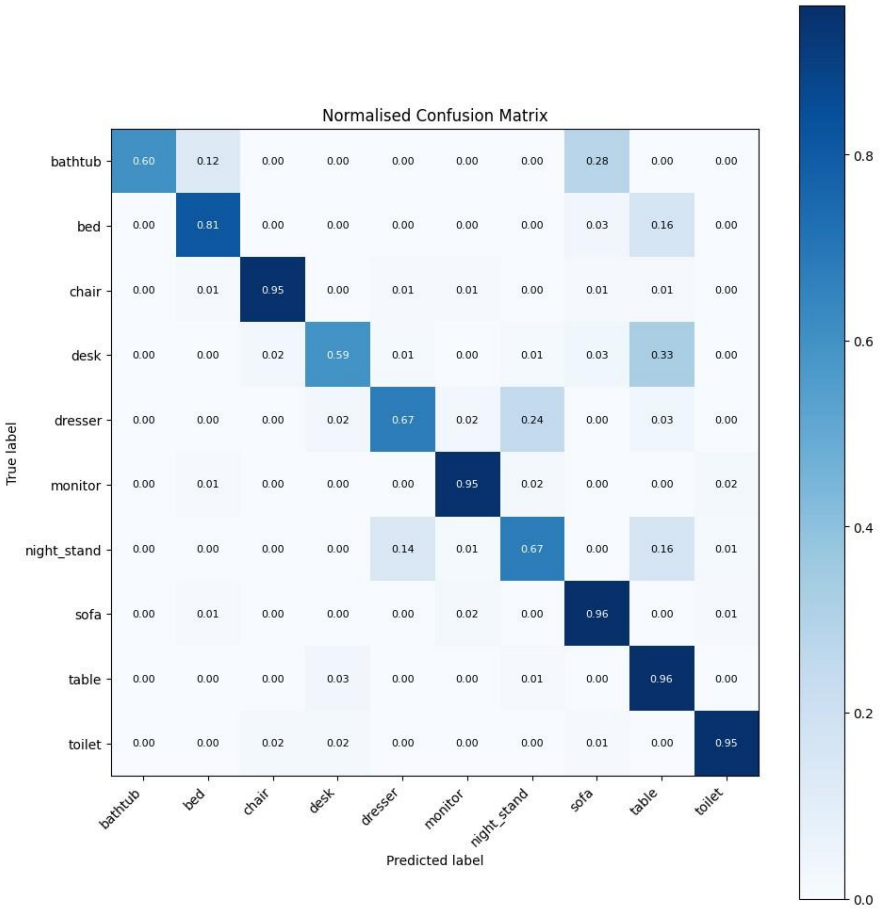
- *Desk* confused with *table* (33%): both are horizontal tops on legs, and the distinction between the two is more functional than geometric.
- *Bathtub* mistaken for *sofa* (28%): Seen as volumes with low walls and upper cavity, bathtubs and sofas have strikingly similar shapes at low resolution.
- *Dresser* confused with *night_stand* (24%), and the reverse (14%): both are vertical rectangular boxes, differing basically in size — and size is precisely the dimension we lose when normalizing each object to the unit cube.
- *Bed* confused with *table* (16%): beds voxelized in 323 essentially become a horizontal plane on supports, geometry close to that of a table.

The network is, in a sense, "right" in seeing geometric similarity—only that these classes were defined by human use, not pure form, and the model only sees form. This pattern also

suggests a concrete direction for improvement: preserving absolute-scale information (rather than normalizing everything to the unit cube) could help separate pairs like *dresser*/*night_stand*.

Figure 11

Normalized confusion matrix in the test set. Diagonal values indicate correct answers by class; Cells outside the diagonal reveal systematic confusions between geometrically similar classes



3.8 CONSIDERATIONS FOR 3D MODELS

In the previous sections we built a complete 3D classifier: we started from meshes in the OFF format, went through the sampling-normalization-voxelization pipeline, trained an adapted ResNet3D-18 and analyzed its performance in ModelNet10. The path taken is parallel to that of the 2D classification presented in the first half of this chapter, but with a fundamental difference: in 3D, the pre-processing is as important as the architecture of the model.

Voxel representation is didactic and fits naturally into conventional CNNs, but it carries important limitations. The cost of memory grows with the cube of resolution—doubling from 323 to 643 multiplies voxels eightfold—which makes it difficult to capture fine details on modest

hardware. Moreover, most of a voxel grid is empty space: the surface of a 3D object is essentially a 2D structure, and storing it in a dense grid represents an inefficient use of memory.

For the reader who wants to take the next steps, a natural direction is to experiment with ModelNet40, the expanded version with 40 classes. The pipeline we've built here—OFF reading, point sampling, PyTorch integration—continues to serve as the foundation and can be reused directly. What remains regardless of scale is the overall structure of the problem: raw data → representation appropriate to the architecture → training → critical evaluation. These steps are the skeleton of any 3D computer vision project.

4 CONCLUSION

Throughout this chapter you have been introduced to a solid foundation of *modern computer vision pipelines*, ranging from the fast applications of the YOLO ecosystem in 2D to the more rigorous pre-processing models required for 3D classifications. While modern architectures such as YOLO abstract much of the complexity of models to provide efficient inferences on real images and videos, Working with three-dimensional volumes imposes other architectural challenges, with a high computational cost of voxelization and the data losses inherent to spatial normalization.

While useful, these templates are not without error. A rigorous analysis, as presented here in the ModelNet10 confounding matrix, reveals the potential for the model to confuse objects for their pure geometric similarities (such as chairs and tables), and the same can happen with YOLO, misclassifying and detecting objects. This demonstrates that the critical evaluation of what the model has effectively learned is as crucial as its training. The implementations and concepts discussed here therefore serve as a practical guide to the implementation and use of these techniques. With these foundations, the reader will now be able to expand these solutions and apply them to new data sets and creating their own solutions for study or real problems - but they will not do without critical evaluation of their applications.

REFERENCES

Bhatt, D., Patel, C., Talsania, H., Patel, J., Vaghela, R., Pandya, S., Modi, K., & Ghayvat, H. (2021). CNN variants for computer vision: History, architecture, application, challenges and future scope. Electronics, 10(20). https://doi.org/10.3390/electronics10202470

Géron, A. (2019). Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems (2nd ed.). O'Reilly Media.

Hara, K., Kataoka, H., & Satoh, Y. (2018). Can spatiotemporal 3D CNNs retrace the history of 2D CNNs and ImageNet? In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 6546–6555). https://doi.org/10.1109/CVPR.2018.00685

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 770–778). https://doi.org/10.1109/CVPR.2016.90

Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., & Zitnick, C. L. (2014). Microsoft COCO: Common objects in context. In Computer Vision – ECCV 2014 (pp. 740–755). Springer. https://doi.org/10.1007/978-3-319-10602-1_48

Maturana, D., & Scherer, S. (2015). VoxNet: A 3D convolutional neural network for real-time object recognition. In 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (pp. 922–928). IEEE. https://doi.org/10.1109/IROS.2015.7353481

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 779–788). https://doi.org/10.1109/CVPR.2016.91

Tran, D., Wang, H., Torresani, L., Ray, J., LeCun, Y., & Paluri, M. (2018). A closer look at spatiotemporal convolutions for action recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 6450–6459). https://doi.org/10.1109/CVPR.2018.00675

Ultralytics YOLO Documentation

Wu, Z., Song, S., Khosla, A., Yu, F., Zhang, L., Tang, X., & Xiao, J. (2015). 3D ShapeNets: A deep representation for volumetric shapes. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 1912–1920). https://doi.org/10.1109/CVPR.2015.7298801